CS 162 Fall 2023

Operating Systems and Systems 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!

Preliminaries

This is a proctored, closed-book exam. You are allowed 1 page of notes (both sides). You may not use a calculator. There are a total of 75 points in this exam. You have 75 minutes to complete as much of the exam as possible. 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. (14 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 points)

i. If you call a function `f(a, b, c)` using the x86 calling convention, the resulting stack would look like the following:

Stack Diagram
padding
a
b
c
return address

Note: In the diagram above, addresses increase upward.

○ True.

● False.

ii. Explain.

Arguments are pushed onto the stack in reverse order.

(b) (2 points)

i. The `pipe()` system call creates a file on disk and returns file descriptors to read and write to it.

○ True.

● False.

ii. Explain.

Pipes are implemented as a kernel buffer in memory, not as files.

(c) (2 points)

i. In PintOS, we validate syscall arguments in user memory.

○ True.

● False.

ii. Explain.

We copy the arguments into kernel memory before we validate them, for security reasons. If it was validated in user memory it is susceptible to malicious changes.

(d) (2 points)

i. In Unix, while `fork()` creates a new process and preserves the parent process, `exec()` creates a new process and kills the parent process.

○ True.

● False.

ii. Explain.

`exec` does not create a new process. Instead, it replaces the originally running parent program with the new exec program.

(e) (2 points)

i. Two processes can agree to share memory by both reading and writing to the virtual address `0xFFFF0000`.

◯ True.

● False.

ii. Explain.

Virtual addresses are per-process.

(f) (2 points)

i. If a single process opens a file twice, a new file descriptor is created for each `open` call and the two file descriptors both point to the different file descriptions.

● True.

◯ False.

ii. Explain.

`open` **creates two different entries in the file descriptor table. Each new file descriptor will also be pointing to a different file description in the global file table.**

(g) (2 points)

i. If you fork a process, changes made to the forked file descriptions are reflected in both the parent and child processes.

- (●) True.
- ( ) False.

ii. Explain.

Forking duplicates descriptors which point to the same file description entry in the global file description table.

2. (14 points) Multiple Select

(a) (2 pt) Select the following statements about sockets that are true.

- [x] Sockets provide bidirectional communication between processes.
- [ ] Open socket file descriptors can be modified using the `lseek()` syscall.
- [ ] A single connection socket can be used to talk to multiple (host, port) endpoints.
- [x] TCP sockets guarantee reliable, in-order communication.
- [ ] None of the above.

A: This is the definition of a socket.
B: A socket file descriptor is used to abstract a network connection and is not a regular file. lseek() is not supported.
C: A connection is represented as a 5-tuple (source IP address, source port, destination IP address, destination port, protocol).
D: The TCP protocol is used to provide guaranteed, in-order delivery of data, with the tradeoff being increased overhead and latency.

(b) (2 pt) Select the following statements about the roles of the operating system that are true.

- [x] The operating system provides common services such as file systems.
- [x] CPU time is an example of a shared resource the operating system manages.
- [x] A process is not allowed to modify the address space of another process.
- [ ] A user program can directly access a file without making a syscall.
- [ ] None of the above.

A: See lecture 1.
B: CPU time is split between processes and the operating system manages this through scheduling.
C: The operating system maintains isolation between processes by giving them separate address spaces.
D: The kernel manages access to the file system and disk.

(c) (2 pt) Select the following statements about x86 calling convention that are true.

- [ ] The caller must always save caller-saved general purpose registers.
- [ ] In the callee epilogue, the callee can deallocate local variables by setting the `ebp` to the `esp`.
- [x] After the instruction `pushl %ebx`, the `esp` will be decremented by 4 bytes.
- [x] After pushing the parameters onto the stack, the stack is 16-byte aligned.
- [ ] None of the above.

A: Caller-saved GPRs only need to be saved if they will be used afterwards.
B: The callee would set the esp to the ebp to deallocate local variables.
C: Pushing something of 4 bytes onto the stack decrements it by 4, since the stack grows downward.
D: This is how x86 stack alignment is defined.

(d) **(2 pt)** Select the following statements about syscalls in PintOS that are true.

- ☐ Syscalls by user programs go directly to the corresponding function implementation.
- ■ The `exit` syscall is eventually called at the end of a user program.
- ■ Syscalls are handled in kernel mode.
- ☐ Syscalls are handled asynchronously.
- ☐ None of the above.

