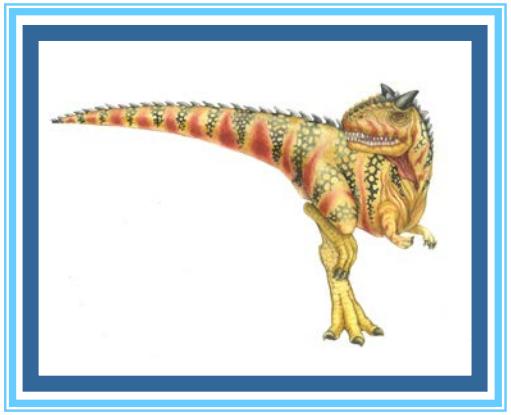
Chapter 7: Deadlocks



Chapter 7: Deadlocks

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

Chapter Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system

Deadlock

- Recall from Chapter 5
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Example: Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

System Model

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

  CPU cycles, files, memory space, I/O devices, etc
- Each resource type R_i has W_i instances.
  - Type: CPU
    - 2 instances - CPU1, CPU2
  - Type: Printer
    - 3 instances - printer1, printer2, printer3

System Model

- Each process utilizes a resource as follows:
  - **Request** resource
    - A process can request a resource of a given type
      - E.g., “I request any printer”
    - System will then assign a instance of that resource to the process
      - E.g., some printer will be assigned to it
    - If cannot be granted immediately, the process must wait until it can get it
  - **Use** resource
    - Operate on the resource, e.g. print on the printer
  - **Release** resource
- Mutexes and Semaphores
  - Special case:
    - Each mutex or semaphor is treated as a separate resource **type**
    - Because a process would want to get not just “any” lock among a group of locks, but a specific lock that guards a specific shared data type
      - e.g., lock that guards a specific queue

Deadlock Characterization

Deadlock can arise if 4 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, and so on:
  - $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_{n-1} \rightarrow P_n \rightarrow P_0$
- Notice: "Circular wait" implies "Hold and Wait"
  - Why then not test for only the "Circular wait"?
  - Because, computationally, "Hold and wait" can be tested much more efficiently than "Circular wait"
  - Some algorithms we consider only need to check H&W

Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc.
- Trivial example
  - Mutexes A, B – unlocked initially
  - Process P1: wait(A); wait(B);
  - Process P2: wait(B); wait(A);

Deadlock Example

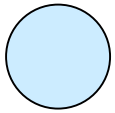
```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/* thread two runs in this 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);
}
```

Resource-Allocation Graph

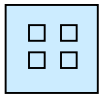
- Resource-Allocation Graph
  - Useful tool to describe and analyze deadlocks
  - Set of vertices V
  - Set of edges E
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
  - Means P_i has requested (an instance of) R_j and now is waiting for it
- **assignment edge** – directed edge $R_j \rightarrow P_i$
  - Means an instance of R_j has been assigned to P_i

Pictorial Representation

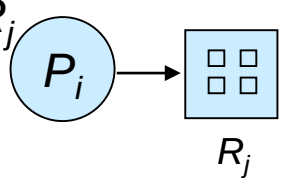
- Process



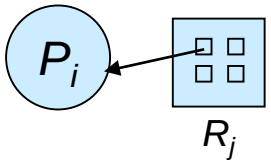
- Resource Type (with 4 dots for 4 instances in this specific example)



