

Chapter - three

Deadlock Management

Outline

- Deadlock: definition
- Conditions for Deadlock
- Deadlock examples
- Starvation
- Resource types
- Methods for handling deadlock
 - Ostrich algorithm
 - Deadlock detection and recovery
 - Deadlock prevention
 - Deadlock avoidance

Deadlock

- For many applications, a process needs exclusive access to not one resource, but several.
- Suppose, for example, two processes each want to record a scanned document on a CD.
 - Process A requests permission to use the scanner and is granted it.
 - Process B is programmed differently and requests the CD recorder first and is also granted it.
- Now A asks for the CD recorder, but the request is denied until B releases it.
- Unfortunately, instead of releasing the CD recorder B asks for the scanner.
- At this point both processes are blocked and will remain so forever.
- This situation is called a deadlock.

Deadlock...

- **Deadlock** can be defined formally as follows:
 - A set of processes is deadlocked if each member of the set of deadlocked processes is waiting for a resource that is owned by a deadlocked process.
 - None of the processes can run,
 - none of them can release any resources, and
 - all the processes continue to wait forever.

Conditions for Deadlock

- Four conditions must hold for there to be a deadlock:
 1. **Mutual exclusion condition.** Each resource is either currently assigned to exactly one process or is available.
 2. **Hold and wait condition.** Processes currently holding resources granted earlier can request new resources.
 3. **No preemption condition.** Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
 4. **Circular wait condition.** There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.
- All four of these conditions must be present for a deadlock to occur.
- If one of them is absent, no deadlock is possible.

Possibility of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait

Existence of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait
- Circular wait

Deadlock Examples

- examples
 - studying students
 - traffic intersection
 - airline reservation system...
- evaluation
 - four conditions: mutual exclusion, hold and wait, no preemption, circular wait

Studying Students

- ◆ **studying students**: both students need the textbook and the course notes to study, but there is only one copy of each
- ◆ consider the following situation:

Student A

get coursenotes

get textbook

study

release textbook

release coursenotes

Student B

get textbook

get coursenotes

study

release coursenotes

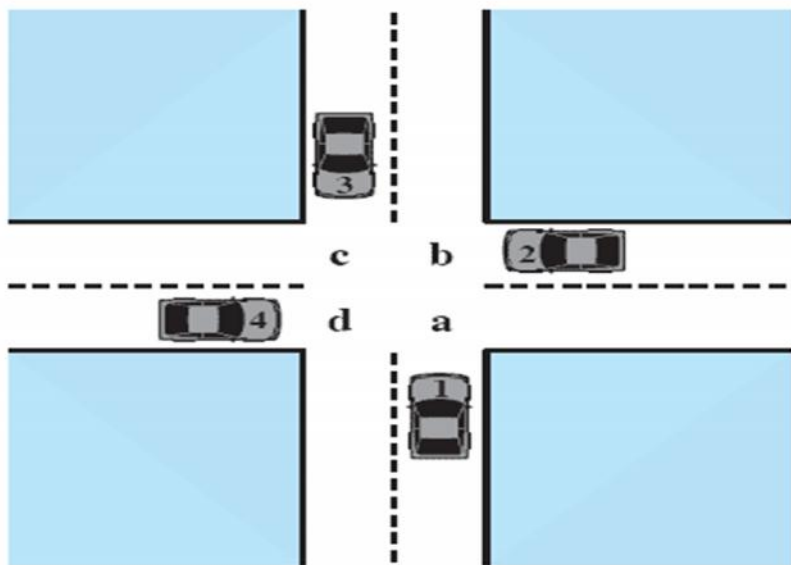
release textbook

Students Evaluation

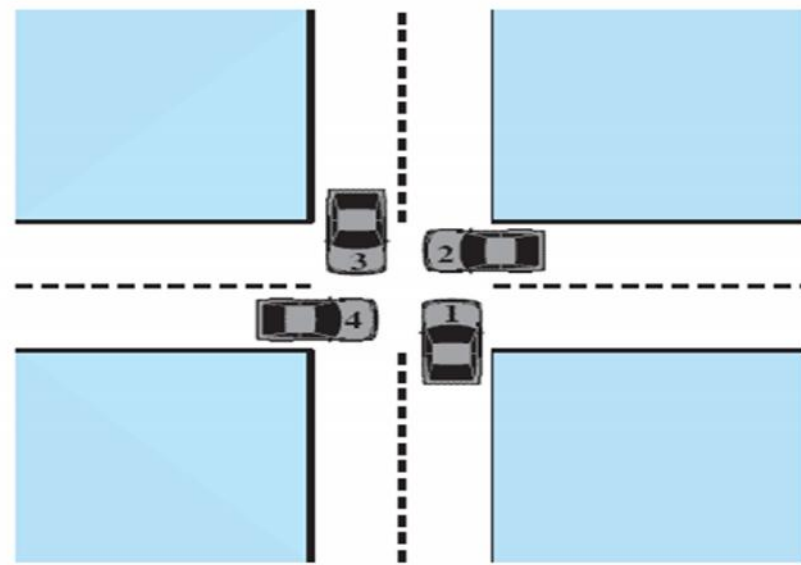
- **mutual exclusion**
 - books and course notes can be used only by one student
- **hold and wait**
 - a student who has the book waits for the course notes, or vice versa
- **no preemption**
 - there is no authority to take away book or course notes from a student
- **circular wait**
 - student A waits for resources held by student B, who waits for resources held by A

Traffic Intersection

- at a four-way intersection, four cars arrive simultaneously
- if all proceed, they will be stuck in the middle



(a) Deadlock possible



(b) Deadlock

Traffic Evaluation

- **mutual exclusion**
 - cars can't pass each other in the intersection
- **hold and wait**
 - vehicles proceed to the center, and wait for their path to be clear
- **no preemption**
 - there is no authority to remove some vehicles
- **circular wait**
 - vehicle 1 waits for vehicle 2 to move, which waits for 3, which waits for 4, which waits for 1

Starvation

- a process can't proceed because other processes always have the resources it needs
- the request of the process is never satisfied
- in principle, it is possible to get the resource, but doesn't happen because of
 - low priority of the process
 - timing of resource requests
 - ill-designed resource allocation or scheduling algorithm
- different from deadlock

Examples Starvation

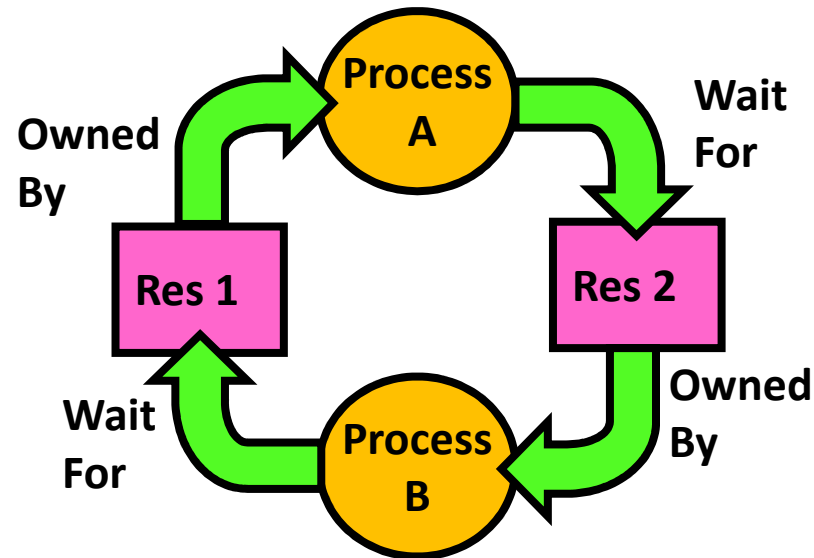
- batch processes with low priority in a heavily used time-sharing system
- crossing a very busy road
- trying to call a very popular phone number
 - radio station giveaways
- getting into a very popular course with limited enrollment

Solution Starvation

- **fairness**: each process gets its fair share of all resources it requests
 - how is fair defined?
 - how is it enforced?
- **aging**
 - the priority of a request is increased the longer the process waits for it

Starvation vs Deadlock

- **Starvation**: process waits indefinitely
 - Example, low-priority process waiting for resources constantly in use by high-priority processes
- **Deadlock**: circular waiting for resources
 - Process A owns Res 1 and is waiting for Res 2
 - Process B owns Res 2 and is waiting for Res 1



