Bilkent University
Department of Computer Engineering
CS342 Operating Systems

Deadlocks

Last Update: April 16, 2023

Objectives and Outline

Objectives

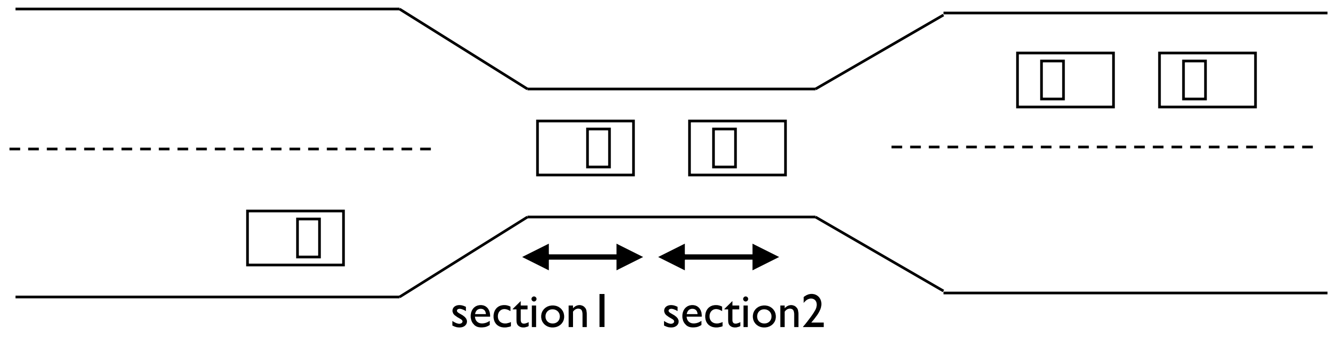
- To develop a **description of deadlocks**, which prevent sets of concurrent processes from completing their tasks
- To present a number of **different methods** for preventing or avoiding deadlocks in a computer system

Outline

The Deadlock Problem

- Deadlocks: *A set of blocked threads (or processes) each holding a resource(s) and waiting to acquire a resource(s) held by another thread in the set*
  - Resources can be hardware or software resources,
    - files, locks, or semaphores.
    - I/O devices
    - Printers
    - Memory
    - ...
- A thread or process may need to use several resources.

A deadlock from real life: Bridge crossing example



- Traffic only in one direction.
- Each section 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).

System Model

- Resources are accessed with the following protocol.
- A process (thread) that needs a resource first makes a request for it.
- If the requested resource is available and is granted, the process can use it.
- After using the resource, the process will release the resource.

System Model

- Request/release may be
  - open()/close() calls for file I/O
  - open()/close() calls for device I/O
  - lock(), unlock() for mutex locks
  - wait(), signal() for semaphores.
  - malloc(), free() for memory allocation
- Resources are usually protected by locks or semaphores.
  - Locks and semaphores can be sources of deadlocks.

Deadlocks in multithreaded applications

Example: deadlock

```
pthread_mutex_lock  L1;  // lock 1 – to protect resource 1
pthread_mutex_lock  L2;  // lock 2 – to protect resource 2

pthread_mutex_init (&L1);
pthread_mutex_init (&L2);
```

thread1

```
void * dowork1 (void * p) {
      pthread_mutex_lock (&L1);
      pthread_mutex_lock (&L2);
      // do some work – use resources
      pthread_mutex_unlock (&L2);
      pthread_mutex_unlock (&L1);
}
```

thread2

```
void * dowork2 (void * p) {
      pthread_mutex_lock (&L2);
      pthread_mutex_lock (&L1);
      // do some work – use resources
      pthread_mutex_unlock (&L1);
      pthread_mutex_unlock (&L2);
}
```

We have deadlock, if thread 1 has L1 and thread 2 has L2.

(order of acquiring locks are not the same)

Deadlocks in multithreaded applications
Livelock

- Livelock is another form of liveness failure. Similar do deadlocks.
  - Processes run (not blocked) but do not make progress. They *repeatedly do the same action* and no progress.
  - In deadlock, processes do not run (blocked).
- For example: *each thread needing two locks may acquire the first and try to get the second lock. If the thread can not get the second lock, it can release the first lock, and repeat the whole action again.*
  - We have non-blocking pthread_mutex_trylock() function.
    - If it can get the lock, returns true (`1`); otherwise returns false (`0`).
    - Does not block in any case.

Deadlocks in multithreaded applications Example: livelock

Thread 1

```
void * dowork1 (void * p) {
    int done = 0;
    while (!done) {
        pthread_mutex_lock (&L1);
        if (pthread_mutex_trylock (&L2)) {
            // do some work
            // release the locks
            pthread_mutex_unlock (&L2);
            pthread_mutex_unlock (&L1);
            done = 1;
        }
        pthread_mutex_unlock (&L1);
   } // while
   pthread_exit(0);
} // end dowork1
```

Thread 2

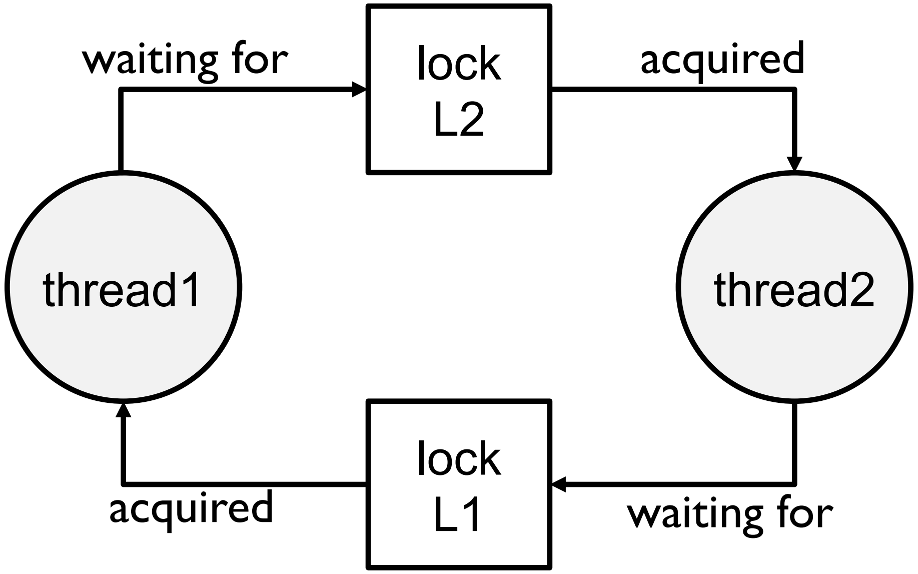
```
void * dowork2 (void * p) {
    int done = 0;
    while (!done) {
        pthread_mutex_lock (&L2);
        if (pthread_mutex_trylock (&L1)) {
            // do some work
            // release the locks
            pthread_mutex_unlock (&L1);
            pthread_mutex_unlock (&L2);
            done = 1;
        }
        pthread_mutex_unlock (&L2);
   } // while
   pthread_exit(0);
} // end dowork1
```

Thread 1 gets L1, and tries to get L2. In a loop.
Thread 2 gets L2, and tries to get L1. In a loop.

