

ITEE



What's New:

There are a few gaps in the answers to give you a few details to think about.

Learning objectives for this week:

- Understand the major concepts we've covered, including:
 - what deadlocks are
 - how they are caused
 - how they are dealt with
- Apply these concepts to a specific situation, to tell whether a deadlock has or could occur
- Apply these concepts to choosing an appropriate strategy for a given situation
- Understand where and how deadlocks are likely to occur, and their significance in real systems

1. Concepts

- Why is each of the 4 conditions for deadlock *necessary* before a deadlock could occur? For the non-mathematicians, a *necessary* condition is one which must hold before something occurs, but the fact that the condition holds doesn't mean it *will* happen.
 - *mutual exclusion* – without it, processes don't have to wait for resources
 - *no pre-emption* – with pre-emption, resources can be taken away from processes to break the deadlock
 - *hold and wait* – if hold and wait doesn't occur, one or more processes can't end up waiting for each other
 - *circular wait* – results in a cycle in a wait-for graph, and hence, a deadlock if the other conditions hold

It's important to note that no one of these conditions on its own is sufficient to cause a deadlock. For example, if you have circular wait but can do pre-emption, you can break the deadlock by taking resources back.

- Show that circular wait could not occur unless hold and wait had occurred.
Each node in the wait-for cycle is holding a resource and waiting for another one, so hold and wait must have occurred.
- Explain why it is useful to have both conditions, even though circular wait implies hold and wait.
If you can show that hold and wait doesn't occur, then you know deadlocks can't occur. It can be easier than showing that circular wait doesn't occur, depending on the algorithm used. For example, deadlocks can be prevented by allowing a process only to request resources in a specific order, which prevents circular wait from happening (see p 252; explained in less detail: slide 15).

2. Safety

- Work through the example (slides 27–29) of the Banker's algorithm in the notes (slides 24–26) showing the values of each data structure at each step, showing that:
 - the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies the safety criteria

The following should be read as showing the sequence of steps as i is varied down the page (calculated values are shown in colour). So, for example, when $i=0$, the test of need against work fails, so we go on to $i=1$, when the test succeeds, and fail and work are updated. By observing the sequence in which the value of i results in the test succeeding, we can see that the given sequence of process completion will result in all being able to complete without deadlock.

	allocation	max	available	need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
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

```

finish      i      work = 3 3 2
false      0: need[0] ≤ work? no
false -> true 1: need[1] ≤ work? yes, work -> 5 3 2
false      2: need[2] ≤ work? no
false -> true 3: need[3] ≤ work? yes, work -> 7 4 3
false -> true 4: need[4] ≤ work? yes, work -> 7 4 5
    
```

try values of i which failed before again:

```

finish      i      work = 7 4 5
false->true 0: need[0] ≤ work? yes, work -> 7 5 5
true
false->true 2: need[2] ≤ work? yes, work -> 10 5 7
true
true
    
```

- if the request (1,0,2) is made by P_1 , the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies the safety criteria

Note in this case, the sequence we are required to show as working doesn't exactly go as increasing i from minimum to maximum, then starting at the beginning. The algorithm only tells us to "Find an i ", not how to find it. What would happen if we changed the order of the last two processes?

	allocation	max	available	need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	2 3 0	7 4 3
P_1	3 0 2	3 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

```

finish      i      work = 2 3 0
false        0: need[0]≤work? no
false -> true 1: need[1]≤work? yes, work -> 5 3 2
false        2: need[2]≤work? no
false -> true 3: need[3]≤work? yes, work -> 7 4 3
false -> true 4: need[4]≤work? yes, work -> 7 4 5

```

try values of i which failed before again:

```

finish      i      work = 7 4 5
false
true
false->true  2: need[2]≤work? yes, work -> 10 4 7
true
true

```

finally, try $i=0$, which passes the $need[2] \leq work$ test

- b. In general terms, is checking for safety a viable strategy? When might you use this strategy?

Advantages:

- won't lead to deadlocks

Disadvantages:

- high overhead, algorithm is $O(mn^2)$ where m = number of resources, n = number of processes; safety needs to be checked at every request
- processes can't get more resources than initially defined
- the number of resources initially requested has to be an over-estimate

Checking for safety is not a good general strategy because of the overhead. However, if you don't have many processes or resource types considered critical, and requests don't get made too often, then it would be a good approach. A hybrid strategy would be not to check for safety for resource types or processes not considered critical but it would be necessary to take into account variations like a non-critical resource being used by a critical process.

- a. How much simpler is the case where there is only one of each resource type? Would that scenario change your answer to (b)?

The Banker's Algorithm is less efficient than algorithms which apply to this special case (e.g., detecting cycles in the wait-for graph, which is $O(n^2)$ – while this looks the same as $O(mn^2)$ for $m = 1$, in practice, a cycle-detection algorithm would be faster).

