Lecture 4: More multiprocessing: execvp and pipes

Note: Reading these lecture notes is not a substitute for watching the lecture. I frequently go off script, and you are responsible for understanding everything I talk about in lecture unless I specify otherwise.

There are two main topics I hope to cover today: how to start executables from disk, and how to allow for freeform communication between two processes.

More notes on fork/waitpid

Warmup: fork-puzzle

What are the possible outputs of this program?

```
int main(int argc, char *argv[]) {
    pid_t pid1 = fork();
    if (pid1 == 0) {
        pid_t pid2 = fork();
        if (pid2 == 0) {
            printf("watermelon\n");
        }
        printf("pineapple\n");
    }
    printf("apple\n");
    waitpid(pid1, NULL, 0);
    printf("carrot\n");
    return 0;
}
```

Calling waitpid without a specific child PID

You can call waitpid passing `-1` instead of a child's PID, and it will wait for *any* child process to finish (and subsequently return the PID of that process). If there are no child processes

remaining, `waitpid` returns -1 and sets the global variable `errno` to `ECHILD` (to be specific about the "error condition." It can return -1 for other reasons, such as passing an invalid 3rd argument.)

This example creates several processes without keeping track of their PIDs, then calls `waitpid` until the parent has no more child processes that it hasn't already called `waitpid` on:

```
int main(int argc, char *argv[]) {
    for (size_t i = 0; i < 8; i++) {
        pid_t pid = fork();
        if (pid == 0) {
            // Child process
            return 110 + i;
        }
    }

    while (true) {
        int status;
        pid_t pid = waitpid(-1, &status, 0);
        if (pid == -1) break;
        printf("Process %d exited with status %d.\n", pid,
               WEXITSTATUS(status));
    }
    assert(errno == ECHILD);
    return 0;
}
```

This calls `waitpid` a total of 9 times (it returns child PIDs 8 times, then returns -1 to indicate that there are no remaining children).

When `-1` is passed as the first argument, waitpid returns children in a somewhat arbitrary order. If several child processes have exited by the time you call waitpid, it will choose an arbitrary child from that set. Otherwise, if you call waitpid before any child processes have stopped, it will wait for at least one of the running children to exit.

waitpid and scheduling

To be clear, waitpid does ***not*** influence the scheduling of processes. Calling waitpid on a

process does not tell the OS, “hey, I am waiting on this process, so please give it higher priority.” It simply blocks the parent process until the specified child process has finished executing.

waitpid summary

- Waits for a PID to exit (or, as we’ll see later, to change state in some other way – to start, stop/pause, or continue). If the specified PID has already exited, then waitpid returns immediately.
- Can *only* be called on a direct child process. Can’t be called on grandchildren, or parent processes, or anything else.
- Once you call waitpid on a process, you can’t call it again.
- In place of a PID, you can pass -1 as the first argument. If one or more child processes have already exited, it will (somewhat arbitrarily) pick one and return immediately, returning the PID of that process. If no child processes have yet exited, it will wait until one of them exits.
- If a process has no children, waitpid returns -1 with errno `ECHILD`.
- You can get information about the child’s execution by passing the address of an integer as the second argument to `waitpid`. Several macros extract information from the resulting bit set. We’ve seen `WIFEXITED(status)` to check whether the program executed to completion and `WEXITSTATUS(status)` to get its return code.

Loading executables: execvp

The `execvp` system call is used to start running an executable from a binary on disk. It completely wipes the virtual address space of the function that calls it, replacing all the segments with new segments from the executable file.

```
execvp(const char *path, char *const argv[]);
```

`path` is the name of the executable we want to run, and `argv` is a NULL-terminated array of strings (which ends up being the `argv` passed to `main` in the new executable!).

`execvp` *never returns*, since the old program gets cannibalized, so there’s nothing to return to.

Implementing `system`

`int system(char* command)` is a standard library function that runs a command and returns

the status code of the command that ran. We're going to implement it (named `mysystem` so as to not conflict with the stdlib definition) in the context of implementing a super basic shell.

The code for the shell:

```
int main(int argc, char *argv[]) {
    while (true) {
        printf("> ");
        char cmd[2048];
        fgets(cmd, sizeof(cmd), stdin);

        // If you press ctrl+d, that closes the stdin file
        if (feof(stdin)) break;

        // fgets doesn't remove the \n from the input
        cmd[strlen(cmd) - 1] = '\0';

        printf("retcode = %d\n", mysystem(cmd));
    }
    return 0;
}
```

Note: This is an example of a read-eval-print loop (REPL). That's a term you may see crop up in various places in software engineering – now you know what it means!

