Slide 1

Part 5: Deadlocks and Starvation

- **Deadlock Problem**
- System Model
- Deadlock Conditions
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection – not examinable

Slide 2

The Deadlock Problem

- Deadlock: A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example 1:
  - System has two different types of tape drives; both P1 and P2 need the two tapes to finish execution
  - P_1 and P_2 each hold one tape drive and need the other
- Example 2: Semaphores *A* and *B*, initialized to 1

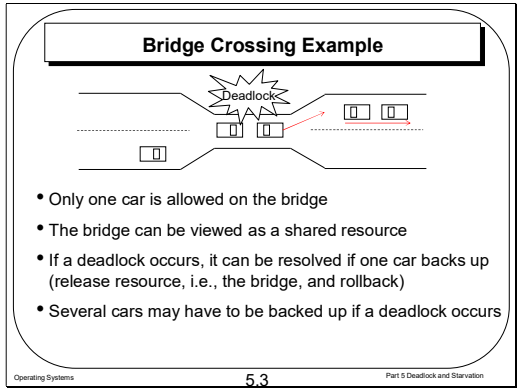
P_0	P_1
wait (*A*);	*wait* (*B*);
wait (*B*);	*wait* (*A*);

Context switch points

We have already seen deadlocks in the previous chapter. A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

Another example is incorrect usage of semaphores. Say, we have two semaphores A and B initialized to 1. The two processes P0 and P1 can deadlock. We have seen this example in the previous chapter.

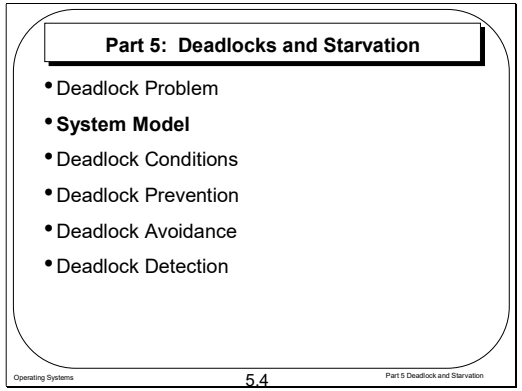
Slide 3



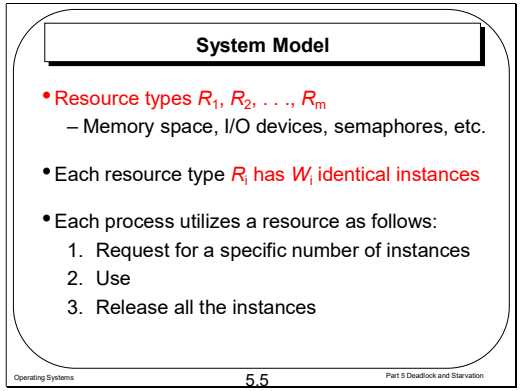
The bridge is a resource and the cars are processes.

Deadlock occurs if there are two cars on the bridge. When deadlock occurs, it can be resolved if one car backs up. Note, several cars may have to be backed up if a deadlock occurs. That means, deadlock recovery may affect other processes. Not only the process in the deadlock state are affected, but also other processes that have dependency on the deadlocked process.

Slide 4



Slide 5

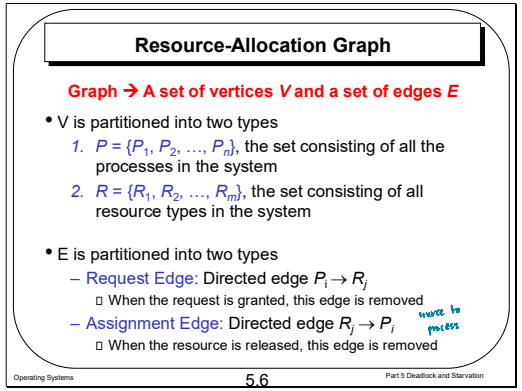


We now introduce the system model for deadlock. In this model, we have different resource types R1… Rm. The resource type can be memory space or I/O devices. Each resource type Ri has Wi identical instances. For example, we have two hard disks in the system. Ri is the hard disk and Wi=2. the instances are all identical.

A process utilizes a resource as follows: first sends a request of a certain number of instances.

If the instances are available, then the request is granted. Otherwise, the process needs to wait. If the request is granted, the process uses the resource, and after some time, releases the resource.

