Lecture 20 — Deadlock Avoidance

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

February 28, 2019

Deadlock Avoidance

We were attempting to rule deadlock out categorically by eliminating one of the three preconditions for deadlock to be possible.

If successful, then we can be sure that deadlock does not occur.

Unfortunately, eliminating the pillars came with some conditions and the best we could accomplish was merely making deadlock less likely.

Thus, we are forced to live with a system where a deadlock is possible.

Deadlock Avoidance

Deadlock being possible is not the same thing as deadlock being inevitable.

We can take steps to avoid it if there is a danger of it actually happening.

The basic strategy is: do not allow a cycle in the resource allocation graph.

Deadlocked Diners

In the dining philosophers problem: limit the number of concurrently-eating philosophers to four, even though the table has five seats.

With only four philosophers and five chopsticks, there were insufficient requests to complete a cycle.

This solution is suitable, but not necessarily generalizable to all situations.

Ordering of Resources

Impose ordering on resource requests. Recall this example:

Thread P

```
1. wait( a )
2. wait( b )
3. [critical section]
4. signal( a )
5. signal( b )
```

Thread Q

```
1. wait( b )
2. wait( a )
3. [critical section]
4. signal( b )
5. signal( a )
```

Deadlock would not take place if both threads requested these two resources in the same order, whether a then b or b then a.

Ordering of Resources

Let's generalize and formalize this principle.

The set of all resources in the system is $R = \{R_0, R_1, R_2, ...R_m\}$.

We assign to each resource R_k a unique integer value. Let us define this function as $f(R_i)$, that maps a resource to an integer value.

This integer value is used to compare two resources: if a process has been assigned resource R_i, that process may request R_j only if $f(R_j) > f(R_i)$.

Ordering of Resources

Note that this is a strictly greater-than relationship.

If the process needs more than one of R_i then the request for all of these must be made at once (in a single request).

To get R_i when already in possession of a resource R_j where $f(R_j) > f(R_i)$, the process must release any resources R_k where $f(R_k) \geq f(R_i)$.

If these two protocols are followed, then a circular-wait condition cannot hold.

Ordering of Resources

Back to the philosophers: we assign each chopstick a number from 0 to 4.

Each philosopher must then request them in ascending order.

The first philosopher requests 0, on her left, and then 1, on her right.

The second requests chopstick 1 and then chopstick 2.

This continues until the last philosopher who would previously have requested chopstick 4 and then 0, but under the new rules, this is forbidden.

This philosopher must instead request 0 on his right, and then 4, on his left.

This last philosopher will be blocked when trying to acquire chopstick 0 and it means chopstick 4 will be available for the second-to-last philosopher.

Thus, deadlock is avoided.

The Proof is the Proof

Assume a circular wait is present.

Let the set of processes in the circular wait be $\{P_0, P_1, \ldots P_n\}$ and the set of resources be $\{R_0, R_1, \ldots R_n\}$.

The cycle is formed as: P_i waits for resource R_i and that resource is held by P_{i+1}.

The exception is the case of P_n, which waits for resource R_n that is held by P_0.

Since Process P_{i+1} holds resource R_i while requesting R_{i+1}, this means $f(R_i) < f(R_{i+1})$ for all i.

But this means that $f(R_0) < f(R_1) < \ldots < f(R_n) < f(R_0)$.

It cannot be the case that $f(R_0) < f(R_0)$: a contradiction.

Ordering Resources

In development this is usually enforced just by coding convention and code review.

If you say that mutexes must always be acquired in alphabetical order (or their order in some file), if everyone sticks to that there will be no issue.

But may not be as easy as that in a real-world scenario…

Stay Alert, Stay Safe

Alternative: each process will need to give the operating system some additional information about what resources might be requested.

Processes need to say in advance of execution what is the maximum number of resources of each type they might conceivably need.

Perhaps process *A* needs the tape drive first, then the printer, and process *B* needs the printer and then the tape drive.

With this knowledge, the system can make more intelligent decisions about when to run a process or make it wait, to avoid getting into a deadlock.

Safe States

We say a state is safe if there is some scheduling order in which every process can run to completion.

Even if all of them suddenly request their maximum resources immediately.

Hence why we needed to know in advance the maximum resources that could be required by the process.

Safe Sequences

More formally, there must exist a safe sequence.

A sequence of processes $< P_1, P_2, \ldots P_n$ is a safe sequence in the current allocation state if:

For each P_i the resource requests that P_i can still make can be satisfied by the currently available resources plus resources held by P_j where $j < i$.

If a resource P_i needs is not currently available, P_i can wait until all P_j have finished and releases its resources.

When P_i terminates, P_{i+1} can obtain its needed resources and continue.

Safe/Unsafe

Any state that is not safe is considered unsafe.

If the system is in a safe state, then there is no deadlock.

Being in an unsafe state does not mean that there is a deadlock, but it means a deadlock is possible.

