MASTER

The design and implementation for a multiprocessor, symmetrical, and hierarchical operating system

Badr El Din, A.H.M.

Award date:
1993

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
Eindhoven University of Technology.
Department of Electrical Engineering.
Digital Systems Group (EB).

THE DESIGN AND IMPLEMENTATION FOR
A MULTIPROCESSOR, SYMMETRICAL, AND
HIERARCHICAL OPERATING SYSTEM.

Master's Thesis by: A.H. Badr El Din


This report is submitted in partial fulfilment of the requirements for the degree
of Electrical Engineer (M.Sc.) at the Eindhoven University of Technology.

Coach: Ir. A.G.M. Geurts.

Supervisor: Prof. Ir. M.P.J. Stevens

The Department of Electrical Engineering of The Eindhoven University of
Technology does not accept any responsibility regarding the contents of student
project and graduation reports.
To my parents Dr. H. Badr El Din, and Dr. R. Agamiah
ACKNOWLEDGEMENTS

First of all, I would like to thank my parents (Dr. H.Badr El_Din, and Dr. R.Agamiah), for their incredible support and understanding. Without their support, patience, and understanding I would never have made it. A word of gratitude and respect here is not enough, since I will be heavily indebted to them forever.

I would like also to acknowledge Prof.Ir.M.P.J.Stevens for enabling me to work on the topic at all and for the overall guidance.

My special appreciation goes to my coach Ir.A.G.M.Geurts, for many useful and encouraging suggestions and discussions, and the reviewing of this report.

I would like also to thank my friend, and school mate, M.Sc. S.M.El_Kassas for his support and fruitful discussion.

Furthermore I want to express my feeling of gratitude towards all employees, and students of the digital systems group with whom I have been working together so pleasantly.

A.BADR_EL_DIN
Summary

An operating system may be viewed as the manager of a variety of hardware and software resources. The primary hardware resources are processors, memories, and I/O devices, while software resources constitute programs and data files. Due to the multitude of different functions with a high degree of complexity required in modern computer systems, operating systems became one of the most complex objects created by mankind. As a result their internal structure has played an important role not only during the development phase but also throughout their entire lifetime.

With multiprocessor operating systems also the processor organization played an important role. In this report we present the design and implementation of an operating system hierarchically structured, and multiprocessor symmetrically organized.

The key to a successful hierarchical design, lies in ordering the layers, so services needed to implement a given layer are only defined in layers beneath it.

Finally, the report contains nine chapters. In each chapter we explained the role of one layer in the system. A lot of work still needs to be done, in order to have a real working system. Since, in this report we introduced the concept, and the basic functions in each layer. Also we discussed some problems, we have to solve within each layer, and some possible solutions. System databases, also discussed, and simulations introduced in the appendixes.
# TABLE OF CONTENTS

## INTRODUCTION

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>i</td>
</tr>
<tr>
<td>Structure of operating system</td>
<td>i</td>
</tr>
<tr>
<td>Monolithic approach</td>
<td>ii</td>
</tr>
<tr>
<td>Kernel approach</td>
<td>ii</td>
</tr>
<tr>
<td>Hierarchical approach</td>
<td>ii</td>
</tr>
<tr>
<td>Object Oriented approach</td>
<td>ii</td>
</tr>
<tr>
<td>Definition of a multiprocessor</td>
<td>iii</td>
</tr>
<tr>
<td>Organization of a multiprocessor</td>
<td>iii</td>
</tr>
<tr>
<td>Master/Slave organization</td>
<td>iv</td>
</tr>
<tr>
<td>Separate executives</td>
<td>iv</td>
</tr>
<tr>
<td>Symmetric</td>
<td>iv</td>
</tr>
<tr>
<td>Target operating system</td>
<td>vi</td>
</tr>
</tbody>
</table>

## CHAPTER 1: LAYER 1 "FIRST LEVEL INTERRUPT HANDLER"

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 What is the problem</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Possible solutions</td>
<td>3</td>
</tr>
<tr>
<td>1.3.1 Simultaneous and nested interrupts</td>
<td>3</td>
</tr>
<tr>
<td>1.3.2 Interprocessor communication</td>
<td>4</td>
</tr>
<tr>
<td>1.3.2.1 Centralized daisy chain</td>
<td>4</td>
</tr>
<tr>
<td>1.3.2.2 Centralized polling</td>
<td>4</td>
</tr>
<tr>
<td>1.3.2.3 Centralized independent requests</td>
<td>4</td>
</tr>
<tr>
<td>1.3.2.4 Decentralized daisy chain</td>
<td>5</td>
</tr>
<tr>
<td>1.3.2.5 Decentralized polling</td>
<td>5</td>
</tr>
<tr>
<td>1.3.2.6 Decentralized independent requests</td>
<td>5</td>
</tr>
<tr>
<td>1.3.2.7 Comparison</td>
<td>5</td>
</tr>
</tbody>
</table>

## CHAPTER 2: LAYER 2 "THE DISPATCHER AND INTERPROCESS COMMUNICATION"

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Introduction</td>
<td>7</td>
</tr>
<tr>
<td>2.2 What is the problem</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 Process concept</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Process status</td>
<td>8</td>
</tr>
<tr>
<td>2.2.3 Process control block</td>
<td>8</td>
</tr>
<tr>
<td>2.2.4 Operations on processes</td>
<td>9</td>
</tr>
<tr>
<td>2.2.5 Interprocess communication</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Possible solutions</td>
<td>9</td>
</tr>
<tr>
<td>2.3.1 Data structures for processes and resources</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2 Interprocess communication</td>
<td>10</td>
</tr>
<tr>
<td>2.3.2.1 Semaphores</td>
<td>10</td>
</tr>
<tr>
<td>2.3.2.2 Message passing with rendezvous principle</td>
<td>11</td>
</tr>
<tr>
<td>2.3.2.3 Message passing using mailboxes</td>
<td>11</td>
</tr>
<tr>
<td>2.4 Implementation</td>
<td>11</td>
</tr>
<tr>
<td>2.4.1 Structure of a process and resource descriptor</td>
<td>11</td>
</tr>
<tr>
<td>2.4.2 Basic operations on processes and resources</td>
<td>14</td>
</tr>
</tbody>
</table>

## CHAPTER 3: LAYER 3 "MEMORY MANAGER"

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>16</td>
</tr>
<tr>
<td>3.2 What is the problem</td>
<td>16</td>
</tr>
<tr>
<td>3.3 Possible solutions</td>
<td>17</td>
</tr>
<tr>
<td>3.3.1 Fixed partitions</td>
<td>17</td>
</tr>
<tr>
<td>3.3.2 Variable partitions</td>
<td>17</td>
</tr>
<tr>
<td>3.3.3 Overlays</td>
<td>18</td>
</tr>
<tr>
<td>3.3.4 Virtual memory</td>
<td>18</td>
</tr>
<tr>
<td>3.3.4.1 Basic idea</td>
<td>19</td>
</tr>
<tr>
<td>3.3.4.2 Sharing with single copy and multiple copies</td>
<td>19</td>
</tr>
<tr>
<td>3.3.4.3 Possible solutions</td>
<td>20</td>
</tr>
</tbody>
</table>
INTRODUCTION

Introduction

An operating system may be viewed as the manager of a variety of hardware and software resources. The primary hardware resources are processors, memories, and I/O devices, while software resources constitute programs and data files. The operating system forms a layer of software that creates the illusion of a machine quite different from the actual underlying hardware. Most of the details of the hardware are hidden from ordinary users which instead provides a variety of commands and facilities to satisfy the users need.

Structure of operating system

Since operating systems are usually large and complex collections of software routines, the designers must place great emphasis on their internal organization and structure. In this section we examine briefly the various possible internal organizations of operating system

Monolithic approach

The most primitive approach for writing an operating system is the monolithic approach. Consisting of a set of programs that execute on the bare hardware. In such an operating system, modules serving different functions, may generally be distinguished. Control is being passed from one module to another by procedure calls and branches.

Due to the multitude of different functions that need to be performed in a modern computing systems, operating systems are becoming larger and more complex, as a result, their internal structure has played an important role not only during the development phase but also throughout their entire lifetime. A good structure is essential to allow modifications, and possible enhancements to be carried out easily. To cope with these requirements, the monolithic approach of early operating systems must be rejected.
Kernel approach

The philosophy underlying this approach is to concentrate on the design of an operating system kernel or nucleus, which embodies only the most vital low level functions performed by an operating system. It is a level of software that, like a monolithic operating system, hides the underlying hardware. However, it provides only a minimal set of operations from which the rest of the operating system may be constructed.

Hierarchical approach

At the core of the hierarchy is the computer hardware, from the next level up in the hierarchy are the various nucleus functions, and at the top of the hierarchy are the user processes. These levels are viewed as creating an extended machine.

Such hierarchical designs have proven easier to debug, modify, and implement. In design in which the nucleus itself is spread over several levels of the hierarchy, the choice of which nucleus function is to be placed at which level requires careful thought. Often in such designs the restriction is imposed that only downward calls are allowed.

Object oriented approach

In this approach the operating system is viewed as a collection of objects. A process is only one type of object: other possible object types may include procedures, storage pages, communication ports, hardware devices, synchronization primitives, and a variety of other concepts. Each object contains a set of rights that define the operations applicable to that object [LUBO88]. Interactions among objects are determined by capabilities, which may be viewed as pointers to objects.

The resulting structure of an object oriented operating system is a network of objects interconnected by capabilities. The principles of abstraction are also applied, each object itself embodies an abstraction of some concept: it hides the internal implementation of that concept and provides a set of operations applicable to the object.
Definition of a multiprocessor

A multiprocessor is defined in the American National Standard Vocabulary for Information Processing as "A computer employing two or more processing units under integrated control". Certainly the requirement that a multiprocessor have "integrated control" is extremely important, for a multiprocessor must have a single integrated operating system, however, the concepts of sharing and interaction, are not included in the ANSI definition.

It is the combination of sharing and interaction that completely characterizes the hardware and software required to provide a "true" multiprocessor, which can now be defined by the following system characteristics:

1- A multiprocessor contains two or more processors of approximately comparable capabilities.
2- All processors share access to common memory.
3- All processors share access to input/output devices.
4- The entire system is controlled by one operating system.
5- Symmetrical organization. The discussion about this organization will be postponed till next section.

Organization of a multiprocessor operating system

Although the facilities and services to be provided by the multiprocessor operating system are quite similar to those required for uniprocessors, the presence of more than one processing unit in the system introduces a new dimension into the design of the operating system.

