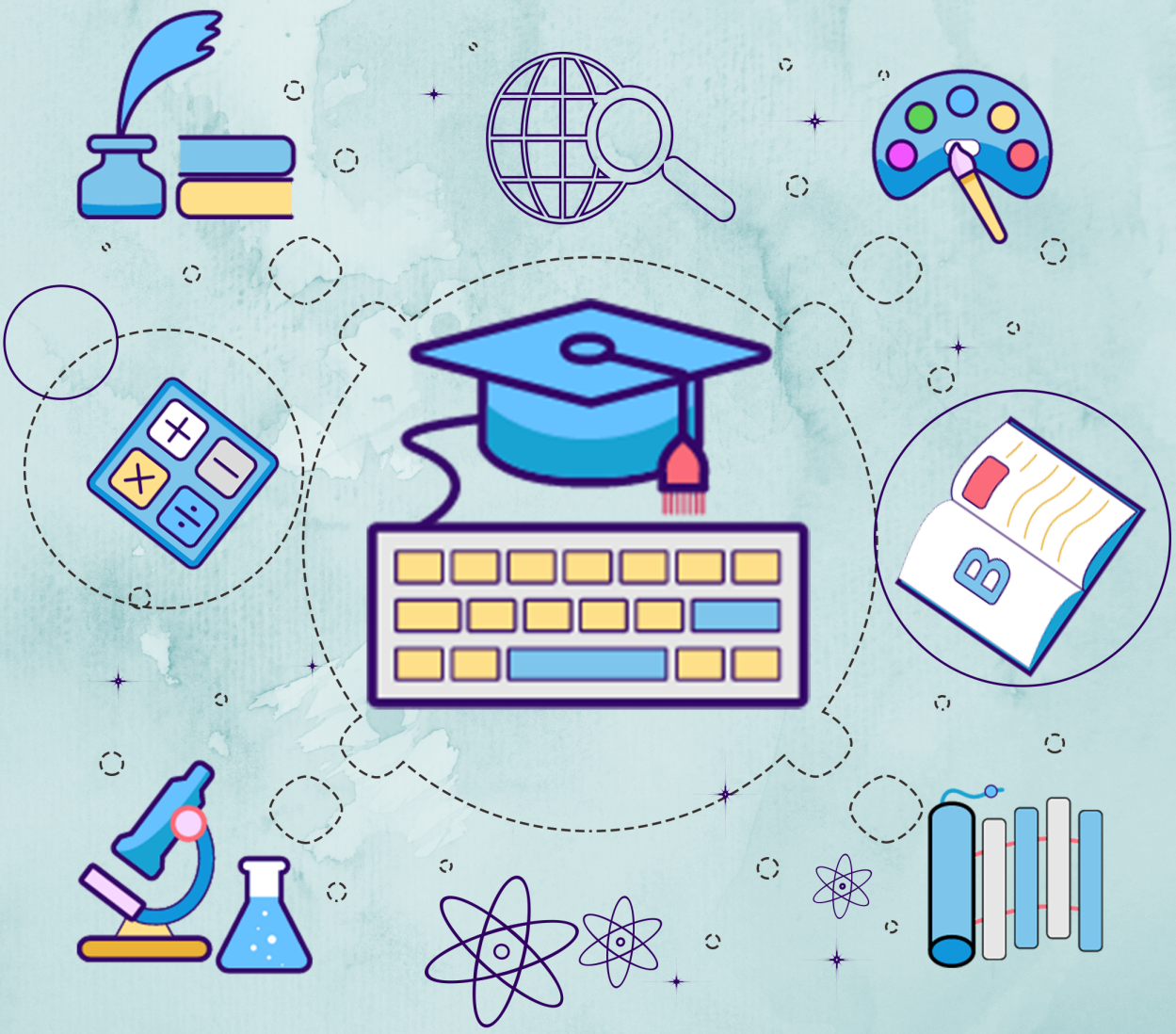
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KeralaNotes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

KTU S4 CSE NOTES

OPERATING SYSTEM (CST200)

Module 3

Related Link :

- KTU S4 CSE NOTES 2019 SCHEME
- KTU S4 SYLLABUS CSE COMPUTER SCIENCE
- KTU PREVIOUS QUESTION BANK S4 CSE SOLVED
- KTU CSE TEXTBOOKS S4 B.TECH PDF DOWNLOAD
- KTU S4 CSE NOTES | SYLLABUS | QBANK | TEXTBOOKS DOWNLOAD

MODULE 3
PROCESS SYNCHRONISATION

Synchronization

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. 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 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.

