**CMPSCI 377 Operating Systems** **Fall 2013**

# Lecture 10: October 7

*Lecturer: Prashant Shenoy* *Scribe: Armand Halbert*

## 10.1 Dining Philosophers Problem

The Dining Philosophers Problem is an illustrative example of a common computing problem in concurrency. The dining philosophers problem describes a group of philosophers sitting at a table doing one of two things - eating or thinking. While eating, they are not thinking, and while thinking, they are not eating. The philosophers sit at a circular table each with a bowl of spaghetti. A chopstick is placed in between each philosopher, thus each philosopher has one chopstick to his or her left and one chopstick to his or her right. As spaghetti is difficult to serve and eat with a single chopstick, it is assumed that a philosopher must eat with two chopsticks. The philosopher can only use the chopstick on his or her immediate left or right.

The philosophers never speak to each other, which creates a dangerous possibility of deadlock. Deadlock could occur if every philosopher holds a left chopstick and waits perpetually for a right chopstick (or vice versa). Originally used as a means of illustrating the problem of **deadlock**, this system reaches deadlock when there is a 'cycle of unwarranted requests'. In this case philosopher $P_1$ waits for the chopstick grabbed by philosopher $P_2$ who is waiting for the chopstick of philosopher $P_3$ and so forth, making a circular chain.

The lack of available chopsticks is an analogy to the locking of shared resources in real computer programming. Locking a resource is a common technique to ensure the resource is accessed by only one program or chunk of code at a time. The challenge occurs when there are multiple resources which must be acquired individually. When several programs are involved in locking multiple resources, deadlock can occur. For example, one program needs two files to process. When two such programs lock one file each, both programs wait for the other one to unlock the other file, which will never happen.

### 10.1.1 Solution to the Dining Philosophers problem

A naive solution is to first wait for the left chopstick using a semaphore. After successfully acquiring it, wait for the right chopstick. After both chopsticks have been acquired, eat. When done with eating, release the two chopsticks in the same order, one by one, by calls to *signal*. Though simplistic, this solution may still lead to starvation (literally) when every philosopher around the table is holding his/her left chopstick and waiting for the right one.

The correct solution is to let a philosopher acquire both the chopsticks or none. This can be done within a *monitor* wherein only one philosopher is allowed to check for availability of chopsticks at a time (within a *test* function). See the lecture slides for the complete pseudocode.

## 10.2 Deadlocks

A deadlock is a situation wherein two or more competing actions are waiting for the other to finish, and thus none ever stop waiting. It is a logical error that can occur when programming with threads. Note that deadlock isn't starvation because in starvation progress is being made but in deadlock there is no progress. A *deadlock* happens when two threads wait on each other. For example:

```
thread A
    printer.wait
    disk.wait

thread B
    disk.wait
    printer.wait
```

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 cannot be used by more than one thread at a time.
2. *Hold-and-wait* condition: atleast one 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.

### 10.2.1 Deadlock detection

Deadlock detection is when we try to find deadlocks after they have occurred and try to take corrective action. Corrective action is usually something harsh like killing one of the threads that are in deadlock. For this reason deadlock detection isn't typically done by the OS but instead it is done by the user or in a library. 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 ($r_1 \ldots r_m$) and one for each thread ($t_1 \ldots t_n$). 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. This is only true if there is one instance of each resource. With multiple instances of resources there might not be deadlock even if a cycle is detected (see the slides for an example). If deadlock is detected we might then either kill all threads in the cycle, or kill threads one at a time (thus forcing them to give up resources) and hope that we will need to kill few threads before the deadlock is resolved.

We can also use the above described graph to avoid deadlock. This would involve checking the graph whenever a resource was requested. The detection of a cycle would indicate an unsafe state. A request for a resource is denied whenever it would result in an unsafe state. This system could also take an edge where the denied thread has a claim on a resource and reverse it so that that thread would have to wait on that resource also (allowing the system to progress). Unfortunately this system doesn't work for when we have multiple instance of the same resource.

### 10.2.2 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 thread may have exclusive access to a resource. This isn't very practical if we want concurrency.

2. A "no preemption" condition may also be difficult or impossible to avoid as a thread has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent. This also isn't very practical.

3. The "hold and wait" conditions may be removed by requiring threads 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 threads to release all their resources before requesting all the resources they will need. This too is often impractical.

4. The circular wait condition: Circular wait prevention consists of allowing threads 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 threads to request resources in order of increasing precedence. That is to say that if a thread holds some resources, and the highest precedence of these resources is m, then this thread 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. Often this can be done by the programmer and this is how programmers that deal with concurrency often tend to program.

### 10.2.3 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 algorithm ensures no circular-wait condition exist, and hence no deadlock.