I/O, Networking, Concurrency

1

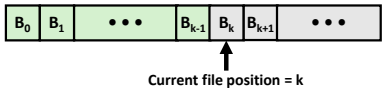
Unix I/O Overview

- A Linux *file* is a sequence of *m* bytes:
  - B_0 , B_1 , ... , B_k , ... , B_{m-1}
- Cool fact: All I/O devices are represented as files:
  - **`/dev/sda2`** (**`/usr`** disk partition)
  - **`/dev/tty2`** (terminal)
- Even the kernel is represented as a file:
  - **`/boot/vmlinuz-3.13.0-55-generic`** (kernel image)
  - **`/proc`** (kernel data structures)

2

Unix I/O Overview

- Elegant mapping of files to devices allows kernel to export simple interface called *Unix I/O:*
  - Opening and closing files
    - **`open()`** and **`close()`**
  - Reading and writing a file
    - **`read()`** and **`write()`**
  - Changing the ***current file position*** (seek)
    - indicates next offset into file to read or write
    - **`lseek()`**



3

File Types

- Each file has a *type* indicating its role in the system
  - *Regular file:* Contains arbitrary data
  - *Directory:* Index for a related group of files
  - *Socket:* For communicating with a process on another machine
- Other file types beyond our scope
  - *Named pipes (FIFOs)*
  - *Symbolic links*
  - *Character and block devices*

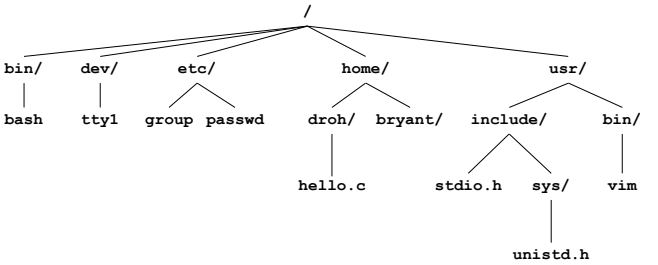
4

Directories

- Directory consists of an array of *links*
  - Each link maps a *filename* to a file
- Each directory contains at least two entries
  - `.` (dot) is a link to itself
  - `..` (dot dot) is a link to *the parent directory* in the *directory hierarchy* (next slide)
- Commands for manipulating directories
  - `mkdir`: create empty directory
  - `ls`: view directory contents
  - `rmdir`: delete empty directory

Directory Hierarchy

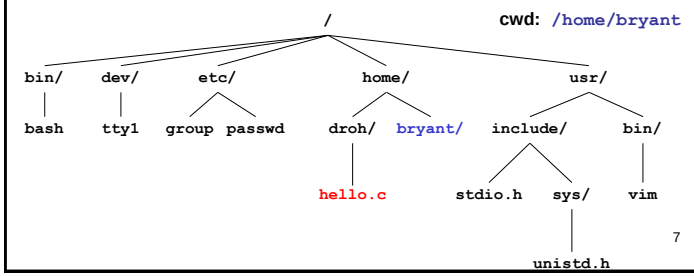
- All files are organized as a hierarchy anchored by root directory named `/` (slash)



- Kernel maintains *current working directory (cwd)* for each process
  - Modified using the `cd` command

Pathnames

- Locations of files in the hierarchy denoted by *pathnames*
  - *Absolute pathname* starts with '/' and denotes path from root
    - `/home/droh/hello.c`
  - *Relative pathname* denotes path from current working directory
    - `../home/droh/hello.c`



Standard I/O Functions

- The C standard library (`libc.so`) contains a collection of higher-level *standard I/O* functions
  - Documented in Appendix B of K&R
- Examples of standard I/O functions:
  - Opening and closing files (**`fopen`** and **`fclose`**)
  - Reading and writing bytes (**`fread`** and **`fwrite`**)
  - Reading and writing text lines (**`fgets`** and **`fputs`**)
  - Formatted reading and writing (**`fscanf`** and **`fprintf`**)

Standard I/O Streams

- Standard I/O models open files as *streams*
  - Abstraction for a file descriptor and a buffer in memory
- C programs begin life with three open streams (defined in `stdio.h`)
  - **`stdin`** (standard input)
  - **`stdout`** (standard output)
  - **`stderr`** (standard error)

```
#include <stdio.h>
extern FILE *stdin;  /* standard input  (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error  (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```


Opening Files

- Opening a file informs the kernel that you are getting ready to access that file

