Unit III – Process Coordination

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 Critical Section Problem

Consider a system consisting of *n* processes {Po, P1 , … , P_{11_1}}. Each process has a segment of code, called a *critical section*, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

There 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 below:

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

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.

Two general approaches are used to handle critical sections in operating systems: (1) **preemptive kernels** and (2) **nonpreemptive kernels.** A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive 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 nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time. It 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.

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. Furthermore, 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.

Peterson's Solution

Next, illustrates 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, it presents 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,* it uses *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, it is now ready to describe the algorithm shown in the following figure. 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 same time. Only one of these assignments will last; the other will occur but will be overwritten immediately.

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

                critical section

        flag [i] = FALSE;

                remainder section

} while (TRUE);
```

The structure of process P_j in Peterson's solution.

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.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

To prove property 1, It notes 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, 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, 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 *Pi* 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 P_j 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 *Pi* will enter the critical section. However, once *Pi* exits its critical section, it will reset flag [j] to false, allowing *P;* 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).

```
do {
        acquire lock

                critical section

        release lock

                remainder section

} while (TRUE);
```

Synchronization Hardware

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.

There several more solutions to the critical-section problem using techniques ranging from hardware to software based API's available to application programmers. All these solutions are based on the premise of locking. It starts by presenting some simple hardware instructions. Hardware features can make any programming task easier and improve system efficiency.

The critical-section problem could be solved simply in a uniprocessor environment if it could prevent interrupts from occurring while a shared variable was being modified. The current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by nonpreemptive kernels.

Many modern computer systems therefore provide special hardware instructions that allow either to test or modify the content of a word or to swap the contents of two words automatically – that is, as one uninterruptible unit. One specific instruction for one specific machine, it abstract the main concepts behind these types of instructions by describing the TestAndSet () and Swap() instructions. The TestAndSet () instruction can be defined as follows:

```
boolean TestAndSet(boolean *target) {
boolean rv = *target;
```

```
*target = TRUE;
return rv;
}
```

The important characteristic of this instruction is that it is executed atomically. Thus, if two TestAndSet () instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet () instruction, then it can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process Pi is shown below:

```
do {
while (TestAndSet(&lock))
;// do nothing
// critical section
lock = FALSE;
// remainder section
} while (TRUE);
```

The Swap() instruction, in contrast to the TestAndSet () instruction, operates on the contents of two words; it is defined as shown below.

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

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 Pi is shown below:

```
do {
key = TRUE;
while (key = = TRUE)
Swap(&lock, &key);
 // critical section
lock = FALSE;
// remainder section
} while (TRUE);
```

Semaphores

Semaphore can use as a synchronization tool for application programmers. A semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a current system such as multitasking OS.

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 ("to test"); signal () was originally called V ("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 ($S \leq 0$) 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.

It can use binary semaphores to deal with the critical-section problem for multiple processes. Then processes share a semaphore, mutex, initialized to 1. Each process Pi is organized as shown below:

```
do {
        wait (mutex) ;
                // critical section
        signal(mutex);
                // remainder section
} while (TRUE);
```

(Mutual-exclusion implementation with 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.

It can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose it requires that S2 be executed only after S1 has completed. It 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 *busy waiting*. 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 necessary 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.

To overcome the need for busy waiting, it 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 fron< 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.

Deadlock 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, it consider a system consisting of two processes, Po and P1, each accessing two semaphores, S and Q, set to the value 1:

Po	***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 cannot be executed, *Po* and P1 are deadlocked.

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 are mainly concerned here are *resource acquisition and release*. Another problem related to deadlocks is *indefinite blocking* or a *starvation* 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, there are 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 how long process H must wait for L to relinquish resource R.

This problem is known as *priority inversion*. It occurs only in systems with more than two priorities, so one solution is to have only two priorities.

Classic Problem of synchronization

Here specifies 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.

The Bounded-Buffer Problem:

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 below

```
do {
// produce an item in nextp
        wait(empty);
        wait(mutex);
// add nextp to buffer
        signal(mutex);
        signal(full);
```

