2024 Winter Final Examination

Course:	ECE353: Systems Software
Examiner:	Jon Eyolfson
Date:	April 30, 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

```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.

```c
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.

```c
void pthread_exit(void *retval);
```

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

```c
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.

```c
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 (7 marks total)

Q1 (2 marks). What is the difference between a joinable thread and a detached thread?

A detached thread releases all its resources when it terminates, a joinable thread must be joined by another thread.

Q2 (1 marks). With 8 identical drives, which RAID level should you choose to ensure no data loss if up to 2 drives fail simultaneously?

RAID 6 (RAID 1, or RAID10 is also acceptable).

Q3 (2 marks). How does a buddy allocator merge free blocks during deallocation?

The buddy allocator checks if the buddy (adjacent block of the same size) of the freed block is also free. If both are free, they are merged to form a larger block. This process is repeated recursively.

Q4 (2 marks). Do virtual machines directly access physical memory? If not, what does?

No, virtual machines do not directly access physical memory. They operate with virtual memory provided by the hypervisor, which maps virtual memory for the VM to physical memory (technically a type 2 hypervisor would not access physical memory, just saying hypervisor is also fine).

Threads (22 points total)

Consider the provided code which handles multiple socket connections (you do not need to know how sockets work), each independent connection runs in a new thread. The key functions include `next_index` for server selection and `run_task` for running the task. We want this code to cycle through all servers in a fair and orderly manner. Also, we want to ensure that **each** server executes the `run_task` function no more than once at any given time, while allowing for simultaneous `run_task` executions on different servers.

```c
#define NUM_SERVERS 4

struct server {
    const char* name;

};

int server_index = 0;
struct server servers[NUM_SERVERS];

int next_index(void) {
    int index = server_index;
    ++server_index;
    if (server_index >= NUM_SERVERS) {
        server_index = 0;
    }
    return index;
}

void* run(void* pfd) {
    /* We're guaranteed each thread gets a unique task_id. */
    int task_id = get_task_id(pfd);
    int index = next_index();
    printf("Start task %d on %s\n", task_id, servers[index].name);
    /* Runs a task in this thread, `run_task` does not access any global
       data. */
    run_task(task_id, &servers[index]);
    printf("End task %d on %s\n", task_id, servers[index].name);
    return NULL;
}

int main(void) {
    int sock = create_socket();
    while (1) {
        int fd = accept(sock, NULL, NULL);
        int* pfd = malloc(sizeof(int));
        *pfd = fd;
        pthread_t thread;
        pthread_create(&thread, NULL, run, pfd);
        pthread_detach(thread);
    }
    return 0;
}
```

An example output should look like:

```
Start task 1 on server0
Start task 2 on server1
Start task 3 on server2
Start task 4 on server3
End task 1 on server0
End task 2 on server1
End task 3 on server2
End task 4 on server3
Start task 5 on server0
End task 5 on server0
```

This output would not be valid:

```
Start task 1 on server0
Start task 2 on server0
End task 1 on server0
End task 2 on server0
```

Q5 (10 marks). Examine the `next_index` function in the provided code for potential data races. Is there a data race when updating the `server_index` variable? If yes, propose a solution to fix it. If no, provide a rationale supporting its thread safety. Reference specific line numbers or annotate directly in the code for clarity. You do not have to write initialization code for new variables, if they require a value to initialize them just state what it is.

Create a global mutex after line 8, call it `m`.

Lock `m` before line 12.

Unlock `m` after line 16.

Q6 (10 marks). Analyze if it's possible for a server to be executing two `run_task` calls simultaneously in the given implementation. If this scenario is possible, suggest modifications to ensure that `run_task` runs only once per server at any given time. If not, explain why the current setup prevents multiple concurrent `run_task` executions for the same server. Reference specific line numbers or annotate directly in the code for clarity. You may add fields to the `server` **`struct`** on line 5. You do not have to write initialization code for new variables, if they require a value to initialize them just state what it is.

Create a mutex for each server on line 5, call it `mutex`.

Lock the mutex for the server `&servers[index].mutex` before line 24.

Unlock the mutex for the server after line 28.

Q7 (2 marks). Assume that the task runs arbitrary code, is there a better approach to take rather than using threads? Briefly explain the better approach.

Have the task run in a new process instead of the thread.

Processes (25 marks total)

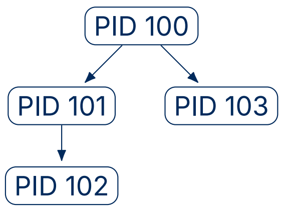
Consider the following code, assuming that all system call are always successful, running under process ID (`pid`) 100:

```c
int main(void) {
    for (int i = 0; i < 2; ++i) {
        int ret = fork();
        if (i == 0 && ret > 0) {
            waitpid(ret, NULL, 0);
        }
        printf("pid: %d, ret: %d, i: %d\n", getpid(), ret, i);
    }
    return 0;
}
```

Q8 (2 marks). How many new processes get created?

3.

Q9 (3 marks). Draw the process tree of parent/child relationships, showing the process IDs they will likely get.



