CSC 453 Lab 03

August Hogen-Esch (ahogenes)

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?**

When user 2 tries to read a file that user 1 has removed after user 2 created a hard link to it, user 2 will still successfully read the file. This is because UNIX-like systems use an inode to store file data and metadata, and they maintain a link count of the directory entries pointing to that inode. When user 1 creates the file, the link count is 1; when user 2 links to it, the count is incremented to 2. User 1's "removal" only deletes their directory entry and decrements the link count back to 1. The operating system only permanently deletes the file's data and frees the inode when the link count reaches zero, so as long as user 2's link exists, the file remains accessible.

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

Multiprogramming significantly increases CPU utilization when the running processes are I/O-bound or a mix of I/O-bound and CPU-bound. I/O-bound processes frequently pause execution to wait for slow I/O operations (like disk reads). The benefit of multiprogramming is that the operating system can perform a context switch and hand the CPU to another ready process while the first process is blocked waiting for I/O. This mechanism ensures the CPU stays busy executing useful work instead of idling, thus covering the time that would otherwise be wasted waiting for I/O to complete.

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)**

To still have half the CPU capacity for running application code, a maximum of 500 million instructions/sec can be used for system calls. If a system call takes 1000 instructions, then

$$\frac{500,000,000 \text{ (instructions/sec)}}{1{,}000 \text{ (instructions/system call)}} = \textbf{500,000} \text{ (system calls / second)}$$

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

A race condition occurs when the outcome of an operation involving shared data depends unpredictably on the relative order and timing of multiple processes or threads accessing or manipulating that data concurrently. The core symptom is non-deterministic, inconsistent, or corrupted data, meaning the system may work correctly most of the time but

occasionally produces incorrect results. A process might read a value and then be preempted, allowing another process to modify that value. When the first process resumes, it overwrites the new value with a calculation based on the stale value it read earlier, leading to corrupted data.

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)**

The busy waiting solution using the turn variable fails on a shared-memory multiprocessor system. This solution relies on processes strictly taking turns by checking the turn variable. However, with two processes running truly simultaneously on separate CPUs, a faster process can complete its critical section, set the turn variable to allow the other process to run, and immediately find the turn variable set back by the slower process (which may have been preempted or simply working more slowly). The faster process can then re-enter its critical section immediately before the slower process ever gets a chance to enter, violating the mutual exclusion requirement and the intended alternation. Busy waiting is overly restrictive and fails when true concurrency is implemented.

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)**

An operating system can safely implement semaphore operations by using the ability to disable interrupts to ensure that the code manipulating the semaphore's internal state is atomic. Disabling interrupts prevents the CPU from initiating a context switch, guaranteeing that no other process can interfere while the counter and waiting list are being modified. For the DOWN (or wait) operation, the process first disables interrupts, then atomically decrements the semaphore counter if the counter was positive. If the counter is zero, (indicating the resource is unavailable), the process is placed onto the semaphore's waiting queue, and the scheduler is called, re-enabling interrupts. If the counter was successfully decremented, the process proceeds immediately after re-enabling interrupts.

Similarly, the UP (or signal) operation begins by disabling interrupts to protect the shared data. The semaphore counter is then incremented. If any processes were waiting, one of the blocked processes is moved from the semaphore's waiting queue to the ready queue. After checking and unblocking a process if necessary, interrupts are re-enabled, and the calling process continues its execution. This interrupt-disabling mechanism is essential for preserving the integrity and correctness of the semaphore operations in a concurrent environment.

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.)**

If a process were to appear twice in a Round Robin scheduler's ready list, it would receive double the CPU time in each full rotation of the queue. A Round Robin scheduler cycles through the list, giving one time quantum to each process in sequence. If process A is listed at two different positions, it will run for one quantum when the scheduler reaches the first position and for a second quantum when it reaches the second position. Consequently, the process's effective share of the CPU bandwidth is doubled compared to processes listed only once

The primary reason for allowing a process to occur multiple times in the ready list is to implement a simple priority scheme (or weighted fair sharing) without modifying the core Round Robin algorithm. By listing a critical process N times, the operating system effectively assigns a higher weight or priority to that job. This simple mechanism ensures the high-priority task receives N quanta for every single quantum given to a standard process, dedicating a larger percentage of the total CPU time to jobs with strict latency requirements or critical system services.

