Lecture 21 — Deadlock Detection & Recovery

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 10, 2019

Elementary...

Detecting Deadlock

Thus far we have examined several ways to prevent or avoid deadlock, but all of those solutions have come with significant drawbacks or limitations.

Avoidance analyses are also conservative; they will prevent a request from taking place if there is even a small chance it could lead to a deadlock.

Perhaps we cannot stop deadlock from happening or cannot live with the performance reduction that avoidance mandates.

Then the next best thing is to let all resource requests proceed and then determine later if a deadlock exists, and if so, do something about it.

Detecting Deadlock

The basic strategy for deadlock detection is like the deadlock avoidance strategy in that it relies on a model of the resource allocation and requests.

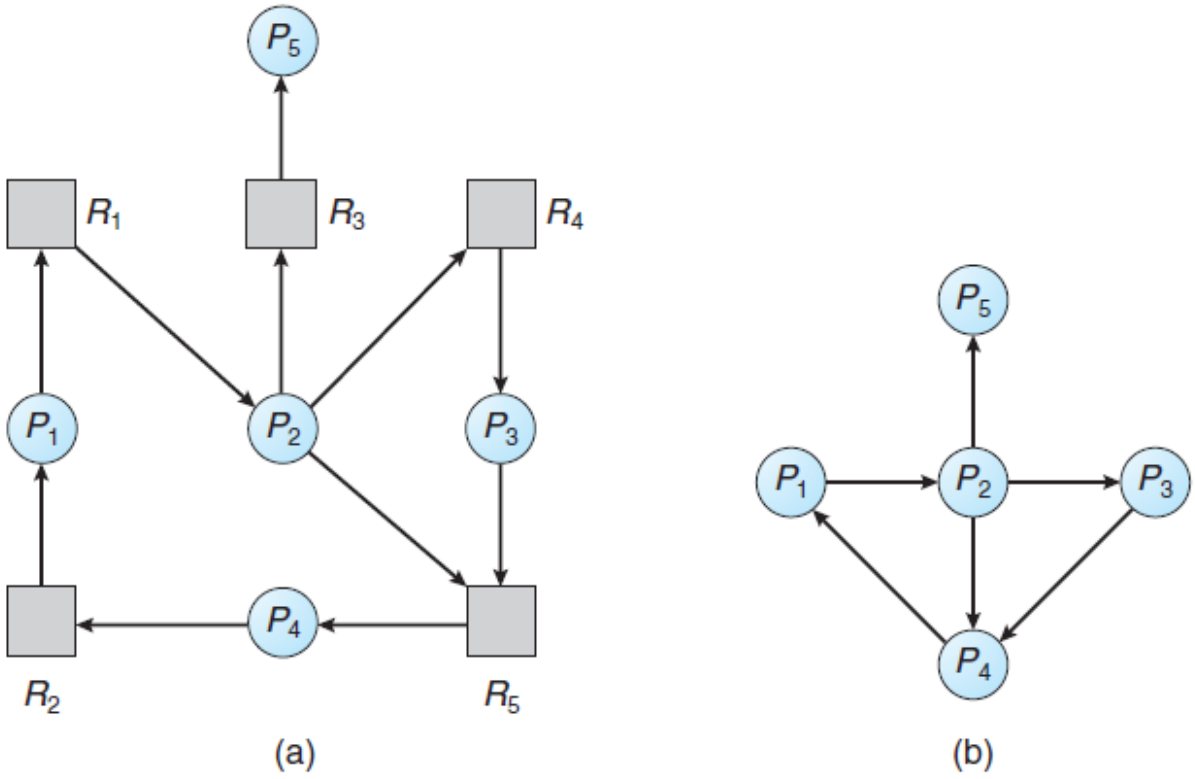
If resources have only a single instance, we may reduce the graph to a simplified version called the *wait-for* graph.

This removes the resource boxes from the diagram.

It indicates that a process P_i is waiting for process P_j rather than for a resource R_k that happens to be held by P_j.

An edge $P_i \rightarrow P_j$ exists in the wait-for graph if and only if the resource allocation graph has a request $P_i \rightarrow R_k$ and an assignment edge $R_k \rightarrow P_j$

Wait-For Graph



Cycle Detection

Given the wait-for graph, it is trivial for humans to look at this and determine if there is a cycle, but for the computer it takes slightly more work.

A cycle exists in the wait for graph if and only if a deadlock exists in the system.

Such cycle detection algorithms tend to have runtime characteristics of $\Theta(n^2)$ where n is the number of nodes in the graph.

