Laboratory Exercise 3 Solutions

cpe 453 Winter 2023

Due by 11:59:59pm, Monday, February 13th.
The Written Exercises (problems) are to be done individually.
No late days will be allowed on this lab due to the upcoming midterm.

Problems

1. In MINIX (or any other UNIX), if user 2 links to a file owned by user 1, then user 1 removes the file, what happens when user 2 tries to read the file? (Tanenbaum and Woodhull, Ch. 1, Ex. 15)

   Solution:
   User 2's link to the file remains intact, exactly as it was before. User 1 remains the owner of the file, however, and any access rights are exactly as before.

2. Under what circumstances is multiprogramming likely to increase CPU utilization? Why?

   Solution:
   Multiprogramming will increase CPU utilization any time the CPU is likely to have to wait longer than the amount of time it would take to switch to another task. This tends to occur when a process will have to be performing IO. Consider the situation where process A has to block. If it takes less time to switch to process B than A would have to wait, the context switch is a net win.

3. Suppose a computer can execute 1 billion instructions/sec and that a system call takes 1000 instructions, including the trap and all the context switching. How many system calls can the computer execute per second and still have half the CPU capacity for running application code? (T&W 1-21)

   Solution:
   Half of 1 billion instructions can be devoted to system calls:

   $$500\,000\,000 \ \textit{cycles} \cdot \frac{1 \ \textit{syscall}}{1000 \ \textit{cycles}} = 500\,000 \ \textit{syscalls}$$

4. What is a *race condition?* What are the symptoms of a race condition?(T&W 2-9)

   Solution:
   A race condition is any situation where the precise ordering of a series events affects the outcome of the entire process. The term is usually only applied where processes are reading or writing some shared data.
   Race conditions manifest themselves as nondeterministic behavior (usually leading to strange behavior at inopportune times).

5. Does the busy waiting solution using the *turn* variable (Fig. 2-10 in T&W) work when the two processes are running on a shared-memory multiprocessor, that is, two CPUs, sharing a common memory? (T&W, 2-13)

Solution:

Yes, it will still work. The change from pseudoparallelism to true parallelism—a situation where both processes really can be trying to access the turn *variable at the same time—doesn't affect the correctness of this solution. There is still no race condition, so it is still safe.*

It is also still busywaiting.

6. Describe how an operating system that can disable interrupts could implement semaphores. That is, what steps would have to happen in which order to implement the semaphore operations safely. (T&W, 2-10)

Solution:

To implement a semaphore:

(a) The operating system will disable interrupts.

(b) Then what it does depends on the operation:

- DOWN(), *counter is non-zero*
  The OS will decrement the counter and go to step 6c.
- DOWN(), *counter is zero*
  The calling process will be suspended and placed on a list of processes waiting for this particular semaphore, then the OS will select another runnable process to run and continue on to step 6c.
- UP(), *counter is non-zero*
  Since the counter is nonzero, there can be no processes waiting. The OS will increment the counter and go to step 6c.
- UP(), *counter is zero*
  If there are processes waiting, the OS will select one of them and place it on the queue of runnable processes and leave the counter at zero. If there are no processes waiting for this semaphore, the OS will increment the counter. Either way, it will pick a process to run and continue on to step 6c.

(c) The OS will re-enable interrupts, and return to the process currently selected to run, or the idle process if there are no runnable processes.

What makes this procedure different from user-level processes doing this on their own is that the OS is never suspended, only user processes.

7. Round robin schedulers normally maintain a list of all runnable processes, with each process occurring exactly once in the list. What would happen if a process occurred twice in the list? Can you think of any reason for allowing this? (T&W, 2-25) (And what is the reason. "Yes" or "no" would not be considered a sufficient answer.)

   Solution:

   If one process occured twice in the list, it would get scheduled twice every time the scheduler passed through the queue. The result of this is that the duplicated process would get a double share of CPU time. Allowing duplication could provide a simple mechanism for giving different processes or users an unequal share of the CPU.

8. Five batch jobs, A through E, arrive at a computer center, in alphabetical order, at almost the same time. They have estimated running times of 10, 3, 4, 7, and 6 seconds respectively. 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 time at which each job completes and the mean process turnaround time. Assume a 1 second quantum and ignore process switching overhead. (Modified from T&W, 2-28)

   (a) Round robin.

