

University of Technology
الجامعة التكنولوجية



Computer Science Department
قسم علوم الحاسوب

Operating System2
نظم التشغيل 2

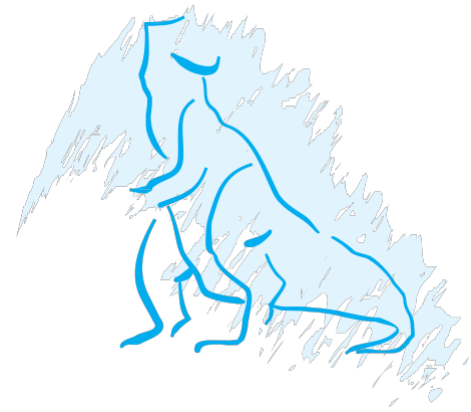
Lect. Rehab .F
ا.د رحاب فليح

Lect. Rana Mohammed
م.د رنا محمد



cs.uotechnology.edu.iq

Virtual Memory



1.1 Virtual Memory

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory.

Virtual memory involves the separation of logical memory as perceived by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure1). Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available

The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address – say, address 0 – and exists in contiguous memory,

physical memory may be organized in page frames and that the physical page frames assigned to a process may not be contiguous. It is up to the memory- management unit (MMU) to map logical pages to physical page frames in memory.

2. Demand Paging

With demand-paged virtual memory, pages are loaded only when they are demanded during program execution.

Pages that are never accessed are thus never loaded into physical memory A demand-paging system is similar to a paging.

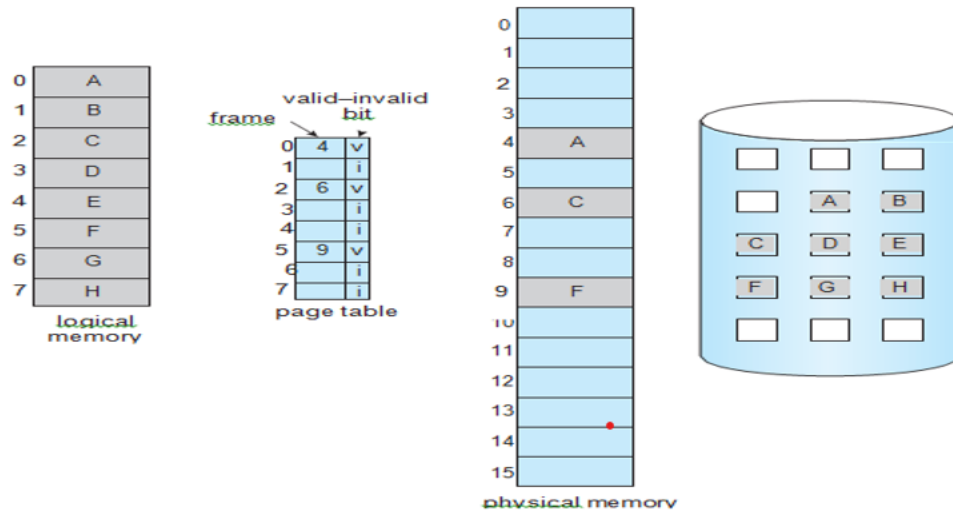


Figure 1 : page table When some pages are not in main Memory.

system with swapping (Figure2) where processes reside in secondary memory (usually a disk). When we want to execute a process, we swap it into memory.

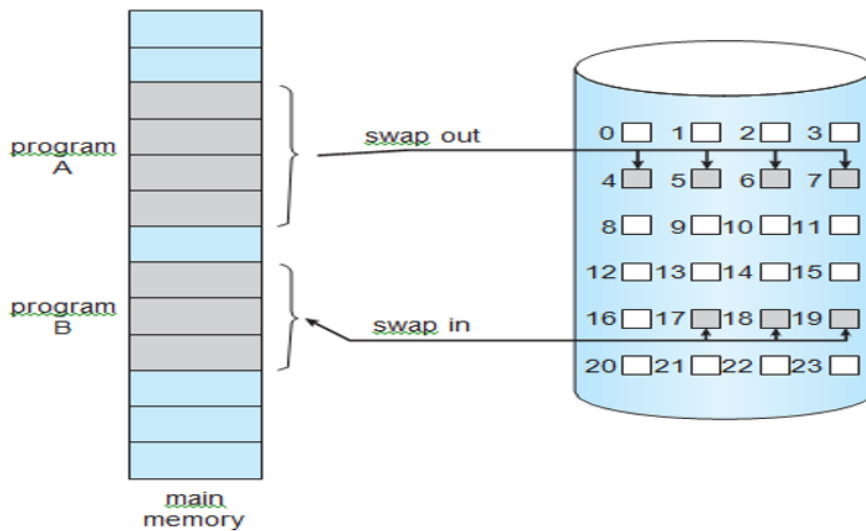


Figure 2: system with swapping

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a **page fault**. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory

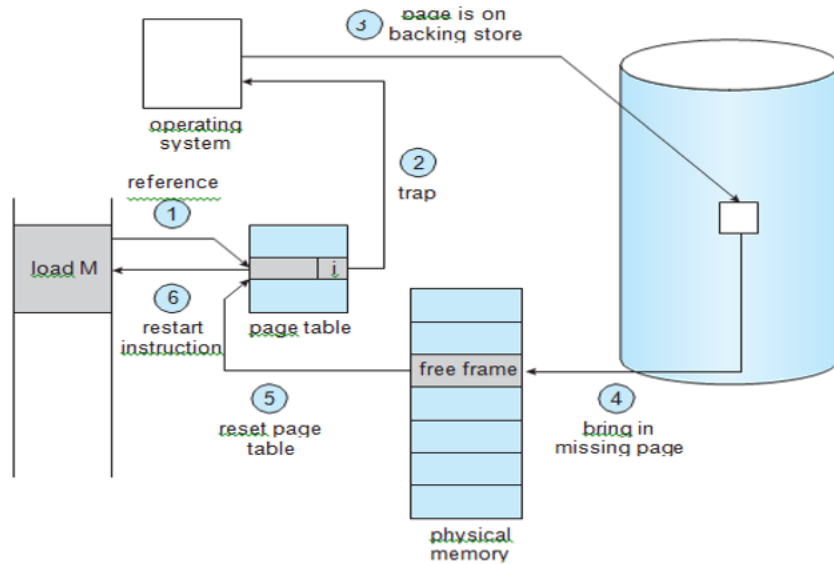


Figure 3: Steps in handling a page fault.

3. 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.

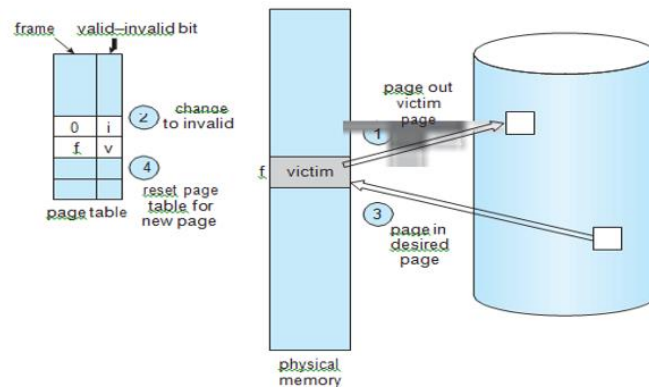


Figure 4: page Replacement.

