Lecturer: Dr. Michael J. May
Kinneret College

I/O Basics, High and low, Drivers, Sockets

15 November 2026
Lecture 5

Slides adapted from John Kubiatowicz (UC Berkeley)

Concept Review

Segments	Process context	Interrupt context	Interrupt handler
Kernel Stack	Process Control Block	Scheduler	Simultaneous Multi Threading (SMT)
	fork	pid_t	


Socket
operations
high
Drivers
Read
Support
kernel
ssizet
Streams
Basic
standard
networks
processes
API
IO
like
sizet
File
fp
C
Level
Write
void
error
include
stream
device
writewfd
outfile
char
Sockets
Driver
Files
const
data
system
LOW
rmax
BUFSIZE
descriptors
stdioh
nreadrfd
wbuf
UNIX
bytes
abstraction
restrict
filedes

Topics for Today

- Basic Support for I/O (drivers, etc.)
  - Files and Streams
  - Low Level
- I/O and Drivers
- Sockets and networks

Key Unix I/O Design Concepts

Uniformity



- file operations, device I/O, and interprocess communication through open, read/write, close
- Allows simple composition of programs
  - `find | grep | wc …`

Open before use



- Provides opportunity for access control and arbitration
- Sets up the underlying machinery, i.e., data structures

Byte-oriented

- Even if blocks are transferred, addressing is in bytes



https://www.pinterest.com/pin/352548095578261411/

Key Unix I/O Design Concepts

Kernel buffered reads

- Streaming and block devices looks the same
- read blocks process, yielding processor to other task



Kernel buffered writes

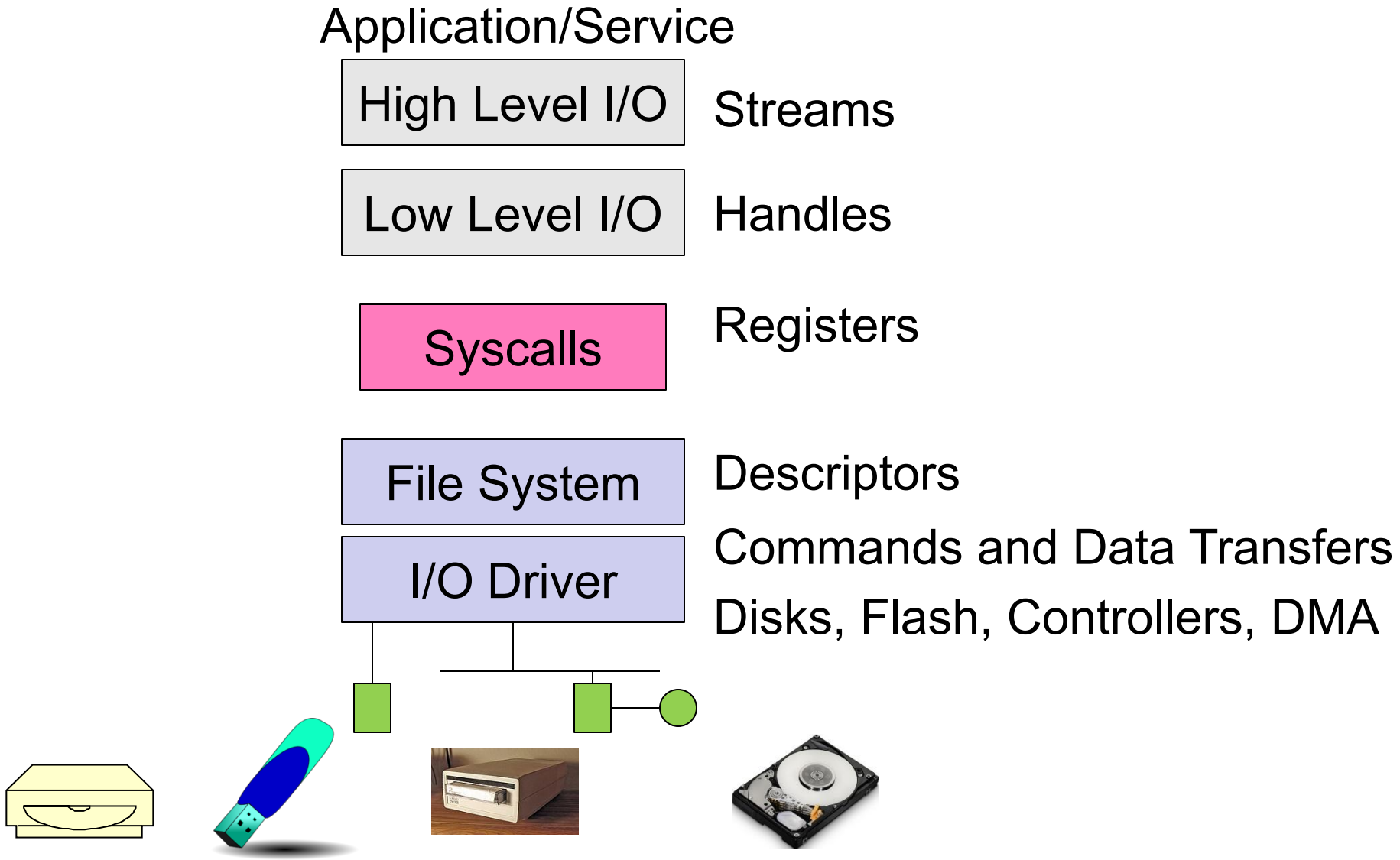
- Completion of out-going transfer decoupled from the application, allowing it to continue





Explicit close

I/O & Storage Layers



The file system abstraction

- File 

Named collection of data in a file system

File data

- Text, binary, linearized objects

File Metadata:

- Size, Modification Time, Owner, Security info
- Basis for access control

- Directory 

"Folder" containing files & Directories

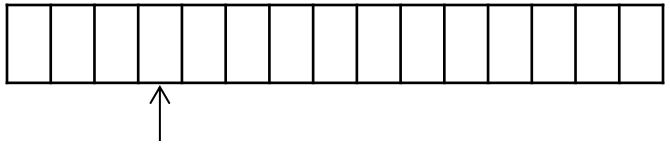
Hierarchical (graphical) naming

- Path through the directory graph
- Uniquely identifies a file or directory
  - `/home/mjmay/se317/public_html/fa25/index.html`

Links and Volumes (later)

C high level File API – streams

- Operate on “streams” - sequence of bytes, whether text or data, with a position



```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
int fclose( FILE *fp );
```

Mode Text	Binary	Description
r	rb	Open existing file for reading
w	wb	Open for writing; create if doesn’t exist
a	ab	Open for appending; create if doesn’t exist
r+	rb+	Open existing file for reading and writing
w+	wb+	Open for reading and writing; truncated to zero if exists, create otherwise
a+	ab+	Open for reading and writing; create if doesn’t exist. Read from beginning, write as append

Connecting Processes, Filesystem, and Users

Every process has a “current working directory”

Absolute Paths

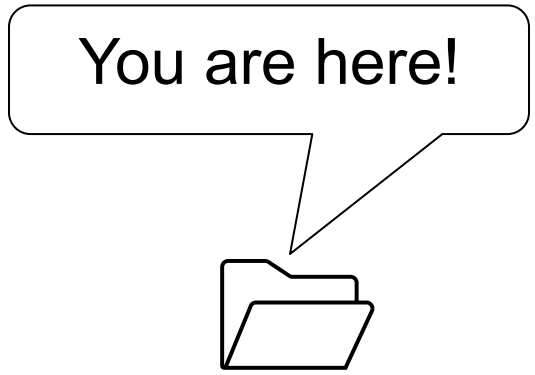
- `/home/mjmay/se317`

Relative paths

- `index.html, ./index.html` - current WD
- `../index.html` - parent of current WD

