Chapter Two:
Users, Files, and the Manual

Objectives

Ideas and Skills

- The role and use of online documentation
- The Unix file interface: open, read, write, lseek, close
- Reading, creating, and writing files
- File descriptors
- Buffering: user level, and kernel level

Ideas and Skills

- Kernel mode, user mode, and the cost of system calls
- How Unix represents time, how to format Unix time
- Using the utmp file to find the list of current users
- Detecting and reporting errors in system calls

System Calls and Functions

- Open, read, write, creat, lseek, close
- perror

Commands

- man
- who
- cp
- login

The who command

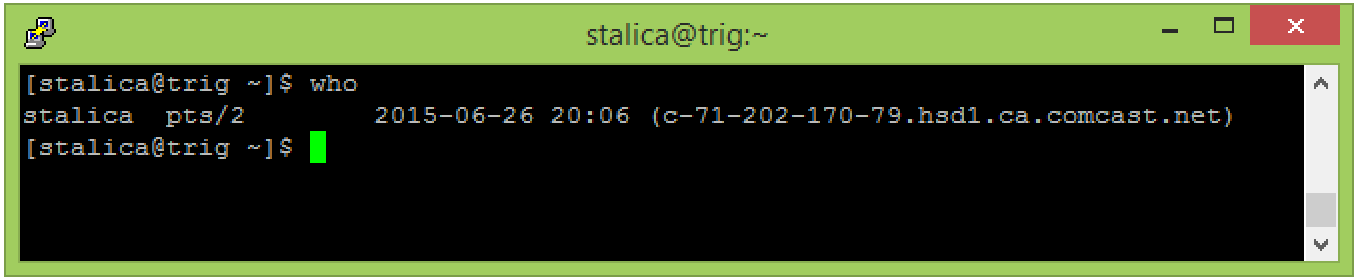
- To learn to use Unix to process files, we'll study the who command.
- We will ask:
  1. What does who do?
  2. How does who work?
  3. Can I write who?

The who command

- Unix commands are things you type at the prompt of a Unix system to do things: ls, cat, who, cp, mkdir, etc
- Commands are just programs written in C
- To add a new command to a Unix system, write a new program and place the executable in a standard system directory such as /bin, /usr/bin, or /usr/local/bin

Question 1: what does who do?

- Tells us who is currently using the system:



Question 1: what does who do?

- For more info, check the man page:

stalica@trig:~

```
WHO(1)                          User Commands                          WHO(1)

NAME
       who - show who is logged on

SYNOPSIS
       who [OPTION]... [ FILE | ARG1 ARG2 ]

DESCRIPTION
       -a, --all
              same as -b -d --login -p -r -t -T -u

       -b, --boot
              time of last system boot

       -d, --dead
              print dead processes

       -H, --heading
              print line of column headings

:
```

Question 1: what does who do?

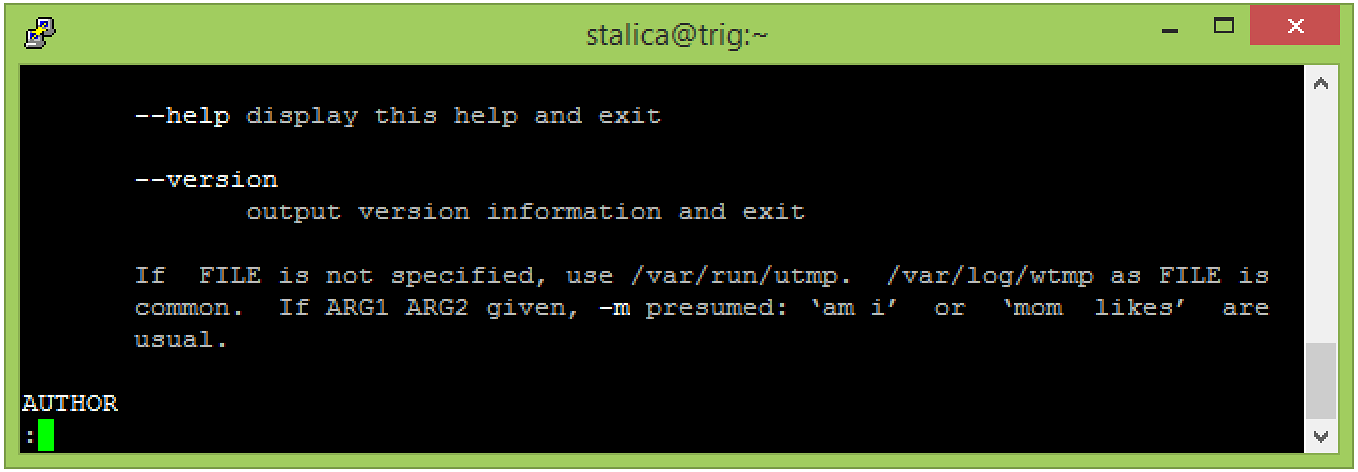
- All man pages share the same format
- Top line tells the name of the command and it's section
- Name section contains command name and a brief description.
- Synopsis section shows how to use the command.
- Square brackets [] indicate optional arguments
- Description section details what the command does.

