

CS162 Operating Systems and Systems Programming Lecture 4

Introduction to I/O, Sockets, Networking

January 31st, 2019
Prof. John Kubiatowicz
<http://cs162.eecs.Berkeley.edu>

Recall: Fork, Wait, and (optional) Exec

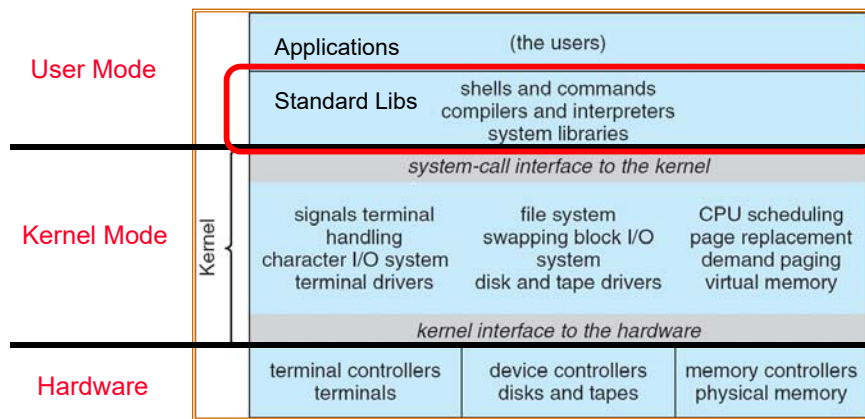
```

cpid = fork();
if (cpid > 0) { // Parent Process
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
    tcpid = wait(&status);
    printf("[%d] bye %d\n", mypid, tcpid);
} else if (cpid == 0) { // Child Process
    mypid = getpid();
    printf("[%d] child\n", mypid);
    execl(filename, (char *)0); // Opt: start new program
} else { // Error! }

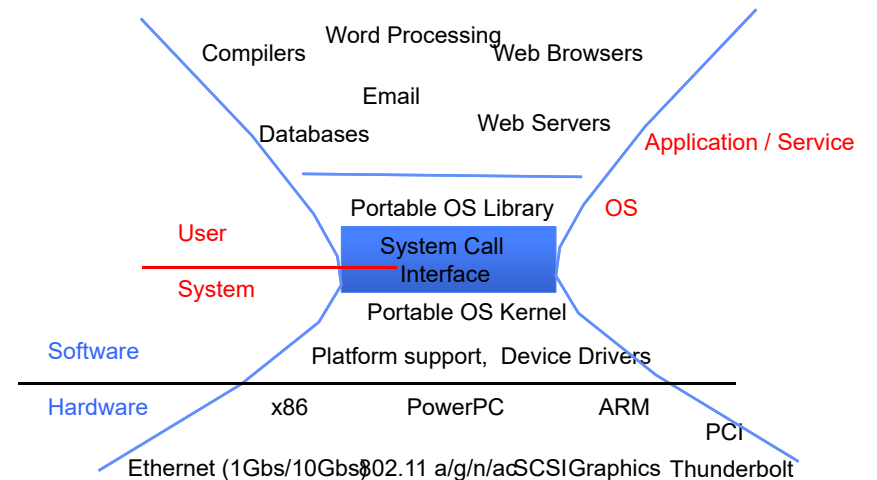
```

- Return value from Fork: integer
 - When > 0: return value is pid of new child (Running in **Parent**)
 - When = 0: Running in new **Child** process
 - When < 0: Error! Must handle somehow
- Wait() system call: wait for next child to exit
 - Return value is PID of terminating child
 - Argument is pointer to integer variable to hold exit status
- Exec() family of calls: replace process with new executable