```
int fd;   /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
   perror("open");
   exit(1);
}
```

- Returns a small identifying integer *file descriptor*
  - **`fd == -1`** indicates that an error occurred
- Each process created by a Linux shell begins life with three open files associated with a terminal:
  - 0: standard input (stdin)
  - 1: standard output (stdout)
  - 2: standard error (stderr)

Closing Files

- Closing a file informs the kernel that you are finished accessing that file

```
int fd;     /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
   perror("close");
   exit(1);
}
```


Reading Files

- Reading a file copies bytes from the current file position to memory, and then updates file position

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */

/* Open file fd ...  */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```

-
  - Return type **`ssize_t`** is signed integer
  - **`nbytes < 0`** indicates that an error occurred
  - ***Short counts*** (**`nbytes < sizeof(buf)`**) are possible and are not errors!

Writing Files

- Writing a file copies bytes from memory to the current file position, and then updates current file position

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes written */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

- Returns number of bytes written from `buf` to file `fd`
  - **`nbytes < 0`** indicates that an error occurred
  - As with reads, short counts are possible and are not errors!

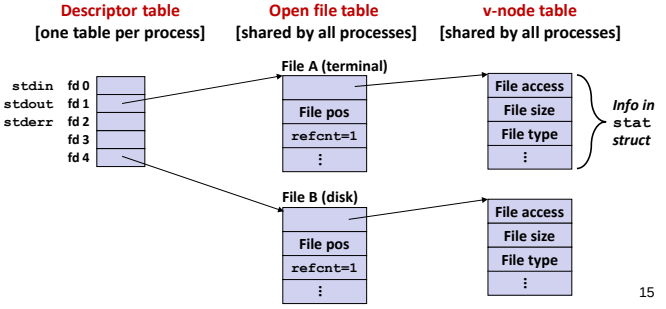
File Metadata

- *Metadata* is data about data, in this case file data
- Per-file metadata maintained by kernel

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;      /* Device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* Protection and file type */
    nlink_t       st_nlink;    /* Number of hard links */
    uid_t         st_uid;      /* User ID of owner */
    gid_t         st_gid;      /* Group ID of owner */
    dev_t         st_rdev;     /* Device type (if inode device) */
    off_t         st_size;     /* Total size, in bytes */
    unsigned long st_blksize;  /* Blocksize for filesystem I/O */
    unsigned long st_blocks;   /* Number of blocks allocated */
    time_t        st_atime;    /* Time of last access */
    time_t        st_mtime;    /* Time of last modification */
    time_t        st_ctime;    /* Time of last change */
};
```

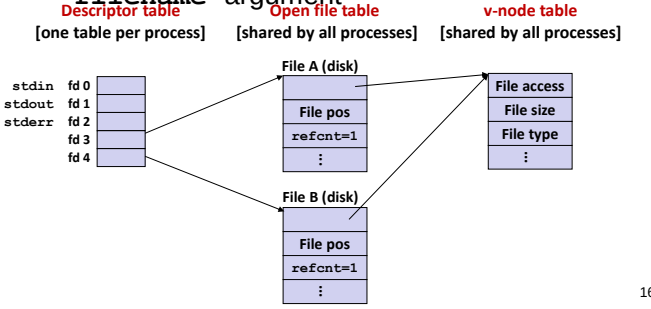

How the Unix Kernel Represents Open Files

- Two descriptors referencing two distinct open files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



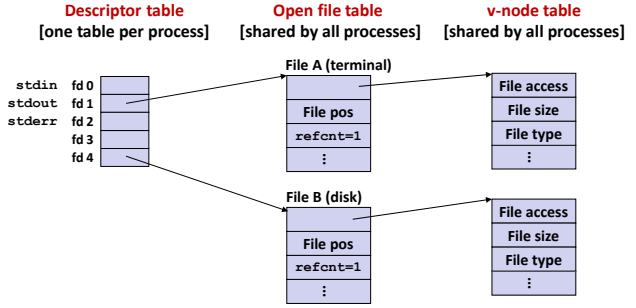
File Sharing

- Two distinct descriptors sharing the same disk file through two distinct open file table entries
  - E.g., Calling **`open`** twice with the same **`filename`** argument