There are three basic organizations that have been utilized in the design of operating systems for multiprocessors: master/slave, separate executive for each processor, and symmetric treatment of all processors.
Master/Slave organization

In this organization only one particular processor, the master processor, may execute the operating system. A slave processor can execute only user programs. When a process executing on a slave processor requires the attention of the operating system, it generates an interrupt, and waits for the master processor to handle the interrupt.

This type is certainly the easiest to implement and may often be produced by making relatively simple extensions to a uniprocessor operating system that includes multiprogramming capabilities. Although the master/slave type of system is simple, it is quite inefficient in its control and utilization of the total system resources, also the failure of the master processor causes a catastrophic system failure.

Separate executives

With separate executives organization, each processor has its own operating system and functions like a uniprocessor with its own resources, such as files and I/O devices. A process assigned to run on a particular processor, runs to completion on that processor. Each processor manages its own I/O devices independently of the other processors. I/O interrupts return directly to the processors that initiate them. Some tables contain information global to the entire system (for example, the list of processors known to the system). These tables must be carefully controlled using mutual exclusion techniques.

The separate executives organization is more reliable than the master/slave organization. The failure of a single processor is unlikely to cause a catastrophic system failure, but restarting the failed processor can be difficult. Reconfiguring the I/O devices on the system may involve switching the devices to different processors. This can be complex and require considerable manual effort.

Symmetric

Symmetrical multiprocessing is the most complex organization to implement, however it is the most powerful. All processors are identical. The operating system manages a pool of identical processors any one of which may be used to control any I/O device.

Because many processors may be executing the operating system at once, mutual exclusion are necessary. Because of the symmetry, it is possible to balance the work load more precisely than it is with the other organizations.
Symmetrical multiprocessing systems are generally the most reliable, a failure in one processor causes the operating system to remove that processor from its pool of available processors and notify the operator. The system may continue to function at reduced levels.

A process running on a symmetric multiprocessor system may be run at different time by any of the identical processors. The operating system floats from one processor to another. Careful design of system tables is essential, since, several processors may be in the supervisor mode at once. This called contention problem.
Target Operating System

It may be obvious after this introduction that a powerful operating system may be hierarchically structured, and multiprocessor symmetrically organized.

It is the design and implementation of such an operating system, I was assigned to fulfil, as a last requirement to earn the Dutch "Ir" (MS.c) degree, by the Digital Systems Division of The Department of Electrical Engineering from the Eindhoven University of Technology.

Fig 0, shows an overview of the layered operating system we will discuss throughout the rest of this report. Although, it is shown in its final form, it was built one layer at a time, for more details see [BADR89].

Each chapter contains three topics describes one layer. The three topics in each chapter are, an introduction to the functionality of the layer, the problems within that layer, and some possible solutions to the problems. In some chapters, the functions which are necessary to implement this layer are presented. Finally, at the end of the report (appendix) the reader will find a variety of algorithms, almost important for any operating system data structures, like binary trees, or queues implementation.
Chapter 1

Layer 1 "First Level Interrupt Handler"

1.1 Introduction

The first level interrupt handler is the part of the operating system which is responsible for responding to signals both from the outside world, and from the computer system itself.

When an interrupt occurs and gets served, the system saves the current state of the interrupted process, determines the cause of the interrupt, and initiates an appropriate interrupt handling routine. By finishing the interrupt handling routine, the system loads again the state of the most suitable process to run, and continues execution.

Basically, there are two classes of interrupts in a computer system: internal, and external interrupts.
Internal interrupts, often called traps, are generated within the CPU as a result of certain internal processor events. Traps may occur because of arithmetic exception conditions, such as overflow, underflow, and divide by zero operations. It may also occur as a result of program faults, such as page faults, protection violation or the execution of an illegal instruction. Hardware faults, such as memory parity errors and power failures, can also generate a trap. System calls also fall under this class of interrupts. Internal interrupts, can be accepted, delayed, or ignored.

External interrupts can be further classified as maskable interrupts and nonmaskable interrupts. Nonmaskable interrupts are considered the highest priority interrupts because they cannot be delayed, even if the CPU interrupt system is not enabled. Maskable interrupts are accomplished through the use of an interrupt enable flag associated with each device. When this flag is set by the CPU, the flag permits the interrupt issued by the corresponding device to be received by the CPU. Otherwise, the device interrupt request does not reach the CPU until the interrupt enable flag is set.

1.2 What is the problem?

The first problem we will address here, is who handle the interrupt. In case of internal interrupts, the interrupted processor, handle the interrupt. Since, internal interrupts depends on the process running on a processor. In case of external interrupts, any processor can handle the interrupt. In the next section we will discuss different kinds of interprocessor communication, in which processors communicate with each other to decide which is going to handle the external interrupt.

There are many occasions when an interrupt occurs while a previous one is being serviced. Sometimes, it is necessary to process this second interrupt immediately. This ability to allow interrupts to interrupt previous interrupt service routines safely, is called nested interrupts. The chance of more than one interrupt occurring at precisely the same instant (simultaneous interrupts), is not very high. However, the interrupt handling system is faced with the decision of which interrupt to service first.
In a multiprogrammed computer system, to have an efficient way of coping with different kinds of interrupts, which can occur nested or simultaneously. Different priority levels of interrupts are necessary because of the relative importance of processes waiting for an interrupt signal to awaken them, and because of the existence of timing constraints.

External interrupts can be nested, or simultaneously. However, internal interrupts can be only nested (page fault can occur while executing a system call).

In multiprocessor systems, processes communicate across processor boards. Consequently, processor communication mechanism (interprocessor communication) must exist. There are a number of reasons for a hardware interprocessor mechanism to be in a multiprocessing system. Since processors share memory, it is clearly possible to have software communication without explicit hardware assistance. Doing that, however, introduces a number of problems. First, all processors have to periodically check to see if there is a message for them. Second, since a processor only recognizes requests when it polls next, response times to requests can become intolerably long.

1.3 Possible solutions

1.3.1 Simultaneous and nested interrupts

The priority interrupts can take two forms. One is static priority interrupt vector, which is the levels of priority interrupts fixed at the writing phase of this layer. The second form is dynamic priority vector, which is the priority levels can be changed while the system is running. In both forms when an internal interrupt occurs, the new interrupt priority will be compared to the one being serviced. Depending on the highest priority interrupt, will get service first, and the other will be queued in the interrupted processor queue. Static priority interrupt levels is fair enough for internal interrupts. The other types of interrupts (external interrupt), will be discussed in the next section, since it is highly dependent on the interprocessor communication.
1.3.2 Interprocessor communication

In this section we will discuss six alternative schemes for controlling a common bus between processors [BOWE80]. The six alternatives are not exhaustive but, they are the most common. The logical distinction here is between centralized, and decentralized control. The mechanisms in either case for receiving requests, and for granting access are called daisy chain, polling, and independent requests.

1.3.2.1 Centralized daisy chain

Requests are raised on a common request line. A central control propagates a bus grant signal. The first processor which has signalled a bus request that receives the grant signal, stops its propagation, raises the busy line, and assumes bus control. On completion, it lowers the busy line and a new grant signal is generated, if further requests are outstanding.

1.3.2.2 Centralized polling

In this scheme the grant line is replaced by a set of wires to address all processors. On a request the controller sequences through the processors addresses. When a processor requesting access recognizes its address, it raises busy. Processor has access until it lowers busy. This technique could incorporate dynamic changes in priority allocation.

1.3.2.3 Centralized independent requests

This scheme requires two lines for each processor (grant, request). The controller can execute any programmed allocation algorithm. Clearly it is the most flexible, and dynamic changes in priority allocation can be programmed. A failed unit can be timed out and not cause system failure.
1.3.2.4 Decentralized daisy chain

Daisy chain is accomplished by creating a circular connection for the grant line. Thus when a processor raises a request, and it receives a grant pulse, it takes control of the bus. On completion it regenerates the grant pulse which circulates until blocked by another processor.

1.3.2.5 Decentralized polling

In this scheme, each processor is responsible for transmitting a polling code. A processor requiring access to the bus must wait until it recognizes its code. It then raises busy and uses the bus. On completion it lowers busy and transmits the next polling.

1.3.2.6 Decentralized independent requests

Each processor has a bus request line to which is assigned a priority. When busy lowered all processors examine all bus requests and the processor recognizing its own priority as highest, raises busy, and assumes control of the bus.

1.3.2.7 Comparison

Daisy chain
This is the simplest structure of all. Only few control lines are required and these are independent of the number of devices. However, priority is fixed by physical location in the centralized version, and round robin in the decentralized version. Furthermore, failure is catastrophic if any processor fails to function. Also, assignment is slow due to the ripple delay and is obviously dependent on the number of units in the chain.
Polling

Either approach to a polling requires a considerable increase in control hardware and bus lines with a resulting increase in complexity. A flexible priority allocation is possible. Another outstanding feature is a failed processor which does not respond to the poll is ignored. On the negative side, the number of processors is limited by the number of polling lines, and the polling rate must be slow enough to allow devices response, but high enough to allow first unit response.

Independent requests

This approach is potentially the fastest and most flexible in allocation. Failed devices can be ignored which yields a high immunity to unit failures. However, it requires a large number of control lines and in either case the controller is complex. Its speed and flexible allocation potential make it attractive, especially for multiprocessor systems.
CHAPTER 2

LAYER 2 "The DISPATCHER AND INTERPROCESS COMMUNICATION"

2.1 Introduction

An operating system achieves the illusion of concurrent processing by rapidly switching one processor among several processes. Because the speed of the processor is extremely fast compared to that of a human, the effect is impressive (many activities appear to proceed simultaneously).

The programs within this layer are responsible for maintaining the environment in which processes exist. Consequently they are required to operate on some form of data structure, which is the representation of all processes within the system. The nature of this data structure is the concern of this layer.

Context switching also lies at this layer. It consists of stopping the current process, saving enough information so it may be restarted later, and restarting another one.

The final part of this layer is the implementation of three forms of inter_process communication mechanism.
2.2 What is the problem?

2.2.1 Process concept

We introduce the concept of a process because it is the unit of work in our operating system.

Informally, a sequential process is the activity resulting from the execution of a program with its data by a sequential processor. However, a program by itself, is not a process. A program is a passive entity, while a process is an active entity. Many definitions have been given for a process. Till now, there is no universally agreed upon definition but the definition have just introduced seems to be most adequate.

In reality, two different processes might be sharing the same program or the same processor. The operating system is then viewed as a collection of processes, all running concurrently.

