WANs

Introduction to Unix Programming and TCP/IP Sockets

UNIX PROGRAMMING

Fundamentals

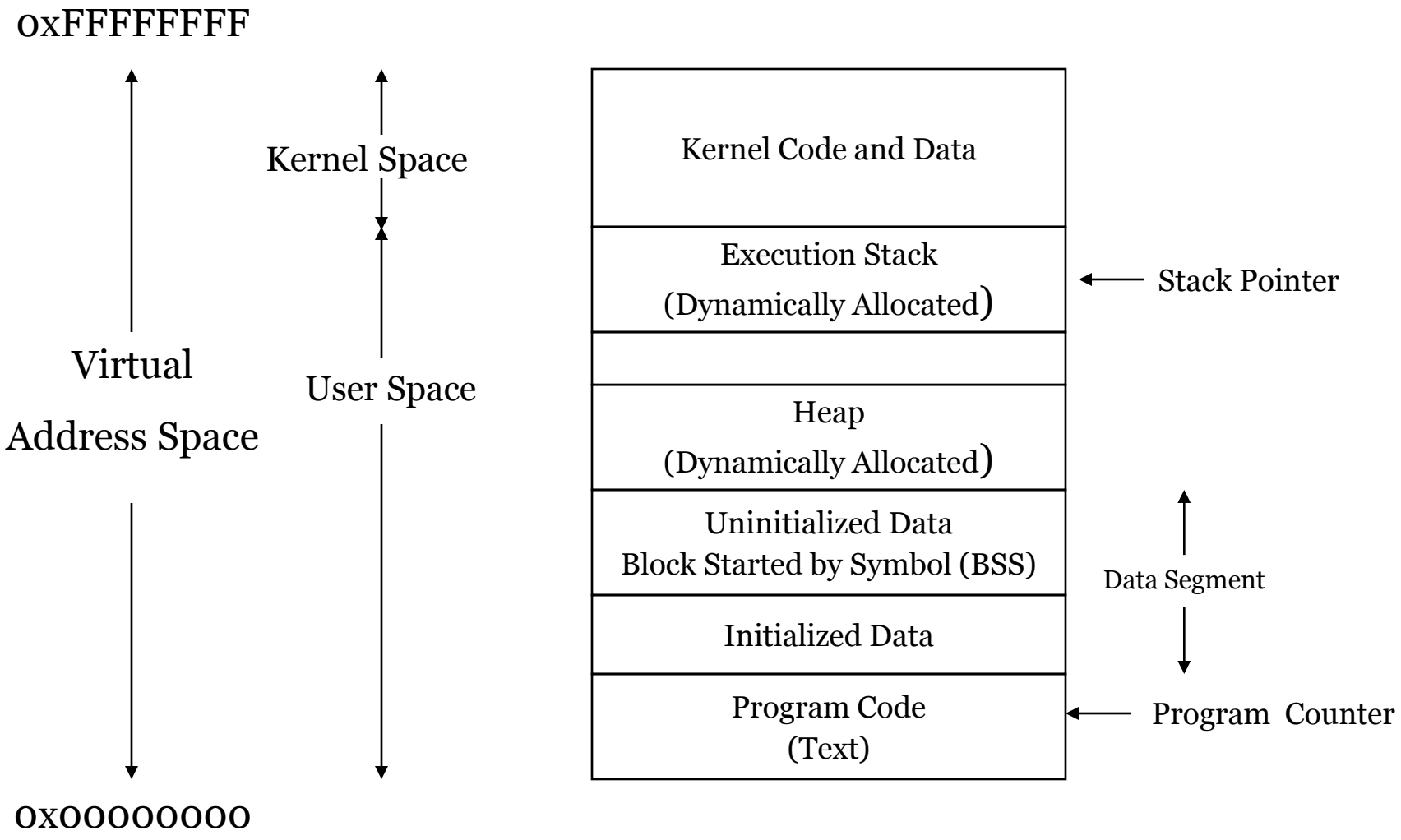
Process Definition

- A process is an instance of an application execution
  - A process is the most basic abstraction provided by the operating system
  - The unit of execution
  - A unit of scheduling
- A process is a dynamic execution context
  - **CPU State + Address Space + Environment**

Process Structure

- A process in Unix comprises:
  - An ***address space*** – usually protected and virtual – mapped into memory
  - The ***code*** for the running program
  - The ***data*** for the running program
  - An ***execution stack*** and ***stack pointer*** (SP)
  - The ***program counter*** (PC)
  - A set of processor ***registers*** – general purpose and status registers
  - A set of system ***resources***
    - Files, Network connections, Privileges, …