   Solution:

   The detail of the scheduling is in Figure 1.

Process	**End Time**
A	30s
B	12s
C	17s
D	27s
E	25s

Mean	22.2s

   (b) Priority scheduling.

   Solution:

   The detail of the scheduling is in Figure 2.

Process	**End Time**
A	19s
B	3s
C	23s
D	30s
E	9s

Mean	16.8s

   (c) First-come, first served (given that they arrive in alphabetical order).

   Solution:

   The detail of the scheduling is in Figure 3.

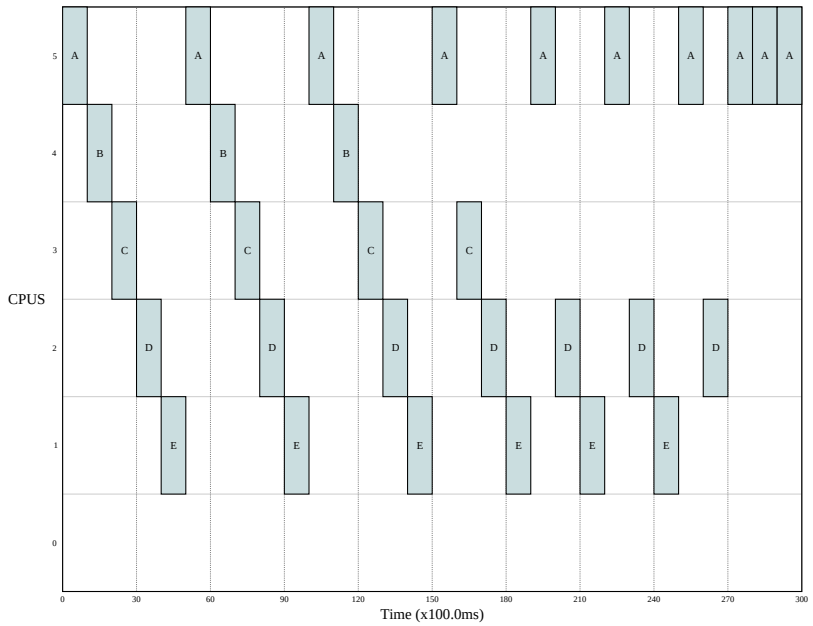

Figure 1: More scheduling detail for problem 8a.

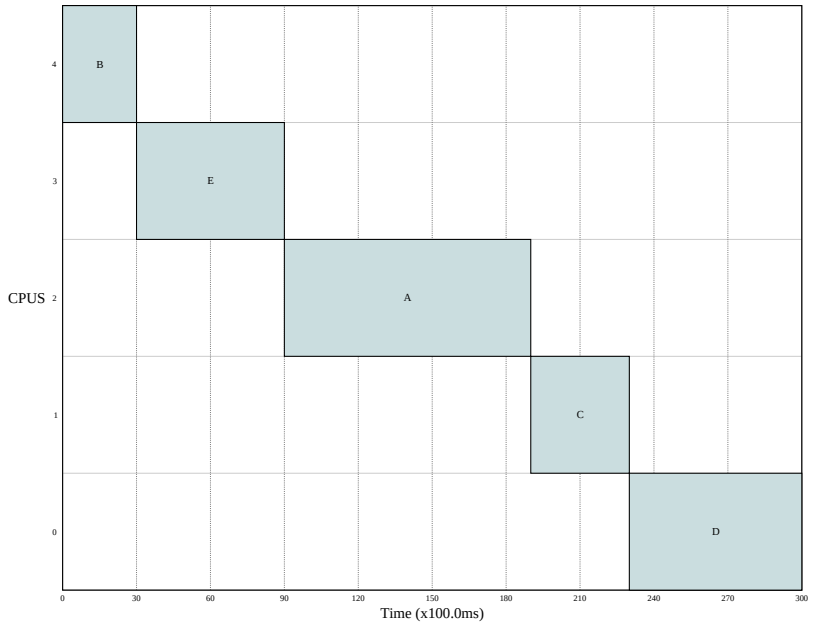

Figure 2: More scheduling detail for problem 8b.

Process	End Time
A	10s
B	13s
C	17s
D	24s
E	30s

Mean	18.8s

(d) Shortest job first.

Solution:

The detail of the scheduling is in Figure 4.

