2024 Fall Midterm

Course: ECE344: Operating Systems
Examiners: Jon Eyolfson, Tome Kosteski
Date: October 21, 2024
Duration: 1 hours 15 minutes (75 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

`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`.

`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.

`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.

`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.

`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.

`void exit(int status);`

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

`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.

`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.

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

Creates a new thread with attributes specified by `attr`. The new thread starts execution by invoking `start_routine` with `arg` as its argument. Returns 0 on success.

`void pthread_exit(void *retval);`

Terminates the calling thread, returning `retval` to any joining thread.

`int pthread_join(pthread_t thread, void **retval);`

Waits for the thread specified by `thread` to terminate. The thread's return value is stored in `retval`. Returns 0 on success.

`int pthread_detach(pthread_t thread);`

Detaches the specified thread, so that its resources can be reclaimed immediately upon termination. Returns 0 on success.

Short Answer (20 marks total)

Q1 (2 marks). Briefly describe why a real-time process contradicts a fairness policy in an operating system.

In any given fairness policy, each process is treated in an "equal" fashion. A real-time process will always run before any other process, and possibly strave other "normal" processes.

Q2 (2 marks). Describe one situtation that will cause the `write` system call to fail.

You could try to write to a file decsriptor that is already closed.

Q3 (2 marks). Describe the role of an MMU (Memory Management Unit).

The role of an MMU is to take the virtual address as an input and output the corresponding physical address in main memory.

Q4 (2 marks). What are the design requirements for an embedded system not to require virtual memory, even if you'd like to run multiple programs?

If you have complete control over all programs, you can manually assign programs to use non-overlapping parts of physical memory.

Q5 (2 marks). Explain a drawback for having all services controlled by the kernel?

You need to perform a system calls, which are slower than function calls. Also, having a large kernel means more of a change for bugs and programs that may exploit the kernel and take control of the system.

Q6 (2 marks). Would the standard C library be considered part of the operating system? Explain why or why not.

Yes, an operating system is the kernel plus all libraries for an application. Basically all programs use the standard C library at some level, so it would be considered part of the operating system.

Q7 (2 marks). Briefly describe what happens to an orphaned process.

An orphaned process will get re-parented, likely to `init`.

Q8 (3 marks). Why does an operating system decouple the process address space from a physical address space?

This is referred to as Virtual Memory and it allows for a greater degree of multiprogramming as there can be multiple programs running and each program gets use all of the physical address space.

Q9 (3 marks). Consider a round-robin scheduling algorithm that is implemented on a system that has a relatively long context switching time. Describe the trade-offs of using a large quanta versus a small quanta.

To reduce the cost of context switching overhead, large time quanta should be used than smaller quanta. However, you will also have a higher response time with a larger time quanta.

Processes (15 marks total)

Consider the following code, assume that all system calls are always successful.

```c
int main(void) {
    int fds[2];
    pipe(fds);
    int pid = fork();
    if (pid > 0) {
        char *str = "x";
        write(fds[1], str, 1);
        /* (A) */
    }
    else {
        /* (B) */
        char buffer[4096];
        int bytes_read;
        while ((bytes_read = read(fds[0], buffer, sizeof(buffer))) != 0) {
            printf("read: %.*s\n", bytes_read, buffer);
        }
        printf("done\n");
    }
    return 0;
```

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

Q10 (1 marks). How many new processes get created?

1.

Q11 (2 marks). Assume process 100 initially runs the main function, what do you expect it to print?

After forking, the parent process returns from `main` which implicitly calls `exit` and terminates the process. Nothing gets printed.

Q12 (2 marks). For each created process, what do you expect it to print? As a hint, you will never see `done` printed.

The created process would print:

```
read: x
```

Q13 (2 marks). For each created process, when process 100 exits, could they be an orphan, zombie, or both? Explain why.

Process 101 would be an orphan, it will not terminate, so it can't be a zombie.

Q14 (2 marks). Assume we add `waitpid(pid, NULL, 0);` instead of `/* (A) */` on line 8. Explain what behaviour we'd see for process 100.

Process 100 would not exit, it would block waiting on the created process to terminate.

Q15 (4 marks). What would you have to do so all processes exit?

You need to close the write end of the pipe in both processes. For the parent you must close it before `waitpid`. For the child you must close it before the `read` loop.

Q16 (2 marks). Now, assume all processes exit (hopefully from the solution in the previous question) and we add `fork();` instead of `/* (B) */` on line 11. Write a possible sequence of print statements you'd see, you do not have to specify which process each print comes from.

