Deadlocks

Reading:

Silberschatz

chapter 8

Additional Reading:

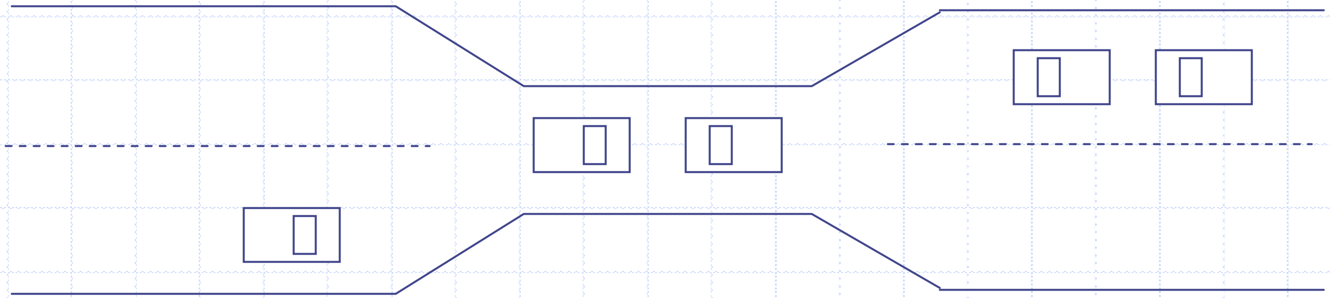
Stallings

chapter 6

Outline

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
  - Safe State
  - Resource Allocation Graph Algorithm
  - Bankers Algorithm
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

Real-life Example



- Bridge traffic can only be in one direction
- Each entrance of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible

The Deadlock Problem

- A set of process → Deadlock state
  - When every process in the set is waiting for an event that can be caused only by another process in set
- Examples
  - Space is available for allocation of 200Kbytes
  - Following sequence of events occur

```
P1
…
Request 80 Kbytes;
…
Request 60 Kbytes;
```

```
P2
…
Request 70 Kbytes;
…
Request 80 Kbytes;
```

Deadlock Example

- Deadlock occurs if receive is blocking

```
P1
. . .
Receive(P2);
. . .
Send(P2, M1);
```

```
P2
. . .
Receive(P1);
. . .
Send(P1, M2);
```

- Design Errors → Deadlocks
  - May be quite subtle and difficult to detect
  - Require rare combination of events → Deadlock
  - Considerable time, may be years to detect the problem