Recall: UNIX System Structure



A Kind of Narrow Waist



Key Unix I/O Design Concepts

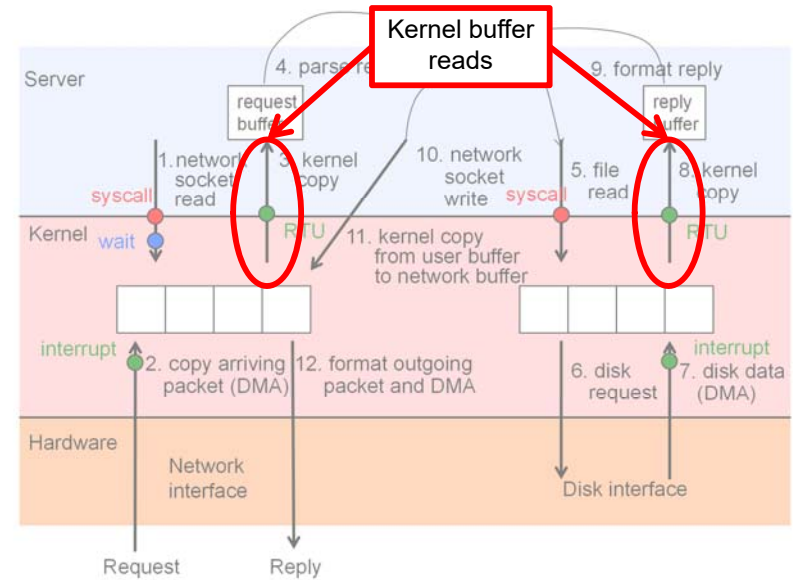
- **Uniformity**
 - file operations, device I/O, and interprocess communication through **open**, **read/write**, **close**
 - Allows simple composition of programs
 - » find | grep | wc ...
- **Open before use**
 - Provides opportunity for access control and arbitration
 - Sets up the underlying machinery, i.e., data structures
- **Byte-oriented**
 - Even if blocks are transferred, addressing is in bytes
- **Kernel buffered reads**
 - Streaming and block devices looks the same
 - read blocks process, yielding processor to other task

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.5

Putting it together: web server



1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.6

Key Unix I/O Design Concepts

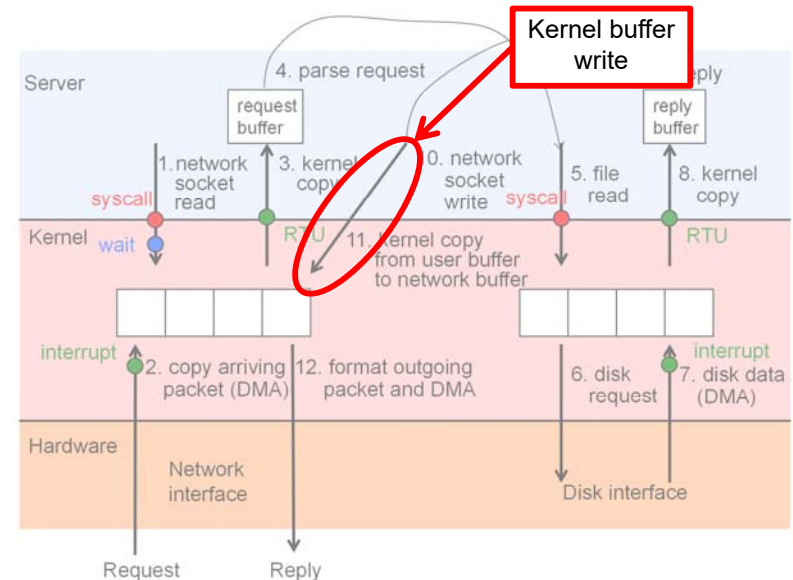
- **Uniformity**
 - file operations, device I/O, and interprocess communication through **open**, **read/write**, **close**
 - Allows simple composition of programs
 - » find | grep | wc ...
- **Open before use**
 - Provides opportunity for access control and arbitration
 - Sets up the underlying machinery, i.e., data structures
- **Byte-oriented**
 - Even if blocks are transferred, addressing is in bytes
- **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

