Deadlock

Deadlock

- 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);

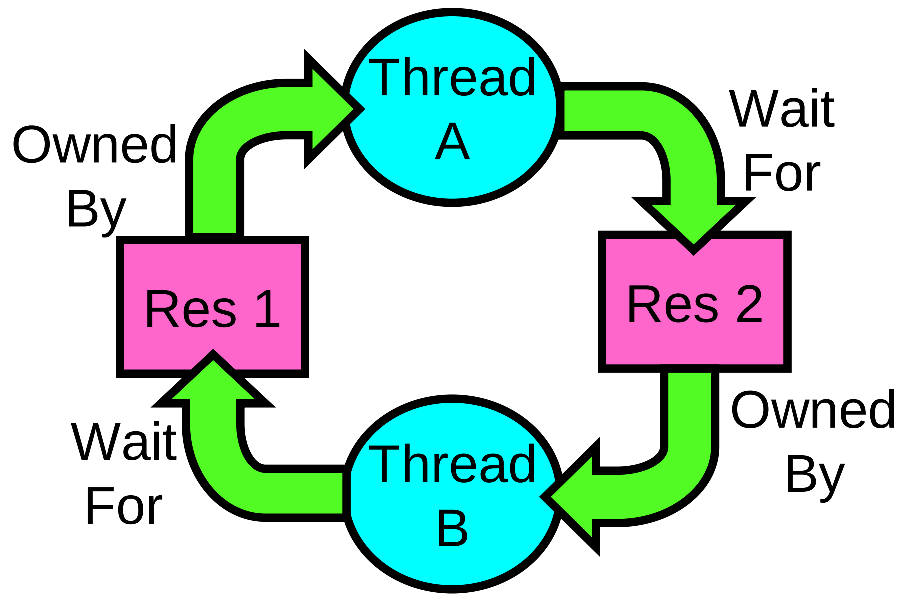
Deadlocks

Deadlock can be defined formally as follows:

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

Types of Scheduling Problems

- Starvation – thread fails to make progress for an indefinite period of time
- Deadlock – starvation due to a *cycle of waiting* among a set of threads
  - each thread waits for some other thread in the cycle to take some action



Deadlock with Space

<u>Thread A</u>
AllocateOrWait(1 MB)
AllocateOrWait(1 MB)
Free(1 MB)
Free(1 MB)

<u>Thread B</u>
AllocateOrWait(1 MB)
AllocateOrWait(1 MB)
Free(1 MB)
Free(1 MB)

If only 2 MB of space, we get same deadlock situation

Deadlock with Locks: "Lucky" Case

Thread A

```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

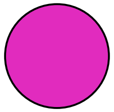
Thread B