There are many different page-replacement algorithms. Every operating system probably has its own 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 using a random-number generator, for example), or we can trace a given system and record 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 use two facts. 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, rather than the entire address. Second, if we have a reference to a page p , then any references to page p that *immediately* follow will never cause a page fault. Page p will be in memory after the first reference, so 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

At 100 bytes per page, this sequence is reduced to the following reference string

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

We next illustrate several page-replacement algorithms. In doing so, we 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

3.1 FIFO Page Replacement

The simplest page-replacement algorithm is a first-in, first-out (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 chosen. Notice that it is not strictly necessary to record the time when a page is brought in. 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.

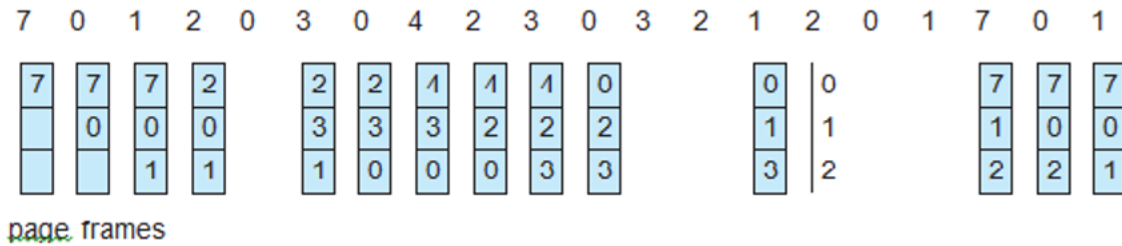


Figure 5: FIFO page Replacement Algorithm.

Every time a fault occurs; we show which pages are in our three frames. There are fifteen faults altogether to illustrate the problems that are possible with a FIFO page-replacement algorithm, consider the following reference string:

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

Notice that the number of faults for four frames (ten) is *greater* than the number of faults for three frames (nine). This most unexpected result is known as **Belady's anomaly**

3.2 Optimal Page Replacement

One result of the discovery of Belady's anomaly was the search for an **optimal page-replacement algorithm** — the algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN. It is simply this:

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.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. (We encountered a similar situation with the SJF CPU-scheduling algorithm. As a result, the optimal algorithm is used mainly for comparison studies.

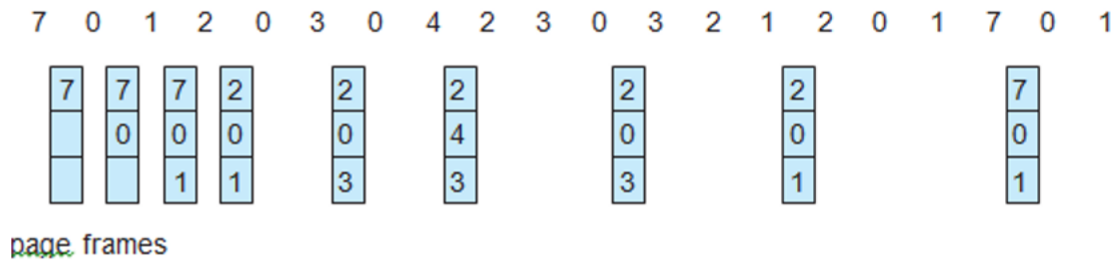


Figure 6: Optimal page Replacement Algorithm.

With only **nine-page** faults, optimal replacement is much better than a FIFO algorithm, which results in **fifteen** faults

3.3 LRU Page Replacement

If we use the recent past as an approximation of the near future, then we can replace the page that *has not been used* for the longest period of time. This approach is the **least recently used (LRU) algorithm**. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. The result of applying LRU replacement to our example reference string is shown in figure 7.

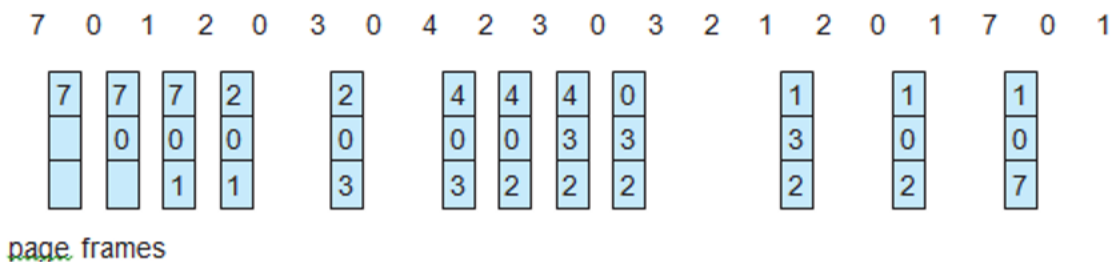
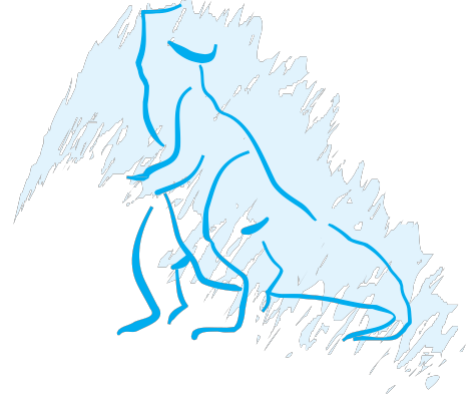


Figure 7: LRU page Replacement Algorithm.

Process Synchronization



2.1 Introduction

A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. The former case is achieved through the use of threads. Concurrent access to shared data may result in data inconsistency, however. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained. We've already seen that processes can execute concurrently or in parallel. introduced the role of process scheduling and described how the CPU scheduler switches rapidly between processes to provide concurrent execution. This means that one process may only partially complete execution before another process is scheduled. In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instruction so another process.

Let's consider an example of how this can happen, we developed a model of a system consisting of cooperating sequential processes or threads, all running asynchronously and possibly sharing data. We illustrated this model with the **producer – consumer problem**, which is representative of operating systems. Specifically. We described how a bounded buffer could be used to enable processes to share memory. We now return to our consideration of the bounded buffer. As we pointed out, our original solution allowed at $\text{BUFFER_SIZE} - 1$ items in the buffer at the same time. Suppose we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable counter, initialized to 0. counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer. The code for the **producer process can be modified as follows:**

```
while (true) {
    /* produce an item in next produced */
    while (counter == BUFFER_SIZE) ;
        /* do nothing */
    buffer[in] = next produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

The code for the **consumer process can be modified as follows:**

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in next consumed */  
}
```

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable `counter` is currently 5 and that the producer and consumer processes concurrently execute the statements “`counter++`” and “`counter--`”. Following the execution of these two statements, the value of the variable `counter` may be 4, 5, or 6! The only correct result, though, is `counter == 5`, which is generated correctly if the producer and consumer execute separately.

We can show that the value of `counter` may be incorrect as follows. Note that the statement “`counter++`” may be implemented in machine language (on a typical machine) as follows:

$$\begin{aligned} \text{register}_1 &= \text{counter} \\ \text{register}_1 &= \text{register}_1 + 1 \\ \text{counter} &= \text{register}_1 \end{aligned}$$

where register_1 is one of the local CPU registers. Similarly, the statement “`counter--`” is implemented as follows:

$$\begin{aligned} \text{register}_2 &= \text{counter} \\ \text{register}_2 &= \text{register}_2 - 1 \\ \text{counter} &= \text{register}_2 \end{aligned}$$

where again *register2* is one of the local CPU registers. Even though *register1* and *register2* may be the same physical register (an accumulator, say), remember that the contents of this register will be saved and restored by the interrupt handler.

