CS 162 Fall 2024 — Operating Systems and System Programming

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 110 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	16
2	18
3	18
4	18
5	9
6	21
Total	100

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 110 minutes to complete as much of the exam as possible. This exam is out of 100 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.

We will overlook minor syntax errors when grading coding questions. **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.

1. **(16.0 points) True/False**

Please explain your answer in **TWO SENTENCES OR LESS**. Longer explanations will **GET NO CREDIT**. Answers without an explanation or that just rephrase the question will **GET NO CREDIT**.

(a) **(2.0 pt)** The interrupt vector table can be safely stored in user memory because user-level applications do not modify these handlers directly.

◯ True. ◯ False.

Explain.

(b) **(2.0 pt)** A thread cannot be preempted during a critical section (i.e. after successfully acquiring a lock and before releasing it).

◯ True. ◯ False.

Explain.

(c) **(2.0 pt)** Upon a mode transfer from user to kernel, user registers are saved in kernel memory.

◯ True. ◯ False.

Explain.

(d) **(2.0 pt)** In a base&bound system, only the kernel would have access to special instructions to change the base&bound registers.

◯ True. ◯ False.

Explain.

(e) **(2.0 pt)** In dual mode operation, a user thread and its associated kernel thread can execute in parallel.

◯ True. ◯ False.

Explain.

(f) **(2.0 pt)** You can choose to overwrite the file descriptors for STDIN, STDOUT, and STDERR in Linux.

◯ True. ◯ False.

Explain.

(g) **(2.0 pt)** A thread can modify the stack variables of another thread in the same process.

◯ True. ◯ False.

Explain.

(h) **(2.0 pt)** Creating a pipe or socket in Unix will allocate a file on disk for two processes to communicate.

◯ True. ◯ False.

Explain.

2. (18.0 points) Multiple Select

(a) (2.5 pt) Select all true statements.

☐ Atomic operations cannot be performed in user mode for security reasons.

☐ The `wait` operation in a condition variable requires an implementation that atomically drops the lock and puts the calling thread on the corresponding wait queue.

☐ You can check the value of a semaphore and decide whether you want to wait or proceed.

☐ Calling pthread_mutex_lock twice in the same thread can lead to deadlock.

☐ Calling sem_wait twice in the same thread results in undefined behavior.

(b) (2.5 pt) Select all true statements about POSIX threads (pthreads).

☐ When the last thread in a process exits, the process will terminate.

☐ Once pthread_create is called, the new thread will start running immediately.

☐ Once pthread_join is called on a sleeping thread, the calling thread will be blocked immediately.

☐ 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 the same thread to run.

(c) (2.5 pt) Select all true statements regarding Pintos.

☐ Virtual address 0xc0004567 always maps to the same physical address in Pintos.

☐ Even in user mode, kernel memory is always mapped.

☐ Virtual address 0x08045671 may refer to multiple locations in physical memory, depending on the currently running user process.

☐ The kernel can read from any user virtual address in Pintos, including unmapped user addresses.

☐ The PCB of a process is stored on the stack of its main thread.

(d) (2.0 pt) Which of the following are guaranteed to terminate the running user process?

☐ Receiving the SIGKILL signal

☐ Segmentation fault (i.e. receiving SIGSEGV)

☐ Interrupts

☐ Executing a trap instruction

☐ None of the above.

(e) (2.5 pt) Select all that is true about sockets and IPC.

☐ `pipe` allocates two new file descriptors, one for reading and another for writing.

☐ A socket is a file descriptor which you can simply `read` or `write` to.

☐ Processes on the same machine cannot communicate with each other through a socket.

☐ Threads in the same process can communicate with each other through a pipe.

☐ Typing CTRL + C in a UNIX shell is a form of interprocess communication.

(f) (2.0 pt) Select all true statements. Assume all syscalls mentioned are successful.

☐ Calling open on the same file twice within the same process creates two distinct file descriptors corresponding to the same file description.