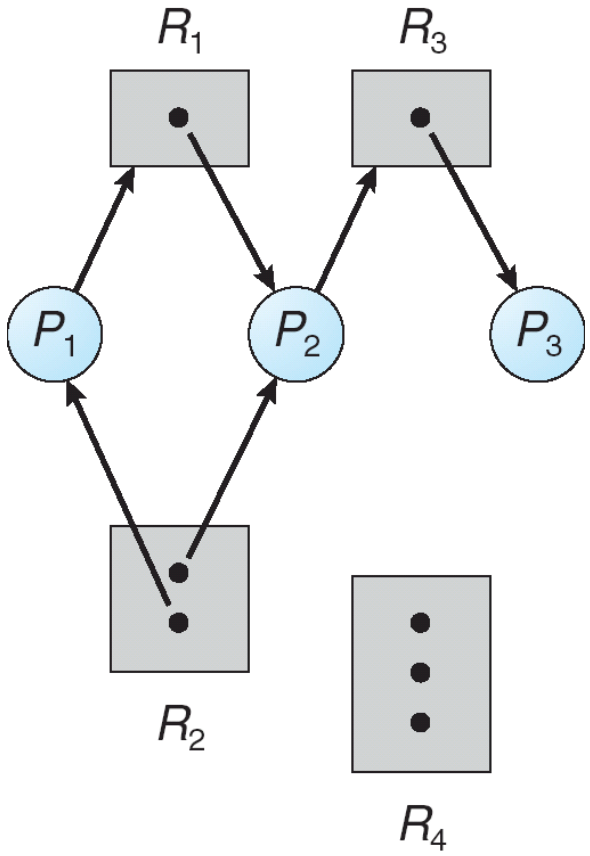
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example of a Resource Allocation Graph



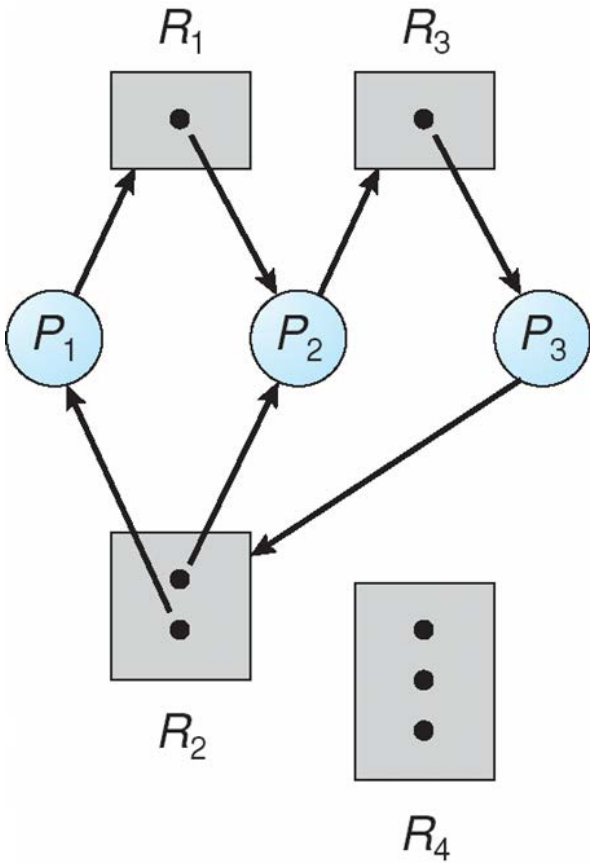
Basic facts

- Consider Resource-Allocation Graph

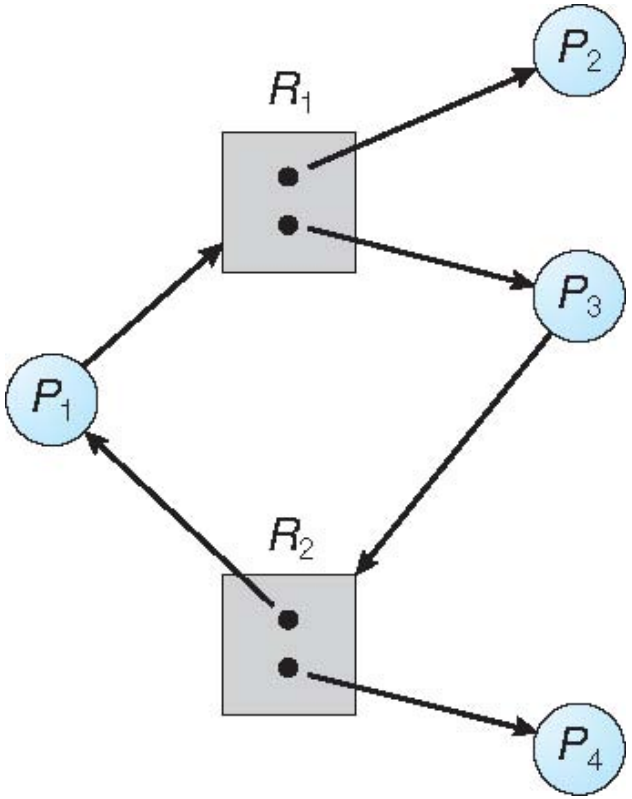
- **Case 1:** Each resource type has only 1 instance
  - A cycle in the graph is a necessary and sufficient condition for a deadlock
  - That is: graph has a cycle ⇔ a deadlock