```
} while (TRUE);
```

The code for the consumer process is shown below:

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

Note the symmetry between the producer and the consumer. It can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

The Readers-Writers Problem:

While accessing a database by number of users simultaneously, it has to share to read or write. In this situation chaos may occur. To ensure that these difficulties do not arise, it requires 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. the reader processes share the following data structures:

```
semaphore mutex, wrt;
int readcount;
```

The semaphores mutex and wrt are initialized to 1; readcount 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 *readcount* is updated. The readcount 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 below:

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

The code for a reader process is shown below:

```
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);
```

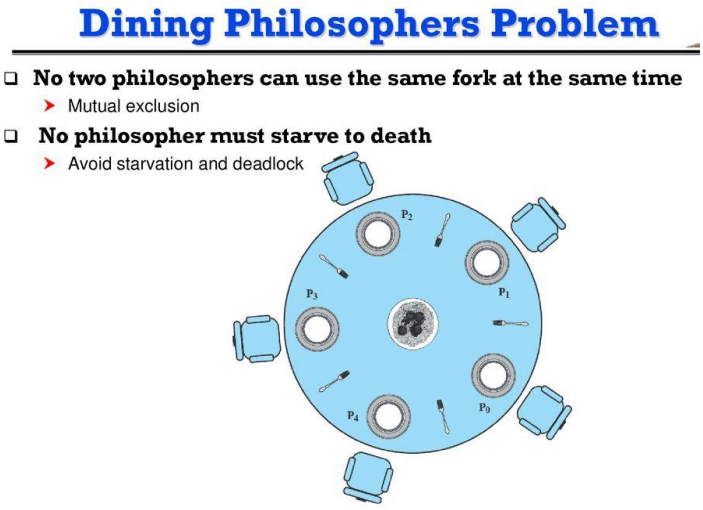
Reader-writer locks are most useful in the following situations:

- 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.

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 centre of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, he/she does not interact with his/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 his/her and his/ her left and right neighbours). A philosopher may pick up only one chopstick at a time. Obviously, he/she cannot pick up a chopstick that is already in the hand of a neighbour. When a hungry philosopher has both her chopsticks at the same time, he/she eats without releasing her chopsticks. When he/she is finished eating, he/she puts down both of his/her 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 await () operation on that semaphore; he/she releases his/ 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 below.

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

Although this solution guarantees that no two neighbours 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 his/her right chopstick, he/she will be delayed forever.

Several possible remedies to the deadlock problem are listed here.

- 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.

Monitors

Various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem. Similar problems may arise in the other synchronization models. To deal with such errors, researchers have developed high-level language constructs. One of the fundamental high-level synchronization construct is the monitor type.

- **Usage:**

An *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 below:

```
monitor monitor name
{
        // shared variable declarations
procedure P1 ( . . . ) {
```

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

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

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.

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 (in figure below).

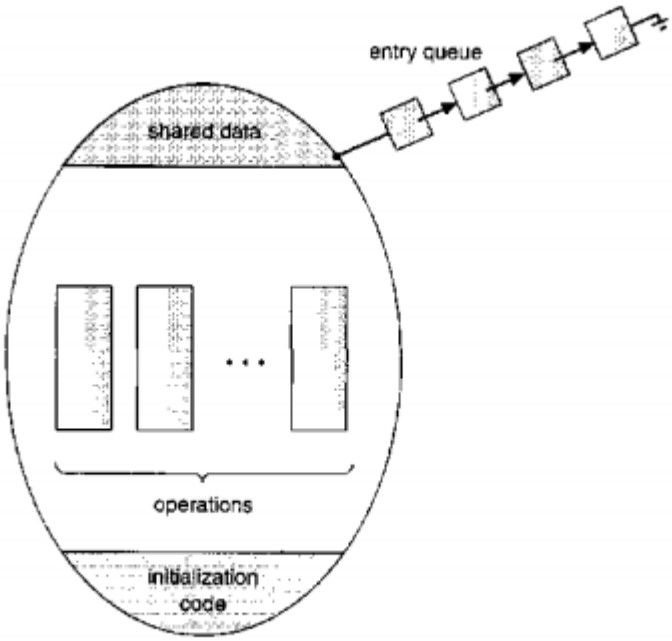


Figure 6.17 Schematic view of a monitor.

However, the monitor construct, as defined so far, is not sufficiently powerful for modelling some synchronization schemes. For this purpose, it needs 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();

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. Contrast this operation with the *signal*() operation associated with semaphores, which always affects the state of the semaphore.

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 signalling 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:

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

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

• Dining-Philosophers Solution Using Monitors

This solution imposes the restriction that a philosopher may pick up his/her chopsticks only if both of them are available. To code this solution, it need to distinguish among three states in which it 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 his/her two neighbours are not eating:
(state [(i+4) % 5] ! = EATING) and (state [(i +1) % 5] '= EATING).

It also needs to declare

condition self[5];

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

It is now in a position to describe the solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor DiningPhilosophers, whose definition is shown below:

```
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;
} }
```

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 putdown() operation. Thus, philosopher *i* must invoke the operations pickup() and putdown() in the following sequence:

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

This solution ensures that no two neighbours are eating simultaneously and that no deadlocks will occur.

- ## Implementing a Monitor Using Semaphores

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 monitor and must execute **signal** (mutex) after leaving the monitor.

Since a signalling process must wait until the resumed process either leaves or waits, an additional sernaphore, next, is introduced, initialized to 0. The signalling processes can use next to suspend themselves. An integer variable *next_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.

Now describes 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 be implemented 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--;
}
```

- **Resuming Processes within a Monitor**

Turn now to the subject of process-resumption order 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 construct can be used; it has the 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 s that is suspended. When x.signa () is executed, the process with the smallest priority number is resumed next. The monitor allocates the resource to the process that has the shortest time-allocation request. 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*.

The monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).

Deadlocks

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.

System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. Memory pace, CPU cycles, files, and I/0 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. 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.

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

- **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.

- **Use**. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

- **Release**. The process releases the resource.

Deadlock Characterization

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

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 { P0 , Pl, ... , P11 } of waiting processes must exist such that Po is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by Po.

Resource-Allocation Graph

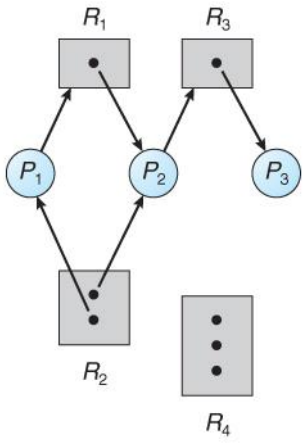
Deadlocks can be described more precisely in terms of a directed graph called a *system resource-allocation graph*. This graph consists of a set of vertices *V* and a set of edges *E*. The set of vertices Vis partitioned into two different types of nodes: *P* = { P1, P2, ... , Pn}, the set consisting of all the active processes in the system, and *R* = {R1, R2, ... , *RmL* the set consisting of all resource types in the system.

A directed edge from process g to resource type Rj is denoted by P → 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, it represents each process Pi as a circle and each resource type Rj as a rectangle. Since resource type Rj may have more than one instance, it represents 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 Pi requests an instance of resource type Rj, 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 the above figure depicts the following situation.

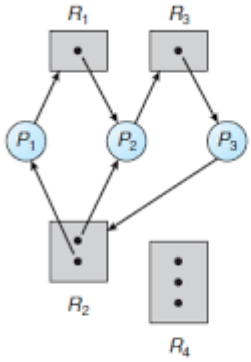
- The sets P, R and E:
  - P = {P1, P2, P3}
  - R= {R1, R2, R3, R4}
  - E = {Pl→ Rl, P2→ R3, F Rl → P2, R2 → P2, R2→ Pl, R3 → P3}
- Resource instances:
  - One instance of resource type R1
  - Two instances of resource type R2
  - One instance of resource type R3
  - Three instances of resource type 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 .

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 the above figure. 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 following graph.



Resource-allocation graph with a deadlock.

At this point, two minimal cycles exist in the system:

P1 → R 1 → P2 → R3 → P3 → R2 → P1
P2 → R3 → P3 → R2 →P2

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 following resource-allocation graph without deadlock.

