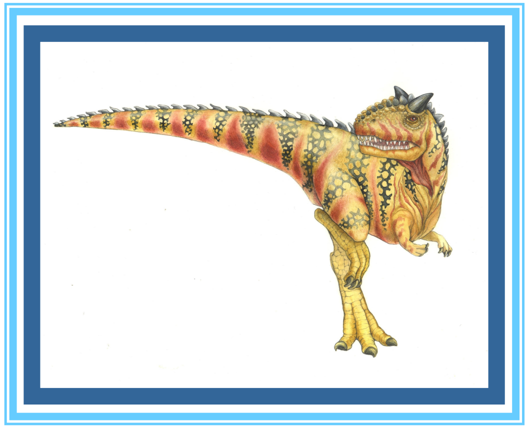
Chapter 7: Deadlocks



Deadlock

- A situation where a set of processes wait for each other's actions indefinitely
- Every process in the set is waiting for some action by some other process which is also blocked
- All processes in deadlock remain blocked permanently

System Model

- System consists of a finite set of resources, to be distributed among a set of competing processes (competing for the resources)
- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type may have several identical instances
  - Let resource type R_i has W_i instances.
  - When a process requests a resource of type R_i, any of the W_i instances may be allocated
- Each process utilizes a resource only in the following sequence:
  - **request resource**
  - **use resource**
  - **release resource**

Example of deadlock in such a model

- A system contains one tape and one printer
- Two processes P0 and P1

Process P0	Process P1
request (tape) request (printer)	request (printer) request (tape)
Use tape & printer	Use tape & printer
release (printer) release (tape)	release (tape) release (printer)

If P0 acquires the tape and P1 acquires the printer, the processes will go into a deadlock.

Deadlock Example 2

```
/* thread one runs in this function */
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);
}
/* thread two runs in this function */
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);
}
```

Characterizing Deadlocks

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

(necessay conditions for deadlock)

- **Mutual exclusion (non-shareable resources)**: only one process at a time can use a resource
- **Hold and wait**: a process continues to hold the resources that are already allocated to it, while waiting to acquire additional resources
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **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.

Resource State Modeling

- State of resource allocation can be modeled as a graph
  - Resource Request and Allocation Graph (RRAG)
  - Also called Resource Allocation Graph for simplicity

Resource-Allocation Graph

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

- V is partitioned into two types of vertices:
- Process nodes $P = \{P_1, P_2, \ldots, P_n\}$
  - Set consisting of all active processes in the system
  - Denoted as circles
- Resource type nodes $R = \{R_1, R_2, \ldots, R_m\}$
  - Set consisting of all resource types in the system
  - Denoted as rectangles

Resource-Allocation Graph

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

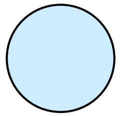
- E is also partitioned into two types of edges:

- **Request edge** – directed edge P_i --> R_j
  - Indicates process P_i has requested an instance of resource type R_j, and is currently waiting for it

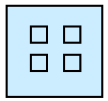
- **Assignment / Allocation edge** – directed edge R_j --> P_i
  - Indicates an instance of resource type R_j has been allocated to process P_i

Resource-Allocation Graph (Cont.)

