CMSC 216: UNIX File Input/Output

Chris Kauffman

Last Updated:
Mon Apr 14 03:26:15 PM EDT 2025

Logistics

Reading: Bryant/O'Hallaron

Ch	Read?	Topic
8	Finish	See specific section guide from previous slides
10	READ Except	UNIX File structure, File System structure, I/O functions
10.5	Opt	Optional: "Robust" I/O library built on top of primitive ops

Assignments

- Grading has commenced for P3 / Exam 2, likely to complete late this week
- Lab09 on `fork()` / `wait()` + HW09 on `fork()`
- P4 up tomorrow

Goals

- Finish up Process Intro
- C Standard I/O library vs UNIX I/O
- File Descriptors, `open()` / `close()` / `read()` `write()`
- I/O Redirection with `dup2()` / `dup()`
- File Atributes / Permissions `stat()` / `chmod()`
- (Optional) Directory Traversal `opendir()` / `readdir()`

Announcements

CS Undergrad Town Hall Mon 21-Apr

- Chance to voice concerns / get attention to issues for CS students
- Nominations for Undergrad rep to Department Council and Education Committee
- Chances to demonstrate service and leadership for those who want those things on their resume
- Food provided

RSVP: `https://go.umd.edu/cstownhall25`

Exercise: C Standard I/O Functions

Recall basic I/O functions from the C Standard Library header `stdio.h`

1. Printing things to the screen?
2. Opening a file?
3. Closing a file?
4. Printing to a file?
5. Scanning from terminal or file?
6. Get whole lines of text?
7. Names for standard input, output, error

Give samples of function calls

Answers: C Standard I/O Functions

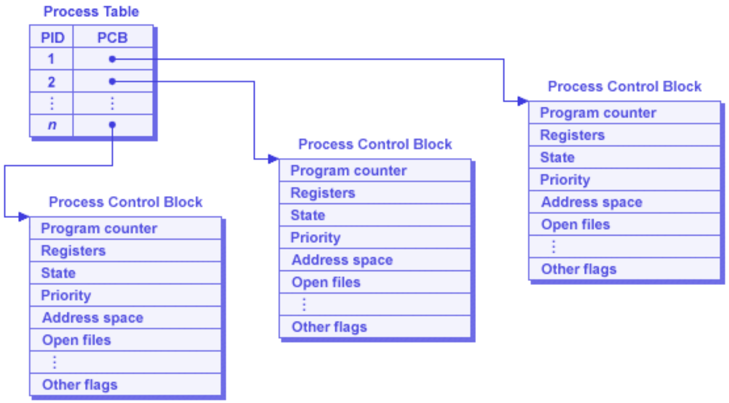
Recall basic I/O functions from the C Standard Library header `stdio.h`

1	`printf("%d is a number",5);`	Printing things to the screen?
2	`FILE *file = fopen("myfile.txt","r");`	Opening a file?
3	`fclose(file);`	Close a file?
4	`fprintf(file,"%d is a number",5);`	Printing to a file?
5	`scanf("%d %f",&myint,&mydouble);`	Scanning from terminal
	`fscanf(file2,"%d %f",&myint,&mydouble);`	or file?
6	`result = fgets(charbuf, 1024, file);`	Get whole lines of text?
7	`FILE *stdin, *stdout, *stderr;`	Names for standard input, etc

The standard I/O library was written by Dennis Ritchie around 1975.
–Stevens and Rago, Advanced Programming for the Unix Environment

- Assuming you are familiar with these and could look up others like `fgetc()` (single char) and `fread()` (read binary)
- Library Functions: available with any compliant C compiler
- On Unix systems, `fscanf()`, `FILE*`, and the like are backed by lower level System Calls and Kernel Data Structures

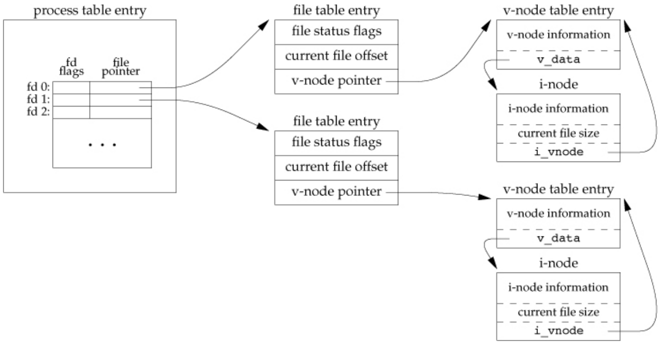
The Process Table



Source: SO What is the Linux Process Table?