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);
   }
```

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 and 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**
  - Set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes
  - $P_0 \rightarrow P_1$, $P_1 \rightarrow P_2$, …, $P_{n-1} \rightarrow P_n$, and $P_n \rightarrow P_0$

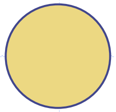
Resource-Allocation Graph

V → Set of vertices; E → Set of edges

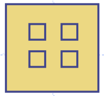
- V is partitioned into two types
  - $P = \{P_1, P_2, \ldots, P_n\}$, set of *all the processes*
  - $R = \{R_1, R_2, \ldots, R_m\}$, set of *all the resource types*
- Request edge – directed edge $P_i \rightarrow R_j$
- Assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph

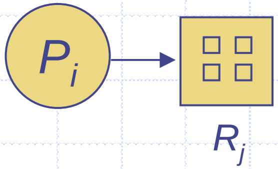
- Process



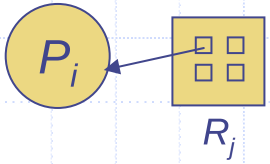
- Resource type with 4 instances



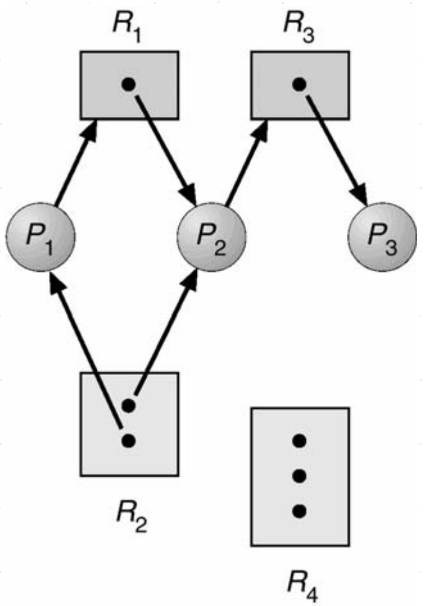
- P_i requests an instance of R_j



- P_i is holding an instance of R_j

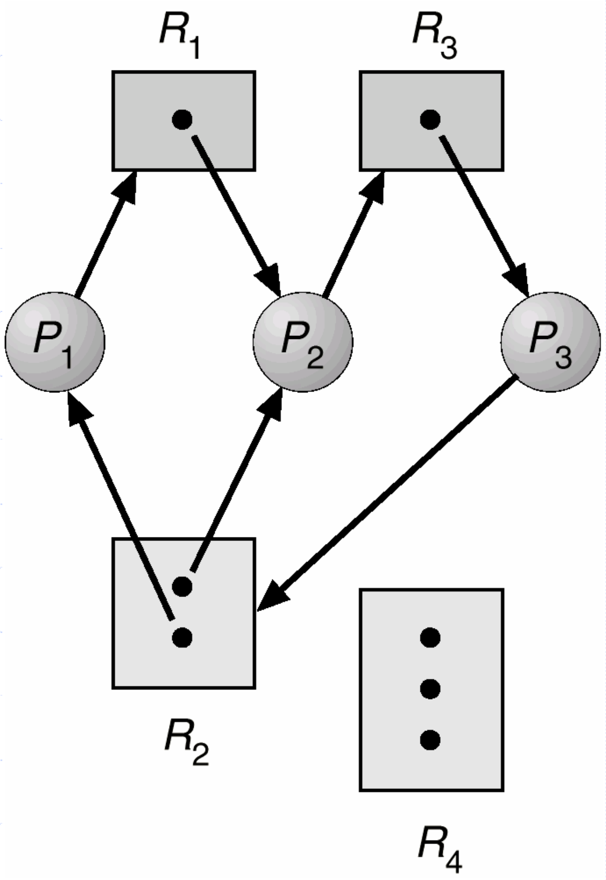


Resource Allocation Graph



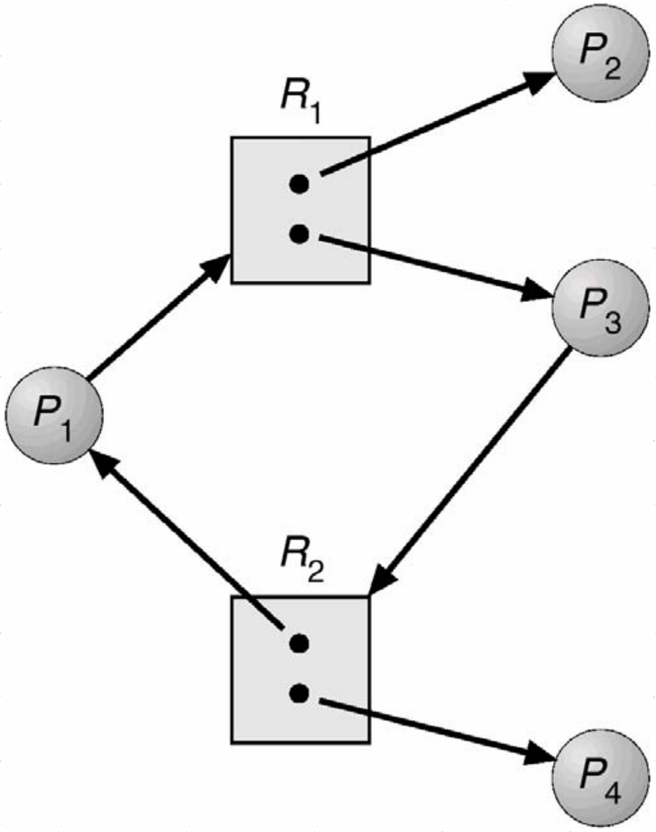
- No Cycles → No Deadlock
- If there is a cycle
  - Resource type has exactly one instance → Deadlock
  - Resource type has several instances → *may or may not* be a Deadlock

Resource Allocation Graph



Deadlock?

Resource Allocation Graph



Deadlock?

Methods for Handling Deadlocks

- **Deadlock Prevention**
  - Ensure that *at least one* of four necessary conditions <u>cannot hold</u>

- **Deadlock Avoidance**
  - Do not allow a resource request → Potential to lead to a deadlock
  - Requires advance info of all requests

- **Deadlock Detection**