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.7

Putting it together: web server



1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.8

Key Unix I/O Design Concepts

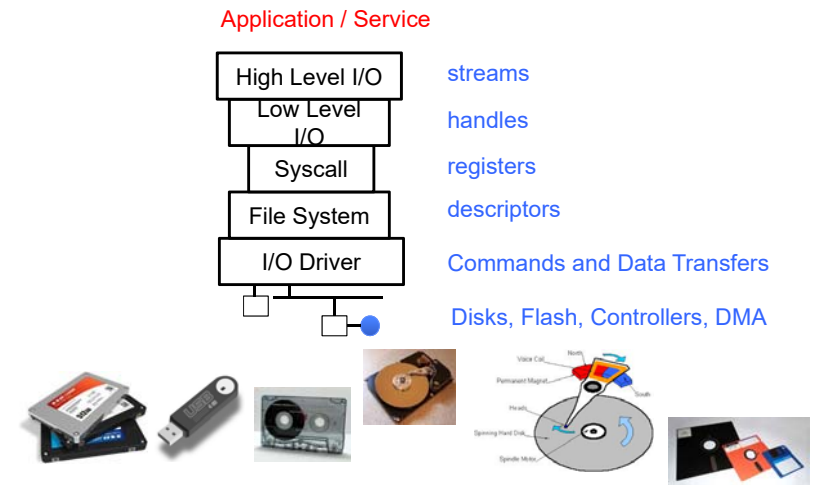
- **Uniformity**
 - file operations, device I/O, and interprocess communication through **open**, **read/write**, **close**
 - Allows simple composition of programs
 - » find | grep | wc ...
- **Open before use**
 - Provides opportunity for access control and arbitration
 - Sets up the underlying machinery, i.e., data structures
- **Byte-oriented**
 - Even if blocks are transferred, addressing is in bytes
- **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**

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.9

I/O & Storage Layers



1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.10

The File System Abstraction

- **High-level idea**
 - Files live in hierarchical namespace of filenames
- **File**
 - Named collection of data in a file system
 - File data
 - » 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
 - Hierarchical (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)

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.11

Recall: 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

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.12

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

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.13

C API Standard Streams

- 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
 - Recall: Use of pipe symbols connects STDOUT and STDIN
 - » find | grep | wc ...

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.14

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 );
```

DESCRIPTION

The fgets() function reads at most one less than the number of characters specified by size from the given stream and stores them in the string str. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is retained. If any characters are read and there is no error, a '\0' character is appended to end the string.

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.15

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);
```

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.16

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,
            ...);
```

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.17

Example Code

```
#include <stdio.h>
#define BUFLen 256
FILE *outfile;
char mybuf[BUFLen];

int storetofile() {
    char *instring;

    outfile = fopen("/usr/homes/testing/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
}
```

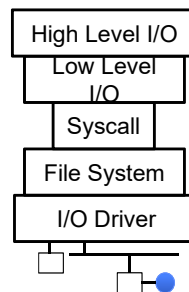
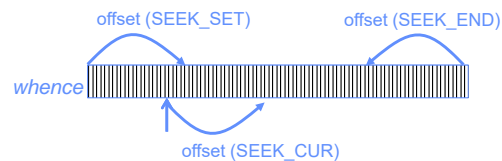
1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.18

C Stream API positioning

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



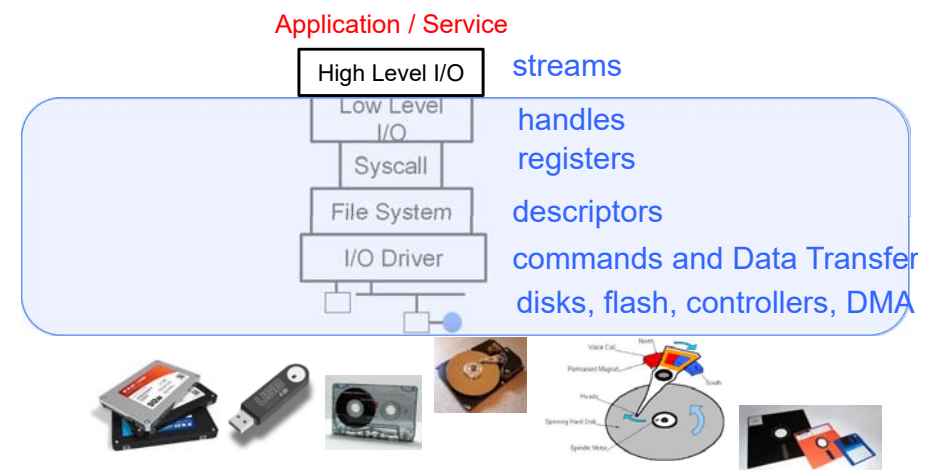
- Preserves high level abstraction of uniform stream of objects
- Adds buffering for performance

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.19

What's below the surface ??



1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.20

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 filedes) - 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)
```


