2025 Winter Final

Course: ECE353: Systems Software
Examiner: Jon Eyolfson
Date: April 30, 2025
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_init(sem_t *sem, int pshared, unsigned int value);`

Initializes a semaphore. The `pshared` argument determines whether the semaphore is shared between processes (pshared = 1) or only within a single process (pshared = 0). The value argument sets the initial count of the semaphore. Returns 0 on success and a non-zero error code on failure.

`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 (20 marks total)

Q1 (2 marks). Describe a way to reduce the number of system calls in your application.

Use buffering to group multiple operations (e.g., writes) into a single system call. For example, writing to a memory buffer and flushing it with one `write()`.

Q2 (2 marks). What happens to a process if it calls `read` on a pipe that has no data and no writers?

The process receives EOF (`read` returns 0) because there is no more data possible.

Q3 (2 marks). Name two differences between one-to-one mapped threads (kernel) and many-to-one mapped threads (user).

Kernel threads are scheduled by the OS, while user-level threads are scheduled by a library. Kernel threads can run in parallel on multicore systems; user threads cannot unless mapped to kernel threads.

Q4 (2 marks). What is priority inversion, and how can it be avoided?

Priority inversion happens when a low-priority process holds a resource needed by a high-priority process. It can be avoided using priority inheritance, which temporarily boosts the low-priority processes' priority.

Q5 (2 marks). Why does RAID 0 offer no fault tolerance, and what is its primary benefit?

RAID 0 stripes data across multiple disks without any redundancy. If one disk fails, all data is lost. Its primary benefit is increased performance due to parallel access to multiple disks.

Q6 (2 marks). Name 2 things stored in an inode itself.

Metadata such as file size, ownership, permissions, timestamps, link count, and pointers to data blocks.

Q7 (2 marks). What does it mean for a file system to be consistent after a crash?

It means that metadata and file contents are in a valid state, without incorrect bitmaps, orphaned blocks or incorrect inodes.

Q8 (2 marks). Why is a container not a virtual machine?

A container shares the host OS kernel (or the same single VM kernel), while a VM emulates an entire OS including the kernel.

Q9 (4 marks). In a buddy allocator managing a single 1024 byte free block initially, what happens when an allocation request for 200 bytes is received? Describe the splitting behavior and resulting free list state.

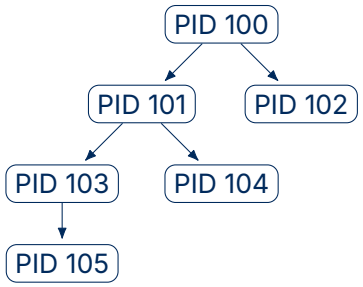
The allocator would round the allocation to the nearest power of 2, which is 256 bytes. It splits larger blocks recursively: 1024 to 512 to 256. The 256 byte block is allocated, and the remaining 512 byte and 256 byte buddies are added to their respective free lists.

Processes (20 marks total)

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

```
1  int main(void) {
2      int x = 1;
3      int pid = fork();
4      fork();
5      if (pid > 0) {
6          ++x;
7          waitpid(pid, NULL, 0);
8      }
9      else {
10         ++x;
11         pid = fork();
12         if (pid > 0) {
13             ++x;
14             waitpid(pid, NULL, 0);
15         }
16         printf("x: %d\n", x);
17     }
18     return 0;
19 }
```

Q10 (5 marks). Draw the process tree of parent/child relationships, showing the process IDs they will likely get.



Q11 (5 marks). Provide **all** possible sequence of outputs of `printf` calls from the provided code after all processes terminate (ignore cases where different processes print the same thing in different orders).

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

and

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

Q12 (4 marks). Using the process IDs from your process tree, describe a situation where a child process becomes a zombie after its parent terminates. If this is not possible, explain why.

> Process 102 may become a zombie. Process 100 creates 102 from the `fork()` on line 4, the `pid` variable is still `101`. Process 102 runs and enter the if statement and get an error from `waitpid` on line 7. It may then terminate immediately before process 100 even continues.

Q13 (4 marks). Using the process IDs from your process tree, describe a situation where a child process becomes an orphan after its parent terminates. If this is not possible, explain why.

> Process 102 may also become an orphan. Process 100 creates 102 from the `fork()` on line 4, the `pid` variable is still `101`. The OS may not run process 102, and process 100 terminates after waiting for process 101 to terminate. At this point process 102 is an orphan.

Q14 (2 marks). Is the read-only code segment (e.g., the text section) duplicated in memory when a process calls `fork()`? Explain why or why not, and how the operating system handles this situtation.

> Since the pages storing the code are read-only they can safely be mapped to the same physical memory in all processes (it's shared between them).

