CS61 Section Notes - Shell Stuff

We'll be using the section repo today. If you don't have it handy run "git clone git@code.seas.harvard.edu:cs61/cs61-section.git" or otherwise just git pull to get the newest code (it'll be in the s08/ directory). Type "make" and then to run some file foo.c referenced in the notes, just do "./foo".

1. fork(), execvp(), waitpid()

What does fork() do?
pid_t fork(void);

- Creates a new child process
- Copies all the memory of the parent process to the child process
    - Note that in pset4 you copied all user-accessible, writeable memory in the parent to the child. Most OS's would actually use copy-on-write, where all writeable memory is marked as readonly in both parent and child, and then copied when either parent or child attempts to write to it.
- child process starts running as though it had called fork(), but 0 is returned to it. (RUN fork_1.c)
- parent process resumes running, with the child's pid returned from the fork() call
- Copies the parent's file descriptor table to the child. This means the child has the same files open as the parent. Also, these open files share state, so if the parent seeks in one of these open files, the seek will effect both parent and child

QUESTION:
Look through **fork_2.c**. What are possible outputs?

int execvp(const char *file, char *const argv[]);

- takes in a file (either a path to a binary like "./prog" or something like "ls", which it will then find the path to), and a NULL terminated array of arguments to the program (like main's argv, with argv[argc] (the end of the array) == NULL)
- executes the given program, replacing the current memory of the calling program with the memory for the given program (but the file descriptor table stays the same--more on this later)
- this also means that on success execvp will not return (because the program that called it "turns into" the program being executed)

pid_t waitpid(pid_t pid, int *status, int options)

- waitpid(pid, NULL, 0); // returns when the process with process id pid exits
- we could also pass in an int status by reference to get various information about the process after it exits. For instance the following:
  int status;
  waitpid(pid, &status, 0);
  // make sure it exited normally (vs e.g. being killed)
  if (WIFEXITED(status))
  printf("%d\n", WEXITSTATUS(status));

Will print the child's exit status (i.e. the return value of main or the number passed to an exit() call).

See **man waitpid** for more details.

Using this we can now make a program that takes as an argument a program, runs that program, waits for it to finish, and then prints out "done" (sounds suspiciously like a shell!). (RUN fork_3.c; Reproduced Below)

```
int main(int argc, char **argv) {
	if (argc != 2) {
		printf("Usage: %s program-name", argv[0]);
		exit(1);
	}
	pid_t pid = fork();
	if (pid == -1) {
		perror("Could not fork!\n");
	} else if (pid == 0) { // child
		// a program's first argument is the program's name/path
		char *child_argv[] = {argv[1], NULL};
		execvp(argv[1], child_argv); // does not return on success
		perror("Couldn't execute program!\n");
	} else { // parent
		waitpid(pid, NULL, 0);
		printf("done\n");
	}
}
```

More on fork()

How many a's will print out when the following programs are run:

1. (RUN fork_4.c; Reproduced Below)

```
int main() {
	printf("a");
```

```
        fork();
        printf("a");
}
```

2. (RUN fork_5.c; Reproduced Below)

```
int main() {
        printf("a\n");
        fork();
        printf("a\n");
}
```

3. What about if we redirected program 1's output to a file?

4. Program 2's output?

2. Backgrounding in Shell

Let's say we want to get a list of all files on our system that changed in the last 24 hours.
The command **find / -ctime -1** will do that for us (see **man find** for more info on how find works), but will take a really long time to run.
So let's background it!
find / -ctime -1 > changed-files.txt &
The & tells the shell to run the given program in the background. So it will write a list of changed files to changed-files.txt, in the background (meaning we'll get a new shell prompt immediately instead of having to wait for it to complete).

But what if we'd already been running **find / -ctime -1 > changed-files.txt** in the foreground for an hour before we saw this?!
You can type **Ctrl+z** to suspend the current process. This stops execution of the process but does not kill it. Now, we can type "**bg**" and the process we just suspended will be resumed but now in the background.

