PRIMON:

The Implementation of a Software Monitor

by

Anthony James McGregor

A thesis presented in partial fulfilment of the

requirements for the degree of

Master of Science in Computer Science

at

Massey University

March 1984

ABSTRACT

This thesis discusses the underlying theory and practical aspects of implementing a software monitor. Current computer performance evaluation techniques are reviewed, emphasising the role of software monitoring both as a self-contained tool and as the source of data for system modelling. The implementation of Primon, a software monitoring package for the Prime 50 series, is discussed in detail.

ACKNOWLEDGEMENTS

I would like to thank,

Tom Docker, my supervisor, particualarly for his efforts while criticising the many drafts of this thesis, and

Susan McGregor, for her help with spelling.

Table of Contents

1 Introduction

Computer Performance Evaluation (CPE) has developed both as an academic and an applied subject of considerable importance, as witnessed by its mention in Discussion Paper 3 for the recent Brownlie Committee report [BROW82]. As an applied discipline CPE is used primarily in the areas of

- o computer design and development
- o capacity planning and
- o system tuning.

A need for access to CPE tools can be identified when administering most computer systems and to this end a range of tools is available on most (large) computers. Prime computers are a notable exception and several groups involved with Prime equipment have shown a desire for access to CPE tools. These groups include computer centres, who wish to improve the performance of the computer systems under their control, researchers in CPE, and teaching staff who wish to demonstrate the use of performance evaluation.

This thesis reports on the implementation of one such tool called Primon. We begin in chapters two and three by discussing CPE in general and measurement in particular. Following this, in chapters

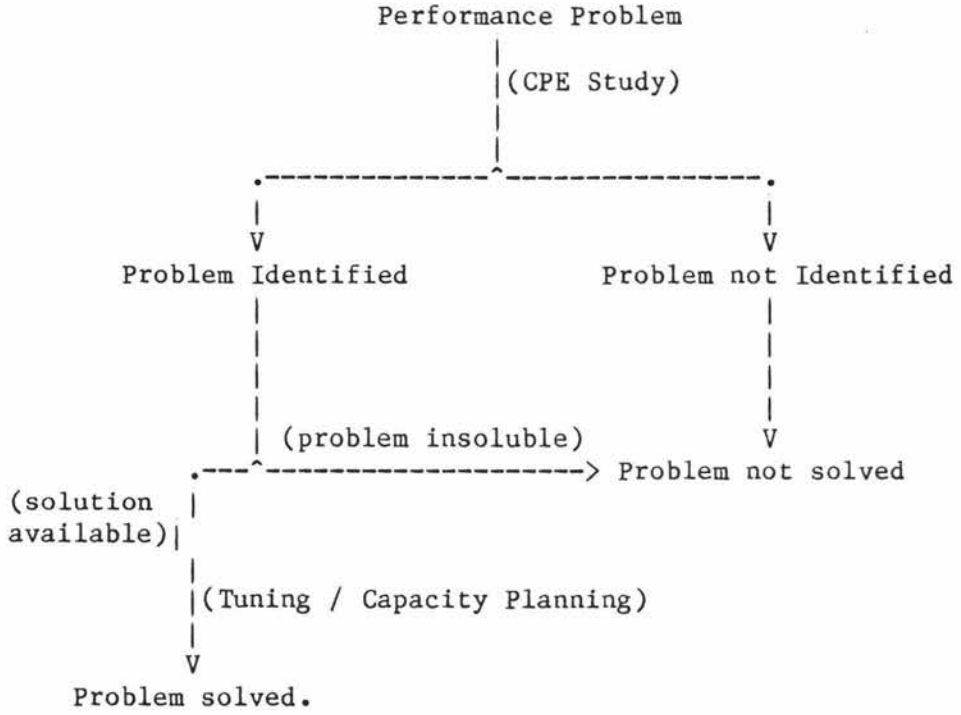
four and five we discuss the specific area of measuring Prime computers and the implementation of Primon. Finally in chapter six we review the development of Primon and suggestions made in previous chapters for further work that could usefully be undertaken. Details of the architecture and systems software of Prime systems are contained in Appendix A, and the program listings for Primon are given in Appendix B.

2 An introduction to computer performance evaluation

The performance evaluation of computer systems has become widespread because of the high cost and complex nature of modern computers.

As an example of the use of CPE consider a computer system operated by Superior Operational Software (SOS), a fictitious software company writing system software for Prime computers. The system functions well until the number of interactive users exceeds a threshold. The System Administrator is faced with the task of raising this threshold, while remaining within a strict capital budget.

He may proceed as in Figure 2.1 with a CPE study. The aim of the study will be to identify the reasons for the current threshold, and it is hoped that this will lead to an informed decision being made on whether the performance can be improved by tuning the operating system constants which the System Administrator has access to (e.g.the length of the time slice, size of buffers etc) or whether the system will need to be upgraded to meet requirements (capacity planning).



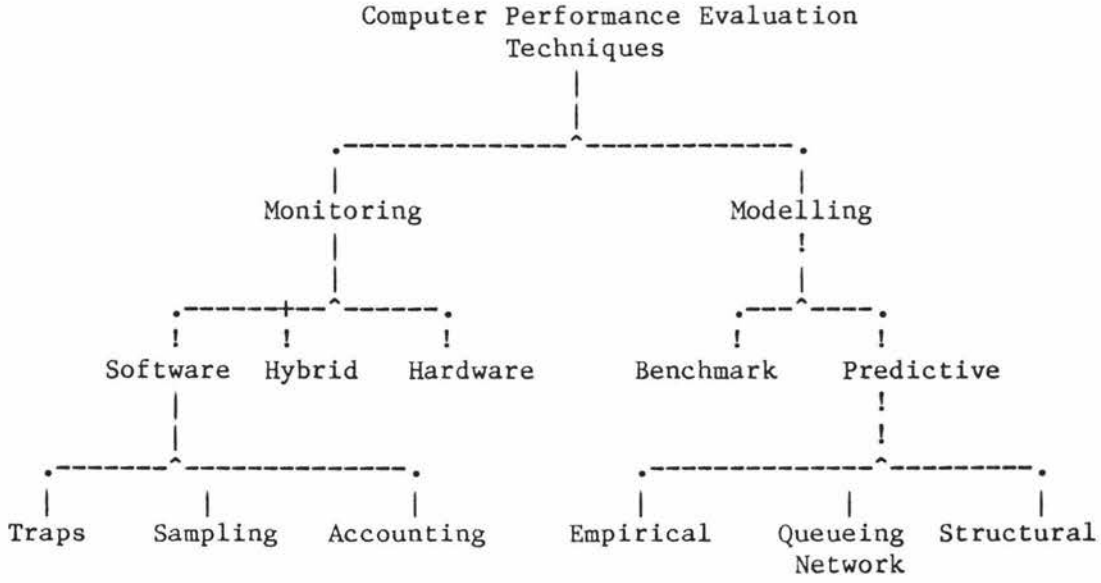
Example use of CPE
Fig 2.1

This type of scenario is common but is by no means the only use of CPE. Performance questions also arise when designing new systems, or when choosing a new or replacement system. Hopefully a company installing a computer system for the first time will want to be satisfied that the proposed system will have sufficient capacity to process the workload within an acceptable time frame. A manufacturer designing a new machine may have specified design criteria which include instruction execution rate improvements over a previous model. Performance Evaluation is used in these situations as well.

From the above discussion it is clear that CPE studies are performed on many types of system, and have a wide range of objectives. This

has led to a range of tools to be developed for use in CPE. Many of these tools have been developed for a particular CPE project, while others are Operations Research techniques. Fig 2.2, and the rest of this chapter, describe some of the more common techniques and their relationships to one another. Other categorizations are possible and some techniques are also known by other names.

A specific monitoring tool could be made up of more than one of these techniques. For an example see the hybrid model of [SCHW78].



The CPE tree
Fig 2.2

2.1 Monitors

To continue the case of SOS Software, the System Administrator believes that the disk drives are the cause of the problem and instructs his staff to make changes to the operating system so that it will monitor the number of I/Os and the number of users logged onto the machine. This data will allow a plot of I/O activity against the number of uses to be drawn. He also arranges for a specialised micro-computer to be connected to the CPU and to each disk drive. Every few milliseconds this computer monitors the state, busy or idle, of each device and produces a plot of the device utilisation against the time of day. It is hoped that on analysis this data will show whether there is a relationship between the interactive threshold and the I/O rate.

In general monitoring tools collect, and present, data from an existing computer system. They are normally categorized by the method(s) used to collect the measurement data.

The micro-computer used by SOS to collect device utilisations is an example of a hardware monitor. Such devices are typically specialised small computers, and are connected to the system to be measured by passive electronic probes. This allows data to be collected without the need to alter the way the host (or target) system functions. The probes may be attached to the backplane of the computer, the pins of a chip, connecting cables, or any other point where the required data can be accessed. A good introduction to

hardware monitoring can be found in [CARL76a] and [CARL76b].

Pieces of monitoring code that are executed by the system to be measured, such as the operating system modification made by SOS, are called software monitors. In chapter 3 we consider the ways that software monitors collect data, particularly event trapping, where code is inserted into the operating system (the case with the SOS software monitor), and sampling, where an extra process is introduced to take samples of the data of interest. We also consider methods of recording the data that has been collected and methods of presenting the data recorded. Data presentation is an important part of any monitoring exercise.

A special type of software monitor found on most computers is the accounting system, which collects and records data on the resources used by individual users for charging purposes. This data is periodically collated into a billing report. While the primary function of this monitor is to allow a charging policy to be implemented, data is often recorded which can be of interest to the performance analyst [COX76].

A third class, called hybrid monitors because they are made up of a combination of hardware and software, is often given theoretical consideration. In practice hybrid monitors are not often implemented as such, but the hardware and software of a computer system may be designed with some consideration of further monitoring requirements. The operating system may, for example, keep a count of some critical

event. It may then be possible to extract the counts from outside the system using a hardware monitor; alternatively the hardware may keep some performance index in a register which a software monitor could read.

There are fundamental differences between hardware and software monitors as described in following Table.

Software	Hardware
Pieces of code executed by target machine.	Attached electrically.
Specifically designed for the target machine.	Generalised hardware is available but the connection points and data reduction techniques are specific to each target machine.
Introduces new overheads to target machine begin measured.	Adds no new overheads.
Simple to rerun.	Reattaching probes or attaching to a new machine takes time and skill.
No access to some hardware measures.	Can measure most hardware components, except those implemented within a single chip.
Easily measures software resources.	Cannot realistically measure most software resources.
Less expensive.	More expensive.
Software damage may occur (software and data held on the system may be corrupted).	Hardware damage may occur.
Most of the information required is about the system software.	Most of the information required is about the system hardware.

Comparison of software and hardware monitors

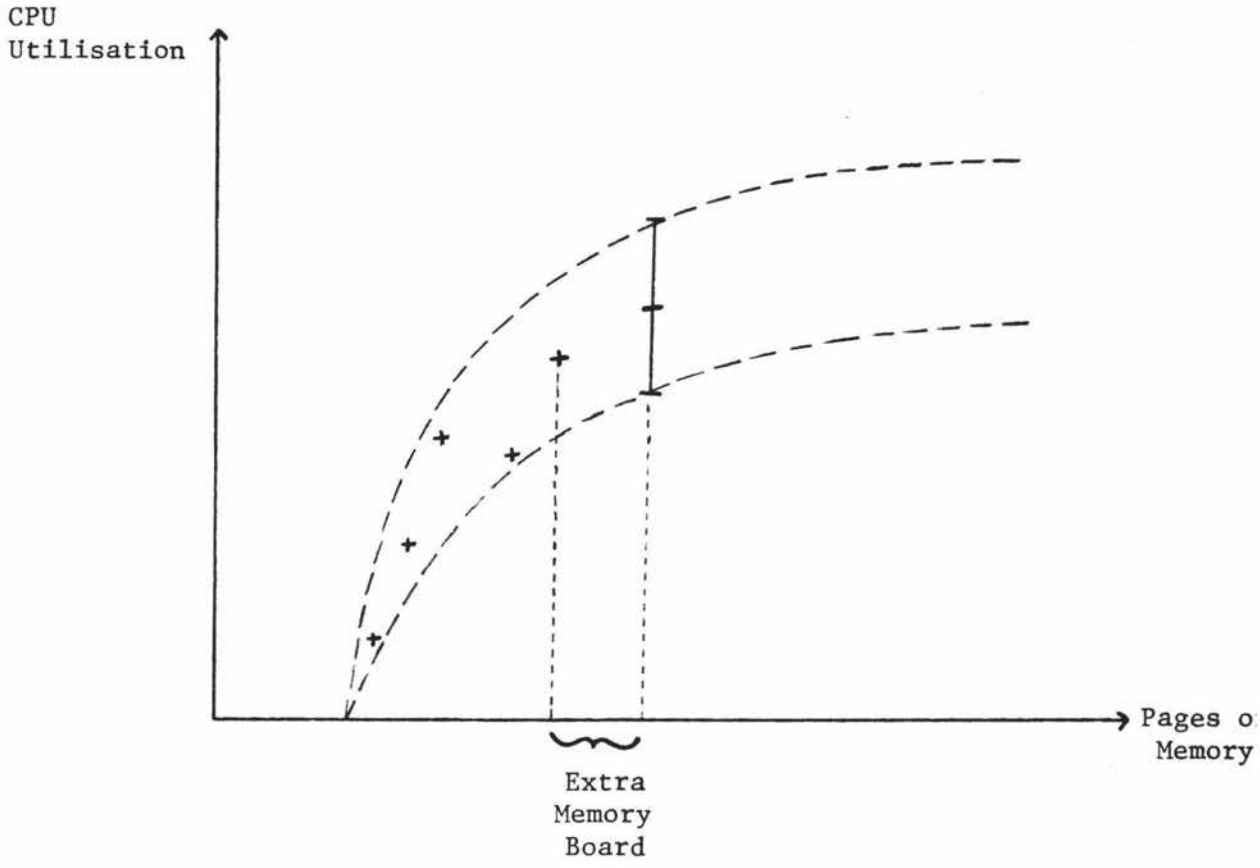
It is interesting to consider the effects that the major trends in hardware development will have on this comparison. The first trend is the increase in power of hardware. As the number of instructions executed each second increases, the percentage overhead introduced by adding a constant number of instructions (to implement a software monitor) will get lower. The second trend is for hardware to be implemented in ever fewer physical devices. As the number of cabinets, the number of boards and the number of chips all decrease there will be fewer places where a hardware monitor can be connected. If these trends continue software monitors will become more attractive at the expense of hardware monitors, although it is likely that both will remain necessary to achieve all desired measurements.

The data derived from a monitor may be used directly in making tuning or capacity planning decisions. It is increasingly more common, however, for it to be used as source data in a model of the system. For example the mean I/O time, although useful in itself, could also form part of the input data to a queueing network model, such as QNEMU [LADY84].

2.2 Models

Returning to our example system, as a result of the monitoring exercise the System Administrator at SOS finds that paging disk utilisation is extremely high at the time the system degrades, and that the CPU utilisation drops at the same time. He assumes that the

system is thrashing (paging-bound) and considers the possibility of adding some extra memory to the system. To lower the possibility of purchasing this memory and being disappointed with the results, he attempts to predict what improvement is likely to result from the extra memory. He assembles a group of typical programs to form a benchmark and runs them on the system overnight when no other users have access. During the running of the benchmark the hardware monitor is used to measure the CPU utilisation. Over successive nights he re-runs the programs with less and less memory on the machine, and enters all the results onto a graph showing the CPU utilisation against the number of pages of memory in the system. Finally he extrapolates (by eye) the trend shown on the graph by assuming a 10% ($\pm$5)% increase in available pages if he adds one extra memory board.



SOS's CPU utilisation plot
Fig 2.3

Computer systems are both complex and expensive. The complexity is increased by the non-deterministic nature of the workload processed by most interactive systems. As a result:

- o measurements are generally non-reproducible. This causes difficulties when
  - comparing machines
  - creating acceptance criteria
  - measuring the improvement in the performance of a system.

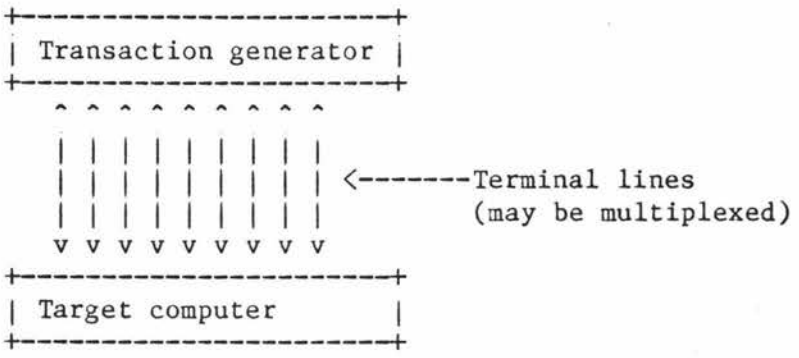
- The behaviour of the system in total is hard to interpret.

As well as these, for a large number of interactive systems,

- The CPE team cannot easily get exclusive access to the computer.

These problems can be addressed to varying degrees by experimenting on a simplified model of the system, its workload, or both.

The benchmark that the SOS System Administrator gathered together formed a model of a batch workload. There are problems with generating reproducible interactive workloads in terms of user typing speeds, fixed think times, etc. Usually interactions are modelled by a transaction generator, which is in essence a separate computer:



The validity of a benchmark is measured in terms of whether or not it is a true representation of the workload that will be run on the target system. Often a benchmark is developed from data collected on a system that already exists (possibly using a monitor). It may even be produced automatically from an existing workload as can be done on the UNIVAC 1100 [HUGH74]. Unfortunately benchmarks produced in this way make no allowance for feedback from the change in performance caused by alterations to the system (or the change in architecture between old and new systems). For example, a faster turnaround of jobs may mean that more people will be encouraged to use the system. Further, the workload could vary considerably if the new users performed different types of work from the current 'average' user. Choosing a representative benchmark is even more difficult if no measurements can be made on a system already processing a similar workload.

Further problems arise in implementing a benchmark. Relatively small changes in an implementation may result in large changes in the elapsed time for the benchmark to complete. This is particularly true for benchmarks involving job control and fourth generation languages. For example, rewriting the job control of a benchmark (but not reordering the execution of the jobs) reduced the elapsed time of a specific workload by a factor of three in a study carried out in the United Kingdom [DOCKER]. This type of effect reduces the usefulness of benchmarks for comparing different machines, if some (or all) of a benchmark must be rewritten.

The involvement of people with a vested interest or bias in the outcome of the project must also be considered carefully. If those with an interest in a particular selection being made are also to determine the nature of the benchmark, there is the possibility that they will choose a set of work which executes particularly well on the machine of their choice [HUGH77].

The high cost of developing and executing a benchmark must be carefully considered as, in many cases, it may exceed the limited benefits obtained from the use of the benchmark [BERN75].

The most powerful use of a model is to predict the performance of a computer system which is not available for direct measurement. The plot of the CPU utilisation against the number of pages of memory, given above, is a simple model of the SOS system which can be used to predict the performance of another system not yet available (namely

the system with additional memory). Models in which predictions are made by directly extrapolating observed data are called empirical models. Empirical models are covered in [SVOB76].

Back at SOS the System Administrator is not happy with the accuracy of his predictions and so he decides to study his system further. He obtains a program, form a software house, which models Prime computer systems, but with a number of simplifications that make it more manageable. Real jobs are replaced by ones in which the demands for resources are simulated using statistical distributions. The memory and CPU of the model are allocated using policies similar to those used by Primos. Jobs arrive at the system in a random manner.

Because the model makes many simplifications and assumptions the System Administrator at SOS decides to cross-check it and sets the parameters controlling the amount of memory, the number of disks etc to match those on his system and the workload parameters to match those of his benchmark. The simulator is run and the results validated against the real system. Finally the simulator is run with the amount of memory increased by the equivalent of one board.

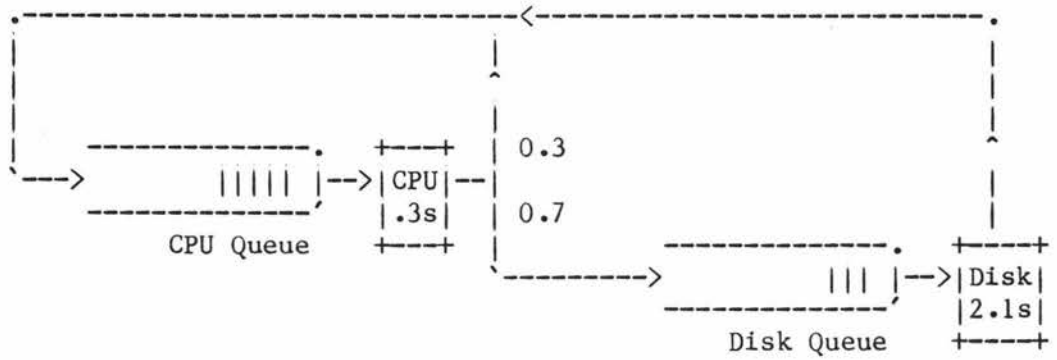
This type of model is known as a structural model and the predictions are made by simulation. Simulation is treated by most texts on computer performance evaluation; for a good introduction see [MACD70]. Because simulation is the most realistic modeling technique and because there is virtually no limit to the type of system that can be simulated, if sufficient effort is dedicated to

building the simulator, this is considered to be the most powerful CPE tool. The main limitations are given by [HIGH77] as

- o The relatively high cost of development of a simulator.
- o The lengthy time for development.
- o The critical need for validation of the model.

A computer system can be considered as a group of resources, and a number of processes which are either using or waiting to use those resources. Each resource and it associated group of waiting processes can be modelled by a queueing model.

After using each resource, there is a fixed probability that a process will request any one of the resources next. Using these routeing probability the queueing models for the individual resources within the system can be combined into a queueing network model.



Simple queueing network model

In theory queueing models can be treated mathematically if three pieces of information are known. These are [WYLE77]

- o The input traffic definition.
- o The service facility definition.
- o The waiting line definition.

and a queueing network model can be formed if, in addition to the above,

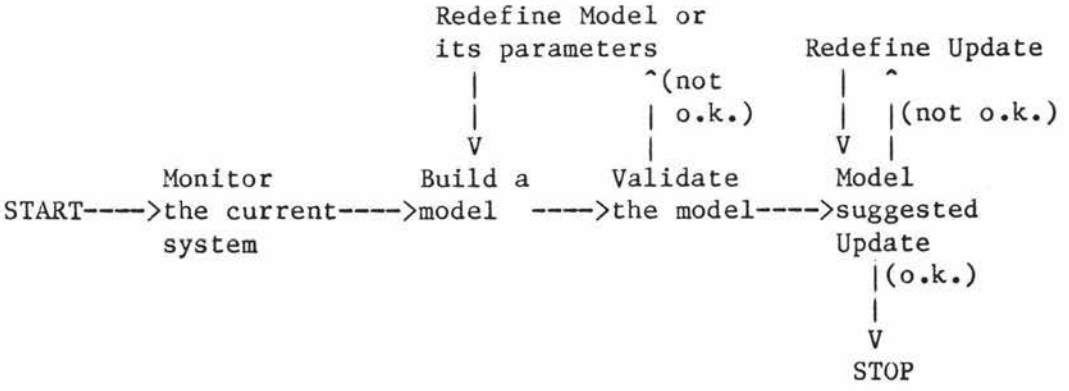
- o the routing probabilities

are known. In practice, however, the mathematics is only developed for simple cases of the above. Simplifications must be applied to real systems if they are to fit those cases.

Simulation may also be used to extract predictions from queueing network models but this is uncommon.

Fig 2.4 below combines the separate components of the SOS scenario, and many similar situations.

scenario : A capacity plan is required
to update the system in
preparation for the addition
of a major application.



Example of a CPE project
Fig 2.4

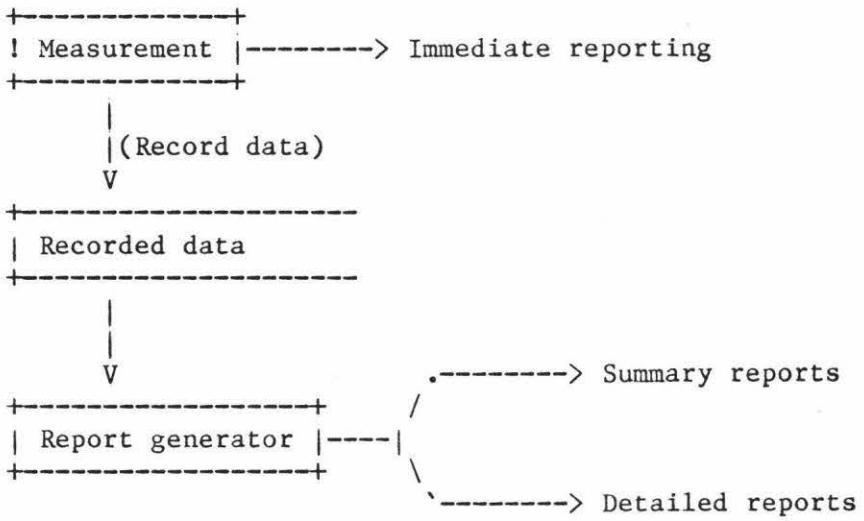
3 The principles of software monitoring

This section examines software monitors in more detail, particularly

- o their structure,
- o the indices that can be usefully measured (and how they are measured), and
- o the ways in which the measurements can be reported to the performance analyst.

3.1 The structure of software monitors

As software monitors are used on many systems for different purposes no general structure covers all the cases, however Fig 3.1 shows the basic components of most monitors. It is interesting to note that the same structure can be applied to hardware monitors.



Structure of software monitors
Fig 3.1

Some monitors (e.g., SUD for the IBM MVS operating system [FLETCH], OPT/3000 [HP2] and Mesa Spy [McDa82]) report their measurements interactively. These monitors have been designed to allow transient problems to be corrected as they occur. For example using SUD may show that some process has so much memory allocated to it that other processes are suffering. The amount of real memory allocated to it can then be reduced to free memory for other processes, and hopefully this will improve the performance of the system as a whole.

If there is no need to obtain immediate results from a monitor, several advantages can be identified to saving the data as it is collected and producing reports at a later time. Some of these advantages are:

- o The monitor does less processing during the measuring period and so there should be less distortion.

- o Results can be studied at leisure rather than in real time.

- o Different levels of report can be generated, such as:
  - summary reports of the data, and
  - detailed reports on request (covering periods of particular interest).

3.2 Performance indices

The following indices have been obtained by one or more of the monitors described in [ROSE78], [BURR1], [BURR2], [McDA82], [KOLE71], [SPERR1], [CALL75], [HP1] and [HP2]:

Number of users on the machine
Mean session length
Standard deviation of session length
Think time
Terminal response time
Transactions per hour
System availability
Data communications availability
CPU utilisation
CPU states
CPU service time
Length of CPU queue
Number and source of interrupts
Memory utilisation
Memory contents (data, program, etc.)
Memory distribution
Number of processes in memory
Channel utilisation
Channel transaction rates
Channel service time
Channel waiting time
Channel queue length

Number of words transferred on each channel
I/Os per second
Mean I/O stream length
Overlay rates
Swapping rate
Device activity rates
Device service times

In addition to these indices some software monitors also keep a trace of various events, such as

I/Os	e.g.	process identification device record number.
Page-outs	e.g.	time process identification page address (virtual or physical).
Process details	e.g.	process identification start time end time I/O time used CPU time used.

3.3 Measuring disciplines

There are two common approaches to data collection by software monitors. These are

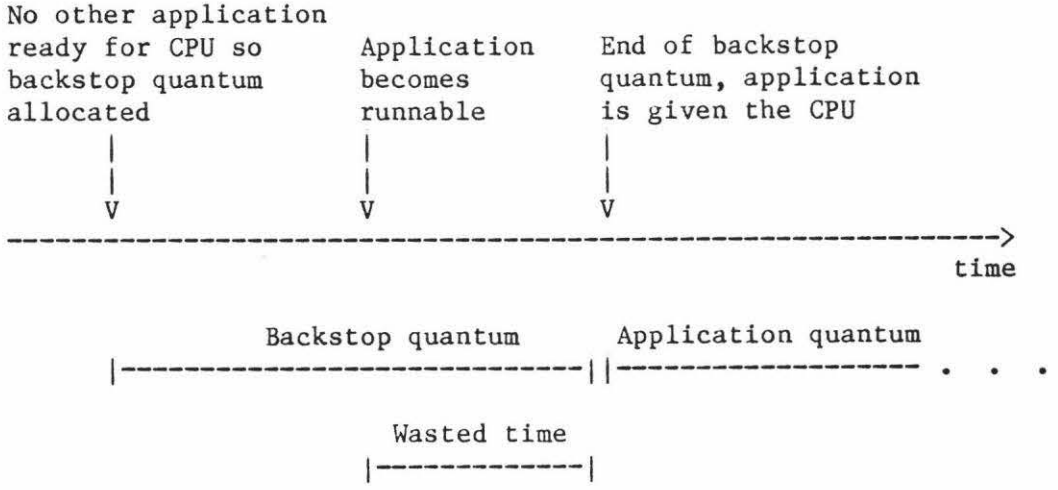
- event trapping, and
- sampling.