- OS maintains data on all processes in a Process Table
- Process Table Entry ≈ Process Control Block
- Contains info like PID, instruction that process is executing*, Virtual Memory Address Space and **Files in Use**

File Descriptors



- Each Process Table entry contains a table of open files
- A use program refers to these via **File Descriptors**
- File Descriptor is an integer index into Kernel's table

```
int fd = open("some_file.txt", O_RDONLY);
```

- FD Table entry refers to other Kernel/OS data structures

File Descriptors are Multi-Purpose

- Unix tries to provide most things via files/file descriptor
- Many Unix system actions are handled via `read()`-from or `write()`-to file descriptors
- File descriptors allow interaction with standard like `myfile.txt` or `commando.c` to read/change them
- FD's also allow interaction with many other things
  - Pipes for interprocess communication
  - Sockets for network communication
  - Special files to manipulate terminal, audio, graphics, etc.
  - Raw blocks of memory for Shared Memory communication
  - Even processes themselves have special files in the file system: ProcFS in `/proc/PID#`, provide info on running process
- We will focus on standard File I/O using FDs Now and touch on some broader uses Later
- Also must discuss FD interactions with previous System Calls: **What happens with `open()` files when calling `fork()`?**

Open and Close: File Descriptors for Files

```c
#include <sys/stat.h>
#include <fcntl.h>

int fd1 = open("firstfile", O_RDONLY); // read only
if(fd1 == -1){                         // check for errors on open
  perror("Failed to open 'firstfile'");
}

int fd2 = open("secndfile", O_WRONLY); // write only, fails if not found
int fd3 = open("thirdfile", O_WRONLY | O_CREAT); // write only, create if needed
int fd4 = open("forthfile", O_WRONLY | O_CREAT | O_APPEND); // append if existing

// 'man 3 open' will list all the O_xxx options when opening.
//  Other common options: O_RDONLY, O_RDWR, O_EXEC

...;                            // Do stuff with open files

int result = close(fd1); // close the file associated with fd1
if(result == -1){        // check for an error
  perror("Couldn't close 'firstfile'");
}
```

`open()` / `close()` show common features of many system calls

- Returns -1 on errors
- Show errors using the `perror()` function
- Use of vertical pipe (|) to bitwise-OR several options

read() from File Descriptors

```c
// read_some.c: Basic demonstration of reading data from
// a file using open(), read(), close() system calls.

#define SIZE 128

{
  int in_fd = open(in_name, O_RDONLY);
  char buffer[SIZE];
  int bytes_read = read(in_fd, buffer, SIZE);
}
```

- Read up to `SIZE` from an open file descriptor
- Bytes stored in `buffer`, overwrite it
- Return value is number of bytes read, -1 for error
- `SIZE` commonly defined but can be variable, constant, etc
- **Examine `read_some.c`**: explain what's happening

Caution:

- Bad things happen if `buffer` is actually smaller than `SIZE`
- `read()` does NOT null terminate, add `\0` manually if needed

Exercise: Behavior of `read()` in `count_bytes.c`

Run `count_bytes.c` on file `data.txt`

```
> cat data.txt
ABCDEFGHIJ
> gcc count_bytes.c
> ./a.out data.txt
???
```

1. Explain control flow within program
2. Predict output of program

```c
// count_bytes.c
#define BUFSIZE 4

int main(int argc, char *argv[]){
  char *infile = argv[1];
  int in_fd = open(infile,O_RDONLY);
  char buf[BUFSIZE];
  int nread, total=0;
  while(1){
    nread = read(in_fd,buf,BUFSIZE-1);
    if(nread == 0){
      break;
    }
    buf[nread] = '\0';
    total += nread;
    printf("read: '%s'\n",buf);
  }
  printf("%d bytes total\n",total);
  close(in_fd);
  return 0;
}
```

