

# **OPERATING SYSTEMS**

## **MODULE 3**

### **PROCESS SYNCHRONIZATION & DEADLOCK**



SHAMEELA SULAIMAN  
COLLEGE OF ENGINEERING, ADOOR




# SYLLABUS

- **Process synchronization-**
  - Race conditions
  - Critical section problem
  - Peterson's solution,
  - Synchronization hardware,
  - Mutex Locks,
  - Semaphores,
  - Monitors –
  - Synchronization problems -Producer Consumer, Dining Philosophers and Readers-Writers.
- **Deadlocks:**
  - Necessary conditions,
  - Resource allocation graphs,
  - Deadlock prevention, Deadlock avoidance – Banker's algorithms, Deadlock detection, Recovery from deadlock.



# 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,
- How concurrent or parallel execution can contribute to issues involving the integrity of data shared by several processes?



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

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

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



# Producer consumer with unbounded buffer

- Add an integer variable counter , initialized to 0.
- counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.
- Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently.
- Actually 5 buffers are full.

$T_0$ :	producer	execute	$register_1 = counter$	$\{register_1 = 5\}$
$T_1$ :	producer	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
$T_2$ :	consumer	execute	$register_2 = counter$	$\{register_2 = 5\}$
$T_3$ :	consumer	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
$T_4$ :	producer	execute	$counter = register_1$	$\{counter = 6\}$
$T_5$ :	consumer	execute	$counter = register_2$	$\{counter = 4\}$



# RACE CONDITION

- We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.
- A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- **To guard against the race condition above**, we need to ensure that only one process at a time can be manipulating the variable counter .
- To make such a guarantee, **we require that the processes be synchronized in some way**.



# CRITICAL SECTION PROBLEM


- Consider a system consisting of  $n$  processes  $\{P_0, P_1, \dots, P_{n-1}\}$ .
- Each process has a segment of code, called a **critical section**, in which the process may be **changing common variables, updating a table, writing a file**, and so on.
- The important feature of the system is that, **when one process is executing in its critical section, no other process is allowed to execute in its critical section.**
- That is, no two processes are executing in their critical sections at the same time.
- **The critical-section problem is to design a protocol that the processes can use to cooperate.**
- Each process must **request permission to enter its critical section**. The section of code implementing this request is the **entry section**.
- The critical section may be followed by an **exit section**.
- The remaining code is the **remainder section**.



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

1. **Mutual exclusion.** If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress.** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely
3. **Bounded waiting.** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.





do {

*entry section*

critical section

*exit section*

remainder section

} while (true);

**Figure 5.1** General structure of a typical process  $P_i$ .



# PETERSON'S SOLUTION

- A classic software-based solution to the critical-section problem known as Peterson's solution.
- It provides a good algorithmic description of solving the critical-section problem
- 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.




# PETERSON'S SOLUTION

- 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  $P_i$  is allowed to execute in its critical section.
- The **flag** array is used to indicate if a process is ready to enter its critical section.
- For example, if **flag[i]** is true, this value indicates that  $P_i$  is ready to enter its critical section



```
do {
```

```
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);
```

critical section

```
    flag[i] = false;
```

remainder section

```
} while (true);
```

**Figure 5.2** The structure of process  $P_i$  in Peterson's solution.



## CODE EXPLANATION

To enter the critical section, process  $P_i$  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 eventual value of `turn` determines which of the two processes is allowed to enter its critical section first.



## CODE EXPLANATION

- We note that each  $P_i$  enters its critical section only if either  $\text{flag}[j] == \text{false}$  or  $\text{turn} == i$
- Also note that, if both processes can be executing in their critical sections at the same time, then  $\text{flag}[0] == \text{flag}[1] == \text{true}$ .
- These two observations imply that  $P_0$  and  $P_1$  could not have successfully executed their while statements at about the same time, since the value of  $\text{turn}$  can be either 0 or 1 but cannot be both.
- Hence, one of the processes—say,  $P_j$ —must have successfully executed the while statement, whereas  $P_i$  had to execute at least one additional statement (“ $\text{turn} == j$ ”).
- However, at that time,  $\text{flag}[j] == \text{true}$  and  $\text{turn} == j$ , and this condition will persist as long as  $P_j$  is in its critical section; as a result, mutual exclusion is preserved.



