Module 3

Asst.Prof Alakananda K
CSE Dept
4-CY

Process Synchronization: Background, critical section problem, Peterson's solution; synchronization hardware- mutex, semaphores, monitors. **Deadlocks:** System model, necessary conditions for deadlocks, methods for handling deadlocks, deadlock prevention, deadlock avoidance -resource allocation graph algorithm, banker's algorithm, deadlock detection, recovery from deadlock.	**8 Hours**

Asst.Prof Alakananda K

Process Synchronization

- ❖On the basis of synchronization processes are categorized as one of the following two types.
- ❖ **Independent process**
- Execution of one process does not affects the execution of other process.
- **co-operative process**
- Execution of one process affects the execution of other processes.
- Process synchronization problem arises in the case of co-operative process also because resources are shared in co-operative processes.

Process Synchronization-Data inconsistency.

- 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 cause data inconsistency.
- Various mechanisms are available to ensure, the orderly execution of cooperating processes, that share a logical address space, so that data consistency is maintained.

Producers & Consumers- bounded buffer problem

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- The consumer-producer problem - solution is given with the buffers.
- A bounded buffer - multiple producers and multiple consumers share a single buffer.
- Producers write data to the butter, and consumers read data tram the buffer.
- Producers must block if the buffer is full.
- Consumers must block if the buffer is empty.

- An integer **count** that keeps track of the number of full buffers.
- Initially, count is set to 0.
- It is incremented by the producer after it produces a new buffer and
- It is decremented by the consumer after it consumes a buffer.

Producer

- The producer produces the buffer- until the size is full (reaches n)

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

Consumer

The consumer consumes the buffer until the size is 0 (reaches 0)

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

Race condition

- count++ could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1
- count-- could be implemented as
  register2 = count
  register2 = register2 - 1
  count = register2
- Consider this execution interleaving with "count = 5" initially:

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

RACE CONDITION

- Consider two cooperating processes, sharing variables A and B and having the following set of instructions in each of them:

Process 1	Process 2	Concurrent access
A=1	B=2	Does not matter
A= B+1 B=B+1	B=B*2	Important!

- Suppose our intention is to get **A as 3** and **B as 6** after the execution of both the processes. The interleaving of these instructions should be done in order to avoid race condition.
- If the order of execution is like:

```
A=1
B=2
A= B+1
B=B+1
B=B*2
```

A will contain 3 and B will contain 6, as desired.

whereas if the order of execution is like:

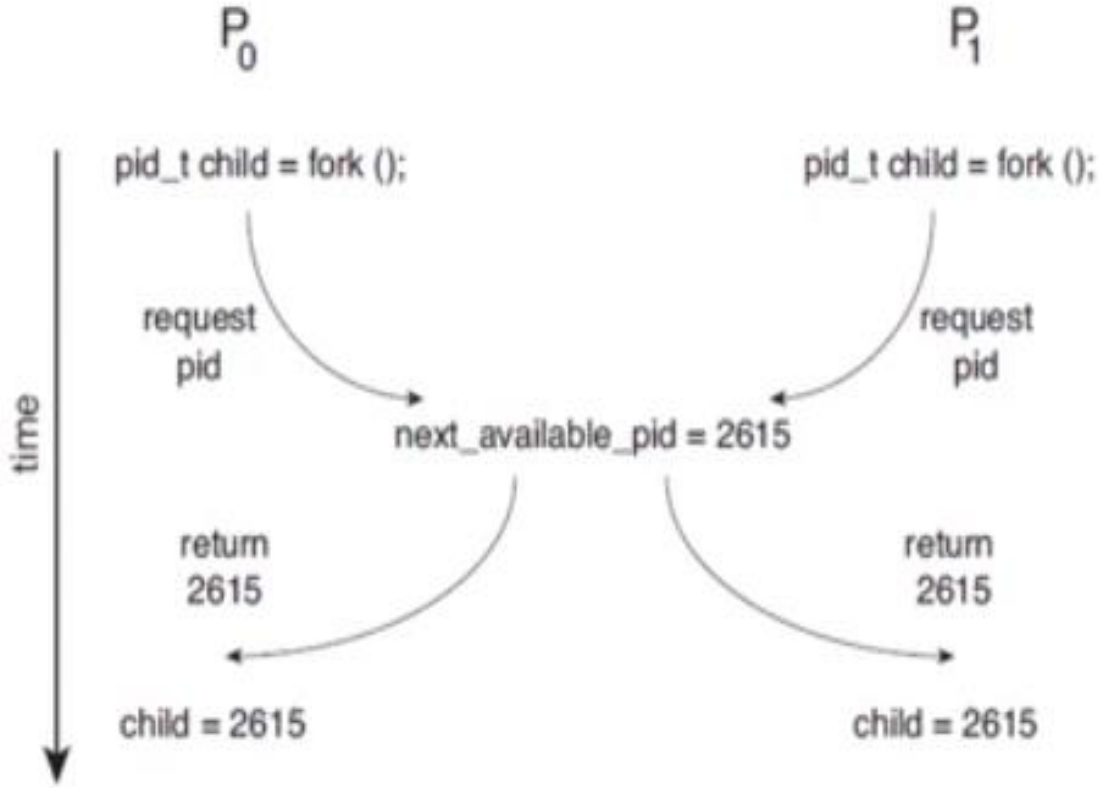
```
A=1
B=2
B=B*2
A= B+1
B=B+1
```

A will contain 5 and B will contain 5 which is not desired.

- We got this incorrect value because of both the processes executes concurrently.
- Several processes access and manipulate the same data concurrently, and the result depends on the particular order in which the access takes place, **is called a race condition.**
- To overcome the race condition, we need to ensure that only one process at a time can be manipulating the variable counter.
- To make such a guarantee, we require that the processes be synchronized in some way.

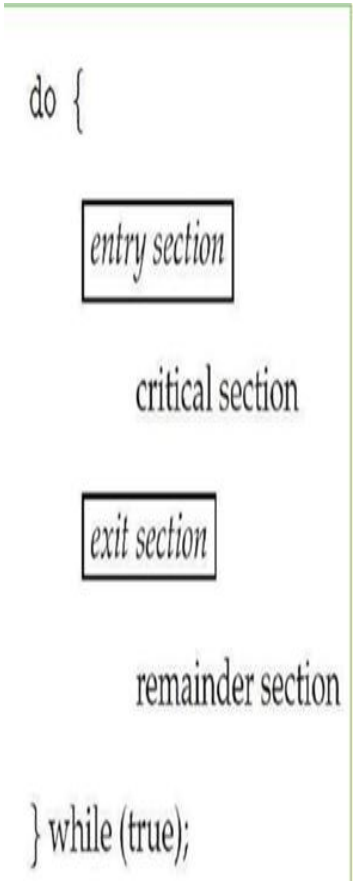
Race condition –when assigning a pid

- Processes P0 and P1 simultaneously calls fork(), the next available pid is 2615 assigned to both children. (that should not be happened)



