CMSC 216: UNIX File Input/Output

Chris Kauffman

Last Updated:
Mon Nov 10 01:11:30 PM EST 2025

Logistics

Announcements

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`

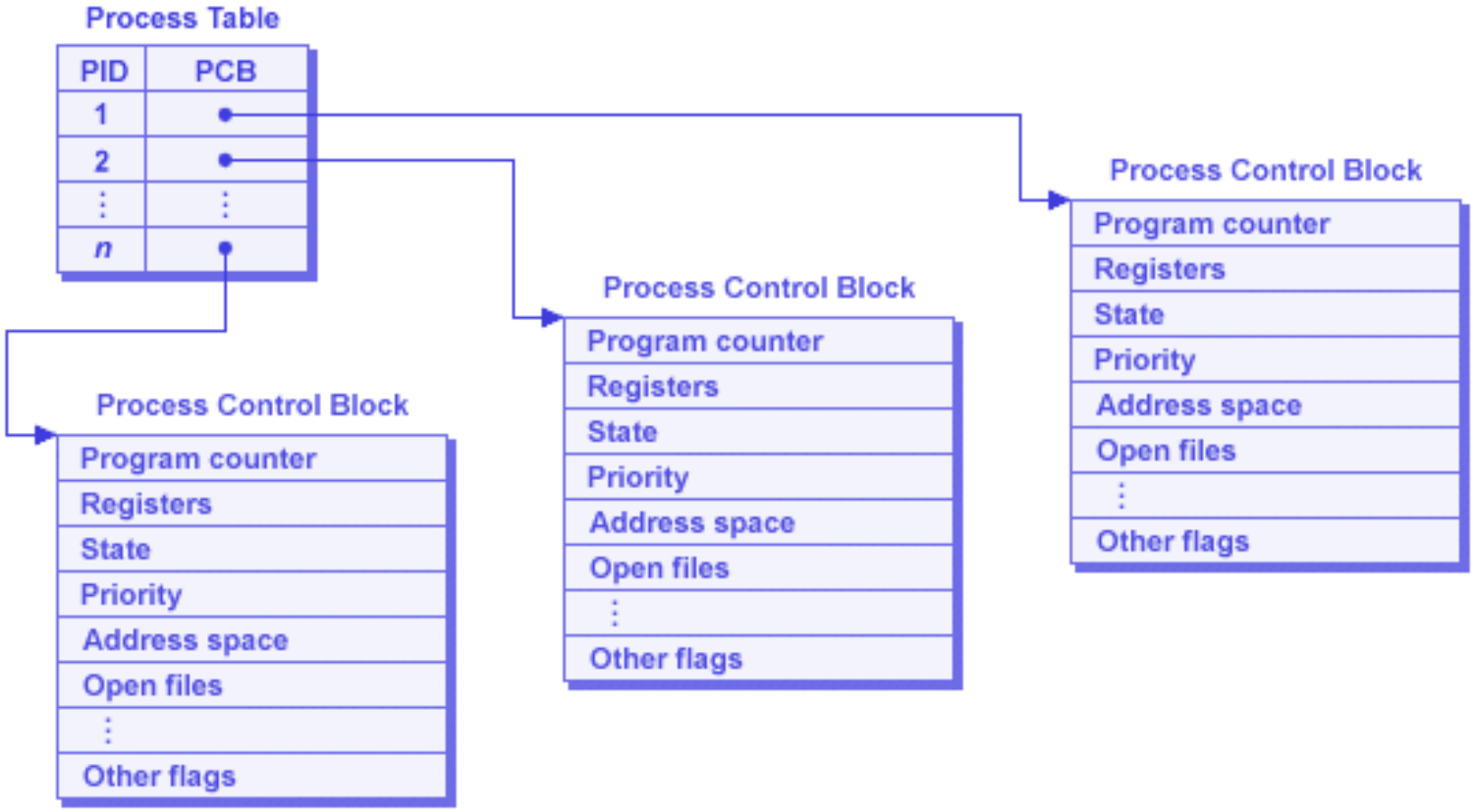
1	`printf("%d is a number",5);`	Printing things to the screen?
2	`FILE *file = fopen("myfile.txt","r");`	Opening a file?
3	`fclose(file);`	Close a file?
4	`fprintf(file,"%d is a number",5);`	Printing to a file?
5	`scanf("%d %f",&myint,&mydouble);`	Scanning from terminal
	`fscanf(file2,"%d %f",&myint,&mydouble);`	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