It is sometimes convenient to use both techniques in obtaining a single index.

A third method that is rarely used, but is sometimes mentioned in discussions on software monitoring, is the

o backstop process.

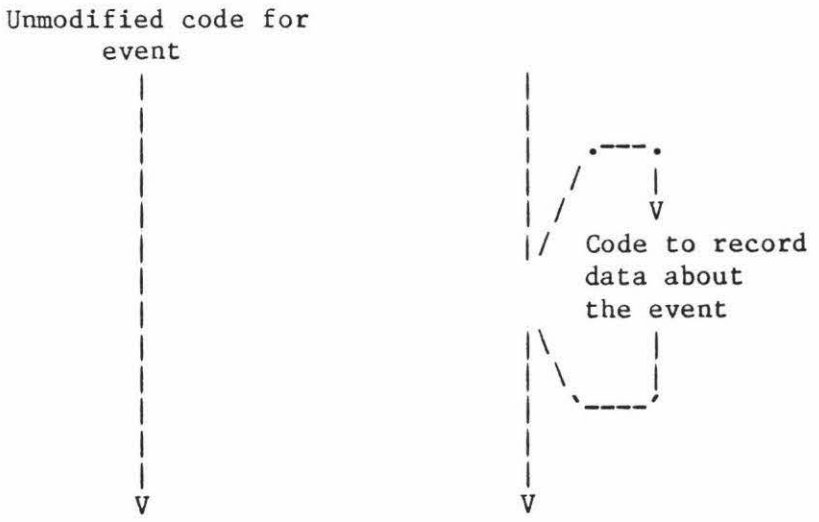
This is a process with the lowest possible priority level whose only task is to measure CPU time which would otherwise be idle. It is worth noting that most machines allocate CPU time in quantum slices, and therefore this extra process could cause an additional overhead during potentially profitable work time, as shown in Fig 3.2.



The backstop overhead
Fig 3.2

3.3.1 Event trapping

Often the index we wish to measure only changes when certain operating system events occur. If we modify the code which produces an event so that it records data on the event, then we have implemented an event trap.



Event trapping
Fig 3.3

Event traps require

- o the data to be recorded at the time of the event;
- o the code producing the event to be identified;
- o the code to be modified to record data about the event.

3.3.1.1 Collecting the data

Collecting the data may simply involve recording the fact that the event has taken place (by incrementing a counter), or it may involve retrieving data from memory or registers.

Depending on the nature of the event and the information required about it, event trapping may be used to record data about every event or, where a statistical measure is sufficient, the overhead of the monitor may be reduced by only collecting data about a fraction of the events.

The index being measured should not, in a statistical sense, be dependent upon the occurrence of the event. For example, attempting to measure the mean length of a queue by sampling its length whenever an item is added to the queue will not produce an accurate result. Rather the queue length should be sampled at fixed time intervals.

3.3.1.2 Recording the data

The data may be recorded by one of three methods:

1) Adding to an accumulator location. For example, the total active time of each disk channel may be obtained by a trap. Code is added so that at the completion of a disk transfer a register, in the disk controller, that contains the time taken to retrieve

the last item is read. This value can be added, by the trap code, to a location in memory, which will then be a running total of the amount of time the disk has been active. At some regular interval this location could be read and a derivation of the summation included in a report. This approach only applies to those cases where the data collected can be reduced to a simple summation, for example to, obtain a mean value.

2) The data collected may be written immediately to a secondary storage device and at some later time a report can be generated from the data. The cost of writing to a secondary device is normally quite high (in terms of the relative time taken) which makes this method unsuitable in many situations. If, for example, a trap that requested a disk I/O was inserted into the process exchange mechanism, the frequency at which this would be executed would degrade the performance of the machine beyond acceptable limits.

3) In an attempt to reduce the overhead of recording the data on secondary media an intermediate buffer may be used. The data is written to a main memory buffer, and another process, which runs throughout the monitoring period, collects the data from this buffer and writes it to a secondary device (or produces a report from it).

This method allows the writing to the secondary media to be at less frequent intervals, and is therefore more efficient. There is also

some flexibility introduced as to the time of the write and this may be scheduled to fit around the normal workload of the machine.

Care must be taken to ensure that the buffer does not overflow causing some data to be lost.

These data recording techniques are also used by hardware monitors.

3.3.1.3 Problems with event trapping

While event trapping is the most powerful and accurate of the software monitoring tools there are difficulties with its implementation, some of which are considered below.

3.3.1.3.1 Access

Many of the resources required to implement an event trap are difficult to obtain. In particular the source code of the operating system is required. Generally manufacturers do not make this available, but if they do they might not be willing to supply the additional tools required to carry out the necessary modification. Such tools might include a data dictionary, external documentation, and compilers for modules written in languages such as PLP (a PL/1 derivative used in Primos).

Even if all the source code and required tools are available, personnel with experience of these tools may not be available.

Getting to understand the way a complex computer system works requires considerable time and effort, and may become so costly that it outweighs the advantages of monitoring the system. Also, altering and testing an operating system requires dedicated access to the system for relatively long periods of time.

3.3.1.3.2 Overheads

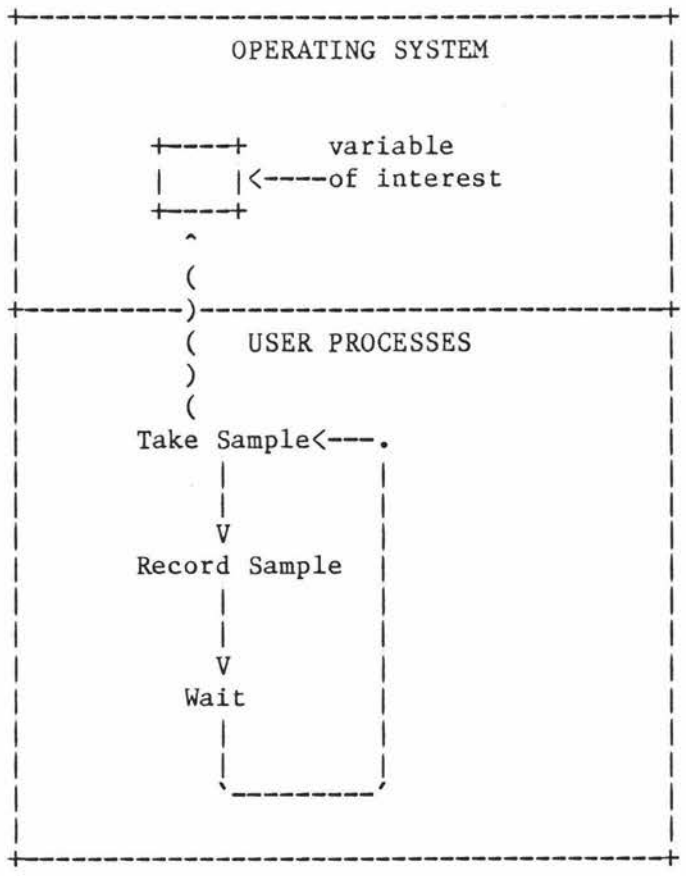
Accurately recording the extra overhead created by the insertion of code into an operating system is generally impossible. It can be estimated by using a benchmark on the two versions of the operating system, but this is subject to the availability of a good (honest) benchmark. Often the recording overhead is simply considered part of the operating system costs and is lost in that great debt.

3.3.1.3.3 Sensitivity

Lister [LIST75] defines reliability as one of the main features of a desirable operating system; however, modifying software very often reduces its reliability by introducing errors [KOPE79]. The complex nature of an operating system makes the introduction of errors more likely than alterations made to applications software. Even if the reliability is not affected the monitor may still be blamed (unfairly) for any problems which are discovered.

3.3.2 (Random) Sampling

The problems of event trapping may be avoided if a random sampling technique can be used. A monitor using random sampling may be implemented as a normal user process and so can avoid many of the difficulties associated with event trapping.



Random sampling
Fig 3.4

By random sampling we mean that the sampling function is not statistically dependent on the index being measured. The normal way

of achieving this is to sample at a regular time interval; however we will also discuss the following alternatives:

- o Weighted sampling.
- o Loosely coupled sampling.

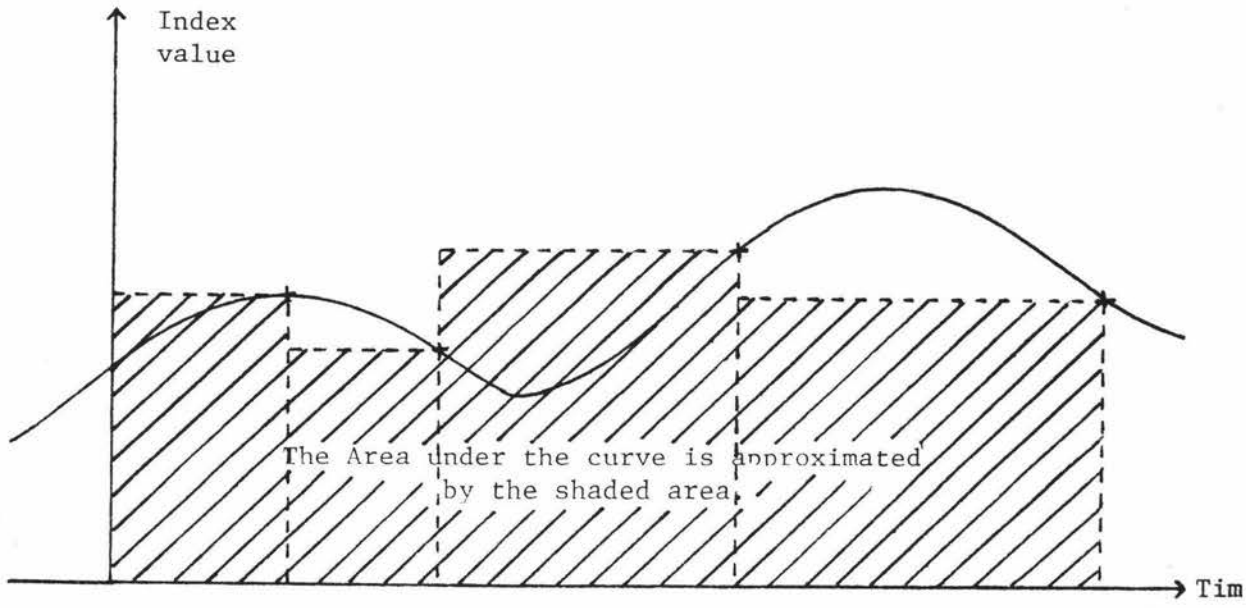
3.3.2.1 Collecting the data

With random samples we do not have the choice of taking a sample every time the variable changes and thus cannot get a complete history of the index in question. We are unable, for example, to measure the time interval between a page being removed from main memory and it being recalled. To do that we would need data (the current time) to be collected at specified events (the page being rolled out and then being accessed again). We can however get a good measure of the mean value of an index, and the way it varies from this mean, provided sufficient samples are taken.

It is necessary to sample in a manner that is not statistically dependent on the index we wish to measure. For example, if we were measuring the length of the queue of processes waiting to use the CPU, but the monitor took fewer samples as the queue lengthened, we would get fewer samples at the higher values of the index. This would not give an accurate measure of the mean length of the queue. Unfortunatly regular sampling is often very difficult to arrange,

particularly from outside of the operating system. To give to users the ability to decide when they should receive service conflicts with the idea that the operating system should schedule resources in an optimum manner [LIST75].

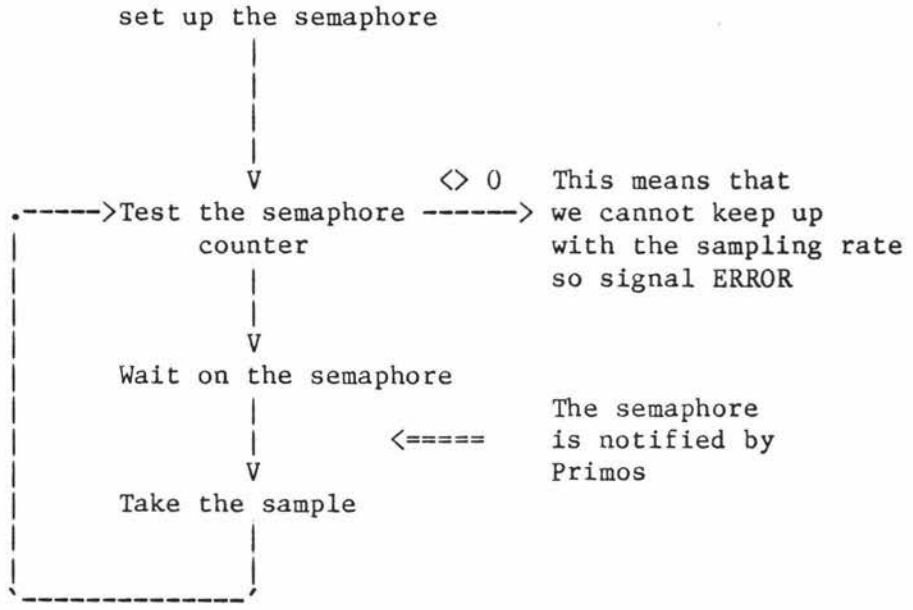
There is no rigorous requirement to maintain an exact interval between samples. If the time at which each sample is taken is recorded then a weighting system may be used in which the value of each sample is multiplied by the time since the last sample was taken.



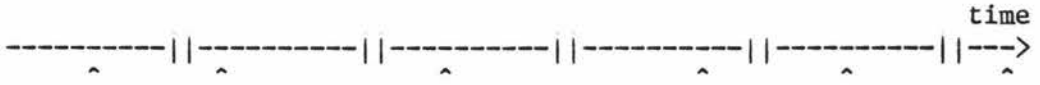
Weighting the samples

Fig 3.5

A second way to avoid the necessity of sampling at fixed times is to use a loosely coupled sampling technique. Under some operating systems it is possible to arrange for the sample to be taken at an unspecified time in a given interval. Primos, for example, supports a mechanism known as a timed notify. A timed notify can be arranged so that the operating system regularly signals a user-accessible semaphore. Fig 3.6 shows how this mechanism can be used to ensure that the number of samples taken over a period of time is constant.



The sampling methodology



|| Semaphore signalled by the operating system.

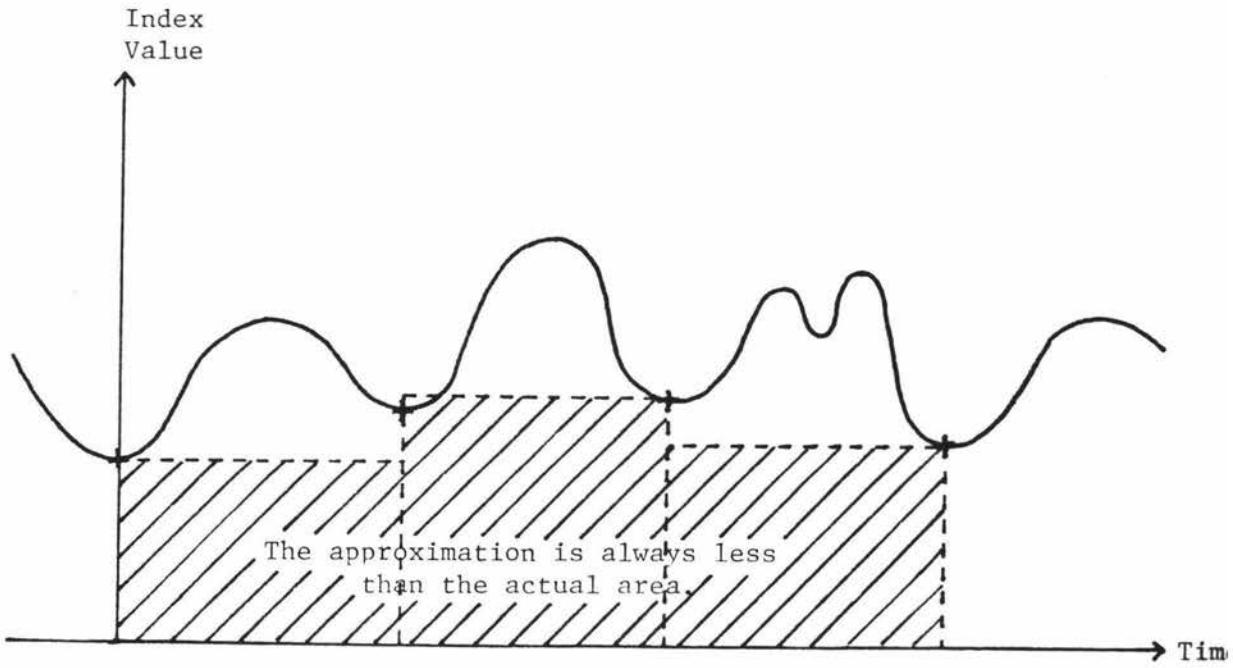
^ Sample taken.

Example of sample timings

A loosely coupled sampling function
Fig 3.6

Both the non-regular sampling methodologies we have presented rely on the underlying assumption that the variables we are sampling behave randomly between samples. For indices measuring the CPU this may not be the case. It may be more common for the sample to

be taken when the variable changes in a certain way. For example if the CPU utilisation is being sampled, it is more likely that the sample will be taken when this has just fallen, rather than when it has just risen. This can lead to distortions, as illustrated in Fig 3.7.



Microscopic distortions

Fig 3.7

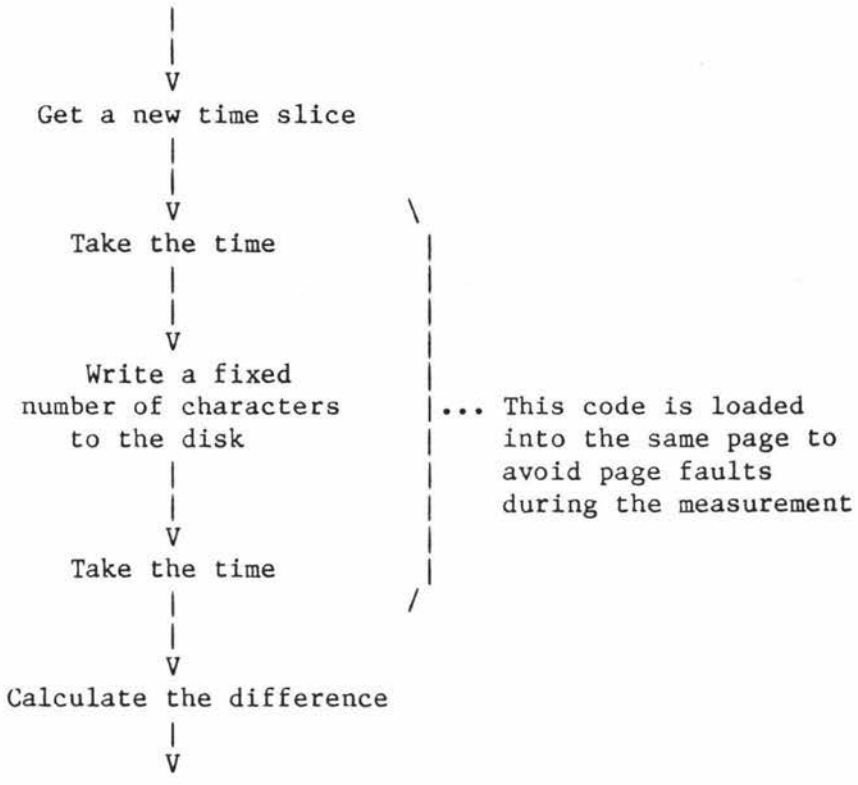
An investigation into the extent of this type of distortion is currently in progress. Present indications, from monitors which

use these techniques, are that this is not the cause of errors that are significant when compared with the accuracy of the monitor.

3.3.2.1.1 Collecting data by unit workloading

In chapter 5 we introduce a monitor that makes use of a variant of the random sampling technique called unit workloading. This is suggested, within this thesis, as an alternative approach to data collection that avoids several of the problems of the usual methods. As far as we can tell this technique has not been used elsewhere.

With unit workloading the sample values are not a direct measure of any resource in the system, but are the times taken to perform some small operation. For example, a measure of how busy a disk drive is can be found from running a small benchmark arranged (as in Fig 3.8) so that the main resource required during the timing period is the disk.



A disk mini-benchmark
Fig 3.8

3.3.2.1.1.1 Advantages of unit workloads

The advantages of this method of gathering data include the following:

- It allows any resource to which the programmer has access to be measured by random sampling.

- In chapter 2 we discussed the tendency for CPE tools to be specific to a particular system. Monitors implemented using unit workloading do not need to be closely tied to

the structure of the underlying system because they make indirect measurements. Most of the code for such a monitor could be used directly in the implementation a similar monitor on another machine, that supported the same programming language(s).

3.3.2.1.1.2 Disadvantage of unit workloads

The main disadvantage is the level of indirection introduced by not measuring the index directly. This requires the monitor to be calibrated and the results quoted as relative to this calibration. The execution time for the benchmark when it is the only user process on the system is a useful baseline for calibration.

3.3.2.2 Recording the data

The same options are available for recording the data in random sampling as in event trapping. However, there is not the same emphasis on recording the data quickly as there was for the event trapping mechanism, because, as a normal user, we are not likely to alter any time critical system function, and also the overheads can be reduced by decreasing the sampling frequency.

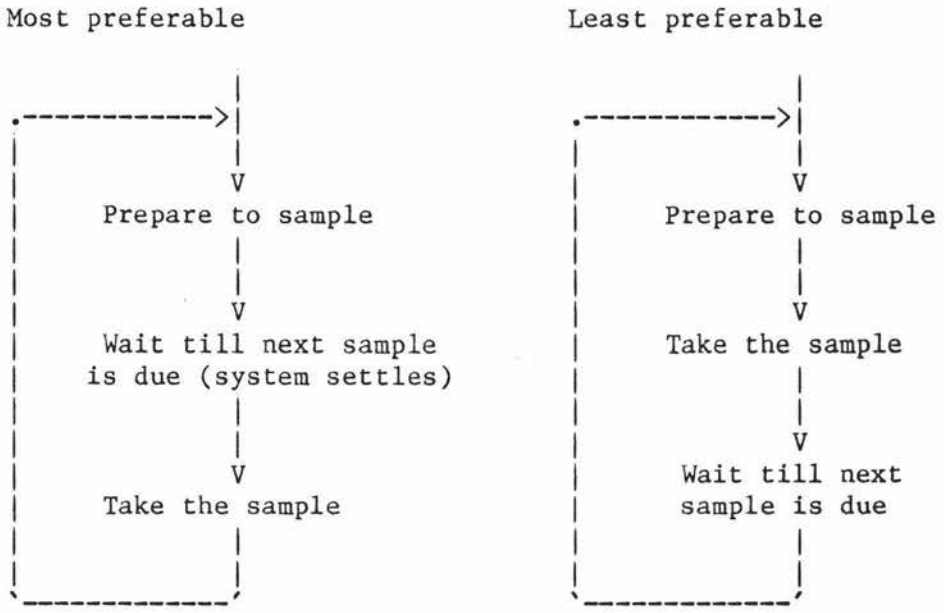
3.3.2.3 Advantages of random sampling

Apart from overcoming the specific disadvantages mentioned in the event trapping section, random sampling has the following advantages:

- o Distortions can be minimised.
- o Installation is simpler.

3.3.2.3.1 Reducing distortion

Distortions introduced by the presence of the monitor may be minimised by arranging to take the sample at an opportune moment. For example if all the preparation required to take a sample is performed before the monitor is suspended then the sample may be taken immediately after the pause. At this time the effects of the distortion due to the presence of the monitor will be smallest, as it will not have made any recent requests for resources (other than the CPU, which it must have when the sample is taken).



Reducing distortion
Fig 3.9

If the operating system is not modified then there is no chance that we will accidentally make it less efficient. Because the operating system is a very active piece of software small errors can have a large affect on system performance. A simple error could, for example, invalidate the disk scheduling algorithm and cause a major performance degradation.

3.3.2.3.2 Simplicity of installation

The monitor is just another user program. It may be added to the system (or removed from it) without any disruption to other users. When it is not in use there is no residual overhead.

3.3.2.4 Problems with random sampling

The problems with random sampling are:

- Only those indices that can be treated statistically can be measured. We can measure queue lengths and device utilisations but cannot measure event counts or event type proportions.

- There is normally a high memory overhead in introducing a new process into the system.

- A random sampling monitor must contain more code than an event trap, as it must have its own control structure to take the sample and then pause for the appropriate time.

3.4 Reporting the results of monitoring

The way the output of a monitor is presented can greatly affect its usefulness. If too much data is presented the user may be swamped with detail, but if too little is presented then important results could be hidden. Two approaches are often used to minimise these problems:

- o multilevel reports.

- o Data presentation techniques.

In section 3.1 we noted that if data reduction is left until after the measurement period, several passes over the recorded data can be made. Initially a summary report could be generated, and if there is a need for detailed reports then these could subsequently be produced for the devices or time periods of interest.

The survey of monitoring projects mentioned in section 3.2 identified a number of ways measurement data could be reported, namely:

- o Quoting the indices (as a number)
- o Tables
- o Histograms
- o Graphs
- o Gantt profiles
- o Kiviat graphs

3.4.1 Quoting the indices

Quoting the value of the index is the simplest form of data presentation. For example

The paging rate = 12.3 pages/sec.

3.4.2 Tables

If a number of indices are to be presented a table can be used. The following example is taken from the Burroughs System Managements Facility (II) manual [BURR2]:

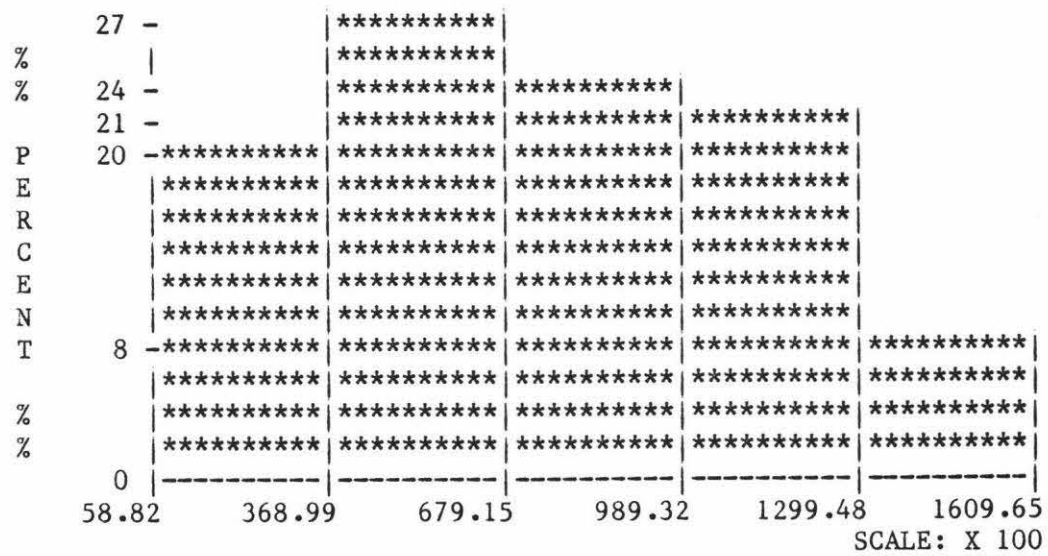
B6800 SMFII ANALYSIS
TASKIOTIME LSS 2000
SORTED BY ITEM 3 (ASCENDING)

USERCODE	MEAN(TASKPROCTIME)	MIN(TASKELAPSEDTIME)
TSG	13.23	0
.	401.73	3
GAS	1.60	3
SPUR68PROD	8.95	5
ADG	2.63	8
LIT	2.81	15
PUBS	21.50	18
AVI	0.71	155
INVENTORY	4.00	327
SPUR68DEV	5.86	346
SECRETARY	2.67	512
SHOOK	3.00	524
SUPERVISOR	9.08	2515

3.4.3 Histograms

A more visual, but generally less accurate, approach to presenting a set of related indices is to use a histogram. The following diagram is also taken from SMFII [BURR2].

