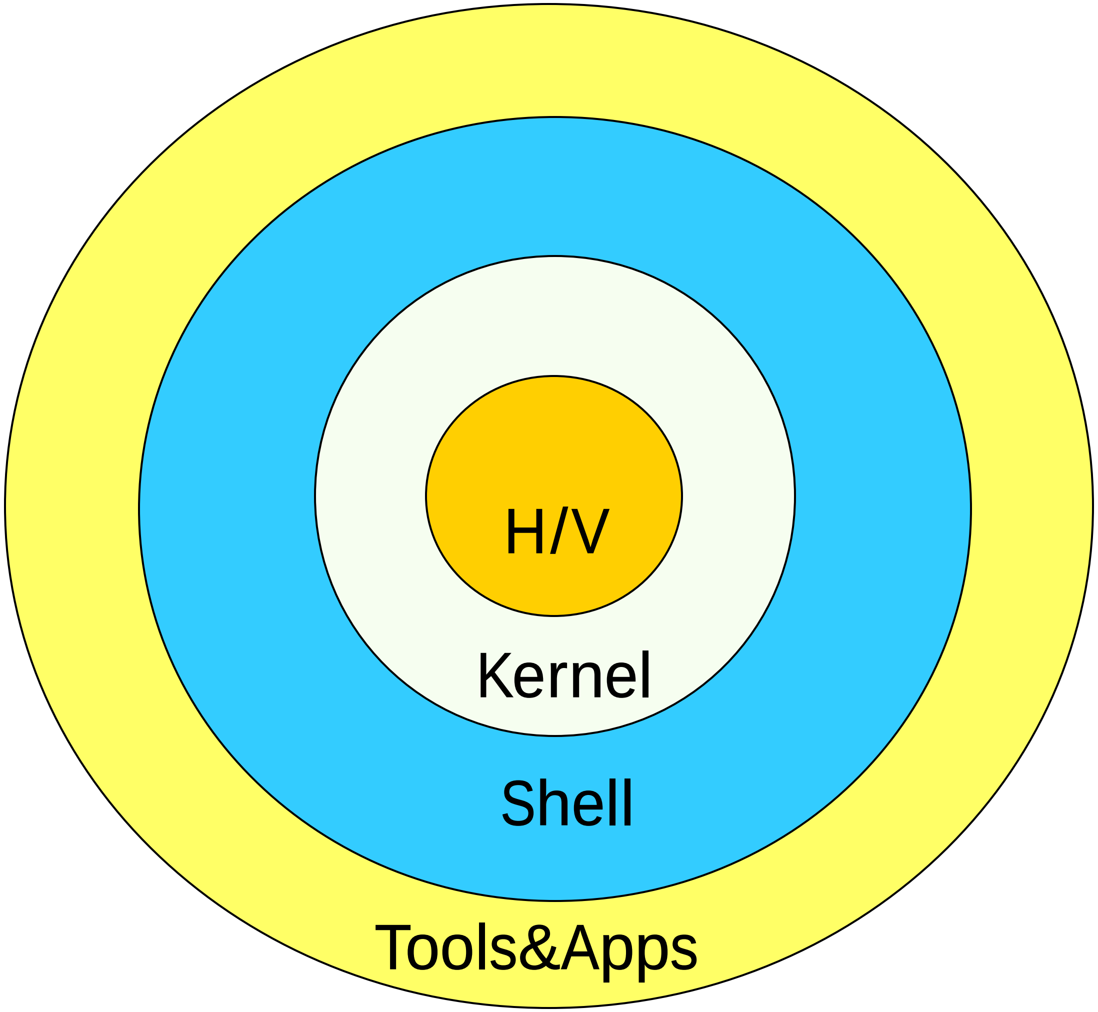
