Computer Science 162
Sam Kumar
University of California, Berkeley
Quiz 1
July 6, 2020

Name	
Student ID	

This is an open-book exam. You may access existing materials, including online materials (such as the course website). During the quiz, you may **not** communicate with other people regarding the quiz questions or answers in any capacity. For example, posting to a forum asking for help is prohibited, as is sharing your exam questions or answers with other people (or soliciting this information from them).

You have 80 minutes to complete it. If there is something in a question that you believe is open to interpretation, please make a private post on Piazza asking for clarification. If you run into technical issues during the quiz, join our open Zoom call and we will try to help you out. The final page is for reference.

We will overlook minor syntax errors in grading coding questions. You do not have to add the necessary `#include` statements at the top.

Grade Table (for instructor use only)

Question:	1	2	3	4	5	6	7	Total
Points:	1	15	10	10	18	26	0	80
Score:								

---

1. (1 point) Please check the boxes next to the following statements after you have read through and agreed with them.

   ☐ I will complete this quiz with integrity, doing the work entirely on my own.

   ☐ I will NOT attempt to obtain answers or partial answers from any other people.

   ☐ I will NOT post questions about this quiz on online platforms such as StackOverflow.

   ☐ I will NOT discuss any information about this quiz until 24 hours after it is over.

   ☐ I understand the consequences of violating UC Berkeley's Code of Student Conduct.

2. (15 points) **Operating System Concepts**

   Choose **either** True or False for the questions below. You do not need to provide justifications. **Each student received 15 questions, chosen randomly from the ones below.**

   (a) (1 point) The processor can be switched to kernel mode during execution of a user process, if the user process issues a system call.
   - ◯ True
   - ◯ False

   (b) (1 point) The processor can be switched to kernel mode during execution of a user process, *without* the user process issuing a system call.
   - ◯ True
   - ◯ False

   (c) (1 point) The number of threads in the system may exceed the number of CPU cores.
   - ◯ True
   - ◯ False

   (d) (1 point) To safely process system calls, the kernel *must* maintain a separate stack in the kernel for each thread in each user process.
   - ◯ True
   - ◯ False

   (e) (1 point) When a process issues a system call in Pintos, the kernel must change the active page table (i.e., update the page table base register) so that the system call handler can access kernel memory.
   - ◯ True
   - ◯ False

   (f) (1 point) Different threads within the same process share the same stack.
   - ◯ True
   - ◯ False

   (g) (1 point) A thread can overwrite the stack of another thread in the same process.
   - ◯ True
   - ◯ False

   (h) (1 point) Different threads within the same process share the same heap.
   - ◯ True
   - ◯ False

   (i) (1 point) On a multi-core system, different threads within the same process may simultaneously run on different cores.
   - ◯ True
   - ◯ False

   (j) (1 point) The `execve` system call creates a new process.
   - ◯ True
   - ◯ False

   (k) (1 point) A successful call to `fork` is *guaranteed* to return different values in the parent and child processes.

◯ True
◯ False

(l) (1 point) A child process can call `wait` to wait for its parent to exit.

◯ True
◯ False

(m) (1 point) The `execve` system call closes any open file descriptors in the calling process before executing the new program.

◯ True
◯ False

(n) (1 point) When a process issues a system call, the processor uses the type of the system call (e.g., `open` vs. `read` vs. `write`) to determine which element of the interrupt vector table to dispatch to.

◯ True
◯ False

(o) (1 point) Every successful call to `fread` *necessarily* results in a `read` system call.

◯ True
◯ False

(p) (1 point) Every successful call to `fclose` *necessarily* results in a `close` system call.

◯ True
◯ False

(q) (1 point) If a process allocates memory with `malloc` but exits without `free`ing it, then the allocated memory remains unavailable to other processes until the system is rebooted.

◯ True
◯ False

(r) (1 point) On a successful call to `open`, the returned file descriptor number is *guaranteed* to be unique across all active file descriptors in all threads/processes in the system.