- **Deadlock** \Rightarrow **Starvation** but not vice versa
 - Starvation can end (but doesn't have to)
 - Deadlock can't end without external intervention

Resource Types

- **reusable resources**
 - can be used repeatedly by different processes
 - does not imply simultaneous use
 - OS examples: CPU, main memory, I/O devices, data structures
- **consumable resources**
 - are produced by one entity, and consumed by another
 - reuse is not possible, or impractical
 - OS examples: messages

Example Reusable Resources

- **main memory allocation**
 - two processes make successive requests for main memory
 - the overall memory size is 16 MByte

Process A

request 5 MBytes;
request 8 MBytes;

Process B

request 7 MBytes;
request 7 MBytes;

❖ **deadlock**

- ◆ no process can proceed unless one gives up some memory (preemption)
- ◆ frequent solutions: preallocation, virtual memory

Example Consumable Resources

- **message passing**

- two processes wait for a message from each other, and then send one to each other
- receive operation is blocking (process can't continue)

Process A

```
receive (B);  
send (B);
```

Process B

```
receive (A);  
send(A);
```

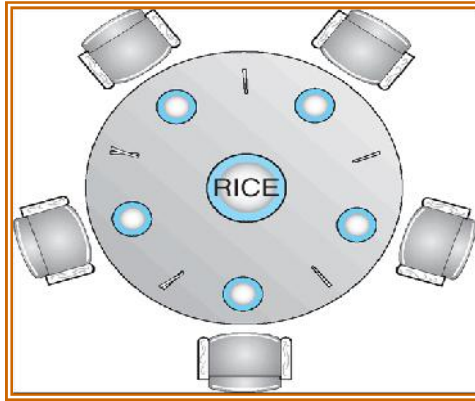
- ◆ **deadlock**

- ◆ no process can proceed because it is waiting for a message from the other
- ◆ no easy solution

Dining Philosophers

- Philosophers sitting around a dining table
- Philosophers only eat and think
- Need two forks to eat
- Exactly as many forks as philosophers
- Before eating, a philosopher must pick up the fork to his right and left
- When done eating, each philosopher sets down both forks and goes back to thinking

Dining Philosophers...



- Five chopsticks/Five Philosophers
 - Free-for all: Philosophers will grab any one they can
 - Need two chopsticks to eat
- What if all grab at same time?
 - Deadlock!
- How to fix deadlock?
 - Make one of them give up a chopstick
 - Eventually everyone will get chance to eat
- How to prevent deadlock?
 - Never let philosopher take last chopstick if no hungry philosopher has two chopsticks afterwards

Dining Room Philosophers...

- Only one philosopher can hold a fork at a time
- One major problem
- what if all philosophers decide to eat at once?
 - if they all pick up the right fork first, none of them can get the second fork to eat
 - deadlock

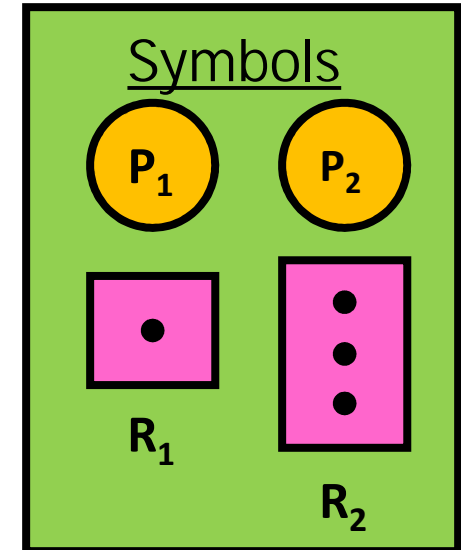
Philosopher Deadlock Solutions

- Make every even numbered philosopher pick up the right fork first and every odd numbered philosopher pick up the left fork first
- Don't let them all eat at once
 - a philosopher has to enter a monitor to check if it is safe to eat
 - each philosopher checks and sets some state indicating their condition

Resource-Allocation Graph

- **System Model**

- A set of **Processes** P_1, P_2, \dots, P_n
- **Resource** types R_1, R_2, \dots, R_m
 - CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
 - Request() / Use() / Release()

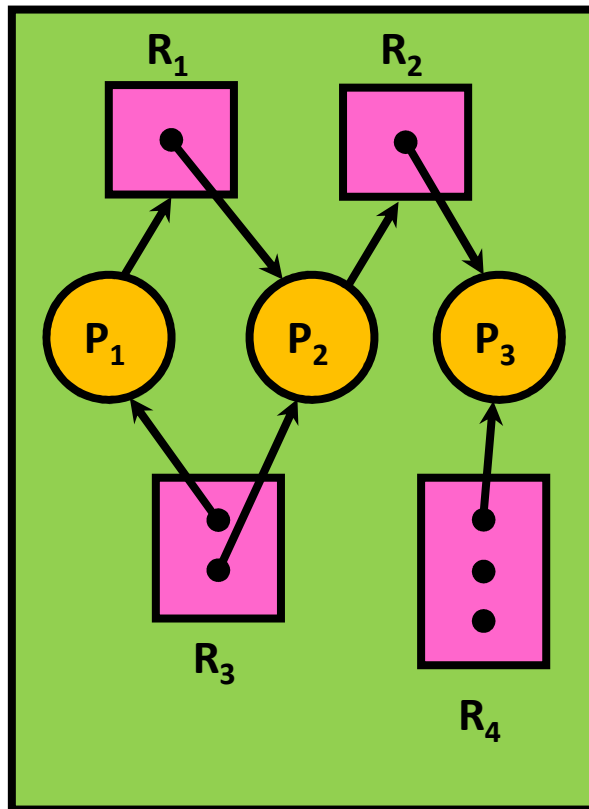


- **Resource-Allocation Graph:**

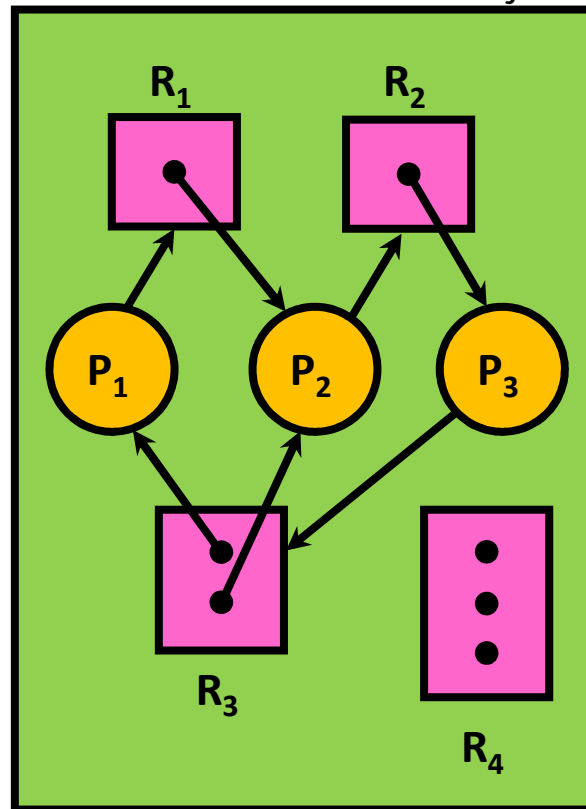
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set processes in the system.
 - $R = \{R_1, R_2, \dots, R_m\}$, the set of resource types in system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource Allocation Graph Examples

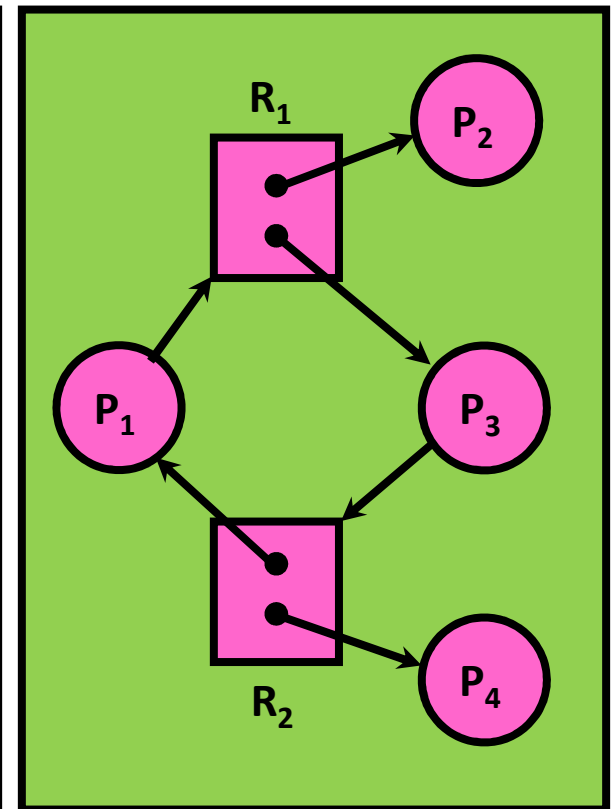
- Recall:
 - request edge – directed edge $P_1 \rightarrow R_j$
 - assignment edge – directed edge $R_j \rightarrow P_i$



Simple Resource Allocation Graph



Allocation Graph With Deadlock



Allocation Graph With 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.
 - if several instances per resource type, possibility of deadlock.