Slide 6

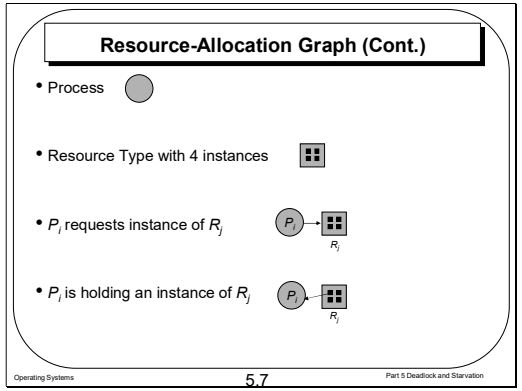


Consider the system has multiple processes and multiple resource types. How to model resource request, use and release?

We use a data structure called the resource-allocation graph to model the resource allocations in the system. The graph has a set of vertices V and a set of edges E. V is partitioned into two types: process vertex and resource type vertex. The process vertices represent all the processes, and resource vertices represent all the resource types in the system.

When a process sends a request for Rj, we add a directed edge from Pi to Rj. *When the request is granted, the request edge is removed. Then, we add an assignment edge between the two vertices in the reverse direction. When the resource is released, the assignment edge is removed.*

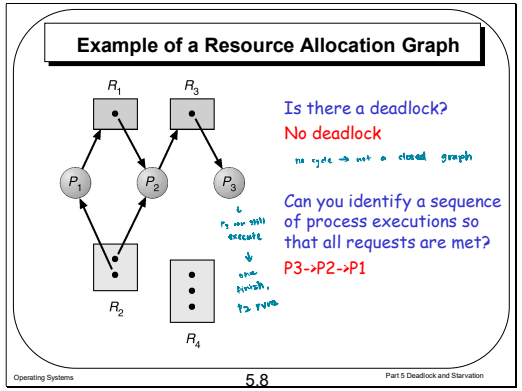
Slide 7



The number of solid black dots within the rectangle represents the number of instances for the resource.

When an instance of Rj is assigned to Pi, we draw an edge from one dot within the rectangle to Pi. Assignment of an instance of resources to the process: a certain dot within the resource vertex to the process vertex.

Slide 8

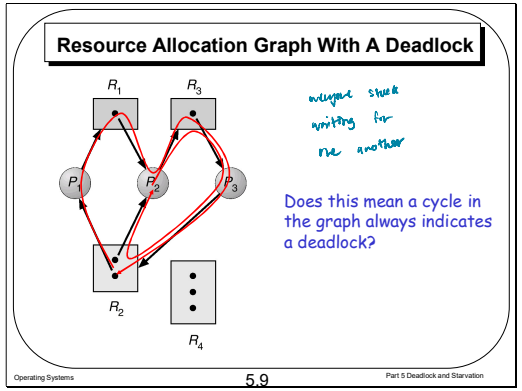


We have four resource types. Three processes.

We can actually examine the structure of the resource allocation graph to see whether the system has deadlock or not. Ideally, if the system has deadlock, the resource allocation must have a cycle. why? By the definition of deadlock ☺

In this case there is no deadlock. Can you find a sequence of process execution that all the processes finish execution? If P3 finishes, P2 can finish, and then P1 can finish.

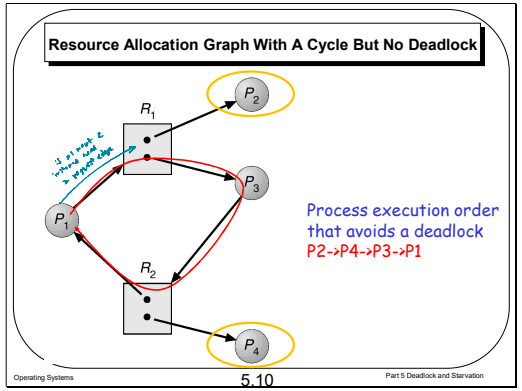
Slide 9



How many cycles are there in this resource allocation graph?

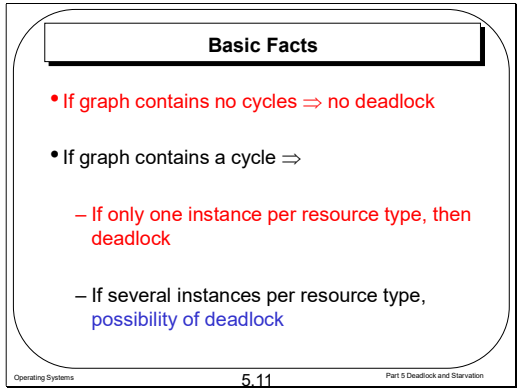
Two cycles. First cycle: P2 waits for P3 and P3 waits for P2. Second cycle: P1 waits for P2, P2 waits for P3, and P3 waits for both P1 and P2. In this case, no process can make any progress. →Deadlock.

