# CS 162 Operating Systems and System Programming
# Spring 2025

MIDTERM 1

**INSTRUCTIONS**

Please do not open this exam until instructed to do so. Do not discuss exam questions for at least 24 hours after the exam ends, as some students may be taking the exam at a different time.

For questions with **circular bubbles**, you should select exactly *one* choice.

◯ You must choose either this option

◯ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

☐ You could select this choice.

☐ You could select this one too!

**GENERAL INFORMATION**

This is a **closed book** exam. You are allowed 1 page of notes (both sides). You have 80 minutes to complete as much of the exam as possible. Make sure to read all of the questions first, as some of the questions are substantially more time consuming.

Write all of your answers directly on this paper. *Make your answers as concise as possible.* On programming questions, we will be looking for performance as well as correctness, so think through your answers carefully. If there is something about the questions that you believe is open to interpretation, please ask us about it!

| Problem | Possible |
|---|---|
| 1 | 12 |
| 2 | 16 |
| 3 | 18 |
| 4 | 10 |
| 5 | 9 |
| 6 | 18 |
| **Total** | 83 |

**Preliminaries**

This is a proctored, closed-book exam. You are allowed 1 page of notes (both sides). You may not use a calculator. You have 80 minutes to complete as much of the exam as possible. This exam is out of 83 points. Make sure to read all the questions first, as some are substantially more time-consuming.

If there is something about the questions you believe is open to interpretation, please ask us about it.

**There is a reference sheet at the end of the exam that you may find helpful.**

**(a)**

Name

**(b)**

Student ID

**(c)**

Discussion TA's Full Name

**(d)**

Please read the following honor code: "I understand this is a closed-book exam. I promise the answers I give on this exam are my own. I understand that I am allowed to use one 8.5x11, double-sided, handwritten cheat-sheets of my own making, but otherwise promise not to consult other people, physical resources (e.g. textbooks), or the internet in constructing my answers."

**Write your full name below to acknowledge that you've read and agreed to this statement.**

This page is intentionally left blank, with the exception of this sentence, the header, and two jokes.

*A programmer had a problem. They thought to themselves, "I know, I'll solve it with threads!"*
*have Now problems. two they*

**1. (12.0 points) True/False**

Please explain your answer in **TWO SENTENCES OR FEWER**. Answers without an explanation or that just rephrase the question will **not receive credit**.

**(a) (2.0 pt)** The process control block (PCB) is located in the user's virtual address space.

◯ True. ◯ False.

Explain.

**(b) (2.0 pt)** Linux uses the same data structure for threads and processes.

◯ True. ◯ False.

Explain.

**(c) (2.0 pt)** Calling `read` on a file in a process **cannot** affect the writing position of the same file in another process.

◯ True. ◯ False.

Explain.

**(d) (2.0 pt)** In Unix, if all calls to `exec` succeed, the following function:

```
void func(void) {
    exec("./program1");
    exec("./program2");
}
```

will create two new processes.

◯ True. ◯ False.

Explain.

**(e) (2.0 pt)** A user program can create a critical section by making a syscall to disable interrupts and another syscall to re-enable them at the end.

◯ True. ◯ False.

Explain.

**(f) (2.0 pt)** The OS receives interrupts from external IO devices in the form of syscalls.

◯ True. ◯ False.

Explain.

**2. (16.0 points) Multiple Select**

**(a) (2.0 pt)** Select all examples of "privileged" instructions.

- ☐ Atomic instructions
- ☐ Instructions change the address space of the running process
- ☐ `call` in x86
- ☐ `iret` in x86
- ☐ Instructions that directly access IO devices

**(b) (2.0 pt)** Select all true statements about `fork`.

- ☐ The address space is copied, so all pointers are initially identical in value between the processes.
- ☐ The new process does **not** share open file descriptions from the parent process.
- ☐ The child process begins execution from the program's `main` function.
- ☐ The child process can modify the global variables of the parent process.
- ☐ All processes share a common ancestor called `init`.