Q10 (5 marks). Using the PIDs in your tree, provide a possible sequence of outputs from the provided code.

```
pid: 101, ret: 0, i: 0
pid: 101, ret: 102, i: 1
pid: 102, ret: 0, i: 1
pid: 100, ret: 101, i: 0
pid: 100, ret: 103, i: 1
pid: 103, ret: 0, i: 1
```

Q11 (5 marks). Show an impossible set of outputs for the provided code. Ensure that the sequence of prints within the same process remains consistent, but that the overall sequence cannot be produced by the code.

```
pid: 100, ret: 101, i: 0
pid: 101, ret: 0, i: 0
pid: 101, ret: 102, i: 1
pid: 102, ret: 0, i: 1
pid: 100, ret: 103, i: 1
pid: 103, ret: 0, i: 1
```

Process 100 always waits for process 101 to finish completely, so it cannot print before it.

Q12 (10 marks).
For each direct child process of PID 100, state whether it will always, maybe, or never be a zombie or an orphan (one answer for each). 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: Maybe. PID 101 could become a zombie if it terminates before PID 100 calls `waitpid()` on it, which it is programmed to do, thus only possibly becoming a zombie temporarily.
Orphan: Never. PID 100 explicitly waits for PID 101 immediately after its creation, ensuring it is always acknowledged.

PID 103:

Zombie: Maybe. PID 103 could finish before PID 100. The opposite may also happen.
Orphan: Maybe. If PID 100 terminates before PID 103. As above the opposite may also happen.

Locking (18 marks total)

Consider the following function that takes two mutex pointers and attempts to lock them. Assume that this function is called by multiple threads.

```
void lock_two_mutexes(pthread_mutex_t *mutex1, pthread_mutex_t *mutex2) {
    pthread_mutex_lock(mutex1);
    pthread_mutex_lock(mutex2);
    /* Critical section */
    pthread_mutex_unlock(mutex2);
    pthread_mutex_unlock(mutex1);
}
```

Q13 (2 marks). What would happen if you make a function call to `lock_two_mutexes` with the same mutex pointer for both arguments (i.e., `lock_two_mutexes(mutex1, mutex1)`)? Explain the behavior of the default pthread mutex in this scenario.

The caller thread would get deadlocked. You cannot lock and already locked mutex.

Q14 (6 marks). Describe how a deadlock might occur in the scenario that the mutexes are always different for a single function call (the same mutex may be used by different threads).

A deadlock can occur if two threads call this function simultaneously with their mutexes passed in opposite orders. For instance, suppose Thread 1 calls `lock_two_mutexes(mutex1, mutex2)` and Thread 2 calls `lock_two_mutexes(mutex2, mutex1)`. If Thread 1 successfully locks `mutex1`, then context switches to Thread 2, which locks `mutex2`, then both threads will wait indefinitely for each other to release the mutexes they need.

Q15 (8 marks). Modify the `lock_two_mutexes` function provided to resolve potential deadlocks without using a partial order based on the mutexes' addresses or any similar ordering strategy. In other words, describe a solution that would work regardless of the order in which mutexes are locked. You may assume the mutexes are always different for the same function call. Briefly explain how your modification prevents deadlock.

A solution to prevent deadlock in the given scenario is to implement a trylock:

```
pthread_mutex_lock(mutex1);
while (pthread_mutex_trylock(mutex2)) {
    pthread_mutex_unlock(mutex1);
    sched_yield();
    pthread_mutex_lock(mutex1);
}
/* Critical section */
pthread_mutex_unlock(mutex2);
pthread_mutex_unlock(mutex1);
```

This strategy ensures that if a thread cannot acquire both mutexes than it gives up the first mutex and yields to another thread while holding no locks, that way another thread can make progress.

Q16 (2 marks). Which of the necessary conditions for deadlock does this modification primarily address?

Hold and wait.

Semaphores (25 marks total)

You're given 4 threads (that get properly created and run) and you want to ensure ordering between them. You decide to use semaphores to accomplish your task. Recall that semaphores, after initialization, use `sem_post(sem_t* sem)` to increment the value and `sem_wait(sem_t* sem)` to decrement the value (waiting until the value is greater than 0). Assume no errors occur, so you never need to check return values. Consider the following code:

```c
/* Global variables: declare sem1, sem2, and sem3 */
void initialize() {
  sem_init(&sem1, 0,  );
  sem_init(&sem2, 0,  );
  sem_init(&sem3, 0,  );
}
void* thread1(void*) {


  f1();


}
void* thread2(void*) {


  f2(); /* only runs after f1 completes */


}
void* thread3(void*) {


  f3(); /* only runs after f1 completes */


  f4();


}
void* thread4(void*) {


  f5(); /* only runs after f2 and f3 complete */


  f6(); /* only runs after f4 completes */


}
```

Q17 (15 marks). Fill in the initial values for each semaphore and insert `sem_post` and `sem_wait` calls in a way that ensures the ordering in the comments. You cannot change the ordering of the `f()` calls, and they always execute in the order written. Write your answers on this page.

