CS110 Practice Midterm 2 Solution

Solution 1: **duet**

Leverage your **pipe**, **fork**, **dup2**, and **execvp** skills to implement **duet**, which has the following prototype:

```
static void duet(int incoming, char *one[], char *two[], int outgoing);
```

incoming is a valid, read-oriented file descriptor, **outgoing** is a valid, write-oriented file descriptor, and **one** and **two** are well-formed, **NULL**-terminated argument vectors. **duet** launches two child processes, the first of which executes the program identified in **one**, the second of which executes the program identified in **two**.

The first process's standard input is rewired to draw bytes from **incoming**, and its standard output is rewired to feed bytes to the standard input of the second process, which itself directs its standard output to whatever resource is bound to **outgoing**. The function waits for the two processes (and only those two processes) to run to completion before returning.

Use this and the next page to present your implementation of **duet**. You may assume that all system calls succeed, and that the executables identified by **one** and **two** always run to completion without crashing. You should close all unused file descriptors (including **incoming** and **outgoing** once you've leveraged their resources).

```
static void duet(int incoming, char *one[], char *two[], int outgoing) {
  pid_t pids[2];
  int fds[2];
  pipe(fds);
  pids[0] = fork();
  if (pids[0] == 0) {
    close(fds[0]);
    close(outgoing);
    dup2(incoming, STDIN_FILENO);
    close(incoming);
    dup2(fds[1], STDOUT_FILENO);
    close(fds[1]);
    execvp(one[0], one);
  }

  close(incoming);
  close(fds[1]);
  pids[1] = fork();
  if (pids[1] == 0) {
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]);
    dup2(outgoing, STDOUT_FILENO);
    close(outgoing);
    execvp(two[0], two);
  } // continued on next page
```

```
    close(outgoing);
    close(fds[0]);
    waitpid(pids[0], NULL, 0);
    waitpid(pids[1], NULL, 0);
  }
```

Solution 2: Short Answer Questions

Unless otherwise noted, 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 true doesn't mean you get all the points.

a. The **dup2** system call accepts two presumably valid file descriptors, detaches the second of the two from its file session, and then attaches it to whatever the first descriptor is attached to. Briefly outline what happens to the relevant file entry table , as a result of **dup2** being called.

- Decrement refcount of old file table entry, increment refcount of new one
- If refcount of former falls to zero, remove it

b. Explain what happens when you type **cd ../..** at the shell prompt. Frame your explanation in terms of your Assignment 1 file system and the fact that the inode number of the current working directory is the only relevant global variable maintained by your shell.

Search cwd's payload for .., set inumber of cwd to inumber associated with ..; repeat one more time.

c. Consider the prototype for the **flock** system call, which is as follows:

```
int flock(int fd, int op);
```

flock can be used to gain exclusive access to the file session bound to **fd**. The **op** parameter can (for the purposes of this problem) be one of two constants, and those constants are:

- **LOCK_EX**, which is a request to grab exclusive access to a file session that should be respected by all other processes. If the resource isn't locked at the time of the call, then it is locked and **flock** returns right away. If the resource is locked, then the process blocks within the **flock** call until the lock is lifted by another process.
- **LOCK_UN**, which releases the lock held on a resource (or is a no-op if the lock wasn't held in the first place).

- Explain why information about the locked state of a file session needs to be stored in a file entry table instead of a file descriptor table.

  For lock to be respected by all processes, info must be stored in data structure referenced by all processes.

- Explain why descriptors created using **`dup`** might reference locked file sessions, but descriptors created using **`open`** initially reference a file session that is guaranteed to be unlocked.

  `dup` returns a descriptor referencing an existing session, promotes its refcount to be at least 2. **`open`** creates new session with refcount of 1.

d. Typically, each page of a process's virtual address space maps to a page in physical memory that no other virtual address space maps to. However, when two processes are running the same executable (e.g. you have two instances of **`emacs`** running,) some pages within each of the two processes' virtual address spaces can map to the same exact pages in physical memory. Identify one segment within the processes' virtual address spaces that could be backed by the same pages of physical memory, and briefly explain why it's possible.

