Chapter 7: Deadlocks

CS370 Operating Systems

Objectives:

- Description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- Different methods for preventing or avoiding deadlocks in a computer system

Slides based on

- **Text by Silberschatz, Galvin, Gagne**
- **Berkeley Operating Systems group**
- **S. Pallikara**
- **Other sources**

Yashwant K Malaiya

Fall 2015

Chapter 7: Deadlocks

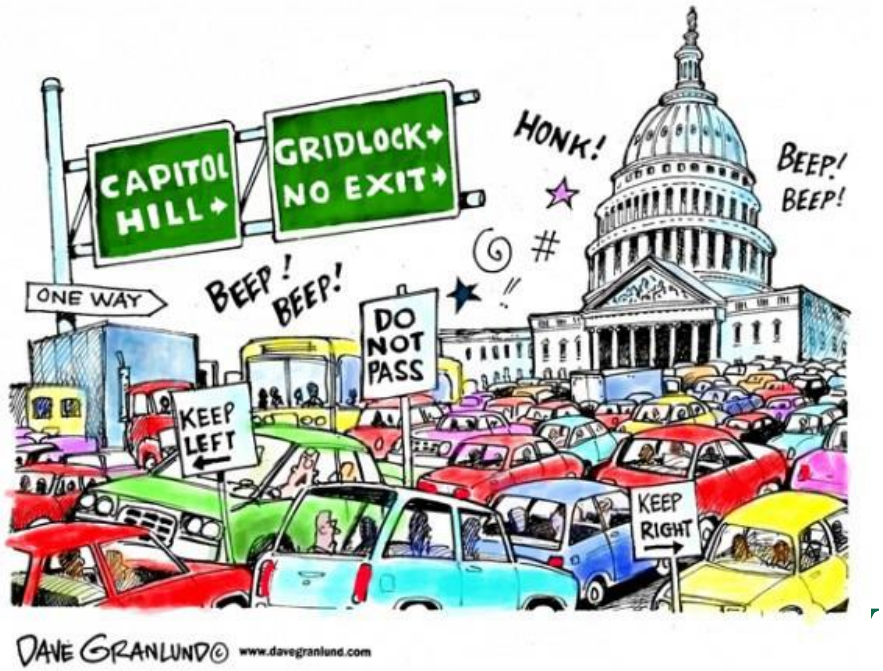
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

A Kansas Law

- Early 20th century
- "When *two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone"*
- *Story of the two goats*

Colorado State University

A Gridlock


CAPITOL HILL
GRIDLOCK
NO EXIT
HONK!
BEEP!
BEEP!
ONE WAY
BEEP!
BEEP!
DO NOT PASS
KEEP LEFT
KEEP RIGHT
DAVE GRANLUND©
www.davegranlund.com

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**

Deadlock Characterization

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
- **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.

Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc.
- See example box in text page 318 for mutex deadlock

In this example, thread one attempts to acquire the mutex locks in the order (1) first mutex, (2) second mutex, while thread two attempts to acquire the mutex locks in the order (1) second mutex, (2) first mutex. Deadlock is possible if thread one acquires first mutex while thread two acquires second mutex.

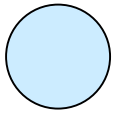
Resource-Allocation Graph

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

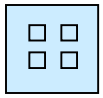
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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

Resource-Allocation Graph (Cont.)