The analysis we do is the worst case scenario: every process immediately requests the maximum resources it could ever use.

Perhaps the processes do not make those requests in reality.

Safe State Example

There are three processes *A*, *B*, and *C*.

Assume there is only one resource, and a maximum of 10 instances exist.

Suppose *A* has 3 resources but may request up to 9.
B has 2 and may request up to 4.
C has 2 and may request up to 7.

There are 3 resources currently free.

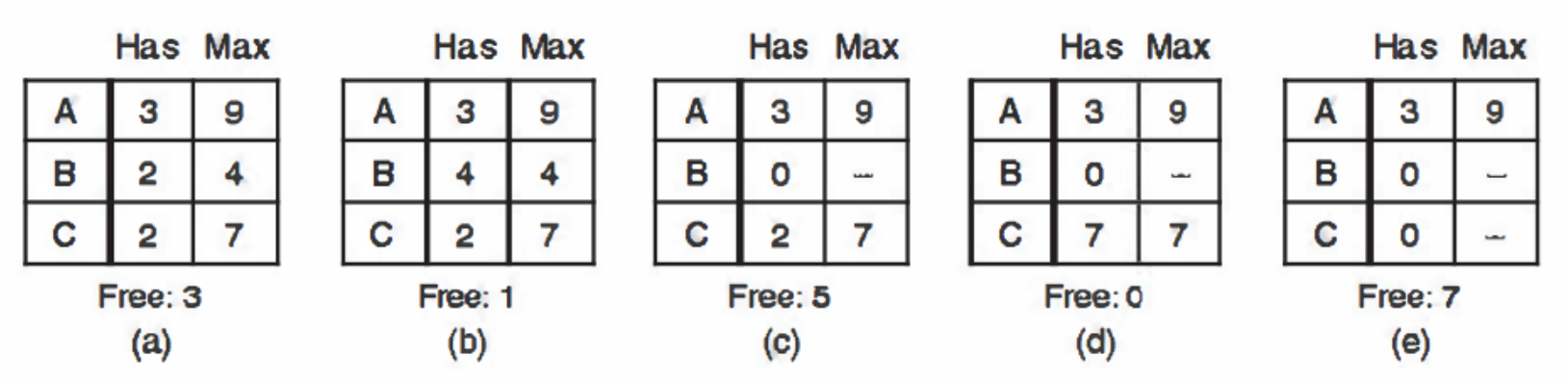
	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

Being Safe

For a state to be safe, we need one path that allows all processes to complete.

Multiple solutions may exist, and there may be paths that lead to deadlock.



	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5
(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0
(d)

	Has	Max
A	3	9
B	0	–
C	0	–

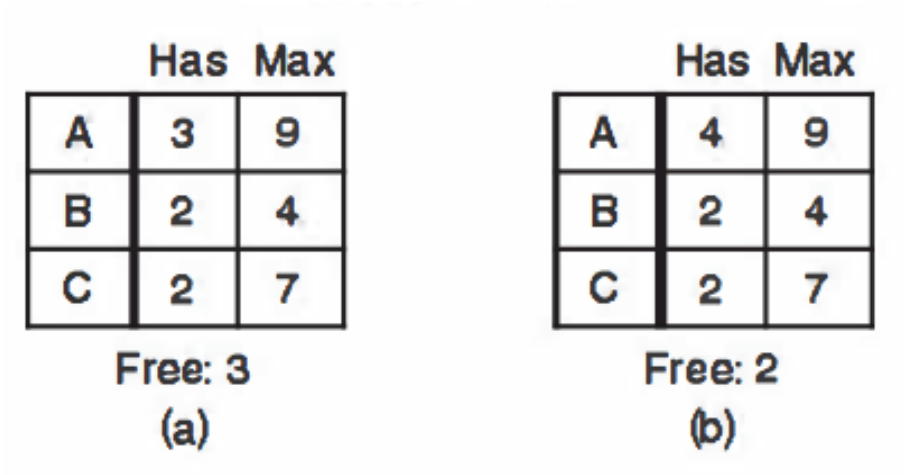
Free: 7
(e)

Crossing the Line

Suppose, however, *A* requests and gets another resource.

In that case, the initial condition has changed so that *A* has 4 resources and there are 2 free resources.

Or, in the diagram below, the state changes from (a) to (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)

Evaluating the New Initial State

Trying to find a way for all processes to complete is not possible.

Thus, this state is unsafe.

This does not mean that deadlock is present or certain.

The analysis is worst-case.

Resource-Allocation-Graph Algorithm

The fourth condition for deadlock is modelled, typically as being a resource allocation graph with a cycle in it.

Idea: let us use that idea to avoid deadlock by having the operating system maintain a resource allocation graph.

This works if there is only one instance of each resource and still requires that all the resources that a process will require must be declared in advance.

This condition does not have to be strictly adhered to if the system allows additional requests only when no requests have yet been granted.