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state.
 - have strict rules that prevent a deadlock from occurring
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; frequently used by most operating systems, like
 - Windows, MacOS, Unix, ...

What kind of resources?

- Resources come in two types:
 - Pre-emptible and non pre-emptible.
- A pre-emptible resource is one that can be taken away from the process owning it with no ill effects.
 - **Memory** is an example of a pre-emptible resource.
- A nonpreemptable resource, in contrast, is one that cannot be taken away from its current owner without causing the computation to fail.
 - Burning file into CD
- **Deadlocks** involve in nonpreemptable resources.

What kind of resources?

- Potential deadlocks that involve **pre-emptible** resources can usually be resolved:
 - by reallocating resources from one process to another.
- Thus our treatment will focus on **nonpreemptable** resources:
- The sequence of events required to use a resource is given below in an abstract form.
 1. **Request** the resource.
 2. **Use** the resource.
 3. **Release** the resource.

Deadlock modeling

- The above four conditions can be modeled using directed graphs.
- The graphs have two kinds of nodes:
 - **Processes**, shown as circles, and
 - **Resources**, shown as squares.
- -An arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process.
- Resource R is currently assigned to process A.



Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock

Deadlock modeling...

- An arc from a process to a resource means that the process is currently blocked waiting for that resource. In Fig. (b), process B is waiting for resource S.
- In Fig. (c) We see a **deadlock**: process C is waiting for resource T, which is currently held by process D. Process D is not about to release resource T because it is waiting for resource U, held by C. Both processes will wait forever.
- A **cycle** in the graph means that there is a deadlock involving the processes and resources in the cycle (assuming that there is one resource of each kind).
- In this example, the cycle is **C-T-D-U-C**.

Deadlock modeling (Example)...

A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

a)

b)

c)



The *Operating system* is free to run any unblocked process at any instant, so it could decide to run A until A finished all its work, then run B to completion, and finally run C.

This ordering does not lead to any deadlocks.

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
Deadlock

d)

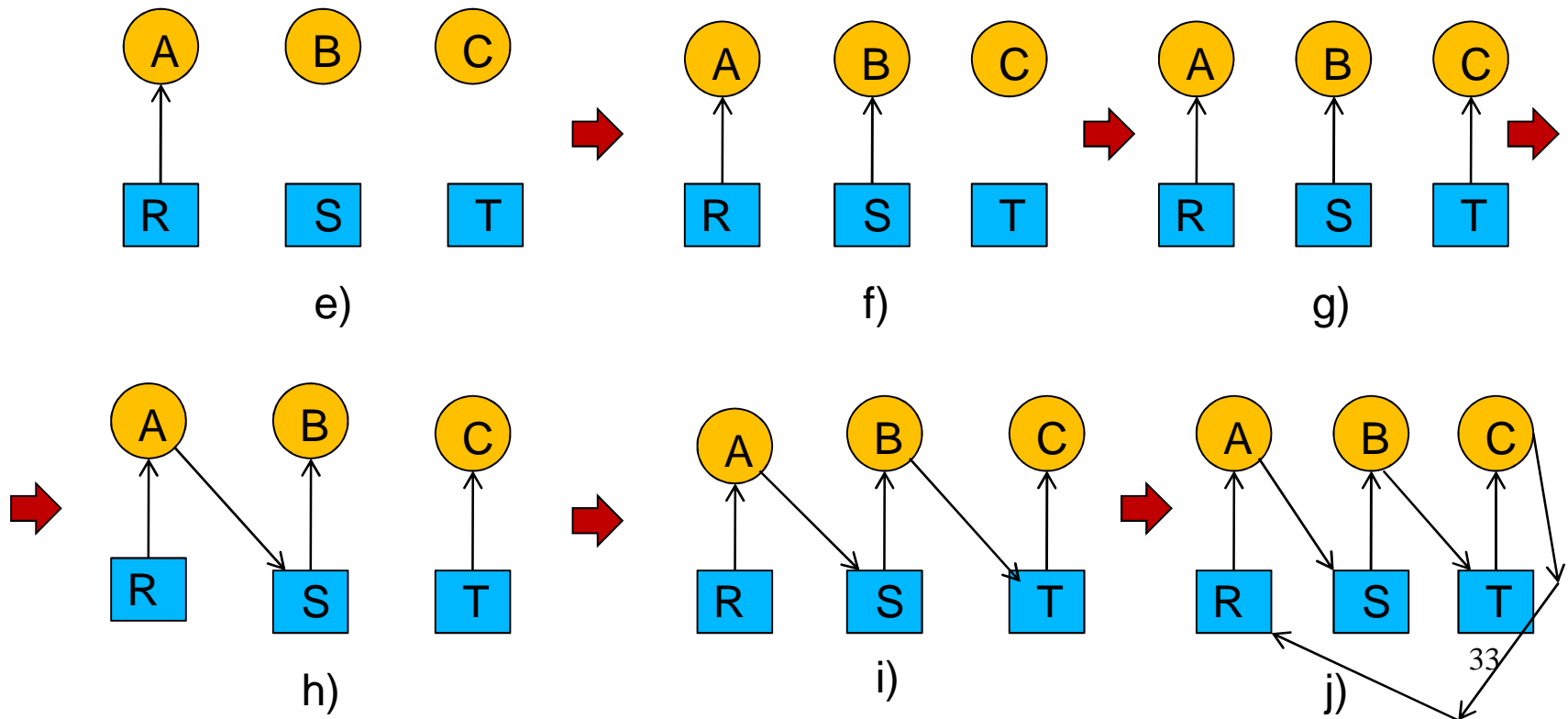


suppose that the processes do both I/O and computing, so that round robin is a reasonable scheduling algorithm. The resource requests might occur in the order

- If these six requests are carried out in this order, the six resulting resource graphs are shown in Fig. (e)-(j).

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
Deadlock

Deadlock modeling...

