Unix Processes

Norman Matloff
Department of Computer Science
University of California at Davis

Contents

1 Unix Processes

A **process** is an instance of running a program. If, for example, three people are running the same program simultaneously, there are three processes there, not just one. In fact, we might have more than one process running even with only person executing the program, because (you will see later) the program can "split into two," making two processes out of one.

Keep in mind that all Unix commands, e.g. **cc** and **mail**, are programs, and thus contribute processes to the system when they are running. If 10 users are running **mail** right now, that will be 10 processes. At any given time, a typical Unix system will have many active processes, some of which were set up when the machine was first powered up.

Every time you issue a command, Unix starts a new process, and suspends the current process (the C-shell) until the new process completes (except in the case of **background processes**, to be discussed later).

Unix identifies every process by a Process Identification Number (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.

Unix is a **timesharing** system, which means that the processes take turns running. Each turn is a called a **timeslice**; on most systems this is set at much less than one second. The reason this turns-taking approach is used is fairness: We don't want a 2-second job to have to wait for a 5-hour job to finish, which is what would happen if a job had the CPU to itself until it completed.[1]

[1]The basic mechanism for setting up the turns is as follows. The machine will have a piece of hardware which sends electrical

2 Determining Information About Current Processes

The ‘ps -x’ command will list all your currently-running jobs. An example is:

```
  PID TT STAT  TIME COMMAND
 6799 co IW    0:01 -csh[rich] (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[rich on xterm] (csh)
 6840 p1 I     0:01 -sh[rich on xterm] (csh)
 6842 p2 S     0:01 -sh[rich on login] (csh)
 6847 p2 R     0:00 ps -x
```

The meaning of the column titles is as follows:

```
PID         process identification number
TT          controlling terminal of the process
STAT        state of the job
TIME        amount of CPU time the process has acquired so far
COMMAND     name of the command that issued the process
```

The TT information gives terminal names, which you can see by typing the **who** command. E.g. we see p2 in the TT column above, which is the terminal listed as ttyp2 in the **who** command.

The state of the job is given by a sequence of four letters, for example, ‘RWNA’. The first of these four is typically one of the following:

first letter runnability of the process R runnable process T stopped process S process sleeping for less than about 20 seconds I processes that are idle (sleeping longer than about 20 seconds)

A state-R process is runnable, i.e. it is be able to make use of a turn given to it, and is waiting for one.

We can put a process in state T, i.e. stop the process, by typing control-z. Suppose, for example, that I am using **ftp** to get some files from some archive site, and I notice a file there called README. I can use the **ftp** ‘get’ command to get the README file, and then type C-z. This will stop (suspend) the **ftp** process, and get me back to the C-shell. At that point I can read the README file, say using **more**, and then reactivate the **ftp** process, by typing ‘fg’ to the shell.

A typical example of an S/I process is one that is waiting for user input. If I am using the **emacs** editor, for example, the process will go to state S when it is waiting for me to type something; if I take more than 20 seconds to decide what to type, the process will be in state I.

signals to the CPU at periodic intervals. These signals force the CPU to stop the program it is running, and jump to another program, which will be the operating system program (OS). The OS can then determine whether the current program’s timeslice is finished, and if so, then give a turn to another program, by jumping to that program. Note the interaction of hardware (the electrical signals, and the CPU’s reaction to them) and software (the OS) here.

3 Foreground/Background Processes

Suppose we want to execute a command but do not want to wait for its completion, i.e. we want to be able to issue other commands in the mean time. We can do this by specifying that the command be executed in the background.

There are two ways to do this. The first is to specify that it be a background process when we submit it, which we can do by appending an ampersand ('&') to the end of the command. For example, suppose we have a very large program, which will take a long time to compile. We could give the command

```
cc bigprog.c &
```

which will execute the compilation while allowing me to submit other commands for execution while the compile is running. The C-shell will let me know what the pid is for this background process (so that I can later track it using **ps**, or kill it), but will also give me my regular prompt, inviting me to submit new commands while the other one is running. [2]

But what about the compiler error messages? We hope we don't have any :-) but if we do have some, we don't want them to be interspersed with the output of other commands we are running while the compile is executing. To avoid this, we redirect the error messages:

```
cc bigprog.c >& errorlist &
```

All error messages will now be sent to the file 'errorlist', which we can view later.[3]

Another good example 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 be able to use the original window. We execute the command

```
xterm &
```

This will start a new window, and allow us to keep using the current window.

The other way to put a job in the background is to stop it, using C-z as described earlier, and then use another command, **bg**, to move the process to the background.

For example, suppose we started our long-running compile,

```
cc bigprog.c
```

but we forget to append the ampersand. We can type control-z to suspend/stop the job, and then type 'bg' to resume the job in the background, allowing us to submit other commands while the compilation takes place. Unix will tell us when the background job has completed, with a statement like

[2] Keep in mind, though, that there is "no free lunch" here. The more processes on the machine, the longer it is between turns for each process, so overall response time goes down.

[3] Though a much better solution to the problem is to use **emacs**, since the error messages will automatically be placed into a special buffer.

```
[1]    Done                  cc bigprog.c
```

By the way, if you log out, whatever background processes you have running at the time will not be killed; they will continue to run.

4 Terminating a Process

We can terminate a process by using the **kill** command. We simply find its pid (say by using **ps**), and then type

```
kill -9 pid
```

If the process is in the foreground, though, the easiest way to kill it is to simply type control-C.