CS 450 Exam Answers

Adam Sumner

March 30th, 2015

1 True/False (with justification) Problems

1. **False**. In reality, context switches take longer than 0 seconds to complete. A RR scheduler with a decreased quantum will have a larger overhead than a RR scheduler with a larger quantum. Thus, decreasing the quantum of a RR scheduler will NOT improve the responsiveness of the scheduled jobs due to the system having to perform so many context switches.

2. **True**. When using non-preemptive schedulers, context switches are all predictable. This is because no outside process will interrupt the currently running process. The only time there will be any overhead using a non-preemptive scheduler is when the currently running process enables it, such as I/O, or when the process is done executing. Given a constant set of jobs, there is a finite amount of context switches, therefore context switch time overhead is also a constant.

3. **False**. Multi-level feedback queues determine priority based on burst CPU length. A High priority job may come in with a large burst CPU length, but because shorter quantum are reserved for higher level priority jobs, this large CPU burst process will be moved to the bottom of the queue. Thus, if a low priority job with a short cpu burst comes in, it will run before the high priority job with a high CPU burst

4. **False**. A preemptive RR scheduler will never have the issue of starvation. This method is fair to all processes so that even longer burst CPU processes will never starve.

5. **True**. A race condition deals with multiple threads accessing shared data. In a non-preemptive scheduler, the process must finish execution before another one begins. Furthermore, in a uniprocessor system, only one thread may run at any instant. It may seem that race conditions will not be an issue, however, they may still occur if the programmer gives up control willingly to another process and doesn't account for the possibility of a race condition between his processes.

2 Numerical Problems

6. **Algorithm #2 is SJF (non-preemptive), Algorithm #3 is FCFS, Algorithm #1 is RR q=50, and Algorithm #4 is SRTF**. The lowest amount of CST will correspond to schedulers that do not implement preemptive policies. This gives us two options either:

   (a) FCFS

   (b) SJF (non-preemptive)

   Our next criteria we look at is average wait time. SJF (non-preemptive) optimizes average wait time, so when comparing 2 and 3 it is clear that 2 is optimal. However, we must still look at the standard deviation. 2 has a higher standard deviation, which would make sense if it is SJF (non-preemptive) because the longer bursts of 75ms must wait until nothing shorter comes in until it can finally run. This creates a huge gap between wait times of small bursts and large bursts, resulting in a high standard deviation. In FCFS, all processes wait roughly the same time, therefore their standard deviation will be a lot lower.

   When comparing 1 and 4 we can also look at the average wait time. SRTF is equivalent to a preemptive SJF scheduler. As stated above, SJF algorithms are designed to optimize average wait time. When comparing 1 and 4, 4 has the lower average wait time. 4 ALSO has a higher standard deviation which is a characteristic of a SJF scheduler. Because RR has a controlled runtime of 50ms, all bursts will wait roughly the same time, resulting in a low standard deviation.

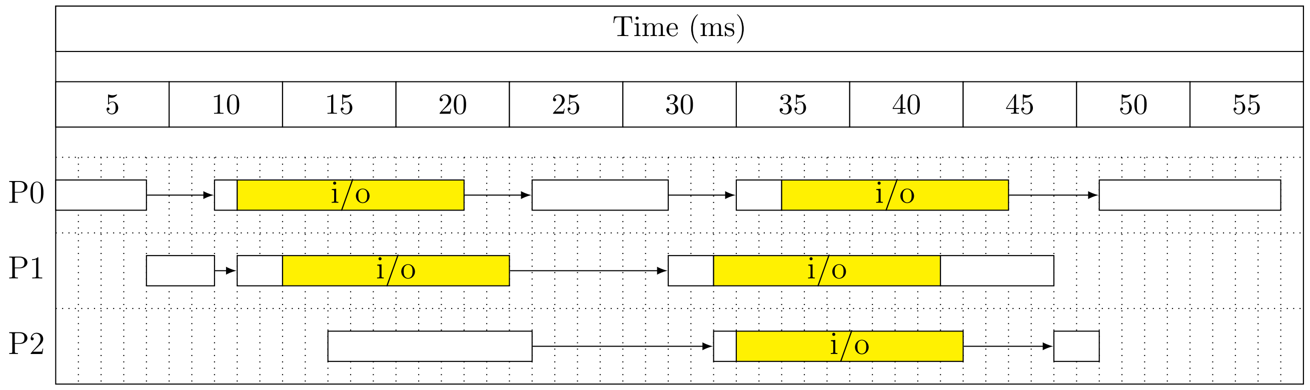

Figure 1: Gantt Chart for Problem 7

7.

Process	Wait Time
P0	13ms
P1	8ms
P2	12ms
Cumulative	33ms

Turnaround Time (Total Runtime) = 54ms

8.

$$\lambda = 3 \quad E(T) \to 2 \quad \mu = ?$$
$$E(T) = \frac{1}{\mu - \lambda}$$
$$2 = \frac{1}{\mu - 3}$$
$$\mu = 3.5$$

Therefore the rate must be 3.5 exams per hour

9.

$$\lambda = 3 \quad E(T) \to 1.8 \quad \mu = ?$$
$$E(T) = \frac{1}{\mu - \lambda}$$
$$1.8 = \frac{1}{\mu - 3}$$
$$\mu = 32/9 \approx 3.5555556$$

$$\lambda = 3 \quad E(T) \to 1.5 \quad \mu = ?$$
$$E(T) = \frac{1}{\mu - \lambda}$$
$$1.5 = \frac{1}{\mu - 3}$$
$$\mu = 11/3 \approx 3.666667$$

$$\lambda = 3 \quad E(T) \to 1 \quad \mu = ?$$
$$E(T) = \frac{1}{\mu - \lambda}$$
$$1 = \frac{1}{\mu - 3}$$
$$\mu = 4$$

$$\lambda = 3 \quad E(T) \to 0.2 \quad \mu = ?$$
$$E(T) = \frac{1}{\mu - \lambda}$$
$$0.2 = \frac{1}{\mu - 3}$$
$$\mu = 8$$

Therefore our increased percentages are 1.5%, 4.76%, 14.28%, and 128.5% respectively.

