DESIGN AND IMPLEMENTATION OF AN EVENT MONITOR FOR THE UNIX OPERATING SYSTEM

By

SUSAN CHUI-SHEUNG CHAN

B. Sc., University of British Columbia, 1982

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

April 1987

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date April 30, 1987.

Abstract

Tuning a computer system effectively requires prior studies on the performance of the system. There are different types of tools available to measure a system: hardware, firmware and software. This thesis presents the design and implementation of an event monitor, which is one type of software tools.

The event monitor was developed on a SUN1 workstation running *UNIX* 4.2bsd version 1.4. Six types of events were selected to be measured, namely transactions, logins/logouts, pageins, pageouts, disk I/Os and forks/exits. The operating system was modified to include probes to trap these events. For a final testing of the event monitor, it was ported and installed onto a SUN3 workstation running *UNIX* 4.2bsd version 3.2. Measurements collected were analyzed by a capacity planning package *condenser*. The results give an indication of the system workload and the system performance. Benchmarks were also set up to measure the overhead incurred by the event monitor.

Contents

List of Figures

List of Tables

Acknowledgement

I would like to thank my supervisor Dr. Sam Chanson for his guidance and cooperation.

Thanks must also go to Dr. Son Vuong who served as the second reader for this thesis.

My greatest gratitude should go to Jee Fung Pang, who has given me numerous valuable ideas, and who adapted the condenser package to run on the event data collected.

Lastly, I am grateful for all help I obtained from Frank Pronk and Rick Sample. They have answered many of my questions which may otherwise take me a long time to figure out.

Chapter 1

Introduction

1.1 Thesis Motivations and Objectives

The studies on *Tuning, Measurements and Performance Evaluations of Computer Systems* have gained recognition in Computer Science research. In most cases, tuning a system improves its performance, and sometimes the improvement can be considerable. However, in order to be able to tune a system effectively, the system must first be evaluated. We are specifically interested in the case where the system already exists and available to be measured.

There are many types of tools which can be used to measure the performance of a system. Each type of tools has its merits as well as drawbacks, and hence the choice of tool depends entirely on the type of measurements desired. For system usage data, such as cpu and disk utilizations, it is most accurate to collect the relevant information as the events occur. This will require either a hardware measurement tool which can be quite expensive, or a piece of software, an *event monitor*, to be inserted into the

operating system to monitor the activities of the system. The thesis deals with the design and implementation of a software event monitor.

The *UNIX*[1] operating system is used extensively both in research and commercial environments. It is a very powerful operating system running on a range of computers from microprocessors to the mainframes. The operating system itself, however, does not maintain adequate measurements statistics. Consequently, performance evaluations on the *UNIX* operating system have been found difficult. However, unlike other operating systems, *UNIX* was written in the high level language C, which makes it easier to decode than other operating systems written in assembler language, and its source is available. In addition, the kernel of the operating system is fairly compact and manageable.

This thesis was motivated by the preceding considerations. It was felt that an event monitor can be developed on a *UNIX* 4.2 bsd operating system, which will capture and record events as they occur. The data collected by such a monitor can be used to characterize the system's workload, and can be fed into other capacity planning packages to study system performance.

Three of the key concerns in Performance Evaluations are the amount of interferences added to the system, the amount of overhead incurred, and the accuracy of the data collected. When designing the event monitor, special attempts were made to cope with the above problems. In addition, the event monitor can be used interactively,

[1] *UNIX* is a trademark of AT&T Bell Laboratories

allowing users to turn it ON or OFF at will. It also provides users the flexibility to select the types of events to be measured, and the number and size of buffers to be used. Since the event monitor is implemented as a separate process, it can be ported to other versions of *UNIX* with ease.

1.2 Thesis Outline

The thesis is organized as follows. Following the introduction in Chapter 1, the reader is presented with the different techniques that are available for measuring computer systems in Chapter 2. The design and implementation of the event monitor developed for *UNIX* is described in Chapter 3. The installation of the event monitor including where to insert probes to trap some selected events is discussed in Chapter 4. Measurements collected on a real system are presented in Chapter 5. Evaluations of the event monitor, and possible future enhancements conclude the thesis in Chapter 6. Appendix A contains a sample session and thus can be used as a simple guide for first time users of the event monitor. Appendix B contains a detailed module design, which may be helpful for programmers who may want to modify the system. Appendix C contains the data structures of some pertinent *UNIX* system variables.

Chapter 2

Measurement Techniques

There are three main categories of measurement tools: *hardware*, *firmware* and *software*. Each type of tools has its own characteristics, and is suitable for collecting different sets of data. Since the *event monitor* developed is one type of software tools, software measuring techniques are presented in details. Hardware and firmware tools are also presented for comparison.

2.1 Criteria for a Good Measurement Tool

Two of the most important criteria for a good measurement tool are its *efficiency* and *accuracy*. A measurement tool should be efficient, and should not impose too much extraneous load on the system. Equally important, the data collected by the tool should reflect accurately a system's workload. The accuracy of a tool is determined in part by its *resolution*, which is the maximum frequency at which events can be detected and correctly recorded. Inevitably, any measurement tool that does not use an external

processor, no matter how perfect its design, will add to the system load, and the data collected will necessarily contain an error margin. Nevertheless, if the overhead is acceptable and can be measured, and the error margin is low, the measurement tool is still useful. Hardware tools, as discussed below, are superior to software and firmware tools in both efficiency and accuracy, though they are generally more expensive and difficult to use. One cannot, however, compare the different types of tools as if they have equivalent capabilities and applications. Even though some measurements can be taken by any one of the tools, one type is usually preferred. The characteristics of each type of tool are discussed in the following sections.

2.2 Hardware Tools

Hardware monitors are electronic devices connected to specific system points where they can detect voltage levels or pulses characterizing the events to be measured. Since they are completely external to the system, they do not interfere with the system's activities, or do they add to the system's workloads. The only energy consumption is at the point where the connection occurs, but the amount is usually considered negligible. Because of their negligible interference and high resolution – capable of detecting high frequency (1MHz or higher) events – their accuracy is generally higher than the other tools.

Hardware monitors are connected to the system via *probes*. These probes are usually

circuits of high impedance, capable of detecting the change in voltage levels. Care must be taken when installing the probes, because at some critical points of the system, the addition of even a slight electrical load can introduce serious system disturbances. After the signals have been collected by the probes, they are sent through an *event filter*, a logic module which processes the signals. From the event filter, signals are then sent to a set of *counters*, one counter for each specific event. At the end of the measurement session, or periodically, depending on the duration, contents of the counters are written onto a mass storage device, usually disk or tape. The analysis of these data, a process known as *data reduction* is usually done off-line to produce reports for capacity planning.

A hardware monitor is represented pictorially in Figure 2.1.

Hardware monitors are more sensitive to changes of the system on a physical level, and since they hardly interfere with the system's activities, they are ideal for measuring microscopic events of high frequencies. Examples of such events are: the transfer rate of a channel, CPU and device utilizations and the seek activities of a disk unit. However, it may be difficult to relate the microscopic events to higher level events. Also, installation of a hardware monitor is usually very complicated, because it involves physical connections to the machine; hence, it will require good knowledge of the hardware architecture to be able to place the probes properly.

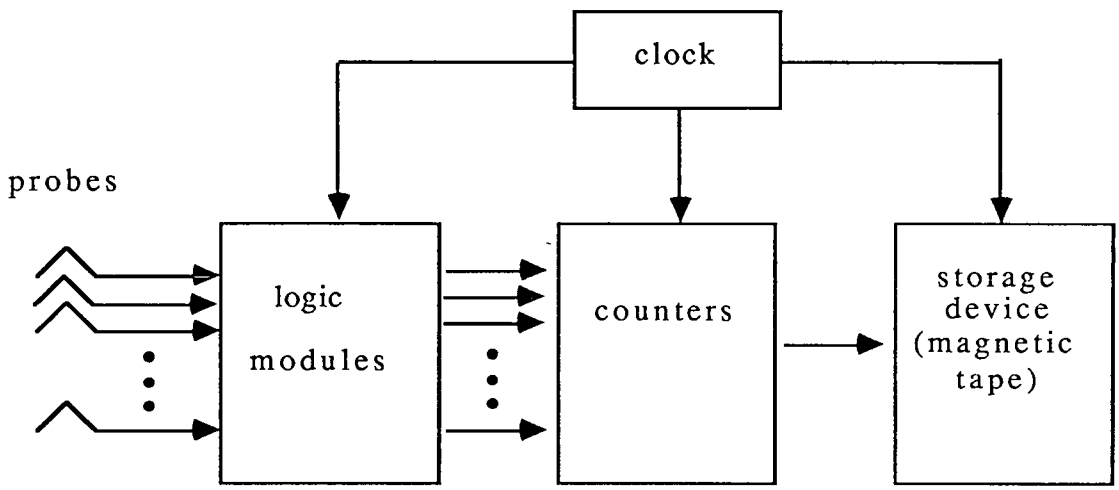

Figure 2.1: A Hardware Monitor

2.3 Firmware Tools

Firmware tools are measurement tools that are micro-programmed into the system. They are not as common as either hardware or software tools. Their many characteristics, such as interference, accuracy, resolution and ease of use, are in between that of hardware and software tools. Installations of firmware tools are fairly complicated, and their costs are higher than those of software tools.