Question 2: How does who do it?

- Let's ask Unix.
- Learn about any Unix command by:
  1. Read the manual
  2. Search the manual
  3. Read the .h files
  4. follow SEE ALSO links
  5. Google it

Question 2: How does who do it?

- From the description section of the who man page, we can see:



```
       --help display this help and exit

       --version
              output version information and exit

       If  FILE is not specified, use /var/run/utmp.  /var/log/wtmp as FILE is
       common.  If ARG1 ARG2 given, -m presumed: 'am i'  or  'mom  likes'  are
       usual.

AUTHOR
:
```

- Interesting, it mentions /var/run/utmp. Let's look for that

Question 2: How does who do it?

- Using man -k utmp, we're searching for keyword utmp:

```
[stalica@trig ~]$ man -k utmp
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
libutempter         (rpm) - A privileged helper for utmp/wtmp updates
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
[stalica@trig ~]$
```

- Hmm, utmp (5) – login records, let's look at that

Question 2: How does who do it?

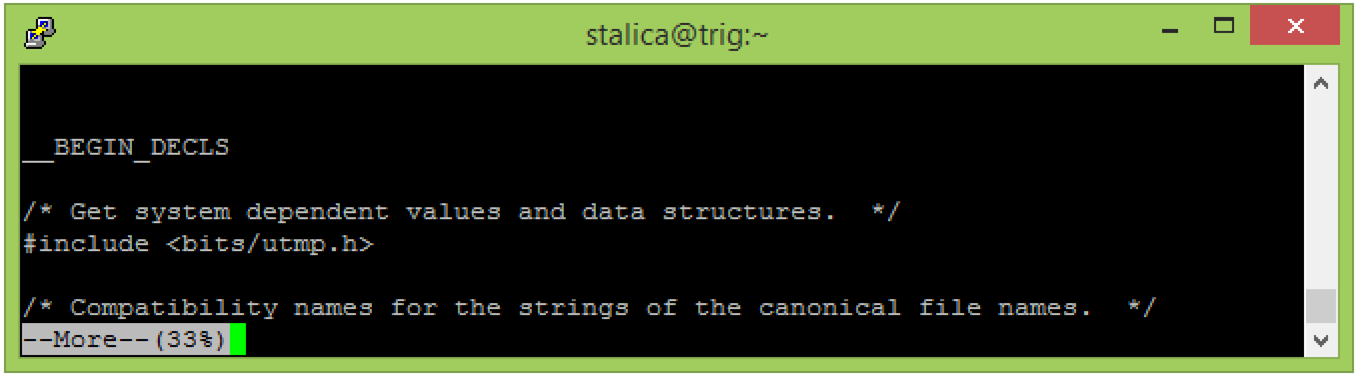
- Let's type, man 5 utmp

```
stalica@trig:~

UTMP(5)                   Linux Programmer's Manual                  UTMP(5)

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.
:
```

- Bingo! Who must use utmp, and it's found in utmp.h

Question 2: How does who do it?

- Let's read the .h file: more /usr/include/utmp.h:



- I scroll down and see this, so let's look in there....

Question 2: How does who do it?

- more /usr/include/bits/utmp.h

stalica@trig:~

```
/* 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.  */
/* The ut_session and ut_tv fields must be the same size when compiled
   32- and 64-bit.  This allows data files and shared memory to be
   shared between 32- and 64-bit applications.  */
--More--(59%)
```

- Awesome.   Found it. Now we know how who works.

Question 3: Can we write it?