Answers: Behavior of read() in count_bytes.c

```
==INITIAL STATE==
data.txt: ABCDEFGHIJ\n
position: ^
buf:     |? ? ? ? |
          0 1 2 3
nread:   0
total:   0
```

```
==ITERATION 1==
nread = read(in_fd,buf,3);
buf[nread] = '\0'
total+= nread;
printf("read: '%s'\n",buf);

data.txt: ABCDEFGHIJ\n
position:    ^
buf:     |A B C \0|
          0 1 2 3
nread:    3
total:    3
output:  'ABC'
```

```
==ITERATION 2==
nread = read(in_fd,buf,3);
buf[nread] = '\0'
total+= nread;
printf("read: '%s'\n",buf);

data.txt: ABCDEFGHIJ\n
position:       ^
buf:     |D E F \0|
          0 1 2 3
nread:    3
total:    6
output:  'DEF'
```

```
==ITERATION 3==
nread = read(in_fd,buf,3);
buf[nread] = '\0'
total+= nread;
printf("read: '%s'\n",buf);

data.txt: ABCDEFGHIJ\n
position:          ^
buf:     |G H I \0|
          0 1 2 3
nread:    3
total:    9
output:  'GHI'
```

```
==ITERATION 4==
nread = read(in_fd,buf,3);
buf[nread] = '\0'
total+= nread;
printf("read: '%s'\n",buf);

data.txt: ABCDEFGHIJ\n
position:             ^
buf:     |J \n\0\0|
          0 1 2 3
nread:    2
total:    11
output:   'J\n'
```

```
==ITERATION 5==
nread = read(in_fd,buf,3);
if(nread == 0){
  break;
}

data.txt: ABCDEFGHIJ\n
position:             ^
buf:     |J \n\0\0|
          0 1 2 3
nread:    0
total:    11
output:   11 bytes total
```

Answers: Behavior of `read()` in `count_bytes.c`

Take-Aways from `count_bytes.c` include

- OS maintains **file positions** for each open File Descriptor
- I/O functions like `read()` use/change position **in a file**
- `read()`'ing into program arrays overwrites data there
- OS **does not** update positions in user arrays: programmer must do this in their program logic
- `read()` returns # of bytes read, may be less than requested
- `read()` returns 0 when at end of a file

Exercise: write() to File Descriptors

```c
#define SIZE 128

{
  int out_fd = open(out_name, O_WRONLY);
  char buffer[SIZE];
  int bytes_written = write(out_fd, buffer, SIZE);
}
```

- Write up to SIZE bytes to open file descriptor
- Bytes taken from buffer, leave it intact
- Return value is number of bytes written, -1 for error

Questions on write_then_read.c

- Compile and Run
- Explain Output, differences between write() / printf()

Answers: `write()` to File Descriptors

```
> gcc write_then_read.c
> ./a.out
0. Recreating empty existing.txt
1. Opening file existing.txt for writing
2. Writing to file existing.txt
3. Wrote 128 bytes to existing.txt
4. Opening existing.txt for reading
5. Reading up to 128 bytes from existing.txt
6. Read 127 chars, printf()'ing:
here is some text to write
7. printf()'ing 127 characters individually
here is some text to write\0\0\0hello\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
8. write()'ing 127 characters to screen
here is some text to write^@^@^@hello^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
```

read()/write() work with bytes

- In C, general correspondence between byte and the char type
- Not so for other types: int is often 4 bytes
- Requires care with non-char types
- All calls read/write actual bytes

```
#define COUNT 16
int out_ints[COUNT];               // array of 16 integers
int bufsize = sizeof(int)*COUNT;   // size in bytes of array
...;
write(out_fd, out_ints, bufsize);  // write whole buffer

int in_ints[COUNT];
...;
read(in_fd, in_ints, bufsize);     // read to capacity of in_ints
```