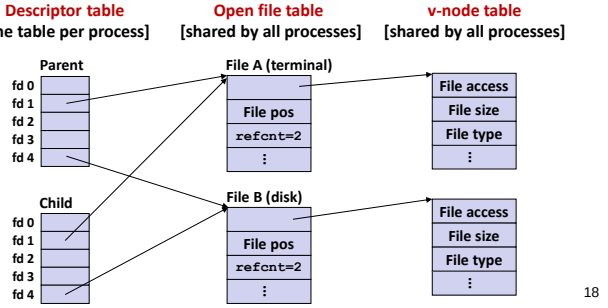
How Processes Share Files: `fork`

- A child process inherits its parent's open files
  - Note: situation unchanged by **`exec`** functions (use **`fcntl`** to change)
- *Before* `fork` call:



How Processes Share Files: `fork`

- A child process inherits its parent's open files
- *After* `fork`:
  - Child's table same as parent's, and +1 to each refcnt

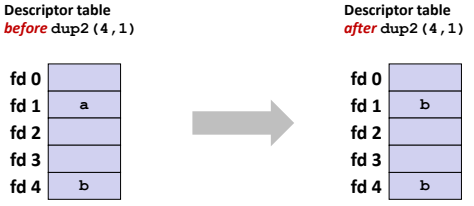


I/O Redirection

- Question: How does a shell implement I/O redirection?

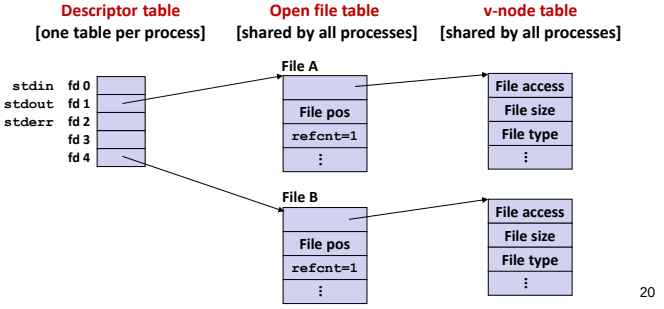
```
linux> ls > foo.txt
```

- Answer: By calling the `dup2(oldfd, newfd)` function
  - Copies (per-process) descriptor table entry **`oldfd`** to entry **`newfd`**



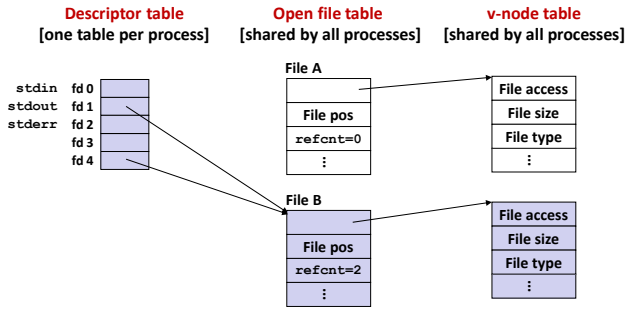
I/O Redirection Example

- Step #1: open file to which stdout should be redirected
  - Happens in child executing shell code, before **`exec`**



I/O Redirection Example (cont.)

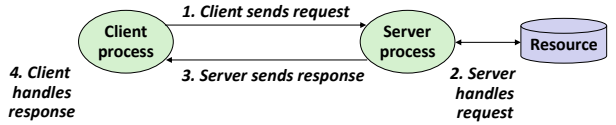
- Step #2: call `dup2(4,1)`
  - cause fd=1 (stdout) to refer to disk file pointed at by fd=4