Process	End Time
A	30s
B	3s
C	7s
D	20s
E	13s

Mean	14.6s

for (a), assume that the system is multiprogrammed, and that each job gets its fair share of the CPU. For (b)–(d) assume that only one job at a time runs, and each job runs until it finished. All jobs are completely CPU bound.

9. Re-do problem 8a with the modification that job D is IO bound. After each 500ms it is allowed to run, it blocks for an IO operation that takes 1s to complete. The IO processing itself doesn't take any noticeable time. Assume that jobs moving from the blocked state to the ready state are placed at the end of the run queue. If a blocked job becomes runnable at the same time a running process's quantum is up, the formerly blocked job is placed back on the queue ahead of the other one.

Solution:

The detail of the scheduling is in Figure 5.

Process	End Time
A	27.5s
B	11.0s
C	15.5s
D	34.0s
E	22.0s

Mean	22.0s

10. A CPU-bound process running on CTSS needs 30 quanta to complete. How many times must it be swapped in, including the first time (before it has run at all)? Assume that there are always other runnable jobs and that the number of priority classes is unlimited. (T&W, 2-29)

Solution:

If there are unlimited priority classes, it must be swapped-in 5 times.

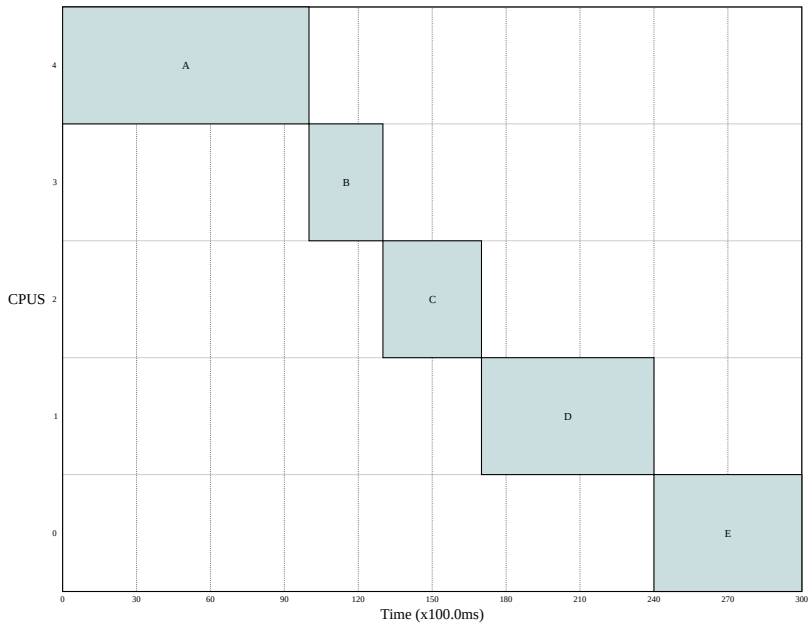

Figure 3: More scheduling detail for problem 8c.

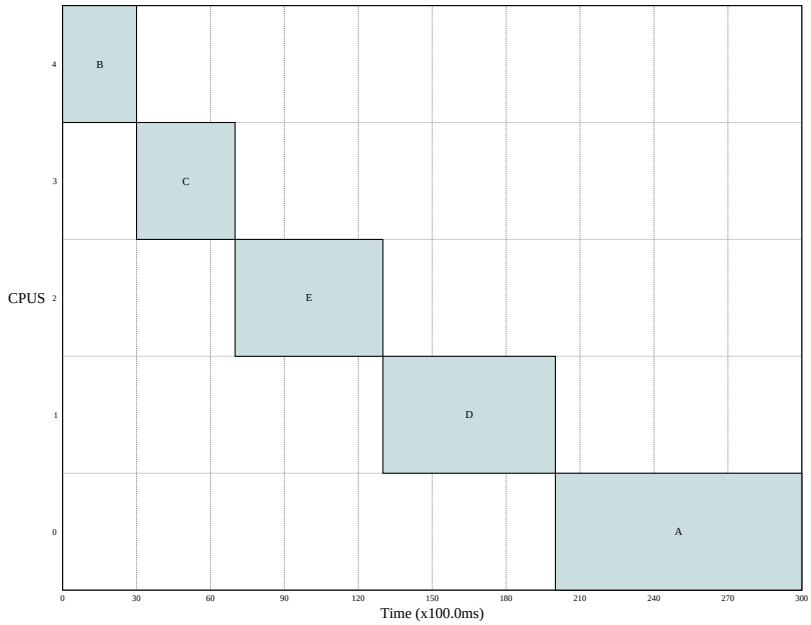

Figure 4: More scheduling detail for problem 8d.

