# Systems Programming

## Input and Output

**Haehyun Cho**

School of Software
Soongsil University

Cyber Security
Research Center

# I/O Kernel Services

We have seen some text I/O using the C Standard Library.

- fread()
- fgets()
- printf()
- ...

However, all I/O is built on kernel system calls.

In this lecture, we'll look at those services *vs.* standard I/O.

# Everything is a File

These services are particularly important on UNIX systems.

On UNIX, “everything is a file”.

Many devices and services are accessed by opening device nodes that behave like files.

Examples:

- /dev/null: Always readable, contains no data. Always writable, discards anything written to it.
- /dev/urandom: Always readable, reads a cryptographically secure stream of random data.

# Everything is a File

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

- 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
  - *Symbolic links*
  - *Character and block devices*
  - …

# Regular Files

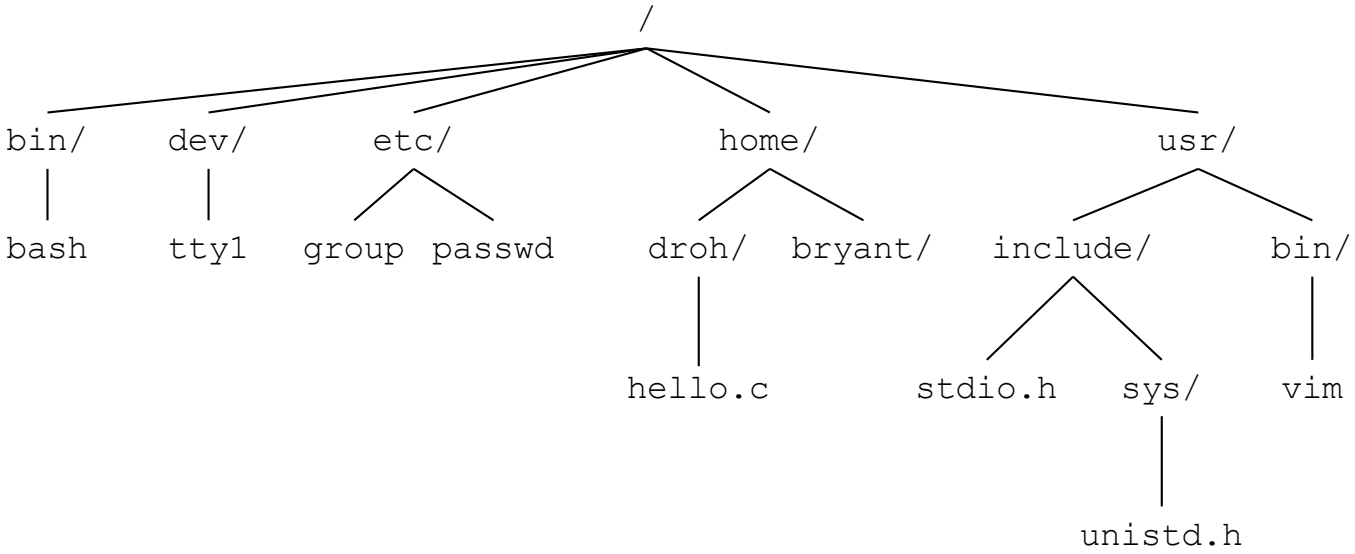
- A regular file contains arbitrary data
- Applications often distinguish between *text files* and *binary files*
  - Text files are regular files with only ASCII or Unicode characters
  - Binary files are everything else
    - e.g., object files, JPEG images
  - Kernel doesn't know the difference!
- 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

# File Descriptors

All access to files is through file descriptors.

A file descriptor is a small integer representing an open file in a particular process.

There are three “standard” file descriptors:

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

...sound familiar? (stdin, stdout, stderr)

# File Modes

Every file on a POSIX system has an owner and group.

File permissions are handled by mode bits.

- Modern POSIX systems also have access control lists.

Mode bits are of the form: rwxrwxrwx

The rwx triplets are user, group, and other permissions.

- The user bits apply to the file's owner.
- The group bits apply to members of the file's group.
- The other bits apply to all other users.

`r` means read, `w` means write, `x` execute.

# Mode Examples

File modes are normally represented as octal numbers.

Octal numbers range from 0-7 and are three bits long.

- `rwxrwxrwx`

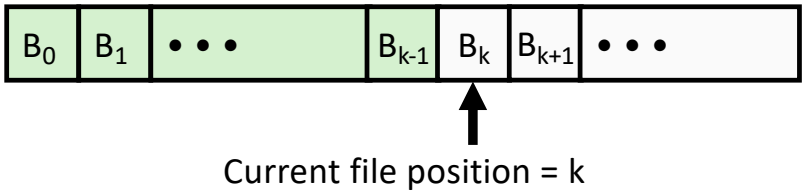
Examples:

750 (111101000b): `rwxr-x---`
User can read, write, execute; group can read and execute; others have no access.

