Additional I/O Functions

COMP 321

Dave Johnson

What We've Done

Earlier, we covered basics of system-level I/O talking directly to the kernel

- Opening or creating a file return a file descriptor
- The current offset (or position) in the file of a file descriptor
- Reading and writing a file using a file descriptor
- Explicitly moving the file descriptor offset with lseek
- Closing a file descriptor
- Getting information with stat and fstat
- The effect of fork on file descriptors
- Using mmap to map a file's data into virtual memory
- Sockets and computer networking basics
- I/O multiplexing and non-blocking I/O

New This Time

A few additional I/O kernel calls

- Some I/O kernel calls needed, for example, by the shell
  - I/O redirection
  - pipes

Some details on library-level I/O

- Using Standard I/O
- Some basics of the implementation of Standard I/O

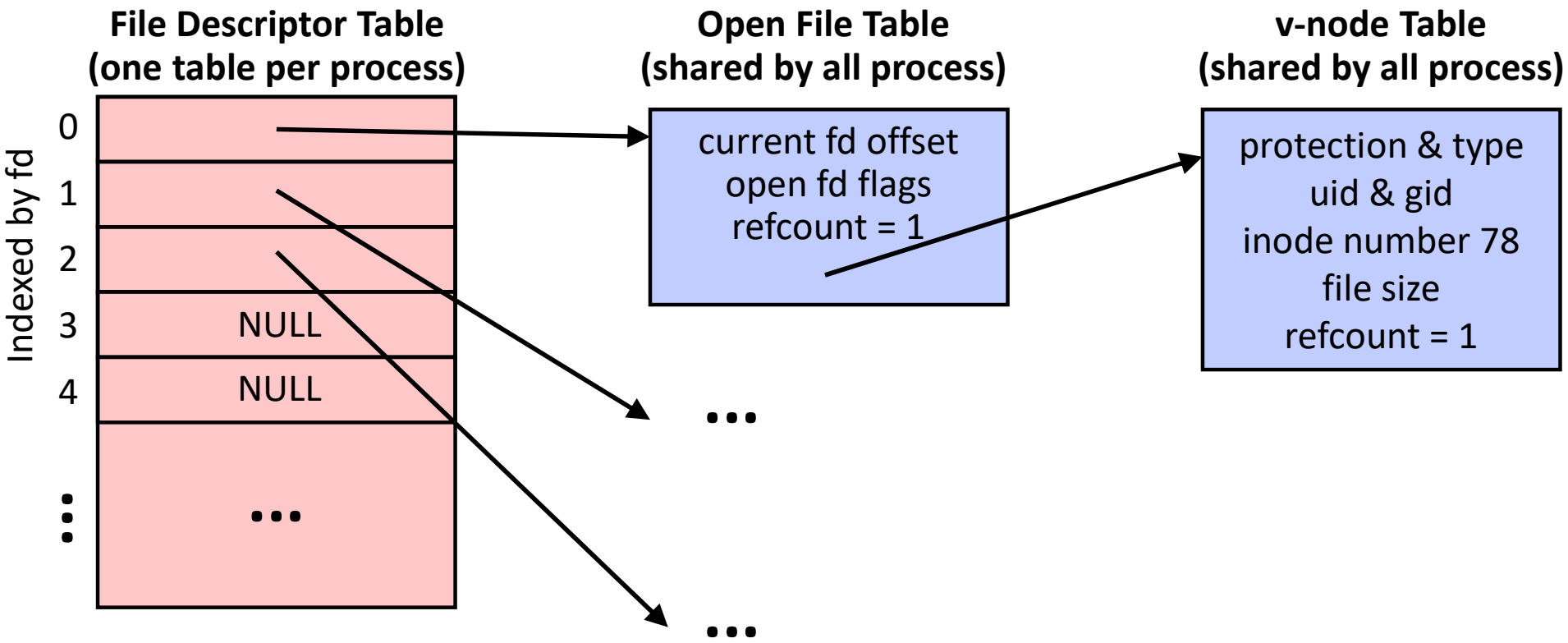
Duplicating a File Descriptor

```
int  dup(int  oldfd);
int  dup2(int  oldfd,  int  newfd);
```

