MODULE 3

Chapter 6: Synchronization

Chapter 7: Deadlocks

Text Book:

1) Abraham Silberschatz, Peter Baer Galvin, Greg Gagne: **Operating Systems Principles**, 8th Edition, Wiley – India.

Chapter 6: PROCESS 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. To maintain data consistency, various mechanisms is required to ensure the orderly execution of cooperating processes that share a logical address space.

Producer- Consumer Problem

- A Producer process produces information that is consumed by consumer process.
- To allow producer and consumer process to run concurrently, A Bounded Buffer can be used where the items are filled in a buffer by the producer and emptied by the consumer.
- The original solution allowed at most **BUFFER_SIZE - 1** item in the buffer at the same time. To overcome this deficiency, an integer variable ***counter***, initialized to 0 isadded.
- ***counter*** is incremented every time when a new item is added to the buffer and is decremented every time when one item removed from thebuffer.

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

```
while (true) {

                    /* produce an item and put in nextProduced*/ while (counter ==
           BUFFER_SIZE)
                                            ; // do nothing
                                    buffer [in] = nextProduced;
                                    in = (in + 1) % BUFFER_SIZE;
                                    counter++;
}
```

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

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

□ Race Condition

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

□ Illustration:

Suppose that the value of the variable ***counter*** is currently 5 and that the producer and consumer processes execute the statements "counter++" and "counter--" concurrently. The value of the variable counter may be 4, 5, or 6 but the only correct result is counter == 5, which is generated correctly if the producer and consumer execute separately.

The value of ***counter*** may be incorrect as shown below:

The statement counter++ could be implemented as

```
register1= counter
register1 = register1 + 1
counter =register1
```

The statement counter-- could be implemented as

```
register2 =counter
register2 =  register2 – 1
counter = register2
```

The concurrent execution of "counter++" and "counter--" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order. One such interleaving is

Consider this execution interleaving with “count = 5” initially:

S0: producer execute register1=counter	{register1 = 5}
S1: producer execute register1 = register1+1	{register1 = 6}
S2: consumer execute register2=counter	{register2 = 5}
S3: consumer execute register2 = register2-1	{register2 = 4}
S4: producer execute counter=register1	{count =6}
S5: consumer execute counter=register2	{count =4}

- ***Note:*** It is arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T4 and T5, we would arrive at the incorrect state "counter==6".
- *Definition* ***Race Condition***: A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **RaceCondition.**
- To guard against the race condition, ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, ***the processes are synchronized*** in some way.]

The Critical Section Problems

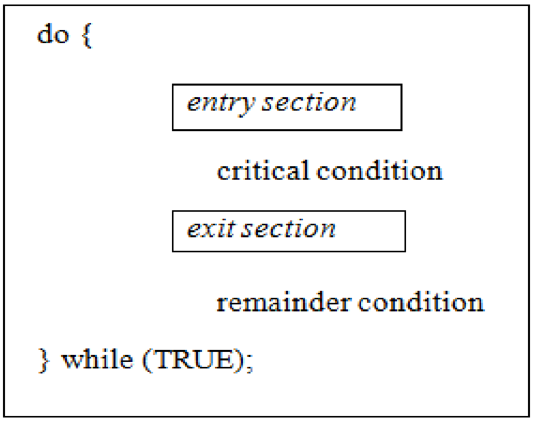
- Consider a system consisting of n processes {Po, P1 , ... ,Pn-1}.
- Each process has a segment of code, called a **critical section** in which the process may be changing common variables, updating a table, writing a file, and soon
- The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the sametime.
- The critical-section problem is to design a protocol that the processes can use to cooperate.

The general structure of a typical process Pi is shown in below figure.

- 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

reminder section.



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

Figure 24: General structure of a typical process Pi

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

1. **Mutual exclusion:**If process Pi is executing in its critical section, then no other processes can be executing in their criticalsections.

   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 postponedindefinitely.

2. **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 isgranted.

PETERSON'S SOLUTION

- This is a classic software-based solution to the critical-section problem. There are no guarantees that Peterson's solution will work correctly on modern computer architectures
- Peterson's solution 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 P_o and P_1 or Pi and P_j where j = 1-i

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

int turn; boolean flag[2];

- **turn:** The variable turn indicates whose turn it is to enter its critical section.

  Ex:

if turn == i, then process P_i is allowed to execute in its criticalsection

flag: The flag array is used to indicate if a process is ready to enter its critical section.

