MONITORS

wait()
CS
signal()

NEED OF MONITORS

- 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:

mutex = 1

```
signal(mutex);
...
critical section
...
wait(mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- 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.

Monitors

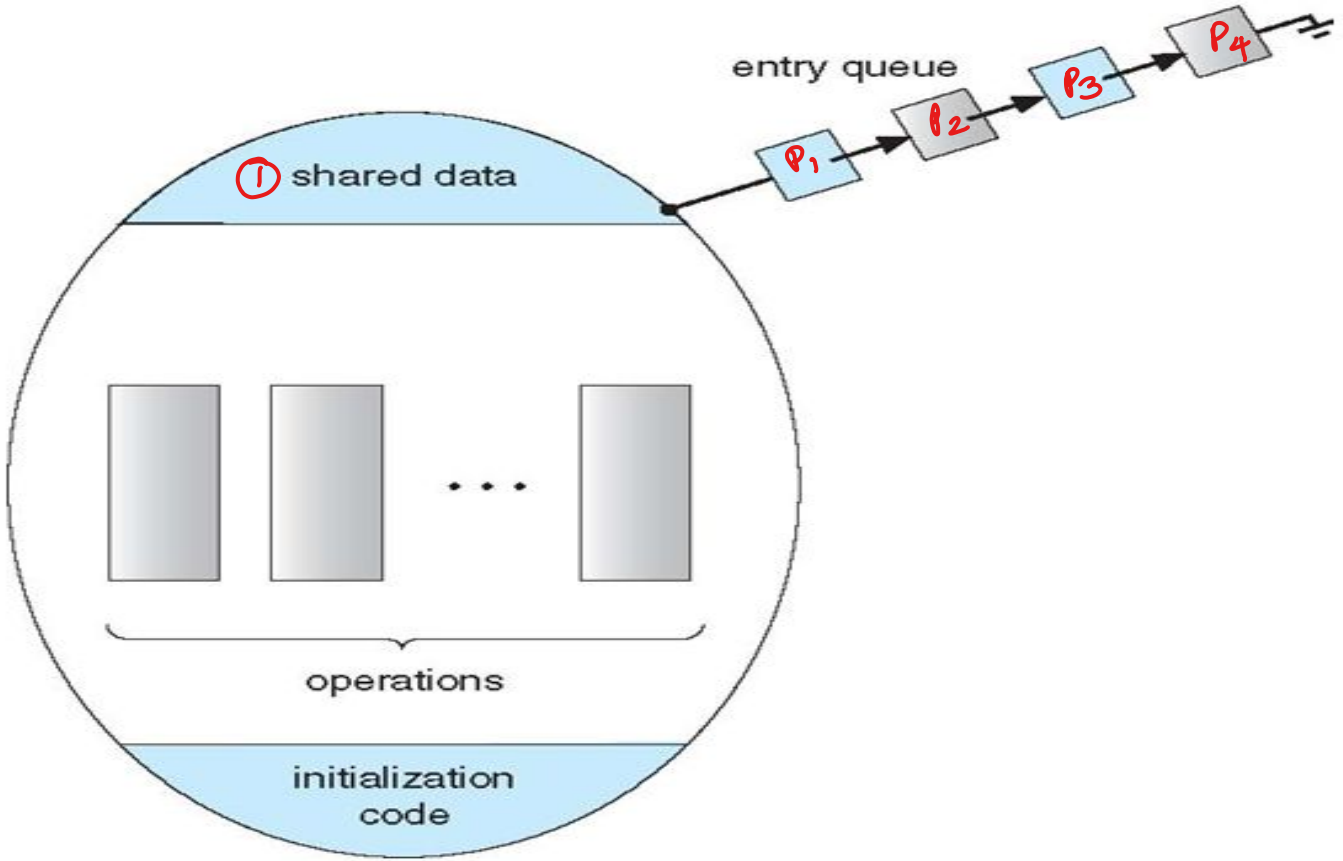
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

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

    procedure Pn (…) {……}

    Initialization code (…) { … }
}
```