Threads (20 points total)

Consider the following code:

```c
void* run3(void* _);

static int x = 0;

void* run1(void* _) {
    x = 4;
    return NULL;
}

void* run2(void* _) {
    pthread_t t3;
    /* (A) */
    pthread_create(&t3, NULL, run3, NULL);
    pthread_detach(t3);
    /* (B) */
    printf("run2 x = %d\n", x);
    return NULL;
}

void* run3(void* _) {
    /* (C) */
    printf("run3 x = %d\n", x);
    return NULL;
}

int main(void) {
    pthread_t t1;
    pthread_create(&t1, NULL, run1, NULL);
    pthread_join(t1, NULL);
    pthread_t t2;
    pthread_create(&t2, NULL, run2, NULL);
    pthread_detach(t2);
    printf("main x = %d\n", x);
    return 0;
}
```

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

Q15 (2 marks). How many threads are created in total **only** from calls to `pthread_create`?

It depends, sometimes 2 and sometimes 3.

Q16 (3 marks). Is it possible for the only output to be `main x = 4` after the process terminates? Justify your answer. If so, what change would ensure multiple `printf` outputs always appear?

Yes, `t2` may not even execute before the main thread prints and terminates the process. We would add a `pthread_exit(NULL);` before `return 0;`.

Q17 (3 marks). Provide a **single** possible sequence of outputs of `printf` calls from the provided code after the process terminates (ignoring the previous scenario of just `main x = 4`).

```
run2 x = 4
run3 x = 4
main x = 4
```

only `main x = 4` must be there, the rest may be there (in any order)

Q18 (3 marks). Does this program contain a potential data race? Justify your answer.

No, it does not. The only write (line 6) occurs not concurrently with any other writes or reads.

Q19 (3 marks). Does this program contain a data race if we replace `/* (A) */` by `++x`? Justify your answer.

Yes, the write (now on line 12), can happen concurrently with the read from the main thread at line 33.

Q20 (3 marks). Does this program contain a data race if we replace `/* (B) */` by `++x` and `/* (C) */` by `++x`? Justify your answer.

Yes, the two writes from the replaced lines can occur concurrently with each other.

Q21 (3 marks). Can a thread become a "zombie" at any point during this program's execution? Explain your answer.

No, the main thread joins `t1` before it's possible to terminate the process and the other threads are detached. This means they'll be cleaned up whenever they terminate.

Locking (20 marks total)

The following code transfers money between two accounts. Each account has a unique ID, a balance (which can go negative), and a mutex to protect its data.

```
struct account {
    int id;
    int balance;
    pthread_mutex_t mutex;
};

void transfer(struct account *from, struct account *to, int amount) {
    pthread_mutex_lock(&from->mutex);
    pthread_mutex_lock(&to->mutex);

    from->balance -= amount;
    to->balance += amount;

    pthread_mutex_unlock(&to->mutex);
    pthread_mutex_unlock(&from->mutex);
}
```

For all questions assume that multiple threads may call `transfer`, all pointers are valid, and the balance may go negative.

Q22 (4 marks). What is one potential problem with the code? Describe a scenario where it could occur.

> A deadlock may occur, we may have a transfer from account 1 to 2, concurrently with another transfer from account 2 to account 1. The first transfer locks the mutex for account 1, then the second transfer locks the mutex for account 2.

Q23 (10 marks). Modify the `transfer` function below to ensure it has no issues. Assume that both mutexes must still be held during the transfer operation (decrement and increment).

One example breaking circular wait is:

```
void transfer(struct account *from, struct account *to, int amount) {
  if (from == to) { return; }
  if (from->id < to->id) {
      pthread_mutex_lock(&from->mutex);
      pthread_mutex_lock(&to->mutex);
  }
  else {
      pthread_mutex_lock(&to->mutex);
      pthread_mutex_lock(&from->mutex);
  }

  from->balance -= amount;
  to->balance += amount;

  pthread_mutex_unlock(&from->mutex);
  pthread_mutex_unlock(&to->mutex);
}
```

Q24 (6 marks). Is it possible to modify the transfer function so that it only locks one of the two accounts at a time? Justify your answer. If it is possible show the code below, and describe a trade-off.

Yes, it is possible, for example:

```
void transfer(struct account *from, struct account *to, int amount) {
  if (from == to) { return; }

  pthread_mutex_lock(&from->mutex);
  from->balance -= amount;
  pthread_mutex_unlock(&from->mutex);

  pthread_mutex_lock(&to->mutex);
  to->balance += amount;
  pthread_mutex_unlock(&to->mutex);
}
```

However, we may see some inconsistent values across all accounts which wouldn't be possible if we treated them like transactions. We would also have more potential parallelism, as we only need one lock at a time (students just need to describe one trade-off).