2.4 Software Tools

Software tools are programs inserted into the operating system to monitor its activities. This may be done in one of three ways:

1. addition of a program

2. modification of the software to be measured

3. modification of the operating system

The first method is generally preferred because it makes it easier to use the tool when required, and remove it when not needed. The integrity of the operating system is also preserved. The second method requires the insertion of codes at some critical points of the program to be measured. The last method is the most cumbersome, as it

involves rewriting part of the operating system, and is usually done because the existing O/S does not provide some of the necessary data.

Since a software monitor competes for resources with the rest of the system, it introduces interferences. In particular, the data collected by the software monitor, which is generally large, has to be stored in main memory, and written out periodically onto secondary storage devices, and thus interrupting the normal I/O activities of the system. The collection and compilation of statistics also consume CPU time. The design of the software monitor can greatly influence the above factors. A good software tool should satisfy the following requirements: [Kole71]

- it should be able to extract quantitative and descriptive data from the system.
- it should require as little modification to the operating system as possible.
- its data collection techniques should not alter the workload characteristics and hence the performance of the measured system.
- it should require as little memory as possible.

Due to the amount of interference, software tools are only good when measuring events of a much lower frequency. It is appropriate for obtaining descriptive and quantitative data, such as page table entries and file access information. Within the software

tools domain, there are two distinct measurement techniques: *event detection* and *event sampling*, which are discussed in the following sections.

2.4.1 Event Detection

An *event* in the computer system is defined to be a change in the system's states. Examples of events can be the start and end of I/O operations, users logging on/off the system and the recognition of a page fault.

In event detection, a piece of software, known as an *event monitor*, is inserted into the operating system, which is capable of collecting and compiling information when an event occurs. Special code, commonly known as *probe* or *hook*, are also placed strategically in various spots of the operating system. When an event of interest occurs, this code will cause control to be transferred to the monitor routine. Inside the monitor, relevant information are collected and written into a *buffer* area, which is a temporary storage. Depending on the size of the buffer, and the frequencies of events, the buffer is emptied periodically onto a secondary storage device. This technique preserves the order in which the events occur and provides the necessary data associated with each event. Detailed and accurate workload characterizations can thus be obtained.

Since an event monitor usually deals with a large volume of data, buffer space is extremely critical. In most machines, buffer space is limited, and hence writing to secondary device has to be done frequently. When an event occurs, if the buffer is full

but the transfer of its contents has not been completed, then the question arises as to whether the system should wait for the completion of the transfer. If the system waits, it will be slowed down appreciably; if not, some event data may be lost. It is up to the implementor to decide how to handle this situation.

2.4.2 Event Sampling

Event sampling is a statistical approach to measuring the behaviour of a computer system. Instead of measuring every event as it occurs, this method collects only selected samples for analysis from which one can usually estimate, with a high degree of accuracy, parameters that can characterize the activities of the computer system.

The main advantage of sampling is that it produces a much smaller set of data, thus reducing the overhead and simplifying its analysis. The problem of buffer management is also less critical. The amount of interference is comparably lower than event detection.

There are two types of sampling techniques, *count* sampling and *time* sampling. In count sampling, a measurement routine is periodically invoked after a fixed number of predefined events have occurred. The more common technique is time sampling, where measurement routines are invoked at pre-specified time intervals. Sampling intervals can be constant or random. Random sampling is particularly useful when the distribution of the data is unknown.

Sample size has to be fairly large in order that the data collected be representative. Sample interval should be short such that the distribution of workload is homogeneous. Sampling is suitable for collecting resource usage data, and particularly those data in which the sequencing of events is unimportant.

Chapter 3

Event Monitor

This chapter discusses the design and implementation of the event monitor for the *UNIX* operating system. Issues of importance are: the implementation environment, its buffer management scheme, data structures, process synchronization, critical sections, security measures, compatibility and dependability. For the purpose of brevity, the term *monitor* is henceforth used synonymously with *event monitor.*

3.1 Environment

3.1.1 Hardware

The monitor was implemented on a 68000 based SUN workstation[1] named *ubc-andrew.* It is one of the early SUN1 workstations which SUN Microsystems no longer manufactures. Even though the implementation did not involve the manipulation of the very low level machine architecture, a knowledge of its structure is useful when

[1]SUN Workstation is a trademark of Sun Microsystems Inc.

designing the monitor and making modifications to the *UNIX* kernel.

The SUN 68000 Board uses two buses: an internal synchronous bus for communicating with local memory and I/O devices, and the Multibus system bus for referencing additional memory and offboard I/O devices. Seven levels of interrupts, numbered 1 through 7, are recognized by the SUN processor. Level 7 has the highest priority, and level 1 has the lowest. Interrupts are acknowledged and processed for all priority levels greater than the current processor priority level contained in the 68000 status register.

ubc-andrew has 1 Mbyte of main memory. There is a Memory Management Unit (MMU) in the workstation which provides address translation, protection, sharing and memory allocation for multiple processes executing on the 68000 CPU. The MMU consists of a context register, a segment map and a page map. Virtual address from the CPU are translated into intermediate addresses by the segment map and then into physical addresses by the page map.

The page size is 2048 bytes, the segment size is 32K bytes (giving 16 pages per segment), and up to 16 contexts can be mapped concurrently. The maximum logical address space that can be mapped simultaneously is 2M bytes.

3.1.2 Operating System

Inasmuch as *monitor* is one type of software tools, it is to be inserted into the operating system. The target operating system is *UNIX* 4.2bsd. A very brief overview

of the relevant aspects of the operating system is presented below.

The *UNIX* O/S provides the processes abstraction. A *process* is a program in execution. The system starts up with the *init* process as process 1, and the *page daemon* as process 2. New processes can be created by the system *fork* command. There are two types of processes: those that reside in the user space, and those that are within the kernel. These two types of processes do not share the same address space; hence, variables cannot be shared between the two layers. Kernel processes have access to all kernel variables, but each user process has its own stack for variables. Communication amongst user processes is via pipes and sockets, while communication between the layers is via system calls. Special kernel routines such as *copyin* and *copyout* are required to copy variables in and out of the kernel space.

Associated with each process is a data structure called the *process structure.* (See Appendix C). The process structures of all running processes are linked together in a process table. Each process structure contains everything that is necessary to know about a process when it is swapped out, such as its unique process identifier (an integer), scheduling information and pointers to other control blocks. Associated with each user is a user structure, which contains information such a user id, group id and resource usages for each user. (See Appendix C). These two structures provide most of the required data for *monitor.*

Memory of the system is available in the forms of buffers, linked up in three separate

queues. The first queue, which contains all the super blocks of the file system, must be kept permanently in main memory. The second queue contains the cache, while the third queue contains I/O buffers for different devices, and also some empty buffers. Buffers for *monitor* usage come from the third queue. The structure of a *UNIX* buffer can be found in Appendix C.

3.1.3 Implementation Language

The language used to develop *monitor* is the C programming language. The choice of this language is obvious, as almost the entire *UNIX* operating system is written in C. Initially, it was felt that parts of *monitor* may have to be coded in assembler language to increase efficiency, but as yet, it has not been found necessary.

3.2 Structure of the Event Monitor

3.2.1 Overall Design

The monitor is implemented partly as a user process, and partly as a kernel process. Because most of the events of interest, such as page faults and disk I/Os, happen within the kernel, the processing of events is most appropriately done within the kernel. To improve efficiency, the large quantity of I/Os and buffer management, are also handled within the kernel. There is a user command interface at the user level, which processes user commands, and then does a context switch to pass parameters into the kernel portion of the monitor. Within the kernel, the probe routine awaits the occurrences of

events. The inter-relationship between *monitor* and the rest of *UNIX* is illustrated in Fig. 3.1. A detailed module design is given in Appendix B.

3.2.2 Buffer Management

Buffer management is handled entirely within the kernel. The monitor allows users to select the number and size of the buffers, but some knowledge of the hardware architecture is essential when optimal usage of buffers is desired. For instance, with the SUN architecture, blocks are always allocated in size of 2048 bytes. Hence, it is only sensible to choose buffer size to be multiples of 2048. The number of buffers should also be chosen wisely, such that there is always an overlap between filling buffers and writing them out onto secondary storage device. The amount of available memory on a machine also governs the number of buffers to be used. For example, with a machine that has only one megabyte of memory, it is not logical to allocate more than 16K bytes for monitor usage. With the preceding considerations in mind, *monitor* is designed to supply suitable default values for buffer size and number of buffers for users who do not have an in-depth knowledge of the machine architecture.

