CSCI 4061: Files, Directories, Standard I/O

Chris Kauffman

Last Updated:
Thu Feb 21 09:46:26 CST 2019

Logistics

Reading

- Stevens/Rago
  Ch 3, 4, 5, 6
- OR Robbins and Robbins
  Ch 4, 5

Goals

- Std I/O vs Unix Syscall
- File / Directory Functions
- Filesystem

Lab04: Pipes

How did it go?

Project 1

Questions?

Exam 1: Next week

- Tue Review
- Thu Exam

Exercise: Quick Recap

1. What is a pipe? What system call is used to create it? Example?
2. How does one put data into a pipe? Get data from a pipe?
3. How can one arrange for communication between a parent and child process?
    - Child to parent
    - Parent to child
4. What syntax do standard shells use to redirect program output to files?
5. What low-level system calls are used to a accomplish redirection?

Answers: Quick Recap

1. What is a pipe? What system call is used to create it? Example?
   - Internal OS communication buffer, created via
     ```
     int pip;
     int result = pipe(pip);
     ```
2. How does one put data into a pipe? Get data from a pipe?
   - ```
     nbytes = write(pip[1], w_buff, BUFLEN);
     nbytes =  read(pip[0], r_buff, BUFLEN);
     ```
3. How can one arrange for communication between a parent and child process?
   - Child to parent: parent opens pipe, child writes, parent reads
   - Parent to child: parent opens pipe, parent writes, child reads
4. What syntax do standard shells use to redirect program output to files? Read input from files?
   - ```
     $> my_program arg1 arg2 > output.txt
     $> other_prog arg1 < input.txt
     ```
5. What low-level system calls are used to a accomplish redirection?
   - `dup2(fd_a, fd_b);`
   - writes to `fd_b` write to `fd_a` instead
   - reads from `fd_b` read from `fd_a` instead

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           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 permissions for files they create via C calls
- Curtailed by the `umask` shell or `umask()` C function: indicates permissions that are not allowed
- 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 C Calls

- Default `open(name,opts)` has NO PERMISSIONS
- When opening with `O_CREAT`, specify permissions for new file
- `int fd = open(name, opts, mode);`

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);
                                     ^^^^^
```

C Standard I/O Implementation

Typical Unix implementation of standard I/O library `FILE` is

- A file descriptor
- Some buffers with positions
- Some options controlling buffering

From `/usr/lib/libio.h`

```
struct _IO_FILE {
  int _flags;                    // options
  char* _IO_read_ptr;            // positions and
  char* _IO_read_end;            // buffers for
  char* _IO_read_base;           // read and write
  char* _IO_write_base;
  ...;
  int _fileno;                   // file descriptor
  ...;
  _IO_lock_t *_lock;             // locking
};
```

Exercise: Subtleties of Mixing Standard and Low-Level I/O

- Predict output of program given input file
- Use knowledge that buffering occurs internally for standard I/O library
- **Note:** Similar subtleties exist if `FILE*` are not properly closed
- `FILE` buffers may contain unflushed data: not written at close
- See `fail-to-write.c`
- File descriptors always get flushed out by OS

```
3K.txt:
 1 2 3 4 5 6 7 8 9 10 11 12 13 14...
37 38 39 40 41 42 43 44 45 46 47 ...
70 71 72 73 74 75 76 77 78 79 80 ...
102 103 104 105 106 107 108 109 1...
...
```

```
mixed-std-low.c:
```

```c
int main(int argc, char *argv[]){

  FILE *input = fopen("3K.txt","r");
  int first;
  fscanf(input, "%d", &first);
  printf("FIRST: %d\n",first);

  int fd = fileno(input);
  char *buf[64];
  read(fd, buf, 63);
  buf[63] = '\0';
  printf("NEXT: %s\n",buf);

  return 0;
}
```

Controlling FILE Buffering

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
void setbuffer(FILE *stream, char *buf, size_t size);
void setlinebuf(FILE *stream);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Series of functions which control buffering. Example:

```
// Turn off buffering of stdout
setvbuf(stdout, NULL, _IONBF, 0);
```

Why should this line be familiar to **ALL** of you?

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

- 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

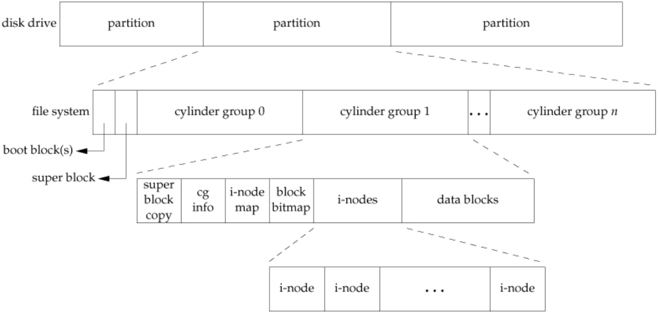
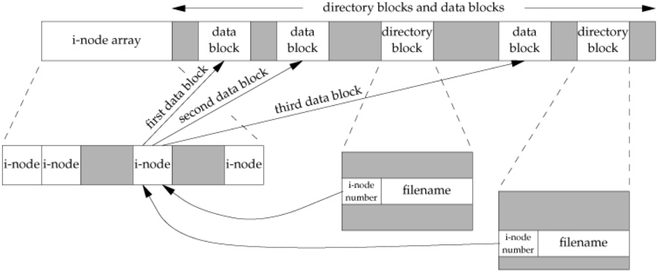


Figure 4.13 Disk drive, partitions, and a file system (Stevens/Rago)



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

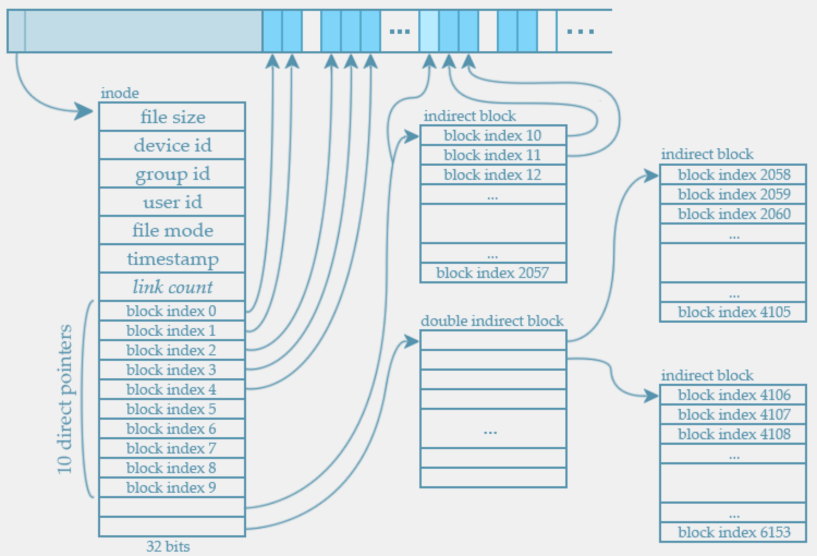
Shell Command	C Function	Effect
`ln fileX fileY`	`link("fileX", "fileY");`	Create a hard link
`rm fileX`	`remove("fileX");`	Unlink (remove) hard link
	`unlink("fileX");`	Identical to `remove()`
`ln -s fileX fileY`	`symlink("fileX", "fileY");`	Create a Symbolic link

- 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 directo
```

  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();`	Synchronize cached writes to persistent storage
	`syncfs(fd);`	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)

Basic File Statistics via `stat`

Command	C function	Effect
`stat file`	`int ret = stat(file,&statbuf);`	Get statistics on file
	`int fd = open(file,...);` `int ret = fstat(fd,&statbuf);`	Same as above but with an open file descriptor

Shell command `stat` provides basic file info such as shown below

```
> stat a.out
  File: a.out
  Size: 12944     ^^IBlocks: 40         IO Block: 4096   regular file
Device: 804h/2052d^^IInode: 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      ^^IBlocks: 8          IO Block: 4096   directory
Device: 803h/2051d^^IInode: 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)
- Subdirectores like `bin`, `~/home`, and `/home/kauffman`
- Useful shell commands and C function calls pertaining to directories are as follows

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

See `dir-demo.c` for demonstrations

Movement within Files

- 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)
- Examine `file-hole.c` and `file-hole2.c`

C function	Effect
`int res = lseek(fd, offset, option);`	Move position in file
`lseek(fd, 20, SEEK_CUR);`	Move 20 bytes forward
`lseek(fd, 50, SEEK_SET);`	Move to position 50
`lseek(fd, -10, SEEK_END);`	Move 10 bytes from end
`lseek(fd, +15, SEEK_END);`	Move 15 bytes beyond end

See also C standard I/O `fseek(FILE *) / rewind(FILE *)` functions

fnctl(): Jack of all trades

- `fcntl()` does a bunch of stuff
- Some previous calls implemented with `fcntl()`
  - `int fd2 = dup(fd1);` OR
  - `int fd2 = fcntl(fd1,F_DUPFD);`

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int fcntl(int fd, int cmd, /* arg */ ...);
```

Command	Effect
F_DUPFD	duplicate a file descriptor
F_GETFD	get file descriptor flags
F_SETFD	set file descriptor flags
F_GETFL	get file status flags and access modes
F_SETFL	set file status flags and access modes
F_GETOWN	get proc ID currently receiving SIGIO and SIGURG signals for fd
F_SETOWN	set proc ID that will receive SIGIO and SIGURG signals for fd
	Locking
F_GETLK	get first lock that blocks description specified by arg
F_SETLK	set or clear segment lock specified by arg
F_SETLKW	same as FSETLK except it blocks until request satisfied
...	

`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

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 nfds =                        // returns nfds 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
```

- Future lab will cover `select()`
- Will need it for a project later in the semester