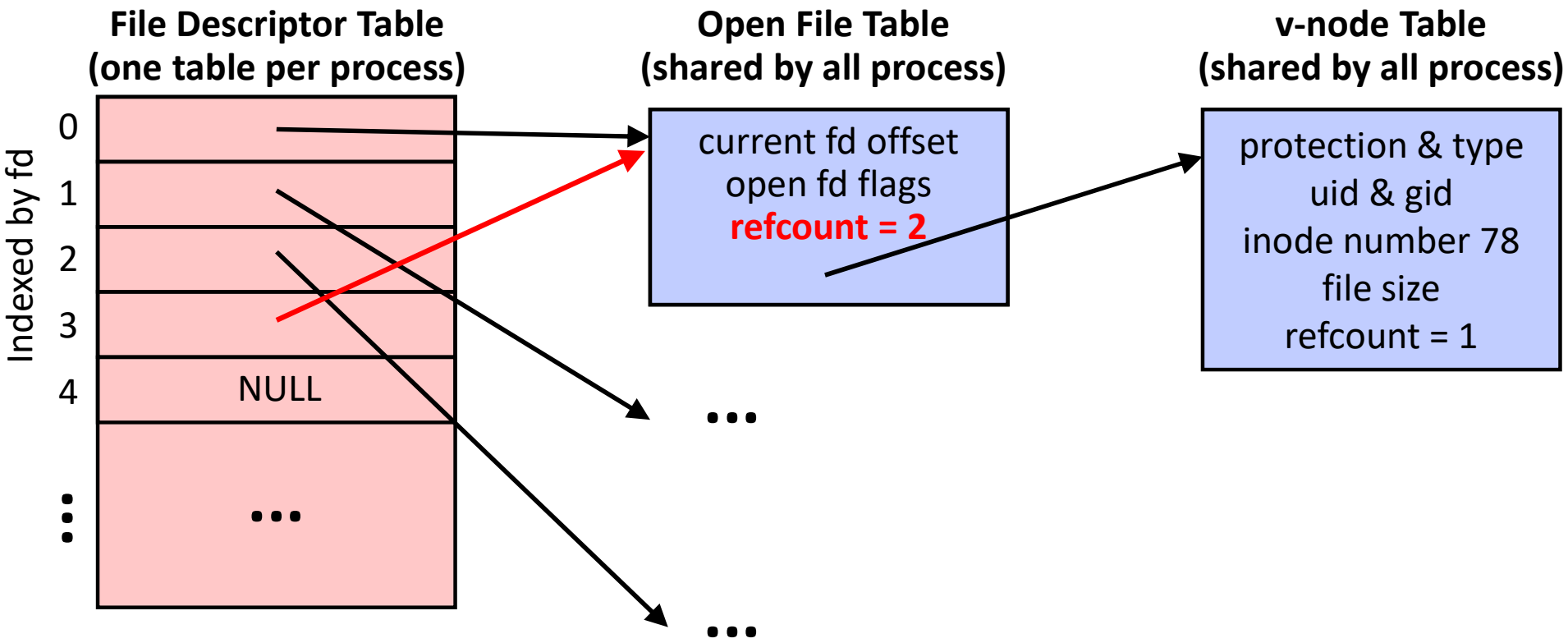
Assigns a new (additional) descriptor number to existing open file instance

- dup() returns the ***lowest numbered*** file descriptor number that is not currently open in this process to something else – like open() does
- dup2() instead uses the specified newfd file descriptor number
  - if newfd is already open, it is automatically closed first
- On return, both old and new file descriptors refer to the same ***shared*** open file instance

The Kernel Data Structures – Doing dup(0)



The Kernel Data Structures – Doing dup(0)



Using dup() or dup2() in the Shell

Example: redirecting standard output (command > file) the <u>unsafe</u> way

```
close(STDOUT_FILENO);
open(file, O_WRONLY);
```

- Doing open() will pick the lowest unused descriptor number, which here should be STDOUT_FILENO
- But for a time, you have no open standard output file!

