ECE 353: Systems Software

Midterm Exam Winter ’23

Instructor: Jon Eyolfson

March 1, 2023

Duration: 1 hour 15 minutes

Name

Student ID

This is a “closed book” exam, you may have a single double-sided aid sheet.
This sheet must be handwritten and not mechanically reproduced.
You are only permitted a pencil or pen to write your answers.
Answer the questions directly on the exam.

If in doubt, write your assumptions and answer the question as best you can.
There are 7 numbered pages (page 7 is blank if you need extra room).
The pace of the midterm is approximately one point a minute.
There are 75 total points. Good luck!

Short Answers (30 points total)

Q1 (2 points) What prevents a process from modifying their own page tables?

> Processes do not run in kernel mode and do not have access to their page tables, nor could they use any special instructions to modify their root page table. (1 point for just saying "the kernel", full marks for valid reasoning)

Q2 (3 points) Describe an undesirable outcome a process could modify its own page tables.

> A process could access any memory on the machine, including the kernel and other processes. It could read or modify any memory on the system it wanted which would cause security issues and unexpected behaviour.

Q3 (5 points) How can processes communicate using file descriptors if they're independent?

> File descriptors act as pointers. When you fork a process, your process gets its own independent set of file descriptors, but they point to the same resources as the parent. Therefore, if you use them to access the same resources in different processes, you can use them to communicate. The file descriptors themselves are still independent (if you `close` in on process, it's still available in the other).

Q4 (5 points) Can a newly created process communicate with an existing process that is not directly related using pipes? Explain why or why not.

> No, if they're not directly related then the existing process would not be able to get the file descriptors from a call to `pipe`. The file descriptors are only shared through `fork`, and you cannot `open` a pipe.

Q5 (2 points) Aside from calling `exit` or `exit_group` what's another way to terminate a process?

> A process can be terminated by sending a signal (`SIGKILL` will definitely do it).

Q6 (3 points) For scheduling processes your friend says we should switch processes after they execute only a few instructions to give the lowest response time possible. Why would you not want to do this?

> The response time would be very low, however, context switching between processes takes time. If you context switch too much then you're wasting most of your CPU time doing context switches instead of doing useful work.

Q7 (5 points) You ask someone to write you some code that prints 1024 bytes from a file descriptor. They come back with two possible solutions. The first solution is as follows:

```c
char byte;
for (int i = 0; i < 1024; ++i) {
  read(fd, &byte, 1);
  printf("%c\n", byte);
}
```

The second solution is:

```c
char buffer[1024];
read(fd, buffer, 1024);
for (int i = 0; i < 1024; ++i) {
  printf("%c\n", buffer[i]);
}
```

Assume that `read` and `printf` are always successful. Which solution would you choose? Justify your answer.

System calls are relatively slow operations, you would want to do as few system calls as possible. Therefore, the second solution that does only one system call would be best.

Consider the following code snippet:

```c
void child(void) {
    int fd = open("/dev/null", O_RDWR);
    dup2(fd, 1);
    printf("Debugging...\n"); /* (A) */
    execlp("ls", "ls", NULL);
    exit(errno); /* (B) */
}
```

Assume that `child` executes after a `fork` where `fork` returns `0` and the calls to `open` and `dup2` are successful. The `"/dev/null"` "file" is a location where any bytes you write never get stored (it's a no-op).

Q8 (3 points) You're trying to debug your program by inserting a print statement at line `(A)` in the code. However, you never see `"Debugging"` in your terminal. You verify with `strace` that this code runs and `open` and `dup2` are successful. Explain why you do not see `"Debugging"` in your terminal.

We changed file descriptor 1 to point to `/dev/null` through the `dup2` call. `printf` just writes to file descriptor 1, so now all the output from `printf` is going to a location that is a no-op.

Q9 (2 points) Would line `(B)` execute the above code? Explain why or why not.

No, it would not execute assuming `execlp` is successful. If successful the process starts running the `ls` program. We would only see that line execute if `ls` did not exist.

(15 points total) Processes.

