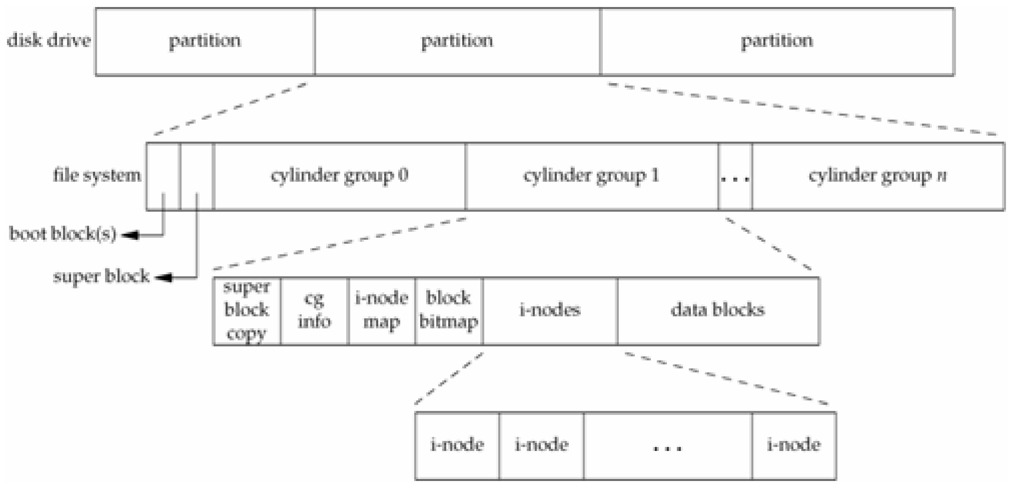
Lecture 5. Files and Directories

Filesystems

System V Release 4 (SVR4) was the most successful release of a UNIX system, originally developed by AT&T, and was the result of an effort known as Unix System Unification. Most UNIX vendors (IBM—AIX, Hewlett Packard—HP-UX, Sun—Solaris) are based on System V.

SVR5 supports two types of disk filesystems: the Unix System V filesystem (S5), and the Unified File System (UFS).

A disk drive is divided into one or more partitions, each of which can contain a filesystem:

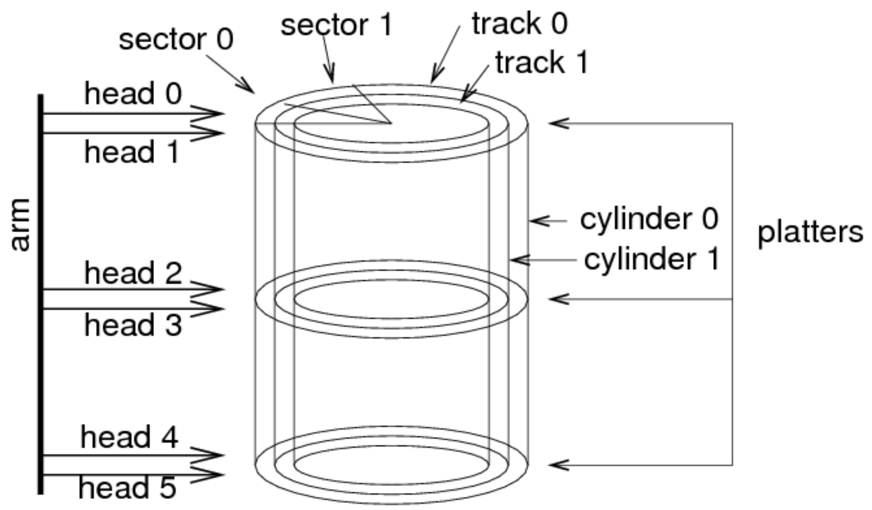


Each partition contains:

- **a boot block** — machine code that gets loaded into the main memory on power up, to allow a boot program (the OS itself) to be loaded from disk into RAM;
- **a super block** — contains filesystem metadata (data about data) and defines the file system type, size, and status. Below is a list of metadata on a super block from oracle.com:
    - size and status of the file system
    - label, which includes the file system name and volume name
    - size of the file system logical block

- date and time of the last update
- cylinder group size
- number of data blocks in a cylinder group
- summary data block
- file system state
- path name of the last mount point