The concurrent execution of “counter++” and “counter--” is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is the following:

T_0 :	<i>producer</i>	execute	$register_1 = \mathbf{counter}$	{ $register_1 = 5$ }
T_1 :	<i>producer</i>	execute	$register_1 = register_1 + 1$	{ $register_1 = 6$ }
T_2 :	<i>consumer</i>	execute	$register_2 = \mathbf{counter}$	{ $register_2 = 5$ }
T_3 :	<i>consumer</i>	execute	$register_2 = register_2 - 1$	{ $register_2 = 4$ }
T_4 :	<i>producer</i>	execute	$\mathbf{counter} = register_1$	{ $counter = 6$ }
T_5 :	<i>consumer</i>	execute	$\mathbf{counter} = register_2$	{ $counter = 4$ }

Notice that we have arrived at the incorrect state “**counter == 4**”, indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T_4 and T_5 , we would arrive at the incorrect state “**counter == 6**”.

We would arrive at this incorrect state because we allowed both processes to manipulate the variable *counter* concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable *counter*. To make such a guarantee, we require that the processes be synchronized in some way. Situations such as the one just described occur frequently in operating systems as different parts of the system manipulate resources. Furthermore, as we have emphasized in earlier chapters, the growing importance of multicore systems has brought an increased emphasis on developing multithreaded applications. In such applications, several threads — which are quite possibly sharing data — are running in parallel on different processing cores.

```

do {
    entry section
    critical section
    exit section
    remainder section
} while (true);

```

Figure 1 General structure of a typical process P_i .

we want any changes that result from such activities not to interfere with one another. Because of the importance of this issue, we devote a major portion of this chapter to **process synchronization** and **coordination** among cooperating processes.

2.2 The Critical-Section Problem

Consider system of n processes $\{p_0, p_1, \dots, p_{n-1}\}$ Each process has critical section segment of code Process may be changing common variables, updating table, writing file, etc. When one process in critical section, no other may be in its critical section.

Critical section problem is to design protocol to solve this each process must ask permission to enter critical section in **entry section**, may follow **critical section** with exit section, then **remainder section**.

General structure of process P_i

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

Figure 2 part of critical section.

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

- 1. Mutual exclusion.** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

- 2. Progress.** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

- 3. Bounded waiting.** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after

process has made a request to enter its critical section and before that request is granted.

Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **non-preemptive kernels**. **A preemptive kernel allows a process to be preempted while it is running in kernel mode. A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.** Obviously, a non-preemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time. We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors.

Why, then, would anyone favor a preemptive kernel over a non-preemptive one?

A preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes. (Of course, this risk can also be minimized by designing kernel code that does not behave in this way.) Furthermore, a preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel. Later in this chapter, we explore how various operating systems manage preemption within the kernel.

2.2.1 Peterson's Solution

Next, we illustrate a classic software-based solution to the critical-section problem known as **Peterson's solution**. Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson's solution will work correctly on such architectures. However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of **mutual exclusion, progress, and bounded waiting**.

```
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    critical section  
    flag[i] = false;  
    remainder section  
} while (true);
```

Figure 3 The structure of process P_i in Peterson's solution.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P_0 and P_1 . For convenience, when presenting P_i , we use P_j to denote the other process; that is, j equals $1 - i$.

Peterson's solution requires the two processes to share two data items:

```
int turn;  
boolean flag[2];
```

The variable `turn` indicates whose turn it is to enter its critical section. That is, if `turn == i`, then process P_i is allowed to execute in its critical section. The `flag` array is used to indicate if a process is ready to enter its critical section. For example, if `flag[i]` is true, this value indicates that P_i is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in Figure 3.

To enter the critical section, process P_i first sets `flag[i]` to be true and then sets `turn` to the value `j`, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, `turn` will be set to both `i` and `j` at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of `turn` determines 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.(mutex lock)**
- 2. The progress requirement is satisfied.**
- 3. The bounded-waiting requirement is met.**

To prove property 1, we note that each P_i 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[1] == true`. These two observations imply that P_0 and P_1 could not have successfully executed their while statements at about the same time, since the say, P_j — must have successfully executed the while statement, whereas P_i had to execute at least one additional statement (“`turn == j`”). However, at that time,

$\text{flag}[j] == \text{true}$ and $\text{turn} == \mathbf{j}$, and this condition will persist as long as P_j is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition $\text{flag}[j] == \text{true}$ and $\text{turn} == \mathbf{j}$; this loop is the only one possible. If P_j is not ready to enter the critical section, then $\text{flag}[j] == \text{false}$, and P_i can enter its critical section. If P_j has set $\text{flag}[j]$ to true and is also executing in its while statement, then either $\text{turn} == \mathbf{i}$ or $\text{turn} == \mathbf{j}$. If $\text{turn} == \mathbf{i}$, then P_i will enter the critical section. If $\text{turn} == \mathbf{j}$, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset $\text{flag}[j]$ to false, allowing P_i to enter its critical section. If P_j resets $\text{flag}[j]$ to true, it must also set turn to \mathbf{i} .

Thus, since P_i does not change the value of the variable turn while executing the while statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

2.2.2 Synchronization Hardware

We have just described one software-based solution to the critical-section problem. However, as mentioned, software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically** — that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Like `test and set ()` and `compare and swap()` instructions. as an exercise.

The main disadvantage of the implementation given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously

2.2.3 Semaphores

Mutex locks, as we mentioned earlier, are generally considered the simplest of synchronization tools. In this section, we examine a more robust tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities.

A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: `wait()` and `signal()`. The `wait ()` operation was originally termed P (from the Dutch *proberen*, “to test”); `signal ()` was originally called V (from *verhogen*, “to increment”). The definition of `wait ()` is as follows:

```
wait(S)
{
while (S <= 0)
; // busy wait
S--;
}
```

The definition of `signal ()` is as follows:

```
signal(S) {
S++;
}
```

All modifications to the integer value of the semaphore in the `wait ()` and `signal()` operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of `wait(S)`, the testing of the integer value of S ($S \leq 0$), as well as its possible modification ($S--$), must be executed without interruption. We shall see how these operations can be implemented in Section 5.6.2. First, let’s see how semaphores can be used.

2.2.3.1 Semaphore Usage

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a `wait()` operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a `signal()` operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

3. Classic Problems of Synchronization

In this section, we present a number of synchronization problems as examples of a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we use semaphores for synchronization, since that is the traditional way to present such solutions. However, actual implementations of these solutions could use mutex locks in place of binary semaphores.

Classical problems used to test newly-proposed synchronization schemes

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

3.1 Bounded-Buffer Problem

The bounded-buffer problem was introduced in Section 5.1; it is commonly used to illustrate the power of synchronization primitives. Here, we present a general structure of this scheme without committing ourselves to any particular implementation. We provide a related programming project in the exercises at the end of the chapter. In our problem, the **producer** and **consumer** processes share the following data structures:

```
int
n;
semaphore mutex =
1; semaphore empty
= n; semaphore
full = 0
```

We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and **full** semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n ; the semaphore **full** is initialized to the value 0.

The code for the producer process is shown in Figure 5.9, and the code for the consumer process is shown in Figure 2.4. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

```
do {
    wait(full);
    wait(mutex);
    . . .
    /* remove an item from buffer to next consumed */
    . . .
    signal(mutex);
    signal(empty);
    . . .
    /* consume the item in next consumed */
} while (true);
```

Figure 2.4 The structure of the consumer process.