Questions

- Examine write_read_ints.c, compile/run
- Examine contents of integers.dat
- Explain what you see

Standard File Descriptors

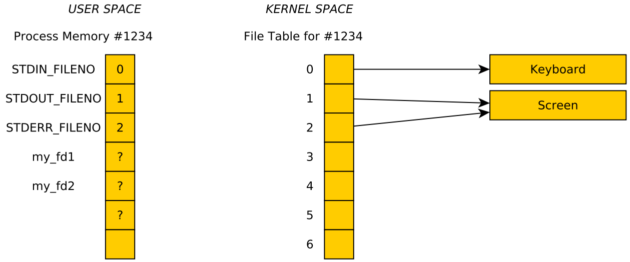
- When a process is born, comes with 3 open file descriptors
- Related to `FILE*` streams in Standard C I/O library
- Traditionally have FD values given but use the Symbolic name to be safe

Symbol	#	FILE*	FD for...
STDIN_FILENO	0	stdin	standard input (keyboard)
STDOUT_FILENO	1	stdout	standard output (screen)
STDERR_FILENO	2	stderr	standard error (screen)

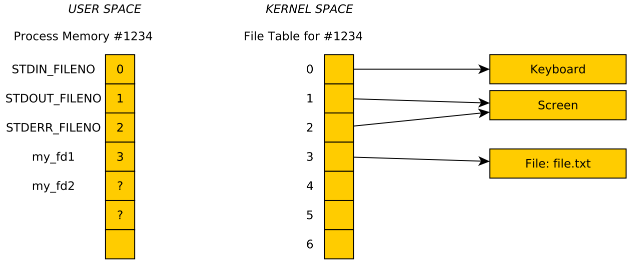
```c
// Low level printing to the screen
char message[] = "Wubba lubba dub dub!\n";
int length = strlen(message);
write(STDOUT_FILENO, message, length);
```

See `low_level_interactions.c` to gain an appreciation for what `printf()` and its kin can do for you.

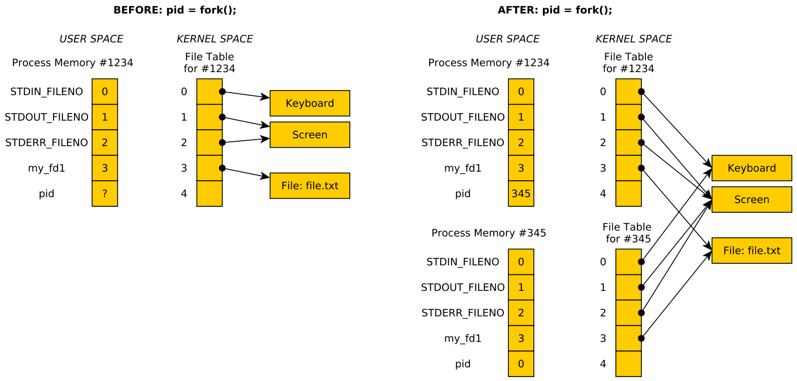
File Descriptors refer to Kernel Structures



my_fd1 = open("file.txt",O_RDONLY);



Processes Inherit Open FDs: Diagram



Typical sequence:

- Parent creates an `output_fd` and/or `input_fd`
- Call `fork()`
- Child changes standard output to `output_fd` and/or `input_fd`
- Changing means calls to `dup2()`

Shell I/O Redirection

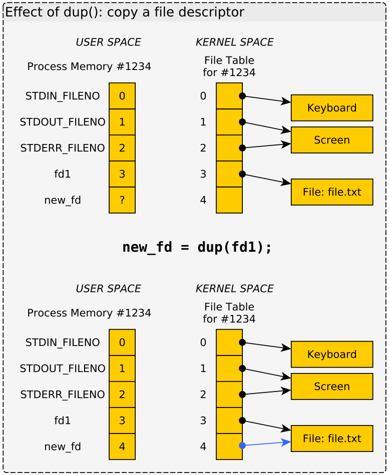
- Shells can direct input / output for programs using < and >
- Most common conventions are as follows

