

# Deadlocks

## Week 13 - Lecture

# Team

- Instructor
  - Giancarlo Succi
- Teaching Assistants
  - Nikita Lozhnikov (also Tutorial Instructor)
  - Manuel Rodriguez
  - Shokhista Ergasheva

# Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bos, Modern Operating Systems: 4th edition, 2013, Prentice-Hall, Inc and customized for the needs of this course
- Additional input for the slides are detailed later

# Resources (1)

- An object granted is called a **resource**
- A resource can be a hardware device (e.g., a Blu-ray drive) or a piece of information (e.g., a record in a database)

## Resources (2)

- A **preemptable resource** is one that can be taken away from the process owning it with no ill effects
- A **nonpreemptable resource** is one that cannot be taken away from its current owner without potentially causing failure

# Resources (3)

- The abstract sequence of events required to use a resource:
  - Request the resource
  - Use the resource
  - Release the resource

# Resource Acquisition (1)

- For some kinds of resources, such as records in a database system, it is up to the user processes rather than the system to manage resource usage themselves
- One way of allowing this is to associate a semaphore with each resource

# Resource Acquisition (2)

- The three steps listed as follows, are implemented as a down on the semaphore to acquire the resource, the use of the resource, and finally an up on the resource to release it
- These steps are shown in Fig. 6-1(a)



# Resource Acquisition (3)

```
typedef int semaphore;  
semaphore resource_1;  
  
void process_A(void) {  
    down(&resource_1);  
    use_resource_1( );  
    up(&resource_1);  
}
```

Figure 6-1. Using a semaphore to protect resources  
(a) One resource

# Resource Acquisition (4)

```
typedef int semaphore;  
semaphore resource_1;  
semaphore resource_2;  
  
void process_A(void) {  
    down(&resource_1);  
    down(&resource_2);  
    use_both_resources( );  
    up(&resource_2);  
    up(&resource_1);  
}
```

Figure 6-1. Using a semaphore to protect resources.  
Two resources

# Resource Acquisition (5)

- Consider a situation with two processes, A and B, and two resources:
  - One of the processes will acquire the first resource before the other one
  - That process will then successfully acquire the second resource and do its work
  - If the other process attempts to acquire the first resource before it has been released, the other process will simply block until it becomes available

# Resource Acquisition (6)

- The situation might be different. It might happen that one of the processes acquires both resources and effectively blocks out the other process until it is done
- However, it might also happen that process A acquires resource 1 and process B acquires resource 2. Each one will now block when trying to acquire the other one
- Neither process will ever run again. Bad news: this situation is a deadlock

# Resource Acquisition (7)

```
typedef int semaphore;
```

```
    semaphore resource_1;  
    semaphore resource_2;
```

```
void process_A(void) {  
    down(&resource_1);  
    down(&resource_2);  
    use_both_resources( );  
    up(&resource_2);  
    up(&resource_1);  
}
```

```
void process_B(void) {  
    down(&resource_1);  
    down(&resource_2);  
    use_both_resources( );  
    up(&resource_2);  
    up(&resource_1);  
}
```

```
    semaphore resource_1;  
    semaphore resource_2;
```

```
void process_A(void) {  
    down(&resource_1);  
    down(&resource_2);  
    use_both_resources( );  
    up(&resource_2);  
    up(&resource_1);  
}
```

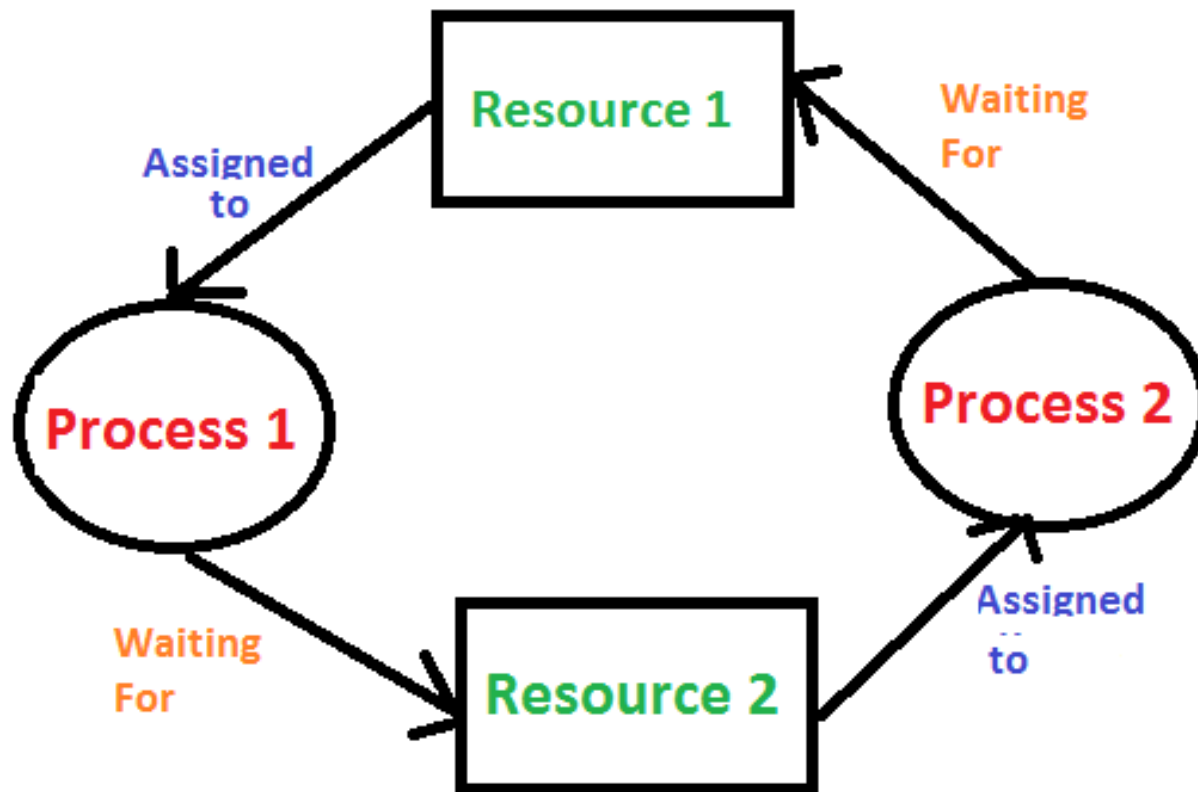
```
void process_B(void) {  
    down(&resource_2);  
    down(&resource_1);  
    use_both_resources( );  
    up(&resource_1);  
    up(&resource_2);  
}
```

