Deadlock

- Definition
- Motivation
- Conditions for deadlocks
- Deadlock prevention & detection

Deadlocks

- **Deadlock** = condition where multiple threads/processes wait on each other

process A

```
printer->wait();
disk->wait();
  do stuffs …
disk->signal();
printer->signal();
```

process B

```
disk->wait();
printer->wait();
 do stuffs …
printer->signal();
disk->signal();
```

Binary semaphore: printer, disk. Both initialized to be 1.

Deadlocks - Terminology

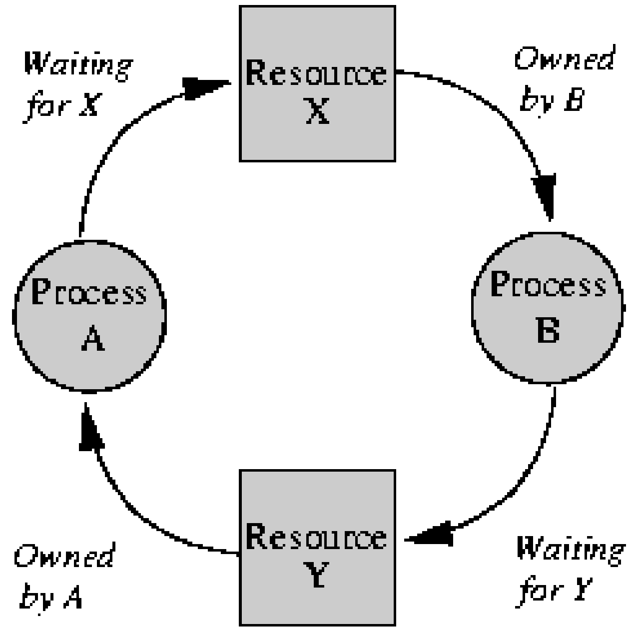
- **Deadlock**:
  - Can occur when several processes compete for finite number of resources simultaneously
- **Deadlock prevention** algorithms:
  - Check resource requests & availability
- **Deadlock detection**:
  - Finds instances of deadlock when processes stop making progress
  - Tries to recover
- **Note: Deadlock ≠ Starvation**

When Deadlock Occurs

All of below *must* hold:

1. **Mutual exclusion**:
   - An instance of resource used by one process at a time
2. **Hold and wait**
   - One process holds resource while waiting for another; other process holds that resource
3. **No preemption**
   - Process can only release resource *voluntarily*
   - No other process or OS can force thread to release resource
4. **Circular wait**
   - Set of processes $\{t_1, \ldots, t_n\}$: t_i waits on t_{i+1}, t_n waits on t_1

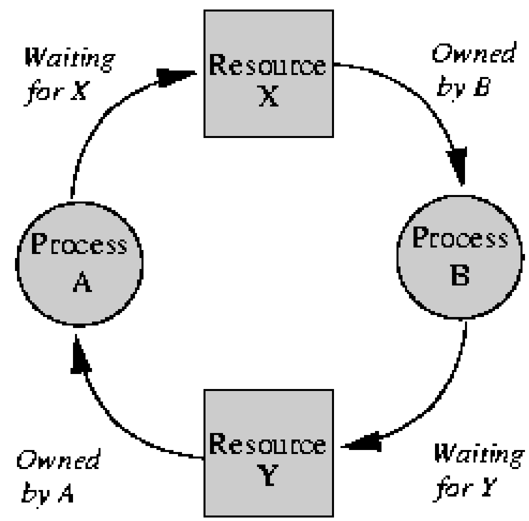
Deadlock: Example



- If no way to free resources *(preemption)*, deadlock

Deadlock Detection: Resource Allocation Graph

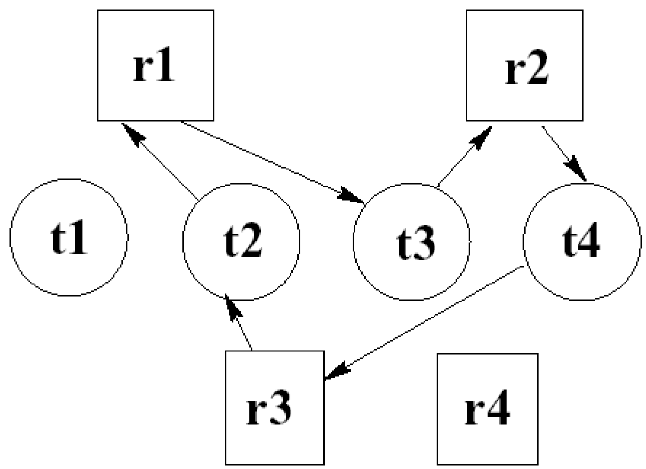
- Define graph with vertices:
  - Resources = $\{r_1, \ldots, r_m\}$
  - Processes/threads = $\{t_1, \ldots, t_n\}$
- *Request edge* from process to resource
  $t_i \rightarrow r_j$
  - Process requested resource but not acquired it
- *Assignment edge* from resource to process
  $r_j \rightarrow t_i$
  - OS has allocated resource to process
- Deadlock detection
  - No cycles → no deadlock
  - Cycle → might be deadlock



Resource Allocation Graph: Example

- Deadlock or not?