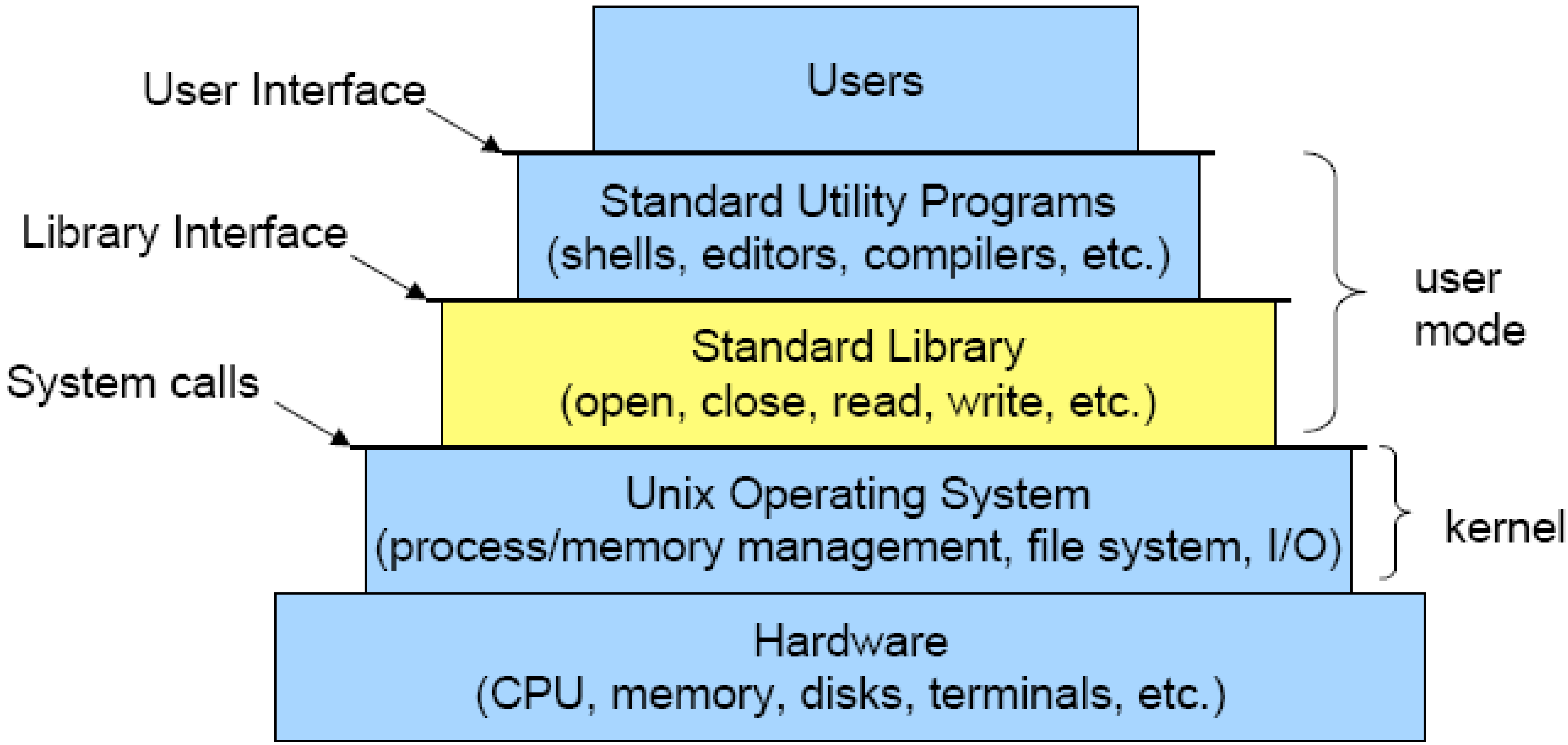
System Programming