- **Case 2:** A resource type can have multiple instances
  - A cycle in the graph is a necessary (but **not** sufficient) condition for a deadlock, that is:
  - If graph has no cycles, then no process is deadlocked
  - If graph has a cycle, may (or may not) be deadlocked

Two Minimal Cycles and P1 P2 P3 are Deadlocked



A Cycle But No Deadlock



- P4 might release its instance of R2
- That instance will then be allocated to P3
- Edge $P_3 \rightarrow R_2$ will become $R_2 \rightarrow P_3$
- Hence, the cycle will break

Methods for Handling Deadlocks

- Three ways to handle deadlocks:

1. Ensure that the system will ***never*** enter a deadlock state:
   - Deadlock prevention
   - Deadlock avoidance
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
   - One reason: Handling deadlocks can be computationally expensive

Deadlock Prevention

- **Key idea:** Restrain the ways request for resources can be made
- Recall, all 4 necessary conditions must occur for a deadlock to happen
  1. Mutual Exclusion
  2. Hold and Wait
  3. No preemption
  4. Circular Wait
- If we ensure at least one condition is not met, we prevent a deadlock

Deadlock Prevention: Mutual Exclusion

- **#1 Mutual Exclusion:** only one process at a time can use a resource
- Solution:
  - Mutual exclusion is not required for sharable resources
    - Example: Accessing the same files, but only for reading
    - So do not use mutual exclusion for such cases
  - However, it must hold for non-sharable resources
    - **Bottom line:** Use mutual exclusion only when you really have to

Deadlock Prevention: Hold and Wait

- **#2 Hold and Wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **Idea:** must guarantee that whenever a process requests a resource, it does not hold any other resources
- Solutions include:
  1. Require process to request and be allocated all its resources before it begins execution, or
  2. Allow process to request resources only when the process has none allocated to it
     - **Cons:** Low resource utilization; starvation possible

Deadlock Prevention: No Preemption

- **#3 No Preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task

One possible solution (is to implement preemption):

- **Assume**
  - Process P is holding some resources - (set) R
  - P then requests another resource r
  - But r cannot be immediately allocated to P
    - That is, P must wait for r
- **Then**
  - All resources R are preempted
    - That is, they are implicitly released
  - Resources from R are added to the list of resources for which P is waiting
  - P will be restarted only when it can regain R , as well as r

Deadlock Prevention: Circular Wait

- **#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, and so on
  - $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_{n-1} \rightarrow P_n \rightarrow P_0$

One possible solution:

- impose a total ordering of all resource types, and
  - OrderOf(R_j) – gives order of R_j
- require that each process requests resources in an increasing order of enumeration
  - Rewrite the code such that this holds
  - If a process holds a lock for R_j it should not request a lock for any R_k such that OrderOf(R_k) < OrderOf(R_j)
- Example
  - Order resources A,B,C,D,E as D < E < C < A < B
  - Assume: Process holds a lock for, say, A and C
  - Then, the process should **not** request locks for D or E

Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);  // notice, lock1 is assigned dynamically
   lock2 = get_lock(to);    // lock2 is assigned dynamically as well
   acquire(lock1);          // yes, lock1 is always acquired before lock2
   acquire(lock2);          // but it does not mean order(lock1) < order(lock2)

   withdraw(from, amount);
   deposit(to, amount);

   release(lock2);
   release(lock1);
}
```

- Transactions 1 and 2 execute concurrently: (1) transaction(A, B, $25) (2) transaction(B, A, $50)
- Deadlock is possible (even though a presumed “ordering” exists)
  - Because locks are acquired/assigned **dynamically**
- How to fix it?
  - don’t fix the order to “lock1 and then lock2”
  - Instead, order lock1 and lock2 each time: after get_lock() but before acquire()

Deadlock Avoidance

Idea: Require that the system has some additional ***a priori*** information about how resources will be used

- 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:
  1. number of available resources
  2. number of allocated resources
  3. maximum resource demands of the processes

Safe State

- The deadlock avoidance algorithm relies on the notion of safe state
- A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.

- **Formally:** System is in **safe state** only if
  - There exists a **safe sequence** (of processes) -- explained shortly
  - That is, it is possible to **order** all processes to form a safe sequence

Safe Sequence

- System is in **safe state** only if there exists a **safe sequence**

- **Safe sequence -** is a sequence (ordering) <P_1, P_2, …, 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, plus
    - resources held by all the P_j, with $j < i$ (that is, by P_1, P_2, …, P_{i-1})

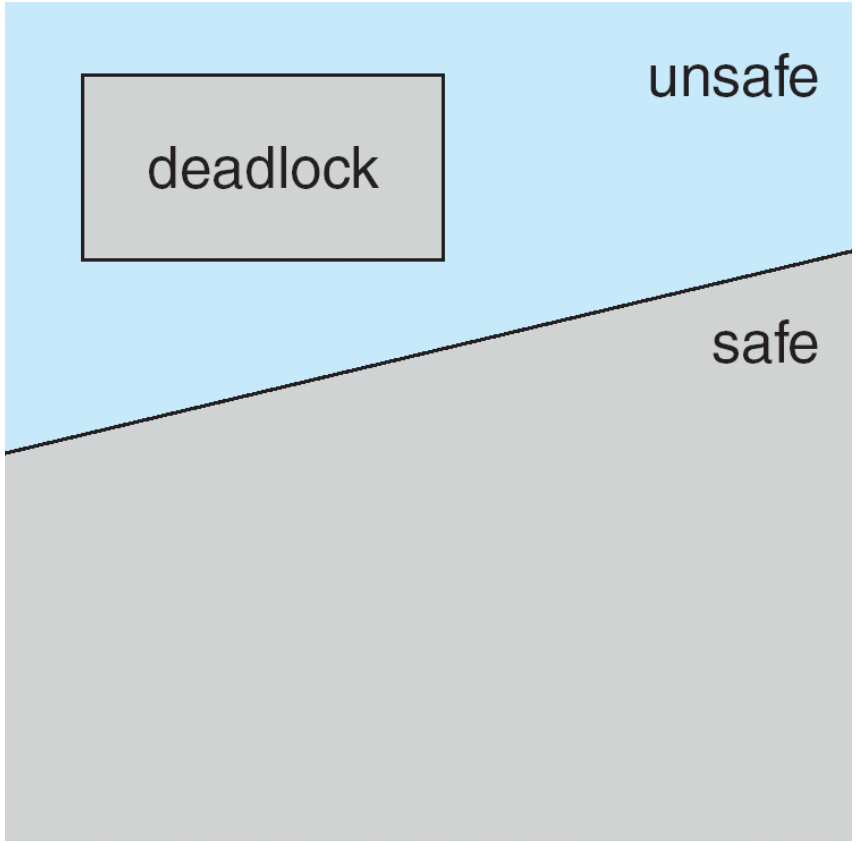
- **Intuition** behind a safe sequence:
  - **if** P_i resource needs are not immediately available
  - **then** P_i can wait until P_1, P_2, …, P_{i-1} have finished
    - at that stage P_i will be guaranteed to obtain the needed resources
      - so P_i can execute, return allocated resources, and terminate

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.

- **Solution:** Whenever a process requests a resource that is available:
  - Decide:
    - If the resource can be allocated immediately, or
    - If the process must wait
  - Request is granted only if it leaves the system in the safe state

Safe, Unsafe, Deadlock State



Deadlock Avoidance Algorithms

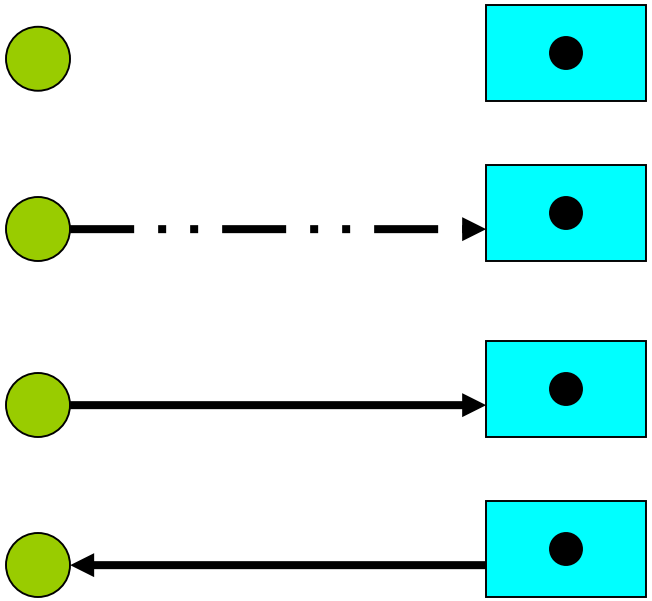
- Deadlock Avoidance algorithms for two cases:

- **Case 1.** Each resource type has exactly 1 instance