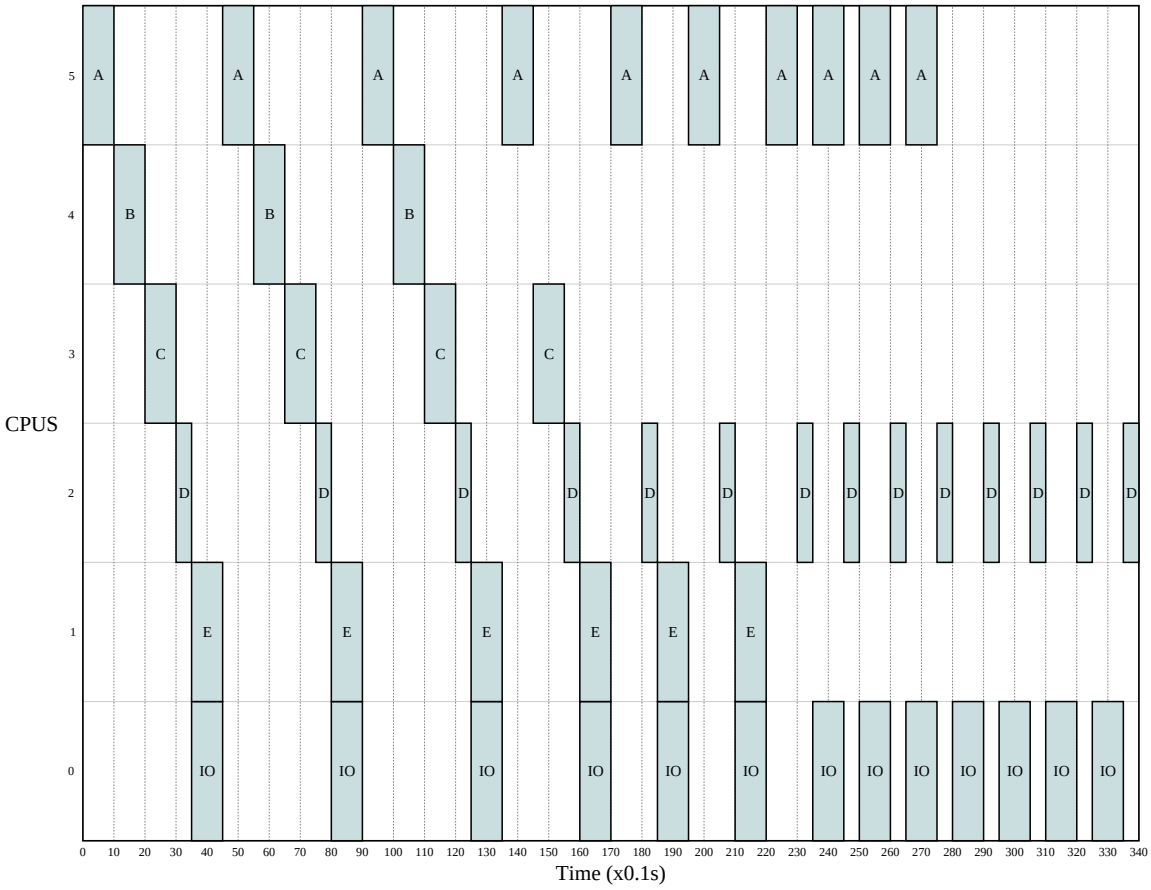

Figure 5: More scheduling detail for problem 9.

Swap	Quanta	Total
1	*1*	*1*
2	*2*	*3*
3	*4*	*7*
4	*8*	*15*
5	*16 (15 used)*	*30*

What to turn in

For the Written Problems: individually written solutions to the problems according to the guidelines set forth in the syllabus.

Since paper won't work this quarter, submit via `handin` to the `lab03` subdirectory of the `pn-cs453` account, as a pdf or text file.[1]

[1] I only have LibreOffice, and you don't want to see what it'll do to your nicely crafted Word document...