Contents

Chapter 7 Deadlocks

Deadlocks

- A set of process is in a *deadlock* state when every process in the set is waiting for an event that can be caused by only another process in the set.
- A System Model
  - Competing processes – distributed?
  - Resources:
    - Physical Resources, e.g., CPU, printers, memory, etc.
    - Logical Resources, e.g., files, semaphores, etc.

Deadlocks

- A Normal Sequence
  - Request: Granted or Rejected
  - Use
  - Release
- Remarks
  - No request should exceed the system capacity!
  - Deadlock can involve different resource types!
    - Several instances of the same type!

Deadlocks

```
void *do_work_one(void *param) {
     pthread_mutex_lock(&first_mutex);
      pthread_mutex_lock(&second_mutex);
     /* Do some work */
      pthread_mutex_unlock(&second_mutex);
     pthread_mutex_unlock(&first_mutex);
     pthread_exit(0); }
void *do_work_two(void *param) {
     pthread_mutex_lock(&second_mutex);
      pthread_mutex_lock(&first_mutex);
     /* Do some work */
     pthread_mutex_unlock(&first_mutex);
      pthread_mutex_unlock(&second_mutex);
     pthread_exit(0); }
…
pthread_mutex_init(&first_mutex, NULL);
pthread_mutex_init(&second_mutex, NULL);
```

Deadlock Characterization

- Necessary Conditions

(deadlock → conditions or ¬ conditions → ¬ deadlock)

  - Mutual Exclusion – At least one resource must be held in a non-sharable mode!
  - Hold and Wait – Pi is holding at least one resource and waiting to acquire additional resources that are currently held by other processes!

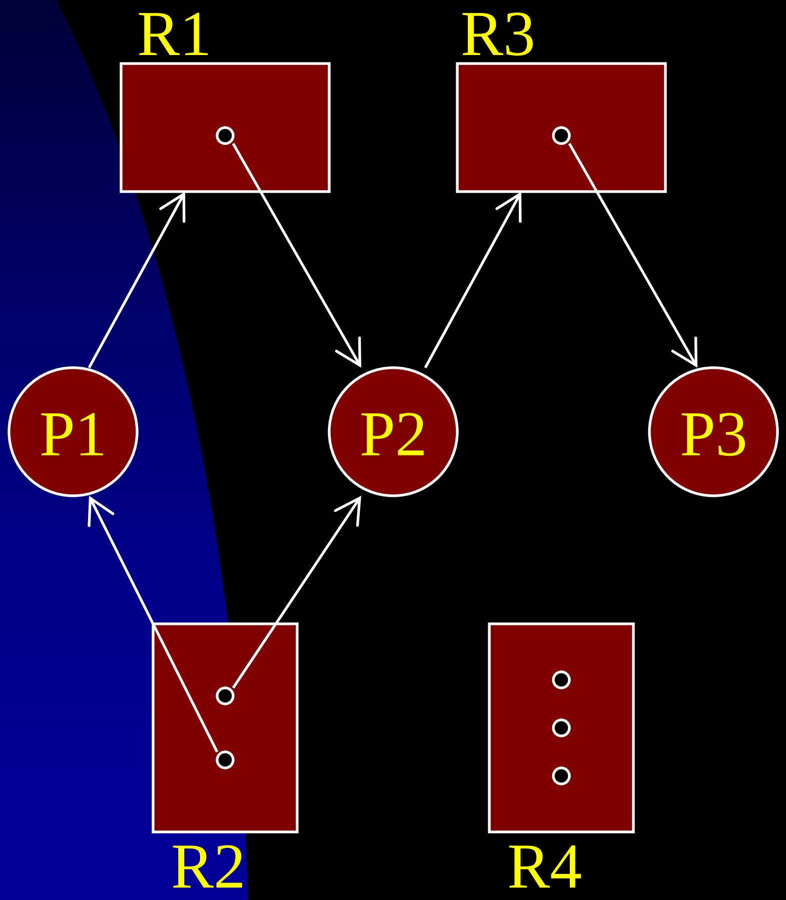
Deadlock Characterization

- No Preemption – Resources are nonpreemptible!
- Circular Wait – There exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting process such that $P_0 \xrightarrow[\text{wait}]{} P_1$, $P_1 \xrightarrow[\text{wait}]{} P_2$, …, $P_{n-1} \xrightarrow[\text{wait}]{} P_n$, and $P_n \xrightarrow[\text{wait}]{} P_0$.