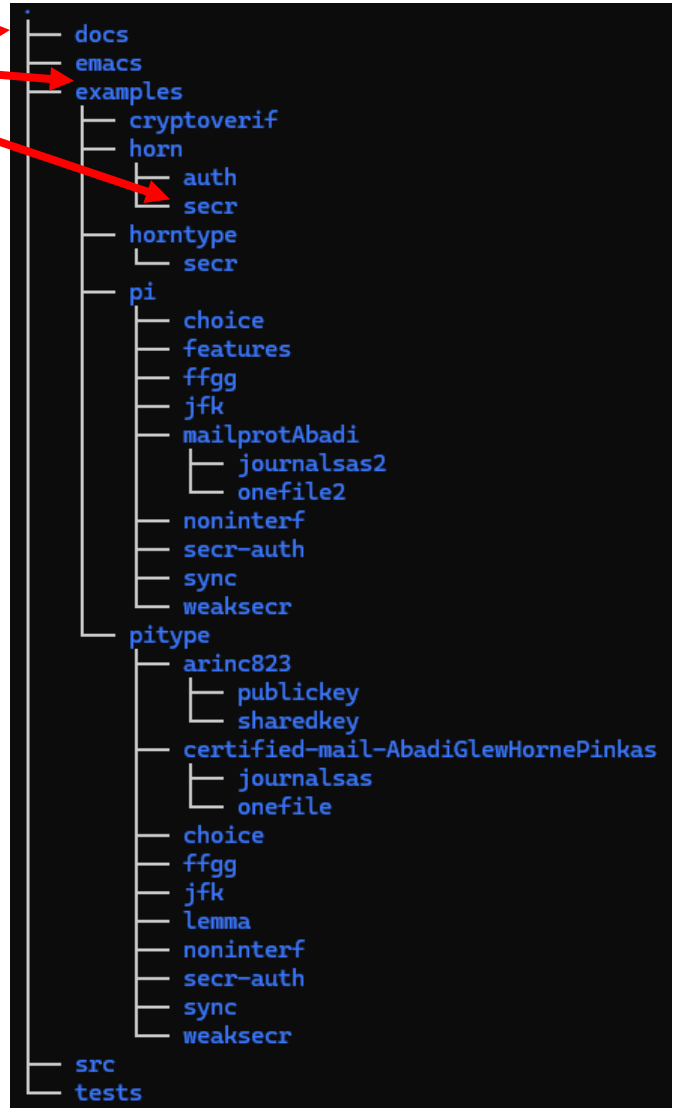
Path aliases

- `~, ~se317` - home directory



Example

CWD




```
cd docs
cd examples
cd horn/secr
```

```
fopen("file.txt", "w+")
```

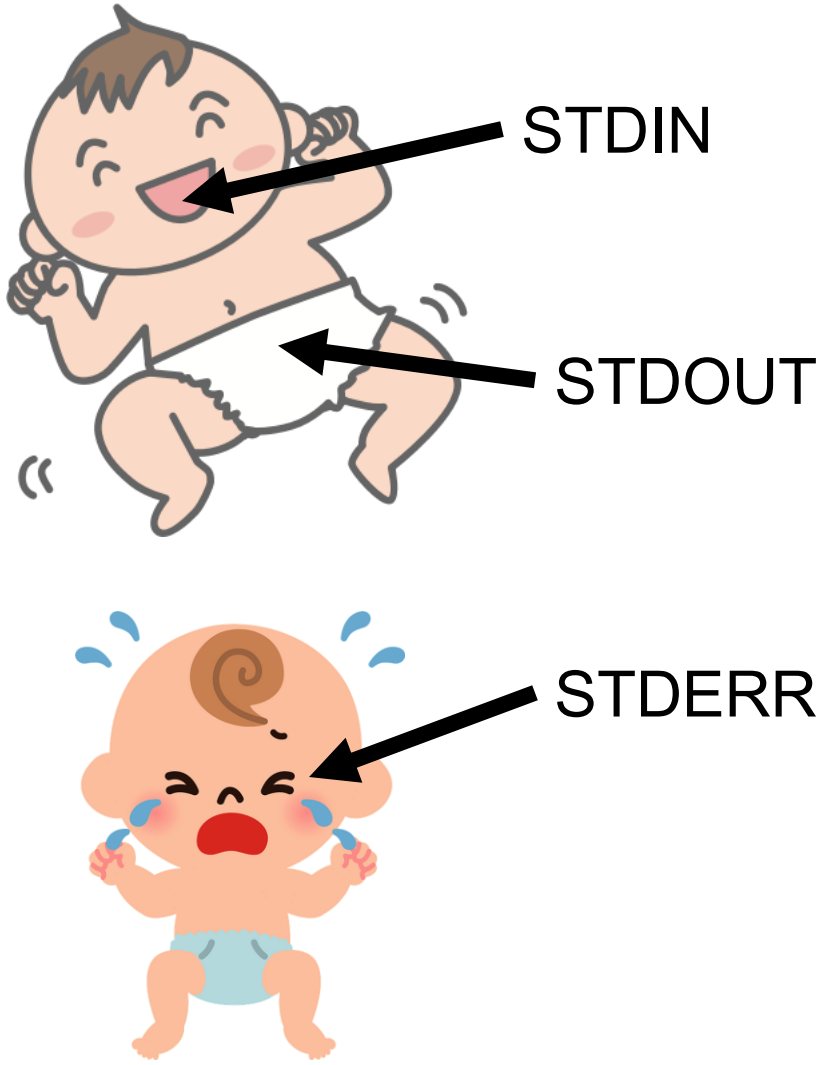
C API Standard Streams

- Three predefined streams are opened implicitly when the program is executed.

1. `FILE *stdin` – normal source of input, can be redirected
2. `FILE *stdout` – normal source of output, can be redirected too
3. `FILE *stderr` – diagnostics and errors

- `STDIN/STDOUT` enable composition in Unix
  - Recall: Use of pipe symbols connects `STDOUT` and `STDIN`
    - `find | grep | wc …`

Picturing it



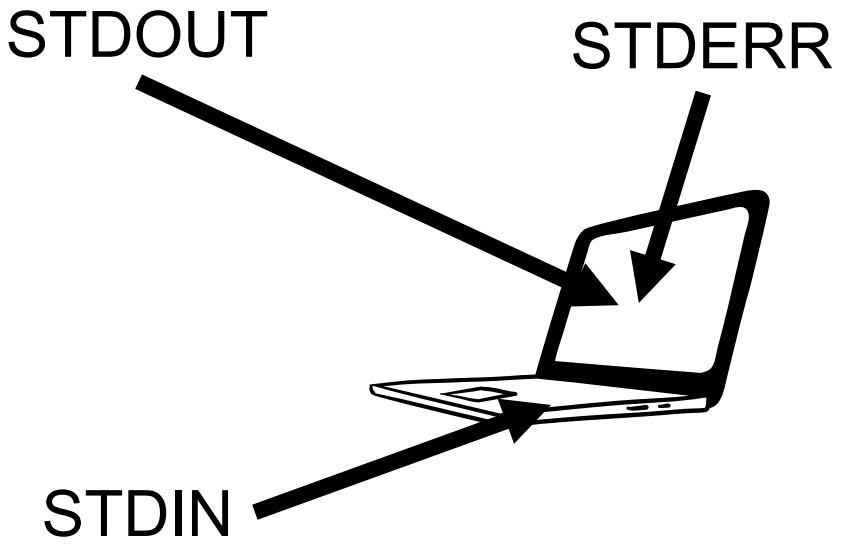


Image sources: https://openclipart.org/detail/300835/baby-2
https://creazilla.com/media/clipart/34014/baby-is-crying

