CS 162 Fall 2022

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. You have 110 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) 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 points) True/False

Please explain your answer in **TWO SENTENCES OR LESS**. Longer explanations may get no credit. Answers without an explanation **GET NO CREDIT**.

(a) (2 points)

i. Base/bound prevents processes from accessing each others' memory in user mode.

- (●) True.
- ( ) False.

ii. Explain.

True: Using base/bound memory translation checks that a memory address accessed by a process is less than its defined bound.

(b) (2 points)

i. If a user types on a keyboard, this will generate a syscall to the application.

- ( ) True.
- (●) False.

ii. Explain.

False: A syscall is an internal interrupt, while an I/O operation like a keyboard input is external.

(c) (2 points)

i. When a server receives a client request and creates a new thread to handle the connection socket, both threads should call close on the connection socket file descriptor to make sure there are no leaked resources.

◯ True.

● False.

ii. Explain.

False: A server that creates a new thread to handle a client request only needs to close the socket once since this will close the socket for both the main and child threads in the shared file descriptor table. Contrast this with a multi-process server, which must close the socket twice for the main and child processes, which have independent FDTs.

(d) (2 points)

i. Hoare semantics make no assumptions about how a thread is scheduled after being taken off the wait queue.

◯ True.

● False.

ii. Explain.

False: Using Hoare semantics, we require that the waiting thread is scheduled immediately after the signaling thread (which must also immediately give up the monitor lock to the waiting thread). Using Mesa semantics, we have an additional check that the condition is false, so we do not require the waiting thread to be scheduled immediately after the signaling thread.

(e) (2 points)

i. In a multithreaded process, calling `exit(0)` in one of the threads will terminate that thread only.

- ○ True.
- ● False.

ii. Explain.

False: Calling `exit(0)` in any of the threads will exit the entire process, thus killing all the other threads that are still running.

(f) (2 points)

i. In Pintos, upon returning to user mode from a syscall, a process's registers are restored from its PCB.

- ○ True.
- ● False.

ii. Explain.

False: The state is restored from the interrupt frame.

(g) (2 points)

i. In Pintos, a virtual address less than `PHYS_BASE` in multiple different processes can map to multiple different physical addresses.

- (●) True.
- ( ) False.

ii. Explain.

True: User memory address can be shared by multiple user processes.

(h) (2 points)

i. Suppose multiple threads read from (but never write to) a global variable. Then we need to use a lock and require each thread to acquire the lock before reading from the global variable to ensure exclusion.

- ( ) True.
- (●) False.

ii. Explain.

False: Multiple readers do not require exclusion when accessing shared memory.

2. (8 points) Multiple Choice

Choose ALL that apply.

(a) (2 points)

i. Select all true statements. (Assume each option is a completely independent scenario).

- [x] If Process B is forked from Process A, they share the same open file descriptions immediately after the fork.
- [x] Suppose Process A and Process B have opened a single pipe with ends represented by file descriptors 3 (reading) and 4 (writing). The following sequence of events will cause an error: (i) Process A writes 5 chars to file descriptor `3` (ii) Process B reads 5 chars from file descriptor `4` (iii) Process B writes 5 chars to file descriptor `4` (iv) Process A reads 5 chars from file descriptor `3`
- [ ] Suppose Process B is the child of Process A, and both have file descriptor `3` in their respective FDTs. If Process A's FD `3` corresponds to `file.txt`, then Process B's FD `3` must also correspond to `file.txt`.
- [x] Suppose Process A opens `file.txt`, Process A creates Process B using `fork`, and Process A closes `file.txt`. Then Process B will still have `file.txt` open in its FDT.
- [ ] None of the above

A: Forking duplicates the process-specific FDT, meaning that Process B will have identical file descriptors pointing to the same file descriptions. B: Pipes are one way, meaning that one end can only be written to and the other can only be read from. Thus, the error occurs when Process A tries to write to file descriptor 3 in step 1. C: After a fork, the parent and child process have separate FDTs. This means that if Process A opens `file.txt` and Process B does not, file descriptor 3 may correspond to `file.txt` in Process A but not in Process B. D: Separate processes have disjoint FDTs, so Process A closing a file descriptor has no impact on Process B's FDT.