```
y.Acquire();
x.Acquire();
…
x.Release();
y.Release();
```

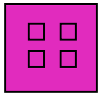
Sometimes schedule won't trigger deadlock

Resource-Allocation Graph

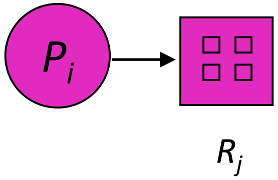
- Process



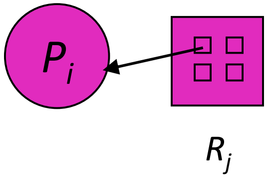
- Resource Type with 4 instances



- P_i requests instance of R_j

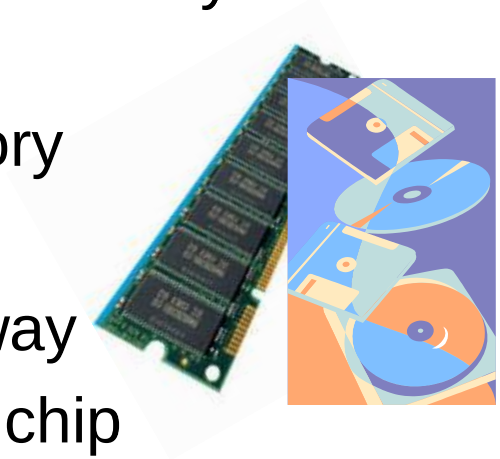


- P_i is holding an instance of R_j

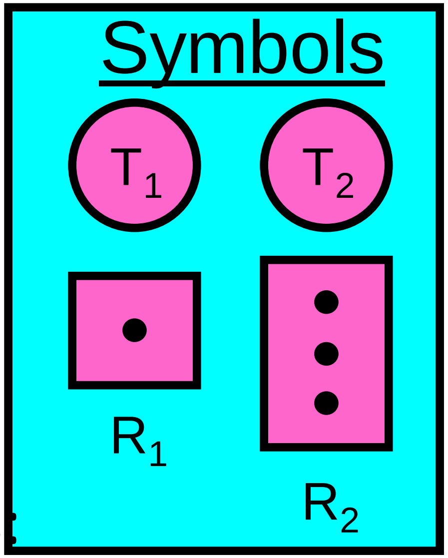


Resources

- Resources – passive entities needed by threads to do their work
  - CPU time, disk space, memory
- Two types of resources:
  - Preemptable – can take it away
    - CPU, Embedded security chip
  - Non-preemptable – must leave it with the thread
    - Disk space, printer, chunk of virtual address space
    - Critical section
- Resources may require exclusive access or may be sharable
  - Read-only files are typically sharable
  - Printers are not sharable during time of printing
- One of the major tasks of an operating system is to manage resources



Resource-Allocation Graph

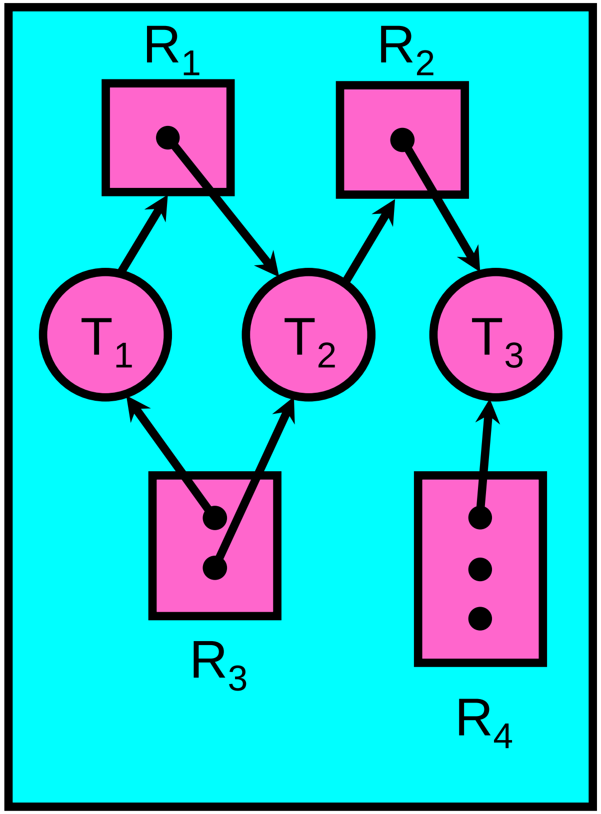


- System Model
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$

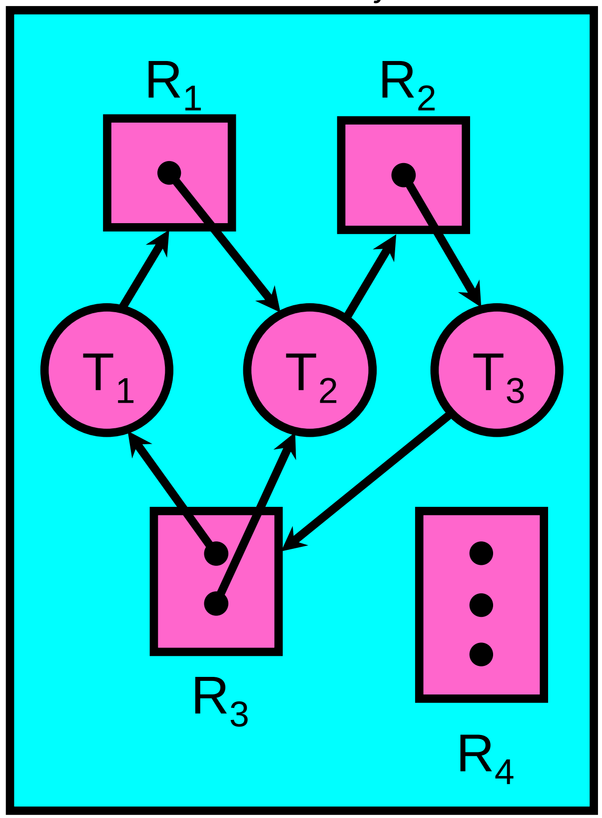
    CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances
  - Each thread utilizes a resource as follows:
    - `Request() / Use() / Release()`
- Resource-Allocation Graph:
  - V is partitioned into two types:
    - $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$

Resource-Allocation Graph Examples

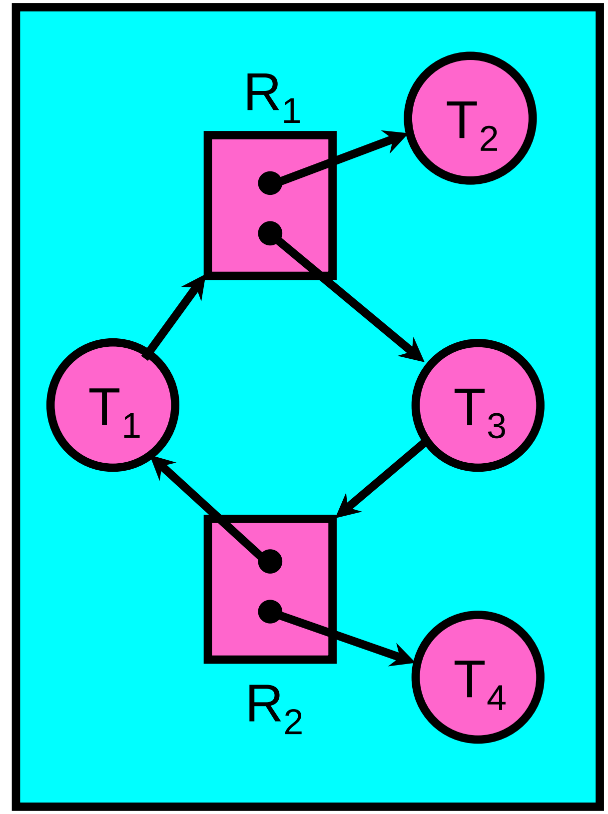
- Model:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph



Allocation Graph With Deadlock



Allocation Graph With Cycle, but No Deadlock

Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock.

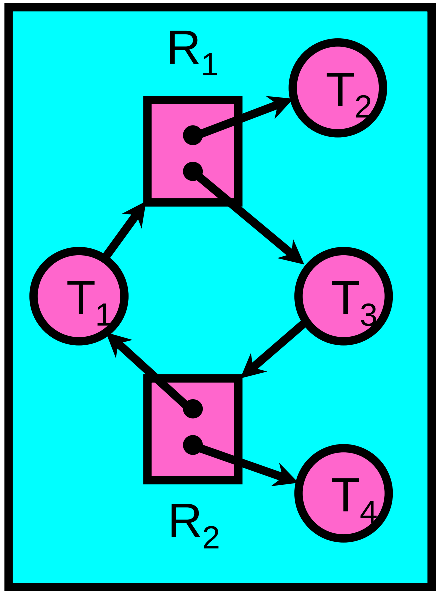
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Deadlock Detection Algorithm

- ⇒ look for loops
- More General Deadlock Detection Algorithm
  - Let [X] represent an m-ary vector of non-negative integers (quantities of resources of each type):

    `[FreeResources]:` Current free resources each type
    `[Request`$_X$`]:` Current requests from thread X
    `[Alloc`$_X$`]:` Current resources held by thread X
  - See if tasks can eventually terminate on their own

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
   done = true
   Foreach node in UNFINISHED {
      if ([Request_node] <= [Avail]) {
         remove node from UNFINISHED
         [Avail] = [Avail] + [Alloc_node]
         done = false
      }
   }
} until(done)
```