Each process is dedicated to a specific function, and its interactions with other processes are limited to only few defined interfaces.

2.2.2 Process states

A process goes through a series of discrete states during its existence. Various events can cause a process to change states. For example, a process is said to be running if it currently has the CPU, a process is said to be ready if it could use a CPU if one were available, or a process is said to be waiting if it is waiting for some event to happen before it can proceed. At this stage the extension to multiprocessing is not difficult. Only several processes will be running.

2.2.3 Process control block

The process control block is data structure containing important information about the process. It is a central store of information that allows the operating system to locate all information about a process. Thus the process control block is the entity that defines a process to the operating system.
The problem here is that processes control blocks need to be manipulated quickly by the operating system, because the performance of the system depends heavily on this structure.

### 2.2.4 Operations on processes

Systems that manage processes must be able to perform certain operations on processes. These include: create, destroy, suspend, resume, change priority, block, wakeup, and dispatch.

A process may create another process. If it does, the creating process is called the parent process, and the created process is called the child process. Such a creation yields to a hierarchical process structure, in which each child has only one parent, but each parent may have many children.

Destruction of a process is more complicated specially when the process has created other processes.

### 2.2.5 Inter_process communication

Processes frequently need to communicate with each other, preferably in a well structured way.

Processes run asynchronously with respect to each other. However, to achieve successful cooperation there are certain points at which processes must synchronise their activities. For example, when several processes asynchronously change the contents of a common data area, it is necessary to protect the data from simultaneous access and change by two or more processes. The updated area, may not, contain the intended changes if this protection is not provided.

The common data shared by several processes most often describes a resource. The resource could be some hardware elements, such as I/O devices, processors, or memory, or it could be software such as files.

### 2.3 Possible solutions

#### 2.3.1 Data structures for processes and resources

Processes and resources are normally represented by a data structure containing there state, identification, and accounting information.
Arrays are the simplest kind of data structure. The index of an array can be the process, or the resource identification, and each entry can be the information related to the process or the resource. Although, arrays allow direct access to arbitrary entries, they must have a fixed number of entries when the operating system programs is written.

Another technique avoids this difficulty, called linked lists. However, to access an element in a linked list, it must be sequentially accessed which gives overhead.

The question now is, when to use arrays and when to use linked lists. The answer might be, using arrays if it is possible to determine the number of entries, otherwise use linked lists. In the implementation section we introduce our data structure which is based on this answer.

2.3.2 Inter_process communication

2.3.2.1 Semaphores

In 1965 Dijkstra introduced two primitive operations that considerably simplified the communication and synchronization of processes. In their abstract form, these primitives designated P and V, operate on non_negative integer variables called semaphores. Let s be such a semaphore variable. The operations are defined as follows:

1- V(s): Increment s by 1 in a single indivisible action; the fetch, increment, and store cannot be interrupted, and s cannot be accessed by another process during the operation.

2- P(s): Decrement s by 1. If s=0 then it is not possible to execute P(s). The process invoking the P operation then waits until it is possible. The testing and decrementing of s are also an indivisible operation.
The semaphore variables are used to synchronize processes. The P primitive includes wait for the calling process, whereas V primitive may possibly activate some waiting processes. The indivisibility of P and V assures the integrity of the values of the semaphores.

2.3.2.2 Message passing with rendezvous principle

The basic idea is, when a process does a send, the system checks to see if the destination is waiting for a message from the sender. If so, the message is copied from the sender's buffer to the receiver's buffer, and both processes are marked as runnable. If the destination is not waiting for a message from the sender, the sender is marked as blocked and put onto a queue of processes waiting to send to the receiver.

When a process does a receive, the system checks to see if a process is blocked trying to send. If so, the message is copied from the blocked sender to the receiver, and both are marked as runnable. If no process is blocked trying to send to it, the receiver blocks until a message arrives.

2.3.2.3 Message passing using mailboxes

Process to process message passing using mailboxes has been designed to ensure that processes do not block while sending messages.

The basic idea is that, every process maintains a queue for messages sent to it. Messages are inserted at one end, and read from the other end. A process reading its mailbox either gets the oldest message, or an indication that the mailbox is empty.

2.4 Implementation

In this section we present a one possible way of implementing layer 2.

2.4.1 Structure of a process and resource descriptor

In Figure 2.2, and Figure 2.3, the process table protable is an array with entries for up to NPes processes. Each entry in protable is a structure named protab that defines the information kept for each process.
struct protab {
    int proqst; /*Process state:PRDY, PMAI, ... */
    int propr; /*Process Priority */
    int prgs[NREG]; /*Saved registers */
    int phase; /*Base of run time stack */
    int pslen; /*Stack length */
    int pslimit; /*Lowest extent of the stack */
    int pargs; /*Initial number of arguments */
    int pcadd; /*Initial code address */
    char *msgtxt; /*Pointer to message text if any */
    struct msg *hadd; /*Pointer to the first message sent*/
    struct msg *tadd; /*Pointer to the last message sent */
    struct msg *hbox; /*Pointer to the head of mailbox */
    struct msg *tbox; /*Pointer to the tail of mailbox */
};

struct protab protab[NProcs];

Figure 2.2: Process Structure

Figure 2.3: Process Table
The system uses the prost field to control the action of any process currently present in the system. Figure 2.4, shows the transition of a process between different states.

Figure 2.4: Process state diagram

```c
struct sempro {
    int psid;  /*Process ID in the semaphore queue*/
    struct sempro *next; /*Pointer to next process in queue*/
};

struct sestab {
    int semid; /*Semaphore ID*/
    int semux; /*Semaphore initial value*/
    int semiv; /*Semaphore current value*/
    int semet; /*Semaphore state:FREE, or BUSY*/
    struct sempro *head; /*Pointer to the head of the queue*/
    struct sempro *tail; /*Pointer to the tail of the queue*/
};

struct sestab sentable[NSEM];
```

Figure 2.5: Resource Table
As the process descriptor, the semaphore resource descriptor is also a table implemented as an array with entries for up to NSEM resources. Each entry is a structure of type semtable (Figure 2.5).

2.4.2 Basic operation on processes and resources

1-PID = Process_Create(table_arguments):
Create a process, and return its PID.

2-Status = Process_Destroy(PID):
Destroy a process, and return system status.

3-Status = Change_Priority(pid,priority):
Change a process priority, and return system status.

4-Status = Change_Status(pid,status):
Change a process status, and return system status.

5-SID = Sem_Create(Initial_value):
Create a semaphore, initialize it, and return SID.

6-Status = Sem_Destroy(SID):
Destroy a semaphore.

7-P(pid,sid):
Request some units of a resource.

8-V(pid,sid):
Release some units of a resource.

9-Send(pid,msg):
Send a message to a process.

10-Receive(pid,msg):
Receive a message from a process.

11-Status = Send_Mail(pid,msg):
Send a mail to a process.

12-MID = Read_Mail():
Read a process mailbox, and return current message MID.
13-Delete_Mail(MID):
   Delete a mail from a process mailbox.
3.1 Introduction

Memory is one of the most important resources in any computer system, that must be carefully managed. The part of operating system that manages memory is called the "memory manager". Its job is to keep track of which parts of memory are in use and which parts are not in use, to allocate memory to processes when they need it and to deallocate it when they are done, and to manage swapping between main memory and disk when main memory is not big enough to hold all the processes.

Memory management systems can be divided into two classes: those that move memory areas back and forth between main memory and disk during execution (swapping and paging), and those that do not. For many reasons specially on large computers with multiprogramming, the arguments in favour of swapping and paging class. The terms swapping and paging are used to describe the implementation of virtual memory and organization of main storage.

3.2 What is the problem?

Even though the cost of random access memory has been decreasing dramatically in recent years, there is still shortage of main memory in most computer systems. Two main problems result from this limitation. One is caused by the size of many applications, which simply exceed the amount of physical memory available in a given system. The second problem arises in multiprogramming systems, where several active processes need to share main
memory at the same time.

### 3.3 Possible solutions

To solve the problems just introduced, programs must somehow be divided into smaller blocks and loaded into memory only as needed. In this section we will introduce some schemes for partitioning main memory in order to solve the problems.

#### 3.3.1 Fixed partitions

The simplest scheme, found in some early systems, is that of fixed memory partitioning. In this scheme, memory is divided into a number of separate spaces, called partitions. Typically, partitions have different sizes to accommodate different programs, however, the sizes are fixed at system initialization and may not be changed while the system is operating. The determination of partition sizes has a strong impact on the system's efficiency. Unfortunately, it is very difficult to estimate the demands that will be placed on a system at future times, bad choices may lead to a severe underutilization of main memory or other resources. A related problem is process scheduling for each of the partitions.

Two possible schemes can be devised. In the first, a separate queue of processes exists for each partition. Typically, a process would be scheduled for the smallest partition that satisfies its memory requirements. This scheme is very simple to implement provided the requirements are specified a priori by each process. The obvious disadvantage is that partitions may remain unused if no processes of the appropriate sizes are available.

The second scheme, employs a single queue for all arriving processes. The assignment to partitions is then made by the operating system dynamically. While more complicated to implement, this scheme offers greater flexibility for the system to adapt to the current workload.

#### 3.3.2 Variable partitions

Instead of determining the partition sizes at system initialization, a variable partition scheme may be employed. As new processes arrive, the operating system assigns to each the exact amount of space required. Since processes do not normally terminate in the order of their arrival, memory will consist of a number of variable sized blocks,
only some of which will be occupied by active processes. Hence the main task of the operating system is to keep track of the free spaces and to assign these to newly arriving processes. It must also compact free spaces released by terminating processes into larger blocks to prevent fragmentation of main memory into spaces too small for any process to use.

A question related to fragmentation is: what should be done in a situation where an arriving process is too large to fit into any single free space currently available, but the total of two or more existing spaces would be sufficient. One possible solution is to let the process wait until a free space of sufficient size is created as other processes terminate. Another solution is to rearrange the current memory allocation to create sufficient contiguous space for the process. Obviously, a dynamic relocation scheme is required here, since the rearranged storage contains linked programs and data.

3.3.3 Overlays

The actual size of main memory imposes severe limitations in multiprogrammed systems. The largest possible space for programs or data is the largest partition. For computations that exceed the given limit, programs must be divided into smaller segments and loaded into memory separately.

In the simplest case, a technique called overlaying is employed, in which program segments overlay each other in memory as execution proceeds. The programmer is required to specify those parts of the program that must reside in memory simultaneously. Such a specification may be obtained from the calling structure (program tree), which describes the dependencies among procedures.

