CS 1217 Operating Systems, Spring 2023

Mid Term Exam Solutions

Date: **Saturday, 18th March, 2023**
Maximum Marks - **80**
Total time **- 90 Minutes (11:30 am - 1 pm)**

Instructions

1. Be as precise as possible in your answers. Please note that the answers are graded for content and not volume. Long, meandering answers will fetch negative points.
2. If you make any additional assumptions other than the ones stated in the question, state them clearly in your answer.
3. Ideal exam taking strategy would be to go over the entire question paper before you begin. Identify the questions that you can formulate a good answer for and attempt those first.
4. This mid-term exam accounts for 25% of your final grade.
5. **Plagiarism will result in a fail grade**. All the work that you submit should be your own.

Q #	Max Points	Earned Points
1	5	
2	5	
3	10	
4	5	
5	5	
6	5	
7	5	
8	5	
9	10	
10	25	
Total		

Q1
What is the purpose of the `exec()` system call? What happens to the address space of the process that calls `exec()`? Where is the information for change in the address space received from? (**5 points**)

- Load a new binary and start executing it
- Original process's address space is lost (overwritten); new one will have a completely new "face"
- The ELF binary that is supplied as an argument to exec()

Q2
Explain how the multi-level feedback queue (MLFQ) scheduling algorithm that we have discussed in class can *starve* processes. Assume the same MLFQ algorithm as discussed in class – there are three levels from where the scheduler can pick a process to schedule from on a single CPU, and the same feedback mechanism for rewarding or punishing process behavior. Propose one solution that addresses this problem. (**5 points**)

- Can have starvation due the CPU intensive nature of a process; if the process is CPU intensive and never does any I/O. or yields the CPU, then it will be relegated to the lowest of the priority levels, in the default MLFQ model.
- Solution: Redo MLFQ do that it periodically reshuffle process priorities, gives a chance to the starved processes to get have higher priorities (or move to higher priority levels) and have a better chance of getting a slice of the CPU

Q3. Consider an operating system where the list of process control blocks (PCBs) is stored as a linked list sorted by pid. The implementation of the wakeup function (which is used to wake up a process waiting on some condition) iterates over the list of processes in order (starting from the lowest pid), and wakes up the first process that it finds to be waiting on the condition. Does this method of waking up a sleeping process guarantee *bounded wait time* for every sleeping process? (**2 points**) If yes, explain why. If not, describe how you would modify the implementation of the wakeup function to guarantee bounded wait. (**8 points**)

No, this design can lead to process starvation. To fix it, we can maintain a pointer to where the wakeup function stopped the last time it was executing, and continue from there on the next call to wakeup().

Note: Any other reasonable solution will be given credit, as long as it retains the linked list as the main data structure.

Q4
Why is masking of interrupts needed? Describe a pathological case where not allowing interrupt masking would not allow the computer system to make any forward progress at all. (**5 points**)

- Needed to prioritize servicing of interrupts; some are more important than others and require bounded service times
- Consider the case where there is no masking of interrupts. The kernel starts servicing an interrupt and the next one happens. When its starts servicing the second one, the third happens. In the pathological case, a machine with a large number of I/O devices could keep receiving interrupts and without masking, none of them would get serviced.

Q5

Explain how the presence of a swap space on the disk allows the operating system to provide an illusion to processes of having practically "infinite" physical memory capacity, when in reality there is only a small amount of memory capacity to manage. Also comment on how the notion of a disk based swap space is a bad idea for a process' performance. Support your explanation by providing an example where the presence of swap space hurts process performance. (**5 points)**

On disk Swap space serves as an extension to main memory capacity. Since disk is cheap (in terms of rupees/GB), a much larger capacity, as compared to main memory, can be carved out as swap space. The OS as a part of virtual memory implementation manages the combined capacity of main memory and disk based swap space as a large virtual address space. Information in any processes' address can be present in memory or on disk and can be retrieved by the OS kernel from either location, without the application having to explicitly specify the location.

As a result of having a much larger combined space, more information can now be made to "fit" in the virtual memory space made up of main memory + disk swap space, which would not have been possible in the absence of swap space.

