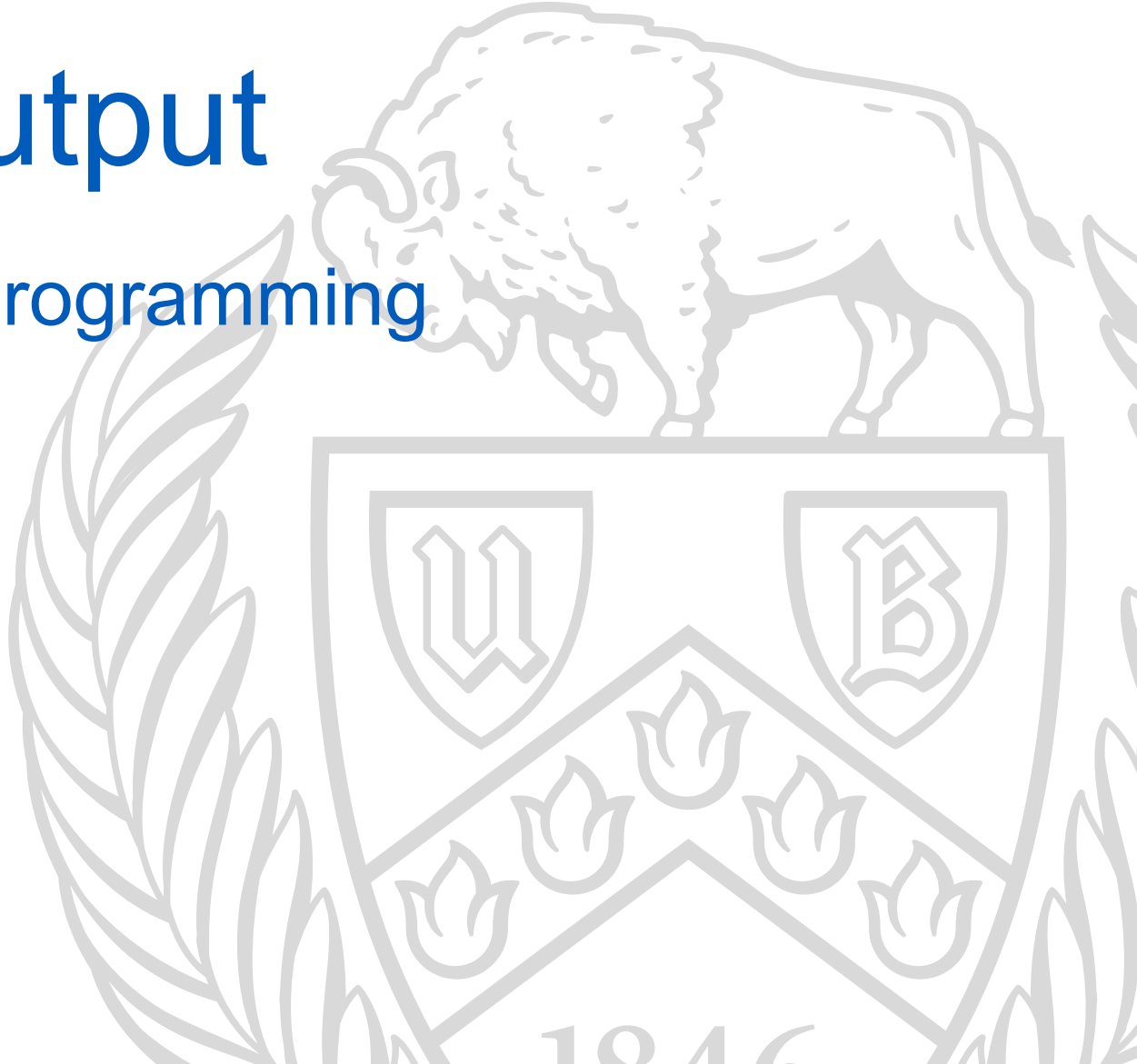
Input and Output

CSE 220: Systems Programming

Ethan Blanton
Jesse Hartloff

Computer Science and Engineering
University at Buffalo



I/O Kernel Services

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

- `printf()`
- `fgetc()`
- …

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

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

© 2026 Ethan Blanton, Jesse Hartloff / CSE 220: Systems Programming

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*.

Device nodes behave like (but are not) 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.

File Descriptors

All access to files is through *file descriptors*.

A file descriptor is a small integer representing alert{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`)

System Call Failures

Kernel I/O (and most other) system calls return −1 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`.

Lecture Question

Ask a review question!

Opening Files

There are two[1] 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 file descriptor on success.

[1] … OK, three. Three-ish.

© 2026 Ethan Blanton, Jesse Hartloff / CSE 220: Systems Programming

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_TRUNC`: When writing, truncate the file to 0 bytes
- `O_CLOEXEC`: Close this file on `exec()`

© 2026 Ethan Blanton, Jesse Hartloff / CSE 220: Systems Programming

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.