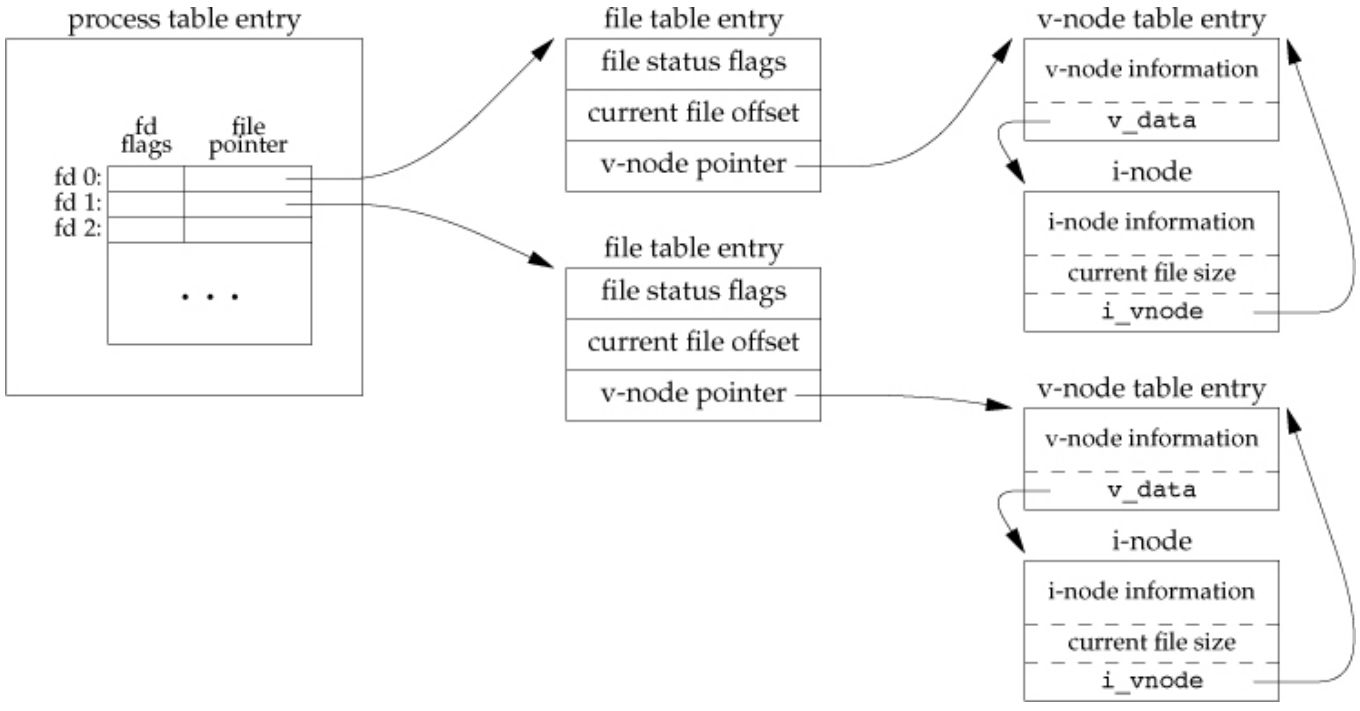
The Process Table



Source: SO What is the Linux Process Table?

