CS110 Practice Midterm 1 Solution

Solution 1: Exceptional Control Flow Calisthenics

a.) Reference 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;
}
```

- How many times would the value of **`counter`** be printed?

  Answer: 10 times

- What's the value of **`counter`** the very first time it's printed?

  Answer: **`counter = 1`**

- What's the value of **`counter`** the very last time it's printed?

  Answer: **`counter = 2`**

- 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.

  In principle, the parent process could run to completion before any of the forked child processes execute. If that were the case, then the first three lines of the output would be:

  ```
  counter = 1
  counter = 2
  counter = 2
  ```

  The first child process (spawned by the original) might finally get processor time, return from fork and advance on to the counter++ line, which promotes its own copy of the counter global from 0 (the value the parent's counter was at the time fork was called) to 1. It could then print the following:

counter = 1

That's enough to illustrate how some **2**'s could precede some **1**'s in the accumulation of all four processes' outputs.

b.) Reference program:

```
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;
}
```

- What is the output of the above program?

  The combination of blocking while (true) loops and exit calls imposes a linearity to the way parent and child block and signal each other, so there's only one possible output:

  counter = 1
  counter = 10
  counter = 1001

- 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.

  The output above (the **`1`**-**`10`**-**`1001`**) output is still possible, because the child process can be swapped out just after the **`kill(getppid(), SIGUSR1)`** call, and effectively emulate the stall that came with the **`while (true)`** loop when it was present.

  However, since the **`while (true)`** loop really is gone, the child process could complete and exit normally before the parent process—via its **`handler1`** function—has the opportunity to signal the child. That would mean **`handler2`** wouldn't even execute, and in that case, we wouldn't expect to see **`counter = 10`**. (Note that the child process's call to **`waitpid`** returns **`-1`**, since it itself has no grandchild processes of its own).

  Redux on possible outputs:

  ```
  counter = 1
  counter = 10
  counter = 1001
  ```

  or

  ```
  counter = 1
  counter = 1001
  ```

- 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)?

  No other potential outputs, because:
  - **`counter = 1`** is still printed exactly once, just in the parent, before the parent fires a **`SIGUSR1`** signal at the child (which may or may not have run to completion).
  - **`counter = 10`** is potentially printed if the child is still running at the time the parent fires that **`SIGUSR1`** signal at it. The **`10`** can only appear after the **`1`**, and if it appears, it must appear before the **`1001`**.
  - **`counter = 1001`** is always printed last, after the child process exits. It's possible that the child existed at the time the parent signaled it to inspire **`handler2`** to print a **`10`**, but that would happen before the **`1001`** is printed.

- Note that the child process either prints nothing at all, or it prints a **10**. The child process can never print **1001**, because its **waitpid** call would return **-1** and circumvent the code capable of printing the **1001**.

Don't freak out by the level of detail I provided defending why there are only two possible outputs, even after the while and exit lines have been removed. I wouldn't need anything as elaborate as what I've provided. I would just need some scientific method defense that a **1** and a **1001** are always printed exactly once, and that a **10** is potentially printed in between them.

c.) Relevant 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;
}
```

- Yes or No: Does the program publish the same value of **counter** every single time?

  Answer: no way

- 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.

  The parent process blocks until all child processes have exited. How the five children and the parent processes are scheduled, however, dramatically impacts how many times handler executes.

  As it turns out, values of **1**, **2**, **3**, **4**, and **5** are all possible.
  - Why 5? Imagine that the five children are scheduled to make progress, but that they complete (e.g. call **exit(0)**) far apart enough that **handler** executes to completion before the next in the series of **SIGCHLD** signals is fired. This would allow **handler** to be executed five times.
  - Why 1? Imagine the scenario that where the parent process (which includes its normal control flow and its **handler** function) gets swapped out after the final

iteration of its **for** loop but before any of the child processes exit. Further imagine that the parent doesn't get any processor time whatsoever until all five child processes have exited and collectively prompted the kernel to deliver five **SIGCHLD** signals to the parent (which results in a single **SIGCHLD** bit being set high). When the parent finally gets processor time after years of waiting, it detects the high **SIGCHLD** bit, is forced to execute **handler** exactly one time to respond to the signal, and returns to the normal control flow to finally move beyond the (until then, blocking) **waitpid** call.

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

    - Now **counter = 0** is possible too. Imagine the parent gets processor time before any of the child processes get enough time to exit, and the parent makes it to (the now non-blocking) **waitpid** call, which immediately returns a 0 (since all child processes are still running), moves past its **while** loop, and then prints the state of its global variable **counter**, which has never been incremented, since **handler** has never been called.

  In practice, you might not see some of the extreme possibilities very often or even at all. But that doesn't mean that you shouldn't care about what's possible. You might test on a single-core machine with one user and less then 50 competing processes, where all processes are scheduled with equal priority. The same code running on another system (64 cores? approximately 10000 processes? experimental scheduling algorithms?) might bring out one of these extremes, so you need to understand what's possible, or else you can't claim an expertise in multiprocessing.

  Note that you didn't need to defend your numbers for this problem. But I figured you wanted to know why the full range of possibilities are in fact possibilities, so I just kept writing.