B6800 SMFII ANALYSIS
TIME RANGE: 18:06:52 09/18/80 - 03:57:56 09/19/80
(*) RELATIVE HISTOGRAM OF AVAILMEM

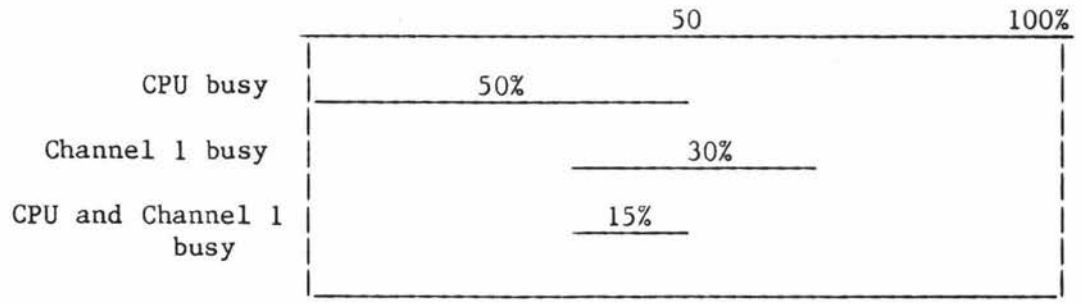


<--- VALUES OF AVAILABLE MEMORY --->

3.4.4 Gantt profiles

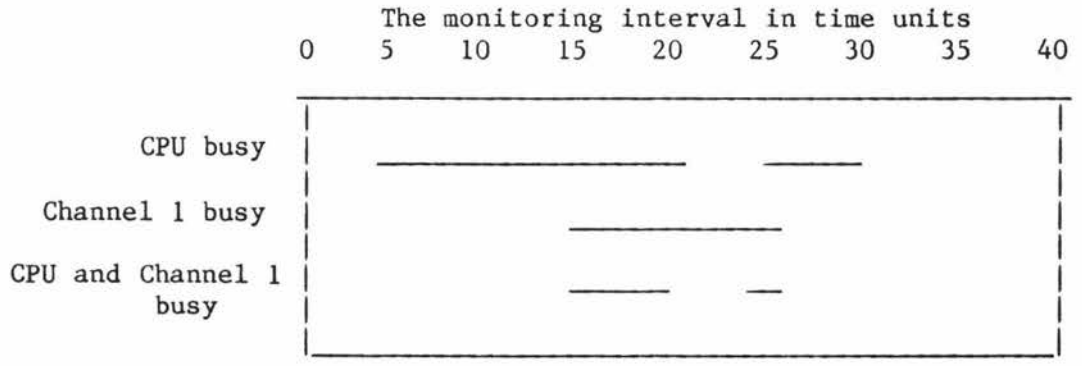
Two variants of the Gantt Chart are discussed in CPE literature. We distinguish between them by the names Utilisation Profile [FERR78], and utilisation Time Profile.

A Utilisation Profile gives a time independent view of resource utilisations, including overlaps. For example, the following profile indicates that the CPU was busy for 50% of the measurement period, Channel 1 was busy for 30%, and both Channel 1 and the CPU were busy at the same time for 15% of the period.



Example utilisation profile
Fig 3.10

A Utilisation Time Profile is a timing diagram showing the times the resources were being used. The following is the time dependent equivalent of Figure 3.10.

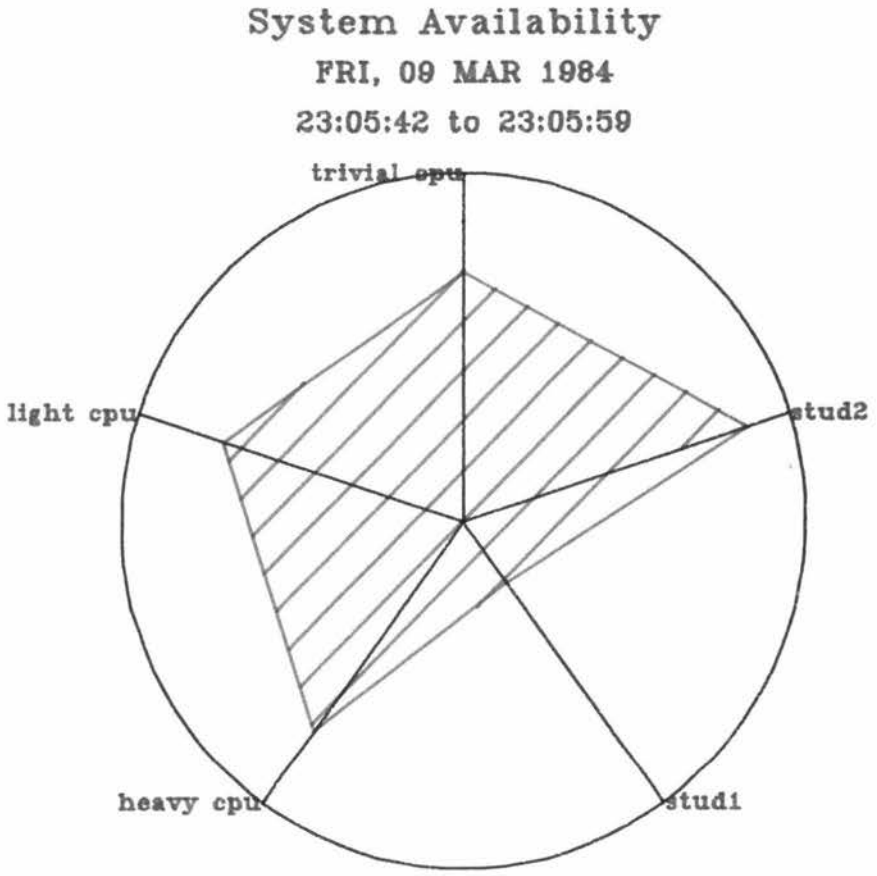


Example utilisation time profile
Fig 3.11

3.4.5 Kiviat graphs

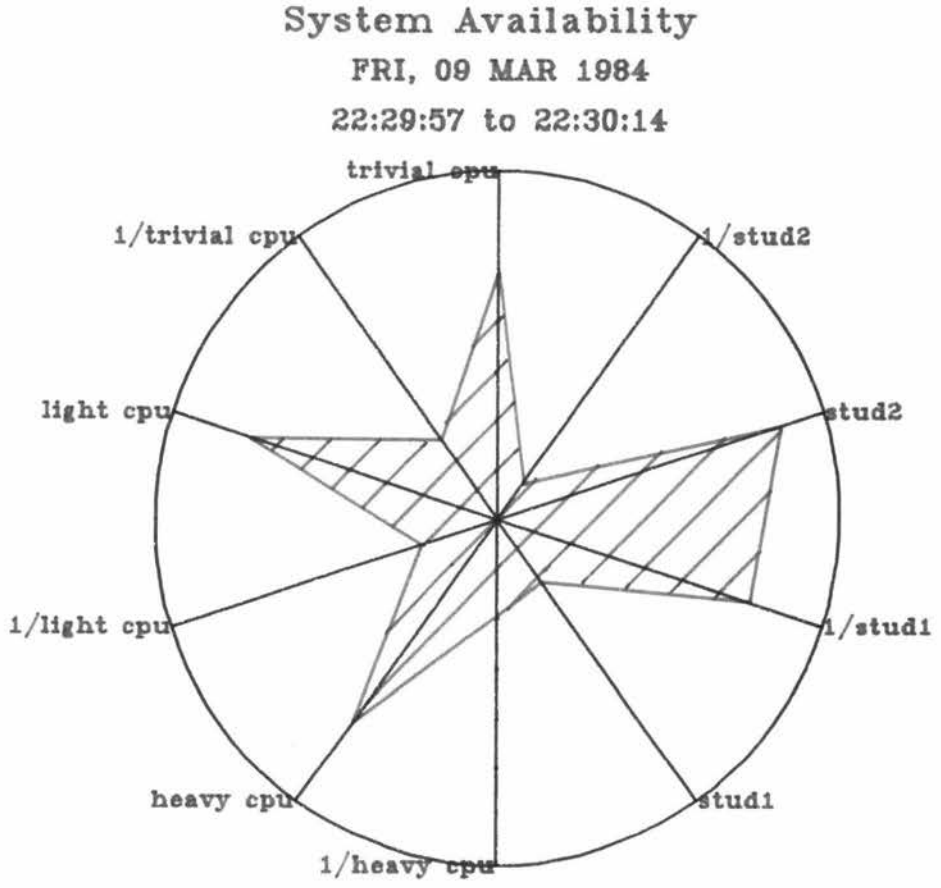
A Kiviat graph is a pictorial method of displaying more than two system indices on a single diagram. The diagram is arranged around a point, with an equally spaced radial axis for each index. The index is plotted as a point on its axis and points on successive axes are joined with line segments.

Variants exist on the basic chart, depending on the discipline (if any) for choosing what index to display on each axis. The two major variants are described by Ferrari [FERR83] as Noe's version (fig 3.12), where no discipline is used to choose the axes, and Kent's version (fig 3.13), where successive axes plot positive then negative indices. In the first case we use only positive indices (ones which get bigger as performance improves), and the bigger the area contained by the plot the better the system is performing; a well balanced system will have a regular polygon (all angles the same). With Kent's version the plot of a system which is performing well will produce a star shape. A well balanced system will have the length of each arm of the star the same.



Example of Noe's version Kiviat graph

Fig 3.12



Example of Kent's version Kiviat graph

Fig 3.13

4 Monitoring the Prime P750

In chapter 3 we listed the following performance indices obtained by one or more of a number of software monitors reported in the literature:

Number of users on the machine
Mean session length
Standard deviation of session length
Think time
Terminal response time
Transactions per hour
System availability
Data communications availability
CPU utilisation
CPU states
CPU service time
Length of CPU queue
Number and source of interrupts
Memory utilisation
Memory contents (data, program, etc.)
Memory distribution
Number of processes in memory
Channel utilisation
Channel transaction rates
Channel service time
Channel waiting time
Channel queue length
Number of words transferred on each channel
I/Os per second
Mean I/O stream length
Overlay rates
Swapping rate
Device activity rates
Device service times

and event traces of

I/Os
Page-outs
Process details

In this chapter we address the feasibility of measuring these indices on the Prime P750.

The operating system for the Prime P750 is Primos. As we will be discussing measuring, interfacing and analysing Primos, an overview of the operating system and the related features of the P750 hardware is given in Appendix A.

4.1 Indices that could be measured

Primos lends itself to the measuring of several of the indices on the list.

4.1.1 Number of users

The number of users logged onto the system can be found by using an operating system routine provided for this purpose. USER$ gives a count of all users logged onto the system, including those from other machines and phantom users (a phantom is identical to a normal user except that input must be read from a file and not a terminal).

A useful time to take a count of the number of users would be when any user logs on or logs off. This can be done using the external logon mechanism which allows the System Administrator to run programs he has supplied at logon and logoff. This is normally used by the systems staff to implement an accounting system, however programs could also be written to record the number of users and the

time.

4.1.2 Session length

If the external logon/logoff programs recorded the user number and the time during logon and logoff, this data could later be analysed, and the difference between logon and logoff time for each session calculated and accumulated into a mean.

4.1.3 Think time

Ferrari [FERR83] defines the think time as " ... the time between the end of a command's processing by the central subsystem and the end of the inputting of the next command by the user."

A common Primos routine (C1IN) is used for all terminal input and this routine could be patched to record data about the length of time between the system requesting a command and the user typing one in. This routine, like DISKIO, has a high usage and so the data should be recorded carefully. Probably the most useful information is the mean think time and this could be found using two accumulator locations which held, respectively,

- the number of commands read;
- the total delay between requesting commands and their being entered.

The mean think time for all terminal inputs since the system was booted could be calculated at any time from these totals. The mean think time over an interval could be calculated by taking the difference between the two totals for the beginning and end of the time period.

The think time as described above, assumes that the CPU time required to accept the individual characters of a command is negligible. A more detailed analysis could be done if two following types of think time were identified:

- o the time between the completion of one command and the user entering the first character of the next command, and

- o the time between successive characters in the command being entered.

These could form part of the source data to a queueing network model with two classes of job.

4.1.4 Terminal response time

There is no single definition of response time and with appropriate patches a number of the possible response times could be measured. For example, the time from the user typing a character to the time the first response character is output could be measured, by patching the terminal input and output routines. As with other

measurements, consideration needs to be given to recording this data without creating very much overhead, as the monitoring code will be relativity heavily utilised. As with think times the mean would be the most useful statistic.

4.1.5 Transactions per hour

The precise nature of a transaction is also unclear and it would need to be defined in Primos terms before it could be measured.

4.1.6 System availability

The down time for the system could be measured by several different methods. For example, if the current date and time were periodically written to the first record of a file, then the last recorded time could be compared with the time when the system is booted (after a crash) and the difference calculated. The accuracy of this method would depend on the frequency at which the time was recorded.

4.1.7 Data communications availability

Primos can support the three types of network:

- Ringnet,

o Point to point, and

o Public data network,

and so the nature of a detailed report concerning data communications availability would depend on the configuration that existed use in the installation under consideration.

It should be possible to record the time for which a connection to each of the other machines in the network could be made. One way for a software monitor to do this would be to execute the Primos command 'status network' at a regular interval and record the results and the current time. This could be analysed later to produce a table giving the percentage of the measurement period that each remote system was available. The amount of quantisation error involved would depend on the sampling frequency.

4.1.8 CPU measures

On the P750 the dispatcher is implemented in firmware. A dispatch of the CPU is initiated by a range of machine level instructions which include the semaphore operations WAIT and SIGNAL. As there is no single routine concerned with dispatching the CPU, it is difficult for a software monitor to extract CPU data not already collected by the hardware.

Provision is made to record how much CPU time is used by each processes. To do this the hardware regularly increments a register in the processor's register file. This register is swapped, along with the user accessible registers, by the process exchange mechanism (process exchange is described more fully in Appendix A). Primos accumulates this time into a total CPU time for each process, consequently it is not too difficult to proportion the total CPU time used between processes.

In particular the amount of time used by the backstop process can be found. This is roughly equal to the amount of time the CPU would be idle if the backstop did not exist. (It is only roughly correct as the backstop does do some useful work (see Appendix A, section 5.3).

The length of the queues used for scheduling the CPU can be measured by reading the count from the semaphore headers as discussed in section 4.1.8.1.

Other data about the CPU, such as the CPU states and CPU service time, is not readily available to a software monitor.

4.1.8.1 The length of the CPU queue

There are several lists on which processes that wish to use the CPU are suspended; there is no single CPU queue. In the early stages the process waits on system semaphores known as wait-lists. Each of these semaphores contains a count of the number of waiting

processes (see Appendix A, section 5.2), and this count could be read and recorded by a software monitor. In the later stages the process waits on the ready-list. No count of the waiting processes is kept in the ready-list structure (Appendix A section 5.3.1), but a software monitor could chain through, and count, the processes waiting on each of the sub-lists.

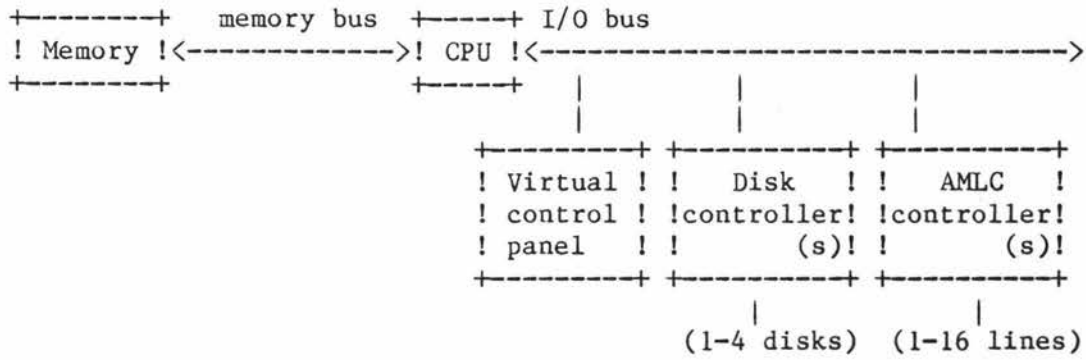
4.1.9 Memory contents

It is not possible to determine the contents (data / program / linkage) of a segment from its virtual address, or from other data available in the system, as the programmer may allocate segments however he wishes. It is however possible to tell, from the segment number, whether the segment is being used for the operating system, other shared utilities (and on any one system the address number will indicate the utility), or private user memory. This is because under Primos segments 1-1777(octal) are reserved for the operating system, segments 2000-3777(octal) are reserved for shared programs other than the operating system (the System Administrator defines these and the addresses which they occupy), and segments 4000-7777(octal) form the user's private (non-shared) address space. This division of the address space is discussed further in Appendix A section 3.1.

A useful measure could be obtained by periodically working through the virtual to physical mapping to produce a breakdown of the percentage of physical memory used in each of these areas.

4.1.10 Channel measure

The exact nature of a channel on the P750 is clouded by the fact that all I/O (including that know as Direct Memory Transfer (DMX)) passes through the central processor. This is implemented by way of a micro-code branch which transfers the data from the I/O bus to the memory bus. The resources involved with disk I/O transactions (most of which use one of the four types of direct memory transfer) are the I/O bus, the cache, the memory bus, the disk controllers and the disks themselves (see Fig 4.1.)



Structure of P750 I/O architecture
Fig 4.1

A software monitor has no direct way of measuring the utilisation of the memory bus, the cache, the I/O bus or the disk controller(s) The time that each disk is active is collected by Primos, and this could be read at the start and end of an interval during which time the utilisation of the disk was sought. An indirect measure of the whole disk I/O system could be obtained using a disk unit workload, as discussed in chapter 3.

4.1.11 I/Os per second and page faults per second

Primos maintains a separate I/O count for each disk and for page faults. The number of I/Os or page faults per second could be found by calculating the variation in this counter over a known interval and dividing by the length of the interval in seconds.

4.1.12 I/O trace

Primos has a common routine (DISKIO) for handling disk I/Os. This routine could be patched to record the details of each I/O. However difficulties could arise when recording the data because DISKIO is

- o heavily used, and
- o cannot be called recursively

which means that the recording code would have to:

- o minimise the data saved for each I/O, and
- o use an intermediate buffer that was read by some other process.

4.1.13 Page out trace

Page outs could be trapped in DISKIO or in one of the paging routines. Similar considerations as for an I/O trace would need to be given to recording the data in an efficient manner. (described in section 4.1.12).

4.2 Inappropriate measures

Because of the way Primos has been implemented some of the indices listed are not appropriate.

4.2.1 Memory utilisation

Primos uses a system-wide least recently used paging mechanism. One result of this is that, except under very light loading with no page turning, the memory will always be full. Memory utilisation is therefore not a useful measure.

4.2.2 Number of processes in memory

The normally accepted concept of a process cannot be applied to Primos. The Primos equivalent can be thought of as a single user virtual machine. The process may be busy, in which case it is serving a user, or idle, when it is waiting for a user to logon.

All processes are potentially interactive (including phantoms which are scheduled for the CPU just like any other user), and are capable of executing only one task at a time. That task does not necessarily terminate, and therefore memory is not deallocated when Primos command-mode is re-entered after completion of some command (or at any other time determined by the system, except at logout time). There is a static number of processes in the system, and when a user logs off the process remains waiting for a terminal input. Eventually another user will log on and the same process will execute his commands. Each one of these processes has some main memory allocated to it, and so there can be no concept of the number of processes using main memory except the maximum number of configured users.

4.3 Difficult measures

4.3.1 Number and source of interrupts

To capture this data with a software monitor it would be necessary to trap each interrupt. On the P750 the address of the interrupt routine is supplied as part of the interrupt code received from the interrupting device. There are only a few bytes of code between each of these addresses so it is very difficult to patch these routines, known as the phantom interrupt code, directly. This problem is compounded by the quick response that is required as interrupts occur at least every 3 milliseconds. The variability of

action and the number of routines involved makes it infeasible to patch more than a few of the secondary routines called by the phantom interrupt code, which are the interrupt handlers themselves. If follows that we could gather data about a few particular types of interrupt but not about all interrupts. As a resualt we could, for example, measure the number of interrupts received from the terminal drivers but we could not measure the total number of interrupts.

5 Primon

In chapter 1 we introduced the need for a software monitor for the Prime 50 series of computers. The following groups were identified as showing an interest in the development of a such monitor:

- The Massey Universities Computer Centre, and other computer centres, both within and outside universities.

- The Massey University Computer Science Department, who want a tool to demonstrate software monitoring to students.

- The Performance Evaluation Group within the Computer Science Department at Massey University who would like a tool to provide input statistics for computer system models.

In chapters 3 and 4 we formed a list of indices that could be usefully measured on the 50 series running under the Primos operating system, namely:

Number of users on the machine
Mean session length
Standard deviation of session length
Think time
Terminal response time
System availability
Data communications availability
CPU utilisation
Length of CPU queues
Memory contents (data, program, etc.)
Disk system availability
I/Os per second
Paging rate

and event traces of

I/Os
Page-outs

The limited time and computing resources available have meant that it has not been possible to implement a monitoring package that measures all of these indices.

We decided to begin by implementing monitors that fulfilled the following criteria, which are in order of importance:

- To use the software monitoring techniques first presented within this theses; to demonstrate their use and verify that they are in fact valid techniques.

- To maximise the usefulness of the package as a teaching tool by covering as wide a range of the standard software monitoring techniques presented in chapter 3.

- To measure indices useful to CPE personnel.

5.1 Features

So far Primon consists of four monitors. These are, respectively, designed to

- Determine the mean length of the various semaphore queues within the system in the short (interactive) term. This monitor collects data by random sampling and reports the data via tables.

- Determine the mean length of the various semaphore queues within the system in the long term (over several days). This monitor collects data using weighted random sampling and produces multi-level reports, formatted as graphs, after the completion of the measurement period.

- Provide a breakdown of disk I/Os, by device, spread across device and by calling routine. This monitor collects data by event trapping and records it via an intermediate buffer in main memory. Reports are formatted using tables and histograms.

- Measure the relative availability of the system to service each user in terms of very small, small and large CPU operations, and in terms of disk operations on each of the attached disks. This monitor collects data by loosely coupled random sampling using unit workloading and reports the data

via tables and Kiviat graphs.

5.2 Development

Primon was developed on the Prime P750 used for teaching undergraduates at Massey University. Several difficulties were faced during the development of Primon, including

- o Primos documentation.

  The documentation available for Primos, [PRIM1], [PRIM2], [PRIM3] and [PRIM4], is largely at the overview level and the single largest problem encountered while writing Primon was gaining detailed information about Primos. Often it was necessary to study the source code to obtain the details required. Much of the operating system code (particularly the older Fortran and assembler modules) has a complex structure and is very difficult to understand. As a result the process of getting detailed information has taken in excess of 80% of the time spent on this research.

- o Access to the machine.

  Dedicated use of the machine was, necessarily, limited to times when it was not available for student use, and when there were Computer Centre staff available for supervision. Because of these conflicting requirements only a little time could be scheduled to install, test and debug operating system

patches. In total this amounted to four five-hour sessions. As a result most of the monitors implemented have had to work from outside of the operating system.

o Changes in the operating system.

In the early stages of the development of Primon the machine ran a pre-release of Primos Rev19.0. In March 1983 Rev19.1 was introduced, and then later in the year Revs 19.1.1, 19.1.2, 19.1.4 and then 19.1.1 again were run on the system. This changing environment has required the package to be as insensitive to operating system changes as possible.

o Operating system problems.

The first attempt to implement an event trap failed because it uncovered a latent operating system bug in the pre-release version of Rev 19.0. It was eventually discovered that the error lay with the failure to ensure a pointer was positioned on an even word boundary. The next release of the operating system (Rev 19.1) was received shortly after this discovery and the same patch was succesfully introduced.

While incorporating the Massey patches to Rev 19.1.1 Computer Centre staff inadvertently omitted the ring 3 gate from the patches. This was rectified with Rev 19.1.4, but unfortunately tape driver problems with that release of Primos required that the system revert to Rev 19.1.1. By this stage the source code for the Computer Centre patches to Rev 19.1.1

had been deleted and the original entry point omission could not be corrected.

o The lack of operating system development tools.

The following tools were available:

- Operating system compiler.

  This program searches the operating system directories for modules that have been changed and compiles them with the appropriate compiler and compiler options. When only one or two modules are changed it is quicker to compile them by hand and consequently we did not use this tool.

- Operating system move.

  This program moves the relevant run files from one directory to another. Improvements to the copy statements at Rev19 have made this simple to do manually, which we found preferable, as it did not require familiarity with the defaults and options of the move program.

- Operating system loader.

  This program loads the operating system modules into the run files from which the system is booted. We used it frequently.

There is no data dictionary for Primos and no tools to assist with debugging or cross-referencing the operating system.

The lack of a cross-referencing tool was particularly troublesome as there was no way to answer questions like "In what modules is this routine used?". There are 925 object modules in Primos Rev 19.1 and this problem became so limiting that a cross-reference program (XREF) was developed.

5.3 XREF

XREF makes use of data saved in all binary files on the Prime that describe the external items referenced. This information is intended to direct the loader to reserve the appropriate areas for external variables and to link the external routines together, but it is also available for debugging purposes through the load maps produced by the loader (SEG).

XREF gets access to this data by calling SEG to load and produce a map for each of the modules in a directory of binary files, one at a time. The maps are cross-referenced to produce the following three reports:

- The external calls list
  which identifies the modules from which each of the external routines is called.

- The external variables list

  which identifies the routines where each external variable or Fortran common area is referenced.

- The definition module

  which identifies the name of the file where each routine is defined.

A fourth report may optionally be produced on semaphore locations:

- The semaphores used by the operating system

  are found by watching the reverse pointers in the Process Control Blocks to see what semaphore each process is actually waiting on.

To be able to call the loader program (SEG) and the editor (ED) XREF is written in CPL. A Pascal routine is used to cross-reference the maps.

5.4 The monitors

Currently Primon consists of 4 individual monitors linked together by a common menu program. The monitors are:

- o the interactive semaphore queue length monitor;
- o the batch semaphore queue length monitor;
- o the disk activity monitor;
- o the relative availability monitor.

This chapter does not consider the full details of how Primon is implemented and how it can be used (these are contained in [MCGR84a], "The Primon User Guide" and [MCGR84b], "The Primon System Documentation"), but concentrates on points of interest, particularly those mentioned in earlier chapters.

5.4.1 Interactive queue length monitor

The major features of this monitor are:

- o random sampling;
- o interactive reporting using tables;
- o pausing before sampling;
- o tabular reports.

5.4.1.1 Purpose

The purpose of this monitor is to identify any bottlenecks, at the time they occur, by displaying the current length of the semaphore queues at the user's terminal.

5.4.1.2 Background

As described in Appendix A, section 5.2, semaphores contain a two word header. The first word is a count of the number of processes waiting on the semaphore queue and the second word is a pointer to the PCB for the first process on the list. Each PCB contains a link to the next PCB on the list and a pointer back to the

semaphore header. The header, the queue and the return pointers are all maintained by the hardware on the execution of the WAIT and SIGNAL instructions.

5.4.1.3 Implementation

By scanning the return pointers in the PCBs we form a list of the semaphores that processes are waiting on. Once we have formed this list the number of processes waiting on each semaphore is read from the semaphore headers.

The monitor repeatedly scans the PCBs for any new semaphores, adding any that are found to a balanced binary tree of semaphores already found. The monitor then examines each semaphore in the tree and adds its current length into an aggregate. After N repetitions (where N is set by the user) the mean semaphore queue lengths are displayed at the terminal.

When monitoring is complete the list of semaphores found is saved into a file, so that next time the monitor is used these semaphores will already be known. This means that the monitor will spend less time adding semaphores to the tree during the initial stages of the monitoring period.

```
Threshold >0.00 Total qs 113 New  qs  37 Samples   1 Change 100.0%
Error 4.7%

CLKSEM-> 1.00 SLCSEM-> 1.00 AMLSEM-> 1.00 MPCSEM-> 1.00 MP2SEM-> 1.00
GP1SEM-> 1.00 GP2SEM-> 1.00 VERSEM-> 1.00 PNCSEM-> 1.00 SP1SEM-> 1.00
DSKSEM-> 2.00 FNTSEM-> 1.00 ASYSEM-> 1.00 SLXSEM-> 1.00 IPQSEM-> 1.00
IBSSEM-> 1.00 ASRSEM-> 64.00 NTWAIT-> 17.00 NBRSEM-> 3.00
```

Sample output

5.4.1.4 Distortion and integrity considerations

Special consideration has been given to minimising the distortion created by the presence of another process in the system. The major step taken, is that before each semaphore is sampled, the monitor will suspend its operation to allow some time for the system to settle, without the activities of the monitor affecting the workload.

5.4.2 Validation

The monitor is validated each time the means are displayed. This is done by adding together the mean length of each of the semaphore queues found. This sum should be equal to the number of processes in the system and any variation between the expected number of processes in the system and the number counted is displayed as the

percentage error.

5.4.2.1 Known variations

There are two areas where processes may be suspended that the monitor cannot access.

- o Primos will not allow segment 0 to be made available for reading. Any processes waiting on semaphore queues in segment 0 will not be counted.

- o The ready-list is not built using semaphores and so the monitor does not count processes waiting on it.

Despite these known causes of error the monitor has been found to be within a few percent of the expected result. In a trial of 100 monitoring sessions the mean error was 3.8% with a standard deviation of 0.3%.

5.5 Batch queue length monitor

The major features of this monitor include:

- o Weighted random sampling.
- o Delaying before sampling.
- o Multi-level reporting, using graphs, following the measurement period.

5.5.1 Purpose

After producing the interactive semaphore queue length monitor two further aims were identified.

Firstly it was found unreasonable to expect the user of the monitor to have a good background against which to judge the length of the semaphore queues displayed. We hope, with this monitor, to provide this background for those who need to obtain quick measures of the semaphore queue lengths in a system which occasionally performs and, for those users who wish to monitor a system which continuously performs badly, to eliminate the need to gain this background at all.

