A Unix Performance Monitor

Richard Maurice

Supervisors:

J. P. Penny
P. J. Ashton
Acknowledgements

Thanks to my supervisors, John Penny and Paul Ashton, and to James Collier who has always been available to discuss abstruse aspects of Unix at great length.

Unix™ is a registered trademark of AT & T.
Contents

1 Introduction 3
2 Performance Monitors 4
   2.1 Software performance monitors 4
   2.2 Motivation for a different type of performance monitor 5
   2.3 The Penny and Ashton model 5
3 A Unix monitor 7
   3.1 Adaptation of the model to multiprocess/user models 7
   3.2 Multiprocesses per user in Unix 8
   3.3 Unix and the user virtual process 8
   3.4 MON - an implementation of the model for Unix 9
4 Overall design of MON 10
   4.1 Timing aspects 10
   4.2 Determining the state of a process 11
      4.2.1 Kernel data structures 11
      4.2.2 The process and user structures 12
   4.3 Associating individual real processes with a user 12
   4.4 Data collection and workload measurement 13
5 Determining the state of a real process 16
   5.1 Relevant fields of the process structure 16
   5.2 The initial analysis 17
   5.3 The CPU run process 17
   5.4 Analysis of blocked processes 17
      5.4.1 States of a blocked process recognised by the monitor 19
6 Results 21
7 Monitor validation and interference 24
   7.1 Validation of MON 24
   7.2 Interference 25
8 Enhancement of MON 26
   8.1 Addition of remote logins 26
   8.2 Enhancement of real process state determination 27
9 Conclusion 28
   9.1 How far does MON go at present? 28
   9.2 Conclusion 29
References

Appendix 1

A1.1 Kernel process synchronisation primitives
A1.2 The block buffer cache
A1.3 Signals
A1.4 Sockets & pipes
A1.5 The Unix scheduling algorithm and the effect of changing process priority
A1.6 Simulation of the short-term scheduler
A1.7 Separating the real processes supporting a user into background and foreground processes
A1.8 State analysis of socket waits

Appendix 2 Some Unix performance tools

A2.1 time
A2.2 prof
A2.3 ps and sps
A2.4 top
A2.5 systat, vmstat, iostat, netstat

Appendix 3 User guide for MON

Program listings
1 Introduction

The aim of this project was to produce a Unix implementation of the Penny and Ashton[86, 87] performance monitor model, a model which attempts to provide insight into the causes of user response time delays.

An implementation of the model had earlier been produced for the Primos operating system[Ashton84], an operating system with a single process per interactive user, but not for an operating system like Unix where interactive users are supported by multiple processes.

Performance monitoring, the Penny and Ashton model, and the adaptation of the model for a Unix system are the subjects of sections 2 and 3.

Sections 4 and 5 examine the design and implementation of the actual Unix monitor, called MON for convenience, and some results produced by MON are given in section 6.

Section 7 considers the veracity of results produced by MON: the validity of techniques used in MON to establish the state of the system at sample time, and the degree to which MON interferes with that state.

Additions to be made to MON are discussed in section 8 and the report concludes in section 9 with an examination of MON as it is at present.
2 Performance Monitors

An empirical performance evaluation of a computer system requires measurement of the values of various indices which describe, qualitatively or quantitatively, some attributes of the system. These indices are known as performance indices - an example is response time. The selection of performance indices and the relative importance placed on them will vary according to the particular interests and requirements of each class of persons associated with a computer system.

The interests of users of an interactive system separate into two major categories, one including things like the ease of use and the various tools that may be provided by the system and the other category including more traditional measures of performance like throughput and response time. Ferrari[78] describes the former category as the functional aspects of a system and reserves the word performance for the latter. Here performance is used in the same sense as Ferrari.

2.1 Software performance monitors

A software performance monitor is a program, or perhaps fragments of code, executed by the system to determine the values of one or more performance indices. Software performance monitors are of three basic types:

1) Event-driven monitor: code in the operating system is executed on the occurrence of some event or event combination. These probes or checkpoints either collect and store data by themselves or call a measurement routine for the same purpose.

2) Sampling monitor: a program usually external to the operating system which, instead of monitoring changes in the system state as in an event-driven monitor, samples the state of the system at intervals. The sampling technique is based on well-known statistical techniques. Its success and precision depends on sample size.[79]

3) Hybrid monitor: a cross between an event-driven and sampling monitor. The event-driven part updates a database which is examined by the sampling part at intervals.
When choosing a monitor, the following are important:

1) The type of information obtainable
2) Accuracy
3) Interference - all software monitors consume system resources, leading to bias in the data collected

2.2 Motivation for a different type of performance monitor

For an interactive user of a time-sharing system the most important aspect of system performance is response time. A significant limitation of most existing performance monitors is that they measure performance indices for aspects of the computer system from the system rather than from the user point of view, e.g. CPU utilisation, disk channel utilisation, paging rate, disk seek time. Although a performance evaluation may be carried out in order to provide information which will enable response time to be improved, establishing causes of poor response time through examination of the performance indices just mentioned can be difficult.

For example, a performance evaluation was made by the author for the University Prime 750 early in this year. Traditional performance indices provided by a tool known as USAGE indicated that, although CPU utilisation was high, so was the system paging rate. This suggested that to improve response time a CPU upgrade and/or main memory expansion was required, the cheaper of these two options being main memory expansion. However, the ASHMON monitor[Ashton84], which breaks user response time into components, showed that the high paging rate was attributable to a small class of user and an insignificant component of general user response time relative to CPU queuing time: main memory expansion would not help most users. A trial period of the Prime with an additional 2 MB of main memory supported this conclusion.

2.3 The Penny and Ashton model

The ASHMON monitor referred to above is an implementation of a model developed by Penny and Ashton[Penny86]. The Penny and Ashton model examines system performance from the user's point of view, providing values of a performance index that should be of interest to any user of an interactive computer system, namely the components of interactive user response time. Measuring the size of components of a user's activities gives insight into the causes of response time delays, especially if these components are broken into queuing and running parts.
The basic method is to describe the system as a set of resources which the user process may either be queuing for, or running on, or neither. Then the state of a user process is described by

1) which resources the user process is queuing for; and
2) which resources the user process is running on.

The suggested implementation of this model is as a hybrid monitor[Penry86]. Each user process would have a stateword, a bit-array with two bits for each resource.

Checkpoints in the operating system code would set and clear bits in the stateword of a process as the state of the process changes. For example, the operating system code which moves a process from blocked on a disk transfer to the CPU ready queue when the disk transfer has been completed would also clear the disk run bit and set the CPU queue bit.

The sampling part of the monitor would periodically examine the process stateword - the state of the process corresponding to the integer value of the stateword - and then update state frequency information.
3 A Unix Monitor

3.1 Adaptation of the model to multiprocess/user systems

With an operating system like Primos there is only one real process for each interactive terminal so the user may only be either queuing for or running on one particular hardware resource at any one time. Thus the number of states that the user process may take on is restricted to $2^r$, where $r$ is the number of resources considered, and it is feasible to implement a monitor which provides a frequency distribution for each of the possible states (e.g. ASHMON[Ashton84]).

However with Unix, a user may at any time be supported by many real processes. The model is still applicable: a user may be regarded as being supported by a single virtual process which is the aggregate of the states of the real processes implementing the virtual process at any one time. A suggested method of deriving the virtual process stateword is to use the bitwise OR of the real process statewords doing work for a user[Penny87].

But the number of possible states is now $2^{2r}$, since the user stateword may no longer just have 1 bit set. This makes complete recording of the components of virtual process time infeasible. However, a modified component breakdown of virtual process time may still be possible if suitable screening of the virtual process states is made as part of the measurement phase; many of the virtual process states might be grouped into more meaningful state sets which are determined by dominating real process states.

The alternative to component separation is selective measurement[Penny87], whereby the sampling part of the monitor, instead of recording state distributions of the user virtual process:

"1) measures resources utilisations - by counting the proportion of samples for which $rbit$ of any stateword = true for that resource.

2) measures the extent of overlapping operation of any subset of resources - by counting the proportion of samples for which the $rbits$ of any stateword = true for each resource of that subset.

3) To find whether any resource is a bottleneck - by counting the proportion of all readings for which the $rbit$ or $qbit$ for that resource = true[Penny87]"

For all but part (2) of selective measurement described above, production of a virtual process stateword is no longer required. It is sufficient to recognise those real processes doing work for any user and examine their statewords. However, overlapping resource usage is one of the more significant aspects of multiprocess/user systems, so the virtual process concept is retained even in selective measurement.
3.2 Multiprocesses/user in Unix

A command line interpreter in Unix is known as a shell, one example being the Cshell, which is the command line interpreter considered here. The Cshell is a user process which exists during the period a user is logged in. Multiple processes per user come about because for each command given to the Cshell, a new process is created to execute that command (except, of course, for built-ins which are executed by the Cshell process). When one or more commands are typed together as a pipeline (a sequence of commands connected through pipes), a single job is made by the Cshell consisting of these commands together as a unit. For each one of the commands a separate process is created, and the processes communicate through the pipes. All of the processes in the job are in the same process group, which is recorded in the control block of the process.

For example, consider this command pipeline:

```
ls -l | sort -n | head -5
```

On receipt of this line, the Cshell creates a process for each of the 3 commands, ls -l, sort -n, head -5, connecting them through pipes.

Furthermore, each of the commands may itself create some processes; processes are often launched in Unix to execute some code which can be done in parallel with other code, or simply because it is an easy way to preserve the state of a process at some point in its execution.

A user may have multiple jobs. Jobs may be run in the background - that is started by the Cshell and then left to execute. The Cshell does not wait for their completion but is immediately ready for another command line to be submitted by the user. Any command line terminated by & is run by the Cshell as a background job. Foreground jobs are those jobs submitted by the user for which the Cshell does wait. A user may have many background jobs but only one foreground job.