Doing it the correct, safe way

```
newfd = open(file, O_WRONLY);
dup2(newfd, STDOUT_FILENO);
close(newfd);
```

- dup2() closes old STDOUT_FILENO; then we close unneeded newfd

Creating a Process Pipeline

```
int pipefd[2];
int pipe(pipefd);
```

Creates a "pipe" – a data channel for interprocess communication

- The pipe provides a unidirectional byte stream channel
  - pipefd[0] is a file descriptor number open O_RDONLY to the pipe (the "read end" of the pipe)
  - pipefd[1] is a file descriptor number open O_WRONLY to the same pipe (the "write end" of the pipe)
- Data written to the write end of the pipe is buffered in the kernel . . .
- . . . until read from the read end of the pipe

Example – Running "`ls | wc -l`"

```
int main(void)
{
      int pipefd[2];
      int pid1, pid2;

      pipe(pipefd);
      if ((pid1 = fork()) == 0) {
            dup2(pipefd[1], 1);
            close(pipefd[0]);
            close(pipefd[1]);
            execl("/usr/bin/ls", "ls", NULL);
      }

      If ((pid2 = fork()) == 0) {
            dup2(pipefd[0], 0);
            close(pipefd[0]);
            close(pipefd[1]);
            execl("/usr/bin/wc", "wc", "-l", NULL);
      }
      close(pipefd[0]);
      close(pipefd[1]);
      waitpid(pid1, NULL, 0);
      waitpid(pid2, NULL, 0);
      exit(0);
}
```

- The first child will exec "ls" to write into the pipe as its standard output
- The second child will exec "wc -l" to count the files in the current directory
- It is important to clean up (i.e., close) all unneeded fd's for the pipe

One Way to Understand Pipe Behavior

This is actually the original implementation of pipes up through 4.1 BSD

- Calling pipe(pipefd) is equivalent to

  pipefd[0] = open(create a new pipe file, O_RDONLY);

  pipefd[1] = open(the same pipe file, O_WRONLY);
- Suppose offset0 = the offset on pipefd[0] file descriptor (read end)
- and suppose offset1 = the offset on pipefd[1] file descriptor (write end)
- The kernel makes sure this condition always holds: offset0 ≤ offset1
  - If a read (thus offset0) tries to bypass offset1, the read blocks
  - Once the writing process writes more (advancing offset1), then the reading side is unblocked
- If offset0 == offset1, kernel truncates the pipe and sets offset0 = offset1 = 0
- ***Modern implementations are much more sophisticated (but equivalent)***

End of File on Reading From a Pipe

What should happen when you read from an empty pipe?

- Suppose the first process has written 100 bytes into the pipe . . .
- . . . and the second process has read 100 bytes from the pipe
- The pipe is now empty (no buffered data), but another read from the pipe now ***should not*** be treated as end of file on the pipe
  - The other process might (sometime) still write more data into the pipe
  - The two processes run asynchronously, so sometimes the ordering of reads vs. writes may be different
- Reading from an empty pipe is ***only*** treated as end of file when
  - You read and the pipe is empty, ***and***
  - No file descriptor is open ***anywhere*** that can write into the pipe
- The read then returns 0 = the number of bytes read

Special Rules for Writing Into a Pipe

Writing into a full pipe is not a problem

- The kernel will only buffer up to some limited number of bytes
- A write into the pipe then just blocks the writing process until after the reading process has read some

What about a write into the pipe when no process can read it?

- Any write (or attempted write) into the pipe is an error
  - If no file descriptor is open ***anywhere*** that can read from the pipe
- Causes a SIGPIPE signal in the writing process (default: terminates process)
- If SIGPIPE is ignored, or if it is caught but the signal handler returns, the write to the pipe returns -1 with errno = EPIPE

The Standard I/O Library

Defined for many types of systems, not just for Unix/Linux

- A library that takes care of the details of system-level I/O
- Provides a (generally) simpler interface for I/O
- And (generally) takes into account system-level details to do the necessary system-level I/O more efficiently (e.g., buffering and block sizes)
- But at the expense of another layer of software overhead to go through
- Interface is generally defined by #include <stdio.h>
- But that only defines the interface, the real procedures are in the C library

