University of California, Berkeley
College of Engineering
Computer Science Division: EECS

Summer 2019 Jack Kolb

Midterm I

July 18th, 2019
CS162: Operating Systems and Systems Programming

Your Name:	
SID AND 162 Login (e.g. s042):	
TA Name:	
Discussion Section Time:	

General Information:
This is a **closed book** exam. You are allowed 1 page of handwritten notes (both sides). You have two (2) hours 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	Score
1	22	
2	12	
3	16	
4	16	
5	16	
Total	82	

Problem 1: True/False [22 pts]

Please **explain** your answer in **two sentences or fewer** (Answers longer than this may not get credit!). Also, answers without an explanation ***get no credit***.

Problem 1a[2pts]: The number of allocated kernel stacks is always equal to the number of allocated userspace stacks.

☐ True ☐ False

Explain:

Problem 1b[2pts]: When using a simple "Base and Bound" scheme to enforce memory protection, the CPU only sees virtual memory addresses.

☐ True ☐ False

Explain:

Problem 1c[2pts]: When fork returns a negative integer, an error has occurred that must be dealt with in both the parent and child process.

☐ True ☐ False

Explain:

Problem 1d[2pts]: Switching between two threads within the same process is generally more efficient than switching between two threads belonging to different processes.

☐ True ☐ False

Explain:

Problem 1e[2pts]: The Linux Completely Fair Scheduler always gives a shorter time slice to lower priority threads.

☐ True ☐ False

Explain:

Problem 1f[2pts]: When using an atomic operation like `test&set` to implement a lock, we must apply the `test&set` operation to check the lock's state (either `BUSY` or `FREE`) within a "spin loop."

☐ True ☐ False

Explain:

Problem 1g[2pts]: Locks can prevent a thread from being preempted.

☐ True ☐ False

Explain:

Problem 1h[2pts]: If the Banker's algorithm blocks a request for a resource, then granting that request would have caused the system to deadlock.

☐ True ☐ False

Explain:

Problem 1i[2pts]: Synchronization primitives in base Pintos are implemented with `test&set`.

☐ True ☐ False

Explain:

Problem 1j[2pts]: According to the End-to-End Principle, reliable transport should be implemented by the two communications endpoints, not the network infrastructure.

□ True □ False

Explain:

Problem 1k[2pts]: After a call to `fork`, `stdin`, `stdout`, and `stderr` are reset to their default states in the child process.

□ True □ False

Explain:

Problem 2: Short Answer [12 pts]

Problem 2a[3pts]: What is the Interrupt Vector Table and what role does it play in protecting the kernel?

Problem 2b[3pts]: Why are device drivers divided into a top half and bottom half? What is each half responsible for?

Problem 2c[3pts]: In Pintos, how would we allow a `struct thread` to be an element of two lists at the same time?

Problem 2d[3pts]: Does First-Come First-Served or Round-Robin scheduling have lower overhead? Explain.

Problem 3: Processes and Syscalls [16 pts]

Assume that we have the following two pieces of code (with all header files included):

`write_and_print.c`, compiled to the binary file `write_and_print`

```
/* Takes a file descriptor (int) and a buffer (string) as input and
   writes the buffer into the provided file descriptor. ALSO prints
   out the buffer to stdout. */
```

`main.c`, compiled to the binary file `main`

```
int new_fd = -1;

void signal_handler(int sig) {
    dup2(new_fd, STDOUT_FILENO);
}

int main() {
    int hello_fd = open("greetings.txt", O_WRONLY | O_CREAT | O_TRUNC);
    char *child_buf = "Child hello!";
    char *parent_buf = "Parent howdy!";

    /* Assume char *arguments[] is correctly formed with ./write_and_print
       hello_fd, and child_buf as arguments */

    new_fd = dup(STDOUT_FILENO);
    dup2(hello_fd, STDOUT_FILENO);

    struct sigaction sa;
    sa.sa_flags = 0;
    sa.sa_handler = &signal_handler;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGTERM, &sa, NULL);

    pid_t pid = fork();
    if (pid == 0) {
        execv(arguments[0], arguments);
        kill(getppid(), SIGTERM);
    } else {
        kill(pid, SIGTERM);
        wait(NULL);
        dprintf(hello_fd, "%s: %d\n", parent_buf, pid);
        printf("%s: %d\n", parent_buf, pid);
    }

    return 0;
}
```

List all possible outputs of the `main` program in each row of the table below (where one row corresponds to one run of the program). You may not need all the tables provided. Assume that all system calls succeed EXCEPT that `execv()` may possibly fail and assume that *the child's PID is 162*.