3.3.4 Virtual memory

The last three subsections have addressed two major issues in memory management:

(1) Sharing of main memory among different processes.

(2) Execution of programs larger than the available physical space.

With the memory management schemes discussed so far, possible solutions to both problems have been offered. However, the existing memory limitations were directly visible to the programmer and thus required that all programs be planned and designed according to the available physical storage space.
In this section we present the concept of virtual memory, which liberates the programmer from the limits of physical memory.

### 3.3.4.1 Basic idea

The basic idea of virtual memory is to hide the features of the real memory system from the user. In particular, virtual memories conceal the fact that physical memory is limited and, in the case of multiprogrammed systems, shared among different processes. The implementation of virtual memory creates the illusion that a process has one or more contiguous private spaces, each beginning at address zero. The sizes of such virtual address may be assumed unlimited.

To implement a virtual memory, three strategies must be resolved. These are placement, replacement, and load control. The placement strategy determines where in main memory to load all or part of virtual memory. A replacement strategy is necessary in systems that employ dynamic memory allocation policies. Here, is the decision of what to remove from physical memory when not enough space is available for a program or data that must be loaded.

Load control, is concerned with the policies when to load portions of virtual memory, and how much of it to load at any given time.

### 3.3.4.2 Sharing with single copy and multiple copies

Active processes often need the same code or data resident in main storage at the same time. When multiple copies of the same data file are updated, it is almost always necessary to have one consistent file that contains all updates. The problem here resembles the familiar critical section problem. Another problem occurs, if each process had its own copy, unnecessary system costs would be incurred, consisting of the I/O overhead in loading the excess copies and the memory to store them.

Sharing procedures with single copy, also requires care since instructions must contain addresses in association with the calling programs. For shared procedures to execute correctly under different processes, they must be assigned the same page numbers in all virtual address spaces.
3.3.4.3 Possible solutions

With procedure sharing, two approaches can be taken to permit each procedure to reference its own content correctly. The first is to require that all shared procedure pages or segments have the same numbers in all virtual spaces. The second approach is to use base registers that at execution time contain the segment or page numbers of the running process. The base registers are loaded by the system when the process is activated and may not be modified by the user. All references are then relative to a base register.

3.4 Implementation

3.4.1 Data structure

The memory management has three data structures, the first is for virtual memory pages, the second is for physical memory frames, and the third is for current processes. In Figure 3.2, one can see the relation between the virtual memory, and the physical memory. A frame in the virtual memory can be equal in size, to a physical memory page.

With the three data structures, the memory management can perform the basic operations.
3.4.2 Basic operations

1- Virtual addr = Mem Alloc(pid, size)
   Allocate memory for process PID, equal to size, and return
   the virtual address allocated.

2- Status = Mem Release(pid, size, start)
   Release memory allocated to a process, starting from start, and
   with size equal size.

3- Real address = Address Translate(virtual addr)
   Translate a virtual address to a real address.

4- Word = Get Word(virtual addr)
   Get the data in memory location, virtual address.

5- Status = Put Word(word, virtual addr)
   Put word in a virtual address, and return status.
CHAPTER 4

LAYER 4 "PROCESSOR MANAGER"

4.1 Introduction

One major task performed by an operating system is to allocate ready processes to the available processors. In a typical state of execution, the number of active processes able to run exceeds the number of available processors. The task of the processor manager, when invoked, is to examine the current allocation of processes to processors and, when appropriate, perform a reallocation according to some scheduling policy.

It is usually desirable to employ different allocation strategies for different types of processes. For example, we might wish to discriminate between processes associated with batch jobs and those of interactive users, between system and user processes, or between processes dealing directly with I/O and those for purely internal activities.

4.2 What is the problem?

4.2.1 Performance criteria

Different scheduling algorithms have different properties and may favour one class of processes over another. In choosing which algorithm to use in a particular situation, the properties of the various algorithms must be considered.

Many criteria have been suggested for comparing CPU scheduling.
algorithms. Which characteristics are used for comparison can make a substantial difference in the determination of the best algorithm. The criteria are:

1- CPU utilization: The idea here is to keep the CPU as busy as possible.
2- Throughput: It is the number of processes that are completed per unit time.
3- Turnaround time: It is the interval from the time of submission a job to the time of completion that job.
4- Waiting time: It is the time a process spends waiting in the ready queue.
5- Response time: It is the time interval from the submission of a request until the first response.

4.2.2 Preemptive Versus Non_Preemptive algorithms

1- Preemptive algorithms
   Once the CPU has been allocated to a process, the process can keep the CPU until it wants to release it, either by terminating or by requesting I/O.

2- Non_preemptive algorithms
   A scheduling discipline is preemptive if the CPU can be taken away from running processes.

An interactive timesharing systems, preemptive scheduling is important in guaranteeing acceptable response times. But, preemption is not without cost. Context switching involves overhead.

4.2.3 Multi_processor configuration

In multi_processor configuration, the scheduling problem is correspondingly more complex. One major factor is the types of processors involved. The processors may be identical (a homogeneous system) or different (a heterogeneous system). If the processors are specialised for certain kinds of tasks, the problem is relatively simpler. If several identical processors are available, then load sharing can occur. It would be possible to provide a separate queue for each processor. However, in this case, one processor could be idle, with an empty queue, while another processor is very busy.

4.3 Possible solutions
4.3.1 Selecting a criterion

It is desirable to maximize CPU utilization and throughput, and to minimize turnaround time, waiting time, and response time. In most cases, it is the average measure that is optimized. However, it may sometimes be desirable to optimize the minimum or maximum values, rather than the average.

It has also been suggested that for time sharing systems, it is more important to minimize the variance in the response time than to minimize the average response time. A system with reasonable and predictable response time may be considered better than a system which is faster on the average, but highly variable.

4.3.2 Preemptive scheduling algorithms

4.3.2.1 Shortest Remaining Time First

Shortest remaining time first algorithm associates with each process the length of its next CPU burst. When a new process arrives at the ready queue while the previous process is executing. The new process may have a shorter next CPU burst that what is left of the currently executing process. The algorithm will preempt the currently executing process, and start the next process with the shortest next CPU burst.

Shortest remaining time first is provable optimal in response time, but the real difficulty is knowing the length of the next CPU burst. The length of the next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts.

4.3.2.2 Priority scheduling

A priority is associated with each process. When a new process arrives at the ready queue its priority is compared with the priority of the currently running processes. The CPU is allocated to the process with the highest priority. Shortest remaining time first is simply a priority algorithm where the priority is the inverse of the predicted next CPU burst.

A major problem with priority scheduling is indefinite blocking or starvation. In a heavily loaded computer system, a steady stream of higher priority processes can prevent a low priority process from ever getting the CPU.

Aging is a solution to the problem of indefinite blocking. The technique is gradually increasing the priority of processes that
wait in the system for long time.

4.3.2.3 Round robin

A small unit of time, called a time quantum or time slice, is defined. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval up to a quantum in length.

The performance of the round robin depends heavily on the time quantum. At one extreme, if the time quantum is very large round robin is the same as first come first serviced. At the other extreme, if the time quantum is very small, round robin is called processor sharing and appears as if each of n processes has its own processor running at 1/n the speed of the real processor. The context switch overhead also depends on the time quantum.

A suggested rule is that 80 percent of the CPU bursts should be shorter than the time quantum [DEIT84], and [PETE85].

4.3.2.4 Multi level queues

A multi level queues scheduling algorithm partitions the ready queue into separate queues. Processes are permanently assigned to one queue. Each queue has its own scheduling algorithm. In addition, there must be scheduling between the queues.

This class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.

4.3.2.5 Multi level feedback queues

Multi level feedback queues allow a process to move between queues. In general, a multi level feedback scheduler is defined by the following parameters:

1- Number of queues.
2- The scheduling algorithm for each queue.
3- When to move a processes from one queue to another.
4- Which queue a process will enter when it needs service.
Although the multi level feedback queue is the most general scheme, it is also the most complex one.

4.3.3 Multi processor scheduling

To solve the problem introduced in section 4.2.3 because of the multi processor configuration. A common ready queue is used. All processes go into one queue and are scheduled onto any available processor.

In such a scheme one of two scheduling approaches may be used. One approach is for each processor to be self scheduling. Each processor examines the common ready queue and selects a process to execute. In this scheme we must insure that two processors do not choose the same process. The other approach appoints one processor as scheduler for other processors, and master/slave structure exists, which has been rejected early in this report, in the introduction chapter.

4.4 Implementation

The multilevel feedback scheduling provides n different priority levels. At each priority level p, there is a corresponding time \((T_p)\), which is the maximum amount of processor time any process may receive at that level. If a process exceeds \(T_p\), its priority is decreased to \(p-1\). One possible scheme is to let \((T_p=2^{(n-1)} \times T_n)\), where \(T_n\) is the maximum time on the highest priority queue, and \(n\) is the number of priority levels.

With this scheme, a process will spend a time \(T_n\) at level \(n\), \(2 \times T_n\) at level \(n-1\), \(4 \times T_n\) at level \(n-2\), and so on. Thus a process migrates to lower priority levels as it consumes more CPU time. When it reaches the lowest level 1, it remains there for a period of \(2^{(n-1)} \times T_n\) time units. Exceeding this time may either be interpreted as error, or this period could explicitly be set to infinity, allowing processes to remain at the lowest level indefinitely.

Each level may be either viewed as following a first in first out discipline, or processes at each level are sharing the processors in a round robin fashion.

Usually, each of the times \(T_p\) is a multiple of the basic quantum \(q\) [LUBO88], thus a process at level \(p\) receives \(T_p/q\) time quanta before it is moved to the next lower priority queue. The organization of the multilevel feedback queues is shown in Figure 4.2.
In our hierarchical operating system, processes in the hierarchical assigned a base priority when they are newly created, which specifies their minimum priority level. When a process is preempted, after receiving its fair share of the CPU usage, its current priority is decremented by 1, which places it into the next lower priority queue. The priority of a process fluctuates between its base priority and the highest priority. Processes dispatched strictly according to their current priority.
CHAPTER 5

LAYER 5 "INPUT/OUTPUT PROCESSES"

5.1 Introduction

One of the main functions of an operating system is to control all computer's input/output devices. It must issue commands to the devices, catch interrupts, and handle errors. It should also provide an interface between the devices and the rest of the system that is simple and easy to use. To the extent possible, the interface should be the same for all devices (device independence). The I/O code represents a significant fraction of the total operating system. How operating system manages I/O is the subject of this chapter.

