dsh & Networking

COMPSCI210 Recitation

21 Sep 2012

Vamsi Thummala

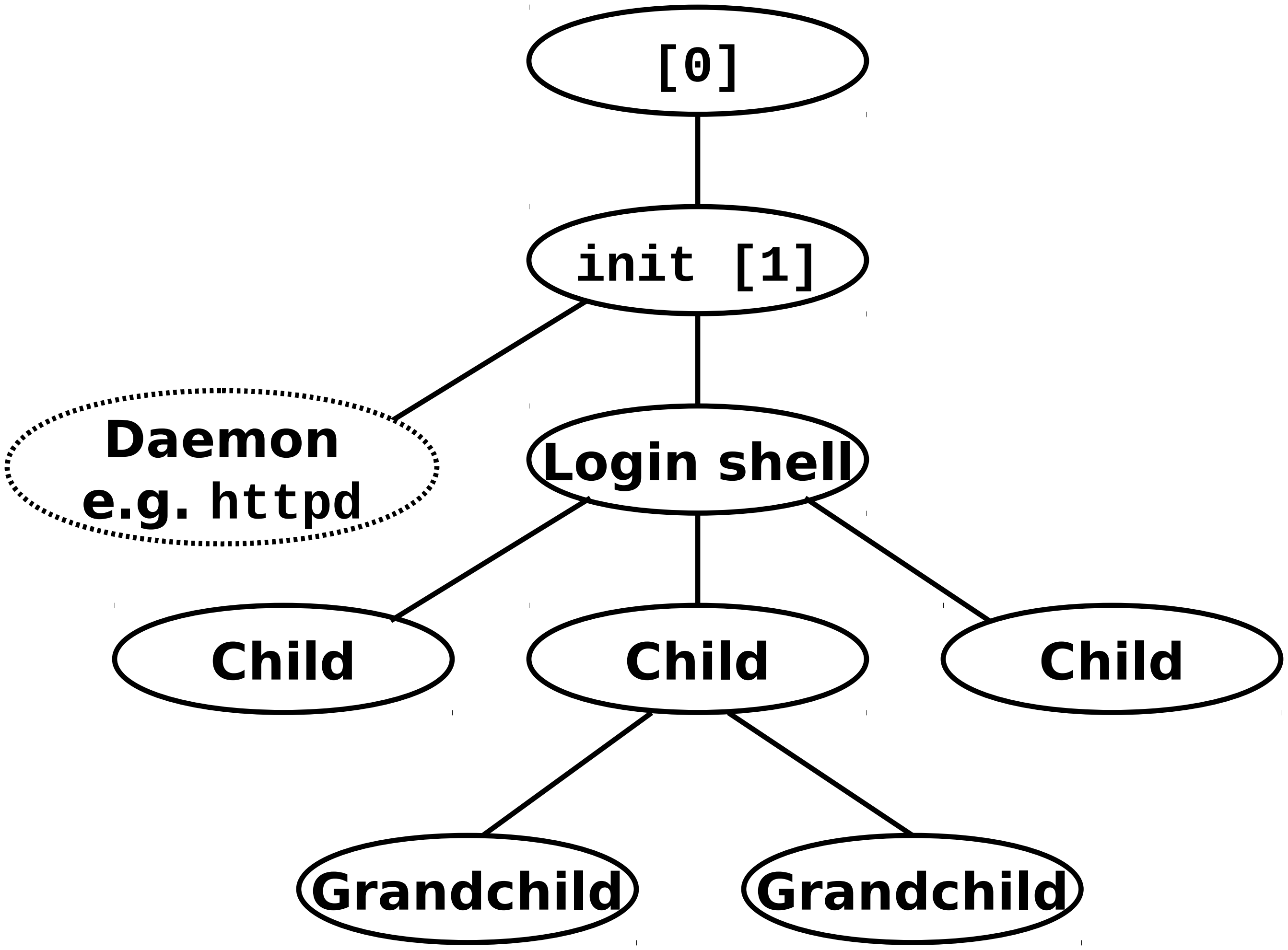
Devil shell (dsh)