Deadlock Characterization

- Deadlocks 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 is waiting to acquire additional resource(s) held by other processes.
  - No preemption: a resource can be released only voluntarily by the process holding it, after the process completed its task.
  - Circular wait: there exists a set $\{P_0, P_1, \ldots, P_n, P_0\}$ 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, …, P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

Circular wait



A deadlock situation between two threads.
A circular wait is happening due to locks.
Locks are like resources here. Threads can have a lock one thread at a time.

Resource-allocation graph

- In the system/applications, we may have resources of different types (classified into different classes).
- Resource types $R_1, R_2, . . ., R_m$

  CPU cycles, memory space, I/O devices, files, data structures, locks, semaphores,
- Each resource type R_i may have multiple *identical* W_i instances.
- We may have one or more processes (threads) interested in using one or more resources.
- Each process, from time to time, may request resources (may be waited for them), use them, and release them.

Resource-allocation graph

- The system state can be represented with a resource allocation graph.
- In the graph we have a set of vertices V and a set of edges E.

Vertices (V): V is partitioned into two types:

$P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes (or threads) in the system.

$R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

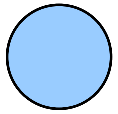
Edges (V):

request edge: directed edge $P_i \rightarrow R_j$

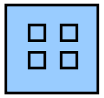
assignment edge: directed edge $R_j \rightarrow P_i$

Resource-allocation graph

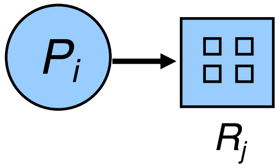
- Process (thread)



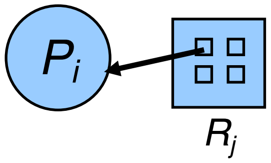
- Resource type with 4 instances



- P_i requests instance of R_j

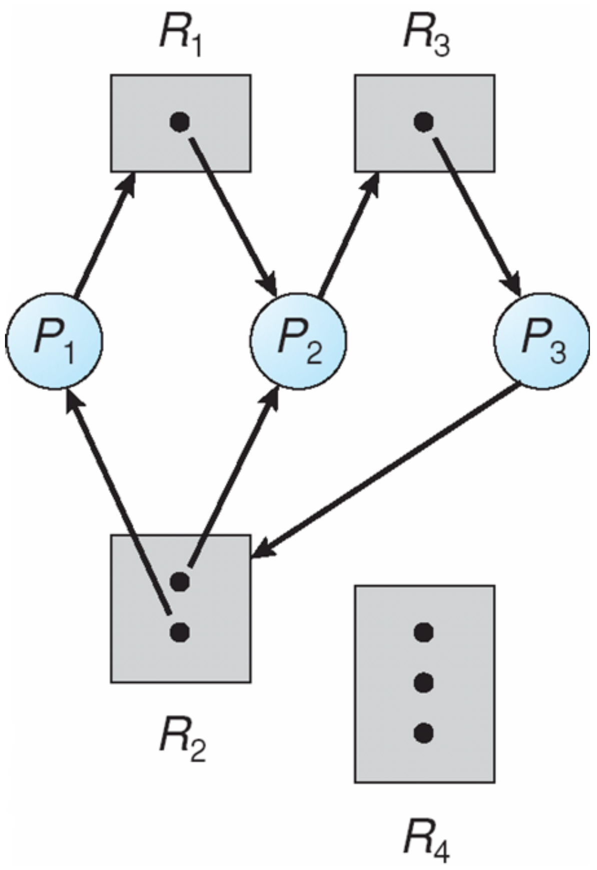


- P_i is holding an instance of R_j



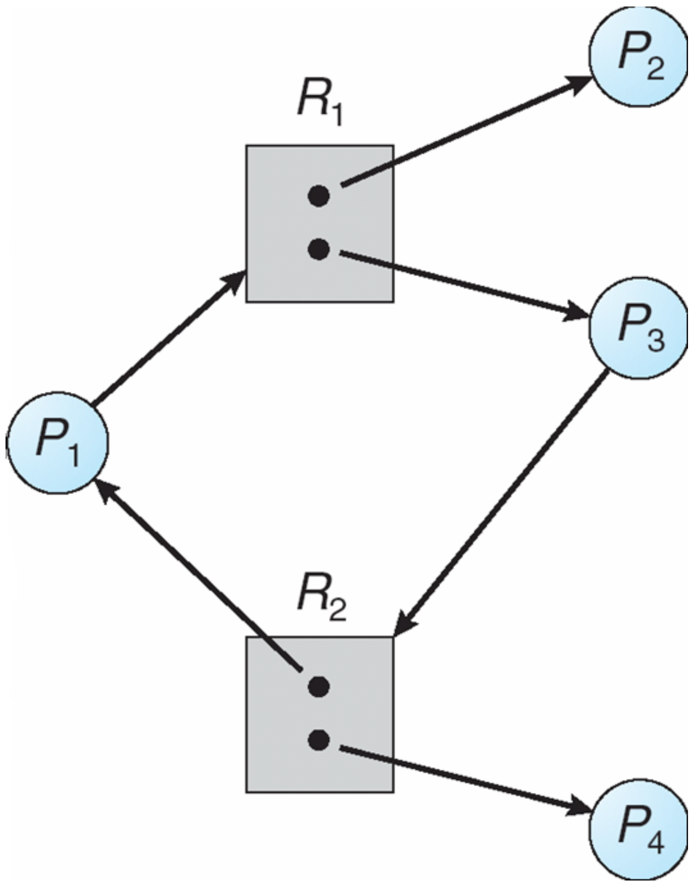
Resource-allocation graph with a deadlock

cycle – deadlock ?



There is a cycle and deadlock.

Graph with a cycle but no deadlock



There is a 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
    - Single Instance (SI) System.
  - if multiple instances per resource type, *possibility of deadlock*
    - Multiple Instances (MI) System.

Methods for Handling Deadlocks

- We can deal with the deadlock problem in one of three ways:
- 1) Ignore the problem and pretend that deadlocks never occur in the system;
  - Used by most operating systems, including UNIX.
  - OS does not bother with deadlocks that can occur in applications.
- 2) Ensure that the system will *never* enter a deadlock state
  - Deadlock prevention or deadlock avoidance methods.
- 3) Allow the system to enter a deadlock state and then recover.
  - Deadlock detection needed.

Deadlock Prevention

Basic Principle: arrange the ways requests can be made

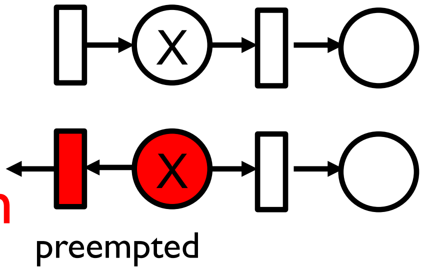
- We need to have 4 conditions for deadlocks to occur.
- We may try to negate one of these conditions.
  - Then we will not have deadlocks.
- Mutual Exclusion:
  - not required for sharable resources; must hold for non-sharable resources.
- How do we negate mutual exclusion?
  - Depends on the resource.
  - Example: spooling for printer

Deadlock Prevention

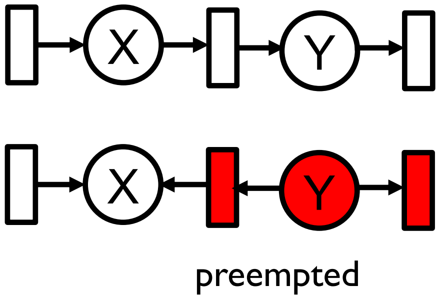
- Hold and Wait:
  - To negate hold and wait, we must guarantee that whenever a process requests a resource, it does not hold any other resource.
  - 1) require that a process to request and be allocated all its resources at the beginning.
  - 2) allow a process to request resources only when it has none.
- Negating hold and wait:
  - causes low low resource utilization.
  - starvation possible.

