Processes and the Shell

Introduction

This document describes features of UNIX and Linux systems. The examples are taken from the CSIF systems, which run Ubuntu 18.04.2 LTS Linux. The Linux interface is based on the UNIX interface, and much of what is here applies to UNIX systems such as FreeBSD, OpenBSD, and other variants. Throughout, we will refer to these systems as "Linux systems" for brevity. Also, in the examples in this document, what the computer outputs is in `Roman Courier typeface` and what you type is in **`bold Courier typeface`**. When interacting with the shell, a "`%`" at the beginning of the line is a shell prompt.

A *shell* is a command interpreter used to manage processes and the environment in which they execute. The most widely used shells are the C Shell (*csh*(1)) and the Bourne-Again shell (*bash*(1)). Both provide a very fine degree of control over the environment and its subprocesses. This handout discusses the C Shell syntax and commands along with how to manage UNIX processes. The Bourne-Again shell syntax differs somewhat, but the functionality is (essentially) the same.

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.

Processes

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

Every time you issue a command, the Linux operating system starts a new process and suspends the current process (the 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 shell is waiting for *gcc* 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 PIDs

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 TTY      STAT   TIME COMMAND
11954 ?        Ss     0:00 /usr/lib/systemd/systemd --user
11962 ?        S      0:00 (sd-pam)
11969 ?        R      0:00 sshd: bishop@pts/0
11971 pts/0    Ss     0:00 -tcsh
14957 pts/0    S      0:00 bash
16224 pts/0    T      0:00 ssh pc12.cs.ucdavis.edu
16256 pts/0    R+     0:00 ps -x
```

Table 1 says what the columns contain. Table 2 says what the first letter in the STAT column means.

PID	process identification number
TT	controlling terminal (usually, where you ran this from)
STAT	process status
TIME	amount of CPU time the process has used
COMMAND	command that produced the process

Figure 1: Columns in *ps* listing

D	non-interruptible wait (usually short-term waits for network or disk I/O to complete)
I	idle (sleeping more than 20 seconds)
R	running or runnable
S	sleeping (usually waiting for something to finish)
T	suspended by a job control signal
Z	zombie process (terminated, but parents not yet notified)

Figure 2: Meaning of the first letter in the STAT column

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 that comment):

```
% 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
```