CS304 Midsem Solutions

1. A) Mode switch: change CPU execution mode from one privilege level to another e.g., user → kernel via a trap or syscall.
Context switch: save one process' execution context & restore that of another process

B) No!
Key reason: Math functions won't benefit from running in the kernel. They do not need privileged instructions or special facilities. You're putting unwanted stuff in the kernel that has no reason to be there.Moreover: The overhead of calling them: mode switch + data copy is more time-consuming than a function call
NOT: Assembly code is difficult to write/read/debug/etc.

C) ?Startup: created → ready
Preemption: running → ready
I/O complete: blocked → ready

D) Increased overhead due to context switching. Context switching takes time. We'll be doing more of it.

NOT because:
The thread may not finish in one quantum.
The thread may never finish.
Anything to do with thread priorities.

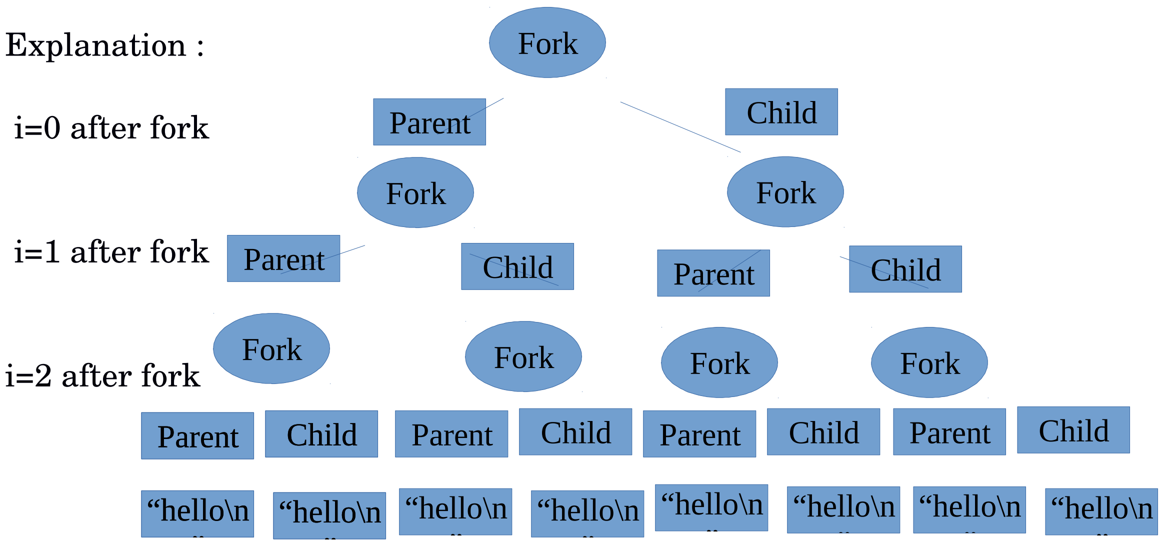
E) Question is about offering sync services via the kernel than via user-level code.
To avoid busy waiting: the waiting thread can go to sleep -> creates better cpu utilization; avoids priority inversion (if an issue)

NOT because:

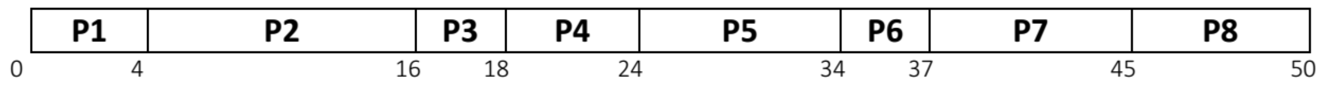
The kernel needs to manage critical sections
Software solutions might be buggy

2. 8 times “hello” will be printed

Explanation :

