CS 110 Midterm Fall 2019

Name: __KEY_____ SUNet: ___lelands_____

Problem 1:

6 points

This question has two parts. The first part has five answers to fill in, and the second part is a short answer.
The following program plays an arithmetic game with a player. The player has to answer multiplication questions before time runs out. Every time the player gets an answer correct, five seconds are added to the timer.

```
int time_left;

void updateTime(int sig) {
    time_left--;
    printf("Time left: %d seconds\n", time_left);
    if (time_left == 0) {
        printf("\nYour time is up!\n");
        _____________________________ // missing line 1
        exit(0);
    }

}

void addFive(int sig) {
    time_left += 5;
}

void quit(int sig) {
    waitpid(-1, NULL, 0);
    exit(0);
}

void setupAlarm(int t) {
    struct itimerval timer;
    timer.it_value.tv_sec = t; // t second timer
    timer.it_value.tv_usec = 0;

    timer.it_interval.tv_sec = t; // repeat every t seconds
    timer.it_interval.tv_usec = 0;

    setitimer (ITIMER_REAL, &timer, NULL);
}

int main(int argc, char **argv) {
    srand(time(NULL));
    if (argc != 2) {
        printf("Usage:\n\t%s total_time\n", argv[0]);
        return -1;
    }
    printf("Welcome to the arithmetic challenge. Each time you\n");
    printf("answer a question, you get 5 more seconds.\n");
    printf("Press <enter> to begin.");
    char buffer[1024];
    scanf("%c", buffer);
```

```
    pid_t pid = fork();
    if (pid == 0) {
        _____________________________ // missing line 2
        _____________________________ // missing line 3
        time_left = atoi(argv[1]);
        setupAlarm(1); // 1 second timer
        while (true) {
            pause();
        }
    }

    _____________________________ // missing line 4
    while (true) {
        int num1 = random() % 15 + 1;
        int num2 = random() % 15 + 1;
        printf("What is %d * %d? ", num1, num2);
        int answer;
        scanf(" %d", &answer);
        if (answer == num1 * num2) {
            printf("Correct! +5 seconds\n");
            _____________________________ // missing line 5
        } else {
            printf("Incorrect!\n");
        }
    }
}
```

You will notice that there are five lines missing from the program. They correspond to the following lines, but not in the following order:

```
kill(pid, SIGUSR1);
kill(getppid(), SIGUSR1);
signal(SIGUSR1, quit);
signal(SIGUSR1, addFive);
signal(SIGALRM, updateTime);
```

Your job is to analyze the code, and then put the five lines into their correct locations.

Note: the ***`setupAlarm(t)`*** *function creates a timer that sends a* ***`SIGALRM`*** *after each interval* **`t`** *seconds.*

a) Order the above lines in the order they should appear in the code, and fill in the line number next to each. [1 point each]

```
kill(pid, SIGUSR1); // line 5
kill(getppid(), SIGUSR1); // line 1
signal(SIGUSR1, quit); // line 4
signal(SIGUSR1, addFive); // line 2 or 3
signal(SIGALRM, updateTime); // line 2 or 3
```

b) There is a minor race condition in the code -- where is it, and why is it a race conditiion? [1 point]

Multiple answers. For example: the time_left check in updateTime() could be checked and found to be 0, but addFive() function has been signaled but not yet run. The player arguably should have received more time but the program exits.

Problem 2: File System Abstraction

10 points

Describe the three layers of abstraction in Linux/POSIX file semantics, and what each abstraction represents. At each layer that has a reference count, specify what it counts. Give 2 additional examples of pieces of state (metadata/data) that are kept at each layer, if any.

The first layer is the file descriptor. It represents a handle to an open file. It is a pointer to an entry in the file table and does not have any additional state. (If a student happens to know a nitty-gritty detail of how the table is implemented and names a piece of table state, that's fine. But saying there is no extra state is good enough.)

