

CS162
Operating Systems and
Systems Programming
Lecture 5

Device Drivers, Sockets and IPC (Finished)

January 31st, 2023
Prof. John Kubiatowicz
<http://cs162.eecs.Berkeley.edu>

Recall: Low-Level File I/O: The RAW system-call interface

```
#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

- Integer return from `open()` is a *file descriptor*
 - Error indicated by `return < 0`: the global `errno` variable set with error (see man pages)
- Operations on *file descriptors*:
 - Open system call created an *open file description* entry in system-wide table of open files
 - *Open file description* object in the kernel represents an instance of an open file
 - Why give user an integer instead of a pointer to the file description in kernel?

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.2

Recall: Example: `lowio.c`

```
int main() {
    char buf[1000];

    int fd = open("lowio.c", O_RDONLY, S_IRUSR | S_IWUSR);
    if (fd < 0) {
        /* Error! errno variable contains specification of which error */
        exit(1);
    }

    ssize_t rd = read(fd, buf, sizeof(buf));
    if (rd < 0) {
        /* Error! errno variable contains specification of which error */
        exit(1);
    }

    int err = close(fd);
    /* check for errors */

    ssize_t wr = write(STDOUT_FILENO, buf, rd);
    /* check for errors */
}
```

- How many bytes does this program read?

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.3

Low-Level I/O: Other Operations

- Operations specific to terminals, devices, networking, ...
 - e.g., `ioctl`
- Duplicating descriptors
 - `int dup2(int old, int new);`
 - `int dup(int old);`
- Pipes – channel
 - `int pipe(int pipefd[2]);`
 - Writes to `pipefd[1]` can be read from `pipefd[0]`
- File Locking
- Memory-Mapping Files
- Asynchronous I/O

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.4

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**

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.5

POSIX I/O: Kernel Buffering

- **Reads are buffered inside kernel**
 - Part of making everything byte-oriented
 - Process is **blocked** while waiting for device
 - Let other processes run while gathering result
- **Writes are buffered inside kernel**
 - Complete in background (more later on)
 - Return to user when data is “handed off” to kernel
- This buffering is part of global buffer management and caching for block devices (such as disks)
 - Items typically cached in quanta of disk block sizes
 - We will have many interesting things to say about this buffering when we dive into the kernel

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.6

Low-Level vs High-Level file API

- | | |
|--|---|
| <ul style="list-style-type: none"> • Low-level direct use of syscall interface: <code>open()</code>, <code>read()</code>, <code>write()</code>, <code>close()</code> • Opening of file returns file descriptor: <pre>int myfile = open(...);</pre> • File descriptor only meaningful to kernel <ul style="list-style-type: none"> – Index into process (PDB) which holds pointers to kernel-level structure (“file description”) describing file. • Every <code>read()</code> or <code>write()</code> causes syscall no matter how small (could read a single byte) • Consider loop to get 4 bytes at a time using <code>read()</code>: <ul style="list-style-type: none"> – Each iteration enters kernel for 4 bytes. | <ul style="list-style-type: none"> • High-level buffered access: <code>fopen()</code>, <code>fread()</code>, <code>fwrite()</code>, <code>fclose()</code> • Opening of file returns ptr to FILE: <pre>FILE *myfile = fopen(...);</pre> • FILE structure is user space contains: <ul style="list-style-type: none"> – a chunk of memory for a buffer – the file descriptor for the file (<code>fopen()</code> will call <code>open()</code> automatically) • Every <code>fread()</code> or <code>fwrite()</code> filters through buffer and may not call <code>read()</code> or <code>write()</code> on every call. • Consider loop to get 4 bytes at a time using <code>fread()</code>: <ul style="list-style-type: none"> – First call to <code>fread()</code> calls <code>read()</code> for block of bytes (say 1024). Puts in buffer and returns first 4 to user. – Subsequent <code>fread()</code> grab bytes from buffer |
|--|---|

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.7

Low-Level vs. High-Level File API

Low-Level Operation:

```
ssize_t read(...) {
```

```
asm code ... syscall # into %eax
put args into registers %ebx, ...
special trap instruction
```