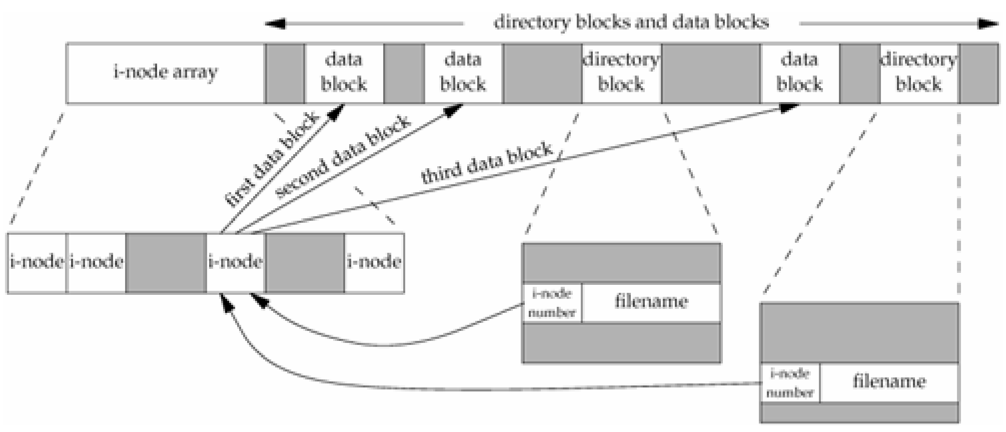
- **cylinder groups** — groups of disk cylinders, usually a megabyte or so in size. Each cylinder group has its own free block list and list of i-nodes;



Each cylinder group contains:

- a copy of the super block
- cylinder group information
- an i-node map
- a block bitmap
- an i-node list
- data blocks

In detail, the filesystem looks as below:



From this representation of the filesystem, we note:

- two directory entries (files) can point to the same i-node. Each i-node has a link count that contains the number of directory entries pointing to it. Only when there are 0 directory entries pointing to an i-node can the directory be deleted. This is why the system call that removes a directory is called `unlink`, and not `delete`. The link count of an i-node is maintained by member `st_nlink` member of the `stat` structure, of type `nlink_t`. These links are called **hard links**. Constant LINK_MAX specifies the max value for the link count for a file.
- symbolic links are data files containing not data, but names of the files that contain the data. For example, to create a symbolic link:

```
$ ln -s source_file target_file
```

```
$ pwd
/Users/awise/Stevens/Lecture4
$ ln -s mystat.c ../Lecture5/link.txt
$ cd ../Lecture5
$ ls -l link.txt
lrwxr-xr-x 1 awise staff 8 Feb 14 20:07 link.txt -> ../Lecture4/
mystat.c
$ readlink link.txt
mystat.c
```

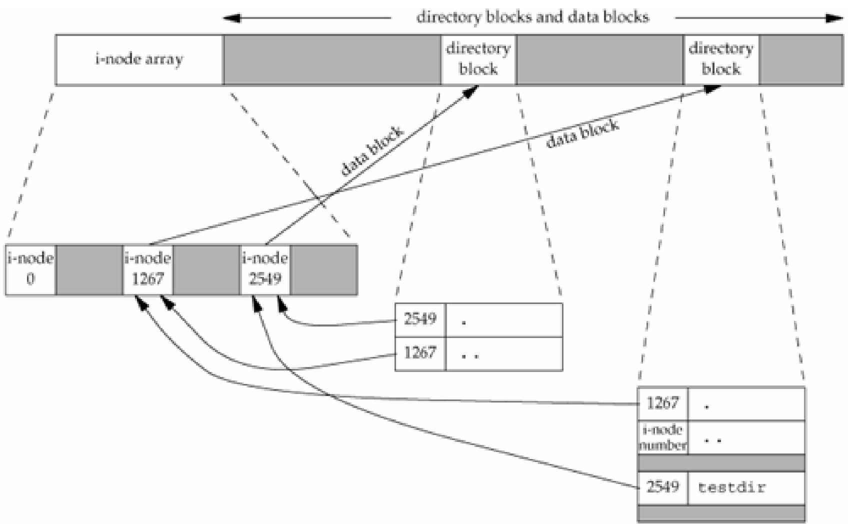
The filename in the directory entry is the string /Users/awise/Stevens/Lecture4/ `link.txt`.

