# Chapter 2 Login Records, File I/O, and Performance

## Concepts Covered

*Man pages and Texinfo pages*
*The UNIX file I/O API*
*Reading, creating, and writing files*
*File descriptors*
*Kernel buffering*
*Kernel versus user mode and the cost of system calls*
*Timing programs*
*Time representation in UNIX*
*The utmp file*
*Detecting and reporting errors in system calls*
*Memory-mapped I/O,*
*Feature test macros*
*open, creat, close, read, write, lseek, perror, ctime, localtime, utmpname, getutent, setutent, endutent, malloc, calloc, mmap, munmap, memcpy*
*Filters and regular expressions*

## 2.1 Introduction

This chapter introduces the two primary methods of I/O possible in a UNIX: *buffered* and *unbuffered*. By trying to write the `who` and `cp` commands, we will learn explore how to create, open, read, write, and close arbitrary files. "Arbitrary" in this context means that they are not necessarily text files. We will write several different versions of the `who` command, simply to illustrate different approaches to the problem of reading from a file. They will differ in their performance characteristics and their portability. The chapter uses this exercise to introduce the UNIX concept of time, and the first of several important databases provided by the kernel, as well as the kernel's interface to those databases. We also write two different versions of a simplified `cp` command, one using `read()` and `write()`, and the other using memory-mapped I/O.

## 2.2 Commands Are (Usually) Programs

In UNIX, most commands are programs, almost always written in C. Some commands are not programs; they are built into the shell and therefore are called *shell builtins*. Exactly which commands are builtins varies from one shell to another[1], but there are some that are common to almost all shells, such as `cd` and `exit`. When you type `cd`, for example, the shell does not run the `cd` program; it jumps to the internal code that implements the `cd` command itself. You can think of the shell as containing a C `switch` statement inside a loop. When it sees that the command is a built in, it jumps to the code to execute it. Some commands, such as `pwd`, are both shell builtins and programs. By default the shell built in will be executed if the user types `pwd`; to get the program version, one can either precede the command with a backslash "\", as in `\pwd`, or type the full path name, `/bin/pwd`.

[1] The list of built-in commands is usually provided in the shell's man page. For example, the command `man builtins` will display the `bash_builtins` man page, and at the very top of that page is the complete list of bash builtins.

Command programs are located in one of several directories, the most common being `/bin`, `/usr/bin`, and `/usr/local/bin`. The `/usr/local/bin` directory is traditionally used as a repository for commands that do not come with the UNIX distribution and have been added as local extras. Many packages that are installed after the operating system installation are placed in subdirectories of `/usr/local`. Administrative commands, such as those for creating and modifying user accounts, are found in `/usr/sbin`. Many UNIX systems still retain the old `/usr/ucb` directory. (The "ucb" in `/usr/ucb` stands for the University of California at Berkeley. The `/usr/ucb` directory, if it exists, contains commands that are part of the BSD distributions. Some of the commands in `/usr/ucb` are also in `/usr/bin` and have different semantics. If the same command exists in both `/usr/bin` and in some other directory such as `/usr/ucb`, the `PATH` environment variable just like the one used in Windows and DOS, determines which command will be run. The `PATH` variable contains a list of the directories to search when the command is typed without a leading path. Whichever directory is earliest in the list is the one whose version of the command is used. Thus, if `more` exists in both `/usr/ucb` and `/usr/bin`, as well as in your working directory, and `/usr/bin` precedes `/usr/ucb` which precedes "`.`" in your `PATH` variable, and if you type

```
$ more myfile
```

then `/usr/bin/more` will run. If instead you type

```
$ ./more myfile
```

then your `PATH` is not searched and your private `more` program will run. If you type

```
$ /usr/ucb/more myfile
```

then your `PATH` is not searched and `/usr/ucb/more` will run.

## 2.3 The `who` Command

There are a few different commands for checking which users are currently using the system. The simplest of these is conveniently named `who`[2]. Other commands that perform similar tasks are `w`, `users`, and `whodo`[3]. The `who` and `w` commands are required by the POSIX standard, so they are more likely to be on a UNIX installation.

The `who` command displays information about who is currently using the system. Running `who` without command-line options produces a listing such as

```
dsutton    pts/1    Jul 23 20:22   (66-108-62-189.nyc.rr.com)
ioannis    pts/2    Jul 24 16:53   (freshwin.geo.hunter.cuny.edu)
dplumer    pts/3    Jul 26 11:34   (66-65-53-41.nyc.rr.com)
rnoorzad   pts/4    Jul 23 09:25   (death-valley.geo.hunter.cuny.edu)
rnoorzad   pts/5    Jul 23 09:25   (death-valley.geo.hunter.cuny.edu)
sweiss     pts/6    Jul 26 13:08   (70.ny325.east.verizon.net)
```

[2]This is unusual. Most UNIX commands have names that are so cryptic that you have to be a wizard to guess their names. Would you have guessed, for example, that to view the contents of a directory, you have to type "`ls`" or that to view the contents of a file you can type "`cat`"?

[3]`whodo` is not available in Linux. It is found in Solaris, AIX, and other UNIX variants.

Each line represents a single login session. The -H option will print column headings, in case the data is not obvious. The first column is the username, the second is the terminal line on which the user is logged in, the third is the time of the login on that terminal, and the last is the source of the login, either the host name or an X display. For example, sweiss was logged in on terminal line pts/6, the session started at 13:08 on July 26th of the then current year, and the login was initiated from a computer identified as 70.ny325.east.verizon.net. Notice that there may be multiple logins with the same username.

The output of who may vary from one system to another. Some of the reasons have to do with how systems treat users who have multiple terminal windows open in a single login or are running terminal multiplexers such as Gnu's screen program. The w command, by the way, is approximately equivalent to the command sequence "uptime; who"; it shows more information than who does.

## 2.4 Researching Commands In UNIX

UNIX is a self-documented operating system. You can use UNIX itself to learn how it works if you do a thorough exploration of the online documentation. In particular, the man pages can be a source of information about how a command might be implemented. This information is not explicit, but can be obtained by using clues within the page. The man page for a command may not have enough content, and will instead have a message such as the following in the SEE ALSO section at the bottom:

```
The full documentation for who is maintained as a Texinfo manual.
If the info and who programs are properly installed at your site,
the command
    info coreutils 'who invocation'
should give you access to the complete manual.
```

In this case, one should use the info command instead. The info command brings up the *Texinfo* pages. The *Texinfo* system is an alternative system for providing on-line documentation. To learn how to use the Texinfo viewer, type

```
info info
```

which will bring up a tutorial on using the Texinfo documentation system. The general idea is that the information is stored in a tree-like structure, in which an internal node represents a topic area, and its child nodes are specific to that topic. The space bar will advance within the entire tree using breadth-first search. To descend into a node's children, d (for down) works. To go back up, u (for up) works. To traverse the siblings from left to right, n (for next) does the trick, and to go back, p (for previous) works. Just picture the tree.

*Note.* On some systems, when you type "info coreutils who" , you will see the page for the whoami command. If you move ahead a few pages, you will find the page for who. On other systems you may have to type "info who" or "info coreutils 'who invocation'" to bring up the proper pages.

The man page for who tells us that the command may be called with zero or more of the command-line options abdHlmpqrstTu. It can also be called as follows:

```
$ who am i
sweiss pts/6 Jul 26 13:08 (70.ny325.east.verizon.net)
```

and, in Linux, if you supply any two words after "`who`", it behaves the same way:

```
$ who you think
sweiss pts/6 Jul 26 13:08 (70.ny325.east.verizon.net)
```

In general, the way to research a UNIX command is to use a combination of these methods:

1. Read the relevant man page.
2. Follow the `SEE ALSO` links on the page.
3. Read the Texinfo page if the man page refers to it.
4. Search the manual.
5. Find and read the header (.h) files relevant to the command.

### 2.4.1 Reading Man Pages

There is no standard that defines what must be contained in most man pages; it is implementation-dependent. However, most systems follow a time-honored convention for man pages in general, which is what we describe in these notes. For the purpose of understanding how a command works, the relevant sections of the man page for that command are the `DESCRIPTION`, `SEE ALSO`, and `FILES` sections.

The `DESCRIPTION` section gives the details of how the command is used. For example, reading about `who` in the man page reveals that `who` has an optional file name argument, and that if it is not supplied, `who` reads the file `/var/run/utmp` to get the information about current logins. The optional argument can be `/var/log/wtmp`. We can infer that the file `/var/run/utmp` contains information about who is currently logged in. What about `/var/log/wtmp`? If you were to try typing

```
$ man wtmp
```

you would be pleasantly surprised to discover that, although `wtmp` is not a command, there is a man page that describes it. This is because there is a section of the man pages strictly devoted to the description of system file formats. `/var/log/wtmp` is a system file, as is `/var/run/utmp`, and they are both described on the same man page in section 5 of the manual. There we can learn that `/var/log/wtmp` contains information about who has logged in previously[4].

Before we dig deeper into the man page for the `utmp` and `wtmp` files, you should also know that it is required of all POSIX-compliant UNIX systems that they also contain man pages for all of the header files that might be included by a function in the kernel's API. To put it more precisely, each function in the System Interfaces volume of POSIX.1-2008 specifies the headers that an application must include to use that function, and a POSIX-compliant system must have a man page for each of those headers. They may not be installed on the system you are using, but they are available. They will only be installed if the system administrator installed the application development files.

The man pages for the header files have a fixed format. From the POSIX.1-2008 standard:

[4] If we consult the `who` Texinfo page, we could learn that as well.

NAME

This section gives the name or names of the entry and briefly states its purpose.

SYNOPSIS

This section summarizes the use of the entry being described.

DESCRIPTION

This section describes the functionality of the header.

APPLICATION USAGE

This section is informative. This section gives warnings and advice to application developers about the entry. In the event of conflict between warnings and advice and a normative part of this volume of POSIX.1-2008, the normative material is to be taken as correct.

RATIONALE

This section is informative. This section contains historical information concerning the contents of this volume of POSIX.1-2008 and why features were included or discarded by the standard developers.

FUTURE DIRECTIONS

This section is informative. This section provides comments which should be used as a guide to current thinking; there is not necessarily a commitment to adopt these future directions.

SEE ALSO

This section is informative. This section gives references to related information.

The important sections are `NAME`, `SYNOPSIS`, `DESCRIPTION`, and `SEE ALSO`.

For example

```
$ man stdlib.h
```

will display the man page for the header file `<stdlib.h>`. This is a useful feature. But if you do not know the name of the command that you need, nor the names of any files that might be useful or relevant, then you do not know which man page to read. UNIX systems provide various methods of overcoming this problem.

### 2.4.2 Man Page Searching

The most basic solution, guaranteed to work on all systems, is to use the search feature of the `man` command. To search for all man pages that contain a particular keyword in their one-line summaries in the `NAME` Section, you can type

```
$ man -k keyword
```

This will only work if the `whatis` database has been built when the man pages were installed however, so you are at the mercy of the system administrator[5]. For example, typing

[5] If you are the administrator, issue the command `/usr/sbin/makewhatis` to build the database.

```
$ man -k utmp
```

will list all man pages that contain the string `utmp` in their summaries. The command

```
$ apropos utmp
```

has the exact same meaning: `apropos` is equivalent to "`man -k`". Unfortunately, the implementation of `apropos` varies from system to system. On some systems, such as Fedora 15, the most current stable version, `apropos` has features that allow multiple keyword searches as well as regular expression searches. To search for man pages whose page names and/or `NAME` sections contain all keywords provided, one can use the `-a` option, as in

```
$ apropos -a convert case
toupper      (3)     - convert letter to upper or lower case
FcToLower    (3)     - convert upper case ASCII to lower case
tolower      (3)     - convert letter to upper or lower case
towlower     (3)     - convert a wide character to lowercase
towupper     (3)     - convert a wide character to uppercase
XConvertCase (3)     - convert keysyms
```

The number in parentheses is the section number. Section 3 contains man pages for library functions. Notice that we have output in which the string "case" is a substring of other words. If we wanted to limit it to those descriptions in which "case" is a word on its own, we could use the regular expression matching feature of `apropos`:

```
$ apropos -ar convert '\<case\>'
toupper     (3)      - convert letter to upper or lower case
FcToLower   (3)      - convert upper case ASCII to lower case
tolower     (3)      - convert letter to upper or lower case
```

Unfortunately, this powerful `apropos` is not available on all systems. In particular, it is absent on the RHEL 6 system installed on our server. This version has no options, so one cannot do such searches. In this case, to get the same effect, one can use a simple search and pipe the output through a `grep` filter. If you are not familiar with `grep` or regular expressions, see the Appendix. The equivalent command would be

```
$ apropos convert | grep '\<case\>'
FcToLower   (3)      - convert upper case ASCII to lower case
tolower     (3)      - convert letter to upper or lower case
toupper     (3)      - convert letter to upper or lower case
```

If the output list is still too long to be useful, you can filter it further with another instance of grep:

```
$ apropos convert | grep '\<case\>' | grep '\<ASCII\>'
FcToLower   (3)      - convert upper case ASCII to lower case
```

## 2.5 Digging Deeper into the who Command

The output of the manual search on the `utmp` file will look something like:

```
endutent [getutent]  (3)  - access utmp file entries
getutent             (3)  - access utmp file entries
getutid [getutent]   (3)  - access utmp file entries
getutline [getutent] (3)  - access utmp file entries
login                (3)  - write utmp and wtmp entries
logout [login]       (3)  - write utmp and wtmp entries
pututline [getutent] (3)  - access utmp file entries
sessreg              (1x) - manage utmp/wtmp entries for non-init clients
setutent [getutent]  (3)  - access utmp file entries
utmp                 (5)  - login records
utmpname [getutent]  (3)  - access utmp file entries
utmpx.h [utmpx]      (0p) - user accounting database definitions
wtmp [utmp]          (5)  - login records
```

The first word is the topic of the man page, the next, the man page title, the third is the section number of the manual, and the last is a brief description of the topic.

