Operating Systems: Resource Management

Deadlocks

- Law passed by Kansas Legislature in early 20th century:

  "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start upon again until the other has gone."

- Neil Groundwater has the following to say about working with Unix at Bell Labs in 1972:

  " ... the terminals on the development machine were in a common room ... when one wanted to use the line printer. There was no spooling or lockout. `pr myfile > /dev/p` was how you sent your listing to the printer. If two users sent output to the printer at the same time, their outputs were interspersed. Whoever shouted, "line printer!" first owned the queue."

Deadlocks

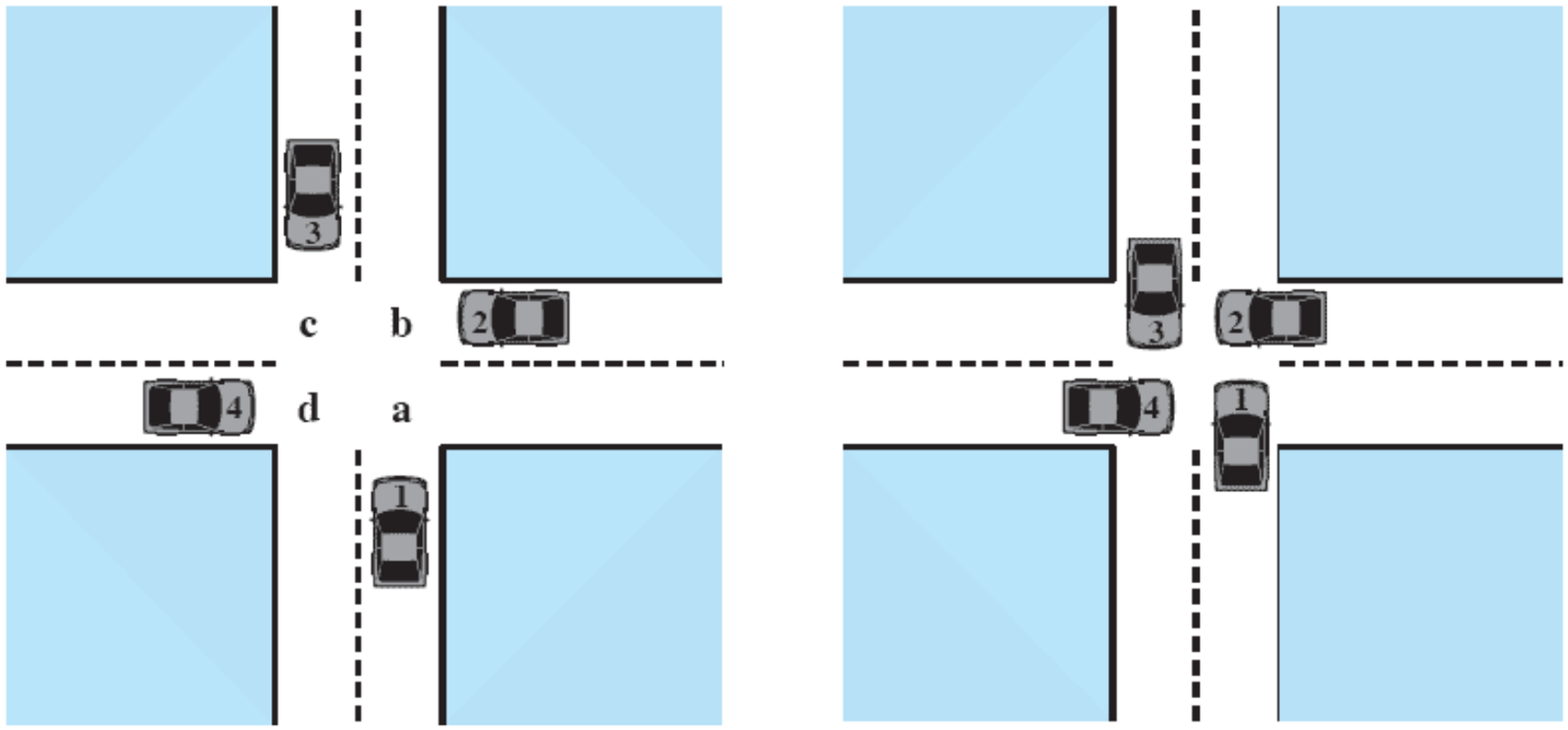


Figure 6.1 Illustration of Deadlock

Deadlocks

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
  - Several processes may compete for a finite set of resources
  - Processes request resources and if a resource is not available, enter a wait state
  - Requested resources may be held by other waiting processes
  - Require divine intervention to get out of this problem

Deadlocks

- A significant problem in real systems, because there is no efficient solution in the general case
- Deadlock problem is more important because of increasing use of multiprocessor systems
    - real-time systems, life support, vehicle monitoring, multicore utilization, grid processing
- Important in answering the question about whether or when a process will complete

