1a) Inter-process communication mechanisms where process A sends data/signal to process B to indicate that an important event has occurred

* Divide by zero exception
* Process writes to a closed pipe

b) In an increasing order of how long they are expected to take if possible (although technically there needn’t be an order, this would just be a good order to choose)

c) idk, I feel like it should work as it’s basically the same thing? But maybe it being initialised to zero screws with things

Alternative:

I don’t think that it’s same. With monitors, signals cannot accumulate - if a thread isn’t waiting when a signal is sent, the signal is lost. But with the described adjustment, since we sema up instead of calling signal(), the signals are effectively accumulated every time we call signal().

di) It isn’t as there are no priorities that can change depending on length of time run etc., it’s just putting the jobs with the shortest time remaining at the top of the list of processes to run

ii)

* The quantum used may have been too small, which leads to a large overhead.
* Some of the processes could rely on others being completed and so too much pre-emption occurs

iii) Yes, as an equal number of tickets essentially equates to there being a time quantum like in round-robin, which leads to basically the same result

iv) Yes, the “lottery tickets” are essentially a form of priority and can be exchanged between processes to act as priority donation

v)

* Soft - It might not be good as there could potentially be other interactive processes that need to have a quick response time and have now been pushed back
* Hard - yes as it would almost guarantee that the task is completed to fruition

2ai)

* Monolithic: single use executable with its own address space
  + + Efficient calls within kernel
  + + Easier to write kernel components due to shared memory
  + – Complex design with a lot of interactions
  + – No protection between kernel components
* Micro Kernels: minimal kernel with functionality in user-level servers
  + + Not complex so less error-prone
  + + Servers have clean interfaces
  + + Servers can crash and restart without crashing the kernel
  + – Overhead of IPC within kernel is high

ii)

* Monolithic: would just completely crash
* Micro: driver will crash and reboot, system will be fine

iii) micro due to small size and errors will just cause part of the system to crash and reboot as opposed to shutting down the whole phone

bi)

* Blocking: when you need the I/O call to always return when completed
* Non-Blocking: When you need application-level polling for I/O
* Asynchronous: when you need parallel operations and good efficiency

ii)

I think it’s like this: program makes a blocking read system call, the system call handler picks this up, creates a semaphore to wait for the data to be completed, and the thread that made the system call downs this semaphore. System call handler creates a new kernel thread to deal with the I/O stuff, this thread sends the request to a device driver throught the device independent OS layer, then the device driver processes this requests, realizes what it has to do and calls the specific block device we’re reading from via an interrupt handler. The interrupt handler accesses the hardware device controller and tells it what to do, then the device controller feeds back info into the buffer, which when filled notifies the driver, and it goes all the way up the chain until the semaphor ethat was blocking the original thread is upped.

iii) Busy-wait is just wasted CPU time and you may also want your I/O call to return on completion