# CS 377 – Operating Systems

Discussion Session 4 Questions

Name: ____________________________

Write your answers individually, but feel free to consult your notes/slides/book this week. Be succinct (complete sentences not necessary). **Remember to turn your paper over.**

1. **CPU Scheduling**. Several types of scheduling policies were discussed in class – first-come-first-served (FCFS), round robin (RR), shortest job first (SJF, including multilevel feedback queues), and lottery scheduling (LS).

   (a) Suppose you want to optimize your scheduler for certain types of workloads. For each type, state and briefly justify which type of scheduler you would use: (i) multiuser workloads in which no individual user should be favored, (ii) workloads with many mixed CPU and I/O jobs, and (iii) workloads with frequent I/O bound jobs and some very long-running, CPU-heavy jobs.

   **Solution.** *For (i), we are interested primarily in fairness, which makes round robin a good choice – starvation is impossible and every job will get equal time on the CPU. For (ii), we want to get I/O work offloaded quickly so that the average waiting time isn't too bad, so SJF/MLFQ would be a good choice – since I/O bound jobs are prioritized, processes will avoid being delayed by I/O work. For (iii), we want to avoid starvation of the long-running CPU-intensive jobs, which makes lottery scheduling a good choice – this will trade off the waiting time benefits of SJF with the fairness and starvation avoidance of round robin.*

   (b) Suppose you have 2 jobs: job $A$ has length 10 and job $B$ has length 20. Job $A$ has 1 second of I/O every other second of work (starting after 1 second of work), while job $B$ has 1 second of I/O every 5 seconds of work. Using multilevel feedback queues and assuming three queues and no context switch time, sketch the scheduling of the jobs below. Remember the notation $Job^{workDone}_{time}$; for example, $B^2_6$ means that job $B$ has completed 2 seconds of work at time $t = 6$. The first two entries are filled in for you.

   **Solution.**

| Queue | Time Slice | Job |
|---|---|---|
| 1 | 1 | $A^1_1$ $B^1_2$ $A^2_3$ $A^3_6$ $A^4_9$ $A^5_{12}$ $A^6_{16}$ $A^7_{19}$ $A^8_{23}$ $A^9_{26}$ $A^{10}_{30}$ |
| 2 | 2 | $B^3_5$ $B^7_{11}$ $B^{12}_{18}$ $B^{17}_{25}$ |
| 3 | 4 | $B^5_8$ $B^{10}_{15}$ $B^{15}_{22}$ $B^{20}_{29}$ |

2. **Threads**. Two primary types of threads were discussed in class – user-level threads and kernel-level threads. Threads complement processes as basic components used to execute jobs on the CPU.

   (a) True or false: user-level and kernel-level threads are exclusive (that is, processes use one or the other). Briefly explain.

   **Solution.** *False – a process always has at least one kernel thread, since kernel threads are what the OS is actually controlling. On top of kernel threads, the program itself may be running any number of user threads. The user threads are under the control of the program, while the kernel threads are under the control of the OS.*

   (b) Suppose you have a multithreaded process that can be configured to use either kernel or user-level threads. Under each of the following situations about the process, which type of threads would you prefer (and why): (i) running on a quad-core machine, (ii) executing long I/O requests, and (iii) running an extremely large number of threads.

   **Solution.** *(i) kernel-level threads, as they would allow the OS to make use of multiple cores to execute threads in parallel. With user-level threads only, the OS will not be able to use the full resources of the machine on the multithreaded process. (ii) kernel-level threads, as user threads will cause the entire process to block on I/O requests. (iii) user-level threads, as the overhead involved is much less – there are no system calls or context switches required to work with user-level threads.*

   (c) Why would we want to use (kernel-level) threads at all instead of just using multiple processes?

   **Solution.** *Coordinating among threads is much easier because they share the address space of the process. This both makes interaction between threads easier from the programmer's perspective and cheaper from the system's perspective, since no system calls or message passing strategies are required to communicate between threads. Finally, since threads are more lightweight than processes (in terms of OS initialization, etc), it is more efficient to create many threads than many processes.*