3.2 The Readers–Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as *readers* and to the latter as *writers*. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the **readers–writers’ problem**. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers–writers problem has several variations, all involving priorities. The simplest one, referred to as the *first* readers–writers’ problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The *second* readers–writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in **starvation**. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. Next, we present a solution to the first readers–writers’ problem. See the bibliographical notes at the end of the chapter for references describing starvation-free solutions to the second readers–writers’ problem.

In the solution to the first readers–writers’ problem, the reader processes

share the following data structures:

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

The semaphores `mutex` and `rw_mutex` are initialized to 1; `read_count` is initialized to 0. The semaphore `rw_mutex` is common to both reader and writer

```
do {
    wait(rw_mutex);
    /* writing is performed */
    signal(rw_mutex);
} while (true);
```

Figure 2.5 The structure of a writer process.

processes. The `mutex` semaphore is used to ensure mutual exclusion when the variable `read_count` is updated. The `read_count` variable keeps track of how many processes are currently reading the object. The semaphore `rw_mutex` functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections. The code for a reader process is shown in Figure 2.6. Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on `rw_mutex`, and $n - 1$ readers are queued on `mutex`. Also observe that, when a writer executes `signal(rw_mutex)`, we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers-writers' problem and its solutions have been generalized to provide **reader-writer** locks on some systems. Acquiring a reader-writer lock requires specifying the mode of the lock: either *read* or *write* access. When a process wishes only to read shared data; it requests the reader-writer lock in read mode. A process wishing to modify the shared data must request the

lock in write mode. Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

Reader–writer locks are most useful in the following situations:

```
do {  
    wait(mutex);  
    read count++;  
    if (read-count == 1)  
        wait(rw-mutex);  
    signal(mutex);  
    . . .  
    /* reading is performed */  
    . . .  
    wait(mutex);  
    read-count--;  
    if (read-count == 0)  
        signal(rw-mutex);  
    signal(mutex);  
} while (true);
```

Figure 2.6 The structure of a reader process.

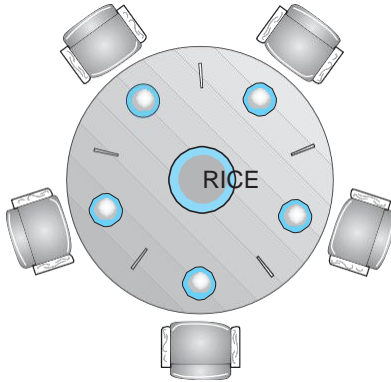


Figure 5.7 The situation of the dining philosophers.

- In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
- In applications that have more readers than writers. This is because reader – writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader – writer lock.

3.3 The Dining-Philosophers Problem

Consider **five philosophers** who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 2.7). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

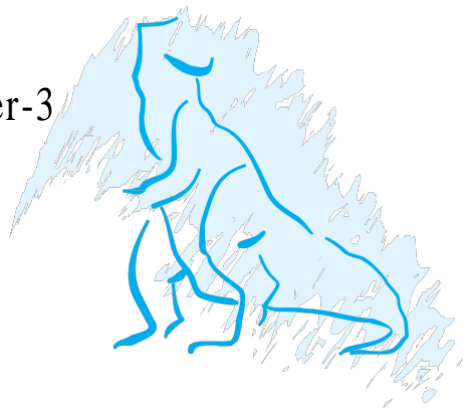
The **dining-philosophers problem** is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a `wait()` operation on that semaphore. She releases her chopsticks by executing the `signal()` operation on the appropriate semaphores. Thus, the shared data are

```
        semaphore chopstick [5];
do { wait (chopstick[i]);
wait(chopstick[(i+1) % 5]);
        . . .
        /* eat for a while */
        . . . signal(chopstick[i]);
signal(chopstick[(i+1) % 5]);
        . . .
        /* think for a while */
        . . .
    } while (true);
```

Figure 2.8 The structure of philosopher *i*.

where all the elements of `chopstick` are initialized to 1. The structure of philosopher *i* is shown in Figure 2.8.

Deadlocks



3. Introduction

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

3.1 System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources may be partitioned into several types (or classes), each consisting of some number of identical instances. CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances. If a process requests an instance of a resource type, the allocation of any instance of the type should satisfy the request. If it does not, then the instances are not identical, and the resource type classes have not been defined properly.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two.

Under the normal mode of operation, a process may utilize a resource in

only the following sequence:

- 1. Request. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.**
- 2. Use. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).**
- 3. Release. The process releases the resource.**

events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, semaphores, mutex locks, and files). However, other types of events may result in deadlocks. To illustrate a deadlocked state, consider a system with three CD RW drives. Suppose each of three processes holds one of these CD RW drives. If each process now requests another drive, the three processes will be in a deadlocked state. Each is waiting for the event “CD RW is released,” which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.

Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process P_i is holding the DVD and process P_j is holding the printer. If P_i requests the printer and P_j requests the DVD drive, a deadlock occurs. Developers of multithreaded applications must remain aware of the possibility of deadlocks. The locking tools presented in Chapter 5 are designed to avoid race conditions. However, in using these tools, developers must pay careful attention to how locks are acquired and released. Otherwise.

3.2 Deadlock Characterization

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. Before we discuss the various methods for dealing with the deadlock problem, we look more closely at features that characterize deadlocks.

DEADLOCK WITH MUTEX LOCKS

Let's see how deadlock can occur in a multithreaded Pthread program using mutex locks. The `pthread_mutex_init()` function initializes an unlocked mutex. Mutex locks are acquired and released using `pthread_mutex_lock()` and `pthread_mutex_unlock()`, respectively. If a thread attempts to acquire a locked mutex, the call to `pthread_mutex_lock()` blocks the thread until the owner of the mutex lock invokes `pthread_mutex_unlock()`.

Two mutex locks are created in the following code example:

```
/* Create and initialize the mutex locks */  
pthread_mutex_t first_mutex;  
pthread_mutex_t second_mutex;  
  
pthread_mutex_init(&first_mutex, NULL);  
pthread_mutex_init (&second_mutex, NULL);
```

Next, two threads — thread one and thread two — are created, and both these threads have access to both mutex locks.

3.3 Necessary Conditions

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

1. Mutual exclusion: at least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and wait: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. No preemption: Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular wait. A set $\{P_0, P_1, \dots, P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1 , P_1 is waiting for a resource held by P_2 , ..., P_{n-1} is waiting for a resource held by P_n , and P_n is waiting for a resource held by P_0 . We emphasize that all four conditions must hold for a deadlock to occur.

3.4 Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. This graph consists of a set of vertices V and a set of edges E . The set of vertices V is partitioned into two different types of nodes: $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the active processes in the system, and $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system. A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$; it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource. A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i . A directed edge $P_i \rightarrow R_j$ is called a request edge; a directed edge $R_j \rightarrow P_i$ is called an assignment edge. Pictorially, we represent each process P_i as a circle and each resource type R_j as a rectangle. Since resource type R_j may have more than one instance, we

represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle R_j , whereas an assignment edge must also designate one of the dots in the rectangle. When process P_i requests an instance of resource type R_j , a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted. The resource-allocation graph shown in Figure 3.1 depicts the following situation.