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.
   (b) Priority scheduling.
   (c) First-come, first served (given that they arrive in alphabetical order).
   (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)–(d) assume that only one job at a time runs, and each job runs until it finished. All jobs are completely CPU bound.

Job	Arrival Order (FCFS)	Estimated Run Time (s)	Priority (5 is highest)
A	1st	10	3
B	2nd	3	5
C	3rd	4	2
D	4th	7	1
E	5th	6	4

a) Round Robin

Time (s)	Job Run	Remaining Run Time	Event
0–5	A, B, C, D, E	A=9, B=2, C=3, D=6, E=5	Cycle 1 completed
5–10	A, B, C, D, E	A=8, B=1, C=2, D=5, E=4	Cycle 2 completed
10–11	A	A=7	A runs
11–12	B	B=0	B Completes at 12s
12–15	C, D, E	C=1, D=4, E=3	C, D, E run
15–16	A	A=6	A runs
16–17	C	C=0	C Completes at 17s
17–19	D, E,	D=3, E=2,	D, E,
19–22	A, D, E	A = 5, D=2, E=1	A, D, E, run
22–25	A, D, E	A=4, D=1, E=0	E Completes at 25s
25–27	A, D	A = 3, D = 0	D Completes at 27s
27–30	A, A, A,	A=2, 1, 0	A runs for its remaining 3s and Completes at 30s

Mean Turnaround time = (12 + 17 + 25 + 27 + 30) / 5 = **22.2s**

b) Priority Scheduling (Non-Preemptive, 5 is highest)

Jobs run until they finish. The execution order is determined by priority: B (5), E (4), A (3), C (2), D (1).

Completion time = start time + run time = turnaround time

Job	Run Time (s)	Start Time (s)	Completion Time (s)	Turnaround Time (s)
B	3	0	3	3
E	6	3	9 (3 + 6)	9

A	10	9	19 (9 + 10)	19
C	4	19	23 (19 + 4)	23
D	7	23	30 (23 + 7)	30

Mean turnaround time = (3 + 9 + 19 + 23 + 30) / 5 = **16.8s**

c) First-Come, First Served (Non-Preemptive)

Jobs run in arrival order: A, B, C, D, E. One at a time until they finish

Job	Run Time (s)	Start Time (s)	Completion Time (s)	Turnaround Time (s)
A	10	0	10	10
B	3	10	13 (10 + 3)	13
C	4	13	17 (13 + 4)	17
D	7	17	24 (17 + 7)	24
E	6	24	30 (24 + 6)	30

Mean turnaround time: (10 + 13 + 17 + 24 + 30) / 5 = 18.8

d) Shortest Job First (Non-Preemptive)

Jobs run until they finish. Execution order is determined by shortest run time: B (3), C (4), E (6), D (7), A (10).

Job	Run Time (s)	Start Time (s)	Completion Time (s)	Turnaround Time (s)
B	3	0	3	3
C	4	3	7 (3 + 4)	7
E	6	7	13 (7 + 6)	13
D	7	13	20 (13 + 7)	20
A	10	20	30 (20 + 10)	30

Mean Turnaround time = (3 + 7 + 13 + 20 + 30) / 5 = 14.6

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.

Time (s)	Process	Remaining Run Time (s)	Blocked Job (I/O Finish Time (s))	Ready Queue (A, B, C, D, E initial order)	Event
0	-	A=10, B=3, C=4, D=7, E=6	-	A, B, C, D, E	Start
0–3	A, B, C	A=9, B=2, C=3	-	D, E, A, B, C	A, B, C run 1s each.
3–3.5	D	D=6.5	D (4.5)	E, A, B, C	D runs 0.5s; blocks (I/O finishes at 4.5s)
3.5–4.5	E	E=5	D (4.5)	A, B, C, E	E runs 1s.
4.5–5.5	A	A=8	-	D, B, C, E, A	T=4.5: D ready. Placed ahead of A. A runs 1s.
5.5–6.0	D	D=6	D (7.0)	B, C, E, A	D runs 0.5s; blocks (I/O finishes at 7.5s)
6.0–7.0	B	B=1	D (7.0)	D, C, E, A, B	B runs 1s. I/O finished for D and is moved to front of queue
7.0–7.5	D	D=5.5	D (8.5)	C, E, A, B	T=7.0: D runs 0.5s; blocks (I/O finishes at 8.5s).
7.5–8.5	C	C=2	D (8.5)	E, A, B, C	C runs 1s.
8.5–9.0	D	D=5	D (10.0)	D, E, A, B, C	T=8.5: D ready. Placed ahead of C. D runs 0.5s; blocks.
9.0–10.0	E	E=4	D (10.0)	A, B, C, E	E runs 1s.
10.0–10.5	D	D=4.5	D (11.5)	D, A, B, C, E	T=10.0: D ready. Placed ahead of E. D runs 0.5s; blocks.