- *Request edge* from process to resource $t_i \rightarrow r_j$
  - Process requested resource but not acquired it
- *Assignment edge* from resource to process $r_j \rightarrow t_i$
  - OS has allocated resource to process

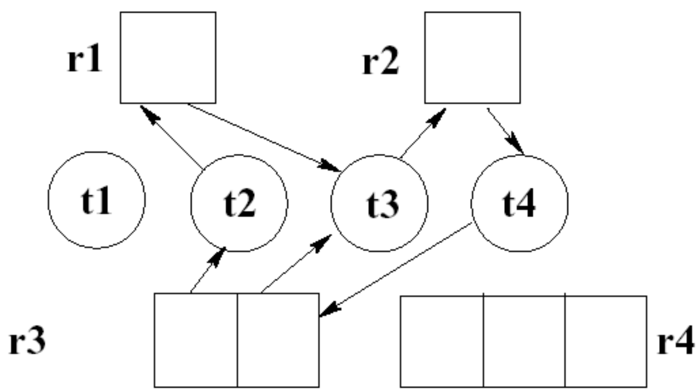


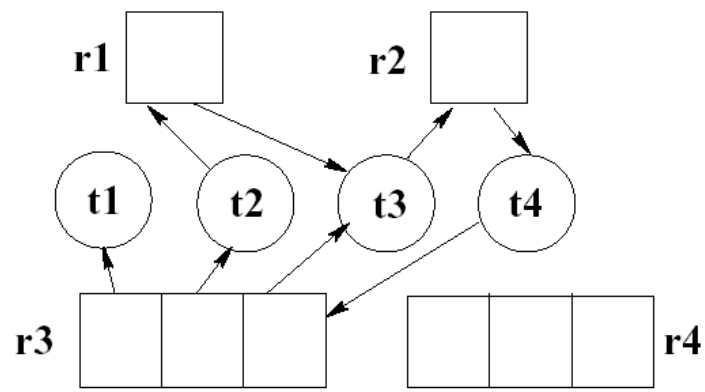
Deadlock Detection: Multiple Instances of Resource

- What if there are *multiple instances* of a resource?
  - Cycle → deadlock *might* exist
  - If any instance held by process outside cycle, progress is possible when process releases resource

Deadlock Detection

- Deadlock or not?

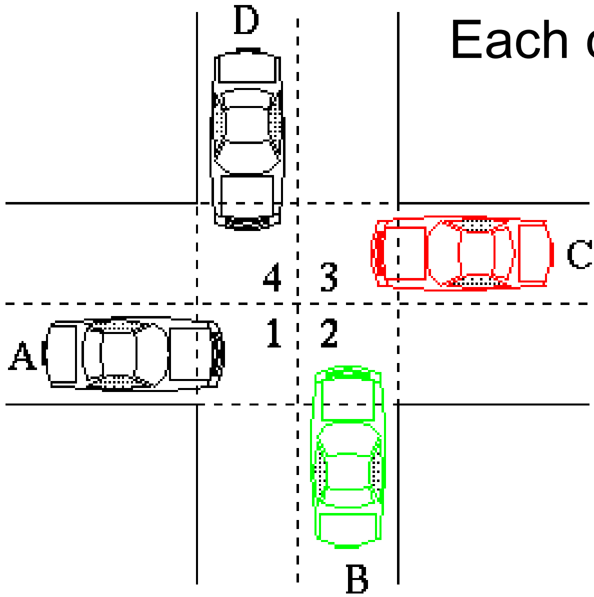




Resource Allocation Graph: Example

- Draw a graph for the following event:

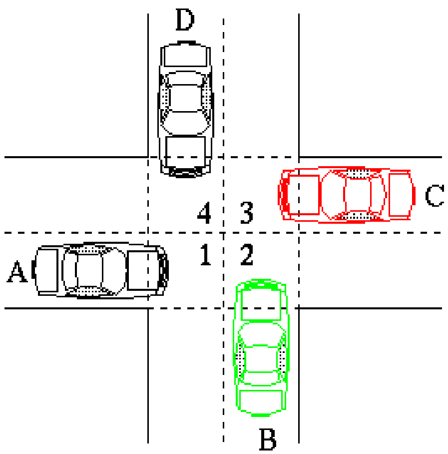
- *Request edge* from process to resource $t_i \rightarrow r_j$
  - Process: requested resource but not acquired it
- *Assignment edge* from resource to process $r_j \rightarrow t_i$
  - OS has allocated resource to process

