CS 111: Operating System Principles

Midterm Exam Fall ’21

Instructor: Jon Eyolfson

October 26, 2021

Duration: 1 hour 50 minutes

Name

Student ID

This is a closed book exam. You are only permitted a pencil or pen.
Answer the questions directly on the exam.

If in doubt, write your assumptions and answer the question as best you can.
There are 8 numbered pages (page 8 is blank if you need extra room).
The pace of the midterm is approximately one point a minute.
There are 100 total points.

Good luck!

(5 points) Explain the concept of virtualization and how it applies to operating systems.

The concept of virtualization is making something believe it has exclusive access to a resource, when it's actually being shared. For instance virtual memory allows multiple processes to think they can access the entire memory while they actually share physical memory.

(5 points) What service would you find in a monolithic kernel, but not in a microkernel?

File systems / Device drivers / Advanced IPC

(5 points) What should you use to monitor all system calls a process makes?

You can use `strace` to monitor system calls.

(5 points) If you include a C `struct` in your library's header, why shouldn't you ever change it?

Because you'd be changing the ABI of your library. The application would compile one version, and a library update would compile an incompatible version. At runtime, when the new library runs, there would be a mismatch and odd bugs would happen.

(5 points) What are the two responsibilities of `pid` 1 (`init`)?

The first job is to create processes, it's the parent of all user processes, either directly or through it's children (grandparent, etc.). The second job is to acknowledge all orphan processes it may receive, so it'll infinitely call `wait`.

(5 points) Why do we not use the least recently used algorithm to do page replacement in practice?

The least recently used algorithm suffers from too many performance problems in practice. You either need to scan every single page, or maintain a recently used queue for every access. Both approachs are too slow.

(20 points total) Process API.

Consider the following code:

```
#include <sys/wait.h>
#include <unistd.h>

int main() {
  pid_t pid1 = fork();
  pid_t pid2 = fork();

  if (pid1 > 0) {
    int wstatus;
    wait(&wstatus);
  }

  if (pid2 > 0) {
    int wstatus;
    wait(&wstatus);
  }

  return 0;
}
```

Recall that `fork` creates a new process, that is a copy of the current running process. It returns a process ID, `pid`. If `pid` is greater than `0` then it represents the process ID of its new child process. If `pid` is equal to `0` then this process is the new child process. We'll assume these are the only possibilities, `fork` never generates errors. The `wait` function waits until one of its child processes terminates, and reads its status information so the kernel can remove its resources. We can assume that all processes exit normally. We don't need to access the information in `wstatus`, so for this question it's irrelevant. We also don't check the return value, so we don't need to know it for this question.

We compile the program on the previous page, and execute it as a new process, `pid` 100. Again, we assume that `fork` does not fail, and all processes that terminate exit normally.

(2 points) How many *new* processes get created (exclude `pid` 100)?

3 proceses get created in total.

(8 points) Does `pid` 100, or any of its children create any orphan processes? Why?

No, they do not. The first process (`pid` 100) creates two children and calls `wait` twice. The first child creates one child itself, and calls `wait` on it.

(10 points) There's an issue with this program. When you run it, it seems fine, but that's because we don't check for any errors. What is this issue, and how would you fix it? (You can just describe what you'd need to do to fix it, instead of writing code.)

It might be easier to look at the processes and the values of the variables:

Process	*`pid1`*	*`pid2`*
1 (`pid` 100)	*>0*	*>0*
2 (first child of `pid` 100)	*0*	*>0*
3 (second child of `pid` 100)	*>0*	*0*
4 (child of process 2)	*0*	*0*

Process 3 (second child of `pid` 100) calls `wait` when it doesn't have a child. To fix it, we would only have processes wait on children it directly creates. We just got a copy of `pid1` in the second of child the original process, even though it didn't directly create that child.

(20 points total) Basic IPC.

Consider the following code:

```
#include <sys/wait.h>
#include <unistd.h>

int main() {
  int fd[2];
  pipe(fd);

  pid_t pid = fork();
  if (pid == 0) {
    /* first child */
    dup2(fd[1], 1);
    close(fd[1]);
    execlp("ls", "ls", NULL);
  }

  pid_t pid = fork();
  if (pid == 0) {
    /* second child */
    dup2(fd[0], 0);
    close(fd[0]);
    execlp("wc", "wc", NULL);
  }
  else {
    close(fd[0]);
    close(fd[1]);
    int wstatus;
    wait(&wstatus);
    wait(&wstatus);
  }

  return 0;
}
```

Recall that `pipe` creates two file descriptors: `fd[0]` and `fd[1]`. You can only write data to `fd[1]` and only read data from `fd[0]`. The `close` function takes a file descriptor as an argument and closes it, allowing the kernel to clean up the entry. The `dup2` function copies the file descriptor in the first argument to the file descriptor in the second argument. If the file descriptor represented by the second argument already exists, it's closed before the copy. The `execlp` function takes a string, representing an executable name, and any number of string arguments terminated with a null pointer. The function searches for the executable and if found, replaces the currently running process with that one (also passing the arguments provided). Assume that no functions ever fail.