Ex: if flag [i] is true, this value indicates that P_i is ready to enter its critical section.

```
do {
                        flag[i] = TRUE; turn = j;
            while (flag[j] && turn == j)
            ; // do nothing critical section
                             flag[i] = FALSE;
             remainder section

             } while (TRUE);
```

Figure 25: The structure of process P_i in Peterson's solution

- To enter the critical section, process Pi first sets flag [i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can doso.
- 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 over written immediately.
- The eventual value of turn determines which of the two processes is allowed to enter its critical sectionfirst

To prove that solution is correct, then we need to show that

1. Mutual exclusion ispreserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

1. To prove Mutual exclusion

- Each pi enters its critical section only if either flag [j] == false or turn ==i.
- 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 Pi and Pj 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 (Pj) must have successfully executed the while statement, whereas Pi had to execute at least one additional statement ("turn==j").
- However, at that time, flag [j] == true and turn == j, and this condition will persist as long as Pi is in its critical section, as a result, mutual exclusion ispreserved.

2. <u>To prove Progress and Bounded-waiting</u>

- 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 onepossible.
- If Pj is not ready to enter the critical section, then flag [j] ==false, and Pi can enter its criticalsection.
- If Pj has set flag [j] = true and is also executing in its while statement, then either turn === i or turn ===j.

▪ If turn == i, then Pi will enter the criticalsection.

▪ If turn== j, then Pj will enter the criticalsection.

- However, once Pj exits its critical section, it will reset flag [j] = false, allowing Pi to enter its criticalsection.
- If Pj 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 (boundedwaiting).

<u>SYNCHRONIZATION HARDWARE</u>

- The solution to the critical-section problem requires a simple tool-**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 and it releases the

lock when it exits the critical section

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

Figure 26: Solution to the critical-section problem using locks.

- The critical-section problem could be solved simply in a uniprocessor environment if interrupts are prevented from occurring while a shared variable was being modified. In this manner, 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 sharedvariable.

- But this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

TestAndSet ( ) and Swap( ) instructions

- Many modern computer systems provide special hardware instructions that allowto **test** and **modify** the content of a word or to **swap** the contents of two words **atomically**, that is, as one uninterruptibleunit.

- Special instructions such as TestAndSet () and Swap() instructions are used to solve the critical-sectionproblem

- The TestAndSet () instruction can be defined as shown in Figure. The important characteristic of this instruction is that it is executed atomically.

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

Figure 27: The definition of the TestAndSet () instruction

Definition:

Thus, if two TestAndSet () instructions are executed simultaneously, they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet () instruction, then implementation of mutual exclusion can be done by declaring a Boolean variable lock, initialized to false.

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

Figure 28: Mutual-exclusion implementation with TestAndSet ()

- The Swap() 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:
         }
```

Figure 29: The definition of the Swap ( ) instruction

Definition:

- Swap() 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 in below

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

               //       critical section
               lock =FALSE;

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

Figure 33: Mutual-exclusion implementation with the Swap() instruction

These algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement.

- Below algorithm using the TestAndSet () instruction that satisfies all the critical- section requirements. The common data structures are

boolean waiting[n]; boolean lock;

These data structures are initialized to false.

```
do {
               waiting[i] = TRUE; key = TRUE;
               while (waiting[i] && key)
               key = TestAndSet(&lock); waiting[i] = FALSE;

                      // critical section j

               = (i + 1) % n;
                      while ((j != i) && !waiting[j]) j = (j + 1) % n;
               if (j == i)
                      lock = FALSE;
               else
                             waiting[j] = FALSE;
                             // remainder section
               } while (TRUE);
```

Figure 34: Bounded-waiting mutual exclusion with TestAndSet ()

1. To prove the mutual exclusionrequirement

□ Note that process Pi can enter its critical section only if either waiting [i] == false or key ==false.

□ The value of key can become false only if the TestAndSet( ) isexecuted.

□ The first process to execute the TestAndSet( ) will find key== false; all others must wait.

□ The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.

2. To prove the progressrequirement

Note that, the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.

3. To prove the bounded-waitingrequirement

□ Note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ... , n 1, 0, ... , i 1).

□ It designates the first process in this ordering that is in the entry section (waiting[j]

==true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n - 1 turns.

SEMAPHORE

□ A semaphore is a synchronization tool is used solve various synchronization problem and can be implementedefficiently.

□ Semaphore do not require busywaiting.

□ A semaphore S is an integer variable that, is accessed only through two standard atomic operations: wait () and signal (). The wait () operation was originally termed P and signal() was calledV.

Definition of wait ():

```
wait (S) {
        while S <= 0
        ; // no-op
   S--;
```

Definition of signal ():

```
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.

