Homework Assignment #2 Solution

Q1. What is a process in an operating system? What is a thread in an operating system?

ANSWER: *A process is a running program (binary, executable file) associated with necessary resources (e.g. address space, registers, open files, signals, etc.) to support running the program. (Process also provides an abstraction to support multiprogramming.)*

A thread is also called a lightweight process. Each thread can fetch and execute instructions independently (represent an independent control stream), but share address space and many other resources, such as open files, signals, accounting info, etc., among each other, which is different from the process model.

Q2. Modeling multiprogramming: (a) Assume the I/O fraction time of all processes is 20%, and assume processes are independent from each other, what's the CPU utilization, if the number of processes, *n* = 1, 2, 4, and 8, respectively? (b) If the I/O fraction time is 50% for all processes, what's the CPU utilization again, if the number of processes, *n* = 1, 2, 4, and 8, respectively?

ANSWER: *We discussed in the class that, under these assumptions, the CPU utilization can be calculated as:*

$$1 - p^n$$

Where p is the I/O fraction time, and n is the number of processes.

(a) Given p=0.2, thus, the results are: 0.8, 0.96, 0.9984, and 0.99999744.
(b) Given p=0.5, thus, the results are: 0.5, 0.75, 0.9375, and 0.99609375.

(Note: you may give a correct result with a different precision.)

Q3. Consider a system that has two CPUs and each CPU has two hardware threads (hyperthreading). Suppose three processes, P0, Pl, and P2, are started with run times of 5, 10 and 20 msec, respectively. How long will it take to complete the execution of these processes? **Please discuss all possibilities** depending on different processes scheduled to run on different CPUs/threads, and **what is the minimum execution time**? Assume that all three processes are 100% CPU bound, do not block during execution, and do not change CPUs once assigned.

ANSWER: *It may take 20, 25, 30, or 35 msec to complete the execution of these processes, depending on how the operating system schedules them. The minimum execution time is 20 msec. Since the assumption is that, all three processes are 100% CPU, which means the process spends all time on the CPU, and no any I/O time, and also processes do not change CPUs once assigned, thus we have the following possibilities:*

- *Each process is assigned to a different hardware thread (two CPUs have four hardware threads in total), and the time needed to complete the execution of these three processes would be determined by the slowest one, which is 20 msec.*

- *If two processes are assigned to a hardware thread, and the third process is assigned to another hardware thread. We can have three cases further:*
    - *If P0 and P1 are scheduled on the same hardware thread and P2 is scheduled on the other hardware thread, it will take 20 msec to complete all three processes. Again, the time needed to complete all three processes would be determined by the slowest one.*
    - *If P0 and P2 are scheduled on the same thread and P1 is scheduled on the other thread, it will take 25 msec to complete all three processes.*
    - *If P1 and P2 are scheduled on the same thread and P0 is scheduled on the other thread, it will take 30 msec.*

- *If all three processes are scheduled on the same hardware thread, it will take 35 msec.*

Among all these possibilities, the minimum execution time is 20 msec.

Q4 (Problem 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?

ANSWER: *If the only way to read from a file is the normal blocking read system call, a worker thread will block when it has to read a Web page from the disk. If user-level threads are being used, this action will block the entire process, destroying the value of multithreading. Thus, it is essential that kernel threads are used to permit some threads to block without affecting the others.*

Q5 (Problem 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? Please explain.

ANSWER: *Each thread calls procedures on its own, so it must have its own stack for the local variables, return addresses, and so on. This is equally true for user-level threads as for kernel-level threads.*

Q6 Please briefly discuss the advantages and disadvantages of implementing threads in user space and kernel space, respectively.

ANSWER: *As discussed in the class, the advantages and disadvantages of implementing threads in user space are below:*

Advantages

+ Fast thread switching than trapping to the kernel

+ Each process can have its own customized scheduling algorithm

Disadvantages

- Blocking system calls from any thread will block the entire process
- Page faults (requiring disk accesses) from any thread block the entire process
- No clock interrupts (no round-robin scheduling), need voluntary yield
- Limited performance gain as system calls block threads

The advantages and disadvantages of implementing threads in kernel are:
Advantages
+ When a thread blocks, the kernel can schedule another thread from the same process (if ready) or a thread from a different process to run, thus not blocking the process
+ No new non-blocking system calls required (including dealing with page faults)
+ Performance gain for mixed computing and I/O, as threads provide higher degree of parallelism, even within a process
Disadvantages
- Higher cost of creating/destroying threads as these require system calls, which is more costly than creating/destroying a user space thread

Q7 (Problem 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 non-preemptive?