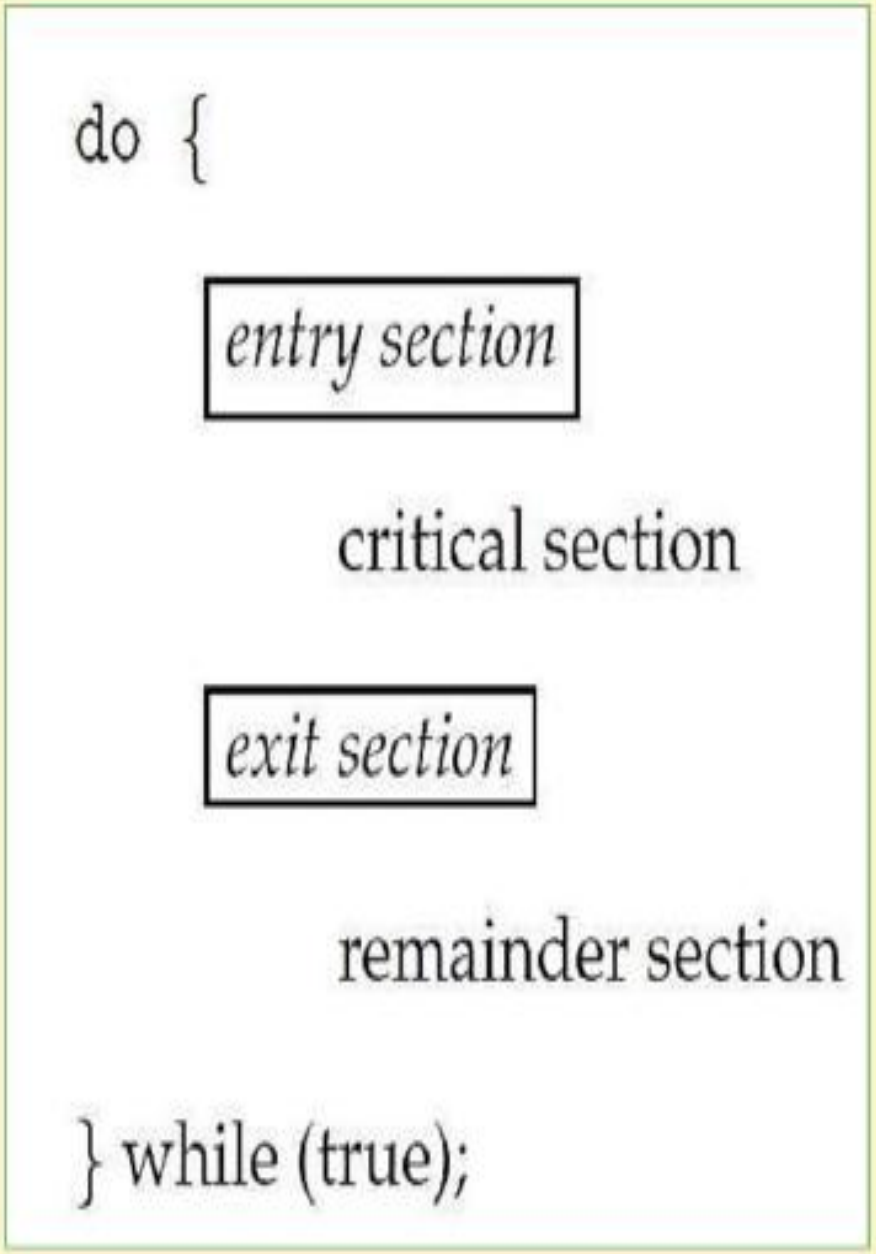
CRITICAL SECTION PROBLEM

- Consider a system consisting of n processes {P0, 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 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.
- 1. Each process must request permission to enter its critical section.
- 2. The section of code implementing this request is the **entry section.**
- 3. The critical section may be followed by an **exit section.**
- 4. The remaining code is the **remainder section.**
- Thus the critical section of a process should not be executed concurrently with the critical section of another process.
- This should be ensuring by the synchronization mechanism.

- The general structure of a typical process Pi is shown in Figure:

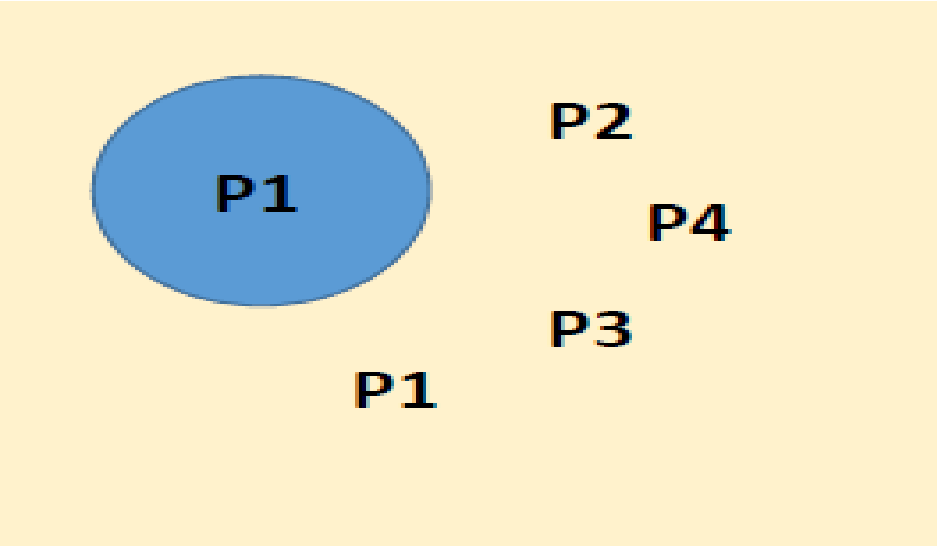


- A solution to the critical-section problem must satisfy the following three requirements:(**Mutual exclusion, Progress, Bounded waiting)**
- **Mutual exclusion:** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.(no two processes will simultaneously be inside their critical section)
- **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.

Progress:

- If no process is executing in its critical section and there exist some processes that wish to enter their critical section,
- then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
- Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it.

- **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.



Bounded waiting:

- A bound must exist 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.
- Bounded waiting means that each process must have a limited waiting time.
- It should not wait endlessly to access the critical section

PETERSON'S SOLUTION

- It is a classic software-based solution to the critical-section problem.

Process Pi Code

```
do
 { flag[i]=TRUE;
   turn = j;
while( flag[ j] and turn = j )
do no-op;
CRITICAL SECTION
flag[ i] = FALSE;
REMAINDER SECTION
} while(1)
```

Process Pj Code

```
do
  { flag[j] = TRUE;
    turn= i;
   while( flag[ i] and turn = i )
      do no-op;
    CRITICAL SECTION
   flag[ j] = FALSE;
    REMAINDER SECTION
   } while(1)
```

- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered P0 and P1. For convenience, when presenting Pi, we use Pj to denote the other process; that is, j equals 1 – i.
- Peterson's solution requires the two processes to share two data items:

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

- The variable turn indicates whose turn it is to enter its critical section. That is, **if turn == i, then process Pi is allowed to execute in its critical section.**

- The flag array is used to indicate if a process is ready to enter (intention) its critical section. For example, if **flag[i] is true**, this value indicates that Pi is ready to enter its critical section.
- To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately.
- The final value of turn determines which of the two processes is allowed to enter its critical section first.

➢Here,

- Mutual exclusion is preserved.
- The progress requirement is satisfied.
- The bounded-waiting requirement is met
- To prove property 1, we note that each Pi enters its critical section only if either flag[j] == false or turn == i.
- To prove properties 2 and 3, we note that a process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j; this loop is the only one possible.

- If Pj is not ready to enter the critical section, then flag[j] == false, and Pi can enter its critical section.
- If Pj has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then Pi will enter the critical section. If turn == j, then Pj will enter the critical section.
- However, once Pj exits its critical section, it will reset flag[j] to false, allowing Pi to enter its critical section. 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 (bounded waiting).

SYNCHRONIZATION HARDWARE

- The software-based solution to the critical-section Problem such as Peterson's are not guaranteed to work on modern computer architectures. The following solutions are based on the premise of locking —that is, protecting critical regions through the use of locks.

❖ Disable interrupts

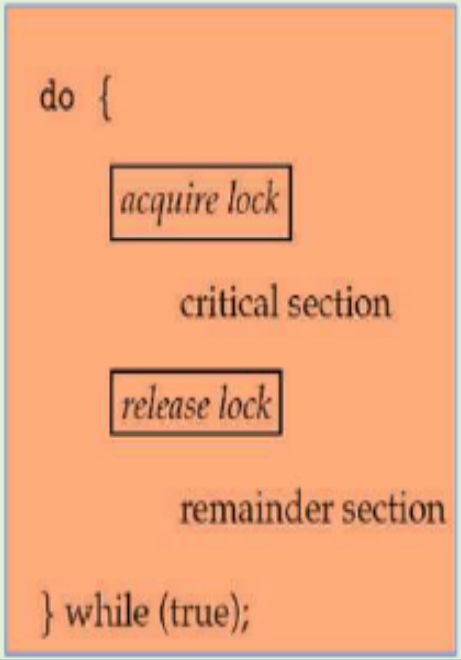
- The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified by a process.

- In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption.
- No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by non-preemptive kernels.
- Unfortunately, this solution is not as feasible in a multiprocessor environment

❖ Mutex Locks

- Operating-systems designers build software tools to solve the critical-section problem

- The simplest of these tools is the mutex lock. (In fact, the term mutex is short for mutual exclusion.)
- We use the mutex lock to protect critical regions and thus prevent race conditions.
- That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The acquire() function acquires the lock, and the release() function releases the lock.



- The simplest software tools to solve the critical-section problem is the **mutex lock.**
- The mutex lock is used to protect critical regions and thus prevent race conditions.
- **A process must get the lock by acquire() before entering a critical section;**
- **it releases the lock by release() when it exits the critical section.**

- A mutex lock has a Boolean variable —**available** whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released

The definition of acquire() is as follows:

```
acquire()
{
  while (!available)
    ;   /* busy wait */
available = false;
}
```

The definition of release() is as follows:

```
release()
{
   available = true;
}
```

- Calls to either acquire() or release() must be performed atomically. The main disadvantage of the implementation given here is that it requires busy waiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().
- In fact, this type of mutex lock is also called a **spinlock** because the process spins while waiting for the lock to become available.
- This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes.

Disadvantage of Mutex

- The main disadvantage is **busy waiting**.
- While one process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().
- This type of mutex lock is also called a **spinlock** because the process "spins" while waiting for the lock to become available.
- 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.

The definition of acquire() is as follow

```
acquire()
{
   while (!available); /* busy wait */
   available = false;
}
```

The definition of release() is as follow

```
release()
{
    available = true;
}
```

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

- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- Spinlocks do have an advantage that no context switch is required when a process must wait on a lock, and a context switch may take considerable time.
- Thus, when locks are expected to be held for short times, spinlocks are useful.
- They are often employed on multiprocessor systems where one thread can spin on one processor while another thread performs its critical section on another processor.

