

CSCI 4061: Files, Directories, Standard I/O

Chris Kauffman

Last Updated:

Wed Feb 17 03:58:38 PM CST 2021

Logistics

Reading

Stevens/Rago Ch 3, 4, 5, 6

Goals for Week

- `read()/write()`
- I/O Redirection
- Pipes
- C FILE* vs Unix FDs
- Filesystem
- Permissions
- Hard/Symbolic Links
- File / Directory Functions

Date	Event
Mon 2/15	Lab: dup2() Basic I/O, Filesystem
Wed 2/17	Filesystem
Mon 2/22	Lab: Review Lec: Practice Exam Project 1 Due
Wed 2/24	Exam 1 Last day to submit P1 late

P1 Questions?

Due date approaching rapidly

Permissions / Modes

- ▶ Unix enforces file security via *modes*: permissions as to who can read / write / execute each file
- ▶ See permissions/modes with `ls -l`
- ▶ Look for series of 9 permissions

```
> ls -l
total 140K
-rwx--x--- 2 kauffman faculty  8.6K Oct  2 17:39  a.out
-rw-r--r-- 1 kauffman devel    1.1K Sep 28 13:52  files.txt
-rw-rw---- 1 kauffman faculty  1.5K Sep 26 10:58  gettysburg.txt
-rwx--x--- 2 kauffman faculty  8.6K Oct  2 17:39  my_exec
----- 1 kauffman kauffman  128 Oct  2 17:39  unreadable.txt
-rw-rw-r-x 1 root      root      1.2K Sep 26 12:21  scripty.sh
U  G  O      O      G      S      M  T      N
S  R  T      W      R      I      O  I      A
E  O  H      N      O      Z      D  M      M
R  U  E      E      U      E      E
      P  R      R      P
~~~~~
PERMISSIONS
```

- ▶ Every file has permissions set from somewhere on creation

Changing Permissions

Owner of file (and sometimes group member) can change permissions via `chmod`

```
> ls -l a.out
```

```
-rwx--x--- 2 kauffman faculty 8.6K Oct 2 17:39 a.out
```

```
> chmod u-w,g+r,o+x a.out
```

```
> ls -l a.out
```

```
-r-xr-x--x 2 kauffman faculty 8.6K Oct 2 17:39 a.out
```

- ▶ `chmod` also works via octal bits (suggest against this unless you want to impress folks at parties)
- ▶ Programs specify file permissions via system calls
- ▶ Curtailed by **Process User Mask** which indicates permissions that are disallowed by the process
 - ▶ `umask` shell function/setting: `$> umask 007`
 - ▶ `umask()` system call: `umask(S_IWGRP | S_IWOTH);`
- ▶ Common program strategy: create files with very liberal read/write/execute permissions, `umask` of user will limit this

Exercise: Regular File Creation Basics

C Standard I/O

- ▶ Write/Read data?
- ▶ Open a file, create it if needed?
- ▶ Result of opening a file?
- ▶ Close a file?
- ▶ Set permissions on file creation?

Unix System Calls

- ▶ Write/Read data?
- ▶ Open a file, create it if needed?
- ▶ Result of opening a file?
- ▶ Close a file?
- ▶ Set permissions on file creation?

Answers: Regular File Creation Basics

C Standard I/O

- ▶ Write/Read data?

```
fscanf(), fprintf()  
fread(), fwrite()
```

- ▶ Open a file, create it if needed?
- ▶ Result of opening a file?

```
FILE *out =  
    fopen("myfile.txt", "w");
```

- ▶ Close a file?

```
fclose(out);
```
- ▶ Set permissions on file creation?
Not possible... dictated by `umask`

Unix System Calls

- ▶ Write/Read data?

```
write(), read()
```

- ▶ Open a file, create it if needed?
- ▶ Result of opening a file?

```
int fd =  
    open("myfile.txt",  
        O_WRONLY | O_CREAT,  
        permissions);
```

- ▶ Close a file?