5.2 What is the problem?

In addition to the main processors and primary memory, any computer installation must be equipped with devices that permit the user to interact with the system and secondary storage units, on which the user may keep information for indefinite periods of time.

By far the most popular device for interactive communication with a system is the terminals. These terminal have almost completely replaced non interactive devices such as punched card and paper tape units, used extensively in past decades. For hard copy output, a variety of printers, ranging from simple dot matrix to the most sophisticated laser printer, exist.
Roughly, I/O devices can be divided into two categories: block devices and character device [TANE87]. A block device is one that stores information in fixed size blocks, for example disks. The essential property of a block device is that it is possible to read or write each block independently of all other ones.

The other type of I/O devices is the character devices. A character device delivers or accepts a stream of characters, without regard to any block structure. Terminals, line printers, and most other devices that are not disk like can be seen as character devices.

This classification scheme is not perfect. Some devices just do not fit. Clocks, for example, are not block addressable. Nor do they generate or accept character streams. All they do is cause interrupts at well defined intervals.

The main issue in the design of I/O software is device independence. The first problem in designing I/O software addresses the fact that one must cope with a multitude of different I/O interfaces, not only for different device types, but even for different models within the same class of devices. The problem is further aggravated because of the requirement that the I/O software must be device independence. Furthermore, the number and type of devices may change as the computing facility is re-configured. It is essential that such changes remain transparent to all but a few special system programs. Because of most of the device independent software is part of the file system, in layer 6. We will postpone our discussion about device independent software to chapter 6. The reason for introducing device independent software here, is to provide some perspective on I/O and show better where the drivers fit in. The emphasis in this chapter is on device drivers.

Another problem is shareable versus dedicated devices. Some I/O devices, such as disks, can be used by many users at the same time. Other devices, such as printers, have to be dedicated to a single user until that user is finished. Introducing dedicated devices also introduces a variety of problems, including deadlock, and critical section.
The last problem we will deal with at this chapter, addresses the inefficiency which might be caused by improper service from storage devices such as disks to users. The process which is responsible for serving disk requests is called disk scheduling. Disk scheduling involves a careful examination of pending requests to determine the most efficient way to service the requests.

5.3 Possible solutions

5.3.1 Deadlock

Deadlock can be defined formally as follows. A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause. Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever.

Four necessary conditions that must be in effect for a deadlock to exist:

1-Mutual exclusion: Processes claim exclusive control of the resources they require.

2-Hold and wait: Processes hold resources already allocated to them while waiting for additional resources.

3-No preemption: Resources cannot be forcibly removed from the process holding them.

4-Circular wait: Each process holds one or more resources that are being requested by the next process in a chain.

In general, three strategies are used for dealing with deadlocks [COFF71]:

5.3.1.1 Deadlock prevention

The basic idea in this strategy is to impose suitable restrictions on processes so that deadlocks are structurally impossible. If we can ensure that at least one of the four necessary conditions is never satisfied, then deadlocks will be impossible.
5.3.1.2 Deadlock avoidance

An alternative strategy for dealing with deadlocks, is to require additional information about when resources are to be requested. With this knowledge of the complete sequence of resources requests and releases, we can decide for each request whether or not the requesting process should wait or granted the resource, depending on whether the request may lead to a deadlock situation in future (unsafe state), or will never lead to a deadlock (safe state).

5.3.1.3 Deadlock detection and recovery

If a system does not employ either a deadlock prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment the system must provide two algorithms. The first, is to examine the state of the system to determine whether a deadlock has occurred. The second algorithm is to recover from the deadlock.

An important question arise in deadlock detection algorithm which is, when should we invoke the algorithm. Deadlocks occurs, when some process makes a request which cannot be immediately granted. In the extreme, we could invoke the algorithm every time a request cannot be immediately granted. Of course, invoking the algorithm for every request may impose considerable overhead in computation time. A less expensive alternative might be simply to invoke the algorithm at less frequent intervals, for example once an hour, or whenever CPU utilization drops below a certain limit.

When a detection algorithm determines that a deadlock exists, the deadlock must be broken with the aid of the recovery algorithm. There are two options for breaking a deadlock. One would be simply to kill one or more processes in order to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.
5.3.2 Disk scheduling

Disk speed is composed of three parts. To access a block on the disk, the system must first move the head to the appropriate track. This head movement is called a seek, and the time to complete it, is seek time. Once the head is at the right track, it must wait until the desired block rotates under the read/write head. This delay is latency time. Finally, the actual transfer of data between the disk and main memory can take place. This last part is transfer time. The total time to service a disk request is the sum of the seek time, latency time, and transfer time.

For most disks, the seek time dominates, so reducing the average seek time can improve system performance substantially.

In multiprogrammed operating systems, many processes may be generating requests for reading or writing disk blocks. Because these processes often make requests faster than they can be serviced by the disk, waiting lines or queues build up for each device.

5.3.2.1 First come first served

In FCFS scheduling, the first request to arrive is the first one serviced. It is the simplest form of disk scheduling to implement. This algorithm is easy to program and fair in the sense that once a request has arrived, its place in the schedule is fixed. However, it may not provide the best service.

The problem with this schedule is the wild swing from one track to another depends on the arriving jobs request.

5.3.2.2 Shortest seek time first

It seems reasonable to service all requests close to the current head position together, before moving the head far away to service another requests. This assumption is the basis of the SSTF.
While SSTF algorithm provides a substantial improvement over FCFS, it is not optimal. It also may cause starvation of some requests. In theory, a continuous stream of requests near each other could arrive, causing the far requests to wait indefinitely.

5.3.2.3 Scan and c_scan

The idea in scan algorithm, is that, the read/write head starts at one end of the disk, moves toward the other end, servicing requests as it reaches each track, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed and servicing continues.

A variant of scan which is designed to provide a more uniform wait time is c_scan (circular scan). As with scan, c_scan moves the head from one end to the other, servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.

The descriptions of both scan and c_scan always move the head from one end to another. In practice, neither algorithm is implemented in this way. More commonly, the head is only moved as far as the last request in each direction. These versions of scan and c_scan are called look and c_look.
6.1 Introduction

The most visible part of any operating system is the file system. Most programs read or write at least one file, and users are always aware of the existence of files and their properties. For many people, the convenience and usability of the operating system is largely determined by the interface, structure, and reliability of the file system.

In this chapter we will look at various ways a file system can appear to its users, how file systems are implemented, and how files are protected against unauthorized usage.

6.2 What is the problem?

As mentioned above, the file system is the most visible part of any operating system, so we will divide the problem to two classes. A class of problems related to the user's requirements, and a class related to designer's requirements.
6.2.1 User's point of view

6.2.1.1 Device independence

From the user point of view, the most important aspect is, device independence.

It should be possible to write programs that can be used with files on any device, without having to modify the programs for each device type. Closely related to device independence is the goal uniform naming.

6.2.1.2 File protection

When information is kept in computer system, a major concern is its protection from both physical damage (reliability), improper access (protection), and correct results (integrity).

6.2.1.3 Access methods

Files store information. When it is used, this information must be accessed. There are several ways that the information in the file can be accessed. Some systems provide only one method, other systems support different methods. Choosing the right one for a particular application depends on what is needed.

6.2.1.4 Directory system

The file system should also provide directories, which, in many systems, are themselves files. It is quite common, that users want to group their files in logical ways.

6.2.2 Designer's point of view

6.2.2.1 Physical file organization

Space on secondary storage devices is organized into sequences of blocks called physical records, the record size is determined by the characteristics of the storage medium.
Several logical records may be mapped onto one physical record, or, conversely, one logical record may be spread over a number of physical records. A collection of physical records containing a logical file will be called a physical file. The designers task here, is the organization of physical files.

6.2.2.2 Management of storage space

The term storage management addresses the actual allocation and deallocation of file space, corresponding to the device independent request, issued by the user. An important part of this task is the management of free space. For example, space must be allocated if the file expands due to write operations.

6.2.2.3 Spooling

Another important problem is the spooling system. Spooling is a way of dealing with dedicated I/O devices. For example, a typical spooled device is the line printer. Suppose a process opened it and did nothing. No other process could print anything.

6.3 Possible solutions

6.3.1 Abstract user interface

An operating system usually provides an abstract interface that presents to the user a greatly simplified view of the computing environment. In this section we introduce a simple abstract interface consisting of the five operations that establish connections between files and users. Each of these operations must be interpreted by the file system and I/O processing modules and transformed into the necessary device specific commands.

Open\(F,op\_type\): This command specifies that the file \(F\) to be opened for the type of operation specified by \(op\_type\). In the case where \(F\) is not a file but an I/O device, the named device must be reserved for the user and also the symbolic name \(F\) must be mapped onto the actual hardware.
Close(F): This command reverses the effect of open, that is, it disconnects the file F from the current process.

Read(F,buf): The read operation makes use of an internal pointer, maintained by the system, that refers to the next logical record to be read. Each time a sequential read operation is issued, the contents of the current record are copied into the memory area designated by the address buf, and the pointer is updated to point to the next logical record.

Write(F,buf): Analogous to the read operation, the write operation transfers the contents of the memory area at address buf into the current logical record of file F, and updates the current record pointer accordingly.

Seek(F,rec_no): This command performs a direct access operation to a record specified by rec_no to the file F.

6.3.2 Protection

Protection mechanisms provide controlled access to files. Access is permitted or denied depending upon the user who is trying to access the file, and the type of access to the file.

Many systems recognize three classifications of users in connection with each file.

Owner: The user who created the file.
Group: A set of users who are sharing the file and need similar access.
Universe: All other users in the system.

For the type of access to the file, several different types of operations may be controlled:

Read: Read from the file.
Write: Write or rewrite the file.
Execute: Load the file into memory and execute it.
Delete: Delete the file and free its space.
With this classification and definitions, many different protection mechanisms have been proposed. As always each has its advantages and disadvantages, and must be selected as appropriate for its intended application.

6.3.3 Access methods

6.3.3.1 Sequential access

Information in the file is processed in order, one record after the other. This is by far the most common and simple mode of access of files. A read operation reads the next record of the file and automatically advances the file pointer. Similarly a write, appends to the end of the file, and advances to the end of the newly written record.

6.3.3.2 Direct access

For direct access, the file is viewed as numbered sequence of records. A direct access file allows arbitrary records to be read or written. Direct access files are of great use for immediate access to large amount of information.