Deadlock Prevention

- No Preemption (how to negate)
  - 1) a thread X holding resources makes a request for new resources held by other threads. Then resoruces are preempted from X. X will wait. Hopefully some other thread can continue.



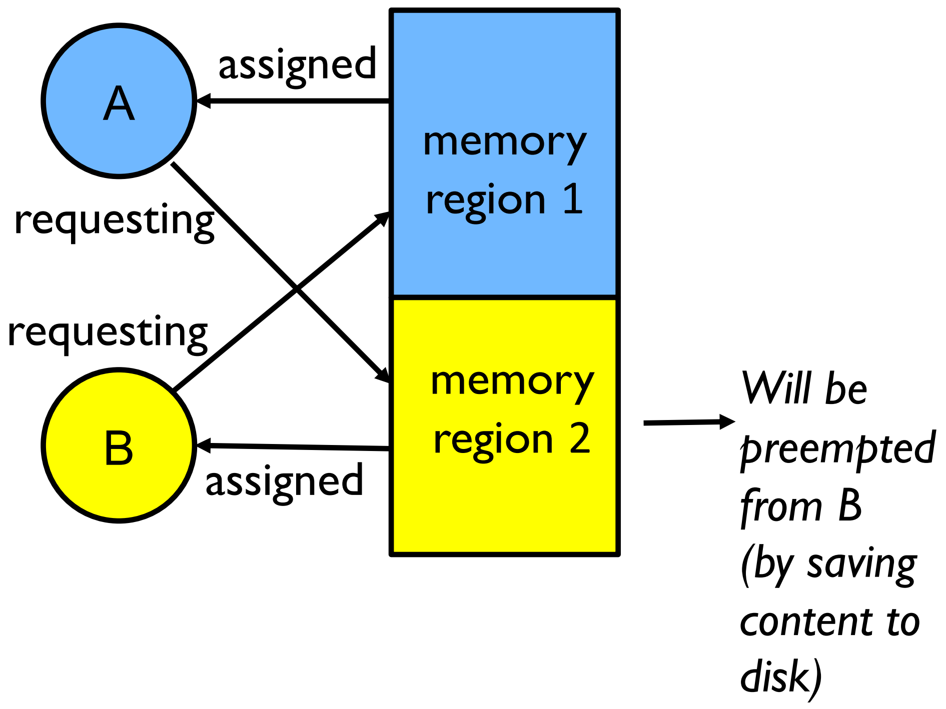
  - or 2) a thread X requests resources held by another thread Y which is waiting for other resource(s). Then resources are preempted from Y and given to X. Y will wait. X can continue.

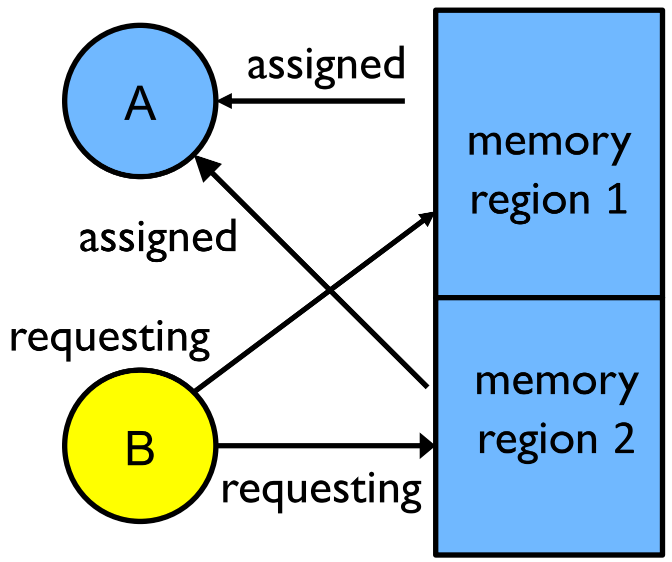


Preemption can be applied for resources whose state can be saved and restored (like CPU, memory).

Deadlock Prevention

Memory as a preemptable resource

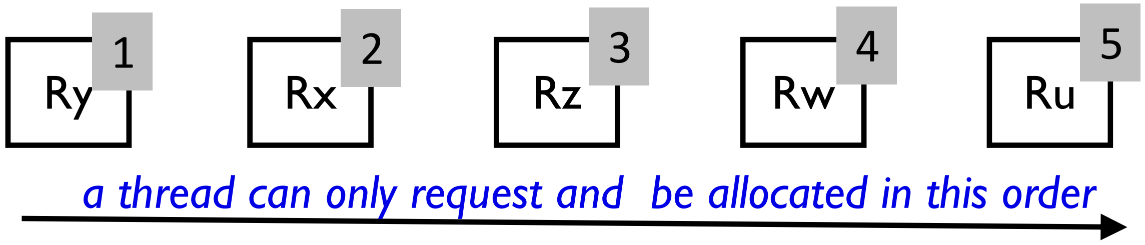




A can continue.
After A; B can use.

Deadlock Prevention

- Circular Wait - (how to negate)
  - Impose a total ordering for all resource types (resources enumerated).
  - a resource X is assigned a unique positive number $F(X)$.
  - Each thread requests resources in increasing order of enumeration.



Thread 1	Thread 2	Thread 3
Request Rx	Request Ry	Request Rz
Request Rw	Request Rx	Request Rw
	Request Rz	

Deadlock Prevention

- If all threads follow this protocol, we will not have a deadlock, since we will not have a circular wait.
- This is a practical scheme that we can use in our multi-threaded applications in which we use locks.
- For example, assume we have lockE, lockS, lockH.
  - We can think in our design that F(lockE) = 1, F(lockS) = 2, F(lockH) = 3.
  - Then, each thread needs to acquire the needed locks in same increasing order of enumeration:

```
pthread_mutex_lock (&lockE);
pthread_mutex_lock (&lockS);
pthread_mutex_lock (&lockH);
```

Deadlock Prevention

- Proof (by contradiction):
  - Resources are enumerated. $F(R_0) < F(R_1) <\dots. < F(R_N)$
  - Assume we have a circular wait as: T_0, R_0, T_1, R_1, …. R_{N-1}, T_N, R_N, T_0.
    - T_i is waiting for R_i held by T_{i+1}
    - T_{i+1} is holding R_i and waiting for R_{i+1} ➔ $F(R_i) < F(R_{i+1})$
      T_N is holding R_N and waiting for R_0 ➔ $F(R_N) < F(R_0)$
      by transitivity, $F(R_0) < \dots < F(R_N) < F(R_0)$
  - ➔ $F(R_0) < F(R_0)$ This is impossible. Therefore, we can not have a circular wait.

Deadlock Avoidance

Basic Principle: Requires the system to have some additional a priori information available from processes.