At the outset of *monitor*, buffers are allocated and linked up cyclically within the kernel. The *ring* approach is chosen over maintaining two separate link lists of empty and full buffer queues, because it simplifies the task of pointer re-assignments when moving a buffer from the empty queue to full queue, and vice versa. Only three

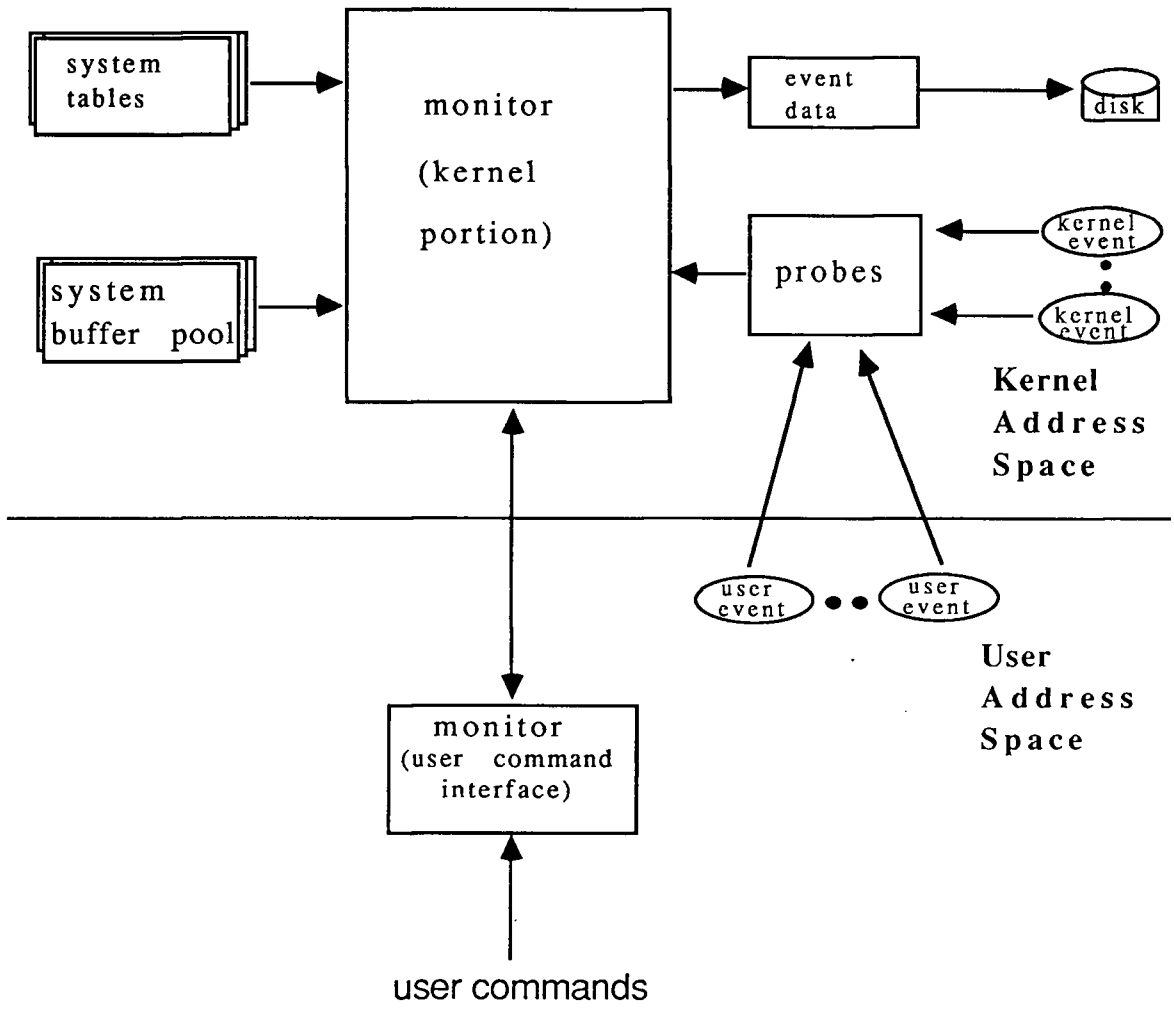

Figure 3.1: Relationship of *monitor* with rest of UNIX

pointers are maintained at any one time: pointer to the current buffer, pointer to the logical head of the entire buffer pool, and pointer to the logical head of the full buffer queue. The current pointer, *mp*, points to the buffer being used to collect data. The full pointer, *fp*, points to next buffer to be written out onto the secondary device. Pictorially, the buffer management system of *monitor* is represented in Fig. 3.2.

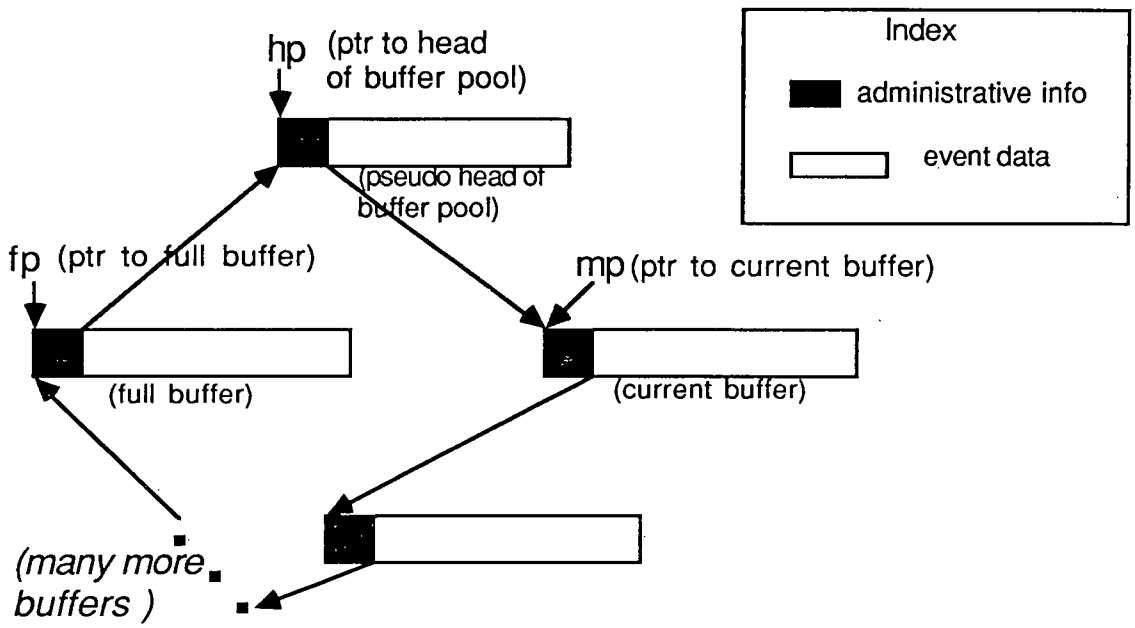

Figure 3.2: Buffer Management Scheme of *monitor*

The first 10 bytes of each buffer are reserved for administrative purposes. The actual data storage area is thus *bufsize* - 10. The administrative information required is summarized in the *monitor_buf* structure in Fig. 3.3. The *filled* field is a status flag,

```
struct monitor_buf {
        short filled;
        struct buf *nextbp;
        struct monitor_buf *nextmp;
};
```

Figure 3.3: monitor_buf structure

turned on when the buffer is full, and off when the buffer is empty. This flag tells the output routine if there are more buffers to write, and the probe routine if there are empty buffers to use. Two separate pointers are also required to point to the next buffer structure. The necessity of these two pointers may need some explanation. From Appendix C, it can be seen that the actual data area of a *UNIX* buffer is at the address pointed to by the field *b_addr*. The monitor buffer's administrative data starts at this address. The address of the entire buffer structure, however, must also be maintained, because the system needs the address when releasing the buffer storage.

A buffer is written out onto disk or tape as soon as it gets filled. When the monitor is turned off, all buffers are deallocated and returned to the system for other usages.

3.2.3 Data Structures

The organization of data collected by *monitor* is illustrated in Fig. 3.4.

A *header* record is written for each invocation of the monitor. Its purpose is for

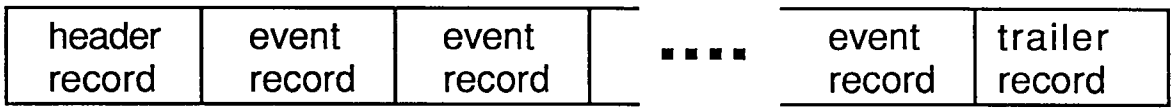

Figure 3.4: Layout of data collected by *monitor*

identification of the system being measured, and to record the start time of *monitor*. This information is useful for the capacity planner when analyzing the event data. The structure of the header record is shown in Fig. 3.5.