- Two such segments? .text/.code and .rodata (i.e. read-only data).
- Why? Possible because segment content is immutable.

e. Your **`assign1`** file system relied on direct indexing for small files and singly and doubly indirect indexing for large files. In the name of code uniformity, you could have just represented all files, large and small, using doubly indirect indexing. Briefly describe the primary advantage (other than uniformity of implementation) and primary disadvantage of relying on doubly indirect indexing for all file sizes.

- Advantage: supports even larger files than original design.
- Disadvantage: accessing payload for very small files requires many disk accesses, cache unfriendly

f. Recall that the stack frames for system calls are laid out in a different segment of memory than the stack frames of user functions. How are the parameters passed to the system calls received when invoked from user functions? And how is the process informed that all system call values have been placed and that it's time to execute?

- system call code and all parameters are received through registers
- system call is invoked by issuing a trap

g. While implementing the **farm** program for **assign2**, you were expected to implement a **getAvailableWorker** function to effectively block **farm** until at least one worker was available. My own **getAvailableWorker** relied on this helper function:

```
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;
}
```

The first quarter I used this assignment, a student asked if one could just use the **pause** function instead, as with:

```
static sigset_t waitForAvailableWorker() {
   sigset_t mask;
   sigemptyset(&mask);
   sigaddset(&mask, SIGCHLD);
   sigprocmask(SIG_BLOCK, &mask, NULL);
   while (numWorkersAvailable == 0) {
      sigprocmask(SIG_UNBLOCK, &mask, NULL);
      pause();
      sigprocmask(SIG_BLOCK, &mask, NULL);
   }
}
```

The zero-argument **pause** function doesn't alter signal masks like **sigsuspend** does; it simply halts execution until the process receives any signal whatsoever and any installed signal handler has fully executed. This is conceptually simpler and more easily explained than the version that relies on **sigsuspend**, but it's flawed in a way my solution is not. Describe the problem and why it's there.

Program can deadlock. Problem: After **SIGCHLD** is unblocked, all workers become available, **numAvailableWorkers** becomes maximum value, **main** execution flow descends into **pause** and no more signals are ever sent.

h. My own **farm** solution included this implementation for **closeAllWorkers**, which you can assume is correct:

```
static void closeAllWorkers() {
   for (size_t i = 0; i < workers.size(); i++) {
      getAvailableWorker();
   }

   signal(SIGCHLD, SIG_DFL);
   for (size_t i = 0; i < workers.size(); i++) {
      close(workers[i].sp.supplyfd);
      kill(workers[i].sp.pid, SIGCONT);
   }
```

```
      for (size_t i = 0; i < workers.size(); i++) {
         waitpid(workers[i].sp.pid, NULL, 0);
      }
   }
```

- [1 point, 25 words] Could I have exchanged the **close** and **kill** calls within the second **for** loop without impacting a worker's ability to exit? Justify your answer.

  Yep. **kill** would prompt worker to continue and perhaps block on standard input, but it would EOF out as soon as **farm** closed **supplyfd**

- [1 point, 25 words] Assume that I called **waitpid** using **WUNTRACED** instead of 0. Would the program have behaved any differently? Justify your answer.

  No. At that point, workers are incapable of self-halting, so **WUNTRACED** is effectively a 0.

i. Your **stsh** supports the **slay** builtin, which was used to terminate a single process, even if the process is stopped at the time it is **slay**ed. You were told to terminate the process using **SIGKILL** instead of **SIGINT**, because **SIGINT** won't terminate a stopped process until it is restarted. Why does a stopped process need to be restarted before a tabled **SIGINT** can terminate it?

   Stopped process could have a **SIGINT** handler installed, and needs to be able to execute it to see if it allows process to live. It can only execute post-**SIGINT** if it's restarted.

j. When establishing a new process group for a pipeline of two or more commands (as with **echo "abcdefgh" | ./conduit --count 4**), your **stsh** implementation needs to call **setpgid** in both the parent and in each of the children ("in order to avoid some race conditions", as the handout stated it). Describe the race condition that could cause problems if the parent didn't call **setpgid** and instead just relied on each of the children to call it.

   Only the parent of a pid or the process with that pid can establish that pid as a new pgid. If second child calls **setpgid** before first child does, **setpgid** will fail and be left in shell's process group.