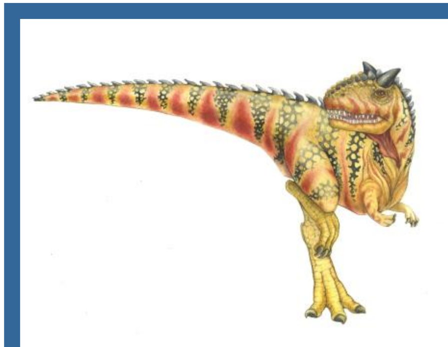
Xi'an Jiaotong-Liverpool University
西交利物浦大學

CPT104 - Operating Systems Concepts

Resource Management
Deadlocks



Deadlocks

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

DEADLOCK

Deadlock Characterization

Deadlock can be defined as the permanent blocking of a set of processes that compete for system resources.

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task.

The first three conditions are necessary but not sufficient for a deadlock to exist. For deadlock to actually take place, a fourth condition is required:

- **Circular wait**: a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain .

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

    CPU cycles, memory space, I/O devices

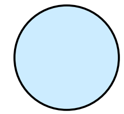
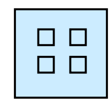
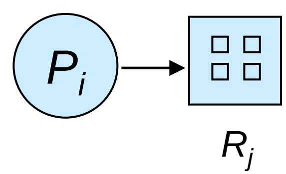
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
    - **request**
    - **use**
    - **release**

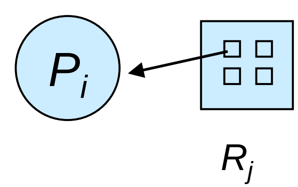
Resource-Allocation Graph

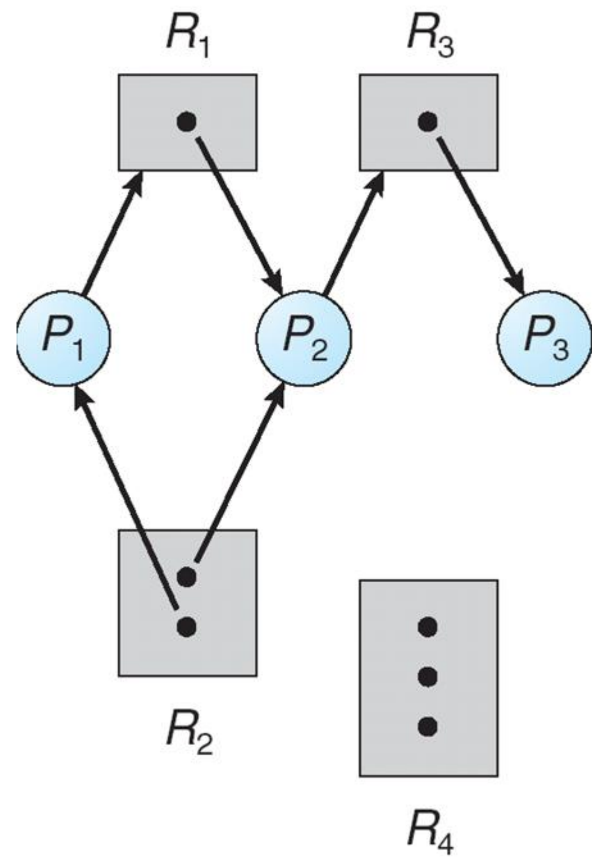
A set of vertices ***V*** and a set of edges ***E***.