Figure 6-2. Deadlock-free code (left)  
Code with a potential deadlock (right)

# Introduction to Deadlocks (1)

- A set of processes is deadlocked if:
  - Each process in the set is waiting for an event
  - That event can only be caused by another process in this set

# Introduction to Deadlocks (2)



# Deadlock (1)

- Example:
  - Processes A and B want to record a scanned document on a Blu-ray disc
  - Process A requests permission to use the scanner and is granted it
  - Process B is programmed differently and requests the Blu-ray recorder first and is also granted it



# Deadlock (2)

- Example (cont.):
  - A asks for the Blu-ray recorder, but the request is suspended until B releases it
  - Instead of releasing the recorder, B asks for the scanner
  - At this point both processes are blocked and will remain so forever
- This situation is called a deadlock

# Conditions for Resource Deadlocks (1)

- Four conditions that must hold for there to be a (resource) deadlock:
  - Mutual exclusion condition
  - Hold and wait condition
  - No preemption condition
  - Circular wait condition

# Conditions for Resource Deadlocks (2)

- Mutual exclusion condition:  
Each resource is either currently assigned to exactly one process or is available
- Hold-and-wait condition:  
Processes currently holding resources that were granted earlier can request new resources

# Conditions for Resource Deadlocks (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
- Circular wait condition:  
There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain

# Deadlock Modeling (1)

- Deadlock conditions can be modeled using directed graphs
- The graphs have two kinds of nodes: processes, shown as circles, and resources, shown as squares
- A directed 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

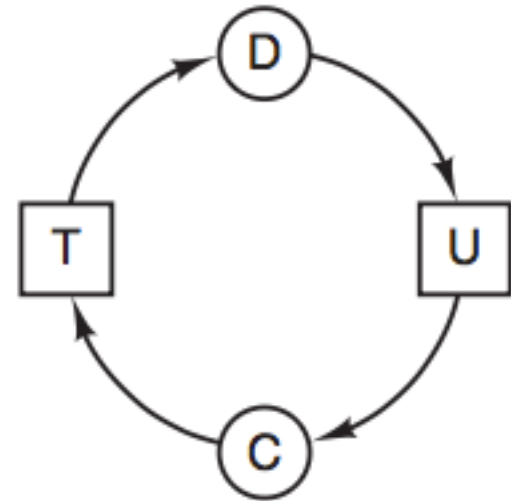
# Deadlock Modeling (2)



Holding  
a resource



Requesting  
a resource



Deadlock

Figure 6-3. Resource allocation graphs

# Deadlock Modeling (3)

- Imagine that we have three processes, A, B, and C, and three resources, R, S, and T. The following order of the requests and releases of the three processes does not lead to any deadlocks:

A  
Request R  
Request S  
Release R  
Release S

B  
Request S  
Request T  
Release S  
Release T

C  
Request T  
Request R  
Release T  
Release R

# Deadlock Modeling (4)

- Let us now 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 following order:

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 (5)