ANSWER: *It certainly works with preemptive scheduling. In fact, it was designed for that case. When scheduling is non-preemptive, it might fail. For instance, if the* `turn` *is initially 0 (*`interested[0] = TRUE`*), but process 1 runs first. It will just loop forever and never release the CPU.*

Q8 (Problem 43). Measurements of a certain system have shown that the average process runs for a time T before blocking on I/O. A process switch requires a time S, which is effectively wasted (overhead). For round-robin scheduling with quantum Q, give a formula for the CPU efficiency, defined as the useful CPU time divided by the total CPU time including the overhead, for each of the following:

(a) $Q = \infty$
(b) $Q > T$
(c) $S < Q < T$
(d) $Q = S < T$
(e) Q nearly 0

ANSWER: *Since the CPU efficiency is the useful CPU time divided by the total CPU time, including the overhead, let's consider the following cases:*

First of all, let's consider when $Q \geq T$, which includes both (a) and (b). Since $Q \geq T$, what happens would be a process scheduled to run for T, then blocks on I/O, and have a context switch to run another process, with a context witch overhead of S. Based on our assumptions, the newly scheduled process will repeat the same pattern, i.e. run for T, then blocks on I/O, and have a context switch to run another process, with a context witch overhead of S. Thus, on average, we have an efficiency of $T / (S + T)$. Again, this would be the result for both (a) and (b).

*Second, when the quantum Q is shorter than T, which means a process runs for Q, then have a context switch to run another process, with a context witch overhead of S. On average, each run of T will require T/Q process switches, wasting a time S*T/Q. Thus, on average, the efficiency is: T/(T + ST/Q), which reduces to Q/(Q + S). This contains the case of (c), (d), and (e). For (d), we just substitute Q for S and find that the efficiency is 50%. Finally, for (e), as Q → 0 the efficiency goes to 0.*

Q9 (Modified Problem 45). Five batch jobs A through E, arrive at a computer center at almost the same time. They have estimated running times of 10, 4, 2, 6, and 8 minutes. Their (externally determined) priorities are 3, 5, 4, 2, and 1, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the average process turnaround time. Ignore process switching overhead.

(a) Round robin.
(b) Priority scheduling.
(c) First-come, first-served (run in order 10, 4, 2, 6, 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.

ANSWER: *Let's consider each case:*
(a) For round robin, during the first 10 minutes, each job gets 1/5 of the CPU. At the end of 10 minutes, C finishes. During the next 8 minutes, each job gets 1/4 of the CPU, after which time B finishes. Then each of the three remaining jobs gets 1/3 of the CPU for 6 minutes, until D finishes, and so on. The finishing times for the five jobs are 10 (job C), 18 (job B), 24 (job D), 28 (job E), and 30 (job A), for an average of 22 minutes.

(b) For priority scheduling, B runs first. After 4 minutes it is finished. The other jobs finish at 6, 16, 22, and 30, for an average of 15.6 minutes.

(c) If the jobs run in the order A through E, they finish at 10, 14, 16, 22, and 30, for an average of 18.4 minutes.

(d) Finally, shortest job first yields finishing times of 2, 6, 12, 20, and 30, for an average of 14 minutes.

Q10 (Problem 55). Consider the procedure *put_forks* in Fig. 2-47. Suppose that the variable *state*[*i*] was set to *THINKING after* the two calls to *test*, rather than *before*. How would this change affect the solution?

ANSWER: *This change would mean that after a philosopher stopped eating, neither of his neighbors could be chosen next. In fact, they would never be chosen. Suppose that philosopher 2 finished eating. He would run test for philosophers 1 and 3, and neither would be started, even*

though both were hungry and both forks were available. Similarly, if philosopher 4 finished eating, philosopher 3 would not be started. Nothing would start him.

THE END.