Suppose we want to modify the producer-consumer algorithm as follows. One possibility is to add an integer variable counter, initialized to0. 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 nextProduced*/
while (counter == BUFFER_SIZE)
; /* do nothing */
buffer[in] = nextProduced;
in= (in + 1) % BUFFER_SIZE ;
counter++;
```

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

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

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
register1 = counter
register1 = *register1* + 1
counter= *register1*

where*register1*is one of the local CPU registers. Similarly, the statement*register2"counter--"* is implemented as follows:
register2 = counter
register2 = *register2* ~ 1
counter= *register2*

where again *register2* is on eo£ the local CPU registers.

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:

To:	*producer*	execute	*register1* =counter	*{register1* = 5}
T1:	*producer*	execute	*register1* = *register1* + 1	*{register1* = 6}
T2:	*consumer*	execute	*register2* = counter	*{register2* = 5}
T3:	*consumer*	execute	*register2* = *register2* 1	*{register2* = 4}
T4:	*producer*	execute	counter= *register1*	*{counter* = 6}
Ts:	*consumer*	execute	counter = *register2*	*{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 theorder of the statements at T4 and *T5,* 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.

3.1 The Critical Section Problem

Consider a system consisting of n processes {Po, P1 , ... , P11 _ l}. Each process has a segment of code, called a **critical section** in which the process may be

changing common variables, updating a table, writing a file, and so on.The important feature of the system is that, **when one process is executing in its critical section, no other process is to be allowed to execute in its critical section**. That is, no two processes are executing in their critical sections at the same time. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section.** The remaining code is the **remainder section**. The general structure of a typical process Pi is shown in fig:3.1

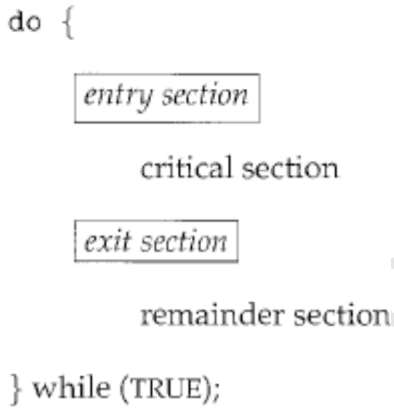


Fig:3.1General structure of a typical process Pi

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

1. **Mutual exclusion.** If process *Pi* 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 a process has made a request to enter its critical section and before that request is granted.

At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system*(kernel code)* is subject to several possible race conditions. Two general approaches are used to handle critical sections in operating systems: (1) **preemptive kernels** and (2) **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.

3.2 Peterson's Solution (Software based solution for critical section)

Next, we illustrate a classic software-based solution to the critical-section problem known as Peterson's solution. 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.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered *Po* and P1. For convenience, when presenting *Pi,* we use *Pj* 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 *Pi* 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 Pi 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 3.2

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Fig:3.2 The structure of process Pi in Peterson's solution.

To enter the critical section, process *Pi* 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 since 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:

- Mutual exclusion is preserved.
- The progress requirement is satisfied.
- The bounded-waiting requirement is met.

To prove property 1, we note that each P; enters its critical section only if either flag [j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag [0] == flag [1] ==true. These two observations imply that Po and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes -say, Pi -must have successfully executed the while statement, whereas Pi had to execute at least one additional statement ("turn== j"). However, at that time, flag [j] == true and turn == j, and this condition will persist as long as Pi is in its critical section; as a result, mutual exclusion is preserved.

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

3.3 Synchronization Hardware

Hardware based solution for critical section problem

However, as mentioned, software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. Instead, we can generally state that any solution to the critical-section problem requires a simple tool-a **lock.** Race conditions are prevented by requiring that critical regions be protected by locks. That is, **a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.**This is illustrated in Figure 3.3.

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

Fig:3.3 Solution to the critical-section problem using locks.

Hardware instructions for solving critical section problem

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. Rather than discussing one specific instruction for one specific machine, we abstract the main concepts behind these types of instructions by describing the **TestAndSet ()** and **Swap()** instructions.

The TestAndSet () instruction can be defined as shown in Figure 3.4. The important characteristic of this instruction is that it is executed atomically. Thus, if two TestAndSet () instructions are executed simultaneously (each ona different CPU), they will be executed sequentially in some arbitrary order. Ifthe machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process *P;* is shown in Figure 3.5.

```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Fig:3.4The definition of the TestAndSet () instruction

```
do {
   while (TestAndSet(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

Fig: 3.5 Mutual-exclusion implementation with TestAndSet ().

The Swap() instruction, in contrast to the TestAndSet () instruction, operates on the contents of two words; it is defined as shown in Figure3.6. Like the TestAndSet () instruction, it is executed atomically. If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows. A global Boolean variable lock is declared and is initialized to false. In addition, each process has a local Boolean variable key. The structure of process *P;* is shown in Figure 3.7.