Deadlocks

- Deadlocks can occur with
  - Serially reusable (SR) resources - printer, tape drive, memory
    - A finite set of identical units, with the number of units constant
    - Can be used safely by only one process at a time and are not depleted by that use
    - Units acquired by processes, used, and released later for use by other processes
    - A process may release a unit only if it has previously acquired it
    - Examples include processors, memory, devices, files, databases, and semaphores

Deadlocks

- Deadlocks can also occur with:
  - Consumable resources like messages
  - Resource gets created dynamically and may be destroyed after use
  - Typically no limit on the number of consumable resources of a specific type
  - Examples are messages, signals, interrupts, and information in I/O buffers

Examples of Deadlocks

- Reusable resources
  - File sharing
    - Consider two processes p_1 and p_2
    - They update a file *F* and require a scratch tape during the updating
    - Only one tape drive *T* available
    - *T* and *F* are serially reusable resources, and can be used only by *exclusive access*

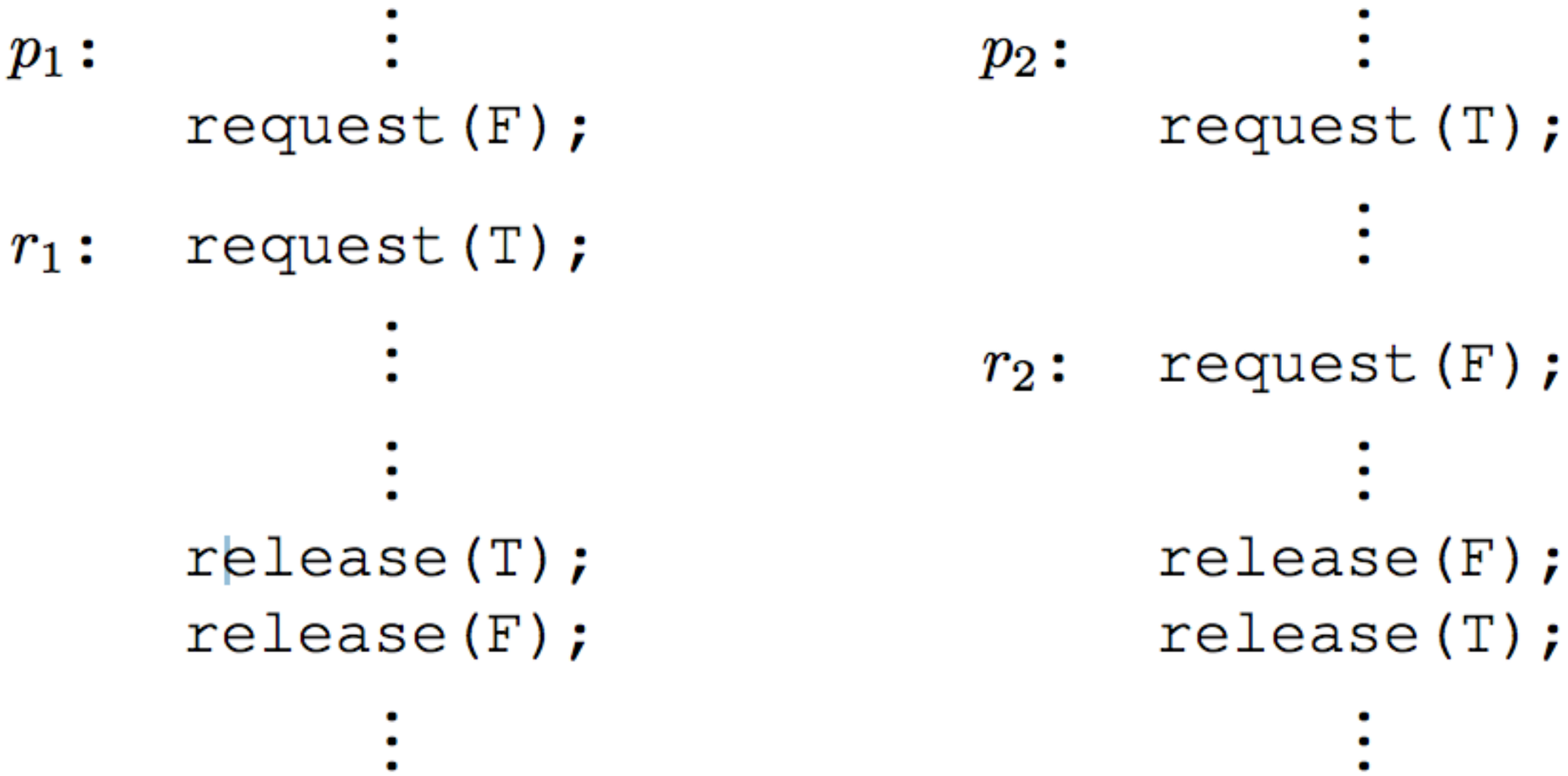
Examples of Deadlocks

- Reusable resources
  - p_2 needs *T* immediately prior to updating
  - *request* operation
    - Blocks the process requesting the resource
    - Puts the process on the wait queue
    - The process remains blocked until requested resource is available
    - If the resource is available, the process is granted exclusive access to it