```
close(fd);
```
- ▶ Set permissions on file creation?
 - ▶ Additional options to `open()`, which brings us to...

Permissions / Modes in System Calls

`open()` can take 2 or 3 arguments

```
int fd = open(name, flags);  
# new file will have NO permissions  
# to read/write, not an issue if opening  
# existing file
```

```
int fd = open(name, flags, perms);  
                ~~~~~  
# new file will have given permissions  
# (subject to the umask), ignored for  
# existing files
```

Symbol	Entity	Sets
S_IRUSR	User	Read
S_IWUSR	User	Write
S_IXUSR	User	Execute
S_IRGRP	Group	Read
S_IWGRP	Group	Write
S_IXGRP	Group	Execute
S_IROTH	Others	Read
S_IWOTH	Others	Write
S_IXOTH	Others	Execute

Compare: `write_readable.c` VERSUS `write_unreadable.c`

```
char *outfile = "newfile.txt";           // doesn't exist yet  
int flags     = O_WRONLY | O_CREAT;      // write/create  
mode_t perms  = S_IRUSR | S_IWUSR;      // variable for permissions  
int out_fd    = open(outfile, flags, perms);  
                ~~~~~
```

Filesystems, inodes, links

- ▶ Unix **filesystems** implement physical layout of files/directories on a storage media (disks, CDs, etc.)
- ▶ Many filesystems exist but all Unix-centric filesystems share some common features

inode

- ▶ Kernel data structure which describes a single file
- ▶ Stores some meta data: inode#, size, timestamps, owner
- ▶ A table of contents: which disk blocks contain file data
- ▶ Does **not** store filename, does store a **link count**

Directories

- ▶ List names and associated inode
- ▶ Each entry constitutes a **hard link** to an inode or a **symbolic link** to another file
- ▶ Files with 0 hard links are deleted

Rough Filesystem in Pictures 1

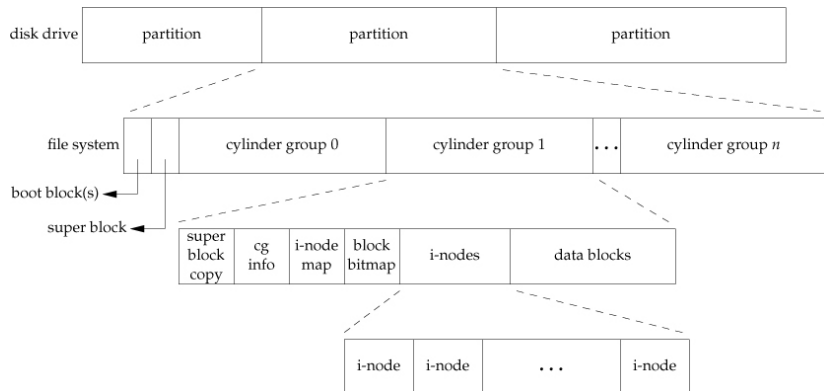


Figure 4.13 Disk drive, partitions, and a file system (Stevens/Rago)

Rough Filesystem in Pictures 2

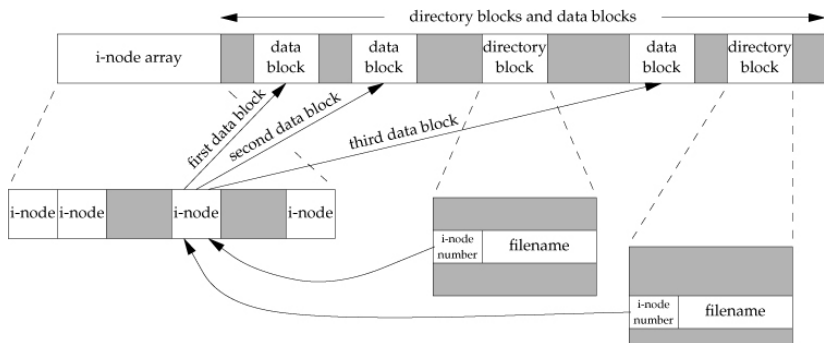


Figure 4.14 Cylinder group's i-nodes and data blocks in more detail (Stevens/Rago)

Shell Demo of Hard and Symbolic Links

```
> rm *
> touch fileX                # create empty fileX
> touch fileY                # create empty fileY
> ln fileX fileZ             # hard link to fileX called fileZ
> ln -s fileX fileW          # symbolic link to fileX called fileW
> ls -li                     # -i for inode numbers
total 12K
6685588 -rw-rw----  2 kauffman kauffman 0 Oct  2 21:24 fileX
6685589 -rw-rw----  1 kauffman kauffman 0 Oct  2 21:24 fileY
6685588 -rw-rw----  2 kauffman kauffman 0 Oct  2 21:24 fileZ
6685591 lrwxrwxrwx  1 kauffman kauffman 5 Oct  2 21:29 fileB -> fileA
6685590 lrwxrwxrwx  1 kauffman kauffman 5 Oct  2 21:25 fileW -> fileX
↑↑↑↑↑↑↑ ↑          ↑          ↑↑↑↑↑↑↑↑
inode#  regular    hard link count      symlink target
        or symlink

