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 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: `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** calls 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[]) {
  pid_t pids[3];
  int fds[4];
  pipe(fds);
  pids[0] = fork();
  if (pids[0] == 0) {
    close(fds[0]);
    dup2(fds[1], STDOUT_FILENO);
    close(fds[1]);
    execvp(one[0], one);
  }
  close(fds[1]);
  pipe(fds + 2);
  pids[1] = fork();
  if (pids[1] == 0) {
    close(fds[2]);
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]);
    dup2(fds[3], STDOUT_FILENO);
    dup2(fds[3], STDERR_FILENO);
    close(fds[3]);
    execvp(two[0], two);
  }
  close(fds[0]);
  close(fds[3]);
  pids[2] = fork();
  if (pids[2] == 0) {
    dup2(fds[2], STDIN_FILENO);
    close(fds[2]);
    execvp(three[0], three);
  }
  close(fds[2]);
  int sum = 0;
  for (size_t i = 0; i < 3; i++) {
    int status;
    waitpid(pids[i], &status, 0);
    sum += WEXITSTATUS(status);
  }
  return sum;
}
```

Solution 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.

- Resolve **oldpath** to its **inode** number, append new **dirent** to sequence of **dirent**s in the directory where **newpath** belongs
- New **dirent** should contain name of file and same inumber
- Increment **inode**'s reference count

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?

Printed to console:

```
myth4:/usr/class/cs110/staff/midterm> publish-to-all one two three four
Publishing date and time to file named "one".
```

Contents of **one** (similar for two and three, lines can be interchanged):

```
Publishing date and time to file named "two".
Fri Oct 28 10:14:14 PDT 2016
```

Contents of **four**:

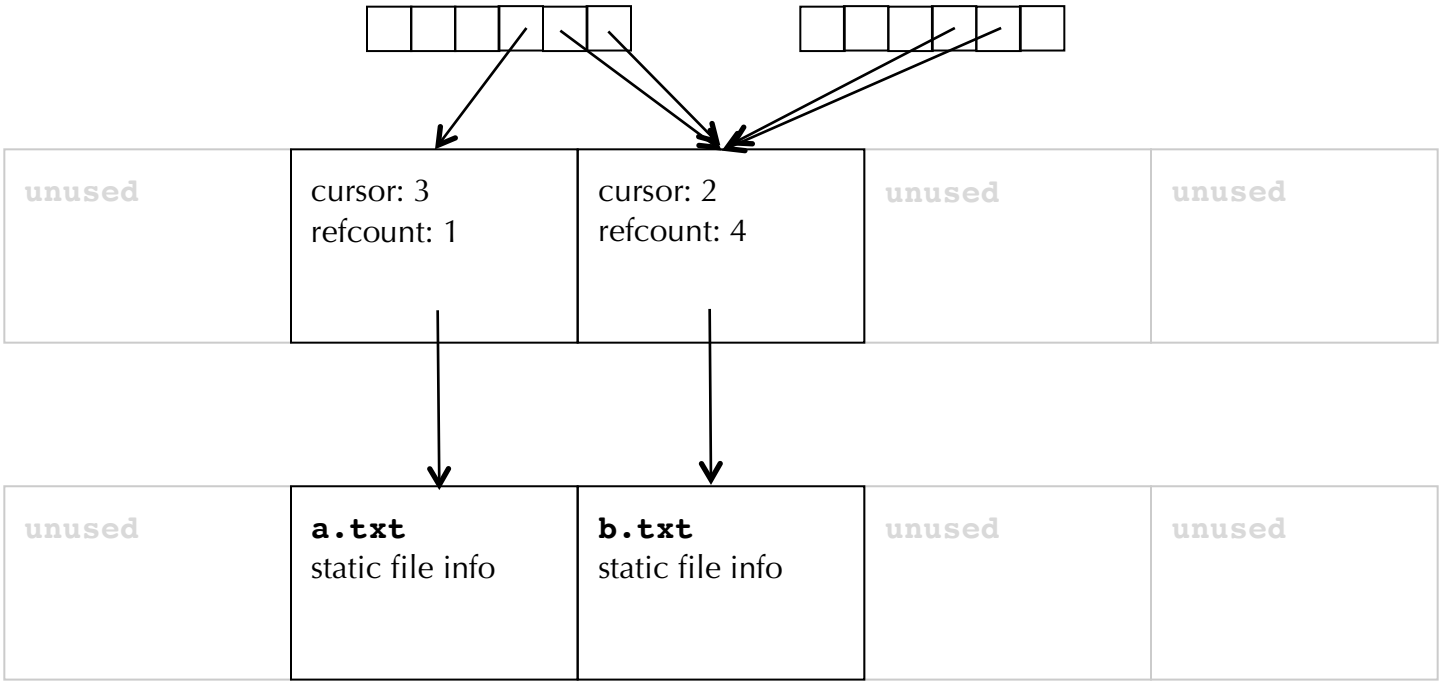
```
Fri Oct 28 10:14:14 PDT 2016
```

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.

  numWorkersAvailable == 0 could pass, **sigsuspend** forces **farm** to deadlock, as only **SIGCHLD** signals are coming in, and they're blocked.

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

  numWorkersAvailable == 0 passes, farm swapped off CPU, all **kNumCPUs** workers self-halt, all **kNumCPUs SIGCHLD**s handled by one **SIGCHLD** handler call, **farm** descends into **sigsuspend**, no additional **SIGCHLD**s ever arrive to wake **farm** up.

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?

   > Register contains the address of a C string in tracee's virtual address space, but **operator<<(ostream& os**, **const char *str)** prints C string at address in tracer's virtual address space.

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 ensures **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!

   > Test could monitor system call activity as **assign2**'s **trace** does, paying attention to exit from second **fork**. When tracee is frozen on **fork** return, write **-EAGAIN** to **RAX** and signal to continue.

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's processes as **kTerminated** or **kStopped**, but the decision to do that was incorrect. Why?

   All processes in a job might have installed **SIGINT** and/or **SIGTSTP** handlers that allow each process to continue without terminating or stopping.

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?

     - first **conduit** would be killed
     - all other **conduit**s would exit gracefully

   - How would each of the four **conduit** processes terminate if I instead send a **SIGKILL** to pid 10007 five seconds after launching the job?

     - first two **conduit**'s would exit via uncaught signal, that signal being a **SIGPIPE**
     - final **conduit** exits gracefully