CS 111: Operating System Principles

Midterm Exam Summer '21

Instructor: Jon Eyolfson

Overview.

“All problems in computer science can be solved by another level of indirection.”

Describe one operating system abstraction of your choice, and what specifically it abstracts.

> *Virtual memory is one of the key operating system abstractions. It abstracts physical memory. Instead of having to partition physical memory to individual processes and hoping they only access memory that they should, we make each process believe it has access to all of physical memory.*

Interfaces.

You wrote a kernel module in Lab 0, which must have run in kernel mode. You also accessed it in user mode (in your terminal) by using `cat /proc/count`. What is the interface between the two CPU modes? Describe how `cat /proc/count` works using this interface, and be specific about the calls.

(Hint: you do not need to describe how your kernel module works).

> *The system call interface is the interface between the two CPU modes. The user process will use the* `open` *system call to open the "virtual file"* `/proc/count` *and the kernel knows this isn't a regular file. The user process then calls the* `read` *system call, after it transitions to kernel mode it reads the bytes printed by the Lab 0 kernel module.*

Libraries.

Assume you're writing a library you intend for lots of people to use it (and they will, because you know good library principles!). Would you deploy your library as a static or dynamic library? Describe the benefits of your choice over the other.

As your library evolves, describe what you'd have to do to ensure a pain-free experience for your users. Also, describe what would users would have to do to use your new version of the library.

> *I would deploy the library as a dynamic library. The benefit is that everyone that uses the library can recieve updates without having to recompile. The operating system can also load one copy of the library and share it between many running applications.*
> *To ensure a pain-free experience I would never change the API or ABI. I would only add new functions, and never change any existing API or ABI. If I had to change something major, I would create a new major revision. Users don't have to do anything to use the new version of the library, they just need to get the library update.*

Processes.

Processes virtualize both CPU and memory, and saves state in a process control block (PCB). For both resources please describe what the PCB would need to store to allow context switching, and why. Please use a separate paragraph for each resource.

For the CPU, the PCB would have to store it's registers (basically it's state). It would just need to save and restore the registers when it context switches.
For virtual memory, each process would have it's own page tables. For a context switch we would just need to save a pointer to the root page table (the highest level page table). To switch processes, we just need to swap this pointer (which is actually just a register).

Basic IPC.

You're running a program in your terminal. You type the command and hit enter. Jon wrote the program you're running, so you're 100% certain it's executing in an infinite loop. You press Ctrl+C, and the process ends, giving you back control of your terminal. Describe how the process was able to exit, even though it was in an infinite loop. Be as specific as possible.

> *When you press Ctrl+C the shell sends a* `SIGINT` *signal to Jon's process. The kernel then switched Jon's process to run it's signal handler. The signal handler than called exit, and the process terminated. (This is the default behavior, Jon was lazy and didn't write his own signal handler).*

Basic Scheduling.

Between FCFS, SRTF, and RR, which scheduling algorithm(s) do NOT suffer from starvation. Explain how the algorithm(s) avoid this problem.

FCFS and RR do not suffer from starvation. FCFS does not suffer from stravation because there's no "line jumping", you sit in queue until it's your turn. You can't be constantly sent back in the queue. The argument is the same for RR, but instead of waiting for processes to finish, we only have to wait for one time slice for every process ahead of us.

Advanced Scheduling.

Assume you have a critical user process on your Linux system that must run before any other process, what would you do to ensure it gets scheduled before other processes?

(Hint: you can't modify your kernel).

> *We would change it to a real-time process. We could also give it the highest priority possible (100, but you don't need to state the number). Since it's a real-time task, it'll get scheduled before any normal processes. It's also the highest priority, so it would only compete with other high priority real-time processes.*

Page Replacement.

Using FIFO page replacement has several drawbacks, including suffering from Bélády's anomaly. Why might you still choose to use FIFO page replacement?

> *It's the easiest to implement, and fastest running algorithm. If we don't expect to swap many pages anyways, the algorithm doesn't matter as much. It's simply just a queue, which is easy to implement and understand.*

Process API.

Consider the following code:

```
#include <unistd.h>

int main() {
  pid_t pid = fork();
  if (pid > 0) {
    sleep(10);
    // (1)
    sleep(10);
  }
  return 0;
}
```

Assume we execute this program, and it gets assigned PID 100. When PID 100 forks, it creates a child with PID 101.

Question 1.

What did we forget to do in the parent?

(Hint: the answer isn't error checking, assume that I checked for errors. The code is like that to be more readable).

> *We forgot to call* `wait`*, we're a bad parent. We did not acknowledge our child.*

Question 2.

At point (1), what is the state of our child process? Why is it in this state?

> *It's a zombie process. It's in that state because it exited (it doesn't* `sleep` *at all), and the parent never calls* `wait`*.*

Question 3.

After PID 100 exits, what happens to PID 101? Explain.

> *As PID 100 exits, PID 101 is still a zombie, as before. When PID 100 terminates then PID 101 also becomes an orphan. It's re-parented, likely to* `init`*, and the new parent calls* `wait`*. The kernel can now remove PID 101's resources.*

Page Tables.

Consider a system with 8192 byte pages, a page table entry (PTE) size of 8 bytes, 40-bit virtual addresses, and 56-bit physical addresses.

Question 1.

How many offset bits are there in the virtual address? *13.*

Question 2.

How many bits of the virtual address are left over for index bits? *27.*

Question 3.

For a single page table (not a multi-level page table), how large would the table have to be (in bytes, KiB, MiB, or GiB)? Explain how you arrived at that size.

> *1 GiB. Our page table would have 2^{27} entries, each are 8 bytes (or 2^3 bytes), therefore it would be 2^{30} bytes in total.*

Question 4.

Considering the size you calculated in the last question, is it realistic to use a single page table in the real world? Explain why or why not.

> *Absolutely not. If we have 16 GiB of RAM on a machine, we could only run 15 processes at most. The 15 processes would have to share the last 1 GiB between themselves (and the kernel).*

Question 5.

How many PTEs could fit on a single page?

> *1024, or 2^{10}.*

Question 6.

How many levels of page tables would you need, if you ensure that each (smaller) page table fits on a page. *3.*

Question 7.

Explain how you arrived at your answer for the previous question.

> *For each smaller page table, we have 10 index bits. In total we need to index 27 bits (40 - 13). Therefore we need $\lceil \frac{27}{10} \rceil$ levels.*