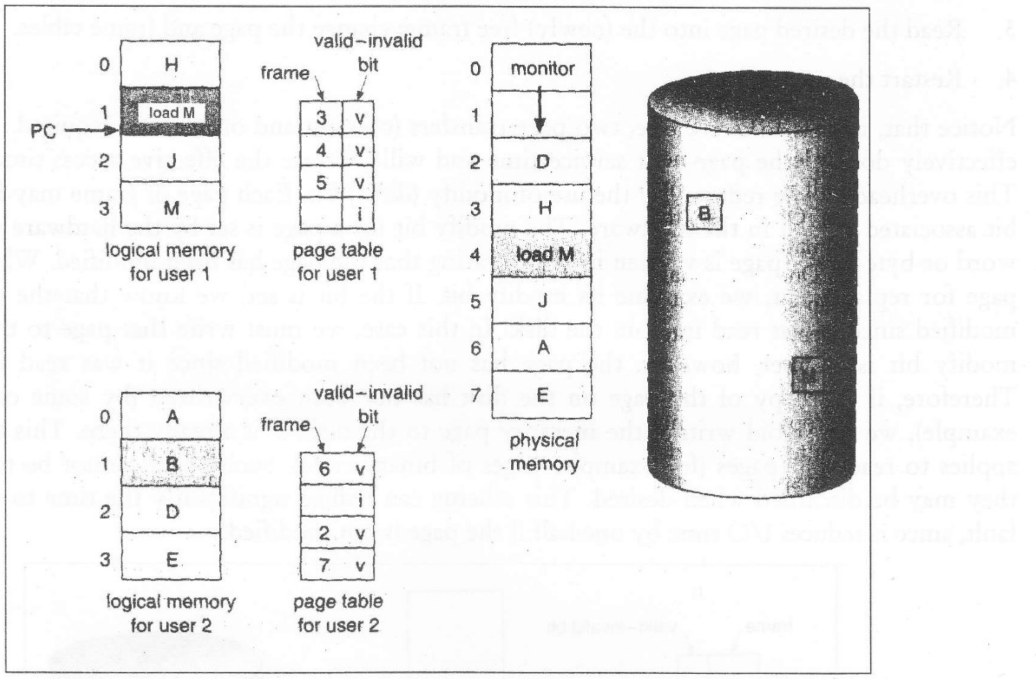

**Figure 3.19: Need for Page Replacement**

Over-allocating will show up in the following way. While a user process is executing, a page fault occurs. The hardware traps to the operating system, which checks its internal tables to see that this is a page fault and not an illegal memory access. The operating system determines where the desired page is residing on the disk, but then finds there are no free frames on the free-frame list; all memory is in use (Figure 3.19).

The operating system has several options at this point. It could terminate the user process. However, demand paging is something that the operating system is doing to improve the computer system's utilization and throughput. Users should not be aware that their processes are running on a paged system. Paging should be logically transparent to the user. So, this option is not the best choice.

We could swap out a process, freeing all its frames, and reducing the level of multi-programming. This option is a good idea at times. Let us discuss one intriguing possibility: page replacement.

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space, and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 3.20). The freed frame can now be used to hold the page for which the process faulted. The page-fault service routine is now modified to include replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
   (a) If there is a free frame, use it.
   (b) Otherwise, use a page-replacement algorithm to select a victim frame.
   (c) Write the victim page to the disk; change the page and frame tables accordingly.

3. Read the desired page into the (newly) free frame; change the page and frame tables.
4. Restart the user process.

Notice that, if no frames are free, two page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and will increase the effective access time accordingly. This overhead can be reduced by the use of modify (dirty) bit. Each page or frame may have a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write that page to the disk. If the modify bit is not set, however, the page has not been modified since it was read into memory. Therefore, if the copy of the page on the disk has not been overwritten (by some other page, for example), we can avoid writing the memory page to the disk; it is already there. This technique also applies to read-only pages (for example, pages of binary code). Such pages cannot be modified; thus, they may be discarded when desired. This scheme can reduce significantly the time to service a page fault, since it reduces I/O time by one-half if the page is not modified.