SEMAPHORES

- Semaphore is used to solve critical section problem. A semaphore **S** is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait()** and **signal()**
- The wait() operation was originally termed P , 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++;
}
```

- Entry to the critical section is controlled by the wait operation.
- Exit from a critical region is taken care by signal operation.

- The wait() operation.
- If argument S is negative or zero, then no operation is performed.
- If the argument S is positive then, it decrements the value
- Definition of P(S) or wait(S):

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

- The signal operation increments the value of its argument S.
- Definition of V(S) or signal(S):

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

- The wait() operation decrements the semaphore value. If the value becomes negative, then the process executing the wait is blocked.
- The signal() operation increments the semaphore value. If the value is not positive, then the process blocked by a wait() operation is unblocked.
- Semaphore may be initialized to a non-negative value.
- Semaphores are executed automatically.

- Types of Semaphores
- The Binary Semaphores
- The Counting Semaphores
- The binary semaphores value is restricted to 0 and 1.
- The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0.
- It is sometimes easier to implement binary semaphores than counting semaphores.

Process1

- The common variable S
- Initially S = 1.

Definition of P(S) or wait(S):

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

Definition of V(S) or signal(S):

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

Process2

Definition of P(S) or wait(S):

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

Definition of V(S) or signal(S):

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

- Counting Semaphores :
- Integer value semaphores has greater
- The semaphore count is the number of available resources, used to coordinate the resource access.
- If the resources are added, semaphore count automatically incremented and
- If the resources are removed, the count is decremented.
- based on the resource count, the same number or processes are allowed in the critical section.
- Here also, the wait() and signal() operations are un-interrupted.

Process1 Process2 Process3

- The common variable S, Initially S = 2.

Definition of P(S) or wait(S):

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

Definition of V(S) or signal(S):

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

Definition of P(S) or wait(S):

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

Definition of V(S) or signal(S):

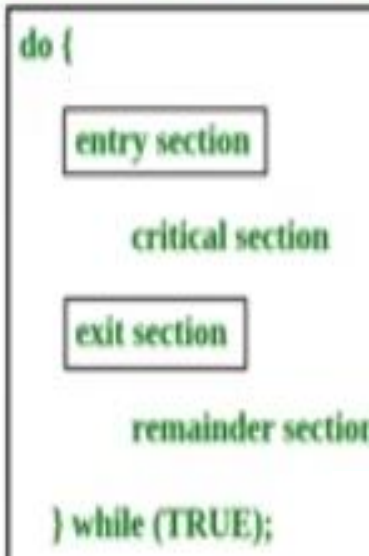
```
signal(S)
{
   S++;
}
```

Definition of P(S) or wait(S):

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

Definition of V(S) or signal(S):

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



❖Semaphore Implementation

- The implementation of mutex locks suffers from busy waiting. The definitions of the wait() and signal() semaphore operations just described present the same problem.
- To overcome the need for busy waiting, we can modify the definition of the wait() and signal()operations as follows:

- 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. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

❖To implement semaphores under this definition, 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.

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.

- Note that in this implementation, semaphore values may b negative. If a semaphore value is negative, its magnitude is th number of processes waiting on that semaphore

❖**Some properties of semaphore are:**

- ➢Semaphores are machine independent
- ➢Semaphores are simple to implement
- ➢Correctness is easy to determine
- ➢Can have many different critical sections with different semaphores
- ➢Semaphore acquire many resources simultaneously

MONITORS

- The incorrect use of semaphores can result in timing errors.
- 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.

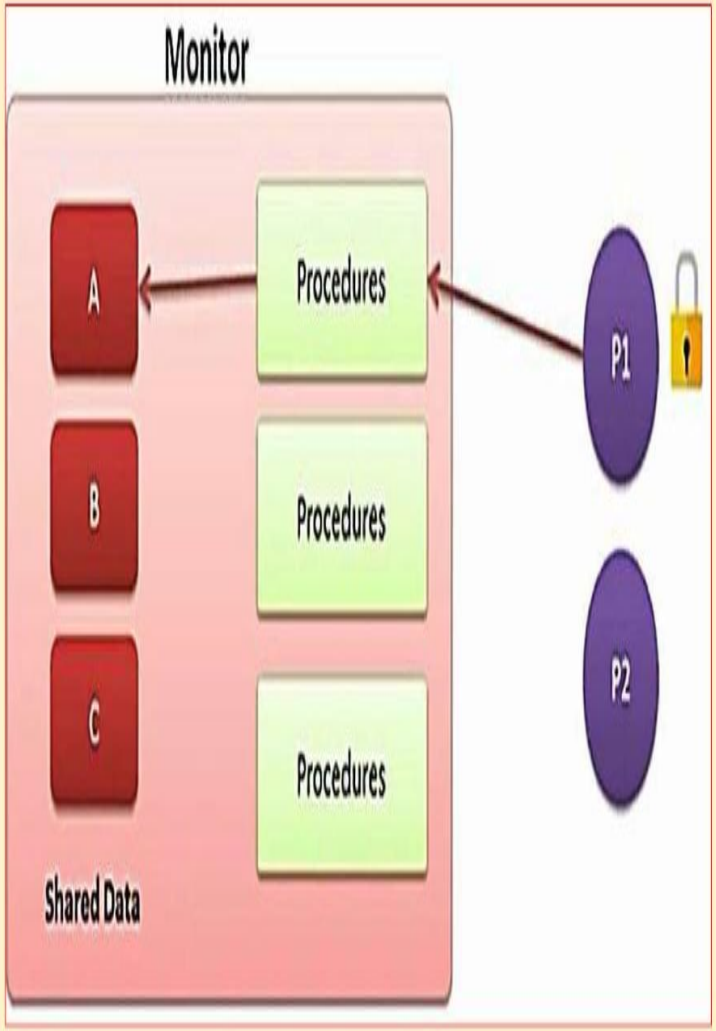
- Suppose that a process omits the wait(mutex), or the signal(mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

- To deal with such errors, researchers have developed high-level language constructs- the **MONITOR**
- Monitors are based on abstract data types.
- A monitor is a programming language construct that provides equivalent functionality to that of semaphores but is easier to control.
- Monitor provides high-level of synchronization.

➢ Monitor is a module that encapsulates

- Shared data structures
- Procedures that operates on shared data
- Synchronization between concurrent procedure invocations.

➢ Only one process is allowed to enter in Monitor at a time



Syntax of a monitor

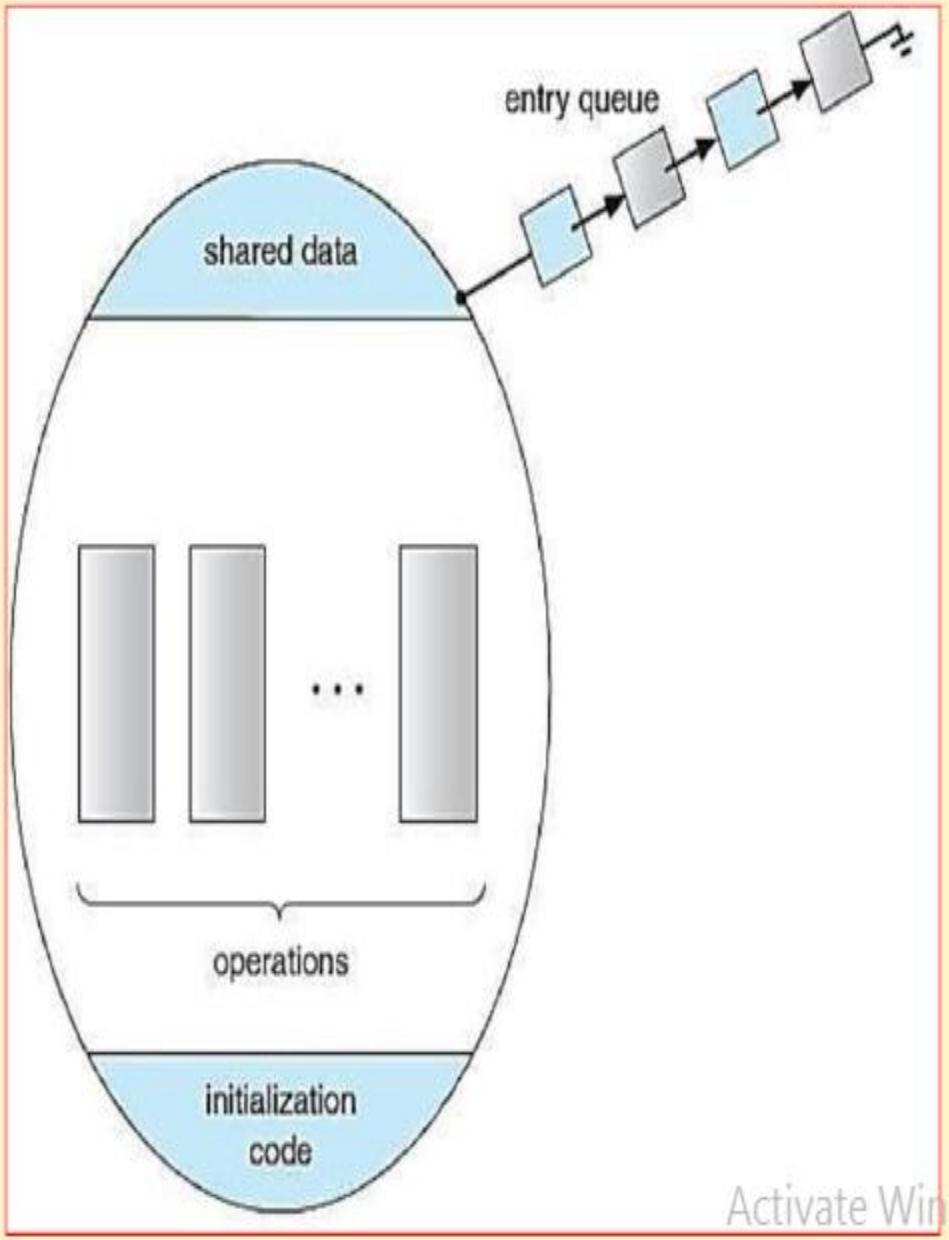
```
monitor monitor name
{
  /* shared variable declarations */