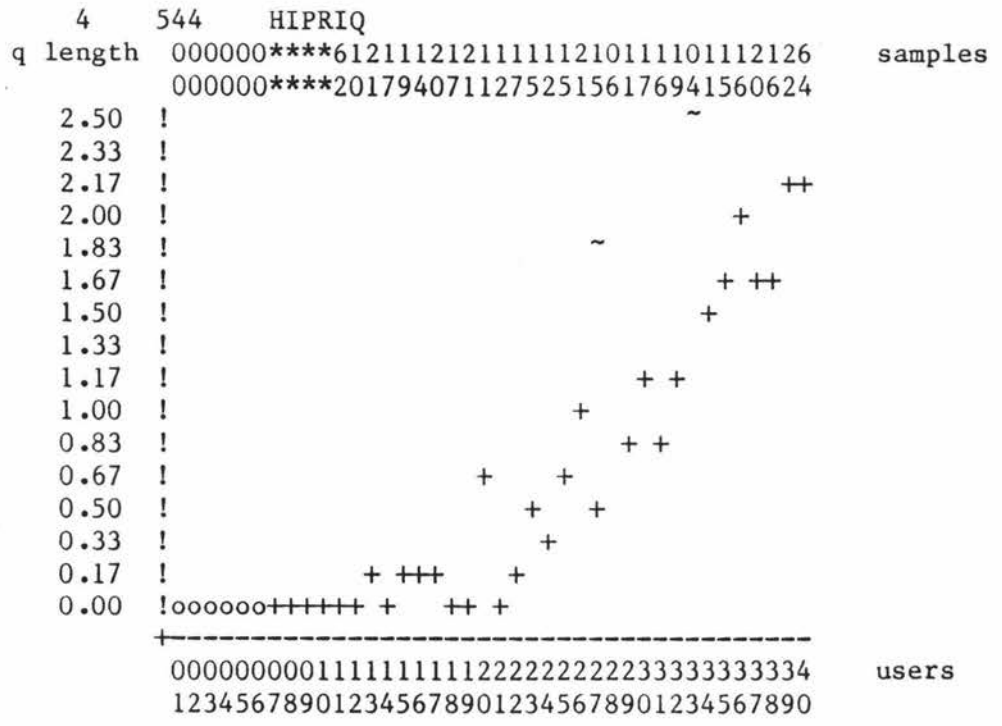
Secondly, for those users who have no need for immediate results, we hope to further reduce the distortions created by the monitor's presence in the system by moving much of the work required to produce usable statistics to a time following the completion of the monitoring period.

5.5.2 Implementation

The method is the same as for the interactive semaphore queue length monitor except that instead of accumulating the semaphore queue lengths and displaying them at the user's terminal, they are simply written to a file. Reports are generated by a separate program at a later date.

5.5.2.1 Reports

Two levels of report can be generated from the saved data to show the way the length of each semaphore queue, or each group of semaphore queues with a similar function, vary as the number of logged on users varies. During the production of the reports the sample values are weighted by the time between them. All graphs follow the general format shown below.



Sample output
Fig 5.1

The vertical scale (0 - 2.50) is the length of the queue. To maintain as much detail as possible over the widest range of possible queue lengths this scale will vary to suit each semaphore. The horizontal scale (0 - 40) is the number of logged on users, one user per division.

The two numbers at the top left hand corner of the plot (4 544) indicate the segment and word number of the semaphore control block, in this case segment 4 word 544.

The word immediately following this (HIPRIQ) is the external name associated with that location.

The two horizontal rows of figures at the top of the plot, read vertically, give the number of samples taken at each user count. For example, there were 16 samples of the HIPRIQ semaphore taken when there were 38 users on the machine. If the number of samples exceeds 100 then two asterisks (*) are printed.

The plus signs (+) and the tildes (~) represent points on the graph. A ~ is used when less than 10 samples were taken to produce a point (indicating a high probability of error). The open points (o) are placed on the bottom line of the plot when no samples where taken at the given number of users.

The following code performs the weighting of samples by the time between the current sample and the previous one.

```
 . . .

{long time is set to the current value of the 16-bit clock}
{last time is the time of the last sample}

 . . .


last_time  : integer;
long_time  : long_integer;


 . . .


IF last_time > long_time
```

```
THEN long_time := long_time + 65536;      {If clock has overflowed}

long_time := long_time - last_time;       {Calculate interval}

tree_grown := false;
tree^.sem_vals[users].samples := tree^.sem_vals[users].samples
                                    + long_time;
tree^.sem_vals[users].sum     := tree^.sem_vals[users].sum
                                    + length * long_time;
```

. . .

To show how the lengths of the semaphore queues vary with the workload the queue lengths are plotted against the number of logged on users (see Fig 5.1). For the number of users to be a valid measure of the workload, the assumption is made that the type of work on the system does not vary greatly during the monitoring interval. Stated another way, we may say that we do not expect the type of user of the system to vary with time, or with the loading of the system. If this is true then we can expect the number of users logged onto the machine to produce a statistically valid measure of the work running on the system.

5.5.2.1.1 Distortion and integrity considerations

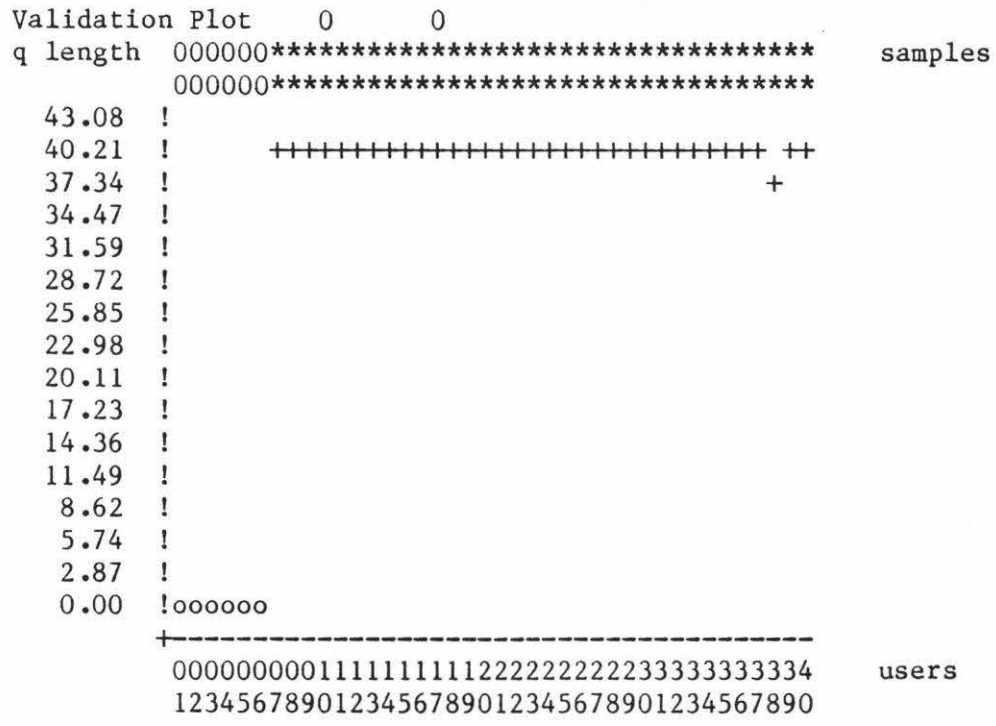
As with the interactive semaphore queue length monitor, consideration has been given to minimising the distortion of the workload created by the presence of another process in the system. In building this monitor the following steps have been taken:

- As with the interactive semaphore queue length monitor, immediately before each semaphore is sampled the monitor will suspend its operation to allow some time for the system to settle, without the activities of the monitor affecting the workload.

- The amount of disk accessing is minimised by only recording the minimum of data (the time, the semaphore's location and its length) and doing so in a binary format (as opposed to saving a character string) so that fewer physical writes to the disk occur.

5.5.2.2 Validation

The monitor is validated with each report generated. After all the semaphore queue lengths have been plotted, a graph showing the sum of the mean lengths of all semaphore queues found, against the number of logged on users is plotted. It is expected that this will form a straight line at the number of configured users as in Fig 5.2 below.



Example validation plot
Fig 5.2

5.5.2.2.1 Known variations

As with the interactive queue length monitor processes may wait

- o on semaphores in segment 0 or
- o on the ready-list,

and as a result not be measured.

Despite these known causes of error the monitor consistently produces the expected validation graph (within 8%, which is the

resolution of the graph). This supports our confidence in the weighted sampling technique and suggests that distortions of the type mentioned in section 3.3.2.1 are, in this case at least, small.

5.5.3 Disk activity monitor

The major features of this monitor are:

- o Event trapping.
- o Data recording via an intermediate buffer.
- o Reports generated outside the measurement interval using tables and histograms.

5.5.3.1 Purpose

On many computer systems the disk sub-system becomes a bottleneck. If this is the case it is useful to know:

- o What routines (or programs) are causing most of the disk I/O.
- o How the I/Os are spread between the drives connected to the system.
- o How well the disk activity is spread across each drive.

5.5.3.2 Background

Disk drives are attached to the CPU via one or two disk controllers, each of which may have up four drives connected to it (see figure 4.1). Each disk mounted on a drive may be broken up into a number of what Prime call physical devices. Each of these physical devices is a group of surfaces on a disk pack and each is described by a 16-bit physical device number. Paging occurs between main memory and one or two physical devices specified at cold start.

Primos contains a common routine, named RWREC in the module DISKIO.PMA, which is used during all disk accesses.

Primos also contains a mechanism used to log error conditions, such as disk read/write errors and memory parity errors. This mechanism is known as the event logger. It is implemented through two routines, LOGEV1 which writes information into a dedicated (locked down) buffer in main memory, and LOGEV2 which checks the buffer for new data each minute and, if any data is found, writes it to a disk file.

5.5.3.3 Implementation

DISKIO.PMA was patched to use the event logging mechanism to record data about disk I/Os.

RWREC was modified and a new routine, M_CNTL, and several words of data were added to the DISKIO.PMA module. M_CNTL was also added to the list of directly-callable operating system routines kept in segment 5.

The function of M_CNTL is to alter the frequency of monitoring, where this frequency is kept in a location in the DISKIO.PMA module.

The patch functions as follows:

Test to see if monitoring is off, if so exit, otherwise ...

- o Increment counter.
- o If counter doesn't hit zero then exit.
- o Call LOGEV1 with the Physical Device Number and Record Address.
- o Call LOGEV1 for the last 5 levels of the stack (or till we hit a frame for a procedure call from user's private address space) to record address of procedure call instruction.
- o Inhibit interrupts.
- o If counter has overflowed, record overflow.

- Reduce value of counter by frequency of monitoring.
- Enable interrupts.
- Exit.

The code for this patch is:

```
*
*
*=============================================================
*               MASSEY PATCH TO LOG DISK ACTIVITY
*
*If monitoring not on skip patch
        LDA    FREQ
        BEQ    ENDPCH
*Increment nleft counter & leave patch if <> 0
        IRS    NLEFT
        JMP    ENDPCH
*If nleft = 0 then record current device#, record# and
        LDA    XDVNO,*                        stack history
        STA    EMES+1
        LDL    XCRA,*
        STL    EMES+2
        CALL   LOGEV1
        AP     EMES,SL
*
*
*
*  Do backward trace
*
*
        LDA    =-5         *Allow for maximum trace back of five
        STA    MAXDPH
        CRA
        STA    EMES+1      *Mark this as a traceback entry
        EAL    SB%         *Pick up current stack frame
LOOP    EQU *
        STL   SAVXB
        TBA                *Put word number into A
        BEQ   ENDSTK       *Stop traceback if wordnumber = 0
        EAXB  SAVXB,*
        LDL   XB%+2        *Pick up PCL which created this frame
        STL   EMES+2       *Put PCL address into message
        ANA   ='4000       *If this is a user seg stop trace back
        BNE   ENDSTK
        CALL  LOGEV1       *Save message in logbuff
        AP    EMES,SL
        EAXB  SAVXB,*      *Move down stack one frame
        EAXB  XB%+4,*
        EAL   XB%
```

```
        IRS   MAXDPH      *Leave if max trace back level reached
        JMP   LOOP
*
*Now reset nleft. This must be inhibited in case others also
*                                                        change it
ENDSTK INHM  <+><+><+><+><+><+><+><+><+><+><+><+>
        LDA   NLEFT
        SUB   FREQ
        BLT   NOOF          *Has an overflow occurred?
*
        LDA   NLEFT         *Overflow has occurred
        ADD   OFCNT
        STA   OFCNT
        CRA
        SUB   FREQ
NOOF    STA   NLEFT
        ENBM  <->><->><->><->><->><->><->><->><->><->><->><->
        JMP   ENDPCH
*
*The control routine. This is called through a gate to
*set the rate of disk logging
*
*
XNEWF   EQU XDVNO     *The new rate (0 => no logging)
MCNTL   ECB   MCNTLX,XNEWF,1
*
MCNTLX ARGT
        LDA   XNEWF,*
        STA   FREQ
        PRTN
*
NLEFT   DATA  -100    *Give us a little time before beginning
FREQ    DATA  0                                    to monitor
OFCNT   DATA  0
MAXDPH DATA  -5
SAVXB   DATA  0
        DATA  0
EMES    DATA  (10.LS.8)+4
        BSZ   3
*
ENDPCH EQU    *
*
*                 END OF PATCH
*=========================================================
```

5.5.3.4 Reports

Programs were written to extract the recorded data from the logging file and produce three reports. These reports are:

- o The device breakdown.

  This gives the percentage of I/Os which fell on each physical device, and the total number of I/Os recorded.

Device	Calls
4060	76.5%
4062	15.3%
100463	8.2%

Total accesses recorded = 98

Example analysis by device

- o The caller list.

  This lists the address and external name of each procedure call found that results in a disk I/O, and the percentage of calls which passed through that routine.

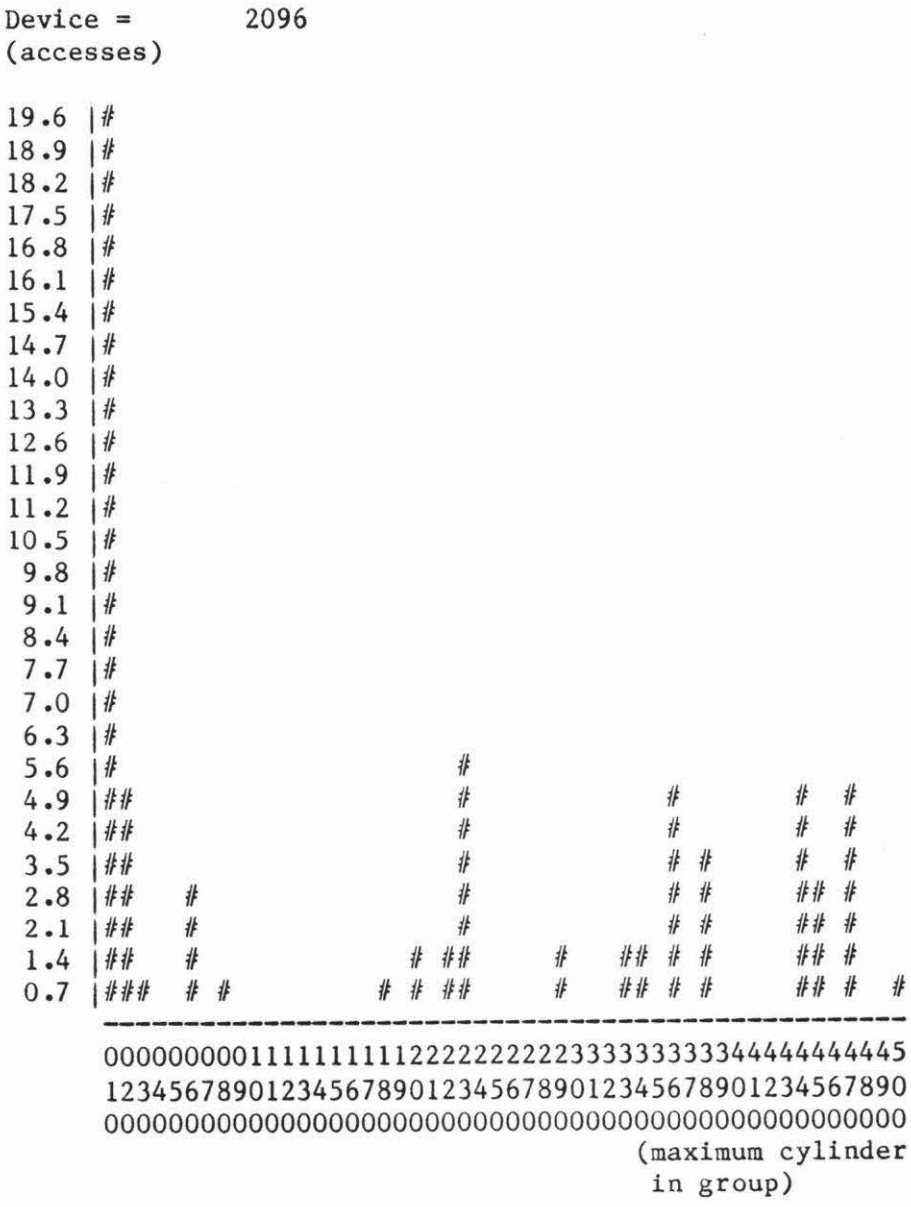
Name	Address		offset		
STIMER	6	4403	1036	...	5.2%
P3TRAP	6	5137	466	...	14.4%
MOV32P	6	6240	67	...	1.0%
SETABT	6	7060	414	...	1.0%

PAGTUR	6	27754	234	...	4.1%
PAGTUR	6	30006	2486	...	6.2%
PAGTUR	6	30464	2944	...	4.1%
total	0	0		...	14.4%
Shared	2035	37565		...	5.2%
Shared	2040	100744		...	1.0%
Shared	2040	101557		...	1.0%
Shared	2040	101754		...	1.0%

Example analysis by caller

- The surface distribution histogram.

  This plots for each small group of adjacent cylinders of a physical device the number of accesses to that group of cylinders.



Example surface plot

5.5.3.5 Testing

The monitor was developed using a scratch disk pack. Before allowing it to be used on the real system the Computer Centre staff required convincing that it was safe. Rigorous testing was done to ensure that the monitor did not introduce or uncover any operating system bugs. This included setting the monitoring frequency to 1 and running many disk intensive programs. The monitor was also switched on and off during times of high system loading. No problems were found.

5.5.3.6 Validation

The monitor was validated by running a program with a known disk accessing behaviour, when no other user processes were using the machine. This program accessed an assigned disk by record, incrementing the record number one record at a time. The monitor recorded every one hundredth disk access as expected (the frequency was set to 100) with the addition that each minute an extra access was recorded, due to LOGEV2. This program (EXEC.PASCAL) is included in Appendix B. We would have liked to have included the validation data here but unfortunately, due to the problems with Primos discussed in section 5.2, we were unable to run the monitor while this thesis was being prepared.

5.5.4 Relative availability monitor

The major features of this monitor are:

- o Data collection by unit workloading.
- o Loosely coupled sampling function.
- o Kiviat graph output.

5.5.4.1 Purpose

This monitor measures how evenly utilised the system is and whether the demands for the main resources (the CPU and the disk drives) are evenly balanced. The availability of the CPU is combined with the availability of the disks on the system to produce a Kiviat graph.

5.5.4.2 Background

The allocation of the CPU to a process depends on the priority of the process and the amount of CPU time it has used since the last terminal input (see Appendix A, section 5.7). Two types of time slice are involved when the CPU is dispatched. These are

o The minor time slice.

This is the normal type of time slice found on most time sharing systems. It ensures that no process is allowed to hold the CPU for a long period of time (more that 0.333 seconds by default on the Prime).

o The major time slice.

This sets the maximum amount of CPU time which a process can use between terminal inputs without losing its interactive classification. It is 2 seconds by default on the Prime.

Consequently three types of CPU user can be identified:

o The trivial user.

From the point of view of the CPU queues a CPU operation which is less than the minor timeslice can be classed as trivial. It is not necessary for the process to wait on any CPU semaphore during its operation.

o The light user.

A CPU operation which exceeds one minor time slice, but does not exceed the major time slice, will require the user to wait on the CPU semaphores intended for interactive processes.

o The heavy user.

A CPU operation which exceeds the major time slice will

require the process to wait on both the semaphores for interactive processes and the semaphores for batch processes.

5.5.4.3 Implementation

This monitor gathers data using the unit workloading technique described in section 3.3.2.1.1. Two types of mini-bench mark were constructed. One to measure the time of a CPU operation and one to measure the time of a disk operation. The CPU benchmark performs a small CPU operation the number of times specified by a parameter, and is used to measure the time taken to perform a trivial, a small and a large CPU operation. The disk benchmark writes a character to the file open on a unit number passed to it as a parameter. This write operation is forced to the disk to defeat the disk buffering system.

There are four modules in the monitor. These are

- o the sampler,
- o the reducer,
- o the reporter, and

o the calibrator.

5.5.4.3.1 The sampler

This module executes each benchmark, records the results, and then waits until the next sample is due. The procedure is repeated the number of times specified by the user when the monitor is run.

5.5.4.3.1.1 The loosely coupled sampling function

Waiting the required time until the sample is due to be taken is done using a timed semaphore, and is therefore described as a loosely coupled sampling function. The following code is used:

```
 . . .
{INITIALISATIONS}

write('Enter frequency of sampling (sec) - ');
readln(delay);

 . . .

{Set up the timed notify}
sem$tn(sem_num,delay*1000,delay*1000,code);
IF code <> 0
THEN writeln('Error ',code,' at 3');

{Set the semaphore count to 0}
sem$dr(sem_num,code);
IF code <> 0
THEN writeln('Error ',code,' at 4');

 . . .
```

```
{MAIN LOOP BEGIN}

   {Check that the semaphore hasn't already been signalled}
   IF (sem$ts(sem_num,code) <> 0)
   THEN writeln('Frequency to high, sem count is ',
                sem$ts(sem_num,code) );

   IF code <> 0
   THEN writeln('Error ',code,' at 1');


   {Wait on semaphore}
   sem$wt(sem_num,code);
   IF code <> 0
   THEN writeln('Error ',code,' at 2');


  . . .

   {TAKE THE SAMPLE}

{MAIN LOOP END}
```

5.5.4.3.1.2 The mini-benchmarks

Several points had to be given special consideration to ensure that only the resource we wished to measure was used during the measurement interval.

- Page faults. To ensure that a page fault did not occur the code for the mini-benchmark was written and loaded into a singe page. The start and finish times were taken from within this page.

o The CPU time to control transfer. While it is possible to perform a CPU operation which does not make any use of the disks on the system, the reverse is not so. Some CPU time will be used to control any disk transfer. If the CPU becomes less available then the real time elapsed to get the required CPU time will increase. This means that the measurement time will increase even though the actual disk related times may not have increased. To determine whether this is a significant part of the total disk transfer time we ran the benchmark 500 times, both with and without the presence of several CPU intensive processes. During the test there was one other user on the machine. The results were

	Mean	Std. dev.
Alone	57.0	1.0
Loaded	58.2	3.8

If we apply a Z-test to the hypothesis that mean(alone) = mean(loaded) we get a Z score of 6.83 which we can not accept, even at less than 50% confidence, and we conclude that the time for the benchmark to complete does increase when the processor is heavily loaded.

If we apply a Z-test to the hypothesis that mean(loaded) - mean(alone) < 3% of mean(alone) we get a Z score of -2.9 giving better than 99% confidence in our hypothesis.

As this is a worst case, and the error introduced is almost certainly less than 3%, we do not consider it necessary to normalize the time measured to allow for CPU utilisation.

o Interrupts. Interrupts that occur while a process is using the CPU are considered to be part of the cost of performing a CPU operation, but are not considered to be part of the cost to perform a disk operation. Most interrupts take only a small fraction of the available CPU time to service, but some interrupts can cause a page fault to occur. If a page fault happens then a lot of time which should not be included in measurements of the disk availability may pass before the interrupted process regains the CPU. To avoid including this time in the measurement, the count of the number of page faults kept by Primos is checked before and after the measurement and the benchmark is rerun if one has occurred.

The code for the CPU unit workload is:

```
        SEG
        ENT  CPU
        DYNM COUNT(3),TIME(3)

START   EQU   *
        ARGT

        LDA    =0              *Initialise loop counter
        STA    DELAY

        CALL   RECYCLE         *Recycle to start of time slice
```

```
        LDX     COUNT,*         *Put iterations count into x
        LDL     GCLOK,*         *Take time

LOOP    IRS     DELAY
        JMP     LOOP            *Unit of work (65536 iterations)

        DRX                     *Decrement count and skip if 0
        JMP     LOOP

        SBL     GCLOK,*         *Take time again
        TCL                     *Calculate difference
        STL     TIME,*

        PRTN

GCLOK   DATA    '0006           *SN of clock location
        DATA    '1216           *WN of first clock location
DELAY   DATA    0
        LINK
CPU     ECB       START,,COUNT,2
        DYNT      RECYCLE
        END
```

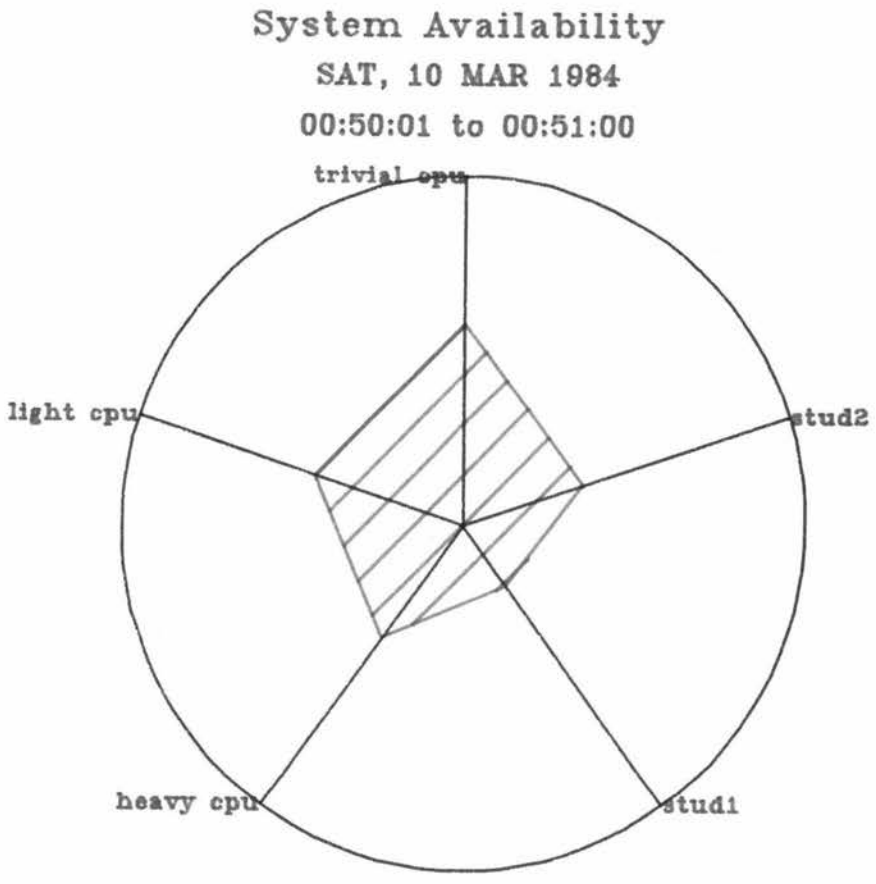
CPU unit workload

5.5.4.3.2 The reducer

This module takes the data collected by the sampler and the base figures produced by the calibrator and produces the relative availability for each type of resource.

5.5.4.3.3 The plotter

There are a number of variations on this module for different devices. The Kiviat plot can be drawn on a dumb terminal, a printer, or an NCAR [NCAR81] graphics library type device (in our case an HP plotter, Printronix printer or Tektronix terminal).