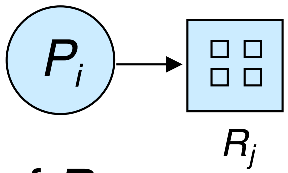
- Process



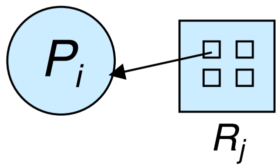
- Resource Type with 4 instances



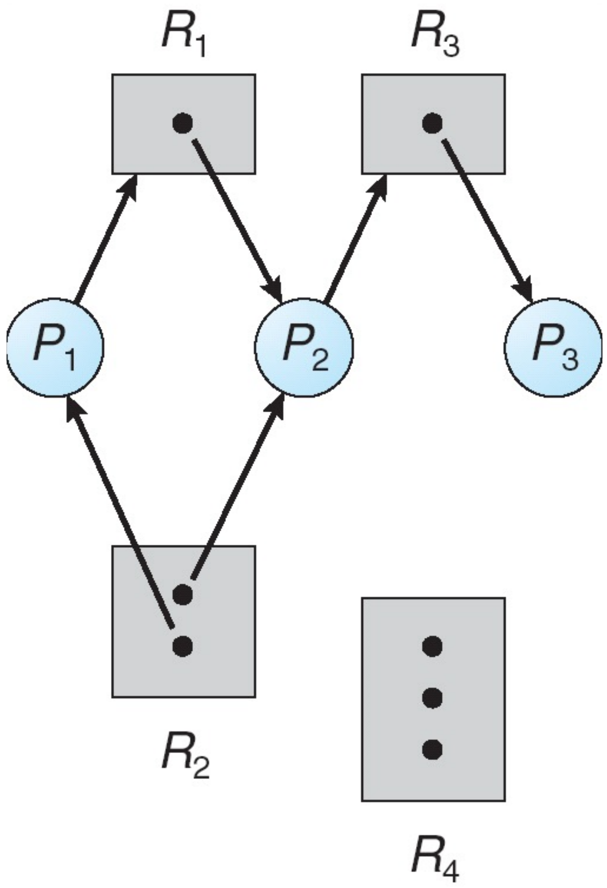
- P_i requests instance of R_j



- P_i is holding an instance of R_j



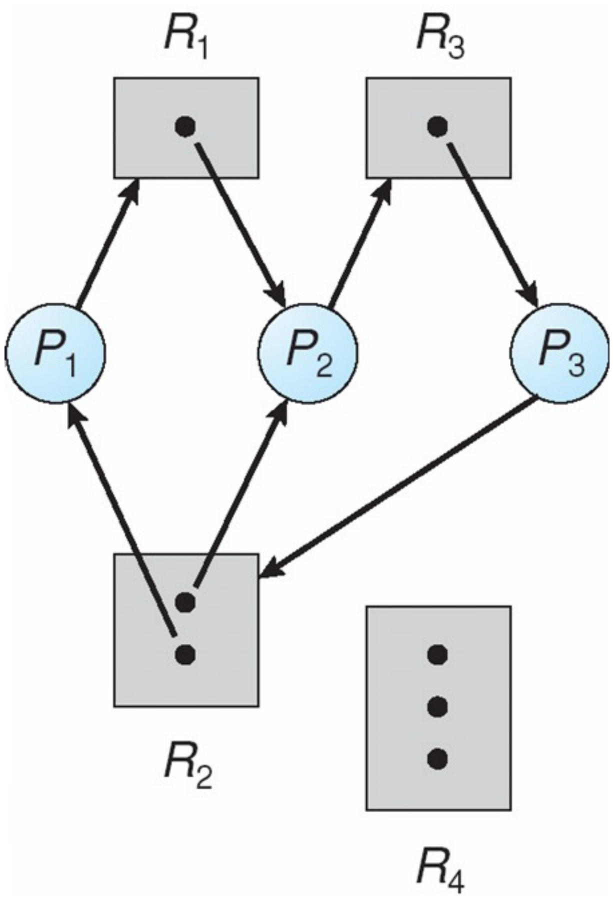
Example of a Resource Allocation Graph



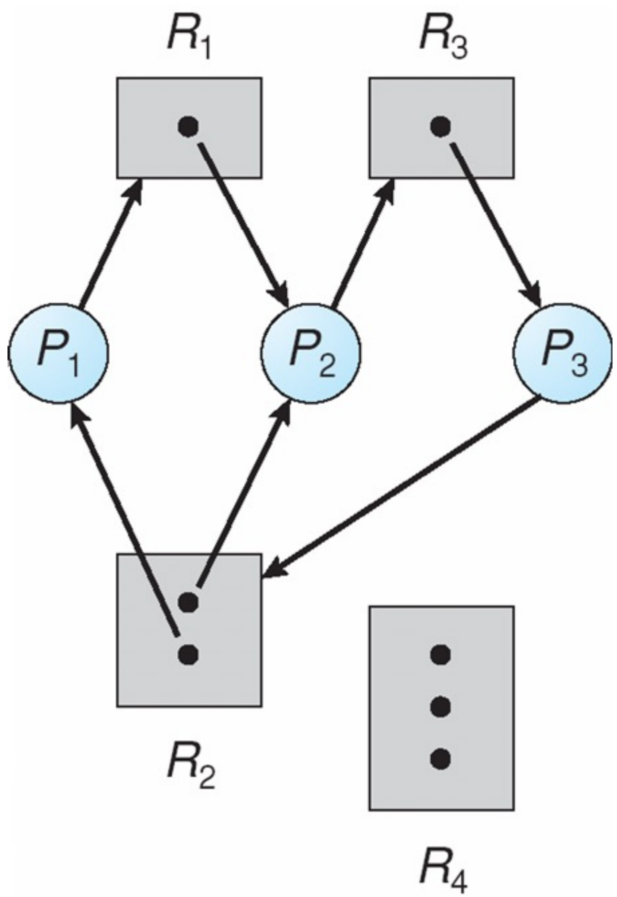
How R-A Graph changes with time

- When P_i requests an instance of resource type R_j, a request edge P_i --> R_j is inserted
- When the request is fulfilled, the edge is changed to an allocation edge R_j --> P_i
- When the process releases the resource, the allocation edge is deleted