```
void Swap(boolean *a, boolean *b) {
   boolean temp = *a;
   *a = *b;
   *b = temp;
}
```

Fig:3.6 The definition of the Swap () instruction

```
do {
   key = TRUE;
   while (key == TRUE)
      Swap(&lock, &key);

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

Fig:3.7 Mutual-exclusion implementation with the Swap() instruction

3.4 Semaphores

The hardware-based solutions to the critical-section problem presented in Section 3.3 are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a **semaphore**.

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
       ; // no-op
    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 :S 0), as well as its possible modification (S--), must be executed without interruption.

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. On some systems, **binary semaphores are known as mutex locks**, as they are locks that provide mutual exclusion.

1) Solution for critical section problem using binary semaphores

We can use binary semaphores to deal with the critical-section problem £or multiple processes. Then n processes share a semaphore, mutex, initialized to 1. Each process *Pi* is organized as shown in Figure 3.8

```
do {
   wait(mutex);

      // critical section

   signal(mutex);

      // remainder section
} while (TRUE);
```

Fig:3.8Mutual-exclusion implementation with semaphores

2) Use of counting semaphores

Counting semaphores can be used to control access to a given resource consisting of a finite number o£ 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) Solving synchronization problems

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently numing processes: P1 with a statement S1 and P2 with a statement S2 .Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0, and by inserting the statements

S1;
signal(synch) ;

in process P1 and the statements

wait(synch);
S2;

in process P2. Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal (synch), which is after statement S1 has been executed.

Implementation

The main disadvantage of the semaphore definition given here is that it requires **While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code**. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spin lock** because the process "spins" while waiting for the lock. (Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful; they are often employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.)

To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations. When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can *block* itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a **wakeup ()** operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

To implement semaphores under this definition, we define a semaphore as a "C' struct:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.

The wait() semaphore operation can now be defined as

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            block();
        }
}
```

The signal () semaphore operation can now be defined as

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

Note that in this implementation, semaphore values may be negative, although semaphore values are never negative under the classical definition of semaphores with busy waiting. **If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore**. This fact results from switching the order of the decrement and the test in the implementation of the wait () operation.

Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be deadlocked. To illustrate this, we consider a system consisting of two processes, *Po* and *P1*, each accessing two semaphores, S and Q, set to the value 1:

```
        P0                  P1
wait(S);            wait(Q);
wait(Q);            wait(S);
   .                   .
   .                   .
   .                   .
signal(S);          signal(Q);
signal(Q);          signal(S);
```

Suppose that *Po* executes wait (S) and then P1 executes wait (Q). When *Po* executes wait (Q), it must wait until P1 executes signal (Q). Similarly, when P1 executes wait (S), it must wait until *Po* executes signal(S). Since these signal() operations cam1ot be executed, *Po* and P1 are deadlocked.

We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are *resource acquisition and release.* Another problem related to deadlocks is or a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Priority Inversion

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process-or a chain of lower-priority processes. Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the

resource. The situation becomes more complicated if the lower-priority process is preempted in favor of another
process with a higher priority. As an example, assume we have three processes, *L, M,* and *H,* whose priorities follow the order *L <M <H.* Assume that process *H* requires resource *R,* which is currently being accessed by process *L* Ordinarily, process *H* would wait for *L* to finish using resource *R.* However, now suppose that process *M* becomes runnable, thereby preempting process *L.* Indirectly, a process with a lower priority-process M-has affected howlong process *H* must wait for *L* to relinquish resource *R.* This problem is known as **priority inversion** Itoccurs only in systems with more than two priorities, so one solution is to have only two priorities. That is insufficient for most general-purpose operating systems, however.

Typically these systems solve the problem by implementing a **priority inheritance protocol**. According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values. In the exan1.ple above, a priority-inheritance protocol would allow process *L* to temporarily inherit the priority of process H, thereby preventing process *M* from preempting its execution. When process *L* had finished using resource *R,* it would relinquish its inherited priority from Hand assume its original priority. Because resource *R* would now be available, process H-not M-would run next.

3.5 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.

3.5.1 The Bounded-Buffer Problem

The *bounded-buffer problem* is related to the *producer – consumer problem.* 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 3.9; the code for the consumer process is shown in Figure 3.10. 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 {
    . . .
   // produce an item in nextp
    . . .
   wait(empty);
   wait(mutex);
    . . .
   // add nextp to buffer
    . . .
   signal(mutex);
   signal(full);
} while (TRUE);
```

Fig:3.9 The structure of the producer process.

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

Fig:3.9 The structure of the consumer process.

3.5.2 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, problems may arise.

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 performs 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.