- V is partitioned into two types:
  - $\boldsymbol{P = \{P_1, P_2, \ldots, P_n\}}$, the set consisting of all the **processes** in the system
  - $\boldsymbol{R = \{R_1, R_2, \ldots, R_m\}}$, the set consisting of all **resource** types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

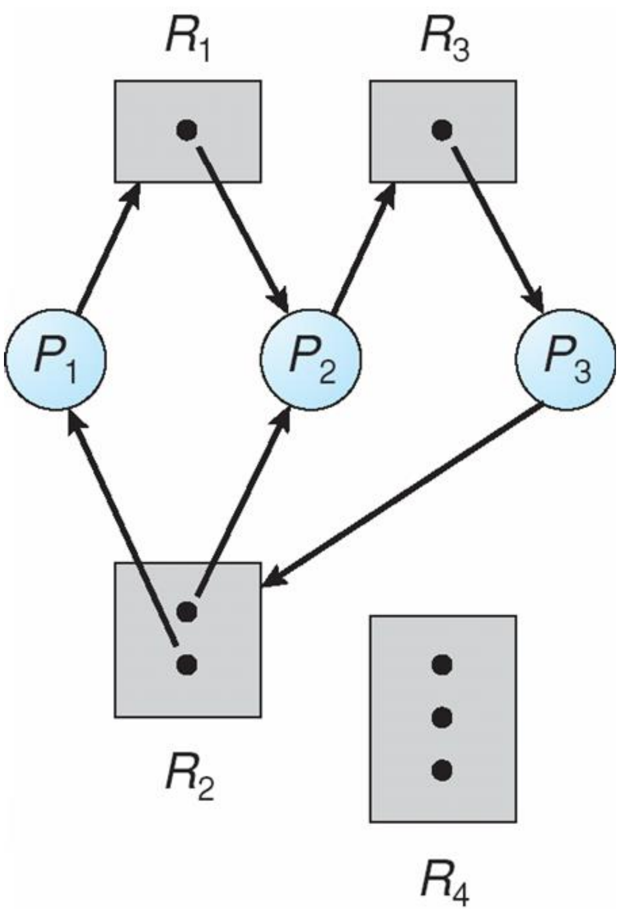
Resource-Allocation Graph (Cont.)

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

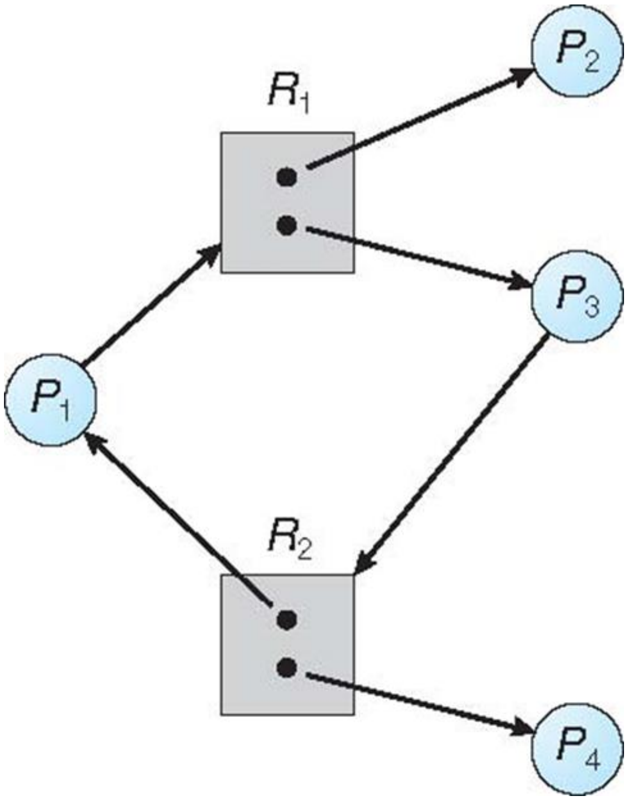
- P_i is **holding** an instance of R_j 



Resource Allocation Graph With a Deadlock



Graph With a Cycle But No Deadlock



Basic Facts

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

HANDLING DEADLOCKS

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state.
- To deal with the deadlock, the following three approaches can be used:
  - Deadlock prevention
  - Deadlock avoidance
  - Deadlock detection and recovery

Ignore the problem and pretend that deadlocks never occur in the system (used by most operating systems, including UNIX)

Deadlock Prevention

- <u>adopting a policy </u>that eliminates one of the conditions (conditions 1 through 4)

Deadlock Prevention

- **Mutual Exclusion** – In general, the first of the four conditions cannot be disallowed. If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS.