  - Nodes left in UNFINISHED ⇒ deadlocked

Four requirements for Deadlock

- Mutual exclusion
  - Only one thread at a time can use a resource.
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- Circular wait
  - There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads
    - T_1 is waiting for a resource that is held by T_2
    - T_2 is waiting for a resource that is held by T_3
    - …
    - T_n is waiting for a resource that is held by T_1

- **To prevent deadlock, make sure at least one of these conditions does not hold**

How should a system deal with deadlock?

- Three different approaches:

1. <u>Deadlock avoidance</u>: dynamically delay resource requests so deadlock doesn't happen
2. <u>Deadlock prevention</u>: write your code in a way that it isn't prone to deadlock
3. <u>Deadlock recovery</u>: let deadlock happen, and then figure out how to recover from it

- Modern operating systems:
  - Make sure the *system* isn't involved in any deadlock
  - Ignore deadlock in applications
    - "Ostrich Algorithm"

Deadlock Avoidance

- Idea: When a thread requests a resource, OS checks if it would result in deadlock
  - If not, it grants the resource right away
  - If so, it waits for other threads to release resources

THIS DOES NOT WORK!!!!

- Example:

Thread A	**Thread B**
`x.Acquire();`	`y.Acquire();`
Blocks… `y.Acquire();`	`x.Acquire();` Wait… But it's too late…
…	…
`y.Release();`	`x.Release();`
`x.Release();`	`y.Release();`

Deadlock Avoidance: Three States

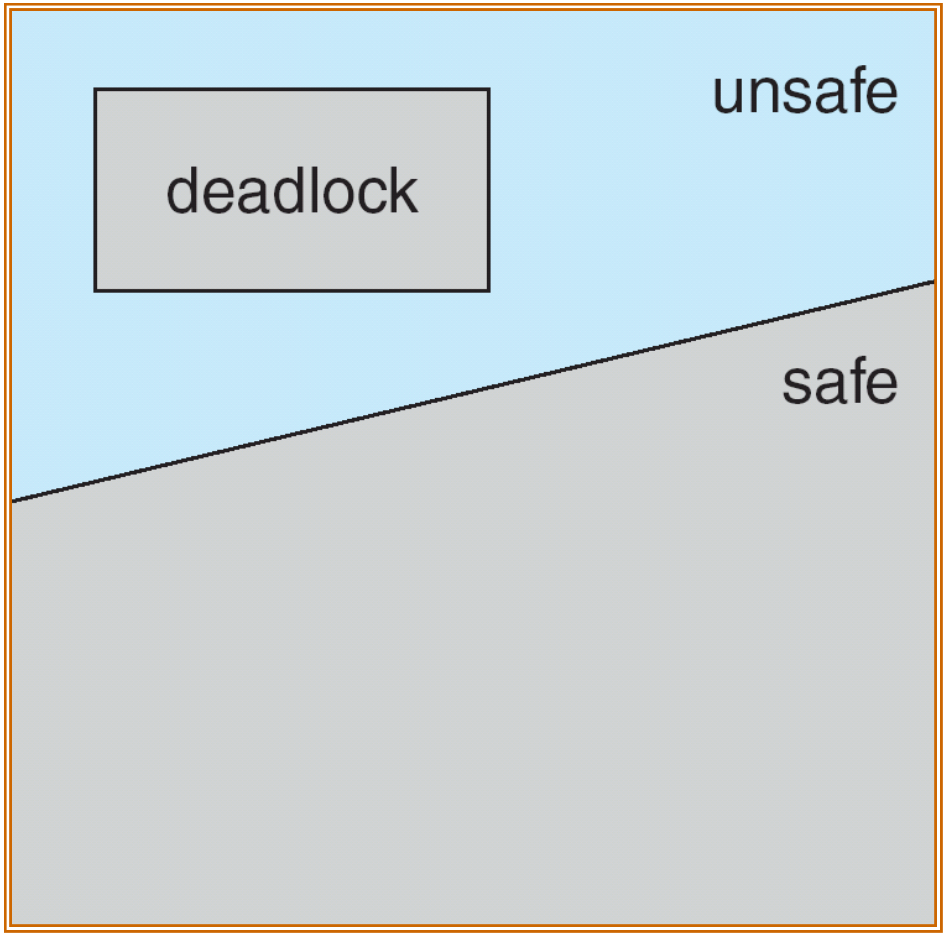
- Safe state
  - System can delay resource acquisition to prevent deadlock

