(1) As we discussed in the classroom, “threads” are introduced after many system programmers were using “processes” for multi-tasking (we even discussed that “threads” were introduced to avoid two problems in “processes”). After all, while “processes” and “threads” have many things in common (and “threads” seem to be better than “processes”). Then why do we still use “processes” (mention one reason)?

Processes always make sure that contents in a process will never be shared by any other processes, while threads actually let pointers in a process allow any threads in the same process to access anything in the same process (even though they are NOT global variables). This means that threads (in the same process) do not have any privacy. Thus, we can not use threads for any application in which each thread has any data that should never be shared by any other threads in the same process (e.g., representing one (but different) customer using a thread in on-line banking applications).

(2) What are the advantages in the kernel-mode threads?

- Preemptive thread scheduling is possible (it is possible for the operating system to control how long a thread uses a processor).

- The kernel-mode threads are more robust than user (application)-mode threads, because when problems (such as infinite loop or any bugs that stop running a thread) happen, those problems can prevent all other threads in the same process from running (one thread can freeze all other threads in the same process).

(3) How is “process deadlock” different from “(process) starvation”?

Process starvation is essentially a fairness issue (at least some processes remain active), while process deadlock is essentially a reliability issue (when a process deadlock happens, all the processes involved in the deadlock all stop running – preventing all of them from running – thus no progress beyond a process deadlock).
(4) What are the two different types of “process deadlock” (any two out of the three we discussed in the classroom)?

Any two of the following three:

(a) Process deadlock due to blocking calls
(b) Process deadlocks due to semaphores
(c) Process deadlocks due to resource assignments

(5) Why is “process deadlocks due to message passing (blocking calls)” easier to solve than “process deadlocks due to resource deadlock (resource conflict)”?

Process deadlocks due to message passing (blocking calls) are actually a flaw in their design (professionals call such a “flaw” as “bugs”) programs. Programs that use a blocking calls can be easily avoided by having two threads. For example, if a network application process has two threads, one for receiving messages and the other for transmitting messages, whenever the receiving process did not receive any message for longer than it is expected (or deadlock due to blocking calls is suspected), the transmitting thread can transmit a "timer expired" message to another (or other) process(es). This design always prevent deadlocks due to blocking receiving calls.