Schematic view of a Monitor

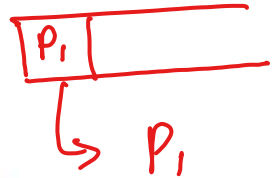


Soln:-

Condition Variables

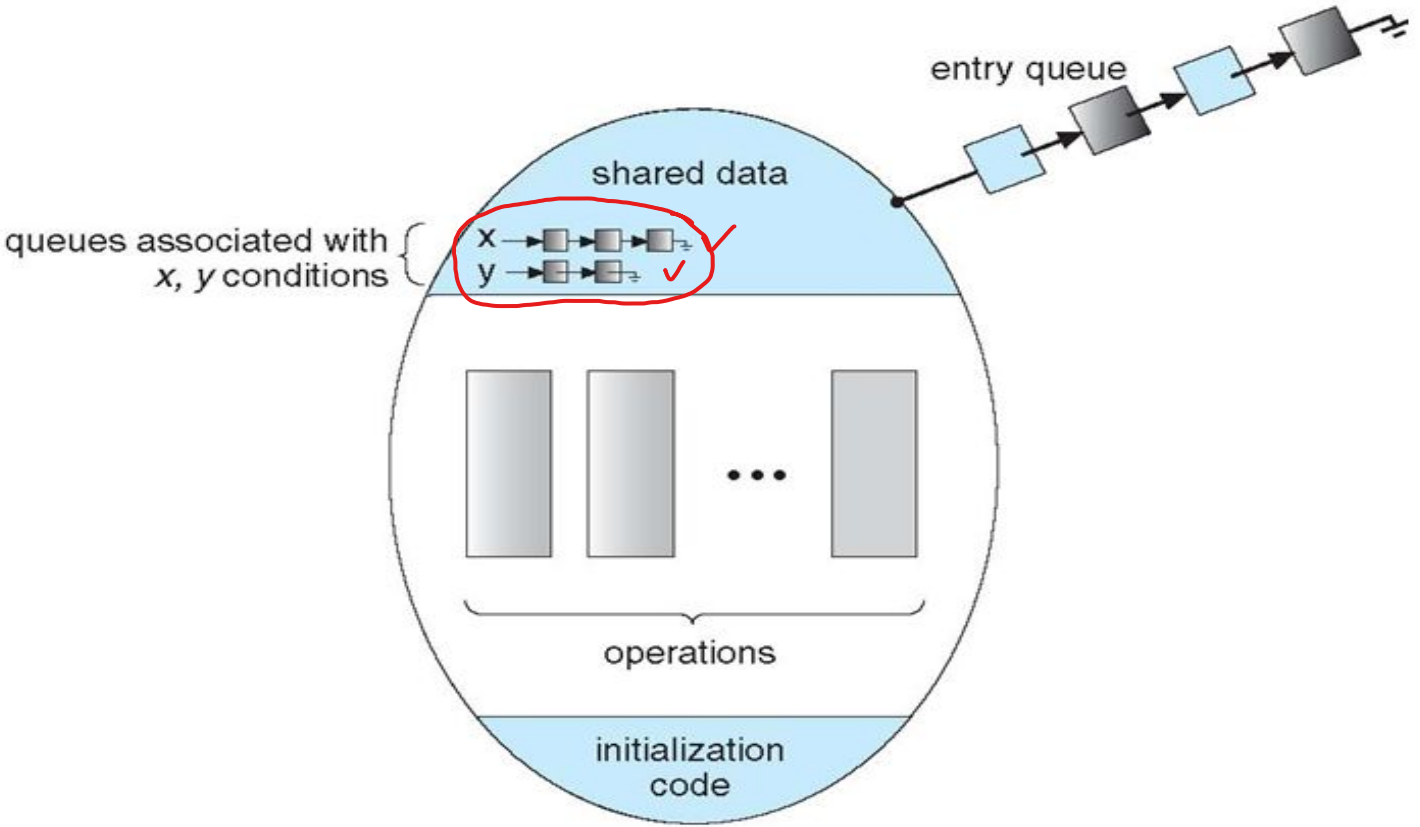
wait() s = s-1
signal() s = s+1

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
    - If no `x.wait()` on the variable, then it has no effect on the variable