The second layer is the open file table. This layer has a reference count, which keeps track of how many file descriptors point to it. It keeps state such as:

- whether the file was opened read-only/write-only/or read-write
- whether the file was truncated
- the current seek pointer (position in the file)
- the file system of the file
- the directory entry of the file

The third layer is the vnode table. If the student says inode that's OK. This layer has a reference count, which keeps track of how many open file entries there are for the file. It keeps state such as:

- The current size of the file
- The last time the file was accessed
- The permissions of the file
- How many hard links there are to the file
- The user and group of the file

The key thing here is that the student understands which kind of state is where: what state is associated with an open file handle, versus with the file itself.

Problem 3: Who Needs sigsuspend?

10 points

This problem has 3 parts.
Answer the following questions.

a) Define atomicity. [2 points]

Atomicity is the property that, to an external observer, an operation has either completed or it hasn't started: an external observer cannot see it in an intermediate state. It's also OK if someone says that atomic operations have the property that they can be defined with a linear ordering. Any description which is a technical definition from databases, programming languages or operating systems is OK. What's important is they understand what it is, not that they've memorized a particular definition.

b) Show an example piece of code using a signal handler, **`sigprocmask()`** and **`pause()`** that has an inherent race condition which cannot be solved with **`sigprocmask()`** and **`pause`** alone. [3 points]

```
void handler(void) {
  // Does something
}

sigprocmask(SIG_BLOCK, &mask_with_alarm_set, NULL);
signal(SIGALRM, handler);
ualarm(100, 0);
sigprocmask(SIG_UNBLOCK, &mask_with_alarm_set, NULL);
pause();
```

c) Write 2-3 **succinct** sentences describing the race condition in your code from part (b). Explain how using **`sigsuspend()`** correctly can prevent the race condition. [5 points]

The race condition is if the signal fires before pause() is called. If this happens, and there are no other signals, then pause will wait forever, as no signal will arrive.

The way to solve this problem is to use sigsuspend to atomically pause and unblock SIGALRM.

For b and c, anything that is a real race condition is fine. This is just one example. The key thing is sigsuspend/pause. If they include a data race as well, that's OK, but the question is about sigprocmask and pause.

Problem 4: Signal Puzzle

5 points

This problem has 1 part, with five possible/not possible questions.

Consider the following C program and its execution. Assume all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations. *Note: There is one **`getppid()`** (get parent pid) and one **`getpid()`** (get current process pid) in the program.*

```
// globals
int a = 1;
int b = 2;
int c = 3;

void signalHandler(int signal) {
    printf("a: %d, b: %d\n", a, b);
    waitpid(-1, NULL, 0);
    if (c > b) {
        int rnd = random() % 2;
        if (rnd && getpid() == 0) {
            printf("child\n");
        }
        exit(0);
    }
}

int main(int argc, char *argv[]) {
    srandom(time(NULL)); // seed the random number generator
    signal(SIGUSR1, signalHandler);
    pid_t pid = fork();
    if (pid == 0) { // child
        kill(getppid(), SIGUSR1);
        a = 10;
        raise(SIGUSR1);
        return 0;
    }
    c = 0;
    b = 100;
    raise(SIGUSR1);
    printf("c: %d\n", c);
    return 0;
}
```

For the given outputs below, answer "possible" or "not possible" for each.

	Output

A)	`a: 10, b: 2`
	`a: 1, b: 2`
	possible (possible or not possible?)
	how: parent off cpu but signaled by child; child signal handler prints; then parent signal handler prints; parent exits

B)	`a: 1, b: 100`
	`a: 10, b: 100`
	`a: 1, b: 100`
	`c: 0`
	not possible (possible or not possible?)
	why: child's b is never 100 (processes have independent virtual memory)

C)	`a: 1, b: 100`
	`a: 10, b: 2`
	`a: 1, b: 100`
	`c: 0`
	possible (possible or not possible?)
	how: child off cpu until parent gets to raise; parent signal handler prints and parent waits for child; child signals parent, and prints; parent's signal handler (first time) ends; parent signal handler (second time) prints and ends; parent prints in main.

