CS162
Operating Systems and Systems Programming
Lecture 4

Introduction to I/O, Sockets, Networking

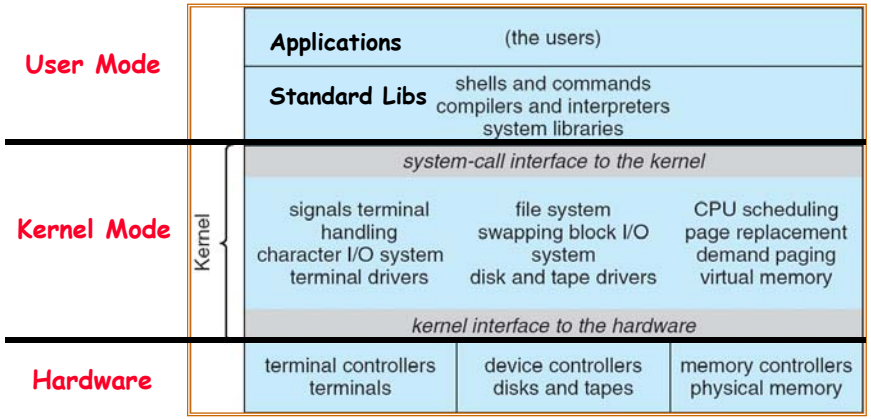
September 9th, 2015
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Recall: Fork and Wait

```
…
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);
}
…
```

- 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

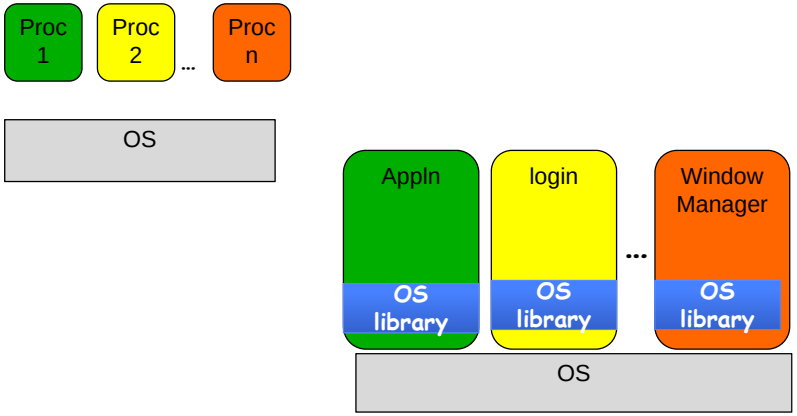
Recall: UNIX System Structure



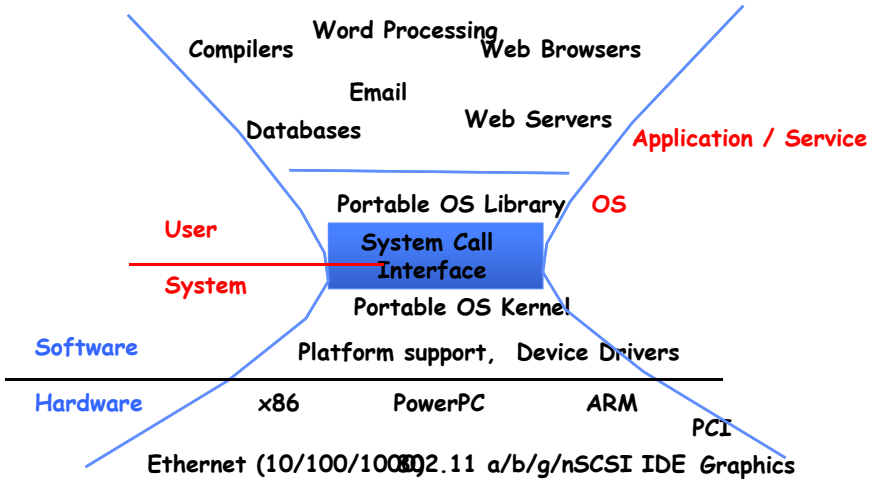
How does the kernel provide services?

