CS162
Operating Systems and Systems Programming
Lecture 8

Introduction to I/O, Sockets, Networking

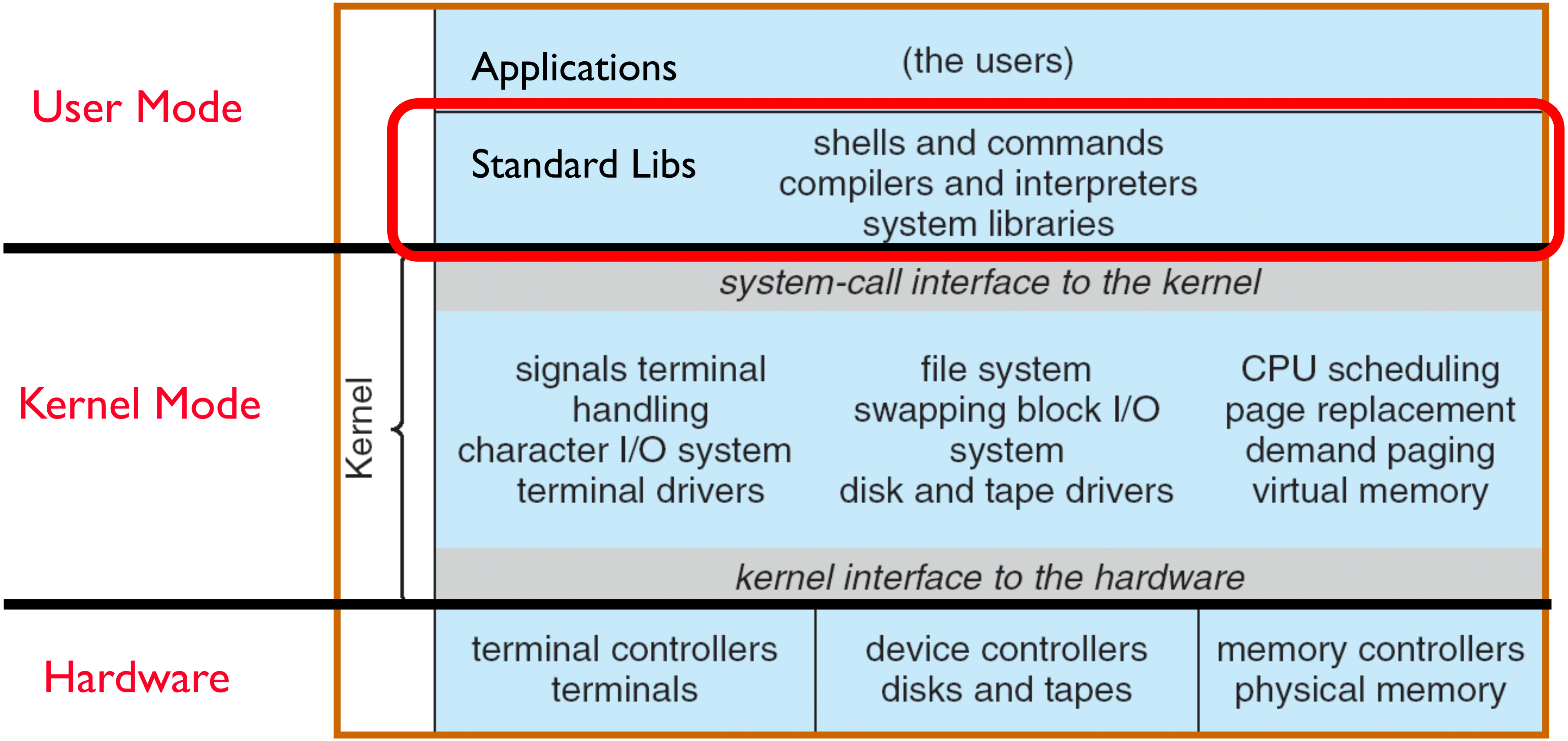
February 18th, 2020

Prof. John Kubiatowicz

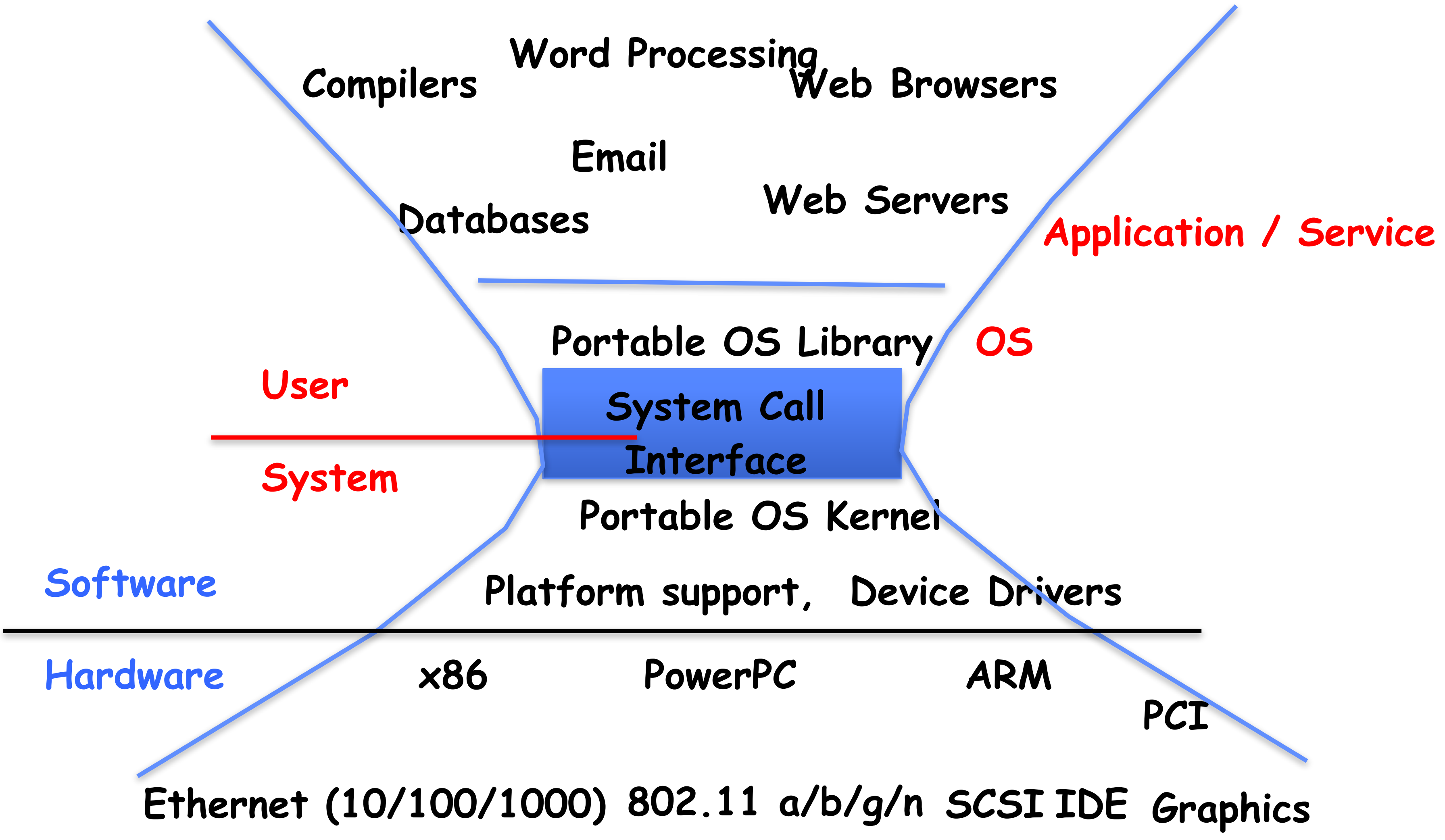
http://cs162.eecs.Berkeley.edu

Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes. *When slides are obtained from other sources, a a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.*