3.3 Unix and the user virtual process

For the purposes of the model, it is the real processes in the foreground job which together form the interactive virtual process associated with a user. However, the ease of creation of background jobs means that almost any substantial commands submitted by a user are given as background jobs, and the user can still be very much interested in the response time of such background jobs. Consequently, the stateword corresponding to a user virtual process is separated into foreground and background statewords, so that "analyses can be performed for response of foreground processes, background processes, and (by combining the two statewords) all processes".[Penny87].
3.4 MON - implementation of the model for Unix

The Penny & Ashton model proposes a hybrid monitor as described in section 2.3. However, when the event-driven part of the monitor has not been incorporated into the operating system by the system designers it is simpler to implement the monitor in a pure sampling form, as was done with ASHMON[Ashton84], than alter operating system code. MON is a sampling monitor which is an implementation for Unix of the multiprocess per user Penny & Ashton model.
4 Overall design of MON

This section discusses the design of MON, which can been separated into three parts:

1) sampling monitor aspects
   i. timing
   ii. determining the state of a real process

2) Penny & Ashton model aspects
   i. finding the real processes which support a user and form the user virtual process

3) Data collection

Only general aspects to do with determining the process state (1. ii.) are considered in this section; section 5 contains the detail.

4.1 Timing aspects

As a sampling monitor, MON must examine the state of the system at different times. Several important factors relating to the intersample time must be considered:

1) The nature of the sampling interval
   Intersample times may be constant or vary according to a frequency distribution. Presently MON uses a constant interval between samples, the value of the intersample time being specified by the user or defaulting to 1.23 seconds. Constant sampling intervals have the advantage that they are easier to implement and, according to Ferrari[58], "...can be used if care is taken not to choose a sampling frequency equal to an integral multiple or submultiple of a frequency component present in the quantities to be sampled". For MON this basically means not having an intersample time which is a multiple of 1/10 second - the time between wakeups of processes sleeping on lbolt (see s. 5.4.1(11)).

2) The nature of the interval timer
   Most operating systems (including Unix) provide a sleep facility which allows a process to suspend itself for a specified time interval. However, this is not the ideal means of providing a sampling interval because the running time of MON varies; if there is a greater number of active processes then MON will have more to do. At high workloads there would be fewer samples because with this technique:

\[ \text{interval} = \text{runtime} + \text{sleptime} \]

and \( \text{runtime}_{\text{highworkload}} > \text{runtime}_{\text{lowworkload}} \)
Unix also provides each process with three interval timers, each of which may be given a value that will be used to automatically reload the timer when it expires. One of these timers, \texttt{ITIMER\_REAL}, is decremented in real time - this is the timer used by MON to implement the sampling interval. Through using this timer, intersample time is independent of monitor run time.

3) Monitor priority
MON runs as a user process (as opposed to part of a kernel interrupt handler), therefore it must compete with each other process for the CPU. When the CPU is under heavy demand MON could be awoken but then have to queue for some time before being able to run. The intersample time could be extended as a result of such queueing. To avoid this situation and to reduce the likelihood of being preempted by another process, MON favourably affects its scheduling priority at initialisation time (see Appendix 1.5 for a more detailed discussion of the Unix scheduling algorithm and the effect of changing process priority)

4.2 Determining the state of a real process

4.2.1 Kernel data structures
Unix is an embedded operating system. The operating system itself is called the \textit{system kernel}, or just the kernel, and is found in the top half of virtual address space (see Fig 4.1). The kernel is really only accessible to a user through the set of system calls, but a user with sufficient privileges may \texttt{read} the kernel address space as a device (\texttt{idev\_kmem}).

![Figure 4.1 Process Address Space](image)

At each sample MON attempts to discover the state of individual real processes. This state (as it has been defined in section 2.3) is described by the value of relevant control blocks in
the kernel. MON contains a list of those kernel data structures which are considered relevant and reads them using a technique copied from *ps* (see Appendix 2.3), namely:

i) At initialisation time MON reads the kernel symbol table to find the kernel addresses of:
- areas of contiguous structures
- pointers to areas of contiguous structures
- variables indicating the number of contiguous structures in an area

It then sets aside appropriate space for these structures.

ii) During a sample the monitor copies the relevant structures from kernel address space into the space allocated by the monitor in user address space.

### 4.2.2 The Process and User structures

Because MON is primarily interested in the behaviour of processes, the examination of process control blocks underlies the whole implementation (see section 5). In Unix, the Process Control Block is divided into two separate structures; the *user structure* and *process structure*. The process structures of all processes are in a table in kernel virtual address space and so are accessible to the monitor via the technique described in section 4.2.1. However, the user structure is in user virtual address space, except that the user structure for the currently running process is mapped into the bottom of kernel address space (see Fig 4.1). This means that to access the user structures associated with a process, MON would need to simulate virtual address translation and read main memory directly. This is not impossible, as *ps* demonstrates. However the user structure may be paged or swapped to disk. For a monitor such as MON it is unacceptable to make disk accesses as part of the monitoring process, principally because the monitor would block until the disk transfer was completed. While the monitor was blocked another process would run and possibly make significant changes to the state of the system. For these reasons the examination of process control blocks is limited to process structures.

### 4.3 Associating individual real processes with a user

Because the ultimate aim of the monitor is to examine the state of the *virtual process* supporting a user (see section 3.1), real processes doing work for each user must be grouped together. The original intention was to do this with the process User ID - a number which identifies the owner of a process. However, this was found to be unsatisfactory for two reasons:
1) A process actually has two UIDs: the real UID which specifies the actual owner of the process and the effective UID which is used by the kernel when checking access permissions. The effective UID does not necessarily correspond to the process owner and is the only UID which is stored in the process structure - the real UID is stored in the user structure which for the reasons discussed in section 4.2.2 is not accessible to the monitor.

2) There is no reason why several people could not be logged in under the same username.

The solution implemented is to consider the terminal virtual process rather than the user virtual process. Each terminal has an associated control block known as the tty structure which includes, inter alia, the foreground process group of the terminal. This is sufficient information to find all foreground and background processes associated with the terminal. At present MON considers only foreground processes, but extension to background processes is relatively simple. (See Appendix 1.7 for a detailed explanation of the method).

4.4 Data collection and workload measurement

Many unresolved issues surround the interpretation of virtual process states. Consequently the present the measurement part of MON is not that suggested in section 3.1. Instead, a measurement part was built which breaks real process time into components, similar to the measurement part in ASHMON [Ashton84].

The components of real process time may be used to:

i) measure resource utilisations

ii) show where real process delays occur

For any monitor collecting data on the state of a computer system it is very important that the context of the collection be recognised. Using the terminology of empirical design [Ferrari78], there are factors which appreciably affect the values of a given performance index. The different levels of these factors must be observed and incorporated into the data collection mechanism. For a computer system the factors may be broadly classed as system configuration and workload. It is assumed that the system configuration is unchanged for any single run of MON and therefore may be ignored by the monitor itself (although not by the experimenter).

However, system workload is likely to vary through a session and to have consequences for the data collected during a particular sample. For example, under high workloads the CPU will be in more demand, but user think time should remain constant. CPU queueing time will then become a greater component of response time relative to think time.
To account for varying system workload MON collects data in an array like that used in ASHMON[Ashton84].

Figure 4.2 -adapted from [Ashton84]

As can be seen from figure 4.2, the workload level at the time of a sample is estimated by the number of logged in terminals. The number of logged in terminals is an ad-hoc measure that should be revised in future development of MON. The major failings that an estimation of workload from the number of logged in terminals exhibits are:

- in a non-charging environment like that which exists for cantuar, the departmental VAX 11/750, some users are likely to be logged in when they are not in fact using the computer at all.
- there is no recognition of the increased workload which occurs from time to time as a result of administrative background processes (not attached to any terminal) being fired, e.g. mailers.
- it assumes consistency over time of user work.

The first problem reasonably simple to solve: the sleep time of a process is recorded in the process structure of that process, therefore any terminal with a Cshell which has been waiting on user input for more than say, 5 minutes, could be considered logged out. However, the other two unsatisfactory aspects require a significant change to the workload measure to be resolved.
Interpretation of the count array

At present, one small tool called components has been written to interpret the count array (Fig. 4.2). Components produces the components of total process time for each level of workload (number of logged in terminals). Each component may be one or a group of real process states: the user specifies the contents of each group.

For example, the user might specify a group of components as {CPU queue, CPU run}. Then, for each vector in the response array, components will output the pair:

(num. logged in terminals, (CPU queue counts + CPU run counts) / total counts )

The output is in a form suitable for reading by the Macintosh Cricketgraph graph-drawing utility. Section 6 shows some example graphs.
5 Determining the state of a real process

Determining the state of the real processes supporting a user is an important part of MON and the aspect which has taken the largest proportion of the author's time. This section discusses the individual real process states and the methods used to determine those states.

5.1 Relevant fields of the process structure

As discussed in the previous section, the data structure associated with a real process which is of primary interest to MON is the process structure. The five fields of this structure examined by MON to assist in determining the state of the process are:

1) p_stat a single byte indicating which of several fundamental and mutually exclusive states a process is in:
   i) sleeping (awaiting an event - see Appendix 1.1 for a discussion of the kernel process synchronisation primitives)
   ii) ready to run on the CPU
   iii) stopped
   iv) intermediate state in process creation
   v) intermediate state in process termination

2) p_flag a 32-bit mask further describing the state of a process.

3) p_wchan a longword (32 bits) specifying the event a sleeping process is waiting for (see Appendix 1.1)

4) p_pri the scheduling priority of the process

Figure 5.1 Basic state sets of a real process
5.2 The initial analysis

The first step (and last for many processes) taken by MON is to examine the p_stat field of the process structure. This separates the processes into the mutually exclusive sets described in section 5.1 (1) and depicted in Fig. 5.1. A process which is ready to run is considered to be queueing for the CPU except when that process has already been designated as the CPU run process. A process blocked on some event proceeds to further analysis. Processes which are in the remaining states (creation, termination and stopped) have the corresponding bit set in their stateword.

5.3 The CPU run process

Obviously at the time of sampling the process running on the CPU will always be the monitor itself. Thus instead of using the actual CPU run process, MON considers the process at the head of the CPU queue to be the process which is running on the CPU. To find the process at the head of the CPU queue, MON simulates the short term scheduler code which simply chooses the first process on the highest priority run queue with a valid entry (see Appendix 1.6 for a more detailed description of the simulation of the short-term scheduler).

