1730 Study Guide
Exam 1

Welcome! Please contribute where you can and feel free to delete anything and rewrite it for ease of reading.

Topics:

Exam Specific Info

UNIX

Pointers

Arrays

Permissions

Makefiles

Debugging

System Call Mechanics

File I/O

Files

Directories

Exam Specific Info:

- Wednesday Exam(09/24): Should take 20 minutes, to be turned in on a piece of paper with output.
    - Possible topics:
        - Links
        - Open/Close
        - Read
        - chdir()

- Thursday Exam(09/25): Should take 1hr, 40 multiple choice, 10 short answer, total 143 possible points; paper exam with answer bubble sheet(like scantron); Possible partial credit on some multiple choice if you justify your answer.
    - Will cover: Book Chapter 1-6
        - Permissions
        - Read in
        - Standard out
        - Effective UID
        - Directory Structure

This website contains these and many more questions for practice.

Example Questions (1pt each):

- The part of the Unix operating system that interacts with the hardware is called:
    - GNU project
    - **The kernel**
    - The shell
    - Linux
- The operating system controls
    - The hard drive
    - The processor
    - printers
    - **all of the above**
    - none of the above
- A file with the permission status of rwxr-xr-- indicates:
    - The owner has all permissions, the group has only Read permissions
    - The owner has only Read and Execute permissions
    - **The owner has all permissions, the group has only Read and Execute permissions**
    - The group has all permissions, the owner has only Read and Execute permissions
- What is the redirection symbol for error?
    - **2>**

- What is the function of an effective UID (User ID)?
  - Each process in Unix has both a user ID and a group ID associated with it. When a user tries to interact with a file, for example open the file and write to it, the IDs associated with the file will determine if this user can perform those open and write actions. **These IDs constitute the *effective* privilege of the process, because they determine what a process can and cannot do.** Most of the time, these will be referred to as the effective uid and gid (effective user ID and group ID)
  - For example, the Effective UID of the passwd utility is set to 0 (root). Thus, this process can modify the /etc/passwd file, only accessible to root users.
    - Obviously, as a normal user, you're only allowed to modify the password of your own account. But how does the process know this? The passwd command is a root process, which means it can modify the password of ANY user on the system, right?
      - Right, but this is where another pair of IDs come into play: **Real UID and GID.** These IDs are used solely to track who a caller of a certain command *really* is.
        - The passwd command reads in your real uid and changes the passwd of *that specific* uid, nobody elses.
  - Here's where things get interesting: let's say you log onto Nike and your uid is 500. Let's say you invoke a root application (passwd, maybe?. The OS will set the e-uid of the process to root (0) but the real uid (500) remains unchanged. This is so that the application or process knows who called the function, and continue the process as a normal user (id 500).
    - Think of a real UID as a name badge, and an effective UID is the set of keys you've been given

UNIX

- **operating system**: software that controls the hardware resources of the computer and provides an environment under which programs can run; generally called the kernel since it is small and resides at the core of the environment.

- **system calls**: layer of software that serves as the interface to the kernel. This is how a program requests a service from an operating system's kernel. Interface between processes and operating system(kernel)

- **shell**: command-line interpreter that reads user input and executes commands; user input to a shell is normally from the terminal (an interactive shell) or sometimes from a file (called a shell script)
- - Boune(sh), Bourne-again(bash), C-shell(csh), Korn(ksh), TENEX C(tsch)

- **library**: a collection of related objects files grouped together.

- **directory**: file that contains directory entries
    - Two filenames are automatically created whenever a new directory is created:
        - .(called dot) refers to the current directory
        - .. (called dot-dot) refers to the parent directory.
            - In the root directory, dot-dot is the same as dot

- **file descriptors**: small non-negative integers that the kernel uses to identify the files accessed by a process. Whenever it opens an existing file or creates a new file, the kernel returns a file descriptor that we use when we want to read or write the file.
    - (read(2), write(2), lseek(2), fcntl(2), etc.)