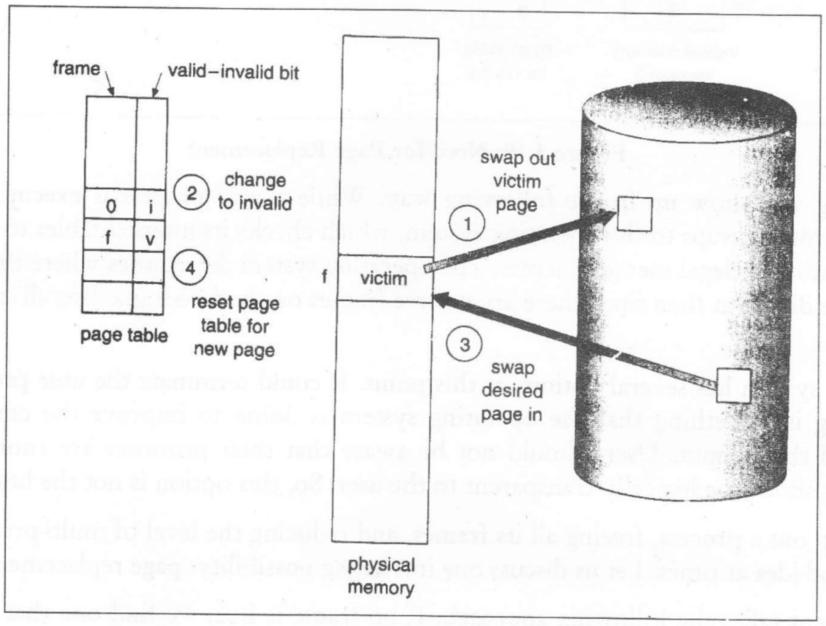

**Figure 3.20: Page Replacement**

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, a very large virtual memory can be provided to programmers on a smaller physical memory. With non-demand paging, user addresses were mapped into physical addresses, allowing the two sets of addresses to be quite different. All of the pages of a process still must be in physical memory, however. With demand paging, the size of the logical address space is no longer constrained by physical memory. If we have a user process 20 pages long, we can execute it in 10 frames simply by using demand paging, and using a replacement algorithm to find a free frame whenever necessary. If a page that has been modified is to be replaced, its contents are copied to the disk. A later reference to the pages will cause a page fault. At the time, the page will be brought back into memory, perhaps replacing some other page in the process.

We must solve two major problems to implement demand paging: We must develop a frame-allocation algorithm and a page-replacement algorithm. If we have multiple processes in memory, we must decide how many frames to allocate to each process. Further, when page replacement is required, we must select the frames that are to be replaced. Designing appropriate algorithms to solve these problems is an important task, because disk I/O is so expensive in terms of time. Even slight improvements in demand-paging methods yield large gains in overall system performance.

*Page-Replacement Algorithms*

Many different page-replacement algorithms exist. Probably every operating system has its own unique replacement scheme. How do we select a particular replacement algorithm? In general, we want the one with the lowest page-fault rate. We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a reference string. We can generate reference strings artificially (by a random-number generator, for example) or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data (on the order of 1 million addresses per second). To reduce the number of data, we note two things.

First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, not the entire address. Second, if we have a reference to page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.

For example, if we trace a particular process, we might record the following address sequence:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,

0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105,

which, at 100 bytes per page, is reduced to the following reference string?

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1.

To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. Obviously, as the number of frames available increases, the number of page faults will decrease. For the reference string considered previously, for example, if we had three or more frames, we would have only three faults, one frame available, we would have a replacement with every reference, resulting in 11 faults. In general, we expect a curve such as that in Figure 3.21. As the number of frames increases, the number of page faults drops to some minimal level. Of course, adding physical memory increases the number of frames.

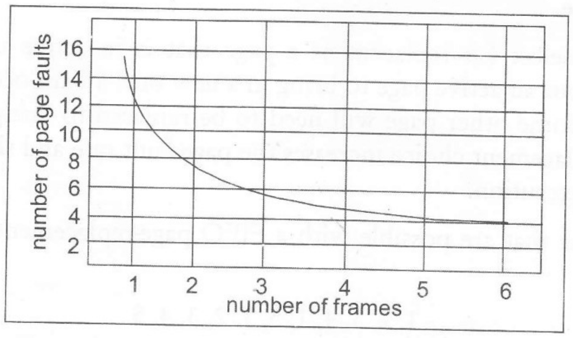

**Figure 3.21: Graph of Page Faults versus the Number of Frames**

To illustrate the page-replacement algorithms, we shall use the reference string

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

for a memory with three frames.

*FIFO Algorithm*

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is selected. We can create a FIFO queue to record the time when a page is brought into the memory. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults, and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in page 0 being replaced, since it was the oldest of the three pages in memory (0, 1, and 2) to be brought in. This replacement means that the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 3.22. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether.

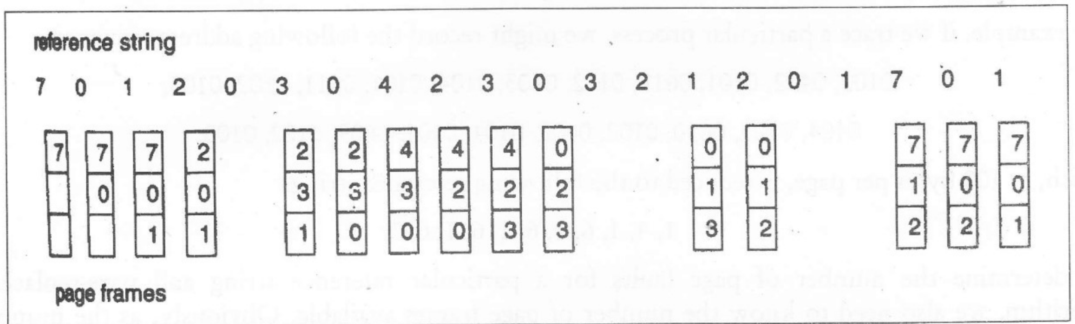

Figure 3.22: FIFO Page-replacement Algorithm

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. The page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.

Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we page out an active page to bring in a new one, a fault occurs almost immediately to retrieve the active page. Some other page will need to be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution, but does not cause incorrect execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, let us consider the reference string

1, 2, 3, 4, 1, 5, 1, 2, 3, 4, 5

The figure shows the curve of page faults versus the number of available frames. We notice that the number of faults for four frames (=10) is greater than the number of faults for three frames (=9). This result is most unexpected and is known as Belady's anomaly. Belady's anomaly reflects the fact that, for some page-replacement algorithms, the page fault rate may increase as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