Binary semaphore

- The value of a binary semaphore can range only between 0 and1.
- Binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion. Binary semaphores to deal with the critical-section problem for multiple processes. Then processes share a semaphore, mutex, initialized to1

Each process Pi is organized as shown in below figure

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

Figure 30: Mutual-exclusion implementation with semaphores

Counting semaphore

- The value of a counting semaphore can range over an unrestricted domain.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore. When a process releases a resource, it performs a signal()operation.
- 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.

Implementation

- The main disadvantage of the semaphore definition requires busywaiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes.
- Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spinlock** because the process "spins" while waiting for the lock.

Semaphore implementation with no busy waiting

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

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

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

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

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

- The wait() semaphore operation can now be defined as:

```
wait(semaphore *S) {

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

The signal () semaphore operation can now be defined as

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

- The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.
- In this implementation semaphore values may be negative. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.

Deadlocks and Starvation

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

```
P0                                        P1

wait(S);                                wait(Q);

wait(Q);                                wait(S);

.                                          .

.                                          .

signal(S);                             signal(Q);

signal(Q);                             signal(S);
```

- Suppose that Po executes wait (S) and then P1 executes wait (Q). When Po executes wait (Q), it must wait until P1 executes signal (Q). Similarly, when P1 executes wait (S), it must wait until Po executes signal(S). Since these signal() operations cam1ot be executed, Po and P1 are deadlocked.
- Another problem related to deadlocks is indefinite blocking or starvation: A situation in which processes wait indefinitely within the semaphore.
- Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

CLASSICAL PROBLEMS OF SYNCHRONIZATION

- Bounded-BufferProblem
- Readers and WritersProblem
- Dining-PhilosophersProblem

Bounded-Buffer Problem

- N buffers, each can hold one item
- Semaphore **mutex**initialized to the value 1
- Semaphore **full** initialized to the value0
- Semaphore empty initialized to the value N.

The structure of the producer process:

The structure of the consumer process:

Readers-Writers Problem

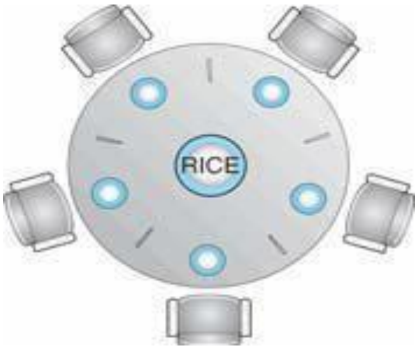
- A data set is shared among a number of concurrent processes
- Readers – only read the data set; they do **not** perform any updates
- Writers – can both read and write.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- Shared Data
- Dataset
- Semaphore **mutex** initialized to 1.
- Semaphore **wrt** initialized to1.
- Integer **readcount** initialized to 0.

The structure of a writer process