File Management

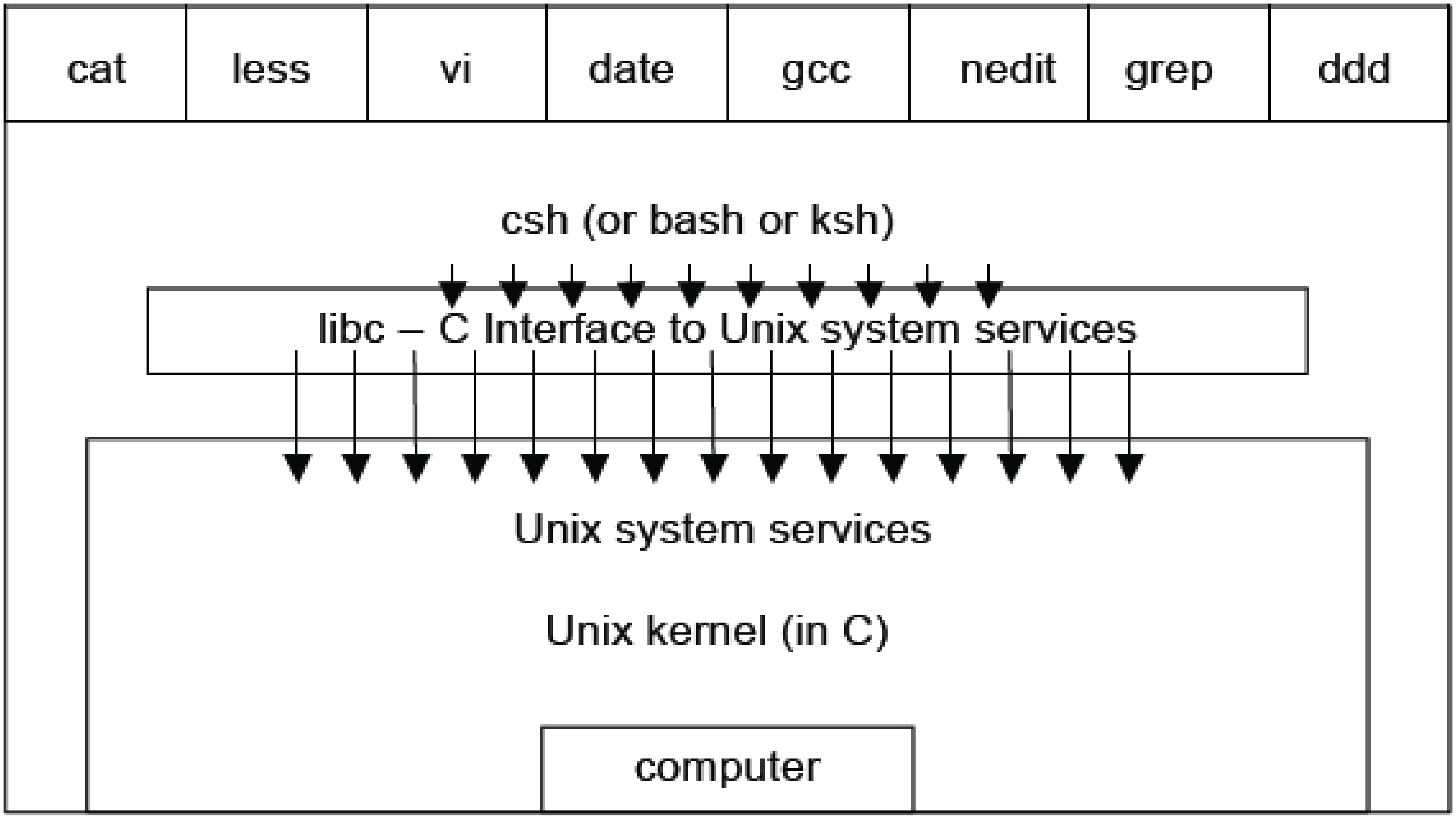
Parts of the UNIX OS



Layers in a Unix-based System



Layers of System Software



Shell Interpreter

- The interpreter is a C program!
- The shell interpreter is the program executed when you write

```
#!/bin/sh
```

- Each line of a shell script is input to a C program that parses the line, and determines how to execute it.

Standard Libraries

- System calls are not part of the C language definition.
- System calls are defined in libraries (.a .so)
- Libraries typically contain many .o object files.
- To create your own library archive file:

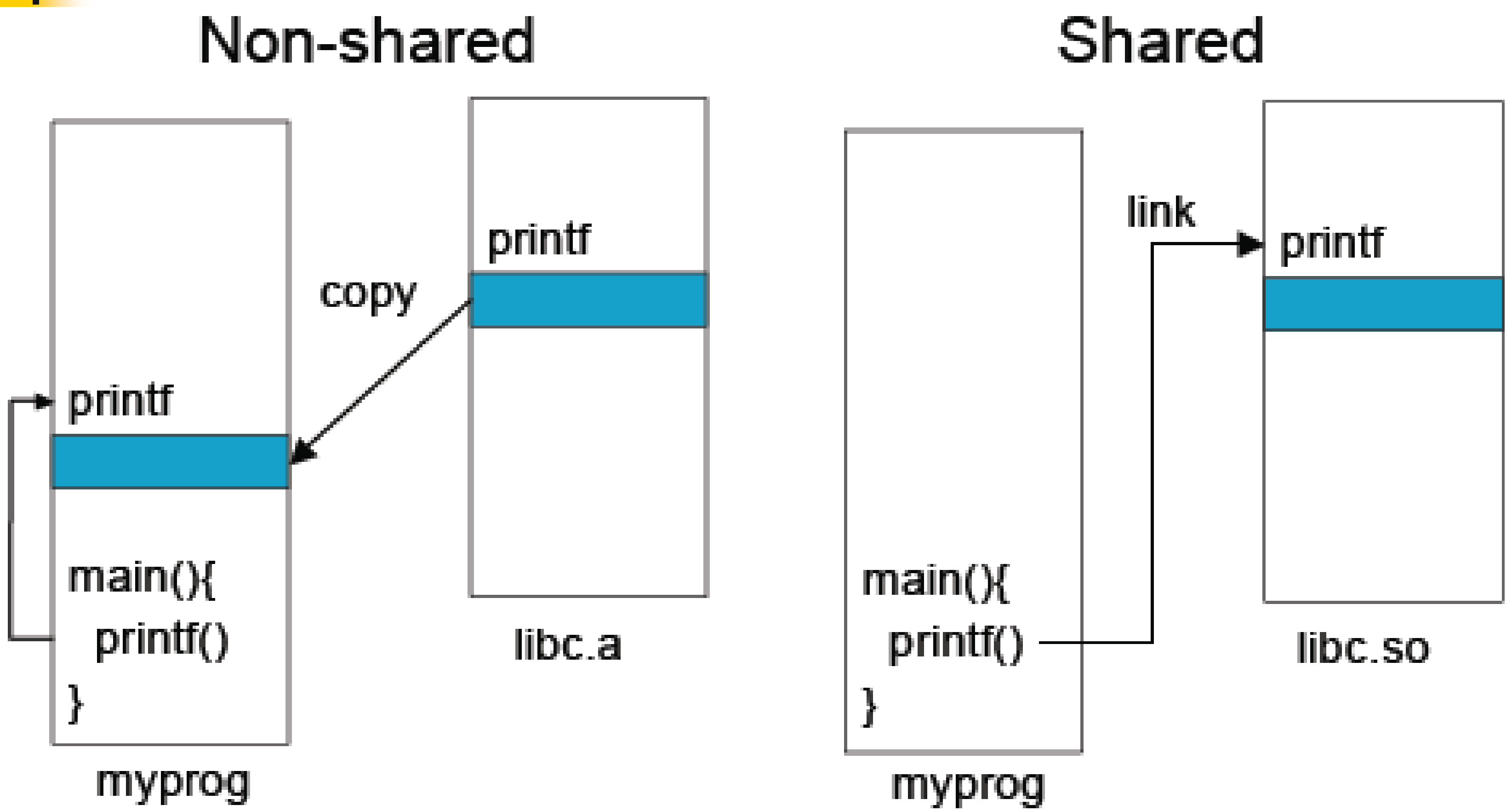
```
ar crv mylib.a *.o
```

- Look in `/usr/lib` and `/usr/local/lib` for system libraries.

Standard Libraries

- .a libraries are not shared. The functions used are copied into the executable of your program.
  - size bloat when lots of processes use the same libraries
  - performance and portability are the wins
- .so libraries are shared. One copy exists in memory, and all programs using that library link to it to access library functions.
  - reduces total memory usage when multiple processes use the shared library.
  - small performance hit as extra work must be done either when a library function is called, or at the beginning.
  - many tradeoffs and variations between OS's

Shared vs. Non-Shared Libraries



System Calls

- Programs make `system calls` via libraries.
  - `libc` provides the C interface to system calls.
  - a subroutine call directly to the Unix kernel.
- 4 main categories of system calls
  - File management
  - Process management
  - Communication
  - Error and signal handling

System Call Routine

Assembly	**Explanation**
`_chdir:`	`_chdir:`
`subl $4,%exp`	
`pushl %ebx`	save address
`movzwl 12(%esp),%eax`	prepare parameters
`movl %eax,4(%esp)`	
`movl $23,%eax`	
`movl 4(%esp),%ebx`	
`int $0x80`	trap to kernel mode and syscall handler
`movl %eax,%edx`	
`testl %edx,%edx`	check for error returned
`jge L2`	if not, jump to L2
`negl %edx`	else handle error
`movl %edx,_errno`	
`movl $-1,%eax`	
`popl %ebx`	
`addl $4,%esp`	
`ret`	
`L2:`	`L2:`
`movl %edx,%eax`	clean up
`popl %ebx`	
`addl $4,%esp`	
`ret`	and return

Executing a Program

- A special start-up routine (`crt0`) is always linked in with your program.
- This routine reads the arguments and calls main.
- The `libc` library is automatically linked into your program, which is how you have access to many C functions (`printf`, `open`, etc.)
- Your program also calls special functions on exit that close file descriptors and clean up other resources.

C versus C++

- No string data types
  - Use character arrays instead
  - Use `strcpy()`, `strncpy()`, `strcmp()`, `strncmp()` to “assign” and compare character arrays
- No embedded declarations
  - Must declare all variables at the beginning of a code block
- Very different File and Standard I/O functions
  - `printf()` versus `cout`
  - `scanf()` and `fgets()` versus `cin`

Unbuffered I/O vs. Standard I/O

Unbuffered I/O

- can perform I/O using system calls (open, read, write, close, lseek)
- must specify buffer size and number of bytes
- no formatting options
- these functions use file descriptors as arguments
- note constants (STDIN_FILENO, etc.) defined in unistd.h

Standard I/O

- a set of C library functions (printf, scanf, getc)
- buffering is automatic
- many formatting options
- the stdio functions are built from the primitive system calls
- note constants (stdin, etc.) defined in stdio.h

Basic File I/O

- Remember everything in Unix is a file
- Kernel maintains a list of open files for each process
- Files can be opened for reading, writing
- Must include `<stdio.h>` to use I/O system calls
  - Note: Some of the system calls used to interact with the Unix kernel are also available in other OS'. However, they may be (probably are) implemented much differently and some are not available at all.

Basic File I/O (cont.)