If every process uses O_CREAT|O_EXCL for a file, the file can be used as a lock.

Removing the file (also atomic) is the unlock operation!

© 2026 Ethan Blanton, Jesse Hartloff / CSE 220: Systems Programming

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 may not read a NUL-terminated string.

Its return value is:

- 0: end of file
- > 0: bytes read; EOF if $< \text{bytes}$
- -1: error

© 2026 Ethan Blanton, Jesse Hartloff / CSE 220: Systems Programming

Writing

```c
#include <unistd.h>

int write(int fd, const void *buffer, size_t bytes);
```

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

Its return value is:

- ≥ 0: bytes written; full disk / *etc.* if $< \text{bytes}$
- -1: error

© 2026 Ethan Blanton, Jesse Hartloff / CSE 220: Systems Programming

Closing File Descriptors

```c
#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

```c
#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;
}
```

Standard I/O? What Standard?

If Unix I/O is part of the POSIX Standard …

Standard I/O is part of the C Standard.

Non-POSIX systems will still have standard I/O!

On Unix systems, the standard I/O functions wrap Unix I/O.

© 2026 Ethan Blanton, Jesse Hartloff / CSE 220: Systems Programming

Opening Streams

An open file in standard I/O is called a *stream*.

On Unix, a 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`.

© 2026 Ethan Blanton, Jesse Hartloff / CSE 220: Systems Programming

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.

© 2026 Ethan Blanton, Jesse Hartloff / CSE 220: Systems Programming

Binary I/O

Unlike Unix I/O, standard I/O 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

```
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

© 2026 Ethan Blanton, Jesse Hartloff / CSE 220: Systems Programming

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

```c
#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);
}
```

Lecture Question

Ask a lecture question!

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 VM page.

This has important implications for correctness!

For example, device I/O may require very precise I/O 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 = 0;
unsigned char c;

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

```
Real time elapsed: 0:00.84
System time used : 0.42
User time used   : 0.41
```


© 2026 Ethan Blanton, Jesse Hartloff / CSE 220: Systems Programming

Buffering and Performance: Standard I/O

```c
FILE *fp = fopen("megabyte.dat", "rb");
int total = 0;
unsigned char c;

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

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

What's the Difference?

`read()`:

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
100.00    5.180035           4   1048578           read
  0.00    0.000006           3         2           close
------ ----------- ----------- --------- --------- ----------------
100.00    5.180041           4   1048580           total
```

`fread()`:

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0       258           read
  0.00    0.000000           0         2           close
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000           0       260           total
```


© 2026 Ethan Blanton, Jesse Hartloff / CSE 220: Systems Programming

Buffering Mechanism

When the user requests a small read, the standard library makes a larger read.

For example, our reads of one byte turn into 4 kB reads.

The standard library buffers the remaining data in memory.

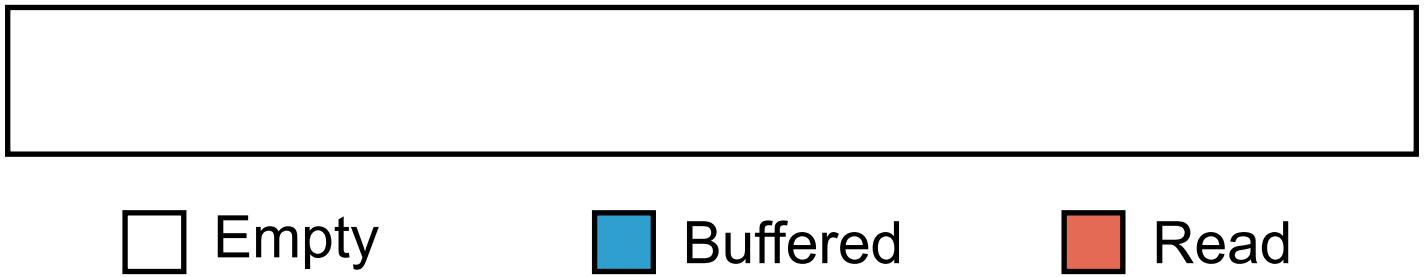
Future reads for buffered data read from memory.

Reads for data not in the buffer cause a new buffer to be fetched.

© 2026 Ethan Blanton, Jesse Hartloff / CSE 220: Systems Programming

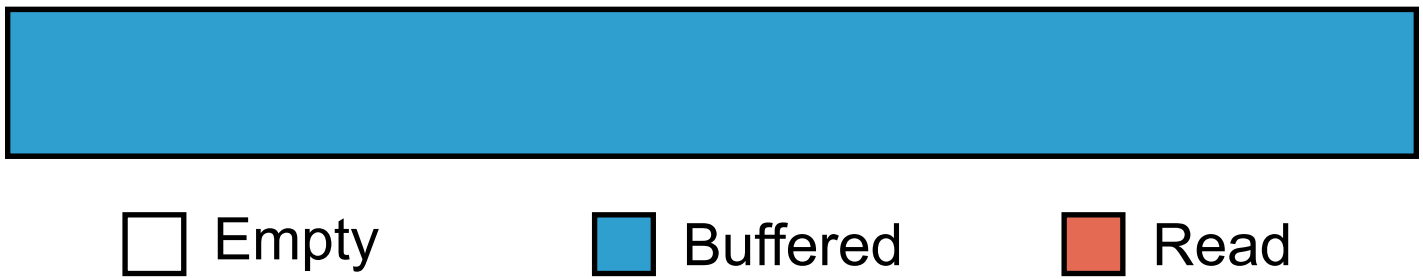
Buffer Example

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



Buffer Example

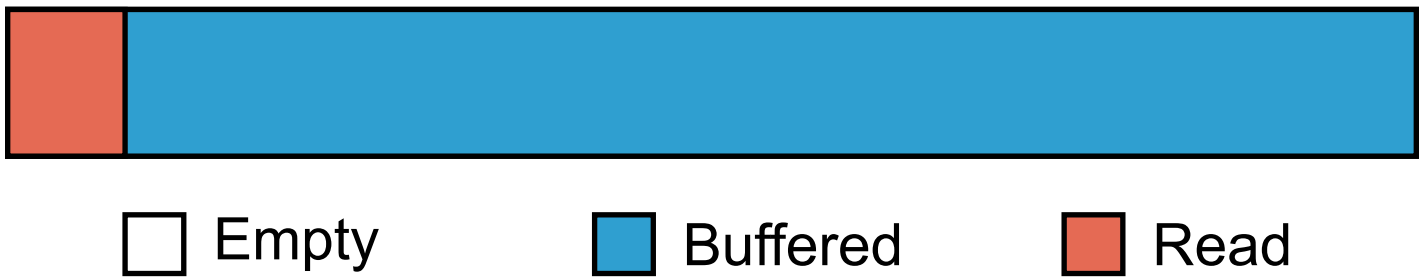
```
fread(&len, sizeof(len), 1, fp);
data = malloc(len);
fread(&data, 1, len, fp);
```



`fread()` triggers a read from `fp` into a memory buffer

Buffer Example

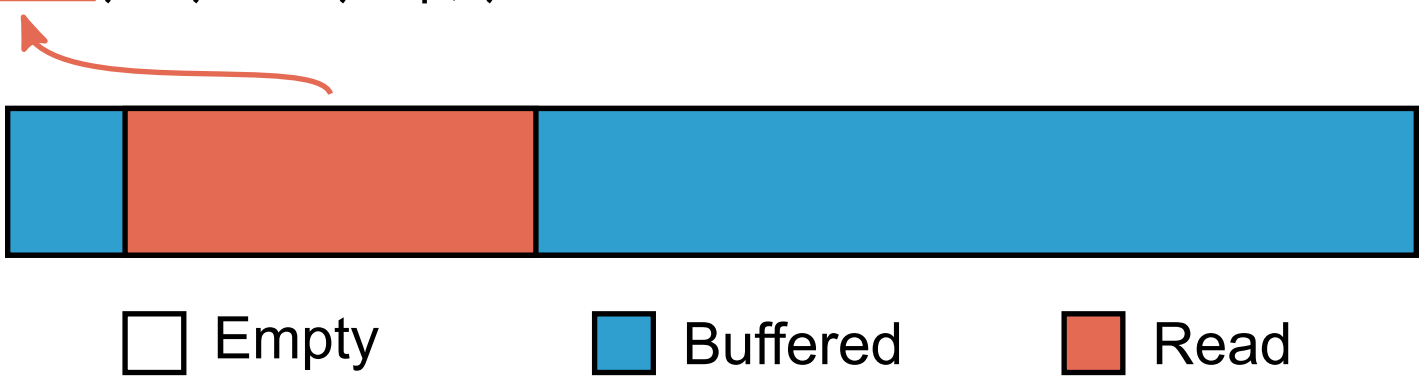
```
fread(&len, sizeof(len), 1, fp);
data = malloc(len);
fread(&data, 1, len, fp);
```



`sizeof(len)` bytes from the buffer are copied into `len`

Buffer Example

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



The next read only copies data from the buffer.

Summary

- Unix I/O is defined by the POSIX Standard
- Standard I/O is defined by the C Standard
- The kernel tracks open files with file descriptors
- All file I/O goes through the kernel
- The standard I/O library is buffered

© 2026 Ethan Blanton, Jesse Hartloff / CSE 220: Systems Programming

Bibliography

Required Readings

[1] Ian Weinand. *Computer Science from the Bottom Up*. Chapter 1: part 1. No date.

Optional Readings

[2] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 10: 10.1-10.4, 10.10-10.12. Pearson, 2016.

Copyright

Copyright 2018–2026 Ethan Blanton, All Rights Reserved.

These slides include material copyright 2022–2026 Carl Alphonce, with permission.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see https://www.cse.buffalo.edu/courses/cse220/.