The structure of a readerprocess

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 singlechopsticks.

A philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. 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 thinkingagain.

It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-freemanner.

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

semaphore chopstick[5];

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

Several possible remedies to the deadlock problem are replaced by:

- Allow at most four philosophers to be sitting simultaneously at the table.

- Allowaphilosophertopickupherchopsticksonlyifbothchopsticksareavailable.
- 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.

Problems with Semaphores

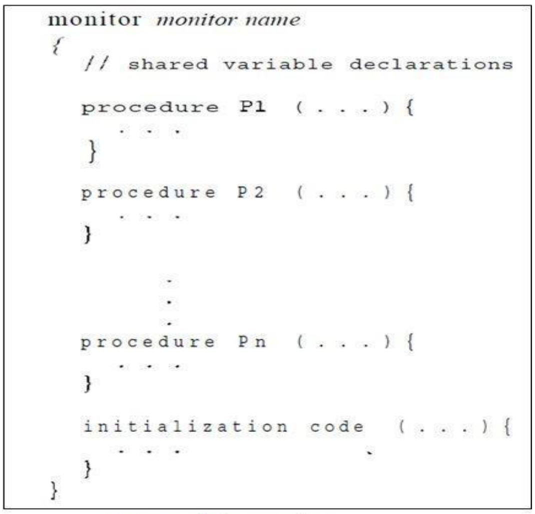
Correct use of semaphore operations:

- signal (mutex) …. wait (mutex) : Replace signal with wait andvice-versa
- wait (mutex) … wait(mutex)
- Omitting of wait (mutex) or signal (mutex) (orboth)

Monitor

- An **abstract data type**—or **ADT**—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.
- A ***monitor type***is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables.

- The monitor construct ensures that only one process at a time is active within the monitor.



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

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

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

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

Syntax of the monitor

Figure :

- To have a powerful Synchronization schemes a *condition* construct is added to the Monitor. So synchronization scheme can be defined with one or more variables of type *condition* Two operations on a conditionvariable:

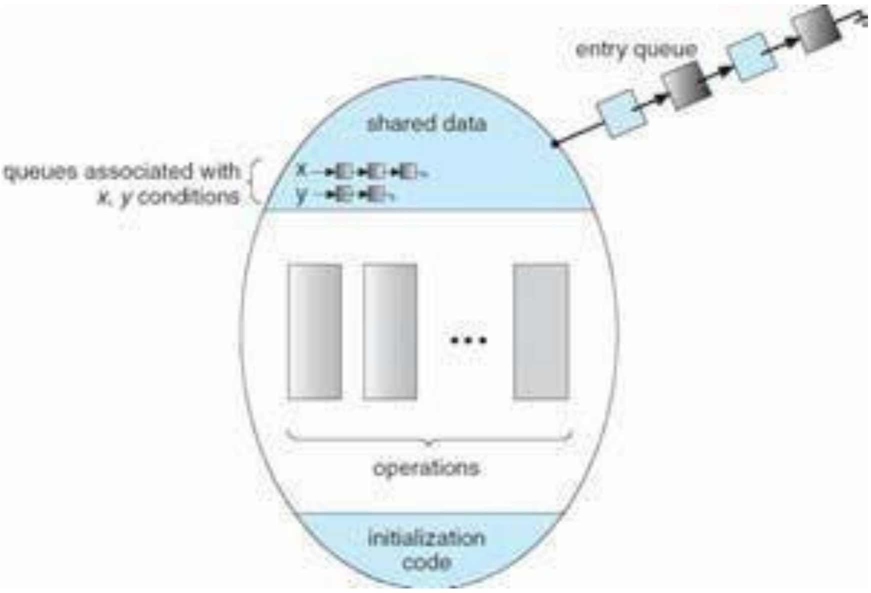
Condition x, y

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

The operation

x.wait () – a process that invokes the operation is suspended.

x.signal () – resumes one of processes (if any) that invoked x.wait ()

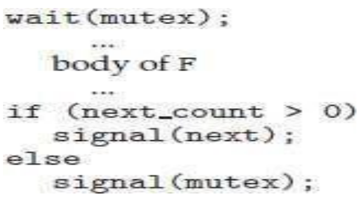


Monitor with Condition Variables

Solution to Dining Philosophers

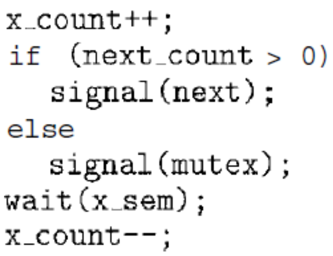
Each philosopher I invokes the operations **pickup()** and **putdown()** in the following

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



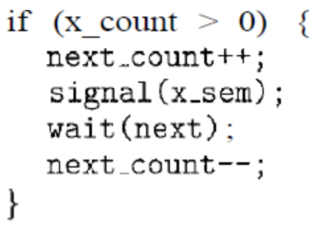
```
wait(mutex);
      ...
   body of F
      ...
if (next_count > 0)
   signal(next);
else
   signal(mutex);
```

- 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 implementedas



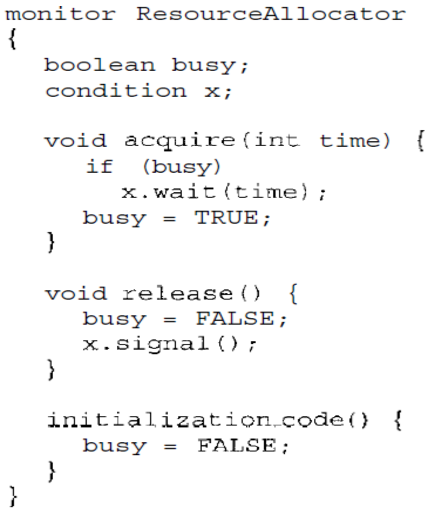
```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

Resuming Processes within a Monitor

If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then to determine which of the suspended processes should be resumed next, one simple solution is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. For this purpose, the **conditional-wait** construct can be used. This construct has theform

x.wait(c);

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



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

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

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

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

- The Resource Allocator monitor shown in the above Figure, which controls the allocation of a single resource among competingprocesses.
- 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 neverrequested.
- A process might request the same resource twice (without first releasing the resource).