```
Kernel:
get args from regs
dispatch to system func
Do the work to read from the file
Store return value in %eax
```

```
get return values from regs
```

```
Return data to caller
```

```
};
```

High-Level Operation:

```
ssize_t fread(...) {
```

```
Check buffer for contents
Return data to caller if available
```

```
asm code ... syscall # into %eax
put args into registers %ebx, ...
special trap instruction
```

```
Kernel:
get args from regs
dispatch to system func
Do the work to read from the file
Store return value in %eax
```

```
get return values from regs
```

```
Update buffer with excess data
Return data to caller
```

```
};
```

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.8

High-Level vs. Low-Level File API

- 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);
```

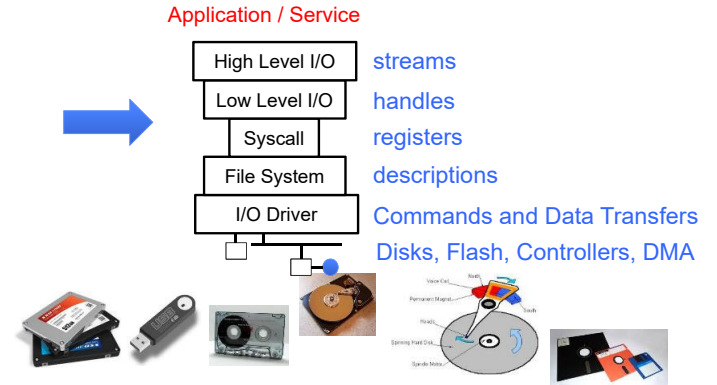
 Outputs "Beginning of line" 10 seconds earlier than "and end of line"

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.9

What's below the surface ??



1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.10

Recall: SYSCALL

#	Name	O	eax	ebx	ecx	edx	esi	edi	Definition
0	sys_restart_syscall	0x00							kernel/rdm/c2058
1	sys_exit	0x01	int error_code						kernel/rx/c1040
2	sys_fork	0x02	kernel/gt/rags *						arch/x86/kernel/entry.S:716
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count				fs/read_write.c:393
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 *pathname	int flags	int mode				fs/open.c:300
6	sys_close	0x06	unsigned int fd						fs/open.c:389
7	sys_waitpid	0x07	int pid	int __user *stat_addr	int options				kernel/rx/c1771
8	sys_create	0x08	const char __user *pathname	int mode					fs/open.c:333
9	sys_link	0x09	const char __user *oldname	const char __user *newname					fs/namei.c:1120

Generated from Linux kernel 2.6.35.4 using [Exuberant Ctags](#), [Python](#), and [DataTables](#).
Project on [GitHub](#). Hosted on [GitHub Pages](#).

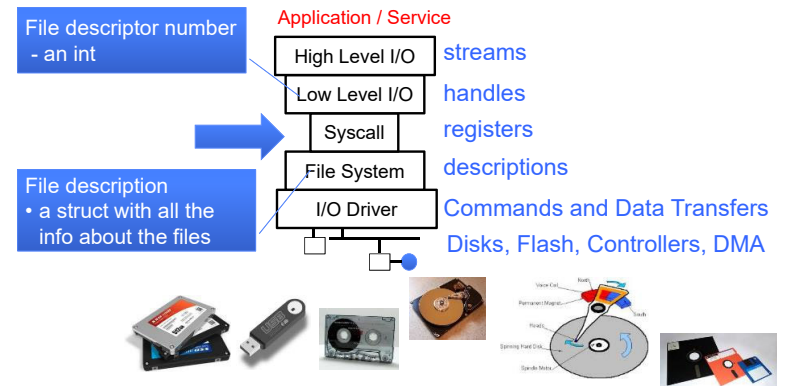
- Low level lib parameters are set up in registers and syscall instruction is issued
 - A type of synchronous exception that enters well-defined entry points into kernel

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.11

What's below the surface ??



1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.12

What's in an Open File Description?

Inside Kernel!

