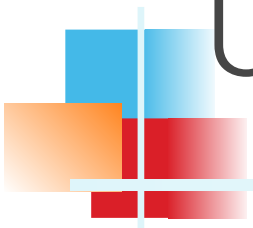
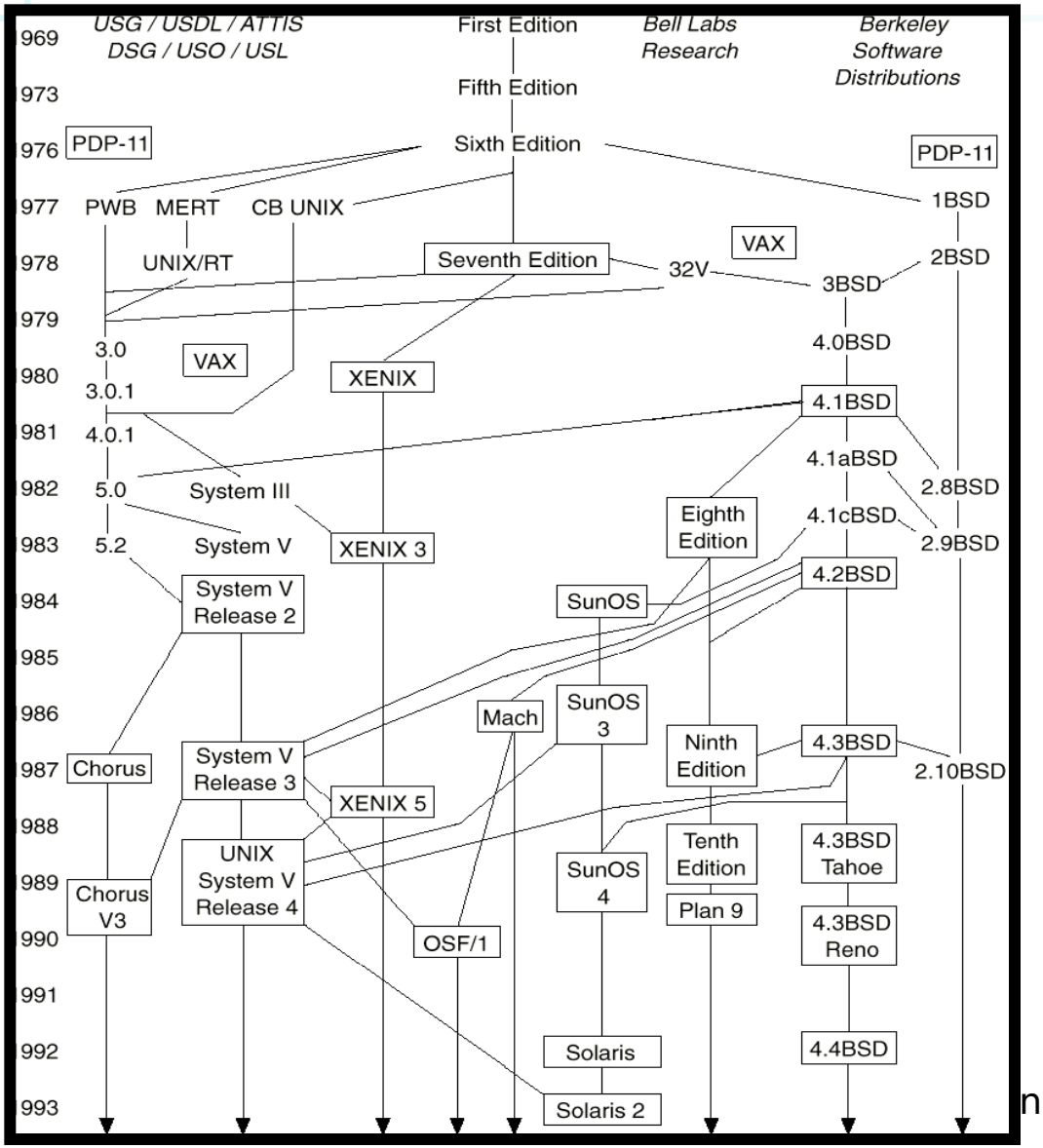


UNIX System Overview

History of UNIX



- Adopted from Operating System Concepts
- by Abraham Silberschatz et al.

For a full history, refer to http://www.levenez.com/unix

ng

General Characteristics of UNIX

Multi-user & Multi-tasking - most versions of UNIX are capable of allowing multiple users to log onto the system, and have each run multiple tasks. This is standard for most modern OSs.

Over 30 Years Old - UNIX is over 30 years old and it's popularity and use is still high. Over these years, many variations have spawned off and many have died off, but most modern UNIX systems can be traced back to the original versions. It has endured the test of time. For reference, Windows at best is half as old (Windows 1.0 was released in the mid 80s, but it was not stable or very complete until the 3.x family, which was released in the early 90s).

Large Number of Applications – there are an enormous amount of applications available for UNIX operating systems. They range from commercial applications such as CAD, Maya, WordPerfect, to many free applications.

Free Applications and Even a Free Operating System - of all of the applications available under UNIX, many of them are free. The compilers and interpreters that we use in most of the programming courses here at UMBC can be downloaded free of charge. Most of the development that we do in programming courses is done under the Linux OS.

Less Resource Intensive - in general, most UNIX installations tend to be much less demanding on system resources. In many cases, the old family computer that can barely run Windows is more than sufficient to run the latest version of Linux.

Internet Development - Much of the backbone of the Internet is run by UNIX servers. Many of the more general web servers run UNIX with the Apache web server - another free application.

Files and Directories

- UNIX file system: a hierarchical arrangement of directories and files
  - File
  - A directory is a **file** that contains directory entries

Files

- **A Linux *file* is a sequence of *m* bytes:**
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$

- **Cool fact: All I/O devices are represented as files:**
  - **`/dev/sda2`** (**`/usr`** disk partition)
  - **`/dev/tty2`** (terminal)

- **Even the kernel is represented as a file:**
  - **`/boot/vmlinuz-3.13.0-55-generic`** (kernel image)
  - **`/proc`** (kernel data structures)

File Types

- **Each file has a *type* indicating its role in the system**
  - *Regular file:* Contains arbitrary data
  - *Directory:* Index for a related group of files
  - *Socket:* For communicating with a process on another machine

- **Other file types (later)**
  - *Named pipes (FIFOs)*
  - *Symbolic links*
  - *Character and block devices*