Every UNIX system has a manual volume that deals with the files used by the commands. The number may vary. From the above output, it appears that the `utmp` file is described in Section 5 of the man pages:

```
utmp [utmp] (5) - login records
```

Also, the line

```
wtmp [utmp] (5) - login records
```

shows that the man page describing the `wtmp` file is the same page as the one describing `utmp`. Obviously, there is a man page for `utmp` in Section 5 of the manual. To specify the specific section to display, you need to specify it as an option. The syntax varies; in RedHat Linux either of these will work:

```
$ man 5 utmp
$ man -S5 utmp
```

There was also a line of output

```
utmpx.h [utmpx] (0p) - user accounting database definitions
```

The `<utmpx.h>` header file describes a POSIX-compliant interface to the `utmp` file. This interface is different from that of the `<utmp.h>` file. We will use the (outdated) `<utmp.h>` interface for our initial attempts, exploring the `utmp` file in greater depth, starting with the man page that our system delivers when we type either of the above man commands. After that we will consider using two other interfaces, the POSIX `utmpx` interface and a GNU extension, the thread-safe functions `getutent_r()` and its cousins.

The beginning of the man page for `utmp` from RedHat Enterprise Linux Release 4 is displayed below.

```
NAME
      utmp, wtmp - login records
SYNOPSIS
      #include <utmp.h>
DESCRIPTION
      The utmp file allows one to discover information about who is currently
      using the system. There may be more users currently using the system,
      because not all programs use utmp logging.

      Warning: utmp must not be writable, because many system programs
      (foolishly) depend on its integrity. You risk faked system logfiles and
      modifications of system files if you leave utmp writable to any user.
      The file is a sequence of entries with the following structure declared
      in the include file (note that this is only one of several definitions
      around;  details depend on the version of libc):
```

*( lines omitted here )*

First note that it tells us which header file is relevant: `<utmp.h>` This is the header file that the compiler will use when the include directive `#include <utmp.h>` is in your program[6]. Next, it issues a warning to system administrators not to leave this file writable by anyone other than its owner, the superuser. Then it warns the rest of us, before showing us the contents of the include file, that the contents may differ from one installation to another.

Since UNIX is a free, community supported operating system, it has been evolving over time. You may find that what is described in a book, or in these notes, is different from what you observe on your system. It is not that anything is correct or incorrect, but that UNIX is a moving target, and that systems can differ in minor ways. For example, the man page for `utmp` in an older version of Linux will be very different from the one shown here. Even the location of the `utmp` file itself is different. Later versions of UNIX added system functions to provide a data abstraction layer so that the programmer would not need to know the actual structure of the file. The problem was that different versions of UNIX had different definitions of the utmp structure, and programs that accessed the structure directly were failing on different systems.

[6]There may be many files named `utmp.h` in the file system. Each compiler will have its own method of deciding which one to use. The GNU compiler collection (`gcc`) installs its own header files in specific places, and it uses these by default. The default search path used by `gcc` is typically
`/usr/local/include`
*target-installdir*`/include`
`/usr/include`
where *target-installdir* is the directory in which `gcc` was installed on the machine. This is explained in more detailed shortly.

The structures displayed in the man page may not be the same as those found on our machine. If you write code that depends critically on the structure definition, it may work on one machine but not another. In spite of this, it is valuable to study these structures. Afterward we will write more portable code. The key to that is to use preprocessor directives to conditionally compile the code based on the values of macros. The man page continues:

```
#define UT_UNKNOWN          0
#define RUN_LVL             1
#define BOOT_TIME           2
#define NEW_TIME            3
#define OLD_TIME            4
#define INIT_PROCESS        5
#define LOGIN_PROCESS       6
#define USER_PROCESS        7
#define DEAD_PROCESS        8
#define ACCOUNTING          9
#define UT_LINESIZE         12
#define UT_NAMESIZE         32
#define UT_HOSTSIZE         256
struct exit_status {
    short int e_termination;    /* process termination status.  */
    short int e_exit;           /* process exit status.  */
    };
struct utmp {
    short ut_type;              /* type of login */
    pid_t ut_pid;               /* pid of login process */
    char ut_line[UT_LINESIZE];  /* device name of tty - "/dev/" */
    char ut_id[4];              /* init id or abbrev. ttyname */
    char ut_user[UT_NAMESIZE];  /* user name */
    char ut_host[UT_HOSTSIZE];  /* hostname for remote login */
    struct exit_status ut_exit; /* The exit status of a process
#if __WORDSIZE == 64 && defined __WORDSIZE_COMPAT32
    int32_t ut_session;         /* Session ID (getsid(2)),
                                   used for windowing */
    struct {
        int32_t tv_sec;         /* Seconds */
        int32_t tv_usec;        /* Microseconds */
    } ut_tv;                    /* Time entry was made */
#else
    long   ut_session;          /* Session ID */
    struct timeval ut_tv;       /* Time entry was made */
#endif
    int32_t ut_addr_v6[4];      /* IP address of remote host.  */
    char __unused[20];          /* Reserved for future use.  */
};
```

The page then contains a brief description of the purpose of the structure:

```
This structure gives the name of the special file associated with the user's
terminal, the user's login name, and the time of login in the form of time(2).
String fields are terminated by '\0' if they are shorter than the size of the
field.
```

More information about the specific members of the structure is contained in the comments in the `struct` definition. The man page does not describe the members in detail beyond that. The rest of the man page, which is not included here, goes on to describe how the various entries in the `utmp` file are created and modified by the different processes involved in logging in and out. We will return to that topic shortly. It reiterates the warning:

```
The file format is machine dependent, so it is recommended that it
be processed only on the machine architecture where it was created.
```

You should have noticed the following line in the man page:

```
#if __WORDSIZE == 64 && defined __WORDSIZE_COMPAT32
```

This causes conditional compilation of the code. It means, if the machine's word size is 64 bits and it is in 32-bit compatibility mode, then use one definition of the `ut_session` and `ut_tv` members, otherwise use a different one. The macros `__WORDSIZE` and `__WORDSIZE_COMPAT32` are defined in the header file `/usr/include/bit/wordsize.h`[7]. We will ignore this subtlety for now, and rather than relying on the man page, we will examine the `<utmp.h>` header file itself.

### 2.5.1 Reading the Correct Header Files

Which header file to read depends upon the particular installation. For example, on my home office workstation, which is running Fedora 14, `gcc` will use `/usr/include/utmp.h`, whereas on the `cs82010` server in the Graduate Center, which is running RedHat Enterprise Linux Release 6, `gcc` will first look for `/usr/lib/gcc/x86_64-redhat-linux/4.4.5/include/utmp.h`. One method of determining which file `gcc` will actually use in a particular installation is the following:

1. Create a trivial C program such as

   ```
   int main() { return 0; }
   ```

   and suppose it is named `empty.c`.

   ```
   echo "int main() {return 0;}" > empty.c
   ```

   is an easy way to do this.

2. Run the command

[7] The macro `__WORDSIZE_COMPAT32` is only defined on 64 bit machines. One can discover this file by doing a recursive grep on the `/usr/include` directory hierarchy of the form "`grep -R WORDSIZE /usr/include/* | grep define`", which will list the files in which these macros are defined.

```
$ gcc -v empty.c
```

3. In the output produced by `gcc`, look for lines of the form

```
#include "..." search starts here:
#include <...> search starts here:
your_current_working_dir/include
/usr/local/include
/usr/lib/gcc/x86_64-redhat-linux/4.4.5/include
/usr/include
End of search list.
```

These lines will show you which directories and in which order `gcc` searches for included header files. The above output shows that `gcc` will search first in `/usr/include/local`, then in the install directory, and then in `/usr/include`. Since there is no `<utmp.h>` file in the first two directories, it will use `/usr/include/utmp.h`.

Returning to the task at hand, if you look at either of the `<utmp.h>` files mentioned above, you will see that they are mostly wrappers for a file which is in the corresponding bits subdirectory:

```
/usr/include/bits/utmp.h,
```

or

```
/usr/lib/i386-redhat-linux3E/include/bits/utmp.h.
```

Taking the liberty of eliminating the 64-bit conditional macros, and the macro names, the important elements of the header file are as follows:

```
/* The structure describing an entry in the database of
   previous logins. */
struct lastlog
{
     __time_t ll_time;
    char ll_line[UT_LINESIZE];
    char ll_host[UT_HOSTSIZE];
};
/* The structure describing the status of a terminated
   process. This type is used in `struct utmp' below. */
struct exit_status
{
    short int e_termination; /* Process termination status.*/
    short int e_exit;        /* Process exit status.       */
};
/* The structure describing an entry in the user accounting
   database. */
struct utmp
```

```
{
    short int ut_type;        // Type of login.
    pid_t ut_pid;             // Process ID of login process.
    char ut_line[UT_LINESIZE];   // Devicename.
    char ut_id[4];               // Inittab ID.
    char ut_user[UT_NAMESIZE];   // Username.
    char ut_host[UT_HOSTSIZE];   // Hostname for remote login.
    struct exit_status ut_exit;  /* Exit status of a process
                                    marked  as DEAD_PROCESS.*/
    long int ut_session;     // Session ID, used for windowing.
    struct timeval ut_tv;    // Time entry was made.
    int32_t ut_addr_v6[4];   // Internet address of remote host.
    char __unused[20];       // Reserved for future use.
};
```

The point is that login records have ten significant members, and we can write code to extract their data in order to mimic the `who` command. In particular, the `ut_user` char array stores the username, the `ut_line` char array stores the name of the terminal device of the login, `ut_time` stores the login time, and `ut_host` stores the name of the remote host from which the connection was made. Unfortunately, we will not be able to ignore indefinitely the way that time is defined on different architectures, but for the moment, we will continue to ignore it.

### 2.5.2 What Next?

It seems likely that `who` opens the `utmp` file and reads the `utmp` structures from that file in sequence, displaying the appropriate data for each login. We will write use this as the basis for our own implementation of the command.

## 2.6 Writing `who`

The program that implements the `who` command has two key tasks:

- to read the `utmp` structures from a file, and
- to display the information from a single `utmp` structure on the display device in a user-friendly format.

We begin by discussing solutions to the first task.

### 2.6.1 Reading Structures From a File

A *binary file* consists of a sequence of bytes, not to be interpreted as characters. It is the most general form of a file. A file consisting of a sequence of structures, such as the `utmp` file, is a binary file and cannot be read using the C I/O functions with which most programmers are familiar, such as `get()`, `getc()`, `fgets()`, and `scanf()`, nor the `istream` methods in C++, because all of these read textual input. They are specifically designed for that purpose. Although you could read structures by reading one char at a time and then reconstructing the structure from the sequence of chars with a lot of type casts, that would be grossly inefficient and error-prone. Clearly there must be a better way.

Let us suppose that you do not know the methods of reading from a binary file. You could use a man page search such as

```
$ man -k binary file | grep read
```

Remember though that when you use multiple words with the `-k` option, they are OR-ed together, so the output includes lines with either word (or both). If you do this search, you will see a list of perhaps several dozen man pages. If you get a long list you can filter it further by limiting the output to only sections 2 or 3 of the man pages with a third stage in the pipeline:

```
$ man -k binary file | grep read | grep '([23])'
```

In this list will be the page for two prospective functions to use:

```
fread (3)         - binary stream input/output
pread (2)         - read from or write to a file descriptor at a given offset
read (2)          - read from file descriptor
```

The first, `fread()`, in Section 3, is part of the C Standard I/O Library; it is C's function for reading binary files. The second, `pread()`, in Section 2, is the prototype of a system call. The third, `read()`, in Section 2, is also the prototype of a system call. As we are primarily interested in what Unix in particular has to offer us, we will look at the system calls. We begin with `read()` and return to `pread()` later. In Chapters 5 and 7, we will revisit the C Standard I/O Library.

We want to see what the man page for `read()` has to say. If you do not specify the section number when you type "`man read`", you will get the man page from the first section, and you will discover that there is also a UNIX command, `/usr/bin/read`:

```
$ man read
```

which will output the man page for the `read` command in Section 1. You must type

```
$ man 2 read
```

to get the man page for the `read()` system call. I have included the important parts of the man page below.

```
NAME
    read - read from a file descriptor
SYNOPSIS
    #include <unistd.h>
```

```
    ssize_t read(int fildes, void *buf, size_t nbyte);
DESCRIPTION
read() attempts to read up to count bytes from file descriptor fd into
the buffer starting at buf.
If count is zero, read() returns zero and has no  other  results.
If count is greater than SSIZE_MAX, the result is unspecified.
RETURN VALUE
On  success, the number of bytes read is returned (zero indicates end
of file),  and the file position is advanced by this number.  It is not
an error if this number is smaller than the number of bytes requested;
this may happen for example because fewer bytes are actually available
right  now  (maybe because we were close to end-of-file, or because we
are reading from a pipe, or from a terminal), or because  read()  was
interrupted  by  a signal.  On error, -1 is returned, and errno is set
appropriately. In this case it is left unspecified  whether  the
file position (if any) changes.
```

To use the `read()` function, the program must include the header file `<unistd.h>`. This header file serves various purposes, the most relevant for our purposes being that it contains the prototypes of the (POSIX compliant) system calls.

