# COMPUTER STRUCTURES FOR DISTRIBUTED SYSTEMS

LIAM MAURICE CASEY

DOCTOR OF PHILOSOPHY
UNIVERSITY OF EDINBURGH
1977

ABSTRACT: A building block approach to configuring large computer systems is attractive because the blocks, either primitive processors or small computers, are daily becoming cheaper and because this approach allows a close match of the power required to the power supplied. This thesis addresses the design goal of an expandable system where there is no premium paid for a minimal configuration and the cost of extra units of capacity is constant. It is shown that a distributed system, a system of homogeneous computers loosely coupled by a communication subsystem, is likely to be the best approach to this design goal. Some consideration is given to the form of the communication subsystem but the main research is directed towards the software organisation required to achieve efficient co-operation between the computers constituting the distributed system. An organisation based on the domain structures of protection schemes is found to have advantages. Hitherto domain management using capabilities has been centred around systems with shared primary memory. This is because central tables have been required to implement the capability mechanism. A model is developed which, by restricting the sharing of some items and providing a 'global object' management scheme to cover essential sharing, enables central tables to be dispensed with and domain management to be distributed. The main goal in achieving this extension is to facilitate dynamic and efficient load sharing but the model could equally well be used to provide, in distributed systems, the protection normally associated with domains. This thesis also considers the wider ramifications of distributed systems. A simulation program is described and results from it are analysed to give some insights into factors affecting distributed system stability and performance. It is concluded that the above design goal of linear expandibility can be attained for a moderate range of systems sizes (perhaps from 1 to 20 computers).

Keywords and Phrases: distributed computer system, multiple computer system, load sharing, homogeneous, domain, capability, simulation.

ACKNOWLEDGEMENTS:

Acknowledgement is due to the New Zealand Department of Scientific and Industrial Research whose generous financial support made this work possible. I wish to thank the supervisors of my study, Professor Sidney Michaelson and Nick Shelness. Also a word of thanks is due to Lee Smith who has assisted with the proof reading of this thesis. Finally I would like to thank my wife Hilary who has borne her 'Ph.D. widowhood' well.

This thesis is dedicated to my son Martin. If it were not for his habit of waking early this thesis would not be completed yet.

DECLARATION:

I hereby declare that this thesis has been composed by myself and that the work reported is my own.

18 January 1977

CONTENTS


| | | |
|---|---|---|
| Chapter 1: | PROLOGUE | 1 |
| Chapter 2: | MERITS OF VARIOUS HARDWARE ORGANISATIONS | 12 |
| Chapter 3: | COMMUNICATIONS | 41 |
| Chapter 4: | OPERATING SYSTEMS ARCHITECTURE | 62 |
| Chapter 5: | THE DEVELOPMENT OF THE DOMAIN CONCEPT | 81 |
| Chapter 6: | OUR MODEL | 106 |
| Chapter 7: | DISTRIBUTED SYSTEM METHODOLOGY | 141 |
| Chapter 8: | DOMAIN MANAGEMENT | 176 |
| Chapter 9: | DESCRIPTION OF SIMULATION | 198 |
| Chapter 10: | RESULTS OF OUR SIMULATION | 230 |
| Chapter 11: | EPILOGUE | 256 |
| Appendix A | | A-1 |
| Bibliography | | B-1 |

# CHAPTER 1

# PROLOGUE

All our yesterdays:

In 1954, a decade after the first digital computer was built, workers at the National Bureau of Standards, USA, connected together two computers, SEAC and DYSEAC, forming the first multiple computer system. The resulting system was capable, so they claimed, of handling efficiently problems which the two component computers could scarcely have handled if each were working alone [CODD62]. This led them to produce the first published proposal for the construction of a multiple computer system [LEIN58,CURT63]. The proposed system, PILOT, consisted of three computers: a primary computer, a clerical or secondary computer and an I/O computer. To quote: 'These computers intercommunicate in a way that permits all three to work together concurrently on a common problem' and 'The system can be .... used in conjunction with other digital computer facilities forming an interconnected communication network in which all the machines can work together collaboratively on large scale problems that are beyond the reach of any single machine'.

Despite the confident use of the present tense above PILOT did not achieve its design goals. It was decommissioned in the mid sixties, its construction (started in 1958) never fully completed although the hardware had been working well enough for 'continuing difficulties in using primary and secondary (computers) together, particularly in program debugging' to be experienced [PYKE74].

This thesis addresses some of the problems involved in getting computers to work together.

Before PILOT the sole approach to achieving more computing power than that provided by a single machine was to build a faster machine. In a 1953 paper Grosch wrote: 'I believe that there is a fundamental rule, which I modestly call Grosch's law giving economy only as the square root of the increase of speed - that is to do a calculation ten times as cheaply you must do it one hundred times as fast' [GROS53] and Grosch's law, re-cast in the positive form as 'the power of a computer is proportional to the square of its cost' has in no small way encouraged this approach against that of trying to form multiple computer systems.

Grosch's law did not go unchallenged [ADAM62] but some ten years later it was given an impressive validation in the study of 225 American computers by Knight [KNIG66]. In a debate on the architecture for large computer

systems in 1967 Amdahl, quoting Knight but conveniently ignoring a proviso he made about large systems in his work, exhorted everyone to 'Keep the faith, baby' in the single processor approach [AMDA67]. Amdahl has kept his faith to this day as has Grosch [GROS76]. Some of the points we raise later (in chapter 2) suggest that there is considerable justification for their steadfastness.

What tomorrow may bring:

Nevertheless since PILOT there has been an increasing number of multiple processor architectures proposed and built. These architectures are justified as circumventing the current technological limits on the power of single processor systems, providing facilities to remote users (when the constituent processors are situated at geographically different sites) [BERN73,BLAN73,CRAI74], or providing more cost effective computing than single processors of equivalent power. This thesis is chiefly concerned with multiple computer architectures that may provide cost effective computing.

The imminent prospect of cheap but primitive microprocessors and 'free' memory [WITH75] has led to an explosion in the size of proposed systems; systems '(of) over one hundred active processors' [GOOD73], 'having not tens or hundreds of processing elements but many thousands' [WIRC75] and '(forming) a network of thousands

or millions of microcomputers .... a range of network sizes from 100 to 1,000,000,000 computers' [WITT76].

In chapter 2 it is shown that queueing theory mitigates heavily against large numbers of low powered processors providing a service equal to that of a single processor with 'equivalent' power. Other chapters describe some of the mechanisms required for running programs on systems with modest numbers of identical computers, the overhead these mechanisms produce in each computer is shown, at best, to be proportional to the number of computers in the system. Thus the day of the million co-operating computers is never likely to arrive.

Perhaps the most telling criticism that can be levelled both at PILOT and these later extravagant architectures is that the designers have concentrated only on the hardware requirements and given no thought to the software required to achieve co-operation among the processors. Anyone attempting to implement one of these over-blown systems would also experience 'continuing difficulties' in achieving co-operation between processors. The mechanisms for co-operation have to be formulated prior to detailed hardware design. The main research reported in this thesis has been on the software structures required to enable separate computers to collaborate to form a single operational entity, a distributed system.

Cost effectiveness:

This thesis describes a system that could consist from 1 to perhaps 20 identical computers. We feel that such a system may prove cost effective. For a given cost the system might provide more power than a single computer system or alternatively a given power might be provided by the multiple computer system more cheaply than by a single computer.

We have used the word 'power' several times now without giving a definition, no satisfactory definition exists [FULL76]. The concept is meant to express the overall speed of a computer, how much work it can perform in unit time. Likewise satisfactory definitions of cost are impossible to formulate. So we will not add another deficient metric of cost effectiveness to the large number already in existence. Instead, we instance below recent examples of computer use that indicate that today Grosch's law is not valid and indeed may have been only a self-fufilling prophesy used by computer manufacturers to price their products.

In 1973 Heart and others studied possible replacements for the IMP machines in the ARPA network [HEAR73,ORNS75]. The IMP machines perform a single function, namely the control of packet switching in the ARPA network. The amount of computing power this function requires varies depending on where the IMP is situated in the network.

Heart and his co-workers, after performing simulations, concluded that systems constructed from upto 14 simple minicomputers would be cheaper than using a single faster machine. Using a multiple computer system they could also vary the number of computers in each IMP system to match the intended load, thus providing even greater savings.

Schaeffer [SCHA75] has reported on a costing exercise that resulted in a chemistry department shifting its computing load away from a centrally run large machine to a 24 bit word minicomputer. The department's computing allowance bought them 32 hours of CPU time a year on a CDC 7600. The rate structure of the CDC 7600 reflected simply the cost of operation of the machine, its purchase price having been paid by an outside agency. Schaeffer found that the same annual budget would, over 4 years, pay the purchase price and running costs of a 16K word minicomputer. The minicomputer was purchased and the department's programs were found to run, on average, 35 times slower on the minicomputer than on the CDC 7600. Twenty hours a day operation of the minicomputer was achieved so that the department's annual budget purchased, in effect, 200 hours of CDC 7600 time instead of 32.

Fuller [FULL76] has attempted a detailed comparison of the price/performance ratio of a PDP 10 and C.mmp, a system of up to 16 minicomputers [WULF72]. He

encountered problems in defining performance and cost. He used two physical characteristics as measures of performance: instructions executed per second and processor memory bandwidth. The former is biased towards primitive machines that do little work with each instruction, while the latter is biased towards large machines which may in each memory cycle be fetching more data than they use. Therefore Fuller claimed, the two measures provided bounds for performance estimates and he calculated a factor of 4 in cost effectiveness of C.mmp against the most cost effective PDP 10 configuration.

There have also been reports of commercial applications being mounted on systems of minicomputers at considerable savings over using single higher powered computers. A hospital in the USA has an operational system of 10 Data General Novas to perform all its data processing [CARR75]. Jagerstrom has described plans for a company to computerize by putting each application on a separate minicomputer [JAGE74]. He claims that the end system will be cheaper than if a single computer was used, with the added advantages that the computer power for each application need be acquired by the company only when it is ready to mount the application and, as in the case of the hospital above, some processing can proceed when one of the minicomputers has failed.

It is not difficult to give reasons for the increasing hegemony of small computers. Large computers are

characterized by low volumes of production and significant manufacturer commitment to software. Successful small computers sell in much larger quantities and their software support is lower, sometimes non-existent. Software production is a fixed overhead independent of the volume of sales. Expected sales volume dictates the fraction of this and other overheads, such as design cost and tooling up cost, which will be included in the individual selling price. Volume of production also affects the construction cost of each unit, greater volumes mean that more automated methods of production will be cost effective. The low volume of sales of large systems means that the same technology has to be retained over a long period to recoup the original investment. But older technology is more expensive per se, and also in assembly costs because of the proportionally higher component counts [BLAK75]. Of course the cheaper computers are, the more will be sold. Overall there is a cascade of effects making small computers cost effective for more and more applications.

The big question is whether or not small computers can be tied together to make more powerful systems that still retain their cost effectiveness. Does the overhead produced in amalgamating small machines into a larger one swamp the cheapness of the small machines?

A building block approach [DAVI72], where identical computers are added to a system until the required power

is achieved would be beneficial both to manufacturers and users. Manufacturers are often required to produce a range of computers of identical architecture but differing power. Each type of computer requires designing afresh and may be implemented using different semiconductor technology from the other types, thus negating many of the benefits of large production volumes. Using a number of low power computers to fabricate the computers at the high power end of the range means that only one design is needed and this will be produced in extra large volumes. The user, unless he buys a computer for a single static task, has to be aware of the cost of obtaining an increase in capacity as his requirements increase. This usually leads him to purchase a system with capacity in excess of his immediate needs and may later involve him in having to dispose of some hardware and buy a more powerful system if his requirements grow too large. Buying a system exactly matched to his needs and expanding it (or contracting it) when those needs change, by altering the number of building blocks, can provide obvious economies for the user.

Both manufacturers and users would be looking for systems with low initial cost and linear expansion costs. If, because of the requirement of being expandable, a small system costs a lot more than an equivalent non-expandable system then it will be difficult to sell the expandable system. Likewise a system where added

building blocks become less and less cost effective because overall performance diminishes as extra building blocks are added, would be limited in usefulness. What is required is a fixed cost/performance ratio. The marginal increase in power with the addition of an extra computer should be constant (or nearly so) no matter how many computers there are already in the system. There is often an expectation of general synergism in multiple computer systems, that is the total power in the system is expected to be somehow greater than the sum of the individual computers' powers. While there can be limited synergistic effects as a system expands, overall the total power available is just that provided by the constituent computers. It is impossible to provide indefinitely a diminishing cost/performance ratio as the number of computers in a system grows.

In the next chapter we look at the two basic ways of amalgamating computers: multiprocessor systems, where primary memory is shared between all processors, and distributed systems, where computers are kept separate but interact with one another using some form of communication system. We describe the drawbacks of both multiprocessor architectures and single processor architectures compared to distributed systems and the rest of the thesis concentrates on distributed systems.

Chapter 3 examines the various forms the communication system can take and chapter 4 looks at operating systems

structures suitable for distributed systems. One of these operating system structures, the kernel/domain architecture is further described in chaoter 5. The rest of the thesis then details facets of the design of a distributed system using the kernel/domain structure and describes a simulation program that was used to study some performance questions that arise concerning the design.

# CHAPTER 2

## MERITS OF VARIOUS HARDWARE ORGANIZATIONS

A useful taxonomy of computer architectures has been defined by Flynn [FLYN72]. He divides systems into 3 types:

- SISD: (single instruction acting on single item of data) the conventional uniprocessor system.
- SIMD: (single instruction acting on multiple data items) systems with vector hardware, associative and parallel processors.
- MIMD: (multiple instructions acting on multiple data items) multiprocessor systems and computer networks.

(For completeness there is also the MISD type, which others have taken to denote instruction pipelining machines [HIGB73,THUR75]).

Systems of the SIMD type have been the chief candidates for solving large scale problems beyond the limit of conventional machines. They have always been 'one-off' and economics would seem to be a secondary consideration in their construction. It is now generally conceded that there are some special problems, weather forecasting being the most often quoted example, for which these architectures are the most appropriate but

'these are special-purpose machines and any attempt to apply them to an incorrectly sized, or designed, problem is an exercise in futility' [THUR75].

There is a spectrum of MIMD systems, ranging from tightly coupled multiprocessors systems to trans-world networks. The system that is described in this thesis is a network that lies towards the multiprocessor end of the MIMD spectrum. It consists of a number of homogeneous (that is highly compatible, if not identical) sites connected by a communications subsystem. The whole system is envisaged to be local in extent, fitting into a cabinet, a room or, at most, a building. Each site is assumed to consist of a single processor with its own memory space. The term 'distributed system' has been arbitrarily appropriated to denote this. There is no logical reason why the sites in a distributed system should not each consist of multiprocessor systems, but to avoid confusion we do not consider such a case here.

To place distributed systems in context we examine the benefits and drawbacks of MIMD systems compared to SISD systems, particularly in relation to time sharing.

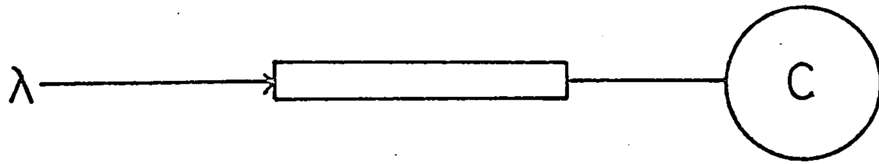
SECTION 1: QUEUEING THEORY CONSIDERATIONS

In order to gain mathematical tractability, queue-theoretic models are always idealized abstractions that omit many of the details of reality. The results of queueing analysis nevertheless often indicate fundamental constraints that cannot be breached by any strategy.

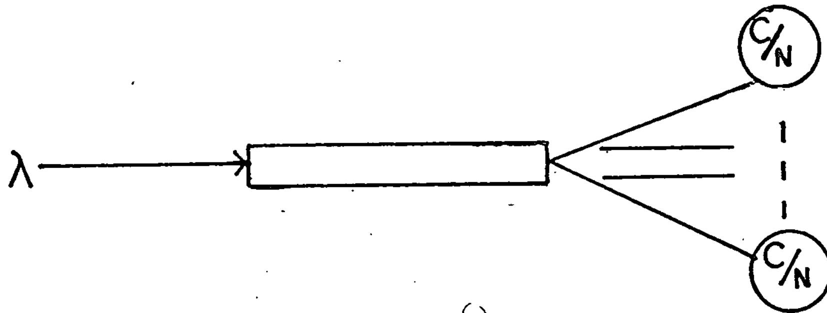
Organizations:

In a queueing theory approach to hardware organization the differences between architectures are represented by replicating servers and by having different queueing mechanisms. Figures 2.1 to 2.7 give the representation of various systems each having a total service capacity of C operations per second and each having an overall arrival rate of jobs, requests for service, of $\lambda$ requests per second. We assume that the mean number of operations requested by each job is $1/\mu$. To ensure that the systems have the capacity to ultimately deal with all jobs arriving the further assumption is made that $\lambda/\mu C < 1$. The ratio $\lambda/\mu C$ is called $\rho$, the utilization, as it gives the ratio of the mean number of operations requested of the system per second $\lambda/\mu$ to the number of operations the system can perform per second, C.

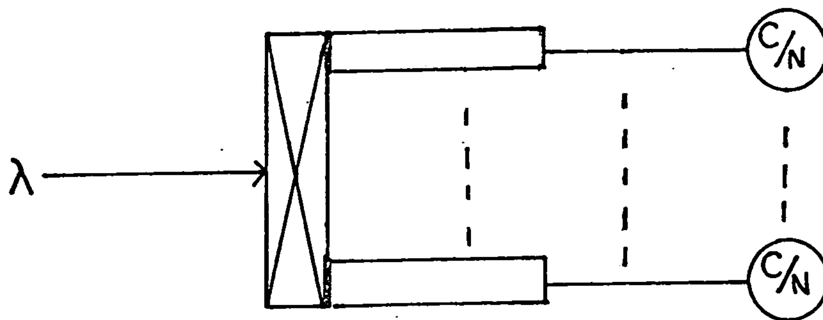
The SISD architecture, the single processor system is represented by figure 2.1. This is the classical single server queueing system, the backbone of queueing theory.




Single processor
Figure 2.1




Multi processor
Figure 2.2




Distributed system with instantaneous jockeying
Figure 2.3

Analytic expressions for the mean response time, T, that is the average elapsed time between job arrival and job completion, have been found for large classes of probability distributions of the arrival rate and service times of jobs, and for a number of queueing disciplines such as first come first served, round robin and so on (see for example KLEI75,KLEI76). The simplest case is for first come first served systems where both the inter-arrival times between jobs and the size of jobs have (negative) exponential distributions. The mean response time is given by

$$T = 1/(\mu C-\lambda)$$

The tightly coupled multiprocessor system, with N processors is represented by figure 2.2. In the multiprocessor system it is assumed that service of jobs is from a common core queue. Analytic results are known for T only when the service time is exponentially distributed [KLEI74]. For the case of N=2 with exponential arrival times the result is

$$T = 2\mu C/((\mu C+\lambda)(\mu C-\lambda))$$

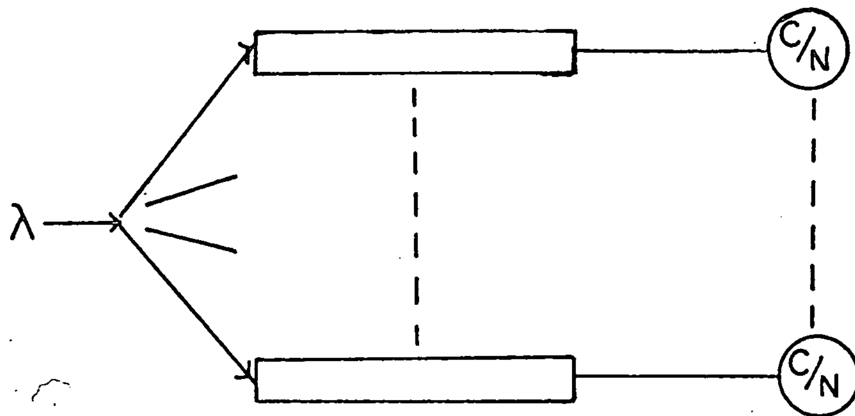
Figure 2.8 gives graphs (adapted from [KLEI74]) for the normalized response time (when $\mu C=1$) for N=1, the single processor case, N=2 and N=10, assuming exponential arrival and service times. There is an approximate solution, Kingman's conjecture, for T for general distributions of arrival times and service times, for when $\rho$ is close to 1, given by

$$T = N\rho/\lambda+(C_a^2+\rho^2C_p^2)/2\lambda(1-\rho)$$

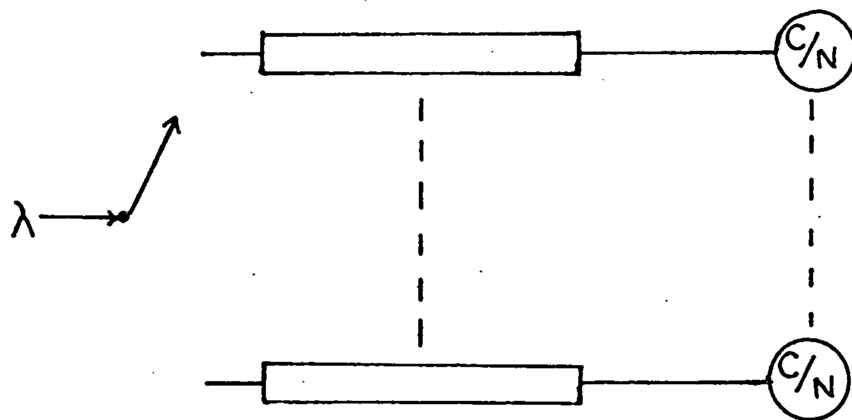
where $C_a^2$ is the squared coefficient of variation of the inter-arrival times and $C_p^2$ is the squared coefficient of variation for the number of operations required by a job [KLEI74].

Figure 2.3 shows a queueing system that has separate queues for each server but is subject to 'instantaneous jockeying'. The last entry in a queue moves instantaneously to another queue if that queue becomes shorter than the queue it is in. This represents the ideal, physically unattainable, for load levelling distributed systems. Because such systems do not share core they do not have a common queue of jobs, but if the distributed system wants to keep the load on all processors the same, then jobs will be moved around to try to attain this. In the real world moving jobs will take some time, during which the load situation could change again. Instantaneous jockeying means that no processor is idle when another server has jobs waiting for service. It has been shown that because the idle time of servers is the same as in the common queue system above that the mean service time will be the same as well [LEEA66].

We make a distinction between load levelling, where jobs are moved about from queue to queue, and load balancing where the system attempts to even out the load on each server solely by directing incoming jobs to the queue of the server that is most likely to be able to




Distributed system: go to shortest queue
Figure 2.4




Distributed system: order of arrival
Figure 2.5

serve them first. Figure 2.4 depicts the system where incoming jobs are allocated to the processor with the shortest queue, and the job remains in that queue. Obviously in this situation it is possible for one server to be idle while another server has jobs waiting, hence the utilization of servers will be lower than in the common queue or instantaneous jockeying system, and the average response time will be higher. Definite formulae for T have not been derived.

Figure 2.5 represents the situation where arriving jobs are allocated to each server in turn, irrespective of load. We would expect this to give worse response times than the case above where jobs go to the processor with the shortest queue. The arrival rate of jobs at each server in this case is $\lambda/N$ and the squared coefficient of variation of the arrival times is $C_a^2/N$. In the case of exponential arrival times the effective arrival time distribution for each server is N stage Erlangian, a situation that has been solved analytically when there is exponential service times. More generally if we consider Kingman's approximation we have

$$T = N\rho/\lambda + (C_a^2 + N\rho^2 C_p^2)/2\lambda(1-\rho)$$

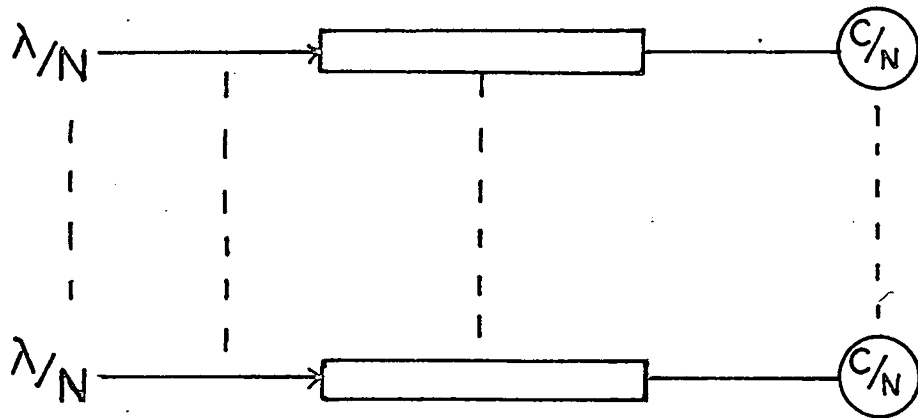
Thus the increase in response time over the common queue system is confined to the term $N\rho^2 C_p^2$ and so depends on the number of servers and the coefficient of variation of the number of operations required for each job. If each job requires exactly the same number of operations (i.e. $C_p^2=0$) then there would be no increase in response time

using this allocation of jobs to each server in turn instead of using a common queue. Unfortunately in computer systems the coefficient of variation of service times is likely to be large.

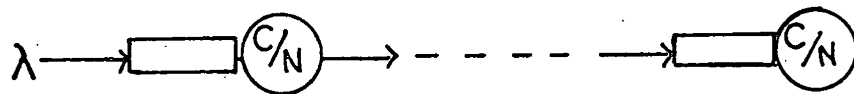
Figure 2.6 represents the extreme situation of no coupling at all between systems. The population of jobs is divided into N categories a priori so that the arrival rate at each server is $\lambda/N$ and the squared coefficient of variation remains $C_a^2$. This type of system can arise when the users are divided into N equal groups and each group is permanently assigned to one computer. It also arises when functionally specialized computers are used so that each server can only handle one type of job. We assume here that there are N types of job and that the overall average number of each type of job is the same. In this case the response time is exactly N times what it would be for the single server with capacity C because each server is an independent server with capacity C/N. For the case of exponential arrival and service times the average response time is given by

$$T = N/(\mu C - \lambda)$$

If the average fraction of jobs going to each server is not identical for all servers then the same mean response time (but not the same variance) will be obtained if the capacity of each server is adjusted to be in proportion to the average number of requests received by that server (keeping the overall capacity equal to C).




Separate systems
Figure 2.6




Pipeline
Figure 2.7

Figure 2.7 shows a pipeline or N stage tandem system. Here we assume that each job requires an average service of $1/N\mu$ operations from each server in turn. In the case of exponential arrival and service times Burke's theorem [KLEI75] states that the mean response time is given by

$$T = N/(\mu C-\lambda)$$

which is the same as the completely decoupled system above.

Implications:

Our excursion into queueing theory results has shown that the various ways of configuring systems to give a capacity of C operations per second do not all give the same response times. Figure 2.8 shows the deterioration in response time (normalizing $\mu C$ to unity) as the capacity C is divided among 1,2 and 10 servers, the 2 and 10 server systems either taking jobs from a common queue or having instantaneous jockeying. These response curves were drawn under the assumption of negative exponential arrival and service times, but similar curves could be drawn using the Kingman approximation. They unequivocally show that unless the utilization $\rho$ is very close to 1, when response times are very long anyway, having a single server gives better response times than dividing up the capacity among N servers. For batch processing systems it is possible to attain a processor utilization close to 1. To attain reasonable response times for time shared systems an operational range of

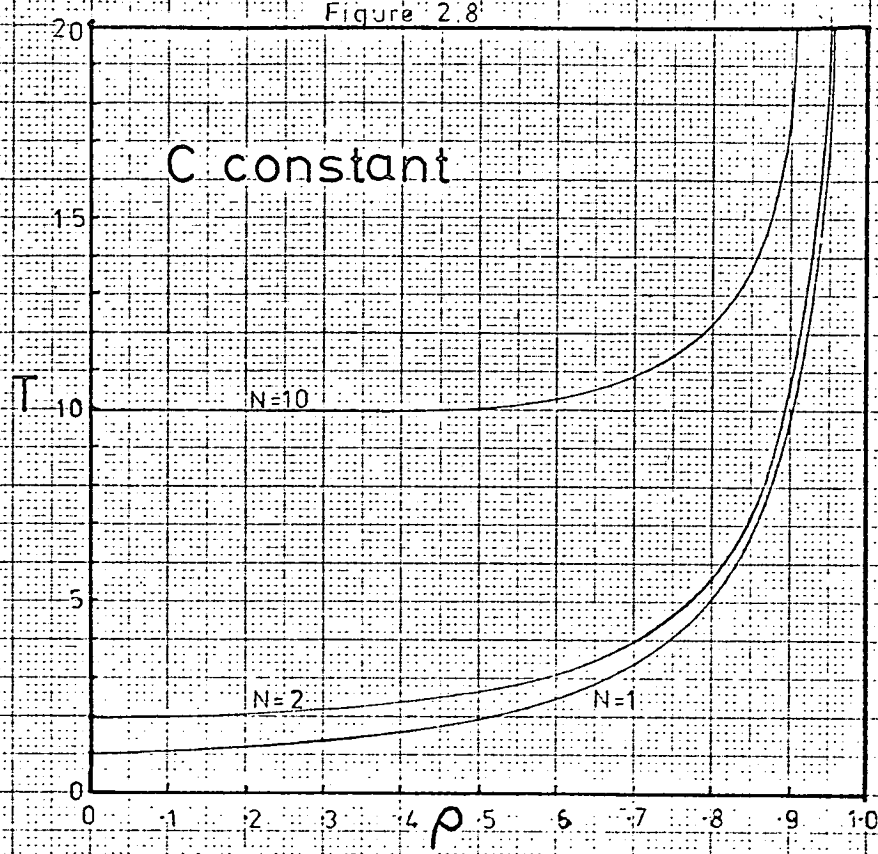



Figure 2.8

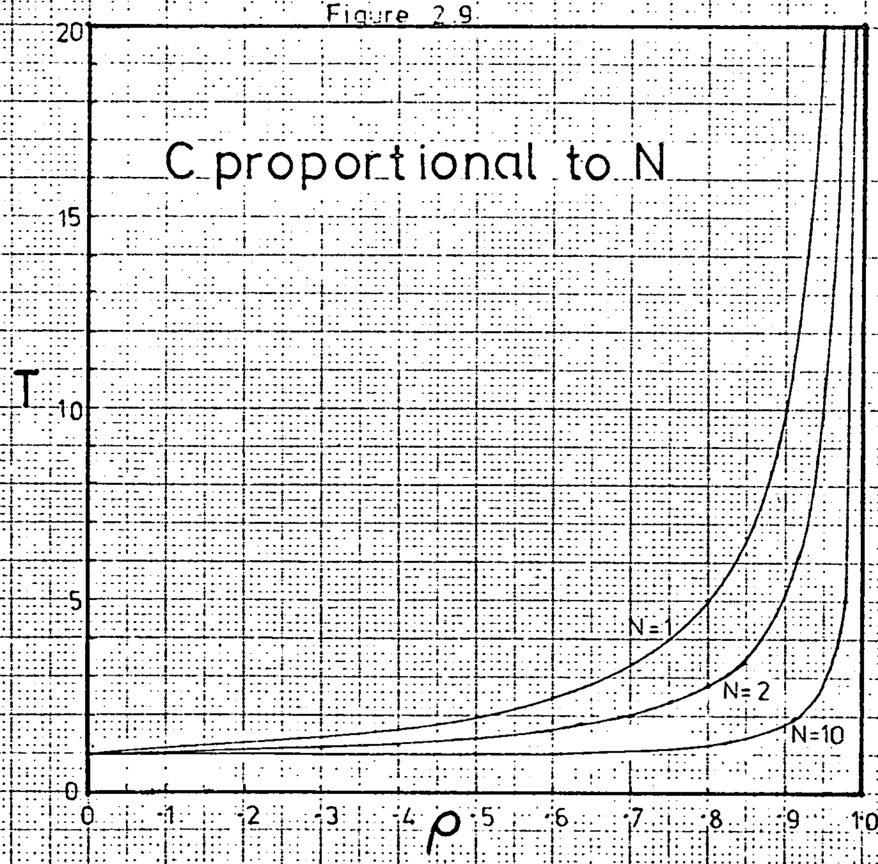



Figure 2.9

utilizations is likely to be in the region of $0.6<\rho<0.9$ [BELL70]. For such an operational range replacing a single large processor by a number of microprocessors, say, of the same total power (ignoring overheads) is going to result in worse response times.

If other considerations lead to the adoption of multiple servers then the results presented above indicate that an effort should be made to maximize the utilization of servers. Systems where no server can be idle while there are jobs waiting for service, the common queue and instantaneous jockeying systems above, have a better mean response time than systems where there is a possibility of servers being idle while there is outstanding work. Specialization of servers so that each can only serve a subset of jobs, or so that every one of them has to be involved in the service of all jobs, gives the worst response time. Thus the above analysis indicates that there are increasing gains to be made by

1) accepting any job at any server (processor)
2) attempting to load balance by directing incoming jobs to the shortest queue
3) attempting to load level by shifting jobs from the ends of queues to shorter queues.

As we stated in chapter 1, the expansion characteristics of a system are important. Figure 2.9 depicts the normalized response time for three systems, each with exponential arrival and service times. The N=1

system has a single server of capacity C and an arrival rate of requests $\lambda$. The N=2 and N=10 systems have 2 and 10 servers respectively, each server having a capacity C, and the arrival rates at these systems are assumed to be $2\lambda$ and $10\lambda$ respectively. (Again equivalent curves could be drawn using the Kingman approximation). Figure 2.9 shows a pleasing feature of expansion of the number of servers while keeping the load per server constant; in the time sharing operational range mentioned above there is a decrease in response time. The minimum possible mean response time is simply the mean service time and this is attained, for all values of $\rho<1$, when there is an infinite number of servers. So the decrease in response time, as another server is added, tends to zero as the number of servers becomes large.

Queueing networks and bottlenecks:

A closed network queueing system consists of a finite number of jobs (customers) that cycle around queueing for service at a number of nodes. After a job has received service at a node it moves to another node to queue there for service. Closed network queueing systems can model the behaviour of time sharing systems better than the models we discussed above. The fixed number of jobs represents the restrictions in time shared systems on the total number of concurrent users. Resources other than the central processor, such as disks, from which there is

a requirement for service can be represented as nodes in the network system.

After Moore [MOOR71] analysed the MTS time sharing system using a closed network queueing model, a rash of papers appeared applying closed network queueing models to the study of time sharing systems. These efforts are surveyed by Kleinrock [KLEI76]. We will not discuss them further except to examine the concept of the 'bottleneck'.

When a system has more than one type of resource in demand, then as the load on the system is increased (by increasing the number of jobs in the system), the utilization of the resources will increase. Eventually the utilization of some resource will get very close to 100% so that the utilization cannot increase significantly as the load increases. At this stage a long queue containing almost all of the jobs in the system will build up waiting to use the resource. This resource is a bottleneck and the overall response time of the system becomes completely dominated by the response time of the bottleneck. (The response time analyses we gave above are valid therefore when processing power is the bottleneck in a system). A system where the utilization of all resources approach 100% together is called a balanced system.

Memory requirements:

Recently Borgerson [BORG76] has examined another facet of replacing a single processor with N slower processors to give the equivalent capacity. He considered a single processor system that achieved adequate processor utilization when it had enough primary memory to sustain a multiprogramming level of K (that is K jobs, or working set, could reside in the primary memory at once). By a very simplistic analysis he determined that the 'equivalent' multiprocessor system (with N processors) would require enough primary memory to contain N+K-1 jobs to achieve the same processor utilization. Certainly N processors cannot all be gainfully employed processing K jobs if K is less than N. The longer response times of N processor systems translate into longer job residency in primary memory.

Adequacy of queueing theory models:

Queueing theory does give some very useful insights into how various systems will behave. But there are restrictions placed on service times, queueing and service disciplines, and particularly interactions between different resources in the system (e.g. queueing theory cannot model the constraint that both primary memory space and a processor have to be available before a job can be executed). In consequence queue theoretic

approaches cannot be used for detailed analysis of systems. Perhaps the last word should go to Kleinrock, whom we have used as a source for many of the results quoted in this section.

> 'The mathematical structures ... created in attempting to describe real situations are merely idealized fictions, and one must not become enamoured with them for their own sake if one is really interested in practical answers' [KLEI76].

## SECTION 2: PHYSICAL AND COST CONSIDERATIONS.

There are of course many factors besides queueing theory predicted performance to be taken into account in considering an architecture. Cost effectiveness is the paramount factor. We now look at a number of factors that affect the cost or performance of various architectures.

### Overheads:

The computation required to manage a list or queue grows at a faster than linear rate as the size of the list or queue grows [HANS73]. Thus the overheads in managing a system with a large number of users are

proportionally much greater than for a system with a small number of users because the former will have longer queues. A multiprocessor system and a single processor system of equivalent power will have approximately the same management overheads (but there will be some added complexity in dealing with multiple processors). However in a distributed system some of the lists and queues are partitioned amongst the sites so that there is a reduction in the overheads of managing them.

Some part, perhaps all, of an operating system must be resident in the primary memory of a computer at all times, using up memory space that would otherwise be available to user programs. In a system with multiple servers which are not completely independent, extra operating system software is required to achieve the necessary co-operation among the servers [BORG76]. However in a multiprocessor system only one copy of an operating system is shared among all the processors. This impacts favourably on the expansion characteristics of a multiprocessor system because added memory can be almost entirely dedicated to user programs. In all multiple computer systems that we know of that do not have shared memory, apart from the system we develop in this thesis, each computer has its own complete, or nearly complete, operating system. One of our chief aims has been to make as much software as possible shared among all the sites in our distributed system so that increasing the size of system means that proportionally

more primary memory space is available for useful work.

Parallelism:

If in the queueing theory analysis above each and every job presented to a multiserver system could be split into exactly N parallel phases of equal duration, one phase for each server, then the response times of the multiserver system would be equal to that of the equivalent capacity single server. However, apart from such operations as overlapping I/O with processing, parallelism in general purpose computing is difficult to find, both at the macro level and the micro level [TJAD70]. Examples of programs decomposed into parallel modules [THOM72,FULL76] seem to us to be rather contrived. We do not think that parallelism can be relied upon as a factor to bring the performance of multiple processor systems up to that of single processor systems.

Functional specialization:

Many designs for distributed systems and multiprocessor systems utilize functionally specialized processors [JUSE74,COLO76,ARDE75,FABR74,REYL74,SELI72]. Computer networks of large machines, usually at separate locations, are often justified by the differing

characteristics, hardware or software, of each computer, or host, in the network [RUBE70,GHEZ73,COLE73].

In the case of networks joining together already existing machines, functional specialization does offer potential for increased throughput and perhaps reduced response time, compared to the original situation of not having a network at all. This is because if each host is offered highly conformable work it can process it faster than if it has to process all types of job. Forms of close co-operation, such as load levelling or balancing, although often cited as design goals for networks [ROBE70, HOWE72] have yet to be realized. Basically the overheads in achieving closer co-operation [HICK71, SMIT72,FRED73] outweigh the benefits. We feel functional specialization will continue to be the raison d'etre of geographically dispersed networks of large computers.

In the case of distributed systems and multiprocessors the gain in effective capacity through functional specialization has to be very large to offset the queueing theory gains in response time that can be achieved by making all processors capable of executing all jobs. Functional specialization often gives rise to very simple forms of operating systems, usually of the hierarchic [REYL74,RUWA74] or pipeline variety [FARB74]. But the overall system can be very inefficient. The average response times we quoted above for functionally specialized servers are valid only when the distribution

of server capacities exactly matches the load characteristics. If there is a mismatch then the average response times will be worse. Thus there is the problem of determining the exact characteristics of the workload on a system and making sure that it stays stable over time. Obtaining a balanced system and expanding it in a balanced fashion is not easy for small systems. For small hierarchical, or star, systems the central supervisory processor which allocates work to the specialized servers is likely to be underutilized (if there is to be any slack capacity for expansion) making the system non cost-effective. For large systems where each type of server is replicated many times balance is easier to achieve and the theoretical response time approaches that of a system with homogeneous servers, because the overall load at any particular instant does not vary far from the average load [KLEI74]. In hierarchical systems though, the central node is likely to run out of processing power so that it cannot handle the allocation of work to specialized servers fast enough to keep them busy.

All hierarchically organised multiple processor systems, ones with a supervisory processor, suffer from the twin problems of underutilization of the supervisory processor, and hence diminished cost effectiveness, in small systems, and eventual debilitating inadequacy of supervisory processor capacity as the system grows large. Since our stated aim is low cost small systems with

linear expandability we do not consider hierarchical systems further in this thesis.

As for functional specialization, we believe that the types of processor that will be manufactured in the greatest volumes will be general purpose processors. Referring back to our discussion of manufacturing costs in chapter 1, general purpose processors therefore will cost the least. So, because of their likely low cost and definite advantages in small systems, we concentrate on systems containing homogeneous processors and ignore functional specialization. It so happens however that the design we develop in this thesis can quite naturally handle functional specialized computers, as we show in chapter 7 when we discuss peripheral handlers.

Availability:

In theory both multiprocessors and distributed systems should be capable of graceful degradation as components fail. In practice, for general purpose systems, this is likely to be translated into high availability; a failing component need only be isolated, not repaired, before the system, with reduced capacity, can be used again. The single processor system is completely unusable in the event of a processor fault until the fault has been repaired.

In the production of highly reliable computers, distributed systems and multiprocessors can be used more effectively than double or triple replication of a single processor system. There are however special techniques involved in the attainment of high reliability [SCO[74], which we are not going to pursue in this thesis.

Large single processor systems:

From the queueing theory results above a single fast processor system would seem to be the best choice. There are however two points that need considering in relation to the queueing analysis:

1) Frequently large computer systems cannot be reasonably modelled as a single queue for processor service. Often channel capacity is a restricting factor and even if the system is balanced it is unlikely that there will be a single channel of sufficient capacity, rather there will be a number of channels (probably specialized) of lesser capacity so the poorer response characteristics of multiple servers could occur anyway.
2) The initial assumption in the comparisons was that the single processor was uniformly N times as fast as each processor in an N processor system. However it is unlikely that the single processor will be N times as fast at context switching. As a processor gets faster it uses more and more fast registers which will have

to be saved (or drained when pipelines are used) on context switching. To avoid too frequent context switching large systems use peripheral processors, communications processors and/or front end processors; hence incurring some of the disadvantages associated with multiple servers and functional specialization. Even with these aids a greater proportion of computing capacity is still likely to be wasted by context switching in the single processor environment than with slower processors where the 'opportunity' loss on a context switch is much smaller.

If it maintains its single server characteristics the large scale single processor system offers superior performance in general purpose computing compared with other architectures of equivalent capacity. But when the above factors are combined with the cost considerations we described in chapter 1, and the poor availability and expansion characteristics of single processor systems we see that the case for overall superiority is not so clear cut. Considering that they give relative ease of expansion, high availability and the possiblity of achieving a capacity not technically feasible with a single processor, multiprocessor systems and distributed systems are certainly worth investigating.

Distributed systems versus multiprocessors:

The distinguishing characteristic of a multiprocessor, its shared memory, gives the multiprocessor its advantages over distributed systems. These advantages are greater speed of interprocessor communication and larger size of contiguous memory. In a distributed system the various sites can only co-operate by sending messages to one another, which takes a longer time than using shared tables and semaphores in multiprocessor systems. (But since, for example, the processors in a distributed system do not have to co-operate over the management of shared primary memory, the inter processor communication mechanisms will be invoked less frequently than in multiprocessor systems). A large contiguous memory usually leads to greater efficiency in handling large problems [WITT68]. The packing problem, fitting complete jobs or working sets into available memory [AGRA75], is obviously less severe for one large memory than for a number of small memories.

Shared memory is also responsible for the poor features of multiprocessor systems: expensive and expansion limiting memory access circuitry, contention and software lockout.

In a multiprocessor system more hardware is required to provide access to shared memory (and to peripherals). The access speeds to memory will be slowed either by the

inclusion of a crossbar switch (with high initial cost and inflexible limit to expansion) or a bus for which processors have to bid. Alternatively the memory units can be multiple ported making them more expensive and again limiting expansion [SEAR75].

Memory contention occurs in multiprocessor systems when a processor cannot access a word of memory because some other processor is using the access circuitry. The partial solution to this can be expensive; replicating the access circuitry by providing storage in modules and then providing interleaving circuitry so that accesses are 'random'. With random access in a system where the number of processors is equal to the number of memory modules the utilisation of processors and memory falls quickly to 50% as the number of processors is increased [BHAN73a,BHAN73b, BURN73,BASK76]. However if the access time for a word is far shorter than the average time to process the word, as is likely to occur if MOS/LSI microprocessors are used [REYL74], then the effects will not be as severe as this. With high performance processors obtaining the necessary extra memory bandwidth to reduce contention could be costly. Cache design for multiprocessors is difficult [TANG76] and of dubious efficacy. In contrast caches can easily be employed in the single processor computers that constitute a distributed system, if they are required.

In multiprocessor systems software lockout occurs [MADN68]. Processors executing certain parts of the operating system will need to alter tables or have unique access to some resource. Other processors executing the same code will have to wait for the previous processor to finish. This problem can be ameliorated by setting many locks, each held for very short periods of time but then the cost of setting the locks begins to erode efficiency.

The two most publicised multiprocessor systems with more than two or three processors are C.mmp [WULF72, WULF74a] and Pluribus [HEAR73,ORNS75]. Both these systems try to circumvent the problems of shared memory by providing all processors with private memory as well. Pluribus is a special purpose system and the decision as to what goes into shared memory and what goes into private memory is a static one taken at design time. In the case of the general purpose C.mmp system there does not seem to be any methodology developed for using private memory. Private memory is only a partial solution to the above problems anyway, it lessens the amount of contention but does not significantly affect software lockout or the cost of the access circuitry.

A system developed to work where there is no shared memory could easily be adapted to a situation where some of the memory is shared, but the converse is not true. So it makes sense to develop a distributed system and

then see if some form of shared memory will improve performance while not degrading the expansion capabilities of the system. We raise this topic again in chapter 11.

With an appropriate communication subsystem and software organization a distributed system can exhibit most of the advantages a multiprocessor system has over a completely decoupled system of computers, while avoiding the limiting effects of shared memory. The next chapter examines the required features of a communication subsystem and chapters 4 to 8 are devoted to the development of the software organization.

Features of the distributed system we propose are:

1) It is a unified system with respect to peripherals.
2) Each memory is private to one processor. Low speed memory, matched to processor speed, can be used and there will not be any contention, bus or switch delays.
3) Less memory is required than for the same number of independent computers because one copy of most of the operating sytem is required for the whole system.
4) It is very modular, easily expandable and has high availability.
5) A form of software lockout will occur but it will probably involve less wasted processor capacity than software lockout in a multiprocessor system.

Delays will arise when a component of the operating system that is shared between sites is required simultaneously at two sites. However the waiting time need not be unproductive; the waiting site can do other work if there is any outstanding, in contrast to the 'busy' wait required at the low levels of multiprocessor operating systems.

6) Some of the management software will be as simple as that required if each site were an independent single computer, although other software will be as complex as that in multiprocessor systems.

7) There will be a communications overhead, which is not present in multiprocesor systems.

8) The response characteristics will be almost those of a multiprocessor system because the software structure comes close to implementing 'instantaneous jockeying'.

# CHAPTER 3

# COMMUNICATIONS

A distinguishing feature of distributed systems is that co-ordination and control of processors is performed by messages rather than by the use of common tables. Since we wished to study the software structures needed to ensure co-operation between the sites in a distributed system, our initial reaction was that the form of communication subsystem for passing messages between the sites was immaterial to our problem. However we soon came to realize that the properties of certain types of communication subsystem could have a significant effect on the nature and efficiency of some of the software mechanisms required. This chapter investigates what kind of interconnection structures, communication subsystems, are appropriate for distributed systems.

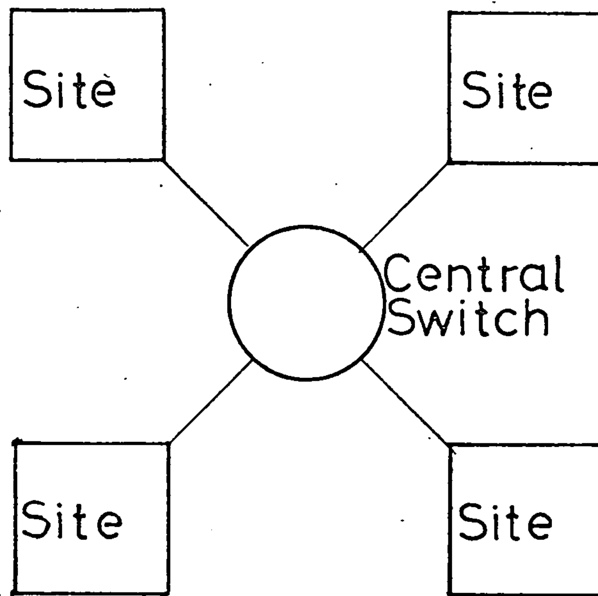
In a distributed system there are two types of communication, one, which we refer to as a message, is intended for one site only while the other, which we call a broadcast, is received by all sites in the system. Messages arise chiefly in the transmission of data and code between sites. Broadcasts can be used to propagate information about the overall state of the system.

First, we examine types of communication subsystem and then, in section 2, we examine how the type of communication subsystem impacts upon the flow of information in the distributed system.

## SECTION 1: COMMUNICATION SUBSYSTEMS.

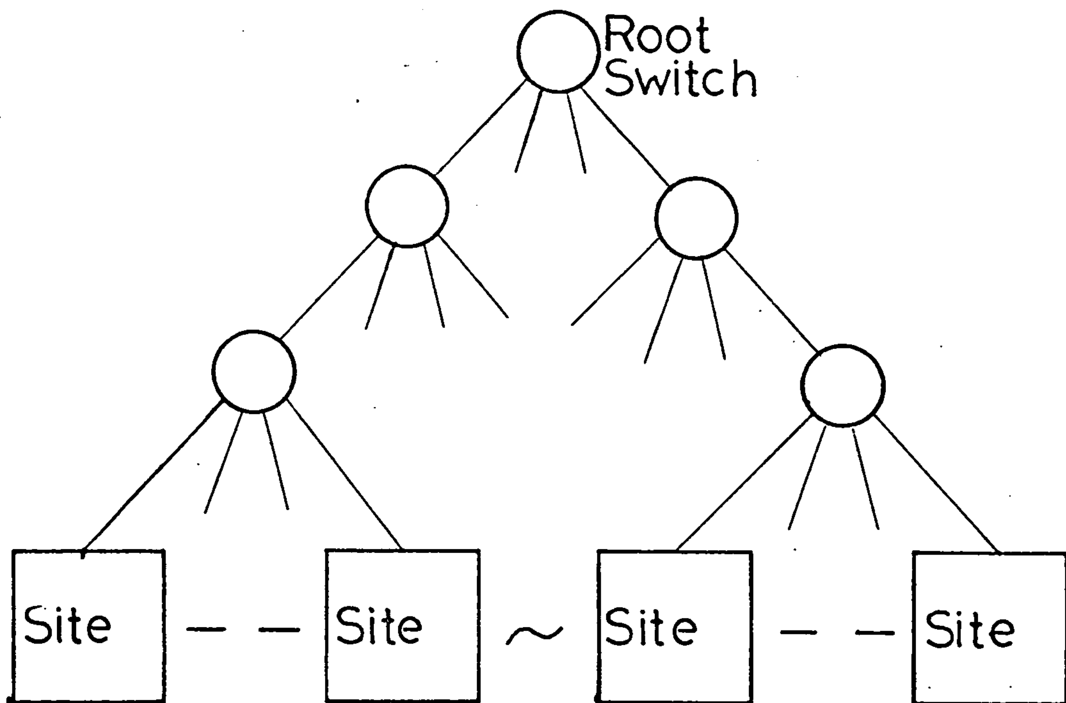
There are a number of criteria that we can use to distinguish the various types of computer interconnections, existing or planned [ANDE75,CHOU75, SEAR75]. For our distributed system we are looking primarily for low initial cost and expansion costs directly proportional to the number of sites in the system. Since we propose our computers to be separated by physically short (although electrically long) distances we do not require the existence of alternative routes between sites. Nevertheless we do not want the failure of a site to disrupt the communications between other sites. It is also desirable that the logic required for directing messages to their final destination be simple.

Centralized (star) communication systems (figure 3.1) undoubtedly offer easy routing but their cost is not




Star Communication System

Figure 3·1




Goodwin's Hierarchical System

Figure 3·2

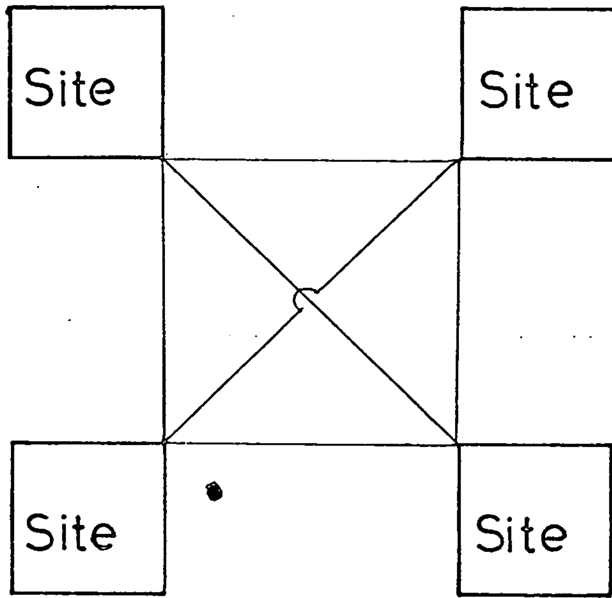
proportional to the size of the network. The central switch, be it a processor or other device [CULU76], is required whether there are two or twenty computers in the system. Further if this switch is going to have sufficient capacity to allow for reasonable expansion then it is going to be underutilized for small systems, probably making the small distributed system unattractive compared with an equal cost single processor system. One method of expanding the capacity of the central switch has been proposed by Goodwin [GOOD73,ANDE75]. He wanted to replace the centre switch by a whole tree of lower capacity switches (figure 3.2), expanding the size of the tree to give greater capacity when required. The cost is logarithmically proportional to the number of leaves (the computers doing the useful work) and the message direction algorithm is simple. But unless (undesirable) measures are taken to confine most communication to be between leaves that are close to each other, on average (n-1)/n of the messages will pass through the root switch when there are n nodes connected to it. Thus for message transmission at least, a tree structure gains little over a star network in capacity and introduces substantial delays to achieve this.

Of the non-centralized interconnection schemes a distinction can be drawn between those where the message travels directly to its destination without being copied and retransmitted, and those where a message travels in stages. The latter is often the preferred method in

trans-world type networks [ROBE70,KLEI70,POUZ73,HIRC74] where the complexity of routing is justified by the reduced cost and enhanced reliability of transmissions. The only simple structure of this type is the loop and as this meets the criteria of expandability and linear cost we will study it further, along with the two kinds of direct distributed communication subsystem: complete connection and shared bus.

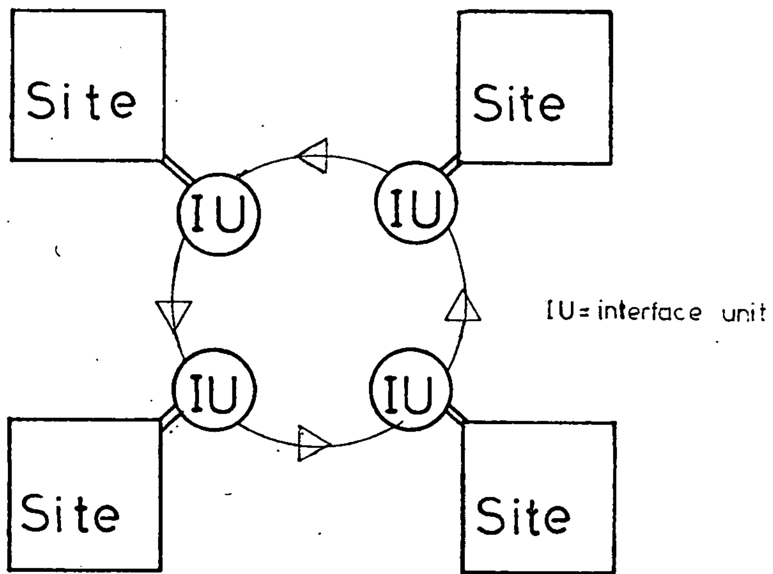
Complete connection:

A complete connection communication subsystem (figure 3.3) was proposed for the Karoline network [MADS72]. For small systems it has favourable features. Most computers have a few unused peripheral slots and simple links are cheap and quite easy to construct [LIND71] making initial cost low. There are no routing problems. Flow control, ensuring that there are not too many messages in the communication subsystem simultaneously, is not required as each link involves only two computers. An inoperative computer does not affect the links between the remaining operational computers. The total bandwidth grows as the number of computers in the system grows. Expansion is not directly limited but it does get progressively more expensive. The nth computer added requires n-1 links. Karoline being a network of 8 machines required 28 links. Bearing in mind that the links are probably quite cheap compared with other resources in the network, 28 links




Completely Connected System

Figure 3·3




Ring System

Figure 3·4

could well be the best form of communication system.

Broadcasting however, will usually consist of separate sequential transmissions to each of the other sites. This will present a greater load on the sending site than systems where a broadcast involves only one transmission.

Loop:

The DCS system [FARB72a], the initial version of the Maryland DCN project [LAYM74] and the Waterloo Mini-net [MANN75] all use a loop or ring communication subsystem as depicted in figure 3.4. In a simple form a ring system is a cheaper alternative to the complete connection system. For n sites n links are required and each site requires only one send slot and one receive slot. A site sends a message to its neighbour which decides if it is the message's destination or not. If it is not, then the message is passed on to the next neighbour. When a message has reached its destination it can be removed from the system (Maryland DCN) or marked as received, a copy kept, and allowed to circulate back to the sender (DCS). This later option provides an automatic though expensive acknowledgement. Given that this complete loop traversal is to take place, a broadcast involves the same overheads as a message.

A message however causes interruptions to all sites it travels through and so sophisticated ring systems such as DCS use special units, ring interfaces, one for each site. Each interface unit buffers messages and only interrupts its site if the message is for it [REAM76]. With intelligent design, the ring interface units also overcome the problem of the whole loop becoming inoperative should one computer in it fail: in such circumstances the ring interface unit can simply pass all messages on. The use of special units means that beneficial features, discussed in section 2, can be added.

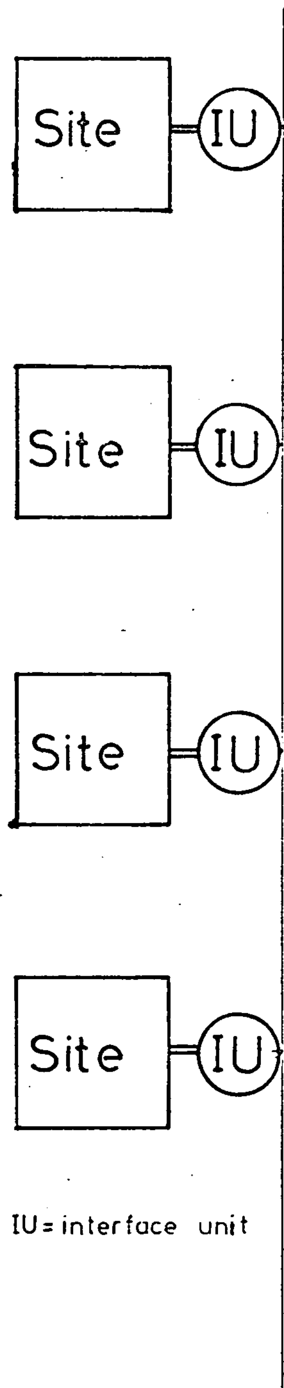
The total bandwidth of a loop system is fixed. As more computers are added the bandwidth available to each decreases and the average time for a message to reach its destination increases. Since there can be a number of messages in the loop the question of flow control arises. If a site puts a new message in the loop without regard for messages that may arrive and require retransmission, messages will have to be destroyed. Simple forms of flow control can involve considerable loss of bandwidth. The flow control schemes of some loop systems have been evaluated by Reames and Liu [REAM76]. The Newhall loop uses a round robin, token to send new message, scheme. A site can only introduce new messages into the loop when it has the token, it sends the token onto the next site in the loop at the end of its new messages. The Pierce loop divides the bandwidth into fixed size slots or

'message crates' and a site can send a new message if an empty crate is passing through its interface unit. Unless messages are all the size of slots or less, they have to be broken into packets with all the attendant problems of disassembly, sequencing, buffering and reassembly [FRAN72]. The DCLN loop of Reames and Liu uses buffers in the interface unit to hold incoming messages (that have to be retransmitted) while new messages are introduced into the loop. Thus any site, providing its buffer has space equivalent to the length of the new message, can introduce a new message almost immediately. Although transmission time around the loop is increased it is shown by Reames and Liu that, overall, messages arrive faster than in the other two schemes because they do not have to wait so long to enter the loop.

If it is desired to stop an errant computer from monopolising the available bandwidth a distributed control scheme leads to further loss of bandwidth. In the DCS system control over runaway sites takes the form of the ring interface units permitting each site one outstanding message at a time [FARB72c].

Shared bus:

The KOCOS system [AISO75] uses a conventional 32 bit wide bus while Ethernet [METC76] is a serial bus of particularly simple construction. For the distributed



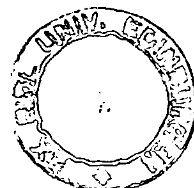

Shared Bus System

Figure 3·5

use of a bus some interfacing unit is mandatory (figure 3.5). KOCOS uses one which also aids in controlling interprocess communication [WALD72].

The Ethernet interface does not have extra functions but would, with the addition of an associative memory function, come closest to what we think would be the ideal type of communication subsystem for a distributed system. As it stands it is an adaptation of a type of ALOHA net [ABRA70,BIND75] with 'radio' transmission constrained to be along about 1 Km of co-axial cable. The interface units have a policy of deferment; they will not start transmitting a message if they detect a transmission is in progress. This means that collisions (and subsequent aborting of transmissions) can only occur in the first part of a transmission, in the period equal to the round trip time - for Ethernet less than 8 microseconds. With long packets, 4096 bytes, and the use of a 'quadratic back-off' policy when transmitting after collisions, a utilization of the communication subsystem of over 95% is expected when it is heavily loaded. [METC76]. Unlike KOCOS which has a round robin policy for control of the bus, in Ethernet any site can send a message immediately if the communication subsystem is not already in use.

The total bandwidth of a bus is limited but, unlike the loop, there is not an increase in message transmission time as more sites are added. Flow control



in KOCOS is provided by the round robin scheme while in Ethernet it is done by a 'back-off' policy whereby if messages collide retransmission is not attempted for a random period, the mean of which increases with the recent collision rate.

A broadcast in a shared bus system can be effected with a single transmission. Suitable design of interface units can ensure that the bus is not brought down by the failure of a site.

## SECTION 2: INFORMATION GATHERING.

### Global object management:

As will be described in detail in later chapters, there are certain objects in the distributed system that are global; any site must be able to locate the sites where these objects currently reside. As the size of a distributed system goes up the movement of global objects between sites will increase. Thus we need to be concerned with the efficiency of management of global objects. There are several ways that the location of global objects can be determined.

1) Continuous updating: Every time a global object moves a broadcast of the form "X has moved to site I" is performed. Each site has a directory of global objects which it updates when it receives the broadcast.

2) Central directory: One site is specially designated as a directory site. Each time an object moves a message of the form "X has moved to site I" is sent to the directory site. To determine the location of an object a site sends a "Where is X" message to the directory site which sends a return message "X is at I". So one message is sent every time an object moves and two are required to determine its location. A central directory is in some sense antithetical to a distributed system. However there exist schemes for nominating a new site as the directory holder should the old one fail [TYME71] and a directory can quickly be reconstituted with a broadcast of "What global objects do you have". We cannot escape the fact that part of the directory site's workload will be inherently different from the rest of the distributed system (perhaps causing problems with load balancing). Should this workload prove to be a bottleneck then a hybrid system with a number of directory sites using continuous updating amongst themselves can be used. Each directory site would service a different set of non-directory sites. So an object move generates a message to one directory site and a 'limited'

broadcast from that directory site to all other directory sites.

3) Search: No directories are held at any site and there are no updating messages or broadcasts when a global object moves. Instead, in this scheme every time a site wants to know where an object is it broadcasts "Where is X". The site where the object resides replies with a message "X is at my site".

4) Associative: The only reason a site can have for wanting to know the location of a global object is so that it can send a message (related to the object) to the site the object is at. An alternative form of the search scheme is simply to broadcast the relevant 'message' and have each site decide if the broadcast is related to any global object currently residing at it. Although this form of search involves less messages than the other, the length of the broadcast is likely to be a lot longer. Hence direct broadcasting is only appropriate when a broadcast involves the same load on the communication subsystem as does a single message, namely when systems have interface units. A direct broadcast scheme can be made most attractive by the use of extra hardware in the interface units. If an associative memory, containing the names of all the global objects at a site, is attached to the message receiver at each site then the decison to accept a broadcast can be made

without reference to the main processor [FARB72c]. There is no need for directories to be kept, or updating information broadcast, when objects move. There is no delay when a message has to be sent to (the site at which resides) a global object and sites are not continually being interrupted to answer "Where is X" broadcasts. Whether an associative memory is used or not, direct broadcasts require care with synchronization; the global object may be in transit between sites when the broadcast is made so that no site picks up the message.

To compare the schemes outlined above we assume that each site requires to know the whereabouts of a global object Q times a second. We further assume that a fixed fraction r of these seekings of global objects results in the object being moved. This fraction r is substantially less than 1. These figures are assumed to be independent of N the number of sites in the distributed system.

Of the above schemes the search method is definitely inferior to continuous updating. The computation required to update a directory may be equivalent to that required to determine if a global object is resident but not all requests for the location of an object result in the eventual moving of the object. Hence the continuous updating method involves fewer broadcasts, and does not involve the extra "X is at my site" message nor the enforced delay while the information is gathered; all for

the cost of memory space to hold a directory at each site. In the distributed system we are proposing the number of global objects is likely to be of the order of 10 to 50 so the cost of holding a directory at each site is not great.

The evaluation of the other schemes requires consideration of the form of broadcasting. We have seen that for the bus and the DCS type loop a broadcast costs the same as a message in terms of the use made of the total bandwidth. Also the work done by the sender is identical for either. (The total work done by the receivers of a broadcast will always be N-1 times that for a message). For simple complete connected schemes a broadcast will use N-1 times the bandwidth that a message uses, and the sending site will probably have to do N-1 times the work. For either type of communication subsystem the total number of messages (related to global object management) received per second for the whole system will be N(N-1)Qr when using continuous updating. When using a central directory scheme (N-1)Qr update messages will be received by the directory site per second, (N-2)Q messages will be received by the directory site requesting the whereabouts of a global object and the same number of replies will be received at the non directory sites, making a total of

Q((N-1)r+2N-4) messages per second.

Thus, considering only the minimization of work done receiving messages for a value of r = 1% (which turns out

to be a high value, see in the sample outputs of appendix A the ratio of TRANSFERRED DOMAINS to TRANSFERRED PROCESSORS), the number of sites, N, would have to be greater than 203 for a central directory to perform better than continuous updating. When a directly connected communication subsystem is used, the number of transmissions is the same as the number of receptions. But for a communication subsystem where a broadcast costs the same as a message then the overall work done using continuous updating is less, so that N will have to be even larger before break-even point is reached. By the time we quantify the inconvenience of having to wait before a global object's location can be retrieved, it is obvious that a central directory is inferior to continuous updating.

We have already mentioned that an associative scheme is not appropriate for a system with a directly connected communication subsystem. So, for such a system, continuous updating of directories held at every site is the best scheme.

In an associative scheme there are no management messages sent whereas, for a loop or bus, a continuous updating scheme gives NQr broadcasts per second resulting in N(N-1)Qr messages received. The fraction of total available processor power used in maintaining the updating is directly proportional to N. For either scheme the fraction of processing power used in actually

shifting global objects is constant. Hence the associative scheme is preferable to continuous updating, at least when large scale sites are envisaged. The interface units required for an associative scheme may not be cost effective for a distributed system of very low powered computers.

