ECE 344: Operating Systems

2023 Fall Midterm Exam

Instructor: Jon Eyolfson

November 15, 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 9 numbered pages (page 9 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 (3 points)

Why does the OS need to flush the TLB when context switching between processes?

> Because each process has their own virtual memory, each would have their own VPN to PPN mappings. If the TLB does not tag entries, the VPN to PPN mapping is not always the same between two processes, therefore we'd have to flush (clear) the TLB to be safe. Otherwise, processes may be able to accidentally access memory belonging to another process.

Q2 (2 points)

Does the OS need to flush the TLB when context switching between threads in same process? Why?

> No, threads running in the same process use the same virtual address space, and would have the same VPN to PPN mapping. Therefore, we do not need to flush the TLB.

Q3 (3 points)

How does the OS prevent your process from directly accessing hardware?

> The OS has a kernel, which is running in another privilege mode on the CPU (kernel mode). Kernel mode contains all the CPU instructions that let you indirect with hardware. Normal processes run in user mode which do have access to these instructions. In order to indirect with hardware processes have to do system calls.

Q4 (2 points)

Why can you not call (blocking) `waitpid` twice on the same process?

> After calling and returning from `waitpid` the first time, the kernel removes all resources associated with the child process, including the process ID (`pid`). If you call it a second time that process no longer exists, so you would get an error.

Q5 (4 points)

(1) If you only ever run a single application on an embedded system, do you need virtual memory? Briefly explain why or why not. (2) What benefit would you see if you did not use virtual memory.

> No, you don't need virtual memory. Virtual memory is mainly to provide isolation for a general purpose OS that needs to run multiple processes. If you're only running one application on an embedded system, it would give you a benefit because you would not need to manage page tables, and not need to do any address translation. Converting a VPN to PPN performs several lookups, which can be avoided (and we wouldn't need to use the TLB). Therefore, the performance of the application would improve without virtual memory.

Q6 (2 points)

In C, what is a change you can make that will change the ABI, but not the API?

> If you re-order fields of a `struct` you would not change the API, but you would change the ABI. The memory layout of the `struct` depends on the order of the fields.

Q7 (2 points)

What system call does "`printf("Hello world"); fflush(NULL);`" make, and what is the value of the first argument?

> `printf` eventually calls `write`. The value of the first argument would be 1 because that is standard output (`stdout`).

Q8 (2 points)

Why does Linux use FIFO and RR scheduling algorithms for soft real-time processes?

> Real-time scheduling needs to be predictable. Therefore, Linux uses simpler scheduling algorithms that are predictable and permformant.

Q9 (5 points)

Describe how you can use the output (from `stdout`) from one process (running `ls`) as input to another process (using `stdin`, running `wc`). Explain any system calls you'd need to make.

> You could create a pipe between the processes. You would need a process to create a pipe with a `pipe` system call. The process could then `fork` twice to create 2 new processes. After the `fork` we can change `stdout` to point to the write end of the pipe using `dup2` (we should also `close` the now unused file descriptors), in one child process, then `execve` to become `ls`. In the other child process we can change `stdin` to point to the read end of the pipe (using `dup2`, and likely also `close` as with the first process), then `execve` to become `wc`.

Q10 (5 points)

(1) Describe a situation where it would be beneficial to create multiple threads in your process, even though the machine only has a single core and cannot run anything in parallel. (2) Assume your process does a lot of I/O, would it be beneficial if the threads were kernel threads instead of user threads? Explain why or why not.

> One example where it would be beneficial is if we were writing a web server. Concurrency allows us to make progress on multiple tasks, if we switch between the tasks quickly enough it appears as parallelism. In a web server we have multiple tasks, one for each connection, so threads would make it easy to make progress on multiple connections. This example does use a lot of I/O, so we would also get a benefit if we used kernel threads. If one kernel thread gets blocked, we can still make progress on other threads because the kernel could execute them. If we only had user threads, one of the user threads blocking would cause the process to block, which would be much slower compared to kernel threads.

(15 points total) Processes.

Consider the following code that gets compiled into a program located at `build/processA`:

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

int main(void) {
    int x = 1;
    pid_t pid = fork();
    if (pid > 0) {
        ++x;
        pid = fork();
    }
    printf("%d: %d\n", getpid(), x); /* (Q12) */
    if (pid == 0) {
        execlp("build/processB", "processB", NULL);
    }
    wait(NULL);
    printf("Child processes done?\n"); /* (Q13) */
    return 0;
}
```

In addition, consider the following code that gets compiled into the program located at `build/processB`:

```
#include <stdio.h>
#include <unistd.h>

int main(void) {
    printf("%d: processB\n", getpid()); /* (Q12) */
    return 0;
}
```

Assume that calls to `fork`, `getpid`, `execlp`, and `wait` are always successful. If you are unfamiliar with the syntax of `execlp`, in this instance it'll just run the program `build/processB`.

In our terminal we execute the program `build/processA`, and the process gets assigned a process ID (`pid`) of 100. Answer the questions on the following page.

Q11 (2 points)
How many *new* processes get created (exclude `pid` 100)?

2.

Q12 (7 points)
After every process completes, show the output of each `printf` call with `(Q12)` in the comment. Ensure that your `pid` values are consistent. Is there any guaranteed ordering of `printf` statements between processes? If so, say what the order is.