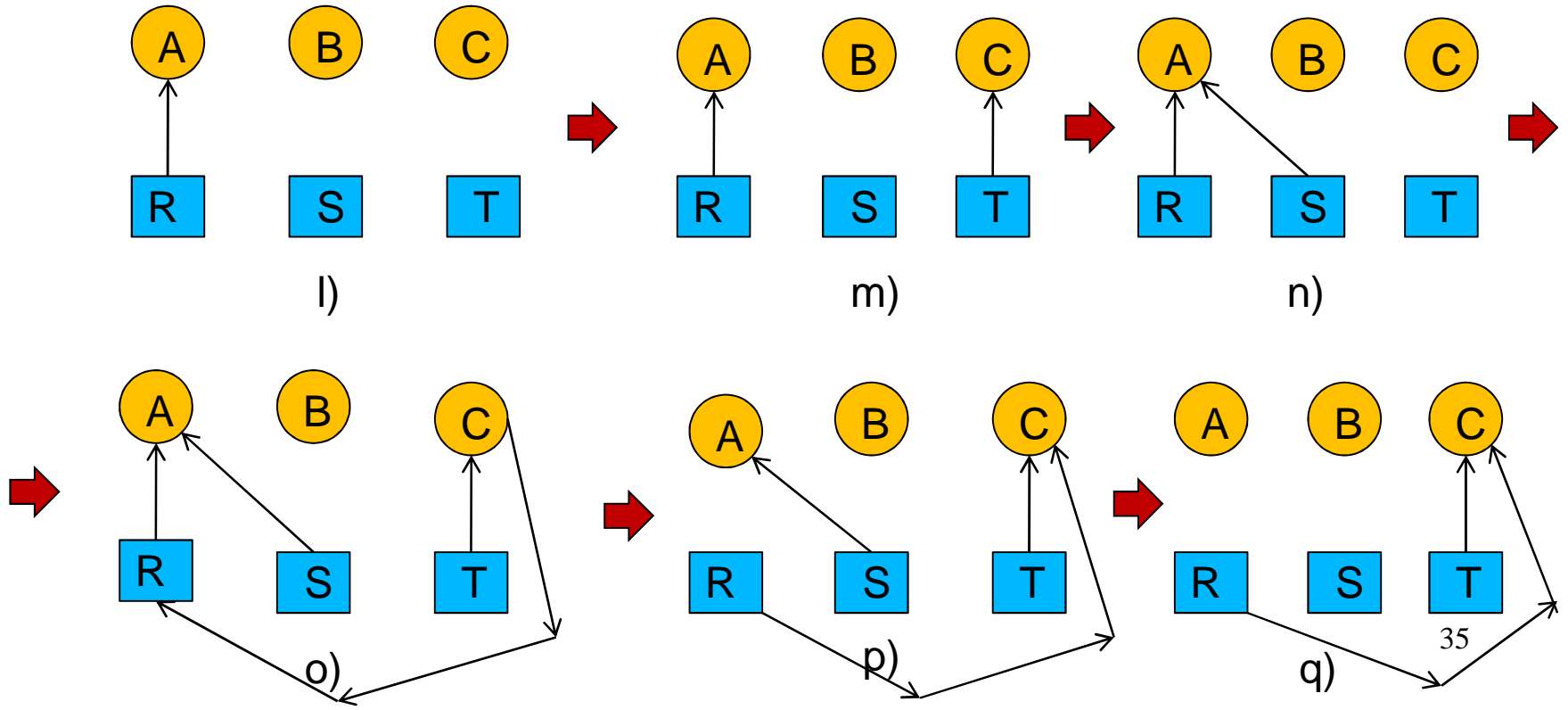


Deadlock modeling...

- If granting a particular request might lead to deadlock, the operating system can simply suspend the process without granting the request.
 - i.e., just not schedule the process until it is safe.
- If the operating system knew about the impending deadlock, it could suspend B instead of granting it S.
- By running only A and C, we would get the requests and releases of (k) instead of Fig. (d).

Deadlock modeling...

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
No deadlock



Four strategies are used for dealing with deadlocks:

- 1- **Just ignore the problem altogether** - Maybe if you ignore it, it will ignore you.
- 2- **Detection and recovery** - Let deadlocks occur, detect them, and take action.
- 3- **Dynamic avoidance** by careful resource allocation.
- 4- **Prevention** - by structurally negating one of the four conditions necessary to cause a deadlock.

THE OSTRICH ALGORITHM

- The simplest approach the ostrich algorithm:
- stick your head in the sand and pretend there is no problem at all.
 - **Mathematicians** find it totally unacceptable and say that dead locks must be prevented at all costs.
 - **Engineers** ask how often the problem is expected, how often the system crashes for other reasons, and how serious a deadlock is.

Deadlock Detection and Recovery

Detection Algorithms

- Deadlock Detection with One Resource of Each Type
- Deadlock Detection with Multiple Resources of Each Type

Recovery from Deadlock

- Recovery through Preemption
- Recovery through Rollback
- Recovery through Killing Processes

Deadlock Detection with One Resource of Each Type

- Assume that only one resource of each type exists.
 - Such a system might have one scanner, one CD recorder, one plotter, and one tape drive.
- We can construct a **resource graph**.
- If this graph contains one or more cycles, a **deadlock exists**.
- Any process that is part of a cycle is deadlocked.
- If no cycles exist, the system is not deadlocked.

Deadlock Detection with One Resource of Each Type...

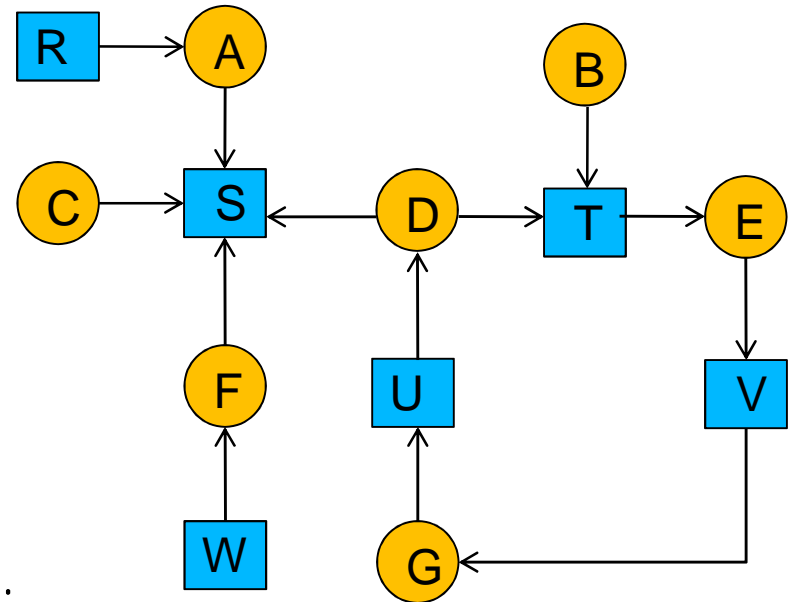
- Maintain wait-for graph
 - Nodes are processes
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph.
- If there is a cycle, there exists a deadlock.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Example

- Consider a system with **seven processes**, A through G, and **six resources**, R through W.
- The state are as follows:
 - Process A holds R and wants S.
 - Process B holds nothing but wants T.
 - Process C holds nothing but wants S.
 - Process D holds U and wants S and T.
 - Process E holds T and wants V.
 - Process F holds W and wants S.
 - Process G holds V and wants U.

Answer

Construct wait for graph



Process **D**, **E**, and **G** are
deadlocked

Question

- "Is this system **deadlocked**, and if so, which processes are involved?"

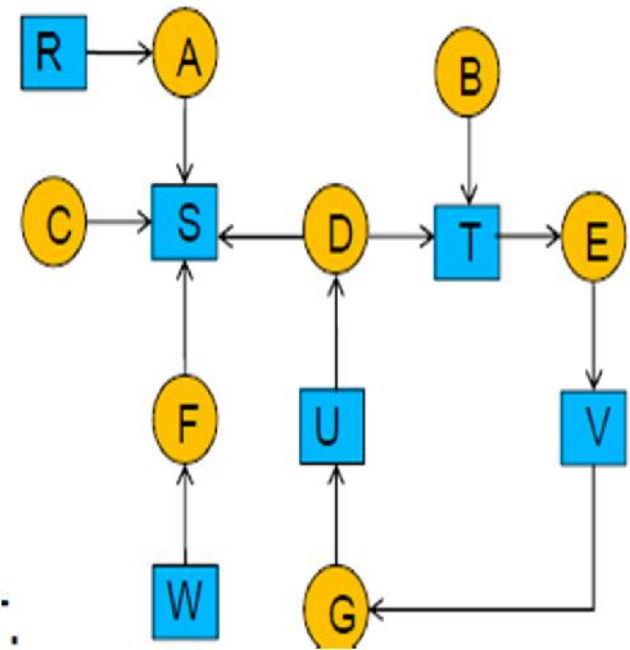
Algorithm to detect deadlock

For each node, N in the graph, perform the following 5 steps with N as the starting node.

1. Initialize L to the empty list, and designate all the arcs as unmarked.
2. Add the current node to the end of L and check to see if the node now appears in L two times. If it does, the graph contains a cycle (listed in L) and the algorithm terminates.
3. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 4; if not, go to step 5.
4. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 2.
5. We have now reached a dead end. Remove it and go back to the previous node, that is, the one that was current just before this one, make that one the current node, and go to step 2.

Algorithm to detect deadlock...