10.

$$E(L) = 5 \quad \lambda = 3$$
$$E(L_q) = \lambda E(T_q)$$
$$\frac{E(L_q)}{\lambda} = E(T_q)$$
$$5/3 = \frac{\lambda}{\mu^2 - \lambda\mu}$$
$$5/3\mu^2 - 5\mu - 3 = 0$$
$$\mu \approx 3.512$$

He must grade exams at a rate of 3.512 exams per hour

3 Discussion Problems

11. It's critical to choose a kernel that is able to handle a wide range of situations. It must be easily updatable, and must not use a lot of memory due to the limited amount of resources. I'd choose to organize my kernel to act as a hybrid of a monolithic and μ-kernel. A monolithic kernel has great performance but terrible extensibility while the μ-kernel has amazing extensibility and poor performance due to context switches. Monolithic kernels are designed so that the kernel oversees essentially every operation of the OS. This makes the source code extremely long and hard to update. If any update is to be made to the operating system, the entire kernel must be recompiled. However, because most processes are in kernel space, there is almost no overhead in execution. This creates an extremely efficient machine, but upgrades can be very time consuming for both the developers and the user(recompilation of the kernel). The μ-kernel has the opposite approach of the monolithic kernel. Kernel space only contains necessary functions of low layer abstractions while most processes such as the VFS, I/O, and device drivers remain in user space. This allows the kernel to be easily updated because of its modularity. The only downside of the kernel is the amount of context switches needed to execute these processes. By combining aspects of both of them, it's possible to develop an OS that does not make many context switches AND has great extensibility. By testing this operating system on a low CPU/RAM embedded system, it can then be determined which components will stay in kernel space and which components will go to user space. For example, device drivers might be better in user space so that they're easily upgradeable, while the VFS will stay in kernel space. For these reasons, a hybrid kernel is clearly the best choice for the design constraints given. It addresses both aspects of the kernel's robust capabilities and its limited amount of resources.

12. It's necessary to know that Policy refers to the overarching idea of what is to be done while the Mechanism refers to how it is done. When specifically talking about operating

systems, schedulers are a great example. For example, let's look at the SJF preemptive scheduler. This is a policy that MUST be separated from its implementation. This is because SJF schedulers are prone to starve longer running processes. If we combine the policy of shortest job first with its mechanism, the shortest job will ALWAYS run first, no exceptions. This means that starvation is a huge threat. This scheduler no longer becomes useful because there's a high probability that longer processes will starve. SFJ is a great policy because it aims to reduce average wait time, but its mechanism must be separate because anti-starvation measures need to be put in place. However, combining policy and mechanism can still be beneficial. This is because by combining policy and mechanism, this reduces the amount of code needed to be written. For example, look at the FCFS scheduler. The policy is simple, but it is not ideal because it incurs large average wait times. It is such a simple policy, that the mechanism can only be written one way with little variation. That is to calculate when processes arrive and then execute each process sequentially based on their arrival time. This is a great case to show that sometimes policy cannot be separate from mechanism. In this specific case, this policy is so universal that it would be a waste to separate its mechanism from its policy.

4 Xv6 Problems

13. Xv6 cannot directly switch from a user-space process to another user-space process. It's done through a user-kernel transition, a context switch to the scheduler, and then a context switch to the new process kernel thread. Every process has its own kernel stack and register set. First there's a call to `trap` (system call) to make the user to kernel transition. From here, the kernel thread calls `sched`. `sched` calls `swtch` which saves and stores the register sets of the first process known as a `context`, and loads the previously saved `context` of the CPU's scheduler thread. Now, the `scheduler` function is run and it runs through the `ptable` to find the next process to run. Once found, `scheduler` calls `swtch` which saves the `context` of the CPU scheduler and restores the `context` of the new process to be run.

14. The inner loop of `scheduler` iterates through the process table `ptable` to run each program in the table for a set period of time. However, it is up to the process being run to both acquire and release the `ptable.lock`. When a process acquires the lock, it is allowed to run assuming its state is RUNNABLE. If a process were to acquire the `ptable.lock` and never release it, no other processes in the table will be able to run. By doing this, every time `scheduler` comes across a RUNNABLE job, the job will not be able to acquire the `ptable.lock` in order to actually run. The "greedy" program will then be able to preempt all other jobs while the scheduler is completely unaware.

15. `DPL_USER` is the constant value of 0x3. There are three privileged levels in xv6 and the value of 0x3 corresponds to the user descriptor privilege level. `tvinit` initialzes the IDT vectors to supervisor mode. However, one interrupt vector is initialzed to user

mode. This is the `SYSCALL` interrupt. Line 3073 is significant because at this line, `tvinit` makes the `SYSCALL` interrupt accessible to the user. It's not used to initialize any other gate because it makes the system insecure. The only interrupt the user should have access to is the `SYSCALL` interrupt.

16. The `iret` instruction is what allows control to return to user space. There is one location in the xv6 source code where this instruction is called. It's called in the `trapret` function. `trapret` is only used in the `allocproc` function. This is in line 2236 of the xv6 source code.

    It's not possible to schedule a new process after a non-terminal system call. The system call does not terminate the process, therefore, the process must return to user space before the scheduler can be called. The only way to return back to user space is through the `trapret` function, therefore, to schedule a new process, the trapping process must precede.