C high level File API – stream ops

```
#include <stdio.h>
// character oriented
int fputc( int c, FILE *fp );   // rtn c or EOF on err
int fputs( const char *s, FILE *fp );   // rtn >0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );
```

```
// block oriented
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
```

```
// formatted
int fprintf(FILE *restrict stream, const char *restrict format, ...);
int fscanf(FILE *restrict stream, const char *restrict format, ... );
```

Example Stream Code

```
#include <stdio.h>
#include <string.h>

#define BUFLEN 256
FILE *outfile;
char mybuf[BUFLEN];

int storetofile(){
          char *instring;

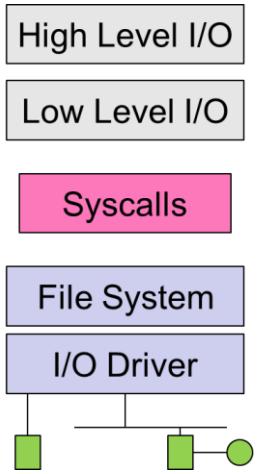
          outfile = fopen("/home/mjmay/Lecture5-Images/tokens", "w+");
          if (!outfile) {
                    return (-1); // Error!
          }
          while (1) {
                    instring = fgets(mybuf, BUFLEN, stdin); // catches overrun!

                    // check for error or end of file (^D)
                    if ( !instring || strlen(instring) == 0) break;

                    // write string to output file, exit on error
                    if (fputs(instring, outfile) < 0) break;
          }
          fclose(outfile); // Flushes from userspace
}
```

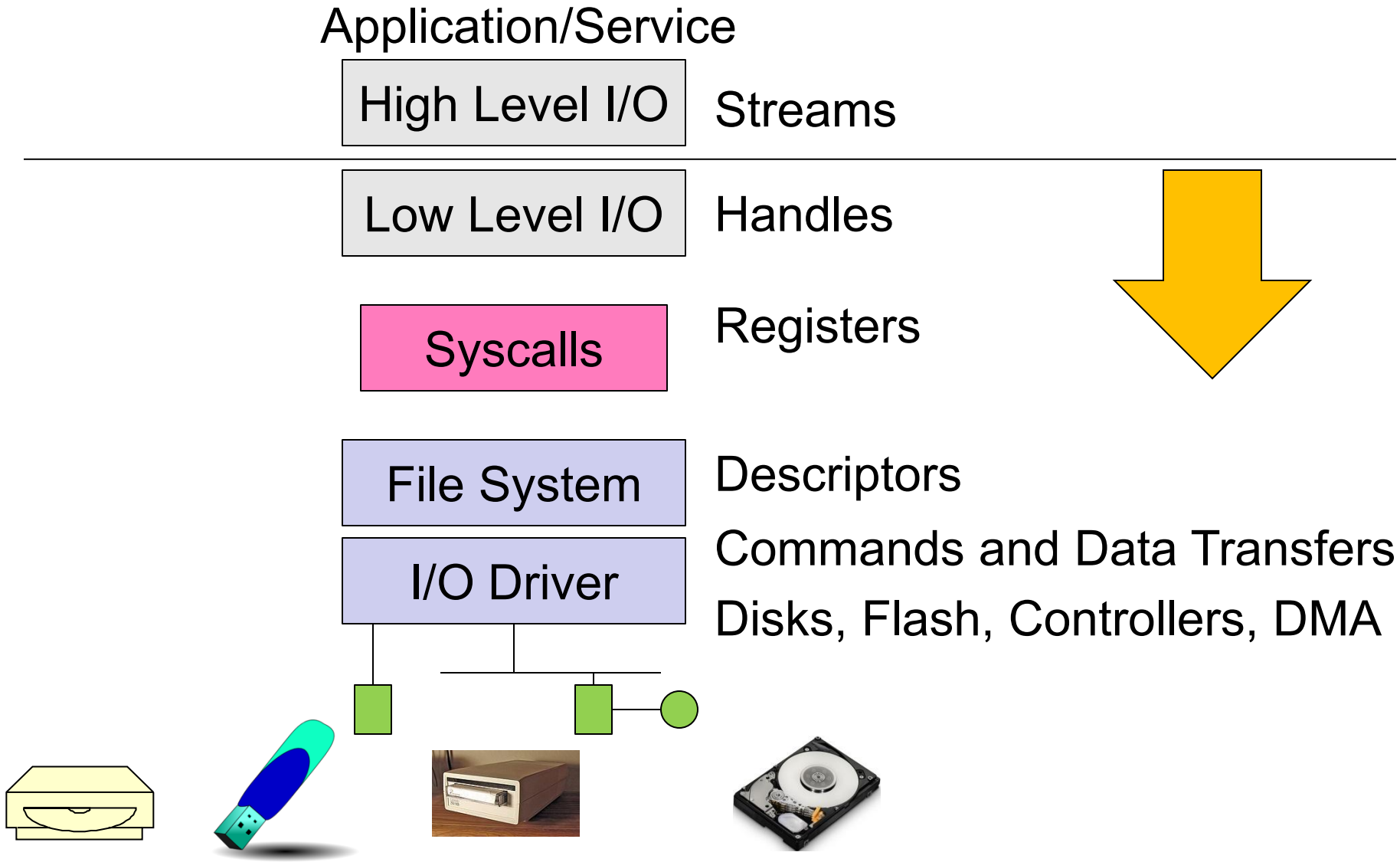
C Stream API positioning

```
int fseek(FILE *stream, long int offset, int whence);
long int ftell (FILE *stream);
void rewind (FILE *stream);
```



- Preserves high level abstraction of a uniform stream of objects
- Adds buffering for performance (don't forget to flush `fflush`)

What's below the surface?



C Low level I/O

- Operations on File Descriptors – as OS object representing the state of a file
  - User has a “handle” on the descriptor

http://www.gnu.org/software/libc/manual/html_node/Opening-and-Closing-Files.html

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int open (const char *filename, int flags [, mode_t mode])
int creat (const char *filename, mode_t mode)
int close (int filedes)
```

Flags: Bit Vector of:
- Access Mode (Rd, Wr, …)
- Open flags (Create,…)
- Operating modes (Appends,…)

mode: Bit Vector of Permission Bits
- User | Group | Other × R | W | X

C Low Level: standard descriptors