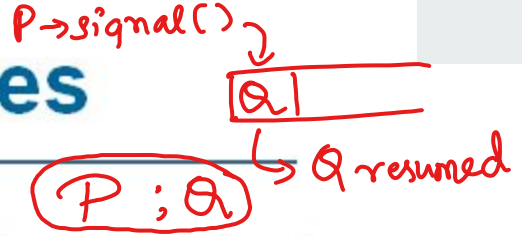


P1

Monitor with Condition Variables



Condition Variables Choices



- If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in paralel. If Q is resumed, then P must wait
- Options include
  - (1) **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition (P)
  - (2) **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition (Q)
  - Both have pros and cons – language implementer can decide

Monitor Solution to Dining Philosophers

⊗ iff both chopsticks are available

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
            state[i] = HUNGRY;
            test(i);
            if (state[i] != EATING) self[i].wait;
   }

   void putdown (int i) {
            state[i] = THINKING;
                    // test left and right neighbors
             test((i + 4) % 5);
             test((i + 1) % 5);
   }
```

⇐ if both/one chopstick is not available

check if both left & right philosophers are not eating

L P R

```
void test (int i) {
        if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING ;
            self[i].signal () ;
        }
}

    initialization_code() {
        for (int i = 0; i < 5; i++)
        state[i] = THINKING;
        }
}
```

(3) (1) (3) (2)

i+4
i+1

True

- Each philosopher *i* invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);

EAT

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Monitor Implementation Using Semaphores

- Variables

entering the monitor

to suspend the signalling process

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

↳ # of suspended process

- Each procedure ***F*** will be replaced by

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

- Mutual exclusion within a monitor is ensured

Monitor Implementation – Condition Variables

- For each condition variable **x**, we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

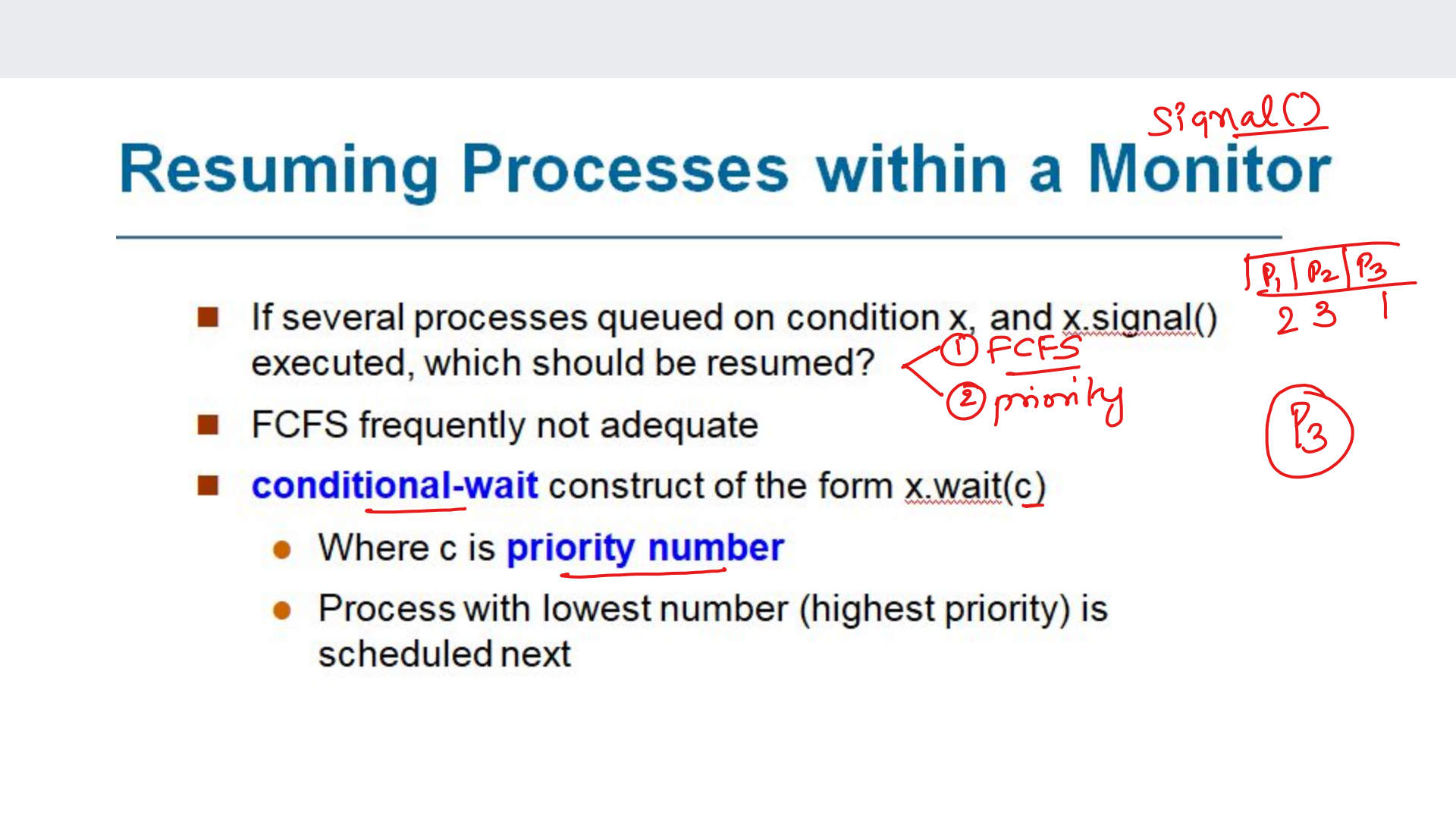
- The operation x.wait can be implemented as:

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

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

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next



Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