D)	`a: 1, b: 100`
	`a: 10, b: 2`
	`c: 0`
	Will accept either answer (possible or not possible?)
	why (argument: not possible): if parent prints 100 once, parent will print 100 again, and will print 0 (never exits in handler
	why (argument: possible): if the parent is signaled at exactly the same time by both child and parent, only one signal will be sent

E)	`a: 1, b: 2`
	`a: 10, b: 2`
	`child`
	not possible (possible or not possible?)
	why: getpid() for child is never 0, so "child" cannot print.

Problem 5: Autograder

10 points

This question has one part.

For this problem, you will write a **C** program that models a basic autograder, in that it runs a program (e.g., a student's submission) with some input, and prints the output, formatted with line numbers. The autograder takes two filenames on the command line, as follows:

```
$ ./autograder input.txt ./program-to-run
```

Your program will run **program-to-run** with the contents of **input.txt** as its standard input.

Your program will then read the standard output from **program-to-run** and will prepend each line with a line number, a vertical bar, and a space. For example, assume that **input.txt** held the following two lines:

```
4
5
```

And, assume that **program-to-run** output the following five lines when fed the two lines above as standard input:

```
20
0.8

9
-1
```

Your program should output the following:

```
1| 20
2| 0.8
3|
4| 9
5| -1
```

Please add comments so we know what you are attempting to do. You must use the **read()** function to read the standard output from **program-to-run**. You cannot assume anything about the number of bytes of output that are produced.

Write the code on the following page.

```
#define _GNU_SOURCE
#include<stdio.h>
#include<stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <wait.h>
#include <stdbool.h>

int main(int argc, char **argv)
{
    if (argc != 3) {
        printf("Usage:\n\t%s input.txt program-to-test\n", argv[0]);
        return -1;
    }
```

Grading note: lots of students made two pipes and read in the input file only to pipe it back out to the autograder. We didn't take points off for that method (it does work if done correctly), but please take a look at our solution, which is much simpler and only needs a single pipe.

```
    // create a pipe for gathering stdout from program-to-test
    int fds[2];
    pipe2(fds);

    // fork / execvp
    pid_t pid = fork();
    if (pid == 0) {
        close(fds[0]);
        // open input file for reading
        int fd_input_file = open(argv[1], O_RDONLY);
        // dup2 fd_input_file to stdin for child
        dup2(fd_input_file, STDIN_FILENO);

        // dup2 fds[1] as stdout for child
        dup2(fds[1], STDOUT_FILENO);
        close(fds[1]);
        execvp(argv[2], argv + 2);
        // should not get here
        printf("Uh-oh. Exiting child.\n");
        exit(0);
    }

    // parent
    close(fds[1]);

    // get output from pipe
    // read one character at a time and put line numbers
    // based on newlines
    int line_no = 1;
    int start_of_line = true;
    char next_ch;
    while (read(fds[0], &next_ch, 1) != 0) {
        if (start_of_line) {
            printf("%d| ", line_no);
            start_of_line = false;
        }
        if (next_ch == '\n') {
            start_of_line = true;
            line_no++;
        }
        printf("%c", next_ch);
    }

    close(fds[0]);
    waitpid(pid, NULL, 0); // wait for child to end
    return 0;
}
```

```
//////////////////
// Code without comments:

    int fds[2];
    pipe2(fds);

    pid_t pid = fork();
    if (pid == 0) {
        close(fds[0]);

        int fd_input_file = open(argv[1], O_RDONLY);
        dup2(fd_input_file, STDIN_FILENO);

        dup2(fds[1], STDOUT_FILENO);
        close(fds[1]);
        execvp(argv[2], argv + 2);
        printf("Uh-oh. Exiting child.\n");
        exit(0);
    }

    close(fds[1]);

    int line_no = 1;
    int start_of_line = true;
    char next_ch;
    while (read(fds[0], &next_ch, 1) != 0) {
        if (start_of_line) {
            printf("%d| ", line_no);
            start_of_line = false;
        }
        if (next_ch == '\n') {
            start_of_line = true;
            line_no++;
        }
        printf("%c", next_ch);
    }

    close(fds[0]);
    waitpid(pid, NULL, 0);
    return 0;
}
```