Resource-Allocation-Graph

The model for resource allocation graphs earlier had two kinds of edges:

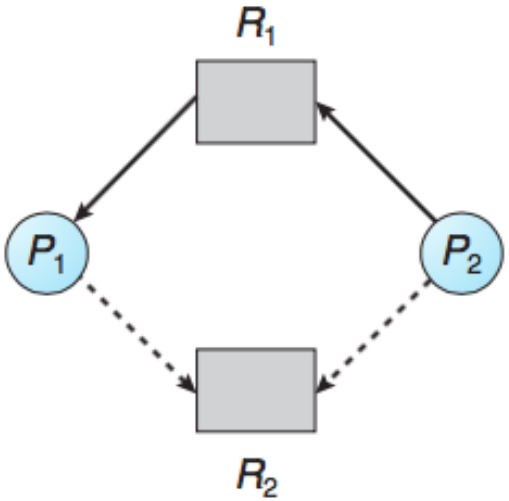
One representing requests (a process requests a resource); and One representing allocation (a resource currently belongs to a process).

We will require a new type of edge in the graph: a claim edge.

When the process actually makes the request for the resource, a claim edge is converted to a request edge.

Upon release the assignment edge reverts to a claim edge.

Resource Allocation Graph



A resource request will only be granted if converting the request edge to an assignment edge will not result in a cycle in the graph.

If no cycle is found, then allocation of the resource will not move the system into an unsafe state.

If a cycle is found, the request should not be granted, as it risks a deadlock.

The Banker's Algorithm

The previous algorithm is applicable only if all resource requests are known in advance and there is only one instance of each resource.

The banker's algorithm is more general: it allows for resources with multiple instances.

It received this name because it is hypothetically an algorithm that a small town banker might follow.

He or she is trying to prevent allocating the cash on hand in such away that he or she could no longer satisfy customers.

The Banker's Algorithm

The analysis we did earlier to determine if a state is safe or unsafe, is the foundation of the banker's algorithm.

	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)

Granting the request from process *A* moved the system safe to unsafe.

The operating system, when it receives a resource request, will evaluate the new state to see if it would transition the system to an unsafe state.

If so, the request will be denied or *A* will be blocked until the request can be fulfilled without putting the system in an unsafe state.

Holding to this condition means deadlock will be avoided.

The banker's algorithm can accommodate multiple resources.

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)

Checking if a State is Safe

1. Look for a row in the matrix, r, where the unmet resource needs are less than or equal to the available resources in A.
   If no such row exists, the system state is unsafe.
2. Assume the process from r gets all the resources it needs. Mark that process as terminated and put all its resources into A, the available pool.
3. Repeat steps 1 and 2 until either:
   (i) all processes are marked terminated and the initial state was safe; or
   (ii) no process remains whose needs can be met; initial state is unsafe.

If more than one process may be chosen in step 1, it does not matter which.

The pool of available resources will either stay the same or get larger.

The Banker's Algorithm

Determine if granting a resource makes the system unsafe: what-if calculation.

Assume the resource granted; do the safe state calculation given that new state.

If the result is that the state is unsafe, the request should be deferred or denied.

The Banker's Algorithm: Useless?

As great as the banker's algorithm is in theory, in practice it is utterly useless.

Processes rarely know in advance what their maximum resource needs will be.

The number of processes is not fixed, but varies with users' wishes.

Finally, a resource that was thought to be available can suddenly vanish.

Thus in practice, the banker's algorithm can almost never be used.

Summary of Avoidance

Avoidance techniques may be useful in preventing a deadlock from occurring without having to preempt resources or forbidding hold-and-wait conditions.

Unfortunately, it requires a fixed number of known resources to allocate.

It also requires foreknowledge about what resources a process might need.

Finally, because it deals with the worst case, it reduces system performance in the name of safety.

Other Recommendations

The avoidance techniques discussed previously require perfect information (or accurate worst-case guesses) about all the resources a process will ever need.

In a world where this information is not known, thus far, all we have managed is to make deadlock less likely.

Deadlock Avoidance Recommendations: 1

Minimize the number of critical sections and their length.

Mutual exclusion should be applied only where it is needed and to the minimal number of program statements.

Deadlock Avoidance Recommendations: 2

All tasks must release semaphores and resources as soon as possible.

This is not just polite, but also makes it less likely that processes are stuck waiting for one another.

Deadlock Avoidance Recommendations: 3

Do not suspend tasks when in the critical region. Avoid calling yield or some other call known to block.

Deadlock Avoidance Recommendations: 4

All critical regions must be error-free.

This is obviously a desirable property for the whole program, but in critical region we must be especially careful.

Deadlock Avoidance Recommendations: 5

Do not allocate resources in interrupt handlers.

Any resource allocation might block, and if an interrupt handler gets blocked, the whole system can be stuck.

Deadlock Avoidance Recommendations: 6

Always perform validity checks on pointers (or null checks) in critical regions.

A segmentation fault or exception in the critical region can also mean never releasing the lock.