CS110 Practice Midterm 1

Problem 1: Exceptional Control Flow Calisthenics

a.) Consider the following program:

```
static int counter = 0;
int main(int argc, char *argv[]) {
  for (int i = 0; i < 2; i++) {
    fork();
    counter++;
    printf("counter = %d\n", counter);
  }
  printf("counter = %d\n", counter);
  return 0;
}
```

Assume there are no errors (e.g. **`fork`** doesn't fail) and that each **`printf`** call atomically flushes its output in full to the console.

- How many times would the value of **`counter`** be printed?
- What's the value of **`counter`** the very first time it's printed?
- What's the value of **`counter`** the very last time it's printed?
- Describe one scheduling scenario where the values of **`counter`** printed by all of the competing processes would not print out values in non-decreasing order.

b.) Consider the following program, which is a variation of a program I presented in lecture:

```
static pid_t pid; // necessarily global so handler1 has access to it
static int counter = 0;

static void handler1(int unused) {
  counter++;
  printf("counter = %d\n", counter);
  kill(pid, SIGUSR1);
}

static void handler2(int unused) {
  counter += 10;
  printf("counter = %d\n", counter);
  exit(0);
}

int main(int argc, char *argv[]) {
  signal(SIGUSR1, handler1);
  if ((pid = fork()) == 0) {
    signal(SIGUSR1, handler2);
    kill(getppid(), SIGUSR1);
    while (true) {}
  }
```

```
  if (waitpid(-1, NULL, 0) > 0) {
    counter += 1000;
    printf("counter = %d\n", counter);
  }

  return 0;
}
```

Again, assume that each call to **printf** flushes its output to the console in full, and further assume that none of the system calls fail in any unpredictable way (e.g. fork never fails, and **waitpid** only returns -1 because there aren't any child processes at the moment it decides on its return value).

- What is the output of the above program?
- What are the two potential outputs of the above program if the **while (true)** loop is completely eliminated? Describe how the two processes would need to be scheduled in order for each of the two outputs to be presented.
- Now further assume the call to **exit(0)** has also been removed from the **handler2** function . Are there any other potential program outputs? If not, explain why. If so, what is it (or are they)?

c.) Finally, consider the following program:

```
static int counter = 0;
static void handler(int sig) {
  counter++;
}

int main(int argc, char *argv[]) {
  signal(SIGCHLD, handler);
  for (int i = 0; i < 5; i ++){
    if (fork() == 0)
      exit(0);
  }

  while (waitpid(-1, NULL, 0) > 0);
  printf("counter = %d\n", counter);
  return 0;
}
```

Assume you know nothing of the fairness of the kernel's scheduler—e.g. assume the schedule is arbitrary about the order it chooses processes to run, that time slice durations might vary, and a process could be given two time slices before another process gets any.

- Yes or No: Does the program publish the same value of **counter** every single time?
- If your answer to the previous question is yes, what is the single value printed every time? If your answer to the previous question is no, list all of the possible values **counter** might be at the moment it's printed.

- How do your answers to each of the above questions change if the third argument to the one **waitpid** is **WNOHANG** instead of 0?

Problem 2: Implementing popen and pclose

We implemented **subprocess** during lecture, but you're actually able to implement the full-fledged **popen** and **pclose** functions (we used them for the **scrabble-word-finder-server** example in lecture) together, relying on global state—an array of **pid_t**s—to remember process IDs associated with the **FILE ***s created by **popen**. The prototypes for **popen** and **pclose** are:

```
FILE *popen(const char *command, const char *mode); // mode is either "r" or "w"
int pclose(FILE *processStream);
```

Recall that **popen** launches a separate process as per the provided command, and returns a **FILE *** to the slave processes's standard out (if the provided mode is **"r"**) or its standard in (if the provided mode is **"w"**). **pclose** takes a **FILE *** previously returned by **popen**, waits for the associated slave process to end, and then returns the process status as surfaced by **waitpid**.

While implementing these two functions, you should make the following assumptions:

- Your implementation should be in pure C, since **popen** and **pclose** have pure C prototypes.
- Assume the maximum number of file descriptors that can be open at any one time is 256, and that file descriptor integers are recycled, so they never need to exceed 255.
- Your implementation of **popen** and **pclose** should rely on a **static** global array of 256 **pid_t**s called **childProcesses**. Assume that **childProcesses[k]** stores the process id associated with the file descriptor number k, or that it stores 0 if the file descriptor isn't associated with some **popen**-issued **FILE ***. Further assume that the array initially contains all zeroes.
- The implementations of these two functions relies on two very simple functions you've not needed before, and they are:

  ```
  FILE *fdopen(int fd, const char *mode);
  int fileno(FILE *);
  ```

  fdopen is like **fopen**, except that it constructs and returns a **FILE *** for the provided integer descriptor instead of a named file. **fileno** returns the integer file descriptor associated with the provided **FILE ***.
- You needn't do any error checking whatsoever, and can assume the **command**, **mode**, and **processStream** parameters are all valid.
- Rely on **pipe**, **close**, **fork**, **execvp**, **fdopen**, **dup2**, and **fileno** for your answer.

Implement these two functions:

```
FILE *popen(const char *command, const char *mode); // mode is either "r" or "w"
int pclose(FILE *processStream);
```

Problem 3: Short Answers

a) Assume **"foo.txt"** is a legitimate, readable file of size 8. Consider the following short program.

```
int main(int argc, char *argv[]) {
   int fd1 = open("foo.txt", O_RDONLY);
   int fd2 = open("foo.txt", O_RDONLY);
   int fd3 = open("foo.txt", O_RDONLY);
   char ch;
   read(fd1, &ch, 1);
   dup2(fd1, fd3);
   if (fork() == 0) {
      read(fd3, &ch, 1);
      dup2(fd2, fd3);
      close(fd2);
   }
   read(fd3, &ch, 1);
   return 0;
}
```

Draw the state of the file descriptor tables, the file entry table, and the vnode table for both processes just prior to exit. Draw them in enough detail that you communicate your understanding of the data structures and the information needed to explain how the above program (or rather, pair of programs) works. Assume that **open** returns 3, 4, and 5 in that order.

Your answers to the following questions should be 50 words or less. Responses longer than 50 words will receive 0 points. You needn't write in complete sentences provided you convey the pertinent information. Full credit will be given to clear, correct, complete, relevant responses.

b) Recall that your Assignment 1 and 2 file system relies on the notion of layering, which is a special, more constrained form of modularity. What is accomplished by implementing the file system using layering, and in what way are layered modules deliberately more constrained than arbitrary modules?

c) In lecture, we discussed an algorithm for resolving an absolute pathname to arrive at its associated inode number, and we *also* discussed an algorithm for resolving a domain name (e.g. **math.uchicago.edu**) to its IP address (e.g. 128.135.10.19). Briefly list three distinct ways the two algorithms are similar.