  function P1 ( . . . ) {
     . . .
  }

  function P2 ( . . . ) {
     . . .
  }

        .
        .
        .
  function Pn ( . . . ) {
     . . .
  }

  initialization_code ( . . . ) {
     . . .
  }
}
```

- The monitor construct ensures that only one process at a time is active within the monitor.
- Consequently, the programmer does not need to code this synchronization constraint explicitly.
- However, the monitor construct, is not sufficiently powerful for modeling some synchronization schemes.
- For this purpose, we need to define additional synchronization mechanisms.
- These mechanisms are provided by the condition construct.

```
condition x, y;
```

- The only operations that can be invoked on a condition variable are wait() and signal().
- The operation x.wait(); means that the process invoking this operation is suspended until another process invokes x.signal();
- The x.signal() operation resumes exactly one suspended process.
- If no process is suspended, then the signal() operation has no effect; that is, the state of x is the same as if the operation had never been executed

Schematic view of a monitor



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

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

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

Monitor Type

- An **Abstract Data Type**—or **ADT**—encapsulates data with a set of functions to operate on that data.
- A ***monitor type*** is an ADT that includes a set of user defined functions (operations) that are provided with **mutual exclusion** within the monitor.
- The monitor type also declares the variables, along with functions that operate on those variables.

CLASSICAL PROBLEMS OF SYNCHRONIZATION

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

Chapter 8

- **Deadlocks:**
- System model,
- necessary conditions for deadlocks,
- methods for handling deadlocks,
- deadlock prevention,
- deadlock avoidance -resource allocation graph algorithm,
- banker's algorithm,
- deadlock detection,
- recovery from deadlock.

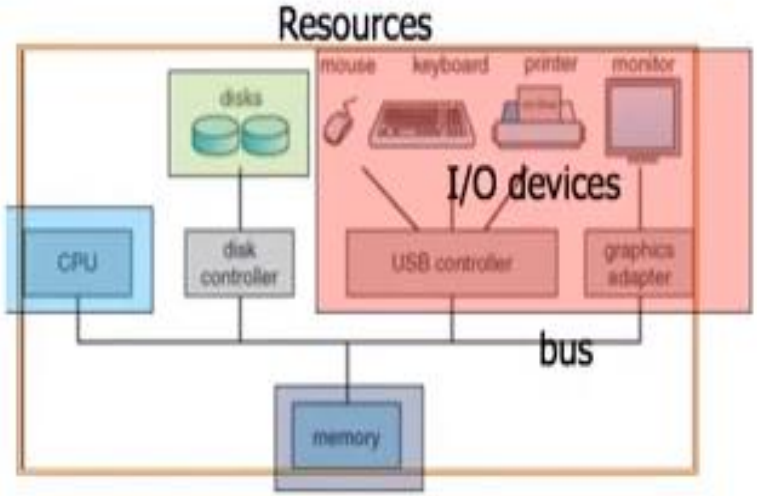
Deadlock

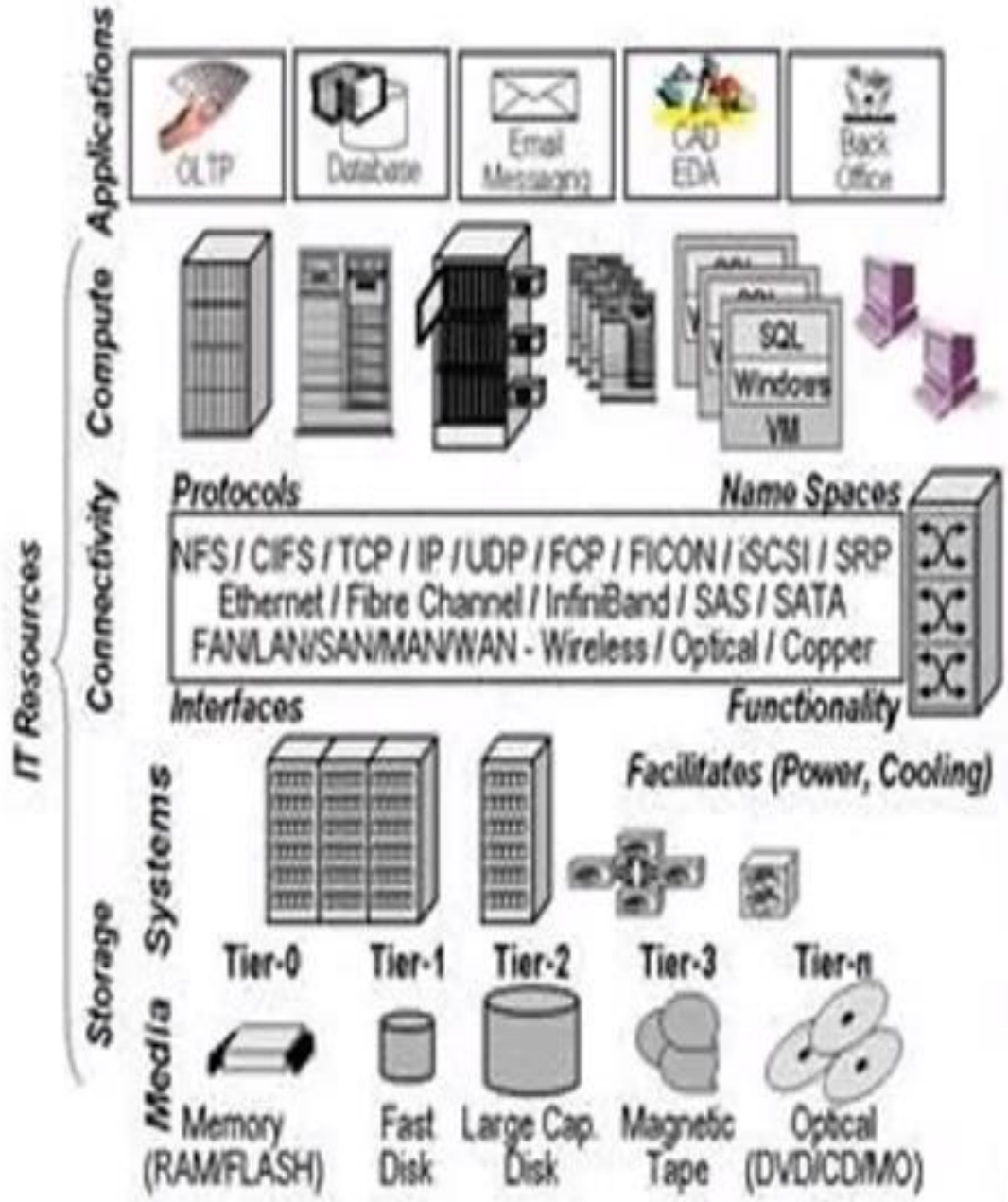
- A process requests resources for its execution,
- if the resources are not available at that time, the process enters a waiting state.
- Sometimes, the resources it has requested are held by other waiting processes, hence the current process never changes its (waiting) state.
- This situation is called a Deadlock.

Resources - 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.

System Resources





Resources - System Model

- If a system has two CPUs, then the resource type CPU has two instances.
- Similarly, the resource type printer may have five instances.
- If a process requests an instance of a resource type, the allocation of any instance of the type should satisfy the request.
- If it does not, then the instances are not identical, and the resource type classes have not been defined properly.

System Model - Sequence of Resource Utilization

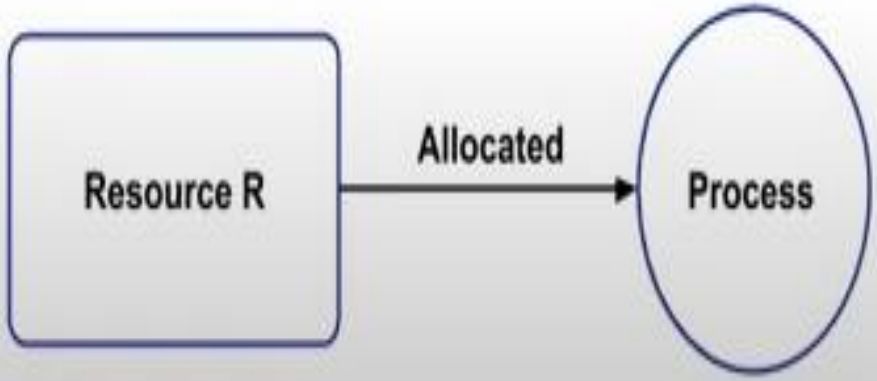
- Under the normal mode of operation, a process utilize a resource in only the following sequence:
- 1. Request.
- The process requests the resource.
- If the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
- 2. Use.
- The process can operate on the resource.
- 3. Release.
- The process releases the resource.

Necessary Conditions

- A deadlock situation can arise, if the following four conditions hold simultaneously in a system
  - 1. Mutual exclusion.
  - 2. Hold and wait.
  - 3. No preemption.
  - 4. Circular wait.

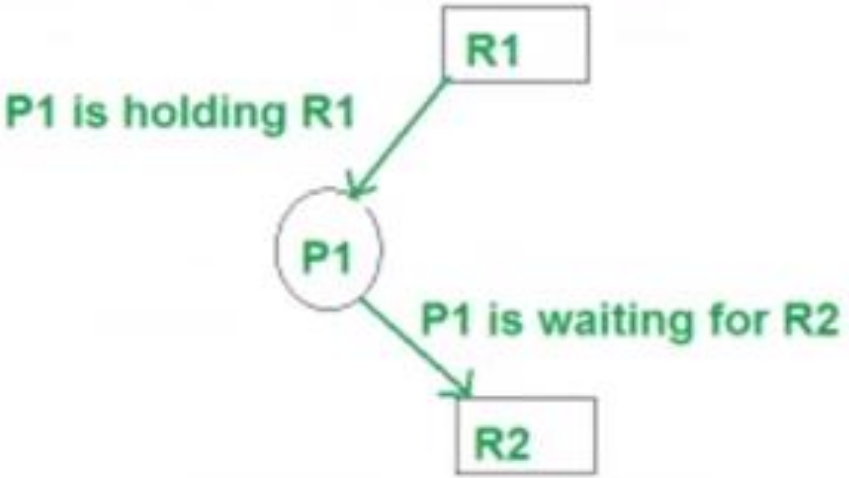
1. Mutual exclusion

- 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 wait until the resource has been released.



2. Hold and wait

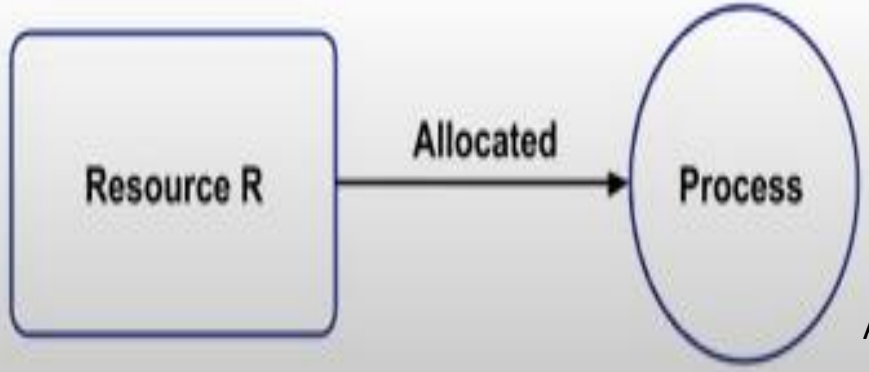
- A process must be holding at least one resource and
- waiting to acquire additional resources