Regular Files

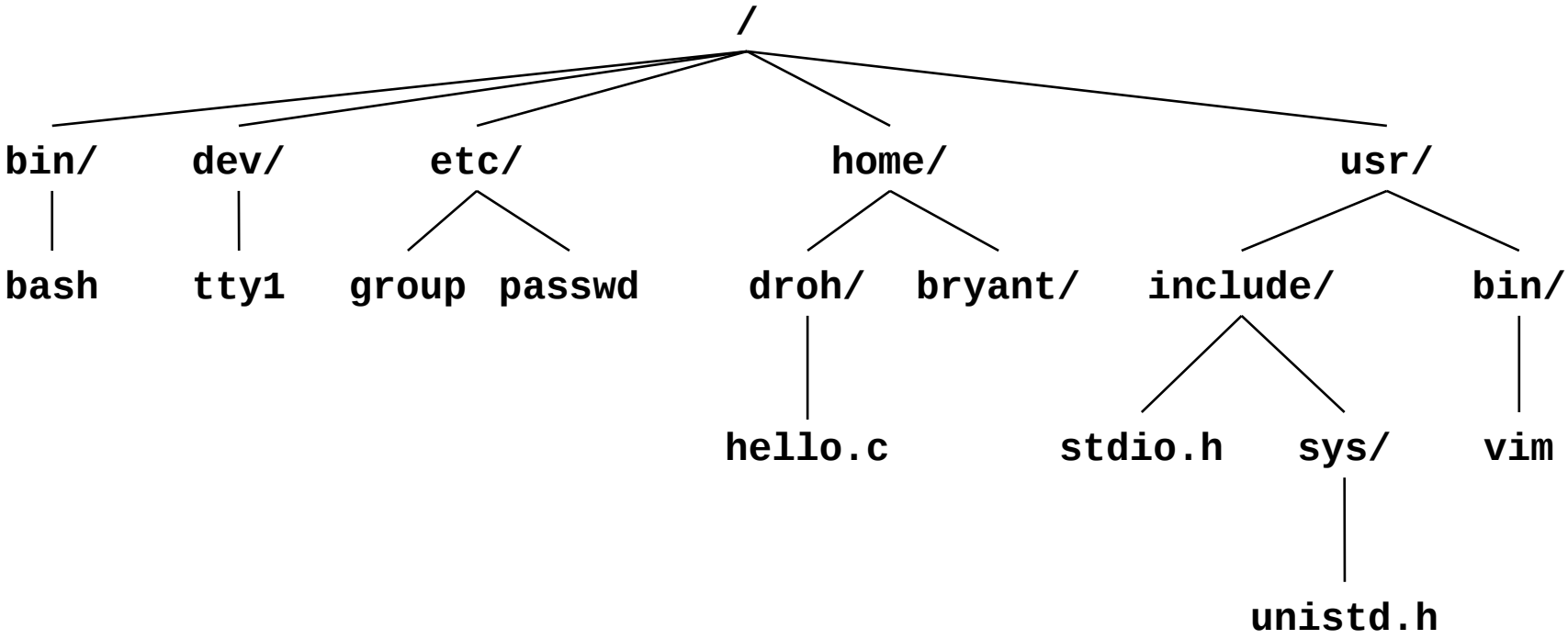
- **A regular file contains arbitrary data**
- **Applications often distinguish between *text files* and *binary files***
  - Text files are regular files with only ASCII or Unicode characters
  - Binary files are everything else
    - e.g., object files, JPEG images
  - Kernel doesn't know the difference!
- **Text file is sequence of *text lines***
  - Text line is sequence of chars terminated by *newline char* ('\n')
    - Newline is 0xa, same as ASCII line feed character (LF)
- **End of line (EOL) indicators in other systems**
  - Linux and Mac OS: '\n' (0xa)
    - line feed (LF)
  - Windows and Internet protocols: '\r\n' (0xd 0xa)
    - Carriage return (CR) followed by line feed (LF)

Directories

- **Directory consists of an array of *links***
  - Each link maps a *filename* to a file
- **Each directory contains at least two entries**
  - `.` (dot) is a link to itself
  - `..` (dot dot) is a link to *the parent directory* in the *directory hierarchy* (next slide)
- **Commands for manipulating directories**
  - `mkdir`: create empty directory
  - `ls`: view directory contents
  - `rmdir`: delete empty directory

Directory Hierarchy

- **All files are organized as a hierarchy anchored by root directory named / (slash)**

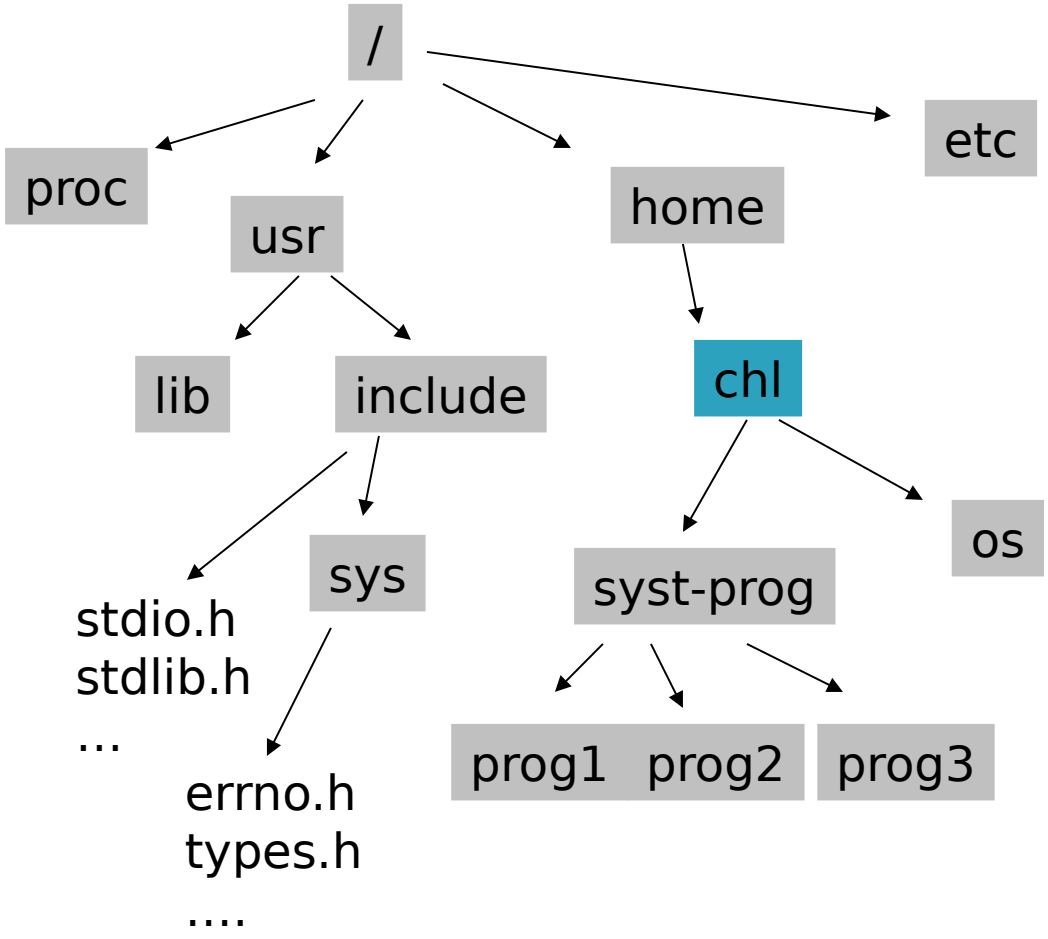


- **Kernel maintains *current working directory (cwd)* for each process**
  - Modified using the cd command

Files and Directories

- Filename
  - At least 255 character filenames (Most commercial UNIX)
  - Special names: “/”, “.”, and “..”
    - The root, current, and parent directories
- Pathname
  - A sequence of zero or more file names, separated by slash, and optionally starting with a slash
  - Absolute pathname vs. relative pathname
    - /home/taesoo/Documents
    - ./Documents
- Current Working Directory
  - “.”
  - pwd