◯ True
◯ False

(s) (1 point) On a successful call to `fork`, the returned child PID is *guaranteed* to be unique across all PIDs of running processes in the system.

◯ True
◯ False

(t) (1 point) When a process reads a file using a `FILE*`, the file data is buffered twice, once by the kernel and again in the process address space.

◯ True
◯ False

(u) (1 point) In a Unix-like operating system, a process can use the same two system calls, `read` and `write`, for interacting with files, pipes, and sockets.

◯ True
◯ False

(v) (1 point) Processes on *different* systems may communicate over the network using sockets.

- ◯ True
- ◯ False

(w) (1 point) Processes on *the same* system may communicate using sockets.

- ◯ True
- ◯ False

(x) (1 point) An RPC client can communicate with a server by calling `read` and `write` on the socket returned by the `socket` system call.

- ◯ True
- ◯ False

(y) (1 point) An RPC server can communicate with a client by calling `read` and `write` on the socket returned by the `socket` system call.

- ◯ True
- ◯ False

3. (10 points) **Operating System Abstractions**

For each question below, select **all** of the choices that apply. You should assume:

- Calls to `open`, `fopen`, `fork`, `pthread_create`, `malloc`, and `realloc` always succeed.
- Calls to `read`, `write`, `dup`, and `dup2` succeed **if a valid file descriptor is provided**.
- The necessary header files from the C standard library are `#included`.
- Before each program is run, `file.txt` is an empty file.
- All threads *eventually* make progress. Make no other assumptions about the scheduler.

(a) (2 points) Which of the following could be the contents of `file.txt` after all processes of the program below terminate?

```
int main(int argc, char** argv) {
    if (fork() == 0) {
        int fd1 = open("file.txt", O_WRONLY);
        write(fd1, "a", 1);
    } else {
        int fd2 = open("file.txt", O_WRONLY);
        write(fd2, "b", 1);
    }
}
```

☐ (empty) ☐ `a` ☐ `b` ☐ `aa` ☐ `ab` ☐ `ba` ☐ `bb`

(b) (2 points) Which of the following could be the contents of `file.txt` after all processes of the program below terminate?

```
char buffer;
int main(int argc, char** argv) {
    int fd = open("file.txt", O_WRONLY);
    if (fork() == 0) {
        buffer = 'a';
    } else {
        buffer = 'b';
    }
    write(fd, &buffer, 1);
}
```

☐ (empty) ☐ `a` ☐ `b` ☐ `aa` ☐ `ab` ☐ `ba` ☐ `bb`

(c) (2 points) Which of the following could be the contents of `file.txt` after the program below terminates?

```
int fd;
void* helper(void* arg) {
    write(fd, "a", 1);
}
int main(int argc, char** argv) {
    fd = open("file.txt", O_WRONLY);
    pthread_t thread;
    pthread_create(&thread, NULL, helper, NULL);
    write(fd, "b", 1);
}
```

☐ (empty) ☐ a ☐ b ☐ aa ☐ ab ☐ ba ☐ bb

(d) (2 points) Which of the following could be the contents of `file.txt` after the program below terminates?

```
int fd;
void* helper(void* arg) {
    write(fd, "a", 1);
    pthread_exit(NULL);
}
int main(int argc, char** argv) {
    fd = open("file.txt", O_WRONLY);
    pthread_t thread;
    pthread_create(&thread, NULL, helper, NULL);
    write(fd, "b", 1);
    pthread_exit(NULL);
}
```

☐ (empty) ☐ a ☐ b ☐ aa ☐ ab ☐ ba ☐ bb

(e) (2 points) Which of the following could be the contents of `file.txt` after the program below terminates?

```
int fd;
char buffer;
void* helper(void* arg) {
    buffer = 'a';
    write(fd, &buffer, 1);
    pthread_exit(NULL);
}
int main(int argc, char** argv) {
    fd = open("file.txt", O_WRONLY);
    pthread_t thread;
    pthread_create(&thread, NULL, helper, NULL);
    buffer = 'b';
    write(fd, &buffer, 1);
    pthread_exit(NULL);
}
```

☐ (empty) ☐ a ☐ b ☐ aa ☐ ab ☐ ba ☐ bb

4. (10 points) **Short Answer**

   Answer the following questions. **Each student received 5 questions, taken from the ones below.**

   (a) (2 points) List a disadvantage of the POSIX File I/O abstraction on a single-node system (i.e., non-distributed system).