☐ Immediately after `fork`, parent processes and child processes own identical sets of file descriptors, which refer to the same file descriptions.

☐ Suppose a process with some valid file descriptor 3 forks a new process. If the child process calls `close(3)`, the file will also be closed for the parent.

☐ Every call to `fread` makes a call to `read`.

☐ None of the above.

(g) (2.0 pt) Select all true statements.

☐ A modern OS regains control of a CPU from a user program stuck in an infinite loop through a trap/exception.

☐ The user-level code implementation for a lock can disable and reenable interrupts without trapping into the kernel.

☐ Atomic instructions provide a better alternative for implementing locks on multiprocessor systems when compared to disabling and reenabling interrupts.

☐ Semaphores can be used to enforce mutual exclusion.

☐ None of the above.

(h) (2.0 pt) Select all true statements.

☐ You can look at the source code of an executable with `objdump`.

☐ You can set a breakpoint in GDB with `break`.

☐ In `gcc`, you can define a macro with the "-D" flag.

☐ An object file's symbol table contains the names, scope, and memory addresses of both stack and global variables.

☐ None of the above.

3. (18.0 points) Short Answer

(a) **(2.0 pt)** Provide 3 scenarios where a thread can go from a RUNNING state to a BLOCKED state. When mentioning specific functions that may be called, clarify the constraints under which the thread becomes BLOCKED. (3 sentences max)

(b) **(2.0 pt)** What needs to be saved and restored on a context switch between two threads in the same process? What if the two threads are in different processes? (2 sentences max)

(c) **(3.0 pt)** Name and define 3 types of user to kernel mode transfer mechanisms and give an example of each. (3 sentences max)

(d) **(3.0 pt)** Explain the key difference between Mesa and Hoare semantics. Provide one advantage of each kind of semantic.

i. **(1.5 pt)** Hoare Semantics (2 sentences max):

ii. **(1.5 pt)** Mesa Semantics (2 sentences max):

(e) **(2.0 pt)**

Provide two specific reasons why concurrency even without parallelism may be useful in an operating system, even when it incurs additional context switching overhead. (3 sentences max)

(f) **(2.0 pt)** Suppose we want to implement a program that echoes its input from STDIN to STDOUT exactly twice. For simplicity of this question and the next, assume that all calls to read/write will succeed and read the maximum number of bytes possible. The length of the input is fixed and defined as a macro. We have an implementation of echo-twice here:

```
#define INPUT_LEN 16
int main(void) {
    pid_t pid = fork();
    char buffer[INPUT_LEN + 1];
    buffer[INPUT_LEN] = '\0';
    read(STDIN_FILENO, buffer, INPUT_LEN);
    printf("%s\n", buffer);
}
```

This will always fail achieve the desired effect. Explain why in 2 sentences or fewer.

(g) **(4.0 pt)** Given the following code, how many threads will be created and what will be printed to the console? Assume that all calls to `pthread_create` succeed. Do not count the main thread.

```
typedef struct share {
  int spawn;
  int target;
} shared_t;

int helper(void *arg) {
  shared_t *shared = (shared_t *)arg;
  shared->target++;
  shared->spawn--;
  if (shared->spawn >= 0) {
    pthread_t new_thread;
    pthread_create(&new_thread, NULL, (void *)helper, (void *)shared);
    pthread_join(new_thread, NULL);
  }
}

int main(void) {
  shared_t shared = {.spawn = 5, .target = 0};
  pthread_t new_thread;
  pthread_create(&new_thread, NULL, (void *)helper, (void *)&shared);
  pthread_join(new_thread, NULL);
  printf("%d", shared.target);
  return 0;
}
```

i. **(2.0 pt)**

How many threads will be created and what will be printed to the console?

ii. **(2.0 pt)**

If the pthread_join method in `helper` was removed, how many threads would be guaranteed to execute and what would be the output? Again, do not count the main thread.