UNIX File System



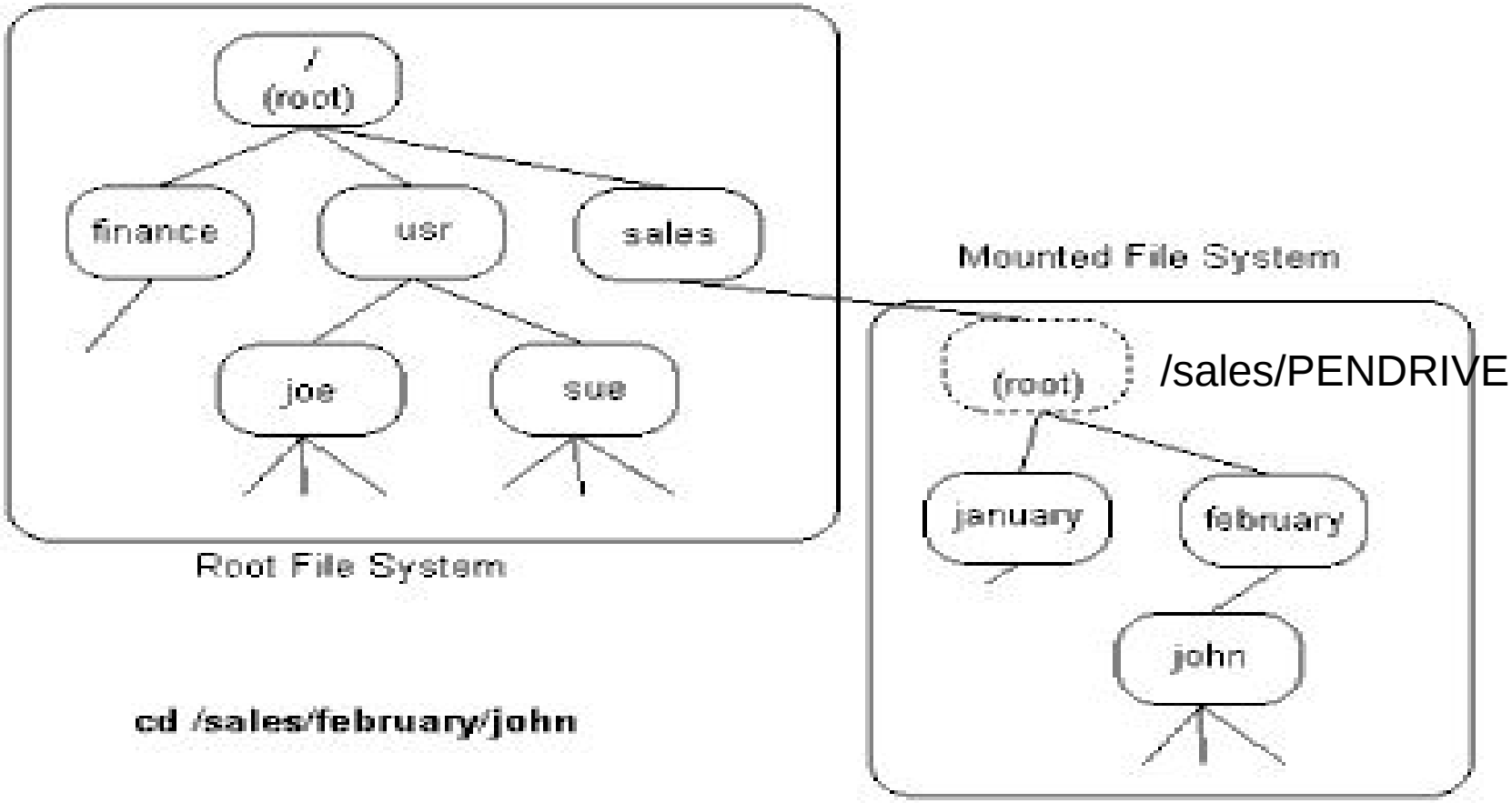
$ **pwd**
/home/chl

$ **ls syst-prog**
prog1 prog2 prog3

$ **ls .././chl/syst-prog**
prog1 prog2 prog3

$ **ls /home/chl/syst-prog**
prog1 prog2 prog3

File System Mounting



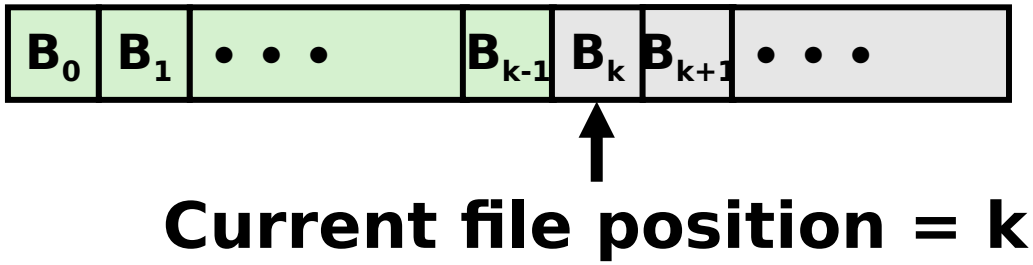
File I/O

시스템 프로그래밍

Division of Computer Science & Engineering
Hanyang University

Unix I/O Overview

- **Elegant mapping of files to devices allows kernel to export simple interface called *Unix I/O:***
  - Opening and closing files
    - **`open()`** and **`close()`**
  - Reading and writing a file
    - **`read()`** and **`write()`**
  - Changing the ***current file position*** (seek)
    - indicates next offset into file to read or write
    - **`lseek()`**



Opening Files

- **Opening a file informs the kernel that you are getting ready to access that file**

```
int fd;   /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
   perror("open");
   exit(1);
}
```

- **Returns a small identifying integer *file descriptor***
  - `fd == -1` indicates that an error occurred
- **Each process created by a Linux shell begins life with three open files associated with a terminal:**
  - 0: standard input (stdin)
  - 1: standard output (stdout)
  - 2: standard error (stderr)

Closing Files

- Closing a file informs the kernel that you are finished accessing that file

```
int fd;     /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
   perror("close");
   exit(1);
}
```

- Closing an already closed file is a recipe for disaster in threaded programs (more on this later)
- Moral: Always check return codes, even for seemingly benign functions such as `close()`

Reading Files

- **Reading a file copies bytes from the current file position to memory, and then updates file position**

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */

/* Open file fd ...  */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```

- **Returns number of bytes read from file `fd` into `buf`**
  - Return type **`ssize_t`** is signed integer
  - **`nbytes < 0`** indicates that an error occurred
  - ***Short counts*** (**`nbytes < sizeof(buf)`**) are possible and are not errors!

Writing Files

- **Writing a file copies bytes from memory to the current file position, and then updates current file position**

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

- **Returns number of bytes written from `buf` to file `fd`**
  - **`nbytes < 0`** indicates that an error occurred
  - As with reads, short counts are possible and are not errors!