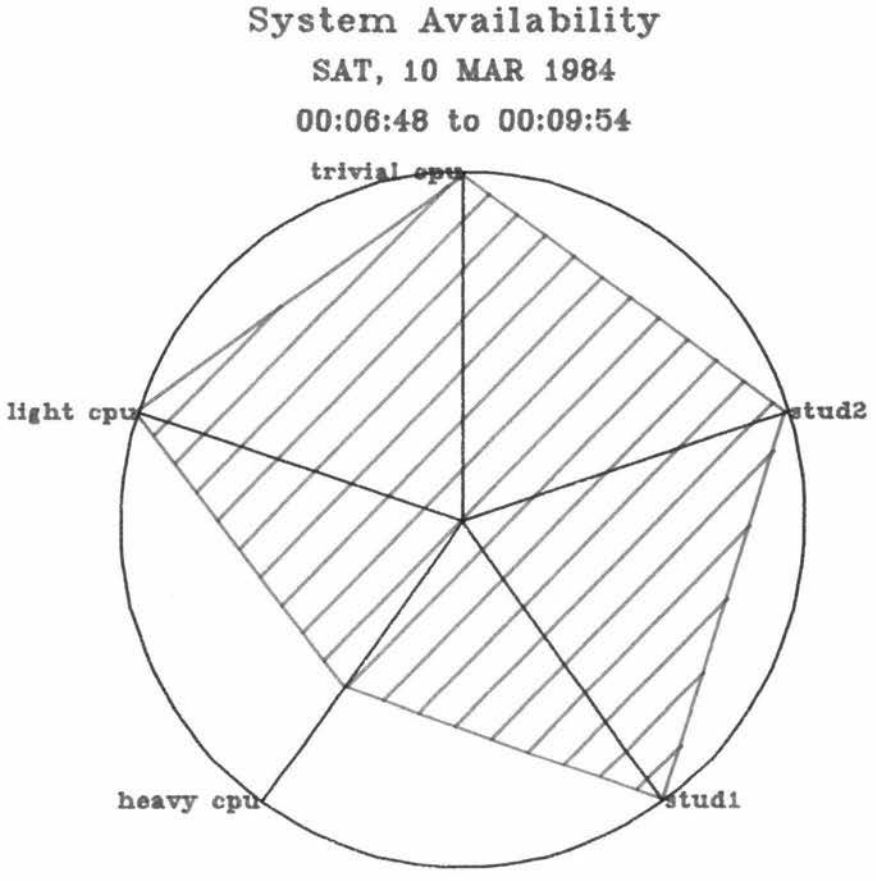
Sample output of relative availability monitor

5.5.4.3.4 The calibrator

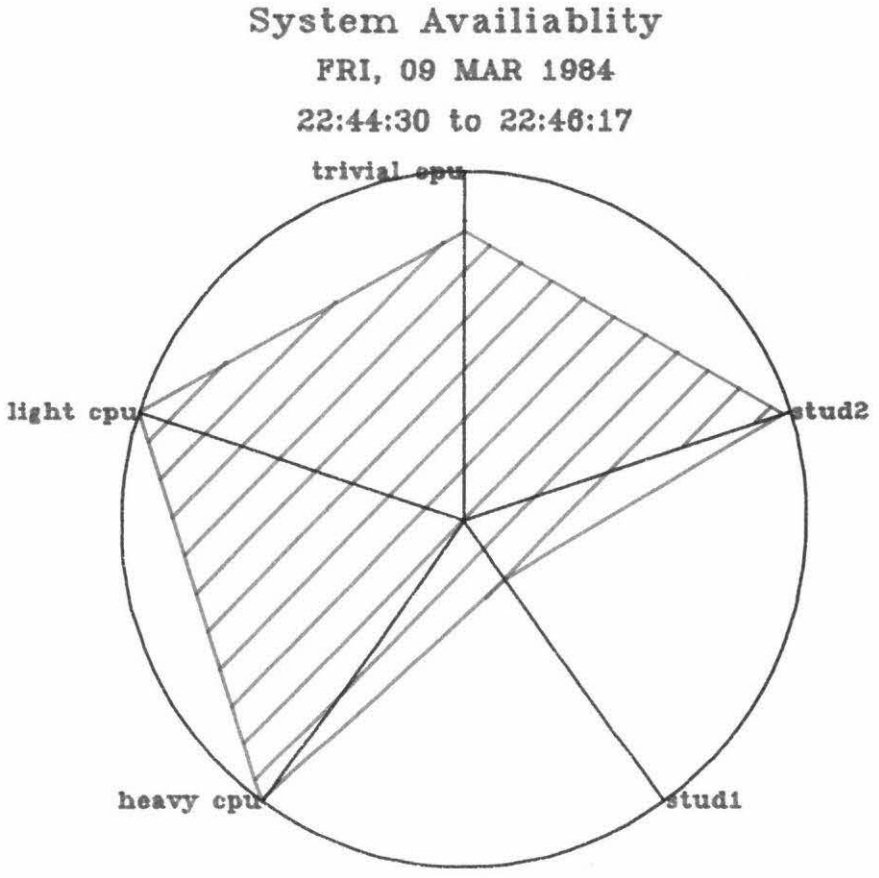
This module is run when there are no other user processes on the system, and collects the maximum values of the indices against which the percentage availability is calculated in the reducer module.

5.5.4.4 Testing

The monitor was tested with the aid of two programs. The first is a CPU-bound process which, assuming sufficient memory is available at the time it is run to avoid any paging occurring, does not use the disk. The second is a diskbound program. These programs and the monitor were run on an otherwise empty system and the resulting graphs are included below.



Large CPU job alone in the system



Disk1 job alone in the system

5.5.4.5 Validation

Because of the indirect nature of this monitor, quantitative validation is difficult to perform. We could prove, for example, that the monitor did measure the time for the benchmark to complete, but that in itself would not be a convincing proof that the monitor correctly measured how heavily used the CPU was during the period under question. Nor would it provide any quantitative measure of how close the monitor was in a given case.

The validation performed was to measure the real time for a program to complete when it was the only user process, besides the monitor, using the system, and the time for it to complete when jobs that used various proportions of CPU and disk were also in the system. The elapsed times were compared with the results of the monitor. As an example the results of the CPU part of the validation are shown below:

Availability of CPU during run **	Predicted elapsed time(sec)*	Measured elapsed time(sec)	Difference (sec)
0.61	26	28	2
0.70	23	24	1
0.75	21	20	-1
0.82	20	19	-1
0.91	18s	16s	-2s

Notes:

** Availability as measured by the monitor.

* Pedicted elapsed time = time at 100% availability multiplied by 1/availability.

The mean and standard deviation of the differences between the expected and the measured results are -0.2 and 1.64 respectively. The t-score for the hypothesis that mean(differences) = 0 is 3.70 (with four degrees of freedom), and this leads us to accept the hypothesis with better than 97% confidence.

5.6 Extensions to Primon

Because of the limitations of time and the problems discussed in section 5.2, the following monitors have not yet been implemented but head the list of extensions to Primon that will be carried out in the future.

5.6.1 The effect of adding extra memory

This monitor would be of most interest to

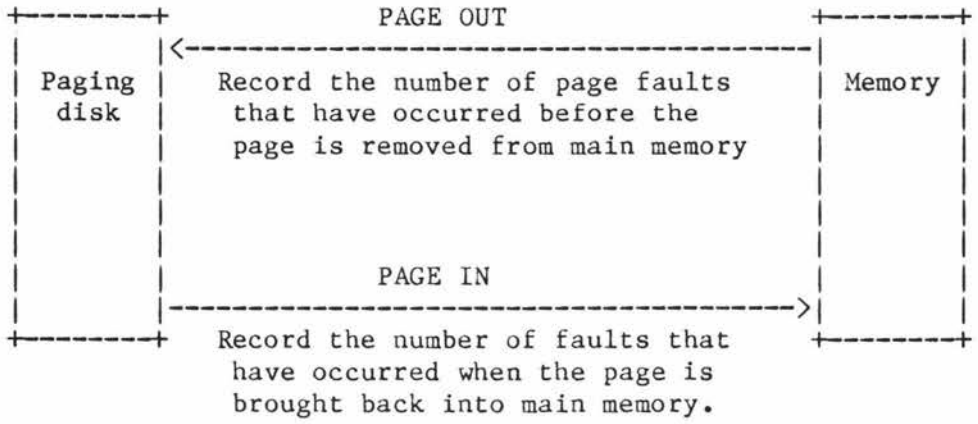
- CPE personnel.

If there is insufficient memory on the system, the paging rate will rise to a level where the processor is often held up waiting for pages to be retrieved from secondary memory, that is thrashing occurs. As described in Appendix A, section 3.3.2 Primos uses an approximate, system-wide, least recently used paging algorithm with only a little pre-pageout. This means that under normal workloads the memory of the system will remain full and the amount of memory in use, which is a common CPE statistic, is not a very useful index. An alternative approach is given below.

The reason for measuring memory usage is to decide whether memory is a scarce resource, and if so to hopefully gain some idea of how the addition of extra memory will affect the system. If the

distribution of the number of page faults that occur while individual pages are on the paging disk is measured we can find that x% of pages spend less than j faults on the paging disk.

By plotting the proportion of pages recalled to main memory against the number of page faults that have occurred since the page was removed from main memory, we get a direct reading of the reduction in the paging rate expected with the addition of each extra page (1KW) of main memory. In fact the axes of the graph could be relabelled "Reduction in the paging rate (%)" and "Memory added (pages)" see Fig 5.3. This is because any page that spends n faults or less on the paging disk would have been kept in main memory if an additional n pages of main memory were available.

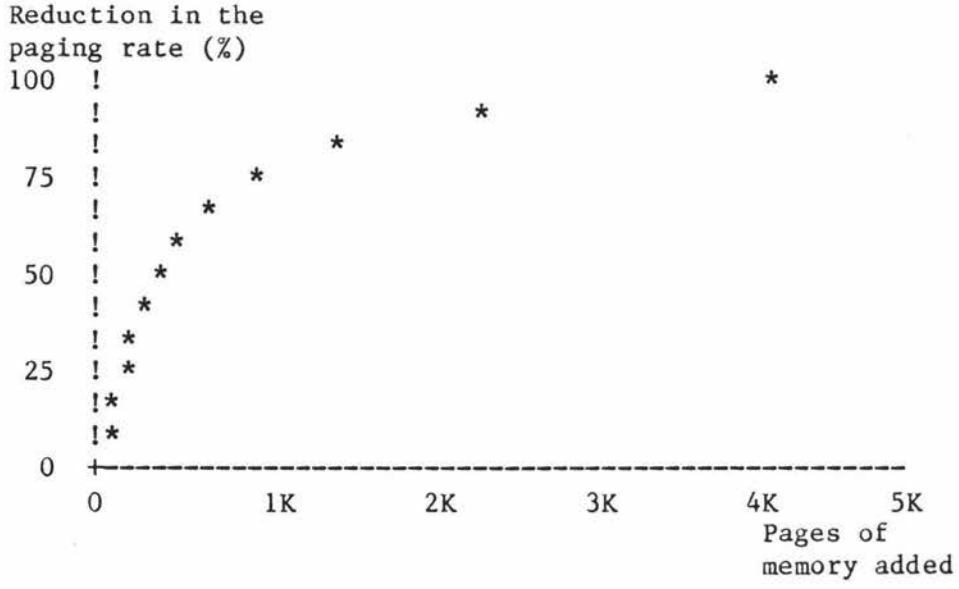


This data could be collected by the following patches.

- Patch the pageout routines so that they check to see if a page is currently tagged, and if not record the page fault count and the disk address of the page being paged out.

o Patch the page fault (page routine) to check if the page being bought in is the tagged page or if a maximum number of faults have occurred since the page was tagged. If so then call the event logging mechanism to record how many faults occurred while the page was out on the paging disk.

From this data we can find that x% of pages are recalled to main memory within j faults.



Predicted effect of adding extra memory
Fig 5.3

5.6.2 Collection of think times

This monitor is primarily of interest to

- o CPE researchers.

In particular at Massey University the QNEMU queueing network package [LADY84] could be used to model Prime computer systems if the mean think time was available (in addition to other information that we can already collect).

A mechanism to determine the mean think times using an event trap and accumulator locations was discussed in Section 4.1.3.

5.6.3 Disk record positioning

With a minor modification to the disk usage monitor, we feel that a significant improvement in the performance of the disk system may be achieved.

Experimentation has shown that when no head movement is necessary the disk I/O system is capable of greatly improved performance:

	Same track	Normal workload
Max I/Os per second	197	37.4

Primos does not attempt to place disk records most commonly used cylinders near to one another. It is suggested that records on the disk should be repositioned so that higher disk throughput rates can be achieved.

Data by the disk activity monitor (section 5.5.3) may be used to form a profile of how frequently particular disk records are used, (although to maximise the allowable sampling rate it would be desirable to inhibit the reverse trace of the stack). A separate utility program could then be written that used this information to gather those records closer together on the disk.

The effectiveness of this process is open to debate, although what statistics we were able to gather while the disk activity monitor was correctly installed, suggest that some cylinders on the disk are much more heavily referenced than others. Many of the Primos data structures also suggest that this is likely to be the case. Consider the paging disk, for example: a spot on the disk is allocated for every possible virtual page although the loaders always favours particular pages, and so the records allocated to favoured pages will be most heavily used. The filing system provides a second example: the records containing those elements

which occur in many pathnames, for example the MFDs and top level UFDs are likely to be referenced most often.

5.7 Evaluation of Primon

From the teaching aspect we feel that we have implemented a simple to use monitoring package that uses a variety of techniques, including:

- o Collecting data by
  - event trapping and
  - random sampling.
- o Recording data by
  - directly calling the disk routines and
  - via an intermediate buffer.
- o Reporting the results of the monitoring session
  - interactively and
  - after the measurement interval.
- o Using
  - tables
  - graphs
  - histograms and
  - Kiviat plots.

As well as this we have introduced some non-standard techniques.

From the point of view of other CPE researchers within the Performance Evaluation Group at Massey University we have measured all the parameters required to apply QNEMU to a real system, with the exception of the mean think time which we have discussed and intend

to measure in the near future.

From the point of view of CPE personnel in computer centres we can:

- Measure the utilisation and overall balance of a system.
- Find the bottlenecks in a system.
- Give detailed information about the use and functioning of the disk system.

As well as this we have suggested:

- A method for predicting the usefulness of adding extra memory to a system.
- A method which should improve the performance of the disk system, and consequently the computer as a whole, particularly if it is disk bound.

6 Conclusions

As a result of this research we have developed the substantial part of a software monitor, and the associated documentation, that is useful to personnel involved in

- o teaching,
- o researching and
- o applying

CPE tools.

The new techniques of

- o unit-workloading,
- o loosely coupled sampling and
- o weighted sampling

have been introduced theoretically and also studied in practice.

Assembling details of the Primos operating system consumed a major part of the time available for this research. It is hoped that Appendix A, which provides details on the structure of Primos can

significantly reduce the effort required of other analysts who wish to develop CPE tools for Prime systems.

Further research that could be usefully undertaken was discussed in chapter 5, and includes implementing tools to

- o predict the effect of adding extra memory,
- o improve the performance of the disk sub-system, and
- o allow QNEMU [LADY84] to be used to model Prime computer systems.

APPENDIX A

Appendix A
Table of Contents

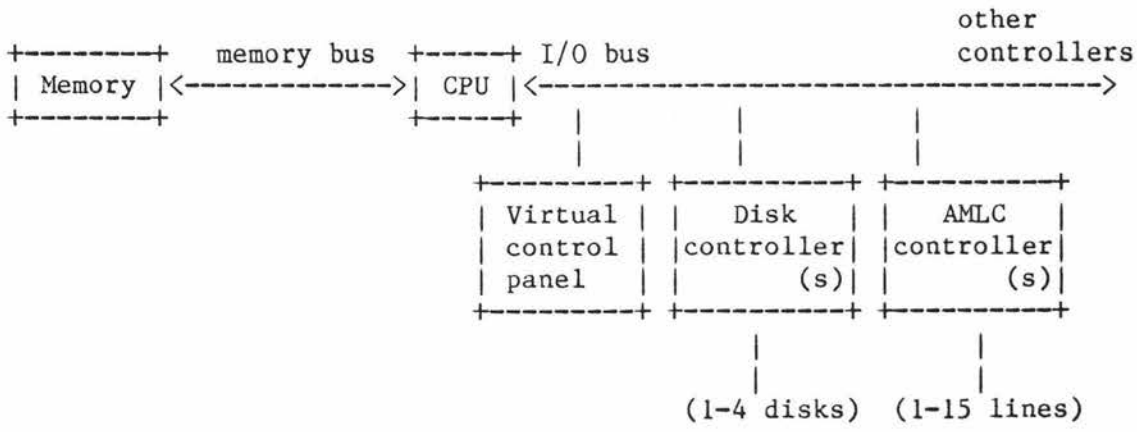
1 Introduction

In 1972 prime released there first processor, the P200. This machine was plug compatible with the Honeywell Series 16, a 16 bit machine with 64K RAM. Since that time Prime has released processors which range in complexity up to the P850, a dual processors with an address space of 512Mb. The P850 is two P750s joined together to give it the dual instruction stream capability. This appendix is centred around the P750, two of which are operated by the Massey University Computer Centre. The information presented in this overview was obtained from the following documents:

- The PMA programmers guide.
- The course notes for the "Primos Concepts and Implementation Course"
- The Subroutines Reference Guide
- The Systems Architecture Guide.

This information is neither complete nor consistent. The source code has been used to resolve these difficulties when they have arisen but this is a time-consuming process and it is hoped that this guide will assist with future research which requires an understanding of the way Primos functions.

As it is nessecary to have an understanding of some parts of the hardware of a computer system if the operating system is to be fully understood (the reverse is also true) this document covers some aspects of both. Fig 1.1 gives on overview of the organisation of the P750 hardware.



Overview of the P750 architecture
Fig 1.1

Some of the features of the P750 and Primos are

- o 512 Mb Virtual address space
- o 8 Mb physical memory
- o Ring structured memory protection scheme
- o Cache Memory
- o Procedural architecture
- o Multi-level CPU dispatching
- o Tree structured file system
- o File system buffering

o Execution of code for any other Prime CPUs

2 Instruction sets and Addressing modes

As Prime introduce new processors they aim to maintain compatibility, at the object file level, with their earlier machines but specify that a higher processor will execute code faster [PRIM4]. To achieve this a number of instruction sets have been implemented on the P750. Each of these instruction sets really reflects a different machine architecture.

The instruction sets available on the P750 are

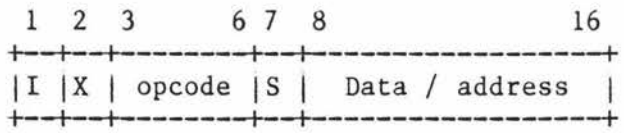
1) 16S mode. (16K sector relative addressing)
2) 32S mode. (32K sector relative addressing)
3) 32R mode. (32K procedure relative addressing)
4) 64R mode. (64K procedure relative addressing)
5) 64V mode. (Virtual memory 64K segments)
6) 32I mode. (32 bit integer arithmetic)

2.1 16S mode

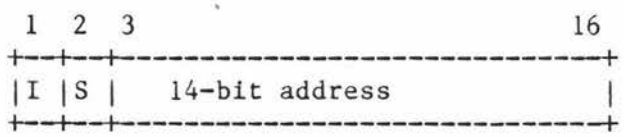
This mode reflects a machine with a 16 bit word and a 16K address space divided into 512 word sectors. All instructions are single words and addressing is relative to the current sector or to sector 0. Indexing and indirection are supported to an arbitrary depth by including bits in the indirect address to specify indexing or indirection. This mode is supported by all Prime CPUs.

No Prime machine has this (or 32S mode) as its basic architecture, but some Honeywell machines do.

Instruction



Indirect Address Word



Where

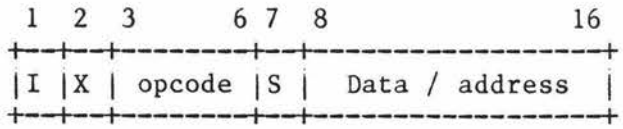
I bit => indirection
X bit => Indexing
S bit => If 0 then address is relative to sector 0
If 1 then address is relative to program counter.

16S Mode Word Formats
Fig 2.1

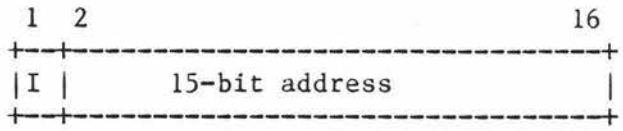
2.2 32S mode

By removing the indexing bit from indirect addresses (therefore only allowing indexing once in the calculation of the final address), 15 bits are available for the address in the indirect word and a 32K address range can be supported. All Prime CPUs support this mode.

Instruction



Indirect Address Word



Where

I bit => indirection
X bit => Indexing
S bit => If 0 then address is relative to sector 0
If 1 then address is relative to program counter.

32S Mode Word Formats
Fig 2.2

2.3 32R mode

This is the first of the instruction sets defined by Prime. It fits a machine with 16 bit words and a virtual address range of 32K. The addressing mode depends upon the value of the 8 bit primary address in the instruction. If this, interpreted as a signed number, is greater than or equal to -240(octal) then the effective address is the current program counter plus the value of the address. If the primary address is less than -240(octal) then the primary address is treated as a code.

In this code

bits 1-2 = 11 (to make address < -240)

bits 9-12 = 0000.

Bits 13-14 are an extension of the op code (the first bits are bits 3-6 of the instruction).

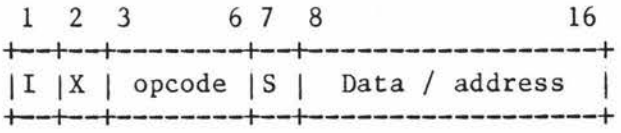
Bits 15-16 are a sub code specifying

- 00 => next word is a 15 bit address (bit 1 =0),
- 01 => add the primary address to the stack pointer,
- 10 => the stack pointer post-incremented is the address,
- 11 => pre-decrement the stack pointer then use it as the address.

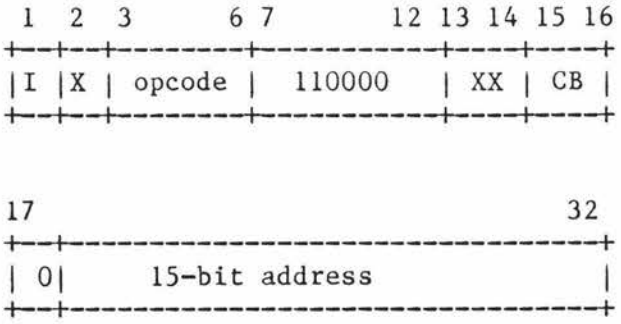
In the short opcode mode multiple indirection is available with indexing only after all indirection. In the extended opcode mode only one level of indirection and one indexing operation are available. The extra opcodes made available by the two opcode extension bits are used mainly to provide instructions for virtual addressing.

This mode is supposedly supported by all Prime CPUs [PRIM2]. The function of the virtual addressing instructions (e.g. EVMX is used to enter virtual memory mode) remain unclear when applied to a machine without virtual addressing e.g. P100 and P200.

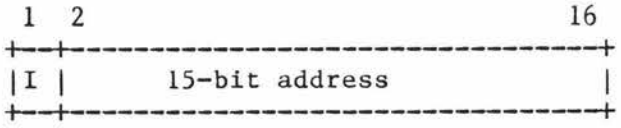
Standard Instruction Word (Bit 7 or 8 = 0)



Extended Instruction Mode (Bit 7 and 8 = 1)



Indirect Address Word



Where

I bit => indirection
X bit => Indexing
S bit => If 0 then address is relative to sector 0
If 1 then address is relative to program counter.
XX => Opcode extension.
CB => Subcode for mode.

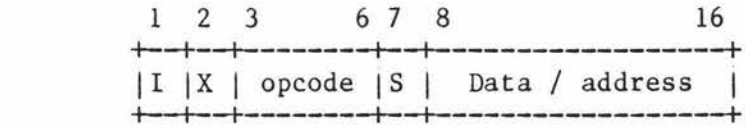
32R Mode Word Formats
Fig 2.3

2.4 64R mode

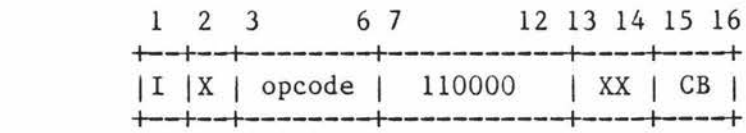
This mode is identical to 32R mode except that the facility for multiple indirection has been removed. This allows the indirect word to be a full 16 bits to provide an address range of 64K. Bit 1 of the second word of extended opcode instructions is no longer kept set to 0.

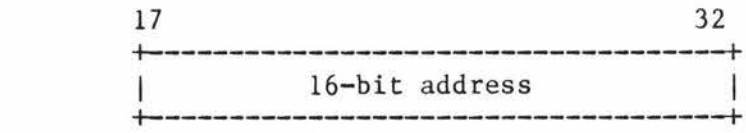
All Prime CPUs support this mode and similar questions arise concerning virtual memory.

Standard Instruction Word (Bit 7 or 8 = 0)

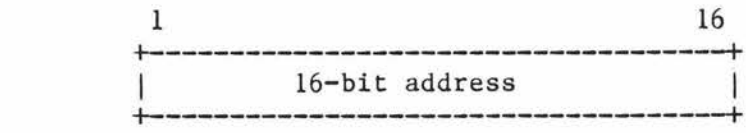


Extended Instruction Mode (Bit 7 and 8 = 1)





Indirect Address Word



Where

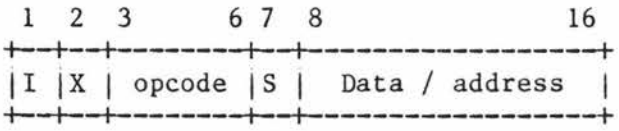
I bit => indirection
X bit => Indexing
S bit => If 0 then address is relative to sector 0
If 1 then address is relative to program counter.
XX => Opcode extension.
CB => Subcode for mode.

64R Mode Word Formats
Fig 2.4

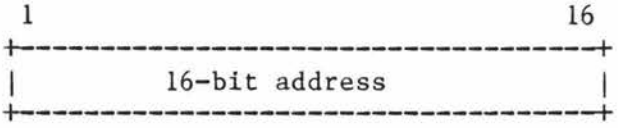
2.5 64V mode

This is the first mode with a segmented address space. The word size is still 16 bits but a total of 512Mb(virtual) are addressable. At any one time four 64K segments are accessible directly and the entire address space is accessible indirectly. The addressing mechanism in 64V mode is discussed further in section 3.2.1 (virtual address translation).

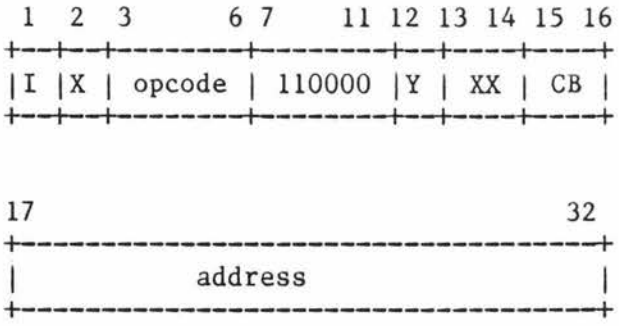
Standard Instruction Word (Bit 7 or 8 = 0)



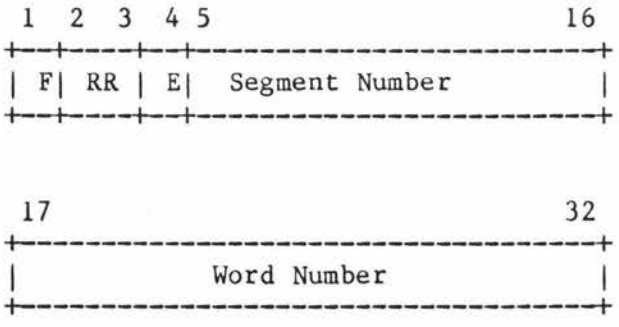
Indirect Address Word



Two word Instruction Mode (Bit 7 and 8 = 1)



Indirect Address (2 or 3 words depending on the E-bit)



and if the E-bit = 1

Where

```
 I bit  =>  indirection
 X bit  =>  Indexing on X register
 Y bit  =>  Indexing on Y register
 S bit  =>  If 0 then address is relative to program counter
            If 1 then address is stack or link base (depending
                 on the value of the address).
XX bits=>  Opcode extension.
BR bits=>  Base register
                 00 => PB
                 01 => SB
                 10 => LB
                 11 => XB
 F bit  =>  Fault bit.
 R bits =>  Ring Number (not used)
 E bit  =>  Three word format.
```

64V Mode Word Formats
Fig 2.5

3 Memory management

The virtual memory system on the P750 is based on paged segments. Paging-in is on a demand basis and paging-out is on an approximate least recently used system with a number of pages paged-out at once. Memory protection is implemented using a ring system with four possible levels of authority. Each segment can be given one of eight different types of access for each of the three lower levels of authority. When the processor is in ring 0 it has full access to all segments. Paging occurs between memory and one or two paging devices.

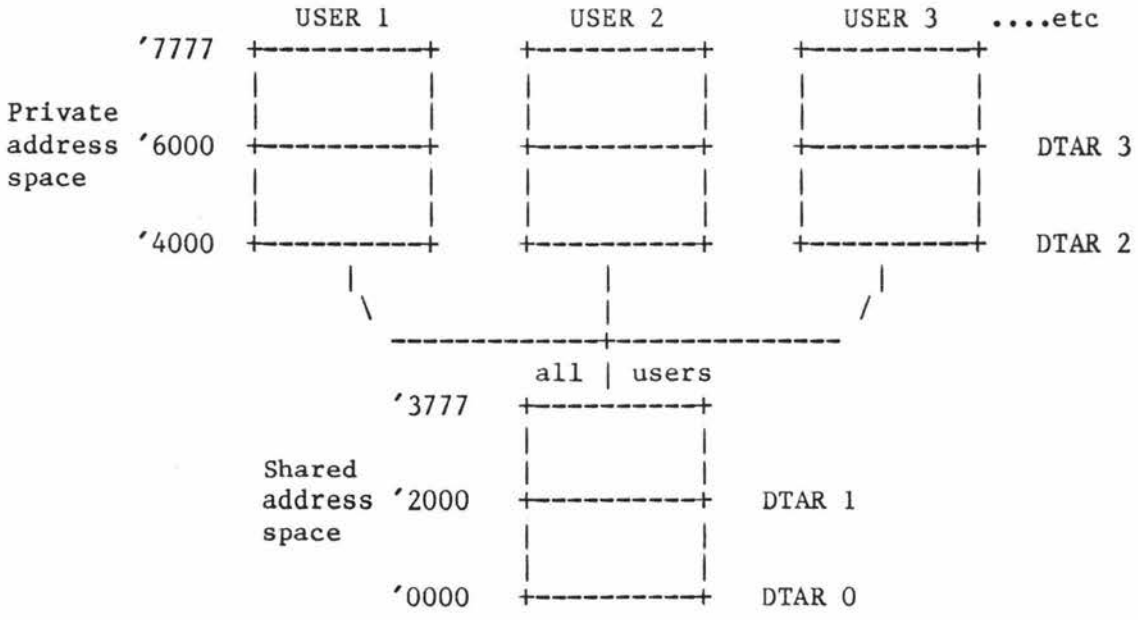
3.1 The address space

The virtual address space is made up of 4096 64Kw segments.

