# ECE-C353 Systems Programming
# Project 2 – Basic Shell Part 2: Job Control

**Due Thursday, Mar 8th *before* midnight.**

START THIS PROJECT EARLY

READ THIS ENTIRE DOCUMENT
(TWICE)

### The Big Idea

In this assignment you will be building upon the basic shell you developed in Project 1. Specifically, we will be focusing on **Process Groups**, **Signals**, and **Signal Handlers** in order to implement a standard job control system similar to the ones found in most shells. In this context, a *job* is the execution of a single command line entry. For example:

```
$ ls -l
```

is a single job consisting of one process. Likewise,

```
$ ls -l | grep .c
```

is a single job consisting of two processes.

Upon successful completion of this project, your shell will have the following additional functionality:

- You will be able to check the status of jobs with the new built-in `jobs` command
- You will be able to suspend the foreground job by hitting `Ctrl+z`

```
$ sleep 100
^Z[0] + suspended       sleep 100
$ jobs
[0] + stopped   sleep 100
$
```

- You will be able to continue a stopped job in the background using the new `bg` command:

```
$ jobs
[0] + stopped   sleep 100
[1] + stopped   sleep 500
$ bg %1
[1] + continued sleep 500
$ jobs
[0] + stopped   sleep 100
[1] + running   sleep 500
$
```

- You will be able to start a job in the background by ending a command with the ampersand (`&`) character. Doing this causes the shell to display the job number and the involved process IDs separated by spaces:

```
$ frame_grabber cam0 | encode -o awesome_meme.mp4 &
[0] 3626 3627
$ jobs
[0] + running   frame_grabber cam0 | encode -o awesome_meme.mp4 &
$
```

- You will be able to move a background or stopped job to the foreground using the new built-in `fg` command:

```
$ jobs
[0] + running   frame_grabber cam0 | encode -o awesome_meme.mp4 &
$ fg %0
encoding frame 42239282 [OK]
encoding frame 42239283 [OK]
encoding frame 42239284 [OK]
encoding frame 42239285 [OK]
encoding frame 42239286 [OK]^Z
[0] + suspended      frame_grabber cam0 | encode -o awesome_mem.mp4 &
$
```

- You will also be able to kill all processes associated with a job using the new built-in `kill` command:

```
$ jobs
[0] + running   frame_grabber cam0 | encode -o awesome_meme.mp4 &
$ kill %0
[0] + done     frame_grabber cam0 | encode -o awesome_meme.mp4 &
$
```

Please keep in mind that you will most probably need the full time allotted for this assignment. There are many asynchronous things going on between the shell and the jobs it manages, which are being coordinated by various signals. Give yourself enough time to get things wrong, figure out what is happening, and correct your code.

### Using Process Groups

*Process Groups* are, as the name would imply, a group of processes. Each process group has a *process group ID (PGID)*, which is generally chosen to be the same as the PID of the first process placed in the group. This process is sometimes referred to as the *process group leader*, but it has no special significance. Process groups are convenient because they provide a simple means to send a group of related processes the same signal — generally for control purposes (i.e. `SIGTSTP`, `SIGCONT`, etc). More importantly, however, process groups are used to determine which processes can read and write to `stdin` and `stdout`. For example, when you send `^C` (`SIGINT`) or `^Z` (`SIGTSTP`) using the keyboard, the terminal catches the keystroke and **sends the corresponding signal to every process in the foreground process group**. For this reason, interactive shells (bash, zsh, etc), put the processes comprising a job into their own process group. Let's look at an illustration:

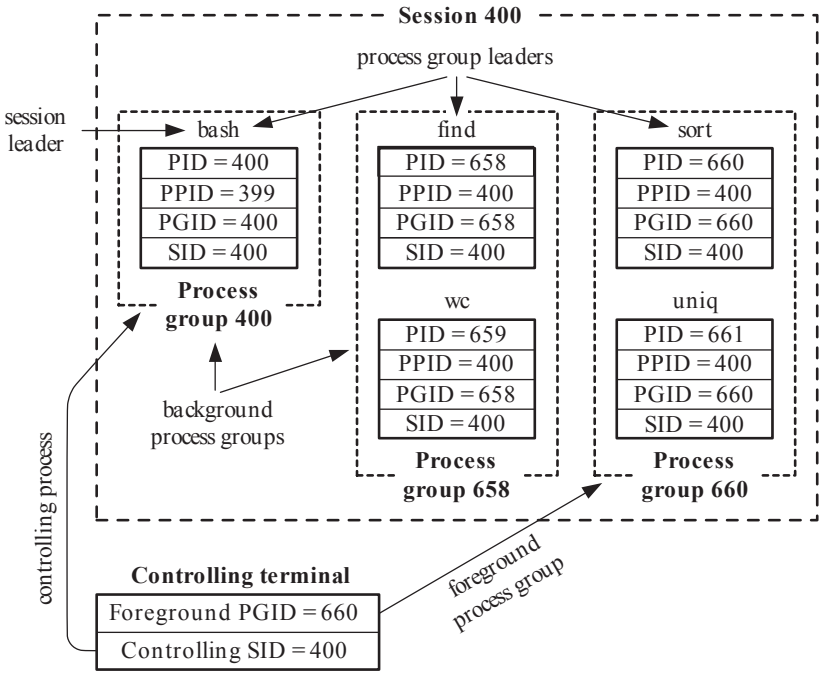


Figure 1: The controlling terminal has a session (set of process groups). The foreground process group receives interactive control signals from the terminal sent by the keyboard. Only the foreground process group may read from `stdin` and write to `stdout`; otherwise, the process will receive the `SIGTTIN` or `SIGTTOU` signal, respectively (which, by default stop the offending process).

Figure 1 shows a situation that can be easily reproduced with the following commands:

```
$ find . -name "*.c" | wc -l &
[1] 658 659
$ sort numbers.txt | uniq
```

It is important to note that the shell (`bash` in this case) places jobs in their own process groups immediately after `fork()`ing. This ensures that the shell is never a member of a child process group. This is important because, otherwise, interrupting the foreground `sort number.txt | uniq` job in our example via `^C` would send `SIGINT` to the `bash` (PID 400) along with PIDs 660 and 661, which would kill not only the foreground process group but our shell as well! By putting jobs into their own process groups, the shell protects itself from receiving such control signals intended for the current foreground process group.

Putting a process in a process group is easily accomplished:

```
pid[t] = fork ();
setpgid (pid[t], pid[0]);

if (pid[t] == 0) {
    /* child stuff */
} else {
    /* parent stuff */
}
```

Obviously, there is nuance to this. First, keep in mind that passing a zero to either argument of `setpgid()` has special meaning—the following are all equivalent:

```
setpgid (0, 0);
setpgid (getpid(), 0);
setpgid (getpid(), getpid());
```

So, what we are doing in our `fork()` example above is to have both the parent and the child attempt to put the child into a new process group. This is necessary because we don't know in what order the scheduler will choose to execute these two processes after the `fork()`. Note, that there is no harm in placing a process into a process group it is already a member of—simply nothing happens. In this way, regardless of which process (parent or child) gets schedule first, the child will get kicked out of the parent's process group as quickly as possible.

### Important `SIGCHLD` Details

Keep in mind that when a child process changes its execution state, the parent process will receive a `SIGCHLD` signal that must be waited on. This may be caused by signals sent to the child such as `SIGTSTP`, `SIGCONT`, `SIGTERM`, `SIGKILL`, etc. When the child receives one of these signals (and changes state), the kernel subsequently sends the parent a `SIGCHLD` signal so that it may act accordingly. The reason the kernel sent the `SIGCHLD` signal can be determined by the parent process by investigating the `status` returned by `waitpid`. This requires the use of the `WIF*` macros detailed in the `waitpid man` page. For example:

```
int status;
...
chld_pid = waitpid (-1, &status, WNOHANG | WUNTRACED | WCONTINUED)
...
if (WIFSTOPPED (status)) {
    /* child received SIGTSTP */
} else if (WIFCONTINUED (status)) {
    /* child received SIGCONT */
} else ...
```

This is an important opportunity for the parent to change which process group is in the foreground—there are other important opportunities as well, such as job creation. This is accomplished by calling `tcsetpgrp()` (read the `man` page). Keep in mind that when a foreground job completes, the shell's process group does not automatically get set to the foreground! It is the shell's responsibility to ensure that happens.

### Other Important Signals: `SIGTTIN` and `SIGTTOU`

We have already discussed the fact that only the foreground process group can read from `stdin` and write to `stdout`. So... what happens when a process not in the foreground process group tries to do one of these two activities? Well, the process will receive `SIGTTIN` if it tries to read from `stdin` and `SIGTTOU` if it tries

to write to `stdout`. The default handler for these signals stops the process. This will obviously be a big issue for your shell, which will potentially attempt to do both of these things (e.g. the prompt!) while a foreground job is running. *Hint:* It may be smart for the shell to check if it is the foreground process using `tcgetpgrp()` when in this scenario. Keep in mind, a process can always `pause()` until the time is right...