**The difference between <stdio.h> and <unistd.h>.**

> The functions that begin with "`f`": `fopen()`, `fread()`, `fwrite()`, `fclose()`, and so on, which operate on file stream pointers (`FILE` pointers) are all part of the ANSI Standard C I/O Library, whose header file is `<stdio.h>`. They are C functions that you can use on any operating system. We used `fopen()` and `fclose()` in Chapter 1 to implement our version of the `more` command.
>
> The functions `open()`, `read()`, `write()`, and `close()` are UNIX system calls and their prototypes are defined in `<unistd.h>`, which is a POSIX header file. The `<unistd.h>` header defines miscellaneous symbolic constants and types, and declares miscellaneous functions, among which are these calls. These functions exist only in UNIX systems and they exist no matter what language you use, as long as the system you are using is POSIX-compliant. POSIX does not specify whether they should be system calls or library functions, but only that they exist as one or the other. These system calls operate on file descriptors, not file streams. The UNIX system calls operate on the kernel directly; the ANSI Standard C I/O Library calls are at a higher level.

The `read()` function has three arguments. The man page says that the `read()` function reads from a file associated with a *file descriptor*. A file descriptor is a small, non-negative integer. We will study file descriptors in greater detail in a later chapter. The second parameter is a pointer to a place in memory into which the bytes that are read are to be stored. The third parameter is the number of bytes to read. The return value is the number of bytes actually read, which can never be larger, but might be smaller, or is –1, if something went wrong.

To illustrate, suppose that `filedesc` is a valid file descriptor that we can use for reading, `buffer` is a char array of size 100, and `num_bytes_read` is an integer variable. The following code fragment shows how to read 100 bytes of data at a time from this file stream until the end of data is found

```
while ( !done ) {
    num_bytes_read = read(filedesc, buffer, 100);
    if ( 0   > num_bytes_read )
        // an error code was returned during reading - bail out
    if ( 0  == num_bytes_read )
        // the end of file was reached - stop reading
        done = 1;
    else
        // do whatever has to be done to the data
}
```

This is a typical read-loop structure. The `read()` call does not fail when there is no data; it just returns 0. This is how to detect the end of the input data.

How can a program associate a file descriptor with a file? Look in the `SEE ALSO` section of the man page and you will find references to `fnctl()`, `creat()`, `open()`[8] and many other system calls. Most of these work with file descriptors. The `open()` system call is the one we need now, because the `open()` call opens a file and assigns a file descriptor to it.

### 2.6.2 The open() and close() System Calls

To read from a binary file, a process must

- open the file for reading,
- read the bytes, and
- close the file.

The `open()` system call creates a connection between the process and the file. Think of a connection as an object that manages the I/O operations on the file from the process. This object contains things such as the offset in the file for the next operation, various status flags, and pointers to kernel functions that the process can invoke. It is represented by a file descriptor. A process can open several files and each will have its own file descriptor. In fact, it can open the same file twice and each connection will have a different file descriptor[9]. UNIX does not prevent you or anyone else from opening the same file many times. It is up to the users and their programs to coordinate accesses to files.

If you look at the man page you will see the following synopsis of the `open()` call.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int oflag, /* mode_t mode */...);
```

[8] All of these are in Section 2 of the man pages.

[9] You might have guessed. The file descriptor is the index into an array of structs. Each of these structs contains, among other things, a pointer to the next character in the file to be read. A process can read from two different parts of the same file at the same time in this way.

The first argument is a character string containing the path to the file to be opened. The second argument is an integer specifying how the file is to be opened: for reading, for writing, for reading and writing, for appending, and so on. If the call is successful, it returns a file descriptor. More accurately, it returns the lowest numbered file descriptor not already in use by the process. If the call is not successful, it returns –1. There are methods of detecting the type of error; these will be examined later.

The value of `oflag` is one of the following constants defined in <`fcntl.h`>:

`O_RDONLY` Open for reading only.

`O_WRONLY` Open for writing only.

`O_RDWR` Open for reading and writing.

It is more complex than this, but this is enough for now. Other values can be bit-wise-OR-ed to these values.

**Example.** Consider the following code:

```
int fd;
if (fd = open("/var/adm/messages.0", O_RDONLY) < 0 )
    exit (-1);
```

This attempts to open the file `/var/adm/messages.0` for reading. If it fails, it exits. If it is successful, the file is ready for reading. The file descriptor stored in `fd` is the one the program must use in the `read()` call. Notice that the call is made within a conditional expression and that the return value of the call is compared to 0 in that condition. This is a common method of error handling in C programs.

Unlike other operating systems, UNIX does not prevent a file that is already open by one process from being opened by another. This is a very important feature to remember about UNIX. It is why it is possible for multiple users to run the same command or change their passwords at the same time[10].

After your process is finished reading a file, it should close the connection to the file. The `close()` system call

```
int close( int filedes)
```

has a single argument which is the file descriptor of the connection to be closed. If a file has been opened by a process via multiple calls to `open()`, then the other connections will remain open and only the one corresponding to `filedes` will be closed. If the kernel cannot close the connection, it will return –1.

Now you might wonder what could possibly go wrong when closing a file, especially when it has been opened for reading. Well, first of all, it is possible you passed it a bad file descriptor when

[10] Of course UNIX does provide the means for a process to open a file and lock it so that no other process can read or write it while it is in use, but this requires actions on the part of the process to make it happen. UNIX does not do this automatically.

you closed it. Secondly, the kernel, in the middle of the system call, may be given an urgent task to complete, so urgent that it has to drop the `close()` call in the middle to deal with it. In this case it will also return a –1. Also, the file may not have been on the local machine or the local drive, and a network connection might have gone down, in which case the file cannot be closed. Furthermore, if this file had been opened for writing, there are more reasons why `close()` might fail, the most important of which is that it is only when `close()` is called that the actual write takes place and at which point the kernel will discover it cannot complete the write for any number of reasons.

### 2.6.3 A First Attempt at Writing `who`

The main program must open the file and then enter a loop in which it repeatedly reads a single `utmp` record and displays it on the screen, until all records have been read. A rough sketch of this is in the listing below, which we call `who1.c`.

```c
Listing 1. who1.c
#include        <stdio.h>
#include    <stdlib.h>
#include        <fcntl.h>
#include        <utmp.h>

int main()
{
        int         fd;
        struct utmp current_record;
        int         reclen = sizeof(struct utmp);

        fd = open(UTMP_FILE, O_RDONLY);
        if ( fd == -1 ) {
            perror( UTMP_FILE );
            exit(1);
        }

        while ( read(fd, &current_record, reclen) == reclen )
            show_info( &current_record );

        close (fd);
        return 0;
}
```

First observe that the first argument to the `open()` call is `UTMP_FILE`. This is a macro whose definition is included in the `<utmp.h>` header file. Its value is system-dependent; it is the path to the actual `utmp` file. It is usually `"/var/run/utmp"`. We would not know about it if we did not read the header file.

Notice which header files are included, notice that `reclen` contains the number of bytes in a `utmp` struct. The `sizeof()` function returns the number of bytes in its argument type. `reclen` will be used in the `read()` call to read exactly one `utmp` structure at a time. The call to `read()` is given the file descriptor returned by `open()`, a pointer to a memory location large enough to hold one `utmp` record, and `reclen`, the number of bytes to be read. If the return value equals `reclen` then a full record was read. If it does not, then an incomplete record was read or the end-of-file was reached. In either case we stop reading. The `show_info()` function remains to be written. It should display the contents of the current record. The `perror()` function is described below.

### 2.6.4 What to Do with System Call Errors

In UNIX, most system calls simply return the value –1 when something goes wrong. This would be rather useless if that is all it did because the calling program would not know what actually went wrong. In addition to returning a –1, the kernel stores an error code in the global variable `errno` that all processes can access if they include `<errno.h>`. When you build a program in UNIX, the variable `errno` is in the namespace of the program if the header file is included.

The `<errno.h>` file defines a number of mnemonic constants for error values, such as

```
#define EPERM 1 /* Operation not permitted */
#define ENOENT 2 /* No such file or directory */
```

Your program can use these symbols directly with code such as

```
if ( fd = open("myfile", O_RDONLY) == -1 ) {
    printf(Cannot open file: ");
    if ( errno == ENOENT )
        printf("No such file or directory\n");
    else if
        ...
}
```

This would be very tedious, since every program you write would have long `switch` statements or cascading if-statements. It is much easier to use the UNIX library function `perror()` to do this for you. The `perror()` function, which conforms to POSIX-1.2001, has a single string as a parameter, and looks up the value of `errno` and displays the string followed by an appropriate message based on the value of `errno`. It is declared in `<stdio.h>`, so you do not need to include `<errno.h>` if you use it. The code snippet above is simplified by using `perror()`:

```
if ( fd = open("myfile", O_RDONLY) == -1 ) {
    perror("Cannot open file: ");
    return;
}
```

and it would print

```
Cannot open file: No such file or directory
```

In short, the perror() function prints the string you pass it followed by the message from the <errno.h> file. It is a good idea to create a function to handle errors, so that you do not have to type these lines all of the time. Very often, the error is a fatal one, meaning that the program cannot proceed if the error occurred. In this case, you would want to exit the program, calling exit() to do so, as in

```
if ( fd = open("myfile", O_RDONLY) == -1 ) {
    perror("Cannot open file: ");
    exit(1);
}
```

The exit() function is declared in <stdlib.h>; its man page is in Section 3. A simple function for handling fatal errors would be

```
#include <stdio.h>
#include <stdlib.h>

void fatal_error(char *string1, char *string2)
{
    fprintf(stderr,"Error: %s ", string1);
    perror(string2);
    exit(1);
}
```

You might also benefit from writing a second function to call when you do not want to terminate the program, or you could combine the two into a single, general-purpose function that does either, by passing a parameter to indicate the error's severity.

### 2.6.5 Displaying login Records

This is the first attempt at show_info():

```
void show_info( struct utmp *utbufp )
{
    printf("%-8.8s", utbufp->ut_name); /* the logname */
    printf(" ");
    printf("%-8.8s", utbufp->ut_line); /* the tty     */
    printf(" ");
    printf("%10ld",  utbufp->ut_time); /* login time  */
    printf(" ");
    printf("(%s)",   utbufp->ut_host); /* the host    */
    printf("\n");                      /* newline     */
}
```

If this were compiled and run on a system that supported this API, the output would look something like

```
$ who1
             system b    952601411    ()
                         952601423    ()
LOGIN        console     952601566    ()
acotton      ttyp3       964319088    (math-guest04.williams.edu)
             ttypc       964319645    ()
```

This output differs from the output of `who` in two significant ways. First, there are records in the output of `who1` that do not correspond to user logins, and second, the login times are in some strange format. Both of these problems are easily fixed.

### 2.6.6 A Second Attempt at Writing who

#### 2.6.6.1 Suppressing Records That Are Not Active Logins

The file `/usr/include/utmp.h` contains definitions of integer constants used for the `ut_type` member. They are

```
#define EMPTY         0
#define RUN_LVL       1
#define BOOT_TIME     2
#define OLD_TIME      3
#define NEW_TIME      4
#define INIT_PROCESS  5   /* Process spawned by "init" */
#define LOGIN_PROCESS 6   /* A "getty" process waiting for login */
#define USER_PROCESS  7   /* A user process */
#define DEAD_PROCESS  8
```

New entries in the `utmp` file are created by the `init` process and are initialized with a `ut_type` of `INIT_PROCESS`. Recall from Chapter 1 that what happens when a user logs in depends upon whether it is a console login, a login on an xterm window, or a login over a network using a protocol such as SSH. In all cases, the `ut_type` of the entry is changed from `INIT_PROCESS` to `LOGIN_PROCESS`, either by a `getty` process or a similar process, depending on the source of the login. The `getty` (or similar) process prints the login prompt, collects the user's input to the prompt (which should be a username) and creates a login process, handing the user's username to the login process. The login process prompts for the password and authenticates it. If it is valid, it changes the `ut_type` to `USER_PROCESS`. When a user logs out, the `ut_type` is changed to `DEAD_PROCESS`.

This implies that the `ut_type` member of a currently logged-in user record will have the value `USER_PROCESS`. No other `utmp` record will be of type `USER_PROCESS` and so all we need to do to suppress non-user records is to print only those records whose `ut_type` member is `USER_PROCESS`. The show_info() function will be modified by the inclusion of this check:

```
show_info( struct utmp *utbufp)
{
    if ( utbufp->ut_type != USER_PROCESS )
        return;
    ...
}
```

This solves the first problem.

#### 2.6.6.2 Displaying Login Time in Human-Readable Form

Solving the second problem requires an understanding of how calendar, or universal, time is represented in UNIX systems and what functions are provided in the API for manipulating time values.

UNIX represents time as the number of seconds elapsed since 12:00 A.M., January 1, 1970, *Coordinated Universal Time* (*UTC*)[11], known as the "Epoch". UTC is essentially like Greenwich Meridian Time except that it includes occasional "leap seconds" to synchronize with the earth's rotation[12]. UNIX stores time in objects of type `time_t`, the implementation of which is not standardized. On many systems `time_t` is a typedef for a 32-bit integer. Such implementations will fail in the year 2038, when it overflows. Representing time as an integer number of seconds since the Epoch makes it easy for the kernel to update times, but not very easy for a human to determine the time.

How can we learn more about UNIX time and the various parts of the API related to it? The answer again is to do a man page search. If you search on the keyword "time" you will find too many man pages that refer to time. A second keyword will be needed to refine the search. Perhaps "convert" or "transform" or something similar, to capture functions that transform time from one form to another. Trying

```
$ man -k time | grep transform
```

we will see several functions related to time, including `ctime()` and `localtime()`. The man page will also include reference to the header file, `<time.h>`, which must be included for most of these functions. These functions share a single man page. Reading this page reveals that `ctime()` converts a `time_t` time into a human readable string of the form

```
"Mon Aug 11 23:12:06 2003\n"
```

To be precise, the `ctime()` function is declared as