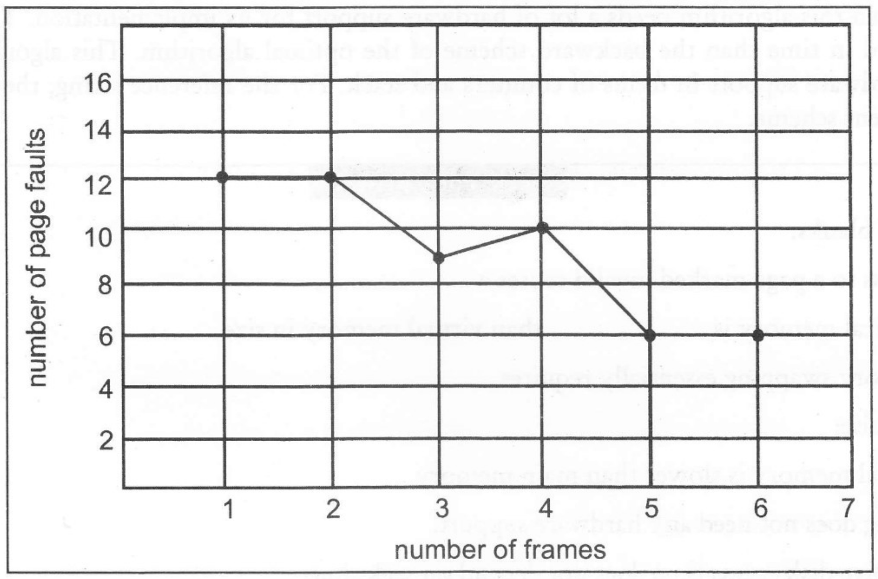

**Figure 3.23: Page-fault Curve for FIFO Replacement on a Reference String**

*Optimal Algorithm*

One result of the discovery of Belady's anomaly was the search for an optimal page-replacement algorithm. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal algorithm will never suffer from Belady's anomaly. An optimal page-replacement algorithm exists, and has been called OPT or MIN. Stated simply, it is:

*Replace the page that will not be used for the longest period of time.*

Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults.

Unfortunately, the optimal page-replacement algorithm is extremely difficult to implement because it requires prior knowledge of the reference string and therefore, is often used for comparison purpose alone.

*LRU Algorithm*

As stated earlier, it is very difficult to implement optimal algorithm. Yet an approximation to this algorithm may be employed, and that is LRU. The algorithm associates to each page brought into the memory, the time at which the page was last used. When a page is needed to be replaced, the page with the longest period of time when it was not used is selected for replacement.

*Replace the page that has not been used for the longest period of time.*

However, even this algorithm needs a lot of hardware support for its implementation. This algorithm looks forward in time than the backward scheme of the optimal algorithm. This algorithm requires extensive hardware support in terms of counters and stack. For the reference string, the figure shows the replacement scheme.

**Check Your Progress**

Fill in the blanks:

1. Access to a page marked invalid causes a ..................... .
2. Physical memory is ..................... than virtual memory in size.
3. Memory swapping essentially requires ..................... .

True or False:

1. Virtual memory is slower than main memory.
2. Paging does not need any hardware support.
3. Effective disk access time does not depend on seek-time.

## 3.6 LET US SUM UP

Memory is the electronic holding place for instructions and data that the computer's microprocessor can reach quickly. When the computer is in normal operation, its memory usually contains the main parts of the operating system and some or all of the application programs and related data that are being used. Memory is central to the operation of a modern computer system. Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses. While various different strategies are used to allocate space to processes competing for memory, three of the most popular are Best fit, Worst fit, and First fit. The memory is divided into two partitions, one for the resident operating system, and one for the user process. Memory is divided into a no. of fixed sized partitions.

Initially all memory is available for user processes, and is considered as large block of available memory. A possible solution to the external fragmentation problem is to permit the logical address space of a process to be non-contiguous, thus allowing a process to be allocated physical memory wherever the latter is available. An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory and the actual physical memory. Virtual memory is a technique that permits the execution of processes that may not be completely resident in the memory. The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified.

## 3.7 KEYWORDS

***Virtual memory*** is a technique that permits the execution of processes that may not be completely resident in the memory. The main advantage of this scheme is that programs can be larger than physical memory.

***Physical memory*** is broken into fixed-sized blocks called frames. Logical memory is also broken into blocks of the same size called pages.

*LRU* (Least recently used) states replace the page that has not been used for the longest period of time.

***Optimal algorithm*** states replace the page that will not be used for the longest period of time.

*FIFO* (First In First Out) replacement algorithm associates with each page the time when that page was brought into memory.

**Check Your Progress: Model Answers**

Fill in the blanks:

1. Page fault trap
2. Less
3. Secondary storage

True or False:

1. True
2. False
3. False

## 3.8 QUESTIONS FOR DISCUSSION

1. Suppose the page size in a computing environment is 1KB. What are the page number and the offset for the following:

   899 (decimal)

   17C (Hexadecimal)

2. Compare the advantages and disadvantages of segmentation and paging.

## 3.9 SUGGESTED READINGS

Andrew S. Tanenbaum, *Modern Operating System,* Published By Prentice Hall

Silberschatz Galvin, *Operating System Concepts,* Published By Addison Wesley

Andrew M. Lister, *Fundamentals of Operating Systems,* Published By Wiley

Colin Ritchie, *Operating Systems,* Published By BPB Publications.

