Overview

In this project, you will implement a simple shell for Linux. This shell will be called "icsh" or "IC shell". The functionality of this shell will be similar to other popular Linux shells such as bash, csh, zsh, but with a subset of features. Basically, your icsh should have the following functionality:

- Interactive and batch mode
- Support built-in some commands
- Allow the user to execute one or more programs from executable files as either background or foreground jobs
- Provide job-control, including a job list and tools for changing the foreground/background status of currently running jobs and job suspension/continuation/termination.
- Allow for input and output redirection to/from files.

There are a number of things that you need to implement in this project. We organize the objectives into milestones below.

Before you begin

Join Assignment: `https://classroom.github.com/a/WIXYXthJLinkstoanexternalsite.`

To start the project, you should clone the starting repo and it should contain `icsh.c` and a `Makefile`. These two files will be your starting point. Make sure to use

```
make icsh
```

to generate the binary file called: `icsh`. We will ask you to `git tag` the commit for each of the milestones for grading. The tag name will be given in each milestone description below.

The look and feel of icsh should be similar to that of other UNIX shells, such as bash, tcsh, csh, etc. For example, your shell's work loop should produce a prompt, e.g., `icsh $`, accept input from the user, and then produce another prompt.

Messages should be written to the screen as necessary, and the prompt should be delayed when user input shouldn't be accepted, as necessary. Needless to say, your shell should take appropriate action in response to the user's input.

Milestone 1: Interactive command-line interpreter

Git tag: `0.1.0` (please tag the commit for this milestone) Points: 10

As your first milestone, you should implement a command-line interpreter functionality. Basically, when you start your shell with `icsh` with no argument, your shell should go into a prompt as shown below:

```
$ ./icsh
Starting IC shell
icsh $ <waiting for command>
```

You can design your own prompt. In the examples, here we use `icsh $` as our prompt symbol so you know we're in our shell. Optionally, you could write a welcome message when the shell starts.

For this milestone, your shell must support 3 built-in commands:

1. `echo <text>` – the echo command prints a given text (until EOL) back to the console.

```
icsh $ echo hello world
hello world
icsh $
```

2. `!!` – Double-bang; repeat the last command given to the shell.

```
icsh $ echo hello world
hello world
icsh $ !!
echo hello world
hello world
icsh $
```

Note that the command first prints out the last command first before running it. When there is no last command, `!!` just gives you back the prompt.

3. `exit <num>` – this command exits the shell with a given exit code.

```
icsh $ exit 1
bye
$
```

Technically speaking, there are two approaches to do this. You can either return from the main function the exit code, or use a system call `exit()`. The exit code should be in the range of 0-255. If a larger number is given, you must truncate them to fit in 8 bits. Optionally, a goodbye message can be shown on screen.

Important notes:

- When users give any other command, your shell should print out `bad command` and return to a prompt. - When users give an empty command, your shell just give a new prompt.

Milestone 2: Script mode

Git tag: `0.2.0` Points: 10

For the milestone, you will enhance your shell to support script mode. When your shell is executed with an argument filename, your shell should read commands from the given file and run them one by one. For example,

```
## test.sh

echo hello
echo world
!!
exit 5
```

Say we have `test.sh` with the above content in the current directory, you should be able to run it through your shell as follows.

```
$ ./icsh test.sh
hello
world
world
$ echo $?
5
$
```

Note the command `echo $?` in the example above, this command will print out the exit code of the previous command. Since we gave `exit 5` to icsh, we expect the bash shell to report exit code = 5 here.

Milestone 3: Running an external program in the foreground

Git tag: `0.3.0` Points: 10

In this milestone, you will implement the most important functionality of a shell: running another program. Specifically, when user is given a command that doesn't match with any built-in command, it is assumed to be an external command. Your shell must spawn a new process, execute it and wait for the command to complete and resume control of the terminal. Here is an example of a user running `ls` command

```
icsh $ ls
file_a file_b file_c
icsh $
```

An external command could also contain argument list. Your shell must support them.

```
icsh $ ls -la
total 8
drwxr-xr-x 2 tusr 1000 4096 May  2 07:04 .
drwxrwxrwt 1 root  root 4096 May  2 07:04 ..
-rw-r--r-- 1 tusr 1000    0 May  2 07:04 file_a
-rw-r--r-- 1 tusr 1000    0 May  2 07:04 file_b
-rw-r--r-- 1 tusr 1000    0 May  2 07:04 fille_c
icsh $
```

Check out the resources section below for how to create process / execute a program / wait for it to complete.

Milestone 4: Signal Handler

Git tag: `0.4.0` Points: 10

Through an interaction with the terminal driver, certain combinations of keystrokes will generate signals to your shell instead of appearing within stdin. Your shell should respond appropriately to these signals.

1. Control-Z generates a SIGTSTP. This should not cause your shell to be suspended. Instead, it should cause your shell to suspend the processes in the current foreground job. If there is no foreground job, it should have no effect.

2. Control-C generates a SIGINT. This should not kill your shell. Instead it should cause your shell to kill the processes in the current foreground job. If there is no foreground job, it should have no effect.