```
100: 2
101: 1
101: processB
102: 2
102: processB
```

Between processes, there's no guaranteed ordering.

Q13 (2 points)
At the line `Q13`, what processes (indicate their `pid`) reach this line? For each process that reaches this line, are all its child processes guaranteed to be terminated?

Only process `100` reaches this line. It has two children, and only calls wait once. We'll only guarantee that one process terminates when we reach line `Q12`. Another child process may still be executing.

Q14 (2 points)
Can process `100` create orphan processes? If so, describe how.

Yes, it may. Process `100` only calls wait once, so one of process `101` or `102` terminates. If wait returns `pid` `101`, then process `102` could still be executing. We could continue executing `100`, it terminates which would mean process `102` is an orphan process.

Q15 (2 points)
Can process `100` create zombie processes? If so, describe how.

Yes, it may, but it's unlikely. Process `100` only calls wait once, so one of process `101` or `102` terminates. If wait returns `pid` `101`, then process `102` could still be executing. We could execute process `102` before `100`, and it could terminate. At this point process `102` is a zombie process.

(15 points total) Scheduling.

Consider the following processes you'd like to schedule:

Process	Arrival Time	Burst Time
P_1	0	4
P_2	0	1
P_3	2	4
P_4	10	4

Process P_1 arrives slightly before P_2.

You decide to use a round robin scheduler with a quantum length of 3 time units.

Q16 (7 points)
Fill in the boxes with the current running process for each time unit (some boxes may be unused).

0–1	1–2	2–3	3–4	4–5	5–6	6–7	7–8	8–9	9–10	10–11	11–12	12–13	13–14
P_1	P_1	P_1	P_2	P_3	P_3	P_3	P_1	P_3	idle	P_4	P_4	P_4	P_4

Q17 (4 points)
What's the average response time? (Your answer can be fractional.)

$$\text{Avg}_{\text{ResponseTime}} = \frac{0+3+2+0}{4} = \frac{5}{4} = 1.25$$

Q18 (4 points)
What's the average waiting time? (Your answer can be fractional.)

$$\text{Avg}_{\text{WaitingTime}} = \frac{4+3+3+0}{4} = \frac{10}{4} = 2.5$$

(15 points total) Page Tables.

Consider a system with a page size of 64 KiB (1 KiB = 2^{10} bytes). The PTE size is 16 bytes, and the system supports 128 bit physical addresses. Some truncated page tables (which fit on a page) are shown below:

Index	PPN	Valid
0	`0x3`	1
1	`0xE`	1
2	`0xF`	0

Page Table at `0xA0000`

Index	PPN	Valid
0	`0xB`	0
1	`0xD`	1
2	`0x6`	1

Page Table at `0xC0000`

Index	PPN	Valid
0	`0xD`	1
1	`0xA`	1
2	`0x4`	1

Page Table at `0xB0000`

Index	PPN	Valid
0	`0x5`	1
1	`0x9`	1
2	`0x8`	0

Page Table at `0xD0000`

Q19 (2 points)
How many PTEs can you fit into a single page? (Answer can be a power of 2.)

$\frac{2^{16}}{2^4} = 2^{12}$

Q20 (2 points)
With only a single-level page table (that fits on a page), what's the maximum size (in bits) of virtual address supported?

28 bit (12 bits for the index, plus 16 bits for the offset)

Q21 (2 points)
For the system with a single-level page table, assume the root page table is at PPN `0xA`. If we translate the virtual address `0x0017777`, what physical address do we get (or page fault)?

It would translate to `0xE7777`

Q22 (1 point)
For the same setup above (Q21), if we translate the virtual address `0x0027777`, what physical address do we get (or page fault)?

We would get a page fault because the valid bit is 0

Q23 (1 point)
With two levels of page tables (each page table fits on a page), what's the maximum size (in bits) of virtual address supported?

40 bit (12 bits for the L1 index, plus 12 bits for the L0 index, plus 16 bits for the offset)

Q24 (3 points)
For the system with a two-level page table, assume the root page table for process `100` is at PPN `0xB`. For process `100`, if we translate the virtual address `0x0017777`, what physical address do we get (or page fault)? If there's a page fault what lookup fails?

We would look at index 0 in page table `0xB0000` (L1). The PPN here is `0xD`, which means we use `0xD0000` as our L0 page table. At index 1 we find PPN `0x9`. Therefore, the physical address is `0x97777`.

Q25 (2 points)
Assume the same system in Q24. Also assume that the root page table for process `101` is at PPN `0xC`. For process `101`, if we translate the virtual address `0x0010024789`, what physical address do we get (or page fault)? If there's a page fault what lookup fails?

We would look at index 1 in page table `0xC0000` (L1). The PPN here is `0xD` (and valid), which means we use `0xD0000` as our L0 page table. At index 2 we find PPN `0x8`, but it's invalid. Therefore, we have a page fault in our L0 lookup.

Q26 (2 points)
Given our process `100` and `101` root page tables in Q24 and Q25 above, is there any issues if we want to make sure the processes have completely independent memory? Explain why or why not.

Yes, given the page tables above, process `100` and `101` actually share a L0 page table. Therefore, virtual addresses `0x000XXXYYYY` in process `100` would map to the same physical addresses as virtual addresses `0x001XXXYYYY` in process `101`. The processes would not be independent because of this.