- OS maintains data on all processes in a Process Table
- Process Table Entry $\approx$ 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
  <MINTED>
- FD Table entry refers to other Kernel/OS data structures

File Descriptors (FDs) 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
- FDs allow interaction with "normal" files like `myfile.txt` or `commando.c` to read/change them
- FDs 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

<MINTED>

`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

<MINTED>

- 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`

<MINTED>

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

<MINTED>

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

<MINTED>

<MINTED>

<MINTED>

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

<MINTED>

- 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

<MINTED>

`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

<MINTED>

Questions

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

Standard File Descriptors

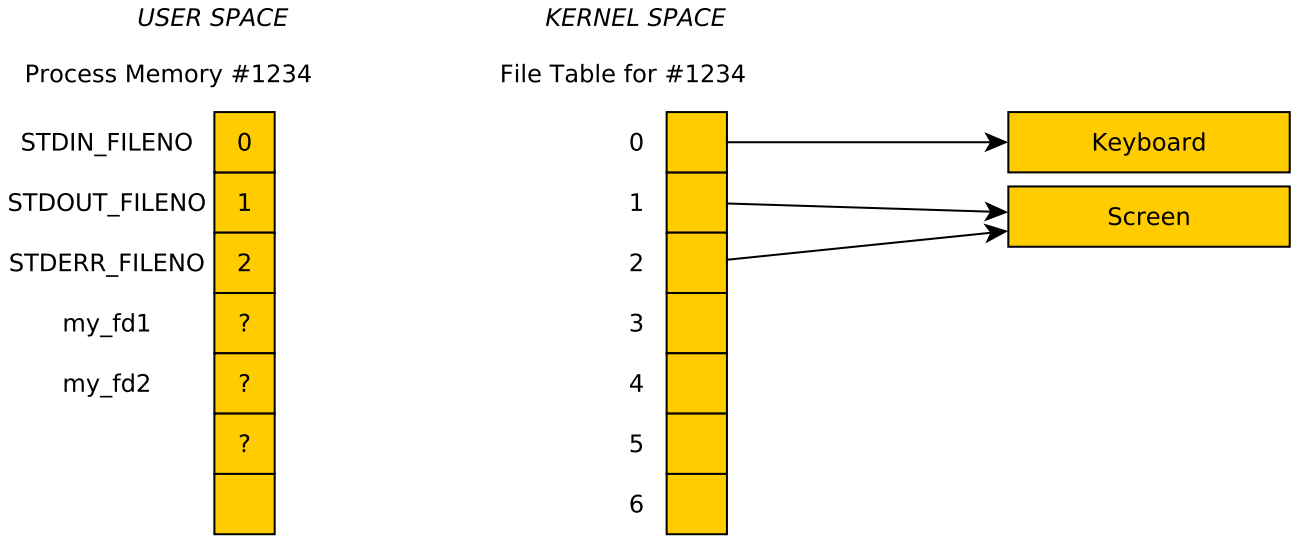
- When a process is born, comes with 3 open file descriptors
- Related to `FILE*` streams in Standard C I/O library
- Traditionally have FD values given but use the Symbolic name to be safe

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)