5.4 Analysis of blocked processes

The w_chan field of the process structure contains a kernel address which denotes the event that will cause the process to be rescheduled (see Appendix 1.1 for a discussion of the kernel process synchronisation primitives and the use of the w_chan field). Part of the technique used to determine the state of the process is a comparison of the value of w_chan to a set of known addresses. In general, an address that is stored in the w_chan field of a process may fall into one of three categories:

1) address in a unique (to kernel memory) data structure - normally just the address of a longword variable
2) address in a data structure which is one of a contiguous array of the same data structures
3) address in a data structure which is within an area of the kernel virtual address space which has been set aside for such data structures.

- Figure 5.2 shows the layout of kernel address space
MON maintains a database of pairs \((\text{low\_address}, \text{high\_address})\) for the events considered interesting. For each of the three categories the pair is:

<table>
<thead>
<tr>
<th>category</th>
<th>low address</th>
<th>high address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>address of variable</td>
<td>address of variable + 1</td>
</tr>
<tr>
<td>2</td>
<td>address of base of array</td>
<td>address of last longword in array + 1</td>
</tr>
<tr>
<td>3</td>
<td>address of base of virtual area</td>
<td>address of end of virtual area + 1</td>
</tr>
</tbody>
</table>

The comparison finds the pair such that \(w\_\text{chan}\) is an element of \([\text{low\_address}, \text{high\_address}]\).

Several additional techniques are used to make state distinctions where necessary. They include:

1) determination of the offset of the \(w\_\text{chan}\) within a particular structure
2) examination of the wakeup priority of the process contained in the field \(p\_\text{pri}\) (the scheduling priority the process will have when notified that an event has occurred)
3) investigation of the contents (usually kernel flags) of other data structures which are linked (directly or indirectly) to the structure the process is waiting on
5.4.1 States of a blocked process recognised by the monitor

1) Page-wait state
Any blocked process with the process structure `page-wait` flag on is considered to be in a page-wait state. Because this flag is set and cleared by the page-fault handler then we can be absolutely sure of correctly interpreting page-wait states. If we did not examine this flag we would need to check buffer flags to distinguish paging from swapping: a process waits on a swap buffer for both.

2) Waiting for swap I/O
Although there is a `swapper process` that swaps processes out/in whole as the need arises, user processes seem to do some swapping themselves, often when exiting. A separate list of buffer headers is used for swap I/O, as well as a special buffer header `rswbuf`. Whenever a process is waiting on a swap buffer header or `rswbuf` it is considered to be in a swap-wait state.

There are two situations when a process will wait on a disk buffer header:
i) when the process is waiting for disk I/O to complete
ii) when the process is waiting for a block in the buffer cache which is in use by another process. (The file system allows only one copy of a disk block to be in the buffer cache at any time, and locks that block when it is in use by a process - see Appendix 1.2 for a discussion of the buffer cache I/O system).
A process in state (i) has wakeup priority `PRIBIO` while a process in state (ii) has wakeup priority `PRIBIO + 1`. MON uses this fact to distinguish between (i) and (ii).

4) Waiting on a locked inode
Each file has an `inode` which is the source of most information about that file on disk. The inode contains the file access, ownership, modification date, etc. and in particular pointers to the disk blocks containing the file data. A copy of the inode is kept in a kernel table while the file is open.
Although in-core inodes are shared to allow simultaneous access to files, they are locked when their contents are being manipulated (to prevent inconsistencies). A process waiting for a locked inode waits at the base of a particular in-core inode structure.

5) Waiting on a file lock
Unix incorporates `advisory` file locks whose presence is indicated by two fields of the inode structure `i_shlockc` and `i_exlockc`. A process blocked on an exclusive lock waits on the `i_exlockc` field of a particular inode structure, whereas a process blocks on a shared lock waits on the `i_shlockc` field.

6) Waiting for a child process
A process waiting for a child process to exit waits on the process structure of the child.
7) Waiting on an InterProcess Communication buffer (IPC buffer: a memory buffer associated with a socket)
-see Appendix 1.4 for a brief discussion of sockets (& pipes).
There are two primary situations when a process will wait on a socket structure (a control block associated with a socket)
i) If an IPC buffer that a process is writing to becomes full then a process will block until that buffer has room for writing.
ii) A process reading from an IPC buffer will block when that buffer becomes empty.
MON distinguishes between these and goes on to determine the communications domain of the socket - see Appendix 1.8 for a full description of the method used.

8) Waiting on terminal output/input
As stated in section 4.3, for each terminal there is a control structure called the tty structure. Included in this structure are two small character buffers (clists - "see-lists") for input from and output to the terminal. The input buffer is known as the raw queue and the output buffer is known as the output queue. A process waits in a tty structure for two main reasons:
i) Waiting for user input: The process waits on the raw queue
ii) Waiting for output to be passed to the terminal: the process waits on the output queue. Ring terminals waiting for output to be passed wait on a structure called port.

9) Selecting
A process may wait to read from or write to multiple buffers (which may be sockets, terminals, files ...) at one time. When in this state the process waits on a longword called selwait. At present MON considers processes waiting on selwait to be in "selecting wait state".

10) Waiting for a signal
Processes may send signals to other processes to indicate the occurrence of an asynchronous event. (See Appendix 1.3). Furthermore, there is a system call (sigpause) which allows a process to wait for the occurrence of one of a set of signals, analogous to the kernel sleep primitive. When a process is awaiting a signal it waits on the user structure.

11) Waiting for a short time, rather than an event
Sometimes it is inconvenient for the kernel to have a process to wait on a specific event, because that event is ill-defined. In these situations the process waits on a longword called lbolt. Processes waiting on lbolt are awoken once every second by the clock interrupt handler. These processes are then free to examine the state of the system, determine whether they should continue executing or perhaps suspend themselves for another second.
6 Results

As indicated in section 4.4, MON presently produces a "count array" which is used by the tool components to produce a breakdown of the components of real foreground process time. Four 6000 sample runs of MON were made on cantuar, the departmental VAX 11/750, over a two day period of medium load (for the department). The output arrays were combined by using the small tool merge and some example component breakdowns produced which are included in this section.

![Graph showing processor time components]

**Fig. 6.1 Processor time components**

Figure 6.1 depicts the CPU components of real foreground process time. As expected there is little queueing for the CPU at low workloads.

![Graph showing I/O components]

**Fig. 6.2 I/O components**
Figure 6.2 shows some I/O components of real foreground process time: paging and non-swap/non-paging I/O. As the number of logged in terminals and hence the number of processes increases, contention for main memory means that there are less real page frames per process. Consequently, there is an attendant increase in the number of page-faults: paging I/O becomes a larger component of process time. In addition to competition for memory there is competition for the disk channel. Increased queueing times for disk transfers enlarges both the paging and normal I/O components of real foreground process time.

![Graph showing % of total process time against number of logged in terminals]

**Fig. 6.3 System and non-system components of process time**

For figure 6.3 the possible states of a real process were divided into two groups, one group containing the states where processes were actually undertaking work for the user - the system states, and the other group consisting of the two states waiting for terminal input and stopped. The distinction is fairly arbitrary: processes waiting to read from a socket are included in the system group, but in most cases they are probably little different from processes waiting on terminal input. Consider also processes waiting for a child process to exit or waiting for a signal. None of these is queueing for or running on a system resource, except perhaps indirectly by waiting on another process which is. However, for the purposes of this example the distinction will do, and, as expected, system time becomes a larger component of process time when the system load grows.
Figure 6.4 Various components of process time

Here the component groups are:

CPU: \{CPU queue, CPU run\}

I/O: \{Paging I/O, Swap I/O, Non-paging/non-swap I/O\}

Input: \{Waiting for terminal input\}

Other: \{all other recognised process states - see section 5\}
7 Monitor validation and interference

7.1 Validation of MON

At present the main validation of MON has been confined to testing that the recognised real process states are correct interpretations of the state of a process. Two techniques have been employed:

i) Comparing a process state determined by the monitor to the process state found by ps and sps (see Appendix 2.3 for a description of ps and sps). Obviously ps and sps are only of limited use because MON generally refines the state of a process considerably further than these two (especially ps).

ii) Forcing a process into a known state and comparing this state to that found by the monitor. For example, the process corresponding to the command "cat" may be forced to wait on a read from a pipe by submitting the command line "sleep 5000 1 cat" to the Cshell. As expected, the monitor found the "cat" process waiting to read from a UNIX domain socket.

A more general validation of the monitor should be made in the same way that as that used to validate ASHMON[Ashton84].

For a run of the monitor the states of all processes were examined, and then performance indices calculated which were also available from USAGE, another performance tool available on Prime machines. USAGE was run at the same time as ASHMON and the values of the indices produced by ASHMON and USAGE compared.
7.2 Interference

As indicated in section 2.1, an attribute of all monitors is the bias that is introduced into the data collected because the monitor itself consumes system resources. MON was examined using some existing Unix performance tools.

CPU consumption

For 1000 samples of MON with an intersample time of 1.23 seconds the following CPU times were recorded by the `time` command (see Appendix 2.1):

<table>
<thead>
<tr>
<th>obs.</th>
<th>system</th>
<th>user</th>
<th>total</th>
<th>real time (s)</th>
<th>CPU utilisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>48.9</td>
<td>79.2</td>
<td>128.1</td>
<td>1261</td>
<td>10.16 %</td>
</tr>
<tr>
<td>2</td>
<td>46.0</td>
<td>73.7</td>
<td>119.7</td>
<td>1245</td>
<td>9.61 %</td>
</tr>
<tr>
<td>3</td>
<td>44.5</td>
<td>75.6</td>
<td>120.1</td>
<td>1234</td>
<td>9.73 %</td>
</tr>
</tbody>
</table>

Average CPU utilisation: 9.83 %

MON was observed for about 15 minutes with the performance tool `top`. `top` provides periodic information of recent process resource utilisation (see Appendix 2.4). The CPU utilisation of MON during the observation period varied between 8.5 % and 9.0 %, confirming the calculations made above.

From this data it can be concluded that the CPU utilisation of MON is about 10 %. Obviously this figure will depend on the the system workload at the time of the monitor run: if there are more processes on the system then MON will have to make more analyses of process states. However, as an informal estimate of the CPU utilisation it will suffice.

CPU utilisation of about 10 % is probably too high. A figure of less than 5 % would be more acceptable. The simple solution is to increase the intersample time: doubling it to about 2.5 seconds should give a utilisation near 5 %. This will, however, also double the time required to take the same number of samples.