- Remark:
  - Condition 4 implies Condition 2.
  - The four conditions are not completely independent!

Resource Allocation Graph

System Resource-Allocation Graph



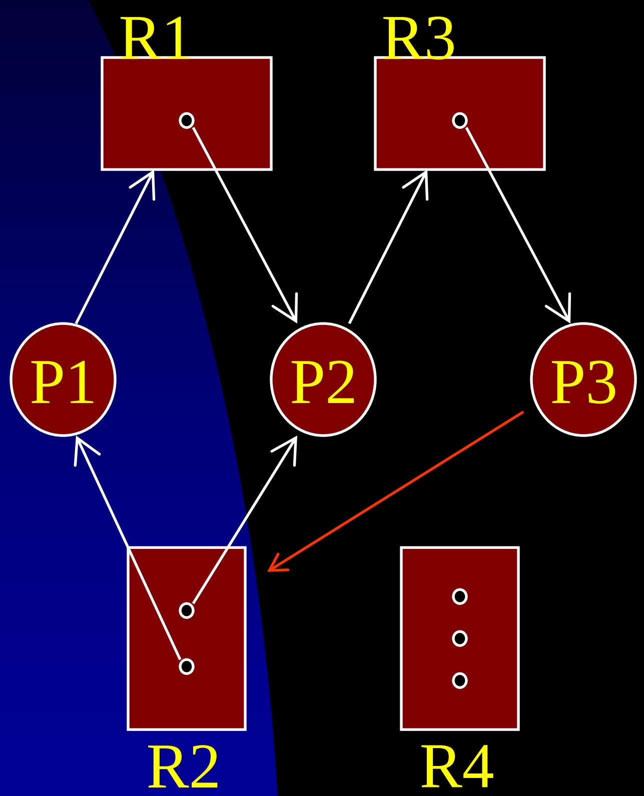
Vertices

- Processes:
  {P1,…, Pn}
- Resource Type :
  {R1,…, Rm}

Edges

- Request Edge:
  Pi → Rj
- Assignment Edge:
  Ri → Pj

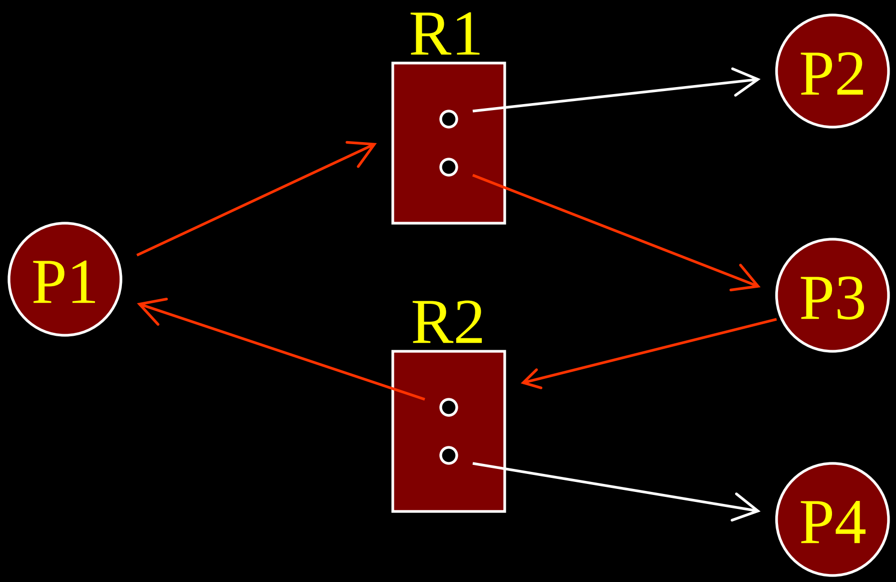
Resource Allocation Graph



- Example
  - No-Deadlock
    - Vertices
      - P = { P1, P2, P3 }
      - R = { R1, R2, R3, R4 }
    - Edges
      - E = { P1→R1, P2→R3, R1→P2, R2→P2, R2→P1, R3→P3 }
  - Resources
    - R1:1, R2:2, R3:1, R4:3
  - → results in a deadlock.