You'd either see:

```
read: x
done
done
```

or

```
done
read: x
done
```

Threads (10 marks total)

Consider the following code, assume that all system calls are always successful.

```c
void* run(void* p) {
    int id = *((int*) p);
    if (id == 4) {
        fork();
    }
    printf("run %d\n", id);
    return NULL;
}

int main(void) {
    pthread_t threads[4];
    int ids[4] = {1, 2, 3, 4};
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, run, &ids[i]);
        pthread_detach(threads[i]);
    }
    fork();
    for (int i = 2; i < 4; ++i) {
        pthread_create(&threads[i], NULL, run, &ids[i]);
        pthread_detach(threads[i]);
    }
    pthread_exit(NULL);
    return 0;
}
```

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

Q17 (1 marks). How many new processes get created?

3.

Q18 (2 marks). How many pthreads get created in total across all processes?

6. 8, is okay if you assume the copied pthreads due to the `fork`. 9, if you also considered the main thread.

Q19 (2 marks). Is it possible for any of the created threads to be "zombie" threads? Explain why or why not.

No, all threads are detached, which means they release their resources when they exit

Q20 (5 marks). Show one valid sequence of print statements after every thread exits.

The intention was to make `ids` a global variable. So given the code, the number may be garbage.

```
run 1
run 2
run 3
run 3
run 4
run 4
run 4
run 4
```

Scheduling (15 marks total)

Consider the following processes you'd like to schedule:

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

Process P_1 is an I/O bound process, it runs for 1 time unit then blocks for 2 time units.

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

Q21 (7 marks). Fill in the boxes with the current running process for each time unit (some boxes may be unused).

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

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

Q22 (3 marks). What's the average response time? (Your answer can be fractional.)

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

Q23 (3 marks). What's the average waiting time? (Your answer can be fractional.)

$$\text{Avg}_{\text{WaitingTime}} = \frac{7+4+7+4}{4} = \frac{22}{4} = 5.5$$

Q24 (2 marks). What technique could you use to ensure the I/O bound process gets scheduled immediately when it's unblocked?

You could add priorities, and make P_1 a high priority process.

Virtual Memory (15 marks total)

Consider a system with a page size of 256 bytes, and a PTE size of 1 byte. Some truncated page tables (which fit on a page) are shown below:

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

Page Table at `0xA00`

Index	PPN	Valid
0	`0x7`	1
1	`0xD`	0
2	`0x8`	1

Page Table at `0xC00`

Index	PPN	Valid
0	`0xC`	1
1	`0x5`	0
2	`0x4`	0

Page Table at `0xB00`

Index	PPN	Valid
0	`0xC`	0
1	`0xB`	0
2	`0xC`	1

Page Table at `0xD00`

Q25 (2 marks). How many different physical addresses could this system use? Justify your answer.

Assuming the PTE only stores a valid bit and PPN, the PPN could be at most 7 bits. Therefore the largest a physical address could be is 15 bits, or 2^{15} physical addresses.

Q26 (2 marks). How many PTEs can you fit into a single page? (Answer can be a power of 2.)

$\frac{2^8}{2^0} = 2^8 = 256$.

Q27 (2 marks). With only a single-level page table (that fits on a page), what's the maximum size (in bits) of virtual address supported?

16 bits (8 bits for the index, plus 8 bits for the offset).

Q28 (2 marks). For the system with a single-level page table, assume the root page table is at PPN `0xD`. If we translate the virtual address `0x210`, what physical address do we get (or page fault)?

It would translate to `0xC10`.

Q29 (1 marks). With two levels of page tables (each page table fits on a page), what's the maximum size (in bits) of virtual address supported?

24 bits (8 bits for the L1 index, plus 8 bits for the L0 index, plus 8 bits for the offset).

Q30 (2 marks). For the system with a two-level page table, assume the root page table for process `100` is at PPN `0xD`. For process `100`, if we translate the virtual address `0x210`, what physical address do we get (or page fault)? If there's a page fault what lookup fails?

We would get a page fault during the first L1 access. The L1 index would be 0. Therefore we'd look the entry at index 0 in `0xD00`, which is invalid.

Q31 (2 marks). For the same setup above, if we translate the virtual address `0x20010`, what physical address do we get (or page fault)?

We would look at index 2 in page table `0xD00` (L1). The PPN here is `0xC`, which means we use `0xC00` as our L0 page table. At index 0 we find PPN `0x7`. Therefore, the physical address is `0x710`.

Q32 (2 marks). Assume all entries not shown in the page tables are invalid, how many pages does process 100 use for data?

2. (4 if you also consider the L1 page table, and the L0 page table as data).