Resource Allocation Graph With A Deadlock



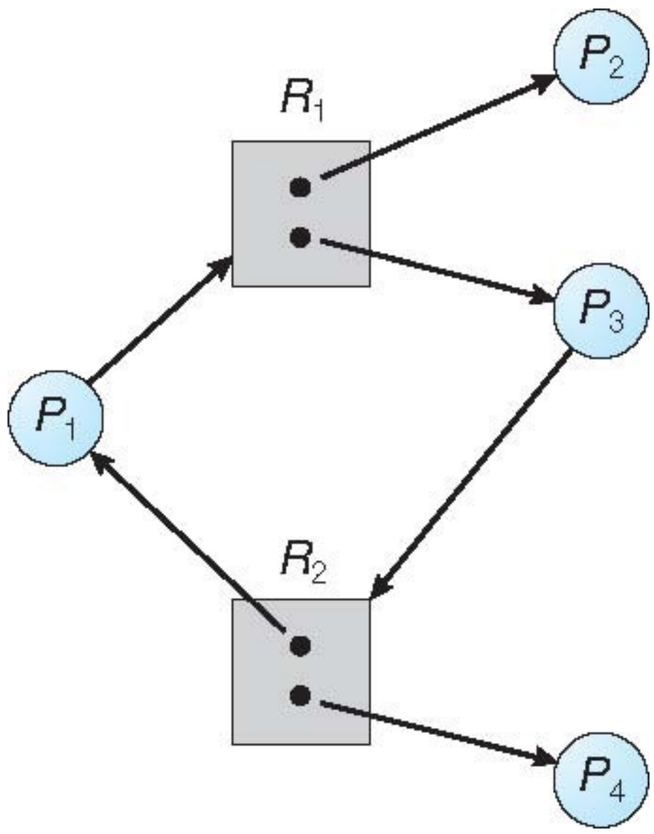
Resource Allocation Graph With A Deadlock



The presence of a cycle in the graph indicates a deadlock

But does every cycle denote a deadlock?

Graph With A Cycle But No Deadlock



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

Handling deadlocks

Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state:
  - **Deadlock prevention**: use a protocol such that one of the necessary conditions for deadlocks cannot hold (apply restrictions on how processes can request for resources)
  - **Deadlock avoidance**: kernel analyzes the resource allocation state, to determine whether granting a resource request might lead to a deadlock later on (Safe and Unsafe states)

- **Deadlock detection and resolution**
  - Kernel (or user) analyzes the resource allocation state to check whether a deadlock exists
  - If so, abort some process(es) and release resource held by them

Deadlock prevention

Deadlock Prevention

Restrain the ways in which resource requests can be made,

So that at least one of the necessary conditions for deadlock remains false

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

- Falsify **Hold and Wait** – can be done in two methods
  - 1. A process blocking on a request should not be permitted to hold any resource
  - 2. A process holding a resource should not be permitted to make additional resource requests
  - A simple approach -- require a process to request and be allocated all its required resources before it begins execution
  - Possibility of low resource utilization

Deadlock Prevention (Cont.)

Restrain the ways in which resource requests can be made,

So that at least one of the necessary conditions for deadlock remains false

- Falsify **No Preemption** –
  - 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

- Falsify **Circular Wait** –
  - We have to break the hold-and-wait cycle
  - One way -- impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock avoidance

Deadlock Avoidance

Requires that the system has some additional ***a priori*** information available about the resource usage patterns of the processes