The Interface uses a "FILE *" Not an Integer fd

A "FILE *" represents a Standard I/O "stream"

- A "FILE *" is really just a pointer to a "FILE"
  - A "FILE" is usually implemented as a C typedef for some struct
  - (But it could be #define FILE struct whatever)
- Everything that Standard I/O needs to keep track of about that open stream is stored inside that struct
  - Passing the address of this struct to each Standard I/O operation on that stream lets the implementation access the contents of that struct
- stdin, stdout, and stderr are predefined pointers to 3 of those structs
- You generally don't have any reason to ever actually look at that struct
- (But I will talk a bit about it later to illustrate how Standard I/O works)

Opening a Standard I/O Stream

```
FILE  *fopen(const char  *restrict pathname,  const char  *restrict mode);
FILE  *freopen(const char  *restrict pathname,  const char  *restrict mode,
        FILE  *restrict stream);
FILE  *fdopen(int  fd,  const char  *mode);
```

- fopen() opens the pathname file for the given mode
- freopen() uses the existing stream but reopens as a new pathname
  - If pathname is NULL, uses the same pathname
  - The new mode is given by mode
- fdopen() opens a new stream on something already open as descriptor fd
- mode is either “r” (file must exist) or “w” (creates if the file doesn’t exist)
  - or many other options on different systems (see “man 3 fopen”)

Types of Standard I/O Buffering

Standard I/O offers choice of three types of buffering for each stream

- Fully buffered
  - A stream has some fixed buffer size
  - Output flushes by calling kernel I/O when the buffer fills up
  - Input calls the kernel for more data when the buffer is used up
- Line buffered
  - The buffer is flushed when outputting a newline character
  - Or (of course) when the fixed-sized buffer fills up
  - This is the default for a terminal
- Unbuffered
  - Every I/O operation immediately calls the kernel to do it

How Can Standard I/O Know If This Is a Terminal?

```
int  isatty(int  fd);
```

The C library provides the isatty() function to check for this

- Returns 1 if fd is open to a terminal device, or 0 otherwise

But how can isatty() figure out if fd is open to a terminal?

- Remember tcgetpgrp() and tcsetpgrp() to get/set terminal process group
  - Really does kernel ioctl TIOCGPGRP or TIOCSPGRP on that fd
- The tcgetattr() library call gets the terminal attributes for the given fd
  - Really does kernel ioctl TIOCGETA on that fd
  - ***Returns -1 with errno = ENOTTY if fd is <u>not</u> open to a terminal!***

Setting the Buffering for a Stream

```
void  setlinebuf(FILE  *stream);
void  setbuf(FILE  *restrict stream,  char  *restrict buf);
int  setvbuf(FILE  *restrict stream,  char  buf[restrict .size],
         int mode,  size_t size);
```

Can explicitly set the buffering of a stream after opening it

- setlinebuf() sets the stream to be line buffered
- setbuf() sets unbuffered (NULL) or buffered (pointer to BUFSIZE buffer)
- setvbuf() lets you also specify the buffer size, with mode of
  - _IONBF = unbuffered
  - _IOLBF = line buffered
  - _IOFBF = fully buffered

A Simplified (But Real, But Old) Definition of FILE

From Version 7 Unix (more recent versions are much more sophisticated)

```
extern  struct  _iobuf {
    char    *_ptr;          // address for next character in the buffer
    int     _cnt;           // count of remaining characters in the buffer
    char    *_base;         // address of the buffer (of size BUFSIZ)
    char    _flag;          // read and/or write, buffering mode, etc.
    char    _file;          // the underlying kernel file descriptor number
};

#define FILE    struct _iobuf

#define fileno(p)      p->_file
```

Example Parts of Standard I/O Implementation

Defined in /usr/include/stdio.h (this is from Version 7 Unix)

```
#define getc(p)         (--(p)->_cnt>=0? *(p)->_ptr++&0377:_filbuf(p))

#define getchar()       getc(stdin)

#define putc(x,p)       (--(p)->_cnt>=0?
        ((int)(*(p)->_ptr++=(unsigned)(x))):_flsbuf((unsigned)(x),p))

#define putchar(x)      putc(x,stdout)
```