- Each process needs to declare the *maximum number* of resources of each type that it may need (simultaneously).
- We can call this as *maximum demand*.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe state

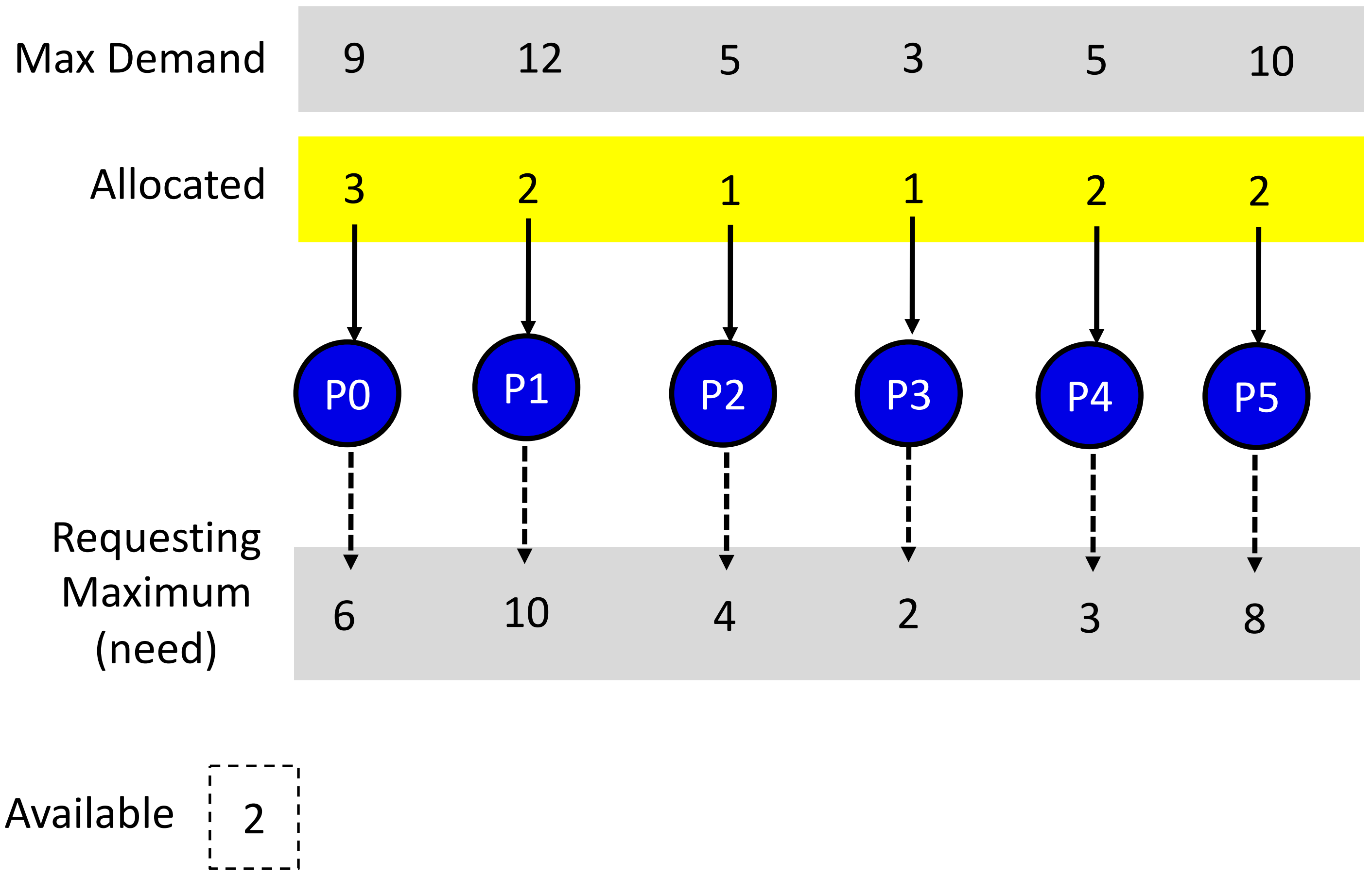
- When a process requests an available resource, system must check if immediate allocation leaves the system in a safe state.
- A state is safe if is possible for the system to allocate resources to each process (up to its maximum) in some order and avoid a deadlock.
- We are considering a worst-case situation here.
- Even in the worst case (processes request up to their maximum), we don’t have deadlock in a safe state.

Safe state

- More formally: A system state is safe if there exists a safe sequence $< P_1 , P_2 , \ldots, P_n >$ of all processes such that for each P_i, the resources that P_i *can still request* can be satisfied by *currently available resources* plus *resources held by all processes* P_j *earlier in the sequence* (all P_j , where $j < i$).
- That is, if there is a safe sequence:
  - If resources needed by a P_i are not immediately available, then P_i can wait (in the worst case) until all P_j $(j < i)$ have finished.
  - Then P_i can obtain needed resources, use them, and release all resources, in the worse case at termination.
  - Then, P_{i+1} can obtain its needed resources, and so on.

Deadlock avoidance: Safe state

Example
One resource type, many instances (13)



	P0	P1	P2	P3	P4	P5
Max Demand	9	12	5	3	5	10
Allocated	3	2	1	1	2	2
Requesting Maximum (need)	6	10	4	2	3	8