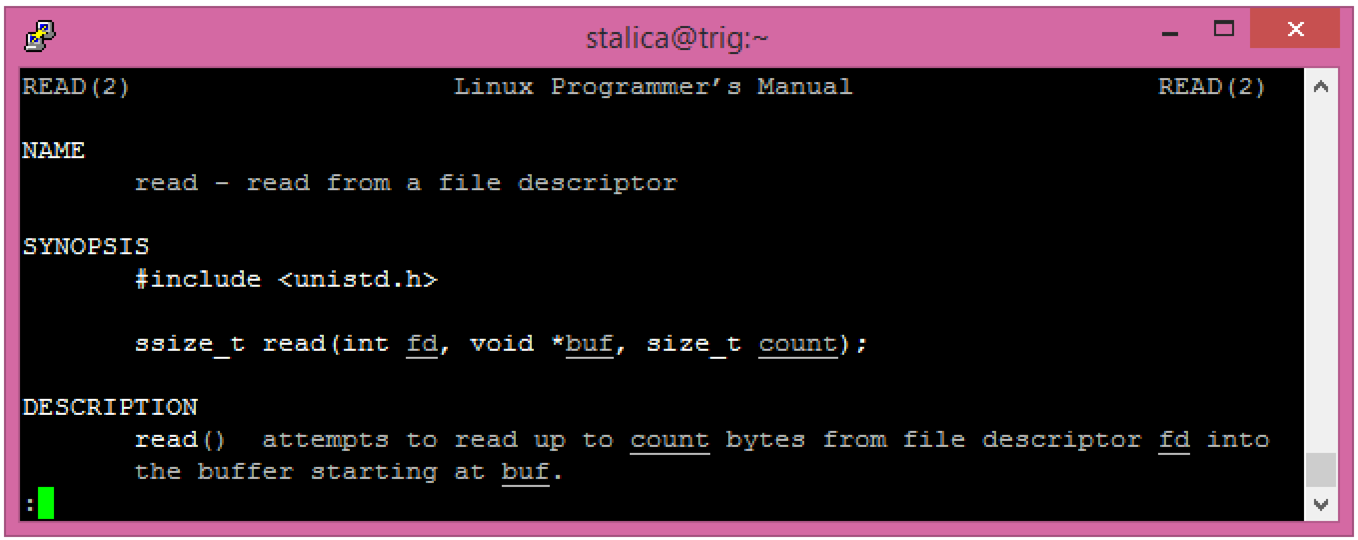
- We need to do a couple things: read structs from a file, display the information in the struct.
- How do we read structs from a file?
- Read the manual: man -k file shows us a LOT of stuff.
- Let's narrow it down, man -k file | grep read

Question 3: Can we write it?

- Read looks promising...

```
stalica@trig:~
pppdump              (8)  - convert PPP record file to readable format
pread                (2)  - read from or write to a file descriptor at a given o
ffset
pread [read]         (3p)  - read from a file
pwrite [pread]       (2)  - read from or write to a file descriptor at a given o
ffset
read                 (2)  - read from a file descriptor
read                 (3p)  - read from a file
readahead            (2)  - perform file readahead into page cache
readahead          (rpm) - Read a preset list of files into memory.
readelf              (1)  - Displays information about ELF files
reader.conf [reader] (5)  - configuration file for pcscd readers' drivers
readlinkat           (2)  - read value of a symbolic link relative to a director
y file descriptor
```

Question 3: Can we write it?



```
READ(2)                    Linux Programmer's Manual                    READ(2)

NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read()  attempts to read up to count bytes from file descriptor fd into
       the buffer starting at buf.
:
```

- So, read is a system call that lets us read n bytes from a file. We can use sizeof to help us out.

Question 3: Can we write it?

- Later on in the man page, system call open is referenced, and open's man page references the system call close.
- We have all we need to do the job.

Question 3: Can we write it?

- Basic usage of the open system call:

	open	
PURPOSE:	Creates a connection to a file	
INCLUDE:	#include <fcntl.h>	
USAGE:	int fd = open(char* name, int how)	
ARGS:	name	name of file
	how	O_RDONLY, O_WRONLY, or O_RDWR
RETURNS:	-1	on error
	int	on success

Question 3: Can we write it?

- Opening a file is a kernel service.
- Unix allows several processes to open the same file.
- If the file opens succesfully, the kernel returns a small positive integer known as a ***file descriptor*** which is used to identify the connection.
- The file descriptor is used for all operations involving the file.

Question 3: Can we write it?