```
1  R.acquire(t);
       ...
2  access the resurce;
       ...

3  R.release;
```

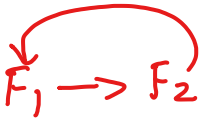
- Where R is an instance of type **ResourceAllocator**

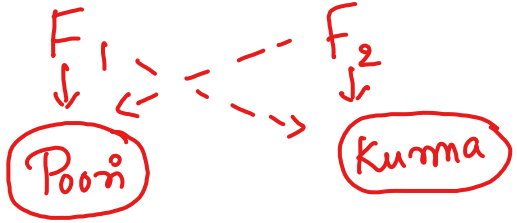

Thank You
my dear students

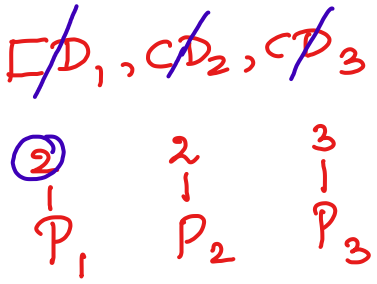
Chapter 7: Deadlocks

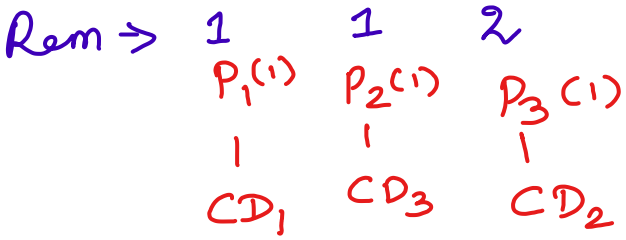
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

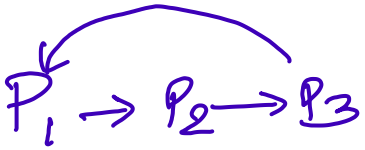
EXAMPLE


F1 -> F2


F1
F2
Poori
Kurma

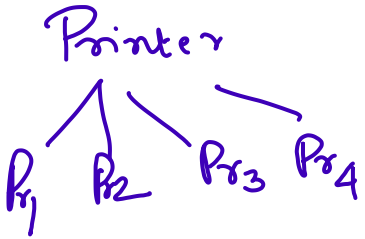

CD1, CD2, CD3
2
2
3
P1
P2
P3


Ram ->
1
1
2
P1(1)
P2(1)
P3(1)
CD1
CD3
CD2


P1 -> P2 -> P3

System Model

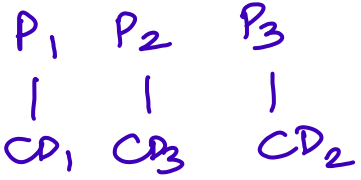
- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  - → *CPU cycles, memory space, I/O devices*
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  1. **request** < yes wait P_1
  2. **use**
  3. **release**



Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

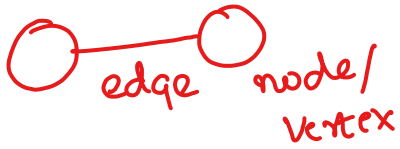
1. **Mutual exclusion**: only one process at a time can use a resource
2. **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
3. **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
4. **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.




Thank You
my dear students

Resource-Allocation Graph

- A set of vertices V and a set of edges E.



- V is partitioned into two types:
  1. $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  2. $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

