Order Number 9020804

Improving the UNIX system V/386 process management

Sánchez, Felix E., Ph.D.

City University of New York, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

IMPROVING THE UNIX SYSTEM V/386 PROCESS MANAGEMENT

by

FELIX E. SANCHEZ

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

1990

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy

Dec 6, 1989
Date

T C Wesselkamper
Professor Thomas C Wesselkamper
Chair of Examining Committee

Dec 6, 1989
Date

T C Wesselkamper
Professor Thomas C Wesselkamper
Executive Officer

Professor Daniel I A Cohen
Professor Philip S DiPiazza
Professor Stewart Weiss

Supervisory Committee

The City University of New York

ABSTRACT

IMPROVING THE UNIX SYSTEM V/386 PROCESS MANAGEMENT

by

Felix E. Sańchez

Adviser: Professor Thomas C. Wesselkamper

The AT&T UNIX* system Operating System compares favorably with most of the present generation of operating systems. This research identifies areas of performance improvement by modifying the algorithms that manipulate *process management*. This research documents the performance improvements and shows how the improvements were developed.

The purpose of this research is to determine: 1.) whether significant improvements can be achieved in the performance of the UNIX system by the modification of the UNIX system kernel; and 2.) how significant such improvements may prove to be.

* UNIX is a Trademark of AT&T Bell Laboratories.

PREFACE

It is easier to optimize working code than to get optimized code working. -
Unknown

The original design of the UNIX system was an elegant piece of work done in the research area, and that design has proven useful in many applications. The UNIX system history began with Ken Thompson's work on a cast-off PDP-7 minicomputer in 1969. He and others who joined him had an overriding objective: to create a computing environment in which they themselves could comfortably and effectively pursue their own work - programming research. The result is an operating system of unusual simplicity, generality, and, above all, intelligibility[1].

One of the early users was Dennis M. Ritchie, who helped move the UNIX system to the PDP-11 in 1970. Dennis M. Ritchie also designed and wrote a compiler for the **C Programming Language**. In 1973, D. Ritchie and K. Thompson rewrote the UNIX kernel in the C programming language. With that rewrite, the system became essentially what it is today.

The most visible UNIX system interface is the **shell**, or command language interpreter, through which other programs are called into execution singly or in combination. However new interfaces to the UNIX system are being developed or ported.

A distinctive software style has grown upon this base. UNIX system software works smoothly together; elaborate computing tasks are typically composed from

loosely coupled small parts. Today, the UNIX system is a well accepted tool in research, government, and industry.

The purpose of this research was to determine if improvements could be made to the differents algorithms that control the UNIX system process management (process creation, process control and process termination) and how significant those improvements might be. The conclusions obtained as result of this research are that the UNIX kernel algorithms that control the process management can be improved significantly. Furthermore, the modifications may be applicable to different hardware architectures though more research will be necessary to make the suggested modifications part of an UNIX system standard release. As my understanding of the UNIX system kernel grows, so has my desire to explore and implement new algorithms, and ideas, and my interest in implementing them. The author acknowledges that the original UNIX system process management scheme has proven both convenient and efficient. However there is always room for improvement. Some of the algorithm changes implemented during this research took a long time to develop and on more than one occasion proved to be inadequate. What is or is not implemented in the kernel, or what can or can not be modified in the kernel represents both a great responsibility and a great power. Ken Thompson wrote: It is a soap-box platform on "the way things should be done." However, every modification was weighed carefully in order to meet the goals of this research.

I hope that two purposes are served by this research effort. On the one hand this research paper reveals my interest in operating systems and their effects on

computer performance. Second, this research paper describes a carefully conducted research effort, which encompassed the analysis of an existing system, the evaluation of the effects of carefully selected but innovative modifications, and the measurement of performance in several dimensions. It should be noted in this context, that the quality of this particular evaluation was enhanced by statistical data that was collected by tools which exist in the UNIX system.

The bibliography contains a number of references to relevant papers, journals, and books. I have attempted to include all the original sources of information used in conducting this research, including the pages numbers. Generally I have tried to include enough information so that the motivated reader will have a good starting point for future research.

I am bound to acknowledge my debt to several people who made this report possible. The contributors are, in the traditional but here especially apposite phrase, too numerous to mention. Certainly, collective salutes are due to my colleagues at AT&T Bell Laboratories Summit Facilities (*Home of UNIX*) for their valuable suggestions in particular to Dennis J. Metzger and Richard E. Menninger for their valuable suggestions, ideas, and criticism; to my colleagues at AT&T Network Services (4 Wood Hollow Road Parsippany) for their generosity in making facilities available for what turned into an ambitious project, for being brave users of this research version of the UNIX system and adopting it in is full form, and for the error reporting and machine time to load the new research versions; to my colleagues at AT&T Bell Laboratories Middletown for hardware support; to Professor Philip S. DiPiazza for his valuable suggestions, editorial comments, and

support; to William R. Wetzel and John P. Hickey for their help in *troff* and in generating figures. I would also like to thank: Brian W. Kernighan and the designers of *nroff/troff*, UNIX text processing tool, which made the typing and retyping a pleasure instead of a chore; Michael T. Clitherow who gave me the opportunity to work in the UNIX system internals; to Professor Daniel I. A. Cohen, Professor Philip S. DiPiazza and Professor Stewart Weiss, members of the dissertation committee, for their dedication, suggestions, and for being available on such short; notice and to my computers *israel* and *moses* for the warm summer evenings that we spent together.

Last but not least, I would like to thank Professor Thomas C. Wesselkamper for his patience and dedication in reading this research paper. I particularly appreciate his inventiveness, thoughtful criticism, and constant support. To my son, Joshua, thank you for being so pleasant and helpful while I was writing and collecting data, for keeping my computer free of gummy bears, and for not standing on my keyboard.

Felix E. Sańchez
Monmouth Junction, New Jersey
October 1989

This research effort is dedicated to the

loving memory of *Isadore Myerowitz*.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

The UNIX operating system has become quite popular since its inception in 1969. It runs on machines of varying processing power from microprocessors to mainframes, and provides a common execution environment across them. The system is divided into two parts. The first part consists of programs and services that have made the UNIX system environment so popular; it is the part readily apparent to users, and includes such programs as the shell, mail, text processing programs and source code control systems. The second part consists of the operating system that supports these programs and services[2].

During the past few years, the UNIX operating system has come into wide use, so wide that its name has become a trademark of AT&T Bell Laboratories. Its important characteristics have become known to many people. It has suffered much rewriting and tinkering since the first publication describing it in 1974[3], but there have been few fundamental changes[4].

1.1 UNIX System Overview

The UNIX system oversees the execution of many user programs (or commands). These programs seem to execute simultaneously because of the system's ability to time-share the processor among all the programs. Actually each program is scheduled (at the appropriate time) to use the processor for a short period of time to the exclusion of all other programs. The UNIX system is a multiuser and multiprogramming system.

The UNIX system includes:

- The UNIX Operating System kernel.
- The shell command interpreter.
- The file system.
- Various user and system commands.

1.1.1 The Kernel

The kernel (comprising from 5 to 10 percent of the operating system software) is the basic resident software on which the entire system relies. It is the only permanently resident part of the system. The job of the operating system kernel is to control user processes and manage system resources.

1.1.2 The Shell The shell command interpreter allows the user to communicate with the UNIX system. The shell, besides providing the user interface to the kernel and interpreting operating system commands, also can be used as a programming language. Because shell procedures are easy to create and use, much of the drudgery associated with programming is eliminated.

1.1.3 The File System

The file system of the UNIX system consists of a set of directories and files arranged in a tree-like structure. The file system is build in a hierarchical way from the root (/) directory. This file system provides automatic file space allocation and deallocation; a complete set of flexible directory and file protection

modes; facilities for creating, accessing, moving, and processing files; directories or sets, mountable and unmountable file systems and volumes; etc. The UNIX system treats each physical input/output device from interactive terminals to main memory like a file, allowing uniform file and device input and output.

1.2 Terminology

Throughout this research paper, the term *priority* refers to the software priority assigned to a process, not the hardware priority of the processor. It should be remembered that in the UNIX system implementation, *switching* refers **only** to the process of selecting a CPU process that is *already in memory* and waiting to use the CPU. *Scheduling* refers **only** to the process of selecting processes to be brought into or removed from memory.

In some sections of this research paper it is necessary to refer to the structure of a user process, i.e., text, data, stack, etc. A brief summary of a process's important details is given in the section on Process Setup.

Italic letters are used to represent UNIX system components and function calls. Capital letters are used to represent UNIX system kernel process names.

1.3 Miscellaneous

State diagrams and examples are used throughout the discussions as an aid to visualization. Four appendices are included. Appendix I summarizes the priorities that are assigned by the system to a process to regulate its share of the processor's

time. Appendix II contains some of the compiled data that illustrates the profiling applied in this research as an aid to implementing modifications to the UNIX system kernel. Appendix III contains a brief description of the programs utilized during this research to measure kernel performance. Appendix IV contains state diagrams, and performance charts.

2. RESEARCH IMPLEMENTATION

The first steps in undertaking this research were to identify the major components of the UNIX System V/386 Release 3.2 *Process Management* system, determine their influence upon the operating environment, and measure the performance of each component. Detailed examination of the code shows where the UNIX system *Process Management* spents most of its time. Performance improvements were made where necessary to meet the goals of this research. However, the modifications to the kernel were limited to certain functions of *Process Control* and *Process Management*, without affecting the basic structure of the UNIX system. (Given the monolithic structure of the UNIX system, some modifications may spread problems through the entire UNIX system kernel).

The Reduced Instruction Set Computers (RISC) philosophy - performance improvement by optimizing the instruction set, data structures and algorithms, for the most common and typically simpliest case(s) - was utilized. This involves identifying the most common case, the case in which most of the time is spent. We apply this philosophy to the UNIX system kernel performance. The UNIX system kernel data structures and algorithms are optimized for the most common cases while still supporting the general case.

2.1 UNIX System Kernel Profiling

The UNIX system operating system contains a number of counters that are incremented as various systems actions occur. The system activity package reports UNIX system system-wide measurements[5] including central processing unit (CPU)

utilization, disk and tape input/output (I/O) activities, terminal device activity, buffer usage, system calls, system switching and swapping, file-access activity, queue activity, and message and semaphore activities. The system activity information reported by this package is derived from the set of systems counters described below. (See Appendix II for examples of collected data during this research).

2.1.1 System Activity Counters

The UNIX system operating system manages a number of counters that record various activities and provide the basis for the system activity reporting. The data structure containing most of these counters is defined in the *sysinfo* structure which resides in */usr/include/sys/sysinfo.h*. The system table overflow counters are kept in the *syserr* structure. The device activity counters (see reference [6] for more information on instruction counting) are extracted from the device status tables.

The following paragraphs describe the system activity counters sampled and utilized throughout this research.

CPU time counters - There are four time counters that may be incremented at each clock interrupt. According to the state the CPU is in at the interrupt (idle, user, kernel, or wait for I/O completion), exactly one of the *cpu[]* counters is incremented.

Lread and lwrite - The *lread* and *lwrite* counters are used to count logical read and write requests issued by the system to block devices.

Bread and bwrite - The *bread* and *bwrite* counters are used to count the number of times data is transferred between the system buffers and the block devices. These physical I/Os are triggered by logical I/Os that cannot be satisfied by the current contents of the buffers. The ratio of block I/O to logical I/O is a common measure of the effectiveness of the system buffering.

Swapin and swapout - The *swapin* and *swapout* counters are incremented for each system request that initiates a transfer from or to the swap device (*/dev/swap*). More than one request is usually involved in bringing a process into or out of memory because text and data are handle separately. Frequently user programs are kept on the swap device and are swapped in rather than loaded from the file system. The *swapin* counter reflects these initial loading operations as well as resumptions of activity; the *swapout* counter reveals the level of actual *swapping*. The amount of data transferred between the swap device and memory are measured in blocks and counted by *bswapin* and *bswapout*.

Pswitch and syscall - These counters are related to the management of multiprogramming. *Syscall* is incremented every time a system call is invoked. The number of invocations of *read(2)*, *write(2)*, *fork(2)*, and *exec(2)* system calls are kept in counters *sysread*, *syswrite*, *sysfork*, *and sysexec* respectively. The *pswitch* counts the times the switcher is invoked. This occurs when:

1. A system call results in a road block.
2. An interrupt occurs resulting in awakening a higher priority process.
3. A one second clock interrupt occurs.

Iget, namei, and dirblk - These counters apply to file-access operations. *Iget* and *namei*, in particular, are the names of the UNIX system routines. The counters record the number of times the respective routines are called. *Namei* is the routine that performs file system path searches. *Iget* is a routine called to locate the inode entry of a file (i-number).

Dirblk - This counter records the number of directory block reads issued by the system.

Runque, runocc, swpque, and swpocc - These counters are used to record queue activities. They are implemented in the *clock.c* routine. At every one second interval, the clock routine examines the process table to see whether any processes are in core and in the ready state. If so, the counter *runocc* is incremented and the number of such processes is added to counter *runque*. While examining the process table, the clock routine also checks whether any process in the swap device is in the *ready* state. The counter *swpocc* is incremented if the swap queue is occupied, and the number of processes in the swap queue is added to counter *swpque*.

Readch and writech - The *readch* and *writech* counters record the total number of bytes (characters) transferred by the *read(2)* and *write(2)* system call, respectively.

2.1.2 Monitoring I/O Activities

Four counters are kept for each disk or tape drive in the device status table. The counter *io_ops* is incremented when an I/O operation occurs on the device. It includes block I/O, swap I/O, and physical I/O. *Io_bcnt* counts the amount of data transferred between the device and memory in 512 byte units. *Io_act* and *io_resp*

measure the active time of a device in time ticks summed over all I/O requests that have completed for each device. The device active time includes the device seek, rotate, and data transfer times, while the response time of an I/O operation is measured from the time the I/O request is queued to the device to the time when the I/O completes.

Inodeovf, fileovf, textovf and procovf - These counters are extracted from the *syserr* structure. When an overflow occurs in any of the inode, file, text, or process tables, the corresponding overflow counter is incremented.

2.1.3 Monitoring Operating System Activities

In order to facilitate an activity study of the UNIX system, operating system profiling system programs were utilized, including an interface to profile data and text addresses contained in the */dev/prf* device.

Prfld - This system program is used to initialize the recording mechanism in the system. It generates a table containing the starting address of each system subroutine as extracted from a *namelist* (*/unix* is default for namelist file).

Prfstat - This system program is used to enable or disable the sampling mechanism. Profiler overhead is less than 1% as calculated for 500 text addresses. *Prfstat* also reveals the number of text addresses being measured.

Prfdc and prfsnap - These system programs are used to perform the data collection function of the profiler by copying the current value of all the text address counters to a file where the data is analyzed.

Prfpr - This system program is utilized to format the data collected by *prfdc* or *prfsnap*. This system call converts each text address to the nearest address of a text symbol of the namelist (*/unix*).

The sections under study were compiled with the *-p* option (*cc -p*) which automatically includes calls for profiling with default parameters; *monitor(3c)* was utilized on critical regions where fine control over profiling was required. A program compiled with the above specified option records a histogram of periodically sampled values of the program counter, and of counts of certain functions, in the buffer. The *prof(1)* and *profil(2)* were utilized to interpret the profile file produced by the *monitor(3c)* function. The *prof(1)* function prints, for each external text symbol, the percentage of time spent executing the code between the address of that symbol and the address of the next text symbol, together with the number of times that the function is called and the average number of milliseconds per call. The UNIX system internal activity is measured by a number of counters contained in the system kernel. Each time an operation is performed, an associated counter is incremented. The *sar(1M)* and *sar(1)* system activity report packages were utilized to monitor the values of these counters.

2.1.4 Profiling Formulae

The formulae shown below for two distinct sampled times *t2* and *t1*, were utilized as an aid to the study and measurement of performance of specific sections of the kernel. The *sar(1)* and *sar(1M)* utilize these formulae in a more generic manner.

CPU utilization -

$$\%\_of\_cpu\_x = \frac{cpu\_x}{(cpu\_idle + cpu\_user + cpu\_kernel + cpu\_wait)} * 100$$

where cpu_x is cpu_idle, cpu_user, cpu_kernel (cpu_sys), or cpu_wait.

Cache hit ratio -

$$\%\_of\_cache\_I/O = \frac{(logical\_I/O - block\_I/O)}{logical\_I/O} * 100$$

where cache I/O is cache read or cache write.

Disk or tape I/O activity

$$\%\_of\_busy = \frac{I/O\_active}{(t2 - t1)} * 100;$$

$$avg\_queue\_length = \frac{I/O\_resp}{I/O\_active};$$

$$avg\_wait = \frac{(I/O\_resp - I/O\_active)}{I/O\_ops};$$

$$avg\_service\_time = \frac{I/O\_active}{I/O\_ops}$$

Queue Activity

$$avg\_x\_queue\_length = \frac{x\_queue}{x\_queue\_occupied\_time};$$

$$\%\_of\_x\_queue\_occupied\_time = \frac{x\_queue\_occupied\_time}{(t2 - t1)}$$

where x_queue is run queue or swap queue.

System activities

$$avg\_rate\_of\_x = \frac{x}{(t2 - t1)}$$

where x is swap in/out, blks swapped in/out, terminal device activities, read/write characters, block read/write, logical read/write, process switches, system calls, read/write, fork/exec, iget, namei, directory blocks, disk/tape I/O activities, message, or semaphore activities.

2.1.5 Additional Profiling Tools

In some cases it was necessary to instrument debugging statements in the kernel in order to gather UNIX system kernel timing information, or trace UNIX system kernel activities. Basically *printf* statements to the */dev/console* were implemented to determine the time it takes for a section of the UNIX system kernel code to execute. This was implemented by generating records at the start and end points of the code under study. The technique utilized is described in *Tracing and Timing Kernel Activities with CASPER*[7].

In summary, the steps involved in the profiling process were:

- identify the major sections to be studied,
- monitor the performance of each of the sections,
- choose the dominant section,
- analyze the performance of each component of the section,
- determine the impact of each component on the kernel,
- select the modification to be performed,

- implement the modification,
- build a new UNIX system kernel,
- install and test the new UNIX system kernel,
- go back to the first step.

In this research process, the absolutely immutable condition was to improve the algorithms used for the implementation of certain functions, without affecting the behavior of the UNIX system kernel. This condition was also applied to the profiling process in order to be able to observe the algorithms with a minimum possible influence upon real performance.

3. PROCESS MANAGEMENT OVERVIEW

This section provides an overview of *scheduling*, *switching*, *forking* and *exec* under the UNIX system and discusses some of the basic timing (acquired during this research) within the operating system. A description of the algorithms used by the *scheduler*, the process *switcher*, and *process creation* is provided; Some of the implementation details are also presented as well as the impact of the suggested modifications.

Topics such as process states, process priorities and process age which are required for a discussion of *switching* and *scheduling* are introduced first. The *switching* and *scheduling* algorithms are then discussed from the standpoint of what system's policy is invoked. Storage allocation is then discussed since it is an important factor in scheduling. This is followed by a brief discussion of the *Traffic Controller*.

3.1 The UNIX System Scheduler And Switcher

The UNIX System V/386 Release 3.1 and 3.2 operating system as implemented on the AT&T 6386 WGS (Intel 80386 microprocessor based) are a timesharing systems[8] that possess a distributed supervisor[9]. The selection of processes to use the CPU (process switcher *swtch()*), for memory use (the scheduler *sched()*) and the process creation (*fork()*) are separate functions within the UNIX system operating system[10].

This dissertation discusses the algorithms used by the *scheduler* and *switcher* and many details of their implementation as well as the impact on both as result of the

modifications implemented in the UNIX system kernel. This document can be used as an aid to understanding the UNIX system as well as the proposed changes, and includes some of the implications of the modifications obtained via mathematical analysis.

The UNIX System V operating system uses a multiple queue feedback system for ordering processes for CPU usage [11]. The concept of process priority as it relates to queue setup and the method by which the processor is relinquished is explained. Reentrancy within the operating system, together with its implications for interrupt processing and preemption, is also discussed. Penalty schemes used for limiting the amount of time a process may use the CPU are discussed for the current Release 3.2 as a guide evaluating the modifications implemented in the UNIX system process creation.

3.2 Process Attributes

This section discuss some of the attributes of processes that are important quantities used by the system to manage the use of the processor and memory.

3.2.1 Process States

For the present discussion, the states in the lifetime of a process are summarized (a pictorial representation is given in Figure 1). A process can be in any one of four states. Processes in the system that are ready to execute are placed in the *ready* state. All new processes are created by the *fork* mechanism as *ready* processes (in memory, if enough memory is available, or on the swap device, if not enough

memory is available to the *fork()* process creation). Only one *ready* state process is executed by the operating system at a time. However, there is no special state for the currently executing process. During its lifetime, a process may make a transition to or from two other states: *sleeping* and *waiting*. These two states are for processes that are roadblocked, that is, for processes that make a request (via a system call) for some system service or resource (e.g., a device, memory, a system table) which is not available. The reason for the use of two different states is that a process may require a resource which the system is certain will be available in a short time (e.g., a disk) or it may require a resource which might not be available for a long time (e.g., the status of a dead child process). (These two states bear no relation to the *sleep* and *wait* system calls available as commands and system calls under the UNIX system.) When a process enters either of these states, it willingly relinquishes the processor and posts an *event* (an identifier) which can be used to synchronize the awakening (placing the process in the *ready* state) of the process. Processes go into the *sleep* or *wait* state waiting for *one* and *only one* event. Processes that are roadblocked because of the unavailability of a resource that should become available after only a short delay enter the *sleep* state, while those processes that are roadblocked because of the unavailability of a resource that will not be available soon enter the *wait* state. In both cases, the transition to either of these states is made as the result of the process willingly relinquishing the processor. There are some differences in the treatment of processes in the two states as described in later sections. The different treatment results from the fact that for efficient memory utilization required in this research implementation, it makes sense to remove, from memory processes that will not be ready to run for a

long time (several seconds) when memory is needed. Besides posting an *event* to synchronize the awakening (return to *ready* state) of a process, a process entering the *sleep* or *wait* state is given a software priority so that when the resource that the process needs is available, the process receives that priority when it is returned to the *ready* state.

Note that in Figure 1, there is a means, besides being awakened, by which a *waiting* process may enter the *ready* state. Any signal (see next section) sent to a process that is in the *wait* state causes it to be made *ready*. The process is made *ready* even if the resource it was waiting for is not available. (*Signals* are means of interprocess communication for the UNIX system and are used to communicate a single bit of information to another process. For example, one *signal* indicates to a process that it should terminate itself.) This is done to insure that *waiting* processes respond to signals, since processes in the *wait* state are typically waiting for events that may not occur shortly or which may never occur (e.g., the death of a child process).

A process may be awakened from the *wait* or *sleep* state by a *wakeup* intended for another process sharing the same resource. Since this is possible, it is necessary that on returning to the *ready* state from either the *wait* or *sleep* state, the system check to see that the resource the process needs is actually available. If the resource is not available, then the process must relinquish the processor and go again into the *sleep* or *wait* state.

The last state of a process is the *zombie* state. In this state, a process has become defunct as the result of doing a normal exit, receiving a *signal* which the process is

not prepared to handle, or being killed (which is a special *signal*) by another process. (The operation of becoming a *zombie* is discussed from the storage allocation standpoint in a later section, since that is the only part affected by this research). The *zombie* lingers in the system until its direct parent or process one (the *init* process) receives status information from the *zombie* and finally discards it.

3.2.2 Process Age

Associated with each process in the system is a process *age*. This *age* is the chief criterion used by the *scheduler* in determining **which** processes are to be brought into or removed from memory and **when** action is to be taken to rearrange memory. This *age* represents the amount of time that a process has spent in memory or, if the process is not in memory, the amount of time the process has spent on the swap area. Process *age* is kept in seconds and is only a crude measure of the true length of time that a process has spent in memory or in the swap device.

The clock interrupt handler which supplies all software timing to the operating system performs the aging process **once a second** by simply increasing the age of **all** processes in the user node by one second, in order to prevent a process from monopolizing use of the CPU. When a process arrives in memory or arrives on the swap area, its age is reset to zero. This method, while being economical in terms of the number of updates to the process *age*, can give a false impression of the length of time that a process has been in memory or on the swap area. Figure 2 illustrates the inequitable aging that can occur due to a late or early arrival of a process (in

memory or on the swap area). This overaging or underaging is important in discussing *scheduling* criterion and in choosing criterion to prevent thrashing.

The aging of a process continues until a maximum age of 127 seconds is reached. The *scheduler* is always locked in memory and therefore reaches and retains this age.

3.2.3 Priorities

In order to share the use of the processor among the processes in memory, the system needs a policy for processor use and a scheme for implementing that policy. One simple policy for a timesharing system is to attempt to give each process in the system the same percentage of CPU time. That is, if there were N processes in the system, the system should attempt to give each process 1/Nth of the processor's time. The *Scheduler Algorithm* on the UNIX system belongs to the general class of operating system schedulers known as *round robin* with *multilevel feedback*, meaning that the kernel allocates the CPU to a process for a time quantum, preempts a process that exceeds its time quantum, and feeds it back into one of several priority queues. This essentially is the overall policy that the UNIX system currently follows.

Because of the different characteristics of processes, the fact that processes are constantly entering and leaving the system and the desire to minimize the amount of system overhead in enforcing the sharing of the processor, implementing this policy or, for that matter, any policy is difficult. Selecting processes on a round robin basis has the obvious disadvantage that even if each process is limited to one

time quantum before the next process is run, I/O bound processes relinquish the processor before completing their time quantum and are at a disadvantage if there are any CPU bound processes in the system. A strict round robin algorithm does not attempt to determine what type of activity processes are engaged in and modify its treatment of processes based on that determination (so that it would only implement a round robin policy if all processes in the system had exactly the same characteristics).

In an effort to determine what the execution characteristics of a process are and thereby to be able to modify the treatment of processes appropriately, the UNIX system uses a priority scheme. This scheme attempts to reward processes that willingly share the processor (I/O bound) and to penalize processes that do not willingly share the processor (CPU bound) so that each process receives an equitable share of processor time.

The priority assigned to a process is intended to signify the importance that the system attaches to giving the CPU to a process. For example, a process which is awakened after doing I/O to a disk is given preference over a process that is awakened because a delay timer has elapsed. This priority is assigned to a process when it is roadblocked so that when it is awakened (i.e., made *ready*) it has the specified priority. Appendix I compiles a list of the values that are assigned to processes and the activity or (unavailable) resource which results in the process being roadblocked at that priority.

There is a narrow range of priority values that can be assigned to processes for the purposes of ordering them by preference for process *switching*. The scale of values

used by the system is shown in Figure 3. Under this scheme, the lower the value of the priority, the higher the priority that the process has with respect to allocating the processor to a process. The priorities range from a maximum value of 0 (PSWP) to a minimum of 127 (PIDLE), as specified in the data structure *param.h* located in */usr/include/sys/param.h*.

Priorities assigned to a process are also coupled to the *state* of a process. Processes that relinquish the CPU are placed in the *sleep* state (PSLEP). These priorities are generally associated with resources that have (relatively) short access time, so that the process can be made *ready* to run after a short delay. (Processes go to *sleep* because they are awaiting the occurrence of some event, such as waiting for system resources, waiting for I/O completion from a peripheral device, and so on. Sleeping processes do not consume CPU resources).

A process can synchronize its execution with the termination of a child process by executing the *wait(2)* system call, placing itself in the *wait* state (PWAIT). (The kernel does not contain an explicit wake up call for a process in the *wait* state, such processes only wake up on receipt of signals). These priorities are associated with resources that may not be readily available (e.g., waiting for a child process to die, waiting for the time of day to reach a specified time before continuing execution, etc.). The final category of priorities are *user* priorities (PUSER). These are priorities that are assigned to a process while it is *ready* to execute, i.e., in *user space* (that is, a process ready to execute within the user's virtual address space). (Processes can be *ready*, yet executing within the operating system. Under these circumstances, we speak of the process as executing, i.e., in *system space*. See

Figure 4 for a pictorial representation of a *ready* process executing in two different address spaces). There is a range of *user* priorities because it is necessary at times to penalize processes which are using more than their fair share of processor time. The *sleeping* and *waiting* priorities are discrete in that a fixed value is assigned to a process entering the *sleep* or *wait* state. This priority remains fixed while the process is in that state and stays with the process when it becomes *ready*. It does not change until the process enters the *wait* or *sleep* state again or until it begins executing in *user space*. Figure 4 illustrates the dynamics of priority changes. The three states shown within the area labeled *system space* represent the states that a process can be in while executing within the system's virtual address space. (When executing a system call, a process is executing in *system space*. Processes in *system space* enter the *sleep* or *wait* state at a particular priority and upon being awakened are returned to the *ready* state at that priority. While completing the system call, the process retains this priority unless of course it enters the *sleep* or *wait* state again, at which time the priority is changed to a new discrete value). When a process that is in the *ready* state is returned to executing in *user space*, it is assigned a user priority (PUSER) plus any penalty (or enhancement) that a user has given the process. Note that while a process is ready to execute in *user space* it can only be in the *ready* state and that while the process is in *ready* state the process retains its user priority, unless it is preempted and penalized by the system for not willingly sharing the processor (i.e., for using its full time quantum).

In contrast to the discrete values of priority for processes executing in *system space*, the *user* priority is dynamic in the sense that a user process has its priority adjusted periodically so that it is penalized for specific types of undesirable

activity. Activity is regarded as undesirable based on its effect on the response of other processes. The UNIX system distinguishes between two types of undesirable activity. *Compute bound* processes are processes that spend virtually all of their time in *user space* while *system bound* processes are those that spend most of their time in *system space*. In both cases, the processes are CPU bound and so never enter the *sleep* or *wait* state. The distinction is somewhat artificial and results from the algorithm used to penalize processes. The penalties applied in each case are discussed in section 4.7 as well as the new penalty scheme implemented as part of this research.

3.2.4 Priority Ranges

The range of priorities described above should not be confused with the priority range specified in the UNIX System Programmer Reference Manual for the *nice(2)* system call or *nice(1)* command (system call ranges 0 to 39, command ranges 1 to 19). The *nice* system call is a crude tool which is intended to force a user to receive degraded service, that is, run in the background. It forces the user's priority to be continually lower than normal. It can, under special conditions, allow a process to receive better service by raising the process priority. (However, it was found in this research that this can have such drastic effects on overall response that it is recommended that it not be used). The difference between the scale of values used by *nice* and that used by the system is the reference point. Within the operating system, zero priority divides processes in the *sleep* state (negative) from those in the *wait* state (positive) or *ready* state (see Figure 3 for a nomograph). Processes that are *ready* and are executing in *user space* usually have a priority between 61

and 126. From the *nice* system call (the user's) standpoint, zero priority divides the priorities used by the system (negative) from those of user processes (positive). Any user may lower his base priority (60) thus willingly penalizing himself, however, only the *super user* (or a process that has made arrangements to assume the permissions allowed the *super user*) can assign himself a priority above that assigned by the system. As an example, priority 100 from the systems standpoint corresponds to priority zero from the *nice* standpoint while priority 40 from the system's standpoint corresponds to -60 from the *nice* standpoint.

3.2.5 Priority Assignments

The tables in Appendix I list many of the discrete priorities and the events or resources with which they are associated with. Each table contains the priorities associated with a particular grouping; *sleep, wait* or *user*. The integer value of the priority, a mnemonic, used within the operating system source code to represent that priority, and a list of the conditions under which a process can receive that priority are given. The tables are useful when using the *ps* (process status - see *ps(1)* in the UNIX System Programmer Reference Manual) command to ascertain the status of processes in the system, since the priority of each process is printed, and by knowing which activities can cause a process to receive that priority, one can estimate in what activity the process is engaged.