The sets P , R , and E :

- $P = \{P_1, P_2, P_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- Resource instances:
 - One instance of resource type R_1
 - Two instances of resource type R_2
 - One instance of resource type R_3
 - Three instances of resource type R_4
- 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 R_1 and an instance of R_2 and is waiting for an instance of R_3 .
 - Process P_3 is holding an instance of R_3 .

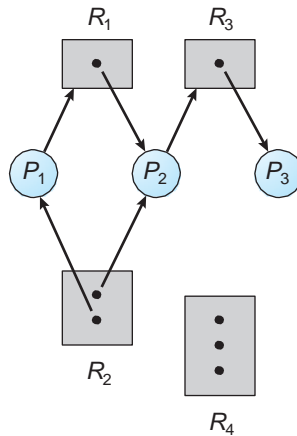


Figure 3.1 Resource-allocation graph.

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist. If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock. If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock. To illustrate this concept, we return to the resource-allocation graph depicted in Figure 3.1. Suppose that process P₃ requests an instance of resource type R₂.

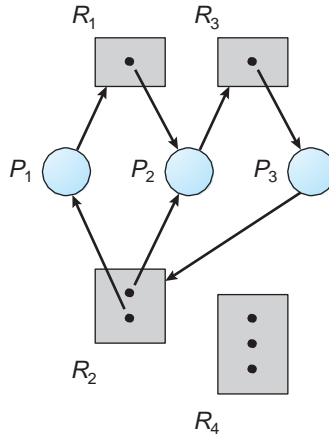


Figure 3.2 Resource-allocation graph with a deadlock.

type R_2 . Since no resource instance is currently available, we add a request edge $P_3 \rightarrow R_2$ to the graph (Figure 3.2). At this point, two minimal cycles exist in the system:

$$\begin{aligned}
 P_1 &\rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1 \\
 P_2 &\rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2
 \end{aligned}$$

Processes P_1 , P_2 , and P_3 are deadlocked. Process P_2 is waiting for the resource R_3 , which is held by process P_3 . Process P_3 is waiting for either process P_1 or process P_2 to release resource R_2 . In addition, process P_1 is waiting for process P_2 to release resource R_1 .

Now consider the resource-allocation graph in Figure 3.3. In this example, we also have a cycle:

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

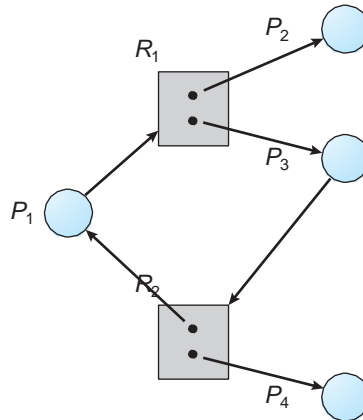


Figure 3.3 Resource-allocation graph with a cycle but no deadlock.

However, there is no deadlock. Observe that process P_4 may release its instance of resource type R_2 . That resource can then be allocated to P_3 , breaking the cycle.

In summary, if a resource-allocation graph does not have a cycle, then the system is *not* in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

3.5 Methods for Handling Deadlocks

Generally speaking, we can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including Linux and Windows. It is then up to the application developer to write programs that handle deadlocks.

3.5.1 Deadlock Prevention

As we noted previously, for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock. We elaborate on this approach by examining each of the four necessary conditions separately.

3.5.1.1 Mutual Exclusion

The mutual exclusion condition must hold. That is, at least one resource must be non-sharable. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

3.5.1.2 Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that we can use requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls. An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated. To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the

printer for its entire execution, even though it needs the printer only at the end. The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates. Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. In the example given, for instance, we can release the DVD drive and disk file, and then request the disk file and printer, only if we can be sure that our data will remain on the disk file. Otherwise, we must request all resources at the beginning for both protocols. Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

3.5.1.3 No Preemption

The third necessary condition for deadlocks is that there is no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting. Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources.

If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting. This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space.

3.5.1.4 Circular Wait

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration. To illustrate, we let $R = \{R_1, R_2, \dots, R_m\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function $F: R \rightarrow \mathbb{N}$, where \mathbb{N} is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows: $F(\text{tape drive}) = 1$ $F(\text{disk drive}) = 5$ $F(\text{printer}) = 12$ We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type — say, R_i . After that, the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$. For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer. Alternatively, we can require that a process requesting an instance of resource type R_j must have released any resources R_i such that $F(R_i) \geq F(R_j)$. Note also that if several instances of the same resource type are needed, a single request for all of them must be issued. If these two protocols are used, then the circular-wait condition cannot hold.

We can demonstrate this fact by assuming that a circular wait exists (proof by contradiction). Let the set of processes involved in the circular wait be $\{P_0, P_1, \dots, P_n\}$, where P_i is waiting for a resource R_i , which is held by process P_{i+1} . (Modulo arithmetic is used on the indexes, so that P_n is waiting for a resource R_n held by P_0 .) Then, since process P_{i+1} is holding resource R_i while requesting resource R_{i+1} , we must have $F(R_i) < F(R_{i+1})$ for all i . But this condition means that $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$. By transitivity, $F(R_0) < F(R_0)$, which is impossible. Therefore, there can be no circular wait. We can accomplish this scheme in an application program by developing an ordering among all synchronization objects in the system. All requests for synchronization objects must be made in increasing order. Keep in mind that developing an ordering, or hierarchy, does not in itself prevent deadlock. It is up to application developers to write programs that follow the

3.5.2 Deadlock Avoidance

Deadlock-prevention algorithms, as discussed in Section 1, prevent deadlocks by limiting how requests can be made. The limits ensure that at least one of the necessary conditions for deadlock cannot occur. Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with one tape drive and one printer, the system might need to know that

process P will request first the tape drive and then the printer before releasing both resources, whereas process Q will request first the printer and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. The various algorithms that use this approach differ in the amount

and type of information required. The simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need. Given this a priori information, it is possible to construct algorithm that ensures that the system will never enter a deadlocked state. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource- allocation *state* is defined by the number of available and allocated resources and the maximum demands of the processes. In the following sections, we explore two deadlock-avoidance algorithms.

3.5.2.1 Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes $\langle P_1, P_2, \dots, P_n \rangle$ is a safe sequence for the current allocation state if, for each P_i , the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j , with $j < i$. In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however (Figure 7.6). An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating system cannot prevent processes from requesting resources in such a way that a deadlock occurs. The behavior of the processes controls unsafe states. To illustrate, we consider a system with twelve magnetic tape drives and three processes: P_0 , P_1 , and P_2 . Process P_0 requires ten tape drives, process P_1 may need as many as four tape drives, and process P_2 may need up to nine tape drives. Suppose that, at time t_0 , process P_0 is holding five tape drives, process P_1 is holding two tape drives, and

process P2 is holding two tape drives.

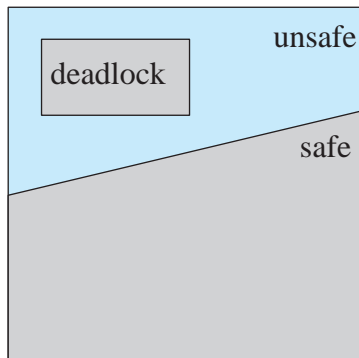


Figure 3.1 Safe, unsafe, and deadlocked state spaces.

	<u>Maximum Needs</u>	<u>Current Needs</u>
P_0	10	5
P_1	4	2
P_2	9	2

At time t_0 , the system is in a safe state. The sequence $\langle P_1, P_0, P_2 \rangle$ satisfies the safety condition. Process P_1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process P_0 can get all its tape drives and return them (the system will then have ten available tape drives); and finally process P_2 can get all its tape drives and return them (the system will then have all twelve tape drives available).