- Read the handout
- Read the “Exceptional Control Flow” from CS:APP
- Form groups of size two; three is okay
- Start early!
- Use piazza for posting questions

Review: Shell

- Interactive command interpreter
- A high level language (scripting)
- Interface to the OS
- Provides support for key OS ideas
  - Isolation
  - Concurrency
  - Communication
  - Synchronization

Unix Process Hierarchy



Shell Concepts

- Process creation
- Execution
- Input/Output redirection
- Pipelines
- Job control
  - Process groups
  - Foreground/background jobs
    - Given that many processes can be executed concurrently, which processes should have accesses to the keyboard/screen (I/O)?
  - Signals (limited for the lab!)
    - SIGCONT

dsh

- Built in commands
  - bg
  - fg
  - jobs
  - cd
  - ctrl-d (quit/exit the dsh)

Data structures

```
/* A process is a single process.  */
typedef struct process {
        struct process *next;       /* next process in pipeline */
        int argc;                   /* useful for free(ing) argv */
        char **argv;                /* for exec; argv[0] is the path of the executable file*/
        pid_t pid;                  /* A process is a single process.  */
        bool completed;             /* true if process has completed */
        bool stopped;               /* true if process has stopped */
        int status;                 /* reported status value from job control; 0 on success and nonzero otherwise */
} process_t;


/* A job is a process itself or a pipeline of processes.
 * Each job has exactly one process group (pgid) containing all the processes in the job.
 * Each process group has exactly one process that is its leader.
 */
typedef struct job {
        struct job *next;           /* next job */
        char *commandinfo;          /* entire command line input given by the user; useful for logging and message 
display*/
        process_t *first_process;   /* list of processes in this job */
        pid_t pgid;                 /* process group ID */
        bool notified;              /* true if user told about stopped job */
        struct termios tmodes;      /* saved terminal modes */
        int stdin, stdout, stderr;  /* standard i/o channels */
        bool bg;                    /* true when & is issued on the command line */
        char *ifile;                /* stores input file name when < is issued */
        char *ofile;                /* stores output file name when > is issued */
} job_t;
```

Parser demo

Getting started on dsh ...

- Include the pid in display prompt
- Start logging all the info
  - Required for dsh!
- Play with parser
- Implement built-in commands
  - cd, jobs
- Input/output redirection
  - Use the MACROs provided
  - dup2()
- Add support for pipelines
- Add support for bg and fg

Shell Refresher

Understanding fork

Process

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Child Process

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

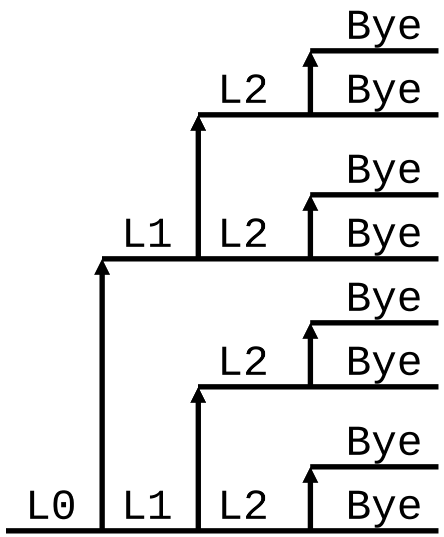
```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

`hello from parent` ***Which one is first?*** `hello from child`

Fork Example

Both parent and child can continue forking