Simple Unix I/O example

- Copying stdin to stdout, one byte at a time

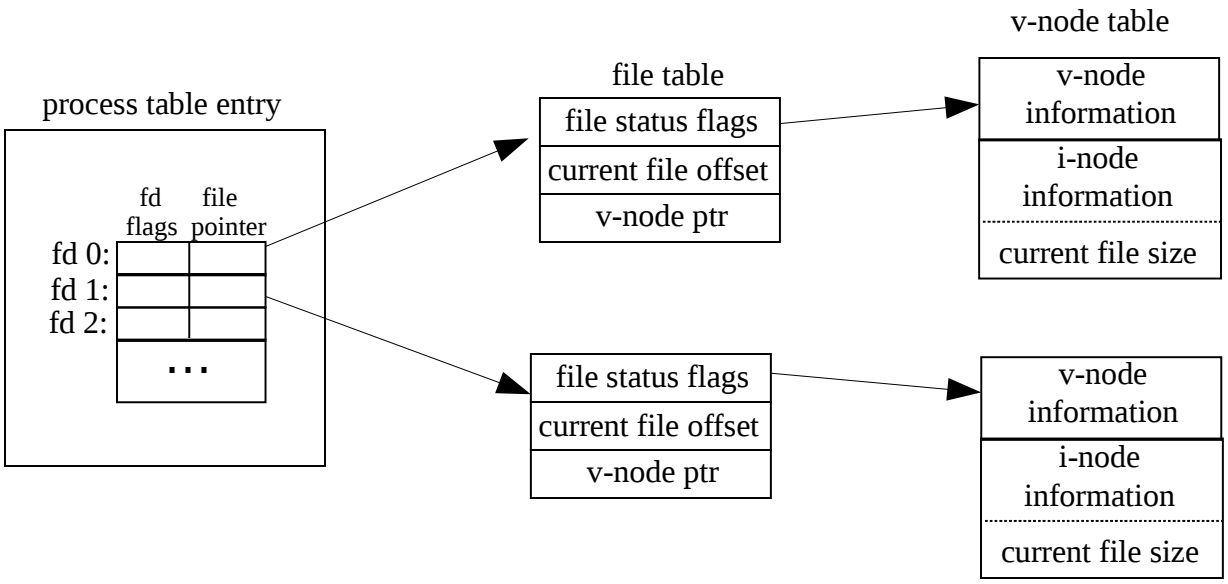
```
#include <unistd.h>

int main(void)
{
    char c;

    while(read(STDIN_FILENO, &c, 1) != 0)
        write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

On Short Counts

- **Short counts can occur in these situations:**
  - Encountering (end-of-file) EOF on reads
  - Reading text lines from a terminal
  - Reading and writing network sockets
- **Short counts never occur in these situations:**
  - Reading from disk files (except for EOF)
  - Writing to disk files
- **Best practice is to always allow for short counts.**

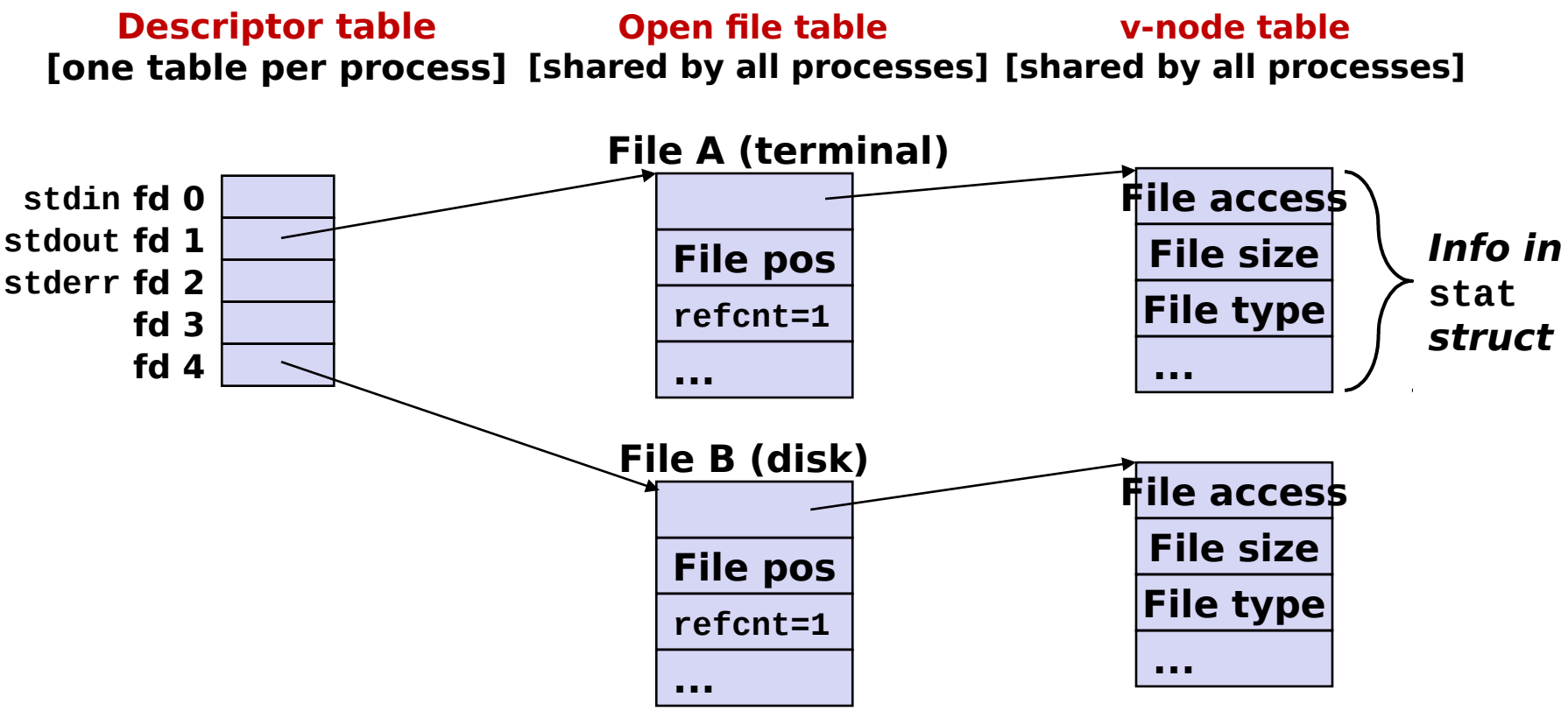
How the Unix Kernel Represents Open Files

- An open file descriptor table within each process table entry
- Kernel *file table* for all open files
- *v-node table*
  - v-node: filesystem independent portion of the i-node
  - i-node: file owner, size, dev, ptrs to data blocks, etc



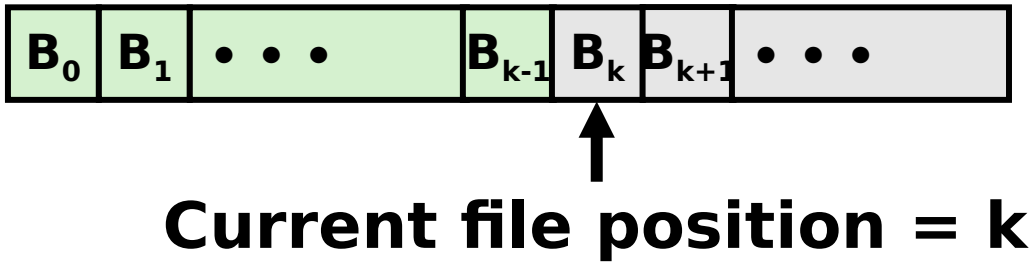
How the Unix Kernel Represents Open Files

- Two descriptors referencing two distinct open files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



Unix I/O Overview

- **Elegant mapping of files to devices allows kernel to export simple interface called *Unix I/O:***
  - Opening and closing files
    - **`open()`** and **`close()`**
  - Reading and writing a file
    - **`read()`** and **`write()`**
  - Changing the ***current file position*** (seek)
    - indicates next offset into file to read or write
    - **`lseek()`**



open Function

```
#include <fcntl.h>
int open(const char *pathname, int oflag, … /* , mod
e_t mode */);
```

- `oflag` argument
  - One and only one of the three
  - O_RDONLY, O_WRONLY, O_RDWR
  - Optional
  - O_APPEND, O_CREAT, O_EXCL, O_TRUNC, O_NOCTTY, O_NONBL OCK,
  - SUS I/O Options: O_DSYNC, O_RSYNC, O_SYNC
- Return Value: guaranteed to be the lowest-numbered unused descriptor

O_APPEND	Append to the end of file on each write. We describe this option in detail in Section 3.11.
O_CREAT	Create the file if it doesn't exist. This option requires a third argument to the open function, the mode, which specifies the access permission bits of the new file. (When we describe a file's access permission bits in Section 4.5, we'll see how to specify the mode and how it can be modified by the umask value of a process.)
O_EXCL	Generate an error if O_CREAT is also specified and the file already exists. This test for whether the file already exists and the creation of the file if it doesn't exist is an atomic operation. We describe atomic operations in more detail in Section 3.11.
O_TRUNC	If the file exists and if it is successfully opened for either write-only or read–write, truncate its length to 0.
O_NOCTTY	If the pathname refers to a terminal device, do not allocate the device as the controlling terminal for this process. We talk about controlling terminals in Section 9.6.
O_NONBLOCK	If the pathname refers to a FIFO, a block special file, or a character special file, this option sets the nonblocking mode for both the opening of the file and subsequent I/O. We describe this mode in Section 14.2.

creat and close Functions

```
#include <fcntl.h>
int creat(const char *pathname, mode_t mode);
```

- Equivalent to `open(`*pathname*`, O_WRONLY | O_CREAT | O_TRUNC,` *mode*`);`

```
#include <unistd.h>
int close(int filedes);
```

- When a process terminates, all open files ar e automatically closed by the kernel.

`read` and `write` Functions