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

4. (18.0 points) Jacob's Atomics

(a) **(3.0 pt)** Assume x = 0 and y = 0 initially.

```
Thread A:                                   Thread B:
test-and-set x, reg1  // reg1=Mem[x], Mem[x]=1    load y, reg2   // reg2 = Mem[y]
add 1, reg1           // reg1 = reg1 + 1          add 1, reg2    // reg2 = reg2 + 1
store reg1, y         // Mem[y] = reg1            store reg2, x  // Mem[x] = reg2
```

After both threads finish executing, what are all possible values of (x, y)?
Consider all interleavings of Thread A and Thread B.

(b) **(4.0 pt)** Here is the syntax and functionality of compare-and-swap:

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

In the following question, we may have many concurrent threads that are accessing and modifying the value at `int* sum`. Implement a thread-safe function that adds an `int val` to the integer stored at `int* sum`, using the atomic instruction compare-and-swap.

```
void adder(int* sum, int val) {
    int old;
    int tmp;
    do {
        old = ___________[A]__________;
        tmp = ___________[B]__________;
    } while (___________[C]___________);
}
```

i. **(1.0 pt)** A (max 1 line)

ii. **(1.0 pt)** B (max 1 line)

iii. **(2.0 pt)** C (max 1 line)

(c) **(11.0 pt)** Now forget about compare-and-swap. Suppose the only atomic operation you had was fetch-and-add whose functionality can be described in the following pseudocode:

```
int FAA(int* swp, int t) { // t can be negative
 int old = *swp;
 *swp += t;
 return old;
}
```

Recall `futex` is a syscall that allows a thread to put itself to sleep on a queue associated with a user-level address. It also allows a thread to ask the kernel to wake up threads that might be sleeping on the same queue. Here is also 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.

We want to implement a user-level lock that allows threads to acquire the lock in a FIFO order that they arrive in. This can be done through a "ticketing" system, where threads acquire a number with fetch-and-add. We also want to optimize for minimal busy-waiting with futex.

```
typedef struct lock {
    int ticket;
    int serving;
    int num_waiting;
} lock_t;

void lock_init(lock_t* lock) {
    lock->ticket = 0;
    lock->serving = 0;
    lock->num_waiting = 0;
}

void lock_acquire(lock_t* lock) {
    int my_ticket;
    int tmp;
    my_ticket = ________[A]________;
    while (1) {
        tmp = ________[B]________;
        if (________[C]________) {
            ________[D]________;
            futex(__________[E]__________);
            ________[F]________;
        } else {
            break;
        }
    }
}

void lock_release(lock_t* lock) {
    ________[G]________;
    futex(__________[H]__________);
}
```

i. **(2.0 pt)** A (max 1 line)

ii. **(1.0 pt)** B (max 1 line)

iii. **(1.0 pt)** C (max 1 line)

iv. **(1.0 pt)** D (max 1 line)

v. **(2.0 pt)** E (max 1 line. Separate arguments with commas.)

vi. **(1.0 pt)** F (max 1 line)

vii. **(1.0 pt)** G (max 1 line)

viii. **(2.0 pt)** H (max 1 line. Separate arguments with commas.)

5. (9.0 points) Aditya's World

Aditya is building a video game and wants your help! His game is cooperative, where n players must work together to press a button that is on the other side of a bridge. Multiple players **must** be allowed to cross the bridge at the same time. The #1 rule of the game is that no player can press the button until **all** players have crossed the bridge. Eventually, all players must press the button. Help Aditya implement the following program to build his game. You may not need to use all lines provided. You may **not** use semicolons.

```
You have access to the following global variables:
int n;
Lock lock;
Semaphore sem;

And the following synchronization interfaces:
sem.wait()
sem.post()
lock.acquire()
lock.release()

And finally, this is what each player should call to cross the bridge:
cross_bridge()

initialize_game() {
    n = NUM_PLAYERS;
    lock.init();
    sem.init(_______[A]_______); // Initialize semaphore to value [A]
}

play_game() {                // Each player enters the game with `play_game`.
    int temp;                // You may use this local variable if necessary.
    _______[B]x5_______;
    if (_______[C]_______)
        _______[D]_______;
    _______[E]x2_______;
    press_button();
}
```

The notation `_____[Y]xN______` indicates a response that can be at most `N` lines long and should be written in the answer box for Part Y.