Slide 10



One cycle. But no deadlock. Because P2 can finish and release R1. Then P4 can finish. Then, P3 and P1 can finish.

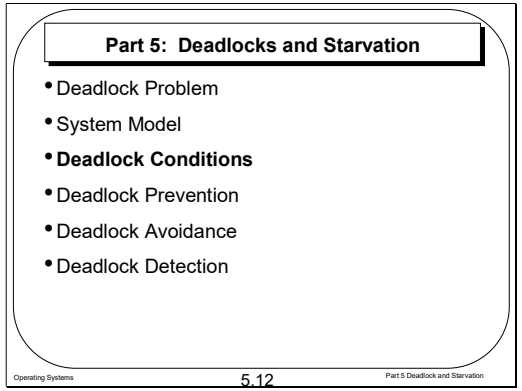
Slide 11



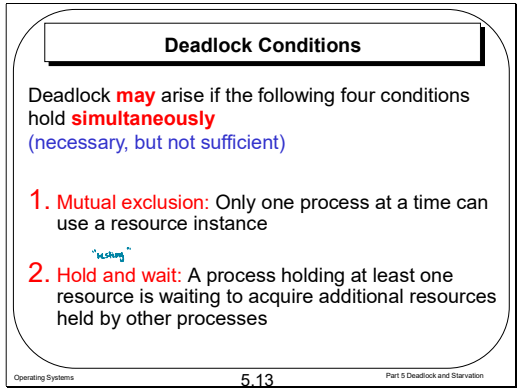
So, in summary, if a graph contains no cycle, no deadlock for sure. Because we do not have the problem of A waiting for B and B waiting for A.
If the graph contains a cycle, we need to look at the resource types. If only one instance per resource type, then there is a deadlock. Because no process can make any progress. If several instances per resource type, deadlock is possible, but not 100% sure.

We will introduce banker's algorithm to solve the problem with multiple instances later on.

Slide 12

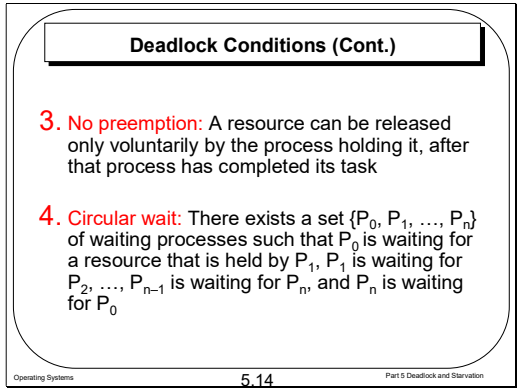


Slide 13



When there are multiple instances per resource type, we are not sure whether the graph has deadlock even if the graph has cycles. To handle this case, we study the deadlock conditions, that is, conditions necessary for a deadlock to occur. Deadlock can arise if the four conditions hold simultaneously. If any of them is violated, there is no deadlock. Also, the four conditions are necessary, but not sufficient.
We will briefly talk about the definitions here, and then give more examples in the later slides.

Slide 14



No preemption means we cannot take the resource from the process by force.
Circular wait corresponds to a cycle in the resource allocation graph.

Slide 15

Part 5: Deadlocks and Starvation

- Deadlock Problem
- System Model
- Deadlock Conditions
- **Deadlock Prevention**
- Deadlock Avoidance
- Deadlock Detection

Slide 16

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system
  - Used by most operating systems, including UNIX

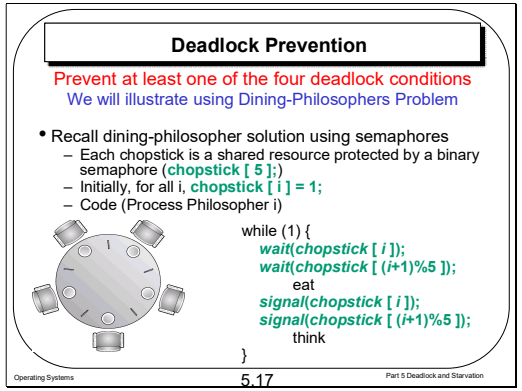
There are multiple approaches to handling deadlocks.