- the **i-node** contains information about the file type and file mode:
    - file type:
        - regular
        - directory
        - block special
        - character special
        - FIFO
        - symbolic link
        - socket
    - file mode (the 9 rwx modes)
    - number of hard links to the file
    - user ID of the owner of the file
    - group ID of the owner of the file
    - number of bytes in the file

- an array of 15 disk block addresses
- date and time the file was last accessed
- date and time the file was last modified
- date and time the i-node was changed

Most of the information from the `stat` structure comes from the i-node. Of this, only the filename and the i-node number are stored in the directory entry.

If we make a new directory `testdir` in the current directory, the filesystem changes as in the following figure:



Each directory block entry has at least 2 fields, by default: the current directory (`.`) and the parent directory (`..`). Each point to their respective i-nodes. When we added a leaf directory `testdir`, the directory block for its parent directory (`1267`) with its parent directory (unspecified i-node #) was added a new field, `2549 testdir`. So `testdir` appears in the directory block of each of its parents + own.

The `link()`, `unlink()`, `remove()`, and `rename()` Functions

Multiple directory entries can point to the same i-node (see above). A link to an existing file (i-node), i.e. a new directory, is created with the link function. For an existing path *`name1`*, only the last component of *`name2`* will be created:

```
#include <unistd.h>

int link(const char *name1, const char *name2);
```

Returns 0 if OK, -1 if error.

[ENOTDIR]	A component of either path prefix is not a directory.
[ENAMETOOLONG]	A component of either pathname exceeded 255 characters, or entire length of either path name exceeded 1023 characters.
[ENOENT]	A component of either path prefix does not exist.
[EOPNOTSUPP]	The file system containing the file named by `name1` does not support links.
[EMLINK]	The link count of the file named by `name1` would exceed {LINK_MAX}.
[EACCES]	A component of either path prefix denies search permission.
[EACCES]	The requested link requires writing in a directory with a mode that denies write permission.
[ELOOP]	Too many symbolic links were encountered in translating one of the pathnames.
[ENOENT]	The file named by `name1` does not exist.
[EEXIST]	The link named by `name2` does exist.
[EPERM]	The file named by `name1` is a directory.
[EIO]	An I/O error occurred while reading from or writing to the file system to make the directory entry.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EFAULT]	One of the pathnames specified is outside the process's allocated address space.

The following code creates a copy of the old file, “oldlink.txt”, into a new file, “newlink.txt”. The new file must *not* exist prior to calling the `link()` function.

```
#include <unistd.h> /* for link() */
#include <stdio.h> /* for printf() */
#include <errno.h> /* for errno() */
#include <string.h> /* for strerror() */

int main(void)
{
     char *file1="/Users/awise/Stevens/Lecture5/oldlink.txt";
     char *file2="/Users/awise/Stevens/Lecture5/newlink.txt";
     int n=link(file1, file2);
     printf("n=%d\n", n);
     printf("%s\n", strerror(errno));
return 0;
}
```

The `unlink()` function removes a directory entry and decrements the link count of the file referenced by *`pathname`*:

```
#include <unistd.h>

int unlink(const char *pathname);
```

Returns 0 if OK, -1 if error.

If there are remaining links to the file, its data can be accessed through the other links. To unlink a file, the user must have write and execute permissions in the directory containing the directory entry (file). Also, if the sticky bit is set in this directory, the user must have write permission for the directory, and either:

- own the file
- own the directory
- have root privileges

The following code removes the new copy “newlink.txt” of the old file “oldlink.txt”, but it can also remove the old file itself. So `unlink()` is `rm`, in effect:

```
#include <unistd.h> /* for link() */
#include <stdio.h> /* for printf() */
#include <errno.h> /* for errno() */
#include <string.h> /* for strerror() */

int main(void)
{
     char *newpath="/Users/awise/Stevens/Lecture5/oldlink.txt";
     int n=unlink(newpath);
     printf("n=%d\n", n);
     printf("%s\n", strerror(errno));
return 0;
}
```

Often the `unlink()` call is used immediately after a process creates a temporary file with `open()` or `creat()`. While the file is still open, it is still available to the process. Only when the process terminates, the file will be *closed* automatically (the kernel will close all the process’ open files), and, due to `unlink()`, also *deleted*.