(a) (1.0 pt) [A] (max 1 line)

(b) (3.5 pt) [B] (max 5 lines)

(c) **(1.0 pt)** [C] (max 1 line)

(d) **(1.0 pt)** [D] (max 1 line)

(e) **(2.5 pt)** [E] (max 2 lines)

6. (21.0 points) Diana's Pintoast

Diana is the owner of Pintoast Bakery. There are multiple chefs that bake customers' orders one at a time. A customer will first enter the bakery and place their order. After a chef bakes their order, the customer will pay and leave the bakery, allowing another customer to come in.

(a) Chefs should bake orders in a first-come first-serve manner.

(b) Chefs should not block other cooks or customers while baking an order.

(c) A bakery can only hold up to 100 customers at any given time.

```
typedef struct bakery {
    int capacity;
    struct list orders;
    struct condition customerWait;
    struct condition customerDone;
    struct condition chefDone;
    struct lock orders_lock;
} bakery_t;

typedef struct order {
   bool cooked;
   struct list_elem elem;
   /* Other fields hidden */
} order_t;

/* Assume bake, enter, and leave always execute successfully. */

void bake(order_t* order) { /* Implementation details hidden */ };
void enter(bakery_t* bakery) { /* Implementation details hidden */ };
void leave(bakery_t* bakery) { /* Implementation details hidden */ };
```

Implement the following program to help Diana set up her Pintoast bakery in Pintos.

```
/* Assume that all members of bakery are properly initialized. Assume that a chef will
continuously call this function while the bakery is open. */

void chef(bakery_t* bakery) {
    _______[A]_______;

    while (_______[B]_______)
        cond_wait(&bakery->customerWait, &bakery->orders_lock);

    struct list_elem* e = _______[C]_______;
    order_t* order = list_entry(e, order_t, elem);
    _______[D]________;

    bake(order);

    _______[E]x4________;
}
```

The notation _____[Y]xN______ indicates a response that can be at most N lines long and should be written in the answer box for Part Y.

(a) (9.0 pt) Chef Implementation

i. (1.0 pt) [A] (max 1 line)

ii. (1.0 pt) [B] (max 1 line)

iii. (1.0 pt) [C] (max 1 line)

iv. (1.0 pt) [D] (max 1 line)

v. (5.0 pt) [E] (max 4 lines)

(b) **(12.0 pt)** Customer Implementation

```
/* Assume that bakery->capacity = 100 before any customer enters the bakery.
If there are already 100 customers in the bakery, a new customer should wait
for someone to leave before entering the bakery. */

void customer(bakery_t* bakery, order_t* order) {
    _______[A]_______
    while (_______[B]_______)
        _______[C]_______

    _______[D]_______
    enter(bakery);

    list_push_back(&bakery->orders, &order->elem);
    _______[E]_______

    while (_______[F]_______)
         _______[G]_______

    leave(bakery);
    _______[H]x3_______
}
```

i. **(1.0 pt)** [A] (max 1 line)

ii. **(1.0 pt)** [B] (max 1 line)

iii. **(1.5 pt)** [C] (max 1 line)

iv. **(1.0 pt)** [D] (max 1 line)

v. **(1.0 pt)** [E] (max 1 line)

vi. (1.0 pt) [F] (max 1 line)

vii. (1.5 pt) [G] (max 1 line)

viii. (4.0 pt) [H] (max 3 lines)

This page is intentionally left blank, with the exception of this sentence, the header, and pi.
3.14159265358979323846264338327950288419716939937510582

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)
```