Another example: 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 = sprintf(buf, BUFSIZE, "Got %zd chars\n", readlen);

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

    exit(0);
}
```

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.25

Administrivia

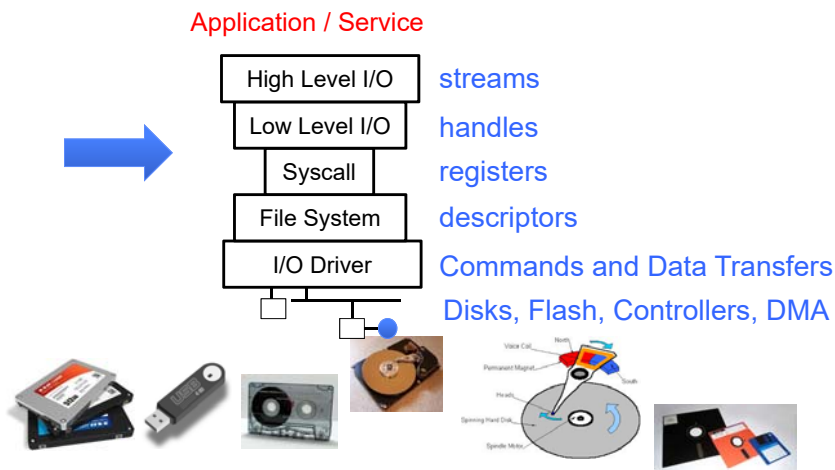
- Kubiatowicz Office Hours:
 - 1-2pm, Monday & Thursday
- No one left on WaitList! Everyone left is in the class.
 - Could take a few concurrent enrollment students: know any?
- **TOMORROW (Friday) is Drop Deadline!**
VERY HARD TO DROP LATER!
- Recommendation: Read assigned readings *before* lecture
- You should be going to sections now – Important information covered in section
 - Any section will do until groups assigned
- Group sign up via autograder form next week
 - Get finding groups of 4 people ASAP
 - Priority for same section; if cannot make this work, keep same TA
 - Remember: Your TA needs to see you in section!
- Midterm 1 conflicts
 - We will handle these conflicts next week

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.26

What's below the surface ??



1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.27

Recall: SYSCALL

#	Name	Registers	Definition
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

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

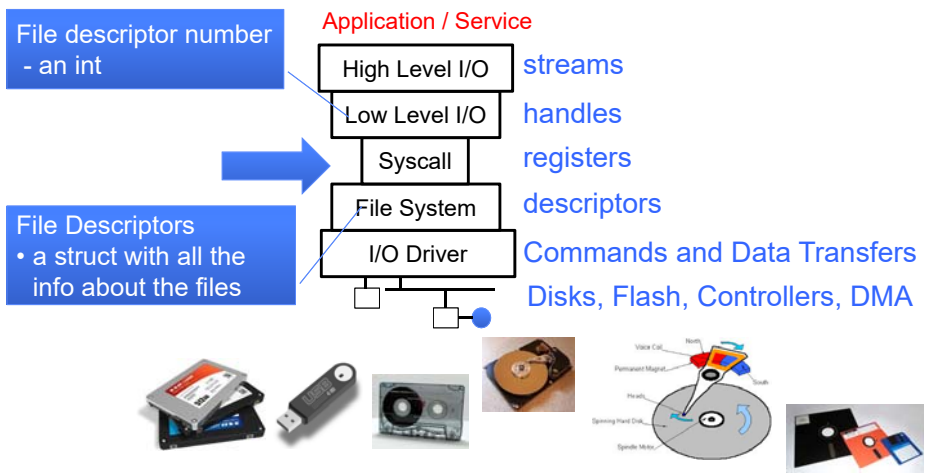
- Low level lib parameters are set up in registers and syscall instruction is issued
 - A type of synchronous exception that enters well-defined entry points into kernel

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.28

What's below the surface ??



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`

```

747 struct file {
748     union {
749         struct list_node    f_u_list;
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 /* needed for tty driver, and maybe others */
775     void                       *private_data;
776
777 #ifdef CONFIG_EPOLL
778     /* Used by fs/eventpoll.c to link all the hooks
779     struct list_head           f_ep_links;
780     struct list_head           f_file_links;
781 #endif /* #ifdef CONFIG_EPOLL */
782     struct address_space      *f_mapping;
783 } __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 && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
        return -EIO;
    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)))
        return -EIO;
    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

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.33

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

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.34

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.