```
#include <unistd.h>
STDIN_FILENO -  macro has value 0
STDOUT_FILENO - macro has value 1
STDERR_FILENO - macro has value 2

int fileno (FILE *stream)
FILE * fdopen (int filedes, const char *opentype)
```

- Crossing levels: File descriptors vs. streams
- Don’t mix them!

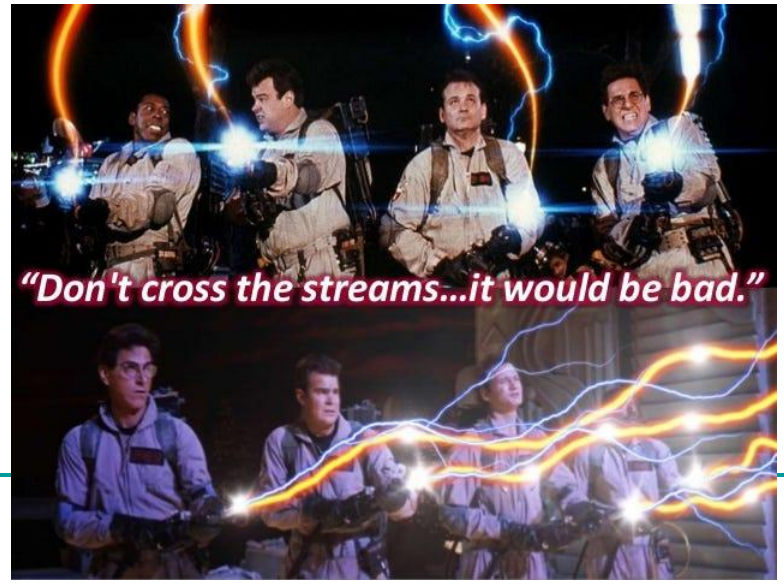


Image source: https://www.truthaboutthreats.com/p/never-cross-the-streams-it-would

C Low Level Operations

```
ssize_t read (int filedes, void *buffer, size_t maxsize)
 /*returns bytes read, 0 => EOF, -1 => error*/
ssize_t write (int filedes, const void *buffer, size_t size)
 /*returns bytes written*/

off_t lseek (int filedes, off_t offset, int whence)

int fsync (int fildes) /*- wait for i/o to finish*/
void sync (void) /*- wait for ALL to finish*/
```

- When write returns, data is on its way to disk and can be read, but it may not actually be permanent!

And lots more!

- TTYs versus files
- Memory mapped files
- File Locking
- Asynchronous I/O
- Generic I/O Control Operations
- Duplicating descriptors

```
int dup2 (int old, int new)
int dup (int old)
```

```
freopen(const char
*filename, const char
*mode, FILE *stream)
```

Example – lowio

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 1024

int main (int argc, char *argv[]) {
        char buf[BUFSIZE];
        ssize_t writelen = write(STDOUT_FILENO, "I am a process.\n", 16);

        ssize_t readlen = read (STDIN_FILENO, buf, BUFSIZE);

        ssize_t strlen = snprintf(buf, BUFSIZE, "Got %zd chars\n", readlen);

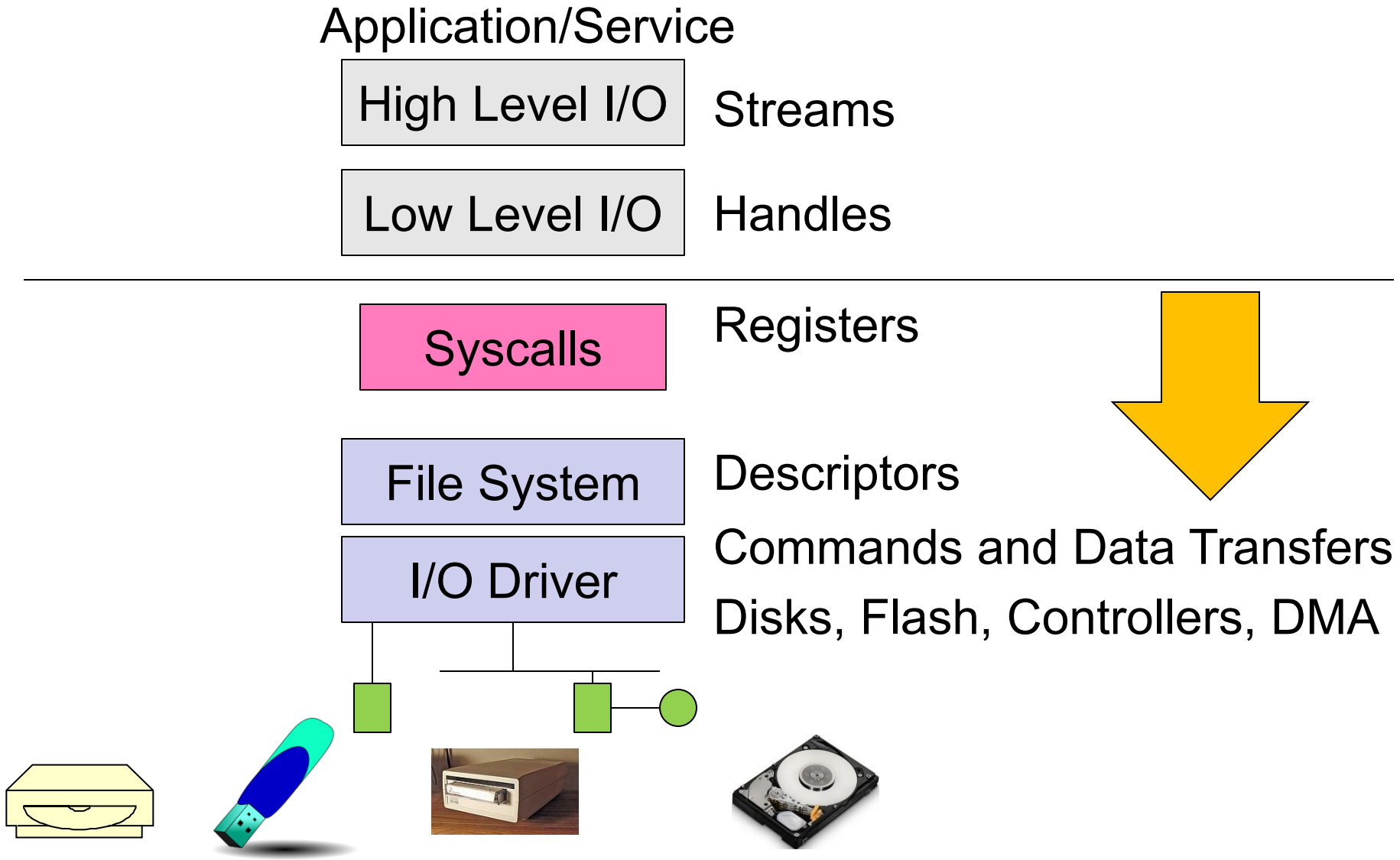
        writelen = strlen < BUFSIZE ? strlen : BUFSIZE;
        write (STDOUT_FILENO, buf, writelen);

        exit(0);
}
```