- Most Unix I/O can be done with 5 system calls.
  - open, read, write, close, lseek
- Each file is referenced by a file descriptor (an integer)
- Three files are opened automatically
  - FD 0: standard input
  - FD 1: standard output
  - FD 2: standard error
- When a new file is opened, it is assigned the lowest available FD
- man –s 2 <systemcall> (for more information)

open( )

int open(char *path, int flags, mode_t mode);

- path: absolute or relative path
- flags:
  - O_RDONLY – open for reading
  - O_WRONLY – open for writing
  - O_RDWR – open for reading and writing
  - O_CREAT – create the file if it doesn't exist
  - O_TRUNC – truncate the file if it exists (overwrite)
  - O_APPEND – only write at the end of the file
- mode: specify permissions if using O_CREAT
- Returns newly assigned file descriptor.

open( )

- permissions on creation
  - 777 vs 0777 vs 0x777

permission	who
read (r--, 4)	user (0700)
write(-w-, 2)	group (0070)
execute(--x, 1)	others (0007)

- fd = open(”name”, O_RDWR|O_CREAT, 0700);
- fd return value
  - fd >= 0 – open succesful
  - fd < 0 – open unsuccesful, erro in errno

read() and write()

ssize_t read(int fd, void *buf, size_t nbytes);

ssize_t write(int fd, void *buf, size_t nbytes);

- fd is value returned by open
- buf is usually an array of data
- nbytes is size of buf (read) or size of data to write
- returned value
  - > 0 number of bytes read or written
  - <= nbytes for read
  - 0 EOF
  - < 0 error

read() example

```
bytes = read(fd, buffer, count);
```

- Reads from file associated with `fd`; place `count` bytes into `buffer`
- Returns the number of bytes read or `-1` on error

```
int fd=open("someFile", O_RDONLY);
char buffer[4];
int bytes =
        read(fd, buffer, 4);
```

write() example

```
bytes = write(fd, buffer, count);
```

- Writes contents of `buffer` to a file associated with `fd`
- Returns the number of bytes written or `-1` on error

```
int fd=open("someFile", O_WRONLY);
char buffer[4];
int bytes =
        write(fd, buffer, 4);
```

Copying stdin to stdout

```
#define BUFFSIZE    8192        // how this is chosen?
int main(void) {
  int n;
  char buf[BUFFSIZE];
  while ((n=read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
    if (write(STDOUT_FILENO, buf, n) != n)
       printf("write error");
    if (n < 0)
       printf("read error");
}
```

I/O Efficiency

- Let's run the program for different BUFFSIZE values, reading a 1,468,802 byte file and holding timing results.

- This accounts for the minimum in the system time occurring at a BUFFSIZE of 8192. Increasing the buffer size beyond this has no effect.

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	#loops
1	23.8	397.9	423.4	1468802
2	12.3	202.0	215.2	734401
4	6.1	100.6	107.2	367201
8	3.0	50.7	54.0	183601
16	1.5	25.3	27.0	91801
32	0.7	12.8	13.7	45901
64	0.3	6.6	7.0	22951
128	0.2	3.3	3.6	11476
256	0.1	1.8	1.9	5738
512	0.0	1.0	1.1	2869
1024	0.0	0.6	0.6	1435
2048	0.0	0.4	0.4	718
4096	0.0	0.4	0.4	359
8192	0.0	0.3	0.3	180
16384	0.0	0.3	0.3	90
32768	0.0	0.3	0.3	45
65536	0.0	0.3	0.3	23
131072	0.0	0.3	0.3	12

`close()`

int close(int fd);

- Closes the file referenced by fd
- Returns 0 on success, -1 on error
- `close(1);`
  - closes standard output.

lseek()

off_t lseek(int fd, off_t offset, int whence);

- Moves file pointer to new location
- fd is number returned by open
- off_t is not necessarily a long or int … could be a quad!
- offset is number of bytes
- whence:
  - SEEK_SET – offset from beginning of file
  - SEEK_CUR – offset from current location
  - SEEK_END – offset from end of file
- Returns offset from beginning of file or –1 on error

lseek() example

- Test if standard input is capable of seeking

```
int main(void) {
  if (lseek(STDIN_FILENO, 0, SEEK_CUR)==-1)
      printf("cannot seek\n");
  else
      printf("seek OK\n");
  exit(0);
}
```

- $ **a.out < /etc/motd** gives “seek OK ”
- $ **cat < /etc/motd | a.out** gives “cannot seek”
- $ **a.out < /var/spool/cron/FIFO** gives “cannot seek”

lseek() example

- char buf1[] = "abcdefghij“, buf2[] = "ABCDEFGHIJ";
  int fd = creat("file.hole", FILE_MODE);
  write(fd, buf1, 10);
  lseek(fd, 40, SEEK_SET);
  write(fd, buf2, 10);
- **a.out**
- $ **ls -l file.hole** *check its size*

  -rw-r--r-- 1 root 50 Jul 31 05:50 file.hole
- $ **od -c file.hole** *let's look at the actual contents*

