UNIT-IV

1.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.

Concurrent access to shared data may result in data inconsistency. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes. A Shared-memory solution to bounded-butter problem allows at most $n - 1$ items in buffer at the same time. A solution, where all N buffers are used is not simple. Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer.

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

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

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

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

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently.

Note that the statement "counter++" may be implemented in machine language as follows:

*register*1 = counter

*register*1 = *register*1 + 1

counter = *register*1

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

*register*2 = counter

*register*2 = *register*2 − 1

counter = *register*2

If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved. Interleaving depends upon how the producer and consumer processes are scheduled.

Assume **counter** is initially 5. One interleaving of statements is:

producer: **register1 = counter** (*register1 = 5*)
producer: **register1 = register1 + 1** (*register1 = 6*)
consumer: **register2 = counter** (*register2 = 5*)
consumer: **register2 = register2 – 1** (*register2 = 4*)
producer: **counter = register1** (*counter = 6*)
consumer: **counter = register2** (*counter = 4*)

The value of **count** may be either 4 or 6, where the correct result should be 5.

Race condition: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

To prevent race conditions, concurrent processes must be **synchronized.**

2. The Critical-Section Problem

Consider a system consisting of *n* processes {*P*0, *P*1, . . . , *Pn*−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 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 [illegible]. The critical section may be followed by an *exit section*. The remaining code is the [illegible].

The general structure of a typical process *Pi* is

```
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: **preemptive kernels** and **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.

3. Synchronization Hardware

Generally any solution to the critical section problem requires lock.

```
do {
        acquire lock
              critical section
        release lock
              reminder section
} while (1);
```

Race conditions are prevented.

The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. Unfortunately, this solution is not as feasible in a multi processor environment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases

Two types of instructions are 1.test and set() 2. compare and swap()

The definition of the test and set() instruction

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Mutual-exclusion implementation with test and set().

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

The definition of the compare and swap() instruction.

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Mutual-exclusion implementation with the compare and swap() instruction.

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

4.Semaphores

A 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 , signal() was originally called V .

The definition of wait() is as follows:

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

The definition of signal() is as follows:

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

Two Types of Semaphores

1. Counting semaphore – integer value can range over an unrestricted domain.
2. Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement.

   Counting semaphores 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.

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 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.

In process $P1$, we insert the statements

```
S1;
signal(synch);
```

In process $P2$, we insert the statements

```
wait(synch);
S2;
```

Because synch is initialized to 0. *P*2 will execute *S*2 only after *P*1 has invoked signal(synch). Wh
is after statement *S*1 has been executed.

we define a semaphore as follows:

```
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.

Now, the wait() semaphore operation can be defined as

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

signal() semaphore operation can 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.

Critical Section of n Processes

Shared data:

semaphore mutex; //initially mutex = 1

Process Pi:

```
do {
    wait(mutex);
     critical section

    signal(mutex);
    remainder section
} while (1);
```

Deadlocks and Starvation

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

Let S and Q be two semaphores initialized to 1

```
        P0                P1

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

Suppose that *P*0 executes wait(S) and then *P*1 executes wait(Q). When *P*0 executes wait(Q), it must wait until *P*1 executes signal(Q). Similarly, when *P*1 executes wait(S), it must wait until *P*0 executes signal(S). Since these signal() operations cannot be executed, *P*0 and *P*1 are deadlocked.

Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

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.

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—processM—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. 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 questio. When they are finished, their priorities revert to their original values.

In the example 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 H and assume its original priority. Because resource R would now be available, process H—not M—would run next.

5.Classic Problems of Synchronization

1.The Bounded-Buffer Problem

The producer and consumer-processes share the following data structures:

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

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

The structure of the producer process

```
do {
   . . .
   /* produce an item in next_produced */
   . . .
   wait(empty);
   wait(mutex);
   . . .
   /* add next_produced to the buffer */
   . . .
   signal(mutex);
   signal(full);
} while (true);
```

The structure of the consumer process.

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

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.

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.

The first readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.

The ***second*** readers– writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve.

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

Semaphore rwmutex = 1;
semaphore mutex = 1;
int read count = 0;

The semaphores mutex and rwmutex are initialized to 1; read count is initialized to 0. The semaphore rwmutex 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 rw mutex functions as a mutual exclusion semaphore for the writers.

The code for a writer process is

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

the code for a reader process is

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

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

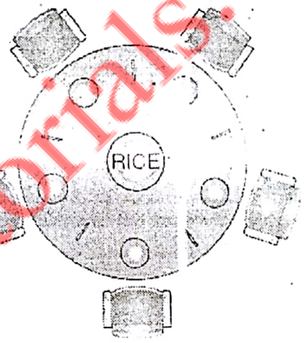


Figure :The situation of the dining philosophers.

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 neighbours). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again

The **dining-philosophers problem** is considered a classic synchronization Problem It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are semaphore chopstick[5];

The structure of philosopher *i* is ,

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} 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 at the same time 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 replaced by:

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-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

6. Monitors

Some of the problems of semaphore are

- 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.

- 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

monitor-High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes

```
monitor monitor name
{
   /*shared-variable declarations*/
   function P1 ( . . . ) {
      . . .
   }
   function P2 ( . . . ) {
      . . .
   }
         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }
   initialization_code ( . . . ) {
      . . .
   }
}
```

To allow a process to wait within the monitor, a **condition**-variable must be declared, as

condition x, y;

Condition variable can only be used with the operations **wait** and **signal**.

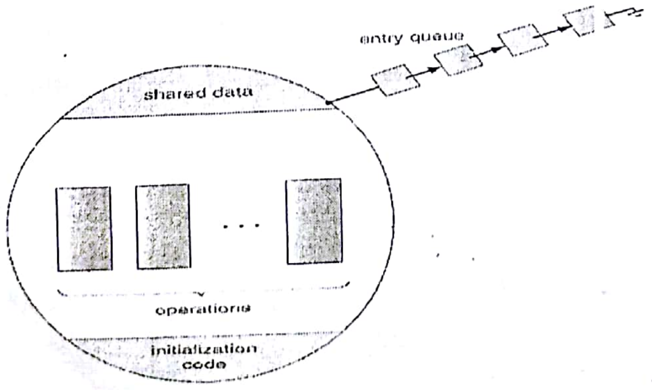
The operation **x.wait();**

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.

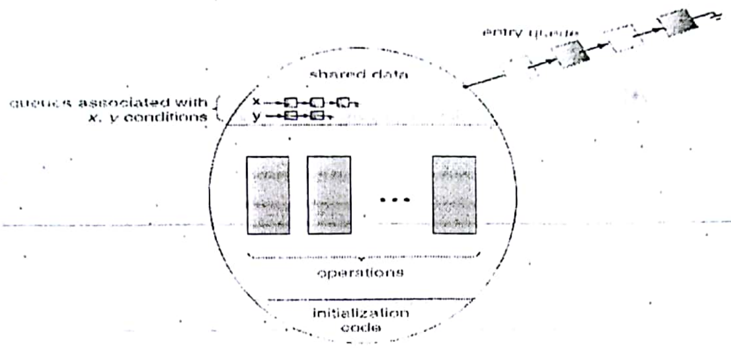
Schematic view of a monitor.



A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition*:

condition x, y;

Monitor with condition variables



Dining-Philosophers Solution Using Monitors

```
monitor dp
    {
            enum {thinking, hungry, eating} state[5];
            condition self[5];
            void pickup(int i)              // following slides
            void putdown(int i)     // following slides
            void test(int i)        // following slides
            void init() {
                    for (int i = 0; i < 5; i++)
                            state[i] = thinking;
            }
    }
  void pickup(int i) {
            state[i] = hungry;
            test[i];
            if (state[i] != eating)
                    self[i].wait();
    }
    void putdown(int i) {
            state[i] = thinking;
            // test left and right neighbors
            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();
        }
}
```

Monitor Implementation Using Semaphores

- Variables

```
        semaphore mutex;  // (initially  = 1)
        semaphore next;   // (initially  = 0)
        int next-count = 0;
```

- Each external procedure *F* will be replaced by

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

- Mutual exclusion within a monitor is ensured.
- For each condition variable *x*, we have:

```
        semaphore x-sem; // (initially  = 0)
        int x-count = 0;
```

- The operation **x.wait** can 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--;
        }
```

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**.

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

Example :

System has 2 tape drives.

P_1 and P_2 each hold one tape drive and each needs another one.

Example

semaphores A and B, initialized to 1

```
P0              P1
wait (A);       wait(B)
wait (B);       wait(A)
```

1.System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources may be partitioned into several types each consisting of some number of identical instances. CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system.

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

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

2. Deadlock Characterization

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

2.1 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.

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, P1, ..., Pn}* of waiting processes must exist such that *P0* 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 *P0*.

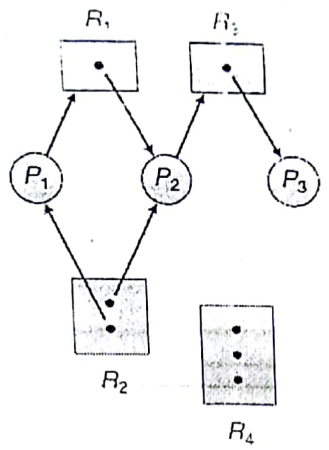
2.2 Resource-Allocation Graph

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* . 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 *Rj* may have more than one instance, we represent each such instance as a dot within the rectangle.