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.35

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()

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.36

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>)

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.37

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)

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.38

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)

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.39

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

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.40

Device Drivers

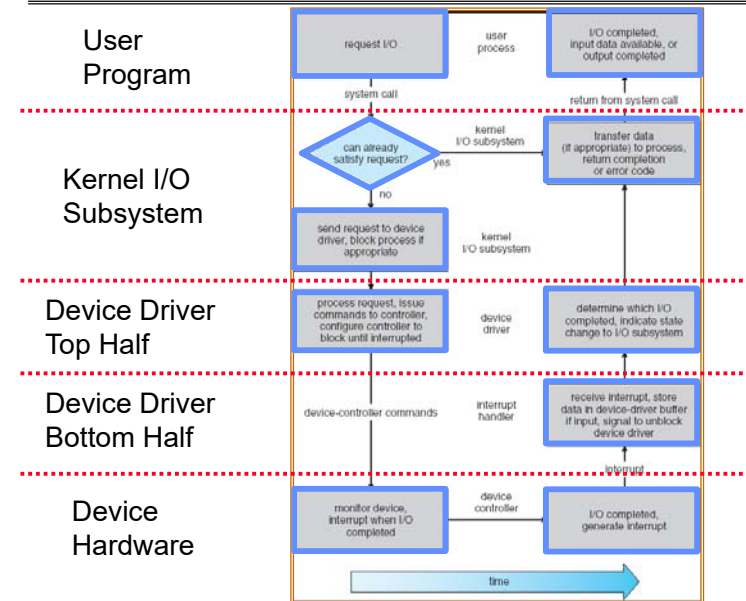
- **Device Driver:** Device-specific code in the kernel that interacts directly with the device hardware
 - Supports a standard, internal interface
 - Same kernel I/O system can interact easily with different device drivers
 - Special device-specific configuration supported with the `ioctl()` system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

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.41

Life Cycle of An I/O Request



1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.42

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!

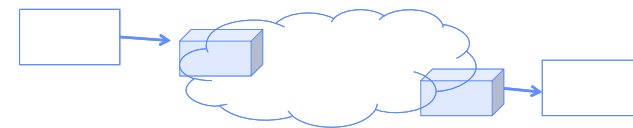
1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.43

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?

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

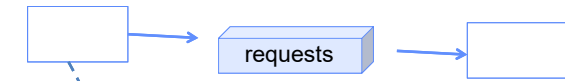
Lec 4.44

Request Response Protocol

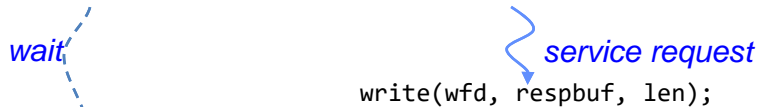
Client (issues requests)

Server (performs operations)

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



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



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



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

wait

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.45

Request Response Protocol

Client (issues requests)

Server (performs operations)

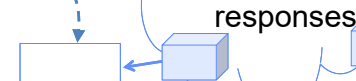
```
write(rqfd, rqbuf, buflen);
```



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



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



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

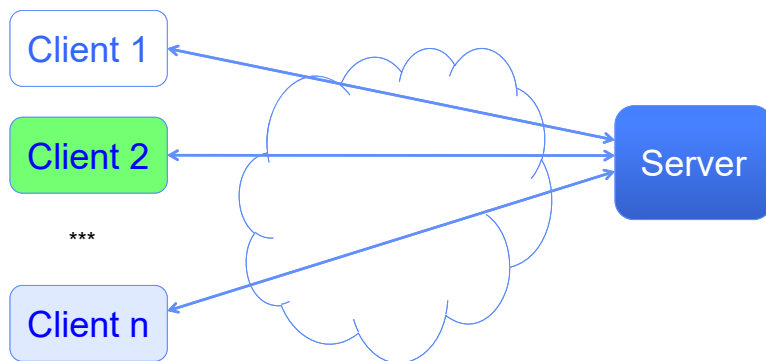
wait

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.46

Client-Server Models



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

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.47

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, ...