Standard Output	`greetings.txt`

Problem 4: DeadLockList [16 pts]

Thanos needs to collect all 6 Infinity Locks. He creates a new structure called a `locklist` to manage the locks he has acquired or still needs. Instead of just acquiring one lock, threads now have to acquire a series of locks to 'lock' the `locklist` structure. Think of this as needing a collection of resources before being able to perform your operations. For all questions, assume the `locklist` has already been initialized for you and that we cannot preempt locks.

```
#DEFINE NUM_LOCKS 6
struct locklist {
      lock my_locks[NUM_LOCKS];
      int magic;
};
```

Problem 4a[2pts]: If all locks are released one after another with no other operations between them, does the sequence they are released in matter with respect to deadlock? *(Short Answer)*

Problem 4b[2pts]: Does the lock release order matter with respect to performance? Explain. *(Short Ans.)*

Thanos is now trying to decide in what order to lock the Infinity Locks in the `locklist`. Which of the following definitions of `lock_locklist` can cause deadlock? Explain your answer for each selection. If deadlock is possible, please provide an acquisition ordering in your explanation.

Problem 4c[2pts]: Lock acquisition code:

```
void lock_locklist(struct locklist *list) {
      int start = 0;
      for (int i = start; i < NUM_LOCKS; i++) {
            int index = i;
            lock_acquire (list->my_locks + index);
      }
}
```

☐ Can Cause Deadlock ☐ Cannot Cause Deadlock

Explain:

Problem 4d[2pts]: Lock acquisition code:

```
void lock_locklist(struct locklist *list) {
      int start = getpid() % NUM_LOCKS;
      for (int i = start; i < start + NUM_LOCKS; i++) {
            int index = i % NUM_LOCKS;
            lock_acquire (list->my_locks + index);
      }
}
```

☐ Can Cause Deadlock ☐ Cannot Cause Deadlock

Explain:

Problem 4e[2pts]: Lock acquisition code:

```
void lock_locklist(struct locklist *list) {
      int start = NUM_LOCKS - 1;
      for (int i = start; i >= 0; i--) {
            int index = i;
            lock_acquire (list->my_locks + index);
      }
}
```

☐ Can Cause Deadlock ☐ Cannot Cause Deadlock

Explain:

Problem 4f[2pts]: Lock acquisition code:

```
void lock_locklist(struct locklist *list) {
      int start = NUM_LOCKS/2;
      for (int i = start; i < start + NUM_LOCKS; i++) {
            int index = i % NUM_LOCKS;
            lock_acquire (list->my_locks + index);
      }
}
```

☐ Can Cause Deadlock ☐ Cannot Cause Deadlock

Explain:

Thinking about how much work it will take to lock the `locklist` structure, Thanos doesn't always want to go through the trouble of acquiring all the Infinity Locks. He decides to create the following function to access the structure faster, acquiring just one lock and later releasing it. **Assume all other threads execute the code from problem 4c when using the `locklist`.**

```
void snap(struct locklist *list) {
      lock_acquire (list->my_locks + OFFSET);
}
```

Problem 4g[2pts]: If `OFFSET = 0`, can `snap` cause deadlock to happen? If so, provide an example execution order that leads to deadlock. If not, explain why deadlock is not possible. *(Short Answer)*

Problem 4h[2pts]: If `OFFSET = NUM_LOCKS - 1`, can `snap` cause deadlock to happen? If so, provide an example execution order that leads to deadlock. If not, explain why deadlock is not possible. *(Short Answer)*

Problem 5: Just One TCP Connection [16 pts]

Bob is trying to build a system which helps people process jobs. After learning about how sockets work in CS162, he wants to write a server program which listens on port 162 and takes jobs from clients. Each job is represented by the following data structure:

```
struct Job {
   int job_number;
   char job_details[200];
   int result;
};
```

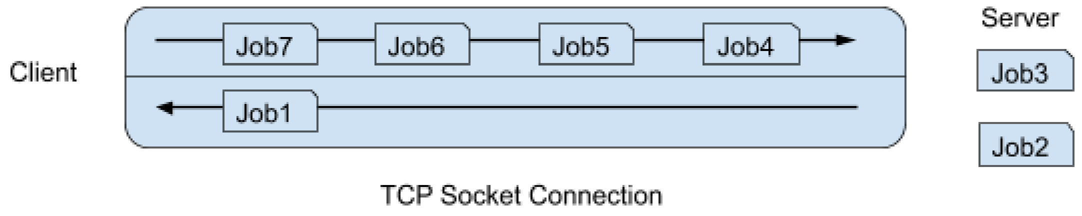
However, Bob thinks that having clients make a new connection for each job they want the server to process seems to be an inefficient design because it consumes resources unnecessarily.