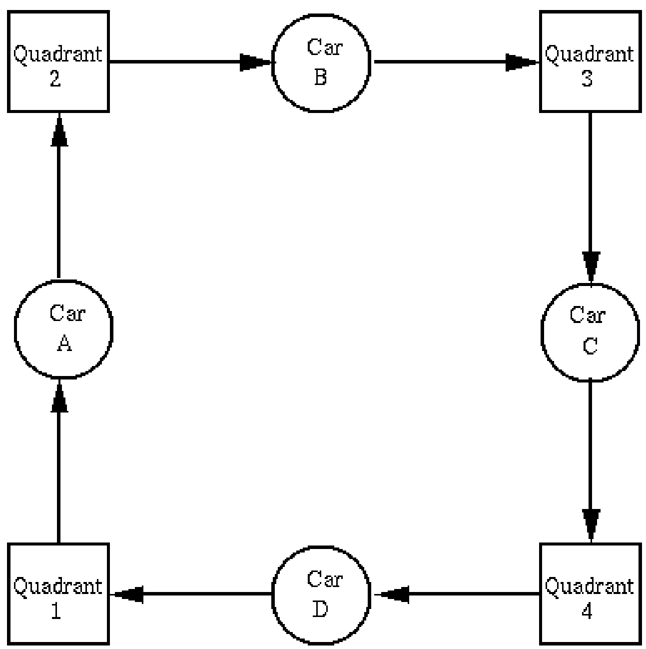


Each car going straight

Resource Allocation Graph : Example

- Draw a graph for the following event:





Detecting & Recovering from Deadlock

- Single instance of resource
  - Scan resource allocation graph for cycles & break them!
  - Detecting cycles takes $O(n^2)$ time
    - DFS with back edge
    - $n = |T| + |R|$
  - When to detect:
    - When request cannot be satisfied
    - On regular schedule, e.g. every hour
    - When CPU utilization drops below threshold

Detecting & Recovering from Deadlock (cont'd)

- How to recover? - break cycles:
  - Kill all processes in cycle
  - Kill processes one at a time
    - Force each to give up resources
  - Preempt resources one at a time
    - Roll back thread state to before acquiring resource
    - Common in database transactions
- Multiple instances of resource
  - No cycle → no deadlock
  - Otherwise, check whether processes can proceed

Deadlock Prevention

- Ensure at least one of necessary conditions doesn't hold
  - **Mutual exclusion**
  - **Hold and wait**
  - **No preemption**
  - **Circular wait**

Deadlock Prevention

- **Mutual exclusion**:
  - Make resources shareable (but not all resources can be shared)
- **Hold and wait**
  - Guarantee that process cannot hold one resource when it requests another
  - Make processes request all resources they need at once and release all before requesting new set

Deadlock Prevention, continued

- **No preemption**
  - If process requests resource that cannot be immediately allocated to it
    - OS preempts (releases) all resources the process currently holds
  - When all resources available:
    - OS restarts the process

- *Problem*: not all resources can be preempted

Deadlock Prevention, continued

- **Circular wait**
  - Impose ordering (numbering) on resources and request them in order

Deadlock Prevention with Resource Reservation

- With future knowledge, we can prevent deadlocks:
  - Processes provide advance information about maximum resources they may need during execution
- Resource-allocation *state*:
  - Number of available & allocated resources, maximum demand of each process

Deadlock Prevention with Resource Reservation (cont'd)

- Main idea: grant resource to process if new state is *safe*
  - Define sequence of processes $\{t_1, \ldots, t_n\}$ as *safe:*
    - For each t_i, the resources that t_i can still request can be satisfied by currently available resources plus resources held by all t_j, $j < i$
  - *Safe state* = state in which there is safe sequence containing all processes
- If new state unsafe:
  - Process waits, even if resource available

Guarantees no circular-wait condition

Resource Reservation Example 1

- Processes t_1, t_2, and t_3
  - Competing for 12 tape drives
- Currently 11 drives allocated
- Question: is current state safe?

- t_1 can complete with current allocation
- t_2 can complete with current resources, + t_1's resources & unallocated tape drive
- t_3 can complete with current resources, + t_1's and t_2's, & unallocated tape drive

- Yes: there exists safe sequence $\{t_1, t_2, t_3\}$ where all processes may obtain maximum number of resources without waiting

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

Resource Reservation Example II

- If t1 requests one more drive:
  - Should OS grant it?