General Deadlock Detection Algorithm

We will use the general deadlock detection algorithm that allows for multiple resources of each type.

There are n processes numbered P_1 through P_n and m resources.

Resources are represented by two vectors:

E – the existing resource vector.

A – the available resource vector.

If resource i has two instances total and one is currently assigned to a process, E_i is 2 and A_i is 1.

General Deadlock Detection Algorithm

We need two matrices to represent the current situation of the system.

The first is C, the current allocation.

Row i of C shows how many of each resource P_i has.

The second matrix is R, the request matrix.

C_{ij} shows the number of instances of resource j that P_i has and R_{ij} shows the number of instances of resource j that P_i wants.

General Deadlock Detection Algorithm

Resources in Existence

$[E_1, E_2, \ldots, E_m]$

Resources Available

$[A_1, A_2, \ldots, A_m]$

Current Allocations

$$\begin{bmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{bmatrix}$$

Requests

$$\begin{bmatrix} R_{11} & R_{12} & \ldots & R_{1m} \\ R_{21} & R_{22} & \ldots & R_{2m} \\ \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & \ldots & R_{nm} \end{bmatrix}$$

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

Less Than? Says Who?

There is one more bit of setup before we are ready to run the algorithm.

The key idea is comparison of vectors, so let us define for notational convenience, the idea of "less than" for two vectors.

We will say that for two vectors A and B of length m, $A \leq B$ means that $A_i \leq B_i$ for all i from 1 to m.

Run, Algorithm, Run

All processes are unmarked and the vectors and matrices are populated.

1. Search for an unmarked process whose requests can all be satisfied with the available resources in A. Mathematically: find a process P_i such that row $R_i \leq A$.
2. If a process is found, add the allocated resources of that process to the available vector and mark the process. Mathematically: $A = A + C_i$. Go back to step 1.
3. If no process was found in the search of step 1, the algorithm terminates.

At the end, any processes that are not marked are deadlocked.

General Deadlock Detection Algorithm Analysis

This approach is similar to the banker's algorithm.

Step 1: look for a process that can run to completion; it will be able to do so because the currently available resources equal or exceed its needs.

That process can finish, and when it does so, its currently-held resources are released and available for another process to acquire.

Step two reflects this by adding the resources it holds to the available set.

Then another process is selected.

At the end, either all processes can finish and there is no deadlock, or there is a set of processes (at least two) that are deadlocked.

This algorithm has a runtime performance characteristic of $\Theta(m \times n^2)$.

Detection of Deadlock Example

System initial state (4 resources, 3 processes):

$$E = [4, 2, 3, 1] \qquad A = [2, 1, 0, 0]$$

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

When to Detect Deadlock

The runtime characteristic of the simple deadlock detection algorithm: $\Theta(n^2)$.

The runtime characteristic of the general algorithm was shown as $\Theta(m \times n^2)$.

(n is the number of processes and m is the number of resources in the system)

This means that the deadlock detection routine is expensive to execute.

When to Detect Deadlock

This prompts a question: how often should the deadlock detection algorithm be run?

One strategy is to run it every time a resource is requested.

Optimization: it should run if a resource request cannot be granted.

Another idea: run it periodically instead.