Disks are slow. If a significant portion of information for a process isn't in main memory, but needs to be retrieved from the disk based swap space, the latency of retrieving this information would be 100s of ms (as compared to 100s of ns, if the same information was located in DRAM/main memory). As opposed to the base case of everything being present in memory, this is bad for performance.

Q6

You have recently graduated from Ashoka University, and secured a job with the LinWin^{TM} Operating Systems kernel team. You are assigned to work on optimizing the process scheduling algorithms for the server version of the LinWin kernel. On your first day of the job, you report to your manager Rahul, who is a big Deep Learning fan. After briefly fanboy-ing about San Altman, Rahul starts to describe the latest DNN based scheduling algorithm that he has currently implemented. Rahul tells you that the new scheduling algorithm does a much better job at finding close-to-optimal schedules for processes compared to the MLFQ based scheduling algorithm that was previously implemented in LinWin. However, he is observing a significant slowdown in the total execution times of a given set of processes. Your first task on the job is to provide explanations regarding why this could be happening. Enumerate one possible explanation for bad performance and one for why the schedule is closer to optimal (**5 points**)

- Overheads of running the algorithm might be too high. Scheduling algorithms are optimized to make sure their run times, as well as their memory requirements might be high. This is not optimal
- Providing closer to optimal schedules might require frequent re-training of the network, assuming that is allowed. This will again take up resources, and will have computational overheads for running the re-trainings.
- All arguments that realize and acknowledge the overheads of ML algos are acceptable

Note: All arguments which diss DNNs as the solution to all the problems in CS will earn extra credit. (I kid, but only slightly)

Q7
List and describe the three thread states we discussed in class. Next, describe three transitions between them including when and how they occur. (**5 points**)

Thread states:
1. Running: actively executing instructions on a processor core.
2. Ready: not scheduled on a core, but ready to execute.
3. Blocked or Waiting: waiting for something to happen and not ready to run until it does.

Examples of transitions:
- Running → Ready: a context switch occurred and a thread was descheduled. Thread state is saved, thread is moved to the ready queue, and a new thread is chosen to run on the now-free core.
- Running → Waiting: a thread performed a block system call or began waiting for some other thing to happen. Context switch occurs, thread state is saved, the thread is moved to the waiting queue and a new thread is scheduled.
- Waiting → Ready: the blocking event that a thread was waiting for completed, and the thread can now continue to run. Thread is moved from the waiting queue to the ready queue.
- Ready → Running: thread was scheduled, chosen by the kernel to run on a processor core. Thread state is reloaded and thread is removed from the ready queue.

Q8
A. (Pick as many as you like) Virtual memory (**2.5 points**).

a. Makes your system run faster
b. Lets you run processes that exceed the size of memory.
c. Uses your I/O system more efficiently than systems without VM.
d. Lets you run more processes than can actually fit in memory.

B. Which is probably **not** a privileged operation? (**2.5 points**)

a. Changing the interrupt mask
b. Loading an entry into the TLB
c. Modifying the exception handlers
d. Adding two registers and placing the result in a third register

Q9

Recall our discussion in class, which classified the baseline version of `fork()` as an "expensive" system call, with non-trivial overheads associated with its execution. First, outline some of the most expensive parts of `fork()`.

Now, imagine a use case where a large majority of the processes that are created as a result of `fork()` end up calling `exec()`. Outline how a significant amount of effort being carried out by the baseline version of fork() is wasted by a call to `exec()`.

Finally, describe the phenomenon of copy-on-write (CoW) and how CoW could help in reducing the overheads of processes that fork() other child processes, but might not call exec() (**10 points**)

- Expensive parts: copying address spaces and replicating the parent's state in the child
- If exec() is called after fork(), then all the effort in replicating the address space of the parent in the child is wasted work – it is going to be overwritten by the call to exec(). It would be best to not copy anything into the child at the start (after fork() succeeds), but just point everything to the actual pages in parent process
- Only copy things in the child process when it tries to write to them (Copy - on Write)
- More details regarding CoW should have been mentioned : Can make all the address space parts that are read-only in the child process. All the child's pages point to that of the parent. The both keep using the same set of pages until the child needs to write something to its address space. At that point, the page in question is replicated and the child is given its own copy, to which it can now write as well.