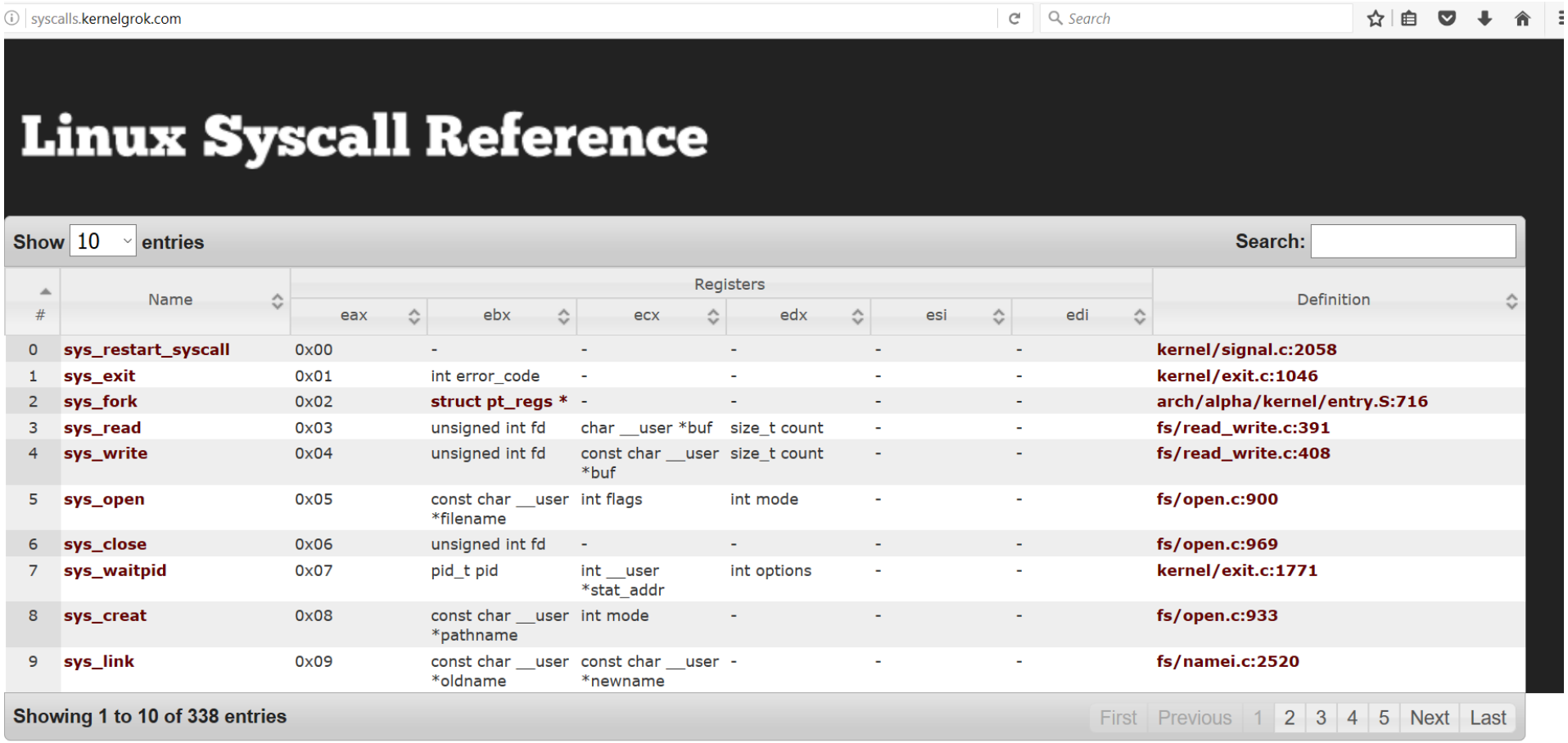
So Far

- Basic Support for I/O (drivers, etc.)
  - Files and Streams
  - Low Level
- I/O and Drivers
- Sockets and networks

What's below the surface?



Linux Syscalls



https://filippo.io/linux-syscall-table/

Linux Syscalls

Low level lib parameters are set up in registers and syscall instruction is issued

A type of synchronous exception that enters well-defined entry points into the kernel

Internal OS File Descriptor

- Internal Data Structure describing everything about the file
  - Where it resides
  - Its status
  - How to access it

lxr.free-electrons.com/source/include/linux/fs.h

```
875
876 struct file {
877         union {
878                 struct llist_node       fu_llist;
879                 struct rcu_head         fu_rcuhead;
880         } f_u;
881         struct path             f_path;
882         struct inode            *f_inode;       /* cached value */
883         const struct file_operations    *f_op;
884
885         /*
886          * Protects f_ep_links, f_flags.
887          * Must not be taken from IRQ context.
888          */
889         spinlock_t              f_lock;
890         atomic_long_t           f_count;
891         unsigned int            f_flags;
892         fmode_t                 f_mode;
893         struct mutex            f_pos_lock;
894         loff_t                  f_pos;
895         struct fown_struct      f_owner;
896         const struct cred       *f_cred;
897         struct file_ra_state    f_ra;
898
899         u64                     f_version;
900 #ifdef CONFIG_SECURITY
901         void                    *f_security;
902 #endif
903         /* needed for tty driver, and maybe others */
904         void                    *private_data;
905
906 #ifdef CONFIG_EPOLL
907         /* Used by fs/eventpoll.c to link all the hooks to this file */
908         struct list_head        f_ep_links;
909         struct list_head        f_tfile_llink;
910 #endif /* #ifdef CONFIG_EPOLL */
```

http://lxr.free-electrons.com/source/include/linux/fs.h

File System: from syscall to driver

```
460 ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
461 {
462         ssize_t ret;
463
464         if (!(file->f_mode & FMODE_READ))
465                 return -EBADF;
466         if (!(file->f_mode & FMODE_CAN_READ))
467                 return -EINVAL;
468         if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
469                 return -EFAULT;
470
471         ret = rw_verify_area(READ, file, pos, count);
472         if (!ret) {
473                 if (count > MAX_RW_COUNT)
474                         count =  MAX_RW_COUNT;
475                 ret = __vfs_read(file, buf, count, pos);
476                 if (ret > 0) {
477                         fsnotify_access(file);
478                         add_rchar(current, ret);
479                 }
480                 inc_syscr(current);
481         }
482
483         return ret;
484 }
```

http://lxr.free-electrons.com/source/fs/read_write.c

File System: from syscall to driver

```
448 ssize_t __vfs_read(struct file *file, char __user *buf, size_t count,
449                    loff_t *pos)
450 {
451         if (file->f_op->read)
452                 return file->f_op->read(file, buf, count, pos);
453         else if (file->f_op->read_iter)
454                 return new_sync_read(file, buf, count, pos);
455         else
456                 return -EINVAL;
457 }
```

http://lxr.free-electrons.com/source/fs/read_write.c

Low Level Driver



http://lxr.free-electrons.com/source/include/linux/fs.h#L1679

```
1679 struct file_operations {
1680         struct module *owner;
1681         loff_t (*llseek) (struct file *, loff_t, int);
1682         ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
1683         ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
1684         ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
1685         ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
1686         int (*iterate) (struct file *, struct dir_context *);
1687         int (*iterate_shared) (struct file *, struct dir_context *);
1688         unsigned int (*poll) (struct file *, struct poll_table_struct *);
1689         long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
1690         long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
1691         int (*mmap) (struct file *, struct vm_area_struct *);
1692         int (*open) (struct inode *, struct file *);
1693         int (*flush) (struct file *, fl_owner_t id);
1694         int (*release) (struct inode *, struct file *);
1695         int (*fsync) (struct file *, loff_t, loff_t, int datasync);
1696         int (*aio_fsync) (struct kiocb *, int datasync);
1697         int (*fasync) (int, struct file *, int);
1698         int (*lock) (struct file *, int, struct file_lock *);
1699         ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
1700         unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
1701         int (*check_flags)(int);
1702         int (*flock) (struct file *, int, struct file_lock *);
1703         ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size
1704         ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_
1705         int (*setlease)(struct file *, long, struct file_lock **, void **);
1706         long (*fallocate)(struct file *file, int mode, loff_t offset,
1707                           loff_t len);
```

Device Drivers



Device Driver: Device-specific code in the kernel that interacts directly with the device hardware

- Supports a standard, internal interface
- Same kernel I/O system can interact easily with different device drivers
- Special device-specific configuration supported with the `ioctl()` system call

Device Drivers typically divided into two pieces:

- Top half: accessed in call path from system calls
  - implements a set of standard, cross-device calls like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
  - This is the kernel's interface to the device driver
  - Top half will *start* I/O to device, may put thread to sleep until finished
- Bottom half: run as interrupt routine
  - Gets input or transfers next block of output
  - May wake sleeping threads if I/O now complete