For our purposes, the two most important things are:

- Where to find the file data on disk
- The current position within the file

```

746
747 struct file {
748     union {
749         struct llist_node    fu_llist;
750         struct rcu_head      fu_rcuhead;
751     } f_u;
752     struct path              f_path;
753     struct f_dentry          f_dentry;
754     struct inode             f_inode; /* caci
755     const struct file_operations *f_op;
756
757     /*
758     * Protects f_op.links, f_flags.
759     * Must not be taken from IRQ context.
760     */
761     spinlock_t              f_lock;
762     atomic_long_t           f_count;
763     unsigned int            f_flags;
764     fmode_t                 f_mode;
765     struct mutex             f_pos_lock;
766     loff_t                  f_pos;
767     struct fown_struct       f_owner;
768     const struct cred        *f_cred;
769     struct file_ra_state     f_ra;
770
771     u64                      f_version;
772 #ifdef CONFIG_SECURITY
773     void                     *f_security;
774 #endif
775     /* needed for tty driver, and maybe others */
776     void                     *private_data;
777
778 #ifdef CONFIG_EPOLL
779     /* Used by fs/eventpoll.c to link all the hooks
780     struct list_head          f_ep_links;
781     struct list_head         f_tfile_llink;
782 #endif /* #ifdef CONFIG_EPOLL */
783     struct address_space     *f_mapping;
784 } __attribute__((aligned(4))); /* test something weird
    
```

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 -EIO;
    if (!(file->f_op || (!file->f_op->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;
}
    
```

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

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.17

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 whether we read from a valid range in the file.

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.18

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()

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.19

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;
}
```

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

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.20

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 bytes read by "current" task (for scheduling purposes)

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.21

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)

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.22

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 *);
    [...]
};
```

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.23

Device Drivers

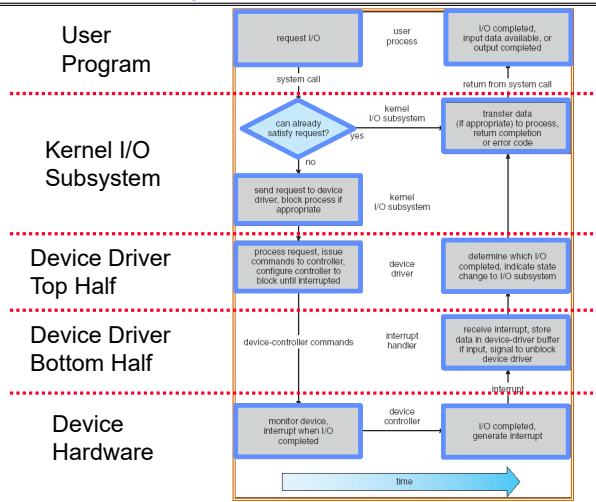
- **Device Driver:** Device-specific code in the kernel that interacts directly with the device hardware
 - Supports a standard, internal interface
 - Same kernel I/O system can interact easily with different device drivers
 - Special device-specific configuration supported with the `ioctl()` system 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

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.24

Life Cycle of An I/O Request



Kubiatowicz CS162 © UCB Spring 2023

Lec 5.25

1/31/2023

Administrivia

- Kubiatowicz Office Hours:
 - 2-3pm, Monday & Wednesday
- Friday was drop deadline. If you forgot to drop, we can't help you!
 - You need to speak with advisor services in your department about how to drop
- Recommendation: Read assigned readings *before* lecture
- Group sign up should have happened already
 - If you don't have 4 members in your group, we will try to find you other partners
 - Want everyone in your group to have the same TA
 - Go to your assigned section on Friday, starting this week!
- Midterm 1 conflicts
 - Watch for announcements on EdStem (remember: MT1 is 2/16)

Kubiatowicz CS162 © UCB Spring 2023

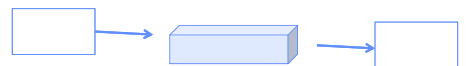
Lec 5.26

1/31/2023

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!

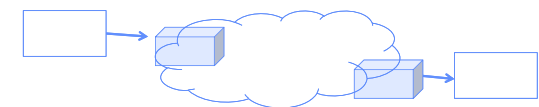
Kubiatowicz CS162 © UCB Spring 2023

Lec 5.27

1/31/2023

Communication Across the world looks like file IO!

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



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

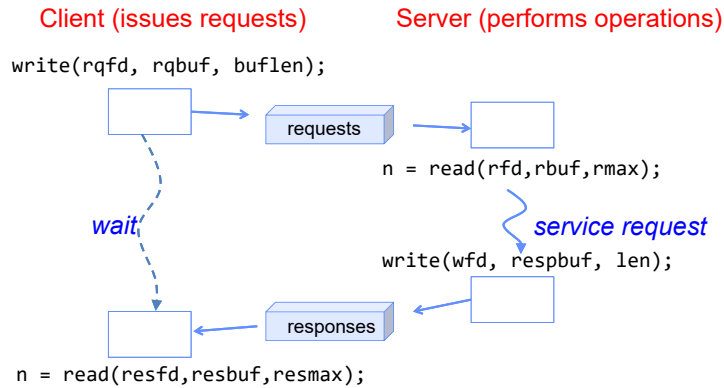
- Connected queues over the Internet
 - But what's the analog of open?
 - What is the namespace?
 - How are they connected in time?

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.28

1/31/2023

Request Response Protocol

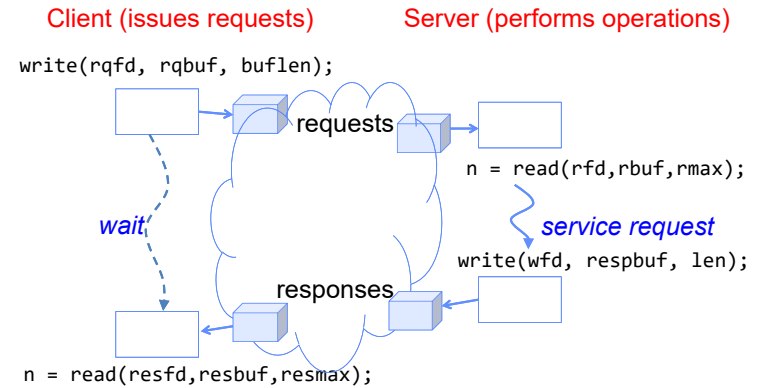


1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.29

Request Response Protocol: Across Network



1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.30

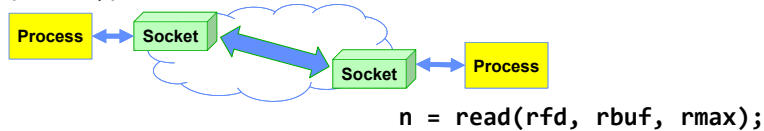
The Socket Abstraction: Endpoint for Communication

- **Key Idea:** Communication across the world looks like File I/O