## CODE EXPLANATION

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



# SYNCHRONIZATION HARDWARE


- TEST & SET LOCK
- COMPARE & SWAP





## test & set

- A hardware solution to the synchronization problem.
- A shared lock variable which can take either two values 0 (unlock) or 1 (lock)
- Before entering into the critical section , a process inquires about the lock.
- If it is locked, it keeps on waiting till it becomes free.
- If it is not locked, it takes the lock (set the lock) & executes the critical section.



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

**figure 5.3** The definition of the `test_and_set()` instruction.

- Atomic operation
- All happens as a single operation that will be uninterrupted.



## Process p1

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = false;  
  
    /* remainder section */  
} while (true);
```

**Figure 5.4** Mutual-exclusion implementation with `test_and_set()`.




# Compare and swap

- Operates on 3 operands
- Always returns the original value of the variable value.

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

**Figure 5.5** The definition of the `compare_and_swap()` instruction.




```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = 0;  
  
    /* remainder section */  
} while (true);
```

**Figure 5.6** Mutual-exclusion implementation with the `compare_and_swap()` instruction.



# MUTEX LOCKS

- Hardware based solutions to the critical section problem is complicated & inaccessible to application programmers.
- Software tools to solve the critical section problem - Mutex Lock.
- MUTual EXclusion.
- We use mutex locks to protect critical regions and thus prevents race conditions.
- A process must acquire the lock before entering the critical section.
- It releases the lock when it exits the critical section.
- `acquire()`- Acquires the lock
- `release()`- Releases the lock



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

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

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

- Call to either `acquire()` or `release()` must be performed automatically



# DISADVANTAGE

## DISADVANTAGE

- Busy waiting
- While a process is in 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 called a spinlock because the process spins while waiting for the lock to become available.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.

## ADVANTAGE

- No context switch is required when a process must wait on a lock & a context switch may take considerable time.
- (when locks are expected to be held for short times, spinlocks are useful)





# SEMAPHORE

- A semaphore  $S$  is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: `wait()` and `signal()` .
- The `wait()` operation was originally termed  $P$  (from the Dutch *proberen*, “to test”);
- `signal()` was originally called  $V$  (from *verhogen*, “to increment”).
- All modifications to the integer value of the semaphore in the `wait()` and `signal()` operations must be executed indivisibly.
- The definition of `wait()` & `signal()` is as follows:

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

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



# Counting Vs Binary Semaphore

- Operating systems often distinguish between counting and binary semaphores.
- The value of a counting semaphore can range over an unrestricted domain.
- The value of a binary semaphore can range only between 0 and 1.
- Thus, binary semaphores behave similarly to mutex locks.
- 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.



# Semaphore Implementation

- When a process executes the `wait()` operation and finds that the semaphore value is not positive, it must wait.
- **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

```

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

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

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

```



# CLASSICAL PROBLEMS OF SYNCHRONIZATION

- The Bounded Buffer Problem
- Readers Writers Problem
- Dining Philosophers Problem



# Bounded Buffer Problem

- In our problem, the producer and consumer processes share the following data structures:
- `int n;`
- `semaphore mutex = 1;`
- `semaphore empty = n;`
- `semaphore full = 0`
- The pool consists of `n` buffers, each capable of holding one item.
- The mutex semaphore provides mutual exclusion for accesses to the buffer pool
- and is initialized to the value 1.
- The empty and full semaphores count the number of empty and full buffers.
- The semaphore empty is initialized to the value `n` ; the semaphore full is initialized to the value 0.



# Producer

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

**Figure 5.9** The structure of the producer process.



# Consumer

```
do {
    wait(full);
    wait(mutex);

    . . .
    /* remove an item from buffer to next_consumed */
    . . .
    signal(mutex);
    signal(empty);