- **program**:executable file residing on disk in a directory that is read into memory and is executed by the kernel

- **process/task**: executing instance of a program

- **environment variables**: set of variables the shell uses for certain operations; name and value; env command lists all current environment variables. *unset "varName"* will remove an environment variable. Make an environment variable with *varName(usually in all caps)=whatever you want*. Be sure to have no space before and after the '='. You can see this variable with *echo $VARIABLE*
    - This video (5 minutes) may help understand

- **alias**: shorthand for frequently used commands
    - alias ll="ls -al"
        - Now when you type ll, it'll perform the action of 'ls -al'

    - alias cc=”gcc -g chunk.c -o chunk”
        - or alias m=”make” (if a makefile is present)
    - View all currently set aliases by using the ‘alias’ command

- **signals**: technique used to notify a process that some condition has occurred
    - Ignore the signal
    - Let the default action occur
    - Provide a function that is called when the signal occurs

- **pipes**: a way to send the output of one command to the input of the next command;
    - | is the pipe character

- **filter**: a program that take input and transforms it in some way
    - wc: gives count of lines/word/char
        - wc -w gives words only
        - wc -l is lines only
        - wc -c is chars only
    - grep: searches for lines with a given string
    - more: view one screen at a time
        - A lot of times, the pipe is used with this, or the ‘less’ command
            - ls -al | less
    - sort: sorts lines numerically or alphabetically

- **Error handling**: When an error occurs in one of the UNIX System functions, a negative value is often returned, and the integer errno is usually set to a value that tells why. The file <errno.h> defines the symbol errno and constants for each value that errno can assume.
    - errno’s value is never cleared by a routine if an error does not occur.
    - The value of errno is never set to 0 by any of the functions, and none of the constants defined in <errno.h> has a value of 0.
    - A fatal error has no recovery action. The best we can do is print an error message on the user ’s screen or to a log file, and then exit.
    - The typical recovery action for a resource-related nonfatal error is to delay and retry later.

- The user ID from our entry in the password file is a numeric value that identifies us to the system.
    - Assigned by the system administrator when our login name is assigned
    - Cannot change
    - Unique for every user

- Groups are normally used to collect users together into projects or departments. This allows the sharing of resources, such as files, among members of the same group.

- UNIX System maintains three values for a process:
    - Clock time [wall clock time]
    - User CPU time
    - System CPU time
    - The primitive system data type clock_t holds these time values.

- Differences between system calls and library functions:
    - both exist to provide services for application programs but only lib functions can be replaced
        - malloc() manages memory allocation but the implementation can be altered if the user needs something different. The sbrk system call will be used in that case.
        - the system call in the kernel allocates some amount of space for the process whereas malloc() manages this space from the user level
    - many library functions invoke a system call
    - system calls provide minimal interface whereas library functions have elaborate functionality
    - stdin, stdout, stderr are all automatically opened with the shell.
        - Redirecting stdout
            - [command] > [filename] will write output of command to filename
            - [command] >> [filename] will append output of command to filename
        - Redirecting stderr
            - 2> will tell program to redirect stderr to somewhere else, ie a file
        - Redirecting stdin
            - < will tell program to read from another file instead of the terminal

I see people have edited stuff on UNIX :) thanks so much for organizing and adding!
-Sharon

Pointers

- Pointers are designed for storing memory address (address of another variable); Declaring a pointer is the same as declaring a normal variable except you stick an asterisk * in front of the variables identifier
    - Syntax: int * pointer
        - //The asterisk can go close to the type: **int*** OR
          In between the type and variable: **int * pointer** OR
          close to the name of the variable **int *pointer**
    - There are two new operators to work with pointers:
        - The address of marker & and the dereferencing operator *
            - When you place the & mark in front of a variable you will get its address; This can be stored in a pointer variable
            - The asterisk * in front of a pointer will give the value at the memory address it is pointed to
    - EXAMPLE of pointers:

```
#include <stdio.h>

int main()
{
  int my_variable = 6, other_variable = 10;
  int *my_pointer;

  printf("the address of my_variable is    : %p\n", &my_variable);
  printf("the address of other_variable is : %p\n", &other_variable);

  my_pointer = &my_variable;

  printf("\nafter \"my_pointer = &my_variable\":\n");
  printf("\tthe value of my_pointer is %p\n", my_pointer);
  printf("\tthe value at that address is %d\n", *my_pointer);

  my_pointer = &other_variable;

  printf("\nafter \"my_pointer = &other_variable\":\n");
  printf("\tthe value of my_pointer is %p\n", my_pointer);
  printf("\tthe value at that address is %d\n", *my_pointer);

  return 0;
}
```

This will produce following result.

```
the address of my_variable is    : 0xbfffdac4
the address of other_variable is : 0xbfffdac0

after "my_pointer = &my_variable":
        the value of my_pointer is 0xbfffdac4
        the value at that address is 6

after "my_pointer = &other_variable":
        the value of my_pointer is 0xbfffdac0
        the value at that address is 10
```

- Pointers And Arrays
    - Most frequent use of Pointers in C is for walking through an array
    - Syntax:
        - char *y;
          char x[100];

- y can point to an element in array x like so:
  - y = &x[0];
    y = x;
- using the increment you can go through the array
  - y = &x[0];
    y++;
  - this leaves y pointing at x[1].

