CS 110 Practice Midterm 4

This is a closed book, closed note, closed computer exam (although you can use your single double-sided cheat sheet). You have 80 minutes to complete all problems. You don't need to **`#include`** any header files, and you needn't guard against any errors unless specifically instructed to do so. Understand that most points are awarded for concepts taught in CS110.

Good luck!

SUNet ID (username):	____________**`@stanford.edu`**
Last Name:	______________________
First Name:	______________________

I accept the letter and spirit of the honor code.

[signed] ____

		Score	Grader
1. Game Simulation	[10]	______	______
2. Parallel **`grep`**	[10]	______	______
3. Short Answer Questions	[10]	______	______
Total	**[30]**	______	______

Relevant Prototypes

```
// filesystem access
int open(const char *path, int oflag, ...);       // returns descriptor
ssize_t read(int fd, char buffer[], size_t len);  // returns num read, 0 at eof
ssize_t write(int fd, char buffer[], size_t len); // returns num written
int close(int fd); // ignore retval
int pipe(int fds[]); // argument should be array of length 2, ignore retval
int pipe2(int fds[], int flags); // common flag: O_CLOEXEC
int dup2(int old, int new); // ignore retval
#define STDIN_FILENO 0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2

// exceptional control flow and multiprocessing
pid_t fork();
pid_t waitpid(pid_t pid, int *status, int flags);
typedef void (*sighandler_t)(int sig);
sighandler_t signal(int signum, sighandler_t handler); // ignore retval
int sigsuspend(const sigset_t *mask); // ignore retval
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset); // ign. retval
int execvp(const char *path, char *argv[]); // ignore retval
int kill(pid_t pid, int sig); // ignore retval
int setpgid(pid_t pid, pid_t pgid); // ignore retval
#define WIFEXITED(status)   // macro
#define WIFSTOPPED(status)  // macro
#define WEXITSTATUS(status) // macro
```

Problem 1: Game Simulation [10 points]

There are artificial intelligence simulators for many games, including tic-tac-toe, checkers, chess, go, etc. Often, an AI improves by playing games against other simulators (or even against itself). For this problem, you will write a program, `game_sim`, that runs two game-playing programs and pits them against one another. Your simulator will pass `STDOUT` from the first program to `STDIN` of the second program, and vice-versa until both programs terminate. In other words, the first program will produce a move and print it to its `STDOUT`, and the second program will expect its `STDIN` to be the first move. Then the second program will print out its move to `STDOUT`, and the first program will expect it. This continues until the game terminates. The first game will be responsible for making the first move and will also be responsible for printing the output of the game to the terminal (to `STDERR`, which you can ignore). The way you will run the simulation will be as follows:

```
./game_sim ./game_engine1 ./game_engine2
```

In your program, you will set up the appropriate pipes, and launch the first program (using `execvp`) as player 1 with the following command line (for this example):

```
./game_engine1
```

The second program should be launched with the following command line:

```
./game_engine2 p2
```

where "`p2`" tells the program it will be player 2 and will expect the first input from player 1.

The output of the game play will be sent by player 1 to `STDERR`, and you do not need to worry about this output. Please add your code under the "`your code here`" section.

```
// file: game-sim.c
#include<stdio.h>
#include<unistd.h> // fork, getpid, getppid
#include <sys/wait.h> // waitpid
#include <fcntl.h> // O_CLOEXEC
#include <unistd.h> // pipe, pipe2

int main(int argc, char **argv)
{
    if (argc < 3) {
        printf("Usage:\n\t %s ./game_engine1 ./game_engine2\n",argv[0]);
        return 0;
    }
    char *game_engine1 = argv[1];
    char *game_engine2 = argv[2];

    // your code here:











    return 0;
}
```

Problem 2: Parallel `grep` [10 points]

The standard UNIX `grep` program searches lines of `STDIN` for matching substrings, and prints out matching lines, in order. For example, the following pipeline would search for the substring **`"cs110"`** in the **`printf`** string, and it would print out three lines:

```
printf "I love CS110\nCS110 is my favorite\nI like CS107 more\nI have no comment
about CS110.\n" | grep CS110
```

Output:

```
I love CS110
CS110 is my favorite
I have no comment about CS 110.
```

The interesting part about grep is that it does not, by default, search across line boundaries, meaning that each line search is independent of the others. We can therefore parallelize it using multiprocessing.