Resource Allocation Graph

- Observation
  - The existence of a cycle
    - One Instance per Resource Type → Yes!!
    - Otherwise → Only A Necessary Condition!!



Methods for Handling Deadlocks

- Solutions:
  - Make sure that the system never enters a deadlock state!
    - Deadlock Prevention: Fail at least one of the necessary conditions
    - Deadlock Avoidance: Processes provide information regarding their resource usage. Make sure that the system always stays at a “safe” state!

Methods for Handling Deadlocks

- Do recovery if the system is deadlocked.
  - Deadlock Detection
  - Recovery
- Ignore the possibility of deadlock occurrences!
  - Restart the system “manually” if the system “seems” to be deadlocked or stops functioning.
  - Note that the system may be “frozen” temporarily!

Deadlock Prevention

- Observation:
  - Try to fail anyone of the necessary condition!

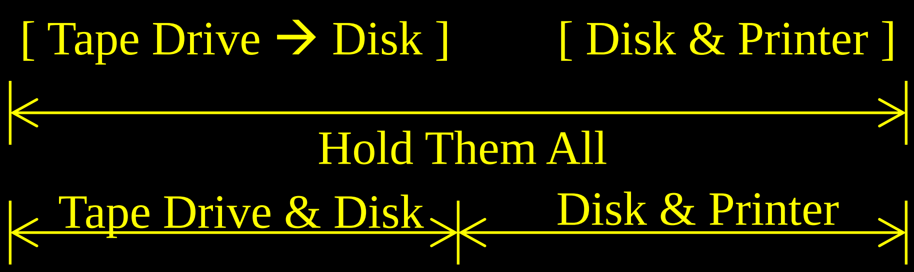
    $\because \neg (\wedge \text{ i-th condition}) \rightarrow \neg \text{ deadlock}$

- Mutual Exclusion

  ?? Some resources, such as a printer, are intrinsically non-sharable??

Deadlock Prevention

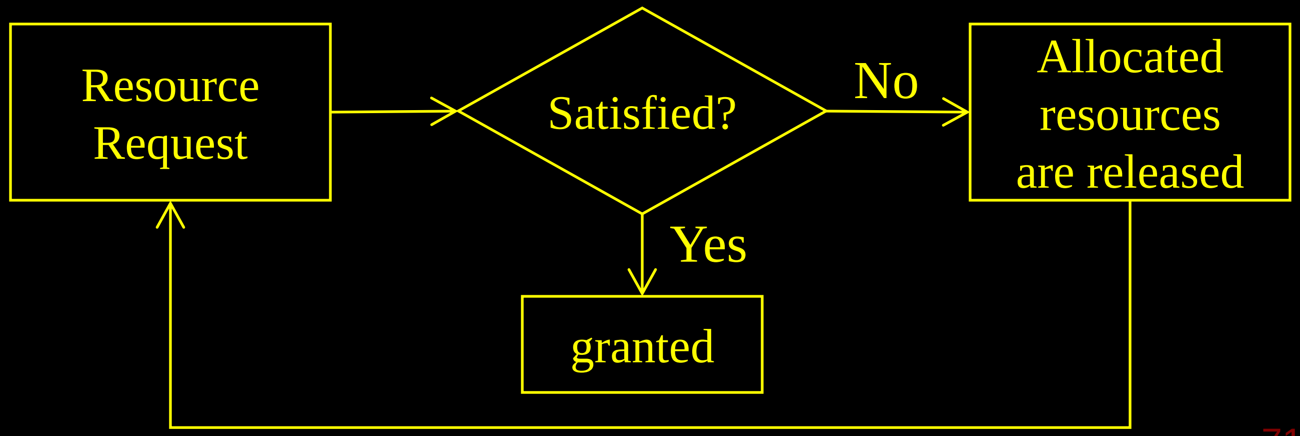
- Hold and Wait
  - Acquire all needed resources before its execution.
  - Release allocated resources before request additional resources!



