2024 Winter Midterm

Course: ECE353: Systems Software
Examiner: Jon Eyolfson
Date: February 26, 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 (10 marks total)

Q1 (2 marks). What is the name of the software component within an operating system that directly interacts with hardware?

The kernel.

Q2 (3 marks). Describe how shared libraries reduce memory usage in a multitasking operating system.

The operating system can share memory pages between processes as long as they're read-only. For instance, the operating system would only have to load the standard C library into memory once and share it between all processes that use that library.

Q3 (2 marks). Are signals in operating systems similar to polling? If not, explain the key differences between signals and polling as mechanisms for event handling.

No, signals are user-level interrupts. The process gets notified when an event occurs immediately.

Q4 (3 marks). In a scenario where a high-priority process depends on the output from a low-priority process, what potential issues could arise? Suggest a solution to address or mitigate these issues, ensuring the high-priority process is not adversely affected.

This is an instance of priority inversion, we should temporarily change the priority of the low-priority process to high-priority until there is no longer a dependency.

Processes (18 marks total)

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

```c
int main(void) {
    int fds[2] = {0};
    pipe(fds);

    int pid = fork();

    if (pid == 0) {
        dup2(fds[0], 0);
        dup2(fds[1], 1);
        execlp("cat", "cat", NULL);
        exit(1);
    }
    else {
        /* (A) */
    }

    return 0;
}
```

Q5 (2 marks). You compile and execute the program and your terminal appears to return immediately, why?

After forking, the parent process returns from `main` which implicitly calls `exit` and terminates the process.

Q6 (6 marks). Explain what happens in the child process. Recall that the `cat` program reads from `stdin` and writes to `stdout`. Your answer should include whether or not its possible for the process to be an orphan or zombie process.

The child process would block on the `read` system call waiting for data. However, there is no process that will write data to the pipe. The child will not see "end of file" because it still has the write end of the pipe open. In this case the child process will never terminate, so it will not become a zombie process. However, it will outlive the original parent process and become an orphan process.

Q7 (3 marks). Assume that we insert the following code at point `/* (A) */`:

```
waitpid(pid, NULL, 0);
```

What would you observe when running the executable in the terminal with this change? Why?

The terminal would just hang there forever. The child process will not terminate, and therefore the parent process will not terminate either.

Q8 (2 marks). Assuming there was a problem with the child process, what is something you can do to make it terminate? (You don't have to be specific, just a high-level explanation)

You can send a terminate (or kill) signal to the process, and the default signal handler will exit the process.

Q9 (5 marks). Assume that we insert the following code at point `/* (A) */`:

```
write(fds[1], "x", 1);
```

How would this change affect the execution of the child process? Describe what happens now in the child process. Specifically mention if you'd notice any difference in CPU usage using a program such as `htop`.

The child process would read an `x` from the read end of the pipe, then write an `x` to the write end of the pipe. There will be an infinite loop of reading a `x` then writing it. The CPU usage for the process will now be at 100%, while before the child process was blocked and would be at 0% CPU usage.

Threads (15 marks total)

Consider the following code:

```c
#define NUM_THREADS 4

void* even(void *p) {
    int i = *((int*) p);
    free(p);
    printf("even(%d)\n", i);
    return NULL;
}

void* odd(void *p) {
    int i = *((int*) p);
    free(p);
    printf("odd(%d)\n", i);
    return NULL;
}

int main(void) {
    pthread_t threads[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; ++i) {
        int *p = malloc(sizeof(int));
        *p = i;
        if (i % 2 == 0) {
            pthread_create(&threads[i], NULL, even, p);
        }
        else {
            pthread_create(&threads[i], NULL, odd, p);
            pthread_join(threads[i], NULL);
        }
    }
    return 0;
}
```

Q10 (2 marks). Why is it necessary to use `malloc` in the main thread to allocate memory for the integer pointer passed to `pthread_create`, rather than passing the address of the loop variable `&i` directly?

Because all threads share memory (they're all in the same address space), we need to allocate memory that only the created thread accesses.

Q11 (3 marks). Is there a guaranteed order in which the print statements from the even and odd threads will execute in the provided code? Explain your reasoning.

We will always see `odd(1)` printed before `odd(3)`. There's no ordering otherwise.

Q12 (3 marks). Can a "zombie thread" exist when the process terminates? Explain the concept of a "zombie thread" and how it might or might not apply in this context.

> Yes, the `even` threads may terminate before the `odd` threads. A "zombie thread" still consumes resources (at least a thread ID), and isn't freed until it's joined (or if it's detached).

Q13 (3 marks). Is it possible that not all `printf` statements are executed when running the program? Explain any scenarios where this might occur and the reasons behind it.

> Yes, after creating the `even` threads, we don't know when they execute. It could be the case that we see the `printf` statements from the `odd` threads, then the main thread exits and terminates the process.

Q14 (4 marks). Address the issues identified in the previous questions, including the potential for "zombie threads" and the possibility of not seeing all printf statements executed. Propose a solution that ensures all threads are properly managed and all `printf` statements are executed, without altering the order in which the threads run.