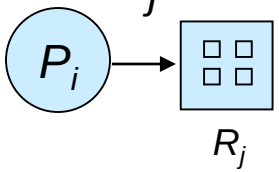
- Process



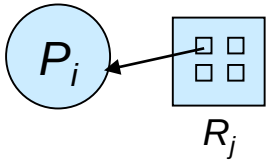
- Resource Type with 4 instances



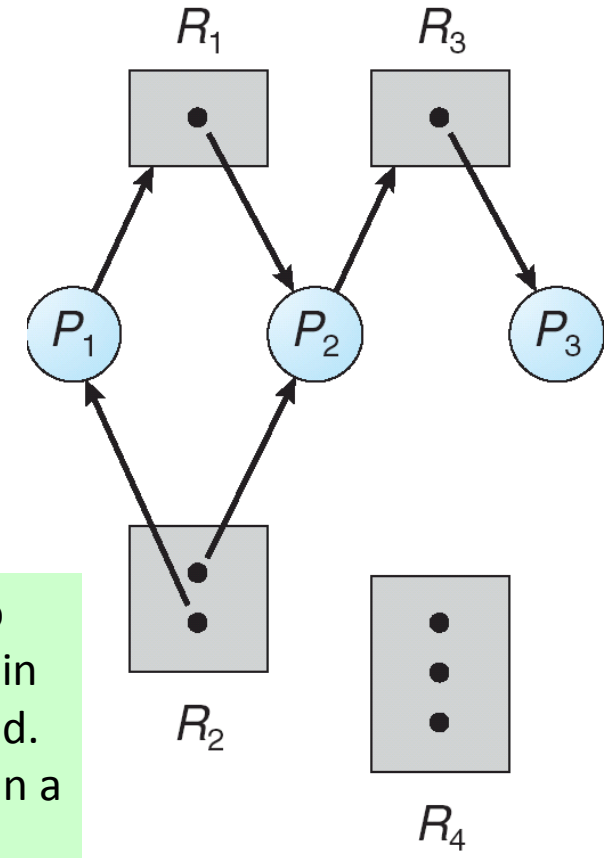
- P_i requests instance of R_j



- P_i is holding an instance of R_j

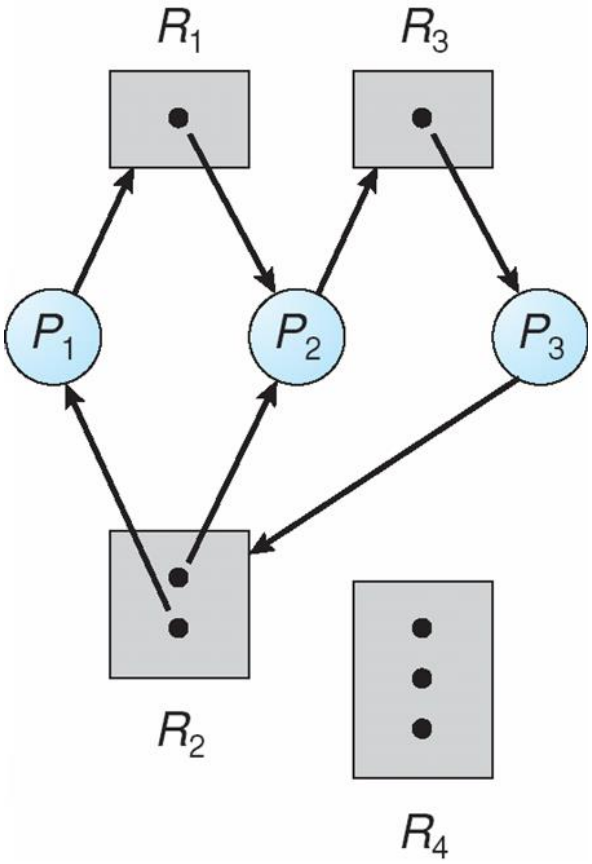


Example of a Resource Allocation Graph



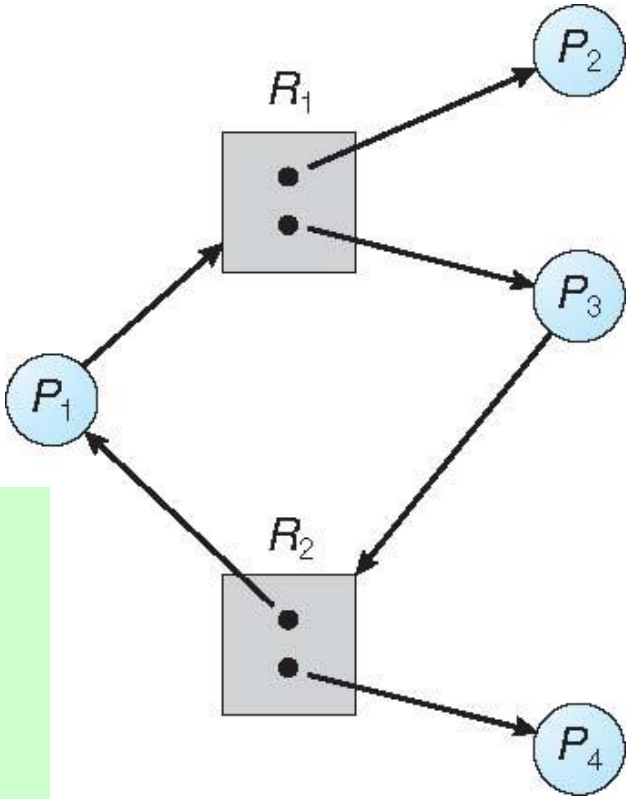
If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