Basically, high priority is assigned to resources with short access times which are critical resources or services in the system. For example, swapping a process into or out of memory has the highest priority (0) since memory is a relatively scarce and important resource from the system's standpoint and operations which make

that resource available are given preference. Since the *scheduler* manages the entrance of processes into memory, any events used to synchronize its operation also use the highest priority. Next highest on the scale of priority values is accessing any system table (I-node Table, I-list in the superblock, etc.) entry that is locked. This occurs most often while the *update* process is running. (The *update* process insures that there is correspondence between the i-node entries in memory and their corresponding entries on file systems.) These tables are locked when they are accessed to insure that the entry is not changed while it is being updated on the file system. I/O to block oriented devices (disks) follows in priority. Serial I/O is given a lower (*wait*) priority since the time between inputs from a terminal (*think time*) is long by comparison with access times for a block device. A process waiting for a child process to die or delaying its own execution has a still lower value of priority. The lowest priorities are assigned to processes that are *ready* to execute within their own virtual address space.

4. THE PROCESS SWITCHER

The process *switcher* is responsible for selecting the next process to use the CPU and thereby is responsible for multiprogramming the system. It is also responsible for implementing the system's policy for sharing the CPU which was outlined when process priorities were discussed. It selects only from among the *ready* processes that are in memory. The *scheduler* is responsible for assuring that there are *ready* processes in memory to be multi-programmed.

Use of the processor is based on a highest priority first scheme where the system itself assigns the priorities to processes based on the type of activity that a process is engaged in and the system's policy for maximizing the use of its resources (see Section 3.2.3).

In discussing process *switching*, there are two separate issues involved. The first issue is *how* a process is selected to use the CPU and the second issue is *when* the selection process is invoked.

4.1 The Switching Algorithm

Process switching is a function of two variables, the priorities of the processes in memory that are *ready* to execute and the identity of the currently executing process. If an attempt were made to represent this as a mathematical function. It would probably be written as,

$$S(p, S_{-t}, N);$$

Where the function S computes which process is next to run, the variable p is the priority of a process (see Section 3.2.3), S_{-1} is the identity of the currently executing process (that is, the result of the last selection) and N is the number of processes currently running. As mentioned earlier, the process *switcher* is constrained to select from processes that are *ready* to execute (i.e., whose state, which is represented by the variable s, indicates that they are in the *ready* state) and which are already *in memory* (i.e., whose location, represented by the variable f, indicates that they are in memory, i.e., loaded).

The process *Switching Algorithm* selects the highest priority process in memory as the next process to use the processor. In the event that there are several processes with the same highest priority, it selects processes in a round robin fashion until all of the processes at that priority have received service. To insure that equitable treatment is given to other processes with the same highest priority, the process *switcher* remembers the identity of the process chosen and chooses the next process with the same highest priority on a subsequent selection. If a higher priority process shows up between selection, that process, of course, receives precedence. This selection process can be expressed as follows,

- Choose the highest priority process from all processes $\alpha(p,s,f)$ that are *ready* (s = *ready*) and *in memory* (f = *loaded*).

In this brief encapsulation of the selection process, each process is considered to be represented by three variables, its priority p, its state s and its location f. The way that round robin service is currently guaranteed among processes at the same priority is that instead of examining the processes in the order in which they exist

in the Process Table, say 1 to N, the N processes are examined starting at the last process selected. This turns process *switching* into a round robin (RR) selection whenever more than one process with the same highest priority exists.

4.2 The Switcher Implementation

It cannot be overemphasized that the multiprogramming provided by the process *switcher* has nothing to do with rearrangement of processes in memory and on the swap area. The *switcher* merely deals with processes that are *ready* to execute and are *in memory*. Controlling the entrance and exit of processes from memory so that they are eligible to use the processor is the province of the *scheduler*.

Note that the *switcher* is not a process in the same sense that the *scheduler* is a process. Rather it is a function within the system used to allow or enforce multiprogramming. To acquire a better understanding of the selection process, the data base that the system uses to keep track of the attributes of processes is discussed.

4.3 The Process Table

Processes are known to the system by their existence in the Process Table. This table is a linear array of entries which contain information on process location, size, state and any other information which the system must always have available to manage that process. The table is not maintained in any predetermined order and new entries are placed in the first available location. A linear search of the array is

done to find a free slot so that allocated entries tend to compact in the lower end of the table[1]. The *switcher*, however, treats the Process Table as a ring (see Figure 5). The anchor for this ring is taken to be the position of the currently executing process so that the anchor position is constantly changing. When it begins its search for a process to run, the *switcher* starts with the process that is currently running (i.e., the process that is currently the anchor). Figure 6 shows an example in which the *shell* is executing when a higher priority process β with priority p_i is awakened. When the process switcher runs, it selects this as the new process to run.

The only bias exerted by the *switcher* in its selection of processes should be the priority of the processes in the ring. There should be no advantage which accrues to any position in the Process Table. This should in general be true; however, there are some processes in the system whose position in the Process Table is fixed: the *scheduler* is a process in the system and is always the first process in the Process Table and the *scheduler* is the only process not created by the *fork*. The *init* process is always the first process in the system and always occupies the second position in the Process Table. The *init* process spawns a series of copies of itself (children) to act as line monitors and wait for carrier detection on each serial I/O (*/dev/tty*) line interface. The child *init* processes are eventually overlaid by the

---

1 There are a large number of linear searches of this table done by various systems functions. The process creation and termination procedures in the system keep track of the last position occupied in the table. This takes advantage of the fact that the table is usually never full and that entries are compacted at the low end of the table because of the way new entries are allocated. Under the current scheme, all of these searches run in time that is proportional to the actual number of processes in the system.

command interpreter (i.e., the *shell*). It can be expected that these processes normally occupy and remain in the lower portion of the Process Table since the algorithm for selecting slots for new processes is on a first available basis. (Over an extended period of time on a heavily loaded system there is a tendency for these children to migrate to random positions in the Process Table). The fact that certain processes consistently occupy fixed positions in the ring over a long period of time would exert no bias on the process switching algorithm unless these processes consistently receive high priority. Unfortunately, the *scheduler* is such a process. When it is ready to execute, it always has priority 0[2] so that it is probably the next process to run. Figure 7 shows an example similar to that of Figure 6 except that the *scheduler* is the process that is awakened. Note that since it is such a high priority process, it is the next process to run and that if the system were giving round robin service to a group of processes with the same (highest) priority, the *scheduler* would disrupt this round and favor processes at the low end of the Process Table. A number of events cause the *scheduler* to be awakened as is discussed later, however, the basic reason for it being awakened are: a *kill* sent to any process, a *wakeup* of a swapped out process or the currently executing process going into the *wait* state. Figure 7 illustrates these conditions showing the three cases in which the *scheduler* would be awakened; α is killed, the currently executing process (β) goes into the *wait* state and a swapped out process (γ) is awakened. Any one of these occurrences puts the *scheduler* in the *ready* state and

2 The highest priority value on the system is 0 Thus, user level 0 has higher priority than user level 1, and so on

at the next opportunity, the process *switcher* runs the *scheduler*. If the *scheduler*, with its fixed position, were the only process to achieve the highest priority, it would disrupt the system's attempt to provide round robin service among processes with the same priority as it would provide a bias towards processes at the low end of the Process Table whenever it ran (i.e., processes residing at the high end of the Process Table would be passed over when the *scheduler* ran). Fortunately, the *scheduler* is not the only process that achieves the highest priority. A process may swap itself out if it requires more memory than is available and under these conditions that process receives the highest priority (0) when it completes the swap. This tends to discount the bias introduced by the *scheduler's* fixed position and priority. The frequency at which the *scheduler* runs (approximately once a second) in relation to that of the process *switcher* also tends to decrease its positional importance.

Figure 6 also illustrates how Round Robin service occurs when two processes have the same priority. Here, the search for a new process to execute is begun at the currently executing process and runs through the entire Process Table (viewed as a ring). Assuming two processes having the same highest priority $\alpha(p_i)$ and $\beta(p_i)$ in Figure 6, the first process encountered in the search is chosen. The fact that the search begins immediately after the last process to execute and continues circularly around the list insures that if these remain the only two eligible processes, one process is not preferred over the other. Figure 6 shows that $\beta(p_i)$ has been chosen and is executing. If α and β remain the two highest priority processes in the system, then α is chosen next to run and thereafter they execute round robin until one completes.

4.4 Microstructure Of Process Switching

In an earlier section, the algorithm used by the process switcher in selecting a process for use of the CPU is described. This shows *how* processes are selected based on the system's policy of giving high or low priority to processes engaged in different activities. The final component of the system's policy for sharing the processor is to understand when the selection process is used. To do this, it is necessary to discuss the application of the policy both at a gross level and at a fine level. The gross level relates to enforced sharing and the penalty schemes used to implement that sharing among CPU bound processes, while the fine level relates to multiprogramming processes willing to share the processor, e.g., I/O bound processes. The gross level examination is simply a review of penalty schemes, while the fine level of sharing is discussed in section 4.11.

4.5 Switching Policy

Before discussing penalty schemes, a summary of the times when the process *switcher* is invoked is given.

1. The process *switcher* is invoked as part of the gross penalty scheme for enforcing processor sharing among processes.
2. It is invoked when processes willingly share the processor because they must wait for the occurrence of some event.
3. It is invoked when some process of higher priority is awakened.

4. It is invoked when a process needs more memory to continue execution and none is available.

5. It is invoked when a process terminates.

The first three cases are discussed in succeeding sections while the fourth and fifth cases are discussed under *Storage Allocation*.

4.6 Time Quanta

One of the chief functions of a timesharing system is to share the amount of time that the processor has available. The simplest way of doing this is to break up the available time into intervals called *quanta* and by some means insure that each process receives an equitable number of *quanta*. There are several means by which this could be implemented. A *quantum* timer could be set when a process is started and when the timer runs out, the process could be preempted. Alternatively, some external clock could be used to provide a time signal at regular intervals. In this case, the system should start a process at the start of one interval and preempt it at the start of the next. The second scheme makes it more difficult to insure that each process has received the same number of *quanta* but has the advantage that one external clock can provide the time of day and *quantum* timing. The second scheme is the one used by the UNIX system.

If a one *quantum* interval is chosen as the maximum amount of time that a process may receive, then the system must preempt the process at the end of each *quantum* in order to enforce this limit. This, however, is not enough to insure that each process receives an equitable share of *quanta* to use the processor. As discussed

earlier, priority schemes can be used to give immediate preference for use of a quantum to a process and they can be used to decrease or limit the number of *quanta* available to a process.

4.7 Penalty Scheme

The standard UNIX system employs a very crude penalty scheme which has been retained from earlier versions of the UNIX system. The scheme makes a distinction between processes that are *compute bound* and processes that are *system bound*. Processes that execute 17 consecutive system calls without roadblocking (*system bound*) are preempted and given a one point priority penalty. (See Figure 8 for the priority variation of a process that is *system bound*.) This penalty is dropped when the process is allowed to execute again. Processes that are *compute bound* are preempted at the end of each time *quantum* that they use and given a one point priority penalty. Unlike the *system bound* case, the penalty applied is cumulative. The variation of priority for a purely CPU bound process is shown in Figure 9.

General amnesty is declared for any CPU bound process that starts to behave interactively (i.e., share the processor). Since the operating system is not reentrant, the preemption can only occur if the processor is not executing within the system. This means that preemption can occur only if the processor is executing a user process (see references [12] and [13]) or is about to execute a user process (i.e., at the end of an interrupt or system call).

The penalty scheme described above does not manage the load on the processor

very well because of the way quanta are implemented (see *Time Quanta*). To be more specific, identification of a CPU bound process depends on the fact that on the average, a CPU bound process is found to be executing in user mode when the external clock indicates the end of a *quantum*. If the process was started in the midst of the *quantum*, then it receives a one point priority penalty even though it did not use the full *quantum*. This technique is sufficient for processes that are purely CPU bound but produces inequities when there are a large number of programs whose execution characteristics are somewhere between CPU bound and I/O bound.

4.8 Research Improvements To Penalty Scheme

A new scheme implemented on the research version of the UNIX system using as base the UNIX System V/386 Release 3.2 developed on the INTEL 80386 microprocessor has a different penalty scheme. This new penalty scheme keeps track of the amount of CPU time (in sixtieths of a second) that a process accumulates between roadblocks. This cumulative CPU time is used to compute the penalty that a process receives. Processes are preempted at the end of a time *quantum* only if the cumulative CPU time indicates that the process has received a full *quantum*. A one point priority penalty is given to the process for each second of CPU time that it has used. One penalty point is accumulated for each of the first five time quanta (seconds) the process uses, thereafter, a process is preempted after using each time quantum, but priority penalties are applied only at fifteen second intervals. The *general amnesty* policy is still used so that if a process begins behaving interactively, any penalty acquired by the process is reset to zero.

Figures 10 and 11 indicate the general trend of priority penalties.

4.9 Research Improvements To Process Switching

The most significant aspect of the changes is that *general amnesty* has been discarded in the *penalty scheme*. Under a *general amnesty* scheme, the system effectively forgets about any previous CPU bound activity a process exhibited once the process begins willingly to share the processor. Thus, processes that had CPU bound phases interspersed with I/O activity could acquire more than their share of processor time because the system did not remember anything about previous (CPU bound) activity. The new scheme attempts to remember previous activity by computing a time based average that is related to the amount of CPU time that a process has received in preceding intervals.

The modifications to the penalty scheme and the process switching increased the service time to processes that willingly relinquish the CPU without an additional overhead (21% of the *switcher* time is spent in the penalty scheme, with the suggested modification, an average of 18% of the *switcher's* time is spent in the penalty scheme). For process that are CPU bound there is a service delay, using the CPU (11% delay); however processes that willingly relinquish the processor are served 23% faster when executed in the research version of the UNIX system (including all the suggested modifications) and 19% faster when executed simultaneously with CPU bound processes. There is an additional 7% overhead on the *swap* process if all the processes do not fit into memory. With a large amount of memory (only 4 Megabytes were used to acquired the above data) there is less

need to run the *swap* process. All processes are aged, but since all processes fit into memory, the processes reach and retain the maximum age in memory (assuming for simplicity, that the processes run for large periods of time). Time slicing occurs at the end of each quanta at which time the process *switcher* is invoked to choose the next process to run, so if there is no need to swap a process out of memory, there is no overhead.

4.9.1 CPU Time Average Penalty

The new penalty scheme keeps track of the number of clock ticks (one sixtieth of a second) that a process has received. The number of CPU ticks is then used to calculate the penalty that a process receives; however, the cumulative CPU time is decayed over time so that it becomes a time average of the fraction of the processor's time that the process has received. To produce this time average, once every time *quantum* the system reduces the CPU average of all users by twenty percent. The resulting value is used to calculate a new value of the process priority (penalty). For each sixteen ticks of CPU time that remain, the process is given one penalty point. As with the previous schemes, processes are preempted when the clock indicates the end of a *quantum*. The process is only preempted if its cumulative CPU average has caused its priority to become lower than that of other processes in the system. As an example, a process that has received 40 ticks of time has its CPU time average reduced by twenty percent to 32 when the readjustment is done. These 32 ticks correspond to 2 penalty points so that the process's priority is set to $100 + 2 = 102$. To illustrate how priority varies, consider one CPU bound process in the system. This process receives all of the 60 ticks (one sixtieth

of a second) in its time quantum. This is true of all succeeding time quanta since it is the only process in the system. Figure 13 shows how the CPU average for the process varies, while Figure 12 shows how there is a gradual decrease in the amount of penalty applied. (These figures should actually be a series of points rather than curves since priority assignment is a discrete function, however, for clarity, a smooth curve has been drawn to indicate the trend of the priority). Note that it takes eight seconds for the process to accumulate twelve penalty points and that half of those points are accumulated in the first three seconds. If we now consider a process reaching the maximum priority and roadblocking, the length of time that it takes for the CPU average to decay can be found. Figure 14 shows that it takes about fifteen seconds for the CPU average to decay from its maximum value (255) to a point where no penalty results.

The previous example shows the maximum rate of accumulation of penalty points because there was only one process in the system. Figure 15 shows how the priority of a CPU bound process changes if there is zero, one or two other CPU bound processes in the system. (A smooth curve has been drawn through the average priority value for the two and three process cases to show the trend of penalty values because there is a wide dispersion of values). Note that because there are other processes in the system available to use the CPU, the *rate* at which penalty points are accumulated is lower and the maximum penalty is lower. By assuming that each process is CPU bound we require that each process use its full *quantum* (one second) before being preempted to allow another CPU bound process to run. Similar results to those of Figure 15 could be obtained if each process only used **1/N** of a time *quantum* before roadblocking. For example, in the case of three CPU

bound processes, each of which runs for 1/3 quantum before roadblocking, the priority penalty curve shows a similar decrease to that of Figure 15 although the maximum penalty is different.

4.9.2 Special Penalty

In addition to the penalty scheme discussed above, processes that have used *one minute of CPU time* have an automatic priority penalty of two points applied. This penalty is applied by automatically using the *nice* system call mechanism to lower the process's priority. This penalty cannot be removed unless the process explicitly uses the *nice* system call to raise its priority. This additional penalty not only decreases the fraction of CPU time that the process receives but makes the process more susceptible to being swapped out and forces the process to wait longer before being swapped into memory. This interaction with the *scheduler* is discussed in a later section (see *Scheduling Penalties*).

4.10 Maximum Penalties

In general, from the schemes discussed above, the gross multi-programming of the processor can be seen. The schemes are intended to make processes that would not normally willingly share the processor, share it in an equitable manner. It should be noted that when there are a number of penalized processes in the system, a new CPU bound process entering the system has no penalty and so is allowed to run for a number of time quanta until it reaches the same priority as the process with the smallest penalty. (For example, if there are several processes in the system with priority 105, a new process entering the system at priority 100 is allowed to

run for enough *quanta* to reach priority 105.) Under the first two penalty schemes described above, there is a linear relationship between the time *quanta* a process receives and its penalty so that, for a 5 point priority difference, 5 quanta are allowed the new process before it reaches the same priority. In addition, it should be noted that for these two schemes, the maximum penalty depends on the number of *quanta* a (CPU bound) process has used and is independent of whether there are other processes in the system. In the latest penalty scheme described above, the maximum penalty decreases as the number of CPU bound processes in the system increases (see Figure 15), so that if a new process enters the system, the number of *quanta* it receives depends on the number of (CPU bound) processes in the system.

4.11 Preemption

The previous sections have dealt with enforced sharing of the processor to limit the amount of time that a process is allowed to use the CPU. Hopefully, the job mix is such that more sharing can be achieved than the processor sharing imposed by the system.

A mechanism exists within the system whereby a process can relinquish its use of the CPU (see *Traffic Controller Section*) until some event occurs (I/O completion, etc.). A complementary mechanism for unblocking the process when the event has occurred also exists. When the process willingly relinquishes the CPU, the process *switcher* is invoked in an attempt to find another process to use the processor. The question which immediately arises is "how long after the completion of the event must a process wait before it regains the processor?" The answer to this is not

simple since the system uses a priority scheme to assign a process to the processor. However, some insight can be gained by examining several examples.

Basically, when a process is roadblocked, it is assigned a priority depending on the activity that requires it to be roadblocked. The system assigns these priorities based upon its own estimate of the relative urgency at which the activities should be serviced when they are roadblocked. The strategy adopted when the process is awakened (via a *wakeup*) is to preempt the currently executing process and invoke the process *switcher* to select the highest priority process to run. This is subject to several restrictions because the operating system is not reentrant. Figure 8 shows an example of processes being multi-programmed. Three states are shown for each process. The kernel mode level (K) indicates that the process is executing within *system space*. The user mode level (U) indicates that the process is executing within *user space* and the last state indicates that the process is not executing (NR - not running). In Figure 16, two processes α and β are shown. Process α is using the processor at the beginning of the time *quantum* and while making a system call roadblocks (time RB) because of the unavailability of some resource. Process β is then given the processor, but before the *quantum* is finished (time 1), the resource that cause α to roadblock becomes available. Since β is executing in *user space*, it can be preempted and the processor returned to α. At the end of the quantum (time 1), the system probably preempts the currently executing process (α). (This depends on which version of the penalty scheme a system is using.)

In Figure 17, the situation is repeated but with a slightly different twist. Here, process β is executing in *system space* (i.e., making a system call) when the resource

that cause α to roadblock becomes available (time A). In this case, the system waits until process β completes its system call (or willingly roadblocks during the system call) before preempting it and giving the processor back to α. A more complex example is shown in Figure 18. Here, three processes are being multiprogrammed and process γ is executing in *system space* when the resources that process α and β are waiting for become available (time A). There is no attempt by the operating system to remember the fact that two processes have been awakened. The system simply behaves as in previous case and waits for the system call to be completed (time P) before preempting process γ and selecting the process with the higher priority.

5. THE SCHEDULER

The UNIX system *scheduler* controls the entrance of processes into memory. It attempts to share memory among all of the processes in the system so that they all, regardless of size, have an equal opportunity to use memory. Since memory is a scarce resource, it may at times be forced to throw some processes out of memory and in this respect, it regards the swap area as an infinite sink for placement of processes. The swap area is, of course, a finite resource and if fragmentation causes the swap area to be filled, the system panics. ("**Panic: out of swap space**" is printed on the system console before the system crashes). Currently, most installations allocate a relatively large swap area as a countermeasure; however, some work has been done to prevent the creation of new processes when there is no swap space available and thus reduces the probability of running out of swap space. Traditional implementations of the UNIX system use one swap device, but the latest implementations of the UNIX System V allows multiple swap devices. The kernel chooses the swap device in a round robin scheme, provided it contains enough contiguous memory. Administrators can create and remove swap devices dynamically. If a swap device is being removed, the kernel does not swap data to it; as data is swapped from it, it empties out until it is free and can be removed.

While the *scheduler* controls all entrances into memory from the swap area, it is not the only function in the system that removes processes from memory. Any process in the system may swap itself out (or a copy of itself in the case of *forking*) if it requires memory that is not available (see *Storage Allocation*).

As with the process *switcher*, understanding *scheduling* under the UNIX system

requires both a knowledge of what the systems policy for sharing memory is and the times when that policy in invoked.

5.1 Scheduling Parameters

The *Scheduling Algorithm* is a function of the size requirements of processes[14], their location, the amount of time that they have resided in memory or in the swap area, and which processes are *ready* to execute. If a function were to be specified to represent the scheduling algorithm, it would be invoked as,

$$M(t, l, s, f)$$

where M is the function which computes which process is to be swapped in or out and the meaning of the four variables is as follows:

t — This is the process *age*. This *age* is the chief criterion used by the *scheduler* in determining which processes are to be brought into or removed from memory and when action is to be taken to rearrange memory. This *age* represents the amount of time the process has spent in memory or, if the process is not in memory, the amount of time the process has spent on the swap area. Process *age* is kept in seconds and is only a crude measure of the true length of time that a process has spent in memory or in the swap device.

l — This is the amount of memory required to bring the process into memory. This is a quantity which can only be determined

by the *scheduler* when making the decision as to whether there is enough memory to fit a process into memory. This complexity arises because of the complication added by sharing reentrant text. That is, a reentrant process is composed of two separate pieces, reentrant and nonreentrant. The reentrant part may be shared with other processes and is managed independently of the nonreentrant part so that only the *scheduler* can decide whether the reentrant part needs to be swapped in and hence only the *scheduler* can determine how much memory is required to swap a process in.

s — This is the state of a process. Only processes in the *sleep*, *wait*, *ready* or *stopped* state are considered by the *scheduler*. Processes that are in the *idle* state are excluded from consideration since they are doing I/O directly from their own address space and could not be removed without terminating the I/O. (The *idle* and *stopped* states are discussed in a later section).

f — This is the location of a process. A process is either resident in memory (*loaded*) or it is on the swap device (*not loaded*). There is actually a third value of this variable. A process can be locked in memory (*locked*). Processes that are *locked* are removed from consideration until they become unlocked (i.e, *loaded*). The *scheduler* is always locked in memory and processes doing physical I/O are locked in memory while I/O

occurs from their address space[3].

As discussed in the following sections, the result of applying the *Scheduling Algorithm* is to select one process to either be brought into or removed from memory. Only when the selected process has completed the transition to or from memory is the *Scheduling Algorithm* reapplied to find another candidate.

5.2 The Scheduling Algorithm

With the foregoing introduction to the variables used in scheduling a process, the algorithm is described in detail. It should be remembered that the major intent of the *scheduler* is to share equally, among all of the processes in the system, the system's limited amount of memory. By *equally*, we mean that each process has the opportunity to use memory for the same amount of time (i.e., the residency time or *age* as described above) regardless of its size. Another fact that should be remembered is that while the *scheduler* usually bases its selection on *age*, it is sometimes more useful to select other processes to be swapped out, namely low priority roadblocked (*waiting*) processes. These processes may be roadblocked for a long period of time so that it is senseless to allow them to remain in memory when an executable process could be brought in. These processes are not, however, automatically thrown out of memory when they roadblock at low priority. Rather, only when there is a demand for memory by processes on the swap area (i.e., there are *ready* processes on the swap area) do they become candidates to be swapped

3. This leads to a problem which was recently discovered. A process doing a number of consecutive physical I/O operations ties up memory for long periods of time as the *scheduler* (on the average) finds that they are locked in memory and are therefore unremovable

out. Roadblocked processes may be awakened at any time so that, rather than select one *ready* process and single mindedly clear enough memory to bring that process into memory, the *scheduler* reexamines the queue of processes *ready* to enter memory to see if any *older* processes have been awakened.

The first step in the *Scheduling Algorithm* is to determine whether there are any processes on the swap area to be brought into memory, since if there is enough memory to fit all processes in memory, there is no need to do any scheduling. This is done by scanning the *Process Table* (a linear search) to find the oldest *ready* process on the swap area. If there are no *ready* processes, the *scheduler* roadblocks and posts an event (*runout*) until there is at least one *ready* process on the swap area. The selection algorithm can be described as follows,

(1) Choose the *oldest* process on the swap area from all processes $\alpha(t,s,f)$ which are *ready* (s = *ready*) and *in memory* (f = *not loaded*).

If two processes have the same oldest age, the *scheduler* chooses the first process it encounters. (Note that the selection is made by doing a linear search of the Process Table so that there is a bias to select processes that are at the lower end of the Process Table. Note also that no preference is given to the processes because of their size). The memory requirements of the process that has been found to be the oldest process are then determined[4]. At this point, there may be enough

4. For reentrant processes that are *ready* to enter memory, but whose reentrant text is not already in memory, the *scheduler* requests a contiguous piece of memory equal to the sum of the reentrant and non-reentrant portions. Since reentrant text is managed separately in memory from the non-reentrant portion, this makes the memory request more severe than it need be.

(contiguous) memory available for the process to be brought into memory, in which case the process is swapped in and the *scheduler* goes back to look for more *ready* processes in the swap area (criterion 1 above). If there is not enough memory available to bring the process in, the *scheduler* goes into a different mode of operation in which it looks for a candidate to remove from memory instead of one to bring into memory. When this situation occurs, the *scheduler* temporarily forgets about the process that it had selected to bring into memory, so that selecting a process to bring into or remove from memory becomes a disjoint operation.

Assuming that the *scheduler* has no available memory to bring processes into memory, it has two criterion that it can use for selecting processes to be removed from memory. The first criterion requires that the *scheduler* remove from memory the *first waiting* process that it finds in the Process Table. That is,

(2) Choose the *first waiting* process from all processes $\alpha(s, l)$ which are *waiting* ($s = wait$) and are *in memory* ($l = loaded$).

This is done because processes that are in the *wait* state are usually roadblocked for a long time (probably greater than one time quantum, i.e., 1 second) so that it is reasonable to place them on the swap area and make their memory available for use by other processes. If a *waiting* process is found, it is swapped out; however, rather than just see if the process selected by criterion 1 (above) now fits in memory, the selection process is restarted. This insures that any process on the swap area that was awakened while the swap took place can be considered and given preference if it is older than the process originally selected. The net effect of

this swap out criterion (criterion 2) is that eventually *all* processes that remain in the *wait* state are thrown out of memory if there are any *ready* processes on the swap area.

If there are still *ready* processes on the swap area after memory has been cleared of *waiting* processes, then more drastic measures are needed to free memory[5]. However, before this new strategy is started, the *scheduler* checks the *age* of the process originally selected to be brought in (i.e., the age of the process selected by criterion 1) to see if it has been on the swap area for three seconds. This is done to prevent the swap device channel from becoming too active as is shown later (see *Thrashing*). If the oldest process on the swap area does not meet this criterion, the *scheduler* roadblocks itself (posting the event *runin*) until some event occurs that frees memory or changes the eligibility of processes (e.g., time passes, a process goes into the *wait* state, etc.). If there are any *ready* processes on the swap area that have been there for at least three seconds, the *scheduler* looks for the oldest *ready* or *sleeping* process in memory as a candidate to be swapped out. That is, it looks for a process that satisfies the following criterion,

(3) Choose the oldest process from all processes $\alpha(t, s, f)$ which are *ready* (s = *ready*) or *sleeping* (s = *sleeping*) and are *in memory* (f = *loaded*).

If two processes have the same oldest age, then the first one encountered is chosen.

5. Under the research version of the UNIX system, processes that are *stopped*, because they are being traced, are treated the same as *waiting* processes.

The process chosen by this criterion must have been in memory (i.e., must be at least two seconds old) before it can be considered for removal from memory. (This is also a criterion to prevent thrashing and is discussed in the next section). If the process selected by this criterion has been in memory for two seconds, the *scheduler* swaps the process out. If the process selected by this criterion has not been in memory for two seconds, the *scheduler* roadblocks itself (posting the *runin* event) until some event occurs that allows it to rearrange memory. When the swap out is completed, the *scheduler* returns to applying criterion 1 to the queue of processes on the swap area to find a candidate to bring into memory.

One deficiency of this Scheduling Algorithm is that since the *scheduler* works mostly on an *age* basis in selecting a process to be thrown out of memory, it is possible for it to do a lot of work throwing many small processes out of memory before it creates a space large enough to bring a large process into memory. Among processes that have the same age, it may also throw out the process which results in the least gain in contiguous memory. A better strategy might consider relative gains in contiguous memory among processes with equal age.

In summary, the *scheduler* runs in a piecemeal fashion, selecting one process to be swapped out or into memory by examining the queue of processes *ready* to be brought into memory and the queue of processes eligible to be swapped out. When a situation arises where no more rearrangements can be done, the *scheduler* roadblocks. When conditions change (e.g., the process *ages* change, processes changes states), the *scheduler* is awakened and starts rearranging memory again.

5.3 Thrashing

Thrashing has been defined in many different ways and studied widely, particularly in connection with demand paged operating systems. Basically, thrashing is a point in the operation of a system when severe performance degradation is reached due to the system's attempt to over-optimize its resources. To quote one author,

"Thrashing inevitably turns a shortage of memory into a surplus of processor time[15]."

Even in a nonpaged operating system, thrashing is possible. This is because in its attempt to share the amount of time that a process may use memory among all process in the system, the fact that there is a finite (and sometimes large) *traverse time* in moving a process between memory and the swap device cannot be neglected.

As discussed in a previous section, there are two limits imposed on the *scheduler* when selecting *ready* or *sleeping* processes to be swapped out. The two limits attempt to insure that a process that has been swapped into memory has a reasonable opportunity to use the processor before being swapped out and that the *scheduler* does not saturate the swap channel with requests.