    . . .
    /* consume the item in next_consumed */
    . . .
} while (true);
```

**Figure 5.10** The structure of the consumer process.





# DEADLOCK



# DEADLOCK

- In a multiprogramming environment, several processes may compete for a finite number of resources.
- A process requests resources; if the resources are not available at that time, the process enters a waiting state.
- Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes.
- This situation is called a deadlock.
- A deadlocked state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.



# 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.
- 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.



# SYSTEM MODEL

- Under the normal mode of operation, a process may utilize a resource in only the following sequence:
- **Request.** The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
- **Use.** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
- **Release.** The process releases the resource.

The request and release of resources may be system calls



# NECESSARY CONDITIONS FOR DEADLOCK

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

- ❖ **MUTUAL EXCLUSION**
- ❖ **HOLD AND WAIT**
- ❖ **NO PREEMPTION**
- ❖ **CIRCULAR WAIT**

ALL 4 CONDITIONS MUST HOLD FOR A DEADLOCK TO OCCUR



## MUTUAL EXCLUSION

- At least one resource must be held in a nonsharable 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.

## HOLD AND WAIT

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



## NO PREEMPTION

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

## CIRCULAR WAIT

- A set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes must exist such that  $P_0$  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 resource held by  $P_n$ , and  $P_n$  is waiting for a resource held by  $P_0$ .



# RESOURCE ALLOCATION GRAPH

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. This graph consists of a set of vertices  $V$  and a set of edges  $E$ .

The set of vertices  $V$  is partitioned into two different types of nodes:  $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the active processes in the system, and  $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system.

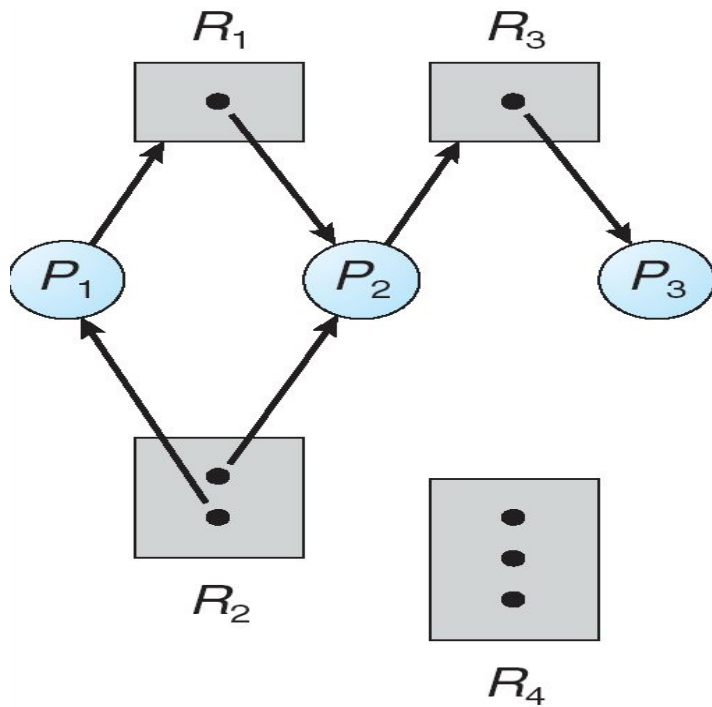
A directed edge from process  $P_i$  to resource type  $R_j$  is denoted by  $P_i \rightarrow R_j$ ; it signifies that process  $P_i$  has requested an instance of resource type  $R_j$  and is currently waiting for that resource.

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





# RAG

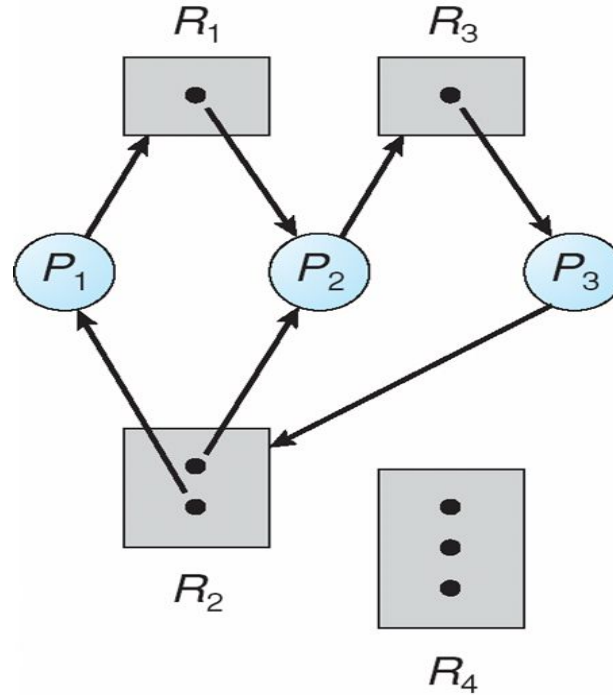




