CS 3214 Spring 2025 Midterm

March 31, 2025

- This is a closed-book, closed-internet, closed-cell/smart phone or watch and closed-computer exam. However, you may refer to your sheet of prepared notes.
- Your exam should have 10 pages with 8 questions totaling 100 points. You have 75 minutes. Please write your answers in the space provided on the exam paper.
- If you finish the exam early you are expected to leave the room very quietly. If you finish within 15 minutes or less before the end of the allotted time, please stay in your seat until the end.
- Answers will be graded on correctness and clarity. The space in which to write answers to the questions is kept purposefully tight, requiring you to be concise. You may lose points if your solution is more complicated than necessary or if you provide extraneous information along with a correct solution.

Name (printed) ______________________ PID ______________________`@vt.edu`

Section: ☐ Dr. Butt ☐ Dr. Williams

I accept the letter and the spirit of the Virginia Tech undergraduate honor code — I have not given or received aid on this exam.

(signed) ______________________________

You are expected to keep the content of this exam secret until told otherwise by your instructor. Please do not start until instructed to do so.

#	Problem	Score
1	Doubtful!	11
2	Guarded!	7
3	Shelled!	19
4	Forked!	13
5	That's BSS!	17
6	Threaded!	17
7	Sema Down!	13
8	A Choice was made!	3
	Total	100

1 Doubtful!

1. Two process can communicate with each other via a pipe. (1 pt)

   ☐ true / ☐ false True

2. Linking order for libraries does not matter for the linker to accurately link a binary. (1 pt)

   ☐ true / ☐ false False

3. The use of static libraries results in smaller binaries than dynamic libraries. (1 pt)

   ☐ true / ☐ false False

4. A system call is the mechanism used for mathematical routines, such as `sin`, `pow`, etc. (1 pt)

   ☐ true / ☐ false False

5. The maximum number of processes allowed in a system is limited by the number of CPUs on the system. (1 pt)

   ☐ true / ☐ false False

6. All system calls are blocking, meaning the process calling the system call will enter the BLOCKED state. (1 pt)

   ☐ true / ☐ false False

7. Threads share an address space and a stack, but have independent register state. (1 pt)

   ☐ true / ☐ false False

8. If a producer process writes to a pipe faster than the consumer process reads from a pipe, some data from the pipe will be lost. (1 pt)

   ☐ true / ☐ false False

9. A process that has called `sleep()` is most likely in the READY state. (1 pt)

   ☐ true / ☐ false False

10. A call to `pthread_cond_wait()` always returns immediately. (1 pt)

    ☐ true / ☐ false False

11. A process scheduler transitions process state from READY to RUNNING and vice versa. (1 pt)

    ☐ true / ☐ false True

2 Guarded!

A CS3214 student is writing an OS for some new embedded device hardware consisting of a CPU and a storage device. Unfortunately the CPU does not have an implementation of a supervisor/user mode. The storage device is accessible via a special instruction `out` that writes a specified byte to a specified offset on the storage device.

In order to ensure that multiple user programs can run without interfering with one another, the student has implemented ”system calls” to manage read/write to the storage device via permissioned files in a filesystem. In other words, the filesystem checks that a user has permission to write to the offset on the device corresponding to where the file is stored before performing the write to the storage device via the out instruction.

1. Fill in the blank: this system lacks ______ mode operation? (2 pts)

   Dual mode operation

2. What are the dangers of running a user program that directly uses the `out` instruction? (5 pts)

   The user may bypass the filesystem checks and/or corrupt stored data

3 Shelled!

Two CS3214 students, Alice and Bob, wrote a shell for project 1. Alice implemented kill as a builtin, whereas Bob did not. When running either Alice or Bob's shell, assume an identical filesystem that contains standard UNIX utilties, including `/usr/bin/kill`, and that the `PATH` contains `/usr/bin` (but nothing else).

1. Fill in the blank: (1 pt)
   PATH is an example of an Environment variable.

2. Recall the `which` utility in UNIX. According to the manpage:

   > `which` returns the pathnames of the files (or links) which would be executed in the current environment, had its arguments been given as commands in a strictly POSIX-conformant shell. It does this by searching the PATH for executable files matching the names of the arguments.