A: They go to a syscall handler first that routes it.
B: The `exit` syscall will not be called in the event of a sudden program exception.
C: By definition syscalls go to kernel mode because it needs kernel-level permissions.
D: They are handled synchronously - in kernel mode and then returned when done.

(e) **(2 pt)** Select the following statements about hardware support for locks that are true.

- ☐ Atomic instructions beyond load and store are required to make a correctly functioning method of mutual exclusion for a uniprocessor.
- ☐ Busy-waiting on a uniprocessor is generally as efficient as implementing locks by disabling and enabling interrupts.
- ☐ A lock implemented using `test_and_set(&lock_val)` as described in lecture can have `lock_val` initialized to any value.
- ☐ All correct lock implementations for uniprocessors function correctly for multiprocessors.
- ■ None of the above.

A: We have seen the solution to the "Too Much Milk" problem in lecture and the textbook which is a safe and live method of mutual exclusion using solely load and store atomic operations.
B: Spin-locks for uniprocessors generally do not make sense as a thread waiting on a lock wastes processor cycles in which the lock cannot be resolved.
C: If lock_val is initialized to 1, then it would not be able to be acquired since any execution of `test_and_set(&lock_val)` will return 1.
D: A basic lock based on solely enabling and disabling interrupts will not function properly on a multiprocessor as enabling and disabling interrupts is not sufficient to prevent other processors from modifying data.

(f) **(2 pt)** In class, we discussed how the `fork()` creates a copy of the old process. What values/structures would `fork` copy into the new process? (Note that in this question, you can assume that we are not using the copy-on-write optimization.)

- ■ Code Segment
- ☐ PID
- ■ ESP
- ■ EIP
- ☐ None of the above.

A naive fork implementation creates a copy of all the address space state from the parent process, including registers, code/data segments, stack, heap, and open file descriptors. The registers copied include the instruction pointer (EIP), stack pointer, and program counter. The process control block of the parent is not copied into the new process, as the child process instead creates its own.

(g) (2 pt) Select the following statements about signals that are true.

- ☐ The `kill` syscall can only be used to send the `SIGKILL` signal.
- ☐ Programs can override the handlers for any signal using the `sigaction` function.
- ☐ The operating system needs to transition from user mode into kernel mode in order to allow the current process to handle a signal.
- ■ The kernel indicates to a process that a segmentation fault has occurred by sending a `SIGSEGV` signal.
- ☐ None of the above.

A: The `kill` syscall can be used to send any signal to the specified process.
B: `SIGKILL`, for example, is not overridable.
C: The operating system transitions from kernel to user mode to enter the signal handler.
D: You can see this for yourself by triggering a segfault and reading the error message.

3. (8 points) Short Answer

(a) (2 pt) Imagine we are trying to write out to a file from a buffer. Name 2 differences that we need to make in our implementation if we use the low-level `write()` compared to if we use the high-level `fwrite()`, and why you need to make them.

- **We need to call write in a loop, as the low-level write does not guarantee that it writes all the bytes out.**
- **Use the low level API instead of high level**
- **FILE* vs. file descriptors**

(b) (2 points)

Recall the different mechanisms for mode transfer that we learned about in class:

- Syscalls
- Interrupts
- Exceptions

For each of the following, list which of the above 3 mode transfer mechanisms most closely matches the described situation. You may use an answer more than once.

i. (0.5 pt) A timer fires, which indicates that a different process will now be scheduled.

Interrupt. An (hardware) interrupt is an external signal to the processor that indicates an event has occurred. The processor enters the interrupt handler.

ii. (0.5 pt) The user programs attempts to access illegal memory and a segfault occurs.

Exception. An exception (software interrupt) is an unexpected condition caused by the user program. This makes the process stop and enter the kernel in the exception handler.

iii. (0.5 pt) An I/O operation which writes to disk while other work is done on the CPU.

Interrupt. Note that the reason why the "other work" portion of the prompt was included was to highlight that this operation is being done asynchronously, which is a marked difference between interrupts and syscalls/exceptions.

iv. (0.5 pt) User program invokes a function to write to a file.

Syscall. A user program is requesting a service from the OS and will invoke a function (e.g. `write` or `fwrite`) to tell the OS what privileged service it wishes to access.