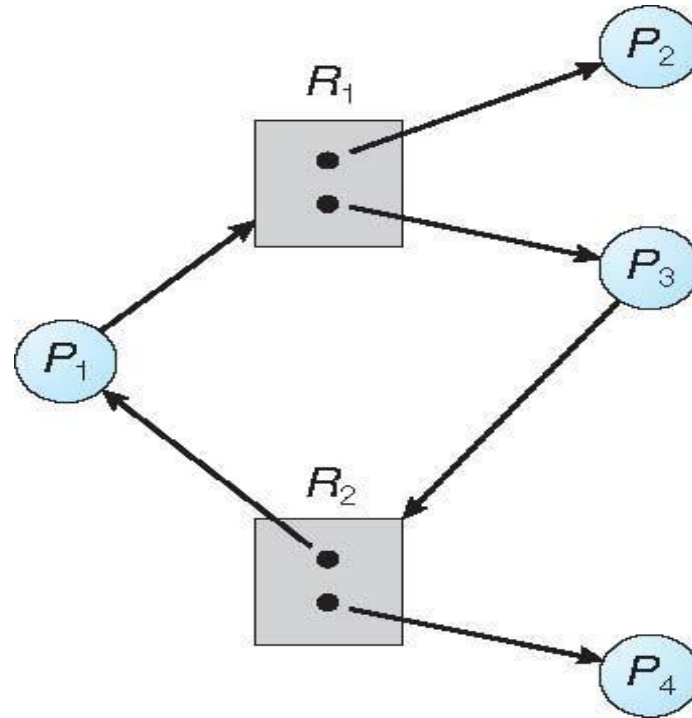
# RAG - Explained

- Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked.
- If the graph does contain a cycle, then a deadlock may exist.
- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.
- Each process involved in the cycle is deadlocked.
- In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.
- In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

# RESOURCE ALLOCATION GRAPH WITH DEADLOCK



# RAG WITH CYCLE BUT NO DEADLOCK





# METHODS FOR HANDLING DEADLOCKS

- Use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
- Allow the system to enter a deadlock state, detect it and recover.
- Ignore the problem altogether & pretend that deadlock never occur in the system.
- Deadlock Prevention:- Set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlock by constraining how requests for resources can be made.
- Deadlock Avoidance:- Requests that the OS be given in advance additional information concerning which resources a process will request & use during its lifetime.



# DEADLOCK PREVENTION



# DEADLOCK PREVENTION

- By ensuring that at least one of the necessary & sufficient conditions cannot hold, we can prevent the occurrence of a deadlock.
- MUTUAL EXCLUSION:- In this at least one resource must be nonsharable. Sharable resources (Eg:Read Only Files) do not require mutually exclusive access and thus cannot be involved in a deadlock. We cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable.(Mutex Locks).
- HOLD AND WAIT:- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that we can use requires each process to request and be allocated all its resources before it begins execution.



# DEADLOCK PREVENTION

- **NO PREEMPTION:-** 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:-** 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.





# DEADLOCK PREVENTION

## -Circular Wait

- let  $R = \{R_1, R_2, \dots, R_m\}$  be the set of resource types. We assign to each resource type a unique integer number.
- We define a one-to-one function  $F: R \rightarrow N$ , where  $N$  is the set of natural numbers.
- Each process can request resources only in an increasing order of enumeration. I.e., a process can initially request any number of instances of a resource type —say,  $R_i$ . After that, the process can request instances of resource type  $R_j$  if and only if  $F(R_j) > F(R_i)$ .



# DEADLOCK AVOIDANCE



# DEADLOCK AVOIDANCE

- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.



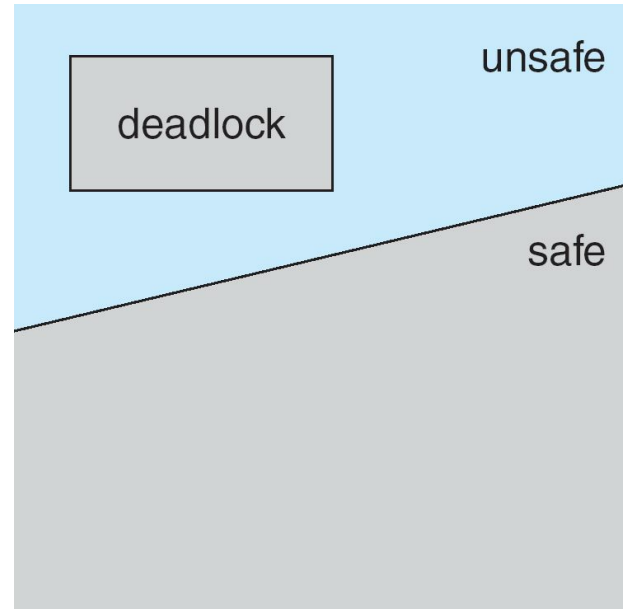
# SAFE STATE

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock.