Memory consumption

Repeated observations of different runs of MON with `ps` (see Appendix 2.3) indicated that real memory consumption of the monitor was slightly less 3.5 % on `cantuar`, a vax 11/750 with 4 MB of main memory.

Consumption of this amount of memory is acceptable.
8 Enhancement of MON

8.1 Addition of remote logins

As it stands, MON examines only the processes of users which are logged into a "standard" terminal - either a directly connected terminal (dz terminal) or a UKC ring terminal (ts29 terminal). An increasingly common user of most Berkeley Unix systems is the remote user. Here we will examine how MON might be extended to accommodate users logged in from remote sites.

Consider the mechanism by which remote logins are supported at the host site as depicted in Figure 8.1

The *pseudo-terminal pair* is a device provided by Unix which acts as an interface between one process and another process which expects to read from and/or write to a terminal. In the case of remote logins the program which expects to read and write to a terminal is the Cshell. Rlogind is the remote login daemon - here the process which is providing the communications link via an INET socket. MON is interested in the Cshell process and the child processes spawned by the Cshell; the processes which will perform work for the remote user. For each pseudo-terminal pair there is a tty structure the same as for "standard terminal". If this tty structure is manipulated in the same way as the "standard" tty structure, then MON should be able to use technique described for "standard" terminals to establish the foreground and background processes associated with the remote user. If not, then the fact that Cshells belonging to remote users are children of rlogind should be enough information to find them, even if not providing a way of distinguishing foreground from background
Additional complications arise with pseudo-terminals in that they are frequently used by other various other programs, including multiwindowing terminal interfaces such as UW. However, processes associated with these pseudo-terminals will not be descendents of rlogind. Furthermore, rlogind uses pseudo-terminals in a particular mode which can be used to distinguish pseudo-terminals providing an interface for a remote user.

8.2 Enhancement of real process state determination

As it stands MON provides an extensive analysis of the state of a real process. It also reports on the percentage of samples which were unrecognised (usually < 1\%) and gives the value of the process \texttt{w\_chan} field for those processes waiting on an undetermined event. However, some of the states could be further defined, in particular states relating to disk I/O (page-wait, swap-wait, other disk waits). Here there is presently no running/queueing distinction. The technique used by ASHMON[Ashton84], namely discovering the disk run process via examination of the I/O wait queues and device status bits is probably implementable. The distinction might also be able to be made through examination of the flags associated with the disk buffer headers, providing care is taken with consistency.

There is a great deal of information available to MON in the form of flags used by the kernel. For example, when a process causes a page fault the process structure flag \texttt{SPAGE} is turned on by the page fault handler to indicate that the process is in a page wait state. When the page has been loaded from disk the page fault handler turns the flag off.

There are flags that could be examined by MON in most other structures. However, only some are of use because they are often manipulated by interrupt handlers which can run any time (the kernel raises the processor priority during critical sections of code to block these routines) and so inconsistencies can arise. For example, MON might find more than one process at the head of a driver queue (indicated by a flag in a buffer header) because the queue head has changed between retrieval of the two disk buffer headers from kernel memory.

At present the only flags examined by MON are:

- \texttt{SPAGE}
- various terminal flags used to determine whether or not a terminal has a logged in user
- some flags associated with the socket mechanism.
9 Conclusion

9.1 How far does MON go at present?

Compare this skeletal outline of the proposed monitor for Unix systems (see section 3) to the outline of MON, as it stands at the moment, that follows it.

Proposed monitor:

for each sample do {
1... find the groups of real processes corresponding to a terminal virtual process
2... separate the real process of a virtual process into background and foreground processes
3... find the states of real processes which are in a virtual process group
4... combine the statewords of a group of real processes into the foreground and background statewords of a virtual process
5... make measurements of the virtual process statewords
}

MON:

for each sample do {
1... find the groups of foreground processes corresponding to a terminal virtual process
2... find the states of real processes in a foreground process group
3... make measurements of the real foreground process statewords
}

The two main differences between MON and the proposed monitor at the moment are:

i) MON does not examine background processes attached to a terminal (although the method to do this is described in Appendix 1.7)

ii) Instead of making measurements of virtual process statewords MON uses the real process statewords to produce breakdowns of the components of real foreground process time.
9.2 Conclusion

MON has succeeded in implementing the state recording part of the Penny and Ashton model for Unix as a pure sampling monitor. States of real processes are determined and the real processes are associated with users. However, the proper interpretation of the state of the user virtual process formed by the real processes doing work for the user needs further exploration before a measurement part based on the virtual process statewords is developed. Instead, because there are only 22 mutually exclusive real process states at present, it was feasible to build a measurement part which records the frequency distribution of real process states in the manner used by ASHMON[Ashton84].
References


[UPM (1)] *Unix Programmers Manual, Section 1*.


*Source Code for the 4.3BSD Unix Operating System*, Regents of the University of California, (1986).
Appendix 1

This appendix describes some aspects of Unix relevant to the project and contains the detail of some of the implementation of MON.

A1.1 Kernel process synchronisation primitives - sleep() and wakeup()

Consider these skeletal (and somewhat oversimplified) outlines of sleep() and wakeup():

```
sleep( chan, pri )
{
    process status (p_stat) := SSLEEP;
    process wait channel (p_wchan) := chan;
    process scheduling priority (p_pri) := pri;
    swtch(); /* make a context switch */
}

wakeup( chan )
{
    for each process sleeping {
        if ( wait channel (p_wchan) = chan ) {
            process status (p_stat) := SRUN;
            process wait channel (p_wchan) := 0;
            put process on CPU ready queue;
            if (process priority > running process priority) preempt;
        }
    }
}
```

`sleep( chan, pri )` is the kernel routine which a process calls when it blocks on an event, i.e. the routine which changes the state of a process from running on the CPU to waiting for some event. `chan` is a number (32 bits on the Vax) which identifies the event the process is waiting for - by convention it is the address of a kernel data structure related to the event. For example, a process waiting for a disk transfer calls `sleep` with the address of the buffer header associated with the disk transfer.

`wakeup( chan )` is the kernel routine which notifies blocked processes that an event has occurred, i.e. the routine which changes the state of a process from waiting for some event to ready to run on the CPU. Wakeup schedules all processes waiting for the event indicated by `chan`. 
NB. This is a greatly simplified look at the synchronisation primitives. Important aspects not indicated above include:

1) The hashed sleep queues used by the sleep and wakeup primitives to quicken wake-up calls
2) The fact that there are no context switches while a process is running in kernel mode: preemption actually occurs by setting a scheduling flag which indicates that a context switch should occur when the running process next enters user mode

A1.2 The block buffer cache

The block buffer cache is the main memory buffer system for block devices (basically disk units). The block buffer cache is designed to reduce the total number of disk transfers. It consists of a set of buffer headers which contain device, block number on device, and a pointer to the memory buffer which contains a copy of a disk block. When a disk block is read, it is first searched for in the cache. If it is not found then a buffer header is allocated. (Buffer headers are allocated in LRU order - if a buffer header has been modified then it is written to disk before being reallocated). Once a buffer header has been allocated then the block requested is read from disk.

Similarly, a block which is being written to is first searched for in the buffer cache. If it is not found then a block is allocated as described above. If the whole buffer is written then it is queued for I/O, otherwise I/O is made when the buffer is reallocated.

A1.3 Signals

Signals are a means by which processes are notified of asynchronous events. For example, a user who types ctrl/c on a terminal causes an interrupt signal to be sent to the foreground process associated with the terminal. There are 31 different signals in 4.3 BSD. Processes themselves may use a system call to send signals to other processes.

A process may make a system call which installs a routine of the process' code as a signal handler. Then whenever the process receives the specified signal, the signal handler routine will be executed. A process which has not installed a signal handler for a specific signal or has not indicated that it wishes to ignore the signal will "die" on the receipt of that signal. For example, programs like yap have installed a handler for the interrupt signal so that when a user types ctrl/c they do not die but instead respond with something like "type 'quit' to leave".
A1.4 Sockets & Pipes

The socket mechanism is a sophisticated means of interprocess communication and the aspect of BSD Unix which sets it apart from other implementations of Unix. Here only two simplistic statements are made about pipes and sockets - notable only for that which is not mentioned. A proper discussion of sockets may be found in [Quarterman85].

A socket is an endpoint of communication. A process may connect to a socket and then write or read from a socket as if it was an ordinary file. A process writing to a socket is really sending data to another process which may be local to a machine (Unix domain sockets) or on a remote machine (AF_INET domain sockets). Similarly for a process reading from a socket.

A pipe in Unix is "a reliable unidirectional byte stream between two processes"[Quarterman85]. Basically it is a way for one process to pass data to another process. In 4.3 BSD pipes are implemented as a pair of sockets.

A1.5 The Unix scheduling algorithm and the effect of changing process priority

Several aspects of Unix scheduling are important when attempting to achieve consistent intersample times for a sampling monitor:

1) there is no preemption of a process running in kernel mode (to preserve the integrity of kernel data structures)
2) scheduling is according to process priority
3) process priority is recalculated every second according to a formula which is a function of:
   - recent CPU time accumulated by the process; and
   - the process' nice factor

MON has affected its scheduling priority by introducing the maximum favourable nice factor (-20). However, this does not mean that MON will run immediately it becomes ready because:

i) the kernel manipulates a process' priority directly (see Appendix 1.1), sometimes giving it priorities greater than those that a user may induce with nice.
ii) point 1 above
iii) point 3 above - process CPU usage is still a primary factor in calculating process priority

Consequently MON may not have a constant intersample time - just the best achievable
without altering kernel code.

1 The *nice* factor of a process is a field in the process structure which allows a process crude control over its scheduling priority. Normally it is zero, but its value may be altered with the *nice()* and *setpriority()* system calls.

N. B. Only processes owned by the superuser may *favourably* affect their scheduling priority with *nice()* and *setpriority()*; so MON runs as a process owned by the superuser.

### A1.6 Simulation of the short-term scheduler

The data structures used by the short-term scheduler are 32 run queues (linked lists of process structures) and a 32 bit mask indicating which of the run queues have entries. Each run queue spans four of the 124 levels of priority. Processes are assigned to a run queue according to their priority, but within a single run queue the ordering is according to arrival, i.e. FIFO. Processes which differ in priority by less than 4 will therefore be scheduled FIFO unless there is a 124 mod 4 boundary between their priority levels.