First, we can ensure that the system will never enter a deadlock state. That is deadlock prevention and avoidance.

Second, we can allow the system to enter a deadlock state and then recover. That is deadlock detection.

Third, we simply ignore the problem and pretend that deadlocks never occur in the system. That means, do nothing. That is so called Ostrich approach. Why do most systems use Ostrich approach? Efficiency. The overhead of deadlock detection and prevention is high. Why do we learn about deadlock detection and prevention then? Because OS doesn't implement deadlock, it is the user's responsibility to handle deadlock in their programs. So, we need to pay attention to deadlock issues when we develop our multi-programmed software.

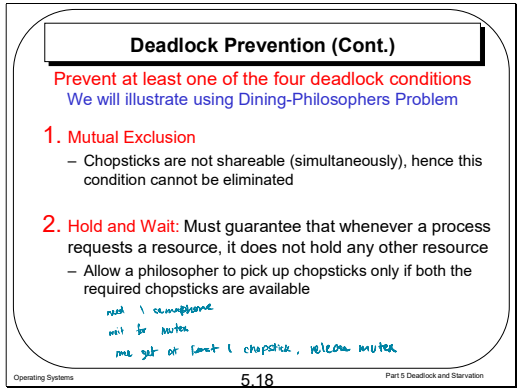
Slide 17



For philosopher i (0<=i<5): He first picks up his left chopstick, and then the right chopstick. This is done by calling the wait operation on the corresponding semaphore. If both operations can succeed, the philosopher can eat. After eating, he releases the left chopstick and then the right one. After that, the philosopher will think, and then repeat the process.

Does this solution have any problem? Each philosopher executes the first line and a context switch occurs. → deadlock.

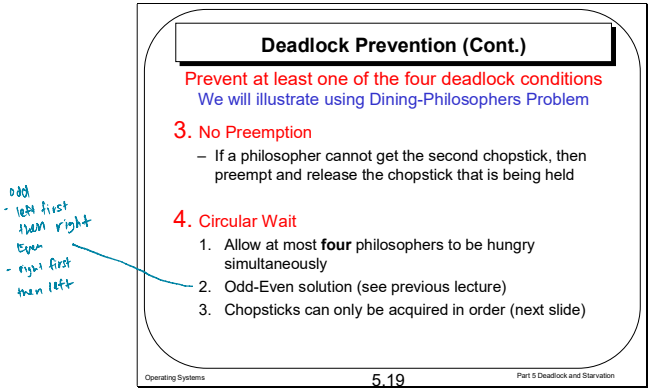
Slide 18



We will talk about how to prevent the deadlock conditions one by one. Recall, the four conditions are necessary to hold simultaneously for a deadlock to occur. So, we can prevent any one of them to happen. We illustrate that with Dining-Philosophers as an example.

Hold and wait: In Dining-Philosophers, allow a philosopher to pick up his chopsticks only if both the chopsticks are available. That means, at the beginning, the philosopher hasn't taken any chopstick. In that case, there is no hold and wait. So, no deadlock.

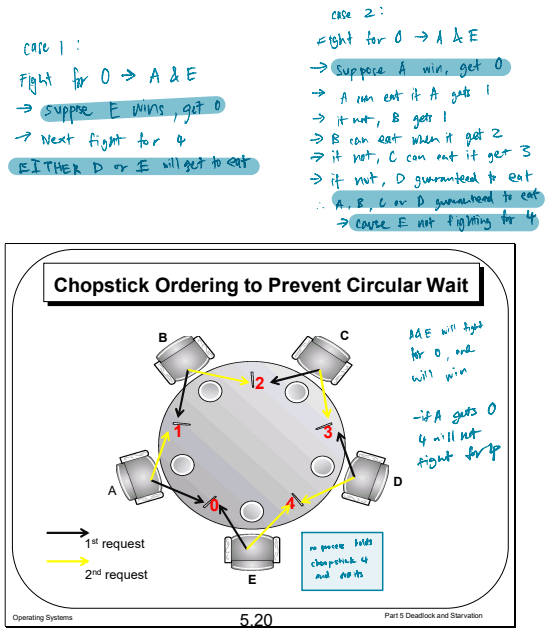
Slide 19



No preemption. To prevent this condition, we need to allow preemption. That is, If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released. The process restarts later. In Dining-Philosophers, if a philosopher cannot get another chopstick for a long time, preempt. That means, the philosopher needs to put down the chopstick, and then restarts.