  - Use a (modified) resource-allocation graph
- **Case 2.** A resource type can have multiple instances
  - Use the banker's algorithm

Case1: Single instance per resource type

- Use the resource allocation graph
- **Idea:** Resources must be claimed *a priori* in the system
  - Before they start, all processes declare which exact resources they may use
- **Recall that:** when only 1 instance per resource type, it holds:
  - graph contains a cycle ⇔ deadlock
- **Solution:** the resource allocation graph is augmented
  - with claim edges
  - to provide deadlock avoidance

Edges in the claim graph



Process may request resource

Process has requested resource

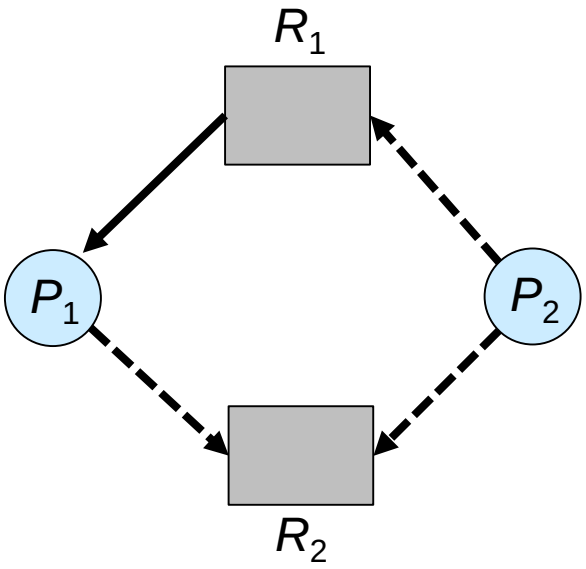
Process is assigned resource

Claim Edges

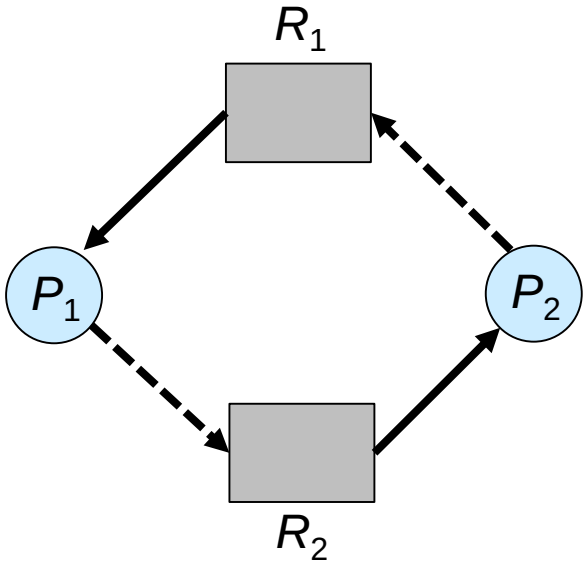
- **Claim edge**
  - $P_i \rightarrow R_j$ indicates that process P_j may request resource R_j
  - It is represented by a dashed line
- When a process requests a resource
  - The claim edge converts to request edge
- When the resource is allocated to the process
  - The request edge converted to an assignment edge
- When a resource is released by a process
  - The assignment edge reconverts to a claim edge

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



Safe state



Unsafe state

Banker's Algorithm

- Now what if Multiple instances of resources per resource type
  - Use Banker's Algorithm
- Each process must a priori claim maximum use
- When a process requests an available 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

- An example on the next slide…
- *n* = number of processes
- *m* = number of resources types
- Process Pi, resource type R_j
- **`Available[m]`***:*
  - vector of length *m*
  - `Available[j]=k` -- means *k* instances of Rj available
- **`Max[n,m]`***:*
  - *n x m* matrix
  - `Max[i,j]=k` -- means P_i may request ≤*k* instances of R_j
- **`Allocation[n,m]`***:*
  - *n* x *m* matrix
  - `Allocation[i,j]=k` -- means P_i is currently allocated *k* instances of R_j
- **`Need[n,m]`***:*
  - *n* x *m* matrix
  - `Need[i,j]=k` – means P_i may need *k* more instances of R_j to complete its task
  - `Need[i,j]=Max[i,j] – Allocation[i,j]`

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5instances), and *C* (7 instances)

- Snapshot at time T_0:

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

- Recall that ***Need*** = ***Max – Allocation***

Safety Algorithm

// **Conceptually:** tries to find a safe sequence by constructing it.

// Adds one process at a time to it

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