```
char *ctime(const time_t *clock);
```

Observe that the argument is the address of a `time_t` value, not the value itself. The return value is a pointer to a string consisting of a 3-letter day abbreviation, a 3-letter month abbreviation, the day of the month, the 24-hour time in hours, minutes, and seconds, and the 4-digit year. The string is allocated statically by `ctime()`, so it might be overwritten by other calls, so it is best to copy it into a local variable if it needs to be available at a later time.

*Note* 1. `ctime()` is one of many functions that return a pointer to a string that is allocated statically. Make sure that you understand what this means. The string itself is allocated by `ctime()` and a pointer to that memory is returned to the caller. Subsequent calls to `ctime()` will overwrite the previously allocated memory. The caller will be unable to retrieve the old value unless it was copied

[11] The abbreviation UTC is a compromise between the English and French abbreviations. In English, it would be CUT and in French, TUC.

[12] The earth's rotation can vary due to astronomical conditions. UNIX systems are not required by POSIX to represent exact UTC; they are allowed to ignore the leap seconds.

to a local. Also, the caller is not responsible for freeing the memory allocated to the string; that is handled by the library. This is just one of many functions that are not *thread-safe*, a topic we discuss below.

The `localtime()` function takes a `time_t` argument but returns a pointer to a `struct tm`, which is a structure whose members are the various components of time, such as the day-of-week, the month, day, and year, and so on.

If you read through the man page carefully, which you should, you will find near the end the conformance section. It states:

```
CONFORMING TO

POSIX.1-2001.  C89 and C99 specify asctime(), ctime(), gmtime(), localtime(),
and mktime().  POSIX.1-2008 marks asctime(), asctime_r(), ctime(), and ctime_r()
as obsolete, recommending the use of strftime(3) instead.
```

The `ctime()` function is disparaged at this point. One should instead use `strftime()`, whose prototype is

```
#include <time.h>
size_t strftime(char *s, size_t max, const char *format, const struct tm *tm);
```

This function, unlike `ctime()`, allows the calling program to specify the format of the character string to be created. It is also safer to use in that the string is passed as an argument to the function, allocated by the caller, instead of allocated statically and returned as the function value. The first argument is a pointer to the string to be filled, the second, the size of the array of chars to fill, the third is a format for the string, and the last is the `tm` structure containing the broken down time representation.

The format specification is described in great detail in the man page for the function. It is similar to the format for the `printf()` function in that it is a string literal enclosed in double-quotes, with conversion specifications of the form `%`*x*, where *x* is a character to be replaced. For example, `%M` represents minutes as a decimal number in the range 00 to 59. and `%b` is the abbreviation of the month name *in the current locale*. This phrase, "in the current locale" means that the locale settings of the user are used in deciding the exact string that `%b` will produce. Every user has a locale in UNIX. The topic of locales will be covered in a later section. The important point now is that `strftime()`, unlike `ctime()`, can use locale information in determining the format of the output string. In chapter 3 we will use this function to display time with more control. For our implementation of the who command, we will use `ctime()`.

The `who` program only displays the date, hours and minutes. For the above example, it would display only `"Aug 11 23:12"`. Our implementation of `who` must extract this substring from the larger string. In other words, given

```
"Mon Aug 11 23:12:06 2003\n"
```

it needs to print

```
"Aug 11 23:12"
```

A simple way to achieve this, perhaps not obvious, is to use pointer arithmetic to print only those characters of the source string in which we are interested. The first character is 4 characters after the start of the string, and the length of the string is exactly 12 characters. Assuming that `t` is a `time_t` variable containing the required time to be printed, the following `printf()`[13] call will do the trick:

```
printf("%12.12s", ctime(&t) + 4 );
```

which prints the 12 chars starting at position 4 in the full string. The format "`%12.12s`" forces the string to use 12 characters on the output. The complete program is shown below. You should study it carefully.

```
Listing who2.c
//     This solves the time display problem and it filters records

#include        <stdio.h>
#include        <stdlib.h>
#include        <unistd.h>
#include        <utmp.h>
#include        <fcntl.h>
#include        <time.h>

void show_time(long);
void show_info(struct utmp *);

int main(int argc, char* argv[])
{
    struct utmp     utbuf;       // read info into here
    int             utmpfd;      // read from this descriptor
    int             reclen = sizeof(utbuf);

    if ( (utmpfd = open(UTMP_FILE, O_RDONLY)) == -1 ){
        perror(UTMP_FILE);
        exit(1);
    }

    while( read(utmpfd, &utbuf, reclen) == reclen )
        show_info( &utbuf );
    close(utmpfd);
    return 0;
}

```

[13]If you are not familiar with the following C functions, you should take the time to familiarize yourself with them: `printf`, `fprintf`, `sprintf`, `scanf`, `fscanf`, and `sscanf`. These are all part of C and hence C++ and any C or C++ book should contain adequate descriptions of them. You can also look at the manpages for them. Once you know `printf` and `scanf`, the others are trivial to understand. The best way to learn them is to write a few very simple programs of course.

```
void show_info( struct utmp *utbufp )
//  displays the contents of the utmp struct only if a user
//  login, with time in human readable form, and host if
//  not null
{
    if ( utbufp->ut_type != USER_PROCESS )
        return;

    printf("%-8.8s", utbufp->ut_name); /* the logname */
    printf(" ");
    printf("%-8.8s", utbufp->ut_line); /* the tty     */
    printf(" ");
    show_time( utbufp->ut_time );      /* login time  */
    printf(" ");
    if ( utbufp->ut_host[0] != '\0' )  /* the host    */
        printf(" (%s)", utbufp->ut_host);
    printf("\n");

}

void show_time( long timeval )
//  displays time in a format fit for human consumption
//  uses ctime to build a string then picks parts out of it
//  Note: %12.12s prints a string 12 chars wide and LIMITS
//  it to 12 chars.
{
    char* timestr = ctime(&timeval);
    // string looks like "Sat Sep  3 16:43:29 EDT 2011"

    // print 12 chars starting at char 4
    printf("%12.12s", timestr + 4 );
}
```

### 2.6.7 A Third Version of who

The preceding versions of `who` read the data from the `utmp` file using the `read()` system call, reading one `utmp` struct at a time. An alternative method of accessing the data in the file is to take advantage of a data abstraction layer that the API makes available. When we did the man page search for man pages related to the `utmp` file, we ignored the functions found on the page named `getutent`:

```
endutent [getutent]  (3)  - access utmp file entries
getutent             (3)  - access utmp file entries
getutid [getutent]   (3)  - access utmp file entries
getutline [getutent] (3)  - access utmp file entries
pututline [getutent] (3)  - access utmp file entries
setutent [getutent]  (3)  - access utmp file entries
utmpname [getutent]  (3)  - access utmp file entries
```

We now take a look at what that page has to offer. The beginning of the page contains the following (depending on what system you have):

```
SYNOPSIS
       #include <utmp.h>
       struct utmp *getutent(void);
       struct utmp *getutid(struct utmp *ut);
       struct utmp *getutline(struct utmp *ut);
       struct utmp *pututline(struct utmp *ut);
       void setutent(void);
       void endutent(void);
       int utmpname(const char *file);
DESCRIPTION
       New applications should use the POSIX.1-specified "utmpx"
       versions of these functions; see CONFORMING TO.
```

The very first sentence in this man page tells us that these functions are not POSIX.1-compliant, and that there are `utmpx` versions of these functions. We will ignore this warning for the moment and see how to use these non-POSIX functions, simply because there is something that needs to be explained about the POSIX.-1-compliant interface, to which we will return afterward.

The man page basically tells us that there is a simple way of reading the records in a `utmp` file, requiring just four steps:

1. Use `utmpname()` to select the file that should be accessed by the other functions.
2. Call `setutent()` to rewind the file pointer to the beginning of the file.
3. Repeatedly call `getutent()` to get the next `utmp` record from the file; `getutent()` will return a `NULL` pointer after it has read the last record from the file.
4. Call `endutent()` when we have read all of the records.

In other words, this interface provides a hidden iterator to the `utmp` file: `setutent()` initializes it, `getutent()` advances it successively, and `endutent()` sends a signal that it is no longer needed. In addition, the `utmpname()` function simply needs to be told the pathname to the file, and it will take care of opening it.

The man page also mentions that `_PATH_UTMP` is a macro whose value is the path to the `utmp` file. We already knew that `UTMP_FILE` contained that path, but if we dig a little deeper by actually reading the header files, we will discover that the `<paths.h>` header file defines `_PATH_UTMP` and `_PATH_WTMP` and that `<utmp.h>` defines `UTMP_FILE` as another name for `_PATH_UTMP`.

We can put all of this together to create a simpler version of `who`, named `who3`. In this version we add the extra feature that the user can optionally supply the word "`wtmp`" on the command line if she wants to see records in the `wtmp` file instead. The `show_info()` and `show_time()` functions are the same, so we just display the main program in the listing.

```
Listing who3.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <utmp.h>
#include <fcntl.h>
#include <time.h>

int main(int argc, char* argv[])
{
    struct utmp *utbufp;

    if ( (argc > 1) && (strcmp(argv[1],"wtmp") == 0) )
        utmpname(_PATH_WTMP);
    else
        utmpname(_PATH_UTMP);

    setutent();
    while( (utbufp = getutent()) != NULL )
        show_info( utbufp );
    endutent();
    return 0;
}
```

This program is not *thread-safe*. Many functions in the various UNIX libraries use static variables to store their results. These variables act like global variables within the programs that call these functions. If a program is multi-threaded, these threads can corrupt each others data if they use the unsafe function calls in an overlapping way. Thread-safe functions do not have this problem. A thread-safe version of the `who3` program can use `getutent_r()`, which is a GNU thread-safe version of `getutent()`.

The man page tells us that to use the `getutent_r()` function, we have to set a macro, the `_GNU_SOURCE` macro, before including the header file `<utmp.h>`. That is the purpose of the following lines from that man page:

```
The above functions are not thread-safe. Glibc adds reentrant versions
#define _GNU_SOURCE     /* or _SVID_SOURCE or _BSD_SOURCE */
#include <utmp.h>
int getutent_r(struct utmp *ubuf, struct utmp **ubufp);
```

The macro definition of `_GNU_SOURCE` is required because the `<utmp.h>` header file contains *feature test macros*. Feature test macros can be used to control which definitions are exposed in the system header files when a program is compiled. This is important for creating portable applications, because it prevents nonstandard definitions from being exposed in the program. If you remove the definition of `_GNU_SOURCE` from your program and try to use `getutent_r()` you will get a compile time error because the declaration of this function in the header file is guarded by a conditional preprocessor directive that is true only if `_GNU_SOURCE` is defined. It is essentially of the form

```
#ifdef  _GNU_SOURCE
    extern int getutent_r (struct utmp *__buffer, struct utmp **__result) __THROW;
    /* more stuff here
#endif
```

If you put the definition of `_GNU_SOURCE` after the include directive, it will be useless because it will not be defined when the header file is preprocessed by `gcc`, and so in this case too you will get an error message.

The `feature_test_macros` man page describes everything you need to know to use these macros.

The main program of this thread-safe `who`, which we call `who4.c`, is almost the same as that of `who3.c`:

```
Listing who4.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define _GNU_SOURCE
#include <utmp.h>
#include <fcntl.h>
#include <time.h>

int main(int argc, char* argv[])
{
    struct utmp  utbuf, *utbufp;
    int          utmpfd;

    if ( (argc > 1) && (strcmp(argv[1],"wtmp") == 0) )
        utmpname(_PATH_WTMP);
    else
        utmpname(_PATH_UTMP);

    setutent();
    while( getutent_r(&utbuf, &utbufp) == 0 )
        show_info( &utbuf );
    endutent();
    return 0;
}
```

### 2.6.8 A POSIX-compliant Version

There is yet another version of the `who` program, named `who_p.c`, in the demos directory for this chapter on the server. This version is distinguished by the fact that it uses the POSIX-compliant `utmpx` interface. The `utmp` structure is not standard across all versions of UNIX. The one we described above is the GNU implementation, which is what is found on Linux systems. This GNU version includes members that may not be present on other systems. In an effort to standardize the `utmp` interface, the POSIX standards since 2001 have replaced the definition of the `utmp` structure with a `utmpx` structure. This structure is only guaranteed to have the following members:

```
char            ut_user[]  User login name.
char            ut_id[]    Unspecified initialization process identifier.
char            ut_line[]  Device name.
pid_t           ut_pid     Process ID.
short           ut_type    Type of entry.
struct timeval  ut_tv      Time entry was made.
```

In addition, the functions `setutent()`, `getutent()`, and `endutent()` are replaced by the corresponding functions `setutxent()`, `getutxent()`, and `endutxent()`. In general, the `utmpx` structure may define a different set of members than those found in a `utmp` structure. Linux systems actually define the `utmpx` structure to be the same as the `utmp` structure, unless the `_GNU_SOURCE` macro is defined. In addition, Linux systems define a larger set of allowed values of the `ut_type` member than does POSIX. Programs that are meant to be portable can use conditional compilation with feature test macros to detect which structure is actually on the system at compile time. The `who_p.c` program demonstrates how this is done, but is not included in these notes.

### 2.6.9 Summary

The preceding set of implementations of the `who` command demonstrates that the man pages and header files can be used to learn enough about a command to implement it. The `utmp` interface may not be the same on every UNIX system, and as a result there are several different methods of approaching the problem. One can use the GNU, non-POSIX, thread-safe version of the interface, for example, or the POSIX-compliant `utmpx` interface. One can also use the lower-level system calls, e.g. `read()`, to access either the `utmpx` or the `utmp` structure directly. A truly portable solution would use feature test macros to conditionally compile the code depending on what system it is to be run on. The exercise introduced various concepts along the way, but we are still not finished with it. Later we will return to the problem with a more efficient solution.

## 2.7 Using a File in Read/Write Mode

Many applications need to have a file open for both reading and writing. A good example of this is the logout command. The logout command has to update the utmp file, finding within it the record to be updated (i.e., reading it) and then modifying that record (writing it). Most I/O libraries allow a file to be opened for both reading and writing.

### 2.7.1 Opening a File in Read/Write Mode

Recall that the `open()` system call's second parameter is a set of flags stored in an integer, and that the flags must include one of the access mode flags: `O_RDONLY`, `O_WRONLY`, and `O_RDWR`. If the access mode is set to `O_RDWR`, then the file is opened in read/write mode. In read/write mode, the process can read from and write to the file. The file is not truncated as it would be if opened with the `O_CREAT` flag. Instead it is opened with the current position pointer pointing to the start of the file. The current position pointer is a member of the *open file structure*, the data structure that is created by the kernel when a file is opened. It points to the position of the next byte to read or write in the file.

For example, to open the file whose path is stored in the C-string `file_to_open`, one could write