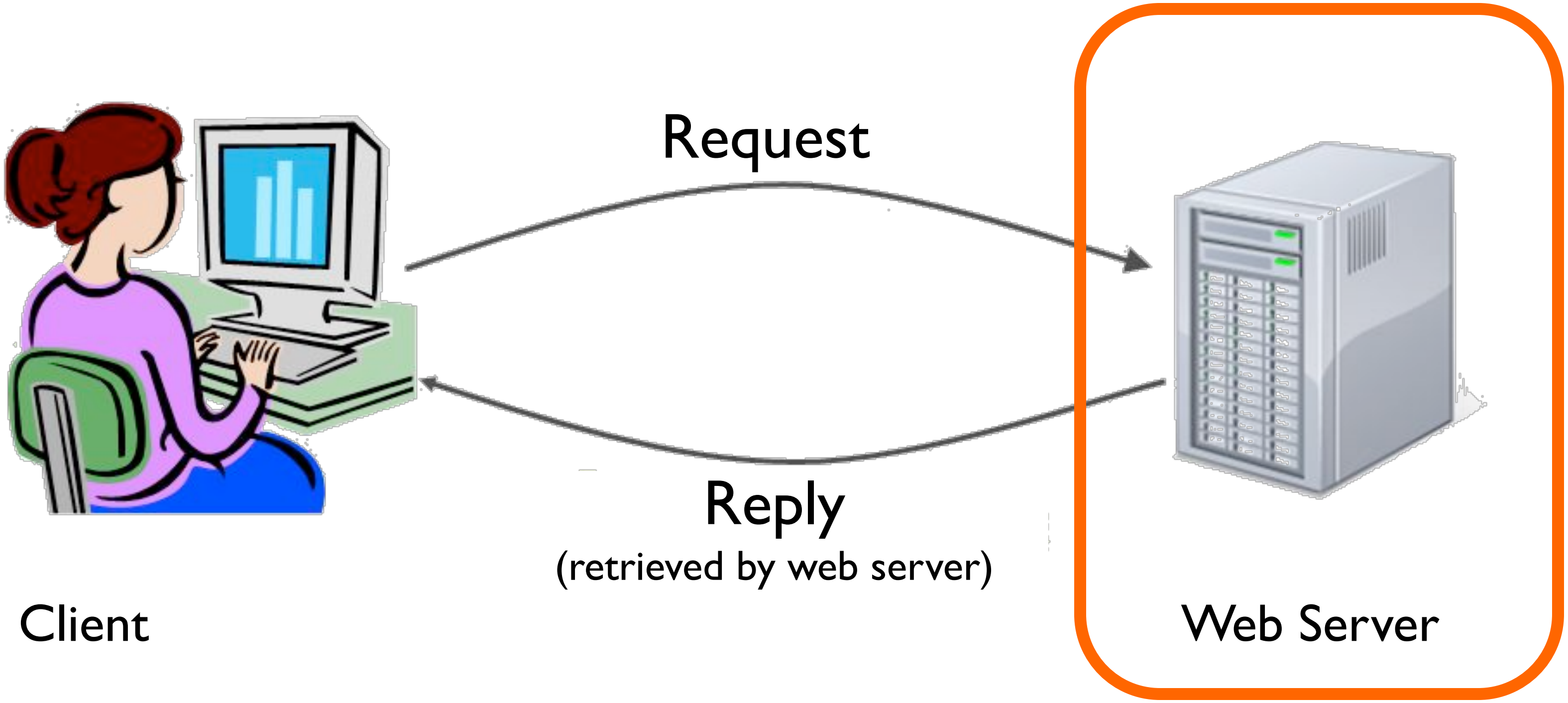
Recall: UNIX System Structure



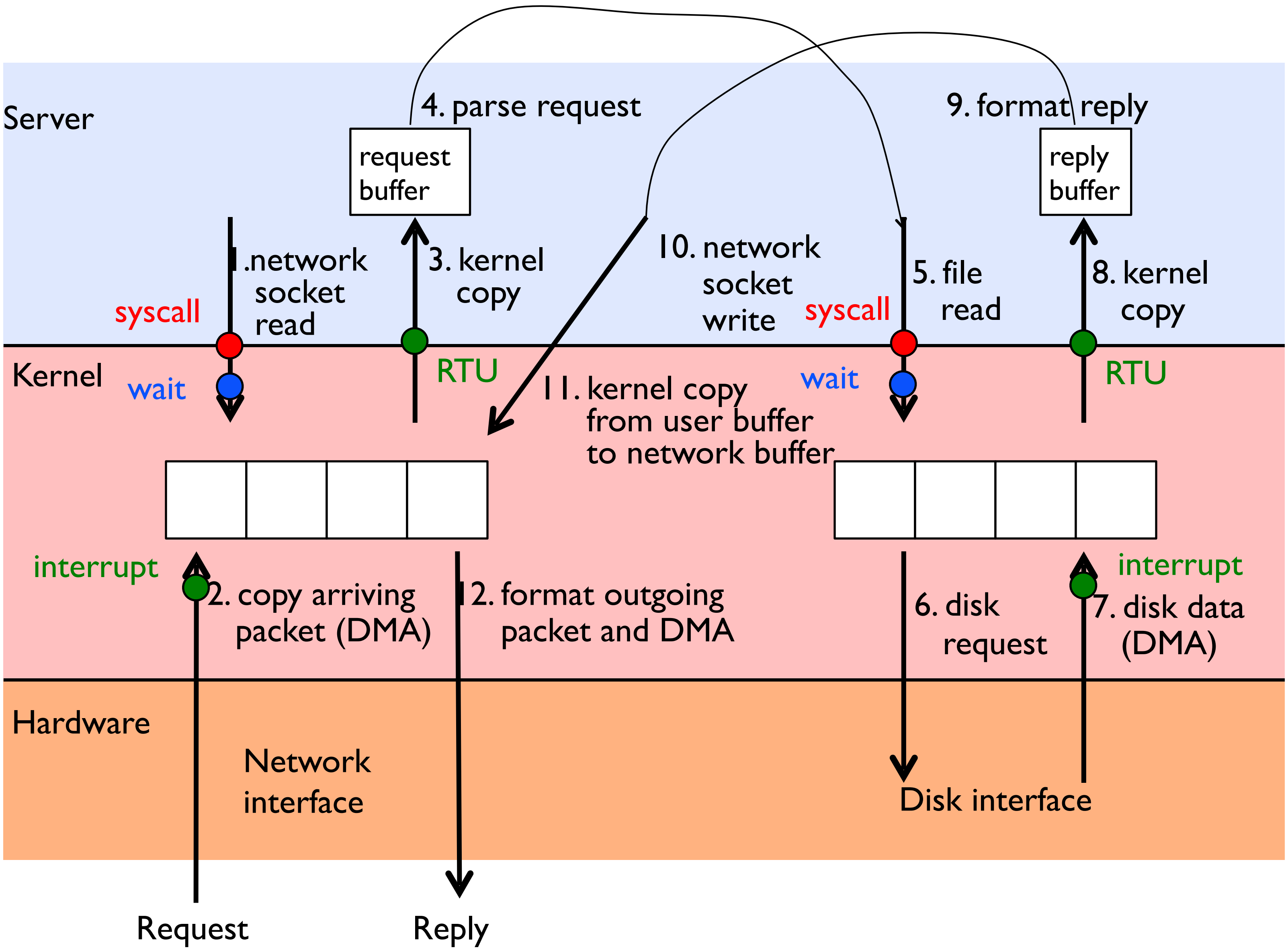
Recall: A Kind of Narrow Waist



Recall: web server



Recall: web server



POSIX I/O: Everything is a "File"

- Identical interface for:
  - Devices (terminals, printers, etc.)
  - Regular files on disk
  - Networking (sockets)
  - Local interprocess communication (pipes, sockets)
- Based on open(), read(), write(), and close()
- Allows simple composition of programs
  - » `find | grep | wc …`

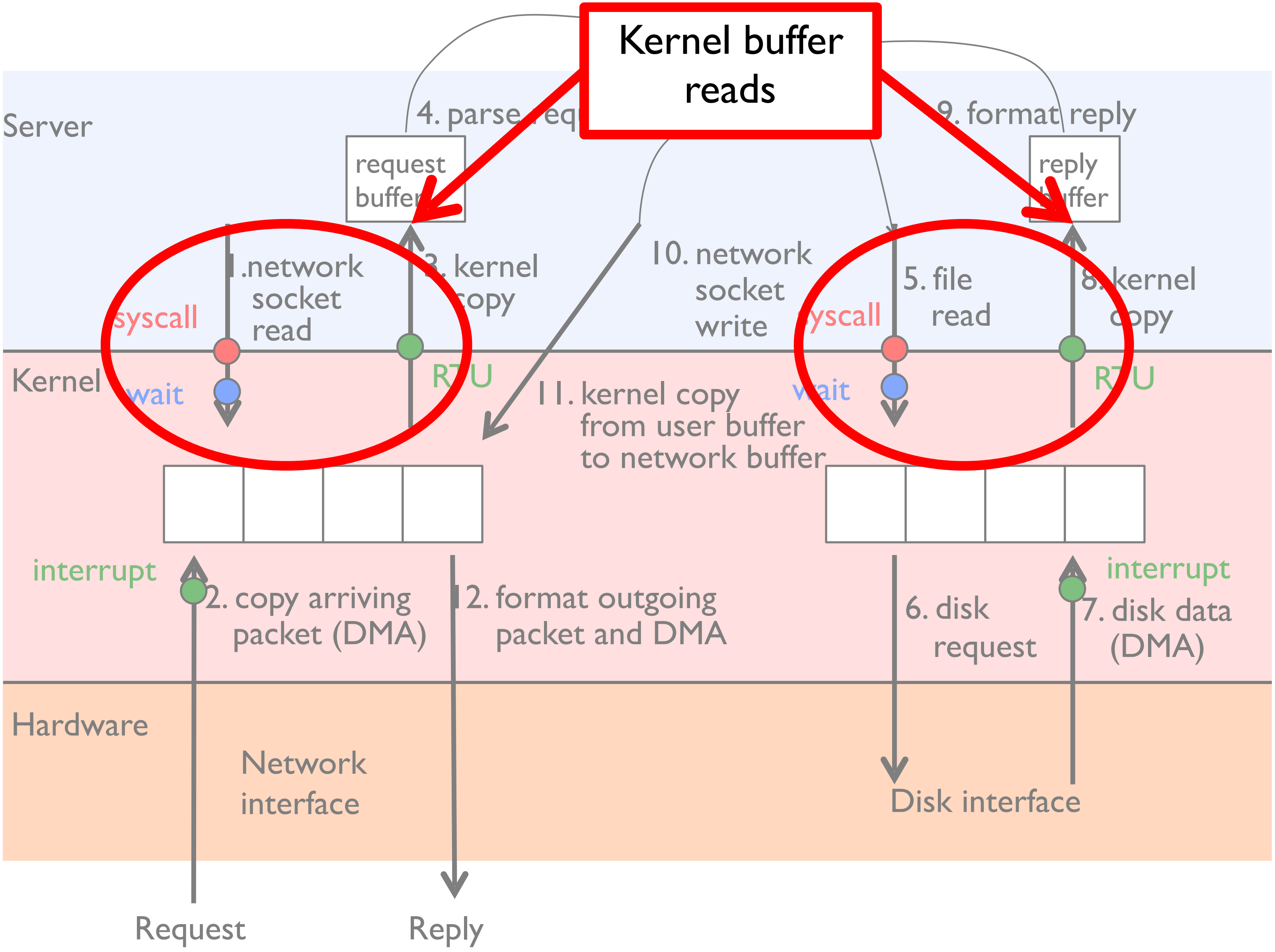
POSIX I/O Design Patterns

- Open before use
  - Access control check, setup happens here
- Byte-oriented
  - Least common denominator
  - OS responsible for hiding the fact that real devices may not work this way (e.g. hard drive stores data in blocks)
- Explicit close

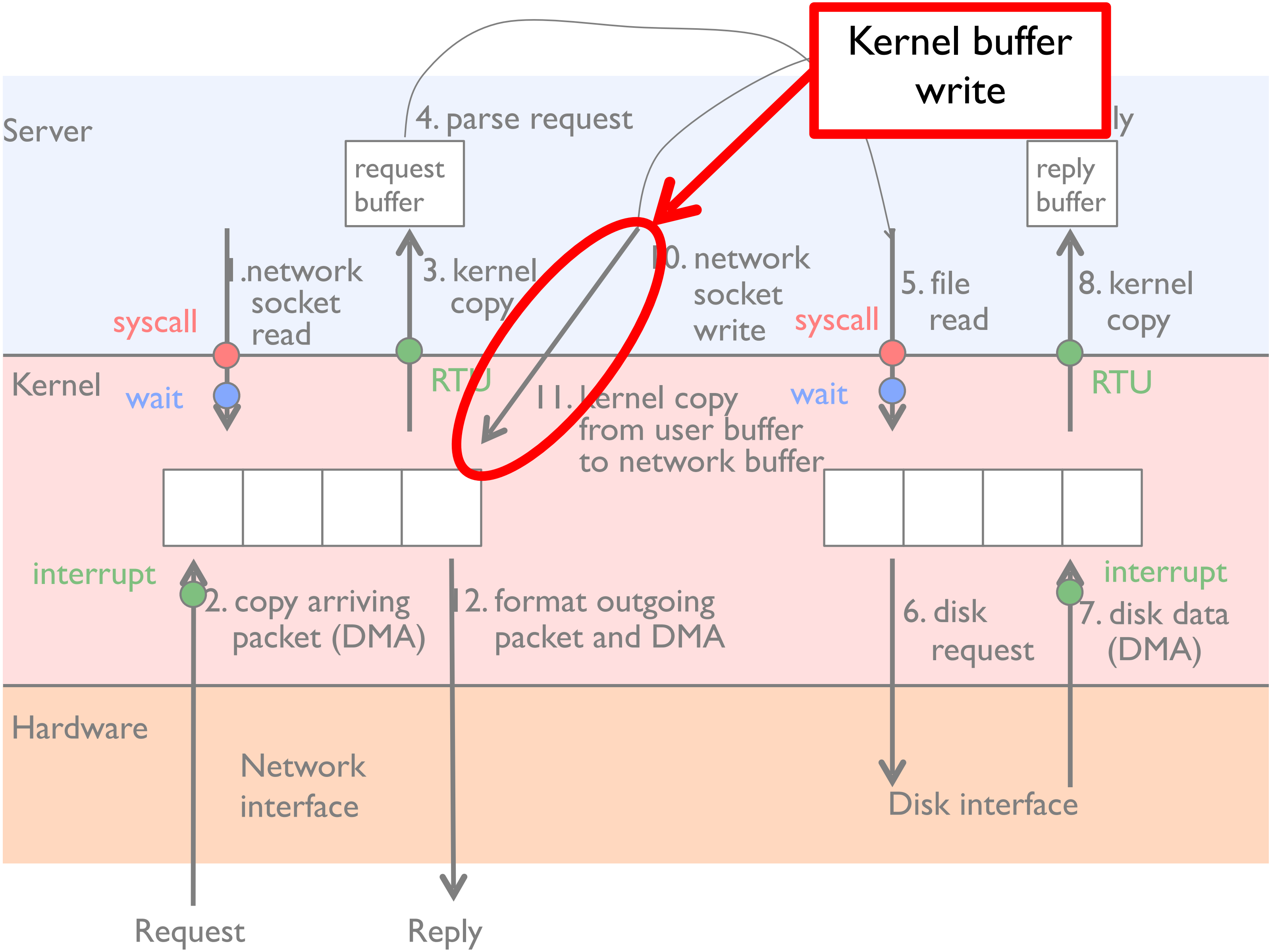
POSIX I/O: Kernel Buffering

- Reads are buffered
  - Part of making everything byte-oriented
  - Process is blocked while waiting for device
  - Let other processes run while gathering result
- Writes are buffered
  - Complete in background (more later on)
  - Return to user when data is "handed off" to kernel

Putting it together: web server



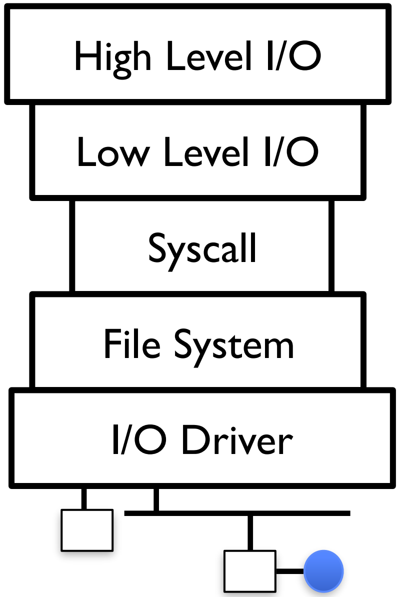
Putting it together: web server



Kernel buffer write

I/O & Storage Layers

Application / Service



High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

streams

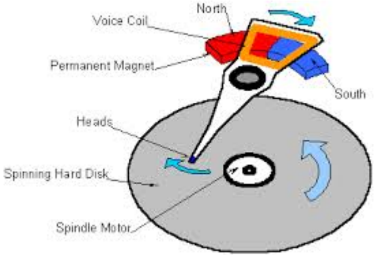
handles

registers

descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA



The File System Abstraction

- High-level idea
  - Files live in hierarchical namespace of filenames
- File
  - Named collection of data in a file system
  - POSIX File data: sequence of bytes
    - » Text, binary, linearized objects, …
  - File Metadata: information about the file
    - » Size, Modification Time, Owner, Security info
    - » Basis for access control
- Directory
  - "Folder" containing files & Directories
  - Hierachical (graphical) naming
    - » Path through the directory graph
    - » Uniquely identifies a file or directory
      - `/home/ff/cs162/public_html/fa18/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	Descriptions
r	rb	Open existing file for reading
w	wb	Open for writing; created if does not exist
a	ab	Open for appending; created if does not exist
r+	rb+	Open existing file for reading & writing.
w+	wb+	Open for reading & writing; truncated to zero if exists, create otherwise
a+	ab+	Open for reading & writing. Created if does not exist. Read from beginning, write as append

Don't forget to flush

Connecting Processes, Filesystem, and Users

- Process has a ‘current working directory’
- Absolute Paths
  - `/home/ff/cs162`
- Relative paths
  - `index.html`, `./index.html` - current WD
  - `../index.html` - parent of current WD
  - `~`, `~cs162` - home directory

C API Standard Streams – `stdio.h`

- Three predefined streams are opened implicitly when a program is executed
  - `FILE *stdin` – normal source of input, can be redirected
  - `FILE *stdout` – normal source of output, can be redirected
  - `FILE *stderr` – diagnostics and errors, can be redirected

- **`STDIN`** / **`STDOUT`** enable composition in Unix
- All can be redirected (for instance, using "pipe" symbol: '|'):
  - `cat hello.txt | grep "World!"`
    - » `Cat's` **`stdout`** `goes to grep's` **`stdin`**!