  - Always allow resource requests
  - Periodically check for deadlocks
  - If a deadlock exists → Recover from it

- **Ignore**
  - Makes sense if the likelihood is very low, say once per year
  - Cheaper than *prevention*, *avoidance* or *detection*
  - Used by most common OS

Prevention Vs Avoidance

- **Deadlock Prevention** (*Traffic Light*)

  - preventing deadlocks by constraining how requests for the resources can be made in system and how they are handled; designing the system.

  - The goal is to ensure that at least one of the necessary conditions cannot hold.

- **Deadlock Avoidance** (*Traffic Policeman*)

  - The system dynamically considers every request at every point and decides whether it is safe to grant the request.

  - The OS requires advance additional information concerning which resources a process will request and use during its lifetime.

Deadlock Prevention

Restrain the ways request can be made;

- **Mutual Exclusion**
  - Allow everybody to use the resources immediately they require!
  - Unrealistic in general, printer output interleaved with others?
- **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
  - *Low resource utilization*, *Starvation possible*

Deadlock Prevention

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*
- *Not realistic* for many types of resources, such as *printers*

Circular Wait

- Impose a total ordering of all resource types
- *Each process requests resources in an increasing* order of enumeration

Possible side effects of preventing deadlocks by the method?

Deadlock Avoidance

- Requires a *priori information* - maximum requirements of each process
- Do not start a process if its maximum requirement can lead to a deadlock