Chapter 7: Deadlocks

DEADLOCKS

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 space, CPU cycles, files, and I/0 devices 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 carrying out its designated task. 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:

Request: The process requests the resource. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.

Use: The process can operate on the resource.

Release: The process releases the resource.

A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources or logical resources

To illustrate a deadlocked state, consider a system with three CD RW drives.

Suppose each of three processes holds one of these CD RW drives. If each process now requests another drive, the three processes will be in a deadlocked state.

Each is waiting for the event "CD RW is released," which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type

Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process Pi is holding the DVD and process Pj is holding the printer. If Pi requests the printer and Pj requests the DVD drive, a deadlock occurs.

DEADLOCK CHARACTERIZATION

Necessary Conditions

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

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

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

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

4. **Circular wait:** A set {P_0, P_l, ... , P_n} of waiting processes must exist such that P_o is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ... , P_{n-1} is waiting for a resource held by P_n and P_n is waiting for a resource held by P_o.

Resource-Allocation Graph

Deadlocks can be described in terms of a directed graph called **System Resource-Allocation Graph**

The graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes:

- P = {P_1, P_2, ...,P_n}, the set consisting of all the active processes in the system.
- R = {R_1, R_2, ..., R_m} the set consisting of all resource types in the system.

A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$ it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource.

A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$ it signifies that an instance of resource type R_j has been allocated to process $P_{i.}$

- A directed edge $P_i \rightarrow R_j$ is called a Request Edge.
- A directed edge $R_j \rightarrow P_i$ is called an Assignment Edge.

Pictorially each process P_i as a circle and each resource type R_j as a rectangle. Since resource type R_j may have more than one instance, each instance is represented as a dot within the rectangle.

A request edge points to only the rectangle R_j, whereas an assignment edge must also designate one of the dots in the rectangle.

When process P_i requests an instance of resource type $R_{j,}$ a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.

The resource-allocation graph shown in Figure depicts the following situation.

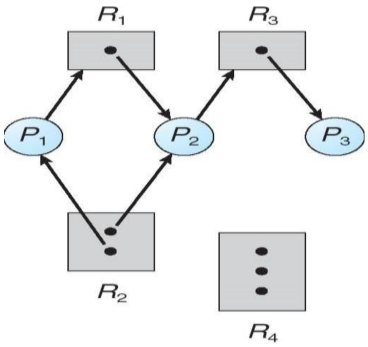


Figure 1:

The sets P, K and E:

- P = {P_1, P_2, P_3}
- R= {R_1, R_2, R_3, R_4}
- E = {$P_l \rightarrow R_l$, $P_2 \rightarrow R_3$, $R_l \rightarrow P_2$, $R_2 \rightarrow P_2$, $R_2 \rightarrow P_1$, $R_3 \rightarrow P_3$ }

- Resource instances:
- One instance of resource type R_1
- Two instances of resource type R_2
- One instance of resource type R_3

- Three instances of resource type R_4

Process states:

- Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
- Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.
- Process P_3 is holding an instance of R_3.

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.
- 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, the resource-allocation graph depicted in below 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 graph. At this point, two minimal cycles exist in the system:+

1. P1 →R1 → P2 → R3 → P3 → R2→P1
2. P2 →R3 → P3 → R2 → P2

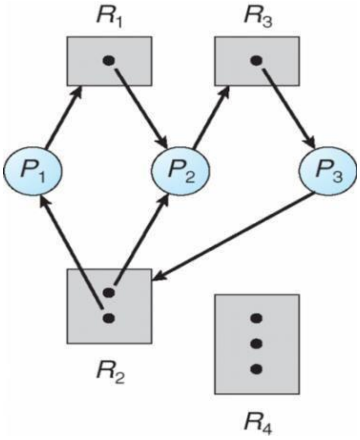


Figure 2: Resource-allocation graph with a deadlock.