Circular wait: In Dining-Philosophers, allow at most **four** philosophers to be hungry simultaneously. This is because of pigeon hole theory.

Slide 20



In the dining philosopher problem, each chopstick has an ID. Suppose each philosopher takes the chopstick with the small id first, and then takes the chopstick with the large id.
First request in black.
Second request in yellow.
→ No circular wait.
→ No deadlock.

Slide 21

Part 5: Deadlocks and Starvation

- Deadlock Problem
- System Model
- Deadlock Conditions
- Deadlock Prevention
- **Deadlock Avoidance**
- Deadlock Detection

Slide 22

Deadlock Avoidance

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that the system never goes into unsafe state
  - When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
  - If safe, the request is granted, and otherwise, the process must wait
- System is in safe state if **there exists a *safe completion sequence*** of all processes without deadlock

The deadlock prevention breaks one of the deadlock conditions. We now consider a more general deadlock avoidance algorithm. The algorithm is run whenever a process requests resources. The algorithm examines the resource-allocation state, and ensures that the system never goes into unsafe state. That is, when a process requests an available resource, system must decide if immediate allocation leaves the system in a **safe** state. If safe, the request is granted. Otherwise, the process must wait.

So what is safe state? System is in safe state if there exists a *safe sequence* of all processes. We do not require all sequences are safe. Only one safe sequence is sufficient.

Slide 23

Safe State

- A process completion sequence $<P_1, P_2, \ldots, P_n>$ is safe, if for each P_i, the resources that P_i requests can be satisfied by currently available resources + resources held by all the P_j, $j< i$
  - If P_i's needs cannot be immediately met, then P_i can wait until all P_j, $j<i$ have finished
  - When all P_j (j<i) are finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its resources, and so on
  - All processes in the sequence can finish

Sequence $<P_1, P_2, \ldots, P_n>$ is safe if for each P_i, the resources that P_i requests can be satisfied by currently available resources + resources held by all the P_j, with $j< i$. That is the processes before Pi in the sequence.

Slide 24

Example of Safe Process Sequence

Available: 1

Processes	Hold	Request
P1	1	1
P2	1	2
P3	1	3

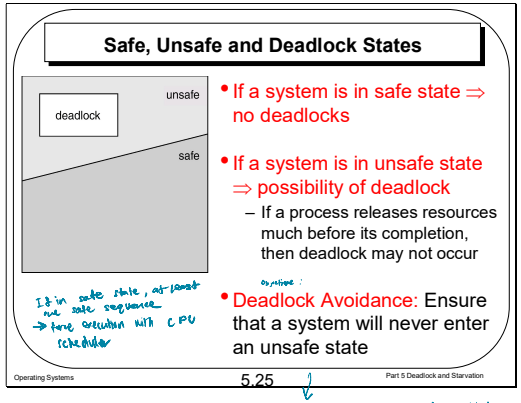
Q1: Is <P1, P2, P3> safe?
Yes
P1.request<=Available
P2.request<=P1.Hold+Available
P3.request<=P1.Hold+P2.Hold+Available

Q2: <P3, P2, P1> safe?
No
P3.request>Available

Q3: Is the system safe?
Yes
Exists one safe sequence
<P1, P2, P3>

Consider a system with a single resource type. The number of instances available is one. There are three processes. P1 holds one instance and requests for one more and so on.

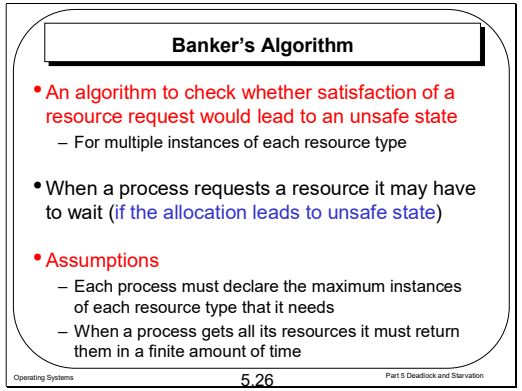
Slide 25



By definition, if a system is in a safe state, no deadlock.