```

write(wfd, wbuf, wlen);

```



- Sockets: Endpoint for Communication
 - Queues to temporarily hold results
- Connection: Two Sockets Connected Over the network ⇒ IPC over network!
 - How to **open()**?
 - What is the namespace?
 - How are they connected in time?

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.31

Sockets: More Details

- **Socket:** An abstraction for one endpoint of a network connection
 - Another mechanism for **inter-process communication**
 - Most operating systems (Linux, Mac OS X, Windows) provide this, even if they don't copy rest of UNIX I/O
 - Standardized by POSIX
- First introduced in 4.2 BSD (Berkeley Standard Distribution) Unix
 - This release had some huge benefits (and excitement from potential users)
 - Runners waiting at release time to get release on tape and take to businesses
- Same abstraction for any kind of network
 - Local (within same machine)
 - The Internet (TCP/IP, UDP/IP)
 - Things "no one" uses anymore (OSI, Appletalk, IPX, ...)

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.32

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.
- 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
 - » Actually need to loop and read until we receive the terminator ('\0')
 - Assumes writing already took place
 - Blocks if nothing has arrived yet

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.37

Socket Creation

- File systems provide a collection of permanent objects in a structured name space:
 - Processes open, read/write/close them
 - Files exist independently of processes
 - Easy to name what file to open()
- Pipes: one-way communication between processes on same (physical) machine
 - Single queue
 - Created transiently by a call to `pipe()`
 - Passed from parent to children (descriptors inherited from parent process)
- Sockets: two-way communication between processes on same or different machine
 - Two queues (one in each direction)
 - Processes can be on separate machines: no common ancestor
 - How do we *name* the objects we are opening?
