2024 Fall Final

Course: ECE344: Operating Systems
Examiner: Jon Eyolfson, Tome Kosteski
Date: December 7, 2024
Duration: 2 hours 30 minutes (150 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:

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.

`int pthread_mutex_lock(pthread_mutex_t *mutex);`

Locks the specified mutex. If the mutex is already locked, the calling thread is blocked until the mutex becomes available. Returns 0 on success.

`int pthread_mutex_trylock(pthread_mutex_t *mutex);`

Attempts to lock the specified mutex. The function returns immediately, regardless of the mutex state. Returns 0 if the lock was acquired successfully, and a non-zero error code if it was not (e.g., already locked).

`int pthread_mutex_unlock(pthread_mutex_t *mutex);`

Unlocks the specified mutex. The mutex must be locked by the calling thread. Returns 0 on success.

`int sem_wait(sem_t *sem);`

Decreases the semaphore count. If the semaphore's value is zero, the calling process is blocked until the semaphore value is greater than zero. Returns 0 on success.

`int sem_post(sem_t *sem);`

Increases the semaphore count. If there are any processes or threads waiting on the semaphore, this operation wakes one of them. Returns 0 on success.

`int sem_trywait(sem_t *sem);`

Similar to `sem_wait`, but returns immediately if the decrement cannot be immediately performed (i.e., the semaphore value is zero). Returns 0 if the semaphore was successfully decremented, otherwise returns a non-zero error code.

`int pthread_cond_signal(pthread_cond_t *cond);`

Unblocks at least one of the threads that are blocked on the specified condition variable `cond`. Returns 0 on success.

`int pthread_cond_broadcast(pthread_cond_t *cond);`

Unblocks all threads currently blocked on the specified condition variable `cond`. Returns 0 on success.

`int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`

Blocks the calling thread on the condition variable `cond`. The thread remains blocked until another thread signals `cond` with `pthread_cond_signal` or `pthread_cond_broadcast`. The `mutex` is assumed to be locked by the calling thread on entrance to `pthread_cond_wait`. Before returning to the calling thread, `pthread_cond_wait` re-acquires `mutex`. Returns 0 on success.

`pid_t getpid(void);`

Returns the process ID (PID) of the calling process. This value can be used to uniquely identify the process within the system.

`int sched_yield(void);`

Causes the calling thread to relinquish the CPU. The thread is moved to the end of the queue and a new thread gets to run. Returns 0 on success.

Short Answer (25 marks total)

Q1 (1 marks). What tool should you use to determine what system calls a process makes?

`strace`

Q2 (2 marks). Describe a benefit of using a library function call versus a system call.

A library function call is fast, a system call requires switching from user space to kernel space that requires significant overhead.

Q3 (2 marks). Are new process ID (PID) numbers always larger than older running processes?

No, when a process is terminated, its PID is put back into a pool. The kernel will use the pool of PID numbers or generate a new number if none are available from the pool.

Q4 (2 marks). Briefly describe the role of the TLB.

The TLB is a cache for the PTE lookups of virtual pages.

Q5 (2 marks). Name two different mechanisms for IPC (inter-process communication).

Pipes, files, sockets, signals, shared memory

Q6 (2 marks). What is one type of information stored in a `ucontext_t`?

The register state (the value of all the registers), the next context to switch to if the run function returns, and stack information (base address and size).

Q7 (2 marks). Name two goals of CPU scheduling that can be in conflict.

Fairness and average waiting time.

Q8 (3 marks). Why might you prefer to use containers (e.g., Docker) over virtual machines to distribute an application?

For virtual machines, you'd have to distribute a kernel with your application, which would waste a lot of space. Containers share the host kernel while providing isolation, like with a virtual machine. They'd also be much faster to start, as they do not require booting a kernel.

Q9 (4 marks). What is a journaling file system, and provide an example of an issue it prevents.

A journaling file system records updates in a log (journal) before applying them to the main file system. This ensures that, in the event of a crash or power failure, the system can recover to a consistent state by replaying the log.

For instance, for deleting a file, you may have removed the entry in the directory, released the inode to the pool of free nodes, and then crashed. In this scenario, the data blocks would not be released. A journal would know that this operation is not complete during the next power on and go back to free the data blocks, ensuring a consistent state.