The following variant of the `myunlink.c` program shows that the space allocated to the file is still available in the directory for the duration that the program is running, and only when the program finishes execution (signaled by the `done` message) the space allocated will be released:

```
#include <unistd.h> /* for link() */
#include <stdio.h> /* for printf() */
#include <errno.h> /* for errno() */
#include <string.h> /* for strerror() */

int main(void)
{
     char *newpath="/Users/awise/Stevens/Lecture5/newlink.txt";
     int n=unlink(newpath);
     printf("File %s unlinked.\n", newpath);
     sleep(15);
     printf("Program done.\n");
     printf("n=%d\n", n);
     printf("%s\n", strerror(errno));
return 0;
}
```

The following sequence of commands illustrates that the space is released:

```
$ ls -l oldlink.txt
-rw-r- -r- - 1 awise staff 13 Feb 15 10:09 oldlink.txt
$ df ~awise/Stevens/Lecture5
Filesystem     512-blocks     Used           Available Capacity
/dev/disk0s2   488555536      290750408      197293128 60%
$ myunlink2 &
[1] 1320
$ File /Users/awise/Stevens/Lecture5/newlink.txt unlinked.
$ Program done.
n=0
Operation timed out

[1]+ Done                      ./myunlink2
$ df ~awise/Stevens/Lecture5
Filesystem     512-blocks     Used           Available Capacity
/dev/disk0s2   488555536      290754792      197288744 60%
```

The `unlink()` function can also be used to remove directories. However, for directories, the function `remove()` could be used instead:

```
#include <stdio.h>

int remove(const char *pathname);
```

Returns 0 if OK, -1 if error.

The `rename()` function simply renames a file or directory:

```
#include <stdio.h>

int rename(const char *oldname, const char *newname);
```
Returns 0 if OK, -1 if error.

If renaming a directory, the ***newname*** must refer to an empty directory (whose entry in the directory block is only `.` and `..` .

Symbolic Links

To circumvent the limitations of hard links—which create a pointer to the i-node of a file—**symbolic links** were introduced. Hard links:

- require that the link and the file reside on the same filesystem;
- for a directory, can be created only by root.

The following table summarizes which functions that take file paths as arguments follow symbolic links and which don't.

Function	**Does not follow symbolic link**	**Follows symbolic link**
`access`		•
`chdir`		•
`chmod`		•
`chown`	•	•
`creat`		•
`exec`		•
`lchown`	•	
`link`		•

`lstat`	•	
`open`		•
`opendir`		•
`pathconf`		•
`readlink`	•	
`remove`	•	
`rename`	•	
`stat`		•
`truncate`		•
`unlink`	•	

The `symlink()` and `readlink()` Functions

Symbolic links are crated with the `symlink()` function:

```
#include <unistd.h>

int symlink(const char *actualpath, const char *sympath);
```
Returns 0 if OK, -1 if error.

The link ***`sympath`*** created with the symbolic link does not have to exist prior to this call. It will be created with the call.

To read the contents of a symbolic link file (the name to the file it points to), use the `readlink()` function:

```
#include <unistd.h>

int readlink(const char *pathname, char *buf, int bufsize);
```
Returns number of bytes read if OK, -1 if error.

The following program creates a local symlink (the current directory being /Users/awise/Stevens/Lecture5), named "link.c", to a file in the previous lecture's directory, /Users/awise/Stevens/Lecture4 (the "mystat.c" file), relative to CWD.

```
#include <fcntl.h> /* for open() */
#include <sys/stat.h> /* for file access permission bits */
#include <stdio.h> /* for FILENAME_MAX */
#include <unistd.h> /* for pathconf() */
#include <string.h> /* for strcat() */
#include <stdlib.h> /* for malloc() */
#include <errno.h> /* for errno */

int main(int argc, char *argv[])
{
     char *newpath="link.c";
     //char *newpath="/Users/awise/Stevens/Lecture5/link.c";
     int      n=symlink("/Users/awise/Stevens/Lecture4/mystat.c",
newpath);
     printf("Function symlink() returned with %d\n", n);

     char buf[100];
     //int     m=readlink("/Users/awise/Stevens/Lecture5/link.c",
buf, 100);
     int m=readlink(newpath, buf, 4096);
     printf("The  contents  of  symlink  %s  is  %s.\n",  newpath,
buf);

     printf("%s\n", strerror(errno));
return 0;
}
```

Note that you can specify the `newpath` as a filename relative to the CWD.