> There's two main solutions, we could either: detach the `even` threads and place `pthread_exit` before `return 0;` in the main thread, or after the for loop we can join on the `even` threads.

Scheduling (15 marks total)

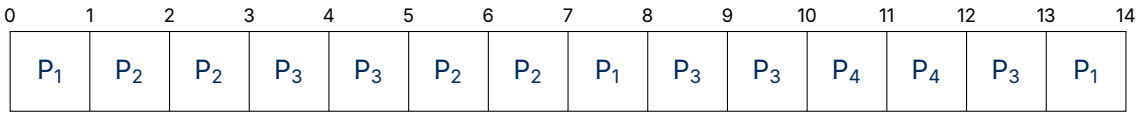
Consider the following processes you'd like to schedule:

Process	Arrival Time	Burst Time
P_1	0	3
P_2	1	4
P_3	2	5
P_4	6	2

Process P_1 is an I/O bound process, it runs for 1 time unit then blocks for 3 time units.

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

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



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

$$\text{Avg}_{\text{ResponseTime}} = \frac{0+0+1+4}{4} = \frac{5}{4} = 1.25$$

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

$$\text{Avg}_{\text{WaitingTime}} = \frac{5+2+6+4}{4} = \frac{17}{4} = 4.25$$

Virtual Memory (17 marks total)

We created an experimental machine that has 16 bit virtual addresses, 8 bit physical addresses, page size of 16 bytes, and a page table entry (PTE) of 1 byte.

Our PTE has the following layout where the physical page number (PPN or PFN) is the most significant 4 bits, and the permissions are the least significant 4 bits:

PPN	Execute? (X)	Write? (W)	Read? (R)	Valid? (V)
4 bits	1 bit	1 bit	1 bit	1 bit

Page Table Entry (PTE) Layout

Consider the following physical memory dump (add the row and the column for the physical address):

	0x00	**0x01**	**0x02**	**0x03**	**0x04**	**0x05**	**0x06**	**0x07**
0x00	0x1F	0x80	0xCF	0xAF	0x00	0x00	0x00	0x00
0x08	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x0F
0x10	0x00	0x90	0xFF	0x2F	0x00	0x00	0x00	0x00
0x18	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x20	0x0F	0x1F	0x4F	0x5F	0xFF	0x00	0x00	0x00
0x28	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x30	0x00	0x70	0xB0	0xDF	0x00	0x00	0x00	0x00
0x38	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x40	0x00	0x50	0x90	0x9F	0x00	0x00	0x00	0x00
0x48	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00

Example: the byte at address `0x24` is `0x3F`.

Q15 (2 marks). How many bytes would we need to store a single-level page table, if we used it?

$2^{12} = 4096$

For the remaining questions, you must use multi-level page tables. Assume that each page table fits in a physical page (or frame).

Q16 (2 marks). What is the minimum number of levels of page tables that we need in our system?

$\lceil \frac{16-4}{4} \rceil = \lceil \frac{12}{4} \rceil = 3$

We're going to translate the virtual address (VA) `0x34C`. Our highest level page table is currently at PPN `0x0`. Answer the following questions:

Q17 (1 marks). What is the value of page offset in the VA?

> `0xC` or 12.

Q18 (1 marks). What is the index (in decimal) of the highest level page table in the VA?

> `0x0`, or 0.

Q19 (1 marks). What is the index (in decimal) of the lowest level page table in the VA?

> `0x4`, or 4.

Q20 (4 marks). For the virtual address `0x34C`, detail every step of the address translation process, including the specific level of the page table and the PTEs accessed at each step. If a page fault occurs, identify the PTE responsible for the fault.

> The L2 page table starts at `0x00`, we access the PTE at index 0, which is the value `0x1F` (at address `0x00`). This means the L1 page table starts at `0x10`, we access the PTE at index 3 which is the value `0x2F` (at address `0x13`). This means the L0 page table starts at `0x20`, we access the PTE at index 4 which is the value `0xFF` (at address `0x24`). The resulting physical address is `0xFC`.

Q21 (4 marks). For the virtual address `0xFF2C`, detail every step of the address translation process, including the specific level of the page table and the PTEs accessed at each step. If a page fault occurs, identify the PTE responsible for the fault.

> The L2 page table starts at `0x00`, we access the PTE at index 15, which is the value `0x0F` (at address `0x0F`). The L1 page table starts at `0x00`, we access the PTE at index 15, which is the value `0x0F` (at address `0x0F`). The L0 page table starts at `0x00`, we access the PTE at index 2, which is the value `0xCF` (at address `0x02`). The resulting physical address is `0xCC`.

Q22 (2 marks). Explain the high-level changes required to support a 16-bit physical address space instead of an 8-bit one in this experimental machine.

> We would need to increase the size of the PTE to 2 bytes. We could use 12 bits of the PTE for the PPN, and the remaining 4 bits for the permissions.