Low Level Driver



Photo by Lewis J Goetz on Unsplash

Associated with particular hardware device

Registers / Unregisters itself with the kernel

Handler functions for each of the file operations

Life Cycle of An I/O Request

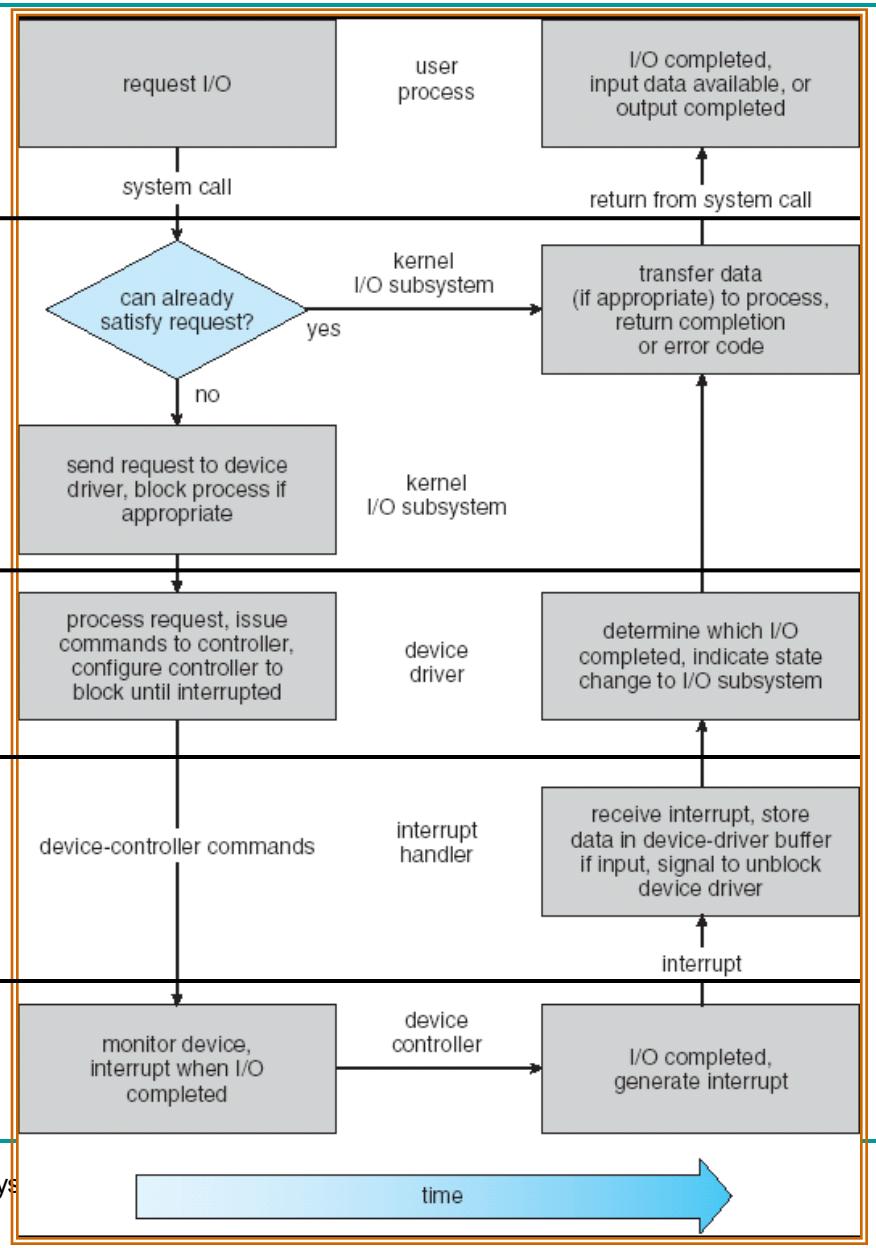
User Program

Kernel I/O Subsystem

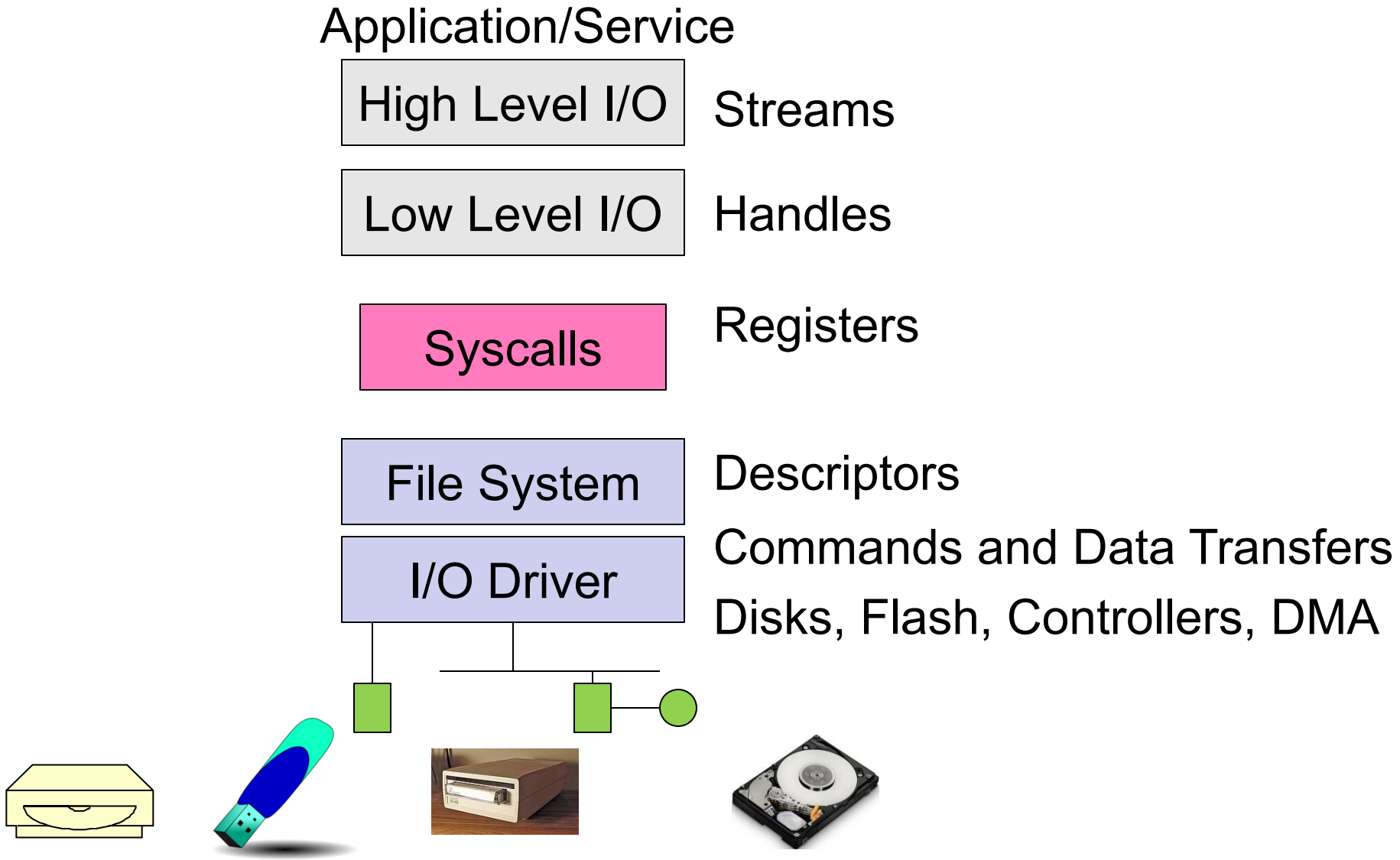
Device Driver top half

Device Driver bottom half

Device Hardware



So what happens when you `fgetc`?