### Process Groups are Not Jobs (but they help!)

In implementing your job system, you will find that you will not find everything you need in process groups alone. Job numbers, for example, will have to be tracked by you; the name of the job (i.e. the command entered at the prompt) will be too. How will you know when a job is done? Better keep track of the PIDs of all your child processes so that you can check every time one dies! For simplicity, you only need to be able to support a **maximum of 100** simultaneously managed jobs. I suggest keeping an array of the following `struct` to keep everything managed:

```
typedef enum {
    STOPPED,
    TERM,
    BG,
    FG,
} JobStatus;

typedef struct {
    char* name;
    pid_t* pids;
    unsigned int  npids;
    pid_t pgid;
    JobStatus status;
} Job;
```

I also suggest writing a small family of functions (i.e. an API) for doing common job activities (e.g. adding, removing, killing, etc). How you actually end up doing this, however, is up to you. If you want to support $N$ simultaneously managed jobs, feel free to implement a linked list of `Job` structures (but I think you already have enough to deal with). Just make sure what you design here works.

### Job Control Built-In Commands

You will need to write four (4) built-in commands: `fg`, `bg`, `kill`, and `jobs`. Unlike the `which` command, I recommend running these job control commands inside the shell process so that they have the ability to easily mutate elements of the `Job` array. Except for the `jobs` command, all of these commands can accept a job number as an argument. As you saw in the examples on Page 1, job numbers are decorated with a leading percent (`%`) character to indicate that the number that follows is a job number.

#### Built-in Command: fg

If no arguments are supplied, the following should be printed to the screen and no job states are modified:

```
Usage: fg %<job number>
```

When a properly formatted job number (that exists) is supplied, the corresponding job is moved to the foreground (and continued if necessary). If the supplied job number is not properly formatted or corresponds to a job that does not exist, the following is printed to `stdout`:

```
pssh: invalid job number: [job number]
```

where `[job number]` is the supplied (malformed or invalid) argument.

### Built-in Command: bg

If no arguments are supplied, the following should be printed to the screen and no job states are modified:

```
Usage: bg %<job number>
```

When a properly formatted job number (that exists) is supplied, the corresponding job is continued but not moved to the foreground. If the supplied job number is not properly formatted or corresponds to a job that does not exist, the following is printed to `stdout`:

```
pssh: invalid job number: [job number]
```

where `[job number]` is the supplied (malformed or invalid) argument.

### Built-in Command: kill

If no arguments are supplied, the following should be printed to the screen and no job states are modified:

```
Usage: kill [-s <signal>] <pid> | %<job> ...
```

This means that a mixed list of PIDs and job numbers can be supplied to the kill command, and the desired signal will be sent to each one specified. (The bar simply means the user can supply a pid OR a job number; the ellipsis just means the user can supply as many of these as they like). By default `SIGINT` is sent. If the optional `-s` argument is provided, then the specified signal number is sent instead.

If an invalid job number is supplied, the following is printed to `stdout`:

```
pssh: invalid job number: [job number]
```

where `[job number]` is the supplied job number.

If an invalid PID is supplied, the following is printed to `stdout`:

```
pssh: invalid pid: [pid number]
```

where `[pid number]` is the supplied (malformed or invalid) argument.

### Built-in Command: jobs

This command takes no arguments. All active jobs are simply printed to `stdout` in the following format:

```
[job number] + state     cmdline
```

where `job number` is the job number, `state` is either `stopped` or `running`, and `cmdline` is the commmand line used to start the job. Here is an example output showing all possible states:

```
[0] + stopped    foo | bar | baz
[2] + running    pi_computation
[3] + stopped    top &
```

notice that it is absolutely possible for jobs to terminate in an order different than they were started in (i.e. job 1 is already done)! **The next job that is launched should therefore be job number 1 since it is the lowest available job number.**

### Providing Job Status Updates

In addition to being able to run the built-in `jobs` command to see if background jobs are stopped or running, the user should receive feedback from your shell when:

1. A foreground job is suspended via `^Z` (`SIGTSTP`). For example:

```
$ ./my_program
Running...
^Z[1] + suspended      ./my_program
$ jobs
[0] + running    pi_computation
[1] + stopped    ./my_program
$
```

2. A stopped background job is continued (either via `bg` or by sending it `SIGCONT` directly using `kill`). For example:

```
$ jobs
[0] + running    pi_computation
[1] + stopped    ./my_program
$ bg %1
[1] + continued  ./my_program
$ jobs
[0] + running    pi_computation
[1] + running    ./my_program
```

3. A background job completes or is terminated. For example:

```
$ jobs
[0] + running    pi_computation
[1] + stopped    ./my_program
$ kill -s 9 %0
[0] + done       pi_computation
```

   This "done" status message should <u>not</u> be displayed when a foreground process completes/terminates—only background processes.