As each event occurs, an *event* record is written into the buffer storage. Structure for the event record is shown in Fig. 3.6. There are two parts to an event record, *fixed* and *variable*. The fixed portion contains pertinent information for each event, such as event id, user id, process id, real time and cpu time. The event id is computed as event_group*256+event_type, where event_group is one of the possible events; event_type is either the start or end of the event. (See Table 4.1). The event id is a compact way to represent the event_group and event_type. The variable portion is for auxiliary information, and varies for each type of event. A table of events measured on a SUN workstation and associated auxiliary information can be found in Chapter 4.

The *trailer* record is another time stamp, indicating the termination time for *mon-*

```
struct header_record {
    short month;
    short day;
    short year;
    short hour;
    short min;
    short sec;
    short userno;
    char username[32];
    char version [16];
    short cpuid;
    short memory;
    short nusers;
    short mon_version;
};
```

Figure 3.5: The Header Structure

```
struct event_record{
   short len;
   short event_id;
   short user_id;
   short pid;
   unsigned long cpu_time;
   unsigned long real_time;
   short *auxinfo;        /* may or may  not present */
}
```

Figure 3.6: The Event Record

itor. The difference between termination time and start time is the elapsed time for the monitor session.

Associated with each monitor buffer is a 2-byte field, *lost_event*, which counts the number of events lost while waiting for the next available buffer. If *lost_event* is too high, the capacity planner may select to discard the buffer of event records for not being representative of the entire workload.

3.3 Probe Routine

The probe routine is the heart of *monitor*, awaiting the occurrences of events. There are two entry points to this routine: *probe* to be called from the user level, and *sys_probe* to be called from the kernel level. The two entry points are necessary, because one would like to trap user level events as well as kernel events. The tasks for probe can be summarized as follows:

1. check that *monitor* is turned on.

2. check that the particular event is selected to be monitored.

3. if no empty buffers is available, increment *lost_events*.

4. otherwise, fill an event_record and add to current buffer.

3.4 Process Synchronization

For maximum efficiency, *monitor* strives to completely overlap the tasks of filling buffers with event records and writing buffers out onto secondary devices. There are two separate routines to assume the two tasks: *sys_probe* to fill buffers, and *write_buf* to invoke an output routine to write buffers out. The two processes are synchronized by two primitives *sleep* and *wakeup.* Depending on the workload of the system, the type of events to be measured, the speed of I/O drivers, and the number of buffers available, one process may have to wait for signals from the other process before it can continue. For instance, if the events of interest occur at such a rapid pace that all available buffers are filled, then *sys_probe* has to wait for empty buffer before it can collect more event records. The proper calling sequence for these primitives are:

```
sleep(chan, prio)
caddr_t chan;
int prio;

wakeup( chan )
caddr_t chan;
```

The first argument of sleep is by convention the address of a kernel data structure, and the second argument is a scheduling priority. When a process goes to sleep, it

gives up the processor until a wakeup occurs, at which time the process enters the scheduling queue at priority prio. The priority, if negative, also prevents the process from being prematurely awakened by some exceptional event, such as a signal. Hence, when *sys_probe* has to wait for empty buffers, it goes to sleep until it is waken up by *write_buf* when empty buffers become available.

3.5 Critical Section

The critical section problem arise when several processes try to asynchronously change the contents of a common data area. The updated area may not, in general, contain the intended changes if protection against contention of competing processes is not provided. In the case of *monitor*, the common data area is the buffer storage. To safeguard the buffer area from contention when it is being filled with an event_record, its interrupt level is raised to level 6, and the old level is restored when it is done. In *UNIX* 4.2bsd, the routines *spl0, spl1, ... spl6* can be used to raise or lower the interrupt levels.

3.6 Security Measures

Presently, *monitor* is designed such that only the super-user of the system can invoke it. Of course, it can easily be modified to give access rights to any user. The potential danger of the second approach, however, cannot be overlooked. If the user

of *monitor* does not have a clear concept of the nature of the events being measured, system workload can increase appreciably and system performance will deteriorate. Since the main objective of *monitor* is to collect data for performance evaluation studies, it is best to grant access permission only to a user with the above objective in mind.

3.7 Compatibility and Dependability

The data collected by *monitor* is designed mainly to be used by the capacity planning package *condenser*, developed by Jee Fung Pang as his master thesis for the Department of Computer Science, University of British Columbia [Pang86]. The event record may have to be modified if it is to be used by other packages, but the tasks should be minimal.

Chapter 4

Installation

This chapter presents the installation procedures of the event monitor on a *UNIX* operating system running on a SUN workstation. To trap events, *probes* have to be inserted in strategic locations of the operating system. For illustrative purposes, six different types of events are selected to be measured and the locations of probes for those events are also discussed.

4.1 Installation Procedures

The event-monitor was developed on a 68000 based SUN workstation running *UNIX* 4.2 bsd, but it can easily be transported and installed in other compatible *UNIX* operating systems. The installation procedures are outlined as below:

1. Copy the object modules for the event monitor, which includes both the user-level and kernel portions, to the target machine. The object modules for these routines are collectively stored in a file *monitor.o*, with its associate source in file

monitor.c, and header file *monitor.h*.

2. Make entries in the system entry table for the two kernel routines that are also callable by users from the user-level. These routines are *setmonitor* and *probe*, and they take 6 and 5 arguments respectively.

3. Make entries in the system C library for the above two routines, such that they can be invoked from the user level of the operating system, which is written in C.

4. Modify the kernel to trap the selected events. See Sec. 4.2.

5. Recompile the *UNIX* kernel linking the resident kernel portion of the event monitor with the rest of the system.

6. Install and load the new *UNIX* kernel.

If the installation is successful, and the target machine meets the minimum memory requirement, then *monitor* is ready to be used. Refer to Appendix A for a user guide.

4.2 Probes

In order to trap the selected events, *probes* have to be inserted at the precise locations in the operating system where the events occur. As mentioned previously, there are two separate entry points to the probe routine: from the user level and from the

kernel level. The calling sequence to these two entry points are:

```
probe (event-group, event-type, auxinfo, auxlen);
int event-group;
int event-type;
short *auxinfo; /* pointer to auxiliary info */
int auxlen; /* length of auxiliary info */

sys-probe (event-group, event-type, auxinfo, auxlen, kern);
int event-group;
int event-type;
short *auxinfo;
int auxlen;
int kern; /* 1 if it's kernal event; 0 otherwise */
```

There are flags within the kernel, such as KERN_FORK, KERN_PAGEIN, KERN_TRANS, which are set when the corresponding event is turned on. Before the sys_probe routine is invoked, the appropriate flag is checked to insure that the event to occur is selected to be measured. This way, unnecessary invocations of sys_probe can be avoided. For example, to trap a pagein, the following statements are inserted into the operating system at the precise location:

```
if (KERN_PAGEIN)
    sys_probe(KERN_PAGEIN,START_EVENT,&pf,2,1);
```

4.3 Events to be Measured

For illustrative purpose, six different types of events are selected to be measured. They are transactions, logins/logouts, pageins, pageouts, disk I/Os and forks/exits. The events and their associated auxiliary information are summarized in Table 4.1.

Event Group	Event Description	Associated Types	Aux info	Auxlen
1	TRANSACTION	start, end	none	0
2	LOGIN/LOGOUT	start, end	none	0
3	PAGEINS	start, end	page no.	2
4	PAGEOUTS	start, end	page no.	2
5	DISK I/O	start, end	device no.	2
6	FORK/EXIT	start, end	none	0

Table 4.1: Event Descriptions and their Auxiliary Information

The following sections give a brief description of how *UNIX* handles the different events, and the precise locations of the different probes. The descriptions are based on the system UNIX 4.2bsd.

4.3.1 Transactions

A transaction is defined as an interaction with the system, whether it is input or output. For instance, when the user issues a shell command, he is initiating a transaction. When the command is acted upon by the system, the transaction is terminated. The system handles input differently, depending on the modes it is in. In NORMAL mode, such as within the shell, input is not processed until a carriage return is encountered. But in RAW and CBREAK modes, which are used within editors, input is processed a character at a time; character is also output without processing. The START_EVENT for NORMAL mode is therefore different from the other modes, and occurs when a carriage return is entered. The END_EVENT for all modes is the output of the first character.

Probes are inserted in *tty_input* and *tty_output* in the file *tty.c*. The statistics collected at START_EVENT and END_EVENT can be used to calculate system response time and users' think times.