- EXAMPLE of arrays working with pointers:

Using Pointer Arithmetic With Arrays:

Arrays occupy consecutive memory slots in the computer's memory. This is where pointer arithmetic comes in handy - if you create a pointer to the first element, incrementing it one step will make it point to the next element.

```
#include <stdio.h>

int main() {
  int *ptr;
  int arrayInts[10] = {1,2,3,4,5,6,7,8,9,10};

  ptr = arrayInts; /* ptr = &arrayInts[0]; is also fine */

  printf("The pointer is pointing to the first ");
  printf("array element, which is %d.\n", *ptr);
  printf("Let's increment it.....\n");

  ptr++;

  printf("Now it should point to the next element,");
  printf(" which is %d.\n", *ptr);
  printf("But suppose we point to the 3rd and 4th: %d %d.\n",
          *(ptr+1),*(ptr+2));

  ptr+=2;

  printf("Now skip the next 4 to point to the 8th: %d.\n",
          *(ptr+=4));

  ptr--;

  printf("Did I miss out my lucky number %d?!\n", *(ptr++));
  printf("Back to the 8th it is then..... %d.\n", *ptr);

  return 0;
}
```

This will produce following result:

```
The pointer is pointing to the first array element, which is 1.
Let's increment it.....
Now it should point to the next element, which is 2.
But suppose we point to the 3rd and 4th: 3 4.
Now skip the next 4 to point to the 8th: 8.
Did I miss out my lucky number 7?!
Back to the 8th it is then..... 8.
```

Arrays

Basic Array Information
Multidimensional Arrays
Arrays As Parameters
Arrays As Return Values
Array and Pointer Intermingling

If anyone has specific questions on arrays in C, ask away! I think some of us can try and answer :) -Sharon

Permissions

Permission Groups

Each file and directory has three user based permission groups:

- **owner** - The Owner permissions apply only the owner of the file or directory, they will not impact the actions of other users.
- **group** - The Group permissions apply only to the group that has been assigned to the file or directory, they will not affect the actions of other users.
- **other users** - The other Users permissions apply to all other users on the system, this is the permission group that you want to watch the most.

Permission Types

Each file or directory has three basic permission types:

- **read** - The Read permission refers to a user's capability to read the contents of the file. (If you have read permission on a directory, you can use ls to view the contents.)
- **write** - The Write permissions refer to a user's capability to write or modify a file or directory. (If you have write permissions on a directory, you can use rm and mkdir etc inside the directory.)
- **execute** - The Execute permission affects a user's capability to execute a file or view the contents of a directory. (If you have execute permissions on a directory, you can use cd to change your current working directory to inside that directory.)

The permission in the command line is displayed as: ***_rwxrwxrwx 1 owner:group***

1. User rights/Permissions
    1. The first character that is marked with an underscore is the special permission flag that can vary.
    2. The following set of three characters (rwx) is for the owner permissions.
    3. The second set of three characters (rwx) is for the Group permissions.
    4. The third set of three characters (rwx) is for the Other Users permissions.

Example Permissions:

Octal 1	777 -> rwxrwxrwx
0 = 000	555 -> r-xr-xr-x
1 = 001	755 -> rwxr-xr-x
2 = 010	700 -> rwx - - - - - -
3 = 011	037 -> - - - - wxrwx
4 = 100	
5 = 101	
6 = 110	
7 = 111	

a better way to calculate the permission.

4 ------- read,
2 ------- write,
1--------- execute
if you want to change the permission to write and execute,
just do chmod 333 because 2+1 = 3. that's the permission for write and execute

Makefiles

Check out this website. It explains the uses, syntax, and variables of a makefile and how it works. Good read if you don't understand the basics of the makefile.

- **note** that this guide is talking about c++ and uses the compiler g++ rather than the gcc we are familiar with however the rest should be pretty much the same.

If you do not feel like reading the whole thing (which i recommend because it is short and to the point unlike everything in our textbook…) here is a quick summary.