Support additional build-in commands:

`echo $?` – this prints out the exit status code of the previous command. You may assume that all build-in commands exits with exit code 0

Check out the resources section below and read more about signal handling.

Milestone 5: I/O redirection

Git tag: `0.5.0` Points: 10

A program's execution may be followed by the meta-character < or > which is in turn followed by a file name. In the case of <, the input of the program will be redirected from the specified file name. In the case of >, the output of the program will be redirected to the specified file name. If the output file does not exist, it should be created. If the input file does not exist, this is an error.

For example,

```
icsh $ ls -l > some_file
```

This above command starts a new process, executes ls, but instead of outputting to the STDOUT, the output is redirected to a file.

The goal for this milestone is the support both input redirection and output redirection. You can read about how to do this in the resource section below.

Milestone 6: Background jobs and job control

Git tag: `0.6.0` Points: 50

Each process executed by itself from the command line is called a job. When executing backgrounds jobs, the shell should not wait for the job to finish before prompting, reading, and processing the next command. When a background job finally terminates a message to that effect must be printed, by the shell, to the terminal. This message should be printed as soon as the job terminates.

To start a background job, user will issue a command and followed by & symbol. Here is an example:

```
1: icsh $ sleep 5 &
2: [1] 843
3: icsh $
```

Explanation:

Line 1, user issues a command "sleep 5" to be ran in the background (indicated by trailing `&`). The shell starts a new process, runs the command, prints out the job id and its PID to the console (Line 2) and returns to the prompt immediately (Line 3). This allows user to run another command while the sleep command is ran in the background.

```
icsh $
[1]+  Done                    sleep 5
icsh $
```

When the job is completed, the shell will asynchronously print out the job information stating the job id, Done status, the original command back to notify the user that one of the background has been completed.

As we have more than jobs running at a time, your shell must support the following commands for job control.

1. `jobs` - list current jobs

```
icsh $ sleep 100 &
[1] 855
icsh $ sleep 200 &
[2] 856
icsh $ jobs
[1]-  Running                 sleep 100 &
[2]+  Running                 sleep 200 &
icsh $
```

The jobs command shows a table of uncompleted job information (job ids, process status, commands).

2. `fg %<job`$_id$ `>`: $Bringsthejobidentifiedby < job_id > intotheforeground.Ifthisjobwasprevio$

```
icsh $ sleep 100 &
[1] 862
icsh $ fg %1
sleep 100
```

3. `bg %<job`$_id$ `>`: $Execute the suspended job identified by < job_id > in the background.$

```
icsh $ sleep 100
^Z
[1]+  Stopped                 sleep 100
icsh $ bg %1
[1]+ sleep 100 &
icsh $
```

Important notes:

This milestone is the most complicated part of the project. Start early!

Milestone 7: Extra features, Extra credits

Git tag: 0.7.0 Points: 10

Congratulations for reaching this milestone. We know you're probably having a lot of fun writing your own shell. This part is for you to throw in any extra features for extra credits.

The points will be awarded proportional to how awesome your features are.

Deliverables

We will have a demo session for you to demo your shell. In addition, you must submit a URL to your git repo on Canvas before the project deadline. In your Git repo, you must have:

- A Makefile.

- Source files that compile, by typing make, into an executable of name icsh.
- Tags of the milestones
- Optionally, a file of name README that contains anything you wish to point out to us.

Resources

System Calls

You have probably already heard the term ‘‘System Call.’’ Do you know what it means? As its name implies, a system call is a ‘‘call’’, that is, a transfer of control from one instruction to a distant instruction. A system call is different from a regular procedure call in that the callee is executed in a privileged state, i.e, that the callee is within the operating system.

Because, for security and sanity, calls into the operating system must be carefully controlled, there is a well-defined and limited set of system calls. This restriction is enforced by the hardware through trap vectors: only those OS addresses entered, at boot time, into the trap (interrupt) vector are valid destinations of a system call. Thus, a system call is a call that trespasses a protection boundary in a controlled manner.

Since the process abstraction is maintained by the OS, icsh will need to make calls into the OS in order to control its child processes. These calls are system calls. In UNIX, you can distinguish system calls from user-level library (programmer’s API) calls because system calls appear in section 2 of the ‘‘manual’’, whereas user-level calls appear in section 3 of the ‘‘manual’’. The ‘‘manual’’ is, in UNIX, what you get when you use the ‘‘man’’ command. For example, man fork will get you the ‘‘man page’’ in section 2 of the manual that describes the fork() syscall, and man -s 2 exec will get you the ‘‘man page’’ that describes the family of ‘‘exec’’ syscalls (a syscall, hence -s 2.)

The following UNIX syscalls may prove to be especially useful in your solution to this homework. There are plenty of others, so

you may find ‘‘man’’ and good reference books useful, especially if you are new to system programming.