If the system is in unsafe state, that means, there is no safe sequence for the system. It only means possibility of a deadlock. Because we are considering the worst case in the safe sequence: the resources are released only when the process finishes **(the worst case)**. If the process can release resources before termination. The later processes in the sequence can actually release the resources earlier. So the previous process have more resources to consume. The safe sequence does not consider this case. So, the unsafe state does not mean deadlock. For example, you can change the example in the previous slide: available =0. the system is unsafe. But, if P2 releases the resource earlier, then, all the processes can finish execution.

Slide 26

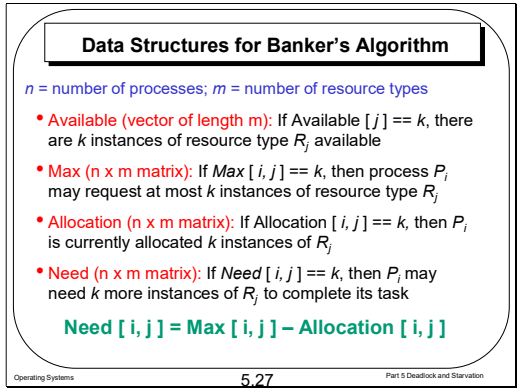


Now, let's learn the banker's algorithm for deadlock avoidance. The algorithm decides whether to meet a resource request, and to avoid the unsafe state. By its name: there are many bankers. They can borrow and lend money to each other. The banker's algorithm is to make sure that the bankers will not have a deadlock problem.

It can be used for multiple instances of each resource type. If all resource types have a single instance, what algorithm can we use? We can use resource-allocation graph, and see if the resource-allocation graph has a cycle.

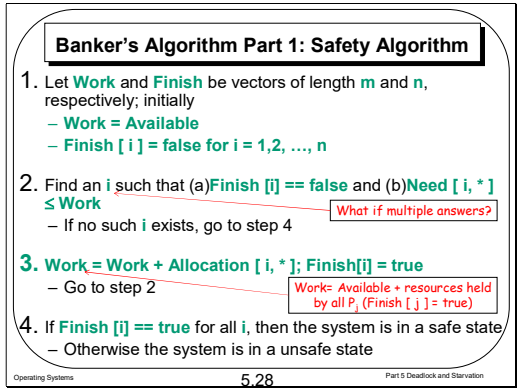
We have some assumptions on the process execution. Before execution, each process must declare the maximum number of instances of each resource type that it needs. When a process requests a resource it may have to wait. that is when the allocation leads to unsafe state. When a process gets all its resource, it must return them in a finite amount of time. That means, eventually, the process will finish, and the waiting time of other processes is bounded.

Slide 27



Before we introduce the banker's algorithm, we define the data structures it uses.

Slide 28

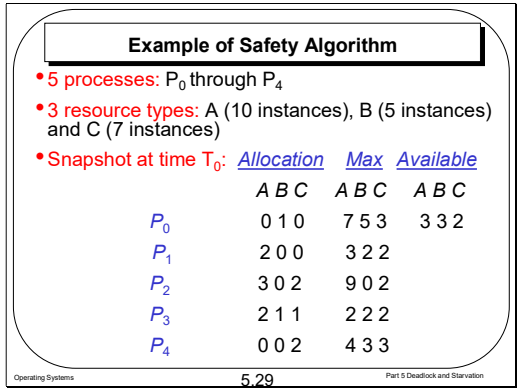


Let's first introduce safety algorithm. The algorithm determines whether the current system is in a safe state or not. If the system is in a safe state, we will find a safe sequence for the system.

In the first step, we define two vectors work and finish to denote the current system state. Work represents the available resources during the execution of the algorithm. We will update it in Step 3. Finish denotes whether a process can be finished. If Finish[i] is true, this means Pi can finish. It is also updated in Step 3.

What if multiple answers? Any one can be chosen. If we go to step 3 and check step 2 again, those processes still satisfy the condition.

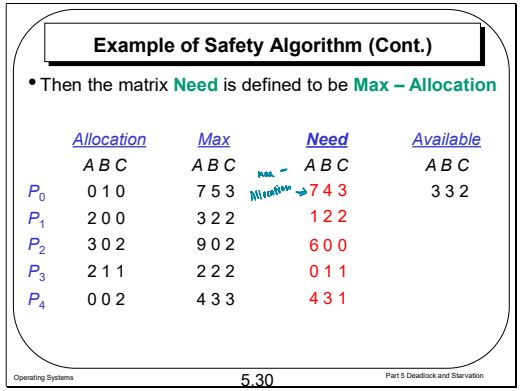
Slide 29



Now, let' s look at one example. There are five processes, P0 to P4. There are 3 resource types: A with 10 instances, B with 5 instances, and C with 7 instances.