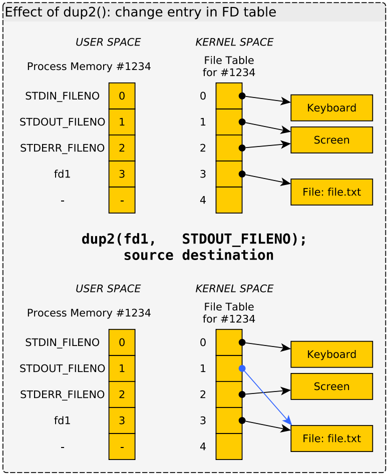
```
$> some_program > output.txt
# output redirection to output.txt

$> interactive_prog < input.txt
# read from input.txt rather than typing

$> some_program &> everthing.txt
# both stdout and stderr to file

$> some_program 2> /dev/null
# stderr silenced, stdout normal
```

- Long output can be saved easily
- Can save typing input over and over
- Even more fun when you incorporate Pipes to make Pipelines
- **Goal:** Demonstrate systems calls to facilitate redirection

Manipulating the File Descriptor Table

- System calls `dup()` and `dup2()` manipulate the FD table
- `int backup_fd = dup(fd);` : copy a file descriptor
- `dup2(src_fd, dest_fd);` : `src_fd` copied to `dest_fd`





Exercise: Redirecting Output with `dup()` / `dup2()`

- `dup()`, `dup2()`, and `fork()` can be combined in interesting ways
- **Diagram fork-dup.pdf** shows how to redirect standard out to a file like a shell does in: `ls -l > output.txt`

Write a program which

1. Prints PID to screen
2. Opens a file named `write.txt`
3. Forks a Child process
4. Child: **redirect standard output** into `write.txt`

   Parent: does no redirection
5. Both: `printf()` their PID
6. Child: **restore** standard output to screen

   Parent: makes no changes
7. Both: `printf()` "All done"

```
> gcc duped_child.c

> ./a.out
BEGIN: Process 1913588
MIDDLE: Process 1913588
END: Process 1913588 All done
END: Process 1913590 All done

> cat write.txt
MIDDLE: Process 1913590
```

Answers: Redirecting Output with dup() / dup2()

```c
// duped_chld.c: solution to in-class activity on redirecting output
// in child process.
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char *argv[]){
  system("echo '' > write.txt");       // ensure file exists, is empty
  printf("BEGIN: Process %d\n",getpid());
  int fd = open("write.txt",O_WRONLY); // open a file
  int backup;
  pid_t child = fork();                // fork a child, inherits open file
  if(child == 0){                      // child only redirects stdout
    backup = dup(STDOUT_FILENO);       // make backup of stdout
    dup2(fd,STDOUT_FILENO);            // dup2() alters stdout so child printf() goes into file
  }
  printf("MIDDLE: Process %d\n",getpid());
  if(child == 0){
    dup2(backup,STDOUT_FILENO);        // child restores stdout
  }
  printf("END: Process %d All done\n",getpid());
  close(fd);
  if(child != 0){                      // parent waits on child
    wait(NULL);
  }
  return 0;
}
```

C `FILE` Structs Use File Descriptors in UNIX

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

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

From `/usr/include/bits/types/struct_FILE.h`

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

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

```
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...
...
```

```c
// mixed_std_low.c: mix C Standard
// and Unix I/O calls. pain++;
#include <stdio.h>
#include <unistd.h>

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;
}
```

Sample compile/run:

```
> gcc mixed_std_low.c
> ./a.out
FIRST: 1
NEXT: 41 1042 1043 1044 1045...
```

- Explain output of program given input file
- Use knowledge that **buffering** occurs internally for standard I/O library

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

- C standard I/O calls like `printf / fprintf()` and `scanf() / fscanf()` use internal buffering
- A call to `fscanf(file, "%d", &x)` will read a large chunk from a file but only process part of it
- From OS perspective, associated file descriptor has advanced forwards / read a bunch
- The data is in a hidden "buffer" associated with a `FILE *file`, used by `fscanf()`

