Deadlocks

- What are deadlocks?
- System model
- Characterizing deadlocks
- Methods for dealing with deadlocks
  - » Deadlock prevention
  - » Deadlock avoidance
  - » Deadlock detection
- Recovering from deadlock
- Deadlock handling in the “real world”

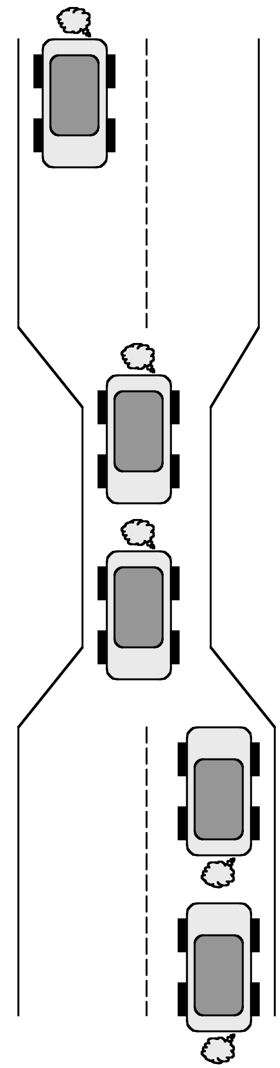
What's a Deadlock?

- Deadlock occurs when each of a set of processes holds a resource and is blocked waiting to acquire a resource held by another process in the set
- Example 1:
  - » A system has a disk drive and a tape drive
  - » P_0 wants to read from tape to disk, so it holds the tape drive and waits for the disk to be available
  - » P_1 wants to read from disk to tape, so it holds the disk and waits for the tape drive to be available
- Example 2 (semaphores A & B initialized to 1)

P_0	P_1
`A.Wait();`	`B.Wait();`
`B.Wait();`	`A.Wait();`

Deadlock on a One-Lane Bridge

- One lane on bridge
- Each bridge section can be viewed as a resource
  - » One section per car
- Deadlock resolution: back up cars
  - » Preemption
  - » Rollback
  - » May need to back up more than one car
- Starvation
  - » Some cars never get to cross



Resources in a System

- System has classes of resources R_0, R_1, R_2, …, R_n
  - » CPU cycles
  - » Pages of memory
  - » I/O devices (printer, tape, disk)
- Each class R_i has W_i instances
  - » Example: class “disk” has 4 instances (4 disks)
  - » Example: class “memory page” has 512 instances (512 pages)
  - » Instances are fully interchangeable (a process can use any free instance in a resource class)
- Processes can use resources by:
  - » Requesting the resource
  - » Using the resource
  - » Releasing the resource

Conditions Necessary for Deadlock

- Mutual exclusion
  - » Only one process at a time can use a resource
- Hold & wait
  - » A process holding at least one resource is waiting to acquire additional resources held by other processes
- No preemption
  - » A resource can only be released 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 held by P_1
    P_1 is waiting for a resource held by P_2, and so on up to
    P_n which is waiting for a resource held by P_0

Example: Dining Philosophers

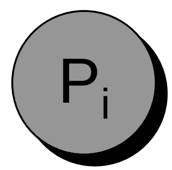
- Each philosopher picks up the chopstick on his/her right
- Each philosopher waits for the chopstick on his/her left to become available
- Deadlock occurs
  - » Mutual exclusion: a chopstick can only be held by one person
  - » Hold & wait: a person holds a chopstick while waiting for another
  - » No preemption: philosophers don't put a chopstick down if they can't get both
  - » Circular wait: P0 waits for P1, which waits for P2, …, which waits for P6, which waits for P0

Resource Allocation Graph

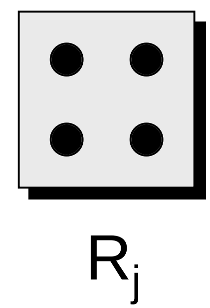
- Use a graph to represent resource usage by processes
  - » Graph consists of a set V of vertices & a set E of edges
  - » Edges connect vertices
- Vertices consist of two types
  - » $P = \{P_0, P_1, \ldots, P_n\}$: one vertex for each process
  - » $R = \{R_0, R_1, \ldots, R_m\}$: one vertex for each class of resources
- Edges are one of two types
  - » Request edge: directed edge from P_i to R_j
  - » Assignment edge: directed edge from R_j to P_i