```
Work=Available;                          // vector of currently available resources
Finish[i]=false, for i=0,1,…,n-1;        // Pi is already assigned to the sequence?
```

2. Find an ***i*** such that both: // Find Pi, such that:

(a) `Finish[i]=false;` // (a) Pi is not yet assigned to the sequence, and

(b) $Need_i \leq Work$; // (b) Pi needs ≤ resources than what's currently available

```
if (no such i exists) goto Step 4;
```

// Here, Pi is OK to add, and hence added to the working sequence, conceptually

3. $Work = Work + Allocation_i$; // add to Work resources that Pi already has

// Pi will release them when it finishes

```
Finish[i] = true;                        // Pi is assigned to the sequence, so it is done
goto Step 2;
```

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

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), B (5instances), and C (7 instances)

- Snapshot at time T_0:

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

- Recall that ***Need*** = ***Max – Allocation***
- The system is in a safe state
  - Since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Resource-Request Algorithm for Process P_i

$Request_i$ = request vector for process P_i.

$Request_i[j]=k$ means P_i wants k instances of resource type R_j

1. if ($Request_i \leq Need_i$) goto Step 2.
   else signal error, since P_i has exceeded its max claim

2. if ($Request_i \leq Available$) goto Step 3.
   else 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 restored

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true
- Available becomes (3,3,2) – (1,0,2) = (2,3,0)

	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
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

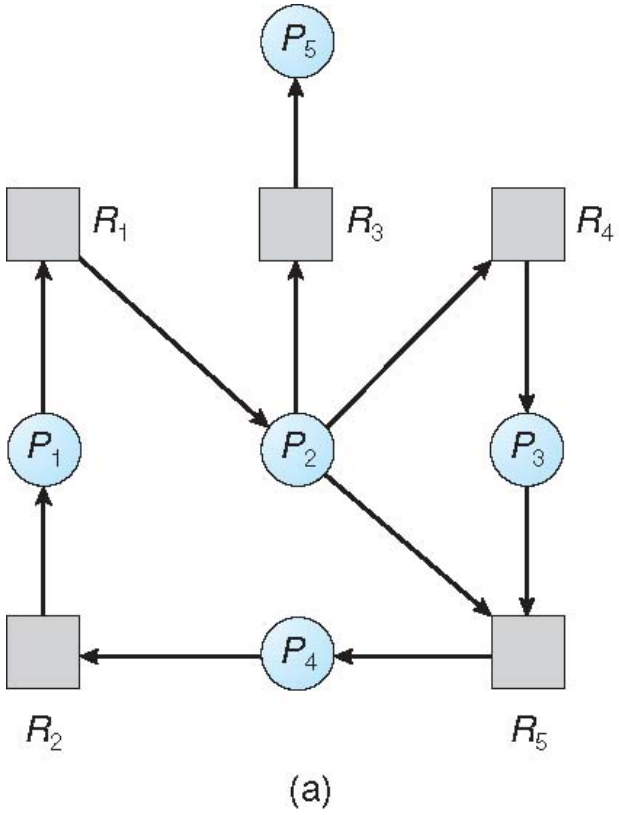
Deadlock Detection

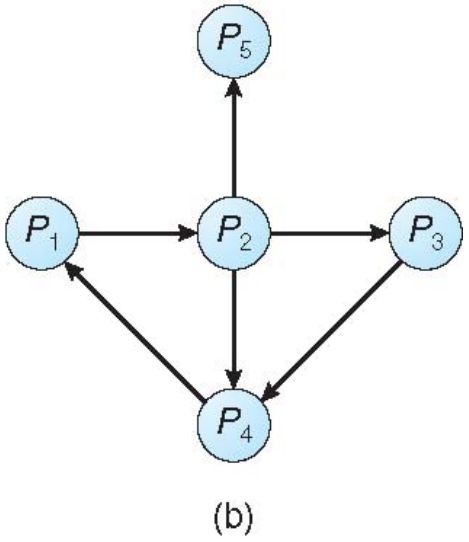
- If no deadlock-prevention or deadlock-avoidance algorithm is used
  - Then system can enter a deadlock state
- In this environment, the system may provide
  - Deadlock detection algorithm
  - Deadlock recovery algorithm

Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - A variant of the resource-allocation 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

Several Instances of a Resource Type

- **`Available[m]`***:*
  - A vector of length ***m***
  - indicates the number of available resources of each type
- **`Allocation[n,m]`***:*
  - An ***n*** **x** ***m*** matrix