- that are currently being held by other processes.



HOLD AND WAIT

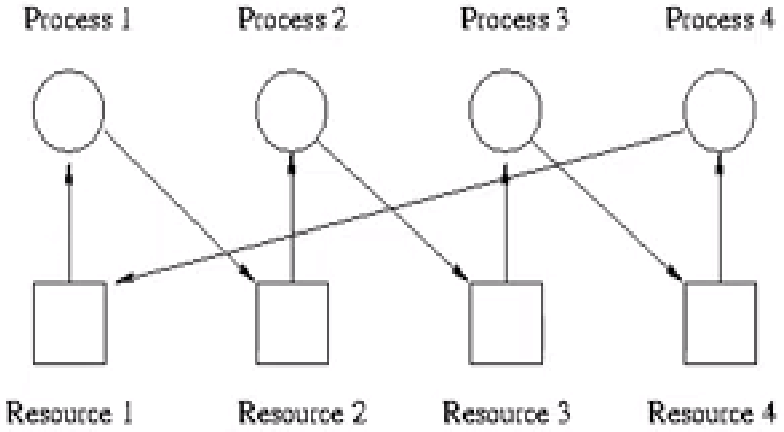
3. No preemption.

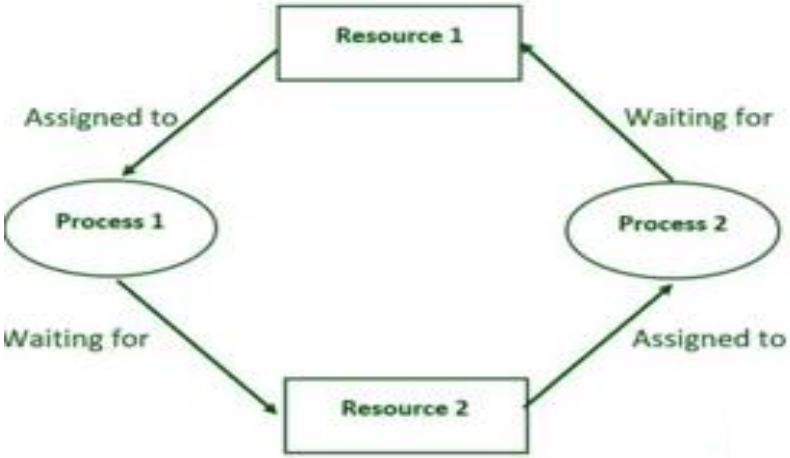
- Resources cannot be preempted;
- that is, a resource can be released only voluntarily by the process holding it,
- after that process has completed its task.



4. Circular wait.

- A set {P0, 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.

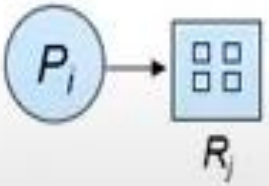


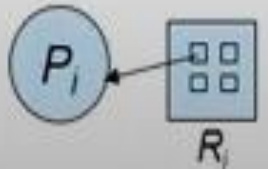


Resource-Allocation Graph (RAG)

- A graph G consists of a set of vertices V and a set of edges E
- G={V,E}
- In Resource-Allocation Graph - RAG,
- V is partitioned into two types:
  - $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system
- Two types of directed edge
- **Request edge** – directed edge $P_i \rightarrow R_j$
- **Assignment edge** – directed edge $R_j \rightarrow P_i$

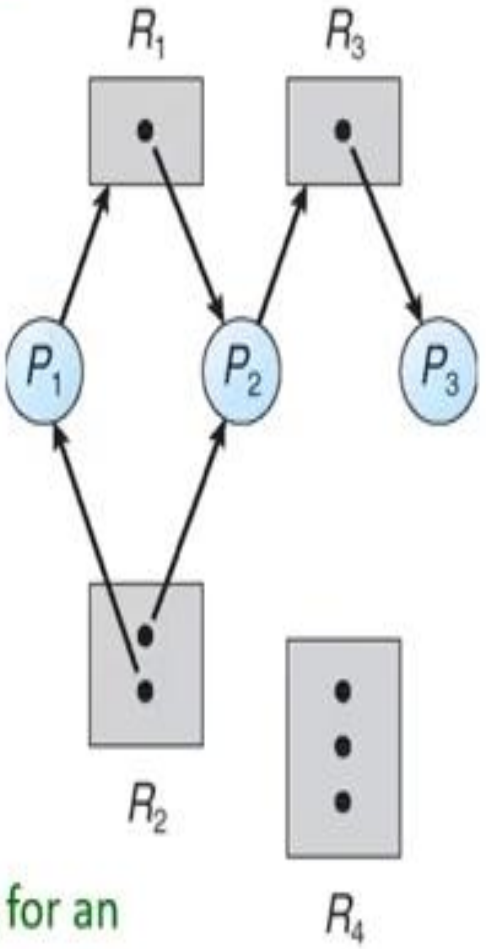
Resource-Allocation Graph (Cont.)

- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

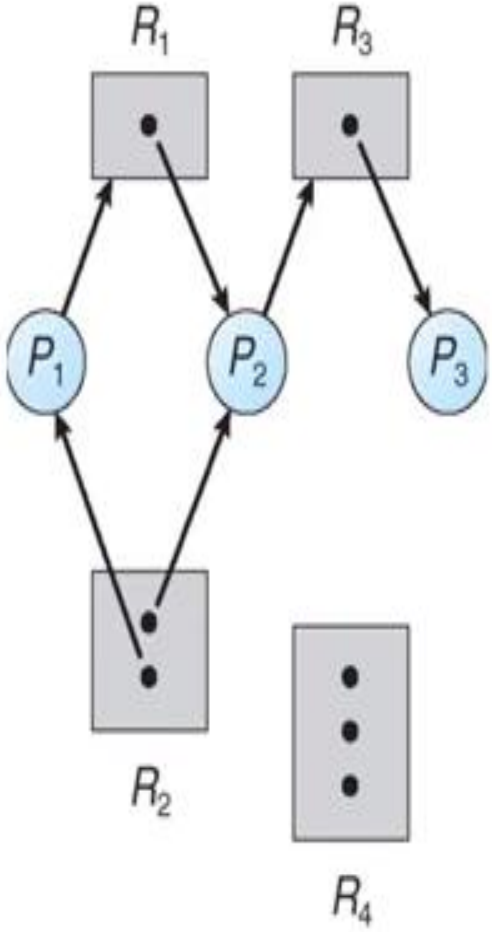
Example of a Resource Allocation Graph

- The sets P, R, and E:
  - P = {P1, P2, P3}
  - R = {R1, R2, R3, R4}
  - E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}
- Resource instances:
  - One instance of resource type R1
  - Two instances of resource type R2
  - One instance of resource type R3
  - Three instances of resource type R4
- Process states:
  - Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
  - Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
  - Process P3 is holding an instance of R3.



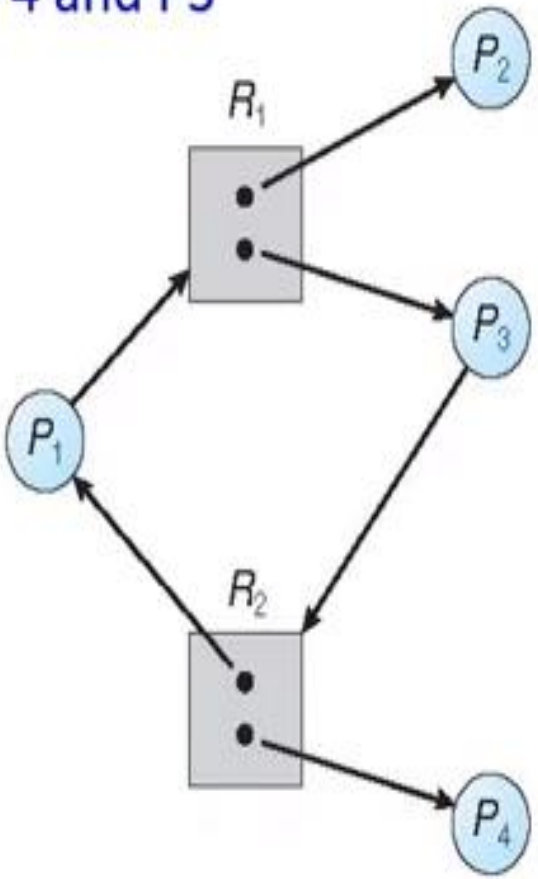
Example of a Resource Allocation Graph...

- If the graph contains no cycles, then no process in the system is deadlocked.
- In this graph, there is no cycle.
- The execution sequence of P1, P2 and P3 are
  - 1. P3 executes first, then it releases the R3
  - 2. R3 then assigned to P2, hence P2 completes its execution
  - Then it releases the resources R1, R2 and R3
  - 3. Now R1 assigned to P1, and P1 completes its execution.

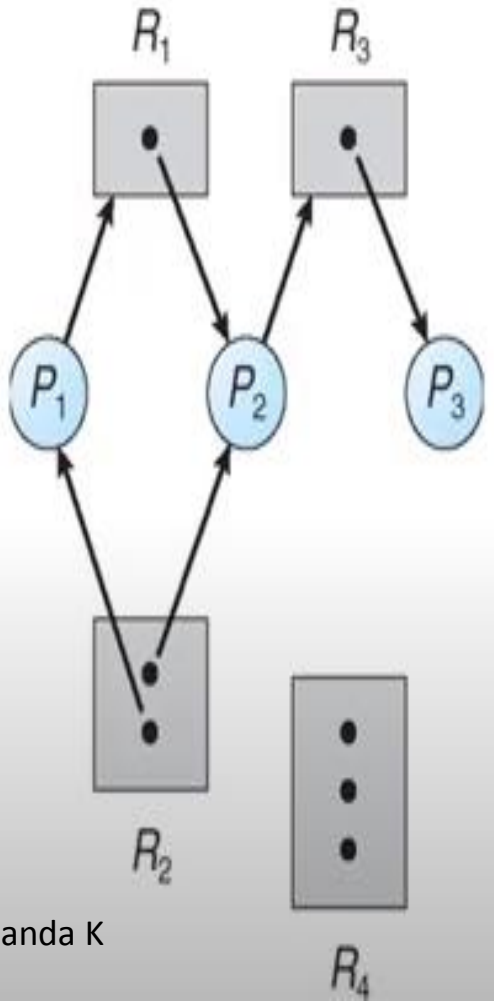


Graph With A Cycle But No Deadlock

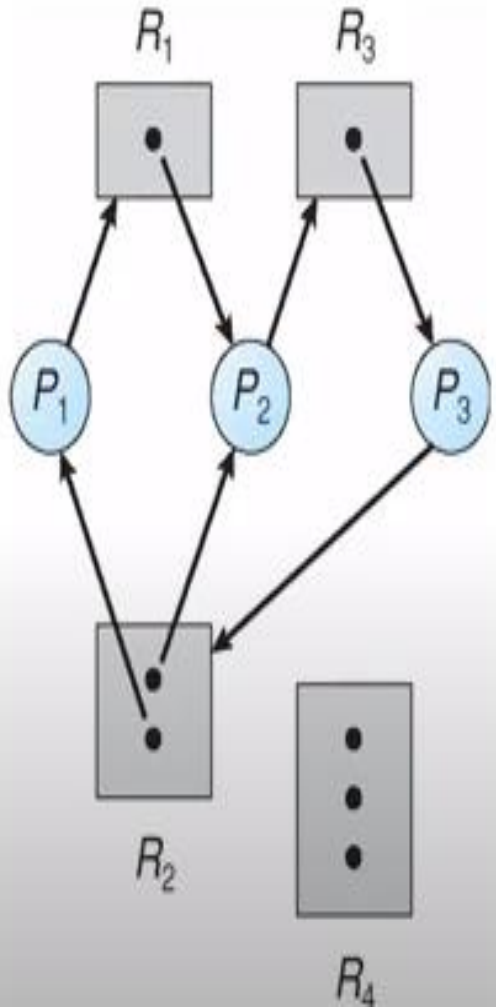
- The execution sequence of P2,P1, P4 and P3



Resource Allocation Graph Without A Deadlock



Resource Allocation Graph With A Deadlock



- Here all the process are holding one resource and waiting for another resource, and those resource are already held by some other process
- Hence none of the process complete its execution, so the system will be in deadlock state.

Basic Facts identified from Resource Allocation Graph

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods of handling deadlocks: There are four approaches to dealing with deadlocks.

- **1.** Deadlock Prevention
- **2.** Deadlock avoidance (Banker's Algorithm)
- **3.** Deadlock detection & recovery
- **4.** Deadlock Ignorance (Ostrich Method)

Deadlock Prevention

Mutual Exclusion

- Two types of resources, sharable and non-sharable.
- The Sharable resources, do not require mutually exclusive access and thus cannot be involved in a deadlock.
- Read-only files are a good example of a sharable resource.
- If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.
- A process never needs to wait for a sharable resource.
- The Non-sharable resources
- That is, at least one resource must be non-sharable.
- **We cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are basically non sharable.**
- For example, a mutex lock cannot be simultaneously shared by several processes.
- The mutual exclusion condition must hold.

Hold and Wait

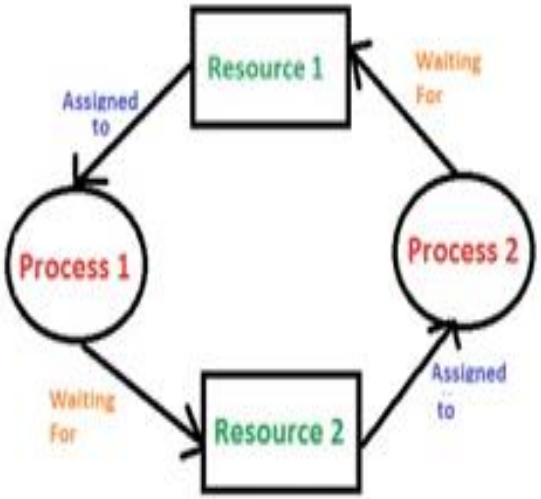
- A process is holding some resources, and it requires additional resource to complete its execution, which is already allocated to some other process.
- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