Processes – Address Space



Processes Representation

- To users and to other processes, a process is identified by its unique ***Process ID*** (PID)
  - PID <= 30,000 in Unix
- In the OS, processes are represented by entries in a ***Process Table*** (PT)
  - PID is index to (or pointer to) a PT entry
  - PT entry = ***Process Control Block*** (PCB)
- PCB is a large data structure that contains or points to all information about the process
  - Linux - Defined in *task_struct*
    - Over 70 fields

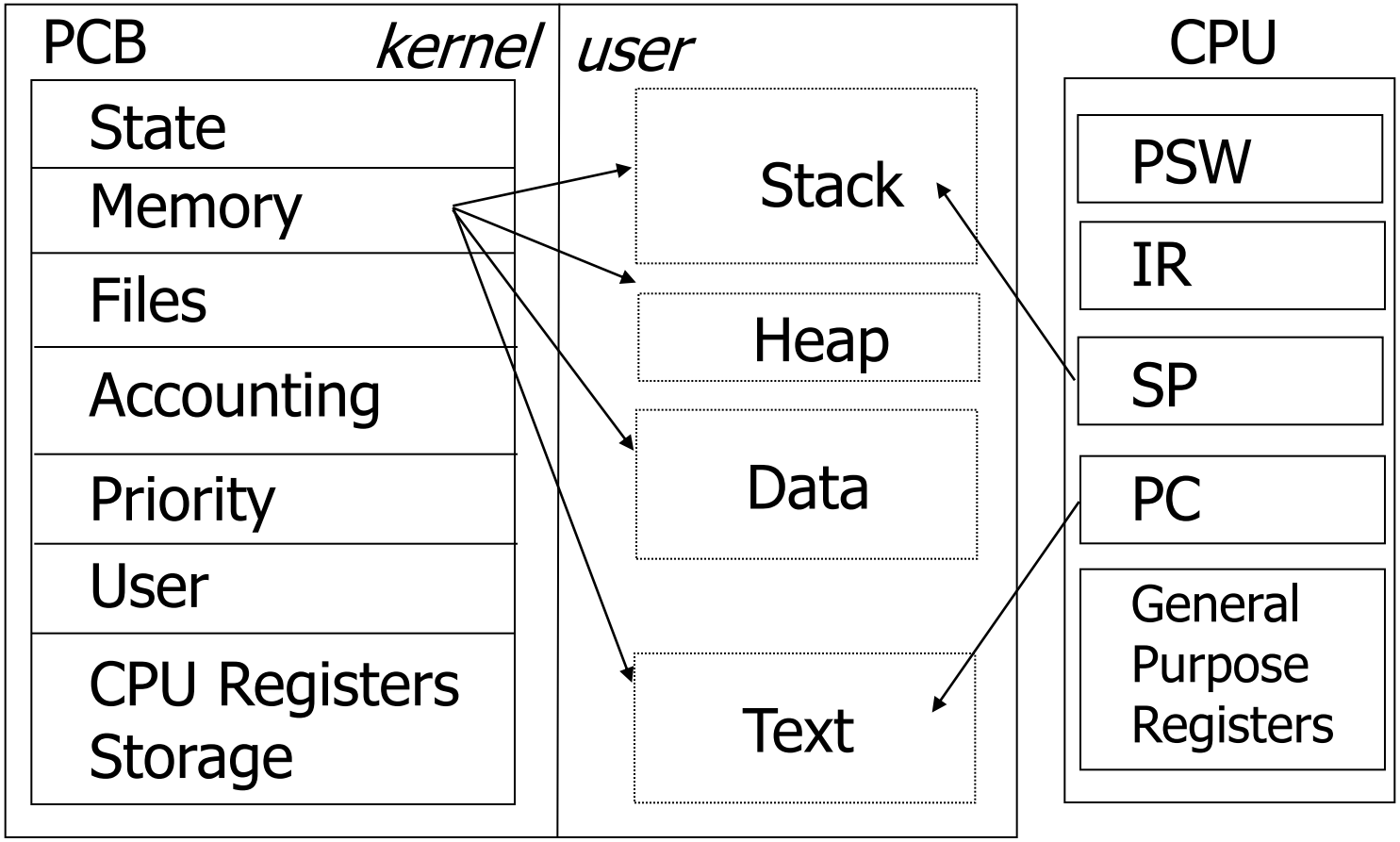
Process Control Block (PCB)