In the solution to the first readers-writers problem, the reader processes share the following data structures:

Semaphore mutex, wrt;

Int read count;

The semaphores mutex and wrt are initialized to 1; read count is initialized to 0.The semaphore wrt is common to both reader and writer 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 wrt 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 writer process is shown in Figure 3.10; the code for a reader process is shown in Figure 3.11. Note that, if a writer is in the critical section and *n* readers are waiting, then one reader is queued on wrt, and *n*- 1 readers are queued on mutex. Also observe that, when a writer executes signal (wrt), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
} while (TRUE);
```

Fig 3.10 The structure of a writer process

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.**

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
} while (TRUE);
```

Fig 3.10 The structure of a reader process

3.5.3 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 3.11). 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 cam1ot 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 her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

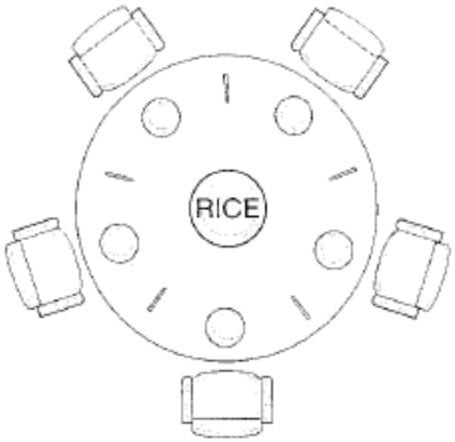


Fig 3.11 The situation of the dining philosophers

The *dining-philosophers problem* 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];
```

where all the elements of chopstick are initialized to 1. The structure of philosopher *i* is shown in Figure 3.12

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   // think
      . . .
} while (TRUE);
```

Fig:3.12 The structure of philosopher *i*..

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because **it could create a deadlock**. Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are listed next.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick

3.6 Monitors

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place and these sequences do not always occur.

- Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

signal(mutex);
critical section
wait(mutex);

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a process replaces signal (mutex) with wait (mutex). That is, it executes

wait(mutex);
critical section
wait(mutex);

In this case, a deadlock will occur.

- Suppose that a process omits the wait (mutex), or the signal (mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem.

To deal with such errors, researchers have developed high-level language constructs. In this section, we describe one fundamental high-level synchronization construct-the **monitor** type.

Usage

A*abstract data type*- or ADT- encapsulates private data with public methods to operate on that data**. A *monitor type* is an ADT which presents a set of programmer-defined operations that are provided mutual exclusion within the monitor**. The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables. The syntax of a monitor type is shown in Figure 3.13. The representation of a monitor type cannot be used directly by the various processes. Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local procedures.

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
         .
         .

   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

Fig 3.13 Syntax of a monitor

The monitor construct ensures that only one process at a time is active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly (Figure 3.14). Howeverthe monitor construct, as defined so faris not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition:*

condition x, y;

The only operations that can be invoked on a condition variable are wait () and signal(). The operation

x. wait();

means that the process invoking this operation is suspended until another process invokes

x. signal();

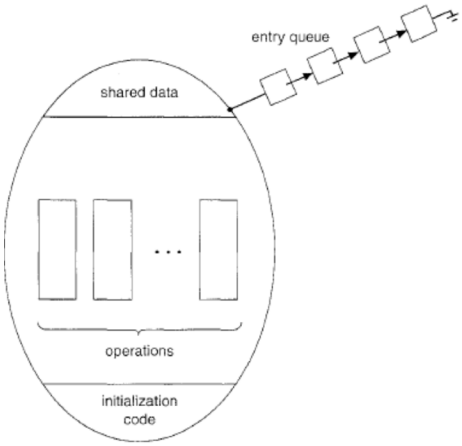


Fig:3.14 Schematic view of a monitor

The x. signal() operation resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect; that is, the state of x is the same as if the operation had never been executed (Figure3.15). Contrast this operation with the signal() operation associated with semaphores, which always affects the state of the semaphore.

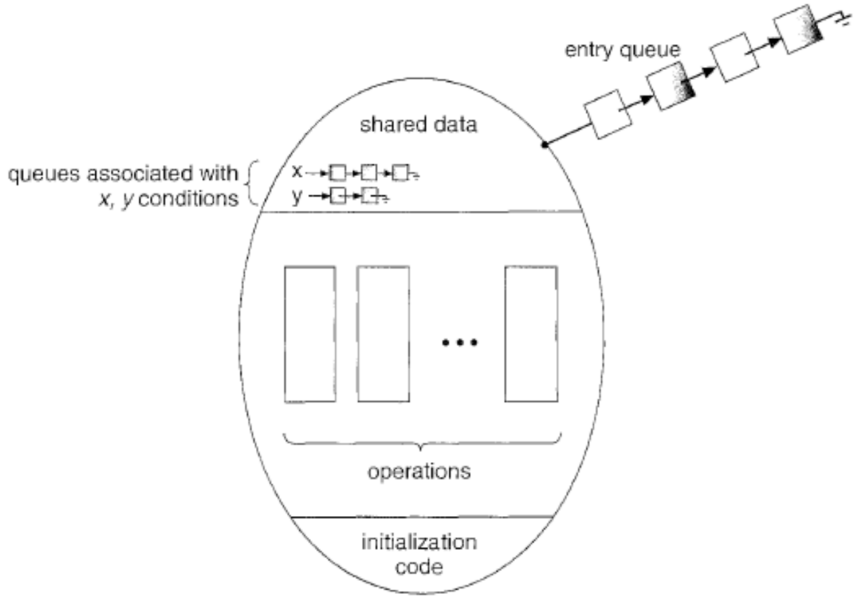


Fig :3.15 Monitor with condition variables

Now suppose that, when the x. signal () operation is invoked by a process P, there exists a suspended process Q associated with condition x. Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor. Note, however, that both processes can conceptually continue with their execution. Two possibilities exist:

- **Signal and wait.** *P* either waits until *Q* leaves the monitor or waits for another condition.

- **Signal and continue.** *Q* either waits until *P* leaves the monitor or waits for another condition.

There are reasonable arguments in favor of adopting either option. On the one hand, since *P* was already executing in the monitor, the *signal-and-continue* method seems more reasonable. On the other hand, if we allow thread *P* to continue, then by the time *Q* is resumed, the logical condition for which *Q* was waiting may no longer hold. A compromise between these two choices was adopted in the language Concurrent Pascal. When thread *P* executes the signal operation, it immediately leaves the monitor. Hence, *Q* is immediately resumed.

3.6.2Dining-Philosophers Solution Using Monitors

Next, we illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

enum{THINKING, HUNGRY, EATING}state[5];

Philosopher i can set the variable state [i] = EATING only if her two neighbors are not eating: ie if

(state [(i +4) % 5] ! = EATING) and (state [(i +1)% 5] '= EATING).

We also need to declare

condition self[5];

in which philosopher i can delay herself when she is hungry but is unable to obtain the chopsticks she needs.

We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor DiningPhilosophers, whose definition is shown in Figure 3.16 . Each philosopher, before starting to eat, must invoke the operation **pickup().** This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the put down() operation. Thus, philosopher *i* must invoke the operationspickup() and put down() in the following sequence:

DiningPhilosophers.pickup(i);
eat
DiningPhilosophers.putdown(i);

```
monitor dp
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
       (state[i] == HUNGRY) &&
       (state[(i + 1) % 5] != EATING)) {
         state[i] = EATING;
         self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

Fig:3.16 A monitor solution to the dining philosopher problem

3.6.3 Implementing a Monitor Using Semaphores

We now consider a possible implementation of the monitor mechanism using semaphores. For each monitor, a semaphore mutex(initialized to **1)** is provided. A process must execute wait (mutex) before entering the n1onitor and must execute signal (mutex) after leaving the monitor.

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0. The signaling processes can use next to suspend themselves. An integer variable ext_count is also provided to count the number of processes suspended on next. Thus, each external procedure F is replaced by

```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
   signal(next);
else
   signal(mutex);
```

Mutual exclusion within a monitor is ensured.

We can now describe how condition variables are implemented as well. For each condition x, we introduce a semaphore x_sem and an integer variable x_count, both initialized to 0. The operation x. wait() can now beimplemented as

```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

The operation x. signal() can be implemented as

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

3.6.4 Resuming Processes within a Monitor

If several processes are suspended on condition x, and an x. signal() operation is executed by some process, then how do we determine which of the suspended processes should be resumed next? One simple solution is to use an FCFS ordering, so that the process that has been waiting the longest is resumed first. In many circumstances, however, such a simple scheduling scheme is not adequate. For this purpose, the **conditional –wait**constructcan be used; it hasthe form

x.wait(c);

where c is an integer expression that is evaluated when the wait () operation is executed. The value of c, which is called a **priority number** is then stored with the name of the process that is suspended. When x. signal () is executed,the process with the smallest priority number is resumed next.

To illustrate this new mechanism, consider the **Resource Allocator monitor** shown in Figure 3.17, which controls the allocation of a single resource among competing processes.

Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest time-allocation request.

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = TRUE;
   }

   void release() {
      busy = FALSE;
      x.signal();
   }

   initialization_code() {
      busy = FALSE;
   }
}
```

A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
    ...
 access the resource;
    ...
R.release();
```

where R is an instance of type ResourceAllocator.

MODULE 3

DEADLOCKS

- Necessary conditions
- Resource allocation graphs
- Deadlock prevention
- Deadlock avoidance – Banker's algorithms, Deadlock detection,Recovery from deadlock

DEADLOCKS

In a multiprogramming environment, several processes may compete for afinite number of resources. A process requests resources; if the resources arenot available at that time, the process enters a waiting state. Sometimes, awaiting process is never again able to change state, because the resources ithas requested are held by other waiting processes. This situation is calleda **deadlock**

System Model

A system consists of a finite number of resources to be distributed amonga number of competing processes. The resources are partitioned into severaltypes, each consisting of some number of identical instances. Memory space,CPU cycles, files, and I/0 devices (such as printers and DVD drives) are examplesof resource types. If a system has two CPUs, then the resource type *CPU* hastwo instances. Similarly, the resource type *printer* may have five instances.