- The order of processing the nodes is arbitrary, so let us just inspect them:
 - from left to right,
 - top to bottom,
- first running the algorithm starting at R, then successively, A, B, C, S, D, T, E, F, and so forth. If we hit a cycle, the algorithm stops.
- We start at R and initialize L to the empty list. Then we add R to the list and move to the only possibility, A, and add it to L, giving $L = [R, A]$.
- From A we go to S, giving $L = [R, A, S]$. S has no outgoing arcs, so it is a dead end, forcing us to backtrack to A.
- Since A has no unmarked outgoing arcs, we backtrack to R, completing our inspection of R.



Restart for A, no cycle
 For B
 $L = [B, T, E, V, G, U, D, T]$
 discovered cycle and stop
 alg. Declare **DEADLOCK**

Deadlock Detection with Multiple Resources of Each Type

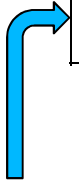
- Matrix-based algorithm for detecting deadlock among n processes, P_1 through P_n .
- Let number of resource classes be m , with E_1 resources of class 1, E_2 resources of class 2, and generally resources of class i ($1 \leq i \leq m$).
- **E is the existing resource vector**. It gives the total number of instances of each resource in existence.
- For example, if class 1 is tape drives, then $E_1 = 2$ means the system has two tape drives.
- Let **A be the available resource vector**, with A_i giving the number of instances of Resource i that are currently available (i.e unassigned).

Deadlock Detection with Multiple Resources...

- Two arrays: C - the **current allocation matrix**, and R - the **request matrix**.
- C_{ij} is the number of instances of resource j that are held by process i.
- R_{ij} is the number of instances of resource j that P_i wants.

Resource in existence
($E_1, E_2, E_3, \dots, E_m$)

Current allocation matrix

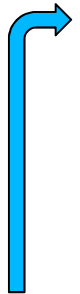


C_{11}	C_{12}	C_{13}	...	C_{1m}
C_{21}	C_{22}	C_{23}	...	C_{2m}
...
C_{n1}	C_{n2}	C_{n3}	...	C_{nm}

Row n is current allocation
to process n

Resource available
($A_1, A_2, A_3, \dots, A_m$)

Request matrix



R_{11}	R_{12}	R_{13}	...	R_{1m}
R_{21}	R_{22}	R_{23}	...	R_{2m}
...
R_{n1}	R_{n2}	R_{n3}	...	R_{nm}

Row 2 is what process 2
needs

Deadlock Detection with Multiple Resources...

- An important invariant holds for these four data structures.
- In particular, every resource either is allocated or is available.
- This observation means that

$$\sum_{i=1}^N C_{ij} + A_j = E_j$$

Deadlock Detection with Multiple Resources...

Algorithm

1. Look for an unmarked process, P_i , for which the i -th row of R is less than or equal to A .
2. If such a process is found, add the i -th row of C to A , mark the process, and go back to step 1.
3. If no such process exists, the algorithm terminates.
4. When the algorithm finishes, all the unmarked processes, if any, are **deadlocked**.

Example

$$E = (4 \ 2 \ 3 \ 1)$$

Tape drives
 Plotters
 Scanners
 CD Roms

$$A = (2 \ 1 \ 0 \ 0)$$

Tape drives
 Plotters
 Scanners
 CD Roms

Current allocation matrix

P_1	0	0	1	0
P_2	2	0	0	1
P_3	0	1	2	0

Request matrix

P_1	2	0	0	1
P_2	1	0	1	0
P_3	2	1	0	0

Process 3 run first and return all its resources: $A = (2 \ 2 \ 2 \ 0)$

Process 2 can run next and return its resources: $A = (4 \ 2 \ 2 \ 1)$

Now process 1 can run. There is no deadlock in the system.

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
- 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

- Suppose that our deadlock detection algorithm has succeeded and detected a deadlock.
- What is next?
- Some way is needed to recover and get the system going again.
- In this section, we will discuss various ways of recovering from deadlock.

Recovery from Deadlock...

- Roll back each deadlocked process to some previously defined **checkpoint**, and restart all process
 - Original deadlock may occur
- Successively **kill** deadlocked processes until deadlock no longer exists
- Successively **preempt** resources until deadlock no longer exists

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose 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?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

Deadlock Prevention

- set of rules ensures that at least one of the four necessary conditions for deadlock doesn't hold
 - mutual exclusion
 - hold and wait
 - no preemption
 - circular wait
- may result in low resource utilization, reduced system throughput

Deadlock Prevention...

1. Prevent the **circular-wait condition** by defining a linear ordering of resource types
 - A process can be assigned resources only according to the linear ordering (e.g., **sequence number**)
 - **Disadvantages**
 - Resources cannot be requested in the order that are needed
 - Resources will be longer than necessary
2. Prevent the **hold-and-wait condition** by requiring the process to acquire all needed resources before starting execution
 - **Disadvantages**
 - Inefficient use of resources
 - Reduced concurrency
 - Process can become deadlocked during the initial resource acquisition
 - Future needs of a process cannot be always predicted

Deadlock Prevention...

3. Denying No Preemption

- means that processes may be preempted by the OS
 - should only be done when necessary
 - resources of a process trying to acquire another unavailable resource may be preempted
 - preempt resources of processes waiting for additional resources, and give some to the requesting process
- possible only for some types of resources
 - state must be easily restorable
 - e.g. CPU, memory

Deadlock Prevention ...

1.e Use of time-stamps

- Example: Use time-stamps for transactions to a database – each transaction has the time-stamp of its creation
- The circular wait condition is avoided by comparing time-stamps: strict ordering of transactions is obtained, the transaction with an earlier time-stamp always wins
 - “**Wait-die**” method

```
if [ e (T2) < e (T1) ]
    halt_T2 ('wait');
else
    kill_T2 ('die');
```
 - “**Wound-wait**” method

```
if [ e (T2) < e (T1) ]
    kill_T1 ('wound');
else
    halt_T2 ('wait');
```

Timestamped Deadlock-Prevention Scheme

- Each process P_i is assigned a unique timestamp
- Timestamps are used to decide whether a process P_i should wait for a process P_j ; otherwise P_i is rolled back.
- The scheme prevents deadlocks.
- For every edge $P_i \rightarrow P_j$ in the wait-for graph, P_i has a higher priority (lower timestamp) than P_j .
- Thus a cycle cannot exist.

Wait-Die Scheme

- Based on a **nonpreemptive** technique.
- If P_i requests a resource currently held by P_j , P_i is allowed to wait only if it has a smaller timestamp than does P_j (P_i is older than P_j).
- Otherwise, P_i is rolled back (dies).
- **Example:** Suppose that processes P_1 , P_2 , and P_3 have timestamps 5, 10, and 15 respectively.
 - if P_1 request a resource held by P_2 , then P_1 will wait.
 - If P_3 requests a resource held by P_2 , then P_3 will be rolled back (dies).

Wound-Wait Scheme

- Based on a **preemptive** technique; counterpart to the wait-die system.
- If P_i requests a resource currently held by P_j , P_i is allowed to wait only if it has a larger timestamp than does P_j (P_i is younger than P_j).
- Otherwise P_j is rolled back (P_j is wounded by P_i).
- **Example:** Suppose that processes P_1 , P_2 , and P_3 have timestamps 5, 10, and 15 respectively.
 - If P_1 requests a resource held by P_2 , then the resource will be preempted from P_2 and P_2 will be rolled back.
 - If P_3 requests a resource held by P_2 , then P_3 will wait.

Deadlock Avoidance

- Basic Principle: Requires that the system has some additional a priori information available
- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need to hold simultaneously. (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

Deadlock Avoidance...

- The system must be able to decide whether granting a resource is safe or not and only **make the allocation when it is safe.**
- Thus, the question arises: Is there an algorithm that can always avoid deadlock by making the right choice all the time?
- The answer is a qualified yes-we can avoid deadlocks.
- Algorithms
 - **Safe and Unsafe States**
 - **The Bankers algorithm**

Safe state

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a **safe state**
- A state is **safe** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- We are considering a worst-case situation here.
- Even in the worst case (process requests up their maximum at the moment), we don't have deadlock in a safe state.

Safe state...