Status updating:

We show later (in chapter 7) that there is a need for each site in a distributed system to have some information about the status of other sites. While the information each site requires about the others is not very much, it must be reasonably up to date. The ideal is that every site has completely accurate information about every other site, but finite communication bandwidth makes its achievement impossible. A distributed system can tolerate some misinformation, but the more inaccuracies there are the less efficient the system will become. Below we discuss four ways that sites can interchange information.

1) Broadcasts at regular intervals: This policy has the obvious disadvantage that the number of broadcasts will go up in direct proportion to the number of sites. Since every site will have to be interrupted to receive its message from every other site, the fraction of computing power in the distributed system

dedicated to updating this information will be directly proportional to the number of computers in the system.

2) Exchanges between neighbours: To mitigate the interruptions caused by receiving broadcasts from every site, each site could be arbitrarily assigned several neighbours with whom they exchange tables of the supposed state of the whole system, similar to the way routing information is updated in the ARPA network [MCQU72]. The neighbour relationship would have to be intransitive so that information about every site in the system would work its way through to all other sites. The items of information built up from exchanged tables will be of different vintages. There can be no guarantee that sites will confine their normal transactions to their neighbours; the frequency of exchange of information will have to be high if a good proportion of the information is not to be hopelessly out of date.

3) Appended to normal messages: Since the amount of information each site would want to propagate about its state is quite small, perhaps 2 bytes worth, it can be appended to normal messages between sites without increasing overheads significantly. Indeed in systems such as Ethernet [METC76] there is a fixed minimum length message and since many control messages could be shorter than this length, the information about the sender's site could be carried for free. The sending of messages is likely to be correlated

with changes of state of the site, and hence with the need to update the information held at other sites. When two sites are interacting heavily they would have their information about each other updated frequently. When a site is idle and not interacting with other sites, its status would not be changing, so it would not interrupt other sites to give them information they already have.

4) Eavesdropping: In a system that appends state information to messages and has associative interface units, such as loop or bus systems, the interface units can take over the intelligence gathering function. Further they need not use messages addressed tojust their site, but can pick the state information (and, of course, source) of all messages that pass on the loop or bus. The interface unit would maintain a status table so as not to interrupt the kernel too frequently. The kernel could consult the table when required.

Compared with the first two methods, appending information to normal messages has the obvious advantages in the conservation of bandwidth and minimization of interruptions to sites. The differences between information gathered from all messages transmitted and from only the messages received at one site will be minor if broadcasts are a frequent occurance. Thus when a directory update scheme of global object management is being used (with its broadcasts of changed object

location) the information contained in only the messages received at a site will probably be sufficient. However, because there are few or no universally received broadcasts, eavesdropping will probably be required in a system with a bus or loop type communication subsystem (that used an associative scheme for managing global objects). There is nothing to stop a site performing a dummy broadcast when it felt its status had reached some critical point and this would help homogenize the information held at all the sites. Whether or not the extra hardware complexity of eavesdropping would be justified requires investigation.

In the simulation of a distributed system described subsequently we assume a completely connected system. Continuous updating is used to locate global objects and status information is appended to normal messages. This, we considered, would represent a practical implementation at the present time. However we feel that any major implementation in the future should involve the construction of an Ethernet type of bus with associative recogniton of addresses and perhaps an eavesdropping mechanism to gather status information. Distributed control serial buses, like Ethernet, offer ultimately very high bandwidths using very cheap materials [ADAM76], the transmitting media (co-axial cables, twisted wire pairs or optical fibres) are passive giving immensely enhanced reliability compared to schemes involving a complex of electronics in the transmission.

# CHAPTER 4

# OPERATING SYSTEMS ARCHITECTURE

The designer of an operating system for a distributed system has two alternatives: he can attempt to 'distribute' some form of existing single site operating system architecture or he can invent something completely new. Lacking the required inspiration for the latter approach we have chosen the former. Consequently, to decide on an appropriate architecture for an operating system, in a distributed system we now look first at those for single or multiprocessor/shared memory systems.

## SECTION 1: SINGLE SITE SYSTEMS ARCHITECTURE.

Apart from manufacturer's monolithic monstrosities, operating systems can be classified into four types of architecture. The classification is made according to how users, resource allocators and other operating system services are permitted to interact. A goal of all architectures is to make interactions between functions 'clean'. Ideally each function does not have to make any assumptions about how other functions are realized. We emphasize before we describe the categories that they are not mutually exclusive.

Hierarchical:

Dijkstra is the initial proponent and publiciser of the strictly hierarchical architecture [DIJK68,DIJK71, PARN74a]. Each function of an operating system is statically assigned a unique level. The first level function is programmed to work on the bare hardware. The second level is programmed for a system consisting of hardware plus the first level. It should not have direct access to the resources controlled by the first level, rather it should invoke the primitives provided by the first level. Likewise the second level provides the environment in which the third level is programmed and so on. Each layer 'rebuilds' the machine into a more attractive machine. In Dijkstra's view an operating system should be regarded as a sequence of layers, built on top of each other and each of them implementing a given imorovement [DIJK71]. Implementing a strictly hierarchical system requires a firm belief that functions can be totally ordered, a foreswearing of co-routine type interactions between functions, and skill in determining the correct ordering. Interactions between functions can be one way only.

Virtual machines:

Variants of the virtual machine architecture form the largest class of extant structured operating systems.

Basically every user has his access to resources (including core and CPU time) controlled by a single virtual machine monitor or kernel (we prefer the later term). This kernel is entered, perhaps by instruction traps, every time the user wishes to acquire or release resources, and it ensures a 'fair' distribution of the resources. The user is encapsulated. He cannot communicate or interact with other users, he is to all intents and purposes using a private computer, a virtual machine.

The pure virtual machine variant provides no more facilities to the user than the bare underlying hardware (or the hardware of another machine) [MEYE70,PARM72,BUZE73,GOLD73]. The user has to provide himself with an operating system to run in his virtual machine. This can lead to horrendous inefficiencies [GOLD74]. The kernel knows nothing of the behaviour of the operating systems in the virtual machines, nor are the operating systems aware that there is a kernel beneath them. The advantages claimed for this kind of virtual machine are that it provides absolute security because there is no interaction between virtual machines [POPE74] (which security has proved elusive [ATIA76]) and allows for the development of new versions of operating systems concurrently with the use of previous versions.

In other virtual machine type operating systems such as EMAS [WHIT73] or MULTICS [CORB72], the kernel (called

Supervisor in EMAS) provides a number of services, such as managing paging and dispatching, to enhance the bare machine. The individual 'operating systems' (Directors in the case of EMAS) are integrated with this kernel. They do not duplicate the provided facilities and they could not run on the bare machine. Tuning of integrated systems does not present the same dificulties as does tuning a pure virtual machine system. The harsh principle of the user having access to his virtual machine, and nothing else, can be softened by kernels that allow limited interaction with other virtual machines, usually via the filing or I/O subsystems.

Intercommunicating processes:

Process orientated systems have received a lot of attention in the literature [KNOT74] and are exemplified by the RC4000 system of Brinch Hansen [HANS70]. The kernel, the basic addition to the hardware, provides the primitives for process management, creation, deletion and intercommunication. The rest of the system is a set of processes. In particular, resources are identified with the processes that control them. Processes are capable of interacting with any other process, which is a considerable difference from the virtual machine situation. This interaction is accomplished using messages. The kernel provides primitives such as 'send', 'receive' and 'wait for answer' which buffer messages and

suspend processes. The kernel normally implements an addressing scheme that gives processes unique names and allows messages to be addressed using these names. In some systems extra refinements are added, such as ports [BALZ71], so that a process does not even have to be aware of the name of the process with which it is communicating.

Parallel execution of a program is catered for in a process orientated system. A subroutine call can be implemented as a message (containing the parameters) to a process, this process returning a message when it is finished. Thus systems often provide for the creation and destruction of processes and the placing of processes in a hierarchy of ownership (parenthood). This feature, although used by Brinch Hansen in RC4000, has recently been criticized by him as being very costly in runtime checking of the validity of process interactions [HANS74,HANS75]. He advocates that an operating system should consist of a fixed number of processes, at least for a given configuration with fixed resources.

Hansen is also critical of messages passing systems because they create an artificial resource, message buffers [HANS73]. Message buffers require management; their allocation has to be carefully controlled if deadlock, through insufficient message buffers, is to be avoided. Transmission of messages involves copying messages into and out of buffers, which is highly

wasteful of processing power, at least in single site systems. (One message passing architecture, that of the GEC 4080 [GECC75], has microprogrammed functions to help with message passing, making it more efficient). Lampson [LAMP71] feels that message systems are not convenient to the user; elaborate conventions, or contortions [SPIE73b], are required to find out the unique names of the operating system facilities the user requires.

Kernel/domain architectures:

Maintaining effective control in operating systems that permit general interactions has been likened to 'running a three ring circus, in one ring, in the dark' [METC72a]. Capabilities are the basis of a mechanism that allows general interactions to be controlled. Capabilities allow each computation access to all the resources it needs at a particular time. All resources are intrinsically shareable, but the computation is not permitted access to resources that, at its current stage, it does not require. Strictly speaking a computation has access to all resources, and only those, for which it possesses a capability [DENN66]; the assumption being made that the ownership of capabilities is so organised to reflect the current requirements of the computation.

The set of resources that, at any time, a computation has access to is called a domain [LAMP71,NEED74], also

sphere of protection [DENN66], parameter space [EVAN67], NCP [SPOO71], local name space (LNS) [WULF74], protected subsystem [SALT74] and domain incarnation [SPIE73a]. Should a computation prove erroneous its effect is likely to be limited to the current set of resources. The resources are enclosed in a 'firewall' and incorrect operations are contained and do not affect the rest of the system. Capabilities normally restrict the type of access a computation has to its resources; for example a segment may be accessed as read/write, read only or execute.

The basic function of a kernel in a capability system is twofold:

1) It enforces, or assists the hardware [NEED74,ENGL74] to enforce, the restrictions on the type of access to resources, including null access to resources for which no capability exists. For example the kernel should detect and disallow a destroy operation on a file when the computation only has the capability to read from the file.
2) The kernel assists computations to change the set of resources that they have access to (when this function is not carried out entirely by the hardware). We call this operation an interdomain jump. The kernel, in giving and removing access to resources, can control allocation of resources if it wishes.

Process dispatching is usually included in the kernel also, either for operational efficiency or to ensure

fairness in the allocation of processing capacity [WULF75b].

Resource management in capability systems is performed in two different manners. Either a computation is given direct access to a resource by being given a capability for the resource, or the computation is given just a capability for the execution of a piece of code that manages the resource. In the latter case, to execute the code, the computation changes its domain, or protection environment, and the resource becomes available to it. But the resource is available to the computation only for as long as it executes the appropriate code.

In many capability systems the kernels provide the facilities by which a computation can create, delete, copy, contract the types of access, or expand the types of access [FERR74] of a capability. These facilities are appropriate when the type of dynamic creation and deletion of processes (and accompanying resources), mentioned above in relation to message passing systems, forms the underlying philosophy of the system. We have adopted the same attitude as Brinch Hansen and tried to do without such dynamic behaviour. There are unsolved problems in combining copying of capabilities with the ability to delete them [REDE74] and we think these problems would only be exacerbated in a network environment.

## SECTION 2: DISTRIBUTED OPERATING SYSTEMS.

One of our goals in designing a distributed system is that there should be as little as possible duplication of operating systems functions at different sites. We want the normal work of the system to be uniformly distributed across the system and, following the philosophy of Spooner [SPOO71] and others that the constituents of the operating system should not be specially privileged, we determined that the ideal is to have systems functions spread across the system as well.

Another goal, derived from the queueing theory considerations expressed in chapter 2, is to have no site idle while there is work waiting to be performed at other sites. This implies that load levelling or balancing operations must occur frequently, and that the overhead of these operations is an important factor in the success of a distributed system.

With these two goals in mind we now examine the four types of operating system architecture, outlined above, for their suitability for extension to distributed systems.

Hierarchical:

The hierarchical scheme is superficially the most attractive of the architectures to extend to a distributed system. The 'only' requirement is to provide a bottom layer that somehow melds the different machines in the systems into a 'more attractive' single machine upon which Dijkstra's or any other operating system can be placed. Goodwin [GOOD73] has tried to take this approach with his tree structured distributed system. The basic layer provides for communication between physical processors and a tree structured naming mechanism. On top of this was planned a process communication system; the bottom layer taking care of messages for processors that do not belong at the same site as the sender process. We have already criticized Goodwin's proposals because of the likelihood of half, or more, of the messages travelling through the root node. A further criticism, stemming from adherence to hierarchical layering, is that there can be no migration of load from overworked sites to idle sites. The bottom layer has no concept of processor allocation, that belongs to higher levels. The higher levels do not know that the underlying machine is in fact a distributed system, for that is against the rules of the game. Also the assigning of processes to sites has to be done outside the system and would have to be done every time the system was reconfigured.

The difficulty with incremental layered machine improvements in a distributed system is that in order to load level and balance the use of resources, there has to be some two way interactions. Structuring systems into layers is a good technique but a practical system must have many interacting functions in each layer.

Virtual machines:

It is pertinent to enquire, if one is adopting a strict virtual machine architecture, whether it is worth having a distributed system at all. The purpose of the virtual machine architecture is to create a set of private 'bare' machines. Quite possibly all the kernel for a distributed system would be doing is tying together a number of physical machines just so that it can simulate the same number of virtual machines. Thus if the division of virtual machines is fairly static, greater efficiency would be obtained by not integrating the physical machines together, dispensing with the virtual machine monitor, and putting the virtual machine operating systems onto the physical machines.

When there is intended to be a multiplicity of virtual machines at each site in the distributed system then a distributed system could be justified by the possibility of load levelling. A kernel would reside at each site and manage all the virtual machines at that site, as it

would in a single site system. But, somehow, a load levelling apparatus could be incorporated so that the kernels could co-operate in moving virtual machines (by copying their total memory space) away from busy sites to idle sites. Problems arise both at the level of determining opportune times to shift virtual machines and then of handling peripheral devices after a virtual machine has been shifted. The kernels would probably waste a lot of time time polling each other to see how busy they all were and would be likely to grow rather large to handle the intricacies of shared peripheral devices.

Karoline [MADS72] was planned to have 8 virtual machines at each of 8 sites, but proposals for load levelling, if they were considered, were not published.

For the less strict virtual machine architecture where the virtual machine monitor or kernel provides many services, and sharing of files is permitted, there have been at least two implemented distributed systems, RSEXEC [THOM73,COSE75] and SBS [AKKO72,AKKO74,AKKO75]. These systems take what might be called the hypervisor approach. Each site maintains a full operating system or supervisor and extra facilities are added, often at a user level, to form the hypervisor, integrating the site into the distributed system. So far these extra facilities have just implemented network wide file systems so that files (and peripherals) can be accessed

by a user from any site in the distributed system. This feature has been exploited, at least in RSEXEC [COSE75], to attempt load balancing at 'log in' or job initiation time by directing users to the least utilized site. No mechanism has been developed for moving a job from site to site once its execution has begun.

The advantage of these types of system is that they can be built on top of existing operating systems, or at least those that have been sympathetically designed [METC72b,ZELK74,RETZ75]. The disadvantages are the duplication of operating systems at each site and the inability to load level, except crudely as above, because these operating systems are really autonomous units.

Intercommunicating processes:

The Distributed Computer System (DCS) being developed by Farber and colleagues [FARB72a,b,c,d,FARB75,ROWE73], is the archetype of process orientated distributed systems [LAYM74]. We have already mentioned two features of the DCS system in chapter 3. It has integrated its hardware into the system design by employing an associative mechanism in its communication system for direct addressing of global objects. The global objects in this case are processes. Also DCS broadcasts are as efficient in the use of bandwidth as are single messages.

The kernel at each site is extended (from single site form) to place any interprocess messages that it cannot deliver at its own site onto the network communication loop. There they will be picked up by the appropriate kernel (because it has set the names of all resident processes in the associative memory of its interface unit) and eventually delivered to the correct process. The other major change in making a distributed system is in resource allocation. Resources, we said, were identified with processes in process orientated systems. The management of these resource controlling processes can be carried out by allocator processes. DCS has one allocator per site (though not necessarily residing at that site). The interaction between users and allocators is modelled on microeconomic theory and is the basis of load balancing in DCS.

When a user requires a service, the execution of a particular type of process (such as a text editor), which will use resources (memory and perhaps peripherals), he (his agent process) sends a message to all allocators requesting a 'bid' for the provision of the service required. The allocators all answer to a common name so that only one message on the communication loop is required to ask for bids. Allocators return bids and after a fixed period of time the user evaluates the bids he has received and chooses the allocator with the smallest bid. He sends this allocator a 'contract' message. The allocator can then create a process of the

required type at its site and return the process name to the user. But bids are not binding and so an allocator could have allocated elsewhere some of its resources in the time taken to evaluate bids, in which case the 'contract' is spurned and the user has to start requesting bids all over again.

Thus DCS load balances basically at a job-step or complete command level. From the above description, for an N site system, 1+(N-1)+2 messages on the communication loop are required for a first time successful allocation of a process to a user (when the allocated process is at a different site from the user's agent process). Thus it would seem that the overhead would be too great for attempting finer load balancing.

There are other process orientated distributed systems under development, DCN [LAYM74,MILL76] is one, POGOS [DUVA75] is another. But, as far as we are aware, a load balancing or load levelling strategy has not been published for any but DCS, and there has been no published evaluation of the operation of DCS. We have been told however that for POGOS, a network of 16 or more identical minicomputers, attempts at load levelling produced instability and were abandoned. Processes were being transferred around the system too fast to do any useful work between moves. Unfortunately, no details have been published. A very recent paper [MILL76] states that load levelling mechanisms are still to be developed

for DCN.

Finally one other feature of DCN, POGOS and DCS is the duplication of non-kernel code at all sites. In DCN all functions, that is the code for the processes that implement these functions, reside at each site. Migration of a function involves shifting only the port name of the process to a new site [LAYM74]. Primary memory space has been permanently traded for decreased traffic on the communication loop. In POGOS a copy of the whole POGOS operating system, which admittedly is quite small and primitive, resides at each site. DCS does have duplication for fail-soft reasons but it is required also because any site, if underloaded, has to be able to create (almost) any process.

Kernel/domain architectures:

The functions of a kernel in a single site domain system are to multiplex ready-to-run computations on the physical processor and to handle the interdomain jumps. We emphasize again that when a computation has entered a domain it accesses resources within the domain and no others. Thus in a distributed system a process will be able to execute unimpeded when all the components of a domain are at one site. If there is a kernel at each site and it provides a distributed interdomain jump which ensures all the components are at one site, the rest of a

single site domain system can run with no alterations, just as an interprocess communication system can run once the communication primitives have been extended.

The distributed interdomain jump is the key to the operation of a distributed kernel/domain system. A process wishing to change domains notifies its local kernel (that is the kernel at the site where it was executing in the domain it now wishes to leave). This kernel has to locate the new domain (domains are the global objects in this system) and in co-operation with other kernels, choose a site at which the process is to enter the new domain. The kernels then have to co-operate, by sending messages to each other, in shifting the domain components to that site, if any need shifting. When all the components of the domain are at the chosen site the kernel there schedules the process for execution again. The distributed interdomain jump allows load balancing, as distinct from load levelling, to be performed at quite a fine level. Work is not arbitrarily moved around to level the load at each site, but each request to enter a domain is taken as an opportunity to shift the components of the domain to another site if the current status of all the sites indicates that this is desirable. Every time an interdomain jump occurs there is an opportunity for the system to move towards balanced loading. The occasion of an interdomain jump is also optimum with respect to the volume of data that has to be moved if the computation

changes site. At most, all components of the new domain will have to change site; frequently some of the components will already be at the new site. The choice of new site can be made to minimize traffic on the communication subsystem.

Synopsis:

In this chapter we have examined various types of operating system architecture and their suitability for extension to distributed systems. We have shown that a strict one function per level hierarchical system is not suitable because load balancing cannot take place. Those systems that have kernels at the lowest level that implement several co-operating functions, can be more readily extended to distributed systems. The inefficiencies of strict virtual machine architectures seem likely to be increased but, logically at least, both process orientated systems and domain systems are suitable for extension to distributed systems. We have indicated areas that are considered by some to be drawbacks of process intercommunication systems per se, and we have stated what we consider to be the drawbacks of process intercommunication systems as a basis for distributed systems. A change in emphasis away from processors towards domains, away from managing messages towards managing environments, provides, we feel, the key to a successful distributed system. A distributed

operating system based on the kernel/domain architecture offers great potential both for minimizing the duplication of code and for fine grain load balancing.

The rest of this thesis describes more thoroughly the kernel/domain architecture, outlines strategies and mechanisms that could be employed in implementing the distributed interdomain jump, develops some of these mechanisms, describes a simulation program that 'exercised' these mechanisms and analyses the results of this simulation.

# CHAPTER 5

# THE DEVELOPMENT OF THE DOMAIN CONCEPT

## Introduction and terminology:

This chapter presents a survey of the development of the domain concept. We show the connection between segments and capabilities and show how capabilities are used to define domains. Our intention is to demonstrate that domains can be considered to be the predominant structure in a computer system.

The concept of a segment dates back at least to the Burroughs B5000 [BURR61]. A segment's attributes are some form of identification or name, and a length or total number of data objects (normally computer words, bytes or instructions). Elements of a segment are accessed by identifying the segment and specifying an offset within the segment. It is assumed that the segment's elements are stored contiguously or, as in a paged system, discontinuities are taken care of by subsidiary addressing mechanisms. If ambiguity of addressing is to be avoided a segment needs a name unique to all the possible environments in which it will be used. If addressing is to be controlled, as in a protection scheme, then the generation of segment names has to be controlled. Both the method of naming segments

and the mapping of segment names into hardware segment starting addresses have been the subject of a great deal of study.

Dennis and Van Horn [DENN66] are generally credited with being the pioneers of protection schemes and being the first to use the term 'capability'. A capability is essentially a name, or a pointer; a computation that possesses a capability can access the item named. Capabilities can name general objects or resources [LAMP71]. The capability concept has been formalized by recent writers [WULF74,FERR74,LAMP76] so that a capability consists of three items:

1) a type denoting the class of object named (of which segment is one such class)
2) a value being the name or identification of the object
3) a set of rights indicating how the named objects may be manipulated by the holder of the capability (the available set of rights will depend on the type of the object).

We discuss later how resources can be associated with segments so we restrict our interest initially to capabilities for segments only [PARN74b] (and later to entry capabilites which are capabilities for special groups of segments). The type of access permitted to a segment is not really germane to the development of the domain concept. Hence we will consider a capability to

be synonymous with the name of a segment or a pointer to a segment. So, in tracing the development of the concept of a domain, we concentrate mainly on models of computer operation where the only resources in a domain are segments. We are interested in the segments accessible to a computation as the computation proceeds.

Of particular importance is the sharing of segments between different domains or environments. When it is desired to shift a computation from one site in a distributed system to another then all the segments currently accessible to the computation (i.e. its domain) have to be collected together at the new site. This operation will be considerably complicated if some of the segments are simultaneously accessible to other computations.

Before we go on to examine various models we attempt to clarify some of our terminology. The term 'process' in Computer Science has collected many different shades of meaning. Spier [SPIE73a] makes a cogent case for using the term 'virtual processor' to denote the idea of execution of a user's sequential computation. A virtual processor is in a one to one relationship with a user, and the user's computation proceeds only when a physical processor is allocated to the virtual processor. A virtual processor executes (potentially) all the code that defines a user's computation but neither code nor state space [DIJK71] define a virtual processor. The

virtual processor is an agent acting on behalf of the user. It is the pseudo-processor of Saltzer [SALT66]. In the following sections we have altered the notation of the original descriptions when these used the term 'process' to mean no more than virtual processor as we have defined it above. We have retained the term 'process' however when there are other connotations; for example when a segment of code defines a process and a subroutine call implies a change of process, or when user level parallelism permits a user to 'own' many processes at once, or when resources are managed by processes.

The Evans and LeClerc model:

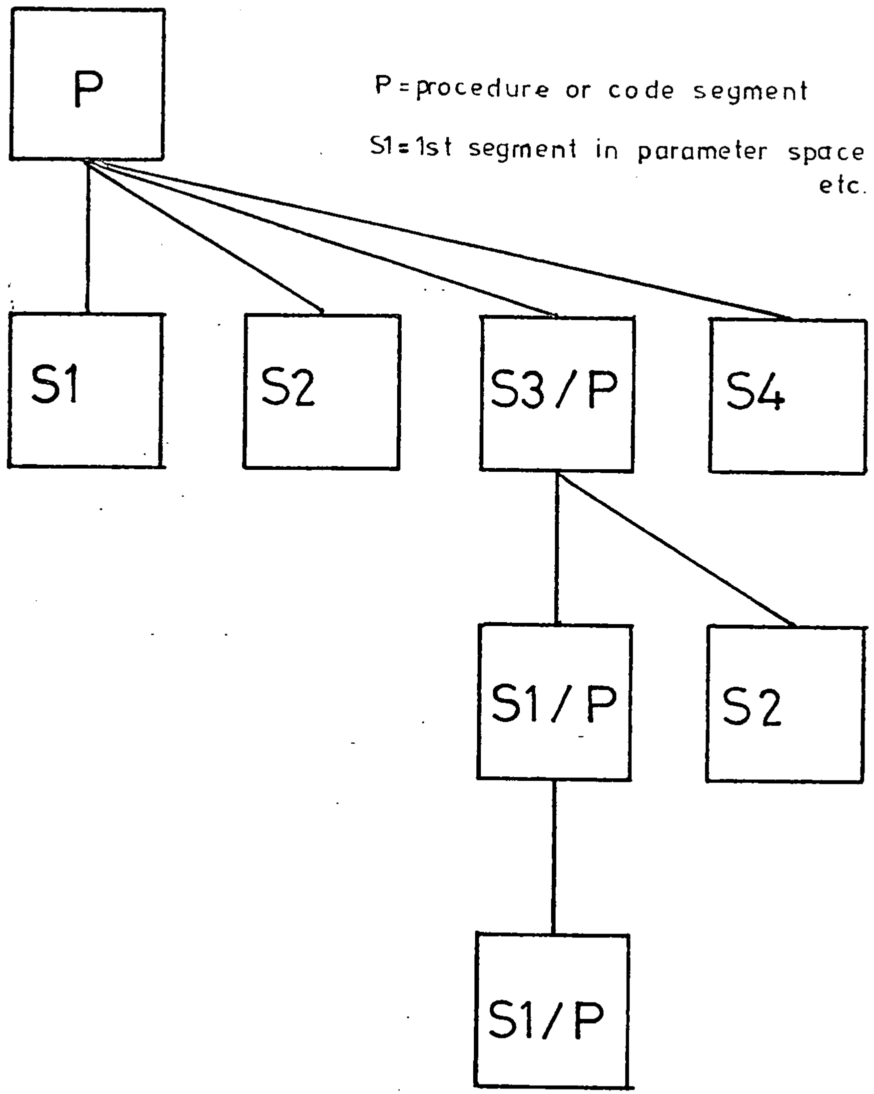
Although Evans and LeClerc [EVAN67] did not use the term 'capability' (using the term 'parameter' instead), they seem to be the first to describe a computation as progressing through different (protection) environments, in each of which the computation possesses different capabilities. They concerned themselves solely with segments and they made a procedure activation, or deactivation, the occasion of altering the environment. When a computation enters a new procedure some (at least) of the segments it accesses will be different. In particular, if we identify each procedure with a separate code segment, then the code segment from which instructions are fetched will be different. Evans and

LeClerc recognised the importance of the code segment in delimiting an environment and called the code segment the 'root' segment of an environment (which they called a parameter space). An environment is defined by an ordered list of capabilities for segments, this list being called a c-list by most writers [DENN66,LAMP71,WULF74,COHE75]. The first capability in the list is for the code segment. The segments referred to by the the other capabilities are of three sorts:-

- fixed: the segment does not change with each entry into the environment
- dummy: a different segment can be used every time the procedure is entered (the conventional parameter)
- scratch: the system will supply a fresh temporary segment for every procedure activation and will reclaim the segment when a return is made from the procedure.

Any of these other segments may be root segments of other environments, leading to a nested structure of environments as depicted in figure 5.1. Any segment may be in many environments simultaneously.

A user formulates addresses by specifying the number in the c-list of the capability for the segment, plus the offset within the segment. Thus programs do not have to worry about segment names or hardware addresses and are not allowed to use them directly. Addresses are taken relative to the current protection environment as defined




An environment hierarchy of Evans and LeClerc.

Figure 5.1

by the c-list. Evans and LeClerc also present mechanisms for addressing items that are in subsidiary environments, so that the whole system structure is not unlike that under a 'rings of protection' regime [GRAH68,SCHR72]. Procedures high up the hierarcny can access everthing lower down.

A procedure call or subroutine call is implemented simply enough, as the address it is desired to transfer to can, and must, be generated in the calling environment. That is, all subroutines that can be called directly from an environment have their code segments as part of that environment. The transmission of arguments is envisaged to be of three kinds:

Entire segments: The calling routine presents the system (kernel or hardware) with a list of segment capability numbers indicating what positions they should occupy in the c-list of the called subroutine. The system makes copies of these capabilities in the new c-list.

Portion of a segment: The calling routine gets the system to create a capability for part of a segment and this is placed in the new c-list.

Individual values: The values of simple variables have to be stored in a stack segment and the capability for this segment passed to the called subroutine.

Unfortunately, procedure or subroutine returns cannot be handled using just an index into the current c-list, because the code segment from which the call originated

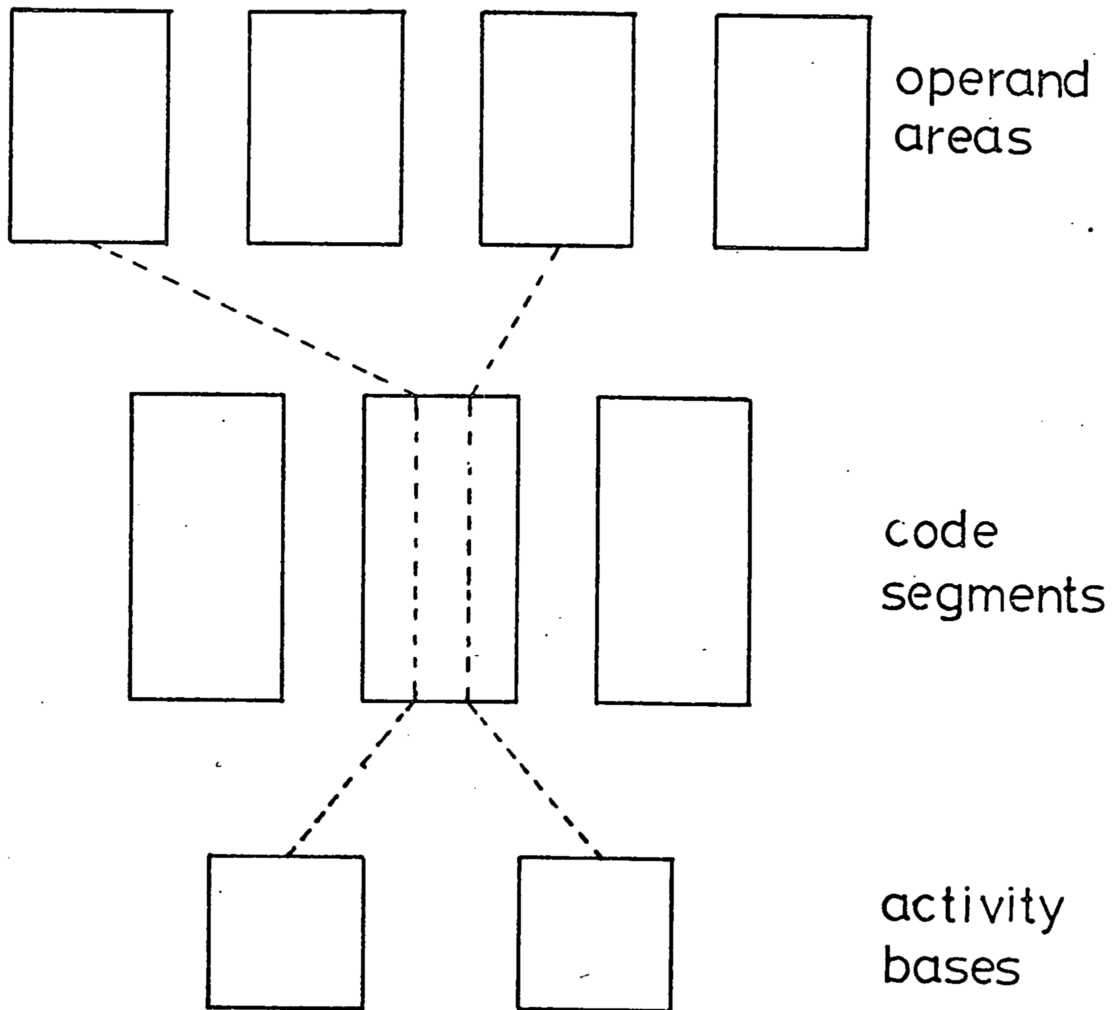
is not likely to be part of the called environment and therefore there is no way for a transfer instruction to formulate the return address. This is where the unique names of the segments should come into operation. Evans and LeClerc use a variation in that they give system wide unique names to every environment. Hence a return link consists of the unique name of the calling environment, which is the same name as the root segment, plus, of course, an offset for that segment. They have then to introduce a protected stack segment attached to each virtual processor to store links.

The application of unique names to environments rather than segments does not appear to be a felicitous choice. By considering every non-root segment in an environment to be a potential root segment of another environment, as Evans and LeClerc do, all segments can be given at least one unique name. Confusion will arise however when the same root segment is part of two different environments. Evans and LeClerc would have been better advised to recognise that a c-list defining an environment can be stored as a segment, and give unique names directly to each segment including the c-list segment.

The Spooner model:

Spooner [SPOU71] also proposes a segment based model and he seems to the first ot use the actual term 'kernel' and define in detail the functions of the kernel. He again attaches great importance to code segments. A code segment defines an operation to be performed by the CPU on an operand area. The same (compound) operation can be performed on different operand areas corresponding to different, but possibly concurrent, activations of the procedure defined by the code segment. Spooner uses the term 'operand area' as he envisages 'windows over core' [SPIE73a], that is segments are permitted to overlap so that the same data item may belong to many segments. However a change of procedure is held to be a possible change of environment and is managed by the kernel.

Spooner introduces a third type of memory area, an activity base. The activity base, as well as providing space for dumping working registers when the virtual processor is suspended, records 'permitted connections', that is capabilities for combinations of code and operand areas (see figure 5.2). These are the forerunners of entry capabilites [NEED72]. Spooner rightly recognises that access rights, or capabilities, should be a function both of the virtual processor and the code it is executing. The possibility exists for a virtual processor to acquire totally new rights when entering a routine, in comparison with the scheme of Evans and




Spooner's model showing 2 possible 'permitted connections'

Figure 5.2

LeClerc where the capabilities are confined to the hierarchy of segments of which the routine is part (figure 5.1).

Spooner also makes the use of protected entry points or gates mandatory. Earlier work described protected entry points for code, the restriction of jumps into the code from other procedures to a number of fixed locations, but the use of them was not thought to be necessary all the time. But without protected entry points no guarantees can be made about the operation of a code segment.

The Spier model:

Spier, working from the ideas of Spooner, and Evans and LeClerc, developed a model for quite a comprehensive protection system [SPEI73a] and also implemented a restricted verson of it [SPEI74]. The following discussion relates primarily to the implemented version, while adopting some of the terminology of the former paper.

Spier identifies five different kinds of memory area (segment). These are:

1) A body of a pure re-entrant procedure. This segment is potentially shareable by all virtual processors. It is called the procedure segment.

2) A protected data base whose information is managed by an associated procedure. Again the single physical copy of this segment is shared by (all) virtual processors. It is called the domain own segment.

3) A working storage area for permanent local values, values preserved from one procedure invocation to the next. These segments are unshareable so that there is one for every procedure that has been executed by each virtual processor. These segments are called incarnation own permanent segments.

4) A temporary segment which contains a virtual processor's execution stack and other temporary variables for the invocation of a procedure. Again not shareable, this segment is called the incarnation own temporary segment.

5) A communication area for transferring parameters between procedures. There is one per virtual processor which is accessible by that virtual processor no matter what procedure it is executing. This segment is called the argument segment.

In the above the importance of a procedure as defining an environment is again seen. A procedure segment, together with its domain own segment, forms the basis of a domain, a 'firewalled' group of segments. A total domain consists of the procedure segment, the domain own

segment and all the incarnation own segments (both permanent and temporary) related to the procedure segment. None of these segments belongs to more that one domain. The argument segment is associated with a virtual processor and is carried along with it as the virtual processor progresses from domain to domain. Thus each argument segment is shared, serially, between domains. Figure 5.3 shows the relationships of segments to two domains and two virtual processors.

A virtual processor always enters a domain by a kernel controlled interdomain jump to a protected entry point, or return point. The set of segments that the virtual processor may access while in the domain is called the domain incarnation for that virtual processor. There are five segments that the virtual processor may access: the procedure segment and the data base or domain own segment of the domain, the two incarnation own segments that relate to both the virtual processor and the domain, and the argument segment. These five segments form the environment of the virtual processor. Until the virtual processor invokes the kernel to change domains it cannot access any other segments. After such a change, it cannot access any of the segments of the original domain incarnation save for the argument segment.

For each virtual processor the kernel maintains an activation area, containing chiefly information about the domain incarnations that the virtual processor is

Virtual processor 1

Virtual processor 2

Domain A procedure and domain own shared by virtual processors 1 & 2

Domain A

Domain B

Argument segment 1 shared by domains A and B

Incarnation A1 own permanent

Incarnation A2 own permanent

Incarnation A1 own temporary

Incarnation A2 own temporary

Incarnation B1 own permanent

Incarnation B2 own permanent

Incarnation B1 own temporary

Incarnation B2 own temporary

Argument segment 2 shared by domains A and B

Domain B procedure and domain own shared by virtual processors 1 & 2

Segment access in
Spier's model

Figure 5.3

permitted to access. This is, in effect, in the form of sets of four capabilities, for the four segments that together with the argument segment constitute each domain incarnation. A current domain pointer indicates the set of capabilities that define the domain incarnation the virtual processor is currently in. The kernel also maintains a hidden stack so that interdomain procedure returns can be controlled. The kernel's action for an interdomain transfer consists essentially of correct handling of the stack and repositioning of the current domain pointer so that the correct environment will be invoked when processing proceeds.

Spier [SPIE74] describes briefly a mechanism whereby a virtual processor's activation area need not contain, at virtual processor initiation time, all the domain incarnation capabilities it will need as a computation proceeds. This involves domains having unique names within the system and being objects in the filing system. When a call to the kernel requests entry to a domain that has not been entered before, the domain procedure and data base segments are copied into active storage (i.e. given hardware addresses). The first time a particular virtual processor requests entry to the domain the incarnation own segments are created in active storage as well as the set of capabilities for the domain incarnation being placed in the virtual processor's activation area. No description of the reverse processes of unloading domains from active storage and removing

capabilities from a virtual processor's activation area is given.

The weakest details of Spier's implementation are a consequence of his having just five segments (one of each kind) per domain incarnation. Firstly, the possibility is denied of segment structure reflecting any underlying divisions of the procedure's variables (other than the permanent/temporary division). Secondly, parameter transmission can become very inefficient. When a few simple items are the only arguments that pass between domains then the overhead of copying these into the argument segment and copying them back again is not too great. But, as Spier's model stands, the accessing of a whole segment's worth of data from more than one domain can be done in one of only three fashions, all unsatisfactory.

1) The data can be made a permanent part of the argument segment thus voiding any claim of confining data to the environments in which it is used.
2) The data can be copied in and out of the argument segment as required.
3) An entry point of the calling procedure can be made available to be used by the called procedure to access items of data as they are required (cf Algol 'thunks'). This involves a domain call/return sequence for every item of data [SPEI73a].

The Cosserat model:

Cosserat [COSS74] proposes a process orientated system where the number of segments accessible to a process is varied. His model is based on an actual hardware architecture, that of the Plessey 250 [COSS72,ENGL72,ENGL74]. Cosserat, following Fabry [FABR74], makes capabilities into data objects which can be copied and overwritten in normal segments by user programs. Cosserat identifies three types of segment:

1) Procedure segments: Cosserat allows his procedure segments to be impure so that they can store data items and/or capabilities for other segments. Thus Cosserat's procedure segment subsumes both the procedure segment and the data base or domain own segment of Spier. The capability for a procedure segment is a type of entry capability. Outwith the procedure the only form of access to the segment is transfer of control to the procedure. When the procedure is being executed then other forms of access are permitted so that data within the segment can be read and written.

2) Data segments: These contain general bit patterns and, as mentioned before, can also contain other capabilities (which the hardware always recognises as such). The capabilities for these segments are freely copiable so that the same segment may belong to more

than one protection environment simultaneously. When a segment is destroyed some (unspecified) procedure has to be carried out to alter all capabilities for that segment to 'null' capabilities.

3) Process base segments: When a process is created (see later) it owns one segment, a special process base segment. This segment contains the capability for the procedure being executed and can contain parameters passed to the procedure. The segment also contains a dump area for temporary storage of working registers by the system and a return link to the calling procedure (see later). The capability for this segment is not explicitly available to the process. It is available to the creating process (with access rights such that the creating process can block and unblock the process but cannot access its data) and it is used by the system in its scheduler table entries.

In Cosserat's model transfers of control to new procedure segments result in the execution of new processes. A GOTO type instruction results in the kernel/hardware placing the capability for the new procedure segment (which the old process must have possessed in order to formulate the address correctly) in a new process base segment and then deleting the old process base segment. Thus processes are truly identified with code sequences. A CALL type transfer results in the creation of a new process base segment

but, this time, the old process base is not de-allocated rather the capability for it, suitably protected, is placed in the new process base. Hence a RETURN instruction can formulate the correct address to which control should be transferred. The newly entered procedure is not allowed to access the calling procedure's process base in any other fashion however. The creation of parallel processes is accomplished using a transfer instruction, but not de-allocating the process base of the creator and not removing it from the the scheduler.

All these forms of transfer of control permit the transfer of parameters. The same convention is used in all cases: a list of data objects (which could include capabilities) in the current process base segment is specified by the appropriate instruction, and these are copied into the target process base. Since a process executing in one procedure segment does not have access directly to other procedure segments or to earlier process bases the only information that can be shared between procedures is that which is pointed to by capabilities embedded in the procedure at compile time, or that which is passed as parameters during a transfer of control.

Cosserat effects the analogue of Spier's incarnation own segments by a modification of the transfer of control mechanism. He allows an 'indirect' transfer, a transfer

to a segment which contains a number of capabilities. One of these capabilites is for the procedure segment, and the rest, which are made accessible to the processes executing the procedure, are capabilities for data segments. If each user accesses the code segment through different 'indirect' segments then the effect of incarnation own segments is achieved.

Thus the following description of general resource handling could be applied to Spier's model if his concept of domain own segment were to be replaced by both domain own segment and domain own resource.

All resources require code to manipulate them and if this code is gathered into a procedure segment then access to the code is equivalent to access to the resource. This is the representation of resources as segments mentioned earlier. Some resources, such as semaphores, can be represented in core so that data space associated with the code is all that is required to make the code segment a resource manager. Other resources, such as line printers, require special I/O instructions to manipulate them and the use of these instructions has to be confined to the code segment that manages the resource. When the control registers for the device are treated by the hardware as memory locations (as in the PDP 11 series and the Plessey 250) then this confinement can be achieved using the capability mechanism unmodified.

For many types of resource a data area for each user of the resource has to be kept. This area could contain buffer space and/or status information such as, in the case of a filehandler, the names of the files currently opened by the user. Management of this information is facilitated by keeping it in separate segments, separate both from the common data and from the information related to other users. This is the function that the 'indirect' entry segments serve. This method of resource management has been successfully used on the Plessey 250.

Other Protection Schemes:

We have not dealt with all the protection schemes that have been proposed, concentrating on those that emphasise the code segment as the basic unit. Our chief omissions are CAP and HYDRA, both of which are being implemented, and the Chicago Magic Number Computer, CAL and SUE, the implementations of which were terminated prematurely.

CAP is a machine with special capability manipulation hardware being developed at Cambridge [NEED72,NEED74]. The main objects in the system are segments and processes and it is similar in many ways to the model of Cosserat. However the concept of a domain own segment, in Spier's terminology, that is a shareable data base, does not exist. Further the system is formulated in the context

of a hierarchy of processes and the accessing of all capabilities through indirection tables to a master capability segment. A master process in the process hierarchy can treat its slave processes' capabilities as simple data. This produces reliability compared with single level capability systems [LAMP74] but we feel it is too general a mechanism to be incorporated in a distributed system.

The Chicago Magic Number Computer [FABR74] was the first attempt at incorporating capabilities into a hardware architecture. Capabilities were for a single type namely those for segments. The resulting machine would, it seems, have been similar to the Plessey 250 but less efficient in its handling of alterations to capabilities when segments change location.

The HYDRA system [WULF74,WULF75b,LEVI75,COHE75] is being mounted on the multiprocessor C.mmp machine. It allows an unlimited set of types of capability. Every object in the system, not just processes or domains, has an associated c-list so that arbitrarily complex objects can be built up. HYDRA is also process orientated allowing for the dynamic creation of processes. One type of object in HYDRA is the procedure. Entering a procedure involves the creation of a new protection environment.

CAL [LAMP76] was an attempt to put a capability based system on a Control Data 6400 computer. It again had a multiplicity of types of object but c-lists belonged only to domains, and it would appear that the number of virtual processors in the system was fixed. The domains in this system were rather static objects compared with the equivalent in HYDRA. Domains existed for long periods of time so that a change of procedure involves a change of domain rather than the creation of a new domain.

Project SUE was to result in a capability based operating system for an IBM 360 computer [SEVC72,SEVC74]. Again the system had many types of objects, and capabilities for them, and it was also organized to support hierarchical processes. Processes were created with an environment that basically did not change. All resources were handed out along the arcs of the process creation tree. Capabilities for the resources were considerably extended from the three field sort described earlier, to contain five fields including a count field. In SUE a capability not only gave access to a resource or object but specified how many times or how much of it would be accessed. It is interesting to note that the nucleus of an operating system for SUE was provided with about 10 processes, that is 10 protection environments, together with the kernel.

Summary:

The use of a code segment, containing a procedure or group of related procedures, to define a protection environment or domain, pervades almost all the models we have mentioned. The remaining contents of the domain vary from model to model but the idea of global data, accessible to all virtual processors that enter the domain, and local data that is different for each processor entering the domain, is common to several models. The models also differ in the latitude given to segments to belong to more than one environment simultaneously, or even sequentially. As we implied at the beginning of this chapter, in a distributed system the less sharing there is of segments the easier management of domains is likely to be.

The entering of a new procedure is usually the occasion of changing a virtual processor's protection environment. Details of this change of domain vary from model to model, particularly in regard to parameters passed to the new environment. We have indicated that some movement of segments as parameters to new domains is essential if gross inefficiencies are to be avoided. This, of course, conflicts with our desire to have no sharing of segments between domains.

The fact that the domain is the key concept in all the proposals and systems we have mentioned in this chapter,

supports our assertion that the domain can be considered the paramount structure in computer systems. However with the successful implementations of capability based systems being so thin on the ground the kernel/domain architecture could hardly be called an established technology. But we believe it to be a viable architecture and in the next chapter we propose a kernel/domain architecture suited to distributed systems. Our enthusiasm for the kernel/domain architecture is tempered, we admit, by one consideration. The size of domains is a question of vital importance to us, and one on which there has been no published information. Both CAP and the Plessey 250 use hardware to effect the interdomain jump so that the overheads of using small domains, with frequent domain changes, are small. HYDRA has a software scheme to handle interdomain jumps and so requires largish domains, as a distributed system probably will, to avoid the interdomain jump overheads swamping the useful computation. As yet there has been no indication as to whether the HYDRA implementors have succeeded in generating large domains. We return to this question in chapter 7 when we look at how programming language structures relate to domains.

# CHAPTER 6

# OUR MODEL

In this chapter we propose a kernel/domain architecture suited to a distributed system. First we give a brief discription of its structure and then give a detailed description of how the model could be implemented using capabilities. Finally we relate our model to those discussed in the previous chapter.

## SECTION 1: THE BASIC COMPONENTS.

The purpose of all the models we have looked at in the previous chapter has been to enhance systems reliability, both by enforcing the run time protection rules of the model and, as Spooner and Spier stress, by the modular structure of software resulting from the design of domains or protection environments. While we do not want to dispense with these aids, our chief reason for wanting a computation divided into a sequence of incarnations of different domains is to allow the computation to be performed at different sites in a distributed system

when, for resource utilization reasons, this is desirable.

Much of the thrust of recent research on capabilities and domains has been to generalize their properties to cover every conceivable type of computational requirement. By analogy with the full flexibility of the Von Neuman architecture often being restricted (without real loss of function) in order to achieve efficiency [BURR61,FEUS73,DORA75], we have sought a minimal set of capability and domain properties that can be realized efficiently in a distributed system and at the same time cover normal computational requirements. Because Spier's model had the highly desirable property, for us, that no segment ever belongs to more than one domain at once, it made a good starting point for the model we have developed to meet the requirements for distributed systems. We give a concise description of the model before expanding on its features and giving a justification for them.

The basic components of the model are:

1) A reference space of segments, spread over and interchangeable between a number of sites.
2) A number of virtual processors; relationships or associations, alterable in time, exist between virtual processors and segments (and also between segments themselves).
3) A kernel, a software extension of the basic machine,

exists at each site and manages virtual processors and segments. Kernels communicate with one another to effect this management. All transfers of segments between sites are performed by kernels.

A segment that contains pure re-entrant code is called a code segment and is potentially executable by all virtual processors. A code segment forms the basis of a domain, together with an associated public (data) segment, if one exists. A public segment consists of data that is usable by all virtual processors, but only when they are executing the related code segment. Local segments constitute the rest of a domain. There are local segments associated with every virtual processor that executes (the code) in the domain. They hold data relevant only to the associated virtual processor. A domain, then, is a group of segments which cannot be accessed by any virtual processor not executing the code segment of the domain. The entry to and exit from domains by virtual processors is carefully controlled by kernels. At the time of virtual processor entry or exit, local segments related to the virtual processor (parameters) may be transferred between domains.

For each virtual processor there exists an associated segment, the processor base segment, which is accessible to only that virtual processor. It is accessible at all times, no matter which domain the virtual processor is executing in. This processor base segment, the code

segment and the public segment of the domain the virtual processor is in, and the related local segments together form a domain incarnation (see figure 6.1). It is a sufficient condition for a virtual processor to proceed, on entering a domain, if all the components of the domain incarnation are at the same site. A domain that does not have a public segment is called a pure (code) domain. A domain incarnation of a pure domain may include a copy of the associated code segment rather than the code segment itself. A domain that does have a public segment is called a monitor (see later).

The Entry Capability:

Capabilities are the normal mechanism used to implement domains. Cosserat's model assumes a tagged architecture [FEUS73] and allows capabilities for segments to reside in normal data segments. However, the implementation from which his model was derived, on the non-tagged architecture of the Plessey 250, insists that capabilities reside in separate segments from data so that appropriate protection of capabilities can be applied [ENGL74]. Either approach means that at a change of domain the identity of all the segments that belong to the new environment is not immediately obvious. When capabilities are kept in separate segments a tree scanning operation is required to determine all segments in the environment. For Cosserat's model the problem is

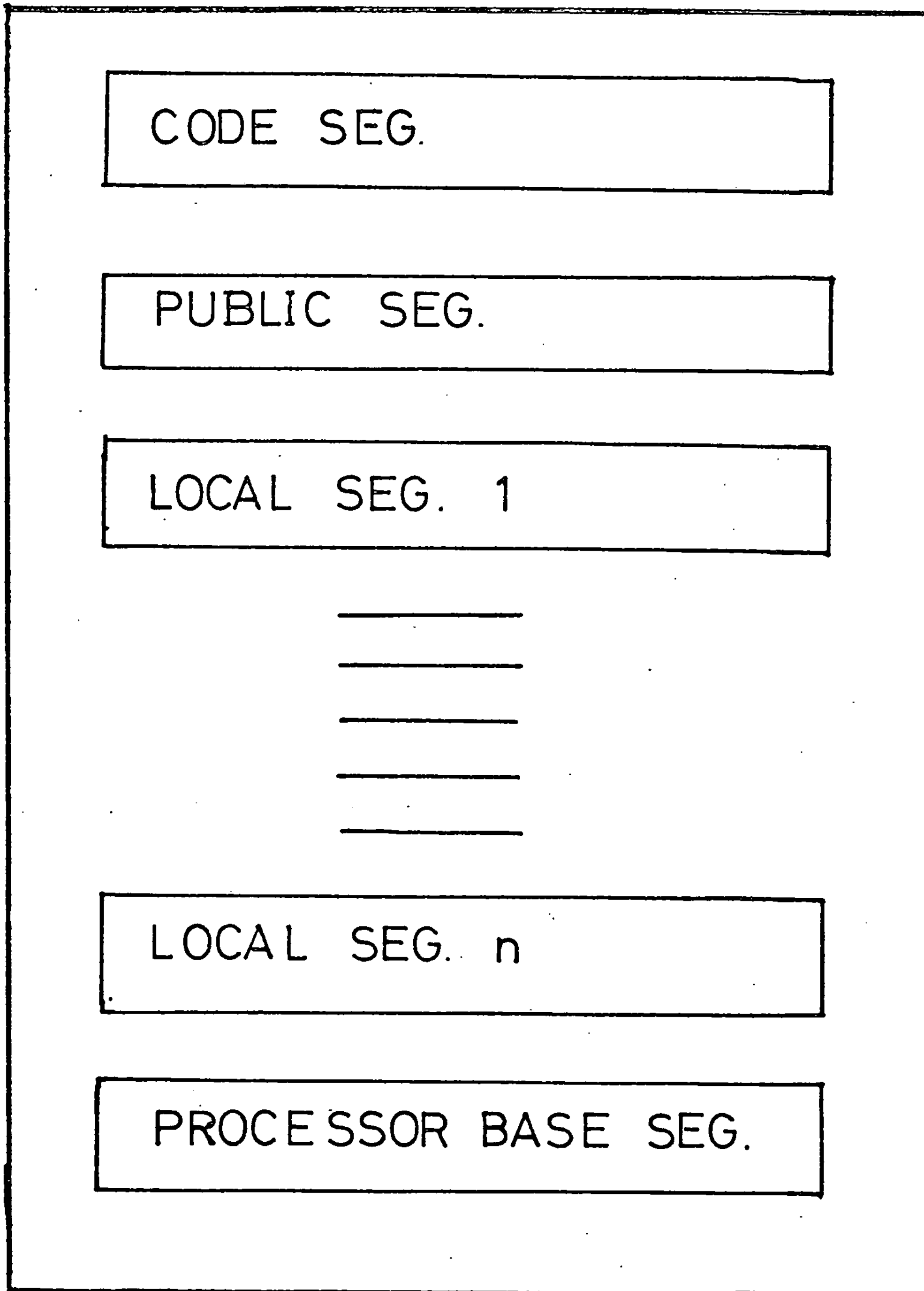



Figure 6.1

an order of magnitude worse; every segment must be systematically searched to make sure that no possible branch in the tree structured environment is overlooked.

Only if we did not require to know what segments constitute the new environment at the time of a domain change could we use Cosserat's scheme (if we had a tagged architecture) or allow some local segments to contain capabilities only. But since, as we will explain, it is necessary to know what segments constitute a new domain incarnation, we have to forego the not inconsiderable advantages of list structured addressing [FABR74]. This necessity arises in a distributed system because space has to be allocated for segments of a domain incarnation that are not at the site chosen for the domain incarnation, and these segments have to be brought to that site. This can be done at the time of domain entry (pre-loading) or the first time a capability is used in the new domain (demand loading). In paging systems pre-loading pages from backing store has been shown to involve less overheads than demand paging [ADAM75]. As we show later, fetching a segment from another site is likely to involve almost as much work as fetching a group of segments together so that a similar trait with respect to segments is likely in distributed systems. Accordingly in our model the capabilities for all the segments that will be involved in a new domain incarnation are placed in a single list, so that they can be quickly scanned to determine the requirements for the

new environment. This list of capabilities for the segments making up a domain incarnation is called an entry capability.

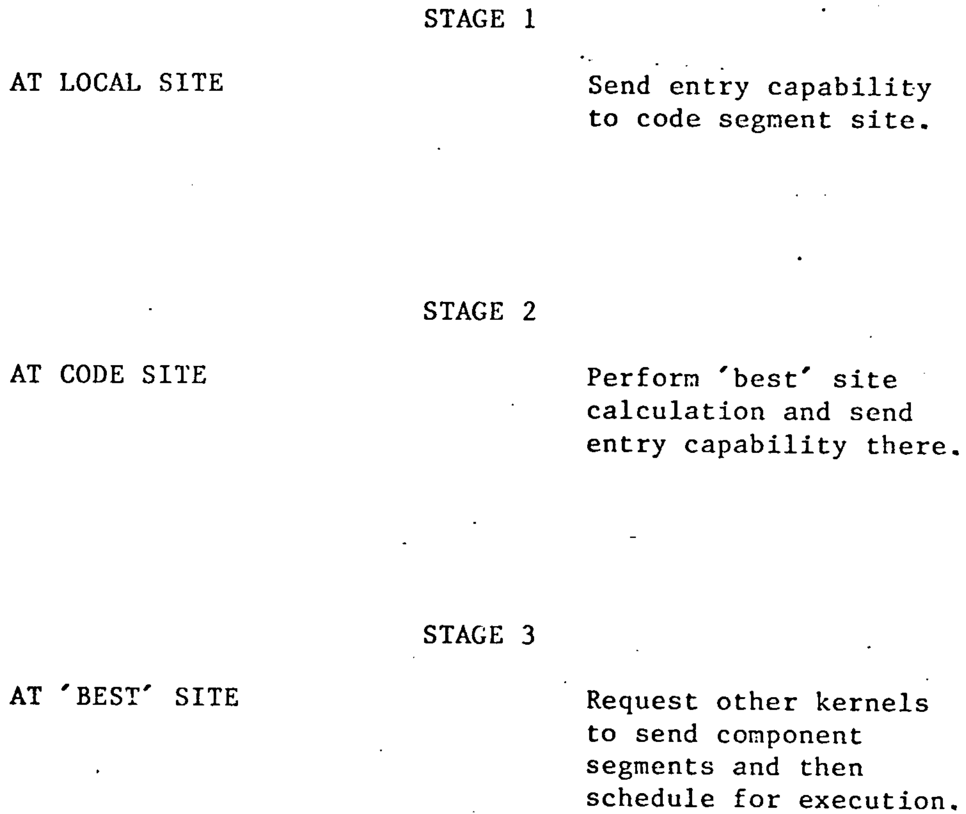
We have found the entry capability to be a very useful concept. The interdomain jump can be thought of as a validation of the entry capability for a new domain incarnation. As its last action in the old domain a virtual processor places the capabilities for the segments of the new domain incarnation into a list (details about how this is done are given later on). We call this list a c-list. The virtual processor calls the local kernel passing it the c-list it has just constructed. The kernel scans all the capabilities on the new c-list and if they all refer to segments resident at that site it (normally) will mark the c-list as a valid entry capability and place it in a queue of ready to run domain incarnations. If all the segments are not resident at the site then the c-list is sent to the kernel where the code segment resides. This kernel calculates what it considers to be the 'best' site for the domain incarntion to take place at, and passes the c-list, suitably marked, to this site. The kernel at the 'best' site could decide that it did not want the incarnation at its site in which case it passes on the c-list to another site, but generally it will accept the c-list, go through it, and request kernels that have the segments in the c-list to send those segments to its site. When all the segments have arrived at its site the

kernel marks the c-list as a valid entry capability and schedules the domain incarnation for execution (see figure 6.2).

The above is a very skimpy description, but it does show the importance of the entry capability in defining the domain incarnation in a compact form. The interdomain jump involves up to three scans, at different sites, of all capabilities for the domain incarnation. This shows the infeasibility of having a more general distribution of segment capabilities if preloading of segments is to take place. Before we give details on how segment capabilities are initially placed in c-lists we discuss the differences between the management of capabilities for code and public segments, processor base segments and local segments.

Local segments:

We mentioned that where capabilities were freely copiable the deletion, or even change of address, of a segment required that all copies of the capability be altered. In practice, in single site systems, all such capabilities are pointers to a master table [NEED74,ENGL74] so that all that is required is the alteration of the master entry to reflect the new situation, coupled perhaps with a usage count so that the master entry can be dispensed with when no capabilities point to it. In distributed systems we are, of course,




BASIC ACTIONS PERFORMED FOR AN INTERDOMAIN JUMP

Figure 6.2

denied the luxury of a central master table of capabilities. For local segments a master table could be kept in each processor base segment but this would complicate any interprocess communication involving the passing of segments. Hence we make capabilities for all local segments 'transfer only' [GRAH72]. This is a step better than Spier's completely static, no transfer, scheme for his incarnation segments. Only one capability for each local segment exists and this is passed from environment to environment as required. The only time a local segment is shifted between sites is at the time of domain incarnation entry. This is when the kernels have the entry capability list and so can easily modify the information in the relevant capability.

A entry capability may sometimes hold pseudo capabilities instead of capabilities for local segments. These are of the following types:

Transient: This pseudo capability just specifies a length so that the system can create a scratch segment for an incarnation at the site chosen for the incarnation to take place. When the segment is created a genuine capability replaces the transient one.

Null: To facilitate the transmission of parameters between domain incarnations a slot in an entry capability may be empty.

On disk: The segment is in a disk buffer.

Descriptor: Used when a segment is not intended to be accessed by the current domain but to be passed on to another domain.

The last two types have been introduced for reasons of operational efficiency and will not be mentioned again in this chapter.

Code and oublic segments:

Domains, as we have already stated, can be identified with their code and public segments. Entering a domain implies execution of the code in the code segment. Logically many virtual processors can be executing in a domain simultaneously (although, when there is one physical orocessor per site and only one copy of the code, only one virtual processor can be progressing through the code). Thus the code and public segments can form part of many different environments at the same time. We cannot make rules which would restrict these segments to single domain incarnations and not, at the same time, so emasculate the distributed system as to make it useless. Therefore to handle code and public segments a distributed equivalent for a master table of capabilities is required. We make the code and public segments of domains global objects (as discussed in chapter 3), the only ones in our system. We assume that at any time the kernels in a system can between them locate the code and public segment of a domain. Chapter

3 detailed how this might be done. This means that domains must have system wide unique names and every program must be appraised of the names of the domains it wishes (is permitted) to enter. Thus a capability for code and public segments is simply a name, it does not have any address information. The kernels have to translate this name into an address.

Processor Base Segment:

The management of the processor base segment is the easiest of the three types of segment. Its capability need never be made explicitly available to a user, nor does it make sense for a processor base segment to be simultaneously part of more than one environment. When a kernel is requested to perform an interdomain jump it can take the processor base segment from the entry capability of the requesting domain incarnation and place it in the new entry capability. The kernel may have to modify the processor base segment itself to fix up return links.

Spier's model uses the argument segment simply for carrying parameters between domain incarnations. Information about virtual processors is held in a special area in the kernel. Cosserat holds this information in his process base segment. This is the solution we prefer as then the information moves from site to site as the virtual processor moves from site to site. The processor

base segment in our model is thus slightly anomalous in construction, consisting of quite separate sub-segments. These sub-segments contain:

1) simple variable values being passed as interdomain parameters
2) entry capabilities for the domains that the virtual processor has entered
3) other information about domains which the virtual processor is permitted to enter
4) general management information, e.g. scheduling parameters and accumulated run time.

## SECTION 2: ENTRY CAPABILITY STRUCTURE AND MANAGEMENT.

We have detailed how once a putative entry capability (c-list) is presented to a kernel, the kernels go about gathering all the segments together and schedule the execution of the new domain incarnation. We now look at the process of creating the entry capability list in the first place. Spier's distinction of two types of local segment, incarnation own permanent and incarnation own temporary, gives a starting point for identifying the mechanisms required. Since we permit local segments to pass as parameters between domain incarnations we require three categories of local segments: temporary, permanent

and argument. We suppose that the local segment capabilities in an entry capability each belong to one of three sublists:

the temporary list or T-list

the permanent list or P-list

the argument list or A-list

Stack organization of entry capabilities:

Consider first a system in which all local segments are of the temporary type. On a computation's entry to a domain the local segments required are created. They exist while the computation proceeds in the domain and while calls are made to other 'inner' domains, to which they may be passed as parameters. They are deleted when the computation exits from the domain. For such a system it is appropriate that skeleton c-lists be kept in a stack in the processor base segment.

To enter a new domain a virtual processor executes the code in the old domain to cause the name of the new domain, that is the name of the code and possible public segments, to be placed in a new c-list which will eventually be placed at the top of the stack of entry capabilities. This name will normally have been embedded in the code at compile time but exceptionally could have been passed as a parameter to the old domain incarnation. The desired entry point is also stored with the domain

name.

We could use compile time information and have the code of the old domain specify pseudo capabilities (of the transient type) for the local segments that are to be created for the new domain incarnation. The alternative is to have a template [COHE75] associated with the code segment and have the kernel at the code site create the pseudo capabilities before it does its 'best' site calculation. This second alternative is to be preferred because the data about the internal structure of a domain is held in just one place, which is in accordance with the principle of information hiding [PARN72], and it leads to less duplication of code.

Parameter handling:

In the case where the domain name is known at compile time then the number and type of any parameters taken by the domain can also be specified at compile time. When these parameters are simple variables they can be loaded into an argument stack or area in the processor base segment.

When the parameters are for local segments (which must form part of the old domain incarnation) there are two approaches that can be taken.

1) Domains can be permitted to shift capabilities between

the T-list and A-list. The code in the old domain can specify the transfer of the segment's capabilities from the T-list of the old domain incarnation to its A-list. The local kernel then transfers the old A-list to the A-list of the new entry capability when invoked to perform the interdomain jump.

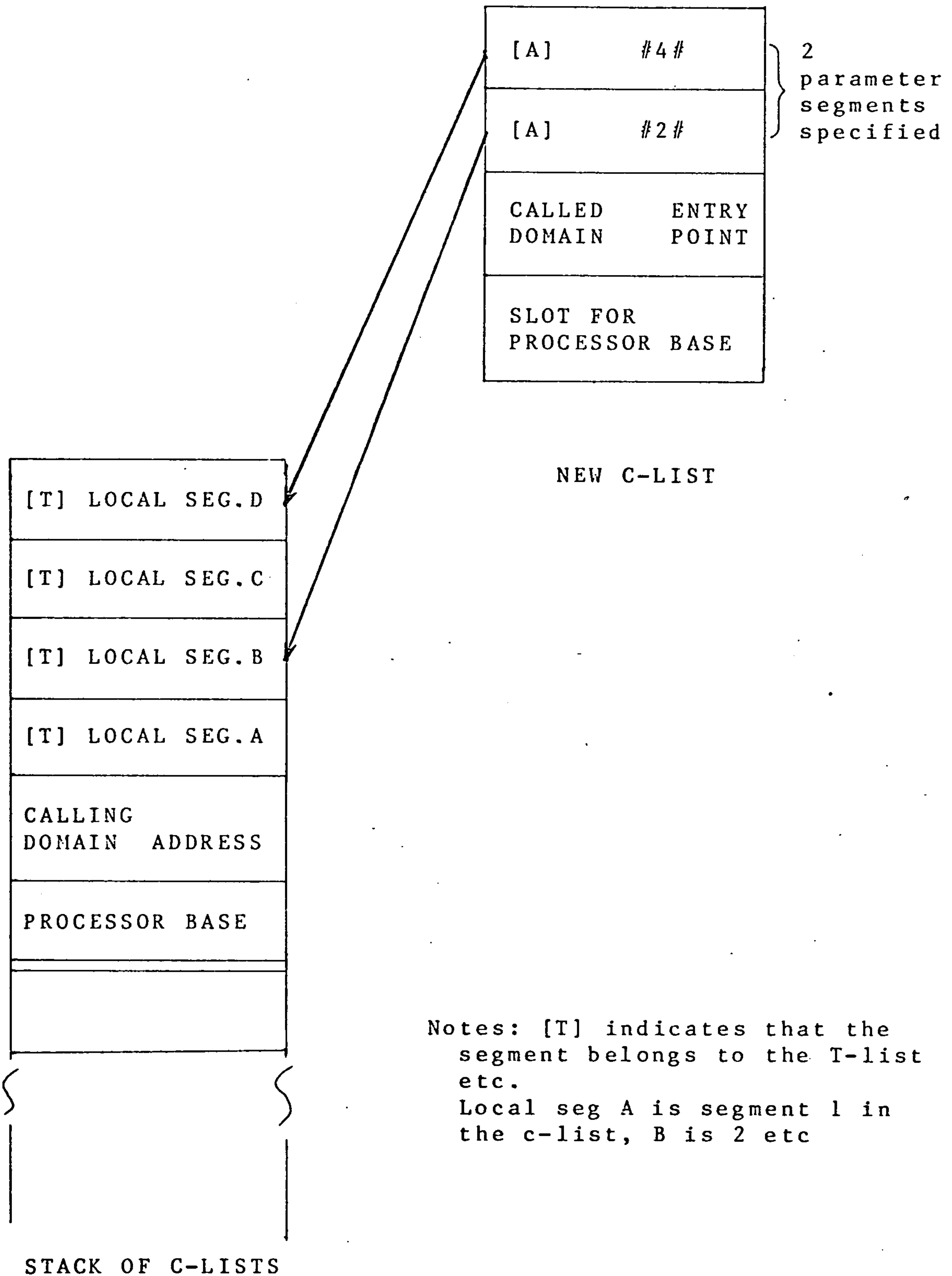
2) Only the kernel is permitted to manipulate capabilities and it transfers the entries direct to the new A-list. In this case the code places pointers in the new A-list back to entries in the full list of local segments in the old entry capability (see figure 6.3). When the interdomain jump request is made the local kernel can transfer the 'pointed at' capabilities to the new A-list, noting in the old c-list to where they were transferred (see figure 6.4).