4.3.2 Logins and Logouts

On each terminal port available for interactive use, *init* forks a new process, which attempts to open the port for reading and writing. The open succeeds when the terminal is turned on, or a telephone call is accepted by a dial-up modem. The program *getty* is then executed by init.

Getty initializes terminal line parameters and prompts the user to type a login name. The login name is passed as an argument to another program, *login.* Login encryptes the typed password and compares it with the encrypted password string for the login name found in file /etc/passwd. If they are the same, login sets the uid of the process to that of the user logging in. The START_EVENT of the LOGINOUT probe is placed at the location after the uid is set, in file *login.c.* Login then executes a shell, a command interpreter. When the user logouts, the shell process dies. The END_EVENT of the LOGINOUT probe is placed in the routine *goodbye()*, in the file *sh.c.*

The LOGINOUT probes are the only two probes that are placed in the user level. The login and logout events usually occur less frequently than the other events.

4.3.3 Paging

Memory pages are arranged into frames, which are represented by the *core map* or *cmap.* This map records the disk block corresponding to a frame that is in use by a process, and also maintains a free-list of frames that are not used by any process.

UNIX 4.2bsd uses a modified *Global Clock Least Recently Used (LRU)* algorithm for memory management [Quar85]. A software clock hand linearly and repeatedly sweeps all frames of main memory that are available for paging. The reference bit of a page is marked invalid, i.e., reclaimable, when the clock hand sweeps over it. If the page is

refereneced before the clock hand next reaches it, a page fault occurs, and the page is made valid again. However, if the page has not been referenced when the clock hand reaches it again, it is reclaimed for other use. Various software conditions are also checked before a page is marked invalid.

Pagein occurs when a process need a page, and the page is not mapped into a memory frame. This causes the kernel to allocate a frame of main memory, map it into the appropriate process page, and read the proper data into it. Pageins do not necessarily mean a disk I/O. If the required page is still in the process' page table, but has been marked invalid by the last pass of the clock hand, it can be marked valid and used without any I/O transfer. Pages can similarly be retrieved from the memory free-list. If the page has to be fetched from disk, it must be locked during the I/O transfer to prevent data from being corrupted. The pagein probe is inserted in the pagein routine in the file vm_page.c. The probe is to catch only those pagins that involve a disk I/O.

The pageout algorithm is the LRU clock hand, which was described earlier. The algorithm is implemented in the *pagedaemon*, which is process 2. The pagedaemon's purpose is to keep the memory free-list large enough, such that paging demands on memory will not exhaust it. This process spends most of its time sleeping, but a check is done several times per second to see if action is necessary. Whenever the number of free frames falls below a threshold, the process is awakened; thus, if there is always a

lot of free memory, the pagedaemon imposes no load on the system because it never runs.

With systems having a large main memory, the clock hand may take a long time to complete a cycle. Thus the second encounter of the hand with a given page has little relevance to the first encounter, and the pagedaemon will have difficulty finding reclaimable page frames. In 4.3bsd, a second clock hand, which follows behind the first clock hand, reclaims pages that are marked invalid by the first hand. (see Fig. 4.1)

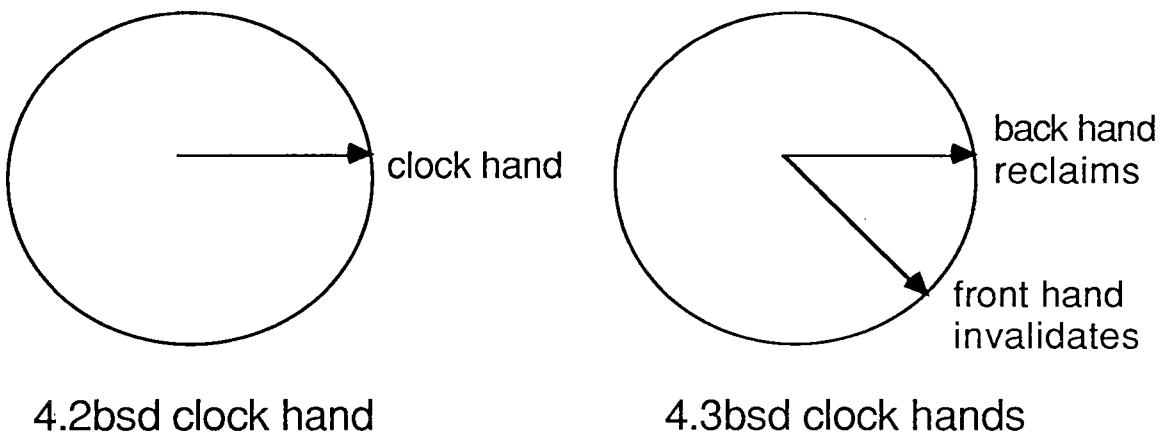

Figure 4.1: The LRU clocks hands for UNIX 4.2bsd and UNIX 4.3bsd

The pageout probe has been placed to trap the reclamation of pages. It is inserted in the *pageout()* routine in the file *vm_page.c*.

4.3.4 Disk I/Os

This category of events include I/O associated with the block devices, namely disk and magnetic tapes. Attached to each device driver is a list of buffers, with each buffer assigned a device name and a device address. This list of buffers also acts as a cache for the block devices, as it is always searched first for a desired block on a read request. If the block is found, the data is made avialable without any physical I/O. If the block is not found, the least recently referenced buffer is used for the transfer. On a write request, the correct buffer is located in the cache and marked "dirty". Physical I/O is deferred until the buffer is reclaimed for a read request.

Buffer I/O routines are collectively stored in file *ufs_bio.c*. Probes are inserted in *bread()*, *breada()*, *bwrite()*, and *bdwrite()*. Only events that cause actual physical I/Os are being trapped.

4.3.5 Forks and Exits

Processes in *UNIX* are created by the *fork* system call. During a fork, a new entry is allocated in the process table. A process structure is created for the new process (the child process), and all relevant information are copied from the parent process. This copying of information preserves open file descriptors, user and group identifiers, signal handling, and other similar properties of a process. The process id of the child process, however, is different from that of its parent. Fork returns 1 to the child process, and 0

to the parent process.

A process is terminated by the *exit* system call. When a process is to be terminated, its parent is notified, its resource utilization statistics are recorded, and all resources allocated are returned to the system.

For our purpose of capacity planning, FORK/EXIT is treated as a single event. The START_EVENT is FORK, and its probe is inserted in the *fork1()* routine in the file *kern_fork.c*, after the process structure is allocated and information copied. The END_EVENT is EXIT, and its probe is inserted in the *exit()* routine in the file *kern_exit.c*. Forks and exits of processes happen very frequently within the *UNIX* operating system; hence, this event usually generates a large amount of event data.

Chapter 5

Testing

As a final testing of the event-monitor, it is ported from the development system, which is a SUN1 workstation running UNIX 4.2bsd version 1.4, and installed on a SUN3 running UNIX 4.2bsd version 3.2. The event-monitor was turned on to measure the system for approximately 6 hours. Data collected were analyzed by the capacity planning package *condenser*. In addition, benchmarks were set up to determine the interference of the monitor on the system.

5.1 Measuring a Real System

The system to be measured is a SUN3/260 running UNIX 4.2bsd version 3.2, known as *ubc-csgrads*. It is a production system that supports most of the research work of the graduate students in the Computer Science Department at UBC. The event-monitor was installed overnight, and was turned on between 9:00 a.m. to 3:00 p.m. on Monday April 6, 1987 to measure the six types of events as outlined in the previous chapter.

The results were analyzed by *condenser*, which reports system utilization on a per user basis, as well as overall system performance. Only overall system performance patterns and those of three classes of users (light, medium, heavy) of the *ubc-csgrads* are reported here. The following is a a brief description of the different types of statistics collected. For more in-depth discussions, refer to Jee Fung Pang's thesis [Pang86].

Definition of statistics:

- *Response time* is the average response time for all interactive users in the class.
- *Think time* is the average think time for all interactive users in the class.
- *True I/O* are those I/O operations (including queue wait times) not caused by page faults.
- *True CPU* are CPU usage excluding any CPU time for page faults.
- *Physical Page Fault* is the number of page faults that actually cause one or more I/O operations.
- *Virtual Page Fault* is the number of page faults that do not cause any I/O operations at all.
- *Login* is the average session length of a terminal user, or the average length of a child process. *Logout* is the average time between logins, i.e. the average time between two terminal sessions or the average time between two child processes.

- *CPU Utilization* is the percentage of time that the CPU is utilized during the measurement session.

- *Disk Utilization* is the percentage of time that the Disk is utilized during the measurement session.

- *Page Fault Rate* is the number of page faults per second.