  - Disadvantage:
    - Low Resource Utilization
    - Starvation

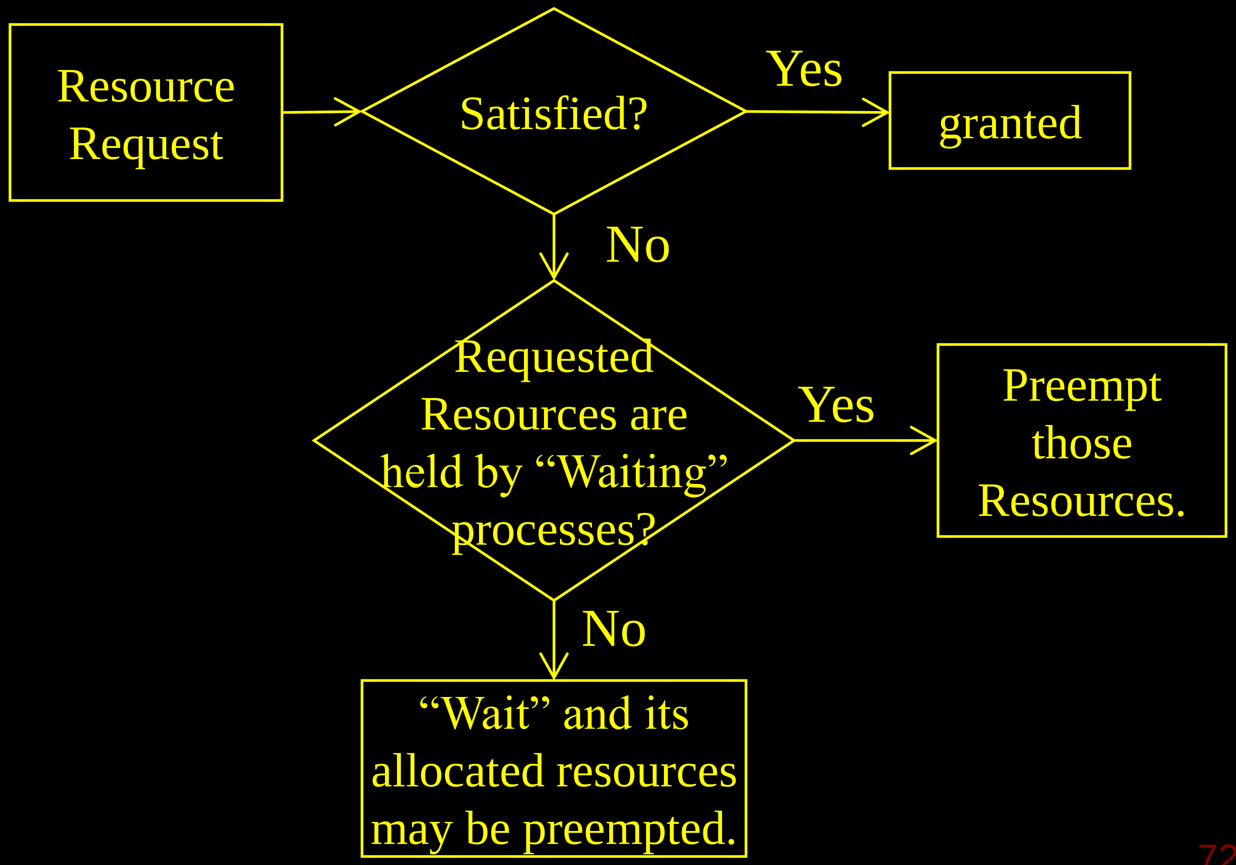
Deadlock Prevention

- No Preemption
  - Resource preemption causes the release of resources.
  - Related protocols are only applied to resources whose states can be saved and restored, e.g., CPU register & memory space, instead of printers or tape drives.
  - Approach 1:



Deadlock Prevention

- Approach 2



Deadlock Prevention

- Circular Wait

  A resource-ordering approach:

  $F : R \rightarrow N$

  Resource requests must be made in an increasing order of enumeration.