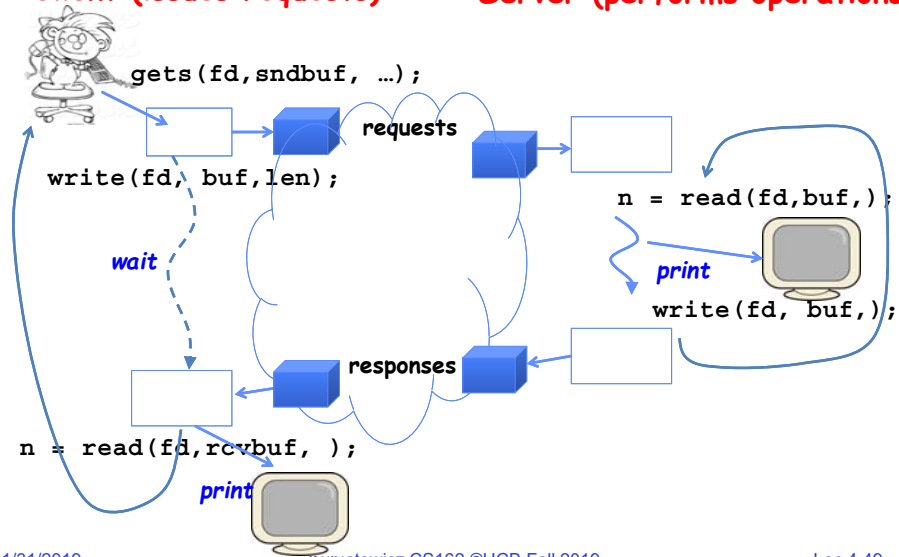
1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.48

Silly Echo Server – running example

Client (issues requests) **Server (performs operations)**



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 */
    }
}
```

1/31/2019

Kubiawicz CS162 ©UCB Fall 2019

Lec 4.50

Prompt for input

```
char *getreq(char *inbuf, int len) {
    /* Get request char stream */
    printf("REQ: "); /* prompt */
    memset(inbuf, 0, len); /* clear for good measure */
    return fgets(inbuf, len, stdin); /* read up to a EOL */
}
```

1/31/2019

Kubiawicz CS162 ©UCB Fall 2019

Lec 4.51

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

1/31/2019

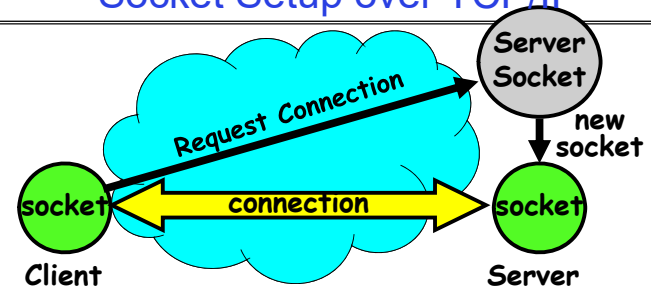
Kubiawicz CS162 ©UCB Fall 2019

Lec 4.52

Namespaces for communication over IP

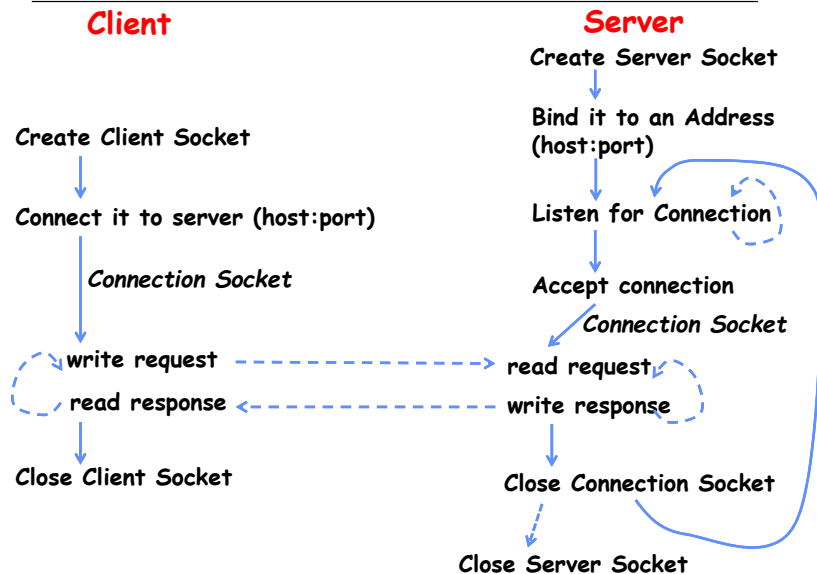
- Hostname
 - www.eecs.berkeley.edu
- IP address
 - 128.32.244.172 (ipv6?)
- Port Number
 - 0-1023 are “well known” or “system” ports
 - » Superuser privileges to bind to one
 - 1024 – 49151 are “registered” ports (registry)
 - » Assigned by IANA for specific services
 - 49152–65535 ($2^{15}+2^{14}$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

Socket Setup over TCP/IP



- Server Socket: Listens for new connections
 - Produces new sockets for each unique 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

Sockets in concept



Client Protocol