When to Detect Deadlock

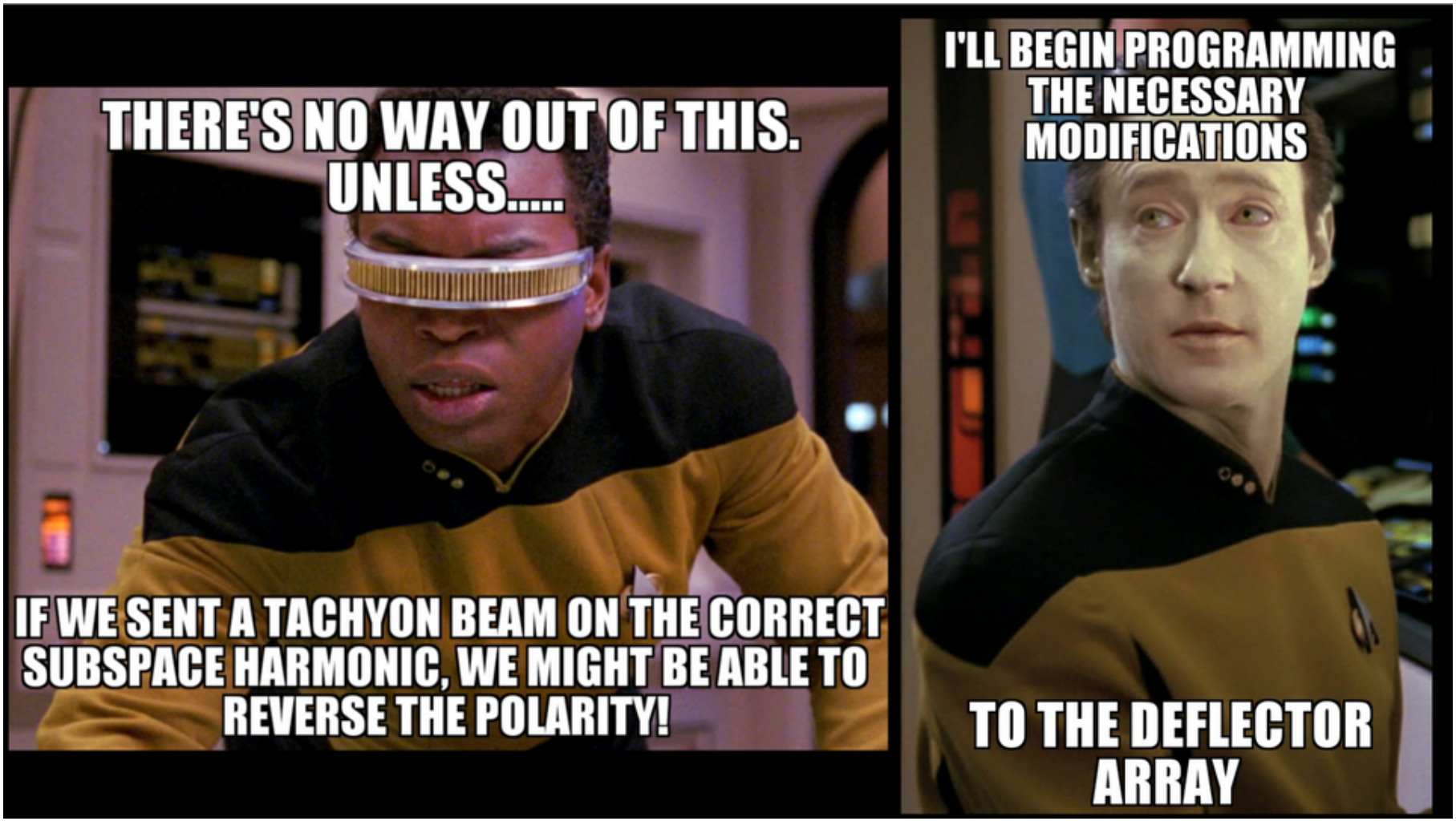
When to run the deadlock detection algorithm depends on how often we expect deadlock to occur, and how severe a problem it is when deadlock occurs.

If deadlock happens a lot, checking for deadlock often will make sense.

If the consequences of a deadlock are severe, it makes sense to check frequently to identify the problem as soon as possible.

Idea: run it when CPU utilization is low…?

We Can Fix This



Deadlock Recovery

Once a deadlock has been detected, a system can recover from that deadlock by "breaking" the deadlock.

These are called recovery strategies and they are ways the system may automatically deal with the problem.

It is possible to have a manual form of deadlock recovery, where an operator is notified and that person is responsible for sorting out the problem.

The manual method needs no further discussion.

Recovery Strategies

There are several strategies that we could apply.

They can result in data loss, delays in completion, other problems…

Ideally we would like to break the deadlock with as little disruption as possible, so the strategies that rely on selection of victims should choose carefully.

These strategies are not mutually exclusive.

Here's One Idea

Robbery

The strategy of “robbery” is just a humorous way of saying preemption.

This is virtually identical to the discussion of deadlock prevention.
But it happens only when deadlock is detected.

The operating system needs to choose a victim to rob…

The resource should be an appropriate type to be preempted!

Robbery

Suppose P_1 has a resource R_1 and needs R_2, while P_2 has R_2 and needs R_1.

The OS may block P_2 and take away R_2 from P_2 and allow P_1 to have it.

After that resource becomes available again, it is returned to P_2.

Mass Murder

Mass Murder

Terminate (kill) all the processes involved in the deadlock.

This solution is surprisingly common.

It is one way to be certain that the deadlock cycle is broken.

The resources that these processes were holding will become available.

There is no need to determine a victim: kill 'em all and let root sort them out.

Mass Murder

This solution, while easy to implement, may not solve the problem.

The circumstances that caused the deadlock may occur again.

If the deadlock were an unlikely situation caused by “unlucky” timing then it will probably not recur, or at least, will not recur for some time.

If P_1 and P_2 are deadlocked, however, is it really necessary to kill both?