In selecting a *ready* or *sleeping* process to be removed from memory, the *scheduler* requires that the *age* (i.e., the length of time that the process has spent in memory) of the process chosen to leave memory be greater than two seconds. Because of the way process aging is implemented, an early or late arrival of a process in memory

can result in the process being overaged or underaged (refer to Figure 2). For the cases that are examined, a worst case situation, where all processes arrive early so that they are overaged, is assumed. (That is, their *age* indicates that they have been in memory longer than they actually have - see Figures 2 and 19). This means that the *scheduler's* requirement that a *ready* process's age be greater than two seconds before it is eligible to be thrown out of memory can mean that the process has been in memory for as little as one second. Figure 19 illustrates a sample case. Here, a situation has been chosen in which there is little memory and there are a number of processes on the swap area that are *ready* to run. The age of the currently executing process (α) is shown as is that of the oldest process on the swap area (β). The start and finish of swap-out and swap-in are indicated and since it is assumed that all of the processes are of uniform size and that there is enough memory to fit only one process at a time, the processor is *idle* while an interchange between memory and the swap area is being made.

Requiring that a process reside in memory for at least one second insures that a swapped in process remains in memory for a minimum of one second (i.e., one time quantum). This assumes of course, that the process does not terminate or relinquish the processor at low priority (*wait* priority). Thus, even if the process were to relinquish the processor at high priority (go into the *sleep* state) it cannot be swapped out until it has been in memory for one second even though the processor may stand *idle* while the process is *sleeping*.

Another important factor to be gleaned from the example is that the amount of time that the process is allowed in memory is *independent of the traverse time*.

That is, the minimum amount of time that a process remains in memory is independent of the speed of the swapping medium. Figure 19 shows two-way traverse times that are of the same order as the time quantum (one second) but the same results hold for very short traverse times. It can also be seen that removing the one second restriction would cause the system to thrash because there would be nothing to prevent the *scheduler* from continuously cycling processes to and from memory.

When choosing a *ready* or *sleeping* process to remove from memory (see criterion 3 - *Scheduling Algorithm*) there is an additional restriction that the oldest process on the swap area must have resided there for at least three seconds. This restriction relates to the idea of *system balance* and is illustrated more easily than it is defined.

In Figure 20, a situation is shown wherein there is enough memory to fit all of a series of (uniformly sized) processes except one. The one way *traverse* time is again shown to be about one time quantum. It can be seen that the processes circulate from memory to the swap device in a round robin (by age) and that the CPU is kept busy. (Here, the processes are shown as CPU bound for the sake of simplicity so little multiprogramming occurs within one quantum.) The channel to the swap device is saturated, however, and the system swaps one process out of memory only to have that same process immediately force another process to be swapped so that it can come back in. The decrease in activity of the swap channel by enforcing a swap residency requirement is shown in Figure 21. Here the three second residency requirement alluded to in the previous section is enforced.

For the AT&T 6386 WGS architecture, the fact that the channel is saturated decreases the bandwidth across the (hardware) AT-BUS, since the controllers communicate with memory over this I/O bus and thereby decrease the number of bus cycles available to the processor[16]. (This is not as severe a problem on computers with multiple I/O busses and cache.) The degradation that occurs because of the interference on the bus is not usually too severe since the amount of time to transfer one word of memory across the AT-BUS (about 1 μsec. see reference [17]) is faster than the transfer time of the fastest disk (about 2.5 μsec.) available for this architecture. The transfer times are hardware dependent and vary among manufacturers.

Also consider that the CPU utilization is affected by the fact that the swap device is on the same device (or controller) as other file systems used by the system (the AT&T 6386 WGS utilized during this research was configured with 1 hard disk only). This occurs because the disk controller can handle only one request at a time and the I/O initiated by a process must wait in the same queue as requests for swapping (disk controllers are not smart enough to differentiate among users). A particularly bad situation occurs if only two processes fit in memory and there is only one device for swapping and for file systems. A process that is allowed into memory for one second quantum might not be able to do any useful work because it issued I/O requests while the I/O channel was tied up doing a swap.

5.4 The Scheduler, Implementation

The *scheduler*, unlike the process *switcher* is a process in the system; however,

there are several major differences between the *scheduler* and ordinary processes.

1. The *scheduler* is not created by the *fork* mechanism as are all other UNIX system processes. During system initialization, the system directly creates the *scheduling* process as the first process in the system. It is always process zero in the Process Table.

2. The *scheduler*, unlike all of the other processes in the system, runs entirely in *system space* so that it may have direct access to system functions. The size and location attributes of the *scheduler* do not correspond to its true size and location as these attributes do for ordinary processes. The scheduler's size and location correspond to the size and location of its *u block*. The real values of these quantities are unimportant for the *scheduler* as it is *always locked in memory*, however, since the *scheduler* is the parent of the *init* process, it is convenient to make the *scheduler's* size appear small to minimize the overhead when creating *init*.

3. The *scheduler* never dies. It is an endlessly looping process within the system constantly looking for an opportunity to rearrange memory.

4. The *scheduler* only goes into the *sleep* state. It never goes into the *wait* state nor does it ever go into the *sleep* state at anything but the highest priority (0). This is done because the *scheduler* manages the most important and most scarce resource in the system (memory) and so deserves the best service.

With these facts in mind, the way that the execution of the *scheduler* is synchronized is discussed.

5.5 Scheduling Policy And Scheduler Synchronization

Now that the algorithm for selecting a process to be swapped into or out of memory has been described and the criterion (policy) that the system uses for scheduling a process has been discussed, the times when this policy is invoked must be discussed.

The *scheduler* is a passive process in the system. Only when there are more processes in the system than there is available memory, does the *scheduler* become active in its role of cycling processes into and out of memory (see Figure 22). To implement this behavior, there must be a means for synchronizing the execution of the *scheduler*. In addition, there must be a set of rules which specify when it is appropriate to notify the *scheduler* that the status of a process has changed and rearrangement of resources is possible. The *scheduler* must be notified rather than invoked as a function since the *scheduler* is a process like any other process in the system and must wait its turn to use the processor.

There are basically two synchronizing events which are used by the *scheduler* to synchronize its execution. The first event, *runout*, indicates that the *scheduler* has no contenders for memory on the swap device. The second event, *runin*, indicates that there is a *ready* process on the swap area, but that more memory is needed before the *scheduler* can bring the process into memory.

The *runout* event is posted when the *scheduler* stops trying to bring processes into memory because there are no more *ready* processes on the swap area. When the *scheduler* is in this state, it is awakened under the following circumstances,

1. When any process on the swap device becomes *ready* the *scheduler* is awakened.

2. The *scheduler* is notified when any *ready* process swaps itself out. (This is essentially a special case of 1). The *scheduler* is not awakened until after the process arrives on the swap area. A process usually swaps itself out of memory when it needs memory to grow in size or to create a new process in memory (see *fork*) and no memory is available. Awakening the *scheduler* under these circumstances probably puts it in a state where it is looking for processes to remove from memory to create space to bring the swapped process in (after the swap residency requirement is satisfied). Of course, the notification need not be sent if the *scheduler* is already looking for free memory to bring a process in (i.e., the *scheduler* is already roadblocked on the *runin* event - see below).

3. The *scheduler* is also notified when any *waiting* process *on the swap area* receives a *signal*. This notification is done because *waiting* processes are made *ready* whenever they receive a *signal* even though the resource that caused them to roadblock is not available. When a *waiting* process on the swap device is awakened, it is usually discovered that it is a relatively old resident of the swap device, so that it is a prime candidate for the *scheduler* to bring into memory. Notification of the *scheduler* is done to improve the responsiveness of *waiting* processes to *signals* and since many processes are terminated by receiving a *signal*, it allows the speedy release of system resources used by the signaled process.

If the *scheduler* can not find enough memory to bring a process into memory, the *runin* event is posted and the *scheduler* roadblocks itself. The event *runin* is used to notify the *scheduler* of the availability of memory or any change in the eligibility of a process to lose the memory it is occupying (i.e., eligibility to be swapped out). The *scheduler* is notified that memory is available under the following two circumstances[6],

1. If a process relinquishes the processor at low priority (*wait* priority), the *scheduler* is notified. This is done because *waiting* processes are prime candidates for removal from memory since there is no age criterion applied to a *waiting* process. They are simply thrown out of memory if there are any ready processes on the swap area.

2. The *scheduler* is notified when process ages change (once every second) thereby changing the eligibility of a process to be swapped. The notification is, of course, sent only if the *scheduler* is looking for memory so that it can swap a process in.

The *scheduler* relinquishes the processor at the highest (software) priority so that whenever it is awakened, it is run shortly thereafter. When it runs, it swaps one process out to free space in memory, then swaps another process in. The *swbuf* event shown in Figure 22 is posted when the swap starts and is used to notify the

6 Strangely enough, the *scheduler* is not notified if a process terminates in memory. Thus, if a process terminates, the memory it frees could remain unoccupied until process ages are updated (a maximum of one second later). The author can find no reason for not notifying the *scheduler*, except that possibly it leaves some memory free so that an in memory process has the opportunity to do an *exec* or *fork* in memory instead of on the swap device.

scheduler that the swap has been completed. The *scheduler* continues in serial fashion, roadblocking to wait for each swap I/O to complete until it ends up roadblocked on one of the two events (*runout* or *runin*) discussed above.

5.6 Interaction Of Process Scheduling And Process Switching

Different models of the system have been constructed to explain its operation. These emphasize *processor* queuing and deemphasize scheduling of processes into memory. The reverse emphasis is illustrated in Figure 23. Here two queues are shown for the process *scheduler*. The *non-resident* queue (i.e., queue of processes on the swap area) contains all of the processes that are ready to be brought into memory. The second queue contains the *resident* (in memory) processes that are waiting their turn to be released onto the swap device (i.e., onto the *non-resident* queue). From the criterion discussed in the section on the *Scheduling Algorithm*, it can be seen that *ready* processes circulate from memory onto the swap device and back again. The delay encountered by a non-resident process in getting back into memory is dependent on the available space in memory and on the number of processes that must be removed from memory to make room for that process to reenter memory. The overall flow is, however, one in which processes circulate from secondary storage through main memory and back again until they have received enough processor time to complete. During the period that a process is resident in memory, it is eligible to share the processor. It receives processor time based on its priority and is subject to certain limits. The coupling shown in Figure 23 between *scheduling* and *switching* indicates that even though there are two different schemes used for the *scheduler* and process *switcher*, the fact that only

processes in memory can be multiprogrammed means that there is a loose coupling between the two[7].

5.7 Scheduling Improvements On Research UNIX System

The research version of the UNIX system product of this effort has had a number of changes made to the *Scheduling Algorithm* in an effort to improve its operation. These improvements are discussed in the following sections.

5.7.1 Research Improvements To The Scheduling Algorithm

The *Scheduling Algorithm* was modified so that when choosing a process to swap out, the *size* is taken into account. In particular, with regard to criterion 2 applied by the *scheduler*, the *scheduler* chooses the *largest* of all *waiting* (or *stopped*) processes, the relative *size* of processes is still not taken into account, only process *ages* are considered.

5.7.2 Research Improvements To The Scheduling Penalties

Another change made to the *Scheduling Algorithm* allows a process's memory usage to be penalized. In particular, the *nice* system call was formerly used to penalize only a process's CPU usage. It now affects a process's memory usage by modifying

7. At least one empirical study of the interaction of ***scheduling*** and ***switching*** using several different algorithms and comparing preemptive versus nonpreemptive schemes has been made by AT&T Bell Laboratories. However, it is difficult to generalize the results of this kind of study, since the amount of core storage (large storage means near-zero swap time) and the time quantum utilized made the observations differ substantially.

the process's age. In particular, each penalty point applied by the *nice* system call causes the process's residency (*age*) to increase by one second. Thus, if a process has been in memory for one second, it appears to the system to have been in memory for two seconds for each penalty point. The chief implication of this is that when the process is swapped into memory, it starts off as prime candidate to be swapped out. There is a further penalty applied to the process by this arrangement. When considering processes to bring into memory, the *age* of a process that has used *nice* is adjusted so that for each penalty point, the process's residency on the swap area (*age*) appears to be decreased by eight seconds. This means that the process is less likely to be brought into memory since it appears to have resided on the swap area for a shorter time. A process that has had a one point priority penalty applied starts off with an age of -8 when it is swapped out. Thus, unless there is a lot of free memory or very few processes in the system, the process spends less time in memory and more time on the swap area.

In the section on ***Improvements In The Process Switcher***, note that the automatic two point penalty applied to processes that have received ***sixty seconds*** of CPU time causes the memory usage of processes penalized by that scheme to be affected. In particular, with the two point penalty applied by that scheme, a process fulfills the two second residency requirement when it is swapped in and so could be swapped out immediately by the ***scheduler*** whenever it is looking for memory.

The modifications performed to the ***Scheduling Algorithm***, increased the service time for small jobs by 11% and for large jobs by 5% because of the need to swap more frequently. This created an overhead of 9% for the swap device (1 Megabytes

of memory and a swap device size of 102 blocks were utilized). This overhead exists only if there are process that need to be swap in/out from disk rather than from memory queue. When more memory is utilized and assuming that is no need to swap any process out, there is no need for a *scheduler* in the system to cycle processes through memory because there is enough memory for all processes to execute (assumed for simplicity). The *scheduler* roadblocks on the *runout* event since there are no processes on the swap area (memory queue) competing for memory. The swap device overhead heavily depends on the amount of memory available, the number of processes executing simultaneously, and the size of the executing processes. The minimum possible memory configuration required by the standard UNIX System V/386 Release 3.2 is 4 Megabytes, and it is the configuration utilized throughout this research. However, when memory was increased to 16 Megabytes the overhead disappeared.

6. STORAGE ALLOCATION

In the following sections, the storage allocation algorithm is described followed by a discussion of the way the space occupied by a process is managed.

6.1 First Fit Algorithm

The *First Fit* algorithm is a storage allocation strategy whereby the first available space equal to or larger than the required size is used to allocate space from. This is in contrast to a *Best Fit* algorithm which examines all free spaces to find one large enough to allocate the required space, but which leaves the smallest amount of unused space. In either case, if the area selected is larger than the required space, only the amount required is carved from the free space.

A *First Fit* algorithm is used to allocate memory or swap area to a process running under the UNIX system. To implement this, a *free list* (table) is kept for each of the available areas of these resources. A separate list is used for each of the two resources to decrease the search time and because the size and addressing of the two resources is different. The lists are ordered by increasing address of the available area and any contiguous areas are combined into one entry. An *occupied list* which specifies what areas are occupied exists in the form of entries in the Process Table and the Text Table, however, these are kept in no particular order.

The relative merit of the *First Fit* algorithm versus the *Best Fit* algorithm has been studied[18] and has shown that there is less overhead with the *First Fit* algorithm since space is probably found without having to search the entire table. Another

interesting feature of the *First Fit* algorithm is that the fragmentation that it produces is not likely to be as bad as that of *Best Fit*. That is, the *Best Fit* algorithm is more likely to produce very small fragments in its effort to find the tightest fit. One problem with the *First Fit* algorithm, however, is that it concentrates its efforts on the low end of the free list which usually results in the fragmentation accumulating in the lower end of the list while the upper end remains relatively compact. This of course means that it takes longer to allocate space because there are a number of small fragment entries concentrated at the beginning of the list. A modified *First Fit* algorithm has been suggested by Knuth[18]. It remembers where the last free block was found and starts the search from space from there, thus distributing the fragmentation over the entire free list rather than concentrating it at one end.

6.2 Allocation Mechanism

Since a *First Fit* algorithm is used for both the allocation of swap space or memory under the UNIX system, and the mechanism is the same for both, we use the term *storage allocation* in this section with the understanding that the mechanisms apply equally to swap or memory allocation.

Figure 24 shows the allocation of storage (unlined areas) from a free area (lined area). The entries in the *Free List* are shown before and after the allocation takes place. Note that the number of entries in the *Free List* (and hence the *maximum* number that must be scanned to determine whether there is space available) has not changed. The required area is carved from the free area. The only change is

that one of the *Free List* entries manages a smaller fragment after the allocation.

Figure 16 also illustrates the process of *recombination* if the inverse operation is examined. That is, if the right hand half of the figure is taken as the initial state, then freeing storage associated with the previously allocated space results in the *recombination* of the small fragment with the newly freed area. A more complex *recombination* is illustrated in Figure 25. Here, a badly fragmented area is shown before the release of an area that is adjacent to two free areas. These are combined to produce only one free area and result in a decrease in the number of entries in the *Free List*.

Figure 26 illustrates the allocation of a free area that is exactly the size required. This results in a decrease in the number of entries in the *Free List*. Figure 27 shows the deallocation of the same free storage which results in the creation of a new area.

Note that in Figures 25, 26 and 27 the allocation has resulted in a net increase or decrease in the number of entries in the *Free List*. The UNIX system maintains the *Free List* as a table so that this increase or decrease in the number of entries means that all of the entries below the inserted or deleted entry must be pushed down or popped up in the table to add or delete an entry.

6.3 Process Setup

The UNIX system maintains programs in a contiguous memory piece to reduce the amount of swapping overhead. (The amount of overhead required to swap out

separate fragments outweighs any gain in efficient memory usage. During this research an empirical study was performed which found this to be the case, at least for the AT&T 6386 WGS computer architecture. Is possible that in a computer architecture capable of command chaining, as on more sophisticated machines, separate segments could be swap out as fast as contiguous ones.) Reentrant programs are, however, broken into *two* contiguous pieces (a reentrant part and a nonreentrant part) to gain the advantage of sharing the reentrant part among several processes. Reentrant text (text refers to the instruction part of a program) is managed by a separate Text Table entry which records the location of the text in memory and on the swap device and keeps track of the number of processes using the text. A copy of the text portion is maintained on the swap device for the length of time that the reentrant process using it is in the system. This eliminates the need to swap the text out. (It must, of course, be swapped in if it is not in memory). The memory occupied by the reentrant text is only released to the system when there are no processes in memory using the text. Similarly, the swap area occupied by the copy of the text is only abandoned when the last process using that text leaves the system.

When processes are brought into memory, any area that they occupied on the swap device (except for reentrant text) is freed. The area could have been retained in the hope that the process would return to the swap area with the same size as it left, however, there is a sufficiently high probability that the process requires more space or terminates and becomes smaller and therefore this approach is not used.

Figure 28 attempts to illustrate how the system manages the storage required for

processes. There are four possible cases that may occur. Greek letters are used to represent different processes. Any Greek letter subscripted with a **T** represents the *reentrant* part of a process (reentrant text). Any Greek letter subscripted with a **D** represents the nonreentrant part of the process (nonreentrant text, data, bss[8] and stack) including the per process information in the *u block*[9]. Process β in Figure 28 is a nonreentrant process loaded in memory. The Process Table entry for β manages the storage that it occupies. Similarly, γ is a nonreentrant process which is on the swap area and is managed by a Process Table entry. Process δ is reentrant but is swapped out. Note that a copy of the reentrant text (δ_T) is managed separately from the nonreentrant part of the process (δ_D). For process δ, there are no processes in the system that are sharing a copy of the text associated with δ_T). Process α is reentrant, but it is loaded in memory. Note that the text α_T remains on the swap area even though the text is brought into memory. The swap space occupied by the nonreentrant part of α is freed when process α is swapped in by the *scheduler*. Note also that the text α_T need only be swapped in if it is not already in memory.

With the separate management of reentrant and nonreentrant parts of a process, the size of the *Free List* table required to manage free areas in memory or on the swap device, can be determined. If a system intends to support **N** sharable processes, then a *Free List* table that can handle the worst case of 2N+1 fragments

8 The name bss comes from an assembly pseudo-operator on the IBM 7090 machine, which stood for "block started by symbol." The data region of a process is (initially) divided into two parts: data initializes at compile time and data not initialized at compile time ("*bss*").

9 The *u* in *u block* stands for user. Another name for the *u block* is *u area*, this research always refers to it as the *u block*.

is required. Thus, for a system of 50 processes, a free list that can contain 101 fragments is required. The UNIX system as implemented in the AT&T 6386 WGS for Releases 3.1 and 3.2 currently uses a table of size N/2 (e.g., 25) for an N (50) process system because fragmentation is not usually a problem.

6.4 Swap Area Free List

The *free list* for the swap area is maintained in disk block granularity (512 bytes/block). (Some disks do have sector sizes that are smaller than 512 bytes, but this is ignored by the swap allocation functions and the I/O subsystem makes all block devices logically organized as 512 byte block (sector) devices even if the physical blocking is different.) Addresses and sizes of free areas are kept in 512 byte block granularity. As discussed previously, reentrant text is managed separately from the nonreentrant parts of the program. This means that reentrant programs require swap space for only one copy of the reentrant text. Thus, both swap space and memory are saved when processes share the same text.

6.5 Process Size

The minimum amount of space that a process can occupy can be calculated by knowing that every process must have a *u block* and that the system always assigns some memory to a process as the initial stack size. The *u block* is a 1764 byte area associated with a user's process that contains unique information about the process (followed by the kernel stack which is only a few hundred bytes deep). It is also used as the system's stack when the user process makes a system call or an

interrupt occurs while the user process is executing. This area is contiguous to the user's program even though it is not within the user's virtual address space. Only the system ever references it and keeps it contiguous with the user's process. It can be swapped in and out with the user's program. In addition, the initial stack size that is allocated to a process is twenty memory blocks (20 * 64 = 1280 bytes), so that the absolute minimum size of an active process is 1764 + 1280 = 3044 bytes. The research implementation of the UNIX system uses a different initial stack size. (See section 7.1.5).

7. RESOURCE ALLOCATION AND DEALLOCATION OPERATIONS

There are a number of functions in the system which use the memory allocation and deallocation functions heavily. These functions are described in the following sections.

7.1 Fork

Fork is the only means by which a *new* process enters the system. A *fork* system call results in the replication of the process making the *fork* call. The newly created process is a *child* of the creating process (the *parent*). Since there may not be enough memory available to do the replication in memory, the *fork* system call must be capable of making the copy on the swap area. The syntax for the *fork* system call is

```
pid = fork();
```

On return from the *fork* system call, the two processes have identical copies of their user-level context except for the return value *pid*. In the parent process, *pid* is the child process ID; in the child process, *pid* is 0. Process 0, created internally by the kernel when the system is booted, is the **only** process not created via *fork*.

7.1.1 The Fork Implementation

The UNIX system kernel performs the following sequence of operations for *fork*.

1. It allocates a slot in the process table for the new process.
2. It assigns a unique ID number to the child process.
3. It makes a logical copy of the context of the parent process. Since certain portions of a process, such as the text region, may be shared between processes, the UNIX system kernel can sometimes increment a region reference count instead of copying the region to a new physical location in memory.
4. It increments file and inode table counters for files associated with the process.
5. It returns the ID number of the child to the parent process, and a 0 value to the child process.

The implementation of the *fork* system call is not trivial, because the child process appears to start its execution sequence out of thin air. The algorithm for the *fork* system call varies slightly for demand paging and swapping UNIX systems; the ensuing discussion is based on traditional swapping systems as implemented on the AT&T 6386 WGS UNIX System V/386 Release 3.1 and 3.2.

7.1.2 Microstructure of Process Creation

When the *fork* system call is first executed, the UNIX system kernel ascertains that it has available resources to complete the *fork* successfully. Since there may not be enough memory available to do the replication in memory, the *fork* system call must be capable of making the copy on the swap area.

Figure 29a and Figure 29b illustrate both cases from the standpoint of the available memory and the *age* of the newly created process. For the case where there is enough available memory to do the replication, the first piece of available memory is allocated and a copy is made in memory. (Since the system is nonreentrant, the process doing the *fork* cannot be preempted until the *fork* is complete). Note that from a queuing standpoint, the newly created process (α') joins the end of the queue to leave memory. This means that from the *scheduler's* standpoint the *child* is allowed to remain in memory. The *parent* (α) however has aged during the replication and is more eligible to be swapped than the child, since the newly created child is created with *age* zero. For the case where there is not enough memory to do the *fork* in memory, the replication is done by locking the *parent* in memory and swapping out a copy of the *parent*. In this case, the *child* joins the end of the queue (on the swap area) of processes that want to be brought into memory. It should also be noted that in both cases, the *parent* and *child* end up in the *ready* state after the *fork*.

7.1.3 Resources Allocation Mechanism

During the execution of the *fork* system call, the UNIX system kernel finds an empty slot in the process table in order to start constructing the context of the child process and makes sure that the user doing the *fork* does not have too many processes already running. (When full, the *fork(2)* system call returns the error number EAGAIN[19]). It also picks a unique ID number for the new process, one greater than the most recently assigned ID number. (When the ID numbers reach a maximum value, assignments start from 0 again.)

The UNIX system imposes a configurable[20] limit on the number of processes a user can simultaneously execute so that no user can steal many process table slots, thereby preventing other users from creating new processes. (For this research the limit was set to 200, in order to collect more mean values.) Ordinary users cannot create a process that would occupy the last remaining slot in the process table, or else the system would deadlock. (The data profile gathered from the UNIX system was running as user and superuser.) The UNIX system kernel cannot guarantee that existing processes *exit* naturally and, therefore, no new processes can be created, because all the process table slots are in use. On the other hand, a superuser can execute as many processes as it likes, bounded by the size of the process table. (Presumably, a superuser could take drastic action and spawn a process that forces other processes to *exit* if necessary).

The UNIX kernel next initializes the child's process table slot, copying various fields from the parent slot. For instance, the child "inherits" the parent's real and effective user ID numbers, the parent's process group, and the parent's *nice* value, used for calculation of *scheduling* priority (see *scheduling* section). The kernel assigns the parent process ID field in the child slot, putting the child in the process tree structure, and initializes various *scheduling* parameters such as the initial priority value, initial CPU usage, and other timing fields. (The initial state of the process is "being created". See Figure 1).

The kernel now adjusts reference counts for files with which the child process is automatically associated. First, the child process resides in the current directory of the parent process. The number of processes that currently access the directory

increases by 1 and, accordingly, the kernel increments its inode reference count. Second, if the parent process or one of its ancestors has ever executed the *chroot(2)* system call to change its root, the child process inherits the changed root and increments its inode reference count. Finally, the kernel searches the parent's user file descriptor table for open files, but it also shares access to the files with the parent process (both processes manipulate the same table entries).

The UNIX kernel is now ready to create the user-level context of the child process. It allocates memory for the child *u block*, regions, and auxiliary page tables, duplicates every region in the parent process, and attaches every region to the child process.

7.1.4 Process Creation

So far, the UNIX kernel has created the static portion of the child context; now it creates the dynamic portion. The UNIX kernel copies the parent context layer 1 containing the user saved register context and the kernel stack frame of the *fork* system call. Since the kernel stack is part of the *u block*, the kernel automatically creates the child kernel stack when it creates the child *u block* (at this point the kernel stacks for the parent and child processes are identical). The kernel then creates a dummy context layer (2) for the child process, containing the saved register context for context layer (1). It sets the program counter and other registers in the saved register context so that it can "restore" the child context, even though it had never executed before, and so that the child process can recognize itself as the child when it runs. (For instance, if the kernel code tests the value of register 0 to decide if the process is the parent or the child, it writes the

appropriate value in the child saved register context in layer 1.)

When the child context is ready, the parent completes its part of *fork* by changing the child state to "ready to run (in memory)" and by returning the child process ID (PID) to the user. The kernel later schedules the child process for execution via the normal *Scheduling Algorithm*, and the child process "completes" its part of the *fork* system call.

7.1.5 Research Improvements To The Process Creation Scheme

The *fork* system call source code was examined in detail to see what algorithmic or data structure changes could be made to improve performance with the following results.

7.1.5.1 Stack Copy

In order to set up the user context, the kernel copies the parent's *u block* and kernel stack to the child process. This is accomplish by copying the entire 6K *u block*. This is unnecessary since most of the space in the *u block* is room for the kernel stack to grow into and is unused during a *fork* system call (see Figure 29c). The *u block* structure itself is 1764 bytes followed by a few words for the user file descriptors and by the kernel stack which is only a few hundred bytes deep (during a *fork* system call). So the amount of data that has to be copied is only around 2K bytes, a third of the total *u block*. This suggested modification improves the process creation by 20%, since less data has to be copied.

7.1.5.2 Page Table Allocation

Examination of the code shows that the kernel allocates inside loops (page table allocator) new segment tables, page tables, and *r_list* array for the child process. In particular the child needs:

- two segment tables, one for section 2 and one for section 3;
- one page table for the data region and one for the stack region;
- an *r_list* array for both the data and stack regions.

The profiling of the *fork* code shows that the page table allocator routine is called six times per *fork*. (About 21% of the *fork* time is spent in the page table allocator routine. Larger processes require more page tables and may required more calls to allocate additional space.)

The code for the page table allocator maintains free page tables using a linked list and a bitmap. The bitmap makes it easier to allocate the physically contiguous page tables that are needed for segment tables, page tables and disk block descriptor (DBD) pairs, and *r_list* arrays. A single page table is 64 words, so there are eight per page. The linked list links together pages that contain free page tables, and the eight bits in the bitmap for each page show the locations of the free page tables (see Figure 29d). *Profiling* shows that only very large processes ever have the need for more than a few contiguous page tables. Segment tables typically use only three contiguous page tables. This makes it seem likely that an allocation algorithm that optimizes for the most frequent cases of one, two and three contiguous page tables would help.

Unfortunately it was noticed about 60% of the time of the page table allocation algorithm is spent in clearing the newly allocated page tables. There is no optimization to be done here since the page tables must be cleared to ensure that all valid bits are off. Also while clearing the newly allocated page tables, an optimization for word aligned data areas (like page tables) is already implemented as part of the standard UNIX System V/386 Release 3.1 and 3.2.

Of the remaining 40%, an allocation algorithm that optimizes for small page table sizes might cut this time in half at best or about 4% of the entire fork. Such a small savings did not seem to warrant the added complexity in the code, especially since there is no guarantee of improvement, and so the author did not alter this portion of the code.

7.2 Expansions

More memory is required by a process if it overruns the stack space that is allocated or if the process wishes to dynamically allocate more program space. Rather than wait for adjacent memory to become free, the strategy used in growing the size of a process is to allocate a new piece of contiguous memory equal to the existing size of the process plus the desired increment. If this space is available, then the process is copied into the new area and adjustments are made for virtual addressing (e.g., the stack must reside in the bottom of the new area, so the stack must be shifted to a new position). The old area occupied by the process is then abandoned. During the copy operation, the process doing the expansion (see α in Figure 30) cannot be preempted. This operation is similar to the *fork* operation

and, as can be seen from Figure 30, the process retains its age and may be more susceptible to being swapped because it has aged while doing the expansion.

If there is not enough available memory, the expansion is done by swapping a copy of the process out into an area equal to the old size of the process plus the increment of memory that the process needs. When the *scheduler* brings the process into memory at a later time, the virtual address space is adjusted as discussed above. Note that under these circumstances the expanded process (α, see Figure 30) has joined the end of the queue of processes that want to be brought into memory. The process doing the expansion (α) is locked in memory until the swap is complete, after which the system can free the memory that was occupied by α. (Unlike the in-memory copy, the processor is free to run another process while the swap occurs).

A process may want to have a negative expansion, that is, to contract its size. Decreasing the size of a process is a simple matter which requires only the release of the unneeded memory to the system.