> file fileW                # file type of fileW
fileW: symbolic link to fileX
> file fileB                 # file type of fileB
fileB: broken symbolic link to fileA
```

Linking Commands and Functions

Shell Command	C Function	Effect
<code>ln fileX fileY</code>	<code>link("fileX", "fileY");</code>	Create a hard link
<code>rm fileX</code>	<code>remove("fileX");</code> <code>unlink("fileX");</code>	Unlink (remove) hard link Identical to <code>remove()</code>
<code>ln -s fileX fileY</code>	<code>symlink("fileX", "fileY");</code>	Create a Symbolic link

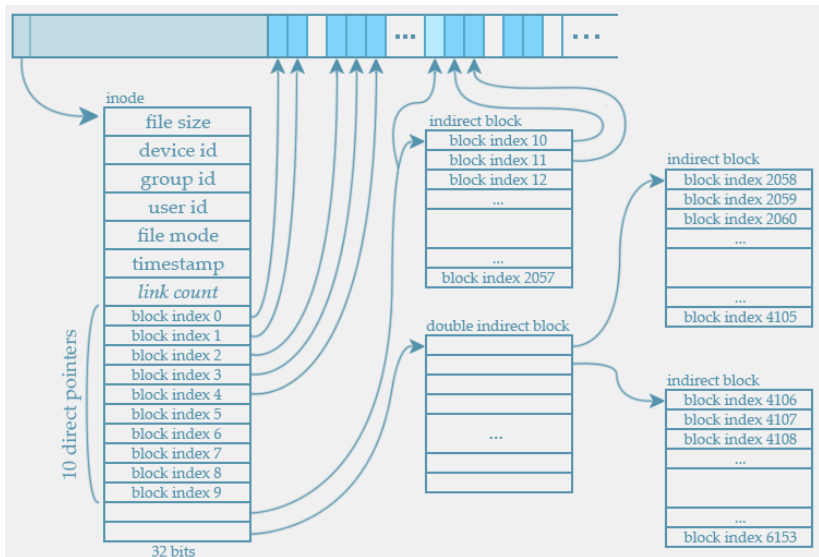
- ▶ Creating hard links preserves inodes
- ▶ Hard links not allowed for directories unless you are root

```
> ln /home/kauffman to-home  
ln: /home/kauffman: hard link not  
    allowed for directory
```

Can create directory cycles if this was allowed

- ▶ Symlinks easily identified so utilities can skip them

FYI: inodes are a complex beast themselves



Source: File System Design by Justin Morgan

sync() and Internal OS Buffers

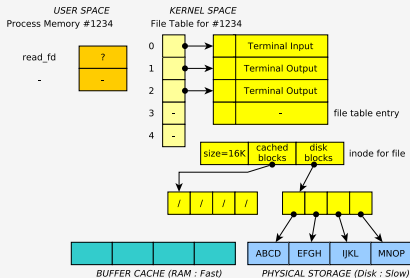
- ▶ Operating system maintains internal data associated with open files
- ▶ Writing to a file doesn't go immediately to a disk
- ▶ May live in an internal buffer for a while before being sync'ed to physical medium (OS buffer cache)

Shell Command	C function	Effect
sync	sync(); syncfs(fd);	Synchronize cached writes to persistent storage Synchronize cached writes for filesystem of given open fd

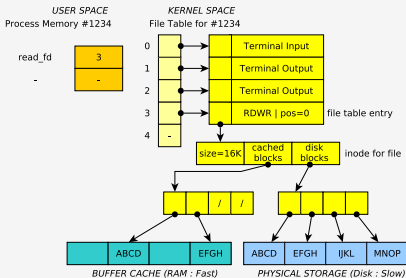
- ▶ Sync called so that one can “Safely remove drive”
- ▶ Sync happens automatically at regular intervals (ex: 15s)

File Caching Demo

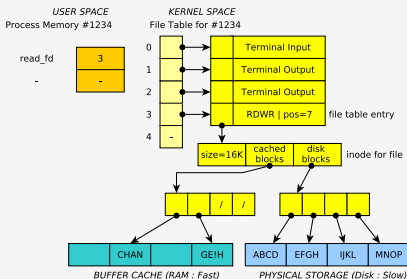
1. Start of program : file not yet opened



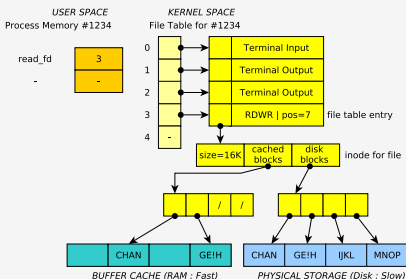
2. read_fd = open(...) completes



3. write(3, "CHANGE!", 7); completes



4. sync() completes (automatically done by OS every few seconds)



Movement within Files, Changing Sizes

- ▶ Can move OS internal position in a file around with `lseek()`
- ▶ Note that size is arbitrary: can seek to any positive position
- ▶ File automatically expands if position is larger than current size - fills holes with 0s (null chars)
- ▶ Can manually set size of a file with `ftruncate(fd, size)`
- ▶ Examine `file_hole1.c` and `file_hole2.c`

C function	Effect
<code>int res = lseek(fd, offset, option);</code>	Move position in file
<code>lseek(fd, 20, SEEK_CUR);</code>	Move 20 bytes forward
<code>lseek(fd, 50, SEEK_SET);</code>	Move to position 50
<code>lseek(fd, -10, SEEK_END);</code>	Move 10 bytes from end
<code>lseek(fd, +15, SEEK_END);</code>	Move 15 bytes beyond end
<code>ftruncate(fd, 64);</code>	Set file to be 64 bytes big If file grows, new space is zero-filled

Note: C standard I/O functions `fseek(FILE*)` and `rewind(FILE*)` mirror functionality of `lseek()`

Basic File Statistics via stat

Command	C function	Effect
stat file	int ret = stat(file,&statbuf);	Get statistics on file
	int ret = lstat(file,&statbuf);	Same, don't follow symlinks
	int fd = open(file,...);	Same as above but with
	int ret = fstat(fd,&statbuf);	an open file descriptor

Shell command stat provides basic file info such as shown below