**(c) (2.0 pt)** Select all true statements about condition variables and monitors.

- ☐ A monitor consists of a lock and zero or more condition variables.
- ☐ Threads waiting on a condition variable sleep while holding the lock.
- ☐ The downside of Mesa scheduling is that threads have to spin in a while-loop, essentially busy-waiting.
- ☐ Hoare monitors are more practical to implement on a real system than Mesa monitors.
- ☐ Mesa monitors immediately transfer the ownership of the lock and schedule the woken up thread upon a call to `cond_signal`.

**(d) (2.0 pt)** Select all ways that processes can communicate with each other.

- ☐ Pipes
- ☐ Sockets
- ☐ Heap memory
- ☐ Shared files
- ☐ Signals

**(e) (2.0 pt)** Select all that is true about threads.

- ☐ `pthread_create` will spawn a new thread that duplicates the register states of the calling thread.
- ☐ Threads are prevented by the OS from modifying the stacks of other threads in the same process.
- ☐ If a thread is blocked waiting to acquire a lock, any thread in the same process will also get blocked.
- ☐ Once `pthread_join` is called, the thread being joined on will start running immediately.
- ☐ After a thread calls `sched_yield`, the OS may immediately reschedule that thread.

**(f)** **(2.0 pt)** Select all true statements about the open file description table.

- ☐ An entry in this table can correspond to two distinct file descriptors within the same process.
- ☐ An entry in this table can correspond to two distinct file descriptors across two different processes.
- ☐ It keeps track of offset and permissions in the file.
- ☐ It resides in a shared region of user memory, allowing all processes to modify it directly.
- ☐ It is global to the entire operating system.

**(g)** **(4.0 pt)** Select possible contents of `low.txt` from the choices below:

```
void* thread_dup(void* arg) {
    int fd = *((int*) arg);
    dup2(fd, STDOUT_FILENO);
    return NULL;
}

int main(void) {
    const char *child_text  = "AAA";
    const char *parent_text = "BBB";

    int fd = open("low.txt", O_WRONLY | O_CREAT); // Creates the file if it doesn't exist.

    pid_t pid = fork();
    pthread_t tid;

    if (pid == 0) {
        pthread_create(&tid, NULL, thread_dup, &fd);
        printf("%s", child_text);
        pthread_join(tid, NULL);
        write(fd, child_text, strlen(child_text));
    } else if (pid > 0) {
        write(fd, parent_text, strlen(parent_text));
        close(fd);
    } else {
        // Assume this case doesn't happen.
    }

    return 0;
}
```

Assume all syscalls succeed. Assume calls to `write` writes the maximum number of bytes possible.

- ☐ AAA
- ☐ AAAAAA
- ☐ AAABBB
- ☐ BBBAAA
- ☐ AAABBBAAA

**3. (18.0 points) Short Answer**

**(a) (4.0 pt)** Fill in the blanks for what precisely happens during a Pintos syscall.

1. The user program places the arguments in/on [ ] . (Be specific!)
2. Then it invokes the instruction: [ ] (Hint: See step 3!)
3. It finds the appropriate handler by indexing into the [ ] with index `0x30`.
4. Pintos kernel jumps to the syscall handler.
5. The syscall handler finds what syscall was invoked by examining the syscall NR/enum.
6. The syscall handler places the return value of the syscall in [ ] for the user program.

**(b) (3.0 pt)** Provide two benefits of using the high-level file API over the low-level file API, specifically for `fread` vs `read`. **Explain and give a proper justification in 2-3 sentences.**

Benefit 1:

Benefit 2:

**(c) (3.0 pt)** How does a modern OS regain control of the processor from a user process stuck in an infinite loop? Define and explain this mechanism in **1-2 sentences**.