```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



`waitpid()`: Waiting for a Specific Process

`waitpid(pid, &status, options)`

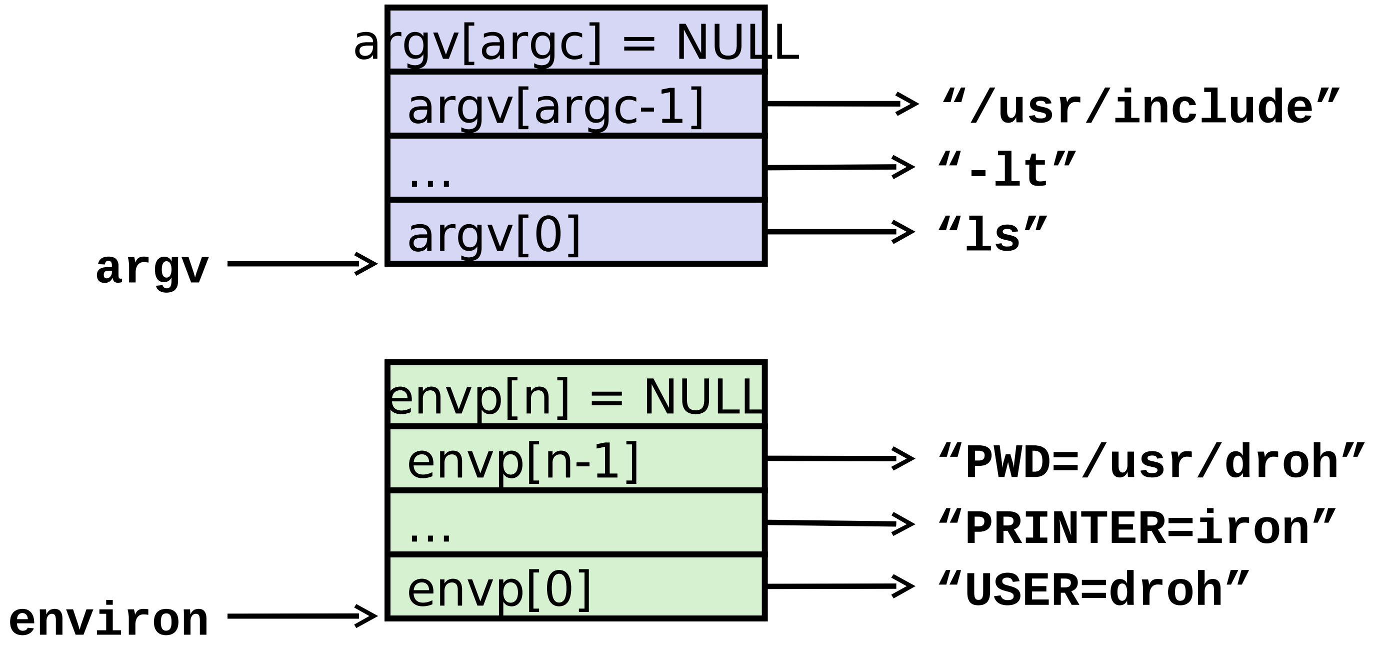
suspends current process until specific process terminates

various options (see CS:APP)

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

execve Example

```
if ((pid = Fork()) == 0) { /* Child runs user job */
    if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found.\n", argv[0]);
        exit(0);
    }
}
```



Pipe between parent/child

```
int fdarray[2];
char buffer[100];
pipe(fdarray);
 switch (pid = fork()) {
        case -1: perror(“fork failed”);
        case 0:  write(fdarray[1], "hello world", 5);
        default: n = read(fdarray[0], buffer, sizeof(buffer)); //block until data is available
 }
```

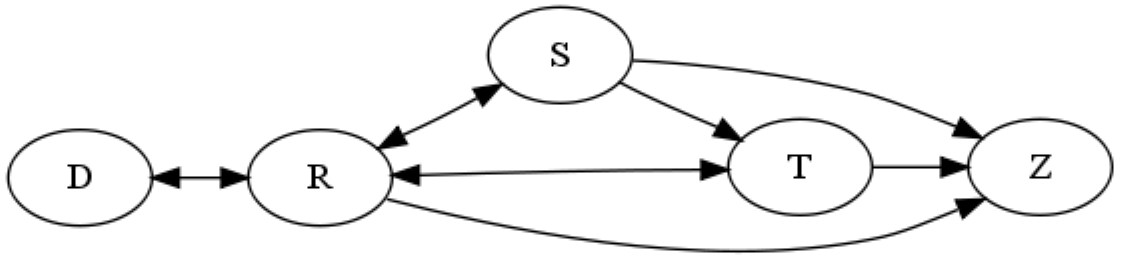
How does the pipes in shell, i.e, “ls | wc”?

