2. Suppose that you were to design an advanced computer architecture that did process

switching in hardware, instead of having interrupts. What information would the CPU

need? Describe how the hardware process switching might work.

**The CPU would need access to the process table to get the process state, program counter, etc so that it has all information needed to restart a process. When an interrupt occurs, the program counter, program status, and registers are saved. Since interrupts are not used, there is no interrupt service routine and the hardware must save the registers and program counter in a stack used to process the switching. Then it can call the C procedure to execute the next process and the hardware the new process is loaded from memory.**

5. A computer system has enough room to hold five programs in its main memory. These

programs are idle waiting for I/O half the time. What fraction of the CPU time is

wasted?

**CPU utilization = 1 – p^n = 1 – (0.5)^5 = 1 – 0.03125 = 0.96875. The CPU is used 96.875% of the time and 3.125% is wasted time.**

7. Multiple jobs can run in parallel and finish faster than if they had run sequentially.

Suppose that two jobs, each needing 20 minutes of CPU time, start simultaneously.

How long will the last one take to complete if they run sequentially? How long if they

run in parallel? Assume 50% I/O wait.

**It will take 40 minutes (20 + 20) if they run sequentially, and 25 minutes (20 + (20 \* 0.25)) if they run in parallel. 1-(0.5)^2 = 1- 0.25 = 0.75 utilization, 25% waste.**

12. In Fig. 2-8, a multithreaded Web server is shown. If the only way to read from a file is

the normal blocking read system call, do you think user-level threads or kernel-level

threads are being used for the Web server? Why?

**Kernel level threads are used because user-level threads need to check if the blocking read system call is safe which leads to constant system calls. The kernel can switch threads far easier than the user-level threads.**

13. In the text, we described a multithreaded Web server, showing why it is better than a

single-threaded server and a finite-state machine server. Are there any circumstances in

which a single-threaded server might be better? Give an example.

**Yes, if the web server runs as a single thread, the CPU runs on a dedicated machine and the CPU is idle when the server waits for the disk and the main loop of the server handles each request one at a time. This is a better when there are no asynchronous calls because then nothing needs to be blocked and the server can perform faster.**

16. Can a thread ever be preempted by a clock interrupt? If so, under what circumstances?

If not, why not?

**Yes it can. If a thread is blocked, it needs an external event to occur. A thread to perform a certain task can wait until the clock interrupt occurs and then become unblocked and then continue running.**

18. What is the biggest advantage of implementing threads in user space? What is the

biggest disadvantage?

**By implementing threads in user space, the operating system that does not support threads can implement a user-level threads package and these threads can be implemented by a library. One disadvantage is in the way that blocking system calls are implemented. When a thread needs to make a blocking system call, it should prevent the blocked thread from affecting other threads. However, when one thread makes a blocking call, it should not block other threads.**

24. Does Peterson’s solution to the mutual-exclusion problem shown in Fig. 2-24 work

when process scheduling is preemptive? How about when it is nonpreemptive?

**When the process scheduling is preemptive, it is cooperative and mutual exclusion succeeds due to all processes needing to call *enter\_region* before entering their critical region. The scheduler will ensure that the process stops after a certain amount of time. If they are non-preemptive, mutual exclusion can still succeed, but only if none of the processes last indefinitely. A non-preemptive scheduler does not stop the process, so there are cases when the mutual-exclusion problem will not be solved.**

26. In Sec. 2.3.4, a situation with a high-priority process, *H*, and a low-priority process, *L*,

was described, which led to *H* looping forever. Does the same problem occur if roundrobin

scheduling is used instead of priority scheduling? Discuss.

**In round robin scheduling, each process gets an allotted amount of CPU time before the CPU moves to another process. If the process did not complete, then it will be returned to after the other processes have their time. This will avoid the problem with the priority scheduling because now instead of L being locked into its critical region and H hogging the CPU, the CPU will give time to L to finish before returning to H.**

27. In a system with threads, is there one stack per thread or one stack per process when

user-level threads are used? What about when kernel-level threads are used? Explain.

**There is one stack per thread because each individual thread has its own memory, variables, and other resources so sharing a stack would not work with multiple threads. This is true for both user-level and for kernel-level threads.**

29. The producer-consumer problem can be extended to a system with multiple producers and consumers that write (or read) to (from) one shared buffer. Assume that each producer and consumer runs in its own thread. Will the solution presented in Fig. 2-28, using semaphores, work for this system?

