CISC 220 Course Notes: Linux and C

David Alex Lamb

July, 2014
External Technical Report
ISSN-0836-0227-2014-621

School of Computing
Queen's University
Kingston, Ontario, Canada K7L 3N6

Version 1.1
Document prepared July 7, 2014

Contents

List of Figures

List of Tables

1 Introduction

There are many Linux books on the market, but many are not suitable for a single-term university-level textbook. Typically they provide far too much detail, or far too little, and don't tie the material to basic concepts taught in other undergraduate computing courses. These notes are meant to address these issues. The initial motivation for writing them was to support the course CISC 220([1]System-Level Programming), offered by the School of Computing at Queen's University in Kingston, Ontario. I hope that eventually they will be useful in a wider context.

These notes assume you are already familiar with using some other operating system, such as MacOS or Windows. In particular you should know what it means to log in and invoke programs, and what file systems and directories (or folders) are.

As a consequence of this intended audience, these notes do not explain every detail of every command; they cover a subset that lets you get a lot done and that get you to the point where you can read and understand more detailed sources. There are typically many different ways to accomplish a given task; the notes usually show only one of the ways. The writing style is deliberately terse. Once you master the basic features described here, you should consult the relevant `man` pages, or search on the World-Wide-Web, for any command you want to understand further. When the notes refer to searching "the manual" for a command or other topic, they mean reading the `man` page for the topic and Wikipedia entries, and using Internet search engines.

Textual conventions in these notes are:

- Sections marked with the dagger symbol (†) are advanced material that might be skipped in an introductory course.
- The first occurrence of a technical term is in *italics*; the explanation usually follows within a few sentences or paragraphs, but may be in a later section indicated by a cross-reference.
- Program names, what you type, and what text the computer prints in response are in a fixed-width `typewriter font`.
- Program listings use variable-width fonts; the listing-formatter package

[1]http://research.cs.queensu.ca/home/cisc220/

uses italics and boldface to indicate comments and keywords, respectively.

- The main text normally describes the basic or normal case for its topic; it may oversimplify to make the explanation easier to understand. There are footnotes to describe minor complexities; you can ignore them on a first reading.

This version of the notes can be printed; URLs are included as footnotes.

Command syntax, options, and behaviour on other UNIX^{TM}systems may differ from that on the version of Linux we used while writing these notes. This version of the notes corresponds to Ubuntu 12.09 Linux in CASlab at Queen's University in July 2013.

The choice of topics for these notes was governed by the class slides for CISC220 from Fall 2012 by Margaret Lamb, who provided much useful guidance on what material to include and how to present it.

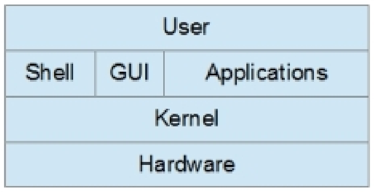
Part I
Linux Basics

2 Overview

Linux is a free *operating system* that runs on many different hardware platforms, from personal computers to mainframes and supercomputers. There is a little ambiguity in the term "operating system." Classically it meant software that manages what happens on a computer: a set of programs for managing memory, the file system, external devices such as terminals and network communication, loading and running *application programs*, and security for reducing the possibility of applications interfering with each others' use of the basic functions of the computer. On Linux, this portion of the operating system is referred to as the *kernel*. These days "operating system" can include some additional software, such as basic user interface facilities and some utility programs.

Figure 1 shows the layers making up the Linux system. A user interacts with some form of *user interface*, an application that runs on top of the kernel. One particular kind of user interface is the *shell*, a textual command-line

Figure 1: Linux Operating System



Classic "layered operating system" diagram specialized for Linux.

interface that is the primary (currently only) way these notes describe how to use Linux.

2.1 History

This is a very brief summary of the history of Linux; for more context, you should consult Wikipedia and other online sources.

Linux is a variant of UNIX. In the 1960s, the research subsidiary of the Bell Telephone System, Bell Laboratories (now ATT Bell Laboratories) and others developed a very complex operating system called MULTICS (`Multi`plexed `I`nformation and `C`omputer `S`ystem). It embodied many good ideas, but had poor performance and was never commercially successful. In the early 1970s, Bell employees created a much simpler and more efficient system called UNICS (`Uni`plexed `I`nformation and `C`omputer `S`ystem); the spelling was later changed to UNIX. It was originally written in assembly language for the PDP-7, but in 1973 was translated to C[2] (sometimes called a "portable assembly language"); this made porting it to other hardware much easier.

In 1988 the Institute of Electrical and Electronic Engineers (IEEE, pronounced "I triple E") released a standard for UNIX-like operating system interfaces, IEEE 1003, that was eventually called POSIX. Many operating sytems were or became POSIX-compliant, at least in part; thus some descriptions of UNIX features refer to POSIX.

On October 5, 1991, Linus Torvalds, then a Finnish university student, released the first version of the Linux kernel. Since it provided a key subset of the facilities of UNIX, it was immediately able to run many of the applications developed by the Free Software Foundation, which were collectively called GNU (a recursive acronym standing for "`G`NU is `N`ot `U`NIX"). Thus the

[2]We cover some aspects of C programming in Section III.

FSF and others prefer to call the whole system (kernel plus key applications) GNU/Linux. It is now "mostly" POSIX-compliant.

UNIX and many of its derivatives are proprietery commercial systems In contrast, GNU/Linux is free, which has made it very popular with general non-commercial users and hobbyists. Because it is free, and source code for it is freely available, several different organizations have released their own "distributions," with variations in exactly what software is available "out of the box" and where various files are placed. The distribution we use in the CASlab, and on which these notes are based, is Ubuntu 12.09.

2.2 Shells

In a typical graphical user interface (GUI), you invoke programs by touching or clicking on some icon, which represents program to run or a file of a particular type (in which case your action runs some specific application associated with that type of file). With a command line interface (CLI), you type names of commands and supply *arguments*. A GUI system is usually much easier to learn and use than a command-line user interface – but only for the most common functions you want to perform. A CLI is usually harder to learn, but makes it possible to do more advanced functions. A GUI is sometimes referred to as WYSIWYG (What You See Is What You Get); however it can also be disparaged as a WYSIAYG (What You See Is All You've Got). A CLI can make it possible to specify advanced functionality, at the cost of a steeper "learning curve."

The fact that the shell is an ordinary application means that there can be many different shells. These notes describe a shell called `bash` (which stands for `B`ourne-`a`gain `sh`ell). There are several others in common use, including

- `sh`, the original Bourne shell, developed by Stephen Bourne of Bell Labs and first realeased in 1977.
- `csh`, the "C shell," which replaced the programming language used in *shell scripts* (Section 5) with one resembling C.
- `ksh`, the "Korn shell," developed by David Korn at Bell Labs in the early 1980's, combined features of `sh` (with which it was backward-compatible) and `csh`.

`bash` was developed by the Free Software Founddation; it combines features of `sh` and `ksh` but is not quite backward-compatible with them.

Linux systems do provide GUIs, and a shell can be accessed from within each GUI. As with shells, there are several different GUIs.

2.3 Interacting with `bash`

When you log into a Linux system like the Queen's University CASlab, you are faced with a *command prompt* such as

```
dal@linux2:~:$
```

Your prompt may look different, because you can customize it (Section 4.3). This particular prompt has the general form

your user name`@` *short computer name*`:` *current working directory*`:$`

The character `~` is an abbreviation for "your home directory," your *current working directory* directly after logging in. At this point you are expected to type *commands* after the prompt, such as

```
dal@linux2:~:$ pwd
/cas/staff/dal
```

This and later examples show you

- the prompt (as above),
- the command you type (`pwd`), and
- the textual output (`/cis/staff/dal`, the full name of the current directory).

The `pwd` command (print working directory) prints the full *path name* of the current working directory – the one in which, by default, any files you examine are found and where those you create will be placed.

Figure 2 shows a short `bash` session.

- `cd` (change directory) changes the current working directory from the home directory, `~`, to one of its subdirectories, `~/220`.
- `ls` (`list`) shows the names of the files (`pride.txt`) and directories (`poems`) stored within the current directory.
- `ls poems` lists the files in subdirectory `poems`.
- `cat poems/jabberwocky` shows the contents of file `poems/jabberwocky`, the first few lines of Lewis Carroll's famous poem.

To understand these examples more fully, you need to understand the Linux file system.

```
dal@linux6:~$ cd c220
dal@linux6:~/c220$ ls
poems/  pride.txt
dal@linux6:~/c220$ ls poems
bed      gentle    jabberwocky  michaelis  road    tolls
birches  gustibus  letter       nov_guest  stop    wall
frog     hope      mary         ring       summer  will
dal@linux6:~/c220$ cat poems/jabberwocky
Jabberwocky
  Lewis Carroll

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.
dal@linux6:~/c220$
```

Figure 2: Example `bash` Session

3 The File System

The Linux file system has four major components.

- A *device* is a physical piece of hardware, such as a hard drive, optical reader, or flash drive. Each device has its own file structures, which typically include a top-level *directory*, a collection of *inodes* describing individual files, tables showing where to find free space for new files, and the actual *contents* of the files.

- An inode is the description of a single file, including what device it is on, where to find the contents of the file, dates (and times) when the file was last accessed or modified, and *permissions* (Section 3.5) governing who can perform what operations on the file. It does *not* contain the name of the file. Each inode has an integer called its "inode number," which you can think of as an index into a big array of inodes. Inode numbers are unique within a device: no two inodes on a device have the same number, but two inodes on different devices might have the same number.

- File contents are stored separately from inodes; inodes contains pointers (such as disk addresses) to file contents. The contents of the file are data *in* the file; the descriptive information in the inode is *metadata* – data *about* the file.

- A directory is one specific kind of file whose contents are essentially just a table of pairs of names and inode numbers.

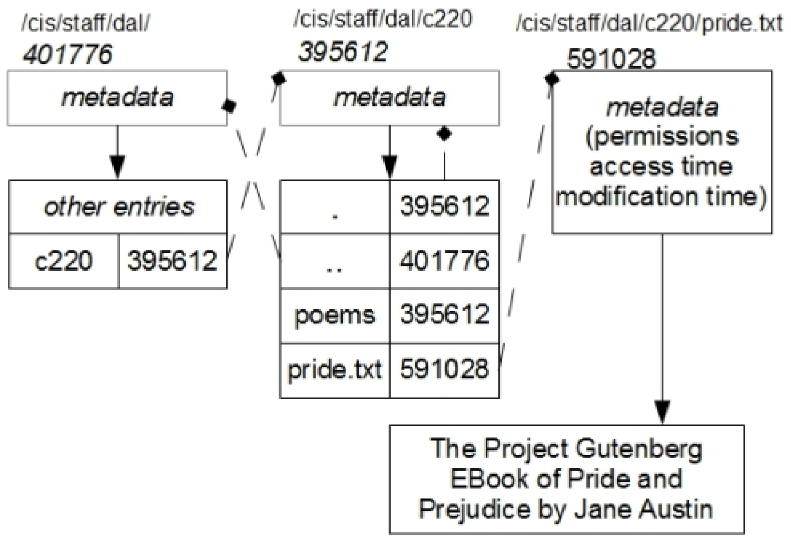
There is a top-level directory named `/` ("slash;" also called the root directory) from which all other directories and files can be reached. The top-level directories of different devices are one or two levels down in the hierarchy. For example, on CASlab the three directories `/proc`, `/cas/staff`, and `/cas/student` are on different devices.[3]

Suppose the current working directory is `/cis/staff/dal` as in the `bash` session of Figure 2, and a program wishes to read file `c220/pride.txt` (relative to the current working directory). Figure 3 shows the corresponding file structures. The top three boxes are inodes, labeled with their path names and inode numbers. Each has a pointer to its file contents. The leftmost two boxes are directories, so their contents are name/number pairs. The leftmost box is the inode for the current working directory; it has inode number 401776. It has a name and inode number pair for `c220` plus other information (represented by ...). When a program wants to open `c220` it calls a kernel function and supplies the path name `c220/pride.txt`. The kernel interprets the name as a set of directions for traversing the directory structure. It finds the entry `c220` in the current directory and gets the corresponding inode number 395612. It opens that inode and reads the information corresponding to the second box. This contains some information about the `c220` directory (not shown) and four entries The entry with name `.` (pronounced "dot") contains the inode number of `c220` itself. The entry with name `..` (pronounced "dotdot") contains the inode number of its parent directory `/cis/staff/dal/`. All directories contain entries for these two names, with the appropriate inode numbers.

Within inode 395612 (`/cis/staff/dal/c220/`) the kernel finds the entry `pride.txt` and finds the inode number 591028. Since this is the end of the path, and a regular file, it stops. Inode 591028 has various information about the file, plus a pointer to the file contents. The bottom box is the file contents (which are *not* an inode): the bytes representing Jane Austin's*Pride and Prejudice*, surrounded by some Project Gutenberg boilerplate text.

[3]The `mount` command links separate devices and file structures into the directory structure; it is beyond the scope of these notes.

Figure 3: Inode Structure for `c220/pride.txt`



Inode structure. Solid lines with arrows show connections between inodes and file contents; this is an abstraction of what might be multiple pointers to multiple chunks of content. Dashed lines with diamonds show indirect connections via inode numbers.

It is important to emphasize that the inode for a file is stored separately from its contents. The contents are the actual data in the file; the inode is metadata – data *about* the file. Metadata can change without changing contents. For example, each time anyone reads a file, the *access time* of the file changes to the date and time the file was opened, but the contents do not change. When someone writes to a file, the *modification time* changes as well as the access time. When someone changes the access control *permissions* (Section 3.3 on page 13), the inode and its *change time* updates but the file contents and modification time do not.

The directory `/dev` is special: the "files" within it represent physical and logical devices. The details of `/dev` are beyond the scope of these notes, but a few are worth noting:

- `/dev/null` is a "device" with no contents. Reading from it immediately yields end-of-file, and writing to it is ignored. If you were testing a program you might tell it read from `/dev/null` just to verify that its end-of-file processing worked, and might tell it to write to `/dev/null` if you wanted to ignore some of its outputs.

- `/dev/stdin`, `/dev/stdout`, and `/dev/stderr` represent the standard input, output, and error streams of the current process (Section 4.2 on page 26).

- `/dev/tty` is the current "terminal" – the terminal or window of the shell that launched the process in question. This is different from `stdin`, `stdout`, and `stderr`, since those can be redirected to regular files.

3.1 Hard Links

The use of inodes means that an ordinary (non-directory) file does not necessarily have a unique name. Two different entries can have the same inode number – either entries in two different directories, or two entries in the same directory with different names. These are often referred to as "hard links" to contrast them with *symbolic links* (Section 3.6 on page 19). You can't create a hard link to a directory;[4] consequently, each directory has a unique name and a unique "parent directory" that contains it.

Figure 4 shows two directories (*inode1* and *inode2*) and an ordinary file (*inode3*). Names are text; arrows represent inode numbers. Each directory has an entry `.` (dot) with its own inode number, and `..` (dotdot) with its parent's inode number. With *inode1* as the current working directory:

- Pathnames `nameN` and `name1/name1a` refer to the same file, *inode3*.

- Pathnames `.` and `name1/..` both refer to the current working directory.

The boxes saying *other entries* stand for an arbitrary number of other name/inode pairs, possibly none.

3.2 File-related `bash` Commands

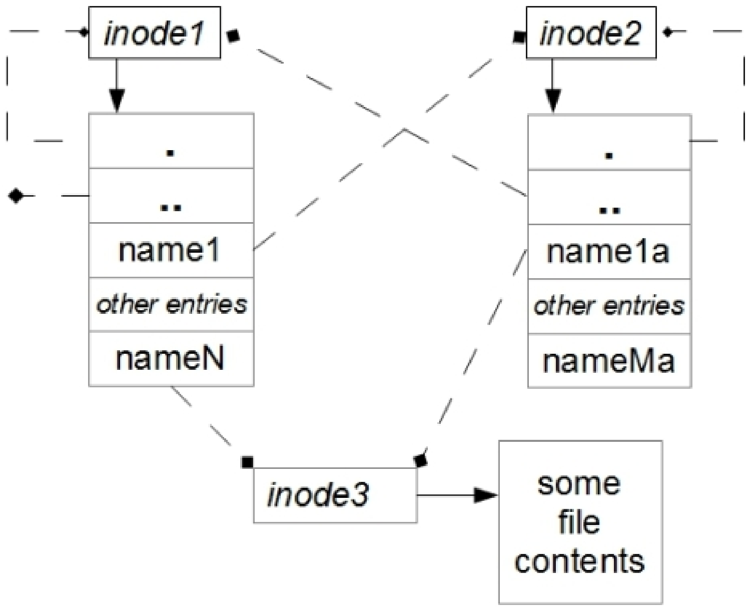
Given a basic understanding of the file system, we can describe some general features of `bash` commands, using file system manipulation commands as examples. The general form of a shell command is

```
    name [ flags ] [ other arguments ]
```

The various portions of this command line are:

[4]Except via the `mkdir` command (page 11).

Figure 4: Multiple Hard Links to a File



Multiple hard links to `inode3`. Dashed arrows represent indirect connections via inode numbers; solid mean direct inode-to-contents connections via disk addresses.

- The *name* identifies the command. The name can be a file system path name for an executable file, but it can also be a simple name without slashes. Some such simple names are "built into" the shell (that is, interpreted directly by the shell program, without invoking any separate programs), while for others the shell finds the executable file by looking for a file with that name in a collection of directories called the *search path* (Section 4.5) (or just "the path").

- Items surrounded by square brackets `[]` are optional.

- *flags* is a set of "command line options" that modify the behaviour of a command. Without them, the command has a certain default behaviour. Each flag starts with a dash or hyphen (`-`) and almost always consists of a single letter. You can run several flags together after a single dash; thus `ls -l -t` means the same thing as `ls -lt`.

- *other arguments* are whatever else the command needs to perform its functionality. They often consist of pathnames for files on which the command operates.

The common `bash` commands dealing with files are:

- Section 2.3 on page 5 introduced the `pwd` command to print the current working directory, and the `cd` command to change it.

- `ls` (as shown in Figure 2 on page 6) shows the contents of the current working directory. Normally it shows the names in several columns, sorted by name. `ls -t` shows them in order of *modification time* (page 8), most recent first. `ls -l` shows a "long" form with many pieces of information about the files, such as size, modification times, what user owns it, and its *permissions* (Section 3.3); `ls -i` adds inode numbers.

- `rm` removes a link to a file. This does *not necessarily* delete the file; the file is deleted when the last hard link to it is removed. It does not place the file in any sort of "recycle bin," so you must use it very carefully. Since commonly there is only a single link to an ordinary file, many Linux system administrators arrange that `rm` prompt for confirmation that you intend to delete a file – but this is not the default. To delete a directory, you should use `rmdir` instead of `rm`. The `-i` ("interactve") flag causes `rm` to ask you to confirm for each file whether you wish to delete it. The `-f` command suppresses error messages about deleting files taht don't exist.

- `mkdir` *pathname* creates a new directory with the given path name (which can include slashes; every name but the last one in the path must already exist). This is distinct from creating a file because a directory contains the two special entries named `.` (a link to itself) and `..` (a link to its parent directory). This means new directories have three specific hard links (Section 3.1 on page 9): one from the parent directory to the new one, and the special links `.` and `..` for the new directory and its parent.

- `rmdir` *pathname* by default removes a directory if it is empty; if you want to remove a whole directory tree, you should delete all files in it, and all its subdirectories, by hand.[5]

[5] There are ways to remove a directory and all its contents recursively, but it is a *very bad idea* to do so unless you're absolutely sure it's what you mean to do.

- `touch` *pathname* creates an empty file if it doesn't already exist, or updates the modification time of the file (Section 3 on page 8) if it does exist. Updating the modification time can be useful with the `-newer` flag of `find` (page 52), or to force recompilations when using the `make` command (Section 14.3 on page 110).

- `cp` copies an ordinary file to a new location. The two typical usages are

  - Copy a single file:
    ```
    cp pathname1 pathname2
    ```
    copies the contents of the file named *pathname1* to one named *pathname2*.
  - Copy a collection of ordinary files to a directory:
    ```
    cp pathname1 ··· pathname2 directory
    ```
    The directory must already exist.

  In both cases `cp` by default overwrites the destination file if it already exists and creates it if it doesn't. `cp -i` asks for confirmation before overwriting.[6]

- `ln` makes a new link for a file. Consider Figure 4 on page 10, and suppose *inode1* were the current directory and were empty (had no files). The diagram could have been established by the sequence of commands
  ```
  mkdir name1
  touch nameN
  ln nameN name1/name1a
  touch name1/nameMa
  ```

- `mv` moves a file. If the destination is in the same file system, it is a combination of `ln` to the new name and `rm` of the old name. If the destination is on a separate file system, it is equivalent to `cp` followed by `rm`.

- `stat` prints the contents of an inode in a readable form. Figure 5 shows the results of a `stat` command.

[6]There is a `bash` shell mechanism called *aliasing* (Section 4.5 on page 35) that lets you arrange for `cp` to mean `cp -i`; some system administrators make a default setting in your initial *`bash` login script* (Section 5 on page 40).

Figure 5: Results of `stat` Command

```
dal@linux3:~/notes/poems$ stat mary
  File: ‘mary’
  Size: 105             Blocks: 16         IO Block: 1048576 regular file
Device: 17h/23d Inode: 48660951    Links: 1
Access: (0600/-rw-------)  Uid: (2133180/     dal)   Gid: (  200/ student)
Access: 2013-08-08 11:45:41.000000000 -0400
Modify: 2013-07-05 14:27:12.000000000 -0400
Change: 2013-07-05 14:27:12.000000000 -0400
 Birth: -
dal@linux3:~/notes/poems$ ls -l mary
-rw------- 1 dal student 105 Jul  5 14:27 mary
dal@linux3:~/notes/poems$
```

The first `Access` is *file permissions*; Uid and Gid are user (owner) and group IDs (Section 3.5 on page 18). The second `Access`, plus `Modify` and `Change`, are the corresponding dates and times (Section 3 on page 8).