- **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

Deadlock Prevention (Cont.)

- **No Preemption** – can be prevented in several ways.

- if a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource.

- if a process requests a resource that is currently held by another process, the OS may preempt the second process and require it to release its resources.

- **Circular Wait** – can be prevented by defining a linear ordering of resource types.

- if a process has been allocated resources of type *R*, then it may subsequently request only those resources of types following *R* in the ordering.

Deadlock Avoidance

- constrain resource requests to <u>prevent at least one </u>of the four conditions of deadlock.

Deadlock Avoidance

Two approaches to deadlock avoidance:

- Do not start a process if its demands might lead to deadlock.
- Do not grant an incremental resource request to a process if this allocation might lead to deadlock.

A **safe state** is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock (i.e., all of the processes can be run to completion).

An **unsafe state** is, of course, a state that is not safe.

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

Deadlock Avoidance

The avoidance approach requires the knowledge of:

Max needs = total amount of each resource in the system

Available resources = total amount of each resource not allocated to any process

Need / Resources needed= future requests of the process *i* for resource *j*

Allocation / Current allocated resources = the resources allocated presently to process *i.*

A resource request is feasible, only if the total number of allocated resources of a resource type does not exceed the total number of that resource type in the system.

Safe state. *Example*

One resource type, multiple instances: **12 magnetic tape drives**

	Maximum Needs	Current needs / Allocation	Needs to complete	Available resources
P0	10	5	5	3
P1	4	2	2	
P2	9	2	7	

At time T0, the system is in a safe state. The sequence *<P1, P0, P2>* satisfies the safety condition.

safe state -> unsafe state.

- If P2 is given 1 more tape drive, then no longer safe -> Av. = 3 -1 = 2
- What if P0 now needs remaining 5 tape drives to complete; since there are only **2 available** (3 – 1), P0 waits
- What if P2 also now needs remaining 6 tape drives to complete; since there are only 2 available, P2 waits
- Even if P1 completes and releases resources, **P0 & P2 deadlocked**.

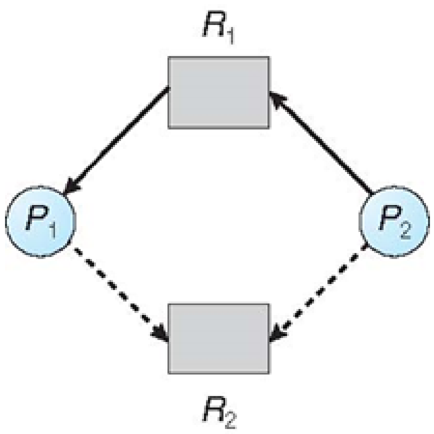
Deadlock Avoidance

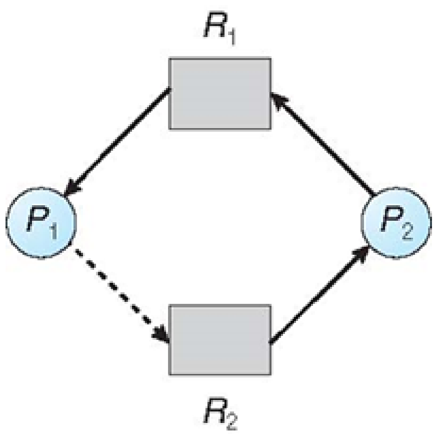
Two approaches to deadlock avoidance:

- Single Instance of Resources
- Multiple Instances of Resources

Deadlock Avoidance in Single Instance of Resources

- Where every resource type has a **single instance of resource**, the RAG can be used
- **Claim edge** $P_i \rightarrow R_j$ indicated that process *Pi* may request resource R_j; represented by a **dashed line**
- *After the cycle check, if it is confirmed that there will be no circular wait, the claim edge is converted to a request edge.*
- *Otherwise, it will be rejected.*
- *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





Deadlock Avoidance in Multiple Instances of Resources
Banker's Algorithm

The banker's algorithm has two parts:

- ***Safety Test algorithm*** that checks the current state of the system for its safe state.
- ***Resource request algorithm*** that verifies whether the requested resources, when allocated to the process, affect the safe state. If it does, the request is denied.