(b) (2 points)

i. Select all of the following that are true about semaphores.

- [ ] Semaphores can only be allocated on the heap.
- [ ] A semaphore's value can become negative.
- [ ] Semaphores can be used to implement locks, but not the other way around.
- [x] Semaphores can be used to implement condition variables.
- [ ] None of the above.

A: Semaphores can be initialized on the stack as well. B: `sema_down` blocks if the value would become negative, waiting until decrementing the value would result in a nonnegative value. C: A semaphore could conceivably be implemented using a variable-length array of locks. D: Condition variables can be implemented using semaphores, as demonstrated in `threads/synch.c` in the Pintos source code.

(c) (2 points)

i. Select all true statements about x86.

- [x] `pushl eax` subtracts 4 from `esp`.
- [ ] `popl eax` subtracts 4 from `esp`.
- [x] Using proper calling convention, a 'caller' pushes the arguments to a 'callee' in reverse order.
- [ ] Using proper calling convention, we store the base pointer (`ebp`) of the 'caller' function frame in a saved register like `eax`, so that we can restore it after the 'callee' has completed.
- [ ] None of the above

A: Pushing to the stack involves decrementing the stack pointer then putting the desired value at the stack pointer. B: Popping from the stack involves incrementing the stack pointer. C: Refer to Discussion 0 for details on x86 calling convention. D: The caller's base pointer is stored on the stack, not in `eax`. `eax` is used to stored the return value of the callee and is thus not a saved register.

(d) (2 points)

i. Select all true statements about the Unix process API.

- [ ] The return value of fork is either `> 0` (if this is the child process), `== 0` (if this is the parent process), or `< 0` (there was an error forking).
- [x] If a parent process has created two child processes, and calls `wait(NULL)` once, then it will halt execution until any one of its child processes exits.
- [ ] The `exec` syscall creates a child process to execute the specified program.
- [x] A process can use the `kill` syscall to terminate a child process.
- [ ] None of the above

A: The return values for the parent and child are reversed (i.e. `fork` returns `> 0` for the parent proces and `== 0` for the child process). B: `wait(NULL)` waits for the first child processes that exits. `waitpid` can be used to wait on a specific child process. C: In Unix, `exec` overwrites the current process to execute the specified program. D: Refer to the `man` pages for a description of how `kill` can be used.

3. (36 points) Short Answer

(a) (2 pt) List two ways an exception is different from an interrupt.

Exceptions are caused by unexpected synchronous internal events such as an instruction fault whereas interrupts handle asynchronous external events such as an I/O device attempting to transfer data.

(b) (2 pt) List 3 examples of how user processes can communicate (i.e., read/write information from/to one another).

Pipes, sockets, files, shared memory

(c) (3 pt) Provide 3 possible console outputs that can result from running the main function below. Assume all calls to `pthread_create`/`pthread_join` succeed.

```
void subtask(void* args_) {
    int idx = (int) args_;
    printf("%d ", idx);
    return;
}

int main(void) {
    pthread_t threads[6];
    for (int i = 1; i <= 6; i++) {
        pthread_create(threads[i - 1], NULL, subtask, (void*) i);
        if (i % 3 == 0) {
            for (int j = i - 2; j <= i; j++) {
                pthread_join(threads[j - 1], NULL);
            }
        }
    }
    return 0;
}
```

Any ordering of 1...6 where each "chunk" of three successive integers (1-3, 4-6) are in order (e.g., 6 can come before 5, but 6 cannot come before 3).