Output Also buffered, Always `fclose()`

- Output is also buffered: `output_buffering.c`
- Output may be lost if `FILE*` are not `fclose()`'d: closing will flush remaining output into a file
- See `fail_to_write.c`
- File descriptors always get flushed out by OS when a program ends BUT `FILE*` requires user action

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);
```

Above functions change buffering behavior of standard C I/O
Examples:

```
// 1. Set full "block" buffering for stdout, use outbuf
#define BUFSIZE 64
char outbuf[BUFSIZE] = {};
setvbuf(stdout, outbuf, _IOFBF, BUFSIZE);

// 2. Turn off buffering of stdout, output immediately printed
setvbuf(stdout, NULL, _IONBF, 0);
```

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

Attributes of Files from `stat()`

`stat_demo.c` shows some attributes that may be obtained about a file after a call to `stat(filename, &statbuf)` which fills in the `statbuff` struct. Attributes include:

Attribute	Notes
Size	In bytes via `st_size` field
File Type	Via `st_mode` field and macros like `S_ISREG(mode)` Limited number of fundamental types: regular, directory, socket, etc.
Permissions	Read/Write/Execute for Owner/Group/Others via `st_mode` field
Ownership	Via `st_uid` (user) and `st_gid` (group), numeric IDs for both
Time Data	Access / Change / Modification times via `st_atime, st_ctime, ...`

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 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

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);
                                     ^^^^^
```

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
`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
`ftruncate(fd, 64);`	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()`

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

Optional Exercise: 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

```c
// 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 (read about this on your own if curious about systems programming)

Extras: Processes Inherit Open FDs

- Child processes share all open file descriptors with parents
- By default, Child prints to screen / reads from keyboard input
- Redirection requires manipulation prior to `fork()`
- See: `open_fork.c`
- Experiment with order
  1. `open()` then `fork()`
  2. `fork()` then `open()`

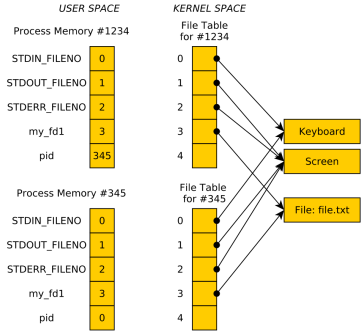
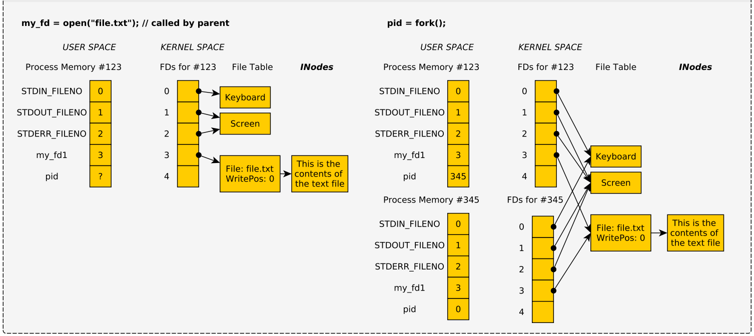
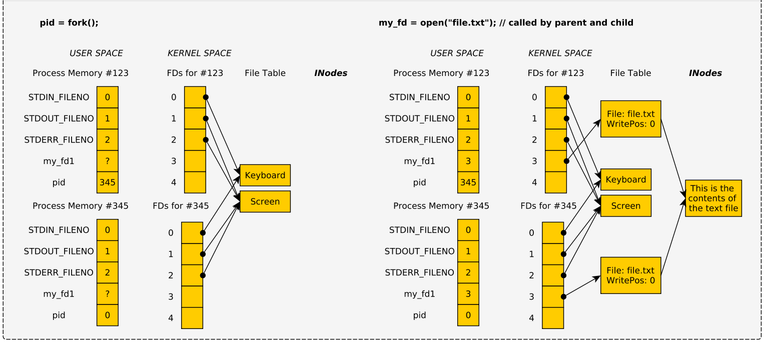


Diagram on next slide shows variations of open-then-fork vs fork-then-open from `open_fork.c` open() normal file then call fork()



fork() then call open() normal file



(Review) 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...