The short-term scheduler (swtch() - an assembler routine) finds the first bit in the mask which is set and schedules the head of the run queue corresponding to that bit.

To determine the CPU run process, MON emulates the short-term scheduler (except, of course, MON does not actually cause a context switch!)

However, there is a difficulty:

Some processes (e.g. those which have just finished a page in) are scheduled at a higher priority than MON so MON may be preempted between the time that the bit mask is examined and the time that the run queue head is examined. If, as a consequence, the highest priority run queue becomes empty, then the queue head pointer loops back onto itself. MON could therefore try to examine the run queue as a process structure, which would give meaningless results. To avoid this, MON checks the queue head, aborting the sample if it is not in the process table.
A1.7 Separating the real processes supporting a user into background and foreground processes

Processes are created by other processes with the *fork* system call. A process which creates another process becomes the *parent* of a *child* process. Each process structure includes a field which contains the process identifier of the parent process; we may therefore discover the ancestry of any process.

![Diagram](image)

**Figure A1.1** The process hierarchy associated with a terminal

Figure A1.1 depicts the process hierarchy associated with a "standard" terminal. For reasons not discussed here (see [Quarterman85]) the top-level shell attached to a "standard" terminal is always a child of a process called *init* which has process identifier 1. (Shells belonging to remote users are different - see section 8.1). MON knows at least one foreground process of a user (see section 4.3), so it can find the shell process: it is the foreground process ancestor which is a child of *init*. Then the background processes associated with the terminal are all of the descendents of the shell process which are not foreground processes.

**State analysis of socket waits**

Each socket has a control block called the *socket structure* which is allocated in a virtual area called the *mbutl* (the *memory buffer table*). The socket structure includes two substructures: the *socket send buffer* and *socket receive buffer*. When a process wants to read from a socket it waits on the socket receive substructure of the socket structure associated with the socket. Similarly, a process wanting to write to a socket waits on the socket send substructure. This is how MON determines whether a process is waiting to read or write.
Figure A1.2 shows the various control blocks associated with a socket structure which must be examined by MON to determine the domain of a socket.

For each socket domain there is a control block called the *domain structure*. The domain structures are linked together in a list. At initialisation time, MON traverses this list and records the addresses of the domain structures for the *AF_INET* (basically remote communications) and the *AF_UNIX* (communications within one machine) domains.

MON follows a pointer in the socket structure associated with the socket the process is waiting on to the *protocol switch structure* (used for implementing the protocols) which in turn contains a pointer to a domain structure. MON can then determine the communications domain of the socket by comparing the value of the domain pointer to the addresses of domain structures recorded at initialisation time.
Appendix 2  Some Unix performance tools

Here we look at some of the performance analysis tools available under Unix: their capabilities, uses, implementation, and relationship to MON (if any).

A2.1  time

Capabilities and implementation
There are two time commands: one is a built-in function of the Cshell and the other is the original time command, executable from the Cshell as /bin/time. Their function is similar; we will look at the simpler command, /bin/time.

Time displays the time that a program takes, breaking that time into real time, CPU user time (time spent in the application program itself) and CPU system time (time spent in the operating system kernel on behalf of the program). Time executes the command line as a set of child processes: the real time is found by comparing the time of day after the children have exited to the time of day when they were dispatched. The system and user time breakdowns are returned in a record to the parent when the last child executes: the system maintains this information for each process in resource usage records attached to the process control blocks.

Use

time is primarily used to examine the CPU use of a single program.

A2.2  prof

Capabilities and implementation
prof provides an execution profile of a program. The execution profile indicates, *inter alia*, the number of calls made to each routine and the percentage of time spent executing each routine. The number of calls of a routine is produced by compiling the program with an option which inserts counting code into the object file, while the percentage of time spent executing a routine is estimated by sampling the program counter at each clock tick.

Use

prof is used to examine the CPU use of routines in a particular program. Knowing CPU expensive routines enables causes for program CPU time use and possible speed-up techniques to be investigated.
A2.3  ps and sps

Capabilities and implementation

*ps* gives information about processes on the system, the particular set of processes being specified by the user (e.g. processes attached to a terminal, all processes etc.). The information includes process ownership, process identifier and various resource usage statistics such as \%cpu utilisation (in the last minute of real time) of the process and process real memory size.

Resource statistics are calculated from resource usage records attached to the process control blocks and system-wide resource utilisation information, while information such as the process id is gleaned from the process control blocks. The most significant aspect of *ps* with respect to the project is that it examines the wait-channel field of the process structure and converts the numeric address held there to a symbol (by comparison with the symbol table packed with the operating system kernel object file).

When MON is trying to determine the states of real processes, it uses some of the symbol-table reading techniques of *ps* to access Unix data structures (see section 4).

*sps* is a new and more sophisticated version of *ps* which makes a considerably deeper examination of a real process states. Because the determination of the state of a real process is a significant component of the design of MON (see sections 4 and 5), if it been available while MON was being developed (*sps* was installed around 18th of September) it would have been a great help in the aforesaid development. Unfortunately, almost everything done in *sps* that MON must also do has already been incorporated into MON. Here are some of the things *sps* does in common with MON:

- separates real processes into foreground and background processes
- determines when a process is waiting on a input from a terminal, or for output to be passed to the terminal (*sps* does not handle ring terminals)
- determines when a process is waiting to read from or write to a socket (MON goes further, finding the communications domain of the socket)
- determines when a process is in page-wait state
- determines when a process is selecting on multiple buffers

Use

*ps* & *sps* provide a snapshot of the system state. They are generally invoked by users to keep track of their processes, perhaps examining process status - sleeping, ready to run, stopped... - rather than as a tool for monitoring process resource consumption. They are of little use in a performance evaluation of the system because the values of performance indices produced relate to each process on the system at the snapshot moment rather than the system as a whole: cumulative statistics providing some overall measurement of the system are not kept.
A2.4 top

Capabilities and implementation

top is almost exactly like a program which runs ps periodically, but displays only the processes consuming the greatest amount of resources.

Use

Like ps, tops is of little use in a performance evaluation of the system because the values of performance indices produced relate to the specific processes on the system at the moment of the latest snapshot rather than the system as a whole over time: general cumulative statistics are not kept.

A2.5 systat, vmstat, iostat, netstat

Capabilities and implementation

These four performance tools are the most comprehensive bundled with Unix. They make periodic examinations of various metering information and control blocks maintained by the kernel, visually displaying the information at each interval.

The default systat display shows the processes getting the largest percentage of the processor.

iostat shows disk i/o statistics such as disk throughput and may operate in two modes:

1) report on statistics since boot time
2) report statistics at intervals, each set of statistics being for the immediately preceding interval

netstat symbolically displays the state of various network-related data structures, i.e. the state of active sockets; it may report the state at intervals.

vmstat "delves into the system and normally reports certain statistics kept about processes, virtual memory, disk, trap and cpu activity"[UPM (1)]. vmstat can operate in a number of different modes:

1) report on statistics since boot time
2) report statistics at intervals, each set of statistics being for the immediately preceding interval
3) report statistics as a running total.

For the purposes of MON, vmstat is interesting in that it includes a very simple breakdown of process states. It shows "the average number of processes that are runnable ('r'), in page wait ('p'), in disk wait other than paging ('d'), sleeping ('s'), and swapped out but desiring to run ('w')."[UPM (1)]
Use

`vmstat` and `ostat` are performance tools that may be used to provide values of performance indices for the system in general and for the disk subsystem. The class of performance indices that they measure are basically the "traditional" indices like CPU utilisation and disk subsystem throughput. The values of these performances indices would be suitable for analysis as part of a performance evaluation study. `vmstat` and `iostat` may also be used interactively to indicate the cause of heavy CPU or I/O activity at a particular moment.

`netstat` provides snapshots of the network subsystem. These snapshots indicate to users and the system administrator the causes of heavy network activity at a particular moment. `netstat` does not, however, produce any cumulative statistics on network activity.
Appendix 3  User guide for MON

mon [-i #] [-b] [-n] [-t] [-s] samples

The -i # option sets the sampling interval to #. If no interval is specified then a default is used.

samples is the number of samples that the user wishes to take

The remaining command-line options turn on diagnostic information:
- b diagnostics associated with disk transfers
- n addresses of various kernel data structures
- t diagnostics associated with terminals
- s all diagnostics

MON writes the collected data (the count array) to a file "monoutput" in the directory where the executable image of MON resides. MON is presently in the directory "/usr/users/honom·s/reg/monitor".

To standard output MON writes a short summary of some aspects of the sampling run, e.g.

| Terminal samples: | 26269 | Bad terminal samples: | 0 |
| Disk buffer samples: | 768 | Bad disk buffer samples: | 0 |
| Inode samples: | 0 | Bad inode samples: | 0 |
| Ring port samples: | 270 | Bad ring port samples: | 0 |
| Invalid run process samples: | 5 |
| Samples not recognised: | 0.5% |
| notrecog1 | 173 | notrecog2 | 300 |

In the summary sample refers to the state of one real process (misleadingly). A bad sample is one where a process is found to be waiting in a recognised structure, but then the actual state determination is not successful.
Invalid run process samples: the number of actual complete monitor samples which are aborted because of a significant change in the state of the run queues during analysis. "Samples not recognised" would be better phrased as "real process states not recognised".
#define TRUE 1
#define FALSE 0

#define X_PROC 0
#define X_NPROC 1
#define X_INODE 2
#define X_SWBUF 3
#define X_BUF 4
#define X_OS 5
#define X_WHICHOS 6
#define X_PROC_NPROC 7
#define X_DZ_TTY 8
#define X_TS29_TTY 9
#define X_NSWBUF 10
#define X_NBUF 11
#define X_NINODE 12
#define X_SELWAIT 13
#define X_MBUTL 14
#define X_LBOLT 15
#define X_PORT 16
#define X_U 17
#define X_FILE 18
#define X_TEXT 19
#define X_CFREE 20
#define X_CALLOUT 21
#define X_SWAPMAP 22
#define X_ARGMAP 23
#define X_KERNELMAP 24
#define X_MBMAP 25
#define X_NAMECACHE 26
#define X_QUOTA 27
#define X_DQUOT 28
#define X_DOMAINS 29
#define X_RSBUF 30