The former of these two approaches is the more flexible but is likely to be less efficient and less secure. For in this approach the A-list becomes simply a receptacle for parameter capabilities at the time of interdomain jumps. The entered domain has to transfer the capabilities for the parameters back to a T-list before it can safely access them. Although it saves the kernel a job, this transferring of capabilities back and forth between T-list and A-list could be error prone. Hence we prefer the second approach which leads to more compact entry capabilities at the cost of slightly more work done by the kernel at domain call and return time.

Then the A-list always contains the parameters passed to the domain. It has no special relation to parameters passed from the domain to inner domains.

When the name of the new domain is not known at compile time, the same action as above can be taken if some form of parameter specification has been given (and checked) at compile time. Otherwise the kernel can accept the parameters as given but, before permitting entry to the new domain, it would have to perform a check to ensure that they corresponded to those expected by the new domain. Such a dynamic check could turn out to be both more costly [HANS74] and coarser [HANS73] than one provided at compile time.

Figure 6.3 gives an example of the old and new c-lists just before the local kernel is invoked to perform the interdomain call. Figure 6.4 shows the transfer of capabilities made by the local kernel before it sends the c-list off to the site of the code segment (or deals with it itself if the new code segment is already resident at its site). Figure 6.5 shows the situation just prior to the 'best' site calculation and figure 6.6 gives the final form of the entry capability stack when the called domain incarnation is ready to run.




START OF INTERDOMAIN CALL
Figure 6.3

| NEW C-LIST |
|---|
| [A] LOCAL SEG.D |
| [A] LOCAL SEG.B |
| CALLED DOMAIN ENTRY POINT |
| PROCESSOR BASE |

NEW C-LIST

| STACK OF C-LISTS |
|---|
| [T] #2# |
| [T] LOCAL SEG.C |
| [T] #1# |
| [T] LOCAL SEG.A |
| CALLING DOMAIN / RETURN ADDRESS |
| SLOT FOR PROCESSOR BASE |

STACK OF C-LISTS

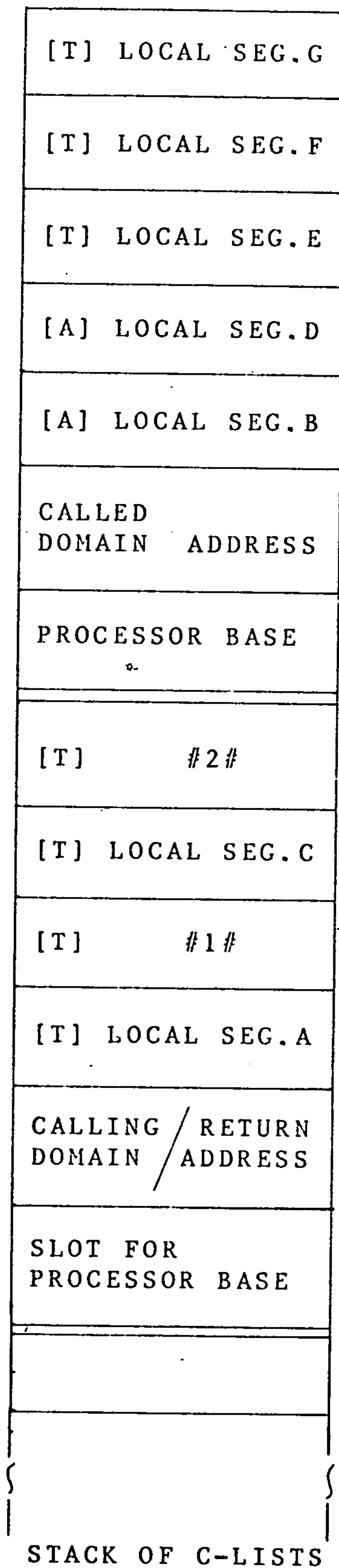
TRANSFER OF CAPABILITIES
TO NEW C-LIST
Figure 6.4

| [T] size=100 |
| --- |
| [T] size=267 |
| [T] size=900 |
| [A] LOCAL SEG.D |
| [A] LOCAL SEG.B |
| CALLED DOMAIN / ENTRY ADDRESS |
| PROCESSOR BASE |

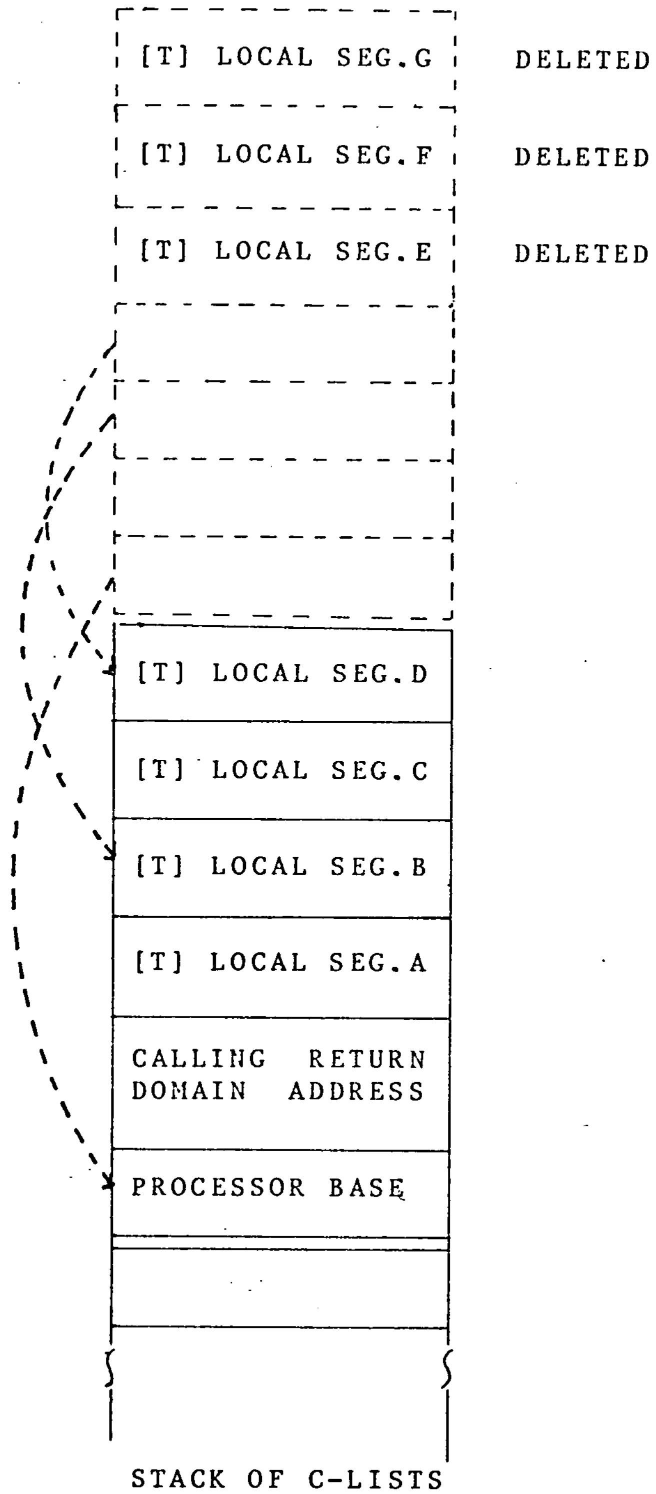
NEW C-LIST

NEW C-LIST AFTER
INFORMATION FROM
CODE SEGMENT TEMPLATE
HAS BEEN ADDED

Figure 6.5




COMPLETED INTERDOMAIN CALL
Figure 6.6




TRANSFER OF CAPABILITIES
AT INTERDOMAIN RETURN
Figure 6.7

Other interdomain jumps:

A return to a domain is also performed by an interdomain jump. If the slots where parameters were passed to were noted (see figure 6.4) then the top two entry capabilities in the entry capability stack contain all the information the kernels require to effect a return. (Except that any simple values to be returned must be placed in the parameter area). When the interdomain jump is requested the local kernel shifts back the processor base segment and all the parameter segment's capabilities (i.e those in the A-list) to the c-list for the domain incarnation being returned to. Then the kernel deletes all segments whose capabilities are in the T-list. The entry capability for the domain being returned to is then validated as before except that there is no requirement for the code segment to supply details of the structure as this is known already. Figure 6.7 depicts the movement of capabilities effected by the local kernel when requested to perform an interdomain return.

Unfortunately not all computations proceed in the nested fashion mirrored by a stack implementation. One simple example of this is where a computation moves serially through domains. If the first domain in the sequence was passed parameters in a normal call, then all domains in the sequence must maintain the same parameters so that the final domain in the sequence can perform a

correct return. Within the sequence of domains a virtual processor wishing to jump to the next domain loads the name (and entry point of the domain) into a new c-list and requests an interdomain jump. The kernel then copies the processor base capability and the capabilities in the called domain incarnation's A-list into the c-list and dispatches the c-list to the code segment site. It also deletes all the local segments whose capabilities remain in the old top of stack entry T-list and removes this entry capability from the stack. The rest of the interdomain jump proceeds as before.

Retaining permanent segments between calls:

Greatèr complications arise when it is desired to retain local segments between calls on the domain. This arises when, for example, there is a co-routine structure between two or more domains, or generally in the handling of peripherals which requires the maintenance of buffers and status information. Although it is quite straight forward to devise rules for kernels to know when t place local segment capabilities in the P-list, so that they will not be deleted at domain exit time, it is more difficult to devise satisfactory rules for kernels to know when to eventually delete segments in the P-list. Consequently we allow a virtual processor to move any local segment in its current domain incarnation between the P-list and the T-list. This gives a greater

flexibility than could be achieved by automatic rules. It shifts the responsibility for deleting permanent segments to the programmer.

In order to use the local segments whose capabilities are stored in the P-list when a domain is re-entered again, the P-list, or the whole c-list with appropriate empty slots, has to be preserved when the entry capability is removed from the top of the stack. The following is a list of options available:

1) Abandon the stack of c-lists altogether, keeping a simple table of c-lists for all domains entered or known about, and maintain a separate stack of return links and a pointer to the current domain incarnation. With appropriate organisation this gives quick access to the c-list, which includes at least the P-list, of any domain. This is the approach taken by Spier [SPIE74]. It restricts all entry points of the domain to taking the same number of argument segments and using the same number of temporary segments (although null segments could be used sometimes). Further it does not allow recursion of any form.

2) Maintain the stack but use another area of the processor base segment for storing P-lists of exited domain incarnations. Together with each P-list, an indication must be kept of the domain to which it belongs. For every call on a domain this area has to be checked to see whether there are local permanent segments for the new domain or not. Also for domains

with multiple entry points the sequence of calling these has to be controlled, or all entry points must use the same structure of permanent local segments. Recursion using the same instance of permanent segments is possible.

3) A variation on the previous option is to store the P-list with the code segment templates and maintain separate templates for every virtual processor that has previously entered the domain. This could provide more flexibility than option 2 in the arrangement of P-lists for different entry points but otherwise the properties of the two options are similar. But this is not a good solution. The altering of data associated with the code segments inhibits the duplication of code at different sites. Also error recovery is made more difficult; information about the resources a virtual processor has (defined by the permanent local segments it owns), is spread throughout the distributed system rather than being concentrated in the processor base segment (which will always be at the scene of any error).

4) Change the stack to a tree arrangement similar to that used in quasi-parallel programming systems [DAHL72] and some forms of parallel processing [ORGA73]. When, at the first domain exit time, the kernel detects that the P-list is not empty it 'splits' the stack. The c-list being exited from is linked, by the kernel, into the tree as a sibling of the original calling domain. Provided that kernels can distinguish first

time entry requests from re-entry requests, full dynamic recursion is possible. In the case of re-entry, a search up the tree may be required to locate the correct c-list. Again care will be required with multiple entry point domains. This option also permits an obvious rule for the ultimate deletion of permanent segments, namely deletion is performed and the tree pruned, when control returns to the parent domain incarnation. But this option confines interdomain calls to conform to a hierarchical structure. Also if two or more P-lists for the same domain incarnation exist with the same parent then some further mechanism is required to identify which P-list is to be used in a domain incarnation.

5) Retain the stack and introduce labels, that is names for c-lists. These named c-lists are stored in a separate area in the processor base segment and contain at least the domain name and entry address, and incarnation P-list. They could also contain slots for parameters and pseudo capabilities for the temporary segments (see figure 6.8). These c-lists are put together by a kernel request during the virtual processor's execution in the domain to which the segments pointed to by the c-list belong. The kernel returns a label which can be passed as a simple parameter. The main use of this is to preserve the P-list of a called domain. Before it exits the virtual processor in this domain sets up the label

| |
|---|
| [T] size=900 |
| [T] size=200 |
| [P] PERMANENT LOCAL SEG. |
| [A] SLOT FOR PARAMETER |
| [A] SLOT FOR PARAMETER |
| DOMAIN ENTRY NAME ADDRESS |
| SLOT FOR PROCESSOR BASE |

EXAMPLE OF A
LABEL C-LIST

Figure 6.8

c-list and then returns with the label to the caller. For subsequent re-entry the caller requests an interdomain jump to the label. The local kernel retrieves the c-list and validates it as usual. The order of calls to multiple entry points is dictated by the called domain by way of the labels it returns at the end of each call. The label mechanism can also be used to implement 'call by name' parameter passing, but this is not something to be encouraged in a distributed system. As the connotations of label would suggest, this option is rather primitive. But it can permit full recursion as well as distinguishing easily between multiple uses of the same domain by the same virtual processor. To enter a domain a first time a virtual processor presents an initial entry capability; for subsequent re-entry it presents the label returned from the previous entry. If the virtual processor wants to use the same domain for a different purpose (e.g. if it is a file handler domain and the virtual processor wants to open a second file) then it presents an initial entry capability again and will be returned a new label. This is the only option proposed so far that can handle this multiple use situation.

6) Abandon the concept of permanent local segments altogether. Instead generate temporary local segments at an outer level and pass them as parameters through all inner levels to the domain that requires to use them. Also, the public segment of the domain could be

used to hold information that would otherwise have been kept in permanent local segments. This solution,as well as violating protection principles, could involve a huge increase in the number and/or size of segments that would have to be shifted from site to site at each interdomain jump.

We feel that the fifth option is the best. It could turn out in practice though that the features this option provides are not required, that permanent segments form such a tiny fraction of the total number of segments that the second or sixth arrangements would be better.

Creating and deleting local segments:

One other topic concerning local segments is their creation and deletion during execution within a domain. We permit virtual processors to make kernel calls to delete local segments in the domain they are in, at any time. The capability in the c-list is replaced by a null capability. A null capability may be passed as a parameter. This is particularly appropriate in a producer/consumer situation; the producer transmits a full segment to the consumer as a parameter in an interdomain call. At the return no useful purpose is served by transmitting back the segment so the consumer can delete the segment when it has finished with it.

The creation of local segments is more difficult as it involves the allocation of a resource, namely memory space, so it could be subject to delays or even the shifting of the domain incarnation to another site. The best time to create new segments is at domain entry time. This is why we provide templates attached to the code segments so that space requirements for a new domain incarnation can be determined before the 'best' site calculation is performed. If it is absolutely necessary for a domain to be able to create segments once its execution has begun, then its request to the kernel to do this is treated as an interdomain jump back to itself. We assume that capabilities for newly created segments belong originally to the T-list.

## SECTION 3: COMPARISONS.

In this chapter we have proposed a domain architecture suitable for distributed systems. We have detailed how, despite the fact that no copying of capabilities is allowed, a quite powerful capability system can be constructed. Our system does not suffer from revocation of capability problems because 1) it is not process orientated and 2) only one capability exists for each segment (other than code and public segments). If

present domain systems are viable on single site systems then our system is powerful enough to be viable on a distributed system. We now identify common points between those domain systems we discussed in the last chapter and our model, and show where improvements have been made.

Evans and LeClerc identified three types of local segment making up a domain; fixed, dummy and scratch. These correspond to the segments whose capabilities are kept in our P-list, A-list and T-list respectively. By allowing segments to be moved between P-list and T-list we cater for domain initialization and allow more flexible deletion of segments.

We have already mentioned what we consider to be the main inadequacy of Spier's model, the fixed number and type of segments in a domain incarnation. Our model allows any number of local segments, the equivalent of Spier's incarnation own temporary segments. We permit local segments to be passed as parameters between domain incarnations. This eliminates much of the potential inefficiency of Spier's model arising from copying whole segments into and out of the processor base (argument) segment. We are also far more flexible in our handling of permanent segments. Using labelled entry capabilities, our scheme will support a virtual processor having two or more different sets of segments in a domain.

There is little to choose between Spier's separation of code and public (domain own) segments, which we have adopted, and having them mixed together as Cosserat allows, provided that a domain which does not have any public data is identifiable as such. In our previous discussion above, and our subsequent discussion of the implementation of our model, we always treat the two segments, when they both exist, as a single entity. If however a system had plenty of active storage but was lacking in communication bandwidth it is conceivable that the code segment would be treated differently from the public segment; copies of the code segment being permitted. There is no point in having copies of public segments because the machinations required to keep them consistent would far outweigh any advantage gained in not having to shift segments around from site to site to form domain incarnations.

We were not aware of Cosserat's work when we undertook the definition of our model, working, as we mentioned before, more from the papers of Spier. There are however quite a few points of similarity between Cosserat's model and ours. Both permit any number of segments to be part of a domain incarnation. Both use the processor base segment for several purposes. Although in Cosserat's model a local segment can be part of many domains at once it is very unlikely in reality that these domains will all be accessing the segment at once (unless the segment is a segment of semaphores). Thus we loose little, if

anything, by making our local segments accessible in one domain at a time. Cosserat's indirect entry mechanism is a generalization of our label mechanism. We only permit a labelled c-list to be built up by the domain to which the segments in the c-list belong. But again we feel that we cater for the major use of the mechanism (the handling of permanent or own data) and that further generalization is not required.

Cosserat's rule of creating a new base segment for every change of domain brings undoubted advantages when it comes to creating new processes, but its efficacy is more open to question when the number of virtual processors in a system is fixed (a feature we shall expound upon further in the next chapter). There is very little information that can be left behind in the old base segment and not transferred to the new base segment. If the simple parameter area is organised as a set of stack frames then only the frame for the parameters being carried to the new domain need be put in the new base segment. Otherwise the only item not required in the new base segment is the old domain's return link. Since processor base segments are the most frequent movers between sites in a distributed system (see the sample results in appendix A) it is important that they be small. But there is a definite trade off between the transmission time saved on one hand and, on the other hand, the extra copying involved. Splitting up the processor base segment may also cause the occasional

delay, when doing a return, when all the required segments save the old processor base are at one site. Only experimental evidence from real implementations can resolve questions such as this and the questions we will be raising in the next chapter as we examine more facets of distributed systems.

# CHAPTER 7

# DISTRIBUTED SYSTEM METHODOLOGY

The last chapter presented a model for a distributed system in terms of segments, capabilities and domains. We did not specify what was to be the function of any of the domains, nor did we indicate how a programmer might go about constructing a domain. We now direct our attention to these and similar topics. This chapter is concerned with the wider perspective of distributed system design.

## SECTION 1: RESOURCE ALLOCATION.

For some years now there has been a school of thought that advocates the limitation of forms of dymanic behaviour in operating systems [HANS73,HANS74,HOAR74a, HOAR74b,HANS76]. The THE operating system [DIJK68] has a fixed number of virtual processors. The recently completed SOLO system [HANS76] has not only a fixed number of virtual processors but is conceptually compilable as a single program, so that all interactions within the system are able to be checked at compile time. We concur with such sentiments, as they lead to a fresh view of resource allocation which we believe is suitable

for distributed systems.

Systems with dynamic creation and deletion of processes usually handle resource allocation on a hierarchical basis. All the system resources are initially vested in an ultimate ancestor [HANS73,SEVC74]. Whenever a process is created it is given some of the resources of the creator process; if not 'consumed', the resources are returned to the creator process when the new process is deleted. The ultimate ancestor represents a potential bottleneck since it has to deal with all the systems resources. In our distributed system resources are associated with domains, allowing control to be spread throughout the system. Each virtual processor can enter any domain (known to it) and access the resources in it. But the virtual processor must execute the code of the domain while accessing the resource. Thus the domain can control all its resources, all of the time.

This form of distributed control does not preclude the use of process hierarchies but it does remove a lot of the justification for them. Ability to freely create processes could also be troublesome if it is desired to limit the total resources available, at any one time, to a user. If we were to allow a process to control the progress of another and even destroy it, as is permitted in many process orientated systems [KNOT74], then a process would have to be a global object. This follows from the requirement to locate the process that is to be

controlled or destroyed. The management of global objects is relatively expensive. The number of processes would grow in proportion to the number of sites, presenting larger and larger directory or associative memory requirements.

Overall, considerable simplicity and efficiency is gained by having a fixed number of virtual processors (which are not global objects), one virtual processor per user. If some form of parallelism is required a user can be permanently allocated more than one virtual processor; in the SOLO system he is given three, one to handle input, one for computation and one for output.

Domains:

Since a system is likely to have a fixed maximum number of resources for long periods of time it is logical to have a fixed number of domains to manage these resources. We include as resources compilers, editors and anything usable by more than one user. Using a fixed number of domains confers two advantages:

1) Every kernel, as part of the management of global objects, needs to keep information about every domain. With a constant number of domains, fixed space for this information can be allocated inside kernels, leading to more efficient operation of the kernels. Of course when there is an increase in the number of

resource types in the system a recompilation of the kernel will be required.

2) The finding of a domain is considerably simplified if it always exists. When a kernel receives a message related to a domain that does not reside at its site, it need only pass the message on to the site where it believes the domain to be. Provided that the message travels faster than the domain (see chapter 8) it will eventually reach the correct site. If domains were dynamically created and deleted then the kernel would have to decide whether to pass the message on to another site, or initiate the creation of the domain, or regard the message as being for a deleted domain and hence erroneous.

Having a fixed number of domains in a system is not an absolute fiat. Arrangements could be made for the locating and loading of some domains from a file store when required (an obvious exception is the basic domains that manage the file store), in a similar fashion to Spier's implementation. As well as the added complexity in domain management described above, knowing when to unload the domains again is likely to be a tricky problem.

## SECTION 2: HANDLING USER PROGRAMS.

So far we have been careful to avoid mentioning user code. We have adopted the attitude of Hoare [HOAR74b] towards user code. He believes that all user code should be interpreted by the operating system. He reasons that a user cannot compromise the security and robustness of a system if all (sensitive) operations are vetted by the operating system.

Our adoption of this philosophy allows us to have a fixed number of domains in our distributed system since users do not generate their own domains. We provide a user supervisor domain. One, or more, of the local segments in an incarnation of this domain is user code. The user supervisor 'interprets' this code. In practice this would mean that the user code is directly executed but the domain fields any 'supervisor' calls, which it translates to interdomain calls.

There is no compelling reason why the appearance, to a user, of a system should bear any relation to the structure used to implement the system. Placing user code in a supervisor cocoon means that the ordinary user need not be appraised of domain structures when it comes to writing his own programs. Interpretation also provides a hook upon which can be hung such facilities as execution time limits, error diagnostics and recovery, and console generated interrupts.

Unfortunately this approach also rules out the sharing between users of the same copy of user code. If some user program is in such demand that the likelihood of two or more people using it simultaneously is significant then the program could be incorporated into the operating system, either directly as a single domain, or, in a rewritten form, as several domains.

## SECTION 3: ADDRESSING.

### Addresses in capabilities:

Another topic we have not yet touched upon is the form of addresses stored in capabilities. Capabilities always reference segments residing at some site in the distributed system. When a segment is said to reside at a site we mean that the segment is stored in the private active storage of that site. The active storage may be simply primary memory or could consist of backing store as well, provided that the backing store is controlled solely by the site. (In the later section on peripheral handling we show that problems can arise with shared control of backing store devices).

A segment's address, as stored in a capability, is assumed to be in two parts. The first part specifies the

site where the segment resides. The second part is some form of address to be interpreted by the kernel at that site. This second part could consist of:

1) a segment starting address (only suitable for one level memory and not allowing any repacking of memory)
2) a segment table offset (allowing backing store and repacking)
3) a key for a segment hash table (also permitting backing store and memory repacking).

The third approach is likely to be the best in a real system because it gives more compact segment tables and, if the keys are made unique system wide [FABR74], it provides a useful robustness [LAMP74].

Moving segments between sites:

A capability for a segment also has a length field and both this field and the segment address play a role in the movement of segments between sites. When a kernel has accepted a domain incarnation c-list, it initiates the transfer to its site of all segments of the domain incarnation. The kernel scans each capability in the c-list and determines the location of each segment from the first part of the segment address. From the length field the kernel determines how much memory space each segment will require when it arrives. The kernel could allocate the space there and then. The kernel sends a message to each site that has one or more of the segments

it requires, specifying each relevant second part of the segment address and requesting the segment be sent to it. When each segment finally arrives, its capability in the c-list is altered to reflect its new address. When all the segments whose capabilities are in the c-list are at the kernel's site, then the c-list is marked as a valid entry capability and the domain incarnation is ready to run.

The action taken is slightly different in the case of code and public segments, for the capability for these is just a name (see chapter 6, section 1). The kernel which wants the segments sends a message to the site where the segments are residing (located with the aid of tables or associative mechanism in the communication subsystem). The segments are sent to the requesting site when they are no longer required at their current site, and action is taken to appraise all the kernels of the new site for the segments. (More details are given in chapter 8).

The advantages of pre-loading all segments of a domain incarnation, rather than requesting segments piecemeal from other sites as they are required, can be deduced from the above description. Firstly, all segments required from a particular site can be requested with a single message, saving some communication bandwidth usage, and, far more importantly, interrupting that site only once, rather than for every segment. Secondly, the total (extra) space requirement of the domain incarnation

can be determined before any segments are requested from other sites. Thus if the site has insufficient space for the domain incarnation the appropriate action, normally sending the c-list to another site, is taken before any segments have been transferred to the site.

Capability hardware:

The generation of addresses within a domain incarnation must be in the form of an index into the c-list to select a capability for a segment followed by an offset within the segment to select the required item. This obviously enforces the confinement of all accesses to be within the domain incarnation.

It depends on the hardware facilities as to how the physical processor uses capabilities. Since each domain incarnation's capabilities are stored in the entry capability or c-list for the incarnation the use of a fixed, and reasonably modest, number of capability registers is one option available. This is the approach used in the Plessey 250 [COSS72,ENGL74]. When a domain is ready to run, the kernel loads the hardware capability registers with the capabilities in the c-list, suitably translated to hardware addresses. Since we have postulated that a domain incarnation should last for some appreciable time, the overhead of loading perhaps 16 registers at the start of a domain incarnation and

unloading them again when an interdomain jump is requested, should not be too large. This is provided that these registers do not have to be unloaded and loaded again every time the kernel receives an interrupt of any sort. As we indicated in chapter 3, the volume of interrupts will grow as the size of the distributed system grows and the preservation of context could quickly become a dominant unproductive factor. The operation of the Plessey 250 has been described by the phrase "Don't interrupt me, I'm computing" [HAYN73] because external interrupts have been abolished [ENGL72]. This extreme philosophy need not be employed in a distributed system provided that kernel operation is clearly differentiated from execution in a domain incarnation, and simpler context switching is provided for the kernel.

Alternatively a set of associative capability registers similar to those used in the CAP system [NEED72] could be employed. This would allow entry capabilities of arbitrary (or near arbitrary) length, that is large numbers of segments in a domain incarnation could be accomodated, but not all of them could be accessed quickly. The whole c-list need not be loaded at the start of a domain incarnation, entries would be added to the associative registers the first time the capability was used. Further, appropriate design could ensure that the contents of the associative capability registers remained usable after the handling of an

interrupt and even after an interdomain jump and subsequent return (assuming the entered domain did not require the use of all the registers for its own c-list).

## SECTION 5: PROGRAMMING LANGUAGES.

### Constructing domains:

The code that constitutes code segments has to be written by someone. We now look at how appropriate present languages are for the task. Our particular interest is in the representation of segments and their manipulation to form domains.

High level languages offer the programmer segments in many guises. In arrays the offset within a segment at which a data item resides is obviously specified by the index. Other structures (e.g. RECORDS in the IMP language [STEP74]) have symbolic names for the various data items in segments, it being one of the functions of compilers to map these names into offsets.

Most languages however do not offer the programmer any means of specifying domains. Automatic rules could be devised for constructing domains from programs in many languages, but the efficiency, particularly in our

distributed system, of such automatically created domains is open to question. Such domains are likely to be so small that the overheads involved in domain changing will dominate the useful work done in the domain.

For example, in ALGOL 60 the only two possible automatic rules are to make the whole program into one domain or to make every procedure the basis of a domain. In the B6700 system [ORGA73] the code for every ALGOL procedure is put in a separate segment. A recent study [BATS76] suggests that the average number of instructions executed from each code segment each time it is entered is of the order of 50 to 100. This is too few instructions to carry the overheads of domain entry so the ALGOL procedure is not a suitable basis for a domain in our system.

FORTRAN does provide a way of generating larger domains than just individual subroutines. The COMMOM block is a suitable structure to be made into a segment. Sometimes it may be possible for all subroutines to be divided into disjoint sets accessing different COMMON blocks in which case domains can be constructed with a code segment containing the set of subroutines, and with local segments containing the mutual COMMON block(s), all other local data, and arrays. When it is not possible to form disjoint sets of COMMON block accessors then some sort of programmer intervention is required to identify which COMMON blocks are to be used as the basis of

domains and which should be passed as parameters between domains. This requires the same sort of techniques that are used to identify overlays [SELI72].

SIMULA 67 [DAHL66,DAHL72,ICHB74] provides in its 'class' concept a programming analogue to domains. A class defines both data objects and the operations, in the form of procedures, to be performed upon them in the same way that the code segment of a domain incarnation defines the operations that are performed on local segments. However in programs these procedures are likely to be very short so that domain changing to enter a class may have unacceptably high overheads. Further in SIMULA 67 access to the data (attributes) of a class is permitted directly without executing one of the class procedures. Nevertheless a restricted form of SIMULA 67 could provide a suitable basis for developing a language for domain handling.

Quite a number of languages provide facilities for separate compilation of parts of a program. There are variations on how much compile time or link time checking is performed. Complete checking is feasible when there is no recursion between separately compiled parts. In some circumstances it is reasonable to assume that these 'external' portions constitute the basis for a domain in that they perform a definite part of a computation. Of course, often these separately compiled sections provide a service environment for the rest of the program so that

the frequency of use of the separate sections is high and the duration of residency is low. But, again with appropriate discipline, the separate compilation facility does provide a basis for the construction of domains.

To summarise, our desiderata for a programming language in which to write domain structured programs include provision for the manipulation of segments as basic items, and structuring rules that lead to easy specification of appropriately sized domains. The entry points to a domain must be obvious. This can be achieved by specifying routines to be 'external' [STEP74], or negatively by employing the 'hidden' feature proposed for SIMULA 67 [HOAR74b].

Language restrictions:

So far we have looked at features that would be conducive to efficient domain structure. Attention is now turned to two language features, the usual generality of which would have to be severely restricted in a domain system. These are parameters and pointers.

The parameter passing mechanisms of many high level languages are too sophisticated for our model to implement efficiently. The model provides in effect the same parameter passing mechanisms as FORTRAN: call by value (with possible copyback) for simple variables and call by reference for arrays (segments). That this, in

some way, is sufficient is demonstrated by the large number of running FORTRAN programs in existence. If, as we would wish, domains embody some complete and quite substantial function then the dictates of good design suggest that the number of parameters to be passed between domains should be small and that possible complexities of side effects and so on should be avoided [PARN72]. Hence we feel our model's mechanism to be adequate; the type of parameter passing employed within domains need not be restricted to that possible between domains.

Pointers, that is stored memory addresses, have recently fallen into disfavour with some programming experts [WIRT74] because they lead to an item having two or more names, and hence detract from program clarity. In our distributed system any pointer to an address in another segment would cause immense difficulties. There would be two ways of storing such a pointer. One way would be to store the full capability of the segment (plus offset) which violates our principle of having only one capability in existence (for local segments) and keeping that capability in a fixed location. The second method would be to store the c-list offset of the segment (and offset within the segment). Problems would arise if the segment containing the pointer was passed to another domain incarnation because then the pointer would be incorrect. Thus in capability systems such as ours the use of intersegment pointers cannot be supported.

We made no mention of public segments in our discussion of programming languages. Only two languages, that we know of, embody such a concept directly. Concurrent Pascal [HANS74,HANS75] is the original language of these two. Details of the second language SIMONE, which is similar to Concurrent Pascal, have been published very recently [KAUB76]. One of the elements of these languages is a 'monitor'. Monitors, before being incorporated in Concurrent Pascal, were developed by Hoare [HOAR73,HOAR74a] and Hansen [HANS73]. A monitor consists of some data, and procedures to manipulate the data. Monitors have the following properties:

1) the data of a monitor is global in the sense that only one instance of the data exists, thus corresponding directly with data in a public segment.
2) the monitor data is only accessible to the monitor procedures; all manipulation of the data is by calling these procedures, just as a domain must be entered to access its public segment.
3) at any one time, at most one virtual processor can be progressing in a monitor; it will maintain exclusive access to the monitor's data until it exits from the code (or suspends itself on an internal queue), thus allowing guarantees to be made about the integrity of the monitor's data.

The finer details of monitors' properties have yet to be agreed upon. For example Hansen has his monitors contain global data only [HANS75] while Hoare's monitors contain both global data and multiple copies of user data (equivalent to local data) [HOAR74b]. It is on the basis of Hoare's type of monitor that we named domains having a public segment 'monitors'.

Exclusive access and the condition queue:

Another undecided property of monitors is that of how long exclusive access to a monitor should prevail. Obviously when a virtual processor finally exits from a monitor access can be given to another virtual processor. The problem arises when the virtual processor makes a call to another domain. Should all other virtual processors be denied access while this call is in progress? To do this poses far more management problems [LIST76] than the approach we have adopted which is that whenever a virtual processor executing in a monitor makes a call on the kernel (as it will to change domains) it looses kernel guaranteed exclusive access.

To allow longer periods of effective exclusive access and to facilitate certain forms of virtual processor intercommunication, monitors have to provide a facility whereby a virtual processor can suspend itself while waiting for some condition to be fulfilled (by some other

virtual processor). When it suspends itself the virtual processor looses exclusive access to the monitor. Hansen provides a general queueing mechanism in a monitor so that other virtual processors can manipulate the queue (called the condition queue) in any desirable fashion. Hoare is more strict; condition queues have to be served either 'first in first out', or in order of a priority specified when joining the queue.

We stated in chapter 6 that kernels kept validated entry capabilites in some form of 'ready to run' queue. The running domain incarnation is at the top of the queue so that its suspension involves removing its entry capability and storing it in the condition queue of the monitor. The condition queue has to be part of the public segment. It is no good making it part of a kernel area unless the monitor is to be tied to a particular site. No major problems arise with entry capabilities being moved, undetected by kernels, from site to site. When another virtual processor, executing in the monitor, wishes to release a suspended virtual processor it removes the entry capability from the condition queue and passes it to the local kernel which re-validates it. Eventually the domain incarnation will be scheduled for execution again.

One difficulty in following Hansen's approach of allowing general manipulation of the condition queue is that suitable constructs must be provided for the domain

code to examine the capabilities in the condition queue. We cannot see any neat way of providing these.

Secretary processors:

The original impetus for monitors came from Dijkstra's 'secretary' concept [DIJK71]. In a process orientated system a secretary process maintains global data, all requests to manipulate it being sent as messages to the secretary. In a monitor type system virtual processors can enter the monitor themselves to manipulate the data. However, particulary when dealing with peripherals, situations could arise where the kernel cannot know which virtual processor should be dispatched, to answer an interrupt, for example.

Thus in our model we make provision for some monitors to have secretary processors (or daemons [SALT66]) associated with them. These special virtual processors execute only in the monitor and may use different code from the normal monitor user. Their purpose is to provide general housekeeping functions on the data structure that constitutes the public segment. Secretaries have a special relationship with kernels. Peripheral interrupts are associated with unique secretaries. When a kernel recognises a peripheral interrupt it schedules the appropriate secretary processor to run. When this secretary processor runs it

can manipulate the queue of the monitor to which it belongs to have the correct virtual processor scheduled. While this arrangement certainly gives flexibility in handling I/O devices we are not so sure of its efficiency. We postpone discussion of this to chapter 11.

## SECTION 7: PERIPHERAL HANDLING

We have just shown how the secretary processor concept can aid in the management of peripherals. Using secretary processors however is just one approach to managing peripherals in a domain structured distributed system. There are a number of possible approaches depending on the functional capabilities of peripheral controllers.

In this section we propose various schemes for handling disk operations, predicated on the intelligence of the disk controller. We have chosen disks as an example because:

1) they could be quite heavily used so that ineffcient operation is less tolerable than for some other peripherals.
2) disk usage involves reading and writing, a read possibly being of something previously written.

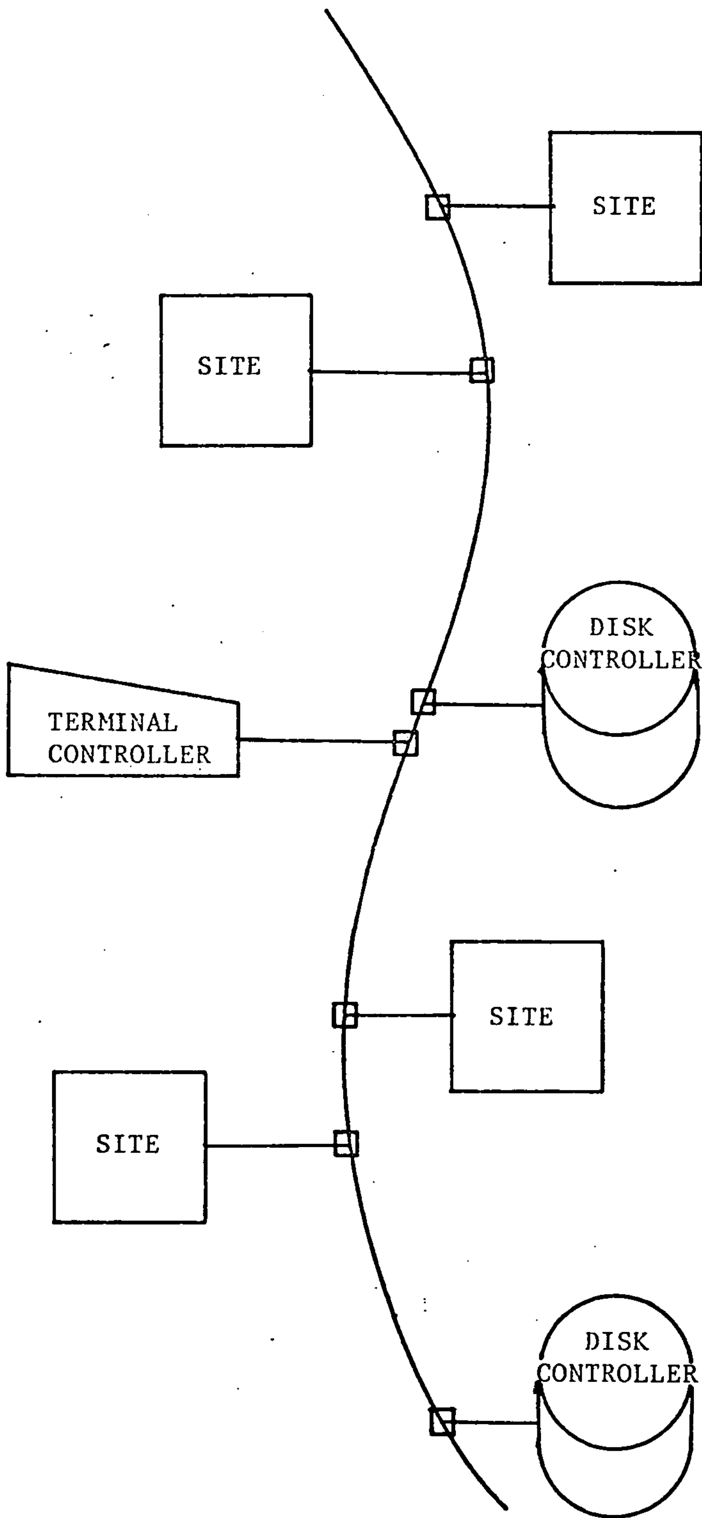
The second point has ramifications for distributed control which we point out later. We assume that the

unit of reading and writing is a segment, more specifically a local segment used as a parameter. A write involves passing the segment to the disk handling domain which on exit returns a null segment. A read involves simple parameters and a null segment being passed to the disk handling domain and a full segment is returned.

The workstation approach:

Undoubtedly the neatest scheme is to assume that the disk controller or other peripheral controller is a site in its own right, fully integrated into the communication system. When using a bus type communication subsystem, which does not require more links as more sites are added, a network architecture such as depicted in figure 7.1 can be achieved. With the advent of microprocessor controllers the workstation concept, as embodied in the CDC 7600 system [ELRO70,JONE71], is becoming practicable for more modest sized systems. Of course, in a distributed system, the workstations do not serve a single large processor but rather interact with all the general purpose sites in the system.

The workstation must, to all intents and purposes, behave like any other site in its interactions with other sites (internally it could be rather different in structure). This site will have only one domain but must be capable of handling entry capabilities correctly.




A DISTRIBUTED SYSTEM USING WORKSTATIONS

Figure 7.1

Thus for example a request for a disk read would be programmed as an interdomain call on the 'disk controller' domain. The entry capability would arrive at the disk controller which would validate it as usual and queue it in an equivalent of the 'ready to run' queue (but presumably so as to optimise disk accesses). When the read had been performed the disk controller would initiate an interdomain return, with the read data being an argument segment to be returned to the calling domain incarnation.

This approach can be viewed as multiplexing virtual processors on the physical processor of the disk controlling site. One requirement of this appraoch is a large buffering capacity at the controller site because the argument and processor base segments do not immediately leave the site when the incarnation has terminated (i.e. the disk operation has been completed). They stay there until a site has been determined for the resumption of the calling domain incarnation and the kernel of this site then requests the segments to be sent to it.

One objection to this scheme that could be raised is that the general purpose power of a microprocessor is being dedicated to a single job in which it could be considerably underutilized. In so far as the the processing power is general and being underutilized this is a valid objection. But the architecture of a

peripheral controller is likely to be rather different from a general purpose computer and since the capacity of a peripheral can be quite easily determined, the power of the controller can be matched to the capacity. Substantial underutilization of peripherals may be unavoidable in small systems but for larger systems it is an indicator of bad design.

Limited capacity controllers:

This is the scheme that we chose to simulate (see chapter 9). Basically it supposes that a controller will not be designed specifically to fit into a domain orientated system but will be capable of using the communications subsystem to transmit segments and a limited repertoire of control messages to and from other sites. A domain, the disk handler domain, is required to reside at some site to assist the disk controller in its work. Whether this domain is tied down or not depends on the sophistication of the controller, the communication system and the kernels in handling interrupt type signals from the controller. The disk handler domain needs to be a monitor with an associated secretary processor, so there are fewer problems when it is tied to one site.

This approach assumes that the disk controller has a number of buffers for holding segments and that it sends a message to the controlling site (i.e. the site where

the disk handler domain resides) whenever one of its buffers becomes free. This message is interpreted, by the receiving kernel, as the secretary processor's entry capability for the disk handler domain. The kernel duly validates this entry capability and so eventually the secretary will run. It will initiate a read or write if there are any outstanding, or set a flag to indicate to any other virtual processor that subsequently enters the domain that it may initiate its own read or write because there is a buffer available.

It was to avoid congestion at the disk handler site that we introduced the pseudo capability states of 'ondisk' and 'desc' (chapter 6). When a virtual processor wishes to write a segment to disk, it transfers as a paramter to the disk handler incarnation simply a descriptor of the segment, not the segment itself. This descriptor is placed in a queue of descriptors of segments waiting to be written to disk and the virtual processor exits immediately from the domain (unless the queue is full). When the disk controller has a free buffer into which it can receive the segment a request for the segment to be dispatched direct to the disk controller is sent to the kernel of the site where the segment is still residing.

Normally reads are executed before writes. A virtual processor enters the diskhandler domain with simple parameters decribing the read. Instead of the processor

suspending itself to wait for the segment to be read then returning to the calling domain the entry capability for the return is prepared, including one segment capability marked 'ondisk'. This entry capability is not validated until the disk read has taken place into a buffer in the disk controller. (In fact this invalid entry capability could be sent to the disk controller as the read request and be returned to the controlling site when the read is complete, whence the kernel there starts to validate it). When a site has been chosen for the incarnation of the calling domain to resume, then the kernel of that site sends a request for the read segment direct to the disk controller. The disk controller dispatches the segment from one of its buffers.

Notice that in a real system a check, on the queue of descriptors of segments that are waiting to be written to disk, will have to be made before a read is performed. A situation could easily arise where a virtual processor is trying to read a segment that it had previously written (that is called the diskhandler domain and returned) but which segment has not in fact got as far as being written on the disk. This is one reason why it is not possible to have every site control the same disk (perhaps as a kernel function). Unless a virtual processor is going to be held up until a disk write is acknowledged as completed, a single list of outstanding writes for each disk is required. Thus notification of writes to a particular disk must pass through a single site.

The other reason that all sites could not control a shared disk is related to buffer management. There is a limited number of buffers in the disk controller, the freeing of one of these buffers indicating that the controller is capable of accepting another request. Although conceivably the disk controller could broadcast that the buffer was free, all the sites would have to agree on which site was permitted to make the next request. The necessity of having all sites in agreement is something that we have studiously avoided, it can be a very time wasting function in a distributed system.

Plain dumb controllers:

It could be that the peripheral requires direct attachment to a central processor for control and has no buffers so that it must transfer directly to or from the main memory of the controlling site. Tying the domain that uses the peripheral to the site, and utilizing the secretary concept to handle completion interrupts and general housekeeping, may well be acceptable when the peripheral is lightly used. But if the peripheral is heavily used, as might be the case for a disk, then the controlling site is likely to become very congested. Before data can be written to disk it has to be moved to the controlling site where it is queued to be written. Data read from disk will initially go to the controlling site where it will exert an influence on the domain that

ultimately uses the data, so that that domain will tend to migrate to the controlling site as well.

Alternatively the controlling site could be 'split' into two sites, partitioning the memory and sharing the physical processor. One site would have a normal kernel and the other would perform the disk controller function we described in the independent workstation section. This scheme, although it would involve extra software to share the computer between two 'logical' sites might be ideal for a small system that was going to expand. As the use of peripherals grew they could be given their own independent sites, freeing the original sites to concentrate on the expanded workload. This is analogous to conventional small computers doing their own terminal handling but as a system grows this function is taken over by front end processors.

Efficiency:

The schemes we have described illustrate the dichotomy of dedicating processor power to a single task and risking underutilization of the processor, versus doing the task with a processor at a general site ,but, because of the special nature of the task, distorting the loading of the site. All the schemes we proposed however suffer in comparison with message passing schemes employed in process orientated systems when we consider the loading

on the communication subsystem. For in a message passing system a peripheral is viewed as a sink or source of messages. No domain incarnation is required at the peripheral controller or handler site, saving at least the movement of the processor base segment from the controlling site to the peripheral controller or handler site and back again. Of course the validation of the returning domain incarnation gives an opportunity for it to move to another site to help load balance. There is no equivalent opportunity in a message passing system. Whether this offsets the extra communication costs we do not know. We raise the question again in chapter 11.

## SECTION 8: FUNCTIONAL SPECIALIZATION.

The workstation approach to handling peripherals can be extended to cover any functionally specialized site. If the site behaves as if it had a (basic) kernel and a single domain which implements the special function then it can easily be integrated into our distributed system. A frequently proposed form of distributed system [FOST72,SELI72,COLO76] gives, in our terms, every domain a site of its own. The physical processors at these sites (inevitably microprocessors) are tailored to the domain resident at them.

We have already indicated, in chapter 2, our scepticism of the effectiveness of modest sized systems

of functionally specialized processors. We concede that because of the smooth demand presented by a very large number of users, functionally specialized sites may be appropriate for large systems. This is providing that the overall system is balanced for the load applied. However systems such as ours necessarily precede the implementation of such large systems because, for these large systems, good estimates are required of the usage of each domain. Without this information bottlenecks are almost certain to be designed into any such system built. Our system could also mature into a system of functionally specialized sites as it grew in size; specialized sites could be added if they proved cost effective.

## SECTION 9: DEADLOCK AND DISTRIBUTED CONTROL.

Spier, when discussing requirements for code segments, stated that they must be re-entrant so as to avoid deadlock between two sequences of interdomain calls, such as A->B->C and C->B->A [SPIE73a]. When used for pure domains and monitors that do not have condition queues re-entrancy will indeed permit deadlock to be avoided. However there is no such guarantee when dealing with monitors that have condition queues. If a virtual processor in a monitor wishes to retain, in effect, exclusive access while it calls another monitor, it needs to set a public variable so that other virtual processors

entering the monitor will test the variable and suspend themselves on a condition queue. In such a situation two virtual processors attempting the call sequences A->B and B->A respectively, where A and B are monitors that have condition queues, can be deadlocked. The banker's algorithm for avoiding deadlock, of dubious usefulness in single site systems because of computational overhead [HANS73], is useless in a distributed system because it requires that allocation of resources be centralized. Thus to avoid deadlock of interdomain calls, we require a hierarchical ordering of monitors that have condition queues. A virtual processor that has entered, but not exited, a monitor at a given level can only call monitors at higher levels. Thus the circular calling sequence required for deadlocks is broken. Checks that monitors obey this calling rule can be applied at compile time.

Unfortunately, it is not only when dealing with interdomain calls that deadlock can occur in a distributed system. Implicit over-allocation of resources leads to deadlock. For example, if too many virtual processors are permitted to operate in a distributed system then there will not be enough memory space at any site for an interdomain call to proceed. If no interdomain calls can proceed then no virtual processor can finish its work and release memory space. The system will be deadlocked.

Since allocation of resources can be performed at any site in a distributed system then over-allocation could easily result. Allowing each site a fixed quota of the system resources is one obvious method of control. But to fix quotas that ensure over-allocation never occurs is to condemn a system to almost constant under-utilization of resources and negates the purpose of joining the sites of the system together in the first place. Resource allocation can be done more intelligently if the allocator has some knowledge of the state of the distributed system. To this end we advocate the exchange between kernels of a couple of carefully selected parameters of the load at each site. We have already outlined, in chapter 3, the mechanisms that can be used to effect this exchange of information.

Ours is a pragmatic approach to deadlock avoidance. Providing information about global status cannot negate the possibility of deadlock occurring. But the frequency of deadlock can, by altering appropriate 'twiddle factors', be brought down to an acceptable level. In this context it is worth quoting Hoare, 'There is no a priori reason why the attempt to split the functions of an operating system into a number of isolated disjoint monitors should suceed....' [HOAR74a]. The question is just how much information do isolated monitors (kernels) need in order to compete with hierarchically controlled systems, often silted up with too much information, and subject to the delays of bureaucracy. We believe that

just a few bytes of information about the global state of a distributed system will suffice.

## SECTION 10: SUMMARY.

This chapter has presented a miscellany of items connected with the implementation and operation of a distributed system. Not every function required for a distributed system needs to be developed from scratch. Those functions of a single site system that do not, or need not, rely on system wide shared memory can be adapted, with little, if any change, for distributed systems. Addressing is modified by the addition of a site identifier, indicating where in the dispersed memory of a distributed system the address refers. The concepts of semaphores and conditional critical regions, which require common memory and a central queue of suspended virtual processors respectively, cannot be readily used in distributed systems, but monitors, with condition queues, can be used, because they localize the management of waiting virtual processors to single domains. Again the banker's algorithm is unsuitable for deadlock avoidance in distributed systems but the hierarchical ordering of calls, enforceable at compile time, can be adopted without change for a distributed system.

Our requirements for language development stem from the particular form of distributed system, domain based,

that we have chosen. While developments in programming languages, particularly recent developments in programming concepts for handling concurrency, mirror many of the features of our model for a distributed system, languages developed so far are wedded to single site systems. They offer no help in constructing reasonable sized domains, their parameter passing mechanisms are too general to be efficient and many of them permit references or addresses to be program data which is only feasible when the whole program resides in the same address space. But the requirements for a language for writing distributed systems are not esoteric and we do not think the design of a suitable language will be difficult.

However distributed systems do require some new techniques. One example is peripheral handling, when peripherals are considered as free standing entities not controlled by one particular site. Another example is the need to encode the state of each site into a few bytes of information and exchange this information between sites so that resource allocation decisions can be made with reference to the global state of the distributed system. The whole technique of managing domains is of course different for distributed systems. Although we presented the basics of domain management in chapter 6 there are still some important aspects of domain management to be dealt with. Having presented a picture of some of the wider operational aspects of a

distributed system in this chapter, we return, in the next chapter, to the narrow details of domain management.

# CHAPTER 8

# DOMAIN MANAGEMENT

This chapter is concerned both with the protocol for the movement of code and public segments and with the determination of where domain incarnations should take place. It continues the development, started in chapter 6, of the mechanisms required for handling interdomain jumps in a distributed system.

To aid the clarity of the following description we have altered our use of the term 'domain'. We have stated before that the code segment and possible public segment identify a domain. We now actually equate the domain with these segments. Thus when we write of domains being at a site or being moved from site to site, what we mean is that the code segments and possible public segments are at a site, or are being moved.

## SECTION 1: MOVEMENT OF DOMAINS.

### Introduction:

Code and public segments are the only segments shared between virtual processors and hence great care must be exercised in moving them from one site to another. Problems that could arise include the moving of segments away while they are being used, moving them away after an entry capability referencing them has been put in the 'ready to run' queue or never moving them because there is always some entry capability in the 'ready to run' queue which references them. Management of domains then requires that they are not moved prematurely and that all sites will get access to them within a reasonable period of time.

A kernel of a site can receive three types of message or request related to a domain:

1) A request, in the form of a putative entry capability, to perform a 'best' site calculation for an incarnation of the domain; called a c-request.
2) A request, also in the form of a putative entry capability (but with a suitable distinguishing tag from the above c-request), to execute a domain incarnation. That is, the site has, been chosen as the 'best' site. This we call an e-request in this chapter.

3) A request to transfer the domain (the code and possible public segments) to another site, called a t-request.

A site will only receive these requests if it is supposed by other sites to have the domain resident at its site. We call a site which is supposed to be the site of residence of a domain, the target site for the domain. The correspondence between target and reality depends on the method of global object management (chapter 3). In an associative scheme the target is the same for all sites. Also if the updating of the associative memory in the interface unit is performed as soon as a domain arrives then the target site will (almost) always be the correct site. In a system that employs the updating of directories the target for a request could be quite out of date; the directory entry could be changed after the request was addressed and put in a queue for transmission, the message to update the directory could be delayed. In what follows, we assume a distributed system using directory updates, as it is obviously the more difficult case, and we make our strategies robust against old information.

Pure domains:

The management of pure domains is easier than dealing with monitors so we describe a strategy for pure domains first.

In our distributed system there is one 'original' of a code segment of a pure domain and possibly many copies. Whenever a kernel receives a t-request for the code segment of a pure domain, the original of which is residing at its site, it sends off the code segment immediately but keeps a copy. The kernel then decides what to do with the retained copy:

1) If the domain was being used when the t-request arrived, or there is some 'ready to run' incarnation of the domain then the copy is kept.
2) If the domain is not required and the kernel is short of memory space then it deletes the copy to free space.
3) If the domain is not required but there is already sufficient space at the site then the copy is kept so that there is no need to fetch the original from another site if the kernel's site is subsequently chosen as 'best' site for another incarnation involving the same domain.

The original is sent from site to site because the sending site may have no use for the code and so could reclaim the space occupied by it immediately, whereas the

requesting site obviously always has a requirement for the code. Immediately prior to sending the code segment to the new site the kernel broadcasts the identification of the new site so that other sites can update their directories. The other sites sending messages related to the domain will, eventually, having updated their target site, send the messages to the new site.

If a t-request arrives when either there is only a copy or no segment at the site then it is passed on to the target site for the original. But if the kernel has itself sent off a t-request then it 'reserves' the domain for the site of the incoming t-request. The kernel sends off the original of the code segment, keeping a copy, as soon as it arrives. If a t-request arrives after the kernel has reserved the code segment for another site then it sends the t-request off to that other site. A chain of sites, each having reserved the domain, could build up if there were delays in the segment being transmitted from site to site.

When a c-request, a request to have a 'best' site calculation performed, arrives at a site the kernel follows one of the following courses of action.

1) If the original of the code segment is present then the 'best' site calculation is performed and the entry capability validated as usual.
2) If a copy of the code segment is present then the 'best' site calculation is performed. If the kernel's

own site is chosen as 'best' site then the entry capability is validated as usual, but if another site is chosen then the c-request is sent off to the target site of the original for the calculation to be repeated. This is done to encourage the aggregation of all incarnations of a particular domain to be at one site (see later).

3) If, when the c-request arrives, the kernel is already expecting the original from another site (that is it has dispatched a t-request), then the c-request is put in a queue to await the arrival of the code segment. When it arrives the 'best' site calculation is performed as usual for all entries in the queue.

4) If none of the above conditions prevail then the kernel passes on the c-request to the site it considers to be the target site for the domain.

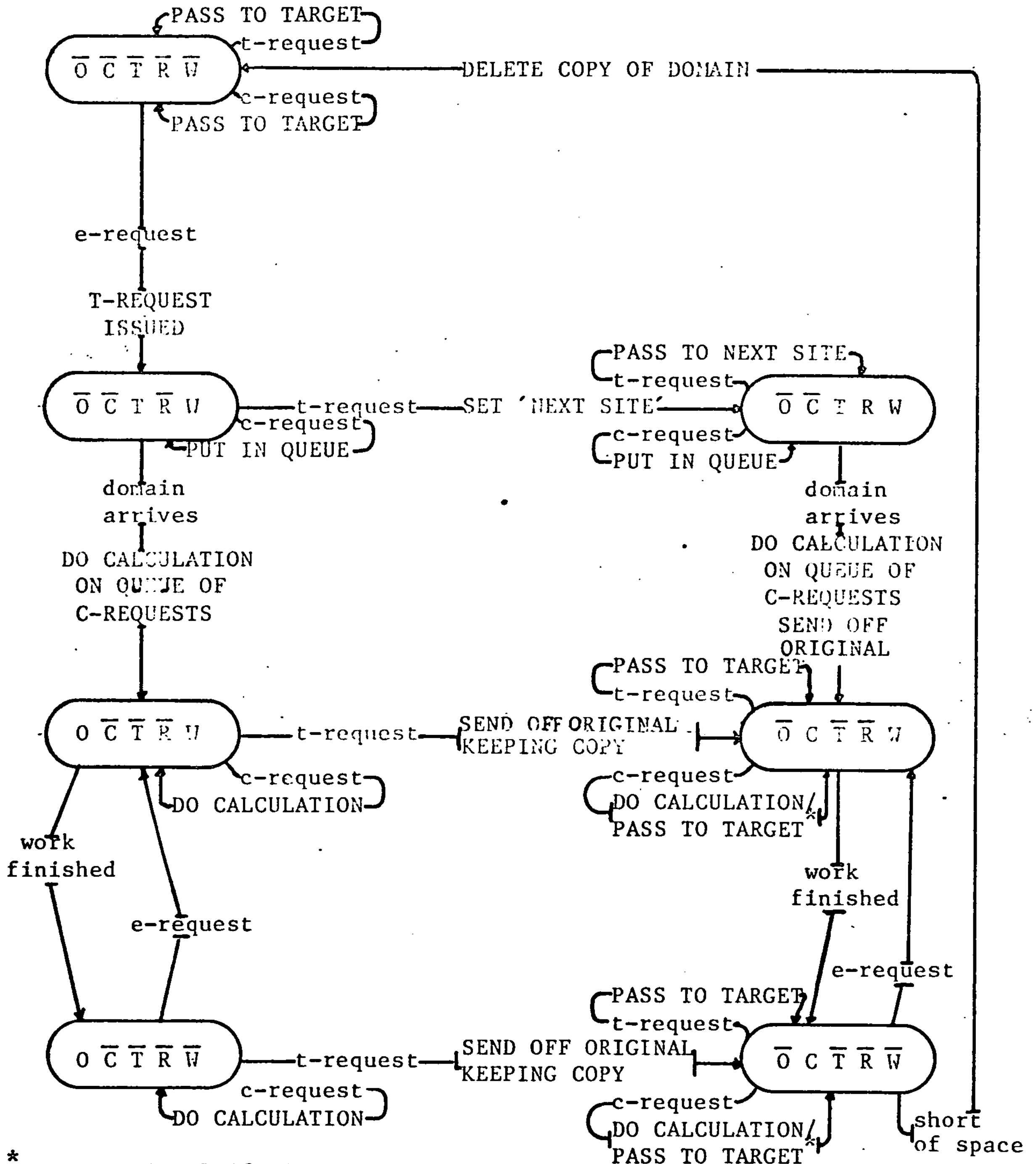
Figure 8.1 is a state diagram for the management of pure domains. It details the various transitions and actions (outputs) that can occur when c-requests, t-requests and domains arrive at a site.

STATE DIAGRAM FOR MANAGEMENT OF PURE DOMAINS.

STATES:

| | | | |
|---|---|---|---|
| O | original of domain is at site | $\overline{O}$ | original resides elsewhere |
| C | a copy of domain is at site | $\overline{C}$ | no copy held at site |
| T | domain requested but not arrived | $\overline{T}$ | no outstanding t-request |
| R | domain reserved for 'next site' | $\overline{R}$ | not in chain of sites |
| W | outstanding work for domain | $\overline{W}$ | no work for domain |

Inputs are given in lower case, outputs are given in upper case. When an input causes no change of state and no output it is omitted.




* PASS TO TARGET if site not chosen as best site.

Figure 8.1

Monitors:

When we are dealing with monitors we cannot make copies of the segments involved and send off the originals when t-requests arrive. Basically what the kernel does in this situation is to reserve the domain for the site that issued the t-request. It refuses to process any more c-requests for the domain, sending them on to the site that requested the domain, where they are queued up. Eventually there will be no more work outstanding at the site where the domain resides so the kernel can then send the domain off to the requesting site. For the sake of efficiency we introduce a modification to this strategy depending on the type of work outstanding.

When a kernel sends off requests to other kernels for the segments required to make up a domain incarnation it notes, amongst the information it keeps about every domain, that there are some 'external segments' outstanding. Also, every time a domain incarnation is placed on the 'ready to run' queue this is noted against the relevant domain. Whenever segments arrive at the site, or domain incarnations finish execution, this information is amended appropriately. Thus kernels can tell whether outstanding work is all at the site or some of it is awaiting the arrival of segments from elsewhere.

In tne case where some of the outstanding domain incarnations are awaiting segments from other sites, the kernel will still evaluate incoming c-requests after it has reserved the domain for another site. If all the segments specified in a c-request are at the local site already, then that domain incarnation is placed in the 'ready to run' queue but otherwise that c-request is passed to the next site, to be queued for re-calculation. Once there are no outstanding external segments all c-requests are passed to the next site regardless. This modified policy means that a kernel can perform useful work while waiting for a segment to arrive from another site, but it cannot hold onto a monitor indefinitely.

Any t-request that arrives after the domain has been reserved for another site is sent to that site. The site which has requested the domain reserves it for the first site from which it receives a t-request, by setting a 'next site' pointer to the requesting site. Thus, again, a chain of sites wanting the domain could be built up. However, because no c-requests (except those involving only segments at the local site) are processed after the first t-request is received, the chain must terminate quite quickly. When the monitor arrives at the first site that it was reserved for, there may be a queue of c-requests waiting to be dealt with. One policy with respect to these is to process them irrespective of whether or not there is a subsequent outstanding t-request. Figure 8.2 gives a state diagram for managing

STATE DIAGRAM FOR THE MANAGEMENT OF MONITORS

STATES:

| | | | |
|---|---|---|---|
| M | monitor at this site | $\overline{M}$ | monitor at another site |
| T | monitor requested but not arrived | $\overline{T}$ | no outstanding t-request |
| R | monitor reserved for 'next site' | $\overline{R}$ | not in chain of sites |
| W | outstanding work for monitor | $\overline{W}$ | no more work at this site |
| E | segments still to arrive | $\overline{E}$ | no external segments |

Inputs are given in lower case, outputs are given in upper case.
When an input causes no change of state and no output it is omitted.

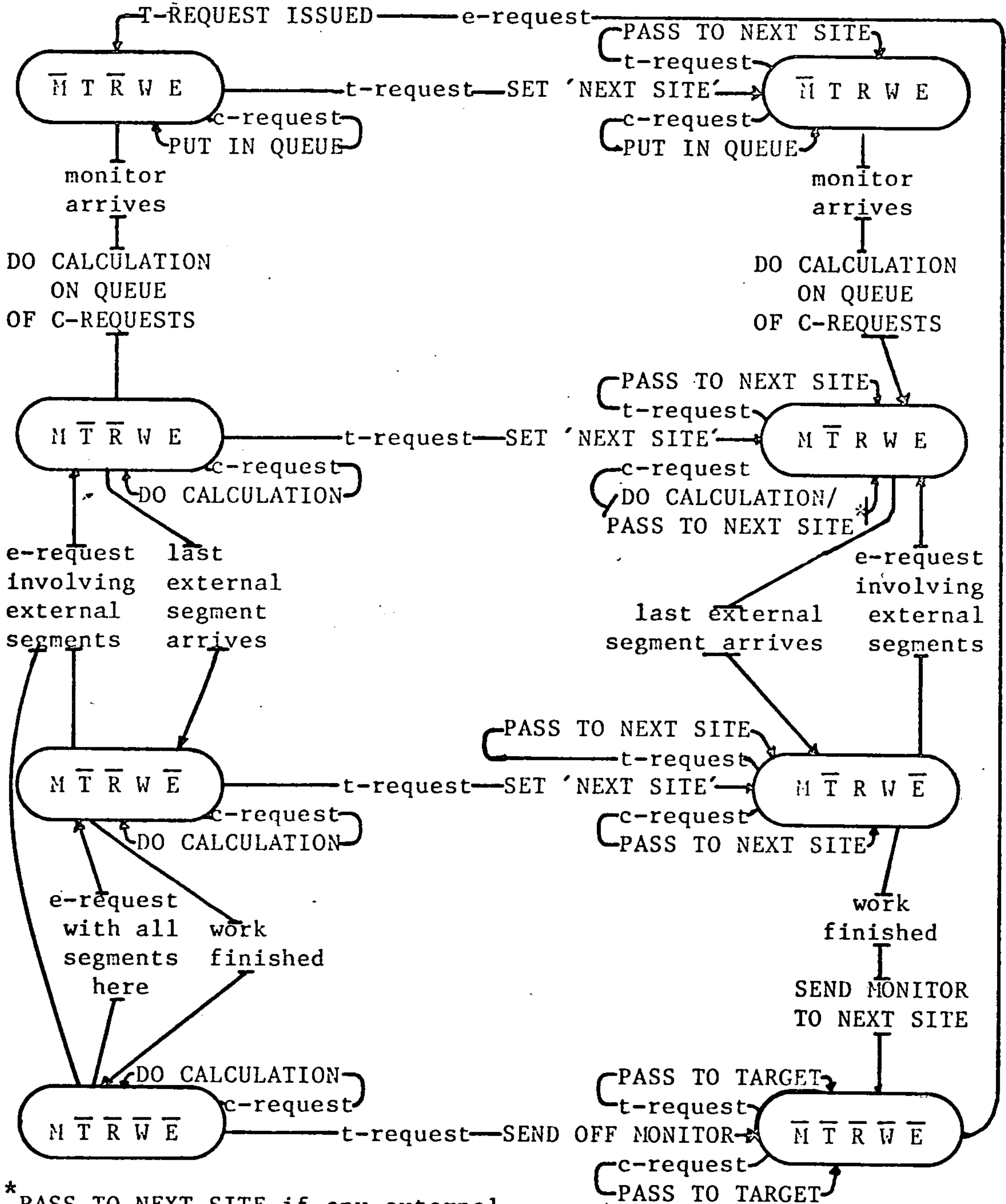



Figure 8.2

monitors which embodies this policy.