Resource Allocation Graph With A Deadlock



At this point, two minimal cycles exist in the system:
P1→ R1→ P2→ R3→ P3→ R2→ P1
P2→ R3→ P3→ R2→ P2
Processes *P1, P2,* and *P3* are deadlocked.

Graph With A Cycle But No Deadlock



There is no deadlock. *P4* may release its instance of resource type *R2* . That resource can then be allocated to *P3* , breaking the cycle

If a resource-allocation graph does not have a cycle, then the system is ***not*** in a deadlocked state.

If there is a cycle, then the system may or may not be in a deadlocked state.

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

Methods for Handling Deadlocks

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

Methods for Handling Deadlocks

- Deterministic: Ensure that the system will ***never*** enter a deadlock state at any cost
- Handle if it happens: Allow the system to enter a deadlock state and then recover
- Ostrich algorithm: Stick your head in the sand; pretend there is no problem at all .
  - My be acceptable if it happens only rarely (Probabilistic view)

Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can ***prevent*** the occurrence of a deadlock.

Restrain the ways request can be made:

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **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** –
  - 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
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Dining philosophers problem: Necessary conditions for deadlock

- Mutual exclusion
  - 2 philosophers *cannot share* the same chopstick
- Hold-and-wait
  - A philosopher *picks up one* chopstick at a time
  - Will not let go of the first while it *waits for the second* one
- No preemption
  - A philosopher *does not snatch chopsticks* held by some other philosopher
- Circular wait
  - Could happen if each philosopher *picks chopstick with the same hand* first

Deadlock Example

```
/* 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);
}
```

Assume that thread one is the first to acquire the locks and does so in the order (1) first mutex, (2) second mutex.

Solution: Lock-order verifier, Witness records the relationship that first mutex must be acquired before second mutex. If thread two later acquires the locks out of order, witness generates a warning message on the system console.

Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

Transactions 1 and 2 execute concurrently. Transaction 1 transfers $25 from account A to account B, and Transaction 2 transfers $50 from account B to account A. Deadlock is possible, even with lock ordering.

Questions from last time

- Monitor vs semaphore
- Monitor and processes: entry queue, condition queues
- Monitor as a class
- Is monitor a wrapper around a critical application?
- If x.signal happens and thre is no x.wait process waiting, what will happen?
- Pascal: programming language popular in 80s
- Zombie: “terminated” state

Questions from last time

- System model: for modeling deadlock
- Multiple cycles in resource allocation graphs?
- Monitor as a class
- Other causes of deadlocks besides resource allocation?
- How do we know how often deadlocks happen?
- Do deadlocks only impact processes involved in the deadlock?