#define CPU_R 1
#define CPU_Q 2
#define PROC_C 3
#define PROC_T 4
#define OUTPUT 5
#define INPUT 6
#define PIO 7
#define NPIO 8
#define SWIO 9
#define BUFV 10
#define SELW 11
#define LINODE 12
#define EXFLOCK 13
#define SHFLOCK 14
#define LBOLT 15
#define SIGPAUSE 16
#define WCHILD 17
#define SUNIXR 18
#define SUNIXW 19
#define SINETR 20
#define SINETW 21
#define STOPPED 22

#define LASTBIT 22
#include "mon.h"
#include <sys/types.h>
#include <ctype.h>
#include <sys/time.h>
#include <signal.h>
#include <stdio.h>
#include <strings.h>
#include <a.out.h>
#include <sys/proc.h>
#include <sys/param.h>
#include <sys/inode.h>
#include <sys/file.h>
#include <sys/locl.h>
#include <sys/tty.h>
#include <sys/buf.h>
#include <sys/mbuf.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/protosw.h>
#include <sys/domain.h>
#include <sys/UOCR/dz.h>
#include <sys/UOCR/ttg.h>
#include <sys/UOCR/trans.h>
#include <sys/portinfo.h>
#include <sys/clock.h>
#include <sys/resource.h>

extern char *nl_names[1];
extern struct nlist nfnl; /* all because we can't init unions */
extern int nllen; /* # of nlist entries */

extern long mypld; /* process Id of the monitor */

long x_sym_contents; /* word at address the symbol value */

short ind_symval = 0 /* valid entries in sym_contents array */

u_long pstatewds, /* real process statewords - 32 bits per sw. */
    tstatewds; /* terminal 'virtual process' statewords */

#define MAXLOG 25

u_long counttable[ MAXLOG + 1 ];
#define PSETBIT(a, b) ( pstatewds[ (a) ] |= ( 1 << ( (b) - 1 ) ) )
#define ISPBITSET(a, b) ( pstatewds[ (a) ] & ( 1 << ( (b) - 1 ) ) )

#define INT_SYMS 12

u_long unrecog[ 1000 ]; /* table of unrecognised w_chan addresses */
short urtidx = 0; /* index into the table */

struct kern_int {
    u_long lower,
    upper;
    int case_selector;
} kern_int[ INT_SYMS ];

#define NOTSET -1

int bstty = 0, /* samples which were found in a tty structure, */
    /* but did not lead to a resource queue/run bit being set */
bsbuf = 0, /* as above, but buffers */
bsinode = 0, /* inodes */
bsport = 0; /* ring ports */
int tteam = 0, /* samples found in a tty struct */
beam = 0, /* samples found in a disk buffer */
insam = 0, /* Inode */
pteam = 0; /* port */
int other = 0; /* num wchans not recognised */
sleepnum = 0; /* total num interesting processes found sleeping */
zerosw = 0;
int kl = 0;
#define SETKERNEL(base, size, caselect) kern_int[ kl ].lower = (base); 
   kern_int[ kl ].upper = kern_int[ kl ].lower + (size); 
   kern_int[ kl ].case_selector = (caselect);
   kl++;
struct tty dz_tty[ NDZ * 8 ], /* VAX DZ11 terminal controller */
ts29_tty[ NTTG ]; /* UKC Cambridge ring terminals */
u_long off_outq;
int t_num_on; /* number of terminals found with a logged in user */
during a particular sample */
struct inode aninode; /* dummy structures; used in calculating the */
port_t aport; /* offsets below */
struct mbuf onembuf;

u_long off_pt_writeq, /* ring port write q offset in port structure */
off_exlock, /* offset of exclusive lock count in inode struct */
off_shlock, /* shared */
off_so_rcv, /* offset of the receive buffer header in a socket */
off_so_snd, /* offset of the send buffer header in a pipe */
off_mbdataprt, /* data area in an mbuf */
struct socket asocket; /* storage area for retrieval of a socket structure */
struct protosw aprotosw; /* protosw structure */
struct domain adomain; /* domain */

u_long unixdomainadd, /* virtual address of AF_UNIX domain struct */
inetdomainadd; /* virtual address of AF_INET domain struct */
#define S12_DZ_TTY (NDZ * 8 * sizeof(struct tty)) /* size in bytes of dz_tty[] */
#define S12_TS29_TTY (NTTG * sizeof(struct tty)) /* size in bytes of ts29_tty[] */
#define MAXTERM (NDZ * 8 + NTTG)

static int alarm_ringing,
    pause_mask, /* boolean: set by alarm_handler */
    seconds = 1, 21500 /* sampling interval secs */
    microseconds = 0; /* sampling interval microseconds */
int kmem; /* fd for /dev/kmem */

/* kmem structures */
#define ADRTOIDX(a, b, c) (((u_long) (c) - (u_long) (b)) / sizeof ( struct a ))
#define KADTOUAD(a, b, c) (((u_long) (c) - (u_long) (b)) + (a))
struct proc pproc; /* real process table */
int nproc; /* number of processes in proc table */
struct fg_proc
    short plindex; /* array index in proc table */
    short p_pgrp; /* process group leader */
    ) *fg_procs;
int fg_num;  \#/ number of foreground processes \#/ #define NQS 32

struct proc (  \#/ linked list of running processes \#/    struct proc *ph_lnk;  \#/     struct proc *ph_rlink;  
)     qs[ NQS ];  \#/ 32 run queues for dispatcher \#/ int whichqs;  \#/ bit mask summarizing non-empty run queues \#/ short nprint = 0,  \#/ flag for printing values in the namelist \#/ siprint = 0,  \#/ flag for printing general sample info \#/ tiprint = 0,  \#/ "terminal" buffer \#/ bprint = 0;  \#/ "buffer" \#/ int invalid_samples = 0;  \#/ as the name suggests \#/ 

main( argc, argv )
{
    int argc;
    char **argv;

    int num_samples,
        sample_count,
        len,
        cmp();

    /* initialisation stuff */
    num_samples = handle_args( argc, argv );
    init_nlist();
    nlist( "vmunix", nl );
    open_files();
    init_kmem_vars();
    if ( nprint ){
        printf( "values in the name list nl\n" );
        for ( len = 0; len < nlen; len++ ) {
            printf( "%w = %w, nl[len].n_un.n_name ");
            printf( N_FORMAT, nl[len].n_value );
            printf( "\n" );
        }
    }

    /* raise priority */
    mypid = getpid();
    setpriority( PRIO_PROCESS, mypid, -20 );
    start_interval_timer( seconds, microseconds );
    for ( sample_count = 1; sample_count < num_samples; sample_count++ ) {
        if ( siprint ) printf( "sample %d\n", sample_count );
        retrieve_kmem_structures();  \#/ find CPU run process \#/        if ( !cpu_run() ) {
            invalid_samples++;
            continue;
        }
        fg_num = tty_fg( fg_procs );  \#/ group real process statewords as foreground processes \#/        if ( siprint ) for ( len = 0; len < nproc; len++ ) {
            if ( !fg_procs[ len ].pindex ) break;
            printf( "%d  %d\n", fg_procs[ len ].pindex, fg_procs[ len ].p_pgrp );
        }
        pass_interesting( fg_procs, fg_num );
    \#/ pass of device/controller queues (if necessary) */
    \#/ analyse groups of process statewords */
    update_counttable();
    \#/ *** with sigpause the monitor is woken by any signal, 
    \#/ so sigpause again unless SIGNALRM */
    fflush( stdout );  \#/ flush per-sample info */
    while ( !alarm_ringing ) sigpause( pause_mask );
alarm_ringing = FALSE;

stop_alarm();

/* do last stuff */
if ( num_samples > 0 ) {
    printf( "Terminal samples:\t\%d\n", tsam, bstty );
    printf( "Disk buffer samples:\t\%d\n", dbuf, bdbuf );
    printf( "Node samples:\t\%d\n", insam, bsnodn );
    printf( "Ring port samples:\t\%d\n", ptsam, bspor );
    printf( "Nvalid run process samples:\t\%d\n", fnvalid_samples );
    printf( "Invalid samples not recognised: \%f\n", ((float) other / (float) sleepnum) * 100 );
    qsort( (char *) unrecog, urtldx, sizeof (u_long), cmp );
    for ( i = 0; i < urtldx; i++ ) printf("%8d \n", unrecog[i]);
}
writecounttable();

handle_args( argc, argv )

int argc;
char **argv;

int n_samp,
    len;
char *dotptr,
    mic[ 7 ];

/*
 * handle command line arguments
 */
if ( argc != 0 ) {
    switch ( **argv ) {
        case 'l':
            n_samp = 0;
            for ( argc--; argc > 0; argc-- ) {
                switch ( **argv ) {
                    case 'i':
                        argv++;
                        argc--;
                        if ( isdigit( **argv ) ) {
                            if ( dotptr = index( **argv, '.' ) ) {
                                *dotptr = '\0';
                                if ( ++dotptr ) {
                                    strncpy( mic, dotptr, 6 );
                                    for ( len = strlen( mic ); len < 6; len++ )
                                        mic[ len ] = '\0';
                                    microseconds = atoI( mic );
                                    