R_1, R_2, R_3

$P_1, P_2 \ldots$

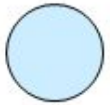
$P_1 \rightarrow R_1$

$R_1 \rightarrow P_1$

- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

- Process

Circle

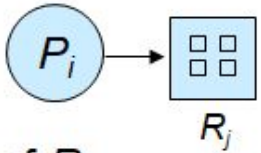


- Resource Type with 4 instances

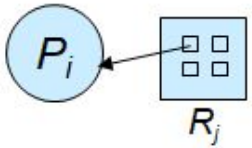
Rectangle

Printer

- P_i requests instance of R_j



- P_i is holding an instance of R_j



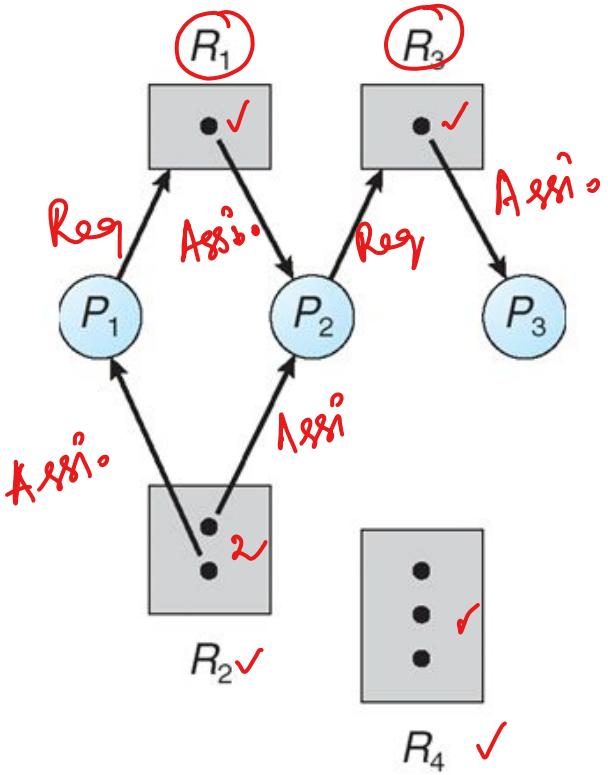
Printer

Pr_1 Pr_2 Pr_3 Pr_4

$P_1 \longrightarrow R_2$

$R_2 \longrightarrow P_1$

Example of a Resource Allocation Graph



Resources — R_1, R_2, R_3, R_4
1 2 1 3

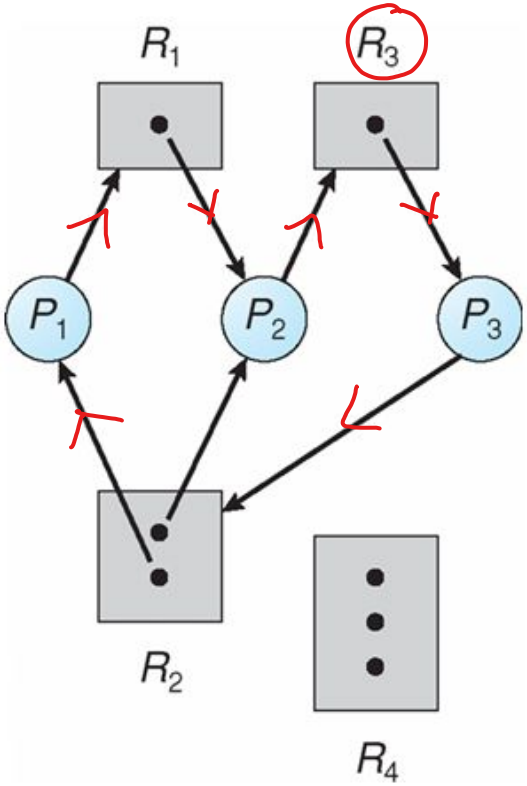
Process — P_1, P_2, P_3

Edge Request

(1) No cycle → no deadlock

(2) cycle < 1 instance resources ⊗ deadlock
multiple instance (may occur)

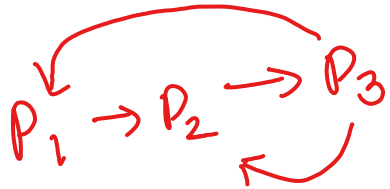
Resource Allocation Graph With A Deadlock



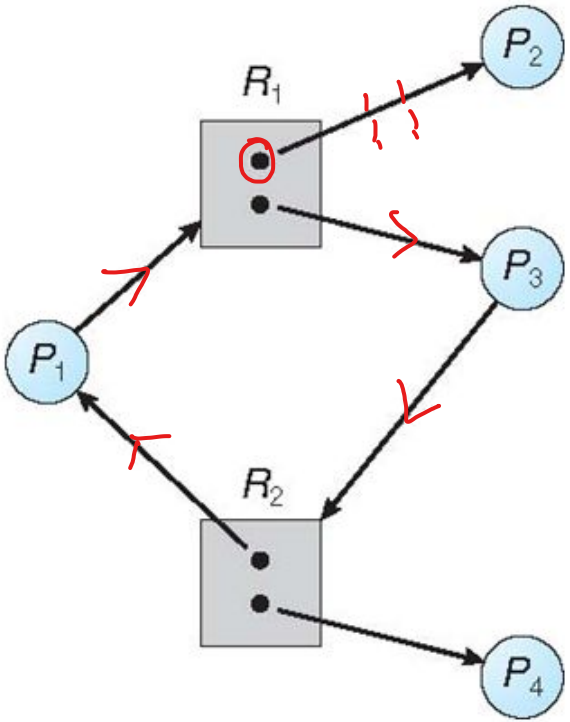
$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_1 \rightarrow R_1 (P_2) \rightarrow R_3 (P_3) \rightarrow R_2 (P_1, P_2)$

$\Downarrow$



Graph With A Cycle But No Deadlock



$P_1 - R_1 - P_3 - R_2 - P_1$

$P_1 - R_1 (P_2, P_3) - R_2 (P_1, P_4)$

Basic Facts

- 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


Thank You
my dear students

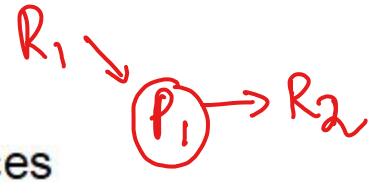
Methods for Handling Deadlocks

1. Ensure that the system will *never* enter a deadlock state:
   - i) Deadlock prevention
