15-395 Project #4
Frequently Asked Questions
(From past semesters' staff-412@cs archive)

Question:

I assume in this assignment the shell doesn't have to implement the "cd" command, since this is not mentioned?

Answer:

Correct -- you don't have to implement it. It is actually a bit trickier than it looks. But feel free to do it, if you want.

Question:

I asked a few of my friends in the class, and they weren't sure about this question either. Should our shell be able to parse command lines of unlimited length or can we impose a practical limit of lets say 1024 bytes? (the latter enabling us to just be use fgets to obtain input from stdin).

Answer:

Unfortunately, nothing in this world is unlimited. Your shell can certainly impose reasonable limits.

A good practice is to reduce these limits to "configuration constants" so that they can be changed by, for example, only changing a #define or constant-type varible.

Question:

In the code on the course web page under the example for manipulating process groups and terminal groups, I don't understand why *both* the child and the parent have to change the child's process group and the controlling process group

Answer:

The reason that it is set in both places because the OS scheduler may choose to run the processes in any order and they could be interrupted at any number of places (we'll talk about this soon).

By placing the same code in both cases, we are ensuring that it is wasted in one place -- but that our code is correct.

Actually, knowing some things about the internals of Solaris scheduling, we might have been able to eliminate one of them. But this type of programming is not very portable and leads to bugs that can be hard to track down, especially in a multi-platform environment.

So -- defensive programming! Please put the code in both places, even though it is wasteful...just to be safe.

Question:

Hey, I was wondering if we were allowed to assume that all commands must have spaces in them. Like, typing "ls|more" would return an error while "ls | more" would not...

Answer:

Yes, you can assume that at least one blank space will delimit the tokens. But remember that there can be multiple spaces in between the tokens and unnecessary spaces at the beginning and ending of lines.

It is also perfectly legal to implement a parser that will function with or without spaces, if you choose. This, of course, might be more work (without grade-wise, any reward).

Question:

I am hitting another problem -- ctrl-z is not generating SIGTSTP for me. Is there anything I have to do to enable this signal, or enable suspension of child processes?

Answer:

This isn't a problem for most UNIX shells, but some shells will actually disable these signals. bash might do this.

Try typing "stty susp ^z" in the solaris shell before you exec your shell. This should cause your shell to do the appropriate ioctl() to make sure that the terminal driver sends the signal. The terminal driver will keep the setting, even after the fork. If this doesn't work, you could do the ioctl() manually, I suppose -- but this shouldn't be necessary. Unless your shell changes it, I think the default behavior of the terminal driver is to send the SIGTSTP to all processes in the foreground process group upon a CTL-Z.

Question:

I am confused about the tcsetpgrp() call. The first argument is a file descriptor, and in the example code, this is always set to 0, representing stdin. Is is also necessary to call tcsetpgrp with the first argument equal to 1, to change the foreground group for the terminal's stdin?

Answer:

stdin (0), stdout (1), and stderr(2) are three different file descriptors, but they are all tied to the same "controlling terminal" -- in the end, they represent the same(virtual) terminal..

tcsetpgrp() takes, as the first argument, any file descriptor for the controlling terminal that you want to manimpulate -- it doesn't matter which one. In other words, stdin, stdout, and stderr are three different aliases for the same terminal, so it doesn't matter which of the file descriptors you give to tcsetpgrp()

Of course, things would be different if you closed any of these descriptors and/or reopened a different file -- in these case we'd need to make sure that we were using one which did still refer to the controlling terminal. If we calosed all three, we'd have to open "/dev/tty" for read/write/both and then use that file descriptor for tcsetpgrp().

Question:

If I run a job in the background in my shell (ie, do not grant it the terminal), it is still able to generate output without receiving SIGTTOU.

Is this desirable? This seems to be the way that normal shells (such as bash and tcsh) work as well. Or should we be doing some stty magic which turns off this behavior?

Answer:

This might be desirable, but it isn't exactly what we prescribed. There might be one of two things going on:

Possibility #1:

> We've had a few reports of people that are not getting SIGTTOU when a background process tries to write to the terminal. This is because the UNIX shell (like tcsh, csh, &c) has instructed the terminal driver not to deliver this signal.
>
> The easiest way to fix this is to do an "stty tostop" ("stty -tostop" is the opposite), before execing your shell.

Another option would be for your shell to get the current terminal attributes using tcgetattr(...), set the TOSTOP flag with the termios struct, and then to set the terminal with these attributes using tcsetattr(...).

```
struct termios term;

tcgetattr(0,&term); /* get existing terminal attributes */
term.c_lflag |= TOSTOP; /* set the flag to request send of SIGTTOU */
tcsetattr(0,TCSANOW,&term); /* Record the new attributes and make changes NOW */
```

This unmasks the SIGTTOU for background processes, and should allow you to handle the signal.

Possibility #2:

A quick lesson in subtlety...

There are two ways of "doing nothing" when a signal occurs:

1) Mask the signal: By setting the appropriate bit in the process's signal mask, no action will be taken if the signal occurs. If the signal is subsequently unmasked, up to one prior instance of the signal can be discovered.
2) Set the signal's handler to SIG_IGN. This handler does nothing. So, when the signal comes in, the signal is handled -- but nothing happens.

In many respects, these two approaches are equivalent -- most especially in the fact that nothing happens when the signal occurs. But they do differ in two important respects:

1) If the signal is handled by the SIG_IGN handler, a prior occurence cannot be discovered after-the-fact (since it/they was/were handled by the do-nothing routine).
2) When an exec occurs, the newly created process inherits the old process's signal mask, but not the old process's signal handlers. This, in and of itself, is not surprising, since the signal handlers are in a different data structure and likely wouldn't be applicable to a different process. But exec does have a somewhat unusual behavior -- it installs the default signal handler for all signals, _except_ those that were being handled by SIG_IGN. Those signals that were being ignored remain ignored -- the handler remains SIG_IGN.

This is important to you, because many shells, like tcsh, install SIG_IGN handlers for signals including SIGTTOU before exec'ing processes. This means that when your shell is exec'd by tcsh, it will be ignoring (not masking) SIGTTOU.

Originally, we thought this behavior to be erronous on behalf of tcsh & co. But, much to our surprise, this is actually the convention among shell-writers. "Job

control" signals are intentionally ignored before exec'ing, because, in the event that the exec'ed process is a legacy non-job control shell. Shells written before job-control signals existed do not install handlers for them -- they themselves would fall victim to the default action of suspending. This would be a bad thing for the users of these shells.

What does this mean to you? After fork'ing a new process and before exec'ing the new image, reset all of the singal hander's to the default.

Code similar to this might work for you:

```
for (i=0; i<32; i++)
{
  action.sa_handler = SIG_DFL;
  action.sa_flags = 0;
  sigaction (i, &action, NULL);
}
```

Question:

One strange behavior we've been noticing with our shell is that if we try to run a program such as man (or some program with output piped to more), where there are multiple pages of output, only the first page is displayed before we are dumped back to the shell prompt. This is not simply a length issue, as programs such as cat and ls with large outputs display fine (albeit with scrolling, as one would expect). Our question is, what could be causing this?

it seems that the problem I had mentioned in the last email also applies to any program that requires user interactivity; i.e. if I try to zephyr while using ysh it will send a blank message because it doesn't wait for user input for the message text; trying to run pine will cause similar problems with lack of interactivity.

Answer:

These are just blind guesses, please let me know how they fare.

1) You are doing a wait() instead of a waitpid(). The result is that when the first child ends (the "ls") you break out of your wait, put the shell back in the foreground, "more" loses the controlling terminal and is suspended.
2) You are doing a waitpid(), waiting for "more" to end, but it exits with EINTR. waitpid() will normally wait until the specified child changes state. But, this isn't necessarily the case. I believe that waitpid() will return early with an EINTR if a signal arrives for which the calling process has a handler. So, if you have a handler installed for any signal (for example for SIGCHLD), you need to put a while loop around the waitpid() to recall it if it returns EINTR. In most cases, this is the only safe way to use waitpid(), since there are many default handlers installed (such as SIGTSTP) that could interrupt.

Following this course, as soon as the "ls" dies, it sends a SIGCHLD to your shell. Your shell has a handler for SIGCHLD, so it scrubs blocking in waitpid() and executes the handler, and, upon returning to user mode, waitpid() returns EINTR

Question:

I am having trouble with wait() or waitpid(). It is returning values that make no sense to me.

Answer:

Please *always* check the return code of any syscall you make. Any blocking syscall (such as wait() and waitpid()) may return -1 and set errno to EINTR if a signal arrives while the process is in the syscall.

This is because the operating system is in a bind when it gets the signal --- it wants to deliver it right away to your process, but it is halfway through processing your blocking syscall. So rather than having the OS figure out how to interrupt itself, and resume the syscall later, it just gives up on the syscall (returning EINTR), and then enters your signal handler.

Because of this, whenever you have a blocking syscall in a program that uses signals you should include SA_RESTART in the set of sa_flags passed to sigaction(). This forces the interrupted function to restart automatically.

Another option is to use a loop, such as the one below:

```
int rc;
while((rc = wait(...)) == -1) {
   if(errno != EINTR) {
      /* Another error happened. */
      perror("Error");
      exit(8);
   }
}
/* Wait succeeded */
```

Question:

We are trying to debug our piping control function. We are reading in all of the commands, parsing and error checking them just fine. In our checking we obviously check for correct format and to see if the commands are shell commands; however, we cannot possibly check to see if the commands are correctly spelled external commands (i.e. ls, more, fold, ...). Thus, when we send a command to exec it could return -1 if it is a misspelling. We can catch one such command just fine. Our problem comes in when we have an improperly spelled command being piped to another. What happens is the output of that command goes to stdin (since we have redirected the next function's input to stdin). However, the next function is an improper function and exec dies and never reads that input from stdin. Then our next prompt for user input receives all of this input and deems it the next command line. Do you have any idea how to get rid of all this input.

We are thinking about using other file descriptors than stdin to pipe output and input between commands, would this solve the problem?

Answer:

Please remember that you should validate all user input as best as you can -- as you've noticed, this is critical in system programming.

I think the right way to solve this problem is to use "stat()" or "access()" to make sure that the program is executable before you try to exec it. If anything is not exec'able, print an error and throw out the whole command line.

If you are allowing relative paths, you should write a quick loop that walks through the PATH (see getenv()) and checks all possibilities.

Question:

We are trying to debug our piping control function. We are reading in all of the commands, parsing and error checking them just fine. In our checking we obviously check for correct format and to see if the commands are shell commands; however, we cannot possibly check to see if the commands are correctly spelled external commands (i.e. ls, more, fold, ...). Thus, when we send a command to exec it could return -1 if it is a misspelling. We can catch one such command just fine. Our problem comes in when we have an improperly spelled command being piped to another. What happens is the output of that command goes to stdin (since we have redirected the next function's input to stdin). However, the next function is an improper function and exec dies and never reads that input from stdin. Then our next prompt for user input receives all of this input and deems it the next command line. Do you have any idea how to get rid of all this input. We are thinking about using other file descriptors than stdin to pipe output and input between commands, would this solve the problem?

Answer:

Please remember that you should validate all user input as best as you can -- as you've noticed, this is critical in system programming.

I think the right way to solve this problem is to use "stat()" or "access()" to make sure that the program is executable before you try to exec it. If anything is not exec'able, print an error and throw out the whole command line.

If you are allowing relative paths, you should write a quick loop that walks through the PATH (see getenv()) and checks all possibilities.

Question:

I am having trouble with wait() or waitpid(). It is returning values that make no sense to me.

Answer:

I think you are referring to the status set by wait(), not the return value of wait(). Please remember to decode this using the macros in <sys/wait.h>, such as WIFEXITED() and WEXITSTATUS().

Question:

You said that we should reset the signals upon initalizing our shell. Do we reset all of the signals to unblocked and SIG_DFL.

Answer:

Yes, you should loop through and reset all signals to the default hander, SIG_DFL. You should also unmask all of the signals -- execpt the job control signals: SIGTSTP, SIGTTIN, and SIGTTOU. We don't want our shell to react to these, because it is the process handling the job control.

Don't forget to re-enable these signals between the fork() and exec_() in children -- we do want the children to take appropriate action in response to signals from the terminal driver.

You might also want to mask SIGINT, so the shell can't be exited using CTRL-Z.

Question:

Is our shell supposed to kill off its children upon "exit"?

Answer:

Very close. Instead of sending a SIGINT or SIGKILL to their children, commercial shells send a SIGHUP -- the "Hang Up" signal. It is designed to tell the children that the shell is "hainging up" on them and going away. The default handler for this shell will terminate the process, but programs are free to specify other handlers.

It is good to use SIGHUP instead of SIGILL or SIGKILL, because it allows a process, such as a long running job, to outlive the shell and record its results on disk, &c, without ignoring what could be important and explicit instructions to die. (Note: SIGKILL cannot be masked or replaced).

Question:

The siginfo structure passed to my sigaction function seems to be empty. I have set the right flags. What could be wrong.

Answer:

With free software – you get what you pay for. Linux just ain't quite right sometimes. If Andrew Linux is still doing this – you'll have to work around it or switch to Andrew Solaris. Sorry!

Question:

When we do "more file" and then suspend more, our prompt gets all funny. Any ideas?

Answer:

This is a bit of a complicated situation. "more' changes the terminal settings in order to do pagination properly. It normally resets them when it is done -- but it doesn't always do this when it suspends or stops on a signal.

Commercial shells compensate for this by getting the state of the terminal before initiating each process group and restoring them before continuing it later on. The terminal settings from the shell are used as the initial terminal settings for new process.

You can get the terminal settings using tcgetattr() and install them using tcsetattr(). The settings should be recorded before calling to tcsetpgrp() to switch to another group -- the same goes for the restore of the incoming group's settings. It might make sense to store each process's terminal settings in the same structure as its arguments.

Question:

I am trying to deal appropriately with the case in which the user has just suspended his/her job with ^Z. As far as I'm aware, one can handle a sigchld, wait_pid() to collect the status of that process, and then act appropriately. I am presently using WIFEXITED in a similar fashion (and apparently successfully). My question is, will WIFSTOPPED return true if the process has been suspended by ^Z. If not, what would be the best way to ascertain that a job has been suspended in such a manner. Thanks in advance for any help. So you don't have to make a blind guess, a code sample follows:

Answer:

Look at the manpage for waitpid. It says:

WUNTRACED
which means to also return for children which are stopped, and

whose status has not been reported.

thus if you use WNOHANG | WUNTRACED then it should work.

Question:

When we run ls | more & , both ls and more get suspended. Our signal handler is pretty much the same as the one you had up on the website and does a waitpid with WNOHANG and WUNTRACED options. We're not quite sure this is whats supposed to happen. I thought that for ls|more& , ls would exit voluntarily and more would get suspended. Also,we did clear all the masks and reset all the handlers to default before forking, and we also did stty tostop before execing our shell.

Answer:

Congratulations! Everything is working fine. When the terminal drivers sends terminal control signals, it sends them to all processes in the foreground process group -- not just the process that got its attention.

Question:

My shell seems to be doing weird things. For example, child processes seem to ignore terminal-gernerated signals such as SIGTSTP, SIGTTOU, and SIGTTIN. We've tried "stty tostop" and setting TOSTOP using tcsetattr(), but that didn't help. Any ideas?

Answer:

Although signal handlers are not preserved across an exec_(), certain other signal state is preserved. For example, the list of signals to block persists. Furthermore, fork() preserves all of the signal handling settings, including the mask and handlers. As a result, some weirdness from your login shell, e.g. tcsh, can make its way into your shell's child processes.

To remedy this situation, be sure to reinitialize the signal mask using sigprocmask() during your shell's initialization and/or between the fork() and exec_() used to launch new processes. You should also reset the signal handlers to SIG_DFL, because some may have been set as SIG_IGN (ignore) by the login shell. This can be done using a for loop (from 0 to 31) and sigaction().

Question:

> I was just wondering if you knew of any easy error handling techniques. I remember you telling me about some functions that already exist to check to see if a program exists, or could a program take arguments etc.. Also, I was just wondering if you knew where I could find functions so that all I would have to type in is ls rather than /usr/ucb/ls etc..

Answer:

> You might want to take a look at pathfind(). It is a fnction inside of libgen (compile with -lgen). It takes a path specification (such as PATH in your enviornment), the name of a file, and the access mode. It then returns the absolute path to an instance of the file from a directory in the path list you specified that has the mode (read, write, execute, &c) that you wanted.