  - Type 1 – strictly increasing order of resource requests.
    - Initially, order any # of instances of Ri
    - Following requests of any # of instances of Rj must satisfy F(Rj) > F(Ri), and so on.

    * A single request must be issued for all needed instances of the same resources.

Deadlock Prevention

- Type 2
  - Processes must release all Ri's when they request any instance of Rj if $F(Ri) \geq F(Rj)$
- $F : R \rightarrow N$ must be defined according to the normal order of resource usages in a system, e.g.,

$$\left.\begin{array}{ll} F(\text{tape drive}) & = 1 \\ F(\text{disk drive}) & = 5 \\ F(\text{printer}) & = 12 \end{array}\right\} \text{?? feasible ??}$$

Deadlock Avoidance

- Motivation:
  - Deadlock-prevention algorithms can cause low device utilization and reduce system throughput!

→ Acquire additional information about how resources are to be requested and have better resource allocation!

  - Processes declare their maximum number of resources of each type that it may need.

Deadlock Avoidance

- A Simple Model
  - A resource-allocation state

    <# of available resources,

    # of allocated resources,

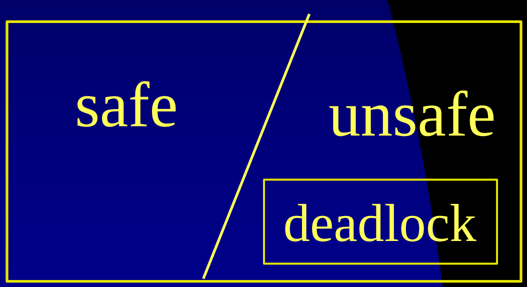
    max demands of processes>
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state and make sure that it is safe.
  - e.g., the system never satisfies the circular-wait condition.

Deadlock Avoidance

- Safe Sequence
  - A sequence of processes <P1, P2, …, Pn> is a safe sequence if

$$\forall Pi, need(Pi) \leq Available + \sum_{j<i} Allocated(Pj)$$

- Safe State
  - The existence of a safe sequence
  - Unsafe



Deadlocks are avoided if the system can allocate resources to each process up to its maximum request in some order. If so, the system is in a safe state!

Deadlock Avoidance

- Example:

	Max Needs	Allocated	Available
P0	10	5	3
P1	4	2	
P2	9	2	

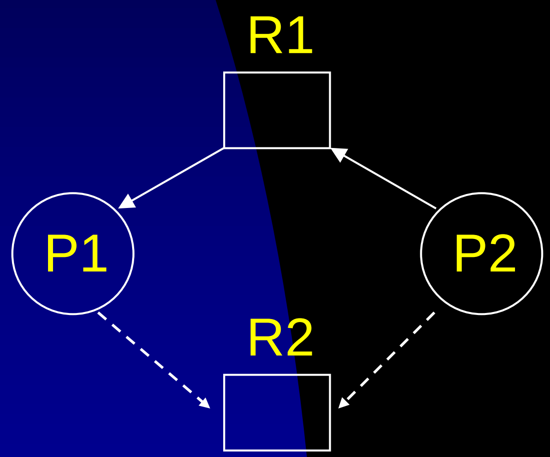
- The existence of a safe sequence <P1, P0, P2>.
- If P2 got one more, the system state is unsafe.