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

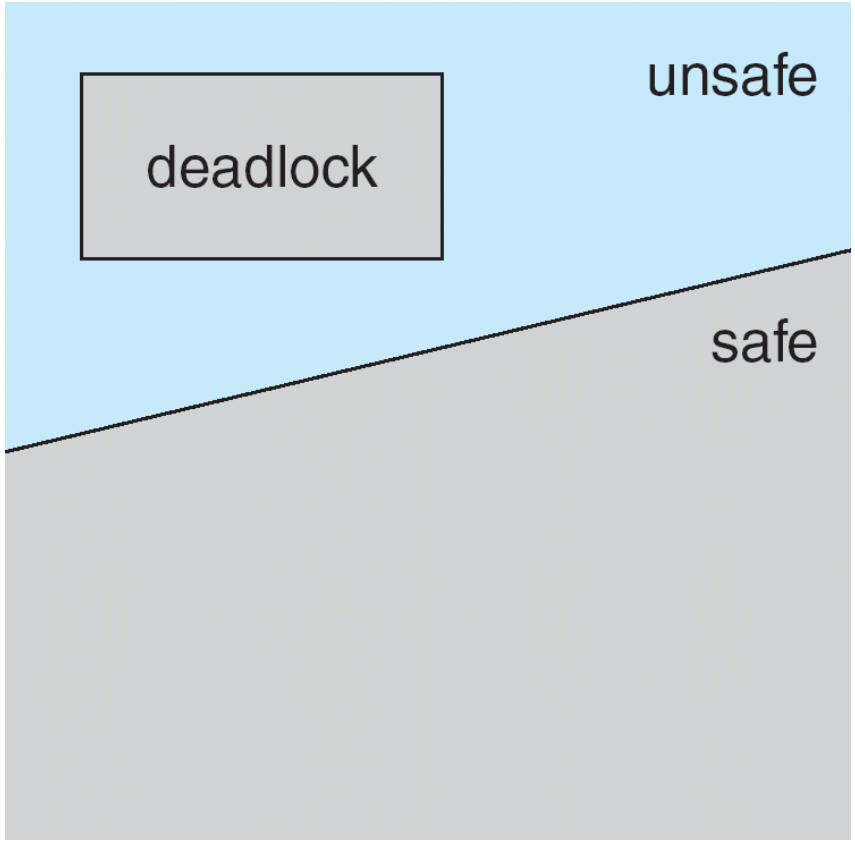
Safe State

- 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

Basic Facts

- 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.

Safe, Unsafe, Deadlock State



Example: 12 Tape drives available in the system

	Max need	Current need
P0	10	5
P1	4	2
P2	9	2

At T0:
3 drives available

Safe sequence
<P1, P0 , P2>

- At time **T0** the system is in a safe state
  - P1 can be given 2 tape drives
  - When P1 releases its resources; there are 5 drives
  - P0 uses 5 and subsequently releases them (# 10 now)
  - P2 can then proceed.

Example: 12 Tape drives available in the system

	Max need	Current need
P0	10	5
P1	4	2
P2	9	2+1

Before T1:
3 drives available

At T1:
2 drives available

- At time **T1,** P2 is allocated 1 more tape drive
  - Now only P1 can proceed.
  - When P1 releases its resources; there are 4 drives
  - P0 needs 5 and P2 needs 6
    - **Mistake** in granting P2 additional tape drive

Avoidance Algorithms

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

Questions from Last time

- **Resource-allocation** graph notation (again)
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of all the processes such that the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$
- Third state between safe and unsafe?
- **Avoidance** $\Rightarrow$ ensure that a system will never enter an unsafe state.
- How is max need determined?