   (a) What is the result of running `which kill` in Alice's shell? (2 pts)

   /usr/bin/kill

   (b) What is the result of running `which kill` in Bob's shell? (2 pts)

   /usr/bin/kill

3. Suppose a program *foo* is running in the background, but stuck in an infinite loop (e.g., `for(;;);`).

   (a) Of the three process states we learned about in class, what state(s) would you expect to see *foo* in? (2 pts)

   Running or Ready

   (b) Describe the actions taken by Alice's shell when executing `kill <pid>` where `<pid>` corresponds to a process *foo* stuck in an infinite loop (e.g., `for(;;);`). (3 pts)

The shell builtin calls the kill system call to terminate the process

(c) Describe the actions taken by Bob's shell when executing `kill <pid>` where `<pid>` corresponds to a process foo stuck in an infinite loop. (3 pts)

The shell forks and execs the binary at /usr/bin/kill which issues the kill system call to terminate the process

4. Now suppose the system is in an overloaded state where the system prevents any new process creation from the user (e.g., fork fails) to due to resource exhaustion.

   (a) What happens when Alice tries to kill a process with `kill <pid>`? (3 pts)

   The shell builtin calls the kill system call to terminate the process

   (b) What happens when Bob tries to kill a process with `kill <pid>`? (3 pts)

   The shell fails to fork and exec the kill binary so Bob cant kill the process

4 Forked!

How many times do each of the following programs output "Hi CS3214!"?

a)
```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
    printf("Hi CS3214! \n ");
    fork();

    char *argv_for_echo[] = {"echo", "Hi CS3214!"};
    execvp("echo", argv_for_echo);
}
```

(3 pts)

3

b)
```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
    fork();
    for (int i=0; i<10; i++) {
        printf("Hi CS3214! \n");
        char *argv_for_echo[] = {"echo", "Hi CS3214!"};
        execvp("echo", argv_for_echo);
    }
}
```

(3 pts)

4

---

c)
```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
    fork();
    fork();
    printf("Hi CS3214! \n");

    char *argv_for_echo[] = {"echo", "Hi CS3214!"};
    execvp("echo", argv_for_echo);
    execvp("echo", argv_for_echo);
}
```

(3 pts)

8

---

d)
```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
    char *argv_for_echo[] = {"echo", "Hi CS3214!"};

    fork();
    execvp("echo", argv_for_echo);

    fork();
    execvp("echo", argv_for_echo);
}
```

(4 pts)

2

---

5 That's BSS!

Two CS3214 students, Alice and Bob, were working on a project outside of class which involved computation on a large array of bytes initially all set to `0xff`. They got into an argument over the very first lines in the program. Here is how their programs start:

```
// Alice's program
#define LARGE 10485760 /* 10 MB */
char my_array[LARGE];

int main(int argc, char **argv) {
    for (int i = 0; i < LARGE; i++)
        my_array[i] = 0xff;
    /* program continues ... */
}
```

```
// Bob's program

#define LARGE 10485760 /* 10 MB */
char my_array[LARGE] = {0xff};

int main(int argc, char **argv) {
    /* program continues ... */
}
```

1. Assuming the remainders of the programs are identical, which program do you think will run faster? Why? (3 pts)

   Bob's because it avoids the initialization loop

2. Assuming the remainders of the programs are identical, after compiling and linking, which executable will take up more space in the filesystem (or specify that they will take up approximately the same space on the filesystem). Why? (3 pts)

   Bob's will use around 10MB more since the array must be included in .data section

3. Assuming the remainders of the programs are identical, which program will use more memory when running (or specify that they will use the same amount of memory). Why? (3 pts)

   Same amount, both have the 10 mb array in memory while executing

4. After compiling and linking Alice's program and running `nm` on the resulting binary, which section will `my_array` show up in? (3 pts)

   .bss
   0000000000004040 B my_array

5. After compiling and linking Bob's program and running `nm` on the resulting binary, which section will `my_array` show up in? (3 pts)

   .data
   0000000000004020 D my_array

6. Suppose Bob adds the keyword `const` to the definition of `my_array` so the line now reads `const char my_array[LARGE] = 0xff;`. Which section will `my_array` show up in now? (2 pts)