It should be emphasized that since **no** attempt is made to find adjacent space, even when memory adjacent to the expanding process is available, there is a minimum time expense involved in doing the *in memory* copy. If it is assumed that memory references take 1 μsec. and therefore an instruction takes about 3 μsec. to copy one word to another, then the cost of expanding just the *u block* (512 words) is 1.536 msec. There is overhead associated with moving each 32 word memory block so that this is a very conservative estimate of the minimum cost of expanding a process's size in this manner. Memory expansions are done a number of times in

performing an *exec* and the *fork* operation is similar to an expansion so that this time cost should be remembered and probably should be more accurately measured.

7.3 Process Termination

The final state that a process enters before leaving the system is the *zombie* state. The process remains in this state until the parent process finds the dead (*zombie*) child and disposes of it. Currently a *zombie* process consists only of the first 512 bytes of the *u block* containing process information. The remainder of the process, including reentrant text, is discarded. (The reentrant portion is really only discarded when the last process in the system using the text terminates. On the research version of the UNIX system, special arrangements have been made so that often used programs such as the loader, assembler, compiler, never have their reentrant parts discarded when they terminate. This creates problems in installing new versions of these programs in the system, but saves time when executing these programs.)

To become a *zombie*, a process must make a copy of the first 512 bytes of the *u block* on the swap area (see Figure 31). It does this by doing a buffered write onto the swap device and waiting until the write has completed. That is, the termination procedure is suspended until the buffer is written onto the swap area. Only when the 512 byte block has reached the disk is the memory occupied by the process released. Most of the memory occupied by the process could have been released to the system before the *u block* was written, but this would require the

release of two memory fragments and might increase memory fragmentation.

Currently, when allocating swap space for a *zombie*, the system overallocates the space needed to hold the *zombie*. That is, the *zombie* fits in one 512 byte disk block; however, the system allocates eight blocks on the swap area to hold the *zombie*. Changing this to allocate exactly one memory block has aroused a certain amount of debate since the minimum program size is 1764 + 1280 = 3044 bytes and allocating only one block means that only another *zombie* or a very small reentrant text could reuse the space. It is not known whether decreasing the space allocated for a *zombie* would increase the fragmentation on the swap area or not, so currently *zombies* are still overallocated on the swap area.

7.3.1 Research Improvements To Process Termination

Since the only known reason for saving the *u block* belonging to a *zombie* is so that the parent of the *zombie* can retrieve the exit status of the process and the amount of CPU time that the process has used, the research UNIX system version has completely eliminated the need for saving the *u block* by overlaying part of the Process Table entry with this information. The entries which must be kept in the Process Table entry include the cumulative user execution time (2 words) and the exit status (1 word). This new scheme allows the quick release of the memory occupied by the process, eliminates unnecessary I/O to the swap device and, since these delays have been eliminated, the parent of the deceased process can be immediately notified of the child process's death.

This modification to the process termination scheme improved the release of the

occupied resources an average of 21%. The memory deallocation process is not affected by this modification. As a direct result of the modifications implemented in the process creation, there is now less memory to be deallocated and the process creation is likely to create a new process in memory rather than on the swap device (disk queue).

The most significant advantage of this modification is the elimination of the need to use the swap device as a message queue as the parent process waits for the child's exit status.

7.4 Exec

The *exec* system call allows one program to overlay itself with another. The *fork* mechanism described earlier creates a *new* process in the system, but it does this by making a copy of an *existing* process. The *exec* system call actually assembles the reentrant and nonreentrant parts of a program from the file system so that it can be executed. (For want of a better term, we use the term *assemble* in discussing *exec* to mean the operation of reading parts of the file containing the program into memory). Since overlaying nonreentrant processes is a special case of overlaying a reentrant process, the following discussion centers on overlaying reentrant processes and only points out the steps that are skipped when overlaying a nonreentrant processes. In the following discussion, the transformation of a process α into a process β by an *exec* (overlay) is traced. The various phases of this transformation are shown in Figure 33 and 34. The encircled number over each phase is used for referencing (i.e., step 1, etc.) and the subscripts U, T and D will

refer to the *u block*, reentrant part and nonreentrant part of the program respectively.

When a process overlays itself with another program, it usually passes several (ascii) arguments to the program. (The system does this by copying the argument list into one 512 byte system buffer and saving the contents of the buffer until the overlay is completed at which time the arguments can be passed to the program. In the discussion that follows, it should be remembered that this buffer remains allocated until the *exec* call is completed). Once the arguments have been saved, the process disembowels itself by discarding the space occupied by the program with the exception of the (entire) *u block* (step 2 Figure 33). In the accompanying diagram, a process α is shown to be composed of three parts - reentrant (α_T), nonreentrant (α_D) and *u block* (α_U). Thus, the only part of the program left is (α_U).

The next step in loading a reentrant process is to assemble the reentrant part (if any) of the process into one contiguous piece from the file system. A new piece of contiguous memory must be allocated to do this (Figure 33 step 3). The piece allocated is equal to the size of the *u block* plus the amount of memory required to assemble the text. Note that in Figure 33 the position of the *u block* (α_U) has shifted. This occurs because a memory expansion must be done to increase the size of the process and as discussed earlier, the expansion involves abandoning the old memory and copying the *u block* (1764 bytes) into a new area, if there is available memory, or being swapped out, if there is not enough memory. Also note that only enough memory to assemble the reentrant part is requested. (The reason for this is

that, as described below, the *exec* forces the process to be ejected from memory to avoid memory deadlock problems). The text is assembled by doing 512 byte buffered reads (using read ahead) of the file system to assemble the program into the space (Figure 33 step 3). This is followed by making a copy of the assembled on the swap area (Figure 33 step 4). The text is locked in memory so that it cannot be swapped while the swap I/O occurs. (The text is written by issuing one write. Until this point, the process took its turn sharing the use of the processor and might possibly even have been swapped while the I/O to assemble the program occurred).

At this point, rather than begin to assemble the nonreentrant part of the program, the process gives up all of the memory that it is using (Figure 33 step 5). The area occupied by the reentrant text is discarded (a copy now exists on the swap area), but the *u block* must be swapped out (Figure 33 step 6). To do this, the memory occupied by the *u block* is locked in memory while the swap occurs.

It may seem strange to abandon the *exec* after having gotten so far, but the following should be considered,

1. When the reentrant text is read into memory, it ends up in the wrong position. That is, it is physically contiguous to and below the *u block*. This occurs because in loading the text, the system pretended that the reentrant part was a nonreentrant region and read it in below the text. This was done for convenience in using existing system functions. By giving up at this point, the *scheduler* swaps the reentrant text and *u block* back into the correct position.

2. Memory deadlocks are a problem in implementing a better strategy.

When the process is again brought into memory, the text and *u block* are in the proper order (Figure 34 step 9) and the assembly of the nonreentrant part may begin. More memory must be allocated for the nonreentrant part (Figure 34 step 10, a memory expansion is done)[10]. This expansion may result in the process being swapped out or, at the very least, the *u block* must be copied to the new larger area. In any event, memory is eventually available and the assembly of the nonreentrant part of the program can begin (Figure 34 step 10). As with the assembly of the reentrant text, 512 byte buffered reads (with read ahead) are performed.

As a final step, the *u block* (α_U) has some transformations performed on it before the overlay is completed (and the *u block* becomes β_U - Figure 34 step 11). (The system buffer containing the arguments can now have the arguments copied into the program and finally the buffer can be released).

The above procedure is used to load a reentrant process the first time it is invoked. All of the steps required to assemble the reentrant text are skipped (Figures 33 and 34 steps 3 - 8) if a copy of the reentrant text is already within the system. For nonreentrant programs, these steps are also skipped. In addition, for nonreentrant programs, only one swap is required by the *scheduler* to bring a process into memory since there is no associated reentrant text.

---

10. This situation could have been avoided if the system had changed the size of the process to indicate to the *scheduler* that a larger area than the *u block* was required to bring the process in.

In summary nonreentrant processes are loaded in a few short steps where the program is assembled from the file system. However, reentrant processes have the advantage that the reentrant text portion need not be assembled if there is already an instance of the text in the system. Reentrant processes pay a penalty the first time that they are invoked as two assemblies are required, one for the reentrant part and one for the nonreentrant part, however, this penalty should be more than made up for by the decrease in memory requirements of processes sharing reentrant text. Reentrant processes actually pay a greater penalty than nonreentrant processes when being invoked if they are not eventually shared by another process[11].

7.4.1 Research Improvements To Exec

Some work was done to the *exec* procedure to improve its performance for reentrant processes. Figure 35 diagrams the new method of operation. The description of the new algorithm parallels that of the current scheme and centers around how a reentrant process is overlaid.

The first step in the *exec* process is to save the arguments from the process that is to be overlaid. Under the current method, arguments are copied into one system buffer, thus limiting the arguments to 512 bytes. The new scheme allocates space (10 - 512 byte blocks) on the swap area to hold arguments and only uses a system buffer temporarily to collect arguments before they are written (asynchronously) on

11 In the research version of the UNIX system, often used programs are marked so that their reentrant text never leaves the swap area to avoid paying this penalty (See Appendix III)

the swap area.

The next step is to discard any reentrant text associated with the process (step 1). The nonreentrant part of the process is not discarded as in the current scheme. Rather, a memory expansion is done to get enough space for the new nonreentrant part of the process and the area is cleared (step 2). Figure 35 shows that an in memory copy to a new area is done, however, if the new nonreentrant data (β_D) is smaller than the old nonreentrant data (α_D) then no copy is done. The extra space need only be released.

Once space is allocated and cleared for the nonreentrant part, space for the reentrant text is allocated. Assuming that the reentrant text must be assembled, space is allocated for it. (If there is no space available, the entire process is swapped out to be brought in later by the *scheduler* when memory is available.) This process is then locked in memory so that the reentrant text can be read from the file system and assembled into the space without permitting the process to be swapped out. Once this is accomplished, the nonreentrant part of the program can be read in. (Note that while the nonreentrant part of the program is assembled from the file system, it is not necessary to lock the program in memory.) As a final step (Figure 35 step 6), some transformations are made on the *u block* and the arguments to the program are copied into the stack from the swap area. (Under the current scheme, a fixed amount of space, 20 blocks, is allocated as the initial stack increment and to contain the maximum of 512 bytes of arguments and pointers to arguments. The new scheme allocates 20 memory blocks in addition to any space needed for arguments.) Note that no copy of the reentrant text is made

on the swap device. Rather, the reentrant text is marked (i.e., its Text Table entry is marked) so that the first time that the text must be swapped out, a copy of the reentrant text is left on the swap area. Note also that the entire program is locked in memory while the reentrant text is assembled from the file. This is done to prevent deadlocks and quickly to make the reentrant text available for sharing.

7.5 The Scheduler's Allocation Of Memory

In allocating space for a nonreentrant process to be swapped into memory, the *scheduler* requests a piece of contiguous memory large enough to hold the process (i.e., text, data, bss, stack and *u block*). For reentrant processes, the *scheduler* need only request space for the nonreentrant part (data, bss, stack, and *u block*) if there is a copy of the reentrant part in memory.

When allocating space for a process whose reentrant part is not already in memory, the *scheduler* is overly *pessimistic* in its request for space to bring the process in. The *scheduler* requests enough memory to fit the reentrant part and nonreentrant parts *contiguously*. This is done even though the reentrant part of a process does not have to be contiguous in physical memory with the nonreentrant part. The *scheduler* could make two requests, each big enough to fit one piece. This, however, poses some problems as it is possible that the first memory request might be satisfied but the second denied. The *scheduler* would then have to deallocate the memory it first requested and take action to free more memory. As discussed in the following section of *scheduling improvements*, some work has been done on the research version of the UNIX system to remedy this problem, however, under

the UNIX System V/386 Release 3.1 and 3.2 the *scheduler* simply makes one request for enough memory to fit the entire process contiguously.

Swapping a reentrant process into memory is done in two steps. The reentrant text is first swapped into memory, followed by the nonreentrant part (see Figure 32). Two requests are necessary because reentrant text is managed separately on the swap area from the nonreentrant part.

7.5.1 Research Improvements To The Scheduler's Allocation Of Memory

In line with the discussion of the previous section on the *scheduler's* allocation of memory, a scheme has been implemented on the research UNIX system, which allows the *scheduler* to allocate memory for a reentrant process in two separate pieces rather than requiring a one piece contiguous allocation. The scheme requires that a lock and unlock mechanism for the reentrant text exist to prevent attempts to use text that is being swapped out. The research *scheduler* allocates space for the nonreentrant part of the program, followed by allocation of space for the reentrant part (if needed) and if the second allocation fails, the *scheduler* can relinquish all of the memory that it had allocated and start to look for processes to remove from memory to create space. While space for the nonreentrant part of the process is allocated before the reentrant part, the reentrant part is swapped in first (if necessary) before the nonreentrant part. This done to make the text available for sharing with other processes as soon as possible.

8. SWAP TIME

Rotational device controllers available for the AT&T 6386 WGS line of computers are very simple in structure. Only one transfer can be initiated at a time. There is no command list chaining which allows the controller to execute a series of I/O operations without intervention by the CPU as with device controllers on larger machines (the AT&T 6386 WGS utilized for this research was configured with a ESDI HDU controller with a 135 Mega-Byte hard disk drive). This is probably the major reason that processes are set up as contiguous entities in memory rather than making an attempt to efficiently allocate memory by breaking a process into small pieces (*paging*). Dividing a process into a number of smaller pieces would require the intervention of the CPU to swap out each piece.

Since processes are loaded contiguously in memory, it is useful to determine what the maximum *traverse* time to or from the swap device can be for several types of disks and several program sizes.

To make this calculation, it is necessary to understand the way the recording medium is set up on PC's disks. Rotational media are organized in concentric tracks. On larger moving head disks, there are a number of surfaces on which to record, so tracks are organized as cylinders. For this type of organization, there is one head for each surface so that when switching between surfaces there is only a very short electronic switching time to change to a different head. Each track is organized into a number of sectors (most commonly 512 bytes per sector. PC's hard disk controllers are set up to automatically reposition disk heads to the next track. This is called a *spiral* read or write operation. Unfortunately, there is a

finite amount of time required to move and set up the disk heads even when moving to adjacent tracks. This called the *single track move* time. Thus, when one track is finished being accessed during a spiral operation, the controller must wait a minimum of this *single track move* before continuing (see Figure 36). The situation is slightly more complicated than this, however, since during a spiral operation, the controller accesses sectors sequentially (i.e., sector 0, 1, 2, 3, etc. on one track followed by sector 0, 1, 2, etc. on the next track). On a spiral operation, the controller could continue to access the sectors with only the *single track move* delay if the pack were formatted so that the starting sector on adjacent tracks were offset by the *single track move time*. This means for spiral operations, that the controller must wait at least one full *rotational time* before continuing with the access.

Another factor that must be taken into account when calculating the swap time is the amount of data that actually must be read or written onto the disk. This should include any formatting information that must be read or written on each physical sector as well as the data transferred.

To account for all of these factors, it is best to calculate all times in terms of the fraction of the disk that must be read and the rotation time. The total swap time can be calculated as follows,

$$T_s = T_r + T_p + T_t$$

where,

$$T_t = \frac{N}{N_{SC}} T_m + N * T_e$$

$$T_t = \frac{N}{N_{SC}} T_{fr} + \frac{N}{N_{ST}} T_{fr}$$

and

T_s = *swap time*
T_p = *average positional time*
T_t = *transfer time*
T_e = *time read one 512 byte logical sector*
T_{fr} = *full rotational time*
T_r = *average rotational time* = $\frac{1}{2} T_{fr}$
T_m = *single track move time.*
N = *Number of (512 byte) sectors be transfered.*
N_{SC} = *Number of sectors/cylinder.*
N_{ST} = *Number of sectors/track.*

The above formulae give the swap time in terms of the rotational times. The first fraction (e.g., N/N_{SC}) is rounded down to its integer value before multiplying by the rotational time. The above formulation can be in error by one full rotational time since the starting sector for the transfer may affect the number of single track moves required.

For fixed head disks, the above formulation can be used with T_p and T_m set to zero since there is one set of heads for each track. The following table contains the *traverse time* in milliseconds for a program size of 32.5 Kwords and for a program size of 4 Kwords

Transfer Time			
Position	Rotation	4 Kwords	32.5 Kwords
0	8.5	17	138
0	16.7	68	550

The positional and rotational times shown are average values that were monitored from *io_act* and *io_resp* defined in the *elog* data structure in */usr/include/sys/elog.h*. The above parameters measure the active time and response time of a device in time ticks. The device active time includes the device seek, rotate, and data transfer times[21], while the response time of an I/O operation is from the time the I/O request is queued to the device to the time the I/O completes. (The hard disk utilized was a MiniScribe 135 Megabytes full height ESDI with an access time of 18 ms, with 1:1 inter-leave and 10 Mbit per second transfer rate[22]).

9. TRAFFIC CONTROL

Basically, the *Traffic Controller* is responsible for the orderly sharing of the processor and sequencing of processes through their various states[23]. Within the UNIX operating system, there are several primitives used for the orderly sharing of the processor. The first of these,

sleep(event, priority)

allows a process to relinquish the processor until some *event* occurs. The value *priority* is the software priority that the process receives when the process is awakened. It is a measure of the urgency which the system attaches to the servicing of that process once *event* occurs. The occurrence of an *event* is signified by the following primitive,

wakeup(event)

As discussed earlier, the value of *priority* also determines the state that the process enters when it relinquishes the CPU. Negative values of *priority* (high priority) result in the process being placed in the *sleep* state, while positive values of *priority* result in the being placed in the *wait* state.

In addition to these synchronization primitives, there is a more basic primitive that can be used for unconditionally relinquishing the processor,

swtch()

This results in a process relinquishing the processor with no change of state (i.e.,

the process remains *ready*). It is used in the implementation of certain system call to give up the processor if the process requires a resource which is not yet available (e.g., memory) and is also used by the system to preempt CPU bound processes to enforce the one second time quantum limit. The sleep-wakeup primitives are responsible for most of the changes of state and priority that a process goes through. The bookkeeping of this state and priority information is done in the Process Table.

10. OTHER RESEARCH MODIFICATIONS TO THE UNIX SYSTEM

Besides the improvements discussed in earlier sections, it was necessary to make other changes to the research UNIX system in order to make it functional. The most significant of these changes are briefly summarized in the following sections. It should be noted that while many of the changes discussed throughout this research paper improve the algorithms used for the implementation of certain functions, they do not affect the basic structure of the UNIX system. However given the monolithic structure of the UNIX system, some of the research modifications triggered the need for changes elsewhere as side effects.

10.1 Switcher Efficiency

In searching for a process to run, the *switcher* examines all of the entries in the Process Table, used or not, to find the *ready* process with the highest priority. This requires a considerable amount of time which may be wasted if the *switcher* is run every time *any* process awakens. In an attempt to minimize this time, the wakeup function has been modified so that if a process is awakened as the result of some interrupt (or system call), the running process is preempted (i.e., the process *switcher* invoked) only if the priority of the awakened process is higher than that of the currently running process. In order to implement this, a minor change was made to the process *switcher* to remember the priority and *identity* (PID) of the currently executing process.

Another change that should be made (it was not made during this research,

because the implications are outside the intended scope of this research. It should be considered as a path for future research.) to reduce the time necessary to switch between processes is to keep track of the last entry used in the Process Table. In this manner, the search for a new process could be terminated early instead of examining all of the entries unused or not. This change would allow a number of other linear searches of the Process Table, made by other UNIX system functions, to be terminated early.

10.2 Tracing And Idle States

Figure 37 shows the changes that have been made to the process states. The *idle* state has been part of the system for some time but was not discussed earlier to avoid confusion. It can be an intermediate state in the creation of a new process using the *fork* system call when there is not enough contiguous memory available to do the replication in memory. (Under these circumstances, to create the child, a copy of the parent must be swapped out.) The simplest means of doing this is for the child to assume the identity (location and size of the process) of the parent and swap out a copy of itself. During the swap, both the parent and child must be removed from consideration by the *scheduler* and *switcher*. The child is *locked* in memory and goes into the *sleep* state since it has assumed the parent's identity to do the swap. The *idle* state is created to remove the parent from candidacy to be swapped out and to identify it clearly as the parent. (The **id** of the parent of a process is kept by each process in the system so that their relationship can be determined.) When the child finishes swapping out a copy of the parent, the child makes the parent *ready* before again interchanging identities.

11. CONCLUSION

It is difficult to provide an up to date description of an operating system like the UNIX system, which is so easy to change since it is written in a high level language. The task is made more difficult, because there are a large number of projects using the UNIX system as the operating system for their application and which may or may not have added updates to their system before they are distributed. This dissertation has discussed the operation of the **UNIX System V/386 Release 3.2** as implemented in the **AT&T 6386 WGS - Intel 80386 processor based**, but also has described in detail the modifications which were fully incorporated and implemented as result of the above described research effort.

Some of the modifications made to the UNIX system kernel (kernel I/O and OS) might produce different results if implemented on a more complex computer architecture. However a great deal of effort was spent in order to produce modifications that were computer architecture and resource independent. The author believes that most of the research modifications can be generalized regardless of the computer architecture that the UNIX system is based on, resulting in a performance improvements.

The original size in bytes units of the UNIX system kernel as distributed by AT&T is for the *os* resident in */etc/conf/pack.d/kernel/os.o* 238454 bytes and for the */unix* is 415237 bytes. For the research version of the UNIX system (product of this effort), the sizes were 262776 and 441907 bytes respectively. Both UNIX System V/386 versions included the drivers for *Simul-Task 986**** Package*

Version 2.0, *FMLI Version 1.2* and the *WEITEK 1167 Coprocessor* code (advanced floating point for the PC).

The contrivances used for the explanations throughout are the author's and hence any inaccuracies or deficiencies are the author's responsibility.

**** Trademark of Phoenix Technologies Ltd and INTERACTIVE Systems Corporation

12. APPENDIX I

Value	Pseudonym	Events at this Priority
0	PSWP	1. The scheduler is roadblocked at this priority when it has run out of processes on the swap area that are *ready* to execute. This amounts to having no processes worth swapping in. 2. The *scheduler* also is roadblocked at this priority when it has *ready* processes on the swap area, but there is not enough contiguous memory available to bring in the oldest *ready* process on the swap area. 3. A process using the swap buffer** (to swap itself out or to swap out a copy of itself) is roadblocked at this priority until the swap is completed. 4. Any process requesting the use of the swap buffer** when

** NOTE The processes requesting use of the swap area are

a The *scheduler* when it swaps a process in or out of memory

b A process calling *exec(2)* to bring in a process that is sharable but which the reentrant text is not currently in memory The text is swapped in before continuing with the *exec*

c A process which executes a *fork(2)* system call when a contiguous piece of memory large enough to do a replication of the process in memory is not available

d Anything requiring an expansion of a process when there is not enough memory available to fulfill the request For example,

1 Dynamic storage allocation (i e , *malloc(3C)* system call)

2 Execution of a *exec* of a process larger than the overlaid process

3 Request for more stack space (i e , when a stack violation occurs)

the swap buffer is busy is roadblocked at this priority.

10 PINOD

1. Processes that are roadblocked because they want to remove or replace a block number on the *Superblock's* freelist, but the list is locked.

2. Processes requesting an i-node number from the *Superblocks* list of free i-nodes when the list is locked***.

3. Processes attempting to access an *i-node table* entry while the entry is locked. Access to an i-node is required by a number of system calls (e.g., create, open, link, unlink, chmod, chown, mknod, etc.). Locking an Inode Table entry occurs for several reasons.

a. The i-node may be allocated as a pipe. Pipes are locked while being read or written by processes.

b. The update process is running. This process locks the *i-node table* entry momentarily, preventing access while the corresponding i-node on the file system is updated.

c. The *i-node table* entry is locked when some other process is accessing the *i-node table* entry.

*** **NOTE**: This condition may result in a situation where processes roadblocked on this event are delayed a long time (on the order of seconds) if the list is emptied. The reason is that the system must do a linear search of the *i-list* area to find 100 free i-nodes (or as many as are left). On large file system the *i-list* area is correspondingly large and may require the reading of thousands of blocks

20 PRIBIO

1. A process that requests a buffer from the free list of buffers (*bfreelist*) when none is available is roadblocked at this priority.

2. A process requesting a buffer that is already in memory on a device's queue, but is busy (doing I/O) is roadblocked at this priority until the buffer is available.

3. When requesting a read or write operation on a block device, a process is roadblocked at this priority until the I/O completes. All read operations issued when a desired block is not in memory result in a roadblock of a process at this priority. Also, write operations that must be done immediately (cannot be delayed or done asynchronously) roadblock at this priority. Updating i-nodes and the *Superblock* require write operations that cannot be delayed.

4. When a process request physical I/O, the process is locked in memory and is roadblocked at this priority until the I/O completes. The process is unlocked when the I/O completes.

5. Since there is only one physical I/O buffer header for each block device type, then when there are several requests to do physical I/O to one device, only one process is allowed to use the header while the other processes are roadblocked at this priority until the buffer header is available.

26 PPIPE

1. During the reading or writing of a pipe,** the file which implements the pipe is locked. If one process attempts to read the pipe while another is in the process of writing into the pipe, the process attempting to read the pipe is roadblocked at this priority.

2. If one process wants to read a pipe,** but the pipe is empty, the process attempting the read is roadblocked at this priority.

3. If a process wants to write into a pipe,** but the pipe is full, the process attempting the write is roadblocked. The maximum size of a pipe is 4096 bytes. The process attempting to write into the pipe remains roadblocked until the process reading the pipe completely empties the pipe.

28 TTIPRI

In general, processes are roadblocked at this priority when are requesting *input* from the serial I/O.

1. If a process attempts to open the serial I/O and carrier is not present, the process is roadblocked at this priority. This is the basis of operation of the *init* process. On a system with several serial I/O ports, all of the *inits* spawned by the parent *init* (process one in the process table) are roadblocked at this priority when the child *init* attempts to *open* the device representing the serial I/O (*/dev/ttyn*). When the serial I/O driver (*asy.c*) detects carrier, it sends an awaken signal and the

open(2) system call returns.

2. If a process attempts to *read* any character from the serial I/O and either of the following two conditions prevail the process is roadblocked.

a. The serial I/O port in question is set up for line at processing time (normal mode) and the new line ("0) character has not been received. To paraphrase, processes waiting for characters from a */dev/ttyn* set up for line at a time processing are awakened only when a new line character is sent.

b. The serial I/O port is set up for raw mode character processing but no characters have been received from the *asy.c* driver. (Raw mode processing means that all characters typed on CRT are reflected to a user process immediately without waiting for the new line character).

29 TTOPRI In general, processes that are roadblocked at this priority want to *write* to a tty port.

1. If a process wants to *output* characters to a tty port and the output queue for that tty port is above a certain high water mark, then the process is roadblocked. The process is awakened by the tty port transmit interrupt handler, when the

output queue has reached a certain low water mark.

2. Whenever a serial I/O port is closed by a process, the process is roadblocked until the characters remaining in the tty port output queue have been output to the tty port. The closing may be the result of explicitly closing the tty port special file or an implicit close as the result of a process terminating. If carrier is lost, of course, the output queue is flushed and no waiting occurs.

30 PWAIT Processes that are *waiting* (see wait(2) UNIX System Programmer Reference Manual) for the death of a *child* do so at this priority. A process terminates as the result of doing an *exit* or receiving a *signal*, which it is not prepared to handle. In any event, the process becomes a *zombie* and the parent is awakened. If the parent is awakened and a child has not died, the parent is roadblocked at this priority again.

39 PSLEEP All processes that use the *sleep* system call (see sleep(2) UNIX System Programmer Reference Manual) to delay execution are roadblocked at this priority. Currently there is only one event for use in synchronizing delayed execution so that *all* processes delaying execution are awakened when the process requesting the least delay is awakened. The system then redetermines the

least delay and roadblocks all processes requesting delayed execution.

60 PUSER Most processes that are *ready* to execute have this priority.

13. APPENDIX II

This Appendix depicts some of the approaches utilized during this research to measure performance management. It describes a general procedure for troubleshooting performance problems utilizing the UNIX system's tools that proved to be very useful and reliable. This Appendix does not present a canned procedure, but rather a sample of a typical approach, covering basic areas and suggesting some of the actions that were taken in order to solve the problem.

One of the analyzed factors during this research was the swapping activity, since the swapping of pages is costly in both disk and CPU overhead. The *sar(1) -qw* table depicts an example of the output of this command. This output indicates the percentage that the swap queue is occupied and the swap out rate. In order to detect the number of pages of memory available to user programs or free memory the command *sar(1) -r* was utilized (see table **sar(1) -r**), also this information was utilized in order to compare it to the tunable parameter *GPGSHI* (high-water mark). The number of buffers and the buffer cache hit ratios were monitored with the command *sar(1) -b* (see table **sar(1) -b**). For hard disk bottlenecks the commands *sar(1) -a*, *sar(1) -u* and *sar -d* were utilized. Also this data was utilized to manage the number of buffers as well as to set the swap space size. The serial I/O interrupts were monitored by the *sar(1) -y* command. Potential kernel table overflows were profiled by the command *sar(1) -v*. This command was also used to check for the number of process tables entries that are used or allocated to the kernel.

The monitoring of the UNIX system process control, was somewhat more

complicated. A combination of the *sar(1)* -*c* command that reports system calls per second, read system calls, write system calls, fork system calls, exec system calls occurring per second and the UNIX system profiler recording mechanism in the system (see profiler table). The combination of both tools facilitate an activity study of the UNIX operating system. (Note that 30 second sampling intervals were utilized when possible).