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, ...);
```

C Streams: char by char I/O

```
#include <stdio.h>

int main(void) {
  FILE* input = fopen(“input.txt”, “r”);
  FILE* output = fopen(“output.txt”, “w”);
  int c;

  c = fgetc(input);
  while (c != EOF) {
    fputc(output, c);
    c = fgetc(input);
  }
  fclose(input);
  fclose(output);
}
```

What if we wanted block by block I/O?

```
#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, ...);
```

stdio Block-by-Block I/O

```
#include <stdio.h>
#define BUFFER_SIZE 1024
int main(void) {
  FILE* input = fopen("input.txt", "r");
  FILE* output = fopen("output.txt", "w");
  char buffer[BUFFER_SIZE];
  size_t length;
  length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
  while (length > 0) {
    fwrite(buffer, length, sizeof(char), output);
    length = fread(buffer, BUFFER_SIZE, sizeof(char),
input);
  }
  fclose(input);
  fclose(output);
}
```

Aside: Systems Programming

- Systems programmers are paranoid
- We should really be writing things like:

```
FILE* input = fopen(“input.txt”, “r”);
if (input == NULL) {
  // Prints our string and error msg.
  perror(“Failed to open input file”)
}
```

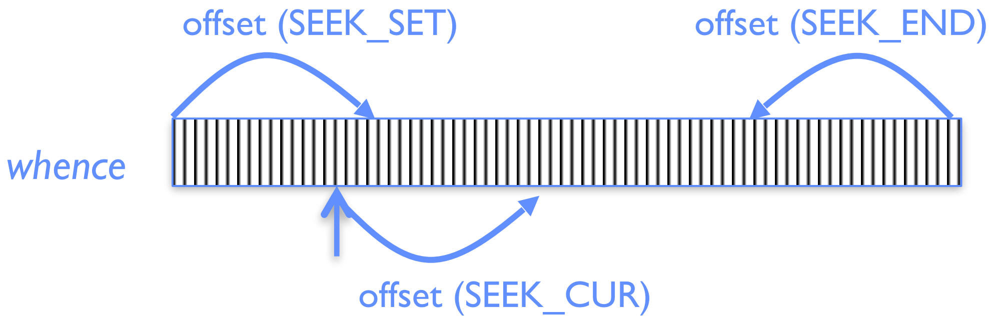
- Be thorough about checking return values
  – Want failures to be systematically caught and dealt with

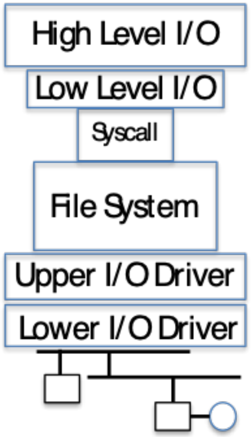
C Stream API: Positioning

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

long int ftell (FILE *stream)

void rewind (FILE *stream)
```

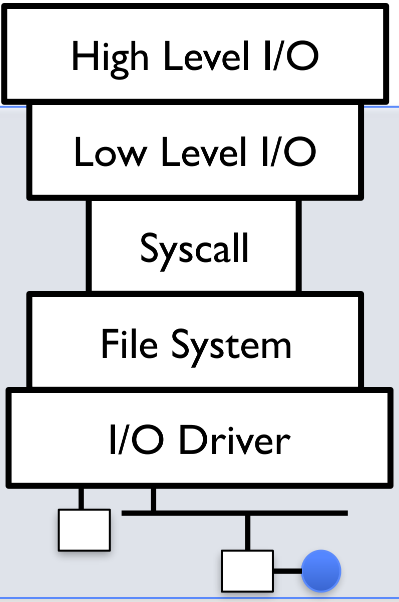




- Preserves high level abstraction of a uniform stream of objects

What’s below the surface ??

Application / Service



High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

streams

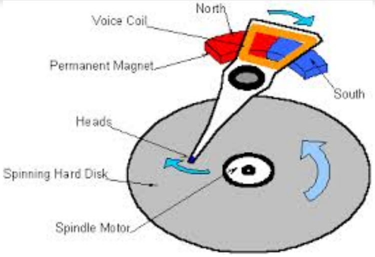
handles

registers

descriptors

commands and Data Transfers

disks, flash, controllers, DMA



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

```
#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)
```

Bit vector of:

- Access modes (Rd, Wr, …)
- Open Flags (Create, …)
- Operating modes (Appends, …)

Bit vector of Permission Bits:

- User|Group|Other X R|W|X

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

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!

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!

A little example: lowio.c

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

int main() {
  char buf[1000];
  int     fd = open("lowio.c", O_RDONLY, S_IRUSR | S_IWUSR);
  ssize_t rd = read(fd, buf, sizeof(buf));
  int    err = close(fd);
  ssize_t wr = write(STDOUT_FILENO, buf, rd);
}
```

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)
```

Another: lowio-std.c

```
#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);
}
```

Low-Level I/O: Example

```
#include <fcntl.h>
#include <unistd.h>

#define BUFFER_SIZE 1024

int main(void) {
  int input_fd = open(“input.txt”, O_RDONLY);
  int output_fd = open(“output.txt”, O_WRONLY);
  char buffer[BUFFER_SIZE];
  ssize_t length;
  length = read(input_fd, buffer, BUFFER_SIZE);
  while (length > 0) {
    write(output_fd, buffer, length);
    length = read(input_fd, buffer, BUFFER_SIZE);
  }
  close(input_fd);
  close(output_fd);
}
```

Streams vs. File Descriptors

- Streams are buffered in user memory:

```
        printf("Beginning of line ");
        sleep(10); // sleep for 10 seconds
        printf("and end of line\n");
```

⇒ Prints out everything at once

- Operations on file descriptors are visible immediately

```
        write(STDOUT_FILENO, "Beginning of line ",
18);
        sleep(10);
        write("and end of line \n", 16);
```

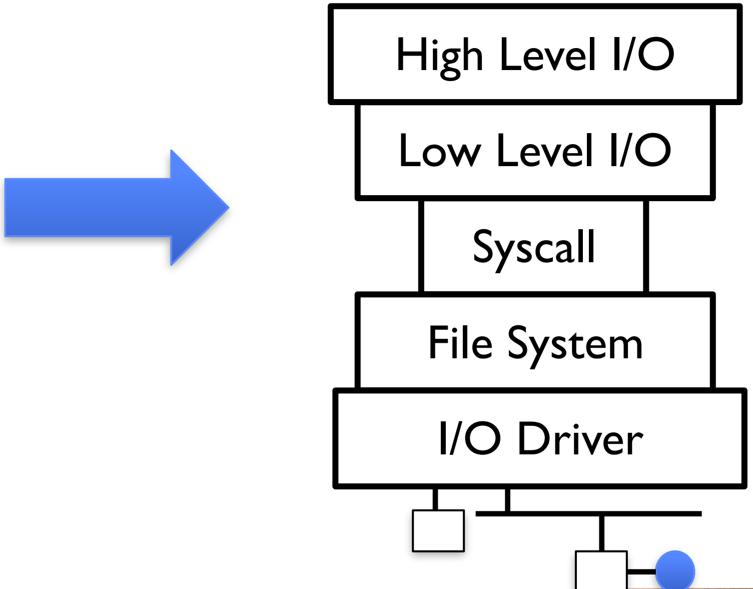
⇒ Outputs "Beginning of line" 10 seconds earlier

Summary: Key Unix I/O Design Concepts

- Uniformity – everything is a file
  - 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
- Kernel buffered reads
  - Streaming and block devices looks the same, read blocks yielding processor to other task
- Kernel buffered writes
  - Completion of out-going transfer decoupled from the application, allowing it to continue
- Explicit close

What’s below the surface ??

Application / Service



High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

streams

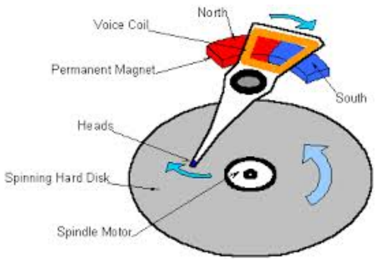
handles

registers

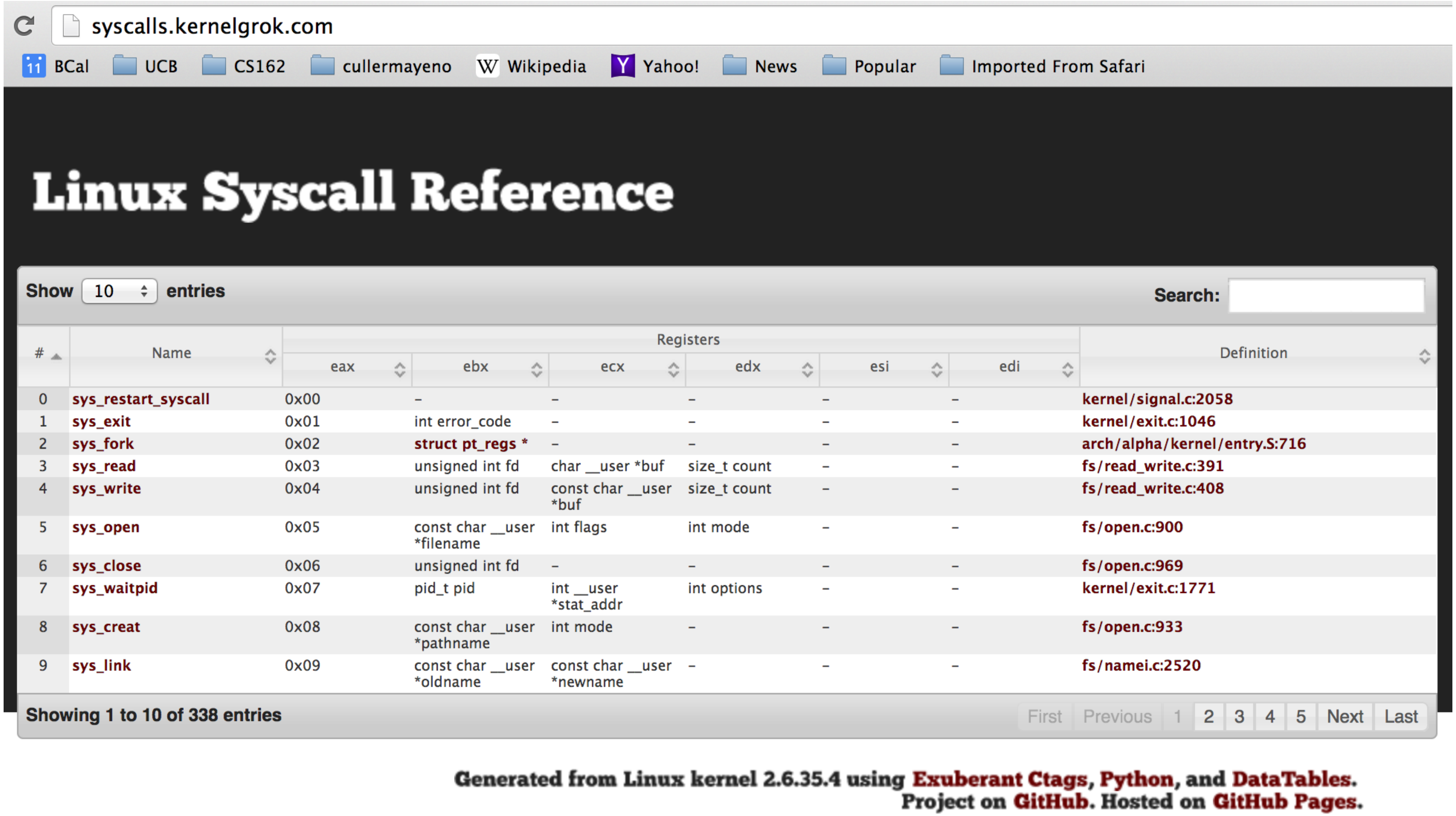
descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA



Recall: SYSCALL



Linux Syscall Reference

#	Name	Registers						Definition
		eax	ebx	ecx	edx	esi	edi	
0	sys_restart_syscall	0x00	-	-	-	-	-	kernel/signal.c:2058
1	sys_exit	0x01	int error_code	-	-	-	-	kernel/exit.c:1046
2	sys_fork	0x02	struct pt_regs *	-	-	-	-	arch/alpha/kernel/entry.S:716
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count	-	-	fs/read_write.c:391
4	sys_write	0x04	unsigned int fd	const char __user *buf	size_t count	-	-	fs/read_write.c:408
5	sys_open	0x05	const char __user *filename	int flags	int mode	-	-	fs/open.c:900
6	sys_close	0x06	unsigned int fd	-	-	-	-	fs/open.c:969
7	sys_waitpid	0x07	pid_t pid	int __user *stat_addr	int options	-	-	kernel/exit.c:1771
8	sys_creat	0x08	const char __user *pathname	int mode	-	-	-	fs/open.c:933
9	sys_link	0x09	const char __user *oldname	const char __user *newname	-	-	-	fs/namei.c:2520

Showing 1 to 10 of 338 entries

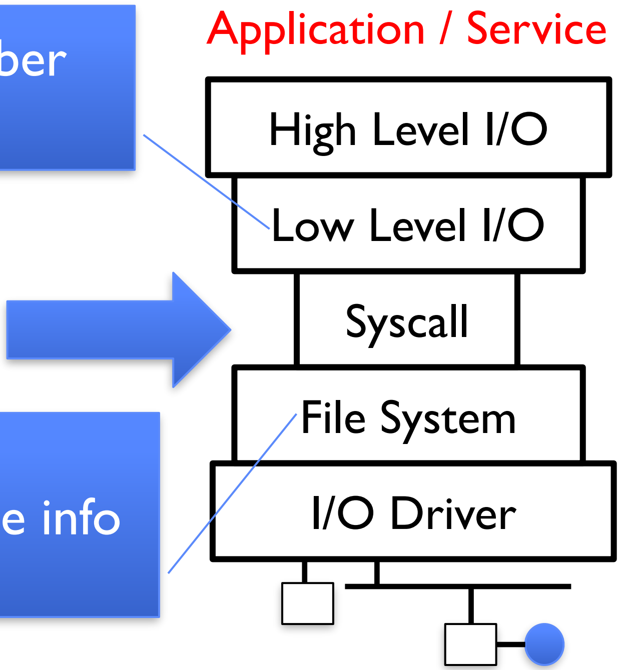
Generated from Linux kernel 2.6.35.4 using Exuberant Ctags, Python, and DataTables.
Project on GitHub. Hosted on GitHub Pages.

- 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 kernel

What's below the surface ??

File descriptor number
- an int

File Descriptors
- a struct with all the info about the files



streams

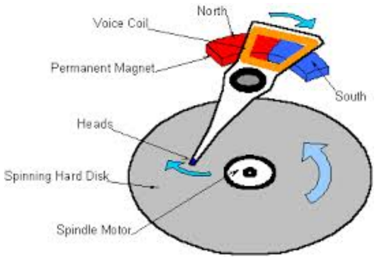
handles

registers

descriptors

Commands and Data Transfers

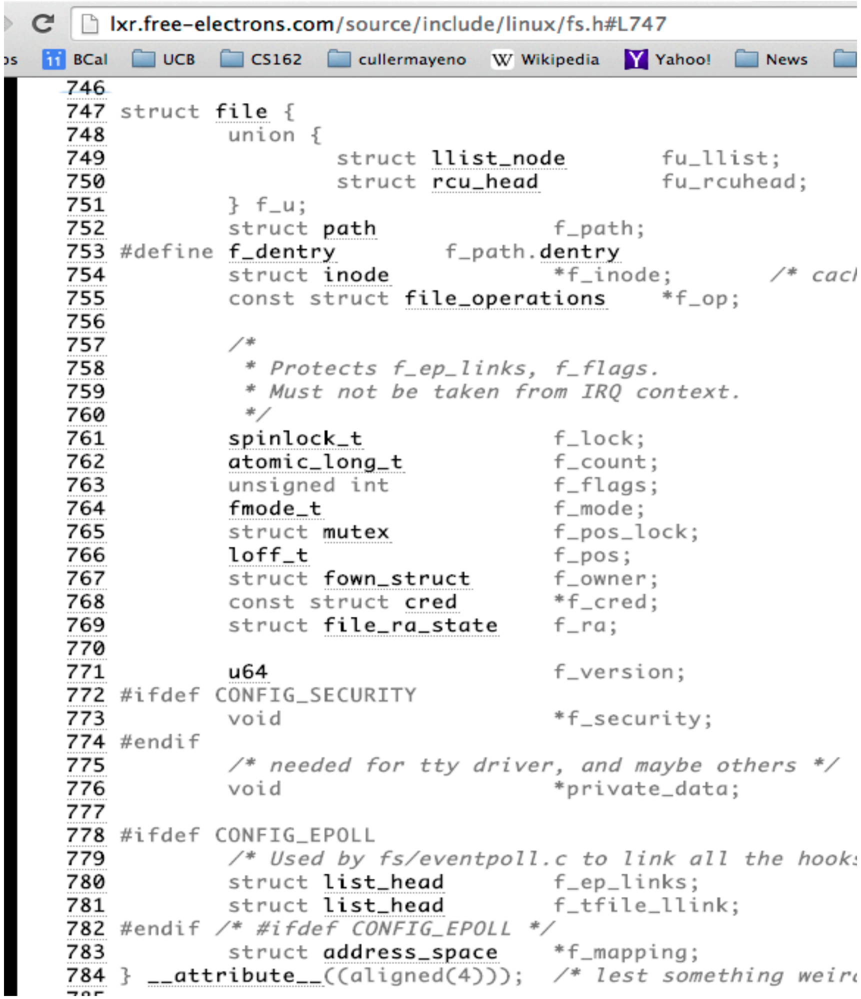
Disks, Flash, Controllers, DMA



Internal OS File Descriptor

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

- Pointer:
  `struct file *file`



```
struct file {
        union {
                struct llist_node       fu_llist;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct path             f_path;
#define f_dentry        f_path.dentry
        struct inode            *f_inode;       /* cac
        const struct file_operations    *f_op;

        /*
         * Protects f_ep_links, f_flags.
         * Must not be taken from IRQ context.
         */
        spinlock_t              f_lock;
        atomic_long_t           f_count;
        unsigned int            f_flags;
        fmode_t                 f_mode;
        struct mutex            f_pos_lock;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        const struct cred       *f_cred;
        struct file_ra_state    f_ra;

        u64                     f_version;
#ifdef CONFIG_SECURITY
        void                    *f_security;
#endif
        /* needed for tty driver, and maybe others */
        void                    *private_data;

#ifdef CONFIG_EPOLL
        /* Used by fs/eventpoll.c to link all the hook
        struct list_head        f_ep_links;
        struct list_head        f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
        struct address_space    *f_mapping;
} __attribute__((aligned(4)));  /* lest something weir
```

File System: from syscall to driver

In `fs/read_write.c`

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t
*pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) ret
  if (!file->f_op || (!file->f_op->read
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

- Read up to "count" bytes from "file" starting from "pos" into "buf".
- Return error or number of bytes read.

File System: from syscall to driver

In `fs/read_write.c`

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t
*pos)
{
  ssize t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

Make sure we are allowed to read this file

File System: from syscall to driver

In fs/read_write.c

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t
*pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

Check if file has read methods

File System: from syscall to driver

In fs/read_write.c

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t
*pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, c
    else
      ret = do_sync_read(file, buf, count
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

- Check whether we can write to buf (e.g., buf is in the user space range)
- unlikely(): hint to branch prediction this condition is unlikely

File System: from syscall to driver

In fs/read_write.c

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t
*pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, p
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

Check whether we read from a valid range in the file.

File System: from syscall to driver

In fs/read_write.c

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t
*pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

If driver provide a read function (f_op->read) use it; otherwise use do_sync_read()

File System: from syscall to driver

In fs/read_write.c

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t
*pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->re
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

Notify the parent of this file that the file was read (see http://www.fieldses.org/~bfields/kernel/vfs.txt)

File System: from syscall to driver

In fs/read_write.c

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t
*pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count,
    else
      ret = do_sync_read(file, buf, count, pos)
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

Update the number of bytes read by “current” task (for scheduling purposes)

File System: from syscall to driver

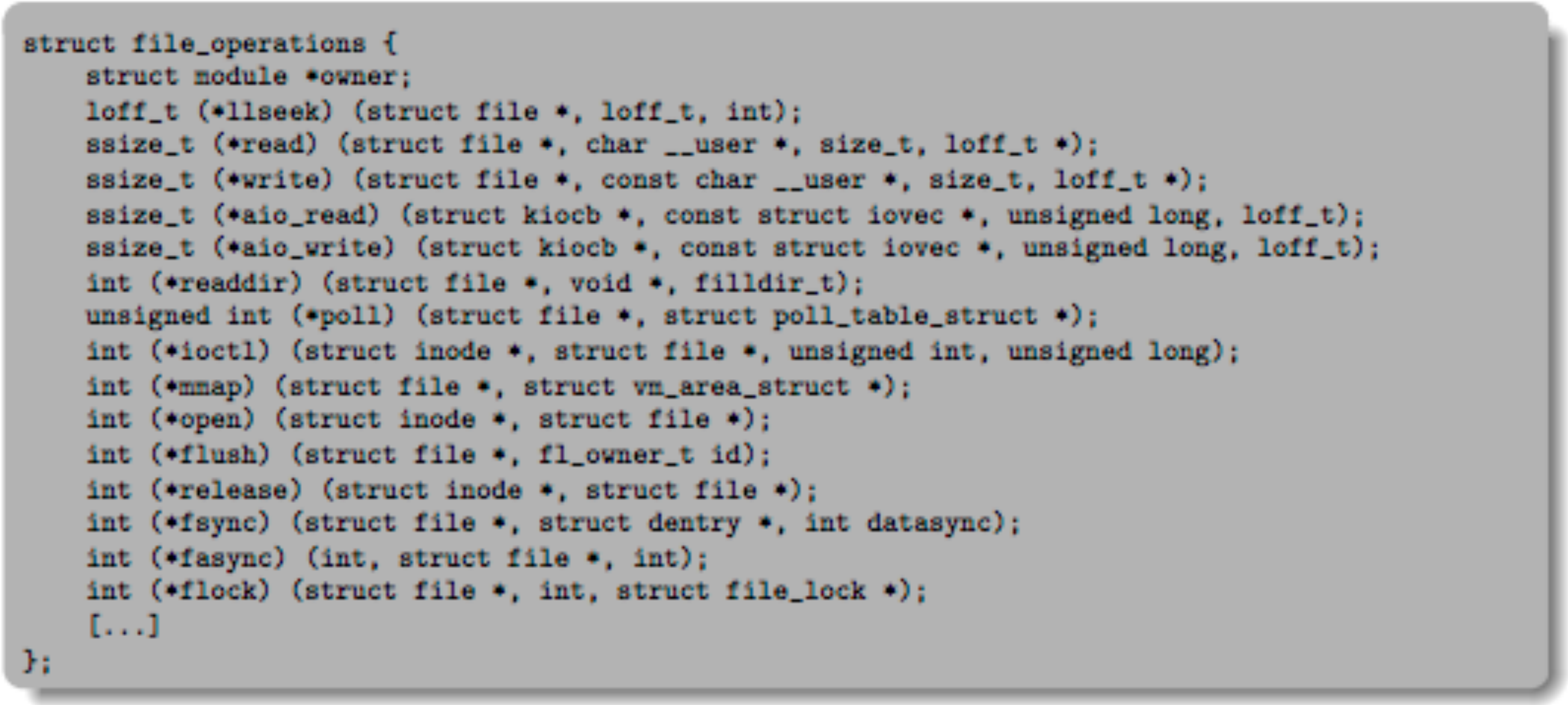
In fs/read_write.c

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t
*pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos)
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

Update the number of read syscalls by “current” task (for scheduling purposes)

Lower Level Driver

- Associated with particular hardware device
- Registers / Unregisters itself with the kernel
- Handler functions for each of the file operations



```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*flock) (struct file *, int, struct file_lock *);
    [...]
};
```