File Times

The following table gives the 3 file times maintained by the kernel for each file:

Field	Description	Example	`ls`(1) option
`st_atime`	last-access time of file data	`read`	`-u`
`st_mtime`	last-modification time of file data	`write`	default
`st_ctime`	last-change time of i-node status	`chmod, chown`	`-c`

Note that there is a difference between `st_mtime` (time of last modification of file's contents) and `st_ctime` (time of last change in its i-node info).

This is because there are operations not involving the change of the file's contents which nevertheless change the file's i-node information, such as: changing access permissions, changing the user ID, changing the link count etc.

The `st_atime` (time of last access) is often used to remove least recently used files.

The `ls` command takes the following options to display the 3 times:

- `ls -l` — last modification of file's data
- `ls -lu` — last access of file's data
- `ls -lc` — last change of i-node status

For example:

```
$ ls -l mysymlink.c
-rw-r- -r- - 1 awise staff 762 Feb 15 13:56 mysymlink.c
$ ls -lu mysymlink.c
-rw-r- -r- - 1 awise staff 762 Feb 15 14:00 mysymlink.c
$ ls -lc mysymlink.c
-rw-r- -r- - 1 awise staff 762 Feb 15 13:56 mysymlink.c
```

The `utime()` Function

The access time and modification time can be *changed* with the `utime()` function:

```
#include <sys/types.h>
#include <utime.h>

int utime(const char *pathname, const struct utimbuf *times);
```

Returns 0 if OK, -1 if error.

The `utimbuf` structure is:

```
struct utimbuf
{
     time_t actime; /* access time */
     time_t modtime; /* modification time */
}
```

The `time_t` data type is implemented as an integral value representing the number of seconds elapsed since 00:00 hours, January 1, 1970 UTC (i.e., a *unix timestamp*).

There are two possibilities:

- either the times argument is null, in which case the access and modification times are both set to the current time. This requires the following permissions:
    (a) effective user ID (who the program runs "as") = file owner's ID (i.e. you can reset the access/modification times of your own files);
    (b) the program has write permission for the file (i.e. whoever else has write on your file).
- or the times argument is a non-null pointer, in which case the access and modification times are set to the value pointed to by `times`. This requires:
    (a) effective user ID (who the program runs "as") = file owner's ID (i.e. you can reset the access/modification times of your own files);
    (b) program is running as root (i.e. root can set the times to any file).

To merely *view* the access times, use the function `stat()` returning the `st_mtime`, `st_atime`, `st_ctime`.

The following program accesses these times for our file "link.c", and then uses the `utime()` function to set them to specified values, respectively reset them to current values.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <utime.h>
#include <time.h> /* for strftime() */
#include <stdio.h>

void FormatTime(time_t rawtime);

int main(int argv, char *argc[])
{
     int n; /* return value of utime() */
     struct stat statbuf; /* gets filled in when calling stat()
*/
     struct utimbuf timebuf; /* either NULL, or specified */

     char *filename="myunlink2.c";
     stat(filename, &statbuf);
     printf("%s:    last    modification    @%ld\n",    filename,
statbuf.st_mtime);
     printf("%s:     last     access     @%ld\n",     filename,
statbuf.st_atime);
```

```
     printf("%s:  last  i-node  status  change@%ld\n",  filename,
statbuf.st_ctime);

     FormatTime(statbuf.st_mtime);
     FormatTime(statbuf.st_atime);
     FormatTime(statbuf.st_ctime);
return 0;
}

void FormatTime(time_t rawtime)
{
     struct tm *info;
     char buffer[80];

     //time(&rawtime); /* sets to current time */
     info=localtime(&rawtime);
     //strftime(buffer, 80, "%x - %I:%M%p", info);
     strftime(buffer, 80, "%b %d %H:%M", info);
     printf("Formatted date & time : | %s |\n", buffer);
}
```

The time returned by `stat()` is in seconds from January 1, 1970. The `strftime()` C function takes the rawtime value returned by the `localtime()` function and displays it in a formatted fashion. Below is the output from the code above, when `stat()` is run on file “myunlink2.c”.