- You said that applications request services from the operating system via *syscall*, but …
- I've been writing all sort of useful applications and I never ever saw a "syscall" !!!
- That's right.
- It was buried in the programming language runtime library (e.g., libc.a)
- … Layering

OS run-time library



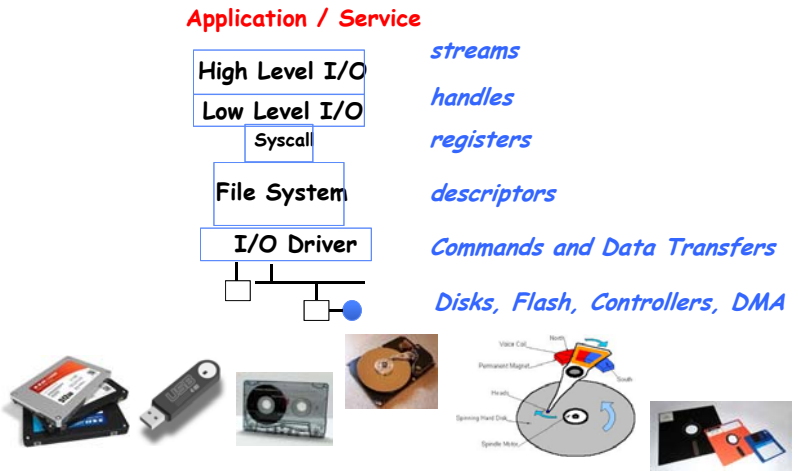
A Kind of Narrow Waist



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
- Kernel buffered writes
  - Completion of out-going transfer decoupled from the application, allowing it to continue
- Explicit close

I/O & Storage Layers



The file system abstraction

- 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
  - Hierachical (graphical) naming
    - » Path through the directory graph
    - » Uniquely identifies a file or directory
      - /home/ff/cs162/public_html/fa14/index.html
  - Links and Volumes (later)

C high level File API – streams (review)

- 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/cs152
- Relative paths
  - index.html, ./index.html - current WD
  - ../index.html - parent of current WD
  - ~, ~cs152 - home directory

C API Standard Streams

- Three predefined streams are opened implicitly when the 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 …

C high level File API – stream ops

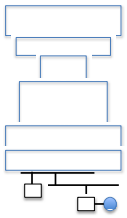
```
#include <stdio.h>
// character oriented
int fputc( int c, FILE *fp );           // rtn c or EOF on err
int fputs( const char *s, FILE *fp );   // rtn >0 or EOF

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

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

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

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


Example 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
}
```


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

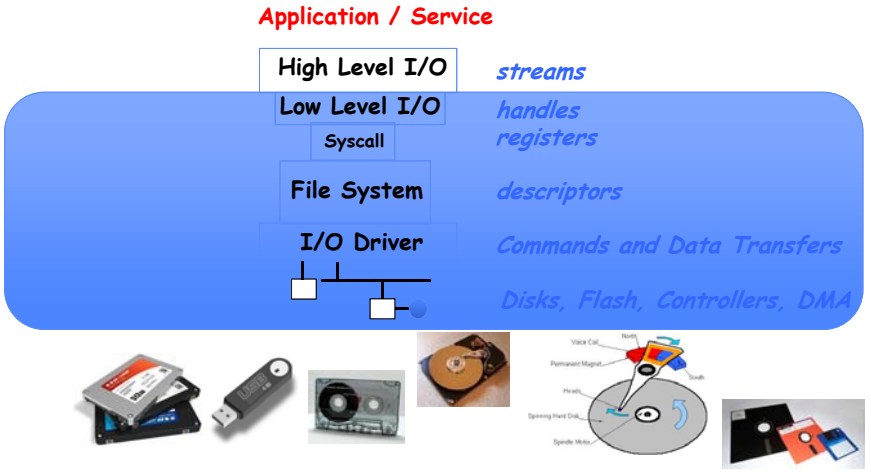
Administrivia: Getting started

- Final class enrollment: ~321
  - Waitlist was emptied
- Homework 1 Due Monday, 9/21
  - Get moving on it!
- Participation: Get to know your TA!
  - We have some seriously under-subscribed sections – consider going to one of them!
  - Under subscribed:
    - » Friday 1-2, 87 Evans
    - » Friday 2-3, 6 Evans
    - » Friday 11-12, B51 Hildebrand
  - OverSubscribed:
    - » Friday 10-11, 385 LeConte
- Group sign up form out this week
  - Get finding groups ASAP
  - 4 people in a group! Same TA! (better – same section)

Administrivia (Con't)

- Kubiatowicz Office Hours (really!)
  - 1pm-2pm, Monday/Wednesday
- Midterm 2 conflict: See Piazza poll!
  - Wednesday 11/18: Conflict with EE16A – evening
  - Monday 11/23: 7-10 pm? Slightly hard on DataScience folks, but all done in one day! (Drown sorrows in turkey?).