```
#include <unistd.h>
ssize_t read(int filedes, void *buf, size_t nbytes);
```

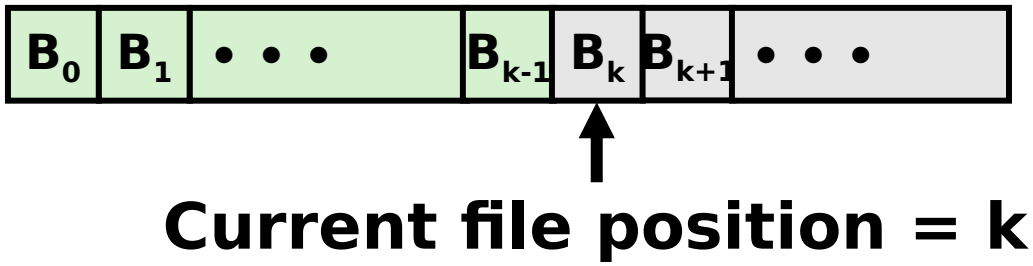
- The number of bytes read is returned. At EOF, 0 is returned.
- Less bytes than the requested are read
  - EOF, a terminal device (per line reading), a network, a pipe or FIFO, a record-oriented device (e.g. a magnetic tape), and interrupted by a signal

```
#include <unistd.h>
ssize_t write(int filedes, const void *buf, size_t nbytes);
```

- The number of bytes written is returned

lseek Function

- Changing the ***current file position*** (seek)
  - indicates next offset into file to read or write
  - **`lseek()`**



`lseek` Function

```
#include <unistd.h>
off_t lseek(int filedes, off_t offset, int whence)
;
```

- "file offset" in bytes from the beginning
- *whence*:
  - `SEEK_SET`: *`offset`* from the beginning of the file
  - `SEEK_CUR`: current file offset + *`offset`*
  - `SEEK_END`: file size + *`offset`*

- To determine the current offset
  - `off_t curpos = lseek(fd,  );`
  - For a pipe, FIFO or socket, it returns -1 and sets `errno` to `ESPIPE`

Program 3.1

```
#include "apue.h"
int
main(void)
{
  if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
     printf("cannot seek₩n");
  else
     printf("seek OK₩n");
  exit(0);
}
```

lseek Function

- Program 3.1

```
$ ./a.out < /etc/motd
seek OK
$ cat < /etc/motd | ./a.out
cannot seek
$ ./a.out < /var/spool/cron/FIFO
cannot seek
```

- Program 3.2 (creating a hole in a file)

```
$ ./a.out
$ ls –l file.hole
-rw-r--r--   1 sar   16394 Nov 25 01:01 file.hole
$ od –c file.hole
“abcdefghij\0…\0ABCDEFGHIJ”
$ ls –ls file.hole file.nohole
  8 –rw-r--r-- r sar 16394 Nov 25 01:01 file.hole
 20 –rw-r--r-- r sar 16394 Nov 25 01:01 file.nohole
```

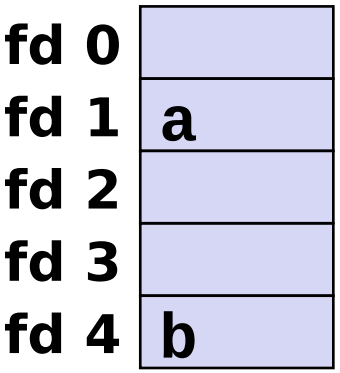
I/O Redirection

- **Question: How does a shell implement I/O redirection?**

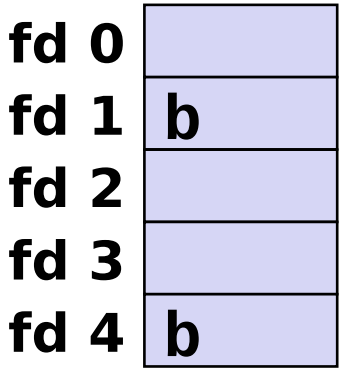
```
$ ls > foo.txt
```

- **Answer: By updating the descriptor table**
  - Copies (per-process) descriptor table entry **`oldfd(4)`** to entry **`newfd(1)`**

Descriptor table *before* `redirection`

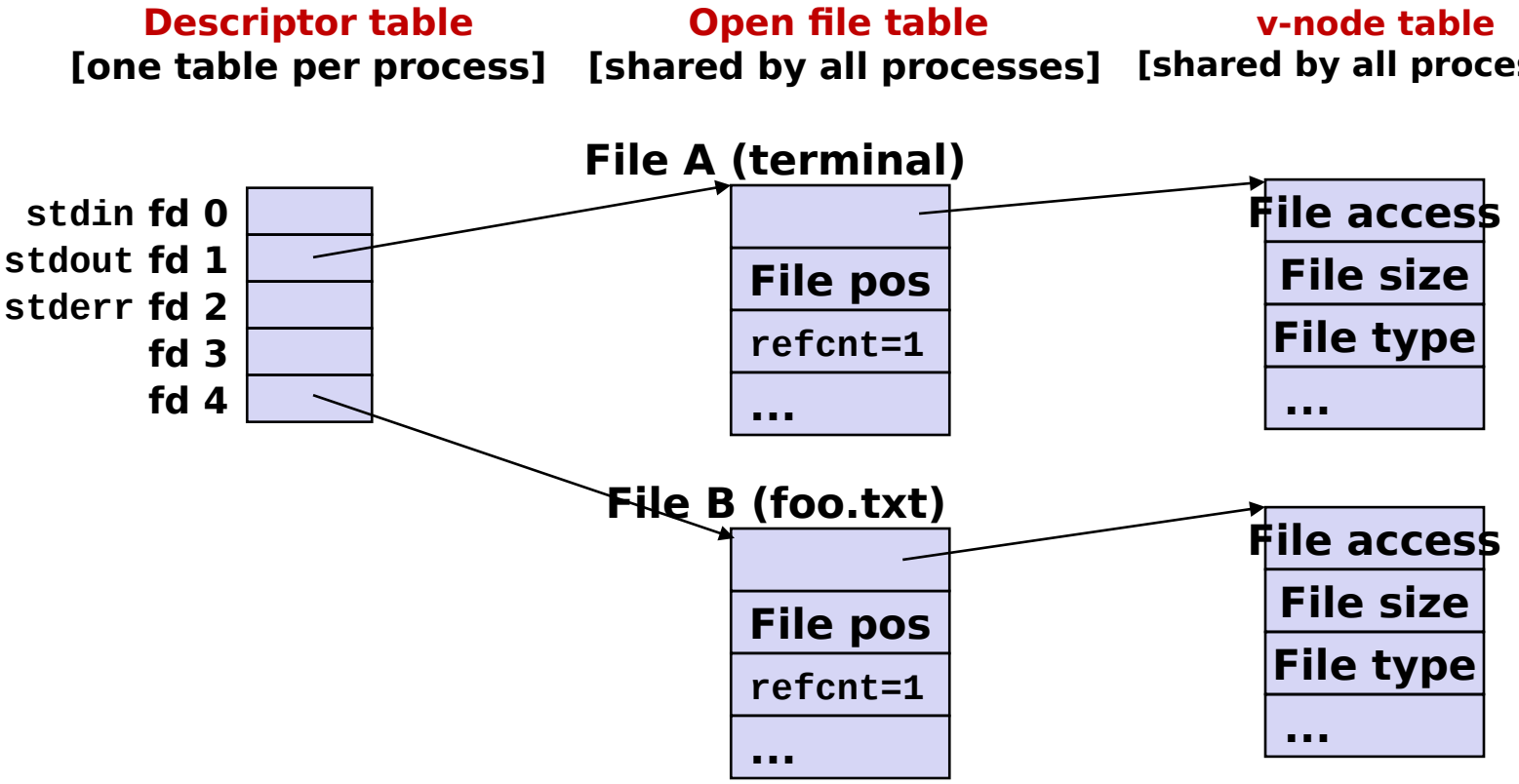


Descriptor table *after* `redirection`



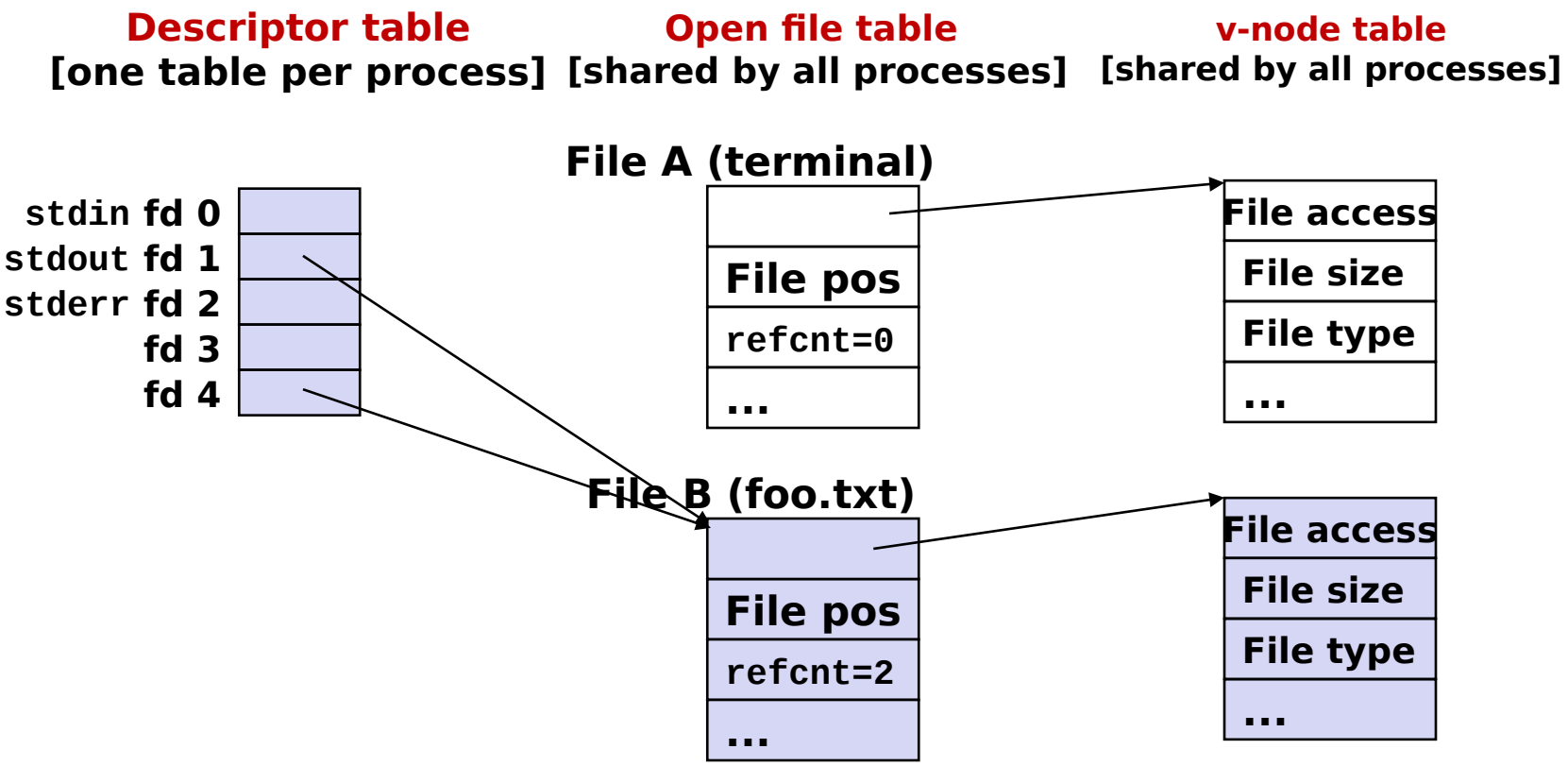
I/O Redirection Example

- **Step #1: open file to which stdout should be redirected**
  - Happens in child executing shell code, before **exec**



I/O Redirection Example (cont.)

- **Step #2: call `dup2(4,1)`**
  - cause fd=1 (stdout) to refer to disk file pointed at by fd=4



Program 3.2