Intelligence ... is the faculty of making artificial objects, especially tools to make tools. - **Bergson**

Following this philosophy, all kinds of profiler ideas and code inspections were performed, including simple *printf(3S)* statements to the *console* when possible. Also the source code of the *profil(2)* and *monitor(3C)* were tailored to obtain executing time profile of different parts of the kernel in particular for the UNIX system process creation. (The *profil(2)* and *monitor(3C)* prove to be excellent tools in their own right and great tool makers).

sar -a			
time	iget/s	namei/s	dirbk/s
07:07:40	185	55	167
07:08:10	236	116	179
07:08:40	45	15	39
07:09:10	8	2	10
07:09:40	19	4	20
07:10:10	13	3	14
07:10:40	13	3	14
07:11:10	17	4	18
07:11:40	22	5	22
07:12:10	13	3	14
07:12:40	26	6	26
07:13:10	50	14	52
Average	54	19	48

sar -b								
time	bread/s	lread/s	%rcache	bwrit/s	lwrit/s	%wcache	pread/s	pwrit/s
07 07 40	16	288	95	4	13	71	0	0
07 08 10	18	389	95	2	9	73	0	0
07 08 40	20	110	82	1	10	87	0	0
07 09 10	0	17	98	1	2	75	0	0
07 09 40	1	36	97	2	9	74	5	2
07 10 10	0	25	98	1	5	75	0	0
07 10 40	0	25	99	1	4	77	0	0
07 11 10	1	40	97	1	5	79	4	2
07 11 40	1	43	97	1	6	80	0	0
07 12 10	2	31	95	1	4	77	0	0
07 12 40	3	53	95	2	9	79	0	0
07 13 10	6	108	94	7	28	75	0	0
Average	6	97	95	2	9	77	1	1

sar -c							
time	scall/s	sread/s	swrit/s	fork/s	exec/s	rchar/s	wchar/s
07:07:40	94	17	2	0.49	0.39	2889	959
07:08:10	151	15	1	0.39	0.28	2526	80
07:08:40	101	30	4	2.06	1.92	20512	859
07:09:10	20	5	1	0.32	0.23	207	13
07:09:40	63	13	4	0.40	0.23	78145	71684
07:10:10	36	6	1	0.35	0.22	1381	309
07:10:40	30	7	2	0.35	0.24	842	251
07:11:10	59	20	5	0.51	0.57	75509	70989
07:11:40	56	16	5	0.64	0.55	1046	180
07:12:10	39	10	3	0.40	0.36	1018	119
07:12:40	63	16	5	0.62	0.64	2609	580
07:13:10	217	48	17	0.80	0.63	7629	2688
Average	77	17	4	0.61	0.52	16193	12393

sar -d							
time	device	%busy	avque	r+w/s	blks/s	avwait	avserv
07:07:40	dsk-0	7	7.6	2	4	200.6	30.5
07:08:10	dsk-0	3	4.6	1	3	82.8	23.2
07:08:40	dsk-0	29	2.2	12	24	29.5	23.7
07:09:10	dsk-0	3	1.2	1	2	4.6	25.6
07:09:40	dsk-0	8	1.4	2	5	12.4	31.1
07:10:10	dsk-0	27	7.5	3	4	11.3	26.7
07:10:40	dsk-0	19	6.7	1	5	88.4	22.9
07:11:10	dsk-0	28	6.5	2	11	10.9	28.7
07:11:40	dsk-0	5	9.1	3	11	313.4	41.7
07:12:10	dsk-0	7	4.5	3	10	15.3	27.7
07:12:40	dsk-0	12	7.1	2	2	275.5	36.6
07:13:10	dsk-0	17	1.2	1	11	11.4	24.7

sar -r		
time	freemem	freeswp
07:07:40	6276	14672
07:08:10	5988	14672
07:08:40	5818	14672
07:09:10	5967	14672
07:09:40	5885	14672
07:10:10	5509	14672
07:10:40	5250	14672
07:11:10	4852	14672
07:11:40	5216	14672
07:12:10	5405	14672
07:12:40	5661	14672
07:13:10	6363	14672
Average	5683	14672

sar -qw					
time	runq-sz	%runocc	swpq-sz	%swpocc	
07:07:40	2.2	53			
07:08:10	2.5	63			
07:08:40	2.6	71			
07:09:10	2.1	29			
07:09:40	1.5	45			
07:10:10	1.3	46			
07:10:40	1.3	45			
07:11:10	1.7	51			
07:11:40	1.9	56			
07:12:10	1.6	55			
07:12:40	2.8	63			
07:13:10	1.3	47			
Average	1.9	52			
time	swpin/s	bswin/s	swpot/s	bswot/s	pswch/s
07:07:40	0.00	0.0	0.00	0.0	24
07:08:10	0.00	0.0	0.00	0.0	29
07:08:40	0.00	0.0	0.00	0.0	28
07:09:10	0.00	0.0	0.00	0.0	4
07:09:40	0.00	0.0	0.00	0.0	11
07:10:10	0.00	0.0	0.00	0.0	3
07:10:40	0.00	0.0	0.00	0.0	3
07:11:10	0.00	0.0	0.00	0.0	12
07:11:40	0.00	0.0	0.00	0.0	9
07:12:10	0.00	0.0	0.00	0.0	9
07:12:40	0.00	0.0	0.00	0.0	11
07:13:10	0.00	0.0	0.00	0.0	13
Average	0.00	0.0	0.00	0.0	13

sar -u				
time	%usr	%sys	%wio	%idle
07:07:40	6	37	16	41
07:08:10	7	44	17	32
07:08:40	39	34	5	21
07:09:10	2	5	1	92
07:09:40	7	13	23	57
07:10:10	3	7	2	88
07:10:40	3	7	1	89
07:11:10	8	15	20	57
07:11:40	7	14	2	77
07:12:10	5	10	2	82
07:12:40	12	16	3	68
07:13:10	20	37	10	33
Average	10	20	9	61

sar -y						
time	rawch/s	canch/s	outch/s	rcvin/s	xmtin/s	mdmin/s
07:07:40	0	0	0	0	0	0
07:08:10	0	0	0	0	0	0
07:08:40	0	0	0	0	0	0
07:09:10	0	0	0	0	0	0
07:09:40	0	0	5	0	5	0
07:10:10	0	0	0	0	0	0
07:10:40	1	0	5	0	0	0
07:11:10	1	0	20	0	5	0
07:11:40	1	0	68	0	1	0
07:12:10	2	0	61	0	0	0
07:12:40	2	0	40	0	0	0
07:13:10	1	0	20	0	5	0
Average	1	0	18	0	1	0

sar -v								
time	proc-sz	ov	inod-sz	ov	file-sz	ov	lock-sz	fhdr-sz
07:07:40	60/200	0	125/500	0	85/500	0	1/200	1/ 50
07:08:10	73/200	0	152/500	0	128/500	0	1/200	1/ 50
07:08:40	51/200	0	128/500	0	74/500	0	1/200	1/ 50
07:09:10	54/200	0	122/500	0	79/500	0	1/200	1/ 50
07:09:40	51/200	0	120/500	0	74/500	0	1/200	1/ 50
07:10:10	51/200	0	118/500	0	75/500	0	1/200	1/ 50
07:10:40	56/200	0	127/500	0	84/500	0	1/200	1/ 50
07:11:10	66/200	0	159/500	0	116/500	0	5/200	5/ 50
07:11:40	72/200	0	167/500	0	122/500	0	6/200	6/ 50
07:12:10	62/200	0	149/500	0	106/500	0	3/200	3/ 50
07:12:40	78/200	0	177/500	0	138/500	0	6/200	6/ 50
07:13:10	78/200	0	177/500	0	128/500	0	1/200	6/ 50

System Calls Profile	
10/23/89	13:11 - 13:21
cmntrap	0.00
sys_call	0.02
cmnint	0.06
ret_user	0.00
setjmp	0.00
spl0	0.00
spl6	0.00
spltty	0.00
spl	0.27
splhi	0.01
idle	81.18
repoutsw	0.04
repinsw	0.13
resume	0.00
bzero	0.04
bcopy	0.07
copyout	0.00
rcopyfau	0.02
lfubyte	0.00
lfuword	0.01
subyte	0.00
suword	0.00
upath	0.01
searchdi	0.02
prele	0.00
tenmicro	0.01
v86pgequ	0.01
v86vint	0.00
fdresult	0.00
hdstrate	0.01
hdstart	0.01
hdio	0.01
hdxfer	0.01
hdintr	0.01
hddone	0.01
ATdocmd	0.00
kdintr	0.00
kdscan2a	0.01
kdxfer_w	0.20
kdclrdis	0.00
kddispch	0.03
putc	0.01
getcb	0.00
user	17.14

14. APPENDIX III

This Appendix depicts some of the programs utilized during this research to measured the kernel performance. The programs described in this Appendix were selected because of their unique running characteristics (CPU usage, running length, I/O orientation, and function calls utilized). A set programs with the above described characteristics was selected and kept throughout this research.

A set of UNIX system commands with short and long response times were utilized in order to exercise the interaction of the various modules that form part of the kernel system (table maintenance, searches and updates) as well as to manipulate larges amount of data. The utilized UNIX system commands in order of execution are: *find(1), cpio(2), mkdir (1), ps(1), cmp(1), cc(1), cp(1), lp(1), make(1), lpstat(1), pr(1), rm(1), rmdir(1), pwd(1), sort(1), diff(1), kill(1), od(1), fdisk(1), nice(1).* Note that different combinations of the above described programs were utilized including redirection and indirection of output.

A set of CPU bound "C" programs that calculate *Latin Squares*, and *Truth Tables* was used. The running time of these programs was controlled in a range from 750 milliseconds to 85 minutes by modifying the size of the input matrix.

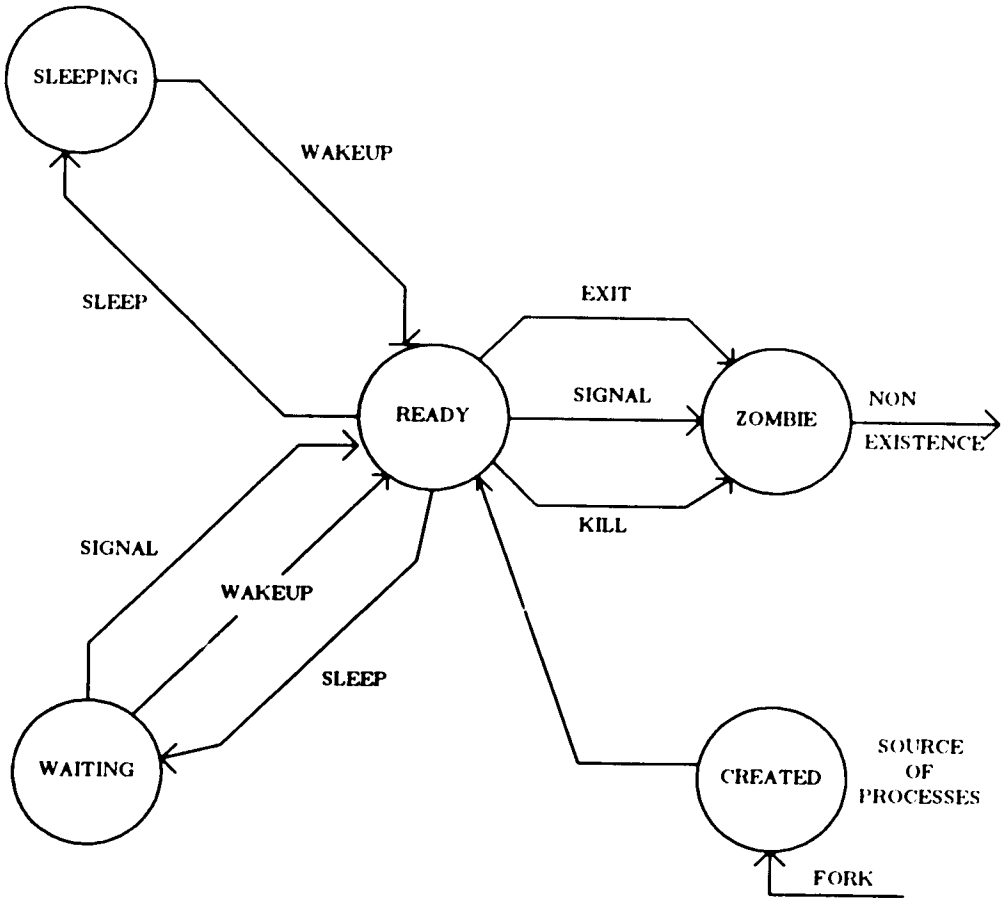
As a set of I/O bound process, a program to access a predetermined sector (usually sector one) was written. A modified version of this program that perform continuous random disk seeks was also utilized.

The above described programs were executed from a *shell* that was controlled by an input parameter for infinite loop or a controlled execution time. In addition kernel

profiles were obtained while using the editor, compiling programs. making UNIX system kernels, using the text processing system, using Simui-Task, *uucp(1)*, *mailx(1)*, etc. The deviation obtained from the profiles regardless of the user programs being executed was never greater than plus minus five percent (for the profiled modules).

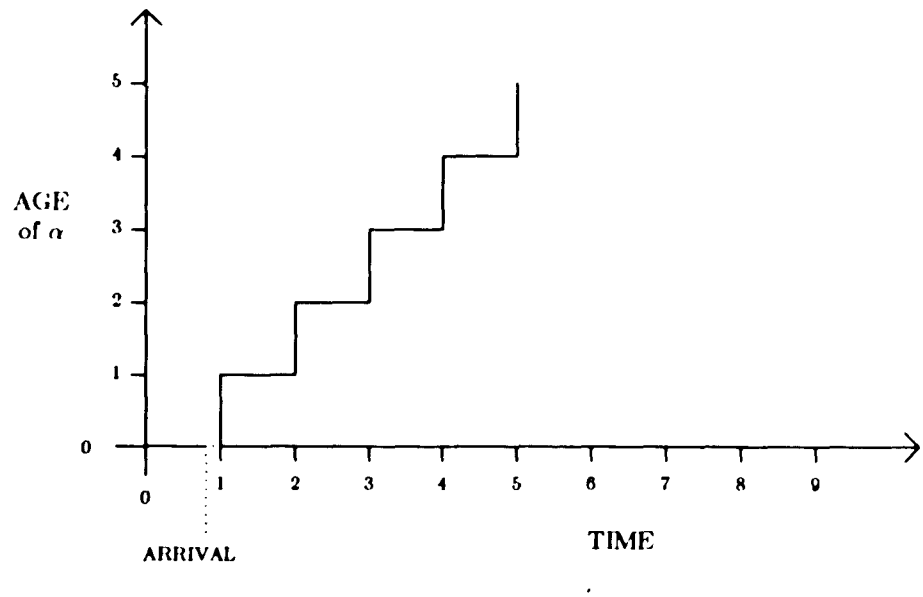
15. APPENDIX IV

FIGURE 1.



STATES OF A PROCESS

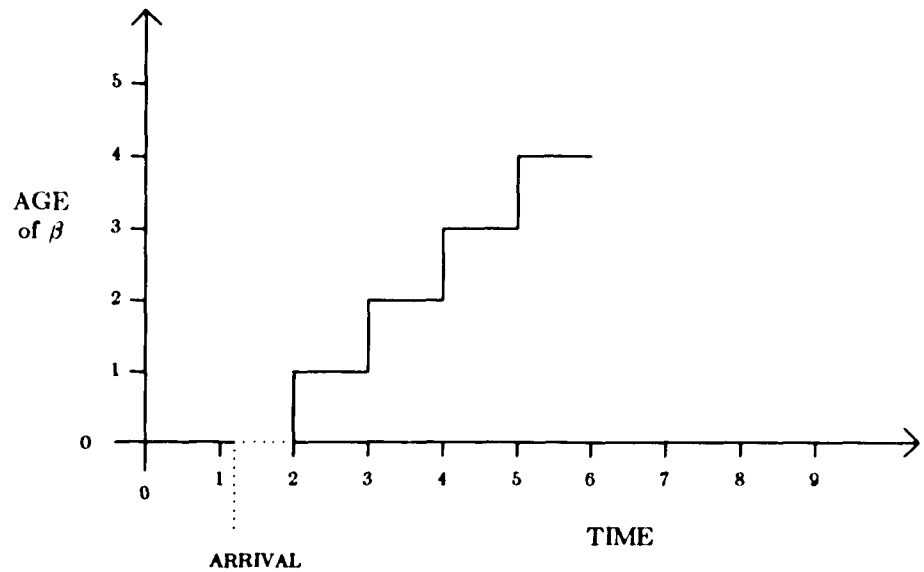
FIGURE 2a.



a) OVERAGING - EARLY ARRIVAL

AGE OF TWO PROCESSES ARRIVING IN MEMORY
AT APPROXIMATELY THE SAME TIME

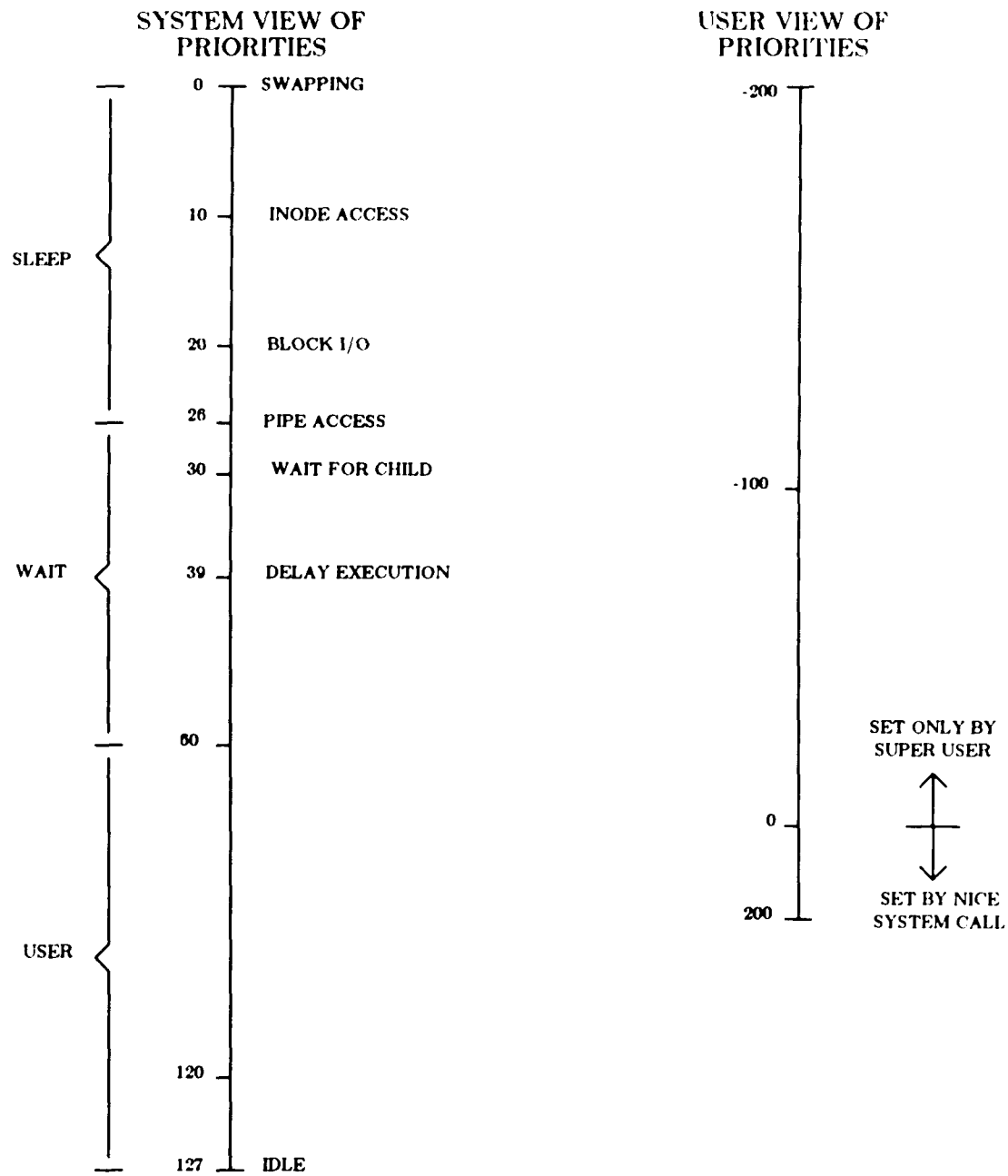
FIGURE 2b.



b) UNDERAGING - LATE ARRIVAL

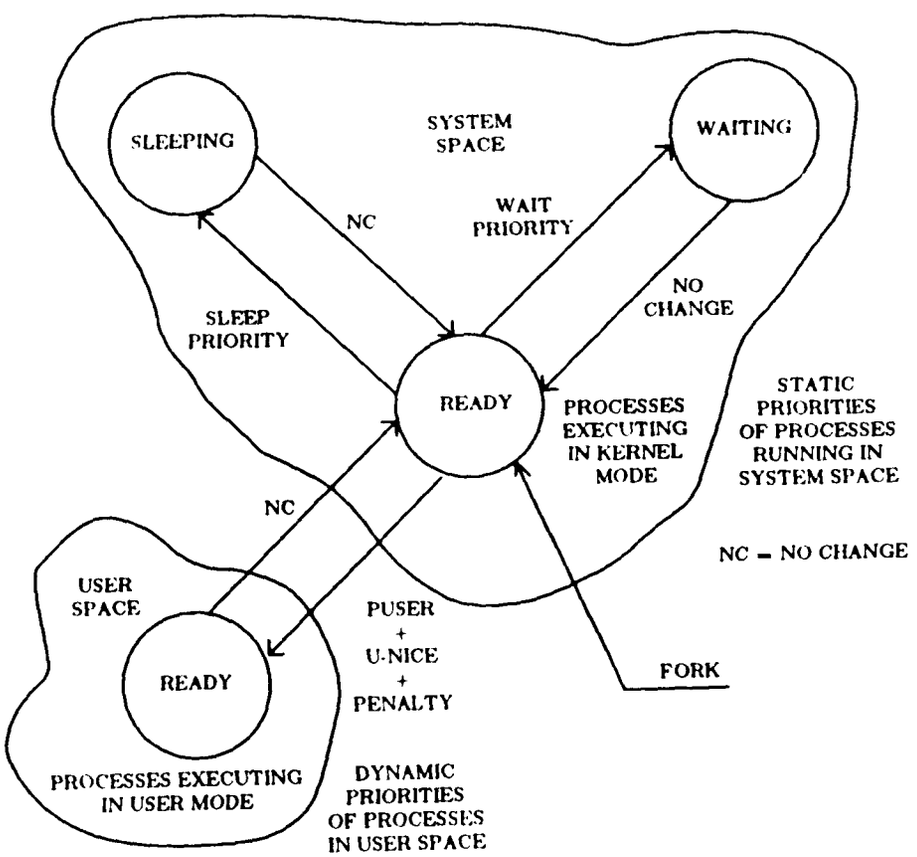
AGE OF TWO PROCESSES ARRIVING IN MEMORY
AT APPROXIMATELY THE SAME TIME

FIGURE 3.



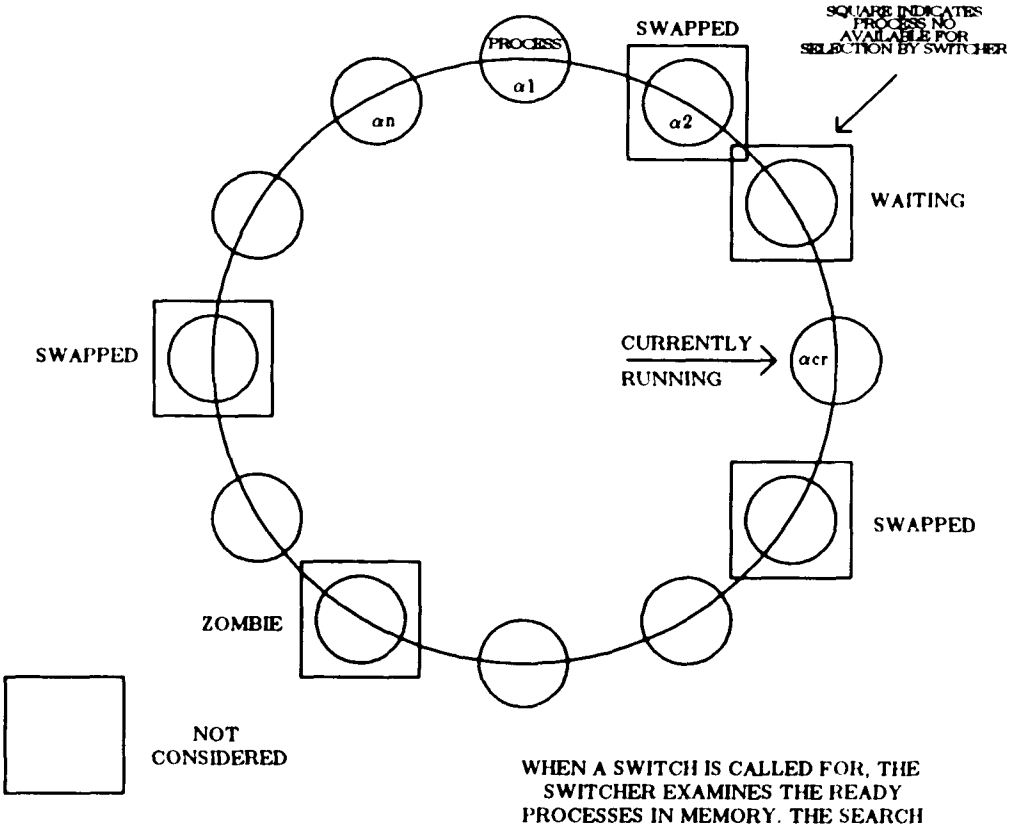
SYSTEM PRIORITIES - NOMOGRAPH

FIGURE 4.



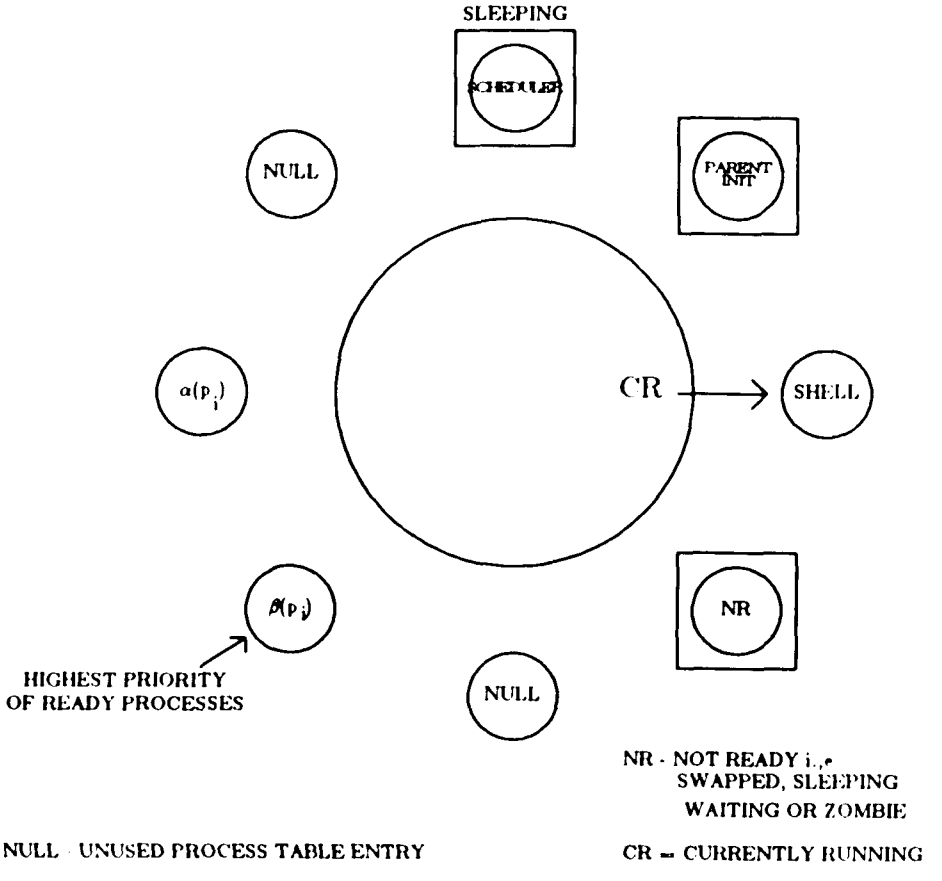
CHANGES IN PRIORITY OF A PROCESS

FIGURE 5.



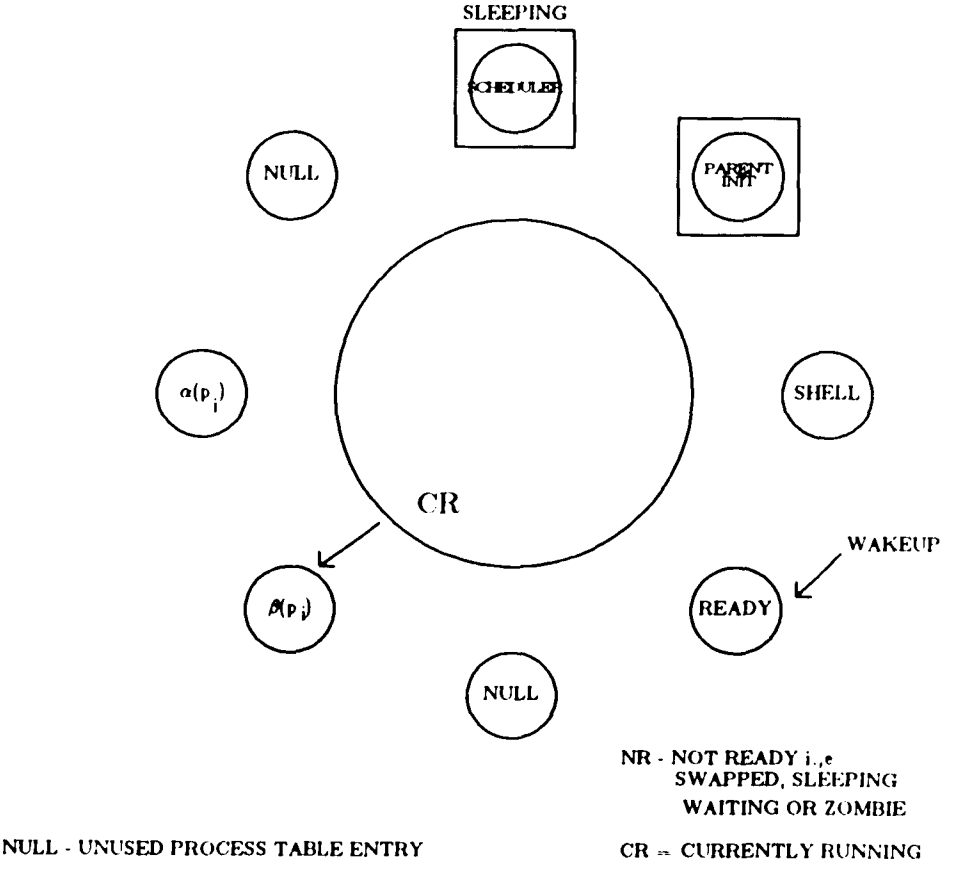
THE SWITCHER - SELECTION OF A PROCESS

FIGURE 6a.



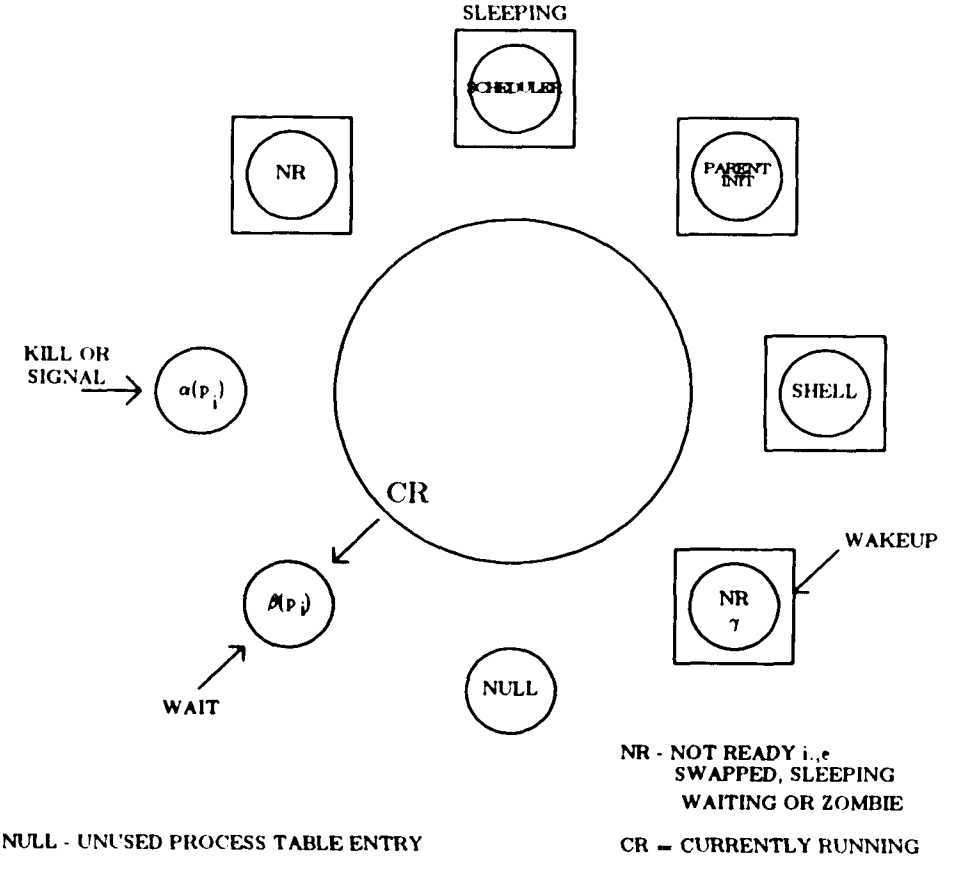
DYNAMICS OF SWITCHER SELECTION OF A PROCESS TO RUN

FIGURE 6b.