Examples of Deadlocks

- Reusable resource
  - release operation
    - Returns the resource being released to the system
    - Wakes up the process waiting for the resource, if any
- p_1 can block on *T* holding *F* while p_2 can block on *F* holding *T*

Examples of Deadlocks



Examples of Deadlocks

- Single resource sharing
  - A single SR resource, such as memory *M*, with *m* allocation units shared by *n* processes p_1, p_2, ..., p_n, 2 <= *m* <= *n*
  - Suppose sequence of operations is as follows:

```
m1 = malloc ( 1024 );
m2 = malloc ( 1024 );
...
free ( m1 );
free ( m2 );
```

  - Deadlock due to no memory being available and existing processes request more memory
  - Fairly common cause of deadlock

Examples of Deadlocks

- Consumable resources
  - Deadlock with messages
    - Have a pair of processes p1 and p2
    - Each process receives a message from other process and then sends a message to the other process

$p_1()$	$p_2()$
⋮	⋮
receive (p_2)	receive (p_1)
⋮	⋮
send (p_2, m_1)	send (p_1, m_1)

- Deadlock with blocking receive

Examples of Deadlocks

- Locking in Database Systems
  - Locking required to preserve integrity and consistency of databases with random request patterns
  - Problem when two records to be updated by two different processes are locked
- Deadlocking by nefarious users

```
void deadlock( task ) {
   wait ( event );
}  /* deadlock */
```

Examples of Deadlocks

- Effective Deadlocks
  - Milder form of indefinite postponement of processes competing for a resource
  - Exemplified by Shortest Job Next Scheduling
    - If many small processes come in, large job never gets any time

Examples of Deadlocks

- Deadlocks in Unix
  - Possible deadlock condition that cannot be detected
  - Number of processes limited by the number of available entries in process table
    - If process table is full, `fork` system call fails
    - Process can try and wait random time before trying `fork` again

```
Example:
   10 processes create 12 children each
   100 entries in process table
   Each process has already created 9 children
   No more space in process table, so deadlock
```