Q10
So far, we have identified two main responsibilities for the OS: (1) providing abstractions and (2) managing system resources. During CS1217 we have been looking at case studies of how this is accomplished for system resources: the CPU and memory. Later, we will look at how this is done for the disk. We have discussed both how the OS performs and enforces resource allocations, and abstractions it provides to simplify resource usage.

In many cases, an equally-important resource for operating systems to manage is energy. Energy consumption matters at both ends of the computing spectrum, including for data centers for cost and power capping reasons, but particularly on mobile devices. For this question let's focus on smartphones.

First, consider how energy as a resource differs from other system resources we have discussed (**10 points**). These differences are significant from the perspective of designing effective approaches to managing energy. You might want to use memory management as a point of comparison, since several of the OS requirements for multiplexing memory don't really have appropriate analogs with energy, and several new capabilities are needed.

Second, present a ***detailed*** design allowing operating systems to manage process energy consumption on battery-powered smartphones, describing any changes needed at the hardware, OS, and system call interface layers (**15 points**). Be careful not to make assumptions about OS capabilities regarding energy. Compared with the resources we have discussed, managing energy has some unique prerequisites that complicate the problem. It might be helpful to consider, as a starting point, an OS that knows nothing about the energy consumed by processes and go from there. You don't need to provide abstractions unless they are required, but your solution should enable multiplexing. You may want to consider typical usage of smartphones as well as charging patterns as part of your design. (**25 points**)

There were two parts of this question, clearly identified within the question. The first asked you to identify differences between energy and other resources that would be significant for energy management. The second asked you to design a way for the OS to manage energy.

First, there are several important differences. One is that, unlike other system resources, energy is depletable. Once it's consumed, it can't be reused, nor can previous allocations be revoked. And when it's gone, the device cannot be used. Contrast this with the processor, which the device never "runs out of": as long as the device is on, there is more processor time. For the disk and memory, the device can run out, but at that point both memory and storage can be reused by revoking previous allocations. Memory can be reused by swapping out pages or, in the worst case, simply by killing processes, in both casing creating new unused pages for a new process. Disk space can be reused by prompting the user to remove old files.

As a result, bad OS energy management has the potential to be more damaging than poor management of other resources. Consider what happens if the OS makes a bad scheduling decision: the wrong thread or process runs for a few tens of milliseconds, and then the OS has the chance to make a better scheduling decision. The device isn't going to have to power down as a result. Also with memory - if the OS swaps out the wrong page, it may have to swap it right back in, hurting performance. But again: the device isn't going to power off, and the contents of the memory page are preserved. But make a poor energy allocation decision and you may have just reduced the device's lifetime by seconds and can never recover from the mistake. So clearly this is important!

A second significant difference is that energy is consumed implicitly by the usage of other system resources, not directly. Processes don't consume energy in order to consume energy, the consume energy in order to use the processor, or read and write to memory, or send data over the network interface, or draw to the display. Pretty much every system component consumes energy to run, but the point isn't to consume energy - it's to accomplish something else. *Energy consumption is a side effect*. Second, as a guide to our design let's return to our memory multiplexing requirements as the question suggested and see if we can modify them for energy management. They were:

1. the kernel should be able to allocate the resource to processes
2. the kernel should be able to enforce resource allocations efficiently.
3. the kernel should be able to reuse the resource if it is unused.
4. the kernel should be able to stop a process from using resources it was previously allocated.

Clearly #4 doesn't apply, since once energy is consumed it is gone. However, we should be able to design mechanisms to accomplish the other three. In addition, energy consumption presents a new requirement: **measurement**. We had really been relying on this all along: in order to enforce allocations, the kernel has to be able to measure the resource being used. But with energy it's complex enough to deserve its own bullet point. (We'll get back to that in a minute.) So what we have is:

- Measure: the kernel should be able to measure the amount of energy used by each process.
- Grant: the kernel should be able to allocate energy to processes.
- Enforce: the kernel should be able to enforce energy allocations efficiently and prevent processes from using more energy than they were allocated.
- Revoke: the kernel should be able to stop a process from using energy it was previous allocated but has not yet consumed. Let's go step by step.