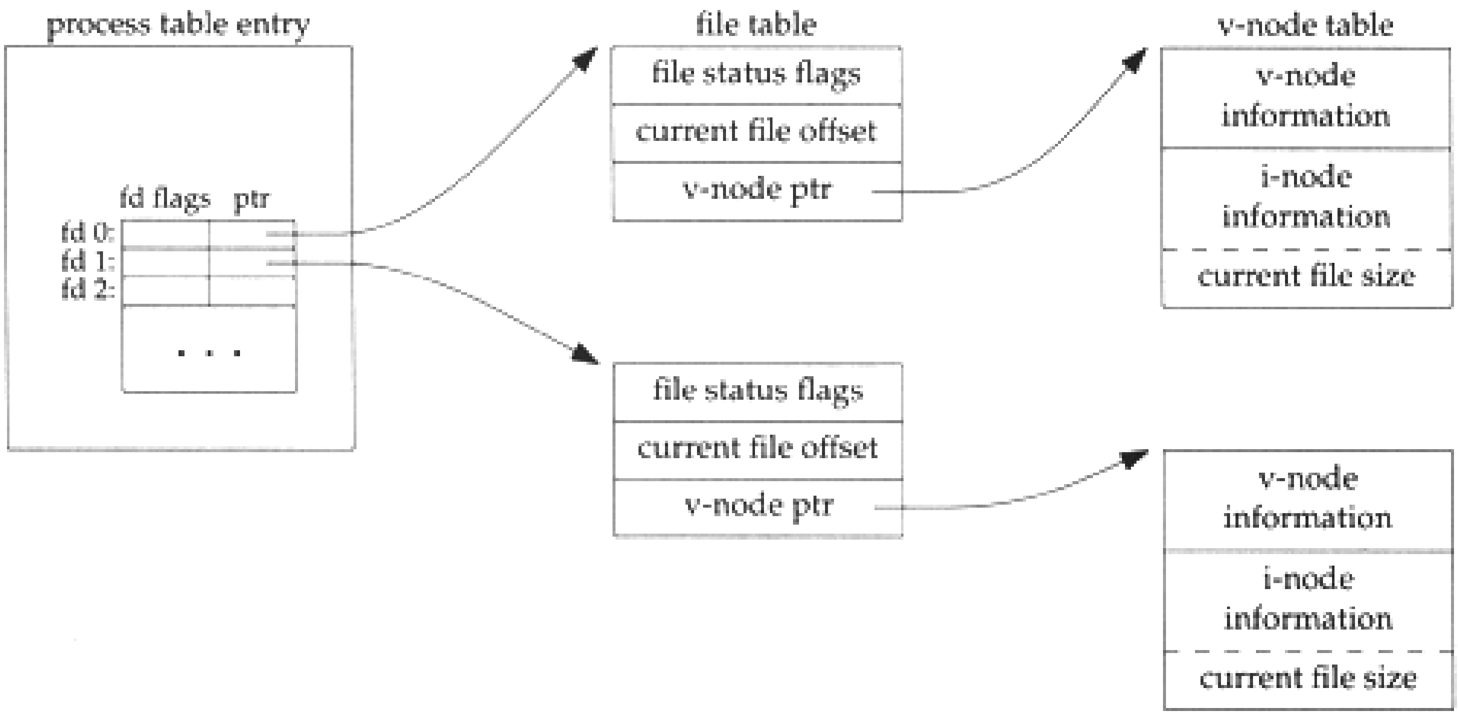
```
0000000  a  b  c  d  e  f  g  h  i  j \0 \0 \0 \0 \0 \0
0000020 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000040 \0 \0 \0 \0 \0 \0 \0 \0  A  B  C  D  E  F  G  H
0000060  I  J
0000062
```

File Sharing

- Process Table -> fd entry
- Open File Table -> status/pointer
- V-Node -> real file entry
- 2 independent processes open a file
  - 2 process tables
  - 2 entries in open file table
  - 1 v-node entry
- 2 processes by fork
  - 2 process tables
  - 1 entry in open file table
  - 1 v-node entry

File Sharing

- Two independent processes with the same file open.
  - v-node is called the filesystem independent portion of the i-node, which supports multiple filesystem types on a given system.