```
$ ./mytimes
myunlink2.c: last modification @1424014005
myunlink2.c: last access @1424014423
myunlink2.c: last i-node status change@1424014006
Formatted date & time : | Feb 15 10:26 |
Formatted date & time : | Feb 15 10:33 |
Formatted date & time : | Feb 15 10:26 |
```

Function `localtime()` is a C function:

```
#include <time.h>

struct tm *localtime(const time_t *clock);
```

Returns a pointer to struct tm if OK, NULL if error.

The argument is a `time_t` type, a time value representing the time in seconds since the Epoch (00:00:00 UTC, January 1, 1970.

The return value is a pointer to the following structure:

```
struct tm
{
     int tm_sec;     /* seconds (0 - 60) */
     int tm_min;     /* minutes (0 - 59) */
     int tm_hour;    /* hours (0 - 23) */
     int tm_mday;    /* day of month (1 - 31) */
     int tm_mon;     /* month of year (0 - 11) */
     int tm_year;    /* year - 1900 */
     int tm_wday;    /* day of week (Sunday = 0) */
     int tm_yday;    /* day of year (0 - 365) */
     int tm_isdst;   /* is summer time in effect? */
     char *tm_zone;  /* abbreviation of timezone name */
     long tm_gmtoff; /* offset from UTC in seconds */
};
```

The function `strftime()` and the format for the tm data type are described in the `man 3` pages.

The `mkdir()` and rmdir() Functions

These two functions create, respectively remove directories.

```
#include <sys/types.h>
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);
```

Returns 0 if OK, -1 if error.

`mkdir()` creates an empty directory, with a `.` and `..` entry in the directory block. The second argument, *`mode`*, specifies the file access permissions. A directory has to have at least one bit set to executable, to allow access to the files within.

```
#include <unistd.h>

int rmdir(const char *pathname);
```

Returns 0 if OK, -1 if error.

Reading Directories

The following functions read directory entries (files), and close directories:

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *pathname);
                                    Returns pointer to DIR if OK, -1 if error.
struct dirent *readdir(DIR *dp);
                         Returns pointer to struct dirent if OK, NULL if error.
void rewinddir(DIR *dp);
int closedir(DIR *dp);
                                                   Returns 0 if OK, -1 if error.
```

The `dirent` structure can be found in/usr/include/sys/dirent.h:

```
struct dirent
{
     ino_t d_ino;                /* file number of entry */
     __uint16_t d_reclen;        /* length of this record */
     __uint8_t  d_type;          /* file type */
     __uint8_t  d_namlen;        /* length of string in d_name */
     char d_name[__DARWIN_MAXNAMLEN + 1];    /* name must be no
longer than this */
};
```

The `DIR` structure can be found in /usr/include/dirent.h:

```
typedef struct
{
     int __dd_fd;/* file descriptor associated with directory */
     long __dd_loc;       /* offset in current buffer */
     long __dd_size;      /* amount of data returned */
     char *__dd_buf;      /* data buffer */
     int  __dd_len;       /* size of data buffer */
     long __dd_seek;      /* magic cookie returned */
     long __dd_rewind;    /* magic cookie for rewinding */
     int  __dd_flags;     /* flags for readdir */
     __darwin_pthread_mutex_t __dd_lock;/* for thread locking */
     struct _telldir *__dd_td; /* telldir position recording */
} DIR;
```

The `chdir()`, `fchdir()`, and `getcwd()` Functions

Every process has a current working directory (CWD), and ever user has a home directory. The CWD is the directory relative to which the search for relative pathnames starts. The CWD of a process can be changed with:

```
#include <unistd.h>

int chdir(const char *pathname);
int fchdir(int fd);
```

Both return 0 if OK, -1 if error.