What's below the surface ??



C Low level I/O

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

```
#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!

And lots more !

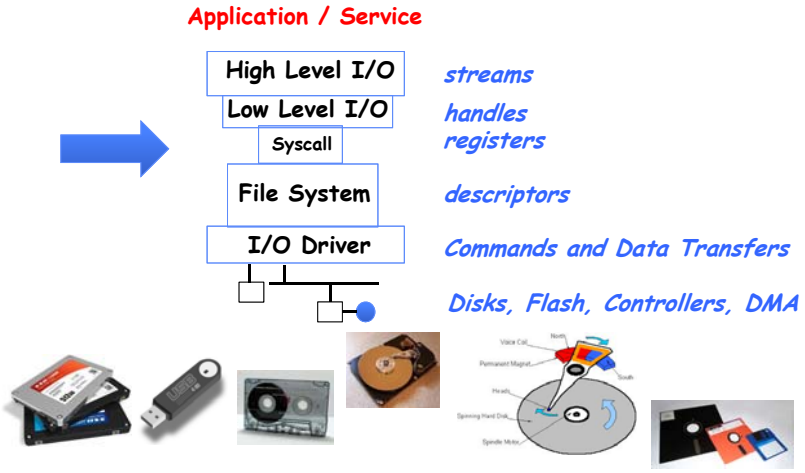
- TTYs versus files
- Memory mapped files
- File Locking
- Asynchronous I/O
- Generic I/O Control Operations
- Duplicating descriptors

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

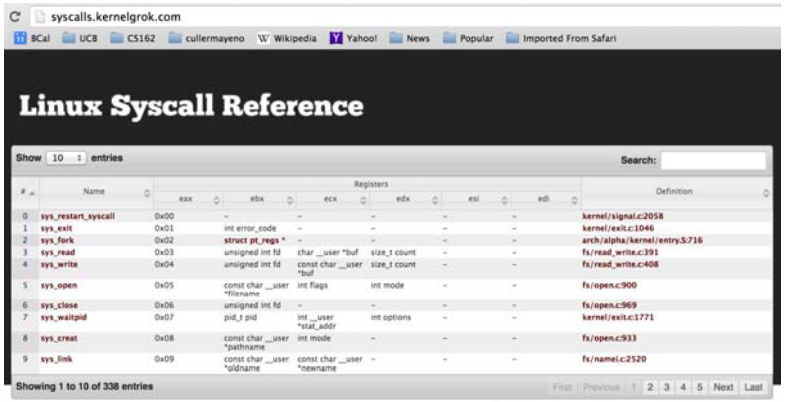

Another example: lowio-std.c

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

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

  exit(0);
}
```


What's below the surface ??

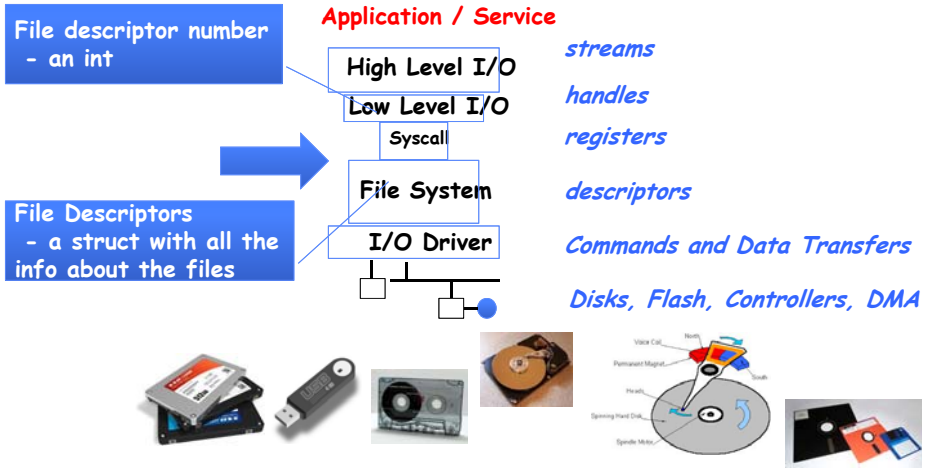


Recall: SYSCALL



- 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 ??