(d) **(3 pt)** Wilson and Zhuowen both want to use a shared TV to watch a show; Wilson wants to watch House of the Dragon, while Zhuowen wants to watch The Rings of Power. To decide which show to watch, they have decided on the following protocols for each to follow.

```
Protocol A (Wilson) {
    1. Leave a note on the TV saying ‘‘Wilson was here’’.
    2. Wait until there is no note on the TV saying ‘‘Zhuowen was here’’.
    3. If there is no note on the TV saying ‘‘Rings’’:
    4.   Leave a note saying ‘‘HOTD’’ on the TV.
    5. Remove the ‘‘Wilson was here’’ note.
}
Protocol B (Zhuowen) {
    1. Leave a note on the TV saying ‘‘Zhuowen was here’’;
    2. If there is no note saying ‘‘Wilson was here’’:
    3.   If there is no note saying ‘‘HOTD’’:
    4.     Leave a note saying ‘‘Rings’’;
    5. Remove the ‘‘Zhuowen was here’’ note;
}
```

Assume that only one person can take an action at a time (i.e., they can't both leave/remove notes at the same time). Provide a brief explanation why once both Wilson and Zhuowen have run through their protocols in any order, there will be exactly one note on the TV saying either "HOTD" or "Rings". **Limit your explanation to 4 sentences (longer answers may not be graded).**

Hint: Split into two cases. What happens if Line A1 happens before Line B2?

Split into two cases:
Case 1: Line A1 happens before Line B2 → Zhuowen cannot leave a note saying "Rings", so Wilson will leave a note saying "HOTD".
Case 2: Line A1 happens after Line B2 → Zhuowen will leave a note saying "Rings" and Wilson will wait until B has completed, so Wilson will not leave a note saying "HOTD".

(e) **(2 pt)** Explain two ways the syscall handler protects the kernel from corrupt or malicious user code in PintOS.

Possible Answers:
When executing a syscall, the user program specifies an index instead of the direct address of the handler, meaning the user program cannot directly execute in kernel mode.
The arguments will be validated by the handler to make sure the user is not intending a malicious attack.
The handler copies over the syscall arguments instead of using them from the user stack directly to prevent a time of check time of use attack.
After the syscall finishes, the results are copied back into user memory, instead of the user being able to access kernel memory.

(f) (10 pt) To test out the functionality of multi-threading, you write the following piece of code in silkworm.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

#define WORMS 8

typedef struct {
    pthread_t tid;
    char* msg;
} pthread_args;

static void* spawn_worm(void* arg) {
    pthread_args* args = (pthread_args*) arg;
    char msg[100];
    // copies formatted string to MSG
    sprintf(msg, "Worm #%ld", args->tid);
    args->msg = msg;
}

int main() {
    pthread_args args;
    int s;
    for (int i = 0; i < WORMS; i++) {
        s = pthread_create(&args.tid, NULL, &spawn_worm, &args);
        if (s != 0) {
            return 1;
        }
    }
    for (int i = 0; i < WORMS; i++) {
        s = pthread_join(args.tid, NULL);
        if (s == 0) {
            printf("%s\n", args.msg);
        }
    }
    return 0;
}
```

You run the program assuming the following:

- Newly spawned threads start at ID 1 and increment by 1 for each additional thread
- Your machine specification is good enough to not fail any memory allocations

You expect that this program prints out all of the worm numbers from 1 to 8 in sequential order:

```
| cs162@hive ~ $ gcc silkworm.c -o silkworm -lpthread
| cs162@hive ~ $ ./silkworm
| Worm 1
| Worm 2
| Worm 3
| Worm 4
| Worm 5
| Worm 6
| Worm 7
| Worm 8
| cs162@hive ~ $
```

However, upon compiling and running this program, you noticed that your output seems bugged; you get the following output:

```
| cs162@hive ~ $ gcc silkworm.c -o silkworm -lpthread
| cs162@hive ~ $ ./silkworm
|
| cs162@hive ~ $
```

Identify which lines of code **(5 lines)** should be replaced and their replacements such that the program will work as expected. **You are not allowed to delete or add lines.**

```
/* Line 10 */ char* msg = (char*) malloc(100 * sizeof(char));
/* Line 16 */ pthread_t tid[WORMS];
/* Line 18 */ int failed = pthread_create(&tid[i], NULL, &spawn_worm, &tid[i]);
/* Line 25 */ int failed = pthread_join(tid[i], &worm_msg);