If we killed only one, the other could proceed…

Hello, Clarice...

Murder

Perhaps instead of killing all processes involved in a deadlock, the operating system chooses to kill processes selectively.

Like preemption, selecting which process is the victim is important.

When the victim is killed, its resources are added to the available set and this will hopefully allow other processes to proceed.

Murder

Run the deadlock detection algorithm again to determine if deadlock still exists.

If so, this strategy needs to be repeated.

Repeat until the logjam is broken.

Presumably kills fewer processes than the “mass murder” strategy.

Next Idea: Back we Go...

Time Travel

Rollback: returning the state of a process to a saved state from an earlier time.

There must be a saved state that was created in advance of a problem.
Otherwise there is no state to roll back to!

The saved state is called a checkpoint and the act of creating and saving a checkpoint is called checkpointing.

Checkpoints may be created periodically or before beginning a particular operation that requires a lot of resources.

Time Travel

A checkpoint contains the memory image, including the call stack, and resource state of a process.

It is written to disk and will usually persist as long as the process executes.

You might already be somewhat familiar with the concept of rollbacks if you have used version control software like Subversion (`svn`) or Git (`git`).

Rollbacks are also common in databases.

Time Travel

Unfortunately, rollback does not always succeed.

Sometimes moving the process back to an earlier state just moves it a few steps back on the same road that leads to the deadlock.

Rollback may be attempted a few times before trying another strategy.

Armageddon


LAUNCHES DETECTED
NIAMEY HIT, 2.3M DEAD
KIGALI HIT, 2.5M DEAD
NAMPULA HIT, 5.3M DEAD
LAUNCHES DETECTED
DEFCON
EVERYBODY DIES

Armageddon

Armageddon - the end of the world.

If a deadlock has occurred, sometimes the best thing to do is reboot the system.

This has a side effect of killing all processes, whether they are stuck or not, but is sometimes the best way to make sure that the system is in a valid state.

NASA's Spirit rover relies on this strategy if it detects a deadlock.

Like killing all affected processes, it is easy to implement, but is most disruptive.

Victim Selection

If we have to choose a victim process for one of the strategies, e.g., termination, then we need a strategy for which process to choose.

We could choose randomly: kill a process and hope that that was enough.

Sometimes this will work, because it breaks the deadlock and the other processes can all proceed.

In general, however, making an informed decision is better.

(Don't) Shoot Innocent Bystanders

It is not necessary to choose one of the processes involved in the deadlock.

If P_1 and P_2 are deadlocked, P_3 may have an instance of a resource P_1 needs.

Killing P_3 will allow both P_1 and P_2 to proceed.

Victim Selection: An Optimization Problem

Define a cost function for choosing each process, evaluate it, choose the lowest.

Some factors to consider:

1. The priority of the process.
2. How long the process has been executing.
3. How long is remaining in execution, if known.
4. What resources the process has (number and type).
5. Future resource requests, if known.
6. Whether the process is user-interactive or in the background.
7. How many times, if any, the process has been selected as a victim.

Geri and the Atrics

Selection routines tend to favour older processes rather than younger ones.

This is not because older processes vote in higher numbers, but because it tends to be more expensive to restart an older process.

A process that has been running for a long time, if it is restarted, has to do a lot more work to get to the point where it was terminated than a younger process.

Geri and the Atrics

If the oldest process were constantly the one selected, that process itself might never get to run to completion (starvation).

If the killing process is very aggressive then no tasks run to completion because each process, shortly after becoming the oldest, is killed.

Therefore, young processes tend to be the ones selected.

Keeping Track

The final element in the list, keeping track of how many times a process has been victimized, is also there to prevent starvation.

The selection process very likely produces the same or similar results each time it is run, so it may happen that the same process is selected over and over again.

It may be advisable to take the number of rollbacks or terminations into account so that no process in particular is starved.

Miscarriages of Justice

The deadlock detection algorithm we have chosen tends to be conservative.
It will err on the side of saying that there is a deadlock.

The worst case is assumed: that processes take resources and keep them until the end of their execution.

We might detect a deadlock when there is none.

We may also, then, kill an innocent process in a system that is not actually deadlocked. Oops!

It’s Okay to Make Mistakes

It turns out that our deadlock detection algorithms do not have to be perfect if we have chosen an appropriate recovery strategy.

If killing the process and restarting it does not have unexpected side effects, then being selected does not have an impact on the correctness of the program.

… Only on how long it takes to execute.

For example, the victim process is a compiler.

If killed and started again, it takes slightly longer for the compile to finish, but there is no impact on the correctness of the binary.