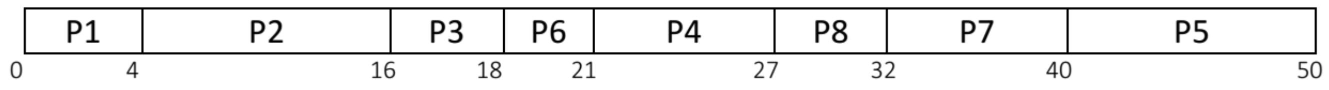


3. **a) FCFS**



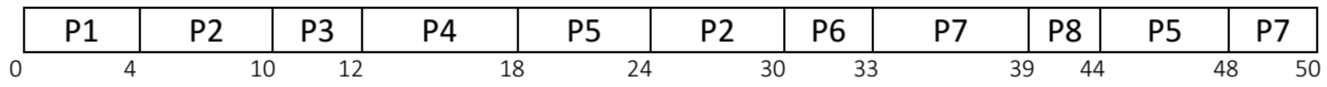
Average waiting time = (0+2+11+12+16+22+22+23)/8 = 108/8 =13.5ms

b) SJF



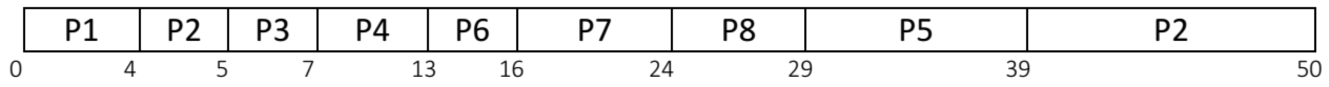
Average waiting time = (0+2+11+15+32+6+17+5)/8 = 88/8 =11ms

c) RR (quantum = 6ms)



Average waiting time = (0+16+5+6+30+18+27+17)/8 = 119/8 =14.875ms

d) SRTF



Average waiting time = (0+36+0+1+21+1+1+2)/8 = 62/8 =7.75ms

4.

A. X= Running. X is only process so it will be scheduled.

B. X = Running. X is highest priority process so it is scheduled.
Y = Ready. Y could be scheduled, but it is lower priority than X.

C. X = Blocked. X is waiting for I/O to complete.
Y = Running. Y is now the only available ready process to schedule.

D. X = Blocked. X is waiting for I/O to complete.
Y = Running. Y is higher priority than Z.
Z = Ready. Z is lower priority than Y.

E. X = Running. X is ready and at higher priority than others.
Y = Ready. Y could be scheduled, but it is lower priority than X.
Z = Ready. Z is lower priority than X.

F . X = X is in terminated state.
Y = Running. Y is highest priority runnable process.
Z = Ready. Z is lower priority than Y.

5.

A. False – each thread has its own stack (specifically its own stack and frame pointer) although the stacks are placed in the same address space.

B. True – since they share an address space, they have the same vpn->ppn translations and the same TLB entries are valid.

C. True – on a uniprocessor, if a thread can't acquire a lock, we'd like the thread holding the lock to have a chance to be scheduled (so it can more quickly release the lock).

D. False – If the thread blocks, it will waste the time of a context-switch; if the thread had just used spin-waiting, it would have wasted less than the time for a context-switch.

E. False – Peterson's algorithm (using a turn variable and an per-thread intention variable) is based entirely on atomic word loads and stores (and is not used in current systems).

F. True; cond_wait() releases the corresponding mutex lock when it is called, but reacquires the lock before it returns (after being signaled).

G. True; producers must have an empty buffer.

H. False; if you use a semaphore for mutual exclusion, it has all the same properties as a traditional lock.

6. A) Performance. Communicating via shared memory is often faster and more efficient since there is no kernel intervention and no copying.

B) Mutual exclusion: there is at most one process that can be inside the critical section;

progress: If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, the selection of the process that will enter the critical section next cannot be postponed indefinitely;
bounded waiting: a bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

7. Note that none of the 3 created threads, A, B, and C, run concurrently with one another! The main thread waits for one to finish (using pthread_join()) before it creates the next thread. So, there is concurrency and no race conditions!
A .200
B. 600

8.

(a) Semaphores variables:

```
pt_waiting = 0
treatment_done = 0
doc_avlbl = 1
```

(b) Patient process:

```
down(doc_avlbl)
consultDoctor()
up(pt_waiting)
down(treatment_done)
noteTreatment()
up(doc_avlbl)
```

(c) Doctor:

```
while(1) {
down(pt_waiting)
treatPatient()
up(treatment_done)
}
```