```

char *hostname;
int sockfd, portno;
struct sockaddr_in serv_addr;
struct hostent *server;

server = buildServerAddr(&serv_addr, hostname, portno);

/* Create a TCP socket */
sockfd = socket(AF_INET, SOCK_STREAM, 0)

/* Connect to server on port */
connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))
printf("Connected to %s:%d\n", server->h_name, portno);

/* Carry out Client-Server protocol */
client(sockfd);

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


Server Protocol (v1)

```

/* Create Socket to receive requests*/
ltnsockfd = socket(AF_INET, SOCK_STREAM, 0);

/* Bind socket to port */
bind(ltnsockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
while (1) {
/* Listen for incoming connections */
listen(ltnsockfd, MAXQUEUE);

/* Accept incoming connection, obtaining a new socket for it */
consockfd = accept(ltnsockfd, (struct sockaddr *) &cli_addr,
                  &clilen);

server(consockfd);

close(consockfd);
}
close(ltnsockfd);

```

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.57

How does the server protect itself?

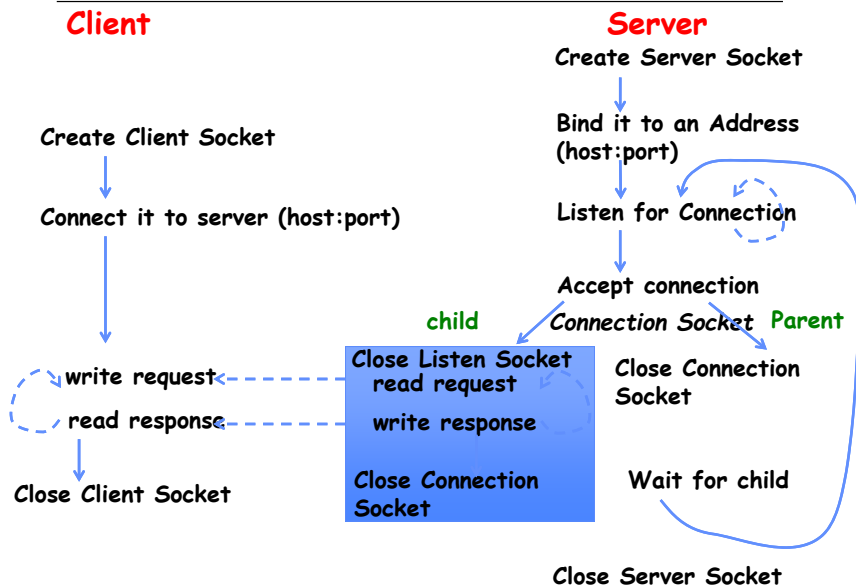
- Isolate the handling of each connection
- By forking it off as another process