We compile the program on the previous page, and execute it as a new process. Again, we assume `fork`, `pipe`, `dup2`, and `close` do not fail, and all processes that terminate exit normally.

(10 points) Explain how `ls` and `wc` communicate using the pipe. You should explain it in terms of each process, and the `read` and `write` system calls (both functions operate on a file descriptor and a sequence of bytes).

> *The original process crewates a pipe and manipulates the file descriptors. In `ls` the write end of the pipe replaces file descriptor 1. Internally `ls` calls `write(1, ...);`, which now goes to the pipe. In wc the read end of the pipe replaces file descriptor 0. Internally wc calls `read(0, ...);`, which now gets data from the pipe.*

(10 points) When you run this program, it looks like it hangs. Why? What would you have to do to fix it?

> *The second child (wc) keeps the write end of the pipe option. The pipe thinks that it could recieve more data and never allows the process to see "end of file" from its `read` call. We would have to `close(fd[1])` in the second child. (You could also `close(fd[0])` in the first child if you want.)*

(20 points total) Scheduling.

Consider the following processes you'd like to schedule:

Process	Priority	Arrival Time	Burst Time
P_1	2	0	4
P_2	1	5	3
P_3	1	7	2
P_4	1	1	3
P_5	2	3	1

You decide to use a round robin scheduler with a quantum length of 2 time units, and a priority queue. You decide that a larger priority number means a process has a higher priority. You decide to schedule the processes such that a higher priority process always runs ahead of a lower priority one. Processes that have the same priority round robin normally.

(13 points) Fill in the boxes with the current running process for each time unit.

0 13

P_1	P_1	P_1	P_1	P_5	P_2	P_2	P_4	P_4	P_3	P_3	P_2	P_4

or

0 13

P_1	P_1	P_1	P_1	P_5	P_4	P_4	P_2	P_2	P_3	P_3	P_4	P_2

(3 points) What's the average response time? (Your answer can be fractional.)

$$\text{Avg}_{\text{ResponseTime}} = \frac{0+0+2+6+1}{5} = \frac{9}{5} = 1.8$$

or

$$\text{Avg}_{\text{ResponseTime}} = \frac{0+2+2+4+1}{5} = \frac{9}{5} = 1.8$$

(4 points) What's the average waiting time? (Your answer can be fractional.)

$$\text{Avg}_{\text{WaitingTime}} = \frac{0+4+2+9+1}{5} = \frac{16}{5} = 3.2$$

or

$$\text{Avg}_{\text{WaitingTime}} = \frac{0+5+2+8+1}{5} = \frac{16}{5} = 3.2$$

(10 points total) Page Tables.

Your system has 2 MiB (2^{21}) pages, a PTE size of 8 bytes, and uses 57 bit virtual addresses. You decide to use multi-level page tables, and fit each smaller page table on a single page.

(2 points) How many PTEs can you fit into a single smaller page table? (Answer can be a power of 2.)

You could fit 2^{18} PTEs into a page size of 2 MiB.

(4 points) How many levels do you need for your multi-level page table? Show your work.

You need 2 levels for your multi-level page table.
$\lceil \frac{57-21}{18} \rceil = \lceil \frac{36}{18} \rceil = 2$

Now that you have a multi-level page table with *n* levels (if you didn't calculate *n* you can assume a value greater than 1). You want to calculate the effective access time of this approach. On this system it takes 10 ns to search the TLB, each memory access takes 100 ns, and we have a hit rate of 50%. Recall that for a single page table the equation for effective access time is:

$$\text{EAT} = \alpha \times \text{TLB}_{\text{HitTime}} + (1 - \alpha) \times \text{TLB}_{\text{MissTime}}$$

$$\text{where}$$

$$\begin{aligned} \text{TLB}_{\text{HitTime}} &= \text{TLB}_{\text{Search}} + \text{Mem} \\ \text{TLB}_{\text{MissTime}} &= \text{TLB}_{\text{Search}} + 2 \times \text{Mem} \end{aligned}$$

(4 points) Calculate the effective access time for this multi-level page table. Show your work.

You need 2 levels for your multi-level page table. Therefore on a TLB miss you need 3 memory accesses, one for each level of the page table and one for the original access.

$$\begin{aligned} \text{TLB}_{\text{HitTime}} &= 10 \text{ ns} + 100 \text{ ns} \\ \text{TLB}_{\text{MissTime}} &= 10 \text{ ns} + 3 \times 100 \text{ ns} \\ \text{EAT} &= 0.5 \times 110 \text{ ns} + 0.5 \times 310 \text{ ns} \\ &= 55 \text{ ns} + 155 \text{ ns} \\ &= 210 \text{ ns} \end{aligned}$$