Computer Science 392
Midterm 2

April 13, 2022
Stevens Institute of Technology

Name	
Student ID	
TA (the person who graded your home-works)	
Section time	

This is a closed-book exam with one 2-sided handwritten pages of notes permitted. It is intended to be a 50 minute exam. You have 50 minutes to complete it. Write all of your answers directly on this exam. Make your answers as concise as possible. If there is something in a question that you believe is open to interpretation, please raise your hand to request clarification. Put your name on every page and check that you have them all. **Any page without your name will not be graded.**

By my signature below, I swear that this exam is my own work. I have not obtained answers or partial answers from anyone, and I have not discussed it with anyone who took the early exam. I promise not to discuss this exam with anyone prior to completion of the exam. I promise not to upload this midterm to anywhere on the Internet.

Signature

A. True or False (30 pts)

For the following questions, select whether they are true or false and explain in two sentences or less. Explanations longer than two sentences or lack of explanation will receive no credit.

(6 pts) A1. After a call to `fork()`, `stdin`, `stdout`, and `stderr` are reset to their default states in the child process.
❏ True ❏ False
Explain:

(6 pts) A2. When a user program successfully calls `execv()`, its process is replaced with a new program. Everything about the old process is destroyed, including all of the CPU registers, program counter, stack, heap, background threads, file descriptors, and virtual address space.
❏ True ❏ False
Explain:

(6 pts) A3. Let n be the size of the virtual address space. On a `fork()` call, the OS does $O(n)$ work to duplicate the parent's address space for the child process.
❏ True ❏ False
Explain:

(6 pts) A4. When you type `Ctrl+C` to a running program in your terminal, you are actually sending a `SIGQUIT` signal to the program, which makes it quit.

❏ True ❏ False

Explain:

(6 pts) A5. Calling `pipe()` on an array like `int fds[2];` creates a read and write stream that local processes can use for unidirectional IPC. Under the hood, a pipe is implemented as a buffer in user space where the read end sits at the 0th index of the pipe and the write end sits at the 1st index of the pipe.

❏ True ❏ False

Explain:

B. Multiple Choices (20 pts)

In the following multiple choice questions, please select all options that apply. Answering a question instead of leaving it blank will NOT lower your score (the minimum score for a single question is 0, not negative).

(5 pts) B1. Which of the following that are true regarding I/O?

- ❏ High level I/O deals with FILE* struct.
- ❏ There is one file descriptor table per system.
- ❏ Low-level I/O is buffered in user space.
- ❏ None of the above.

(5 pts) B2. Which of the following statements about files are true?

- ❏ The same file descriptor number can correspond to different files for different processes.
- ❏ Reserved 0, 1, and 2 (`stdin`, `stdout`, `stderr`) file descriptors cannot be overwritten by a user program.
- ❏ File descriptions keep track of the file offset (file position).
- ❏ An `lseek()` within one process may be able to affect the writing position for another process.

(5 pts) B3. Assume all calls to open(), `read()`, `write()`, and `fork()` succeed. Suppose that `montymole.txt` was empty before running this block of code. The following code was run:

```
int main(int argc, char** argv) {
    int fd1 = open("montymole.txt", O_WRONLY);
    int fd2 = open("montymole.txt", O_WRONLY);
    write(fd1, "mole", 4);
    write(fd2, "whack", 5);
    write(fd2, "mole", 4);
    write(fd1, "mole", 4);
    write(fd1, "mole", 4);
    close(fd1);
    close(fd2);
 }
```

Which of the following could be the content of "`montymole.txt`"?

