Pipes, Non-blocking IO and other cool stuff

CS61 Section Notes #10

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-sections.git
```

or, if you already have the repo, just update to the latest code:

```
$ git pull
```

Then, navigate to today's material.

```
$ cd s10
```

A Couple More Thoughts on the Shell

Where do the commands that come with your terminal originate from?

Try typing:
`cd /; ls`

You will notice that (if you're on a 'nix' OS) you are not in the ~/ directory. You are actually in the base directory of the system and you can see the bare bones of your computers operating system. (Look around)

If you are on the appliance, you should see a directory called `bin`. This stands for binary and it includes executable programs. If you type `echo $PATH`, and look carefully, you will see the directory `/bin` between two colons. `The PATH` variable is an environment variable similar to the `HOME (try echo $HOME)` and many others that define the users environment.

When the shell receives a command it searches the PATH environment variable from left to right and executes the first binary with a matching name; therefore, directories earlier in the PATH have precedence.

Try running `which cat` this program will print the location of cat your `PATH` variable is using. If you would like to see all of the places in where you have binaries called cat run `sudo ./find_bin.sh cat`.

Passing Environment Variables, ./ vs . ./

Take a look at the code for environ1.sh and environ2.sh

environ1.sh

```
CLASS=cs61
SCRIPT=environ2.sh
echo “I <3” $CLASS
./$SCRIPT
```

environ2.sh

```
echo $CLASS “is really cool”
```

If you run environ1.sh what will be printed?

Describe 1 way in which you could fix environ1.sh so that environ2 will print what we expect. Why does this work?

Describe a second way in which you make environ1.sh print what we expect.

Explain why the . ./ command may be dangerous?

Exercise: Rendezvous

Here are three system calls that define a new abstraction called a *rendezvous.* In this exercise, we will use the rendezvous abstraction to emulate pipes.

1. **int newrendezvous(void)**: Returns a rendezvous ID that hasn’t been used yet.
2. **int rendezvous(int rid, int data)**: Blocks the calling process P1 until some other process P2 calls **rendezvous()** with the same **rid** (rendezvous ID). Then, both of the system calls return, but P1’s system call returns P2’s **data** and vice versa. Thus, the two processes swap their data. Rendezvous acts pairwise; if three processes call

rendezvous, then two of them will swap values and the third will block, waiting for a fourth.

   a. To write a byte c to a pipe with ID p, call rendezvous(p, c).
   b. To read a byte from a pipe with ID p (and store the result in variable c), call c = rendezvous(p, -2).
   c. If a pipe is closed, the returned data will be -1.

3. **void freezerendezvous(int rid, int freezedata)**: Freezes the rendezvous **rid**. All future calls to **rendezvous(rid, data)** will immediately return **freezedata**.

Here's an example. The two columns represent two processes. Assume they are the only processes using rendezvous ID 0.

```
int result;                          int result;
result = rendezvous(0, 5);           printf("About to rendezvous\n");
                                     result = rendezvous(0, 600);

                /* The processes swap data;   both become runnable */

printf("Process A got %d\n",         printf("Process B got %d\n",
result);                             result);
```

This code will print

```
About to rendezvous
Process B got 5
Process A got 600
```

(the last 2 lines might appear in either order).

How might you implement **pipes** in terms of rendezvous? Try to figure out analogues for the pipe(), close(), read(), and write() system calls (perhaps with different signatures), but only worry about reading and writing 1 character at a time. Can you support all pipe features?

```
int pipe(void) {
}

void close(int rid) {
}

ssize_t read1char(int rid, char* buf) {

}
```

```
ssize_t write1char(int rid, char c) {

}
```

Exercises with pipes and non-blocking IO!

In this series of exercises, we'll get more practice using pipes and learn to use select!

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
                   fd_set *exceptfds, struct timeval *timeout);
```

- nfds is one greater than the maximal file descriptor number from the following three fd_sets.
- file descriptors in readfds will be watched to see if characters become available for reading, file descriptors in writefds will be watched to see if a write will not block, and file descriptors in exceptfds will be watched for exceptions.
- On exit, the sets are modified in place to indicate which file descriptors actually changed status.
- Each of the three file descriptor sets may be specified as NULL if no file descriptors are to be watched for the corresponding class of events.
- The timeout argument specifies the interval that select should block waiting for a file descriptor to be ready
- On return, select returns the number of file descriptors contained in the three returned descriptor sets

```
fd_set x;
FD_ZERO(&x) clears the set
FD_SET( 10, &x) adds the file descriptor 10 to the set
FD_ISSET( 10, &x) checks if the file descriptor 10 is part of the set
```

We've provided template code for each of the following programs in the s10 directory. Fill in the marked sections.