Observations:

It remains to be shown that the two strategies we have outlined above give rise to desirable behaviour. Firstly, domains are not removed prematurely from sites in the following sense: if there are any segments being fetched from another site for an incarnation of the domain or any incarnations at the site are ready to run, or indeed running, then the code and possible public segments for the domain incarnation will not be moved until the relevant domain incarnations have run. Secondly, because either the domain is dispatched to the next site in the chain immediately, in the case of a pure domain, or as soon as all outstanding work is completed, in the case of a monitor, the domain will not remain at a site indefinitely once a t-request has been received (assuming that domain incarnations are of limited duration).

Showing that all sites which issue t-requests eventually receive the domain, requires more formalized argument. Our first concern is to show that a t-request never gets lost in a closed loop of reservations (each site reserving the domain for its successor with the last site reserving the domain for the first site). We make the following assumptions.

Assumption 1: No site issues a repeat of a t-request until it has actually received the domain and passed it on to another site.

Assumption 2: After a domain has left a site, the target of that site for the domain, can be any of the sites subsequently visited by the domain. The target cannot be any site not visited by the domain since it was last at the particular site.

Assumption 3: The relative speeds of movement of t-requests (and c-requests) from site to site and of domains from site to site is such that a domain cannot always stay one step ahead of a t-request (or c-request).

By assumption 1 there can be no loops in the chain of reservations whose head is the site where the domain is currently resident (we call this the main chain), because a site that is in this chain has not received the domain and so does not send out extra t-requests. Nor does a site produce forks in the chain by reserving the domain for two or more other sites.

It is possible for temporary independent chains to form. A site issues a t-request and before it becomes part of the main chain a second site, having the first site as its target, sends it a t-request so that the second site is duly noted as 'next site' for the first site. But since the first site is a target for the second site then, by assumption 2, the set of possible

targets, direct and indirect, for the t-request of the first site cannot include the second site. This argument can be extended to any other site that subsequently joins the independent chain, so that it is impossible for the outstanding t-request of the first site to arrive at any site in the independent chain and so form a closed loop. By assumption 3 the outstanding t-request of the head of this independent chain will eventually reach the main chain and the whole independent chain will be appended to it.

Now that we have shown that no closed loops form we can state that every t-request issued results in a site reserving the domain for the issuer of the t-request. Since, once a domain has arrived at a site where it has been reserved for another site, it is eventually dispatched to that site, by induction the domain will eventually arrive at every site that issues a t-request.

The validity of the above conclusion, that every site that issues a t-request will eventually receive the domain, depends on the correctness of the three assumptions listed above. Assumption 1 is a matter of the policy implemented in each kernel. In a directory update system, assumption 2 requires that no update messages issued prior to the domain arriving at a site are accepted by the site after the domain has left the site. This could be ensured by affixing a generation number to each update message and allowing a site to

accept an update message only if its generation number is greater than that of the previous update message it received. But we feel that in a real system it is very unlikely that messages will get that out of date before being acted upon. Likewise we feel that in a real system no special precautions would be needed to ensure that assumption 3 is correct. A monitor, when it arrives at a site, must have some work to perform before it can move again. The original code segment of a pure domain can move as soon as it arrives at a site, but as it leaves behind a copy it would soon run out of sites where it was required. Also communication subsystems may well transmit c-requests and t-requests with higher priority than whole code and public segments because the former are likely to be very short.

It might be supposed that it would be better for a kernel to broadcast the identity of the next site in the chain as soon as it had made the reservation of the domain for that site. Subsequent t-requests could then be sent to the end of the chain with considerable savings in overheads compared with the strategy we have outlined, where normally t-requests will arrive at the top of the chain and have to be passed through every site on the chain to the end of the chain. Disregarding the likelihood of long chains building up as being very small, an equivalent scheme would indeed be possible for associative type global management. But in the case of updating directories, such a scheme leads to the

violation of our assumption 2 and could give rise to a self-contained loop of reservations for a domain. So the extra overhead of passing t-requests down each site in the chain is unavoidable if we are to use a directory update scheme for locating domains.

## SECTION 2: THE 'BEST' SITE CALCULATION.

In the first part of this chapter our concern has been to show that, once it has been decided that a domain incarnation will take place at a particular site, the domain will actually move to that site. We now look at the decision procedure for determining at what site a domain incarnation will take place.

The determination of where the next domain incarnation is to take place is the keystone of our distributed system. If domain incarnations are moved around the system too frequently, virtual processors will be subject to extra transmission time delays and the communication subsystem may become overloaded. If movement is too infrequent then the loading at different sites can be become seriously unbalanced, some sites idle while others are choked with work.

In discussing the DCS system in chapter 4 we criticized open bidding for work by all sites as taking too long and putting too great a load on the

communication subsystem. Our approach can be likened to a single tender policy. An attempt is made to identify a site that will run a domain incarnation quickly and preferably with minimum demands placed upon the communication subsystem. This site, the 'best' site, is passed the entry capability for the domain incarnation. The situation at this site, and to a lesser extent in the rest of the distributed system, may have altered significantly between the time of generation of the information upon which the decision was made, and the time when the kernel examines the newly arrived entry capability. So if the 'best' site, on the information available to it, calculates that another site would be a substantially better choice, then it sends off the entry capability to that site. This site also has the freedom to accept or reject the domain incarnation. (A maximum number of transfers can be set to stop the domain becoming a 'hot potato').

So there are two types of calculation to be described, one to nominate the initial 'best' site and the other to decide to accept or reject the nomination. We assume that each site has a table of the number of virtual processors at each site in the 'ready to run' queue, or some similar measure of outstanding work, and a table of the amount of free memory at each site. This information would be gathered from the exchange of status information described in chapter 3.

The initial calculation:

The initial calculation at the code site considers (normally) only three possible sites as candidates for 'best' site.

1) the site of the code and possible public segment
2) the site of the processor base segment and any local segments that are parameters; since they were part of the domain incarnation being exited, the parameters will be at the same site as the processor base segment
3) the site of any other local segments which, since they were all last used in the previous incarnation of the domain being entered, will all be at one site.

If the code segment cannot move because it is 'tied down' to drive a peripheral attached to a particular site then that site is chosen as 'best' site. Otherwise the basic policy is for the code site kernel to consider the total size, at each distinct site, of segments for the domain incarnation. The site with the largest aggregate size is chosen as the 'best' site. So, when all the segments are at the same site that site is chosen, and otherwise transmission on the communication links is minimized.

Minimization of communication bandwidth requirements is only one of a number of criteria that can be

considered. Before the aggregate sizes are compared there are a number of biases that can be applied to them.

1) Since the calculation takes place at the code segment site it is possible to know how many other virtual processors are using the domain (i.e their domain incarnations are in the 'ready to run' queue). If the domain were to go to another site then either these domain incarnations would have to follow (if they are re-entered again) or the domain must return to the original site. Hence the size of the domain (the code and possible public segments) can be biased upwards by a factor representing its outstanding work, encouraging incarnations for the same domain to be all at the same site.
2) The load, that is the number of ready to run domain incarnations, at each of the three possible sites can be taken into account. If the domain incarnation is sent to a lightly loaded site it will be executed quicker than at a heavily loaded site. And if transmission times are less than average execution times, substantial advantages accrue by choosing an idle site rather than a site with even one other domain incarnation to run. Thus sizes can be biased downwards by a factor representing the overall load at a site.
3) A kernel is likely to spend a lot of its time on memory management if it has very little free memory left. Not only should attempts be made to balance

loads at sites but also an attempt should be made to ensure that a site does not run out of memory space when other sites have plenty available. So the sizes of segments can be biased upwards proportional to the amount of free memory at their site. This will encourage migration away from sites with little free space.

4) A good part of a computation may consist of repeated calls between the same pair of domains, or, more generally, a repeated sequence of calls. If the domains reside at different sites and all are larger in size than the processor base segment then the processor base segment could continually travel between the sites concerned. Obviously the computation would proceed faster if all the domains were at one site. If a 'shadow' stack is kept with the processor base segment it is easy to generate a count of how many consecutive interdomain jumps have been made in the same sequence of calls. If this count is used to bias upwards the size of the processor base segment then eventually all the domains involved will come to the same site. In particular domains that call 'tied down' peripheral handler domains will tend to migrate to the site of the handler domain. Of course, if a computation uses two or more peripherals controlled from different sites then the processor base segment, together with parameter segments, is doomed to traverse back and forward between sites. This observation lends weight

to the desirability of the workstation approach to controlling peripherals (chapter 7).

The effects of the first stage calculation can be summarized as follows:

It effects at most three sites and it tends to balance the load and free memory of these sites.

It tends to aggregate all incarnations of one domain at one site.

It tends to localize to one site domains that are used together.

Other things being equal, it minimizes the load on the communication subsystem.

The second stage calculation:

The first stage calculation we have described above only takes into consideration a maximum of three sites, and nominates one of them 'best' site. A site that is completely idle would have no way of breaking into the circle of the elect if only this calculation were used. The criteria we use for accepting or rejecting the entry capability at the 'best' site overcomes this problem.

Assuming that the domain is not 'tied down' at its site, the 'best' site kernel scans the table it has of the loads at other sites and if some other site is substantially less busy than itself (and, according to

the free space table, has the space to accomodate the domain incarnation) then that site will be nominated as 'best' site. If the kernel does not find a substantially less busy site then it checks to see if its own site has enough space not only for the incoming segments but also for any temporary local segments that have to be created. If it does have the space it accepts the entry capability, requests any external segments to be sent to it, and eventually schedules the domain incarnation ready to run.

If the kernel does not have enough space then a second, less critical, scan of the load and free memory tables is made to find a site that does have enough space. If no such site can be found, or the domain is tied down to the present site, then the incarnation is placed in a queue of incarnations waiting for space. It is when the distributed system has sites that have reached this stage that the danger of deadlock becomes real. In the case of an incarnation of a domain that is tied down, the kernel could scan its ready to run queue and invalidate the entry capability of an incarnation of a domain that is not tied down, in the hope that this incarnation will move to another site and so release space for the tied down domain incarnation.

The effects of the second stage calculation can be summarized as follows:

At low loading of the system it has little effect,

there is no point in moving computations to idle sites if they leave an idle site behind.

At moderate loading of the system the policy is to try to keep all sites busy.

At heavy loading of the system the concern is more with finding space for incarnations.

## SECTION 3: REMARKS ON DOMAINS IN DISTRIBUTED SYSTEMS.

This chapter concludes the description of our kernel/domain model for distributed systems presented in chapter 6. Together these two chapters extend the field of application of domains. Hitherto domain management using capabilities has been centred around single site systems. This is because central tables have been required to implement the capability mechanism. By restricting the sharing of segments and providing a global object management scheme to cover the essential sharing required, we have been able to dispense with central tables and hence distribute domain management. Our main goal in achieving this extention has been to facilitate dynamic and efficient load sharing but our model can equally well be used to provide, in distributed systems, the protection normally associated with domains in single site systems.

# CHAPTER 9

# DESCRIPTION OF SIMULATION

The developers of the DELTA language, a successor to SIMULA 67, say in an introduction to a definition of the language,

> 'Computer simulation has become an important methodological tool in the study of systems. Quite often, the actual simulation model runs on the computer provide useful information. What is nearly always useful, however, is the effort invested in writing the simulation program. This work requires a careful attention to both the main structure of a system and to its details. The result is often an understanding which makes the later computer analysis less important in comparison.' [HOLB75].

We have written a simulation of a distributed system. Our main purpose in writing it was to make sure that our concept of the requirements for a distributed system was complete. Our major interest was to learn more of the problems of distributed computing rather than to accurately predict performance. Nevertheless there were two semi-quantitative questions of considerable importance that we wished our simulation to provide

guidance upon. These were:

1) What would be an adequate bandwidth, approximately, to support what could be a considerable movement of segments?

2) Would increasing the number of sites in the system give close enough to a linear increase in power, or would overheads swamp the modest decrease in response time predicted in chapter 2 (figure 2.9)?

But perhaps the question of greatest concern to us was that of stability. We wanted to determine if we would achieve stable load balancing using the various strategies we have outlined for distributed control. The demonstration that stability could be achieved in a simulation would bode well for its achievement in an actual implementation.

The system we simulated uses directory updating for global object management and appends status information to each message (see chapter 3). We discuss the results of our simulation in chapter 10. The actual simulation program and some sample outputs are reproduced as appendix A.

Choice of language:

We chose SIMULA 67 [DAHL72] as the language in which to write our simulation program. The CLASS concept of this language allows for the easy and flexible definition of objects in the simulation. The built-in CLASS of Simulation provides facilities for linking objects into lists and primitives for handling the flow of time. These features are expanded upon in a tutorial paper by Ichbiah and Morse [ICHB74]. We assume familiarity with the language in the rest of the chapter.

When we came to use SIMULA 67 we found it had a number of drawbacks. The first is that it uses a single precision real variable for representing time. In a computer system's simulation the span of times that are of interest ranges from instruction execution times, of the order of one microsecond, to say a total simulation time of two hours. With 7 digits significance, as for the IBM 360 version of SIMULA, lack of differentiation [HUTC68] of events would occur after the simulated time reached 10 seconds. In fact in our simulation the smallest period of interest was around 100 microseconds and, because of the expense of computer time to run the simulation, its duration (in simulation time) was around 1000 seconds, so we just avoided the problem.

Our second problem arose from a language rule that classes can only be used as a prefix at the level at

which they are declared (or in the case of system classes at the level enclosed by the simulation block).

Declarations of the form

```
Simulation BEGIN
  Process CLASS communication_system;
  . . . .
  CLASS site;
    Process CLASS kernel;  ... ;
    Process CLASS clock; ... ;
    . . . .
  END of class site;
  . . . .
END of simulation block;
```

are illegal!

All objects that make up a site have to be declared at the outer level, including all objects that are linked into queues inside the kernel. This has produced a large separation in the program between the declaration of objects and their use. The program structure is not clarified by this separation.

Another problem was that none of the implementations of SIMULA 67 that we had access to, IBM SIMULA versions 02.03 and 04.00 on IBM 360/65 and IBM 370/168 under OS, and Dec-10 SIMULA KA version 1C, could garbage collect correctly. This meant that the creation of new objects during the simulation had to be severely curtailed so that garbage collection would be invoked infrequently;

thus lessening the chance of garbage collection being invoked in a situation that the implementation could not handle. This restriction, while perhaps aiding the run time of the simulation, has led to considerable artificiality in the program. For example, instead of control messages being created when required, acted upon and abandoned (to be garbage collected) instances of every type of control message have to be created at the start of the simulation and the same instance altered for each individual message.

The implementations had other faults also and we finally abandoned work with the IBM versions. We should note however the high compatibility between the languages accepted on the various machines. We moved our program from IBM to Dec versions and it immediately compiled. We also performed the reverse move at a later date, and with the assistance of a short conversion program, again achieved immediate compilation.

The basic structure of the simulation:

SIMULA 67 simulations use the Process class to describe entities in the simulation that are active. In our simulation there are six types of process. Each process is activated by some process (except for initiation), performs some actions, which can include the activation of other processes, and then passivates

itself. The activation of a process by another can be specified to take place immediately, or after the actions of the activating process are complete, or with a simulated time delay. Figure 9.1 depicts this simple description. The six types of process are:

1) Process CLASS kernelc;

   (lines 573 to 1918 in the program in appendix A).

   Each kernelc instance simulates the behaviour of a site and its kernel.

2) Process CLASS clockc; (lines 3112 to 3139).

   For each site there is a clock 'ts_clock', which can be set to interrupt the site.

3) Process CLASS s_channelc; (lines 2994 to 3029).

   Each site has one s_channelc instance to handle the transmission of messages to other sites.

4) Process CLASS consolec; (lines 3035 to 3094).

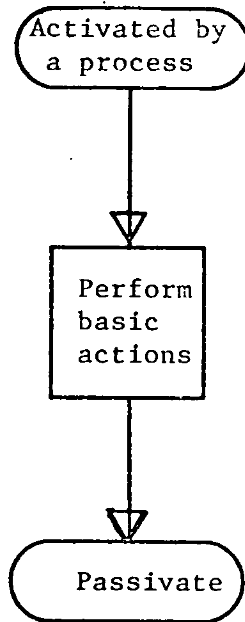
   This class simulates the action of a user, presenting the distributed system with work, waiting for the work to be performed, waiting for a 'thinktime' period and then presenting another request for service from the system.

5) Process CLASS diskc; (lines 3271 to 3319).

   There is a instance of this class for every disk in the distributed system being simulated. The function of this class is to define the delays in accessing information on a disk.

6) Process CLASS disk_controllerc; (lines 3151 to 3263).

   There is one instance of this class for every disk. It deals with communication between the disk and




Sequencing in a quasi-parallel process

Figure 9.1

sites, and does some buffering of requests for use of the disk.

Segments:

The basic entity of the domain oriented system we have described in the previous chapters is the segment. The simulation however uses more basic entities than this. All objects that are to be queued in the system have to be Link CLASS objects. All objects that are transferred between sites, that is inter-kernel control messages and segments, belong to the Link CLASS contentc (line 530). The attributes of a contentc object are a length, an origin site, a destination site and status information of the origin (for the updating of the destination network status tables described in chapter 3).

A segment is one of the subclasses of the contentc class (line 546). Additional attributes include site and key to the hash organised segment table at that site. Each of the types of segment in our model are represented as subclasses of the segment class. Thus we have:

segmentc CLASS domainc; (line 2061).

The extra attributes of this class are an identification number or name (did) and an indication (tied) as to whether or not the domain must remain at one site always. This class also defines a set of kernel calls such as requests for interdomain jumps.

domainc CLASS monitorc; (line 2161).

This class distinguishes the sizes of the two segments that make up the basis of a monitor, c_size being the size of the code segment and db_size being the size of the public segment. The two segments are always treated as a single unit in the simulation.

monitorc CLASS secretaryc; (line 2172).

This class corresponds to monitors with condition queues (chapter7). Its extra attribute is a queue, my_q, and it provides procedures for 'first in first out' manipulation of the queue. It also provides procedures for the creation of a secretary processor and control procedures for it.

segmentc CLASS virtual_processor; (line 2227).

The attributes of this class include information for the kernel, such as priority, a stack of entry capabilities and space for parameters.

segmentc CLASS local_seg; (line 2328).

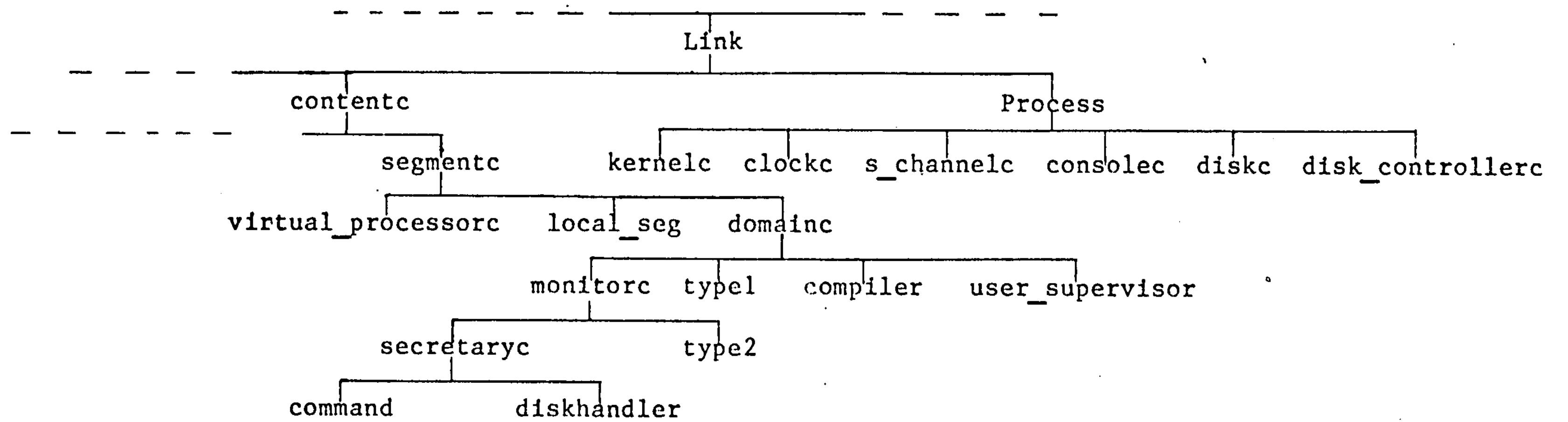
This class includes a procedure for moving local segments from entry capabilities to A_lists. The handling of local segment capabilities is not as general as the description given in chapter 7, mainly because of difficulties arising from not being able to dynamically create and delete either local segments or entry capabilities.

The relationship of the various classes in the simulation is depicted in figure 9.2.

The action of the kernel:

The basic action of the kernel is to examine entries one at a time from the 'ready to run' queue, called driverq in the simulation (line 1623). Its action depends on the type of the entry. When driverq is empty the kernel is idle and so passivates; some other process class object, usually in the communication subsystem, has to activate the kernel again, which is done by calling the procedure switch_context (line 1528) when it has placed a new entry in driverq.

The queue driverq has in fact 4 priority levels; 'high', 'monitor', 'medium' and 'low'. The class definition and associated procedures are given in lines 217 to 284. At each priority level entries are queued first come first served. Entries at the 'high' level pre-empt kernel attention from lower priority levels. After execution of all tasks at the 'high' priority level, execution of the the first entry in the next highest priority, non empty queue takes place. Since all entries to 'monitor', 'medium' and 'low' are the result of executing 'high' priority tasks the effect is that each level pre-empts lower levels. The use of the various levels is as follows:




THE CLASS HIERARCHY OF OUR SIMULATION

Figure 9.2

high: All messages requiring kernel attention including putative entry capabilities for domain incarnations, but excluding valid ready to run domain incarnations, are queued at this level. Any entry arriving on this queue pre-empts kernel attention from lower priority levels, by executing the switch_context procedure if necessary.

monitor: All valid (i.e. ready to run) incarnations of monitors are queued here. Since this is the highest level of valid domain incarnations once an incarnation is at the top of this queue it will remain there until it gives up control to the kernel, ensuring the exclusive access we discussed in chapter 7. For although any high priority message will cause execution of the monitor incarnation to be interrupted, there is no way to put a valid incarnation ahead of the interrupted one in driverq. So when the interruption has been dealt with the original monitor incarnation will be resumed.

medium and low: All other valid domain incarnations are queued at these two levels. When a virtual processor has had a period of service greater than 'longtimeslice' since it last interacted with a console, it is moved from medium priority to low priority.

Although we have not used such a category we suspect that in a real system a 'top' priority may be required for initial loading and dealing with catastrophic

failures.

The kernel contains some general sections of code to assist it in handling the various driverq entries. These sections are:

Memory management. (lines 640 to 776)

The program does not simulate any particular memory management policy. It just keeps count of how much free space there is and increases the simulated time taken to grant space when there is not much free space. The memory management section handles the possible queue of domain incarnations waiting for space. Two queues are kept, one for small requests which are given priority, and one for larger requests. Included also in this section is the procedure 'make_space' which scans information about domains to delete copies of pure domain code segments when space is scarce.

Segment management. (lines 778 to 846)

This section provides procedures for adding, deleting and retrieving segments at a site. A hash function is used on the unique identifiers of segments (key) so that the segment table at each site can be kept reasonably compact. The simulation blurs the distinction between a capability for a segment and the segment itself. A capability contains all the attributes of a segment and thus is analogous to an

instance of the class segmentc, which contains all the attributes of a segment as well. Thus, for example, there are always exactly two references pointing to an instance of a local_seg. One, stored in an entry capability, represents the capability for the local segment, the other one, stored in a segment table (except when the segment is being moved from site to site), represents the segment itself.

Communications interface. (lines 848 to 941)

This section of the kernel interacts with the communication subsystem. When a message arrives the status information it contains about the sender is examined and the message is placed in driverq. In the case of the message being a segment, the segment is registered in the segment table.

The kernel places all messages it wants to send to other sites in one of two queues. The higher priority queue is for short control messages, the other is for segments. Each time the s_channelc process has finished transmitting a message it interrupts the kernel, by placing a message in driverq and calling switch_context. The kernel releases the space the message occupied (except perhaps when the message was a pure domain code segment) and initiates the transmission of another message if there are any to be sent.

Broadcasting is performed by placing individual messages for all the other sites in the send queue and

transmitting them serially.

Load monitoring. (lines 943 to 1025)

Part of the heuristics for avoiding deadlock (chapter 7) are contained in this section. Its main purpose is to monitor the amount of free memory at all sites, including its own, and maintain a boolean 'overload' which it sets when the free memory in the total system has diminished past a critical amount. The procedure 'ootimum_site' is used to assist in the second stage 'best' site calculation for domain incarnations.

Domain management. (lines 1027 to 1526)

The various procedures in this section implement the strategies outlined in chapter 8.

The other main procedures in the kernel are switch_context, which we have already described, and execute. The procedure execute (line 1543) simulates the execution of a domain incarnation. Since domain incarnations can be interrupted the simulation of their execution requires some care. Our simulation program allows the action of each domain to be broken into as many as five steps, and associates with each step an execution time. The kernel process passivates itself for the execution time of the step and then 'instantaneously' executes the step. If the action of the step does not result in the termination of the domain incarnation (that

is its removal from driverq) the kernel process passivates for the duration of the execution time of the next step after which it 'instantaneously' executes that step, and so on. During any period of the kernel being passivated it can be activated by another process executing the switch context procedure. Since the interruption may result in another domain incarnation being placed in driverq at higher priority than the incarnation whose simulated execution was interrupted, the name of the next step and the time remaining until it was due to be executed (runtt) are stored with the entry capability for the domain incarnation. Thus simulated execution of the domain incarnation can take place at any time.

The entries in driverq:

We now describe in more detail what actions the kernel takes for the various sorts of messages it finds in driverq.

Entry capabilities. (line 1653)

The kernel checks to see if the entry capability has been validated. If so, control is passed to the code segment of the domain (simulated by invoking the procedure execute). If the entry capability is not valid the kernel initiates action to make it valid using the procedures in the domain management section.

Time slice interrupt. (line 1676)

Virtual processors are subject to time slicing. When a timer interrupt occurs the kernel continues with the current domain incarnation (giving it a new quantum of processing time) only if the domain is a monitor or the kernel has no other work to do and the domain has not been reserved for another site. Otherwise the entry capability is invalidated so that the domain incarnation will end up last in the medium or low queue (perhaps at a different site), depending on how much service time the virtual processor has received since it last interacted with a console.

Message from a console. (line 1705)

If the secretary processor for handling console input (see later) is not already scheduled then it is placed in driverq. The message is placed in a special queue for the attention of the secretary processor when its incarnation is run.

Request to transfer a domain to another site. (line 1719)

The kernel carries out the action described in the section on domain management in chapter 8. If it has the domain, and has not reserved it for another site, it reserves the domain for the requesting site and checks if it can send the domain off immediately. If the domain is already reserved then the request is sent on to the site that has the reservation,

otherwise the request is sent to the site supposed by the kernel to have the domain.

Arrival of a domain. (line 1750)

A domain is placed in driverq when it arrives from another site having been previously requested (except for initial program loading). The kernel places into driverq all entry capabilities that were waiting for the arrival of the domain so that their 'best' site calculation could be performed. It also checks to see if the domain is the last outstanding external segment for any domain incarnation otherwise ready to run. If so, that incarnation is also placed in driverq, at the appropriate priority level.

Domain change of site update. (line 1796)

The kernel registers the changed site in the information it keeps about every domain.

Request for local segments. (line 1801)

The kernel prepares to send the segments off immediately.

Request for processor base segment and parameters. (line 1814)

Again the kernel sends off the required segments immediately.

End of message interrupt from s_channelc. (line 1829)

The kernel initiates the transmission of the next message if there are any queued. If the previous message was a segment it frees the space occupied by it. In the case of pure domain code segments a test is carried out to see whether to keep a copy or not.

Arrival of processor base or local segment.
(line 1853)

The arrival is always the result of a previous request, made to another site, that the segment be sent. The domain incarnation to which the segment belongs is determined, and if it is the last segment required for the domain incarnation the domain incarnation is placed in driverq.

Hopefully we have built up a picture of how kernels co-operate to make the distributed system run. Their chief action is of course to execute domain incarnations but such executions give rise to many different types of messages to be passed between kernels. Kernels deal with incoming messages as fast as they can so that other sites will not be held up unduly.

Action of domains:

Kernels do not provide most of the facilities of an operating system, that is the function of the various domains in the system. In our program we have simulated the action of four areas of an operating system; command analysis, diskhandling, compiling and user program supervision. We have also provided some unspecified domains and monitors that simulate the resource requirements of other operating system facilities. Details of the domains are as follows:

domainc CLASS type1; (line 2408)

This class has no specific purpose. Its possible actions (determined on a probabilistic basis) are to call another domain of this or the type2 class (see below) and to call the diskhandler domain to read a buffer from disk. An incarnation of a domain of type1 class has two local segments, one of which is passed as a parameter to the diskhandler domain. In our simulation we used ten instances of this class of domain intending to represent areas of an operating system that handle various sorts of trivial requests (c.f. the number of domains in the SUE system nucleus [SEVC72], chapter 5).

monitorc CLASS type2; (line 2493)

The function of this class is similar to type1. Instances of this class handle trivial requests that

involve the use of system wide tables. Its action is simpler than type1, consisting of processing followed by a return. Each incarnation has one local segment. The simulation program in appendix A has five instances of this class of domain.

domainc CLASS compiler; (line 2530)

The system we simulated is assumed to have two compilers. The compiler class of domain makes substantial demands for processing power and makes large transfers to and from disk. It has two local segments, one of which is used as a buffer for disk transfers.

domainc CLASS user_supervisor; (line 2589)

This class simulates the 'interpretation' of user programs. We use one instance of this class in our simulation, but multiple instances could be used to simulate different supervisors available to different users. The supervisor domain has three local segments, one for user code, one for data and the third is a disk buffer for reads and writes to disk. The action of a user program is assumed to consist of a cycle of processing followed by disk request.

secretaryc CLASS command; (line 2886)

Each virtual processor has a random number seed associated with it. This seed is used (and updated) by the code of domains when it is desired to simulate

different behaviour for different users. The command domain is entered when a request for service is received from a console. It determines, using the random number seed, what domain (from a choice of user-supervisor, 2 compilers or any of the trivial command domains) the virtual processor will enter next. When execution in the chosen domain is complete the virtual processor returns to this domain whence the controlling console is notified that service is finished, and the virtual processor is suspended until another request is received from the console.

The secretary processor associated with this domain executes different code from all other virtual processors. Its function is to choose the correct virtual processor, from among those suspended, to respond to a request for service from a console. The kernel at the site that receives messages from consoles (assumed always to be the same site) schedules the command secretary processor for execution on receipt of a console message. Once it runs, the secretary processor schedules an incarnation of the command domain by the virtual processor associated with the console.

The command domain is tied down so that it can communicate with consoles. It does not have any local segments but does, of course, have a public segment. This domain is one of the two types in our simulation that co-operate with the kernels to try to ensure stability and freedom from deadlock in the system.

Since all virtual processors pass through this domain, a count is maintained of how many there are active in the total system. If this number exceeds a certain figure or the boolean 'overload' at the local site is set, then only trivial commands are allowed to proceed. The rest are held up until the number of virtual processors is reduced. This scheme is similar to that described by Wilkes [WILK73], where processes have to move from a waiting list to an accepted list before they are eligible to be considered for running. Our scheme, in examining the nature of a request for service before placing it in an 'accepted' or 'waiting' state, produces better response time characteristics for requests that are known to be small because they involve specific domains.

secretaryc CLASS diskhandler; (line 2661)

This domain is quite complex. In conjunction with its associated disk_controllerc process it performs the actions described in chapter 7, in the section on limited capacity controllers. All requests for reads or writes for the disk belonging to the diskhandler domain are programmed as interdomain jumps to the diskhandler domain. In the domain, writes to the disk are dealt with immediately, the descriptor type capability for the segment to be written is placed in a queue (wq) and the return to the calling incarnation is made. If wq is full then the incarnation is suspended instead, on a condition queue, wql or wqh,

depending on the priority of the virtual processor. As wq empties, by segments being transferred to the disk, entries are removed from wqh first, since it has incarnations of medium priority virtual processors, and then from wql. The virtual processors are allowed to resume their incarnations in the calling domains. Thus there is limited buffering of disk writes, but a virtual processor cannot fill up the distributed system with segments destined for the disk.

Since a virtual processor doing a write is not usually held up, a virtual processor entering the domain to read from disk is normally given priority. As we mentioned in chapter 7, the entry capability for the resumption of the calling domain is prepared so that there will be no delay when the read has been performed. This entry capability is stored in either the rqh or rql queues depending on whether the priority of the virtual processor is medium or low. If the boolean 'transfer_in_progress' is false then the virtual processor can initiate its own read or write operation. Otherwise this is the function of the secretary processor. Its incarnation of the domain is made ready to run every time the kernel receives a message from the disk_controllerc process indicating that it has a free buffer (so that a read or write can take place). Actually the form of this message is simply the secretary's entry capability for the domain.

The diskhandler domain also contributes to the

maintainence of system stability. If the 'overload' boolean at its site is set true then the normal ordering of reads before writes is reversed to free space. Also virtual processors that have low priority (i.e. have been running for a long time since their last console interaction) will be suspended after a write request (in my_q) until the overload conditon has been overcome, whence they will be released one at a time.

We should point out that the fixed number of buffers in the disk controller together with the fixed amount of memory in the system could be a fertile source of deadlocks. One of the segments (with status 'ondisk') of a domain incarnation being resumed after a disk read, occupies a disk buffer. Free space at a site is required before the segment can be transferred to the site, the buffer freed and the domain incarnation permitted to proceed. The incarnations of other virtual processors occupy memory at sites and, if they need to do a disk read or write, require a disk buffer to be free before they can proceed.

Statistics:

SIMULA 67 is well suited for the gathering of relevant statistics in a simulation. The procedures we used are defined in the statistics section of the program (lines 286 to 528). These procedures were designed so that

whatever the number of sites, users, disks etc. in a system the program would automatically generate correctly annotated statistics. Most of the last part of the program (line 3321 onwards) is concerned with the setting up of these statistics. An example of the output produced is given at the end of appendix A. The simulation was allowed to run for a simulated time of 'settle_time' after which all the statistic counters and timers were set to zero. Then the simulation was continued for a period 'sim_time' when all the accumulated statistics were output and the simulation terminated. The amount of CPU time used on a Dec-10 KA system varied between about 10 and 50 minutes per simulation run, depending on how many consoles and how many sites were being simulated.

Performance parameters:

We complete this chapter by summarizing the configuration we simulated and giving figures for the simulated load presented by consoles. The simulated distributed system consists of N sites directly interconnected and, for the transmission of segments, directly connected to the one or several disks in the system. Each site has primary memory only. All consoles (M of them) are controlled from one site and each disk, when there is more than one, is controlled from a separate site.

Adams and Millard have published figures for the load presented to EMAS [ADAM75]. They give distributions for the service times required for important classes of work; compilations and the running of compiled programs. We arbitrarily decided that each site in our distributed system would execute programs at one quarter of the speed of the EMAS central processors. So the times we give here are quadruple those given in ADAM75.

EMAS has two different compilers in common use so we used two instances of the compiler class in our simulation. The mean time for a compilation is 20 seconds. (The complete histogram of compilation times is defined by the arrays A, cumulative probability, and B, compile time, at line 2580 of the program).

The mean execution time for a user program is 24 seconds. (The histogram is given by the arrays ueserp, cumulative probability, and usert, execution time, at line 161 of the program). In this case we did not quite follow the distribution given by Adams and Millard. Their distribution was biased by a few long execution times and would have required us to cater for executions of up to 480 seconds. We imposed a cut off at 180 seconds. The pragmatic reason for this is that 480 seconds is about the duration of a simulation run so that the statistics from a run including such a request would be considerably distorted. But there is also a deeper justification for our action. We believe that the

performance of a system affects the characteristics of the load offered to it. The person who submits the equivalent 120 second job to EMAS probably runs it at times of slack demand so that a high fraction of total processing power is devoted to the job and the resulting response time is satisfactory. In our distributed system the best that can be done with a 480 second job is to dedicate one site to executing it so that the response time is necessarily much longer than 120 seconds. Users will thus be discouraged from running long jobs.

Seventy four percent of all commands issued to EMAS are of the trivial kind. This high proportion of trivial requests in EMAS also supports our assertion that performance affects the load presented. EMAS responds well to short commands (and our simulation is designed to give good response to them also). If everyone had to wait an average of say 40 seconds to find out how many users were logged into the system, or to have the time printed out, then they would not make such requests very often. Adams and Millard do not give distributions for trivial command execution times. The strategy we picked for the operation of type1 and type2 domains (see above) was chosen more to exercise the interdomain call mechanism than reflect reality. A trivial command in our simulation can involve up to 18 interdomain jumps. The execution time of a trivial command averages almost 200 milliseconds and is approximately negative exponentially distributed.

The following table gives, for our simulation, the number of each type of console request as a percentage of the total number of requests for service (see lines 2905 to 2909). It also shows how 'useful' physical processor time (i.e. ignoring overheads and idle time) is distributed among the categories.

Distribution of command types

| Type | relative frequency | % cpu time |
|---|---|---|
| Compilations | 9 | 31 |
| User executions | 17 | 67 |
| Trivial | 74 | 2 |

Table 9.1

The overall average time to execute a command is just over 6 seconds.

Although the average think time (including console output and input time) is reputed to be 35 seconds on EMAS, which agrees exactly with that reported elsewhere on other interactive systems [SCHE67,ESTR67], we have used a negative exponential think time with a mean of 30 seconds. We wanted to include each interaction with a text editor as a trivial command by itself and so shortened the average think time to compensate for a higher interaction rate for editing.

The other timings in the simulation have been arbitrarily chosen. By and large we assumed that the

hardware constituting a distributed system would be efficient at performing domain management type tasks.

When it came to specifying the other resource requirements of domains, namely the amount of space they require and the number and frequency of disk requests, we had no guidance from published sources. Agrawala and colleagues have recently published a study [AGRA76] correlating cpu demands with memory requirements and I/O to disk (among other factors) but there is no way of deducing from the categorization of jobs (done by cluster analysis) which domains would belong to what category.

Each site in the simulation is assumed to have 128,000 bytes of memory, 4,000 bytes of which is occupied by the kernel. This amount of memory at a site could be considered large, but there is no point in simulating the addition of extra sites to a distributed system when the addition of extra memory at each of the existing sites would produce the same results. The size of a processor base segment is made to be 200 bytes.

The following table (9.2) lists the size (or range of sizes) in bytes, of incarnations of the various domains (assuming 20 users, as some public segment sizes are determined by the number of users).

Sizes of Domain Incarnations

| Incarnation of ... | Bytes |
|---|---|
| type1 | 1480-3144 |
| type2 | 1430-3950 |
| command | 3122 |
| diskhandler | 2760 |
| compiler | 17632-17760 |
| user supervisor | 13192-36392 |

Table 9.2

The total size of all domains and processor base segments, before any virtual procesor enters any domain is 52,032 bytes for a twenty user system.

If a trivial request goes to the maximum depth of 4 type1 domain calls and performs a disk read at each level it will require about 20,000 bytes of memory in total for all its domain incarnations, and will invoke a kernel to change domains 18 times. Thus even trivial requests can place quite substantial demands on the resources of the simulated distributed system.

When it came to disk usage we arbitrarily decided that one in four calls of type1 domains would involve a disk read. For compiling we assumed an average compiling speed of 12 lines a second and, relating this to the size of buffer used, fixed the I/O to disk as a pair of requests, a write followed closely by a read, on average once a second. The assumed distribution of the interval

between the write/read pairs is 6 stage Erlang (line 2556). For user programs the mean headway between disk I/O requests, roughly two thirds reads and one third writes, was assumed to be 250 milliseconds. This is twice what the equivalent rate on EMAS is thought to be. This is to compensate for the fact that a user program in a distributed system that had only primary memory, would probably be restricted in size (in the simulation 32,000 bytes is the largest size that user code, data and buffers can occupy) and so would make more transfers to disk than in EMAS, which is a virtual memory system with drum backing store.

We freely admit that many of our performance parameters have been rather arbitrarily chosen. But we are in neither the business of detailed workload construction nor of high resolution performance evaluation. As we have indicated we believe the characteristics of a distributed system will affect the nature of the workload presented to the system. The only way to accurately estimate the workload, as well as determine the number and size of domains, is to actually implement such a system. We feel that the results we present in the next chapter show the practicality of building a domain orientated distributed system that will perform useful work.

# CHAPTER 10

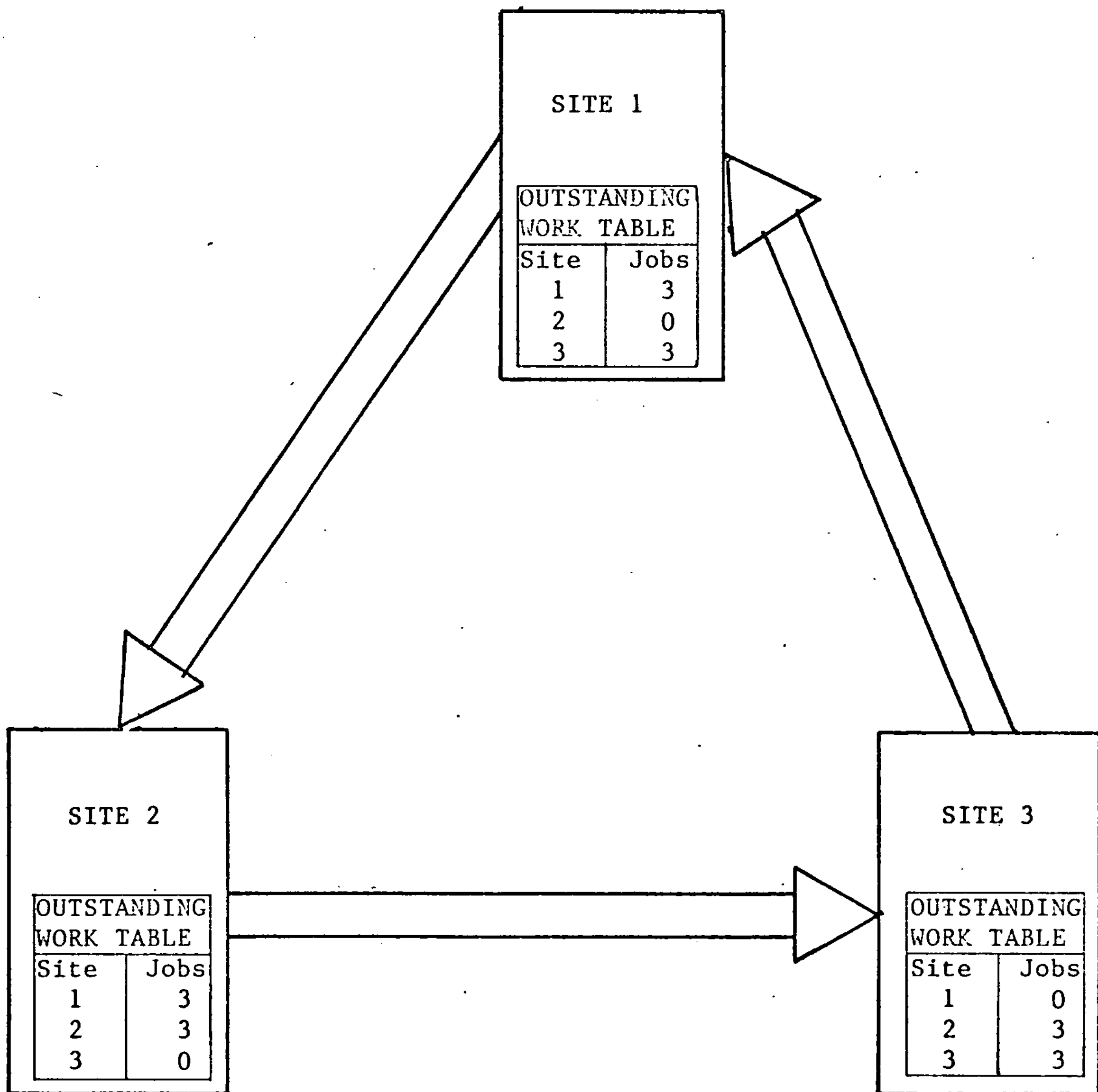
# RESULTS OF OUR SIMULATION

## General experience:

The main result of our simulation, a deeper understanding of the requirements of distributed systems has, we hope, been displayed in the earlier chapters. One lesson that was quickly brought home to us by runs of early versions of our simulation program was the necessity to keep all sites as busy as possible. Since for many types of communication subsystem the total bandwidth does not increase as the number of sites in the system goes up, our original 'best' site calculation used minimization of segment transmission between sites as its main criterion, so as to conserve bandwidth. However simulation runs showed that this produced widely disparate utilization of processors, with consequent longish queues at the well patronised sites. We quickly introduced a factor in the initial 'best' site calculation so that a site with no work to do would almost always be chosen as 'best' site when the other sites involved had other work. This, of course, accords with the ideal of instantaneous jockeying and considerably narrowed the range of physical processor utilizations, as shown, for example, in the 'IDLE TIME' figures in the example outputs at the end of appendix A.

Another notion of which we were quickly disabused was that keeping copies of pure code domains would not result in significant gains. There is a lot of extra work required to treat pure domains differently from monitors. However, simulations with and without the sharing of pure domain code segments showed substantially decreased loads on the communication subsystem when the segments were copied. Also since there were only three domains, the two compilers and the user supervisor, that received really heavy usage the existence of copies meant that the load could be spread more evenly when there was more than three sites in the network.

The accidental retention of some tracing statements in a full simulation run led us to restrict the number of times an entry capability could be passed from site to site before actually resulting in a domain incarnation. The simulation was of a three site system and the trace output degenerated to a constant pattern towards the end of the simulation. Investigation revealed that the status information that each site held had become so arranged that all messages flowed one way around the communications system (see figure 10.1). The information each site had about the succeeding site was well out of date and indicated, falsely, that the site was underutilized. So all requests for a domain incarnation to take place at the site were refused, the succeeding site was nominated as 'best' site and the request passed to it. We were aware of the possibility of such a loop




Note: Each site's own entry of its own outstanding work is the correct value. This value is appended to all messages sent from the site.

OUTSTANDING WORK TABLES
IN A 3 SITE SYSTEM
GIVING RISE TO A SITUATION
WHERE ALL MESSAGES TRAVEL
IN ONE DIRECTION

Figure 10.1

forming, though we considered it highly unlikely. We assumed that the completion of a domain at a site would probably result in a message being sent in the counter-flow direction. This message would have update information about the sending site so that the loop would be broken. What appeared to be happening though, was that the kernels were so busy pushing around their rejected domain incarnations that they had no time to do any useful work, and so complete any domain incarnations already running at their sites. The forcing of a site to accept a domain incarnation after it has been through a fixed number of sites has its unfortunate aspects but it does lead to the quick breakdown of any loops.

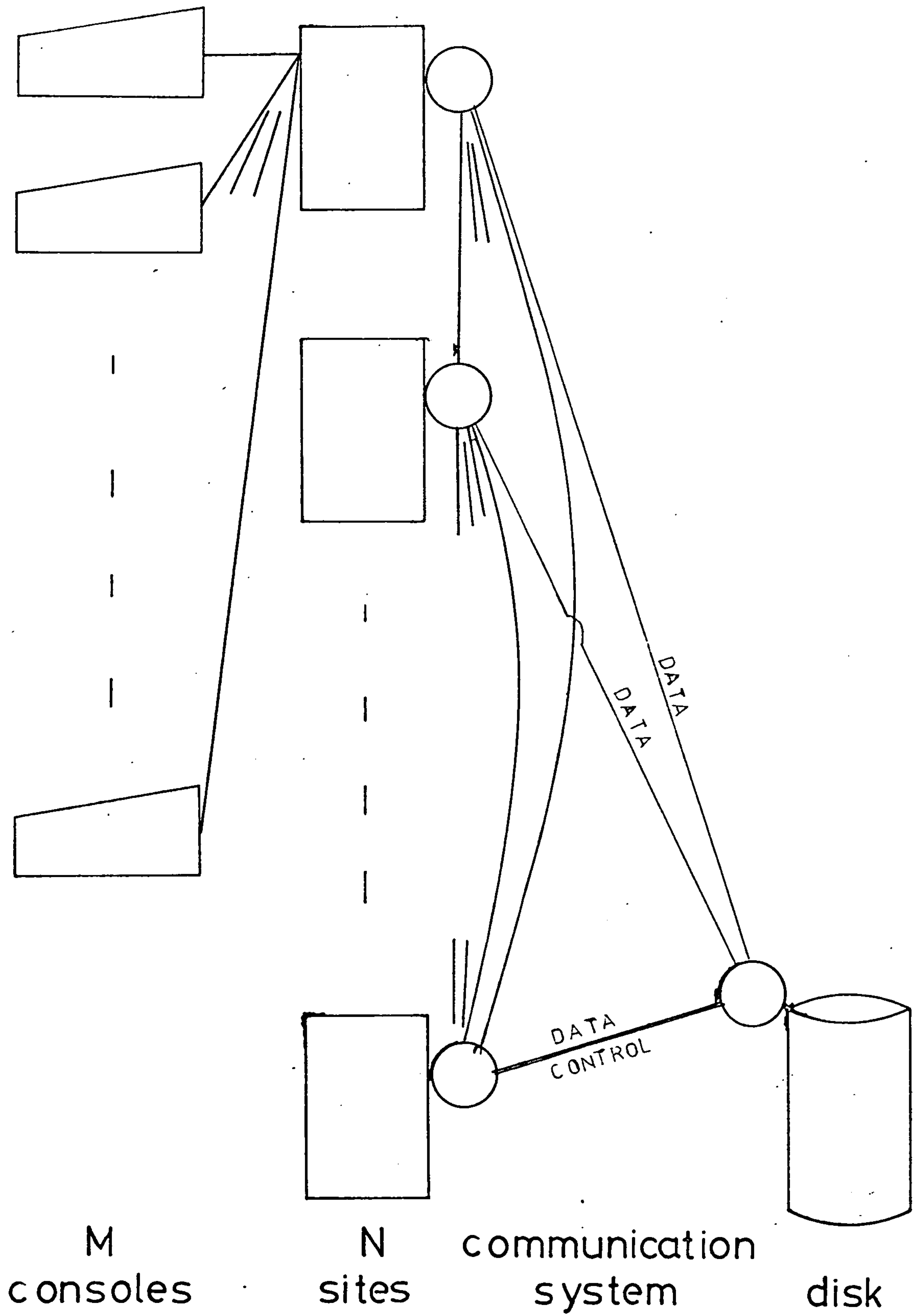
These few examples suffice to show the qualitative benefits of the simulation program. We now go on to present and discuss the semi-quantitative aspects of the simulation.

Performance measures:

The initial simulation runs were to determine suitable values for the various 'twiddle factors' to ensure both system stability and high throughput. That such values, valid for a wide spread of system sizes and loads, did indeed exist is very encouraging. After these tuning runs 3 series of simulation runs were performed. The

load characteristics were varied by using a different random number seed for each series. In each series of runs the number of users (consoles), M, was varied from 4 to 24 in steps of 4 and the number of sites, N, was varied from 1 to 6. One disk was simulated in all these runs. The configuration that was simulated is depicted in figure 10.2. In one series of runs the simulation time, T (sim_time in the program), was 1000 seconds, with a prior settle_time of 200 seconds. In the other two series the simulation time was 500 seconds, with a settle_time of 200 seconds in one series and 300 seconds in the other. (The extra 100 seconds did not make any noticeable difference to the results so we assumed that 200 seconds was sufficient to damp down transients caused by there being no work outstanding in the distributed system when simulation started).

The chief measurements made during the simulation were response time and service time. Each virtual processor corresponding to a user (i.e. not the secretary processors associated with console serving and disk handling) kept a tally of how much service time it received and how long it was active in the system, the total time T less all periods of 'thinking time'. These tallies were zeroed at the start of the period T and were recorded at the end of the period.




INITIAL SIMULATED CONFIGURATION
Figure 10.2

Thus the statistic total service time (T.S.T) can be defined by

$$T.S.T = \sum_{i}^{M} \int_{T} s_i(t).dt$$

where $s_i(t) = \begin{cases} 1 & \text{if any domain incarnation of virtual processor } i \text{ is being executed at time } t \\ 0 & \text{otherwise} \end{cases}$

(this is printed as GRAND TOTAL OF SERVICE TIMES in the output example in appendix A).

The total response time (T.R.T) is similarly defined by

$$T.R.T = \sum_{i}^{M} \int_{T} r_i(t).dt$$

where $r_i(t) = \begin{cases} 0 & \text{if the console associated with virtual processor } i \text{ is in the thinking state} \\ 1 & \text{otherwise} \end{cases}$

(this is printed as GRAND TOTAL OF RESPONSE TIMES in appendix A).

From these two statistics were calculated two more;

a response factor, RF, given by

$$RF = T.R.T \;/\; T.S.T$$

that is the overall ratio of response time to service time, whicn is also the ratio of the mean response time

to mean service time (this is given as PERFORMANCE MEASURE in appendix A).

an average processor utilization, U, given by

$$U = T.S.T / (N \times T)$$

Since there are overheads associated with kernel operations and secretary processors which do not appear in T.S.T a value of 1 for U is impossible. Table 10.1 gives the overall response factor for the three series of simulations and figure 10.3 depicts this information graphically (T.R.T and T.S.T were both totalled over the 3 runs before their ratio was taken). Table 10.2 gives the overall average useful work done in each configuration, U*N (again with the numerator and denominator of U being separately totalled first) and figure 10.4 gives a graphical representation of the informatiom.

RESPONSE FACTOR

| | Consoles | | | | | |
|---|---|---|---|---|---|---|
| Sites | 4 | 8 | 12 | 16 | 20 | 24 |
| 1 | 1.82 | 3.65 | 7.38 | 11.55 | 15.50 | 19.32 |
| 2 | 1.45 | 1.80 | 2.77 | 4.34 | 5.51 | 8.23 |
| 3 | 1.36 | 1.52 | 1.80 | 2.51 | 3.02 | 4.02 |
| 4 | 1.33 | 1.42 | 1.59 | 1.98 | 2.25 | 2.74 |
| 5 | 1.33 | 1.39 | 1.50 | 1.74 | 1.95 | 2.29 |
| 6 | 1.33 | 1.37 | 1.46 | 1.63 | 1.78 | 2.07 |

Table 10.1

WORK DONE - EQUIVALENT NUMBER OF PROCESSORS

| | Consoles | | | | | |
|---|---|---|---|---|---|---|
| Sites | 4 | 8 | 12 | 16 | 20 | 24 |
| 1 | 0.60 | 0.94 | 0.98 | 0.98 | 0.97 | 0.97 |
| 2 | 0.71 | 1.30 | 1.70 | 1.89 | 1.94 | 1.96 |
| 3 | 0.73 | 1.39 | 1.99 | 2.48 | 2.68 | 2.83 |
| 4 | 0.74 | 1.42 | 2.05 | 2.71 | 3.06 | 3.42 |
| 5 | 0.74 | 1.43 | 2.09 | 2.84 | 3.23 | 3.71 |
| 6 | 0.74 | 1.44 | 2.11 | 2.91 | 3.35 | 3.85 |

Table 10.2

RESPONSE FACTOR
AS A FUNCTION OF
THE NUMBER OF SITES
AND CONSOLES

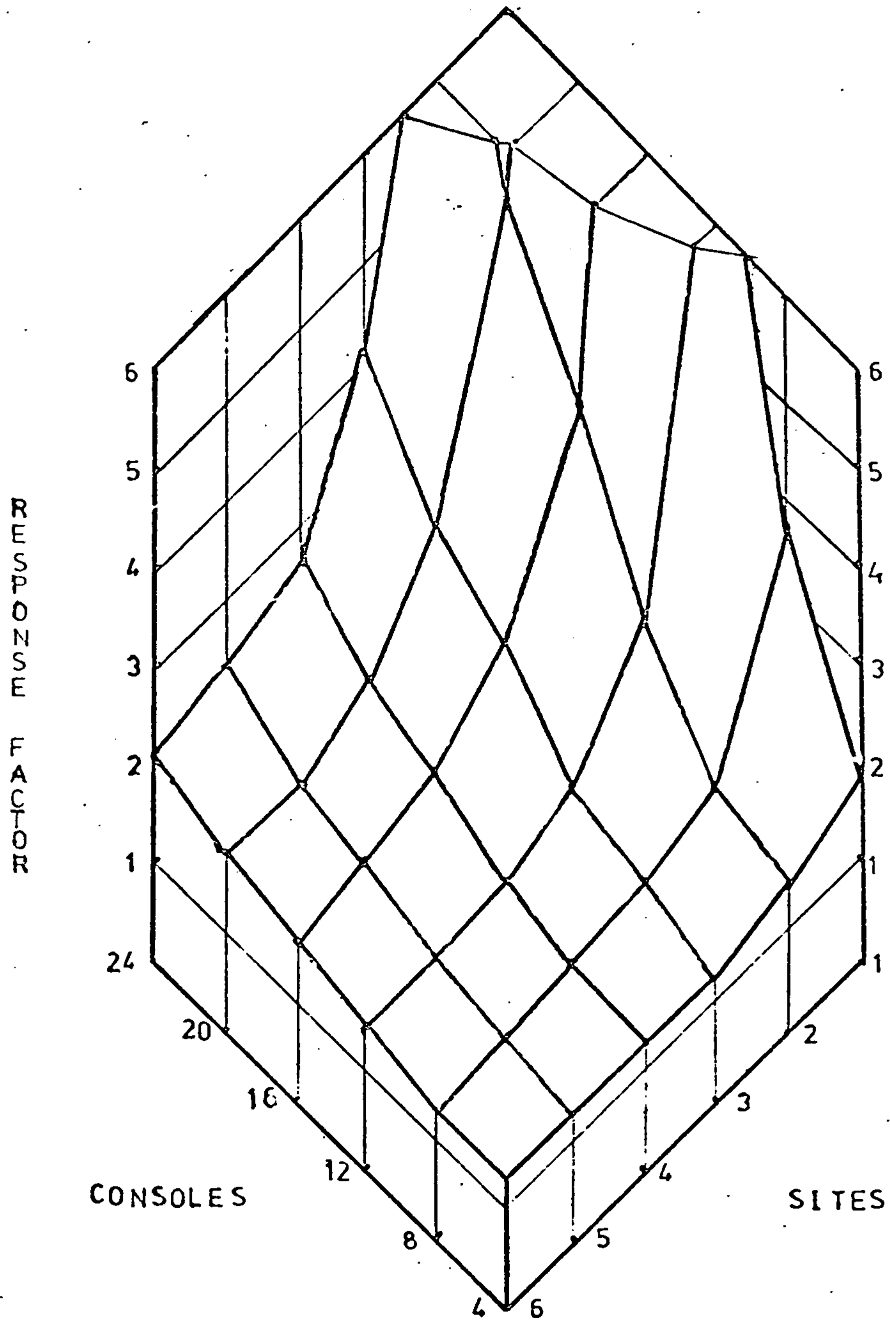



Figure 10.3

THROUGHPUT
(as the fraction of the capacity of one site)
AS A FUNCTION OF
THE NUMBER OF SITES
AND CONSOLES

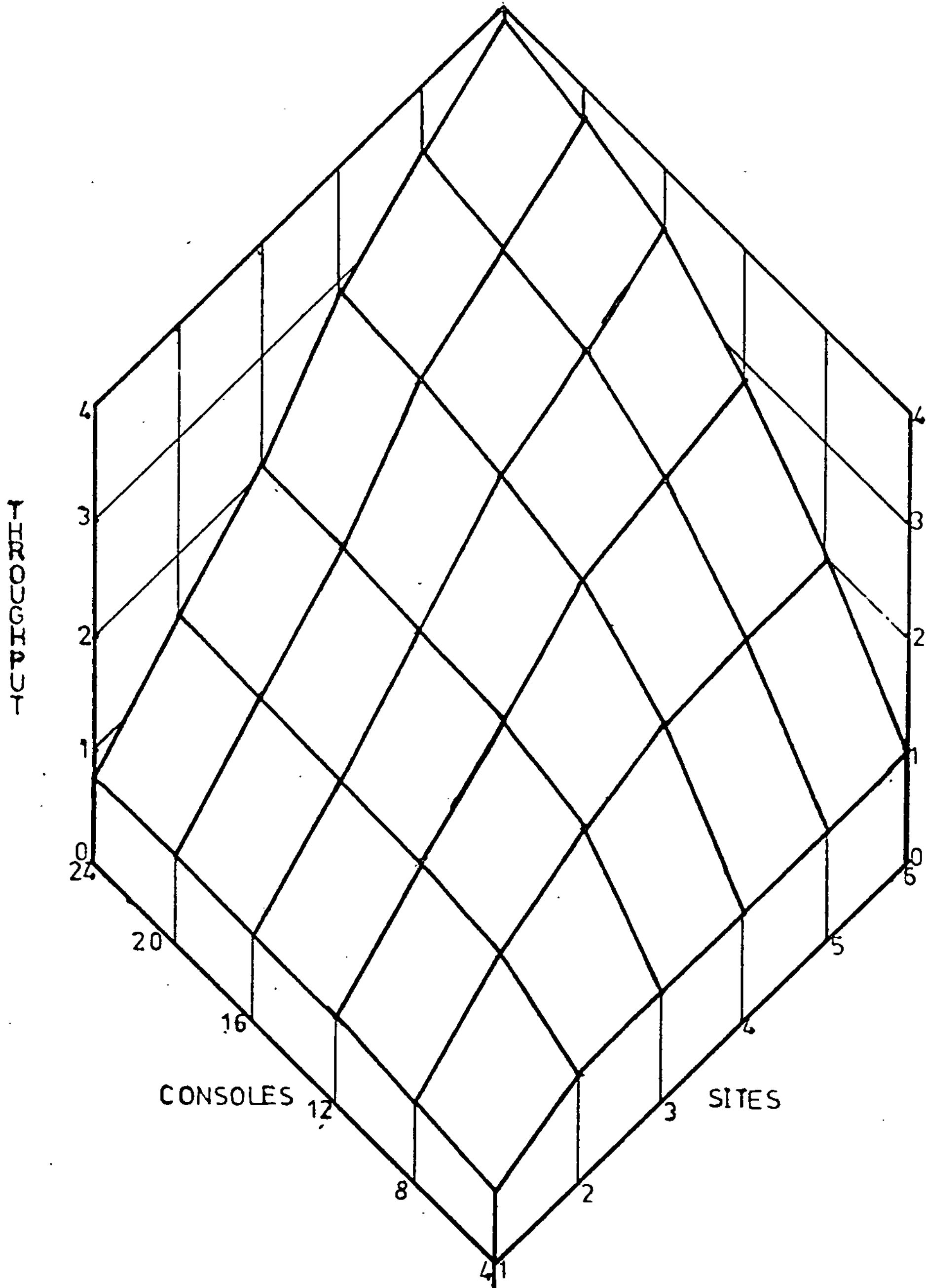



Figure 10.4

The next three tables give some other figures derived from the simulations.

Table 10.3 shows the number of bytes transmitted per second over all communication links, including those to disk.

Table 10.4 gives the number overall, and per site, of control messages transmitted per second for the various configurations.

Table 10.5 gives the utilization of the disk, that is the ratio of the total time it was carrying out a read or write (including seeking) to the overall simulation time.

BANDWIDTH - Kilobytes per second

| | Consoles | | | | | |
|---|---|---|---|---|---|---|
| Sites | 4 | 8 | 12 | 16 | 20 | 24 |
| 1 | 13 | 21 | 23 | 25 | 27 | 26 |
| 2 | 18 | 33 | 49 | 56 | 59 | 58 |
| 3 | 19 | 42 | 64 | 89 | 95 | 105 |
| 4 | 20 | 44 | 72 | 105 | 125 | 142 |
| 5 | 19 | 43 | 73 | 112 | 137 | 165 |
| 6 | 19 | 41 | 73 | 114 | 139 | 174 |

Table 10.3

CONTROL MESSAGES PER SECOND (: PER SITE)

| | Consoles | | | | | |
|---|---|---|---|---|---|---|
| Sites | 4 | 8 | 12 | 16 | 20 | 24 |
| 1 | 4:4 | 7:7 | 7:7 | 8:8 | 8:8 | 8:8 |
| 2 | 9:5 | 17:8 | 25:12 | 30:15 | 31:15 | 32:16 |
| 3 | 11:4 | 22:7 | 35:12 | 47:16 | 50:17 | 56:19 |
| 4 | 11:3 | 24:6 | 41:10 | 57:14 | 65:16 | 74:19 |
| 5 | 11:2 | 24:5 | 43:9 | 63:13 | 74:15 | 86:17 |
| 6 | 11:2 | 24:4 | 43:7 | 66:11 | 79:13 | 95:16 |

Table 10.4

% DISK UTILIZATION

| | Consoles | | | | | |
|---|---|---|---|---|---|---|
| Sites | 4 | 8 | 12 | 16 | 20 | 24 |
| 1 | 11 | 15 | 17 | 18 | 19 | 18 |
| 2 | 12 | 21 | 28 | 32 | 34 | 34 |
| 3 | 12 | 23 | 33 | 45 | 46 | 50 |
| 4 | 12 | 23 | 34 | 47 | 53 | 60 |
| 5 | 12 | 24 | 35 | 49 | 56 | 65 |
| 6 | 12 | 24 | 36 | 50 | 58 | 68 |

Table 10.5

Analysis:

Queueing theory considerations [KLEI68,KLEI76] stipulate that the response factor curve must lie above and to the left of two asymptotes;

$$RF = 1 \qquad \text{for } M \ll M'$$

and

$$RF = 1 + (M - M')/N \qquad \text{for } M \gg M'$$

where $M'$ is the saturation number of consoles, given by

$M'/N$ =

(mean service time + mean think time)/(mean service time)

From the figures given in chapter 9, $M'$ is 6*N.

Figure 10.5 is a re-presentation of the data in table 10.1 for N = 1, 2 and 3, with the corresponding asymptotes. The correct position and indeed close fitting to the asymptotes gives us confidence that our simulation is not wildly erroneous.