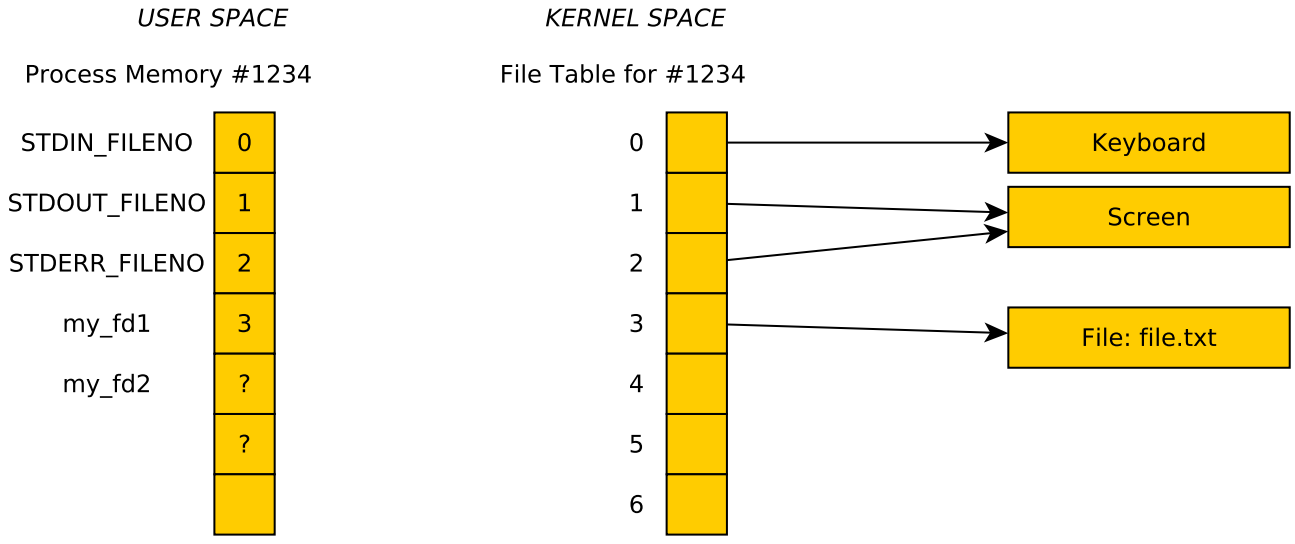
<MINTED>

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

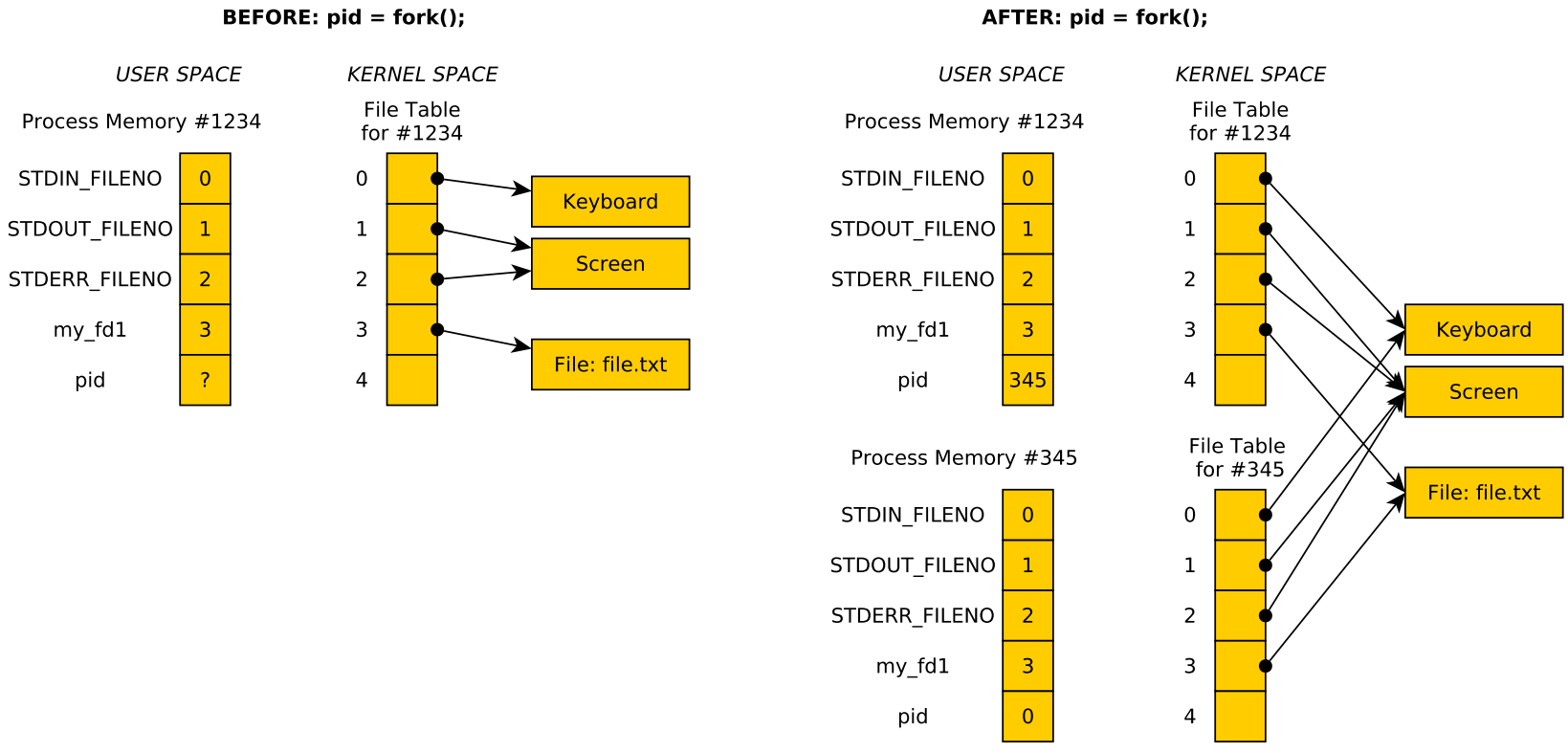
File Descriptors refer to Kernel Structures