 - How do these completely independent programs know that the other wants to “talk” to them?

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.38

Namespaces for Communication over IP

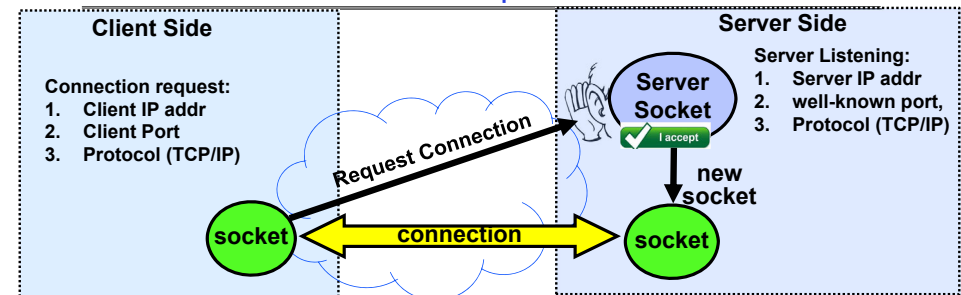
- Hostname
 - `www.eecs.berkeley.edu`
- IP address
 - `128.32.244.172` (IPv4, 32-bit Integer)
 - `2607:f140:0:81::f` (IPv6, 128-bit Integer)
- Port Number
 - 0-1023 are “well known” or “system” ports
 - » Superuser privileges to bind to one
 - 1024 – 49151 are “registered” ports (*registry*)
 - » Assigned by IANA for specific services
 - 49152–65535 ($2^{15}+2^{14}$ to $2^{16}-1$) are “dynamic” or “private”
 - » Automatically allocated as “ephemeral ports”

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.39

Connection 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

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

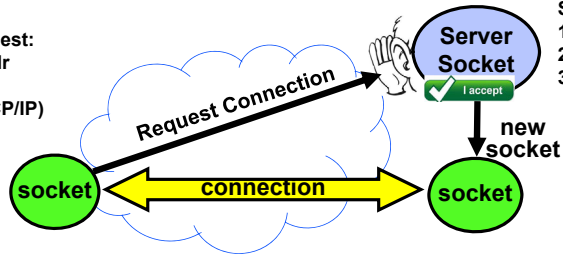
Lec 5.40

Connection Setup over TCP/IP

Client Side

Connection request:

1. Client IP addr
2. Client Port
3. Protocol (TCP/IP)



Server Side

Server Listening:

1. Server IP addr
2. well-known port,
3. Protocol (TCP/IP)

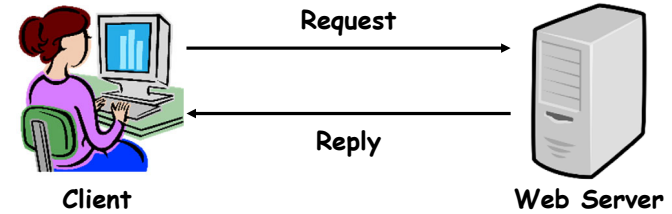
- 5-Tuple identifies each connection:
 1. Source IP Address
 2. Destination IP Address
 3. Source Port Number
 4. Destination Port Number
 5. Protocol (always TCP here)
- Often, Client Port “randomly” assigned
 - Done by OS during client socket setup
- Server Port often “well known”
 - 80 (web), 443 (secure web), 25 (sendmail), etc
 - Well-known ports from 0—1023

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.41

Web Server

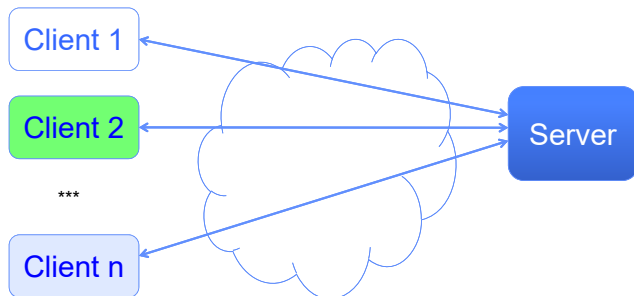


1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.42

Client-Server Models



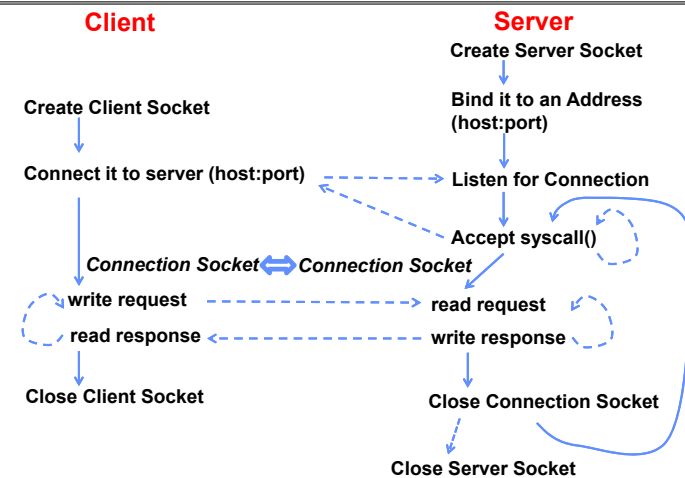
- File servers, web, FTP, Databases, ...
- Many clients accessing a common server

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.43

Sockets in concept



1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.44

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);
```