The system state at time T0 is as follows.
Allocation
Max
Available
For example: P0 is allocated with 1 instance of B, and P0 may request 7 instances of A, and 5 instances of B, and 3 instances of C at most.

Slide 30



So, now, let's calculate Need. By definition, Need=Max- Allocation.

Slide 31

possible to have many completion sequences

Example of Safety Algorithm (Cont.)

- The working is as follows:

	Need	Allocation		Work A	Work B	Work C	Finished Process
	A B C	A B C					
				3	3	2	
P_0	7 4 3 ✓	0 1 0	+	2	0	0	P_1
				5	3	2	
P_1	1 2 2 ✓	2 0 0	+	2	1	1	P_3
				7	4	3	
P_3	0 1 1 ✓	2 1 1	+	0	0	2	P_4
				7	4	5	
P_4	4 3 1 ✓	0 0 2	+	3	0	2	P_2
				10	4	7	
			+	0	1	0	P_0
				10	5	7	

The system is safe (sequence <P_1, P_3, P_4, P_2, P_0>)

doesn't matter which one we start with first actually ① ②

Is there any other completion sequence? As long as you can find a completion sequence for the process, we can say the system is safe.

Slide 32

Banker's Part 2: Resource-Request Algorithm

$Request_i$ = Request vector for process **P_i**

– If **$Request_i$ [j] == k** then process **P_i** wants **k** instances of resource type **R_j**

V1<=V2 ≡ V1[j]<=V2[j], for all j

The i^{th} row of the matrix

1. If **$Request_i \le Need_i$**, go to step 2
   - Otherwise, raise error condition, since process has exceeded its maximum claim
2. If **$Request_i \le Available$**, go to step 3
   - Otherwise P_i must wait, since resources are not available

We now describe the algorithm for resource-request algorithm for a process Pi. We decide whether to grant the resource to the process or let the process wait.

The resource request by the process Pi is represented in a vector Request. If *Request* [*j*] == *k* then process P_i wants *k* instances of resource type R_j

Slide 33

Resource-Request Algorithm (Cont.)

3. Pretend to allocate requested resources to P_i by modifying the state as follows:
   - **Available = Available – Request$_i$;**
   - **Allocation$_i$ = Allocation$_i$ + Request$_i$;**
   - **Need$_i$ = Need$_i$ – Request$_i$;**

4. Run the safety algorithm
   - Safe ⇒ the resources are allocated to P_i
   - Unsafe ⇒ P_i must wait, and the old resource-allocation state is restored

Slide 34

Example of Resource-Request Algorithm

- Suppose P_1 requests (1, 0, 2) resources
- Step1: Check Request$_1$ ≤ Need$_1$ (1,0,2) ≤ (1,2,2) ⇒ *true*
- Step2: Check that Request$_1$ ≤ Available, i.e. (1,0,2) ≤ (3,3,2) ⇒ *true*
- Steps 3 & 4:

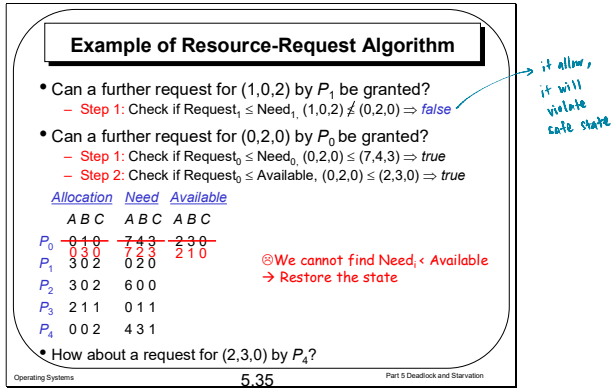
	Allocation	*Max*	*Need*	*Available*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	~~3 3 2~~ **2 3 0**
P_1	~~2 0 0~~ **3 0 2**	3 2 2	~~1 2 2~~ **0 2 0**	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Step 3: Pretend to allocate requested resources

Step 4: Executing safety algorithm shows that sequence <P_1, P_3, P_4, P_0, P_2> is safe

In step 3, we pretend to allocate the requested resources.
What are the parameters that we need to change?
Do we need to change Max? no.