Under the normal mode of operation, a process may utilize a resource inonly the following sequence:

- Request. The process requests the resource. If the request cannot begranted immediately (for example, if the resource is being used by anotherprocess), then the requesting process must wait until it can acquire theresource.
- Use. The process can operate on the resource (for example, if the resourceis a printer, the process can print on the printer).
- Release. The process releases the resource.

A set of processes is in a deadlocked state when every process in the set iswaiting for an event that can be caused only by another process in the set. Theevents with which we are mainly concerned here are resource acquisition andrelease. The resources may be either physical resources (for example, printers,tape drives, memory space, and CPU cycles) or logical resources (for example,files, semaphores, and monitors).

Deadlock Characterization

Necessary Conditions

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

- **Mutual exclusion.** At least one resource must be held in a nonsharablemode; that is, only one process at a time can use the resource. If anotherprocess requests that resource, the requesting process must be delayeduntil the resource has been released.
- **Hold and wait**. A process must be holding at least one resource andwaiting to acquire additional resources that are currently being held byother processes.
- **No preemption.** Resources cannot be preempted; that is, a resource canbe released only voluntarily by the process holding it, after that processhas completed its task.
- **Circular wait**. A set { P0 , Pl, ... , P11 } of waiting processes must exist suchthat Po is waiting for a resource held by P1, P1 is waiting for a resourceheld by P2, ... , Pn-1 is waiting for a resource held by Pn and Pn is waitingfor a resource held by Po.

We emphasize that all four conditions must hold for a deadlock to occur.

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* == { P1, P2, ... , Pn},** the set consisting of all the active processes in the system, and ***R* == {R1, R2, ... , Rm}** the set consisting of all resource types in the system.

A directed edge from process Pi to resource type Rj is denoted by Pi ->Rj it signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource. A directed edge from resource type Rj to process Pi is denoted by Rj-> Pi it signifies that an instance of resource type Rj has been allocated to process Pi. A directed edge Pi-> Rj is called a **request edge**; a directed edge Rj->Pi is called an **assignment edge** .

Pictorially we represent each process Pi as a circle and each resource type Rj as a rectangle. Since resource type Ri 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 Rj, whereas an assignment edge must also designate one of the dots in the rectangle.

When process *P;* requests an instance of resource type *Ri,* 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 3.17 depicts the following situation.

The sets *P,* R and *E:*

- P == {P1, P2, P3}
- R== {R1, R2, R3, R4}
- E == {Pl->R1, P2-> R3, Rl-> P2, R2-> P2, R2-> Pl, R3-> P3}

Resource instances:

- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type R3
- Three instances of resource type R4

Process states:

- Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1 .
- Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.

- Process *P3* is holding an instance of R3 .

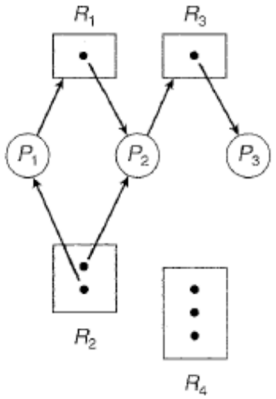


Fig 3.17:Resource allocation graph

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 3.17. Suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge P3 ->R2 is added to the graph (Figure 3.18). At this point, two minimal cycles exist in the system:

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$
$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

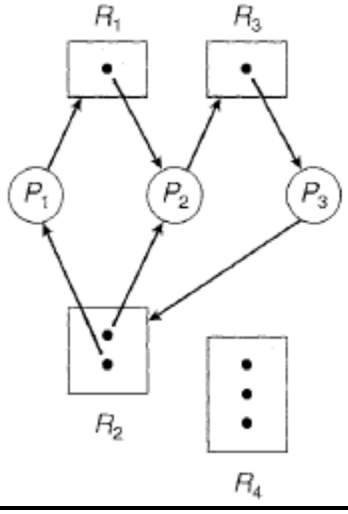


Fig:3.18 Resource allocation graph with a deadlock

Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

Now consider the resource-allocation graph in Figure 3.19. In this example, we also have a cycle:

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

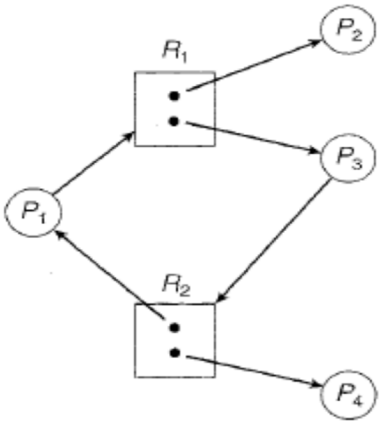


Fig:3.19 Resource-allocation graph with a cycle but no deadlock

However, there is no deadlock. Observe that process *P4* may release its instance of resource type R2. That resource can then be allocated to P3, 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.**

3.7.3 Methods for handling deadlock

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.