The tables and diagrams we have presented show that

* increasing the number of sites increases the throughput and reduces the response factor for a fixed number of consoles but both effects level off (when there are so many sites that all requests from consoles can be met without any queues forming).
* increasing the number of consoles without increasing the number of sites leads to greater throughput and a higher response factor, the response factor grows very fast when the throughput approaches the total capacity

RESPONSE CURVES AND ASYMPTOTES

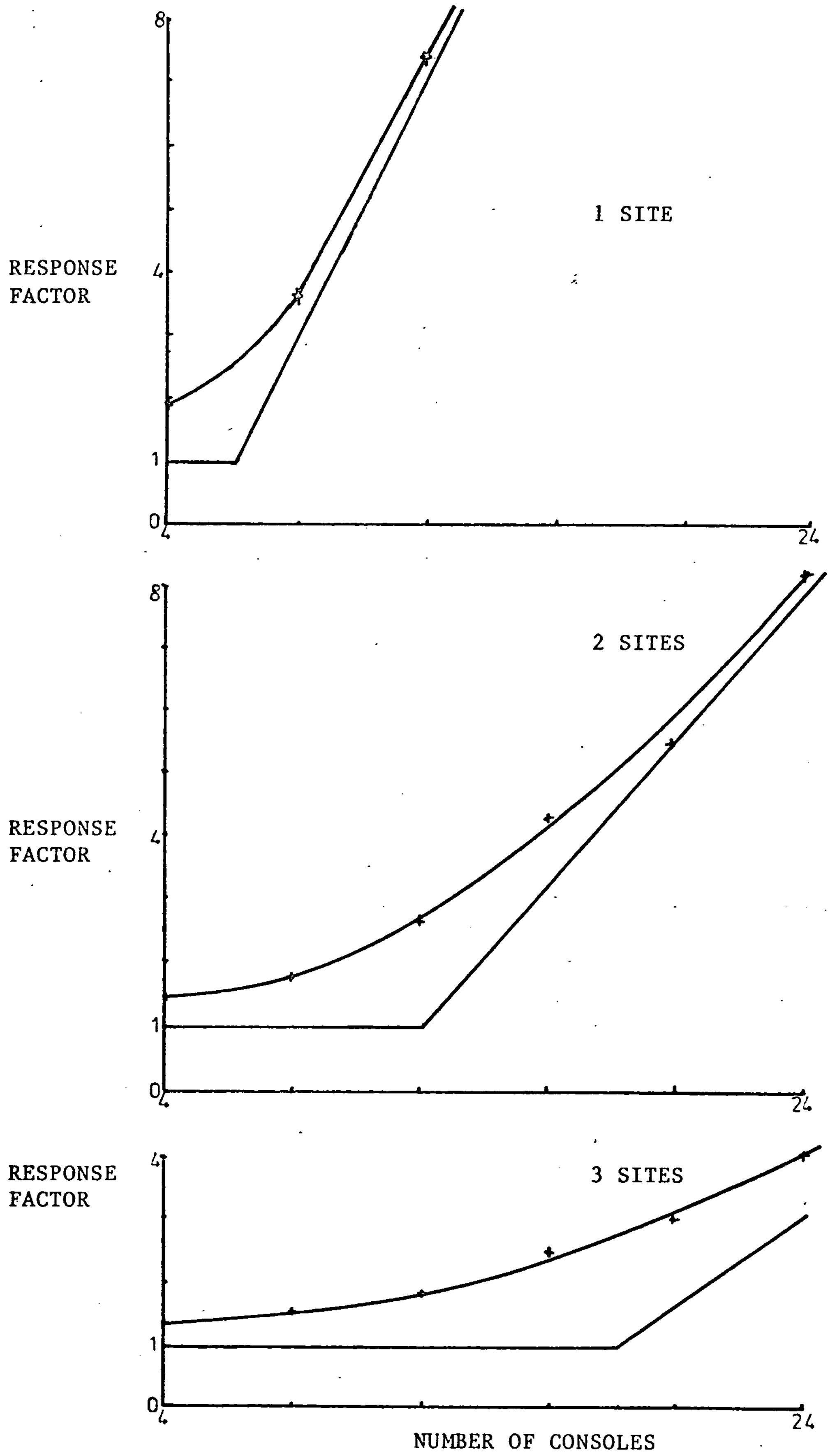



Figure 10.5

of the sites and the throughput actually drops slightly when the system is grossly overloaded (presumably because 'wasted' processor power is required to cope with the congestion).

* for a constant ratio of consoles to sites the number of control messages (indicative of general network management overheads) increases at a greater than linear rate with increasing size of system.
* at the size of systems considered, control messages (each 32 bytes) account for less than 2% of the total bandwidth. Otherwise use of the available bandwidth grows linearly with increasing size of the system (except, obviously, for the jump from one site to two sites because a system with one site only uses the communication subsystem for accessing the disk).
* disk usage is correlated with throughput, which is to be expected; roughly 6 seconds of processing gives rise to 1 seconds worth of disk utilization.

Response factors:

The presentation of data in figures 10.1 and 10.2 is too coarse to determine the effect that the number of sites has on the relation between utilization, or throughput, and response factor. Figure 10.7 gives the response factor as a function of average processor utilization for two series of simulation runs with 3, 6 and 9 sites and overall 4, 6 and 8 consoles per site.

These simulation runs differed from the previous runs in that firstly, there was 1 disk per 3 sites for each system and secondly, the period of simulation depended on the number of consoles being simulated (see lines 131 and 158 of the program in appendix A). The reasons for the differences can be appreciated by referring to figure 10.6 which is an equivalent graph of response factor versus utilization for 1 and 6 sites, derived from the data from the first series of simulations (augmented by more runs for the N=1 case to give the low utilization figures). The N=6 curve starts to break away upwards from the N=1 curve when the utilization is only 0.6. This is because with 6 sites the single disk is the 'bottle neck' in the system, rather than the processors, so that the response factor is predicated by the disk utilization. One disk per three sites is adequate disk capacity so that processor utilization is the the chief determinant of the response factor in the later series of simulations. Also notice that the variance, or spread, of points from the N=1 curve is large in figure 10.6. This we realized, was because the number of console interactions in a fixed period of simulation is smaller when there are few consoles and sites than when there are many consoles and sites, and consequently the variance of estimates made from the results of the interactions must be larger. Hence, to get equal variance independent of the number of sites, the simulation should be conducted for the same number of interactions, which is roughly proportional to the number of consoles.

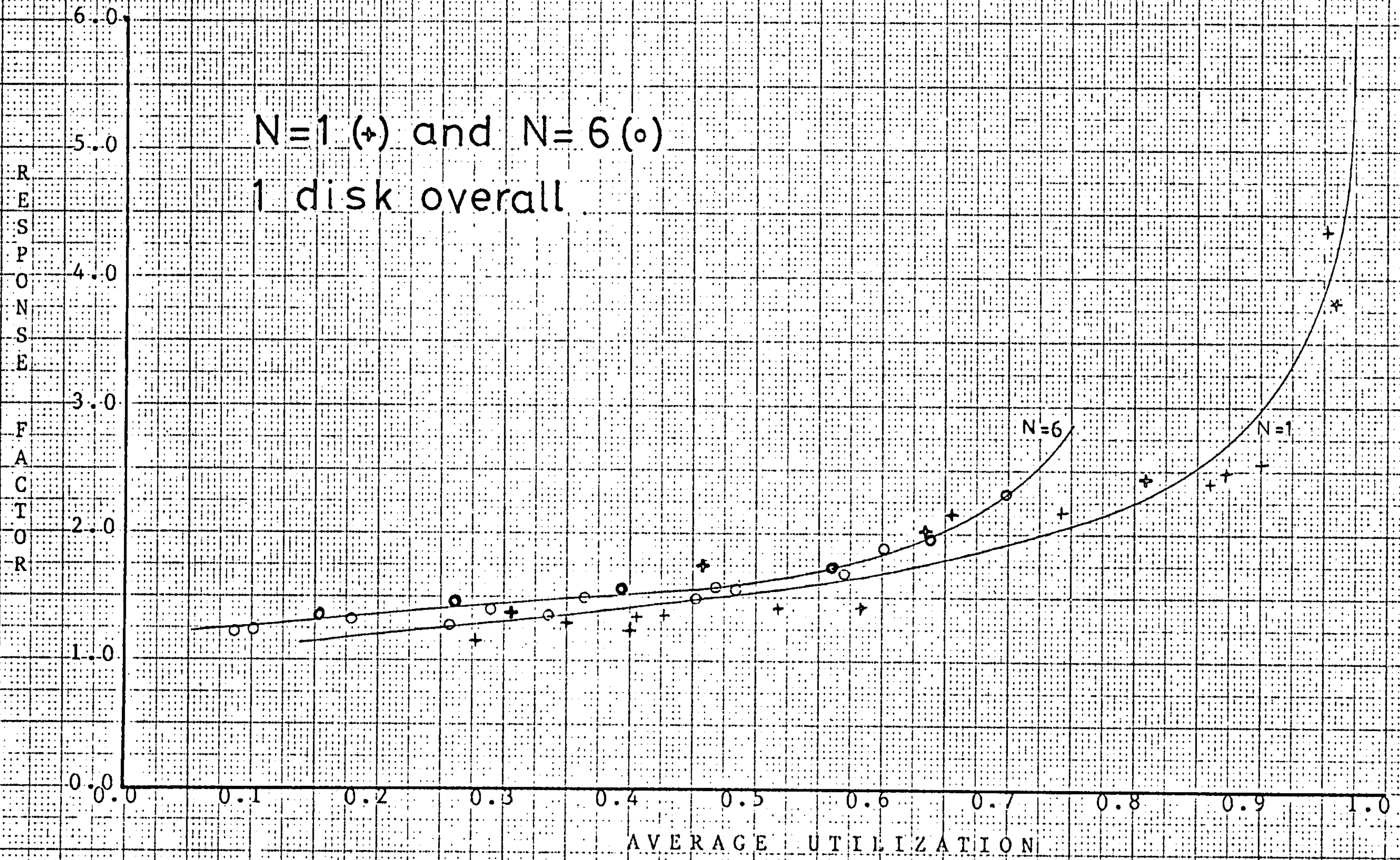



Figure 10.6

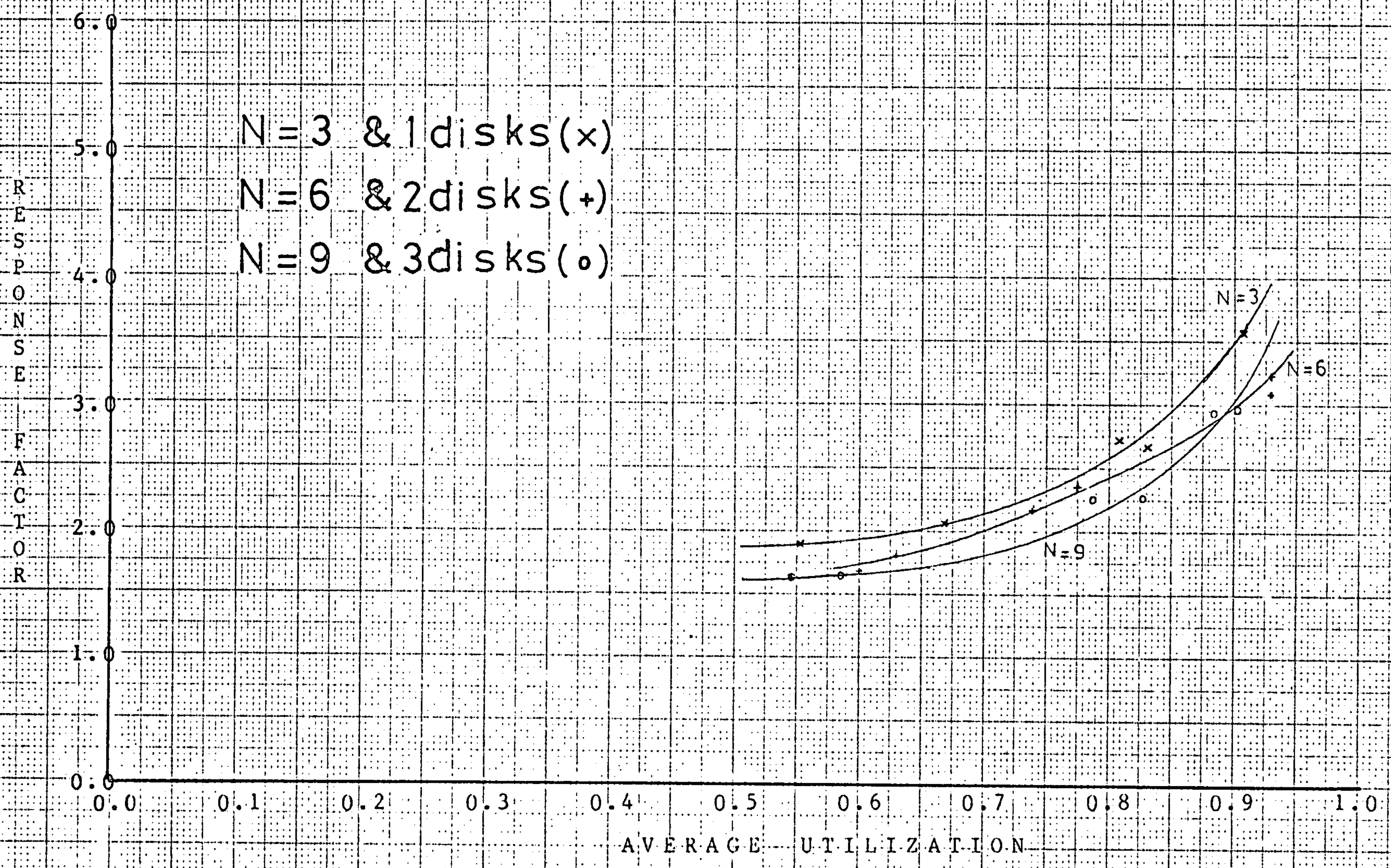



Figure 10.7

Returning to figure 10.7 we see that there is a slight decrease in response time in the region of processor utilization, 0.6 to 0.9, most likely to be operated in. (The cross-over of the N=9 and N=6 curves, when the utilization is 0.9, is a reflection of the proportionally higher overheads in the 9 site system; see later). The gain predicted in chapter 2 (cf figure 2.9) occurs even though the simulated service time is not exponentially distributed and the service discipline is not first come first served from a common queue. However this gain will only occur provided that no other resource, such as available bandwidth, saturates as the size of the distributed system is increased. Given this constraint though, the behaviour of the system is very encouraging. The system builder has some leeway to use strategies that involve overheads at each site that increase with the number of sites, and yet still attain approximately linear increases in throughput (for a constant response factor) with system expansion. There must be a limit to this process however because the response time is ultimately constrained to be at least the service time.

Bandwidth requirements:

As we mentioned earlier the simulated communication subsystem is a directly connected one but each site can only transmit to one other site at a time. Hence the effective bandwidth available in the system increases

linearly with the number of sites (including disks). In both the sets of simulations runs described above the capacity of the links from each site was fixed at 1 MHz or, equivalently, 125 Kilobytes per second.

Another series of simulations were run for a system of three sites, 18 consoles and one disk, and a system with nine sites, 54 consoles and three disks, when the capacity of the links was varied, in powers of ten, between 0.01 MHz to 10 MHz. Table 10.6 gives figures from this series for the response factor, average throughput per processor (as a fraction of the theoretical maximum) and the bandwidth used, both in absolute terms and as a fraction of the available bandwidth.

EFFECTS OF VARYING BANDWIDTH

| Link MHz | Sites | Response factor | Processor utilization | Bandwidth used (MHz) | Fraction of total |
|---|---|---|---|---|---|
| 10.0 | 3 | 2.27 | 0.87 | 0.763 | 0.019 |
| | 9 | 1.75 | 0.86 | 2.468 | 0.021 |
| 1.0 | 3 | 2.66 | 0.83 | 0.793 | 0.198 |
| | 9 | 2.26 | 0.79 | 2.494 | 0.208 |
| 0.1 | 3 | 36.3 | 0.13 | 0.156 | 0.389 |
| | 9 | 23.7 | 0.16 | 0.583 | 0.486 |
| 0.01 | 3 | 338 | 0.02 | 0.016 | 0.396 |
| | 9 | 218 | 0.02 | 0.067 | 0.565 |

Table 10.6

Table 10.6 shows that 1 MHz available bandwidth per site is adequate but that anything less leads to a severe degradation of response factor and throughput. An increase in available bandwidth, above 1 MHz per site, gives a small increase in performance. Note that directly connecting sites is a relatively inefficient way of using bandwidth; at 0.01 Mhz per link the intercomputer links in the 9 site system are only 57% utilized even though the communication subsystem is a substantial bottleneck. Indeed the same argument we used for processors in chapter 2, that it is desirable to have 1 server with capacity C rather than N servers with capacity C/N applies to communication subsystems as well. So a bus or loop communication subsystem will have a lower total bandwidth requirement than a directly connected subsystem because of the more efficient utilization of the available bandwidth.

Extrapolations:

Lack of memory space on the computer used to perform the simulations prevented simulating systems with more than 9 sites. We would have liked to increase the number of sites further to determine if there is a practical upper limit to the system size after which the throughput drops or even becomes zero. We suspect that there is such a limit.

From the data given in table 10.4 for systems with a ratio of 4 consoles to a site (i.e the diagonal of the table) the best fit quadratic for the number of messages as a function of the number of sites, N, is

$$\text{control messages/sec.} = 1.1\ N^2 + 10.4\ N - 6.1$$

For a system of twenty sites and eighty consoles we could predict 650 control messages a second, presenting a communication bandwidth requirement of 20 Kilobytes per second. A one hundred site system with four hundred consoles would require a bandwidth of approximately 400 Kilobytes per second (3.2 MHz) just to the control messages. Each site in such a system would receive a control message on average every 8 milliseconds. The gain shown in figure 10.6 for increasing sizes of system cannot offset this squared growth. The overheads associated with dealing with the control messages would substantially reduce the capacity of each site to perform useful work so that saturation, defined above, would occur with fewer consoles and the response factor would be increased. As the throughput falls at each site so, in general, will the number of control messages issued by the site. Thus it is possible that some form of equilibrium state will be reached where the adding of new sites has no effect on the total throughput of the system. But equally it is possible that distributed system management will take up more and more of the total processing power of the system as sites are added, until eventually the whole system is just dedicated to managing itself and can do no useful work.

Examining the total bandwidth used, as presented in table 10.3, we see that after subtracting the load due to control messages (each 32 bytes long) the growth of bandwidth used is reasonably constant at 35 Kilobytes per second for each increment, after the first, of one site and four consoles. Thus the bandwidth used by a system with 20 sites and 80 consoles would be around 700 Kilobytes per second or 5.6 MHz. From table 10.6 a total bandwidth available of 25 MHz would be adequate to support this load, probably a lot less would be required if a bus or loop type communication subsystem is used. One hundred sites would use a bandwidth of 35 MHz. The designers of Ethernet [METC76] anticipate no problems in increasing the present capacity from 3 MHz to 15 MHz, sufficient probably for a 20 site system, but still totally inadequate for a 100 site system. Indeed probably the only way of realizing the required bandwidth for a 100 site system is to directly connect the sites which requires 4950 links, rather impractical.

To our knowledge, in our simulation of up to 9 sites, no domain became a bottleneck. Monitor domains can only be at one site at a time. The larger the system, the greater the risk that one of them will be in continuous demand and so hold back the whole system. Predictions as to when this will occur require figures on the use of individual domains. In a hundred site system no monitor can have an overall average use of more than 1% of processor time if bottlenecks are to be avoided.

There is another reason to think that a distributed system of 100 sites would never be implemented along the lines we have described in this thesis. With 400 to 800 active consoles, by the law of large numbers, the load presented to the system would be very smooth. Thus functional specialization of sites is appropriate. Throughout this thesis we have upheld the principle of keeping processors general purpose so that they can deal with random variations in the nature of the load presented. But when the load is almost deterministic this principle does not apply, provided that the relative numbers of each type of functional unit matches exactly the characteristics of the load. Functional specialization should reduce the management overheads [REYL74]. As we showed in chapter 7 functional specialization does slot neatly into our system as the size of the system goes up.

So for both technical reasons and theoretical reasons we feel that the ultimate size of a distributed system based on the kernel/domain architecture will be around 20 or so sites. Based on the results we have obtained simulating systems with up to 9 sites, we make the prediction that a system of 20 sites will have 20 times the throughput of a system with 1 site and will be able to maintain the same response time characteristics.

Summary:

The simulation program achieved it goals, as outlined in chapter 9.

* In writing the program and analysing the results we gained a deeper knowledge of the requirements of distributed computing.
* The existence of results shows that it is possible to define control strategies that minimize the chances of deadlock, eliminate load levelling thrashing and yet permit useful work to be done.
* The simulation results show that the bandwidth used by our distributed system is in the order of 3 MHz for the larger systems simulated, which is about that provided in other local networks such as DCS [FARB72c]. Thus the bandwidth requirements are not impossibly high.
* Within the range of 1 to 9 sites, and over the operational range of processor utilizations, the simulation results show that there is a modest decrease in response time with increasing number of sites. This leads us to predict that expansion will be approximately linear up to about 20 sites.

# CHAPTER 11

# EPILOGUE

## SECTION 1: ACHIEVEMENTS.

In this thesis we have presented a philosophy for the software design of a distributed system. By choosing to concentrate on the domain concept, rather than being process orientated, we have arrived at a system with the following properties:

1) The opportunities for load balancing occur frequently, every time there is an interdomain jump. These load balancing points are optimal in the sense that the minimum possible context is involved. This is because computations are changing their (protection) environment at times of load balancing.
2) There is no duplication of code except when efficiency considerations dictate that there should be. Since domains are identified with functions, all users of a function will use the same copy of the code for the function when memory space is short. This is in contrast to process orientated systems where either functions have to be statically allocated to sites or all sites have to have the code for all functions.
3) The domain mechanism neatly handles the control of operating system tables, allowing a single system wide

operating system. This frees the maximum amount of primary memory for use by non-operating system programs.

4) We have incorporated naturally into our system the present state of the art with respect to protection. Domain structured systems have been shown to be more versatile than message passing systems in the kind of protection they can offer.

5) The philosophy we have adopted leads, in Spier's experience [SPIE74], to better structuring of software and greater reliability.

In this thesis we have detailed the special mechanisms required to handle domains in a distributed system. These mechanisms have been incorporated into a simulation program which demonstrated that it is possible to achieve stable operation of a load balancing distributed system. Further, the statistics derived from the simulation show that the design goal we set in chapter 1, a system with low initial cost and nearly constant cost/performance ratio with increasing size, can be achieved.

## SECTION 2: FURTHER RESEARCH.

We feel that this thesis raises more questions than it answers. The following is an incomplete list of research topics that we think could be profitably pursued.

### Communication between virtual processors:

In chapter 7 we expressed some concern about the efficiency of transferring local segments between two virtual processors. This arose in the context of passing buffers to and from peripheral controllers, but the problem is the same for any form of communication between virtual processors. The processor base segment has to be carried along with the buffer, increasing the load on the communication subsystem. We would like to investigate whether a mechanism similar to ports [BALZ71,AKKO75], used in process orientated systems, can be incorporated into our kernel/domain model. A virtual processor would pass a message, addressed to a port, to its local kernel and the kernels will ensure that the message is delivered to the appropriate place.

The use of ports would also affect the concept of secretary processors. The secretary processor would be the sole virtual processor to execute in domains handling peripherals.

Of course it may turn out that the management of ports involves more overheads than carrying the processor base segment along with all messages. Research is required to ascertain whether this is so.

Implementation:

In chapter 9 we pointed out that we had no real idea of the characteristics both of user behaviour and program behaviour upon which to base simulation parameters. This, of course, is the lot of all simulators of unbuilt systems, but it is a particularly severe problem for us because our system is quite different from any actually in existence. Building and operating a distributed system would enable research to be carried out in these areas of behaviour. It would also help identify what hardware or firmware features would assist the domain management function.

Spier did not publish any performance figures for his implementation of a single site kernel/domain architecture. He contented himself with saying 'The operating system is within the realm of the possible, contingent only upon the emergence of next-generation domain-orientated hardware machines' [SPIE74]. Undoubtedly domain management could be very expensive in systems with inappropriate hardware. The severity of the problem can only be gauged by implementing the system.

Only then can a realistic determination be made as to whether our system belongs to the class of toy operating systems or is a viable technique for building large systems out of small scale computers. As we have stated before, one very important determinant of viability is the size of domains.

Many of the algorithms used in our simulation program could be simply transliterated to a real implementation. (Indeed one of our main reasons for simulating a directly connected system with directory updating, rather than a bus type system with an associative mechanism, is that the former architecture could be immediately implemented whereas the later would require development of the communication subsystem). The figures derived from the real implementation could be fed back to the simulation program to validate it and enable it to be used to predict the performance of bigger configurations.

Programming languages:

Building an actual system would also assist in evaluating the requirements of a programming language for domain based systems. An easy to use programming language would be of widespread benefit. We have been told the implementors of the CAP machine have found it difficult to link segments into domains. Also once a language had been developed it would be of assistance in

gaining an idea of the natural size for domains (when domains are 'glued together' from some present language code they will probably be made large enough to be efficient irrespective of underlying structure).

Communication subsystems:

The area of computer communications is one where there are still plenty of research problems [OPDE75]. Of particular interest to us is the design of local communication subsystems. We would like to know if transmission schemes such as that employed in Ethernet are stable [KLEI76] and if they can be married with intelligent interface units. Functions of the interface units could include error control, the global object management we have outlined (including ensuring that no messages get lost when a global object changes sites) and intelligence gathering. The intelligence gathering, or eavesdropping, function needs researching to determine how effective it is. Some evaluation of its effectiveness could be performed using our simulation program.

Alternative architectures:

Our distributed kernel/domain scheme need not be confined to distributed systems. We mentioned in chapter 2 that a scheme devised for a system without shared memory may well be appropriate for a system with shared memory. A knowledge of which segments a computation will access could be used to place the segments so as to reduce or even eliminate memory contention.

One kind of architecture that could be investigated is that of PRIME [BASK72,FABR73], but without a supervisory processor. In PRIME memory modules (and backing store units) are switchable between processors. Once switched to a processor, a memory module is accessed privately by that processor. The switching of modules can be considered as a high speed method of transferring segments between processors. The frequency of switching is intended to be very infrequent compared to the frequency of memory accesses.

Perhaps the most promising alternative architecture to consider is a system where each processor has its own primary memory but where relatively high performance secondary memory is shared between all processors as a replacement for, or supplement to, the communication subsystem. Fuchel and Heller [FUCH68] have proposed a system of two CDC 6600 computers sharing extended core store (ECS). The ECS was to contain a common job queue

and core images of all swopped out jobs. At the other end of the power scale, Wensly [WENS75] proposes a system with small computers sharing an electronic disk. Arden and Berenbaum [ARDE75] have given consideration to the type of access circuitry needed for this shared second level of memory. This kind of architecture can be regarded as a multiprocessor system with a cache for each processor. But with the type of system we have proposed, based on domains, there is a massive simplification of the operation of the cache. If the cache holds all the segments of a domain incarnation then it can be guaranteed a priori that there will not be a consistency problem. There will be no need to check every cache write operation [TANG76] to make sure that the altered word is not also in another cache. It is interesting to note the direction being taken by the Minerva multiprocessor system [WIDD76]. Cache memory is being introduced to save loading on the shared bus to main memory. The implementors plan to use Concurrent Pascal as their programming language so that a write operation to a shared memory location can be detected at compile time and the consistency problem eased.

Parting remark:

One day soon some microcomputer is going to become the 'de facto' industry standard. Abundant software will be produced for this microcomputer, locking all manufacturers into producing compatible architectures. If these architectures do not have have the capability for easy integration into multiple computer systems then a great and irreversible loss will have occured. But the requirements for multiple computer working have yet to be generally delineated. Our research is a small step towards this goal, a lot more work is needed quickly.