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



Processes Inherit Open FDs: Diagram



Typical sequence:

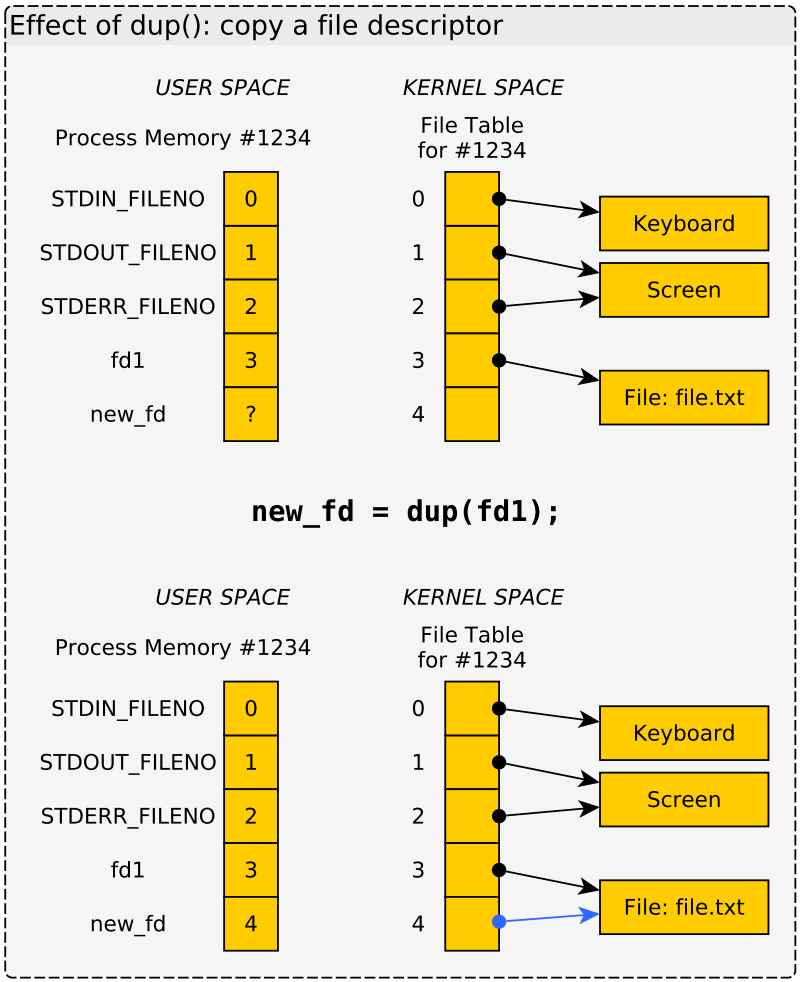
- Parent creates an `output_fd` and/or `input_fd`
- Call `fork()`
- Child changes standard output to `output_fd` and/or `input_fd`
- Changing means calls to `dup2()`