1)Deadlock Prevention

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.

Mutual Exclusion: The mutual-exclusion condition must hold for nonsharable resources. For example, a printer cannot be simultaneously shared by several

processes. 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 nonsharable.

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 can be used requires each process to request and be allocated all its resources before it begins execution. **Second type of 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, however, 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 again 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 again request the disk file and printe1~ 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.

No Preemption: The third necessary condition for deadlocks is that there be 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.

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* = { R1, R2, ... , Rm} 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 ->N,* where *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 (tape drive) = 1
F (disk drive) = 5
F (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, *Ri* After that, the process can request instances of resource type *Rj* if and only if F(Rj) > F(Ri).** 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 *Rj* must have released any resources *Ri* such that F(Ri) > =F(Rj).** It must also be noted 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.

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 ordering. Also note that **the function *F* should be defined according to the normal order of usage of the resources in a system.** For example, because the tape drive is usually needed before the printer, it would be reasonable to define F(tape drive) < F(printer).

2)Deadlock Avoidance

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. 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.

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 <P1, P2, ... , Pn> is a safe sequence for the current allocation state if, for each
Pi, the resource requests that *Pi* can still make can be satisfied by the currently available resources plus the resources held by all *Pj,* with j < i. In this situation, if the resources that *Pi* needs are not immediately available, then *Pi* can wait until all *Pj* have finished. When they have finished, *Pi* can obtain all of its
needed resources, complete its designated task, return its allocated resources, and terminate. When *Pi* terminates, Pi+I 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, (Figure 3.20). **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.

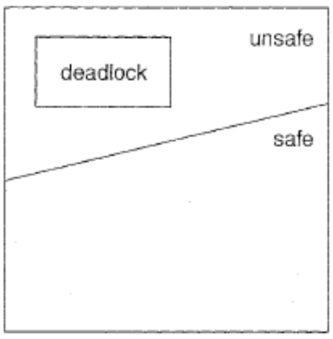


Fig:3.20 Safe, unsafe, and deadlocked state spaces.

To illustrate, we consider a system with twelve magnetic tape drives and three processes: *Po,* P1, and P2. Process *Po* requires ten tape drives, process P1 may need as many as four tape drives, and process P2 may need up to nine tape drives. Suppose that, at time *to,* process *Po* is holding five tape drives, process P1 is holding two tape drives, and process P2 is holding two tape drives. (Thus, there are three free tape drives.)

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

At time *t0,* the system is in a safe state. The sequence <P1, *P0,* P2> satisfies the safety condition. Process P1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process *Po* can get all its tape drives and return them (the system will then have ten available tape

drives); and finally process *P2* can get all its tape drives and return them (the system will then have all twelve tape drives available).

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.**

Deadlock avoidance algorithms

1)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 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 *Pi ->Rj* indicates that process *Pi* may request resource *Rj* 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 *Pi* requests resource *Rj,* the claim edge *Pi->Rj* is converted to a request edge. Similarly, when a resource *Rj* is released by *Pi,* the assignment edge *Rj-> Pi* is reconverted to a claim edge *Pi-> Rj.* Before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph.

Now suppose that process *Pi* requests resource *Rj.* **The request can be granted only if converting the request edge *Pi-> Rj* to an assignment edge *Rj->Pi* does not result in the formation of a cycle in the resource-allocation graph.** We check for safety by using a cycle-detection algorithm.

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;* will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource-allocation graph of Figure 3.21. Suppose that *P2* requests R2 . Although R2 is currently free, we cannot allocate it to *P2,* since this action will create a cycle in the graph (Figure 3.22). A cycle, as mentioned, indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.

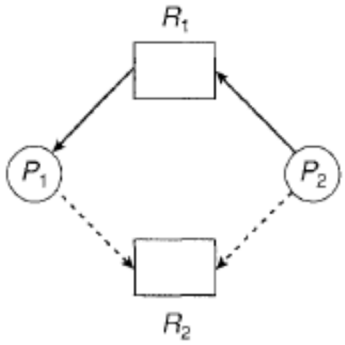


Fig:3.21 Resource-allocation graph for deadlock avoidance.

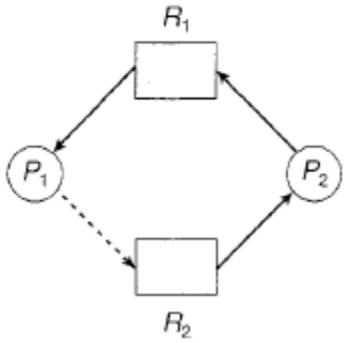


Fig:3.22 An unsafe state in a resource-allocation graph.