# UNIT III

# LESSON 4

# CONCURRENCY

## CONTENTS

4.0 Aims and Objectives

4.1 Introduction

4.2 Principles of Concurrency

4.2.1 Bernstein's Conditions

4.2.2 Two-Process Solutions

4.2.3 Classical Process Co-ordination Problems

4.2.4 Cigarette Smoker's Problem

4.3 Mutual Exclusion

4.3.1 Mutual Exclusion Conditions

4.3.2 Proposals for Achieving Mutual Exclusion

4.4 Software Support for Mutual Exclusion

4.4.1 Peterson's Algorithm

4.4.2 Dekker's Algorithm

4.5 Hardware Support for Mutual Exclusion

4.5.1 Interrupt Disabling

4.5.2 Special Instructions: TestAndSet

4.5.3 Operating System Support

4.6 Semaphores Implementation

4.7 Let us Sum up

4.8 Keywords

4.9 Questions for Discussion

4.10 Suggested Readings

## 4.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain principles of concurrency
- Concept of mutual exclusion problem
- Software, hardware and operating system support for mutual exclusion
- Understand Semaphores

## 4.1 INTRODUCTION

Modern operating systems, such as Unix, execute processes concurrently: Although there is a single Central Processor (CPU), which execute the instructions of only one program at a time, the operating system rapidly switches the processor between different processes (usually allowing a single process a few hundred microseconds of CPU time before replacing it with another process.) Some of these resources (such as memory) are simultaneously shared by all processes. Such resources are being used in parallel between all running processes on the system. Other resources must be used by one process at a time, so must be carefully managed so that all processes get access to the resource. Such resources are being used in concurrently between all running processes on the system. The most important example of a shared resource is the CPU, although most of the I/O devices are also shared. For many of these shared resources the operating system distributes the time a process requires of the resource to ensure reasonable access for all processes. Consider the CPU: the operating system has a clock which sets an alarm every few hundred microseconds. At this time the operating system stops the CPU, saves all the relevant information that is needed to re-start the CPU exactly where it last left off (this will include saving the current instruction being executed, the state of the memory in the CPUs registers, and other data), and removes the process from the use of the CPU. The operating system then selects another process to run, returns the state of the CPU to what it was when it last ran this new process, and starts the CPU again. Let's take a moment to see how the operating system manages this. In a multi-programming environment, more than one process exist in the system competing for the resources. Based on criteria and algorithms employed by the system, one of them is selected at a time, is granted requisite resources and is executed while other candidate processes wait for their turn.

## 4.2 PRINCIPLES OF CONCURRENCY

In a multi-programming environment, more than one process exist in the system competing for the resources. Based on criteria and algorithms employed by the system, one of them is selected at a time, is granted requisite resources and is executed while other candidate processes wait for their turn.

The processes can execute in two different modes - sequential and concurrent. In sequential mode of execution, processes do not interfere with each other. A process, in this mode is executed only when the currently executing process has finished its execution. While a process is executing, it is allocated all the available resources and therefore there is no resource contention between the processes.

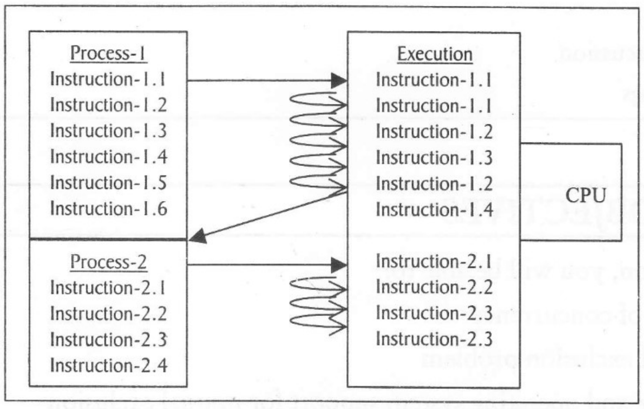

**Figure 4.1: Sequential Mode of Execution**

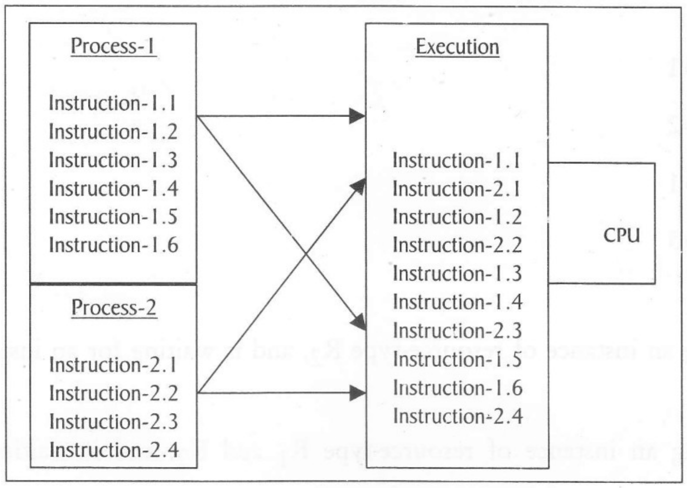

**Figure 4.2: Concurrent More of Execution**

If concurrency is not controlled adequately, it may turn into a condition in which no process makes any headway. Such a condition is called deadlock.

The situation is very different in case of concurrent mode of execution. In this case, each of the processes, try to acquire required resources and therefore, a lot of control is required on the part of the operating system to ensure a smooth operation.

***Precedence Graph or Resource-allocation Graph***

Deadlocks can be expressed more clearly using a directed graph, called *a system resource-allocation graph or precedence graph*. This graph has vertices so arranged that depicts the before-after relationship between the processes.

The graph has set of vertices V, divided into two sets of nodes. A set P= $\{P_1, P_2, P_3....P_n\}$ of all active processes in the system and set R = $\{R_1, R_2, R_3,......R_m\}$ of all the resource-types in the system. Each resource-type may have more than one instances of that type of resources. For example, a printer-type resource may have three instances, meaning there are three printers available in the system. $n(R_j)$ represents the number of instances of resource-type $R_i$.