A system can go from a safe state to an unsafe state. Suppose that, at time t_1 , process P_2 requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process P_1 can be allocated all its tape drives. When it returns them, the system will have only four available tape drives. Since process P_0 is allocated five tape drives but has a maximum of ten, it may request five more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process P_2 may request six additional tape drives and have to wait, resulting in a deadlock. Our mistake was in granting the request from process P_2 for one more tape drive. If we had made P_2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

In this scheme, if a process requests a resource that is currently

available, it may still have to wait. Thus, resource utilization may be lower than it would otherwise be.

3.6 Resource-Allocation-Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph defined in Section () for deadlock avoidance. In addition to the request and assignment edges already described, we introduce a new type of edge, called a claim edge. A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process P_i requests resource R_j , the claim edge $P_i \rightarrow R_j$ is converted to a request edge. Similarly, when a resource R_j is released by P_i , the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$.

Note that the resources must be claimed a priori in the system. That is, before process P_i starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge $P_i \rightarrow R_j$ to be added to the graph only if all the edges associated with process P_i are claim edges.

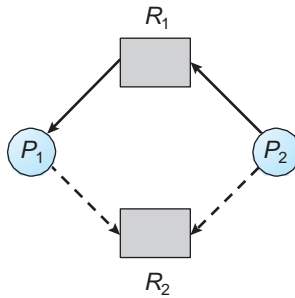


Figure 3.2 Resource-allocation graph for deadlock avoidance.

Now suppose that process P_i requests resource R_j . The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of processes in the system.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process P_i will have to wait for its requests to be satisfied. To illustrate this algorithm, we consider the resource-allocation graph of Figure 3.2. Suppose that P_2 requests R_2 . Although R_2 is currently free, we cannot allocate it to P_2 , since this action will create a cycle in the graph (Figure 3.3). A cycle, as mentioned, indicates that the system is in an unsafe state. If P_1 requests R_2 , and P_2 requests R_1 , then a deadlock will occur.

3.7 Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the **banker's algorithm**. The name was chosen because the algorithm could be used in a banking system to ensure that the bank never

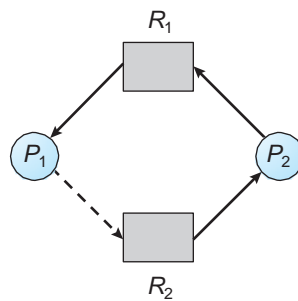


Figure 3.3 An unsafe state in a resource-allocation graph.

allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

- Available. A vector of length m indicates the number of available resources of each type. If $\text{Available}[j]$ equals k , then k instances of resource type R_j are available.
- Max. An $n \times m$ matrix defines the maximum demand of each process. If $\text{Max}[i][j]$ equals k , then process P_i may request at most k instances of resource type R_j .
- Allocation. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If $\text{Allocation}[i][j]$ equals k , then process P_i is currently allocated k instances of resource type R_j .
- Need. An $n \times m$ matrix indicates the remaining resource need of each process. If $\text{Need}[i][j]$ equals k , then process P_i may need k more instances of resource type R_j to complete its task. Note that $\text{Need}[i][j]$ equals $\text{Max}[i][j] - \text{Allocation}[i][j]$.

These data structures vary over time in both size and value.

To simplify the presentation of the banker's algorithm, we next establish some notation. Let X and Y be vectors of length n . We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, 2, \dots, n$. For example, if $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y \leq X$. In addition, $Y < X$ if $Y \leq X$ and $Y \neq X$.

We can treat each row in the matrices Allocation and Need as vectors and refer to them as Allocation i and Need i . The vector Allocation i specifies the resources currently allocated to process P_i ; the vector Need i specifies the additional resources that process P_i may still request to complete its task.

3.7.1 Safety Algorithm

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let **Work** and **Finish** be vectors of length m and n , respectively. Initialize **Work** = **Available** and **Finish** $[i] = \text{false}$ for $i = 0, 1, \dots, n - 1$.

2. Find an index i such that both a.

Finish $[i] == \text{false}$

b. **Need** $_i \leq \text{Work}$

If no such i exists, go to step

4.

3. **Work** = **Work** + **Allocation** $_i$

Finish $[i] =$

true

Go to step

2.

4. If **Finish** $[i] == \text{true}$ for all i , then the system is in a safe state.

This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

3.7.2 Resource-Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted.

Let $\mathbf{Request}_i$ be the request vector for process P_i . If $\mathbf{Request}_i[j] == k$, then process P_i wants k instances of resource type R_j . When a request for resources is made by process P_i , the following actions are taken:

1. If $\mathbf{Request}_i \leq \mathbf{Need}_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $\mathbf{Request}_i \leq \mathbf{Available}$, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$$\mathbf{Available} = \mathbf{Available} - \mathbf{Request}_i ;$$

$$\mathbf{Allocation}_i = \mathbf{Allocation}_i +$$

$$\mathbf{Request}_i ; \mathbf{Need}_i = \mathbf{Need}_i -$$

$$\mathbf{Request}_i ;$$

If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for $\mathbf{Request}_i$, and the old resource-allocation state is restored.

3.7.3 An Illustrative Example

To illustrate the use of the banker's algorithm, consider a system with five processes P_0 through P_4 and three resource types A , B , and C . Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that, at time T_0 , the following snapshot of the system has been taken:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the matrix *Need* is defined to be $Max - Allocation$ and is as follows:

	<u><i>Need</i></u>		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

We claim that the system is currently in a safe state. Indeed, the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies the safety criteria. Suppose now that process P_1 requests one additional instance of resource type A and two instances of resource type C , so $Request_1 = (1,0,2)$. To decide whether this request can be immediately granted, we first check that $Request_1 \leq Available$ — that is, that $(1,0,2) \leq (3,3,2)$, which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	<u><i>Allocation</i></u>	<u><i>Need</i></u>	<u><i>Available</i></u>
	A	B	C
P_0	0	1	0
P_1	3	0	2
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies the safety requirement. Hence, we can immediately grant the request of process P_1 .

You should be able to see, however, that when the system is in this state, a request for (3,3,0) by P_4 cannot be granted, since the resources are not available. Furthermore, a request for (0,2,0) by P_0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

We leave it as a programming exercise for students to implement the banker's algorithm

3.8 Deadlock Detection

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 may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

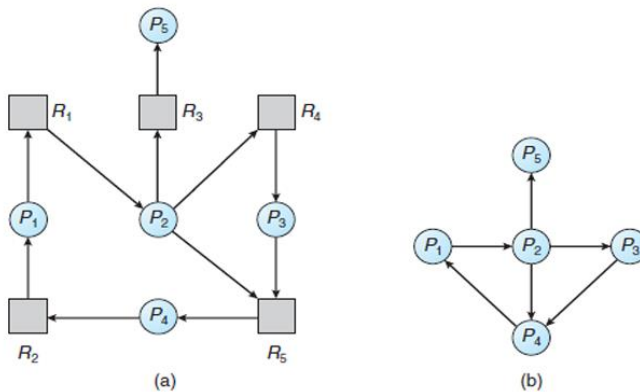


figure 3.8 : (a) Resource-allocation graph. (b) Corresponding wait-for graph.

