2025 Winter Test 1

Course: ECE353: Systems Software
Examiners: Jon Eyolfson
Date: February 12, 2025
Duration: 45 minutes

Exam Type: A

A “closed book” examination.
No aids are permitted other than the information printed on the examination paper.

Calculator Type: 3

Non-programmable calculators from a list of approved calculators as issued by the Faculty Registrar.

Instructions:

Do not write answers on the back of pages as they will not be graded. Use the blank sheet at the end of the exam for extra space, and clearly indicate in the provided answer space if your response continues.

If a question seems unclear or ambiguous, state your assumptions and answer accordingly. In case of an error, identify it, provide a corrected version, and respond as if the question has been fixed.

Provide brief and specific answers. Clear and concise responses will receive higher marks compared to vague and wordy ones. Note that marks will be deducted for incorrect statements in your answers.

Functions

```c
int fork();
```

Creates a new process that's a clone of the currently running process. In the original process, it returns the process ID (`pid`) of the newly created child process. In the child process it returns `0`.

```c
int execlp(const char *file, const char *arg, ...);
```

Replaces the current process with a new program specified by `file`. The new process is given the command-line arguments specified by `arg` and any additional arguments. Returns only if there is an error.

```c
int dup2(int oldfd, int newfd);
```

Duplicates the file descriptor `oldfd` to `newfd`. If `newfd` is already open, it is first closed. Returns the new file descriptor on success.

```c
int waitpid(pid_t pid, int *status, int options);
```

Waits for a specific child process (`pid`) to change state. The state change is stored in `status`. The `options` argument can modify the behavior of `waitpid`, use `0` for the defaults (blocking). Returns the `pid` of the child process on success.

```c
int pipe(int pipefd[2]);
```

Creates a unidirectional data channel. `pipefd[0]` is set up for reading, and `pipefd[1]` is set up for writing. Returns 0 on success.

```c
void exit(int status);
```

Terminates the calling process with an exit status of `status`.

```c
ssize_t write(int fd, const void *buf, size_t count);
```

Writes `count` bytes from `buf` to the file or device associated with `fd`. Returns the number of bytes written.

```c
ssize_t read(int fd, void *buf, size_t count);
```

Reads up to `count` bytes from the file or socket associated with `fd` into `buf`. Returns the number of bytes read.

Short Answer (9 marks total)

Q1 (3 marks). Describe the role of the TLB, and why it's important.

The TLB acts as a cache for VPN lookups. It's important because the VPN lookups are much slower than accessing physical memory directly. We need the TLB to bring the performance of virtual memory closer to physical memory.

Q2 (3 marks). Explain the trade-offs if you had to decide whether to put device drivers in the kernel.

If we don't put the device drivers in the kernel, our system would have to perform many (slow) system calls in order to do common operating system tasks. However, the trade-off is our system may be more secure. When we run device driver code in user-mode and not kernel-mode, if it gets exploited it won't have access to the entire system.

Q3 (3 marks). When would you prefer to use static libraries over dynamic libraries? Also explain at what point you'd see significant benefits using dynamic libraries instead.

If we want to ensure our application always behaves the same, we would use static libraries. Static libraries copy the code into the executable itself, and would be only if there's only a handful of applications using the library. If many applications use the same function, it would be better to share it using a dynamic library (assuming it's well maintained, and updates do not break anything). It would also be okay to say if an application only uses a few functions from a library, just they would be included, without the need to deploy a large library as well (uncommon).

Processes (13 marks total)

Consider the following code, assume that all system calls are always successful, except `waitpid`, which may return -1 if the calling process has no children.

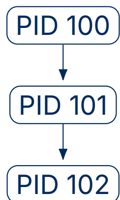
```
1  int main(void) {
2      int x = 1;
3      int pid = fork();
4      if (pid == 0) {
5          ++x;
6          pid = fork();
7          if (pid == 0) {
8              ++x;
9          }
10     }
11     printf("x: %d\n", x);
12     if (pid > 0) {
13         waitpid(pid, NULL, 0);
14     }
15     return 0;
16 }
```

Assume that process ID 100 begins running the `main` function.

Q4 (1 marks). How many **new** processes get created?

2.

Q5 (2 marks). Draw the process tree of parent/child relationships, showing the likely process IDs.