Networking

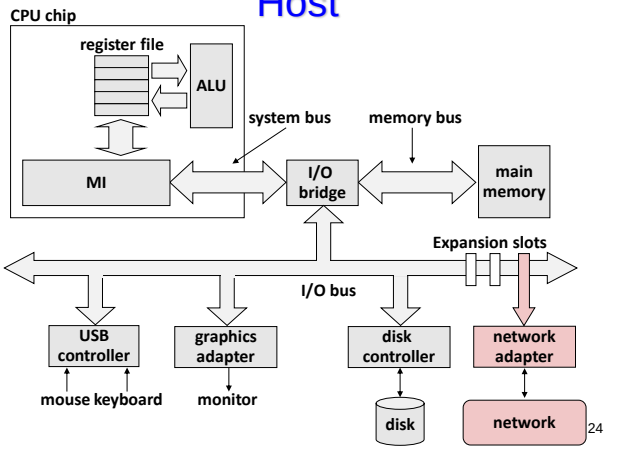
A Client-Server Transaction

- Most network applications are based on the client-server model:
  - A ***server*** process and one or more ***client*** processes
  - Server manages some ***resource***
  - Server provides ***service*** by manipulating resource for clients
  - Server activated by request from client (vending machine analogy)



Note: clients and servers are processes running on hosts (can be the same or different hosts)

Hardware Organization of a Network Host



Computer Networks

- A *network* is a hierarchical system of boxes and wires organized by geographical proximity
  - SAN (System Area Network) spans cluster or machine room
    - Switched Ethernet, Quadrics QSW, …
  - LAN (Local Area Network) spans a building or campus
    - Ethernet is most prominent example
  - WAN (Wide Area Network) spans country or world
    - Typically high-speed point-to-point lines
- An *internetwork* (*internet*) is an interconnected set of networks
  - The Global IP Internet (uppercase "I") is the most famous example of an internet (lowercase "i")
- Let's see how an internet is built from the ground up

25

25

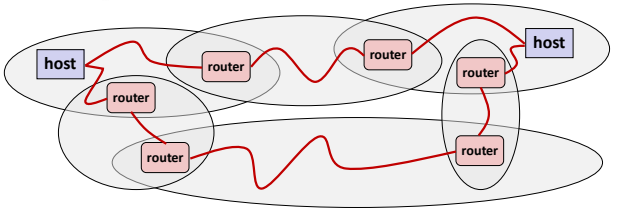
Conceptual View of LANs

- For simplicity, hubs, bridges, and wires are often shown as a collection of hosts attached to a single wire:

26

26

Logical Structure of an internet



- Ad hoc interconnection of networks
  - No particular topology
  - Vastly different router & link capacities
- Send packets from source to destination by hopping through networks
  - Router forms bridge from one network to another
  - Different packets may take different routes

27

27

The Notion of an internet Protocol

- How is it possible to send bits across incompatible LANs and WANs?
- Solution: *protocol* software running on each host and router
  - Protocol is a set of rules that governs how hosts and routers should cooperate when they transfer data from network to network.
  - Smooths out the differences between the different networks

28

28

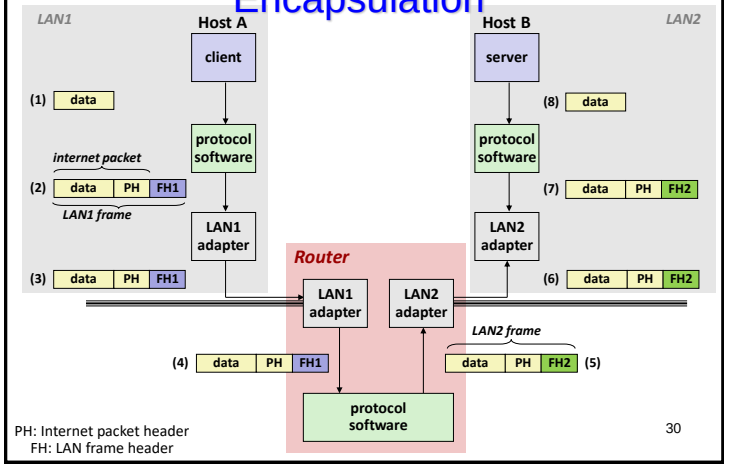
What Does an Internet Protocol Do?