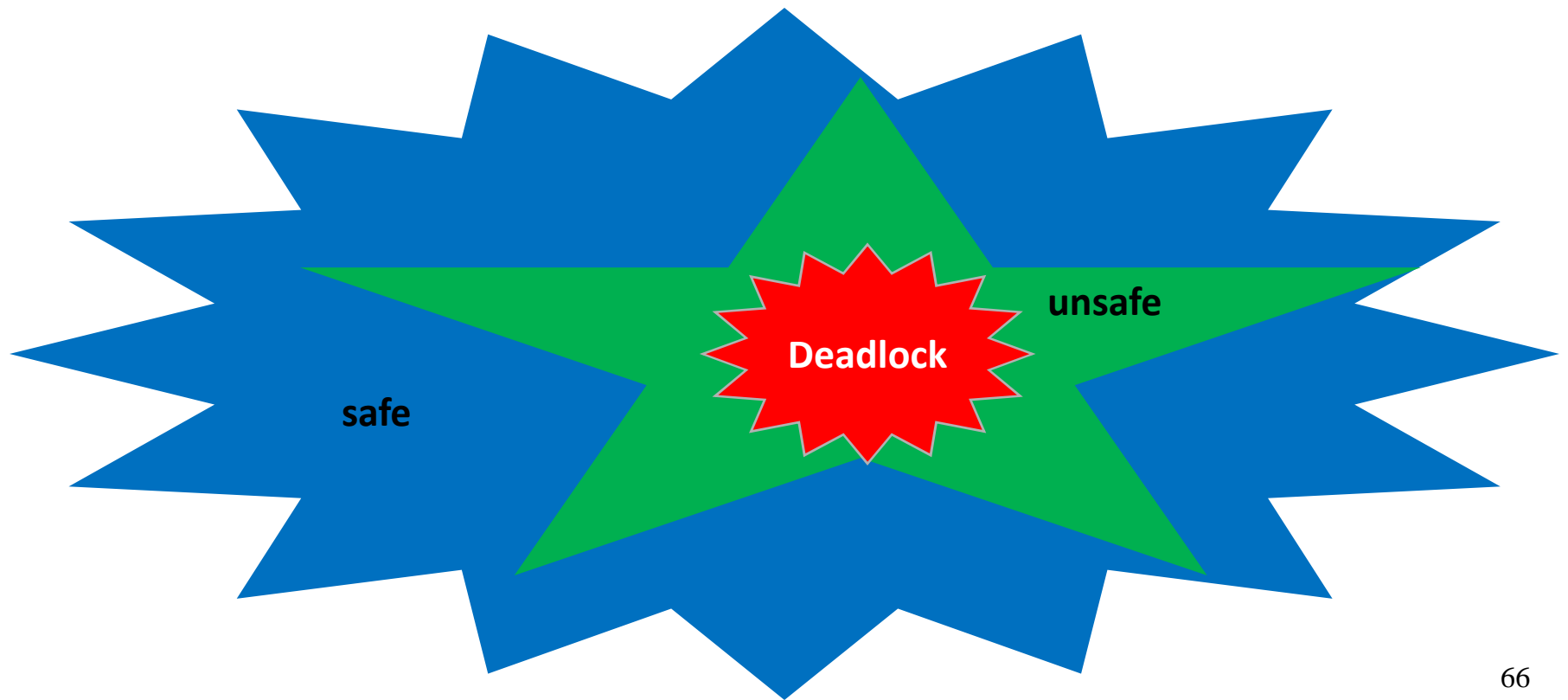
- More formally: A system state is safe if there exists a safe sequence of all processes $\langle P_1, P_2, \dots, P_n \rangle$
- such that
 - for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$
- That is:
 - 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.

Basic Facts

- If a system is in safe state \Rightarrow **no deadlocks**
- If a system is in unsafe state \Rightarrow **possibility of deadlock**
- Avoidance \Rightarrow ensure that a system will never enter an unsafe state.
 - When a request is done by a process for some resource(s):
 - check before allocating resource(s);
 - if it will leave the system in an unsafe state, then do not allocate the resource(s);
 - process is waited and resources are not allocated to that process.

Safe State Space

- if a system is in a safe state there are no deadlocks
- in an unsafe state, there is a possibility of deadlocks

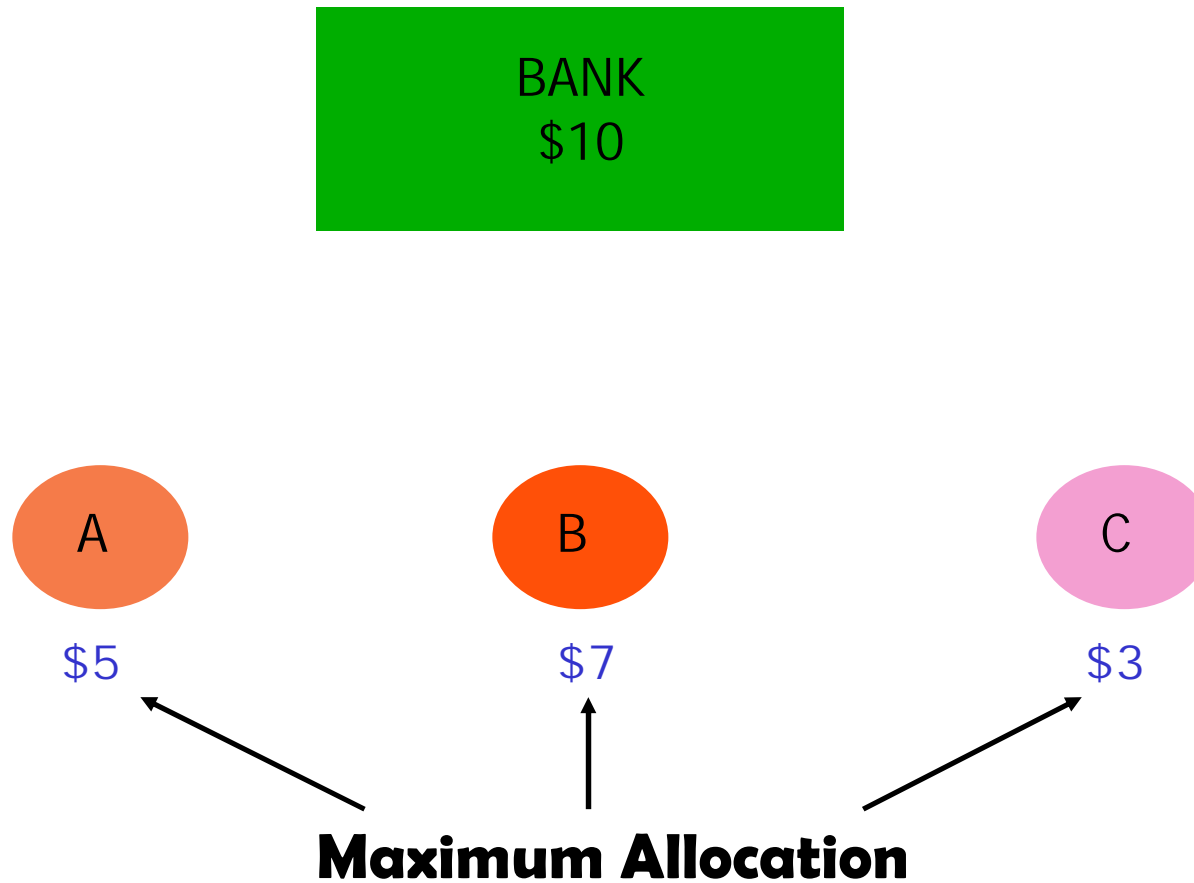


Deadlock Avoidance Algorithms

- Single instance of a resource type
 - Use a resource-allocation graph
- Multiple instances of a resource type
 - Use the banker's algorithm