Alternative:
/* Line 10 */ char* msg = (char*) calloc(100, sizeof(char));
/* Line 16 */ pthread_t* tid = (pthread_t*) malloc(WORMS * sizeof(pthread_t));
/* Line 18 */ int failed = pthread_create(&tid[i], NULL, &spawn_worm, &tid[i]);
/* Line 25 */ int failed = pthread_join(tid[i], &worm_msg);
```

This program has 2 issues:

Memory is allocated on the stack for the string, which is why we see an empty line being outputted.

Each thread reuses the same tid, so the first pthread_join will block until tid 8 finishes. The remainder of the pthread_join will fail.

Full credit was given for removing the # from the print statement, though this was an unintentional error.

4. (10 points) Semaphore Implementation

In the Pintos source code, we implement many synchronization primitives by disabling and re-enabling interrupts to ensure atomic behavior. An alternative to disabling interrupts is to use atomic hardware operations. Examples of atomic hardware operations include "Test and set", "Compare and Swap", and "Atomic Swap".

Your task is to implement a semaphore using atomic functions"Test and Set" and a made-up function called "Set and Block":

```
Test and Set (atomic)
int test_and_set(int* addr) {
    int res = *addr;
    *addr = 1;
    return res;
}

Set and Block (atomic)
void set_and_block(int* ptr, int val) {
    *ptr = val;
    thread_block();
}
```

`thread_block()`: Puts the current thread to sleep. It will not be scheduled again until awoken by `thread_unblock()`.

You may use the Pintos list API as you see fit. You may additionally find the following functions useful for your implementation:

`void thread_yield()`: Yields the CPU. The current thread is not put to sleep and may be scheduled again immediately at the scheduler's whim.

`void thread_unblock(struct thread* t)`: Transitions a blocked thread T to the ready-to-run state.

```
typedef struct semaphore {
    int value;
    int sema_lock;
    struct list waiters;
} sema_t;

void sema_init(sema_t* sema, unsigned value) {
    _________[A]__________
    _________[B]__________
    _________[C]__________
}

void sema_down(sema_t* sema) {
    while (_________[D]__________) {
        thread_yield();
    }

    if (_________[E]__________) {
        list_push_back(_________[F]__________, &thread_current()->elem);
        _________[G]__________
    } else {
        sema->value -= 1;
        _________[H]__________
    }
}

void sema_up(sema_t* sema) {
```

```
    while (_________[I]__________) {
        thread_yield();
    }
    if (!list_empty(&sema->waiters))
        thread_unblock(list_entry(list_pop_front(&sema->waiters), struct thread, elem));
    sema->value++;
    _________[J]__________;
}
```

(a) (1 pt) [A]

```
sema->value = 0
```

(b) (1 pt) [B]

```
sema->lock = 0
```

(c) (1 pt) [C]

```
list_init(&sema->waiters)
```

(d) (1 pt) [D]

```
test_and_set(&sema->lock)
```

(e) (1 pt) [E]

```
sema->value == 0
```

(f) (1 pt) [F]

```
&sema->waiters
```

(g) (1 pt) [G]

```
set_and_block(&sema->lock, 0)
```

(h) (1 pt) [H]

```
sema->lock = 0
```

(i) (1 pt) [I]

```
test_and_set(&sema->lock)
```

(j) (1 pt) [J]

```
sema->lock = 0;
```

5. (20 points) Wait Groups

Sometimes, it is useful to wait for a certain number of jobs to complete. One way of doing this is to use a synchronization structure called a wait group, which keeps a counter of how many jobs still need to finish and wakes up the waiting thread once its internal counter reaches zero.

In this problem, you will implement a wait group that implements the following API:

```
typedef struct wait_group {
    pthread_mutex_t lock;
    pthread_cond_t cv;
    int count;
} wait_group_t;

// Initializes a wait group with its counter at 0.
void wait_group_init(wait_group_t* wait_group);

// Adds delta, which may be negative, to the wait group's counter
// The counter may not go below zero. For example, if delta is -5
// and the counter is 4, the counter should be set to 0.
// Once the counter reaches 0, all waiting threads should be released.
void wait_group_add(wait_group_t* wait_group, int delta);

// Decrements the wait group's counter by 1.
void wait_group_done(wait_group_t* wait_group);