- Provides a *naming scheme*
  - An internet protocol defines a uniform format for ***host addresses***
  - Each host (and router) is assigned at least one of these internet addresses that uniquely identifies it
- Provides a *delivery mechanism*
  - An internet protocol defines a standard transfer unit (***packet***)
  - Packet consists of ***header*** and ***payload***
    - Header: contains info such as packet size, source and destination addresses
    - Payload: contains data bits sent from source host

29

29

Transferring internet Data Via Encapsulation



30

30

Other Issues

- We are glossing over a number of important questions:
  - What if different networks have different maximum frame sizes? (segmentation)
  - How do routers know where to forward frames?
  - How are routers informed when the network topology changes?
  - What if packets get lost?
- These (and other) questions are addressed by the area of systems known as *computer networking*

31

31

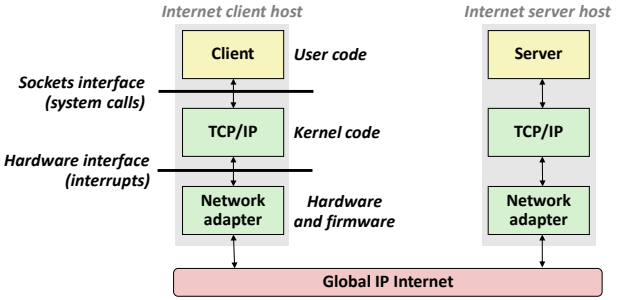
Global IP Internet (upper case)

- Most famous example of an internet
- Based on the TCP/IP protocol family
  - IP (Internet Protocol) :
    - Provides *basic naming scheme* and unreliable *delivery capability* of packets (datagrams) from *host-to-host*
  - UDP (Unreliable Datagram Protocol)
    - Uses IP to provide *unreliable* datagram delivery from *process-to-process*
  - TCP (Transmission Control Protocol)
    - Uses IP to provide *reliable* byte streams from *process-to-process* over *connections*
- Accessed via a mix of Unix file I/O and functions from the *sockets interface*

32

32

Hardware and Software Organization of an Internet Application



A Programmer's View of the Internet

1. Hosts are mapped to a set of 32-bit *IP addresses*
   - 128.2.203.179

2. The set of IP addresses is mapped to a set of identifiers called Internet *domain names*
   - 128.2.203.179 is mapped to www.cs.cmu.edu

3. A process on one Internet host can communicate with a process on another Internet host over a *connection*

Aside: IPv4 and IPv6

- The original Internet Protocol, with its 32-bit addresses, is known as *Internet Protocol Version 4* (IPv4)
- 1996: Internet Engineering Task Force (IETF) introduced *Internet Protocol Version 6* (IPv6) with 128-bit addresses
  - Intended as the successor to IPv4
- As of 2015, vast majority of Internet traffic still carried by IPv4
  - Only 4% of users access Google services using IPv6.
- We will focus on IPv4, but will show you how to write networking code that is protocol-independent.

(1) IP Addresses

- 32-bit IP addresses are stored in an *IP address struct*
  - IP addresses are always stored in memory in *network byte order* (big-endian byte order)
  - True in general for any integer transferred in a packet header from one machine to another.
    - E.g., the port number used to identify an Internet connection.

```
/* Internet address structure */
struct in_addr {
    uint32_t  s_addr; /* network byte order (big-endian) */
};
```

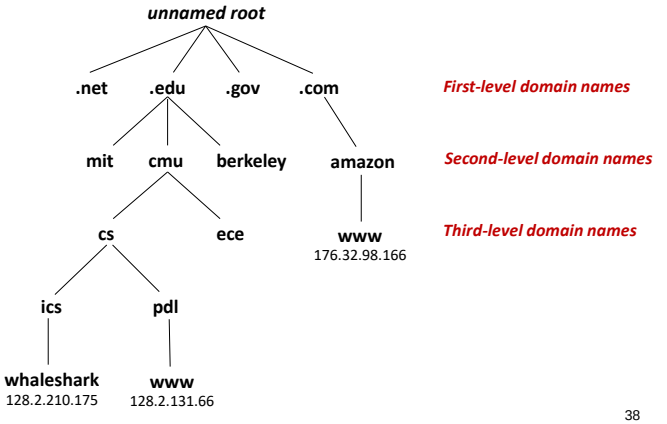
Dotted Decimal Notation

- By convention, each byte in a 32-bit IP address is represented by its decimal value and separated by a period
  - IP address: `0x8002C2F2 = 128.2.194.242`
- Use `getaddrinfo` and `getnameinfo` functions (described later) to convert between IP addresses and dotted decimal format.