- Before it can request any additional resources, it must release all the resources that it is currently allocated.

No Preemption

- There should be no preemption of resources, that have already been allocated.
- 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.
- The preempted resources are added to the list of resources for which the process is waiting.
- The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Circular Wait

- We assign to each resource type a unique integer number, which allows us to compare two resources
- Each process requests resources in an increasing order of enumeration.
- For example, if the set of resource types R includes scanner, disk drives, and printers, then the function F might be defined as follows:
- F(scanner) = 1
- F(disk drive) = 5
- F(printer) = 12
- the process can request instances of resource type Rj if and only if
- F(Rj) > F(Ri).
- Where Rj= new request for resource
- Ri = already hold resource



Deadlock Avoidance

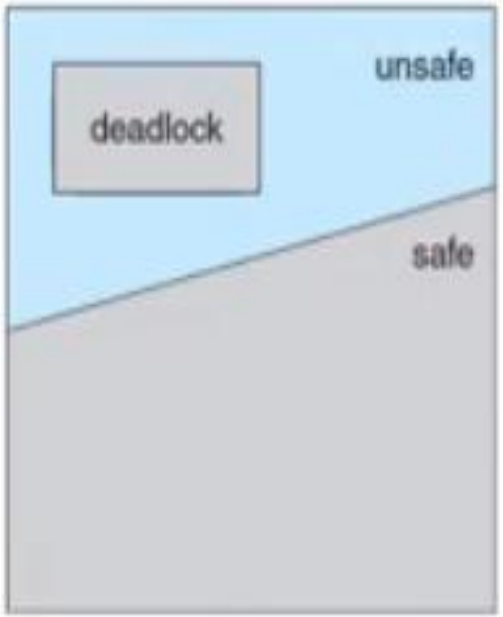
- Avoiding deadlocks is to require additional information about how resources are to be requested.
- This model requires that each process declare the maximum number of resources of each type that it may need.
- With this information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state.
- The resource allocation state is defined by the number of available and allocated resources and the maximum demands of the processes.

Safe State

- When a process requests an available resource, system must decide if immediate allocation, leaves the system in a safe state.
- Assume a system consists of a sequence of processes $<P_1, P_2, ..., P_n>$ and some of resources are allocated to that.
- Then,
  - 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
- System is in **safe state**

Safe State, Unsafe state and Deadlock state Spaces

- A safe state is not a deadlocked state.
- Conversely, a deadlocked state is an unsafe state.
- Not all unsafe states are deadlocks, but an unsafe state may lead to a deadlock

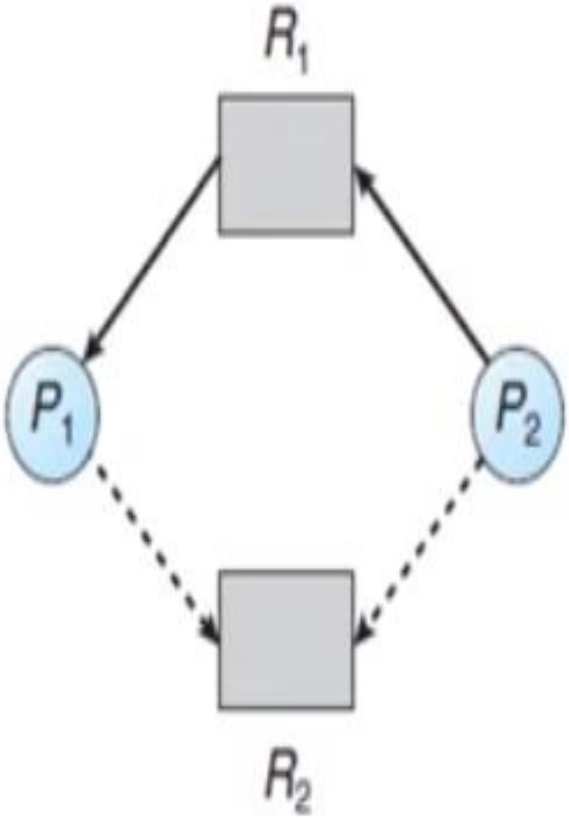


Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the Banker's Algorithm

Resource-Allocation-Graph Algorithm

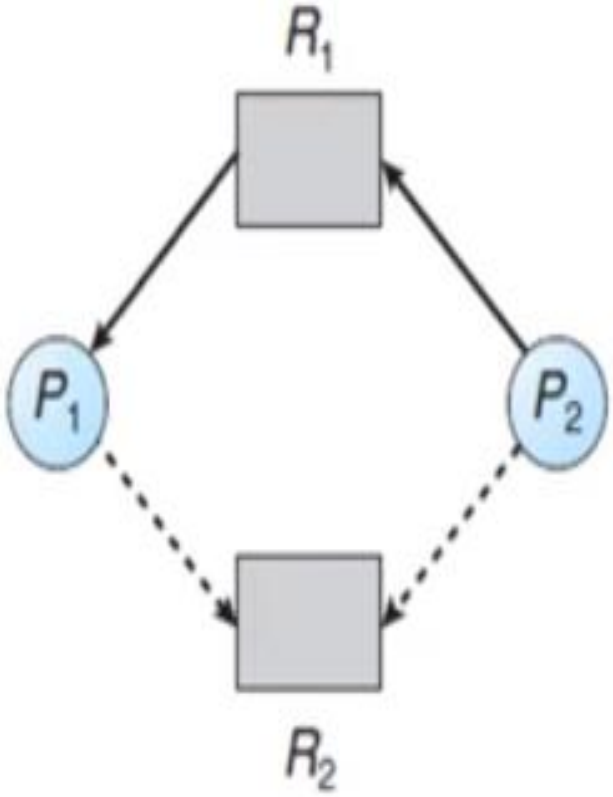
- Introduce a new type of edge, represented in dashed line, called a claim edge.
- A claim edge Pi → Rj indicates that process Pi may request resource Rj at some time in the future.
- Suppose the process Pi requests resource Rj.
- The request can be granted only if converting the request edge Pi → Rj to an assignment edge Rj → Pi does not form a cycle in the resource-allocation graph.



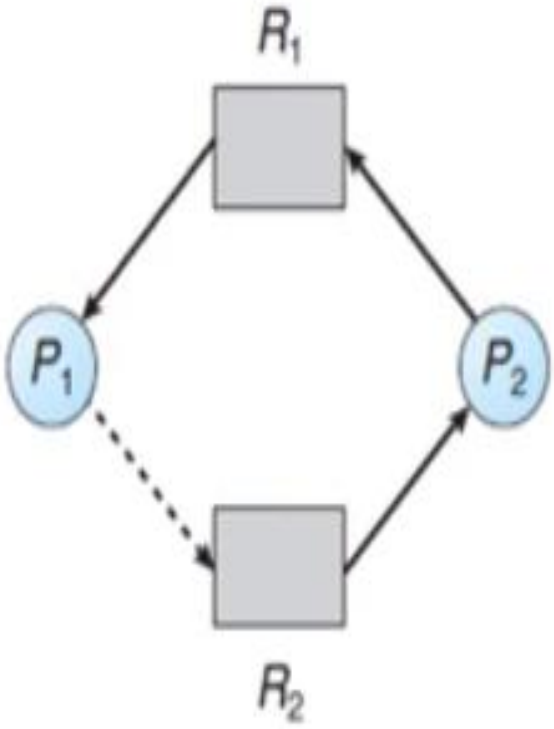
Resource-Allocation-Graph Algorithm...

- We check for safety by using a cycle-detection algorithm.
- 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.

- P2 requests R2.
- Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph.



Resource-allocation graph for deadlock avoidance



An unsafe state in a resource-allocation graph

Banker's Algorithm

- If a system has multiple instances of each resource type, then resource-allocation-graph algorithm is not applicable, hence we use another powerful algorithm to avoid deadlock.
- This algorithm is commonly known as the banker's algorithm.
  - Available. A vector of length m indicates the number of available resources of each type.
  - Max. defines the maximum demand of each process.
  - Allocation. defines the number of resources of each type currently allocated to each process.
  - Need. indicates the remaining resource need of each process.
  - Need = Max – Allocation.

Bankers algorithm also known as safety algorithm ,because it ensure the safety of the system

Safety Algorithm

- 1. Initialize Work = Available and Finish[i] = false for i = 0, 1, ..., n − 1.
- 2. Find an process i such that both
  - a. Finish[i] == false
  - b. Needi ≤ Work
- 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.

Banker's Algorithm - Example