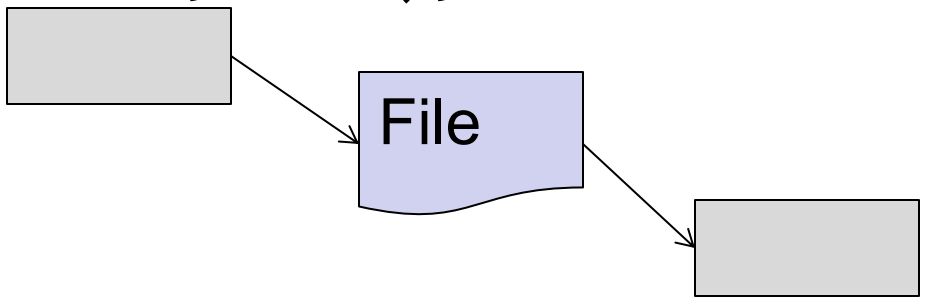
So Far

- Basic Support for I/O (drivers, etc.)
  - Files and Streams
  - Low Level
- I/O and Drivers
- Sockets and networks

Communication Between Processes

- Can we view files as communication channels?

```
write(wfd, wbuf, wlen);
```

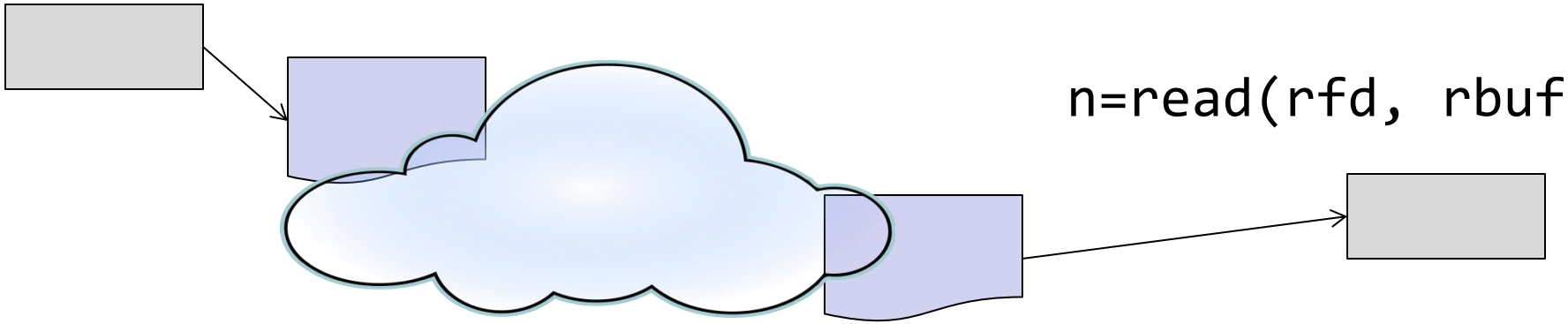



```
n=read(rfd, rbuf, rmax);
```

- Producer and Consumer of a file may be different processes
  - May be separated in time (or not)
- However, what if data written once and consumed once?
  - Would be more like a queue, but still look like File I/O!

Communication across the world

```
write(wfd, wbuf, wlen);
```



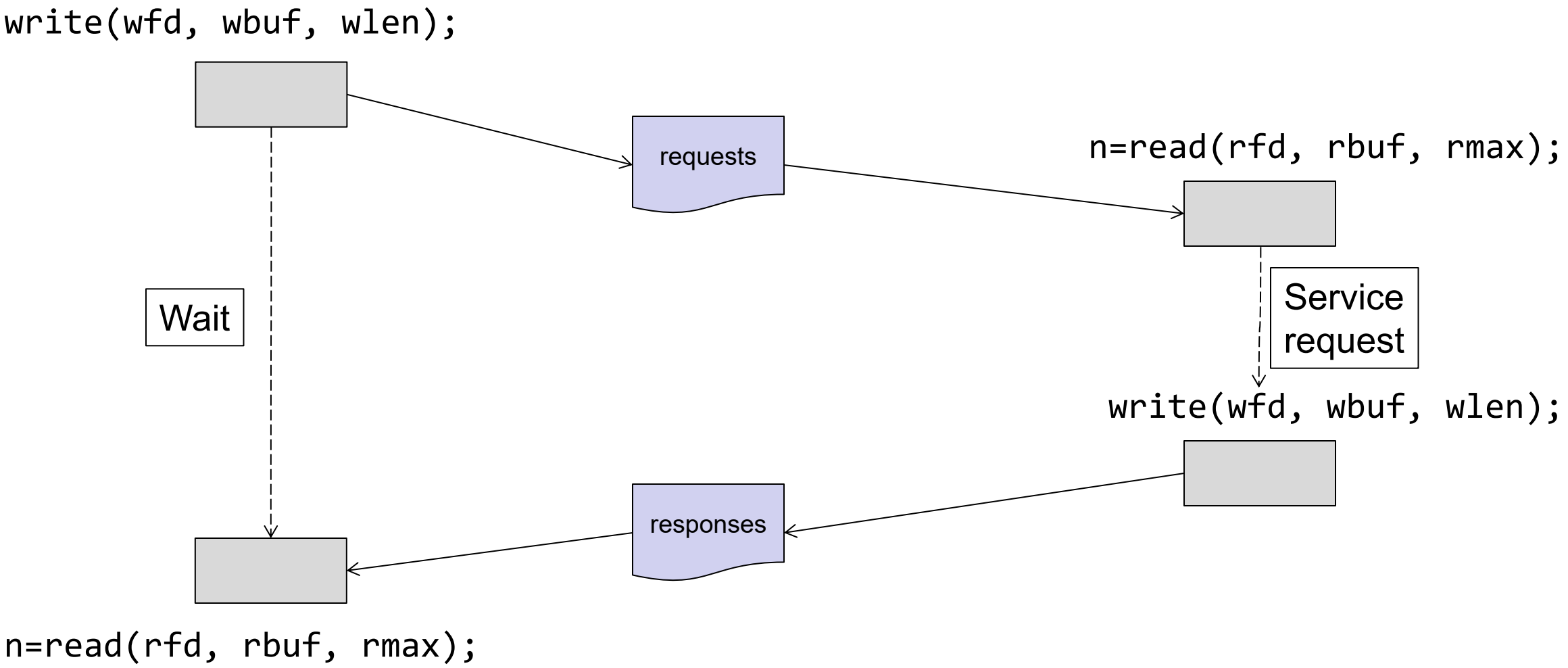
```
n=read(rfd, rbuf, rmax);
```

- Connected queues over the internet
  - What's the analog of open?
  - What is the namespace?
  - How are they connected in time?