Internal OS File Descriptor

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

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;
}
```

Lower Level Driver

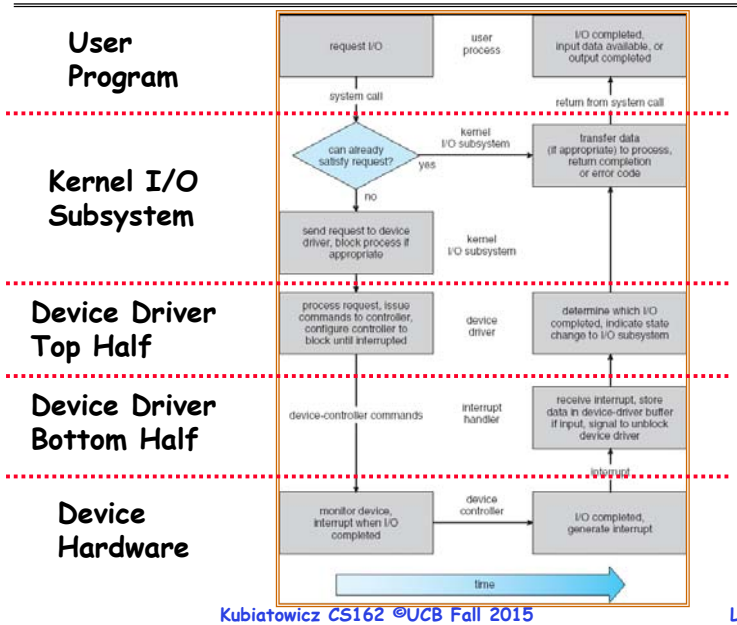
- Associated with particular hardware device
- Registers / Unregisters itself with the kernel
- Handler functions for each of the file operations

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

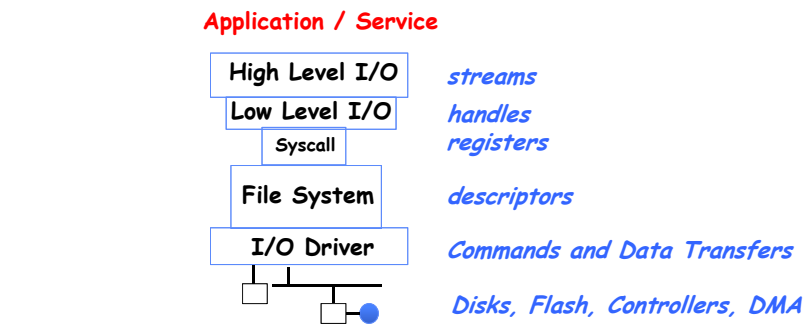

Recall: Device Drivers

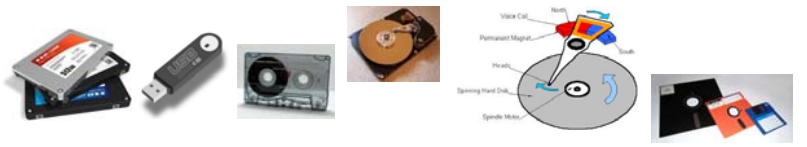
- Device Driver: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call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



So what happens when you fgetc?





Communication between processes

- Can we view files as communication channels?

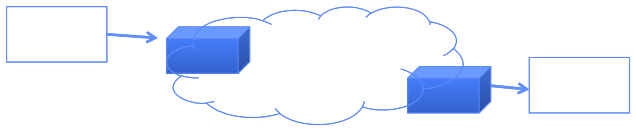
```
write(wfd, wbuf, wlen);
```