The edge set of this graph has two types of edges – request edge and assignment edge. $P_i$ ® $R_j$ is a request edge, meaning that ith process has requested jth type of resource and is currently waiting for it. Similarly, $R_j$ ® $P_i$ is assignment edge, meaning that an instance of the jth resource type has been allocated to the ith process.

The resource-allocation graph shown in the figure describes the following resource-allocation situation:

The sets P, R and E:

$$P = \{P_1, P_2, P_3, P_4\}$$

$$R = \{R_1, R_2, R_3, R_4\}$$

$$E = \{P_1 ® R_1, P_2 ® R_3, R_1 ® P_2, R_2 ® P_2, R_2 ® P_1, R_3 ® P_3, P_4 ® R_4\}$$

*Resource Instances*

$n(R_1) = 1$

$n(R_2) = 2$

$n(R_3) = 1$

$n(R_4) = 3$

*Process States*

Process $P_1$ is holding an instance of resource-type $R_2$, and is waiting for an instance of resource-type $R_1$.

Process $P_2$ is holding an instance of resource-type $R_1$ and $R_2$, and is waiting for an instance of resource-type $R_3$.

Process $P_3$ is holding an instance of resource-type $R_3$.

Process $P_4$ is holding an instance of resource-type $R_4$, and is waiting for another instance of resource-type $R_4$.

If there is no cycle in a resource-allocation graph, it implies that there is no deadlock in the system. However, a cycle indicates, there may be a deadlock. If multiple instances exist for each resource, then deadlock does not occur even if there is a cycle as illustrated by the Figure 4.3.

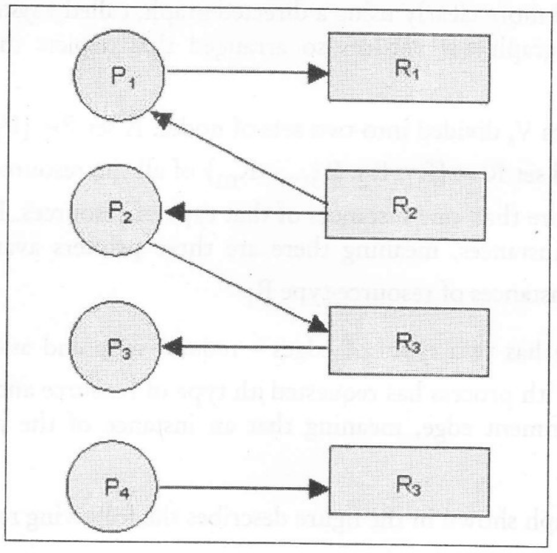

**Figure 4.3: Dependency Graph**

### 4.2.1 Bernstein's Conditions

*Necessary Condition for a Deadlock to Occur*

A deadlock situation can arise if four conditions hold simultaneously in a system:

1. ***Mutual exclusion:*** At least one resource must be held exclusively by one process, i.e. the held resource cannot be shared with any other process. If a process requests that resource, it must wait till the process possessing the resource releases the resource.
2. ***Hold and wait:*** There must be a process in the system holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.
3. ***No pre-emption:*** A resource can be released only voluntarily by the process holding it.
4. ***Circular wait:*** There must exist a set $\{P_0, P_1, P_2, .....P_n\}$ of waiting processes such that $P_0$ is waiting for a resource that is held by $P_1$, $P_1$ is waiting for a resource that is held by $P_2$, ....and $P_n$ is waiting for $P_0$.

***Time-dependency, Mutual Exclusion Problem and Critical Code-section***

Consider a system consisting of n processes $\{P_0, P_1, ..., P_n - 1\}$ Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on.

The important feature of the system is that, when one process is executing its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section.

The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.

A solution to the critical-section problem must satisfy the following three requirements:

- ***Mutual Exclusion:*** If process P, is executing in its critical section, then no other processes can be executing in their critical sections.
- ***Progress:*** If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision of which will enter its critical section next, and this selection cannot be postponed indefinitely.
- ***Bounded Waiting:*** There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

We assume that each process is executing at a non-zero speed. However, we can make no assumption concerning the relative speed of the n processes.

The solutions to this problem do not rely on any assumptions concerning the hardware instructions or the number of processors that the hardware supports. We do, however, assume that the basic machine language instructions (the primitive instructions such as load, store; these are executed atomically).

That is, if two such instructions are executed concurrently, the result is equivalent to their sequential execution in some unknown order. Thus, if a load and a store are executed concurrently, the load will get either the old value or the new value, but not some combination of the two.

When presenting an algorithm, we define only the variables used for synchronization purposes, and describe only a typical process P, whose general structure is. The entry section and exit section are enclosed boxes to highlight these important segments of code.

### 4.2.2 Two-Process Solutions

In this section, we restrict our attention to algorithms that are applicable to only two processes at a time. The processes are numbered Po and Pi. For convenience, when presenting P we use P, to denote the other process; that is j=l-l.

*Algorithm 1*

Our first approach is to let the processes share a common integer variable turn initialized to 0 (or 1). If turn = i, then process P is allowed to execute in its critical section. The structure of process P, is This solution ensures that only one process at a time can be in its critical section. However, it does not satisfy the progress requirement, since it requires strict alternation of processes in the execution of the critical section.

For example, if turn = 0 and Pi is ready to enter its critical section. $P_1$ cannot do so, even though Po may be in its remainder section.