- when "make" is run the program will look for the file named Makefile
- With many different files that need to be compiled in a project it can be tedious typing out each file name that needs to be compiled. This is where makefile comes into play
- syntax:
    - *target: dependencies*
      [tab] system command
- example:
  all:
  (tab) gcc 'filename(s)' -o 'executableName'
- Here 'all' is the target which is followed by no dependencies
- all is the default target meaning that this target will be executed if no others are specified
- The above example will compile everything every time make is run
- In order to only compile the files that haven't been modified we use different targets and dependencies
- example:

```
all: hello

hello: main.o factorial.o hello.o
	g++ main.o factorial.o hello.o -o hello

main.o: main.cpp
	g++ -c main.cpp

factorial.o: factorial.cpp
	g++ -c factorial.cpp

hello.o: hello.cpp
	g++ -c hello.cpp

clean:
	rm -rf *o hello
```

- target 'all' has no system command now but a dependency. For 'make' to run, the called target ('all' in this case as it is default) must meet all its dependencies ('hello' is the only dependency here).
- This process is essentially repeated, moving from dependency to its associated target.

- Dependencies is where my knowledge of makefiles get fuzzy so feel free to edit/correct/add.
- **Variables** can be used to represent compilers and compiler options
- They go on the top of the makefile with syntax:
  - ‘variableName’=’whatever you want the variable to represent’
  - ex: CC=gcc
    - with this variable, CC can be used instead of gcc anywhere in the makefile
  - another example: CFLAGS=-c -Wall
    - This variable will represent the flags you want to use while compiling
- Syntax to use variables: ‘$(varName)’
- Example makefile using variables:

CC=gcc
CFLAGS=-c -Wall

all:
$(CC) $(CFLAGS) hello.c -o hello

- Variables can be in variables ex)

```
___________________________
|        FOO=bar          |
|        STUFF=$(FOO)     |
“”””””””””””””””””””””””””””””””””””””””””””””””””
```

Please edit anything I may have said that is wrong

Debugging

We use the GDB tool to debug our code.
In-Depth Look at GDB (Easy to Watch!)

System Call Mechanics

System calls allow for a layer of abstraction in the OS. They allow us to interface with the OS, which interfaces with the underlying mechanics, to perform certain actions. There are a plethora of system calls, such as read, write, lseek, etc etc. Typically, they're more primitive/harder to use or understand, but they are more efficient than system functions wrapped in library calls (fread, fwrite, etc).

For example, let's say we wanted to avoid system calls altogether. Let's say we wanted to write some bytes to the hard disk. Accessing a raw disk really involves specifying the data, the length of the data, the actual disk drive itself, the track location(s), and sector location(s), which is all done on a 150 mph spinning drive. A method needed to access all this would look something like:

```
write( block, len, device, track, sector );
```

Looks gross and overly complicated. This is where system calls swoop in to save the day. If I wanted to write some bytes to a file, all I have to do is specify a file via file descriptor, where the bytes are coming from, and how many bytes I should write. That is:

```
write( fd, buf, num_bytes);
```

Looks super easy and I still don't know how a hard drive works, nor do I need to know! System calls allow for this level of abstraction to be present so, as end users who don't have time to worry about sector locations of a file, don't need to worry about it.

File I/O (Ch 3)

File Descriptors [definition in UNIX portion] are returned when a new file is created or an existing file is opened. Reading and writing to a file requires the file descriptor as an input.

- 0 for stdin
- 1 stdout
- 2 stderr

`int open(path, flags, mode) → <stdlib.h> <fcntl.h>`

- path: string that gives absolute or relative pathname
- flags:
  - O_RDONLY
  - O_WRONLY
  - O_RDWR
  - O_CREAT
  - O_TRUNC
  - O_APPEND
- mode: specify perms is using O_CREAT
- returns filedescriptor int
- `openat()`
  - gives threads a way to use relative pathnames to open files in directories other than the current working directory
  - provides a way to avoid time-of-check-to-time-of-use (TOCTTOU) errors
    - a program is vulnerable if it makes two file-based function calls where the second call depends on the results of the first call
    - the file can change between the two calls, thereby invalidating the results of the first call, leading to a program error

`int creat(path, mode) → <fcntl.h>`

- path: string that gives absolute or relative pathname
- mode: specify perms
- opened only for writing
- returns filedescriptor int

`int close(fd) → <unistd.h>`

- Closing a file also releases any record locks that the process may have on the file
- many programs don't explicitly close open files