# APPENDIX A

## SIMULATION PROGRAM

PROGRAM LISTING A-2

CROSS REFERENCE TABLE A-77

SAMPLE OUTPUT LISTINGS A-85

```
!Simulation of a Domain Based Distributed System

!Written by: L. Casey    Date:Oct 76  Version: PRINT;
                                                                    5

!The aim of this program is to simulate the operation of;
!a network of n computers.;

!Each computer has a kernel whose functioning is modelled in;       10
!the class kernelc.  The basic operation of the;
!kernel is to examine entries in a prioritized queue (driverq);
!and take appropriate action (see around line 1350).;

!The basic structure of the program is as follows;                  15
!lines 41 to 173 declare and define constants and parameters;
!of the simulation. Two values, the number of sites and the;
!number of consoles, are read as input data.;
!lines 180 to 528 declare  some primitives for controlling
!errors,queues and the gathering of statistics;                     20
!lines 530 to 556 declare the basic classes (contentc and segmentc);
!lines 573 to 1918 declare the kernelc class, defining the action;
!of each computer in the network;
!lines 1920 to 2989 declare the classes required for the;
!manipulation of domains;                                           25
!lines 2991 to 3319 declare the other process classes (s_channelc, ;
!consolec,clockc,disk_controllerc and diskc);
!from line 3320 onwards is mainly initialization code for the;
!running of the simulation;
                                                                    30



                                                                    35
!After the program a cross reference listing is given.;
!The letter D after a line number indicates that the variable is;
!declared at that line while M indicates that it occurs more than;
!once.;
                                                                    40
```

```
!!!!!!!!!!constants of the simulation run!!!!!!!!!!!!!!!!;


BEGIN
  INTEGER                                                        45
  n;
  !the number of sites in the network (maximum=128);

  INTEGER msize;            !size of primary memory at each site;
                                                                 50
  INTEGER
  fixed_domains,            !no of operating system domains;
  max_consoles,
  !no of active consoles attached to system (< max_processors);
  ipld,                     !no of domains existing at ipl time;
  max_disks,                !number of disks (not greater than n);
  max_disk_bufs,
  !number of buffers for each disk controller;
  max_writes_pending,       !a control factor for access to disks;
  compl,                    !no of domains that are compilers;   60
  ddl,ddu,mntrl,mntru,diskl;                 !for naming domains;
  !diskhandler domains numbered from diskl to diskl+max_disks-1;

  REAL
  contextdelay,                                                  65
  !the time to preserve context on accepting an interrupt;
  timeslice,                !intervals for user processes;
  longtimeslice,
  mesdelay;                 !the physical delay/byte in sending;
  !a message from one site to any other;                         70
  BOOLEAN
  running,                  !generally true;
  full_diags,q_trace,mem_trace;

  INTEGER max_local_segs,                                        75
  !the number of local segments in an incarnation;
  max_param_segs,           !number of parameter segments permitted;
  stack_depth;              !for number of incarnations;

  INTEGER low,medium,monitor,high;           !priorities;        80
  TEXT ARRAY priority_text(1:4);
  INTEGER null,incore,ondisk,trans,desc;     !status;
  INTEGER supern,           !domain number for user supervisor;
  commandn,
  !domain number for interpreting commands;                       85
  cnsl_site;                !site where all consoles are attached;
  INTEGER size_divider,     !constant used in memory management;
  t_length;                 !length of hash table at each site;
  INTEGER load_shed;
  !factor deciding when to migrate processors;                    90
  INTEGER max_shifts;
  !another factor for migrating when space is tight;
  INTEGER chopfactor,chopsize;
  !global constraints on number of active processors;
  INTEGER i_chopf;                                               95
  !desirable limit on processors at individual sites;
  REAL ARRAY userp,usert(1:7);
```

```
!constants defining user program behaviour;

INTEGER wait_for_d,seek_d_site,seek_choice,spaceclaimed,valid;  100
!constants used in domain_incarnation class;
INTEGER random_seed;
REAL sim_time,settle_time;
!duration of simulation;
REF (Printfile) results;   !file for results of simulation;     105

Outtext("NUMBER OF SITES*"); Breakoutimage;
n:=Inint;
mesdelay:=8.0;              !microsecs;
contextdelay:=200;          !microsecs;                          110
timeslice:=100000;
longtimeslice:=500000;      !half a second;
running:=TRUE;
low:=1; medium:=2; monitor:=3; high:=4;
priority_text(low) :- Copy("LOW");                               115
priority_text(medium) :- Copy("MEDIUM");
priority_text(monitor) :- Copy("MONITOR");
priority_text(high) :- Copy("HIGH");
incore:=1;ondisk:=2;trans:=3;desc:=4;
Outtext("NUMBER OF CONSOLES*"); Breakoutimage;                   120
max_consoles:=Inint;
max_local_segs:=2;
max_param_segs:=1;
stack_depth:=5;
msize:=128000;              !bytes;                              125
size_divider:=1024;         !bytes;
t_length:=20+(max_consoles*(6+n))//n;
supern := 2;
commandn := 1;
cnsl_site := 1;                                                  130
max_disks:=1+(n-1)//3;
ipld:=2;
!two  special domains (supern and commandn);
compl:=2;                   !number of compilers;
ddl:=ipld+compl+1;          !first ordinary domain;              135
ddu:=ddl+9;                 !10 typel domains;
mntrl:=ddu+1;
mntru:=mntrl+4;             !5 'ordinary' monitors;
diskl:=mntru+1;
fixed_domains:=mntru+max_disks;                                  140
!each disk has its own handler domain;
max_disk_bufs:=3;
max_writes_pending:=n+1+max_disk_bufs;
wait_for_d:=1;
seek_d_site:=0;                                                  145
seek_choice:=2;             !for domain_incarnationc;
spaceclaimed:=3;
valid:=4;
load_shed:=2;
max_shifts:=n-1;                                                 150
chopfactor:=n*4;
i_chopf:=(3*chopfactor)//2+1;
!allow individual sites 50% more than average load;
chopsize:=32000+(max_consoles//4)*500;
```

```
!a stab at a formula;                                              155
random_seed:=787;

sim_time := 12000/max_consoles;
!simulation over a constant number of interactions;
settle_time := 30+2400/max_consoles;             !seconds;        160
userp(1):=0; userp(2):=0.37; userp(3):=0.5; userp(4):=0.64;
userp(5):=0.86; userp(6):=0.92; userp(7):=1.0;;
usert(1):=0.5; usert(2):=4; usert(3):=8; usert(4):=20;
usert(5):=40; usert(6):=80; usert(7):=180;
!last value should be 480;                                         165


full_diags:=FALSE;
mem_trace:=FALSE;
q_trace:=FALSE;             !a lot of output produced when true;

results :- NEW Printfile("RESULT/A:APPEND");
results.Open(Blanks(132));

INSPECT results DO                                                 175


Simulation BEGIN

  !some utility functions;                                         180

  PROCEDURE ptime;
  BEGIN                     !print the time;
    Outfix(Time,0,12); Outtext(Blanks(2));
  END;                                                             185

  PROCEDURE error(t);
  VALUE t;TEXT t;
  BEGIN
    INSPECT Sysout DO BEGIN                                        190
      Outtext("ERROR OCCURED"); Outimage;
    END;                    !notify terminal user;
    Outtext(">>>>>> ERROR IN MODEL AT TIME");
    ptime; Outtext(Blanks(10)); Outtext(t);
    Outimage;                                                      195
    IF NOT full_diags THEN audit;
    !done automatically otherwise;
    running:=FALSE;
    REACTIVATE Main;        !continue execution of main block;
  END;                                                             200

  TEXT PROCEDURE fillin(string,i);
  VALUE string;
  TEXT string;
  INTEGER i;                                                       205
  BEGIN
  !returns a text 3 longer than string with i edited into space;
    TEXT t;
    t:-Blanks(string.Length+3);
    t:=string;              !in left most part;                    210
    t.Sub(string.Length+1,3).Putint(i);
```

```
  !add integer at end;
  fillin:-t;
END;
                                                                      215

CLASS qheadc;
BEGIN                     !queueing system with 4 priority levels;
  REF(Head) ARRAY q(low:high);
  INTEGER i;              !work count;                                220
  REF(Link) PROCEDURE first;
  BEGIN
    i:=high;
    WHILE (IF i<low THEN FALSE ELSE q(i).Empty) DO i:=i-1;
    IF i<low THEN first:-NONE ELSE first:-q(i).Suc;                   225
  END;

  INTEGER PROCEDURE total_entries;
  BEGIN
    INTEGER t;                                                        230
    t:=0;
    FOR i:=low STEP 1 UNTIL high DO t:=t+q(i).Cardinal;
    total_entries:=t;
  END;
                                                                      235
  INTEGER PROCEDURE b_entries;
  b_entries:=q(low).Cardinal+q(medium).Cardinal;

  BOOLEAN PROCEDURE qempty;
  BEGIN                   !true when nothing in queueing system;
    i:=low;
    WHILE (IF i>high THEN FALSE ELSE q(i).Empty) DO i:=i+1;
    IF i>high THEN qempty:=TRUE;
  END;
                                                                      245
  FOR i:=low STEP 1 UNTIL high DO q(i):- NEW Head;
END of class qheadc;

PROCEDURE queue(qhead,entry,priority);
REF(qheadc) qhead; REF(Link) entry; INTEGER priority;                 250
IF priority GE low AND priority LE high THEN
entry.Into(qhead.q(priority))
!insert behind all entries of same priority;
ELSE error("WRONG PRIORITY USED IN QUEUE COMMAND");
                                                                      255

PROCEDURE q_analysis(qhead,heading,items);
VALUE heading;
REF(qheadc) qhead;
TEXT heading;                                                         260
TEXT PROCEDURE items;
!to map ref type variables into descriptive text;
BEGIN
  REF(Link) ptr; INTEGER i;
  Outimage;                                                           265
  ptime;
  Outtext(heading); Outtext(" TOTAL NUMBER OF ENTRIES = ");
  Outint(qhead.total_entries,3);
```

```
  Outimage;
  IF NOT qhead.qempty THEN                                         270
  FOR i := high STEP -1 UNTIL low DO
  BEGIN
    Outtext(priority_text(i)); Outimage;
    ptr :- qhead.q(i).Suc;
    !pick off each member of queue;                                275
    WHILE ptr =/= NONE DO
    BEGIN
      Outtext(items(ptr));
      !returns a text value 12 characters long;
      ptr:-ptr.Suc;                                                280
    END;
    Outimage;
  END;
END of procedure q_analysis;
                                                                   285
```

```
!***********statistics section**********************;

Link CLASS statistic(heading);
VALUE heading; TEXT heading;                                        290
VIRTUAL: PROCEDURE clear, print;
THIS statistic.Into(statistic_list);

REF(Head) statistic_list;
REF(Head) grand_t_list;                                             295

statistic CLASS groupheading;
!this helps format output;
BEGIN
  PROCEDURE clear; ;                                                300
  PROCEDURE print;
  BEGIN
    Outimage;
    Outtext(heading);
    Outimage;                                                       305
  END;
END of class groupheading;


statistic CLASS counter;                                            310
BEGIN
  INTEGER count;         !counts occurences;
  PROCEDURE clear;
  count:=0;
                                                                    315
  PROCEDURE print;
  BEGIN
    Outtext("   NUMBER OF ");
    Outtext(heading);
    Outint(count,IF count<1000 THEN 4 ELSE                          320
    IF count<10000000 THEN 8 ELSE 12);
  END;

  PROCEDURE incr;
  count:=count+1;                                                   325

  PROCEDURE add(number);
  INTEGER number;
  count:=count+number;
                                                                    330
END of class counter;

statistic CLASS timer(master);
REF(grand_total) master;
!this class is for accumulating times of operations;                335
!the timer is 'turned on' by procedure start and;
!'turned off' by procedure stop;
!the final value is added into master;
BEGIN
  REAL start_time, total;                                           340
  BOOLEAN keeping;       !true when in action;
```

```
  PROCEDURE start;
  IF NOT keeping THEN BEGIN
  !multiple calls  o.k.;                                           345
    keeping:=TRUE;
    start_time:=Time;
  END;

  REAL PROCEDURE stop;                                             350
  !in simula can call without using returned value;
  IF keeping THEN BEGIN  !multiple calls o.k.;
    total:=total+(Time-start_time);
    keeping:=FALSE;
    stop:=Time-start_time;                                         355
  END;

  PROCEDURE clear;
  BEGIN
    total:=0;                                                      360
    IF keeping THEN start_time:=Time;
    !reset and start timing from now;
  END;

                                                                   365
  PROCEDURE print;
  BEGIN
    REAL t;
    Outtext("    TOTAL ");Outtext(heading);
    t:=(total+(IF keeping THEN Time-start_time ELSE 0))*&-6;       370
    Outfix(t,1,7);
    !printing in seconds;
    IF master=/=NONE THEN master.add(t);     !update grand total;
  END;
                                                                   375
  keeping:=FALSE; total:=0;
  !default values anyway;
END of class timer;

statistic CLASS time_average;                                      380
!for non-negative numbers;
BEGIN
  REAL initial_time,start_time,total;
  INTEGER val,max;
                                                                   385
  PROCEDURE change_value(level);
  INTEGER level;
  BEGIN
    total:=total+val*(Time-start_time);
    val:=level;                                                    390
    IF val>max THEN max := val;
    start_time:=Time;
  END;

  PROCEDURE clear;                                                 395
  BEGIN
    total:=0;
    max:=val;
    initial_time:=start_time:=Time;
```

```
  END;                                                              400

  PROCEDURE print;
  BEGIN
    Outtext("   AVERAGE "); Outtext(heading);
    Outfix(IF Time-initial_time>0 THEN                              405
    (total+val*(Time-start_time))/(Time-initial_time)
    ELSE 0,0,9);
    Outtext("   MAXIMUM");  Outint(max,8);
  END;
                                                                    410
END of class time_average;


statistic CLASS grand_total;
BEGIN                                                               415
  REAL total;
  PROCEDURE clear;
  total:=0;

  PROCEDURE print;                                                  420
  BEGIN
    Outtext("GRAND TOTAL OF ");
    Outtext(heading);
    Outfix(total,0,7);
    Outimage;                                                       425
  END;

  PROCEDURE add(t);
  REAL t;
  total:=total+t;                                                   430

  THIS grand_total.Into(grand_t_list);
  !overrides statistic_list;
END of class grand_total;
                                                                    435

statistic CLASS regression(heading2);
VALUE heading2;TEXT heading2;
BEGIN
  INTEGER n;                                                        440
  REAL sx,sy,sx2,sy2,sxy;

  PROCEDURE data(x,y);
  REAL x,y;
  BEGIN                                                             445
    n:=n+1;  sx:=sx+x;  sy:=sy+y;
    sx2:=sx2+x*x;  sy2:=sy2+y*y;
    sxy:=sxy+x*y;
  END;
                                                                    450
  PROCEDURE clear;
  BEGIN
    n:=0;
    sx:=sy:=sx2:=sy2:=sxy:=0;
  END;                                                              455
```

```
  PROCEDURE print;
  BEGIN
    REAL a0,a1,d,sd,r2;
    Outimage;                                                          460
    Outtext("REGRESSION ANALYSIS OF  ");
    Outtext(heading); Outtext(" VERSUS ");
    Outtext(heading2);  Outimage;
    IF n>5 THEN BEGIN
      !convert data to seconds;                                        465
      sx:=sx*&-6; sy:=sy*&-6;
      sx2:=sx2*&-12; sy2:=sy2*&-12; sxy:=sxy*&-12;
      d:=n*sx2-sx*sx;
      a1:=(n*sxy-sx*sy)/d;
      a0:=(sy*sx2-sx*sxy)/d;                                           470
      sd:=Sqrt((sy2-a0*sy-a1*sxy)/(n-2));
      !standard deviation of y;
      r2:=(n*sxy-sx*sy)**2/(d*(n*sy2-sy*sy));

      Outtext("NUMBER OF DATA POINTS"); Outint(n,4); Outimage;
      Outtext("MEAN OF ");Outtext(heading); Outfix(sx/n,1,7);
      Outtext("  MEAN OF "); Outtext(heading2); Outfix(sy/n,1,7);
      Outtext("   RESIDUAL STANDARD DEVIATION");
      Outfix(sd,2,6); Outimage;
      Outtext("ESTIMATE OF REGRESSION COEFFICIENT");                   480
      Outfix(a1,2,7);
      Outtext("  INTERCEPT"); Outfix(a0,2,7);
      Outtext("  STANDARD DEVIATION OF REGRESSION COEFFICIENT");
      !has a students t distribution with n-2 degrees of freedom;
      Outfix(n*sd/Sqrt((n-2)*d),2,6);                                  485
      Outtext("CORRELATION COEFFICIENT");
      Outfix(Sqrt(r2),3,5);
    END ELSE Outtext("INSUFFICIENT DATA");
    Outimage;
  END of procedure print;                                              490

END of class regression;


PROCEDURE outputstatistics;                                            495
BEGIN
  REF(statistic) ptr;
  ptr:-statistic_list.Suc QUA statistic;
  WHILE ptr =/= NONE DO
  BEGIN                                                                500
    ptr.print;            !call virtual procedure;
    ptr :- ptr.Suc;
  END;
  Outimage;
  Eject(Line+3);                                                       505
  ptr:-grand_t_list.Suc QUA statistic;
  WHILE ptr=/=NONE DO BEGIN
    ptr.print;
    ptr:-ptr.Suc;
  END;                                                                 510
END;
PROCEDURE clearstatistics;
BEGIN
```

```
  REF(statistic) ptr;
  ptr:-statistic_list.Suc QUA statistic;                        515
  WHILE ptr =/= NONE DO BEGIN
    ptr.clear;
    ptr:-ptr.Suc;
  END;
  ptr:-grand_t_list.Suc QUA statistic;                          520
  WHILE ptr =/= NONE DO BEGIN
    ptr.clear;
    ptr:-ptr.Suc;
  END;
END;                                                            525

!************end of statistics section****************;
```

```
Link CLASS contentc;                                                    530
VIRTUAL: TEXT PROCEDURE dump;
BEGIN
  INTEGER size;          !in bytes;
  INTEGER orgn,dest;
  !for use only when being transfer betweeen sites by;                 535
  !communication sub_system;
  INTEGER mem,qf;        !used for kernel to kernel updating;
  TEXT PROCEDURE dump;   !for diagnostics;
  dump:-Copy("* * * * *   ");
  !always 12 characters;                                                540

END of class contentc;



                                                                        545
contentc CLASS segmentc(site);
INTEGER site;            !where segment resides;
BEGIN
  INTEGER key;           !for segment hash table at site;
  INTEGER default,status;                                               550
  !take values of null,incore,ondisk,trans;
  INNER;
  IF status=incore  AND key>fixed_domains THEN INSPECT k(site) DO
  add_seg(THIS segmentc);
  !after key has been set;                                              555
END;


REF(kernelc) ARRAY k(1:n);
REF(grand_total) usage,  !for total service time of system;             560
total_response;          !for total of all response times;
REF(counter) xfered_domains,xfered_processors,
!counting how many domains and processors shift site;
xfered_locals,
new_incarnations,                                                       565
migrations,
short_commands,over2,over5,
!for analysing response times;
chopcount,spacecount;    !for counting blocked processors;
REF(regression) non_trivial;                                            570
!for response times to substantial commands;
```

```
Process CLASS kernelc(id);
INTEGER id;
BEGIN                                                                  575
  REF(qheadc) driverq,    !all ready to run tasks held in it;
  spaceq;
  !for incarnations waiting only for primary memory space;
  REF(domain_incarnationc) cu,
  !pointer to current domain incarnation;                              580
  d_secretary,            !pointer to process handling disk;
  c_secretary;            !ditto for consoles;
  BOOLEAN
  maskf,
  !true when process switching is not permitted;                       585
  iflag;
  !raised when an interrupt has caused the kernel to switch task;

  REF(clockc) ts_clock;
                                                                       590
  INTEGER mfree,
  !size of current free primary memory at this site;
  copyspace;
  !amount of memory used holding code copies;
  BOOLEAN spaceqempty,                                                 595
  !true when no one at this site is waiting for space;
  !(=spaceq.qempty);
  deadlock_warning;
  !true when printed a warning about possible deadlock;
                                                                       600
  REF(s_channelc) s_channel;
  !for communicating with other sites;
  REF(timer) idle_timer;
  !for collecting statistics;
  REF(time_average) memory_use;                                        605
  REF(contentc) ARRAY space_situation(1:n);

  PROCEDURE initialization;
  BEGIN
    driverq :- NEW qheadc;                         !set up task queue;
    spaceq :- NEW qheadc;
    !set up blocked on memory space queue;
    spaceqempty:=TRUE;
    iflag := FALSE;
    maskf := TRUE;                                                     615
    s_channel:-NEW s_channelc(id);
    cq:-NEW Head;
    sq:-NEW Head;         !queues for messages being sent;

    ts_clock :- NEW clockc(id);                                        620
    idle_timer :- NEW timer("IDLE TIME",NONE);
    memory_use:-NEW time_average("MEMORY USE");
    mfree:=msize-4000;
    !4000 bytes is assumed size of the kernel code;
    m_max:=m_min:=id;                                                  625
    memory_use.change_value(msize-mfree);
    FOR w:=1 STEP 1 UNTIL n DO m_use(w):=mfree;
    !first estimate;
    FOR w:=1 STEP 1 UNTIL fixed_domains DO
```

```
  dmn_info(w):-NEW dmn_infoc;                                         630
  !for handling information about domains;
  FOR w:=1 STEP 1 UNTIL n DO BEGIN
    space_situation(w):-NEW contentc;
    !for warning other kernels that near deadlock;
    !or that have backed of again;                                    635
    space_situation(w).size:=32;
  END;
END;
```

```
!!!!!!!!!!!!!!!!!!!!!! memory management !!!!!!!!!!!!!!!!!!;
! routines to be used by the kernel only                  !;
! contains some assumed timings                           !;
! memory management policy is not actually simulated,     !;
! just arithmetic count kept of free memory.              !;
!---------------------------------------------------------!;

PROCEDURE claim(size,success);
NAME success;
BOOLEAN success;
INTEGER size;           !of block of memory required;          650
BEGIN
  Hold(200+(2000*size/(size+mfree)));
  !reflects the assumption that as;
  !space gets tighter more time is required to find free space;
  IF spaceqempty THEN BEGIN                                    655
    WHILE (IF size GE mfree THEN make_space ELSE FALSE) DO ;
  END;
  !delete enough code copies to give space if possible;
  IF size < mfree THEN
  BEGIN                                                        660
    mfree:=mfree-size;
    !mfree always greater than 0;
    register_m_use(id,mfree+copyspace);
    !alter status information of site;
    memory_use.change_value(msize-mfree);                      665
    !statistic;
    success:=TRUE;
    IF mem_trace THEN BEGIN
      Outint(msize-mfree,10);
      Outchar('>'); Outint(id,1);                              670
    END;
  END
  ELSE success:=FALSE;
END of procedure claim;
                                                               675

PROCEDURE q4space(inc);
REF(domain_incarnationc) inc;
BEGIN
  !assume that claim has been called immediately before;      680
  !i.e. do not check if really has to queue;
  spaceqempty:=FALSE;
  overload:=TRUE;
  queue(spaceq,inc, IF inc.extra_space>size_divider THEN
  low ELSE medium);                                            685
  !small requests have priority over large ones;
  IF mfree>size_divider THEN BEGIN
    inc.extra_space:=inc.extra_space-(mfree-size_divider);
    claim(mfree-size_divider,running);
  END;                  !stake claim to available space;       690
  Hold(2000);           !to sort things out;
  IF full_diags THEN BEGIN
    ptime; Outtext(inc.dump);Outtext(" IN SPACEQ AT SITE");
    Outint(id,3); Outimage;
  END;                                                         695
  spacecount.incr;      !statistic;
```

```
END;

PROCEDURE release(size);                                           700
INTEGER size;
BEGIN                   !giving back a block of memory;
  REF(domain_incarnationc) sr;
  BOOLEAN more;
  Hold(50);                                                        705
  mfree:=mfree+size;
  register_m_use(id,mfree+copyspace);
  memory_use.change_value(msize-mfree);
  IF mem_trace THEN BEGIN
    Outint(msize-mfree,10);                                        710
    Outchar('<'); Outint(id,1);
    !diagnostic message;
  END;
  IF NOT spaceqempty THEN BEGIN
    sr:-spaceq.first;                                              715
    more:=TRUE;
    WHILE sr=/=NONE AND more DO
    BEGIN
    !see if can run incarnation waiting for space;
      claim(sr.extra_space,more);                                  720
      IF more THEN BEGIN
        sr.stage:=spaceclaimed;
        queue(driverq,sr,high);
        !place freed entry capability back in driverq;
        sr:-spaceq.first;                                          725
        IF deadlock_warning THEN BEGIN
          ptime;
          Outtext(fillin("  DEADLOCK AVERTED AT SITE",id));
          Outimage;    !because removed something from spaceq;
          deadlock_warning:=FALSE;                                 730
        END;
      END ELSE
      IF mfree > size_divider THEN
      BEGIN
        sr.extra_space := sr.extra_space -(mfree-size_divider);
        claim(mfree-size_divider,running);
        !try to keep size-divider of memory;
        !to be available for small requests;
      END;
    END;                                                           740
    IF sr==NONE THEN spaceqempty:=TRUE;
  END;
END of procedure release;

                                                                   745


BOOLEAN PROCEDURE make_space;
IF copyspace>0 THEN BEGIN
  !each site can keep copies of shared re-entrant code;           750
  !the least recently used is deleted;
  !(only done when spaceq empty);
  INTEGER i,j;
```

```
    REAL lru;
    Hold(400);                                                          755
    lru:=Time;
    FOR i:=1 STEP 1 UNTIL fixed_domains DO
    IF dmn_info(i).copy THEN BEGIN
      IF dmn_info(i).work.Empty AND
      dmn_info(i).external_segs.Empty                                   760
      AND NOT dmn_info(i).going
      THEN BEGIN
        IF lru > dmn_info(i).lasttime THEN BEGIN
          lru:=dmn_info(i).lasttime;
          j:=i;                                                         765
        END;
      END;
    END;
    IF j>0 THEN BEGIN     !found a segment to delete;
      delete_domain_copy(j);                                            770
      make_space:=TRUE;
    END;
  END of make_space;

                                                                        775
!***************end of memory management****************!;
```

```
!!!!!!!!!!!!!!!!!!segment management!!!!!!!!!!!!!!!!!!;
                                                                  780

REF (segmentc) ARRAY seg_table(0:t_length-1);
INTEGER ARRAY st(0:t_length-1);

INTEGER PROCEDURE hash(key);                                      785
INTEGER key;
hash:=2*Mod(key,t_length//2)+(IF key>16384 THEN 1 ELSE 0);
!small numbers will predominate;

INTEGER PROCEDURE add(key);                                       790
INTEGER key;
BEGIN
  INTEGER hk,i;
  hk:=hash(key);
  IF st(hk) NE 0 THEN                                             795
  BEGIN
    !if first slot not free then search table;
    i:=hk;
    FOR hk:=Mod(hk+1,t_length) WHILE i NE hk AND st(hk) NE 0
    DO;                                                           800
    IF i=hk THEN error (fillin("HASH TABLE FULL AT SITE",id));
  END;
  st(hk):=key;
  add:=hk;
END;                                                              805

INTEGER PROCEDURE retrieve(key);
INTEGER key;
BEGIN
  INTEGER hk,i;                                                   810
  hk:=hash(key);
  IF st(hk) NE key THEN
  BEGIN
    i:=hk;
    FOR hk:=Mod(hk+1,t_length) WHILE i NE hk AND st(hk) NE key
    DO;
    IF i=hk THEN
    error ("ITEM NOT FOUND IN HASH TABLE");
  END;
  IF seg_table(hk)==NONE THEN error("BAD SEGMENT MANAGEMENT");
  retrieve:=hk;
END;

PROCEDURE add_seg(s);
REF (segmentc) s;                                                 825
BEGIN
  seg_table(add(s.key)):-s;
  s.status:=incore;
  s.site:=id;
END;                                                              830

PROCEDURE delete_seg(s);
REF (segmentc) s;
BEGIN
```

```
!this may be called after segment has arrived at another site;
  INTEGER i;
  i:=retrieve(s.key);
  st(i):=0;
  IF s.site=id THEN s.status:=null;
  !not gone anywhere else yet;                                  840
  seg_table(i):-NONE;
  release(s.size);     !give back space;
END;

                                                                845
!*****************end of segment management**************!;
```

```
!!!!!!!!!!!!!! communications section !!!!!!!!!!!!!!!!!!;

                                                                850
!Communication interface - receiving messages;
!Reception of messages takes place in three stages:-;
!1) The message arrives - call on procedure int.;
!2) The message is stored and the kernel notified by placing;
!an entry in driverq - call on queue.;                          855
!3) When driverq entry is examined action is taken on entry.;

PROCEDURE int(m);
REF(contentc) m;
BEGIN                                                           860
  IF NOT m IS contentc THEN BEGIN
  !not an empty message;
    IF m IN segmentc THEN BEGIN
      IF m QUA segmentc.key LE fixed_domains THEN
      dmn_info(m QUA segmentc.key).d:-m QUA domainc             865
      !domain kept separate from other segments;
      ELSE
      add_seg(m QUA segmentc);
      !it is assumed that space has already been claimed;
      queue(driverq,m,high);                                    870
    END ELSE
    IF m IS domain_incarnationc THEN BEGIN
      IF m QUA domain_incarnationc.stage=valid
      THEN queue(driverq,m,monitor) ELSE
      !unless it is a secretary being used as an interrupt;
      !a domain incarnation can not be valid at this stage;
      BEGIN
        queue(driverq,m,high);
        qfs(id):=qfs(id)+1;
        !more work at this site;                                880
      END;
    END ELSE queue(driverq,m,high);
    !other message types;
    switch_context;    !notify kernel;
  END;                                                          885
  IF m.orgn LE n AND m.orgn > 0 THEN BEGIN
    register_m_use(m.orgn,m.mem);
    !update memory utilization table;
    qfs(m.orgn):=m.qf;                     !and size of queues;
  END;                                                          890
END;

!communications interface - sending messages;

REF(Head) cq,          !for high priority control messages;
sq;                    !for segments;
BOOLEAN channel_busy;

PROCEDURE send_message(dest,contents);
INTEGER dest; REF(contentc) contents;                           900
IF dest  NE id THEN
BEGIN
  contents.orgn:=id;
  contents.dest:=dest;
```

```
  IF channel_busy THEN                                              905
  contents.Into(IF contents IN segmentc THEN sq ELSE cq)
  ELSE signal_channel(contents);
  !wait if busy else send message straight away;
END ELSE
BEGIN                   !short circuit;                             910
  contents.orgn:=id;
  queue(driverq,contents,high);
  !do not send secretaries/interrupts to oneself;
END of send_message;
                                                                    915
PROCEDURE broadcast(contents);
!to all other kernels;
!must have a different object to go to each site;
REF(contentc)ARRAY contents;
BEGIN                                                               920
  INTEGER i;
  FOR i:= 1 STEP 1 UNTIL n DO
  IF i NE id THEN send_message(i,contents(i));
END of broadcast;
                                                                    925
PROCEDURE signal_channel(contents);
REF(contentc) contents;
BEGIN
  channel_busy:=TRUE;
  !channel deals with one message at a time;                        930
  contents.mem:=m_use(id);
  IF contents IS domain_incarnationc THEN qfs(id):=qfs(id)-1;
  !if monitors with condition queues moved sites;
  !then this would have to be altered;
  contents.qf:=qfs(id);                                             935
  !information for receiving kernel;
  contents.Out;          !if in cq or sq;
  s_channel.initiate(contents);
END of signal_channel;
                                                                    940
!end of communications interface;
```

```
!******************* load monitoring ***************!;

BOOLEAN overload;                                                    945
!If kernel detects that no site has chopsize of free memory;
!or that some site (probably) has entries in its spaceq;
!i.e. when its free memory is less than size_divider;
!then overload is set true.;
!Overload is used by secretaries to modify their behaviour.;

INTEGER ARRAY m_use,qfs(1:n);
!tables of (supposed) utilization of memory and number of;
!domain incarnations at each site;
                                                                     955
INTEGER m_max,m_min;   !sites with most and least free memory;

PROCEDURE register_m_use(site,mem);
INTEGER site,mem;      !the amount of free memory at the site;
BEGIN BOOLEAN sort_required; INTEGER j;                              960
  IF n > 1 THEN
  BEGIN
    IF site NE m_max AND site NE m_min THEN
    BEGIN
      IF mem>m_use(m_max) THEN m_max:=site                           965
      ELSE IF mem<m_use(m_min) THEN m_min:=site;
    END
    ELSE
    sort_required:=(site=m_max AND mem<m_use(m_max)) OR
    (site=m_min AND mem>m_use(m_min));                               970
  END;
  m_use(site):=mem;
  IF sort_required THEN
  BEGIN
    m_max:=m_min:=1;                                                 975
    FOR j:=2 STEP 2 UNTIL n DO
    !never sort required when n=1;
    IF m_use(j)>m_use(m_max) THEN m_max:=j ELSE
    IF m_use(j)<m_use(m_min) THEN m_min:=j;
  END;                                                               980

  IF m_use(m_max)<chopsize OR m_use(m_min) LE size_divider THEN
  BEGIN
    IF site=id AND NOT overload THEN
    broadcast(space_situation);                                      985
    !this site has been cause of pushing network into overload;
    !so it notifies the other sites;
    overload:=TRUE;
  END
  ELSE                                                               990
  BEGIN
    IF site=id AND overload THEN broadcast(space_situation);
    !first site out of overload - tell others;
    overload:=FALSE;
  END;                                                               995
END of register_m_use;
```

```
INTEGER PROCEDURE optimum_site(size,qf_min,qf_max);            1000
INTEGER size,qf_min,qf_max;
!Returns the identity of the site with minimum current work;
!load (between qf_min and qf_max-1) and free space greater;
!than size. Where there is more than one site at the level the;
!one with the most space is chosen.;                            1005
!If there are no sites satisfying the conditions returns zero.;

IF m_use(m_max)>size THEN
BEGIN                      !worth looking;
  INTEGER i,j,k;                                                1010
  k:=qf_min-1;
  FOR k:=k+1 WHILE k<qf_max AND j=0 DO
  FOR i:=1 STEP 1 UNTIL n DO
  IF qfs(i)=k THEN
  BEGIN                                                         1015
    IF m_use(i) GE size THEN
    BEGIN
      j:=i;
      size:=m_use(i);
    END;                                                        1020
  END;
  optimum_site:=j;
END of optimum_site;

!****************** end of load monitoring **************;
```

```
!!!!!!!!!!!!!!!!!!!!domain management!!!!!!!!!!!!!!!!!!!;

                                                            1030
REF(dmn_infoc)ARRAY dmn_info(1:fixed_domains);


BOOLEAN PROCEDURE validated(dmn_rqst);
REF(domain_incarnationc) dmn_rqst;                          1035
!this procedure handles most of the stages of transferring;
!a processor to a new domain ;
!it checks if the entry capability - dmn_rqst - is valid;
!if not it takes steps to make it valid;
                                                            1040
IF dmn_rqst.stage=valid THEN validated:=TRUE
!ready to run;
ELSE
BEGIN
  Hold(200);            !calculation overhead;              1045

  IF dmn_rqst.stage=seek_d_site OR dmn_rqst.stage=wait_for_d
  THEN
  INSPECT dmn_info(dmn_rqst.did) DO
  BEGIN                                                     1050
  !must have entry capability at the site of domain before;
  !can work on it;
    IF NOT (here OR coming OR (copy AND NOT overload)) THEN
    send_message(d_loc,dmn_rqst)
    ELSE do_domain_calculation(dmn_rqst);                   1055
  END;

  IF dmn_rqst.stage=seek_choice THEN
  BEGIN
    IF dmn_rqst.choice NE id THEN          !at wrong site; 1060
    send_message(dmn_rqst.choice,dmn_rqst)
    ELSE
    examine_choice(dmn_rqst);
  END;
  !dont set validated so that incarnation goes to end of queue;
  IF dmn_rqst.stage=spaceclaimed THEN bring_together(dmn_rqst);

END of procedure validated;

                                                            1070
REAL PROCEDURE cost_formula(site,size);
INTEGER site,size;
!attempt to give a factor corresponding to congestion;
BEGIN
  REAL d;                                                   1075
  d:=qfs(site);
  IF d=0 THEN d:=0.01;                      !no work at site;
  cost_formula:=size*(m_use(site)/(d*(msize-m_use(site))));
END;
                                                            1080

PROCEDURE do_domain_calculation(dmn_rqst);
REF(domain_incarnationc) dmn_rqst;
```

```
!this procedure calculates the 'best' site for the domain;
!incarnation to take place;                                   1085
INSPECT dmn_rqst DO
INSPECT dmn_info(did) DO BEGIN
  INTEGER big_d_size,big_p_size;
  REAL d_cost,l_cost,p_cost;
  INTEGER i;                                                  1090
  IF coming THEN
  BEGIN
    stage:=wait_for_d;
    !note that has waited;
    dmn_rqst.Into(rqst_list);                                 1095
  END
  ELSE
  BEGIN
    !first sort out where all the segments are;
    l_site:=0;                                                1100
    !until determined that local segments exist;
    total_size:=l_size:=0;

    d_site:=id;
    d_size:=dmn_info(did).d.size;                             1105
    big_d_size:=IF d IN monitorc THEN
    d_size*(1+work.Cardinal+external_segs.Cardinal)
    ELSE d_size;
    !try to form a clumping of virtual processors using;
    !particular monitors;                                     1110

    INSPECT processor DO
    BEGIN
      p_size:=size;
      FOR i:=1 STEP 1 UNTIL max_param_segs DO                 1115
      IF params(i).status=incore THEN
      p_size:=p_size+params(i).size
      ELSE IF params(i).status=trans OR params(i).status=ondisk
      THEN total_size:=total_size+params(i).size;
      p_site:=site;                                           1120
      big_p_size:=p_size*sameness;
      !an agregating factor;
    END;
    total_size:=total_size+p_size;
    !so far size of processor and all parameter segments;  1125

    !assume all local segments at same site;
    FOR i:=1 STEP 1 UNTIL max_local_segs DO
    IF locals(i).status=incore THEN BEGIN
      l_site:=locals(i).site;                                 1130
      l_size:=l_size+locals(i).size;
    END
    ELSE
    IF locals(i).status = trans THEN
    !to be created;                                           1135
    total_size:=total_size+locals(i).size;

    IF l_site NE 0 THEN
    BEGIN               !some local segments involved;
      total_size:=total_size+l_size;                          1140
```

```
END;

!now choose site domain incarnation is to take place;

IF d.tied NE 0 THEN                                                  1145
BEGIN                  !domain cannot move;
  choice:=id;
  stage:=seek_choice;
END
ELSE                                                                 1150
BEGIN
  d_cost:=cost_formula(id,big_d_size+
  (IF p_site=id THEN big_p_size ELSE 0)+
  (IF l_site=id THEN l_size ELSE 0));
  IF p_site NE id THEN                                               1155
  p_cost:=cost_formula(p_site,big_p_size+
  (IF l_site=p_site THEN l_size ELSE 0));
  IF l_site NE 0 AND l_site NE id AND l_site NE p_site THEN
  l_cost:=cost_formula(l_site,l_size);
                                                                     1160
  choice:= IF d_cost GE p_cost AND d_cost GE l_cost THEN id
  ELSE IF p_cost GE l_cost THEN p_site
  ELSE l_site;

  !now check that choice is o.k.;                                    1165

  !first check that if domain is going to move;
  !from this site that it is not already promised;
  !elsewhere or only a copy exists here;
  IF choice NE id THEN                                               1170
  BEGIN
    IF next_site NE 0 THEN
    send_message(next_site,dmn_rqst)
    ELSE
    IF copy THEN                                                     1175
    send_message(d_loc,dmn_rqst)
    ELSE
    !no objections to domain moving;
    stage:=seek_choice;
  END                                                                1180
  ELSE
  BEGIN
    !this site has been chosen;
    !if a monitor that has been promised to;
    !another site is involved, the incarnation will;                 1185
    !only take place if
    !1) the incarnation was waiting before the
    !   monitor arrived at this site;
    !   (stage = wait_for_d);
    !or;                                                             1190
    !2) all required segments are at this site;
    !   and other incarnations have outstanding;
    !   requests for segments from other sites;

    IF next_site NE 0 AND stage NE wait_for_d THEN                   1195
    BEGIN
      IF external_seg.Empty OR p_site NE id OR
```

```
        NOT (l_site=0 OR l_site=id)
        THEN send_message(next_site,dmn_rqst)
        ELSE                                                          1200
        stage:=seek_choice
      END
      ELSE stage:=seek_choice;
      !no objection to its staying;
    END;                                                              1205
  END;
END;

END of procedure do_domain_calculation;
                                                                      1210
PROCEDURE examine_choice(dmn_rqst);
REF(domain_incarnationc) dmn_rqst;

INSPECT dmn_rqst DO
BEGIN                                                                 1215
  BOOLEAN spacefound,
  !true if this site has enough space for incarnation;
  tied,                 !true if  domain tied down here;
  gone;                 !set if incarnation sent elsewhere;
  INTEGER opt_site,i;                                                 1220

  PROCEDURE send_off;
  BEGIN
    choice:=opt_site;
    send_message(opt_site,dmn_rqst);                                  1225
    gone:=TRUE;
    shifts:=shifts+1;
    migrations.incr;
  END;
                                                                      1230
  tied:=IF dmn_info(did).here THEN dmn_info(did).d.tied NE 0
  ELSE FALSE;

  !first see if some other site is under_utilized while this;
  !site is overbusy;                                                  1235
  IF NOT(tied OR shifts>max_shifts OR
  (overload AND dmn_info(did).here AND shifts>0)) THEN
  BEGIN
    opt_site:=
    optimum_site(total_size+d_size,0,qfs(id)//load_shed);  1240
    IF opt_site NE 0 THEN send_off;
    !substantially less busy;
  END;

  IF NOT gone THEN                                                    1245
  BEGIN
  !now claim the extra space required and make up lists;
  !for requesting segments;
    processor.Into(dmn_info(did).work);
    !mark domain as used;                                             1250

    IF NOT (dmn_info(did).coming OR dmn_info(did).here OR
    dmn_info(did).copy) THEN
    extra_space:=d_size ELSE extra_space:=0;
```

```
!it is possible that domain is no longer at d_site;     1255
extra_space:=extra_space+total_size -
(IF p_site=id THEN p_size ELSE 0) -
(IF l_site=id THEN l_size ELSE 0);

IF extra_space>0 THEN                                   1260
claim(extra_space,spacefound) ELSE spacefound:=TRUE;

IF NOT(spacefound OR tied) THEN
BEGIN              !see if another site is suitable;
  IF shifts LE                                          1265
  (IF dmn_info(did).here AND overload THEN 0
  ELSE max_shifts)
  THEN BEGIN
    opt_site:=optimum_site
    (total_size+d_size,qfs(id)//load_shed,i_chopf);     1270
    IF  opt_site NE id AND opt_site NE 0 THEN BEGIN
      send_off;
      processor.Out;
      !reverse marking of domain;
    END;                                                1275
  END;
END;

IF NOT gone THEN INSPECT dmn_info(did) DO
BEGIN              !sort out domain whereabouts;        1280
  count:=0;
  IF NOT (coming OR here OR copy) THEN BEGIN
    count:=1;
    !keep count of segments required from other sites;
    processor.d_transfer.did:=did;                      1285
    processor.d_transfer.rqstor:=id;
    coming:=TRUE;
    !so as subsequent dmn_rqsts dont claim extra space;
  END;
                                                        1290
  IF spacefound THEN stage:=spaceclaimed ELSE
  q4space(dmn_rqst);
  !wait until space is available;

  IF tied AND NOT spacefound THEN BEGIN                 1295
    IF (IF dmn_info(did).here THEN
    dmn_info(did).d.tied NE 0
    ELSE FALSE)
    THEN BEGIN
      !if the domain is genuinely tied here then;       1300
      !see if there is a non-tied domain whose incarnation;
      !can be moved away;
      !in a really tight situation most incarnations;
      !will be shifted around;
      !however this will stop when there are none;      1305
      !left that have not been shifted;
      REF(domain_incarnationc) ptr;
      BOOLEAN found;
      ptr:-driverq.q(low).Pred;
      WHILE ptr=/=NONE  DO                              1310
      !work backwards along low priority queue;
```

```
            BEGIN
              IF dmn_info(ptr.did).d.tied=0 AND
              ptr.shifts<max_shifts THEN
              BEGIN        !domain not tied here;                  1315
                opt_site:=
                optimum_site
                (ptr.total_size+ptr.d_size,0,i_chopf);
                IF opt_site NE 0 AND opt_site NE id THEN
                BEGIN      !found somewhere to go;                 1320
                  found:=TRUE;
                  retire(ptr);
                  queue(driverq,ptr,high);

                  !since there is no space here this incarnation;
                  !will go to another site;
                END;
              END;
              IF found THEN ptr:-NONE ELSE ptr:-ptr.Pred;
            END;                                                   1330
          END;
        END;

      END;
    END;                                                           1335
  END of examine_choice;


  PROCEDURE bring_together(dmn_rqst);
  REF(domain_incarnationc) dmn_rqst;                               1340
  !now request other sites to send segments;
  !and create temporary ones;
  INSPECT dmn_rqst DO
  INSPECT processor DO BEGIN
    INTEGER i;                                                     1345
    IF count=1 THEN BEGIN
      send_message(dmn_info(did).d_loc,d_transfer);
      !domain at another site;
      count:=0;            !and not currently requested;
    END;                                                           1350

    IF site NE id THEN
    BEGIN                  !processor not here;
      count:=count+1;
      p_seg_list.rqstor:=id;                                       1355
      FOR i:=1 STEP 1 UNTIL max_param_segs DO
      IF params(i).status=incore THEN
      BEGIN
        count:=count+1;
        p_seg_list.a(i):=params(i).key;                            1360
      END
      ELSE
      p_seg_list.a(i):=0;

      send_message(site,p_seg_list);                               1365
    END;

    FOR i:=1 STEP 1 UNTIL max_param_segs DO
```

```
BEGIN
  IF params(i).status=trans THEN add_seg(params(i))          1370
  !work segment created;
  ELSE IF params(i).status=ondisk THEN BEGIN
    count:=count+1;
    disk_read.rqstor:=id;
    !can only be one read at a time;                          1375
    disk_read.key:=params(i).key;
    send_message(params(i).site,disk_read);
    !'site' is a diskcontroller;
  END;
END;                                                          1380

FOR i:=1 STEP 1 UNTIL max_local_segs DO
IF locals(i).status=incore THEN
BEGIN
  IF locals(i).site NE id THEN BEGIN                          1385
    count:=count+1;
    l_seg_list.a(i):=locals(i).key;
  END;
END
ELSE BEGIN                                                    1390
  l_seg_list.a(i):=0;
  IF locals(i).status=trans THEN add_seg(locals(i));
END;

IF l_site NE 0 AND l_site NE id THEN                          1395
BEGIN
  l_seg_list.rqstor:=id;
  send_message(l_site,l_seg_list);
END;
                                                              1400
IF count>0 OR dmn_info(did).coming THEN
dmn_rqst.Into(dmn_info(did).external_segs)
ELSE put_in_ready_state(dmn_rqst);
END of procedure bring_together;
                                                              1405



PROCEDURE put_in_ready_state(inc);
REF(domain_incarnationc) inc;                                 1410
BEGIN
  !assumes that all segments are at site;
  inc.site:=id;         !give site of execution;
  inc.stage:=valid;
                                                              1415
  inc.processor.Into(dmn_info(inc.did).work);
  !keep track of run state work;

  inc.domain:-dmn_info(inc.did).d;
  !domain must be here;                                       1420
  IF inc.domain==NONE THEN
  error("ATTEMPT TO RUN WITHOUT DOMAIN");
  IF inc.domain IN monitorc THEN BEGIN
    !not going to be there long so give favourable priority;
    inc.processor.rts:=timeslice;                             1425
```

```
    queue(driverq,inc,monitor);
   END
   ELSE
   queue(driverq,inc,inc.processor.priority);
   !the invalid form of inc had high priority;                    1430
   !an alternative would be to schedule into medium or;
   !low depending on rts, the remaining time slice;
 END of procedure put_in_ready_state;

                                                                  1435
 PROCEDURE retire(inc);
 REF(domain_incarnationc) inc;
 BEGIN
 !called when an incarnation is removed from driverq;
   inc.processor.Out;   !of dmn_info work list;                   1440
   action_transfer(dmn_info(inc.did));
   !does domain want to go to another site?;
   inc.Out;
   !removed from driverq completely;
   inc.stage:=seek_d_site;                                        1445
   !set entry capability back to base state;
   inc.shifts:=0;       !start counting forced migrations again;
 END of procedure retire;

                                                                  1450
 PROCEDURE action_transfer(h);
 REF(dmn_infoc) h;
 BEGIN
   PROCEDURE send_domain;
   BEGIN                                                          1455
     !broadcast(h.update);
     !tell other sites that domain is going to next_site;
     h.here:=FALSE;
     send_message(h.next_site,h.d);
     h.next_site:=0;                                              1460
   END;

   IF h.next_site NE 0 AND h.here THEN
   BEGIN                  !domain wanted elsewhere;
     IF h.d IN monitorc THEN                                      1465
     BEGIN                !must wait until domain is free;
       IF h.work.Empty AND h.external_segs.Empty THEN
       send_domain;
     END
     ELSE                                                         1470
     BEGIN                !can send domain and keep copy;
       h.going:=h.copy:=TRUE;
       copyspace:=copyspace+h.d.size;
       !going = true protects copy from being deleted;
       !unil it is transmitted;                                   1475
       send_domain;
       !decide after actual transmission if want to keep copy;
     END;
   END
                                                                  1480
   ELSE IF h.copy THEN BEGIN
   !domain is not wanted elsewhere;
```

```
  !see if want to delete copy;
    IF h.work.Empty AND h.external_segs.Empty THEN BEGIN
      IF NOT (spaceqempty OR h.going) THEN                   1485
      delete_domain_copy(h.d.did)          !tight for space;
      ELSE h.lasttime:=Time;
    END;

  END;                                                       1490
END of procedure action_transfer;


PROCEDURE domain_copy(domain);
REF(domainc) domain;                                         1495
!this procedure is called by communication section;
!when it has completed the transmission of a domain;
BEGIN
  xfered_domains.incr;                     !statistics;
  IF domain IN monitorc THEN BEGIN                           1500
    release(domain.size);
    !never keep copies of monitors;
    dmn_info(domain.did).d:-NONE;
  END
  ELSE                                                       1505
  INSPECT dmn_info(domain.did) DO BEGIN
    going:=FALSE;
    IF ((NOT spaceqempty) AND work.Empty
    AND external_segs.Empty)
    THEN delete_domain_copy(d.did);                          1510
  END;
END of procedure domain_copy;


PROCEDURE delete_domain_copy(did);                           1515
INTEGER did;
!this interacts with memory management;
INSPECT dmn_info(did) DO BEGIN
  copy:=FALSE;
  copyspace:=copyspace-d.size;                               1520
  release(d.size);     !give back space;
  d:-NONE;             !domain deleted;
END of delete_domain_copy;

                                                             1525
!**********end of domain management****************;
```

```
PROCEDURE switch_context;
!this procedure is only ever called from outside the kernelc;
IF Idle THEN ACTIVATE THIS kernelc DELAY 0                    1530
!assume no switching delay;
ELSE
IF NOT maskf THEN BEGIN
  !when executing a domain incarnation;
  iflag := TRUE;        !have raised genuine interrupt;      1535
  maskf := TRUE;        !but dont want to be interrupted now;
  REACTIVATE THIS kernelc DELAY 0;
END;

                                                              1540


PROCEDURE execute(x);
REF(domain_incarnationc) x;
                                                              1545
!This procedure is the simulation of execution of;
!a domain incarnation.;
!Although an actual domain incarnation would be;
!interruptable at any point of execution of the;
!domain code it is assumed that the domain code;              1550
!is divided up into at most five steps and;
!interrupts occur between these steps. The passage;
!of time is simulated by 'hold' for the assumed;
!execution time of a step;
!followed by the 'instantaneous' execution of the;           1555
!step.;
!The context of a domain incarnation is thus preserved;
!by noting how long it has to remain in the 'hold';
!state, 'runtt', and which step it must execute;
!next, 'next_step'.;                                          1560
!Steps, which are virtual procedures of the class domainc,;
!have the following properties:;
!1) at end must set the next set of values for runtt;
!and next_step.;
!2) can call kernel primitives, which may remove this;      1565
!incarnation from driverq by manipulating cu.;
!3) should exercise care in calling more than one kernel;
!primitive in a step.;
BEGIN
  REAL finisht;         !work variable;                       1570
  IF x.domain==NONE THEN
  error("ATTEMPT TO MAKE SKELETON LIVE");
  IF x.runtt NE 0 THEN
  BEGIN
    maskf:=FALSE;                                             1575
    !normal domain execution is interruptable and pre-emptable;
    finisht:=Time+x.runtt;
    x.processor.service_timer.start;
    !statistics collection;
    Hold(x.runtt);      !simulate time to complete action;  1580
    !when the next instruction is executed either the;
    !processor has 'finished' or it has been interrupted;
    x.processor.service_timer.stop;
    x.processor.c_time:=
```

```
    x.processor.c_time+x.runtt-(finisht-Time);                     1585
    x.runtt:=IF iflag THEN finisht-Time ELSE 0;
    maskf:=TRUE;        !no longer permitted to interrupt;
  END;

  IF NOT iflag THEN                                                1590
  BEGIN
  !perform step whose execution time has just been simulated;
    SWITCH s:=stp1,stp2,stp3,stp4,stp5;
    !a case statement;
    IF x.next_step<1 OR x.next_step>5 THEN                         1595
    error("DOMAIN NOT FORMULATED CORRECTLY");
    GOTO s(x.next_step);
    stp1: x.domain.step1(x); GOTO esac;
    stp2: x.domain.step2(x); GOTO esac;
    stp3: x.domain.step3(x); GOTO esac;                            1600
    stp4:x.domain.step4(x); GOTO esac;
    stp5: x.domain.step5(x);
    esac:
  END;
END of procedure execute;                                          1605



REF(Link) ptr;                                                     1610

initialization;

WHILE running DO
BEGIN                                                              1615

  IF iflag THEN BEGIN
  !simulate time to switch from user process;
    Hold(contextdelay);
    iflag := FALSE;                                                1620
  END;

  ptr:-driverq.first;
  !examine top entry of driverq;
                                                                   1625
  IF q_trace AND ptr=/=NONE THEN
  BEGIN
    Outtext("    ("); Outint(id,2); Outchar(')');
    ptime; Outtext(dissect(ptr));
  END;                                                             1630

  !determine type of entry;
  IF ptr IN domain_incarnationc THEN
  BEGIN
    !cu always points to the current domain_incarnation which;
    !keeps its place in driverq unless a kernel primitive;
    !removes it;
    cu :- ptr;

    IF validated(cu) THEN BEGIN                                    1640
      !continue if valid entry capability;
```

```
      ts_clock.set(cu.processor.rts);
      !for remaining timeslice;

      execute(cu);                                                1645
      !A step of the domain code - consisting  in fact;
      !of a 'hold' followed by 1 of the virtual;
      !procedures stepl...step5 - is performed;
      !or;
      !an interrupt, during the 'hold' section, has occurred;
      !when control returns here.;

      ts_clock.halt(cu.processor.rts);     !stop clock;

      !after higher priority entries in driverq have;             1655
      !been examined, execution of the remaining part;
      !of an interrupted step or the next step;
      !will take place.;

      IF NOT iflag THEN cu:-NONE;                                 1660
      !clearing cu is indicative of the succesful completion;
      !of a step;
    END
    ELSE cu:-NONE;
    !incarnation not executed because invalid;                    1665
  END



  ELSE                                                            1670
  BEGIN
    IF ptr=/=NONE THEN ptr.Out;
    INSPECT ptr

                                                                  1675
    WHEN clockc DO IF ptr==ts_clock THEN
    BEGIN
    !end of time slice for current domain incarnation;
      IF cu =/= NONE THEN BEGIN
        IF NOT cu.domain IN monitorc THEN BEGIN                   1680
          IF cu.processor.c_time GE longtimeslice THEN BEGIN
            cu.processor.rts:=longtimeslice;
            cu.processor.priority:=low;
          END ELSE cu.processor.rts:=timeslice;
          cu.Out;                                                 1685
          IF driverq.b_entries=0 AND
          dmn_info(cu.did).next_site=0
          THEN
          !no other work and domain not wanted elsewhere;
          queue(driverq,cu,cu.processor.priority)                 1690
          ELSE BEGIN
          !redetermine best location for domain_incarnation to;
          !continue - gives domain chance to go to other sites;
            retire(cu);
            queue(driverq,cu,high);                               1695
          END;
        END ELSE cu.processor.rts:=timeslice;
        !give processor time to get out of monitor domain;
```

```
  END;
END                                                         1700



WHEN consolec DO                                            1705
BEGIN                   !input  has arrived from a console;
  IF c_secretary.Prev == NONE THEN     !see if in a queue;
  BEGIN
    !assume that it is not in driverq.;
    !note this means domain must be tied;                   1710
    queue(driverq,c_secretary,medium);
    c_secretary.stage:=valid;
  END;
  ptr.Into(c_arrival_list);
  !messages from all consoles kept in one list;             1715
END


WHEN dmn_transfer DO
BEGIN                   !request to transfer domain has arrived;
  IF dmn_info(did).next_site NE 0 THEN
  !domain promised to someone else;
  send_message(dmn_info(did). next_site,ptr)
  !pass on the request;
  ELSE IF NOT (dmn_info(did).here OR dmn_info(did).coming)
  THEN                  !domain already gone elsewhere;
  send_message(dmn_info(did).d_loc,ptr)
  ELSE                  !domain is here or coming;
  IF rqstor NE id THEN BEGIN
    INSPECT dmn_info(did) DO BEGIN                          1730
      d_loc:=next_site:=rqstor;
      !prepare for updating broadcast;
      IF update(1)==NONE THEN
      BEGIN             !update not initialized;
        FOR w := 1 STEP 1 UNTIL n DO                        1735
        update(w):-NEW d_loc_update(did,rqstor);
      END
      ELSE
      FOR w:=1 STEP 1 UNTIL n DO
      update(w).new_site:=rqstor;                           1740

    END;
    action_transfer(dmn_info(did));
    !see if domain can be sent off immediately;
                                                            1745
  END ELSE error("DOMAIN MANAGEMENT HAS FAILED");
END


WHEN domainc DO                                             1750
BEGIN                   !domain has arrived from another site;
  site:=id;
  INSPECT dmn_info(did) DO
  BEGIN
    REF(domain_incarnationc) dr,f;                          1755
```

```
    IF NOT coming THEN
    BEGIN          !arrival unexpected e.g. from ipl;
      claim(size,running);
      !memory space;
      IF NOT running THEN                                   1760
      error("NO ROOM FOR A NEW DOMAIN");
      FOR w:=1 STEP 1 UNTIL n DO
      update(w):-NEW d_loc_update(did,id);
      broadcast(update);
      !tell other sites it is here;                         1765
      d_loc:=id;
    END;
    coming:=FALSE;
    here:=TRUE;    !alter state;
    WHILE NOT rqst_list.Empty DO                            1770
    BEGIN          !examine list of waitng domain requests;
      dr:-rqst_list.Suc;
      queue(driverq,dr,high);

      !have chosen a strategy such that  a request that;
      !is not going to be filled at this site is not acted;
      !upon if domain has been promised elsewhere;
    END;
    dr:-external_segs.Suc;
    WHILE dr=/=NONE DO                                      1780
    !check incarnation(s) that have been waiting for;
    !domain to arrive to see if any are ready to run;
    BEGIN
      f:-dr.Suc;
      IF dr.count=0 THEN                                    1785
      BEGIN
        dr.Out;
        put_in_ready_state(dr);
      END;
      dr:-f;                                                1790
    END;
  END;
END

                                                            1795
WHEN d_loc_update DO
BEGIN              !a domain has changed sites;
  dmn_info(did).d_loc:=new_site;
END
                                                            1800
WHEN l_seg_listc DO
BEGIN
!request to transfer a list of local segments;
  INTEGER i;
  REF (segmentc) s;                                         1805
  FOR i:=1 STEP 1 UNTIL max_local_segs DO IF a(i) NE 0 THEN
  BEGIN
    s:-seg_table(retrieve(a(i)));        !fetch segment;
    xfered_locals.incr;                  !update statistics;
    send_message(rqstor,s);              !transmit;         1810
  END;
END
```

```
WHEN p_seg_listc DO
BEGIN                                                             1815
  !request to transfer processor segment and;
  !possible parameters;
  INTEGER i;
  send_message(rqstor,seg_table(retrieve(p_key)));
  xfered_processors.incr;                                         1820
  !send processor segment;
  FOR i:=1 STEP 1 UNTIL max_param_segs DO
  IF a(i) NE 0 THEN BEGIN
    send_message(rqstor,seg_table(retrieve(a(i))));
    xfered_locals.incr;                                           1825
  END;
END

WHEN s_channelc DO
BEGIN               !transmission to another site completed;
  REF(contentc) m;
  m:-ptr QUA s_channelc.m;
  !retain reference to message just sent;

  !now see if more messages to be sent;                           1835
  IF NOT(cq.Empty AND sq.Empty) THEN
  signal_channel
  ((IF cq.Empty THEN sq.Suc ELSE cq.Suc) QUA contentc)
  ELSE              !none left;
  channel_busy:=FALSE;                                            1840

  !now free space from previous message;
  !(could be time consuming);
  IF m IN domainc THEN domain_copy(m QUA domainc)
  !determine if keeping a copy of domain code;                    1845
  ELSE IF m IN segmentc THEN delete_seg(m QUA segmentc);
  !return space;
  !short control messages are assumed not to interact with;
  !memory management;
END                                                               1850


WHEN segmentc DO    !domainc already dealt with;
BEGIN               !processor or local segment has arrived;
                                                                  1855
  INTEGER new_did;
  REF(virtual_processor) p;
  REF(domain_incarnationc) f;
  IF ptr IN virtual_processorc THEN BEGIN
    p:-ptr;                                                       1860
    new_did:=p.e_stack(p.stackp).did;
  END
  ELSE BEGIN
    !assume that it is local segment;
    p:-ptr QUA local_seg.pr;                                      1865

    new_did:=ptr QUA local_seg.did;
  END;
```

```
    !locate incarnation they belong to;                          1870
    f:-dmn_info(new_did).external_segs.Suc;
    WHILE (IF f==NONE THEN FALSE ELSE f.processor=/=p) DO
    f:-f. Suc;
    !search for incarnation record;
    IF f==NONE THEN error("SEGMENT HAS LOST ITS DOMAIN") 1875
    ELSE
    BEGIN
      f.count:=f.count-1;
      !one  more segment has arrived;
      IF f.count=0 AND (dmn_info(new_did).here OR            1880
      dmn_info(new_did).copy) THEN
      BEGIN
        f.Out;       !of external_segs list;
        put_in_ready_state(f);
        !can run because all segments here;                   1885
      END;
    END;
  END

                                                              1890


  OTHERWISE BEGIN
    IF NOT spaceqempty THEN
    BEGIN                                                     1895
    !have processes waiting for space but.none running;
      IF spaceq.total_entries=qfs(id) AND
      NOT deadlock_warning
      THEN
      BEGIN                                                   1900
        ptime;
        Outtext("POSSIBLE DEADLOCK AT SITE ");
        Outint(id,2);
        Outtext("   NUMBER WAITING=");
        Outint(spaceq.total_entries,2);                       1905
        Outimage;
        deadlock_warning:=TRUE;
      END;
    END;
    idle_timer.start;                                         1910
    Passivate;
    idle_timer.stop;
    !accumulate idle time;
  END;
 END;                                                         1915
END;
error("KERNEL HAS STOPPED RUNNING");
END of class kernelc;
```

```
!declarations related to domain management;                           1920



CLASS dmn_infoc;                                                      1925
!this class is a data structure holding all information that the;
!kernel needs to know about a domain;
BEGIN
  REF(domainc) d;         !actual reference to domain;
  BOOLEAN                                                             1930
  copy,
  !true if only a copy of the domain is here;
  coming,
  !true when this site has requested domain to be sent here;
  here,                                                               1935
  !true when domain is actually at this  site;
  going;
  !true when domain is in transit between sites;
  INTEGER
  d_loc,                  !estimate of site of original domain;
  next_site;
  !non-zero when domain reserved for another site;
  REF(Head)
  work,
  !list of all 'ready to run' incarnations of domain;          1945
  external_segs,
  !list of all incarnations waiting for segments;
  !from other sites;
  rqst_list;
  !list of all rqsts to have optimum site calculation performed;
  REAL lasttime;          !for copies last time it was used;
  REF(d_loc_update) ARRAY update(1:n);
  !kept for efficieny reasons;

  work:-NEW Head; external_segs:-NEW Head; rqst_list:-NEW Head;
  copy:=FALSE;
  coming:=FALSE;
  here:=FALSE;
  next_site:=0;
END of class dmn_infoc;                                               1960



contentc CLASS dmn_transfer;
BEGIN                                                                 1965
  INTEGER did,rqstor;
  !for requesting the transmission of code and possible public;
  !segments to another site (rqstor);
  size:=32;
END of class dmn_transfer;                                            1970


contentc CLASS d_loc_update(did,new_site);
INTEGER did,new_site;
!for informing sites of new domain locations;                         1975
BEGIN
```

```
  TEXT PROCEDURE dump;
  dump:-Copy("D_LOC_UPDATE");

  size:=32;                                                    1980
END;


contentc CLASS domain_incarnationc(processor);
REF(virtual_processorc) processor;                             1985
!this is basically an entry capability;
!the data it contains is fleshed out by various procedures;
!in the kernel to make the capability valid;
BEGIN
  INTEGER stage;                                               1990
  !progress indicator in making capability valid;
  INTEGER did;
  !identity of domain (going to be) entered;
  REF(domainc) domain;   !set just before entering domain;
                                                               1995
  REF (local_seg) ARRAY locals(1:max_local_segs);

  INTEGER choice;
  !site calculated as 'best' for incarnation;
                                                               2000
  INTEGER shifts;
  !count of number of times incarnation forced to another site;

  INTEGER next_step;     !entry point information;
  REAL runtt;                                                  2005

  REAL local_data;
  !required to simulate data stored in local_segments;

  !the above are all that are strictly necessary;              2010
  !the rest aid computation;
  INTEGER d_site,p_site,l_site;
  !sites of domain, processor and parameter, and local segments;
  INTEGER d_size,p_size,l_size;
  !size of code (& data-base), processor (& parameter);         2015
  !and local segments;

  INTEGER total_size,
  !of all segments except domain segments but including those;
  !that are to be created and on disk;                          2020
  extra_space;
  !size of segments not resident at chosen site or to be created;
  INTEGER count;
  !number of segments required from other sites;
  INTEGER site;                                                2025
  INTEGER i;

  PROCEDURE re_initialization(didn);
  INTEGER didn;
  BEGIN                   !sets capability back to base state;
    stage:=seek_d_site;
    runtt:=0; next_step:=1;
    did:=didn;
```

```
    FOR i:=1 STEP 1 UNTIL max_local_segs DO
    locals(i).status:=locals(i).default:=null;                    2035
  END of re_initialization;


  TEXT PROCEDURE dump;
  BEGIN                                                           2040
  !for diagnostic identification - 12 chars long;
    TEXT dumpv;
    dumpv :- Copy("DI P=  D=   ");
    dumpv.Sub(6,2).Putint(processor.pid);
    dumpv.Sub(10,2).Putint(did);                                  2045
    dump:-dumpv;
  END;

  size:=32;
  !for transmission over communication links;                     2050
  FOR i:=1 STEP 1 UNTIL max_local_segs DO
  locals(i):-NEW local_seg(0,processor);
  !instead of creating a new local segment every time one is;
  !used the same template is used;
                                                                  2055
END of domain_incarnationc;



                                                                  2060
segmentc CLASS domainc(did);
INTEGER did;            !identification number;
VIRTUAL: PROCEDURE
step1,                  !executed on entering a domain;
step2,step3,step4,step5;                                          2065
!optional extra 'instructions';
BEGIN
  INTEGER
  tied;                 !if non-zero fixed site for domain;
  INTEGER i;                                                      2070

  PROCEDURE interdomain_call(x,new_did);
  REF (domain_incarnationc) x;
  INTEGER new_did;
  BEGIN                                                           2075
    interdmn_jump(x,x.processor.fetch_c(new_did));
    !set up new entry capability;
  END;
  PROCEDURE return(x);
  REF(domain_incarnationc) x;                                     2080
  INSPECT x DO BEGIN
    INTEGER i;
    FOR i:=1 STEP 1 UNTIL max_local_segs DO
    IF locals(i).status=incore THEN
    BEGIN                                                         2085
      !determine what to do with local segments;
      IF locals(i).default=null
      THEN
      INSPECT k(x.site) DO delete_seg(locals(i));
    END;                                                          2090
```

```
    interdmn_jump(x,processor.return);
    !retrieve previous entry capability;
  END of return;

  PROCEDURE interdmn_jump(x,y);                                    2095
  REF (domain_incarnationc) x,y;
  !set in train the transfer of virtual processor;
  !from incarnation x to y;
  BEGIN
    FOR i:=1 STEP 1 UNTIL max_param_segs DO                        2100
    x.processor.params(i).did:=y.did;
    !mark parameter segments as belonging to new domain;
    INSPECT k(x.site) DO BEGIN
      retire(x);
      !remove current domain incarnation from driverq;             2105
      queue(driverq,y,high);
    END;
    new_incarnations.incr;                        !statistic;
  END of interdmn_jump;
                                                                   2110
  REF(domain_incarnationc) PROCEDURE putative_return(x);
  REF(domain_incarnationc) x;
  BEGIN
    !gives the entry capability for a return but does not;
    !instigate the return;                                         2115
    return(x);
    x.processor.Current.Out;                      !of driverq;
    putative_return:-x.processor.Current;
  END;
                                                                   2120


  PROCEDURE setup_disk_read(x,size);
  REF(domain_incarnationc) x; INTEGER size;
  !for preparing the parameters of a disk read;                    2125
  x.processor.simple_parameter:=size;
  !considerably simplified;


  TEXT PROCEDURE dump;                                             2130
  BEGIN                  !diagnostic;
    TEXT t;
    t:-Copy("DOMAIN      ");
    t.Sub(7,2).Putint(did);
    dump:-t;                                                       2135
  END;


  key := did;
  default:=status:=incore;                        !always;         2140
  INNER;
  INSPECT k(site) DO int(THIS domainc);           !instal itself;

END;
                                                                   2145
CLASS formatc(did);
INTEGER did;
```

```
!for handling 'compile-time' information about domain structure;
VIRTUAL: PROCEDURE format;
                                                                 2150
BEGIN
  PROCEDURE format(l); REF(local_seg)ARRAY l; ;
  !default;
  dp(did):-THIS formatc;
  !make universally available;                                   2155
END;

REF(formatc) ARRAY dp(l:fixed_domains);

                                                                 2160
domainc CLASS monitorc;
BEGIN
  INTEGER c_size,          !size of code segment;
  db_size;                 !size of public (data_base) segment;
  !n.b. db_size can not be zero.;                                2165


  INNER;
  size:=c_size+db_size;
END of class monitorc;                                           2170

monitorc CLASS secretaryc;
BEGIN
  !in this simulation all the monitors that have condition;
  !queues also have secretaries;                                 2175
  REF(head) my_q;          !conditon queue;



  !a virtual processor can suspend ;
  !itself on the condition queue;                                2180
  PROCEDURE suspend;
  k(site).cu.Into(my_q);
  !this assumes that the domain is tied down;

  PROCEDURE restart_processor;                                   2185
  !removes the first processor from condition queue;
  !because still at same site could be ready to;
  !continue in monitor;
  INSPECT k(site) DO
  BEGIN                                                          2190
    REF(domain_incarnationc) inc;
    inc:-my_q.Suc;
    IF inc=/=NONE THEN queue
    (driverq,inc,IF inc.stage=valid THEN monitor ELSE high);
  END of procedure restart_processor;                            2195

  PROCEDURE wait_for_signal;
  k(site).cu.Out;
  !secretary processor removed from driverq;
                                                                 2200
  PROCEDURE create_secretary(secretary,u);
  NAME secretary;
  REF(domain_incarnationc) secretary;
  INTEGER u;
```

```
  BEGIN                                                              2205
    REF(virtual_processor) pr;
    INSPECT k(site) DO
    BEGIN
      pr:-NEW virtual_processorc(site,max_consoles+did,u);
      claim(pr.size,running);                                        2210
      pr.service_timer.Out;
      !not giving user service;
      secretary:-pr.fetch_c(did);
      secretary.domain:-THIS domainc;
      secretary.stage:=valid;                                        2215
      !always ready to run;
    END;
  END;

  my_q :- NEW Head;                                                  2220
  tied:=site;
  !because secretary is only known at one site;
END of class secretaryc;

                                                                     2225

segmentc CLASS virtual_processor (pid,u);
INTEGER pid,              !identification number;
u;
!random number seed to determine execution path;                     2230
BEGIN
  INTEGER priority;       !determines scheduling;
  REAL rts,               !remaining time slice;
  c_time;
  !processing time received since last command started;              2235
  REF(local_seg) ARRAY params(1:max_param_segs);
  !there is one A_list for the virtual processor;
  !local segments are moved to and from it if necessary;
  INTEGER simple_parameter;
  !for non array type data;                                           2240
  INTEGER unique;
  INTEGER PROCEDURE uniquenumber;
  BEGIN
    unique:=unique+1;
    IF unique < 16rffff  !16bits;                                    2245
    THEN uniquenumber:=unique
    ELSE error("PROCESSOR HAS RUN OUT OF UNIQUE NUMBERS");
  END;

  REF(domain_incarnationc) ARRAY e_stack(1:stack_depth);             2250
  !for keeping control of sequence of domains visited ;
  INTEGER stackp;
  INTEGER sameness;
  !counts how many times processor on same domain call sequence;
                                                                     2255
  REF(domain_incarnationc) PROCEDURE Current;
  !the incarnation the processor is in or will be in when;
  !it next executes;
  Current:-e_stack(stackp);
                                                                     2260
  REF(domain_incarnationc) PROCEDURE fetch_c(did);
```

```
  INTEGER did;
  IF stackp < stack_depth THEN
  BEGIN                   !set up incarnation template;
    stackp:=stackp+1;                                              2265
    IF e_stack(stackp).did=did THEN          !ghost;
    sameness:=sameness+1 ELSE sameness:=1;
    e_stack(stackp).re_initializa tion(did);
    dp(did).format(e_stack(stackp).locals);
    !set up temporary space etc;                                   2270
    fetch_c:-e_stack(stackp);
  END
  ELSE error(fillin("STACK OVERFLOW FOR PROCESSOR",pid));

  REF(domain_incarnationc) PROCEDURE return;                       2275
  IF stackp>1 THEN
  BEGIN
    stackp:=stackp-1;
    return:-e_stack(stackp);
  END ELSE error(fillin("STACK UNDERFLOW FOR PROCESSOR",pid));

  REF(timer) service_timer;

  !the next 4 items do not logically belong here;
  !i.e. not part of processor base segment;                        2285
  !they are included for program efficiency;
  REF(dmn_transfer) d_transfer;
  REF(p_seg_listc) p_seg_list;
  REF(l_seg_listc) l_seg_list;
  REF(disk_rqst) disk_read;                                        2290



  priority:=medium;
  rts:=timeslice;                                                  2295
  service_timer :- NEW timer("SERVICE TIME",usage);
  size := 200;
  key:=pid+fixed_domains;
  status:=default:=incore;
  FOR w:=1 STEP 1 UNTIL stack_depth DO                             2300
  e_stack(w):-NEW domain_incarnationc(THIS virtual_processorc);
  FOR w:=1 STEP 1 UNTIL max_param_segs DO
  params(w):-NEW local_seg(site,THIS virtual_processorc);
  d_transfer:-NEW dmn_transfer;
  p_seg_list:-NEW p_seg_listc(key);                                2305
  l_seg_list:-NEW l_seg_listc;
  disk_read:-NEW disk_rqst;

END of class virtual_processor;
                                                                   2310

contentc CLASS p_seg_listc(p_key);
INTEGER p_key;
BEGIN
  !carries kernel request for processor and parameter segsments;
  INTEGER rqstor;
  INTEGER ARRAY a(1:max_param_segs);
  size:=32;
```

```
END;
                                                                        2320
contentc CLASS l_seg_listc;
BEGIN                    !for requests for local segments;
  INTEGER rqstor;
  INTEGER ARRAY a(1:max_local_segs);
  size:=32;                                                             2325
END;

segmentc CLASS local_seg(pr);
REF(virtual_processor) pr;
BEGIN                                                                   2330
  INTEGER did;
  INTEGER PROCEDURE setkey;
  setkey:=pr.pid*16rffffff+site*16rffff+pr.uniquenumber;
  !created before use and reused for sake of program efficiency;
  !site, size and key have to be given each time;                       2335

  PROCEDURE make_workspace(d_id,newsize);
  INTEGER d_id,newsize;
  BEGIN
    did:=d_id;                                                          2340
    key:=setkey;
    status:=trans;       !so as created on domain entry;
    size:=newsize;
  END;
                                                                        2345
  PROCEDURE make_disk_read(newsize);
  INTEGER newsize;
  BEGIN
    !for setting up the parameter segment for a disk read;
    key:=setkey;                                                        2350
    status:=ondisk;
    size:=newsize;
  END;

  TEXT PROCEDURE dump;                                                  2355
  dump:-fillin("L_SEG * P",pr.pid);
END of class local_seg;


PROCEDURE move(lfrom,lto);                                              2360
REF(local_seg) lfrom,lto;
!for transferring a segment into or out of A_list;
BEGIN
  !an error in simula runtime system prevents swopping 'REFs';
  INSPECT lfrom DO BEGIN                                                2365
    lto.pr:-pr;
    lto.size:=size;
    lto.site:=site;
    lto.key:=key;
    lto.default:=default;                                               2370
    lto.status:=status;
    lto.did:=did;
    !assume that lto previous status not incore;
    IF status=incore OR status=desc THEN INSPECT k(site) DO
    seg_table(retrieve(key)):-lto;                                      2375
```

```
    END;
    !keep the same number of segment templates in the system;
    lfrom.status:=null;
  END;
                                                              2380
```

```
PROCEDURE ipl;
BEGIN INTEGER ul,nn,j;
  ul:=random_seed;        !random number seed;

  NEW user_supervisor(1,supern);                                    2385
  !simulate part of ipl;
  NEW command(cnsl_site,commandn);
  FOR nn:=1 STEP 1 UNTIL max_disks DO
  NEW diskhandler(nn*(n//max_disks),diskl+nn-1,n+nn,ul*(2*nn+1));
  !spread handlers around to different sites;                      2390

  FOR nn:=ipld+1 STEP 1 UNTIL ipld+compl DO
  NEW compiler(Mod(nn,n)+1,nn);
  FOR nn:=ddl STEP 1 UNTIL ddu DO
  NEW type1(Mod(nn,n)+1,nn);                                        2395
  FOR nn:=mntrl STEP 1 UNTIL mntru DO
  NEW type2(Mod(nn,n)+1,nn);

  FOR nn:=1 STEP 1 UNTIL max_consoles DO
  BEGIN                                                             2400
    console(nn) :-  NEW consolec(nn,(2*Randint(1,1000,ul)+1));
    ACTIVATE console(nn) DELAY 5000;
    !give system time to settle down;
  END;
END of procedure ipl;                                               2405
```

```
domainc CLASS type1;
BEGIN
  !this class simulates trivial commands;
  !overall about half the commands make at least 1 disk transfer;
  !the average processor time is around 200 msec;

  PROCEDURE step1(x);
  REF(domain_incarnationc) x;
  BEGIN                                                                  2415
    IF full_diags THEN BEGIN
      TEXT t;
      t:-Copy("PROCESSOR    ENTERS DOMAIN    AT SITE    ");
      t.Sub(11,2).Putint(x.processor.pid);
      t.Sub(28,2).Putint(did);                                           2420
      t.Sub(39,2).Putint(x.site);
      ptime; Outtext(t);
    END;
    x.next_step:=2;
    x.runtt:=Negexp(1/100000,x.processor.u);                             2425
  END of procedure step1;

  PROCEDURE step2(x);
  REF (domain_incarnation) x;
  BEGIN                                                                  2430
    IF Draw(0.25,x.processor.u) THEN
    BEGIN                       !going to make a disk transfer;
      setup_disk_read(x,512);                  !read 512 bytes;
      interdomain_call(x,diskhandlern(x.processor.pid));
      x.next_step:=5;  x.runtt:=200;                                     2435
    END ELSE BEGIN
      x.next_step:=3; x.runtt:=0;
    END;
  END of step2;
                                                                         2440
  PROCEDURE step3(x);
  REF(domain_incarnationc) x;
  !determine next domain to be entered;
  BEGIN
    REF(virtual_processor) p;                                            2445
    INTEGER new_did;
    p:-x.processor;
    IF Draw(0.5,p.u)
    AND p.stackp<stack_depth THEN BEGIN
      new_did := Randint((did+1),mntru,p.u);                             2450
      IF full_diags THEN BEGIN
        ptime; Outint(p.pid,6); Outtext("  CHOOSES");
        Outint(new_did,6);
      END;
                                                                         2455
      interdomain_call(x,new_did);
    END;
    x.next_step:=4;
    x.runtt:=100;
  END of procedure step3;                                                2460

  PROCEDURE step4(x);
  REF(domain_incarnationc) x;
```

```
  BEGIN
    IF full_diags THEN BEGIN                                          2465
      ptime; Outint(x.processor.pid,6);
      Outtext(" RETURNS TO/REMAINS AT");
      Outint(did,4); Outimage;
    END;
    return(x);                                                        2470
  END of procedure step4;

  PROCEDURE step5(x);
  REF(domain_incarnationc) x;
  BEGIN                  !completed a disk read;                      2475
    move(x.processor.params(1),x.locals(2));
    x.next_step:=3;  x.runtt:=0;
  END of step5;

                                                                      2480
  size:=1024;
  NEW typelf(did);       !to set up domain correctly;
END of class typel;

formatc CLASS typelf;                                                 2485
BEGIN
  PROCEDURE format(l);
  REF(local_seg) ARRAY l;
  l(1).make_workspace(did,(ddu+2-did)*128);
  !different size for each domain to get different behaviour;
END of typelf;
```

```
monitorc CLASS type2;
!this class simulates actions involving the use of global tables;
BEGIN                                                                   2495

  PROCEDURE step1(x);
  REF(domain_incarnationc) x;
  BEGIN
    IF full_diags THEN BEGIN                                            2500
      TEXT t;
      t:-Copy("PROCESSOR    ENTERS MONITOR    AT SITE    ");
      t.Sub(11,2).Putint(x.processor.pid);
      t.Sub(29,2).Putint(did);
      t.Sub(40,2).Putint(x.site);                                       2505
      ptime; Outtext(t);
    END;
    x.next_step:=2;
    x.runtt:=Negexp(1/100000,x.processor.u);
  END of step1;                                                         2510

  PROCEDURE step2(x);
  REF(domain_incarnationc) x;
  return(x);
                                                                        2515
  c_size:=400;
  db_size:=(mntru-did+1)*max_consoles*32;
  !expect global tables to be bigger the more users there are;
  NEW type2f(did);
END of type2;                                                           2520

formatc CLASS type2f;
BEGIN
  PROCEDURE format(l);
  REF(local_seg) ARRAY l;                                               2525
  l(1).make_workspace(did,10*did);
  !space for a small stack;
END of type2f;
```

```
domainc CLASS compiler;                                                     2530
!this domain interacts only with the diskhandler and makes large;
!computational demands;
BEGIN
  REAL ARRAY a,b(1:7);
  PROCEDURE step1(x);                                                       2535
  REF(domain_incarnation) x;
  BEGIN
    IF full_diags THEN BEGIN
      TEXT t;
      t:-Copy("PROCESSOR    ENTERS COMPILER");                              2540
      t.Sub(11,2).Putint(x.processor.pid);
      ptime; Outtext(t);
    END;
    x.next_step:=2;
    x.local_data:=Linear(a,b,x.processor.u)* 1.0&6;                         2545
  END of step1;

  PROCEDURE step2(x);
  REF(domain_incarnation) x;
  BEGIN                                                                     2550
    IF x.processor.params(1).status=incore THEN
    INSPECT k(x.site) DO delete_seg(x.processor.params(1));
    !free previous buffer;
    IF x.local_data > 0 THEN
    BEGIN                                                                   2555
      x.runtt:=Erlang(1.0&-6,6,x.processor.u);
      !we assume an average of 12 lines/sec compilation speed;
      !with 2 disk transfers for every 12 lines;
      x.local_data:=x.local_data-x.runtt;
      setup_disk_read(x,4096);                                              2560
      interdomain_call(x,diskhandlern(x.processor.pid));
      x.next_step:=3;
    END
    ELSE
    return(x);                                                              2565
  END of step2;

  PROCEDURE step3(x);
  REF(domain_incarnationc) x;
  BEGIN                    !disk write;                                     2570
    x.processor.params(1).status:=desc;
    interdomain_call(x,diskhandlern(x.processor.pid));
    x.next_step:=2;
    x.runtt:=500;
  END of step3;                                                             2575


  size:=12000;
  a(1):=0; a(2):=0.27; a(3):=0.54; a(4):=0.78; a(5):=0.88;
  a(6):=0.93; a(7):=1.0;                                                    2580
  b(1):=0.5; b(2):=4; b(3):=8; b(4):=16; b(5):=40;
  b(6):=80; b(7):=200;
  !cumulative density function for cpu times;
  NEW type1f(did);
                                                                            2585
END of class compiler;
```

A
BLANK
PAGE

```
domainc CLASS user_supervisor;
BEGIN
                                                                      2590
  !supervise the 'interpretation' of users code;
  !can soak up a lot of cpu time;

  PROCEDURE stepl(x);
  REF(domain_incarnationc) x;                                         2595
  BEGIN
    IF full_diags THEN BEGIN
      ptime;    Outtext(fillin
      ("EXECUTION OF USER CODE BY PROCESSOR",x.processor.pid));
    END;                                                              2600
    x.local_data:=Linear(userp,usert,x.processor.u)*&6;
    setup_disk_read(x,Randint(300,16000,x.processor.u));
    !for code from disk;
    interdomain_call(x,diskhandlern(x.processor.pid));
    x.next_step:=2                                                    2605
  END of stepl;

  PROCEDURE step2(x);
  REF (domain_incarnationc) x;
  BEGIN                                                               2610
    move(x.processor.params(1),x.locals(2));
    !code lives in  local segment 2;
    x.runtt:=200;
    x.next_step:=3;
  END of step2;                                                       2615

  PROCEDURE step3(x);
  REF(domain_incarnationc) x;
  IF x.local_data>0 THEN BEGIN
    x.runtt:=Negexp(1/250000,x.processor.u);                          2620
    !mean headway between disc operations is 250msecs;
    !this is twice the equivalent emas rate but we must allow for
    !overlays or equivalent;
    x.local_data:=x.local_data-x.runtt;
    INSPECT x.processor DO                                            2625
    BEGIN
      IF Draw(0.5,u) AND params(1).status=incore THEN
      BEGIN
        params(1).status:=desc;              !for write;
      END                                                             2630
      ELSE

      BEGIN
        IF params(1).status=incore THEN INSPECT k(x.site) DO
        delete_seg(params(1));                                        2635
        simple_paramater:=8192;              !for read;
      END;
    END;
    interdomain_call(x,diskhandlern(x.processor.pid));
  END                                                                 2640
  ELSE
  BEGIN
    IF x.processor.params(1).status=incore THEN
    INSPECT k(x.site) DO delete_seg(x.processor.params(1));
```

```
    return(x);              !supervisor finished;                   2645
  END of step3;

  size:=4000;
  NEW supervisorf(did);
END of class user_supervisor;                                       2650

formatc CLASS supervisorf;
BEGIN
  PROCEDURE format(l);
  REF (local_seg) ARRAY l;                                          2655
  BEGIN
    l(l).make_workspace(did,Randint(500,8000,l(l).pr.u));
  END;
END;
                                                                    2660
```

```
secretaryc CLASS diskhandler(disk,u);
INTEGER disk,u;
BEGIN
  !This domain handles both writes to and reads from disk.;
  !Requests are queued depending on the priority of the;      2665
  !processors making the request.;
  !When in overload situation low priority processors are;
  !suspended after they have issued write requests until;
  !situation improves.;
                                                               2670
  REF(virtual_processor) pr;
  REF(domain_incarnationc) next_inc;

  BOOLEAN transfer_in_progress,
  read_next,              !true if previous operation was a write;
  overload;
  !set to give priority to writes to free memory;

  REF(Head) wfreeq,wq;
  !used to manage limited read before write scheme;           2680
  REF(Head) wql,wqh;
  !low and high priority write queues feed into wq;
  REF(Head) rql,rqh;      !read queues;
  REF(l_seg_listc) disk_write_rqst;
  REF(Link) trans_seg;                                         2685
  BOOLEAN warning_message;
  !true when printed warning about deadlock possibility;

  PROCEDURE signal_disk_write;
  BEGIN                                                        2690
  !the kernel where the segment to be written resides is;
  !requested to send the segment straight to the disk;
    disk_write_rqst.a(1):=wq.Suc QUA segmentc.key;
    INSPECT k(site) DO
    send_message(wq.Suc QUA segmentc.site,disk_write_rqst);   2695
    trans_seg:-wq.Suc; trans_seg.Out;
    IF NOT(wqh.Empty AND wql.Empty) AND NOT wfreeq.Empty THEN
    BEGIN
    !move another write request into pipeline;
      next_inc:-IF wqh.Empty THEN wql.Suc ELSE wqh.Suc;       2700
      move(next_inc.processor.params(1),wfreeq.Suc
      QUA local_seg);
      wfreeq.Suc.Into(wq);

      IF next_inc.processor.priority=low THEN                  2705
      BEGIN
        IF overload OR NOT my_q.Empty THEN
        BEGIN
          next_inc.Into(my_q);
          !hold back to allow other processors to complete;   2710
          IF overload THEN next_inc:-NONE
          ELSE next_inc:-my_q.Suc;
        END;
      END;
      IF next_inc=/=NONE THEN                                  2715
      INSPECT k(site) DO queue(driverq,next_inc,monitor);
      !in monitor and still valid;
```

```
  END;
END;
                                                                        2720

BOOLEAN PROCEDURE initiate_read;
!if there is a read to be done, starts it and returns true;
BEGIN
  initiate_read:=NOT(rqh.Empty AND rql.Empty);                          2725
  IF NOT rqh.Empty THEN
  k(site).send_message(disk,rqh.Suc QUA contentc)
  ELSE
  IF NOT rql.Empty THEN
  k(site).send_message(disk,rql.Suc QUA contentc);                      2730
END of initiate_read;


PROCEDURE step1(x);
REF(domain_incarnationc) x;                                             2735
BEGIN
  IF full_diags THEN BEGIN
    ptime;
    Outtext(fillin("DISK HANDLER ACTIVATED FOR PROCESSOR",
    x.processor.pid));                                                  2740
    Outimage;
  END;
  x.runtt:=200;
  x.next_step:=2;
END of step1;                                                           2745

PROCEDURE step2(x);
REF(domain_incarnationc) x;
!for a disk write params(1) has a segment desc(riptor) while;
!for a read simple_parameter describes the requirements;                2750

IF x.processor.params(1).status=desc THEN
BEGIN                    !write;
  IF NOT wfreeq.Empty THEN
  BEGIN                  !go straight into delay line buffer;
    move(x.processor.params(1),wfreeq.Suc QUA local_seg);
    wfreeq.Suc.Into(wq);
    IF NOT transfer_in_progress THEN
    BEGIN
      !can use disk straight away;                                      2760
      signal_disk_write;
      transfer_in_progress:=TRUE;
    END;
  END ELSE BEGIN
    !delay line full so processor has to wait;                          2765
    IF x.processor.priority=low THEN x.Into(wql)
    ELSE x.Into(wqh);
  END;
  x.next_step:=3;
  x.runtt:=400;                                                         2770
END
ELSE
BEGIN
  !entered to do a disk read;
```

```
  x.processor.params(1).make_disk_read                             2775
  (x.processor.simple_parameter);
  x.processor.params(1).site:=disk;
  !identify disk that will read segment;
  next_inc:-putative_return(x);
  !prepare entry capability for when read complete;                2780
  IF transfer_in_progress THEN
  next_inc.Into(IF x.processor.priority=low THEN rql ELSE rqh)
  !suspend for secretary to look after;
  ELSE
  BEGIN                                                            2785
    transfer_in_progress:=TRUE;
    INSPECT k(site) DO send_message(disk,next_inc);
  END;
END of step2;
                                                                   2790
PROCEDURE step3(x);
REF (domain_incarnationc) x;
BEGIN
  return(x);
END of step3;                                                      2795

PROCEDURE step4(x);
!executed by secretary processor;
REF(domain_incarnationc) x;
BEGIN                                                              2800
  !in a normal situation read done before writes;
  !unless wq is full in which case done turn;
  !and turn about;
  !in overload situation all writes and high priority reads;
  !are done turn and turn about;                                   2805

  IF trans_seg=/=NONE THEN BEGIN
    trans_seg:-NONE;
    read_next:=TRUE;
  END ELSE read_next:=FALSE;                                       2810

  IF read_next THEN overload:=k(site).overload;

  IF overload THEN
  BEGIN                                                            2815
    IF NOT((read_next AND NOT rqh.Empty) OR wq.Empty) THEN
    signal_disk_write
    ELSE
    IF NOT initiate_read THEN
    BEGIN                                                          2820
      transfer_in_progress:=FALSE;
        !could be in trouble here as nothing to do yet;
        !system close to deadlock;
      IF NOT warning_message THEN BEGIN
        ptime; Outtext("DISKHANDLER IMPOTENT"); Outimage;          2825
      warning_message:=TRUE; END;
    END
  END
  ELSE
  IF NOT(wql.Empty AND wqh.Empty) THEN                             2830
  BEGIN
```

```
      IF NOT read_next OR (rql.Empty AND rqh.Empty) THEN
      signal_disk_write
      ELSE initiate_read;
    END                                                          2835
    ELSE
    BEGIN
    !the normal situation - reads before writes;
      IF NOT initiate_read THEN
      BEGIN                                                      2840
        IF NOT wq.Empty THEN signal_disk_write
        ELSE
        BEGIN
          !nothing to do;
          transfer_in_progress:=FALSE;                           2845
          IF NOT my_q.Empty THEN BEGIN
            restart_processor;
            !judge it safe to release one held up processor;
            warning_message:=FALSE;
          END;                                                   2850
        END;
      END;
    END;
    wait_for_signal;
    x.runtt:=200+300*(my_q.Cardinal+rql.Cardinal);               2855
    !the bigger the disk queues the more time spent;
    !manipulating them;
  END of step4;

  c_size := 512; db_size:=2048;                                  2860
  wq:-NEW Head;
  wfreeq:-NEW Head;
  wqh:-NEW Head;
  wql:-NEW Head;
  rqh:-NEW Head;                                                 2865
  rql:-NEW Head;
  disk_write_rqst:-NEW l_seg_listc;
  disk_write_rqst.rqstor:=disk;
  FOR w:=1 STEP 1 UNTIL max_writes_pending DO
  NEW local_seg(0,NONE).Into(wfreeq);                            2870
  NEW formatc(did);
  INSPECT k(site) DO BEGIN
    create_secretary(d_secretary,u);
    d_secretary.next_step:=4;
    d_secretary.runtt:=200;                                      2875

    ACTIVATE NEW disk_controller(disk,site,d_secretary) DELAY 0;
  END;
END of diskhandler;
                                                                 2880
INTEGER PROCEDURE diskhandlern(pid);
INTEGER pid;
!rule for associating virtual processors with disks;
diskhandlern:=disk1+Mod(pid,max_disks);
                                                                 2885
```

```
secretaryc CLASS command;
!this domain analyses console input - in a random fashion;
BEGIN
  REF(virtual_processor) pr;
  REF(Head) c_wait_list;                                          2890
  INTEGER active_processors;


  PROCEDURE step1(x);
  REF(domain_incarnationc) x;                                     2895
  BEGIN
    x.next_step:=2; x.runtt:=200;
  END of step1;

  PROCEDURE step2(x);                                             2900
  REF(domain_incarnationc) x;
  BEGIN
    REAL r; INTEGER new_did;
    r:=Uniform(0,1,x.processor.u);
    new_did:=                                                     2905
    IF r<0.09 THEN Randint(ipld+1,ipld+compl,x.processor.u)
    ELSE IF r<0.26 THEN supern              !user program;
    ELSE Randint(ddu,mntru,x.processor.u);  !type1 or type2;
    interdomain_call(x,new_did);
    !choose new domain;                                           2910
    IF new_did < ddl THEN
    BEGIN
      !see if can accept another incarnation of a large domain;
      IF k(site).overload OR active_processors>chopfactor OR
      NOT c_wait_list.Empty THEN BEGIN                            2915
        x.processor.Current.Into(c_wait_list);
        chopcount.incr;  !statistics;
        IF active_processors<chopfactor AND NOT k(site).overload
        THEN
        queue(k(site).driverq,c_wait_list.Suc,high)               2920
        !free first in queue if nothing overloaded;
        ELSE
        BEGIN
          active_processors:=active_processors-1;
          k(site).qfs(site):=k(site).qfs(site)-1;                 2925
          !taken out of driverq to wait until system is;
          !less congested;
        END;
      END;
    END;                                                          2930
    x.next_step:=3; x.runtt:=50;
  END of step2;

  PROCEDURE step3(x);
  REF(domain_incarnationc) x;                                     2935
  BEGIN                 !executes here when finished processing;
    ACTIVATE console(x.processor.pid) DELAY contextdelay;
    !notify console that service is complete;

    x.next_step:=1; x.runtt:=0;                                   2940
    active_processors:=active_processors-1;
    k(site).retire(x);   !in theory should go in my_q;
```

```
    k(site).qfs(site):=k(site).qfs(site)-1;
    !see if any chopped processors can run;
    IF (IF c_wait_list.Empty THEN FALSE ELSE                       2945
    active_processors=0 OR (active_processor<chopfactor AND
    NOT k(site).overload)) THEN
    BEGIN
      INSPECT k(site) DO BEGIN
        queue(driverq,c_wait_list.Suc,high);                       2950
        qfs(id):=qfs(id)+1;
        active_processors:=active_processors+1;
      END;
    END;
  END of step3;                                                    2955

  PROCEDURE step4(x);
  REF(domain_incarnationc) x;
  !this is the code executed by the secretary processor;
  BEGIN                                                            2960
    REF(linkage) ptr;
    ptr:-c_arrival_list;
    FOR ptr:-ptr.Suc WHILE ptr=/=NONE DO BEGIN
      pr:-ptr QUA consolec.pr;
      INSPECT k(site) DO BEGIN                                     2965
        queue(driverq,pr.e_stack(pr.stackp),high);
        !removes from my_q (if there);
        active_processors:=active_processors+1;
        qfs(id):=qfs(id)+1;
      END;                                                         2970
    END;
    c_arrival_list.Clear;
    !all outstanding inputs dealt with;
    wait_for_signal;
    x.runtt:=200;                                                  2975
  END of step4;


  !initialization;
  NEW formatc(did);                                                2980
  c_size:=512; db_size:=120*max_consoles;    !buffer space;
  INSPECT k(site) DO BEGIN
    create_secretary(c_secretary,1);
    c_secretary.next_step:=4;
    c_secretary.runtt:=200;                                        2985
  END;
  c_arrival_list:-NEW Head;
  c_wait_list:-NEW Head;
END of class command;
                                                                   2990
```

```
REF(counter) channel_use;
REF(counter) control_count;

Process CLASS s_channelc(orgn);
INTEGER orgn;                                                        2995
!it is assumed that each site can transmit to only one other;
! site at a time but that a site can receive many;
!messages simultaneously;
BEGIN
  REF(contentc) m;       !all messages are of type contentc; 3000

  PROCEDURE initiate(message);
  REF(contentc) message;
  BEGIN
    m:-message;                                                      3005
    !channel deals with one message at a time;
    ACTIVATE THIS s_channelc DELAY(m.size*mesdelay);
    !time to transmit;
  END;
                                                                     3010

  WHILE running DO BEGIN
    channel_use.add(m.size);                     !update statistics;
    IF m.size=32 THEN control_count.incr;
    IF m.dest LE n THEN                                              3015
    BEGIN
      k(m.dest).int(m)   !message arrival is signalled;
    END
    ELSE dsk(m.dest).int(m);
                                                                     3020
    INSPECT k(orgn) DO BEGIN
      queue(driverq,THIS s_channelc,high);
      switch_context;
      !interrupt kernel to notify end of transmission;
    END;                                                             3025
    !dealt with message;
    Passivate;
  END;
END of s_channelc;
                                                                     3030
```

```
REF(Head) c_arrival_list;
!part of the public segment of the command domain;


Process CLASS consolec(userid,u);                                    3035
INTEGER userid,
u;                          !random number seed;
BEGIN
  REF(virtual_processor) pr;
  REF(counter) completed_commands;                                   3040
  REF(timer) response_timer,think_timer;
  REAL rt;
  REAL PROCEDURE thinktime;
  thinktime:= Negexp(1/30,u)*&6;
                                                                     3045
  NEW groupheading(fillin("CONSOLE",userid));
  think_timer :- NEW timer("THINKING TIME",NONE);
  response_timer :- NEW timer("RESPONSE TIME",total_response);

  pr:- NEW virtual_processor(cnsl_site,userid,u);                    3050
  pr.fetch_c(commandn);
  !initialize virtual processor to serve console;
  completed_commands:-NEW counter("COMPLETED COMMANDS");

  INSPECT k(cnsl_site) DO                                            3055
  BEGIN
    claim(pr.size,running);
    !space for process base;

    WHILE running DO                                                 3060
    BEGIN
      think_timer.start;
      Hold(thinktime);
      think_timer.stop;
      queue(driverq,THIS consolec,high);                             3065
      switch_context;
      IF full_diags THEN BEGIN
        Outimage;
        ptime; Outtext("INPUT FROM CONSOLE #"); Outint(userid,3);
      END;                                                           3070

      rt:=Time;
      pr.c_time:=0;

      response_timer.start;                                          3075
      Passivate;          !wait until processing finished;
      response_timer.stop;
      !accumulate response times;
      rt:=Time-rt;
      IF pr.c_time<longtimeslice THEN                                3080
      BEGIN               !analyse response to trivial command;
        short_commands.incr;
        IF rt>2.0&6 THEN over2.incr;
        IF rt>5.0&6 THEN over5.incr;
      END ELSE non_trivial.data(pr.c_time,rt);                       3085
      completed_commands.incr;
      !another command completed;
```

```
      IF full_diags THEN BEGIN
        ptime; Outtext("OUTPUT TO CONSOLE #"); Outint(userid,3);
        Outimage;                                                     3090
      END;
    END;
  END;
END of class consolec;
                                                                      3095

REF(consolec) ARRAY console(1:max_consoles);

PROCEDURE thinking_consoles;
BEGIN                                                                 3100
  INTEGER i,j;
  Outtext("  NUMBER OF CONSOLES IN THINKING STATE");
  FOR i:=1 STEP 1 UNTIL max_consoles DO
  IF NOT console(i).Idle THEN j:=j+1;
  Outint(j,4);                                                        3105
  Outtext("  NUMBER AWAITING ENTRY TO SYSTEM");
  Outint(k(cnsl_site).c_secretary.domain QUA
  command.c_wait_list.Cardinal,4);
  Outimage;
END;                                                                  3110
```

```
Process CLASS clockc(id);
INTEGER id;                    !site it belongs to;
BEGIN
  PROCEDURE set(interval);                                        3115
  REAL interval;
  REACTIVATE THIS clockc DELAY interval;

  PROCEDURE halt(remaining);
  NAME remaining;                                                 3120
  REAL remaining;          !time of processors time slice;
  IF THIS clockc=/=Current AND NOT THIS clockc.Idle THEN
  BEGIN
  !called when want to suspend flow of time;
    remaining:=THIS clockc.Evtime-Time;                           3125
    Cancel(THIS clockc);
  END ELSE remaining:=0;

  WHILE TRUE DO BEGIN
    INSPECT k(id) DO BEGIN                                        3130
      queue(driverq,THIS clockc,high);
      switch_context;
    END;
    IF full_diags THEN BEGIN
      Outint(id,2); Outchar('!');                                 3135
    END;
    Passivate;
  END;
END of class clockc;
                                                                  3140
```

```
REF(disk_controllerc) ARRAY dsk(n+1:n+max_disks);

Link CLASS disk_buffer;
BEGIN                      !data structure;
  REF(segmentc) seg;       !space for data;                          3145
  REF(domain_incarnationc) inc;
  !control information;
  BOOLEAN read;            !true when read false when write;
END;
                                                                     3150
Process CLASS disk_controllerc(id,handler_site,d_secretary);
INTEGER id,handler_site;
REF(domain_incarnationc) d_secretary;
BEGIN
  REF(Head) sq,            !for messages sent to kernels;            3155
  freeq,                   !for free buffers;
  xferq,                   !for actual disk reads and writes;
  matchq;                  !for holding reads until claimed;

  REF(diskc) disk;         !associated disk;                         3160
  REF(contentc) message;
  REF(Link) ptr;
  REF(disk_buffer) buf;    !pointer to current buffer;

  PROCEDURE int(m);                                                  3165
  REF(contentc) m;
  INSPECT m
  WHEN disk_rqst DO
  BEGIN
  !required segment has already been read from disk;                 3170
    buf:-matchq.Suc;       !and is waiting in matchq;
    WHILE IF buf==NONE THEN FALSE ELSE buf.seg.key NE key  DO
    buf:-buf.Suc;
    IF buf==NONE THEN error("DISK READS OUT OF SEQUENCE")
    ELSE BEGIN                                                       3175
      buf.seg.dest:=rqstor;
      buf.Into(sq);
      IF THIS disk_controllerc.Idle THEN
      ACTIVATE THIS disk_controllerc DELAY 0;
    END;                                                             3180
    !send segment away;
  END

  WHEN domain_incarnationc DO
  BEGIN                    !request to read from disk;               3185
    buf:-freeq.Suc;
    IF buf=/=NONE THEN BEGIN
      buf.read:=TRUE;
      buf.seg:-processor.params(1);           !set up information;
      buf.inc:-m QUA domain_incarnationc;                            3190
      buf.Into(xferq);    !and put into fifo queue for disk;
      IF disk.Idle THEN disk.transfer;
      IF NOT freeq.Empty THEN signal_free_buffer;
    END ELSE error("DISK MANAGEMENT MESSED UP");
  END                                                                3195

  WHEN segmentc DO
```

```
BEGIN                          !segment to be written to disk;
  buf:-freeq.Suc;
  IF buf=/=NONE THEN BEGIN                                          3200
    buf.read:=FALSE;
    buf.inc:-NONE;
    buf.seg:-m QUA segmentc;
    buf.Into(xferq);
    IF disk.Idle THEN disk.transfer;                                3205
    IF NOT freeq.Empty THEN signal_free_buffer;
  END ELSE error("DISK MANAGEMENT MESSED UP");
END

OTHERWISE error("UNRECOGNISED MESSAGE TO DISK");                    3210

PROCEDURE signal_free_buffer;
BEGIN
!this is only executed when d_secretary is not scheduled;
  d_secretary.dest:=handler_site;                                   3215
  d_secretary.Into(sq);
  IF THIS disk_controllerc.Idle THEN
  ACTIVATE THIS disk_controllerc DELAY 0;
  !equivalent to send_message(handler_site,d_secretary);
END;                                                                3220

PROCEDURE signal_read_complete(buf);
REF(disk_buffer) buf;  !holding read data;
BEGIN
  buf.Into(matchq);     !to await incarnation claiming it;  3225
  buf.inc.dest:=handler_site;
  !where process base is;
  buf.inc.Into(sq);
  IF THIS disk_controllerc.Idle THEN
  ACTIVATE THIS disk_controllerc DELAY 0;                           3230
END of signal_read_complete;

sq:-NEW Head; freeq:-NEW Head; xferq:-NEW Head;
matchq:-NEW Head;
dsk(id):-THIS disk_controllerc;                                     3235
disk:-NEW diskc(THIS disk_controllerc);
FOR w:=1 STEP 1 UNTIL max_disk_bufs DO
NEW disk_buffer.Into(freeq);
ACTIVATE disk DELAY 0;                          !initialize;
                                                                    3240
WHILE running DO
BEGIN
  WHILE sq.Suc =/= NONE DO
  BEGIN
    ptr:-sq.Suc;                                                    3245
    ptr.Out;            !of sq;
    IF ptr IS disk_buffer THEN message:-ptr QUA disk_buffer.seg
    ELSE message:-ptr;
    message.orgn:=id;
    Hold(message.size*mesdelay);                                    3250
    channel_use.add(message.size);
    IF message.size=32 THEN control_count.incr;
    INSPECT k(message.dest) DO int(message);
    IF ptr IS disk_buffer THEN
```

```
    BEGIN                                                              3255
      IF freeq.Empty THEN signal_free_buffer;
      ptr.Into(freeq);
    END;
  END;
  Passivate;            !no more messages to send;                     3260
END;

END of class disk_controllerc;

                                                                       3265
contentc CLASS disk_rqst;
BEGIN
  INTEGER rqstor,key;
END;
                                                                       3270
Process CLASS diskc(controller);
REF(disk_controllerc) controller;
BEGIN
  REF(disk_buffer) buf;
  INTEGER u;              !random number seed;                         3275
  REF(timer) idle_timer;
  REF(counter) transfers,bytes;

  PROCEDURE transfer;
  BEGIN                                                                3280
    !called by controller to initiate transfer;
    buf:-controller.xferq.Suc;
    idle_timer.stop;
    REACTIVATE THIS diskc DELAY
    (Abs(Randint(1,20,u)-Randint(1,20,u))*2000                         3285
    !find track 2msecs intertrack time;
    +Uniform(0,50000,u)  !50 msec rotation;
    +buf.seg.size*2);    !0.5 mbytes/sec transfer rate;
  END;
                                                                       3290
  u:=3031;
  NEW groupheading("DISK PERFORMANCE");
  bytes:-NEW counter("BYTES TRANSFERED");
  transfers:-NEW counter("COMPLETED TRANSFERS");
  idle_timer:-NEW timer("DISK IDLE TIME",NONE);                        3295
  idle_timer.start;
  Passivate;
  WHILE running DO BEGIN
    transfers.incr;
    bytes.add(buf.seg.size);                                           3300

    IF buf.read THEN controller.signal_read_complete(buf)
    ELSE BEGIN             !write so buffer is free;
      buf.seg.Into
      (controller.d_secretary.domain QUA diskhandler.wfreeq);
      !a big fix to keep number of segment objects constant;
      IF controller.freeq.Empty THEN
      controller.signal_free_buffer;
      buf.Into(controller.freeq);
    END;                                                               3310
    IF controller.xferq.Empty THEN
```

```
    BEGIN
      idle_timer.start;
      Passivate;          !no more disk transfers to perform;
    END                                                         3315
    ELSE transfer;        !another segment;

  END;
END of class diskc;
                                                                3320
```

```
PROCEDURE system_initialization;
BEGIN
  INTEGER i;
  statistic_list:-NEW Head;
  grand_t_list:-NEW Head;                                              3325
  !initialize statistic lists;
  total_response:-NEW grand_total("RESPONSE TIMES");
  usage :- NEW grand_total("SERVICE TIMES ");
  NEW groupheading("UTILIZATION OF PROCESSORS");
  FOR i:=1 STEP 1 UNTIL n DO                                           3330
  BEGIN
    NEW groupheading(fillin("SITE",i));
    k(i) :- NEW kernelc(i);
    ACTIVATE k(i);
  END;                                                                 3335
  NEW groupheading("");
  new_incarnations:-NEW counter("CHANGES OF DOMAIN");
  xfered_domains:- NEW counter("TRANSFERED DOMAINS");
  xfered_processors:-NEW counter("TRANSFERED PROCESSORS");
  xfered_locals:-NEW counter("TRANSFERED LOCAL SEGMENTS");       3340
  migrations:-NEW counter("FORCED MIGRATIONS");
  chopcount:-
  NEW counter("PROCESSORS BLOCKED ON ENTRY TO NETWORK");
  spacecount:-
  NEW counter("INCARNATIONS BLOCKED WAITING FOR SPACE");          3345
  NEW groupheading("RESPONSE TIMES");
  short_commands:-NEW counter("COMPLETED SHORT COMMANDS");
  over2:- NEW counter("RESPONSE TIMES OVER 2 SECS");
  over5:-NEW counter("RESPONSE TIMES OVER 5 SECS");
  non_trivial:-                                                        3350
  NEW regression("NON TRIVIAL SERVICE TIMES","RESPONSE TIME");
  NEW groupheading("COMMUNICATIONS SUBSYSTEM");
  channel_use:-NEW counter("BYTES TRANSFERED");
  control_count:-NEW counter("CONTROL MESSAGES (32 BYTES) SENT");
END;                                                                   3355


INTEGER w;                  !work variable;

TEXT PROCEDURE dissect(p);                                             3360
REF(Link) p;
!gives a text description of a class;
IF p IN contentc THEN dissect:-p QUA contentc.dump
ELSE dissect:- Copy(" - LINK -  ");
                                                                       3365
PROCEDURE audit;
BEGIN
!examines various queues to check on operation of system;
  INTEGER i,j,size;
  REF(Link) ptr;                                                       3370
  thinking_consoles;
  FOR i:=1 STEP 1 UNTIL n DO
  INSPECT k(i) DO
  BEGIN
    q_analysis(driverq,fillin("DRIVERQ",i),dissect);                   3375
    q_analysis(spaceq,"SPACEQ",dissect);
    size:=0;
```

```
  Outtext("DOMAINS"); Outimage;
  FOR j:=1 STEP 1 UNTIL fixed_domains DO
  IF dmn_info(j).here OR dmn_info(j).copy THEN                  3380
  INSPECT dmn_info(j) DO
  BEGIN
    size:=size+d.size;
    !keep track of all space actually being used;
    Outint(j,6);                                                3385
    IF here THEN Outtext(" HERE") ELSE Outtext(" COPY");
    IF NOT external_segs.Empty THEN
    BEGIN
      Outchar('(');
      ptr:-external_segs.Suc;                                   3390
      WHILE ptr=/=NONE DO
      BEGIN
        Outint(ptr QUA domain_incarnationc.processor.pid,3);
        ptr:-ptr.Suc;
      END;                                                      3395
      Outchar(')');
    END;
    IF NOT work.Empty THEN
    BEGIN
      Outtext(" PROCESSOR LIST(");                              3400
      ptr:-work.Suc;
      WHILE ptr=/=NONE DO
      BEGIN
        Outint(ptr QUA virtual_processor.pid,3);
        ptr:-ptr.Suc;                                           3405
      END;
      Outchar(')');
    END;
  END  of looking at domains;
  Outimage;                                                     3410
  Outtext("OUTSTANDING MESSAGES ");
  Outint
  ((sq.Cardinal+cq.Cardinal+
  (IF s_channel.Idle THEN 0 ELSE 1)),3);
  Outimage;                                                     3415
  FOR j:=1 STEP 1 UNTIL n DO BEGIN
    Outint(m_use(j),8);
    Outchar(':');
    Outint(qfs(j),2);
  END;                                                          3420
  Outtext("   ACTUAL FREE MEMORY");
  FOR j:=0 STEP 1 UNTIL t_length-1 DO
  IF seg_table(j)=/=NONE THEN size:=size+seg_table(j).size;
  Outint(msize-4000-size,8);
  Outtext("   COPYSPACE"); Outint(copyspace,8);                 3425
  Outimage;
END;
Outimage;
Outtext("NUMBER QUEUED FOR DISK READS");
j:=0;                                                           3430
FOR i:=1 STEP 1 UNTIL max_disks DO
INSPECT k(i*(n//max_disks)).d_secretary.domain
WHEN diskhandler DO
j:=j+rql.Cardinal+rqh.Cardinal;
```

```
  !could be a transfer in progress;                                3435
  Outint(j,3);
  Outimage;
  Eject(Line+6);
END;
                                                                   3440
```

```
statistic_list:-NEW Head;
grand_t_list :- NEW Head;
!for keeping control of statistics;
Eject(Line+4);
Outtext("            SIMULATION OF NETWORK WITH");            3445
Outint(n,IF n<10 THEN 2 ELSE 3); Outtext(" SITES AND WITH");
Outint(max_consoles,3);  Outtext(" CONSOLES");
Outimage; Eject(Line+3);

FOR w:= 1 STEP 1 UNTIL 132 DO Outchar('*'); Outimage;        3450
Outchar('*');
Outtext("        DIRECTLY CONNECTED SITES AND DISK:");
Outtext(" COPYING OF CODE PERFORMED:");
Outtext(" BIAS TOWARDS PROCESSOR UTILIZATION");
Image.Setpos(132); Outchar('*'); Outimage;                   3455

Outtext("* PERIOD OF SIMULATION (SECS) =");
Outfix(sim_time,0,4);
Outtext("   NUMBER OF SYSTEM DOMAINS =");
Outint(fixed_domains,3);                                     3460
Outtext("   NUMBER OF COMPILERS ="); Outint(compl,2);
Image.Setpos(132); Outchar('*'); Outimage;

Outtext("* MEMORY SIZE ="); Outint(msize,7);
Outtext("BYTES   SIZE DIVIDER CONSTANT=");Outint(size_divider,5);
Outtext("   LOAD SHEDDING FACTOR="); Outint(load_shed,2);
Outtext("   MAXIMUM MIGRATIONS="); Outint(max_shifts,2);
Image.Setpos(132); Outchar('*');Outimage;
Outtext("* LARGE DOMAINS NOT ALLOWED TO START WHEN");
Outtext(" GREATEST FREE MEMORY IS LESS THAN ");                3470
Outint(chopsize,8);
Outtext(" OR NUMBER IN SYSTEM IS GREATER THAN ");
Outint(chopfactor,2);
Image.Setpos(132); Outchar('*');Outimage;
Outtext("* CONSOLE CONTROL SITE="); Outint(cnsl_site,2);        3475
Outtext("   NUMBER OF DISKS=");  Outint(max_disks,2);
Outtext("   DISK BUFFERS="); Outint(max_disk_bufs,2);
Outtext("   DISK SITE(S)=");
FOR w:=1 STEP 1 UNTIL max_disks DO
Outint(w*(n//max_disks),3);                                  3480
Image.Setpos(132);Outchar('*'); Outimage;
Outtext("* TIME SLICE ="); Outfix(timeslice*1&-3,0,3);
Outtext("MSECS.   LONG TIME SLICE =");
Outfix(longtimeslice*1&-3,0,4);
Outtext("MSECS.   COMMUNICATION FREQUENCY (MHZ) =");          3485
Outfix(8/mesdelay,2,5);
Outtext("   RANDOM NUMBER SEED ="); Outint(random_seed,5);
Image.Setpos(132); Outchar('*');
Outimage;
FOR w:=1 STEP 1 UNTIL 132 DO Outchar('*'); Outimage;         3490




system_initialization;                                       3495
ipl;
Hold(settle_time*&6);
```

```
    clearstatistics;
    Outtext("START OF RUN "); thinking_consoles;
    IF full_diags THEN audit;                                    3500
    IF running THEN Hold(sim_time*&6);
    Outtext("END OF RUN "); thinking_consoles;
    !a check that system has not seized up;
    IF running THEN BEGIN
      outputstatistics;                                          3505
      Outtext("PERFORMANCE MEASURE");
      Outfix(total_response.total/usage.total,2,8);
      Outimage;
      Outtext("FRACTION USEFUL PROCESSOR UTILIZATION");
      Outfix(usage.total/(n*sim_time),3,6);                      3510
      Outimage;
    END;

  END;                            !of simulation block;
  results.Close;                                                 3515
END;                              !of program;
```