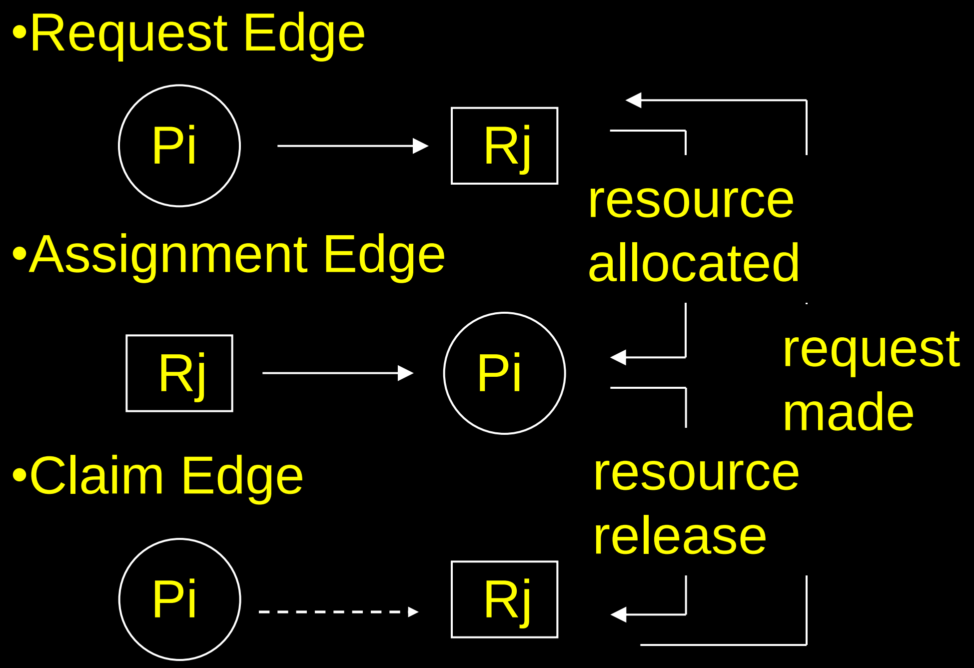
  ((P0, 5), (P1, 2), (P2, 3), (available, 2))

How to ensure that the system will always remain in a safe state?

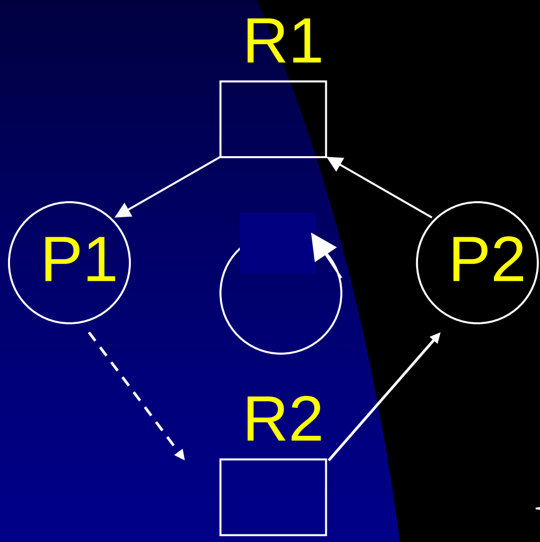
Deadlock Avoidance – Resource-Allocation Graph Algorithm

- One Instance per Resource Type





Deadlock Avoidance – Resource-Allocation Graph Algorithm



A cycle is detected!
➔ The system state is unsafe!

- R2 was requested & granted!

Safe state: no cycle
Unsafe state: otherwise

Cycle detection can be done in $O(n^2)$

Deadlock Avoidance – Banker's Algorithm

n: # of processes,
m: # of resource types

- Available [m]
  - If Available [i] = k, there are k instances of resource type Ri available.
- Max [n,m]
  - If Max [i,j] = k, process Pi may request at most k instances of resource type Rj.
- Allocation [n,m]
  - If Allocation [i,j] = k, process Pi is currently allocated k instances of resource type Rj.
- Need [n,m]
  - If Need [i,j] = k, process Pi may need k more instances of resource type Rj.

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

Deadlock Avoidance – Banker's Algorithm

n: # of processes,
m: # of resource types

- Safety Algorithm – A state is safe??
  1. Work := Available & Finish [i] := F, $1 \leqq i \leqq n$
  2. Find an i such that both
     1. Finish [i] =F
     2. Need[i] $\leqq$ Work

     If no such i exist, **then goto** Step4
  3. Work := Work + Allocation[i]
     Finish [i] := T; **Goto** Step2
  4. **If** Finish [i] = T for all *i*, **then** the system is in a safe state.