`off_t lseek(int fd, off_t offset, int whence) → <unistd.h>`

- current file offset: non-negative integer that measures the number of bytes from the beginning of the file.
- By default, this offset is initialized to 0 when a file is opened, unless the O_APPEND option is specified.
- `offset` depends on `whence`
  - `SEEK_SET:` file's offset set to `offset` bytes
  - `SEEK_CUR:` file's offset is set to its current value plus the `offset`
  - `SEEK_END:` file's offset is set to the size of the file plus the `offset`

- used to determine if a file is capable of seeking

`ssize_t read(int fd, void *buf, size_t nbytes) → <unistd.h>`

- fd: the file descriptor for the file to be read
- buf: buffer, uses void* instead of char* to include generic pointers
- Returns number of bytes read, 0 if end of file, -1 if error.
- There are several cases in which the number of bytes actually read is less than the amount requested:
    - if the end of file is reached before the requested number of bytes has been read
    - reading from a terminal device
    - reading from a network
    - reading from a pipe or FIFO
    - reading from a record-oriented device
    - interrupted by a signal and a partial amount of data has already been read

`ssize_t write(int fd, const void *buf, size_t nbytes) → <unistd.h>`

- For a regular file, the write operation starts at the file's current offset.
- If the O_APPEND option was specified when the file was opened, the file's offset is set to the current end of file before each write operation.

I/O Efficiency- nothing super important in this section; just talks about how we depend on the shell to open and close some files.

File Sharing

- The UNIX System supports the sharing of open files among different processes.
- 3 data structures to represent an open file and the relationships among them determine the effect one process has on another with regard to file sharing.
    - every process has an entry in the process table with a table of open file descriptors. With each file descriptor:
        - file descriptor flags
        - pointer to a file table entry
    - kernel maintains a file table for all open files and each entry contains
        - file status flags [read, write, append, sync, and nonblocking]
        - current file offset
        - pointer to the v-node table entry for the file
    - each open file has a v-node structure that contains information about the type of file and pointers to functions that operate on the file; also contains the i-node for the file
- The tables can be arrays, linked lists, etc.
    - after each `write`, current file offset is increased by the number of bytes written to the file
        - if current file offset is greater than current file size, then the current file size in the i-node table entry is set to the current file offset
    - if a file is opened with the O_APPEND flag, a corresponding flag is set in the file status flags of the file table entry

- If a file is positioned to its current end of file using lseek, all that happens is the current file offset in the file table entry is set to the current file size from the i-node table entry

Files

Types of files

- plain: includes binary and text files
- directory: points to another set of files
- link: a pointer to another file or directory
    - Hardlink
        - Create a new file which shares the exact same inode as the targeted file. Any edits to the link file will be reflected in the target file
        - As long as one link is not deleted or corrupted, the data is still reachable
        - Can't be linked to other volumes

Property/Action		Symbolic link	Hard link
When symbolic link/junction/hard link is deleted...		target remains unchanged	reference counter is decremented; when it reaches 0, the target is deleted
When target is moved		symbolic link becomes invalid	hard link remains valid
Relative path		allowed	*not applicable*
Drive requirement		any drive allowed	only same drive
Read of target		allowed	only same drive (link stored in descriptor)
Windows	for files	Windows Vista/2008 and later (administrator rights required)	yes
	for folders		no
Unix	for files	yes	yes
	for directories	yes	no

    - Softlink/Symbolic Link
        - Creates a file that references another file
        - If the link file is deleted, nothing changes. If the linked original file is deleted, The link file will point to nothing. These are considered *orphaned, broken, dead,* or *dangling* .
- others

`ln -s <existingfile> <linkname>` will create a symbolic link so the file, linkname, will be a pointer to the file, existingfile, which can be a file, a directory, etc.
`chown` [only root user] can be used to change owner
`touch` update mod date to current date or create a file if it doesn't exist

view file info (meta-data) via stat command

Directories

File system is interface to

- physical storage on disk
- storage on other machines
- output devices

Working Directory is the current directory

Home directory is where all users store personal files and is represented by ~

ftw(): File Tree Walk // Did she state that this would be on the test tomorrow?

this is most likely on our exam tomorrow

- depth is the maximum number of directories that can be opened at once. Safest value is 1, although it slows down ftw()
- Returns 0 on success (visiting every file), -1 on error
- file is the pathname relative to the start directory, it is passed to MyFunc() automatically by ftw() as it visits each file
- sbuf is a pointer to the stat information for the current file examined