CROSS REFERENCE TABLE

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| a | 1360 | 1363 | 1387 | 1391 | 1806 | 1808 | 1823 | 1824 | 2317D | 2324D |
| | 2534d | 2545 | 2581M | 2582M | 2693 | | | | | |
| a0 | 459D | 470 | 471 | 482 | | | | | | |
| a1 | 459D | 469 | 471 | 481 | | | | | | |
| abs | 3285 | | | | | | | | | |
| action_trans | 1441 | 1451D | 1743 | | | | | | | |
| active_proce | 2891D | 2914 | 2918 | 2924M | 2941M | 2946M | 2952M | 2968M | | |
| add | 327D | 373 | 428D | 790D | 804 | 827 | 3013 | 3251 | 3300 | |
| add_seg | 554 | 824D | 868 | 1370 | 1392 | | | | | |
| audit | 196 | 3366D | 3500 | | | | | | | |
| b | 2534D | 2545 | 2583M | 2584M | | | | | | |
| big_d_size | 1088D | 1106 | 1152 | | | | | | | |
| big_p_size | 1088D | 1121 | 1153 | 1156 | | | | | | |
| blanks | 173 | 184 | 194 | 209 | | | | | | |
| breakoutimag | 107 | 120 | | | | | | | | |
| bring_togeth | 1066 | 1339D | | | | | | | | |
| broadcast | 916D | 985 | 992 | 1764 | | | | | | |
| buf | 3163D | 3171 | 3172M | 3173M | 3174 | 3176 | 3177 | 3186 | 3187 | 3188 |
| | 3189 | 3190 | 3191 | 3199 | 3200 | 3201 | 3202 | 3203 | 3204 | 3222D |
| | 3225 | 3226 | 3228 | 3274D | 3282 | 3288 | 3300 | 3302M | 3304 | 3309 |
| bytes | 3277D | 3293 | 3300 | | | | | | | |
| b_entries | 236D | 237 | 1686 | | | | | | | |
| cancel | 3126 | | | | | | | | | |
| cardinal | 232 | 237M | 1107M | 2855M | 3108 | 3413M | 3434M | | | |
| change_value | 386D | 626 | 665 | 708 | | | | | | |
| channel_busy | 897D | 905 | 929 | 1840 | | | | | | |
| channel_use | 2991D | 3013 | 3251 | 3353 | | | | | | |
| choice | 1060 | 1061 | 1147 | 1161 | 1170 | 1224 | 1998D | | | |
| chopcount | 569D | 2917 | 3342 | | | | | | | |
| chopfactor | 93D | 151 | 152 | 2914 | 2918 | 2946 | 3473 | | | |
| chopsize | 93D | 154 | 982 | 3471 | | | | | | |
| claim | 647D | 689 | 720 | 736 | 1261 | 1758 | 2210 | 3057 | | |
| clear | 291D | 300D | 313D | 358D | 395D | 417D | 451D | 517 | 522 | 2972 |
| clearstatist | 512D | 3498 | | | | | | | | |
| clockc | 589 | 620 | 1676 | 3112D | 3117 | 3122M | 3125 | 3126 | 3131 | |
| close | 3515 | | | | | | | | | |
| cnsl_site | 86D | 130 | 2387 | 3050 | 3055 | 3107 | 3475 | | | |
| coming | 1053 | 1091 | 1252 | 1282 | 1287 | 1401 | 1725 | 1756 | 1768 | 1933D |
| | 1957 | | | | | | | | | |
| command | 2387 | 2886D | 3108 | | | | | | | |
| commandn | 84D | 129 | 2387 | 3051 | | | | | | |
| compiler | 2393 | 2530D | | | | | | | | |
| compl | 60D | 134 | 135 | 2392 | 2906 | 3461 | | | | |
| completed_co | 3040D | 3053 | 3086 | | | | | | | |
| console | 2401 | 2402 | 2937 | 3097D | 3104 | | | | | |
| consolec | 1705 | 2401 | 2964 | 3035D | 3065 | 3097 | | | | |
| contentc | 530D | 546 | 606 | 633 | 859 | 861 | 900 | 919 | 927 | 1831 |
| | 1838 | 1964 | 1973 | 1984 | 2312 | 2321 | 2727 | 2730 | 3000 | 3003 |
| | 3161 | 3166 | 3266 | 3363M | | | | | | |
| contents | 899D | 903 | 904 | 906M | 907 | 911 | 912 | 916D | 923 | 926D |
| | 931 | 932 | 935 | 937 | 938 | | | | | |
| contextdelay | 65D | 110 | 1619 | 2937 | | | | | | |
| controller | 3271D | 3282 | 3302 | 3305 | 3307 | 3308 | 3309 | 3311 | | |
| control_coun | 2992D | 3014 | 3252 | 3354 | | | | | | |
| copy | 115 | 116 | 117 | 118 | 539 | 758 | 1053 | 1175 | 1253 | 1282 |
| | 1472 | 1481 | 1519 | 1881 | 1931D | 1956 | 1978 | 2043 | 2133 | 2418 |
| | 2502 | 2540 | 3364 | 3380 | | | | | | |
| copyspace | 593D | 663 | 707 | 749 | 1473M | 1520M | 3425 | | | |
| cost_formula | 1071D | 1078 | 1152 | 1156 | 1159 | | | | | |
| count | 312D | 314 | 320M | 321 | 325M | 329M | 1281 | 1283 | 1346 | 1349 |
| | 1354m | 1359m | 1373M | 1386M | 1401 | 1785 | 1878M | 1880 | 2023D | |
| counter | 310D | 562 | 2991 | 2992 | 3040 | 3053 | 3277 | 3293 | 3294 | 3337 |
| | 3338 | 3339 | 3340 | 3341 | 3343 | 3345 | 3347 | 3348 | 3349 | 3353 |
| | 3354 | | | | | | | | | |
| cq | 617 | 895D | 906 | 1836 | 1838M | 3413 | | | | |
| create_secre | 2201D | 2873 | 2983 | | | | | | | |
| cu | 579D | 1638 | 1640 | 1642 | 1645 | 1653 | 1660 | 1664 | 1679 | 1680 |
| | 1681 | 1682 | 1683 | 1684 | 1685 | 1687 | 1690M | 1694 | 1695 | 1697 |
| | 2182 | 2198 | | | | | | | | |
| current | 2117 | 2118 | 2256D | 2259 | 2916 | 3122 | | | | |
| c_arrival_li | 1714 | 2962 | 2972 | 2987 | 3031D | | | | | |
| c_secretary | 582D | 1707 | 1711 | 1712 | 2983 | 2984 | 2985 | 3107 | | |
| c_size | 2163D | 2169 | 2516 | 2860 | 2981 | | | | | |
| c_time | 1584 | 1585 | 1681 | 2234D | 3073 | 3080 | 3085 | | | |
| c_wait_list | 2890D | 2915 | 2916 | 2920 | 2945 | 2950 | 2988 | 3108 | | |
| d | 459D | 468 | 469 | 470 | 473 | 485 | 865 | 1075D | 1076 | 1077M |
| | 1078 | 1105 | 1106 | 1145 | 1231 | 1297 | 1313 | 1419 | 1459 | 1465 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1473 | 1486 | 1503 | 1510 | 1520 | 1521 | 1522 | 1929D | 3383 | |
| data | 443D | 3085 | | | | | | | | |
| db_size | 2164D | 2169 | 2517 | 2860 | 2981 | | | | | |
| ddl | 61D | 135 | 136 | 2394 | 2911 | | | | | |
| ddu | 61D | 136 | 137 | 2394 | 2489 | 2908 | | | | |
| deadlock_war | 598D | 726 | 730 | 1898 | 1907 | | | | | |
| default | 550D | 2035 | 2087 | 2140 | 2299 | 2370M | | | | |
| delete_domai | 770 | 1486 | 1510 | 1515D | | | | | | |
| delete_seg | 832D | 1846 | 2089 | 2552 | 2635 | 2644 | | | | |
| desc | 82D | 119 | 2374 | 2573 | 2629 | 2752 | | | | |
| dest | 534D | 899D | 901 | 904M | 3015 | 3017 | 3019 | 3176 | 3215 | 3226 |
| | 3253 | | | | | | | | | |
| did | 1049 | 1087 | 1105 | 1231M | 1237 | 1249 | 1252M | 1253 | 1266 | 1279 |
| | 1285m | 1296 | 1297 | 1313 | 1347 | 1401 | 1402 | 1416 | 1419 | 1441 |
| | 1486 | 1503 | 1506 | 1510 | 1515D | 1518 | 1687 | 1721 | 1723 | 1725M |
| | 1727 | 1730 | 1736 | 1743 | 1753 | 1763 | 1798 | 1861 | 1867 | 1966D |
| | 1973d | 1992d | 2033 | 2045 | 2061D | 2101M | 2134 | 2139 | 2146D | 2154 |
| | 2209 | 2213 | 2261D | 2266M | 2268 | 2269 | 2331D | 2340 | 2372M | 2420 |
| | 2450 | 2468 | 2482 | 2489M | 2504 | 2517 | 2519 | 2526M | 2586 | 2649 |
| | 2657 | 2871 | 2980 | | | | | | | |
| didn | 2028D | 2033 | | | | | | | | |
| disk | 2661D | 2727 | 2730 | 2777 | 2787 | 2868 | 2877 | 3160D | 3192M | 3205M |
| | 3236 | 3239 | | | | | | | | |
| diskc | 3160 | 3236 | 3271D | 3284 | | | | | | |
| diskhandler | 2389 | 2661D | 3305 | 3433 | | | | | | |
| diskhandlern | 2434 | 2561 | 2574 | 2604 | 2639 | 2881D | 2884 | | | |
| diskl | 61D | 139 | 2389 | 2884 | | | | | | |
| disk_buffer | 3143D | 3163 | 3223 | 3238 | 3247M | 3254 | 3274 | | | |
| disk_control | 2877 | 3141 | 3151D | 3178 | 3179 | 3217 | 3218 | 3229 | 3230 | 3235 |
| | 3236 | 3272 | | | | | | | | |
| disk_read | 1374 | 1376 | 1377 | 2290D | 2307 | | | | | |
| disk_rqst | 2290 | 2307 | 3168 | 3266D | | | | | | |
| disk_write_r | 2684D | 2693 | 2695 | 2867 | 2868 | | | | | |
| dissect | 1629 | 3360D | 3363 | 3364 | 3375 | 3376 | | | | |
| dmn_info | 630 | 758 | 759 | 760 | 761 | 763 | 764 | 865 | 1031D | 1049 |
| | 1087 | 1105 | 1231M | 1237 | 1249 | 1252M | 1253 | 1266 | 1279 | 1296 |
| | 1297 | 1313 | 1347 | 1401 | 1402 | 1416 | 1419 | 1441 | 1503 | 1506 |
| | 1518 | 1687 | 1721 | 1723 | 1725M | 1727 | 1730 | 1743 | 1753 | 1798 |
| | 1871 | 1880 | 1881 | 3380M | 3381 | | | | | |
| dmn_infoc | 630 | 1031 | 1452 | 1925D | | | | | | |
| dmn_rqst | 1034D | 1041 | 1047M | 1049 | 1054 | 1055 | 1058 | 1060 | 1061M | 1063 |
| | 1066m | 1082d | 1086 | 1095 | 1173 | 1176 | 1199 | 1211D | 1214 | 1225 |
| | 1292 | 1339d | 1343 | 1402 | 1403 | | | | | |
| dmn_transfer | 1719 | 1964D | 2287 | 2304 | | | | | | |
| domain | 1419 | 1421 | 1423 | 1494D | 1500 | 1501 | 1503 | 1506 | 1571 | 1598 |
| | 1599 | 1600 | 1601 | 1602 | 1680 | 1994D | 2214 | 3107 | 3305 | 3432 |
| domainc | 865 | 1495 | 1750 | 1844M | 1929 | 1994 | 2061D | 2142 | 2161 | 2214 |
| | 2407 | 2530 | 2589 | | | | | | | |
| domain_copy | 1494D | 1844 | | | | | | | | |
| domain_incar | 579 | 678 | 703 | 872 | 873 | 932 | 1035 | 1083 | 1212 | 1307 |
| | 1340 | 1410 | 1437 | 1544 | 1633 | 1755 | 1858 | 1984D | 2073 | 2080 |
| | 2096 | 2111 | 2112 | 2124 | 2191 | 2203 | 2250 | 2256 | 2261 | 2275 |
| | 2301 | 2414 | 2429 | 2442 | 2463 | 2474 | 2498 | 2513 | 2536 | 2549 |
| | 2571 | 2595 | 2609 | 2618 | 2672 | 2735 | 2748 | 2792 | 2799 | 2895 |
| | 2901 | 2935 | 2958 | 3146 | 3153 | 3184 | 3190 | 3393 | | |
| do_domain_ca | 1055 | 1082D | | | | | | | | |
| dp | 2154 | 2158D | 2269 | | | | | | | |
| dr | 1755D | 1772 | 1773 | 1779 | 1780 | 1784 | 1785 | 1787 | 1788 | 1790 |
| draw | 2431 | 2448 | 2627 | | | | | | | |
| driverq | 576D | 610 | 723 | 870 | 874 | 878 | 882 | 912 | 1309 | 1323 |
| | 1426 | 1429 | 1623 | 1686 | 1690 | 1695 | 1711 | 1773 | 2106 | 2194 |
| | 2716 | 2920 | 2950 | 2966 | 3022 | 3065 | 3131 | 3375 | | |
| dsk | 3019 | 3141D | 3235 | | | | | | | |
| dump | 531D | 538D | 539 | 693 | 1977D | 1978 | 2039D | 2046 | 2130D | 2135 |
| | 2355d | 2356 | 3363 | | | | | | | |
| dumpv | 2042D | 2043 | 2044 | 2045 | 2046 | | | | | |
| d_cost | 1089D | 1152 | 1161M | | | | | | | |
| d_id | 2337D | 2340 | | | | | | | | |
| d_loc | 1054 | 1176 | 1347 | 1727 | 1731 | 1766 | 1798 | 1940D | | |
| d_loc_update | 1736 | 1763 | 1796 | 1952 | 1973D | | | | | |
| d_secretary | 581D | 2873 | 2874 | 2875 | 2877 | 3151D | 3215 | 3216 | 3305 | 3432 |
| d_site | 1104 | 2012D | | | | | | | | |
| d_size | 1105 | 1107 | 1108 | 1240 | 1254 | 1270 | 1318 | 2014D | | |
| d_transfer | 1285 | 1286 | 1347 | 2287D | 2304 | | | | | |
| eject | 505 | 3438 | 3444 | 3448 | | | | | | |
| empty | 224 | 242 | 759 | 760 | 1197 | 1467M | 1484M | 1508 | 1509 | 1770 |
| | 1836m | 1838 | 2697M | 2700 | 2707 | 2725M | 2726 | 2729 | 2754 | 2816M |
| | 2830m | 2832m | 2841 | 2846 | 2915 | 2945 | 3193 | 3206 | 3256 | 3307 |
| | 3311 | 3387 | 3398 | | | | | | | |
| entry | 249D | 252 | | | | | | | | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| erlang | 2556 | | | | | | | | | |
| error | 187D | 254 | 801 | 818 | 820 | 1422 | 1572 | 1596 | 1746 | 1761 |
| | 1875 | 1917 | 2247 | 2273 | 2280 | 3174 | 3194 | 3207 | 3210 | |
| esac | 1598 | 1599 | 1600 | 1601 | 1603D | | | | | |
| evtime | 3125 | | | | | | | | | |
| examine_choi | 1063 | 1211D | | | | | | | | |
| execute | 1543D | 1645 | | | | | | | | |
| external_seg | 760 | 1107 | 1197 | 1402 | 1467 | 1484 | 1509 | 1779 | 1871 | 1946D |
| | 1955 | 3387 | 3390 | | | | | | | |
| extra_space | 684 | 688M | 720 | 735M | 1254M | 1256M | 1260 | 1261 | 2021D | |
| e_stack | 1861 | 2250D | 2259 | 2266 | 2268 | 2269 | 2271 | 2279 | 2301 | 2966 |
| f | 1755D | 1784 | 1790 | 1858D | 1871 | 1872M | 1873M | 1875 | 1878M | 1880 |
| | 1883 | 1884 | | | | | | | | |
| fetch_c | 2076 | 2213 | 2261D | 2271 | 3051 | | | | | |
| fillin | 202D | 213 | 728 | 801 | 2273 | 2280 | 2356 | 2598 | 2739 | 3046 |
| | 3332 | 3375 | | | | | | | | |
| finisht | 1570D | 1577 | 1585 | 1586 | | | | | | |
| first | 221D | 225M | 715 | 725 | 1623 | | | | | |
| fixed_domain | 52D | 140 | 553 | 629 | 757 | 864 | 1031 | 2158 | 2298 | 3379 |
| | 3460 | | | | | | | | | |
| format | 2149D | 2152D | 2269 | 2487D | 2524D | 2654D | | | | |
| formatc | 2146D | 2154 | 2158 | 2485 | 2522 | 2652 | 2871 | 2980 | | |
| found | 1308D | 1321 | 1329 | | | | | | | |
| freeq | 3156D | 3186 | 3193 | 3199 | 3206 | 3233 | 3238 | 3256 | 3257 | 3307 |
| | 3309 | | | | | | | | | |
| full_diags | 73D | 168 | 196 | 692 | 2416 | 2451 | 2465 | 2500 | 2538 | 2597 |
| | 2737 | 3067 | 3088 | 3134 | 3500 | | | | | |
| going | 761 | 1472 | 1485 | 1507 | 1937D | | | | | |
| gone | 1219D | 1226 | 1245 | 1279 | | | | | | |
| grand_total | 334 | 414D | 432 | 560 | 3327 | 3328 | | | | |
| grand_t_list | 295D | 432 | 506 | 520 | 3325 | 3442 | | | | |
| groupheading | 297D | 3046 | 3292 | 3329 | 3332 | 3336 | 3346 | 3352 | | |
| h | 1451D | 1458 | 1459M | 1460 | 1463M | 1465 | 1467M | 1472M | 1473 | 1481 |
| | 1484m | 1485 | 1486 | 1487 | | | | | | |
| halt | 1653 | 3119D | | | | | | | | |
| handler_site | 3151D | 3215 | 3226 | | | | | | | |
| hash | 785D | 787 | 794 | 811 | | | | | | |
| head | 219 | 246 | 294 | 295 | 617 | 618 | 895 | 1943 | 1955M | 2176 |
| | 2220 | 2679 | 2681 | 2683 | 2861 | 2862 | 2863 | 2864 | 2865 | 2866 |
| | 2890 | 2987 | 2988 | 3031 | 3155 | 3233M | 3234 | 3324 | 3325 | 3441 |
| | 3442 | | | | | | | | | |
| heading | 257D | 267 | 289D | 304 | 319 | 369 | 404 | 423 | 462 | 476 |
| heading2 | 437D | 463 | 477 | | | | | | | |
| here | 1053 | 1231 | 1237 | 1252 | 1266 | 1282 | 1296 | 1458 | 1463 | 1725 |
| | 1769 | 1880 | 1935D | 1958 | 3380 | 3386 | | | | |
| high | 80D | 114 | 118 | 219 | 223 | 232 | 242 | 243 | 246 | 251 |
| | 271 | 723 | 870 | 878 | 882 | 912 | 1323 | 1695 | 1773 | 2106 |
| | 2194 | 2920 | 2950 | 2966 | 3022 | 3065 | 3131 | | | |
| hk | 793D | 794 | 795 | 798 | 799M | 801 | 803 | 804 | 810D | 811 |
| | 812 | 814 | 815M | 817 | 820 | 821 | | | | |
| hold | 652 | 691 | 705 | 755 | 1045 | 1580 | 1619 | 3063 | 3250 | 3497 |
| | 3501 | | | | | | | | | |
| i | 202D | 211 | 220D | 223 | 224M | 225M | 232M | 241 | 242M | 243 |
| | 246m | 264d | 271 | 273 | 274 | 753D | 757 | 758 | 759 | 760 |
| | 761 | 763 | 764 | 765 | 793D | 798 | 799 | 801 | 810D | 814 |
| | 815 | 817 | 836D | 837 | 838 | 841 | 921D | 922 | 923M | 1009D |
| | 1012 | 1013 | 1015 | 1017 | 1018 | 1090D | 1115 | 1116 | 1117 | 1118M |
| | 1119 | 1128 | 1129 | 1130 | 1131 | 1134 | 1136 | 1220D | 1345D | 1356 |
| | 1357 | 1360m | 1363 | 1368 | 1370M | 1372 | 1376 | 1377 | 1382 | 1383 |
| | 1385 | 1387m | 1391 | 1392M | 1804D | 1806M | 1808 | 1818D | 1822 | 1823 |
| | 1824 | 2026d | 2034 | 2035M | 2051 | 2052 | 2070D | 2082D | 2083 | 2084 |
| | 2087 | 2089 | 2100 | 2101 | 3101D | 3103 | 3104 | 3323D | 3330 | 3332 |
| | 3333m | 3334 | 3369D | 3372 | 3373 | 3375 | 3431 | 3432 | | |
| id | 573D | 616 | 620 | 625 | 663 | 670 | 694 | 707 | 711 | 728 |
| | 801 | 829 | 839 | 879M | 901 | 903 | 911 | 923 | 931 | 932M |
| | 935 | 984 | 992 | 1060 | 1104 | 1147 | 1152 | 1153 | 1154 | 1155 |
| | 1158 | 1161 | 1170 | 1197 | 1198 | 1240 | 1257 | 1258 | 1270 | 1271 |
| | 1286 | 1319 | 1352 | 1355 | 1374 | 1385 | 1395 | 1397 | 1413 | 1628 |
| | 1729 | 1752 | 1763 | 1766 | 1897 | 1903 | 2951M | 2969M | 3112D | 3130 |
| | 3135 | 3151d | 3235 | 3249 | | | | | | |
| idle | 1530 | 3104 | 3122 | 3178 | 3192 | 3205 | 3217 | 3229 | 3414 | |
| idle_timer | 603D | 621 | 1910 | 1912 | 3276D | 3283 | 3295 | 3296 | 3313 | |
| iflag | 586D | 614 | 1535 | 1586 | 1590 | 1617 | 1620 | 1660 | | |
| image | 3455 | 3462 | 3468 | 3474 | 3481 | 3488 | | | | |
| inc | 677D | 684M | 688M | 693 | 1409D | 1413 | 1414 | 1416M | 1419M | 1421 |
| | 1423 | 1425 | 1426 | 1429M | 1436D | 1440 | 1441 | 1443 | 1445 | 1447 |
| | 2191d | 2192 | 2193 | 2194M | 3146D | 3190 | 3202 | 3226 | 3228 | |
| incore | 82D | 119 | 553 | 828 | 1116 | 1129 | 1357 | 1383 | 2084 | 2140 |
| | 2299 | 2374 | 2551 | 2627 | 2634 | 2643 | | | | |
| incr | 324D | 696 | 1228 | 1499 | 1809 | 1820 | 1825 | 2108 | 2917 | 3014 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3082 | 3083 | 3084 | 3086 | 3252 | 3299 | | | | |
| inint | 108 | 121 | | | | | | | | |
| initializati | 608D | 1612 | | | | | | | | |
| initial_time | 383D | 399 | 405 | 406 | | | | | | |
| initiate | 938 | 3002D | | | | | | | | |
| initiate_rea | 2722D | 2725 | 2819 | 2834 | 2839 | | | | | |
| int | 858D | 2142 | 3017 | 3019 | 3165D | 3253 | | | | |
| interdmn_jum | 2076 | 2091 | 2095D | | | | | | | |
| interdomain_ | 2072D | 2434 | 2456 | 2561 | 2574 | 2604 | 2639 | 2909 | | |
| interval | 3115D | 3117 | | | | | | | | |
| into | 252 | 292 | 432 | 906 | 1095 | 1249 | 1402 | 1416 | 1714 | 2182 |
| | 2703 | 2709 | 2757 | 2766 | 2767 | 2782 | 2870 | 2916 | 3177 | 3191 |
| | 3204 | 3216 | 3225 | 3228 | 3238 | 3257 | 3304 | 3309 | | |
| ipl | 2381D | 3496 | | | | | | | | |
| ipld | 55D | 132 | 135 | 2392M | 2906M | | | | | |
| items | 257D | 278 | | | | | | | | |
| i_chopf | 95D | 152 | 1270 | 1318 | | | | | | |
| j | 753D | 765 | 769 | 770 | 960D | 976 | 978M | 979M | 1009D | 1011 |
| | 1017 | 1021 | 2382D | 3101D | 3104M | 3105 | 3369D | 3379 | 3380M | 3381 |
| | 3385 | 3416 | 3417 | 3419 | 3422 | 3423M | 3430 | 3434M | 3436 | |
| k | 553 | 559D | 1009D | 1010 | 1011M | 1013 | 2089 | 2103 | 2142 | 2182 |
| | 2189 | 2198 | 2207 | 2374 | 2552 | 2634 | 2644 | 2694 | 2716 | 2727 |
| | 2730 | 2787 | 2812 | 2872 | 2914 | 2918 | 2920 | 2925M | 2942 | 2943M |
| | 2947 | 2949 | 2965 | 2982 | 3017 | 3021 | 3055 | 3107 | 3130 | 3253 |
| | 3333 | 3334 | 3373 | 3432 | | | | | | |
| keeping | 341D | 344 | 346 | 352 | 354 | 361 | 370 | 376 | | |
| kernelc | 559 | 573D | 1530 | 1537 | 3333 | | | | | |
| key | 549D | 553 | 785D | 787M | 790D | 794 | 803 | 807D | 811 | 812 |
| | 815 | 827 | 837 | 864 | 865 | 1360 | 1376M | 1387 | 2139 | 2298 |
| | 2305 | 2341 | 2350 | 2369M | 2375 | 2693 | 3172M | 3268D | | |
| l | 2152D | 2487D | 2489 | 2524D | 2526 | 2654D | 2657M | | | |
| lasttime | 763 | 764 | 1487 | 1951D | | | | | | |
| length | 209 | 211 | | | | | | | | |
| level | 386D | 390 | | | | | | | | |
| lfrom | 2360D | 2365 | 2378 | | | | | | | |
| line | 505 | 3438 | 3444 | 3448 | | | | | | |
| linear | 2545 | 2601 | | | | | | | | |
| link | 221 | 250 | 264 | 289 | 530 | 1610 | 2685 | 3143 | 3162 | 3361 |
| | 3370 | | | | | | | | | |
| linkage | 2961 | | | | | | | | | |
| load_shed | 89D | 149 | 1240 | 1270 | 3466 | | | | | |
| locals | 1129 | 1130 | 1131 | 1134 | 1136 | 1383 | 1385 | 1387 | 1392M | 1996D |
| | 2035m | 2052 | 2084 | 2087 | 2089 | 2269 | 2476 | 2611 | | |
| local_data | 2007D | 2545 | 2554 | 2559M | 2601 | 2619 | 2624M | | | |
| local_seg | 1865 | 1867 | 1996 | 2052 | 2152 | 2236 | 2303 | 2328D | 2361 | 2488 |
| | 2525 | 2655 | 2702 | 2756 | 2870 | | | | | |
| longtimeslic | 68D | 112 | 1681 | 1682 | 3080 | 3484 | | | | |
| low | 80D | 114 | 115 | 219 | 224 | 225 | 232 | 237 | 241 | 246 |
| | 251 | 271 | 685 | 1309 | 1683 | 2705 | 2766 | 2782 | | |
| lru | 754D | 756 | 763 | 764 | | | | | | |
| lto | 2360D | 2366 | 2367 | 2368 | 2369 | 2370 | 2371 | 2372 | 2375 | |
| l_cost | 1089D | 1159 | 1161 | 1162 | | | | | | |
| l_seg_list | 1387 | 1391 | 1397 | 1398 | 2289D | 2306 | | | | |
| l_seg_listc | 1801 | 2289 | 2306 | 2321D | 2684 | 2867 | | | | |
| l_site | 1100 | 1130 | 1138 | 1154 | 1157 | 1158M | 1159 | 1163 | 1198M | 1258 |
| | 1395m | 1398 | 2012D | | | | | | | |
| l_size | 1102 | 1131M | 1140 | 1154 | 1157 | 1159 | 1258 | 2014D | | |
| m | 858D | 861 | 863 | 864 | 865M | 868 | 870 | 872 | 873 | 874 |
| | 878 | 882 | 886M | 887M | 889M | 1831D | 1832M | 1844M | 1846M | 3000D |
| | 3005 | 3007 | 3013 | 3014 | 3015 | 3017M | 3019M | 3165D | 3167 | 3190 |
| | 3203 | | | | | | | | | |
| main | 199 | | | | | | | | | |
| make_disk_re | 2346D | 2775 | | | | | | | | |
| make_space | 656 | 748D | 771 | | | | | | | |
| make_workspa | 2337D | 2489 | 2526 | 2657 | | | | | | |
| maskf | 584D | 615 | 1533 | 1536 | 1575 | 1587 | | | | |
| master | 333D | 373M | | | | | | | | |
| matchq | 3158D | 3171 | 3225 | 3234 | | | | | | |
| max | 384D | 391M | 398 | 408 | | | | | | |
| max_consoles | 53D | 121 | 127 | 154 | 158 | 160 | 2209 | 2399 | 2517 | 2981 |
| | 3097 | 3103 | 3447 | | | | | | | |
| max_disks | 56D | 131 | 140 | 2388 | 2389 | 2884 | 3141 | 3431 | 3432 | 3476 |
| | 3479 | 3480 | | | | | | | | |
| max_disk_buf | 57D | 142 | 143 | 3237 | 3477 | | | | | |
| max_local_se | 75D | 122 | 1128 | 1382 | 1806 | 1996 | 2034 | 2051 | 2083 | 2324 |
| max_param_se | 77D | 123 | 1115 | 1356 | 1368 | 1822 | 2100 | 2236 | 2302 | 2317 |
| max_shifts | 91D | 150 | 1236 | 1267 | 1314 | 3467 | | | | |
| max_writes_p | 59D | 143 | 2869 | | | | | | | |
| medium | 80D | 114 | 116 | 237 | 685 | 1711 | 2294 | | | |
| mem | 537D | 887 | 931 | 958D | 965 | 966 | 969 | 970 | 972 | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| memory_use | 605D | 622 | 626 | 665 | 708 | | | | | |
| mem_trace | 73D | 169 | 668 | 709 | | | | | | |
| mesdelay | 69D | 109 | 3007 | 3250 | 3486 | | | | | |
| message | 3002D | 3005 | 3161D | 3247 | 3248 | 3249 | 3250 | 3251 | 3252 | 3253M |
| mfree | 591D | 623 | 626 | 627 | 652 | 656 | 659 | 661M | 663 | 665 |
| | 669 | 687 | 688 | 689 | 706M | 707 | 708 | 710 | 733 | 735 |
| | 736 | | | | | | | | | |
| migrations | 566D | 1228 | 3341 | | | | | | | |
| mntrl | 61D | 137 | 138 | 2396 | | | | | | |
| mntru | 61D | 138 | 139 | 140 | 2396 | 2450 | 2517 | 2908 | | |
| mod | 787 | 799 | 815 | 2393 | 2395 | 2397 | 2884 | | | |
| monitor | 80D | 114 | 117 | 874 | 1426 | 2194 | 2716 | | | |
| monitorc | 1106 | 1423 | 1465 | 1500 | 1680 | 2161D | 2172 | 2493 | | |
| more | 704D | 716 | 717 | 720 | 721 | | | | | |
| move | 2360D | 2476 | 2611 | 2701 | 2756 | | | | | |
| msize | 49D | 125 | 623 | 626 | 665 | 669 | 708 | 710 | 1078 | 3424 |
| | 3464 | | | | | | | | | |
| my_q | 2176D | 2182 | 2192 | 2220 | 2707 | 2709 | 2712 | 2846 | 2855 | |
| m_max | 625 | 956D | 963 | 965M | 969M | 975 | 978M | 982 | 1007 | |
| m_min | 625 | 956D | 963 | 966M | 970M | 975 | 979M | 982 | | |
| m_use | 627 | 931 | 952D | 965 | 966 | 969 | 970 | 972 | 978M | 979M |
| | 982m | 1007 | 1015 | 1018 | 1078M | 3417 | | | | |
| n | 46D | 108 | 127M | 131 | 143 | 150 | 151 | 440D | 446M | 453 |
| | 464 | 468 | 469 | 471 | 473M | 475 | 476 | 477 | 485M | 559 |
| | 606 | 627 | 632 | 886 | 922 | 952 | 961 | 976 | 1012 | 1735 |
| | 1739 | 1762 | 1952 | 2389M | 2393 | 2395 | 2397 | 3015 | 3141M | 3330 |
| | 3372 | 3416 | 3432 | 3446M | 3480 | 3510 | | | | |
| negexp | 2425 | 2509 | 2620 | 3044 | | | | | | |
| newsize | 2337D | 2343 | 2346D | 2352 | | | | | | |
| new_did | 1856D | 1861 | 1867 | 1871 | 1880 | 1881 | 2072D | 2076 | 2446D | 2450 |
| | 2453 | 2456 | 2903D | 2905 | 2909 | 2911 | | | | |
| new_incarnat | 565D | 2108 | 3337 | | | | | | | |
| new_site | 1740 | 1798 | 1973D | | | | | | | |
| next_inc | 2672D | 2700 | 2701 | 2705 | 2709 | 2711 | 2712 | 2715 | 2716 | 2779 |
| | 2782 | 2787 | | | | | | | | |
| next_site | 1172 | 1173 | 1195 | 1199 | 1459 | 1460 | 1463 | 1687 | 1721 | 1723 |
| | 1731 | 1941d | 1959 | | | | | | | |
| next_step | 1595M | 1597 | 2004D | 2032 | 2424 | 2435 | 2437 | 2458 | 2477 | 2508 |
| | 2544 | 2562 | 2575 | 2605 | 2614 | 2744 | 2769 | 2874 | 2897 | 2931 |
| | 2940 | 2984 | | | | | | | | |
| nn | 2382D | 2388 | 2389M | 2392 | 2393M | 2394 | 2395M | 2396 | 2397M | 2399 |
| | 2401m | 2402 | | | | | | | | |
| non_trivial | 570D | 3085 | 3350 | | | | | | | |
| null | 82D | 839 | 2035 | 2087 | 2378 | | | | | |
| number | 327D | 329 | | | | | | | | |
| ondisk | 82D | 119 | 1118 | 1372 | 2351 | | | | | |
| open | 173 | | | | | | | | | |
| optimum_site | 998D | 1021 | 1240 | 1269 | 1317 | | | | | |
| opt_site | 1220D | 1224 | 1225 | 1239 | 1241 | 1269 | 1271M | 1316 | 1319M | |
| orgn | 534D | 886M | 887 | 889 | 903 | 911 | 2994D | 3021 | 3249 | |
| out | 937 | 1273 | 1440 | 1443 | 1672 | 1685 | 1787 | 1883 | 2117 | 2198 |
| | 2211 | 2696 | 3246 | | | | | | | |
| outchar | 670 | 711 | 1628 | 3135 | 3389 | 3396 | 3407 | 3418 | 3450 | 3451 |
| | 3455 | 3462 | 3468 | 3474 | 3481 | 3488 | 3490 | | | |
| outfix | 184 | 371 | 405 | 424 | 476 | 477 | 479 | 481 | 482 | 485 |
| | 487 | 3458 | 3482 | 3484 | 3486 | 3507 | 3510 | | | |
| outimage | 191 | 195 | 265 | 269 | 273 | 282 | 303 | 305 | 425 | 460 |
| | 463 | 475 | 479 | 489 | 504 | 694 | 729 | 1906 | 2468 | 2741 |
| | 2825 | 3068 | 3090 | 3109 | 3378 | 3410 | 3415 | 3426 | 3428 | 3437 |
| | 3448 | 3450 | 3455 | 3462 | 3468 | 3474 | 3481 | 3489 | 3490 | 3508 |
| | 3511 | | | | | | | | | |
| outint | 268 | 320 | 408 | 475 | 669 | 670 | 694 | 710 | 711 | 1628 |
| | 1903 | 1905 | 2452 | 2453 | 2466 | 2468 | 3069 | 3089 | 3105 | 3107 |
| | 3135 | 3385 | 3393 | 3404 | 3412 | 3417 | 3419 | 3424 | 3425 | 3436 |
| | 3446 | 3447 | 3460 | 3461 | 3464 | 3465 | 3466 | 3467 | 3471 | 3473 |
| | 3475 | 3476 | 3477 | 3480 | 3487 | | | | | |
| outputstatis | 495D | 3505 | | | | | | | | |
| outtext | 107 | 120 | 184 | 191 | 193 | 194M | 267M | 273 | 278 | 304 |
| | 318 | 319 | 369M | 404M | 408 | 422 | 423 | 461 | 462M | 463 |
| | 475 | 476m | 477M | 478 | 480 | 482 | 483 | 486 | 488 | 693M |
| | 728 | 1628 | 1629 | 1902 | 1904 | 2422 | 2452 | 2467 | 2506 | 2542 |
| | 2598 | 2739 | 2825 | 3069 | 3089 | 3102 | 3106 | 3378 | 3386M | 3400 |
| | 3411 | 3421 | 3425 | 3429 | 3445 | 3446 | 3447 | 3452 | 3453 | 3454 |
| | 3457 | 3459 | 3461 | 3464 | 3465 | 3466 | 3467 | 3469 | 3470 | 3472 |
| | 3475 | 3476 | 3477 | 3478 | 3482 | 3483 | 3485 | 3487 | 3499 | 3502 |
| | 3506 | 3509 | | | | | | | | |
| over2 | 567D | 3083 | 3348 | | | | | | | |
| over5 | 567D | 3084 | 3349 | | | | | | | |
| overload | 683 | 945D | 984 | 988 | 992 | 994 | 1053 | 1237 | 1266 | 2676D |
| | 2707 | 2711 | 2812M | 2814 | 2914 | 2918 | 2947 | | | |

| Identifier | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| p | 1857D | 1860 | 1861M | 1865 | 1872 | 2445D | 2447 | 2448 | 2449 | 2450 |
| | 2452 | 3360d | 3363M | | | | | | | |
| params | 1116 | 1117 | 1118M | 1119 | 1357 | 1360 | 1370M | 1372 | 1376 | 1377 |
| | 2101 | 2236d | 2303 | 2476 | 2551 | 2552 | 2573 | 2611 | 2627 | 2629 |
| | 2634 | 2635 | 2643 | 2644 | 2701 | 2752 | 2756 | 2775 | 2777 | 3189 |
| passivate | 1911 | 3027 | 3076 | 3137 | 3260 | 3297 | 3314 | | | |
| pid | 2044 | 2227D | 2273 | 2280 | 2298 | 2333 | 2356 | 2419 | 2434 | 2452 |
| | 2466 | 2503 | 2541 | 2561 | 2574 | 2599 | 2604 | 2639 | 2740 | 2881D |
| | 2884 | 2937 | 3393 | 3404 | | | | | | |
| pr | 1865 | 2206D | 2209 | 2210 | 2211 | 2213 | 2328D | 2333M | 2356 | 2366M |
| | 2657 | 2671d | 2889D | 2964M | 2966M | 3039D | 3050 | 3051 | 3057 | 3073 |
| | 3080 | 3085 | | | | | | | | |
| pred | 1309 | 1329 | | | | | | | | |
| prev | 1707 | | | | | | | | | |
| print | 291D | 301D | 316D | 366D | 402D | 420D | 457D | 501 | 508 | |
| printfile | 105 | 172 | | | | | | | | |
| priority | 249D | 251M | 252 | 1429 | 1683 | 1690 | 2232D | 2294 | 2705 | 2766 |
| | 2782 | | | | | | | | | |
| priority_tex | 81D | 115 | 116 | 117 | 118 | 273 | | | | |
| process | 573 | 2994 | 3035 | 3112 | 3151 | 3271 | | | | |
| processor | 1112 | 1249 | 1273 | 1285 | 1286 | 1344 | 1416 | 1425 | 1429 | 1440 |
| | 1578 | 1583 | 1584 | 1585 | 1642 | 1653 | 1681 | 1682 | 1683 | 1684 |
| | 1690 | 1697 | 1872 | 1984D | 2044 | 2052 | 2076 | 2091 | 2101 | 2117 |
| | 2118 | 2126 | 2419 | 2425 | 2431 | 2434 | 2447 | 2466 | 2476 | 2503 |
| | 2509 | 2541 | 2545 | 2551 | 2552 | 2556 | 2561 | 2573 | 2574 | 2599 |
| | 2601 | 2602 | 2604 | 2611 | 2620 | 2625 | 2639 | 2643 | 2644 | 2701 |
| | 2705 | 2740 | 2752 | 2756 | 2766 | 2775 | 2776 | 2777 | 2782 | 2904 |
| | 2906 | 2908 | 2916 | 2937 | 3189 | 3393 | | | | |
| ptime | 182D | 194 | 266 | 693 | 727 | 1629 | 1901 | 2422 | 2452 | 2466 |
| | 2506 | 2542 | 2598 | 2738 | 2825 | 3069 | 3089 | | | |
| ptr | 264D | 274 | 276 | 278 | 280M | 497D | 498 | 499 | 501 | 502M |
| | 506 | 507 | 508 | 509M | 514D | 515 | 516 | 517 | 518M | 520 |
| | 521 | 522 | 523M | 1307D | 1309 | 1310 | 1313 | 1314 | 1318M | 1322 |
| | 1323 | 1329m | 1610D | 1623 | 1626 | 1629 | 1633 | 1638 | 1672M | 1673 |
| | 1676 | 1714 | 1723 | 1727 | 1832 | 1859 | 1860 | 1865 | 1867 | 2961D |
| | 2962 | 2963m | 2964 | 3162D | 3245 | 3246 | 3247M | 3248 | 3254 | 3257 |
| | 3370d | 3390 | 3391 | 3393 | 3394M | 3401 | 3402 | 3404 | 3405M | |
| putative_ret | 2111D | 2118 | 2779 | | | | | | | |
| putint | 211 | 2044 | 2045 | 2134 | 2419 | 2420 | 2421 | 2503 | 2504 | 2505 |
| | 2541 | | | | | | | | | |
| put_in_ready | 1403 | 1409D | 1788 | 1884 | | | | | | |
| p_cost | 1089D | 1156 | 1161 | 1162 | | | | | | |
| p_key | 1819 | 2312D | | | | | | | | |
| p_seg_list | 1355 | 1360 | 1363 | 1365 | 2288D | 2305 | | | | |
| p_seg_listc | 1814 | 2288 | 2305 | 2312D | | | | | | |
| p_site | 1120 | 1153 | 1155 | 1156 | 1157 | 1158 | 1162 | 1197 | 1257 | 2012D |
| p_size | 1114 | 1117M | 1121 | 1124 | 1257 | 2014D | | | | |
| q | 219D | 224 | 225 | 232 | 237M | 242 | 246 | 252 | 274 | 1309 |
| q4space | 677D | 1292 | | | | | | | | |
| qempty | 239D | 243 | 270 | | | | | | | |
| qf | 537D | 889 | 935 | | | | | | | |
| qfs | 879M | 889 | 932M | 935 | 952D | 1013 | 1076 | 1240 | 1270 | 1897 |
| | 2925m | 2943m | 2951M | 2969M | 3419 | | | | | |
| qf_max | 998D | 1011 | | | | | | | | |
| qf_min | 998D | 1010 | | | | | | | | |
| qhead | 249D | 252 | 257D | 268 | 270 | 274 | | | | |
| qheadc | 217D | 250 | 259 | 576 | 610 | 611 | | | | |
| queue | 249D | 684 | 723 | 870 | 874 | 878 | 882 | 912 | 1323 | 1426 |
| | 1429 | 1690 | 1695 | 1711 | 1773 | 2106 | 2193 | 2716 | 2920 | 2950 |
| | 2966 | 3022 | 3065 | 3131 | | | | | | |
| q_analysis | 257D | 3375 | 3376 | | | | | | | |
| q_trace | 73D | 170 | 1626 | | | | | | | |
| r | 2903D | 2904 | 2906 | 2907 | | | | | | |
| r2 | 459D | 473 | 487 | | | | | | | |
| randint | 2401 | 2450 | 2602 | 2657 | 2906 | 2908 | 3285M | | | |
| random_seed | 102D | 156 | 2383 | 3487 | | | | | | |
| read | 3148D | 3188 | 3201 | 3302 | | | | | | |
| read_next | 2675D | 2809 | 2810 | 2812 | 2816 | 2832 | | | | |
| register_m_u | 663 | 707 | 887 | 958D | | | | | | |
| regression | 437D | 570 | 3351 | | | | | | | |
| release | 700D | 842 | 1501 | 1521 | | | | | | |
| remaining | 3119D | 3125 | 3127 | | | | | | | |
| response_tim | 3041D | 3048 | 3075 | 3077 | | | | | | |
| restart_proc | 2185D | 2847 | | | | | | | | |
| results | 105D | 172 | 173 | 175 | 3515 | | | | | |
| retire | 1322 | 1436D | 1694 | 2104 | 2942 | | | | | |
| retrieve | 807D | 821 | 837 | 1808 | 1819 | 1824 | 2375 | | | |
| return | 2079D | 2091 | 2116 | 2275D | 2279 | 2470 | 2514 | 2566 | 2645 | 2794 |
| re_initializ | 2028D | 2268 | | | | | | | | |
| rqh | 2683D | 2725 | 2726 | 2727 | 2782 | 2816 | 2832 | 2865 | 3434 | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| rql | 2683D | 2725 | 2729 | 2730 | 2782 | 2832 | 2855 | 2866 | 3434 | |
| rqstor | 1286 | 1355 | 1374 | 1397 | 1729 | 1731 | 1736 | 1740 | 1810 | 1819 |
| | 1824 | 1966d | 2316D | 2323D | 2868 | 3176 | 3268D | | | |
| rqst_list | 1095 | 1770 | 1772 | 1949D | 1955 | | | | | |
| rt | 3042D | 3072 | 3079M | 3083 | 3084 | 3085 | | | | |
| rts | 1425 | 1642 | 1653 | 1682 | 1684 | 1697 | 2233D | 2295 | | |
| running | 72D | 113 | 198 | 689 | 736 | 1614 | 1758 | 1760 | 2210 | 3012 |
| | 3057 | 3060 | 3241 | 3298 | 3501 | 3504 | | | | |
| runtt | 1573 | 1577 | 1580 | 1585 | 1586 | 2005D | 2032 | 2425 | 2435 | 2437 |
| | 2459 | 2477 | 2509 | 2556 | 2559 | 2576 | 2613 | 2620 | 2624 | 2743 |
| | 2770 | 2855 | 2875 | 2897 | 2931 | 2940 | 2975 | 2985 | | |
| s | 824D | 827M | 828 | 829 | 832D | 837 | 839M | 842 | 1593D | 1597 |
| | 1805d | 1808 | 1810 | | | | | | | |
| sameness | 1121 | 2253D | 2267M | | | | | | | |
| sd | 459D | 471 | 479 | 485 | | | | | | |
| secretary | 2201D | 2213 | 2214 | 2215 | | | | | | |
| secretaryc | 2172D | 2661 | 2886 | | | | | | | |
| seek_choice | 100D | 146 | 1058 | 1148 | 1179 | 1201 | 1203 | | | |
| seek_d_site | 100D | 145 | 1047 | 1445 | 2031 | | | | | |
| seg | 3145D | 3172 | 3176 | 3189 | 3203 | 3247 | 3288 | 3300 | 3304 | |
| segmentc | 546D | 554 | 782 | 825 | 833 | 863 | 864 | 865 | 868 | 906 |
| | 1805 | 1846m | 1853 | 2061 | 2227 | 2328 | 2693 | 2695 | 3145 | 3197 |
| | 3203 | | | | | | | | | |
| seg_table | 782D | 820 | 827 | 841 | 1808 | 1819 | 1824 | 2375 | 3423M | |
| send_domain | 1454D | 1468 | 1476 | | | | | | | |
| send_message | 899D | 923 | 1054 | 1061 | 1173 | 1176 | 1199 | 1225 | 1347 | 1365 |
| | 1377 | 1398 | 1459 | 1723 | 1727 | 1810 | 1819 | 1824 | 2695 | 2727 |
| | 2730 | 2787 | | | | | | | | |
| send_off | 1222D | 1241 | 1272 | | | | | | | |
| service_time | 1578 | 1583 | 2211 | 2282D | 2296 | | | | | |
| set | 1642 | 3115D | | | | | | | | |
| setkey | 2332D | 2333 | 2341 | 2350 | | | | | | |
| setpos | 3455 | 3462 | 3468 | 3474 | 3481 | 3488 | | | | |
| settle_time | 103D | 160 | 3497 | | | | | | | |
| setup_disk_r | 2123D | 2433 | 2560 | 2602 | | | | | | |
| shifts | 1227M | 1236 | 1237 | 1265 | 1314 | 1447 | 2001D | | | |
| short_comman | 567D | 3082 | 3347 | | | | | | | |
| signal_chann | 907 | 926D | 1837 | | | | | | | |
| signal_disk_ | 2689D | 2761 | 2817 | 2833 | 2841 | | | | | |
| signal_free_ | 3193 | 3206 | 3212D | 3256 | 3308 | | | | | |
| signal_read_ | 3222D | 3302 | | | | | | | | |
| simple_param | 2126 | 2239D | 2636 | 2776 | | | | | | |
| simulation | 178 | | | | | | | | | |
| sim_time | 103D | 158 | 3458 | 3501 | 3510 | | | | | |
| site | 546D | 553 | 829 | 839 | 958D | 963M | 965 | 966 | 969 | 970 |
| | 972 | 984 | 992 | 1071D | 1076 | 1078M | 1120 | 1130 | 1352 | 1365 |
| | 1377 | 1385 | 1413 | 1752 | 2025D | 2089 | 2103 | 2142 | 2182 | 2189 |
| | 2198 | 2207 | 2209 | 2221 | 2303 | 2333 | 2368M | 2374 | 2421 | 2505 |
| | 2552 | 2634 | 2644 | 2694 | 2695 | 2716 | 2727 | 2730 | 2777 | 2787 |
| | 2812 | 2872 | 2877 | 2914 | 2918 | 2920 | 2925M | 2942 | 2943M | 2947 |
| | 2949 | 2965 | 2982 | | | | | | | |
| size | 533D | 636 | 647D | 652M | 656 | 659 | 661 | 700D | 706 | 842 |
| | 998d | 1007 | 1015 | 1018 | 1071D | 1078 | 1105 | 1114 | 1117 | 1119 |
| | 1131 | 1136 | 1473 | 1501 | 1520 | 1521 | 1758 | 1969 | 1980 | 2049 |
| | 2123d | 2126 | 2169 | 2210 | 2297 | 2318 | 2325 | 2343 | 2352 | 2367M |
| | 2481 | 2580 | 2648 | 3007 | 3013 | 3014 | 3057 | 3250 | 3251 | 3252 |
| | 3288 | 3300 | 3369D | 3377 | 3383M | 3423M | 3424 | | | |
| size_divider | 87D | 126 | 684 | 687 | 688 | 689 | 733 | 735 | 736 | 982 |
| | 3465 | | | | | | | | | |
| sort_require | 960D | 969 | 973 | | | | | | | |
| spaceclaimed | 100D | 147 | 722 | 1066 | 1291 | | | | | |
| spacecount | 569D | 696 | 3344 | | | | | | | |
| spacefound | 1216D | 1261M | 1263 | 1291 | 1295 | | | | | |
| spaceqempty | 595D | 613 | 655 | 682 | 714 | 741 | 1485 | 1508 | 1894 | |
| spaceq | 577D | 611 | 684 | 715 | 725 | 1897 | 1905 | 3376 | | |
| space_situat | 606D | 633 | 636 | 985 | 992 | | | | | |
| sq | 618 | 896D | 906 | 1836 | 1838 | 3155D | 3177 | 3216 | 3228 | 3233 |
| | 3243 | 3245 | 3413 | | | | | | | |
| sqrt | 471 | 485 | 487 | | | | | | | |
| sr | 703D | 715 | 717 | 720 | 722 | 723 | 725 | 735M | 741 | |
| st | 783D | 795 | 799 | 803 | 812 | 815 | 838 | | | |
| stackp | 1861 | 2252D | 2259 | 2263 | 2265M | 2266 | 2268 | 2269 | 2271 | 2276 |
| | 2278m | 2279 | 2449 | 2966 | | | | | | |
| stack_depth | 78D | 124 | 2250 | 2263 | 2300 | 2449 | | | | |
| stage | 722 | 873 | 1041 | 1047M | 1058 | 1066 | 1093 | 1148 | 1179 | 1195 |
| | 1201 | 1203 | 1291 | 1414 | 1445 | 1712 | 1990D | 2031 | 2194 | 2215 |
| start | 343D | 1578 | 1910 | 3062 | 3075 | 3296 | 3313 | | | |
| start_time | 340D | 347 | 353 | 355 | 361 | 370 | 383D | 389 | 392 | 399 |
| | 406 | | | | | | | | | |
| statistic | 289D | 292 | 297 | 310 | 333 | 380 | 414 | 437 | 497 | 498 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 506 | 514 | 515 | 520 | | | | | | |
| statistic_li | 292 | 294D | 498 | 515 | 3324 | 3441 | | | | |
| status | 550D | 553 | 828 | 839 | 1116 | 1118M | 1129 | 1134 | 1357 | 1370 |
| | 1372 | 1383 | 1392 | 2035 | 2084 | 2140 | 2299 | 2342 | 2351 | 2371M |
| | 2374m | 2378 | 2551 | 2573 | 2627 | 2629 | 2634 | 2643 | 2752 | |
| step1 | 1598 | 2064D | 2413D | 2497D | 2535D | 2594D | 2734D | 2894D | | |
| step2 | 1599 | 2065D | 2428D | 2512D | 2548D | 2608D | 2747D | 2900D | | |
| step3 | 1600 | 2065D | 2441D | 2570D | 2617D | 2791D | 2934D | | | |
| step4 | 1601 | 2065D | 2462D | 2797D | 2957D | | | | | |
| step5 | 1602 | 2065D | 2473D | | | | | | | |
| stop | 350D | 355 | 1583 | 1912 | 3064 | 3077 | 3283 | | | |
| stp1 | 1593 | 1598D | | | | | | | | |
| stp2 | 1593 | 1599D | | | | | | | | |
| stp3 | 1593 | 1600D | | | | | | | | |
| stp4 | 1593 | 1601D | | | | | | | | |
| stp5 | 1593 | 1602D | | | | | | | | |
| string | 202D | 209 | 210 | 211 | | | | | | |
| sub | 211 | 2044 | 2045 | 2134 | 2419 | 2420 | 2421 | 2503 | 2504 | 2505 |
| | 2541 | | | | | | | | | |
| suc | 225 | 274 | 280 | 498 | 502 | 506 | 509 | 515 | 518 | 520 |
| | 523 | 1772 | 1779 | 1784 | 1838M | 1871 | 1873 | 2192 | 2693 | 2695 |
| | 2696 | 2700m | 2701 | 2703 | 2712 | 2727 | 2730 | 2756 | 2757 | 2920 |
| | 2950 | 2963 | 3171 | 3173 | 3186 | 3199 | 3243 | 3245 | 3282 | 3390 |
| | 3394 | 3401 | 3405 | | | | | | | |
| success | 647D | 667 | 673 | | | | | | | |
| supern | 83D | 128 | 2385 | 2907 | | | | | | |
| supervisorf | 2649 | 2652D | | | | | | | | |
| suspend | 2181D | | | | | | | | | |
| switch_conte | 884 | 1528D | 3023 | 3066 | 3132 | | | | | |
| sx | 441D | 446M | 454 | 466M | 468M | 469 | 470 | 473 | 476 | |
| sx2 | 441D | 447M | 454 | 467M | 468 | 470 | | | | |
| sxy | 441D | 448M | 454 | 467M | 469 | 470 | 471 | 473 | | |
| sy | 441D | 446M | 454 | 466M | 469 | 470 | 471 | 473M | 477 | |
| sy2 | 441D | 447M | 454 | 467M | 471 | 473 | | | | |
| sysout | 190 | | | | | | | | | |
| system_initi | 3321D | 3495 | | | | | | | | |
| s_channel | 601D | 616 | 938 | 3414 | | | | | | |
| s_channelc | 601 | 616 | 1829 | 1832 | 2994D | 3007 | 3022 | | | |
| t | 187D | 194 | 208D | 209 | 210 | 211 | 213 | 230D | 231 | 232M |
| | 233 | 368d | 370 | 371 | 373 | 428D | 430 | 2132D | 2133 | 2134 |
| | 2135 | 2417d | 2418 | 2419 | 2420 | 2421 | 2422 | 2501D | 2502 | 2503 |
| | 2504 | 2505 | 2506 | 2539D | 2540 | 2541 | 2542 | | | |
| thinking_con | 3099D | 3371 | 3499 | 3502 | | | | | | |
| thinktime | 3043D | 3044 | 3063 | | | | | | | |
| think_timer | 3041D | 3047 | 3062 | 3064 | | | | | | |
| tied | 1145 | 1218D | 1231M | 1236 | 1263 | 1295 | 1297 | 1313 | 2069D | 2221 |
| time | 184 | 347 | 353 | 355 | 361 | 370 | 389 | 392 | 399 | 405 |
| | 406m | 756 | 1487 | 1577 | 1585 | 1586 | 3072 | 3079 | 3125 | |
| timer | 333D | 603 | 621 | 2282 | 2296 | 3041 | 3047 | 3048 | 3276 | 3295 |
| timeslice | 67D | 111 | 1425 | 1684 | 1697 | 2295 | 3482 | | | |
| time_average | 380D | 605 | 622 | | | | | | | |
| total | 340D | 353M | 360 | 370 | 376 | 383D | 389M | 397 | 406 | 416D |
| | 418 | 424 | 430M | 3507M | 3510 | | | | | |
| total_entrie | 228D | 233 | 268 | 1897 | 1905 | | | | | |
| total_respon | 561D | 3048 | 3327 | 3507 | | | | | | |
| total_size | 1102 | 1119M | 1124M | 1136M | 1140M | 1240 | 1256 | 1270 | 1318 | 2018D |
| trans | 82D | 119 | 1118 | 1134 | 1370 | 1392 | 2342 | | | |
| transfer | 3192 | 3205 | 3279D | 3316 | | | | | | |
| transfers | 3277D | 3294 | 3299 | | | | | | | |
| transfer_in_ | 2674D | 2758 | 2762 | 2781 | 2786 | 2821 | 2845 | | | |
| trans_seg | 2685D | 2696M | 2807 | 2808 | | | | | | |
| ts_clock | 589D | 620 | 1642 | 1653 | 1676 | | | | | |
| type1 | 2395 | 2407D | | | | | | | | |
| type1f | 2482 | 2485D | 2586 | | | | | | | |
| type2 | 2397 | 2493D | | | | | | | | |
| type2f | 2519 | 2522D | | | | | | | | |
| t_length | 88D | 127 | 782 | 783 | 787 | 799 | 815 | 3422 | | |
| u | 2201D | 2209 | 2227D | 2425 | 2431 | 2448 | 2450 | 2509 | 2545 | 2556 |
| | 2601 | 2602 | 2620 | 2627 | 2657 | 2661D | 2873 | 2904 | 2906 | 2908 |
| | 3035d | 3044 | 3050 | 3275D | 3285M | 3287 | 3291 | | | |
| u1 | 2382D | 2383 | 2389 | 2401 | | | | | | |
| uniform | 2904 | 3287 | | | | | | | | |
| uniquenumber | 2242D | 2246 | 2333 | | | | | | | |
| unique | 2241D | 2244M | 2245 | 2246 | | | | | | |
| update | 1733 | 1736 | 1740 | 1763 | 1764 | 1952D | | | | |
| usage | 560D | 2296 | 3328 | 3507 | 3510 | | | | | |
| userid | 3035D | 3046 | 3050 | 3069 | 3089 | | | | | |
| userp | 97D | 161M | 162M | 2601 | | | | | | |
| usert | 97D | 163M | 164M | 2601 | | | | | | |
| user_supervi | 2385 | 2589D | | | | | | | | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| val | 384D | 389 | 390 | 391M | 398 | 406 | | | | |
| valid | 100D | 148 | 873 | 1041 | 1414 | 1712 | 2194 | 2215 | | |
| validated | 1034D | 1041 | 1640 | | | | | | | |
| virtual_proc | 1857 | 1859 | 1985 | 2206 | 2209 | 2227D | 2301 | 2303 | 2329 | 2445 |
| | 2671 | 2889 | 3039 | 3050 | 3404 | | | | | |
| w | 627M | 629 | 630 | 632 | 633 | 636 | 1735 | 1736 | 1739 | 1740 |
| | 1762 | 1763 | 2300 | 2301 | 2302 | 2303 | 2869 | 3237 | 3358D | 3450 |
| | 3479 | 3480 | 3490 | | | | | | | |
| wait_for_d | 100D | 144 | 1047 | 1093 | 1195 | | | | | |
| wait_for_sig | 2197D | 2854 | 2974 | | | | | | | |
| warning_mess | 2686D | 2824 | 2826 | 2849 | | | | | | |
| wfreeq | 2679D | 2697 | 2701 | 2703 | 2754 | 2756 | 2757 | 2862 | 2870 | 3305 |
| work | 759 | 1107 | 1249 | 1416 | 1467 | 1484 | 1508 | 1944D | 1955 | 3398 |
| | 3401 | | | | | | | | | |
| wq | 2679D | 2693 | 2695 | 2696 | 2703 | 2757 | 2816 | 2841 | 2861 | |
| wqh | 2681D | 2697 | 2700M | 2767 | 2830 | 2863 | | | | |
| wql | 2681D | 2697 | 2700 | 2766 | 2830 | 2864 | | | | |
| x | 443D | 446 | 447M | 448 | 1543D | 1571 | 1573 | 1577 | 1578 | 1580 |
| | 1583 | 1584 | 1585M | 1586 | 1595M | 1597 | 1598M | 1599M | 1600M | 1601M |
| | 1602m | 2072d | 2076M | 2079D | 2081 | 2089 | 2091 | 2095D | 2101 | 2103 |
| | 2104 | 2111d | 2116 | 2117 | 2118 | 2123D | 2126 | 2413D | 2419 | 2421 |
| | 2424 | 2425m | 2428D | 2431 | 2433 | 2434M | 2435M | 2437M | 2441D | 2447 |
| | 2456 | 2458 | 2459 | 2462D | 2466 | 2470 | 2473D | 2476M | 2477M | 2497D |
| | 2503 | 2505 | 2508 | 2509M | 2512D | 2514 | 2535D | 2541 | 2544 | 2545M |
| | 2548d | 2551 | 2552M | 2554 | 2556M | 2559M | 2560 | 2561M | 2562 | 2566 |
| | 2570d | 2573 | 2574M | 2575 | 2576 | 2594D | 2599 | 2601M | 2602M | 2604M |
| | 2605 | 2608d | 2611M | 2613 | 2614 | 2617D | 2619 | 2620M | 2624M | 2625 |
| | 2634 | 2639m | 2643 | 2644M | 2645 | 2734D | 2740 | 2743 | 2744 | 2747D |
| | 2752 | 2756 | 2766M | 2767 | 2769 | 2770 | 2775 | 2776 | 2777 | 2779 |
| | 2782 | 2791d | 2794 | 2797D | 2855 | 2894D | 2897M | 2900D | 2904 | 2906 |
| | 2908 | 2909 | 2916 | 2931M | 2934D | 2937 | 2940M | 2942 | 2957D | 2975 |
| xfered_domai | 562D | 1499 | 3338 | | | | | | | |
| xfered_local | 564D | 1809 | 1825 | 3340 | | | | | | |
| xfered_proce | 562D | 1820 | 3339 | | | | | | | |
| xferq | 3157D | 3191 | 3204 | 3233 | 3282 | 3311 | | | | |
| y | 443D | 446 | 447M | 448 | 2095D | 2101 | 2106 | | | |