Solution 2: Implementing `popen` and `pclose`

```
static pid_t childProcesses[256]; // static globals are initially zeroed out

FILE *popen(const char *command, const char *mode) {
  int fds[2];
  pipe(fds);
  pid_t pid = fork();
  if (pid == 0) {
    int index = mode[0] == 'w' ? 0 : 1;
    int stdfd = mode[0] == 'w' ? STDIN_FILENO : STDOUT_FILENO;
    dup2(fds[index], stdfd);
    close(fds[0]);
    close(fds[1]);
    char *argv[] = {"/bin/sh", "-c", (char *) command, NULL};
```

```
    execvp(argv[0], argv);
  }

  int indexToClose = mode[0] == 'w' ? 0 : 1;
  int indexToKeep = mode[0] == 'w' ? 1 : 0;
  close(fds[indexToClose]);
  FILE *fp = fdopen(fds[indexToKeep], mode);
  childProcesses[fds[indexToKeep]] = pid;
  return fp;
}
```

One note about the above implementation: The implementation assumes that the parent process hasn't closed **`stdin`** and/or **`stdout`**. Had the parent done that, file descriptors 0 and/or 1 could have been reused and reintroduced to the parent process by the call to **`pipe`**. A fully robust solution would need to guard against this (so you don't risk calling something like **`dup2(1, 1)`**), but I certainly wouldn't require that from your answer, particularly given that I didn't mention it in the problem statement.

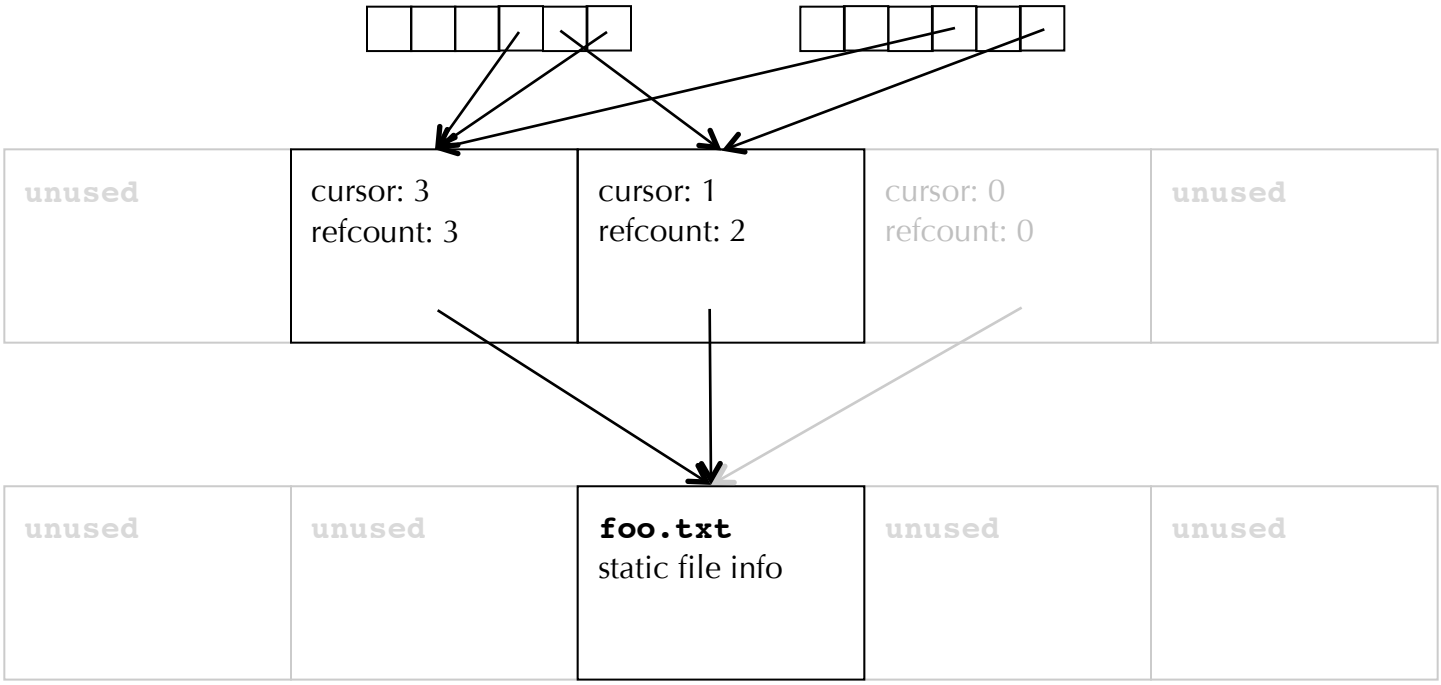
```
int pclose(FILE *processStream) {
   int fd = fileno(processStream);
   pid_t pid = childProcesses[fd];
   assert(pid > 0); // not needed for correct solution.. just a sanity check
   childProcesses[fd] = 0;
   fclose(processStream);
   int status;
   waitpid(pid, &status, 0);
   return status;
}
```

Solution 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?

- The layers work to make hardware (the inode and block-number-oriented hard drive) look like software (the name-oriented filesystem): 2 points
- Layered modules are constrained to interact with just the layers above and below it, whereas arbitrary modules can interact with any other module it wants to: 2 points

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.

- Both work to turn something human-readable into something more easily handled by software and hardware.
- Both approaches manage a lookup of a number based on the tokenization of a name.
- Both have well-defined bootstrap locations (well-defined inode number for the root directory, well-known IP addresses of bootstrap servers consulted by initial DNS lookup)

d) Your implementation of **`tsh`**—your tiny shell assignment—relied on custom signal handlers to intercept and forward **`SIGINT`** and **`SIGTSTP`** signals on to the foreground process group, and it did so by passing the negative of the primary process's pid to the **`kill`** system call (e.g. **`kill(-pid, SIGINT)`**). Many students encountered bugs that resulted from calling **`kill(pid, SIGINT)`** instead of **`kill(-pid, SIGINT)`**, but the bugs only presented themselves for certain type of foreground jobs. What types of foreground jobs are immune to this bug, and what types aren't?

Foreground jobs that don't spawn child processes are immune, those that do aren't.