The C Shell and UNIX Processes

Introduction

A shell is a command interpreter used to manage processes and the environment in which they execute. The most widely used shell is the C Shell (*csh*(1)) or its variants, because it provides a very fine degree of control over the environment and its subprocesses. For that reason, this handout discusses the C Shell syntax and commands along with how to manage UNIX processes.

A shell is simply a process, and any command you run is executed on your behalf by the shell. So, let's start with what a process is.

UNIX Processes

A UNIX process or job is the result of executing a UNIX command. Processes are created by UNIX commands (including the commands that open windows in X), program executions (including *gcc*(1), *mail*(1) and programs you write and compile), and the C-shell command interpreter itself. At any moment a process may be either running or stopped. The UNIX operating system provides many ways to control these processes, such as suspending, resuming and terminating.

Every time you issue a command, the UNIX operating system starts a new process and suspends the current process (the C-shell) until the new process completes. For example, consider compiling a program. When you type

```
gcc program.c
```

you cannot issue other commands in that same window (or to that same shell) until the compilation has completed. The C-shell is waiting for *gcc*(1) to finish before continuing; we say that the compiler is executing or running in the foreground. If we tell the shell to continue to accept new commands even while the compiler is running, we say that the compiler is executing or running in the background. In the sections below we will discuss how to cause jobs to run in the background.

Associated with each process is a unique Process Identification Number, or PID, which is assigned when the process is initiated. When we want to perform an operation on a process we usually refer to it by its PID.

Determining Process Identification Numbers

The command

```
ps -x
```

tells the system to list all your jobs currently running on the machine that you are logged in to:

```
% ps -x
  PID TT STAT  TIME COMMAND
 6799 co IW    0:01 -csh (csh)
 6823 co IW    0:00 /bin/sh /usr/bin/X11/startx
 6829 co IW    0:00 xinit /usr/lib/X11/xinit/xinitrc --
 6830 co S     0:12 X :0
 6836 co I     0:01 twm
 6837 co I     0:01 xclock -geometry 50x50-1+1
 6841 p0 I     0:01 -sh (csh)
 6840 p1 I     0:01 -sh (csh)
 6842 p2 S     0:01 -sh (csh)
 6847 p2 R     0:00 ps -x
```

The meaning of the column titles is as follows:

column	meaning
PID	process identification number
TT	controlling terminal of the process
STAT	state of the job (see below for more information)
TIME	amount of CPU time the process has acquired
COMMAND	the command that the process represents

The state of the job is given by a sequence of four letters, for example, RWNA. We will consider the meaning of the first letter only.

first letter	runnability of the process
R	Runnable processes.
T	Terminated (stopped) processes which can be restarted.
P	Processes waiting for pages to be swapped in or out.
D	Processes in non-interruptible waits; typically short-term waits for (disk or network) I/O.
S	Processes sleeping for less than about 20 seconds.
I	Processes that are idle (sleeping longer than about 20 seconds).
Z	Processes that have terminated but whose parents have not yet been notified (zombie processes).

The letters that you will see most often are R, T, S, I, and Z.

C Shell Variables

The environment in which a subprocess executes has two components: the local environment, which applies only to that subprocess, and the global environment, which applies to all subprocesses. The shell's environment is controlled by environment variables which may be local (and then apply only to that shell process) or global (and apply not only to that shell process but also to all subprocesses).

C Shell distinguishes between the two very simply. To set a local environment variable called **THISVAR** to the value 12345, just say

```
set THISVAR=12345
```

If you run a subprocess, this value will be invisible to the subordinate processes (note that "#" begins a comment that runs to the end of the line; when you try these, don't type these comments):

```
% set THISVAR=12345
% echo $THISVAR
12345
% csh                          # start a subshell
% echo $THISVAR
THISVAR: Undefined variable.
```

On the other hand, if you want to make **THISVAR** global (or, as is sometimes said, make it exportable, or visible to subprocesses, or inherited), use the *setenv* command:

```
setenv THISVAR 12345
```

Note there is no equals sign. Now:

```
% setenv THISVAR 12345
% echo $THISVAR
12345
% csh                          # start a subshell
% echo $THISVAR
12345
```

To clear a shell variable, so it is undefined, say

```
unset THISVAR
```

if it is a local environment variable or

```
unsetenv THISVAR
```

if it is a global environment variable.

One last point: there's no way for a subprocess to alter the environment of a parent (or any ancestor) process. So, if you try to set something in a subshell, for example, it won't affect anything in the parent shell:

```
% setenv THISVAR 12345
% echo $THISVAR
12345
% csh                          # start a subshell
% echo $THISVAR
12345
% setenv THISVAR "hello, world"
% echo $THISVAR               # new value in subshell, but …
hello, world
% exit                         # go back to the parent shell
% echo $THISVAR               # the original value is intact
12345
```

Here are some useful built-in shell variables. For a few, the issue is whether or not they are set; the specific value is unimportant. In this case, just say

```
set VARIABLE
```

or

```
setenv VARIABLE
```

depending on whether you want it to be inherited.

home The user's home directory.

ignoreeof If set (the value is unimportant), you must type `logout` or exit to terminate the shell. If not set, typing ^D will also terminate the shell. This is a godsend to lousy typists like me.

mail Set this to a list of files to watch for new mail; if the first word is an integer *n*, check for mail once every *n* seconds. For example,

```
set mail=(300 /usr/spool/mail/account /usr/msgs)
```

will make the shell look for changes in the file "/usr/spool/mail/account" and in the directory "/usr/msgs", and report that

```
New mail has arrived in file
```

where file is the appropriate file (or directory) name.

path Set this to a list of directories to search for executables. When you give a command that does not contain a '/' character, the shell looks in each directory in this list for an executale file with the name you typed. It checks the directories in the given order, and stops when it has found the first such file.

prompt Set this to what you want the shell to prompt you with. The default is '% '; an exclamation point '!' in the value is replaced by the current command number, which is very useful for using the history mechanism. Thus:

```
% set prompt="\! % "
34 %
```

(You need the backslash to keep the shell from interpreting the '!' as a history character in the set command. Read on!) By custom, if working as the superuser, the prompt includes the sharp sign '#' instead of the percent '%'.

shell The absolute path name of the shell you are executing.

status When a UNIX command or program exits, its success or failure is indicated by its exit status code. If a command is successful, by convention its exit status code is 0; if not, the exit status code is nonzero. This code is stored in this variable, which is reset after the execution of each command. So, if you want to see the exit satus code of a command, do it like this:

```
% pwd
/usr/bin
% echo $status
0
```

time If this is set to *n*, then all commands which take over n seconds of CPU time to execute will have statistics printed. By default, these are (in order):

- Number of seconds of CPU time devoted to the user's process
- Number of seconds of CPU time consumed by the kernel on behalf of the user's process
- Elapsed (wallclock) time for the command
- Total CPU time – U (user) plus S (system) – as a percentage of E (elapsed) time
- Average amount of shared memory used in Kilobytes
- Average amount of unshared data space used in Kilobytes
- Number of block input operations
- Number of block output operations
- Page faults

We'll mention other shell variables as we go along.

Foreground and Background Processes

If we want to execute a command or run a program, but do not want to wait for its completion to be able to issue other commands we specify that it be executed in the background. There are two ways to have a job execute in the background. The first is to specify that it be a background process when we submit it, the second is to tell the shell to make it a background job after it has begun execution. To specify that a job be executed in the background we append an ampersand '&' to the end of the command. For example

```
a.out < inputfile > outputfile &
```

will execute my compiled program while allowing me to submit other commands for execution while it is running. A good example of an instance when we would like to execute a job in the background is when we start a window during a X session. We would like to start the window from an existing window, but we still want to use the original window. We execute the command

```
xterm &
```

and this starts a new window while allowing us to keep using the current window. When you do this, the shell will tell you the PID of the command after the job starts:

```
% xterm &
[1] 14638
```

means that *xterm*(1) has started and its PID is 14638. The ".X11Startup" file that you have on your account contains lines like the following:

```
xclock -geom 144x66-0+0 &
xterm -geometry +1050+70 -fn 9x15 -fg white -bg black &
```

```
xterm -geometry +450+450 -fn 8x13bold -bg white -fg black &
xterm -geometry +1050+370 -fn 9x15 -fg white -bg black &
xload -geom 300x120-0-0 -bg Red - fg White -update 5 -scale 2 &
```

The middle three lines tell the shell that you want three *xterm* processes (with more specific information being given by the options) and that they should be run in the background. The first and last line cause the clock and the load indicator to be displayed, respectively ,and that they should both be run in the background.

If we start a job and we decide that we want to move it to the background there are a number of ways to do it. The simplest way to move a job to the background is to use ^Z (control-Z) in the window that the job is executing, which will suspend the job (not to be confused with ^C which kills the job). The message `Stopped` will appear on the screen. We then resume the job in the background with the 'bg' command. For example, we want to compile the program "bigprog.c" and we know that it will take a long time and we do not want to wait for its completion. We type the command

```
gcc -ansi bigprog.c
```

but we forget to append the ampersand. We type

```
^Z
```

to suspend the job and then type

```
bg %gcc
```

which resumes the job in the background, allowing us to submit other commands while the compilation takes place. The shell will tell us when the background job has completed with a statement like

```
[1]    Done                    gcc -ansi bigprog.c
```

Normally, this message will appear only when the shell is ready to prompt you for another command. If you want it to appear whenever a background job terminates, whether or not the shell is ready for you to type another command, use the shell variable

notify Set this if you want the shell to notify you immediately upon the completion of a job rather than waiting until just before prompting.

A job can be moved into the foreground with the *fg* command. For example, if we wanted to resume the compilation of our program after we suspended it with ^Z we could use

```
fg %gcc
```

which would resume the job in the foreground (we would have to wait for it to complete before issuing other commands).

When we run a job in the background the output will still come out on the screen as if we ran it in the foreground. There are two ways to handle this. Issuing the command

```
stty tostop
```

will cause any background process to block before generating output to the screen; you will be told using the notification mechanism described earlier in this section. You can then move the job into the foreground. As an alternative, you can use redirection.

Redirection

To avoid having the output from our background job and foreground jobs being interspersed we can redirect the output to a file. To send output to a file, use output redirection:

```
command > outputfile
```

If *outputfile* does not exist it will be created. If it does exist, what happens depends on the setting of an environment variable:

noclobber If set and output is redirected to an existing file, the command is not executed and an error message is given. If set and the file does not exist, it will be created. If this variable is not set, the output will be sent to outputfile (erasing it if necessary).

If you append output to a file using

```
command >> outputfile
```

the sense of the **noclobber** variable is reversed; if **noclobber** is set, then you will get an error message if *outputfile* does not exist; but if *outputfile* does exist, you will not get an error and the output will be appended to it. If **noclobber** is not set, *outputfile* will be created if it does not exist.

In any case, if **noclobber** is set and you want the shell to pretend it isn't for one command, append the exclamation point '!' to the redirection:

```
command >! outputfile
```

overwrites outputfile whether or not noclobber is set, and

```
command >>! outputfile
```

will create *outputfile* if it does not exist. Notice there is no space between the '>' and '!'.

Input can also be redirected; for example,

```
wc < inputfile
```

will cause wc to act as though the contents of inputfile were typed at the keyboard.

In addition to redirecting the standard output (called `stdout`) of the background process we must also redirect any error messages to a file. Error messages are sent to `stderr` (standard error), normally `stderr` is sent to the screen so that you see the error messages as they occur. The following syntax is used to redirect the stderr information to the same file that we redirect the standard output to

```
somecommand >& outputfile
```

We can combine it with input redirection, too:

```
somecommand < inputfile >& outputfile
```

causes all output from somecommand to be written to the file outputfile. Note that somecommand could be a UNIX command, “a.out”, etc. This construct also responds to the setting of noclobber, and that can also be overridden by placing an '!' right after the '&', with no intervening spaces.

A final type of redirection is the pipe. A pipe connects the output of one process to be the input of another. For example,

```
date | wc
```

takes the output of the command *date*(1) and feeds it into the standard input of *wc*(1). As another example,

```
ps | head -4
```

prints the first four lines of the process status listing.

Terminating a Process

We can terminate a process using the *kill*(1) command. Let's say that we start a new xterm by executing the command

```
xterm &
```

and later decide to get rid of it. We find out the PID (for example, by using ps –x); call this pid. Then we type

```
kill -9 pid
```

and the window will disappear. The option -9 ensures that job/process will be killed. A simplier way to kill a process that you are running in the foreground in a window is to type ^C in the window that the job is running in. Next time you log in try starting up a new window in the background and then killing it with the commands given above.

The Shell and Filename Substitution

The shell provides a convenient mechanism for naming files called wildcards. These characters are interpreted as pattern matching commands. For example,

```
ls chap*
```

lists all files in the current directory which begin with the letters “chap”. The ‘*’ metacharacter means to match 0 or more ordinary characters.

Other metacharacters are '?', which means match one character; so the pattern

```
a?c
```

would list the files “abc”, “acc”, and “adc”, but not the file “abbc”. The characters '[' and ']' delimit a range, so the pattern

```
a?c[0123]
```

matches the files “abc1”, “acc3”, and “axc1”, but not “abbc1”, “abc4”, or “azc9”. You can use a hyphen '–' to indicate a contiguous range, so the above pattern could also have been written

```
a?c[0-3]
```

Be careful, though; the pattern “[A-z]” matches all the letters and several other characters, too, because the range is over the ASCII character set!

The pattern matching matches file names; if no files matching the pattern exist, you get an error message:

```
% echo ab*ef
No match.
```

To change this behavior, you can use a shell variable:

nonomatch If set, return the filename substitution pattern rather than an error if the pattern is not matched. Note that malformed patterns (for example, leaving a closing ']' out) will still result in errors.

You can also disable these metacharacters by setting another shell variable:

noglob If set, inhibit filename substitution; the metacharacters lose all special meaning. This is most often done in shell scripts or startup files when the character strings involve the metacharacters, but require them to be interpreted as regulatr characters.

If you just want them disabled for one command, you can put a '\' in front:

```
% echo  a*
ab ac ad
% echo a\*
a*
```

Two other metacharacters deserve mention. The character '~' expands either to your login directory (if it is followed by a '/') or to the home directory of the named user, if he or she exists (if the tilde is not followed by a '/'). So,

```
~/.cshrc
```

refers to your “.cshrc” file, and

```
~bishop/.cshrc
```

refers to my “.cshrc” file. Also, the characters '{' and '}' enclose alternate substitutions; for example,

```
% echo ab{c,d,e}f
abcf abdf abef
```

Unlike the pattern matching metacharacters, there need be no files “abcf”, “abdf”, or “abef”.

History Substitution

The shell makes past commands available to you through the history mechanism. You can simply repeat the commands, or extract portions of previous commands. For example:

```
% history                      # show the last few commands
131 date
132 vi write.c
133 gcc -g write.c
134 write root
135 diff write.c oldwrite.c
% !da                          # repeat the last command that
                               # began with a 'da'
Tue Jan 18 11:58:01 PST 1994
% !!                           # repeat the last command
```

```
Tue Jan 18 11:58:30 PST 1994
% !132                          # repeat command number 132
… vi write.c …
% !gcc:s/-g/-O/                 # repeat the last gcc command,
                                # but replace -g with -O
% !?root?                       # repeat last command containing
                                # "root" anywhere (here, 134)
% mail root < !vi:$             # mail root the file last edited
                                # with vi
% ^root^admin^                  # same as !!:s/root/admin/
% mv write.c chat.c
% mv oldwrite.c oldchat.c
% !diff:gs/write/chat/          # diff the new file names; g
                                # means global (everywhere);
                                # default is to replace first
                                # match
% !{diff}old                    # rerun the last command,
                                # appending "old"; without the
                                # curly braces it is read as
                                # !diffold, which you didn't
                                # mean.
```

Four variables control how the history is handled and saved:

histchars Set this to a string, the first character of which indicates history substitution (and by default is '!') and the second of which indicates quick history substitutions (the default is '^').

history Set this to the number of commands you want the shell to remember.

savehist Because environments are not preserved when a process terminates, your history gets lost whenever you log out. Set this variable to the number of commands to save across invocations; that many commands will be saved in the file ".history" in your home directory, and when you start a new shell, those commads will be read in.

verbose Print the command after history substitution is completed. This helps keep down the aggravation, because you can see what the shell thinks you told it to do.

Aliasing

An alias is another way of saying something, and the C Shell offers you the ability to rename your most common commands and options. For example, a very common alias is

```
alias ls ls -F
```

When you type the command *ls*(1), the shell looks at the first word in the command (if it were part of a pipeline, the shell would look at each command in the pipeline) and sees if it is an alias. If so, it replaces the alias with its definition and repeats the process unless the first word of the new command is the same as the first word of the old one (this prevents obvious loops). So, after the above command, the command "ls -a" would be replaced by "ls -a -F" and that executed.

You can use the history mechanism to pull out parts of the command after the aliasing is applied. In this case, the old (pre-alias) command is treated as the previous command. So:

```
% alias lookup grep \!^ /etc/passwd
% lookup root
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:
```

Note the backslash '\' in fromt of the exclamation point '!'; this is needed to prevent the shell from treating !^ as a reference to the previous command, as history substitutions are done before alias substitutions.

Quotation Marks

There are three types of quotation marks. Strings enclosed by single quotes "'" are not interpreted further; this turns off all substitutions except the history substitutions. The resulting string is seen as one entity, so if you ever want to create a file with a blank in the name, do it like this:

```
touch 'this file has 8 words in its name'
```

Strings enclosed in back quotes "`" are treated as shell commands; they are executed, and the string is replaced by the output. In the output, each contiguous sequence of blanks, tabs, and newlines is replaced by a single blank, so (for example):

```
% cat XYZ
hi there
goodbye
% echo `cat XYZ`
hi there goodbye
```

Strings enclosed in double quotation marks '"' have any shell variables replaced by their value, history substitutions done, and any commands in "`" executed as described above; otherwise, the string is left alone. For example,

```
% echo "$user files???d !32"
bishop files???d vi pwd.c
```

Note the filename pattern was not expanded, but that history substitution was done and the shell variable was evaluated.

Startup Files

When you start the C Shell, it begins by reading the file ".cshrc" in your home directory and executing the commands there as though you had typed them at your keyboard. If this is a login shell (that is, if you started the shell by logging in), it will then read the ".login" file in your home directory.

Here is a sample ".cshrc" file:

```
set path=( /bin /usr/{bin,hosts} /usr/*/bin .)
alias term 'tset -s -n \!* > /tmp/t$user; \
      source /tmp/t$user;\
      /bin/rm /tmp/t$user'
set history=50
set noclobber ignoreeof
echo "Welcome to `hostname`, your friendly UNIX host\!"
```

The first line sets the search path; notice the use of file name metacharacters. The second line (which continues onto the third line) sets up an alias to run the program *tset*(1), which resets the terminal modes (use it if you think your screen is messed up; a good clue is if *pico*(1), *pine*(1), or *vi*(1) refuses to work in anything other than open mode). The third line says that the shell is to remember the last 50 commands, the fourth line sets two options discussed earlier, and the fifth line names the host you're on as it greets you.

The ".cshrc" file contains aliases and other things to be executed whenever a C Shell starts. For things like initial terminal setup or setting global environment variables, the ".login" file is more appropriate, becasue it will only be executed once during the session:

```
setenv EXINIT 'set ai'
set nonomatch
eval `tset -s -Q -m network:vt100`
set term=$TERM
setenv term $TERM
unset nonomatch
```

The first line sets an environment variable used by *vi*; whenever *vi* starts, autoindent mode is on. The rest of the file sets the terminal type to vt100 if the user logs in over a network. The **nonomatch** variable prevents pattern expansion (necessary because of the output of *tset*). Then *tset* is run. The next two lines set the terminal type both locally and globally, and then pattern matching is turned back on. (You don't need to understand *tset*, but look in the

manual page, or ask an experienced user, if you want to.)

Acknowledgement

This document used parts of a handout on processes written by Kevin Rich.