

CSIS0230A Principles of Operating Systems (Class A)

Tutorial 11

Multiplexing input

In this tutorial, we will write a simple shell-like program that executes commands in the background. All output of these commands will be trapped by the program, and are reprinted on the screen with an identifier showing which command makes the output.

1. A shell-like program

You are given a very simple program that works like a shell: it waits until you enter a command into it, and uses fork to execute the command. A pipe is created for the output of the command to be sent back to the program itself. The program then waits until the pipe is closed by the command (presumably meaning that the command completes). After setting up the terminal to non-canonical mode (i.e., don't wait for end of line, and don't process characters like backspace), the program enters its main loop:

```
void cmd_loop() {
    int cmd_count = 1;
    string str;
    cout << cmd_count << "> ";
    for (;;) {
        char ch;
        cin.get(ch);
        if (ch != '\n')
            str += ch;
        else {
            int fd = run_get_pipe(str); // Run the command and get the pipe
            string fdstr;
            while (read(fd, &ch, 1) != 0) // Read the pipe until pipe closed
                if (ch != '\n')
                    fdstr += ch;
                else {
                    cout << cmd_count << ": " << fdstr << endl;
                    fdstr = "";
                }
            close(fd);
            str = "";
            ++cmd_count;
            cout << cmd_count << "> ";
        }
    }
}
```

2. Your task: Running commands in the background

Your challenge is to modify this program (more exactly, this loop) so that all the commands are executed in the background. That means the program will collect and print output of the executing commands while continuing to read and execute new commands you type.

For this to work, your program must be prepared for multiple commands to execute concurrently. Use the *select()* system call to wait for input to become available from any of the pipes, or from the standard input. We suggest using the following array to remember, for each file descriptor, whether a command with that fd is executing, what is the string outputted by that command, and what is the identifier of the command.

```
struct {
    bool used;
    int cmd;
    string str;
} fds[1024];
```

This assumes the file descriptor will never have values above 1024 (or, if you like, the constant `FD_SETSIZE`). The main loop should then have the following structure:

```
// Output prompt
for (;;) {
    // Create a fd_set containing only the fd 0
    for (fd = 1; fd < 1024; ++fd) {
        // if fds[fd] is used, add it to the fd_set
    }
    // call select
    if (input ready from stdin) {
        // Get a character from stdin
        if (got a character other than '\n')
            // add it to str
        else
            // run the command and get the file descriptor
            // modify the fds array to reflect the new descriptor
    }
    for (fd = 1; fd < 1024; ++fd) {
        if (input ready from fd) {
            // Get a character from fd
            if (failed)
                // close fd, modify fds[fd] to reflect this
            else if (got a character other than '\n')
                // add it to fds[fd].str
            else
                // print fds[fd].cmd and fds[fd].str of the fd
                // re-print the command prompt
        }
    }
}
```

Hint:

- Before calling `select`, the `fd_set` represents the fd's to wait for. After calling `select`, the `fd_set` represents the set of ready fd's. I.e., `select` changes the `fd_set`.
- You can test your program with commands like “`sleep 5; echo Hi`”. This allows you to easily make commands which take a long time to produce output.
- Be very careful when you type commands: you cannot backspace over them. Unless, of course, you write a handler for the delete character yourselves.
- There is no way to exit the program, except pressing Control-C.
- You need to find new places to output the prompt.
- Normally, `cout` will buffer output until an end-of-line character is sent, or until you read from `cin`. But after the modification, you will not be reading `cin` most of the time. So you will need to use the `flush` manipulator to force out the prompt.