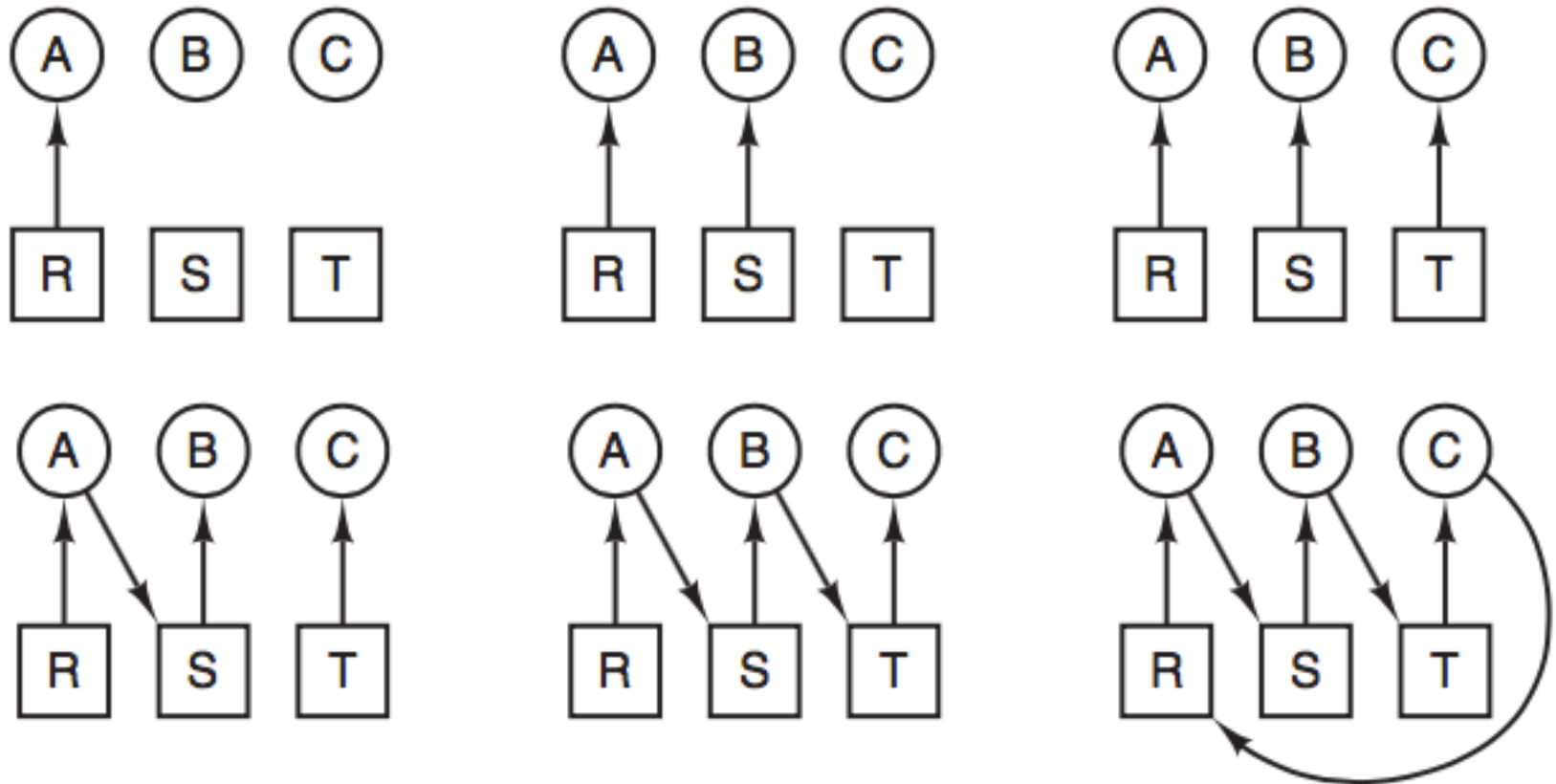


Figure 6-4. An example of how deadlock occurs

# Deadlock Modeling (6)

- The operating system is not required to run the processes in any special order. In particular, if granting a 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

# Deadlock Modeling (7)

- 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 sequence of requests and releases which do not lead to deadlock which is shown on the next slide

