OS - 2010-2011

1 a) Signals are software interrupts sent to a program to indicate that an important event has occured. They are asynchronous

The kill signal is used to kill a process.
Interrupt character (ctrl-C) interupts a process.

b) The order in which they are released does not matter, so they can be released in any order (mentimeter)

c) No it is not possible. Since the semaphore is initialised to zero, the wait() operation to block the current thread on the condition variable will not occur since the down() on a 0 initialised semaphore will fail meaning that the queue for the condition variable is empty. Consequently, when signal() is called, the semaphore will not be upped since the condition variable queue is empty (due to wait() failing) and hence no shared resource control will occur.

d i) Yes, SRT is a form of priority scheduling in which the process with the smallest amount of time remaining until completion is selected to execute. It is a preemptive version of shortest job next scheduling.

ii) One reason for RR taking longer than FCFS is that the time quantumn could be of too small a value, leading to an increase in over-heated time due to excessive context switching. (If the time quantum is too large RR reduces to FCFS).
A second reason could be that when processes in the queue are interacting with other processes (such as waiting for the completion of another process for data), and are continuously preempted from completing any work until the other process of the queue has finished its run.

iii) In lottery scheduling the proportion of tickets allocated to a process indicates how long a process can use a particular resource for. If all processes are given the same number of tickets then that means that they have equal proportions of time of a resource which can be equated to the time quantum of RR, since the time quantum is the same for all process. It would have to be implemented in such a way that finished process distribute tokens equally in order to make sure the token count per process is equal.

iv) Yes. The number of tokens a particular process has the greater its priority since a greater number of tokens means a greater share of a particular resource. (higher probability of obtaining resource)

Illustrating a case where priority donation occurs:
Consider two tasks H & L, of high & low priority, either of which can acquire use of a shared resource R. If H attempts to acquire R after L has acquired it, then H becomes blocked until L relinquishes (frees) the resource. It is possible that a third task M of medium priority becomes runnable during L's use of R. At this point, M being higher in priority than L, preempts L, causing L to not be able to relinquish R promptly, in turn causing H - the highest priority process to be unable to run. This is called priority inversion and is solved by priority donation (H gives its priority to L).

v) Yes since the new process will have a very high probability of accessing all the relevant resources, thus allowing it to complete.
No since it would block more important processes from using the relevant resources.

Actually I think it's No on second thought since it won't have all the tokens so there is no guarantee that the hard deadline will be met.

2 a i) A monolithic kernel is a single large process running entirely in a single address space. All kernel services exist and execute in the kernel address space. The kernel can invoke functions directly.

In micro kernels, the kernel is broken down into separate processes known as servers. Some of the servers run in kernel space and some run in user space. All servers are kept separate and are run in different address spaces (∴ they are protected from each other) Servers invoke "services" from each other by sending messages via IPC (Inter process communication) This separation has the advantage that if one server fails, other processes can still work efficiently.

Monolithic

+ Fast processing
− less secure – device drives are in kernel space
− difficult to extend (inflexible).

Micro kernel

+ Crash resistant.
+ Portable due to small size.
+ Easily extensible
− slower processing due to message passing

ii) (Not sure about this one).

Monolithic – will crash, need to reboot the whole system.
Micro kernel – driver will crash, not entire system, so driver will reboot

iii) I would choose the micro kernel since it is small, it will not take up too much space in memory (given that smartphone memory is limited) & also because it is extensible so it can easily have bugs fixed.

b i) Blocking system calls are those which must wait for an action to complete before any progress can be made such as when reading from a buffer.

Non-blocking and asynchronous system calls immediately return to the caller (hence allow good resource utilisation) and can occur when polling a resource.

ii) The system call handler selects the read() system call to execute. read() then downs the semaphore on the buffer, which will contain the read content of the file, in order to acquire the resource and then locates where the file to read is located. The device independent OS layer locates the correct device driver which then handles the read request by checking if the device controller, which is connected to the disk containing the file is busy. If not then the request to read the file can be executed and the information is copied into the read buffer. Otherwise the request is added to the ready request queue. Once ~~the read/busy finished~~ all the data has been read, the interrupt handler is called to indicate that the process has finished executing, the buffer semaphore is upped and the scheduler is called to schedule the next ready process.

iii) Busy-waiting wastes system resources as it prevents access to them. A better implementation would be to have a ~~blocked~~ queue of blocked processes.