Examples of Deadlocks

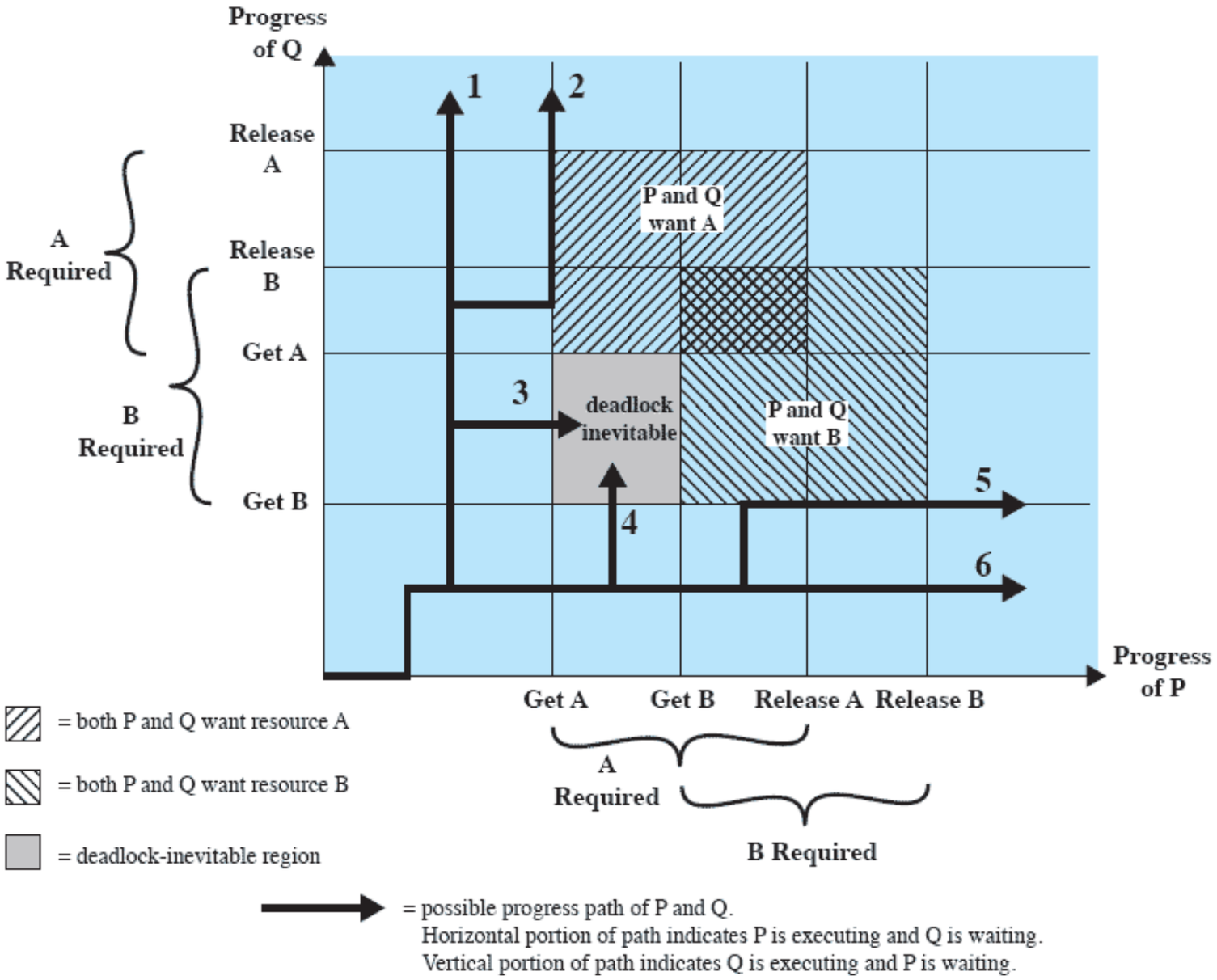


Figure 6.2 Example of Deadlock

Examples of Deadlocks

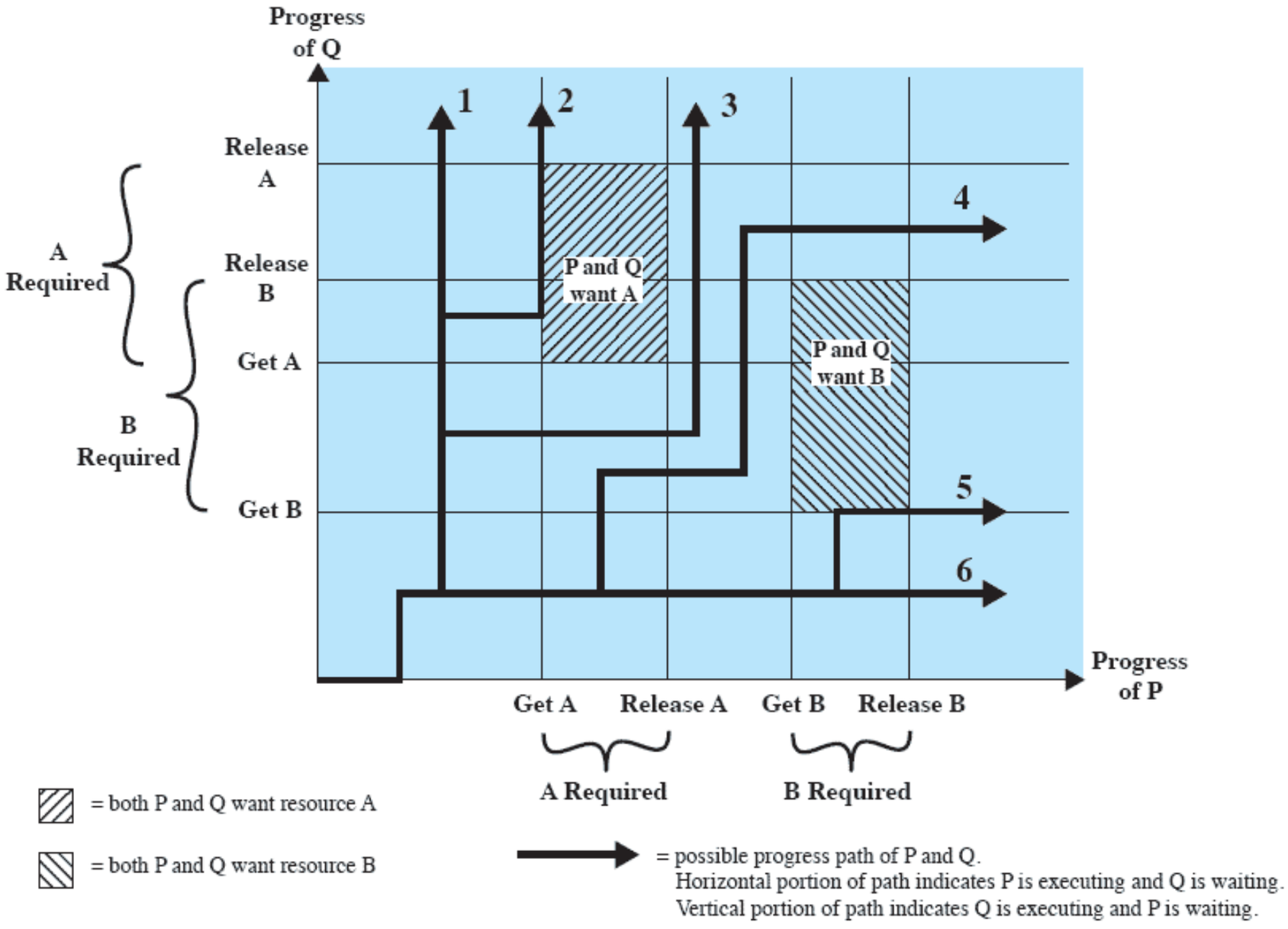


Figure 6.3 Example of No Deadlock [BACO03]

Deadlocks Problem Characterization

- Deadlock Detection
  - Process resource graphs
- Deadlock Recovery
  - What is “best” way of recovering from a deadlock
- Deadlock Prevention
  - Not allowing a deadlock to happen

Resource Allocation Graph

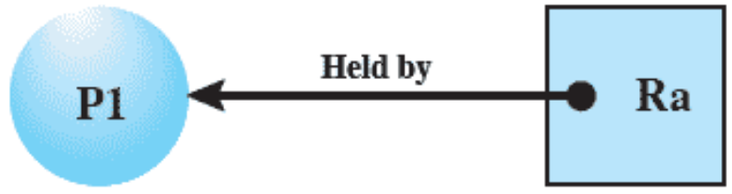
- Directed graph to describe state of the system of resources and processes
- Set of vertices V consisting of
  - Set of processes P = P1, P2, ...
    - Process nodes are represented as circles
  - Set of resource types R = R1, R2, ...
    - Resource nodes represented as squares with a dot representing each instance of the resource
    - Resource types with multiple instances could be I/O devices allocated by a resource management module

Resource Allocation Graph

- Set of edges E
  - Request edge
    - Directed edge from P_i to R_j
      - Denoted by $P_i \rightarrow R_j$
      - Pi has requested an instance of Rj and is currently waiting for that resource
  - Assignment edge
    - Directed edge from R_j to P_i
      - Denoted by $R_j \rightarrow P_i$
      - An instance of R_j has been allocated to P_i

