1. I suck at OSs and here is a testament to that (feel free to change for the benefit of people who deserve the correct answers):
   * 1. Process 1

\_| |\_

| |

Process 2 Process 3

* On a fork the parent process will have the child’s pid returned and the child process will return with pid = 0 on successful creation
  + 1. Process 1 executes all lines (4,5,7,8,9,13,14,17)

Process 2 executes lines 5 to 17 (5,7,11,13,15,17)

Process 3 execute lines 9 to 17 (9,13,15,17)

* ABDDC possible

If we assume that within a process their code is executed synchronously (i.e. in the order it is written) and allow for interleaving between processes

Process 1: A\n C\ne

Process 2: B\n D\n

Process 3: D\n

This leaves us with the following ordering of interleavings to make this output possible:

12321 or 12231

* ADDBC not possible

Again assuming that code for each process is executed synchronously, this combination is not possible as process 2 should output B\n followed by D\n. So for every possible combination of process interleaving this output is not possible

* + 1. If (pid != 0) {

waitpid(pid);

printf(“A\n”);

Pid = fork();

} else { ….

* + 1. I may be wrong but I think this might still allow for mutual exclusion as it leads to the same behaviour in the code, however, something niggles at me about the atomicity of the execution of it.
    2. This no longer permits mutual exclusion as the statement turn == other inside the while always evaluates to false, which makes the entire while condition false and thus no thread will end up holding the CPU and causing another to busy wait. This means that multiple threads can be inside the critical region at the same time thus violating mutual exclusion.
  1. No Longer Examinable
     1. A Disk Block is a unit of memory of fixed size on a physical hardware disk.
     2. A Disk Cylinder is a stack of tracks that are all at the same head position on a disk. (Diagram may be appropriate here)
     3. A Buffer Cache is a form of cached memory, usually implemented by the device independent OS software layer which holds data as it is being transferred to/from a device.
     4. RAID is an array of physical drives appearing as a single virtual drive where the data is distributed over the physical disks to allow for parallel operation/redundancy. (Diagram here as well)
     5. SSTF is the most unfair as it prefers requests that are close together and can end up starving further out requests near the edge of the disk.
     6. C-SCAN is completely fair as it only moves the read/write head in one direction so there is no bias towards requests in the middle..

OR

FCFS as it shows no bias to certain requests and each request will have a wait time proportional to the number of requests in the queue when it joined (therefore if number of requests remains fairly constant so should average wait time)

* + 1. SCAN scheduling serves requests with the shortest seek time moving the head in a constant direction, only changing direction when it either reaches the outermost/innermost track or runs out of requests in that direction.
    2. (Assuming head moves from outermost to innermost track as information given suggests. )

768, 2001, 3593, 8732, 22, 23, 24, 25

* + 1. ( Not really sure but I think maybe ) A request for some PHAT data stored over multiple cylinders.

If the data to be accessed happened to extend across the end of a track, then some of the sectors will be on one cylinder, and some on the next cylinder.

* + 1. Every 10ms would dramatically decrease the throughput as the read/write head will have moved to another request before the consecutive one is scheduled. This is because the seek time is increased and thus the overall access time is increased.

Every 0.1ms gives a much better seek time and access time as newly scheduled requests can be processed in consecutive order, one after the other which gives good throughput.

* + 1. Use an N-Step variation of (C-)SCAN which ignores incoming requests until the following pass.

Could also implement anticipatory scheduling?