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 x m* matrix. If ***Max* [*i,j*] = *k***, then process P_i may request at most *k* instances of resource type R_j
- **Allocation***: n* x *m* matrix. If **Allocation[*i,j*] = *k*** then P_i is currently allocated *k* instances of R_j
- **Need***: n* x *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*]**

Banker's algorithm: *Safety Test algorithm*

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

   Work* = *Available

   Finish* [*i*] = *false for ***i* = 0, 1, …, *n*- 1**

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

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

   (b) $\boldsymbol{Need_i \leq Work}$

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

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

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

Banker's algorithm: *Resource request algorithm*

$\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$.

If $\boldsymbol{Request_i[j] = k}$ then process $\boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$

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

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$
- If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ 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:

	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	

Example (Cont.)

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

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

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

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 < $\boldsymbol{P_1, P_3, P_4, P_0, P_2}$> satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

Deadlock Detection

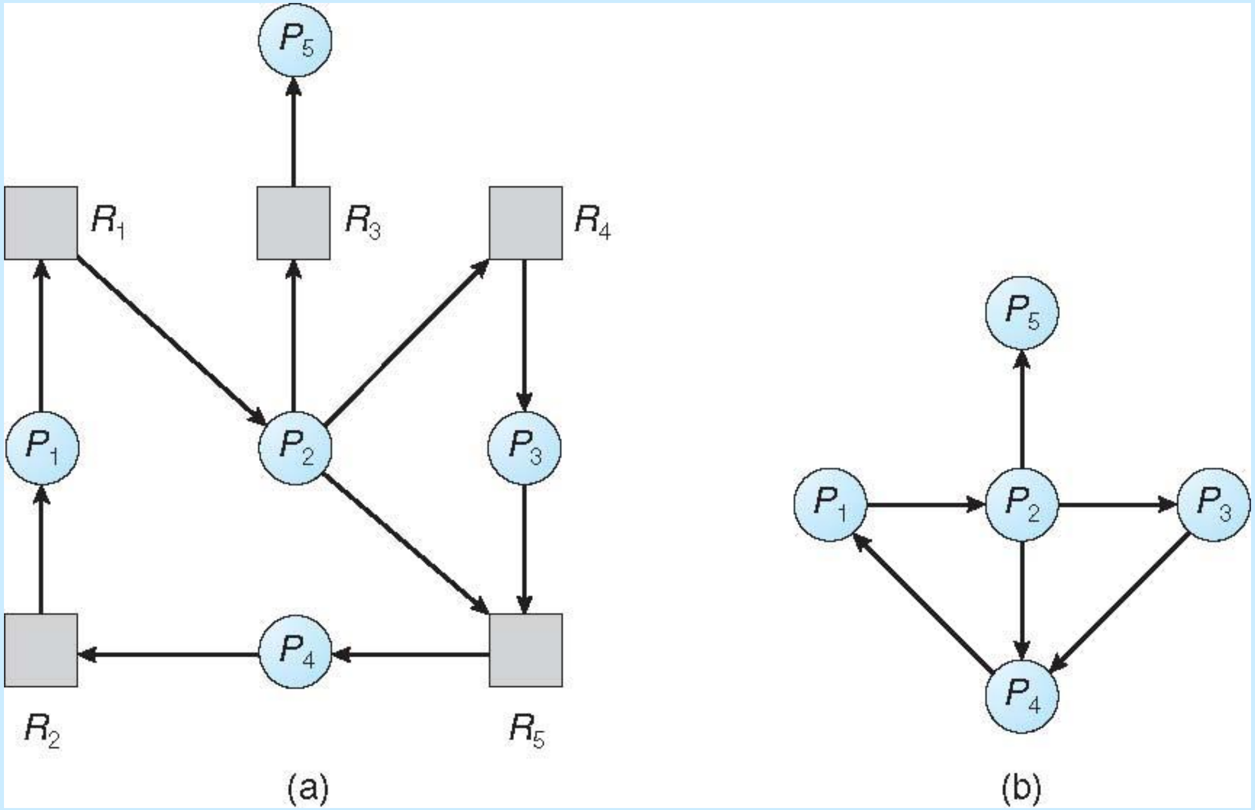
Deadlock Detection

Deadlock detection has two parts:

- ❑ Detection of single instance of resource
- ❑ Detection for multiple instances of resources

Detection of single instance of resource

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph   Corresponding **Wait-for Graph**

Detection of single instance of resource

- Maintain **Wait-for Graph**
  - **Nodes** are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
  - an **edge** exists between the processes, only if one process waits for another.
- **Periodically invoke an algorithm that searches for a cycle in the graph**. If there is a cycle, there exists a deadlock

Detection for multiple instances of resources

- **Available**: A vector of length ***m*** indicates the number of **available resources** of each type

- **Allocation**: An ***n* x *m*** matrix defines the number of resources of each type **currently allocated** to each process

- **Request**: An ***n* x *m*** matrix indicates the **current request** of each process. If ***Request* [*i*][*j*] = *k***, then process P_i is requesting ***k*** more instances of resource type R_j.

Detection Algorithm