Resource Allocation Graph

(a) Resouce is requested

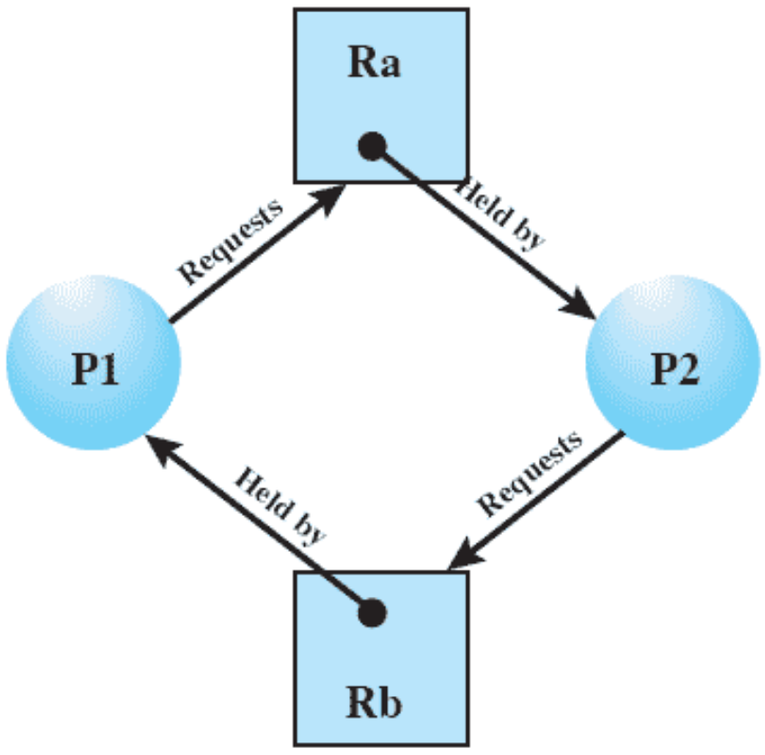


(b) Resource is held

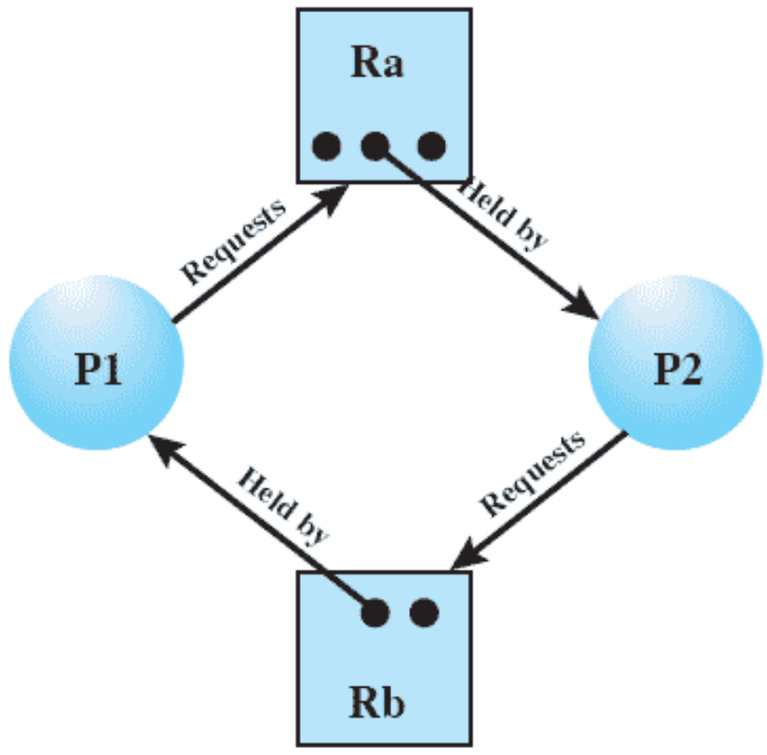
Resource Allocation Graph

- No cycles in the graph means no deadlock
- Cycle in the graph means could be a deadlock
  - If not multiple instances
- Each process involved in a cycle is deadlocked
- Cycle in the resource graph is necessary and sufficient condition for existence of a deadlock
- If a graph contains several instances of a resource type, a cycle is not a sufficient condition for a deadlock but it is a necessary condition

Resource Allocation Graph



(c) Circular wait



(d) No deadlock

Resource Allocation Graph

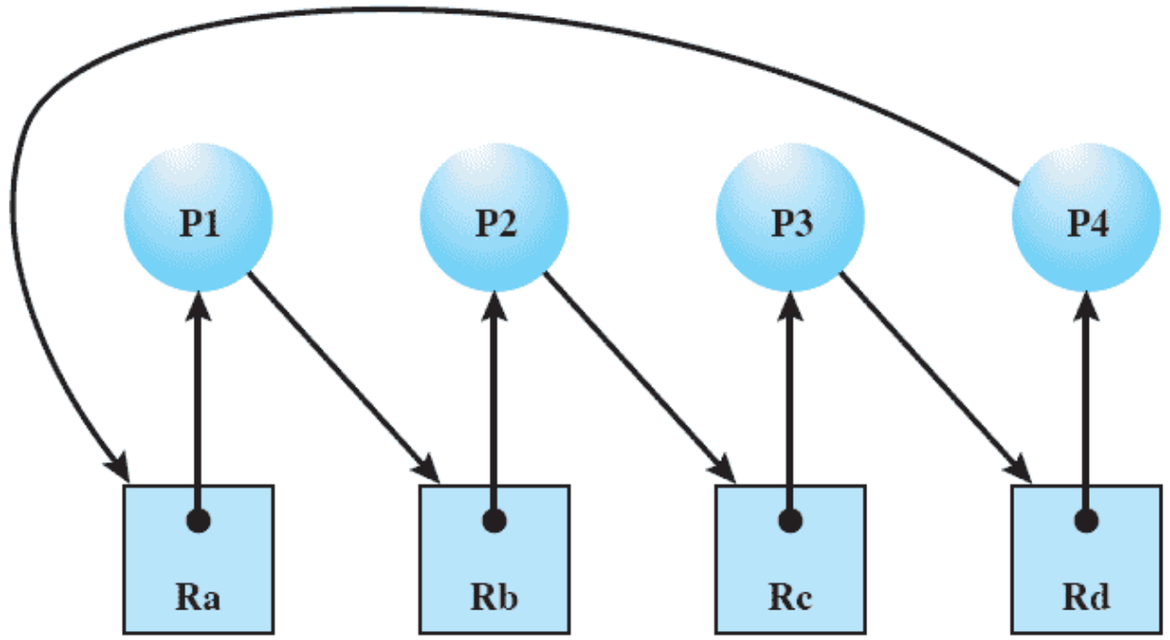


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Deadlock Characterization

- Four necessary and sufficient conditions for deadlock
  - Mutual Exclusion
    - Only one process may use a resource at a time
    - At least one resource must be held in a non-sharable mode
  - Hold and wait
    - Existence of a process holding at least one resource and waiting to acquire additional resources currently held by other processes
  - No preemption
    - Resources cannot be preempted by system
  - Circular wait
    - Processes waiting for resources held by other waiting processes