Available 2

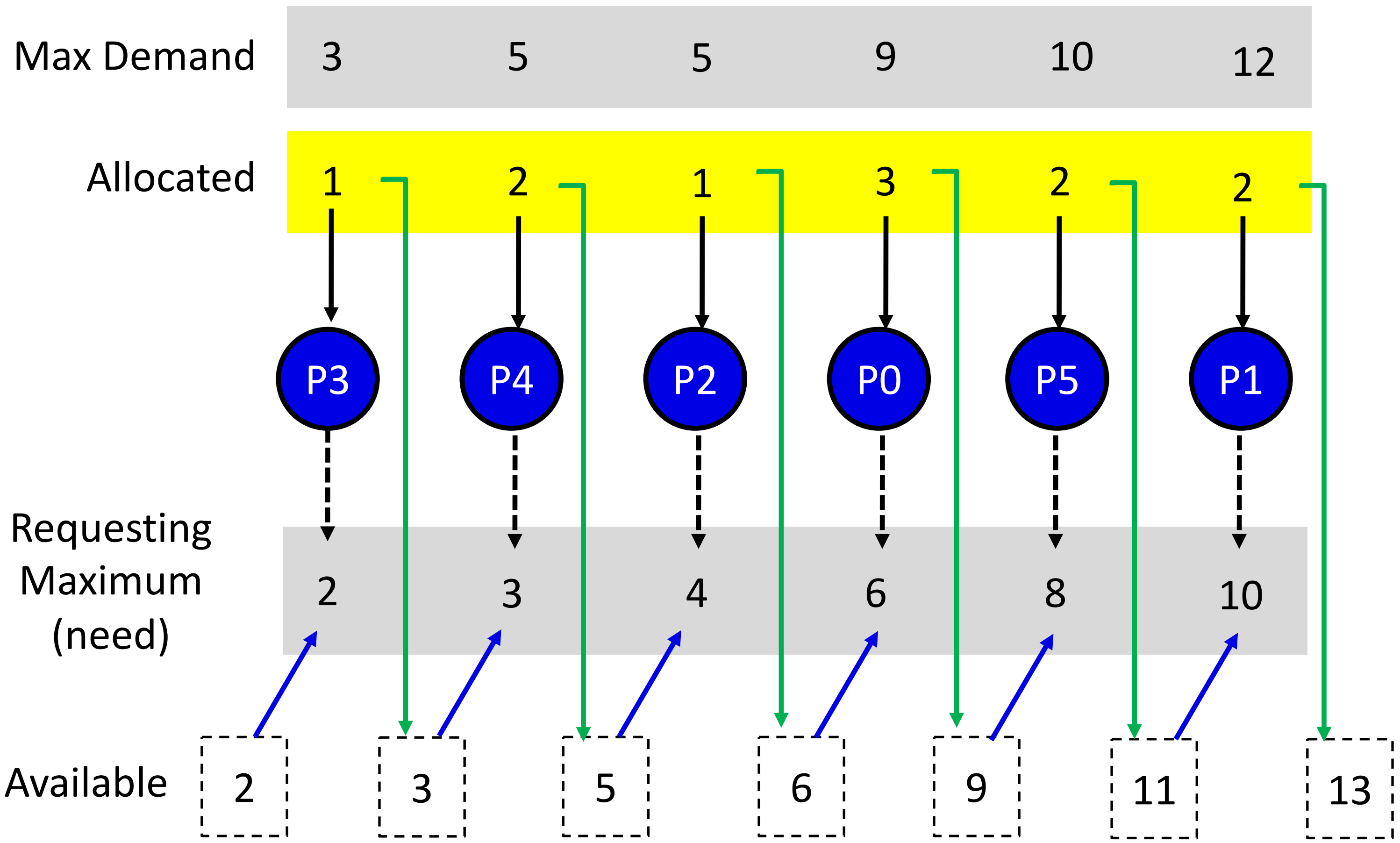
State

13 instances of a resource type exist in the system.

Is it safe state?

Example

One resource type, many instances

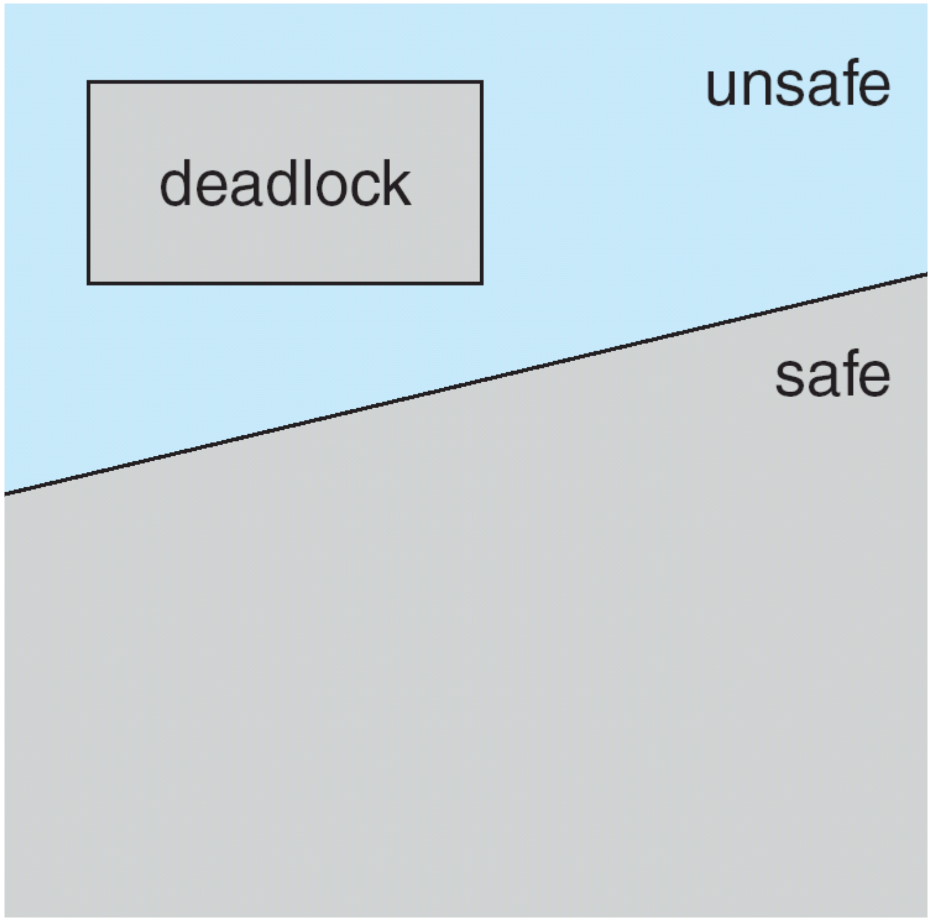


This state is safe. Order: P3, P4, P2, P0, P5, P1.

Basic Facts

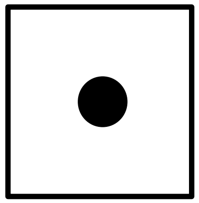
- If a system is in safe state, then no deadlocks.
- If a system is in unsafe state, then there is possibility of deadlock.
- Avoidance algorithms ensure that a system will never enter an unsafe state.
- This is done as follows. Each time a request is made by a process for some resources:
  - do a safety check before allocating resources.
  - if allocation would leave the system in an unsafe state, then do not allocate the resources. the process is waited.
  - if allocation would leave the system in a safe state, the resources are allocated.

Safe, Unsafe , Deadlock State

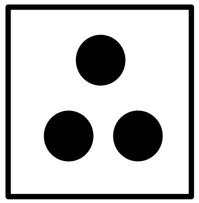


Avoidance Algorithms

- Single instance (SI) of a resource type
  - Use a resource-allocation-graph algorithm



- Multiple instances (MI) of a resource type
  - Use the Banker's algorithm



Resource-Allocation-Graph algorithm

- Claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j; represented by a dashed line.
  - At start, a process indicates the resources it may use (claims).
  - Claim edges added to the graph when process starts.
  - Resources must be claimed *a priori* in the system.
- When a resource is requested by a process:
  - Claim edge is converted to a request edge.
- When a requested resource is allocated:
  - Request edge is converted to an assignment edge.
- When a resource is released:
  - Assignment edge is reconverted to a claim edge