Vertices in a Resource Allocation Graph

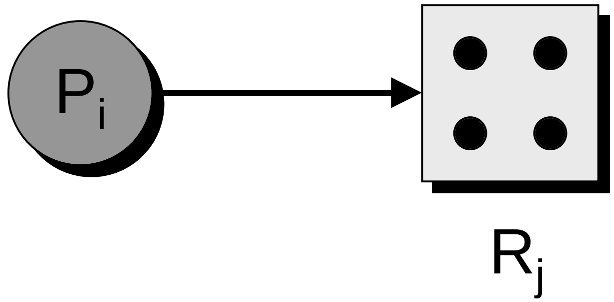
Process



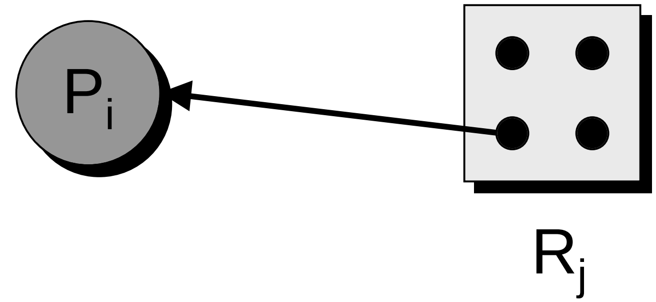
Resource with 4 instances



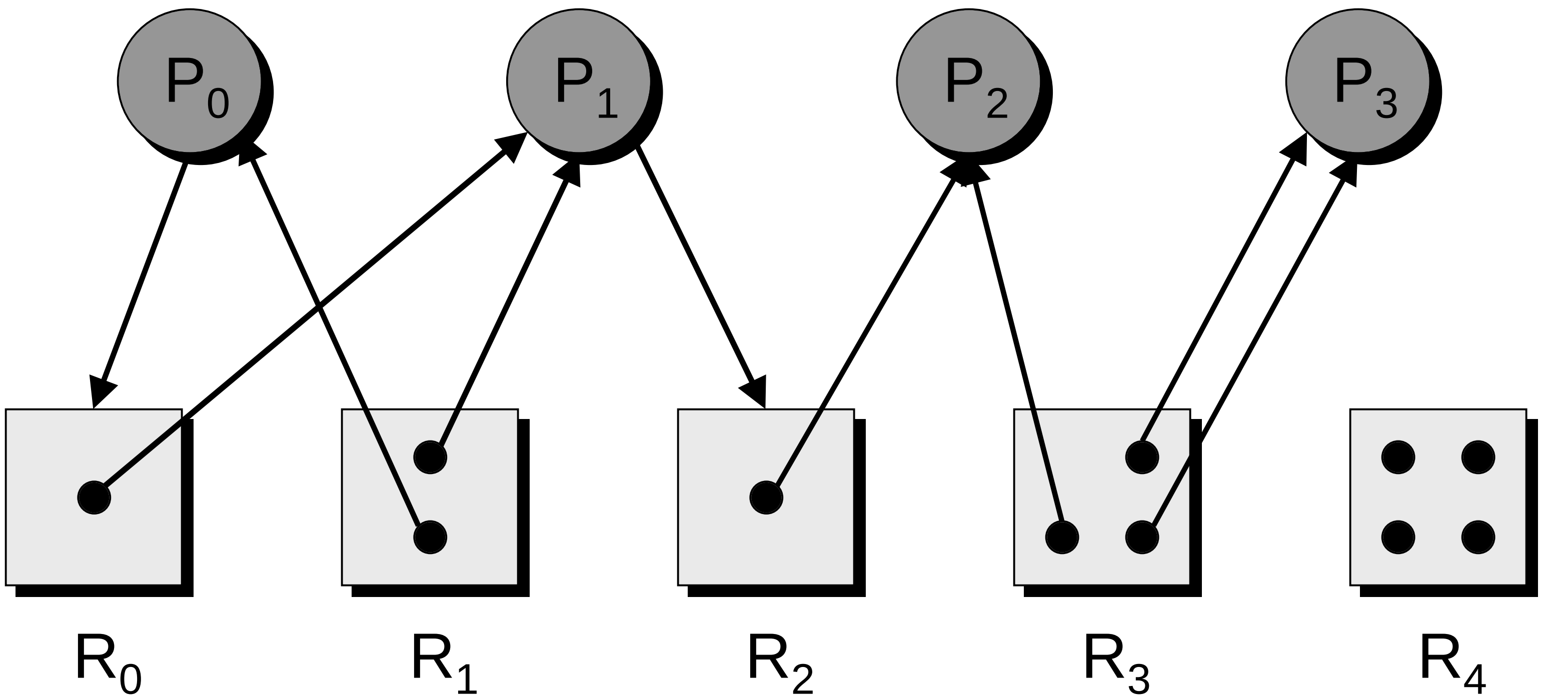
P_i requests instance of R_j



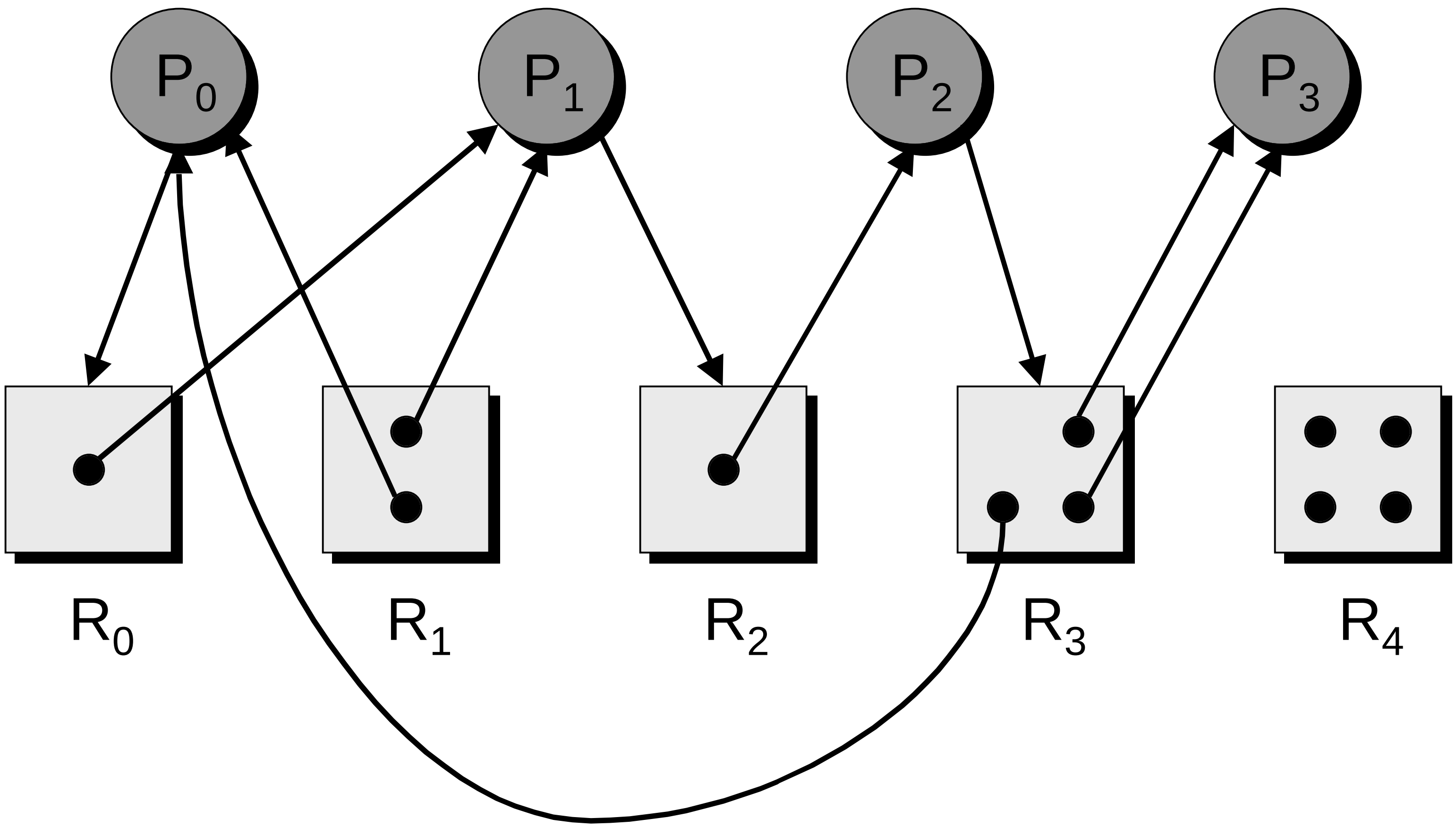
P_i holds an instance of R_j



Resource Graph Without Cycles



Resource Graph With a Cycle



Resource Graph Cycles and Deadlock

- If the resource graph contains no cycles => no deadlock
- If the graph contains a cycle
  - » If each resource class has exactly one instance => deadlock
  - » If some (or all) resource classes have more than one instance => deadlock possible (but not certain)
- Detecting deadlock: can processes run and release their resources to eliminate the cycle in the graph?
  - » If yes, there's no deadlock
  - » If no, there's deadlock
- Possibility: no deadlock yet, but deadlock may happen shortly

Dealing With Deadlocks

- Ensure the system will *never* enter a deadlock state
- Allow system to enter deadlock state and then recover
  - » Kill off processes causing the deadlock
  - » Preempt processes, taking their resources away
- Ignore deadlock problem (maybe it'll go away…)
  - » Pretend that deadlocks never happen
  - » Used by most operating systems because
    - – Deadlocks are rare : resources are plentiful
    - – Detecting / preventing them is time-consuming

Preventing Deadlock

- Ensure that at least one of the 4 conditions for deadlock can never occur
- Mutual exclusion