Deadlock Detection

- Do not restrict process actions or limit resource access
  - If resources are available, satisfy requests
- Periodically detect circular wait condition
  - Using a deadlock detection algorithm

Deadlock Detection

- Simulate the *most favored execution* of each unblocked process
  - An unblocked process may acquire all the needed resources
  - Run and then release all the acquired resources
  - Remain dormant thereafter
  - Released resources may wake up some previously blocked process
  - Continue the above steps as long as possible
  - If any blocked processes remain, they are deadlocked

Deadlock Detection

- Reduction of resource graphs
- Process blocked if it cannot progress by either of the following operations
  - Request
  - Acquisition
  - Release

Deadlock Detection

- Reduction of resource graph
  - Reduced by a process p_i
    - by removing all edges to and from p_i
      - p_i is neither blocked nor isolated node
    - After reduction, p_i becomes an isolated node
    - Graph is irreducible if it cannot be reduced by any process
    - Completely reducible if a sequence of reductions deletes *all* the edges in the graph
- Lemma 1. All reduction sequences of a given resource graph lead to the same irreducible graph

Deadlock Detection Algorithm

- Algorithm will attempt to reduce our resource graph
- Four lists:
  - Resources requested by each process (Request matrix Q)
  - Resources allocated to each process (Allocation matrix A)
  - How many total resources (of each) we have (Resource vector)
  - How many we have left to allocate (Allocation vector)

Deadlock Detection Algorithm

- Algorithm will attempt to reduce our resource graph
- Works by looping through process request matrix Q
    - Attempting to satisfy a request
        - If able to satisfy, can then release all resources
- If unable to clear out request matrix, then we have detected a deadlock
    - Most favorable scenario did not result in system being able to finish
    - Only processes left over are guaranteed to result in deadlock
- For n processes and m resources, worst case execution time of mn^2

Deadlock Detection Algorithm

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector

Figure 6.10 Example for Deadlock Detection

Deadlock Detection Algorithm

```
bool deadlock( const int *available, const int m, const int n, const int *request, const int *allocated) {
   int work[m];   // m resources
   bool finish[n];   // n processes

   for (int i (0); i < m; work[i] = available[i++]);
   for (int i (0); i < n; finish[i++]; = false;

   int p (0);
   for (; p < n; p++) {
      if ( finish[p] ) continue;
      if (req_lt_avail (request, work, p, m)) {
         finish[p] = true;
         for (int i (0); i < m; i++)
            work[i] += allocated[p*m+i];
         p = -1;
      }
   }

   for (p = 0; p < n; p++)
      if ( !finish[p] )
         break;

   return (p != n);
}

bool req_lt_avail(const int *req, const int *avail, const int pnum, const int num_res) {
   int i (0);
   for (; i < num_res; i++)
      if (req[pnum*num_res+i] > avail[i] )
         break;
   return ( i == num_res );
}
```

Deadlock Recovery

- So we discover system is in a deadlock, how do we recover?
- Two ways:
  - Recovery by process termination
  - Recovery by resource pre-emption

Process Termination

- Abort all deadlocked processes
  - Most commonly adopted solution
    - Do we need to abort them all though?
- Back up system to a checkpoint
  - Requires restarting at that point
  - Requires a system have rollback and restart mechanisms
  - Will original deadlock reoccur?

Process Termination

- Terminate deadlocked processes in systematic way
  - Keep terminating until no longer in a deadlock
    - Do deadlock detection at each process termination
  - Should terminate based on some policy
    - Process priority, cpu time used and expected usage, number resources taken up, is it interactive or batch?
  - A few problems:
    - If a process has open files, termination could lead to invalid files
    - Same issues with other i/o: Terminated print job, etc

Resource Preemption

- Preempt resources from some processes
    - Allows satisfaction of requests from other processes
        - Could resolve the deadlock
- How do we select a victim?
- Need to rollback the state
- Starvation could occur
    - Keep preempting resources from one job and it will never get to finish

Deadlock Prevention

- Let us make sure we can never get into a deadlock.
- How?
  - Need to set up our system so we prevent one of the four conditions necessary for deadlock to occur

Deadlock Conditions

- Four necessary and sufficient conditions for deadlock
  - Mutual Exclusion
    - Only one process may use a resource at a time
    - At least one resource must be held in a non-sharable mode
  - Hold and wait
    - Existence of a process holding at least one resource and waiting to acquire additional resources currently held by other processes
  - No preemption
    - Resources cannot be preempted by system
  - Circular wait
    - Processes waiting for resources held by other waiting processes

Deadlock Prevention

- Require each process to request and acquire all needed resources at once
  - Eliminates the *hold and wait* necessary condition of deadlock
  - Works decently for processes that perform a single burst of activity
  - Doesn't require preemption
  - Serious downsides:
    - Can delay process initialization
    - Have to know at the start what resources it will need
      - Not always possible
    - Can end up holding on to resources for much longer than it needs to