Slide 35

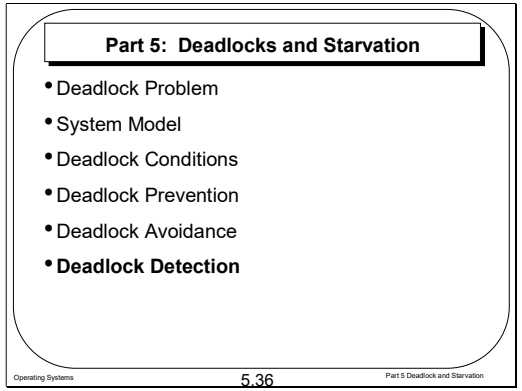


Consider another case: Can further request for (0,2,0) by P_0 be granted?
We can actually find that step 1 and step 2 are true. Thus, we pretend to give the resource to P0. Thus, we update the system state by modifying Allocation0, Need0 and Available0.
Then, in step 4, we run the safety algorithm. However, in step 2 of the safety algorithm, we cannot find any process with Needi<=Available. Thus, the system is unsafe.
What should we do? We should restore the old state.

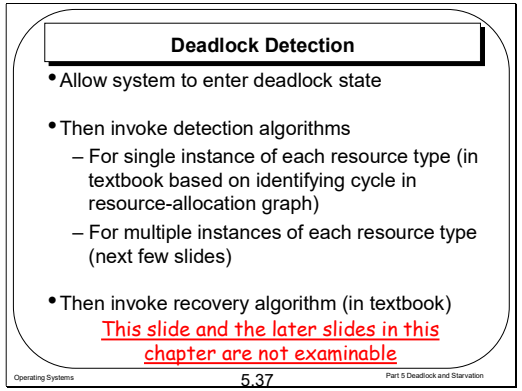
How about a request for (2,3,0) by P_4 ?
This is for your homework.

Slide 36

NOT EXAMINABLE

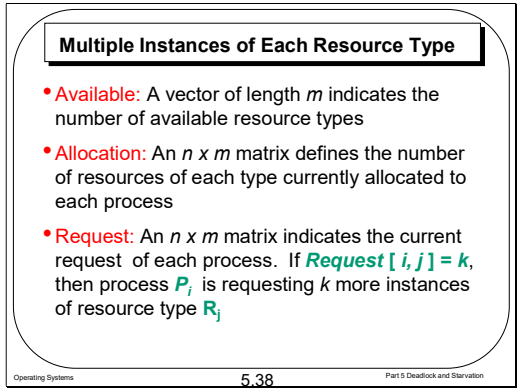


Slide 37



The basic idea of deadlock detection is to allow system to enter deadlock state, then invoke detection algorithm periodically. If there is a deadlock, we invoke the recovery algorithm.
The deadlock detection algorithm is actually similar to deadlock avoidance algorithm. If you are interested in them, you can refer to textbooks or just go through them in slides. They are not examinable.

Slide 38



Slide 39

Detection Algorithm

1. Let **Work** and **Finish** be vectors of length **m** and **n**, respectively; initially
   - **Work = Available**
   - For i = 1,2, …, n, if **Allocation$_i$!= 0**, then **Finish[i] = false**, else **Finish[i] = true**
2. Find an **i** such that (a)**Finish [i] == false** and (b)**Request [i, *] ≤ 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**, then system (process **P$_i$**) is deadlocked

Let's first introduce safety algorithm. The algorithm determines whether the current system is in a safe state or not. If the system is in a safe state, we will find a safe sequence for the system.

In the first step, we define two vectors work and finish to denote the current system state. Work represents the available resources during the execution of the algorithm. We will update it in Step 3. Finish denotes whether a process can be finished. If Finish[i] is true, this means Pi can finish. It is also updated in Step 3.

What if multiple answers? Any one can be chosen. If we go to step 3 and check step 2 again, those processes still satisfy the condition.

Slide 40

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_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

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

Slide 41

Example of Detection Algorithm (Cont.)

Suppose P_2 requests an additional instance of type *C*

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0 ✓	0 0 0
P_1	2 0 0	2 0 2 ☹	
P_2	3 0 3	0 0 1 ☹	
P_3	2 1 1	1 0 0 ☹	
P_4	0 0 2	0 0 2 ☹	

Work	Finished
A B C	
0 0 0	P_0
+ 0 1 0	
0 1 0	

Slide 42

Example of Detection Algorithm (Cont.)

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other process requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4