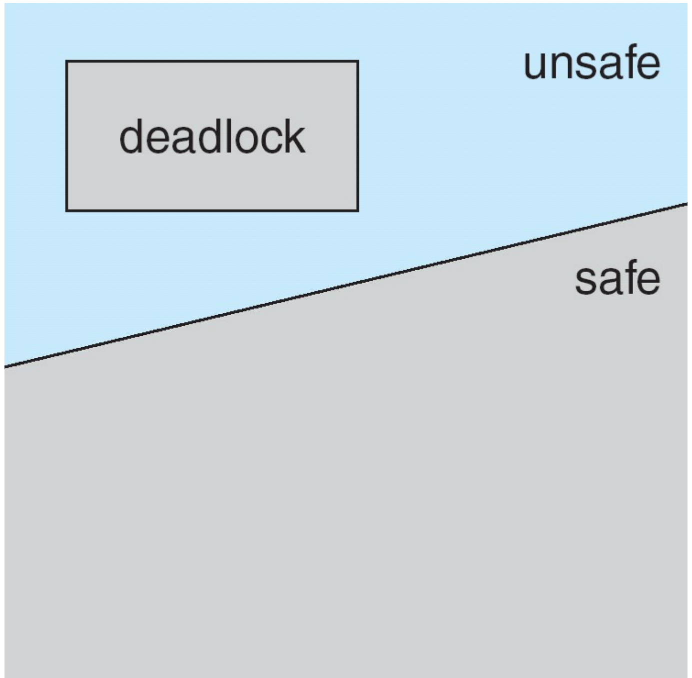
- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- Before allocating a resource, check whether this allocation may lead to a potential deadlock situation in future
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes
  - State can be safe or unsafe
  - When a process requests an available resource, decide if immediate allocation will leave the system in a safe state

Safe State

- A state is safe if the system can allocate resources to each process **in some order**, and still avoid a deadlock
- More formally:
  - System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL processes in the system 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 $j < i$
  - Such a sequence of processes is called a **safe sequence**
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished ($j < i$)
  - When all P_j ($j < i$) are 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, Unsafe, Deadlock State

- If a system is in safe state: no deadlocks
- If a system is in unsafe state (i.e., a safe sequence of processes does NOT exist): possibility of deadlock
- Deadlock avoidance: ensure that system will never enter an unsafe state



An example

- Consider a resource type with 12 instances, shared by 3 processes
- Instantaneous state:

Process	Maximum need	Current allocation
P0	10	5
P1	4	2
P2	9	2

- Does there exist a safe sequence?

An example (contd.)

- Consider a resource type with 12 instances, shared by 3 processes
- Instantaneous state:

Process	Maximum need	Current allocation
P0	10	5
P1	4	2
P2	9	2

- Does there exist a safe sequence?
  - Yes, safe sequence is <P1, P0, P2>

An example (contd.)

- Consider a resource type with 12 instances, shared by 3 processes
- Instantaneous state:

Process	Maximum need	Current allocation
P0	10	5
P1	4	2
P2	9	~~2~~ 3

- What if P2 requests and is allocated one more instance?
  - System will go to an unsafe state (from the present safe state)
  - Now, only P1 can be allocated all its required instances
  - Even after P1 terminates, system will have 4 instances, but both P0 and P2 may ask for more than 4 instances (so both P0 and P2 will have to wait and there will be deadlock)

Deadlock Avoidance Algorithms

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

This is what we will study

Banker's Algorithm

- Assumes multiple instances of each resource type
- Each process must declare the maximum resource requirement a priori
- When a process requests for a resource, it may have to wait
- When a process gets all its resources, it must return them in a finite amount of time

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 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: Notations

- If X and Y are vectors of length n, we say $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, 2, \ldots, n$

- If T denotes an n x m matrix, we use T_i to denote a vector corresponding to the i^{th} row of T
  - $Allocation_i$ vector: resources currently allocated to process P_i
  - $Need_i$ vector: additional resources that process P_i may still request

Safety Algorithm

Find out whether the current state is safe

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

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

Basically, trying to find a safe sequence

Resource-Request Algorithm for Process P_i

Determine whether a resource request can be safely granted

Request**$_i$* = request vector for process ***P**$_i$*. If ***Request**$_i$* **[*j*] = *k then process ***P**$_i$* wants ***k*** instances of resource type ***R**$_j$*

1. If ***Request**$_i$* $\leq$ ***Need**$_i$* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If ***Request**$_i$* $\leq$ ***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:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If resulting state is safe, the resources are allocated to ***P**$_i$*
- Else ***P**$_i$* is made to wait, and the old resource-allocation state is restored

Sequence of running the two algos

- When a process requests for resources
  - The Resource-Request algorithm is run
  - The Safety Algorithm may be run as part of the Resource-Request algorithm (in step 3)

- So, in practice, the sequence of running the algorithms is reverse of the order in which we discussed them

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

State after trial allocation to P_1

	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 < $\boldsymbol{P_1}$, $\boldsymbol{P_3}$, $\boldsymbol{P_4}$, $\boldsymbol{P_0}$, $\boldsymbol{P_2}$> satisfies safety requirement; hence request can be granted
- 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 and recovery from deadlock