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

Life Cycle of An I/O Request

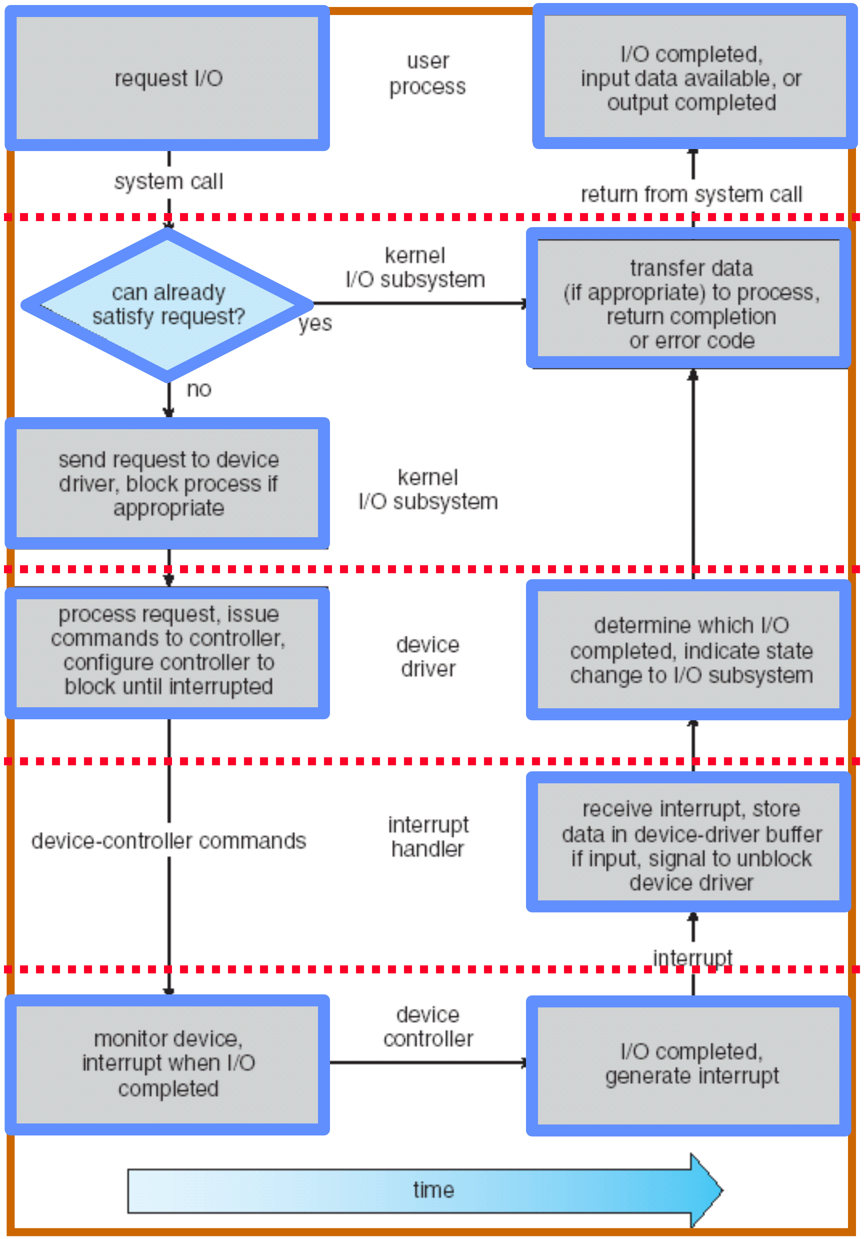
User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

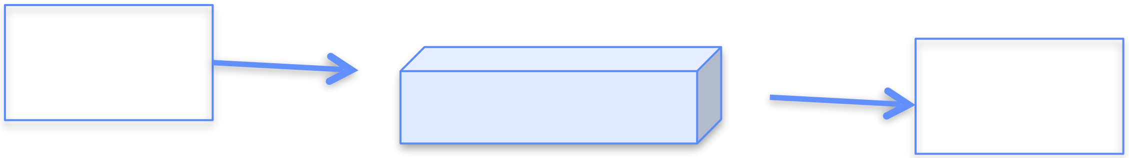
Device Hardware


request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
yes
kernel I/O subsystem
transfer data (if appropriate) to process, return completion or error code
no
send request to device driver, block process if appropriate
kernel I/O subsystem
process request, issue commands to controller, configure controller to block until interrupted
device driver
determine which I/O completed, indicate state change to I/O subsystem
device-controller commands
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt
time

Communication between processes

- Can we view files as communication channels?

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

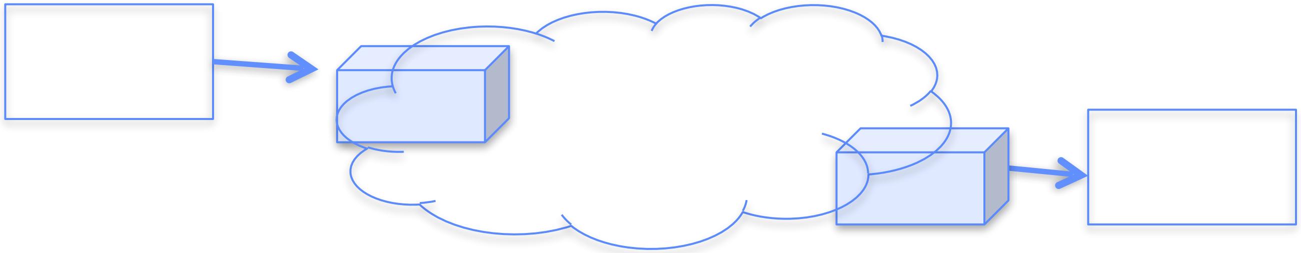


```
n = read(rfd,rbuf,rmax);
```

- Producer and Consumer of a file may be distinct processes
  – May be separated in time (or not)
- However, what if data written once and consumed once?
  – Don't we want something more like a queue?
  – Can still look like File I/O!

Communication Across the world looks like file IO

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



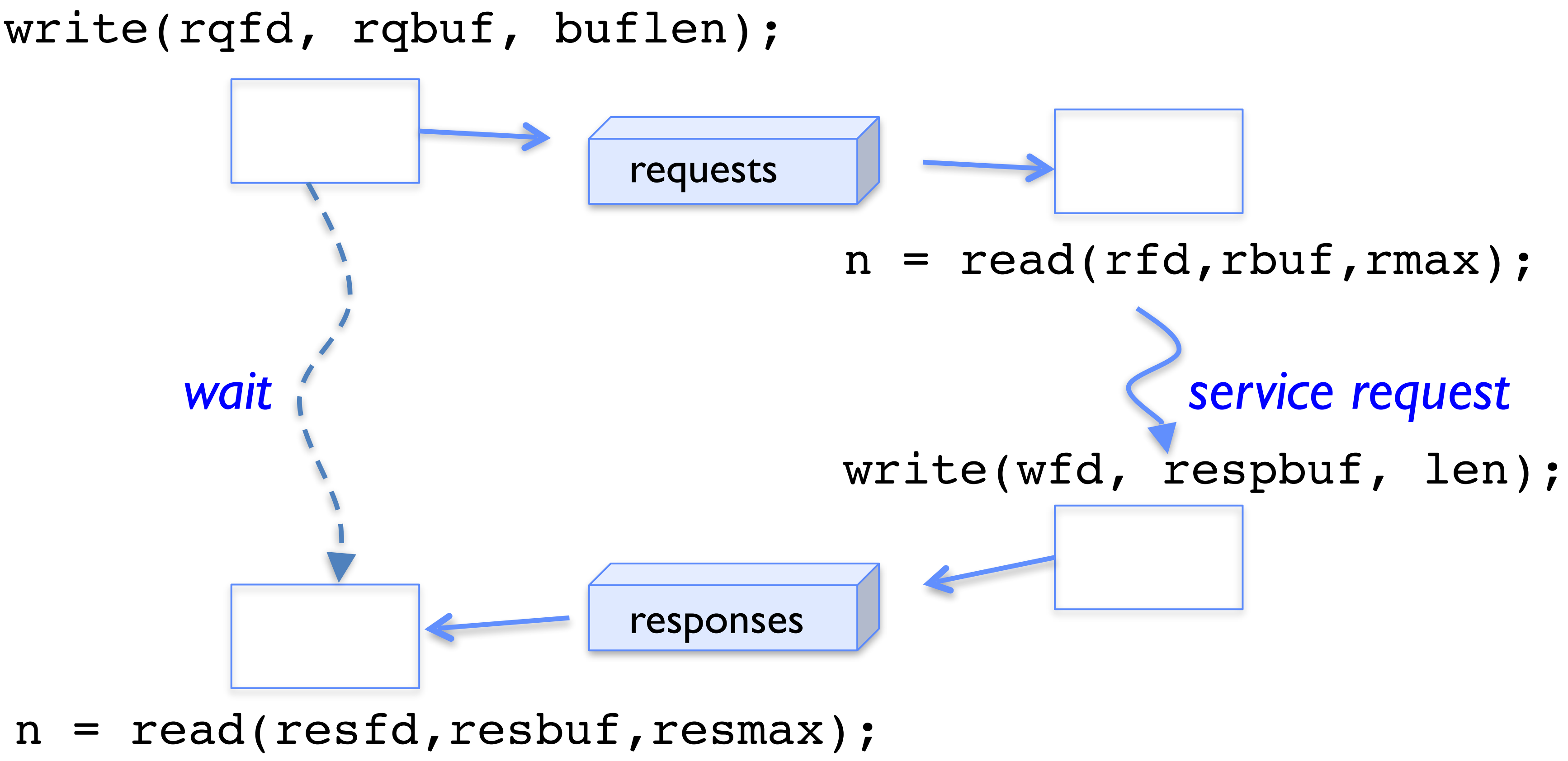
```
n = read(rfd,rbuf,rmax);
```

- Connected queues over the Internet
  - But 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)



```
write(rqfd, rqbuf, buflen);
```

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

```
write(wfd, respbuf, len);
```

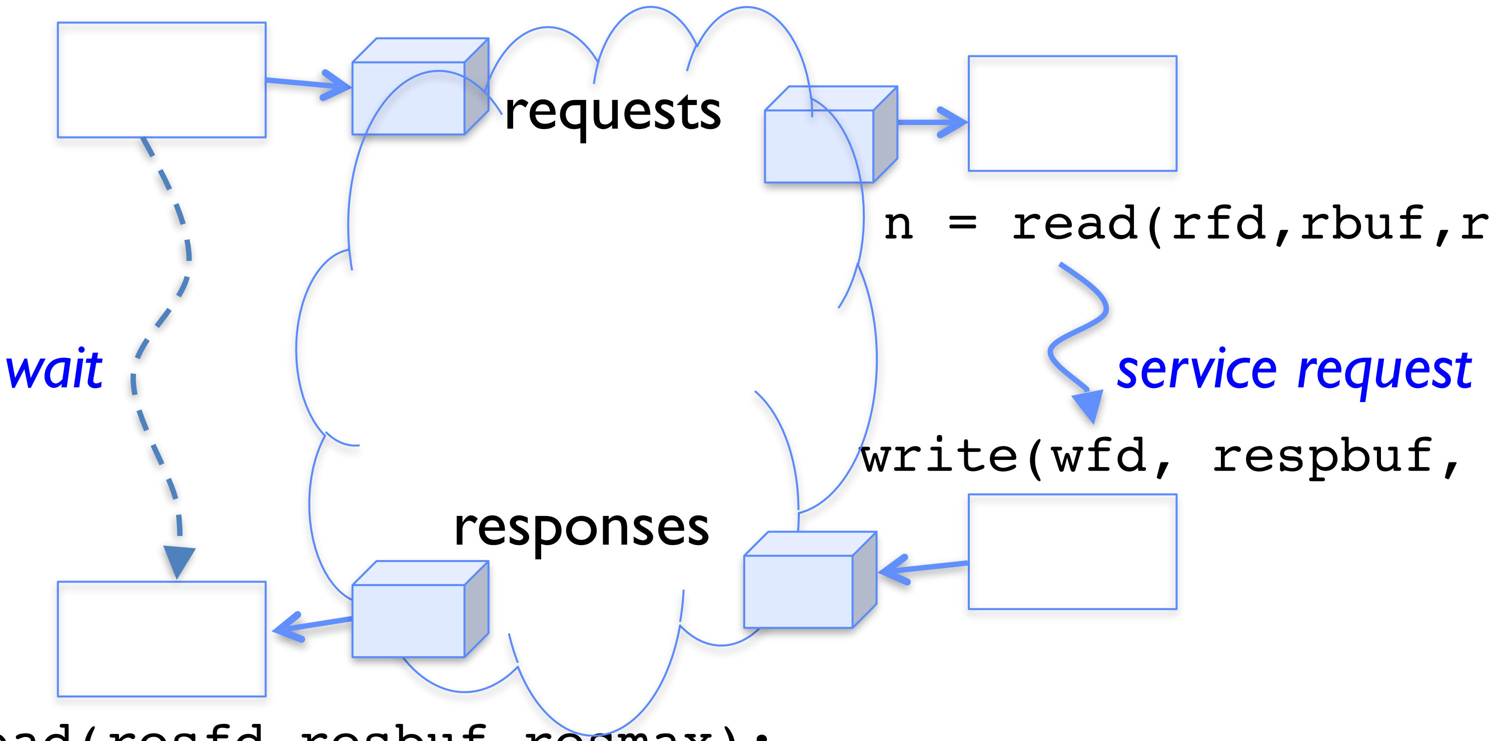
```
n = read(resfd,resbuf,resmax);
```

Request Response Protocol

Client (issues requests)