   (b) (2 points) When might it be appropriate to use `FILE*` instead of file descriptors?

   (c) (2 points) Suppose you are writing an OS for a processor that supports paged address translation but not dual-mode operation. Is it possible to protect processes from each other and the kernel from buggy/malicious processes? Explain.

   (d) (2 points) In Project 0, you saw that when a user program in Pintos is started, it begins executing in `_start`, which calls `main`. What problems might occur if we just started the user process at the beginning of `main`, without using `_start`?

   (e) (2 points) What is the role of DNS in RPC?

(f) (2 points) Does it ever make sense to make an RPC to a local process on the same machine, as opposed to a remote process on a different machine? Explain.

(g) (2 points) Suppose that a process has multiple connections open to the same server, obtained by making multiple calls to `connect`. When the OS receives data (e.g., a packet) from the server, how does it know which connection it belongs to?

(h) (2 points) Explain why calling `fork` in a program that uses `FILE*` might lead to unexpected results. What operation can we call on each `FILE*` to prevent this from occurring?

5. (18 points) **System Programming**

   (a) (2 points) In the 32-bit x86 assembly below, the function `foo` intends to make the function call `bar(1, 2, 3)`. Fill in **one** of the lines below with a single assembly instruction, so that the function call to `bar` is made with a properly aligned stack. **Leave the other lines blank.** You may assume that the stack was properly aligned when calling `foo`.

```
foo:
    pushl %ebp

    ______________________

    movl %esp, %ebp

    ______________________

    pushl $3
    pushl $2
    pushl $1

    ______________________

    call bar
    leave
    ret
```

   (b) (10 points) Write the function `void run(void)` that invokes the program `/bin/foo` and the program `/bin/bar` as follows:

   - `foo` should read its standard input from the file `input.txt`.
   - `bar` should write its standard output to the file `output.txt`.
   - `foo`'s standard output should be redirected into the standard input of `bar`.

   For this problem, you may assume that all system calls succeed when provided with valid arguments; for simplicity, you need not check return values to handle error cases. You do not have to add the necessary `#include` statements at the top. Where possible, you should make sure to close *all* file descriptors that you create once they are no longer necessary. `run` should return after spawning the two processes, without waiting for them to complete.

 Student ID: ____________________

(c) (6 points) Implement `write_all`, a function that writes the contents of `buffer` to the file descriptor `fd`. It should write all `len` bytes, unless an error occurs. It should return `-1` if there was an error, or otherwise the number of bytes written (which should be `len`). Where possible, you must avoid making a separate system call for each byte of the buffer. **Remember that `write` may return short, so use a loop as appropriate. You do not have to add the necessary `#include` statements at the top.**

6. (26 points) **Remote Procedure Calls**

An RPC server's `main` function is as follows:

```
int main(int argc, char **argv) {
    struct addrinfo *server = setup_address(argv[1]);
    if (server == NULL) {
        return 1;
    }
    int server_socket = socket(server->ai_family, server->ai_socktype,
                               server->ai_protocol);
    if (server_socket == -1) {
        return 1;
    }
    if (bind(server_socket, server->ai_addr, server->ai_addrlen) == -1) {
        return 1;
    }
    if (listen(server_socket, 50) == -1) {
        return 1;
    }
    run_service(server_socket);
    return 0;
}
```