- pid$_t fork(void)$: $It creates a process that is an almost-exact copy of the calling process; in partic$
  $Loads the executable file path, or a file found through a search path, into the memory associated w$
- void exit(int status): Exits the calling program, destroying the calling process. It returns status as the exit value to the parent, should the parent be interested. The parent receives this exit value through the wait syscall, below. Note that the linker introduces an exit() call at the end of every program, for instance, at the end of a C main procedure, even if the C code doesn't explicitly have one.
- pid$_t wait(int * stat_loc)$: $Returns the exit status of an exited child, if any. Returns error if there$
- int setpgid(pid_t pid, pid_t pgid): Sets the process group ID of the process with ID pid to pgid.
- int dup2 (int filedes, int filedes2): Causes the file descriptor filedes2 to refer to the same file as filedes.
- int pipe(int filedes[2]): Creates a pipe, placing the file descriptors into the supplied array of two filedescriptors.

Process Creation

To create a new process we use the fork() system call. The fork system call actually clones the calling process, with very few differences. The clone has a different process id (PID) and parent process id (PPID). There are some other minor differences, see the man page for details.

The return value of the fork() is the only way that the process can tell if it is the parent or the child (the child is the new one). The fork returns the PID of the child to the parent and 0 to the child. This subtle difference allows the two separate processes to take two different paths, if necessary.

The wait_() family of functions allows a parent process to wait for a child process to complete. You may want to do this when you create a foreground process from your shell.

The following example shows a waitpid():

```
int main(int argc, char *argv[])
{
  int status;
  int pid;
  char *prog_arv[4];

  /* Build argument list */
  prog_argv[0] = "/usr/local/bin/ls";
  prog_argv[1] = "-l";
  prog_argv[2] = "/";
  prog_argv[3] = NULL;

  /*
   * Create a process space for the ls
   */

  if ((pid=fork()) < 0)
  {
    perror ("Fork failed");
    exit(errno);
  }
  if (!pid)
  {
    /* This is the child, so execute the ls */
    execvp (prog_argv[0], prog_argv);
  }

  if (pid)
  {
    /*
     * We're in the parent; let's wait for the
        child to finish
     */
    waitpid (pid, NULL, 0);
  }
}
```

I/O Redirection

To implement I/O redirection, you'll need to use the dup2() function:

```
int dup2(int fildes, int fildes2);
```

> Each process contains a table with one entry for each open file. [...] The following is an example of I/O redirection.

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
  int in;
  int out;
  size_t got;
  char buffer[1024];

  in = open (argv[1], O_RDONLY);
  out = open (argv[2], O_TRUNC | O_CREAT |
     O_WRONLY, 0666);

  if ((in <= 0) || (out <= 0))
  {
    fprintf (stderr, "Couldn't open a file\n");
    exit (errno);
  }

  dup2 (in, 0);
  dup2 (out, 1);

  close (in);
  close (out);

  while (1)
  {
    got = fread (buffer, 1, 1024, stdin);
    if (got <=0) break;
    fwrite (buffer, got, 1, stdout);
  }
}
```

Signals

Signals are the simplest primitive for interprocess communication (IPC). [...] The following is an example of a signal handler:

```
/*
 * This example shows a "signal action function"
 * Send the child various signals and observe
    operation.
 */

void ChildHandler (int sig, siginfo_t *sip, void *
   notused)
{
  int status;

  printf ("The process generating the signal is
     PID: %d\n",
           sip->si_pid);
  fflush (stdout);

  status = 0;
  /* The WNOHANG flag means that if there's no
     news, we don't wait*/
  if (sip->si_pid == waitpid (sip->si_pid, &status
     , WNOHANG))
  {
    /* A SIGCHLD doesn't necessarily mean death -
       a quick check */
    if (WIFEXITED(status)|| WTERMSIG(status))
      printf ("The child is gone\n"); /* dead */
    else
      printf ("Uninteresting\n");    /* alive */
  }
  else
  {
    /* If there's no news, we're probably not
       interested, either */
    printf ("Uninteresting\n");
  }
}

int main()
{
  struct sigaction action;

  action.sa_sigaction = ChildHandler; /* Note use
     of sigaction, not handler */
  sigfillset (&action.sa_mask);
  action.sa_flags = SA_SIGINFO;       /* Note flag
```

Process Groups, Sessions, and Job Control

[...]

Here's an example that illustrates tcsetpgrp() and setpgid():

```
#include <stdio.h>
#include <signal.h>
#include <stddef.h>
#include <sys/wait.h>
#include <sys/ioctl.h>
#include <sys/termios.h>

/* NOTE: This example illustrates tcsetpgrp() and
   setpgid(), but doesnt function correctly
   because SIGTTIN and SIGTTOU arent handled.*/

int main()
{
  int status;
  int cpid;
  int ppid;
  char buf[256];
  sigset_t blocked;

  ppid = getpid();

  if (!(cpid=fork()))
  {
    setpgid(0,0);
    tcsetpgrp (0, getpid());
    execl ("/bin/vi", "vi", NULL);
    exit (-1);
  }

  if (cpid < 0)
    exit(-1);

  setpgid(cpid, cpid);
  tcsetpgrp (0, cpid);

  waitpid (cpid, NULL, 0);

  tcsetpgrp (0, ppid);
```



```
  while (1)
  {
    memset (buf, 0, 256);
    fgets (buf, 256, stdin);
    puts ("ECHO: ");
    puts (buf);
```