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.58

Sockets With Protection



1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.59

Server Protocol (v2)

```

while (1) {
listen(ltnsockfd, MAXQUEUE);
consockfd = accept(ltnsockfd, (struct sockaddr *) &cli_addr,
                  &clilen);

cpid = fork(); /* new process for connection */
if (cpid > 0) { /* parent process */
close(consockfd);
tcpid = wait(&cstatus);
} else if (cpid == 0) { /* child process */
close(ltnsockfd); /* let go of listen socket */

server(consockfd);

close(consockfd);
exit(EXIT_SUCCESS); /* exit child normally */
}
}
close(ltnsockfd);

```

1/31/2019

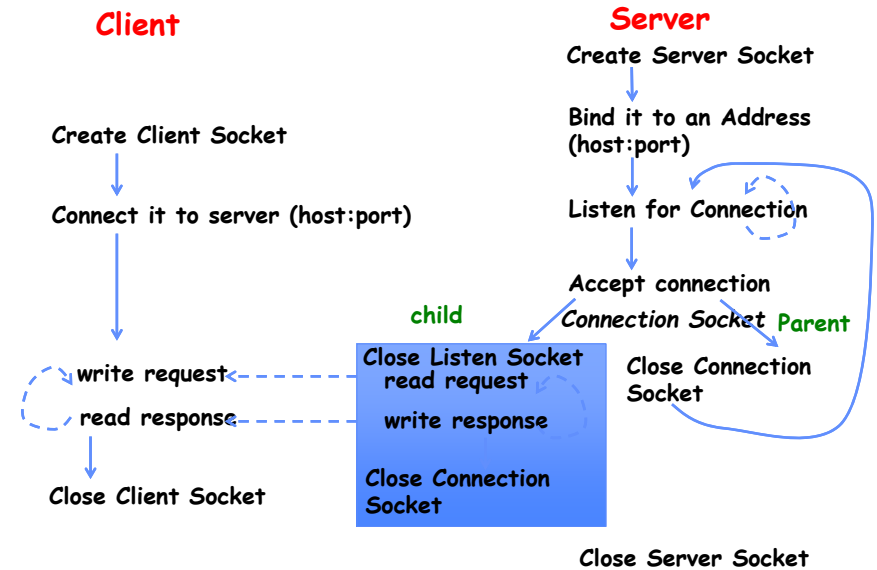
Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.60

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



Server Protocol (v3)

```
while (1) {
    listen(lstnsockfd, MAXQUEUE);
    consockfd = accept(lstnsockfd, (struct sockaddr *) &cli_addr,
                      &clilen);

    cpid = fork();          /* new process for connection */
    if (cpid > 0) {        /* parent process */
        close(consockfd);
        //tcpid = wait(&cstatus);
    } else if (cpid == 0) { /* child process */
        close(lstnsockfd); /* let go of listen socket */

        server(consockfd);

        close(consockfd);
        exit(EXIT_SUCCESS); /* exit child normally */
    }
}
close(lstnsockfd);
```

Server Address - itself

```
memset((char *) &serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);
```

- Simple form
- Internet Protocol
- accepting any connections on the specified port
- In “network byte ordering”

Client: getting the server address

```
struct hostent *buildServerAddr(struct sockaddr_in *serv_addr,
                               char *hostname, int portno) {
    struct hostent *server;

    /* Get host entry associated with a hostname or IP address */
    server = gethostbyname(hostname);
    if (server == NULL) {
        fprintf(stderr, "ERROR, no such host\n");
        exit(1);
    }

    /* Construct an address for remote server */
    memset((char *) serv_addr, 0, sizeof(struct sockaddr_in));
    serv_addr->sin_family = AF_INET;
    bcopy((char *)server->h_addr,
          (char *)&(serv_addr->sin_addr.s_addr), server->h_length);
    serv_addr->sin_port = htons(portno);

    return server;
}
```

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.65

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 ...

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.66

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

1/31/2019

Kubiatowicz CS162 ©UCB Fall 2019

Lec 4.67