```
#include "apue.h"
#include <fcntl.h>
char buf1[] = "abcdefghij";  char buf2[] = "ABCDEFGHIJ";
int main(void)
{  int fd;
   if ((fd = creat("file.hole", FILE_MODE)) < 0)
      err_sys("creat error");
   if (write(fd, buf1, 10) != 10)
      err_sys("buf1 write error");
   /* offset now = 10 */
   if (lseek(fd, 16384, SEEK_SET) == -1)
      err_sys("lseek error");
   /* offset now = 16384 */
   if (write(fd, buf2, 10) != 10)
      err_sys("buf2 write error");
   /* offset now = 16394 */
   exit(0);
}
```

Buffered I/O vs Unbuffered I/O

- Unbuffered I/O (part of POSIX.1 and the Single UNIX Specification, but not of ISO C)
- File sharing among multiple processes
  - An atomic operation

`open`, `read`, `write`, `lseek`, and `close`

`dup`, `fcntl`, `sync`, `fsync`, and `ioctl`

Buffered I/O - ISO c

```c
#include <stdio.h>

int main(void)
{
    char ch;
    while((ch=fgetc(stdin))!=EOF)
        fputc(ch, stdout);
    return 0;
}
```

Unbuffered I/O - POSIX.1

```c
#include <stdio.h>

int main(void)
{
    char ch[2];
    while(read(0, ch, 1))
        write(1, ch, 1);
    return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
    char ch;
    while((ch=fgetc(stdin))!=EOF)
        fputc(ch, stdout);
    return 0;
}
```

Buffered IO
- ISO c
- usually faster
- why?

Unbuffered IO
- Posix.1