SIMULATION OF NETWORK WITH 1 SITES AND WITH 6 CONSOLES

```
***************************************************************************************************************************
*        DIRECTLY CONNECTED SITES AND DISK: COPYING OF CODE PERFORMED: BIAS TOWARDS PROCESSOR UTILIZATION                 *
* PERIOD OF SIMULATION (SECS) =2000   NUMBER OF SYSTEM DOMAINS = 20   NUMBER OF COMPILERS = 2                             *
* MEMORY SIZE = 128000BYTES   SIZE DIVIDER CONSTANT= 1024   LOAD SHEDDING FACTOR= 2   MAXIMUM MIGRATIONS= 0               *
* LARGE DOMAINS NOT ALLOWED TO START WHEN GREATEST FREE MEMORY IS LESS THAN    32500 OR NUMBER IN SYSTEM IS GREATER THAN  4 *
* CONSOLE CONTROL SITE= 1   NUMBER OF DISKS= 1   DISK BUFFERS= 3   DISK SITE(S)=  1                                       *
* TIME SLICE =100MSECS.   LONG TIME SLICE = 500MSECS.   COMMUNICATION FREQUENCY (MHZ) = 1.00   RANDOM NUMBER SEED =  787  *
***************************************************************************************************************************
```

113397709 DISKHANDLER IMPOTENT
START OF RUN   NUMBER OF CONSOLES IN THINKING STATE   4   NUMBER AWAITING ENTRY TO SYSTEM   0
END OF RUN   NUMBER OF CONSOLES IN THINKING STATE   4   NUMBER AWAITING ENTRY TO SYSTEM   0

UTILIZATION OF PROCESSORS

SITE 1
TOTAL IDLE TIME 372.1   AVERAGE MEMORY USE   84780   MAXIMUM 123850

NUMBER OF CHANGES OF DOMAIN   11656   NUMBER OF TRANSFERED DOMAINS   0   NUMBER OF TRANSFERED PROCESSORS   0   NUMBER OF TRANSFERED LOCAL SEGMENTS   1941   NUMBER OF FORCED MIGRATIONS   0   NUMBER OF PROCESSORS BLOCKED ON ENTRY TO NETWORK   12
NUMBER OF INCARNATIONS BLOCKED WAITING FOR SPACE   0
RESPONSE TIMES
NUMBER OF COMPLETED SHORT COMMANDS 182   NUMBER OF RESPONSE TIMES OVER 2 SECS   2   NUMBER OF RESPONSE TIMES OVER 5 SECS   0
REGRESSION ANALYSIS OF NON TRIVIAL SERVICE TIMES VERSUS RESPONSE TIME
NUMBER OF DATA POINTS 65
MEAN OF NON TRIVIAL SERVICE TIMES   21.5   MEAN OF RESPONSE TIME   61.0   RESIDUAL STANDARD DEVIATION 35.42
ESTIMATE OF REGRESSION COEFFICIENT   2.33   INTERCEPT   10.90   STANDARD DEVIATION OF REGRESSION COEFFICIENT   0.16
CORRELATION COEFFICIENT 0.883

COMMUNICATIONS SUBSYSTEM
NUMBER OF BYTES TRANSFERED   42470472   NUMBER OF CONTROL MESSAGES (32 BYTES) SENT   12823
DISK PERFORMANCE
NUMBER OF BYTES TRANSFERED   42060136   NUMBER OF COMPLETED TRANSFERS   5569   TOTAL DISK IDLE TIME 1702.8

| CONSOLE | TOTAL THINKING TIME | TOTAL RESPONSE TIME | TOTAL SERVICE TIME | NUMBER OF COMPLETED COMMANDS |
|---|---|---|---|---|
| 1 | 1219.0 | 781.0 | 282.3 | 40 |
| 2 | 1033.6 | 966.4 | 322.5 | 40 |
| 3 | 1097.3 | 902.7 | 338.9 | 39 |
| 4 | 1405.4 | 594.6 | 147.1 | 46 |
| 5 | 848.2 | 1151.8 | 382.1 | 21 |
| 6 | 1636.0 | 364.0 | 144.4 | 61 |

GRAND TOTAL OF RESPONSE TIMES 4760
GRAND TOTAL OF SERVICE TIMES 1617
PERFORMANCE MEASURE 2.94
FRACTION USEFUL PROCESSOR UTILIZATION 0.809

```
SIMULATION OF NETWORK WITH 6 SITES AND WITH 48 CONSOLES


*******************************************************************************************************************************
*       DIRECTLY CONNECTED SITES AND DISK: COPYING OF CODE PERFORMED: BIAS TOWARDS PROCESSOR UTILIZATION                    *
* PERIOD OF SIMULATION (SECS) = 250   NUMBER OF SYSTEM DOMAINS = 21   NUMBER OF COMPILERS = 2                               *
* MEMORY SIZE = 128000BYTES   SIZE DIVIDER CONSTANT= 1024   LOAD SHEDDING FACTOR= 2   MAXIMUM MIGRATIONS= 5                 *
* LARGE DOMAINS NOT ALLOWED TO START WHEN GREATEST FREE MEMORY IS LESS THAN    38000 OR NUMBER IN SYSTEM IS GREATER THAN 24 *
* CONSOLE CONTROL SITE= 1   NUMBER OF DISKS= 2   DISK BUFFERS= 3   DISK SITE(S)=  3  6                                      *
* TIME SLICE =100MSECS.   LONG TIME SLICE = 500MSECS.   COMMUNICATION FREQUENCY (MHZ) = 1.00   RANDOM NUMBER SEED =  787    *
*******************************************************************************************************************************
START OF RUN   NUMBER OF CONSOLES IN THINKING STATE  34  NUMBER AWAITING ENTRY TO SYSTEM   0
END OF RUN   NUMBER OF CONSOLES IN THINKING STATE  26  NUMBER AWAITING ENTRY TO SYSTEM   0

UTILIZATION OF PROCESSORS

SITE  1
    TOTAL IDLE TIME   22.8   AVERAGE MEMORY USE    85923   MAXIMUM  127961
SITE  2
    TOTAL IDLE TIME   11.8   AVERAGE MEMORY USE    85319   MAXIMUM  126082
SITE  3
    TOTAL IDLE TIME    6.9   AVERAGE MEMORY USE    88036   MAXIMUM  127633
SITE  4
    TOTAL IDLE TIME   15.3   AVERAGE MEMORY USE    77067   MAXIMUM  127356
SITE  5
    TOTAL IDLE TIME   12.3   AVERAGE MEMORY USE    78316   MAXIMUM  126079
SITE  6
    TOTAL IDLE TIME    4.8   AVERAGE MEMORY USE    81423   MAXIMUM  126745

   NUMBER OF CHANGES OF DOMAIN    9896   NUMBER OF TRANSFERED DOMAINS  61   NUMBER OF TRANSFERED PROCESSORS    6369   NUMBER OF
TRANSFERED LOCAL SEGMENTS     5187   NUMBER OF FORCED MIGRATIONS     1461   NUMBER OF PROCESSORS BLOCKED ON ENTRY TO NETWORK   0
   NUMBER OF INCARNATIONS BLOCKED WAITING FOR SPACE   0
RESPONSE TIMES
   NUMBER OF COMPLETED SHORT COMMANDS 187   NUMBER OF RESPONSE TIMES OVER 2 SECS   2   NUMBER OF RESPONSE TIMES OVER 5 SECS   0
REGRESSION ANALYSIS OF  NON TRIVIAL SERVICE TIMES VERSUS RESPONSE TIME
NUMBER OF DATA POINTS  59
```

MEAN OF NON TRIVIAL SERVICE TIMES 16.4 MEAN OF RESPONSE TIME 49.7 RESIDUAL STANDARD DEVIATION 13.50
ESTIMATE OF REGRESSION COEFFICIENT 2.92 INTERCEPT 1.68 STANDARD DEVIATION OF REGRESSION COEFFICIENT 0.11
CORRELATION COEFFICIENT 0.965

COMMUNICATIONS SUBSYSTEM
NUMBER OF BYTES TRANSFERED 55422984 NUMBER OF CONTROL MESSAGES (32 BYTES) SENT 29604
DISK PERFORMANCE
NUMBER OF BYTES TRANSFERED 18885714 NUMBER OF COMPLETED TRANSFERS 2563 TOTAL DISK IDLE TIME 115.2
DISK PERFORMANCE
NUMBER OF BYTES TRANSFERED 15662283 NUMBER OF COMPLETED TRANSFERS 2116 TOTAL DISK IDLE TIME 138.9

| CONSOLE | TOTAL THINKING TIME | TOTAL RESPONSE TIME | TOTAL SERVICE TIME | NUMBER OF COMPLETED COMMANDS |
|---|---|---|---|---|
| 1 | 242.8 | 7.2 | 4.4 | 5 |
| 2 | 204.4 | 45.6 | 17.6 | 6 |
| 3 | 24.3 | 225.7 | 63.4 | 1 |
| 4 | 124.2 | 125.8 | 33.4 | 4 |
| 5 | 204.9 | 45.1 | 18.9 | 13 |
| 6 | 98.8 | 151.2 | 37.2 | 5 |
| 7 | 169.2 | 80.8 | 31.6 | 6 |
| 8 | 208.7 | 41.3 | 9.4 | 8 |
| 9 | 248.8 | 1.2 | 0.3 | 5 |
| 10 | 235.3 | 14.7 | 3.5 | 10 |
| 11 | 225.6 | 24.4 | 5.7 | 7 |
| 12 | 23.3 | 226.7 | 91.0 | 1 |
| 13 | 158.6 | 91.4 | 30.6 | 9 |
| 14 | 77.7 | 172.3 | 45.2 | 2 |
| 15 | 216.3 | 33.7 | 11.9 | 8 |

CONSOLE 16

```
    TOTAL THINKING TIME  146.4    TOTAL RESPONSE TIME  103.6    TOTAL SERVICE TIME   25.5    NUMBER OF COMPLETED COMMANDS   5
CONSOLE 17
    TOTAL THINKING TIME  238.8    TOTAL RESPONSE TIME   11.2    TOTAL SERVICE TIME    3.5    NUMBER OF COMPLETED COMMANDS   8
CONSOLE 18
    TOTAL THINKING TIME  243.7    TOTAL RESPONSE TIME    6.3    TOTAL SERVICE TIME    2.7    NUMBER OF COMPLETED COMMANDS   6
CONSOLE 19
    TOTAL THINKING TIME   59.0    TOTAL RESPONSE TIME  191.0    TOTAL SERVICE TIME   52.8    NUMBER OF COMPLETED COMMANDS   4
CONSOLE 20
    TOTAL THINKING TIME  230.0    TOTAL RESPONSE TIME   20.0    TOTAL SERVICE TIME    8.9    NUMBER OF COMPLETED COMMANDS   8
CONSOLE 21
    TOTAL THINKING TIME  214.0    TOTAL RESPONSE TIME   36.0    TOTAL SERVICE TIME   13.4    NUMBER OF COMPLETED COMMANDS   8
CONSOLE 22
    TOTAL THINKING TIME   76.6    TOTAL RESPONSE TIME  173.4    TOTAL SERVICE TIME   49.2    NUMBER OF COMPLETED COMMANDS   2
CONSOLE 23
    TOTAL THINKING TIME  249.0    TOTAL RESPONSE TIME    1.0    TOTAL SERVICE TIME    0.2    NUMBER OF COMPLETED COMMANDS   2
CONSOLE 24
    TOTAL THINKING TIME  236.8    TOTAL RESPONSE TIME   13.2    TOTAL SERVICE TIME    2.4    NUMBER OF COMPLETED COMMANDS   3
CONSOLE 25
    TOTAL THINKING TIME  248.8    TOTAL RESPONSE TIME    1.2    TOTAL SERVICE TIME    0.7    NUMBER OF COMPLETED COMMANDS   7
CONSOLE 26
    TOTAL THINKING TIME    2.2    TOTAL RESPONSE TIME  247.8    TOTAL SERVICE TIME   69.3    NUMBER OF COMPLETED COMMANDS   0
CONSOLE 27
    TOTAL THINKING TIME    0.0    TOTAL RESPONSE TIME  250.0    TOTAL SERVICE TIME   70.6    NUMBER OF COMPLETED COMMANDS   0
CONSOLE 28
    TOTAL THINKING TIME  235.1    TOTAL RESPONSE TIME   14.9    TOTAL SERVICE TIME    4.3    NUMBER OF COMPLETED COMMANDS  10
CONSOLE 29
    TOTAL THINKING TIME   83.3    TOTAL RESPONSE TIME  166.7    TOTAL SERVICE TIME   47.8    NUMBER OF COMPLETED COMMANDS   3
CONSOLE 30
    TOTAL THINKING TIME  190.3    TOTAL RESPONSE TIME   59.7    TOTAL SERVICE TIME   16.2    NUMBER OF COMPLETED COMMANDS   6
CONSOLE 31
    TOTAL THINKING TIME  167.1    TOTAL RESPONSE TIME   82.9    TOTAL SERVICE TIME   24.6    NUMBER OF COMPLETED COMMANDS   4
CONSOLE 32
    TOTAL THINKING TIME  219.3    TOTAL RESPONSE TIME   30.7    TOTAL SERVICE TIME    8.8    NUMBER OF COMPLETED COMMANDS   8
CONSOLE 33
    TOTAL THINKING TIME  184.3    TOTAL RESPONSE TIME   65.7    TOTAL SERVICE TIME   26.9    NUMBER OF COMPLETED COMMANDS   6
CONSOLE 34
    TOTAL THINKING TIME  211.1    TOTAL RESPONSE TIME   38.9    TOTAL SERVICE TIME   10.3    NUMBER OF COMPLETED COMMANDS   6
CONSOLE 35
    TOTAL THINKING TIME  236.9    TOTAL RESPONSE TIME   13.1    TOTAL SERVICE TIME    5.6    NUMBER OF COMPLETED COMMANDS   4
CONSOLE 36
    TOTAL THINKING TIME  230.8    TOTAL RESPONSE TIME   19.2    TOTAL SERVICE TIME    4.5    NUMBER OF COMPLETED COMMANDS   8
```

CONSOLE 37
TOTAL THINKING TIME 67.8 TOTAL RESPONSE TIME 182.2 TOTAL SERVICE TIME 59.3 NUMBER OF COMPLETED COMMANDS 2
CONSOLE 38
TOTAL THINKING TIME 61.2 TOTAL RESPONSE TIME 188.8 TOTAL SERVICE TIME 50.3 NUMBER OF COMPLETED COMMANDS 1
CONSOLE 39
TOTAL THINKING TIME 113.3 TOTAL RESPONSE TIME 136.7 TOTAL SERVICE TIME 55.1 NUMBER OF COMPLETED COMMANDS 5
CONSOLE 40
TOTAL THINKING TIME 160.5 TOTAL RESPONSE TIME 89.5 TOTAL SERVICE TIME 28.8 NUMBER OF COMPLETED COMMANDS 5
CONSOLE 41
TOTAL THINKING TIME 231.5 TOTAL RESPONSE TIME 18.5 TOTAL SERVICE TIME 5.8 NUMBER OF COMPLETED COMMANDS 6
CONSOLE 42
TOTAL THINKING TIME 66.2 TOTAL RESPONSE TIME 183.8 TOTAL SERVICE TIME 66.1 NUMBER OF COMPLETED COMMANDS 4
CONSOLE 43
TOTAL THINKING TIME 110.9 TOTAL RESPONSE TIME 139.1 TOTAL SERVICE TIME 45.7 NUMBER OF COMPLETED COMMANDS 2
CONSOLE 44
TOTAL THINKING TIME 15.6 TOTAL RESPONSE TIME 234.4 TOTAL SERVICE TIME 98.2 NUMBER OF COMPLETED COMMANDS 0
CONSOLE 45
TOTAL THINKING TIME 138.8 TOTAL RESPONSE TIME 111.2 TOTAL SERVICE TIME 34.1 NUMBER OF COMPLETED COMMANDS 10
CONSOLE 46
TOTAL THINKING TIME 51.1 TOTAL RESPONSE TIME 198.9 TOTAL SERVICE TIME 58.6 NUMBER OF COMPLETED COMMANDS 1
CONSOLE 47
TOTAL THINKING TIME 181.2 TOTAL RESPONSE TIME 68.8 TOTAL SERVICE TIME 18.7 NUMBER OF COMPLETED COMMANDS 5
CONSOLE 48
TOTAL THINKING TIME 133.4 TOTAL RESPONSE TIME 116.6 TOTAL SERVICE TIME 30.0 NUMBER OF COMPLETED COMMANDS 7

GRAND TOTAL OF RESPONSE TIMES 4502
GRAND TOTAL OF SERVICE TIMES 1403
PERFORMANCE MEASURE 3.21
FRACTION USEFUL PROCESSOR UTILIZATION 0.935

SIMULATION OF NETWORK WITH 9 SITES AND WITH 54 CONSOLES

```
**************************************************************************************************************************
*        DIRECTLY CONNECTED SITES AND DISK: COPYING OF CODE PERFORMED: BIAS TOWARDS PROCESSOR UTILIZATION                *
* PERIOD OF SIMULATION (SECS) = 222   NUMBER OF SYSTEM DOMAINS = 22   NUMBER OF COMPILERS = 2                            *
* MEMORY SIZE = 128000BYTES   SIZE DIVIDER CONSTANT= 1024   LOAD SHEDDING FACTOR= 2   MAXIMUM MIGRATIONS= 8              *
* LARGE DOMAINS NOT ALLOWED TO START WHEN GREATEST FREE MEMORY IS LESS THAN    38500 OR NUMBER IN SYSTEM IS GREATER THAN 36 *
* CONSOLE CONTROL SITE= 1   NUMBER OF DISKS= 3   DISK BUFFERS= 3   DISK SITE(S)=  3  6  9                                *
* TIME SLICE =100MSECS.   LONG TIME SLICE = 500MSECS.   COMMUNICATION FREQUENCY (MHZ) = 1.00   RANDOM NUMBER SEED =  787 *
**************************************************************************************************************************
START OF RUN   NUMBER OF CONSOLES IN THINKING STATE  41  NUMBER AWAITING ENTRY TO SYSTEM   0
END OF RUN   NUMBER OF CONSOLES IN THINKING STATE  36  NUMBER AWAITING ENTRY TO SYSTEM   0

UTILIZATION OF PROCESSORS

SITE  1
    TOTAL IDLE TIME   60.8   AVERAGE MEMORY USE    71696   MAXIMUM  121965
SITE  2
    TOTAL IDLE TIME   41.6   AVERAGE MEMORY USE    65136   MAXIMUM  126458
SITE  3
    TOTAL IDLE TIME   23.8   AVERAGE MEMORY USE    71719   MAXIMUM  115889
SITE  4
    TOTAL IDLE TIME   46.8   AVERAGE MEMORY USE    62161   MAXIMUM  112435
SITE  5
    TOTAL IDLE TIME   46.7   AVERAGE MEMORY USE    62969   MAXIMUM  112889
SITE  6
    TOTAL IDLE TIME   20.7   AVERAGE MEMORY USE    70847   MAXIMUM  120689
SITE  7
    TOTAL IDLE TIME   40.0   AVERAGE MEMORY USE    66601   MAXIMUM  119485
SITE  8
    TOTAL IDLE TIME   42.8   AVERAGE MEMORY USE    68012   MAXIMUM  123993
SITE  9
    TOTAL IDLE TIME   18.3   AVERAGE MEMORY USE    68560   MAXIMUM  108749

   NUMBER OF CHANGES OF DOMAIN   11139   NUMBER OF TRANSFERED DOMAINS  69   NUMBER OF TRANSFERED PROCESSORS    7626   NUMBER OF
```

TRANSFERED LOCAL SEGMENTS 6768 NUMBER OF FORCED MIGRATIONS 2999 NUMBER OF PROCESSORS BLOCKED ON ENTRY TO NETWORK 0
NUMBER OF INCARNATIONS BLOCKED WAITING FOR SPACE 0

RESPONSE TIMES
NUMBER OF COMPLETED SHORT COMMANDS 211 NUMBER OF RESPONSE TIMES OVER 2 SECS 0 NUMBER OF RESPONSE TIMES OVER 5 SECS 0

REGRESSION ANALYSIS OF NON TRIVIAL SERVICE TIMES VERSUS RESPONSE TIME
NUMBER OF DATA POINTS 65
MEAN OF NON TRIVIAL SERVICE TIMES 17.2 MEAN OF RESPONSE TIME 39.6 RESIDUAL STANDARD DEVIATION 9.19
ESTIMATE OF REGRESSION COEFFICIENT 2.26 INTERCEPT 0.70 STANDARD DEVIATION OF REGRESSION COEFFICIENT 0.07
CORRELATION COEFFICIENT 0.975

COMMUNICATIONS SUBSYSTEM
NUMBER OF BYTES TRANSFERED 66558763 NUMBER OF CONTROL MESSAGES (32 BYTES) SENT 38987

DISK PERFORMANCE
NUMBER OF BYTES TRANSFERED 13383539 NUMBER OF COMPLETED TRANSFERS 1910 TOTAL DISK IDLE TIME 123.8

DISK PERFORMANCE
NUMBER OF BYTES TRANSFERED 14095901 NUMBER OF COMPLETED TRANSFERS 1915 TOTAL DISK IDLE TIME 122.3

DISK PERFORMANCE
NUMBER OF BYTES TRANSFERED 10538990 NUMBER OF COMPLETED TRANSFERS 1447 TOTAL DISK IDLE TIME 146.8

CONSOLE 1
TOTAL THINKING TIME 215.6 TOTAL RESPONSE TIME 6.6 TOTAL SERVICE TIME 4.9 NUMBER OF COMPLETED COMMANDS 5

CONSOLE 2
TOTAL THINKING TIME 184.4 TOTAL RESPONSE TIME 37.9 TOTAL SERVICE TIME 17.6 NUMBER OF COMPLETED COMMANDS 6

CONSOLE 3
TOTAL THINKING TIME 28.6 TOTAL RESPONSE TIME 193.6 TOTAL SERVICE TIME 73.8 NUMBER OF COMPLETED COMMANDS 2

CONSOLE 4
TOTAL THINKING TIME 128.5 TOTAL RESPONSE TIME 93.7 TOTAL SERVICE TIME 33.2 NUMBER OF COMPLETED COMMANDS 4

CONSOLE 5
TOTAL THINKING TIME 185.9 TOTAL RESPONSE TIME 36.4 TOTAL SERVICE TIME 18.3 NUMBER OF COMPLETED COMMANDS 11

CONSOLE 6
TOTAL THINKING TIME 113.9 TOTAL RESPONSE TIME 108.3 TOTAL SERVICE TIME 38.8 NUMBER OF COMPLETED COMMANDS 5

CONSOLE 7
TOTAL THINKING TIME 159.6 TOTAL RESPONSE TIME 62.6 TOTAL SERVICE TIME 31.6 NUMBER OF COMPLETED COMMANDS 6

CONSOLE 8
TOTAL THINKING TIME 195.4 TOTAL RESPONSE TIME 26.8 TOTAL SERVICE TIME 9.0 NUMBER OF COMPLETED COMMANDS 6

CONSOLE 9
TOTAL THINKING TIME 222.0 TOTAL RESPONSE TIME 0.3 TOTAL SERVICE TIME 0.2 NUMBER OF COMPLETED COMMANDS 4

CONSOLE 10
TOTAL THINKING TIME 215.9 TOTAL RESPONSE TIME 6.3 TOTAL SERVICE TIME 2.9 NUMBER OF COMPLETED COMMANDS 9

CONSOLE 11
TOTAL THINKING TIME 220.3 TOTAL RESPONSE TIME 2.0 TOTAL SERVICE TIME 0.5 NUMBER OF COMPLETED COMMANDS 7

CONSOLE 12

TOTAL THINKING TIME 33.6 TOTAL RESPONSE TIME 188.7 TOTAL SERVICE TIME 102.2 NUMBER OF COMPLETED COMMANDS 3

CONSOLE 13

TOTAL THINKING TIME 155.4 TOTAL RESPONSE TIME 66.9 TOTAL SERVICE TIME 30.6 NUMBER OF COMPLETED COMMANDS 9

CONSOLE 14

TOTAL THINKING TIME 82.8 TOTAL RESPONSE TIME 139.4 TOTAL SERVICE TIME 50.4 NUMBER OF COMPLETED COMMANDS 2

CONSOLE 15

TOTAL THINKING TIME 196.8 TOTAL RESPONSE TIME 25.4 TOTAL SERVICE TIME 11.9 NUMBER OF COMPLETED COMMANDS 9

CONSOLE 16

TOTAL THINKING TIME 147.2 TOTAL RESPONSE TIME 75.0 TOTAL SERVICE TIME 27.2 NUMBER OF COMPLETED COMMANDS 6

CONSOLE 17

TOTAL THINKING TIME 212.8 TOTAL RESPONSE TIME 9.4 TOTAL SERVICE TIME 4.4 NUMBER OF COMPLETED COMMANDS 7

CONSOLE 18

TOTAL THINKING TIME 214.4 TOTAL RESPONSE TIME 7.9 TOTAL SERVICE TIME 4.7 NUMBER OF COMPLETED COMMANDS 5

CONSOLE 19

TOTAL THINKING TIME 64.5 TOTAL RESPONSE TIME 157.7 TOTAL SERVICE TIME 63.6 NUMBER OF COMPLETED COMMANDS 4

CONSOLE 20

TOTAL THINKING TIME 204.4 TOTAL RESPONSE TIME 17.9 TOTAL SERVICE TIME 7.6 NUMBER OF COMPLETED COMMANDS 4

CONSOLE 21

TOTAL THINKING TIME 201.6 TOTAL RESPONSE TIME 20.7 TOTAL SERVICE TIME 9.0 NUMBER OF COMPLETED COMMANDS 7

CONSOLE 22

TOTAL THINKING TIME 86.1 TOTAL RESPONSE TIME 136.1 TOTAL SERVICE TIME 56.8 NUMBER OF COMPLETED COMMANDS 4

CONSOLE 23

TOTAL THINKING TIME 221.9 TOTAL RESPONSE TIME 0.3 TOTAL SERVICE TIME 0.2 NUMBER OF COMPLETED COMMANDS 2

CONSOLE 24

TOTAL THINKING TIME 222.0 TOTAL RESPONSE TIME 0.2 TOTAL SERVICE TIME 0.2 NUMBER OF COMPLETED COMMANDS 3

CONSOLE 25

TOTAL THINKING TIME 220.6 TOTAL RESPONSE TIME 1.6 TOTAL SERVICE TIME 0.7 NUMBER OF COMPLETED COMMANDS 7

CONSOLE 26

TOTAL THINKING TIME 7.7 TOTAL RESPONSE TIME 214.5 TOTAL SERVICE TIME 77.3 NUMBER OF COMPLETED COMMANDS 0

CONSOLE 27

TOTAL THINKING TIME 0.0 TOTAL RESPONSE TIME 222.2 TOTAL SERVICE TIME 79.5 NUMBER OF COMPLETED COMMANDS 0

CONSOLE 28

TOTAL THINKING TIME 216.8 TOTAL RESPONSE TIME 5.4 TOTAL SERVICE TIME 2.9 NUMBER OF COMPLETED COMMANDS 9

CONSOLE 29

TOTAL THINKING TIME 99.1 TOTAL RESPONSE TIME 123.1 TOTAL SERVICE TIME 48.6 NUMBER OF COMPLETED COMMANDS 5

CONSOLE 30

TOTAL THINKING TIME 178.7 TOTAL RESPONSE TIME 43.5 TOTAL SERVICE TIME 16.2 NUMBER OF COMPLETED COMMANDS 6

CONSOLE 31

TOTAL THINKING TIME 160.1 TOTAL RESPONSE TIME 62.2 TOTAL SERVICE TIME 24.3 NUMBER OF COMPLETED COMMANDS 3

CONSOLE 32

TOTAL THINKING TIME 220.4 TOTAL RESPONSE TIME 1.8 TOTAL SERVICE TIME 1.0 NUMBER OF COMPLETED COMMANDS 8

CONSOLE 33
TOTAL THINKING TIME 168.8 TOTAL RESPONSE TIME 53.4 TOTAL SERVICE TIME 26.9 NUMBER OF COMPLETED COMMANDS 6
CONSOLE 34
TOTAL THINKING TIME 216.8 TOTAL RESPONSE TIME 5.4 TOTAL SERVICE TIME 2.3 NUMBER OF COMPLETED COMMANDS 6
CONSOLE 35
TOTAL THINKING TIME 210.9 TOTAL RESPONSE TIME 11.4 TOTAL SERVICE TIME 5.6 NUMBER OF COMPLETED COMMANDS 4
CONSOLE 36
TOTAL THINKING TIME 209.4 TOTAL RESPONSE TIME 12.8 TOTAL SERVICE TIME 4.5 NUMBER OF COMPLETED COMMANDS 8
CONSOLE 37
TOTAL THINKING TIME 67.8 TOTAL RESPONSE TIME 154.4 TOTAL SERVICE TIME 68.5 NUMBER OF COMPLETED COMMANDS 2
CONSOLE 38
TOTAL THINKING TIME 67.0 TOTAL RESPONSE TIME 155.3 TOTAL SERVICE TIME 62.8 NUMBER OF COMPLETED COMMANDS 1
CONSOLE 39
TOTAL THINKING TIME 118.9 TOTAL RESPONSE TIME 103.3 TOTAL SERVICE TIME 55.1 NUMBER OF COMPLETED COMMANDS 5
CONSOLE 40
TOTAL THINKING TIME 164.9 TOTAL RESPONSE TIME 57.3 TOTAL SERVICE TIME 18.3 NUMBER OF COMPLETED COMMANDS 4
CONSOLE 41
TOTAL THINKING TIME 210.8 TOTAL RESPONSE TIME 11.4 TOTAL SERVICE TIME 5.6 NUMBER OF COMPLETED COMMANDS 5
CONSOLE 42
TOTAL THINKING TIME 70.4 TOTAL RESPONSE TIME 151.8 TOTAL SERVICE TIME 66.1 NUMBER OF COMPLETED COMMANDS 4
CONSOLE 43
TOTAL THINKING TIME 110.9 TOTAL RESPONSE TIME 111.3 TOTAL SERVICE TIME 48.6 NUMBER OF COMPLETED COMMANDS 2
CONSOLE 44
TOTAL THINKING TIME 21.1 TOTAL RESPONSE TIME 201.1 TOTAL SERVICE TIME 111.3 NUMBER OF COMPLETED COMMANDS 0
CONSOLE 45
TOTAL THINKING TIME 144.6 TOTAL RESPONSE TIME 77.7 TOTAL SERVICE TIME 31.8 NUMBER OF COMPLETED COMMANDS 10
CONSOLE 46
TOTAL THINKING TIME 51.1 TOTAL RESPONSE TIME 171.1 TOTAL SERVICE TIME 65.8 NUMBER OF COMPLETED COMMANDS 1
CONSOLE 47
TOTAL THINKING TIME 186.8 TOTAL RESPONSE TIME 35.4 TOTAL SERVICE TIME 12.4 NUMBER OF COMPLETED COMMANDS 5
CONSOLE 48
TOTAL THINKING TIME 145.1 TOTAL RESPONSE TIME 77.1 TOTAL SERVICE TIME 30.1 NUMBER OF COMPLETED COMMANDS 7
CONSOLE 49
TOTAL THINKING TIME 34.9 TOTAL RESPONSE TIME 187.3 TOTAL SERVICE TIME 100.8 NUMBER OF COMPLETED COMMANDS 0
CONSOLE 50
TOTAL THINKING TIME 221.5 TOTAL RESPONSE TIME 0.7 TOTAL SERVICE TIME 0.6 NUMBER OF COMPLETED COMMANDS 7
CONSOLE 51
TOTAL THINKING TIME 168.5 TOTAL RESPONSE TIME 53.7 TOTAL SERVICE TIME 23.6 NUMBER OF COMPLETED COMMANDS 6
CONSOLE 52
TOTAL THINKING TIME 216.9 TOTAL RESPONSE TIME 5.3 TOTAL SERVICE TIME 3.2 NUMBER OF COMPLETED COMMANDS 6
CONSOLE 53

TOTAL THINKING TIME 190.6 TOTAL RESPONSE TIME 31.6 TOTAL SERVICE TIME 15.4 NUMBER OF COMPLETED COMMANDS 11

CONSOLE 54

TOTAL THINKING TIME 151.6 TOTAL RESPONSE TIME 70.6 TOTAL SERVICE TIME 23.6 NUMBER OF COMPLETED COMMANDS 8

GRAND TOTAL OF RESPONSE TIMES 3834
GRAND TOTAL OF SERVICE TIMES 1633
PERFORMANCE MEASURE 2.35
FRACTION USEFUL PROCESSOR UTILIZATION 0.817

BIBLIOGRAPHY

ABRA70 ABRAMSON N. 'The ALOHA system - another alternative for computer communications' AFIPS FJCC Vol 37 1970 pp 281-285

ADAM62 ADAMS C.W. 'Grosch's law repealed' Datamation Vol 8 No 7 Jul 1962 pp 38-39.

ADAM75 ADAMS J.C. & MILLARD G.E. 'Performance measurements on the Edinburgh multi-access system' EMAS Report 7, Dept of Computer Science Univ. of Edinburgh 1975 14pp.

ADAM76 ADAMS C. 'Over the horizon: a report on the June 1975 computer elements technical committee workshop' Computer Vol 9 No 2 Feb 1976 pp 8-11.

AGRA75 AGRAWALA A.K. & BRYANT R.M. 'Models of memory scheduling' Proc 5th Symp. on O.S. principles (Texas) SIGOPS Vol 9 No 5 Nov 1975 pp 217-222.

AGRA76 AGRAWALA A.K., MOHR J.M. & BRYANT R.M 'An approach to the workload characterisation problem' Computer Vol 9 No 6 Jun 1976 pp 18-32.

AISO75 AISO H. et al. 'A minicomputer complex - KOCOS' Proc 4th Data Communications Symp. (Quebec) IEEE Catalogue 75 CH1001-7DATA, Oct 1975 pp 5_7-5_12.

AKKO72 AKKOYUNLU E. et al. 'An operating system for a network environment' Proc. Symp. on Computer Communications Networks and Teletraffic (Brooklyn) Apr 1972 pp 529-538.

AKKO74 AKKOYUNLU E. et al. 'Interprocess communication facilities for network operating systems' Computer Vol 7 No 6 Jun 1974 pp 46-55.

AKKO75 AKKOYUNLU E.A. et al. 'Some constraints and tradeoffs in the design of network communications' Proc. 5th Symp. on O.S. Principles (Texas) Nov 1975 pp 67-74.

AMDA67 AMDAHL G.M. 'Validity of the single processor approach to achieving large scale computing capabilities' AFIPS SJCC Vol 30 1967 pp 483-485.

ANDE75 ANDERSON G.A & JENSEN E.D. 'Computer interconnection structures: taxonomy, characteristics and examples' Computing Surveys Vol 7 No 4 Dec 1975 pp 197-213.

ARDE75 ARDEN B.W. & BERENBAUM A.D. 'A multi-microprocessor computer system architecture' Proc. 5th Symp. on O.S. principles (Texas) SIGOPS Vol 9 No 5, Nov 1975 pp 114-121.

ATTA76 ATTANASIU C.R., MARKSTEIN P.W.& PHILLIPS R.J. 'Penetrating an operating system: a study of VM/370 integrety' IBM Systems Journal Vol 15 No 1 1976 pp 102-116.

BALZ71 BALZAR R.M. 'Ports - a method for dynamic interprogram communication and job control' AFIPS SJCC Vol 38 1971 pp 485-489.

BASK76 BASKETT F. & SMITH A.J. 'Interference in multiprocessor computer systems with interleaved memory' CACM Vol 19 No 6, Jun 1976 pp 327-334.

BASK72 BASKIN H.B., BORGERSON B.R. and ROBERTS R. 'PRIME - A modular architecture for terminal orientated systems' Proc AFIPS SJCC Vol 40 1972 pp 431-437.

BATS76 BATSON A.P., BRUNDAGE R.E. & KEARNS J.P. 'Design data for ALGOL 60 machines' Proc 3rd Annual Symp. on Computer Architecture, as SIGARCH Vol 4 No 4 Jan 1976 pp 151-154.

BELL70 BELL C.G. et al. 'Computer Networks' Computer Vol 9 No 5 Sept/Oct 1970. pp 13-23

BERN73 BERNARD D. 'Intercomputer networks - an overview and bibliography' Masters Thesis Pennsylvania Univ. May 1973 as Microfiche AD-769 232.

BHAN73a BHANDARKAR D.P. 'Analytic models for memory interference in multiprocessor computer systems' Ph.D. Thesis Computer Science Dept. Carnegie-Mellon Univ. Sept 73.

BHAN73b BHANDARKAR D.P. & FULLER S.H. 'Markov chain models for analysing memory interference in multiprocessor computer systems' Proc. 1st Annual Symp. on Computer Architecture (Florida) Dec 1973 pp 1-6.

BIND75 BINDER R. et al. 'ALOHA packet broadcasting - a retrospect' AFIPS NCC Vol 44 1975 pp 203-215.

BLAK75 BLAKESLEE T.R. 'Digital Design with Standard MSI & LSI' John Wiley and Son (New York) 1975.

BLAN73 BLANC R.P. et al. 'Annotated Bibliography of the Literature on Resource Sharing Computer Networks' National Bureau of Standards - NBS Special Publication 384, 1973.

BORG76 BORGERSON B.R. 'The viability of multimicroprocessor systems' Computer Jan 1976 pp 26-30.

BURN73 BURNET S.J. & COFFMAN E.G. 'Combinatorial problem related to interleaved memory systems' JACM Vol 20 No 1 1973 pp 39-45.

BURR61 BURROUGHS CORP. 'The descriptor - a definition of the B5000 information processing system' The Burroughs Corp. (Detroit) No 5000-20002-P, Feb 1961.

BUZE73 BUZEN J.P. & GAGLIARDI U.O. 'The evolution of virtual machine architecture' AFIPS NCC Vol 42 1973 pp 291-299.

CARR75 CARREN D.M. 'Multiple minis for information management' Datamation Vol 21 No 9 Sept 1975 pp 54-58.

CHOU75 CHOU W. 'Computer communication networks - the parts make up the whole' AFIPS NCC Vol 44 1975 pp 119-128.

CODD62 CODD E.F. 'Multiprogramming' in ALT F.L & RUBINOFF M. (eds) 'Advances in Computers' Vol 3 Academic Press (New York) 1962 pp 77-153.

COHE75 COHEN E. & JEFFERSON D. 'Protection in the HYDRA operating system' Proc. 5th Symp. O.S. Principles (Texas) Nov 1975 pp 141-160.

COLE73 COLEMAN M.L. 'ACCNET - a corporate computer network' AFIPS NCC Vol 42 1973 pp 133-140.

COLO76 COLON F.C. ET ALIA. 'Coupling small computers for performance enhancement' AFIPS NCC Vol 45 1976 pp 755-764.

CORB72 CORBATO F.J., SALTZER J.H. & CLINGEN C.T. 'Multics - the first seven years' AFIPS SJCC Vol 40 1972 pp 571-583.

COSE75 COSELL B.P. et al. 'An operational system for computer resource sharing' Proc. 5th Symp. on O.S. principles (Texas) Nov 1975 pp 75-81.

COSS72 COSSERAT D.C. 'A capability orientated multi-processor system for real-time applications' Proc 1st Int. Conf. on Computer Communication (Washington D.C.) Oct 1972 pp 282-289.

COSS74 COSSERAT D.C. 'A data model based on the capability protection mechanism' IRIA International Workshop on Protection in O.S. (Paris) Aug 1974 pp 35-53.

CRAI74 CRAIG D. & GROOMS D. 'Computer Networks. A Bibliography with Abstracts' NTIS (Springfield) NTIS/WIN 74 081, Oct 1974 169pp

CURT63 CURTIN W.A. 'Multiple computer systems' in ALT F.L. & RUBINOFF M. (eds) 'Advances in Computers' Vol 4 Academic Press (New York) 1963 pp 245-303.

DAHL66 DAHL O.J. and NYGAARD K. 'SIMULA - an ALGOL-based simulation language' CACM Vol 9 No 9 Sept 1966 pp 671-678.

DAHL72 DAHL O.J., MYHRHAUG B. & NYGAARD K. 'Common Base Language' Norwegian Computer Center (Oslo) Publication No S-22, 1970.

DAVI72 DAVIS R.L., ZUCKER S. & CAMPBELL C.M. 'A building block approach to multiprocessing' AFIPS SJCC Vol 40 pp 685-703.

DENN66 DENNIS J.B. & VAN HORN E.C. 'Programming semantics for multiprogrammed computations' CACM Vol 9 No 3 Mar 1966 pp 143-155.

DIJK68 DIJKSTRA E.W. 'The structure of the THE multiprogramming system' CACM Vol 11 No 5 May 1968 pp 341-346.

DIJK71 DIJKSTRA E.W. 'Hierarchical ordering of sequential processes' Acta Informatica Vol 1 No 2 1971 pp 115-138.

DORA75 DORAN R.W. 'Architecture of stack machines' in CHU Y. (Ed) 'High-level Language Computer Architecture' Academic Press (New York) 1975 pp 63-108.

DUVA75 DUVALL W.S. 'POGOS - An Operating System for a Network of Small Machines' Private Communication 1975.

ELRO70 ELROD T.H. 'The CDC 7600 and SCOPE 76' Datamation Vol 16 No 4 April 1970 pp 80-85.

ENGL72 ENGLAND D.M. 'Architectural features of system 250' Infotech State of the Art Report 14; Operating Systems, 1972.

ENGL74 ENGLAND D.M. 'Capability concept mechanism and structure in system 250' IRIA International Workshop on Protection in O.S. (Paris) Aug 1974 pp 63-82.

ESTR67 ESTRIN G. & KLEINROCK L. 'Measures, models and measurements for time-shared computer utilities' Proc. 22nd Nat. Conf. of ACM 1967 pp 85-96.

EVAN67 EVANS D.C. & LECLERC J.Y. 'Address mapping and the control of access in an interactive computer' AFIPS SJCC Vol 30 1967 pp 23-30.

FABR73 FABRY R.S. 'Dynamic verification of operating system decisions' CACM Vol 16 No 11 Nov 1973 pp 659-688.

FABR74 FABRY R.S. 'Capability-based addressing' CACM Vol 17 No 7 Jul 1974 pp 403-412.

FARB72 a FARBER D.J. et al. 'The distributed computer system' Compcon 72 - Proc. 7th Int. Computer Soc. Conf. 1972 pp 31-34.

FARB72 b FARBER D.J. and HEINRICH F.R. 'The structure of the distributed computer system - the distributed file system' Proc 1st Int. Conf. on Computer Communication (Washington) Oct 1972 pp 364-370.

FARB72 c FARBER D.J. & LARSON K.C. 'The system architecture of the distributed computer system - the communications system' Symp. Computer-Communications Networks and Teletraffic (New York) Apr 1972 pp 21-27.

FARB72 d FARBER D.J. & LARSON K.C 'The structure of a distributed computing system - software' Proc. Symp. Computer-Communications Networks and Teletraffic (New York) Apr 1972 pp 539-545.

FARB74 FARBER D.J. 'Software considerations in distributed architectures' Computer Vol 7 No 3 Mar 1974 pp 31-35

FARB75 FARBER D.J. 'A ring network' Datamation Vol 21 No 2 Feb 1975 pp 44-46.

FERR74 FERRIE J. et alia. 'An extensible structure for protected systems design' IRIA International Workshop on Protection in Operating Systems (Paris) Aug 1974 pp 83-105.

FEUS73 FEUSTAL E.A. 'On the advantages of tagged architectures' IEEE Trans. on Computers Vol C-22 No 7 July 1973 pp 644-656.

FLYN72 FLYNN M.J. 'Some computer organizations and their effectiveness' IEEE Trans. on Computers Vol C-21 No 9, Sept 1972 pp 948-960.

FOST72 FOSTER C.C. 'A view of computer architecture' CACM Vol 15 No 7 Jul 1972 pp 557-565.

FRAN72 FRANK H., KAHN R. & KLEINROCK L. 'Computer communication network design - experience with theory and practice' AFIPS SJCC Vol 40 1972 pp 255-270.

FRED73 FREDERICKSEN D.H. 'Describing data in computer networks' IBM Systems Journal Vol 3 No 3 1973 pp 257-282.

FUCH68 FUCHEL K. & HELLER S. 'Considerations in the design of a multiple computer system with extended core storage' CACM Vol 11 No 5 May 1968 pp 334-340.

FULL76 FULLER S.H. 'Price/performance comparison of C.mmp and the PDP-10' Proc 3rd Annual Symp. on Computer Architecture, as SIGARCH Vol 4 No 4, Jan 1976 pp 195-202.

GECC75 GEC COMPUTERS LIMITED 'GEC 4080 Technical Description' (Borehamwood England WD6 1RX) June 1975.

GHEZ73 GHEZZI C. et al. 'Introduction to POLI computer network design' Proc ACM Int. Computing Symp. (Davos) 1973 pp 271-278.

GOLD73 GOLDBERG R.P. 'Architecture of virtual machines' AFIPS NCC Vol 42 1973 pp 309-318.

GOLD74 GOLDBERG R.P. & HASSINGER R. 'The double paging anomaly' AFIPS NCC Vol 43 1974 pp 195-199.

GOOD73 GOODWIN R.J. 'A Design for a Distributed Control Multiple-Processor Computer System' Masters thesis: Naval Postgraduate School (Monterey, Calif.) as Microfiche AD 722 883, 1973 42pp

GRAH72 GRAHAM G.S. & DENNING P.J. 'Protection - principles and practice' AFIPS SJCC Vol 40 1972 pp 417-429.

GRAH68 GRAHAM R.M. 'Protection in an information processing utility' CACM Vol 11 No 5 1968 pp 365-369.

GROS53 GROSCH H.R.J. 'High speed arithmetic: the digital computer as a research tool' Journal of the Optical Society of America Vol 43 No 4 Apr 1953 pp 306-310.

GROS76 GROSCH H.R.J. 'Distributed intelligence' Computer World Vol 10 No 23 June 7th 1976.

HANS70 HANSEN P.B. 'The nucleus of a multiprogramming system' CACM Vol 13 No 4 Apr 1970 pp 238-241 and p 250.

HANS73 HANSEN P.B. 'Operating System Principles' Prentice-Hall (Englewood Cliffs) 1973.

HANS74 HANSEN P.B. 'A program methodology for operating system design' IFIP Congress (Stockholm) 1974 pp 394-397.

HANS75 HANSEN P.B. 'The programming language Concurrent Pascal' IEEE Trans. on Software Engineering Vol SE-1 No 2 June 1975 pp 199-207.

HANS76 HANSEN P.B. '"The SOLO papers"' Software Practice and Experience Vol 6 No 2 1976 pp 139-205.

HAYN73 HAYNES J. 'Please don't interrupt me while I'm computing!' Computer Vol 6 no 12 Dec 1973 pp 45-47.

HEAR73 HEART F.E. et al. 'A new minicomputer/multiprocessor for the ARPA network' AFIPS NCC Vol 42 1973 pp 529-537.

HICK71 HICKEN G.M. 'Experience with an information network' Digest Proc. IEEE Conf. on Hardware Software Firmware Trade-offs (Boston) Sep 1971 pp 169-170.

HIGB73 HIGBIE L.C. 'Supercomputer architecture' Computer Vol 6 No 12 Dec 1973 pp 48-58.

HIRC74 HIRCH P. 'SITA: rating a packet switched network' Datamation Mar 1974 pp 60-63.

HOAR73 HOARE C.A.R. 'A structured paging system' Computer Journal Vol 16 No 3 1973 pp 209-215.

HOAR74 a HOARE C.A.R. 'Monitors: an operating system structuring concept' CACM Vol 17 No 10 Oct 1974 pp 549-557.

HOAR74 b HOARE C.A.R. 'A structured operating system' Presented at SRC Summer School on Computer Architecture and Operating Systems (Cambridge) Sep 1974.

HOLB75 HOLBAEK-HANSSEN E., HANDLYKKEN P. & NYGAARD K. 'System Description and the DELTA Language' DELTA Project Report No 4, Norwegian Computing Center (Oslo) 1975.

HOWE72 HOWELL R.H. 'The integrated computer network system' Proc 1st Int. Conf. on Computer Communication (Washington) Oct 72 pp 214-219.

HUTC68 HUTCHINSON G.K. 'Some problems in the simulation of multiprocessor systems' in BUXTON J.N. (Ed) 'Simulation Programming Languages' North-Holland (Amsterdam) 1968 pp 305-324.

ICHB74 ICHBIAH J.D & MORSE S.P. 'General concepts of the SIMULA 67 programming language' in HALPER et al. (Eds) 'Annual Review in Automatic Programming' Pergamon Press (Oxford) 1974 pp 65-93.

JAGE74 JAGERSTROM J. 'A multi mini system' Proc. European Computing Congress May 1974 pp 717-725.

JONE71 JONES P.D., LINCOLN N.R. & THORNTON J.E. 'Whither computer architecture' IFIP Congress (Ljubljana) 1971 pp 729-736.

JOSE74 JOSEPH E.C. 'Innovation in heterogeneous and homogeneous distributed function architectures' Computer Vol 7 No 3 Mar 1974 pp 17-24.

KAUB76 KUBISCH W.H., PERROT R.H. & HOARE C.A.R. 'Quasiparallel programming' Software - Practice and Experience Vol 6 No 3 1976 pp 341-356.

KIMB75 KIMBLETON S.R. & SCHNEIDER G.M. 'Computer communication networks: approaches, objectives and performance considerations' Computing Surveys Vol 7 No 3 Sep 1975 pp 129-173.

KLEI68 KLEINROCK L. 'Certain analytic results for time-shared processors' IFIP Congress (Edinburgh) 1968 pp 838-845.

KLEI70 KLEINROCK L. 'Analytic and simulation methods in computer network design' AFIPS SJCC Vol 36 1970 pp 569-579.

KLEI74 KLEINROCK L. 'Resource allocation in computer systems and computer-communication networks' IFIP Congress (Stockholm) 1974 pp 11-18.

KLEI75 KLEINROCK L. 'Queueing Systems- Volume 1: Theory' John Wiley (New York) 1975.

KLEI76 KLEINROCK L. 'Queueing Systems - Volume 2: Computer Applications' John Wiley (New York) 1976.

KNIG66 KNIGHT K.E. 'Changes in computer performance' Datamation Vol 12 No 9 Sep 1966 pp 40-54.

KNOT74 KNOTT G.D. 'A proposal for certain process management and intercommunication primitives' SIGOPS Vol 8 No 4 Oct 1974 pp 7-44, continued in Vol 9 No 1 Jan 1975 pp 20-41.

LAMP71 LAMPSON B.W. 'Protection' Proc. 5th Princeton Conf. on Information Sciences and Systems, Mar 1971 pp 437-443, reprinted in SIGOPS Vol 8 No 1 Jan 1974 pp 18-24.

LAMP74 LAMPSON B.W. 'Redundancy and robustness in memory protection' IFIP Congress (Stockholm) 1974 pp 128-132.

LAMP76 LAMPSON B.W. & STURGIS H.E. 'Reflections on an operating system design' CACM Vol 19 No 5 May 1976 pp 251-265.

LAYM74 LAY W.M., MILLS D.L. and ZELKOWITZ M.V. 'Operating systems architecture for a distributed computer network' Computer Networks: Conf. IEEE Computer Soc. and NBS, (Gaithersburg) May 1974 pp 39-44.

LEEA66 LEE A.M. 'Applied Queueing Theory' Studies in Management Series, Macmillan (London) 1966.

LEIN58 LEINER A.L. et alia 'PILOT, the NBS multicomputer system' Proc. Eastern Joint Computer Conf. (Philadelphia) 1958 pp 71-75.

LEVI75 LEVIN R. et alia 'Policy/mechanism separation in HYDRA' Proc 5th Symp. on O.S. Principles (Texas) Nov 1975 pp 132-140.

LIND71 LINDSAY P.J. 'A simple asynchronous interface for linking small computers' Proc DECUS Conf. 1971 pp 253-256.

LIST76 LISTER A.M. & MAYNARD K.J. 'An implementation of monitors' Software - Practice and Experience Vol 6 No 3 1976. pp 377-385.

MADN68 MADNICK S.E. 'Multi-processor software lockout' Proc 23rd National Conf. ACM 1968 pp 19-24.

MADS72 MADSEN O.B. 'Karoline: a network computer project' RECAU-72-14, University of Aarhus, Denmark 1972.

MANN75 MANNING E. and PEEBLES R.W. 'Segment transfer protocols for a homogeneous computer network' ACM Operating Systems Review Vol 9 No 3 Jul 1975 pp 65-73.

MCQU72 McQUILLAN J.M. et alia. 'Improvements in the design and performance of the ARPA network' AFIPS FJCC Vol 41 1972 pp 741-754.

METC72 a METCALFE R.M. 'Strategies for interprocess communication in a distributed computing system' Symp. on Computer-Communications Networks and Teletraffic (Brooklyn) Apr 1972 pp 519-526.

METC72 b METCALFE R.M. 'Strategies for operating systems in computer networks' Proc. of 25th Annual Conf. of ACM (Boston) 1972 pp 278-281.

METC76 METCALFE R.M. & BOGGS D.R. 'Ethernet: distributed packet switching for local computer networks' CACM Vol 19 Jul 1976 pp 395-404.

MEYE70 MEYER R.A. & SEAWRIGHT L.H. 'A virtual machine time-sharing system' IBM Systems Journal Vol 9 No 3 1970 pp 199-218.

MILL76 MILLS D.L. 'An overview of the distributed computer network' AFIPS NCC Vol 45 1976 pp 523-531.

MOOR71 MOORE C. 'Network Model of Time Shared Computer Systems' Ph.D. Thesis Univ. of Michigan (Ann Arbor) Microfiche AD 727 206, 1971.

NEED72 NEEDHAM R.M. 'Protection systems and protection implementations' AFIPS FJCC Vol 41 1972 pp 571-578.

NEED74 NEEDHAM R.M. & WILKES M.V. 'Domains of protection and the management of processes' The Computer Journal Vol 17 No 2 May 1974 pp 117-120.

OPDE75 OPDERBECK H. 'Problems in the design of control procedures for computer networks' ACM Computer Communication Review Vol 5 No 2 Apr 1975 pp 1-7.

ORGA73 ORGANICK E.I. 'Computer System Organization - The B5700/B6700 Series' ACM Monograph Series, Academic Press (New York) 1973.

ORNS75 ORNSTEIN S.M. et alia. 'PLURIBUS - a reliable multiprocessor' Proc AFIPS NCC Vol 44 1975 pp 551-559.

PARM72 PARMLEE R.P. et alia 'Virtual storage and virtual machine concepts' IBM Systems Journal Vol 11 No 2 1972 pp 99-130.

PARN72 PARNAS D.L. 'On the criteria to be used in decomposing systems into modules' CACM Vol 15 No 12 Dec 1972 pp 1053-1058.

PARN74 a PARNAS D.L. 'On a "buzzword": hierachical structure' IFIP Congress (Stockholm) 1974 pp 336-339.

PARN74 b PARNAS D.L & PRICE W.E. 'Using memory access control as the only protection mechanism' IRIA Int. Workshop on Protection in Operating Systems, (Paris) Aug 1974 pp 177-181.

POPE74 PUPEK G.J. & KLINE C.S. 'Verifiable secure operating system software' AFIPS NCC Vol 43 1974 pp 145-151.

PYKE74 PYKE T.N. Private communication, 1974.

POUZ73 POUZIN L. 'Network architectures and components' Proc of 1st European Workshop on Computer Networks (Arles) Apr 1973 pp 227-266.

REAM76 REAMES C.C. & LIU M.T. 'Design and simulation of the distributed loop computer network (DLCN)' Proc 3rd Annual Symp. on Computer Architecture, as SIGARCH Vol 4 No 4 Jan 1976 pp 124-129.

REDE74 REDELL D.D. 'Naming and Protection in Extensible Operating Systems' Ph.D. Thesis M.I.T., Project MAC Report MAC-TR-140, Nov 1974.

RETZ75 RETZ D.L. 'Operating system design considerations for the packet-switching environment' AFIPS NCC Vol 44 1975 pp 155-160.

REYL74 REYLING G. 'Performance and control of multiple microprocessor systems' Computer Design Vol 13 No 3 Mar 1974 pp 81-87.

ROBE70 'ROBERTS L.G. and WESSLER B.D. 'Computer network development to achieve resource sharing' AFIPS SJCC Vol 36 1970 pp 543-549.

ROWA74 ROWAN J.H., SMITH D.A. & SWENSEN M.D. 'Towards the design of a network manager for a distributed computer network' in FENG I. (Ed) 'Parallel Processing' Lecture Notes in Computer Science No 24, Springer Verlag (Berlin) 1974.

ROWE73 ROWE L.A. et al. 'Software methods for achieving fail-soft behaviour in the distributed computer system' IEEE Symp. on Computer Software Reliability (New York) 1973 pp 7-11.

SALT66 SALTZER J.H. 'Traffic control in a multi-plexed computer system' MIT Technical Report MAC-TR-30 Jul 1966.

SALT74 SALTZAR J.H. 'Protection and control of information sharing in Multics' CACM Vol 17 No 7 Jul 1974 pp 388-402.

SCHA75 SCHAEFER H.F. 'Are minicomputers suitable for large scale scientific computations?' 11th IEEE Comp. Soc. Conf., Fall Compcon75 (Washington) Sep 1975 pp 61-64.

SCHE67 SCHERR A.L. 'An Analysis of Time-Shared Computer Systems' M.I.T. Research Monograph No 36, The M.I.T. Press (Massachusetts) 1967.

SCHR72 SCHROEDER M.D. & SALTZAR J.H. 'A hardware architecture for implementing protection rings' CACM Vol 15 No 3 Mar 1972 pp 157-170.

SCOT74 SCOTT C.T. 'An annoted bibliography on computer systems reliability' Infotech State of the Art Report 20, 1974.

SEAR75 SEARLE B.C. & FREBERG D.E. 'Microprocessor applications in multiple processor systems' Computer Vol 8 No 10 Oct 1975 pp 22-30.

SELI72 SELIGMAN L. 'LSI and minicomputer system architecture' AFIPS SJCC Vol 40 1972 pp 767-773.

SEVC72 SEVCIK K.C. et alia. 'Project SUE as a learning experience' AFIPS FJCC Vol 41 1972 pp 331-338.

SEVC74 SEVCIK K.C. & TSICHRITZIS D.C. 'Authorization and access control within overall system design' IRIA Int. Workshop on Protection in Operating Systems (Paris) Aug 1974 pp 211-224.

SMIT72 SMITH B.T. 'Mixed computer networks: benefits, problems and guidelines' Proc. 1st Int. Conf. on Computer Communication (Washington) Oct 1972 pp 201-208.

SPIE73 a SPIER M.J. 'A model implementation for protective domains' Int. Journal of Computer and Information Sciences Vol 2 No 3 Sep 1973 pp 201-229.

SPIE73 b SPIER M.J. 'Process communication prerequisites or the IPC-setup revisited' Proc of the 1973 Sagamore Computer Conf. on Parallel Processing, IEEE publication #73 CH0812-8 C Aug 1973 pp 79-88.

SPIE74 SPIER M.J., HASTINGS T.N. and CUTLER D.N. 'A storage mapping technique for the implementation of protective domains' Software - Practice and Experience Vol 4 No 3 1974 pp 215-230.

SPOO71 SPOONER C.R. 'A software architecture for the 70's: part 1 - the general approach' Software - Practice and Experience Vol 1 No 1 pp 5-37.

STEP74 STEPHENS P.D. 'The IMP language and compiler' The Computer Journal Vol 17 No 3 1974 pp 216-221.

TANG76 TANG C.K. 'Cache system design in the tightly coupled multiprocessor system' AFIPS NCC Vol 45 1976 pp 749-753.

THOM72 THOMAS R.H. & HENDERSON D.A. 'McROSS - a multi-computer programming system' AFIPS SJCC Vol 40 1972 pp 281-293

THOM73 THOMAS R.H. 'A resource sharing executive for the ARPANET' AFIPS NCC Vol 42 1973 pp 155-163.

THUR75 THURBER K.J. & WALD L.D. 'Associative and parallel processors' Computing Surveys Vol 7 No 4 Dec 1975 pp 215-255.

TJAD70 TJADEN G.S. & FLYNN M.J. 'Detection and parallel execution of independent instructions' IEEE Trans. on Computers Vol C19 No 10 Oct 1970 pp 889-895.

TYME71 TYMES L.R. 'TYMNET - A terminal orientated communication network' AFIPS SJCC Vol 38 1971 pp 211-216.

WALD72 WALDEN D.C. 'A system for interprocess communication in a resource sharing computer network' CACM Apr 1972 pp 221-230

WENS75 WENSLY J.H. 'The impact of electronic disks on system architecture' Computer Vol 8 No 8 Feb 1975 pp 44-48.

WHIT73 WHITFIELD H. & WIGHT A.S. 'EMAS - the Edinburgh Multi-access system' The Computer Journal Vol 16 No 4 1973 pp 331-346.

WIDD76 WIDDOES L.C.Jr. 'The Minerva multi-processor' Proc 3rd Annual Symp. on Computer Architecture, as SIGARCH Vol 4 No 4 Jan 1976 pp 34-39.

WILK73 WILKES M.V. 'The dynamics of paging' The Computer Journal Vol 16 No 1 1973 pp 4-9.

WIRC75 WIRCHING J.E. 'Computer of the 1980's - is it a network of microcomputers' Proc Fall Compcom75 (Washington), IEEE publication 75CH0988-6C, Sep 1975 pp 23-26.

WIRT74 WIRTH N. 'On the design of programming languages' IFIP Congress (Stockholm) 1974 pp 386-393.

WITH75 WITHINGTON F.G. 'Beyond 1984: a technology forecast' Datamation Vol 21 No 1 Jan 1975 pp 54-73.

WITT68 WITT B.I. 'M65MP: an experiment in OS/360 multiprocessing' Proc 23rd Nat. Conf. of ACM 1968 pp 691-703.

WITT76 WITTIE L.D. 'Efficient message routing in mega-micro-computer networks' Proc 3rd Annual Symp. on Computer Architecure, IEEE publication 76CH1043-5C, Jan 1976 pp 136-140.

WULF72 WULF W.A. and BELL C.G. 'C.mmp - a multi-mini-processor' AFIPS FJCC Vol 41 1972 pp 765-777.

WULF74 WULF W.A. et al. 'HYDRA: the kernel of a multiprocessor operating system' CACM Vol 17 No 6 Jun 1974 pp 337-345.

WULF75 a WULF W.A. & LEVIN R. 'A local network' Datamation Vol 21 No 2 Feb 1975 pp 47-50.

WULF75 b WULF W.A., LEVIN R. and PIERSON C. 'Overview of the HYDRA operating system development' Proc 5th Symp. O.S. Principles (Texas) Nov 1975 pp 122-131.

ZELK74 ZELKOWITZ M.V. 'Structured operating system organization' Information Processing Letters Vol 3 No 2 Nov 1974 pp.