dup2(oldfd, newfd); // duplicates fd; closes and copies at one shot

Implementing bg and fg

- Set process group
  - setpgid()
  - Tcsetpgrp()
- ctrl-z
  - stops a fg job
- In dsh, you cannot stop a bg job
- Resuming jobs
  - kill(-(j->pgid), SIGCONT)
  - Note the negative sign
    - Interpreted as process group

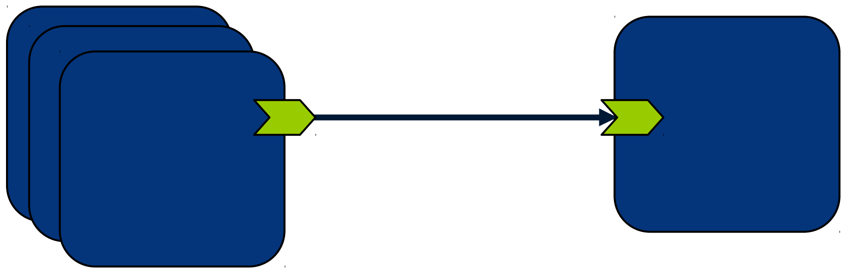
Process States



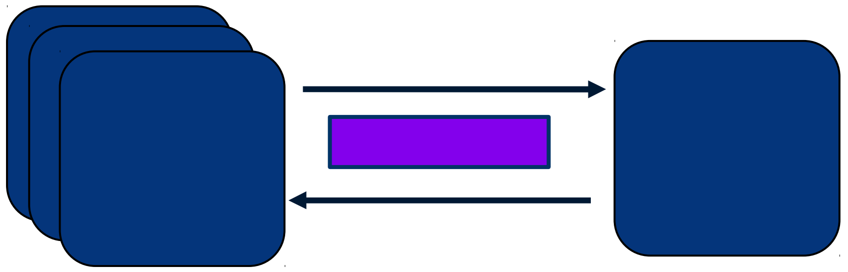
- R	Running or runnable (on run queue)
- D	Uninterruptible sleep (waiting for some event)
- S	Interruptible sleep (waiting for some event or signal)
- T	Stopped, either by a job control signal or because it is being traced by a debugger.
- Z	Zombie process, terminated but not yet reaped by its parent.