(c) (2 points)

Indicate the output produced by running the following code. Please separate the output in terms of what would be seen in the terminal and what would be seen in the file. *Assume that the printf statements are atomic.*

```
void* helper(void* args) {
    int *fd = (int*) args;
    printf("Printing from thread helper\n");
    dup2(*fd, STDOUT_FILENO);
    return NULL;
}

int main() {
    printf("Starting main\n");
    int file_fd = open("test.txt", O_WRONLY | O_TRUNC | O_CREAT, 0666);
    int* temp_fd = (int*) malloc(sizeof(int));
    *temp_fd = dup(STDOUT_FILENO);
    pthread_t thread;
    dup2(file_fd, STDOUT_FILENO);
    pthread_create(&thread, NULL, &helper, temp_fd);
    pthread_join(thread, NULL);
    printf("Ending main\n");
    return 0;
}
```

i. In terminal:

Starting main
Ending main

ii. In `file.txt`:

Printing from thread helper

(d) (2 points)

Suppose you were given the following implementation of a producer-consumer queue, based on the code provided in the lecture slides. Note that `dequeue` will segfault if the `queue` that is passed in is empty.

In `queue.h`:

```
typedef struct queue {
    // ... omitted ...
} queue_t;

void enqueue(queue_t* queue, int val);
// dequeue segfaults if the queue is empty
int dequeue(queue_t* queue);
bool is_empty(queue_t* queue);
```

In `pc_queue.c`:

```
#include <pthread.h>
#include "queue.h"

typedef struct pc_queue {
    pthread_mutex_t lock;
    pthread_cond_t cv;
    queue_t queue;
} pc_queue_t;

void produce(pc_queue_t* pcq, int item) {
    pthread_mutex_lock(&pcq->lock);
    enqueue(&pcq->queue, item);
    pthread_cond_signal(&pcq->cv);
    pthread_mutex_unlock(&pcq->lock);
}

int consume(pc_queue_t* pcq) {
    pthread_mutex_lock(&pcq->lock);
    ___________________________ {
        pthread_cond_wait(&pcq->cv, &pcq->lock);
    }
    int item = dequeue(&pcq->queue);
    pthread_mutex_unlock(&pcq->lock);
    return item;
}
```

i. **(0.5 pt)** Given that the `pthread` library uses Mesa semantics, what is the best line to fill in the blank above?

```
while (is_empty(&pcq->queue))
```

ii. **(0.5 pt)** Suppose instead that the `pthread` library used Hoare semantics. How could you simplify your answer to the previous question?

Can change the `while` to just an `if`.

iii. **(1 pt)** Suppose that the programmer tried to fill in the blank line assuming that `pthread` used Hoare semantics, while the actual library uses Mesa semantics. Would any bugs result? If so, what might happen? If not, explain why.

A thread running `consume` might see the queue is empty, wait, but open waking find that the queue is still empty. They will then try to call `dequeue` on an empty queue and cause a segfault.

4. (10 points) My Very Own fread!

Provide an implementation of `my_fread` using the low-level file I/O API. `my_fread` should behave similarly to `fread`, where we read data using the low-level file API and buffer the results to speed up future calls.

You may not use `fread` or any of the other functions defined in `stdio.h`. You may assume that the number of bytes requested by `my_fread` will be at most the number of bytes remaining in the open file.

```
#define BUF_SIZE 1024
#define MIN(X, Y) (((X) < (Y)) ? (X) : (Y))

typedef struct file {
    int fd;
    char buffer[BUF_SIZE];
    int buf_offset;          // initialized to 0
    bool has_data;           // initialized to false
} File;

int refill_buffer(File* file) {
    ssize_t bytes_read;
    int read_so_far = 0;
    while ((bytes_read = ______[A]______) > 0) {
        read_so_far += ______[B]______;
    }
    file->buf_offset = ______[C]______;
    file->has_data = true;
    return read_so_far;
}

/* You may assume that `count` is at most the remaining
 * number of bytes in the open file `file->fd`. */
void my_fread(char* dest, size_t count, File* file) {
    if (!file->has_data) {
        ______[D]_______;
    }
    int written_so_far = 0;
    while (written_so_far < count) {
        int bytes_to_copy = MIN(______[E]______);
        memcpy(______[F]______);
        file->buf_offset += _______[G]______;
        if (file->buf_offset == BUF_SIZE) {
            ______[H]______;
        }
        written_so_far += ______[I]______;
    }
}
```

(a) (2 pt) [A] (max 1 expression)

```
read(file->fd, &(file->buffer[read_so_far], BUF_SIZE - read_so_far))
```

(b) (1 pt) [B] (max 1 expression)

```
bytes_read
```

(c) (1 pt) [C] (max 1 expression)

```
0
```

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

```
refill_buffer(file)
```