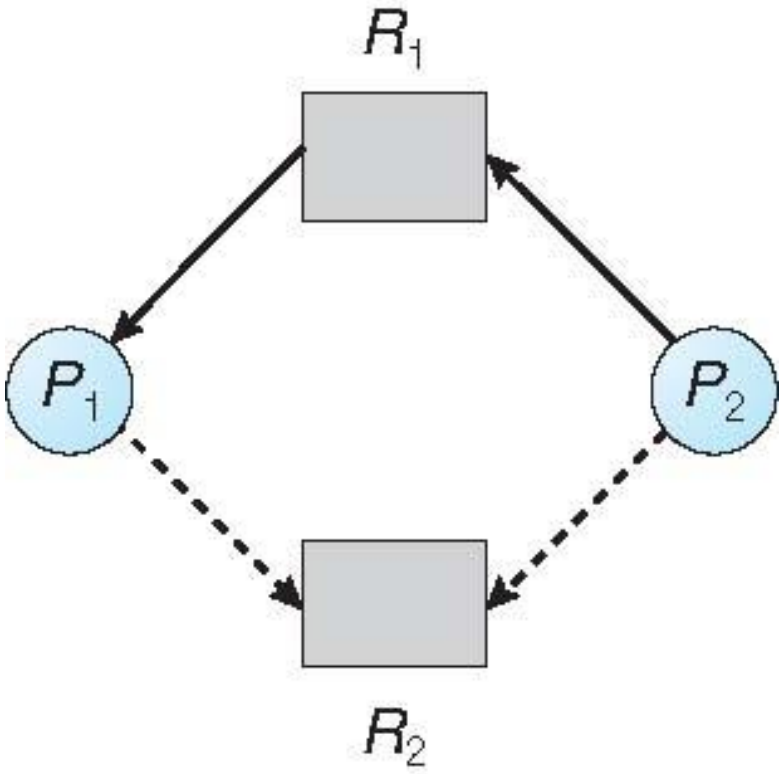
Questions from Last time

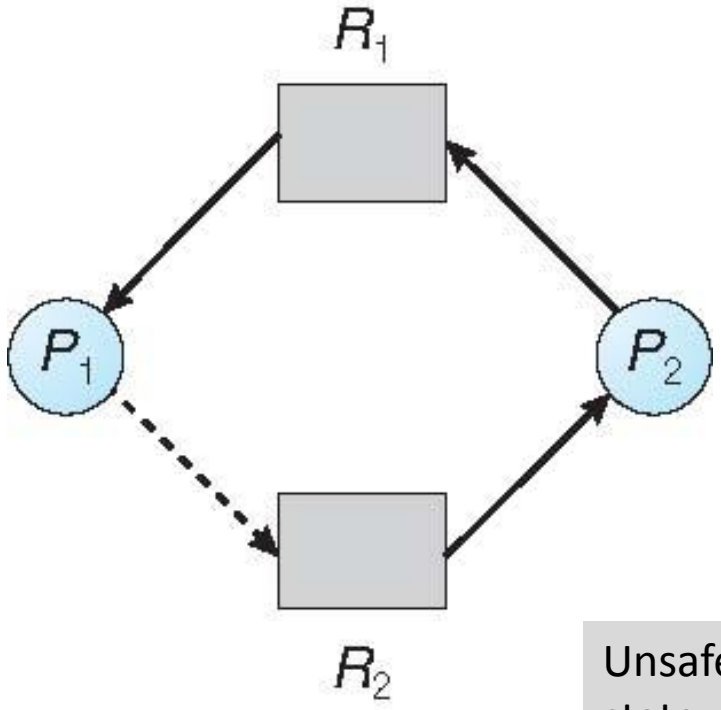
- Why is a process not currently running allocated resources?
- Other ways to end a deadlock? (soon)
- Project groups: You should all be in a project group and working now.

Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i 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

Suppose *P2* requests *R2.* Although *R2* is currently free, we cannot allocate it to *P2,* since this action will create a cycle getting system is in an unsafe state. If *P1* requests *R2,* and *P2* requests *R1,* then a deadlock will occur.

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

- Multiple instances of resources.
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait (resource request algorithm)
- Request not granted if the resulting system state is unsafe (safety algorithm)
- When a process gets all its resources it must return them in a finite amount of time
- Modeled after a banker in a small town making loans

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

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$$

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, …, *n*- 1**

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

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

   (b) $Need_i \leq Work$

   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*] == *true*** for all ***i***, then the system is in a safe state

Need: additional needed
Work: currently free
Finish: processes finished

Resource-Request Algorithm for Process P_i

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 safe $\Rightarrow$ the resources are allocated to P_i
   - If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is preserved.

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***

Available
A B C
3 3 2

	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 (see next)

Example Cont.

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

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

P1 run completion. Available becomes [3 3 2]+[2 0 0] = [5 3 2]

P3 run completion. Available becomes [5 3 2]+[2 1 1] = [7 4 3]

P4 run completion. Available becomes [7 4 3]+[0 0 2] = [7 4 5]

P2 run completion. Available becomes [7 4 5]+[3 0 2] = [10 4 7]

P0 run completion. Available becomes [10 4 7]+[0 1 0] = [10 5 7] Hence state above is safe

Ex:Assume now P_1 Requests (1,0,2)

- Check that Request ≤ Available
  - (1,0,2) ≤ (3,3,2) ⇒ true. Check for safety after pretend allocation.

