CMPSCI 377 Operating Systems **Fall 2008**

Lecture 12: October 9

Lecturer: Prashant Shenoy *Scribe: Shashi Singh*

12.1 Deadlocks

A deadlock is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does. It is a logical error that can occur when programming with threads. A *deadlock* (also known as "the deadly embrace") happens when two things (threads, processes, etc) wait on each other. For example:

```
thread A
    printer.wait
    disk.wait

thread B
    disk.wait
    print.wait
```

Yet another example of a deadlock is known as the *Dining Philosophers problem*. In this abstract problem, philosophers alter between thinking and eating. Each philosopher needs two forks to eat with. The problem is that each philosopher only gets one fork at a time; if one philosopher gets one of the forks, the next of gets the other, etc, in a circular way, we get a deadlock. The philosophers will **starve**! One of the main aspects to notice about this problem is that we must necessarily have threads competing for a finite number of resources; if we have had an infinite amount of forks, there would be no deadlock. We now enumerate the conditions needed for a deadlock to occurs; notice that *all* of them are necessary, and none is sufficient:

1. *Mutual exclusion* condition: a resource that cannot be used by more than one process at a time.
2. *Hold-and-wait* condition: each thread holds one resource while waiting for another.
3. *No preemption* condition: thread can only release resources voluntarily. No other thread (or OS) can force the thread to release.
4. *Circular wait* condition: two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds.

Deadlock can only occur in systems where all 4 conditions hold true.

12.1.1 Deadlock detection

Deadlocks can be detected on-the-fly, by running cycle detection algorithms on the graph that defines the current use of resources. Let the graph being discussed have one vertex for each resources $(r_1 \dots r_m)$ and one for each thread $(t_1 \dots t_n)$. We say that there is an edge from a thread to a resource if that thread is using

that resource; if there is an edge from a resource to a thread, that resource is owned by the thread. Given this graph, we can run any cycle detection algorithm. If a cycle is found, we have a deadlock; we might then either kill all threads in the cycle, or kill threads one at a time (thus forcing them to give up resourcers) and hope that we will need to kill few threads before the deadlock is resolved.

12.1.2 Deadlock prevention

Preventing deadlocks is fairly easy. Remember that the list presented before enumerates conditions which are *all* necessary for a deadlock to occur; therefore, it suffices that at least *one* of those conditions does not hold.

1. Removing the mutual exclusion condition means that no process may have exclusive access to a resource. This proves impossible for resources that cannot be spooled, and even with spooled resources deadlock could still occur. Algorithms that avoid mutual exclusion are called non-blocking synchronization algorithms.

2. The "hold and wait" conditions may be removed by requiring processes to request all the resources they will need before starting up (or before embarking upon a particular set of operations); this advance knowledge is frequently difficult to satisfy and, in any case, is an inefficient use of resources. Another way is to require processes to release all their resources before requesting all the resources they will need. This too is often impractical. (Such algorithms, such as serializing tokens, are known as the all-or-none algorithms.)

3. A "no preemption" (lockout) condition may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent or thrashing may occur. However, inability to enforce preemption may interfere with a priority algorithm. (Note: Preemption of a "locked out" resource generally implies a rollback, and is to be avoided, since it is very costly in overhead.) Algorithms that allow preemption include lock-free and wait-free algorithms and optimistic concurrency control.

4. The circular wait condition: Circular wait prevention consists of allowing processes to wait for resources, but ensure that the waiting can't be circular. One approach might be to assign a precedence to each resource and force processes to request resources in order of increasing precedence. That is to say that if a process holds some resources, and the highest precedence of these resources is m, then this process cannot request any resource with precedence smaller than m. This forces resource allocation to follow a particular and non-circular ordering, so circular wait cannot occur. Another approach is to allow holding only one resource per process; if a process requests another resource, it must first free the one it's currently holding (or hold-and-wait).

12.1.3 Deadlock Prevention with Resource Reservation

Threads provide advance information about the maximum resources they may need during execution. A sequence of threads $t_1, ..., t_n$ is *safe* if for each t_i, the resources that t_i can still request can be satisfied by the currently available resources plus the resources held by all $t_j, j < i$. A *safe* state is a state in which there is a safe sequence for the threads. An unsafe state is not equivalent to deadlock, it just may lead to deadlock, since some threads might not actually use the maximum resources they have declared. We grant a resource to a thread only if the resulting new state is safe. If the new state is unsafe, the thread must wait even if the resource is currently available. This alagorithm ensures no circular-wait condition exist, and hence no deadlock.

12.2 Increasing concurrency

Suppose we have one object that is shared among several threads. Suppose also that each thread is either a reader or a writer, that readers only read data but never modify it, and that writers read and modify data. Now the question is: if we know which threads are reading and which ones are writing, what can we do to increase concurrency?

First, we obviously have to forbid two writers from writing at the same time. Moreover, a reader cannot read while a writer is writing. There is not problem, however in allowing lots of people reading at the same time.