CS 105

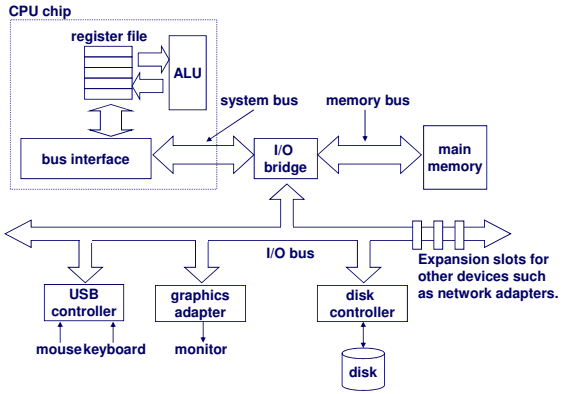
"Tour of the Black Holes of Computing"

Input and Output

Topics

- I/O hardware
- Unix file abstraction
- Robust I/O
- File sharing

I/O: A Typical Hardware System



Abstracting I/O

Low level requires complex device commands

- Vary from device to device
- Device models can be very different
  - Tape: read or write sequentially, or rewind
  - Disk: "random" access at block level
  - Terminal: sequential, no rewind, must echo and allow editing
  - Video: write-only, with 2-dimensional structure

Operating system should hide these differences

- "Read" and "write" should work regardless of device
- Sometimes impossible to generalize (e.g., video)
- Still need access to full power of hardware

Unix Files

A Unix *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/sda1` (`/boot` disk partition)
- `/dev/tty2` (terminal)

Even the kernel is represented as files:

- `/dev/kmem` (access to kernel memory)
- `/proc` (kernel data structures)
- `/sys` (device discovery and control)

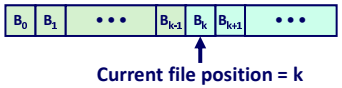
Unix I/O Overview

Elegant mapping of files to devices allows kernel to export a simple interface called Unix I/O

Key Unix idea: All input and output is handled in a consistent and uniform way

Basic Unix I/O operations (system calls):

- Opening and closing files: `open()` and `close()`
- Reading and writing a file: `read()` and `write()`
- Changing the *current file position* (seek): `lseek` (not discussed)



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 character* (`'\n'`)
  - Newline is `0xa`, same as ASCII line feed character (LF)
- Note that a proper text file always ends with a newline!

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 a dictionary of *links*

- Each link maps a *filename* to a file

Each directory contains at least two entries

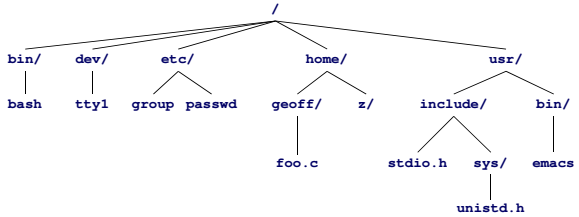
- `.` (dot) is a link to itself
- `..` (dot dot) is a link to *the parent directory* in the *directory hierarchy* (next slide)

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)



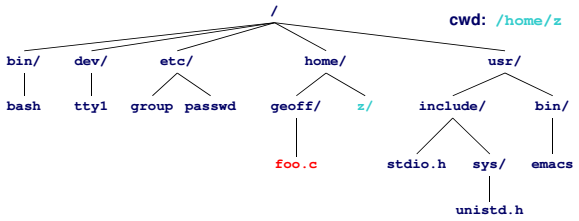
Kernel maintains *current working directory (cwd)* for each process

- Modified using the `cd` command

Pathnames

Locations of files in the hierarchy denoted by *pathnames*

- ***Absolute pathname*** starts with '/' and denotes path from root
  - `/home/geoff/foo.c`
- ***Relative pathname*** denotes path from current working directory
  - `../geoff/foo.c`



Opening Files

```
#include <errno.h>
...
    int fd;   /* file descriptor */

    fd = open("/etc/hosts", O_RDONLY);
    if (fd == -1) {
        fprintf(stderr, "Couldn't open /etc/hosts: %s", strerror(errno));
        exit(1);
```