  - » Use sharable resources
  - » Not practical for resources that can’t be shared
- Hold and wait: guarantee that processes waiting for resources don’t hold any while they wait
  - » Require a process to acquire all of its resources at once
    - – When process starts
    - – At any time when it doesn’t hold any resources
  - » May lead to starvation and/or low resource utilization
    - – All-or-nothing approach tends to lead to “nothing”
    - – Many processes held up by a few processes using “popular” resources

Preventing Deadlock (continued)

- No preemption: allow preemption to eliminate deadlock
  - » Release all of a process' resources if it fails to acquire a resource that it requests
  - » Add released resources to the list of "needed resources" for that process
  - » Restart process only when it can gain all of the resources it needs, including those preempted
  - » Problem: process may be in the middle of using some of the resources
- Circular wait: eliminate cycle in the resource graph
  - » Impose an order on all resource classes
  - » Require that processes request resources in fixed order (same order for all processes)

Dining Philosophers & Circular Wait

- For simple solution to deadlock in Dining Philosophers, order all resources (chopsticks)
  - » Label chopsticks 0 through *n*-1
  - » Require that philosopher grab his/her lower-numbered chopstick first
- Avoids circular wait by eliminating cycle in resource graph
  - » Cycle requires both
    - – Philosopher waiting for chopstick *a*, holding *b* ($a > b$)
    - – Philosopher waiting for chopstick *c*, holding *d* ($d > c$)
  - » Otherwise, no cycle is possible
  - » Request lower number first => no deadlock

Avoiding Deadlock

- Requires that the system have some additional information about each process
  - Simple model: process declares *maximum number* of each class of resources that it may need
  - Requests for more resources than the maximum are automatically denied
- Deadlock avoidance algorithm examines the resource allocation state to ensure that there's never a circular wait
  - Done before granting each request
  - Not done when process declares maximum resources requested
- Resource allocation state includes:
  - Available & allocated resource amounts
  - Maximum demands of each process

Safe State

- Granting a resource request must leave the system in a safe state
  - » Safe state is one in which there exists at least one safe sequence of processes
  - » Safe sequence is an order that allows all processes to finish
- Sequence <P_0, P_1, …, P_n> is safe if, for each P_i, the resources that P_i can still request may be satisfied by currently available resources + resources held by previously completed processes (P_j , $j < i$)
  - » P_i can wait until all previous processes have finished if current resources aren't sufficient
  - » P_i gets its resources, executes, frees all resources, and exits
  - » P_{i+1} can run after P_i terminates, freeing all its resources