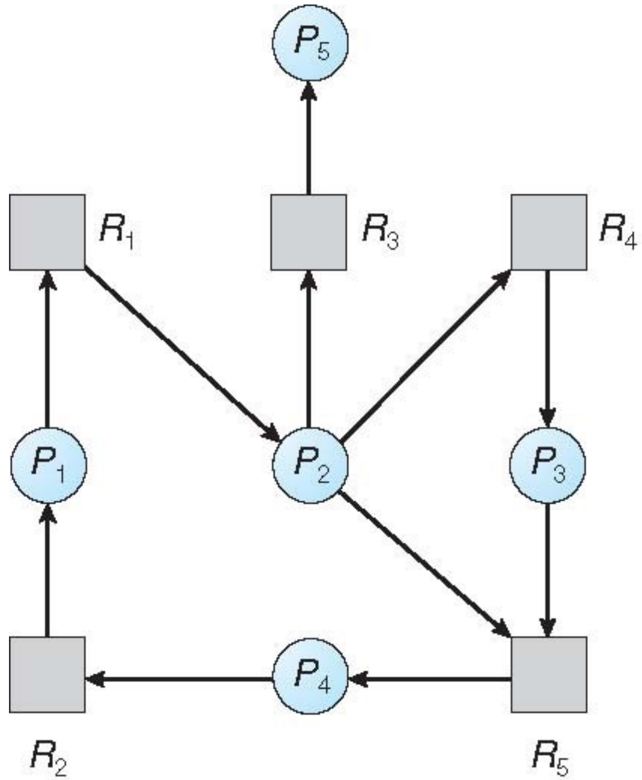
Deadlock Detection

- Peiodically run deadlock detection algorithm
- If deadlock detected, recovery scheme
- We will discuss a simple deadlock detection algorithm that assumes a single instance of each resource type

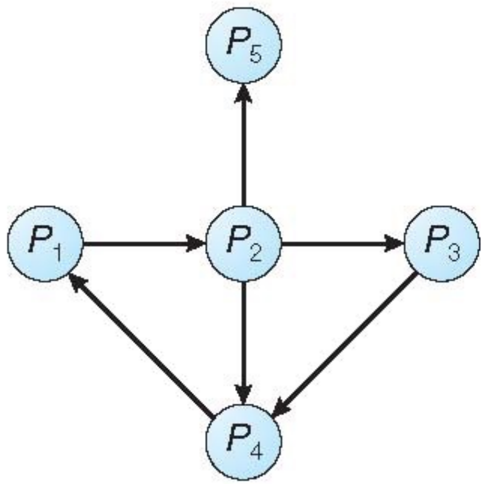
Single Instance of Each Resource Type

- Maintain a **wait-for graph**
  - Nodes are processes
  - P_i --> P_j if P_i is waiting for P_j
- Wait-for graph can be derived from Resource Allocation Graph

Resource-Allocation Graph and Wait-for Graph



(a)



(b)

Resource Allocation Graph Corresponding wait-for graph

Single Instance of Each Resource Type

- Deadlock detection:
  - Periodically invoke an algorithm that searches for a cycle in the wait-for graph
  - If there is a cycle, there exists a deadlock
- Detecting a cycle in a graph requires O(n^2) operations, where n is the number of vertices in the graph (processes)
  - Inefficient

If multiple instances of each resource type, then algorithms are more complex, with higher time complexity

Recovery from deadlock

- Two broad approaches
  - Process termination
  - Resource preemption

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 that the process has used
  4. Resources that the process needs to complete
  5. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Preempt some resources from processes and give them to other processes, until the deadlock cycle is broken

- **Selecting a victim (from which process to preempt resources)** – minimize cost

- **Rollback** – selected process has to be returned to some previously known safe (consistent) state, and restarted later from that state

- **Starvation** – same process may always be picked as victim, include number of prior rollbacks in cost factor

- Not easy to implement in practice

Finally, the Ostrich Algo for handling deadlocks

- Pretend there is no problem
- Reasonable if
  - Deadlocks occur very rarely
  - The cost for prevention / detection is high
- UNIX and Windows take this approach
- Tradeoff between correctness, convenience, cost, ...

Summary

- Deadlock characterization
  - Necessary conditions for deadlock
  - Resource Allocation Graph (cycles may indicate deadlock)

- Methods for handling deadlocks
  - Deadlock prevention
    - Ensure that some necessary condition for deadlock does not hold
  - Deadlock avoidance
    - Safe and unsafe states
    - Banker's algorithm
  - Deadlock detection and recovery

- Ostrich algorithm – just pretend deadlocks never occur