The arguments are the newly specified CWDs, either explicitly entered as as string, or specified through a file descriptor. The directory will be changed for the calling process of the `chdir()` function, but not the shell that runs that process. The following program illustrates the use of `chdir()`:

```
#include <unistd.h> /* for chdir() */
#include <stdio.h> /* for printf() */
#include <errno.h> /* for errno */
#include <string.h> /* for strerror() */

int main(void)
{
     char *filename="/Users/awise/Stevens/Lecture6";
     printf ("%s\n", filename);
     int i=chdir(filename);
     if (i==-1)
          printf("%s\n", strerror(errno));
     else
          printf("Directory was changed to %s.\n", filename);
return 0;
}
```

The output of this program will be:

```
$ pwd
/Users/awise/Stevens/Lecture5
$ ./mycd
/Users/awise/Stevens/Lecture6
Directory was changed to /Users/awise/Stevens/Lecture6.
$ pwd
/Users/awise/Stevens/Lecture5
```

The `getcwd()` function returns the CWD:

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

Returns `buf` if OK, `NULL` if error.

The arguments are:

- *`buf`* — the address of a buffer, which will acquire the pathname during this call
- *`size`* — size of the buffer, large enough to accommodate the pathname

To illustrate the effect of the `chcwd()` function written earlier, we can add a call to `getcwd()` and also write a file into the new directory, to actually see the directory change:

```
#include <unistd.h> /* for chdir() */
#include <stdio.h> /* for printf() */
#include <errno.h> /* for errno */
#include <string.h> /* for strerror() */
#include <stdlib.h> /* for malloc() */
#include <fcntl.h> /* for open() */

int main(void)
{
     char *filename="/Users/awise/Stevens/Lecture6";
     char *buf; /* address of buffer holding new pathname */
     int size; /* size allocated for pathname */
     char *ptr; /* return value from getcwd() */
     int fd; /* file descriptor of the file created in the new
dir */

     size=1024;
     buf=malloc(size);

     printf ("%s\n", filename);
     int i=chdir(filename);
     if (i==-1)
          printf("%s\n", strerror(errno));
     else
          printf("Directory was changed to %s.\n", filename);

     ptr=getcwd(buf, size);
     if (ptr==NULL)
```

```
          printf("%s\n", strerror(errno));
     else
          printf("CWD=%s\n", ptr);

     char *newfile="/foo";
     strcat(ptr, newfile);
     fd=open(ptr, O_RDWR|O_CREAT|O_TRUNC);
     printf("File descriptor for file %s is %d\n", ptr, fd);
return 0;
}
```

This creates file “foo” in the new working directory “../Lecture6”.

Device Special Files

There are two more members of the stat structure: `st_dev` and `st_rdev` (see Lecture 4). Every filesystem has **major** and **minor device numbers**, encoded in the primitive data type `dev_t`. The major number identifies the device driver; the minor number identifies the sub-device (for example, a disk drive is the major device, and the filesystems are the minor devices).

Current systems (FreeBSD and MacOS X) use a 32-bit integer for the device number, of which 8 bits are for the major number, and 24 are for the minor number. Other systems use 64 bits, with various distributions for the major/minor device IDs.

The `st_dev stat` member, associated with every filename on a system, is the device number of the filesystem containing that file and its i-node. The `st_rdev`, associated with character special files and block special files, is the device number for the actual device.

```
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(int argc, char *argv[])
{
     int i;
     struct stat buf;
     char *filename=argv[1];
     i=stat(filename, &buf);
     if (i<0)
          printf("%s\n", strerror(errno));
     else
     {
          printf("Major device #: %d\n", major(buf.st_dev));
          printf("Minor device #: %d\n", minor(buf.st_dev));
          if (S_ISCHR(buf.st_mode))
               printf("Device %d %d is a character device.\n",
major(buf.st_dev), minor(buf.st_dev));
          else if (S_ISBLK(buf.st_mode))
               printf("Device %d %d is a block device.\n",
major(buf.st_dev), minor(buf.st_dev));
          else
               printf("Neither.\n");
     }
return 0;
}
```

This program lists the major and minor file device #s for a special device file. I ran it for my terminal:

```
$ ./myspecialdev /dev/ttys000
Major device #: 61
Minor device #: 16470152
Device 61 16470152 is a character device.
```

Homework (Due Tuesday, Feb-23-2016):

Copy, adapt, and compile the program that traverses a file hierarchy from Chapter 4, Files and Directories. Taking as input a starting pathname, the program descends the file hierarchy from that point, and returns how many files of each of the seven types there are, and what percentage of the total that represents. (You will need to (re)visit Chapter 2, UNIX Standardizations and Implementations, and create a *.c and a *.h file for `path_alloc()`.)