Implementation of `mysystem`:

```
int mysystem(char *command) {
    pid_t pid = fork();
    if (pid == 0) {
        char *argv[] = {command, NULL};
        execvp(argv[0], argv);
        // If we get here, there was an error
        printf("Command not found: %s\n", command);
        exit(0);    // DANGER: what if we did "return 0" here instead?
    }
```

```
    // At this point, we want to wait for the child process to exit, and get
    // its return code. (The child process *is* the executable we ran)
    int status;
    waitpid(pid, &status, 0);
    if (WIFEXITED(status)) {
        return WEXITSTATUS(status);
    } else {
        return -1;
    }
}
```

It turns out this doesn't work for commands like `make clean`, because `execvp` tries to find a binary called `make clean` (including the space) when, in reality, we want to find a binary called `make` and pass it an argument `clean`. As a sort of ugly fix, we can invoke `/bin/sh` to do the tokenization for us:

```
char *argv[] = {"/bin/sh", "-c", command, NULL};
```

This isn't a great solution; our shell is invoking another shell to finally invoke the program we want. You'll implement a much more robust shell in Assignment 3.

File descriptor/file entry/vnode tables revisited

Refresh your memory on the three tables involved with managing file descriptors; I wrote up a summary here.

In addition to supporting regular files, Unix has a lot of "virtual files": resources that are not really files, but which we make look like files in order to use file-related abstractions. For example, there are standard in/out/error files linked to your terminal; if you write to the stdout file, the text appears on your screen. File descriptors 0, 1, and 2 point to stdin, stdout, and stderr files, respectively.

The file descriptor table is cloned on `fork` and preserved across `execvp` boundaries. (On `fork`, reference counts in the file entry table are doubled.) Consequentially, on `fork`, a child process inherits the stdout linked to the terminal, and if it calls `execvp`, the new executable can still write to the terminal.

Pipes

Another type of "virtual file" is a *pipe*. Pipes are one of several mechanisms for interprocess communication that we'll study this quarter. They allow processes to exchange free-form data during process execution.

The `pipe` syscall creates two new "files" that are linked to each other. If you write to one of the file descriptors, you can read what you wrote from the other file descriptor, almost as if the two file descriptors were linked like a cup-and-string phone. This is a minimal example of pipes at work:

```
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);

    write(fds[1], "hello", 6);

    char buffer[6];
    read(fds[0], buffer, 6);
    printf("%s\n", buffer);

    return 0;
}
```

Because the file descriptor table is shared across a fork boundary, we can have one process write, and have the other process read:

```
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);

    pid_t pid = fork();
    if (pid == 0) {
        char buffer[6];
        read(fds[0], buffer, 6);
        printf("%s\n", buffer);
        return 0;
    }

    write(fds[1], "hello", 6);
```

```
    return 0;
}
```

Implementing `subprocess`

Similar to `system`, subprocess launches an executable in a child process. However, instead of waiting for the child process to exit, it returns *immediately*, so that the parent can communicate with the child while it runs. It returns the PID of the child, as well as a file descriptor; if the parent writes to this file descriptor, the child will be able to read that data by reading from stdin.

```
typedef struct {
    pid_t pid;
    int infd;
} subprocess_t;

subprocess_t subprocess(const char *command);

// Example demonstrating how to use subprocess. We start "sort," feed it 4
// words (by writing to the input file descriptor, which is wired to stdin of
// "sort"), then close the input file descriptor (equivalent to pressing ctrl+D
// on the keyboard -- tells "sort" we're done feeding it words).
int main(int argc, char *argv[]) {
    subprocess_t sp = subprocess("/usr/bin/sort");
    const char *words[] = {"pen", "pineapple", "apple", "pen"};
    for (size_t i = 0; i < sizeof(words) / sizeof(words[0]); i++) {
        dprintf(sp.infd, "%s\n", words[i]);
    }
    close(sp.infd);

    int status;
    pid_t pid = waitpid(sp.pid, &status, 0);
    return pid == sp.pid && WIFEXITED(status) ? WEXITSTATUS(status) : -1;
}

subprocess_t subprocess(const char *command) {
    int fds[2];
    pipe(fds);
```

```
    subprocess_t process = { fork(), fds[1] };
    if (process.pid == 0) {
        close(fds[1]);  // The child isn't writing to the pipe, so we can close
                        // the write end
        dup2(fds[0], STDIN_FILENO); // Rewires fd 0 to point to the read end of
                        // the pipe. The read end of the pipe now has 2 file
                        // descriptors pointing to it.
        close(fds[0]);  // Now that STDIN_FILENO points to the read end of the
                        // pipe, we can close this extra file descriptor.

        // Start the target exectuable:
        char *argv[] = {"/bin/sh", "-c", (char *) command, NULL};
        execvp(argv[0], argv);
    }

    // The parent isn't reading from the pipe, so we can close the read end:
    close(fds[0]);

    return process;
}
```