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

# Deadlock Modeling (8)

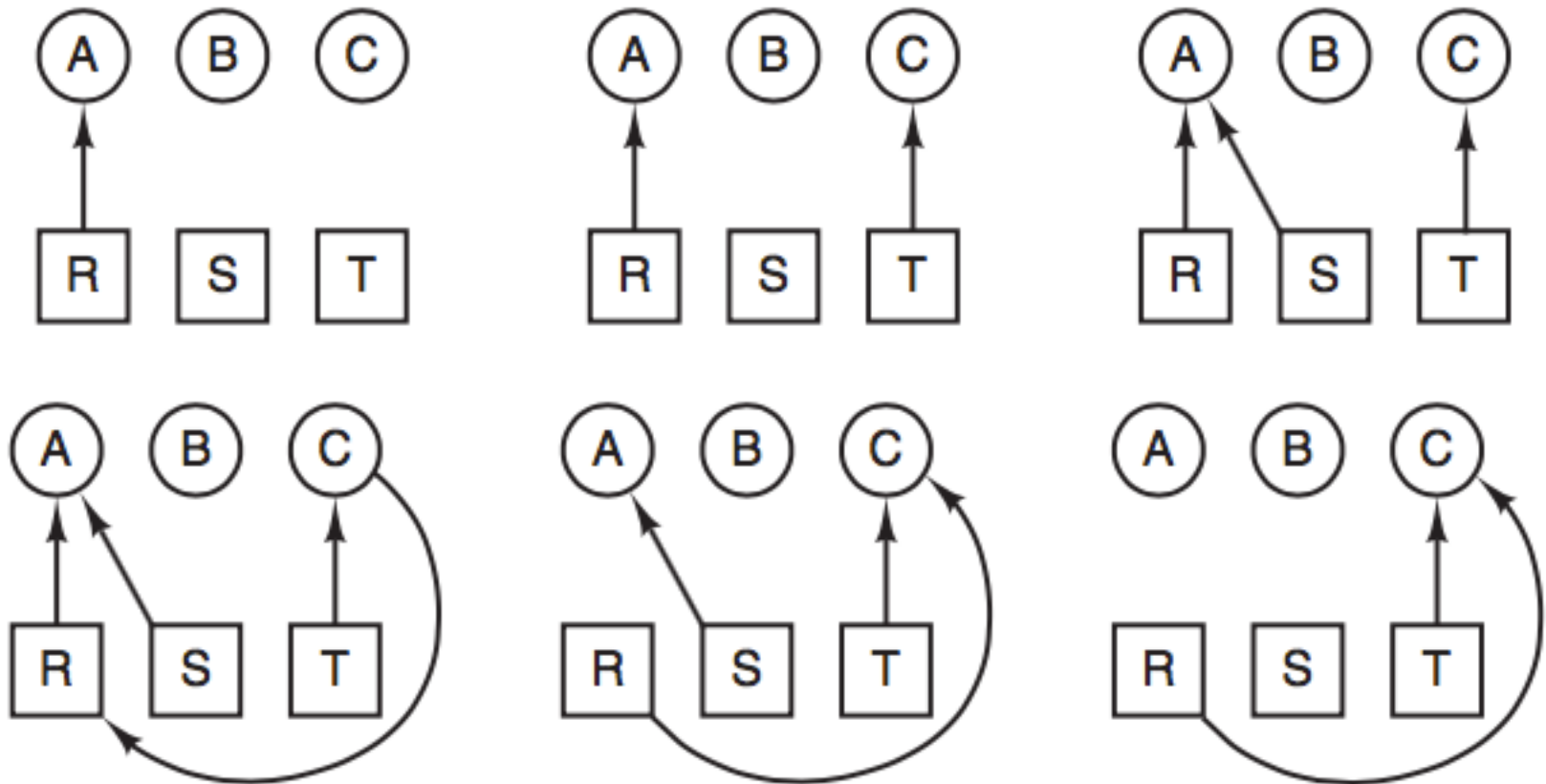


Figure 6-4. An example of how deadlock can be avoided

# Dealing with Deadlocks

- Strategies are used for dealing with deadlocks:
  - Ignore the problem, maybe it will go away
  - Detection and recovery. Let deadlocks occur, detect them, and take action
  - Dynamic avoidance by careful resource allocation
  - Prevention, by structurally negating one of the four required conditions

# The Ostrich Algorithm

- The simplest approach is the ostrich algorithm:
  - Stick your head in the sand and pretend there is no problem

# Deadlock Detection with One Resource of Each Type (1)

- Consider a system with seven processes, A through G, and six resources, R through W:
  - Process A holds R, wants S
  - Process B holds nothing, wants T
  - Process C holds nothing, wants S
  - Process D holds U, wants S and T
  - Process E holds T, wants V
  - Process F holds W, wants S
  - Process G holds V, wants U

# Deadlock Detection with One Resource of Each Type (2)

- The question is: is this system deadlocked, and if so, which processes are involved?
- Let's construct a resource graph