- We read from the file using the file descriptor.

read		
PURPOSE	Transfer up to qty bytes from fd to buf	
INCLUDE	#include <unistd.h>	
USAGE	ssize_t numread = read(int fd, void* buf, ssize_t qty)	
ARGS	fd	source of data
	buf	destination of data
	qty	number of bytes to transfer
RETURNS	-1	on error
	numread	on success

Question 3: Can we write it?

- Need to close a file when we're done with it, here's the call:

close	
PURPOSE	Close a file
INCLUDE	#include <unistd.h>
USAGE	int result = close(int fd)
ARGS	fd file descriptor
RETURNS	-1 on error 0 on success

Question 3: Can we write it?

- Let's walk through who1.c, the first version, paying specific attention to the show_info function.
- How closely does who1.c match the real who?
- Not close enough, we need to suppress blank records and get log-in times correct.

Question 3: Can we write it?

- Doing some further digging around in utmp.h, we find:

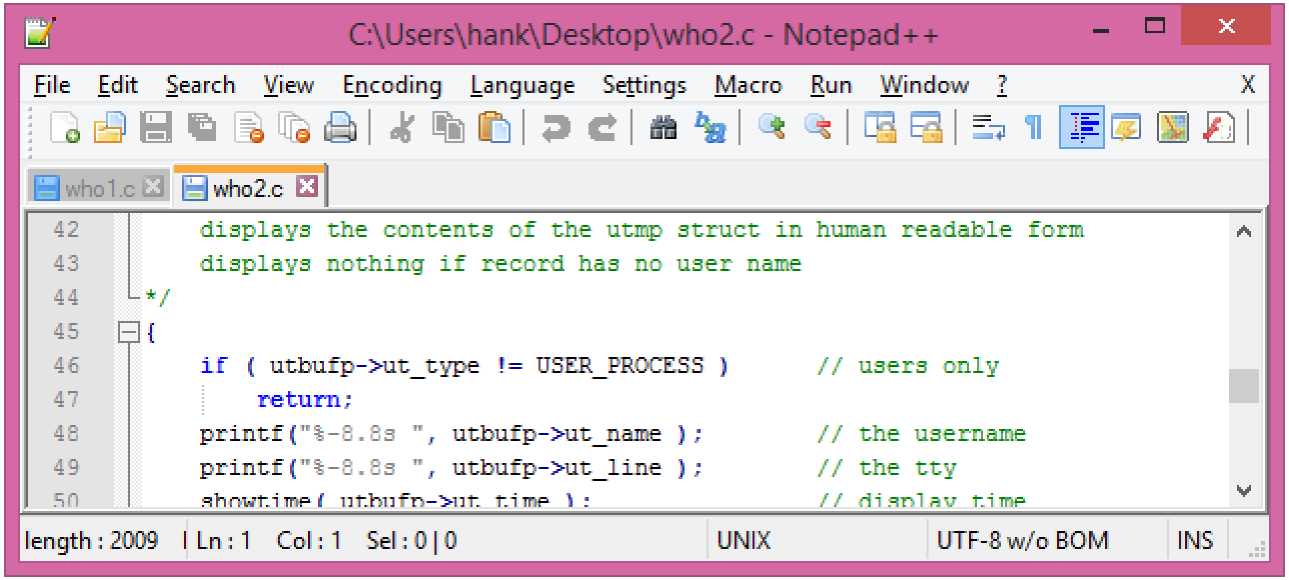
stalica@trig:~

```
/* Values for the `ut_type' field of a `struct utmp'.  */
#define EMPTY           0       /* No valid user accounting information.  */

#define RUN_LVL         1       /* The system's runlevel.  */
#define BOOT_TIME       2       /* Time of system boot.  */
#define NEW_TIME        3       /* Time after system clock changed.  */
#define OLD_TIME        4       /* Time when system clock changed.  */

#define INIT_PROCESS    5       /* Process spawned by the init process.  */
#define LOGIN_PROCESS   6       /* Session leader of a logged in user.  */
#define USER_PROCESS    7       /* Normal process.  */
#define DEAD_PROCESS    8       /* Terminated process.  */
--More--(92%)
```

- Type 7 looks like something we can use

Question 3: Can we write it?