```
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);
```



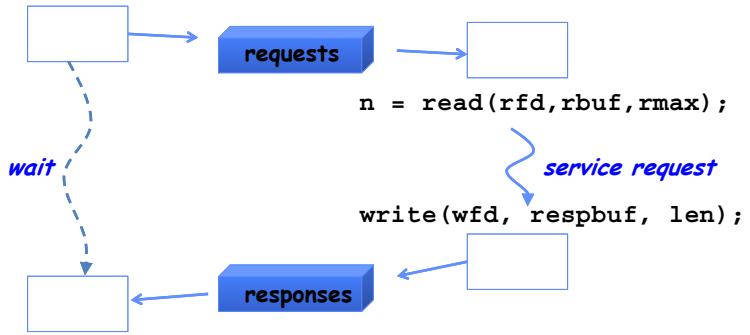
```
n = read(rfd,rbuf,rmax);
```

- Connected queues over the Internet
  - But what's the analog of open?
  - What is the namespace?
  - How are they connected in time?
- Answer these questions in a moment…

Request Response Protocol

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

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

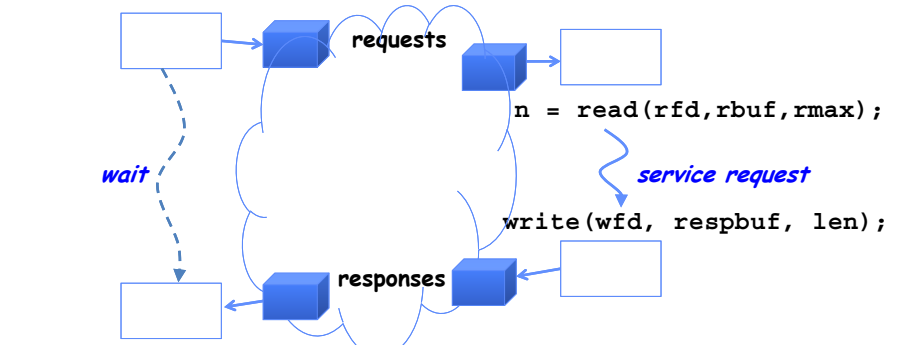



```
n = read(resfd,resbuf,resmax);
```


Request Response Protocol

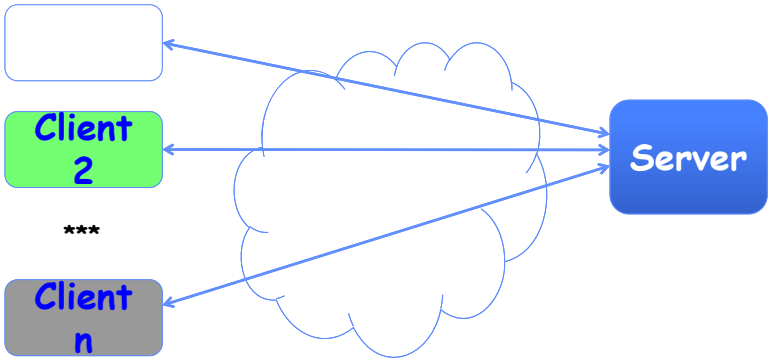
Client (issues requests) **Server (performs operations)**

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




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

Client-Server Models

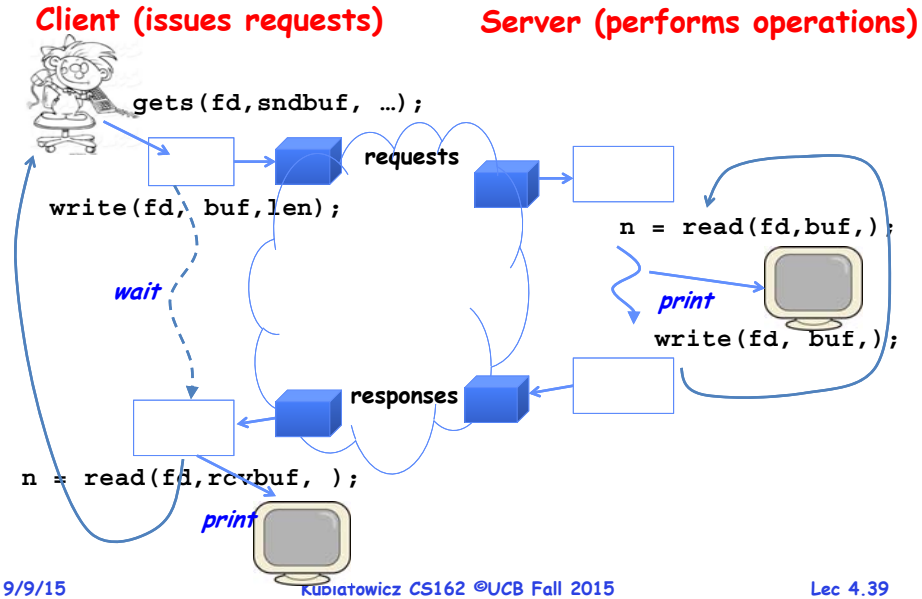


- File servers, web, FTP, Databases, …
- Many clients accessing a common server

Sockets

- Socket: an abstraction of a network I/O queue
  - Mechanism for inter-process communication
  - Embodies one side of a communication channel
    - » Same interface regardless of location of other end
    - » Could be local machine (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



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*/
  }
}
```