Q10 (5 marks). Using a buddy allocator managing 4096 bytes, we receive allocation requests in the following order: 2048 bytes, 512 bytes, and 1536 bytes. Describe what happens during these allocations, including the free lists after each successful allocation. Would a general heap allocator behave differently?

- 2048 bytes: The allocator splits the 4096 byte block into two 2048 byte blocks. One block is allocated, leaving one free entry in the 2048 byte list.
- 512 bytes: The remaining 2048 byte free block is split into two 1024 byte blocks. One 1024 byte block is further split into two 512 byte blocks. One 512 byte block is allocated, leaving the other free. After, we have one free entry in the 1024 byte list and one free entry in the 512 byte list.
- 1536 bytes: This allocation would be rounded up to 2048 bytes. There is no 2048 byte, or greater, block free, therefore the allocation would fail.

In this scenario, the general heap allocator would be able to successfully use all the space fulfill every allocation request.

Processes (18 marks total)

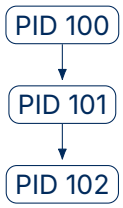
Consider the following code, assuming that all system call are always successful, running under process ID (`pid`) 100:

```
1  int main(void) {
2      /* Allocate a LOT of memory. */
3      int x = 4;
4      int pid = fork(); /* (A) */
5      if (pid > 0) {
6          waitpid(pid, NULL, 0);
7          printf("x = %d\n", x);
8      }
9      else {
10         x = 6;
11         pid = fork();
12         if (pid > 0) {
13             exit(0);
14         }
15     }
16     printf("x = %d\n", x);
17     return 0;
18 }
```

Q11 (2 marks). How many new processes get created?

2.

Q12 (2 marks). Draw the process tree of parent/child relationships, showing the process IDs they will likely get.



Q13 (4 marks). Provide a possible sequence of outputs from the provided code.

```
x = 6
x = 4
x = 4
```

Q14 (6 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 explicitly 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: Maybe. PID 102 could finish before PID 100. The opposite may also happen.
Orphan: Always. PID 100 only continues from line 6 after PID 101 terminates. PID 102 will always have to be re-parented. We'd also accept "Maybe" if it's clear it has to be re-parented.

Q15 (4 marks). Assume that before line 4 (also shown with the `/* (A) */` comment) the process uses 1 GiB (2^{30}) of memory. How can the kernel quickly create a "copy" of this memory during a `fork` without actually duplicating 1 GiB of data? What independent structures must the kernel create for the new process? Describe the implementation, including the kernel's actions to support virtual memory and how the new process's virtual memory relates to the physical memory.

The kernel would implement copy-on-write. The new process would need independent page tables and PTEs. The new process' PTEs would point to the same physical memory as the original process. As soon as either process actually modified memory, the kernel would make a new copy then.

Threads (22 points total)

Consider the following code:

```c
void* run1(void*) {
    printf("r1 from %d\n", getpid());
    return NULL;
}

void* run2(void*) {
    printf("r2 from %d\n", getpid());
    return NULL;
}

int main(void) {
    pthread_t t1;
    pthread_create(&t1, NULL, run1, NULL);
    pthread_detach(t1);
    fork();
    pthread_t t2;
    pthread_create(&t2, NULL, run2, NULL);
    pthread_detach(t2);
    printf("main from %d\n", getpid());
    pthread_exit(NULL);
    return 0;
```

Assume we run this program as process `pid = 100`, and all system calls are successful.

Q16 (2 marks). How many threads are created in total **only** from calls to `pthread_create`?

3.

For the following questions assume that `printf` does not use a global buffer, i.e. every `printf` call immediately does a `write` system call.

Q17 (3 marks).
How many **lines** will the program print when run? Provide only the total number of lines printed for each possible execution.

This program will always print 5 lines when run.

Q18 (3 marks). Describe any ordering between the printed lines.

This is no ordering between any lines.

Q19 (4 marks). Show a possible outcome of running this program.

```
r1 from 100
main from 100
r2 from 100
main from 101
r2 from 101
```

Q20 (3 marks). If we remove `pthread_exit`, how many **lines** will the program print when run? Provide only the total number of lines printed for each possible execution.