Unlike most machines which implement virtual memory the P750 requires the user to specify the segments which his program will use. Because of this static segment allocation, dynamic linkage of programs is not readily available.

With static segment allocation the virtual address space can be divided into four areas, each intended for a different use. The first 1024 segments are reserved for the operating system. The next 1024 segments are for other shared programs, editors, compilers and the like. All references to a segment in these first two areas (segments 1-2048) will access the same information. There is, however, a different copy of the other two areas (segments 2049-4096) for each user and these form

the users private address space which is used by the (nonshared) programs that he runs. Dividing the address space up into four areas, two of which are shared and two of which are private, is achieved by having base pointers to each 1024 segment block. These pointers are kept in the Segment Descriptor Table Address Register (DTAR). The upper two pointers are unique to each process and are saved and restored during process exchange.



Note: 'xxxx <=> xxxx(octal)

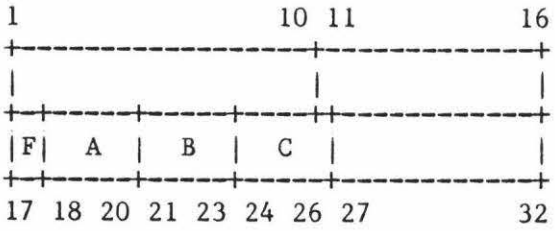
Virtual address space

For example user 3s address space is the lower block (segments 0-'3777) and the upper right hand block (segments '4000-'7777)

3.2 Address translation

3.2.1 Full address translation

The first two bits of the segment number (see Figure 3.1) are used as an index into the DTARs. Each DTAR points to a Segment Descriptor Table (SDT) in real memory that contains 1024 entries, where each entry describes one segment. The format for one of these Segment Descriptor Words (SDWs) is



Where

Bits 1-10 & 27-32 contain the real memory address of the page map table.

Bit 17 is the fault bit which, if set, means that the page maps for this segment do not exist.

Bits 11-16 are unused and are set to 0.

Bits 18-20, 21-23, & 24-26 contain the access rights to this segment from ring 1, ring 2 and ring 3 respectively.

The ring number in the virtual address defines which (if any) of the access permission bits are checked. If the reference is not an allowed

type then an ACCESS_VIOLATION$ fault occurs. If the fault bit is set then an ILLEGAL_SEGNO$ fault will occur.

Assuming that the access is allowed, the appropriate page table is located from the pointer in the SDW. The first 6 bits of the virtual address word number are used as an offset into the 64-entry page table. The format of entries in this part of the page table (or HMAP) is

Where

Bit 1 (V) This bit is set if the page is in memory.

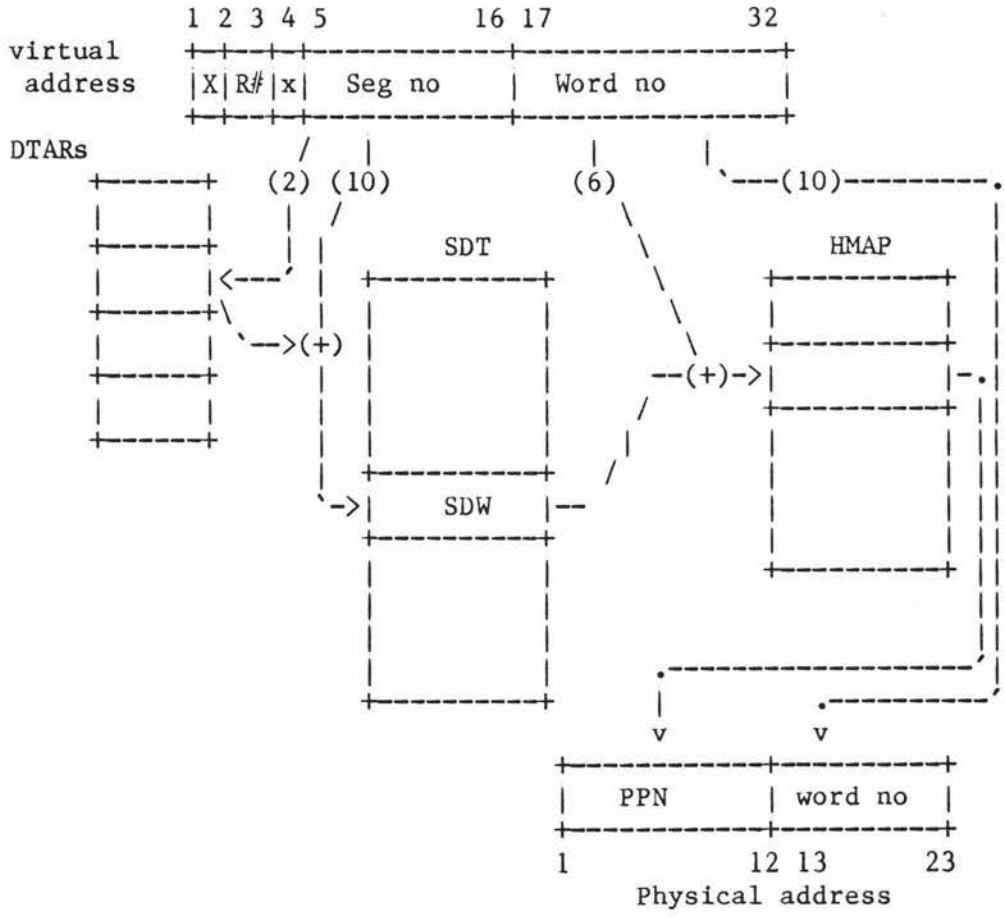
Bit 2 (R) This bit is set by the hardware when the page is referenced.

Bit 3 (U) This bit is reset by the hardware when the page is modified.

Bit 4 (S) This bit is set when the page is shared between processors, and inhibits the use of cache.

Bits 5-16 These contain the real memory address of the page (the physical page number).

If the V-bit is set then the page is in memory and the remaining 10 bits of the word offset in the virtual address are appended to the 12 bit physical page number in the HMAP to form a 22 bit real memory address. If the V-bit is not set then the page is not in memory and a page fault will occur.



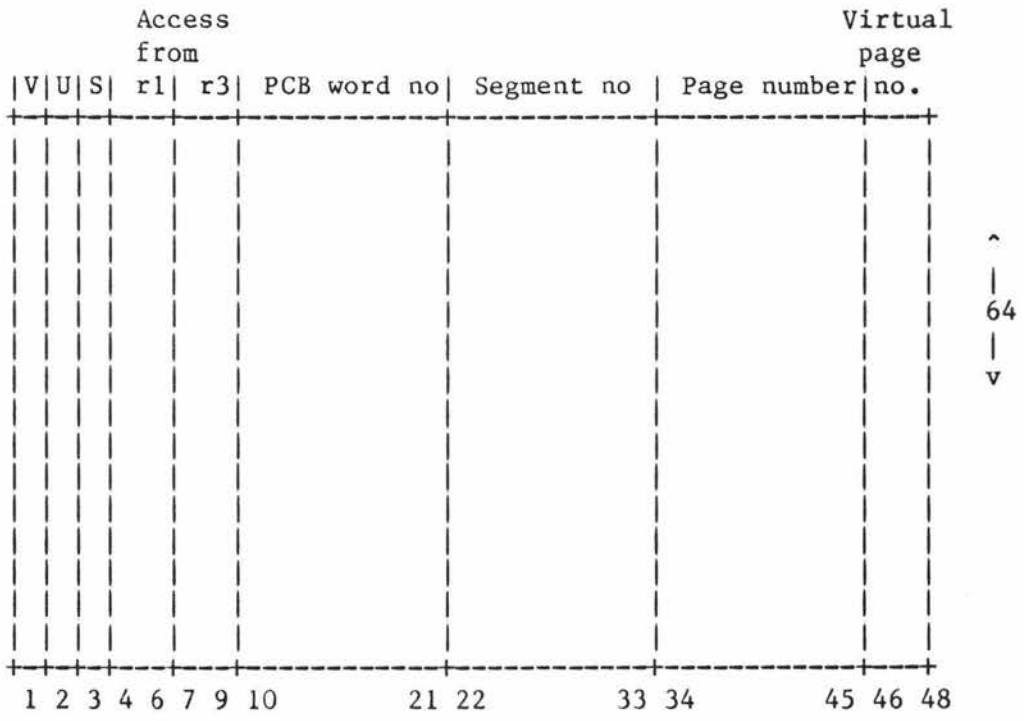
X = Unused (set to 0)
R# = Ring number (from current processor state)
DTAR = Descriptor table address register
SDT = Segment descriptor table
SDW = Segment descriptor word
HMAP = Page map
PPN = Physical page number

Full address translation
Fig 3.1

3.2.2 STLB and cache

The combined purpose of the Segment Table Lookaside Buffer (STLB) and the cache memory is to reduce the number of references required to real memory.

The STLB format is

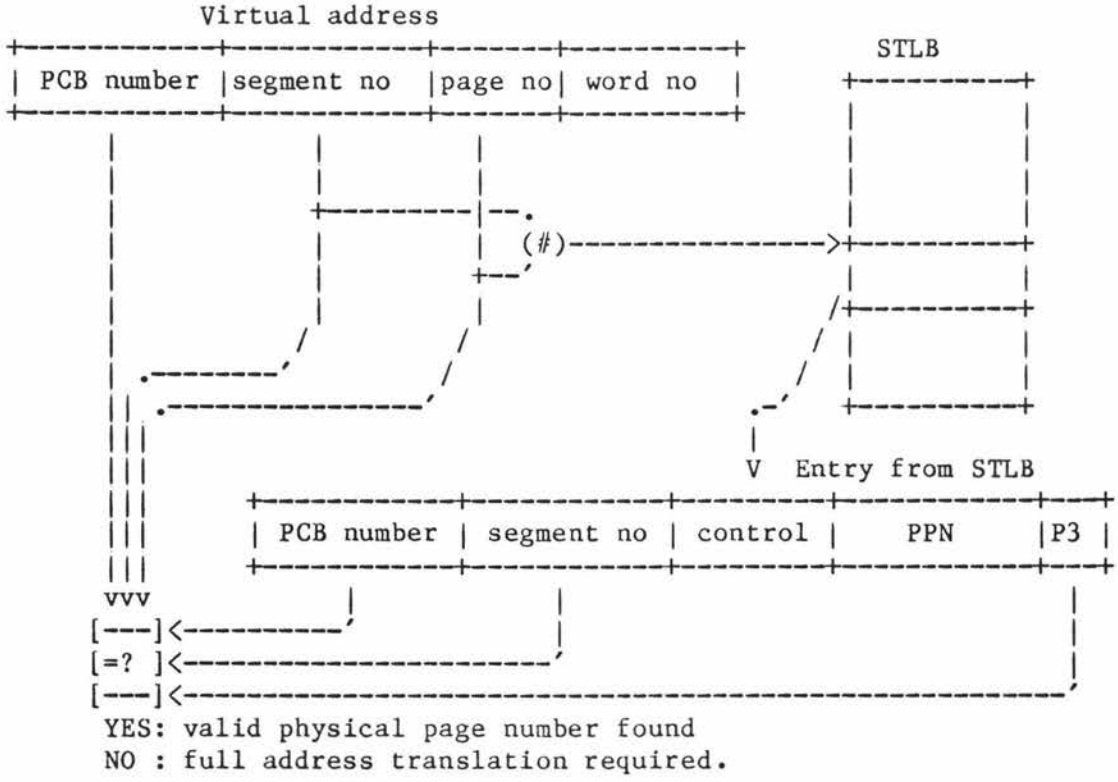


STLB format

Where

V = Valid Bit
U = Unmodified
S = Shared

When a memory reference occurs, the virtual page number and part of the segment number are applied to a hashing algorithm and an entry in the STLB is selected. The hashing algorithm is organised so that for any one segment each page will map to a different STLB entry. If the V-bit is set then the process number, segment number and 3 virtual page number bits in the STLB are compared with those for the current reference. If the comparison is successful then we are assured of having the correct physical page number.

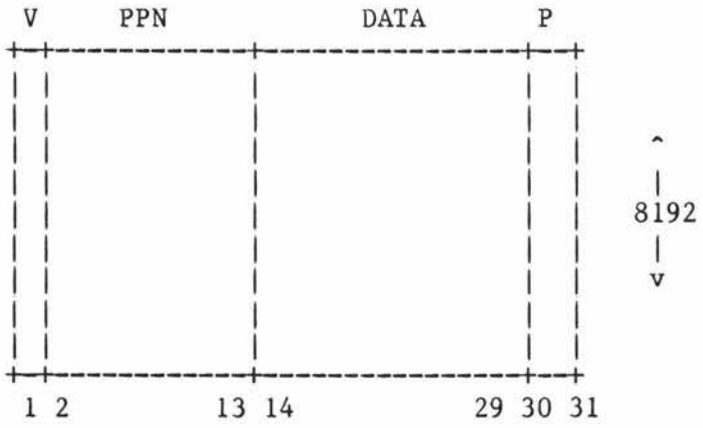


STLB function
Fig 3.2

At this stage the access permission is checked. The ring number in the processors current state is used to select one (or none) of the access

permission bits in the control information. Although ring two is not implemented in the STLB the operating system software is only designed to use rings zero and three so this is not a real restriction.

While the STLB lookup is going on an entry from the cache is also retrieved. The cache is organised as 8192 31-bit entries:

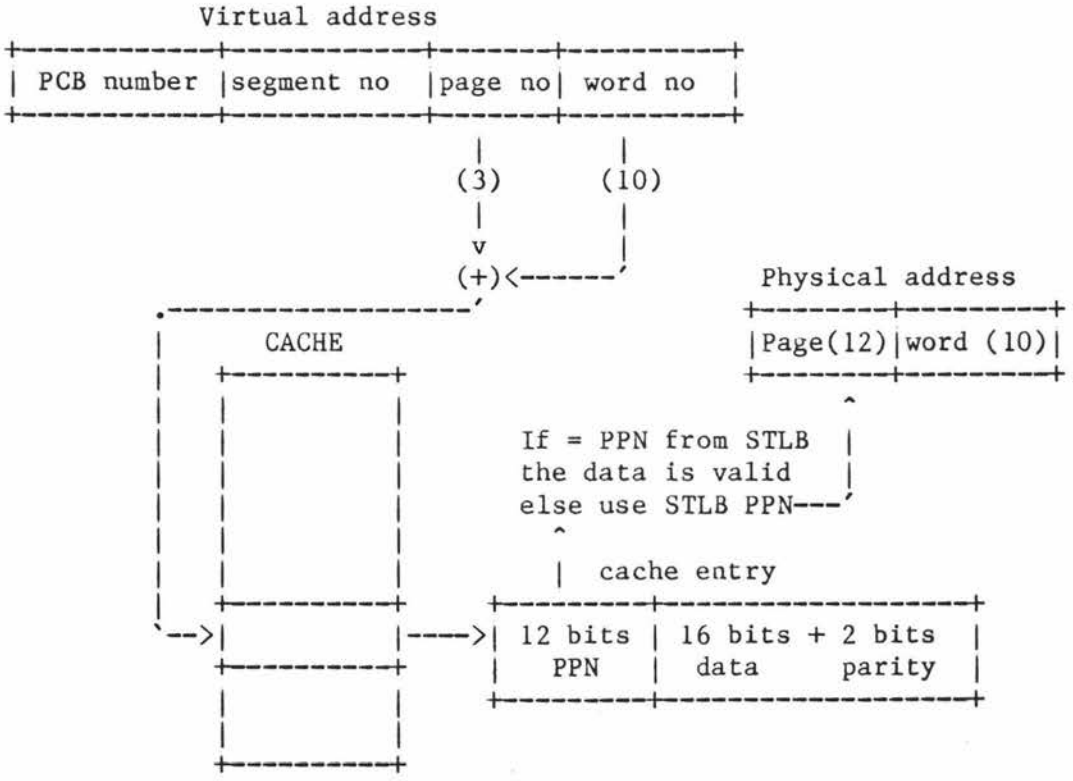


V = Valid Bit
PPN = Physical Page Number
DATA = Memory Data
P = 2 Parity Bits

The first bit is set when the entry is valid. The next 12 bits contain the physical page number of the entry and the remaining bits contain the data and parity information.

The word number and 3 least significant bits of the page number from the virtual address are used as an index into the cache. If the entry is valid it must be for a virtual address with the same word number as the current reference. Next the physical page number is compared with that retrieved from the STLB. If these are the same then the entry

must be for this process, this segment and this virtual page, and must therefore contain the desired data.



This cache scheme would be most effective if all words within the 'average page' were equally likely to be referenced. Because half filled pages are almost always filled from the lowest word this is closest to reality if there are few partly filled pages, i.e. the average job is much bigger than the size of a page.

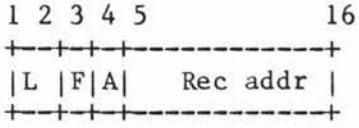
Whenever a page fault occurs, it is likely that the mapping between real and virtual pages will change. If this happens then the STLB will no longer contain valid data and so the STLB V-bit (bit 1) is reset to

0. This validity flagging is not necessary for the cache as only real (not virtual) addresses are used, and therefore no mapping is done.

3.3 Paging

Not all of the virtual pages can be in main memory at one time, but all pages used by a process, whether in memory or not, have an assigned position on the paging device(s). Two extra types of page table are used by the paging routine to keep track of the mapping between the paging device(s) and the pages in real memory. The first is the LMAP, which is similar in structure to the HMAP but contains disk addresses. Each LMAP is 64 words long and immediately follows the corresponding 64 words long HMAP. This means that for each HMAP entry giving the physical page number, if one exists, there is an LMAP entry 64 words away giving the disk address of the page.

The format for each LMAP entry is



Where

Bits 1-2	Contain a lock number if the page is to reside permanently in main memory.
Bit 3	Is set when the page is brought into memory to ensure that it is kept for at least one full cycle.
Bit 4	Indicates that the record is on the second paging device.

Bits 5-16 Contains the disk address of the 8-page block in which the page resides.

The last 12 bits give the address of a block of 8 pages on the disk in which the required page resides. The 3 least significant bits of the page number are appended to these 12 bits to give the actual 2048 byte record address. This provides a total capability of two 32Mb paging areas.

The second extra page table is the MMAP. This has one entry for each possible page of real memory. This table contains: a negative number if the page does not exist (ie. that memory is not installed); a zero if the page is available; or a process number (+ve) if the page is in use.

3.3.1 Paging-in

When a page fault occurs the value of the HMAP entry is tested to see that the page is not in transit. If it is, we wait until the current operation is completed and either return, if it was coming in, or continue with the page-in if it was going out. Next a counter is tested to see if there are any free pages. If there aren't then paging-out occurs.

Once there is a free page the LMAP entry is read to find the device and record number of the required page, and the MMAP is searched for a free

page. Once both these are found the new page is read into the space and the HMAP, LMAP and MMAP are updated.

3.3.2 Paging-out

When a page fault occurs and there are no free pages left in memory, at least one page must be moved out to its paging device. Normally a few pages are moved at once (3, by default, on current systems). The pages removed are determined using an approximate least recently used system called the Clock Scan Algorithm.

The referenced bit in the HMAP is central to this algorithm. This bit is set by the hardware when the page is referenced. During the paging-out all the HMAP entries are examined, in order, to see if their R-bit is set. If so, it is reset and the next page examined. Otherwise, as it is not set, the associated page has not been referenced since the last time it was examined and is paged-out. This procedure is modified slightly by the existence of the first-time-bit, which is set when a page is first brought into memory. When an MMAP entry is tested to see whether paging-out of the associated page should occur, the first-time-bit is examined and if set stops the page being paged-out. This mechanism inhibits a page being removed before it is accessed if its reference is tested soon after it is bought into memory.

4 Procedural architecture

The P750, like almost all modern computers, has been designed with a procedural architecture. The most important features of this architecture, which we will examine in more detail shortly, are

- o The three program areas.
- o The Procedure Call and Return mechanism.

4.1 Program areas

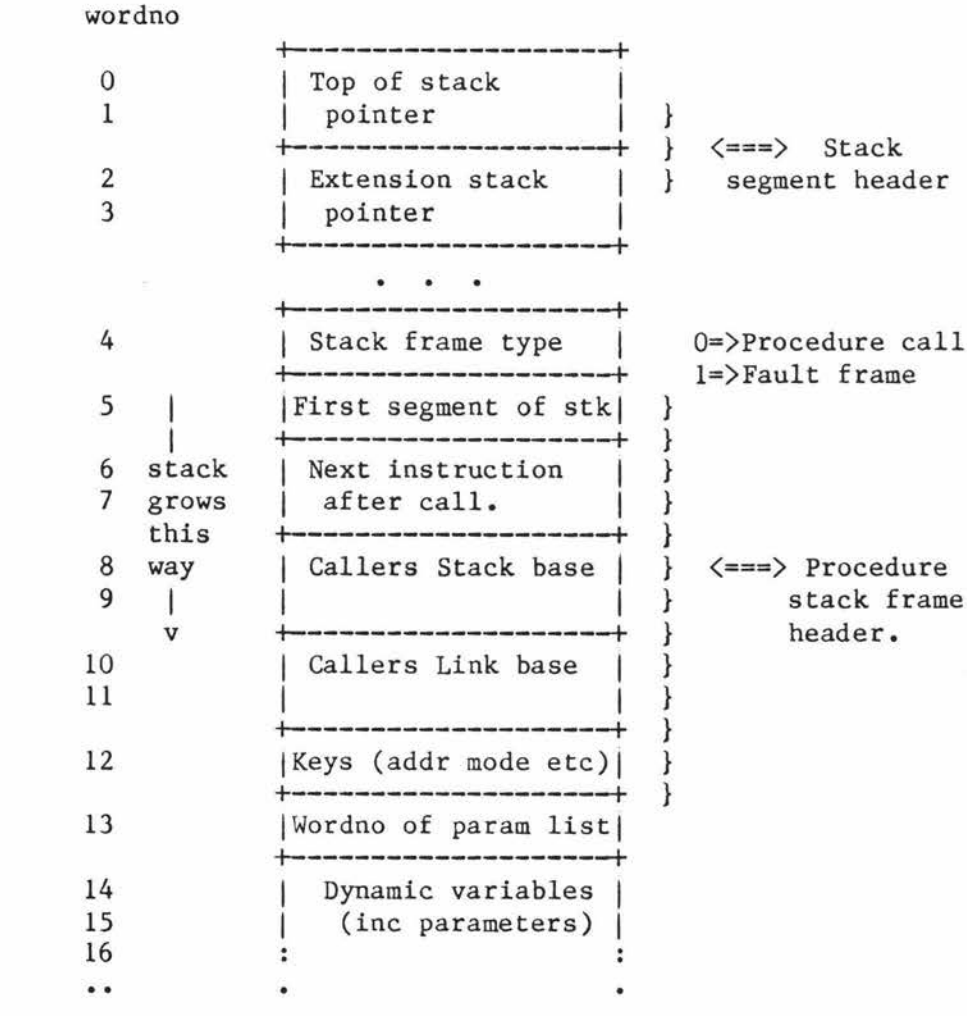
Three base registers are used to divide the memory accessible to a procedure into one code and two data areas. These registers are the Procedure base (PB), the Stack base (SB) and the Link base (LB) respectively.

4.1.1 Procedure area

This area is intended for pure code, and should be set up as read only. Unfortunately much of Primos which resides in this area is itself impure.

4.1.2 Stack area

The general format of the stack is a Stack Segment Header followed by a number of stack frames. A stack may share a segment with other data, in which case the stack should reside after the other data as there is no mechanism other than the segment top to limit the extent of the stack. The Stack Segment Header must still however reside at words 0 to 3 of the segment.



The stack format

Two types of stack frame exist, one for procedure calls and one for procedure faults.

Some confusion often occurs when considering stacks under Primos because of the static nature of segment allocation. As segments are allocated at the time a program is loaded each program must have its own stack, and so it is possible that a user may have a number of stacks at any one time.

4.1.3 Linkage area

Unlike the stack area the linkage area is not dynamic. It contains the statically allocated variables and the linkage information (hence the name) required to allow separate compilation of procedures. As this is not a pure area, because it contains variables, each user of a shared program has his own copy of the linkage area.

Example

Consider a small Pascal program which calls an assembler routine, as below

```
PROGRAM pcbs;

TYPE pcb_list_type = ARRAY[1..200] OF RECORD
                                        segno  :  integer;
                                        wordno :  integer;
                                      END;

VAR dump_record : pcb_list_type;

{$E+}
    outside : integer;          {this is an external variable}
{$E-}


PROCEDURE lokpcb(VAR pcblist : pcb_list_type);  EXTERN;


BEGIN
   lokpcb(dump_record);
END.
```

```
        SEG
        ENT   LOKPCB
        DYNM XWHERE(3)
        DYNM WAITED(3)

START   EQU   *
        ARGT
        CALL ASSUR$           *GIVE US A FULL TIME SLICE
        AP   ='515,S
        AP   WAITED,SL

        LDA  ='76303          *FIRST PCB - '77 + '2
        PRTN                  *RETURN

PCBPNT  DATA '00004
        DATA '76300
*
        LINK
LOKPCB  ECB  START,,XWHERE,1
        DYNT ASSUR$
*
        END
```

The program loader, provided by Prime to load the separately compiled modules of a program into one runfile, can also produce a load map describing where parts of the program (and associated data) exist. For the above program we could obtain

```
*START  4002  000004  *STACK  4001  001450  *SYM    000034

SEG. #     TYPE         LOW       HIGH       TOP
  4001     PROC##     001000    001447     001447
  4002     DATA       000000    002316     002316


ROUTINE          ECB         PROCEDURE    ST. SIZE  LINK FR.
PCBS    4002 000004   4001 001005   000056  000671  4002 177400
LOKPCB  4002 001566   4001 001024   000020  000026  4002 001166
F$ERX   4002 001616   4001 001130   000020  000026  4002 001214
P$AINT  4002 001646   4001 001172   000054  000120  4002 001242
P$AINI  4002 001766   4001 001213   000054  000160  4002 001362
P$AINO  4002 002146   4001 001333   000054  000155  4002 001542

DIRECT ENTRY LINKS
ASSUR$  4001 001064   P$ENTP     4001 001124    EXIT 4001 001156
TNOU    4001 001162   TNOUA      4001 001166

COMMON BLOCKS
```

```
P$AINP  4002  000672  000313  P$AOUT    4002  001206  000313
P$ASET1 4002  001522  000020  P$ASET2   4002  001542  000020
P$ASETI 4002  001562  000001  OUTSIDE   4002  001564  000001

OTHER SYMBOLS

  F191RETS  4001  001130
```

From the map we see that the stack starts in segment 4001 at word 1450 (4001/1450). The rest of seg 4001 (words 1000-1447) contains the procedure code. The linkage information and static variables are in segment 4002/0000-2316.

The main line (Pascal code) of our program has been loaded into 4001/1005-1023. Its link frame is from 4002/0-671. [Note that the address of the link frame in the map is wrong by octal 400. This is apparently done to maintain compatibility with the PM command which gives the LB register minus 400 after a crash. Its ECB is at locations 4002/0004-23. Each time the procedure is invoked (which will only be once as it is the mainline) it will require 56 words of space on the stack.

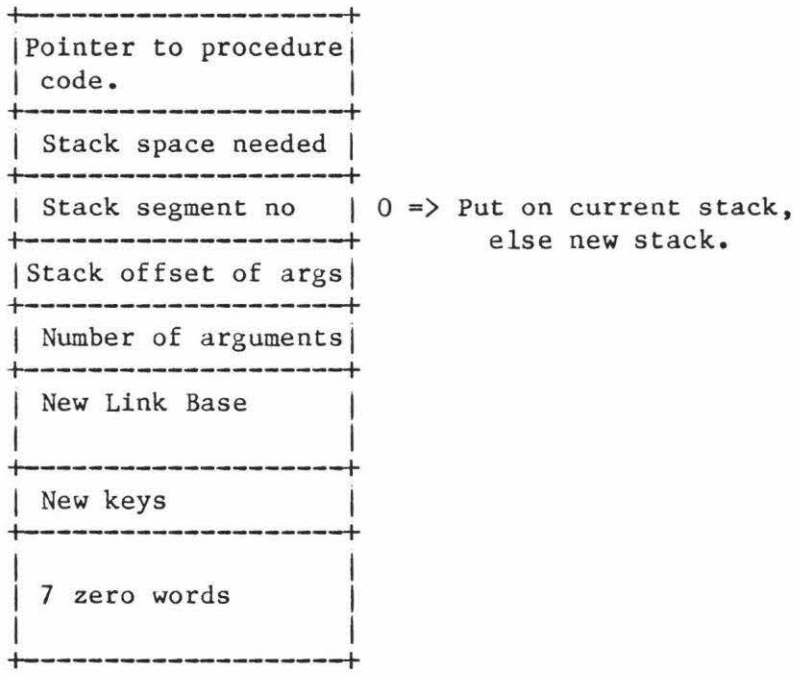
The code for the PMA (Prime Macro Assembler) part of the program follows the code for the Pascal part and takes up to word 4001/1063.