Example

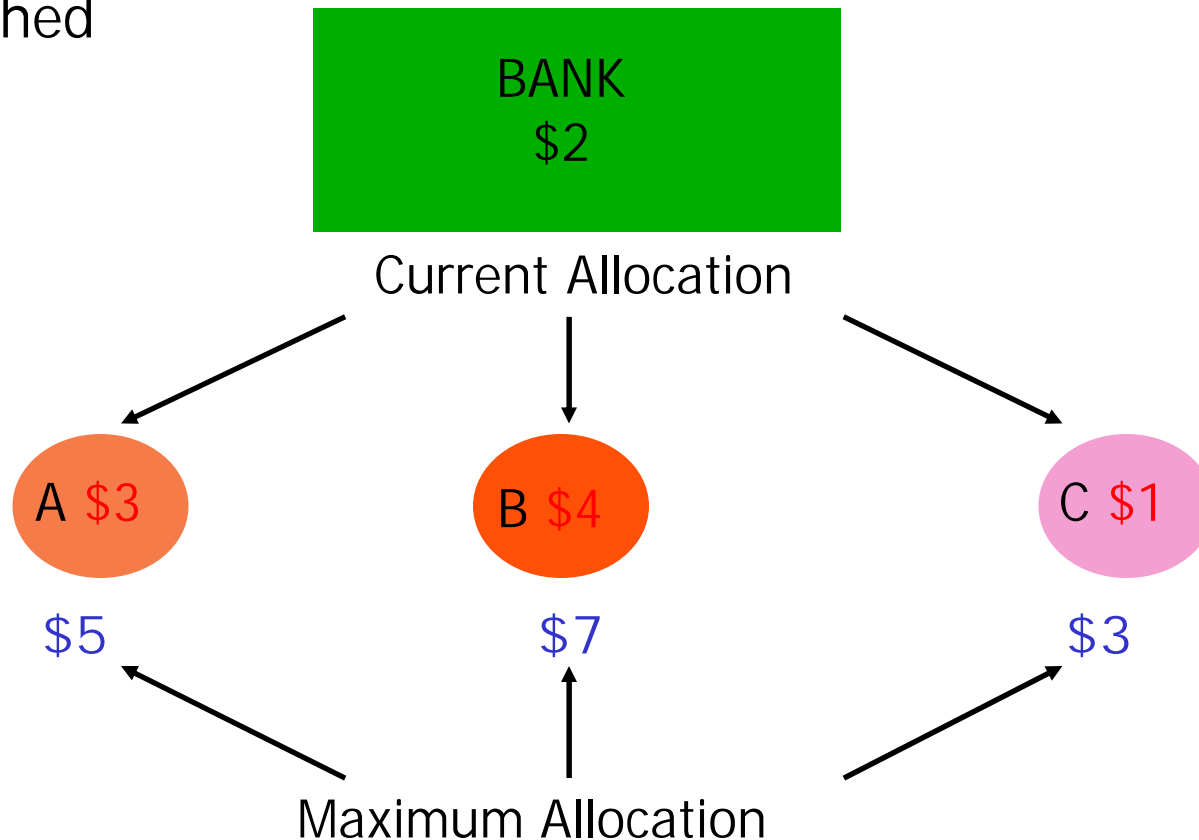
- Bank gives loans to customers
 - maximum allocation = credit limit



- Safe State?

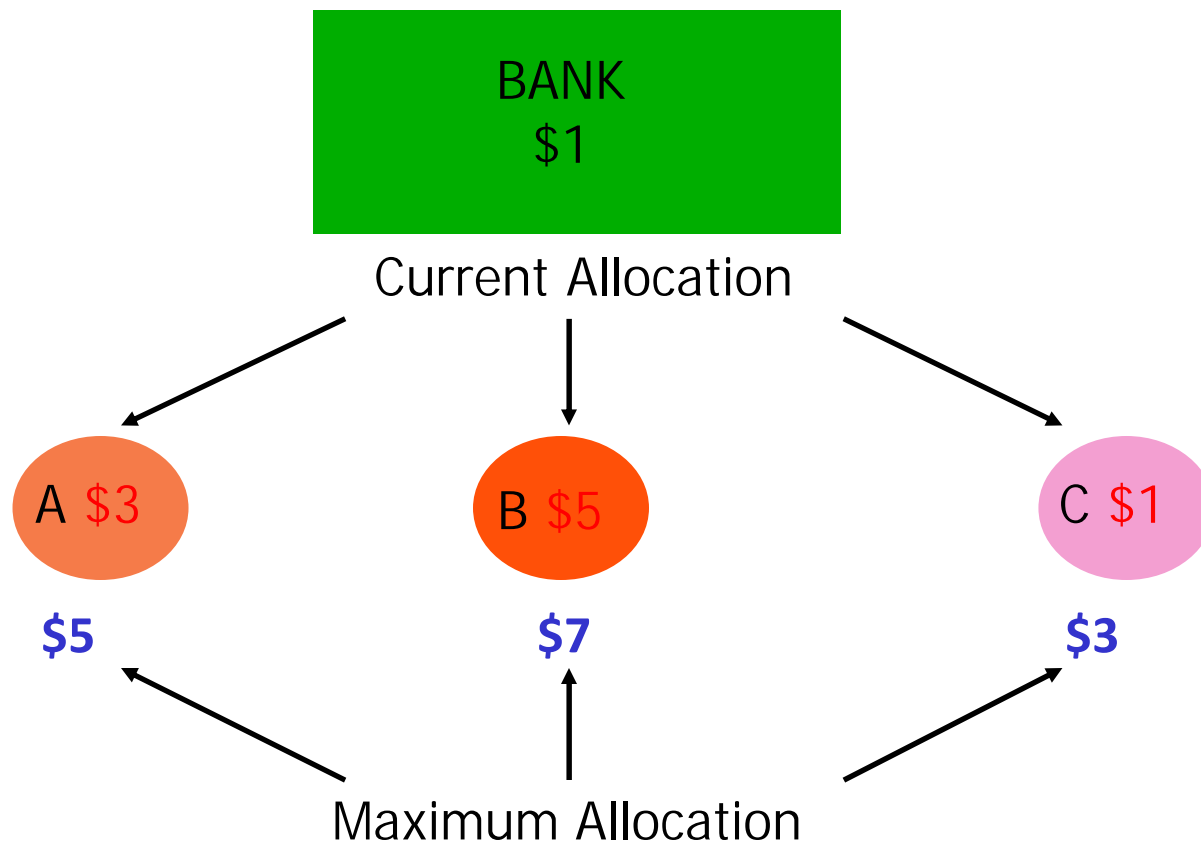
- Will the bank be able to give each customer a loan up to the full credit limit?

- not necessarily all customers simultaneously
- order is not important
- customers will pay back their loan once their credit limit is reached



- Still Safe?

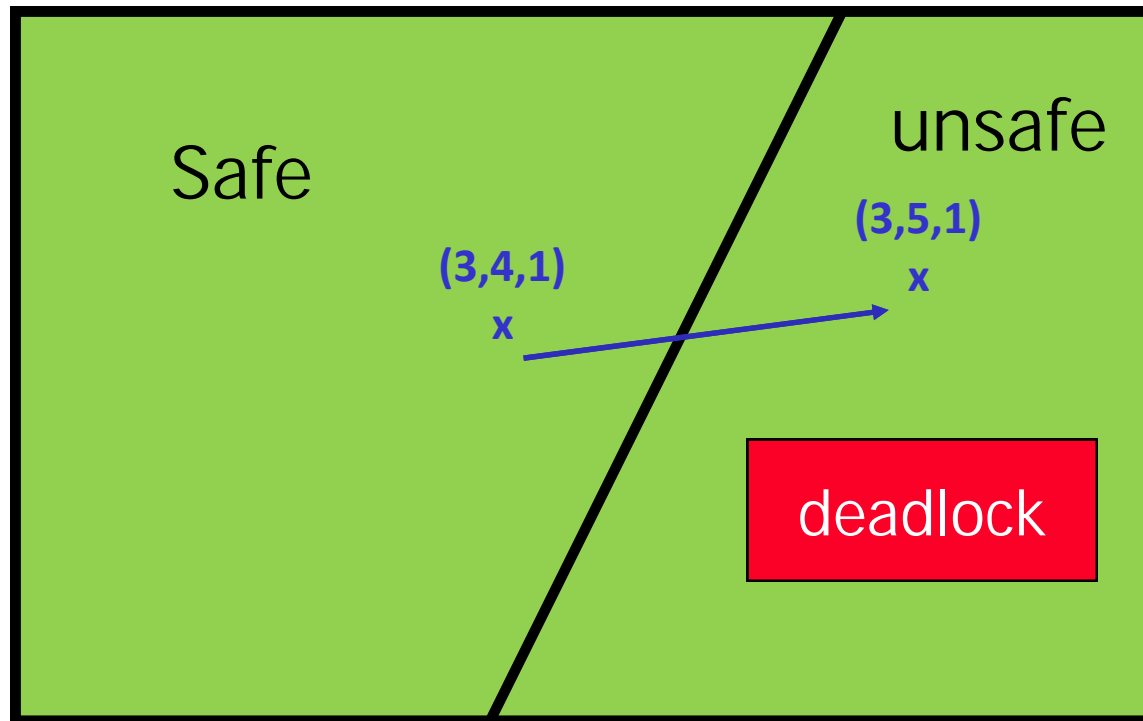
- after customer B requests and is granted \$1, is the bank still safe? **NO**



Safe State Space



Bank Safe State Space



Safe and Unsafe States: Example

- A total of 10 instances of the resource exist, so with 7 resources already allocated, there are 3 still free. Is the state safe or not?