The resource-allocation graph example is



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.

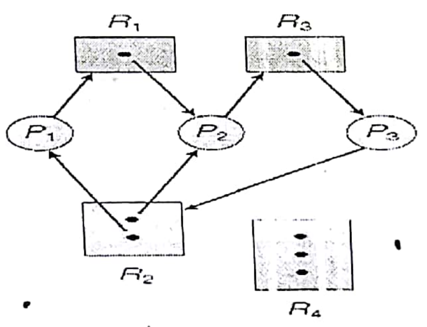
Resource-allocation graph with a deadlock

Suppose that process $P3$ requests an instance of resource type $R2$. Since no resource instance is currently available, we add a request edge $P3 \rightarrow R2$ to the graph . At this point, two minimal cycles exist in the system:

$P1 \rightarrow R1 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P1$

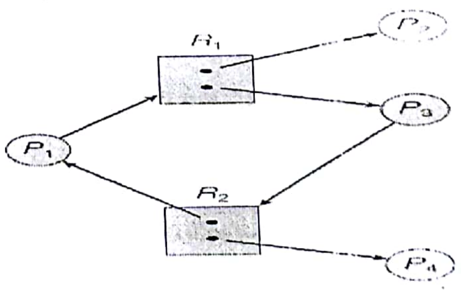
$P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P2$

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 $R1$.



Resource-allocation graph with a cycle but no deadlock

$P1 \rightarrow R1 \rightarrow P3 \rightarrow R2 \rightarrow P1$



3.Methods for Handling Deadlocks

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.

4.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

1.Mutual Exclusion

The mutual exclusion condition must hold. That is, at least one resource must be **Non-sharable.** Sharable resources. in contrast. do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read only file at the same time, they can be granted simultaneous access to the file. however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

2.Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

One protocol that we can use requires each process to request and be allocated all its resources before it begins execution.

An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period.

Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

3.No Preemption

The third necessary condition for deadlocks is that there be no pre-emption 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.

Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are neither available nor held by a waiting process, the requesting process must wait.

4.Circular Wait

Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

To illustrate, we let consider *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.

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*).

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

5. Deadlock Avoidance

Require additional information about how resources are to be requested. Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

5.1 Safe State

When a process requests an available resource. system must decide if immediate allocation leaves the system in a *safe state*.

A System is in safe state if there exists a safe sequence of all processes.

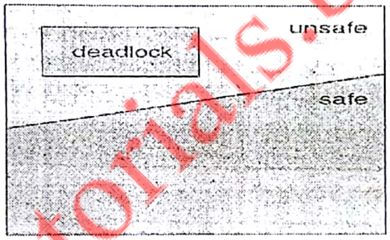
Sequence $<P_1, P_2, \ldots, P_n>$ is safe if for each P_i, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the P_j, with $j<I$.

If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.

When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.

When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

- If a system is in safe state $\Rightarrow$ no deadlocks.
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.



Example:

We consider a system with twelve magnetic tape drives and three processes: $P0$, $P1$, and $P2$. Process $P0$ 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 $t0$, process $P0$ 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**
$P0$	10	5
$P1$	4	2
$P2$	9	2

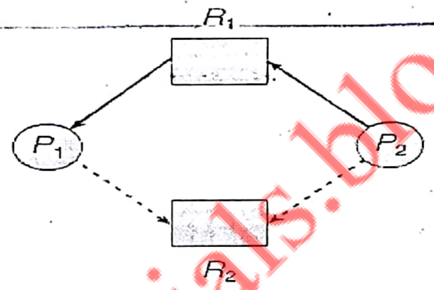
At time $t0$, the system is in a safe state. The sequence $<P1, P0, P2>$ satisfies the safety condition.

5.2 Resource-Allocation-Graph Algorithm

A claim edge $Pi \rightarrow Rj$ 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 $Pi \rightarrow Rj$ is converted to a request edge. Similarly, when a resource Rj is released by Pi, the assignment edge $Rj \rightarrow Pi$ is reconverted to a claim edge $Pi \rightarrow Rj$.

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.

Resource-allocation graph for deadlock avoidance



Now suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge $Pi \rightarrow Rj$ to an assignment edge $Rj \rightarrow Pi$ does not result in the formation of a cycle in the resource-allocation graph.

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.

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. If $P1$ requests $R2$, and $P2$ requests $R1$, then a deadlock will occur.