The following is a summary of measured statistics for *ubc-csgrads.*

Monitor started on 04/06/87 09:00:36

Monitor Version: 1 on 4.2BSD

Monitor User Name: schan Monitor User Number: 1022

CPU: ubc-csgrads Memory: 8 Megabytes Maximum users: 12

Elapsed time = 22377.760 seconds

Number of events processed = 208536

Total blocks/buffers read = 1804

System Parameters	Light	Medium	Heavy	Overall
Response time (secs)	2.748	6.946	6.870	3.312
Think time (secs)	9.393	76.050	28.517	13.871
True CPU (ms)	1.318	36.927	8017.077	3440.912
True I/O (ms)	6411.319	6681.434	9972.275	9080.095
Physical Page Fault (number)	0	0	2	2
Virtual Page Fault (number)	198	130	301	629
Login (secs)	n/a	n/a	n/a	10.4142
Logout (secs)	n/a	n/a	n/a	131.6857

Table 5.1: Summarized measurement results of *ubc-csgrads*

The global statistics are given in table 5.2. They indicate that the bottleneck of the system is probably the disk, with a utilization of 57.9837 percent, as compared to the CPU, which has a utilization of only 43.4541 percent. The system has virtually no page faults (0.0282 page fault/sec), which is probably due to the relatively large

Statistic	True	Virtual	Total
CPU Utilization (percent)	43.4539	0.0002	43.4541
Disk Utilization(percent)	57.9837	0.0000	57.9837
Page Fault Rate (no. page faults/sec)	0.0001	0.0281	0.0282

Table 5.2: Global Statistics of *ubc-csgrads*

physical memory of the machine (8 Megabytes). The workload of the system during the measurement session was considered heavier than normal, since it was end of term and students were finishing term projects and assignments. However, it should be noted that the statistics are the mean over the 6-hour session, including lunch hours. Peak loads during small intervals in the session will have much higher utilization figures. Nevertheless, the system was not saturated. The results suggest that the system can accommodate more users than its current maximum. The number of lines connected to *ubc-csgrads* via the switch can also be increased from the present 12.

Classification of users into the three classes (light, medium and heavy) is based on CPU usage. One would expect response time will increase from light to heavy users. It is, however, not necessarily the case. In the execution of a *fork*, for example, the parent process has to wait for its child process to die, and hence a large response time, but it is not using any CPU. Similarly, any process that initiates a pipe also has to wait for the other process to complete before exiting. This will explain the average

response time for the medium user (6.946 secs) which is slightly higher than the heavy user (6.870 secs).

The true I/O times collected included disk wait times. When calculating disk utilization the queuing times have been removed.

5.2 Benchmarks

Benchmarks are artificial and reproducible workloads processed by the system. They enable meaningful comparison of system performance before and after system changes by providing the same workload for each case. To measure the interferences added to *ubc-csgrads* due to the installation of *monitor*, two sets of benchmarks were executed on the system under three different conditions:

1. *monitor* is not installed.
2. *monitor* installed but not turned on.
3. *monitor* installed and turned on with six events selected to be measured.

The contents of the two benchmark files are :

Benchmark I:

- echo 'date' Begin MIXTEST 'NUSERS'
- mtime "cc -DSYS3 -c driver.c" &
- sleep 10
- mtime -u2 "ed < edscript" &
- sleep 5

- mtime "sort -r words -o /dev/null" &
- sleep 50
- mtime "nroff -man nroff.1 > /dev/null" &
- sleep 60
- mtime "cp editor.c editor.cc" &
- sleep 24
- mtime "pwd" & mtime "cd /tmp" & fork 5 &
- sleep 15
- mtime "who" & mtime "tsh" &
- sleep 17
- mtime "cc -DSYS3 -c editor.c" &
- mtime "cat driver.c | tr $a-z$ $A-Z$ > driver.cc" &
- wait
- echo 'date' End MIXTEST 'NUSERS'

Benchmark II:

- date
- mtime "cc -DSYS3 driver.c -o driver" &
- mtime -u4 "ed < edscript" &
- sleep 5
- mtime "sort -r words -o /dev/null" &
- mtime "nroff -man nroff.1 > /dev/null" &
- mtime "cc -DSYS3 editor.c -o editor" &
- sleep 4

- mtime "pwd" & mtime "cd /tmp" & fork 20 &
- mtime "who" & mtime "tsh" &
- mtime "cp editor.c editor.cc" &
- mtime "cat driver.c | tr $a-z$ $A-Z$ > driver.cc" &
- wait
- date

The two benchmarks differ in the order in which the commands are executed, the number of users executing each command, and the time interval between each command. *mtime* is a program written in C that will spawn subshells to time the execution of a given command by any number of users as given by the -u option. Each user is independently running the command the number of times as indicated by the -r option. *mtime* is modelled after the standard UNIX *time* command, and enjoys comparable accuracy. Internally, /bin/sh is called to excecute the "command", and the subprocess times are returned via the system call *time*.

The two sets of benchmarks were executed on a single-user SUN3 machine, *ubc-csfs2*, with no other workload on the system. Each benchmark was executed ten times, and the CPU time averages for each command were used in the comparison. Variability in response times is not reported, because response times depend largely on the timing and the order that the commands were executed in. Results of the benchmarks are summarized in table 5.3. The three conditions that the benchmarks were executed in are as outlined earlier.

command	condition 1	condition 2	condition 3
cc -DSYS3 -c driver.c	7.832	7.836	7.835
ed edscript	0.133	0.134	0.133
sort -r words	0.149	0.150	0.152
nroff -man nroff.1	3.967	3.967	3.968
cp editor.c editor.cc	0.117	0.117	0.118
pwd	0.084	0.083	0.089
cd /tmp	0.017	0.016	0.015
who	0.162	0.164	0.164
cc -DSYS3 -c editor.c	12.048	12.047	12.049
cat driver.c driver.cc	0.534	0.535	0.535

Table 5.3: Variability in CPU time (secs) under the three conditions

From the tabulated results, it can be seen that the largest increase in CPU time from conditon 1 to condition 3 is 5 msec., in the *pwd* command. In some commands, however, the CPU time in condition 3 is even lower than in condition 1. With most commands, whether it is a large compilation or text formatting job, or a simple shell command, there is hardly any variability at all. The accuracy of these results depends on the accuracy of the shell time command, but they are good indications that the event-monitor does not increase the system load noticeably.

Chapter 6

Concluding Remarks

This thesis discusses the design and implementation of an event-monitor to be used in a UNIX operating system to trap and record events as they occur. The data collected is used for capacity planning. The event-monitor was developed on a SUN1 workstation, and ported to a SUN3. The tool is evaluated based on four criteria: scope, interference, accuracy and portability. Suggestions for possible future enhancements conclude the thesis.

6.1 Tool Evaluation

6.1.1 Scope

The scope of a measurement tool is the classes of events it can detect. Events can be macroscopic, for example, the number of users logging on, or microscopic, such as the utilization of disks and cpu. Obviously, the wider the scope of a measurement tool, the greater is its range of applications. As seen in the earlier chapters, the scope of

monitor is very wide. Probes can be inserted in any part of the kernel, as well as in user programs. The only limitation with *monitor* is that it cannot be used to probe events lower than the kernel layer, such as hardware events that occur at the instruction and macro layers. Being a software tool, it is also not suitable to measure events that occur with very high frequency.

6.1.2 Interference

Every measurement tool extracts energy from the system. Interference can be classified in terms of resources and memory utilization. For *monitor*, the amount of interference introduced depends heavily on the workload, and on the types of events selected. When designing *monitor*, special care was taken to allow the tool to make optimal use of resources available. For instance, user can select the number and size of buffers to suit the particular system. Output data can reside on disk if space is available, but they can also be stored on magnetic tape to conserve space. Interference introduced on *ubc-csgrads* was found to be minimal in the previous chapter.

6.1.3 Accuracy

The accuracy of a tool is often reflected by the error affecting the data collected. Due to the interference caused by the event-monitor, the time statistics – cpu time and real time – collected in the event record necessarily constitute an error margin, which fortunately should be constant for all types of events. Since events are always probed

in pairs, START_EVENT and END_EVENT, analysis performed using the differences in statistics between each pair of probes would eliminate the effect of the overhead. Generally, if probes are well-placed within the operating system, data collected by *monitor* is very accurate, since events are always trapped as they occur.

6.1.4 Portability

monitor is implemented mainly as an individual kernel process, which functions independently from the rest of the system. It does not depend on the very low level registers or machine architecture. As a result, *monitor* can easily be ported to other systems running compatible versions of UNIX. For this thesis, *monitor* was initially developed on a SUN1 workstation running UNIX 4.2bsd version 1.4, but was later installed on a SUN3 running UNIX 4.2bsd version 3.2.

6.2 Future Enhancements

This event-monitor is designed to run on a centralized system with a single processor. With the advent of distributed computing and computer systems with multiple cpus, there is a need for measurement tools for the more complicated systems. *monitor* can be adapted to collect statistics for systems with multiple cpus, and maybe to measure the traffic flow across networks.