**(d) (6.0 pt)** Fill in the blanks with functions from the `pthread` library to ensure the following print output:

MAIN THREAD

THREAD 1

THREAD 2

Do not insert additional semicolons, `printf` statements, loops, or conditions (including ternary operators).

```
void* thread_1(void* arg) {

    ____________________________________; // Fill in the blank
    printf("THREAD 1\n");
    return NULL;
}

void* thread_2(void* arg) {

    ____________________________________; // Fill in the blank
    printf("THREAD 2\n");
    return NULL;
}

int main() {
    pthread_t main_tid = pthread_self(); // Gets the thread ID of the main thread.
    pthread_t temp_tid;

    pthread_create(_____________, NULL, _____________, _____________); // Fill in the blank

    pthread_create(_____________, NULL, _____________, _____________); // Fill in the blank

    printf("MAIN THREAD\n");

    ____________________________________; // Fill in the blank
}
```

**(e) (2.0 pt)** Recall the Readers-Writers problem. What is the key downside of the implementation of the Readers-Writers lock as presented in lecture? Fill in the blank with either "Readers" or "Writers". (Note: There will be no partial credit.)

As long as there is a stream of [ ] that arrive constantly,

[ ] can wait indefinitely to access the lock.

This page is intentionally left blank, with the exception of this sentence, the header, and a joke.

*A programmer had a problem. They thought to themselves, "I know, I'll solve it with semaphores!"*
*Now they can't tell if the problem is waiting on them, or they're waiting on the problem.*

**4. (10.0 points) Episode IV: A New Loop**

(a) **(7.0 pt)** Recall the atomic instruction `compare-and-swap`, which was presented in lecture and discussion. Here is the syntax and pseudocode for its functionality (shortened to `CAS`):

```
bool CAS(int* mem, int old, int new) {
    if (*mem == old) {
        *mem = new;
        return true;
    } else {
        return false;
    }
}
```

We have writers that want to share strings to readers, using a buffer that contains at most one string at a time. You must make sure that every string supplied by `write_string` is printed exactly once by a call to `read_string` in a race-free manner. Fill in the blanks in the code snippet below to complete the implementation.

You may use `CAS`, but you may not insert additional semicolons, other function calls, loops, or conditionals (including ternary operators).

```
#define IDLE  0
#define BUSY  1
#define FULL  2

int state = ______________________;
char data[MAX_STR_LEN + 1];

/* Puts a string in data to be read and printed exactly once in read_string.
   Assume that tx_str is no more than MAX_STR_LEN characters long. */
void write_string(char* tx_str) {

    while (___________________________________) {
        <------ SUBPART (b.i) ------>
    }

    strcpy(data, tx_str); // Copies the string.

    ___________________________________;
}

/* Receives the string from write_string and prints it. */
void read_string(void) {

    while (___________________________________);
    printf("Received: %s\n", data);

    ___________________________________;

    <------ SUBPART (b.ii) ------>
}
```

(b) **(3.0 pt)** With a low frequency of reader arrivals, it's possible that writers spin in the loop for a long time.

Add appropriate calls to `futex` in the space indicated by `SUBPART (b.i)/(b.ii)` to minimize busy-waiting for writers, in the case that no readers are present while many writers are. (Hint: Read the note below this question.)

i. **(1.5 pt)**

ii. **(1.5 pt)**

*Note: If you forgot how* `futex` *works, refer to the note below.*

Here is the function signature for 'futex':

```
int futex(int *uaddr, int futex_op, int val);
```

If `futex_op == FUTEX_WAIT`, the kernel checks atomically as follows:

If `(*uaddr == val)`: The calling thread will be put to sleep.

If `(*uaddr != val)`: The calling thread will keep running (i.e. futex returns immediately)

If `futex_op == FUTEX_WAKE`, this function will wake up to `val` waiting threads associated with the address `uaddr`.

**5. (9.0 points) To read, or not to read**

(a) **(7.0 pt)**

Instead of having a full-blown monitor, here is an alternative solution to the Readers-Writers problem.