- Two algorithms
  - Only one instance of each resource type – Resource Allocation Graph Algorithm
  - If multiple instances of each resource type – Bankers Algorithm

Safe State

- *State is safe* if a system can allocate resources to each process (up to Max) *in some order* and still avoid deadlock

- System is in safe state if there exists a **safe sequence**
- $<P_1, P_2, \ldots, P_n> \rightarrow$ The resources that P_i can request be satisfied by *currently available resources* + *resources held* by all the P_j $(j<I)$
  - 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

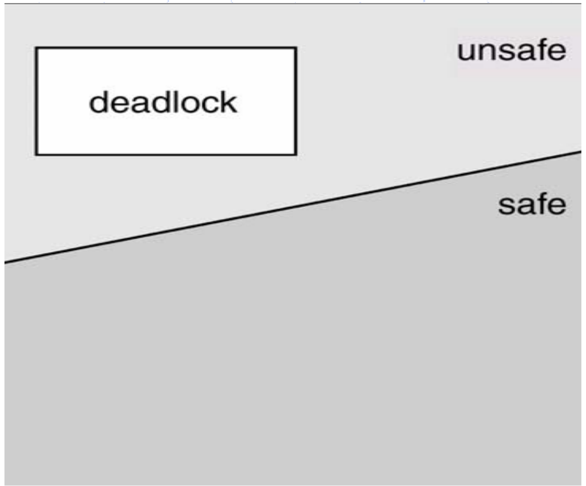
Example: 12 tape drives

	Maximum Needs	**Current Needs**
P_0	10	5
P_1	5	2
P_2	9	2

Safe? Sequence?

Basic Facts

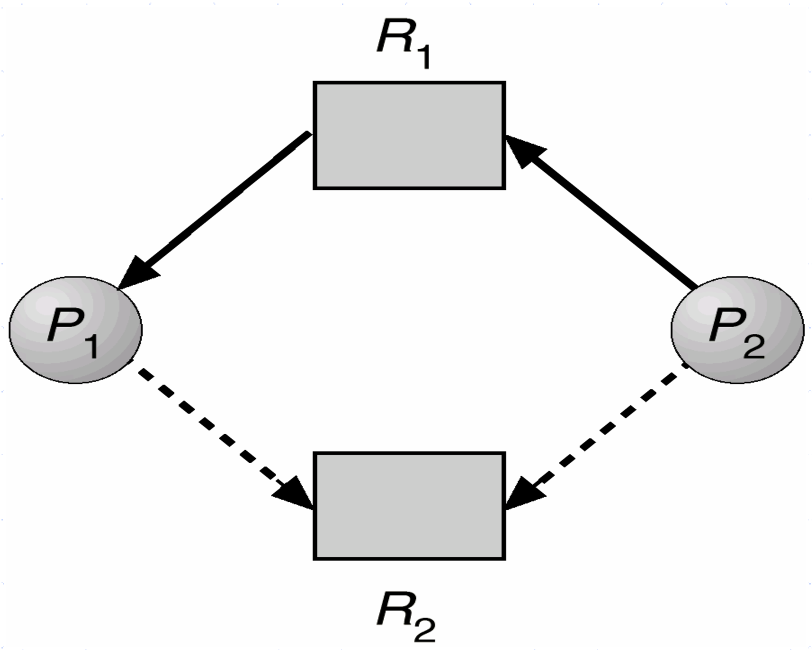
- Safe state ⇒ no deadlocks
- Unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state



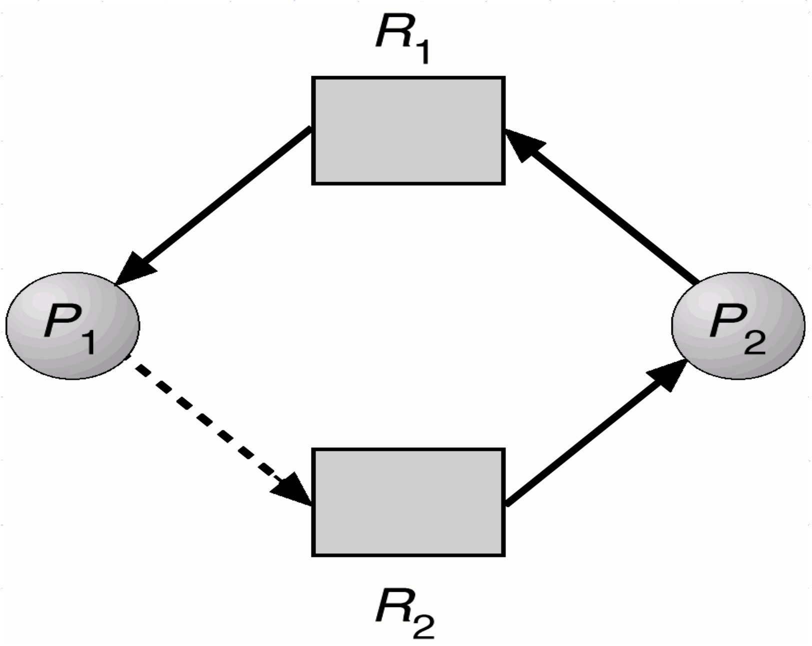
Resource-Allocation Graph Algorithm

- RAS with only one instance of each resource type
- *Claim edge* $P_i \rightarrow R_j$ indicates that process P_j may request resource R_j in future
  - Representation → dashed line
- Claim edge converts to request edge when a process requests a resource
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system
  - Request to assignment edge → No cycle in RAG, Safe state
  - Cycle detection → Unsafe state, P_i waits for its request

Resource-Allocation Graph Algorithm



Safe?



P_2 Must Wait! A cycle found.

Complexity – Finding a cycle in the graph per resource request

Banker’s Algorithm

- Multiple instances, Less efficient, Banking system
- Each process *must* declare priori maximum number of instances per resource type it may need