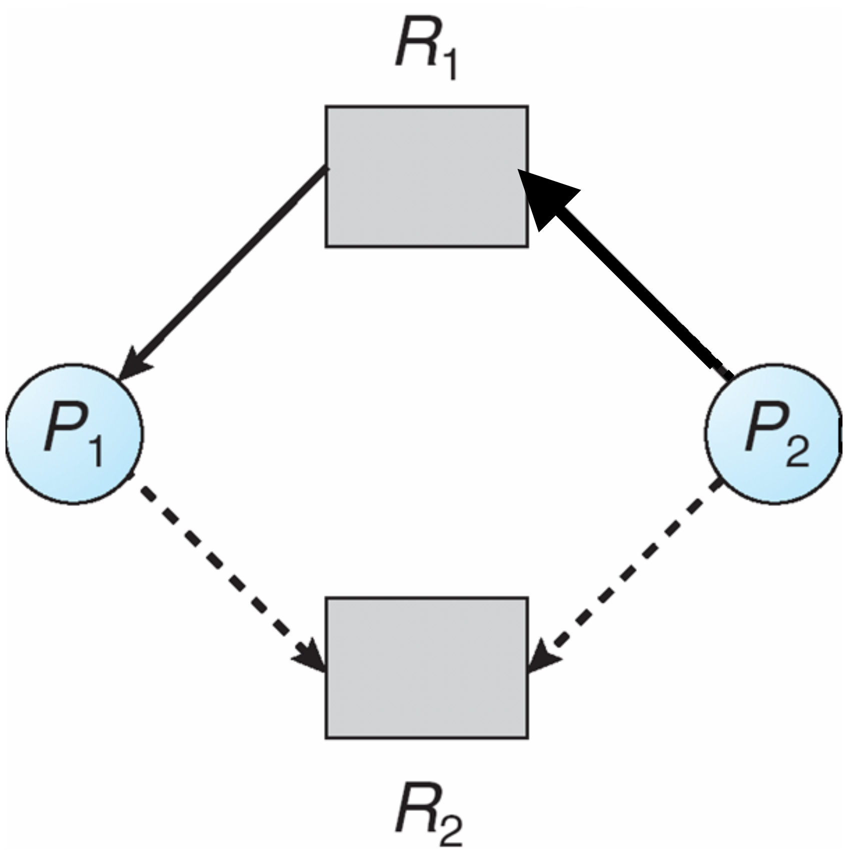
Resource-Allocation-Graph algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph.

Resource-Allocation-Graph algorithm

PROCESS P1	PROCESS P2
Request (R1)	
	Request (R1, R2)
....	....
Request (R2)	

time

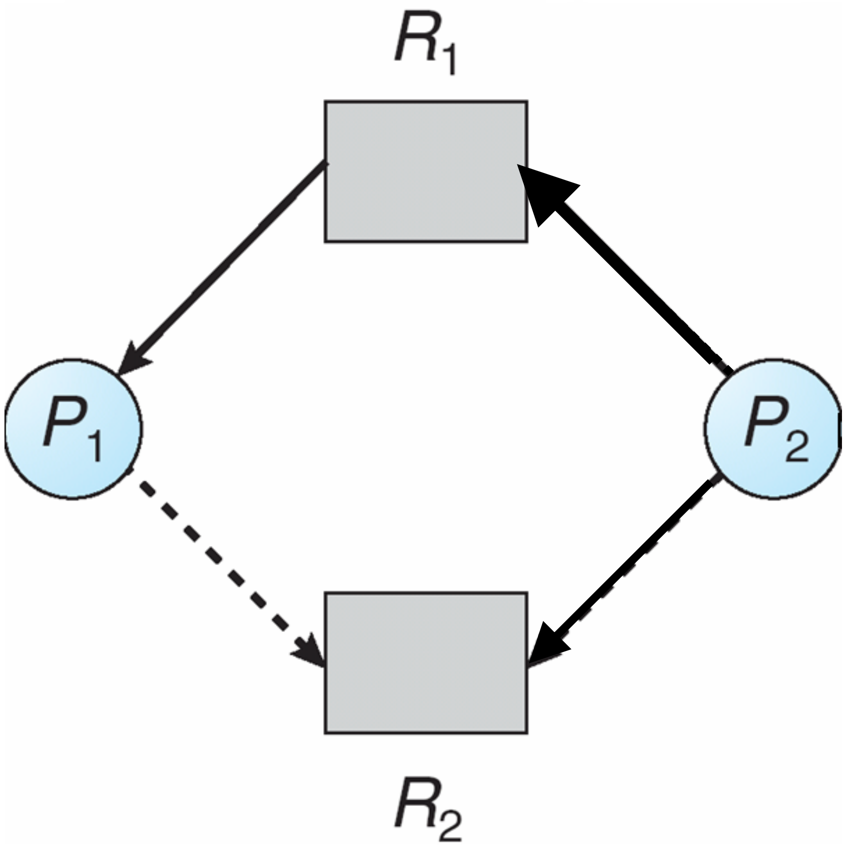


P2 requesting R1 and R2; R1 is not available.

Resource-Allocation-Graph algorithm

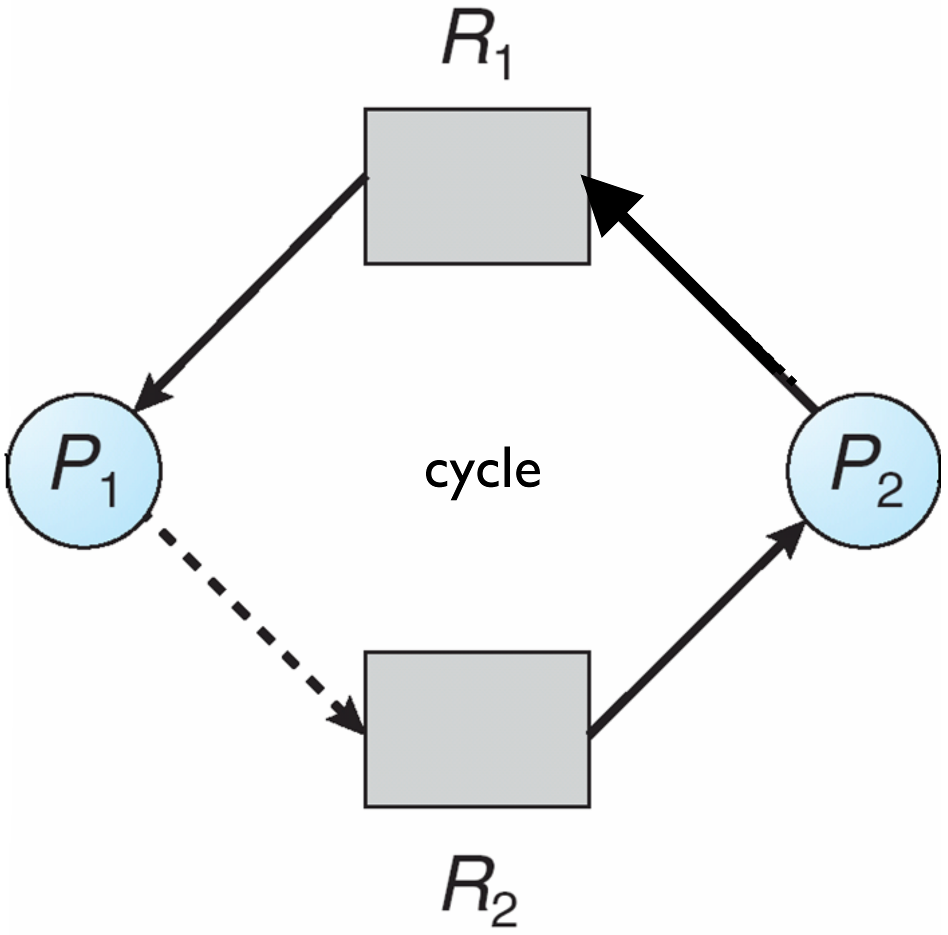
PROCESS P1	PROCESS P2
Request (R1)	
	Request (R1, R2)
……..	……..
Request (R2)	

time



P2 requesting R1 and R2; R2 is available; should we allocate?

Resource-Allocation-Graph algorithm unsafe state



If we allocate, there is a cycle. Hence, possibility of deadlock. Therefore, we do not allocate R2 to P2. P2 is waited to get R1 and R2.