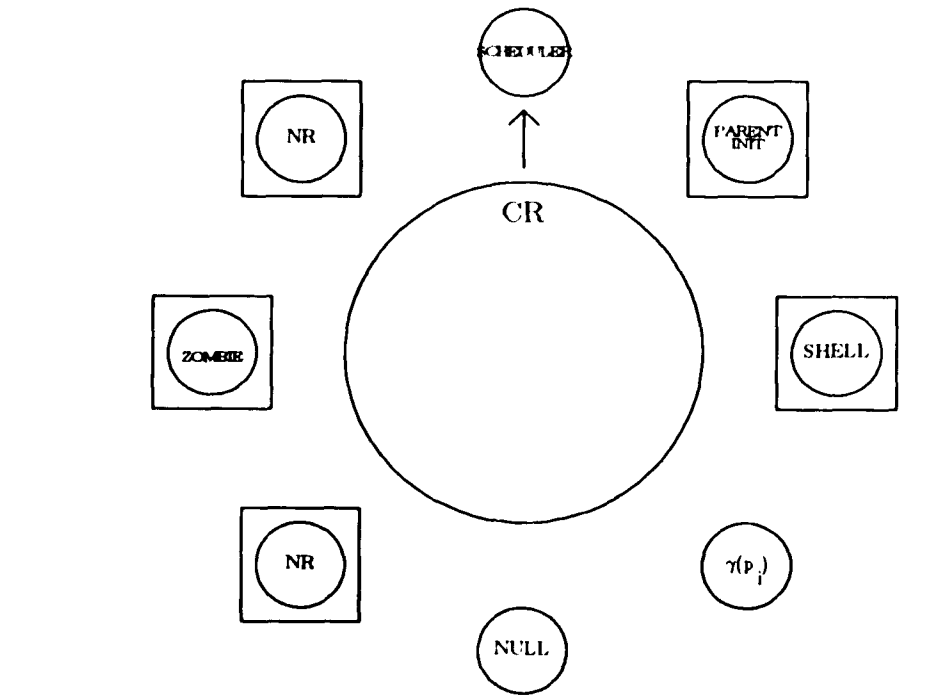
DYNAMICS OF SWITCHER SELECTION OF A PROCESS TO RUN

FIGURE 7a.



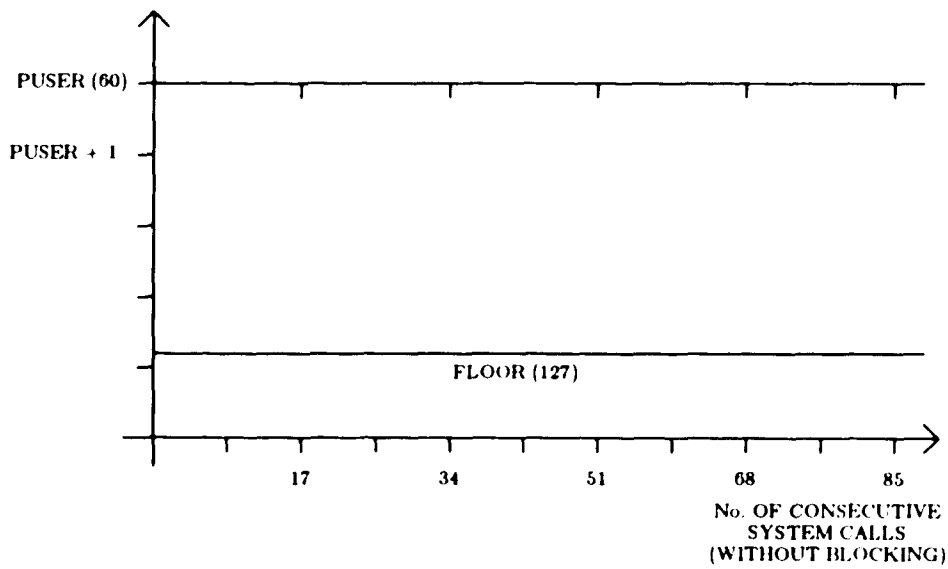
EVENTS CAUSING THE SWITCHER TO RUN THE SCHEDULER

FIGURE 7b.



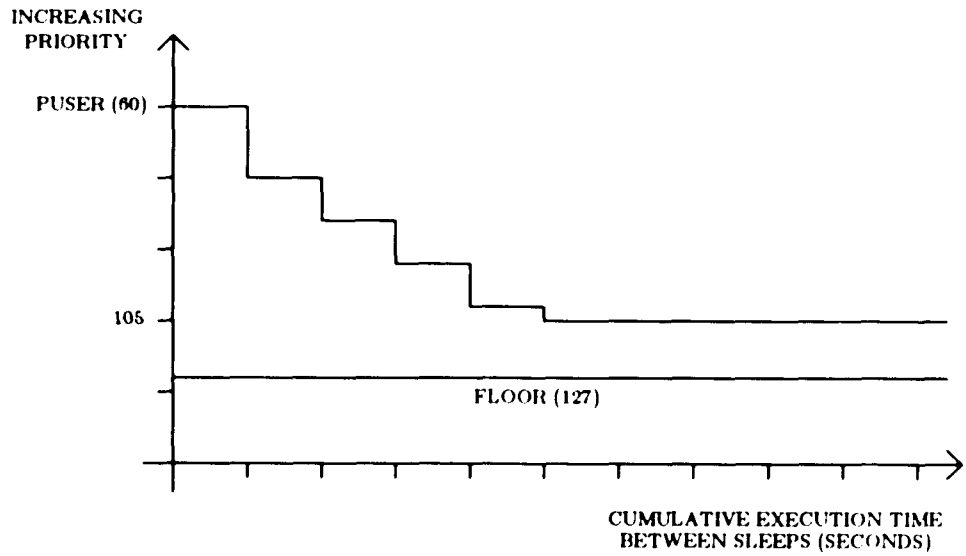
EVENTS CAUSING THE SWITCHER TO RUN THE SCHEDULER

FIGURE 8.



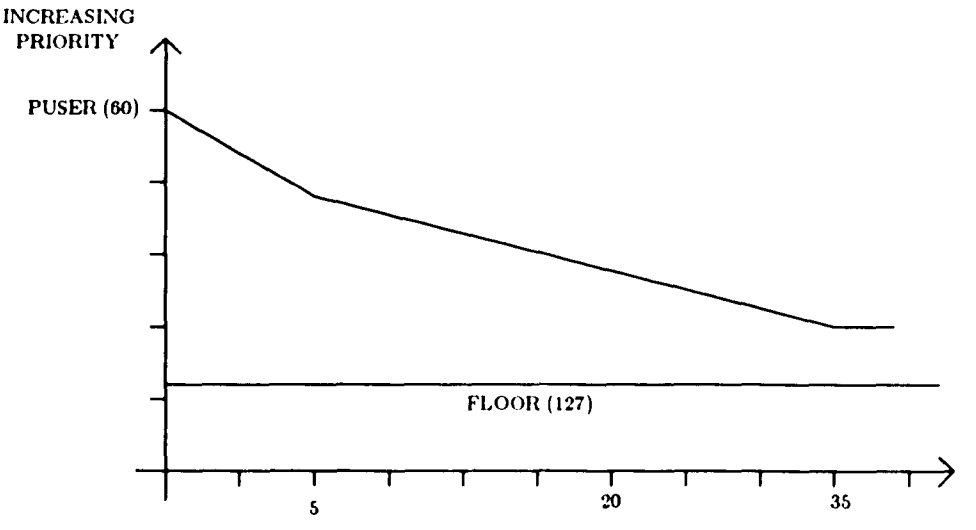
PENALTY SCHEME FOR SYSTEM
BOUND PROCESSES

FIGURE 9.



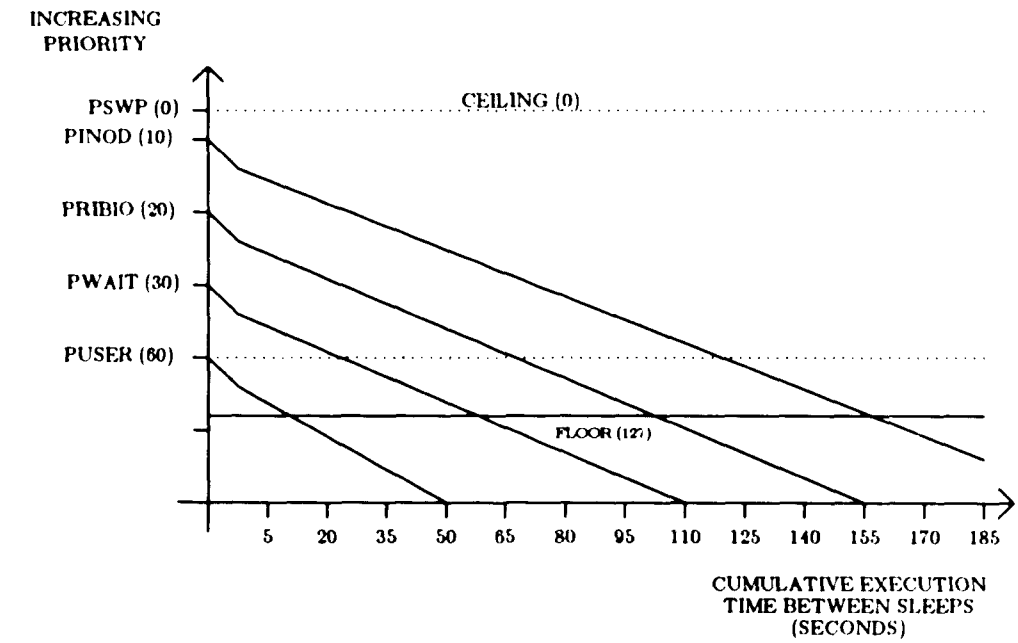
PENALTY SCHEME FOR CPU
BOUND PROCESSES

FIGURE 10.



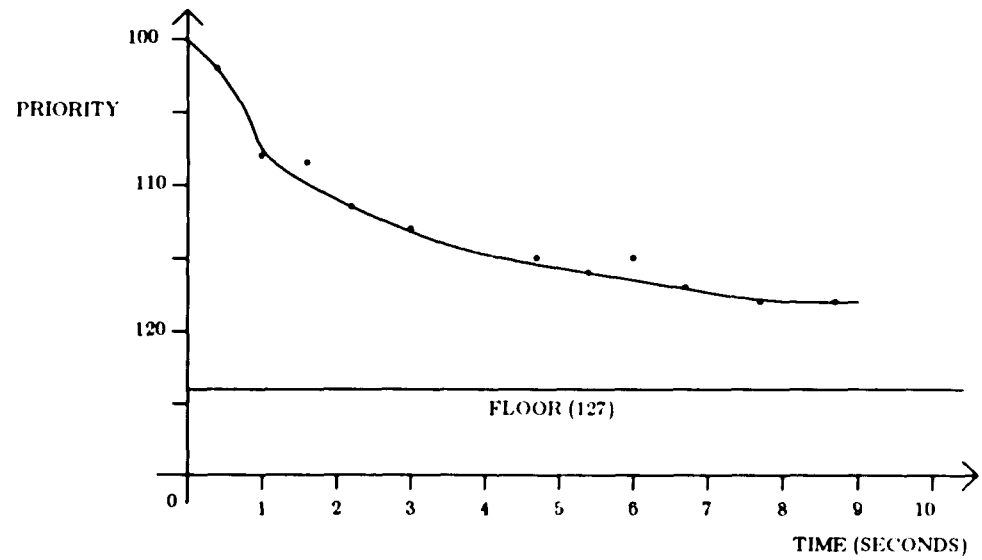
PRIORITY PENALTIES - GENERAL TREND

FIGURE 11.



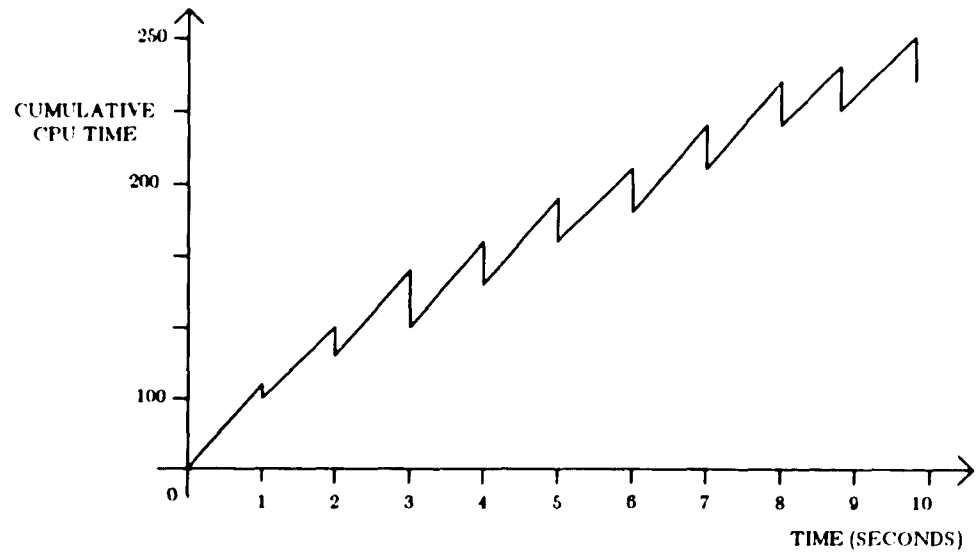
PRIORITY PENALTIES - CUMULATIVE EXECUTION TIMES

FIGURE 12.



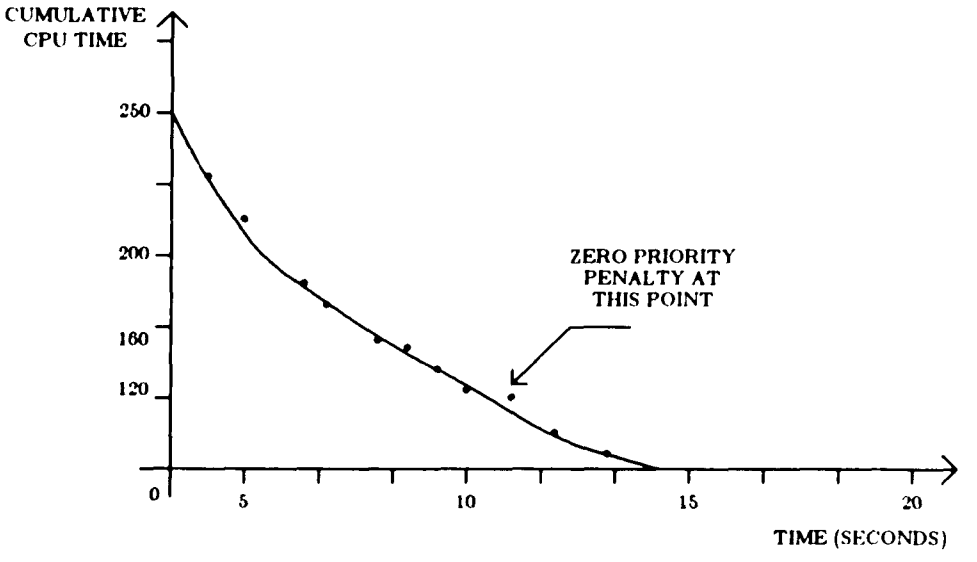
PRIORITY PENALTY FOR SINGLE
CPU BOUND PROCESS

FIGURE 13.



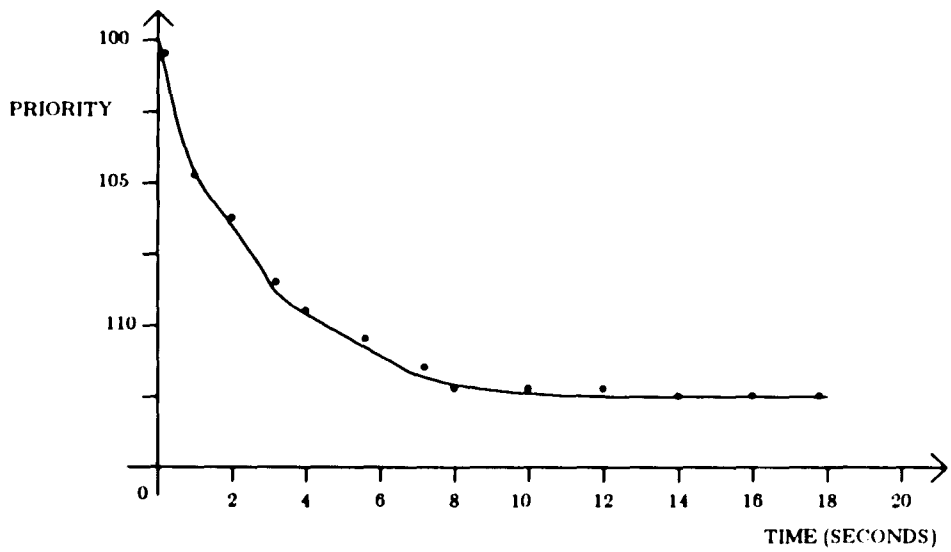
ADJUSTMENT OF CUMULATIVE CPU TIME

FIGURE 14.



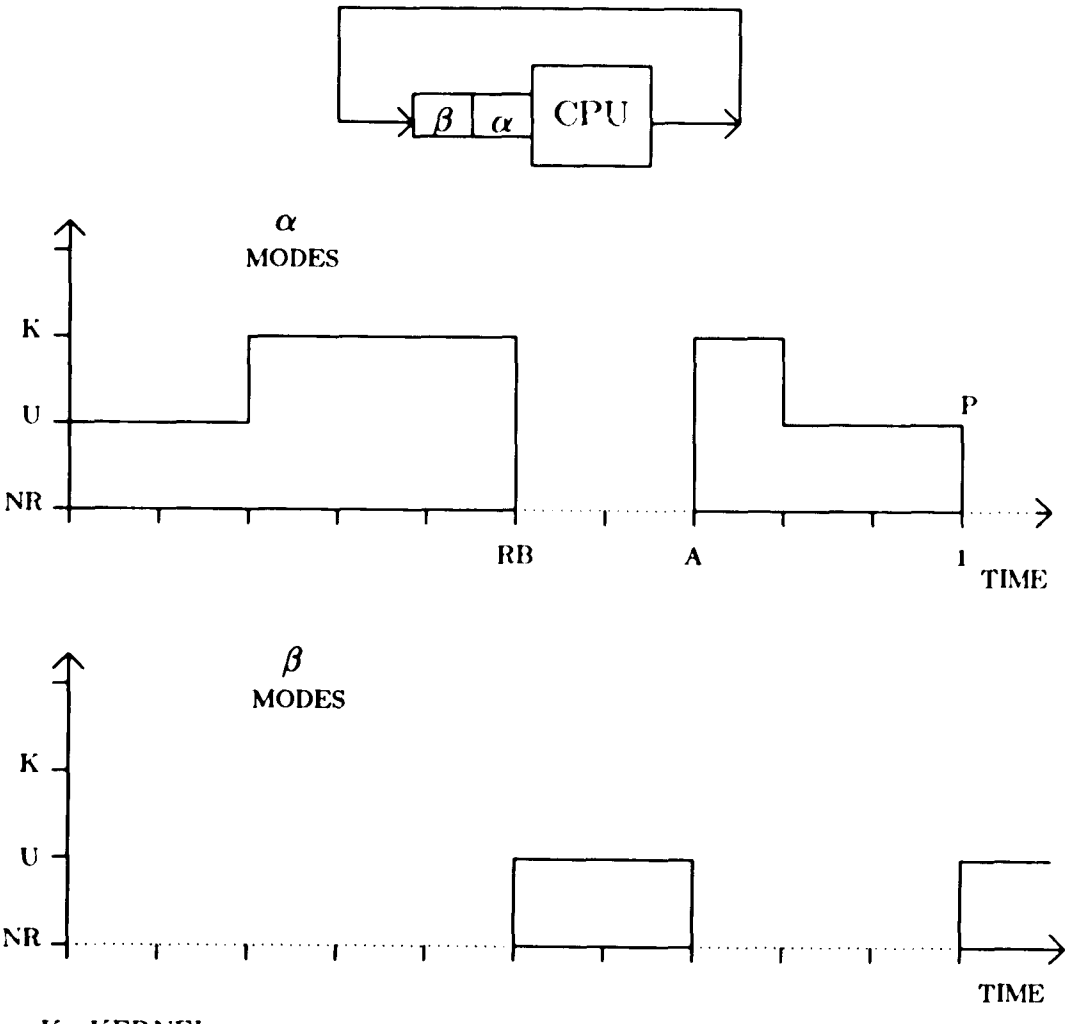
CPU TIME AVERAGE

FIGURE 15.



PRIORITY VARIATIONS FOR
CPU BOUND PROCESSES

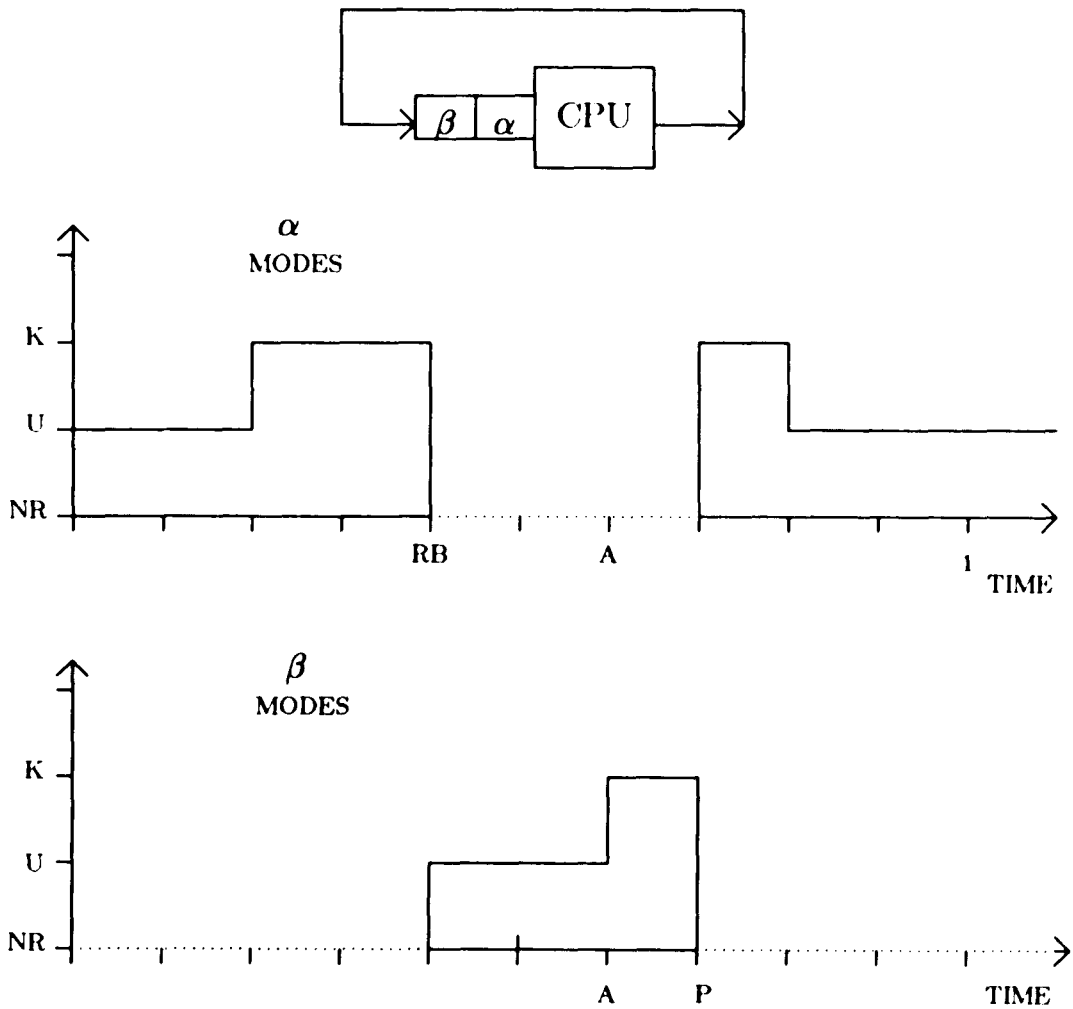
FIGURE 16.



K - KERNEL
U - USER
NR - NOT RUNNING
RB - ROADBLOCKED
A - AWAKENED
P - PREEMPTED
1 - 1 SECOND CLOCK

PROCESSOR SHARING - USER MODE PREEMPTION

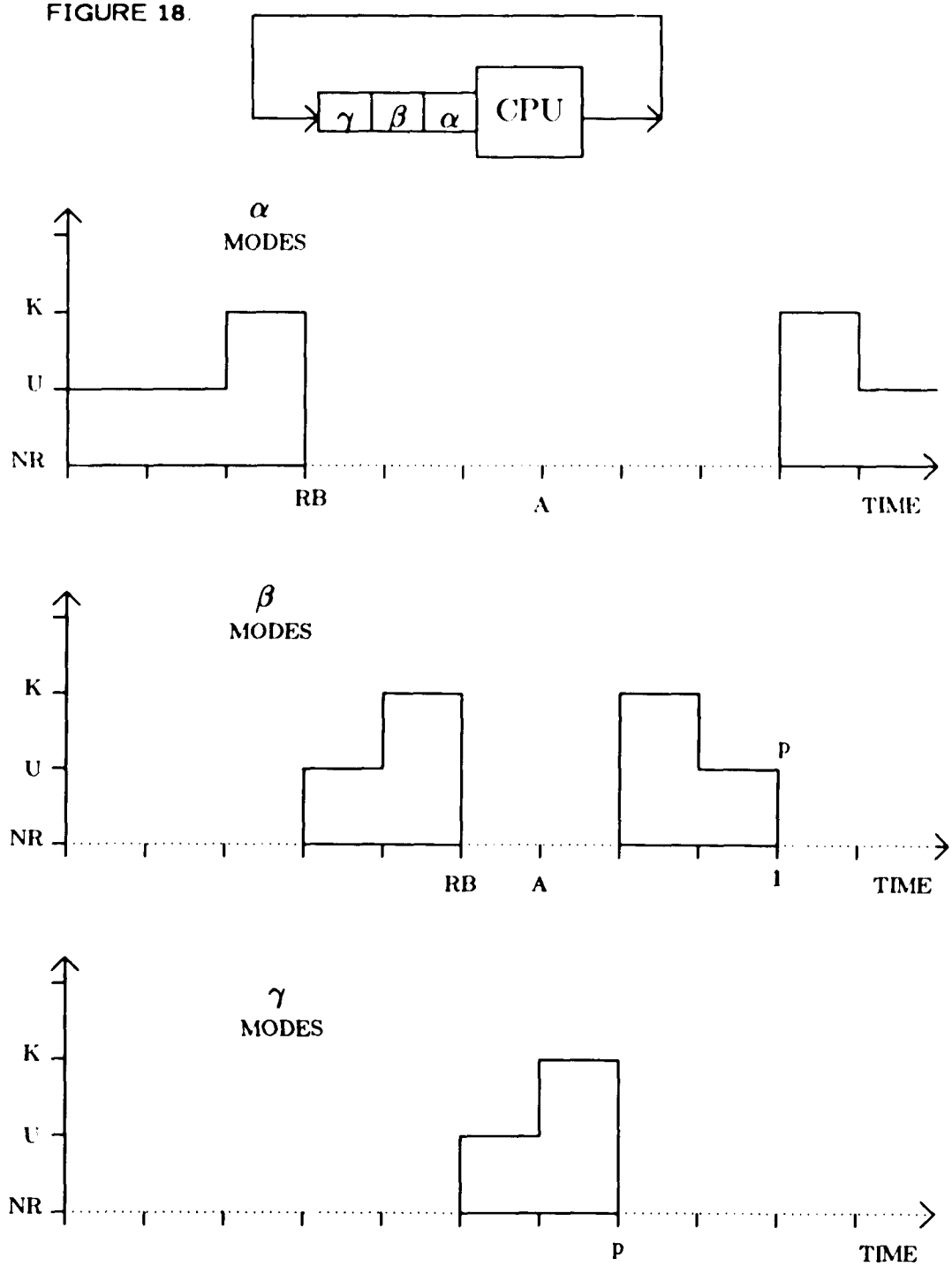
FIGURE 17.



K - KERNEL
U - USER
NR - NOT RUNNING
RB - ROADBLOCKED
A - AWAKENED
P - PREEMPTED
1 - 1 SECOND CLOCK

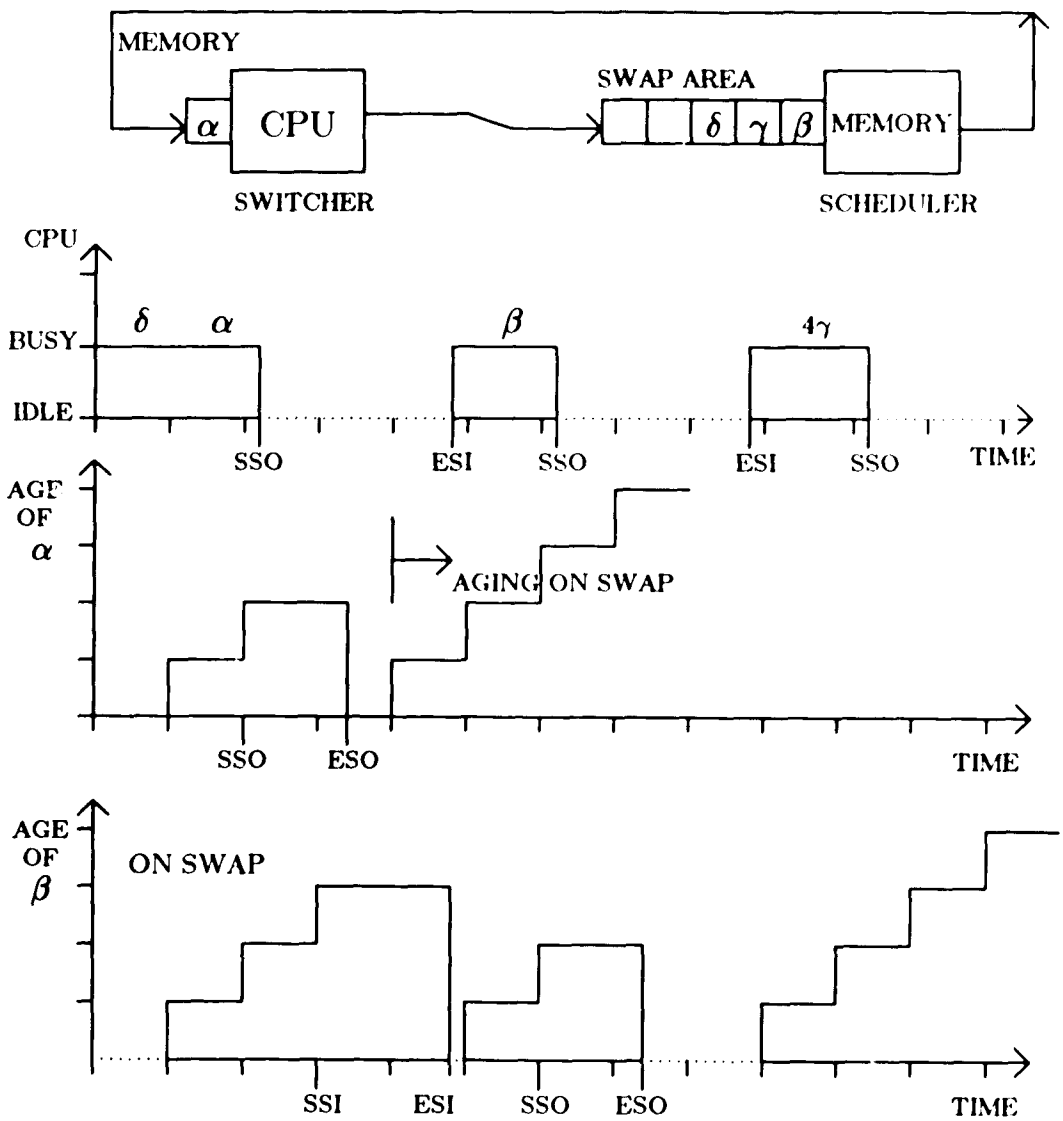
PROCESSOR SHARING - KERNEL MODE PREEMPTION

FIGURE 18.