- ❏ `whacmolemole`
- ❏ `whacmolek`
- ❏ `molewhackmolemolemole`
- ❏ `whackmolemole`
- ❏ Nothing

(5 pts) B4. How many new processes could be created?

```
#include <sys/types.h>
#include <unistd.h>
int main() {
      for (int i = 0; i < 2; i++) fork();
      return 0;
}
```

- ❏ 0
- ❏ 1
- ❏ 2
- ❏ 3
- ❏ 4

C. Short Answers (50 pts)

C1. Below is a simple Linux C program which redirects `stdin` so that executed process greps the word `"cs392"` from the file `"cs392.txt"`. Assume:

- Calls to `open()`, `close()`, and `execvp()` succeed;
- All the necessary header files are included;
- `stdin` has a file descriptor of 0.

```
int main(int argc, char** argv) {
      char* grep_args[] = {"grep", "cs392", NULL};
      int in = open("cs392.txt", O_RDONLY);

      (Do not write your answer here)

      close(in);   /* close unused file descriptors */
      execvp("grep", grep_args);    /* execute grep */
}
```

(a) **(5 pts)** Fill in the blanks to complete the code. Please provide two solutions *(that are both theoretically correct)*, in which one solution can have only one line, while the other can have as many lines as you want.

Solution 1 (only one line allowed)	Solution 2 (as many lines as you like)

(b) **(10 pts)** Based on the two solutions you provided above, compare them. Which one do you prefer? Why?

C2. In the following program, we want to print out, "Process 392 says 42" once. Assume that the process ID of the child process is 392, the fork() is successful, and we want the behavior to be predictable. Do not add extra lines or try to compact your code onto the lines. **No hard-coding/assignment of values** is allowed for your blanks inside of `main()`.
Fill in the blanks below, to output "Process 392 says 42"

```
void helper(void) {
    exit(42);
}
int main(void) {
      int kirito = 0;
      pid_t pid = fork():

      if ( pid ) {[(a) (Do not write your answer here)]}
      else helper ();

      printf ("Process %d says %d\n", pid,[(b) (Do not write your answer here)] );
}
```

(5 pts) (a)

(5 pts) (b)

C3. Harry has a big test coming up at LogPorts Castle, and he hasn't read the textbook yet! He needs to be able to perform a specific spell, but he doesn't know where it is in the textbook.

Harry has asked you, a computer magic major at LogPorts, to write a multiprocessing program for him that can rapidly find the spell in the textbook. **Your solution must create `num_processes` processes, and each process should search an equal portion of the textbook using the `search_for_spell()` function.**

You can assume that `num_processes` is a power of 2, and that `textbook_size` is a multiple of `num_processes`.

NOTE: For each blank, you can use as many lines as you need to solve the problem. You may be able to leave some blanks empty. You can assume that all syscalls will not error.

```
/* This function will read the textbook file into array `arr`.
    You can assume that the textbook will be exactly `textbook_size`
    characters long. Assume that this is already implemented. */

void read_textbook_into_array(char* arr, int textbook_size);
```

```
/* This function will search for Harry's spell in array `arr`,
    and print the spell if found.
    The function will only search for the spell in the subarray between
    `start_index` (inclusive) and `end_index` (exclusive).
    Assume that this is already implemented. */

void search_for_spell(char* arr, int start_index, int end_index);
```

```
void process_array(int num_processes, int textbook_size) {
    char* arr = malloc(sizeof(char) * textbook_size);
    read_textbook_into_array(arr, textbook_size);

    int start_index = 0;
    int end_index   = textbook_size;

    for (_____(a)_____) {
        pid_t pid = fork();
        if (pid == 0) {_____(b)_____}
        else {_____(c)_____}
    }
    search_for_spell(arr, start_index, end_index);
}
```

(5 pts) (a)

(5 pts) (b)

(5 pts) (c)

(10 pts) (d) Why is this program not memory-efficient with respect to the size of the textbook?

Reference Sheet

Processes

```
pid_t fork(void);
pid_t wait(int* status);
pid_t waitpid(pid_t pid, int* status, int options);
int   execvp(const char* path, char* const argv[]);
void  exit(int status);
```

Low-Level File I/O

```
int     open(const char* pathname, int flags); (O_APPEND|O_CREAT|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);
```

Congratulations on reaching the end of the exam!

There is no more exam material from here.

Remember that no matter how you do, someone cares for you.

We hope you enjoy systems programming so far. 🥰