2)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.***

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 *Available[j]* equals *k,* then *k* instances of resource type *Rj* are available.
- **Max**. An *n* x *m* matrix defines the maximum demand of each process. If *Max[i] [j]* equals *k,* then process *Pi* may request at most *k* instances of resource type Rj.

- **Allocation**. An *n* x *m* matrix defines the number of resources of each type currently allocated to each process. If *Allocation[i][j]* equals k, then process *Pi* is currently allocated k instances of resource type *Rj.*
- **Need**. An *n* x *m* matrix indicates the remaining resource need of each process. If *Need[i][j]* equals *k,* then process *i;* may need *k* more instances of resource type *Rj* to complete its task. Note that

Need[i][j]* equals *Max[i][j] - Allocation [i][j]

2.1)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 $Finish[i] = false$ for $i = 0, 1, \ldots, n-1$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Need_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.
4. If $Finish[i] == true$ for all i, then the system is in a safe state.

This algorithm may require an order of m x n2 operations to determine whether a state is safe.

2.2) Resource-Request Algorithm

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

Let Request; be the request vector for process Pi. If Requesti [j] == k, then process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, 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:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

If the resulting resource-allocation state is safe, the transaction is completed, and process Pi is allocated its resources. However, if the new state is unsafe, then Pi must wait for Request;, and the old resource-allocation state is restored.

Example

To illustrate the use of the banker's algorithm, consider a system with five processes Po through P4 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 T0 , the following snapshot of the system has been taken:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
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:

	Need
	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 **< P1, P3, P4, P2, Po>** satisfies the safety criteria.

Suppose now that process P1 requests one additional instance of resource type *A* and two instances of resource type C, so *Request1* = (1,0,2). To decide whether this request can be immediately granted, we first check that *Request1* <=*Available*-that is, that (1,0,2) <=(3,3,2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence **<P1, P3, P4, Po, P2>** satisfies the safety requirement. Hence, we can immediately grant the request of process P1.

3) 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

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.

3.1)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 *Pi* to *Pj* in a wait-for graph implies that process *Pi* is waiting for process *Pj* to release a resource that *Pi* needs. An edge *Pi* → *Pj* exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges *Pi* → *Rq* and *Rq* → *Pj* for some resource *Rq.* For example, in Figure 3.23, 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 *n2* operations, where *n* is the number of vertices in the graph.

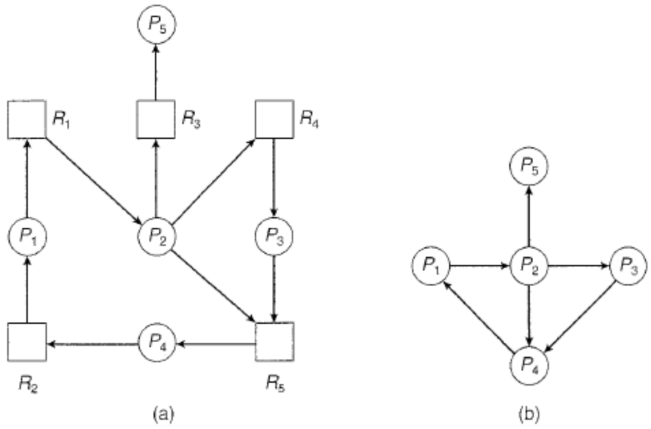


Fig:3.23 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

3.2)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 *nz* indicates the number of available resources of each type.
- **Allocation**. Ann x *nz* matrix defines the number of resources of each type currently allocated to each process.
- **Request.** An *n* x *m* matrix indicates the current request of each process. If *Request[i][j]* equals *k,* then process *Pi* is requesting *k* more instances of resource type *Rj.*

The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed.

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$. For $i = 0, 1, ..., n\text{-}1$, if $Allocation_i \neq 0$, then $Finish[i] = false$; otherwise, $Finish[i] = true$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Request_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.
4. If $Finish[i] == false$ for some i, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then process P_i is deadlocked.

This algorithm requires an order o£ *m* x *n2* operations to detect whether the system is in a deadlocked state.

To illustrate this algorithm, we consider a system with five processes *Po* through *P4* 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 *T0,* we have the following resource-allocation state:

	Allocation	*Request*	*Available*
	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 ***<Po, P2, P3, P1, P4>***results in *Finish[i]* == *true* for all *i.*

Suppose now that process *P2* makes one additional request for an instance of type C. The *Request* matrix is modified as follows:

	Request
	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 *Po,* the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a **deadlock exists, consisting of processes P1, *P2, P3,* and *P4.***

Recovery form deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available. 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.**

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.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. 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?

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:

- **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 during its execution.
- **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.
- **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 that must be dealt with in any practical system. 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.**