```c
#include <stdio.h>

int main(void)
{
    char ch[2];
    while(read(0, ch, 1))
        write(1, ch, 1);
    return 0;
}
```

I/O Efficiency

- Program 3.4

- BUFFSIZE 4096
  - Reading a 103,316,352-byte file, using 20 different buffer sizes
  - Tested against the Linux `ext2` file system with 4,096-byte blocks

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	#loops
1	124.89	161.65	288.64	103,316,352
2	63.10	80.96	145.81	51,658,176
4	31.84	40.00	72.75	25,829,088
8	15.17	21.01	36.85	12,914,544
16	7.86	10.27	18.76	6,457,272
32	4.13	5.01	9.76	3,228,636
64	2.11	2.48	6.76	1,614,318
128	1.01	1.27	6.82	807,159
256	0.56	0.62	6.80	403,579
512	0.27	0.41	7.03	201,789
1,024	0.17	0.23	7.84	100,894
2,048	0.05	0.19	6.82	50,447
4,096	0.03	0.16	6.86	25,223
8,192	0.01	0.18	6.67	12,611
16,384	0.02	0.18	6.87	6,305
32,768	0.00	0.16	6.70	3,152
65,536	0.02	0.19	6.92	1,576
131,072	0.00	0.16	6.84	788
262,144	0.01	0.25	7.30	394
524,288	0.00	0.22	7.35	198

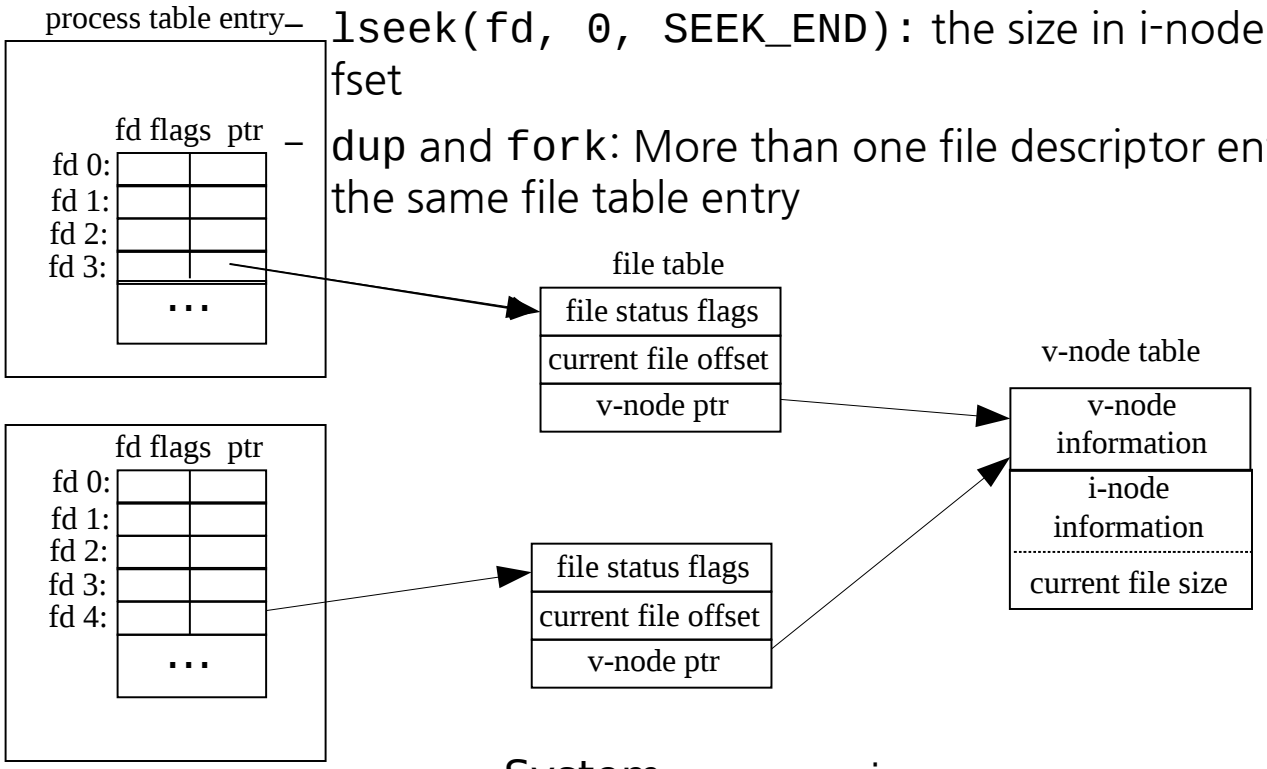
Program 3.4

```
#include "apue.h"
#define BUFFSIZE 4096
int
main(void)
{  int n;
   char buf[BUFFSIZE];
   while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
      if (write(STDOUT_FILENO, buf, n) != n)
         err_sys("write error");
   if (n < 0)
      err_sys("read error");
   exit(0);
}
```

File Sharing

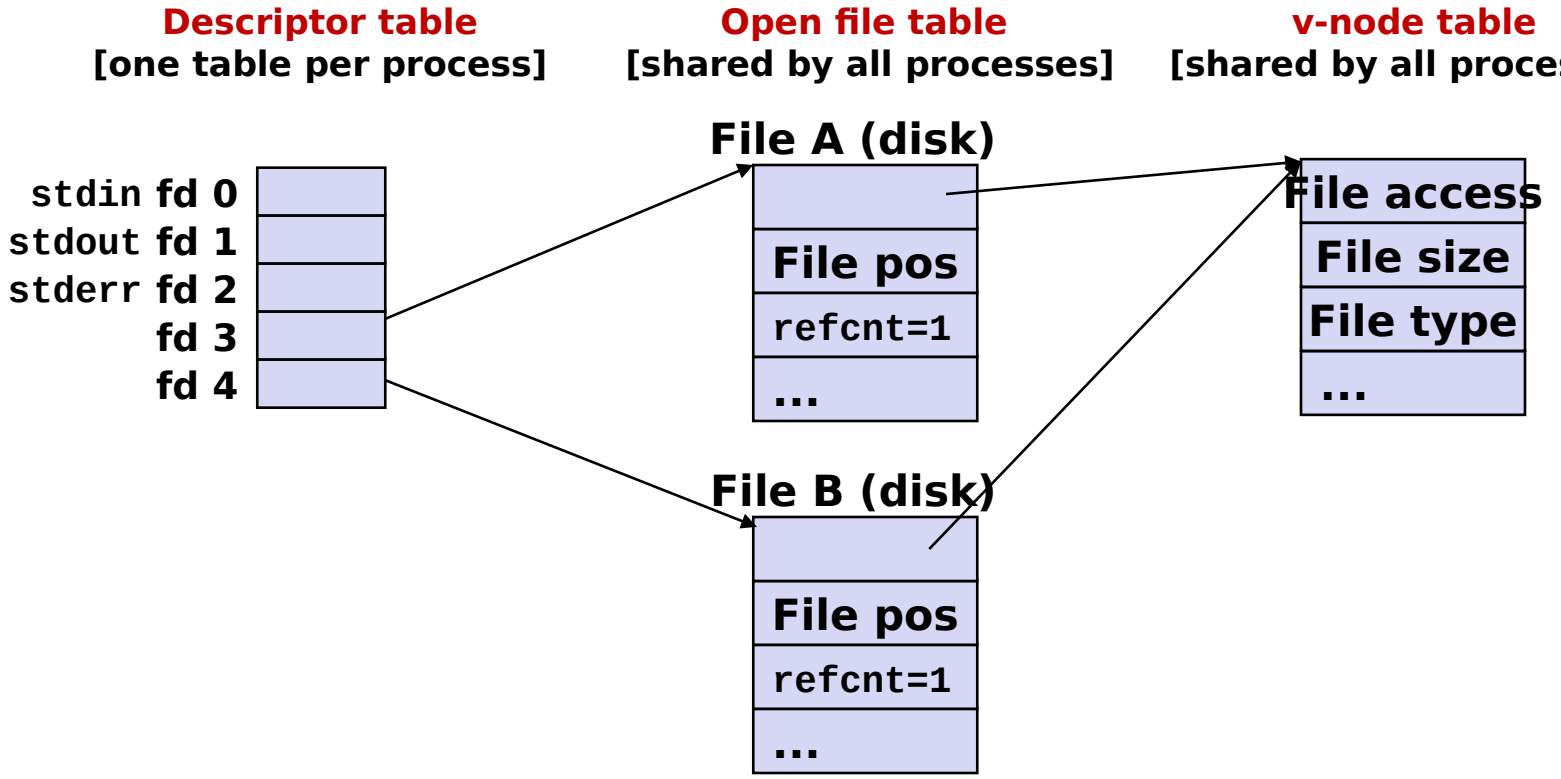
(e.g. Error-log file /var/log/dmesg/)

- Each process gets its own **current offset**.
  - Each `write` changes the offset (and the size in the i-node table.)
  - For each `write` to a file opened with O_APPEND flag set, the size in the i-node table ☒ the offset
  - `lseek(fd, 0, SEEK_END)`: the size in i-node table ☒ the offset
  - `dup` and `fork`: More than one file descriptor entry pointing to the same file table entry



File Sharing

- **Two distinct descriptors sharing the same disk file through two distinct open file table entries**
  - E.g., Calling **`open`** twice with the same **`filename`** argument



Atomic Operations

(e.g. Error-log file /var/log/dmesg/)

- When a process wants to append to the end of a file,

```
if (lseek(fd, 0L, SEEK_END) < 0)
 err_sys("lseek error");

if (write(fd, buf, 100) != 100)
 err_sys("write error");
```

- Assuming process A and B that append to the same file,
  - A's `lseek` sets its offset to 1500.
  - B's does the same thing, and calls a `write` to increase its offset t o 1600. Kernel also updates the size in the v-node table entry.
  - A calls a `write`, which will overwrite the B's data.
- Any operation that requires more than one function call can not be atomic.
- `write` to a file opened with O_APPEND flag ☒ atomic

Atomic Operations

- The Single UNIX Specification