```
> stat a.out
  File: a.out
  Size: 12944          Blocks: 40          IO Block: 4096   regular file
Device: 804h/2052d   Inode: 6685354     Links: 1
Access: (0770/-rwxrwx---)  Uid: ( 1000/kauffman)   Gid: ( 1000/kauffman)
Access: 2017-10-02 23:03:21.192775090 -0500
Modify: 2017-10-02 23:03:21.182775091 -0500
Change: 2017-10-02 23:03:21.186108423 -0500
 Birth: -

> stat /
  File: /
  Size: 4096          Blocks: 8          IO Block: 4096   directory
Device: 803h/2051d   Inode: 2           Links: 17
Access: (0755/drwxr-xr-x)  Uid: (   0/   root)   Gid: (   0/   root)
Access: 2017-10-02 00:56:47.036241675 -0500
Modify: 2017-05-07 11:34:37.765751551 -0500
Change: 2017-05-07 11:34:37.765751551 -0500
 Birth: -
```

See `stat_demo.c` for info on C calls to obtain this info

Directory Access

- ▶ Directories are fundamental to Unix (and most file systems)
- ▶ Unix file system rooted at / (root directory)
- ▶ Subdirectories like `bin`, `~/home`, and `/home/kauffman`
- ▶ Useful shell commands and C function calls pertaining to directories are as follows

Shell Command	C function	Effect
<code>mkdir name</code>	<code>int ret = mkdir(path,perms);</code>	Create a directory
<code>rmdir name</code>	<code>int ret = rmdir(path);</code>	Remove empty directory
<code>cd path</code>	<code>int ret = chdir(path);</code>	Change working directory
<code>pwd</code>	<code>char *path = getcwd(buf,SIZE);</code>	Current directory
<code>ls</code>	<code>DIR *dir = opendir(path);</code>	List directory contents
	<code>struct dirent *file = readdir(dir);</code>	Start reading filenames from dir
	<code>int ret = closedir(dir);</code>	Call in a loop, NULL when done After <code>readdir()</code> returns NULL

See `dir_demo.c` for demonstrations

Exercise: Sketch Code for Total Size of Regular Files

- ▶ Code which will scan all files in a directory
- ▶ Will get file statistics on each file
- ▶ Skips directories, symlinks, etc.
- ▶ Totals bytes of all Regular files in current directory

Use techniques demoed in `dir_demo.c` and `stat_demo.c` from codepack

```
> gcc total_size.c
> ./a.out
    26 readable1.txt
  1299 buffered_output.c
   2512 stat_demo.c
...
   584 file_hole2.c
SKIP .
SKIP my_symlink
SKIP subdir
   907 dir_demo.c.bk
...
  1415 write_umask.c
=====
  67106 total bytes
```

Answers: Sketch Code for Total Size of Regular Files

