2025 Winter Test 2

Course: ECE353: Systems Software
Examiners: Jon Eyolfson
Date: March 17, 2025
Duration: 45 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.

`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.

Virtual Memory (14 marks total)

Consider a system with a page size of 4096 bytes, and a PTE size of 16 bytes. It uses a two-level page table to translate virtual addresses. Some truncated page tables (which fit on a page) are shown below:

Index	PPN	Valid
0	0xA	0
1	0x8	1
2	0xB	0

Page Table at 0x6000

Index	PPN	Valid
0	0x5	1
1	0xD	0
2	0x4	1

Page Table at 0x8000

Index	PPN	Valid
0	0x9	1
1	0x5	0
2	0x4	0

Page Table at 0x7000

Index	PPN	Valid
0	0xC	0
1	0x2	1
2	0x3	1

Page Table at 0x9000

Q1 (2 marks). How many PTEs can you fit into a single page? (Answer can be a power of 2.)

$\frac{2^{12}}{2^4} = 2^8 = 256$.

Q2 (1 marks). What is the maximum size (in bits) of virtual address supported?

28 bits (8 bits for the L1 index, plus 8 bits for the L0 index, plus 12 bits for the offset).

Q3 (1 marks). For the PPN `0xA000`, what range of physical memory addresses does this page correspond to?

`0xA000000` to `0xA000FFF`

Q4 (4 marks). For this system, assume the root page table is at PPN `0x6`, and any entries not shown in the page tables are invalid. Show all the valid VPN to PPN translations.

VPN: `0x0100` to PPN: `0x5`

and

VPN: `0x0102` to PPN: `0x4`

Q5 (2 marks). How many levels of page tables would you need to support a 40 bit virtual address?

We need 28 VPN bits (40 - 12), and each level handles 8 index bits. Therefore, we need $\lceil \frac{28}{8} \rceil$, or 4 levels of page tables.

Q6 (4 marks). If you had to support a 40 bit address, and could re-design the system by changing the PTE size, what's a reasonable number to change it to? Justify your decision with advantages and disadvantages over the current design.

You would want to make the PTE smaller so more entries can fit on a page. There aretwo reasonable answers: 8 bytes or 4 bytes. For 8 bytes, we can fit more PTEs on a page, which may reduce the number of pages we need to use for page tables. However, it would not reduce the number of levels of page tables we need. We would also have less bits available to store the PPN. For 4 bytes, we can fit even more PTEs on a page, to the point of reducing the number of levels we need as well from 4 to 3. However, we would have even fewer bits to store the PPN.

Threads (14 marks total)

Consider the following code, assume that all system calls are always successful, and `printf` is thread-safe.

```c
static int x = 3;

void* r(void*) {
    --x;
    printf("pid: %d, r x: %d\n", getpid(), x);
    return NULL;
}

int main(void) {
    pthread_t t1;
    pthread_t t2;
    pthread_create(&t1, NULL, r, NULL);
    pthread_join(t1, NULL);
    int pid = fork();
    if (pid == 0) {
        --x;
    }
    pthread_create(&t2, NULL, r, NULL);
    printf("pid: %d, main x: %d\n", getpid(), x);
    return 0;
}
```

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

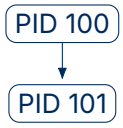
Q7 (1 marks). How many **threads** are created in total **only** from calls to `pthread_create`?

3.

Q8 (1 marks). How many mew **processes** are created in total?

1.

Q9 (1 marks). Draw the process tree of parent/child relationships, showing the likely process IDs.



Q10 (2 marks). Write a **single** possible output from `printf` statements from process ID 100.

```
pid: 100, main x: 2
```

Q11 (5 marks). Write **all other** possible outputs from `printf` statements from process ID 100 (not including the one you already wrote above).

```
pid: 100, main x: 2

pid: 100, main x: 2
pid: 100, r: 1

pid: 100, main x: 1
pid: 100, r: 1

pid: 100, r: 1
pid: 100, main x: 1
```

Q12 (2 marks). What is a "zombie" thread? How would we ensure we never have them?

A zombie thread is a terminated joinable thread that hasn't been joined yet. We could create detached threads to ensure they'll never be "zombies".

Q13 (2 marks). Is it possible to have any "zombie" threads in any processes when they terminate? Explain why or why not.

Yes, it is, because neither process joins `t2`, so it's possible for that thread to terminate before the process terminates.

Synchronization (8 marks total)

Consider the following code:

```
/* Create semaphores here */
```

```
void* run1(void*) {
```

```
    printf("run1: start\n");
```

```
    printf("run1: end\n");
```

```
    return NULL;
}
```

```
void* run2(void*) {
```

```
    printf("run2\n");
```

```
    return NULL;
}
```

```
int main(void) {
    pthread_t t1;
    pthread_t t2;
    /* Initialize semaphores here */
```

```
    pthread_create(&t1, NULL, run1, NULL);
    pthread_create(&t2, NULL, run1, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

Q14 (8 marks). Modify the code such that we always see the output:

```
run1: start
run2
run1: end
```

You may either write your answer on the code directly, or describe your changes here.

Create two semaphores:

```
static sem_t sem1;
static sem_t sem2;
```

Initialize them as follows:

```
sem_init(&sem1, 0, 0);
sem_init(&sem2, 0, 0);
```

Add the following:

Line 13: `sem_post(&sem1);`

Line 23: `sem_wait(&sem1);`

Line 14: `sem_wait(&sem2);`

Line 25: `sem_post(&sem2);`

Short Answer (9 marks total)

Q15 (3 marks). What are the conditions for a data race? How would you prevent one?

You must have two concurrent accesses to the same location (variable) with at least one write. You can prevent one by using a mutex to eliminate concurrent execution of the critical section.

Q16 (3 marks). What's the difference between concurrency and parallelism?

Concurrency is about switch between multiple tasks over time. Parallelism is running two tasks at the same time instant.

Q17 (3 marks). Describe a scenario where 3 threads can all a deadlock

Assume we have 3 mutexes as well: M1, M2, and M3.

T1 wants to lock M1 and then M2.

T2 wants to lock M2 and then M3.

T3 wants to lock M3 and then M1.

What may happen is T1 runs first and locks M1, then T2 runs and locks M2, and finally T3 runs and locks M3. Now all 3 threads are deadlocked.