Deadlock Prevention

- Impose a total ordering on all resource types
  - Intention is to eliminate possibility of *circular wait* condition
  - Processes must request resources in order
  - If several instances are needed, must request all instances at once
  - Example: Resources R_1, R_2, ..., R_5
    - Process could request R_2 and then R_4
    - Could not request R_3 and then R_2
  - Disallows incremental resource requests

Deadlock Prevention

- Allow preemption of resources
  - Eliminates *no preemption* condition
  - Different ways to accomplish this:
    - If process gets denied a resource request, release all resources and must then request them all
    - If a process requests a resource that is currently held by another, require the other process to release its resources
- Only useful if state can be saved and restored easily
- Does preemptions that aren't strictly necessary

Deadlock Avoidance

- Deadlock prevention constrains resource requests to prevent deadlock
  - Prevent at least one of four necessary conditions of deadlock
  - Leads to inefficient use of resources and inefficient process execution
- Deadlock avoidance allows three of the four conditions
  - Then makes choices to ensure that deadlock point is never reached
  - Allows more concurrency than prevention does

Deadlock Avoidance

- We will discuss two approaches to deadlock avoidance:
  - Do not start a process if its demands might lead to a deadlock
  - Do not grant a request if the allocation could lead to deadlock
- Dynamically exist the resource allocation status to ensure that no circular wait condition can exist
- Requires knowledge of maximum possible claims for each process

Deadlock Avoidance

- Called Bankers Algorithm
    - Banking system that never allocates its available cash such that it can no longer satisfy the needs of all its customers
- Examine current state of the system, determine if it is *safe*
- *Safe state* is where at least one sequence does not result in a deadlock
    - A *safe sequence* exists : If execution follows this path, no deadlock
- *Unsafe state* may lead to a deadlock
    - *Unsafe state* will not always result in a deadlock

Deadlock Avoidance

- Note that it is possible to from a *safe state* to an *unsafe state*
- Our task:
    - Detect the possibility of an *unsafe state* and deny requests even if resources are available

Deadlock Avoidance

System with 12 magnetic tape drives

Process	Max needs	Allocation	Current needs
p_0	10	5	5
p_1	4	2	2
p_2	9	2	7

Current availability : 3

Safe sequence: $\langle p_1, p_0, p_2 \rangle$

p_1 acquires 2 more tapes...

System with 12 magnetic tape drives

Process	Max needs	Allocation	Current needs
p_0	10	5	5
p_1	4	4	0
p_2	9	2	7

Current availability : 1

Fine, but what if p_2 is allowed to request and acquire one more?

Deadlock Avoidance

- We need a system state, so as before, *m* resources and *n* processes
  - Resource vector resource[m]
    - Total resources available in system
  - Available vector available[m]
    - Current resources available to be allocated
  - Maximum claims vector claim[n][m]
    - Process p_i may request at most *k* instances of resource R_j
  - Allocation vector alloc[n][m]
    - Process pi is currently allocated k instances of resource R_j

Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) Initial state

Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

Determination of a Safe State

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

Determining an Unsafe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(b) P1 requests one unit each of R1 and R3

Deadlock Avoidance Logic

```
// global data structures
struct state {
   int resource[m];
   int available[m];
   int claim[n][m];
   int alloc[n][m];
}


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

Deadlock Avoidance Logic

```
boolean safe (state S) {
   int currentavail[m];
   process rest[<number of processes>];  // list of processes running
   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};    // Remove Pk from list of running processes
      }
      else
         possible = false;
   }
   return (rest == null);
}
```

Deadlock Avoidance

- Pros:
    - No preemption necessary
- Cons:
    - Maximum resource requirement must be stated in advance
    - Processes under consideration must be independent
        - No synchronization requirements
    - Fixed number of resources

Integrated Strategies

- All strategies have their strengths and weaknesses
    - In practice must design OS's that employ different strategies in different situations
- One integrated strategy:
    - Group resources into different resource classes
    - Use linear ordering between resource classes
        - Prevents circular wait
    - Within a specific resource class, use the appropriate algorithm

Integrated Strategies

- Possible division of resources into classes:
  - Swappable space
    - Blocks of memory on secondary storage for swapping processes
  - Process resources
    - Assignable devices like tape drives and files
  - Main memory
    - Pages or segments of memory assigned to processes
  - Internal resources
    - I/O channels and the like

Integrated Strategies

- Strategies for each resource class:
  - Swappable space
    - Require all resources be grabbed at once, or deadlock avoidance
  - Process resources
    - Avoidance or resource ordering
  - Main memory
    - Preemption, when a process is preempted it is just swapped to secondary memory
  - Internal resources
    - Resource ordering

End ofResource Management

- Any questions?