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 */
}
```

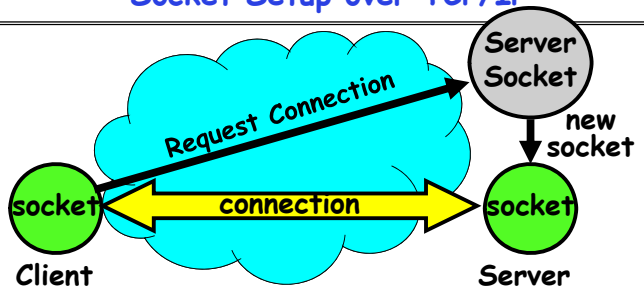

Socket creation and connection

- 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

Namespaces for communication over IP

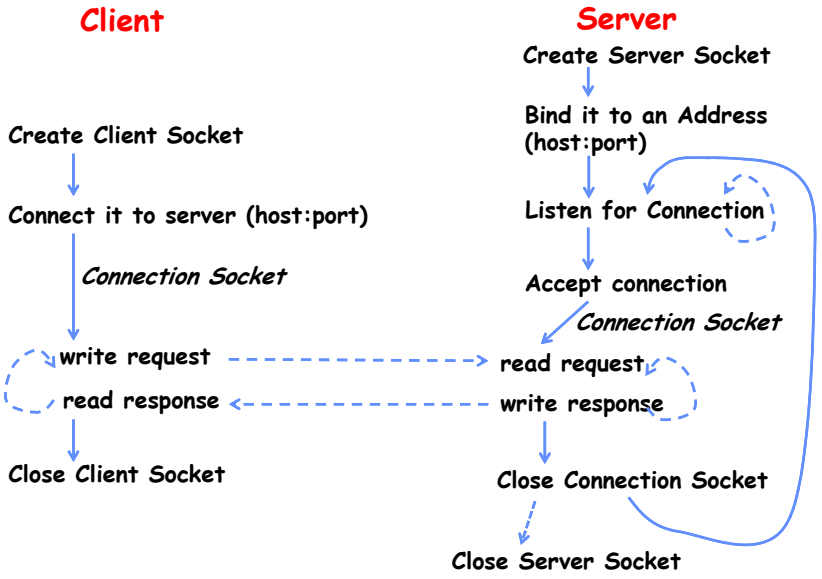
- Hostname
  - www.eecs.berkeley.edu
- IP address
  - 128.32.244.172 (ipv6?)
- Port Number
  - 0-1023 are "well known" or "system" ports
    - » Superuser privileges to bind to one
  - 1024 – 49151 are "registered" ports (registry)
    - » Assigned by IANA for specific services
  - 49152–65535 ($2^{15}+2^{14}$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

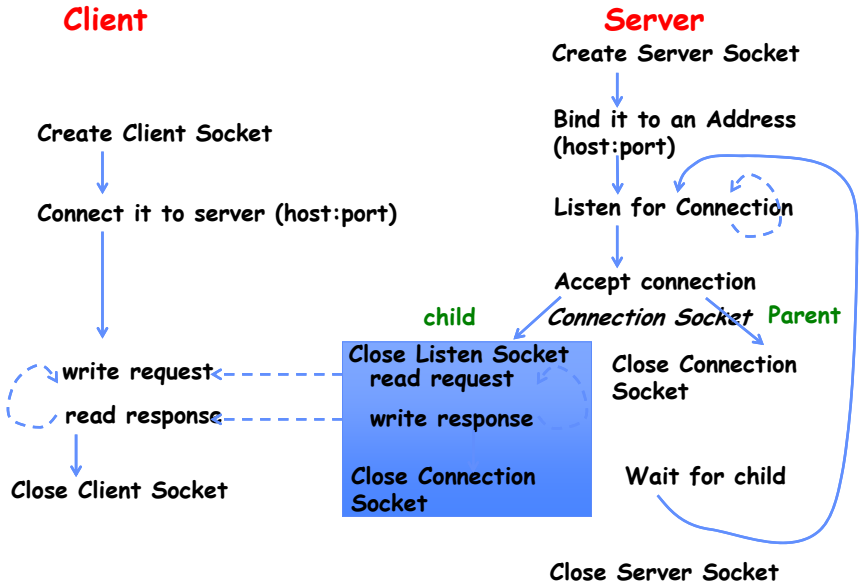

Server Protocol (v1)

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

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

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

   server(consockfd);

   close(consockfd);
  }
close(lstnsockfd);
```