6.3.3.3 Index access

This access method generally involve the construction of an index for the file. To find an entry in the file, we first search the index and then use the pointer to directly access the file desired entry.

6.3.4 Directory management

The most general directories organization is a tree structure, where each node of the tree is a directory (file), and each branch is a directory entry that points to either another directory or a data file. The root of the tree is called the master directory. A file name is formed by concatenating all branch identifiers found along the path from the master directory to the desired file.
While path names uniquely identify any file within the directory structure, their use would be within the directory structure, their use would be inconvenient if the user had to specify the full path name each time a file were to be accessed. In many systems, each user (process) is assigned a working directory, files are then referenced relative to the current working directory.

A link is an entry in a directory that points directly to another directory entry, rather than a file. Links may be established between directory entries to increase the efficiency of cross directory references.

6.3.5 Physical file organization

In this section we will describe four common organizations of physical files: contiguous, linked, index sequential, and B tree structured. Depending on the particular organization, the read and write operation of the abstract user interface must follow different strategies in determining the physical records that contain the logical record to be accessed.

6.3.5.1 Contiguous organization

A logical file may be mapped onto a sequence of adjacent physical records. The resulting physical file may be viewed as a contiguous sequence of secondary storage locations, starting at some address location. Records length can be either fixed or variable. In case of variable length, a length field directly precedes each record. With this organization sequential access and direct access (fixed length record) can be implemented. Under direct access, the record number is an integer between 0 and (number_of_records - 1) designating the desired record.
The main attraction of a contiguous organization of physical files is the simplicity with which both sequential and direct access may be accomplished. In the case of devices that permit only sequential access, such as magnetic tapes, contiguous organization is the only scheme that may be implemented efficiently.

The major problem with this organization is its inflexibility in deleting, inserting, and in case of variable length records, changing the length of a record. In all three cases, it is generally necessary to physically move all records following or preceding the location of the modification to preserve the file contiguity of allocation. Alternatively, insertions and deletions may be permitted at the end of a file. A related problem is the need to declare the maximum file length a priori. If too little space is allocated, the file may not be able to expand, too much space, on the other hand, results in wasted resources.

6.3.5.2 Linked organization

The logical records of a physical linked file are not, in general, stored in contiguous physical records. Instead, they may be scattered throughout the secondary storage device. Each record is linked to the logically next one by a forward pointer. If backspacing is desired, an additional backward pointer may be used.

The main advantage of a linked organization is the ease with which records may be inserted, deleted, and modified within the structure, thus expanding and contracting the file. Consequently, no upper limit must be imposed on the file length a priori, and the sizes of individual records can be easily changed during execution.

However, to access a record directly requires that the read or write operation follow the chain of pointers until the desired record is found, resulting, in some cases, an unacceptable overhead.
6.3.5.3 Index_sequential organization

The purpose of this organization is to permit direct access to records while eliminating the problems of insertion and deletion inherent to schemes with contiguous allocation of storage. This implies that logical records may be scattered throughout the secondary device. However, some mechanisms must exist for locating each record efficiently without scanning long chains of pointers.

These requirements can be accomplished by dividing the file into groups of logical records. Each record group is stored on a contiguous sequence of secondary storage location. Individual groups, however, may be scattered throughout the storage device arbitrarily. Finally, an index table is provided and ordered so that each entry contains the key of the first record within the group. The advantage of this method is that, for direct access to a record, the necessary sequential search is limited to only one group of records, rather than the entire file. Sequential access to records can also be implemented easily. The main difficulty with this organization is the need for periodic massive reorganization of the index structure, since the number of records comprising each group change as insertions and deletions are performed.

A variation of this scheme has been implemented in the UNIX operating system. Each file descriptor, or i_node in UNIX terminology, contains 13 entries describing the mapping of the logical file onto physical blocks of storage. The first 10 entries point directly to blocks holding the contents of the file, each block consists of 512 bytes. If more than 10 blocks are needed, the eleventh entry of the index table is used to point to an indirection block that contains up to 128 pointers to additional blocks of storage. The twelfth entry points by a twofold indirection to an additional 128x128 block. The last entry points to 128x128x128 blocks.
6.3.5.4 B_tree organization

B trees have become one of the most common techniques for organizing files on secondary storage. The basic assumption underlining this organization is that, index files are generally too large to be kept in main storage. Consequently, the primary objective is to minimize the number of accesses to the index structure.

A B_tree is organized as follows. Every node may be viewed as containing slots, each capable of holding one record key and pointers to nodes at the next lower level of the tree. The nodes are organized according to the following rule: for any given key K, the subtree pointed to by the left hand pointer contains only keys whose value is less than K, the subtree pointed to by the right hand pointer contains only keys whose value is greater than K. This determines a simple rule for locating a record given a key K. Insertion/deletion operation may require the splitting or collapsing of nodes on the path between the root and the affected node. The cost of these, however, may at most double the cost of a search, hence, the total cost of insertion and deletion still proportional to the logarithm of the file size.

6.3.6 Management of storage space

Important parts of the physical organization methods are the allocation strategies and schemes for keeping track of space on secondary memory, the space on a device is assumed to be shared among several files rather than entirely dedicated to one file. Not surprisingly, some of the techniques for main storage described in Chapter 3 are also applicable here.

The basic function of the storage allocator is to satisfy requests for fixed size blocks. This is because of the fixed length physical record organization of many storage media, the large amount of available storage, and the relative simplicity of such a policy.
There are several ways to keep track of the available space. The most obvious method is to link all free blocks together by pointers. This has the advantage of simplicity but it is severely inefficient. To add or remove free space from the lists, extensive I/O operation must be executed to find the appropriate number of blocks and to modify the pointers.

For this reason, it is more common to employ index table techniques. A separate table or set of tables containing all free space is maintained. Tables can be organized as linked lists. It is relatively easy to release and allocate blocks with this organization, but it consumes a large amount of storage. Much less table space is necessary if each block is represented by a single bit in the table: a 1 is interpreted as free block and a 0 as unavailable.

Since the storage is limited in capacity, facilities must be provided for moving interactive files off line. Typically, the bases for moving include the current filing load on the system, the priorities of the file users, the relative activities of the files, and the file sizes. In addition to file movement routines, there is a need for file copying to provide back_up capability.

6.3.7 Spooling directory

The problem of spooling can be solved by creating a special process, called a daemon, and a special directory, called a spooling directory. For the printer example given before in section 6.2.2.3, a process first generates the entire file to be printed, and puts it in the spooling directory. It is up to the daemon, which is the only process having permission to use the printer's special file, to print the files in the spooling directory. By protecting the special file against direct use by users, the problem of having some one keeping it open unnecessarily long is eliminated. Spooling is not only used for printers. It is also used in other situations. For example, file transfer over a network, often uses a network daemon.
CHAPTER 7

"NETWORK LAYER"

7.1 Introduction

This layer basically accepts messages from the host (layer 8, Transport layer), converts them into packets (the units of information exchanged at this layer), and sees that these packets get directed toward the destination [TANE81].

A key design issue is how the route is determined. Also, if too many packets are present at the same time, they will get in each other's way, forming bottlenecks. The control of such congestion also belongs to the network layer.

7.2 What is the problem?

The routing algorithm is that part of this layer responsible for deciding which output line an incoming packet should be transmitted on. Certain properties are desirable in the routing algorithm. Besides correctness and simplicity, the routing algorithm must be able to cope with changes in the topology and traffic without requiring all jobs in all hosts to be aborted and the network to be rebooted every time some node crashes. Stability is also an important goal for the routing algorithm.

Figure 7.1: Network Layer
7.3 Possible solution

Routing algorithms can be grouped into two classes: adaptive and nonadaptive. Nonadaptive do not base their routing decisions on measurements or estimates of the current traffic and topology, whereas adaptive ones do.  

7.3.1 Nonadaptive algorithms

7.3.1.1 Flooding

Flooding is one of the simplest routing algorithms. In which every incoming packet is sent out on every outgoing line except the one it arrived on. Flooding generates an infinite number of packets unless some measures are taken to damp the flow of packets. One such measure is to have a hop counter contained in the header of each packet, which is decremented at each hop, with the packet being discarded when the counter reaches zero. Ideally, the hop counter should be initialized the length of the path from source to destination. If the sender does not know how long the path is, it can initialize the counter to the worst case.

7.3.1.2 Static routing

Static routing is a simple algorithm and one of the most widely used. Each node maintains a table with one row for each possible destination node. A row gives the best, second best, etc. Before forwarding a packet, a node chooses among the alternatives using the weights as probabilities. The tables are worked out manually by the network operator, loaded into the nodes before the network is brought up, and not changed thereafter.

7.3.2 Adaptive algorithms

The main problem with nonadaptive algorithms, is just that they do not adapt to the changes in traffic. If the traffic levels in different parts changes dramatically and often, nonadaptive algorithms are unable to cope with these changes. Unfortunately, much computer traffic is bursty in nature. Adaptive algorithms are different in that sense, and can be divided into: centralized, isolated, and distributed.
7.3.2.1 Centralized

Centralized routing is similar to static routing in that, each node maintains a table telling how to forward a packet. The difference between static and centralized routing lies in how the routing tables inside the nodes are constructed.

In centralized routing, there is a routing control centre. Periodically, each node sends status information to the routing control centre. The control collects all this information, and then, based upon its global knowledge of the entire network, computes the optimal routes from every node to every other node in the network.

7.3.2.2 Distributed

In this class, each node periodically exchanges explicit routing information with each of its neighbours. Typically, each node maintains a routing table indexed by, and containing one entry for each other node in the network. This entry contains the preferred outgoing line to use for that destination, and some estimate of the time or distance to that destination. The metric used might be number of hops, estimated time delay, estimated number of packets queued along the path, or something similar.
8.1 Introduction

The transport layer has the task of providing a reliable and efficient transport service between user's processes rather than just between machines, like the network layer.

The most important single factor affecting transport protocol design is the kind of service provided by the network layer. If the network layer provides virtual circuit service, guaranteeing that messages are delivered in order from the sender to the receiver, without error, loss, or duplication, the transport protocol becomes relatively simple, because the network layer is in fact doing all the work. If, however, the network layer provides datagram service, it is up to the transport layer to make sure that messages are delivered in order, without error, loss, or duplication. This situation requires a much more sophisticated transport protocol.

Figure 8.1: Transport Layer
8.2 What is the problem?

