15-395: Lab # 4
Implementing a Job Control Shell

15-395: Lab # 4
Implementing a Job Control Shell

Times of Interest:

- Class on Tuesday, October 2, 2007 – Assignment distributed
- 11:59PM, Thursday, October 25, 2007 – Assignment submission deadline

Educational Objectives:

For many people this project will be practice and/or a warm-up. For others, it will be a learning exercise. Regardless of your background, by the end of this project, we hope that you will comfortably and confidently be able to do the following:

- Develop clear, readable, well-documented and well-designed programs in the C Programming Language.
- Develop software in the Andrew/Unix using tools such as gcc, gdb, and make.
- Locate and interpreting "man pages" applicable to application-level system programming.
- Use the POSIX/Solaris API to system functions to manage process and sessions as well as use signals and pipes for inter-process communication.
- Understanding how synchronization might become problematic in light of concurrency.
- Understand how to communicate and cooperate with a project partner.

Project Overview

In this project you are asked to implement a simple command-interpreter, a.k.a. "shell," for Unix. The shell that you will implement, known as xsh, should be similar to popular shells such as bash, csh, tcsh, zsh, &c, but it is not required to implement as many features as these commercial-grade products.

Although we don't require all of the "bells and whistles" that are incorporated into commercial-grade products, xsh should have much of the important functionality:

- Allow the user to execute one or more programs, from executable files on the file-system, as 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 the piping of several tasks as well as input and output redirection.

With respect to the other programming projects, past, present, and future, this may well be a small project, but we want you to approach it as if it is a bigger and more complex project so that you are prepared for the big ones. Specifically, we'd like you to do the following:

- Use the make utility to build your project
- Use a debugger instead of print-and-hunt debugging whenever practical.
- Produce clean, well-documented, and well-designed solutions.

Specification

Form

Your solution should be an application program invoked without command-line parameters or configuration files, &c. If you want to be fancy and support for a resource file similar to those used with commercial-grade shells, e.g. .cshrc, you're a welcome to do this. But, like csh, your shell should function correctly in absence of this file.

Look-and-Feel

The look and feel of xsh should be similar to that of other UNIX shells, such as bash, tcsh, csh, &c. For example, your shell's work loop should produce a prompt, e.g., xsh>, 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.

Internal Commands vs. External Programs

In most cases, the user's input will be a command to execute programs stored within a file system. We'll call these *external programs.* Your shell should allow these programs to execute with stdin and/or stdout reassigned to a file. It should allow programs I/O to be chained together using pipes. For our purposes, a collection of piped processes or a single 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. The syntax for doing this will be described in the section of this document describing the shell's parser.

Your parser should also support several *internal commands* these commands, if issued by the user, should direct the shell to take a particular action itself instead of directing it to execute other programs. The details of this are discussed in the section describing internal commands.

Foreground vs. Background Jobs

Your shell should be capable of executing both foreground and background jobs. Whereas your shell should wait for foreground jobs to complete before continuing, it should immediately continue, prompt the user, &c, after placing a job into the background.

Your shell should print a message *immediately* when a background job terminates. This is a different behavior than most commercial shells. But, it ensure that you handle signals in a certain way, so we are requiring this.

Command lines

When the user responds to a prompt, what they type composes a *command line string*. Your shell should store each command-line string, until the job is finished executing. This includes both background and suspended jobs.

The shell should assign each command-line string a non-negative integer identifier. The data structure used to store the jobs should allow access to each element using this identifier. Once the actions directed by a command-line string are completed, your shell should remove it from the data structure. Identifiers can be recycled if you choose. Please note that this data structure should keep track of whole command line strings, not just the names of the individual tasks that may compose them.

You should not keep track of command line strings that contain internal commands, since, by their nature, they will complete before this information could become useful.

Internal Commands

The following are the internal commands. If an internal command is submitted by the user, the shell should take the described actions itself.

- **`exit`**: Kill all child processes and exit xsh with a meaningful return code.
- **`jobs`**: Print out the command line strings for jobs that are currently executing in the background and jobs that are currently suspended, as well as the identifier associated with each command line string. You may format the output of this command in way that is convenient to the user. Please remember that jobs itself is an internal command and consequently should not appear in the output.
- **`echo $?`**: Prints the exit status of the most recent foreground child process to have exited. Return 0 if no such child has existed.
- **`fg %<int>`**: Brings the job identified by <int> into the foreground. If this job was previously stopped, it should now be running. Your shell should wait for a foregound child to terminate before returning a command prompt or taking any other action.
- **`bg %<int>`**: Execute the suspended job identified by <int> in the background.