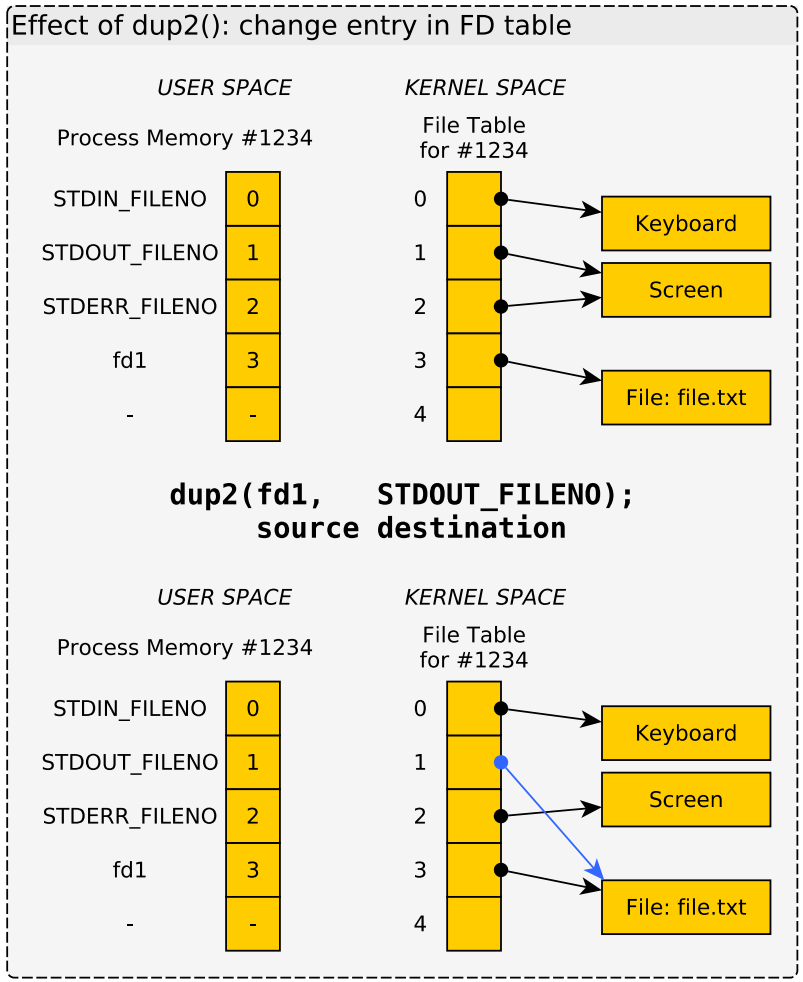
Shell I/O Redirection

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

   <MINTED>
5. Both: `printf()` their PID
6. Child: **restore** standard output to screen

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

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

<MINTED>

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`

<MINTED>

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

<MINTED>

<MINTED>

Sample compile/run:

<MINTED>

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

Answers: Subtleties of Mixing Standard / 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
- To force output, use `fflush(some_file);`

Controlling `FILE` Buffering

<MINTED>

Above functions change buffering behavior of standard C I/O

Examples:

<MINTED>

- When testing lab/project code, buffering is disabled as it makes it easier to understand some bugs