The items in the map beginning with P$A are Pascal library routines, and those beginning F$ are from the Fortran library (which are called from the Pascal library routines).

Finally, the element in the OTHER SYMBOLS section of the map is an unreferenced entry point, for which is included in the Fortran library to indicate what version of Primos and librarys were used in this load.

4.2 Procedure call mechanism

Procedures are described by an Entry Control Block (ECB). The format of this is



The general format of a procedure call is

```
PCL  <Procedure Entry Control Block Address>
AP   <1st argument>,S
AP   <2nd argument>,S
      . . .
AP   <Last argument>,SL
```

Execution of a procedure call proceeds as follows:

1) Space is allocated on the stack.

If the stack segment number in the ECB is 0 then the new stack frame is built on the current stack, otherwise it will be built on the stack whose base is in the segment number given in the ECB. Wherever the frame is to be built the segment is checked to see if it contains enough space to accommodate the new frame. If there is insufficient space then the extension segment, if one is given, will be used. If no extension segment is given then a STACK_OVF$ fault is generated.

2) The new stack frame is built.

assuming that the space is available, the new stack frame is built. The first word, which identifies the stack frame type, is set to 0. The caller's Stack Base (SB) and Link Base (LB) are saved, and the address of the location after the PCL instruction is saved as the return address (this will not be correct if the procedure has arguments but is corrected later).

3) The state of the called routine is set.

The PB, LB and keys registers are set to the values contained in the ECB{new_link_base} and ECB{pointer_to_procedure_code}. The SB is set to the base of the allocated stack area frame.

If ECB{no_of_arguments} = 0 then execution continues from PB+0. However if ECB{no_of_arguments} <> 0 then the first instruction of the called routine must be an ARGT instruction. This instruction specifies that the words following the PCL are to be interpreted as argument templates. They are evaluated into a set of pointers which are loaded onto the stack and become the parameters for the call. If there are more templates than ECB{no_of_arguments} specifies, the extras are ignored. If there are too few then extra arguments are created with their fault bit set. Once all the AP words have been read STACK{next_instruction} is set to the first word after the templates.

If an ARGT instruction was not the first instruction of a procedure containing parameters, the argument templates would not be read and (if the procedure returned) they would be treated as executable code. Similarly if an ARGT was included in a procedure which did not have arguments, then executable code would be interpreted as templates and not be executed on return from the called procedure (which would function normally if it expected zero arguments).

4.3 Procedure return mechanism

The last executable statement in a procedure will be a PRTN (procedure return) instruction. The effect of this instruction is to

1) Deallocate the stack space [STACK{tos} <- SB].

2) Set SB, LB, PB, and keys to the values in the stack frame STACK{tos}.

3) Continue execution from the instruction pointed to by PB.

Example

Below is shown the code for the previous Pascal/PMA program

Segment 4001 (Procedure code)

```
4001/ 1005 ( 3406)                          {
4001/ 1006 (  404)  EAL% LB%+ 404           {
4001/ 1007 (71432)                          {
4001/ 1010 ( 1267)  JSXB% LB%+ 1267,*       {
4001/ 1011 (61432)                          {
4001/ 1012 ( 1265)  PCL% LB%+ 1265,*        {  Pascal initialisation
4001/ 1013 ( 3400)                          {
4001/ 1014 ( 1015)  JMP%  1015              {
4001/ 1015 (    1)  NOP
4001/ 1016 (61432)                          {
4001/ 1017 ( 1261)  PCL% LB%+ 1261,*        {lokpcb(dump_record);
4001/ 1020 ( 1300)                          {          4002/661
4001/ 1021 (  441)  AP   LB%+ 441,SL        {
4001/ 1022 (    1)  NOP
4001/ 1023 (  611)  PRTN                    {end of main program
4001/ 1024 (  605)  ARGT                    {start of PMA program
4001/ 1025 (61432)
4001/ 1026 (  422)  PCL% LB%+ 422,*         {4002/1210 ie LB%+22}
4001/ 1027 ( 1100)
4001/ 1030 (  420)  AP   LB%+ 420,S
4001/ 1031 (  700)
4001/ 1032 (   15)  AP   SB%+ 15,SL
4001/ 1033 ( 4421)  LDA# LB%+ 421
4001/ 1034 (  611)  PRTN                    {end of PMA program
4001/ 1035 (    4)  DAC   4
4001/ 1036 (76300)  DIV# SB%+ 300,X
4001/ 1037 (    0)  HLT
```

In the linkage section (seg 4002) we find

```
4002/ 3 0          The ECB for the Pascal routine
4002/ 4 4001
4002/ 5 1005
4002/ 6 56
4002/ 7 0
4002/ 10 55
4002/ 11 0
4002/ 12 4002
4002/ 13 177400
4002/ 14 14040
4002/ 15 0
4002/ 16 0
```

```
4002/ 17 0
4002/ 20 0
4002/ 21 0
4002/ 22 0
4002/ 23 0

     ...

4002/ 661 4002       PCL indirect pointer to the ECB
4002/ 662 1566           for the PMA routine

     ...

4002/ 1566 4001      The ECB for the PMA routine
4002/ 1567 1024
4002/ 1570 20
4002/ 1571 0
4002/ 1572 12
4002/ 1573 1
4002/ 1574 4002
4002/ 1575 1166
4002/ 1576 14000
4002/ 1577 0
4002/ 1600 0
4002/ 1601 0
4002/ 1602 0
4002/ 1603 0
4002/ 1604 0
4002/ 1605 0
```

5 Process description and scheduling

The P750 is designed to support a multi-user environment. It is therefore necessary to share the available resources among competing users. The CPU is often considered to be the scarcest resource in the system and special attention is given to its allocation. This section describes how the CPU is allocated.

There are a fixed number (defined at system initialisation) of processes in the system. Associated with each process is a Process Control Block (PCB). Each of these PCBs is linked, at all times, into the ready-list or one semaphore queue. When the CPU is to be allocated to a process it is assigned to the most eligible member of the ready-list. Processes move from the ready-list to a semaphore by executing a wait operation on that semaphore (this action may be forced upon the process after the exhaustion of a time slice). The most eligible process is moved from a semaphore queue to the ready-list by some other process (possibly the scheduler) signalling that semaphore.

5.1 The Process Control Block

The PCB contains the information relevant to scheduling a process for CPU usage. All PCBs are 100 (octal) words long and located together in segment 4.

Each PCB contains :

1) The priority of the process.

2) A link used to chain PCBs together into lists and queues.

3) A pointer to the header of the most recent semaphore waited on.

4) Pointers (DTAR2 & 3) to the process' private memory area.

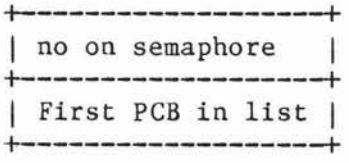
5) The saved processor status when the process is not using the CPU.

6) Fault vectors, interval timing and other information concerned with process exchange.

5.2 Semaphores

A semaphore consists of a two word header and a list of PCBs waiting on that semaphore.

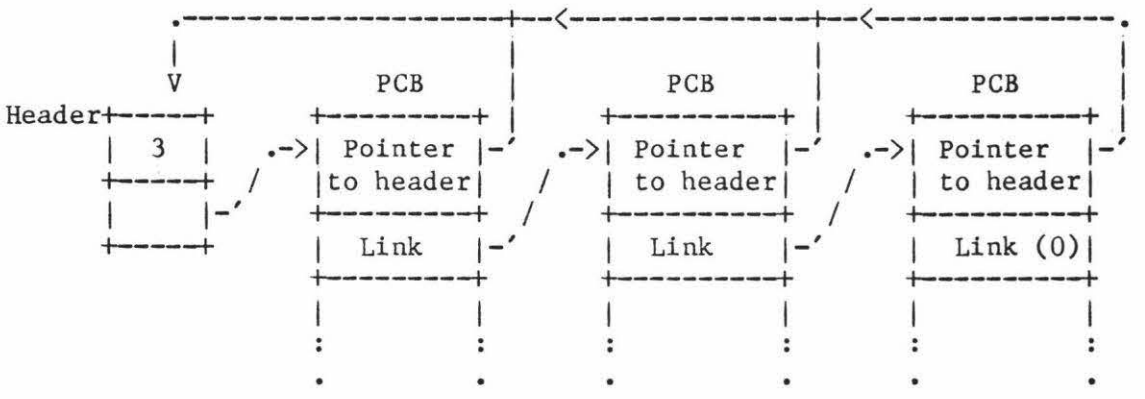
The header has the form



Semaphore header

The first word is a count of the number of processes waiting on this semaphore (or if negative the number of resources available) and the second is a pointer to the first PCB on the queue. Only the word offset is used as all PCBs reside in segment 4.

The linked list is constructed using the link field (1 word) in each PCB. Each PCB also contains a 2 word link back to the semaphore control block (which may be anywhere in memory):



PCBs are linked into the semaphore list and/or the header is updated when a WAIT instruction is executed by a process. There are two types of WAIT instruction and the list is maintained as a priority queue (based on the PCB priority) in both cases. The first type causes the process to be linked into the queue as the last process with this priority (priority FIFO); the second causes the PCB to be linked into the queue as the first process with this priority (priority LIFO). As the waiting process must have had the CPU at the time of the WAIT and no longer needs the processor, a WAIT operation causes the dispatcher to reallocate the CPU.

PCBs are removed from the semaphore and/or the header is updated when a process executes a SIGNAL operation. The SIGNAL causes the process at the head of the semaphore queue to be moved to the ready-list. The CPU will again be dispatched, causing the transfer of the CPU to the

signalled process if (and only if) it is of higher priority than the signalling process.

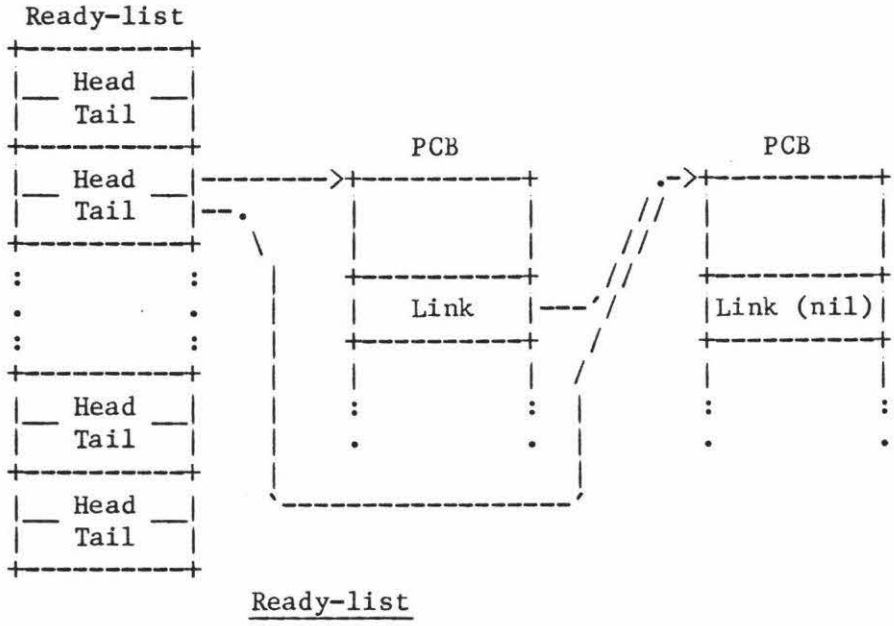
5.3 The Ready-list and the Wait-Lists

The principle structures involved in scheduling the CPU between eligible processes are the ready-list and the wait-lists (described below).

Primos attempts, where possible, to maintain a 'better than random' mix of work immediately eligible for the CPU. The PCBs for these processes are linked into the ready-list. Other processes waiting for the CPU are excluded from the ready-list and held on the wait-list appropriate to their priority and recent history of CPU usage. Processes are moved from the wait-lists to the ready-list by a process known as the scheduler (because of this activity although it does not dispatch the CPU, see section 5.4) or the backstop (because it has the lowest possible priority and is always ready to accept the CPU).

5.3.1 The ready-list

The ready-list is implemented as an array of pointer pairs, with one pair for each allowable priority. PCBs with the same priority are chained into a list through their link fields. The top element points to the first PCB on the ready-list with this priority and the second points to the last PCB on the list with this priority:



5.3.2 The wait-lists

Each wait-list is a simple semaphore and the only distinction between it and normal semaphores is its use. There are seven wait-lists associated with (non operating system) process scheduling. These are

1) HIPRIQ
2) ELIGQ
3) LOPRIQ (USER 1)
4) LOPRIQ3 (USER LEVEL 3)
5) LOPRIQ2 (USER LEVEL 2)
6) LOPRIQ1 (USER LEVEL 1)
7) LOPRIQ0 (USER LEVEL 0)

The algorithm for mmoving jobs from the wait-lists to the ready-list is given in section 5.6.

5.4 The Dispatcher

When the process currently using the CPU executes a WAIT or SIGNAL operation the dispatcher is invoked to reallocate the CPU. The dispatcher assumes that the most eligible process is the process at the head of the first, in priority order, non-empty list in the ready-list.

Registers are used to point at the next process in line and so remove the necessity to re-scan the entire list when the CPU is dispatched. The semaphore instructions and the dispatcher are implemented in firmware and appear as single instructions.

5.5 SCHED

A single routine (known as SCHED or the scheduler) has the responsibility of allocating time slices and, if necessary, suspending execution of a process on one of the wait-lists. (The conflict of names with the backstop/scheduler process suggests that it is wise to avoid the latter name. We will use SCHED.)

SCHED is called when the end of a time slice is detected, and also at various points within the ring 0 part of the operating system when it is necessary to reallocate part, or all, of a time slice and/or cause the process to wait on a wait-list.

5.6 The backstop

As mentioned in section 5.3, the backstop moves processes from the wait-lists to the ready-list. The algorithm for this is as follows.

```
Repeat forever

    If any jobs on HIPRIQ
       Then move one of these

    Else If any jobs on ELIGQ
       Then move one of these

    Else If any jobs on LOPRIQ
       Then If count < max
                Then move one of these
                     count := count + 1
                Else count := 0

    Else If any jobs on LOPRIQ3
       Then If count3 < max3
                Then move one of these
                     count3 := count3 + 1
                Else count3 := 0

    Else If any jobs on LOPRIQ2
       Then If count2 < max2
                Then move one of these
                     count2 := count2 + 1
                Else count2 := 0

    Else If any jobs on LOPRIQ1
       Then If count1 < max1
                Then move one of these
                     count1 := count1 + 1
                Else count1 := 0

    Else If any jobs on LOPRIQ0
       Then If count0 < max0
                Then move one of these
                     count0 := count0 + 1
                Else count0 := 0

End-until
```

The values of max, max3, max2, max1 and max0 are compiled into the operating system and are currently set at 16,8,4,2 and 1 respectively.

Because the backstop has the lowest possible priority after each job is signalled, the backstop will lose the CPU and the signalled process will continue. Once that process, and any others it signalled during its execution, have waited on a semaphore (possibly as the result of a call to SCHED at the end of their timeslice), the backstop will get the CPU again and will SIGNAL another process (if one is ready to run) or loop (if none are ready).

5.7 Operation of the scheduling system

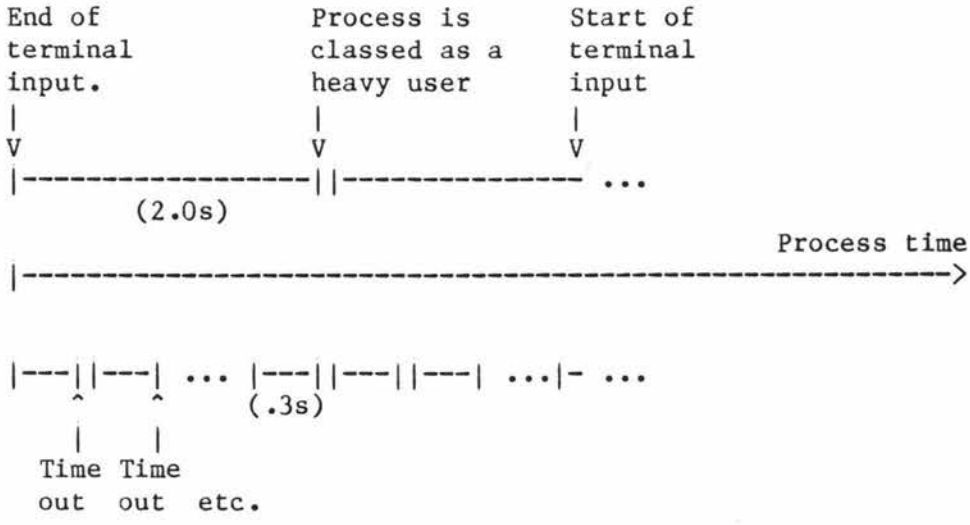
Primos attempts to provide good service to light users of the CPU, without unduly effecting service to heavy users. 'Unduly' means that if any request for the CPU loses out it will be for a heavy CPU user, but this should not be to such an extent that the process receives no service at all.

A light CPU user is defined as one who uses less than a (preset) amount of CPU time between terminal inputs. A heavy user is any process which is not a light user.

To understand in more detail the way the scheduling system is implemented, it is convenient to first examine the scheduling of light users and then go on to examine the way heavy users are scheduled within this environment.

5.7.1 Light CPU users

Consider a process entering the scheduling system from the command line processor on the detection of the end of a command which is signalled by a carriage return. The command line processor calls SCHED which places the process on the HIPRIQ. Two types of timeslice are allocated to the process. The first is the major time slice, which is 2 sec by default, and the process must wait on terminal input within this time if it is to be considered a light user throughout its execution. The second type of time slice, the eligibility timeslice (1/3 sec by default), is provided to ensure light users get regular service from the CPU. This relationship is shown in fig 5.1.



The relationship between the minor and major time slices
Fig 5.1

When the backstop process next moves a process from the wait-lists onto the ready-list it will move any processes on the HIPRIQ first. The process will remain on the ready-list until it executes a WAIT operation. During this time the CPU may be allocated to another process if the current process signals a semaphore where the first process on the queue is of higher priority than the signaller. The suspended process will remain on the ready-list and will, eventually, be given back the CPU.

Process exchange at the end of the eligibility time slice occurs as follows. One word in the PCB is used as a process timer and this location is copied into and out of a register during process exchange. Every 1024 microseconds this register is incremented. When it overflows, a second register is set with a fault code indicating that a process fault has occurred and execution is transferred to the fault handling mechanism (ie. a fault occurs). During the handling of a process fault the CPU time used by the process is added to that already accumulated, and the process re-enters the scheduling system by calling SCHED. SCHED will place the process on the eligibility queue (ELIGQ).

The backstop will move processes from the ELIGQ queue if there are no processes waiting on the HIPRIQ.

Eventually the light user will again wait on terminal input and SCHED will reallocate a new major timeslice.

5.7.2 Heavy user

If a process uses more than 2 seconds of processor time between successive terminal inputs the major time slice will be exhausted. When this occurs the process is considered a heavy user of the CPU, and at the end of the eligibility time slice SCHED will place it on one of the LOPRIQ queues instead of the ELIGQ. The queue used is LOPRIQ for user 1 (the console) and LOPRIQ's 3, 2, 1 and 0 for processes with priority 3,2,1 and 0 respectively. These queues are serviced by the backstop if there are no (light user) processes waiting on the HIPRIQ or the ELIGQ.

6 The file system

This section discusses the tree structured directory system and the file types supported by Primos. It does not consider packages (such as MIDAS) which are designed to sit on top of the operating system.

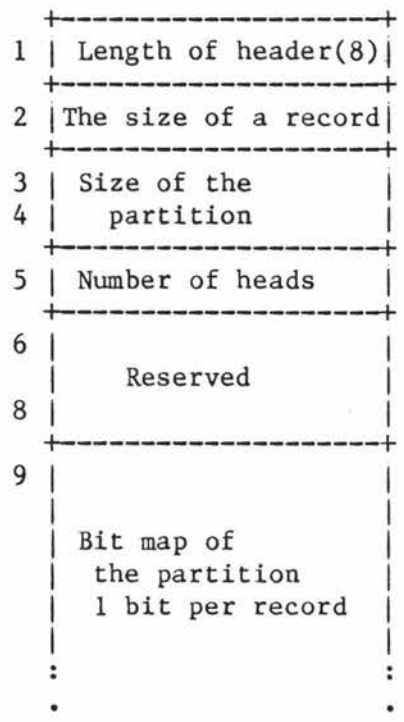
6.1 The file system structure

The file system, at the top level, consists of a number of disk partitions. A partition is a user selected set of adjacent heads on a disk drive. Each partition contains at least

1) A Master File Directory (MFD); the root of the file system.
2) A Disk Record Availability Table (DSKRAT).
3) A boot sector for initial loading of Primos (this is always at track 0 record 0).
4) Two User File Directories (DOS and CMDNCO) for system use.

6.1.1 The Disk Record Availability Table

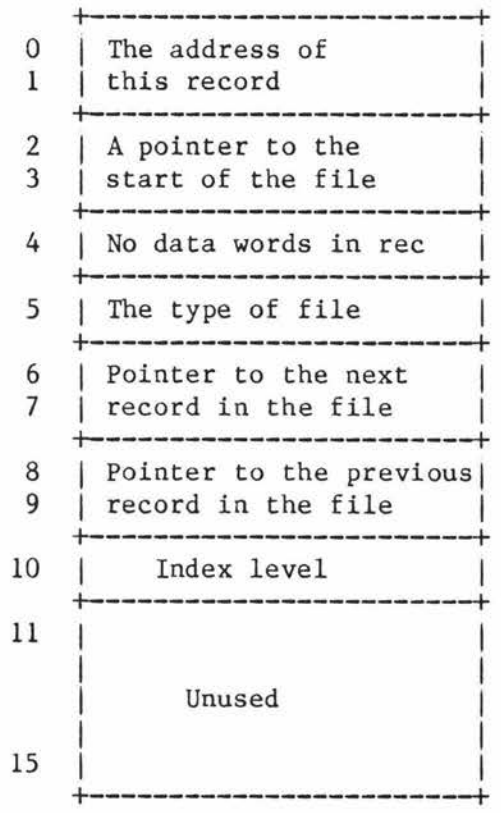
The DSKRAT has the following format:



6.2 Files

There are two basic types of files and three derivatives of these. The basic types are Sequential Access Method (SAM) files, which contain the control information necessary to move forwards and backwards a record at a time, and Direct Access Method (DAM) files, which contain in addition to the SAM structure an index that allows positioning to an arbitrary record with only a few reads. The same operations can be performed on both types of file, the only difference to the applications programmer being the efficiency.

Each record in a file contains a 16-word header with the following format:



The record address of this record is included to make the record complete on its own. For example it is not necessary to pass the record and its address to WREC to have it written to the disk.

The pointer to the start of the file points to the first record of the file in all records except the first, where it is used to point to the directory for the file.

The field indicating the type of file is only valid in the first record of the file. It has the following values

0 ... SAM file
1 ... DAM file
2 ... SAM segment directory
3 ... DAM segment directory
4 ... User file directory (UFD)

If the file is BOOT or DSKRAT then bit one of the record type will be set.

The forward and backward pointers will be 0 if the current record is the last or first record, respectively, in the file.

The index level is only used in DAM files, where it indicates the number of index levels finer than the current one. More detail on this is given in section 6.4.

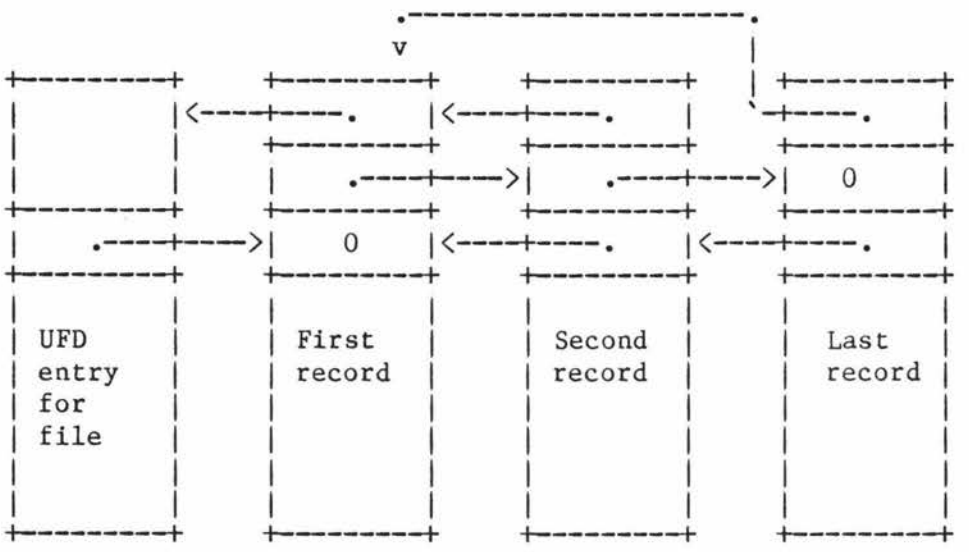
6.3 SAM files

Sequential Access Method files are optimised for space efficiency and sequential access but not for unsequenced accesses.

6.3.1 Organisation

The records in a SAM file are linked together by the forward and backward pointers, and each record is linked to the start of the file by the information kept in the record header. There is no direct

mechanism provided to position the file pointer.



SAM File organisation

6.3.2 Accessing a record

When the file is to be positioned to an arbitrary record the distance from the current record is computed and the file pointer moved, one record at a time, to that position. If the end of the file is struck an error code is returned and the file pointer remains at the end of the file. The read and write operations work with the data immediately following the file pointer, that is a read of 1 word will increment the file pointer then read a word from where it points.

6.3.3 Adding records

If during a write operation, the file pointer passes the current end of the file and there is not enough free space in the last record of the file to accommodate the data to be added, a new record will be appended to the end of the file. This requires the new record to be obtained, the data and header to be written to it, and the previous end of file record to have its forward pointer updated to point at the new record.

6.3.3.1 Obtaining the new record

When a new record has to be obtained the Disk Record Availability Table (DSKRAT) is used. The bit map in the table is searched for a free record starting at the next word boundary which is numerically greater than the current word. If the end of the map is reached before a record is found, then the search continues from the start of the table. If no free record is found then a 'disk full' error occurs. If a free record is found the quota information for all ancestor directories is checked and assuming no quota is exceeded, the record is allocated.

6.3.3.2 Truncating the file

The file may be truncated from the file pointer to the end of the file. This is a simple enough operation to perform using the forward pointers of the header and resetting the appropriate bits in the DSKRAT.

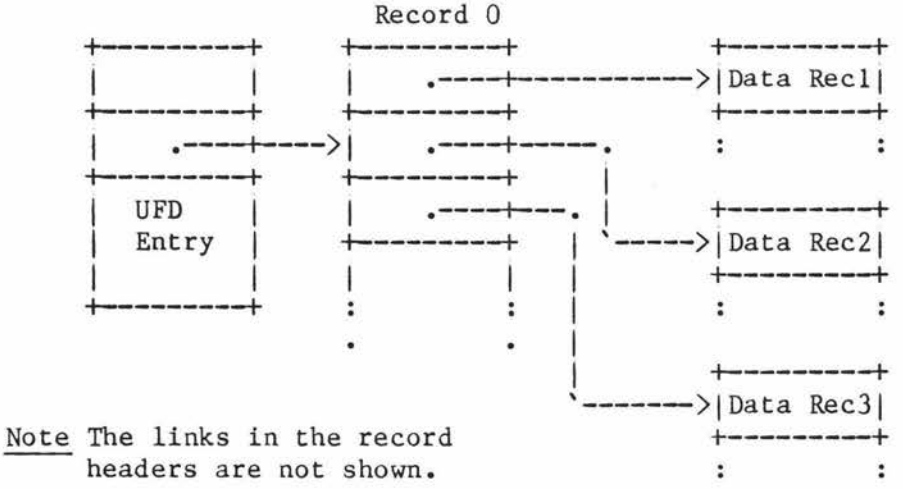
6.4 DAM files

Direct Access Method files are optimised for unsequenced access, but not for sequenced access or for space.

6.4.1 Organisation

Each record in the body of the file is linked to the next, the previous and first record of the file in exactly the same manner as for a SAM file. In addition to this a variable level index mechanism is maintained.

At initial creation, an index record is created as record 0 of the file. This record will always remain the top level index of the file. A table of record addresses is maintained in the data portion of the index records.



Note The links in the record headers are not shown.

Small DAM File

For most files (of less than around 500KW for storage module devices) this will be the only index required. If however the file grows sufficiently large for the index record to become full, a new level of index will be created.

With a two level index the first record of the index (record 0) will contain pointers to a second level of index records, and these will contain the pointers to the records in the file. If the second level of index has just been created then the entire index structure will consist of three index records. Record 0 (the second, or coarsest, level index) will contain two pointers to first level index records. The first of these records will be full of pointers to data records and the second will contain a single pointer to the last record of the file.