```
if ( ( fd = open(file_to_open, O_RDWR)) == -1 ) {
    perror(file_to_open);
    // handle error here
}
```

### 2.7.2 Logout Records

When a user logs out of a UNIX system, the kernel does some bookkeeping tasks. One of the tasks is to update the `utmp` file to indicate that the user logged out. In particular, it has to change the `utmp` record for the login session by changing the `ut_type` member from `USER_PROCESS` to `DEAD_PROCESS`. It also has to change the `ut_time` member to the current time and zero out the `ut_user` and `ut_host` members.

In short, the logout process has to do the following:

1. Open the `utmp` file for reading and writing
2. Read the `utmp` file until it finds the record for the terminal from which the logout took place.
3. Modify a copy of the `utmp` record in the process's memory, and replace the `utmp` record in the file with the modified one, i.e., modify the `utmp` file.
4. Close the `utmp` file.

The first and last steps need no discussion. The second step requires being able to identify which `utmp` record in the file corresponds to the one logout is trying to modify. It cannot use the `ut_user` member because a single user might have several lines open at a time. The piece of information that is unique is stored in the `ut_line`. The `ut_line` member stores the name of the pseudo-terminal as a string such as `"pts/4"`. Only one person can be using a given terminal at the same time, so it is sufficient to match the line.

The more interesting part of this task is how to replace the `utmp` record in the file. The record may be in the middle of the file, so this operation involves replacing a fixed-size sequence of bytes starting at some specific position in a file with a sequence of the exact same size.

### 2.7.3 Using `lseek` to Move the File Pointer

As noted above, when a file is opened and a file descriptor is returned for it, a data structure is created by the kernel. This data structure represents the connection to the file. The current position pointer of the data structure is the position of the next byte to read or write in the file. If the file is open for reading, a read of $N$ bytes starts at this position, and then the current position pointer is advanced $N$ bytes. If it is open for writing and writes $N$ bytes, it writes starting at the current position and then advances it $N$ bytes. Usually when a file is open for writing the current position pointer is at the end of the file.

The `lseek()` system call changes the current position pointer in an open file.

```
#include <sys/types.h>
#include <unistd.h>
off_t  lseek( int fd, off_t dist, int base)
```

`lseek()` is given a file descriptor, `fd`, a distance in bytes, `dist`, and an integer flag, `base`. `base` can be one of three values. The distance, `dist`, is used by `lseek()` to move the current position pointer. If `dist` is positive, it moves forward; if it is negative, it moves backwards. The value of `base` determines the starting position of the current position pointer from which it is to be moved. The three values are

`SEEK_SET` the distance dist is forwards relative to the start of the file,

`SEEK_CUR` the distance, dist, is relative to the current position pointer and may be positive or negative

`SEEK_END` the distance, dist, is relative to the end of the file and may be positive or negative.

If `lseek()` is successful, its return value is the resulting offset location as measured in bytes from the beginning of the file, otherwise it returns –1.

When the value of the offset is positive and the base is `SEEK_END`, the file pointer is moved beyond the end of the file. Data can be written to this position, and this in effect creates a "hole" in the file. For example, if a file is currently open and has the contents "`123456789`", and a seek is performed that moves the file pointer 5000 bytes past the end, after which the characters "`abcde`" are written to the file, then the file size will be 5014 bytes, even though there is a hole of 5000 bytes within it. More will be said about this in Chapter 3.

The `lseek()` call can be used to code the third step of the logout procedure.

### 2.7.4 Updating the `utmp` File on Logout

The problem with updating the `utmp` file is the following. We have to find the record that corresponds to the login record on the line on which the logout occurred. Therefore we need to repeatedly read a `utmp` record and check whether the `ut_line` member matches the line. When we find the record, which has been read into a local variable in our function, we modify it and then have to write it back. But at this point, the current position pointer has already been advanced to point to the record immediately following the one we just read. Figure 2.1 illustrates this.

In the figure, the matching record is numbered k. After it is found, the pointer has been advanced to the start of record k+1. In order to write the modified record where the original was, we need to move the current position pointer back with `lseek()`. The following program demonstrates the key ideas.

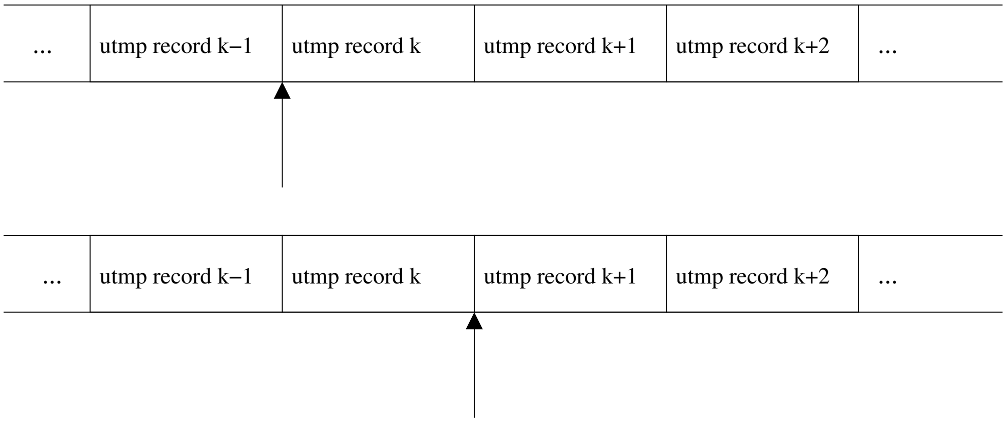


Figure 2.1: Updating a utmp record in read/write mode

```
Listing who5.c
#include ....

int main(int argc, char* argv[])
{
    struct utmp  utbuf;        // stores a single utmp record
    int          fd;           // file descriptor for utmp file
    int          utsize   = sizeof(utbuf);
    int          utlinesize = sizeof(utbuf.ut_line);

    if ( argc < 3 ){          // check usage
        fprintf(stderr,
                "usage: %s <utmp-file> <line>\n", argv[0]);
        exit(1);
    }

    // try to open utmp file
    if ( (fd = open(argv[1], O_RDWR)) == -1 ) {
        fprintf(stderr, "Cannot open %s\n", argv[1]);
        exit(1);
    }

    // If the line is longer than a ut_line permits do not
    // continue
    if ( strlen(argv[2]) >= UT_LINESIZE ) {
        fprintf(stderr, "Improper argument:%s\n", argv[1]);
        exit(1);
    }

    while ( read(fd, &utbuf, utsize ) == utsize )
        if ((strncmp(utbuf.ut_line, argv[2],utlinesize) == 0)
             && ( utbuf.ut_user[0] == '\0' ) ) {
             utbuf.ut_type      = DEAD_PROCESS;
```

```
                utbuf.ut_user[0] = '\0';
                utbuf.ut_host[0] = '\0';

                if ( gettimeofday(&utbuf.ut_tv, NULL) == 0 ) {
                    if ( lseek(fd, -utsize, SEEK_CUR) != -1 )
                        if ( write(fd, &utbuf, utsize) != utsize)
                            exit(1);
                }
                else {
                    fprintf(stderr, "Error getting time of day\n");
                    exit(1);
                }
                break;
        }
        close(fd);
        return 0;
}
```

Notice that every system call is tested for failure before its result is used (except for the call to `write()`). Here, the calls are embedded within the conditional expressions of the `if` and `while` statements above. The first `if` checks whether the record read in the `while` condition has the same terminal line as the one we are looking for (stored in the variable `line`) and the user member is not null. If this is successful, the type member `ut_type` of the record is set to the `DEAD_PROCESS` type, the user and host members are set to null strings, and the time member, `ut_tv`, is updated to the current time. If this is successful, the `lseek()` call moves the current pointer back to the start of the last matched record, so that the write operation that follows will replace the old record. If the write operation is reached and executes without error (determined by checking that the number of bytes written is equal to the number requested to be written), then the program returns 0 for success.

### 2.7.5 Another Use of lseek

One other use of `lseek()` is determining an open file's size without having to look at its properties. Recall that the return value of `lseek()` is the location of the file pointer after it has been moved, relative to the beginning of the file, and expressed in bytes. If we move the file pointer to the end of the file using `lseek()`, then we get its size as the return value. If `fd` is a file descriptor for the given file, then

```
size_t filesize = lseek(fd, 0, SEEK_END);
```

stores the size of the file into the variable `filesize`. We will make use of this soon.

## 2.8 Performance and Efficiency : Writing the cp Command

The `who` program was an exercise in reading a system data file and extracting information from it. It was a naive start, in that we did not pay much attention to its efficiency, which is of utmost concern with most software. To demonstrate the problem a bit more clearly, we will implement a different command, one whose efficiency or lack thereof will be much more obvious. Then we will take what we learned from that exercise and apply it to the `who` program in our final version. The command of interest is the `cp` command, which copies one or more files or directories.

### 2.8.1 What `cp` Does

If you are familiar with the `cp` command you can skip this section. There are several different ways in which the `cp` command can be used. The simplest is to make a copy of a single file:

```
$ cp source_file target_file
```

Whether or not `target_file` already exists, `cp` makes a copy of `source_file` named `target_file`. If it does exist, it will be overwritten, an act known as *clobbering*. This is dangerous, as you cannot recover the file once you have clobbered it. To prevent accidental overwrites, the interactive option `-i` should always be used, as in

```
$ cp -i source_file target_file
cp: overwrite ‘target_file’? n
```

It is a good idea to define an alias in the `.bashrc` file,

```
alias cp='cp -i'
```

so that you never forget to use the interactive mode.

If a new file is created, it will have the permissions and ownership of the source file. If an existing file is overwritten, it retains the permissions and ownership it had before the copy. No other attributes are preserved in a copy. To preserve the time-stamps and other attributes, you must use the `-p` (`p` for *preserve*) option.

Another form of the `cp` command is

```
$ cp source_file ... target_dir
```

in which the very last word on the command line, `target_dir`, is a directory and all preceding words are non-directory files. In this case, if the directory does not exist, it is an error. Otherwise all of the source files are copied into the directory with their existing permissions and names. If any names already exist in the target directory, the rules described above apply.

In the last form,

```
$ cp -r |-R source source ... target_dir
```

the sources can include directory names. All of the files and directories specified on the command-line, up to but not including `target_dir`, are copied into `target_dir`, which must already exist. The `-r` or `-R` option must be specified otherwise it is a syntax error. The `-r` specifies that the directories will be copied recursively. The `-R` is essentially the same; the difference has to do with how they handle pipes, which is unimportant now.

For the remainder of the chapter, we try to understand the implementation of the simple form of the command, without any options.

### 2.8.2 Opening/Creating Files For Writing

The `cp` command has to create a file if it does not exist and open it for writing, or overwrite it if it does exist. To overwrite a file, it is first truncated, i.e., its length is set to 0, and then the new data is written to the empty file.

#### 2.8.2.1 Creating/Truncating Files

The first task is to learn how to create files and truncate them. In fact, one call accomplishes both. The `creat()` system call is used to open a file for writing, if it exists, setting its length to 0 first, or if it does not exist, to create it. Notice that there is no "e" at the end of `creat`. If you type "`man creat`" you will get the man page for the `open()` system call:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *path, mode_t mode);
```

The `creat()` system call has two arguments, a C string and a `mode_t`. The string should contain the path name of the file to be created and the `mode_t` specifies the file's mode, i.e., its permission string, as an octal number. For example,

```
fd = creat("prototype", 0751)
```

creates a file named `prototype` in the current working directory, if it does not exist, with permission 0751 (owner can read, write, and execute, group can read and execute, others can execute only) provided that the process's *umask* does not modify the permission. Umasks are covered in the next chapter. If the file exists, the mode argument is ignored and the file is truncated[14]. In either case, upon termination of the call, `fd` is a file descriptor associated with the write-only connection to the file.

#### 2.8.2.2 Writing to Files

Having opened a file for writing, the next step is to write data into it. The `write()` system call is a symmetric counterpart to the `read()` call. It is used for writing sequences of bytes to the file specified by a given file descriptor:

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

[14] It is possible to prevent the file from being overwritten in case it exists, but not if you use the creat() call to try to create it. Instead the open() call must be used. Chapter 4 covers the various methods of opening a file for writing.

The size_t type stores the sizes of things in bytes. It is usually a typedef of an unsigned long integer, which may be 32 or 64 bits. The ssize_t type is almost the same as the size_t type. It differs only in that it is signed and that it can also store a –1. If successful, the write() call transfers nbyte bytes from the memory location pointed to by buf in the process's address space to the position of the file-pointer in the file associated with fd, and returns the number of bytes transferred. If the kernel cannot copy any of the data, write() returns –1.

The word "buffer" is used to describe the second parameter in the read() and write() system calls. It is declared as a void pointer. It is called a buffer because it is a storage location in the memory space of the calling process that is used to hold the data to be transferred to or from the file.

The code fragment

```
if (write(fd, buffer, num_bytes) != num_bytes ) {
        fprintf(stderr, "Problem writing to file.\n");
    }
```

attempts to transfer num_bytes bytes from the memory location pointed to by buffer to the position of the file pointer in the file opened for writing via the file descriptor fd. (By default, the file pointer is at "the end" of the file, unless it has been moved elsewhere.) The reason for the condition

```
if (write(fd,buffer,num_bytes) != num_bytes)
```

is that the return value of write() is the number of bytes actually written and it may not be equal to the number of bytes that were supposed to be written. The number of bytes successfully written may be less than num_bytes for any number of reasons. The file might have reached a predefined maximum size, the disk might be full, or the user's disk quota might be reached. This is why it is necessary to compare the return value of the write() call with the value of its third parameter.