Q6 (4 marks). For every process created from running PID 100, state whether it will always, maybe, or never be a zombie or an orphan (one answer for each) **when PID 100 terminates**. Briefly justify your reasoning. Use the following format for your answers:

PID 101:
Zombie: Answer (Always/Maybe/Never). Justification.
Orphan: Answer (Always/Maybe/Never). Justification.

PID 101:

Zombie: Never. Since PID 100 waits for PID 101, it'll be cleaned up before PID 100 terminates.
Orphan: Never. PID 100 explicitly waits for PID 101 immediately after its creation, ensuring it is always acknowledged.

PID 102:

Zombie: Never. Since PID 101 waits for PID 102, and P100 waits for PID 101, it'll be cleaned up before PID 100 terminates.
Orphan: Never. PID 101 waits for PID 102 before terminating, ensuring it is always acknowledged.

Q7 (3 marks). Write a **single** set of possible print statements from all processes when running the `main` function. State whether this is the only possible ordering of print statements.

```
x: 3
x: 2
x: 1
```

Any ordering of the above 3 lines is possible.

Q8 (3 marks). Consider a non-preemptible FCFS (first come, first serve, or FIFO) scheduler Would there be a set order of print statements now from running the `main` function? Briefly justify your answer.

```
x: 1
x: 2
x: 3
```

We would always see the above ordering, because P100 would run until it prints `x: 1` and blocks for the `waitpid` call. After, PID101 would be next in line, printing `x: 2` and blocking for its `waitpid` call. Finally, P102 would print `x: 3`.

Scheduling (11 marks total)

Consider the following processes you'd like to schedule:

Process	Arrival Time	Burst Time
P_1	0	5
P_2	3	3
P_3	4	2

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

Q9 (5 marks). Fill in the boxes with the current running process for each time unit.

0–1	1–2	2–3	3–4	4–5	5–6	6–7	7–8	8–9	9–10
P_1	P_1	P_1	P_2	P_2	P_2	P_1	P_1	P_3	P_3

Q10 (2 marks). What's the average response time? (Your answer can be fractional.)

$$\text{Avg}_{\text{ResponseTime}} = \frac{0+0+4}{3} = \frac{4}{3} = 1.33$$

Q11 (2 marks). What's the average waiting time? (Your answer can be fractional.)

$$\text{Avg}_{\text{WaitingTime}} = \frac{3+0+4}{3} = \frac{7}{3} = 2.33$$

Q12 (2 marks). What could change to lower response time? Would there be any drawbacks, in general, with this change? Explain.

You could use a shorter quantum length (time slice) for a lower response time. However, there would be more context switches, in general.

Virtual Memory (12 marks total)

Consider a system using Sv39. Recall Sv39 has a 39-bit virtual address, a page size of 4096, a PTE size of 8 bytes, resulting in a 3 level page table.

Q13 (4 marks). Assume a process uses 1025 pages for **data**, how many pages does this require *at minimum* for **page tables**? Justify your answer by stating what the role is for each page.

> Since each L0 page table can hold 512 entries, we would need at minimum 3 to store 1025 entries for each data page. In the best case we can have a single L1 page table pointing to these 3 L0 page tables, and a single L2 page table pointing to the L1 page table. In total, we need **5** pages to store the page tables.

Q14 (4 marks). Assume a process uses 1025 pages for **data**, how many pages does this require *at maximum* for **page tables**? Justify your answer by stating what the role is for each page.

> The worst case is we have 1025 separate L0 page tables, each with a single entry to a data page. We can only have 512 L1 page tables, at most, because we only have 512 entries in our single L2 page table. Therefore, in the worst case we have 1 L2 page table full of entries to 512 L1 page tables. Across the L1 page tables, they point to our 1025 L0 page tables with a single entry each. Therefore, in total, we'd need **1 + 512 + 1025** or **1538** pages for the page tables.

Q15 (4 marks). Practically, what would cause the minimum case to happen, rather than the second? Briefly justify your answer.

> If a program uses virtual memory sequentially, then it's likely the L2 and L1 index are the same, so you'd only be adding entries sequentially in the L0 page tables. This works because we use the lowest VPN bits for the L0 index, so they'll be sequential as well.