3.3 File Permissions

Any system shared by multiple users needs a way to control access to files. Linux does this by introducing the idea of an *owner* for each file – the login name of whoever created it – and *group* – a collection of users created by the system administrator.[7] Each file (including each directory) has three sets of *permissions* describing who can manipulate the file and in what way. For an ordinary file,

- `r` (“read”) means the user can read the file: list its contents.
- `w` (“write”) means the user can write to the file: change its contents.
- `x` (“execute”) means the user can execute the file, that is, run it as a program.

Normally file owners have `rw-` permissions on their ordinary files and `rwx` permission on their programs (where the `-` indicates a lack of the `x` permission).

For a directory, the permissions are slightly different.

- `r` means the user can list the contents of the directory; this is a consequence of being able to read it like an ordinary file. If you don’t have read permission, you can’t discover the names of any files in the directory.
- `w` means the user can create and delete files in the directory; this is a consequence of being able to change the directory.

[7]For more on owners and groups see Section 3.5 on page 18.

- `x` means the user can look up a file in the directory – that is, get the inode for a file, if the user knows its exact name. You can have `x` permission on a directory without having `r` permission. This means the owner of the file can arrange for other people to read (and even write) files a directory without being able to find the names of every file. The Linux kernel reads the directory itself to find a specific file name, but won't let users read the directory themselves. See Section 3.4 on page 17 for an example.

Normally owners have `rwx` permissions on their directories, and give read and execute permissions (`r-x`) for everyone else on directories they want to be publicly readable.

The representation of a permission (which you can view with the command `ls -l`; see Figure 6) is a sequence of seven characters of the form *kabcdefghi* The first letter (indicated by *k*) is not actually a permission; it is the kind of file (`d` for a directory, `-` for an ordinary file, and `l` for a *symbolic link*.[8] The next three sets of three letters describe permissions. The first group of three (*abc*) show permissions for the owner, the second (*def*) for the group, and the third (*ghi*) for everyone else. In each three-letter group, each position corresponds to one of the three permissions (read, write, or execute). If the letter is a `-`, it means the corresponding group of people lack that permission. If it isn't `-`, it is a specific other letter.

1. In the first position (*a*, *d*, and *g*) an `r` means the corresponding people have read permission.
2. In the second position (*b*, *e*, and *h*) a `w` means the corresponding people have write permission.
3. In the third position (*c*, *f*, and *i*) an `x` means the corresponding people have execute permission.

To see information about a directory in detail, you can use the command `ls -liaF`.[9] Figure 6 shows the results of `ls -liaF` in the directory with inode *inode1* in Figure 4 on page 10. There are several things to note about this example.

[8]See Section 3.6 on page 19.

[9]`-l` ("long") shows most of this information. `-i` adds inode numbers. `-F` "flags" entries with a character indicating its type; in particular, this is where the trailing `/` comes from at the end of directory names. There is a convention that `ls` by default doesn't show "hidden" files (those whose names start with a dot); the `-a` option shows these files.

Figure 6: Long-Form directory Listing

```
dal@linux6:~/notes/experiment$ ls -liaF . name1
.:
total 28
48660968 drwx------ 3 dal student 4096 Jul  9 09:18 ./
48660888 drwx------ 4 dal student 4096 Jul  9 09:14 ../
48660971 drwx------ 2 dal student 4096 Jul  9 09:17 name1/
48660972 -rw------- 2 dal student    0 Jul  9 09:16 nameN

name1:
total 24
48660971 drwx------ 2 dal student 4096 Jul  9 09:17 ./
48660968 drwx------ 3 dal student 4096 Jul  9 09:18 ../
48660972 -rw------- 2 dal student    0 Jul  9 09:16 name1a
48660974 -rw------- 1 dal student    0 Jul  9 09:17 nameMa
dal@linux6:~/notes/experiment$
```

Results of `ls -liaF`. Columns are inode number, permissions, number of hard links, owner (`dal`), group (`student`), size, modification date/time, name.

- Since the shell prompt shows the current directory (between the `:` and the `$`), `.` is also `~/notes/experiment`.
- `name1` and all four entries starting with `.` are marked as directories (first character of the permission is `d`). All the ordinary files have a `-` in this position.
- None of the files and directories are accessible to anyone but the owner (`-` in the last two groups of three permission flags).
- The owner can read (list), write (create files in) and execute (look up files in) all the directories.
- The owner can read and write all the ordinary files, but cannot execute them as programs.

- The inode numbers for `nameN` and `name1/name1a` are the same; the latter had been created with a `ln` (create hard link) command.
- `./name1` and `name1/.` have the same inode number. Each has 2 hard links.
- `.` and `name1/..` have the same inode number. There are three links, because the parent of `.` (`~/notes`) also links to `.`
- Even "small" directories like this are fairly large – 4 kilobytes in this case.
- All the ordinary files are of size 0, since they were created with `touch` rather than with something that would give them real content.

To change permissions on a file you can use the `chmod` command.[10] The simplest form is

```
    chmod octal file(s)
```

The octal number is a set of three digits each composed by adding 4 for `r` permission, 2 for `w`, and 1 for `x`. Thus `rw-` permission is 6, and `r-x` is 5. If there are less than three octal digits, the leading ones are taken to be 0. The first digit is permissions for the owner, the second for the group, and the third for everyone else.

```
    chmod 750 file1 file2
```

There is a complex mnemonic syntax that uses letters instead of numbers; the two most common are

```
    chmod go= file1 file2
```

to remove all permissions for group (`g`)and other (`o`),

```
    chmod a+rx file1 file2
```

to add execute permission for everyone (useful after you create a shell script, Section 5).

By default you should set group and other permissions to none (`go=`). This isn't the default unless your system administrators have made it so. The command

```
    umask octal
```

sets the *file mode creation mask* to the given octal number. When a program creates a file specifying certain permissions, the kernel turns off any bits with a 1 in the umask. Thus `umask 077` leaves owner permissions alone but ensures

[10]For historical reasons this isn't something less unexpected like `chperm`; the name stands for `change mode`.

group and other permissions are turned off. There is also a symbolic form of `umask` as with `chmod`. To find the current `umask` you can type:

```
dal@linux6:~$ umask
0077
dal@linux6:~$ umask -S
u=rwx,g=,o=
```

3.4 Directory Read and Execute Permissions†

Section 3.3 explained what read and execute permissions meant for directories, but typically beginners have trouble understanding why there should be a difference. To recap,

- `r` permission lets you read the directory as a file, and thus list all its contents – the names and inode numbers of every file and subdirectory within it.

- `x` permission lets you look up the inode number associated with a specific name, even if you can't see all the names in the directory.

This difference lets you make a very fine distinction in how people can access your files. Suppose you have a `shared` directory into which you put files you want other people to read. You might want anyone to be able to read `rant.txt` but only members of your group to read `iHateC220.txt`. Furthermore you might not want the professor to even know `iHateC220.txt` exists. You could achieve this via

```
cd shared
chmod a+r rant.txt
chmod o=,g=r iHateC220.txt
chmod g=rx,o=x .
```

Figure 7 shows the effects of these commands. More calls to `chmod` would be needed to give `x` permission to `~` and `~/notes` – but beware that opening up directories like this would would let people read and even write files for which you forgot to turn off "group" and "other" permissions.

You can tell everyone "see `~me/shared/rant.txt`" and they could use a program like `less` to read it. Only your group (`student`) can `ls ~me/shared`"

Figure 7: Directory Without Read Permission

```
dal@linux6:~/notes$ cd shared
dal@linux6:~/notes/shared$ ls -liaF .
total 32
48660973 drwx------ 2 dal student 4096 Jul 11 10:44 ./
48660888 drwx------ 5 dal student 4096 Jul 11 10:43 ../
48660977 -rw------- 1 dal student   96 Jul 11 10:44 iHateC220.txt
48660976 -rw------- 1 dal student   68 Jul 11 10:44 rant.txt
dal@linux6:~/notes/shared$ chmod a+r rant.txt
dal@linux6:~/notes/shared$ chmod g=r iHateC220.txt
dal@linux6:~/notes/shared$ chmod g=rx,o=x .
dal@linux6:~/notes/shared$ ls -liaF .
total 32
48660973 drwxr-x--x 2 dal student 4096 Jul 11 10:44 ./
48660888 drwx------ 5 dal student 4096 Jul 11 10:43 ../
48660977 -rw-r----- 1 dal student   96 Jul 11 10:44 iHateC220.txt
48660976 -rw-r--r-- 1 dal student   68 Jul 11 10:44 rant.txt
dal@linux6:~/notes/shared$
```

Creating a directory with `x` but not `r` permissions to make one file readable to everyone and another only to the group and invisible outside the group.

and see both file names, and only they can read `iHateC220.txt`. Everyone else would get an error message.[11]

No one can `cd` to `~me` or `~me/notes`, since they lack `r` permission.

3.5 Owners and Groups

The owner of a file is *initially* the user name (login identifier) of whoever created it. There is a `chown` command to change owners; it only works if invoked by the current owner or by a system administrator. "A system administrator" means the special "superuser", the login id `root`; it has full permission to change any files in the system.

[11]Of course there is nothing preventing a system administrator from putting professors in the `student` group as well as a `faculty` or `staff` group.

A group is a list of user names created by a system administrator. A user can be in several groups; the `groups` command lists them. One of these groups is primary (the one you're in when you log in). The *initial* group of a file is the primary group of its creator; the `chgrp` command can change it. If *any* of the groups you are in are the same as that of the file, you have the group permissions for that file (the second group of three permissions shown by `ls -l`) – unless, of course, you are the owner, in which case owner permissions apply instead.

If you are in a team project, typically your team would have its own group; you'd be in a general `student` group as well your team's special group. You might create shared directories and files whose group is the one for your team.

3.6 Symbolic Links

Hard links have three problems:

- You can't make hard links across file systems; a directory on one disk drive can't make a hard link to something on a different drive, because inodes are device-specific.
- You can't make hard links to directories except via `mkdir` commands.
- Some editors (such as `emacs`) write a new file and unlink the old file instead of modifying the old file in-place (as `vi` does). Their operation is basically
  - Copy the contents of the old file into internal memory.
  - Create a new temporary file with a different name, usually in the same directory as the old file.
  - When the user "saves" the file, write the contents of internal memory to the temporary file, unlink the old file,[12] and rename the temporary file to the old name.

  A hard link elsewhere references the old file, not the new one.

To solve all these problems, Linux has *symbolic links*. A symbolic link is a special file type whose contents are a pathname. When someone opens the

[12]If the editor "keeps backups" it renames the old file instead of unlinking it. Linux users on CASlab will see Emacs backup files with a ~ at the end of their name.

symbolic link, Linux interprets the pathname to find the file it names, and opens that file instead.

You create a symbolic link with the `-s` option to the `ln` command. For example, when the current directory is

```
    /cis/staff/dal/notes/experiment
```

(the same as in Figure 4) the command

```
    ln -s /cis/student/somebody/asgt1.c student.c
```

creates a file in the current directory whose name is `student.c`, marked as a symbolic link, and whose content is the string

```
    /cis/student/somebody/asgt1.c
```

(possibly including a bit more information, depending on the details of the Linux implementation).

When using a symbolic link, two sets of permissions are involved: those of the symbolic link itself, and those of the file it names. When you do an `ls -l` on a symbolic link, you see permissions `lrwxrwxrwx`, but this doesn't mean you can actually do anything with the file it names. Linux follows the path contained in the symbolic link and uses the permissions of the actual file. The permisions on the symbolic link itself are never used.

3.7 (Lack of) File Formats

Linux does not impose any particular constraints on the contents of files; any "format" is just a convention shared among some user-level programs. The standard UNIX/Linux convention for text files is to interpret a "newline" character (`\n`, "line feed") as the end of the line. Unfortunately, files imported from MicroSoft Windows follow a different convention, wherein lines end with `\r\n` ("carriage return" followed by "linefeed"). The Windows convention is historical: on typewriters and hardcopy computer terminals there were two separate characters to end a line. "Carriage return" (`\r`) returned the mechanism holding the paper so that new keystrokes would place ink on the left-hand edge. "Line feed" (`\n`) would rotate the cylinder holding the paper so that new keystrokes would occur on the next line. The two mechanisms were separate; a bare carriage return would allow overstriking of keystrokes, whereas a bare line feed would start a new line in the middle of the paper (wherever the carriage happened to be at that moment).

The commands `fromdos` and `todos` convert their arguments between Windows and Linux formats "in-place:" they modify the file rather than making a copy.

Figure 8 shows the contents of the file `mary` containing the poem "Mary had a little lamb." `ls` shows that it is 105 bytes long. `wc` ("word count") shows

Figure 8: File Contents, `od`, and `wc`

```
dal@linux3:~/notes/poems$ ls -l mary
-rw------- 1 dal student 105 Jul  5 14:27 mary
dal@linux3:~/notes/poems$ wc mary
  4  20 105 mary
dal@linux3:~/c220/poems$ cat mary
Mary had a little lamb,
Little lamb, little lamb.
Mary had a little lamb
Whose fleece was white as snow.
dal@linux3:~/notes/poems$ od -c mary
0000000   M   a   r   y       h   a   d       a       l   i   t   t   l
0000020   e       l   a   m   b   ,  \n   L   i   t   t   l   e       l
0000040   a   m   b   ,       l   i   t   t   l   e       l   a   m   b
0000060   .  \n   M   a   r   y       h   a   d       a       l   i   t
0000100   t   l   e       l   a   m   b  \n   W   h   o   s   e       f
0000120   l   e   e   c   e       w   a   s       w   h   i   t   e    
0000140   a   s       s   n   o   w   .  \n
0000151
dal@linux3:~/notes/poems$ od -x mary
0000000 614d 7972 6820 6461 6120 6c20 7469 6c74
0000020 2065 616c 626d 0a2c 694c 7474 656c 6c20
0000040 6d61 2c62 6c20 7469 6c74 2065 616c 626d
0000060 0a2e 614d 7972 6820 6461 6120 6c20 7469
0000100 6c74 2065 616c 626d 570a 6f68 6573 6620
0000120 656c 6365 2065 6177 2073 6877 7469 2065
0000140 7361 7320 6f6e 2e77 000a
0000151
dal@linux3:~/notes/poems$ cd ..
dal@linux3:~/notes$ cp poems/mary mary.txt
dal@linux3:~/notes$ todos mary.txt
dal@linux3:~/notes$ od -c mary.txt
0000000   M   a   r   y       h   a   d       a       l   i   t   t   l
0000020   e       l   a   m   b   ,  \r  \n   L   i   t   t   l   e    
0000040   l   a   m   b   ,       l   i   t   t   l   e       l   a   m
0000060   b   .  \r  \n   M   a   r   y       h   a   d       a       l
0000100   i   t   t   l   e       l   a   m   b  \r  \n   W   h   o   s
0000120   e       f   l   e   e   c   e       w   a   s       w   h   i
0000140   t   e       a   s       s   n   o   w   .  \r  \n
0000155
dal@linux3:~/notes$
```

how many lines (4), words (20), and characters/bytes (105) are in the file. `cat` shows the contents of the file with each character interpreted in whatever way "the terminal" interprets it – so that newlines start text on a new line. `od -c` ("octal dump") shows the contents as individual characters with escape sequences for non-printing characters (just `\n` in this case); the numbers on the left edge are character counts in octal (16 characters per line). The last few lines show the results of converting a Linux-format poem to DOS (Windows) format.

Figure 9 shows a file with non-printing "control characters" – in this case, control-A and control-H. As before, `cat` shows the characters in whatever form "the terminal" interprets them. The control-A does not print (note the two consecutive spaces after the word "control-A") whereas the control-H (backspace) moves back one character (note the single space after the word "control-H"). `od` shows the control-A as octal 001, and the control-H as the

Figure 9: File Containing Control Characters

```
dal@linux3:~/notes$ cat controlChar.txt
This line has a control-A  and
a control-H character, each
surrounded by spaces.
dal@linux3:~/notes$ wc controlChar.txt
 3 13 84 controlChar.txt
dal@linux3:~/notes$ od -c controlChar.txt
0000000   T   h   i   s       l   i   n   e       h   a   s       a
0000020   c   o   n   t   r   o   l   -   A     001       a   n   d  \n
0000040   a       c   o   n   t   r   o   l   -   H      \b       c   h
0000060   a   r   a   c   t   e   r   ,       e   a   c   h  \n   s   u
0000100   r   r   o   u   n   d   e   d       b   y       s   p   a   c
0000120   e   s   .  \n
0000124
dal@linux3:~/notes$
```

escape sequence `\b`.

4 The `bash` Shell And Basic Commands

Section 3.2 on page 9 introduced the basics of the `bash` shell, including commands that affect the file system. This section describes `bash` more thoroughly.

A `bash` command line is of the form

```
name [ flags ] [ other arguments ]
```

It is very common for most of the arguments to be file system path names identifying which files the command will operate on.

Two simple commands for illustrating some `bash` features are `cat` (short for "concatenate") and `echo`. The directory `~/notes/poems` contains the files:

```
dal@linux6:~/notes/poems$ ls
backup/  frog      hope         mary       ring  summer  will
bed      gentle    jabberwocky  michaelis  road  tolls
birches  gustibus  letter       nov_guest  stop  wall
```

```
dal@linux6:~/notes/poems$
```

Figure 10 shows a typical usage of `cat`. In the current (`poems`) directory

Figure 10: Example of Using `cat`

```
dal@linux6:~/notes/poems$ cat jabberwocky bed
Jabberwocky
  Lewis Carroll

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.
Bed in Summer
  Robert Louis Stevenson

In winter I get up at night
And dress by yellow candle-light.
In summer quite the other way,
I have to go to bed by day.

I have to go to bed and see
The birds still hopping on the tree,
Or hear the grown-up people's feet
Still going past me in the street.

And does it not seem hard to you,
When all the sky is clear and blue,
And I should like so much to play,
To have to go to bed by day?
dal@linux6:~/notes/poems$
```

it finds the two files `jabberwocky` and `bed` and writes their contents to the terminal[13] in the order you listed them on the command line. In contrast, the `echo` command simply writes its arguments – their literal text – to the terminal:

```
dal@linux6:~/notes/poems$ echo put jabberwocky to bed
put jabberwocky to bed
dal@linux6:~/notes/poems$
```

[13]This is an oversimplification; see Section 4.2 on page 27.

`echo` may seem trivial but is useful for illustrating several more complex features of how `bash` interprets a command line. For example, `echo` takes some flags as arguments, which it doesn't write to the terminal:

```
    dal@linux6:~/notes/poems$ echo -n here is a prompt
here is a promptdal@linux6:~/notes/poems$
```

The `-n` switch causes it to omit the newline at the end, which causes the next `bash` prompt to start on the same line as the echoed text. This can be used in shell scripts (Section 5 on page 39) to write their own prompts, but the `read` command's `-p` option is a better way to do this (Section 5.1 on page 41).

If you `cat` some very long file, your terminal will scroll down so that you only see the last screenful of text. The commands `head` and `tail` show the first and last few lines of a long file, respectively. If you want to see pause at the end of each screenful, you can use `less` instead of `cat`. Table 1 shows you the main commands you can use within `less` to move around in the file.

Table 1: Commands Within `less`

Command	**Functionality**
`f` or space	forward one screenfull
`b`	backward one screenfull
`e` or ENTER	forward one line
`y`	backward one line
`h`	display help screen, which describes more commands
`q`	exit

If you want to find out more about a given command, you can use the `man` ("manual") command. `man cat` for example shows documentation about the `cat` command. `man man` shows the `man` commands own documentation. Since these notes only show the basics of each command, you should use `man` to find out more. The argument doesn't need to be a shell command; the "manual" includes, for example, information about various functions in the standard C program library. Internally, `man` uses a program similar to `less` to display screensful, so you can use the commands of Table 1 to move around in the manual.

4.1 Wildcards and Filename Expansion

The shell interprets certain characters on a command line as special; `*` (asterisk or star) is one of them. If you try:

```
dal@linux6:~/notes/poems$ echo this g* has a star
this gentle gustibus has a star
dal@linux6:~/notes/poems$
```

instead of a copy of the arguments you typed, you get

1. The word `this`.

2. A list of all the files in the current directory starting with `g`

3. The words `has a star`.

The shell turned the `g*` into item 2. Had we omitted the `g` we would have seen a list of all the files in the current directory.

This is an example of *command line expansion*, wherein the shell turns some sequences of characters you type into other text before passing it to the command. This particular example is called a *filename wildcard* or just *wildcard*. A slightly more complex example is

```
dal@linux6:~/notes/poems$ ls -l *op*
-rw------- 1 dal student 378 Jul  5 14:27 hope
-rw------- 1 dal student 671 Jul  5 14:27 stop
dal@linux6:~/notes/poems$
```

The shell took `*op*` to mean "any file in the current directory whose name contains the charactes `op`, possibly preceded or followed by other characters." The `*` was a *pattern* matched by any sequence of characters. The other wildcard character is `?`, which matches any single character:

```
dal@linux6:~/notes/poems$ echo w?ll
wall will
dal@linux6:~/notes/poems$
```

The sequence `[`*`list of characters`*`]` matches any one character in the list. Thus the previous example could have been written

```
dal@linux6:~/notes/poems$ echo w[ai]ll
wall will
dal@linux6:~/notes/poems$
```

It would *not* have matched a file named `well`, whereas `?` would have.

A very special form of expansion is the tab character (or control-i, written `^i`). If you type this after a partially-typed file name, the shell will immmediately substitute the rest of the file name (plus a trailing space). Thus in the `poems` directory, `ja^i` completes to `jabberwocky`. If there were two files that started with the same characters, it would complete up to just before the first different character; thus if there were a `jabbering` file, `ja^i` completes to `jabber` without a trailing space. A second tab at this point would show all the file names with the current prefix, then echo the current command line:

```
dal@linux6:~/notes/poems$ ls -l jabber^i^i
jabbering    jabberwocky
dal@linux6:~/notes/poems$ ls -l jabber
```

At this point typing `w^i` would complete the name to `jabberwocky`. An online experiment or demo should clarify this example.

Section 4.3 on page 30 describes several other expansions `bash` performs.