Server (performs operations)

```
write(rqfd, rqbuf, buflen);
```

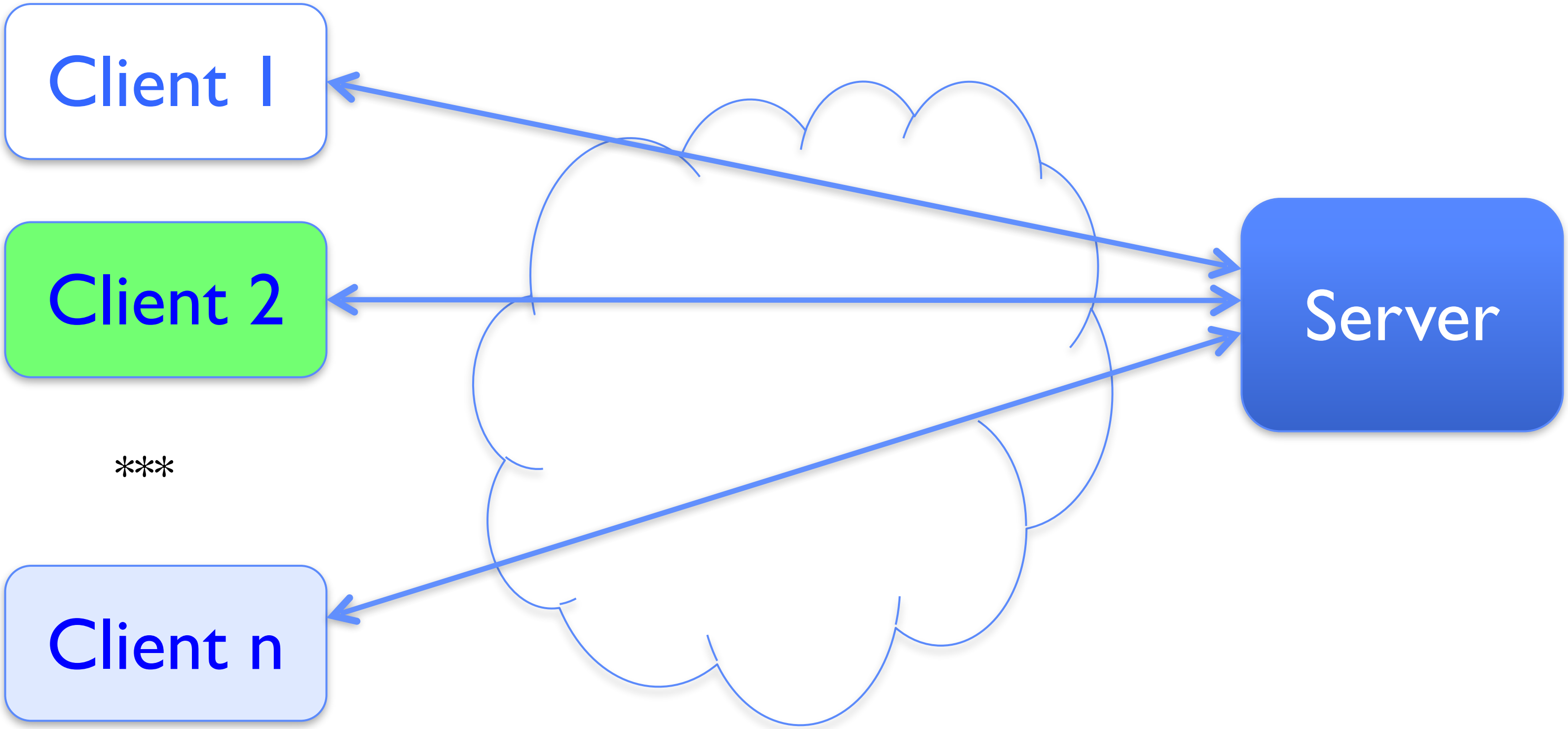


```
n = read(rfd,rbuf,rmax);
```

```
write(wfd, respbuf, len);
```

```
n = read(resfd,resbuf,resmax);
```

Client-Server Models

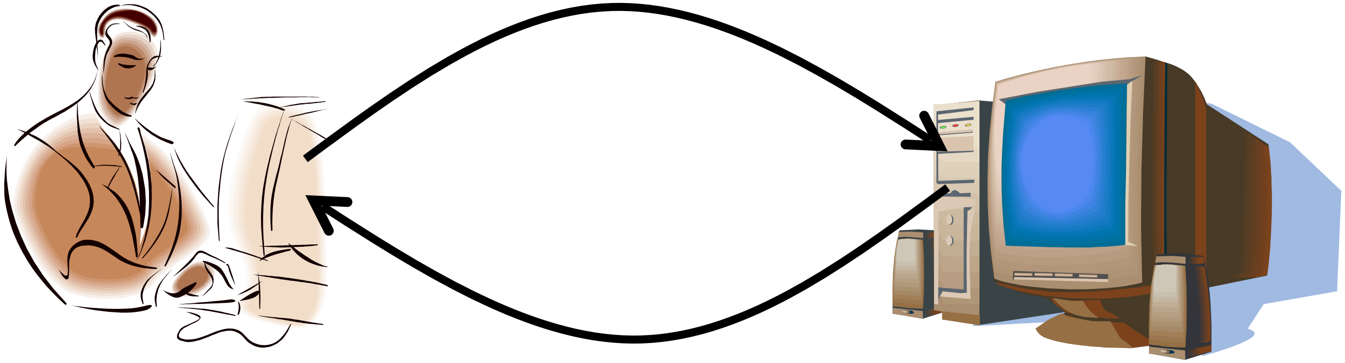


- File servers, web, FTP, Databases, …
- Many clients accessing a common server

Client-Server Communication

- Client "sometimes on"
  - Initiates a request to the server when interested
  - E.g., Web browser on your laptop or cell phone
  - Doesn't communicate directly with other clients
  - Needs to know the server's address

- Server is "always on"
  - Services requests from many client hosts
  - E.g., Web server for the *www.cnn.com* Web site
  - Doesn't initiate contact with the clients
  - Needs a fixed, well-known address



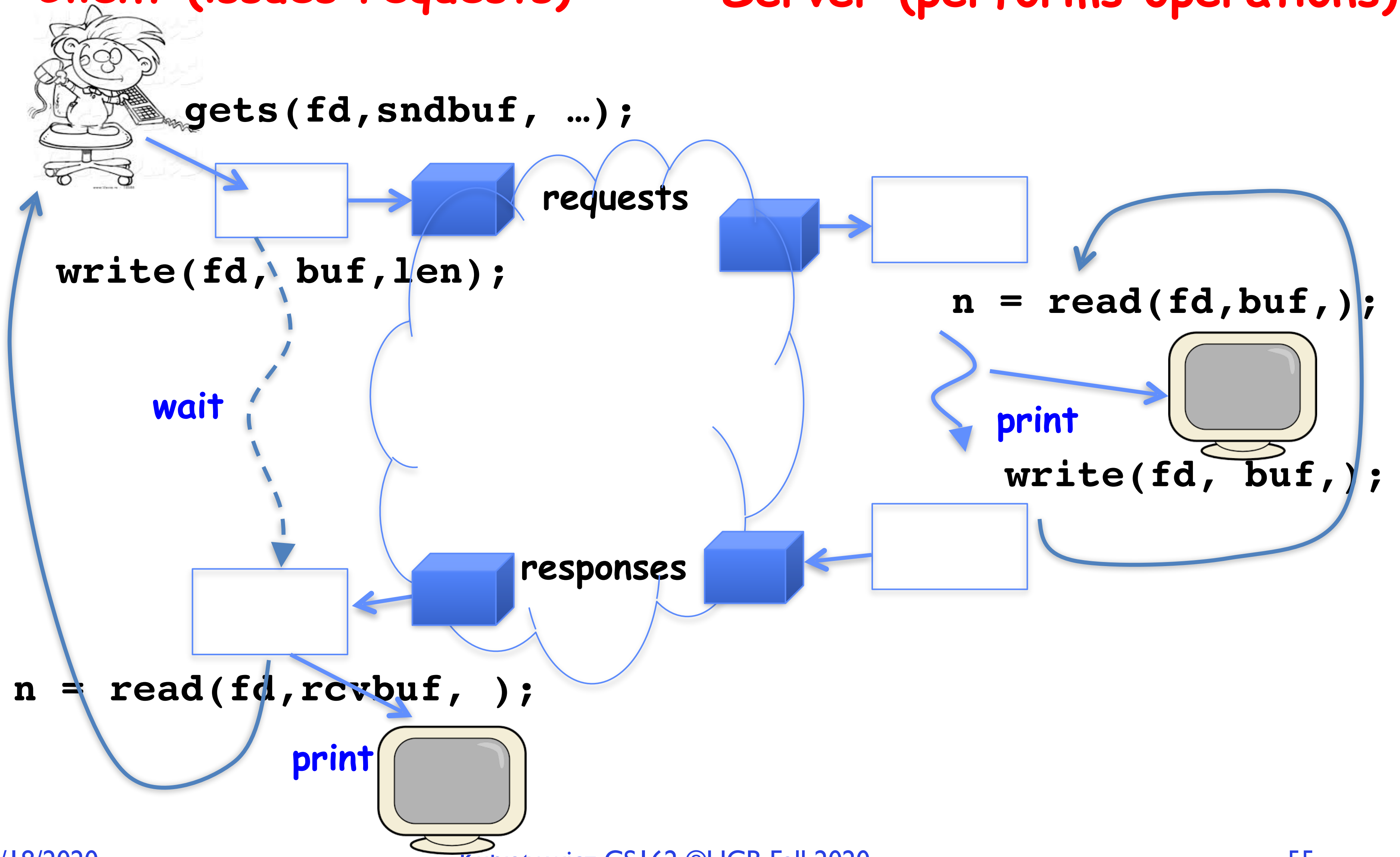
Sockets

- 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 (called "UNIX socket") or remote machine (called "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, …

Silly Echo Server – running example

Client (issues requests)

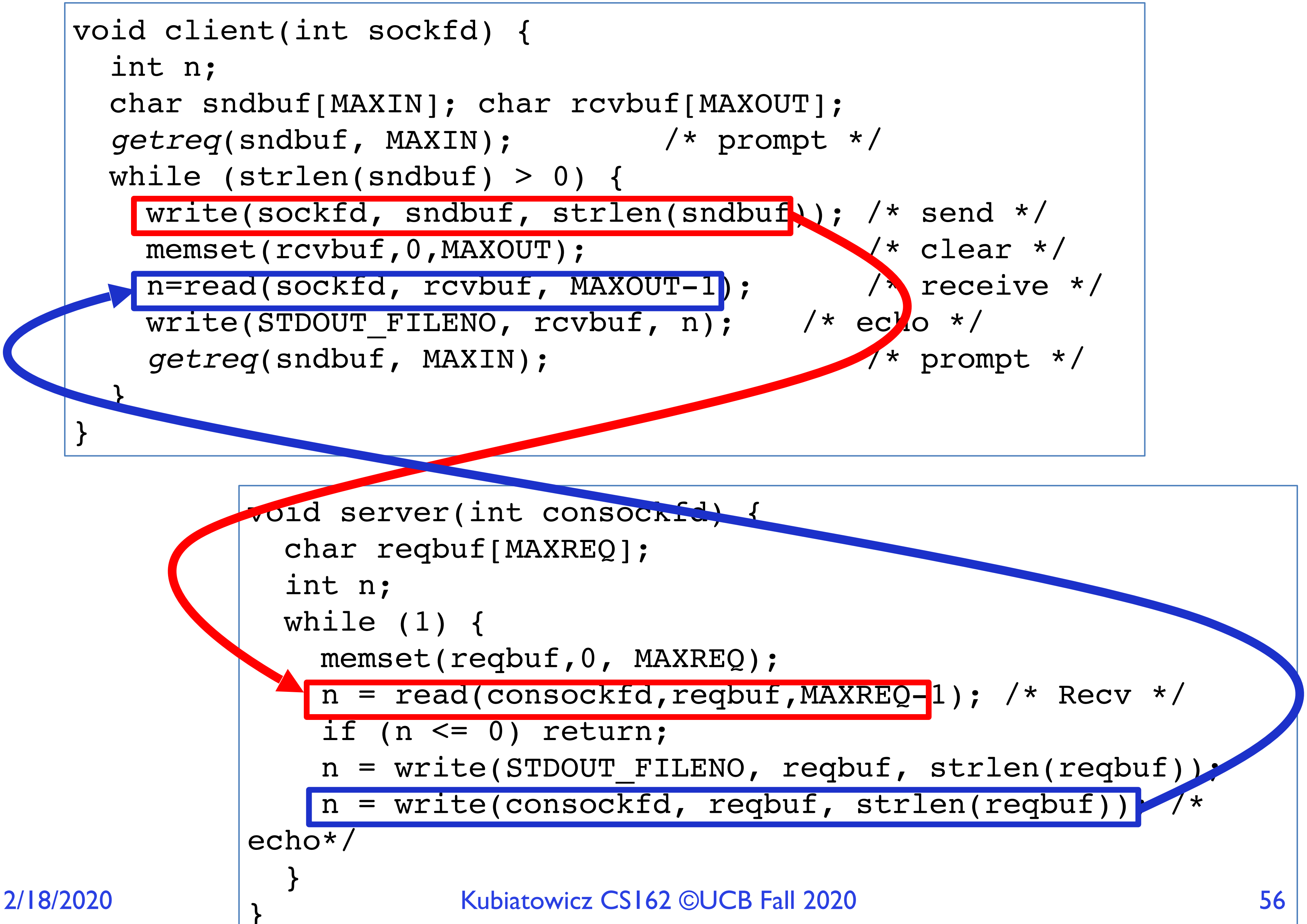
Server (performs operations)



```
gets(fd,sndbuf, …);
write(fd, buf,len);
n = read(fd,rcvbuf, );
```

```
n = read(fd,buf,);
write(fd, buf,);
```