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

Consider the resource-allocation graph in below Figure. In this example also have a cycle:

P1→R1→P3→R2→P1

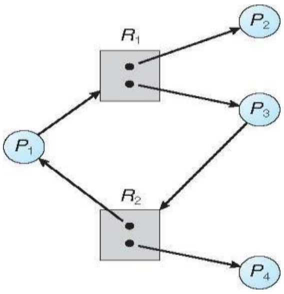


Figure 3: Resource-allocation graph with a cycle but no deadlock

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

METHODS FOR HANDLING DEADLOCKS

The deadlock problem can be handled in one of three ways:

1. Use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
2. Allow the system to enter a deadlocked state, detect it, and recover.
3. Ignore the problem altogether and pretend that deadlocks never occur in the system.

To ensure that deadlocks never occur, the system can use either deadlock prevention or a deadlock-avoidance scheme.

Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Deadlock-avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an **algorithm** that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.

In the absence of algorithms to detect and recover from deadlocks, then the system is in a deadlock state yet has no way of recognizing what has happened. In this case, the undetected deadlock will result in deterioration of the system's performance, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

DEADLOACK PREVENTION

Deadlock can be prevented by ensuring that at least one of the four necessary conditions cannot hold.

Mutual Exclusion

- The mutual-exclusion condition must hold for non-sharable resources. Sharable resources, do not require mutually exclusive access and thus cannot be involved in a deadlock.
- Ex: Read-only files are 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.
- Deadlocks cannot prevent 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, then 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.
- Another 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.

Ex:

- 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.

The two main disadvantages of these protocols:

1. Resource utilization may be low, since resources may be allocated but unused for a long period.
2. Starvation is possible.

No Preemption

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated.

To ensure that this condition does not hold, the following protocols can be used:

- If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources the process is currently holding are preempted.
- 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.

If a process requests some resources, first check whether they are available. If they are, allocate them.

If they are not available, check whether they are allocated to some other process that is waiting for additional resources. If so, preempt the desired resources from the waiting process and allocate them to the requestng process.

If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them.

A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

Circular Wait

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. Assign a unique integer number to each resource type, which allows to compare two resources and to determine whether one precedes another in ordering. Formally, it defined as a one-to-one function

F: R ->N, where N is the set of natural numbers.

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
(print

er) =
12

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 -R_i. After that, the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$.

DEADLOCK AVOIDANCE

- To avoid deadlocks an additional information is required about how resources are to be requested. With the knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock
- Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.
- The various algorithms that use this approach differ in the amount and type of information required. The simplest model requires that each process declare the *maximum number* of resources of each type that it may need. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. Such an algorithm defines the ***deadlock-avoidance approach.***

Safe State

- **Safe state:** A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. A system is in a safe state only if there exists a safe sequence.
- **Safe sequence:** A sequence of processes <P1, P2, ... , Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j <i.

In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+1 can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks as shown in figure. An unsafe state may lead to a deadlock.

As long as the state is safe, the operating system can avoid unsafe states

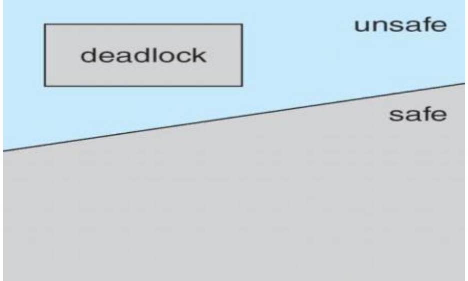


Figure 4: Safe, unsafe, and deadlocked state spaces.

Resource-Allocation-Graph Algorithm

- If a resource-allocation system has only one instance of each resource type, then a variant of the resource-allocation graph is used for deadlock avoidance.
- In addition to the request and assignment edges, a new type of edge is introduced, called a claim edge.
- A claim edge *Pi* ->*Rj* indicates that process *Pi* may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a **dashed line**.
- When process *Pi* requests resource *Rj*, the claim edge *Pi* ->*Rj* is converted to a request edge. When a resource *Rj* is released by *Pi* the assignment edge *Rj*->*Pi* is reconverted to a claim edge *Pi*->*Rj*.

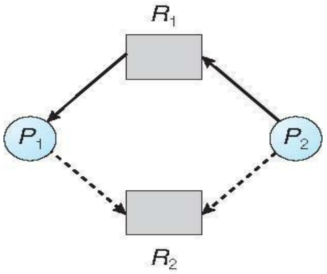


Figure 5: Resource-allocation graph for deadlock avoidance.

Note 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.
We can relax this condition by allowing a claim edge *Pi* ->*Rj* 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.