Synchronization (25 marks total)

The following code initializes an array of hardware:

```
struct hardware {
    void (*init)(void);


};

struct hardware_config {
    int count;
    struct hardware *hardware;




};

void initialize_remaining_config(struct hardware_config *config) {









}

void* run(struct hardware_config *config) {

    for (int i = 0; i < config->count; ++i) {
        struct hardware *hw = &config->hardware[i];

        hw->init();

    }

    /* All hardware should be initialized before continuing here. */
    remaining_code();
    return NULL;
}
```

Previously, this function ran on a single-core system without threads.

This function has now been moved into a multi-threaded environment and may be called by any number of threads on a machine with an unknown number of cores. Each thread receives the same **`struct hardware_config`** as input. You may add additional fields and initialize them in the `initialize_remaining_config` (this function runs once sequentially before any `run` function).

Your task is to improve the `run` function such that:

1. Each hardware component's `init()` function is called exactly once
2. The call to `remaining_code();` occurs only after all hardware has been initialized.

Q25 (25 marks). Modify the code directly, or clearly describe your changes below.

For task 1, you can use semaphores. You would have a semaphore for each `struct hardware` initialized to 1. In the for loop in the run function, only run the `init` function if `sem_trywait` successfully decrements (returns 0). You can also do this with mutexes.

For task 2, you can use condition variables. Add a mutex, a condition variable, and a variable such as `int intitialized_count;`. After running the `init` function from task 1: lock the mutex, increment the `initialized_count`, and if `initialized_count == config->count` then call `pthread_cond_broadcast`, finally unlock the mutex.

Before the `remaining_code()` call, add the following code:

```
pthread_mutex_lock(&config->mutex);
while (config->initialized_count < config->count) {
    pthread_cond_wait(&config->cond, &config->mutex);
}
pthread_mutex_unlock(&config->mutex);
```

Page Replacement (18 marks total)

Assume the following accesses to physical page numbers:

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

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

Q26 (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.

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

9 page faults.

Q27 (6 marks). Now, use the optimal 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.

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

6 page faults (last replacement could be any page).

Q28 (2 marks). What hardware support is required to implement the clock algorithm efficiently?

The MMU would have to implement updating the reference bit (the reference bit is stored in the PTE).

Filesystems (17 marks total)

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.

Q29 (5 marks). What is the maximum file size this filesystem can support using a single inode? Show how each level of indirection contributes to the total size (you do not have to simplify your answer).

$(12 + 2^{10} + 2^{20} + 2^{30}) \times 2^{12}$

Q30 (2 marks). Why is triple indirection rarely used in practice?

Files are rarely larger than approximately 4 GiB (except for video files).

Q31 (4 marks). Explain how reading a byte from the end of a large file stored with triple indirection differs in I/O cost compared to a small file.

For a small file (fitting within the direct pointers of the inode), the inode points directly to the data block, so only one block read is needed to read the byte (assuming the inode is already in memory).

For a large file that requires triple indirection, the inode points to a triple indirect block, which points to a double indirect block, which points to a single indirect block, which finally points to the data block. This requires reading 4 separate blocks.

Assume you have a file whose filename is `course.txt` with the content:

```
ECE353
```

After running the following commands:

```
ln course.txt alt.txt
rm course.txt
```

Q32 (2 marks). what is the content of `alt.txt`?

`ECE353`

Q33 (4 marks). After running `rm course.txt`, what changes occur to the file's inode, data blocks, and its presence in the file system?

The directory entry for `course.txt` is removed.

The inode's link count is decremented (from 2 to 1, since `alt.txt` still exists).

The inode and data blocks are not deleted because the link count is still non-zero.

Virtual Memory (10 marks total)

Consider a system with a page size of 16 KiB (1 KiB is 2^{10} B), a PTE size of 8 bytes, and 3 levels of page tables.

Q34 (5 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.

> For this system, each page table can hold 2048 entries ($\frac{2^{14}}{2^3} = 2^{11}$). Since each L0 page table can hold 2048 entries, we would only need 1 to store 1025 entries for each data page. In the best case we can have a single L1 page table pointing to this L0 page table, and a single L2 page table pointing to the L1 page table. In total, we need **3** pages to store the page tables.

Q35 (5 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 also have 1025 separate L1 page tables, each with a single entry to an L0 page table. We also need one L2 page table. Therefore, in the worst case we have 1 L2 page table with 1025 entries to 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 + 1025 + 1025** or **2051** pages for the page tables.