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.45

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;      /* Includes AF_INET and AF_INET6 */
    hints.ai_socktype = SOCK_STREAM; /* Essentially TCP/IP */

    int rv = getaddrinfo(host_name, port_name,
                        &hints, &server);

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

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.46

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);
```

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.47

Server Address: Itself (wildcard IP), Passive

```
struct addrinfo *setup_address(char *port) {
    struct addrinfo *server;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;      /* Includes AF_INET and AF_INET6 */
    hints.ai_socktype = SOCK_STREAM; /* Essentially TCP/IP */
    hints.ai_flags = AI_PASSIVE;     /* Set up for server socket */

    int rv = getaddrinfo(NULL, port, &hints, &server); /* No address! (any local IP) */
    if (rv != 0) {
        printf("getaddrinfo failed: %s\n", gai_strerror(rv));
        return NULL;
    }
    return server;
}
```

- Accepts any connections on the specified port

1/31/2023

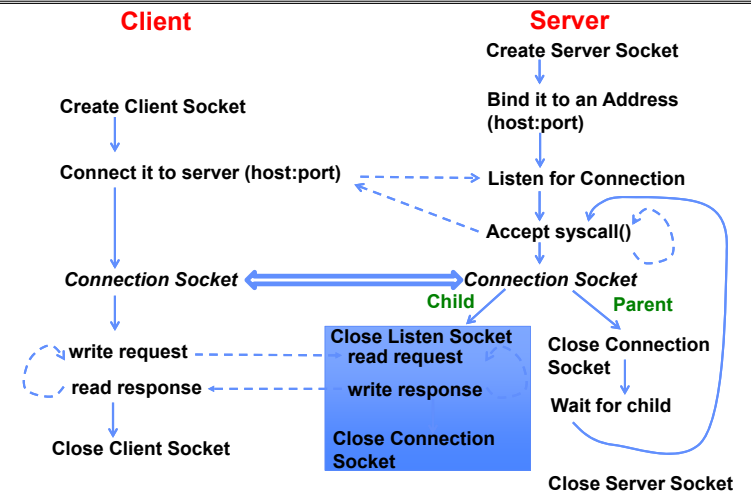
Kubiatowicz CS162 © UCB Spring 2023

Lec 5.48

How Could the Server Protect Itself?

- Handle each connection in a separate process
 - This will mean that the logic serving each request will be “sandboxed” away from the main server process

Sockets With Protection (each connection has own process)



Server Protocol (v2)

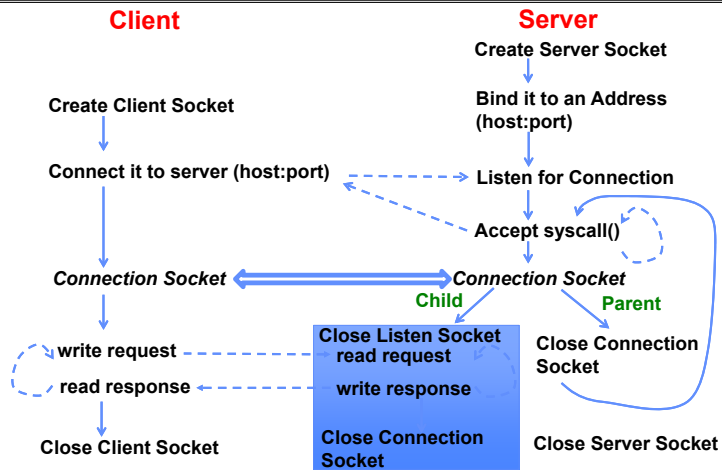
```