If we later wanted to put our process back in the foreground, we simply type "**fg**".

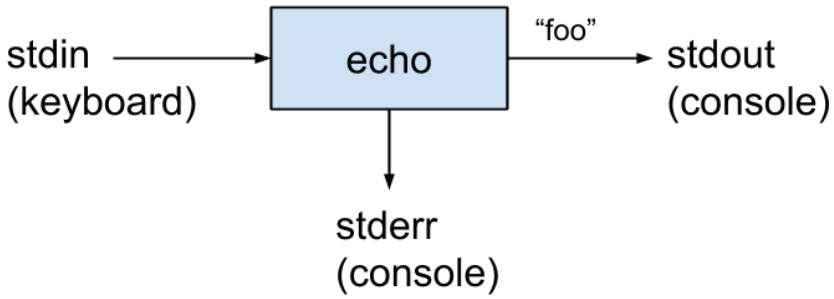
We can get a list of (and the statuses of) backgrounded processes by typing "**jobs**".
So if you have multiple backgrounded processes running, you can get a list of them with "jobs" and then refer to a specific process with "%2" (or whatever the number of the process listed in jobs is). i.e. **kill %1**

Now, how would we modify our earlier program (fork_3.c) to "background" the program it runs (i.e. return immediately without waiting for the program)?

3. stdin, stdout, stderr, and piping/redirection

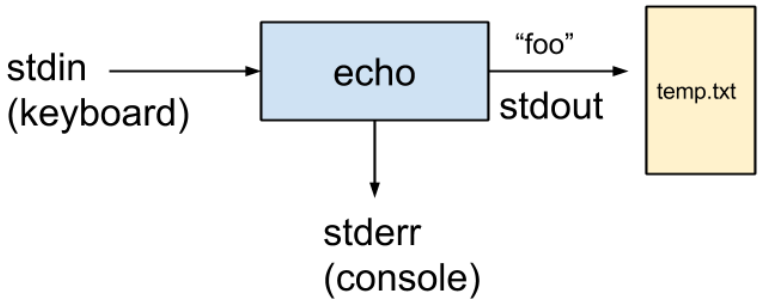
stdin: standard input, a stream for input to a program (e.g. when the bomb prompted you for input)
stdout: standard output, a stream for output (where printf goes)
stderr: standard error, another stream for output, usually used for debugging information and errors.

For instance if we run **echo foo**, we get the following diagram:



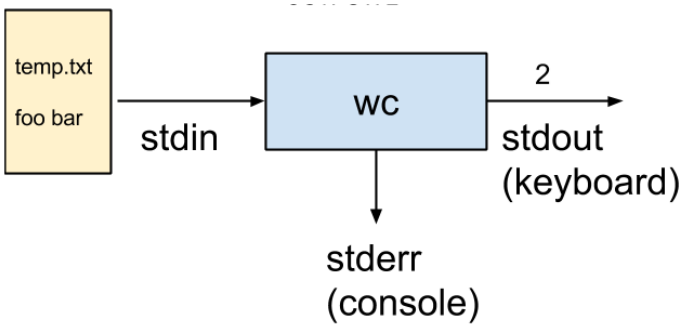
Note that "foo" is a command line argument, *not* from stdin, and echo only prints to stdout (not stderr).

Let's now run the command **echo foo > temp.txt**. ">" redirects stdout to a file. So now what would have been printed to stdout will be written to temp.txt:



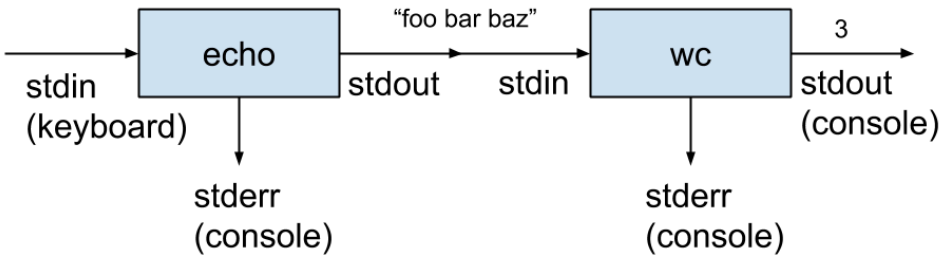
Note that “>” will overwrite the previous contents of temp.txt.
Let’s say we instead want to append to temp.txt. Let’s run
echo “ bar” >> temp.txt
(note the space before bar)
and now temp.txt will consist of “foo bar”.

We can use this same trick with stdin:
Run **wc -w < temp.txt**
and you should get “2” as output:



What’s wc do? Let’s find out! Run **man wc**.
It will tell us that wc (stands for word count) tells us the word, line, character, etc. count of either standard input or a file passed in as an argument. The -w asks specifically for the word count. By using “<”, we are redirecting stdin to be the file temp.txt, so wc is getting the word count of that file (which it “thinks” is stdin).

We can also pipe the output of a program to the input of a different one:
Run **echo “foo bar baz” | wc -w**
We should get output 3 (since there are 3 words). The “|” tells the shell to take the stdout of the command on the left side, and feed in as the stdin to the program on the right side. This looks like so:



OK, then what’s the point of stderr, especially if stderr and stdout both get printed out to our console?

Let's run **fd_1** in the section repo. If we just do **./fd_1** it mixes normal program output with warning/error output. This is annoying and makes it harder to separate the real output from debugging info/warnings/errors.
But luckily, if we look at fd_1.c it's printing its output to stdout, but it prints the warning message to stderr, so we can separate them!
e.g. to store the output in a file and just keep the error messages in the terminal we can run **./fd_1 > out.txt**
If we wanted to store both stdout and stderr, we can do **./fd_1 > out.txt 2> err.txt** and now the output (stdout) will be in the file out.txt and the error messages will be in the file err.txt.

A practical example: if you remember to when we ran **find / -ctime -1**, you may have seen some permissions errors getting printed, even when we backgrounded the process (because backgrounding does not suppress stdout/stderr from being output).
We could instead run **find / -ctime -1 > changed-files.txt 2> /dev/null &**
and now those permissions errors would be ignored (/dev/null is a special file in unix where writing to it does essentially nothing). Or if we did want to know about the errors we'd do
find / -ctime -1 > changed-files.txt 2> errors.txt &

How do we implement things like input/output redirection and pipes?

int dup2(int oldfd, int newfd)

- syscall which makes newfd a copy of oldfd (closing the current newfd if necessary)
- i.e. reading and writing from newfd will read and write from the file referred to by oldfd.

Extending our earlier program, how can we make it take 2 more arguments, an input and an output file, so that it will run the given program but with the first file as standard input and the 2nd file as standard output? (RUN dup2_1.c)

```
int main(int argc, char **argv) {
	if (argc != 4) {
		printf("Usage: %s program-name infile outfile", argv[0]);
		exit(1);
	}
	pid_t pid = fork();
	if (pid == -1) {
		perror("Could not fork!\n");
	} else if (pid == 0) { // child
		int fdin = open(infile, O_RDONLY);
		int fdout = open(outfile, O_WRONLY|O_CREAT);
		dup2(fdin, STDIN_FILENO); // copy fdin fd to stdin fd
		dup2(fdout, STDOUT_FILENO); // copy fdout to stdout fd
		// close now unused fds
		close(fdin);
		close(fdout);
		char *child_argv[] = {argv[1], NULL};
```

```
            execvp(argv[1], child_argv); // does not return on success
            perror("Couldn't execute program!\n");
        } else { // parent
            waitpid(pid, NULL, 0);
            printf("done\n");
        }
}
```

What this is doing: we open the given in and out files, and keep track of their file descriptors. Then we duplicate those file descriptors into the stdin and stdout file descriptors respectively. Now, the stdin file descriptor is actually referring to the infile we opened, rather than terminal input, and stdout refers to our outfile.