   .rodata
   0000000000002020 R my_array

6 Threaded!

A CS3214 student missed a few classes and thinks that processes and threads are "basically two words for the same thing, dude". The student was instructed to write a program that uses two threads, each of which increments a global shared counter 1000000 times. At the end, the main thread should print the final value of the counter (which should be 2000000). The student wrote the following code:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int global_shared;

static void thread1()
{
    printf("thread1 working!\n");
    for (int i = 0; i < 1000000; i++) {
        global_shared++;
    }
    printf("thread1 done!\n");
    exit(0);
}

static void thread2()
{
    printf("thread2 working!\n");
    for (int i = 0; i < 1000000; i++) {
        global_shared++;
    }
    printf("thread2 done!\n");
    exit(0);
}

int main()
{
    /* start two children */
    if (!fork())
        thread1(); /* child */
    if (!fork())
        thread2(); /* child */
    /* wait for them to finish */
    wait(NULL);
    wait(NULL);

    printf("global_shared = %d\n", global_shared);
    return 0;
}
```

(a) Does this program use multiple processes or multiple threads in a single process? (3 pts)

Multiple processes

(b) What possible values will be output for `global_shared`? Why? (5 pts)

Only 0. The global shared variable is actually private not shared because fork made a new process

(c) Suppose the student introduced a mutex and calls to `pthread_mutex_lock()` and `pthread_mutex_unlock()` to protect all accesses to `global_shared`. Would this fix the program? Why? (5 pts)

No, in that case the mutex also would not be shared because the processes have their own copies of the address space and copies of the mutex

(d) Without changing the use of `fork()`, what is a way to ensure that each thread's work is accurately communicated and reflected in `global_shared()`? (4 pts)

One method is to use pipes for IPC, other IPC mechanisms also work

7 Sema Down!

Consider the following program that we discussed in the class. One thread flips a coin, and shares the result with the other. For this question, you can assume that the *printf* command runs atomically.

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

int coin_flip;

static void * thread1(void *_) {
    coin_flip = rand() % 2;
    printf("T1: %d ", coin_flip);
    return NULL;
}

static void * thread2(void *_) {
    printf("T2: %d ", coin_flip);
    return NULL;
}

int main() {
    const int N = 2;
    pthread_t t[N];
    srand(getpid());
    pthread_create(&t[0], NULL, thread1, NULL);
    pthread_create(&t[1], NULL, thread2, NULL);

    for (int i = 0; i < N; i++)
        pthread_join(t[i], NULL);
    printf("\n");
    return 0;
}
```

(a) What is the output when the program is run? (3 pts)

T1: 0 T2: 0 - Thread 1 runs first, sets coin_flip to 0, then Thread 2 reads and prints the same value
T1: 1 T2: 1 - Thread 1 runs first, sets coin_flip to 1, then Thread 2 reads and prints the same value
T2: 0 T1: 0 - Thread 2 runs first (reads uninitialized coin_flip which is 0), then Thread 1 randomly sets and prints 0
T2: 0 T1: 1 - Thread 2 runs first (reads uninitialized coin_flip which is 0), then Thread 1 randomly sets and prints 1

(b) Is the program deterministic? In a short sentence explain why? (3 pts)

No the threads can run in any order, providing different output in a non deterministic manner

(c) In a class demo, we ran this program 100 times and received the same output everytime. Does this mean the program always produces correct output? (4 pts)

Nope, just because we observed correct output x number of times doesn't mean an incorrect output can't be observed if the program is non deterministic

(d) The program is modified such that *thread*2 is created before *thread*1. Will this ensure correct

Nope, thread synchronization will

execution? (3 pts)

---

8 A Choice was made!

Recall the definition of the fuctions:
int dup(int oldfd)
int dup2(int oldfd, int newfd)

Consider the following code:

```
int main() {
   int fd[2];
   for (int c = 0; c < 2; c++) {
      fd[i] = dup(0);
   }

   dup2(1, fd[0]);
   dup2(fd[0], fd[1])
   dup2(2, fd[0]);

   write(fd[1], "CS3214\n" , 7);
}
```

Where is *CS3214* printed when the following program is run?

Circle one of the following choices: (3 pts) STDOUT

A STDIN

B STDOUT

C STDERR

D Not printed