The detection algorithm investigates **every possible allocation sequence for the processes that remain to be completed**.

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

   (a) ***Work = Available***

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

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

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

   (b) $\boldsymbol{Request_i \leq Work}$

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

Detection Algorithm (Cont.)

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

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

Example of Detection Algorithm

- **5** processes P_0 through P_4;
- **3** 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	

- **Initial**: No deadlock
- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Example (Cont.)

- ***Assume*** $\boldsymbol{P_2}$ requests **1** instance of type ***C***

Request

	A B C
	0 0 0
	2 0 2
	0 0 0
	1 0 0
	0 0 2

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?
  - We can reclaim resources held by process $\boldsymbol{P_0}$, but **insufficient resources** to fulfill other processes; requests
  - Deadlock exists, consisting of processes $\boldsymbol{P_1}$, $\boldsymbol{P_2}$, $\boldsymbol{P_3}$, and $\boldsymbol{P_4}$

Detection-Algorithm Usage

When should we invoke the detection algorithm?

- *How often a deadlock is likely to occur?*
- *How many processes will be affected by deadlock when it happens?*

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

Recovery from Deadlock

Recovery from Deadlock

Two options for breaking a deadlock.

- ❑Process Termination / Abort Process
- ❑Resource Preemption

1. Process Termination

There are two methods:

- **Abort all deadlocked processes**
- **Abort one process at a time until the deadlock cycle is eliminated**

- In which order should we choose to abort? *Many factors may affect which process is chosen.*
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resource's process needs to complete
  5. How many processes will need to be terminated

2. Resource Preemption

Three issues need to be addressed:

- ***Select a victim*** – a process, whose execution has just started and requires many resources to complete, will be the right victim for preemption (***minimize cost***).

- ***Rollback*** – return the process to some safe state (***safe checkpoint***), restart it from that state

- ***Starvation*** – it may be possible that the same process is always chosen for resource preemption, resulting in a starvation situation. Thus, it is important to ensure that the process will not starve. This can be done by ***fixing the number of times a process can be chosen as a victim.***

End of Lecture

- **Summary**
  - System Model
  - Deadlock Characterization
  - Methods for Handling Deadlocks
    - Deadlock Prevention
    - Deadlock Avoidance
    - Deadlock Detection
  - Recovery from Deadlock
- **Reading**
  - Textbook 9th edition, **chapter 7 of the module textbook**