Consider the following code:

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

int main() {
  int x = 1;
  pid_t pid = fork(); /* (A) */
  if (pid == 0) {
    exit(0);
  }
  sleep(1);
  pid = fork(); /* (B) */
  int y = 3;
  if (pid == 0) {
    sleep(3);
    int wstatus;
    wait(&wstatus); /* (C) */
  }
  else {
    x = 2;
    y = 4;
  }
  printf("x: %d, y: %d\n", x, y); /* (D) */
  return 0;
}
```

Assume that `fork` and `sleep` are successful. The call to `sleep` puts the process to sleep for `t` seconds, where `t` is the argument to the function. We compile the program and execute it as a new process, `pid` 100. Answer the questions on the following page.

Q10 (2 points) How many *new* processes get created (exclude `pid` 100)?

2.

Q11 (8 points) Describe the state of all `pid` 100's children after `pid` 100 terminates.
Briefly justify why they're in this state.

The child created at line `(A)` immediately exits. It's never acknowledged by the parent process, so it's a *zombie process*. In addition, would now also be an orphan, so it's a *zombie orphan process*.

The child created at line `(B)` is still sleeping (it sleeps for 3 seconds). The parent process will exit very quickly, well before 3 seconds. It would now be an *orphan process* after `pid` 100 terminates.

Q12 (3 points) What do you expect to happen after executing line `(C)`? Without changing the placement of the `wait` is there anything you should do to improve this code?

You would expect it to fail, because the child process calls `wait` and it doesn't have any children. We should've checked for errors, since it would've returned `-1` and set `errno`. We likely didn't mean for the child to wait, so it's best to get an error and just handle it if you intended for it to occur.

Some students may have changed the `if` condition, however that would cause the processes to print opposite outputs in the next question.

Q13 (2 points) Show the output of line `(D)` when it's executed for all processes.

Only the parent process and the process created at `(B)` make it to line `(D)`.

The parent process would print: `x: 2, y: 4`.

The process created at line `(B)` would print: `x: 1, y: 3`.

(15 points total) Scheduling.

Consider the following processes you'd like to schedule:

Process	Arrival Time	Burst Time
P_1	0	4
P_2	3	2
P_3	8	1
P_4	1	4

You decide to use a round robin scheduler with a quantum length of 2 time units, and a priority queue.

Q14 (11 points) Fill in the boxes with the current running process for each time unit.

0 11

P_1	P_1	P_4	P_4	P_1	P_1	P_2	P_2	P_4	P_4	P_3

Q15 (2 points) What's the average response time? (Your answer can be fractional.)

$$\text{Avg}_{\text{ResponseTime}} = \frac{0+3+2+1}{4} = \frac{6}{4} = 1.5$$

Q16 (2 points) What's the average waiting time? (Your answer can be fractional.)

$$\text{Avg}_{\text{WaitingTime}} = \frac{2+3+2+5}{4} = \frac{12}{4} = 3$$

(15 points total) Page Tables.

Consider an odd system with a page size of 256 bytes. It uses 16 bit virtual addresses, with a PTE size of 16 bytes. The system supports 32 bit physical addresses.

Q17 (2 points) How many PTEs can you fit into a single page? (Answer can be a power of 2.)

> 16 ($\frac{2^8}{2^4} = 2^4$)

Q18 (3 points) If we used multi-level page tables, how many levels of page tables would we need? Show your work.

> You need 2 levels for your multi-level page table.
> $\lceil \frac{16-8}{4} \rceil = \lceil \frac{8}{4} \rceil = 2$

Q19 (1 points) Is virtual address `0xH1H1H1` valid? Why or why not?

> No, `H` is not a valid hex character. Also, assuming `H` is `F` this would be a 24-bit address, which is too large for our system.

Q20 (4 points) Explain the steps to translate the virtual address `0x1234` into physical address `0xFEED0034` (assume the page tables only have valid PTEs)

> Your process would have a root L1 page table (stored in a register), you would read the PTE at index `1` (from the virtual address). The PTE you read there would point you to the physical page where the L0 page table is. Assuming there's a valid page table there, we would look at index 2 (from the virtual address) and read that PTE. That PTE would contain the PPN `0xFEED00`, and we'd read the same offset from that page: `0xFEED0034`

Q21 (3 points) For this system, could you use a smaller PTE size? If so, select the smallest power of 2 size that would be appropriate and justify your decision.

> Our system supports 32 bit physical addresses, so our PPN must be 24 bits. We would also need at least a valid bit, which means our PTE would need to be at least 25 bits. We could fit this in a PTE that's only 4 bytes (32 bits). (You could justify we want at least 10 bits for flags and a PTE size of 8 would also work) The PTE does not need to be 16 bytes (128 bits).

Q22 (2 points) For your PTE size above, would the system see any benefits in terms of access time? Explain your answer.

> No, even with a PTE of 4 bytes, we could now have 64 PTE entries per page size (the index would be 6 bits). We'd still need 2 levels of page tables, so our access time would not improve.