Client/Server/Networking

Services

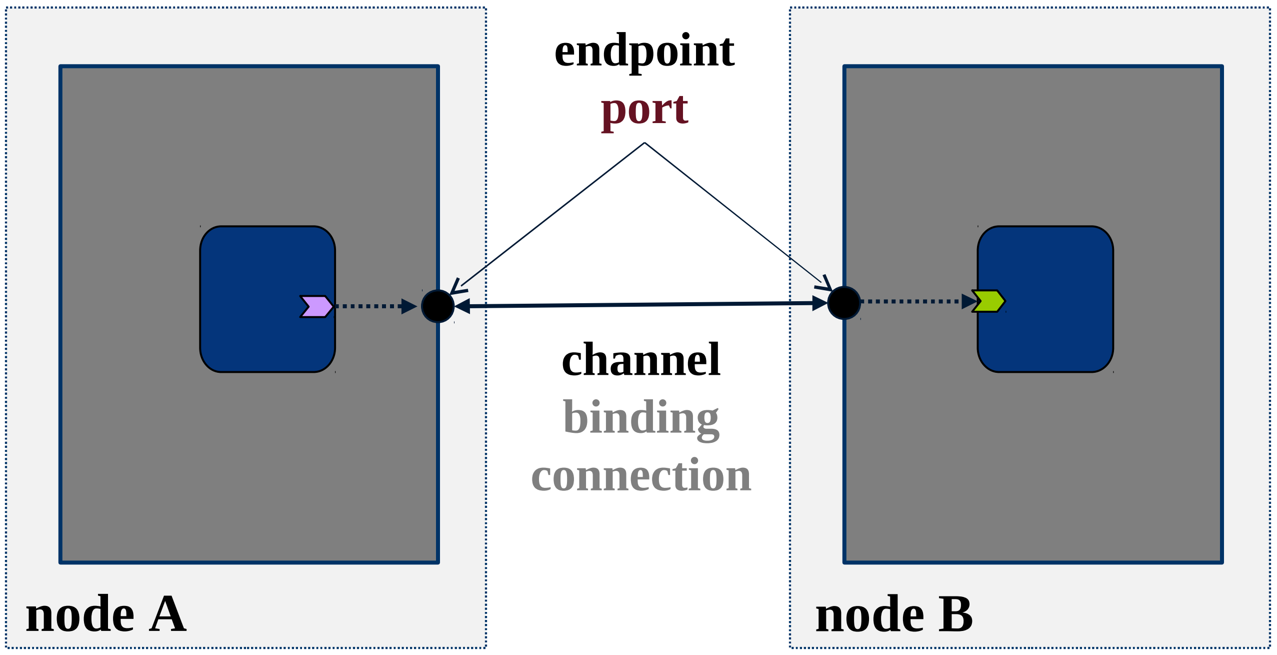


RPC



GET (HTTP)

Networking



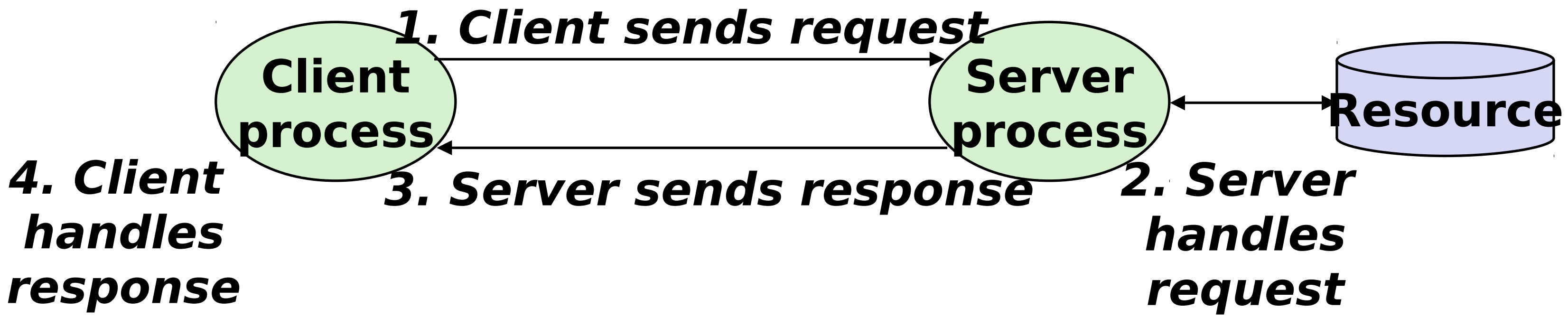
operations
advertise (bind)
listen
connect (bind)
close

write/send
read/receive

Some IPC mechanisms allow communication across a network.
- E.g.: sockets using Internet communication protocols (TCP/IP).
- Each endpoint on a node (host) has a port number.

Each node has one or more interfaces, each on at most one network.
Each interface may be reachable on its network by one or more names.
- E.g. an IP address and an (optional) DNS name.