- System is in safe state if there exists a sequence  $\langle P_1, P_2, \dots, P_n \rangle$  of ALL the processes in the systems such that for each  $P_i$ , the resources that  $P_i$  can still request can be satisfied by currently available resources + resources held by all the  $P_j$ , with  $j < i$
- If  $P_i$  resource needs are not immediately available, then  $P_i$  can wait until all  $P_j$  have finished
- When  $P_j$  is finished,  $P_i$  can obtain needed resources, execute, return allocated resources, and terminate
- When  $P_i$  terminates,  $P_{i+1}$  can obtain its needed resources, and so on



# SAFE STATE

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





# DEADLOCK AVOIDANCE ALGORITHM

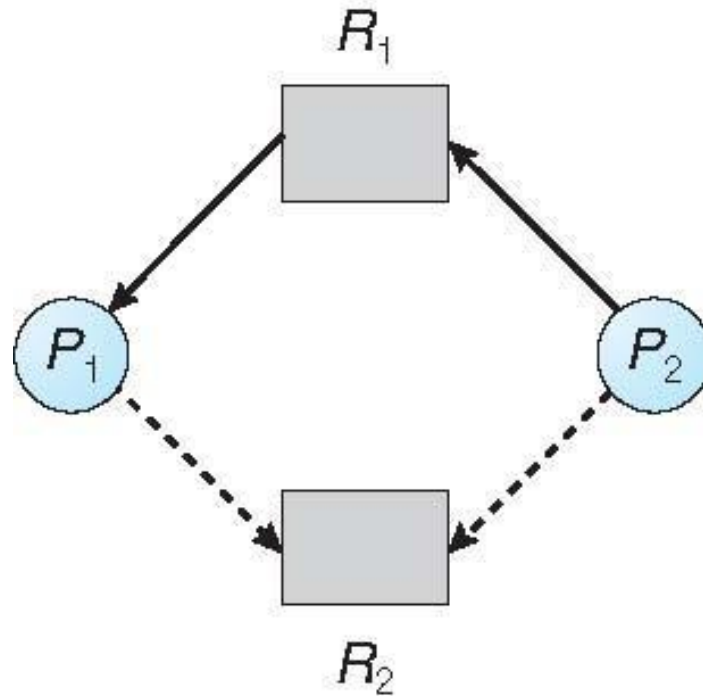
- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm



# RESOURCE ALLOCATION GRAPH SCHEME

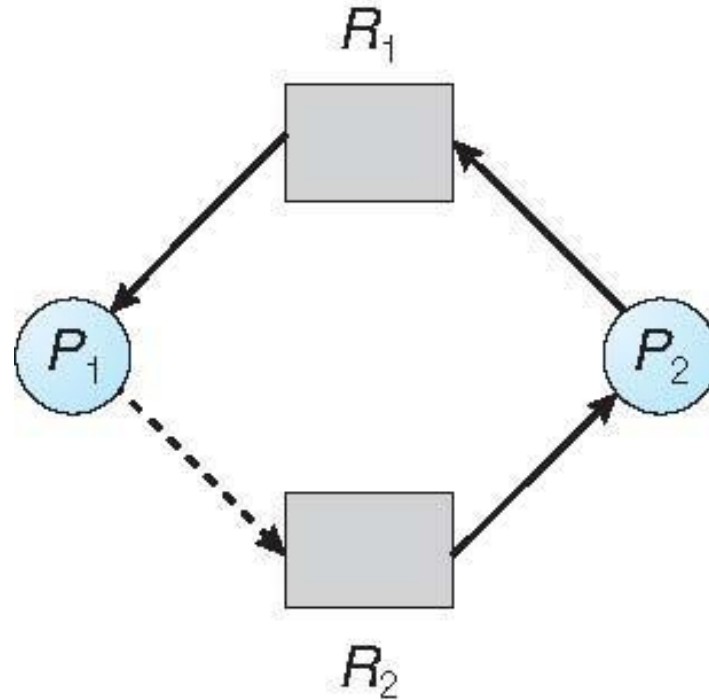
- Claim edge  $P_i \rightarrow R_j$  indicated that process  $P_j$  may request resource  $R_j$ ; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed a priori in the system

# RESOURCE ALLOCATION GRAPH





## Unsafe State In Resource-Allocation Graph





# RESOURCE ALLOCATION GRAPH ALGORITHM

- Suppose that process  $P_i$  requests a resource  $R_j$
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph



# BANKER'S ALGORITHM

- Deadlock Avoidance Method in case of multiple instances of resources.
- 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.



# Data Structures for the Banker's Algorithm

- Let  $n$  = number of processes, and  $m$  = number of resources types.
- **Available:** Vector of length  $m$ . If  $available[j] = k$ , there are  $k$  instances of resource type  $R_j$  available
- **Max:**  $n \times m$  matrix. If  $Max[i,j] = k$ , then process  $P_i$  may request at most  $k$  instances of resource type  $R_j$
- **Allocation:**  $n \times m$  matrix. If  $Allocation[i,j] = k$  then  $P_i$  is currently allocated  $k$  instances of  $R_j$
- **Need:**  $n \times m$  matrix. If  $Need[i,j] = k$ , then  $P_i$  may need  $k$  more instances of  $R_j$  to complete its task
- $Need[i,j] = Max[i,j] - Allocation[i,j]$



# SAFETY ALGORITHM

1. Let **Work** and **Finish** be vectors of length  $m$  and  $n$ , respectively. Initialize:  
    **Work = Available**  
    **Finish [i] = false for  $i = 0, 1, \dots, n-1$**
2. Find an  $i$  such that both:
  - (a) **Finish [i] = false**
  - (b) **Need<sub>i</sub> ≤ Work**If no such  $i$  exists, go to step 4
3. **Work = Work + Allocation<sub>i</sub>**  
    **Finish[i] = true**  