Safe State & Deadlock

- System is in a safe state:
  - » Deadlock cannot occur
  - » Future resource requests could cause the system to move into an unsafe state
- System is in an unsafe state:
  - » Deadlock is possible
  - » Deadlock isn’t necessarily certain
    - – Processes might not request all of the resources they’ve “reserved”
    - – Processes might release some of the resources they already hold before requesting more
- Deadlock avoidance: ensure that the system will never enter an unsafe state

Banker’s Algorithm

- Allows multiple instances of any resource class
- Each process must state its maximum resource usage before starting
- A process requesting a resource may have to wait
- A process that acquires resources must release them in a finite amount of time (no infinite loops)
- Called the Banker’s Algorithm because it can be used by a bank to make sure that the bank never runs out of cash (presumably a limited resource)

Data Structures for the Banker's Algorithm

- Basic definitions
  - » *n* = number of processes
  - » *m* = number of resource classes
- Data structures
  - » `int available[m];`
    if available[j] == k, there are *k* instances of resource R_j available
  - » `int max[n][m];`
    If max[i][j]==k, then process P_i may request at most *k* instances of resource R_j
  - » `int allocation[n][m];`
    If allocation[i][j]==k, then process P_i currently has *k* instances of resource R_j
  - » `Int need[n][m];`
    If need[i][j]==k, then process P_i may need *k* more instances of R_j
- Invariant: `need[i][j] == max[i][j]-allocation[i][j]`

Safety Algorithm

- Define `work[m]` and `finish[n]`
- Initialize `work[] = AVAIL` and `finish[] = FALSE`
- Repeat while, for some *i*, `finish[i]==FALSE`
  - » Find an *i* such that both:
    - – `finish[i] == FALSE`
    - – `need[i][j] <= work[j]` for all *j*
  - » If no such *i* exists, exit the repeat loop
  - » `finish[i] = TRUE`
  - » For all j:
    - – `work[j] += allocation[i][j]`
- If `finish[i]==TRUE` for all *i*, the system is in a safe state

Resource Request from Process P_i

- $\texttt{Request}_i$ is the request vector for process P_i
- If $\texttt{Request}_i\texttt{[j] == }$ k, then process P_i wants *k* instances of resource class R_j
- If $\texttt{Request}_i \texttt{ > Need}_i$, process wants more than its max
- If $\texttt{Request}_i \texttt{ > Available}$, process must wait for more resources to become available
- “Allocate” resources to P_i by modifying the state by:

  $\texttt{Available -= Request}_i$

  $\texttt{Allocation}_i \texttt{ += Request}_i$

  $\texttt{Need}_i \texttt{ -= Request}_i$

  » If this is a safe state, the resources are allocated and the temporary changes above are made permanent

  » If this is not a safe state, P_i must wait and the previous allocation state is restored

Banker's Algorithm Example

- System contains:
  - » 5 processes (P_0 through P_4)
  - » 3 resources (A: 8 instances, B: 9 instances, C: 3 instances)
- Need is defined as `Max - Allocation`
- Safe state: processes run in order P1, P4, P0, P2, P3
  - » Other orderings are possible
- Initial snapshot:

P	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	3	0	1	6	4	3	3	4	2	2	3	1
P1	0	1	1	1	3	2	1	2	1			
P2	2	1	0	6	5	0	4	4	0			
P3	0	4	0	3	9	3	3	5	3			
P4	1	0	0	2	2	2	1	2	2			

Banker's Algorithm, continued

- Process P4 requests (1,1,0)
  - » Check to see if resources available
  - » Check to see if we're still in a safe state
  - » Yes: P1, P4, P0, P2, P3
- Consider system before P4 request
  - » Can P2 request (1,1,0)?