Q18 (10 marks). For each function state which functions *could* run in parallel with it (not including itself). Write “none” if it can only run by itself.

f1: none

f2: f3, f4

f3: f2

f4: f2, f5

f5: f4

f6: none

Solution to the previous question

```
/* Global variables: declare sem1, sem2, and sem3 */
void initialize() {
  sem_init(&sem1, 0, 0);
  sem_init(&sem2, 0, 0);
  sem_init(&sem3, 0, 0);
}
void* thread1(void*) {
  f1();
  sem_post(&sem1);
  sem_post(&sem1);
}
void* thread2(void*) {
  sem_wait(&sem1);
  f2();
  sem_post(&sem2);
}
void* thread3(void*) {
  sem_wait(&sem1);
  f3(); /* only runs after f1 completes */
  sem_post(&sem2);
  f4();
  sem_post(&sem3);
}
void* thread4(void*) {
  sem_wait(&sem2);
  sem_wait(&sem2);
  f5(); /* only runs after f2 and f3 completes */
  sem_wait(&sem3);
  f6(); /* only runs after f4 completes */
}
```

Page Replacement (18 marks total)

Assume the following accesses to physical page numbers:

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

You have 3 physical pages in memory. Assume that all pages are initially on disk.

Q19 (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.

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

7 page faults.

Q20 (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.

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

5 page faults.

Q21 (2 marks). Briefly explain why the clock algorithm might be implemented instead of the Least Recently Used (LRU) algorithm.

The clock algorithm is often implemented instead of LRU due to its lower overhead on memory accesses. LRU must perform an expensive update on each access, the clock algorithm only updates a single bit.

Filesystems (15 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.

Q22 (3 marks). Calculate the total bytes needed for 100 files, each 100 bytes in size. Include the size of the inodes in your calculation. You may skip the final calculation if you don't have a calculator.

Each file requires one inode (128 bytes) and one block (4096 bytes), there we need 422 400 bytes in total to store the 100 files.

Q23 (2 marks). How many bytes are lost due to internal fragmentation for the files described?

Internal fragmentation occurs because each file only uses 100 bytes of a 4096 byte block. Each file wastes 4096 - 100 = 3996 bytes. For 100 files, we lose 399 600 bytes due to internal fragmentation.

Q24 (3 marks). What is the maximum file size supported by the triple indirect block?

Each block can fit 2^{10} pointers, therefore with the triple indirect block we can point to $(2^{10})^3$ or 2^{30} blocks. Since each block is 2^{12} the maximum file size for the triple indirect block is 2^{42}, or 4 TiB.

Q25 (4 marks). Why do we not just use a single triple indirect block instead of having direct blocks?

Using only a triple indirect block would increase the number of pages used, and increase access time for small files.

Q26 (3 marks). How would you support a maximum file size of at least 100x more than currently supported without changing the size or number of pointers (you may change how the pointers are used)?

You could remove one of the direct blocks, and use a quad indirect block instead.

Virtual Memory (20 marks total)

Assume a virtual memory system, Sv48, with 48-bit virtual addresses, 8-byte page table entries (PTEs), 56-bit physical addresses, and a 4 KiB page size. Answer the following questions:

Q27 (2 marks). How many virtual page numbers (VPNs) are there in this system? Provide your answer as a power of 2.

With 48-bit virtual addresses and a 4 KiB (or 2^{12} bytes) page size, the number of VPNs is 2^{36}.

Q28 (2 marks). Calculate the size of a single-level page table in bytes for this system, assuming it maps the entire virtual address space.

Each page table entry (PTE) is 8 bytes. We need one entry for each VPN, therefore we need $2^{36} \times 2^3$, or 2^{39} bytes (512 GiB).

Q29 (6 marks). Determine the number of levels of page tables used in this system. Explain your reasoning and/or show your calculations.

For our page size of 2^{12}, and 8 byte (2^3) PTEs, we'd need 2^9 entries, or 9 index bits. Since we're using 36 bits for VPNs, we would need 4 levels of page tables ($\lceil \frac{36}{9} \rceil$).

Q30 (4 marks). If this system had one fewer level of page tables, would you expect performance to increase or decrease? Briefly justify your answer.

Removing a level would increase performance due to one less memory access during translation. The higher the miss ratio, the slower it would be comparatively.

Q31 (2 marks). Describe the modifications necessary if the system only supported a 48-bit physical address instead of 56-bit.

With a 48-bit physical address, we only need to store 36 bits for the PPN instead of 44 bits. The PTE size could potentially be reduced, but since it's usually a power of 2 number of bytes, it would still likely be 8 bytes.

Q32 (4 marks). Your friend, who hasn't taken this course, notices a strange performance issue: accessing elements in an array for the first time is slow every 1024th element. Explain why accessing an element in an array might be slow every 1024th element for the first time.

This likely results from a TLB miss from accessing a new page. We would need to perform 5 memory accesses total instead of just one. Likely, given this system, each element was an int (or another type occupying 4 bytes).