Deadlock avoidance in a system with multiple resource intances

- If we can have multiple instances for each resource type, we can not use resource-allocation-graph algorithm.
- System state is represented with matrices in this case.
- An algorithm, called Banker's algoritm, can be used to avoid deadlocks.

Banker's Algorithm

- There may be multiple instances for a resource type.
- Each process must a priori claim maximum use.
  - Process tells to the system at most how many instances it may need, for each resource type.
  - In this way, the system will know maximum demand of each process (worst-case demand).
- When a process requests a resource it may have to wait.
  - Even if resource is available
- When a process gets all its resources, it must return them in a finite amount of time.
  - Processes are not running forever.

Simple Example

- Assume a resource A (printer) has 5 instances.
- There are 2 processes: P1, P2
- Max demand is: 5 5
- Current Allocation is: 3 0

- Need is: 2 5
- Available is: 2
- Is the current state safe? Yes
  - Even each process makes its worst-case request, they will not be deadlocked. The can execute in the following order: P1, P2.
  - Therefore the state is safe.

Simple Example

- Assume P2 requests 1 instance and it is granted. Is the new state safe?
- Current Allocation will be : 3 1
- Need is: 2 4
- Available is: 1
- Is the current state safe? No
  - No process can be satisfied each process makes its worst-case request (maximum request)?
  - They would be deadlocked in that case.
  - Therefore this is not a safe state.

Data Structures for the Banker's Algorithm representing system state

n = number of processes

m = number of resource types.

- Available: Vector of length m. If $Available[j] == k$, there are k instances of resource type R_j at the time deadlock avoidance algorithms is run.
- Max: $n \times m$ matrix. If $Max[i, j] == k$, then process P_i may request at most k instances of resource type R_j.
- Allocation: $n \times m$ matrix. If $Allocation[i, j] == k$ then P_i is currently allocated k instances of R_j.
- Need: $n \times m$ matrix. If $Need[i, j] == k$, then P_i may need k more instances of R_j to complete its task.

$$Need[i, j] = Max[i, j] - Allocation[i, j]$$

An example system state

All resources in the system

Existing
A B C
10 5 7

Available
A B C
10 5 7

Initially Available == Existing

system state at some time t (may change)

Need = Max - Allocation

	Max A B C
P0	7 5 3
P1	3 2 2
P2	9 0 2
P3	2 2 2
P4	4 3 3

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

	Need A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available
A B C
3 3 2

Notation

	X A B C
P0	0 1 0
P1	2 0 0
P2	3 0 2
P3	2 1 1
P4	0 0 2

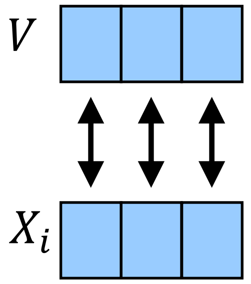
X is a matrix.

X_i is the i^{th} row of the matrix: it is a vector.
For example, X_3 = [2 1 1]

V A B C
3 3 2

V is a vector; V = [3 3 2]

Compare two vectors:
Ex: compare V with X_i



$V == X_i$?
$V \leq X_i$?
$X_i \leq V$?
....

Ex: Compare [3 3 2] with [2 2 1]

[2 2 1] <= [3 3 2]

Safety Check Algorithm

1. Let $Work$ (or *Pool*) and $Finish$ be vectors of length m and n, respectively. Initialize:

   $Work = Available;\quad Finish[i] = false$ for i = 0, 1, …, $n - 1$

   $Work$ is a temporary vector initialized to $Available$ (i.e., free) resources at the time of safety check. It represents the available *Pool* of resources.

2. Find an i such that both:

   (a) $Finish[i] = false$

   (b) $Need_i \leq Work$

   If no such i exists, go to step 4

3. $Work = Work + Allocation_i$

   $Finish[i] = true$

   go to step 2.

4. If $Finish[i] == true$ for all i, then the system state is safe; otherwise it is unsafe.

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

	Need A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available
[3 3 2]

Resource-Request Algorithm for Process P_i

Let $Request_i$ be the request vector for process P_i.

if $Request_i[j] == k$, then process P_i wants k instances of resource type R_j.

When a process P_i makes such a request, the system executes the following algorithm.

Algorithm

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

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

Resource-Request Algorithm for Process P_i

- 3. Pretend to allocate the requested resources to P_i by calculating the new state as follows.
  - New state calculation:

    $$Available = Available - Request_i;$$
    $$Allocation_i = Allocation_i + Request_i;$$
    $$Need_i = Need_i - Request_i;$$

  - Run the safety check algorithm on the new state.
  - If the new state is safe, then the requested resources are allocated to P_i. We go to new state.
  - If the new state is unsafe, then the requested resources are not allocated to P_i. P_i must wait. We don't go to the new state. Old state is restored. We stay in the old (current) state.

Example of Banker' s Algorithm

- 5 processes P_0 through P_4;
  3 resource types: A, B, and C

Existing Resources: A (10 instances), B (5 instances), and C (7 instances)

Existing = [10, 5, 7]

initially, Available = Existing.

Assume, processes indicated their maximum demand as follows:

	Max A B C
P0	7 5 3
P1	3 2 2
P2	9 0 2
P3	2 2 2
P4	4 3 3

Initially, Allocation matrix will be all zeros. Need matrix will be equal to the Max matrix.

Example of Banker's Algorithm

- Assume later, at an arbitrary time t, we have the following system state:

Existing = [10 5 7]

$$Need = Max - Allocation$$

	Max A B C
P0	7 5 3
P1	3 2 2
P2	9 0 2
P3	2 2 2
P4	4 3 3

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

	Need A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available A B C
3 3 2

Is it a safe state?

Example of Banker’s Algorithm

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

	Need A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available A B C
3 3 2

Try to find a row in $Need_i$ that is $\leq$ Available.

P1. run completion. Available becomes = [3 3 2] + [2 0 0] = [5 3 2]
P3. run completion. Available becomes = [5 3 2] + [2 1 1] = [7 4 3]
P4. run completion. Available becomes = [7 4 3] + [0 0 2] = [7 4 5]
P2. run completion. Available becomes = [7 4 5] + [3 0 2] = [10 4 7]
P0. run completion. Available becomes = [10 4 7] + [0 1 0] = [10 5 7]

We found a sequence of execution: P1, P3, P4, P2, P0. State is safe

Example: P_1 requests (1,0,2)

- At that time Available is [3 3 2]
- First check that there are enough resources available: ([1,0,2] ≤ [3,3,2])
- Then calculate the new state and check if it is a safe state.

	Max A B C
P0	7 5 3
P1	3 2 2
P2	9 0 2
P3	2 2 2
P4	4 3 3

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

	Need A B C
P0	7 4 3
P1	0 2 0
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available A B C
2 3 0

new state (we did not go to that state yet; we are just checking)

Example: P_1 requests (1,0,2)

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

	Need A B C
P0	7 4 3
P1	0 2 0
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available A B C
2 3 0

new state

Can we find a sequence of execution even processes make their maximum request?

Run P1. Available becomes = [5 3 2]
Run P3. Available becomes = [7 4 3]
Run P4. Available becomes = [7 4 5]
Run P0. Available becomes = [7 5 5]
Run P2. Available becomes = [10 5 7]

Yes we could find a sequence.
P1, P3, P4, P0, P2
New State is safe.
We can grant the request: allocate the desired resources to process P1.