Opening a file tells kernel you are getting ready to access it

Returns small identifying integer *file descriptor*

- `fd == -1` indicates that an error occurred; `errno` has reason
- `strerror` converts to English (Note: use `strerror_r` for thread safety)

Each process created by a Unix shell begins life with three open files (normally connected to terminal):

- 0: standard input
- 1: standard output
- 2: standard error

Redirecting Files

One of the most powerful ideas in Unix

You can easily redirect `stdin/stdout/stderr`

- `./echoclient < /etc/passwd` redirects input
- `grep knuth /etc/hosts > ~/knuthip` redirects output
- `ls -Rl /proc 2> /dev/null` redirects error

You can even hook programs together ("piping"):

- `find / -name core | wc -l`
- `find / -name core -print0 | xargs -0 rm -f`

You're not true Unix expert until you're good with pipes

- Two-command pipes: advanced learner
- Three commands: excellent competence
- Six or more: scary ninja
- `cat foo | bar` is *always* incorrect (and sign of ignorance)
  - Use `bar < foo` instead
  - `Don't let stackoverflow fool you!`

Closing Files

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

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

Closing a file tells kernel that you're finished with it

Closing an already closed file is recipe for disaster in threaded programs (more on this later)

Some error reports are delayed until close!

Moral: Always check return codes, even for seemingly benign functions such as `close()`

`perror` is simplified `strerror/fprintf`; see man page

Reading Files

```
char buf[4096];
int fd;                 /* file descriptor */
unsigned int nbytes;    /* number of bytes read */

/* Open file fd ...  */
/* Then read up to 4096 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof buf)) == -1) {
   perror("read");
   exit(1);
}
```

Reading a file copies bytes from *current file position* into memory, then *updates* file position

You must provide the memory (buffer)

Returns number of bytes read from file `fd` into `buf`

- `nbytes == -1` indicates error occurred
- `nbytes == 0` indicates end of file (EOF)
- *Short counts* (`nbytes < sizeof buf`) are possible and are not errors!

Writing Files

```
char buf[4096];
int fd;                 /* file descriptor */
unsigned int nbytes;    /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 4096 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof buf) == -1) {
   perror("write");
   exit(1);
}
```

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

Returns number of bytes written from `buf` to file `fd`

- `nbytes == -1` indicates that an error occurred
- `nbytes == 0` will never happen
- As with reads, short counts are possible and are not errors!