This program will print any number between 2 and 5 lines when run.

Q21 (3 marks). For this question assume that we used *user threads* instead of *kernel threads* (from `pthread`) for the original code listing with `pthread_exit`. Show a possible outcome of running this program that isn't possible with kernel threads. If it's not possible to get a different outcome, briefly explain why.

```
r1 from 100
r1 from 101
main from 100
r2 from 100
main from 101
r2 from 101
```

Q22 (4 marks). For this question, assume the original code using *kernel threads* again. The only difference is that now `printf` uses a global buffer. Assume that `printf` only does a single `write` system call printing all lines just before the process exits. Show a possible outcome of running this program that isn't possible with the non-buffering `printf` function. If it's not possible to get a different outcome, briefly explain why.

```
r1 from 100
main from 100
r2 from 100
r1 from 100
main from 101
r2 from 101
```

It's possible that r1 runs before the `fork`, filling the buffer with `r1 from 100`. After the fork the new process will have a copy of that buffer as well. This will cause us to see that line twice, once from each process. Students do not need to provide this explanation for full marks.

Locking (15 marks total)

Consider the following code.

```
1  static pthread_mutex_t buffer_mutex = PTHREAD_MUTEX_INITIALIZER;
2  static pthread_mutex_t socket_mutex = PTHREAD_MUTEX_INITIALIZER;
3
4  void code1(void) {
5      pthread_mutex_lock(&buffer_mutex);
6      pthread_mutex_lock(&socket_mutex);
7      use_buffer_and_socket(); /* Uses the buffer and socket */
8      pthread_mutex_unlock(&socket_mutex);
9      pthread_mutex_unlock(&buffer_mutex);
10 }
11
12 void code2(void) {
13     pthread_mutex_lock(&buffer_mutex);
14     use_buffer(); /* Uses just the buffer */
15     pthread_mutex_unlock(&buffer_mutex);
16 }
17
18 void code3(void) {
19     pthread_mutex_lock(&socket_mutex);
20     use_socket(); /* Uses just the socket */
21     pthread_mutex_unlock(&socket_mutex);
22 }
23
24 void code4(void) {
25
26
27
28     use_buffer_and_socket(); /* Uses the buffer and socket */
29
30
31
32 }
```

Assume that `mutex` calls never return an error.

Q23 (4 marks). Assume that `code1`, `code2`, and `code3` are called by *any* number of threads. Is it possible for the process to deadlock? Explain why or why not.

> No, it is not possible for the process to deadlock. `code1` is the only function whose critical section uses two locks, and they're always locked in the same order. Therefore, it's not possible to have a circular wait, and a deadlock cannot occur.

Q24 (6 marks). Assume now that `code1`, `code2`, `code3`, and `code4` are called by any number of threads. Add `pthread_mutex_lock` and `pthread_mutex_unlock` calls to `code4` such that: 1) the `use_buffer_and_socket` function is only called when both mutexes are held, 2) at the end of the function both mutexes are unlocked, and 3) a deadlock **may** occur. (Yes, you're creating a bug that may be difficult to find and fix). You may add your code directly to the previous page, or write your version of `code4` below.

```
pthread_mutex_lock(&socket_mutex);
pthread_mutex_lock(&buffer_mutex);
use_buffer_and_socket(); /* Uses the buffer and socket */
pthread_mutex_unlock(&buffer_mutex);
pthread_mutex_unlock(&socket_mutex);
```

Q25 (5 marks). Describe a sequence of four threads executing concurrently, where:

- Thread 1 calls `code1`
- Thread 2 calls `code2`
- Thread 3 calls `code3`
- Thread 4 calls a modified version of `code4`

Explain how the order of execution can lead to all threads deadlocking. Include the sequence of events and the specific conditions that cause the deadlock.