	Has	Max
A	3	9
B	2	4
C	2	7
Free = 3		

	Has	Max
A	3	9
B	4	4
C	2	7
Free = 1		

	Has	Max
A	3	9
B	0	--
C	2	7
Free = 5		

	Has	Max
A	3	9
B	0	--
C	7	7
Free = 0		

	Has	Max
A	3	9
B	0	--
C	0	--
Free = 7		

- Scheduler can run B first, then C and finally A.
- Thus, the state is safe because the system, by careful scheduling can avoid deadlock.

	Has	Max
A	3	9
B	2	4
C	2	7
Free = 3		

	Has	Max
A	4	9
B	2	4
C	2	7
Free = 2		

	Has	Max
A	4	9
B	4	4
C	2	7
Free = 0		

	Has	Max
A	4	9
B	--	--
C	2	7
Free = 4		

unsafe

The Banker's Algorithm

- before a request is granted, check the system's state
 - assume the request is granted
 - if it is still safe, the request can be honored
 - otherwise the process has to wait
 - overly careful
 - there are cases when the system is unsafe, but not in a deadlock

The Banker's Algorithm: Single resource

- What the algorithm does is check to see if granting the request leads to an unsafe state. If it does, the request is denied.
- If granting the request leads to a safe state, it is carried out.
- The banker reserved 10 instead of 22.

SAFE

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7
Free = 10		

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7
Free = 2		

	Has	Max
A	1	6
B	1	5
C	4	4
D	4	7
Free = 0		

	Has	Max
A	1	6
B	1	5
C	--	--
D	4	7
Free = 4		

	Has	Max
A	1	6
B	5	5
C	--	--
D	4	7
Free = 0		

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7
Free = 10		

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7
Free = 2		

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7
Free = 1		

UNSAFE

Banker's Algorithm: Multiple resource

- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

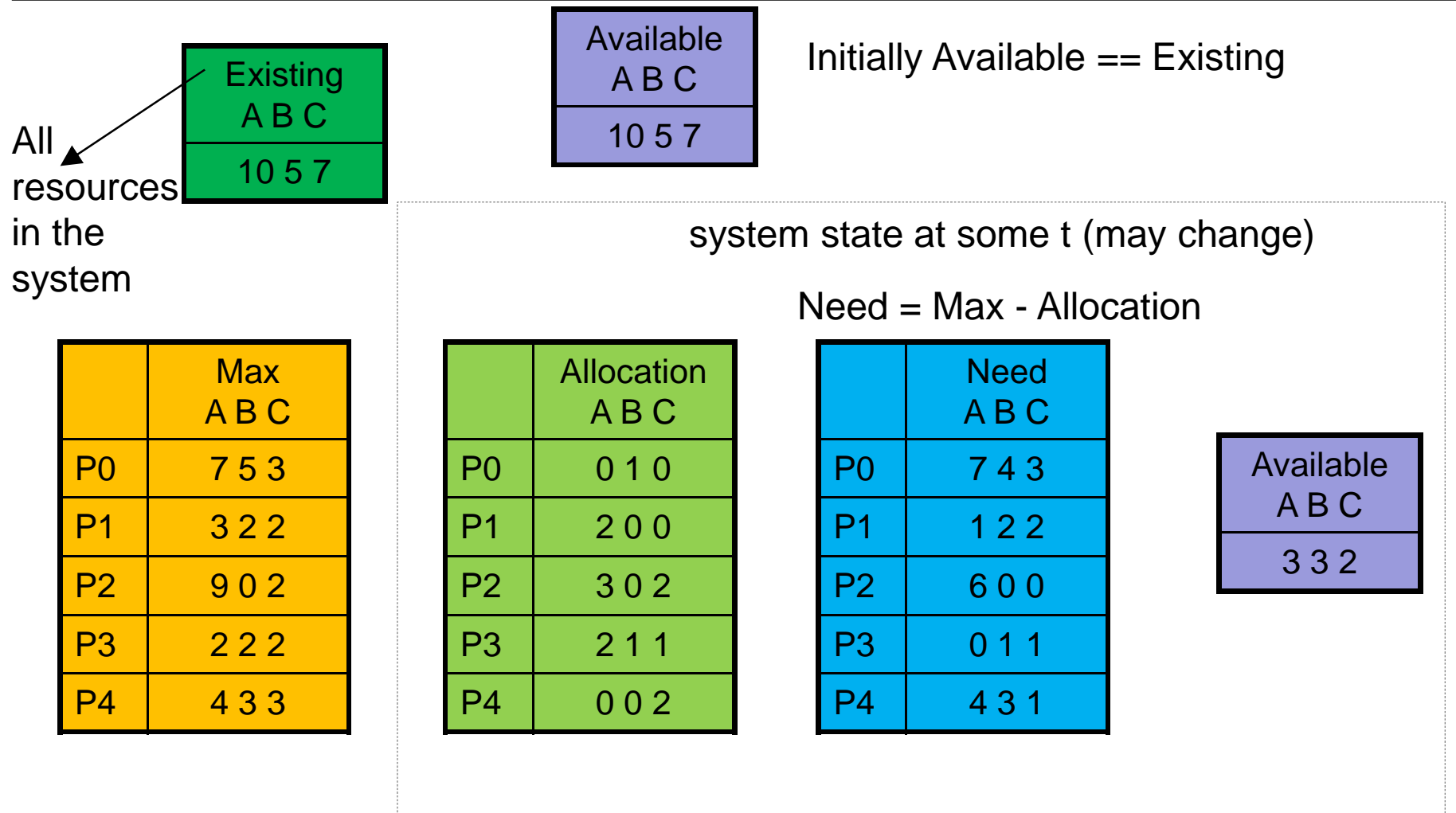
Data Structures for the Banker's Algorithm

Let n = number of processes, and
 m = number of resources types.

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

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$

An example system state



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.

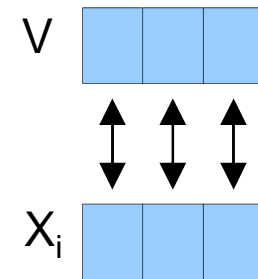
X_i is the i^{th} row of the matrix: it is a vector.

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



$V == X_i ?$

$V <= X_i ?$

$X_i <= V ?$

....

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

$[2\ 2\ 1] <= [3\ 3\ 2]$

Safety Algorithm

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

Work = Available (initialize Work temporary vector)

Finish [i] = false for $i = 0, 1, \dots, n-1$

(Work is a temporary vector initialized to the Available (i.e., free) resources at that time when the safety check is performed)

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

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

Available
[3 3 2]

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

Resource-Request Algorithm for Process P_i

Request : request vector for process P_i .

If **Request** _{i} [j] == k , then process P_i wants k instances of resource type R_j

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 requested resources to P_i by modifying the state as follows:

$$\text{Available} = \text{Available} - \text{Request}_i ;$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i ;$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i ;$$

Run the Safety Check Algorithm:

- If safe \emptyset the requested resources are allocated to P_i
- If unsafe \emptyset The requested resources are not allocated to P_i .
 P_i must wait.

The old resource-allocation state is restored.

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 Request \leq Available (that is, (1,0,2) \leq (3,3,2) \Rightarrow true.
- Then check the new state for safety:

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

new state

Can we find a sequence?

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]

Sequence is:

P1, P3, P4, P0, P2

Yes, New State is safe.

We can grant the request.

Allocate 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 P_4 requests (3 3 0)?

There is no available resource to satisfy the request. P_4 will be waited.

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

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

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 P_0 . The new state will be as follows.

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

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 would be **UNSAFE**.
Hence we should not go to the new state.
The request is not granted. **P0 is waited.**

Combined Approach to Deadlock Handling

- Combine the three basic approaches
 - prevention
 - avoidance
 - detectionallowing 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.

Project

- Solve the dining Philosophers problem using any method. Research the methods. You can program and implement using any programming language like C, C++, Java,...