This example transfers *up to* 4096 bytes from address `buf` to file `fd`

Simple Unix I/O Example

```
#include "csapp.h"

int main(void)
{
    char c;

    while(Read(STDIN_FILENO, &c, 1) > 0)
        Write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

(Inefficiently) copies standard input to standard output one byte at a time (basically, this is `cat`)

Note the use of error-handling wrappers for read and write (Appendix B in text)

Dealing with 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 or Unix pipes

Short counts never occur in these situations:

- Reading from disk files, except for EOF
- Writing to disk files

How should you deal with short counts in your code?

- Use the RIO (Robust I/O) package from your textbook's `csapp.c` file (Appendix B)
  - (But note that it handles EOF wrong on terminal)
- Use C stdio or C++ streams (also sometimes blows EOF!)
- Write your code very, very carefully
- Ignore the problem and accept that your code is fragile

"Foolproof" I/O

Low-level I/O is difficult because of short counts and other possible errors

Textbook provides RIO package, a (fairly) good example of how to encapsulate low-level I/O

RIO is set of wrappers that provide efficient and robust I/O in applications (e.g., network programs) that are subject to short counts.

Download from `csapp.cs.cmu.edu/public/ics2/code/src/csapp.c`
`csapp.cs.cmu.edu/public/ics2/code/include/csapp.h`

Unbuffered I/O

RIO provides buffered and unbuffered routines

Unbuffered:

- Especially useful for transferring data on network sockets
- Same interface as Unix `read` and `write`
- `rio_readn` returns short count only if it encounters EOF
  - Usually incorrect if reading from terminal
- `rio_writen` never returns a short count
- Calls to `rio_readn` and `rio_writen` can be interleaved arbitrarily on the same descriptor
- Small unbuffered I/Os are horribly inefficient

Buffered I/O: Motivation

Applications often read/write one character at a time

- `getc, putc, ungetc`
- `gets, fgets`
  - Read line of text one character at a time, stopping at newline

Implementing that as Unix I/O calls is expensive

- `read` and `write` require Unix kernel calls
  - > 10,000 clock cycles *per character*

Solution: Buffered read

- Use Unix `read` to grab block of bytes
- User input functions take one byte at a time from buffer
  - Automatically refill buffer when empty

Buffered Input

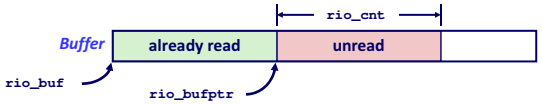
Buffered:

- *Efficiently* read text lines and binary data from file partially cached in an internal memory buffer
- `rio_readlineb` reads text line of up to `maxlen` bytes from file `fd` and stores it in `usrbuf`. Especially useful for reading lines from network sockets
- `rio_readnb` reads up to `n` bytes from file `fd`
- Calls to `rio_readlineb` and `rio_readnb` can be interleaved arbitrarily on same descriptor
  - Warning: Don't intermix calls to `rio_readn` with calls to `*b` versions

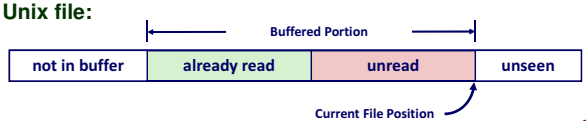
Buffered I/O: Implementation

For reading from file

File has associated buffer to hold bytes that have been read from file but not yet read by user code

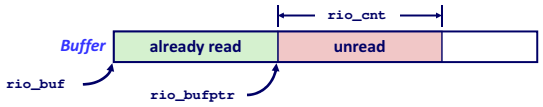


Layered on Unix file:



Buffered I/O: Declaration

All information contained in `struct`




```
typedef struct {
    int rio_fd;                /* descriptor for this internal buf */
    int rio_cnt;               /* unread bytes in internal buf */
    char *rio_bufptr;          /* next unread byte in internal buf */
    char rio_buf[RIO_BUFSIZE]; /* internal buffer */
} rio_t;
```


Buffered RIO Example

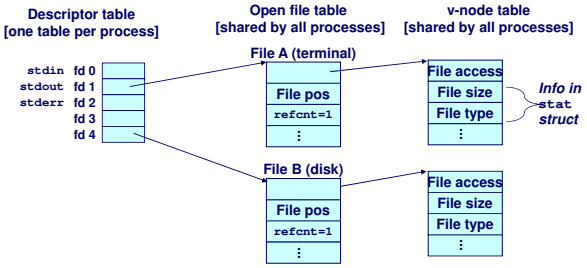
Copying the lines of a text file from standard input to standard output

```
#include "csapp.h"

int main(int argc, char **argv)
{
    int n;
    rio_t rio;
    char buf[MAXLINE];