There is need to check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of processes in the system.

- If no cycle exists, then the allocation of the resource will leave the system in a safe state.
- If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process Pi will have to wait for its requests to be satisfied.

To illustrate this algorithm, consider the resource-allocation graph as shown above. 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.

A cycle, indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.

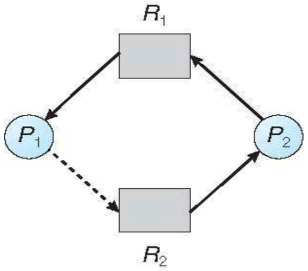


Figure 6: An unsafe state in a resource-allocation graph

Banker's Algorithm

The Banker’s algorithm is applicable to a resource allocation system with multiple instances of each resource type.

- 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.

To implement the banker's algorithm the following data structures are used.

Let n = number of processes, and m = number of resources types

Available: A vector of length *m* indicates the number of available resources of each type. If available [j] = k, there are k instances of resource type Rj available.

Max: An *n x m* matrix defines the maximum demand of each process. If Max [i,j] = 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] = k then Pi is currently allocated k instances of Rj

Need: An n x m matrix indicates the remaining resource need of each process. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task.

Need [i,j] = Max[i,j] – Allocation [i,j]

Safety Algorithm

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
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 + $Allocation_i$
Finish[i] = true
go to step 2

4. If Finish [i] == true for all i, then the system is in a safe state

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

Resource-Request Algorithm

The algorithm for determining whether requests can be safely granted.
Let $Request_i$ be the request vector for process P_i. If $Request_i$ [j] == k, then process P_i wants k instances of resource type R_j. When a request for resources is made by process Pi, the following actions are taken:

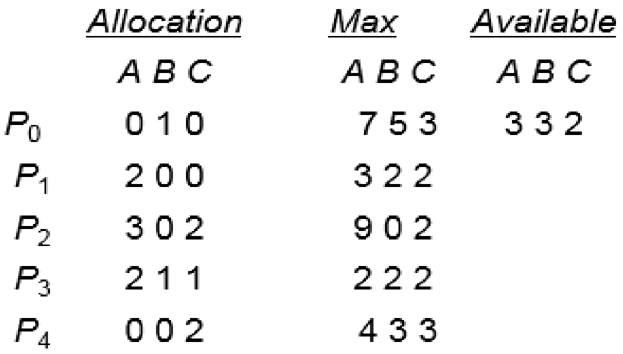
1. If $Request_i \leq Need_i$ *go* to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Have the system pretend to allocate requested resources to P_i by modifying the state as follows:

$Available = Available –$
$Request; Allocation_i=$
$Allocation_i + Request_i;$
$Need_i=Need_i –$
$Request_i;$

If safe ⇒ the resources are allocated to Pi
If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored

Example

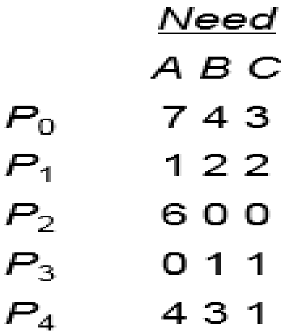
Consider a system with five processes P_o through P_4 and three resource types *A*, *B*, and C. Resource type *A* has ten instances, resource type *B* has five instances, and resource type C has seven instances. Suppose that, at time $T_0$the following snapshot of the system has been taken:



	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	

Figure 7:

The content of the matrix *Need* is defined to be *Max* - *Allocation*



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

Figure 8:

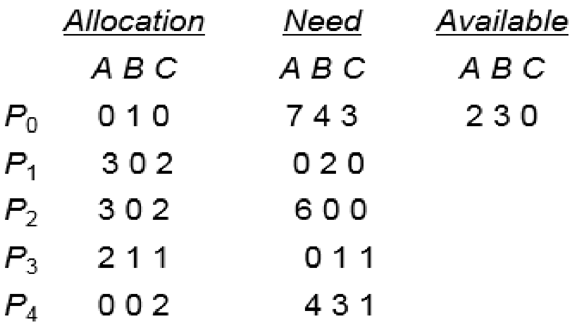
The system is currently in a safe state. Indeed, the sequence <P_1, P_3, P_4, P_2, P_0> satisfies the safety criteria.

Suppose now that process P_1 requests one additional instance of resource type A and two instances of resource type C, so $\text{Request}_1 = (1,0,2)$. Decide whether this request can be immediately granted.