    go to step 2
4. If **Finish [i] == true** for all  $i$ , then the system is in a safe state

# Resource-Request Algorithm for Process $P_i$

- **Request** <sub>$i$</sub>  = request vector for process  $P_i$ . If **Request** <sub>$i$</sub> [ $j$ ] =  $k$  then process  $P_i$  wants  $k$  instances of resource type  $R_j$ 
  1. If **Request** <sub>$i$</sub>   $\leq$  **Need** <sub>$i$</sub> , go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
  2. If **Request** <sub>$i$</sub>   $\leq$  **Available**, go to step 3. Otherwise  $P_i$  must wait, since resources are not available
  3. Pretend to allocate requested resources to  $P_i$  by modifying the state as follows:
    - Available** = **Available** - **Request** <sub>$i$</sub> ;
    - Allocation** <sub>$i$</sub>  = **Allocation** <sub>$i$</sub>  + **Request** <sub>$i$</sub> ;
    - Need** <sub>$i$</sub>  = **Need** <sub>$i$</sub>  - **Request** <sub>$i$</sub> ;
- If safe  $\Rightarrow$  the resources are allocated to  $P_i$
- If unsafe  $\Rightarrow P_i$  must wait, and the old resource-allocation state is restored



# Example of Banker's Algorithm

- 5 processes  $P_0$  through  $P_4$ ;  
3 resource types:  
     $A$  (10 instances),  $B$  (5 instances), and  $C$  (7 instances)
- Snapshot at time  $T_0$ :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	$A\ B\ C$	$A\ B\ C$	$A\ B\ C$
$P_0$	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	



## Example (Contd...)

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

	<u>Need</u>
	<i>A B C</i>
$P_0$	7 4 3
$P_1$	1 2 2
$P_2$	6 0 0
$P_3$	0 1 1
$P_4$	4 3 1

- The system is in a safe state since the sequence  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  satisfies safety criteria



# Example: P1 Request (1,0,2)

- Check that Request  $\leq$  Available (that is,  $(1,0,2) \leq (3,3,2) \Rightarrow \text{true}$ )

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	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  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  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?



# DEADLOCK DETECTION



# Deadlock Detection

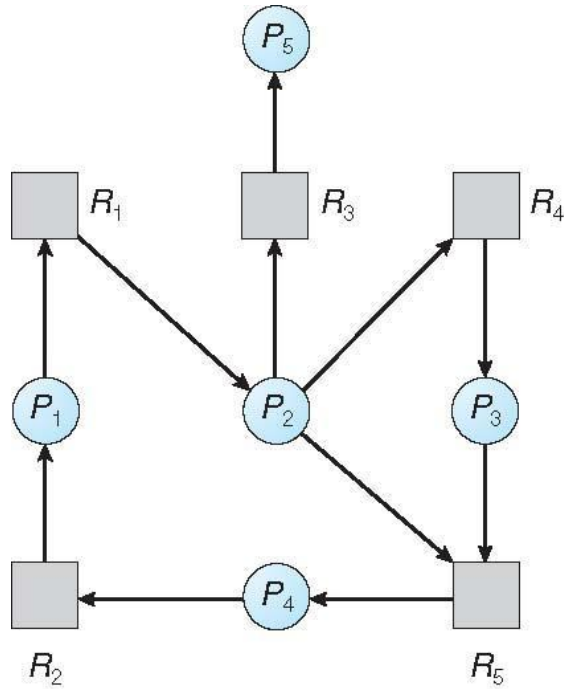
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme



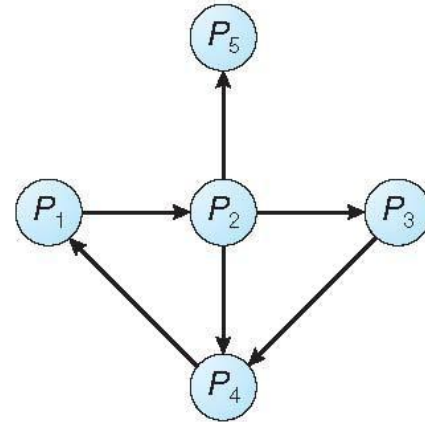
## Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$  if  $P_i$  is waiting for  $P_j$
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- 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

# Resource-Allocation Graph(a) and Wait-for Graph(b)



(a)



(b)