# Deadlock Detection with One Resource of Each Type (3)

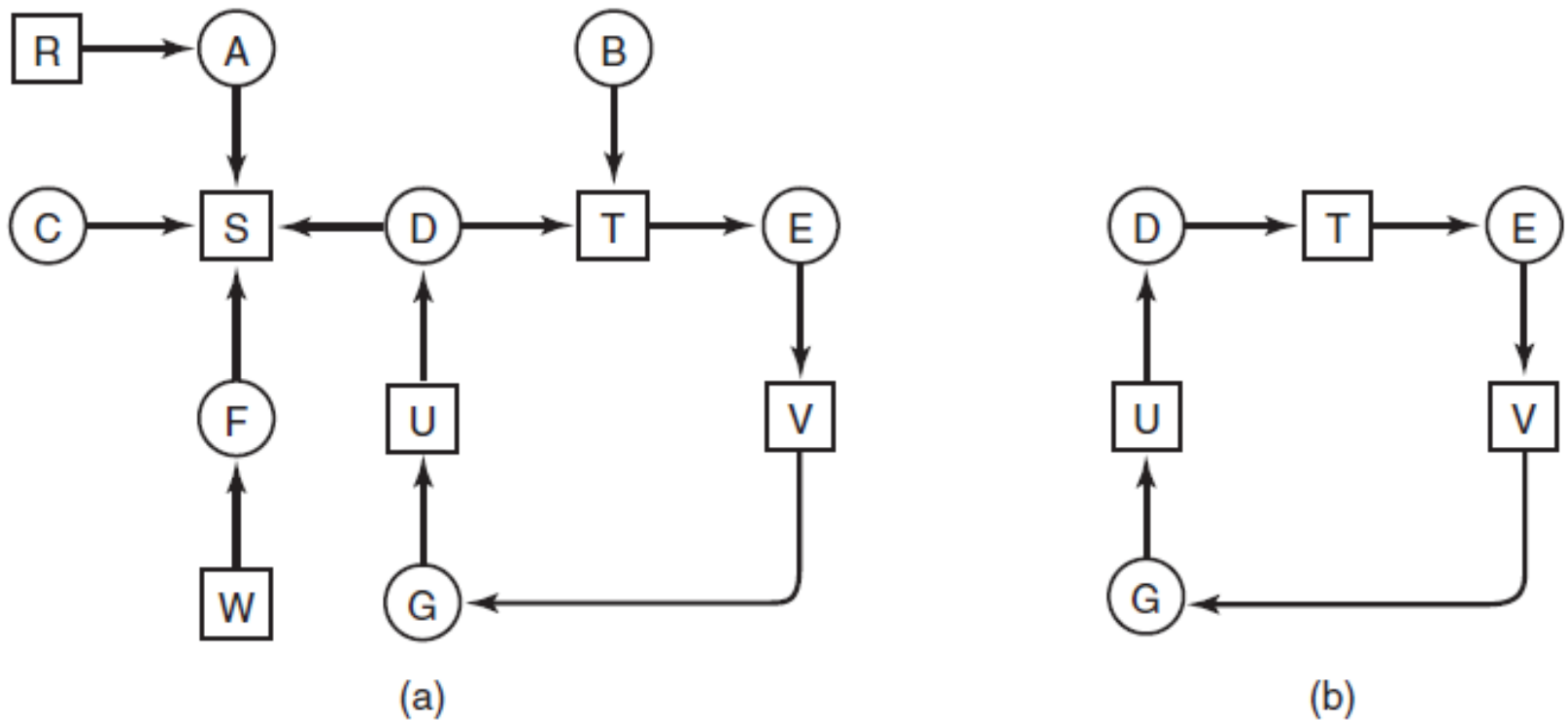


Figure 6-5. (a) A resource graph  
(b) A cycle extracted from (a)

# Deadlock Detection with One Resource of Each Type (4)

1. For each node, N in the graph, perform following five steps with N as starting node
2. Initialize L to an empty list and designate all arcs as unmarked
3. Add current node to end of L, check to see if node now appears in L two times. If so, graph contains a cycle (listed in L) and algorithm terminates

# Deadlock Detection with One Resource of Each Type (5)

4. From given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6
5. Pick unmarked outgoing arc at random, mark it. Then follow to new current node and go to step 3
6. If this is initial node, graph does not contain cycles, algorithm terminates. Otherwise, dead end. Remove it and go back to the previous node

# Deadlock Detection with Multiple Resources of Each Type (1)

- A matrix-based algorithm for detecting deadlock among  $n$  processes,  $P_1$  through  $P_n$  is presented
- Let the number of resource classes be  $m$ , with  $E_1$  resources of class 1,  $E_2$  resources of class 2, and generally,  $E_i$  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

# Deadlock Detection with Multiple Resources of Each Type (2)


- 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).
- Now we need two arrays,  $C$ , the current allocation matrix, and  $R$ , the request matrix. The  $i_{th}$  row of  $C$  tells how many instances of each resource class  $P_i$  currently holds. Thus,  $C_{ij}$  is the number of instances of resource  $j$  that are held by process  $i$ . Similarly,  $R_{ij}$  is the number of instances of resource  $j$  that  $P_i$  wants.
- At any instant, some of the resources are assigned and some are not. These four data structures are shown in Fig. 6-6.

# Deadlock Detection with Multiple Resources of Each Type (3)

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

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


Current allocation matrix



$C_{11}$	$C_{12}$	$C_{13}$	$\dots$	$C_{1m}$
$C_{21}$	$C_{22}$	$C_{23}$	$\dots$	$C_{2m}$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
$C_{n1}$	$C_{n2}$	$C_{n3}$	$\dots$	$C_{nm}$

Row n is current allocation  
to process n

Request matrix



$R_{11}$	$R_{12}$	$R_{13}$	$\dots$	$R_{1m}$
$R_{21}$	$R_{22}$	$R_{23}$	$\dots$	$R_{2m}$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
$R_{n1}$	$R_{n2}$	$R_{n3}$	$\dots$	$R_{nm}$

Row 2 is what process 2 needs

Figure 6-6. The four data structures needed  
by the deadlock detection algorithm

# Deadlock Detection with Multiple Resources of Each Type (4)

- Deadlock detection algorithm:
  - Look for unmarked process  $P_i$ , for which the  $i_{th}$  row of  $R$  is less than or equal to  $A$
  - If such a process is found, add the  $i_{th}$  row of  $C$  to  $A$ , mark the process, go back to step 1
  - If no such process exists, algorithm terminates

# Deadlock Detection with Multiple Resources of Each Type (5)

- Example (Fig. 6-7):
  - We have 3 processes and 4 resource classes, which we have arbitrarily labeled tape drives, plotters, scanners, and Blu-ray drives
  - Process 1 has one scanner
  - Process 2 has two tape drives and a Blu-ray drive
  - Process 3 has a plotter and two scanners. Each process needs additional resources, as shown by the R matrix



# Deadlock Detection with Multiple Resources of Each Type (6)

	Tape drives	Plotters	Scanners	CD Roms
$E =$	4	2	3	1

	Tape drives	Plotters	Scanners	CD Roms
$A =$	2	1	0	0

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Figure 6-7. An example for the deadlock detection algorithm

# Deadlock Detection with Multiple Resources of Each Type (7)

- To run the deadlock detection algorithm, we look for a process whose resource request can be satisfied:
  - The first one cannot be satisfied because there is no Blu-ray drive available
  - The second cannot be satisfied either, because there is no scanner free
  - The third one can be satisfied, so process 3 runs and eventually returns all its resources, giving  $A = (2\ 2\ 2\ 0)$
  - At this point process 2 can run and return its resources, giving  $A = (4\ 2\ 2\ 1)$
  - Remaining process can run. There is no deadlock in the system