// Wait until the wait group's counter reaches 0.
// Keep in mind that multiple threads may be waiting on a wait group
// and that all of them should return from this function as soon
// as the counter reaches 0.
void wait_group_wait(wait_group_t* wait_group);
```

To clarify the interface, you might use a wait group as follows:

```
#include "waitgroup.h"
#include <stdio.h>
#include <pthread.h>

void* pthread_func(void* arg) {
    wait_group_t* wait_group = (wait_group_t*) arg;
    printf("Hello world!\n");
    wait_group_done(wait_group);
    return NULL;
}

int main(int argc, char** argv) {
    wait_group_t wait_group;
    wait_group_init(&wait_group);
    for (int i = 0; i < 5; i++) {
        wait_group_add(&wait_group, 1);
        pthread_t tid;
        pthread_create(&tid, NULL, pthread_func, &wait_group);
    }
    wait_group_wait(&wait_group);
    printf("All threads completed.\n");
}
```

Using pthread locks, semaphores, and/or condition variables, implement the required wait group methods by

filling in the blanks below. You may find the reference sheet useful.

NOTE: You should use AT MOST the number of lines specified for each section. Extra lines will not be graded. You may use loops, conditional statements, and other control structures in your solution as you see fit.

```
#include <stdio.h>
#include <pthread.h>

typedef struct wait_group {
    pthread_mutex_t lock;
    pthread_cond_t cv;
    int count;
} wait_group_t;

void wait_group_init(wait_group_t* wait_group) {
    ______________[A]x3______________
}

void wait_group_add(wait_group_t* wait_group, int delta) {
    ______________[B]x10_____________
}

void wait_group_done(wait_group_t* wait_group) {
    ______________[C]x1______________
}

void wait_group_wait(wait_group_t* wait_group) {
    ______________[D]x8______________
}
```

(a) (3 pt) [A] (max 3 lines)

```
void wait_group_init(wait_group_t* wait_group) {
        pthread_mutex_init(&wait_group->lock, NULL);
        pthread_cond_init(&wait_group->cv, NULL);
        wait_group->count = 0;
}
```

(b) (9 pt) [B] (max 10 lines)

```
void wait_group_add(wait_group_t* wait_group, int delta) {
        pthread_mutex_lock(&wait_group->lock);
        wait_group->count += delta;
        if (wait_group->count <= 0) {
                pthread_cond_broadcast(&wait_group->cv);
                wait_group->count = 0;
        }
        pthread_mutex_unlock(&wait_group->lock);
}
```

(c) (2 pt) [C] (max 1 lines)

```
void wait_group_done(wait_group_t* wait_group) {
        wait_group_add(wait_group, -1);
}
```

(d) (6 pt) [D] (max 8 lines)

```
void wait_group_wait(wait_group_t* wait_group) {
        pthread_mutex_lock(&wait_group->lock);
        while (wait_group->count > 0) {
                pthread_cond_wait(&wait_group->cv, &wait_group->lock);
        }
        pthread_mutex_unlock(&wait_group->lock);
}
```

6. (16 points) Syscall History

In Linux, a user can type in history to retrieve the most recent commands they've run. We are going to implement something similar - each user process will be able to retrieve each syscall it has run in its lifetime using a new type of syscall. This will be done in a Pintos-like system, but will not follow the Pintos skeleton source code exactly for simplicity.

The output format should be the following:

```
[syscall 1 name] [syscall 1 arglist]
[syscall 2 name] [syscall 2 arglist]
...
[syscall n name] [syscall n arglist]
```

The syscalls should be in order from earliest to latest. The list of syscalls printed does not include the history syscall, it is added to the list only after it returns. You must support any number of syscalls. You must not leak any memory, but you can assume that all calls to malloc succeed. You may not need all the lines provided, but cannot go over the limit.

If the user makes the syscalls "open a.txt", "write a.txt", "close a.txt", "ls", running the history syscall will print the following.

