CS110 Practice Midterm 3 Solution

Solution 1: `findexec`

Implement the **`findexec`** function, which has the following prototype:

```
void findexec(char *root, char *pattern, char *command[]);
```

This **`findexec`** function runs two sister processes—the first running **`find`** and a second running **`xargs`**—such that the standard output of the first leads to the standard input of the second. The **`root`** and **`pattern`** parameters shape how the first process does its job, and **`command`** is a **`NULL`**-terminated vector of arguments that shape how the second process does its job. Restated, the **`root`**, **`pattern`**, and **`command`** parameters prompt **`findexec`** to do the same thing that the following command would have achieved in **`stsh`**:

```
stsh> find root -name pattern -print | xargs command
```

Your implementation must do the following:

- construct the argument vector for the first process to include the supplied **`root`** and **`pattern`** variables
- construct the argument vector for the second process where **`"xargs"`** has been prepended to the series of tokens residing within **`command`**
- create two new sister processes such that the standout output of the first is wired to the standard input of the second
- transform the first to execute find, and the second to execute xargs
- close all unused file descriptors
- wait for each of the two processes to finish before returning

You may assume all system calls work as intended and need not do any error checking at all. You may not use **`pipeline`** or **`subprocess`** from lecture or the assignment set. In fact, I basically want you to re-implement something akin to Assignment 3's **`pipeline`**, but specific to **`find`** and **`xargs`**. Note that we're not asking you to implement **`find`** or **`xargs`**, as we've already implemented our own versions of them in lecture and in discussion section.

Solution 1: `findexec` [continued]

```
size_t countTokens(const char *command[]) {
  size_t count;
  for (count = 0; command[count] != NULL; count++) {}
  return count;
}

void findexec(char *root, char *pattern, char *command[]) {
  int fds[2];
  pipe(fds);
  pid_t pid1 = fork();
  if (pid1 == 0) {
    dup2(fds[1], STDOUT_FILENO);
    close(fds[0]);
    close(fds[1]);
    const char *find[] = {"find", dir, "-name", pattern, "-print", NULL};
    execvp(find[0], const_cast<char **>(find));
  }

  close(fds[1]);
  pid_t pid2 = fork();
  if (pid2 == 0) {
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]);
    size_t count = countTokens(command);
    const char *xargs[count + 2];
    xargs[0] = "xargs";
    memcpy(xargs + 1, command, (count + 1) * sizeof(char *));
    execvp(xargs[0], const_cast<char **>(xargs));
  }

  close(fds[0]);
  waitpid(pid1, NULL, 0);
  waitpid(pid2, NULL, 0);
}
```

Solution 2: Short Answer Questions [14 points]

Unless otherwise noted, your answers to the following questions should be 75 words or fewer. 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. [2 points] **`remove`** is a C library function that removes a name from the file system. If the supplied name was the last one to identify the file, then the file itself is truly deleted and its resources donated back for reuse. In the context of your **`assign2`** filesystem design, explain how the file system would need to be updated to fully realize a call to **`remove`**.

Find relevant directory entry in parent directory's payload, remove entry after decrementing the corresponding inode's reference count. If that reference count falls to zero, deallocate all payload blocks and mark inode as free.

b. [2 points] Recall that the **struct inode** from **assign2** looked like this:

```
struct inode { // some fields irrelevant to the problem are omitted
  uint16_t  i_mode;     // bit vector of file type and permissions
  uint8_t   i_size0;    // most significant byte of size
  uint16_t  i_size1;    // lower two bytes of size
  uint16_t  i_addr[8];  // device addresses constituting file
};
```

One CS110 student once proposed the following idea: for files with sizes that are just slightly larger than a perfect multiple of the block size (e.g. 1027, when the block size is 256), those last few bytes (e.g. the last three bytes of a 1027-byte file) could be stored in the inode itself. Describe how you would support this optimization so that entire blocks needn't be allocated just to store a few bytes of payload.

Store last few bytes in the unused **i_addr** entries. In the case of the 1027-byte file, those three extra bytes could be dropped in the space set aside for **i_addr[4]** and the first byte of **i_addr[5]**. It's easy to discern from the overall file size how many extra bytes there are and if they'll fit in unused **i_addr** entries.

c. [2 points] Referring to your implementation of **findexec** in Problem 1, identify the one **close** call that, if omitted, would prevent **findexec** from doing its job. Then explain why that one **close** call is so crucial to everything working as expected.

The call to **close(fds[1])** in the parent is needed, else the reference count of the write end of the pipe will never fall to zero, and **EOF** will never be detected by the process running **xargs**.

d. [2 points] Your implementations of **farm** and **stsh** each relied on signal handlers—**farm** relied on a **SIGCHLD** handler to identify stopped processes, and **stsh** relied on a **SIGCHLD** handler to identify processes that have stopped, exited, crashed, or exited normally. In fact, **farm** could have been implemented just as easily without custom handlers, whereas **stsh** really needed them. Explain why this is true.

The orchestrator in **farm** has nothing to do other than wait for workers to become available, so the **waitpid** calls could be done inline without impacting correctness and code clarity. Not true with **stsh**! Child processes may finish while the shell is blocking on standard input, waiting for a foreground process to fall out of the foreground, and so forth. Restated, **stsh** might be blocked on something unrelated to a process's state change.

e. [2 points] When establishing a new process group for a pipeline of two or more commands (as with **echo "abcdefgh" | ./conduit --count 4**), your **stsh** implementation needed 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 first child didn't call **`setpgid`** and instead just relied on the parent to call it before moving on to the create the second child.

First process might finish before parent calls **`setpgid`**, at which point the child's pid is no longer a valid pgid.

f. [2 points] Explain what the scheduler does when a program makes an otherwise valid call to **`read`** at a time when no data is available. Further explain what the scheduler does so that it's informed when data does become available.

Scheduler pulls process off the CPU, constructs a process control block out of the CPU state, and places that PCB in the blocked set. The scheduler also schedules an I/O interrupt to be triggered when data become available so it can lift it out of the blocked set into the ready queue.

g. [2 points] Many students asked if one signal handler can be interrupted by a signal of a different type. Describe a simple coding experiment you could run to list all the signals capable of interrupting a **`SIGCHLD`** handler.

Install a signal handler for **`SIGCHLD`** that for loops around calls to **`raise(sig)`**, where sig loops though all signal numbers. Then install a generic signal handler for all non-**`SIGCHLD`** signals that runs **`cout << "I interrupted SIGCHLD: " << sig`**, where **`sig`** is the generic handler's one parameter.