- So let's make fix one:



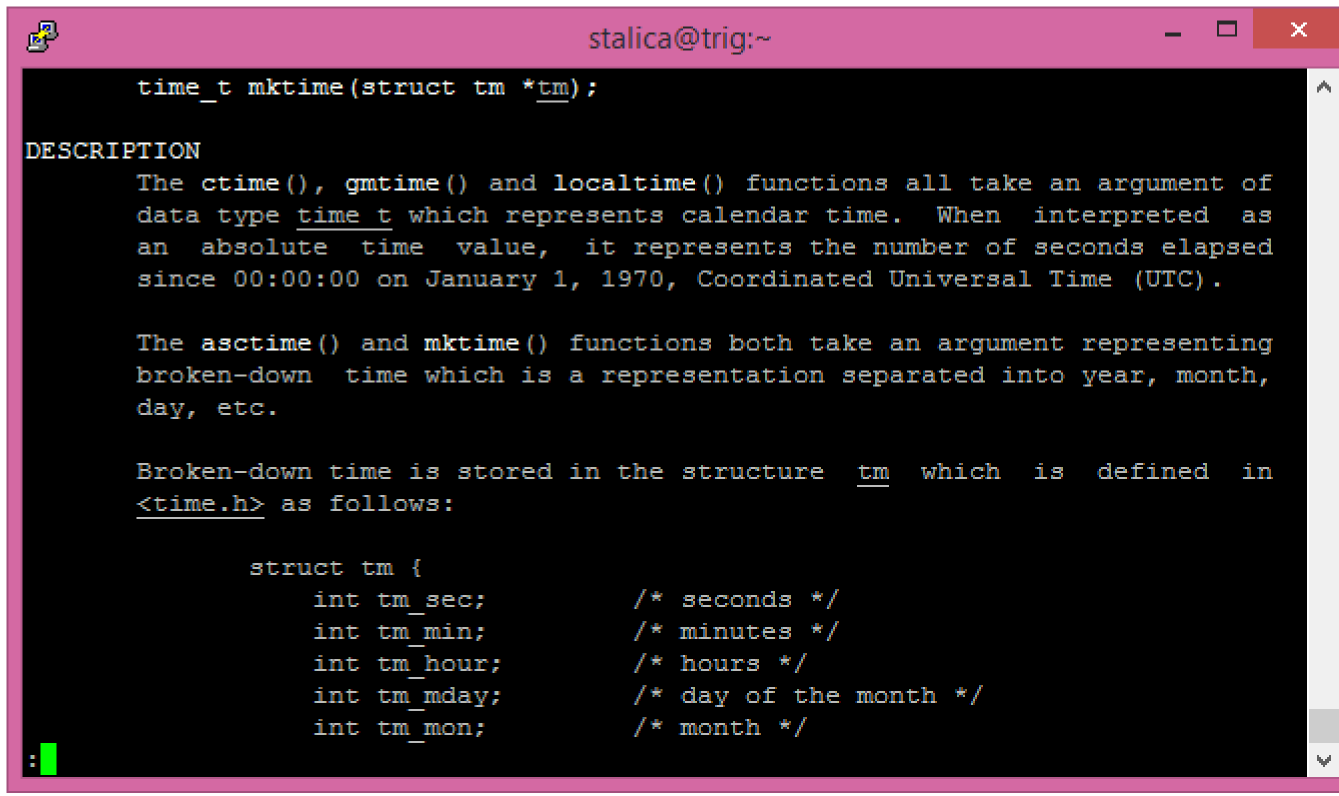
- Lines 46, 47 fix this problem.

Question 3: Can we write it?

- The next problem is fixing that time display
  man -k time which leads us to
- /usr/include/time.h has information we can use
- Unix has a data type it uses to store time, time_t.
- It's an integer that stores all the seconds since midnight, Jan 1, 1970.
- ut_time in utmp represent the login-time as seconds since the beginning of *The Epoch*.

Question 3: Can we write it?

- Need to convert that to something readable. Let's use a function to do that: ctime
- Man 3 ctime:

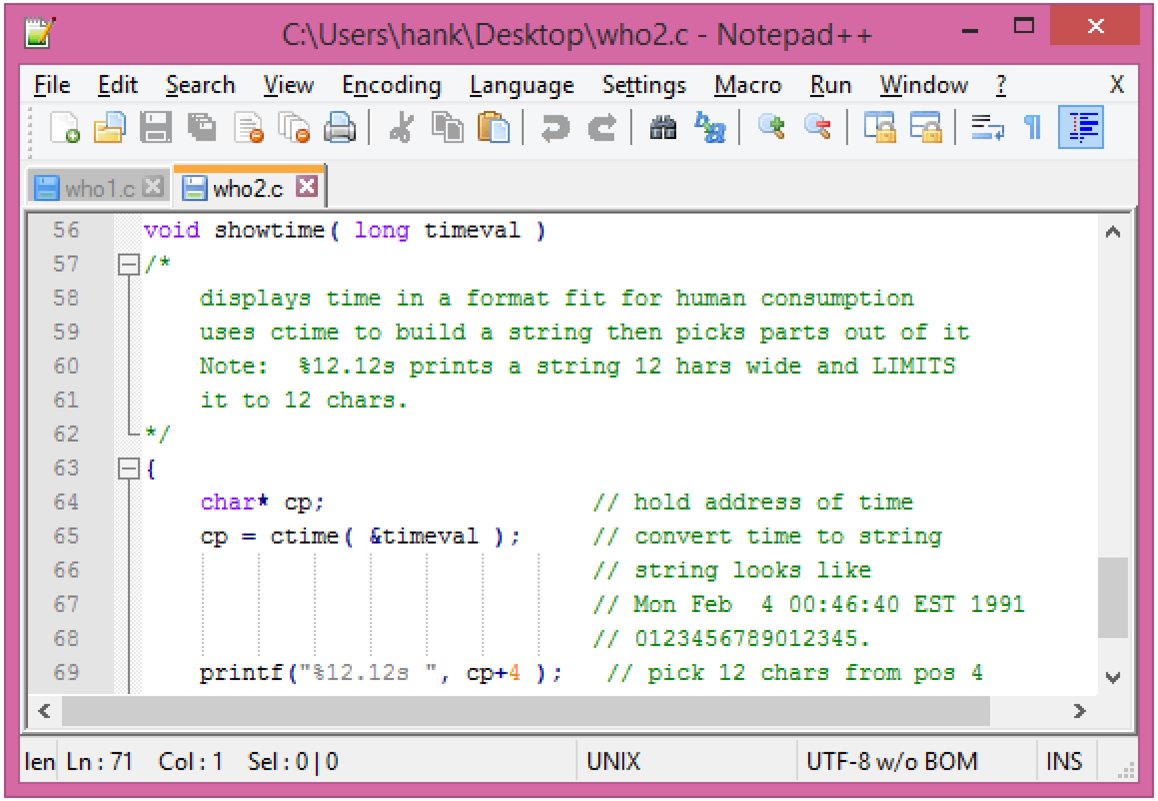


```
stalica@trig:~

       time_t mktime(struct tm *tm);

DESCRIPTION
       The ctime(), gmtime() and localtime() functions all take an argument of
       data type time_t which represents calendar time.  When  interpreted  as
       an  absolute  time  value,  it represents the number of seconds elapsed
       since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC).

       The asctime() and mktime() functions both take an argument representing
       broken-down  time which is a representation separated into year, month,
       day, etc.

       Broken-down time is stored in the structure  tm  which  is  defined  in
       <time.h> as follows:

              struct tm {
                  int tm_sec;         /* seconds */
                  int tm_min;         /* minutes */
                  int tm_hour;        /* hours */
                  int tm_mday;        /* day of the month */
                  int tm_mon;         /* month */
:
```

Question 3: Can we write it?

- Looks like just what we need. The function takes a time_t pointer and returns a pointer to a string that looks something like: Wed Jun 30 21:49:08 1993\n
- Let's put it all together and look at who2.c, our finished version.

Question 3: Can we write it?



Project Two:
Writing cp (read and write)

Writing cp

- Let's follow a similar procedure, and write our own cp command.
- Remember the approach,
  1. What does cp do?
  2. How does cp Create and Write?
  3. Can we write our own cp?

What does cp do?

- The cp system command makes a copy of a file.
- Typical usage: cp source-file   target-file
- If there is no target file, cp creates it. If there is, cp replaces it.

How does cp do it?

- We can create or rewrite a file using the creat system call.

	creat		
PURPOSE	Create or zero a file		
INCLUDE	#include <fcntl.h>		
USAGE	int fd = creat(char* fn, mode_t mode)		
ARGS	fn:	the name of the file	
	mode:	access permission	
RETURNS	-1	on error	
	fd	on success	

How does cp do it?

- An example,
  fd = creat("addressbook“, 0644);
- This creates or truncates a file named addressbook.
- If the file doesn't exist, the permissions are set to
  rw--r--r--   which we'll explain in Chapter 3.
- fd represents a file open for writing only

How does cp do it?