### 2.8.3 A First Attempt at cp

The structure of the cp command is

*open the sourcefile for reading*
*open the copyfile for writing*
*while a read of data from the sourcefile to a buffer is not an empty read*
*write the data from the buffer to the copyfile*
*close the sourcefile*
*close the copyfile*

We know how to open and close files and we know how to read and write them, so this is a relatively easy program for us at this point. The only points that need explanation are how we create and use buffers. For example, how big should the buffer be? How do we declare it and pass it to the calls?

```
Listing cp1.c
// First attempt at cp command, based on a program
// by Bruce Molay in Understanding Uunix/Linux Programming, p.53

#include   <stdio.h>
#include   <unistd.h>
#include   <fcntl.h>

#define BUFFERSIZE      4096
#define COPYMODE        0644

void die(char* string1, char* string2);  // print error and quit

int main(int argc, char *argv[])
{
    int      source_fd, target_fd, n_chars;
    char     buf[BUFFERSIZE];

    if ( argc != 3 ){
        fprintf( stderr, "usage: %s source destination\n",
                 *argv);
        exit(1);
    }

    // try to open files
    if ( ( source_fd = open(argv[1], O_RDONLY))    == -1 )
        die("Cannot open ", argv[1]);
    if ( ( target_fd = creat( argv[2], COPYMODE)) == -1 )
        die( "Cannot creat", argv[2]);

    // copy from source to target
    while ( (n_chars = read( source_fd , buf, BUFFERSIZE) )
            > 0 ) {
        if ( n_chars != write( target_fd, buf, n_chars ) )
            die("Write error to ", argv[2]);
    }
    if ( -1 == n_chars )
        die("Read error from ", argv[1]);

    // close both files
    if ( close(source_fd) == -1 || close(target_fd) == -1 )
        die("Error closing files", "");

    return 0;
}

void die(char *string1, char *string2)
{
```

```
49            fprintf(stderr,"Error: %s ", string1);
50            perror(string2);
51            exit(1);
52   }
```

**Comments**

- The buffer is declared as an array of `BUFFERSIZE` chars, which is equal to the maximum number read in a `read()` call.
- The `die()` function encapsulates the error handling logic and calls the `perror()` function.
- Every call to a function in the API is checked for a possible error.
- The main work is in the while loop (lines 32-36). The entry condition is that the `read()` call transferred one or more bytes. The body is the call to write the bytes just read to the output file. The return value of `write()` is checked to see if the number of bytes transferred equals the number requested by the call.

If you compile and run this program you will see that it works correctly. But does it run fast? How long will it take to copy a very large file? How does one time programs in UNIX?

### 2.8.4 Timing Programs

The `time` command is a means of measuring the amount of time (and other resources) that a command uses. The `time` command has many options, but its simplest form is

```
$ time -p command
```

where `command` is the command that you wish to know about. The '`-p`' option tells `time` to display the traditional POSIX output, which consists of three values, each measured in seconds to two decimal places:

- Elapsed clock time, denoted “real”
- User time, denoted “user”
- System time, denoted “sys”

Elapsed time is the number of seconds from when the command was invoked until it completed. User time is the total amount of time that the process, and any children executing on its behalf, spent running in user mode. System time is the total amount of time spent on the process's behalf running within the kernel, i.e., in privileged mode, including such time spent by its children as well. Non-POSIX output may be more voluminous; you can read the man page for further details. Also, shells such as `bash` typically define their own version of the `time` command, so it is best to type the full path name when using it, if you want the non-`bash` version.

I created a file named `bigfile` containing about 30 MB of data. When I ran

```
$ time -p cp1 bigfile copy_of_bigfile
```

I got the following output on one of the UNIX systems at Hunter College:

```
real   4.05
user   0.01
sys    0.02
```

What accounts for the difference between the sum of user and system times and the elapsed time? It is the time that the process spent waiting for I/O to complete. When a process issues a request for I/O, it is blocked until the I/O is complete. The time that it spends in this blocked, or waiting, state is part of the elapsed (real) time. `cp1` spent about 4 seconds waiting for I/O. Although the amount of time that a process spends waiting for I/O depends heavily on what else the system is doing, the more calls it makes, the longer it will take, on average. The reason for this will be explained below.

As we use `cp1` on larger and larger files, we will see worse performance. To create a spreadsheet with the results of the time command I used a different option to it:

```
/usr/bin/time -f "\t%e\t%U\t%S"
```

The `-f` option expects a format string, which I supplied as a tab-separated string of real-time, user-time, and system-time format symbols. This allowed me to open the output with a spreadsheet program for analysis:

| File Size (bytes) | Real | User | Sys |
|---:|---|---:|---|
| 19,004,256 | 17.28 | 0.00 | 0.05 |
| 38,008,512 | 39.17 | 0.01 | 0.11 |
| 76,017,024 | 73.69 | 0.00 | 0.21 |

Notice that the real and system times increase roughly in proportion to the size of the file over this small sample.

### 2.8.5 Buffering and its Impact on Performance

Consider the `cp1` program above. Suppose that $N$ is the size of the file to be copied, measured in bytes. Then the while loop in lines 32 through 36 iterates $\lceil N/BUFFERSIZE \rceil$ times, since each iteration copies `BUFFERSIZE` bytes. It follows that as `BUFFERSIZE` is increased the number of iterations decreases inversely, i.e., if we double the buffer size, we halve the number of calls to both `read()` and `write()`. The question is, how is the total running time affected as the buffer size increases, in general? Is the amount of time to make a single call to `read()` proportional to the number of bytes to be read, or are there other components of its running time that are not related to the size of the read?

To answer this question, we will first perform a little experiment. We will revise the `cp` program so that the buffer size is an input parameter, and run the program on a very large input file with successively larger buffer sizes, recording the three components of running time reported by the time command for each run, and tabulating results. The revised program, called `cp2.c`, is in the listing below.

```
1  Listing cp2.c: a version of cp with buffer size given on the
2                 command line
3  // includes and defines omitted here
4
5  int main(int argc, char *argv[])
6  {
7      int     BUFFERSIZE;
8      char    endptr[255];
9      int     source_fd, target_fd, n_chars;
10     char    *buf;
11
12     // need to check for 3 arguments instead of 2
13     if ( argc != 4 ){
14         fprintf(stderr,
15                 "usage: %s buffersize source destination\n",
16                 argv[0]);
17         exit(1);
18         }
19     //  extract number from string
20     BUFFERSIZE = strtol(argv[1], (char**) &endptr, 0);
21     if (BUFFERSIZE == 0) {
22         fprintf(stderr,
23                 "usage: buffersize must be a number\n");
24         exit(1);
25     }
26
27     //  SNIP: code cut out here, including error handling
28
29     /* allocate buffer of size BUFFERSIZE */
30     buf = (char*) calloc(BUFFERSIZE, sizeof(char));
31     if ( NULL == buf ) {
32         fprintf(stderr,
33                 "Could not allocate memory for buffer.\n");
34         exit(1);
35     }
36
37     // Everything else is the same from this point forward,
38     // and omitted from the listing
```

For those who have not seen it before, `calloc()` (in line 30) and its companion, `malloc()` are dynamic memory allocation functions in C. The prototype for `calloc()` is

```
void *calloc(size_t nelem, size_t elsize);
```

Unlike `malloc()`, `calloc()` takes two arguments: the number of elements, and the size in bytes of each element, and it attempts to allocate space for an array of `nelem` elements, each of size `elsize`. If it is successful, it returns a `void*` pointer to the start of the array and fills the allocated memory with zeros. This pointer should be cast to the appropriate type before using it.

The table below shows the effect of buffer size on the elapsed, user, and system times when copying a file of size 19MB on a particular host in the Computer Science Department network at Hunter College running RHEL 4. As you can see, the user and system times roughly decrease in inverse proportion to the buffer size for most of the sampled range of values. The user time decreases because the process spends less time in its own code, since there are fewer iterations of the loop and hence fewer instructions to execute. The system time decreases for the same reason – the `read()` and `write()` system calls are executed fewer times and therefore less time is spent in the kernel. The elapsed time tends to reach a steady value after the buffer size reaches 16. Since the total of the user and system time continues to decrease for buffer sizes greater than 16, this suggests that the limiting factor is the time that the process spends waiting for the I/O operations to complete.

| **Buffer Size(bytes)** | **Real** | **User** | **Sys** |
|---:|---:|---:|---:|
| 2 | 50.19 | 3.11 | 28.27 |
| 4 | 33.27 | 1.59 | 13.09 |
| 8 | 24.28 | 0.76 | 6.08 |
| 16 | 22.56 | 0.39 | 3.08 |
| 32 | 20.53 | 0.21 | 1.57 |
| 64 | 21.66 | 0.10 | 0.78 |
| 128 | 20.12 | 0.04 | 0.43 |
| 256 | 18.27 | 0.02 | 0.24 |
| 512 | 19.70 | 0.00 | 0.15 |
| 1024 | 18.86 | 0.00 | 0.09 |

As the buffer gets larger, the kernel is called fewer times to transfer the data: as we stated above, if $N$ is file size and $B$ is buffer size, the number of calls is $c = \lceil N/B \rceil$. Another way to say this is that $cB$ is constant. The table shows that, if $s$ is total system time, $sB$ is also approximately constant, except for $B > 256$. In other words, the total system time is roughly proportional to the number of calls made for small values of $B$. For larger values of $B$, the total system time is not in proportion to the number of calls, but is larger than it. Why is this?

There are two components to the running time of an I/O operation: the *transfer time* and the *overhead*. The overhead is largely independent of the number of bytes to be read or written; each read or write request to the disk has overhead that does not depend much on how much data is to be transferred. This includes various components of time required by the device to set up and initiate the transfer. It also includes the cost of the system call itself, which is not always negligible.

The transfer time is the time that it actually takes to copy data between the device and memory and is a function of the amount of data. The kernel's involvement in this transfer in modern machines with DMA is minor; it mostly just starts it and does more work when it is finished. Nonetheless, the kernel's involvement is a function of the amount of data to be transferred. Therefore, if $B$ is buffer size, $O$ is the overhead of a I/O operation, and $t$ is a constant such that $tB$ is the amount of time the kernel spends in a single transfer operation, a single `read()` or `write()` system call uses $O + tB$ time units, and the program takes $(\frac{N}{B}) \cdot (O + tB) = \frac{ON}{B} + tN = N \cdot (\frac{O}{B} + t)$ time. Since

$N$ is the size of our data and does not change, you can see that the system time is proportional to $(\frac{O}{B} + t)$. This explains why the system time does not keep diminishing by half. Eventually the $t$ term is large in proportion to the $\frac{O}{B}$ term. When $O$ is very large in comparison to $t$, doubling B halves the expression, but otherwise it does not.

As we shall see shortly, in UNIX in particular, the design of a buffering system within the I/O system makes the transfer time on average even smaller.

### 2.8.6 System Call Overhead

System calls have overhead. When a user process makes a call to the kernel for some kind of service, the user process stops executing instructions in its own user space and starts executing instructions that are physically located in kernel space. Prior to making the call, the process executes the user program in a non-privileged mode, also known as *user mode*, and this phase of the process is called the *user phase*. During the system call, the process executes system code with the privileges accorded the kernel, and is said to be in *supervisor* or *kernel mode*; this is called the *kernel phase* of the process[15]. When the call terminates, this kernel phase terminates and the user phase resumes. This is a form of context-switch. A context-switch occurs when the kernel changes the currently executed memory image (the context). This can happen because a new process is run or because the kernel runs on behalf of a process, requiring that the memory image be switched. In some versions of UNIX such as Linux 2.6, a full context switch is not performed when a process changes from user phase to kernel phase or vice-versa.

The kernel needs to execute in kernel mode because it has to have access to all hardware instructions. In contrast, user processes must be prevented from executing special instructions. Therefore, when the system call is made, the machine must change mode twice, at the start and at the end of the call. It must also change the CPU state, because when the kernel runs, it has a different address space, different sets of resources, and so on. All of this changing means that a system call adds overhead to the running time of the program.

### 2.8.7 System Buffering

In addition to the overhead of the system call itself, there is overhead involved with `read()` and `write()` system calls. When a user process issues a read request from a disk, for example, the kernel does not transfer the data directly from the disk to the address space of the user process. Instead, it transfers the data from the disk to a buffer in kernel memory, and when all of the data has been transferred, it copies it into the user process's address space. This copying of data from kernel memory to user memory takes additional time. The symmetric situation occurs on writes: the kernel copies the data from the user address space into kernel memory, and from there it transfers it to disk[16].

UNIX uses this buffering scheme only for certain types of input and output[17], particularly for read

[15]On some UNIX systems, such as Linux 2.6, the user phase and kernel phase are called user mode and kernel mode respectively.

[16]There is a way to avoid this copying of data back and forth. Memory mapping is a method of I/O in which disk files are mapped directly into user memory. This topic will be discussed in a later chapter. If you are curious, read about the `mmap()` and `munmap()` system calls.

[17]There are two types of I/O in UNIX: block I/O and character I/O. The block I/O system in UNIX is used for block devices such as magnetic and optical disks and tapes. Character I/O is used for devices that are inherently

and write operations to and from disks. While it may seem at first that it just adds overhead, in fact it is a powerful and efficient method of reducing overall time spent performing I/O.

The buffering scheme for both reading and writing makes it seem as if read operations read directly from the device and write operations take place immediately. In fact, the kernel hides from the user an important layer of complexity. To understand this complexity, one needs to know a bit about how the disk is organized.