In the following discussion, we elaborate on these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type. At this point, however, we note that a detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm but also the potential losses inherent in recovering from a deadlock.

3.8.1 Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock-detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for** graph. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

More precisely, an edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q . In Figure 7.9, we present a resource-allocation graph and the corresponding wait-for graph.

As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to *maintain* the wait-for graph and periodically *invoke an algorithm* that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

3.8.2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock-

detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm.

- **Available.** A vector of length m indicates the number of available resources of each type.
- **Allocation.** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request.** An $n \times m$ matrix indicates the current request of each process. If $\mathbf{Request}[i][j]$ equals k , then process P_i is requesting k more instances of resource type R_j .

The \leq relation between two vectors is defined as in Section 7.5.3. To simplify notation, we again treat the rows in the matrices *Allocation* and *Request* as vectors; we refer to them as $\mathbf{Allocation}_i$ and $\mathbf{Request}_i$. The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed. Compare this algorithm with the banker's algorithm.

1. Let *Work* and *Finish* be vectors of length m and n , respectively. Initialize $\mathbf{Work} = \mathbf{Available}$. For $i = 0, 1, \dots, n-1$, if $\mathbf{Allocation}_i = 0$, then $\mathbf{Finish}[i] = \mathbf{false}$. Otherwise, $\mathbf{Finish}[i] = \mathbf{true}$.
2. Find an index i such that both a.
 - $\mathbf{Finish}[i] == \mathbf{false}$
 - b. $\mathbf{Request}_i \leq \mathbf{Work}$
 If no such i exists, go to step 4.
3. $\mathbf{Work} = \mathbf{Work} + \mathbf{Allocation}_i$
 $\mathbf{Finish}[i] = \mathbf{true}$
 Go to step 2.
4. If $\mathbf{Finish}[i] == \mathbf{false}$ for some $i, 0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $\mathbf{Finish}[i] == \mathbf{false}$, then process P_i is deadlocked.

This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

You may wonder why we reclaim the resources of process P_i (in step 3) as soon as we determine that $\mathbf{Request}_i \leq \mathbf{Work}$ (in step 2b). We know that P_i is currently *not* involved in a deadlock (since $\mathbf{Request}_i \leq \mathbf{Work}$). Thus, we take

an optimistic attitude and assume that P_i will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time the deadlock-detection algorithm is invoked.

To illustrate this algorithm, we consider a system with five processes P_0 through P_4 and three resource types A , B , and C . Resource type A has seven instances, resource type B has two instances, and resource type C has six instances. Suppose that, at time T_0 , we have the following resource-allocation state:

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ results in $Finish[i] == true$ for all i .

Suppose now that process P_2 makes one additional request for an instance of type C . The **Request** matrix is modified as follows:

	<u>Request</u>
	A B C P_0
	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process P_0 , the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P_1, P_2, P_3 , and P_4 .

3.8.3 Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:

1. How *often* is a deadlock likely to occur?
2. How *many* processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

Deadlocks occur only when some process makes a request that cannot be granted immediately.

This request may be the final request that completes a chain of waiting processes. In the extreme, then, we can invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately. In this case, we can identify not only the deadlocked set of processes but also the specific process that “caused” the deadlock. (In reality, each of the deadlocked processes is a link in the cycle in the resource graph, so all of them, jointly, caused the deadlock.) If there are many different resource types, one request may create many cycles in the resource graph, each cycle completed by the most recent request and “caused” by the one identifiable process.

Of course, invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at defined intervals — for example, once per hour or whenever CPU utilization drops below 40 percent. (A deadlock eventually cripples system throughput and causes CPU utilization to drop.) If the detection algorithm is invoked at arbitrary points in time, the resource graph may contain many cycles. In this case, we generally cannot tell which of the many deadlocked processes “caused” the deadlock.

3.9 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

3.9.1 Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes.** This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated.** This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job. If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may affect which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

3.9.2 Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim.** Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
2. **Rollback.** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.

Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

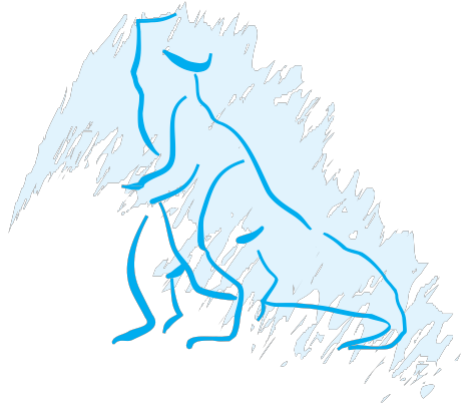
3. **Starvation.** How do we ensure that starvation will not occur? That is,

how can we guarantee that resources will not always be preempted from the same process?

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation any practical system must address. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Chapter-4-

Mass-Storage Structure



4.1 Overview of Mass-Storage Structure

In this section, we present a general overview of the physical structure of secondary and tertiary storage devices.

4.1.1 Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Conceptually, disks are relatively simple (Figure 10.1). Each disk **platter** has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 3.5 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.

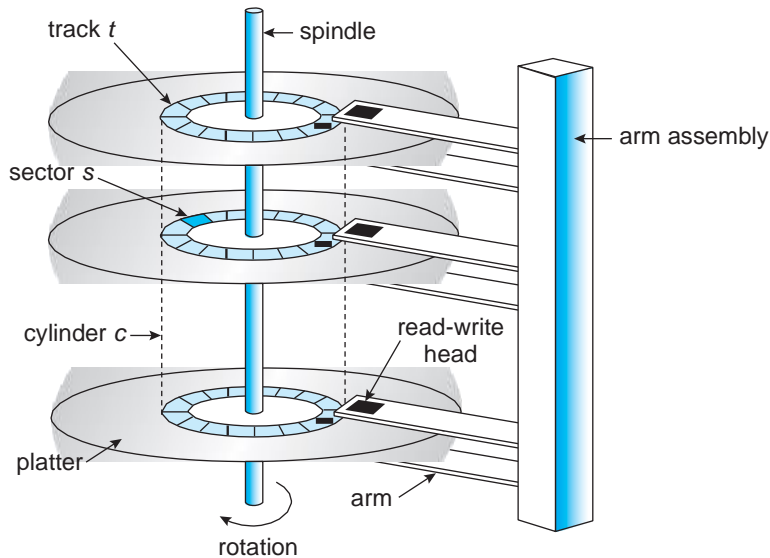


Figure 4.1 Moving-head disk mechanism.

A read–write head “flies” just above each surface of every platter. The heads are attached to a **disk arm** that moves all the heads as a unit. The surface of a platter is logically divided into circular **tracks**, which are subdivided into **sectors**. The set of tracks that are at one arm position makes up a **cylinder**. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of common disk drives is measured in gigabytes. When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 250 times per second, specified in terms of rotations per minute (**RPM**). Common drives spin at 5,400, 7,200, 10,000, and 15,000 RPM. Disk speed has two parts. The **transfer rate** is the rate at which data flow between the drive and the computer. The **positioning time**, or **random-access time**, consists of two parts: the time necessary to move the disk arm to the desired cylinder, called the **seek time**, and the time necessary for the desired sector to rotate to the disk head, called the **rotational latency**. Typical disks can transfer several megabytes of data per second, and they have seek times and rotational latencies of several milliseconds.

4.4 Disk Scheduling

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. For magnetic disks, the access time has two major components, as mentioned in Section 10.1.1. The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector. The **rotational latency** is the additional time for the disk to rotate the desired sector to the disk head. The disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by managing the order in which disk I/O requests are serviced.