  Where Allocation[i] and Need[i] are the *i*-th row of Allocation and Need, respectively, and
  $X \leqq Y$ if $X[i] \leqq Y[i]$ for all i,
  $X < Y$ if $X \leqq Y$ and $Y \neq X$

Deadlock Avoidance – Banker's Algorithm

- Resource-Request Algorithm

Request$_i$ [*j*] =k: P$_i$ requests k instance of resource type Rj

1. If Request$_i$ $\leqq$ Need$_i$, then Goto Step2; otherwise, Trap
2. If Request$_i$ $\leqq$ Available, then Goto Step3; otherwise, Pi must wait.
3. Have the system pretend to have allocated resources to process P$_i$ by setting

   Available := Available – Request$_i$;

   Allocation$_i$:= Allocation$_i$ + Request$_i$;

   Need$_i$:= Need$_i$ – Request$_i$;

   Execute "Safety Algorithm". If the system state is safe, the request is granted; otherwise, Pi must wait, and the old resource-allocation state is restored!

Deadlock Avoidance

- An Example

	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2			
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

  - A safe state

    ∵ <P1,P3,P4,P2,P0> is a safe sequence.

Deadlock Avoidance

Let P1 make a request Request*i* = (1,0,2)
Request$_i$ ≦ Available (i.e., (1,0,2) ≦ (3,3,2))

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	2	3	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

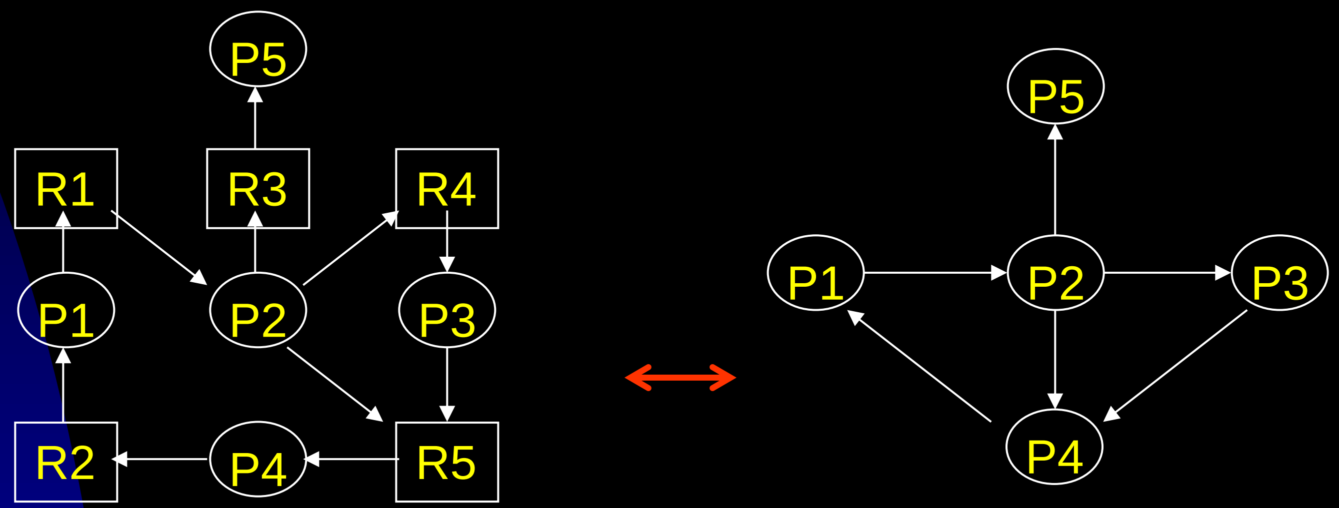
→ Safe ∵ <P1,P3,P4,P0,P2> is a safe sequence!

- If Request4 = (3,3,0) is asked later, it must be rejected.
- Request0 = (0,2,0) must be rejected because it results in an unsafe state.