The disk is organized as a collection of fixed-size disk blocks. Disk blocks are numbered so that they can be identified. Each logical disk or disk partition has a unique name in UNIX, such as `sd0a` or `rsd2b`.

The kernel maintains a pool of buffers in kernel memory that can be assigned to each device. Each buffer is given a name, corresponding to the device to which it is assigned and the particular block whose contents it holds. For example, a buffer might be assigned block 511 from disk rsd2b.

On a read request by a process, the buffer pool is searched for a buffer whose name matches the block being sought on the disk. If a buffer is found, the data is read directly from memory without any physical I/O. If the buffer is not found, the data must be read from disk. A buffer will most likely have to be reused for this data. A least recently used (LRU) algorithm is used to decide which buffer to replace. After the buffer is selected, if it is "dirty" its contents are written to disk. Buffers are dirty if they were modified since the last time they were written to disk. The buffer is renamed to match the block being read and the read is performed.

Write requests are handled similarly. When a process requests a write to a specific block on a disk, the buffer pool is searched and if a buffer is not found whose name matches the disk address to be written, a new buffer is allocated for this write operation. If no buffer is available, a block is chosen using the LRU algorithm and relabeled. The data is stored in the buffer without any physical I/O (i.e, disk accesses) and the buffer is marked dirty. The write will be performed only when the block is renamed.

Note that this scheme can greatly reduce the need to perform disk I/O, because reads and writes can take place in memory, which is much faster, and it is completely transparent to the user. But what happens if the system suddenly comes to an unexpected halt? Unless the system has time to "flush" its buffers, the updates are lost. This is why one should never halt a system in the wrong way.

The advantages of buffering are a reduction in physical I/O and therefore a decrease in the overall effective disk access time. The disadvantages include that

- I/O error reporting can lag behind the logical I/O and therefore can become meaningless,
- delayed disk writes can cause loss of data and file system inconsistencies in the event of unexpected system halts, and
- the order in which buffers are written to the external device may not be the same as the order in which the logical I/O occurs, and unless programs are designed with this in mind, disk-based data structures can become inconsistent.

Writes to sequential devices such as tape drives generally do not exhibit this problem because the drivers are only allowed one outstanding write request per drive. In other words, if a logical write

one-character-at-a-time devices, such as the keyboard and terminals in general. Character I/O does not use kernel buffers for I/O. All block I/O uses the kernel's buffering system.

operation is requested for a particular drive, but there is a request that has not yet been satisfied by a physical write, the second request cannot be satisfied until the first physical write takes place. A device like a tape drive will reject requests for service until it finishes what it is doing. It is a one-job-at-a-time device.

In Linux 2.6 and later, the kernel offers a service named direct I/O for processes that wish to bypass the kernel buffering system for block I/O. Certain types of programs such as database servers need to implement their own caching schemes for efficiency. Forcing them to also use the kernel buffering system would slow them down significantly and make the system inefficient, because then there would be duplicate copies of blocks: those in the database server's cache and those in the kernel's cache. With direct I/O transfers, the kernel transfers data directly between the disk and user space. Unfortunately, there are many problems associated with direct I/O, which you can read about in the man page for the `open()` system call. An apt conclusion is reached at the bottom of that page, with a quote from Linus Torvalds:

> In summary, `O_DIRECT` is a potentially powerful tool that should be used with caution. It is recommended that applications treat use of `O_DIRECT` as a performance option which is disabled by default.
>
> "The thing that has always disturbed me about `O_DIRECT` is that the whole interface is just stupid, and was probably designed by a deranged monkey on some serious mind-controlling substances."
>
> — Linus

### 2.8.8 Memory Mapped I/O

Memory mapping is a way to perform I/O without kernel buffering, and it is fully supported on almost all systems. The concept has been around for a long time. The idea in its simplest form is easy to understand: a process can request that a file be mapped to a set of virtual memory addresses. Changes to those addresses are, in effect, writes to the file. Reads of those addresses are reads of the file.

The actual use of the memory mapping system calls, `mmap()` and `munmap()`, is a bit more complex than this. The purpose of `munmap()`, as its name suggests, is to undo a mapping. The `mmap()` call has several parameters. We introduce memory mapping by writing the `cp` program a third way, using memory mapped I/O instead of reading and writing.

The basic idea is to follow the sequence of steps outlined below:

1. Map the entire input file to a region of memory. Assume it starts at address `source_addr`.
2. Determine the size of the input file in bytes. Call it `filesize`.
3. Create an output file with the given name and make it the same size as the input file.
4. Map the output file to a region of memory the exact same size as the file. Assume it starts at address `dest_addr`.
5. Do a single memory-to-memory copy of `filesize` bytes from `source_addr` to `dest_addr` using `memcpy()`.

6. Undo the mappings and close the files.

This causes the input to be copied to the output without any reads or writes. In order to implement these steps we need to know the prototypes of the mapping functions and `memcpy()`. The prototypes are

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

int munmap(void *addr, size_t length);
```

The `mmap()` call creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in the first argument, `addr`. The second argument, `length`, specifies the length in bytes of the mapping.

If `addr` is `NULL`, then the kernel chooses the address at which to create the mapping; this is the most portable method of creating a new mapping. If `addr` is not `NULL`, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary. The address of the new mapping is returned as the result of the call. It is best to always use `NULL` as the first argument.

The third argument describes the memory protection of the mapping; it must not conflict with the open mode of the file. The possible values are

`PROT_EXEC` Pages may be executed.

`PROT_READ` Pages may be read.

`PROT_WRITE` Pages may be written.

`PROT_NONE` Pages may not be accessed.

They can be or-ed together. In other words, if the file was opened read-only (`O_RDONLY`), then the value should be `PROT_READ`. If it was opened read-write, then it should be set to `PROT_READ|PROT_WRITE`. A warning about this follows below.

The fourth argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file. This behavior is determined by including exactly one of the following values in flags:

`MAP_SHARED` Share this mapping. Updates to the mapping are visible to other processes that map this file, and are carried through to the underlying file. The file may not actually be updated until `msync()` or `munmap()` is called.

`MAP_PRIVATE` Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the `mmap()` call are visible in the mapped region.

Because we want to do I/O we need to set the flag to `MAP_SHARED`, otherwise no changes will appear in the output file. There are other values that can be or-ed to this flag, but we will not discuss them at this point.

The next two arguments are the file descriptor of the file to be mapped and the offset in bytes relative to the start of the file at which to map the file. In other words, if you want to map only the portion of the file after the first N bytes, you would pass N as the last argument.

What you need to know is that the memory region is always a multiple of the page size of the machine and must be allocated as such. If the length is not a multiple of page size, the last page will be partly filled. The starting address must always be a multiple of page size. For now this is not our concern. After we learn how to get the page size of the machine, we will return to this issue.

A caveat – the documentation on my Linux system states that `mmap()` has been deprecated in favor of `mmap2()`, but `mmap2()` does not exist on it. In fact, `glibc` (GNU's C Standard Library) implements `mmap()` as a wrapper for the kernel's `mmap2()` call, so `mmap()` is actually `mmap2()`.

Our third copy program is in the listing below. It does not include all of the error-checking and handling that it should, but most is included. It makes use of `memcpy()` to do the actual transfer of bytes from the source to the destination, but it does so within memory. The prototype for `memcpy()` is

```
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```

where `src` is a pointer to the start of the memory to be copied, `dest` is the starting address where the bytes should be written, and `n` is the number of bytes to copy. The memory areas cannot overlap. In other words the absolute value of (`dest - src`) must be greater than `n`.

```
Listing cp3.c -- a copy program using memory-mapped I/O

#include <sys/mman.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include "../utilities/die.h"

#define COPYMODE        0666

int main(int argc, char *argv[])
{
    int      in_fd, out_fd;
    size_t   filesize;
    char     nullbyte;
    void     *source_addr;
    void     *dest_addr;
```

```
    /* check args        */
    if ( argc != 3 ){
        fprintf( stderr, "usage: %s source destination\n", *argv);
        exit(1);
    }

            /* open files */
    if ( (in_fd = open(argv[1], O_RDONLY)) == -1 )
        die("Cannot open ", argv[1]);

    /* The file to be created must be opened in read/write mode
       because of how mmap()'s PROT_WRITE works on i386 architectures.
       According to the man page, on some hardware architectures (e.g.,
       i386), PROT_WRITE implies PROT_READ.  Therefore, setting the
       protection flag to PROT_WRITE is equivalent to setting it to
       PROT_WRITE|PROT_READ if the machine architecture is i386 or the
       like. Since this flag has to match the flags by which the mapped
       file was opened, I set the opening flags differently for the
       i386 architecture than for others.
    */
#if defined (i386) || defined (__x86_64) || defined (__x86_64__) \
      || defined (i686)
    if ( (out_fd = open( argv[2], O_RDWR | O_CREAT |
           O_TRUNC, COPYMODE )) == -1 )
        die( "Cannot create ", argv[2]);
#else
    if ( (out_fd = open( argv[2], O_WRONLY | O_CREAT |
           O_TRUNC, COPYMODE )) == -1 )
        die( "Cannot create ", argv[2]);
#endif
    /* get the size of the source file by seeking to the end of it:
       lseek() returns the offset location of the file pointer after
       the seek relative to the beginning of the file, so this is a
       good way to get an opened file's size.
    */
    if ( (filesize = lseek(in_fd, 0, SEEK_END) ) == -1 )
        die( "Could not seek to end of file", argv[1]);

    /* By seeking to filesize in the new file, the file can be grown
       to that size. Its size does not change until a write occurs
       there.
    */
    lseek(out_fd, filesize -1, SEEK_SET);

    /* So we write the NULL byte and file size is now set to filesize.
    */
    write(out_fd, &nullbyte, 1);
```

```
    /* Time to set up the memory maps */
    if ( ( source_addr = mmap(NULL, filesize , PROT_READ,
              MAP_SHARED, in_fd , 0) ) == (void *) -1 )
        die( "Error mapping file ", argv[1]);

    if ( ( dest_addr = mmap(NULL, filesize , PROT_WRITE,
              MAP_SHARED, out_fd , 0) ) == (void *) -1 )
        die( "Error mapping file ", argv[2]);

    /* copy the input to output by doing a memcpy */
    memcpy( dest_addr , source_addr , filesize );

    /* unmap the files */
    munmap(source_addr , filesize );
    munmap(dest_addr , filesize );

    /* close the files */
    close(in_fd);
    close(out_fd);

    return 0;

}
```

## 2.9 Returning to `who`

Our previous implementations of `who` read one `utmp` record at a time. Each read requires a system call, even though a single `utmp` record is quite small and there are many of them. We now know that this is inefficient. Just as the `cp` command can benefit by increasing buffer size, so too can `who`. We will modify it so that it reads several `utmp` records at a time and stores them in an internal array. We are now up to version 5, and this version will be called `who5.c`[18].

### 2.9.1 User-Defined Buffering

A process is said to perform *input buffering* when it requests more data than it can process in an input operation and stores the extra data in its own memory space until it is ready to use it. Input buffering is a way to reduce the cost of input operations because it decreases the amount of time that the process spends in system calls.

In order for `who` to perform input buffering, it needs a place to store the extra records until it is ready to use them. The logical place is in an array of records. If it reads 20 records at a time, for example, then these 20 records will be placed into its internal array. It can maintain a pointer to a *current record*. Each time it needs to examine a new record, it checks whether the current record pointer has exceeded the array bounds. If it has, it attempts to fetch the next 20 records from the

[18] This idea is borrowed from Bruce Molay, *Understanding Unix/Linux Programming*, Prentice Hall.2003.

`utmp` file and fill the array with them. If no records are left in the file, it cannot obtain a new record, and it is finished. Otherwise, it fetches as many as it can, up to 20, and then gets the current record from the array and advances the current record pointer.

The logic for input buffering is encapsulated into a separate library of routines for interacting with the `utmp` records, called `utmp_utils.c`. The interface to this library consists of three functions: `open_utmp()`, `next_utmp()`, and `close_utmp()`. The `open_utmp()` function opens the given `utmp` file, the `next_utmp()` function delivers the next record, reading a new chunk from the file if the buffer is empty, and the `close_utmp()` closes the file. The interface follows.

```
Listing utmp_utils.h
typedef struct utmp utmp_record;

int open_utmp( char * utmp_file );
// opens the given utmp_file for buffered reading
// returns: a valid file descriptor on success
//          -1 on error

utmp_record *next_utmp();
// returns: a pointer to the next utmp record from the
//          opened file and advances to the next  record
//          NULL if no more records are in the file

void close_utmp();
// closes the utmp file and frees the file descriptor
```

The implementation of the library is next. It uses global variables (static variables) so that the functions can communicate. We do not want to pass these as parameters, because then client code would have to do that as well, breaking the abstraction. If this were written in C++, this library would be a class instead, and the globals would be member variables.

```
Listing utmp_utils.c
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <utmp.h>

#define NUM_RECORDS            20
#define NULL_UTMP_RECORD_PTR ((utmp_record *) NULL)
#define SIZE_OF_UTMP_RECORD  (sizeof(utmp_record))
#define BUFSIZE              (NUM_RECORDS * SIZE_OF_UTMP_RECORD)

static char utmpbuf[BUFSIZE];    // buffer of records
static int  number_of_recs_in_buffer; // num records in buffer
static int  current_record;      // next rec to read
static int  fd_utmp = -1;        // file descriptor for utmp file

```

```
int open_utmp( char * utmp_file )
{
    fd_utmp = open( utmp_file, O_RDONLY );
    current_record   = 0;
    number_of_recs_in_buffer = 0;
    return fd_utmp;    // either a valid file descriptor or -1
}

int fill_utmp()
{
    int    bytes_read;

    // read NUM_RECORDS records from the utmp file into buffer
    // bytes_read is the actual number of bytes read
    bytes_read = read( fd_utmp , utmpbuf, BUFSIZE );
    if ( bytes_read < 0 ) {
        die("Failed to read from utmp file","");
    }

    // If we reach here, the read was successful
    // Convert the bytecount into a number of records
    number_of_recs_in_buffer = bytes_read/SIZE_OF_UTMP_RECORD;

    // reset current_record to start at the buffer start
    current_record  = 0;
    return number_of_recs_in_buffer;
}

utmp_record * next_utmp()
{
    utmp_record * recordptr;
    int             byte_position;

    if ( fd_utmp == -1 )
        // file was not opened correctly
        return NULL_UTMP_RECORD_PTR;

    if ( current_record == number_of_recs_in_buffer )
        // there are no unread records in the buffer
        // need to refill the buffer
        if ( utmp_fill() == 0 )
            // no utmp records left in the file
            return NULL_UTMP_RECORD_PTR;

    // There is at least one record in the buffer,
    // so we can read it
    byte_position = current_record * SIZE_OF_UTMP_RECORD;
    recordptr = ( utmp_record *) &utmpbuf[byte_position];
```

```

    // advance current_record pointer and return record pointer
    current_record++;
    return recordptr;
}

void close_utmp()
{
    // if file descriptor is a valid one, close the connection
    if ( fd_utmp != -1 )
        close( fd_utmp );
}
```

**Comments**

1. In `next_utmp()`, if

   `( current_record == number_of_recs_in_buffer )`

   is true, it means that the number of records read so far is equal to the number of records in the buffer, which implies that it is time to read from the file again.