                                }
                                seconds = atoI( **argv );
                            } else err( 1 );
                            break;
                    case 'b': blprint++;
                    case 'n': nlprint++;
                    case 's': slprint++; tlprint++; blprint++;
                    case 't': tlprint++;
                    }}}}
break;
default:
    if ( lsdiglt( argv ) && n_samp == 0 ) n_samp = atoi( argv );
    break;
)
return( n_samp );
}

update_counttable();
{
    int i, j, pidx;
    for ( i = 0; i < fg_num; i++ )
        pidx = fg_procs[ i ].pindex;
    for ( j = 0; j < LASTBIT + 1; j++ )
        if ( pgstatewds( j ) == 0 ) zero[w]++;
        counttable[ t_num_on ][ j ]++; }
    for ( i = 0; i < nproc; i++ ) pstatewds[ j ] = 0;
}

writecounttable();
{
    FILE *outfile;
    if ( outfile = fopen( "monoutput", "w" ) ) err( 3, "monoutput" );
    for ( i = 0; i <= MAXLOG; i++ )
        fprintf( outfile, "%6d", i );
    for ( j = 0; j <= LASTBIT; j++ )
        fprintf( outfile, "\%6d", counttable[ i ][ j ] );
    fprintf( outfile, \"\n\" );
}

open_files();
{
    if ( ( kmem = open( "/dev/kmem", O_RDONLY ) ) < 0 ) err( 3, "/dev/kmem" );
}

init_kmem_vars();
{
    int set;
    short index,
    i;
    u_long temp,
    domainadd;
    off_outq = (u_long) &dz_tty->t_outq.c_cc - (u_long) dz_tty;
    off_exlock = (u_long) &aninode.i_exlockc - (u_long) &aninode;
    off_shlock = (u_long) &aninode.i_shlockc - (u_long) &aninode;
    off_pt_writeq = (u_long) &aport.pt_writeq - (u_long) &aport;
    off_mbdataprt = (u_long) &onembuf.m_dat[ 0 ] - (u_long) &onembuf;
    off_so_rcv = (u_long) &asocket.so_rcv.sb_cc - (u_long) &asocket + off_mbdataprt;
    off_so_snd = (u_long) &asocket.so_snd.sb_cc - (u_long) &asocket + off_mbdataprt;
    if ( biprint ) printf( \"RCVoff %3d, SNDoff %3d\n\", off_so_rcv, off_so_snd );
    if ( ( sym_contents = (u_long) malloc( nllen * sizeof( u_long ) ) ) ) <= 0 )
        err( 4, "sym_contents" );
    for ( index = 0; index < ( ( sizeof ind_syms ) / sizeof ( short ) ); index++ ) (  }
```c
l = ind_sym[ index ];
sym_contents[ l ] = getw( nlf[ l ].n_value );
if ( nlprint ) printf( "%s contents = %8x\n", nl[ l ].n_un.n_name, sym_contents[ l ] );

nproc = sym_contents[ X_NPROC ];
if( ( pstatewds = (u_long *) malloc( nproc * sizeof( u_long ) ) ) <= 0 )
  err( 4, "pstatewds" );
for( i = 0; i < nproc; i++ ) pstatewds[ i ] = 0;
if( ( proc = (struct proc *) malloc( nproc * sizeof( struct proc ) ) ) <= 0 )
  err( 4, "proc" );
if( ( fg_procs = (struct fg_proc *) malloc( nproc * sizeof( struct fg_proc ) ) ) <= 0 )
  err( 4, "fg_procs" );

for( set = 0, domainadd = sym_contents[ X_DOMAINS ]; domainadd != 0;
    domainadd = (u_long) domain.add.dom_next )
  (get_bytes( domainadd, (char *)&domain, sizeof( struct domain ) ));

switch( domain.add.dom_family ) {
  case AF_UNIX : unixdomainadd = domainadd; set++; break;
  case AF_INET : inetdomainadd = domainadd; set++; break;
}
if( set == 2 ) break;

if( blprint ) printf( "UnixD: %8x, INETD: %8x\n", unixdomainadd, inetdomainadd );

alarm_handler()
/**
** handler for SIGALRM
*/
{
  alarm_ringing = TRUE;
}

start_interval_timer( secs, usecs )
/**
** set ITIMER_REAL value and interval to be sampling interval
*/
int secs,
  usecs;
{
  int oldmask;
  struct itimerval itv;
  struct sigvec sv;

  itv.it_interval.tv_sec = itv.it_value.tv_sec = secs;
  itv.it_interval.tv_usec = itv.it_value.tv_usec = usecs;
  sv.sv_handler = alarm_handler;
  sv.sv_mask = sv.sv_onstack = 0;
  oldmask = sigblock( sigmask( SIGALRM ) );
  (void) sigvec( SIGALRM, &sv, (struct sigvec *) 0 );
  alarm_ringing = FALSE;
  (void) setitimer( ITIMER_REAL, &itv, (struct itimerval *) 0 );
  sigsetmask( oldmask );
  pause_mask = oldmask & ~sigmask( SIGALRM );
```
stop_alarm()
/
** clear ITIMER_REAL value and interval
*/

{  
struct itimerval itv;
int oldmask;

  oldmask = sigblock( sigmask( SIGALRM ) );
timerclear( &itv.it_interval );
timerclear( &itv.it_value );
setitimer( ITIMER_REAL, &itv, (struct itimerval *) 0 );
sigsetmask( oldmask );
}

retrieve_kmem_structures()
{
  get_bytes( sym_contents[ X_PROC ], (char *)proc, nproc * sizeof( struct proc ) );
  whichqs = getw( nll[ X_WHICHOS ].n_value );
  get_bytes( nll[ X_QS ].n_value, (char *)qs, NOS * sizeof ( struct proc ) );
  get_bytes( nll[ X_DZ_TTY ].n_value, (char *)dz_tty, SIZ_DZ_TTY );
  get_bytes( nll[ X_TS29_TTY ].n_value, (char *)ts29_tty, SIZ_TS29_TTY );
}

getw( address )

unsigned long address;
{
  int word;

  lseek( kmem, (long) address, L_SET );
  if ( read( kmem, (char *) &word, sizeof word ) < sizeof word ) err( 2, "kmem" );
  return( word );
}

get_bytes( address, buf, num_bytes )

int address,
  num_bytes;
char *buf;
{
  lseek( kmem, address, L_SET );
  if ( read( kmem, buf, num_bytes ) < num_bytes ) err( 2, "kmem" );
}

cpu_run()
/
** apply dispatcher technique to determine (pseudo) CPU run process (if one)
*/
{
  int index,
  runproc;
  u_long pr_addr,
  ptbl;

  if ( ( index = ffs( whichqs ) - 1 ) >= 0 ) (  
    pr_addr = (u_long) qs[ index ].ph_link;
}
ptbl = sym_contents[X_PROC];
if ( ( pr_addr < ptbl ) || ( pr_addr > ( ptbl + nproc * sizeof (struct proc) ) ) ) {
    return( 0 );
} else {
    runproc = ADRT0IDX( proc, ptbl, pr_addr );
PSETBIT( runproc, CPU_R );
    if ( ( siprint ) printf( "Next process to run is %5d\n", proc[ runproc ].p_pid );
        return( 1 );
    }
}

tty_fg( fgprocs )
struct fg_proc *fgprocs;
{
    short t_prgrps[ MAXTERM ],
    i,
    index,
    loc;
    int compar();
    t_num_on = 0;
    getprgrps( t_prgrps, &t_num_on, NDZ, &c_dev );
    getprgrps( t_prgrps, &t_num_on, NTTG, &ts29_dev );
    qsort( (char *)t_prgrps, t_num_on, sizeof( short ), compar );
    for ( i = 0, index = 0; index < nproc; index++ ) {
        if ( ( loc = bsearch( t_prgrps, t_num_on, proc[ index ].p_pgrp ) ) >= 0 ) {
            fgprocs[ i ].index = index;
            fgprocs[ i ].p_pgrp = t_prgrps[ loc ];
            i++;
        }
    }
    return( i );
}

getprgrps( t_prgrps, index, tty_num, tty_ptr )
short *t_prgrps,
    *index;
int tty_num;
struct tty *tty_ptr;
{
    short tempdx;
    for ( tempdx = *index; tty_num > 0; tty_num--, tty_ptr++ ) {
        if ( tprint ) {
            printf( "t_line %5d t_dev %5d %5d %11x %11x %3x %3x \n",
                tty_ptr->t_line,
                major( tty_ptr->t_dev ),
                minor( tty_ptr->t_dev ),
                tty_ptr->t_flags,tty_ptr->t_state,tty_ptr->t_delt,tty_ptr->t_col );
        }
        if ( ( tty_ptr->t_line == NTTYDISC ) && ( tty_ptr->t_state & TS_ISOPEN ) ) {
            if ( tprint ) printf( "%5d %5d\n", tty_ptr->t_dev, tty_ptr->t_pgrp );
            t_prgrps[ tempdx ] = tty_ptr->t_pgrp;
            tempdx++;
        }
    }
    *index = tempdx;
}

pass_interesting( fg, n_fg )
struct fg_proc *fg;
int n_fg;
(int fg_index,
pix;

for ( fg_index = 0; fg_index < n_fg; fg_index++ ) {
pix = fg[ fg_index ].pindex;
switch (proc( pix ).p_stat ) {
  case SSLEEP: sleepanalysis( pix ); break;
  case SIDL : PSETBIT( pix, PROC_C ); break;
  case SZOMB : PSETBIT( pix, PROC_T ); break;
  case SRUN : if ( !SPBITSET( pix, CPU_R ) ) PSETBIT( pix, CPU_Q ); break;
  case SSSTOP : PSETBIT( pix, STOPPED ); break;
  case SWAIT :
}
}
sleepanalysis( pldx )

int pldx;