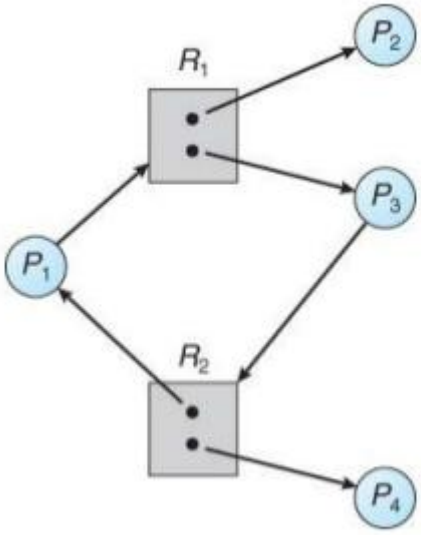


Figure 7.4 - Resource allocation graph with a cycle but no deadlock

In this case also have a cycle: P1 → R1 → P3 → R2 → P1

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.

Methods for Handling Deadlocks

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

- It can use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlocked state.
- It can allow the system to enter a deadlocked state, detect it, and recover.
- It 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 UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.

Deadlock Prevention

• Mutual Exclusion

The mutual-exclusion condition must hold for non-sharable 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 non-sharable.

• Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, it 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. It 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, 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.

• 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, it 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, it first check whether they are available. If they are, it allocate them. If they are not, it checks whether they are allocated to some other process that is waiting for additional resources. If so, it preempts the desired resources from the waiting process and allocates 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 pre-empted while it was waiting.

- **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, let R = { R1, R2, ... , Rm} be the set of resource types. It assigns 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, It 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

It 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, Rj. 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, it 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. It 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 { P0 , P1, ... , Pn}, where Pi is waiting for a resource Ri, which is held by process Pi+l· (Modulo arithmetic is used on the indexes, so that Pn is waiting for a resource Rn held by P0 .) Then, since process Pi+l is holding resource Ri while requesting resource Ri+l it must have F(Ri) < F(Ri+1) for all i. But this condition means that F(Ro) < F(R1) < ... < F(Rn) < F(Ro). By transitivity, F(Ro) < F(Ro), which is impossible. Therefore, there can be no circular wait.

Deadlock Avoidance

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.

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.

- **Resource-Allocation-Graph Algorithm**

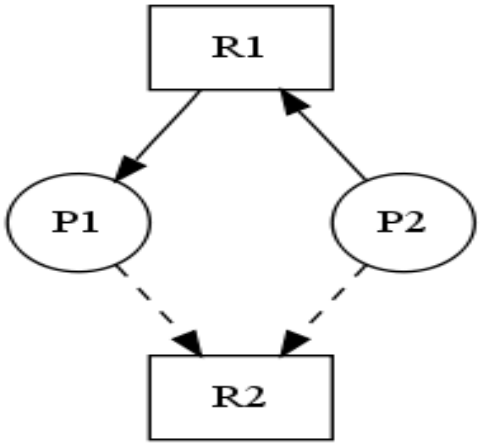
If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph for deadlock avoidance. In addition to the request and assignment edges it introduces a new type of edge, called a *claim edge*. A claim edge Pi → Rj indicates that 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 R1, the claim edge Pi → R1 is converted to a request edge. Similarly, when a resource R1 is released by Pi, the assignment edge Rj → Pi is reconverted to a claim edge Pi →Rj.

It notes that the resources must be claimed a priori in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph. It can relax this condition by allowing a claim edge Pi → R1 to be added to the graph only if all the edges associated with process Pi are claim edges.

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. An algorithm for detecting a cycle in this graph requires an order of n2 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 Pi will have to wait for its requests to be satisfied.

To illustrate this algorithm, consider the resource-allocation graph given below. Suppose that P2 requests R2. Although R2 is currently free, it cannot allocate it to P2, since this action will create a cycle in the graph 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.



- **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 describes 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 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. It needs 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 Pi 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].

Safety Algorithm

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, ... , *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;*
   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 n^2 operations to determine whether a state is safe.

Resource-Request Algorithm

This is the algorithm for determining whether requests can be safely granted.