PROCESSOR SHARING - NO ATTEMPT TO COUNT NUMBER OF WAKEUPS

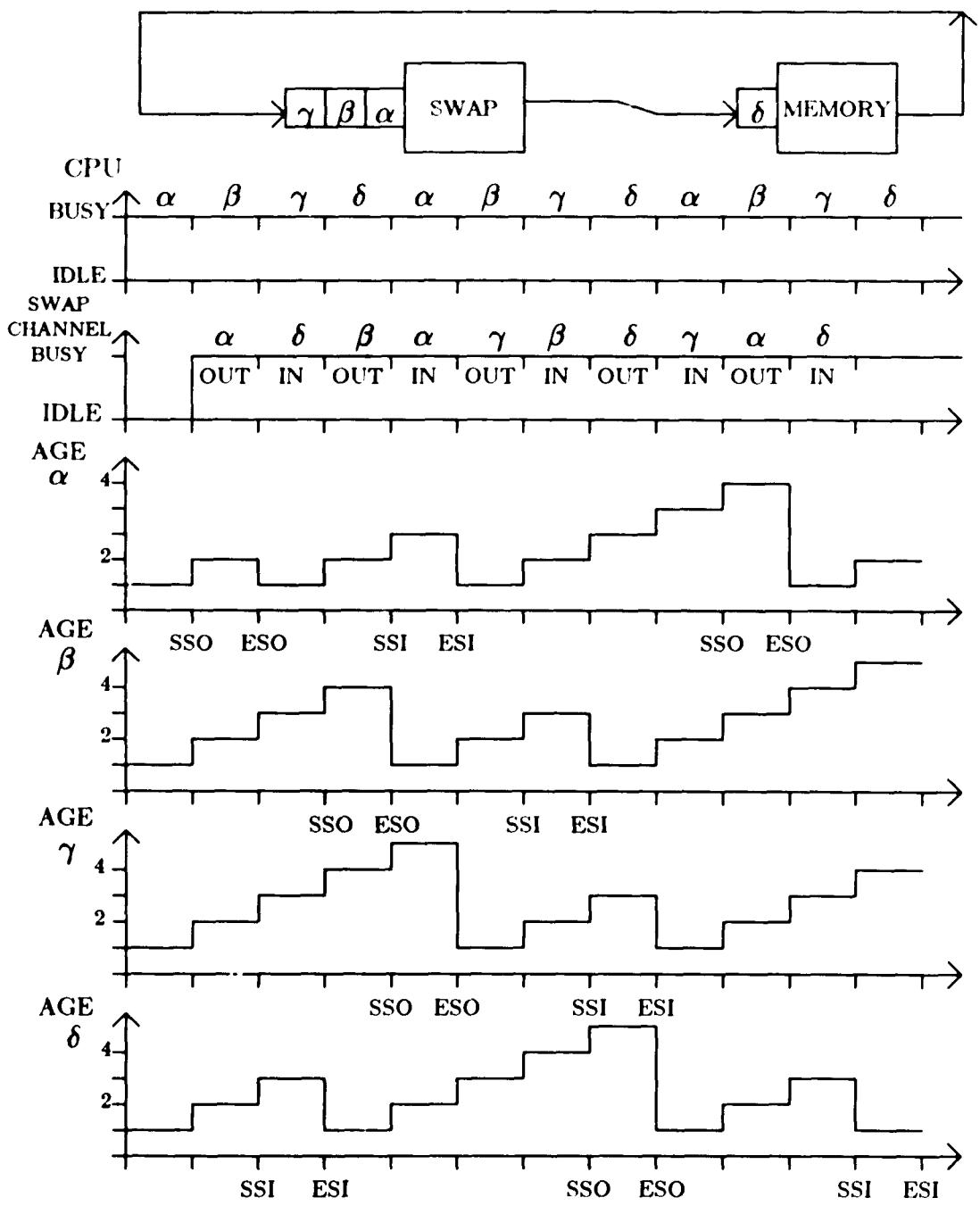
FIGURE 19.



SSI - START SWAP IN
SSO - START SWAP OUT
ESI - END SWAP IN
ESO - END SWAP OUT

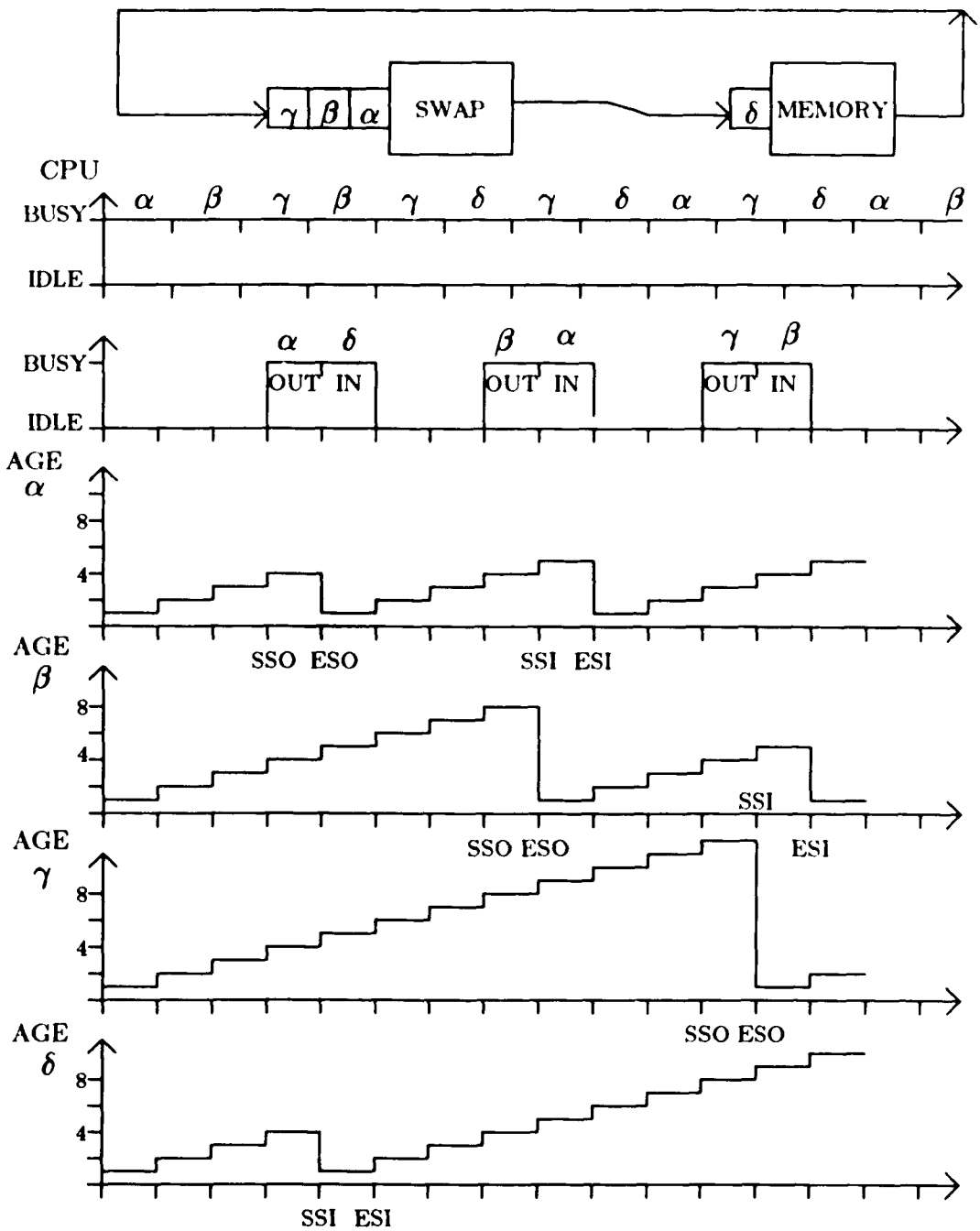
THRASHING PREVENTION - INFINITE POPULATION
ON SWAP; ROOM FOR ONLY ONE IN MEMORY

FIGURE 20.



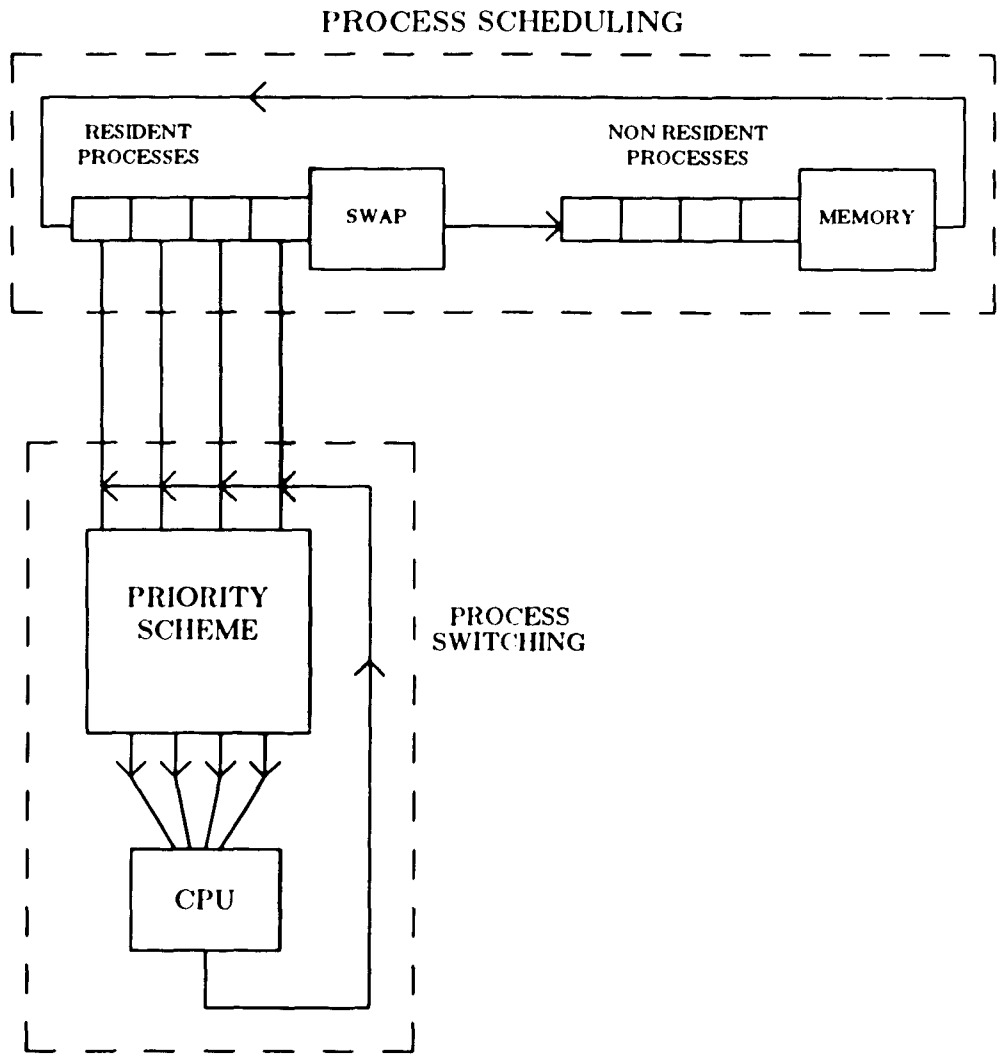
THRASHING - ONE DEVICE CHANNEL - NO MEMORY RESIDENCY REQUIREMENTS

FIGURE 21.



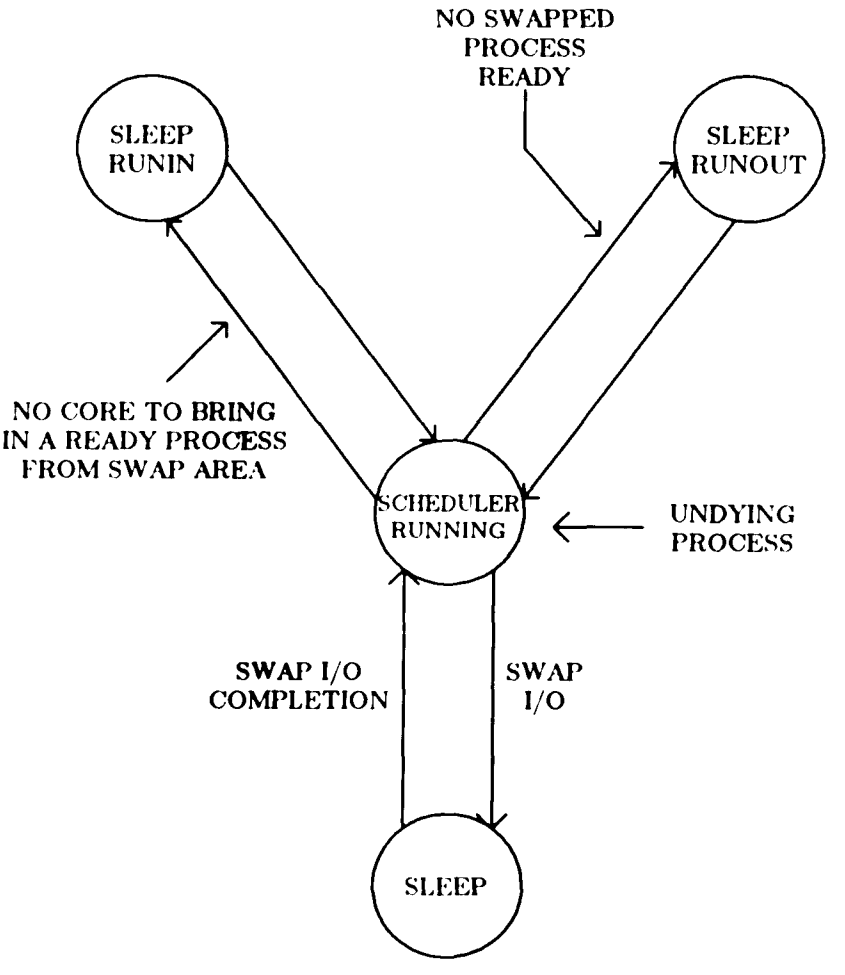
THRASHING PREVENTION
SWAP RESIDENCY REQUIREMENTS

FIGURE 22.



LOOSE COUPLING OF PROCESS SCHEDULING AND SWITCHING

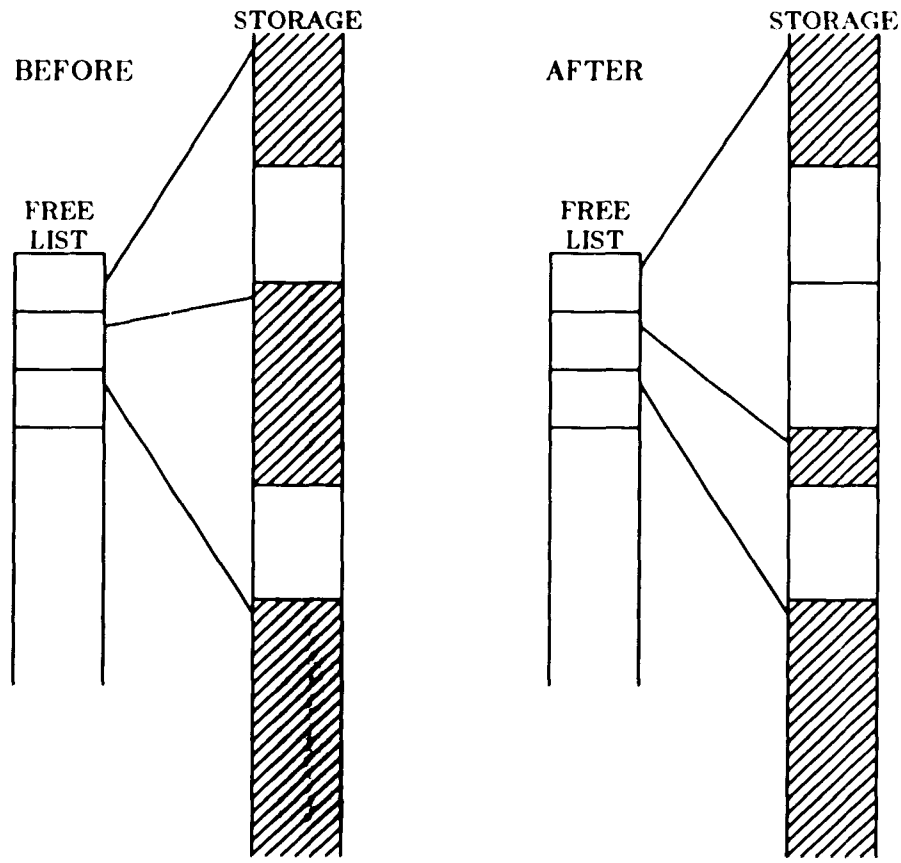
FIGURE 23.



NOTE: SCHEDULER IS A PROCESS THAT RUNS IN KERNEL SPACE SO THAT IT MAY HAVE DIRECT ACCESS TO SYSTEM SERVICES.

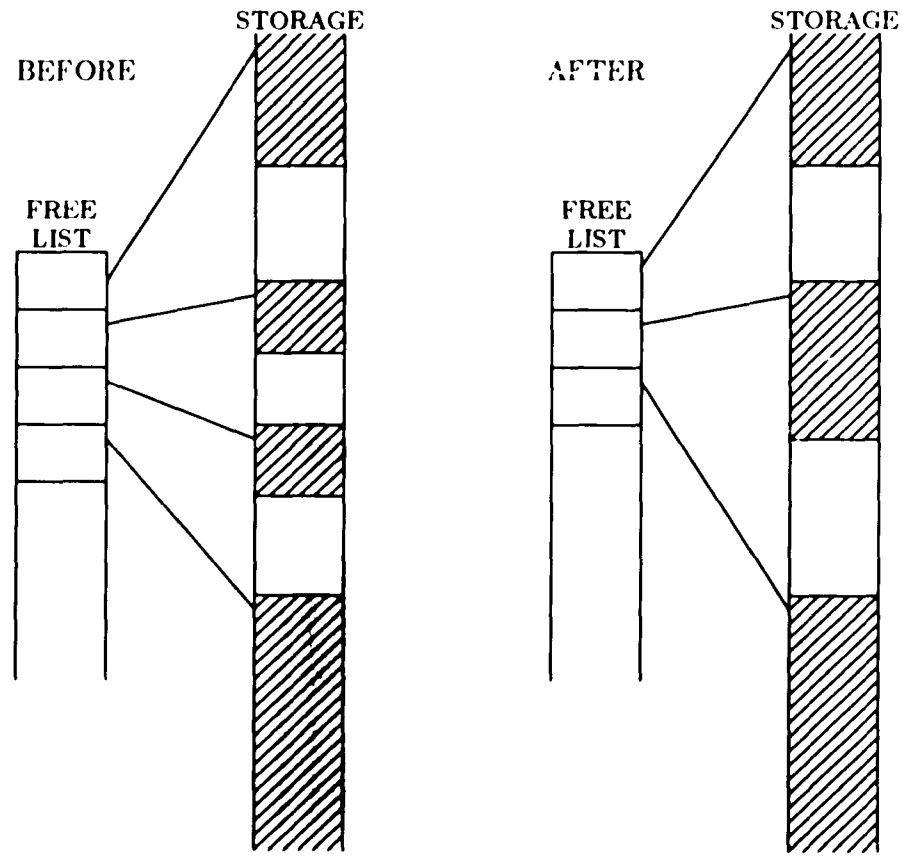
SCHEDULER STATES

FIGURE 24.



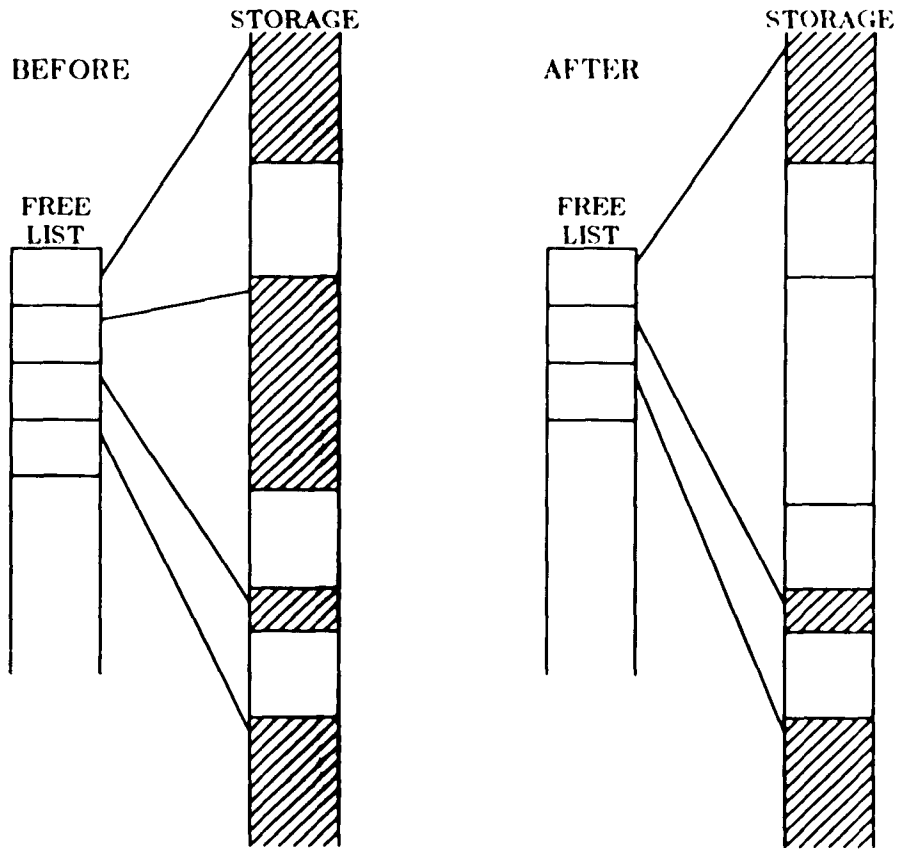
ALLOCATION OF STORAGE

FIGURE 25.



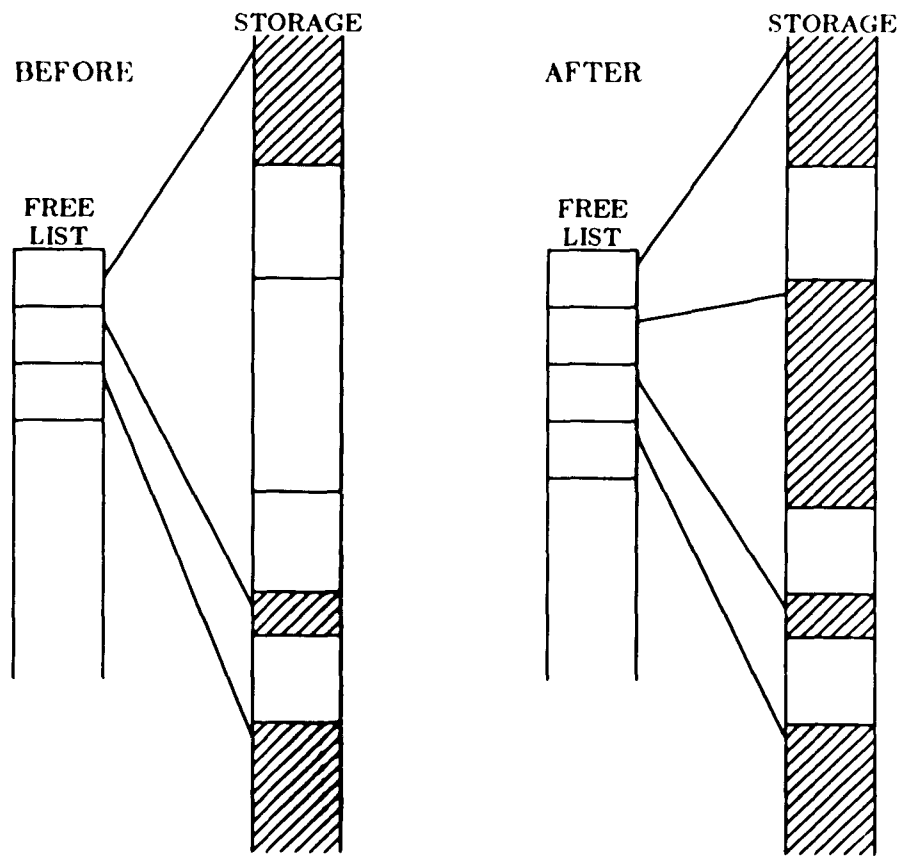
RECOMBINATION OF CONTIGUOUS
FREE AREAS AND STORAGE
LIST COMPRESSION

FIGURE 26.



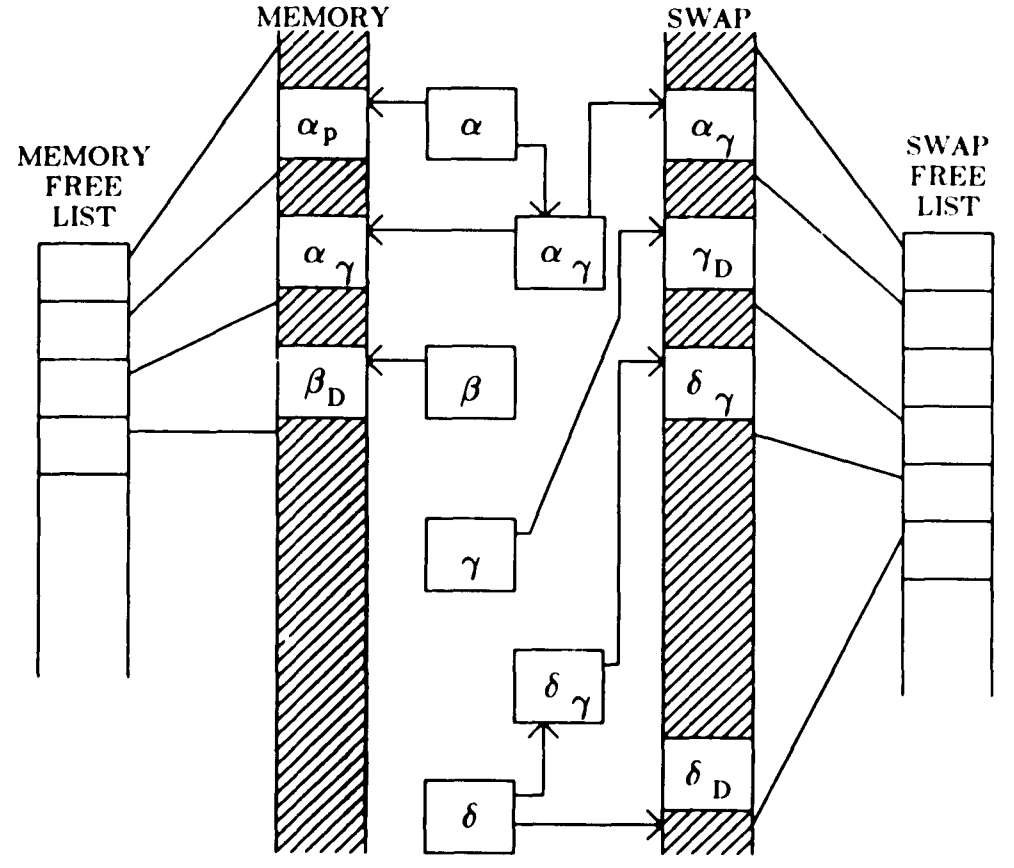
ALLOCATION OF FREE STORAGE

FIGURE 27.



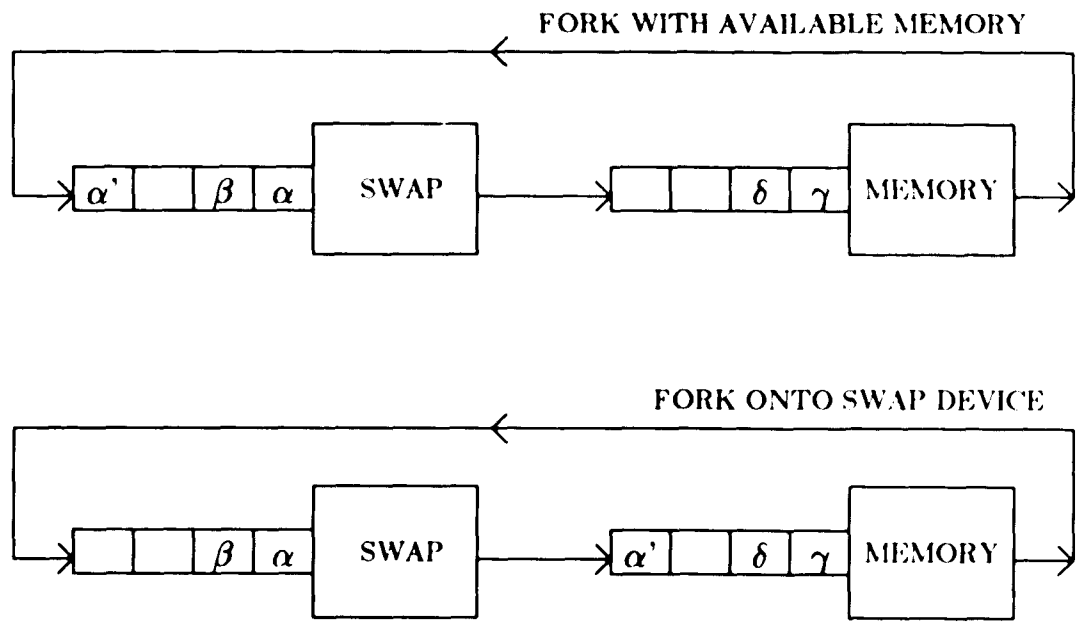
DEALLOCATION OF FREE STORAGE

FIGURE 28.