37

37

(2) Internet Domain Names



38

38

Domain Naming System (DNS)

- The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called *DNS*
- Conceptually, programmers can view the DNS database as a collection of millions of *host entries.*
  - Each host entry defines the mapping between a set of domain names and IP addresses.
  - In a mathematical sense, a host entry is an equivalence class of domain names and IP addresses.

39

39

Properties of DNS Mappings

- Can explore properties of DNS mappings using `nslookup`
  - Output edited for brevity
- Each host has a locally defined domain name `localhost` which always maps to the *loopback address* `127.0.0.1`

```
linux> nslookup localhost
   Address: 127.0.0.1
```

- Use `hostname` to determine real domain name of local

```
linux> hostname
whaleshark.ics.cs.cmu.edu
```

40

40

Properties of DNS Mappings (cont)

- Simple case: one-to-one mapping between domain name and IP address:

```
linux> nslookup whaleshark.ics.cs.cmu.edu
         Address: 128.2.210.175
```

- Multiple domain names mapped to the same IP address:

```
linux> nslookup cs.mit.edu
     Address: 18.62.1.6
linux> nslookup eecs.mit.edu
     Address: 18.62.1.6
```

41

41

Properties of DNS Mappings (cont)

- Multiple domain names mapped to multiple IP addresses:

```
linux> nslookup www.twitter.com
Address: 199.16.156.6
Address: 199.16.156.70
Address: 199.16.156.102
Address: 199.16.156.230

linux> nslookup twitter.com
Address: 199.16.156.102
Address: 199.16.156.230
Address: 199.16.156.6
Address: 199.16.156.70
```

- Some valid domain names don't map to any IP address:

```
linux> nslookup ics.cs.cmu.edu
*** Can't find ics.cs.cmu.edu: No answer
```

42

42

(3) Internet Connections

- Clients and servers communicate by sending streams of bytes over *connections*. Each connection is:
  - *Point-to-point*: connects a pair of processes.
  - *Full-duplex*: data can flow in both directions at the same time,
  - *Reliable*: stream of bytes sent by the source is eventually received by the destination in the same order it was sent.
- *A socket* is an endpoint of a connection
  - *Socket address* is an `IPaddress:port` pair
- A *port* is a 16-bit integer that identifies a process:
  - ***Ephemeral port*:** Assigned automatically by client kernel when client makes a connection request.
  - ***Well-known port:*** Associated with some *service* provided by a server (e.g., port 80 is associated with Web servers)

43

43

Well-known Ports and Service Names

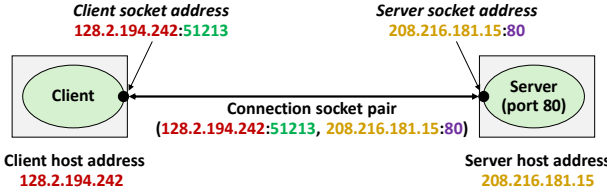
- Popular services have permanently assigned *well-known ports and* corresponding *well-known service names*:
  - echo server: 7/echo
  - ssh servers: 22/ssh
  - email server: 25/smtp
  - Web servers: 80/http
- Mappings between well-known ports and service names is contained in the file `/etc/services` on each Linux machine.

44

44

Anatomy of a Connection

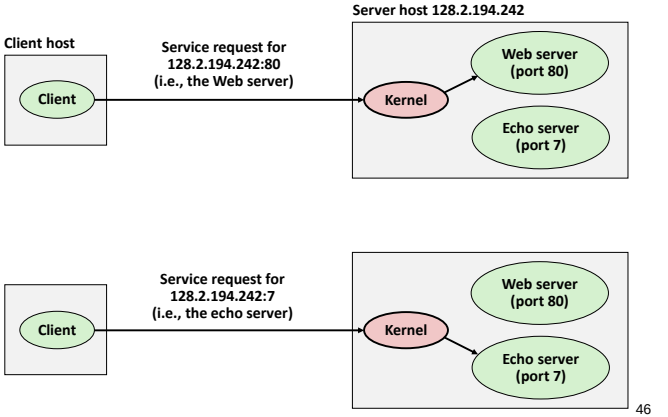
- A connection is uniquely identified by the socket addresses of its endpoints (*socket pair*)
  - `(cliaddr:cliport, servaddr:servport)`