1) Write a program `catafter` that has the following semantics.

- `catafter TIME FILE` waits for TIME seconds, then opens FILE, writes it to the standard output, and exits. If FILE isn't given, it reads from standard input.

```
sleep(atoi(argv[1]));

if agrc == 3
{
        int fd = open(argv[2], O_RDONLY)
        dup2(fd, STDIN_FILENO)
        close(fd)
}

execvp("cat", ["cat", NULL]])
```

2) Write a program `runafter` that has the following semantics

- `runafter TIME COMMAND [ARG...]` runs COMMAND after TIME has elapsed.

3) Write a program `tripipe` that has the following semantics.

- `tripipe a b c -- d e f g -- h i j k l` creates three processes and two pipes. The first process runs the program `a b c`. Its standard output goes to pipe A. The second process runs the program `d e f g`. Its standard output goes to pipe B. The third process runs the program `h i j k l`. Its standard input reads from pipe A, and its file descriptor 3 reads from pipe B. The `tripipe` process exits when `h i j k l` exits and has the same status as `h i j k l`.

```
int main( int argc, char **argv ) {
  char *delim = "--";
  int pipefd1[2];
```

```
pipe(pipefd1);

argv = &argv[1];
int next_delim = get_next_delimiter(argv, delim);
pid_t firstpid = fork();
if (firstpid == -1) {
        perror("Could not fork!\n"); exit(1);
} else if (firstpid == 0 ) {        //first child process
        close(pipefd1[0]); // close unused read end for first child
        dup2(pipefd1[1], STDOUT_FILENO);
        close(pipefd1[1]); //no need to keep 2 copies of pipe
        argv[next_delim] = NULL;
        execvp(argv[0], argv);
}

argv = &argv[next_delim + 1];
int pipefd2[2];
pipe(pipefd2);
next_delim = get_next_delimiter(argv, delim);

pid_t secondpid = fork();
if (secondpid == -1) {
        perror("Could not fork!\n"); exit(1);
} else if (secondpid == 0 ) {  //second child process
        close(pipefd1[0]); close(pipefd1[1]);
        close(pipefd2[0]); // close unused read end for second child
        dup2(pipefd2[1], STDOUT_FILENO);
        close(pipefd2[1]); //no need to keep 2 copies of pipe
        argv[next_delim] = NULL;
        execvp(argv[0], argv);
}

argv = &argv[next_delim + 1];
next_delim = get_next_delimiter(argv, delim);

pid_t thirdpid = fork();
if (thirdpid == -1) {
        perror("Could not fork!\n"); exit(1);
} else if (thirdpid == 0 ) { //third child process
        close(pipefd1[1]); close(pipefd2[1]); //close unused write ends
        dup2(pipefd1[0], STDIN_FILENO);
        close(pipefd1[0]); //no need to keep 2 copies of pipe
        dup2(pipefd2[0], 3);
```

```
            close(pipefd2[0]);
            argv[next_delim] = NULL;
            execvp(argv[0], argv);
    }

    close(pipefd1[0]); close(pipefd1[1]);
    close(pipefd2[0]); close(pipefd2[1]);
    waitpid(thirdpid, NULL, 0);
}

int get_next_delimiter( char **argv, char *delim ) {
    int curr_pos = 0;
    while( argv[curr_pos] != NULL && strcmp(argv[curr_pos], delim) != 0) {
            curr_pos++;
    }
    return curr_pos;
}
```

4) Write a program `printfirst` that has the following semantics.

    - `printfirst` reads from file descriptors 0 [standard input] and 3. It waits until at least one of those file descriptors to have a readable byte. As soon as that happens, it picks a readable file descriptor and closes the other one. It reads everything from the chosen file descriptor and writes the results to the standard output. [This will require `select`.]

      ```
      fd_set readfds;
      ```

```
        FD_SET(STDIN_FILENO, &readfds);
        FD_SET(3, &readfds);

         select(4, readfds, NULL, NULL, NULL);

        if (FD_ISSET(STDIN_FILENO, &readfs)) {
                close(3);
                read_all(STDIN_FILENO);
        } else {
                close(STID_IN_FILENO)
                read_all(3);
        }


void read_all(int fd) {
  char ch;
  while( read( fd, &ch, 1 ) > 0 )
        printf("%c", ch);
}
```

5) Test `tripipe` using `printfirst` and `catafter`.

```
echo “Hello!” > tmp.txt
echo “Goodbye!” > lala.txt
./tripipe ./catafter 5 tmp.txt -- ./runafter 3 cat lala.txt -- ./printfirst
```