How does the server protect itself?

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

Sockets With Protection



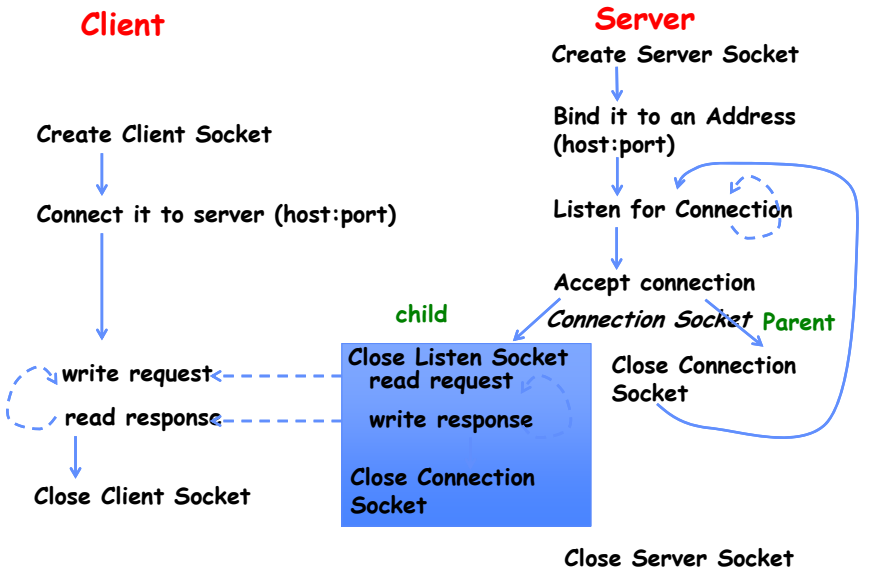
Server Protocol (v2)

```
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);
```


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"

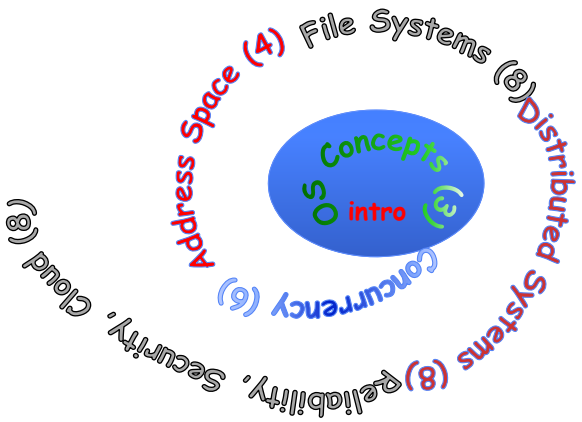
Client: getting the server address

```
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;
}
```


BIG OS Concepts so far

- Processes
- Address Space
- Protection
- Dual Mode
- Interrupt handlers (including syscall and trap)
- File System
  - Integrates processes, users, cwd, protection
- Key Layers: OS Lib, Syscall, Subsystem, Driver
  - User handler on OS descriptors
- Process control
  - fork, wait, signal, exec
- Communication through sockets
- Client-Server Protocol

Course Structure: Spiral



Conclusion

- System Call Interface provides a kind of "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 …
- 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
  - Can work across the Internet
- Socket: an abstraction of a network I/O queue
  - Mechanism for inter-process communication