Deadlock avoidance: prevent system from reaching an *unsafe* state

- Unsafe state
  - No deadlock yet…
  - But threads can request resources in a pattern that unavoidably leads to deadlock

- Deadlocked state
  - There exists a deadlock in the system
  - Also considered “unsafe”

Safe, Unsafe , Deadlock State



Deadlock Avoidance

- Idea: When a thread requests a resource, OS checks if it would result in ~~deadlock~~ an unsafe state
  - If not, it grants the resource right away
  - If so, it waits for other threads to release resources

- Example:

Thread A	Thread B
`x.Acquire();`	`y.Acquire();`
`y.Acquire();`	`x.Acquire();`
…	…
`y.Release();`	`x.Release();`
`x.Release();`	`y.Release();`

Wait until Thread A releases the mutex

Deadlock Avoidance with Resource Reservation

- Threads provide advance information about the maximum
- resources they may need during execution
- Define a sequence of threads $\{T_1, T_2, \ldots T_n\}$ as *safe* if for each T_i the resources that T_i can still request can be satisfied by the currently
- available resources plus the resources held by all T_j, $j < i$.
- A *safe state* is a state in which there is a safe sequence for the threads.
- An unsafe state is not equivalent to deadlock, it just may lead to deadlock, since some threads might not actually use the maximum resources they have declared.
- Grant a resource to a thread is the new state is safe
- If the new state is unsafe, the thread must wait even if the resource is currently available.
- This algorithm ensures no circular-wait condition exists.

Example

Threads t1, t2, and t3 are competing for 12 tape drives.

- Currently, 11 drives are allocated to the threads, leaving 1 available.
- The current state is *safe* (there exists a safe sequence, {t1, t2, t3} where all threads may obtain their maximum number of resources without waiting)

– t1 can complete with the current resource allocation

– t2 can complete with its current resources, plus all of t1's resources, and the unallocated tape drive.

- t3 can complete with all its current resources, all of t1 and t2's resources, and the unallocated tape drive.

	max need	in use	could want
t_1	4	3	1
t_2	8	4	4
t_3	12	4	8

Banker's Algorithm

- idea:
  - State maximum resource needs in advance
  - Allow particular thread to proceed if:

    (available resources - #requested) $\geq$ max remaining that might be needed by any thread

- Banker's algorithm
  - Allocate resources dynamically
    - Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - Keeps system in a "SAFE" state, i.e. there exists a sequence $\{T_1, T_2, \ldots T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..
  - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

Banker's Algorithm

- Technique: pretend each request is granted, then run deadlock detection algorithm, substitute

$$([Request_{node}] \leq [Avail]) \rightarrow ([Max_{node}]-[Alloc_{node}] \leq [Avail])$$

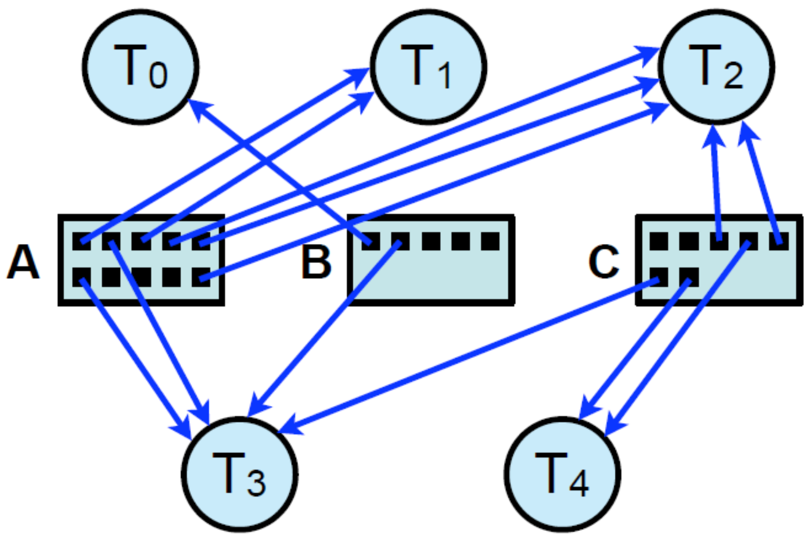
```
[FreeResources]:          Current free resources each type
   [Alloc_X]:             Current resources held by thread X
      [Max_X]:            Max resources requested by thread X

      [Avail] = [FreeResources]
   Add all nodes to UNFINISHED
   do {
      done = true
      Foreach node in UNFINISHED {
     if ([Max_node]–[Alloc_node]<= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc_node]
            done = false
         }
      }
   } until(done)
```