3. Detection

- a. Single instance of each resource type:

- i. Why is it not generally practical to check for deadlock every time a resource is requested, using the approach for a single resource type covered in lectures?

It takes $O(n^2)$ time to check – and time proportional to n^2 could be impractically high with a reasonable number (n) of processes.

- ii. Would it be practical to check for a deadlock periodically? Explain difficulties which could arise.

Possibly. It would still be possible to get into a deadlocked state. You'd have to decide what to do once the deadlock is detected (killing processes, etc. which may cause undesired consequences; it may be hard to decide which ones really caused the deadlock). The deadlock detection algorithm could be run e.g. if CPU utilization dropped below a given level; deciding when to run it, in general, has to depend on the specific situation (how likely deadlocks are, the difficulty of undoing them, the importance of avoiding them). Another issue is determining which could most efficiently be terminated, taking into account lost work, highest chance of breaking the deadlock and any other performance criteria which fit the situation.

- b. Multiple instances of each resource type:

- i. Work through the given example of the detection algorithm, writing out contents of all the data structures at each step, for the example of slides 36–37.

Here is the initial example

	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			

```

finish      i      work = 0 0 0
false -> true 0: request[0]≤work? Y, work -> 0 1 0
false        1: request[1]≤work? no
false -> true 2: request[2]≤work? Y, work -> 3 1 3
false -> true 3: request[3]≤work? Y, work -> 5 2 4
false -> true 4: request[4]≤work? Y, work -> 5 2 6

```

try values of i which failed before again:

```

finish      i      work = 5 2 6
true
false        1: request[1]≤work? Y, work -> 7 2 6
true
true
true

```

book does 1 before 4: check that this works too

Now, P_2 requests 1 more resource 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			

```

finish      i      work = 0 0 0
false -> true 0: request[0] ≤ work? Y, work -> 0 1 0
false      1: request[1] ≤ work? no
false -> true 2: request[2] ≤ work? no
false -> true 3: request[3] ≤ work? no
false -> true 4: request[4] ≤ work? no

```

deadlock!

ii. Based on this example, how easy is it to back out of a deadlock?

Not very easily. Even if you kill one of the processes, you'd still have to run the detection algorithm again. It is unreasonable to kill all processes, and not all processes can be indiscriminately killed. We know here that the deadlock was caused by P_2 requesting 1 more resource of type C, but that's only because we ran the detection algorithm immediately before and after the request. If we were going to do this every time, we would instead run the safety algorithm. At the point where we detected the deadlock, if we hadn't done the previous example, all we know is that all processes except P_0 are now stuck. With a bit of work, in this case, we could determine that killing P_2 would fix the problem, but that may not be so obvious in other situations.

c. You are writing some code using a database in which records can be locked. Your code has to be able to roll back to a safe state if a deadlock is detected by the system. Outline steps you would take in designing your program to make it possible to undo calculations. You need to consider points at which a resource (in this case, a database record) is requested, and how to record what you've done since the last successful request of a resource. Assume that once you've released a resource you will not have to roll back. The algorithm without handling deadlocks looks like this:

```

lock customer record // nothing else locked at this point
authorize credit card payment
if authorization failed
    unlock customer record
    report failure
    exit
lock flight record
book seat
unlock flight record
debit credit card
unlock customer record

```

Think of each lock as acquiring a resource (the purpose of locks is to enforce mutual exclusion). The locks are marked in colour. The first doesn't present a hard problem: if a deadlock occurs there, there is no change in state to save: the process could back off. At the second, you would need to reverse authorization of the credit card. In real life, banks don't trust others billing a credit card to get this sort of thing right: usually, a credit card authorization sticks for a few days, even if it's not used (which can reduce your effective credit limit even if you didn't buy anything). Do you think handling deadlocks correctly is important, based on this example?

4. Think of any situation (real-world or computer) where deadlocks could occur

a. Show that the 4 conditions required for a deadlock all hold.

Traffic jam example as in the lectures. Four conditions:

- *mutual exclusion – only one car can be in one location at a time*
- *no pre-emption – cars cannot simply be removed from the road*
- *hold and wait – a car holds a space by occupying it, and waits for the space in front of it*
- *circular wait – the cars are waiting for the space held by the cars in front of them in a cycle around the block*

a. How would you prevent the deadlock by changing the rules which lead to it?

Deadlocks would be prevented if cars can't enter an intersection unless they can clear it. But don't you also need a priority rule, otherwise a clear intersection becomes the resource being waited for? Other than traffic lights, how else could this problem be solved? Think of road rules which may apply.

b. If instead you choose to do deadlock detection, will rolling back work?

Yes, the cars could be reversed. However, in general, not all things can be rolled back, e.g., in this case, if the traffic was backed up a long way, reversing wouldn't be practical.