- Internal commands can take advantage of piped I/O, execute in the background, &c, as appropriate.

Special Keystrokes

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.

- Control-Z generates a SIGSTOP. 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.
- 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.

Parsing User Input – Overview, Delimiters and Special Characters

Your parser should be generated using (f)lex and yacc/bison. It should be capable of accepting input from the user as described in this section. It should also detect improper input from the user. If the user enters something improper, your shell should produce a meaningful error message.

Just like commercial-grade shells, your shell should accept input from the user one line at a time. You should begin parsing the users input when he/she hits enter. Empty command lines should be treated as no-ops and yield a new prompt.

Blank-space characters should be treated as delimiters, but your shell should be insensitive to repeated blank spaces. It should also be insensitive to blank spaces at the beginning or end of the command line.

Certain characters, known as *meta-characters*, have special meanings within the context of user input. These characters include &, |, <, and >. Your shell can assume that these meta-characters cannot occur inside strings representing programs, arguments, or files. Instead they are reserved for use by the shell. The purpose of meta-characters is discussed later in this section.

Parsing User Input – Internal Commands

If the command line matches the format of an internal command as described earlier in this document, it should be accepted as an internal command. If not, it should be considered to specify the execution of external programs, or an error, as appropriate.

Parsing User Input – Executing A Single Program

The execution of a program is specified by a sequence of delimited strings. The first of these is the name of the executable file that contains the desired program (modulo a search path as explained in the `execvp` man page, see `man -s 2 execvp`) and the others are arguments passed to the program. The command is an error if the executable file named by the first string does not exist, or is not an executable.

Paring User Input – I/O Redirection

A program's execution specified as above 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.

Parsing User Input – Pipes and Cooperating Programs

Several program invocations can be present in a single command line, when separated by the shell meta-character ``|''. In this case, the shell should fork all of them, chaining their outputs and inputs using pipes appropriately. For instance, the command line

```
progA argA1 argA2 < infile | progB argB1 > outfile
```

should fork `progA` and `progB`, make the input for `progA` come from file `infile`, the output from `progA` go to the input of `progB`, and the output of `progB` go to the file `outfile`. This should be accomplished using a pipe IPC primative.

A command line with one or more ``pipes'' is an error if any of its component program invocations is an error. A command line with ``pipes'' is an error if the input of any but the first command is redirected, or if the output of any but the last command is redirected. A job consisting of piped processes is not considered to have completed until all of its component processes have completed.

Parsing User Input – Background Jobs

The user can specify that a job should be executed in the background by ending the command line with the meta-character &. If this is the case, all program invocations required by the command line are to be carried out in the background.

Parsing User Input – A Grammar

The grammar below provides a more formal description of the syntax governing user input. It is the same one you saw – and converted into a yacc-able form – for your last assignment. This grammar doesn't include the special keystrokes, because they won't show up in stdin as user input and should be handled separately.

A `CommandLine` is legal input provided by the user, as a direction to the shell, in response to the prompt. The grammar assumes that the existence of a lexical analyzer that considers blank-space to be a delimiter, recognizes the meta-characters as tokens, &c.

```
CommandLine    :=  NULL
                   FgCommandLine
                   FgCommandLine &

FgCommandLine  :=  SimpleCommand
                   FirstCommand MidCommand LastCommand

SimpleCommand  :=  ProgInvocation InputRedirect OutputRedirect

FirstCommand   :=  ProgInvocation InputRedirect

MidCommand     :=  NULL
                   | ProgInvocation MidCommand

LastCommand    :=  | ProgInvocation OutputRedirect

ProgInvocation :=  ExecFile Args

InputRedirect  :=  NULL
                   < STRING

OutputRedirect :=  NULL
                   > STRING

ExecFile       :=  STRING

Args           :=          NULL
                   STRING Args
```