Modify the partially written program on the next page that performs the search in parallel. You need to write code at the end of the **`printStoppedProcesses`** function, and you also need to write the **`childHandler`** function. Be sure to read through the entire starter code to understand the program flow.

```
const int kMaxProcesses = 80;
int childrenStopped = 0; // global

void printStoppedProcesses(vector<pid_t> &children) {
   // wait for all returning jobs, in order, and
   // continue each job to print out its line
   sigset_t suspendset, childset;
   sigemptyset(&suspendset);
   sigemptyset(&childset);

   sigaddset(&childset,SIGCHLD);
   // block SIGCHLD for while loop
   sigprocmask(SIG_BLOCK, &childset, NULL);

   while (children.size() > childrenStopped) {
      sigsuspend(&suspendset);
   }
   sigprocmask(SIG_UNBLOCK, &childset, NULL);
   // your code below
```

```
}
```

```
void childHandler(int sig) {
   // update childrenStopped each time a child stops or exits
   // your code below
```

```
}

void findAndPrint(string line, string searchStr) {
   size_t result = line.find(searchStr);
   raise(SIGSTOP);
   // only print if there is a match
   if (result != string::npos) cout << line << endl;
}

int main(int argc, char **argv) {
   if (argc < 2) {
      cout << "usage:\n\t" << argv[0] << " searchstring" << endl;
      return 1;
   }
   string searchStr = argv[1];

   signal(SIGCHLD,childHandler);

   string line;
   vector<pid_t> children;
   while (getline(cin,line)) {
      if (children.size() >= kMaxProcesses) {
         printStoppedProcesses(children);
         children.clear();
      }
      pid_t pid = fork();
      if (pid == 0) { // child
         findAndPrint(line, searchStr);
         close(STDIN_FILENO); // child should not be reading data
         exit(0);
      }
      children.push_back(pid);
    }
    printStoppedProcesses(children);
    return 0;
}
```

Problem 3: Short Answer Questions [10 points]

Unless otherwise noted, your answers to the following questions should be 75 words or fewer. You needn't write in complete sentences provided it's clear what you're saying. Full credit will only be given to the best of responses. Just because everything you write is true doesn't mean you get all the points.

a. [2 points] The **rename** system call renames a file, moving it from one directory to another if necessary. It comes with the following prototype:

```
int rename(const char *ep, const char *np);
```

ep is short for **e**xisting **p**ath, and we'll assume it's an absolute path to a valid file you have permission to rename. **np** (short for **n**ew **p**ath, and also absolute) identifies where the file should be moved to and what new name it should assume. Any intermediate directories needed for the move are created. So, a call to

```
rename("/WWW/index.html","/archive/winter-2019/index-w19.html");
```

would remove **index.html** from **WWW** and move it to **archive/winter-2019**, creating **archive** and **winter-2019** if necessary, with the name of **index-w19.html**. The renaming works even if the file being moved is a directory or a symbolic link.

Without worrying about error checking, describe how **rename** could be efficiently implemented in terms of your Assignment 2 file system.

b. [2 points] Consider the following program, which creates and opens a file in read-write mode, writes five characters to the file, and then attempts to read a single character through the same descriptor:

```
int main(int argc, char *argv[]) {
  int f = open("foo.txt", O_RDWR | O_CREAT | O_EXCL, 0644);
  write(f, "abcde", 5);
  int ch;
  int count = read(f, &ch, 1);
  if (count == 0)
    printf("No data!\n");
  else
    printf("Got this: %c\n", ch);
  close(f);
  return 0;
}
```

When the above program runs, the output is **`No data!`**. Given the output, and leveraging what you know about descriptor, open file, and vnode tables, explain why the output is what it is.

c. [2 points] Some students questioned the need to use **pipe2** and **O_CLOEXEC** while implementing **subprocess**, claiming it was fine to just call **pipe** and manually close all pipe endpoints in the child process before the **execvp** call. That, as it turns out, is not true.

Consider the following test program:

```
int main(int argc, char *argv[]) {
   char *args[] = {const_cast<char *>("/usr/bin/sort"), NULL};
   subprocess_t sp1 = subprocess(args, true, false);
   subprocess_t sp2 = subprocess(args, true, false);
   dprintf(sp1.supplyfd, "hello\n");
   dprintf(sp1.supplyfd, "goodbye\n");
   dprintf(sp2.supplyfd, "bonjour\n");
   dprintf(sp2.supplyfd, "aurevoir\n");
   close(sp1.supplyfd);
   waitpid(sp1.pid, NULL, 0);
   close(sp2.supplyfd);
   waitpid(sp2.pid, NULL, 0);
   return 0;
}
```

When the above program is run with a **subprocess** implementation that uses **pipe2** and **O_CLOEXEC**, we see the expected output. When the above program is run using a **subprocess** implementation that just uses **pipe** (and manually closes the pipe endpoints in the child, before **execvp**), the process hangs.

Where does the program hang? And why?

d. [2 points] When **fork** is called, the set of currently blocked signals is preserved, and the set of installed signal handlers is preserved as well. When **execvp** is called, the set of currently blocked signals is preserved, but all installed signal handlers are cleared and replaced by default handlers.

- Defend the decision to preserve blocked signal sets across **execvp** boundaries.
- Explain why **execvp** restores all signal handlers to be the defaults.

e. [2 points] Consider the following **restartJob** function from my own **stsh** solution, where all error checking has been removed.

```
static void restartJob(const command& command, STSHJobState state) {
  size_t num = parseNumber(command.tokens[0]);
  STSHJob& job = joblist.getJob(num);
  forwardSignal(job, SIGCONT);
  job.setState(state);
  if (state == kForeground)
    waitForForegroundJob(job);
}
```

Because **restartJob** accesses and updates the global job list, one or more signals need to be blocked while **restartJob** is executing. Since you've only installed handlers for **SIGINT**, **SIGTSTP**, and **SIGCHLD**, those are the only three candidates. Which signal or signals need to be blocked, and why?

Scratch Paper

Scratch Paper