- 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

Banker's Algorithm – Data Structures

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*]

Simulate evolution of system over time under the assumptions of worst case resource demands

Banker’s Algorithm – Safety Procedure

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

   Work = *Available*

   Finish [i] = *false* for *i* - 1,3, …, *n.*

2. Find process *i* such that both:

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

   (b) $\mathit{Need}_i \leq \mathit{Work}$

   If no such *i* exists, go to step 4.

3. $\mathit{Work} = \mathit{Work} + \mathit{Allocation}_i$
   Finish [i] = *true*
   go to step 2.

4. If *Finish* [i] == true for all *i*, then the system is in a safe state; otherwise process whose index is false *may potentially be in deadlock* in future

Banker's Algorithm – Resource Request

$Request_i \rightarrow$ request vector (P_i); *e.g.* $Request_i[j] = k$

1. If $Request_i \leq Need_i$ go to step 2; Else *raise error condition* → process exceeds its maximum claim

2. If $Request_i \leq Available$, go to step 3; Else P_i *must wait*, since resources are not available

3. Tentatively allocate requested resources to P_i by modifying the state as follows:

   $Available = Available - Request_i;$

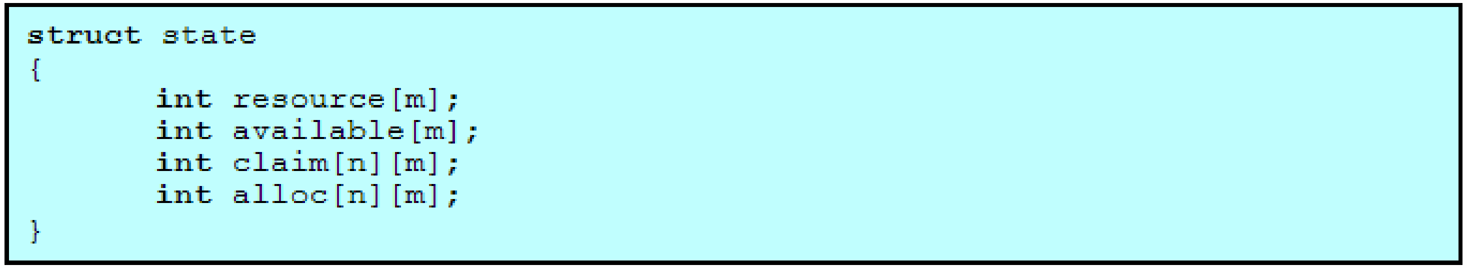
   $Allocation_i = Allocation_i + Request_i;$

   $Need_i = Need_i - Request_i;$

   Check the safety of state -

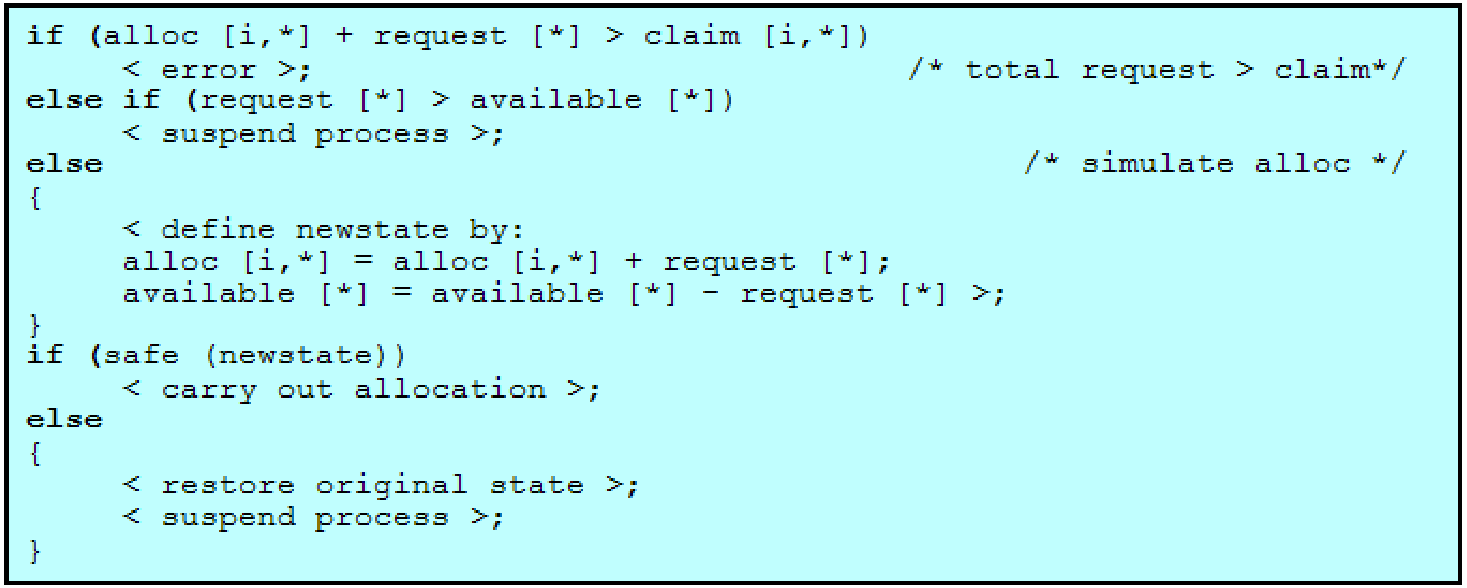
   - *If safe* ⇒ *the resources are allocated to* P_i
   - *If unsafe* ⇒ P_i *must wait, and the tentative resource allocation is cancelled*

Banker’s Algorithm