```
// total_size.c
int main(int argc, char *argv[]){
    size_t total_size = 0;
    DIR *dir = opendir(".");
    while(1){
        struct dirent *file = readdir(dir);
        if(file == NULL){
            break;
        }
        struct stat sb;
        lstat(file->d_name, &sb);
        if(S_ISREG(sb.st_mode)){
            printf("%8lu %s\n",
                sb.st_size, file->d_name);
            total_size += sb.st_size;
        }
        else{
            printf("%-8s %s\n",
                "SKIP", file->d_name);
        }
    }
    closedir(dir);
    printf("=====\n");
    printf("%8lu total bytes from REGULAR files\n",
        total_size);
    return 0;
}
```

- ▶ Scans only current directory
- ▶ **Recursive scanning** is trickier and involves... recursion
- ▶ OR the very useful `nftw()` library function, discussed in upcoming HW
- ▶ Techniques required for upcoming P2

Files in Trees

- ▶ Frequently one wants to visit all files in a directory tree
- ▶ P2: check all files for changes / revision control
- ▶ Options for this on the command line and via system calls

find utility on Shell

```
> find .
.
./c
./c/d.txt
./b.txt
./src
./src/tests
./src/tests/results.txt
./src/main.c
./src/code.c
./a.txt

> find . -name '*.c'
./src/main.c
./src/code.c
```

nftw() System Call in C

- ▶ File Tree Walk : visit all files in a directory
- ▶ A Higher Order function: function parameter

```
nftw(filename, count_file,      ...);
nftw(filename, print_file_info, ...);
nftw(filename, delete_file,    ...);

int print_file_info(const char *filename,
                   const struct stat *sb,
                   ...);
```

- ▶ Covered in HW7, used in P2

Multiplexed Input/Output

- ▶ Occasions arise when one must `read()` from several sources BUT it is unclear which source is ready and which is not
- ▶ OS can provide information on ready sources
- ▶ Future HW will cover `poll()` and/or `select()` system calls which are used for this
- ▶ Will need it for a project later in the semester
- ▶ Remaining slides will be revisited then

select() and poll(): Non-busy waiting

- ▶ Recall **polling** is a busy wait on something: constantly check until ready
- ▶ Alternative is **interrupt-driven** wait: ask for notification when something is ready, go to sleep, get woken up
- ▶ Waiting is often associated with input from other processes through pipes or sockets
- ▶ Both `select()` and `poll()` allow for waiting on input from multiple file descriptors
- ▶ Confusingly, **both `select()` and `poll()` are interrupt-driven**: will put process to sleep until something changes in one or more files
- ▶ `poll()` doesn't do polling (busy wait) - it does interrupt driven I/O (!!)
- ▶ Example application: database system is waiting for any of 10 users to enter a query, don't know which one will type first

poll() System Call

- ▶ Modern usage favors `poll()` for multiplexed I/O
- ▶ Despite name, `poll()` blocks a process until one of several input/output sources are immediately ready
- ▶ Allows for an interrupt-driven style of programming
- ▶ Covered in [Demo usage of the poll\(\) System Call](#)

select() System Call and File Descriptor Sets

- ▶ `select()` uses file descriptor **sets**
- ▶ `fd_set` tracks descriptors of interest, operated on with macros

```
fd_set my_set;
void FD_ZERO(fd_set *set);           // clear entire set
void FD_SET(int fd, fd_set *set);    // fd now in set
void FD_CLR(int fd, fd_set *set);    // fd now not in set
int  FD_ISSET(int fd, fd_set *set);  // test if fd in set
```

- ▶ Example: setup set of potential read sources

```
int pipeA[2], pipeB[2], rd_fd;      // set up several read sources
pipe(pipeA);
pipe(pipeB);
rd_fd = open("myfile.txt",RD_ONLY);

fd_set read_set;                    // set of file descriptors for select()
FD_ZERO(&read_set);                 // init the set

FD_SET(pipeA[PREAD], &read_set);    // include read ends of pipes in set
FD_SET(pipeB[PREAD], &read_set);
FD_SET(rd_fd, &read_set);           // include read file in the set
```

Multiplexing: Efficient input from multiple sources

- ▶ `select()` block a process until at least one of member of the `fd_set` is “ready”
- ▶ Most common use: waiting for input from multiple sources
- ▶ Example: Multiple child processes writing to pipes at different rates

```
#include <sys/select.h>
fd_set read_set, write_set,      // sets of fds to wake up for
    except_set;

struct timeval timeout;          // allows timeout: wake up if nothing happens

int nfd =                         // returns nfd changed
    select(maxfd+1,              // must pass max fd+1
           &read_set,            // any of set may be NULL to ignore
           &write_set,
           &except_set,
           &timeout);           // NULL time waits indefinitely
```

- ▶ `poll()` performs similar multiplexed block but has a different interface