P	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	3	0	1	6	4	3	3	4	2	1	2	1
P1	0	1	1	1	3	2	1	2	1			
P2	2	1	0	6	5	0	4	4	0			
P3	0	4	0	3	9	3	3	5	3			
P4	2	1	0	2	2	2	0	1	2			

Detecting Deadlock

- Deadlock prevention can be slow and difficult
- Rather than prevent deadlock, allow it to happen
  - » Deadlocks are infrequent
  - » Deadlocks can be “backed out” if they’re detected
- To do this, we need
  - » Deadlock detection algorithm
  - » Deadlock recovery mechanisms

Single Instances of Each Resource

- Simple case: one instance of each resource
- Solution: keep a “wait-for” graph
  - » Each process is a node
  - » Directed edge from P_i to P_j indicates that P_i is waiting for P_j to release a resource
- Periodically invoke an algorithm that searches for cycles in the graph
  - » Don’t need to invoke each time a resource is used
  - » Invocation frequency depends on how urgently a deadlock must be dealt with
- Graph algorithm to detect cycles requires $O(n^2)$ operations, where n is the number of vertices (processes) in the graph

Multiple Instances of Resources

- Normal case: several instances of one or more resources
- Data structures required:
  - » Available: vector of length *m* indicating the number of available resources of each class
  - » Allocation: an *n* x *m* matrix holding the number of resources of each type currently held by each process
  - » Request: an *n* x *m* matrix indicating the current request of each process

Deadlock Detection Algorithm

- Define
  - » `Work[m] = Available`
  - » `Finish[i] = false` if any `Allocation[i][j] != 0`
- Repeat until done
  - » Find an index *i* such that both:
    - – `Finish[i] == false`
    - – `Request[i] <= Work[i]`
    - – If none is found, the system is deadlocked, and the processes that haven't yet finished are the ones causing it
  - » Process *i* found in the previous step can finish
    - – `Work += Allocation`
    - – `Finish[i] = TRUE`
- Algorithm requires $O(m \times n^2)$ operations to detect deadlock

Deadlock Detection Example

- System contains:
  - 5 processes (P_0 through P_4)
  - 3 resources (A: 7 instances, B: 2 instances, C: 6 instances)
- Sequence P0, P2, P3, P1, P4 will allow all processes to finish

P	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

Deadlock Detection Example, continued

- P2 requests another instance of resource C
  - » P0 can still finish, returning its resources to the pool
  - » P1, P2, P3, P4 are deadlocked
- How can the system recover from a deadlock?

P	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	1			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

Using Deadlock Detection

- How frequently (and when) to invoke depends on:
  - » Frequency with which deadlocks occur
  - » Number of processes that need to be rolled back (forced to give up their resources and restart from an earlier point)
- If deadlock detection is invoked arbitrarily, it’s hard to find the “keystone” for the deadlock
  - » Try to roll back as few processes as possible
  - » Many cycles in graph might be undone by rolling back just one or two processes, undoing the “log jam”

Recovering from Deadlock: Termination

- Terminate all deadlocked processes
  - » Drastic, but fast and guaranteed to work
- Abort processes one at a time until the deadlock is gone
  - » Slower, but less disruptive
- How do we choose processes to kill?
  - » Process priority (keep more important processes)
  - » Execution time or time to completion (conserve CPU time)
  - » Other resources used by the process
  - » Other resources needed to complete process execution
  - » Number of processes that must be aborted (the fewer the better, usually)
  - » Other factors (interactive/batch, user running the process, interrelation of processes)

Recovery from Deadlock: Preemption

- Rather than kill a process, steal some (or all) of its resources
  - » Computation not wasted
  - » Process may be able to proceed in a limited way
- To select a victim, minimize cost (as with termination)
- Process may need to be “rolled back”
  - » Free up resources
  - » Restart execution at point just before resources were requested
- Potential problems with preemption
  - » Starvation: same process may always be victimized
  - » Code complexity: allowing roll back isn’t always easy

Deadlock Handling

- Combine basic approaches
  - » Prevention: very conservative, but guaranteed to work
  - » Avoidance: less restrictive than prevention, but requires *a priori* knowledge from processes
  - » Detection: useful if deadlocks are unlikely because the overhead of prevention and avoidance is too high
- Use different approaches for different resources
  - » Printer: prevention
  - » Memory pages: detection
  - » Disk space: avoidance