664 (110110100b): `rw-rw-r--`
User and group can read and write, others can read.

# 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()`**



# System Call Failures

Kernel I/O system calls (indeed, most system calls) return a negative integer on failure.

When this happens, the global variable errno is set to a reason.

Include `errno.h` to define `errno` in your code.

The functions `perror()` and `strerror()` produce a human-readable error from errno.

# Opening Files

There are two calls to open a file on a POSIX system:

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

The `creat()` system call is exactly like calling:

```
open(path, O_CREAT|O_WRONLY|O_TRUNC, mode);
```

Both functions return a filedescriptor on success.

# Open Flags

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

The flags parameter controls how open() behaves:

- O_RDONLY: Open read-only
- O_WRONLY: Open write-only
- O_RDWR: Open for reading and writing
- O_CREAT: When writing, create the file if it doesn't exist
- O_EXCL: When creating a file, fail if it already exists
- O_APPEND: When writing, start at the end of the file
- O_TRUC: When writing, truncate the file to 0 bytes
- O_CLOEXEC: Close this file on exec()

# O_CREAT|O_EXCL

The combination of flags O_CREAT|O_EXCL allows for exclusive access among cooperating processes.

The kernel will create the file if and only if it doesn't already exist.

- This is an atomic action:

  An atomic action appears to be indivisible from the outside.

- If every process uses O_CREAT|O_EXCL for a file, the file can be used as a lock.
  (More about locking later ...)

# Reading

```
#include <unistd.h>
int read(int fd, void *buffer, size_t bytes);
```

The read() system call reads data from an open file.

It reads raw bytes with no translation!

In particular, it will (maybe) not read a NUL-terminated string.

Its return value is:

- `0`: end of file
- `>0`: bytes read; EOF if `<` bytes
- `<0`: error

# Writing

```
#include <unistd.h>
int write(int fd, const void *buffer, size_t
bytes);
```

The write() sytem call writes data to an open file.

Like read(), it deals in raw binary data.

Its return value is:

≥0: bytes written; full disk / *etc.* if < bytes

<0: error

# Closing File Descriptors

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

An open file can be closed with the close() system call.

Using a descriptor after close is an error.

A closed descriptor may be reused by subsequent opens.

# UNIX I/O Example

```
#include <unistd.h>
#include <fcntl.h>
int main(int argc, char *argv[]) {
      char buf[1024];
      int fd, bytes;
      if ((fd = open(argv[1], O_RDONLY)) < 0)
            { return -1; }
      while ((bytes = read(fd, buf, sizeof(buf))) > 0) {
            if (write(1, buf, bytes) < 0) {
                  return -1; }
      }
      return bytes >= 0;
}
```

# UNIX I/O Example with `perror();`

```
if ((bytes = read(fd, buf, sizeof(buf))) < 0) {
    perror("read");
    exit(1);
}

if ((bytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

# Opening Streams

- A standard I/O stream wraps a file descriptor.

```
#include <stdio.h>
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
```

`fopen()` opens a file, `fdopen()` wraps an open file descriptor.

The mode parameter here confusingly corresponds to open flags.

# Stream Modes

```
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
```

A stream can be opened for various purposes, according to mode:

- "r": reading
- "w": writing, with truncation
- "a": writing, without truncation (append)
- "r+": reading and writing, without truncation
- "w+": reading and writing, with truncation

Write modes always create the file if necessary.

# Binary I/O

The standard I/O functions may perform transformations.

They may assume that they operate on text files.

You can open for binary I/O using `"b"` after the mode character:

```
fopen("somefile", "rb");
```

On POSIX systems, the `"b"` is ignored.

This is a feature of the C Standard that is unused on POSIX systems.

# Reading and Writing

```
#include <stdio.h>
size_t fread(void *dest, size_t size, size_t nmemb, FILE *fp);
size_t fwrite(const void *buf, size_t size, size_t nmemb, FILE *fp);
```

These functions read and write binary data.
(This is in contrast to the string I/O functions.)

Both write in terms of items of size bytes.

The return value is:

- the number of items read/written (up to nmemb)
- 0 on error or EOF

# Errors and EOF

Unlike UNIX I/O, errors and EOF return the same value.

There are two functions provided to detect errors and EOF:

- `int feof(FILE *fp);`
- `int ferror(FILE *fp);`

These functions return non-zero if EOF or an error has occurred.

`clearerr()` will reset the error/EOF status of a stream:

- `void clearerr(FILE *fp);`

# Standard I/O Example