// Socket setup code elided...
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();
    if (pid == 0) {
        close(server_socket);
        serve_client(conn_socket);
        close(conn_socket);
        exit(0);
    } else {
        close(conn_socket);
        wait(NULL);
    }
}
close(server_socket);
    
```

Concurrent Server

- So far, in the server:
 - Listen will queue requests
 - Buffering present elsewhere
 - But server waits for each connection to terminate before servicing the next
- A concurrent server can handle and service a new connection before the previous client disconnects

Sockets With Protection and Concurrency



1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.53

Server Protocol (v3)

```
// Socket setup code elided...
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();
    if (pid == 0) {
        close(server_socket);
        serve_client(conn_socket);
        close(conn_socket);
        exit(0);
    } else {
        close(conn_socket);
        //wait(NULL);
    }
}
close(server_socket);
```

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.54

Concurrent Server without Protection

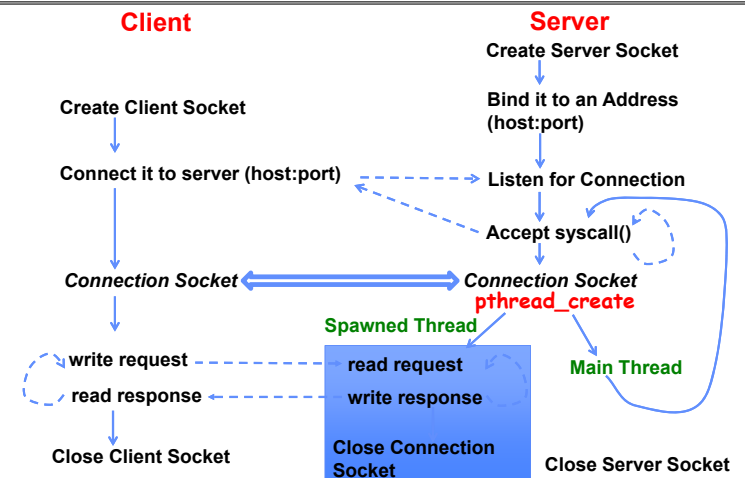
- Spawn a new thread to handle each connection
- Main thread initiates new client connections without waiting for previously spawned threads
- Why give up the protection of separate processes?
 - More efficient to create new threads
 - More efficient to switch between threads

1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.55

Sockets with Concurrency, without Protection



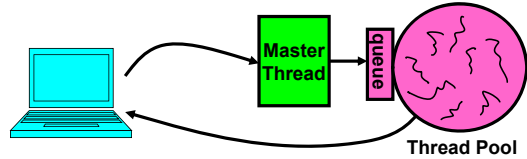
1/31/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 5.56

Thread Pools: More Later!

- Problem with previous version: Unbounded Threads
 - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded “pool” of worker threads, representing the maximum level of multiprogramming



```
master() {  
    allocThreads(worker, queue);  
    while(TRUE) {  
        con=AcceptCon();  
        Enqueue(queue, con);  
        wakeUp(queue);  
    }  
}  
  
worker(queue) {  
    while(TRUE) {  
        con=Dequeue(queue);  
        if (con==null)  
            sleepOn(queue);  
        else  
            ServiceWebPage(con);  
    }  
}
```

Conclusion

- 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 (IPC mechanism)
- Processes have two parts
 - One or more Threads (Concurrency)
 - Address Spaces (Protection)
- Stack is an essential part of a computation stream
 - Every thread has a stack!