- Typical PCB contains:
  - Execution state
  - PC, SP & processor registers
    - Stored when process is made inactive
  - Memory management information
  - Privileges and owner information
  - Scheduling priority
  - Resource information
  - Accounting information

Process CPU State

- The CPU state is defined by the registers' contents
  - Process Status Word (PSW)
    - Execution mode, Last Operation Outcome, Interrupt Level
  - Instruction Register (IR)
    - Current instruction being executed
  - Program Counter (PC)
  - Stack Pointer (SP)
  - General purpose registers

Process control block (PCB)



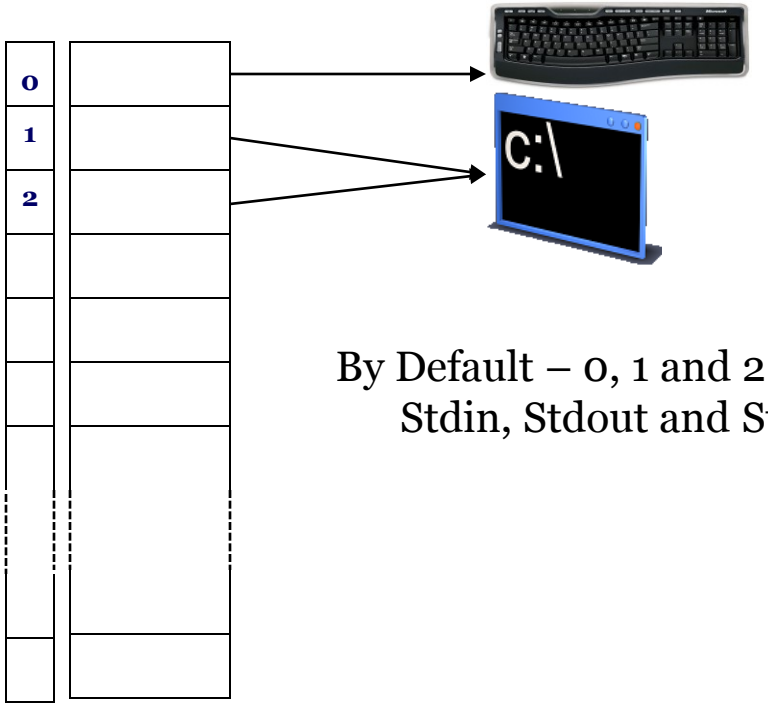
Process Environment

- External entities
  - Terminal
  - Open files
  - Communication channels
    - Local connections
    - Remote connections to other machines

Unix Descriptors

- Each process has its own Descriptor Table
  - A descriptor table is a data structure which allows a process to access the rest of the system's objects, when it is allowed
  - Each descriptor is a handle allowing a process to reference the corresponding object
    - Files, devices, …. and sockets.

Standard Input, Output and Error Descriptors



By Default – 0, 1 and 2 point to Stdin, Stdout and Stderr

Per-Process Descriptor Table

Descriptor Operators

- Descriptors are created by invoking:
  - ***open( ),***
  - ***dup( ),***
  - ***dup2( ),***
  - ***pipe( ),***
  - ***socket( )***
- Descriptors are removed by invoking
  - ***close( )***