A Suggested Plan Of Attack

1) Read the man pages for `fork`, `exec`, `wait` and `exit`.
2) Write a few small programs to experiment with these commands.
3) Read the man pages for `tcsetgrp()` and `setpgid()`
4) Write some code to experiment with process groups, &c. Pay attention to SIGTTIN & SIGTTOU.
5) Take the parser from your last assignment and flesh it out a bit for this one. I particular, think about the data structures that you'll need to build and the helper functions you'll need to call. Start to stub these out.
6) Using your parser, write a simple shell that can execute single commands.
7) Add support for running programs in the background, but don't worry about printing the message when a background job terminates (asynchronous notification). Add the jobs command while you are doing this – it may prove helpful for debugging.
8) Add input and output redirection
9) Add code to print a message when a background job terminates.
10) Add job control features - implement the behavior of Control-Z (and, if applicable, CONTROL-C), `fg` and `bg`.
11) Add support for pipes.
12) Finish up all of the details
13) Test, test test.
14) Celebrate

Deliverables

- You should electronically submit the following items into your group's submission directory before the project deadline:
    - A `Makefile`.
    - Source files that compile, by typing `make`, into an executable of name `xsh`.
    - Optionally, a file of name `README` that contains anything you wish to point out to us.

- Soon after everyone is part of a group, we will be creating the following directory for you to submit your work:

    /afs/andrew.cmu.edu/course/15/395/handin/lab4/andrewid1-andrewid2

The Project Review

After you submit your project, a member of the staff will review it. He or she will test your code to determine its completeness and correctness. He or she will also examine your source code to understand how you designed and implemented your solution, as well as the reason for any failures during the testing.

The grader will then meet with your group to discuss the project. During this meeting the grader will clear up any questions he or she may have about your project, discuss any problems that were found, and check to make sure that both members of your group seem to be knowledgeable. This is also your opportunity to get answers to any questions you might have. If you run out of time, please schedule a follow-up meeting.

You'll receive mail from your grader to schedule a review soon after you've submitted your project. Feel free to e-mail ahead with specific questions, things you'd like to discuss, or to request more time. We're here to help!

It is important to understand that your grade won't be determined until after this meeting. In order to standardize grading, projects are graded at a meeting when the entire staff is present.

Matchmaking

If you are having difficulties finding a partner, please send me (gkesden+@cs.cmu.edu) mail and I will try to play the part of a matchmaker. Please do try to find a partner before you do this.

Environment

Whereas you can do this assignment on any UNIX, it must run on the Andrew UNIX machines for your demo.

Although you can solve this assignment in your choice of languages, it would probably be more difficult in anything other than C, or perhaps C++. For future projects you'll almost certainly have to use C or C++.

Some Useful Information (Some of which is a review)

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, xsh 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 project. 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 particular, after a successful return from `fork()`, both parent and child processes are executing the same program. The two processes can be distinguished by the return value from `fork()`.
- **int execvp(const char * file, char * const argv[]):** Loads the executable file `path`, or a file found through a search path, into the memory associated with the calling process, and starts executing the program therein. If successful, it obliterates whatever program is currently running in the calling process. There are several other, similar forms of exec.
- **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 `exit`ed child, if any. Returns error if there are no children running. Blocks the calling process until a child exits if there are children but they are all currently running.
- **pid_t waitpid(pid_t pid, int *stat_loc, int options):** Similar to `wait()` but allows you to wait for a specific process of group of processes, and allows the specification of flags such as `WNOHANG`.
- **wait3(...), wait4(...):** Similar to `wait()` but allow different combinations of parameters and flags.
- **int tcsetpgrp(int fildes, pid_t pgid_id):** Sets the foreground process group id to be the foreground group associated with the controlling terminal. The controlling terminal is usually associated with stdin, stdout, and stderr (file descriptors 0, 1, and 2)
- **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 form your shell.

It is important to note that the wait_() family of functions returns any time the child changes status -- not just when it rolls over or exits. Many status changes you may want to ignore. You may also want to take a look at some of the flags in the man page for waitpid(), you may find WNOHANG, and others helpful. (WNOHANG makes the wait non-blocking, if there's no news -- it just lets you collect information, if available)

The following example shows a waitpid(). It waits for a specific child. wait() will wait for any child. There are several other flavors. We'll discuss more about what the execve() within the child does shortly.

```
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);
  }
}
```

It is important for your shells to wait for the children that they create. This can either be done in a blocking fashion for foreground processes, or in a non-blocking fashion (WNOHANG) when the child signals. Although many of the resources composing a process are freed when it dies, the *process control block(PCB)*, or at least some of its information, is not. The PCB contains status information that the parent can collect via wait_(). A process that is in this state is called *defunct*. After the wait_(), the PCB is freed. If the parent dies before the child, the child is reparented to the init() process which will perform a wait_() for any such process, allowing the PCB to be freed. Orphan process that are waiting for init to clean them up are called *zombies*.