**Yes the solution will work because mutual exclusion is guaranteed if binary semaphores are used. This way, interrupts can be hidden and run again while the scheduler is busy with another process. A mutex semaphore guarantees that only one process can read or write from the shared buffer at a time. This mutual exclusions keeps order and prevents overwriting.**

30. Consider the following solution to the mutual-exclusion problem involving two processes

*P0* and *P1*. Assume that the variable turn is initialized to 0. Process *P0*’s code is

presented below.

/\* Other code \*/

while (turn != 0){}/\*Do nothing and wait. \*/

Critical Section /\* . . . \*/

turn=0;

/\* Other code \*/

For process *P1*, replace 0 by 1 in above code. Determine if the solution meets *all* the

required conditions for a correct mutual-exclusion solution.

**If no other code effects turn, then this does satisfy mutual exclusion. This is because neither process locks the other one out because the critical section for P0 will only be entered when P1 exits its critical section (the same is true for P1 based on P0 exiting). Additionally, both cannot be in their critical sections at the same time due to the while loops holding the process until turn changes.**

34. Can two threads in the same process synchronize using a kernel semaphore if the

threads are implemented by the kernel? What if they are implemented in user space?

Assume that no threads in any other processes have access to the semaphore. Discuss

your answers.

**If the threads are implemented by the kernel then they can share a kernel semaphore, but the efficiency of sharing the kernel semaphore is not close to the efficiency of user-level threads. User level threads have no problem sharing the same semaphore. If the semaphore is stored in the kernel, then it is only accessed through system calls and the threads can share variables and a common buffer.**

37. Suppose that we have a message-passing system using mailboxes. When sending to a

full mailbox or trying to receive from an empty one, a process does not block. Instead,

it gets an error code back. The process responds to the error code by just trying again,

over and over, until it succeeds. Does this scheme lead to race conditions?

**This scheme does not lead to race conditions because it simply responds with a message. When that error message does not return, then the process assumes the message was sent. The only time there could be a race condition is if the process is waiting for a response and none is returned, but that is the condition for success so there would be no race condition.**

42. Explain how time quantum value and context switching time affect each other, in a

round-robin scheduling algorithm.

**Quantum time is the time interval a process is allowed to run for, and at the end of that time, the CPU is given a new process to run. The context switching time is the amount of time it takes to load the new process and save the resources from the previous process. These two times determine the amount of CPU waste time where the CPU is not doing useful work and is instead doing administration work (switching processes). The smaller the waste time, the greater the CPU efficiency.**

45. Five batch jobs. *A* through *E*, arrive at a computer center at almost the same time.

They hav e estimated running times of 10, 6, 2, 4, and 8 minutes. Their (externally determined)

priorities are 3, 5, 2, 1, and 4, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the mean process turnaround time. Ignore process switching overhead.

(a) Round robin.

(b) Priority scheduling.

(c) First-come, first-served (run in order 10, 6, 2, 4, 8).

(d) Shortest job first.

For (a), assume that the system is multiprogrammed, and that each job gets its fair

share of the CPU. For (b) through (d), assume that only one job at a time runs, until it

finishes. All jobs are completely CPU bound.

**A)**

**B) 4 + 6 + 16 + 24 + 30 = 90/5 = 18 minutes**

**C) 10 + 16 + 18 + 22 + 30 = 96/5 = 19.2 minutes**

**D) 2 + 6 + 12 + 20 + 30 = 70/5 = 14 minutes**

50. A soft real-time system has four periodic events with periods of 50, 100, 200, and 250

msec each. Suppose that the four events require 35, 20, 10, and *x* msec of CPU time,

respectively. What is the largest value of *x* for which the system is schedulable?

**Sum(c\_i / p\_i) <= 1… (35/50)+(20/100)+(10/200) +(x/250) <= 1…**

**7/10 + 2/10 + 0.5/10 = 9.5/10 + x/250 <= 1**

**x/250 = 0.05… x = 12.5msec**

51. In the dining philosophers problem, let the following protocol be used: An even-numbered

philosopher always picks up his left fork before picking up his right fork; an odd-numbered philosopher always

picks up his right fork before picking up his left fork. Will this protocol guarantee deadlock-free operation?

**This solution does not deadlock, but it causes problems, especially if there are an odd number of philosophers. If there are an odd number of philosophers, the first pairs of philosophers will reach for the same fork at the same time, but the last philosopher will not and will always get their forks and therefore will have more time to eat. The second concern is that when two philosophers reach for the same fork, one will be unable to eat since they do not have a fork. When their neighbor stops, there is no guarantee that the philosopher who didn’t eat will be able to pick up the fork before that philosopher picks it up to begin eating again.**