Measure: this isn't as trivial as it sounds. In fact, it's about half of the battle when managing energy. There are two main challenges. First, each hardware component consumes energy differently: one CPU consumes more energy than another, and may consume a different amount of energy depending on which frequency level it is running at. Using a Wifi wireless network consumes less energy-per-byte than mobile data networks such as 3G or 4G, and more than using a wired network. Spinning disks may consume more energy per operation than Flash. Normally systems work around this by using a power model, which allows energy consumption to be estimated in software based on usage of the component. For example, if a process sent a certain number of bytes over a particular network interface, then it consumed a certain amount of energy to do so. Power models can be quite complicated, however - when estimating network energy consumption you might need to incorporate things such as the signal strength of the network connection at the time that the data was sent. In addition, each component requires its own power model, further complicating the OS. Alternatively, we could abandon the power model and simply rely on each hardware component to perform its own energy measurement, exposing the energy consumed to the OS through a hardware interface. This is a nice way of avoiding the problem while still respecting hardware differences.

Second, it can be difficult even to figure out what process is using which resource. When resources are used synchronously—i.e., you have to be running on the CPU to use them—then this is somewhat simplified. (But what about memory energy consumed on a multi-core system?) But asynchronous resources pose their own problems, partly due to layering caused by good interface design. For example, by the time a disk write reaches the disk driver—which knows the power model—it may no longer identify the process that issued it, which also will not be the process currently running. Solving this problem is an open engineering problem, so we didn't expect you to—but you would receive credit for identifying it.

Grant and Enforce: if you can measure, you can grant and enforce. Granting energy simply allows the process to consume it. Depending on your design, you might have exposed energy grants to the process in some way, or not. Telling the process how much energy it is allowed to use may allow it to make better decisions about how to execute. Alternatively, it may have no idea how to use this information. So there are arguments both ways.

Enforcement, however, is a bit different due to the implicit nature of energy consumption. If the OS wants a process to use less memory, it can move its pages to disk. If it wants a process to use less CPU, it can schedule it more often. However, if it wants a process to use less energy, it has to prevent it from using other system resources— any of them—that would consume energy. The simplest and most effective thing to do is simply stop it from running. That way it can't get the the processor to run, use memory, allocate memory, or use the network or other peripherals, all of which would consume energy.

Revoke: this one is a bit interesting in the case of energy. To allow processes to plan around their energy allocation, you might want to give them a bundle of energy to use and then force them to request more when need it. Various research operating systems have explored ideas like this, using abstractions like energy "tickets" which express a reservation on some of the energy remaining in the battery.

The problem is hoarding: what does the OS do if a process requests energy that it doesn't use? This could happen because it doesn't understand its own energy consumption, which is possible, or simply because it wasn't run as often as it thought it would be. In either case a revocation mechanism might be required to allow the OS to reallocate available energy by invalidating previous unused allocations.

Allocation Policy: a final issue that you should have addressed is how to allocate energy, a unique challenge given its depletable nature. Resources such as the CPU and memory are usually allocated as needed based on some prioritization scheme, but allocating energy this way doesn't really allow the OS to perform energy management— it just ends up being purely a side-effect of other allocations.

Instead, it might be worth considering a goal of energy allocation, similar to how we discussed interactivity or throughput as potential goals of processor scheduling. One possible goal that many previous systems have explored is the idea of meeting a lifetime target. Given a length of time, say 8 hours, the OS is in charge of metering energy consumption to ensure that the device lasts for at least 8 hours. Users could configure this based on their charging patterns to ensure that the device would never run out of energy before reaching a plug. One way to accomplish this is to simply divide available energy over that time interval and meter it out evenly. At any point in time, if the device is over its energy budget processes must stop running; if it is below the budget, they can run freely.

Unfortunately, this isn't necessarily a great idea. First, smartphones don't consume energy smoothly but in bursts. I may use my smartphone intensively for an hour and then not again for four hours. If I meter smoothly, then I'm bumping up against the energy budget repeatedly during the first hour and reducing performance, but then have built up a big surplus during the interactive period. Another problem is that stopping interactive tasks that have run out of energy is bound to frustrate users. Instead of taking 10 s to load, my energy-limited browser takes 1 min to load the same page because of an energy budget.

PS : There are still no good solutions for managing energy budgets of mobile devices, especially from an OS perspective. If you think you have good ideas on how to do this, and have ideas on how to implement and verify the proposed techniques, come talk to the instructor.