- Consider a system with five processes P0 through P4 and three resource types A, B, and C.
- Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances.
- Suppose that, at time T0, the following snapshot of the system has been taken:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Banker's Algorithm – Example...

- The Need is defined to be Max – Allocation and is as follows:

	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	

	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

- A B C
- 10 5 7

- Work = Available and Finish[i] = false
- for i = 0, 1, ..., n – 1.
- P0: 1. work = available
- Work = 3 3 2
- 2. Need <= work
- 7 4 3 <= 3 3 2 (false)
- P1: work = available
- Work = 3 3 2
- 2. Need <= work
- 1 2 2 <= 3 3 2 (true)
- 3. work = work + allocation
- = 3 3 2 + 2 0 0 = 5 3 2

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

F	F	F	F	F

- P2 : 1. work = available
- Work = 5 3 2
- 2. Need <= work
- 6 0 0 <= 5 3 2 (false)
- P3: work = available
- Work = 5 3 2
- 2. Need <= work
- 0 1 1 <= 5 3 2 (true)
- 3. work = work + allocation
- = 5 3 2 + 2 1 1 = 7 4 3
- P4: work = available
- Work = 7 4 3
- 2. Need <= work
- 4 3 1 <= 7 4 3 (true)
- 3. work = work + allocation
- = 7 4 3 + 0 0 2 = 7 4 5

	Allocation	*Max*	*Available*		*Need*
	A B C	*A B C*	*A B C*		*A B C*
P_0	0 1 0	7 5 3	3 3 2	P_0	7 4 3
P_1	2 0 0	3 2 2		P_1	1 2 2
P_2	3 0 2	9 0 2		P_2	6 0 0
P_3	2 1 1	2 2 2		P_3	0 1 1
P_4	0 0 2	4 3 3		P_4	4 3 1

F	~~F~~ T	F	F	F

P_0 P_1

	Allocation	Max	Available		Need
	A B C	A B C	A B C		A B C
P_0	0 1 0	7 5 3	3 3 2	P_0	7 4 3
P_1	2 0 0	3 2 2		P_1	1 2 2
P_2	3 0 2	9 0 2		P_2	6 0 0
P_3	2 1 1	2 2 2		P_3	0 1 1
P_4	0 0 2	4 3 3		P_4	4 3 1

F	F	F	F	F

- P0: work = available
- Work = 7 4 5
- 2. Need <= work
- 7 4 3 <= 7 4 5 (true)
- 3. work = work + allocation
- = 7 4 5 + 0 1 0 = 7 5 5
- P2: work = available
- Work = 7 5 5
- 2. Need <= work
- 6 0 0 <= 7 4 5 (true)
- 3. work = work + allocation
- = 7 5 5 + 3 0 2 = 10 5 7

Banker's Algorithm – Example...

- Result – if the process execution is in following order, then the system will be safe.

Suppose now that process P_1 requests one additional instance of resource type A and two instances of resource type C, so $Request_1 = (1,0,2)$. Decide whether this request can be immediately granted.

Check that Request ≤ Available

$(1,0,2) \leq (3,3,2) \Rightarrow$ 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	

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

8.3 Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
T_0	0 0 1 2	0 0 1 2	1 5 2 0
T_1	1 0 0 0	1 7 5 0	
T_2	1 3 5 4	2 3 5 6	
T_3	0 6 3 2	0 6 5 2	
T_4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix *Need*?

b. Is the system in a safe state?

c. If a request from thread T_1 arrives for (0,4,2,0), can the request be granted immediately?

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

	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	

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Process	Allocation A B C D	Max A B C D	Available A B C D	Need A B C D
P_0	2 0 0 1	4 2 1 2	3 3 2 1	2 2 1 1
P_1	3 1 2 1	5 2 5 2	5 3 2 2	2 1 3 1
P_2	2 1 0 3	2 3 1 6	6 6 3 4	0 2 1 3
P_3	1 3 1 2	1 4 2 4	7 10 6 6	0 1 1 2
P_4	1 4 3 2	3 6 6 5	10 11 8 7	2 2 3 3
	9 9 6 9		12, 12, 8, 10	

P_0, P_3, P_4, P_1, P_2

① Need matrix

② Is system is safe find the safe sequence

③ If request from @ P_1 arrives for (1, 1, 0, 0) can request be immediately granted.

④ P_4 (0, 0, 2, 0)

Process	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	2	0	0	1	4	2	1	2	2	2	2	1	2	2	1	1
P_1	4	2	2	1	5	2	5	2	4	2	2	2	1	0	3	1
P_2	2	1	0	3	2	3	1	6	5	5	3	4	0	2	1	3
P_3	1	3	1	2	1	4	2	4	6	9	6	6	0	1	1	2
P_4	1	4	3	2	3	6	6	5	10	11	8	7	2	2	3	3
									12	12	8	10				

Safe sequence : P_0, P_3, P_4, P_1, P_2

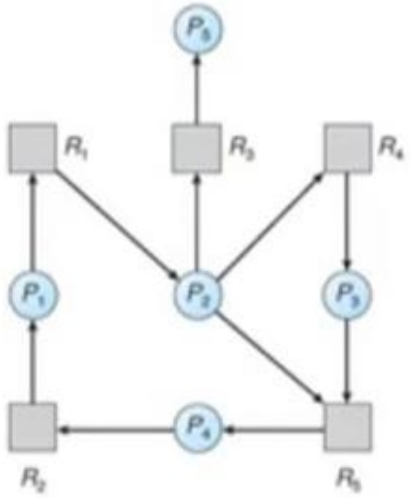
Deadlock Detection

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

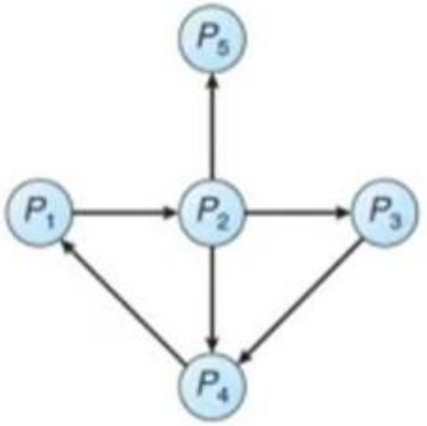