## Data structures

- **Available:** A vector of length  $m$  indicates the number of available resources of each type
- **Allocation:** An  $n \times m$  matrix defines the number of resources of each type currently allocated to each process
- **Request:** An  $n \times m$  matrix indicates the current request of each process. If **Request**  $[i][j] = k$ , then process  $P_i$  is requesting  $k$  more instances of resource type  $R_j$ .



# DETECTION ALGORITHM

1. Let **Work** and **Finish** be vectors of length **m** and **n**, respectively Initialize:
  - (a) **Work = Available**
  - (b) For  $i = 1, 2, \dots, n$ , if **Allocation<sub>i</sub>  $\neq 0$** , then **Finish[i] = false**; otherwise, **Finish[i] = true**
2. Find an index **i** such that both:
  - (a) **Finish[i] == false**
  - (b) **Request<sub>i</sub>  $\leq$  Work**

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



## DETECTION ALGORITHM

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  $O(m \times n^2)$  operations to detect whether the system is in deadlocked state**




# EXAMPLE OF DETECTION ALGORITHM

- Five processes  $P_0$  through  $P_4$ ; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time  $T_0$ :

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
$P_0$		0	1	0		0	0	0	0
$P_1$		2	0	0	2	0			
$P_2$				3	0	3	0	0	0
$P_3$		2	1	1	1	0	0		
$P_4$		0	0	2	0	0	2		

- Sequence  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$  will result in **Finish[i] = true** for all  $i$

- 
- $P_2$  requests an additional instance of type **C**

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

- State of system?
  - Can reclaim resources held by process  $P_0$ , but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes  $P_1$ ,  $P_2$ ,  $P_3$ , and  $P_4$



# DEADLOCK RECOVERY



## Deadlock Recovery: Process Termination

- To inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- To let the system recover from the deadlock automatically.
- For breaking a deadlock.
  - To abort one or more processes to break the circular wait.
  - To preempt some resources from one or more of the deadlocked processes



# Recovery from Deadlock: Process Termination

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 factors that affect which process is chosen**

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



# Deadlock Recovery: Resource Preemption

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.
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, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.
3. **Starvation.** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?



THANK YOU