Problem 5a[2pts]: Briefly explain why establishing new connections consumes additional resources *in userspace.*

Problem 5b[2pts]: Briefly explain why establishing new connections consumes additional resources *in the OS kernel.*

Bob comes up with a scheme where the server just needs to maintain, for each client, a single connection through which multiple jobs can be sent and through which processed jobs can be sent back. More specifically, *each connection is managed by one thread* and *whenever a new job is received, a new thread is created to do the work in parallel and then send back the results via the same connection*.

Graphically, it looks like this:



Problem 5c[2pts]: Bob approaches you for help with implementing his design. He's done most of the coding but left out some critical functions. You are required to fill in the missing lines below. Note that you might not need all the lines provided.

Assumptions:

1. When you call `ssize_t read(int fd, void *buf, size_t count)`, it always reads `count` bytes if there is data available. Otherwise, the call blocks.
2. When you call `ssize_t write(int fd, const void *buf, size_t count)`, it always writes `count` bytes successfully. You do not have to handle the case when `read` fails because of client disconnection.

```
struct Job {
   int job_number;
   char job_details[200];
   int result;
};

struct Arg_struct{
  Job *job;
  int socket_fd;
}

void do_work (Job *job) {
  /* This function does the work and set the result back into job */
  /* This function is compute-intensive */
}

void *new_conn(void *arg) {
  /* Handles a new client connection */
  struct  Job         *new_job;
  struct  Arg_struct  *new_args;
  ssize_t bytes_read;

  int  con_sockfd = (int) arg;
  int pending_jobs = 0;
  while (1) {

    new_job = ___________________________________________________

    bytes_read = ________________________________________________

    new_args = __________________________________________________

    new_args->job = new_job;
    new_args->socket_fd = con_sockfd;
    pthread_create(&thread, NULL, process_job, (void*) new_job);
  }
}
```

Code Continues on Next Page...

```
void *process_job(void *arg) {
  struct Arg_struct *new_args = (struct Arg_struct*) arg;

  /* Do the work by calling do_work() and send back response */

  ___________________________________________________

  ___________________________________________________

  ___________________________________________________



  /* Free resources */

  ___________________________________________________

  ___________________________________________________

}

/* Code Segment in main() */
/* Assume bind() and listen() have been called */

while (1) {
  /* When a new client connects */

  con_sockfd = _________(lstn_sockfd, NULL, NULL);
  pthread_create(&thread, NULL, new_conn, (void*) con_sockfd);
}

close(lstn_sockfd);
```

After you are done writing the program, you realize that it doesn't work. Clients are receiving gibberish when they try to read each struct that is sent back. You show it to master systems programmer Jeff Dean, who sees two problems in the code that cause the program to fail.

Firstly, the assumptions that `read` and `write` will always return `count` bytes do not hold. In fact, these sys calls often read/write fewer than `count` bytes before returning.

Problem 5d[2pts]: Provide a reason why a `write` system call might write fewer than count bytes.

Problem 5e[2pts]: To solve this problem, Bob proposes writing a new function with the signature:

`write_bob(int fd, const void* buf, size_t count)`

`write_bob` makes use of `write` syscall in its implementation and makes sure that it always writes `count` bytes before returns. Provide an implementation for `write_bob`:

Hint: If `write` returns 3, then 3 bytes are already written. You don't have to write them again.

```
void write_bob(int fd, const void *buf, size_t count) {
  char *buffer = (char*) buf;
  /* Your code below */







}
```

Problem 5f[3pts]: Jeff Dean points out a new problem with the code, because the threads share the same connection socket. Bob proposes solving this problem with a mutex lock. List between which of the lines above you need to add `lock_acquire()` and `lock_release()` with a global mutex to solve Jeff's problem. For instance, if you needed a `lock_acquire()` between lines 1 and 2, you would write (1, 2) under `lock_acquire()` below. *You may not need all six spaces.*

`lock_acquire()`

(__________, __________) (__________, __________) (__________, __________)

`lock_release()`

(__________, __________) (__________, __________) (__________, __________)

Problem 5g[3pts]: Bob successfully builds the server above, but his computer is slow, and can only handle up to 5 jobs being processed concurrently per connection. Jeff proposes a solution where a semaphore would block new jobs from being processed if 5 other jobs are already being processed, and only allow new jobs to start once an existing job is finished. Assume each thread has a semaphore initialized to 5 and that the line numbers are as originally (ignore the lock operations above). List between which lines you need to add `sema_up()` and `sema_down()` to allow only up to 5 jobs to be concurrently processed per connection.

`sema_down()`

(__________, __________) (__________, __________) (__________, __________)

`sema_up()`

(__________, __________) (__________, __________) (__________, __________)

[Scratch Page: Do not put answers here!]

[Scratch Page: Do not put answers here!]