```
struct state
{
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}
```

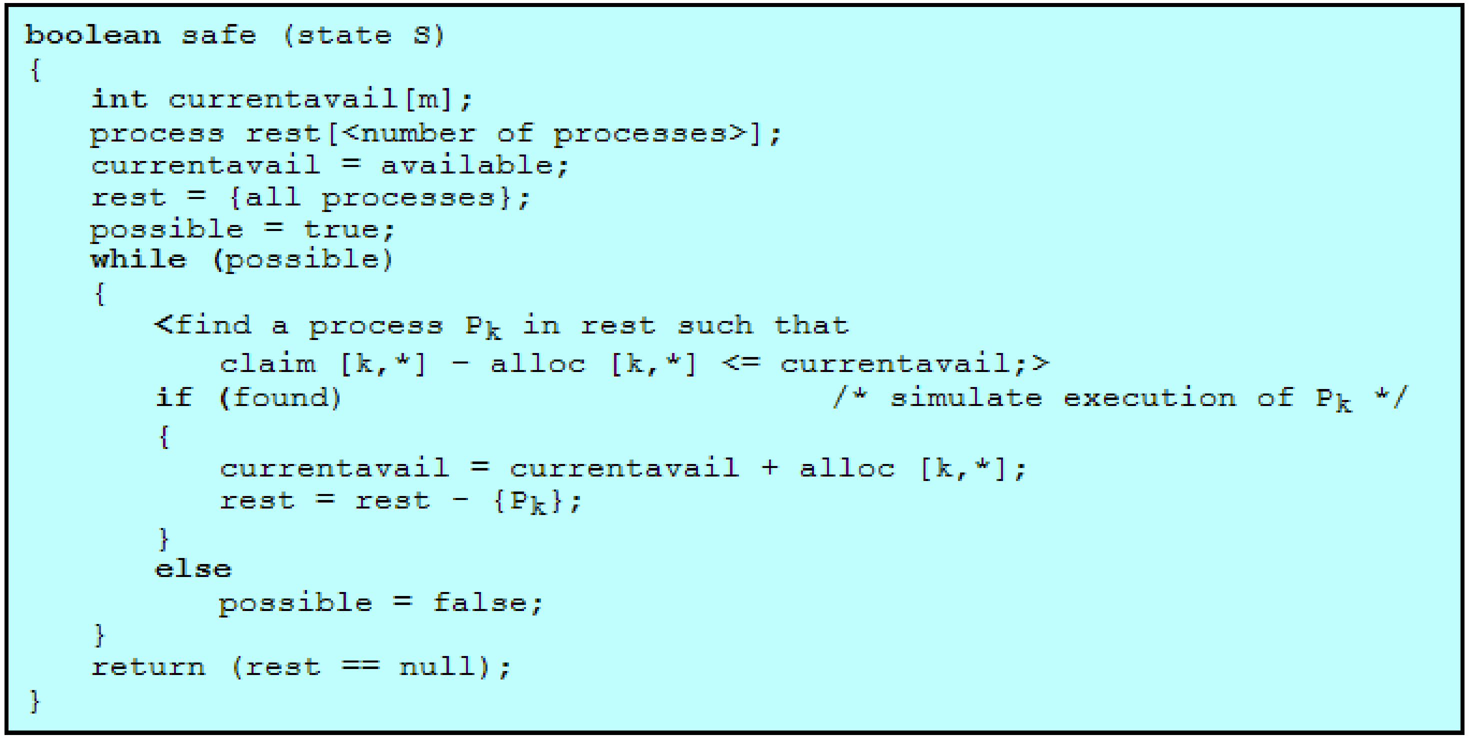
(a) global data structures



