SYSTEM PROGRAMMING

From the book by STEWART WEISS

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
- The utmp file
- Detecting and reporting errors in system calls
- Memory mapped I/O
- *open, creat, close, read, write, lseek, perror, ctime, localtime, utmpname, getutent, setutent, endutent, malloc, calloc, mmap, munmap, memcpy*
- Filters and regular expressions

Commands are (Usually) Programs

- In UNIX, most commands are programs, almost always written in C.
- Shell builtins (e.g., *cd* and *exit*) are not programs.
- Some commands, such as *pwd*, are both shell builtins and programs.
- By default the shell builtin *pwd* will be executed; or one can either type *\pwd* or */bin/pwd*

Locating Command Binaries

- The most common locations:

  /bin
  /usr/bin
  /usr/sbin: Administrative commands
  /usr/local/:* Packages installed after the OS installation
  /usr/local/bin: Commands that do not come with the UNIX distribution
  /usr/ucb: (ucb: University of California at Berkeley)

The *who* Command

Displays info about who is currently using the system.

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)
```

Researching Commands In UNIX

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.

Reading Man Pages

- man page for a command have the DESCRIPTION, SEE ALSO, and FILES sections
- DESCRIPTION gives the details of usage.
- For example, man page for *who* says:

  who has an optional file name argument and its default is /var/run/utmp which has the info about current logins. The optional argument can be /var/log/wtmp

- We can infer that */var/run/utmp* contains info about who is currently logged in.
- There is a section of the man pages for the description of system file formats so man wtmp will bring the man page for the wtmp file.
- /var/log/wtmp and /var/run/utmp are system files and they are described on the section 5 of the manual.
- There we can learn that /var/log/wtmp contains information about who has logged in previously

Man Pages and Headers

- All POSIX-compliant UNIX systems also contain man pages for all of the header files included by a function in the kernel's API.
- For example
  - *$ man stdlib.h*

  will display the man page for the header file <stdlib.h>

Man Page Searching

- 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*
- To this work; *whatis* database should have been built when the man pages were installed.

Man Page Searching

- *$ man k utmp* or
- *$ apropos utmp*

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

- On some systems apropos allow multiple keyword and regular expression searches.
- The -a option makes search in man pages whose page names and/or NAME sections contain all keywords provided, 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

Man Page Searching: Examples

- *$ 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*

- *$ 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 lter it further with another instance of grep:*

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

Texinfo

- The man page for a command may not have enough content, SEE ALSO section will have a message such as the following:

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.

Texinfo: info command

- The info command brings Texinfo pages, an alternative to man pages.
- To learn how to use the Texinfo viewer, type
  - *info info*
- The information is stored in Texinfo as a tree-like structure.
- 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 BFS search.
- Shortcuts are d (down) u (up) n (next) p (previous).

Digging Deeper into the who Command

- The manual search on the utmp file outputs like:

[Topic]	[Title]	[Chapter]	[Brief Description]
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	*[utmp]*	*(5)*	*- login records*
utmpname	*[getutent]*	*(3)*	*- access utmp file entries*
utmpx.h	*[utmpx]*	*(0p)*	*- user accounting database definitions*
wtmp	*[utmp]*	*(5)*	*- login records*

Man Page for utmp

- To display the specific chapter, type:
  $ man 5 utmp
  $ man S5 utmp
- The beginning of the man page for utmp is displayed below (may change from distro to distro):

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):

(...)

Reading the Correct Header Files

- *`echo int main() {return 0;} > empty.c`*
- *gcc -v empty.c*
- The output produced by gcc:

  #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

utmp.h (stripped down version)