# Recovery from Deadlock

- Possible methods of recovery
  - Preemption
  - Rollback
  - Killing processes

# Recovery through Preemption

- In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process
- The ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource

# Recovery through Rollback (1)

- Recovery through rollback involves checkpointing:
  - Checkpointing a process means that its state is written to a file so that it can be restarted later

# Recovery through Rollback (2)

- To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired that resource by starting at one of its earlier checkpoints
- In effect, the process is reset to an earlier moment when it did not have the resource, which is now assigned to one of the deadlocked processes
- If the restarted process tries to acquire the resource again, it will have to wait until it becomes available

# Recovery through Killing Processes

- The crudest but simplest way to break a deadlock is to kill one or more processes
- The process to be killed is carefully chosen because it is holding resources that some process in the cycle needs
- Where possible, it is best to kill a process that can be rerun from the beginning with no ill effects

# Deadlock Avoidance (1)

- Deadlocks can be avoided, but only if certain information is available in advance. Such as:
  - The system must be able to decide whether granting a resource is safe or not and make the allocation only when it is safe



# Deadlock Avoidance (2)

- In Fig. 6-8 we see a model for dealing with two processes and two resources, for example, a printer and a plotter:
  - The horizontal axis represents the number of instructions executed by process A
  - The vertical axis represents the number of instructions executed by process B

Printer

Plotter

Printer

Plotter

● u (Both processes finished)

Giancarlo Succi. Operating Systems. Innopolis University. Fall 2019.

# Deadlock Avoidance (4)

- At  $l_1$ , A requests a printer; at  $l_2$  it needs a plotter
- The printer and plotter are released at  $l_3$  and  $l_4$ , respectively
- Process B needs the plotter from  $l_5$  to  $l_7$  and the printer from  $l_6$  to  $l_8$
- If the scheduler chooses to run A first, we get to the point  $q$ , in which A has executed some number of instructions, but B has executed none
- At point  $q$  the trajectory becomes vertical, indicating that the scheduler has chosen to run B

# Deadlock Avoidance (5)

- When A crosses the  $l_1$  line on the path from  $r$  to  $s$ , it requests and is granted the printer. When B reaches point  $t$ , it requests the plotter
- If the system ever enters the box bounded by  $l_1$  and  $l_2$  on the sides and  $l_5$  and  $l_6$  top and bottom, it will eventually deadlock when it gets to the intersection of  $l_2$  and  $l_6$
- At this point, A is requesting the plotter and B is requesting the printer, and both are already assigned. The entire box is unsafe and must not be entered

# Deadlock Avoidance (6)

- At point  $t$  the only safe thing to do is run process A until it gets to  $l_4$
- At point  $t$ , B is requesting a resource. The system must decide whether to grant it or not. To avoid the deadlock, B should be suspended until A has requested and released the plotter.

# Safe and Unsafe States (1)

- A state is said to be safe if there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately

# Safe and Unsafe States (2)

- Consider the following situation (Fig. 6-9a):
  - We have a state in which A has three instances of the resource but may need as many as nine eventually
  - B currently has two and may need four altogether, later
  - Similarly, C also has two but may need an additional five
  - A total of 10 instances of the resource exist, so with seven resources already allocated, three there are still free
- The state is safe because there exists a sequence of allocations that allows all processes to complete

# Safe and Unsafe States (3)

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

Figure 6-9. The initial state



# Safe and Unsafe States (4)

- Namely, the scheduler can simply run B exclusively, until it asks for and gets two more instances of the resource, leading to the state of Fig. 6-9b
- When B completes, we get the state of Fig. 6-9c
- Then the scheduler can run C, leading eventually to Fig. 6-9d. When C completes, we get Fig. 6-9e
- Now A can get the six instances of the resource it needs and also complete
- Thus, the state of Fig. 6-9a is safe because the system, by careful scheduling, can avoid deadlock

# Safe and Unsafe States (5)

Has Max			Has Max			Has Max		
A	3	9	A	3	9	A	3	9
B	2	4	B	4	4	B	0	—
C	2	7	C	2	7	C	2	7
Free: 3			Free: 1			Free: 5		
(a)			(b)			(c)		

Has Max			Has Max		
A	3	9	A	3	9
B	0	—	B	0	—
C	7	7	C	0	—
Free: 0			Free: 7		
(d)			(e)		

Figure 6-9. Demonstration that the state in (a) is safe

# Safe and Unsafe States (6)

- Now suppose we have the initial state shown in (a), but this time A requests and gets another resource, giving state (b)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

# Safe and Unsafe States (7)

- The scheduler could run B until it asked for all its resources, as shown in (c)
- Eventually, B completes and we get the state of (d). At this point we are stuck

Has Max		
A	4	9
B	4	4
C	2	7

Free: 0

(c)

Has Max		
A	4	9
B	—	—
C	2	7

Free: 4

(d)

# Safe and Unsafe States (8)

- There is no sequence that guarantees completion. Thus, the allocation decision that moved the system from (a) to (b) went from a safe to an unsafe state. Running A or C next starting at (b) does not work either
- The difference:
  - from a safe state the system can guarantee that all processes will finish
  - from an unsafe state, no such guarantee can be given