	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 < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement. Yes, safe state.

Additional

- Given State is

	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	

- P4 request for (3,3,0) cannot be granted - resources are not available.
- P0 request for (0,2,0) cannot be granted since the resulting state is unsafe.

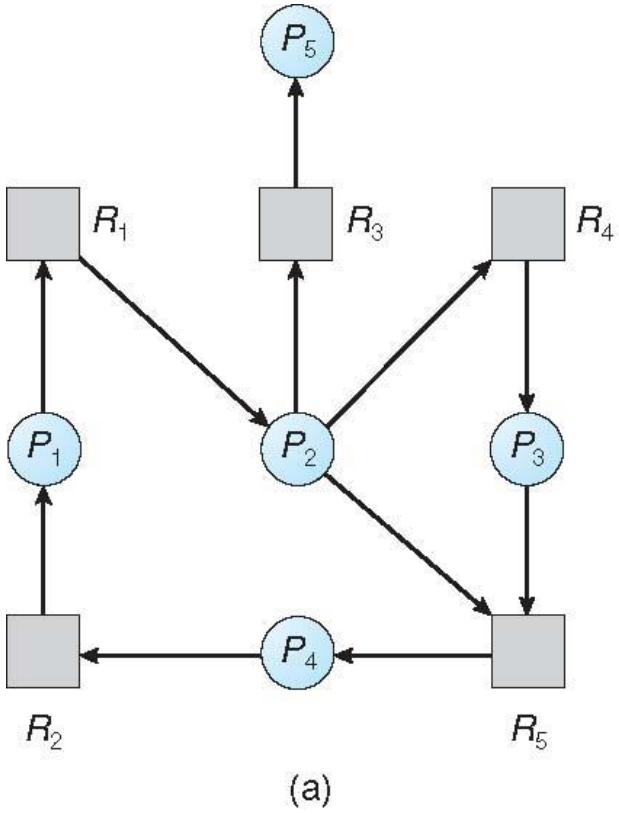
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
  - Single instance of each resource:
    - wait-for graph
  - Multiple instances:
    - detection algorithm (based on Banker's algorithm)
- Recovery scheme

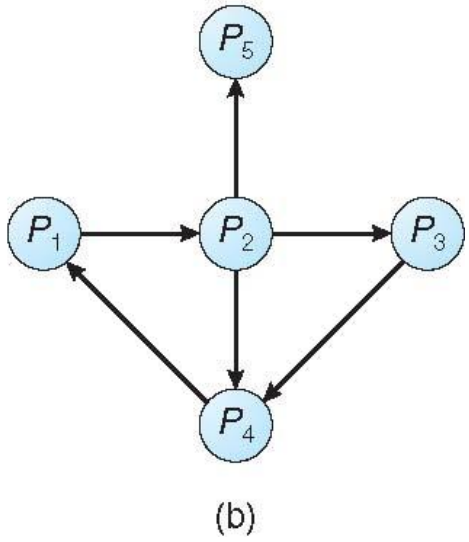
Single Instance of Each Resource Type

- Maintain **wait-for** graph (based on resource allocation graph)
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
  - ***Deadlock if cycles***
- 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

3 cycles. Deadlock.

Colorado State University

Several Instances of a Resource Type

- **Available***:* A vector of length ***m*** indicates the number of available (currently free) 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

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 (find next process)

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 x 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:

	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. ***No deadlock***

Example (Cont.)

- 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

Available
A B C
0 0 0

- 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.

Recovery from Deadlock: Process Termination

Choices

- 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

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

Deadlock recovery through rollbacks

- **Checkpoint** process periodically
  - Contains memory image and resource state
- Deadlock detection tells us *which* resources are needed
- Process owning a needed resource
  - **Rolled back** to before it acquired needed resource
    - Work done since rolled back checkpoint discarded
  - **Assign** resource to deadlocked process