Whenever a process needs I/O to or from the disk, it issues a system call to the operating system. The request specifies several pieces of information:

- Whether this operation is input or output
- What the disk address for the transfer is
- What the memory address for the transfer is
- What the number of sectors to be transferred is

If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive. For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed, the operating system chooses which pending request to service next. How does the operating system make this choice? Any one of several disk-scheduling algorithms can be used, and we discuss them next.

4.4.1 FCFS Scheduling

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders

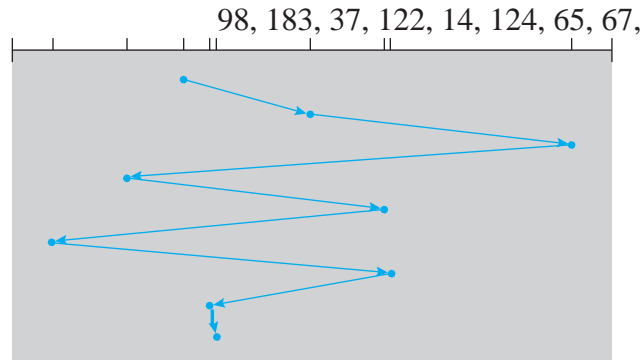


Figure 4.4 FCFS disk scheduling.

in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders. This schedule is diagrammed in Figure 4.4. The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests for 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

4.4.2 SSTF Scheduling

It seems reasonable to service all the requests close to the current head position before moving the head far away to service other requests. This assumption is the basis for the shortest-seek-time-first (SSTF) algorithm. The SSTF algorithm selects the request with the least seek time from the current head position. In other words, SSTF chooses the pending request closest to the current head position.

For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than the

one at 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98, 122, 124, and finally 183 (Figure 4.5). This scheduling method results in a total head movement of only 236 cylinders — little more than one-third of the distance needed for FCFS scheduling of this request queue. Clearly, this algorithm gives a substantial improvement in performance.

SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling; and like SJF scheduling, it may cause starvation of some requests. Remember that requests may arrive at any time. Suppose that we have two requests in the queue, for cylinders 14 and 186, and while the request from 14 is being serviced, a new request near 14 arrives. This new request will be serviced next, making the request at 186 wait. While this request is being serviced, another request close to 14 could arrive. In theory, a continual stream of requests near one another could cause the request for cylinder 186 to wait indefinitely.

queue = 98, 183, 37, 122, 14, 124, 65, 67 head starts at 53

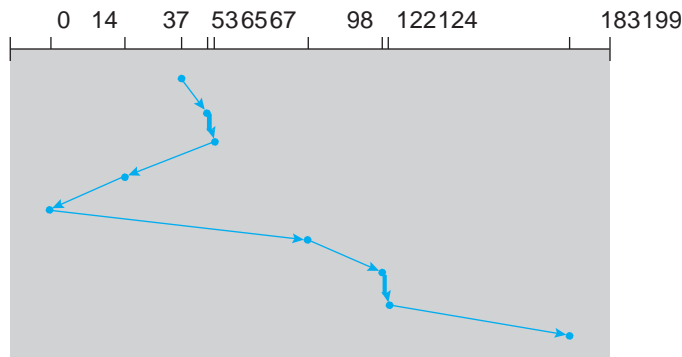


Figure 4.5 SSTF disk scheduling.

This scenario becomes increasingly likely as the pending-request queue grows longer. Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, it is not optimal. In the example, we can do better by moving the head from 53 to 37, even though the latter is not closest, and then to 14, before turning around to service 65, 67, 98, 122, 124, and 183. This strategy reduces the **total head movement to 208 cylinders**.

4.4.3 SCAN Scheduling

In the SCAN algorithm, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the elevator algorithm, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

Let's return to our example to illustrate. Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know the direction of head movement in addition to the head's current position. Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 (Figure 4.6). If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

Assuming a uniform distribution of requests for cylinders, consider the density of requests when the head reaches one end and reverses direction. At this point, relatively few requests are immediately in front of the head, since these cylinders have recently been serviced. The heaviest density of requests

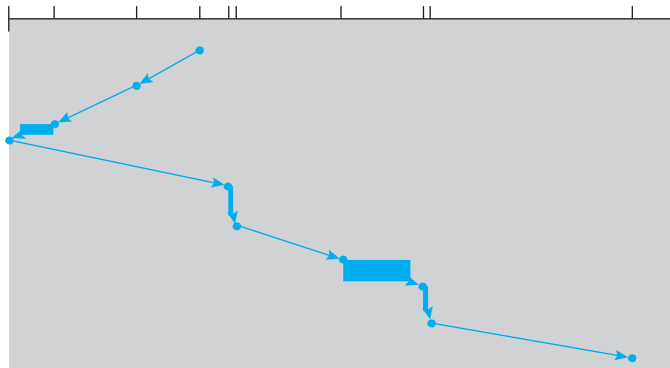


Figure 4.6 SCAN disk scheduling.

is at the other end of the disk. These requests have also waited the longest, so why not go there first? That is the idea of the next algorithm.

4.4.4 C-SCAN Scheduling

Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip (Figure 4.7). The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

queue = 98, 183, 37, 122, 14, 124, 65, 67 head starts at 53

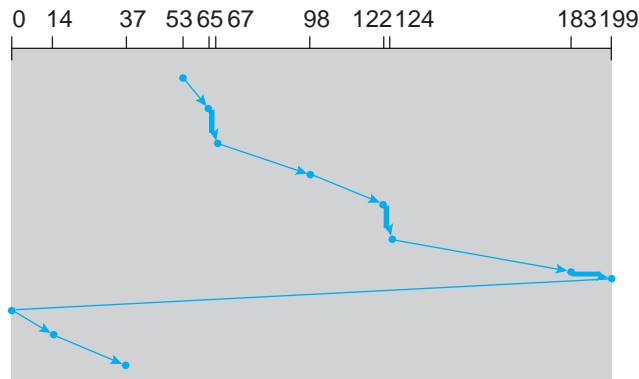


Figure 4.7 C-SCAN disk scheduling.

4.4.5 LOOK Scheduling

As we described them, both SCAN and C-SCAN move the disk arm across the full width of the disk. In practice, neither algorithm is often implemented this way. More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk. Versions of SCAN and C-SCAN that follow this pattern are called **LOOK** and **C-LOOK scheduling**, because they *look* for a request before continuing to move in a given direction (Figure 4.8).

queue = 98, 183, 37, 122, 14, 124, 65, 67 head starts at 53

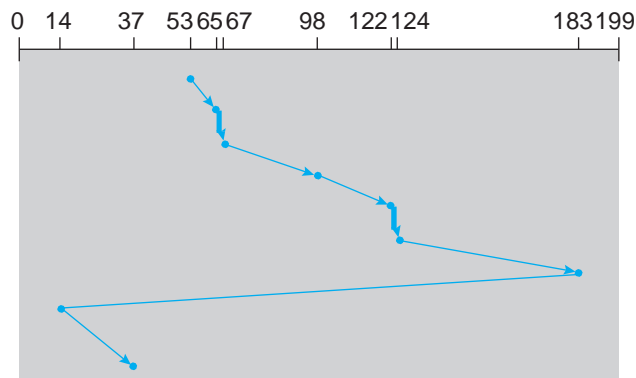


Figure 4.8 C-LOOK disk scheduling.