Echo client-server example



```
void client(int sockfd) {
  int n;
  char sndbuf[MAXIN]; char rcvbuf[MAXOUT];
  getreq(sndbuf, MAXIN);        /* prompt */
  while (strlen(sndbuf) > 0) {
    write(sockfd, sndbuf, strlen(sndbuf)); /* send */
    memset(rcvbuf,0,MAXOUT);               /* clear */
    n=read(sockfd, rcvbuf, MAXOUT-1);      /* receive */
    write(STDOUT_FILENO, rcvbuf, n);    /* echo */
    getreq(sndbuf, MAXIN);                 /* prompt */
  }
}
```

```
void server(int consockfd) {
  char reqbuf[MAXREQ];
  int n;
  while (1) {
    memset(reqbuf,0, MAXREQ);
    n = read(consockfd,reqbuf,MAXREQ-1); /* Recv */
    if (n <= 0) return;
    n = write(STDOUT_FILENO, reqbuf, strlen(reqbuf));
    n = write(consockfd, reqbuf, strlen(reqbuf)); /*
echo*/
  }
}
```

What assumptions are we making?

- Reliable
  - Write to a file => Read it back. Nothing is lost.
  - Write to a (TCP) socket => Read from the other side, same.
  - Like pipes
- In order (sequential stream)
  - Write X then write Y => read gets X then read gets Y
- When ready?
  - File read gets whatever is there at the time. Assumes writing already took place.
  - Like pipes!

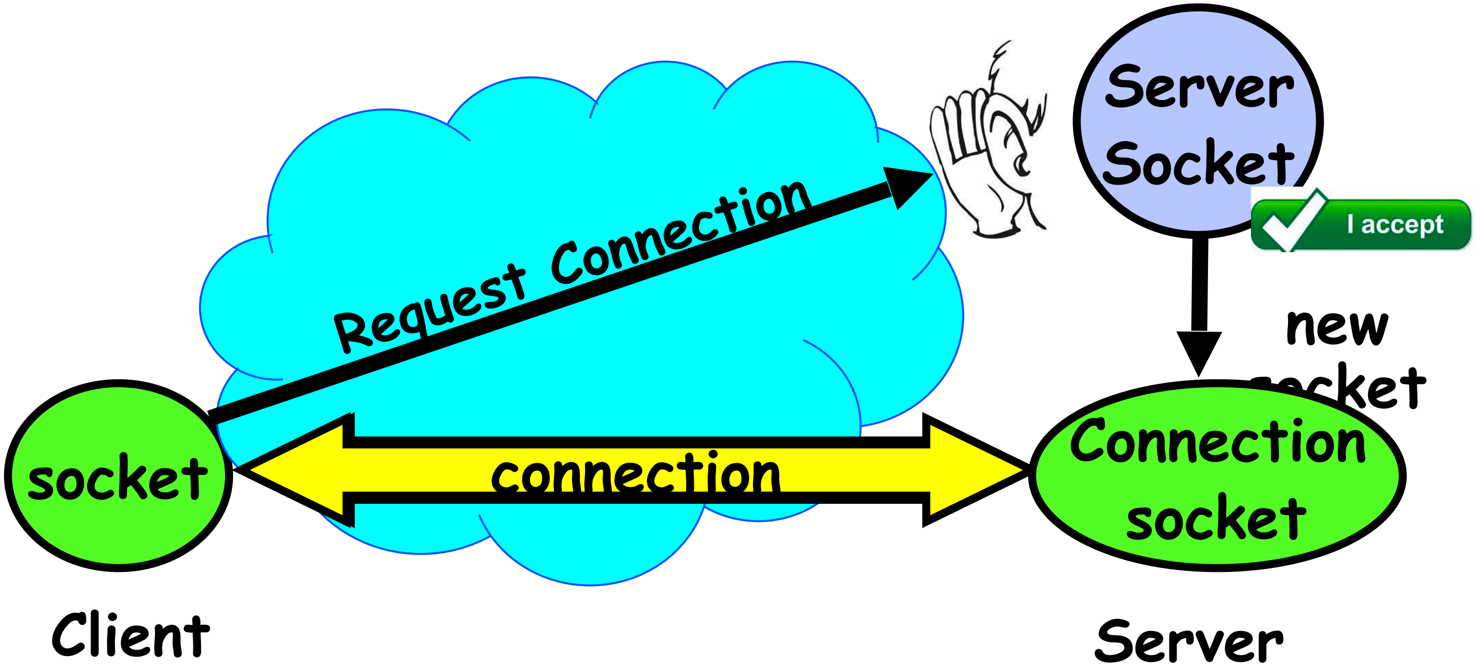
Socket creation and connection

- 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
- How do we name them?
- How do these completely independent programs know that the other wants to "talk" to them?

Namespaces for communication over IP

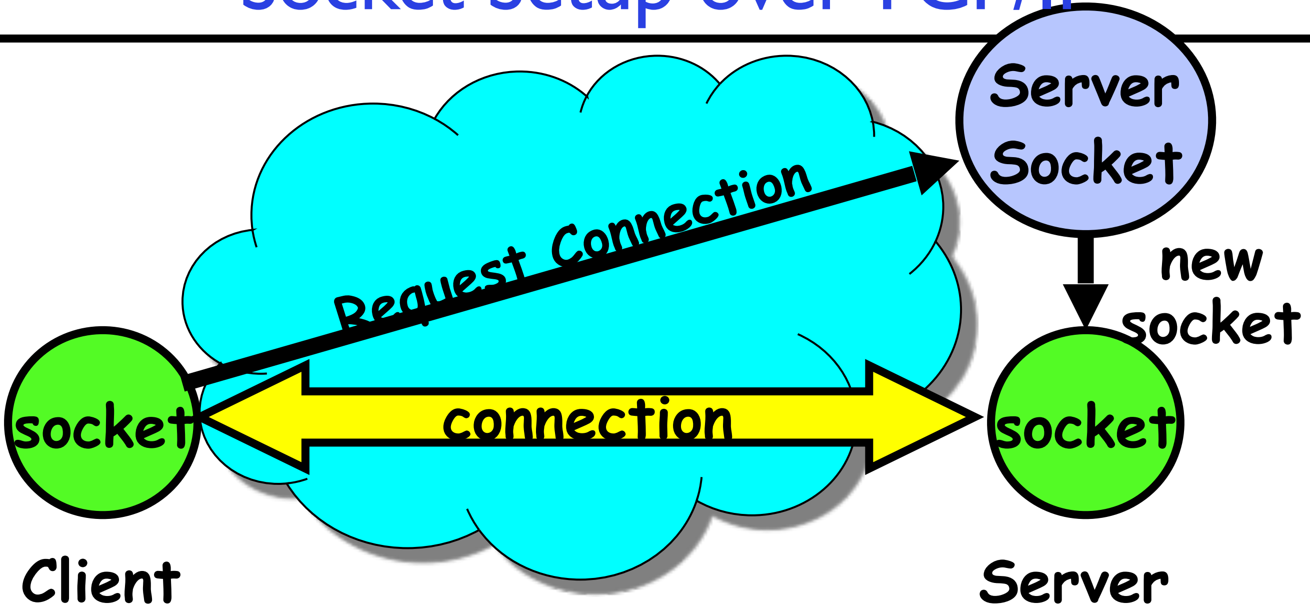
- Hostname
  - www.eecs.berkeley.edu
- IP address
  - 128.32.244.172 (ipv6?)
- Port Number
  - 0-1023 are "well known" or "system" ports
    - » Superuser privileges to bind to one
  - 1024 – 49151 are "registered" ports (registry)
    - » Assigned by IANA for specific services
  - 49152–65535 ($2^{15}+2^{14}$ to $2^{16}-1$) are "dynamic" or "private"
    - » Automatically allocated as "ephemeral Ports"

Socket Setup over TCP/IP



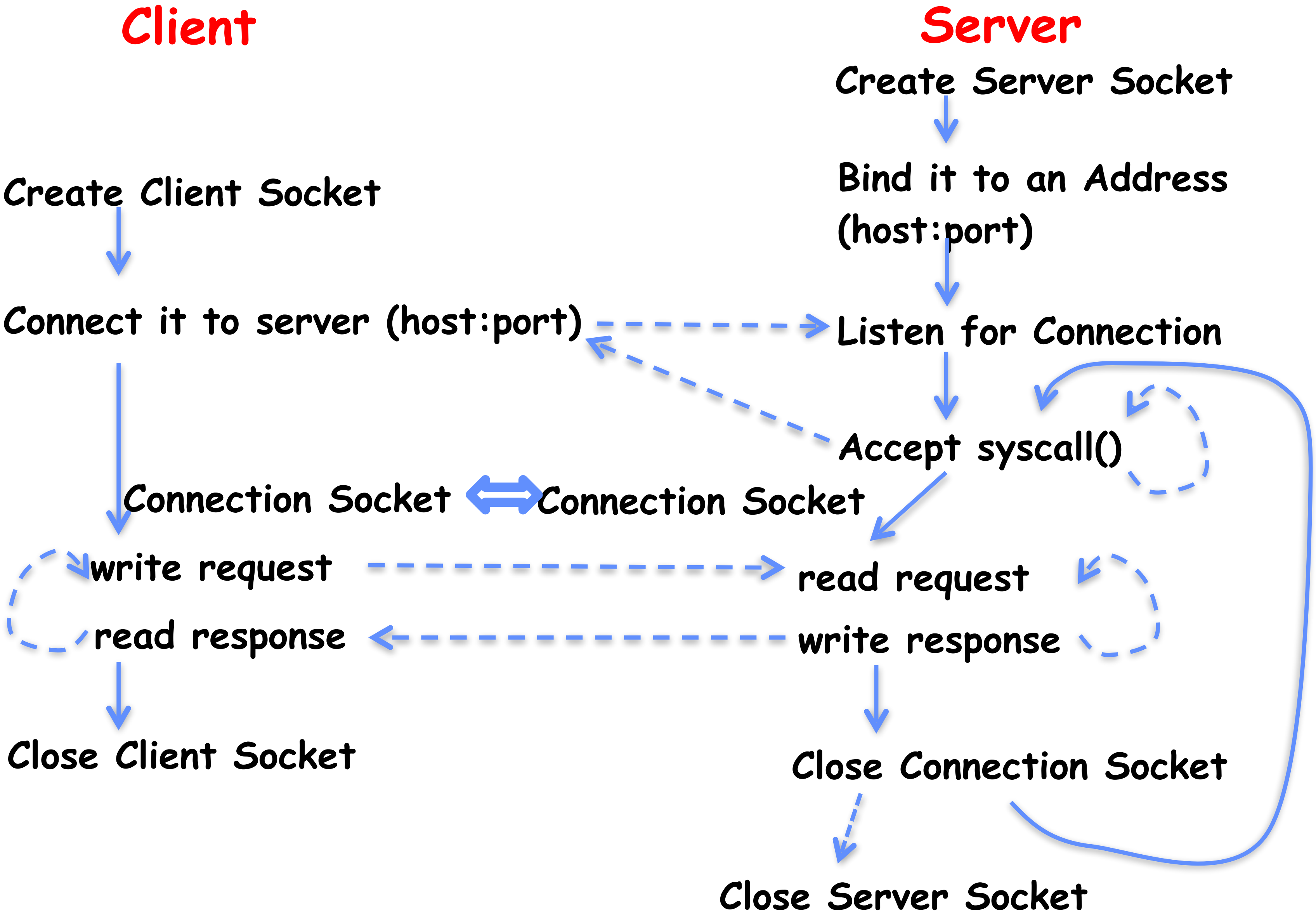
- Special kind of socket: server socket
  - Has file descriptor
  - Can't read or write
- Two operations:
  1. **`listen()`**: Start allowing clients to connect
  2. **`accept()`**: Create a *new socket* for a *particular* client connection