Request Response Protocol

Client (issues requests)

Server (performs operations)

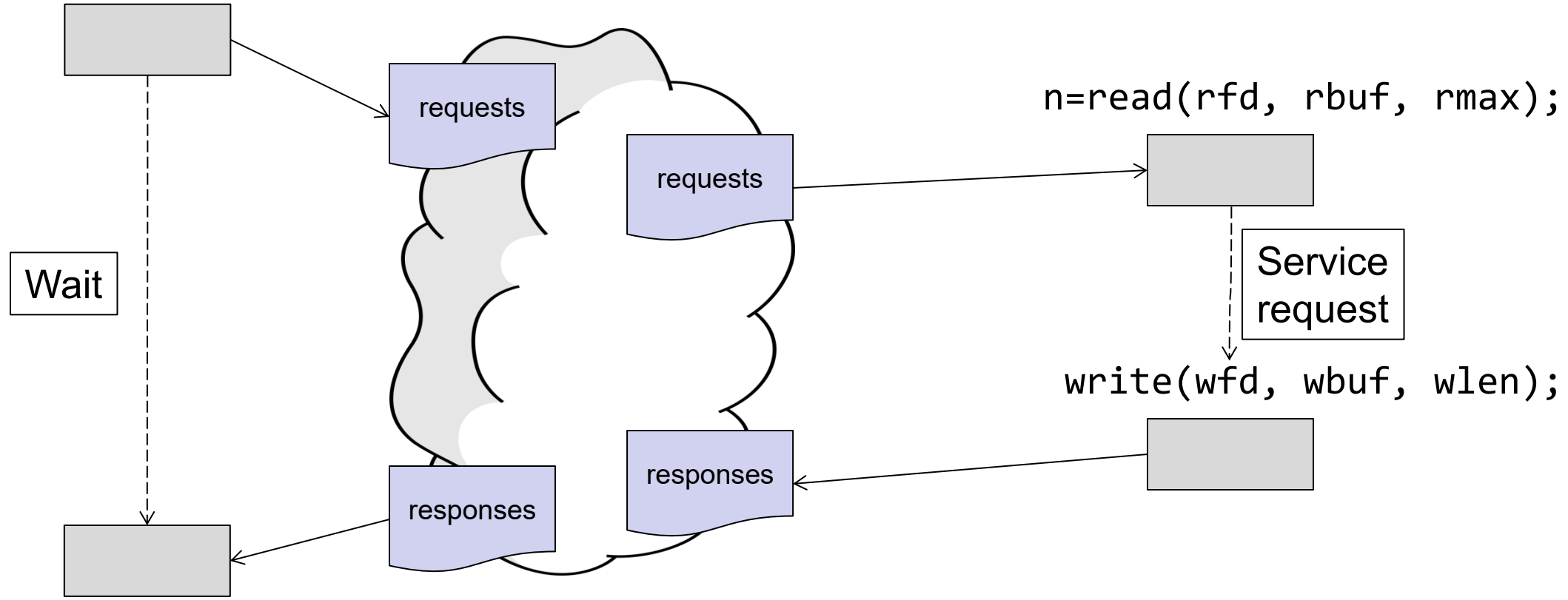


Request Response Protocol

Client (issues requests)

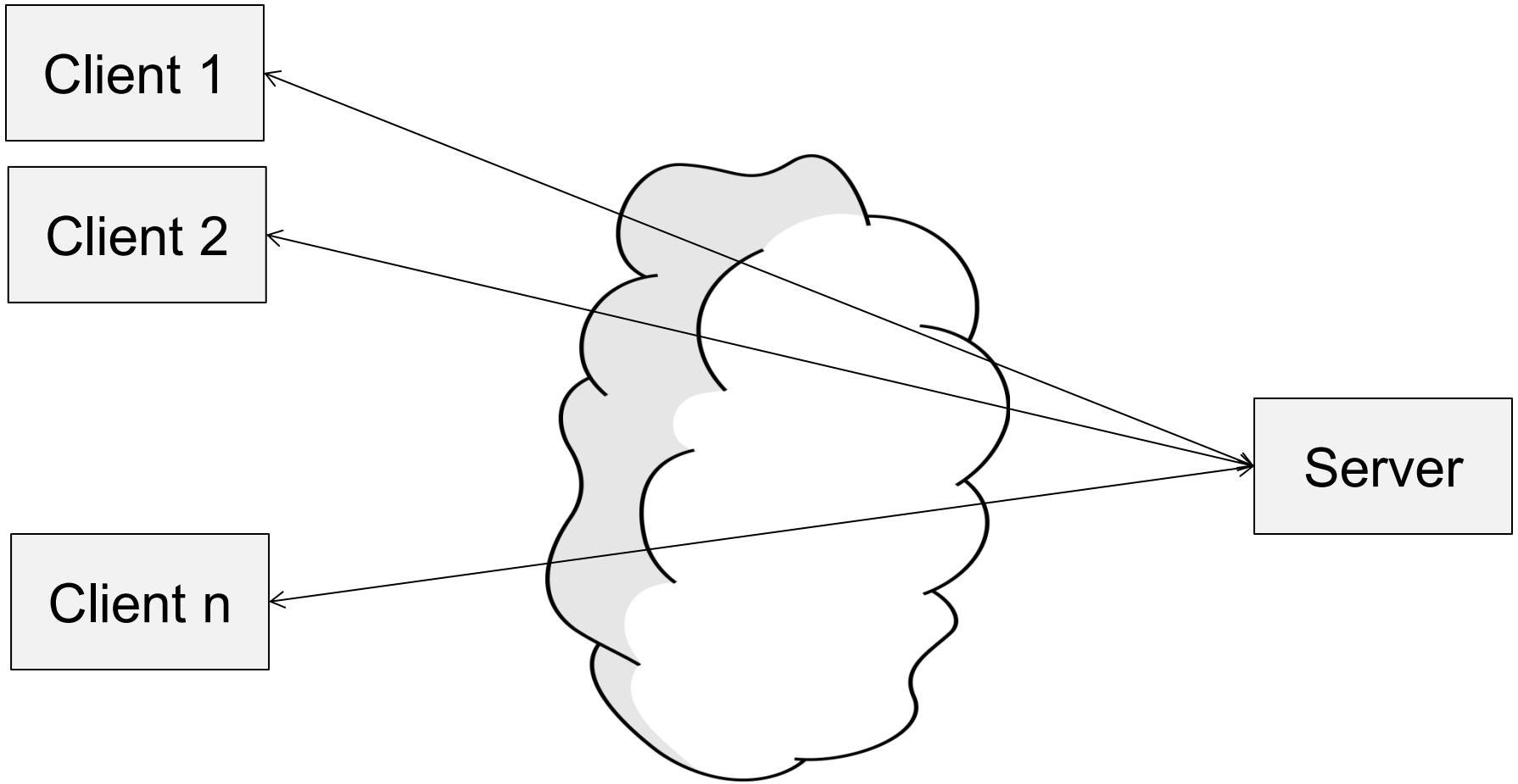
Server (performs operations)

```
write(wfd, wbuf, wlen);
```




```
n=read(rfd, rbuf, rmax);
```

Client-Server Models



- File servers, Web servers, FTP servers, Databases
- Many clients access a common server

Sockets



Socket: an abstraction of a network I/O queue

- Mechanism for inter-process communication
- Embodies one side of a communication channel
  - Same interface regardless of location of other end
  - Could be local machine ("UNIX socket") or remote machine ("network socket")
- First introduced in 4.2 BSD UNIX: big innovation at time
  - Now most operating systems provide some notion of socket

Data transfer like files

- Read / Write against a descriptor

Over any kind of network

- Local to a machine
- Over the internet (TCP/IP, UDP/IP)
- OSI, Appletalk, SNA, IPX, SIP, NS, …

Socket Creation and Connection

File systems

- Provide a collection of permanent objects in structured name space
- Processes open, read/write/close them
- Files exist independent of the processes

Sockets

- Provide a means for processes to communicate (transfer data) to other processes.
  - Creation and connection is more complex
  - Form 2-way pipes between processes, possibly worlds away

Conclusion

- Basic Support for I/O (drivers, etc.)
  - Files and Streams
  - Low Level
- I/O and Drivers
- Sockets and networks