   - ii) Deadlock avoidence
2. Allow the system to enter a deadlock state and then recover
3. Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

1 Deadlock Prevention

4 — mut ex
— hold & wait
— no preemption
— circular wait
} 1 Break

Restrain the ways request can be made



1. **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

2. **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
   - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
   - Low resource utilization; starvation possible

1) Before start all resources

2) 

$R_1 \rightarrow P_1$

- **No Preemption –**

(3)

  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

$R_2 R_3$ free → give to P_1

x free → free

R_1

P_1 | R_2 | R_3 | R_1 |

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

(4)

$R_1 \rightarrow$ Tape → 1

$R_2 \rightarrow$ CD → 5

$R_3 \rightarrow$ Printer → 17

$\Rightarrow$ $P_1 \rightarrow$ prin + tape (tape ; printer)

$P_1 \rightarrow$ tape ; prin

$P_2 \rightarrow$ prin ; tape


Thank You
my dear students

(13) Theory + Numerical

max

P1
P2
P3

(2) Deadlock Avoidance

Requires that the system has some additional ***a priori*** information available

- 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
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ 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 $j < I$
- That is:
  - 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

$R_1 \rightarrow 10$

$P_1 - 2$
$P_2 - 4$
$P_3 - 3$

$<P_1\ P_2\ P_3>$

$P_3\ P_2\ P_1$

$P_3\ P_1\ P_2$

$P_i \rightarrow P_j$

Basic Facts

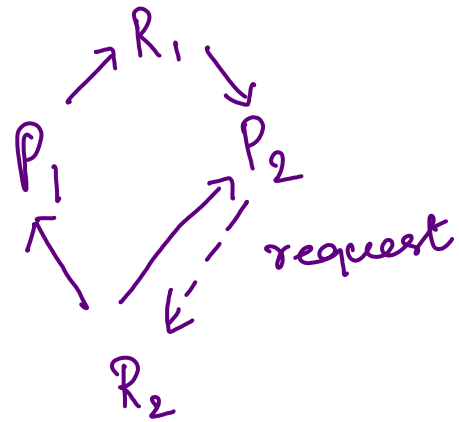
- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

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

- 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



(X) into cycle


Thank You
my dear students

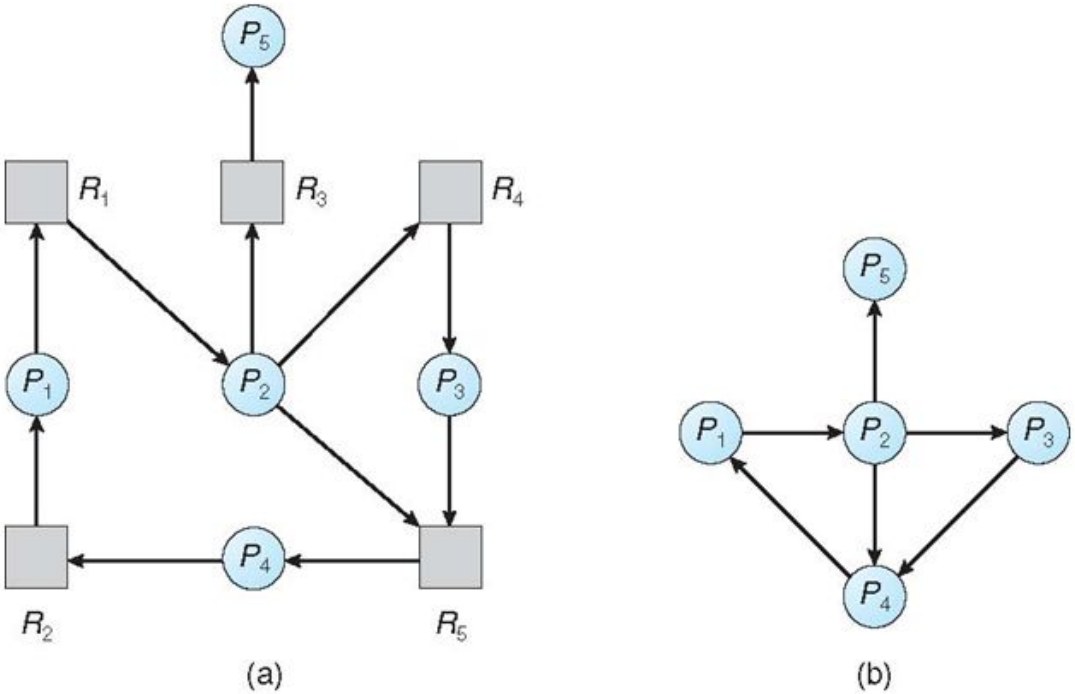
Deadlock Detection

- 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 and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph

Detection Algorithm

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

   If no such i exists, go to step 4
3. $Work = Work + Allocation_i$
   Finish*[*i*] = *true
   go to step 2

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

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:

	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	

- Sequence <P_0, P_2, P_3, P_1, P_4> 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

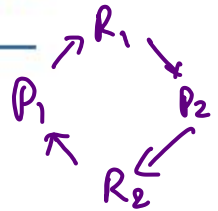
- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.


Thank You
my dear students

Recovery from Deadlock: Process Termination



- Abort all deadlocked processes ✓
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

1. **Selecting a victim** – minimize cost

2. **Rollback** – return to some safe state, restart process for that state

3. **Starvation** – same process may always be picked as victim, include number of rollback in cost factor