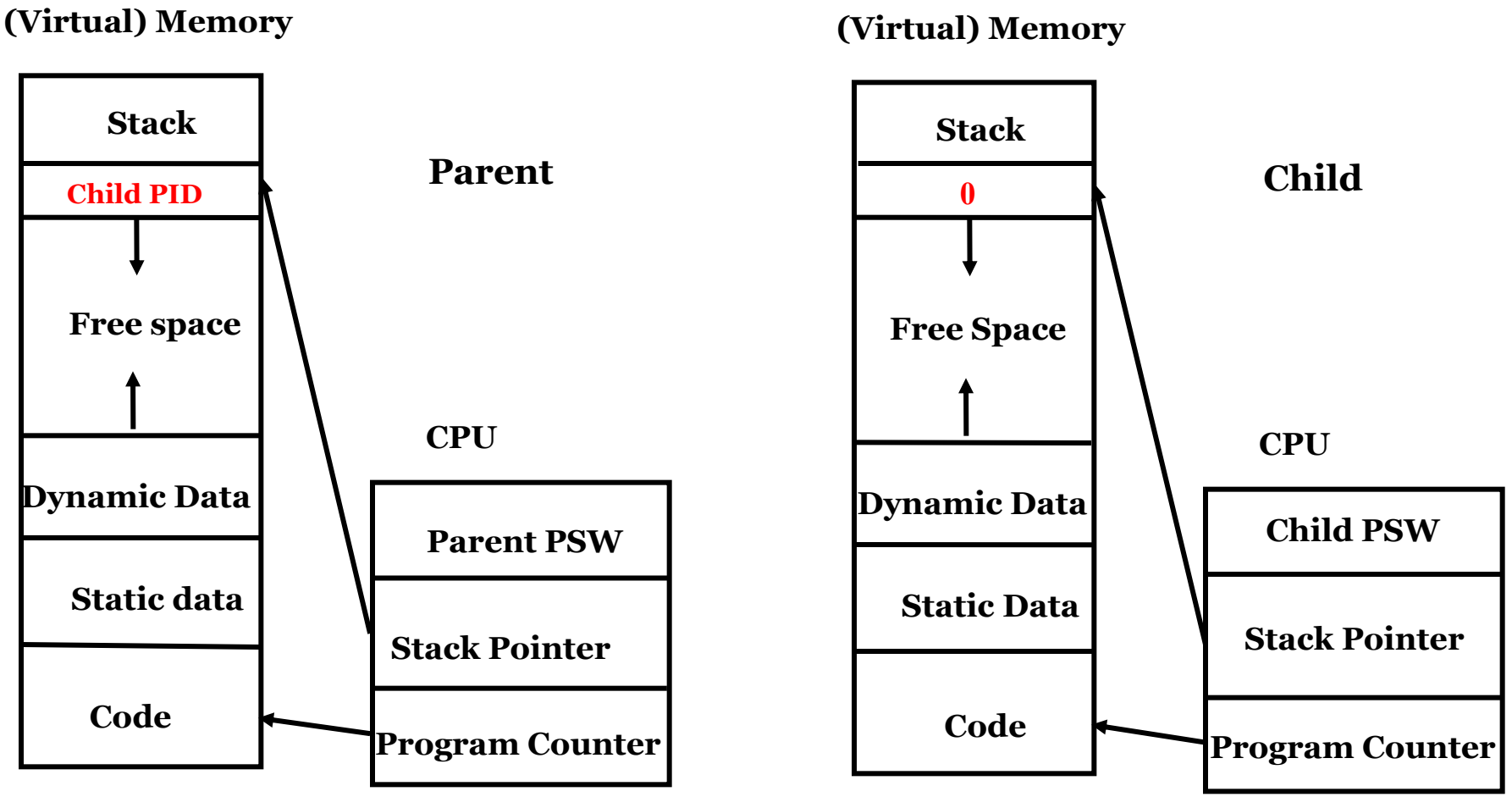
Process Creation in UNIX

- UNIX supports multiprogramming, allowing more than one process to execute concurrently at any given time
  - How processed are created?
  - How processes are manipulated?
- Processes are created in UNIX by invoking the ***fork()*** system call.
  - When a process P creates a process Q, Q is called the **child** of P and P is called the **parent** of Q.

The Fork System Call

- The fork() system call creates a "clone" of the calling process.
- The two processes are identical in every respect except for return value
  - The parent process is returned a non-zero value
    - The value represents the process id of the child
  - The child process is returned zero
- The process id returned to the parent can be used by parent in a wait() or kill() system call.
  - wait() – causes the parent to wait on the child
  - Kill() – terminates the execution of the child

Snapshots after fork()



Example using fork

```
1. #include <unistd.h>
2. main(){
3.    pid_t pid;
4.    printf("Just one process so far\n");
5.    pid = fork();
6.    if (pid == 0) /* code for child */
7.       printf("I'm the child\n");
8.    else if (pid > 0) /* code for parent */
9.       printf("My child pid is =%d\n", pid);
10.   else /* error handling */
11.      printf("An error has occurred\n");
12. }
```

Fork() Example

```
1. main(){
2.       int x=0;
3.       fork();
4.       x++;
5.       printf("The value of x is
   %d\n", x);
6.    }
```

What would be the value of x?