```
#include <stdio.h>
int main(int argc, char *argv[]) {
      char buf[1024];
      FILE *fp; int bytes;
      if ((fp = fopen(argv[1], "r")) == NULL) {
            return -1; }
      while (!feof(fp) &&
                  (bytes = fread(buf, 1, sizeof(buf), fp)) > 0) {
            if (fwrite(buf, 1, bytes, stdout) == 0) {
                  return -1;
            }
      }
      return ferror(fp) || ferror(stdout);
}
```

# System Call Overhead

The overhead of calling a system call is often not small.

This overhead is due to the cost of:

- Changing protection domains
- Validating pointers
- Adjusting memory maps
- ...

It is better to make fewer system calls that do more work.

# Standard I/O Buffering

The standard I/O functions use buffering to reduce overhead.

For example, `fread()` for 1 byte *might* read a full disk block.

This has important implications for correctness!

For example, device I/O may require very precise read/write sizes.

Write buffering can cause short writes.

Buffer flushing fixes this short write problem:

```
int fflush(FILE *fp);
```

# Buffering and Performance: UNIX I/O

```
int fd = open("megabyte.dat", O_RDONLY);
int total;
unsigned char c;

while (read(fd, &c, 1) == 1) {
    total += c;
}
```

Time:

```
Real time elapsed: 0:00.58
System time used : 0.41
User time used   : 0.16
```

# Buffering and Performance: Standard I/O

```
FILE *fp = fopen("megabyte.dat", "rb");
int total;
unsigned char c;

while (!ferror(fp) && fread(&c, 1, 1, fp) == 1) {
    total += c;
}
```

Time:

```
Real time elapsed: 0:00.02
System time used : 0.00
User time used   : 0.02
```

# Buffer Example

```
fread(&len, sizeof(len), 1, fp);
data = malloc(len);
fread(&data, 1, len, fp);
```

Standard I/O buffer for fp:

# Buffer Example

```
fread(&len, sizeof(len), 1, fp);
data = malloc(len);
fread(&data, 1, len, fp);
```

Standard I/O buffer for `fp`:

First, `fread` reads a buffer of data from `fp`.

# Buffer Example

```
fread(&len, sizeof(len), 1, fp);
data = malloc(len);
fread(&data, 1, len, fp);
```

Standard I/O buffer for fp:

Then it returns `sizeof(size_t)` bytes from that buffer.

# Buffer Example

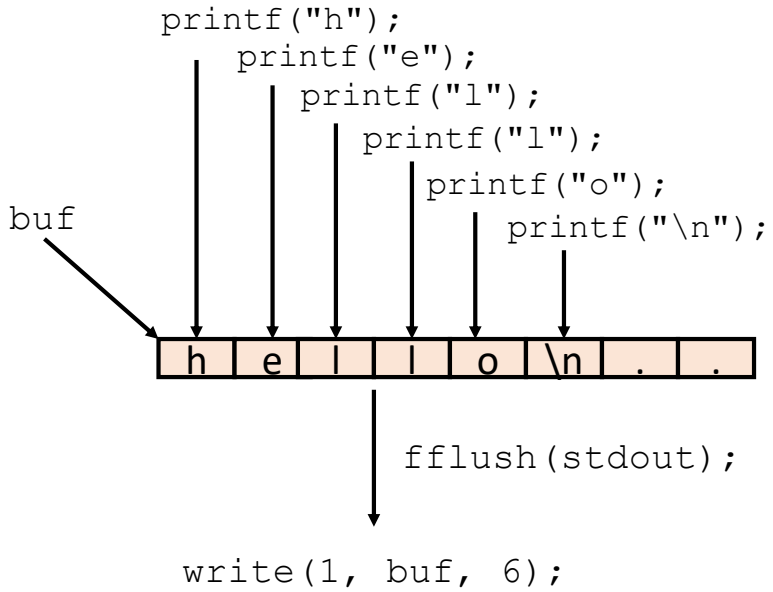
```
fread(&len, sizeof(len), 1, fp);
data = malloc(len);
fread(&data, 1, len, fp);
```

Standard I/O buffer for fp:

The next read reads only from the buffer.

# Buffer Example 2

Buffer flushed to output fd on "\n", call to `fflush` or `exit`, or return from `main`.



# Buffer Example 2

```
#include <stdio.h>

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

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

# Summary

- UNIX I/O is defined by the POSIX Standard
- Standard I/O is defined by the C Standard
- All file I/O goes through the kernel
- The standard I/O library is buffered