Thread 1 locks the `buffer_mutex`

Thread 4 locks the `socket_mutex`

Now:

- Thread 1 cannot make progress, lock held by thread 4
- Thread 2 cannot make progress, lock held by thread 1
- Thread 3 cannot make progress, lock held by thread 4
- Thread 4 cannot make progress, lock held by thread 1

Condition Variables (22 marks total)

Consider the following code:

```c
#define LAST_VALUE 3

static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static int counter = 0;

void* run1(void*) {
    while (1) {
        pthread_mutex_lock(&mutex);
        while (counter % 2 != 0) {
            pthread_cond_wait(&cond, &mutex);
        }
        printf("Even: %d\n", counter);
        ++counter;
        pthread_cond_signal(&cond);
        if (counter >= LAST_VALUE) {
            pthread_mutex_unlock(&mutex); /* (A) */
            return NULL;
        }
        pthread_mutex_unlock(&mutex);
    }
}

void* run2(void*) {
    while (1) {
        pthread_mutex_lock(&mutex);
        while (counter % 2 == 0) {
            pthread_cond_wait(&cond, &mutex);
        }
        printf("Odd: %d\n", counter);
        ++counter;
        pthread_cond_signal(&cond);
        if (counter >= LAST_VALUE) {
            pthread_mutex_unlock(&mutex); /* (B) */
            return NULL;
        }
        pthread_mutex_unlock(&mutex);
    }
}

int main(void) {
    pthread_t thread[2];
    pthread_create(&thread[0], NULL, run1, NULL);
    pthread_create(&thread[1], NULL, run2, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    return 0;
}
```

Assume that all system calls are successful.

Q26 (10 marks). Show a possible output of the code. If it is the only possible output, state so. Otherwise, provide one alternative output.

There is only one possible output:

```
Even: 0
Odd: 1
Even: 2
Odd: 3
```

Q27 (4 marks). Your friend modifies the original code and removes the call to `pthread_mutex_unlock` on line 17 (it also has `/* (A) */` on the same line). Does the program terminate? Briefly explain why or why not.

The program will not terminate in this case. The last time the "even" thread runs, it'll print `Even: 2`, increase the counter to 3, wake up the other thread, then terminate this thread. The other thread will not be able to acquire the mutex and return from `pthread_cond_wait`, it'll block forever.

Q28 (4 marks). A different friend modifies the original code and removes the call to `pthread_mutex_unlock` on line 34 (it also has `/* (B) */` on the same line). Does the program terminate? Briefly explain why or why not.

The program will terminate in this case. After the "even" thread prints `Even: 2`, it'll unlock the mutex and terminate. The "odd" thread can then return from `pthread_cond_wait`, print `Odd: 3`, then terminate with the mutex locked. However, since this is the last thread, it'll terminate, the main thread will return from the joins, and the process exits.

Q29 (4 marks). Would the original code always terminate if there are two threads running `run2` instead of one? Note, there would still be one thread running `run1` and the main thread would join on all three threads. Explain why it always terminates, or a situation where it will not terminate.

It may not terminate, consider both threads running `run2` are blocked doing a `pthread_cond_wait` when `counter = 2`. The other thread will increment counter to `3` then only wake up one thread. One thread will print out `Odd: 3`, increment the counter to `4` and terminate. The other thread will never be able to make it out of the inner **`while`** loop.

Semaphores (15 marks total)

Consider the following code:

```
1  static sem_t sem;
2
3  void initialize_semaphore(void) {
4      sem_init(&sem, 0, x);
5  }
6
7  void* run(void*) {
8
9
10     resource_acquire(r);
11
12
13     /* Use the resource */
14
15
16     resource_release(r);
17
18
19
20     perform_other_work();
21     return NULL;
22 }
```

You're given the task of protecting a resource the `run` function. This function may be called in parallel with any number of threads.

Q30 (10 marks). Using a single semaphore, ensure that only a maximum of 4 threads could use the resource in parallel. You'll need to insert `sem_post` and `sem_wait` calls, and provide an initial value (replacing `x`). You want to ensure that the `perform_other_work` function can run in parallel with any number of threads. Do not change the order of function calls in the `run` function. Write your answers on this page.

> Change `x` to `4`. Add `sem_wait` to line 9, and `sem_post` to line 17.

Q31 (5 marks). Assume that the `perform_other_work` function did not have to run after using the resource. Describe how you would change the code so that if the resource is busy, threads can still run in parallel without blocking. You do not have to write any code, you can state which function(s) you'd use.