Socket Setup over TCP/IP



- Server Socket: Listens for new connections
  - Produces new sockets for each unique connection
  - 3-way handshake to establish new connection!
- Things to remember:
  - Connection involves 5 values:
    [Client Addr, Client Port, Server Addr, Server Port, Protocol]
  - Often, Client Port "randomly" assigned
    - » Done by OS during client socket setup
  - Server Port often "well known"
    - » 80 (web), 443 (secure web), 25 (sendmail), etc
    - » Well-known ports from 0—1023

Web Server using Sockets (in concept)



Client

Create Client Socket

Connect it to server (host:port)

Connection Socket

write request

read response

Close Client Socket

Server

Create Server Socket

Bind it to an Address (host:port)

Listen for Connection

Accept syscall()

Connection Socket

read request

write response

Close Connection Socket

Close Server Socket

Client Protocol

```
char *host_name, port_name;

// Create a socket
struct addrinfo *server = lookup_host(host_name, port_name);
int sock_fd = socket(server->ai_family, server->ai_socktype,
                     server->ai_protocol);

// Connect to specified host and port
connect(sock_fd, server->ai_addr, server->ai_addrlen);

// Carry out Client-Server protocol
run_client(sock_fd);

/* Clean up on termination */
close(sock_fd);
```

Client: getting the server address

```
struct addrinfo *lookup_host(char *host_name, char *port) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;

  int rv = getaddrinfo(host_name, port_name,
                       &hints, &server);
  if (rv != 0) {
    printf("getaddrinfo failed: %s\n", gai_strerror(rv));
    return NULL;
  }
  return server;
}
```

Server Protocol (v1)

```
// Create socket to listen for client connections
char *port_name;
struct addrinfo *server = setup_address(port_name);
int server_socket = socket(server->ai_family,
      server->ai_socktype, server->ai_protocol);

// Bind socket to specific port
bind(server_socket, server->ai_addr, server->ai_addrlen);

// Start listening for new client connections
listen(server_socket, MAX_QUEUE);

while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  serve_client(conn_socket);
  close(conn_socket);
}

close(server_socket);
```

Server Address - itself

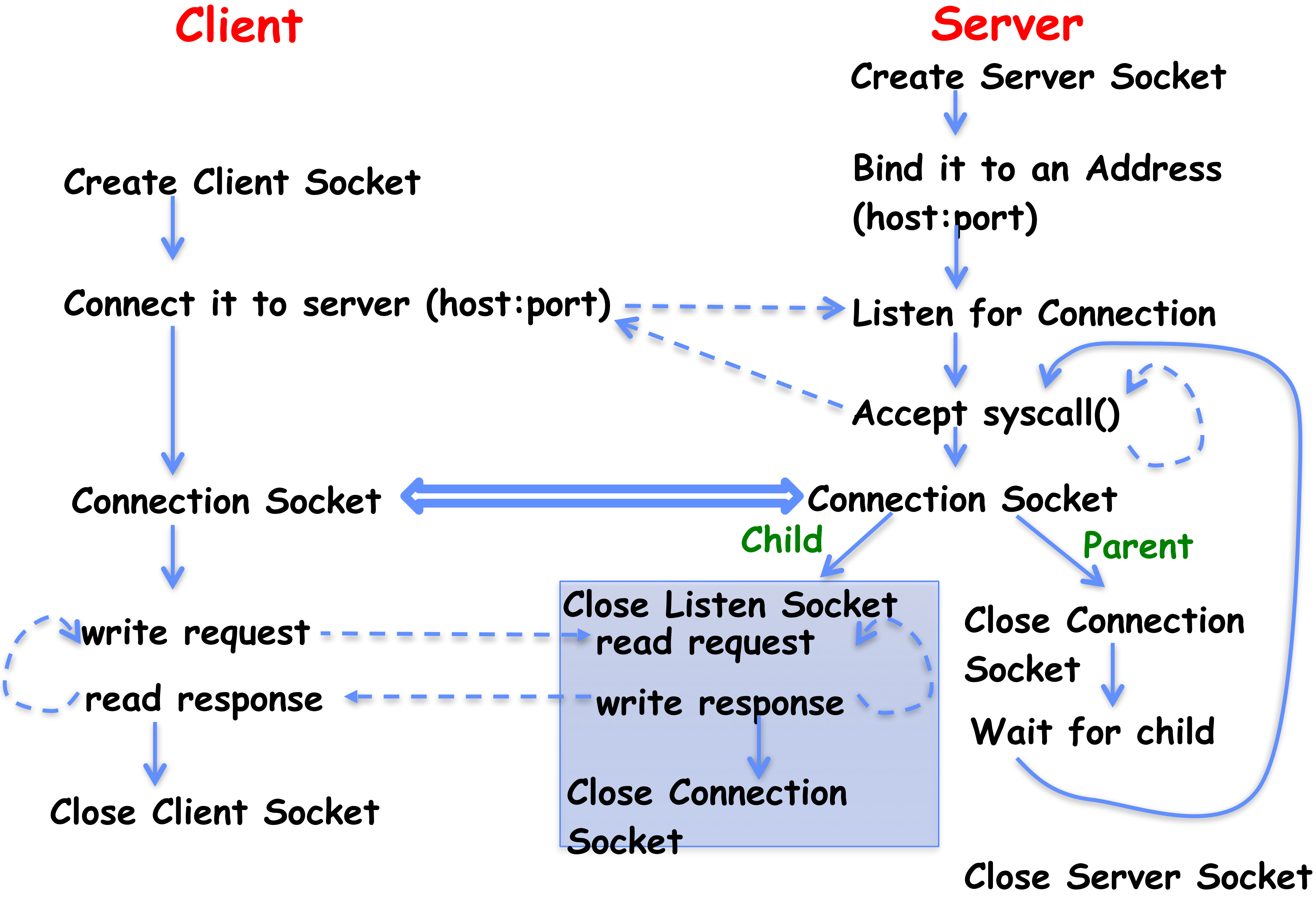
```
struct addrinfo *setup_address(char *port) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;
  getaddrinfo(NULL, port, &hints, &server);
  return server;
}
```

- Simple form
- Internet Protocol, TCP
- Accepting any connections on the specified port

How does the server protect itself?

- Isolate the handling of each connection
- By forking it off as another process

Sockets With Protection



Client

Server

Create Server Socket

Bind it to an Address (host:port)

Create Client Socket

Connect it to server (host:port)

Listen for Connection

Accept syscall()

Connection Socket

Connection Socket

Child

Parent

Close Listen Socket

read request

write response

Close Connection Socket

Close Connection Socket

Wait for child

write request

read response

Close Client Socket

Close Server Socket

Server Protocol (v2)

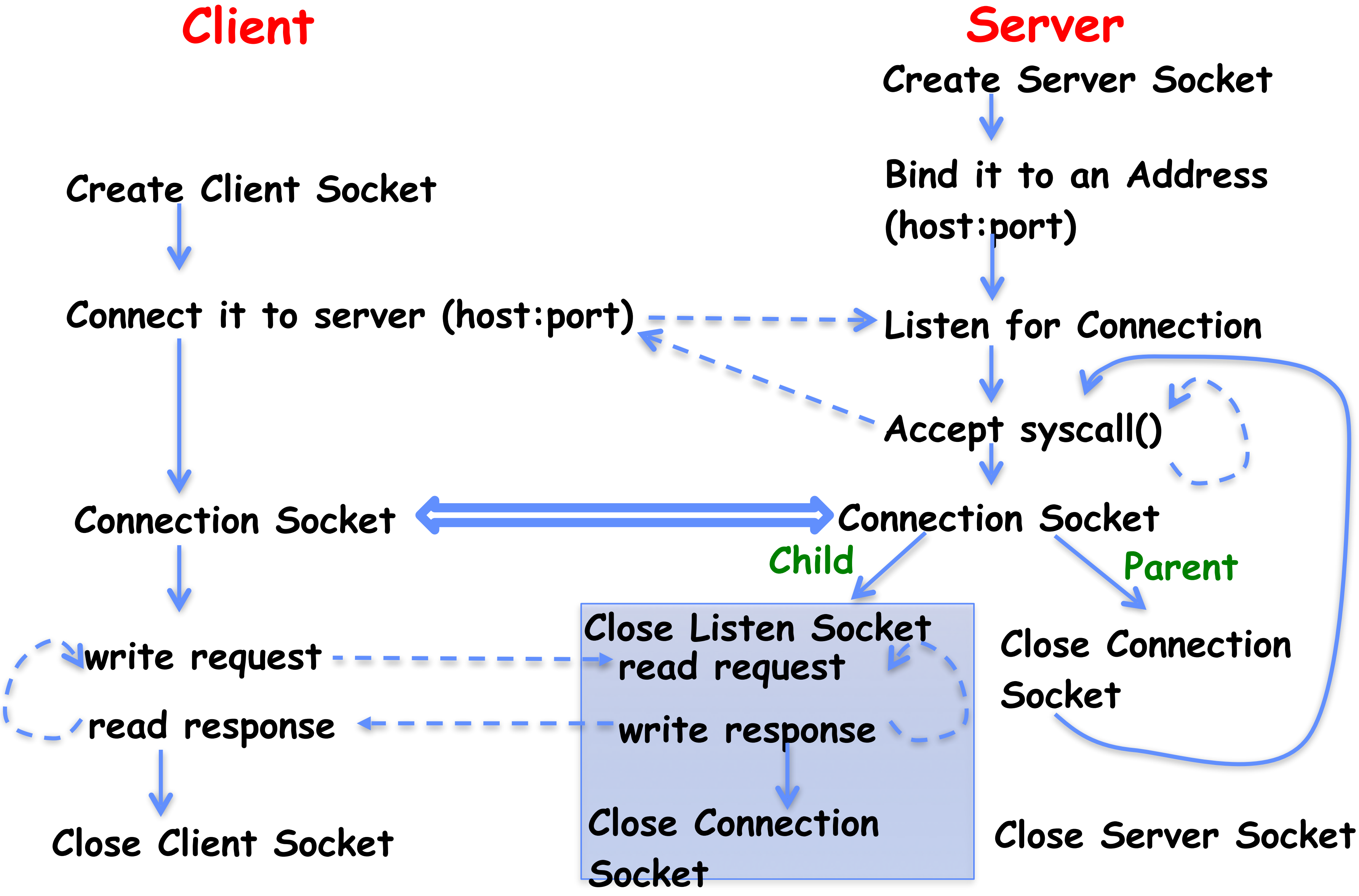
```
// Start listening for new client connections
listen(server_socket, MAX_QUEUE);
while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);

  pid_t pid = fork();                // New process for connection
  if (pid == 0) {                    // Child process
    close(server_socket);            // Doesn't need server_socket
    serve_client(conn_socket);       // Serve up content to client
    close(conn_socket);              // Done with client!
    exit(EXIT_SUCCESS);
  } else {                           // Parent process
    close(conn_socket);              // Don't need client socket
    wait(NULL);                      // Wait for our (one) child
  }
}
close(server_socket);
```

Concurrent Server

- Listen will queue requests
- Buffering present elsewhere
- But server waits for each connection to terminate before initiating the next

Sockets With Protection and Parallelism



Client

Create Client Socket

Connect it to server (host:port)

Connection Socket

write request

read response

Close Client Socket

Server

Create Server Socket

Bind it to an Address (host:port)

Listen for Connection

Accept syscall()

Connection Socket

Child

Close Listen Socket

read request

write response

Close Connection Socket

Parent

Close Connection Socket

Close Server Socket

Server Protocol (v3)

```
// Start listening for new client connections
listen(server_socket, MAX_QUEUE);
signal(SIGCHLD,SIG_IGN);          // Prevent zombie children
while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);

  pid_t pid = fork();             // New process for connection
  if (pid == 0) {                 // Child process
    close(server_socket);         // Doesn't need server_socket
    serve_client(conn_socket);    // Serve up content to client
    close(conn_socket);           // Done with client!
    exit(EXIT_SUCCESS);
  } else {                        // Parent process
    close(conn_socket);           // Don't need client socket
    // wait(NULL);                // Don't wait (SIGCHLD
                           //   ignored, above)
  }
}
close(server_socket);
```

Conclusion (I)

- System Call Interface is "narrow waist" between user programs and kernel

- Streaming IO: modeled as a stream of bytes
  - Most streaming I/O functions start with "f" (like "**`fread`**")
  - Data buffered automatically by C-library functions

- Low-level I/O:
  - File descriptors are integers
  - Low-level I/O supported directly at system call level

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

Conclusion (II)

- 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

- File abstraction works for inter-processes communication (local or Internet)

- Socket: an abstraction of a network I/O queue
  - Mechanism for inter-process communication