```
#include <unistd.h>
ssize_t pread(int filedes, void *buf, size_t nbytes,
              off_t offset);
ssize_t pwrite(int filedes, void *buf, size_t nbytes,
               off_t offset);
```

- `pread` equivalent to calling `lseek` followed by a call to `read`
- `pwrite` equivalent to calling `lseek` followed by a call to `write`

Atomic Operations

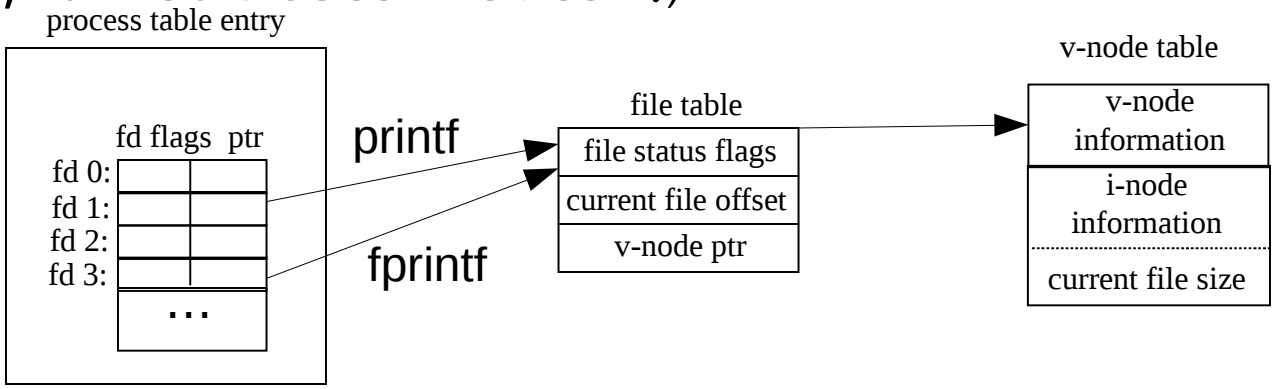
- `open(fd, …|O_CREAT|O_EXCL, modes)`
  - The check for the existence of the file and the creation of t he file is performed as an **atomic operation.**
  - **e.g. Lock file**
- Otherwise, we might try

```
   if ((fd = open(pathname, O_WRONLY)) < 0){
 if (errno == ENOENT) {

 if ((fd = creat(pathname, mode)) < 0)

 err_sys("creat error");

 } else

 err_sys("open error");

 }
}
```

dup and dup2 Functions

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

- dup returns the lowest-numbered available file descriptor.
- dup2 makes a copy of *filedes* into *filedes2* (if opened, it first closes *filedes2*.)



sync, fsync, and fdatasync Functions

```
#include <unistd.h>
int fsync(int filedes);
int fdatasync(int filedes);
void sync(void);
```

- *"Delayed write"* via a buffer cache
  - Data is copied into one of its buffers and queued for writing to disk at some later time.
- sync simply queues all the modified block buffers for writing and returns.
- fsync waits for disk writes, including file attributes, to complete before returning. (only to **a single file**)
- fdatasync affects only the data portions of **a file**.

`fcntl` Function

- can change the properties of a file

```
#include <fcntl.h>
int fcntl(int filedes, int cmd, ... /* int arg */);
```

- *cmd* argument
  - duplicate an existing descriptor (F_DUPFD)
  - Its FD_CLOEXEC file descriptor flag is cleared.
  - get/set file descriptor flags (F_GETFD, F_SETFD)
  - Currently only one file descriptor flag is defined: FD_CLOEXEC flag
  - get/set file status flags (F_GETFL, F_SETFL)
  - the flags used by **open**
  - get/set asynchronous I/O ownership (F_GETOWN, F_SETOWN)
  - get/set record lock (F_GETLK, F_SETLK, F_SETLKW)

`fcntl` Function

- Program 3.10 – Print file flags for a specified descriptor.
- Program 3.11 – Turn on one or more of the file status flags.
- Figure 3.12 (Linux ext2 timing results)

Operation	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)
read time from Figure 3.5 for `BUFFSIZE` = 4,096	0.03	0.16	6.86
normal `write` to disk file	0.02	0.30	6.87
`write` to disk file with `O_SYNC` set	0.03	0.30	6.83
`write` to disk followed by `fdatasync`	0.03	0.42	18.28
`write` to disk followed by `fsync`	0.03	0.37	17.95
`write` to disk with `O_SYNC` set followed by `fsync`	0.05	0.44	17.95

- Figure 3.13 (Mac OS X timing results)

Operation	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)
`write` to `/dev/null`	0.06	0.79	4.33
normal `write` to disk file	0.05	3.56	14.40
`write` to disk file with `O_FSYNC` set	0.13	9.53	22.48
`write` to disk followed by `fsync`	0.11	3.31	14.12
`write` to disk with `O_FSYNC` set followed by `fsync`	0.17	9.14	22.12

Program 3.10

```
$ ./a.out 0
read write
$ ./a.out 1
read write
$ ./a.out 0 < /dev/tty
read only
$ ./a.out 1 > foo
$ cat foo
write only
$ ./a.out 2 2>>temp.foo
write only, append
$ ./a.out 5 5<>temp.foo
read write
```

Program 3.10

```
#include "apue.h"
#include <fcntl.h>
int main(int argc, char *argv[])
{  int val;
   if (argc != 2)
      err_quit("usage: a.out <descriptor#>");
   if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) <
0)
      err_sys("fcntl error for fd %d", atoi(argv
[1]));
   switch (val & O_ACCMODE) {
   case O_RDONLY:
      printf("read only"); break;
   case O_WRONLY:
      printf("write only"); break;
   case O_RDWR:
      printf("read write"); break;
   default:
      err_dump("unknown access mode");
   }
   if (val & O_APPEND)
      printf(", append");
   if (val & O_NONBLOCK)
      printf(", nonblocking");
#if defined(O_SYNC)
   if (val & O_SYNC)
      printf(", synchronous writes");
#endif
#if !defined(_POSIX_C_SOURCE) && define
d(O_FSYNC)
   if (val & O_FSYNC)
      printf(", synchronous writes");
#endif
   putchar('₩n');
   exit(0);
}
```

Program 3.11

```
#include "apue.h"
#include <fcntl.h>
void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{  int val;
   if ((val = fcntl(fd, F_GETFL, 0)) < 0)
      err_sys("fcntl F_GETFL error");
   val |= flags;/* turn on flags */
   if (fcntl(fd, F_SETFL, val) < 0)
      err_sys("fcntl F_SETFL error");
}
```

`ioctl` Function

```
#include <unistd.h> /* SVR4 */
#include <sys/ioctl.h> /* BSD and Linux */
#include <stropts.h>    /* XSI STREAMS */
int ioctl(int filedes, int request, …);
```

- The catchall for I/O operations
- Each device driver defines its own set of `ioctl` commands.
- FreeBSD `ioctl` operations

Category	Constant Names	Header	Number of `ioctl`s
disk labels	DIOxxx	`<sys/disklabel.h>`	6
file I/O	FIOxxx	`<sys/filio.h>`	9
mag tape I/O	MTIOxxx	`<sys/mtio.h>`	11
socket I/O	SIOxxx	`<sys/sockio.h>`	60
terminal I/O	TIOxxx	`<sys/ttycom.h>`	44

/dev/fd

- Opening `/dev/fd/n` is equivalent to duplicating descriptor `n` (assuming the descriptor n is open)
  - `fd = open("/dev/fd/0", mode);`
  - `fd = dup(0);`
  - The `mode` could be ignored or must be a subset of the original mode.
- Uniformity and cleanliness
  - `cat file2 | cat file1 – file3 | lpr`
  - `cat file2 | cat file1 /dev/fd/0 file3 | lpr`