Client-Server Transaction

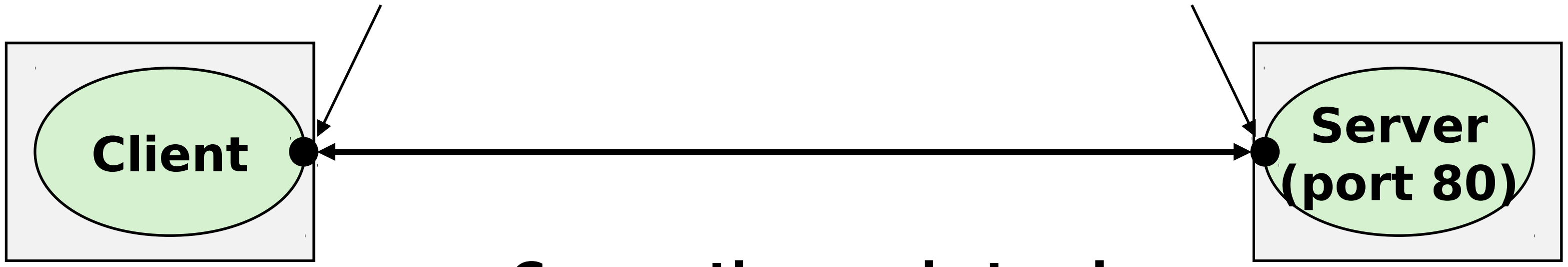


Note: clients and servers are processes running on hosts (can be the same or different hosts)

A detailed example: Client/Server Transaction

Client socket address
128.2.194.242:51213

Server socket address
208.216.181.15:80



Connection socket pair
(128.2.194.242:51213, 208.216.181.15:80)