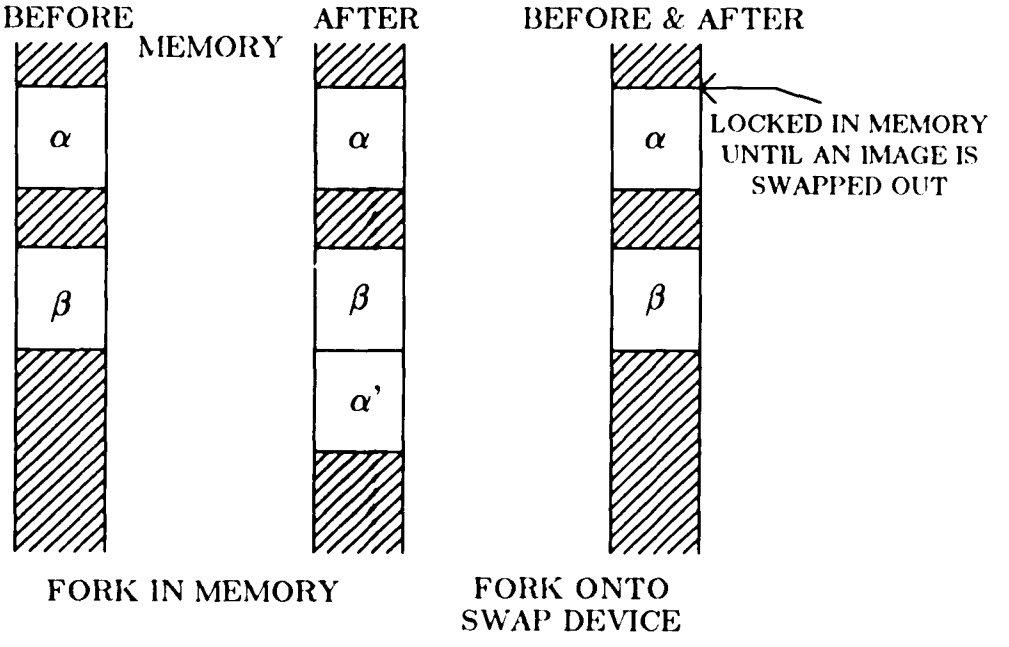
MEMORY AND SWAP MANAGEMENT FOR
REENTRANT AND NON-REENTRANT PROCESSES

FIGURE 29a.



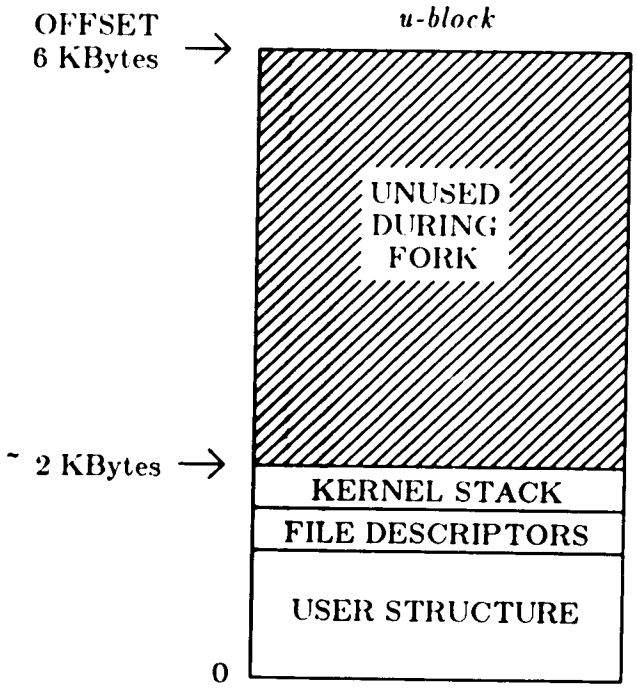
PROCESS CREATION - (FORK OPERATION)

FIGURE 29b.



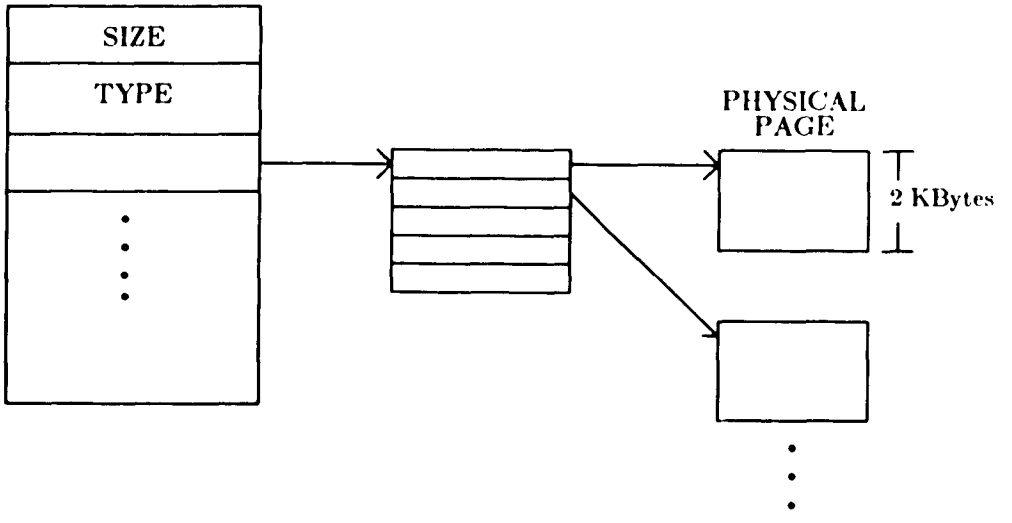
FORK OPERATION

FIGURE 29c.



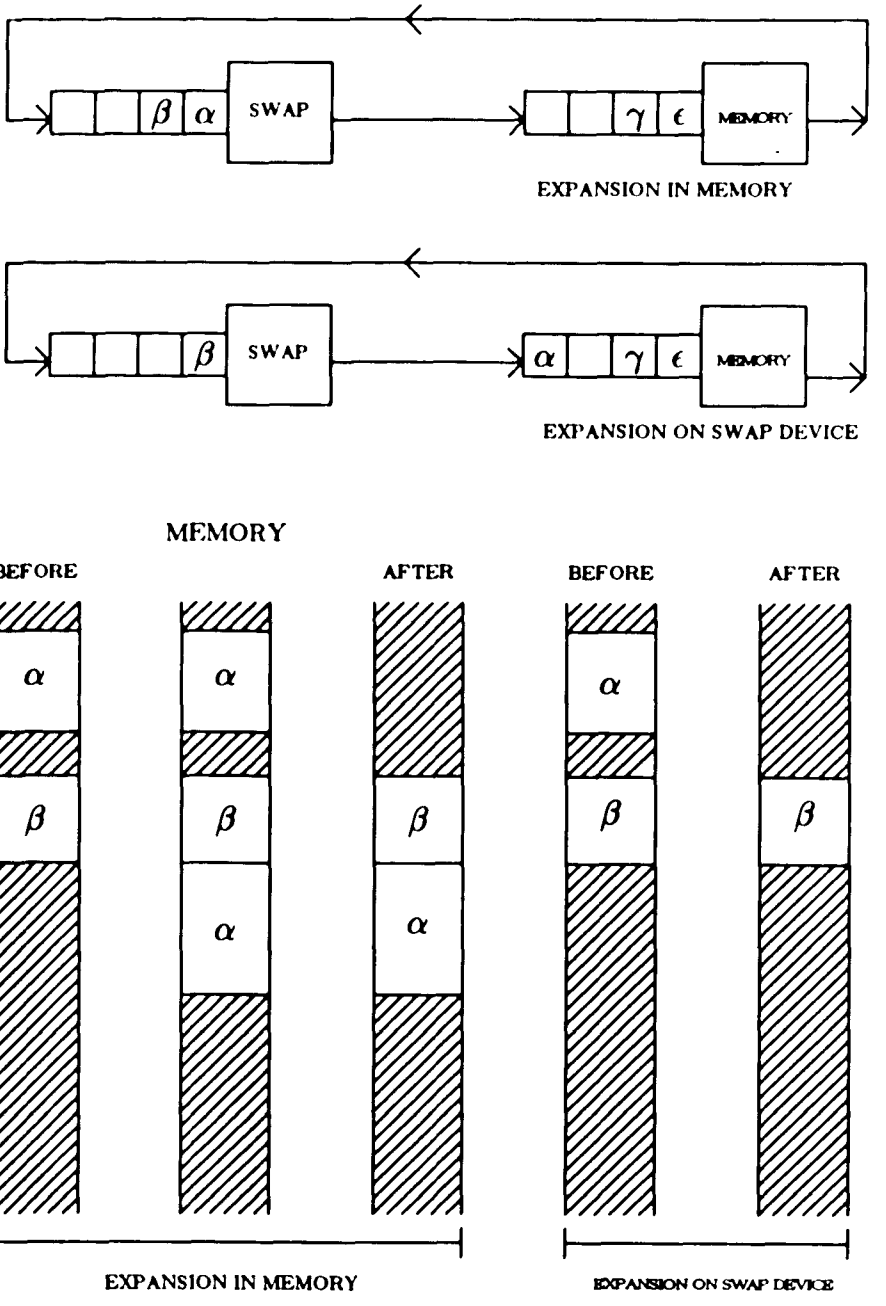
LAYOUT OF THE USER BLOCK (u-block)

FIGURE 29d.



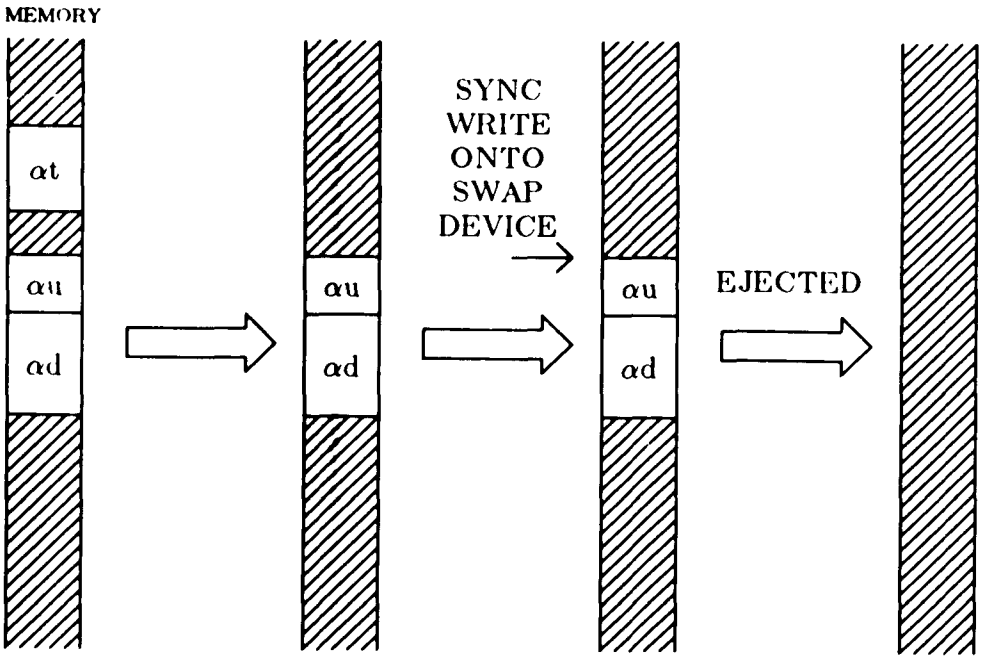
REGION DATA STRUCTURE WITH DIRECT
PAGE-TABLE POINTER

FIGURE 30.



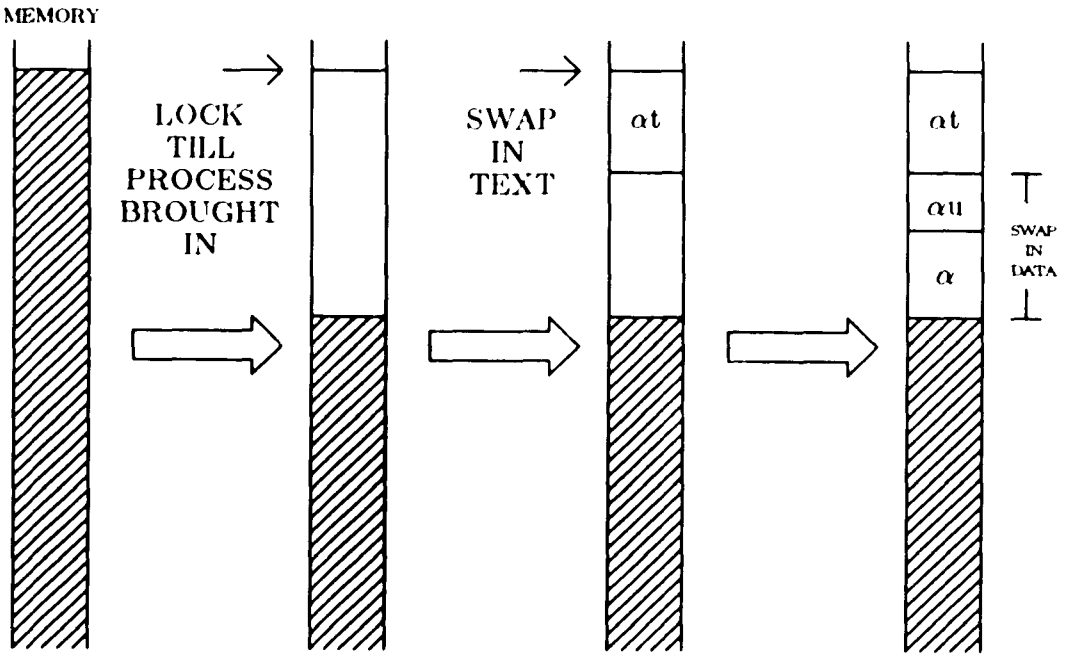
EXPANDING THE SIZE OF A PROCESS

FIGURE 31.



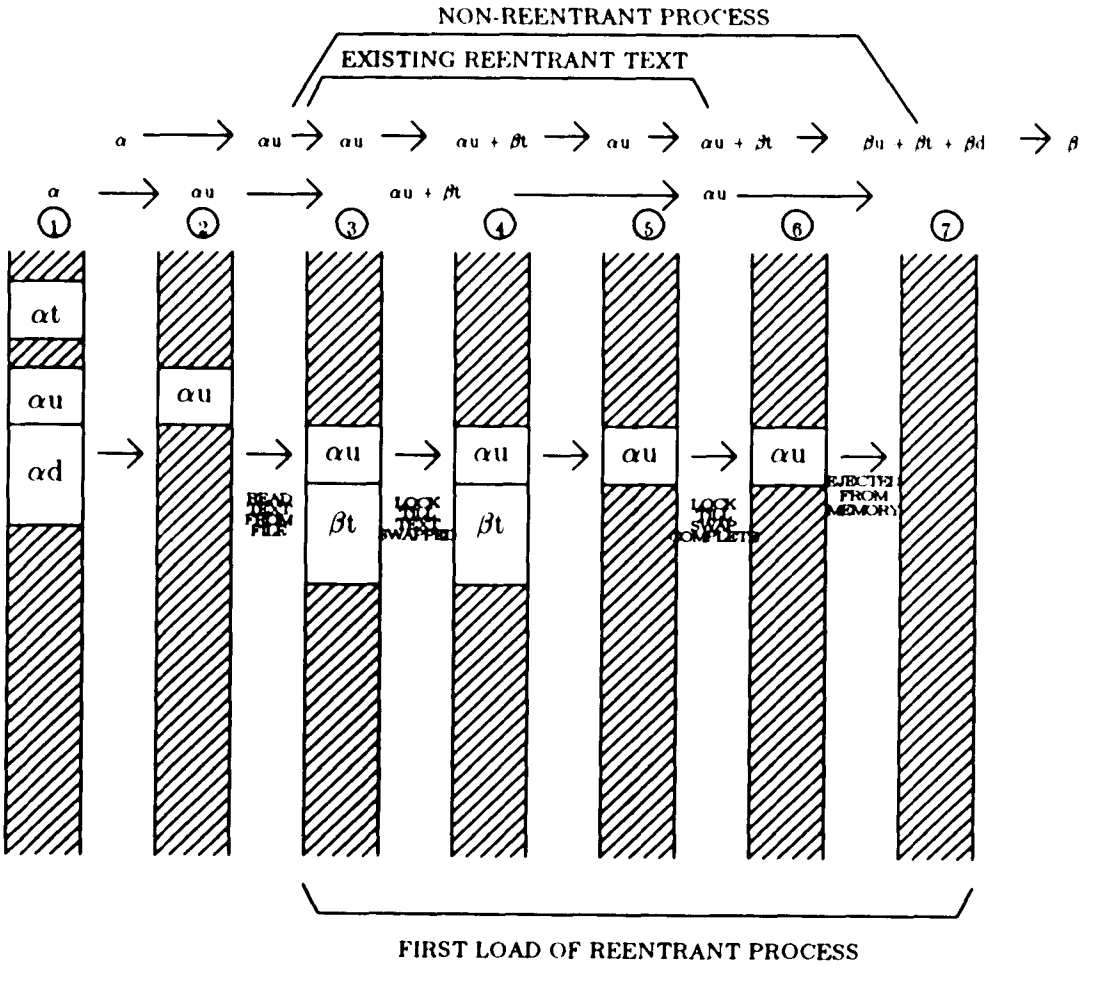
PROCESS BECOMING A ZOMBIE

FIGURE 32.



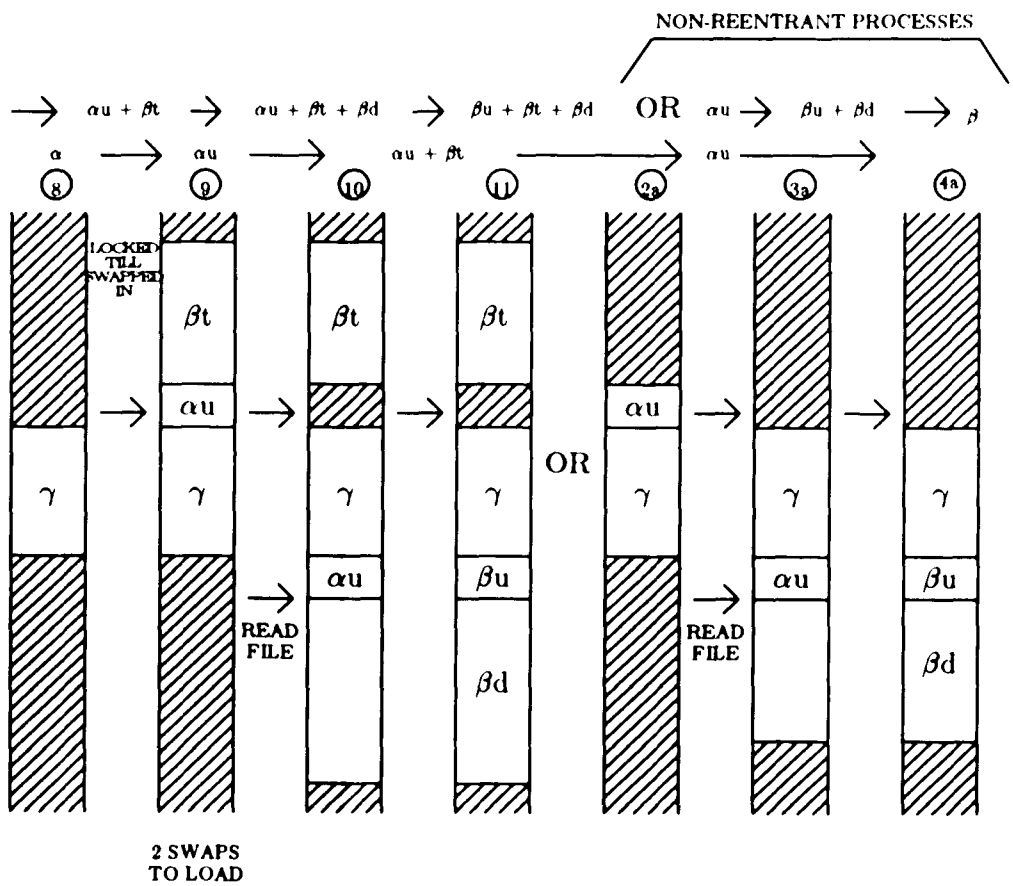
SCHEDULER SWAPPING REENTRANT PROCESS
INTO MEMORY WHEN TEXT NON-RESIDENT

FIGURE 33.



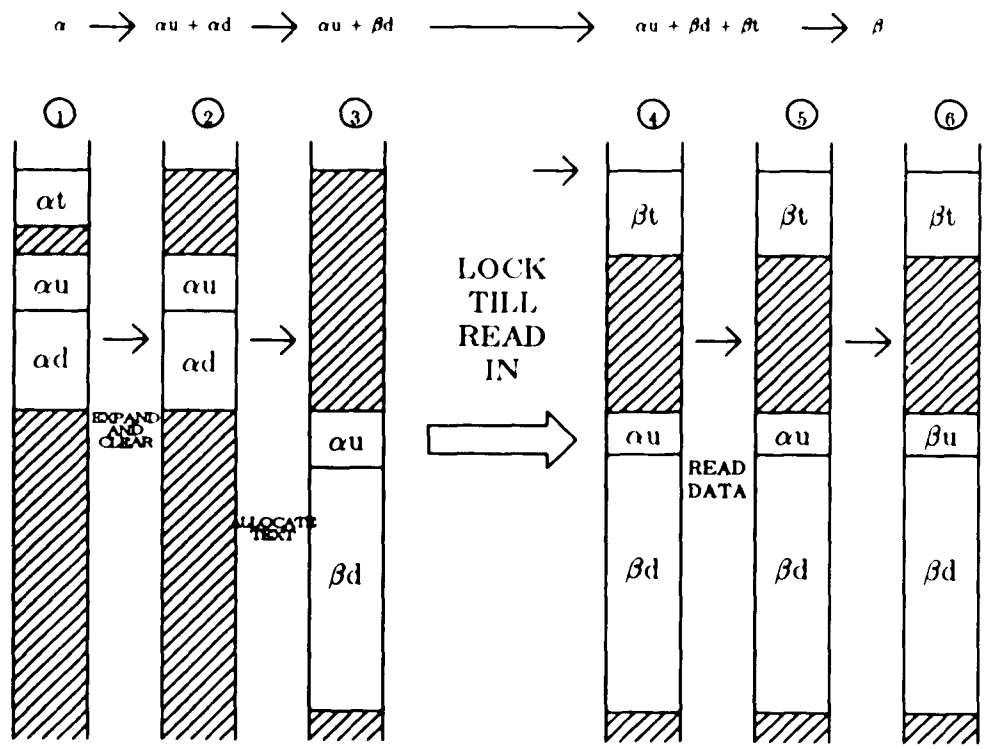
EXEC OPERATION - PART 1 LOADING REENTRANT TEXT

FIGURE 34.



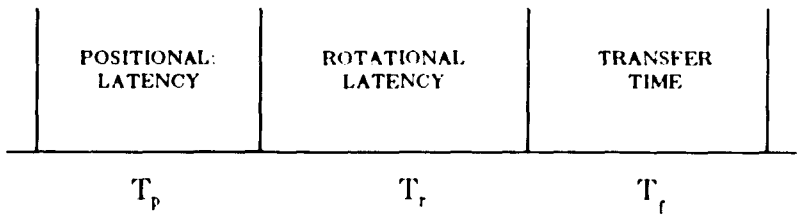
EXEC OPERATION - PART 2, LOADING NON-REENTRANT PARTS

FIGURE 35.

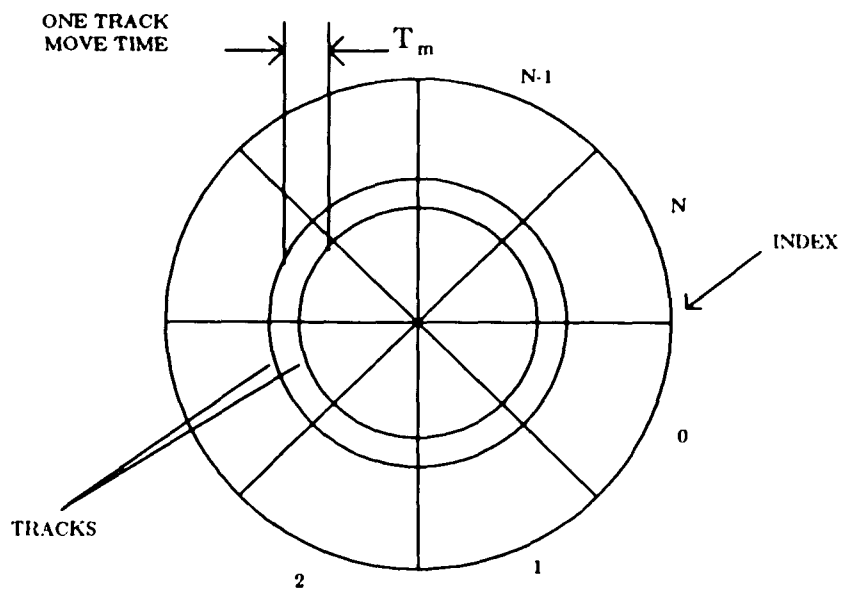


EXEC OPERATION - IMPROVEMENTS

FIGURE 36.

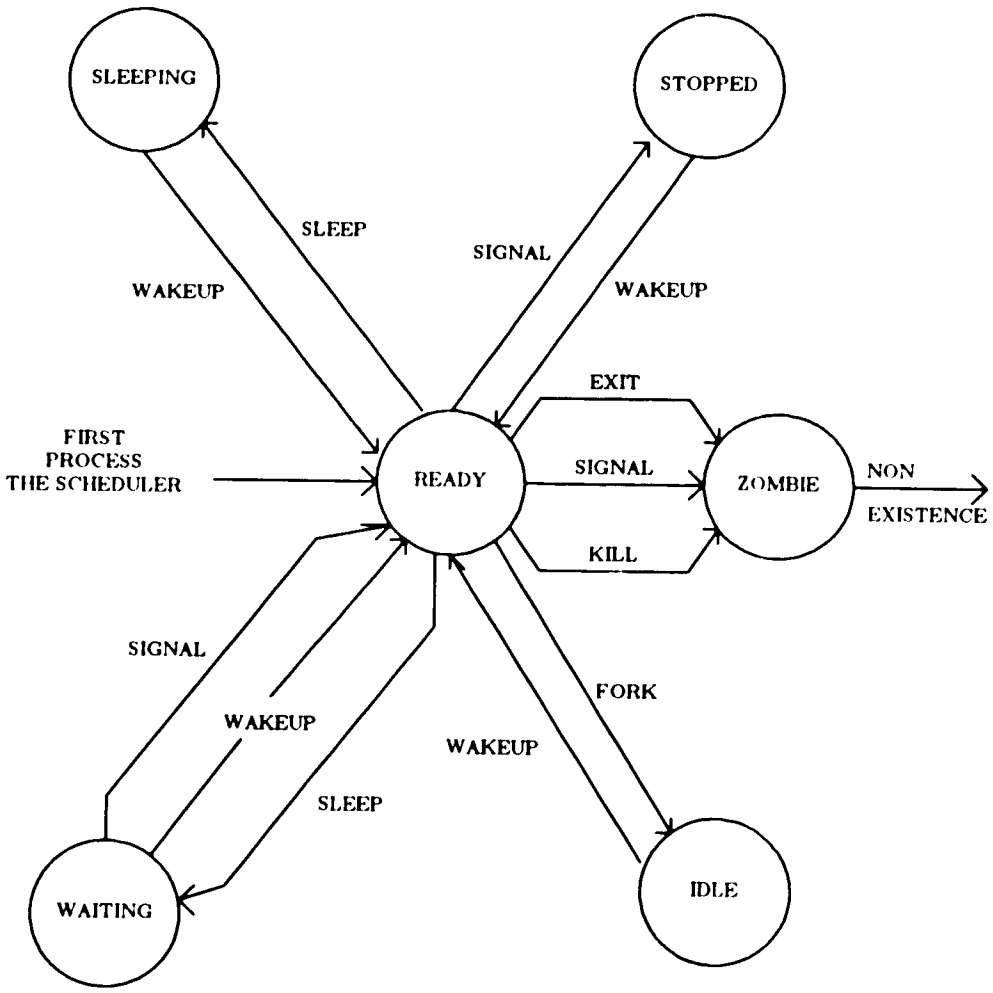


MODEL OF ROTATING STORAGE DEVICE



ADDITIONAL TIME DELAY FOR SPIRAL READ

FIGURE 37.



STATES OF A PROCESS

REFERENCES

1. M. D. McIlray, E. N. Pinson, and B. A. Tague "UNIX Time-Sharing System: Foreword", *The Bell System Technical Journal* July-August 1978, Vol. 57, No. 6, Part 2, pp. 1899-1890

2. Maurice J. Bach "The Design of The UNIX Operating System," *AT&T - Prentice-Hall*, 1986. Englewood Cliffs, New Jersey 07632, pp 146-283.

3. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Comm. Assoc. Comp.* March., 17, No. 7 (July 1974), pp. 365-375.

4. D. M. Ritchie, "The Evolution of the UNIX Time-sharing System," *AT&T Bell Laboratories Technical Journal.* October 1984, Vol. 63 No. 8 Part 2, pp. 1577-1593.

5. AT&T UNIX System V Administrator Guide. April 1984.

6. Weinberger, P. J., "Cheap Dynamic Instruction Counting," The AT&T Bell Laboratories Technical Journal, Vol. 63, No. 6, Part 2, October 1984, pp. 1815-1826.

7. Barkley R.E., Chen D., "Tracing and Timing Kernel Activities with CASPER," *AT&T Bell Laboratories*, July 11, 1987.

8. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *The Bell System Technical Journal*, July-August 1978, Vol. 57, Part 2, pp. 1905-1930.

9. Scheduling and Switching under UNIX: Algorithms and Implementation *AT&T Bell Laboratories Technical Memorandum.* October 1976.

10. Felix E. Sańchez, "The UNIX System Scheduler and Switcher. Algorithms and Implementation" *AT&T Bell Laboratories Technical Memorandum.* December 1987.

11. Bunt, R. B., "Scheduling Techniques for Operating Systems," *Computer*, October 1976, pp. 10-17.

12. Raleigh, T. M., "Introduction To Scheduling and Switching under UNIX," *Bell Laboratories, Internal Memorandum* October 20, 1975.

13. Raleigh, T. M., "Introduction To Scheduling and Switching under UNIX," *Proceedings of the Digital Equipment Computer Users Society*, Atlanta, Ga., May 1976, pp. 867-877.

14. Peachey, D. R., R. B. Bunt, C. L. Williamson, and T. B. Brecht, "An Experimental Investigation of Scheduling Strategies for UNIX," *Performance Evaluation Review, 1984 SIGMETRICS Conference on Measurement and Evaluation of Computer Systems*, Vol. 12(3), August 1984, pp. 158-166.

15. Denning, P. J., "Thrashing: Its causes and prevention", *AFIPS FJCC* 1968, pp. 915-922

16. AT&T 6386 WGS Processor "Hardware Reference Manual," *AT&T 6386 WGS Computer*, Code 4116190 V (0) Printed in Italy July 1988, pp. 2-1 to 4-19.

17. IBM Personal Computer "AT Technical Reference Manual", IBM Corporation, 1988, pp. 1-7.

18. Knuth, D.E., "The Art of Computer Programming: Vol. 1 Fundamental Algorithms", *Addison-Wesley*, pp. 435-451, 1979.

19. AT&T UNIX System V/386 Release 3.2 Programmer Reference Manual. Section 2. *AT&T 6386 WGS Computers*. October 1988.

20. AT&T UNIX System V/386 Release 3.2 Driver Design Series. *AT&T 6386 WGS Computers*. October 1988.

21. AT&T UNIX System V Administrator Guide. *AT&T 6386 WGS Computers*, October 1988.

22. CompuAdd Summer Catalog. *CompuAdd Corporation*, Austin, Texas 78727, April 1989.

23. Saltzer, J. H., Traffic Control in a Multiplexed Computer System, *Ph. D. Thesis*, MIT, 1966