Bibliography

[Ferr83] D. Ferrari, G. Serazzi, A. Zeigner, *Measurement and Tuning of Computer Systems*, Prentice-Hall, Englewood Cliffs, N.J., U.S.A., 1983.

[Hu86] I. Hu, *Measuring File Access Patterns in UNIX*, Performance Evaluation Review, Vol. 14, #2, 1986.

[Kern78] B.W. Kernighan, D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J., U.S.A., 1978.

[Kern84] B.W. Kernighan, R. Pike, *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, N.J., U.S.A., 1984.

[Kole71] K. Kolence, *A software view of measurement tools*, Datamation, pp. 32-38, Jan. 1971.

[Lion77] J. Lions, *A Commentary on the UNIX operating system*, Dept. of Computer Science, University of New South Wales, 1977.

[Pang86] J.F. Pang, *Characterizing User Workload for Capacity Planning*, M.Sc. Thesis, University of British Columbia, Oct. 1986

[Quar85] J.S. Quarterman, A. Silberschatz, J.L. Peterson, *4.2BSD and 4.3BSD as Examples of the UNIX System*, Computing Surveys, Vol. 17, #4, Dec. 1985

[Ritc78] D.M. Ritchie, K. Thompson, *The UNIX time-sharing system*, Bell Sys. Tech. J., Vol. 57, #6, Jul. 1978.

[SUN82] *Programmer's Reference Manual for the SUN Workstation* Version 1.0, SUN Microsystem Inc., Oct. 1982.

[SUN83] *SUN 68000 Board*, Revision B, SUN Microsystem Inc., Feb. 1983.

[SUN85] *System Interface Manual for the SUN Workstation*, SUN Microsystem Inc., Oct. 1985.

[Thom78] K. Thompson, *UNIX Implementation*, Bell Sys. Tech. J., Vol. 57, #6, Jul. 1978.

Appendix A

User Guide

This appendix serves as a quick user guide to the user command *monitor*. An overview of the command line options is given, followed by a sample session with the monitor.

The command line options for *monitor* are as follows:

monitor -on | -off | -status | -help [-buffer nbufs bufsize] [-event evtnos] [-file fname]

The meanings of the options are elaborated below. Options separated by commas are equivalence.

-on | *-off* | *-status, -s* | *-help, -h*

One of the above options must be selected, otherwise the monitor command is void. *-on* is to turn monitor on, *-off* is to turn monitor off, *-s* to query the status of the monitor, and -h prints out help information.

-buffer, -b nbufs bufsize

This is an optional parameter. It takes two arguments, number of buffers (nbufs) and size of each buffer (bufsize) to be used by *monitor.* If not given, defaults suitable to the particular system is supplied.

-events, -e evtnos

This is also optional. Monitor can accommodate up to pre-defined maximum number of events. Evtnos can be a list of event-groups separted by blanks. Currently, only events 1 to 6 are defined. They are, respectively, transaction, login/logout, pagein, pageout, disk I/O and forks/exits. Only those events of the corresponding evtnos are being monitored.

-file, -f filename

An optional parameter, which takes an output file name as an argument. If the output file exists the user will be queried before it is overwritten. If the file does not exist, it will first be created. The default output file is *monitor.out.*

A sample session is given below. User's input is given in bold face.

% **monitor -on -b 2 2048 -e 1 3**

MONITOR version 1.0 - April 1987

Number of buffers to be used : 2

Size of each buffer (bytes) : 2048

Output file name : monitor.out

Events to be monitored are :

1. Transaction

3. Pagein

% **monitor -on**

MONITOR is already turned on

% **monitor -s**

Status of MONITOR version 1.0 - April 1987

Number of buffers being used : 2

Size of each buffer (bytes) : 2048

Events being monitored are :

1. Transaction

3. Pagein

% **monitor**

Usage: monitor -on | -off | -s | -h [-b nbufs bufsize] [-e evtnos] [-f filename]

% **monitor -off**

MONITOR is turned OFF

Event data collected in file monitor.out

%

Appendix B

Module Design

This appendix gives a brief module design of the event-monitor. It may serve as a guide for programmers who may want to modify or update the system.

There are two portions to the event-monitor, one residing in the user address space, and the other in the kernel address space. The relationship between the two parts is illustrated in Fig. 3.1.

For the modules below, a design/execution level number is included to indicate the module's position in the *monitor's* hierarchical algorithm. A brief description of each module's algorithm is also included.

User Level

main(argc,argv) - level 0

int argc; char **argv;

The main program that acts as the interface between user and the kernel portion

of *monitor*. It records the startup time of the *monitor*, sets default values for the buffer size, number of buffers and the output file. It then processes the user command line, allowing user options to override the default values. It creates the output file if it does not exist, or it prompts the user if he likes the file to be emptied. If everything seems fine, it echoes the parameters and options of *monitor* that are in effect. It then forks a child process which does a context switch into the kernel. The parent process exits and dies.

GetTime(tmbuf) - level 1

struct timedat *tmbuf;

It is called from the main program to return the time of day. It uses the system routines *gettimeofday* and *localtime* to get and convert time to the form mm:dd:yy:hh:mm:ss.

MapArg(arg) - level 1

char *arg;

It is called from the main program to return the next argument on the user command line. A -1 is returned if it is an illegal argument.

Kernel Level

setmonitor(tmbuf,fdes,nbuf,bsize,flag,event_on) - level 0

struct timedat *tmbuf;

int fdes, *nbufs, *bsize, flag, event_on;

This is a system call, i.e., callable from the user level. It is invoked by the main routine in the user level. If the option is -status, the current values of *mon_numbufs*, *mon_bufsize* and *kern_event_on* are copied out to the user address space. If the option is -off, *haltmonitor* is called to stop the monitor. If the option is -on, then relevant system variables are set with values passed from the user level. Other routines are also invoked to allocate buffer storage, to write the header record, and to write out a full buffer when one is available.

writebuf() - level 1

As long as the monitor is turned on, this routine will make sure that any full buffers will get written out, by calling the routine *dump_buffer_to_file*. When the monitor is turned off, it also calls *monitor_off* to do the final clean up.

clear_buffer(bp,bphead) - level 1

struct buf *bp;

struct monitor_buf *bphead;

It clears the storage area pointed to by *bp*, such that it can be used for storing

event data. The position of the buffer within the cyclic buffer pool is preserved.

dump_buffer_to_file() - level 2

This routine calls the low level I/O routine *rdwri* to dump buffer onto a peripheral device, usually a disk file or a magnetic tape. The device is specified by the inode pointer.

getstorage() - level 1

This routine transforms a *UNIX* buffer to a *monitor* buffer. The first 10 bytes of the *monitor* buffer is set aside for administrative purpose. The rest of the buffer is casted to type short.

PutHeader(tm) - level 1

timedat *tm;

The header record is filled with the necessary information, and the *sys_probe* is invoked with HEADER as the event-group.

haltmonitor() - level 1

This routine is invoked when the monitor is to be turned off. It writes the trailer record, resets all the kernel variables, and wakes up any buffers waiting to get filled.

monitor_on() - level 1

This routine is invoked at the start of the monitor. Its main task is to allocate the specified storage buffers, and link them up into a cyclic pool. It initializes the current buffer pointer, *mp*, the full buffer pointer, *fp*, and the pseudo head of the buffer pool, *hp*.

monitor_off() - level 1

This routine is invoked at the end of the monitor session to return all the buffer storage areas to the system.

probe(group,type,auxinfo,auxlen) - level 0

int group, type, auxlen;

short *auxinfo;

This is another system call. It is designed to be invoked from the user layer, such that events of interest happening in the user address space can also be recorded. An example of such an event is login.

This routine does not collect any statistics. It calls *sys_probe* to process the event.

sys_probe(group,type,auxinfo,auxlen,kern) - level 1

int group, type, auxlen, kern;

short *auxinfo;

This is the central routine that processes an event-record. It first checks if buffer storage is available. If no buffer is available it increments the *lost_events* count and exits. It does not wait around for available buffer. If buffer is available, it collects the necessary information for the event_record and writes it into the buffer. When a buffer is filled, it wakes up the process sleeping on the full pointer *fp* if necessary. It then calls *getstorage* to get another buffer ready for event data.