4.2 I/O Redirection

So far we have shown all program output going to the screen ("the terminal"). You can *redirect* output to a file instead, using the sequence `>`*filename*:

```
dal@linux6:~/notes/poems$ cat jabberwocky bed >catenated
```

At this point we can use the `head` and `tail` commands to look at the first and last few lines of the file; they take an argument saying how many lines to show.

```
dal@linux6:~/notes/poems$ head -4 catenated
Jabberwocky
  Lewis Carroll

'Twas brillig, and the slithy toves
dal@linux6:~/notes/poems$ tail -3 catenated
When all the sky is clear and blue,
And I should like so much to play,
To have to go to bed by day?
dal@linux6:~/notes/poems$
```

What is going on here is that programs generally don't write to "the terminal;" they write to their *standard output*, which *defaults* to the terminal. The `>` character causes the shell to set the command's standard output to the given file.

Suppose we tried to `cat` several filenames, one of which did not exist:

```
dal@linux6:~/notes/poems$ cat jabberwocky ug bed >catenated
cat: ug: No such file or directory
```

The standard output still goes to file `catenated` as before, but an error message has appeared on the terminal. A program typically writes all its *normal* output to its "standard output," but it typically sends its error messages to its *standard error* instead. Like standard output, standard error defaults to the terminal. When you redirect standard output, the only thing left to go to the terminal is the error messages.

You can redirect standard error separately from standard output by writing the number 2 before the `>` character.

```
dal@linux6:~/notes/poems$ cat jabberwocky ug bed >catenated 2>fred
dal@linux6:~/notes/poems$ cat fred
cat: ug: No such file or directory
dal@linux6:~/notes/poems$
```

Standard output went to `catenated`; standard error went to `fred`.

The significance of the number 2 is that, when a program opens a file, Linux sets up a data structure called a *file descriptor*. Three special file descriptors are set up by the shell before the program executes. 2 is the number of a file descriptor for standard error. 1 is standard output.[14] Thus the above example could equally well have been:

```
dal@linux6:~/notes/poems$ cat jabberwocky ug bed 2>fred 1>catenated
dal@linux6:~/notes/poems$ cat fred
cat: ug': No such file or directory
dal@linux6:~/notes/poems$
```

The two redirections can go in either order.

Sometimes you want to direct *both* outputs to the same place. The shell provides a special syntax for this:

[14] 0 is "standard input," which we discuss later.

```
someCommand 1>someFile 2>&1
```

This redirects standard output (`1>`) to `someFile`, then standard error (the `2>` part) to the same file as standard input (the `&1` part). Alternatively you could type:

```
someCommand 2>someFile 1>&2
```

which redirects standard error to `someFile`, then standard output to the same file as standard error. The two achieve identical effects via slightly different paths.

If the output file for a redirect (`>` or `>&`) already exists, `bash` can be told not to overwrite it via a *shell option* called `noclobber`. To see the values of all options, use the command `set -o`. To set the option, use `set -o noclobber`. To turn off the option, use `set +o noclobber`. When `noclobber` is set, you can force the overwrite of a file by using `>|` instead of `>`.

Commands also have a *standard input*, which also defaults to the terminal. For example, the `sort` command (without arguments) reads from standard input, sorts it, and sends the result to standard output.

```
dal@linux6:~/notes/poems$ sort
this is the first typed line
but this line will come before it
and this line will come first.
^d
and this line will come first.
but this line will come before it
this is the first typed line
dal@linux6:~/notes/poems$
```

The first line invokes the `sort` command. The next four I typed; the `^d`, control-d, told Linux I wanted to end the input. The last four are the sorted output, and the next command prompt. You can redirect standard input, too, using the `<` character, as shown in Figure 11.

There also is a special form of redirecting standard input, called a *here document*, which is sometimes used in shell scripts (page 41).

It is often useful to use the standard output of one program as the standard input of another. The command

```
cat * | less
```

Figure 11: Redirecting Standard Input

```
dal@linux6:~/notes/poems$ cat frog
I'm Nobody!  Who Are You?
Emily Dickinson

I'm nobody!  Who are you?
Are you nobody, too?
Then there 's a pair of us -- don't tell!
They 'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
dal@linux6:~/notes/poems$ sort <frog


Are you nobody, too?
Emily Dickinson
How dreary to be somebody!
How public, like a frog
I'm Nobody!  Who Are You?
I'm nobody!  Who are you?
Then there 's a pair of us -- don't tell!
They 'd banish us, you know.
To an admiring bog!
To tell your name the livelong day
dal@linux6:~/notes/poems$
```

Contents of file `frog` in order, then sorted. Note the `<` on the `sort` command to redirect its standard input. The two empty (blank) lines in the output of `sort` are from the stanza separator lines in `frog`.

causes `cat` to sent the contents of all the files in the current directory to its standard output, and `less` to take its standard input from `cat`'s output. This is called *piping*; the shell introduces a special sort of "file" called a *pipe* between the two programs. It is similar in effect to

```
cat * >temp
less <temp
rm temp
```

but potentially more efficient; pipes are often implemented in memory, saving file space, and the two programs can proceed in parallel, interleaving `cat` writing a little of its output with `less` reading a little of its input. `less` can do part of its work without waiting for `cat` to finish.

You can build a long series of pipes; each program's output becomes the next's input. In a way it is like function composition in a functional programming language like Haskell, taught in CISC 260: Programming Paradigms.[15]

4.3 Other Command Line Expansions

bash interprets several characters or character sequences in special ways, substituting the results at the place where the characters occurred. We have already seen pathname wildcards * and ?. This section describes several others.

As we will see in Section 5 on page 39, the shell interprets a full-blown programming language. One feature of any programming language is variables. The shell keeps track of several predefined variables; for example, the sequence $SHELL or ${SHELL} substitutes the value of the variable SHELL (the pathname of the current shell) at that point in the command line. For example:

```
dal@linux6:~/notes/poems$ echo ${SHELL} $SHELL x${SHELL}y
/bin/bash /bin/bash x/bin/bashy
dal@linux6:~/notes/poems$
```

Other important shell variables are

- $HOME, the name of your "home" directory (the current directory immediately after you logged in). The character ~ means the same as $HOME; ~name means the home diretory of user name.
- $PATH, the current *search path* (Section 4.5 on page 36)
- $PWD, the current working directory.
- $USER, your user name

To see the values of all the (many!) shell variables, use the set command.[16]

The shell variable PS1 is the prompt string bash uses to tell you it is ready for a new command. It can contain literal characters plus some special ones:

- \d: current date
- \h: name of the host machine

[15] http://www.cs.queensu.ca/students/undergraduate/courses/desc/CISC-260.html

[16] set will also show you the currently-defined *shell functions*, which are beyond the scope of these notes.

- `\j`: number of *jobs* you have running (Section 4.6 on page 36).
- `\s`: shell name
- `\u`: user's name
- `\w`: current working directory
- `\!`: history number of the command you're about to enter.

You can also set your own variables:

```
dal@linux6:~$ today=Tuesday
dal@linux6:~$ echo This day is $today.
This day is Tuesday.
dal@linux6:~$
```

If you use `$` on a varible that hasn't been assigned to, its value is the empty string.

`bash` expands the sequence `x{a,b,c}y` (*without* a preceding `$`) into the three strings

```
xay aby xcy
```

There can be multiple such brace expansions in a line:

```
dal@linux3:~/notes$ echo x{a,b}y{c,d}z
xaycz xaydz xbycz xbydz
dal@linux3:~/notes$
```

One convenient use for brace expansion is copying a file to a backup with a similar name:

```
cp longComplexFileName{,1}.txt
```

This has the same meaning as

```
cp longComplexFileName.txt longComplexFileName1.txt
```

Sometimes you want to issue a command that is the same, or nearly the same, as one you recently typed. `bash` keeps track of what commands it has executed recently (its *history*) and provides several means for reducing typing based on it. The sequence `!!` means "substitute the last command at this place in the command line" and `!`*number* means "substitute the command with the given number at this point in the command line." At one point where I was generating examples for these notes, the last few lines of the history were:

```
  344  cat jabberwocky bed
  345  echo jabberwocky bed
  346  echo put jabberwocky to bed
  347  cat put jabberwocky to bed
  348  man echo
  349  echo -n here is a prompt
  350  ls
  351  echo this * has a star
  352  ls -l *op*
  353  echo w?ll
  354  ls
  355  history
```

The command line `!352` would re-execute `ls -l *op*`, whereas `echo pre !! post` would execute `echo pre history post`.

To turn off these command line expansions, you can surround a portion of your command line in quotes. Double quotes (`"`) turn off wildcards. Single quotes (`'`, apostrophe) turn off shell variable expansion (`$`) and history expansion (`!`) also.

A more powerful form of command repetition is to edit the previous (or current) command line; this can save a lot of retyping if you make a mistake. The up-arrow key or the character `^p` (control-p) fetches the previous command and lets you edit it; more `^p` commands go back further in the history. You can use the arrow keys to move around in the line, or the characters `^f` and `^b` to move forward or backward one character. Normal characters are inserted at the current spot. When you hit the enter key, the shell executes the edited command. Table 2 shows the various line-editing commands. You can use these same commands in the middle of the current command line, too. For full details, search the `bash` manual for information about "readline," the name of the library that handles such editing.

The final form of command line expansion is *command substitution.* The string `$(`*`command`*`)` (with parentheses instead of braces) causes `bash` to execute the command between the parentheses and substitute its standard output.[17] Suppose you had a command `findSome` that would output a list of file names containing a particular string, and you wanted to copy them to a backup directory:

```
    cp $(findSome) backup
```

[17]There is an older way to do this with *back-quotes* (`` ` ``), but it is harder to read.

Table 2: Command-Line Editing

`^b`, left arrow	backward (left) one character
ESC-`b`	move backward one word
`^d`, DEL	delete current character
ESC-`d`	delete current word
`^f`, right arrow	forward (right) one character
ESC-`f`	move forward one word
`^h`, backspace	delete previous character
ESC-`^h`	delete previous word
`^n`, down arrow	replace line with next (more recent) line of history
`^p`, up arrow	replace line with previous (less recent) line of history
`^r`*text*	search backward in this line for *text*.
`^s`*text*	search forward in this line for *text*.

Most editing commands are those of the `emacs` text editor. `^`*letter* means to hold down the control key while typing the letter. `DEL` means the delete key. `ESC-`*character* means to type the escape key followed by the character, or hold down the alt key while typing the character. Deleting and moving by a "word" involves characters from the current point either forwards or backwards to a word boundary.

Page 55 has an example of using `grep` to do this.

There are two commands whose main use is within `$(...)`:

- `dirname` *pathname* prints the directory part of the pathname – everything up to the last slash (`/`). If there is no slash, it prints "`.`" (dot, meaning the current directory).

- `basename` *pathname* prints just the part of the pathname after the last slash.

- `basename` can take a second argument, a file extension. If the extension matches, it prints the regular basename but without the extension. If it doesn't, it prints as though the second argument were omitted.

Thus:

```
dal@linux6:~/notes$ dirname /cas/staff/dal/c220/pride.txt
/cas/staff/dal/c220
dal@linux6:~/notes$ basename /cas/staff/dal/c220/pride.txt
pride.txt
dal@linux6:~/notes$ basename /cas/staff/dal/c220/pride.txt .txt
pride
dal@linux6:~/notes$ basename /cas/staff/dal/c220/pride.txt .cpp
pride.txt
dal@linux6:~/notes$
```

4.4 Recursive Calls on `bash`

One of the many commands you can call from `bash` (or any other shell) is `bash` itself. There are many reasons for doing so; the most common is what happens "behind the scenes" when you invoke a *shell script* (Section 5 on page 39). However, you may wish to do so explicitly if you are playing around with shell state (such as by changing the contents of important shell variables) and want to be able to easily revert to the original values. The value of `$SHLVL` tells you how many levels deep in shell calls you are.

When you call a shell recursively, you can quit and get back to the shell that invoked it via the `exit` command. Exiting from the top-level shell (created by logging in) does a `logout`.

A recursive shell inherits several things from the shell that calls it, including the current working directory and the umask (Section 3.3 on page 16). It also inherits *some* of the shell variables: those in "the environment." You add a variable to the environment with the `export` builtin command:

```
 export MYNAME
```

adds MYNAME to the environment:

```
dal@linux3:~$ echo $SHLVL
1
dal@linux3:~$ MYNAME="David Alex Lamb"
dal@linux3:~$ echo $MYNAME
David Alex Lamb
dal@linux3:~$ bash
dal@linux3:~$ echo $SHLVL
2
dal@linux3:~$ echo $MYNAME
```

```
dal@linux3:~$ exit
exit
dal@linux3:~$ export MYNAME
dal@linux3:~$ bash
dal@linux3:~$ echo $MYNAME
David Alex Lamb
dal@linux3:~$
```

You can find the values of all environment variables with the `printenv` command.

Recursive shells don't inherit aliases (Section 4.5 on page 35) or shell options(Section 4.2 on page 28), so you need to put them in your initialization script, `.bashrc` (Section 5 on page 40).

4.5 Finding the Command†

If your command is a full path name (with slashes), there is no ambiguity about where `bash` looks to find it: it simply follows the path, and, if the file it finds has execute permission (`x`), it executes it. If instead the command is just a name without slashes, `bash` uses several mechanisms to figure out what to do.

First, if you are in an *interactive* shell (one reading commands from the terminal, not a script), `bash` looks to see if you have defined an *alias* with that name via the `alias` command. Without arguments, this command lists the current aliases (in the same format you'd use to define them):[18]

```
dal@linux6:~$ alias
alias ll='ls -l'
alias ls='ls -F'
alias rm='rm -i'
dal@linux6:~$
```

With these aliases, if you are in the `poems` directory used in previous examples and type `rm w*`, `bash` substitutes the alias to get `rm -i w*`, then does wildcard expansion to get

```
 rm -i wall will
```

and finally executes the revised command line – which, for each of the two files, asks you to confirm that you wish to delete it.

[18]To remove an alias, use the `unalias` *name* command.

After aliasing, if the command is still a single word, `bash` consults the *search path.* This is a shell variable named `PATH` whose value is a string of directory names separated by colons (`:`). `bash` examines each of these directories in turn, looking for an executable file of the same name as the command. It substitutes the resulting path name for the command word. Thus given the path `~/bin:/usr/local/bin:/user/bin` and command `xx`, where there are is a *non*-executable file `/usr/local/bin/xx`, an *executable* `/user/bin/xx`, but no file named `~/bin/xx`, `bash` will turn `xx` *arguments* into

`/usr/bin/xx` *arguments.*

The exact process is

- Break up the value of `PATH`at the colons into its three components, `~/bin`, `/usr/local/bin`, and `/user/bin`
- Look for `~/bin/xx`; none exists.
- Look for `/usr/local/bin/xx`; it exists, but isn't executable.
- Look for `/user/bin/xx`; it exists and is executable, so use substitute it for `xx` on the command line.

The command "`which xx`" tells you the full pathname that results from this expansion.

4.6 Running Multiple Programs

When you issue a command to `bash` that isn't built in, it creates a separate *process* in which to run it. A process has its own section of the computer's memory, and Linux keeps its state separate from all other processes. This means, among other things, that anything that goes wrong during a command's execution can't affect other processes, such as your shell.

There can be more than one program running at once. Each can be in one of three states: foreground, background, or suspended. Only one is in the "foreground," while the others are either in the "background" or are suspended. The main distinction is that only the foreground program can read from the terminal; Linux blocks background processes temporarily when they try to do so. Multiple processes can *write* to the terminal; their output gets interleaved in what is likely to be a confusing manner.

When you are running a program, typing control-z will *suspend* it – temporarily stopping it from running, and return to the shell. For example, you might be running an editor and need to look up something in a very large manual page. You could suspend the editor, run the `man` command, find the right spot in the middle of the manual, then suspend `man` and resume the editor.[19] If you needed to consult the manual again, you could suspend the editor and continue the `man` command, which would redisplay the manual at the point you last viewed it.

Each of the programs you start from the shell is called a *job*. The `jobs` command gives you a list of the current jobs. Each job has a number; you can type `%`*number* to continue a suspended job or bring a background job into the foreground. Alternatively, if the job has a unique prefix you can type `%`*prefix*. For example, if you are running both `man` and `emacs`, and suspend both, you would see

```
dal@linux6:~/notes/poems$ jobs
[1]-  Stopped                  man bash  (wd: ~)
[2]+  Stopped                  emacs
dal@linux6:~/notes/poems$
```

`%1` or `%man` would continue the `man` command; `%2` or `%em` would continue the `emacs` command. The `+` means that `%` by itself would run `emacs`.

When you suspend a job, you can send it into the background to continue running with the `bg` command, which takes `%` arguments as described above. You can also *start* a command in the background by terminating the command line with an & (ampersand); this can be used to start a *server* process that waits for incoming internet connections, for example, but servers are beyond the scope of these notes.

To get rid of a job, use the `kill` command. `kill %1` would kill the `man` process.[20]

Processes other than the shell can themselves start other processes; Section 11 on page 80 shows how to do this from C programs. For example, the `man` command runs a program called `pager` to print a screenful of text at a time. Also, you can run `bash` (or a different shell) recursively from within `bash`. This creates many processes, only some of which are actual jobs. For example, suppose we run `man` and `emacs` as above, then run a second `bash` com-

[19]In a modern GUI, each program would have its own separate window, both likely visible at the same time. The Linux mechanisms were designed for an older time, when users typically had only a small screen without graphics. Even so, it can save on system resources, and reduce the amount of screen space that you need to keep visible at one time.

[20]The `kill` command can also send *signals* to processes; see Section 12 on page 89.

mand and suspend it (with the suspend command; control-z doesn't suspend a shell). The ps command will show all processes.

```
dal@linux6:~/notes/poems$ bash
dal@linux6:~/notes/poems$ suspend
[3]+  Stopped                 bash
dal@linux6:~/notes/poems$ jobs
[1]   Stopped                 man bash  (wd: ~)
[2]-  Stopped                 emacs
[3]+  Stopped                 bash
dal@linux6:~/notes/poems$ ps
  PID TTY          TIME CMD
11172 pts/2    00:00:01 bash
11742 pts/2    00:00:00 man
11752 pts/2    00:00:00 pager
11974 pts/2    00:00:00 emacs
11992 pts/2    00:00:00 bash
12002 pts/2    00:00:00 ps
dal@linux6:~/notes/poems$
```

The first bash in the output from ps is the current one; the second is the suspended one, job 3. man and pager are the processes making up job 1; the wd beside man bash in the jobs listing shows its working directory. ps is the command itself; it has terminated by the time we get back to the shell prompt, but was active when it was discovering what processes to display.

4.7 Order of Command-Line Expansions†

Earlier notes covered many different things the shell does before executing a command line; this section summarizes the order in which it does so.

- First, the shell splits the command lines into *tokens* – words and operators. Quotes (" and ') group items together into a single token.
- *Brace expansion*: The shell turns a token like xx{aa,bb,cc}yy into xxaayy xxbbyy xxccyy.
- *Tilde expansion.* This is where ~x/something/ becomes *x's home directory*/something.

- *Parameter/variable expansion* – the various `${...}` expressions described in previous sections.
- *Command substitution* – the `$(...)` construct, which involves running a sub-command before executing the main one.
- *Arithmetic expansion* – `$((`*`expression`*`))` is evaluated and substituted.
- *Process substitution*, which is beyond the scope of these notes.
- *Word splitting*: If the results of parameter expansion, command substitution, and arithmetic expansion did not occur within double quotes, it is split into words.
- *File name expansion* – the wildcards `*`, `?`, and `[...]` described previously.
- *Redirection.* The shell sets up redirections and eliminates the corresponding tokens (such as `>` and associated filenames) from the command line.
- *Execution.* The shell runs the command (or commands, if there were pipes).

Understanding this order is occasionally important.

5 Shell Scripts

Suppose you have a complicated command you need to use repeatedly. For example, Section 6.2 on page 52 shows an example of combining the `find` and `grep` commands to locate `import` statements in a collection of subdirectories:

```
     find ~/java/src -name "*.java" -exec grep -nH import "{}" \;
```

It is not necessary to understand this example: it is sufficient to notice that you wouldn't want to type it more than once. If you wanted avoid memorizing and retyping this command, you could put it in a file called a *shell script.* In general, given some complex command, you could create a file such as `findImport` in which you would put the two lines

```
     #!/bin/bash
     the complicated command
```

The first of these lines is a comment explained below; the second is the actual command. You'd then make `findImport` executable by telling `bash`:

```
    chmod u+x findImport
```

Thereafter, when you type `./findImport`,[21] your current shell would invoke `bash` and tell it to take commands from `findImport` in the current directory, which would then execute the complicated command on the second line (and go on to execute any other lines in `findImport`, if there were any). After that, the shell running the script finishes and returns to the shell from which you invoked the script.[22]

Lines starting with a hash mark (`#`) are comments. The first line of `findImport`, `#!/bin/bash`, was optional. It tells the shell to run `bash` and pass it the contents of the file to execute. The term "scripting" is a general term for writing "executable text files" – text files meant to be interpreted by some conventional executable program, of which `bash` is one example. The initial two characters `#!` are part of a general mechanism used by many different shells to tell them what interpreter to run for whatever language the text file contains. Other possible interpreters include `perl` and other shells like `ksh`.

Scripts can take parameters, which become shell variables while the script is executing. `$0` refers to the command name itself, `$1` to the first parameter, `$2` to the second, and so on. Thus if `findImport` were

```
    #!/bin/bash
    find $1 -name "*.java" -exec grep -nH import "{}" \;
```

then typing

```
    findImport ~/java/src
```

would have the same effect as the original example.

Normally a script executes in a separate shell process from the one in which you invoked it (see Section 4.6 on page 36). Any changes to shell state that happen within a script have no effect on the original shell. Thus if you assign to a variable or make an alias within a script, their effects "go away" when the script finishes. If you want to execute a script within the *current* shell, you type

```
    source scriptFileName [ arguments ]
```

`bash` uses the same `source` mechanism to initialize itself. Whenever you login, `bash` looks for several files to `source`, including one under your control: `~/.bash_profile`. Whenever you invoke `bash` directly as a subshell, it `sources` `~/.bashrc`. You should examine both these files to see what the

[21] See page 36 about the `PATH` variable for why the `./` is necessary.

[22] If you want to ensure this script would execute from whatever other directories you might `cd` to, make sure `findImport` is in some directory on your search path (Section 4.5 on page 36).

system administrator set up as your default initialization; you can edit them to change your preferences for how `bash` should behave.

There are several commands which, while they are legal to type to an interactive shell, are more common and convenient in scripts. The following sections discuss several of them.

5.1 Reading From the Command Line

Sometimes instead of (or in addition to) passing a list of arguments to a shell you want to read some input from the terminal while the script executes. The command `read` *var* reads a line from the standard input and places it in the variable. With several variables, the first word is placed in the first variable, the second in the second variable, and so on to the second-last variable; the rest of the line is placed in the last.

With the flag `-p` *prompt*, and if input is coming from a terminal instead of a file or pipe, the shell first prints the prompt (without a trailing newline). The flag `-s` ("silent") causes it not to echo characters; you would typically use this to read a password. For example:

```
#!/bin/bash
some setup code
read -p "Which key?  " which rest
read -ps "Password:  " pass rest
jarsigner -keystore cryptkey -keypass ${pass} jar1.jar ${which}
jarsigner -keystore cryptkey -keypass ${pass} jar2.jar ${which}
```

uses the Java `jarsigner` program to digitally sign two jar files; keys are kept in an encrypted "key storage" file called `cryptkey`, which requires a password to decrypt. The first `read` asks for which key to use (several can be stored in the same file). The second asks for the password in "silent" mode (suppressing echoing). The last two lines invoke the jar signer. Without the `-keypass` option each call to the signer would prompt for the password; having the script read the password saves duplication.

A specialized form input redirection called a *here document*, introduced with double angle brackets `<<`, lets you enter the contents of a file on a command line. You follow the brackets with a word; when that word appears alone at the start of an input line, the shell recognizes it as the end of the input. This mechanism can be used to create short sequences of inputs in for testing, without having to create and later delete an input file. For example

```
dal@linux3:~/notes$ sort <<END | uniq
```

```
> first line of input
> second line of input
> another line
> END
another line
first line of input
second line of input
dal@linux3:~/notes$
```

It is more useful in scripts than when typing commands to the shell; it is reasonably common, for example, to write scripts to test and re-text sequences of commands; here documents keep the test commands and their (presumably short) inputs in a single place.

5.2 `if` and Conditionals

The formal syntax of the `if` statement is

```
if test-commands ; then
    commands ;
[ elif test-commands ; then
    commands ;]*
[ else
    commands ; ]
fi
```

Unlike Java and C, conditionals in shell scripts don't use braces to surround groups of statements; the keywords `then`, `elif` ("else if"), `else`, and `fi` delimit the set of commands to be executed. The `[ elif ... ]*`, as with *regular expressions* (Section 8 on page 57), means any number of repetitions of the `elif` clause. Like Java but unlike Python, the indentation doesn't affect the meaning of the program but is very useful for readability.

`bash` interprets a *line-oriented* language, which means that almost all commands end at a newline. The semicolon (`;`) before `then` was only needed because `then` was on the same line as `if` (and `elif`).

There are several different kinds of test commands.

The basic boolean element in the shell is the *exit status* of the previous `bash` command. Every command returns a numerical exit status when it finishes. Contrary to the meaning in C, a zero exit status means success, and anything else means failure. Thus you can write

```
if cmp -s file1 file2; then
```

```
   echo files are identical
else
   echo files are different
fi
```

The `cmp -s` command compares two files and prints nothing. It exits with status 0 if they are the same, and nonzero if they differ. Note the semicolon before `then`.

The shell variable `$?` means the exit status of the most recent command. The default exit status of a script is that of the last command it executes. You can specify an explicit exit status with `exit` *number*, which means "exit the script now, returning *number* as the exit status."

To distinguish command execution from boolean expressions that the shell interprets, you put the expression inside double square brackets:[23]

`[[` *`expression`* `]]`

You must have spaces around the square brackets so the shell will distinguish them from parts of the expression. You can combine boolean expressions with parentheses, `&&` (and), `||` (or), and `!` (not). Two common kinds of expressions are string comparisons and file property tests.

To compare strings you can use use `==`, `<`, and `<` (but not `<=` or `>=`). Thus to test whether the current directory is the home directory:

```
if [[ $HOME == $PWD ]]
then
    echo I am home
else
    echo I am somewhere else
fi
```

Since `then` is on a separate line, there is no need for a semicolon at the end of the first line.

There are also several boolean tests of the form

`-letter filename`

that test for properties of files; see Table 3

You can evaluate arithmetic expressions inside `((...))` (double parentheses). The *exit status* is zero (successful) if the *expression* yields a non-zero number. Thus `((5+7))` evaluates to `12`, and

[23]In old scripts you may see a holdover form with single square brackets; Section 5.3 on page 44 shows why it is obsolete.

Table 3: File Property Tests

`-e` *file*	File exists
`-f` *file*	File exists and is a regular file
`-d` *file*	File exists and is a directory
`-h` *file*	File exists and is a symbolic link
`-r` *file*	File exists and is readable
`-w` *file*	File exists and is writable
`-x` *file*	File exists and is executable
file1 `-nt` *file2*	File1 exists and is newer than file2, or file2 doesn't exist

```
    if ((5+7)); then echo nonzero; else echo zero; fi
```

executes the `echo nonzero` command after `then`. You can make numerical comparisons inside double-parentheses; these allow `<=` and `>=` as well as `<`, `>`, and `!=`.

You can also write assignments (`=`) inside double parentheses (which, of course, means you need to be very careful about distinguishing `=` from `==`). The difference between *var=expression* inside double-parentheses and outside them is that, inside, the expression is taken as numerical, and outside is taken as a string. Thus:

```
dal@linux6:~$ X=5+2
dal@linux6:~$ ((Y=5+2))
dal@linux6:~$ echo $X '!=' $Y
5+2 != 7
dal@linux6:~$
```

5.3 Aside: Single Square Brackets†

There is an older form of boolean expression that uses single brackets instead of double. It behaved slightly differently: I/O redirection happened inside them, whereas with double square brackets the redirection operators `<` and `>` are interpreted as comparison operators. Figure 12 shows two scripts that differ only in bracketing. The double square bracket case produces the expected "no" response. The single square bracket case produces the unexpected "yes" response and has the side effect of creating a file called `Downsview`. You may

Figure 12: Single Versus Double Square Brackets

Double and Single Bracket Scripts

```
#!/bin/bash                    #!/bin/bash
# doubleBracket.sh             # singleBracket.sh
X=Calgary                      X=Calgary
if [[ $X > Downsview ]]        if [ $X > Downsview ]
then                           then
  echo yes                       echo yes
else                           else
  echo no                        echo no
fi                             fi
#!/bin/bash
X=Calgary
if [[ $X > Downsview ]]
then
  echo yes
else
  echo no
fi
```

Result of executing the two scripts

```
dal@linux3: /notes$ ls D*
ls: cannot access D*: No such file or directory
dal@linux3: /notes$ ./doubleBracket.sh
no
dal@linux3: /notes$ ls D*
ls: cannot access D*: No such file or directory
dal@linux3: /notes$ ./singleBracket.sh
yes
dal@linux3: /notes$ ls D*
Downsview
dal@linux3: /notes$
```

find some equality and inequality comparisons inside single square brackets in older `bash` scripts.

5.4 Looping

The two looping constructs are `while` and `for`. The syntax of `while` is:

`while` *`test-commands`*`; do` *`commands`*`; done`

As with the `if` command, you can omit the semicolons if you put `do` and `done` on separate lines.

The syntax of `for` is

`for` *`name`* `[ in` *`words`* `...]; do` *`commands`*`; done`

The *words* can be any sequence of items separated by spaces; the commands get executed once for each word in the sequence; in the body of the loop, `$`*`name`* refers to the current word. For example:

- `for X in *; do echo $X; done`

  echoes the names of all the files in the current directory.

- The token `$*` expands to the list of parameters, separated by spaces, so

  ```
  for PARAM in $*; echo $PARAM; done
  ```

  echoes all the parameters of the current script, one per line.

- `for VAL in 3 5 9; ...`

  performs commands for the values 3, 5, and 9.

- `for FILE in a* *.txt; ...`

  performs commands for all the file names starting with `a` or ending in `.txt`; as usual for combining multiple wildcards in a command line, file names of the form `a*.txt` will occur twice.

If you omit the *words* the default is `$*`, the list of all the command arguments (as used in the second example).

If you want to iterate over a sequence of numbers, you can combine the `seq` command, which prints such a sequence, with the `$(...)` construct for command substitution (Section 4.3 on page 32).

```
for X in $(seq 1 10)
do
    echo $X
done
```

prints the numbers from 1 to 10, while

```
for X in $(seq 1 2 10)
do
    echo $X
done
```

prints the odd numbers from 1 to 9 inclusive.

Sometimes one wants to deal with multiple arguments at a time, then advance to the next group of arguments. A primary example is with flags of the form `-`*`flag word`*. The `shift` ***`n`*** command was meant for this situation. It "shifts" parameters left by n. For example, if there were originally six parameters, then after `shift 2`, `$1` takes on the old value of `$3`, `$2` the old value of `$4`, parameters `$5` and `$6` are "unset," and `$#` (the number of parameters) becomes 4. Given the script

```
#!/bin/bash
while (($#>0)); do
  if (($#>=2)); then
    echo "Next pair: " $1 $2
    shift 2
  else
    echo "Singleton: " $1
    exit
  fi
done
```

in file `pairs`, the results of a call on `./pairs` with five arguments would be:

```
dal@linux6:~/notes$ ./pairs p1 p2 p3 p4 p5
Next pair:  p1 p2
Next pair:  p3 p4
Singleton:  p5
dal@linux6:~/notes$
```

5.5 Advanced Shell Variables

We have seen that `$X` and `${X}` expand to the value of shell variable X, `$`n to the n^{th} parameter to the script, `$*` to the list of parameters, and `$#` to the number of parameters.

There are several other variants using `$`:

- `${var:-alt}` expands to the contents of `var` if it has a value, but to `alt` if `var` is unset or an empty string.

```
dal@linux6:~/notes$ X=hello
dal@linux6:~/notes$ echo ${X:-bye}
hello
dal@linux6:~/notes$ unset X
dal@linux6:~/notes$ echo ${X:-bye}
bye
dal@linux6:~/notes$
```

- `${var:offset:length}` expands to a substring of the given length starting at the given offset (zero origin). Omitting the length or giving a length that would go beyond the end of the string goes to the end of the string instead.

```
dal@linux6:~/notes$ X=SomeString
dal@linux6:~/notes$ echo ${X:1:4}
omeS
dal@linux6:~/notes$ echo ${X:5}
tring
dal@linux6:~/notes$ echo ${X:5:15}
tring
dal@linux6:~/notes$
```

- `${#var}` is the length of the contents of `var`.

```
dal@linux6:~/notes$ echo $X ${#X}
SomeString 10
dal@linux6:~/notes$
```

- `${var/pattern/string}` is the value of `var` with the *first* match of `pattern` replaced by `string`.

```
dal@linux6:~/notes$ echo ${X/S/Z}
ZomeString
dal@linux6:~/notes$
```

The pattern can contain wild cards as with file names.

```
dal@linux6:~/notes$ echo ${X/o*S/Z}
SZtring
dal@linux6:~/notes$
```

- `${var//pattern/string}` is the value of `var` with *all* matches of `pattern` replaced by `string`.

```
dal@linux6:~/notes$ echo ${X//S/ZZ}
ZZomeZZtring
dal@linux6:~/notes$
```

6 `find`: Finding Files

Suppose you are looking for a Java program with particular string (such as `Tree`) in the middle of its name. If you knew what directory it was in, you could type

```
ls *Tree*.java
```

to find its exact name. However, if you had several packages, the file could be in one of several different directories (since the Java language makes use of the Linux directory tree structure to represent its package hierarchy). If you weren't sure which directory to look in, you'd have to issue several `ls` commands, or a single command with several arguments, one per level in the hierarchy. For example,

```
ls *Tree*.java */*Tree*.java */*/*Tree*.java
```

IF there were another level in the directory tree, this command would miss it.

The `find` command lets you specify this search more concisely:

```
find .  -name "*Tree*.java"
```

You can think of the general form of the command as

```
find directories [ tests ] [ actions ]
```

Each test and action starts with a multi-character keyword preceded with a hyphen (-). The *tests* govern which files `find` deals with; `-name` is one example. The *directories* is a list of directories in which to search; if omitted, it defaults to the current directory (`.`). `find` stops looking for directory names after the first occurrence of `-` after the first directory name. `find` starts in each of these directories in turn, applies the *tests*, performs the *actions* on any files that pass the tests, then repeats for each subdirectory (recursively, to the bottom of the directory tree). By default (with no actions) it prints the names of any files that pass its tests.

Technically, the general form of the command is:

```
find directories [ predicates ]
```

Actions are just predicates with side effects; such "predicates" usually evaluate to **true**.

The most common predicates are

- `-name` *pattern* is true for file names matching the pattern. If the pattern contains the `*` and `?` wildcards, it must be quoted to protect the wildcards from expansion by the shell.
- `-iname` *pattern* ignores case in matching the pattern. Two major uses of this are finding file names for which you aren't sure of the capitalization (such as Java class names), and finding `.html` and other files accessible from the World-Wide Web (which Internet conventions allow to have any possible capitalization).
- `-type` *letter* finds specific types of files. `-type f` finds regular files; `-type d` finds directories; `-type l` finds symbolic links.[24]

You can combine predicates; all must be true. Thus

```
find ~/java/src -type d -name "test*"
```

finds all subdirectories of `~/java/src` whose names start with `test`.

The most common actions are

- `-print` echoes the file name to standard output; this is so common that it is the default.
- `-delete` deletes the file without confirmation. One reasonably safe use for this is

  ```
  find someDir -name "*~" -delete
  ```

  which deletes all Emacs backup files in `someDir` and its subdirectories.

[24]There is no letter for hard links, since those are indistinguishable from regular files.

6.1 Predicates About File Properties

The following are moderately common searches based on specific properties of files, such as size and access and modification times (Section 3 on page 8). Specific numbers are just examples and can be replaced by any other integers.

- Find all files modified less than a day ago.
  ```
  find .  -mtime -1
  ```
  A `+1` instead of `-1` would mean "more than one day ago" and no plus or minus would mean "one day ago."[25]

- Find all files modified in the last hour and a half:
  ```
  find .  -mmin -90
  ```
  `-mtime` uses days; `-mmin` uses minutes. `-mmin` interprets its argument in the same way as `-mtime`: `+` for "more than," `-` for "less than," and nothing for "exactly" (after rounding).

- Find all "large" files, taken arbitrarily as meaning those bigger than two megabytes:
  ```
  find .  -size +2M
  ```
  The sign and its absence have the same meanings as with `-mtime` and `-mmin`.[26] The final letter can be `c` for bytes, `k` for kilobytes, `M` for megabytes, and `G` for gigabytes. Eventually someone might add `T` for terabytes, but as of this writing that hasn't happened.

Other file property predicates include

- `-atime` and `-amin`, similar to `-mtime` and `-mmin` but for access time (the last time the file was opened for reading).

- `-perm` *octal* for all files with the given exact permission. When preceded by a `-` it means to check only the bits in the file permission that correspond to `1` bits in the octal number. You can use this to, for example, find files executable by the owner:
  ```
  find .  -perm -100
  ```

[25] The four predicates `-mtime`, `-mmin`, `-atime`, and `-amin` round times *down*. Thus `-mtime 1` actually means from 1.0 to 1.999999... days ago.

[26] Contrasting with `-mtime` and other time-related predicates, `-size` rounds its argument *up* to the nearest multiple of its resolution.

There is a more complex version that uses the same symbolic syntax as `chmod`, where you can turn on and off specific bits using mnemonics to represent read, write, and execute permissions for the owner, group, and other segments.

- `-newer` *file*, all files modified later than the given file. This might be used for some form of incremental file backup.
- `-follow` is always true; it causes `find` to follow symbolic links to directories, detecting cycles to prevent infinite loops.

6.2 The `-exec` Predicate†

If one of `find`'s built-in predicates doesn't do what you want, you can use the `-exec` predicate to execute an arbitrary command. The two forms are

```
    -exec command arguments "{}" arguments ";"
    -exec command arguments "{}" +
```

The first form is more common. It executes the command with the given arguments, replacing {} with the current file name. The `";"` (an escaped semicolon) ends the command to be executed. Thus

```
    find ~/java/src -name "*.java" -exec grep -nH import "{}" \;
```

finds all `.java` files in a particular Java source directory and its subdirectories, running `grep -nH import` *file* on each file it finds.[27] The `-exec` predicate returns true if the command succeeds (return code 0) and false otherwise.

The second form, ending with `+`, is for efficiency. It causes `find` to pass as many matched filenames as possible to the command, rather than executing the command once for each filename. You can't pass any arguments between {} and ; because of the way `find` simply appends file names to the command line.

Since `-exec` is a predicate, it is possible to use several in the same `find` command. The later ones will only execute if the earlier ones succeed. This is the best way to simulate the `&&` operator of Java, C, and C++. However, if you want to do something complex with the command, it may be simpler to write a shell script (Section 5 on page 39) taking one or more filename arguments.

[27]See the `grep` command in Section 7 on page 54.

Part II

Textual Pattern Matching

Several Linux tools include facilities for matching patterns in strings. These notes cover three of them, in increasing order of functionality:

- `grep` (Section 7 on page 54), which finds lines in text files.
- `sed` (`s`tream `ed`itor, Section 9 on page 62), which finds substrings within lines and replaces them with other strings.
- `awk` (Section 10 on page 67), a C-like programming language with special facilities for matching patterns in arbitrary strings.

All use *regular expressions* (Section 8 on page 57) for pattern matching.

We first introduce `grep`, the simplest of the three to explain, then cover the syntax of regular expressions, and finally describe `sed` and `awk`.

This part of the notes assumes you have mastered basic Linux operation from earlier notes (or elsewhere), and that you are comfortable experimenting with program features to explore the details of how something works. In particular, you should be familiar with.

- The hierarchical file system.
- Basic use of the `bash` shell, including standard input, standard output and pipes (`|` operator).
- The way of describing shell command invocation:

  ```
  command [ flags ] filename(s)
  ```

  where the square brackets indicate optional elements and the flags begin with hyphens (`-`).
- Wildcards (`*` and `?`) in file names on shell command lines.
- The use of a escapes (\ prefix) or quotes (`"` and `'`) to "turn off" the special meaning of a character like `*`.

7 grep: Finding Strings in Files

The `find` command (Section 6 on page 49) searches for files based on its properties: name, type, size, modification date, and so on. Sometimes one wants to search based on content; that is the main purpose of `grep`.

To search files for lines matching a pattern, invoke

```
grep [ flags ] pattern filenames
```

For example,

```
grep -nH "import java.util" *.java
```

finds all lines that import items from package `java.util` in all Java files in the current directory.[28] If no such lines exist, this will produce no output (the typical convention with Linux tools). In a Java package directory, it would produce output such as

```
DynamicStringLocalizer.java:3:import java.util.ArrayList;
DynamicStringLocalizer.java:4:import java.util.Iterator;
DynamicStringLocalizer.java:5:import java.util.List;
ItemLocalizer.java:3:import java.util.Iterator;
LocaleEvent.java:3:import java.util.Locale;
Localizer.java:3:import java.util.ArrayList;
Localizer.java:4:import java.util.Iterator;
Localizer.java:5:import java.util.List;
Localizer.java:6:import java.util.Locale;
Localizer.java:7:import java.util.ResourceBundle;
RecordLocalizer.java:3:import java.util.ArrayList;
RecordLocalizer.java:4:import java.util.Iterator;
StringLocalizer.java:3:import java.util.ArrayList;
StringLocalizer.java:4:import java.util.HashMap;
StringLocalizer.java:5:import java.util.Iterator;
StringLocalizer.java:6:import java.util.List;
StringLocalizer.java:7:import java.util.Map;
```

Each line of the output corresponds to a single line of some input file, preceded by the name of the file and the line number (1-origin) within the file. This convention is reasonably widespread in Linux, and is often used by compilers. Many programming environments such as Eclipse and editors like Emacs have some way to run a program like `grep`, collect their output, scan the output for each file-name-and-line-number pair, and use them to position a program

[28]For a more flexible version of this pattern, see Section 8.2 on page 60.

editor at the corresponding line in the file.

7.1 Command Line Options

There are several variants of the basic output:

- The `-o` flag shows just the parts of the line that match the pattern, instead of the whole line.
- The `-v` flag prints lines that *don't* match instead of those that do.
- the `-w` flag matches only “words” – sequences of letters, numbers, and underscores (`_`) surrounded by line boundaries or other characters.
- If you supplied a single file name on the command line, `grep` by default omits the filename (and first `:`) from the output lines.
- With the `-h` flag, it never shows file names.
- With the`-H` flag it always shows file names. Before this flag was added, Linux users often supplied `/dev/null` as an extra file name argument, so you may see this usage in old shell scripts or documentation.
- With the `-l` (letter l) flag, it shows only the file names and not the lines (or their line numbers, even with the `-n` flag).
- With the `-n` flag, it includes line numbers as in the example. By default it omits line numbers.
- Using a number as a flag (such as `-2`) shows that number of lines of context before and after the line that matches.

Omitting file names and line numbers can occasionally be useful. For example, to find the names of all packages imported into the current directory's Java programs, you might say

```
grep -h "import" *.java | sort | uniq
```

To copy all poems containing the word “had” to directory `backup`:

```
cp $( grep -liw had poems/* ) backup
```

The match is case-sensitive, which is appropriate for a case-sensitive language like C or Java. In natural language text, you might want to ignore case.

```
grep -i had *
```

finds all lines with the text `had` somewhere in the line, with any mix of cases (and anywhere within a word!). Thus it would find `had`, `Had`, `HAd`, but also `shade` and `Hades`. In a directory with several poems, it might find:

```
birches:You'd think the inner dome of heaven had fallen.
gentle:Because their words had forked no lightning they
mary:Mary had a little lamb,
mary:Mary had a little lamb
road:Had worn them really about the same,
road:In leaves no step had trodden black.
stop:And I had put away
summer:Nor shall Death brag thou wander'st in his shade,
wall:Not of woods only and the shade of trees.
```

Note the two lines each from files `mary` and `road`, and the match for `shade` in `summer` and `wall`. To limit it to entire words matching `had`, add the `-w` flag.

As with many Linux programs, if you don't pass any file names `grep` matches the standard input (which counts as a single input file, and thus prints no file names or line numbers if you omit the relevant flags).

```
history | grep -w cd
```

finds all "change directory" commands in the recent `bash` command history; this can be useful if you have switched to many different directories with long path names and want to avoid retyping the correct one (by using history substitution, Section 4.3 on page 31).

If you want to find several patterns at once, you can put them in a file, separated by newline characters, and use the `-f` flag, followed by the file name. Thus to find all occurrences of several literary characters in several text files, put the names

```
Ron
Hermione
Harry
Hagrid
```

in file `harryPotter` and invoke

```
grep -nH -f harryPotter *.txt
```

7.2 Caveat: Selecting a "Matcher"

`grep` has several different methods for matching patterns. Many punctuation characters such as `.*+()[]|{}^$` have special meaning to `grep`: they allow for more complex patterns called *regular expressions*, described in the next section. Historically, there have been several variants of regular expressions; to force the use of the version described here, use the `-E` flag.[29] Since by default `grep` matches regular expressions, if you want to just match these "special characters" as strings you can force it to treat them as ordinary characters by passing the `-F` flag.

For example, the `.` (period) character in a regular expression matches any character, so

```
    grep -wE sear.h *.c
```

searches all C source files for the `sear` library's header file `sear.h`, but also finds ordinary word `search`.

```
    grep -wF sear.h *.c
```

finds all occurrences of the exact string `sear.h`

The separate `fgrep` command is equivalent to `grep -F`.

The default pattern matcher (also selected by the `-G` flag) omits several of the features described in Section 8 or requires escape sequences (\) to force some punctuation to be interpreted as pattern operators. Use the `-E` flag!

8 Regular Expressions

Many Linux programs allow string pattern matching more complex than that of shell wildcards, and use *regular expressions* for constructing pattens. In particular, these notes describe `grep` (Section 7 on page 54), `sed` (Section 9 on page 62), and `awk` (Section 10 on page 67). Other Linux programs also use regular expressions; for example, `lex` generates lexical analyzers for compilers from regular expressions describing lexemes like identifiers and numbers. C has a regular expression library via

```
    #include <regexp.h>
```

Java has one via

```
    import java.util.regex
```

All examples of regular expressions in this section assume you use the `-E` flag for `grep` and equivalent flags for other programs such as `sed`.

Regular expressions in Linux are extensions of the language of the same name from automata theory: type 0 Chomsky grammars, which are equivalent

[29]Before `grep` had the `-E` flag, people used the separate `egrep` command; it is the same as invoking `grep -E`.

to finite state machines. In fact some implementations of regular expressions translate them into a compact state machine form. These notes do not assume you understand grammars or state machines; they are covered in courses such as CISC 223: Software Specifications.[30]

8.1 Minimal Regular Expressions

We first introduce the Linux regular expressions that correspond directly to those from automata theory:

- Normal characters (letters, digits, most punctuation) match themselves.
- A sequence of regular expressions $re_1 \dots re_n$ (concatenation) matches the first regular expression, followed by the second starting wherever the first finished, and so on to the last regular expression.
- The sequence *re*`*` (repetition) matches any number of occurrences of regular expression *re*, including zero. Thus `ab*c` matches `ac`, `abc`, `abbc`, `abbbc`, and so on.
- *re*`+` matches one or more occurrences. Thus `ab+c` matches `abc`, `abbc`, and so on, but not `ac`.
- The sequence `(`*re*`)` (grouping) matches the same thing as *re*, but treats it as a single unit. This allows you to use the other operators on complex regular expressions, not just single characters. The pattern `(abc)+` represents one or more occurrences (concatenations) of the sequence `abc`. Thus it matches `abc`, `abcabc`, `abcabcabc`, and so on. Without the parentheses it would represent `ab` followed by one or more occurrences of `c`: `abc`, `abcc`, `abccc`, and so on.
- The expression re_1`|`re_2 (alternation) means either regular expression re_1 or re_2 (but not both). Thus `a(b|c)d` matches the two strings `abd` and `acd`.

The precedence of the operators from highest to lowest is grouping, followed by repetition, followed by concatenation, followed by alternation. Thus

```
ab(c|d)|def*
```

[30]http://www.cs.queensu.ca/students/undergraduate/courses/desc/CISC-223.html

means “either `ab(c|d)` or `def*`” where the former means “`ab` followed by either `c` or `d`” and the latter means “`de` followed by any number of `fs`.” Thus `abc`, `abd`, and `deffff` all match, but `abddef` and `defdef` do not. The pattern *does* match the last three characters of `abddef`, plus the first three and last three of `defdef`, but not the whole strings.

Both binary operators (`|` and concatenation) are associative, so `((a|b)|c)` means the same thing as `(a|(b|c))`; thus the inner parentheses can be omitted, yielding `(a|b|c)`.

8.2 Simple Extensions

Linux regular expressions have a few extensions that significantly reduce how much one has to type.

- The character `.` (period) matches any character (except newline, `\n`). To match a period, use `\.` (as you should expect). It is equivalent to (c_1`|`...`|`c_n) for (almost) every symbol c_i in the ASCII character set. Thus `a..b` matches any four characters starting with an `a` and ending with a `b`.

- The sequence `x?` matches an optional `x`: zero or one occurrences.[31] It is equivalent to `(x|)`, where the absence of anything after the `|` means the empty string.

- A few “escape sequences” match specific characters; the most common are `\t` to match a tab character, and `\` followed by a character such as `*` or `+` or `?` that has a special meaning.

- The sequence

  `[` sequence of characters `]`

  matches a single occurrence of any of the listed characters – as usual, aside from a few special characters like `-` (minus sign or hyphen). It corresponds to an expression with the same set of characters separated by `|`. Thus `[abc]` is equivalent to `(a|b|c)`, and `[01234567]` matches any octal digit. Within `[]` the period matches itself, not any character; in fact most characters with special meanings outside `[]` match themselves.

[31] There is a syntax for specifying between `n` and `m` occurrences, described in Section 8.3 on page 60. Any realistic example would be especially horrendous to specify without such a special syntax.

- It is so common to want to represent a range of characters within `[]` that there is a special syntax: `a-z` matches any character in the ASCII character set between lower-case letters `a` and `z`, inclusive. If you want to include a hyphen in the set of matched characters, you can either quote it with a \ as usual, or make it the first character after the opening `[`.

Normally, Linux programs interpret regular expressions as matching a pattern anywhere in a line. Two special characters are "anchors": they ensure the pattern matches only at the start or end of a line. `^` (caret) matches the start of the line; `$` (dollar sign) matches the end of the line. Technically they "match the empty string" at those positions; they don't match the first or last character of the line. Thus

```
    grep -E '^[ \t]*$' *.c
```

finds all blank lines in all C source files in the current directory.

`\<` and `\>` are like `^` and `$`: they anchor a pattern ("match the empty string") at the beginning and end of a word, instead of the beginning and end of a line. Thus the two commands

```
    grep -wi fred *.txt
    grep -Ei '\<fred\>' *.txt
```

both find all examples of the word Fred (ignoring case distinctions) and omit longer words such as Frederick and Alfredo. This use of \ is the inverse of the usual one: it means "turn on the special meaning of `\<` and `\>`" instead of turning it off.

Examples of combining these features include:

- Matching a typical identifier in a programming language:

  ```
      [A-Za-z][0-9a-zA-Z_]*
  ```

  This means any ASCII alphabetic character, followed by any number of occurrences of alphabetical, numerical, and underscore characters.

- To ignore extra white space in the Java `import` search of page 54:

  ```
      ^[ \t]*import[ \t]+java.util
  ```

Aside: The shell wildcards `*` and `?` correspond to regular expressions `.*` and `.?` respectively.

8.3 Additional Regular Expressions†

The regular expressions of Sections 8.1 on page 58 and 8.2 on page 59 suffice for the most common usages of Linux tools. Here are some features used less frequently.

The expression `x{n,m}` matches *n* through *m* occurrences of `x`. Thus a FORTRAN I identifier, which had to be at most six characters long, would be `[A-Za-z][A-Za-z0-9]{0,5}` (an alphabetic, followed by zero to five alphanumerics). Omitting `,m` matches exactly `n` occurrences. Omitting just `m` (and leaving the comma) matches `n` or more. Thus `x+` is the same as `x{1,}`.

The expression `[^abc]` matches any character *except* `abc`. This means that `^` has two different meanings:

- Inside square brackets, it means to complement the set of characters matched; the pattern matches everything *except* the listed characters.

- Outside square brackets, it anchors the pattern at the start of the input line.

Thus the command

```
grep -nE '^[^#!]' *
```

shows all lines *starting with* (first `^`, at the start of the pattern) some character *other than* `#` or `!` (second `^`, inside the square brackets).

The patterns `[0-9]` and `[A-Za-z]` are so common that they have their own special notation: within square brackets, `[:digit:]` is equivalent to the first and `[:alpha:]` to the second; `[:alnum:]` matches both. Thus an ASCII identifier is

```
[[:alpha:]][[:alnum:]]*
```

Note the doubled brackets. In a programming language that allowed a dollar sign or an underscore in an identifier, other than in the first position, one would write

```
[[:alpha:]][$_[:alnum:]]*
```

To match just letters and underscores, use `[:word:]`.

In fact these bracket expressions are more general: they match any letters or digits of the "current locale." Thus in a Russian-locale Linux installation, `[:alpha:]` would mean a Cyrillic letter. Additional bracket expressions include:

- `[:lower:]` and `[:upper:]` – lower and upper case letters, respectively.

- `[:punct:]` – punctuation such as `!@#^&*(){}[];:'''<>?+-=_`.

- `[:ascii:]` – any ASCII character. `[:print:]` represents "printable" characters in the range 0x20-0x7E, while `[:cntrl:]` is the complementary set of ASCII "control characters" 0x00-0x1E and 0x7F.

- `[:blank:]` – space and tab. `[:space:]` represents any whitespace character, including line breaks.
- `[:xdigit:]` – any hexadecimal digit

Finally, one extension to regular expressions takes them beyond what a theoretician would mean by that term (Chomsky type 0 grammars) and allow some forms of context-sensitive pattern matching. $\backslash n$, where n is a digit between 1 and 9 inclusive, matches the exact text matched by the n^{th} group from the beginning of the pattern. Thus `([ab])c\1` matches `aca` and `bcb` but not `acb` or `bca`. `([[:digit:]])\1` matches a pair of identical digits. Back-references like this are especially useful to define replacements in editors such as `sed` (Section 9).

Group numbers are defined starting with the left parenthesis of the group. Thus in nested groups such as `ab(cd(ef)g)(hi)`:

- `\1` is `cdefg`
- `\2` is `ef`
- `\3` is `hi`

9 `sed`: Simple Editing of Text Streams

`sed` (`s`tream `ed`itor) reads a sequence of files (or the standard input) and produces an edited output stream. The two main forms are

```
    sed -r -e 'command' file(s)
    sed -r -f scriptFile file(s)
```

The former executes the given command for each line of each file, writing the result to standard output. The second interprets the script file as a program (according to the loop described in Section 9.1) and executes that program for each line of each file. The `-r` flag tells `sed` to use the extended regular expression syntax described in Section 8; you should always use it.

A simple example is

```
    sed -r -e 's/cat/dog/' someFile
```

This “substitute” command (`s`) causes `sed` to output an edited copy of `someFile` where the *first* occurrence of `cat` on each line is replaced by `dog`, even in the middle of a word. The line

```
    The Alcatraz catalog discusses cats.
```

becomes

```
     The Aldograz catalog discusses cats.
```
To make the `s` command apply to all occurrences of `cat` on a line, append the `g` flag:
```
     sed -r -e 's/cat/dog/g' someFile
```
gives
```
     The Aldograz dogalog discusses dogs.
```
To make it change the nth occurrence (if it exists), append a number:
```
     sed -r -e 's/cat/dog/3' someFile
```
changes the the 3rd occurrence, giving:
```
     The Alcatraz catalog discusses dogs.
```
A command can be preceded by one or two "addresses" governing which lines of the file it applies to. An address can be a specific line number or a regular expression. Addresses let you select a subset of lines on which to apply the command. For example:

- Suppose you know a file starts with five lines of "boilerplate" text whose content you know well, where the year 2012 needs to be updated to 2013:

  ```
  sed -re '1,5s/2012/2013/g' oldFile >newFile
  ```

- In some Linux programming languages, comment lines start with `#`. To change all occurrences of `cat` to `dog` on comment lines you would say:

  ```
  sed -r -e '/^#]/s/cat/dog/g' someProg >newProg
  ```

You would likely invoke these commands in shell scripts or `find` commands (Section 6) that apply them to several files.

The most common commands are[32]

- `p` – "print" the line (that is, write it explicitly to the output). Normally this would produce a duplicate line. It is most useful when you use the `-n` flag on the command line, which suppresses the default printing of each input line. You can simulate the command
  ```
       grep -E pattern file
  ```
  with
  ```
       sed -r -n '/pattern/p' file
  ```

[32]These are simplified descriptions that applies to basic use of `sed`. There are some subtleties with their exact meaning, covered in Section 9.2.

- `s/pattern/replacement/flags` – find the first occurrence of the pattern on each line and replace it with the given text. The two main flags are `g`, which applies the substitution to all occurrences of the pattern on each line, and `p`, which prints the line after doing the replacement (normally used with the `-n` command line flag).
- `d` – "delete" the line (that is, don't write it to the output).

Sometimes an input line can match a regular expression in more than one way, particularly when you use repetition (`+` and `*`). When such a pattern could match two different strings starting at the same place, `sed` takes the longer match. For example, given `sed -re "s/a.+b/Z/"` and input line `MaxxbxxbN`, the pattern could match either `axxb` or `axxbxxb`; `sed` takes the longer match, yielding an output line `MZN`. If you want to find out exactly how such matches behave, you should experiment.

Sometimes you want to make an edit where the replacement contains pieces of the original pattern match. To do so, in the pattern you use grouping to identify the parts that will be copied, and in the replacement you use *back references* (Section 8.3 on page 62) to copy them. Thus to swap the first and second "columns" of a tab-separated table, you could write

```
    s/^([^\t]*)\t([^\t]*)/\2\t\1/
```

The character `&` is a special back-reference meaning "the whole string matched by the entire pattern." Thus

```
    sed -re 's/\<cat|dog\>/the &/'
```

puts the word `the` and a space before each occurrence of the words `cat` and `dog`.

9.1 sed Scripts

With the `-f` flag you can supply a file of commands to be executed for each line of the input. In such a script any lines starting with `#` are comments. Regular command lines take the form

```
    [address [ ,address ]]command
```

The square brackets mean that the addresses are optional. If the second is omitted, it is taken to be the same as the first. Addresses are either absolute line numbers, `$` (meaning the file's last line), or a pattern (a regular expression). The command line means, roughly, "for all lines of the input between the first occurrence of the first address and the next occurrence of the second address, execute the command."

It is not quite accurate to say that `sed` commands operate on input lines. Technically, they operate on a special internal buffer called the "pattern space." An oversimplified description of how `sed` works is: For every input line,

Read the line into the pattern space.
Execute all commands in order, potentially modifying the pattern space.
Print the pattern space.

Because of the pattern space, an earlier `s`ubstitute command changes the context in which later ones are executed. Thus given script file[33]

```
s/day/night/g
s/Tuesday/Fri/g
```

and the input file

```
Today is Tuesday.
Tomorrow is Wednesday.
```

the output would be

```
Tonight is Tuesnight.
Tomorrow is Wednesnight.
```

With the commands in the reverse order, the output would be

```
Tonight is Fri.
Tomorrow is Wednesnight.
```

9.2 Advanced `sed`†

The main "execution loop" of `sed` is a little more complex than in the previous section. [34]

1. Start with an empty pattern buffer, considering it to be line number 0.
2. For each command introduce an "active/inactive" flag that governs whether to execute the command or not. If no address precedes the command, the flag is always on; otherwise it is initially off.
3. Execute any active commands with address `0` (those meant to be executed before reading the input).
4. Read the next (or first) line from the file into the pattern space.
5. Examine the next (or first) command line.

[33]I intend to replace or augment this with something more startling if I can think of it.
[34]***I need to verify some of the finer details of this loop***

6. If it is inactive and has a first address, test if the line matches the address. If so, make the command active. If not, go on to the next command (step 5).

7. If the command has a second address, test if the line matches it.

8. If the command is active, execute it. Commands affect the pattern space, not the original line.

9. If the second address matched the original line, make the command inactive.

10. Repeat starting with step 5 if there are more commands.

11. If there are no more commands, and there was no `-n` switch on the command line, execute a `p` command (print the modified pattern space).

12. Repeat starting with step 4 if there are more input lines.

There are several commands in addition to `s`, `p` and `d`. Some of them only allow zero or one addresses:

- `r`: Append the contents of a file to the pattern space.
  `r` *filename*

- `a`: Append several lines to the pattern space.
  `a\`
  line 1 \
  ··· \
  line n

  Each line but the last terminates with a backslash (an "escaped newline").

- `i`: Insert several lines at the beginning of the pattern space.
  `i\`
  line 1 \
  ··· \
  line n

- `q`: Quit the `sed` script (first printing the pattern space unless the original command line had the `-n` flag).

Some of these commands insert newlines into the pattern space. The `substitute` command can also add newlines, and match patterns that cross lines – but only within the pattern space; you can't match a pattern across multiple lines of the input file.

Other commands allow zero, one, or two addresses:

- `s`, `p`, and `d`, which apply to the whole pattern space.
- `w` *filename*: write the pattern space to the end of a file, creating it it necessary.
- {: Execute a group of commands.

  ```
  {
  several commands on separate lines
  }
  ```

  This might be used to apply several substitutions, but limit them to a region given by one or two addresses.

There is a special flag, `!`, which you can put between the addresses and the command. It means to execute the command for all lines that are *not* in the address range. Thus

```
sed -wre '/^#/!s/cat/dog/g'
```

changes the word `cat` to `dog` except on comment lines (which start with a hash mark).

There are many more facilities, including labels, conditional branches, commands that only operate up to the first newline in the pattern space, and an auxiliary buffer called the "hold space;" consult the `man` pages or a reference manual. However, if what you are doing becomes this complex, you should consider writing an `awk` script instead (Section 10).

10 `awk`: Programmable Editing of Text Streams

`awk` is a programming language for text processing; it combines the pattern matching facilities of `sed` with a C-like programming language. This section describes GNU `awk` (`gawk`). The usual means of invoking it is

```
gawk [ flags ] -f scriptFile filename(s)
```

You can also supply a program on the command line instead of a script file (using the option `--source 'program'`), but that is usually a bad idea unless the command is very simple. Unlike `grep` and `sed`, `awk` always uses the extended regular expressions of Section 8 on page 57.

`awk` is a line-oriented language; with few exceptions, most commands terminate at the end of the line unless you use a single `\` as the last character on the line. Furthermore, it is case-sensitive (like C, Java, and the Linux file system); two variables with different case are distinct identifiers. Thus `SomeVariable` and `SOmeVariable` are distinct; `awk` can't tell if the latter came from holding down the shift key slightly too long.

There are three kinds of lines in `awk`:

- Comments, which start with `#` (anywhere outside of string literals and regular expressions) and end with newline.
- Action statements, of the form

  ```
      pattern { action }
  ```

  where the action is normally a multi-line program, and the pattern is often a regular expression but can also be other kinds of pattern.
- *User-defined functions* (Section 10.5 on page 74), of the form

  ```
      function name ( parameter_1, ..., parameter_n ) {
          statements
      }
  ```

`awk`, like `sed`, is built around a standard loop that reads from an input stream and executes commands for each input line. Its basic operation is:

- Read a *record* from the input; normally this is the same as a "line" (ending with newline) but you can specify a different *record separator* (variable `RS`).
- Split the line into *fields* at each occurrence of a *field separator* (variable `FS`): normally, a sequence of spaces and tabs is a single separator.
- For each action statement, in order: if its pattern matches, execute it.

The most common patterns are

- A regular expression delimited by slashes (Section 8 on page 57), which matches input lines in the expected way.
- The word `BEGIN` or `END` (all capitals), which label actions to be done before reading the first line or after finishing executing commands for the last line.
- A boolean relational expression, which matches if the expression is true (non-zero).

10.1 Language Constructs

`awk` has constructs similar to those of C: expressions, assignments, `if/else`, `for`, `while`, `break` (ending the innermost loop), and `return` (from a function). There are also several statements concerning input and output (Section 10.2 on page 71). Two statements deal directly with the default loop that reads records from input files and performs actions.

- The `next` statement reads the next input record and goes back to the start of the sequence of commands in the script. Thus if you want to treat lines in the input file beginning with `#` as comments, you might include the action
  ```
  /^#/ { next; }
  ```
- `nextfile` stops processing the current input file, begins reading from the next, and goes back to the start of the sequence of commands in the script. Thus given the command line
  ```
  gawk -f script.awk file1 file2 file3
  ```
  the first execution of `nextfile` in `script.awk` causes the next input line to come from `file2` instead of `file1`; the second execution switches to `file3`, and the fourth executes the `END` commands and exits.

`awk` has all the arithmetic and logical operators you'd expect from C (including assignment operations like `+=`), plus `^` for exponentiation. Writing two expressions separated by a space (that is, with no operator between them) means string concatenation; most other operations on strings are built-in functions (Section 10.3 on page 72). The special operator

`string~pattern`

is true if the pattern matches the string; `!~` is true if the pattern doesn't match. The pattern can be a constant (surrounded by `/` characters) or a string expression.

Variables don't need to be declared; you just assign to them. If you read from an uninitialized variable, its value is the null string. This is common in scripting languages; in combination with the case sensitivity of variable names, this can make it difficult to detect bugs arising from misspelled identifiers or unitialized variables.

Variables are not typed. You can assign a string to a variable in one part of a program and an integer in another. `awk` implicitly converts between strings and integers when you use one in a context that expects the other. Floating point numbers are treated as integers if they represent the exact value of an integer. Figure 13 shows a short program and its output, illustrating assignment of several different types of value to the same variable.

Figure 13: `awk` Program With Several Value Types

```
# Assign different types of value to a variable.
# $Revision: 0.2 $
# $Id: varTypes.awk,v 0.2 2013/08/06 16:48:21 dalamb Exp $
BEGIN {
  var = 1+1; print var;
  var = var/3; print var;
  var = var*6.0+1.0; print var;
  var = "a string " var; print var;
  print 1e7;
  print 1e12;
}
```

Output

```
2
0.666667
5
a string 5
10000000
1e+12
```

There are many built-in variables; the most commonly used include:

- `$i`, the ith field (`$1` being the first field). `$0` means the whole record.
- `FS`, the string or regular expression used as a field separator on input. A single space character means any sequence of spaces and tabs.
- `RS`, the record separator for input files (newline by default)
- `OFS` and `ORS`, the field and record separators when printing to output files (see `print/printf` in Section 10.2).
- `NR`, the number of input lines read so far.
- `FILENAME` and `FNR`, the name of the current input tile and the current record number (normally the current line) within that file.
- `NF`, the number of fields in the current record.
- `RSTART` and `RLENGTH`, used with the `match` function (Section 10.3 on page 72).

Arrays are *associative*, similar to dictionaries in Python or Maps in Java: you can use any string as an “array index.”

```
phone["Smith"] = "555-1913";
dial = phone["Jones"];
```

The boolean operator

`*expression* in *array*`

is true if the array element *array*[*expression*] has been assigned a value. The `for` statement has a special form to iterate over all array indices:

`for` (*variable* in *array*) *statement*

Thus, to print a table of phone numbers (sorted in whatever order `awk` happened to store the array indices):

```
for(X in phone) printf "%s\t%s\n",X,phone[X];
```

If you want to eliminate an index from an array, you use the `delete` statement:

```
delete phone["Jones"]
```

If you leave out the square brackets and the index, it deletes the whole array.

10.2 I/O in awk

Unlike C, the I/O operations in `awk` are statements rather than functions.

The command

`getline [ *variable* ] [ < *file* ]`

sets the given variable to the next line from the given file; it returns 1 if successful, 0 for end-of-file, and -1 if there is an error (in which case it sets built-in variable `ERRNO`). If the variable is omitted, `awk` uses `$0` this also involves sedding `NF` and the `$i` variables. If the file is omitted, `awk` reads the next line from the main input stream (which also sets `NR` and `FNR`, and possibly `FILENAME`) but does *not* restart executing commands from the beginning. Thus if you like you can use `getline` to override the basic `awk` input/action loop.

The command

`print` [$expression_1, \ldots, expression_n$] [> *file*]

writes the list of expressions to the given file, separated by the value of `OFS`, and terminated with the value of `ORS`. If the expression list is omitted, it is taken to be `$0`. `printf` interprets its first parameter as a format string, similar to that of the `printf` function in C.

The first `print` or `printf` that writes to a file opens it for writing starting at the beginning of the file; thereafter they append to the file. Using `>>` appends to the file on the first `print` or `printf` instead of rewriting it.

Because of the special `/dev` “device” files within the Linux file system (Section 3 on page 8), you can print on `/dev/stderr` to produce error messages.

Thus for example:

```
     if (something bad happened) {
       printf "%s:%s:%s\nbad input\n",
         FILENAME, FNR, $0 >"/dev/stderr"
     }
```

prints an error message to the standard error stream; it includes the file name and line number where the error occurred, and the contents of the line being examined.

10.3 Built-In Functions

The following are the most commonly used string manipulation functions built in to `awk`:

`length` finds the length of a string.

```
len = length( [ string ] )
```

If the string is omitted, it uses `$0`. In recent versions of `gawk`, the length of an array is the number of indices; in older versions, applying `length` to an array was illegal.

`substr` takes a substring of a string expression:

```
substr( string, index [ , length ] )
```

returns the substring starting at the given index. The index is 1-origin (that is, the first character is at index 1, instead of 0 as in C). The result will be of at most the given length – shorter if there are fewer characters in the string than `i+`*`length`*`-1`. If the length is omitted, the result goes from the given index to the length of the string.

`match` finds the position of a regular expression in a string:

```
position = match(string, regular expression [,array ])
```

returns the position of the first match of the given regular expression in the given string, or zero if there is none. It sets `RSTART` to the index of the first matched character (the same as the result of the match) and `RLENGTH` to the length of the match. If the array argument is given, `match` assigns the strings matched by internal groupings – parts of the pattern delimited by parentheses. `array[i]` is equivalent to the `\i` back-reference in regular expressions and `sed` substitutions (Section 9 on page 64). Figure 14 shows how grouping, `RSTART`, and `RLENGTH` can be combined to "remove" parts of a matched string. The output consists of the substring up to the part of the string matched by the pattern, the "middle" part matched by the first (and only) grouping, and the substring after the end of the part that matches the pattern; this eliminates the bracketing substrings `left` and `right`.

Figure 14: Use of `match` to Remove Context

```
# Show use of match, RSTART, and RLENGTH $Revision: 0.1 $
# Removes some context around a pattern.
# $Id: match.awk,v 0.1 2013/08/06 16:45:12 dalamb Exp $
BEGIN {
  string = "start␣markleft␣some␣middle␣text␣rightmark␣end";
  if (match(string,"left␣(.*)␣right",parts)) {
    printf "%s%s%s\n",substr(string,1,RSTART-1),
      parts[1], substr(string,RSTART+RLENGTH);
  }
}
```

Output

```
start marksome middle textmark end
```

`split` splits a string into parts using a regular expression to delimit fields, and places each part in successive elements of an array.

`numParts = split(`*`string, array`* `[,` *`regular expression`*`] )`

It returns the number of fields found, or 0 if the string was empty. The default regular expression is `FS`, the field separator.

`sub` finds the first occurrence of a regular expression in a string variable and replaces it; this is equivalent to the `s` command from `sed`, without the `g` flag.

```
sub(regular expression, replacement string
    [ , string variable ])
```

If you omit the string variable, it is taken as `$0` (the whole record). To change *all* occurrences, use `gsub`, which takes the same parameters.

`index` returns the 1-origin index of the first occurrence of one string in another, or zero if there is no match.

`index(`$string_1, string_2$`)`

finds $string_2$ in $string_1$).

`toupper` and `tolower` take a single string parameter and returns a new string with the alphabetic characters converted to upper or lower case, respectively.

There are also numeric functions, bit-manipulation functions, and internationalization functions.

10.4 Command-Line Arguments

In addition to the `-f` *scriptfile* switch already mentioned, `awk` has many command line arguments, only a few of which I describe here.

Sometimes you want to pass arguments to your `awk` program instead of to `awk` itself. One way to do so is to use the `-v` flag.

```
gawk -v var=val -f script.awk someFile
```

sets the given variable to a specific value before `awk` starts executing the program in `script.awk` – even before the `BEGIN` pattern. Your `BEGIN` code can detect whether such variables are null and assign a default value. Thus given:

```
BEGIN {
  if (linePrefix="") linePrefix = "\t"
}
```

the command

```
gawk -v linePrefix="// " -f script.awk someFile
```

uses double-slash followed by a space as the "line prefix," whereas

```
gawk -f script.awk someFile
```

uses a tab.

Some options are preceded with a double hyphen. For example,

```
--field-separator pattern
```

sets the field separator variable FS. The default field separator breaks on every run of spaces and tabs. If one input file has fields with embedded spaces you could set FS to a single tab. If another has both spaces and tabs within fields, but doesn't have slashes, you could set the field separator to `/`. This particular option also has the single-hyphen form `-F` ***`pattern`***.

Some options have both a GNU and a POSIX form; `--option` is GNU, while `-W option` is POSIX.

- `--traditional` disables `gawk` extensions to the original UNIX `awk`.
- `--posix` disables a few more features to make `gawk` compatible with the POSIX standard.
- `--lint` prints warnings for program lines with unportable constructs.

10.5 User-Defined Functions

A user-defined function has the form:

`function` ***`name`*** `(` $variable_1, \ldots, variable_n$ `) {` ***`statements`*** `}`

A call looks like

`name` `(` $expression_1, \ldots, expression_m$ `)`

m can be less than n. Earlier I said that a function definition has a sequence of *parameters*, but the actual situation is slightly more subtle. By default all variables in `awk` are global, so any function can refer to variables assigned to by any of the actions in the program. When you call a function, values of all the variables named in the function definition[35] are saved, and restored to their original values when the function returns. Passing too many arguments is an error, but passing "too few" merely initializes the "missing" arguments to null. This feature can be used to define local variables. There is no semantic distinction between local variables and parameters – the "local variables" are just formal parameters for which no actual parameter is passed in the function call. There is a stylistic convention to add extra spaces or tabs in the function definition between the last parameter and the first local variable.

Within a function, `return` *value* returns to the caller with the given value as the result; if *value* is omitted (or if the function "falls off the end" of its statement list), the return value is undefined.

Figure 15 shows an `awk` program using a function to define processing to be done at the end of a collection of records, which must also be done at the end of the input. It detects the end of a "group" of records when the value of the first field changes and calls `endGroup` to print all the values associated with that label, separated by commas instead of newlines. At the start of the program, the "last label" is blank, so `endGroup` does nothing the first time the label "changes" (from the emtpy string to an actual label). At the end of the input (`END` pattern), the same processing is needed. The initialization of `FS` at the start of the program (`BEGIN` pattern) is needed to allow embedded spaces in values (or labels, for that matter), but the other initialization is purely for documentation purposes. `endGroup` has no local variables or parameters; all the variables it references are global. Given the input

```
someLabel       value1
someLabel       value0
otherLabel      firstVal
otherLabel      secondVal
        this will be treated as a comment
otherLabel      thirdVal
```

(where the runs of spaces indicate single tab characters) the script produces the output

[35]These might be called "formal parameters" in conventional programming language terminology.

Figure 15: Sample awk Program with User-Defined Function

```
# Generate comma-separated value list $Revision: 0.1 $
# $Id: commaValue.awk,v 0.1 2013/07/25 19:48:24 dalamb Exp $
# Input lines of the form
#   label <tab> value
# in sorted order by label are turned into
#   label <tab> value1, ... valueN
# Lines with missing labels are ignored.
BEGIN {
  FS = "\t";
  valList = lastLabel = "";
}
function endGroup() {
  if (lastLabel=="") return;
  printf "%s\t%s\n", lastLabel, valList
}
$1 == "" { next; }
$1 != lastLabel {
  endGroup();
  lastLabel = $1;
  valList = $2;
  next;
}
{ valList = valList ", " $2; }
END { endGroup(); }
```

Program to detect changes in the first field of a record. `endGroup` encapsulates end-of-group processing when a label changes or when the end of the input is reached.

```
someLabel        value1, value0
otherLabel       firstVal, secondVal, thirdVal
```

10.6 Some Longer Examples

Figure 16 shows an `awk` script that converts a file with many names on a line to one per line, reversing the order of given names and surnames. Given the input file

```
# this is a comment: it will be deleted
Ilke ten Boom
Marjorie Smith:Piet de Vos:Elsa von Braun
Piotr Ivanov:Simon van Dyke:Chen, Xiaoping
```

the command

```
gawk -f splitNames.awk splitNames.txt | sort -f >splitNameOut.txt
```

generates the output file

```
    Chen, Xiaoping
    de Vos, Piet
    Ivanov, Piotr
    Smith, Marjorie
    ten Boom, Ilke
    van Dyke, Simon
    von Braun, Elsa
```

Figure 16: awk Script to Reorder Names

```
# Convert "given-name surname" to "surname, given-name"
# paying attention to common multi-word  surnames.
# $Id: splitNames.awk,v 0.1 2013/07/30 16:38:11 dalamb Exp $
# Input file consists of lines of the form
#   name1 : ... : nameN
# $Revision: 0.1 $

# Setup
BEGIN {
  FS = ":";
  # specify name prefixes in an easy-to-write form
  namePrefixes = "de/von/van/ten";
  numPrefixes = split(namePrefixes, parts, "/");
  # convert to table form to use "in" operator
  for(i=1; i<=numPrefixes; i++) {
    prefix[parts[i]] = 1;
  }
}

# delete comment lines
/^#/ { next; }

# Actual conversion
{
  for(i=1; i<=NF; i++) { # for each field
    fullname = $i;
    # if name already has a comma it is in the correct form
    if (fullname ~ /,/) printf "%s\n",fullname;
    else {
        # split each full name into parts
        len = split(fullname,parts," ");
        surname = parts[len]; # last name
        if (len>2) {
          # check whether 2nd last part is one of the prefixes
          if (parts[len-1] in prefix) {
            # if so, construct the proper surname and
            # arrange to print one fewer given names.
            len--;
            surname = parts[len] " " surname;
          }
        }
        # print fullnames one per line
        # in format "surname, list of given names"
        printf "%s,", surname;
        for(j=1; j<len; j++) printf " %s",parts[j];
        printf "\n";
    } # if
  } # for
} # actual conversion
```

Part III
C Programming for Linux

This portion of the notes describes how to write system-level C programs under Linux. It currently assumes you are familiar with basic C programming; if you do not already know the language, you should consult the primary source: *The C Programming Language*, 2^{nd} edition, by Brian Kernighan and Dennis Ritchie. You must be able to wite C programs using:

- Types, declarations, expressions, pointers, structs, functions;
- The main constructs `if/else`, `for`, `while`, `break`, `return`;
- The standard library: string functions, `malloc/free`, `printf`, and `scanf`;
- File I/O: file descriptors, `fopen`, `read`, `write`, `fclose`, and the use of `errno` to indicate errors from system calls;
- Multi-file programs, header files, and the difference between declarations and definitions.

C was designed in the early 1970s at Bell Labs. It was a successor to a typeless language called B, which was in turn a successor to BCPL[36] (Basic CPL). CPL (Cambridge Programming Language), designed by D.W. Barron and Christopher Strachey the early 1960s, turned out to be difficult to implement, a major factor leading to BCPL. Strachey was also one of the key figures in developing *denotational semantics*, a mathematical means of describing the meaning of computer programming languages; it is covered in formal specifications courses such as CISC 465: Foundations of Programming Languages.[37]

Successors to C include C++ (taught in CISC 320: Fundamentals of Software Development),[38] and Java (taught in CISC 124: Introduction to Computing Science II).[39]

[36] http://en.wikipedia.org/wiki/BCPL
[37] http://www.cs.queensu.ca/students/undergraduate/courses/desc/CISC-465.html
[38] http://www.cs.queensu.ca/students/undergraduate/courses/desc/CISC-320.html
[39] http://www.cs.queensu.ca/students/undergraduate/courses/desc/CISC-124.html

11 Processes

Section 4.6 on page 36 introduced the idea of a *process* within the context of the shell. This section shows what happens at a lower level – what system library calls are involved, and the details of how to write C programs that create their own sub-processes.

You have already seen and used processes for foreground/background jobs, multiple programs with their own windows in a GUI interface, and multiple users on a single machine. Multiple processes give the illusion of multiple separate machines, but in reality one processor[40] switches rapidly back and forth between the separate processes.

11.1 What is A Process?

Basically, a process is a means for simulating multiple processors on a single processor (or simulating a large number of processors on a machine with fewer). Each process has its own version of:

- an *address space* – a section of computer memory, including space for its own executable code, stack (for procedure calls) and heap (for `malloc` and `free`).
- *program state* – information such as where its next instruction in the program will come from (the "program counter").
- *system state* – data Linux keeps outside the address space where the program can't accidentally change it, such as what files are open and where in the file the next `read` or `write` will take place.

A process can't change any memory or state allocated to a different process, nor can hardware errors (such as "segmentation faults") affect any other process.

Switching between processes – "context switching" – is very slow compared with normal program execution. Table 4 shows typical times for a typical machine as of this writing, translated to a human-comprehensible scale.

- *Processor cycle* is the rate at which a computer performs basic internal CPU operations; there can be several cycles for a typical assembly language instruction.

[40] Or a small collection of tightly-coupled processors on a multi-core machine.

Table 4: Process Timescale

Activity	**Actual Times**	**Scaled**
processor cycle	0.333 ns (3GHz)	1 sec
memory access	50-150 ns	2.5-7.5 min
context switch	3.5 μs	3.4 hr
disk rotational latency	8.3 ms (7200 RPM)	289 day
time slice quantum	100 ms	9.5 yr

Times for computer activities on typical mid-range personal computers, versus times at a human scale. Human reaction times (the fastest a human's nervous system can respond to any stimulus) are on the order of a few hundred milliseconds, roughly a factor of a billion (10^9) slower than current computers. All times are rough approximations based on Internet searches as of July 2013.

- *Memory access* is the time between the CPU's request for a word of dynamic random access memory (DRAM) and the time it arrives.
- *Context switch* is the time to switch from one Linux process to another.
- *Disk rotational latency* is the average time the CPU must wait before a "hard drive" disk rotates to the starting point of a block of information (one of several factors involved in time to wait for disk I/O). The highest-performance disks are about twice as fast.
- *Time slice quantum* is the typical time Linux runs one process before switching to another.

Section 4.6 on page 36 described several shell features related to processes: that commands run in separate processes; the concepts of foreground, background, and suspended jobs; and the commands `jobs`, `ps`, and `kill`.

11.2 The Process Manager

The process manager is the part of the Linux kernel that deals with processes: creating them, destroying them, and switching from one to another. It stops a process under several circumstances:

- The process has finished (such as by making an `exit` system call). In this case the manager has several cleanup tasks to perform.
- The process has requested a `sleep` – an amount of time to wait before it runs again.
- The process is waiting on something else – such as for another process to complete, or for input/output.
- A higher-priority process needs to interrupt. The details of this are beyond the scope of these notes.
- Its running time since its last "stop" exceeds a time limit (the *quantum* mentioned in Table 4).

If the computer has multiple processors, usually the manager allocates a process to each processor.

When one process stops, the manager resumes another one; if there are no other processes, it runs a special "idle" process.[41] When choosing a process to run, the kernel considers several factors:

- The process priority, inherited from the user's priority. Most users have the same priority, but the root user has a higher one. A process can voluntarily lower its priority.
- How long the process has been waiting since the last time it ran.
- How much processor time it has taken already.

11.3 Creating a New Process

Every process has a *process ID* – an integer unique to the process; these are the numbers you can see in the output from the `ps` command. Every process has a unique *parent* – the process that created it. The "init" process, the first one created after the system boots, is the ultimate ancestor of all processes; it is its own parent. The system call

```
    pid_t getpid()
```

from `<unistd.h>` returns a process' own ID; the call

[41] Technically this might be a special lightweight "process" called a "thread." An idle process can do anything, from infinitely looping to computing the digits of π – as long as it never blocks.

`pid_t getppid()`

returns the parent's ID. `pid_t` is from `<sys/types.h>`. The `fork` system call creates a new process:[42]

`pid_t fork();`

This creates a new process identical to the old one, with its own copy of all the process' state; there is no shared memory between the two. The only difference is that the return value is zero when `fork` returns in the child process, and the child's process ID when `fork` returns in the parent process. If the parent gets back a negative value, it means that the system was unable to create a child process.

Figure 17 on page 87 shows a simple program that creates several child processes and waits for all of them to finish (Section 11.5 discusses waiting for a child). Running it thrice with zero sleep time produces different output each time, because the scheduler happens to make different decisions about which child process to run first.

```
dal@linux3:~/notes/progs$ ./fork
Create 3 children, sleep 0
I am child 0 9068
I am child 1 9069
Finished child 0 9068 status 0
I am child 2 9070
Finished child 2 9070 status 512
Finished child 1 9069 status 256
dal@linux3:~/notes/progs$ ./fork
Create 3 children, sleep 0
I am child 0 9073
I am child 2 9075
Finished child 0 9073 status 0
I am child 1 9074
Finished child 2 9075 status 2
Finished child 1 9074 status 1
dal@linux3:~/notes/progs$ ./fork
Create 3 children, sleep 0
I am child 1 9087
I am child 2 9088
Finished child 1 9087 status 1
```

[42]The name comes from the image of a fork in the road, or the handle of a fork splitting into several tines.

```
    Finished child 2 9088 status 2
    I am child 0 9086
    Finished child 0 9086 status 0
```

With a positive sleep interval, the order would be fixed, since successive child processes wait longer before printing their messages.

11.4 Executing a Different Program

This simple form of process creation is useful in limited circumstances – primarily when there are several processors and some way to break up the parent's functionality into several independent steps, one per process. One way to do something different is to tell the system to load a different program to replace the current one; the child process would typically do this, leaving the parent to continue with whatever it was doing previously.

There are several ways to do this, each involving a system call whose name starts with `exec`. One simple version is

```
     int execvp(const char *file, char *const argv[]);
```

The `file` parameter is the path name of the program to run.[43] The `argv` parameter becomes the `argv` passed to the `main` procedure of the new program. It is an array consisting of a the same pathname string as the `file` parameter (the value of `$0` in a shell script), a list of strings for the "command line arguments" of the new program, and a null (0) to show the end of the list. It does not return if it succeeds; instead execution starts with the `main` procedure of the new program.[44] If it returns, `errno` indicates what caused the call to fail.

The arguments are passed unchanged to the new process; no shell is involved. Thus an `exec` call does not provide *any* of the facilities described in Section 4.7 on page 38: aliases, command line expansions/substitutions, wildcards, I/O redirection, pipes, or references to shell variables.

Figure 18 on page 88 shows a program that creates a child process, which in turn uses `execvp` to run a `find` command (Section 6 on page 49). It outputs:

```
    dal@linux3:~/notes/progs$ ./execvp
    Child 9161 invoking find . -name *.c -exec ls -l  ;
    -rw------- 1 dal student 1024 Jul 22 07:59 ./fork.c
```

[43] Or a text file whose first line starts with `#!`, as described in Section 5 on page 39, in which case the program's name follows the `#!`.

[44] Technically, the part of the standard C library that invokes the `main` procedure.

```
-rw------- 1 dal student 780 Jul 22 08:07 ./execvp.c
-rw------- 1 dal student 1247 Jul 22 08:10 ./sigpause.c
Finished child 9161 status 0
dal@linux3:~/notes/progs$
```

11.5 Parent/Child Interaction

Sometimes, as when the shell creates a foreground process, the parent has nothing to do but wait until the child terminates. This is what the `wait` system call from `<sys/wait.h>` is for:

```
int status;
pid_t child_pid = wait(&status);
```

This waits for *any* of the parent's children to stop; the status indicates which of several possibilities caused termination.`status` is set to the "exit status" of the child;[45] The return value is the process ID of the child that stopped. If you passed null (0) instead of the address of an `int`, it means you didn't care to know its status.

The slightly more general variant

```
pid_t child_pid = waitpid(pid, &status, options);
```

lets you wait for either a single child (with a positive `pid`), or all children (`pid = -1`), or any child in a particular *process group* (`pid < -1`, but process groups are beyond the scope of these notes). Typical values for options are 0 to just wait for a child to stop, or the constant `WNOHANG` to just check whether the child has finished, without waiting. The return value is either the process ID of the child, or zero if you use `WNOHANG` and no child has exited. If several children have finished, you get the process ID of only one of them. A call to `wait(&status)` is equivalent to `waitpid(-1,&status,0)`.

There are two special statuses for child processes:

- An *orphan* process is one whose parent has finished but which hasn't itself finished. This isn't necessarily a mistake; it's typical of server programs, which are meant to run "forever." Orphan processes become children of the special "init" process owned by the "superuser" mentioned in Section 3.5 on page 18.

- A *zombie* process is one that has stopped, whose parent hasn't stopped, but whose parent isn't waiting for it. It can be problematic since is still

[45]†"Exit status" consists of several parts encoding things like the parameter to `exit` and what signal, if any, terminated the process. To get the code from `exit`, one uses `WEXITCODE(status)` as in the example programs.

consuming limited system resources, such as memory space and a spot in the kernel's process table.

Normally when the parent terminates the system will clean up all its terminated child processes (leaving the still-running children to become orphans), so zombie processes aren't necessarily a problem for long – except when the parent is something like a login shell that typically runs for a very long time and creates very many child processes. To reduce the zombie problem, the parent gets a `SIGCHLD` *signal* (Section 12) when each child terminates.

Figure 17: Simple Example of `fork` and `wait`

```
/* Example of fork and wait, $Revision: 0.3 $
   $Id: fork.c,v 0.3 2013/07/29 16:34:08 dalamb Exp $ */
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>

#define NUM_CHILD 3
#define SLEEP_INTERVAL 0

/* Processing for the index'th child.
   Sleeps for a multiple of the given interval,
   then identifies itself and exits, reporting
   its number */
void child(int index) {
  unsigned int interval = index*SLEEP_INTERVAL;
  sleep(interval);
  printf("I am child %d %d\n",index, getpid());
  exit(index);
} /* child */

static pid_t children[NUM_CHILD];
/* Find the child with the given process ID */
int findChildIndex(pid_t child) {
  int i;
  for(i=0; i<NUM_CHILD; i++)
    if (children[i] == child) return i;
  return -1;
} /* findChildIndex */

/* Main program. Create several children and wait
   for all of them to finish */
int main() {
  printf("Create %d children, sleep %d\n",NUM_CHILD,SLEEP_INTERVAL);
  int i;
  for(i=0; i<NUM_CHILD; i++) {
    pid_t pid = fork();
    if (pid==0) { /* processing in child */
      child(i); exit(-1);
    }
    /* processing in parent */
    children[i] = pid;
  }
  int status;
  for(i=0; i<NUM_CHILD; i++) {
    pid_t child = wait(&status);
    printf("Finished child %d %d status %d\n",
           findChildIndex(child), child, WEXITSTATUS(status));
  }
  return 0;
} /* main */
```

Figure 18: Simple Example of `fork` and `execvp`

```
/* Example of fork and execvp, $Revision: 0.3 $
   $Id: execvp.c,v 0.3 2013/07/29 16:34:08 dalamb Exp $ */
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>

#define NUM_ARGS 9
static char *args[] =
  { "find", ".", "-name", "*.c",
    "-exec", "ls", "-l", "{}", ";",
    NULL };

/* Set up and call an execvp. */
void child() {
  /* Identify self and show execvp args */
  printf("Child %d invoking", getpid());
  int i;
  for (i=0; i<NUM_ARGS; i++) {
    printf(" %s", args[i]);
  }
  printf("\n");
  /* force buffered output to terminal before invoking exec */
  fflush(stdout);
  execvp(args[0], args);
  exit(1); // execvp failed
} /* child */

int main() {
  pid_t pid = fork();
  if (pid==0) child();
  int status;
  pid_t child = wait(&status);
  int ret = WEXITSTATUS(status);
  printf("Finished child %d status %d\n",
         child, ret);
  return ret;
} /* main */
```

12 Signals

The example of the previous section didn't have any communication between parent and child processes. One simple way but limited way for processes to send information to each other is via signals. Signalling a process can interrupt what it is doing, and gives it a small piece of information (the *signal type*, a small integer). The various signals have names starting with `SIG` and are defined in the `<signal.h>` library file.

Some signals are reserved for the kernel, while others can be uses by ordinary processes. A process can catch many of the possible signals and take some action in response, but a few (such as `SIGKILL`) can't be caught.

Normally a process can only send signals to processes owned by the same user, but the superuser (and thus kernel processes like `init`) can send signals to any process.

Within a C program, you can issue signals via the procedure

```
kill(process ID, signal number);
```

You can send yourself a signal via

```
raise(signal number);
kill(getpid(), signal number);
```

which are equivalent to each other.

12.1 Common Signals

The `kill` command of Section 4.6 on page 37 actually sends a signal to the process you designate. If you designate a job instead of a process, the signal goes to the top-level (often, the only) process of the job. By default it sends `SIGTERM`, which normally terminates the process – but the process can use a *handler* (Section 12.2 on page 90) to catch the signal and ignore it or clean up its resources before exiting. Thus the three commands

```
kill 40137
kill -TERM 40137
kill -15 40137
```

all send `SIGTERM` to the process with ID 40137. The commands

```
kill -KILL 40137
kill -9 40137
```

both send the `SIGKILL` signal, which kills the process immediately (not giving it a chance to catch and possibly ignore the signal).

Four other signals are reasonably common.

- `SIGCHLD` means that a child process has terminated (Section 11.5 on page 86). By default it is ignored.
- `SIGINT` by default terminates the process like `SIGTERM`; it is what the system sends when you type control-C. A program might catch the signal (Section 12.2) and ask the user for confirmation before quitting.
- `SIGTSTP` suspends the process; it corresponds to control-Z and can be caught.
- `SIGSTOP` can't be caught; it pauses the process until the next `SIGCONT`.
- `SIGUSR1` and `SIGUSR2` have no built-in meaning, and by default kill the process. They can be used to experiment with signalling and catching signals.
- `SIGHUP` initially meant "signal hangup" – that the modem associated with a terminal had disconnected, often because the user hung up the phone connected to the modem. If your process is in the foreground, the sensible thing is to terminate it. However, it can catch the signal and do something else. A common convention is for a long-running background process (not connected to a terminal), like a server, to reinitialize itself, perhaps by re-reading a potentially changed initialization file.

A process can use the `SIGALRM` signal to interrupt itself after a time interval.

```
alarm(seconds);
```

sends the process `SIGALRM` in the given number of seconds. It is different from `sleep` because the process continues executing, not waiting for the interval to expire. `alarm(0)` turns off the alarm timer.

`kill -l` (lower-case L) lists all the signal names and their numbers.

12.2 Signal Handlers

A program can set up a *signal handler* for each signal it's allowed to catch. You can have one handler for all your signals, or separate ones for different signals. A signal catcher has the general form

```
void catcher(int signum) {
    handle the signal
    possibly differently for each signal
} /* end catcher */
```

You tell the kernel which catcher to use for what signal with the `signal` function from `<signal.h>` (which also defines all the signal names). It takes two parameters: a signal number, and the name of the catcher. Thus

```
void reinitialize(int signo) {
   redo whatever the process did when it started
} /* reinitialize */
void cleanup(int signo) {
   close files, etc.
} /* cleanup */
...
signal(SIGHUP, reinitialize)
signal(SIGTERM, cleanup)
signal(SIGINT, cleanup)
```

The handler can inspect the `signo` parameter and perform different actions depending on which signal it got, or you can have separate handlers for each signal and ignore `signo`, or you can have some mixture of both approaches.

When the handler returns to its "caller," processing resumes wherever the program was executing before the signal. The only way to tell the rest of the program that something has happened is to set some global variable, which the normal code can examine when convenient. Thus for example

```
int INTsignaled = 0;
void handleINT(int signo) { INTsignalled = 1; }
...
signal(SIGINT,handleINT);
while (...) {
  something that shouldn't be interrupted
  if (INTsignaled) break;
} /* while */
normal termination
```

There are two default handlers: `SIG_IGN` ignores the signal and `SIG_DFL` restores the default action for the signal.

On some systems, after you catch a signal, it is reset back to the default action. On the CASlab systems this currently happens only if you use the `-ansi` compiler flag. To be safe and portable each handler should call `signal` again to re-enable catching whatever signal invoked it.

To wait for a signal to arrive, the process can invoke `pause()`, which is the equivalent for signals of `wait` for child processes. When a signal arrives, the catcher (if any) is invoked, and when it returns the program continues after the call to `pause`.

Figure 19 on page 95 shows a simple program that creates several child processes and waits for all of them to finish via `pause` and a handler for `SIGCHLD`; contrast it with Figure 17 on page 87. There are several subtleties about this program, which you can discover by running it and sending it various signals.

- The main program sets up handlers before using `fork` to create the child processes. This means that each child has the same handlers as the main program, albeit in separate address spaces.
- `SIGTERM` and `SIGINT` are sent to the main process and all its children, so each responds to it independently.
- The `sleep` system call wakes up early if the process receives a signal, so if you type a control-C, all the children finish quickly instead of waiting for their time interval to expire.

The following shows one example, sending SIGTERM to the parent process after one child finishes. I added the line numbers for ease of reference:

```
 1 dal@linux3:~/notes/progs$ ./sigpause&
 2 [2] 9173
 3 dal@linux3:~/notes/progs$ 9173 creating 3 children, sleep 5
 4 I am child 1 9175
 5 I am child 2 9176
 6 I am child 0 9174
 7 jobs
 8 [1]+  Stopped                 emacs makefile
 9 [2]-  Running                 ./sigpause &
10 Child 0 9174 done
11 Process 9173 caught 17: child done, 2 left
12 dal@linux3:~/notes/progs$ kill %2
13 Process 9173 caught 15: terminating 9173 with 2 children left.
14 Process 9175 caught 15: terminating 9175 with 3 children left.
15 Process 9176 caught 15: terminating 9176 with 3 children left.
16 [2]-  Exit 10                 ./sigpause
17 dal@linux3:~/notes/progs$
```

- On line 1, I start the program in the background; line 2 says it becomes job 2, process 9173.
- Three child processes start on lines 4-6
- On line 7, I type the `jobs` command to the prompts of line 3.
- On line 10, Child 0 finishes, and on line 11 the parent catches signal 17, `SIGCHLD`.
- On line 12 I send `SIGTERM` (the default) to "job 2," all three remaining processes. Each catches it: the parent on line 13, child 1 on line 14, and child 2 on line 15.
- Line 16 is the shell reporting that job 2 has exited with status 10, the argument to `exit`.

Notice that the children show "3 children left" when terminated. That is because the `fork` call makes an exact copy of the parent when it spins off each child process, which means they have copies of exactly the same data (`numChildren`) and signal handlers as the parent. A slightly more complex example should have the child processes setting up their own handlers when they start, with their own `SIGINT` and `SIGTERM` handlers to produce slightly different messages that don't refer to non-existent children.

When I send a signal to a child process, only that one responds:

```
 1 dal@linux3:~/notes/progs$ ./sigpause&
 2 [2] 9228
 3 dal@linux3:~/notes/progs$ 9228 creating 3 children, sleep 5
 4 I am child 1 9230
 5 I am child 2 9231
 6 I am child 0 9229
 7 Child 0 9229 done
 8 Process 9228 caught 17: child done, 2 left
 9 kill -INT 9231
10 Process 9231 caught 2: ignored 2
11 Child 2 9231 done
12 dal@linux3:~/notes/progs$ Process 9228 caught 17: child done, 1 left
13 Child 1 9230 done
14 Process 9228 caught 17: child done, 0 left
15
```

```
16 [2]-  Done                    ./sigpause
17 dal@linux3:~/notes/progs$
```

- Lines 1-8 are similar to lines 1-11 of the previous example.
- On line 9 I reply to the prompt of line 3 by sending `SIGINT` (code 2) to child 2. On line 10 it catches and ignores the signal, but since the signal interrupts the `sleep`, child 2 finishes immediately anyway (line 11). Had I written more complex code in the `child` function, such as looping a few times to repeat the `sleep`, and printing a message just after each sleep, you would have seen that it managed to continue after the interrupt.
- On lines 12-14 the remaining child finishes and the parent responds to `SIGCHLD`.

It would make a good exercise to modify `sigpause.c` to:

- Use different signal handlers in parent and child. Note: the parent still needs to set up the `SIGCHLD` handler before creating any children.
- Print a message when the `sleep` command wakes up showing the time elapsed since the call, to show that it terminates early when interrupted.

Figure 19: Simple Example of `SIGCHLD` Handler

```c
/* Example of signal handling and pause, $Revision: 0.3 $
   $Id: sigpause.c,v 0.3 2013/07/29 16:34:08 dalamb Exp $ */
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_CHILD 3
#define SLEEP_INTERVAL 5

/* Index'th child process identifies itself before and after
   it sleeps for a multiple of the interval, then exits. */
void child(int index) {
  unsigned int interval = (index+1)*SLEEP_INTERVAL;
  printf("I am child %d %d\n",index, (int) getpid());
  sleep(interval);
  printf("Child %d %d done\n",index, (int) getpid());
  exit(index);
} /* child */

int numChildren = NUM_CHILD;

/* Handler for all signals. Ignore most, handle
   SIGCHLD and SIGTERM */
void catcher(int signo) {
  printf("Process %d caught %d: ", (int) getpid(), signo);
  if (signo==SIGCHLD) {
    numChildren --;
    printf("child done, %d left\n",numChildren);
  } else if (signo==SIGTERM) {
    printf("terminating %d with %d children left.\n",
          (int) getpid(),numChildren);
    exit(10);
  } else printf("ignored %d\n",signo);
  signal(signo,catcher); /* set up same handler again */
} /* catcher */

/* Set up signal handler, create several children
   and wait for appropriate number of SIGCHLD signals. */
int main() {
  signal(SIGCHLD,catcher);
  signal(SIGINT,catcher);
  signal(SIGTERM,catcher);
  printf("%d creating %d children, sleep %d\n",
         (int) getpid(), NUM_CHILD, SLEEP_INTERVAL);
  int i;
  for(i=0; i<NUM_CHILD; i++) {
    pid_t pid = fork();
    if (pid==0) {
      child(i); exit(-1);
    }
  }
  while(numChildren>0) pause();
  return 0;
} /* main */
```

13 Pipes

Section 4.2 on page 28 describes what pipes do in the context of the shell; this section shows you how to write C programs that create and use pipes. You must already understand Chapter 8 Sections 1-3 of Kernighan and Ritchie, covering low-level I/O using file descriptors, `open`, `close`, `read`, `write`, and the `errno` error-reporting variable. You should also be familiar with Sections 11 and 12 concerning processes and signals.

There are two sorts of pipes: those meant for short-term communication among parent and child processes (such as those created for the `|` operator by a shell), and longer-term file-like *named pipes*.

13.1 Basic Pipes

A pipe is a stream with both input and output "ends." Normally one process writes to one end and another reads from the other.

```
int pipe(int pipe_fd[2]);
```

from `<unistd.h>` creates both ends – the read end in `pipe_fd[0]`, and the write end in `pipe_fd[1]`. After creating the pipe, the process would use `fork` (Section 11.3 on page 82) to create a child process. Since both processes are initially identical (aside from the return value from `fork`), both have accecss to the same file descriptors. One would read from the first file descriptor (using the low-level `read` system call), and the other would write to the second (using the low-level `write` system call). Alternatively, the parent could create two different children that would communicate with each other.

This kind of pipe is normally implemented by the kernel as a fixed-size buffer in memory, so it is much faster than passing through the file system (see activity timings in Table 4 on page 81). When the buffer fills, the writer blocks waiting for free space; when it empties, the reader blocks waiting for content. Since the process manager (Section 11.2 on page 81) switches processes when the current one blocks, on a single processor the two processes take turns executing (along with any other processes in the system at the time).[46] Since the process manager also switches processes when a higer-priority one needs to run, and when a process' running time exceeds the quantum, the two can switch while the buffer is only partly full. Buffers will be at least 512 bytes on any POSIX-compliant UNIX system; on CASlab they are currently 2^{16} bytes (64 Kb).

[46]On a multiprocessor, the two can work in parallel when not waiting for pipe I/O.

Recall that a call to `read` from a normal file "blocks" when waiting for the disk to spin to the right position and transfer data. With a pipe, a read blocks when there is no data in the buffer *and* it is possible for some other process to eventually write to it. What this means is that there must be some other process with a file descriptor for the same pipe opened for writing. If there is no such process, the `read` will return 0, meaning "end of file." When you first create a pipe, both ends of the pipe are open, one for reading and one for writing, so if the reading process goes first it will block.

Figures 20 and 21 on pages 101 and 102 show a simple use of a pipe. The parent writes its command-line arguments to the pipe; the child reads from the pipe and writes the results to standard output, inverting the case of all letters. To show exactly how the characters are buffered, the parent puts double-colons (`::`) between segments it writes (its arguments), and the child puts square brackets (`[]`) around segments it reads. Debugging messages from both parent and child processes go to `stderr`, which appears immediately on the terminal. Normal output from the child process goes to standard output.

The child closes the output end of the pipe. When the child starts, each process has open file descriptors for both ends of the pipe; if the child didn't close the write end, there would *always* be a file descriptor open for writing, and the child would hang on its last read when the parent closed its own copy of the write descriptor.

Running the program, directing standard output to a file, yields:

```
dal@linux3:~/notes/progs$ ./caseEcho Some argUMenTS to be Inverted >ceout
    Child 9390 maxLen 10
    Writing './caseEcho'
    Writing 'Some'
    Just read 10 characters
    Writing 'argUMenTS'
    Just read 6 characters
    Writing 'to'
    Writing 'be'
    Writing 'Inverted'
    Just read 10 characters
    Just read 10 characters
    Just read 9 characters
    dal@linux3:~/notes/progs$ cat ceout
    [./CASEeCHO][::sOME][::ARGumENt][s::TO::BE:][:iNVERTED]
    dal@linux3:~/notes/progs$
```

There are several things to note about the output:

- The terminal output of `caseEcho` all results from printing to `stderr`.
- The parent's "writing" lines (corresponding to individual calls on `write`) are interleaved with the child's "echo" lines (corresponding to individual calls on `read`).
- The `read` calls yield varying number of bytes, often not the 10 requested.
- consequently, the arguments are echoed in different chunks from those you'd expect from the `write`s.

13.2 Multiple Readers and Writers†

It is possible to have multiple readers and writers for a single pipe, but interaction can be tricky because of the way pipes are buffered. Suppose a multiple-process application considers a "record" to be the data a writer sends in one operation, such as a line terminated by newline. On Linux the low-level `read` operation works on streams of bytes, with no way to detect internal formatting such as "end of record" until the process has potentially read part-way into the next "record." Furthermore, a `read` isn't guaranteeed to read exactly the number of bytes requested (as seen in the previous section); it could read fewer depending on how much data is left in the buffer. Thus multiple readers are likely to get partial or overlapping information. Because of the unpredictability of which process the manager will choose to run next, it is also possible for one reader to *starve* the others: reading all the input before the others get any data.

One saving grace of the way pipes work is that writes of less than a certain size[47] are required to be *atomic*. This means that the write must finish before anyone else can write to the same pipe. If the buffer fills up before such a "small" write finishes, the only legal sequence of events is for some reader to partly empty the data, followed by the partial `write` finishing. All other potential writers are blocked during this time. However, writes of larger amounts of data may be split up and interleaved.

Instead of having multiple writers for one pipe, you can create one pipe per writer; it is possible for a single reader to wait for any of these pipes to have data. However, this requires mechanisms beyond the scope of these notes; consult the manual entries for `select` and `pselect`.

[47]At least 512 bytes for POSIX; 4 Kb on Linux.

The fundamental ideas of coordinating readers and writers are taught in courses about concurrent programming, such as CISC 324: Operating Systems.[48]

13.3 Named Pipes (FIFOs)

A named pipe, or FIFO (First In, First Out) is a special kind of file, stored in a directory, and thus has a name that multiple programs can look up. You create a FIFO with the system call

```
int mkfifo(pathname, permissions)
```

from `<sys/stat.h>`. The name and permisions are just what you'd expect for creating any file. A return value of -1 means that an error occurred, and `errno` gives the error code. `EEXIST` means there is already a file of that name. The pathname can then be `open`ed for reading or writing as with a normal file.

In the shell, the equivalent of the `mkfifo` system call is the pair of commands:

```
mkfifo pathname
chmod permissions pathname
```

Once opened, a FIFO behaves like a normal pipe as described in Section 13.1. Figure 22 on page 103 shows an example of FIFO behaviour via shell commands; I added line numbers by hand. The first 10 lines show two input files. The FIFO is created on line 11, and line 12 spawns a pair of background processes (connected by a basic, unnamed, pipe) that `sort` input from the FIFO and run the results through `uniq` to eliminate duplicate lines. Since no one has opened the FIFO for writing yet, on its first `read` it pauses waiting for some process to do so. Line 14 creates a background process that writes the contents of `source1` to the FIFO and waits for more input from the terminal (`-` argument to `cat`); without this wait, the `sort|uniq` job would terminate as soon as its only input source (this first `cat` process) finished writing. Line 16 spawns a second `cat` procecss that outputs the contents of `source2` to the FIFO and terminates. Line 19 resumes the first `cat` process; line 20 is the shell reporting what job was resumed. Lines 21-22 are new input; in between lines 22 and 23 I typed a control-D to terminate this input. After this, there are no more writers for the FIFO, so it returns end-of-file to the `sort` process' last `read` operation . `sort` then performs its work, outputting results to `uniq`, which prints on standard output; lines 23-30 are the results. Lines 21-33 are

[48]http://www.cs.queensu.ca/students/undergraduate/courses/desc/CISC-324.html

output from the shell after all jobs finish.

Figure 20: Pipe Example (a): Case Inversion of Program Arguments

```
/* Use a pipe to invert the case of letters in program arguments. $Revision: 0.2 $
   $Id: caseEcho.c,v 0.2 2013/07/29 16:34:08 dalamb Exp $ */
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <sys/wait.h>
#include <string.h>
#include <ctype.h>

int fd[2]; /* pipe file descriptors, 0 for read, 1 for write */

/* Error reporting */
void showError(char *msg) {
  fprintf(stderr,"%s errno %d (0x%x): %s\n",
          msg, errno,errno,strerror(errno));
  fflush(stdout); fflush(stderr);
} /* showError */

/* Parent writes to the pipe */
void parent(int argc, const char* argv[]) {
  int i;
  for(i=0; i<argc; i++) { /* write each argument */
    fprintf(stderr,"Writing '%s'\n",argv[i]); fflush(stderr);
    if ((i>0 && write(fd[1],"::",2) < 0) /* omit :: for arg 0 */
        ||
        write(fd[1],argv[i],strlen(argv[i])) <0) /* write argument */
      { /* one of the writes failed */
        fprintf(stderr,"Write error %s\n",strerror(errno));
      } /* if */
    fflush(stderr);
  } /* for */
  close(fd[1]); /* done with writing */
  /* wait for the echoing child to finish */
  int status;  pid_t pid = wait(&status);
  if (pid==-1 && errno!=ECHILD)
    fprintf(stderr,"Wait error %d (0x%x): %s\n",
               status, status,  /* show full status, not just exit code */
               strerror(errno));
} /* parent */
```

Figure 21: Pipe Example (b): Case Inversion of Program Arguments

```
/* Child reads from the pipe and echoes, inverted, to stdout */
void child(int maxLen) {
  fprintf(stderr,"Child %d maxLen %d\n", (int) getpid(), maxLen);
  close(fd[1]); /* we'll never write to it */
  char *buf = malloc(maxLen+2);
  while(1) {
    int len = read(fd[0],buf,maxLen);
    if (len<=0) {
      if (len==0) { printf("\n"); fflush(stdout); exit(0); }
      else { showError("Read"); exit(errno); }
    }
    fprintf(stderr,"Just read %d characters\n",len);
    fflush(stderr);
    putchar('['); int i;
    for(i=0; i<len; i++) {
      char c = buf[i];
      if (isalpha(c)) c = islower(c)?toupper(c):tolower(c);
      putchar(c);
    } /* for */
    putchar(']'); fflush(stdout);
  } /* while */
} /* child */

/* Creat the pipe and child process */
int main(int argc, const char* argv[]) {
  if (pipe(fd)<0) {
    showError("Can't open pipe");
    exit(1);
  }
  int maxLen = 0; int i;
  for(i=0; i<argc; i++) {
    int len = strlen(argv[i]);
    if (len>maxLen) maxLen = len;
  }
  pid_t childID = fork();
  if (childID==0)
    parent(argc,argv);
  else if (childID>0)
    child(maxLen);
  else
    fprintf(stderr,"Error from fork: %d %s\n",
            (int) childID, strerror(errno));
  fflush(stdout);
  exit(0);
} /* main */
```

Figure 22: FIFO Behaviour Via Shell Commands

```
dal@linux3:~/notes/progs$ cat source1
This is a sample.
It will be sorted.
Then run through uniq.
So the next line will be deleted.
This is a sample.
But this line won't.
dal@linux3:~/notes/progs$ cat source2
This is a sample.
It is in a separate file.
dal@linux3:~/notes/progs$ mkfifo FIFO
dal@linux3:~/notes/progs$ sort <FIFO | uniq &
[1] 18091
dal@linux3:~/notes/progs$ cat source1 - >FIFO&
[2] 18092
dal@linux3:~/notes/progs$ cat source2 >FIFO

[2]+  Stopped                 cat source1 - > FIFO
dal@linux3:~/notes/progs$ %2
cat source1 - > FIFO
This is a line typed at the terminal.
After control-D the sort will run.
dal@linux3:~/notes/progs$ After control-D the sort will run.
But this line won't.
It is in a separate file.
It will be sorted.
So the next line will be deleted.
Then run through uniq.
This is a line typed at the terminal.
This is a sample.

[1]+  Done                    sort < FIFO | uniq
dal@linux3:~/notes/progs$
```

14 The make Command

Many of the programs you write for introductory courses consist of a single source file and produce a single product, an executable program. When problems become significantly more complex, such as when you have many files to compile, building your program or rebuilding it after you change something can become complex as well.[49] `make` is a tool for describing such *build processes*, allowing a single command to rebuild an entire system.

14.1 Build Processes

When you start programming in C or any other language, you typically build your executable program in a single step, perhaps by issuing a shell command:

```
gcc -o myProg myProg.c
```

If your usage of `gcc` becomes more complex, you may find yourself adding command line options, such as `-W` to specify warnings to check for, or `-D` to specify variable definitions. Since it is tedious and error-prone to have to retype such options every time, you might put your command in a shell script:[50]

```
dal@linux3:~/notes/progs$ ls -lt fork*
-rw------- 1 dal student 1383 Jul 29 10:31 fork.c
dal@linux3:~/notes/progs$ cat recompile
# recompile sample program fork.c
gcc -Wall -DVARIANT=Intel -o fork fork.c
dal@linux3:~/notes/progs$ ./recompile
dal@linux3:~/notes/progs$ !ls
ls -lt fork*
-rwx------ 1 dal student 8733 Aug  3 09:27 fork*
-rw------- 1 dal student 1383 Jul 29 10:31 fork.c
dal@linux3:~/notes/progs$
```

If you want to specify options for the linker as well as the compiler, you might split this into two lines to keep straight which options apply to what utility

```
# recompile sample program fork.c
```

[49]Java is a *partial* exception, with its own built-in `make`-like process. It often (but not always) requires only a single compile command to rebuild a system; see Section 14.5 on page 113.

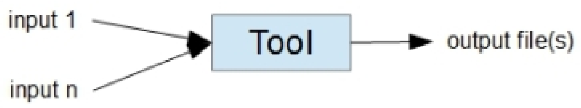
[50]It isn't necessary to know what the individual flags to `gcc` do – just that there are several of them.

```
    gcc -Wall -DVARIANT=Intel -c fork.c
    gcc -lsomeLib -o fork fork.o
```

Each of these is a simple example of a *build process*: the steps required to turn at set of inputs into a set of outputs.

Figure 23 shows an abstraction of what goes on in a single step of a build process. An input or output is normally a file; Section 14.4 on page 110

Figure 23: Abstraction of a Build Step



A step runs one "tool" taking several inputs and producing an output. Some steps may produce more than one output.

discusses a few other things that might constitute "inputs." The output of one step might become the input of another; for example, each `.o` output of a compilation step becomes an input of the final linking step.

Consider the program summarized in Table 5: an oversimplified simulation of a set of elevators, perhaps a trivial subset of the old SimTower® PC game. One build process for this program is:

```
  gcc -c random.c
  gcc -c policy.c
  gcc -c hardware.c
  gcc -c elevator.c
  gcc -c input.c
  gcc -o elevator elevator.o hardware.o random.o policy.o input.o
```

There are several things to note about this example. This is only one possible script, since the first four steps could be done in any order. However, the fifth step *must* be last since it depends on the outputs of each of the other steps.

Each of the compilation steps has some *implicit* inputs: the header files `#included` in them. A simple script file doesn't record these dependencies, which can become important when you need to figure out what to recompile after a change. Suppose `policy.c` and `policy.h` change, perhaps to add a new field in some `struct`. The only steps that need to be rerun are the compilations of those files that `#include policy.h` (in this case, just `policy.c`

Table 5: Modules for a Simple Elevator Simulation

Module	**Purpose**
`elevator.c`	Main program; calls functions from other modules
`hardware.c`	Simulation of buttons and motors and how to get safely from one floor to another (definitions)
`hardware.h`	declarations for `hardware.c`
`policy.c`	Policies for which floor to select next given current state of elevator and buttons (definitions).
`input.c`	Input functions (definitions). Generates a sequence of timed "button pushes" either by reading a file or by using random numbers or by some combination.
`input.h`	Input functions (declarations)
`policy.h`	declarations for `policy.c`
`random.c`	Random number functions (definitions)
`random.h`	Random number functions (declarations)

and `elevator.c`), plus the final linking. Rerunning the other steps is a waste of resources.

The `make` program lets you describe build processes in a *makefile*, which guides it in running the minimum number of steps after you change a source file. You can think of `make` as a tool that takes many inputs: all your source files, plus the `makefile`. It can produce many outputs: for example, all your `.o` files and the final executable program.

14.2 Basics of `make` and `makefiles`

Figure 24 shows a `makefile` for the system of Table 5. The first three lines define *variables* – similar to variables in a shell. They are used to avoid having to type the same string multiple times, and to ensure that when some aspect of the build process needs to change, they change consistently throughout the build process. In this example, `CC` says which compiler to use, `CFLAGS` says which command line options to pass to the compiler, and `OFILES` lists all the object files that make up the executable program.[51]

[51]You can also define variables on the `make` command line, using the same syntax. This overrides definitions of the same variables in the `makefile`. Thus `make CFLAGS=` would set `CFLAGS` to null, meaning the compilation steps wouldn't make the `-Wall` checks.

Figure 24: A Simple `makefile`

```
CFLAGS=-Wall
CC=gcc
OFILES=random.o policy.o hardware.o elevator.o input.o

elevator: ${OFILES}
	${CC} -o elevator ${OFILES}

random.o: random.c random.h
	${CC} -c ${CFLAGS} random.c
input.o: input.c input.h random.h
	${CC} -c ${CFLAGS} input.c
hardware.o: hardware.c hardware.h
	${CC} -c ${CFLAGS} hardware.c
policy.o: policy.c policy.h hardware.h
	${CC} -c ${CFLAGS} policy.c
elevator.o: elevator.c elevator.h input.h policy.h
	${CC} -c ${CFLAGS} elevator.c

cleanall: clean
	rm -f elevator
clean:
	rm -f ${OFILES}
```

`makefile` for the elevator simulation. It can be simplified using the *implicit rule* for compiling `.o` files from `.c` files.

Each subsequent pair of lines describes one step in the build process, and has the form:

```
target: prerequisites
	recipe
```

The first line starts in column 1; the second starts with a tab. The target is the output of the step; the prerequisites are the inputs. Abstractly, such an entry describes when an output needs to be brought up to date and how to do so. When you ask `make` to build a specific target, it proceeds as follows:

1. If the target doesn't exist but there is no rule for regenerating it, print

an error message and abort `make`. If the target exists and is more recent than all its prerequisites, nothing need be done (so go on to the next target). Otherwise the target needs to be rebuilt (so continue to step 2).

2. For each prerequisite that is the output of some other step, recursively apply the process starting at Step 1.
3. When all prerequisite steps finish successfully, run the recipe. A recipe is usually a single shell command, but can be a sequence of such commands.
4. If the recipe fails, print an error message and abort `make`.[52]
5. If the recipe succeeds, return to whatever step caused this one to run.

Thus if you change `policy.c`, only the steps for `policy.o` and `elevator` need to be run. If you also change `policy.h`, then the `elevator.o` step also needs to run.

Step 3 only requires that all the prerequisites finished successfully; it doesn't actually require that they produce the expected "files." This is why the `cleanall` step in Figure 24 proceeds even though the `clean` step doesn't produce any file named `clean`.

Figure 25 shows the build graph for the system in Table 5 and illustrates what happens if `input.h` changes. Boxes show what files change (`input.h`), what steps need to be rerun (two `gcc` compilations and one `gcc` link), and what files are regenerated (`input.o`, `elevator.o`, and `elevator`). Unboxed files don't change and unboxed steps don't get rerun.

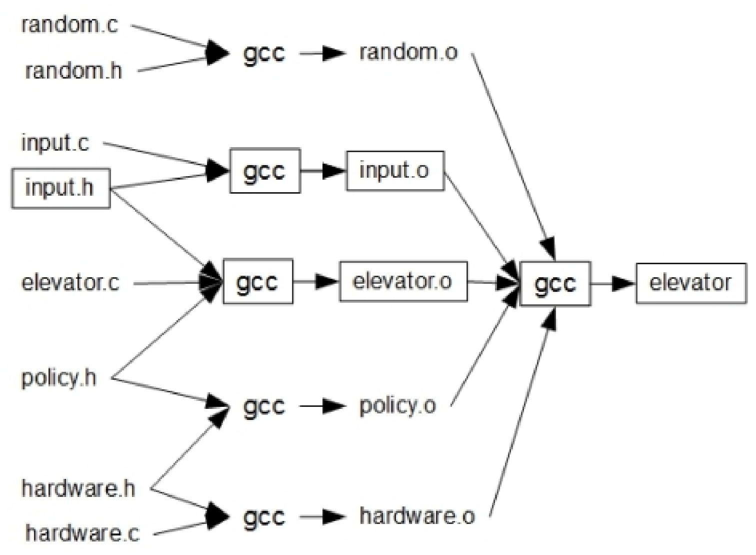
You run make with the command

```
make [ -f makeFileName ] [ flags ] [ outputFileName ]
```

Table 6 summarizes the flags you can pass to `make`; they are described more fully elsewhere in this section. The "output file name" is also called the *target* of the build process; you are "taking aim" at generating that particular file, and tracing a path through prerequisite graph to get to it. If you omit the `-f` *makeFileName* it looks for one called `makefile` in the current directory. If you omit the *outputFileName* it builds the output of the first step in the make file. `make` figures out an order in which to run the steps so that every generated file is produced before it gets used. If several steps (such as the `.c` file compilations) can be run in parallel, it is free to pick any order it chooses (but in fact in the current version of `make` it uses the order in which you listed the steps in the `makefile`).

[52]With the `-k` command line flag, instead just skip all the steps that depend on this one and go on to any other pending targets.

Figure 25: Processing when `input.h` Changes



Graph of dependencies for the elevator program of Table 5. When `input.h` changes, `input.o`, `elevator.o`, and `elevator` must be rebuilt.

14.3 Intermediate `make`

There are a few things to note about the example of Figure 24 on page 107.

First, the commands are quite repetitive. Every time you generate a `.o` file from a `.c` file, you run a command that differs only in which source file it names. It turns out that all the commands mentioning `${CFLAGS}` can be omitted. `make` has a collection of *implicit rules* that cover such common patterns. Writing implicit rules is beyond the scope of these notes; the Free Software Foundation lists the standard ones.[53]

Second, the step labeled `clean` has no dependencies and doesn't actually generate a file called `clean`; it just removes all the generated files (the executable file and all the object files). This kind of step is commonly used instead of writing a tiny `bash` script that performs a standard task. A `clean` step in particular is useful for removing object files to save space once you've built the executable program (which, in many cases, you would have moved somewhere

[53]http://www.gnu.org/software/make/manual/html_node/Catalogue-of-Rules.html

Table 6: Flags for `make`

Flag	**Page**	**Meaning**
`-B`	110	Rebuild all targets
`-f` *filename*		Use a specific makefile.
`-k`	110	Keep going after a step fails.
`-n`		Show what recipes would be executed, without actually running them.
var=*value*	111	set a variable, overriding the definition (if any) in the makefile.

from which other people can find and execute it).

You can use a step like `clean` as part of forcing a full rebuild of a system. You can use the `touch` command on a small number of source files to force a recompilation of everything that depends on them, but for a full recompilation a step like `clean` is better. Even better is to use the `-B` command line flag, which for each step assumes rebuilding is always necessary; it pretends all source files have changed.

If a step in the build process fails (that is, returns a nonzero exit status), `make` aborts.[54] Putting a hyphen before a command suppresses this behaviour.

Finally, modules like `input.c` and `elevator.c` probably `#include` library declaration files like `<stdio.h>`, the standard I/O package, but the dependency lines didn't list them. Conventionally, files that change extremely rarely, like system library files, don't get mentioned. This means that if such a library *does* change, `make` won't notice. In this case you must manually delete the object files (via `make clean`) and then rerun `make` to produce the executable.

Writing out all the dependencies can be tedious, especially if you need to discover and record *indirect* dependencies, such as when one header file includes another. Many modern compilers will generate dependency information for inclusion in a `makefile`; the `gcc` compiler does so with the `-M` switch.

14.4 Fundamentals of Build Management†

This section describes some of the fundamental concepts underlying build management systems, and outlines how an ideal one would operate. I know of no such ideal system.

[54] If you supply the `-k` switch on the command line, it keeps going but doesn't try to run any steps that depend on the output of the failed step.

A build process corresponds to a directed graph; the graph has to be acyclic for the build process to terminate. Directed acyclic graphs define *partial orders*: some files depend on ("are related to") other files, but there is not necessarily a relationship between every pair of files. What a build system does is perform a *topological sort* of the partial order, resulting in one of many possible *total orders* of the files: a linear sequence of nodes where each node comes after all its prerequisites. Directed graphs and orders are discussed in courses involving *graph theory*, such as CISC 203: Discrete Mathematics for Computing Science.[55]

Technically, a *source* is any object that cannot be recreated exactly via an automated process. It is conventional to refer to program files like `.c` files as "source files," and indeed most of them fit the definition – they are written by human beings instead of by programs. However, there are *program generators* that create other programs. A primary example is a parser generator like `bison`, which takes a grammar description as its source and outputs C or C++ (code and header files). Figure 26 shows what a `makefile` looks like with program generators. `grammar.y` is a source file written in the input language for `bison`. `grammar.tab.c` and `grammar.tab.h` are the definition and declaration files for the output of `bison`; they are generated files, not source files. The steps for generating object files rely on a default rule.

An ideal build process, given the same source, produces identical generated files. "Source" is not equivalent to the conventional meaning of "source file," however. The definition covers more kinds of files than just program code: Data from an experiment and input files for testing purposes both count as source. The `gcc` compiler itself might change (although, fortunately, this happens quite rarely). Also, in the example of Figure 24 on page 107, the compilation and linker steps would produce different output if the `CC` and `CFLAGS` variables change – or if the recipes change in any other way. The makefile itself need not change: If, with the makefile of Figure 24 on page 107, you override a variable on the command like, such as

```
    make CFLAGS="-mcpu=something" elevator
```

to change the type of machine for which `gcc` should generate code, all the compilation steps need to be rerun. Thus these variables and recipes are source objects; unfortunately `make` and many other build systems don't know this.

Typical build process managers like `make` have a simple notion of "change" to a file: a step is rerun if any of its inputs has a modification date later than

[55]http://www.cs.queensu.ca/students/undergraduate/courses/desc/CISC-203.html

Figure 26: `makefile` with Program Generators

```
BISONOUTPUT=grammar.tab.c grammar.tab.h
GENERATED=*.o $BISONOUTPUT
OBJECT=main.o parser.o util.o

# relies on standard .o.c default rule
main.o: main.c parser.h util.h
util.o: util.h
parser.o: parser.c parser.h $BISONOUTPUT

$BISONOUTPUT: grammar.y
        bison grammar.y

cleanall: clean
        rm -f $GENERATED
clean:
        rm -f *~
```

Use `man bison` for a little more on the parser generator. The key idea is that `parser.tab.c` and its header file are *generated* files and not *source* files. Note especially the use of variables to abstract the outputs of `bison`

at least one of its outputs. Unfortunately modification date is an imperfect reflection of change. If you add a comment to a source program, there is no need to regenerate the object file; it will be identical to the previous one. This is especially problematic when the output of a compilation step might be used in a cascade of other compilation steps, as happens with Java `.class` files; adding a comment to a widely-used `.java` file in principle forces redundant recompilation of many other files. An ideal process would notice when a newly-generated file is the same as the previous version, and avoid rerunning later steps that depend on it.

There are a few programs that take a file as input and overwrite the same file as output. The LaTeX variant of the TeX document processor is like this; it produces auxiliary files with information about things like forward references to sections of the document. If you recorded such a file as a dependency, the build process would never stabilize. This is another case where it would help to be able to verify that the new version of an output was the same as (or semantically equivalent to) the old version.

Sometimes you want to regenerate an old version of a system. For example, you might have released version 1, and have gone on to work on version 2. If a customer reports a bug in version 1, you would need to rebuild it exactly as it was at the time of release in order to track down the error, which means starting from exactly the same source. There are *version management systems* such as RCS,[56] CVS,[57] and Apache Subversion®[58] that can keep track of multiple versions and retrieve old ones. When a version management system restores an old version of a source file, it typically sets its modification date to that of the stored version. This interacts poorly with the use of modification dates to indicate change – another case where a `clean` step might be needed.

14.5 Other Build Systems

When there are many steps that can proceed in parallel (such as the `gcc` compilations of Figures 24 on page 107 and 25 on page 109), some versions of `make` can use multiple processors to perform several steps simultaneously.

Some languages such as Java have their own partial implementation of dependency analysis; if you compile a Java language file, it will recompile any other `.java` files it uses whose `.class` files are missing or out of date. Such specialized implementations tend to have two sorts of limitations. The first and most obvious is that they don't help with systems with source files written in multiple languages. For example, a system with `bison` (Section 14.4 on page 111) technically has two languages: C and the `bison` grammar description language.

The second limitation is that, without the full dependency information you're expected to record in a `makefile`, they sometimes recompile too much or too little.F as of 2010, `javac` sometimes failed to recompile some indirect dependencies.[59] Suppose `A.java` depends on `B.java`, which depends on `C.java`, and you change `C.java`. If you compile `A.java`, but `B.class` is more recent than `B.java`, the compiler might not notice that `C.java` and then `B.java` need to be recompiled.

[56]http://www.gnu.org/software/rcs/

[57]http://sourceforge.net/apps/trac/sourceforge/wiki/CVS

[58]http://subversion.apache.org/

[59]By the time you read this more recent versions of the Java compiler might have addressed these problems.

`make` itself has several problems and limitations that begin to crop up as your build processes become larger and more complex, particularly when you are producing several variants of a system (such as debug versus production versions, or versions for different operating systems). There have been several variants of `make` that address some of its limitations, such as imake[60] and CMake.[61] Apache Ant$^{\mathrm{TM}}$[62] is a very different portable build tool (written in Java) that solves many of these problems.

[60]http://en.wikipedia.org/wiki/Imake
[61]http://www.cmake.org/
[62]http://ant.apache.org/