The `run_service` function is intended to receive and process client requests in a loop. Currently, the `run_service` function creates a new process to serve each client, as follows:

```
void run_service(int server_socket) {
    for (;;) {
        int client_socket = accept(server_socket, NULL, NULL);
        if (client_socket == -1) {
            return;
        }
        if (fork() == 0) {
            close(server_socket);
            serve_client(client_socket);
            exit(0);
        } else {
            close(client_socket);
        }
    }
}
```

(a) (3 points) Suppose that the server has handled n RPCs from clients. How many times has the server called `socket`, `bind`, `listen`, `connect`, and `accept`? (List a count for each one.)

(b) (1 point) Would deleting the line `close(server_socket);` result in a resource leak? Explain.

(c) (1 point) Would deleting the line `close(client_socket);` result in a resource leak? Explain.

(d) (2 points) Suppose that, instead of handling each RPC call in a separate process, we handle each RPC call in a separate thread, within the same process.

i. (1 point) Should the child thread execute `close(server_socket)`? Explain.

ii. (1 point) Should the parent thread execute `close(client_socket)`? Explain.

(e) (3 points) What might be a reason to create a *thread*, rather than a *process*, for each RPC call?

(f) (3 points) Is there any benefit to using a *process* for each RPC call, rather than a *thread*? Explain.

(g) (6 points) Implement a new version of `run_service` such that a new process is still created to serve each client, but the maximum number of clients being served at a time is at most 10. You do not have to add the necessary `#include` statements at the top.

(h) (6 points) We would like to extend this RPC server by adding support for a new operation, `char* get_student_info(uint32_t);`. This function takes in one argument, the Student ID (represented as a `uint32_t`). Its return value is the student's email address as a string.

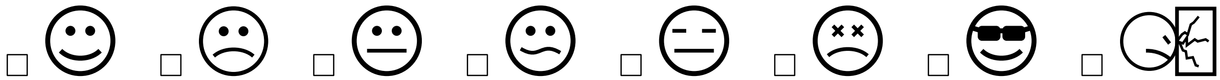
i. (3 points) Write the function
`void marshal_student_info(char* email_addr, int sock_fd);`
which marshals the student's email address and writes it to the connection socket. You may assume that `email_addr` is provided to `marshal_student_info` as a null-terminated string, and you may use your `write_all` implementation from earlier in the exam if it is useful. Assume that no error will occur when writing to the socket (i.e., you may omit error handling for simplicity). You do not have to add the necessary `#include` statements at the top.

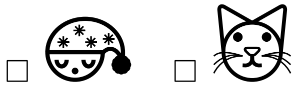
ii. (2 points) Should `marshal_student_info` be called by the server stub or client stub? Explain.

iii. (2 points) After the connection is set up, is there any other data that the client should send to the server other than the Student ID? Explain.

7. (0 points) **Optional Questions**

   (a) (0 points) Having finished the exam, how do you feel about it? Check **all** that apply:

    ☐ Other (please draw):

   (b) (0 points) If there's anything you'd like to tell the course staff (e.g., feedback about the class or exam, suspicious activity during the exam, new logo suggestions, etc.) you can write it on this page.

```
/****************************** Threads ******************************/
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);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_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);
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
/***************************** 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[]);
void exit(int status);
/*************************** High-Level I/O ****************************/
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
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);
int fclose(FILE *stream);
/**************************** Low-Level I/O ****************************/
int open(const char *pathname, int flags); (O_APPEND|O_CREAT|O_TMPFILE|O_TRUNC)
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int pipe(int pipefd[2]);
int close(int fd);
/**************************** Pintos Lists *****************************/
void list_init(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);
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);
struct list_elem *list_remove(struct list_elem *elem);
struct list_elem *list_pop_front(struct list *list);
struct list_elem *list_pop_back(struct list *list);
bool list_empty(struct list *list);
#define list_entry(LIST_ELEM, STRUCT, MEMBER) ...
```