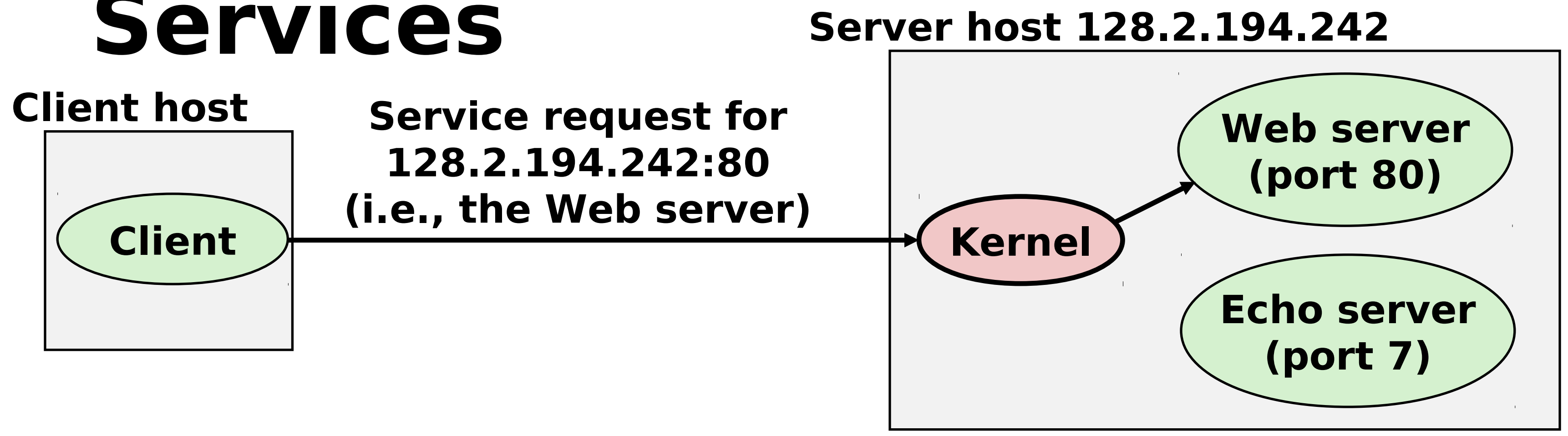
Client host address
128.2.194.242

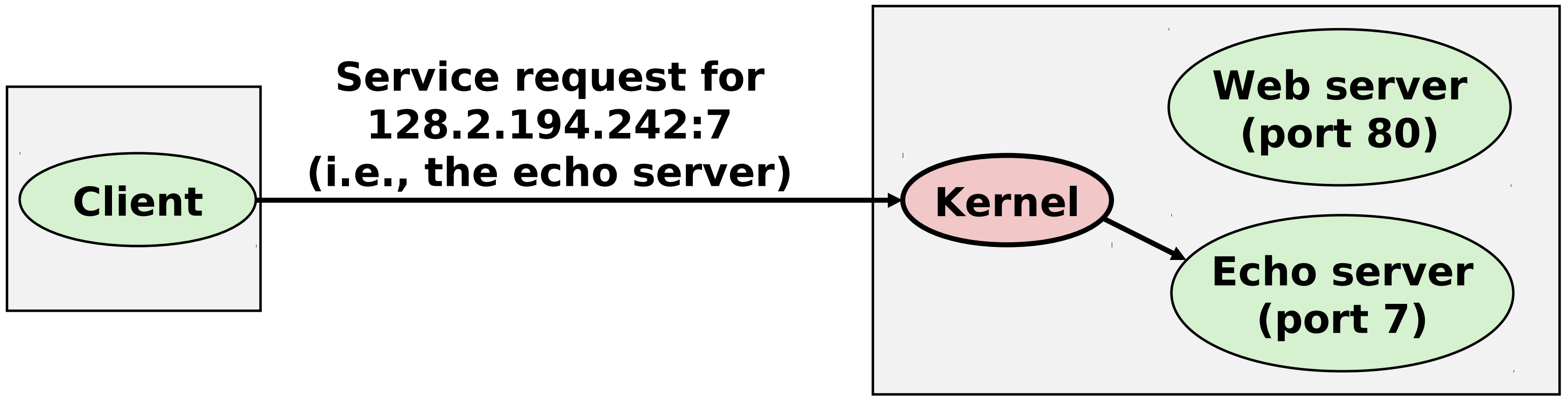
Server host address
208.216.181.15

51213 is an ephemeral port allocated by the kernel

80 is a well-known port associated with Web servers

Using Ports to Identify Services





Servers

Servers are long-running processes (daemons)

Created at boot-time (typically) by the init process (process 1)

Run continuously until the machine is turned off

Each server waits for requests to arrive on a well-known port associated with a particular service

Port 7: echo server

Port 23: telnet server

Port 25: mail server

Port 80: HTTP server

A machine that runs a server process is also often referred to as a "server"

Server Examples

Web server (port 80)

Resource: files/compute cycles (CGI programs)

Service: retrieves files and runs CGI programs on behalf of the client

FTP server (20, 21)

Resource: files

Service: stores and retrieve files

See `/etc/services` for a comprehensive list of the port mappings on a Linux machine

Telnet server (23)

Resource: terminal

Service: proxies a terminal on the server machine

Mail server (25)

Resource: email “spool” file

Service: stores mail messages in spool file

Sockets Interface

Created in the early 80’s as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols

Provides a user-level interface to the network

Underlying basis for all Internet applications

Based on client/server programming model

Sockets

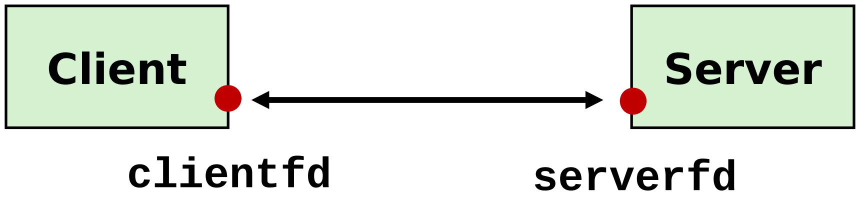
What is a socket?