10.5–11.5	A	A=7	D (13.5)	B, C, E, A	A runs 1s.
11.5–12.0	D	D=4	D (15.5)	D, B, C, E, A	T=13.5: D ready. Placed ahead of A. D runs 0.5s; blocks.
12-13	B	B =0	D (15.5)	C, E, A,	B finishes at 13 seconds
13.0–14.0	C	C=1	D (15.5)	E, A, C	C runs 1s .
14.0–14.5	D	D=3.5	D (17.5)	D, E, A, C	T=15.5: D ready. Placed ahead of C. D runs 0.5s; blocks.
14.5–15.5	E	E=3	D (17.5)	A, C, E	E runs 1s.
15.5–16.0	D	D=3	D (19.5)	D, A, C, E	T=17.5: D ready. Placed ahead of E. D runs 0.5s; blocks.
16.0–17.0	A	A=6	D (19.5)	C, E, A	A runs 1s.
17.0–17.5	D	D=2.5	D (21.5)	D, C, E, A	T=19.5: D ready. Placed ahead of A. D runs 0.5s; blocks.
17.5–18.5	C	C=0	D (21.5)	E, A	C runs 1s. C completes at 18.5s (4s run time).
18.5–19.0	D	D=2	D (23.5)	D, E, A	T=21.5: D ready. Placed ahead of E. D runs 0.5s; blocks.
19.0–20.0	E	E=2	D (23.5)	A, E	E runs 1s.
20.0–20.5	D	D=1.5	D (25.5)	D, A, E	T=23.5: D ready. Placed ahead of E. D runs 0.5s; blocks.
20.5-21.5	A	A=5	D (25.5)	E, A	A runs 1s.
21.5–22.0	D	D=1	D (27.5)	D, E, A	T=25.5: D ready. Placed ahead of A. D runs 0.5s; blocks.
22.0–23.0	E	E=1	D (27.5)	A, E	E runs 1s.

23.0–23.5	D	D=0.5	D (29.5)	D, A, E	T=27.5: D ready. Placed ahead of E. D runs 0.5s; blocks.
23.5–24.5	A	A=4	D (29.5)	E, A	A runs 1s.
24.5–25.0	D	D=0	-	D, E, A	T=23.5: D ready. Placed ahead of A. D runs its final 0.5s. D completes at 245.
25.0–26.0	E	E=0	-	A	E runs 1s. E completes at 26s.
26.0–30.0	A	A=0	-	-	A runs remaining 4s (4 quanta). A completes at 30s.

A = , B = , C = , D = E,
Mean turnaround time = (13+18.5 + 24+ 25 + 29) / 5 = 21.9s

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)

The CPU-bound process requiring 30 quanta on the CTSS scheduler must be swapped in 5 times, including the initial load. The CTSS employs a multi-level feedback queue where the quantum for class i is $2^{(i-1)}$, and a process is only swapped out and demoted if it uses its entire quantum. The process consumes its quanta sequentially in these classes: Class 1 (1 quantum, cumulative 1), Class 2 (2 quanta, cumulative 3), Class 3 (4 quanta, cumulative 7), and Class 4 (8 quanta, cumulative 15). Each time the full quantum is used, the process is swapped out and then swapped back in (a new swap) to the next lower class. Finally, it is swapped in for the fifth time to Class 5, which has a quantum of 16, but since the process only needs the remaining 30 - 15 = 15 quanta to complete, it finishes during this final execution and is not demoted further.