ECE254: Operating Systems and Systems Programming Spring 2015

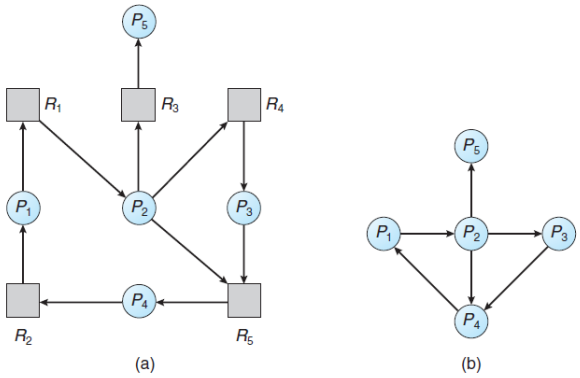
Lecture 17 — Deadlock Detection & Recovery

Jeff Zarnett

Deadlock Detection

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. If we cannot stop deadlock from happening or cannot live with the performance reduction that avoidance mandates, then perhaps 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.

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 and 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 $R_k \rightarrow P_j$ [SGG13].



(a) A resource allocation graph and (b) its corresponding wait-for graph [SGG13].

Given the wait-for graph, detection of a deadlock is fairly simple. It is virtually trivial for humans to look at this and determine if there is a cycle, but for the computer it takes slightly more work. We must execute an algorithm to determine if there is a cycle. 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 from [Tan08] that allows for multiple resources of each type. In this algorithm, there are n processes numbered P_1 through P_n and m resources. Resources are represented by two vectors: E, the existing resource vector – the total number of instances of each resource; and A, the available resource vector – how many instances of each resource are currently available (not assigned to a process). If resource i has two instances total and one is currently assigned to a process, E_i is 2 and A_i is 1.

We need two matrices to represent the current situation of the system. The first is C, the current allocation; it contains data about what resources are currently assigned to each process. Thus, row i of C shows how many of each resource P_i has. The second matrix is R, the request matrix. Row i of this matrix shows how many of each

resource P_i wants. Thus, 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. Or, to show what those looklike:

Resources in Existence
$[E_1, E_2, ..., E_m]$

Resources Available
$[A_1, A_2, ..., A_m]$

Current Allocations

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

Requests

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

Note that at all times a resource is considered either allocated or available. This means this mathematical relationship always holds:

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

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.

At last, the algorithm. The starting condition is that all processes are unmarked and the vectors and matrices described above are populated. The algorithm will go through all processes and determine if they can complete, under worst-case conditions (keeping all resources until termination), and if they can, marks them. At the end, any processes that are not marked are deadlocked.

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, the algorithm terminates.

This approach is similar to the banker's algorithm. Step one looks for a process that can run to completion, and we are certain 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 2) that are deadlocked.

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

Deadlock Detection Algorithm Example Let's now use this algorithm on an example also from [Tan08]. Let us assume we have 4 types of resources and 3 processes. The names and types of the resources do not matter - they can be anything - all that matters are the numbers. Each process has some resources currently allocated to it, and each process has some requests outstanding. Thus, the initial state of the system is:

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

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

Now, carry out the algorithm. The first process cannot proceed because its request R_{14} cannot be met; there are no instances of resource 4 available. Process 2 cannot proceed either, because it needs resource 3 and there are none of those available either. Process 3 can run and will eventually return its resources, changing the available vector to $A = [2, 2, 2, 0]$. Mark process 3.

Process 1 still cannot run because its need for resource 4 still cannot be met. Process 2, however, can, and it will do so, returning all its resources. The available vector is changed to $A = [4, 2, 2, 1]$. Mark process 2.

Process 1 can now run. Return its resources: $A = [4, 2, 3, 1]$. Mark process 1.

There are no remaining unmarked processes in the system, and therefore no deadlock. As a sanity check, compare vector E and the final values for A – they should be the same if there is no deadlock.

When to Detect Deadlock

The runtime characteristic of the simple deadlock detection algorithm was identified as $\Theta(n^2)$, and the runtime characteristic of the general algorithm was shown as $\Theta(m \times n^2)$ where 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.

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. Running the algorithm might be rather expensive, so perhaps this is too often. An obvious optimization: it should only run every time a resource request cannot be granted (that is, a process gets blocked). Another idea: run it periodically instead.

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 deadlock is severe, it makes sense to check frequently to identify the problem as soon as possible.

Several sources including [Tan08] suggest running the deadlock detection algorithm when CPU utilization is low. This is not only because it would be a bad idea to run an algorithm that is time consuming and computationally expensive while the system is busy. When a deadlock is present in the system, many processes are stuck and cannot proceed, so a drop in CPU usage may be an indication that many processes are deadlocked.

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, but the manual method needs no further discussion.

Recovery Strategies

There are several strategies that we could apply, which we will refer to in humorous terms. All are valid solutions of various complexity. Unfortunately, none of the solutions are particularly pleasant. They can result in data loss, delays in completion of tasks, and may occasionally be used on an innocent victim.

Robbery

The strategy of "robbery" is just a humorous way of saying preemption. A process P_1 has a resource R_1 and needs R_2, while a process P_2 has R_2 and needs R_1. A classic deadlock. The operating system may block process P_2 and take away R_2 from P_2 and allow P_1 to have it. In a general case, the operating system will take resources from ("rob") some process(es) and give those resources to other processes until the deadlock cycle is broken.

The resource must be an appropriate type to be preempted. As in the discussion of deadlock prevention, it must be possible to save and restore the state. Preemption of a printer is not realistic, nor is memory. However, other

resources may be. Ideally we would like to do this as little as possible, so we need to choose a victim properly.

Mass Murder

If the operating system detects a deadlock, it may choose to terminate (kill) all the processes involved in the deadlock. This is one way to be certain that the deadlock cycle is broken. The resources that these

This may not solve the problem, however, as the circumstances that caused the deadlock may occur again if all the processes are restarted. 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.

Murder

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

Time Travel

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.

Victim Selection and Miscarriages of Justice

The deadlock detection algorithm we have chosen tends to be conservative in that it will err on the side of saying that there is a deadlock. This is because the worst case is assumed: that processes take resources and keep them until the end of their execution. In practice, however, processes will release resources (or at least they should!) so we might detect a deadlock when there is none. We may also, then, kill an innocent process. Oops!

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

References

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.

[Tan08] Andrew S. Tanenbaum. *Modern Operating Systems, 3rd Edition*. Prentice Hall, 2008.