- Let's send data to an open file using the write system call:

write		
PURPOSE	send data from memory to a file	
INCLUDE	#include <unistd.h>	
USAGE	ssize_t result = write(int fd, void* buf, size_t amt)	
ARGS	fd	a file descriptor
	buf	an array
	amt	how many bytes to write
RETURNS	-1	on error
	num bytes written	on success

How does cp do it?

- So write copies data from process memory to a file.
- If there's a problem, the kernel returns -1.
- Otherwise, returns number of bytes written.

Can we write cp?

- Here's a general outline for cp:

```
open sourcefile for reading
open copyfile for writing
while not end of file do
    read from sourcefile to buffer
    write from buffer to copyfile
close sourcefile
close copyfile
```

Can we write cp?

- Consider what's going to happen:
- The files are on the disk.
- Process is living in user space.
- A buffer is a chunk of memory in the process.
- Process has two file descriptors.
- Bytes are read from the original file into the buffer then to the copy
- Let's look at an implementation, cp1.c. (see sample code)

More efficient file i/o: buffering

- Let's ask a new question: *How can I make this run better?*
- cp1 contains a symbol BUFFERSIZE.
- Defines size of the array that holds the bytes being read/written.
- Does the size of the buffer matter?

Does buffer size matter?

- Absolutely. Consider copying a file 2500 bytes long. Then,

  Filesize = 2500 bytes

  If buffer = 100 bytes, then
  copy requires 25 read() and 25 write() calls

  If buffer = 1000 bytes, then
  copy requires 3 read() and 3 write() calls

Does buffer size matter?

- Going from 100 → 1000 means we go from 50 calls to 6.
- HUGE difference.
- System calls take time. Programs that make a lot of them run slower and take cpu time away from other users.

Why System Calls Consume Time

- Consider the cp1 program.
- In our system's memory, the cp1 process is in *user space*, and the kernel is in *system space* : two distinct chunks of memory.
- cp1 wants to do a read, but the code to do the reading is actually in the kernel. So, control jumps from the code in cp1 in user space to the kernel code in system space.
- CPU runs the code, this takes time to transfer the data.

Why System Calls Consume Time

- Additionally, jumping into and out of the kernel takes time.
- When the kernel code is running, the CPU is running in *supervisor* mode with it's own stack and memory.
- When user code is running, the CPU is running in *user* mode, with it's own stack and memory.
- The kernel needs special access to resources that user code must not have.
- The more system calls that are made, the more time is spent switching between these modes. This is *expensive*.

Does this mean who2.c is inefficent?

- Yes! One system call for reading each utmp record makes as much sense as buying a carton of eggs, one egg at a time.
- Better idea: read a bunch of records at once, then process the records in local storage one by one.
- Consider that egg carton, and some pseudocode:

Does this mean who2.c is inefficent?

```
getegg() {
  if ( eggs_left_in_carton == 0 )  {
        refill carton at store
        if ( eggs_at_store == 0 )
                return EndOfEggs
        eggs_left_in_carton = 12
   }
   eggs_left_in_carton--:
   return one egg;
}
```

Does this mean who2.c is inefficent?

- Each call to getegg fetches one egg, but not from the store
- Only when the carton is empty does the function go to the store.
- If you look at a lot of Unix functions, they are implemented using similar logic.
- Adding buffering to who2.c is left as an exercise.

Buffering and the Kernel

- If buffering is so great, why doesn't the kernel do it? *It does.*
- Switching between modes takes time, but reading disks takes forever in comparison.
- To save time, the kernel keeps copies of disk blocks in memory.
- Disks are collection of blocks of data, similar to how the utmp file is a collection of log-in records.
- Kernel copies blocks from the disk into kernel buffers.

Buffering and the Kernel

- When a process requests file data, the kernel copies it from it's own buffer to the process buffer, not from the disk to user space.
- What happens if the data isn't in the kernel buffer?
- Kernel adds it to the shopping list, and suspends the requesting process.
- Kernel lets other processes run in the mean time.

Buffering and the Kernel

- Later, the kernel moves the requested data from disk into the kernel buffer, and can then copy the data into the buffer in user space and wake the sleeping requesting process.
- So, read() causes the kernel to copy data from kernel buffer into the process buffer. write() causes data to be copied from the process to a kernel buffer.
- The kernel is then free to copy data to disk when it gets around to it.

