(1) What is “process deadlock”? How is it different from “process starvation”?

- Process deadlock is a situation where two or more processes are unable to make any progress when they are waiting for each other (e.g., waiting for the resources held by each other) *1 and *2.

- Process deadlocks are different from process starvations in that no process (those involved in a process deadlock) will be able to make any progress once a process deadlock occurs. It means that they will be stuck forever (unless with some intervention by (human) administrator). Process starvations are just fairness issue. There always is (are) some processes running, leaving a chance that everyone still can finish if lucky without any intervention.

**Note *1:** Any solution such as “process deadlock is a situation where processes can not proceed because they can not get the resources they need” is not a good definition for “process deadlocks” (it is even wrong as a definition for “process deadlock”).

**Note *2:** Any solution such as “process deadlock is a situation where processes can not proceed because the resource(s) a process needs is currently held by another process” is not good enough.
If more than one mutex (binary) semaphore is used, can process deadlock occur? If no, explain why not. If yes, explain how using an example.

YES. If more than one semaphore is used, semaphores can cause a process deadlock can (will) happen in the following way:

1. Assume that Process A is assigned a processor first and it performs “wait” on MUTEX1. A sets MUTEX1 to be ‘0’ and it proceeds.
2. After A sets MUTEX1 to be ‘0’, it is preempted by the operating system (RR scheduling, its processor tie slice expires).
3. Then, the processor is context-switched from A to B.
4. When B starts running, it performs “wait” on MUTEX2. B sets MUTEX2 to be ‘0’ and it proceeds.
5. When B performs “wait” on MUTEX2, it will be blocked at MUTEX2 (since MUTEX2 has been locked by A).
6. Since B is blocked, the operating system context-switches the processor from B to A.
7. A performs “wait” on MUTEX2, which blocks A there.
8. Since A logically waits for B, while B logically waits for A and neither A or B can wake up the other (because they are both sleeping (blocked), a process deadlock occurs here.

Note 1: How a process deadlock can occur needs to be explained (described). Just showing a sample codes using two semaphores (but without an explanation) is not enough for full credit.

Note 2: Although a complete description (as shown above) is not necessary, your explanation (description) should be one that convinces Dr. Fujinoki that you have a correct idea (about how two semaphores can cause a process deadlock) for full credit.
(3) Why is it difficult to eliminate the condition of “non preemptive resources” to prevent a process deadlock from occurring?

It is difficult since some resources, such as printers and DVD-burners, require to be non-preemptive (i.e., making such I/O devices preemptive will cause a serious problems in using them).

(4) In the classroom, we discussed what we can do to make sure one of the four necessary conditions for a deadlock is not satisfied. Is it possible to have a technique that never causes “circular wait”? If yes, describe how.

Yes, it is possible.

The one we discussed in the classroom for eliminating the necessary condition of “circular wait” is as follows.

- First, we assign a unique identification number (“resource ID”) to each resource in a system.

- Then, we require every process to make requests to resources in the same order (either in the ascending order or in the descending order of the resource IDs).

As long as all processes follow the order, the necessary condition of “circular wait” will never be satisfied, eliminating a chance of process deadlocks.

(5) Why is it difficult to eliminate the condition of “hold & wait” (by applying “request all after you drop what all you currently hold” method) to prevent a process deadlock from occurring?

The technique we discussed in the classroom to avoid “hold & wait” is to force (require) every process to drop all resources each process has before each process makes request for addition resource(s). Then, if a requesting process gets all resources it needs, the process gets them all. Otherwise (even one missing), the process gives up them all.

Applying this approach to all processes in a system obviously gives a huge favor to those process that need a small number of resources, resulting starvations to those that need a large number of resources, since the chance of all resources they need gets smaller, while those that need a small number of resources can “steal” resources as soon as a process drops (gives up) them.