```
if (alloc [i,*] + request [*] > claim [i,*])
      < error >;                                  /* total request > claim*/
else if (request [*] > available [*])
      < suspend process >;
else                                                    /* simulate alloc */
{
      < define newstate by:
      alloc [i,*] = alloc [i,*] + request [*];
      available [*] = available [*] - request [*] >;
}
if (safe (newstate))
      < carry out allocation >;
else
{
      < restore original state >;
      < suspend process >;
}
```

(b) resource alloc algorithm

Banker’s Algorithm



```
boolean safe (state S)
{
   int currentavail[m];
   process rest[<number of processes>];
   currentavail = available;
   rest = {all processes};
   possible = true;
   while (possible)
   {
      <find a process Pk in rest such that
         claim [k,*] – alloc [k,*] <= currentavail;>
      if (found)                            /* simulate execution of Pk */
      {
         currentavail = currentavail + alloc [k,*];
         rest = rest - {Pk};
      }
      else
         possible = false;
   }
   return (rest == null);
}
```

test for safety

Deadlock Avoidance

- Maximum resource requirement must be stated in advance
- Processes under consideration must be independent; no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

Example - 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	

Is the system in safe state?

Example - 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			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

Safe sequence → $<P_1, P_3, P_4, P_2, P_0>$

Example - Banker's Algorithm

- Check that Request ≤ Available (that is, $(1,0,2) \le (3,3,2) \Rightarrow$ *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 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ is also a safe sequence
- Further, can request for (3,3,0) by P_4 be granted?
- What if P_0 requests (0,2,0)?

Deadlock Detection

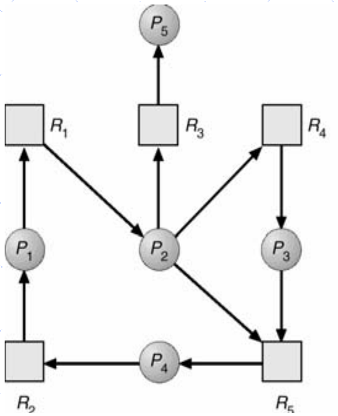
- Third Option → Allow system to enter deadlock state

- Then system must provide
  - An algorithm to *periodically determine whether deadlock* has occurred in the system
  - An algorithm to *recover* from the deadlock

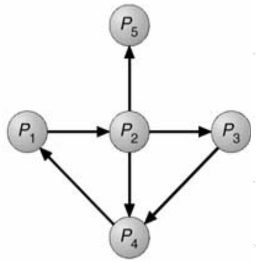
- Two algorithms
  - Single instance of each resource type
  - Multiple instances of resource type

Single Instance per Resource Type

- Maintain a *wait-for* graph → Variant of RAG
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Same as RAG but optimizes it for the search by collapsing edges



Resource allocation graph



Corresponding *wait-for* graph

- Periodically invoke an algorithm that searches for a cycle in the graph

Several Instances per Resource Type

- Similar to the *Banker's algorithm* *safety test* with the following difference in semantics;
  - Replacing $Need_i \rightarrow Request_i$; where $Request_i$ is the actual vector of resources, process *i* is currently waiting to acquire
  - May be slightly optimized by initializing *Finish* [i] to *true* for every process *i* where $Allocation_i$ is zero
  - Optimistic and *only care if deadlock now;*
    In future → deadlock, discovered in future