What If I Don't Want A Clone?

The exec_() family of calls allows a process to substitute another program for itself. Typically a program will call fork() to generate a duplicate copy of itself and the child will call an exec_() function to start another process.

There are several different flavors of exec_(). They all boil down to the same call within the kernel. One parameterization may be more or less convenient from time-to-time.

An exec'd process isn't completely different from the calling process. It does inherit some things, PPID, GID, and signal mask, but not signal handlers. Please see the man page for the details.

The exec_() functions do not return (a new process is now in charge). At least it is fair to say tat if they do return, something bad has happened.

The previous example code also illustrates execvp().

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. This table contains some information about the state of the open file, such as the current offset into the file (the location where the next operation will occur). It also contains a pointer to the system-wide open file table.

This table contains exactly one entry for each open file in the system. If multiple processes have the same file open, the corresponding entry in each process's file descriptor table will point to the same entry in the system-wide open file table. This table contains some information about the file, including a count of how many processes currently have it open. It also contains a pointer to the file's inode, the data structure that associates a file with its physical storage on disk. We'll talk more about this when we get to file systems.

It is also important to realize that many non-files use the same interface, although they operate differently under the hood. For example, in many ways, terminals can be manipulated as if they were files. By default the first three entries in each process's open file table are open and reference the terminal: stdin (0), stdout(1), and stderr(2).

To perform I/O redirection, we open a file and then copy this file's file descriptor entry over either standard in or standard out (or standard error). If we need to restore the original entry later, we need to save it in another entry in the table.

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). We'll talk more about these tools later in the semester.

Signals allow one process to communicate the occurance of an event to another process. The number of the signal indicates which event occured. No other information can be communicated via signals.

But signals will be very important in this project. They will indicate changes in the state of a child background process -- such as its termination, and other important events....that its time for a process to sleep, for example.

When a process receives a signal, it can take an action. Many signals have a default action. For example, certain signals, by default, cause core dumps, or process's to suspend themselves.

We can also specify how we want our process to handle a particular signal (Except for KILL, which isn't really a signal, although it looks like one to the programmer). We do this by specifying a signal handler.

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");
    }
}
```

(cont)

```
(from previous page)

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,otherwise NULL in
function*/

  sigaction (SIGCHLD, &action, NULL);

  fork();

  while (1)
  {
    printf ("PID: %d\n", getpid());
    sleep(1);
  }
}
```

Pipes

Pipes are a more sophisticated IPC tool. They allow for a one-way flow of data from one process to another. (Okay -- SYSVR4 pipes can be bidirectional, but we'll stick to Posix pipes for this discussion).

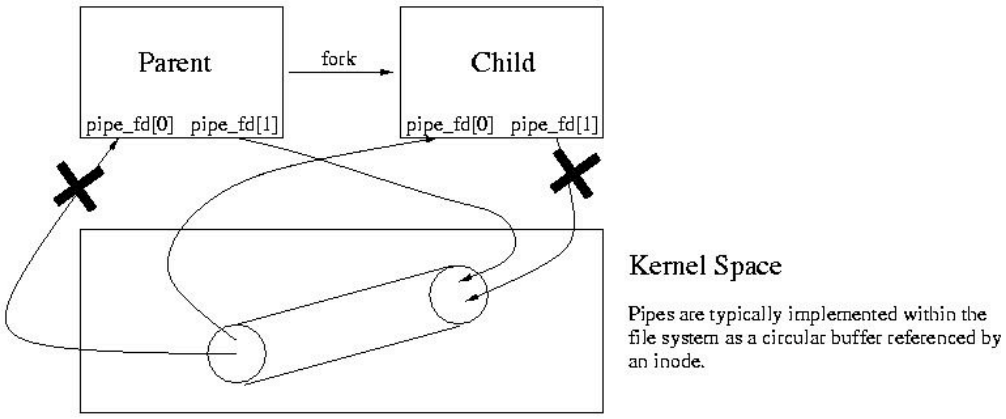
We'll talk more about pipes later in the semester, but here's the basic idea. A pipe is basically a circular buffer that hides in the file system. We use it in a producer-consumer fashion. One process writes to the pipe, and blocks if the buffer becomes full. Another process reads from the pipe and blcoks if it becomes empty. A read will fail if the producer closes the pipe or dies. And a write will fail if the consumer closes the pipe or dies.

Here's how it works. We create a pipe in the parent process using the pipe() system call, by passing it an array of two file descriptors: pfd[0] and pfd[1].

1. Much like file descriptor 0, we will use pfd[0] for input. And we will use pfd[1] for output.
2. We fork and create a child.
3. Now the child and the parent both share the pipe file descriptors. Each will close one side of the pipe (which side depends on whether they will be reading or writing.

4. Next each process will use dup2 to copy the open pipe file descriptor over stdin or stdout, as appropriate. We then close the pipe file descriptors, since they are no longer needed. (If we will later need to restore stdin, or stdout, they should be saved, as we discussed with redirection).
5. Now the two processes can communicate using the pipe via stdin and stdout.

If we do this in between the time we fork and we exec_(), we can tie processes together using pipes -- even though they are ignorantly communicating using stdin and stdout.



Fork Example: The parent process is tthe producer and the child process is the consumer. Notice that the parent closes the read end of the pipe and the child closes the write end of the pipe. Although the SYS V pipe is bi-directional, the traditional UNIX, Posix, and BSD pipes only allow data to flow from the pipe_fd[1] to the pipe_fd[0].

Here's a simple example establishing a pipe:

```
int main(int argc, char *argv[])
    {
      int status;
      int pid[2];
      int pipe_fd[2];

      char *prog1_argv[4];
      char *prog2_argv[2];

      /* Build argument list */
      prog1_argv[0] = "/usr/local/bin/ls";
      prog1_argv[1] = "-l";
      prog1_argv[2] = "/";
      prog1_argv[3] = NULL;

      prog2_argv[0] = "/usr/ucb/more";
      prog2_argv[1] = NULL;

(cont)
```

```
(from prior page)

      /* Create the pipe */
      if (pipe(pipe_fd) < 0)
      {
        perror ("pipe failed");
        exit (errno);
      }

      /* Create a process space for the ls */
      if ((pid[0]=fork()) < 0)
      {
        perror ("Fork failed");
        exit(errno);
      }

      if (!pid[0])
      {
          /*
           * Set stdout to pipe
           */
          close (pipe_fd[0]);
          dup2 (pipe_fd[1], 1);
          close (pipe_fd[1]);

        /* Execute the ls */
        execvp (prog1_argv[0], prog1_argv);
      }

      if (pid[0])
      {
        /* We're in the parent */
        /* Create a process space for the more */
        if ((pid[1]=fork()) < 0)
        {
          perror ("Fork failed");
          exit(errno);
        }

        if (!pid[1])
        {
          /* We're in the child */

          /* Set stdin to pipe */
          close (pipe_fd[1]);
          dup2 (pipe_fd[0], 0);
          close (pipe_fd[0]);

          /* Execute the more */
          execvp (prog2_argv[0], prog2_argv);
        }

        /* This is the parent */
        close(pipe_fd[0]);
       close(pipe_fd[1]);

        waitpid (pid[1], &status, 0);
        printf ("Done waiting for more.\n");
      }
  }
```

Process Groups, Sessions, and Job Control

When we log into a system, the operating system allocates a terminal for our *session*. A session is an environment for processes that is (or at least can be) associated with one controlling terminal.

Our shell is placed into the *foreground process group* within this session. A process group is a collection of one process or of related processes -- they are usually related by one or more pipes. At most, one terminal can be associated with a process group. The *foreground* process group is the group within a session that currently has access to the controlling terminal. Since there is only one controlling terminal per session, there can only be one foreground process group.

Processes in the foreground process group have access to the stdin and stdout associated with the terminal. It also means that certain key combinations, cause the terminal driver to send signals to all processes in the foreground process group. In the case of CONTROL-C, SIGINT is sent to each process. In the case of CONTROL-Z, SIGTSTP is sent to each process. These key combinations do not result in character being placed in stdin.

There can also be background process groups. These are process groups that do not currently have access to the sessions controlling terminal. Since they don't have access to the controlling terminal, they can't perform terminal I/O to/from the controlling terminal. If a background process tries to interact with the controlling terminal, it is sent a SIGTTOU or SIGTTIN, as appropriate. By default, these signals act like a SIGTSTP and suspend the process. The parent process (the shell) is notified about this change, much like it would be if the child process received a SIGTSTP, died, &c. It can discover these changes through the status returned by wait(). Your shell will have to handle these changes in its children.

Processes are placed into process groups using the setpgid() function. Process groups are named by the PID of the *group leader*. The group leader is the first process to create a group -- it's PID becomes the GID. The group leader can die and the group can remain.

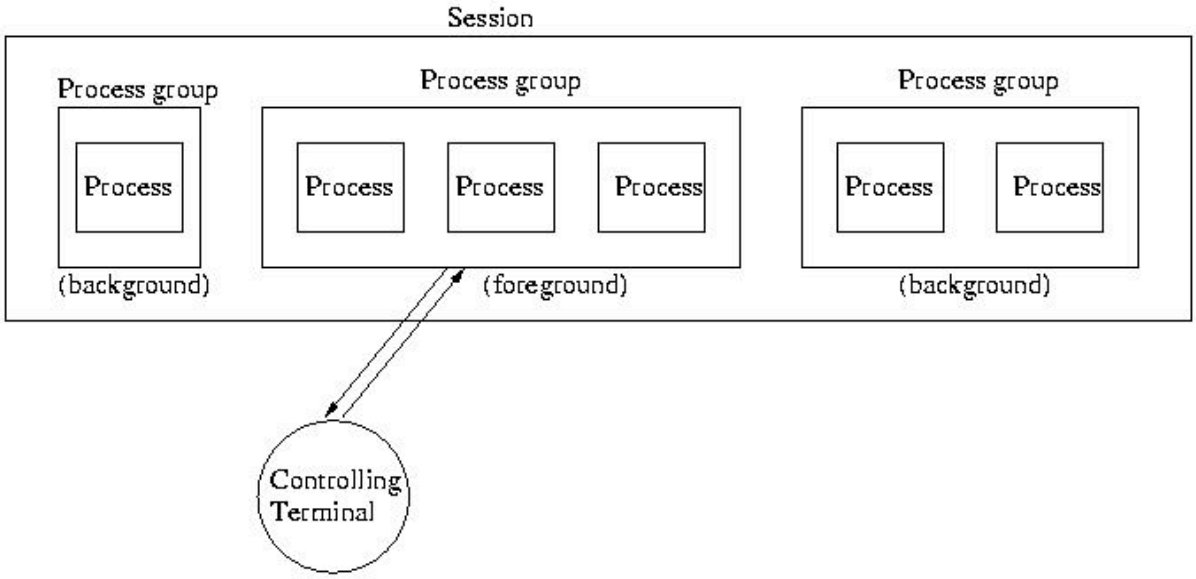
A group becomes the foreground group using the tcsetpgrp() call. This call makes the specified group the foreground group. It can affect itself or any of its children.

If a process forms a new session by calling setsid(), it becomes both a session leader and a group leader. For a new session to interact with a terminal, it must allocate a new one -- you won't need to create a new session or allocate a terminal. Instead, exec ("exec xsh") your xsh from the login shell (csh, sh, bash, csh, etc). This will replace the original shell with your shell, making your shell the only process in the foreground process group.

You will have to create process groups, and manipulate the foreground process group to make sure the right process group is the current foreground process (which could be your shell).

By making the right group the foreground process group, you are not only ensuring that it has a connection to the terminal for stdin, stdout, and stderror, but you are also ensuring that ***every*** process in the foreground group will receive terminal control signals like SIGTSTP.

Please remember that you have a choice in this project – you can take the short-cut. But either way, we'd like you to understand how this works. If you do take the shortcut, you can leave all of the child processes in the same group as the shell and masked SIGTSTP when they are created, so that only the shell can recieve it. The shell can then propogate the equivalent (but unmaskable) SIGSTOP to the appropriate children. The approach falls apart, for example, if you want to try to run your shell form within your shell – but it is good enough for this project.



Here's an example that illustrates tcsetgrp() and setpgrp():

```
#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 tcsetgrp() and setpgrp(), but doesn't
function correctly because SIGTTIN and SIGTTOU aren't 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);
    puts ("\n");
  }
}
```