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 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: `triplet`

Leverage your multiprocessing skills to implement **triplet**, which has the following prototype:

```
static int triplet(char *one[], char *two[], char *three[]);
```

triplet creates three processes (executing **NULL**-terminated argument vectors **one**, **two**, and **three**) and wires the standard output of the first to the standard input of the second, and both the standard output **and** standard error of the second to the standard input of the third. **triplet** should wait until all three processes exit, and should return the sum of the three return codes. Assume all system calls succeed (and in particular, the **execvp** call needed never return), and assume the executables expressed via **one[0]**, **two[0]**, and **three[0]** never crash. Your implementation must close all unused descriptors, and it should not wait on any child processes other than the three it creates.

```
static int triplet(char *one[], char *two[], char *three[]) {
```

Problem 3: Short Answers Questions

Your answers to the following questions should be 50 words or fewer. Responses longer than 50 words will receive 0 points. 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 technically true doesn't mean you get all the points.

a. Consider the prototype for the **link** system call, which is as follows

```
int link(const char *oldpath, const char *newpath);
```

A successful call to **link** updates the file system so the file identified by **oldpath** is also identified by **newpath**. Once **link** returns, it's impossible to tell which name was created first. (To be clear, **newpath** isn't just a symbolic link, since it could eventually be the only name for the file.)

In the context of your **assign1** file system design, briefly outline how **link** might be implemented.

b. The **publish-to-all** user program takes an arbitrary number of filenames as arguments and attempts to publish the date and time (via the **date** executable that ships with all versions of Unix and Linux). **publish-to-all** is built from the following source:

```
static void publish(const char *name) {
   printf("Publishing date and time to file named \"%s\".\n", name);
   int outfile = open(name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
   dup2(outfile, STDOUT_FILENO);
   close(outfile);
   if (fork() > 0) return;
   char *argv[] = {"date", NULL};
   execvp(argv[0], argv);
}

int main(int argc, char *argv[]) {
   for (size_t i = 1; i < argc; i++) publish(argv[i]);
   return 0;
}
```

Someone with a fractured understanding of processes, descriptors, and file redirection might expect the program to have printed something like this:

```
myth4> ./publish-to-all one two three four
Publishing date and time to file named "one".
Publishing date and time to file named "two".
Publishing date and time to file named "three".
Publishing date and time to file named "four".
```

However, that's not what happens. What text is actually printed, and what do each of the four files contain?

c. Assume **a.txt** and **b.txt** are each 5 byte files. Consider the following:

```
static void consume(int fd) {
   char ch;
   read(fd, &ch, 1);
}

int main(int argc, char **argv) {
   int one = open("a.txt", O_RDONLY);
   int two = open("b.txt", O_RDONLY);
   consume(one);
   pid_t pid = fork();
   int three = pid == 0 ? one : dup(one);
   consume(two);
   consume(three);
   dup2(two, three);
   sleep(10);
   return 0;
}
```

When the above program is executed, it splits into two and each, with high probability, takes 10 seconds naps, more or less at the same time. In the space below, draw the state of the file descriptor tables, file entry table, and vnode table while the two processes are within their **sleep(10)** calls. If the final drawing depends on how the two processes are selected for CPU time, then say so, and present any valid final drawing.

d. While implementing the **farm** program for **assign2**, you were expected to implement a **getAvailableWorker** function to return the index of a self-halted worker record. My own solution is presented here:

```
static sigset_t waitForAvailableWorker() {
   sigset_t existing, additions;
   sigemptyset(&additions);
   sigaddset(&additions, SIGCHLD);
   sigprocmask(SIG_BLOCK, &additions, &existing);
   while (numWorkersAvailable == 0) sigsuspend(&existing);
   return existing;
}

static size_t getAvailableWorker() {
   sigset_t existing = waitForAvailableWorker();
   size_t i;
   for (i = 0; !workers[i].available; i++);
   assert(i < workers.size());
   numWorkersAvailable--;
   workers[i].available = false;
   sigprocmask(SIG_SETMASK, &existing, NULL); // restore original block set
   return i;
}
```

- Had I accidentally passed in **&additions** to the **sigsuspend** call instead of **&existing**, the farm could have deadlocked. Explain why.

- Had I accidentally omitted the **sigaddset** call and not blocked **SIGCHLD**, **farm** could have deadlocked. Explain how.

e. Your implementation of **trace** relied on **ptrace**'s ability to read system call arguments from registers via the **PTRACE_PEEKDATA** command. When a system call argument was a C string, you needed to rely on repeated calls to **ptrace** and the **PTRACE_PEEKUSER** option to pull in characters, eight bytes at a time, until a zero byte was included. At that point, the entire **'\0'**-terminated C string could be printed.

Was this more complicated than need be? If, after all, the argument register contains the base address of a **'\0'**-terminated character array, why can't you just **<<** the **char *** to **cout** and rely on **cout** to print the C string of interest?

f. For **assign2**, you were asked to implement the **pipeline** function, which accepts two argument vectors and creates sister processes such that the standard output of the first fed the standard input of the second.

In the name of simplicity, your **pipeline** implementation wasn't expected to do any error checking at all and could assume all system calls succeeded. However, a robust **pipeline** implementation would handle all possible system call errors. Had I instead required you to handle system call failure and all possible **errno** values, how could you have used **ptrace** to implement a test framework that confirm that **pipeline** gracefully fails when its *second* **fork** call fails with **errno EAGAIN** (i.e. the number of available user processes would be exceeded)?

Hint: recall **ptrace**'s ability to read and even **modify** another process's registers; **PTRACE_PEEKUSER** allows you to read another process's registers, and **PTRACE_POKEUSER** allows you to change one!

g. Your implementation of **stsh**—your tiny shell assignment—relied on custom signal handlers to intercept and forward **SIGINT** and **SIGTSTP** signals on to the foreground process group (a.k.a. job), and it did so by passing the negative of the job's group id, or pgid, to the **kill** system call (e.g. **kill(-pgid, SIGINT)**). Some students also updated the job list within the handler to mark the job as **kTerminated** or **kStopped**, but the decision to do that was incorrect. Why?

h. When I type the following line in at the command prompt on a **myth** machine, I create a background job with five processes.

```
myth15> echo abcd | ./conduit --delay 2 | ./conduit | ./conduit | ./conduit &
[1] 10004 10005 10006 10007 10008
```

The **echo** process, which immediately prints and flushes **abcd** to the standard input of the first **conduit** process in the pipeline, has a process id of 10004. The first **conduit** process—the one fed by **echo**—has a process id of 10005, the second has a process id of 10006, and so forth. (Recall that **assign3**'s **conduit** simply passes everything read from standard input on to standard output, although the **--delay** flag specifies the number of seconds that should pass before the next single character is read and printed.)

- How would each of the four **conduit** processes terminate if I send a **SIGKILL** (the kill-program signal that can't be blocked or handled) to pid 10005 five seconds after launching the job?
- How would each of the four **conduit** processes terminate if I instead send a **SIGKILL** to pid 10007 five seconds after launching the job?