Appendix C

System Data Structures

This appendix gives the data structures of some *UNIX* system variables.

```
/*    The process structure
 *
 * One structure allocated per active
 * process. It contains all data needed
 * about the process while the
 * process may be swapped out.
 * Other per process data (user.h)
 * is swapped with the process.
 */
struct proc {
struct proc *p_link; /* linked list of running processes */
struct proc *p_rlink;
struct pte *p_addr; /* u-area kernel map address */
char p_usrpri; /* user-priority based on p_cpu and p_nice */
char p_pri; /* priority, negative is high */
char p_cpu; /* cpu usage for scheduling */
char p_stat;
char p_time; /* resident time for scheduling */
char p_nice; /* nice for cpu usage */
char p_slptime; /* time since last block */
char p_cursig;
int p_sig; /* signals pending to this process */
int p_sigmask; /* current signal mask */
```

```
int p_sigignore; /* signals being ignored */
int p_sigcatch; /* signals being caught by user */
int p_flag;
short p_uid; /* user id, used to direct tty signals */
short p_pgrp; /* name of process group leader */
short p_pid; /* unique process id */
short p_ppid; /* process id of parent */
u_short p_xstat; /* Exit status for wait */
struct rusage *p_ru; /* mbuf holding exit information */
short p_poip; /* page outs in progress */
short p_szpt; /* copy of page table size */
size_t p_tsize; /* size of text (clicks) */
size_t p_dsize; /* size of data space (clicks) */
size_t p_ssize; /* copy of stack size (clicks) */
size_t  p_rssize;  /* current resident set size in clicks */
size_t p_maxrss; /* copy of u.u_limit[MAXRSS] */
size_t p_swrss; /* resident set size before last swap */
swblk_t p_swaddr; /* disk address of u area when swapped */
caddr_t p_wchan; /* event process is awaiting */
struct text *p_textp; /* pointer to text structure */
struct pte *p_p0br; /* page table base P0BR */
struct proc *p_xlink; /* linked list of procs sharing same text */
short p_cpticks; /* ticks of cpu time */
long p_pctcpu; /* %cpu for this process during p_time */
short p_ndx; /* proc index for memall (because of vfork) */
short p_idhash; /* hashed based on p_pid for kill+exit+... */
struct proc *p_pptr; /* pointer to process structure of parent */
struct itimerval p_realtimer;
struct quota *p_quota; /* quotas for this process */
#ifdef sun
struct context *p_ctx; /* pointer to current context */
#endif
};

/*
 *      The User Structure
 *
 */
```

```
struct user {
struct pcb u_pcb;
struct proc *u_procp; /* pointer to proc structure */
int *u_ar0; /* address of users saved R0 */
char u_comm[MAXCOMLEN + 1];

/* syscall parameters, results and catches */
int u_arg[8]; /* arguments to current system call */
int *u_ap; /* pointer to arglist */
label_t u_qsave; /* for non-local gotos on interrupts */
char u_error; /* return error code */
union { /* syscall return values */
struct {
int R_val1;
int R_val2;
} u_rv;
#define r_val1 u_rv.R_val1
#define r_val2 u_rv.R_val2
off_t r_off;
time_t r_time;
} u_r;
char u_eosys; /* special action on end of syscall */

/* 1.1 - processes and protection */
short u_uid; /* effective user id */
short u_gid; /* effective group id */
int u_groups[NGROUPS]; /* groups, 0 terminated */
short u_ruid; /* real user id */
short u_rgid; /* real group id */

/* 1.2 - memory management */
size_t u_tsize; /* text size (clicks) */
size_t u_dsize; /* data size (clicks) */
size_t u_ssize; /* stack size (clicks) */
struct dmap u_dmap; /* disk map for data segment */
struct dmap u_smap; /* disk map for stack segment */
struct dmap u_cdmap, u_csmap; /* shadows of u_dmap, u_smap, for
   use of parent during fork */
label_t u_ssave; /* label variable for swapping */
```

```
size_t u_odsize, u_ossize; /* for (clumsy) expansion swaps */
time_t u_outime; /* user time at last sample */

/* 1.3 - signal management */
int (*u_signal[NSIG])(); /* disposition of signals */
int u_sigmask[NSIG]; /* signals to be blocked */
int u_sigonstack; /* signals to take on sigstack */
int u_oldmask; /* saved mask from before sigpause */
int u_code; /* ''code'' to trap */
struct sigstack u_sigstack; /* sp & on stack state variable */
#define u_onstack u_sigstack.ss_onstack
#define u_sigsp u_sigstack.ss_sp

/* 1.4 - descriptor management */
struct file *u_ofile[NOFILE]; /* file structures for open files */
char u_pofile[NOFILE]; /* per-process flags of open files */
#define UF_EXCLOSE  0x1 /* auto-close on exec */
#define UF_MAPPED  0x2 /* mapped from device */
struct inode *u_cdir; /* current directory */
struct inode *u_rdir; /* root directory of current process */
struct tty *u_ttyp; /* controlling tty pointer */
dev_t u_ttyd; /* controlling tty dev */
short u_cmask; /* mask for file creation */

/* 1.5 - timing and statistics */
struct rusage u_ru; /* stats for this proc */
struct rusage u_cru; /* sum of stats for reaped children */
struct itimerval u_timer[3];
int u_XXX[3];
time_t u_start;
short u_acflag;

/* 1.6 - resource controls */
struct rlimit u_rlimit[RLIM_NLIMITS];
struct quota *u_quota; /* user's quota structure */
int u_qflags; /* per process quota flags */

/* BEGIN TRASH */
char u_segflg; /* 0:user D; 1:system; 2:user I */
```

```
caddr_t u_base; /* base address for IO */
unsigned int u_count; /* bytes remaining for IO */
off_t u_offset; /* offset in file for IO */
union {
   struct { /* header of executable file */
int Ux_mag; /* magic number */
unsigned Ux_tsize; /* text size */
unsigned Ux_dsize; /* data size */
unsigned Ux_bsize; /* bss size */
unsigned Ux_ssize; /* symbol table size */
unsigned Ux_entloc; /* entry location */
unsigned Ux_unused;
unsigned Ux_relflg;
   } Ux_A;
   char ux_shell[SHSIZE]; /* #! and name of interpreter */
} u_exdata;
#define ux_mag Ux_A.Ux_mag
#define ux_tsize Ux_A.Ux_tsize
#define ux_dsize Ux_A.Ux_dsize
#define ux_bsize Ux_A.Ux_bsize
#define ux_ssize Ux_A.Ux_ssize
#define ux_entloc Ux_A.Ux_entloc
#define ux_unused Ux_A.Ux_unused
#define ux_relflg Ux_A.Ux_relflg
caddr_t u_dirp; /* pathname pointer */
struct direct u_dent; /* current directory entry */
struct inode *u_pdir; /* inode of parent directory of dirp */
/* END TRASH */
struct uprof { /* profile arguments */
short *pr_base; /* buffer base */
unsigned pr_size; /* buffer size */
unsigned pr_off; /* pc offset */
unsigned pr_scale; /* pc scaling */
} u_prof;
struct nameicache { /* last successful directory search */
int nc_prevoffset; /* offset at which last entry found */
ino_t nc_inumber; /* inum of cached directory */
dev_t nc_dev; /* dev of cached directory */
time_t nc_time; /* time stamp for cache entry */
```

```
} u_ncache;
#ifdef sun
int u_lofault; /* catch faults in locore.s */
int u_memropc[12]; /* state of ropc */
struct skyctx {
unsigned usc_regs[8]; /* the Sky registers */
short  usc_cmd; /* current command */
short  usc_used; /* user is using Sky */
} u_skyctx;
struct  hole { /* a data space hole (no swap space) */
int uh_first; /* first data page in hole */
int uh_last; /* last data page in hole */
} u_hole;
#endif
int u_stack[1];
};

/*
 *    The UNIX buffer structure
 */

struct buf
{
long b_flags; /* too much goes here to describe */
struct buf *b_forw, *b_back; /* hash chain (2 way street) */
struct buf *av_forw, *av_back; /* position on free list if not BUSY */
#define b_actf av_forw /* alternate names for driver queue */
#define b_actl av_back /*    head - isn't history wonderful */
long b_bcount; /* transfer count */
long b_bufsize; /* size of allocated buffer */
#define b_active b_bcount /* driver queue head: drive active */
short b_error; /* returned after I/O */
dev_t b_dev; /* major+minor device name */
union {
    caddr_t b_addr; /* low order core address */
    int *b_words; /* words for clearing */
    struct fs *b_fs; /* superblocks */
    struct csum *b_cs; /* superblock summary information */
    struct cg *b_cg; /* cylinder group block */
```

```
    struct dinode *b_dino; /* ilist */
    daddr_t *b_daddr; /* indirect block */
} b_un;
daddr_t b_blkno; /* block # on device */
long b_resid; /* words not transferred after error */
#define b_errcnt b_resid /* while i/o in progress: # retries */
struct  proc *b_proc; /* proc doing physical or swap I/O */
int (*b_iodone)(); /* function called by iodone */
int b_pfcent; /* center page when swapping cluster */
#ifdef sun
caddr_t b_saddr; /* saved address */
short b_kmx; /* saved kernelmap index */
short b_npte; /* number of pte's mapped */
#endif
};
```