> We could use a `sema_trywait`, if we did not successfully decrement then we can call the `perform_other_work` function (or a smaller part of it), and try again. Whenever you fail to decrement, go back to doing other work.

Page Replacement (18 marks total)

Assume the following accesses to physical page numbers:

5, 4, 3, 2, 3, 1, 2, 3, 5, 1, 2, 4

You have 4 physical pages in memory. Assume that all pages are initially on disk.

Q32 (10 marks). Use the clock algorithm for page replacement. Recall on a page hit, you'll set the reference bit to 1. For each access write all the pages in memory *after the access* in the boxes below. State the number of page faults after all the accesses.

5	4	3	2	3	1	2	3	5	1	2	4
5	5	5	5	5	***1***	1	1	1	1	1	1
	4	4	4	4	4	4	4	***5***	5	5	5
		3	3	3	3	3	3	3	3	3	***4***
			2	2	2	2	2	2	2	2	2

7 page faults.

Q33 (6 marks). Now, use the LRU algorithm for page replacement. All the other constraints are the same as the previous clock algorithm question. For each access write all the pages in memory *after the access* in the boxes below. State the number of page faults after all the accesses.

5	4	3	2	3	1	2	3	5	1	2	4
5	5	5	5	5	***1***	1	1	1	1	1	1
	4	4	4	4	4	4	4	***5***	5	5	5
		3	3	3	3	3	3	3	3	3	***4***
			2	2	2	2	2	2	2	2	2

7 page faults.

Q34 (2 marks). Briefly explain why we'd choose to implement the clock algorithm over LRU.

In practice LRU will be too slow because multiple updates need to happen on a page reference. For the clock algorithm only a single bit may change on a reference.

Filesystems (15 marks total)

Consider the following output from running `ls -li` (as a reminder the first column is the inode number):

```
 31 -rw-r--r-- 1 ece  ece   96 Dec 7 14:00 a.txt
 32 -rw-r--r-- 1 ece  ece 4000 Dec 7 14:01 b.txt
 64 lrwxrwxrwx 1 ece  ece    5 Dec 7 14:02 c.txt -> a.txt
128 lrwxrwxrwx 1 ece  ece    5 Dec 7 14:03 d.txt -> c.txt
```

Assume a filesystem with a block size of 4096 bytes, 4-byte block pointers, and 128-byte inodes. inodes have 12 direct pointers, 1 indirect pointer, 1 double indirect pointer, and 1 triple indirect pointer.

Q35 (2 marks). How many bytes are lost due to internal fragmentation for the regular files in the output?

In total there are 4096 bytes lost. 4000 from `a.txt` and 96 from `b.txt`.

Q36 (1 marks). How many I/O blocks are needed to store the content of `c.txt`?

0, the content would be stored on the inode itself.

Q37 (4 marks). Assume the user runs `cat d.txt`, assuming nothing is cached, how many I/O blocks need to be read from disk? Ignore reads to the directory. Describe the order of the block reads and what they contain.

1. Read inode 128 from inode block 3 (inode contains `c.txt`)
2. Read inode 64 from inode block 1 (inode contains `a.txt`)
3. Read inode 31 from inode block 0
4. Read file content block, pointed to by inode 31

(we'd also give full marks for 6 if the answer of the previous question was 1)

Q38 (5 marks). A file has 1050636 (or $12 + 2^{11} + 2^{20}$) **I/O blocks** of content. How many **index blocks** are needed, in total, to store the pointers using an inode? You may skip the final calculation if you don't have a calculator.

The first 12 pointers are stored on the inode itself. The next 2^{10} pointers would be stored on 1 single indirect block. We would need 1 double indirect block, that would point to 2^{10} more single indirect blocks, this stores 2^{20} pointers. We have 2^{10} pointers left. Finally, there would need to be 1 triple indirect block, pointing to 1 double indirect block, pointing to 1 single indirect block storing the last 2^{10} pointers. In total, we need 1029 (or $1 + 1 + 2^{10} + 3$) index blocks.

Q39 (3 marks). Assume a user ran the following commands:

```
mv a.txt e.txt
ln b.txt a.txt
```

Write the resulting (inode, name) pairs for *regular files* in the directory.

(32, `a.txt`)

(32, `b.txt`)

(31, `e.txt`)