Example of Banker's Algorithm

- 5 Process/Treads T_0 through T_4;
  3 resource types:
  A (10 instances), *B* (5instances), and *C* (7 instances).

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

Example of Banker's algorithm



Non-allocated resource instances.

Available		
A	B	C
3	3	2

Allocated resource instances.

	Allocation		
	A	B	C
T_0	0	1	0
T_1	2	0	0
T_2	3	0	2
T_3	2	1	1
T_4	0	0	2

Example of Banker's algorithm

Available		
A	B	C
3	3 2	

	Max		
	A	B	C
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

	Need		
	A	B	C
T_0	7	4	3
T_1	1	2	2
T_2	6	0	0
T_3	0	1	1
T_4	4	3	1

	Allocation		
	A	B	C
T_0	0	1	0
T_1	2	0	0
T_2	3	0	2
T_3	2	1	1
T_4	0	0	2

Need = Max - Allocation

Example of Banker's algorithm

Available			Available		
A	B	C	A	B	C
3	3	2	2	1	0

	Max		
	A	B	C
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

	Need		
	A	B	C
T_0	7	4	3
T_1	1	2	2
T_2	6	0	0
T_3	0	1	1
T_4	4	3	1

	Allocation			Allocation		
	A	B	C	A	B	C
T_0	0	1	0	0	1	0
T_1	2	0	0	3	2	2
T_2	3	0	2	3	0	2
T_3	2	1	1	2	1	1
T_4	0	0	2	0	0	2

Allocation for T_1 increases and now T_1 holds its max (3, 2, 2).

Example of Banker's algorithm

Available		
A	B	C
3	3	2

	Max		
	A	B	C
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

	Need		
	A	B	C
T_0	7	4	3
T_1	1	2	2
T_2	6	0	0
T_3	0	1	1
T_4	4	3	1

	Allocation		
	A	B	C
T_0	0	1	0
T_1	2	0	0
T_2	3	0	2
T_3	2	1	1
T_4	0	0	2

Need = Max - Allocation

Example of Banker's algorithm

$\text{Available}_{before} + \text{Allocation}[T_1]_{before}$

(3, 3, 2) + (2, 0, 0) = (5, 3, 2)

	Before			Under			After		
	Available			**Available**			**Available**		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
	3	3	2	2	1	0	5	3	2

	Max		
	A	**B**	**C**
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

	Need		
	A	**B**	**C**
T_0	7	4	3
T_1	1	2	2
T_2	6	0	0
T_3	0	1	1
T_4	4	3	1

	Allocation			**Allocation**			**Allocation**		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
T_0	0	1	0	0	1	0	0	1	0
T_1	2	0	0	3	2	2	0	0	0
T_2	3	0	2	3	0	2	3	0	2
T_3	2	1	1	2	1	1	2	1	1
T_4	0	0	2	0	0	2	0	0	2

Example of Banker's algorithm

	Need			Allocation			
	A	B	C	A	B	C	
T_0	7	4	3	0	1	0	Done
T_1	1	2	2	2	0	0	Done
T_2	6	0	0	3	0	2	Done
T_3	0	1	1	2	1	1	Done
T_4	4	3	1	0	0	2	Done

	Available				
Step	A	B	C	Done	Choice
1	3	3	2	-	T_1
2	5	3	2	T_1	T_3
3	7	4	3	T_3	T_4
4	7	4	5	T_4	T_2
5	10	4	7	T_2	T_0
6	10	5	7	T_0	

sequence < T_1, T_3, T_4, T_2, T_0> that allowed all tasks to get their resources - **state is safe**!