51213 is an ephemeral port allocated by the kernel

80 is a well-known port associated with Web servers

Using Ports to Identify Services

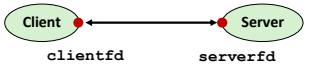


Sockets Interface

- Set of system-level functions used in conjunction with Unix I/O to build network applications.
- Created in the early 80's as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols.
- Available on all modern systems
  - Unix variants, Windows, OS X, IOS, Android, ARM

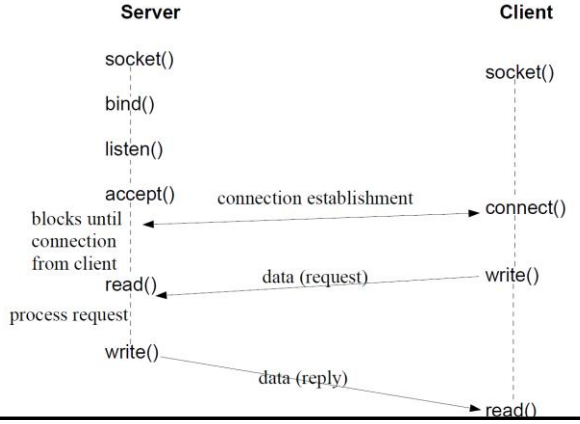
Sockets

- What is a socket?
  - To the kernel, a socket is an endpoint of communication
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network
    - ***Remember:*** All Unix I/O devices, including networks, are modeled as files
- Clients and servers communicate with each other by reading from and writing to socket descriptors



- The main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors

Socket System Calls for Connection-Oriented Protocol: TCP



49

Concurrency

50

50

Why employ concurrency?

- Resource sharing, information exchange, collaboration
- Tolerate delays such as slow I/O devices
- Provide good response times, e.g., with human interaction
- Separate logical tasks
  - Garbage collection
  - Separate logical flow for each client in a concurrent server
- Reduce latency by deferring work
- Execute in parallel on hardware such as multicore machines

51

51

Concurrent Programming is Hard!

- The human mind tends to be sequential
- The notion of time is often misleading
- Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible

52

52

What do we need?

- Communication
  - Messages versus shared memory
- Coordinate
  - Synchronization
    - Mutual exclusion
    - Events

53

53

Why is Parallel Computing Hard?

- Amdahl's law – insufficient available parallelism
  - Speedup = 1/(fraction_enhanced/speedup + (1-fraction_enhanced)
- Overhead and complexity of communication and coordination
  - Classic problems concurrent programs
    - ***Races:*** outcome depends on arbitrary scheduling decisions elsewhere in the system
    - ***Deadlock:*** improper resource allocation prevents forward progress
    - ***Livelock / Starvation / Fairness***: external events and/or system scheduling decisions can prevent sub-task progress
- Portability – knowledge of underlying architecture often required

54

54

Steps in the Parallelization Process

- Decomposition into tasks
- Assignment to processes
- Orchestration – communication of data, synchronization among processes

55

55

Breakout: Sum Vector Elements in Parallel

```
#define LENGTH 1024
int array[LENGTH]; /* initialized elsewhere */
int sum = 0;

void combine()
{
    long int i;
    for (i = 0; i < LENGTH; i++) {
       sum = sum + array[i];
    }
}
```

56

56

Types of Dependences

- Flow (or True) dependence – RAW
- Anti-dependence – WAR
- Output dependence – WAW

57

57

Synchronization

- Basic types
  - Mutual exclusion
  - Events
- Components of a synchronization operation
  - Acquire method (enter critical section, proceed past event)
  - Waiting algorithm (busy waiting, blocking)
  - Release method (enable others to proceed)

58

58

Data Sharing: CPU and Cache Support

- Special atomic read-modify-write instructions
  - Test-and-set, fetch-and-increment, load-linked/store conditional
- Coherent caches
  - Ensure that modifications propagate to copies

59

59

Lessons Learned

- Must have parallelization strategy
  - Partition into K independent parts
  - Divide-and-conquer
- Inner loops must be synchronization free
  - Synchronization operations very expensive
- Beware of Amdahl's Law
  - Serial code can become bottleneck
- You can do it!
  - Achieving modest levels of parallelism is not difficult
  - Set up experimental framework and test multiple strategies

66

66