(int l,
domore,
a_bit,
base,
tidx;

u_long wchan,
offset;

struct tty */tt_base;

sleepnum++;
if ( proc( pldx ).p_flag & SPACE ) {
  if ( blprint ) printf( "PAGEWAIT\n" );
PSETBIT( pldx, PIO );
  return;
}

domore = NOTSET;
for ( i = 0; i < INT_SYM; i++ ) {
  wchan = (u_long) proc( pldx ).p_wchan;
  if ( wchan < kern_intl i .upper && wchan >= kern_intl i .lower ) {
    base = kern_intl i .lower;
    domore = kern_intl i .case_selector;
    break;
  }
}
if ( domore != NOTSET ) {
switch ( domore ) {
  case X_U:
    if ( proc( pldx ).p_pri == PSLEEP ) PSETBIT( pldx, SIGPAUSE );
    break;
  case X_PROC:
    if ( proc( pldx ).p_pri == PWAIT ) PSETBIT( pldx, WCHILD );
    break;
  case X_DZ_TTY:
  case X_TS29_TTY:
    tsam++;
    if (( offset = ( wchan - base ) % sizeof( struct tty ) ) != 0 ) {
      if ( ( offset - off_outq ) == 0 ) {
        tt_base = (domore == X_DZ_TTY ) ? dz_tty : ts29_tty;
tidx = ADRTIDX( tt_base, wchan - offset );
        if ( tiprint ) printf( "Outq!! BUSY? %ix \n", tt_base[ tidx ].t_state & TS_BUSY );
        if ( tt_base[ tidx ].t_state & TS_BUSY ) PSETBIT( pldx, OUTPUT );
      } else batty++;
    }
    else PSETBIT( pldx, INPUT );
    break;
  case X_BUF:
bsam++;
if ( offset == ( wchan - base ) % sizeof( struct buf ) ) == 0 ) {
        switch( proc[ pldx ].p_pri ) {
            case PRIBIO: PSETBIT( pldx, NPIO ); break;
            case PRIBIO + 1: PSETBIT( pldx, BUFW ); break;
            default: bsbuf++;
        }
    } else bsbuf++;
if ( biprint ) printf( "DB: Offset %d Pri %d\n", offset, proc[ pldx ].p_pri );
break;
case X_SWBUF:
    bsam++;
    if ( offset == ( wchan - base ) % sizeof( struct buf ) ) == 0 ) {
        if ( biprint ) printf( "SB: proc flag: %8x", proc[ pldx ].p_flag );
        switch( proc[ pldx ].p_pri ) {
            case PSWP: PSETBIT( pldx, SWIO ); break;
            case PRIBIO + 1: PSETBIT( pldx, BUFW ); break;
            default: bsbuf++;
        }
    } else bsbuf++;
    if ( biprint ) printf( "SB: Offset %d Pri %d\n", offset, proc[ pldx ].p_pri );
break;
case X_RSWBUF: PSETBIT( pldx, SWIO ); break;
case X_SELWAIT: PSETBIT( pldx, SELW ); break;
case X_MBUTL:
    offset = ( ( wchan - base ) % sizeof( struct mbuf ) );
    if ( biprint ) printf( "SOCK: offset: %3d\n", offset );
    if ( offset == off_so_rcv ) {
        get_bytes( wchan - offset + off_mbdataprt, (char *)&asocket, sizeof(struct socket) );
        get_bytes( (u_long)asocket.so_proto, (char *)&aprotosw, sizeof(struct protosw) );
        if ( (u_long)aprotosw.pr_domain == unixdomainadd ) {
            if ( biprint ) printf( "UNIX domain, READ\n" );
            PSETBIT( pldx, SUNIXR );
        } else if ( (u_long)aprotosw.pr_domain == inetdomainadd ) {
            if ( biprint ) printf( "INET domain, READ\n" );
            PSETBIT( pldx, SINETR );
        }
    break;
    }
    if ( offset == off_so_snd ) {
        get_bytes( wchan - offset + off_mbdataprt, (char *)&asocket, sizeof(struct socket) );
        get_bytes( (u_long)asocket.so_proto, (char *)&aprotosw, sizeof(struct protosw) );
        if ( (u_long)aprotosw.pr_domain == unixdomainadd ) {
            if ( biprint ) printf( "UNIX domain, WRITE\n" );
            PSETBIT( pldx, SUNIXW );
        } else if ( (u_long)aprotosw.pr_domain == inetdomainadd ) {
            if ( biprint ) printf( "INET domain, WRITE\n" );
            PSETBIT( pldx, SINETW );
        }
    break;
    }
break;
case X_LBOLT: PSETBIT( pldx, LBOLT ); break;
case X_INODE:
    insam++;
    offset = ( ( wchan - base ) % sizeof( struct inode ) );
    if ( offset == 0 ) {
        PSETBIT( pldx, LINODE );
        break;
    } else if ( (u_long)aprotosw.pr_domain == unixdomainadd ) {
        if ( biprint ) printf( "UNIX domain, WRITE\n" );
        PSETBIT( pldx, SUNIXW );
    } else if ( (u_long)aprotosw.pr_domain == inetdomainadd ) {
        if ( biprint ) printf( "INET domain, WRITE\n" );
        PSETBIT( pldx, SINETW );
    } break;
break;
case X_INODE:
    insam++;
    offset = ( ( wchan - base ) % sizeof( struct inode ) );
    if ( offset == 0 ) {
        PSETBIT( pldx, LINODE );
        break;
    } else if ( (u_long)aprotosw.pr_domain == unixdomainadd ) {
        if ( biprint ) printf( "UNIX domain, WRITE\n" );
        PSETBIT( pldx, SUNIXW );
    } else if ( (u_long)aprotosw.pr_domain == inetdomainadd ) {
        if ( biprint ) printf( "INET domain, WRITE\n" );
        PSETBIT( pldx, SINETW );
    } break;
if ( offset == off_shlock ) {
PSETBIT( pldx, SHFLOCK );
break;
}
bslnode++; break;

case X_PORT:
ptsam++;
if ( ( offset = ( wchan - base ) % sizeof( port_t ) ) == off_pt_writeq )
PSETBIT( pldx, OUTPUT );
else bsport++;
break;
default: ++other;
}
}
else {
++other;
if ( urtidx < 1000 ) {
unrecog( urtidx ) = wchan;
urtidx++;
}
}

compar( a, b )
short *a, *b;
{
return( ( *a < *b ) ? -1 : 1 );
}

bsearch( array, size, element )
short *array;
int size,
element;
{
short a, b, c;
a = 0;
if ( size ) c = b = (short) ( size - 1 );
else return( -1 );
while ( array[ c ] != element ) {
c = ( a + b ) / 2;
if ( c == a ) break;
if ( array[ c ] > element ) b = c;
else a = c;
}
return ( array[ c ] == element ) ? c : -1;
}

err( e_num, which )
/*
** errors!
*/

int e_num;
char *which;
{
switch ( e_num ) {
case 1:
fprintf( stderr, "usage: -l <number>\n" );
break;
case 2:
fprintf( stderr, "unable to successfully read %s\n", which );
break;
case 3:
fprintf( stderr, "unable to open file %s\n", which );
break;
break;
case 4:
    fprintf( stderr, "unable to allocate heap memory for %s\n", which );
    break;
default:
    fprintf( stderr, "error\n" );
}
fflush( stderr );
exit( 0 );
}
Various statically allocated data structures in the kernel contain information that we are interested in. The locations (virtual addresses) of these data structures will be only be fixed between compilations of the kernel. However, the names of the data structures should remain the same over much longer periods. (In fact, changing the name of a global data structure would usually require additional modifications to the kernel source code — all references to the data structure would have to be altered accordingly). If we know the names of the kernel data structures, then we may find their kernel addresses examining the symbol table information that is included with the /vmunix object file.

This technique is used by ps, and the functions that implement the method have been copied from the source code of ps to this file, with a couple of modifications.

```c
#include "mem.h"
#include <stdio.h>
#include <a.out.h>
#include <sys/types.h>

char *nl_names[] = {
    "_proc",
    "_nproc",
    "_inode",
    "_swbuf",
    "_buf",
    "_qs",
    "_whichqs",
    "_prochnpPROC",
    "_dtty",
    "_ts29_tty",
    "_nswbuf",
    "_nbuf",
    "_ninode",
    "_selwait",
    "_mbuf",
    "_lhold",
    "_port",
    "_u",
    "_file",
    "_text",
    "_cfree",
    "_callout",
    "_swapmap",
    "_argsap",
    "_kernelmap",
    "_abmap",
    "_namecache",
    "_quota",
    "_dquot",
    "_domains",
    "_rsbwbuf",
    "
};

struct nlist *nl;    /* all because we can't init unions */
int nllen;            /* # of nlist entries */

/*
* since we can't init unions, the cleanest way to use a.out.h instead
* of nlist.h (required since nlist() uses some defines) is to do a
* runtime copy into the nl array -- sigh
*/
init_nlist();
```
register struct nlist *np;
register char **namep;

nllen = sizeof nl_names / sizeof (char *);  
np = nll = (struct nlist *) malloc(nllen * sizeof (struct nlist));
if (np == NULL) {  
    fprintf(stderr, "out of memory allocating namelist\n");  
    exit(1);
}
namep = &nl_names[0];
while (nllen > 0) {  
    np->n_un.n_name = *namep;
    if (**namep == '\0')  
        break;
    namep++;
    np++;
}


#include "nlist.h"

/*
 * nlist - retrieve attributes from name list (string table version)
 */

nlist(name, list)
char *name;
struct nlist *list;
{
    register struct nlist *p, *q;
    register n, m;
    int maxlen, nreq;
    FILE *f;
    FILE *sf;
    off_t sa;
    off_t ss;
    int type;
    struct exec buf;
    struct nlist space[BUFSIZ/sizeof (struct nlist)];
    char nambuf[BUFSIZ];
    maxlen = 0;
    for (q = list, nreq = 0; q->n_un.n_name && q->n_un.n_name[0]; q++, nreq++) {
        q->n_type = 0;
        q->n_value = 0;
        q->n_desc = 0;
        q->n_other = 0;
        n = strlen(q->n_un.n_name);
        if (n > maxlen)
            maxlen = n;
    }
    f = fopen(name, "r");
    if (f == NULL)  
        return (-1);
    fread((char *)&buf, sizeof buf, 1, f);
    if (N_BADNAG(buf)) {  
        fclose(f);
        return (-1);
    }
    sf = fopen(name, "r");
    if (sf == NULL) {  
        fclose(f);
        return(-1);
    }
    sa = N_SYMOFF(buf);
    ss = sa + buf.a_syms;
    n = buf.a_syms;
    fseek(f, sa, 0);
    while (n) {  
        m = sizeof (space);
        if (n < m)  
            break;
    }
}
n = n;
if (fread((char *)space, m, 1, f) != 1)
    break;

n -= m;
for (q = space; (m -= sizeof(struct nlist)) >= 0; q++) {
    if (q->n_un.n_strx == 0 || q->n_type & N_STAB)
        continue;
    
    /* since we know what type of symbols we will get,
       we can make a quick check here -- crl */
    type = q->n_type & (N_TYPE | N_EXT);
    if (type == N_ABS)
        continue;
    fseek(sf, ss+q->n_un.n_strx, 0);
    fread(nambuf, maxlen+1, 1, sf);
    for (p = list; p->n_un.n_name && p->n_un.n_name[0]; p++) {
        if (strcmp(p->n_un.n_name, nambuf) == 0) {
            p->n_value = q->n_value;
            p->n_type = q->n_type;
            p->n_desc = q->n_desc;
            p->n OTHER = q->n_ander;
            --nreq;
            break;
        }
    }
}

all done:
fclose(f);
fclose(sf);
return (nreq);