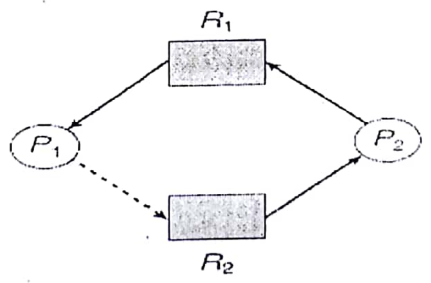


Fig: An unsafe state in a resource-allocation graph.

5.3 Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

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 $n \times m$ 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 $n \times m$ 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 $n \times m$ 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

For finding out whether or not a system is in a safe state.

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

Resource-Request Algorithm

The algorithm for determining whether requests can be safely granted.

Let *Requesti* 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 $\boldsymbol{Request_i} \le Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If $\boldsymbol{Request_i} \le \boldsymbol{Available}$, 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 P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for $\boldsymbol{Request_i}$, and the old resource-allocation state is restored.

6. Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, th 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

6.1 Single Instance of Each Resource Type

Maintain *wait-for* graph. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges. 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 $Pi \rightarrow Pj$ exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges $Pi \rightarrow Rq$ and $Rq \rightarrow Pj$ for some resource Rq.

Periodically invoke an algorithm that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

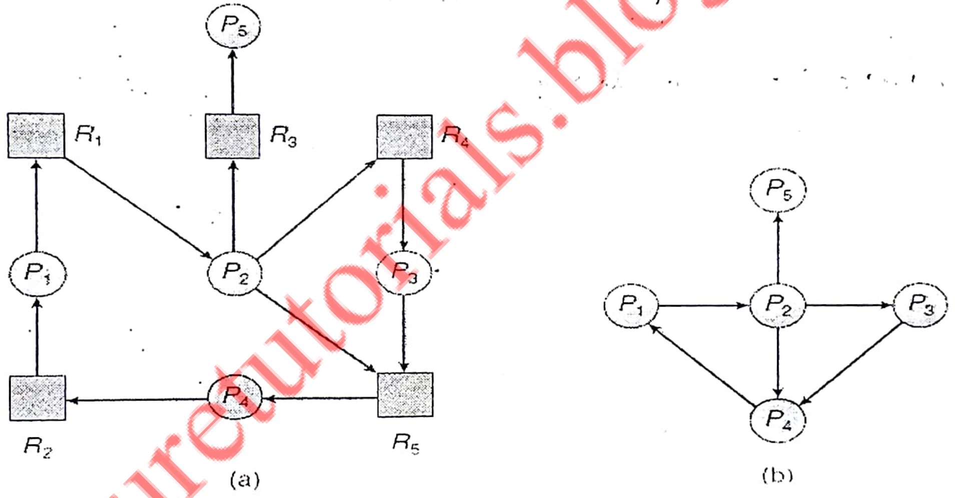


Figure : (a) Resource-allocation graph. (b) Corresponding wait-for graph.

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. The algorithm employs several time-varying data structures that are

Available. A vector of length m indicates the number of available resources of each type.

Allocation. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.

Request. An $n \times m$ matrix indicates the current request of each process. If ***Request***$[i][j]$ equals k, then process Pi is requesting k more instances of resource type Rj.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize

   Work=*Available*. For *i*=0, 1, ..., *n*–*1*, if *Allocationi* _= 0, then

   Finish[*i*]=*false*. Otherwise, *Finish*[*i*]=*true*.

2. Find an index *i* such that both

   a. ***Finish***[*i*] == ***false***

   b. ***Requesti*** ≤ ***Work***

   If no such *i* exists, go to step 4.

3. *Work*=*Work*+*Allocationi*

   Finish[*i*]=***true***

   Go to step 2.

4. If *Finish*[*i*] ==*false* for some *i*, 0≤*i*<*n*, then the system is in a deadlocked state. Moreover, if *Finish*[*i*] == *false*, then process *Pi* is deadlocked.

Example:

We consider a system with five processes *P*0 through *P*4 and three resource types *A*, *B*, and *C*. Resource type *A* has seven instances, resource type *B* has two instances, and resource type *C* has six instances. Suppose that, at time *T*0, we have the following resource-allocation state:

	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 <*P*0, *P*2, *P*3, *P*1, *P*4> 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
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 claim that the system is now deadlocked. Although we can reclaim the resources held by process *P*0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes *P*1, *P*2, *P*3, and *P*4.

Detection-Algorithm Usage

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

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

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

7. Recovery from Deadlock

7.1 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.
- Abort one process at a time until the deadlock cycle is eliminated.

Many factors may affect which process is chosen, including:

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

7.2 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** – minimize cost.
- **Rollback** – return to some safe state, restart process for that state.
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor