Unix for beginners
Version 0.11, DRAFT

D. Vermeir
Dept. of Computer Science
Free University of Brussels, VUB
dvermeir@vub.ac.be

28 January, 2002

Abstract

This document is a short tutorial on using the unix/linux system. It is mainly concerned with using the system for software development, hence e.g. the chapter on "make". I try to explain features, based on a, slightly simplified, view on the underlying operating system. This makes the text more suitable for someone who wants to have an idea on what is going on "behind the scenes", e.g. beginning (computer) science students. Although the text was written with solaris in mind, most of it should be useful for other variations such as linux. A postscript version of this text is available in the file **uintro.ps**.

Contents

1 Introduction **4**

2 Files **5**

2.1 Introduction 5
2.2 Current directory, relative pathnames 5
2.3 Inodes, contents of directories, hard links. 6
2.4 Users, file ownership and permissions 8
2.4.1 Users 8
2.4.2 Ownership and permissions 8
2.5 File systems 9
2.5.1 Sharing directories between machines 10
2.5.2 Implementation of disk-based file systems 10
2.5.3 Other types of file systems 11
2.6 File types, symbolic links 12

3 Processes **14**

3.1 Creating processes: fork() and exec() 16
3.2 Exec and interpreters 17
3.3 Accessing files from programs, stdin, stdout, stderr 18
3.4 Process return value 19

4 A shell **20**

4.1 Logging in, choosing a shell 20
4.2 Bash basics 20
4.2.1 PATH 21
4.2.2 Completion 22
4.2.3 Background processes 23
4.2.4 History 23
4.2.5 I/O redirection 24
4.2.6 Pipes and filters 24
4.2.7 Shell variables 25
4.2.8 Filename substitution 27

4.2.9 Command substitution . . . 27
4.2.10 Quoting . . . 28
4.2.11 Shell command line processing revisited . . . 29
4.2.12 Bash startup files . . . 29
4.3 Shell scripts . . . 31
4.3.1 Control structures . . . 31
4.3.2 Here documents . . . 33

5 Text editors **35**

6 Make **36**
6.1 Make basics . . . 36
6.2 Pattern rules . . . 39
6.3 Automatic variables . . . 41
6.4 Other uses of make . . . 41
6.5 Built-in rules, automatically generating dependencies . . . 43
6.6 Make variables . . . 44
6.7 Built-in variables used by built-in rules . . . 45
6.8 Recursive make . . . 47
6.9 Beyond make . . . 47

Index **48**

1 Introduction

Unix is an "old" operating system, dating from the late 1970's. It is still thriving, especially on larger servers and, recently, also on smaller machines (**linux**[1]). This may be because it is arguably a rather reliable system where up-time is measured in months rather than days, as is the case for some other operating systems.

Another advantage of unix over other systems is that it is mostly "**open source**", meaning that anyone can inspect the source code to find or fix errors. Since its conception, unix has always been the favorite operating system of computer science researchers; hence most new developments (e.g. the internet, www, java etc.) have first been developed under unix.

While unix may be lacking in "idiot-proof" user interfaces (although that seems to be changing with the advent of desktop environments such as **CDE** and **KDE**[2]), it is very easy for a user to tailor his environment by adding scripts that automate any repetitive tasks.

Of course, this text scratches only the surface of what is available in unix. Typically, a unix system supports hundreds of commands (e.g. **wendy** has 1720 commands in the "standard" program directories). As a simple (although not perfect way) to find out whether there are any commands available that deal with a certain topic, you can use the **apropos** command. Once you find out the name of a relevant command, use **man** to see the actual "manpage[3]". Note that unix manual pages are divided into sections which have names that are numbers, possibly followed by a letter. If a command appears in several sections, the "`-s`" option of the **man** command should be used to specify the version of interest.

For solaris, manual pages and other documentation are also available on the web, e.g. at **wendy**[4].

[1] http://www.linux.org/
[2] http://www.kde.org/
[3] "Manpage" is unix jargon for manual.
[4] http://wendy.vub.ac.be:8888/

2 Files

2.1 Introduction

As with any operating system, unix stores persistent information in files. Files are organized in a tree-structure where nodes that have children are called "*directories*" (directories are also files, see section 2.3), as illustrated in figure 1.

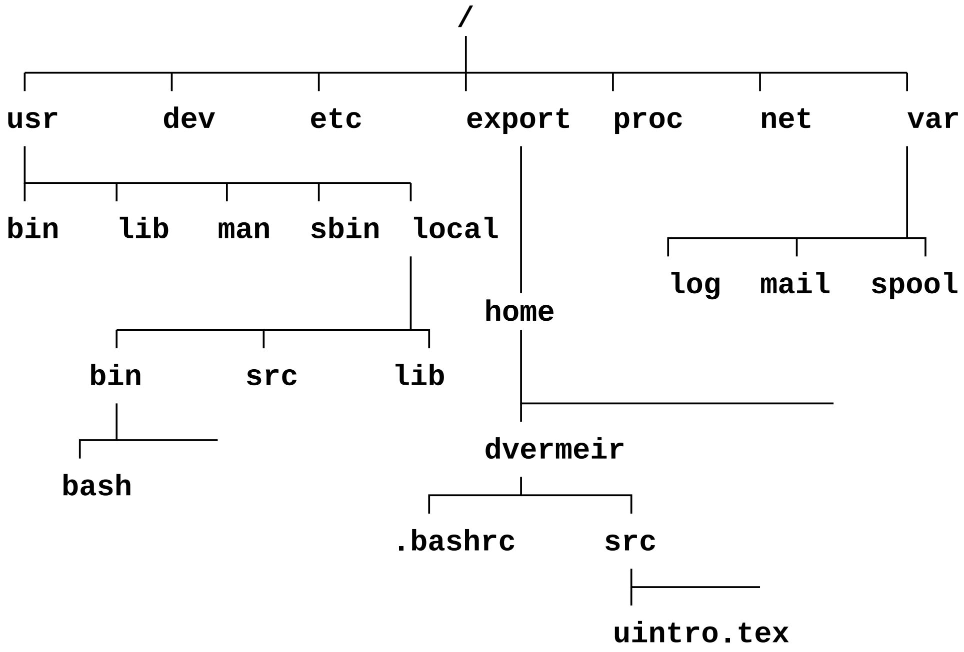

Figure 1: The unix directory structure

You can refer to a particular file by a string that contains the names of the directories on the path from the root (/) to the file, separated by slash ("`/`") characters. E.g. `/usr/local/bin/bash` refers to the file `bash` in the directory `/usr/local/bin` which itself is part of the directory `/usr/local` etc. Such a string is called an *absolute pathname*.

2.2 Current directory, relative pathnames

At all times, a user (or a process, see section 3) is located in a particular directory, called the *current working directory*. When a user logs in, his current directory is his *home directory*. While any directory can serve as a user's home directory, by convention[5], a home directory is a subdirectory of /export/home with the same name as the user's login name, e.g. the user dvermeir's home directory is /export/home/dvermeir.

You can use the **cd** ("change directory") command to change working directories. You can use the **pwd** ("print working directory") command to show the current directory.

It is usually easier to refer to a file using a *relative pathname*, which describes the path to follow from the current working directory to the file at hand. E.g. if the current directory is /export/home/dvermeir, then the pathname src/uintro.tex refers to the file with absolute pathname
/export/home/dvermeir/src/uintro.tex. In order to allow relative pathnames to refer to files not in the subtree of the current directory, ".." is used to refer to the unique parent directory of a directory.
E.g., when in /export/home/dvermeir, you can refer to /usr/local/bin/bash using ../../../usr/local/bin/bash. Another abbreviation is "." for the current directory.

2.3 Inodes, contents of directories, hard links.

While you can refer to files using symbolic names, the system uses *inode number*s rather than names to refer to a file. An inode number is a number that uniquely identifies a file in a file system. The mapping between inode numbers and symbolic names is done via *directories*. Intuitively, a directory is just a normal file containing a 2-column table where the first column in a row contains an inode number and the second column contains a symbolic name (see figure 4). It is easy to see how the system can use the contents of directory files to determine the inode number of a file refered to by an absolute or relative path.

It is possible to have several names in the same of different directories referring to the same file/inode number. Such a file is said to have several (hard) *links* to it. You can create such an extra link to an existing file using the **ln** (**lin**k) command. The **rm** command **rem**oves a file. Actually, only the directory entry (often called "link[6]") is removed; the file itself remains until the last link referring to it disappears. Files can be "moved" (or renamed) using the **mv** command. This command only manipulates directory entries, unless the new path refers to another file system, in which case the actual data must be moved, see section 2.5.2. To copy the actual data of a file onto another one, use the **cp** command. From the above, it follows that such a copy will also affect any hard links to the target file.

[5] On linux, home directories can be found in /home.
[6] This is why the system function to remove files is called **unlink**

```
tinf2% ln /tmp/empty /tmp/newlink
tinf2% cp /etc/passwd /tmp/empty
```

`/tmp/empty` and `/tmp/newlink` still point to the same inode and thus their contents will be the same.

Use the **ls** ("**lis**t") command to show the contents of a directory. **Ls** has (too) many options, but the following are often useful: `-l` shows details like ownership, permissions, size and date of last modification, as shown in figure 2, while `-t` sorts the output according to the most recent modification time.

```
51 tinf2:~/software/packages/uintro/doc$ ls -lt
total 356
-rw-r--r--   1 dvermeir staff      11537 Jul 26 13:10 uintro.tex
-rw-r--r--   1 dvermeir staff      12435 Jul 26 12:57 uintro.html
-rw-r--r--   1 dvermeir staff      54708 Jul 26 12:57 uintro.ps
-rw-r--r--   1 dvermeir staff       1278 Jul 26 12:57 uintro.aux
-rw-r--r--   1 dvermeir staff      15508 Jul 26 12:57 uintro.dvi
-rw-r--r--   1 dvermeir staff        269 Jul 26 12:57 uintro.lof
-rw-r--r--   1 dvermeir staff       6419 Jul 26 12:57 uintro.log
-rw-r--r--   1 dvermeir staff        611 Jul 26 12:57 uintro.toc
drwxr-xr-x   2 dvermeir staff        512 Jul 25 14:10 CVS/
-rw-r--r--   1 dvermeir staff         94 Jul 22 19:53 config.tex
-rw-r--r--   1 dvermeir staff       7198 Jul 22 19:53 Makefile
-rw-r--r--   1 dvermeir staff       7130 Jul 22 19:53 Makefile.in
-rw-r--r--   1 dvermeir staff       1270 Jul 22 19:53 Makefile.am
-rw-r--r--   1 dvermeir staff         96 Jul 22 19:53 config.tex.in
-rw-r--r--   1 dvermeir staff       2411 Jul 22 19:33 mount.gif
-rw-r--r--   1 dvermeir staff       6164 Jul 22 19:33 disk.gif
-rw-r--r--   1 dvermeir staff       4600 Jul 22 19:33 mount.eps
-rw-r--r--   1 dvermeir staff      13027 Jul 22 19:33 disk.eps
-rw-r--r--   1 dvermeir staff       2423 Jul 22 19:33 filesys.gif
-rw-r--r--   1 dvermeir staff       6315 Jul 22 19:33 filesys.eps
-rw-r--r--   1 dvermeir staff       6249 Jul 22 19:32 disk.fig
-rw-r--r--   1 dvermeir staff       3439 Jul 22 19:32 filesys.fig
-rw-r--r--   1 dvermeir staff       1390 Jul 22 19:32 mount.fig
-rw-r--r--   1 dvermeir staff        194 Jul 22 19:32 uintro.dict
52 tinf2:~/software/packages/uintro/doc$
```

Another useful **ls** option is "`-a`" which also shows filenames that start with "`.`" (dot) which are normally not visible. Such files (and directories) are often used to store configuration information for various packages. Chances are that you will find e.g. a `.dt` (desktop configuration for **CDE**) and a `.netscape` subdirectory in your home directory.

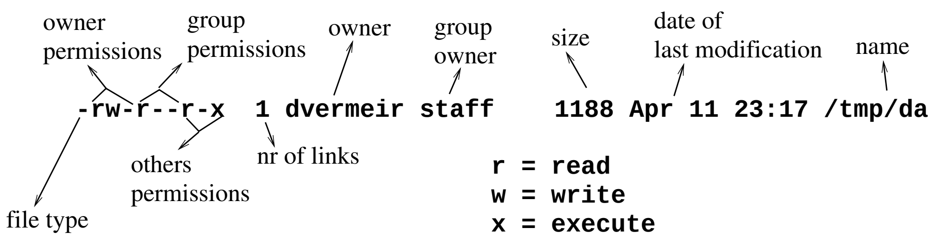

Figure 2: File access permissions

2.4 Users, file ownership and permissions

2.4.1 Users

Users have a *user name* and a *password*. In addition, a user also has a *home directory*, (see section 2.2) and a *shell program*. Upon login, the user supplies his name and password, after which the shell program is executed in the users' home directory, see section 4.1.

Internally, the system uses so-called *UID* numbers to identify users.

All this information is stored in the file **/etc/passwd**. This file also contains a "primary" *group id* (*GID*) identifying a *group* to which the user belongs. A *group* is an arbitrary set of users (you can find the defined groups on your system in the `/etc/group` file). Note that a user may belong to several groups.

There is one special user with UID 0, called *root*. This user is often called the "*super user*" because he can access all resources on the system, independently of any specific permissions. Therefore, root's password is usually a closely guarded secret.

2.4.2 Ownership and permissions

Each file has a user as *owner* and a group as *group owner*. The owner can use the **chmod** command to set permissions that determine the type of access (*read*, *write* or *execute*) allowed to three categories of users: the *owner* herself, the users belonging to the *group owner* group, and all *other users*.

Note that "execute" permission on a directory is interpreted as "permission to traverse". E.g. if someone has "execute", but not "read", permission on `/tmp/dir/`, she can access `/tmp/dir/txt` (provided she has the appropriate permissions on this file) but she cannot use **ls** to see all files in `/tmp/dir`.

If the "three categories of user" approach does not fit your needs, you can use

access control lists to give selected permissions to arbitrary (groups of) users.

2.5 File systems

While the system always presents a view of all the files in a single tree structure as in figure 1, this does not correspond with the need to access files on several (possibly removable) disk devices. Therefore the system groups subtrees into *file systems*[7].

A file system can most easily be thought of as a subtree of the *root* `/` directory. The file system containing the root (`/`) is called the *root file system*. The **mount** command can be used to *mount* a file system on some directory in the root's hierarchy, as shown in figure 3.

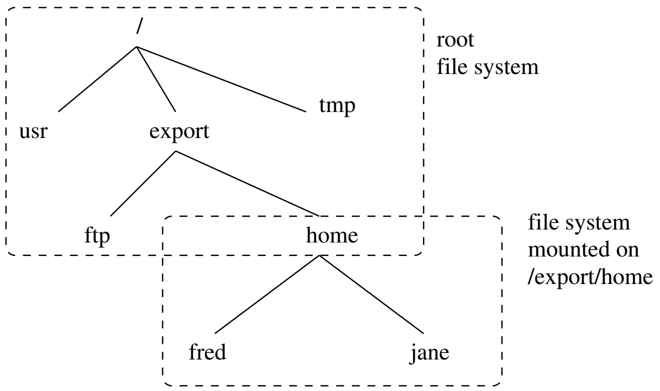

Figure 3: Mounting file systems

In the figure, the directory `/export/home` in the root file system is used as a *mount point* for a file system containing users' home directories.

You can use **mount** or **df** (**d**isplay **f**ile systems) to see which file systems are currently mounted on which mount points.

```
30 tinf2:/usr/local$ df -k
Filesystem            kbytes    used   avail capacity  Mounted on
/proc                      0       0       0     0%    /proc
/dev/dsk/c0t0d0s0    1952573 1413891  343425    81%    /
fd                         0       0       0     0%    /dev/fd
swap                  430784   23568  407216     6%    /tmp
```

[7] Originally, file systems were limited to a single disk, but recent versions support file systems spanning several disk volumes.

```
/dev/dsk/c0t5d0s0    8551141 4101355 4364275    49%    /export/home
/dev/dsk/c0t5d0s3    8551141 7056857 1408773    84%    /usr/local
tinf1:/var/mail       963662  801576  161123    84%    /var/mail
31 tinf2:/usr/local$
```

2.5.1 Sharing directories between machines

Note the last line in the output of **df**: it is also possible (using the same **mount** command) to mount "remote" file systems (in the example: the directory `/var/mail` is mounted from machine `tinf1` onto `tinf2`). This happens using the **nfs** (**n**etwork **f**ile **s**ystem) services.

A machine may make a directory available for such sharing over the net using the **share** command. Actually, any remote directory a machine has been allowed access to can be referenced via the "`net/machine-name`" directory.

```
44 tinf2:~$ cd /net/tinf1
45 tinf2:/net/tinf1$ ls
cdrom/  var/
46 tinf2:/net/tinf1$ cd var
47 tinf2:/net/tinf1/var$ ls
mail/
48 tinf2:/net/tinf1/var$
```

2.5.2 Implementation of disk-based file systems

The traditional implementation of a file system is shown in figure 4 which shows the layout of (part of) a disk containing a unix file system.

Hence an inode number is simply an index in the array of *inodes* occupying the first part of the disk. The second part of the disk consists of equal-size data blocks. The inode information includes ownership, permissions and a number of pointers to data blocks. The first few pointers point directly to blocks containing data associated with the file while the last pointers point to so-called *indirect*, *double-indirect* and *triple-indirect blocks*. An indirect block does not contain data but just pointers to (direct) data blocks. Similarly, a double-indirect block consists of pointers to indirect blocks.

Assuming that the size of a block is 8K bytes, that an inode contains 10 (direct) data block pointers, that a pointer to a block needs 8 bytes and that the size of a file is 10MB, this implies that, to access byte number 8072001 of a file, the system would need to access the first indirect block and then take the 1000th address of

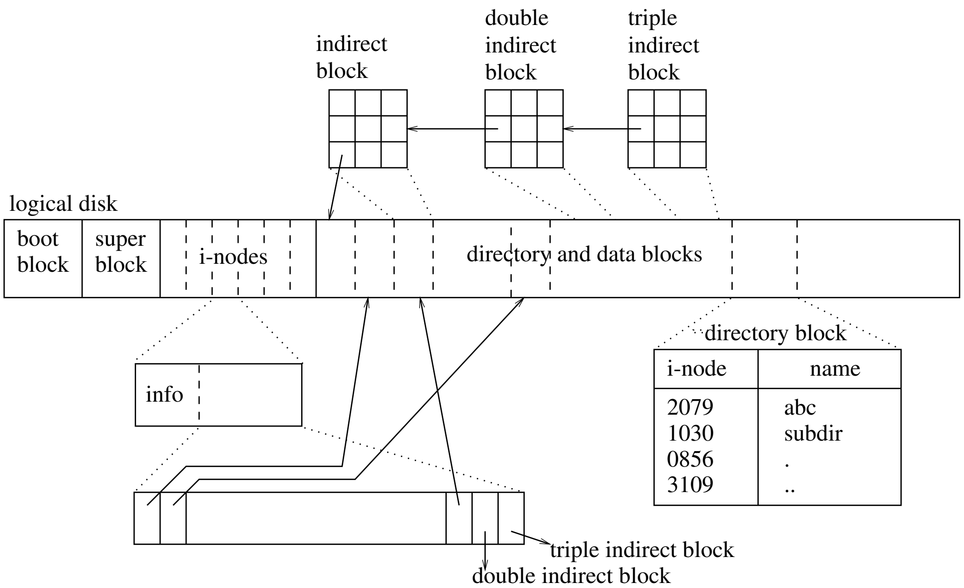

Figure 4: Traditional layout of a file system on disk

a data block in this indirect block. Byte number 8072001 is then the first byte in this data block.

From this layout, it is clear that access to small files (under 80KB in our example) is very efficient (at most one disk access). Random access to larger files is reasonably fast too, also because unix keeps often-accessed (index) blocks in main memory.

Finally, note that, since inode numbers are simply indexes into an array on disk, it follows that "*hard links*" (see section 2.3) are limited to files that belong to the same file system.

2.5.3 Other types of file systems

In the previous sections, we have already encountered two kinds of file systems: disk-based, called *ufs*, file systems and network-based, called *nfs*, file systems. In solaris, the concept of file system has been generalized to that of *virtual file system* which can be regarded as an abstract data type that specifies the interface that must be implemented by any kind of file system.

Thus several more kinds of file systems are available in solaris:

- A *tmpfs* file system supports simulating a file system in main memory, possibly backed up by swap storage. This is ideal for temporary files for which fast access is important.

- A *swap* file system is used to provide backup storage for processes that must temporarily be “swapped out”.
- The *proc* file system (see the `/proc` directory) provides a “file view” on the attributes of processes. E.g. the “file” `/proc/222/as` “contains” the address space of the process with id 222 (see section 3).
- The *fd* file system provides a simplified view on a process's *file descriptors* (see section 3.3); e.g. `/dev/fd/1` refers to its *standard output*, see section 3.3.

2.6 File types, symbolic links

A nice feature of the unix architecture is that it tries to package all sorts of things as “files”, thus providing a uniform interface for programs accessing such resources, see section 3.3.

Typically, the following file types are supported:

- *regular files* are simply arrays of bytes.
- *directory* files map names to inode numbers (see section 2.3).
- *character-special files* represent devices that are accessed as “streams” of characters. Examples include network interface devices (e.g. `/dev/hme`), serial ports (e.g. `/dev/ttya`), tape drives (e.g. `/dev/rmt/0`) etc. A neat file in this category is the “pseudo device” `/dev/null` which acts as a black hole: it happily absorbs all data written to it while reading immediately returns “end of file” (`<eof>`). */dev/null* is useful in combination with I/O redirection (see section 4.2.5), e.g. to get rid of certain output streams.
- *block-special files* represent “random access” devices such as disks (e.g. /dev/dsk/c0t5d0s3 is the disk on **c**ontroller 0, SCSI **t**arget 5, **d**rive 0, **s**egment 3) but also the “kernel” address space `/dev/kmem`.
- *named pipe* files represent FIFO[8] streams that can be used for inter-process communication: a process can write data to such a file while another (unrelated) process can read those data using the same file.
  E.g. print commands can write to `/var/spool/lp/fifos/FIFO` which is read from by the print spooler.

[8]FIFO stands for “first-in, first-out”.

- *socket* files represent so-called *unix sockets*, a "local" network connection. You can treat such a file as a (local) network address and access it as you would a normal "socket". E.g. */usr/local/Hughes/msql2.sock* is a socket file that interfaces to the local **msql** database server.

- *symbolic link* files are simply files that just contain the pathname of another file. An access on a symbolic link file is delegated to the file the symbolic links refers to[9]. Note that, if you remove a symbolic link file, only the link is removed, not the file that refers to it. Similarly, removing a file that is pointed to by a symbolic link leaves the symbolic link file "dangling".

  ```
  tinf2% ln -s /tmp/emptyfile /tmp/symlink
  tinf2% rm /tmp/emptyfile # now /tmp/symlink is ``dangling''
  tinf2% cp /tmp/symlink /tmp/copy
  tinf2% cp: cannot access /tmp/symlink
  ```

  Symbolic links can be created using the **-s** option of the **ln** command.

The following shows the output of `ls -l` on some "special" files. Note that the file type can be deduced from the first letter of the output.

```
drwxrwxrwt 23 sys      sys      3515 Apr 12 12:48 /tmp/
-rw-r--r--  1 dvermeir staff    1188 Apr 11 23:17 /tmp/dates
srwxrwxrwx  1 dvermeir staff       0 Apr  5 19:58 /usr/local/Hughes/msql2.sock
crw-------  1 root     sys    11, 40 Dec  7 17:24 /devices/pseudo/clone@0:le
lrwxrwxrwx  1 root     root       10 Dec  7 17:16 /usr/tmp -> ../var/tmp/
prw-rw-rw-  1 lp       lp          0 Apr 10  1996 /var/spool/lp/fifos/FIFO
brw-r-----  1 root     sys    32,  0 Apr 10  1996 /devices/sbus@1f,0/SUNW,fas@e,8800000/sd@0,0:a
```

[9]Note that removing a symbolic link does **not** remove the underlying file.

3 Processes

All activities that take place in the system are carried out by *processes*.

Intuitively, a process is the execution of a *program* by the system on behalf of a user, where a program is a file containing instructions that can be interpreted by the CPU.

Unix has always been a multiprocessing system, which means that many processes may be active at the same time. Of course, this concurrency is only simulated on single processor systems, where time on the CPU is divided between all processes, using small (milliseconds) chunks, creating the illusion of real concurrency.

The following illustrates a snapshot, which was taken using the “top” command, of activities in a simple workstation. If the “top” command is not available on your system, you can try the **ps** (**p**rocess **s**tatus) command.

There are only a few programs that directly deal with processes. However, most command line interpreters that are used under unix have an extensive set of primitives to manage processes (see section 4).

```
last pid: 27696;  load averages:  0.01,  0.10,  0.13
88 processes:  87 sleeping, 1 on cpu
CPU states: 99.2% idle,  0.0% user,  0.6% kernel,  0.2% iowait,  0.0% swap
Memory: 256M real, 4472K free, 228M swap in use, 262M swap free

  PID USERNAME THR PRI NICE  SIZE   RES STATE   TIME    CPU COMMAND
27696 dvermeir   1  59    0 2176K 1704K cpu     0:00  0.54% top
  340 root      11  58    0   81M 5120K sleep   5:06  0.06% mibiisa
  176 root      13  56    0 3848K 1544K sleep   0:18  0.02% syslogd
28239 root       1  59    0  128M   27M sleep  56:52  0.02% Xsun
  196 root      12  53    0 2832K 1912K sleep  32:11  0.01% nscd
    1 root       1  58    0  736K  152K sleep   8:00  0.01% init
27688 dvermeir   1  18   10 6048K 3064K sleep   0:00  0.01% dtscreen
 7603 dvermeir   1  58    0  611M   21M sleep 581:09  0.00% msql2d
22587 dvermeir   1  48    0   26M 3448K sleep   2:08  0.00% .netscape.bin
  232 root       1  58    0  992K  520K sleep   1:31  0.00% utmpd
  313 root       4  58    0 2056K 1008K sleep   1:10  0.00% in.rarpd
  216 root       1  58    0 1808K 1200K sleep   1:03  0.00% lp
  290 root       1  58    0 6552K 1640K sleep   0:25  0.00% dtlogin
28328 dvermeir   8  59    0 8536K 4976K sleep   0:19  0.00% dtwm
  237 root       1  59  -12 2040K  864K sleep   0:18  0.00% xntpd
```

One may wonder how processes come into existence. The answer is that processes can only be created by other processes, using the **fork** system call. This results in a tree-structure of active processes where a process is the *parent process* of all processes that it created. The root of the tree is the first process that was magically created when the system started. It is called *init* and is considered to be its own parent.

Associated with a process is an *address space* in (virtual) memory containing the instructions and data on which the process is operating. Unix keeps quite a bit of further information on each process, of which we mention only a selection:

- The *process ID* (usually abbreviated as *PID*) is simply an index in the system's array of process data structures (the so-called *process table*). Several commands, such as the **kill** command expect a PID to identify the process(es) to operate upon.
- The PID of the unique parent process that created this process.
- The real *user id* (*UID*) and *group id* (*GID*) that "own" the process. There is also an *effective* UID and GID that determine the process's permissions to access files or other resources. Usually, the effective UID and the real UID are the same. However, if the file corresponding to the program has the *set-uid* bit set[10], then the process will change its effective UID to the UID of the owner of that file (see section 2.4.2). This is useful, e.g. when a process needs to access information that is normally not accessible to its real user. E.g. the **ps** program needs access to the kernel data structures. Therefore, the corresponding file is owned by the super user (who has ID 0) and has the set-uid bit set.
- A table of open *file descriptors* (see section 3.3) connecting it with a number of open files and/or devices.
- The current working directory (see section 2.2), and root (see section 2.1) directory[11].
- The *file creation mask* which determines the default permissions for any new files that are created by the process. This mask can be manipulated using the **umask** command.
- A number indicating the priority the process gets from the scheduler (higher numbers mean lower priority). The priority of a process can be manipulated by the **nice** or **renice** command.

[10]This can be done using the **chmod** command.

[11]The **chroot** command allows a process to run with a different idea of where the root directory is; this is useful, e.g. to construct a "sandbox" for untrusted processes (e.g. an *ftp* server).

- A number of string-valued *arguments*, and a pointer to an *environment* which itself is a list of ⟨name,value⟩ pairs. These correspond to the standard arguments of the C (or C++) function

```
int main(int argc,char*argv[],char* envp)
```

3.1 Creating processes: fork() and exec()

The following C++ code illustrates how just two[12] system calls implement a flexible system to create new processes.

- The **fork** system call clones the calling (parent) process: after a call to **fork**, a new process has been created that inherits most attributes from its parent (of course, its PID and parent PID will be different). In the new child process, the call to **fork**, will return 0 while in the parent process, the return value will be the PID of the child. Note that, by itself, **fork** is already useful, e.g. to implement internet servers that create a new child process to handle each new connection while the parent process continues to wait for new connections[13].
- While **fork** supports the creation of new processes, **exec** lets a process change the program it is executing. Essentially, **exec** takes the pathname of an executable file as its first argument and modifies the current process to start executing this file. This implies that a call to **exec** never returns.

There are a few other system functions that deal with processes. E.g. **waitpid** allows a parent process to check the status of a child etc.

The program below illustrates the use of **fork** and **exec** in a simple implementation of a trivial *command line interpreter* (also called *shell* in the unix jargon).

```
// $Id: shell.C,v 1.3 1999/08/02 10:47:03 dvermeir Exp $

// This program implements a simple ``shell'': it waits for
// an input line of the form
//
//      program argument..
//
// then starts a child process and makes it execute ``program''
// with the given arguments. The shell waits for the child
// to finish, then prompts for another input line.
```

[12]Actually, there exist a number of variations on these function calls.

[13]Of course, in many cases, this might be done more efficiently using multithreading.

```
#include        <iostream>
#include        <unistd.h>
#include        <sys/types.h>
#include        <sys/wait.h>

int
main(int,char**) {
const int            MAXCMD = 1024; // max length of a command line
char                 cmd[MAXCMD]; // command line
pid_t                pid; // PID of child process
int                  status; // return status of child process

while (cout << "> ",cin.getline(cmd,MAXCMD)) {
        // we got a command line after the '>' prompt
        // next, we create a new process using fork()
        if ((pid=fork())<0) { // create new process (pid)
                cerr << "error: cannot create child process" << endl;
                // could not fork(), try again with another command
                continue;
                }
        if (pid==0) { // we are the child process
                // execute the command as typed
                execl(cmd,cmd,0); // this should never return
                // if it does, something went wrong
                cerr << "error: cannot run program \"" << cmd << "\"" << endl;
                }
        else
                { // we are the parent process
                // just wait for the first child to die
                // (it should be the one with PID pid)
                if (waitpid(pid,&status,0)<0)
                        cerr << "error: while waiting for " << pid << endl;
                }
        }
cout << endl;
return 0;
}
```

3.2 Exec and interpreters

The first argument to **exec** (see section 3.1) need not be a file containing machine instructions. It is possible to make the system call another program (typically an interpreter, e.g. `/usr/local/bin/perl`) that "executes" the orginal executable file (which, in case of a perl interpreter, would contain a perl script). This is done by letting the first "line" of the executable file start with `#!`, in which case the behavior of **exec** is as follows (note that `argv` and `argc` refer to the standard arguments of `main(int argc,char*argv[],char* envp)` in a C (or C++) program.

```
int
exec(const char* path, const char* arg0,...,const char* argn, char* /* NULL*/)
{
if (path is executable file)
  if (first line of path starts with '#! newpath optional-arg') {
    make current process execute main() from newpath with argv[0] = newpath,
    argv[1] = optional-arg, argv[2] = path, argv[3] = second arg in exec etc. }
  else {
    make current process execute function main() from path
    with argc, argv[] as specified by the parameters
    }
else
  error
}
```

3.3 Accessing files from programs, stdin, stdout, stderr

From a program, files can be opened[14] using the **open** system function.

```
int open(const char* pathname, int openflag,.../*
mode_t mode */);
```

The function returns an integer value, called a *file descriptor* which is an index in the process's *file descriptor table* (also called *fd table*), as illustrated in figure 5.

Note that, in figure 5 only the *vnode table* (a vnode is a solaris generalization of an inode), is global and shared by all processes. However if file descriptors are duplicated, either explicitly using the **dup** system function, or because of inheritance by a child process, they share the "current position" in the file[15].

The file descriptors 0, 1 and 2 have a conventional meaning:

- 0 is *standard input* (*stdin*), usually the user's keyboard,
- 1 is *standard output* (*stdout*), usually the user's window/terminal,
- 2 is *standard error* (*stderr*), usually the user's window/terminal. Error and status messages are often sent to `stderr` in order to avoid mixing them

[14]Note that this is the "low level" system interface, many languages and libraries provide more convenient methods.

[15]Of course, unix has facilities for locking (parts of) a file, e.g. using the **lockf** library function.

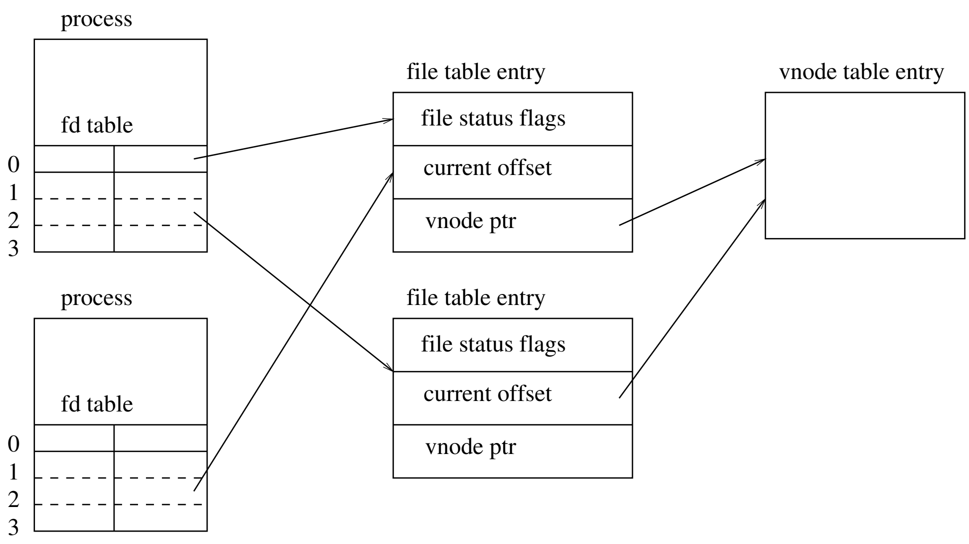

Figure 5: File I/O implementation

with "normal" output (on `stdout`). Also, `stderr` is usually not buffered, which guarantees that anything written to it also shows up, even if the process dies soon afterwards.

The main other I/O system functions are **close**, to close a file, **dup** to manipulate the file descriptor table, **read** and **write** to read, resp. write, data from/to a file (descriptor). Information about the status, ownership, locks etc. on a file (descriptor) can be manipulated using the **fcntl** system function. Device-specific operations may be requested using the catch-all **iocntl** function.

Finally, an interesting alternative to the **read** and **write** functions is available (for random-access device files) by "mapping" a file (descriptor) into virtual memory using the **mmap** function. After mapping, I/O can be performed by simply manipulating memory contents.

3.4 Process return value

When a process terminates, e.g. because the program's `main` function returns or because the **exit** function was called, it returns the integer value specified in the `return` or `exit` statement.

If the process was succesful, this value should be 0. All other return values indicate some kind of error, see section 4.3 where this is used to implement control structures in shell scripts.

4 A shell

4.1 Logging in, choosing a shell

When a user logs into the system, his identity is checked using a password. After that, a process is started that executes the executable file associated with the user. When this program finishes, the user is automatically logged out. The file executed upon login is determined by the system, based on the contents of **/etc/passwd**, see section 2.4.1.

Usually, this executable file is a so-called *shell* (see section 3.1) program that interactively reads and executes commands for the user. Note that this program is not part of the operating system; it is perfectly possible (and this is often done) to substitute another program in `/etc/passwd`. E.g. one could define a user "date" with an associated program `/usr/bin/date`. Each time "date" logs in, the program `/usr/bin/date` would run, i.e. display the date, and exit. More usefully, one can develop special purpose "*restricted shell*" programs that allow only certain predetermined operations to be executed by naive users, e.g. using menus.

In this chapter we will briefly introduce one of the more convenient shell programs, the so-called *bash* (**B**ourne **a**gain **sh**ell[16]).

After some experience, you may find that using a decent command-line interpreter such as `bash` results in a higher productivity than many so-called user-friendly graphical interfaces. This should not be surprising if one compares the efficiency, i.e. the amount of information transferred[17] to the system vs. the amount of physical effort and time needed to e.g. move the mouse, then click (or worse, type) vs. typing a few keystrokes[18].

4.2 Bash basics

The behavior of the shell can best be understood from the following pseudocode:

```
while (true) {
  bool background = false;
  show prompt on screen # if interactive
```

[16]Bourne is the developer of the old "Bourne shell" `/bin/sh` (which is still widely used for shell scripts, due to the fact that it is available on every unix flavor).

[17]Use e.g. the Shannon measure

[18]Of course, the need for thought is conveniently left out of the above definition of efficiency. Indeed, using a shell implies a bigger need for knowledge as the choice of what to do is much larger than with a restricted graphical or menu-based interface.

```
read a command line from input
if (the command line consists of <eof> only)
  exit
if (the command line ends with '&')
  background = true
perform substitutions and
  split command line into words # details later
find an executable file F that corresponds to
  the first word (taking $PATH into account)
create a new child process and let it
  exec(F, path-of-F, 2nd word, 3rd word, ..)
if (not background)
  wait for the child process to finish
}
```

Thus each command of the form

`command parameter`$_1$ `...parameter`$_n$

will eventually result in a new process that executes `command` using `parameter`$_1$ .. `parameter`$_n$ as parameters. Thus, if `command` is a compiled C program, its `main` function will be called using

```
main(n+1,argv,envp)
```

where

- `argv[0] = absolute path of command`, `argv[1] = parameter`$_1$,..., `argv[n] = parameter`$_n$
- `envp` is a pointer to the parent (shell) process's environment (see section 3).

Note that the absolute path of the executable file is passed as the first parameter in `argv[]`.

4.2.1 PATH

It would not be very convenient to each time completely specify the full path of the file we want to execute. E.g. typing

```
/usr/bin/ls /tmp
```

seems like a lot of work. Luckily, some versions of **exec** use the *PATH* environment variable to construct a full path name for `command`. The value of `PATH` is a list of directories, separated by colons (`:`). If `command` is not a (full or relative) path, **exec** will look for a file called `command` in each of the directories in `PATH`. The first executable file that is found will be executed (and its full path will be passed as argument 0). Thus, if the value of `PATH` is

```
.:/usr/local/bin:/usr/bin
```

then typing

```
ls /tmp
```

will have exactly the same effect as the previous example.

It is interesting to note that, if the current directory (".") is not in the `PATH` then typing

```
myprogram
```

will not work if `myprogram` appears only in the current directory. But, according to the rules mentioned above, typing

```
./myprogram
```

will work just fine.

4.2.2 Completion

Bash provides a further device to shorten typing: if, while entering a command line, you enter `<tab>`, then **bash** will attempt to complete the current word, if there is only one possibility. This works for commands, files and user names, and apparently is context-sensitive (i.e. for completing the first word in a command line, **bash** will only consider executable files).

If there are several possibilities, hitting `<tab>` twice will show them all, allowing you to continue typing until you have a unique prefix.

For example, typing `zm<tab>` at the beginning of a command line will result in `zmore` because there is only one program in the path that starts with "`zm`". Similarly, `ls m<tab>` will extend the command line to

```
ls my_subdirectory_with_a_long_name
```

if there is only one file in the current directory starting with “m”.

If there is another subdirectory called my_subdirectory_with_a_name,

```
ls m<tab>l<tab>
```

will do the job: after the first <tab>, **bash** will complete the “m” to “my_subdirectory_with_a_”. Typing “l<tab>” then allows **bash** to uniquely identify the full name.

4.2.3 Background processes

As explained in section 4.2, the default behavior of the shell is to wait for the command to finish (e.g. the shell process waits for its child that executes the command to die). As shown in the pseudocode in section 4.2, adding an ampersand (“&”) at the end of the command line alters this behavior. Thus typing

```
bigjob&
```

will immediately return while the bigjob process executes “in the *background*”.

A process that is not running in the background is running in the *foreground*. Such a process can be put in the background while running by typing ctrl-z, which suspends the foreground process. Then typing the built-in bg (**b**ack**g**round) command will put the foreground process in the background. Conversely, typing fg (**f**ore**g**round) will bring a background process to the foreground[19].

4.2.4 History

Bash keeps a list, called the **history**, of all previously entered command lines. This list can be shown, e.g. using the built-in “history” command. The up- and down arrow keys can be used to navigate the list. The current entry can then be edited and (re)executed by typing <return>. Alternatively, to execute the last command starting with a certain string, e.g. “pre”, it suffices to type !pre. For example, the author’s most popular command line is

```
!m
```

which typically invokes the **make** (see section 6) program.

[19]One imagines possible problems when several processes want to read input from the window/terminal at the same time. This aspect is not further discusssed in this text, the interested reader can consult e.g. the section on job control in the **bash** manual.

4.2.5 I/O redirection

Many programs write their output data to *stdout* and/or take their input from *stdin*, see section 3.3.

From the shell, it is possible (and easy to implement, given that child processes inherit open file descriptors) to *redirect* file descriptors to particular files.

As an example, consider the **cat** program which basically copies *stdin* to *stdout*. Then

```
cat <in >out
```

will copy the contents of file in to (and overwrite) the file out. An application of this is the following (macho) way to enter text into a file without using an editor: type

```
cat >outputfile
```

followed by whatever text you want, followed by <eof> (usually <ctrl-d>).

It is also possible to *append* the standard output to an existing file, as in[20]

```
cat message >>archive
```

Other file descriptors may be redirected using n> where n is the file descriptor. E.g.

```
latex uintro 2>/dev/null
```

will run **latex** and send *stderr* to */dev/null*.

4.2.6 Pipes and filters

A *pipe* consists of two file descriptors that are "internally conected"[21], i.e. what is written to one descriptor can be read from the other descriptor, see e.g. the **pipe** function call. The shell allows the use of anonymous pipes using the "|" symbol between commands. E.g.

```
who | wc -l
```

[20] << is used for *here documents*, see section 4.3.2.

[21] Under solaris, pipes are implemented using the powerful and general concept of **stream**.

will create two concurrent processes running **who** and **wc** where the standard output (file descriptor 1) of **who** is connected by a pipe to the standard input (file descriptor 0) of the **wc** command. The net output of the second process will thus be the number of users that are currently logged in.

Pipes fit well with the unix tradition of *filter* programs, i.e. programs that perform a useful transformation on data read from standard input, writing the result in a (human and) machine processable form on standard output. The latter implies that one should refrain from writing fancy "headers" and such that make the output hard to parse for a subsequent process. By combining such simple filters using pipes, useful functionality can be achieved. E.g. the command line

```
    tr -cs "[:alpha:]" "[\n*]" <input | sort | uniq |
comm -23 - /usr/dict/words
```

uses several programs connected using pipes to "spell check" the text in the file `input`. Roughly, the above pipeline works as follows:

1. **tr** splits its input into words (1 per line, with many empty lines) by replacing all non-alphabetic characters with `<newline>`'s.
2. then **sort** sorts the words in alphabetic order while
3. **uniq** will remove all (consecutive) duplicate lines.
4. Finally, **comm** compares its standard input (indicated by its "−" argument) with the dictionary file */usr/dict/words*[22]. The option `-23` ensures that only lines (words) that appear in the input but not in the dictionary, make it to the standard output.

Thus the whole pipeline produces the spelling errors[23], i.e. the words that appear in the file `input` but not in the dictionary, on its standard output.

4.2.7 Shell variables

The shell supports the manipulation of (an extension of) the *environment*, see section 3. The names in the environment are called (shell) *variables*, which are all string-valued[24]. Assignment is done using "=" as in

```
PATH=.:/usr/local/bin:/usr/bin
```

[22] `/usr/dict/words` contains approx. 25000 english words.
[23] Admittedly, we use a rather naive notion of "spelling error".
[24] **Bash** supports a wider selection of variables, e.g. arrays.

(note that there should be no space around "="). Changes to variables are strictly local to the current process; one can use the *export* built-in command to ensure that changes are also propagated to child processes. Thus, if one wants to ensure that `PATH` retains its new value also in child processes (whether they be shell or other programs), the above becomes.

```
PATH=.:/usr/local/bin:/usr/bin; export PATH
```

Here `;` is used to separate commands.

The value of a variable can be accessed by preceding it with a "`$`", e.g.

```
echo $PATH | tr ":" "\n"
```

will print the directories in `PATH`, one per line: **echo** simply copies its arguments (here the value of the `PATH` variable) to its standard ouput while **tr** replaces each "`:`" by a `<newline>`.

The shell supports a number of special variables, some of which are shown in figure 6.

name	**what**
?	the exit status (see section 3.4) of the last command executed.
0	the pathname of the command being executed
i	(`i` is a number) the i'th command argument
#	the number of command arguments
$	the PID of this shell process
HOME	the user's home directory
PATH	list of directories, separated by "`:`", where executable files are searched for
MANPATH	like PATH, but these are the directories where **man** will try to find the requested manpages.
IFS	characters that can separate words in a command line
PS1	the (primary) prompt
PS2	the secondary (for continuation lines) prompt

Figure 6: Some special shell variables

E.g.

```
PS1="yes, dear?  "
```

(note the use of double quotes to force the shell to treat "yes, dear? " as a single word, see section 4.2.11) will cause the shell to prompt for subsequent commands using *yes, dear?* as a prompt.

4.2.8 Filename substitution

The shell replaces certain patterns on a command line by pathnames that match the pattern. E.g.

```
vi *.C
```

will be processed[25] as if the user typed

```
vi prog.C help.C
```

if `prog.C` and `help.C` are the only files in the current directory whose names end with `.C`.

The meta-characters used for filename substitution are listed in the figure 7.

character	**meaning**
*	any string not starting with dot ("`.`")
?	any character except fot ("`.`")
[..]	any of the characters between the brackets
[c_1-c_2*]*	any character between c_1 and c_2 (inclusive)
other character	stands for itself

Figure 7: Filename substitution patterns

E.g. the pattern

```
chapter[1-9].tex
```

will match all filenames starting with "`chapter`" followed by a digit followed by "`.tex`" while

```
ls /export/home/*/public_html
```

lists all "home page" directories on the system.

4.2.9 Command substitution

The shell can also perform *command substitution*: any part of the command line between backquotes ("`") will be executed as a separate command and its output will be used to replace the original command.

E.g.

[25] **vi** (**vi**sual editor) is an editor program that is available on all unix systems.

```
vi `grep -il dirk messages/*`
```

will edit (using **vi**) all messages containing "`dirk`" (the **grep** (**g**et **re**gular ex**p**ressions) command searches files for the occurrence of patterns).

```
SOURCES=`echo *.C`
```

will assign the list of all C++ source file names to the shell variable `SOURCES`.

4.2.10 Quoting

The special meaning of a special character (e.g. the ones in figure 7) can be taken away (this is called *quoting*) by putting \ in front of it (to quote a \, type \\).
Single (`'`) and double (`"`) quotes can also be used for quoting:

- *Single quotes* (`'`) prevent the special interpretation of any characters (except another single quote) [26]. E.g.

  ```
  echo '?*.C\'
  ```

  will simply **echo** `?*.C\`.

- *Double quotes*(`"`) prevent filename substitution but not variable- of command substitution. A single quote between double quotes has no special meaning. E.g.

  ```
  echo "* '"
  ```

  will **echo** `* '`.

Anything between single or double quotes will be considered to be one word. E.g. `prog *` is different from ``prog "`ls *`"``[27]: in the former case, `prog` will receive a parameter for each filename in the current directory while in the latter case, `prog` will get only a single parameter consisting of a list (separated by spaces) of all filenames in the current directory.

[26] To get an uninterpreted single quote, use `\'` outside of single quotes or between double quotes, i.e. "`'`".

[27] Notice the command substitution inside the double quotes.

4.2.11 Shell command line processing revisited

A more detailed version of the processing of a command line by the shell, showing the relationship between the various substitution and quoting mechanisms, is shown below.

```
while (true) {
  bool background = false;
  show prompt $PS1 on screen # if interactive
  read a commandline from input
  if (the commandline consists of <eof> only)
    exit
  if (the commandline ends with '&')
    background = true

  # do transformations
  perform variable substitution, i.e. replace each
    variable reference $var by its value
    (or the empty string, should var not be defined)
  perform command substitution
  split commandline into words, using characters from
    $IFS as separators but keep text between
    single or double quotes as a single word
  perform filename substitution, each filename is a new
    word

  find an executable file F that corresponds to
    the first word (taking $PATH into account)
  create a new process and
    exec(F, path-of-F, 2nd word, 3rd word, ..)
  if (not background)
    wait for the process to finish
  }
```

4.2.12 Bash startup files

When a user logs in, **bash** executes all command lines in `$HOME/.profile` or `$HOME/.bash_profile`. Whenever a shell (script) is started, all commands in `$HOME/.bashrc` are also executed. In order to prevent confusion, it is probably easier (and it avoids some problems with **CDE**) to make `$HOME/.bash_profile` a (hard) link to `$HOME/.bashrc`.

The following is a typical `.bashrc` file.

```
# $Id: sample.profile.sh,v 1.3 1999/08/03 08:45:53 dvermeir Exp $
#          Minimal .profile file
```

```
#          Note the \ at the end of a line: it indicates
#          that the next line is a continuation of the same
#          command.
#          Note also that bash (but not the original Bourne
#          shell /bin/sh) allows ``export var=blabla''
#          as a shorthand for ``var=blabla; export var''
#
export OPENWINHOME=/usr/openwin
export CVSROOT=$HOME/cvsroot
export PATH=".:\
$HOME/bin:\
/usr/local/bin:\
/usr/bin:\
$OPENWINHOME/bin:\
/usr/dt/bin:\
/usr/ccs/bin:\
/usr/ucb/bin:\
/usr/ucb:
#
export MANPATH="\
/usr/local/man:\
$OPENWINHOME/man:\
/usr/dt/man:\
/usr/share/man:\
/usr/man:"
#
export LD_LIBRARY_PATH="\
/usr/local/lib:\
$OPENWINHOME/lib:"
# EXINIT initializes vi preferences
export EXINIT='set ai sm magic wm=2 shell=/usr/local/bin/bash'
export PS1='\# \h:\w\$ '
export HISTFILE=
# my favorite editor
export EDITOR=vi
# locale stuff
export LANG=en_US
export LC_CTYPE=en_US
export LC_NUMERIC=en_US
export LC_TIME=en_US
export LC_COLLATE=en_US
export LC_MONETARY=en_US
export LC_MESSAGES=C
export LC_ALL=
# some shortcut versions of often used commands
alias ls='ls -F'
alias la='ls -a'
alias j=jobs
alias h='history 20'
```

```
alias from_profile=
# if i use emace, i want white background
alias em='emacs -i -bg white'
# tell bash I want to edit command lines using vi-style commands
set -o vi
```

4.3 Shell scripts

4.3.1 Control structures

Nobody wants to type the command line in section 4.2.6 more than once. That's what *shell scripts* are for: you package one or more commands into an executable text file, put this file somewhere in a `$PATH` directory and use its name as a new command.

Thus we could create an executable[28] file "`sspell`" (simple **spell**, **spell** already exists) with the contents shown below.

```
#!/bin/sh
# $Id: sspell.sh,v 1.2 1999/08/02 11:06:26 dvermeir Exp $
#
#        Usage: sspell
#
#        will flag any words from stdin that are not in /usr/dict/words
#
tr -cs "[:alpha:]" "[\n*]" | sort | uniq | comm -23 - /usr/dict/words
```

Notice the first line: it ensures (see section 3.2) that the file, when executed, will be interpreted by **/bin/sh**, the omnipresent Bourne shell (conveniently, "#" causes **/bin/sh** to consider the rest of the line as a comment).

Commands in a script can be combined sequentially by just inserting them one after the other (on different lines or separated by "`;`") in the script, but more sophisticated control structures are also available. These include, `if...then...else`, `while`, `case` and others, see the **manual**.

Conditions are commands; a condition is true iff the command returns 0, see section 3.4. Consider e.g. the following script.

```
#!/bin/sh
# $Id: waitfor.sh,v 1.2 1999/08/02 11:06:27 dvermeir Exp $
# usage: waitfor name
# will wait until user called "name" is logged in
while true # i.e. forever
```

[28] Use **chmod** to set the correct permissions.

```
do
        if who | grep $1 >/dev/null # don't want to see output
        then
                echo "$1 is here!"; exit 0
        else
                echo "$1 not logged in"
        fi
        sleep 30 # wait 30 secs before next try
done
```

It contains the command

```
who | grep $1
```

as a condition: **who** reports who is currently logged in and **grep** checks whether the first argument string "`$1`" is among them.

The following example prints nicely formatted listings of C++ source files.

```
#!/bin/sh
# $Id: lpcpp.sh,v 1.3 1999/08/03 08:45:53 dvermeir Exp $
#
#       usage: lpcpp c++-source-file..
#
#       print c++ source files, e.g.
#
#               lpcpp f.C f.h g.C
#
for f   # for every argument filename f, in the example f.C, f.h, g.C
do
        if test -r $f # only if there is a readable file $f
        then
                F=`echo $f | sed -e 's/\./_/g'` # replace '.' by '_' in f,
                        # latex does *not* like filenames with more than 1 dot..
                lgrind -lc++ $f >tmp_$F.tex # translate source to tex input, e.g. on tmp_f_C.tex
                latex tmp_$F >/dev/null  # throw away tex's blurb, result in tmp_f_C.dvi
                dvips tmp_$F.dvi -o tmp_$F.ps # translate dvi to postscript, e.g. on tmp_f_C.ps
                lp -c tmp_$F.ps # and print
        fi
done
/bin/rm -f  tmp_*       # clean up by removing all temporary files
```

In the example, the body of the "for" loop is executed once for each argument `$f` presented to the script. In the body, first the (built-in) program **test** is executed which checks the readability of the file `$f`. If this is ok, **echo** sends `$f` to **sed**'s[29] standard input. **Sed** substitutes (using a "`s`" editor command) all occurrences of "`.`" in `$f` by underscore (_). The result is saved in a variable `$F`, which will be the base for some temporary filenames. Then *lgrind* is called to translate C++ source to **latex** input in a temporary file "`tmp_$F.tex`". E.g. if the original filename is "`f.C`" then the **latex** version will be in "`tmp_f_C.tex`". Next **latex** processes `tmp_$F.tex`, resulting in a "`.dvi`" ("*device indepdendent file*") file `tmp_$F.dvi`. Afterwards, *dvips* is used to translate `tmp_$F.dvi` to postscript in `tmp_$F.ps`. Finally, **lp** will send the postscript file to the printer.

4.3.2 Here documents

If a script needs a substantial amount of template text, *here documents* are often more convenient than storing the template in a separate file. A line

```
command <<MARKER
```

where `MARKER` is an arbitrary string, will cause `command` to take its standard input from the text on the lines following the present one until (and not including) a line that starts with "`MARKER`". Variable substitution is done in the template text, unless `MARKER` is preceeded by a \ (or the "`$`" in the variable reference is quoted).

The example below is run each night to create an html page containing hypertext pointers to all home pages on the system.

```
#!/bin/sh
# $Id: genhomepages.sh,v 1.3 1999/09/29 10:50:48 dvermeir Exp $
HOST=`hostname`
# output here document containing first part of page
# note that EOF marker is not quotes as we want $HOST
# to be replaced by its value.
cat <<EOF
<html>
<head>
<title>
$HOST home pages
</title>
</head>
<body>
<body bgcolor="#FFFFFF">
<h2>$HOST home pages</h2>
<ul>
```

[29] Sed is a very efficient stream-oriented text editor

```
EOF
# sort passwd file on the name field
cat /etc/passwd | sort -t : -k 5,5 |
while read line # for each line in the passwd file, i.e. for each user
do
        # build H/public_html/index.html where H is her home directory
        homepage=`echo $line | awk -F: '{print $6}'`/public_html/index.html
        if [ -f $homepage -and -r $homepage ] # if the page exists
        then
                # determine login name (field 1 in line) and
                # full name (field 5 in line)
                login=`echo $line | awk -F: '{print $1}'`
                name=`echo $line | awk -F: '{print $5}'`

                if [ -z "$name" ] # if full name is unknown
                then # use login name instead
                        name=$login
                fi
                # generate ``hyper reference'', note quoting of double quotes
                echo "<li><a href=\"~$login/\">$name</a>"
        fi
done
today=`date`
# Add a ``last change'' clause to the end of the page
cat <<EOF
</ul>
<hr>
Last Change: $today
</body>
</html>
EOF
```

Note the use of the **awk** command. **Awk** is one of a number of “little languages” that provide simple but powerful text processing facilities.

5 Text editors

Probably the most important program for a developer is a text editor. Note that we do mean a *text editor*, not a *word processor*. A good editor should make it easy to manipulate raw text (e.g. source code).

Besides extremely simple (and limited) toy editors like *pico*, there are basically two choices: you either go for **vi** or its more recent extension **vim**, or you go for **emacs**.

The difference between the two can be summarized by the quote[30]

> "**vi** is the god of the editors, but **emacs** is the editor of the gods"

Vi is small and efficient. The basic operations are very simple. Powerful commands (especially for searching and replacing text) are available. However, possibilities for customisation are limited. The latter disadvantage has been largely overcom in **vim** which also supports syntax highlighting, filename completion and much more. A **tutorial**[31] and **reference card**[32] for **vi** are available.

Emacs is programmable in a lisp-like language; it integrates with lots of other software tools (such as **cvs** and **make**). People that use **emacs** tend to stay in the editor at all times; there seems to be little that cannot be done (including web browsing) from within **emacs**.

[30] Unfortunately, I don't remember the source.
[31] http://tinf2.vub.ac.be/ dvermeir/manuals/vitutorial/index.html
[32] http://tinf2.vub.ac.be/ dvermeir/manuals/vi.ps

6 Make

Altough scripts are useful to make customised commands for repetitive tasks, they are no good in deciding whether it is actually necessary to perform such a task. E.g. when developing a software system that involves many source files, it may become a nuisance to figure out which files need to be recompiled because of recent changes to other files. The **make** program is extremely useful to make this decision for you.

Typing

```
make
```

will cause **make** to look for a file called `Makefile` (or `makefile`) in the current directory. This `Makefile` contains rules describing dependencies between files. It then proceeds to (recursively) verify whether the target files are "up-to-date" (i.e. not older than any files they depend upon) and, if necessary, executes commands to regenerate targets using appropriate commands.

6.1 Make basics

Consider the following C++ program.

```
// $Id: hello.C,v 1.2 1999/08/02 10:47:06 dvermeir Exp $
#include        <iostream>

int
main(int argc,char *argv[])
{
cout << "hello world" << endl;
}
```

Below is a simple, but unnecessarily large (see below), `Makefile` for this program.

```
# $Id: Makefile,v 1.3 1999/08/03 08:45:54 dvermeir Exp $
hello:          hello.C
                g++ -o hello hello.C # action lines start with <tab>s!
```

It contains a single rule of the form shown in figure 8.

$$
\begin{array}{rcl}
\mathbf{target}_1 \ldots \mathbf{target_n} & \mathbf{:} & \mathbf{dependency}_1 \ldots \mathbf{dependency}_m \\
\texttt{<tab>} & & \mathit{command}_1 \\
\texttt{<tab>} & & \mathit{command}_2 \\
\ldots & & \ldots
\end{array}
$$

Figure 8: Format of a rule in a Makefile

- A rule starts with a number of *targets* (usually these are names of files). In the example, the target is "`hello`" (the name of the executable file for the program `hello.C`).
- The right hand side (after the colon "`:`") contains the names of files on which the targets depend, these files are called *dependencies*. The only dependency in the example is the source file `hello.C`.
- The second and following lines (which should all start with a `<tab>` character) contain (in order) *actions* needed to reconstruct the target from the dependencies. In the example, the command uses **g++** to compile (and link) the program source `hello.C`, saving the result in the executable file `hello`.

Now we can run make.

```
tinf2% rm -f hello
tinf2% make hello
g++ -o hello hello.C
tinf2%
```

The **make** command takes "target" files as (optional) arguments. If no arguments are specified, **make** will process the target of the first rule in the *Makefile*.

Make will ensure that all argument target files are (made) up-to-date. To achieve this, it finds the rules for the target, then recursively processes all its dependencies as subtargets, ensuring that these dependencies are up-to-date. Finally, if any of the (possibly reconstructed) dependencies is younger than the target, the actions in the rules for the target are executed (note that there may be several rules with the same target but only one of those may contain actions) in order to reconstruct the target.

In pseudocode, this becomes

```
make(target t)
{
```

```
find rules r that have t as a target
# only 1 of these rules should have actions
let deps = all dependencies of t

for each d in deps
do
        make(d) # recursive call
done
if any file in deps is younger than t
then
        execute actions from r
fi
}
```

To illustrate what happens if the source file is modified, we update its "last modified" timestamp using the **touch** command.

```
tinf2% touch hello.C # updates "last modified" time
tinf2% make hello
g++ -o hello hello.C
tinf2%
```

The next example shows a `Makefile` for a program that has two source files, `hello.C`

```
// $Id: hello.C,v 1.2 1999/08/02 10:47:11 dvermeir Exp $
#include        <iostream>
#include        "message.h"
int
main(int argc,char *argv[])
{
cout << message << endl;
}
```

and `message.C`.

```
// $Id: message.C,v 1.2 1999/08/02 10:47:11 dvermeir Exp $
#include        "message.h"

const string message("hello world");
```

`Message.C` has an associated "*header file*" `message.h`

```
#ifndef  MESSAGE_H
#define  MESSAGE_H
// $Id: message.h,v 1.3 1999/08/07 08:54:50 dvermeir Exp $
```

```
#include      <string>

extern const string      message;
#endif
```

The `Makefile` shows that e.g. the "*object file*" `hello.o` must be linked into the final executable file `hello`. `Hello.o` depends not only on the source file `hello.C` but also on the header file `message.h`, as can be read from the rule corresponding to the target `hello.o`. Note also that **make** considers "`#`", and anything following it on the same line, to be a comment.

```
# $Id: Makefile,v 1.2 1999/08/02 10:47:11 dvermeir Exp $
hello:          hello.o message.o
                g++ -o hello hello.o message.o
hello.o:        hello.C message.h # because hello.C includes message.h
                g++ -c hello.C
message.o:      message.C message.h # because message.C includes message.h
                g++ -c message.C
```

```
tinf2% make # default is first target in Makefile
g++ -c hello.C
g++ -c message.C
g++ -o hello hello.o message.o
tinf2%
```

6.2 Pattern rules

Many rules are similar, especially with respect to the action part. E.g. a file *f.o* can be reconstructed from a C++ source file *f.C* by executing

```
g++ -c f.C
```

Such "patterns" can be specified in a `Makefile` using *pattern rules*.

The above pattern is written in a pattern rule as follows:

```
%.o:  %.C
      g++ -c $<
```

Such a rule looks very much like a normal rule but:

- it uses "%" to match any string in the target, occurrences of % in the dependencies are replaced by the same string. Using the example rule, **make** would deduce that *f.o* depends on *f.C* (if it exists) by replacing % in the pattern rule by "*f*". Note that % is not allowed in the action section of a pattern rule.

- *automatic variables* (see section 6.3) can be used in commands to refer to (parts of) the target and/or dependencies (such variables can also be used in normal rules). In the example the automatic variable `$<` stands for "the first dependency" of the target. If the pattern rule would be instantiated with "*f*" for %, the value of `$<` would be "*f.C*".

If a pattern rule contains several targets, then **make** assumes that all targets are made at the same time by the actions. E.g., **bison** is a parser generator that, when presented with a file *f.y* containing a grammar, will produce two files, *f.tab.c* and *f.tab.h* containing the source of a C function implementing a parser for this grammar. This can be represented using the following pattern rule

```
%.tab.c %.tab.h:  %.y
                  bison -d $<
```

Thus, if **make** needs `f.tab.c` and `f.tab.h` it will run

```
bison -d f.y
```

only once to produce both targets.

Below, we show a version of a `Makefile` with pattern rules for the example from section 6.1. Although the example `Makefile` has become a bit longer[33], one should appreciate that, in a more realistic case, there will be many more C++ source files and the pattern rule will turn out to be a real saving.

```
# $Id: Makefile,v 1.4 1999/08/03 08:45:55 dvermeir Exp $
%.o:            %.C
                g++ -c $< # '$<' is first dependency
hello:          hello.o message.o
                g++ -o hello hello.o message.o
hello.o:        hello.C message.h # no need for action, provided by pattern rule
message.o:      message.C message.h
```

Note that, taking into account pattern rules, the above `Makefile` has several rules for e.g. `hello.o`. However, since only one of them (the one instantiated from the pattern rule) has actions, this will not confuse **make**.

[33] It will get shorter in the next sections.

```
tinf2% rm hello *.o # remove all that can be reconstructed
tinf2% make
g++ -c hello.C # 'hello.C' is first dependency
g++ -c message.C # 'message.C' is first dependency
g++ -o hello hello.o message.o
tinf2%
```

6.3 Automatic variables

As shown in section 6.2, automatic variables that represent selected components of the (target/dependency part of the) rule are almost necessary in pattern rules (without such variables, action instantiations could not depend on dependencies and/or targets). Such variables are also useful in ordinary rules, as can be seen in yet another version of a `Makefile` for the example in section 6.1.

```
# $Id: Makefile,v 1.2 1999/08/03 08:45:55 dvermeir Exp $
%.o:            %.C
                g++ -c $< # '$<' is first dependency
hello:          hello.o message.o
                g++ -o $@ $^ # '$
hello.o:        hello.C message.h # no need for action, provided by pattern rule
message.o:      message.C message.h
```

Figure 9 shows most of the automatic variables that are understood by **make**.

6.4 Other uses of make

The use of **make** is, of course, not limited to software development.

The `Makefile` below illustrates how one can easily write rules that maintain a text (like the present one) that is available as a postscript and as an html file; both of which are generated from a single **latex** source file.

```
# $Id: Makefile,v 1.3 1999/08/03 08:45:55 dvermeir Exp $
%.eps:          %.fig
                fig2dev -L ps -p portrait $< >$@
%.dvi:          %.tex
                latex $<
%.ps:           %.dvi
                dvips $^ -o $@
all:            srd.ps srd.html # no action, just make dependencies
srd.dvi:        srd.tex sysmodel.eps webindexer.pvs.tex
srd.html:       srd.dvi webindexer.pvs rfc1808.txt
                tth -Lsrd <srd.tex >$@
```

`$@`	the target
`$<`	the first dependency
`$?`	the dependencies that are newer than the target
`$^`	all dependencies (from all rules for the target)
`$*`	the stem of the target (corresponds to %)
`$(@D)`	the directory of the target
`$(@F)`	the file-within-directory of the target
`$(*D)`	the directory of the stem of the target
`$(*F)`	the file-within-directory of the stem of the target
`$(<D)`	the directory of the first dependency
`$(<F)`	the file-within-directory of the first dependency
`$(^D)`	the directories of the dependencies
`$(^F)`	the file-within-directory's of the dependencies
`$(?D)`	the directories of the dependencies that are newer than the target
`$(?F)`	the file-within-directory's of the dependencies that are newer . . .

Figure 9: Make automatic variables

Here, *fig2dev* is a program that translates *fig* files (that can be produced, e.g. by the drawing program *xfig*) to any of a large number of formats (here: postscript). **Latex** produces *.dvi* (see section 4.3.1) files from *.tex* source files while *dvips* translates *.dvi* files to *postscript* files. On the other hand, **tth** translates latex source files to *.html* files.

The `Makefile` contains `all` as the first target of the first (non-pattern) rule. Hence typing

```
make
```

is equivalent to

```
make all
```

The rule for `all` does not contain any actions. The net result is that **make** will simply ensure that `all`'s dependencies, `srd.ps` and `srd.html` are up-to-date.

```
tinf2% make -n # show commands without executing
fig2dev -L ps -p portrait sysmodel.fig >sysmodel.eps # 'sysmodel.eps' is targe
latex srd.tex
dvips srd.dvi -o srd.ps
tth -Lsrd <srd.tex >srd.html
tinf2%
```

6.5 Built-in rules, automatically generating dependencies

Make has a large number of built-in pattern rules that know about C, C++, ... compilers, yacc, bison, lex, flex etc.

This allows us to reduce the `Makefile` for the example in section 6.1 still further.

```
# $Id: Makefile,v 1.2 1999/08/02 10:47:18 dvermeir Exp $
hello:          hello.o message.o
                g++ -o $@ $^
hello.o:        hello.C message.h # apply built-in rules
message.o:      message.C message.h
```

```
tinf2% make
g++    -c hello.C -o hello.o
g++    -c message.C -o message.o
g++ -o hello hello.o message.o
tinf2%
```

Better still, we can also generate the dependencies for `hello.o` and `message.o` automatically, using the *-M* option of **g++** (the C++ compiler) and **make**'s *include* directive.

```
# $Id: Makefile,v 1.2 1999/08/02 10:47:21 dvermeir Exp $
hello:          hello.o message.o
                g++ -o $@ $^
include make.depend
make.depend:    hello.C message.C
                g++ -M $^ >$@
```

```
tinf25 make
Makefile:4: make.depend: No such file or directory
g++ -M hello.C message.C >make.depend
g++    -c hello.C
g++    -c message.C
g++ -o hello hello.o message.o
tinf2%
```

Make first complains about the missing file `make.depend` that it is supposed to include, but then thinks better of it and generates this file using the last rule in the `Makefile`. Note also that, because of the dependencies of the target `make.depend`, this file will be regenerated each time one of the source files has been updated.
A fragment of the `make.depend` file generated for the example is shown below.

```
hello.o: hello.C /usr/local/include/g++/iostream \
 /usr/local/include/g++/iostream.h /usr/local/include/g++/streambuf.h \
 /usr/local/include/g++/libio.h \
 /usr/local/sparc-sun-solaris2.7/include/_G_config.h \
 /usr/local/lib/gcc-lib/sparc-sun-solaris2.7/egcs-2.91.66/include/stddef.h \
 message.h /usr/local/include/g++/string \
 /usr/local/include/g++/std/bastring.h /usr/local/include/g++/cstddef \
 /usr/local/include/g++/std/straits.h /usr/local/include/g++/cctype \
 /usr/include/ctype.h /usr/include/sys/feature_tests.h \
 /usr/include/sys/isa_defs.h /usr/local/include/g++/cstring \
 /usr/include/string.h /usr/local/include/g++/alloc.h \
 /usr/local/include/g++/stl_config.h \
 /usr/local/include/g++/stl_alloc.h /usr/include/stdlib.h \
 /usr/local/sparc-sun-solaris2.7/include/assert.h \
 /usr/local/include/g++/iterator /usr/local/include/g++/stl_relops.h \
 /usr/local/include/g++/stl_iterator.h \
 /usr/local/include/g++/std/bastring.cc
```

6.6 Make variables

If we add a new source file to the example of section 6.1, updating the `Makefile` in the previous section involves adding two filenames (one with extension `.C`, one with extension `.o`) to the list of dependencies of the rules for `hello` and `make.depend`. This work can be reduced by using user-defined *variables*, as illustrated in the `Makefile` below.

```
# $Id: Makefile,v 1.2 1999/08/02 10:47:23 dvermeir Exp $
sources=        hello.C message.C
hello:          $(sources:%.C=%.o)
                g++ -o $@ $^
include make.depend
make.depend:    $(sources)
                g++ -M $^ >$@
clean:
                rm -f $(sources:%.C=%.o) hello
```

Now we only need to add the new file to the definition of the *sources* variable.

A **make** variable definition has the form

```
name = text
```

which must all be on a single line, but *continuation lines* may be used by escaping the `<newline>` on the previous line using a \[34] (this continuation line mechanism works also in rules).

The `Makefile` above could be written using continuation lines, as shown below.

[34]Note that \ **must** be the **last** character on the line.

```
# $Id: Makefile,v 1.2 1999/08/02 10:47:25 dvermeir Exp $
sources=        hello.C \
                message.C
hello:          $(sources:%.C=%.o)
                g++ -o $@ $^
include make.depend
make.depend:    $(sources)
                g++ -M $^ >$@
```

A reference to a variable "*name*" has the form

```
$(name)
```

References can use pattern matching to alter the resulting value. E.g. in the example reference `$(sources:%.C=%.o)`, the strings in the value of *sources* will be matched with the pattern "`%.C`" and then be transformed to "`%.o`", where "`%`" is replaced by the part that matches "`%`" in the pattern "`%.C`". This results in the value "`hello.o message.o`" for the reference `$(sources:%.C=%.o)` if the value of "*sources*" is "`hello.C message.C`".

The above `Makefile` also illustrates the use of a target without any dependencies: the rule for *clean* had no dependencies. Moreover, the rule's actions do not create a file "`clean`". Thus

```
make clean
```

will cause **make** to attempt to create a file "`clean`" by executing the associated actions.

6.7 Built-in variables used by built-in rules

The built-in rules of **make** (see section 6.5) use several variables, defining e.g. the name of the C++ compiler, the linker, the flags passed to the linker etc. Some of those variables are listed in figure 10 (see also the **make** manual).

Customising these variables in a `Makefile` saves the bother of writing ad-hoc rules, e.g. to link the executable file from the objects files, as in the example below.

```
# $Id: Makefile,v 1.3 1999/08/02 11:06:29 dvermeir Exp $
sources=        hello.C message.C
objects=        $(sources:%.C=%.o)
#
CXXFLAGS=       -O2 # flags for g++: optimize
```

CPP	C and C++ preprocessor
CPPFLAGS	flags passed to $(CPP)
CC	C compiler, linker (for both C and C++)
CFLAGS	flags passed to C compiler
CXX	C++ compiler
CXXFLAGS	flags passed to $(CXX)
LDFLAGS	linker flags
LDLIBS	libraries to link with
MAKE	the name of the "make" program

Figure 10: Some built-in vars of make

```
CPPFLAGS=       -I/usr/local/include # g++ preprocessor flags
LDFLAGS=        -R/usr/ucblib:/usr/local/lib # linker flags
LDLIBS=         -L/usr/local/lib -ltbcc # linker libraries
CXX=            g++ # (default) use g++ as C++ compiler of .cc, .C files
CC=             g++ # use g++ as linker to ensure linking with C++ library
#
hello:          $(objects)
include make.depend
make.depend:    $(sources)
                g++ -M $^ >$@
clean:
                rm -f $(sources:%.C=%.o) hello
```

The built-in rule for linking can be written as a pattern rule.

```
%:  %.o
    $(CC) $(LDFLAGS) $^ $(LOADLIBS)
```

In the example, we use *LDFLAGS* to set the *run-time library path*, the list of directories (separated by "`:`" colons) where the program will attempt to find any dynamically linked shared libraries it wants to use.

```
tinf2% make
Makefile:13: make.depend: No such file or directory
g++ -M hello.C message.C >make.depend
g++  -O2  -I/usr/local/include   -c hello.C
g++  -O2  -I/usr/local/include   -c message.C
g++  -R/usr/ucblib:/usr/local/tbcc/lib   hello.o message.o \
         -L/usr/local/tbcc/lib -ltbcc  -o hello
tinf2%
```

6.8 Recursive make

It is possible for **make** to recursively call itself from an action.

```
# $Id: Makefile,v 1.2 1999/08/02 10:47:08 dvermeir Exp $
#
# modules (subdirectories), topologically sorted according to dependencies
#
MODULES=	url inet docu db indexer cgi query
#
# note the use of shell variables in make: $$x i/o $x
# also note the use of '\' to put shell commands on 1 command line
all:
	for m in $(MODULES); \
	do \
		( cd $$m; $(MAKE) all install; ) \
	done

install check clean:
	for m in $(MODULES); \
	do \
		( cd $$m; $(MAKE) $@; ) \
	done
```

Note that the "for" (shell) statement is a single action and thus **make** wants it on a single line. This implies

1. using "`;`" to separate the shell statements (see section 4.2.7), and
2. using continuation lines (see section 6.6) if we want to spread the shell statements over several "editor lines". Also, shell variables (which are different from "make variables") are referenced using 2 "`$`" signs, to avoid confusing **make**.

6.9 Beyond make

For complex and portable packages that need to compile and install on many different platforms, writing makefiles becomes a complex task. In such a case, it is better to use more sophisticated tools such as **autoconf**, **automake** and **libtool** that automatically generate makefiles from higher level specifications. The **autocookbook** tutorial may be useful for learning about these tools, which are often used in conjunction with the **cvs** configuration management system.

Index

’, 28
*, 27
-M, 43
., 6
.., 6
.dvi, 42
.html, 42
.tex, 42
/bin/sh, 31
/dev/null, 12, 24
/etc/passwd, 8, 20
/usr/dict/words, 25
/usr/local/Hughes/msql2.sock, 13
?, 26, 27
[c_1-c_2], 27
[..], 27
#, 26, 39
$, 26
$<, 40
%, 40

0, 26

absolute pathname, 5
access control lists, 9
actions, 37
address space, 15
append standard output to file, 24
apropos, 4
arguments of process, 16
autoconf, 47
autocookbook, 47
automake, 47
automatic variables, 40
Awk, 34
awk, 34

background processes, 23
Bash, 22, 23, 25
bash, 20, 22, 23, 29
bison, 40
block-special files, 12

cat, 24
CC, 46
cd, 6
CDE, 4, 7, 29
CFLASG, 46
character-special files, 12
chmod, 8, 15, 31
chroot, 15
clean, 45
close, 19
comm, 25
command line interpreter, 16
command substitution, 27
continuation line, 44
cp, 6
CPP, 46
CPPFLAGS, 46
cvs, 35, 47
CXX, 46
CXXFLAGS, 46

dependencies, 37
device indepdendent file, 33
df, 9, 10
directories, 6
directory, 5, 12
double quote, 28
double-indirect blocks, 10
dup, 18, 19
dvips, 33, 42

echo, 26, 28, 33
effective, 15
Emacs, 35
emacs, 35

environment, 16, 25
exec, 16–18, 22
execute, 8
exit, 19
export, 26

fcntl, 19
fd, 12
fd table, 18
fig, 42
fig2dev, 42
file creation mask, 15
file descriptor, 12, 18
file descriptor table, 18
file descriptors, 15
file systems, 9
filter, 25
foreground, 23
fork, 14, 16
ftp, 15

g++, 37, 43
GID, 8, 15
grep, 28, 32
group, 8
group id, 8, 15
group owner, 8

hard links, 11
header file, 38
here documents, 24, 33
history, 23
HOME, 26
home directory, 8

IFS, 26
include, 43
indirect block, 10
init, 15
inode number, 6
inodes, 10
iocntl, 19

KDE, 4
kill, 15

Latex, 42
latex, 24, 33, 41
LDFLAGS, 46
LDLIBS, 46
lgrind, 33
libtool, 47
links, 6
linux, 4
ln, 6, 13
lockf, 18
lp, 33
Ls, 7
ls, 7, 8

MAKE, 46
Make, 37, 43
make, 23, 35–37, 39–45, 47
    continuation line, 44, 47
    include, 43
make variables, 44
Makefile, 37
man, 4, 26
MANPATH, 26
manual, 31
mmap, 19
mount, 9, 10
mount point, 9
msql, 13
mv, 6

named pipe, 12
nfs, 10, 11
nice, 15

object file, 39
open, 18
open source, 4
other users, 8
owner, 8

parent process, 14
password, 8
PATH, 22, 26
pattern rules, 39
pico, 35
PID, 15
pipe, 24
    named, 12
postscript, 42
proc, 12
process ID, 15
process table, 15
processes, 14
program, 14
ps, 14, 15
PS1, 26
PS2, 26
pwd, 6

quoting, 28

read, 8, 19
redirect, 24
reference card, 35
regular files, 12
relative pathname, 6
renice, 15
restricted shell, 20
rm, 6
root, 8, 9
root file system, 9
run-time library path, 46

Sed, 33
sed, 33
set-uid, 15
share, 10
shell, 16, 20
shell program, 8
shell scripts, 31
single quote, 28
socket, 13
sort, 25
spell, 31
standard error, 18
standard input, 18
standard output, 12, 18
stderr, 18, 24
stdin, 18, 24
stdout, 18, 24
stream, 24
super user, 8
swap, 12
symbolic link, 13

target, 37
test, 32
text editor, 35
tmpfs, 11
touch, 38
tr, 25, 26
triple-indirect blocks, 10
tth, 42
tutorial, 35

ufs, 11
UID, 8, 15
uintro.ps, 1
umask, 15
uniq, 25
unix sockets, 13
unlink, 6
user id, 15
    UID, 8
user name, 8

variables, 25
Vi, 35
vi, 27, 28, 35
vim, 35
virtual file system, 11
vnode table, 18

waitpid, 16

wc, 25
wendy, 4
who, 25, 32
word processor, 35
write, 8, 19

xfig, 42

List of Figures

1 The unix directory structure 5
2 File access permissions 8
3 Mounting file systems 9
4 Traditional layout of a file system on disk 11
5 File I/O implementation 19
6 Some special shell variables 26
7 Filename substitution patterns 27
8 Format of a rule in a Makefile 37
9 Make automatic variables 42
10 Some built-in vars of make 46