Should this second level index become full (a somewhat unlikely event requiring a file of approximately 250 MW) then a third level of index will be created. Just after its creation this structure will consist of record 0, the third level of index, containing two pointers to second level index structures. The first of these will be full of pointers to first level index blocks, each of which will be full of pointers to data records. The second of the second level index blocks will contain a single entry to a first level index block which will contain a single pointer to a data record.

Were it possible to create a sufficiently large file on a single device the above scheme would extend to a further index level.

This structure is similar to a balanced B-tree.

6.4.2 Accessing a record

When it is desired to access a particular record in the file, the system begins by checking to see if the current first (finest) level index block contains the address of the new record. If it does then it is retrieved, the address of the new data record is extracted, and the data record is fetched from the disk. This requires two records to be accessed even if the record required is the next record in sequence from the previous one read.

If the required record is not in the current first level index then its location is found by full indexing from the top level. At each level a pointer is chosen using the algorithm

```
word in current record =
    record number MOD [ (data words in rec / 2) **  (level - 1) ] * 2
```

The final pointer will yield the data record itself.

6.4.3 Adding a record

When a record is to be added to the file a calculation is made first to see how many, if any, new index blocks will be required. The new records will be obtained before anything else is done. This is to ensure that, in the event of a 'disk full' error occurring, the file can be left in its original state and, once some space has been made available, the program can be restarted using the START command.

If no new index block is required, the record will be added and a pointer to the record will be placed in the current first level index block. Otherwise new records will be added to the index as far back as is necessary and the appropriate pointers updated (the code for this imposes a practical limit of four index levels). In addition the current first level index pointer of all other file units with the file open will be invalidated to force full indexing for the next read. This ensures that the record is not missed.

If a new top level index is required then the current top level index will be copied to the next level of the index, rather than just relinking the record. This ensures that record 0 remains the beginning of the file, so that the other records in the file do not have to be updated with a new beginning of the file address.

6.4.4 Truncating the file

If the file is truncated, the records are removed from the main body of the file in the same manner as for a SAM file and returned to the pool of available records. The index entries which pointed to the released records are removed from the index records, and any index records which are emptied, except the first on each level, are also released. An entire level is never discarded as this would cause undue disk access if a file size fluctuated around a length that required the creation of a new index level.

6.5 Segment directories

Both SAM and DAM files are sequential files in that the data in the file has a well defined sequence and there are no gaps between allocated records. If it is necessary to have a record number 1000 then records 1-->999 must also be allocated. If it is wished to maintain a file which has a sparce distribution of records these will not be very efficient mechanisms. The requirement to store a program

for use in a segmented address space presents just such a case and is the main use of the segment directory.

A segment directory has the structure of a normal SAM or DAM file, but the information stored within the file is a sequence of record addresses. Each record address, which is two words long, points to the record address of the start of a subfile. The subfile can only be accessed through its entry in the segment directory, and may be a SAM file, a DAM file, another segment directory, or null (no file). It cannot be a User File Directory (UFD).

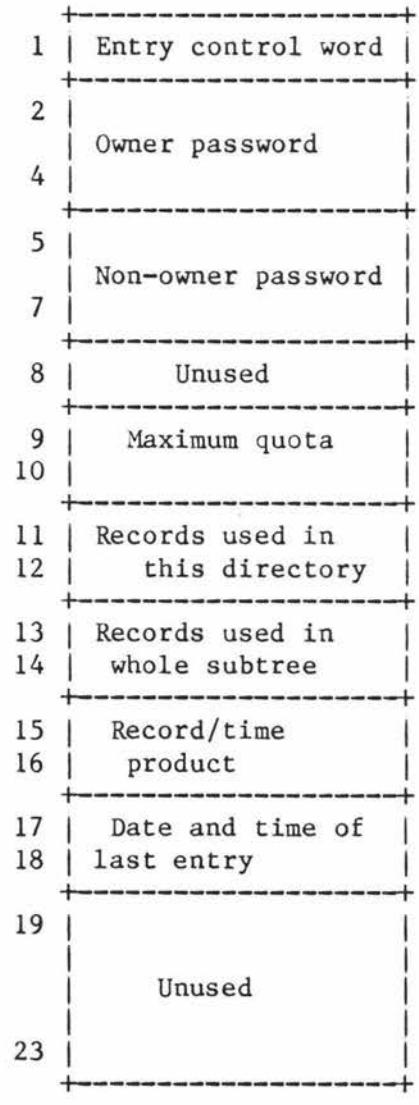
Special operations on the segment directory will open or close a specified subfile. Once opened a subfile can be treated just like any other file.

This method is similar to that of dope vectors.

6.6 User File Directories

A UFD is a SAM file which contains data that describes a subgroup of files.

The UFD begins with a header whose format is:



The entry control word is divided into two subfields. The first 8 bits are 00000001, specifying that this is a directory header. This is redundant as it is known that the first entry in a directory is the header, but it does keep the format consistent with other directory entries. The second 8 bits specify the length of the header.

The rest of the file is taken up with UFD entries. These describe files, directories, access categories, and access control lists.

7 The Input Output Control Section (IOCS)

As we have seen, the file management system requests disk records to be fetched from or written to the disk subsystem. Requests also originate from the paging routines and other routines accessing the disk from outside the file system.

We consider the IOCS to be that part of the operating system which takes, and eventually satisfies, these requests.
It is involved with

- o buffering the most recently used records,

- o scheduling disk accesses to efficiently use the disk,

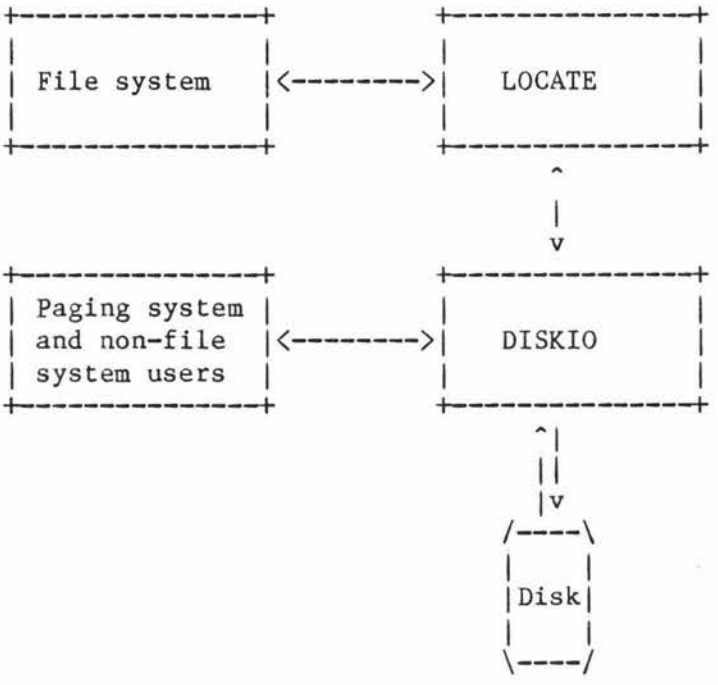
- o handling the physical to real translation between a number of possible physical devices and the internal representation of the data and

- o controlling the execution of those processes waiting for requests to be satisfied.

The IOCS can be considered as being made up of two functional blocks, each of which is implemented as a separate module (source file). The first is the LOCATE module, which is responsible for all requests originating from the file system and contains the buffers used to reduce the number of disk accesses for frequently used records.

The second is the DISKIO module. This is responsible for accepting requests to be satisfied from the actual physical devices. The requests mainly originate from the LOCATE module (when it does not contain the record requested), and from the paging system. Other accesses arrive here from any use of the disks which might occur outside of the file management system.

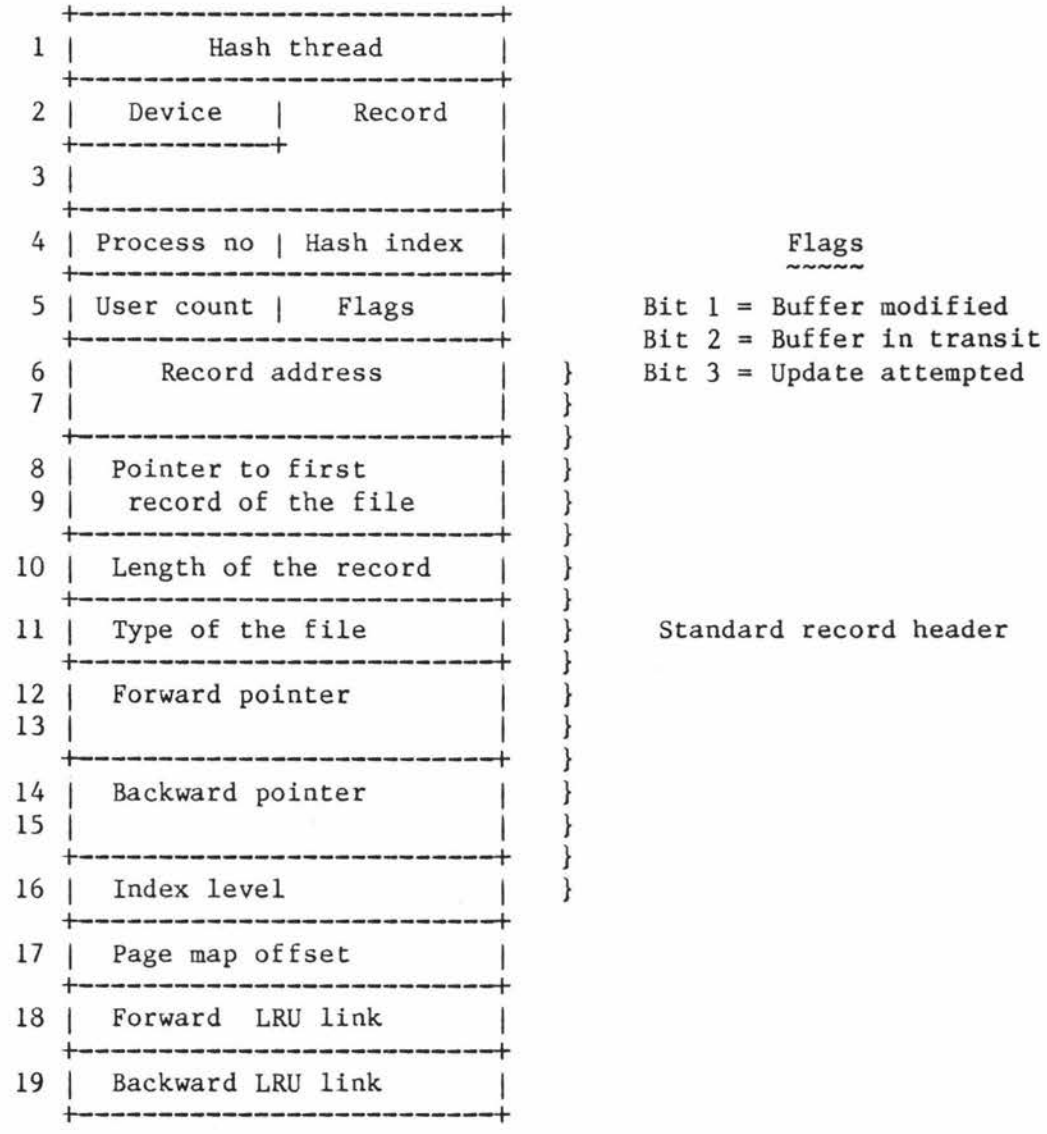
The following describes the general structure of the IOCS:



Functional blocks of the IOCS

7.1 LOCATE

LOCATE maintains internal buffers which hold the records it has most recently been asked to find. The following data structure is used to keep track of these buffers:



The maximum number of LOCATE buffers is defined at cold start time and can be configured from 8 to 256. One header (as shown above) is kept for each configured buffer.

7.1.1 The LOCATE interface

When LOCATE finds a record for a process it sets a pointer to the header for the buffer in which the record lies. The pointer (NEWBUF) is dedicated to this purpose and is contained in the work area describing the current state of the user (PUDCOM). By 'fiddling' with the page maps, the buffer is mapped into both the user's private address space in segment 6000 and permanent buffer space in segments 50-53. This mechanism means that any user may hold at most one LOCATE buffer at a time. The user's process will keep this buffer until it is specifically released, or another is requested.

In the Prime file management system more than one user may have concurrent access to the same file. The significance of this to the LOCATE mechanism, is that there may be more than one user wishing to access the same record at any time. To accommodate this with the least possible conflict, LOCATE allows multiple owners of a buffer. Two users sharing a buffer will have the same value in their NEWBUF pointer, and their buffer area in segment 6000 will map to the same address space in the LOCATE buffer area. The user count in the buffer descriptor is used to keep track of the number of concurrent users of the buffer.

7.1.2 The unowned buffer structure

When the last user relinquishes the buffer, either explicitly by calling FORGET or implicitly by a subsequent call to LOCATE, the buffer is linked into a least recently used list of unowned buffers. When a process requests a record that is not currently owned by any process, a new buffer must be allocated to accommodate the record once it is read in from the disk. The least recently used buffer is unlinked from the list and assigned to the calling process.

7.1.3 The hashing algorithm

To allow speedy access to the buffers a hashing algorithm is used to place each record into a small 'bucket' of buffers. To avoid having a much larger target space than the expected number of objects to be stored (as is the case with most hashing algorithms) a thread is used to link buffers which hash to the same value. This resolves clashes without requiring any rehashing.

The hash is performed by the following instructions

```
*
     LDL   RADEV     A <- device no, B <- record number
     ICL             A(9-16) <- A(1,8); A(1-8) <- 0
     ERA   RADEV     Exclusive-OR A with the device number
     ANA   ='37      AND A with '0000000000011111b'
     DIV   =HTSIZE   Divide A by the size of the hash table.
     TBA             Swap A and B (product and remainder)
     STA   HASHX     Save A as the hash table index
*
```

This algorithm gives us an index into a table of pointers known as the hash table. Each pointer is the start of a thread containing all the records currently in the LOCATE buffers which hash onto that hash table entry. The thread is continued through the hash thread field in the buffer descriptor.

7.1.4 Locked down memory

For the sake of speed, the buffer headers are kept in main memory at all times, but the buffers themselves are not normally locked down. The module which fetches records from the disk (DISKIO) cannot handle recursive calls. This means that a page fault must not occur while the user is in DISKIO. To ensure that this is the case, LOCATE will lock down a buffer before it is written to, or read from, the disk. This is done by setting the lock bits in the LMAP to stop the page from being paged out, and then referencing the page to ensure that it is in memory before the call. At the end of the call the L-bits are reset and the page can again be paged out.

The subset of buffers which are owned by processes are expected to be referenced and are kept locked down.

7.1.5 The LOCATE algorithm

When LOCATE is asked to find a record, it first tests to see if the record required is in the buffer already held by the requesting process (if it holds a buffer). If the process does hold a buffer, and the record is not there, the buffer will be released and is added onto the end of the LRU list.

Next the hashing algorithm is used to try to find the record in the buffers. If it is found it may be unowned, and in the LRU list, or it may be owned by another process. In the first case it is unlinked from the LRU list and assigned to the user requesting it. If the buffer is already owned, the owner count is incremented and a copy of the pointer to the header and the pagemap entry are taken for the new owner.

If the record is not found in a buffer then the least recently used buffer will be extracted from the LRU list and assigned to the calling process. This will then be locked down and the current contents written out to the disk if they have been modified since the last physical read or write. The new record will then be read from the disk into the buffer.

7.1.6 Notes

7.1.6.1 Insufficient buffers

As the number of buffers may be smaller than the number of users, it is possible for all the buffers to be in use when a new one is required. In this case the calling process will wait on a semaphore until some other process explicitly frees a buffer, or until another user enters locate and successfully finds a buffer, for itself, in which case a buffer may hve been implicitly freed. When a process waiting for a buffer is released from the semaphore the entire LOCATE procedure is retried.

7.1.6.2 Short record address

Note that the record address field in the buffer descriptor is 24 bits long. This is shorter than the two words allowed in most places and represents an inconsistency although it is unlikely to be a problem with current technology limiting disk sizes (for the Prime anyhow) to around 600Mb (24 bits for the record address and a 1040 word record allow for around 1600Mb of addressable data).

7.1.6.3 Concurrency

Almost the entire LOCATE mechanism is protected against concurrent execution by the inhibition of interrupts. Exceptions to this are the read and write routines and for users waiting on the 'no buffers' semaphore.

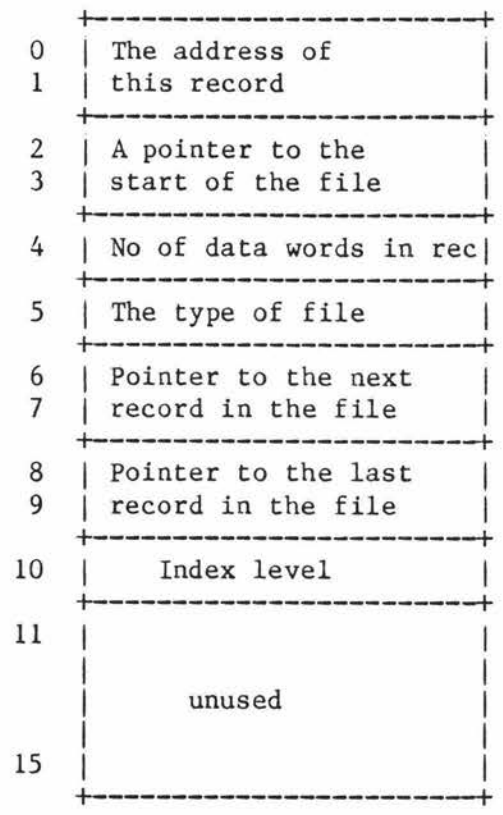
When the buffers are being written to or from the disk, the in transit bit is set in the flag field. This bit is tested before a buffer is assigned to a new user or removed from an old one.

7.1.6.4 The 30 second abort

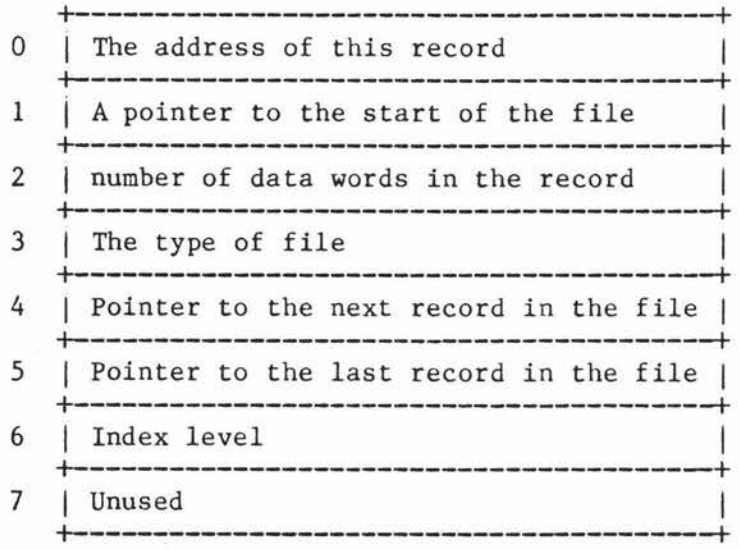
To avoid the possibility of data sitting in the LOCATE buffers for a long period of time (which would create uncertainty about the current state of the files if the system should crash) every 30 seconds any modified, but unwritten, buffers are written to the disk.

7.1.6.5 Record headers

There are two different types of record header used on Prime disks. Storage module type disks, which have a 1040 word physical block, have a 16 word header with the following format:



Other, 448 word, devices have an 8 word header with the following format:



The conversion between the type of record header and the internal format is performed by LOCATE immediately after each read and immediately before each write.

7.2 DISKIO

When the system wants a record off any kind of disk (1040 word storage module through to a floppy disk) DISKIO is called.

DISKIO has two functional parts:

- o The called side of DISKIO and

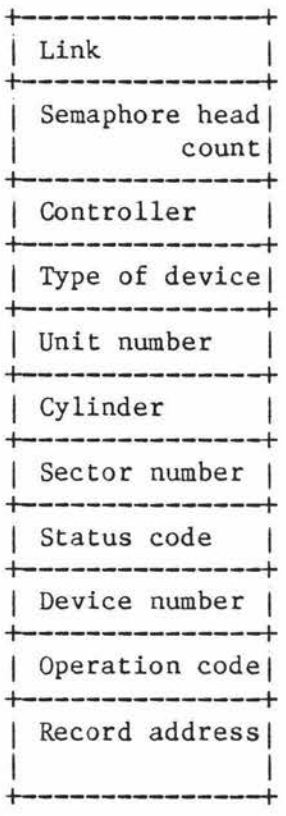
o the disk processes.

The called side of DISKIO has two entry points, one for reading from the disk (RREC), and one for writing to the disk (WREC). The called side of DISKIO is responsible for filling in the details of the transaction into a control block, scheduling that block into a queue of those waiting to access records on the disk, and then suspending the caller until the IO is complete.

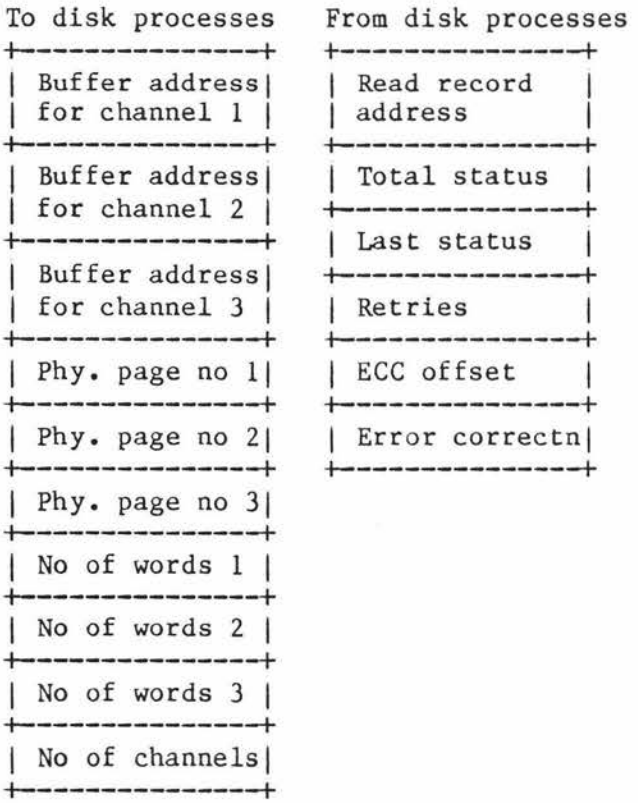
The disk processes are responsible for removing control blocks from the queue, creating the channel program for the controller, initiating the transfer, and finally notifying the waiting process that the IO is complete.

7.2.1 Disk Scan Queue

This queue consists of 17 control blocks, each of which is in one of the following formats (depending on the direction of transmition):



(this part is common to both directions)



There are three lists threaded through the control blocks, each control block being present on exactly one list. These are

1) The list for controller 1.
2) The list for controller 2.
3) The list of unused blocks.

There is also a semaphore on which any processes which cannot be immediately assigned a queue control block are held.

7.2.2 RREC and WREC

The called side of DISKIO begins by obtaining a free queue control block and continues by filling in the block with all the information about the transaction.

The fields are used as follows:

Link — The link for the three lists mentioned above.

Semaphore — The semaphore on which the process will await completion of the transaction.

Controller — The address of the controller.

Type of device	0 : Moving head disk with 8 sectors; 1 : Fixed head disk with 8 sectors; 2 : Floppy disk with 4 sectors; 3 : Prime moving head disk with 8 sectors; 4 : Prime fixed head disk with 64 sectors; 5 : Prime moving head disk with 32 sectors; 6 : 1040 word storage module.
Unit number	Unit number of the device (within the controller).
Cylinder	Cylinder number.
Sector	Sector number.
Status	1: Operation not completed; 2: Correctable error; 3: Write protected disk; 4: Data deleted (for floppies).
Device number	Physical device number.
Operation	Read, write or seek the disk.
Record	The address of the first record to transfer.

address

The format for the rest of the control block differs in the way it is used to pass information to and from the disk processes. The called side of DISKIO can handle up to three channels for a single transaction (each channel is physically limited). This allows larger quantities of data to be shifted in a single operation. The information about the destination of each channel is included in the control block. When the control block is returned, information about the success of the read is included.

After filling in the control block DISKIO will link the queue into the list for the appropriate controller using the 'C-scan' algorithm. This algorithm arranges records in ascending order. The disk drive is then made to scan towards bigger cylinders until the end of the list is reached. The head then returns to the smallest cylinder (without servicing any requests on the return trip) and repeats the process [TOBY].

Next the caller will wait on the semaphore in the control block until the disk processes have satisfied the request. After being awoken it will check for errors and report these where applicable. Finally execution returns to the caller.

APPENDIX B

Appendix B, the Primon listings, are seperatly bound.

References

BERN75 "Manufacturers' Attitudes towards Benchmarking"
C M Berners-Lee
(A paper from)
Benchmarking; Computer Evaluation and Measurement
Edited by N Benwell
Hemisphere Publishing Corporation 1975.

BROWN82 University Grants Committee
Review Comittee.

BURR1 "System Management Facility II - Capibilities Manual"
Burroughs Corporation 1981.

BURR2 "System Management Facility II - Query Program Manual"
Burroughs Corporation 1981.

CALL75 "Performance Measurement Tools for VM/370"
P H Callaway
IBM Systems Journal No 2 1975
IBM 1975.

CARL76a "A Guide to the Use of Hardware Monitors - Part 1"
G Carlson
EDP Performance Review Vol 4 No 9
Applied Computer Research 1976.

CARL76b "A Guide to the Use of Hardware Monitors - Part 2"
G Carlson
EDP Performance Review Vol 4 No 10
Applied Computer Research 1976.

COX76 "Performance Evaluation Using Existing Accounting Information"
B G Cox
Computer Performance Evaluation Seminar Workbook
University of Waikato.

DOCKER Personel communication with
T W G Docker 1984.

FERR78 "Computer Systems Performance Evaluation"
D Ferrari
Prentice-Hall 1978.

FERR83 "Measurement and Tuning of Computer Systems"
D Ferrari, G Serazzi, A Zeigner
Prentice-Hall 1983.

FLETCH "Vogel Computer Centre Monitoring and Tuning"
A L Fletcher

Computer Performance Evaluation Seminar Workbook
University of Waikato.

HP1 "HP 3000 Performance Guide For Installed Systems"
Hewlett Packard 1981.

HP2 "On Line Performance Tool/3000"
Performance Measurement Package
Hewlett Packard 1981.

HUGH74 "Towards Precise Benchkmarks"
P H Hughes
Computer Systems Management
Infotech International Limited 1974.

HUGH77 "Benchmarks, Workloads and System Dynamics"
P H Hughes
Performance Modelling and Prediction
Infotech International Limited 1977 (8553-9410-2).

KOLE71 "A Software View of Measurement Tools"
K W Kolence
Datamation
Technical Publishing Company January 1971.

KOPE79 "Software Reliability"
H Kopetz
MacMillan.

LADY84 "The design and Implementation of QNEMU:
an Interactive Queueing Network Analysis Package"
N R Ladyman
Thesis
Massey University.

LIST75 "Fundamentals of Operating Systems"
A M Lister
MacMillan 1975.

MACD70 "Computer Systems Simulation: An Introduction"
M H MacDonald
Computing Surveys Vol 2, No 3 1970
ACM.

MCGR84a "The Primon User Guide"
A J McGregor
Technical Report
Massey University Computer Science Department 1984.

MCGR84b "The Primon System Documentation"
A J McGregor
Technical Report
Massey University Computer Science Department 1984.

McDa82 "The Mesa Spy: An Interactive Tool for Performance Debugging"
G McDaniel
Special Interest Group on Performance
ACM.

NCAR81 "The System Plot Package"
NCAR Technical Note NCAR-TN/162+IA
National Centre For Atmospheric Research 1981.

PRIM1 "Primos Concepts and Implementation Course Notes REV 18"
Prime 1981.

PRIM2 "The System Administrators Guide REV17.2"
Prime 1980.

PRIM3 "The System Administrators Guide REV19.0"
Prime 1982.

PRIM4 "The Systems Architecture Reference Guide Rev A"
Prime 1980.

PRIM5 "The Subroutine Reference Guide REV19.0"
Prime 1982.

ROSE78 "A Measurement procedure for Queueing Network Models of Computer Systems"
C A Rose
Computing Surveys Vol 10 NO 3 1978
ACM.

SCHW78 "Hybrid Simulation Models of Computer Systems"
H D Schwetman
Communications of the ACM Vol 21 No 9 1978
ACM.

SPERR1 "Sperry Univac 1100 Series"
"Operating System Installation Reference"
Chapter 5 System Tuneing and Testing
Sperry Univac.

SVOB76 "Computer Performance Measurement and Evaluation Methods: Analysis and Applications"
L Svobodova
American Elsevier Publishing Company 1976.

THEO72 "Properties of Disk Scheduling Policies in Multiprogrammed" Computer Systems
T J Teorey
AFIPS Fall joint computer conference procedings 1972.

WILE77 "Just Enough Queueing Theory"
J M Wiley
Datamation January 1977
Technical Publishing Co 1977.