(e) (1 pt) [E] (parameters)

```
count - written_so_far, BUF_SIZE - file->buf_offset
```

(f) (1 pt) [F] (parameters)

```
dest + written_so_far, &(file->buffer[file->buf_offset]), bytes_to_copy
```

(g) (1 pt) [G] (max 1 expression)

```
bytes_to_copy
```

(h) (1 pt) [H] (max 1 line)

```
refill_buffer(file)
```

(i) (1 pt) [I] (max 1 expression)

```
bytes_to_copy
```

5. (15 points) Reminders

Saahya is creating an interactive productivity tool in PintOS to help her team track tasks in their CS162 project. One feature of this tool is that users can schedule reminder messages to be printed to the screen at regular intervals. All times are specified in seconds.

For example: At `time=100`, a user might schedule a "help!" reminder with interval 10. Then, at `time=110, 120, 130, 140, ...`, "help!" will be printed to the screen.

Note that Saahya is implementing this user program in a modified version of PintOS which includes the following two syscalls:

```
int time(); // returns current time in seconds
void sleep(unsigned int seconds); // puts the current thread to sleep for X seconds.
```

For all parts, you may assume that reminder strings are under 100 characters, and that all the intervals are on the order of minutes to hours (so you don't need to worry about integer overflow). Currently, the main thread of Saahya's tool looks like the following:

```
/* Returns input string from the user.
 * If there is no input ready, put the calling thread to sleep until input is available. */
char* get_user_input();

int main() {
    while (true) {
        char* input = get_user_input();
        // ... process user input ...
    }
}
```

(a) (1 pt) Saahya intends to implement the reminder tracking/printing logic on the main thread in the process user input section. Why is this a bad idea?

`get_user_input()` **blocks the main thread until user input is available. Thus, reminders will not be printed while waiting for user input.**

(b) (14 points)

Taking into account your concerns, Saahya refactors her existing codebase in preparation for adding reminders:

```
typedef struct reminder {
    char* message; // The reminder message
    int interval; // The interval at which the message is sent
    int target_time; // The timestamp marking when the reminder should be sent next;
    struct list_elem elem;
} reminder_t;

typedef struct reminder_list {
    struct list list; // List of reminder_t's
    struct lock lock; // Lock for accessing the list
    struct condition cond;
} reminder_list_t;

int main() {
    reminder_list_t reminders;
    initialize(&reminders);

    pthread_t* thread;
    pthread_create(&thread, NULL, runner, &reminders);

    /* ... long-running logic which will occasionally call
       add_reminder based on user input ... */
}

void initialize(reminder_list_t* reminders) {
    // TODO
}

void add_reminder(reminder_list_t* reminders, char* message, int interval) {
    // TODO
}

void wake_after_delay(void* arg) {
    // TODO
```

```
}

void runner(void* arg) {
    // TODO
}
```

Your task is to help Saahya fill in the unimplemented functions.

i. (1 pt) Consider the following example:

- At `time=0`, main calls `add_reminder(reminders, "Work on design!", 15);`
- At `time=20`, main calls `add_reminder(reminders, "Stay focused", 5);`
- At `time=31`, main exits the process, killing all threads.

Please list the `(time, message)` pairs for every message sent.

(15,"Work on design!")
(25,"Stay focused")
(30,"Work on design!")
(30,"Stay focused")

ii. (1.5 pt) Implement the `initialize` function, which initializes the `reminders` struct.

```
void initialize(reminder_list_t* reminders) {
    __________[A (x3)]_____________
}
```

[A] (max 3 lines)

```
void initialize(reminder_list_t* reminders) {
    list_init(&reminders->list);
    lock_init(&reminders->lock);
    cond_init(&reminders->cond);
}
```

iii. (3.5 points)

Implement the `add_reminder` function, which should add a new reminder struct into the `reminders` list and then signal the condition variable. Ensure that all necessary synchronization is in place.