Example of Banker's algorithm

	Need			Allocation			
	A	B	C	A	B	C	
T_0	7	2	3	0	3	0	Done
T_1	1	2	2	2	0	0	Done
T_2	6	0	0	3	0	2	Done
T_3	0	1	1	2	1	1	Done
T_4	4	3	1	0	0	2	Done

	Available				
Step	A	B	C	Done	Choice
1	3	1	2	-	T_3
2	5	2	3	T_3	T_1
3	7	2	3	T_1	T_0
4	7	5	3	T_0	T_2
5	10	5	5	T_2	T_4
6	10	5	7	T_4	

sequence < T_3, T_1, T_0, T_2, T_4> that allowed all tasks to get their resources - **state is safe!**

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

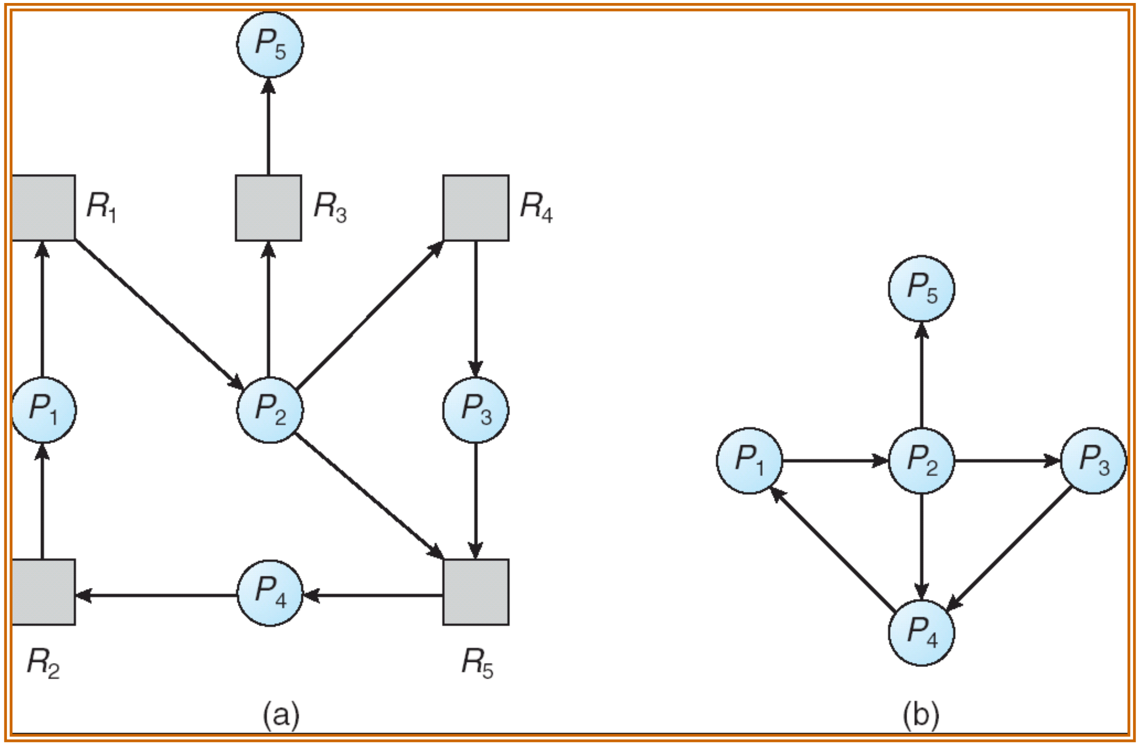
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain *wait-for* 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**:* 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.

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 1
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

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For *i* = 1,2, …, *n*, if $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

   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, $1 \leq i \leq n$, 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.