```
open a.txt
write a.txt
close a.txt
ls
```

```
struct sys_info {
    char* name;
    char* arglist;
    struct list_elem elem;
}

struct process {
    char* name;
    int pid;

    // TODO Add your own fields as needed.
    ______________[A]x3______________
}

/*
 * This function is called to init a process.
 * You can assume the process isn't run until after this function finishes.
 */
void process_init(struct process* process) {
    // Other setup code not shown...

    // TODO Implement this function.
    // You only need to implement the code relevant to the history syscall.
    ______________[B]x3______________
}

/*
 * This function is called to exit a process.
 * You can assume the process won't run after this function finishes.
 */
void process_exit(struct process* process) {
    // Other exit code not shown...
```

```
    // TODO Implement this function.
    // You only need to implement the code relevant to the history syscall.
    // Don't forget to avoid any memory leaks.
    ______________[C]x10______________
}

/*
 * This function is called at the end of every syscall.
 * You may not assume that the contents of sys_info
 * will be preserved after this function returns.
 */
void sys_end(struct process* process, struct sys_info* info) {
    // TODO Implement this function.
    ______________[D]x10______________
}

/*
 * This function should implement the actual history syscall.
 */
void sys_hist(struct process* process) {
    struct list_elem* e;
    for (e = list_begin(&process->sys_list); e != list_end(&process->sys_list); e = list_next(e)) {
        struct sys_info* sys_info = list_entry(e, struct sys_info, elem);
        // You can assume that printf doesn't call any syscalls.
        printf("%s %s\n", sys_info->name, sys_info->arglist);
    }
}
```

(a) (2 pt) [A] (max 3 lines)

```
struct list sys_list;
```

(b) (2 pt) [B] (max 3 lines)

```
list_init(&process->sys_list);
```

(c) (6 pt) [C] (max 10 lines)

```
struct list_elem* e;
for (e = list_begin(&process->sys_list);
        e != list_end(&process->sys_list); ) {
    struct sys_info* sys_info = list_entry(e, struct sys_info, elem);
    free(sys_info->name);
    free(sys_info->arglist);
    e = list_next(e); // We must set the next list_elem before we free the sys_info struct.
    free(sys_info);
}

Alternative:
while (!list_empty(&process->sys_list)) {
    struct list_elem* e = list_pop_front(&process->sys_list);
    struct sys_info* sys_info = list_entry(e, struct sys_info, elem);
    free(sys_info->name);
    free(sys_info->arglist);
    free(sys_info);
}
```

(d) (6 pt) [D] (max 10 lines)

```
struct sys_info* proc_sys_info = malloc(sizeof(struct sys_info));
proc_sys_info->name = malloc(strlen(sys_info->name) + 1);
proc_sys_info->arglist = malloc(strlen(sys_info->arglist) + 1);
strlcpy(proc_sys_info->name, sys_info->name, strlen(sys_info->name));
strlcpy(proc_sys_info->arglist, sys_info->arglist, strlen(sys_info->arglist));

list_push_back(&process->sys_list, &proc_sys_info->elem);
```

7. (20 points) I/O Library

Let's implement some high-level I/O library functions! If you recall from class, high-level I/O functions like fopen, fclose, etc. improve upon existing low-level I/O syscalls like read, write, etc. by buffering in user memory. We're going to create a simple high-level API for a **read-only filesystem** defined below:

```
/* store an open stream's data */
struct BFILE {
    int fd;                 // file descriptor
    int8_t buffer[1024];    // 1024 byte buffer
    int start;              // starting position of file data in buffer
    int end;                // ending position of file data in buffer
};

/* open file and create new stream */
struct BFILE* bopen(char* file_name);

/* close file and return whether close succeeded */
bool bclose(struct BFILE* bfile);

/* read COUNT bytes into BUFFER from BFILE stream and return the number of bytes read */
int bread(struct BFILE* bfile, int8_t* buffer, int count);
```

Additionally, we are going to assume that **there will be at most ONE `struct BFILE` referring to a given file across the entire system**. That way, we don't have to worry about multiple processes reading from the same file and altering their shared position within the file. Additionally, when we call `bclose` on a `struct BFILE` we can be sure to close the underlying file descriptor, since we know our process has no other `struct BFILE`s that reference the same file.

(a) Part 1: Opening and Closing

We're going to use a `struct BFILE` stored on the heap to keep track of our file information. We also need to make sure we clean up all used memory when we successfully close a file.