Let $Request_i$ be the request vector for process Pi. If $Request_i$ [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 $Request_i \leq Available_i$, go to step 3. Otherwise, Pi must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

Available= Available- Request;;
Allocation; =Allocation; +Request;;
Need; =Need;- Request;;

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 $Request_i$, and the old resource-allocation state is restored.

An Illustrative 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 T_0 , 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
	ABC
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	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 $Request_1$ = (1,0,2). To decide whether this request can be immediately granted, first check that *Request1* ≤ *Available* - that is, that (1,0,2) ≤ (3,3,2), which is true. Then pretend that this request has been fulfilled, and it arrive at the following new state:

	Allocation	*Need*	*Available*
	ABC	*ABC*	*ABC*
Po	010	743	230
Pl	302	020	
P2	302	600	
P3	211	0 11	
P4	002	431	

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

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:

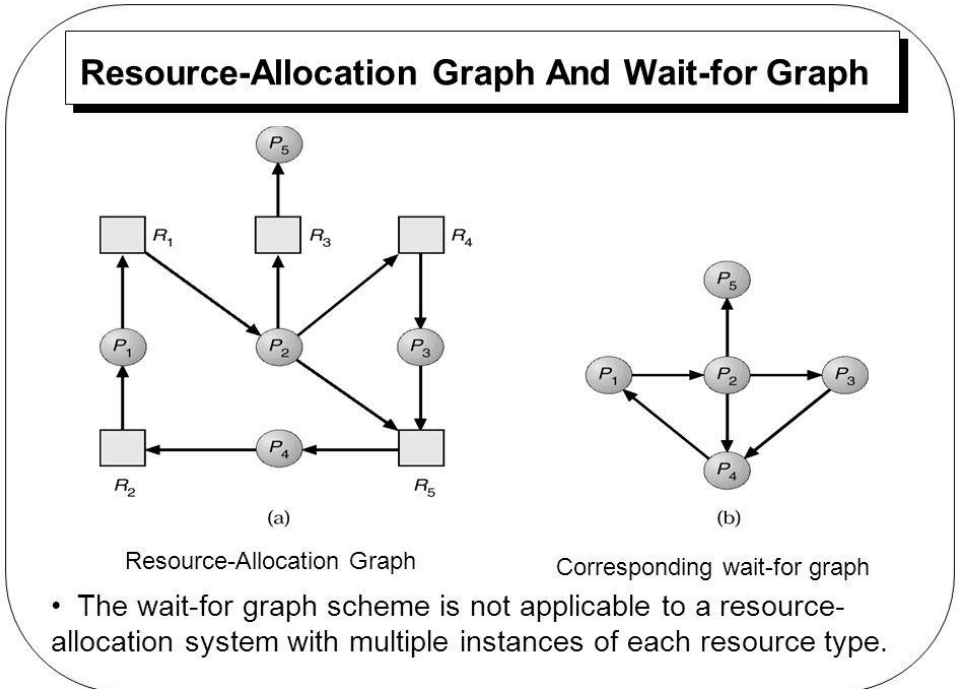
- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock

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.

- **Single Instance of Each Resource Type**

If all resources have only a single instance, then it can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. It obtains 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. In the following figure, it presents a resource-allocation graph and the corresponding wait-for graph.



- **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. It now turns 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.

To illustrate this algorithm, it 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 *To*, has the following resource-allocation state:

	Allocation	*Request*	*Available*
	A B C	A B C	A B C
Po	0 1 0	0 0 0	0 0 0
Pl	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

/'It claims that the system is not in a deadlocked state. Indeed, if it execute the algorithm, it will find that the sequence <Po, P2, P3, Plt P4> 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:

	Request
	A B C
Po	0 0 0
Pl	2 0 2
P2	0 0 1
P3	1 0 0
P4	0 0 2

It claims that the system is now deadlocked. Although it 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.

- **Detection-Algorithm Usage**

While invoking the detection algorithm it 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, it 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. 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.

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.

- ## Process Termination

To eliminate deadlocks by aborting a process, it uses 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 it 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; it should abort those processes whose termination will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one.

- ## Resource Preemption

To eliminate deadlocks using resource preemption, it successively pre-empt 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, it 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.

2. **Rollback**. If it 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. It 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?