*Algorithm 2*

The problem with algorithm I is that it does not retain sufficient information about the state of each process; it remembers only which process is allowed to enter that process's critical section. To remedy this problem, we can replace the variable turn with the following array:

Var flag: array [0..1] of Boolean;

The elements of the array are initialized to false. If flag[i] is true, this value indicates that P, is ready to enter the critical section.

In this algorithm, process Pi first sets flag [i.] to be true, signalling that it is ready to enter its critical section. Then, P, checks to verify that process P, not also ready to enter its critical section. If Pj were ready, then P; would w: until P, had indicated that it no longer needed to be in the critical section is, until flag[i] was false). At this point, P, would enter the critical section. Exiting the critical section, P; would set its flag to be false, allowing the other process (if it is waiting) to enter its critical section.

In this solution, the mutual-exclusion requirement is satisfied. Unfortunately, the progress requirement is not met. To illustrate this problem, we consider the following execution sequence:

To: Po sets flag[0] = true

TI: Pi sets flag{1] = true

Now PO and Pi are looping forever in their respective while statements.

This algorithm is crucially dependent on the exact timing of the two processes. The sequence could have been derived in an environment where there are several processors executing concurrently, or where an interrupt (such as timer interrupt) occurs immediately after step To is executed, and the CPU switched from one process to another.

Note that switching the order of the instructions for setting flag[j], will test the value of a flag[j], will not solve our problem. Rather, we will have a situation where it is possible for both processes to be in the critical section the same time violating the mutual-exclusion requirement.

*Combined Algorithm*

By combining the key ideas of algorithm 1 and algorithm 2, we obtain a correct solution to the critical-section problem, where all three requirements are met. The processes share two variables:

var flag: array [0. .1] of boolean,

turn: 0.. 1;

Initially flag{0] =flag{l] = false, and the value of turn is immaterial (but is either 0 or 1).

To enter the critical section, process P1; first flag {i] to be true, and then asserts that it is the other process' turn to enter if appropriate (turn =j). If both processes try to enter at the same time, turn will be set to both; and at roughly the same time. Only one of these assignments will last; the other will occur, I will be overwritten immediately. The eventual value of turn decides which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved,
2. The progress requirement is satisfied,
3. The bounded-waiting requirement is met.

To prove property I, we note that each P, enters its critical section only if either flag[j] = false or turn = i. Also note that, if both processes can be executing in their critical sections at the same time, then flag{0] =flag{l] = true. These two observations imply that $P_0$ and $P_1$ could not have executed successfully their while statements at about the same time, since the value of turn can be either 0 or I, but cannot be both. Hence, one of the processes — say Pj — must have executed successfully the while statement, whereas P; had to execute at least one additional statement ("turn = j"). However, since, at that time, flag{j] = true, and turn = i, and this condition will persist as long as Pj is in its critical section, the result follows: Mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process P, can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] = true and turn = j', this loop is the only one. If P, is not ready to enter the critical section, then flag{j] = false, and P, can enter its critical section. If Pj has set flag{j] = true and is also executing in its while statement, then either turn = i or turn = j. If turn = i, then P, will enter the critical section. If turn = j, then Pj will enter the critical section. However, once P, exits its critical section, it will reset flagij] to false, allowing P, to enter its critical section. If P, resets flag[j] to true, it must also set turn = i. Thus, since P, does not change the value of the variable turn while executing the while statement, P, will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

### 4.2.3 Classical Process Co-ordination Problems

There are a number of different process co-ordination problems arising in practical situations that exemplify important associated issues. These problems also provide a base for solution testing for process co-ordination problems. In this section, we will see some of such classical process co-ordination problems.

*The reader and writer's problem*

Courtois, Heymans and Purnas posed an interesting synchronization problem called the readers-writers problem. Suppose a resource is to be shared among a community of processes of two distinct types: readers and writers. A reader process can share the resource with any other reader process but not with any writer process. A writer process requires exclusive access to the resource whenever acquires any access to the resource.

This scenario is similar to one in which a file is to be shared among a set of processes. If a process wants only to read the file, then it may share the file with any other process that also only wants to read the file. If a writer wants to modify the file, then no other process should have access to the file writer has access to it.

***Solution:***

```
Monitor for Readers_writers problem
monitor reader_writer()
{
int numberofReaders = 0;
boolean busy = FALSE;
condition oktoWrite ;
public :
startRead
if  ( busy  II 9oktoWrite - queue) ) oktoRead . wait;
numberofReaders = numberofReaders  - 1;
oktoRead . signal
) ;
finished (
if ( ( numberofReaders 1 = 0) II busy )
oktoWrite . wait;
busy = TRUE
};
finish Write (
busy = FALSEif ( oktoWrite . queue)
oktoWrite.signal
else
oktoRead . signal
);
);
```

***The Dining-philosopher's Problem***

Five philosophers are seated around a table on which are placed five plates of pasta and five forks. While the philosophers think, they ignore the pasta and do not require a fork.

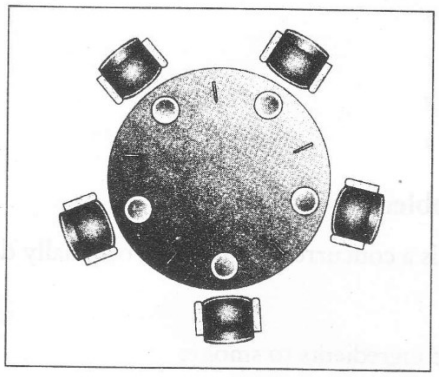

**Figure 4.4: Dining Philosopher's Problem**

When a philosopher decides to eat, he or she must obtain two forks, one from the left of the plate and one from the right of plate. To make the analogy fit the behaviour of sequential processes, assume a philosopher can pick up only a single fork at one time. After consuming food, the philosopher replaces the forks and resumes thinking. A philosopher cannot eat while the dining philosopher is eating, since forks are a shared resource.