Check that Request $\le$ Available

$(1,0,2) \le (3,3,2) \Rightarrow$ true

Then pretend that this request has been fulfilled, and the following new state is arrived.



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

Figure 9:

Executing safety algorithm shows that sequence <P_1, P_3, P_4, P_0, P_2> satisfies safety requirement.

Resource allocation

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

Single Instance of Each Resource Type

- If all resources have only a single instance, then define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a ***wait-for*** graph**.**
- This graph is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- An edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_i$ for some resource R_q.

Example: In below Figure, a resource-allocation graph and the corresponding wait-for graph is presented.

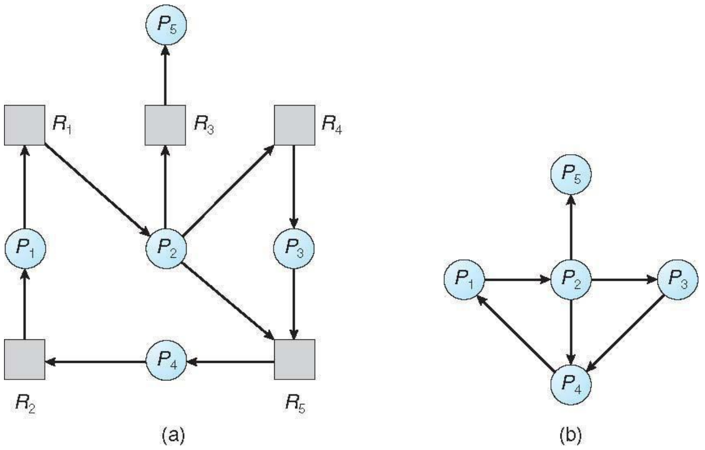


Fig 10: (a) Resource-allocation graph (b) Corresponding wait-for graph.

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

Several Instances of a Resource Type

A deadlock detection algorithm that is applicable to several instances of a resource type. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm.

- **Available**: A vector of length *m* indicates the number of available resources of each type.
- **Allocation:** An*n x m* matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An *n* x *m* matrix indicates the current request of each process. If

Request[i] [j]

equals *k,* then process *P;* is requesting *k* more instances of resource type *Rj.*

<u>Algorithm:</u>

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

(a) Work = *Available*

(b) For *i* = 1,2, …, *n*, if $Allocation_i \neq 0$, then
Finish[i] = false; otherwise, *Finish*[i] = *true*

2. Find an index *i*such that both:

(a) Finish[*i*] == *false*

(b)$Request_i \leq Work$

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

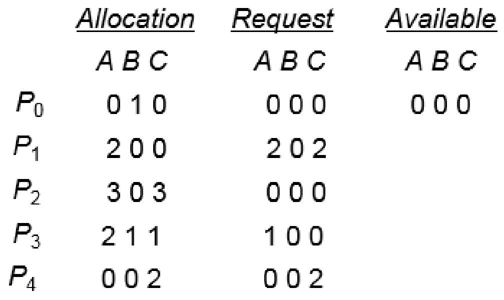
3. Work = *Work* + $Allocation_i$
Finish[*i*] = *true*
go to step 2

4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state.
Moreover, if
Finish[*i*] == *false*, then P_i is deadlocked

Algorithm requires an order of O(*m* x *n*$^{2)}$ operations to detect whether the system is in deadlocked state

<u>Example of Detection Algorithm</u>

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 T_0, the following resource-allocation state:

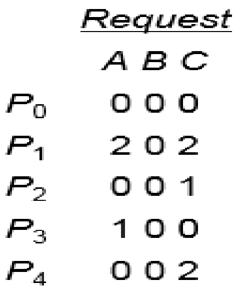


	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Figure 11:

After executing the algorithm, Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*

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



	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

Figure 12:

The system is now deadlocked. Although we can reclaim the resources held by process Po, the number of available resources is not sufficient to fulfill the requests of the other processes.
Thus, a deadlock exists, consisting of processes P1, P2, P3, and P4.

Detection-Algorithm Usage

The detection algorithm can be invoked 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.

If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

RECOVERY FROM DEADLOCK

The system recovers 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, use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

1. **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.

2. **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. Many factors may affect which process is chosen, including:

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

Resource Preemption

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

1. **Selecting a victim.** Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.

2. **Rollback.** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since it is difficult to determine what a

safe state is, the simplest solution is a total rollback: abort the process and then restart it.

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?