  - defines the number of resources of each type currently allocated to each process
- **`Request[n,m]`***:*
  - an ***n*** **x** ***m*** matrix
  - indicates the current request of each process.
  - ***`Request[i][j] = k`*** means process P_i is requesting ***k*** more instances of resource type R_j.

Detection Algorithm

//Tries to see if there is at least on way to assign resources to processes, otherwise=>deadlock

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

```
Work = Available;                                  //Resources currently Available
for (i=0; i<n; i++)
   if (Allocation_i!=0) Finish[i] = false;  // Pi could be deadlocked: check
   else                 Finish[i] = true;   // only a process that holds some resources
                                            // can be in a deadlock, otherwise - cannot
```

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

   (a) `Finish[i] == false;` // (a) Pi is not done yet, needs to be processed

   (b) $Request_i \leq Work$; // (b) Pi requests ≤ resources than available

   // optimistic: using $Request_i$ not $Need_i$

```
if (no such i exists) goto Step 4;
```

// Pi can be allocated

3. `Work = Work + Allocation_i` // add to Work resources that Pi already has

```
Finish[i] = true;
goto Step 2
```

4. `if (Finish[i] == false, for some i)`

- 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***

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

- 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

- Abort all deadlocked processes
  - **Cons:** Very costly – processes could have been running for long time.
    They will need to be restarted.
- Abort one process at a time until the deadlock cycle is eliminated
  - **Cons:** High overhead since the deadlock detection algorithm is called each time one process is terminated

- 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

- Resource preemption is another method to recover from deadlocks
- **Selecting a victim** – which process to choose to preempt its resources?
  - Minimize “cost”.
- **Rollback** – return to some safe state, restart process for that state.
  - Process cannot continue “as is”, since its resources are preempted
  - The rollback state should be such that the deadlock breaks
- **Starvation** – same process may always be picked as victim
  - One solution: include number of rollback in cost factor

End of Chapter 7