8.2.1 Mapping

A service may be defined as a set of actions for customers. Some possible services are, time of day, file storage, and retrieval, and database management. Associated with each service is a process called a server, whose job is to provide the service to any authorized process that requesting the service. At this point, it is useful to distinguish between a name that is, which service the user wants, and an address, that is, which transport address the server is listening to. A name is generally, a character string intended for use by people rather than machines. Consequently, a mapping must be done between the name and the address.

8.2.2 Buffering

At the transport layer to keep a fast transmitter from overrunning a slow receiver, a solution is needed. Buffering is a solution to such a problem, however, there still remains the question of the buffer size. If most messages are nearly the same size, it is natural to organize the buffers as a pool of identical size buffers, with one message per buffer. However, if there is wide variation in message size, a pool of fixed sized buffers presents problems. If the buffer size is chosen equal to the largest possible message, space will be wasted whenever a short message arrives. If the buffer size is chosen less than the maximum message size, multiple buffers will be needed for long messages, with the attendant complexity.

8.2.3 Synchronization

Delayed duplicates occur when the routing algorithm gets into a loop, or when congestion causes a long delay. Either way, the sending host eventually times out and retransmits. If the new packet takes a different route it may arrive before the original, or after the original in both cases there will be a duplicate message. To solve such a problem, each host required to maintain a certain amount of history information.
8.3 Possible solution

8.3.1 Mapping

Although there are no general rules about how services are named, the naming being possibly local to each host or specific to each application, there are two general strategies for allocating transport addresses, hierarchical addresses, and flat addresses. With hierarchical addresses, the address consists of a sequence of fields, used to disjointly partition the space. With hierarchical addresses, knowledge of an address tells where the server is located.

With flat addresses, this is not the case. Flat addresses have no particular relationship to geography or any other hierarchy. In this way addresses would be unique but not related to location.

Both hierarchical and flat addressing have advantages and disadvantages. Hierarchical addressing makes routing easy. Another advantage is the ease of creating new ports. Hierarchical addressing also has some disadvantages. If a process migrates to new machines, its address must be changed. Flat addressing has just the reverse properties from hierarchical addressing.

8.3.2 Buffering

In the previous section we discussed the buffer size problem. An approach to this problem is to use variable size buffers. The advantage here is better memory utilization, at the price of far more complicated buffer management. Another possibility is to dedicate a single large circular buffer per connection. This also makes good use of memory, provided that all connections are heavily loaded, but it is poor if some connections are lightly loaded.
8.3.3 Synchronization

The delayed duplicate problem can be attacked in various ways. One way is to use throwaway transport addresses. In this approach, each time a transport address is needed, a new unique address is generated, typically based on the current time. When a connection is closed, the addresses are discarded forever. A variation is merely to forbid the same combination of address to be linked up again by making the sequence number so large that it never recycles.

Another strategy is based on ensuring that no packet lives longer than some known time, by putting a hop counter in each packet. In this method data link protocol simply discard any packet whose hop counter has exceeded a certain value.
CHAPTER 9

LAYER 9 "USER INTERFACE"

9.1 Introduction

Globally, a user interface is the hardware and software with which users interact to specify a computer problem, and observe the results. Restricted to the operating system context, the term usually refers to the interactive software that accepts commands from the user and carries out the processing they specify. Thus, the user interface lies between a human who specifies a computer problem and a system capable of carrying it out.

The goal of user interface design is to create an environment in which users can solve their computer problems, easily, and efficiently.

9.2 What is the problem?

If the goal of the interface design is to make users easily, and efficiently solve their computer problems. Then, the design will be difficult. First, because humans have individual preferences, and talents, we cannot expect universal agreement on simple. Second, because users employ computing systems to solve a wide variety of problems, we cannot expect universal agreement about efficiency.
9.3 Possible solution

Although no single user interface will be best for all users, we can establish a few principles that help guide our design decision. Here are four such principles.

1- The users should have access to all facilities provided by the operating system.

2- There should be exactly one way to perform a command.

3- The users should be able to predict the output and syntax of any command.

4- The users should be able to change the syntax of any command easily.

The UNIX system has become quite popular, running on machines of varying power from microprocessors to mainframes. By the beginning of 1984, there were about 100,000 UNIX system installations in the world. Because of the popularity of UNIX, we choose the UNIX specific style of user interface.
CONCLUSIONS AND RECOMMENDATIONS

This report is a guide to the design and implementation of a layered, and symmetrical multiprocessor operating system.

In each chapter we explained the role of one layer in the system. The key to a successful design lies in ordering the layers, so services needed to implement a given layer are only defined in layers beneath it.

A lot of work still needs to be done, in order to have a real working system. Since, in this report we introduced the concept, and the basic functions in each layer. Also we discussed some problems, we have to solve within each layer, and some possible solutions. System databases, also discussed in this report, and simulations introduced in the appendixes.

Since we gave at each layer a variety of solutions, we opened the discussion more about the implementation of each layer. Fortunately, we are expecting some people with, and some against, each with a different point of view. Here, is the role of the layered approach, since changing and modifying, is relatively an easy task.

We have two recommendations to make. First, an interest reader might think about real time processes, which we passed through, in chapter 4. Second, is based on the unit of work in our operating system, which is a process. In order to reach a higher degree of parallelism, and more speed, an interest reader might think about threads instead of processes.

Finally, I would like to urge the reader not to regard this report as in any way a definitive study of operating system. There are many excellent papers, and books which will give fresh insight to the material covered here. It is hoped that the references given will provide an adequate starting point.
APPENDIX A: Processes and Resources Data Structure

#include <stdio.h>

/* General constants */
#define NPes 10 /* Number of processes available in the system */
#define NPor 2 /* Number of processors available in the system */
#define NREG 5 /* Number of registers */
#define SYSERR -1 /* Error has been detected */
#define SYSOK 0 /* No error detected */

/* Semaphore constants */
#define NSEM 50 /* Number of semaphores available in the system */
#define FREE 0 /* This semaphore is free and can be allocated */
#define BUSY -1 /* This semaphore is busy and can not be allocated */
#define RDYQ 0 /* Semaphore 0 is the ready queue */

/* Process status */
#define PRFR 0 /* Process slot is free and can be allocated */
#define PRDY 1 /* Process is waiting in the ready queue */
#define PWAIT 2 /* Process is waiting in one of the semaphore queues */
#define PRUN 3 /* Process is running in one of the processors */
#define PTER 4 /* Process is terminating */
#define PBOS 5 /* Process blocked trying to send a message */
#define PBOR 6 /* Process blocked trying to receive a message */

/* Message structure */
struct msg {
  int bpid;  /* Blocked process id */
  char *msg;  /* Pointer to the message text */
  struct msg *nextm;  /* Pointer to the next message */
};

/* Process table */
struct protab {
  int prost;  /* Process state:PRDY,PWAIT,.... */
  int propr;  /* Process priority */
  int preg[NREG];  /* Saved registers */
  int pbase;  /* Base of run time stack */
  int pslen;  /* Stack length */
  int plimit;  /* Lowest extent of the stack */
  int pargs;  /* Initial number of arguments */
  int pcadd;  /* Initial code address */
  char *msgtxt;  /* Pointer to the message text */
  struct msg *hadd;  /* Pointer to the first message sent */
  struct msg *tadd;  /* Pointer to the last message sent */
  struct msg *hmbox;  /* Pointer to the head of mail box */
  struct msg *tmbox;  /* Pointer to the tail of mail box */
};
struct protab protable[NPes];

/* Semaphore table */

struct sempro {
    int prsid; /* Process id in the semaphore queue */
    struct sempro *next; /* Pointer to next process in the queue */
};

struct semtab {
    int semid; /* Semaphore id */
    int semmx; /* Semaphore max value */
    int semvl; /* semaphore current value */
    int semst; /* Semaphore state:FREE or BUSY */
    struct sempro *head; /* Pointer to the head of the queue */
    struct sempro *tail; /* Pointer to the tail of the queue */
};

struct semtab semtable[NSEM];
APPENDIX B: Semaphores

```c
signal(sid, pid)
int sid;
int pid;
{
    if (countqueue(sid) == 0)
        (semtable[sid].semvl)++;
    else
    {
        switch(sid)
        {
            case RDYQ: protable[semtable[sid].head->prsid].prost = PRUN;
                break;
            default : protable[semtable[sid].head->prsid].prost = PRDY;
                tailinsert(RDYQ, semtable[sid].head->prsid);
                break;
        }
        headdelete(sid);
    }
}

wait(sid, pid)
int sid;
int pid;
{
    int i;
    if (semtable[sid].semvl > 0)
    {
        if (sid == RDYQ)
            protable[pid].prost = PRUN;
        (semtable[sid].semvl)--;
    }
    else
    {
        switch(sid)
        {
            case RDYQ: protable[pid].prost = PRDY;
                break;
            default : protable[pid].prost = PWAIT;
                tailinsert(RDYQ, pid);
                break;
        }
        tailinsert(sid, pid);
    }
}
screate(svl)
int svl;
{
    int i;
}
```
for (i=0;i<NSEM;i++)
    if (semtable[i].semst == FREE)
        { 
            semtable[i].semst = BUSY;
            semtable[i].semmx = svl;
            semtable[i].semmv = svl;
            semtable[i].semid = i;
            semtable[i].head = (struct sempro *)malloc(sizeof(struct sempro));
            semtable[i].head = semtable[i].tail = NULL;

            return(semtable[i].semid);
        }
    return(SYSERR);
}
sdelete(sid)
int sid;
{
    struct sempro *temp;

    semtable[sid].semmv = 0;
    semtable[sid].semst = FREE;
    while (semtable[sid].head != NULL)
        { 
            temp = semtable[sid].head;
            semtable[sid].head = (semtable[sid].head)->next;
            free(temp);
        }
}
tailinsert(sid,pid)
int sid;
int pid;
{
    struct sempro *pt;
    pt = (struct sempro *)malloc(sizeof(struct sempro));
    pt->prsid = pid;
    pt->next = NULL;
    switch ( semtable[sid].tail )
        { 
            case NULL:semtable[sid].tail=semtable[sid].head = pt;
                break;
            default :(semtable[sid].tail)->next=pt;
                semtable[sid].tail=(semtable[sid].tail)->next;
                    break;
        }
}
headdelete(sid)
int sid;
{
    struct sempro *temp;

    temp = semtable[sid].head;
    if(sid==RDYQ)
        protab[temps->prsid].prost = PRUN;

    semtable[sid].head = (semtable[sid].head)->next;
    if(semtable[sid].head == NULL)
        semtable[sid].tail = NULL;