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

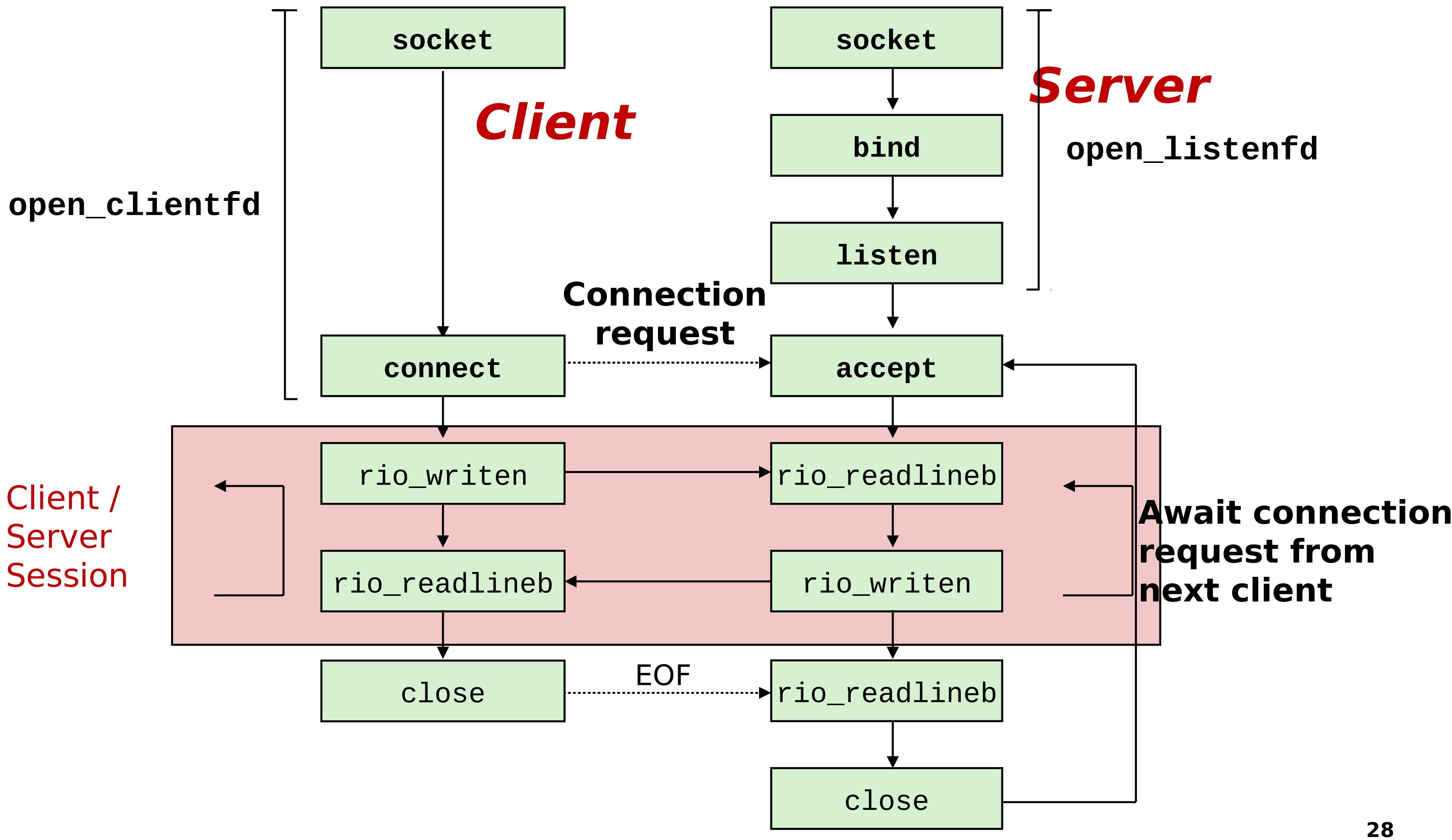
- ***Remember:*** All Unix I/O devices, including networks, are modeled as files

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

Overview of the Sockets Interface



java.net

- Low level API
  - Addresses
  - Sockets
  - Interfaces
- High level API
  - URIs
  - URLs
  - Connections