2. In `next_utmp()`, the line

   ```
   recordptr = ( utmp_record *) &utmpbuf[byte_position];
   ```

   sets `recordptr` to point to the address of the array entry at the given byte position. We have to cast the address of the linear array of bytes to a `utmp_record` pointer type.

The main program must be revised to use these functions, as follows.

```
Listing who4.c
#include "utmp_utils.h"

int main(int argc, char* argv[])
{
    utmp_record *utbufp;          // pointer to a utmp record

    if ( open_utmp( UTMP_FILE ) == -1 ){
        perror(UTMP_FILE);
        exit(1);
    }
    while ( ( utbufp = next_utmp() ) != NULL_UTMP_RECORD_PTR )
        show_info( utbufp );

    close_utmp( );
    return 0;
}
```

### 2.9.2 Final Comments

This last version of the `who` command improved performance by reading larger amounts of the file at a time, thereby reducing the overhead of disk reads. It follows that if we could read the entire file all at once with a single `read()` call, then we would reduce the amount of overhead to the least it could be. In fact, some versions of the `who` command do precisely this. At this point we cannot write this implementation because it depends upon our knowing how to use the `stat()` system call and some knowledge of the structure of the file system, which will come later. However, this method has a pitfall: the file may be larger than the available memory for the process. In this case, the program must be able to identify this and adjust how it reads the file. The GNU implementation of `who` does exactly this.

# Appendix A

## A.1 Filters: An Introduction

A *filter* is a program that gets its input from the standard input (`stdin`), transforms it, and sends the transformed input to the standard output (`stdout`). The data passes "through" the filter, which typically has command-line options that control its behavior. A filter may also perform a "null" transformation, making no change at all to its input (which is what `cat` does.) Filters process text only, either from input files or from the output end of another Unix command (i.e., through a *pipe*.) All filters can be given optional filename arguments, in which case they take their input from the named files rather than from standard input. For example, in the command

```
$ cat first second third > combinedfile
```

`cat` reads files `first`, `second`, and `third` in that order and concatenates their contents, sending them to the standard output, which has been redirected to a file named `combinedfile`.

The most useful filters are

| | |
|---|---|
| `cut` | (usually System V only)1 simple text cutting |
| `grep` | simple regular expressions as filtering pattern |
| `egrep` | extended (more powerful) regular expressions as filtering patterns |
| `fgrep` | fast, string matching expressions with alternation as patterns |
| `sed` | line-oriented text editing filter |
| `awk` | pattern-matching, field-oriented filter and full-fledged Turing computable programming language |
| `cat` | primitive filter with little transformation |
| `sort` | very general sorting filter |
| `head,tail` | lets only the top or bottom of a stream pass through |
| `fold` | wraps each input line to fit in a specified width |

If your time is limited and you could learn but one of these, the most important would be `grep` – the return on your investment will be greatest. Coming in second would be `sed`, and then `awk`. The remaining filters are easy to learn and use and are described briefly first.

### A.1.1 `sort`

`sort` is easy to use:

```
$ sort file
```

will sort the text file named `file` and print it on standard output. By default is uses collating order, the order of the characters in the character code of the terminal, which is usually ASCII or UTF-8. In this case uppercase letters precede lowercase letters. There are versions of `sort` that ignore case by default, but if your does not, you can turn off case-sensitivity with the `-i` option.

If you want to sort numerically, use the `-n` option, as in

```
$sort -n numeric_data
```

which will sort numbers correctly. Without the `-n`, 9 will precede 10 because 1 precedes 9 in the collating sequence. Read the man page for details.

### A.1.2 head and tail

Simply put, `head` displays the first $N$ lines of its input and `tail`, the last $N$ lines. By default $N = 10$. To print a different number of lines, use

```
$ head -N
```

or

```
$ tail -N
```

respectively.

### A.1.3 cut

`cut` is a lesser filter. You will rarely use it. It does simple tasks well. It cuts out selected pieces of lines of the input.

```
$ cut -c1-10
```

copies the first 10 characters from every line, removing the rest.

```
$ cut -f2,4
```

copies only fields 2 and 4 of every line to the output stream. Fields are delimited by the TAB character unless the delimiter character is changed using the `-d` option. Fields are 1-based, so the first field is field 1. The delimiter must be a single character:

```
$ cut -f1,5 -d: /etc/passwd
```

will display fields 1 and 5 of the `/etc/passwd` file, which are the username and gcos fields.

### A.1.4 Regular Expressions and grep

We focus on `grep` and regular expressions. The regular expressions used by `grep` are the same as those used by `sed` and the visual text editor, `vi`. The simplest form of the `grep` command is

```
$ grep <regularexpression> files
```

where *<regular expression>* is an expression that represents a set of zero or more strings to be matched. The syntax and interpretation of regular expressions is found in the `regex` man page in Volume 7, as well as the man page for `grep`, so typing

```
$ man 7 regex
```

or

```
$ man grep
```

will give you everything you need to know on how to use them. The simplest patterns are strings that do not contain regular expression operators of any kind; those match themselves. For example,

```
$ grep  print file1 file2 file3
```

prints each line in files `file1`, `file2`, and `file3` that contains the word "`print`". It will print these in the order in which the files are listed, first lines in `file1`, then `file2`, then `file3`. If you want just a count of those lines, use the `-c` option; if you want the non-matching lines, use the `-v` option. If you want the line numbers, use the `-n` option. There are many more useful options described in its man page.

If you want to match a string that contains characters that have special meaning to the shell, such as white-space, asterisks, slashes, dollar-signs, and so on, it should be enclosed in single-quotes:

```
$ grep  'atomic energy' file1 file2 file3
```

will match all lines in the given files that have the exact string '`atomic energy`' somewhere in the line. Note that the lines merely have to contain the string as a substring; they do not have to match the the string exactly. If you want the pattern to match an entire line, you have to bracket it with operators called *anchors*. The start of line anchor is the caret `^` and the end of line anchor is the dollar sign `$`:

```
$ grep  '^atomic energy$' file1 file2 file3
```

matches lines in the given files that are exactly the string atomic energy.

Regular expressions can be formed with various operators such as the asterisk `*`, which multiplies the expression to its left 0 or more times, as in

```
a*
```

which matches strings with zero or more a's: a, aa, aaa, and the null string. To match a string like ababab, you have to enclose it in \(...\), as in

```
\(ab\)*
```

which matches 0 or more sequences of ab. Note that

```
(ab)*
```

will match strings like (ab)(ab)(ab), not ababab because in regular expressions, the parentheses by themselves are not special characters.

The period matches any character. There are character classes, which are formed by enclosing a list (or a range) in square brackets []. A character class represents a single character from that class. Because the special characters in regular expressions typically have special meaning in the shell as well, it is a good idea to always enclose the pattern in single quotes. In particular, if you give it a regular expression using an asterisk you must enclose the string in quotes[1].

#### A.1.4.1 Examples

In the following examples, the file argument is omitted for simplicity. In this case grep would apply the pattern against standard input, which means if you actually type this, it will wait for you to enter text followed by an end-of-file signal, Cntrl-D.

```
$ grep 'while *(.*)'
```

matches lines containing the word 'while' followed by zero or more space characters, followed by a parenthesized expression.

```
$ grep '^[a-zA-Z][a-zA-Z0-9_]*'
```

matches lines that begin with a word that starts with a letter, upper or lowercase, following by zero or more letters or digits or underscores.

```
$ grep '[0-9][0-9]*\.[0-9][0-9]\>'
```

The pattern selects strings that have 1 or more digits followed by a single period, followed by exactly two digits. The period must be preceded by a backslash so that grep does not treat the period as the special character meaning "match any character". The "\>" tells grep to anchor the pattern to the end of the word. A word is a sequence of letters and/or digits. This forces grep to select only those words that end in two digits. If I omitted the "\>" grep would have matched strings such as 1.234 or 1.23ab. There is a matching operator, \<, that anchors to the beginning of the word.

Now take a look at this one.

[1] Single quotes are better than double quotes. Single quotes prevent the shell from doing any interpretation of the enclosed characters, whereas when the shell sees a double-quoted string, it does a certain amount of interpretation. Until you understand what the shell will attempt to interpret inside double-quoted strings, use single quotes for enclosing grep patterns.

```
$ grep '\/\*.*\*\/'
```

Since `/` is a special character, if I want to match it I have to escape it with a \ like this: `\/`. Similarly, since `*` is a special character in regular expressions, `\*` is how you have to match a single asterisk `*`. So to match the two-character sequence `/*` I have to write `\/\*` and to match `/*` followed by any number of characters and then followed by `*/`, I have to write

```
\/\*.*\*\/
```

in which `.*` matches zero or more characters of any kind (including the period itself). This finds lines with C-style comments in them.

Regular expressions also provide a means of "remembering" matched expressions, for re-use in the expression. This is very handy in `vi` and `sed`, which have substitution operators. The same operator used for grouping is also used for remembering matching strings. The remembered string is then referenced using the *back-reference* `\1` (or `\2`, `\3`... if there are multiple strings remembered):

```
$ grep '\([a-z]\)\1\1\1\1'
```

matches any line that contains a sequence of 5 copies of a letter, such as `xxxxx` or `bbbbb`.

```
$ grep '\([1-9][0-9]\).*\1'
```

matches any line that has a two digit number that is repeated later in the line. The command

```
$ grep '\([a-z]\)\([a-z]\)\([a-z]\)\3\2\1'
```

has three remembered matches in the back-references `\1`, `\2`, and `\3`, but in reverse order. Each will have a copy of the single lower-case letter that it matched, so this pattern matches palindromes of length 6 such as `xyzzyx`.

You are encouraged to read the man page for grep. There is a lot more to regular expressions than is covered here. The best way to learn them is to experiment. You can open a terminal window and type grep followed by a pattern. It will then wait for you to type lines on the keyboard. Lines that match will be repeated. Lines that don't will not. Try it.

### A.1.5 The Rest of the `grep` Family

#### A.1.5.1 `egrep`

`egrep` (*extended grep or expression grep*) has a larger set of regular expressions meta-symbols than `grep`, including '|', '?', '+', and parentheses. It is not a strict superset of `grep` because it does not allow `\( \)`, `\{ \}`, `\< \>`. These are equivalent to `()`, `{}`, and `<>`, in egrep.

For example, you can write

```
$ egrep 'March|April|May'
```

and

```
$ egrep 'M(iss)+ippi'
```

which matches Mississippi as well as Mississississippi. Another extension in egrep is the + operator. A "+" after a regular expression indicates to search for one or more occurrences of the regular expression, as in

```
$ egrep '[a-z]+'
```

which matches 1 or more letters.

#### A.1.5.2 fgrep

The `fgrep` variant of `grep` does not support regular expressions but does support multiple strings. It is used to search quickly for many different fixed strings. For example, you can put a list of frequently misspelled words into a file and then call `fgrep` to search for them:

```
$ fgrep -f errors document
```

will print all lines in `document` that contain one of the strings in the file named `errors`.

## A.2 File Globs

All UNIX shells have the ability to parse patterns that represent sets of files. These patterns are called *file globs*, or simply *globs*, or *wildcard* expressions. In essence, the shell will replace a file-glob by the list of files that it represents. For example,

```
$ ls *.c
```

is a command to list all files in the current working directory that have zero or more characters followed by a "`.c`".

The regular expressions that the shell uses for file-globbing have a different syntax from those used by `vi`, `grep`, and the other filters and commands. They are not really regular expressions. File-globs are more limited, and the asterisk `*` does not multiply the character that precedes it. It, by itself, represents zero or more characters of any kind. Thus,

```
$ rm *.o
```

removes all files ending in "`.o`" and

```
$ for i in hwk2_*.gz ; do unzip $i ; done
```

will run `unzip` on every file in the current working directory whose name starts with hwk2_ and ends in ".gz" (in `bash` and `sh` and other Bourne-shell-like shells). You must be very careful when using file globs, especially with dangerous commands such as rm that are not reversible, because they may represent files that you did not think they did. One disastrous example would be

```
$ rm -r .*
```

which a naive user might think removes the "hidden" files in the given directory and their descendants. But the pattern `.*` matches `..`, which implies that the command will recursively remove everything in `..`, the parent directory. There are many other things to know about file globs; the complete description can be found in the man page in Volume 7:

```
$ man 7 glob
```

will display it.