***Solution:***

```
philosopher (int I [
while (TRUE) {
..../* Thinking */
P (fork [j):  /*Pick up left fork */
P (fork [I+1) mod 5]; /Pick up right fork */
eat ();
V (fork [I+1) mod 5]);
V (fork [i];
}
}
philosopher 4() {
while (TRUE {
..../* Thinking */
P (fork [0]);  /* Pick up right fork */
P (fork [4];  /* Pick up left fork */
eat ();
V (fork[4]
V (fork{0{();
}
}
semaphore fork [5] = {1, 1, 1, 1, 1};
fork (philosopher), 1, 0);
fork (philosopher), 1, 1);
fork (philosopher), 1, 2);
fork (philosopher), 1, 3);
fork (philosopher), 4, 0);
```

### 4.2.4 Cigarette Smoker's Problem

The cigarette smoker's problem is a concurrency problem, originally described in 1971 by S. S. Patil.

*Problem description*

Assume a cigarette requires three ingredients to smoke:

1. Tobacco
2. Paper
3. A match

Assume there are also three chain smokers around a table, each of whom has an infinite supply of one of the three ingredients — one smoker has an infinite supply of tobacco, another has an infinite supply of paper, and the third has an infinite supply of matches.

Assume there is also a non-smoking arbiter. The arbiter enables the smokers to make their cigarettes by selecting two of the smokers at random (non-deterministically), taking one item out of each of their supplies, and placing the items on the table. He then notifies the third smoker that he has done this. The third smoker removes the two items from the table and uses them (along with his own supply) to make a cigarette, which he smokes for a while. Meanwhile, the arbiter, seeing the table empty, again chooses two smokers at random and places their items on the table. This process continues forever.

The smokers do not hoard items from the table; a smoker only begins to roll a new cigarette once he is finished smoking the last one. If the arbiter places tobacco and paper on the table while the match man is smoking, the tobacco and paper will remain untouched on the table until the match man is finished with his cigarette and collects them.

The problem is to simulate all four roles as software programs, using only a certain set of synchronization primitives. In Patil's original formulation, the only synchronization primitive allowed was the semaphore, and none of the four programs were allowed to contain conditional jumps — only the conditional waits implied by the semaphores' wait operations.

*Argument*

Patil's argument was that Edsger Dijkstra's semaphore primitives were limited. He used the cigarette smoker's problem to illustrate this point by saying that it cannot be solved with semaphores. However, Patil placed heavy constraints on his argument.

1. The agent code is not modifiable.
2. The solution is not allowed to use conditional statements or an array of semaphores.

With these two constraints, a solution to the cigarette smokers problem is impossible.

The first restriction makes sense, as Downey says in The Little Book of Semaphores, because if the agent represents an operating system, it would be intractable to modify it every time a new application came along. However, as David Parnas points out, the second restriction makes almost any non-trivial problem impossible to solve.

It is important, however, that such an investigation [of Dijkstra primitives] not investigate the power of these primitives under artificial restrictions. By artificial we mean restrictions that cannot be justified by practical considerations. In this author's opinion, restrictions prohibiting either conditionals or semaphore arrays are artificial. On the other hand, prohibition of "busy waiting" is quite realistic.

***Solution***

If we remove the second of Patil's constraints, the cigarette smokers problem becomes solvable using binary semaphores or mutexes. Let us define an array of binary semaphores A, one for each smoker; and a binary semaphore for the table, T. Initialize the smokers' semaphores to zero and the table's semaphore to 1. Then the arbiter's code is:

```
while true
{
Wait (T);
Choose smokers i and j nondeterministically, making the third smoker k
Signal (A[k]);
}
```

and the code for smoker i is:

```
    while true
{
Wait (A[i]);
Make a cigarette;
Signal (T);
Smoke the cigarette;
}
```

## 4.3 MUTUAL EXCLUSION

A way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing.

Formally, while one process executes the shared variable, all other processes desiring to do so at the same time moment should be kept waiting; when that process has finished executing the shared variable, one of the processes waiting; while that process has finished executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this fashion, each process executing the shared data (variables) excludes all others from doing so simultaneously. This is called Mutual Exclusion.

Note that mutual exclusion needs to be enforced only when processes access shared modifiable data - when processes are performing operations that do not conflict with one another they should be allowed to proceed concurrently.

### 4.3.1 Mutual Exclusion Conditions

If we could arrange matters such that no two processes were ever in their critical sections simultaneously, we could avoid race conditions. We need four conditions to hold to have a good solution for the critical section problem (mutual exclusion).

- No two processes may at the same moment inside their critical sections.
- No assumptions are made about relative speeds of processes or number of CPUs.
- No process outside its critical section should block other processes.
- No process should wait arbitrary long to enter its critical section.

### 4.3.2 Proposals for Achieving Mutual Exclusion

The mutual exclusion problem is to devise a pre-protocol (or entry protocol) and a post-protocol (or exist protocol) to keep two or more threads from being in their critical sections at the same time.

*Problem:* When one process is updating shared modifiable data in its critical section, no other process should allowed to enter in its critical section.

*Proposal 1 - Disabling Interrupts (Hardware Solution)*

Each process disables all interrupts just after entering in its critical section and re-enable all interrupts just before leaving critical section. With interrupts turned off the CPU could not be switched to other process. Hence, no other process will enter its critical and mutual exclusion achieved.