```
/* 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.
};
```

What Next?

- We see, *who* opens the *utmp.h* and reads the structures in sequence and displaying the appropriate data for each login.

- We will use this info as the basis for our own implementation of the command.

Writing *who* command

- The program that implements the *who* command has two key tasks:
  - read the *utmp* structures from a file
  - display the information from a single *utmp* structure on the display device in a user friendly format.

Reading Structures From a File

- A binary file, e.g., *utmp*:
  - Consists of a sequence of structures
  - Cannot be read using the C I/O functions, such as *get(), getc(), fgets(),* and *scanf(),* nor the *istream* methods in C++.

- Suppose you do not know the methods of reading from a binary file.
- You search man pages as:

  - *$ man k binary file | grep read*

Reading Structures From a File (Contd.)

- Search result may give:
  - fread(), in Section 3, is part of the C Standard I/O Library.
  - read(), in Section 2, is the prototype of a system call.

- Type:

  $ man 2 read

  NAME
  read - read from a file descriptor
  SYNOPSIS
  #include <unistd.h>
  ssize_t read(int fildes, void *buf, size_t nbyte);
  DESCRIPTION
  …

The Difference Between <stdio.h> and <unistd.h>

- Functions *fopen(), fread(), fwrite(), fclose(),* etc. defined in *<stdio.h>* operates on FILE pointers and are part of ANSI C I/O Library. They are C functions one can use on any OS.
- Functions *open(), read(), write(),* and *close()* are UNIX system calls defined in *<unistd.h>*
- The *<unistd.h>* defines misc. symbolic constants and types, and declares misc. functions.
- These functions exist only in POSIX-compliant UNIX systems.
- These system calls operate on file descriptors, not file streams.
- The UNIX system calls operate on the kernel directly while ANSI C I/O Library calls are at a higher level.

The *read()* System Call

- 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.
- The second parameter is a pointer to memory buffer.
- 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.

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

- According to manpage of *open()*:

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

- The value of *oflag* is one of the following constants dened in *<fcntl.h>*:
- *O_RDONLY* open for reading only
- *O_WRONLY* open for writing only
- *O_RDWR* open for reading and writing

A First Attempt at Writing *who*

```
#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;
}
```

```
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
}
```

A First Attempt at Writing *who*

- *$ who1*

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

A First Attempt at Writing *who*: Problems

- There are records in the output that do not correspond to user logins
- Login times are in some strange format

A Second Attempt at Writing *who*

- The file */usr/include/utmp.h* contains denitions 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
```

A Second Attempt at Writing *who*

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

A Third Attempt at Writing *who*

- First two 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

A Third Attempt at Writing *who*

There is a simple way of reading the records in a *utmp* file:

1. Use *utmpname()* to select the file 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. If no record in file, NULL pointer returns.
4. Call *endutent()* when finished.

A Third Attempt at Writing *who*

```
#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) //getutent_r() is thread safe version of getutent()
            show_info(&utbuf);

      endutent();
      return 0;
}
```

Summary of *who*

- Man pages and header files can be used to learn about a command to implement.
- The utmp interface may not be the same on every UNIX system. There are GNU, non-POSIX, thread-safe version of the interface, and POSIX-compliant *utmpx* interface.
- A truly portable solution includes test macros to conditionally compile the code depending on the target system.

Using a File in Read/Write Mode

- *open()* system call's second parameter is access mode flags: *O_RDONLY, O_WRONLY*, and *O_RDWR*.
- *O_RDWR* opens file in read/write mode.
- The file 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 created by the kernel when a file is opened.
- It points to the position of the next byte to read/write in the file.

OS Logout Records

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.
4. Close the *utmp* file.

Using *lseek* to Move the File Pointer

- 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)
```

- fd is the file descriptor returned by *open()*. The distance (in bytes), dist, is used to move the current position pointer. If dist is positive, it moves forward; if it is negative, it moves backwards. The value of base flag determines the starting position of the current position pointer from which it is to be moved:

1. SEEK_SET dist is forwards relative to the start of the file,
2. SEEK_CUR dist, is relative to the current position pointer and may be positive or negative
3. SEEK_END dist, is relative to the end of the file and may be positive or negative.

Another Use of *lseek()*

- One other use of *lseek()* is determining an open file's size:

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

Performance and Efficiency : The *cp* Command

- The simplest usage is to make a copy of a single file:

- *$ cp source_file target_file*

- The *cp* command has to create a file if it does not exist and open it for writing, or overwrite if it exists.
- 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.

Creating/Truncating Files: *creat()* System Call

- The *creat()* opens a file for writing, if it exists, setting its length to 0, or if it does not exist, to create it. Man page for the *open()* system call shows the usage:

```
#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);

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

Writing to Files: The write() System Call

- Used for writing sequences of bytes to the file specied by a given file descriptor:

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

A First Attempt at *cp*

- The structure of the *cp* command is:

1. open the source file for reading
2. open the copy file for writing
3. while a read of data from the source file to a buffer is not an empty read
4. write the data from the buffer to the copy file
5. close the source file
6. close the copy file

A First Attempt at *cp*

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

#define BUFFERSIZE 4096
#define COPYMODE 0644

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

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);
      }
      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(targe t_fd) == -1)
            die("Error closing files", "");

      return 0;
}
```

Comments

- The *buffer* declared an array of *BUFFERSIZE* chars.
- The *die()* encapsulates error handling logic and calls the *perror()*.
- Every call to API functions checked for a possible error.
- The code works correctly.
- But does it run fast? How long will it take to copy a very large (>100MB) file?

Timing Programs

- The *time* command measures the amount of time (and other resources) that a command uses.
- The simplest form is:

  - *$ time -p command*

- The '-p' option displays the traditional POSIX output, which consists of three values:
  1. Overall elapsed clock time, real
  2. User mode elapsed clock time, user
  3. Kernel mode (System) elapsed clock time, sys

Buffering and its Impact on Performance (cp)

Buffer Size (Bytes)	Real (sec.)	User (sec.)	Sys (sec.)
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

System Call 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.

System Call Overhead

- 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.

- When the call terminates, this kernel phase terminates and the user phase resumes. This is a form of context-switch.

System Call Overhead

- 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.

System Call Overhead

- 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.

System Buffering

- When a user process issues a read request from a disk, the kernel transfers the data from the disk to a buffer in kernel memory, and when all of the data transferred, it copies the buffer 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 transfers it to disk.

System Buffering

- The buffering scheme makes as if read/write operations read/write directly from/to the device and operations take place immediately.
- In fact, the kernel hides details from the user.

Memory Mapped I/O: *mmap(), munmap(), memcpy()*

- A process can request that a file be mapped to a set of virtual memory addresses.
- Read/writes to those addresses are, in effect, reads/writes to the file.
- *mmap()* maps an entire input file to a region of memory.
- *munmap()* undoes a mapping.
- *memcpy()* does single memory to memory copy.

Memory Mapped I/O: *mmap(), munmap(), memcpy()*

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

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

int munmap(void *addr, size_t length);

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

Memory Mapped I/O: *mmap(), munmap(), memcpy()*

- The first argument, *addr* is the starting address for the new mapping. If *addr* is NULL, kernel chooses the address. It is best to always use NULL as the first argument (portability).
- The second argument, *length* is the length in bytes of the mapping.
- The third argument describes the memory protection of the mapping. The possible values are:

1. *PROT_EXEC* Pages may be executed.
2. *PROT_READ* Pages may be read.
3. *PROT_WRITE* Pages may be written.
4. *PROT_NONE* Pages may not be accessed.

- The address of the new mapping is returned as the result of the call.

Memory Mapped I/O: *mmap(), munmap(), memcpy()*

- 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. The following flags are available:

  - *MAP_SHARED* Share this mapping. The file may not actually be updated until msync() or munmap() is called.
  - *MAP_PRIVATE* Create a private copy-on-write mapping. 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.

Second Attempt at *cp*

```
#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;

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

  if ((in_fd = open(argv[1], O_RDONLY)) == -1)
    die("Cannot open ", argv[1]);
  if ((out_fd = open (argv[2], O_RDWR | O_CREAT | O_TRUNC, COPYMODE)) == -1)
    die( "Cannot create", argv[2]);

  // get the sizeof the source file by seeking to the end of it:
  if ((filesize = lseek(in_fd, 0, SEEK_END)) == -1)
    die( "Could not seek to end of file ", argv[1]);

  lseek(out_fd, filesize -1, SEEK_SET);

  //So we write the NULL byte and filesize 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);

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

  close(in_fd); //close the files
  close(out_fd);
  return 0;
}
```

Returning to *who*

- Previous implementations of *who* read one *utmp* record at a time.
- Each read requires a system call, although a single *utmp* record is quite small and there are many of them; which is inefficient.
- *who* can benefit increasing the buffer size, just as the *cp* command.
- Modify *who* so that it reads several *utmp* records at a time and stores them in an internal array

Thanks…