Deadlock Detection

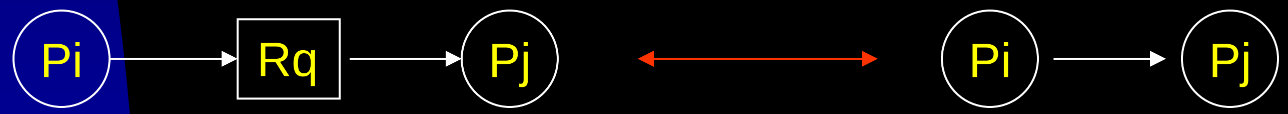
- Motivation:
  - Have high resource utilization and “maybe” a lower possibility of deadlock occurrence.
- Overheads:
  - Cost of information maintenance
  - Cost in executing a detection algorithm
  - Potential loss inherent from a deadlock recovery

Deadlock Detection – Single Instance per Resource Type



A Resource-Allocation Graph

A Wait-For Graph



- Detect an cycle in $O(n^2)$.
- The system needs to maintain the wait-for graph

Deadlock Detection – Multiple Instance per Resource Type

- Data Structures
  - Available[1..m]: # of available resource instances
  - Allocation[1..n, 1..m]: current resource allocation to each process
  - Request[1..n, 1..m]: the current request of each process
    - If Request[i,j] = k, Pi requests k more instances of resource type Rj

n: # of processes, *m*: # of resource types

Deadlock Detection – Multiple Instance per Resource Type

Complexity = $O(m * n^2)$

1. Work := Available. For i = 1, 2, …, n, **if** Allocation[i] $\neq$ 0, **then** Finish[i] = F; **otherwise,** Finish[i] =T.
2. Find an i such that both
   a. Finish[i] = F
   b. Request[i] $\leqq$ Work
   If no such i, **Goto** Step 4
3. Work := Work + Allocation[i]
   Finish[i] := T
   Goto Step 2
4. **If** Finish[i] = F for some i, **then** the system is in a deadlock state. **If** Finish[i] = F, then process Pi is deadlocked.

Deadlock Detection – Multiple Instances per Resource Type

- An Example

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	2	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

➔ Find a sequence <P0, P2, P3, P1, P4> such that Finish[i] = T for all i.

If Request2 = (0,0,1) is issued, then P1, P2, P3, and P4 are deadlocked.

Deadlock Detection – Algorithm Usage

- When should we invoke the detection algorithm?
  - How often is a deadlock likely to occur?
  - How many processes will be affected by a deadlock?

Every rejected request

overheads

$+ \longleftrightarrow -$

processes affected

$- \longleftrightarrow +$

∞

- Time for Deadlock Detection?
  - CPU Threshold? Detection Frequency? …

Deadlock Recovery

- Whose responsibility to deal with deadlocks?
  - Operator deals with the deadlock manually.
  - The system recover from the deadlock automatically.
- Possible Solutions
  - Abort one or more processes to break the circular wait.
  - Preempt some resources from one or more deadlocked processes.

Deadlock Recovery – Process Termination

- Process Termination
  - Abort all deadlocked processes!
    - Simple but costly!
  - Abort one process at a time until the deadlock cycle is broken!
    - Overheads for running the detection again and again.
    - The difficulty in selecting a victim!

But, can we abort any process?
Should we compensate any
damage caused by aborting?

Deadlock Recovery – Process Termination

- What should be considered in choosing a victim?
  - Process priority
  - The CPU time consumed and to be consumed by a process.
  - The numbers and types of resources used and needed by a process
  - Process's characteristics such as "interactive or batch"
  - The number of processes needed to be aborted.

Deadlock Recovery – Resource Preemption

- Goal: Preempt some resources from processes and give them to other processes until the deadlock cycle is broken!
- Issues
  - Selecting a victim:
    - It must be cost-effective!
  - Roll-Back
    - How far should we roll back a process whose resources were preempted?
  - Starvation
    - Will we keep picking up the same process as a victim?
    - How to control the # of rollbacks per process efficiently?

Deadlock Recovery – Combined Approaches

- Partition resources into classes that are hierarchically ordered.

  $\Rightarrow$ No deadlock involves more than one class

  - Handle deadlocks in each class independently

Deadlock Recovery – Combined Approaches

Examples:

- Internal Resources: Resources used by the system, e.g., PCB
  - $\rightarrow$ Prevention through resource ordering
- Central Memory: User Memory
  - $\rightarrow$ Prevention through resource preemption
- Job Resources: Assignable devices and files
  - $\rightarrow$ Avoidance $\because$ This info may be obtained!
- Swappable Space: Space for each user process on the backing store
  - $\rightarrow$ Pre-allocation $\because$ the maximum need is known!