Lab 03 – Midterm Problem Set – Ben Soto (bsoto07)

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)

When calling link, a new entry into the inodes list occurs, making the two 'files' simply pointers to the same file descriptor for a file. When one gets removed, the syscall unlink is called and it will just remove the pointer to that FD described, and delete it from the inodes list IF no other links point to that file. Thus, in this example, user 2 would be able to read the file just fine.

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

Multiprogramming will increase CPU utilization in many ways, but primarily because of context switching. Since in multiprogramming, we have multiple processes running concurrently, being able to swap between them (during mem reads or other long actions) adds processing overhead that will increase CPU utilization.

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)

1 000 000 000 instr/sec

1 000 instr/syscall

want 1/2 of the CPU, 1/2 CPU:

500 000 000 instr/sec

500 000 000 / 1 000 =

500 000 syscall/sec

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

A race condition is the event where two or more tasks attempt to access a shared resource at the same time. This is unfortunate if one or more of the tasks desire to change or update the shared resource because we might corrupt our data or see undefined actions occuring to it. For example, if two processes want to edit a very long array, then print it out, one process could be halfway done editing it, then the other begins editing it, so when the first goes to print, it will have wrong data from the second process.

Some symptoms include: data corruption, undefined behavior, debugging nightmares

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)

Yes. This turn approach utilizes strict alternation which works because it will explicitly state whose turn it is for the critical section. However, a turn based approach that uses a boolean to see if 'you' own the 'turn' would not work since multiple people could try to get that bool at the same time and cause issues.

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)

One way to implement semaphores if we are able to disable interrupts would be to first disable interrupts, then grab the semaphore, then re-enable interrupts. This order works because the worry of an interrupt is getting interrupted while attempting to grab the semaphore, but if we just turn interrupts off before grabbing it, then we should be fine.

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.)

The round robin scheduler should keep just one copy of each process in its list because it utilizes a circular list which gives everyone a turn fairly. If you had more than one entry for the same process, then that process would get extra CPU time, mkaing it unfair. You may want to allow multiple entries of the same process in the RR scheduler if you wanted to give it more time explicitly since this guarantees that the same process will get poked at least one time more than others.

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)

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.

(a) Round robin.

[process name] [time left in sec] – [total time]

A 9 – 1

B 2 – 2

C 3 – 3

D 6 – 4

E 5 – 5

A 8 – 6

B 1 – 7

C 2 – 8

D 5 – 9

E 4 – 10

A 7 – 11

B 0 – 12 **B finish @ 12 sec**

C 1 – 13

D 4 – 14

E 3 – 15

A 6 – 16

C 0 – 17 **C finish @ 17 sec**

D 3 – 18

E 2 – 19

A 5 – 20

D 2 – 21

E 1 – 22

A 4 – 23

D 1 – 24

E 0 – 25 **E finish @ 25 sec**

A 3 – 26

D 0 – 27 **D finish @ 27 sec**

A 2 – 28

A 1 – 29

A 0 – 30 **A finish @ 30 sec**

Mean Turnaround:

30 + 12 + 17 + 27 + 25 = 111 sec

111 / 5 =

22.2 seconds

(b) Priority scheduling.

10, 3, 4, 7, and 6 seconds respectively. Their (externally determined) priorities are 3, 5, 2, 1, and 4,

- B finishes after 3 seconds
- E finishes after 9 seconds
- A finishes after 19 seconds
- C finishes after 23 seconds
- D finishes after 30 seconds

Mean Turnaround:

3 + 9 + 19 + 23 + 30 = 84 seconds

84 / 5 =

16.8 seconds

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

- A finishes after 10 seconds
- B finishes after 13 seconds
- C finishes after 17 seconds
- D finishes after 24 seconds
- E finishes after 30 seconds

Mean Turnaround:

10 + 13 + 17 + 24 + 30 = 94

94 / 5 =

18.8 seconds

(d) Shortest job first.

10, 3, 4, 7, and 6 seconds respectively

- B finishes after 3 seconds
- C finishes after 7 seconds
- E finishes after 13 seconds
- D finishes after 20 seconds
- A finishes after 30 seconds

Mean Turnaround:

3 + 7 + 13 + 20 + 30 = 73 seconds

73 / 5 =

14.6 seconds

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.

They have estimated running times of 10, 3, 4, 7, and 6 seconds respectively.

MY INTERPRETATION OF THIS QUESTION:

- still using RR scheduler
- when process D gets its turn, it only uses 0.5 sec of the quantum before blocking and yielding to next
- when it yields, it itself is blocked for 1 sec
- since the quantum for each turn is 1 sec, it will not move around (really at all) in the queue, but it will just change the end timing of the others (sooner) and itself take longer

I have re-read this question many times and I am able to come up with a few interpretations of it, but this was the one I spent a long time working thru so hopefully it suffices.

[process name] [time left in sec] – [total time]

A 9 – 1

B 2 – 2

C 3 – 3

D 6.5 – 3.5 Blocked for 1 sec

E 5 – 4.5 D is now back in queue ahead of E

A 8 – 5.5

B 1 – 6.5

C 2 – 7.5

D 6 – 8 Blocked for 1 sec

E 4 – 9 D is now back in queue ahead of E

A 7 – 10

B 0 – 11 **B finish @ 11 sec**

C 1 – 12

D 5.5 – 12.5 Blocked for 1 sec

E 3 – 13.5 D is now back in queue ahead of E

A 6 – 14.5

C 0 – 15.5 **C finish @ 15.5 sec**

D 5 – 16 Blocked for 1 sec

E 2 – 17 D is now back in queue ahead of E

A 5 – 18

D 4.5 – 18.5 Blocked for 1 sec

E 1 – 19.5 D is now back in queue ahead of E

A 4 – 20.5

D 4 – 21 Blocked for 1 sec

E 0 – 22 **E finish @ 22 sec**

A 3 – 23

D 3.5 – 23.5 Blocked for 1 sec

A 2 – 24.5 D is now back in queue ahead of A

D 3 – 25 Blocked for 1 sec

A 1 – 26 D is now back in queue ahead of A

D 2.5 – 26.5 Blocked for 1 sec

A 0 – 27.5 **A finish @ 27.5 sec**

D 2 – 28.5 Blocked for 1 sec

D 1.5 – 30 Blocked for 1 sec

D 1 – 31.5 Blocked for 1 sec

D 0.5 – 33 Blocked for 1 sec

D 0 – 34.5 Blocked for 1 sec **D finish @ 34.5 sec**

Mean Turnaround:

27.5 + 11 + 15.5 + 34.5 + 22 = 110.5 seconds

110.5 / 5 =

22.1 seconds

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)

Assuming quanta increases by 2^n where n is the # of swaps:

1 – 1 quanta 3 – 2 quanta 7 – 4 quanta 15 – 8 quanta 31 – 16 quanta

this will take 5 swaps