Buffering and the Kernel

- It's possible those kernel buffers don't get written to disk.
- Make sure you shut down your computer correctly!
- Consequences of Kernel Buffering:

  Faster "disk" I/O
  Optimized disk writes
  Need to write buffers to disk before shutdown

Reading and Writing a File

- Recall the read/write pointers from C++? Unix has a system call that mimics this behavior, lseek.
- lseek allows us to set the file pointer to a specific offset within the file.
- We can then read from that offset, or overwrite data beginning at that offset.
- Let's look at the call:

Reading and Writing a File

- **lseek**

PURPOSE	set file pointer to specified offset in file
INCLUDE	#include <sys/types.h> #include <unistd.h>
USAGE	off_t oldpos = lseek(int fd, off_t dist, int base)
ARGS	fd: file descriptor dist: a distance in bytes base: SEEK_SET ==> start of file SEEK_CUR ==> current position SEEK_END ==> end of file
RETURNS	-1 on error or the previous position in the file

Reading and Writing a File

- lseek sets the current position for open file *fd* to the position defined by the pair *dist* and *base*.
- Example:

  ```
  lseek( fd, -(sizeof( struct utmp) ), SEEK_CUR);
  ```

  That's gonna move the current position a distance sizeof(struct utmp) bytes *before* the current position.
- Notice: `lseek( fd, 0, SEEK_CUR)` returns the current position.

Reading and Writing a File

- Let's see an example, lseek.c.

What To Do with System-Call Errors

errno

- Every system call has it's own set of errors.
- Consider open(). The file might not exist, you might not have permissions, you might have too many files open.
- How to identify what went wrong: errno
- The kernel will tell your program the error cause by storing an error code in a global variable called errno.
- Every program can access this variable.

errno

- The manpage for errno(3) include error-code symbols:

stalica@trig:~

```
       E2BIG  Argument list too long (POSIX.1)

       EACCES Permission denied (POSIX.1)

       EADDRINUSE
              Address already in use (POSIX.1)

       EADDRNOTAVAIL
              Address not available (POSIX.1)

       EAFNOSUPPORT
              Address family not supported (POSIX.1)
:
```

errno

- You can use these symbols to figure out what went wrong.
- For example:

```c
#include <errno.h>
#include <fcntl.h>

int main()
{
    int fd = open( "file", O_RDONLY );

    if ( fd == -1 )
    {
        int errsv = errno;

        if ( errsv == ENOENT )
            printf("No such file\n");
        else if ( errsv == EACCESS )
            printf("You do not have permission to open file\n");
    }
```

errno

- To access errno on older systems, an extern call is required:
```
extern int errno;
```
- On modern systems and versions of the C library, this doesn't work. Instead, include a header file:
```
#include <errno.h>
```
- Be sure to save the error message before making another system call! Otherwise, you risk overwriting the previous error number.

Reporting Errors: perror(3)

- You can print an error message describing the error doing as above.
- Often, it's easier to use a built-in function perror(string).
- This function looks up the error code and prints, to standard error, the string you pass it and a descriptive message.

Reporting Errors: perror(3)

- Here's a revised version of the previous sample:

```c
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>

int main()
{
    int fd = open( "file", O_RDONLY );

    if ( fd == -1 )
    {
        perror("Can't open file\n");
        exit(1);
    }
```

Summary

- who command reports the list of current users by reading a system log file.
- Unix systems store data in files. Unix manipulates this data using six system calls:

  open(filename, how)
  creat(filename, mode)
  read(fd, buffer, amt)
  write(fd, buffer, amt)
  lseek(fd, distance, base)
  close(fd)

Summary

- A process reads/writes data through *file descriptors* which identify a connection between a process and a file.
- Each time a process makes a system call, the computer changes from user mode to kernel mode and executes kernel code. Programs are more efficient when this is minimized.
- Programs can reduce the number of system calls when reading/writing data by using buffers and calling the kernel when they are full/empty.

Summary

- Unix kernel uses buffers inside kernel memory to minimize data transfers.
- Unix stores times as the number of seconds since beginning of Unix time.
- When Unix system calls have an error, the system sets errno to an error code and returns the value -1. That error code can be used to diagnose the error and take action.
- Use man pages to find out about commands.
- Header files contain definitions of structures, symbolic constant values, and function prototypes used to build system tools.