  - Wait-for graph - Single Instance of Each Resource Type
  - Banker's Algorithm - Several Instances of a Resource Type

Wait-for- Graph - Single Instance of Each Resource Type

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



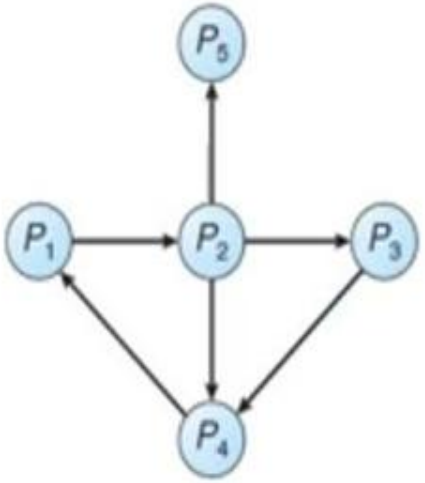
(a) Resource-allocation graph.



(b) Corresponding wait-for graph.

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.



Several Instances of a Resource Type – Bankers Algorithm

- **Available**. indicates the number of available resources of each type.
- **Allocation**. defines the number of resources of each type currently allocated to each process.
- **Request**. indicates the current request of each process.

Bankers Algorithm...

- **1.** Initialize ***Work*** = ***Available.*** For i = 0, 1, ..., $n-1$,
- if ***Allocation***$i \neq 0$, then ***Finish***[i] = ***false.***
- Otherwise, ***Finish***[i] = ***true.***
- **2.** Find an index i such that both
  - a. ***Finish***[i] == ***false***
  - b. ***Request***$i \leq$ ***Work***
  - If no such i exists, go to step 4.
- **3.** ***Work*** = ***Work*** + ***Allocation*** i
- ***Finish***[i] = ***true***
- Go to step 2.
- **4.** If ***Finish***[i] ==***false*** for some i, $0 \leq i < n$, then the system is in a deadlocked state.
- if ***Finish***[i] == ***false,*** then process Pi is deadlocked.

Example

- 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	

- Work = Available and Finish[i] = false
- for i = 0, 1, ..., n – 1.
- P0: 1. work = available
- Work = 0 0 0
- 2. Request <= work
- 0 0 0 <= 0 0 0 (true)
- 3. work = work + allocation
- = 0 0 0 + 0 1 0 = 0 1 0
- P1: work = available
- Work = 0 1 0
- 2. Request <= work
- 2 0 2 <= 0 1 0 (false)

	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	

F	F	F	F	F

- P2: 1. work = available
- Work = 0 1 0
- 2. Request <= work
- 0 0 0 <= 0 1 0 (true)
- 3. work = work + allocation
- = 0 1 0 + 3 0 3 = 3 1 3
- P3: 1. work = available
- Work = 3 1 3
- 2. Request <= work
- 1 0 0 <= 3 1 3 (true)
- 3. work = work + allocation
- = 3 1 3 + 2 1 1 = 5 2 4

	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	

F T	F	F	F	F
P_0	P_1	P_2	P_3	P_4

	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	

F	F	F	F	F

- P4: 1. work = available
- Work = 5 2 4
- 2. Request <= work
- 0 0 2 <= 5 2 4 (true)
- 3. work = work + allocation
- = 5 2 4 + 0 0 2 = 5 2 6
- P1: work = available
- Work = 5 2 6
- 2. Request <= work
- 2 0 2 <= 5 2 6 (true)
- 3. work = work + allocation
- = 5 2 6 + 2 0 0 = 7 2 6

Result

- The following Sequence will result in ***Finish[i] = true*** for all ***i***

❖RECOVERY FROM DEADLOCK

- Recovery can be handled by manually or automatically.
- There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

➢Process Termination

- Here all processes are terminated which is in the deadlock state. Two methods for the process termination methods are:

✓Abort all deadlocked processes: This method clearly will break the deadlock cycle, but at a great expense. If 5 processes are existed, these 5 processes must be terminated. So the used resources are wasted.

✓ Abort one process at a time until the deadlock cycle is eliminated:

This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

- Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Many factors may determine 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 type of resources the process has used (for example, whether the resources are simple to preempt)

4. How many more resources the process needs in order to complete

5. How many processes will need to be terminated?

6. Whether the process is interactive or batch

➢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. The three conditions are:

1. **Selecting a victim:** Which resources and which processes are to be preempted?

   we must determine the order of preemption to minimize cost.

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