Notice that we're doing this in our child process, so it is our child's stdin/stdout that are being changed. execvp() doesn't mess with file descriptors, so whatever program we run ends up with stdin/stdout that are actually going to our input and output files!

What about piping from one program to another?
Let's look at the Unix pipe syscall:
int pipe(int pipefd[2]);

- takes in an (uninitialized) array of 2 ints
- after calling pipe, the first element of the array (pipefd[0]) will be the "read" end of the pipe, and the 2nd element (pipefd[1]) will be the "write" end of the pipe. So anything written to pipefd[1] can then be read from pipefd[0].

Let's see this in action by extending our earlier program once more, to where we pass in a program name and some text which will be made the standard input to the program:
(RUN dup2_2.c)

```
int main(int argc, char **argv) {
    if (argc != 3) {
        printf("Usage: %s program-name text-to-pipe", argv[0]);
        exit(1);
    }
    int pipefd[2];
    pipe(pipefd); // creates the pipe
    pid_t pid = fork();
    if (pid == -1) {
        perror("Could not fork!\n");
    } else if (pid == 0) { // child
        close(pipefd[1]); // close unused write end (for child)
        // make child's stdin the pipe's read end
        dup2(pipefd[0], STDIN_FILENO);
```

```
            close(pipefd[0]);
            char *child_argv[] = {argv[1], NULL};
            execvp(argv[1], child_argv); // does not return on success
            perror("Couldn't execute program!\n");
        } else { // parent
            close(pipefd[0]); // close unused read end (for parent)
            // write the given text to the pipe's write end
            write(pipefd[1], argv[2], strlen(argv[2]));
            close(pipefd[1]);
            waitpid(pid, NULL, 0);
            printf("done\n");
        }
    }
```

And now if the program we pass in reads from stdin, it will get the contents of the 2nd argument! (because we made its stdin the pipe's read end, and we wrote our 2nd argument to the pipe's write end)

QUESTION: Now that we know the basics of how pipes are implemented, how many a's would we expect **./fork_5 | cat** to output?

See **pipe.c** to see how you can pipe the output of one program into another, e.g.

- ./pipe ls wc ===> would perform the equivalent of ===> ls | wc

```
int main(int argc, char **argv) {
  if (argc != 3) {
    printf("Usage: %s first-prog second-prog\n", argv[0]);
    exit(1);
  }

  // This will perform: first-prog | second-prog
  int pipefd[2];
  pipe(pipefd);

  pid_t firstpid = fork();
  if (firstpid == -1) {
```

```
    perror("Could not fork!\n");
    exit(1);
  } else if (firstpid == 0) {
    // first child process
    close(pipefd[0]); // close unused read end for first child

    // make first child's stdout the pipe's write end
    dup2(pipefd[1], STDOUT_FILENO);
    close(pipefd[1]); // no need keep 2 copies of pipe
    char *child_argv[] = {argv[1], NULL};
    execvp(argv[1], child_argv); // does not return on success
    perror("Couldn't execute program!\n");
    exit(1);
  }

  pid_t secondpid = fork();
  if (secondpid == -1) {
    perror("Could not fork!\n");
    exit(1);
  } else if (secondpid == 0) {
    // second child process
    close(pipefd[1]); // close unused write end of second child

    // make second child's stdin the pipe's read end
    dup2(pipefd[0], STDIN_FILENO);
    close(pipefd[0]); // no need keep 2 copies of pipe
    char *child_argv[] = {argv[2], NULL};
    execvp(argv[2], child_argv); // does not return on success
    perror("Couldn't execute program!\n");
    exit(1);
  }

  // back in main parent process
  close(pipefd[0]); // parent does not read or write to pipe
  close(pipefd[1]);
  waitpid(firstpid, NULL, 0);
  waitpid(secondpid, NULL, 0);
  printf("done\n");
}
```