# The Banker's Algorithm for a Single Resource (1)

- The algorithm is modeled on the way a small-town banker might deal with a group of customers to whom he has granted lines of credit
- It checks to see if granting the request leads to an unsafe state. If so, the request is denied. If granting the request leads to a safe state, it is carried out

# The Banker's Algorithm for a Single Resource (2)

- In Fig. 6-11(a) we see four customers, A, B, C, and D, each of whom has been granted a certain number of credit units (e.g., 1 unit is 1K\$)
- The banker knows that not all customers will need their maximum credit immediately, so he has reserved only 10 units rather than 22 to service them
- The customers go about their respective businesses, making loan requests from time to time (i.e., asking for resources). At a certain moment, the situation is as shown in Fig. 6-11(b). This state is safe because with two units left, the banker can delay any requests except C's, thus letting C finish and release all four of his resources

# The Banker's Algorithm for a Single Resource (3)

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

Safe

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

Free: 2

Safe

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

Free: 1

Unsafe

Figure 6-11. Three resource allocation states



# The Banker's Algorithm for a Single Resource (4)

- Consider what would happen if a request from B for one more unit were granted in Fig. 6-11(b). We would have situation Fig. 6-11(c), which is unsafe
- An unsafe state does not have to lead to deadlock, since a customer might not need the entire credit line available, but the banker cannot count on this behavior
- The banker's algorithm considers each request as it occurs, seeing whether granting it leads to a safe state. If it does, the request is granted; otherwise, it is postponed until later

# The Banker's Algorithm for Multiple Resources (1)

- Consider Fig. 6-12, we see two matrices:
  - The one on the left shows how many of each resource are currently assigned to each of the five processes
  - The matrix on the right shows how many resources each process still needs in order to complete

# The Banker's Algorithm for Multiple Resources (2)

- As in the single-resource case, processes must state their total resource needs before executing, so that the system can compute the right-hand matrix at each instant
- The three vectors at the right of the figure show the existing resources,  $E$ , the possessed resources,  $P$ , and the available resources,  $A$ , respectively

# The Banker's Algorithm for Multiple Resources (3)

	Process	Tape drives	Plotters	Printers	CD ROMs
A	3	0	1	1	
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	

Resources assigned

	Process	Tape drives	Plotters	Printers	CD ROMs
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	

Resources still needed

E = (6342)  
P = (5322)  
A = (1020)

Figure 6-12. The banker's algorithm with multiple resources

# The Banker's Algorithm for Multiple Resources (4)

- The algorithm works as follows:
  - Look for a row,  $R$ , whose unmet resource needs are all smaller than or equal to  $A$ . If no such row exists, system will eventually deadlock
  - Assume the process of row chosen requests all resources needed and finishes. Mark that process as terminated, add its resources to the  $A$  vector
  - Repeat steps 1 and 2 until either all processes are marked terminated (safe state) or no process is left whose resource needs can be met (deadlock)

# Deadlock Prevention (1)

- Assure that at least one of the conditions stated below is never satisfied:
  - Mutual-Exclusion
  - Hold-and-Wait
  - No-Preemption
  - Circular wait

# Deadlock Prevention (2)

- Attacking the Mutual-Exclusion condition:
  - Avoid assigning a resource unless absolutely necessary, and try to make sure that as few processes as possible may actually claim the resource.

# Deadlock Prevention (3)

- Attacking the Hold-and-Wait condition:
  - All processes should be required to request all their resources before starting execution. If everything is available, the process will be allocated whatever it needs and can run to completion
  - Another approach is to require a process requesting a resource to first temporarily release all the resources it currently holds. Then it tries to get everything it needs all at once



# Deadlock Prevention (4)

- Attacking the No-Preemption condition:
  - First solution is to take back resources that process uses (nearly impossible)
  - Some resources can be virtualized. Spooling printer output to the disk and allowing only the printer daemon access to the real printer eliminates deadlocks involving the printer, although it creates a potential for deadlock over disk space

# Deadlock Prevention (5)

- Attacking the Circular Wait Condition:
  - One way is simply to have a rule saying that a process is entitled only to a single resource at any moment. If it needs a second one, it must release the first one

# Deadlock Prevention (6)

- Attacking the Circular Wait Condition (cont.):
  - Another way is to provide a global numbering of all the resources
  - Every process is granted access to a resource with number that is greater than number of previously acquainted resources
  - With this rule, the resource allocation graph can never have cycles (prove it yourselves)

# Deadlock Prevention (7)

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Figure 6-14. Summary of approaches to deadlock prevention

# Communication Deadlocks (1)

- Resource deadlock is a problem of competition synchronization
- Communication deadlock is an anomaly of cooperation synchronization
- Another kind of deadlock can occur in communication systems (e.g., networks), in which two or more processes communicate by sending messages

# Communication Deadlocks (2)

- Technique that can usually be employed to break communication deadlocks is timeouts
- In most network communication systems, whenever a message is sent to which a reply is expected, a timer is started

# Communication Deadlocks (3)

- If the timer goes off before the reply arrives, the sender of the message assumes that the message has been lost and sends it again (and again and again if needed). In this way, the deadlock is broken
- Not all deadlocks occurring in communication systems or networks are communication deadlocks. Resource deadlocks can also occur there
- Consider, for example, the network of Fig. 6-15