Note: You may not assume that `*message` will persist beyond this function call, and you are not responsible for freeing it.

```
void add_reminder(reminder_list_t* reminders, char* message, int interval) {
    reminder_t* reminder = ___________[A]____________
    ___________[B (x3)]___________

    ____________[C]_____________
    ____________[D]_____________
    cond_signal(&reminders->cond, &reminders->lock);
    ____________[E]_____________
}
```

A. (0.5 pt) [A] (max 1 expression)

```
malloc(sizeof(reminder_t))
```

B. (1.5 pt) [B] (max 3 lines)

```
reminder->message = strdup(message);
reminder->interval = interval;
reminder->target_time = time() + interval;
```

C. (0.5 pt) [C] (max 1 line)

```
lock_acquire(&reminders->lock)
```

D. (0.5 pt) [D] (max 1 line)

```
list_push_back(&reminders->list, reminder)
```

E. (0.5 pt) [E] (max 1 line)

```
lock_release(&reminders->lock)
```

iv. (3 points)

In preparation for implementing runner, implement the `wake_after_delay` helper function. This function should sleep until `info->target_wake_time`, then signal the condition variable.

```
typedef struct delay_info {
    struct condition* cond; // Pointer to the reminder_list's cond
    struct lock* lock; // Pointer to the reminder_list's lock
    int target_wake_time; // Target time to signal cond
} delay_info_t;

void wake_after_delay(void* arg) {
    delay_info_t* info = (delay_info_t*) arg;
    int time_till_wake = ___________[A]___________ - time();
    if (___________[B]___________) {
        sleep(time_till_wake);
    }

    ___________[C]___________;
    cond_signal(___________[D]___________, ___________[E]___________);
    ___________[F]___________;

    free(info);
}
```

A. (0.5 pt) [A] (max 1 expression)

```
info->target_wake_time
```

B. (0.5 pt) [B] (max 1 expression)

```
time_till_wake > 0
```

C. (0.5 pt) [C] (max 1 line)

```
lock_acquire(info->lock)
```

D. (0.5 pt) [D] (max 1 expression)

```
info->cond
```

E. (0.5 pt) [E] (max 1 expression)

```
info->lock
```

F. (0.5 pt) [F] (max 1 line)

```
lock_release(info->lock)
```

v. (5 points)

Finally, implement the runner function. This function should implement the reminder-sending feature. If n is the number of notifications sent over some timespan, the number of while loop iterations must be at most $2n$ in all cases.

```
void runner(void* arg) {
    reminder_list_t* reminders = (reminder_list_t*) arg;
    ___________[A]___________;
    while (true) {
        cond_wait(&reminders->cond, &reminders->lock);
        int t = time();
        int target_wait_time = INT_MAX; // target_time of the closest reminder

        for (_______[B]_______; _______[C]_______; _______[D]_______) {
            reminder_t* reminder = ___________[E]___________;
            if (___________[F]___________) {
                printf(reminder->message);
                ___________[G]___________;
            }
            if (___________[H]___________) {
                target_wait_time = reminder->target_time;
            }
        }

        delay_info_t* info = ___________[I]___________;
        info->cond = &reminders->cond;
        info->lock = &reminders->lock;
        info->target_wake_time = target_wake_time;
        ___________[J]___________;
    }
}
```

A. (0.5 pt) [A] (max 1 line)

```
lock_acquire(&reminders->lock)
```

B. (0.5 pt) [B] (max 1 expression)

```
struct list_elem* e = list_begin(&reminders->list)
```

C. (0.5 pt) [C] (max 1 expression)

```
e != list_end(&reminders->list)
```

D. (0.5 pt) [D] (max 1 expression)

```
e = list_next(e)
```

E. **(0.5 pt)** [E] (max 1 expression)

```
list_entry(e, reminder_t, elem)
```

F. **(0.5 pt)** [F] (max 1 expression)

```
reminder->target_time <= t
```

G. **(0.5 pt)** [G] (max 1 line)

```
reminder->target_time += reminder->interval
```

H. **(0.5 pt)** [H] (max 1 expression)

```
reminder->target_time < target_wait_time
```

I. **(0.5 pt)** [I] (max 1 expression)

```
malloc(sizeof(delay_info_t))
```

J. **(0.5 pt)** [J] (max 1 line)

```
pthread_create(NULL, NULL, wake_after_delay, (void*) info)
```

6. (14 points) Hybrid Thundering Locks

WebKit—a browser engine developed by Apple—features optimized custom implementations of several synchronization primitives. In particular, it includes a high-throughput "adaptive lock" implementation which performs especially well under microcontention. *Microcontended locks* are locks that protect short critical sections, so they should be available soon after failed attempts to acquire them.

In this problem, you will implement a similar "hybrid thundering lock" in C by leveraging the simplified `ParkingLot` API and atomic operations. Note that limited busy waiting in parts of this problem is allowed, as long as it remains bounded within the described limits.