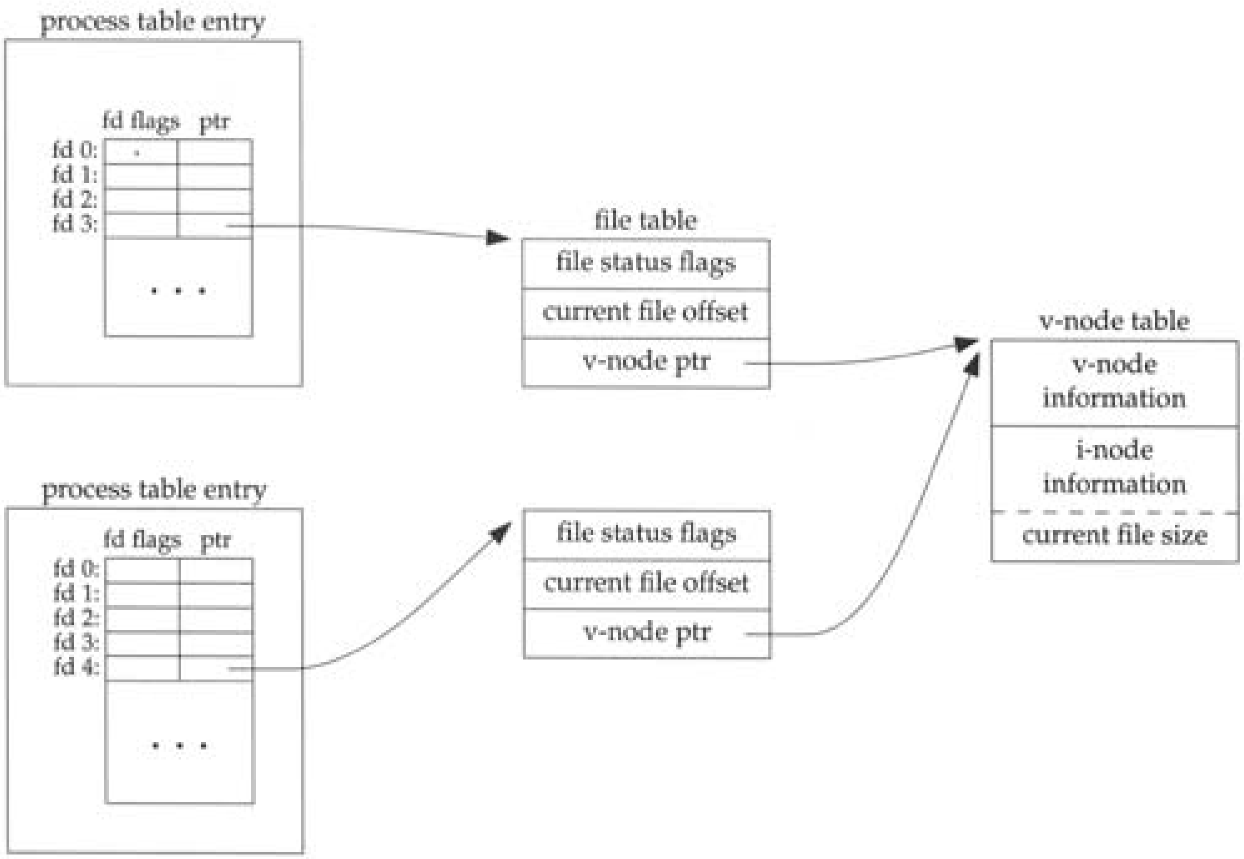


Atomic Operations

- Appending to a file
  - if (lseek(fd, 0L, 2) < 0) printf("lseek error");   /* position to EOF */
    if (write(fd, buff, 100) != 100) printf("write error"); /* and write */
    Does two processes append to the same file? (see next slide).
  - Solution: To avoid the overwrite,
    - Set the O_APPEND flag when a file is opened.
    - Thus, kernel sets the file pointer to the current end before each write.
- Creating a file
  - ```
    if ( (fd = open(pathname, O_WRONLY)) < 0)
      if (errno == ENOENT) {
        if ( (fd = creat(pathname, mode)) < 0)
          printf("creat error");
      } else
          printf("open error");
    ```
    Does another process create and write the same file between the open and the creat?
  - Solution: To avoid the recreation and data loss,
    - Use the open call with the O_CREAT and O_EXCL flags.
    - In this way, the open fails if the file already exists.
    ```

Atomic Operations

- With the previous code, one process may overwrite the data that the other process wrote to the file.



More system calls

- int chmod(char *path, mode_t mode);
- int fchmod(int fildes, mode_t mode);
  - set access permissions of the file pointed to by path or fildes.
- int chdir(const char *path);
- int fchdir(int fildes);
  - change to the directory pointed to by path or fildes.
- int unlink(char *path);
  - remove the link to the file pointed to by path and decrements its link count by one.

More system calls 2

- int link(char *old, char *new);
  - create a new link (directory entry) for the existing file old and increments its link count by one.
- int symlink(char *old, char *new);
  - create a symbolic link new to the file old.
- int access(char *path, int mode);
  - check if path is readable, writable, executable, etc.
- int stat(char *path, struct stat *buf);
- int fstat(int fildes, struct stat *buf);
  - obtain information about the file pointed to by path or fildes.

More system calls 3

- int creat(char *path, mode_t mode);
  - create files
  - equivalent to open() called with the flag
    O_WRONLY | O_CREAT | O_TRUNC
- int fcntl(int fd, int cmd, ...);
  - duplicate fds
  - get/set flags
  - record locks
- int ioctl(int fd, unsigned long request, void arg);
  - “Catchall” for I/O operations
  - special hardware control (disk labels, mag tape I/O)
  - file I/O, socket I/O, terminal I/O (e.g. baudrate)

fcntl() example

- Turn on one or more of the file status flags for a descriptor.