  - Processes *in the end* remaining *with false entry* are the ones *involved in deadlock* at this time
- *Complexity* → $m \times n^2$ operations

Detection Algorithm

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

   $\mathit{Work} = \mathit{Available}$

   If $\mathit{Allocation}_i \neq 0$ for $i = 1, 2, \ldots, n$ then

   $\mathit{Finish}[i] =$ **false**, else $\mathit{Finish}[i] =$ **true**

2. Find process i such that both:

   (a) $\mathit{Finish}[i] =$ **false**

   (b) $\mathit{Request}_i \leq \mathit{Work}$

   If no such i exists, go to step 4.

3. $\mathit{Work} = \mathit{Work} + \mathit{Allocation}_i$
   $\mathit{Finish}[i] = \mathit{true}$
   go to step 2

4. If $\mathit{Finish}[i] ==$ **false**, for some $1 \leq i \leq n$, → deadlocked;

   If $\mathit{Finish}[i] ==$ **false** then process P_i is deadlocked

Example – Detection Algorithm

- 5 Processes P_0 through P_4; 3 resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time T_0:

	Allocation A B C	Request A B C	Available 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	

Is the system in deadlock state?

Example – Detection Algorithm

- 5 Processes P_0 through P_4; 3 resource types
  A (7 instances), *B* (2 instances), and *C* (6 instances)
- *Suppose P_2 requests an additional instance of type C*

	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 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Is the system in deadlock state?

Detection Algorithm Usage

- When, and how often, to invoke?
  - In the extreme – every time a request for resource allocation cannot be granted
  - Every resource request → invoke deadlock detection
    - Considerable overhead in computation time, cost/complexity
  - Reasonable alternative is to invoke the algorithm periodically
    - What period? How much can you wait once deadlock is detected? → *e.g.* once per hour or CPU utilization < 40%
    - How many resources we can commit for the detection?

Deadlock Recovery: Process Termination

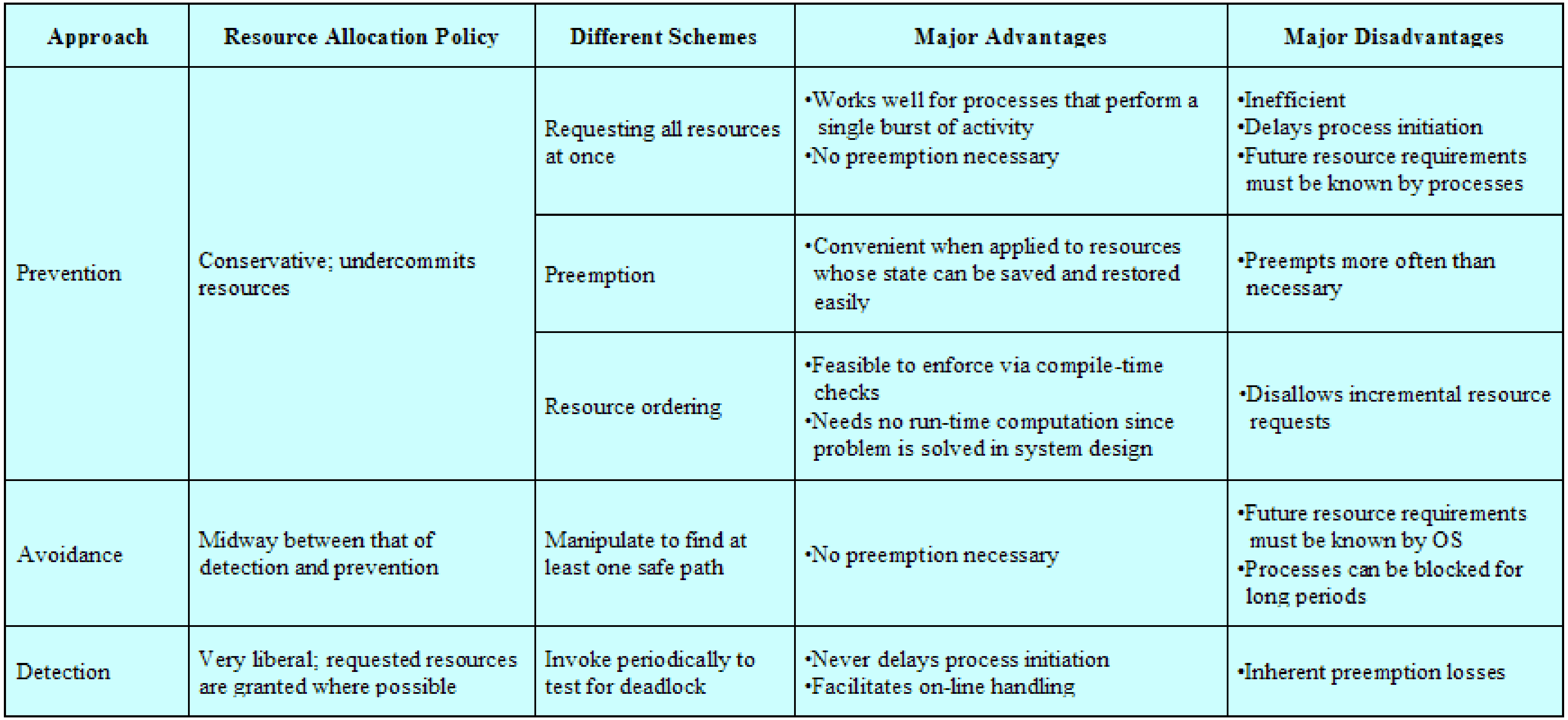
- Abort all deadlocked processes → Fast but expensive
- Abort one process at a time until the deadlock cycle is eliminated
  - Considerable overhead
  - If in the midst of job, *e.g.* file updating or printing
- How to select the order of process to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Deadlock Recovery: Resource Preemption

- *Selecting a victim* – minimize cost
- If we preempt resources, *what to do with process*? **Rollback** → return to some safe state, restart process for that state
- *Starvation* → Same process may always be picked as victim, include # of rollbacks in cost factor

Strengths and Weaknesses of the Strategies

Summary of Detection, Prevention and Avoidance approaches



Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates on-line handling	•Inherent preemption losses

Combined Approach to Deadlock Handling

- Combine the three basic approaches
  - Prevention
  - Avoidance
  - Detection

  allowing the use of the optimal approach for each of resources in the system

- Partition resources into hierarchically ordered classes

- Use most appropriate technique for handling deadlocks within each class