You have access to the following `ParkingLot` API functions, which enable efficient and safe thread queueing and dequeuing, similar to `futex`:

Parking threads effectively puts them to sleep and queues them.

```
/* Compares the `lock` and `desired_state` and parks the current thread if they match. */
void compare_and_park(hybrid_lock_t* lock, hybrid_lock_t desired_state);
```

Unparking threads dequeues them and wakes them up.

```
/* Unpark one thread that is waiting on `lock`. */
void unpark_one(hybrid_lock_t* lock);

/* Unpark all threads that are waiting on `lock`. */
void unpark_all(hybrid_lock_t* lock);
```

Assume you also have access to the compare-and-swap (CAS) and swap atomic operations:

```
/* pseudocode for behavior of the atomic compare and swap instruction */
bool CAS(int* addr, int expr1, int expr2) {
  if (*addr == expr1) {
    *addr = expr2;
    return true;
  } else {
    return false;
  }
}

/* pseudocode for behavior of the atomic swap instruction */
int swap(int* addr, int expr1) {
  int temp = *addr;
  *addr = expr1;
  return temp;
}
```

(a) (1 points) Lock Init

Each lock has 3 distinct states, which we represent with the following `typedef` and constants:

```
#define UNLOCKED 0 // lock is free
#define LOCKED 1 // lock is held and no parked threads are waiting for it
#define LOCKED_AND_PARKED 2 // lock is held and at least 1 parked thread is waiting for it

typedef int hybrid_lock_t;
```

Using the provided API's, implement the required lock functions by filling in the blanks below. You may find the reference sheet useful.

```
void lock_init(hybrid_lock_t* lock) {
    ______________[A]_________________
}
```

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

```
* lock = Unlocked;
```

(b) (8 points) Lock Acquire

Lock acquisition will have 2 phases:

i. A primary bounded spin-lock phase which spins for a maximum of 40 times before moving on
ii. A secondary slow phase which should park the requesting thread if the lock is not free.

```
void lock_acquire(hybrid_lock_t * lock) {
    // Bounded spin-lock phase
    for (____________[B]___________________) {
        if ( ________[C]_________ && CAS(_________[D]__________)) {
            return;
        }

        if ( ________[E]__________) {
            break;
        }
    }


    // Slow phase
    while (true) {
        if ( __________[F]__________ && CAS(________[G]__________)) {
            return;
        }
        CAS(___________[H]_____________);
        compare_and_park(__________[I]___________);
    }
}
```

i. (1 pt) [B] (max 3 lines)

```
int i = 0; i < 40; i++
```

ii. (1 pt) [C] (max 1 expression)

```
* lock == Unlocked
```

iii. (1 pt) [D] (parameters)

```
lock, Unlocked, Locked
```

iv. (1 pt) [E] (max 1 expression)

```
* lock == LockedAndParked
```

v. (1 pt) [F] (max 1 expression)

```
* lock == Unlocked
```

vi. (1 pt) [G] (parameters)

```
lock, Unlocked, Locked
```

vii. (1 pt) [H] (parameters)

```
lock, Locked, LockedAndParked
```

viii. (1 pt) [I] (parameters)

```
lock, LockedAndParked
```

(c) (3 points) Lock Release

Upon lock release, if any parked threads are waiting for the lock, they should be unparked, thus releasing a "thundering herd" of threads that will try to acquire it.

```
void lock_release(hybrid_lock_t * lock) {
    if (CAS(_______[J]_________)) {
        return;
    }
    if (swap(lock, Unlocked) == _________[K]_______) {
        ___________[L]____________
    }
}
```

i. (1 pt) [J] (parameters)

```
lock, Locked, Unlocked
```

ii. (1 pt) [K] (max 1 expression)

```
LOCKED_AND_PARKED
```

iii. (1 pt) [L] (max 1 line)

```
unpark_all(lock);
```

(d) (2 points) Design Review

i. (2 pt) Assume all uses of the hybrid thundering lock are under microcontention as defined above. What is one potential advantage and one potential disadvantage of this particular lock design?

Advantage: Spin-lock phase is ideal for micro contention as we avoid costly thread parking operation in most cases since waiting period is short Disadvantage: Releasing a thundering herd of threads at once could slow down system upon lock release with many threads waiting

7. Reference Sheet (feel free to remove)

```
/* Process */
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_destroy(pthread_cond_t *cond);
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_push_front(struct list *list, struct list_elem *elem);
void list_push_back(struct list *list, struct list_elem *elem);

/* 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 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)
```

8. Scratch Paper (feel free to remove)

9. Scratch Paper (feel free to remove)