```
void set_fl(int fd, int flags) {
    int val;
    if ((val=fcntl(fd, F_GETFL, 0)) < 0)
    printf("fcntl F_GETFL error");
    val |= flags;        /* turn on flags */
    if (fcntl(fd, F_SETFL, val) < 0)
     printf("fcntl F_SETFL error"); }
```

- To turn flags off, change the middle statement to

```
    val &= ~flags;       /* turn flags off */
```

/dev/fd

- Newer systems provide files named 0,1, 2, ... in /dev/fd. Opening the file /dev/fd/*n* is equivalent to duplicating descriptor *n* (assuming that *n* is open).
  - open("/dev/fd/0", mode) is equivalent to dup(0)
  - the descriptors 0 and fd share the same file table entry
  - open("/dev/fd/0", O_RDWR) succeeds, but it cannot be written
- The pathnames /dev/stdin, /dev/stdout, and /dev/stderr are equivalent to /dev/fd/0, /dev/fd/1, and /dev/fd/2.
- The main use of the /dev/fd files is from the shell. It allows programs that use pathname arguments to handle standard input and standard output in the same manner as other pathnames.
  - `cat /dev/fd/0`
  - `exec 5< fileR ; cat /dev/fd/5`

File I/O using FILES

- Most Unix programs use higher-level I/O functions
  - `fopen()`
  - `fread()`
  - `fwrite()`
  - `fclose()`
  - `fseek()`
- These use the FILE datatype instead of file descriptors
- Need to include `<stdio.h>`

fopen()

- `FILE *file_stream = fopen(path, mode);`
- `path: char*`, absolute or relative path
- `mode:`
  - `r` – open file for reading
  - `r+` – open file for reading and writing
  - `w` – overwrite file or create file for writing
  - `w+` – open for reading and writing; overwrites file
  - `a` – open file for appending (writing at end of file)
  - `a+` – open file for appending and reading
- `fclose(file_stream);`
  - Closes open file stream

Using datatypes with file I/O

- All the functions we've seen so far use raw bytes for file I/O, but program data is usually stored in meaningful datatypes (int, char, float, etc.)
- `fprintf(), fputs(), fputc()`
  - Used to write data to a file
- `fscanf(), fgets(), fgetc()`
  - Used to read data from a file
- man <*function*> (for more info)

printf()

- `printf(format_string,x,y,...)`
- `format_string`: string that describes the output information
  - formatting commands are escaped with %
- `x,y,...`: values to print according to formatting commands in `format_string`

Formatting Commands

- %d, %i – decimal integer
- %u – unsigned decimal integer
- %o – unsigned octal integer
- %x, %X – unsigned hexadecimal integer
- %c - character
- %s – string or character array
- %f – float
- %e, %E – double (scientific notation)
- %g, %G – double or float
- %% - print a % character

`printf()` Examples

- `printf("The sum of %d, %d, and %d is %d\n", 65, 87, 33, 65+87+33);`
  - Output: The sum of 65, 87, and 33 is 185
- `printf("Character code %c has ASCII code %d.\ n", 'A', 'A');`
  - Output: Character code A has ASCII code 65.
- `printf("Error %s occurred at line %d\n", emsg, lno);`
  - `emsg` and `lno` are variables
  - Output: Error invalid variable occurred at line 27

`printf()` Examples (cont.)

- `printf("Octal form of %d is %o \n", 59, 59);`
  - Output: Octal form of 59 is 73
- `printf("Hexadecimal form of %d is %x \n", 59, 59);`
  - Output: Hexadecimal form of 59 is 3B
- `printf("Square root of 2 is %f \n", sqrt( 2.0));`
  - Output: Square root of 2 is 1.414214

`printf()` Examples (cont.)

- `printf("Square root of 157 is %e \n", sqrt( 157.0));`
  - Output: Square root of 157 is 1.252996e+01
- `printf("You scored %d out of %d for %d%%.\n", 17, 25, 68);`
  - Output: You scored 17 out of 25 for 68%.

`scanf()`

- `scanf(format_string,&x,&y,...)`
- Similar syntax to `printf`, except `format_string` specifies data to read
- **Must** pass variables by reference
- Example
  - `scanf("%d %c %s", &int_var,`
  - `&char_var, string_var);`

printf() and **scanf()** Families

- `fprintf(file_stream, format_string, ...)`
  - Prints to a file stream instead of stdout
- `sprintf(char_array, format_string, ...)`
  - Prints to a character array instead of stdout
- `fscanf(file_stream, format_string, ...)`
  - Reads from a file stream instead of stdin
- `sscanf(char_array, format_string, ...)`
  - Reads from a string instead of stdin