*Conclusion:* Disabling interrupts is sometimes a useful interrupts is sometimes a useful technique within the kernel of an operating system, but it is not appropriate as a general mutual exclusion mechanism for users process. The reason is that it is unwise to give user process the power to turn off interrupts.

*Proposal 2 - Lock Variable (Software Solution)*

In this solution, we consider a single, shared, (lock) variable, initially 0. When a process wants to enter in its critical section, it first test the lock. If lock is 0, the process first sets it to 1 and then enters the critical section. If the lock is already 1, the process just waits until (lock) variable becomes 0. Thus, a 0 means that no process in its critical section, and 1 means hold your horses - some process is in its critical section.

*Conclusion:* The flaw in this proposal can be best explained by example. Suppose process A sees that the lock is 0. Before it can set the lock to 1 another process B is scheduled, runs, and sets the lock to 1. When the process A runs again, it will also set the lock to 1, and two processes will be in their critical section simultaneously.

*Proposal 3 - Strict Alteration*

In this proposed solution, the integer variable 'turn' keeps track of whose turn is to enter the critical section. Initially, process A inspect turn, finds it to be 0, and enters in its critical section. Process B also finds it to be 0 and sits in a loop continually testing 'turn' to see when it becomes 1.Continuously testing a variable waiting for some value to appear is called the Busy-Waiting.

*Conclusion:* Taking turns is not a good idea when one of the processes is much slower than the other. Suppose process 0 finishes its critical section quickly, so both processes are now in their noncritical section. This situation violates above mentioned condition 3.

*Using Systems calls 'sleep' and 'wakeup'*

Basically, what above mentioned solution do is this: when a processes wants to enter in its critical section , it checks to see if then entry is allowed. If it is not, the process goes into tight loop and waits (i.e., start busy waiting) until it is allowed to enter. This approach waste CPU-time.

Now look at some interprocess communication primitives is the pair of steep-wakeup.

*Sleep*

It is a system call that causes the caller to block, that is, be suspended until some other process wakes it up.

*Wakeup*

It is a system call that wakes up the process.

Both 'sleep' and 'wakeup' system calls have one parameter that represents a memory address used to match up 'sleeps' and 'wakeups'.

*The Bounded Buffer Producers and Consumers*

The bounded buffer producers and consumers assumes that there is a fixed buffer size i.e., a finite numbers of slots are available.

*Statement*

To suspend the producers when the buffer is full, to suspend the consumers when the buffer is empty, and to make sure that only one process at a time manipulates a buffer so there are no race conditions or lost updates.

As an example how sleep-wakeup system calls are used, consider the producer-consumer problem also known as bounded buffer problem.

Two processes share a common, fixed-size (bounded) buffer. The producer puts information into the buffer and the consumer takes information out. Trouble arises when:

1. The producer wants to put a new data in the buffer, but buffer is already full.

   **Solution:** Producer goes to sleep and to be awakened when the consumer has removed data.

2. The consumer wants to remove data the buffer but buffer is already empty.

   **Solution:** Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

*Conclusion:* This approach also leads to same race conditions we have seen in earlier approaches. Race condition can occur due to the fact that access to 'count' is unconstrained. The essence of the problem is that a wakeup call, sent to a process that is not sleeping, is lost.

## 4.4 SOFTWARE SUPPORT FOR MUTUAL EXCLUSION

What if the only hardware support is no simultaneous access to memory locations? Dekker's algorthm implements mutual exclusion for two processes in software without deadlock or starvation

"Software" mutual exclusion

- With Dekker's correct algorithm, two processes take turns insisting on right to enter critical section
- Peterson's algorithm is a simpler correct solution. Both involve some busy waiting

### 4.4.1 Peterson's Algorithm

```
boolean flag [2];
int turn;
void P0( ){
    while (true){
          flag [0] = true;
          turn = 1;
          while (flag [1] && turn == 1)
                < do nothing >;
          < critical section >;
          flag [0] = false;
          < remainder >;
    }
}
void main( ){
    flag [0] = false;
    flag [1] = false;
    parbegin (P0, P1);
}
void P1( ){
    while (true){
          flag [1] = true;
          turn = 0;
          while (flag [0] && turn == 0)
                < do nothing >;
          < critical section >;
          flag [1] = false;
          < remainder >;
    }
}
```

*Notes:*

- flag[i] announces Pi's desire to enter critical section
- *turn* indicates which process should insist on entering
- Mutual exclusion enforced by *flag* in while statement
- Deadlock/starvation prevented by *turn*
- Only one processes is blocked at a time

### 4.4.2 Dekker's Algorithm

Dekker's algorithm is the first published software-only, two-process mutual exclusion algotithm.

```
int favored_process;
int interested[2];
void enter_region(int process) {
    int other = 1-process;

    interested[process] = TRUE;
    while ( interested[other] ) {
        if ( favored_process == other ) {
            interested[process] = FALSE;
            while ( favored_process == other );
            interested[process] = TRUE;
        }
    }
}

void leave_region(int process) {
    int other = 1 - process;
    favored_process = other;
    interested[process] = FALSE;
}
```

Dekker adds the idea of a favored process and allows access to either process when the request is uncontested. When there is a conflict, one process is favored, and the priority reverses after successful execution of the critical section. It's worthwhile to trace through the code seeing why each test is in the code and why.

***Dekker's algorithm was simplified by Peterson:***

```
int turn;
int interested[2];
void enter_region(int process) {
```