```
/* open file and create new stream */
struct BFILE* bopen(char* file_name) {
    struct BFILE* bfile = ________[A]__________;
    if (bfile == NULL) {
        return NULL;
    }
    bfile->________[B]__________ = open(________[C]__________, O_RDONLY|O_CREAT));
    if (bfile->________[D]__________ == -1) {
        ________[E]__________
        return NULL;
    }
    bfile->________[F]__________ = 0;
    bfile->________[G]__________ = 0
    return bfile;
}

/* close file and return whether close succeeded */
bool bclose(struct BFILE* bfile) {
    int success = close(________[H]__________);
    if (success == -1) {
        return false;
    }
    ________[I]__________;
    return true;
```

```
}
```

i. **(1 pt)** [A]

```
malloc(sizeof(struct BFILE))
```

ii. **(1 pt)** [B]

```
fd
```

iii. **(1 pt)** [C]

```
file_name
```

iv. **(1 pt)** [D]

```
fd
```

v. **(1 pt)** [E]

```
free(bfile);
```

vi. **(1 pt)** [F]

```
start
```

vii. **(1 pt)** [G]

```
end
```

viii. **(1 pt)** [H]

```
bfile->fd
```

ix. **(1 pt)** [I]

```
free(bfile)
```

(b) Part 2: Reading

Now we want to read from our file and we should use the buffer in our `struct BFILE` to store file data. So, we should copy memory from our bfile buffer when we can, and only call `read` to fill the bfile buffer with more file data. Also recall that `bread` should return the total number of bytes read into `BUFFER`.

```
#include <stdlib.h>

/* read COUNT bytes into BUFFER from BFILE stream and return the number of bytes read */
int bread(struct BFILE* bfile, int8_t* buffer, int count) {
    int read_in = 0;
    int8_t* curr_pos = buffer;
    while (read_in < count) {
        if (bfile->start < bfile->end) {
            // read in from bfile's buffer
            int remaining_to_read = count - read_in;
            int remaining_in_buff = ________[A]__________;
            int total_to_copy = min(remaining_to_read, remaining_in_buff);
            memcpy(curr_pos, bfile->buffer + ________[B]__________,
                   ________[C]__________);
            bfile->________[D]__________ += total_to_copy;
            curr_pos += total_to_copy;
            read_in += total_to_copy;
        } else {
            // fill bfile's buffer with as many bytes as possible
            int buff_read_in = 0;
            int buff_count = 1024;
            bfile->start = 0;
            bfile->end = 0;
            while (buff_read_in < buff_count) {
                // keep reading until we fail, hit EOF, or fill our BFILE's buffer
                int curr_read = read(________[E]__________,
                                     bfile->buffer + ________[F]__________,
                                     ________[G]__________);
                if (curr_read == -1 || curr_read == 0) {
                    break;
                }
                bfile->________[H]__________ += curr_read;
                buff_read_in += curr_read;
            }
            // prematurely return if we did not read any bytes from file
            if (bfile->________[I]__________ == 0) {
                return ________[J]__________;
            }

        }
    }
    return ________[K]__________;
}
```

i. (1 pt) [A]

`bfile->end - bfile->start`

ii. **(1 pt)** [B]

```
bfile->start
```

iii. **(1 pt)** [C]

```
total_to_copy
```

iv. **(1 pt)** [D]

```
bfile->start
```

v. **(1 pt)** [E]

```
bfile->fd
```

vi. **(1 pt)** [F]

```
bfile->end
```

vii. **(1 pt)** [G]

```
buff_count - buff_read_in
```

viii. **(1 pt)** [H]

```
end
```

ix. **(1 pt)** [I]

```
end
```

x. **(1 pt)** [J]

```
read_in
```

xi. **(1 pt)** [K]

```
read_in
```

8. Reference Sheet

```
/* Semaphore */
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);

/* Lock */
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

/* Condition Variable */
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);

/* 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[]);

/* Thread */
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);

/* 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);
```