We can use a "sequence number" to detect whether there is a concurrent write happening or not. Consider this structure that contains data with a sequence number associated with it.

```
typedef struct seqlock {
    unsigned int sequence; // Assume this is initialized to 0.
    int writer_mutex;      // Assume this is initialized to 0.
    char content[CONTENT_SIZE];
} seqlock;
```

Use the structure above to ensure that both reading and writing to `content` is race-free. Many readers and at most one writer can access `content` concurrently, but a call to `read_synchronized` must **not** return after reading an intermediate result of `write_synchronized`. Fill in the blanks in the code snippet below to complete the implementation.

Assume all members of `seqlock` are initialized to 0. You may **not** insert additional semicolons, conditionals, or loops (including ternary operators).

```
/* Copies the array new_content to lock->content in a race-free manner. */
void write_synchronized(seqlock* lock, char new_content[CONTENT_SIZE]) {
    while (TSET(&lock->writer_mutex));

    ______________________________________;

    memcpy(lock->content, new_content, CONTENT_SIZE); // Write new contents.

    ______________________________________;

    ______________________________________;
}

/*  Reads from lock->content to buf in a race-free manner.
    The data read upon returning from the function must not be
    an intermediate result of write_synchronized. */
void read_synchronized(seqlock* lock, char buf[CONTENT_SIZE]) {
    int temp;

    do {

        ______________________________________;

        memcpy(buf, lock->content, CONTENT_SIZE);

    } while (_____________________________________);
}
```

(b) **(2.0 pt)** What kind of access pattern does this lock optimize for? (Select one.)

☐ Infrequent reads and frequent writes

☐ Frequent reads and infrequent writes

☐ Highly contended data access (frequent reads and writes)

**6. (18.0 points) The Requiem for the OH Scheme**

Annie realizes that she barely gets help in OH and thinks that the OH queue is broken. She wants to make a program that matches students on the OH queue with TAs **in PintOS** without any busy waiting or deadlock.

```
typedef struct ticket {
   bool served;  // indicates if a student has finished asking the question for their ticket.
   int student_num; // uniquely identifies the student who created the ticket.
   int staff_num; // uniquely identifies the staff member who is taking the ticket.
   struct list_elem elem;
} ticket_t;

/* OH PintOS List Implementation. */
struct list queue;
ticket_t* pop(struct list* queue) { return list_entry(list_pop_front(queue), ticket_t, elem); }
void push(struct list* queue, ticket_t* ticket) { list_push_back(queue, ticket->elem); }

void ask_question(int staff_num);  // staff_num identifies the staff member being asked a question.
void take_ticket(int student_num); // student_num identifies the student who made the ticket.
void resolve_ticket(ticket_t* ticket); // resolves and frees the ticket.

/* NOTE: Use the below condition variables IN ORDER FROM LEFT TO RIGHT. */
struct condition ticket_created, ticket_taken, ticket_served;
struct lock oh_lock;
```

(a) **(12.0 pt) Write your answers directly in the boxes below.** Assume many threads in the same process all either run `staff_func` or `student_func`. Each student and staff member is assigned a **unique positive** integer identifier (i.e. `staff_num` or `student_num`). **Assume `queue` and all synchronization variables are properly initialized.** Use **Mesa logic** for monitors. Remember to use **PintOS** functions.

i. Student should wait for staff to **take their ticket** before **asking their question**.

ii. Staff should wait for students to **ask their question** before **resolving their ticket**.

```
/* STAFF IMPLEMENTATION */
void staff_func(int staff_num) {

    ____________________________;

    ____________________________

        ____________________________;

    ticket_t* ticket = pop(&queue);
    ticket->staff_num = staff_num;
    take_ticket(ticket->student_num);

    ____________________________;

    ____________________________

        ____________________________;

    resolve_ticket(ticket);

    ____________________________;
}
```

```
/* STUDENT IMPLEMENTATION */
void student_func(int student_num) {

    ____________________________;

    ticket_t* ticket = calloc(1, sizeof(ticket_t));
    ticket->student_num = student_num;
    push(&queue, ticket);

    ____________________________;

    while (!ticket->staff_num)

        ____________________________;

    ask_question(ticket->staff_num);
    ticket->served = true;

    ____________________________;

    ____________________________;
}
```

(b) **(2.0 pt)** Suppose there is a large number of threads running the functions `student_func` and `staff_func`. Why is using a **global** condition variable for `ticket_taken` and one for `ticket_served` very inefficient in causing excessive scheduling overhead? Explain in **1-3 sentences.**

.

