CS 390
Chapter 8 Homework Solutions

8.1 `List three examples of deadlocks ...`

Here are several. See if you can think of more.

- Two people meet on a narrow sidewalk, and each tries to move to the side, but both move in the same direction, repeatedly.
- Two cars try to cross a single-lane bridge from opposite directions.
- Two people use a ladder - one is trying to go up and the other come down.
- Two trains are traveling in opposite directions on the same track.
- Two dogs try to use a doggy-door at the same time and get their heads stuck, but because they are dogs who don't understand computer science, they try to force their way through anyway.
- That kid who was always trying to climb the slide when you wanted to go down during recess in elementary school and wouldn't get off.

8.7 `Can a system detect that ...`

Recall that "starvation" occurs when a process is indefinitely denied access to a resource that it needs. The problem with detecting starvation is that we need to determine when a process is being *indefinitely* denied some resource. If a process is denied a resource for 1 ms, does that mean the resource will be denied indefinitely? What about 1 hour? One day?

One way to solve this problem might be to arbitrarily decide that any process that is denied a resource for t units of time is being starved (even if we cannot prove that the resource will be denied indefinitely). When a process requests a resource, a timer is created that will expire in t units of time. If the resource request is granted, the timer is deleted. If the timer expires then the process is considered to be starving. Can you think of an example of a situation where this would not work?

8.9 `Consider the following snapshot of a...`

a. The system state is unsafe.

b. The system state is safe. $<P_1, P_2, P_0, P_3, P_4>$ is a safe sequence.

8.11 `Is it possible to have ...`

Yes. Suppose a system has a single instance of a resource. (Say, the system has one DVD burner.) A process requests this instance, the request is granted, and then the process requests another instance. The process will wait forever on itself.

If we don't allow processes to request more instances of a resource than exist in the system, then a single-threaded process cannot deadlock itself. This follows directly from the hold-and-wait condition.

Additional Exercises

A Consider a traffic situation where we have four one-way streets surrounding a single block. (Think of the square downtown.) Each street is bumper-to-bumper with cars, and each intersection at the end of a street is blocked by a car that has just moved into the intersection from the perpendicular street.

1. Show that the four necessary conditions for deadlock hold in this example.
2. State a simple rule for avoiding deadlocks in this system.

B This question illustrates that deadlock can happen in ways that are not always unintuitive.

Consider a system that has 10 GB of memory, and allows at most 9 processes in the system at any time. Each process is allowed at most 2 GB of memory.

1. Explain how this system could deadlock over memory requests.
2. Suppose that we only allow processes to request memory in chunks of 1GB. Explain why the system now is no longer vulnerable to memory deadlocks.

Answers to Additional Exercises

A This is an example of what is known as *gridlock* in the field of traffic engineering. We can model gridlock using OS techniques if we treat the cars as processes and the roads as resources. In this case, the amount of space taken up by one car is considered one "instance" of the road resource.

1. **Mutual Exclusion** Since only one car can occupy a given spot on the road, the "road resources" are mutually exclusive.

   Hold and Wait The hold and wait property is true, because each car holds one instance of the road resource, and is waiting for another instance. (In particular, each car is waiting for the road instance held by the car in front of it.) No car will release the space it occupies until it can move forward.

   No Preemption It is not possible to preempt a car that holds a piece of roadway, since there is no way (short of using a crane) to stop the car from holding the resource.

   Circular Wait Notice that each car is waiting on the car ahead of it, with the exception of the cars trying to enter the intersection - each of these cars is waiting on the car that is currently in the intersection. This chain forms a cycle of cars: each car is holding a section of road and also waiting on a section of road, so the hold and wait property is true.

2. Do not allow a car to advance into the intersection unless it is clear that the car can pass completely through the intersection. This prevents the circular wait property from holding.

   Many cities try to prevent gridlock by posting signs that say "Do Not Block Intersection" at each intersection. Unfortunately, there is always some driver (hopefully, none of you) who ignores the sign and enters the intersection, even when it is clear that they cannot completely pass through it.

   This pinhead then has to stop in the middle of the intersection. Usually, these are the same people who read and send text messages, make phone calls, and eat while they are driving. You should feel free to shake your fist at these people.

B 1. Suppose that every process requests $2^{30} - 1$ bytes of memory. (This is one byte less than 1GB.) Each of these memory requests can

be granted. Each process can still request $2^{30} + 1$ bytes and without exceeding its maximum memory request. However, the amount of free memory remaining in the system is less than that amount, so no process's maximum memory request can be satisfied. The system is vulnerable to deadlock.

2. If any process requests 1 GB of memory, its request will be satisfied immediately. Now suppose one of these processes requests an additional GB of memory. Its request will be satisfied because there is at least 1GB still free. This process will terminate, releasing all the memory it holds allowing the system to eventually satisfy all future 1GB requests.

This shows that requiring processes to request *more* instances of a resource can actually be safer under certain circumstances than allowing them to request fewer.