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

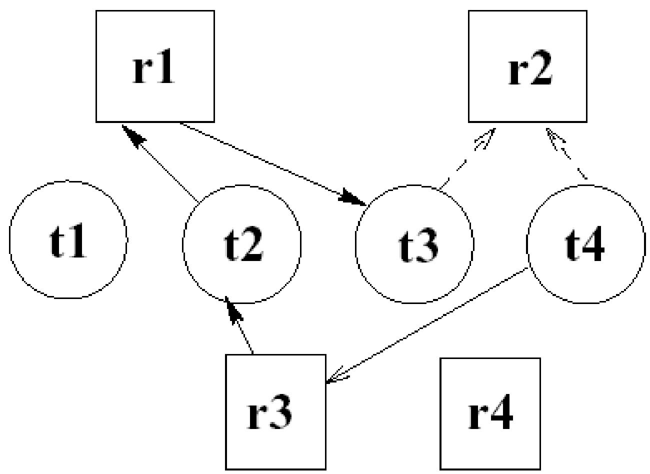
Resource Reservation Example III

- If t3 requests one more drive:
  - Must wait because allocating drive would lead to unsafe state: 0 available drives, but each thread might need at least one more drive

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

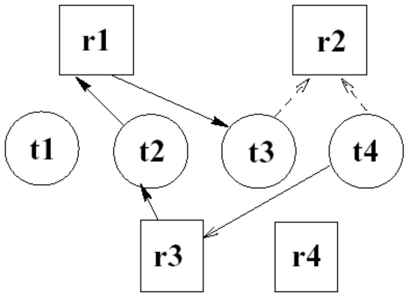
Single-Instance Resources: Deadlock Avoidance via Claim Edges

- Add *claim edges*:
  - Edge from process to resource that may be requested in future

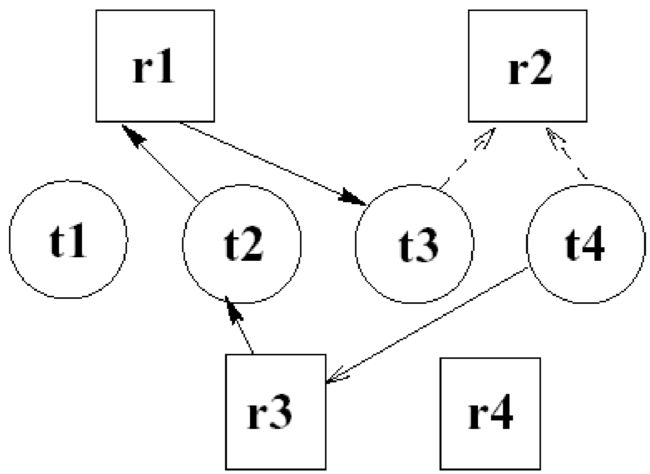


Single-Instance Resources: Deadlock Avoidance via Claim Edges (cont'd)

- To determine whether to satisfy a request:
  - convert claim edge to allocation edge
  - No cycle: grant request
  - Cycle: unsafe state; Deny allocation, convert claim edge to request edge, block process



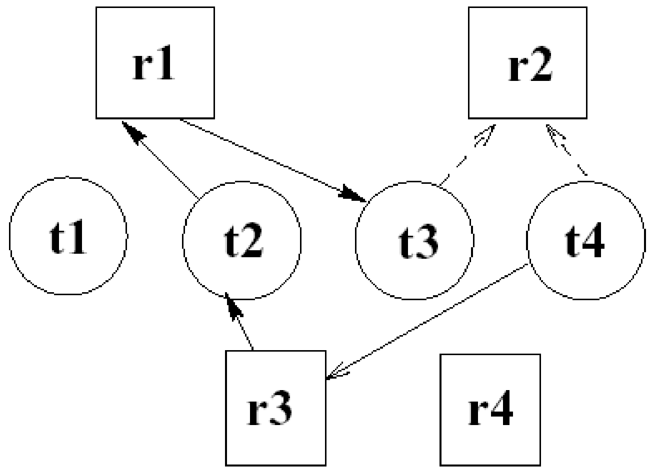
Single-Instance Resources: Deadlock Avoidance via Claim Edges (cont’d)



resource-allocation graph at time T

Q1: suppose t3 requests r2 at time T1 (T1>T), should OS grant it?

Single-Instance Resources: Deadlock Avoidance via Claim Edges (cont'd)



resource-allocation graph at time T

Q2: suppose t4 requests r2 at time T1 (T1>T), should OS grant it??

Banker's Algorithm

- Multiple instances
  - Each process must a priori claim maximum use
  - When a process requests 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* 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) $\boldsymbol{Need_i \le Work}$

   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

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$_i$*** ≤ ***Need$_i$*** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If ***Request$_i$*** ≤ ***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 ⇒ the resources are allocated to ***P$_i$***
- If unsafe ⇒ ***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* (5instances), 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 < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?