    free(temp);
printall(lim)
int lim;
{
  int i;
  for (i=0;i<lim;i++)
    if (semtable[i].semst == BUSY)
      {
        printf("semid=\[%d\] semvl=\[%d\] queue:\n", i, semtable[i].semvl);
        printqueue(i);
      }
}

printqueue(sid)
int sid;
{
  struct sempro *qhead;
  qhead=semtable[sid].head;
  while(qhead != NULL)
    {
      printf("pid=\[%d\] pst=\[%d\]->\n", qhead->prsid, protable[qhead->prsid].prost);
      qhead = qhead->next;
    }
  printf("NULL\n");
}

coutqueue(sid)
int sid;
{
  int count;
  struct sempro *qhead;
  count=0;
  qhead=semtable[sid].head;
  while(qhead != NULL)
    {
      count++;
      qhead = qhead->next;
    }
  return(count);
}
APPENDIX C: Message Passing Using RENDEZVOUS Principle

sendmsg(spid, rpid, msgt)
    int spid;
    int rpid;
    char *msgt;
    
    if( (protable[rpid].prost == PBOR) && (foundmsg(rpid,protable[spid].hadd) != 0))
        
        protable[rpid].msgtxt = (char *)malloc(strlen(msgt)+1);
        copystr(protable[rpid].msgtxt,msgt);
        protable[rpid].prost = PRDY;
        wait(RDYQ,rpid);
        deletemsg(rpid,protable[spid].hadd,spid);
    
    else
        
        protable[spid].msgtxt = (char *)malloc(strlen(msgt)+1);
        copystr(protable[spid].msgtxt,msgt);
        protable[spid].prost = PBOS;
        insertmsg(spid,msgt,rpid);
        signal(RDYQ,spid);
    
receivemsg(spid, rpid, msgt)
    int spid;
    int rpid;
    char *msgt;
    
    if( (protable[rpid].prost == PBOS) && (foundmsg(rpid,protable[spid].hadd) != 0))
        
        protable[spid].msgtxt = (char *)malloc(strlen(protable[rpid].msgtxt)+1);
        copystr(protable[spid].msgtxt,protable[rpid].msgtxt);
        free(protable[rpid].msgtxt);
        protable[rpid].msgtxt = NULL;
        wait(RDYQ,rpid);
        deletemsg(rpid,protable[spid].hadd,spid);
    
    else
        
        msgt = NULL;
        protable[spid].prost=PBOR;
        insertmsg(spid,msgt,rpid);
        signal(RDYQ,spid);
    
rec_msg_any(spid, msgt)
    int spid;
    char *msgt;

C-1
int rpid;
if (protable[spid].hadd == NULL)
    rpid = -1;
else
{
    rpid = (protable[spid].hadd)->bpid;
    receivemsg(spid, rpid, msgt);
}
return(rpid);

insertmsg(pid, msgt, inpid)
int pid;
char *msgt;
int inpid;
{
struct mesg *temp;
    temp = (struct mesg *)malloc(sizeof(struct mesg));
    temp->bpid = pid;
    temp->msg = msgt;
    temp->nextm = NULL;
switch (protable[inpid].tadd)
{
    case NULL: protable[inpid].hadd = protable[inpid].tadd = temp;
        break;
    default : (protable[inpid].tadd)->nextm = temp;
        protable[inpid].tadd = (protable[inpid].tadd)->nextm;
        break;
}
}

deletemsg(pid, addr, cpid)
int pid;
struct mesg *addr;
int cpid;
{
struct mesg *temp;
    if (addr == NULL)
        return(SYSERR);
    else
    {
        if (addr->bpid == pid)
            hdeletemsg(cpid);
        else
            while(addr->nextm != NULL)
            {
                if (addr->nextm->bpid == pid)
                {
                    temp=addr->nextm;
                    addr->nextm = addr->nextm->nextm;
                    free(temp);
                }
                else
                    addr = addr->nextm;
            }
    }
}

foundmsg(pid, addr)
int pid;
struct mesg *addr;
{
int found;

found=0;
while ((addr != NULL) && (!found))
  if (addr->bpid == pid)
    found = 1;
  else
    addr = addr->nextm;
return(found);
}

hdeletemsg(pid)
int pid;
{
  struct msg *temp;
  temp = protab[pid].hadd;
  protab[pid].hadd = (protab[pid].hadd)->nextm;
  if (protab[pid].hadd == NULL)
    protab[pid].tadd = NULL;
  free(temp);
}

copystr(estr,fstr)
char *estr;
char *fstr;
{
  while (*fstr != '\0')
  {
    *estr = *fstr;
    estr++;
    fstr++;
  }
}
APPENDIX D: Message Passing Using MAILBOXES

sendmail(spdi, rpdi, mail)
    int spid;
    int rpdi;
    char *mail;
    {
        struct msg *temp;

        temp = (struct msg *)malloc(sizeof(struct msg));
        temp->bpdi = spid;
        temp->msg = mail;
        temp->nextm = NULL;
        switch(protable[rpdi].tmbox)
        {
            case NULL: proteble[rpdi].hmbox=protable[rpdi].tmbox=temp;
                       break;
            default : (protable[rpdi].tmbox)->nextm=temp;
                      proteble[rpdi].tmbox=(protable[rpdi].tmbox)->nextm;
                       break;
        }
    }

deleteemail(spdi, mid)
    int spid;
    int mid;
    {
        struct msg *temp1,*temp2;
        int i;

        temp1 = proteble[spid].hmbox;
        switch(mid)
        {
            case 1 : temp2 = proteble[spid].hmbox;
                     proteble[spid].hmbox==(protable[spid].hmbox)->nextm;
                     if(protable[spid].hmbox == NULL)
                     proteble[spid].tmbox = NULL;
                     free(temp2);
                     break;

            default :for(i=1;i<(mid-1);i++)
                     temp1=temp1->nextm;
                     temp2 = temp1->nextm;
                     temp1->nextm = temp1->nextm->nextm;
                     free(temp2);
                     break;
        }
    }

reademail(spid)
    int spid;

D-1
{ 
  struct msg *temp;
  int count;

  count=0;
  temp = protable[spid].hmbox;
  while(temp != NULL) {
    count++;
    printf("mail id=%d\tsender process id=%d\t mail=%s
",count,temp->bpid,temp->msg);
    temp=temp->nextm;
  }
  printf(" end of mail box\n");
}
APPENDIX E: READING LIST AND BIBLIOGRAPHY:


[COFF71] Coffman E.G, Elphick M.J, and Shoshani A.
"System Deadlocks ".
Computing Surveys, June 1971, Volume 3, Number 2.

[COME84] Comer D.
"Operating System Design The XINU Approach. VOLUME 1 & 2 ".

[DEIT84] Deitel H.M.
"An Introduction To Operating Systems ".

[DENN80] Denning P.J.
"Working Sets Past And Present ".

[ENSL77] Enslow P.H.
"Multiprocessor Organization A Survay"

[FATH83] Fathi E.T, and Krieger M.
"Multiple Microprocessor Systems:What, Why, And When"
IEEE Computer, March 1983.

[FISH89] Fisher S.
"Switching To Token Ring Networks ".
Unixworld, August, 1989.

[HANS70] Hansem B.
"The Nucleus Of A Multiprogramming System ".

[HANS72] Hansem B.
"Structured Multiprogramming ".

[HOLT83] Holt R.C.
"Concurrent EUCLID, The UNIX System, and TUNIS ".

[JANS86] Janssens M.D, Annot J.K, and Van De Goor A.J.
"Adapting UNIX For A Multiprocessor Environment ".
[JOSE84] Joseph M, Prasad V.R, and Natarajan N.  
"A Multiprocessor Operating System ".  

[JURC77] Jourca I.  
"A Multiprocessor System With Multitasking Facilities ".  
Dutch Efficiency Bureau, 1977.

[KELL84] AL Kelley and Pohl Ira.  
"An Introduction To Programming In C: A Book On C "  

[KENA84] Kenah L.J. and Bate S.F.  
"VAX/VMS Internals And Data Structures "  

[KERN88] Kernighan B.W and Ritchie D.M.  
"The C Programming Language "  

[KRUS89] Kruse R.L  
"Programming With Data Structures "  

[LION77] Lion J.  
(1) UNIX Operating System Source Code Level 6.  
Department Of computer Sciences, University of New South Wales, 1977.

[List78] Lister A.M.  
"Fundamentals Of Operating Systems "  

[LUBO88] Lubomir B.  
"The Logical Design Of Operating Systems"  
[MEBI86] Mebius J.E.
"Fairness Aspects Of Dijkstra's Semaphore Concept ".
Report Number 8659, Delft University, 1986.

[MEBI88] Mebius J.E.
"A Generalisation Of A.N.Habermann's Theorem for Semaphores ".

[MEIJ83] Meijer A, and Peeters P.
"Computer Network Architectures "

[OUST88] Ousterhout J.K.
"The Sprite Network Operating System "

[PETE85] Peterson J.L, and Silberschatz A.
"Operating System Concepts ".

1- Machine Independent Virtual Memory Management For Paged Uniprocessor And Multiprocessor Architectures ".

2- The Duality Of Memory And Communication In The Implementations Of A Multiprocessor Operating System.

[RUSS77] Russo P.M.
"Interprocessor Communication For Multi Microprocessor Systems"
IEEE Computer, April,1977.

[RUSS87] Russell C.H, and Waterman P.J.
"Variations On UNIX For Parallel Processing ".
Communication of the ACM, December 1987, Volume 30, Number 12.

"Process Management And Exception Handling In Multiprocessor Operating System Using Object Oriented Design Techniques ".
[SAAD89] Saad Y. and Schultz H.
"Data Communication In Parallel architectures"

[SAXE76] Saxena A.
"A Verified Specification Of A Hierarchical Operating Systems".
University Microfilms International, Ann Arbor, Michigan USA, 1976.

[SCHI88] Schildt H.
"Advanced C, Second Edition"

[TANE81] Tanenbaum A.S.
"Computer Networks".

[TANE84] Tanenbaum A.S.
"Structured Computer Organization".

[TANE87] Tanenbaum A.S.
"Operating Systems Design And Implementation".

[TSIC74] Tsichritzis D.C. and Bernstein P.A
"Computer Science And Applied Mathematics".

[TURN86] Turner R.W.
"Operating Systems Design And Implementations".

[WELT89] Welten P.J.M.
"An Investigation Into Multiprocessor Systems Based On UNIX".

[WILL86] Willis N.
"Computer Architecture And Communications"