    Rio_readinitb(&rio, STDIN_FILENO);
    while(1) {
        n = Rio_readlineb(&rio, buf, sizeof buf);
        if (n == 0)
            break;
        Rio_writen(STDOUT_FILENO, buf, n);
    }
    exit(0);
}
```


How the Unix Kernel Represents Open Files

Two descriptors referencing two distinct open files

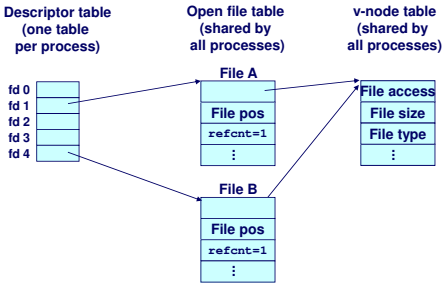
Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



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

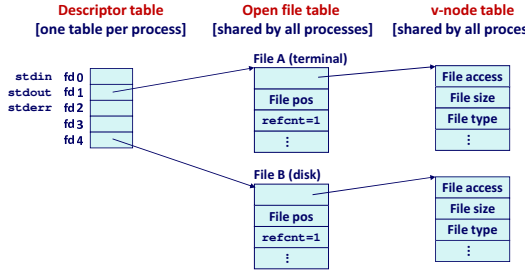


How Processes Share Files: `fork`

A child process inherits its parent's open files

- Note: situation unchanged by `exec` functions (use `fcntl` to change)

Before `fork` call:

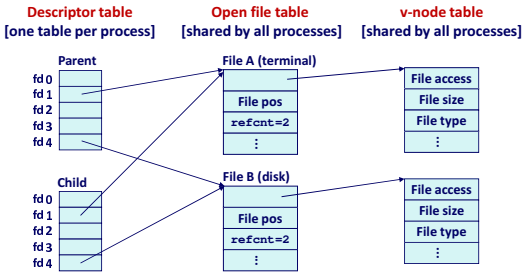


How Processes Share Files: `fork`

A child process inherits its parent's open files

After `fork` call:

- Child's table same as parent's; add +1 to each refcnt



File Metadata

Metadata is data about data, in this case file data.

Maintained by kernel, accessed by users with the `stat` and `fstat` functions.

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;     /* device */
    ino_t         st_ino;     /* inode */
    mode_t        st_mode;    /* protection and file type */
    nlink_t       st_nlink;   /* number of hard links */
    uid_t         st_uid;     /* user ID of owner */
    gid_t         st_gid;     /* group ID of owner */
    dev_t         st_rdev;    /* device type (if inode device) */
    off_t         st_size;    /* total size, in bytes */
    unsigned long st_blksize; /* blocksize for filesystem I/O */
    unsigned long st_blocks;  /* number of blocks allocated */
    time_t        st_atime;   /* time of last access */
    time_t        st_mtime;   /* time of last modification */
    time_t        st_ctime;   /* time of last change */
};
```

Standard I/O Functions

The C standard library (`libc.so`) contains a collection of higher-level *standard I/O* functions

- Documented in Appendix B of K&R

Examples of standard I/O functions:

- Opening and closing files (`fopen` and `fclose`)
- Reading and writing bytes (`fread` and `fwrite`)
- Reading and writing text lines (`fgets` and `fputs`)
- Formatted reading and writing (`fscanf` and `fprintf`)

Standard I/O Streams

Standard I/O models open files as *streams*

- Abstraction for a file descriptor and a buffer in memory

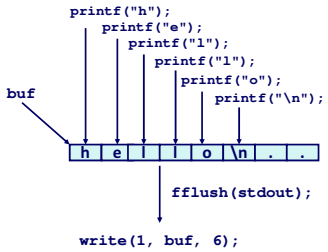
C programs begin life with three open streams (defined in `stdio.h`)

- `stdin` (standard input)
- `stdout` (standard output)
- `stderr` (standard error)

```
#include <stdio.h>
extern FILE *stdin;  /* standard input  (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error  (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```


Buffering in Standard I/O

Standard I/O functions use buffered I/O



Buffer flushed to output fd on '\n' (if terminal), call to `fflush` or `exit`, or return from `main`.

Standard I/O Buffering in Action

You can see this buffering in action for yourself, using the always fascinating Linux `strace` program:

```
#include <stdio.h>

int main()
{
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    printf("\n");
    fflush(stdout);
    exit(0);
}
```

```
linux> strace ./hello
execve("./hello", ["hello"], [/* ... */]).
...
write(1, "hello\n", 6)                  = 6
...
exit_group(0)                           = ?
```

Aside: Working with Binary Files

Functions you should never use on binary files

- Text-oriented I/O such as `fgets, scanf, rio_readlineb`
  - Interpret EOL characters.
  - Use functions like `rio_readn` or `rio_readnb` instead
- String functions
  - `strlen, strcpy, strcat`
  - Interprets byte value 0 (end of string) as special