*Hint:* These messages to `stdout` are reports generated by the shell when one of its child process groups changes status. Consequently, these messages will be triggered within your shell's `SIGCHLD` signal handler. Be careful though—some jobs will consist of multiple processes! If a job with 5 processes changes state, you don't want the shell to report 5 status changes when the job is continued, for example—you only want it to notify the user once that the entire job has continued.

For the purposes of this exercise, feel free to put the necessary `printf()` statements directly in your signal handler (although, this is bad practice and should never be done in production code because `printf()` is non-reentrant!).

**Grading Rubric & Convenient Feature Checklist**

When assessing the following table prior to submission (notice the fancy check boxes for your benefit) please keep in mind that although partial credit may be possible in extremely select circumstances, it is not very likely. Do not halfway implement a feature in a way that is completely non-functional, give up, and expect partial credit. Your features must actually work to receive credit.

| | **Feature** | **Points** |
|---|---|---|
| ☐ | A process group is created for each new job. The PID of the first process in the job is used as the PGID. | 10 |
| ☐ | A new job intended to run in the foreground is made the foreground process group. | 10 |
| ☐ | A new job intended to run in the background is correctly setup and launched in the background. Job number and associated PIDs are printed to `stdout` by the shell as specified in the problem description. | 10 |
| ☐ | The shell process does not become stopped by SIGTTOU or SIGTTIN. | 10 |
| ☐ | The shell process properly waits on all child processes comprising jobs (i.e. the shell does not generate a legion of zombie processes). | 10 |
| ☐ | A new background job is given the lowest available job number. | 5 |
| ☐ | Ctrl+Z (`SIGTSTP`) properly suspends the foreground job. The `suspended` message specified in the project description is printed to `stdout` by the shell informing the user accordingly. The shell itself is not suspended and is moved to the foreground. | 5 |
| ☐ | Ctrl+C (`SIGINT`) properly terminates the foreground job. The `done` message specified in the project description is NOT printed to `stdout` by the shell. The shell itself is not terminated and is moved to the foreground. | 5 |
| ☐ | Completed/terminated background jobs result in the shell printing the `done` message specified in the project description to `stdout`. The job is appropriately removed from the job management system. | 5 |
| ☐ | A job continued via either `bg` or `SIGCONT` properly resumes running in the background and the `continued` message specified in the project description is printed to `stdout` by the shell. | 5 |
| ☐ | The `fg` built-in command functions as specified in the project description. | 5 |
| ☐ | The `bg` built-in command functions as specified in the project description. | 5 |
| ☐ | The `jobs` built-in command functions as specified in the project description. | 5 |
| ☐ | The `kill` built-in command can send signals to specified PIDs. | 5 |
| ☐ | The `kill` built-in command can send signals to an entire process group associated with a specified job number. | 5 |
| ☐ | The `kill` built-in command can send a user specified signal number as specified by the optional `-s` parameter. | 5 |

### A Few Final Tips

- When calling `tcsetpgrp()`, you may find that the kernel is sending your shell `SIGTTOU` for apparently no reason. This is generally handled by temporarily ignoring the `SIGTTOU` signal very briefly while performing the call. For example:

```
void (*old)(int);

old = signal (SIGTTOU, SIG_IGN);
tcsetpgrp (STDIN_FILENO, pgid);
tcsetpgrp (STDOUT_FILENO, pgid);
signal (SIGTTOU, old);
```

  Do NOT ignore `SIGTTOU` all the time, though—this will yield undesirable behavior.

- Be careful not to accidentally close the `STDIN_FILENO` or `STDOUT_FILENO` file descriptors (i.e. 0 or 1) when setting up a job. This is a great way to terminate your process prematurely.

- The `kill()` function provided by `signal.h` has the in built ability to send a signal to all processes within a process group. I suggest you read the `man` page:

```
$ man 2 kill
```

- The following man pages also make for good reading and could prove to be extremely useful references:

```
$ man 2 waitpid
$ man 2 signal
$ man 2 setpgid
$ man 3 tcgetpgrp
$ man 2 pause
```

### Submitting Your Project

Again, you will be building upon the codebase you developed for `pssh` in Project 1. Once you have implemented all of the required features described in this document submit your code by doing the following:

- Run `make clean` in your source directory. We must be able to build your shell from source and we don't want your precompiled executables or intermediate object files. **If your code does not at the very least compile, you will receive a zero.**

- Zip up your code:

```
jdoe@thanos:~/src$ ls -F
pssh/   pssh_v2/
jdoe@thanos:~/src$ zip -R jd619_pssh_v2.zip pssh_v2/*
```

- Name your zip file `abc123_pssh_v2.zip`, where abc123 is your Drexel ID.

- Upload your zip file using the Blackboard Learn submission link found on the course website.

**Failure to follow these simple steps will result in your project not being graded.**

Now, have fun!

**Due Thursday, Mar 8th *before* midnight.**