(c) **(2.0 pt)** Detail a solution to the issue you discussed in **6(b)** that replaces the global condition variables for `ticket_served` and `ticket_taken`. Be specific. Explain in **1-3 sentences**.

.

(d) **(2.0 pt)** Luca argues that we could just replace each condition variable with a semaphore, and completely remove the `oh_lock` from **6(a)** without any race conditions. Is Luca right? Why or why not?

.

**Wait, don't miss the extra credit question on the last page!**

**7. (Extra credit: each 0.5 pt)** What are the **full names** of the two class professors?

(Note: You must spell them correctly for full credit!)

**Reference Sheet**

```
/* Processes */
pid_t fork(void);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int execv(const char *path, char *const argv[]);
/* pthreads */
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
      void* (*start_routine (void *)), void *arg);
int pthread_join(pthread_t thread, void **retval);
void pthread_exit(void *retval);

/* pthread Semaphore interface */
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem); /* The p() or down() operation */
int sem_post(sem_t *sem); /* The v() or up() operation */

/* pthread Lock/mutex operations */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

/* pthread Condition Variable */
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

/* PintOS locks */
void lock_init(struct lock *lock);
void lock_acquire(struct lock *lock);
void lock_release(struct lock *lock);

/* PintOS semaphore interface */
void sema_init(struct semaphore *sema, unsigned value);
void sema_down(struct semaphore *sema);
void sema_up(struct semaphore *sema);

/* PintOS condition variables */
void cond_init(struct condition *cond);
void cond_wait(struct condition *cond, struct lock *lock);
void cond_signal(struct condition *cond, struct lock *lock);
void cond_broadcast(struct condition *cond, struct lock *lock);

/* PintOS Readers/Writers Locks */
void rw_lock_init(struct rw_lock*);
void rw_lock_acquire(struct rw_lock*, bool reader);
void rw_lock_release(struct rw_lock*, bool reader);

/* PintOS List */
void list_init(struct list *list);
```

```
struct list_elem *list_head(struct list *list);
struct list_elem *list_tail(struct list *list);
struct list_elem *list_begin(struct list *list);
struct list_elem *list_next(struct list_elem *elem);
struct list_elem *list_end(struct list *list);
struct list_elem *list_remove(struct list_elem *elem);
bool list_empty(struct list *list);
#define list_entry(LIST_ELEM, STRUCT, MEMBER) ...
void list_insert(struct list_elem *before, struct list_elem *elem);
void list_insert_ordered(struct list* list, struct list_elem* elem, list_less_func* less, void* aux);
void list_push_front(struct list *list, struct list_elem *elem);
void list_push_back(struct list *list, struct list_elem *elem);
struct list_elem* list_pop_front(struct list* list);
struct list_elem* list_pop_back(struct list* list);

/* Strings */
char *strcpy(char *dest, char *src);
char *strdup(char *src);

/* Interrupt enable/disable */
enum intr_level {};
enum intr_level intr_get_level(void)
enum intr_level intr_set_level(enum intr_level level)
enum intr_level intr_enable(void)
enum intr_level intr_disable(void)

/* High-Level IO */
FILE *fopen(const char *pathname, const char *mode);
int fclose(FILE *stream);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
fprintf(FILE * restrict stream, const char * restrict format, ...);

/* Low-Level IO */
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int pipe(int pipefd[2]);
int close(int fd);

/* Socket */
int socket(int domain, int type, int protocol);
int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, structure sockaddr *addr, socklen_t *addrlen);
int connect(int sockfd, struct sockaddr *addr, socklen_t addrlen);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);

/* Memory */
void *memcpy(void *dest, const void * src, size_t n)
```