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

<MINTED>

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

<MINTED>

- Every file has permissions set from somewhere on creation

Changing Permissions

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

`<MINTED>`

- `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

<MINTED>

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`

<MINTED>

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

<MINTED>

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

Answers: Sketch Code for Total Size of Regular Files

<MINTED>

- 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()`

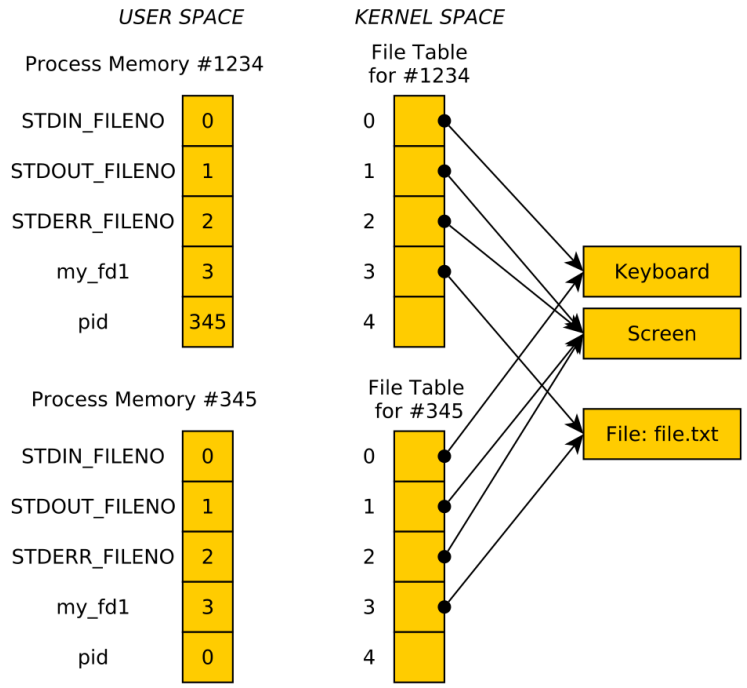
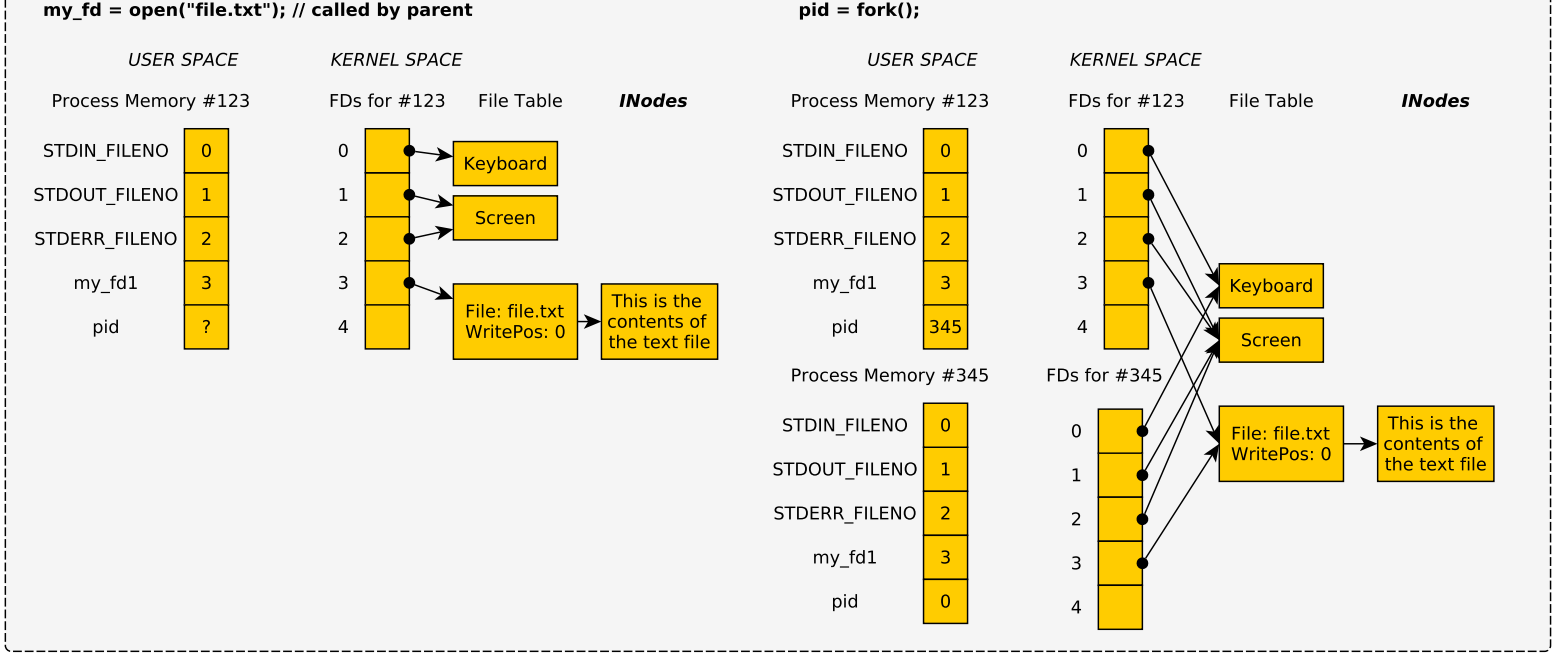
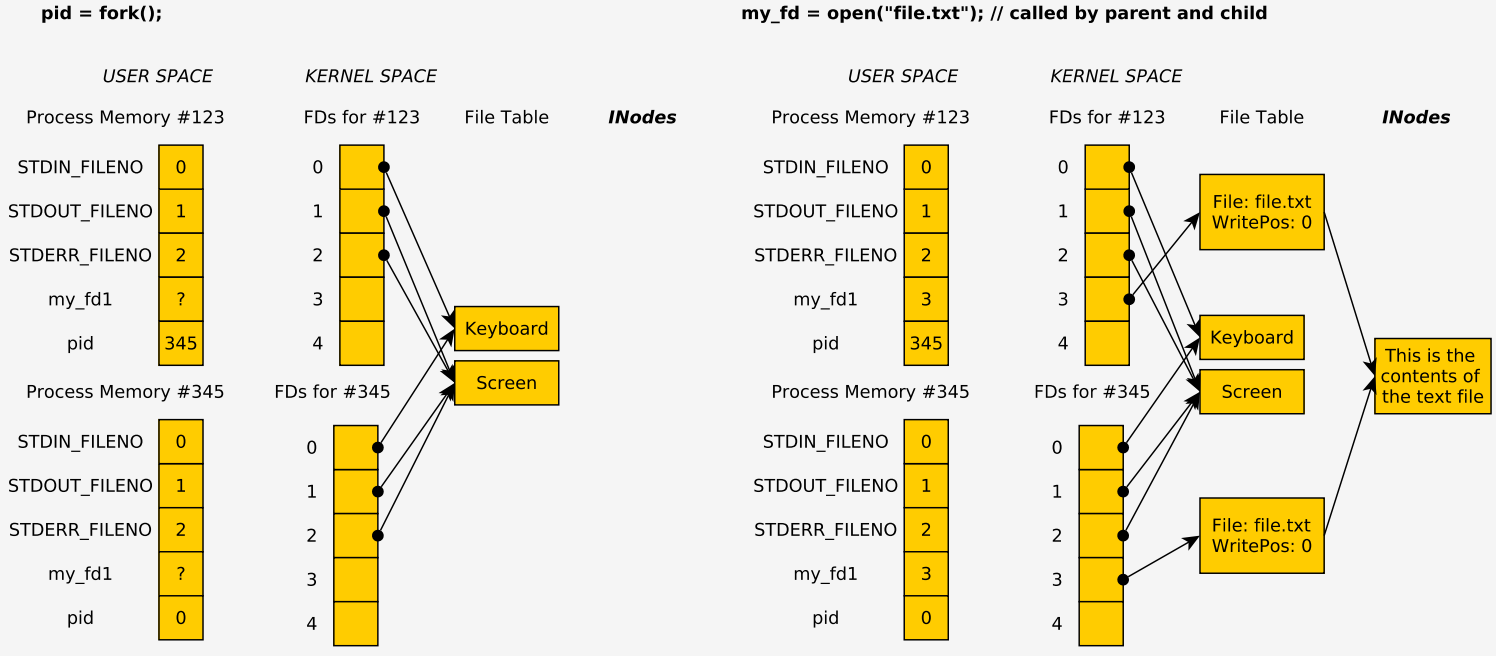


Diagram on next slide shows variations of open-then-fork vs fork-then-open from `open_fork.c`

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...