# Communication Deadlocks (4)

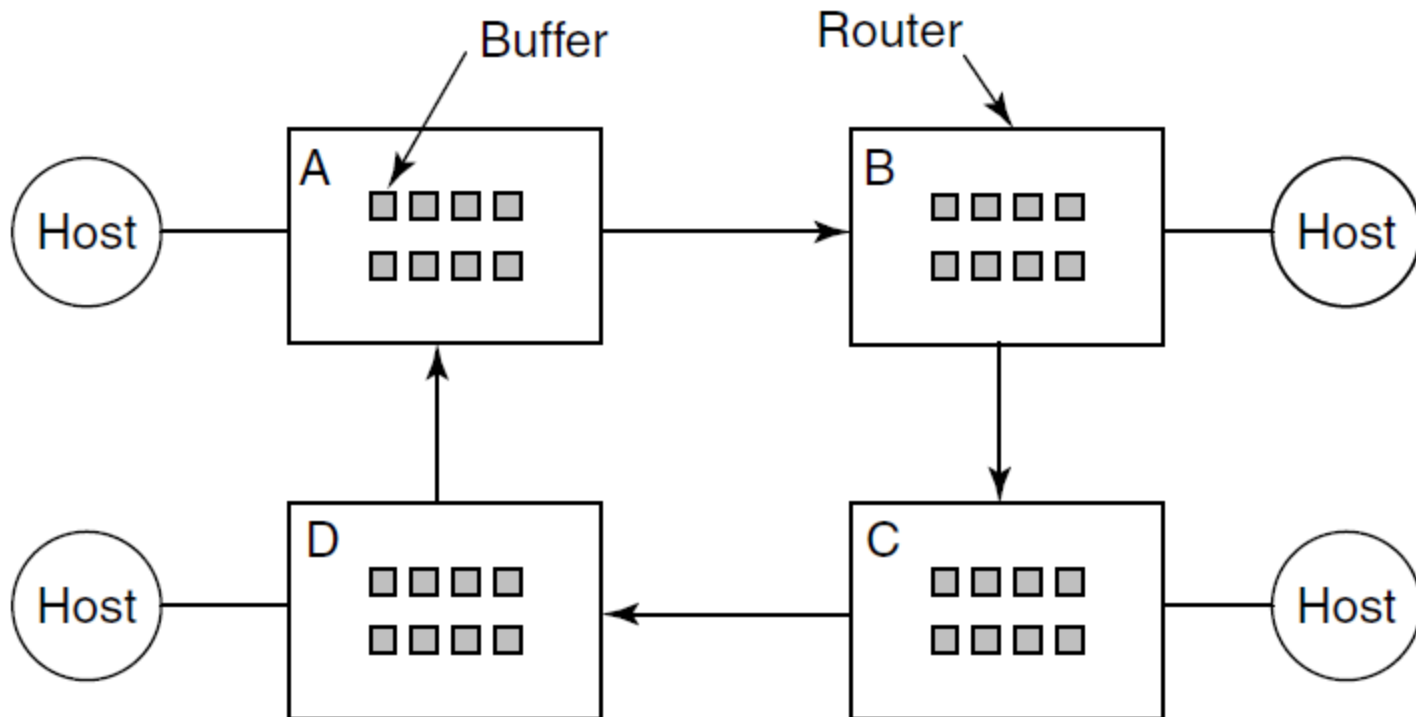


Figure 6-15. A resource deadlock in a network



# Communication Deadlocks (5)

- When a packet comes into a router from one of its hosts, it is put into a buffer for subsequent transmission to another router and then to another until it gets to the destination. These buffers are resources and there are a finite number of them

# Communication Deadlocks (6)

- Suppose that all the packets at router A need to go to B and all the packets at B need to go to C and all the packets at C need to go to D and all the packets at D need to go to A
- No packet can move because there is no buffer at the other end and we have a classical resource deadlock, albeit in the middle of a communications system

# Livelock (1)

- In some situations, a process tries to be polite by giving up the locks it already acquired whenever it notices that it cannot obtain the next lock it needs. Then it waits a millisecond, say, and tries again

## Livelock (2)

- However, if the other process does the same thing at exactly the same time, they will be in the situation of two people trying to pass each other on the street when both of them politely step aside
- And yet no progress is possible, because they keep stepping the same way at the same time

# Starvation (1)

- A problem closely related to deadlock and livelock is **starvation**
- The situation in which some processes are never getting service even though they are not deadlocked is called **starvation**

# Starvation (2)

- Starvation can be avoided by using a first-come, first-served resource allocation policy
- With this approach, the process waiting the longest gets served next
- In due course of time, any given process will eventually become the oldest and thus get the needed resource

End

Week 13 - Lecture

# References (1 / 2)

- Tanenbaum & Bos, Modern Operating Systems: 4th edition, 2013, Prentice-Hall, Inc.



# References (2/2)

- <http://rodrev.com/graphical/programming/Deadlock.swf>
- <http://www.utdallas.edu/~ilyen/animation/cpu/program/prog.html>
- <https://courses.cs.vt.edu/csonline/OS/Lessons/Processes/index.html>
- <http://inventwithpython.com/blog/2013/04/22/multithreaded-python-tutorial-withthreadworms/>