P_4 requests (3,3,0)?

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

	Need A B C
P0	7 4 3
P1	0 2 0
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available A B C
2 3 0

Current state

If this is current state, what happens if P4 requests (3 3 0)?

There is no available resource to satisfy the request. P4 will be waited.

P_0 requests (0,2,0)? Should we grant?

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

	Need A B C
P0	7 4 3
P1	0 2 0
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available A B C
2 3 0

Current state

System is in this state.
P_0 makes a request: [0, 2, 0]. Should we grant?

P_0 requests (0,2,0)? Should we grant?

Assume we allocate [0,2,0] to P0. The new state will be as follows.

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

	Need A B C
P0	7 2 3
P1	0 2 0
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available A B C
2 1 0

New state

Is it safe?

No process has a row in Need matrix that is less than or equal to Available. Therefore, the new state is unsafe. Hence we should not go to the new state. The request is not granted. P_0 is waited.

Deadlock Detection

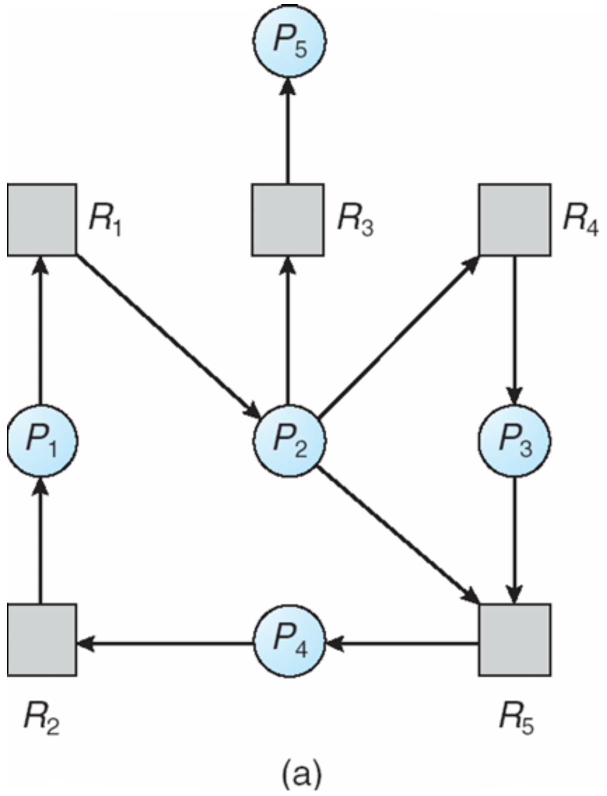
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

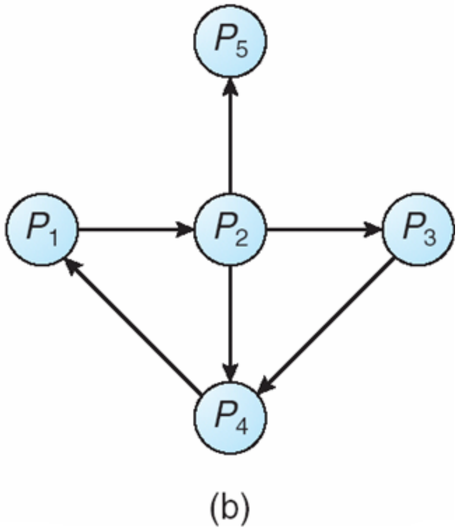
Single Instance of Each Resource Type

- 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.
- Cycle detection algorithm complexity: O(n^2). n is number of vertices.

Single Instance of Each Resource Type





resource allocation graph

wait-for-graph

Several Instances of Resource Type

- Available: A vector of length m indicates the number of available resources of each type.
- Allocation: An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- Request: An $n \times m$ 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.

System state is represented by the information above.

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.
   $Finish[i] = FALSE$

$Work$: represents the available pool

2. Find an index i such that
   $Finish[i] == FALSE$ *and* $Request_i \leq Work$
   If no such i exists, go to step 4
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.

Detection Algorithm

- In detection algorithm, we are taking an optimistic approach.
  - If the request of a process is satisfiable with available pool of resources, then the process runs and finishes and releases all its resources (we have such a chance; most optimistic case). Available pool is increased further. Then maybe another process can start running.
- Hence we are looking for a possibility if the processes can execute sooner or later in some order.
- If there is no such possibility (even we are considering the most optimistic cases), then there is deadlock at the moment.
- If there is such a possibility, there is no deadlock at the moment. There can be later.

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*:

Existing
A B C
7 2 6

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

	Request A B C
P0	0 0 0
P1	2 0 2
P2	0 0 0
P3	1 0 0
P4	0 0 2

Available A B C
0 0 0

Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*

Example of Detection Algorithm

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

	Request A B C
P0	0 0 0
P1	2 0 2
P2	0 0 0
P3	1 0 0
P4	0 0 2

Available A B C
0 0 0

Can we find a row i in Request that can be satisfied with Available, i.e. Request_i <= Available?

P0 is not deadlocked at the moment. Run completion. Available becomes: [0 1 0]
Then P2 can be satisfied. Can run completion. Available becomes: [3 1 3]
Then P3 can be satisfied. Can run completion. Available becomes: [5 2 4]
Then P1 can be satisfied. Can run completion. Available becomes: [7 2 4]
Then P4 can be satisfied. Can run completion. Available becomes: [7 2 6]

Another example

- Lets assume at time t2, P_2 makes a request for an additional instance of type C. Then Request matrix becomes"

	Request A B C
P0	0 0 0
P1	2 0 2
P2	0 0 1
P3	1 0 0
P4	0 0 2

Is the system deadlocked? Check it.

Checking for Deadlock

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

	Request A B C
P0	0 0 0
P1	2 0 2
P2	0 0 1
P3	1 0 0
P4	0 0 2

Available A B C
0 0 0

We can run P0. Then Available becomes: [0 1 0]
Now, we can not find a row of Request that can be satisfied.
Hence all processes P1, P2, P3, and P4 have to wait in their requests. We have a deadlock.

Processes P1, P2, P3, and P4 are deadlocked processes.

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle.

- a) Execute detection algorithm at each request. You will know which process caused the deadlock. Costly.
- b) If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock

- We can recover by terminating all or some processes.
  - a) abort all deadlocked processes.
  - b) abort one process at a time until the deadlock cycle is eliminated. In which order should we abort? Factors?
    - Priority of the process.
    - How long process has computed, and how much longer to completion.
    - Resources the process has used.
    - Resources process needs to use.
    - How many processes will need to be terminated.
    - Is process interactive or batch.

Recovery from Deadlock

- We can recover by preempting resource(s) from a (victim) process.
  - Selecting a victim. Criteria: minimize cost.
  - Rollback for the victim. a) return to some safe state; restart process from that state (not easy). b) restart process from beginning.
  - Starvation – same process may always be picked as victim. Solution: include number of rollbacks in cost factor.

References

- Operating System Concepts, Silberschatz et al. Wiley.
- Modern Operating Systems, Andrew S. Tanenbaum et al.
- OSTEP: Remzi Arpaci-Dusseau et al.
- Operating Systems: Principles and Practice.