UNIX Process Control

- UNIX provides a number of system calls for process control including:
  - ***fork()*** - used to create a new process
  - ***exec()*** - to change the program a process is executing
  - ***exit()*** - used by a process to terminate itself normally
  - ***abort()*** - used by a process to terminate itself abnormally
  - ***kill()*** - used by one process to kill or signal another
  - ***wait()*** - to wait for termination of a child process
  - ***sleep()*** - suspend execution for a specified time interval
  - ***getpid()*** - get process id
  - ***getppid()*** - get parent process id

Spawning Applications

- **fork()** is typically used in conjunction with **exec()**
  - There are multiple variants of exec()

```
1.  main(){
2.  pid_t pid;
3.  if ( ( pid = fork() ) == 0 ) {
4.     /* child code: replace executable image */
5.     execv( "/usr/games/tetris", "-easy" )
6.  } else {
7.     /* parent code: wait for child to terminate */
8.     wait( &status )
9.  }
```

A Redirect Example – ls > temp

Create a subprocess from program “ls”; redirect standard output of “ls” into file named “temp”

Unix shell notation: `$ ls > temp`

```
int main(int argc, char *argv[]) {
    pid = fork();
    if (pid == 0) {
        fd = open("temp", O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU);
        dup2(fd, STDOUT_FILENO);
        if (execl(“/bin/ls”, “ls”, NULL) == -1)
            perror("execl");
    } else {
        close(fd);
        wait(&status);
    }
}
```

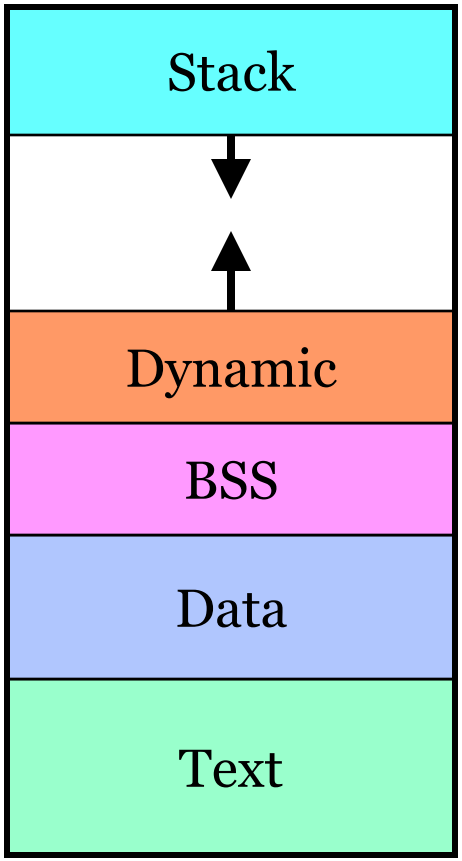
The exec() System Call

- A family of routines, exec(), execl(), execle(), execlp(), execv(), execve(), execvp(), or exect() Subroutine
  - execve(program_name, arg1, arg2, ..., environment)

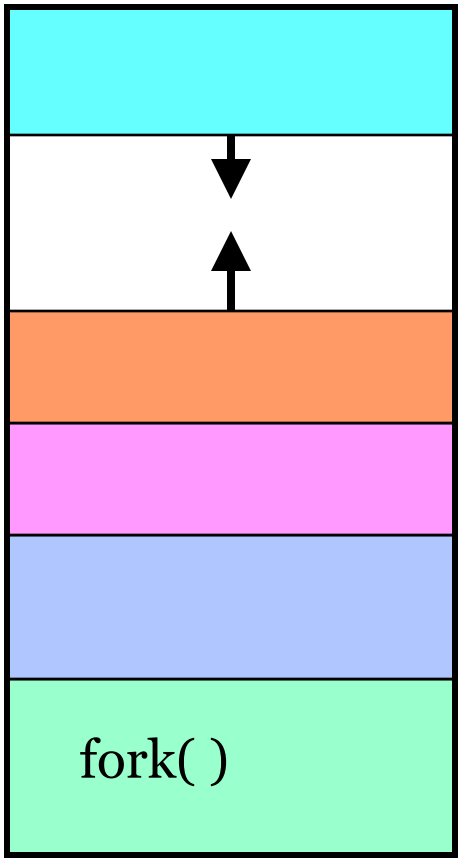
- The text and data segments of current process are replaced with those of program_name
- The stack is reinitialized with parameters
- The open file table of current process remains intact
- The last argument can pass environment settings
- The program_name is actually a path name of the executable file containing the program

Note – Unlike subroutine call, there is no return after this call. The program calling exec() is gone forever!

The Address Space of an Unix Process

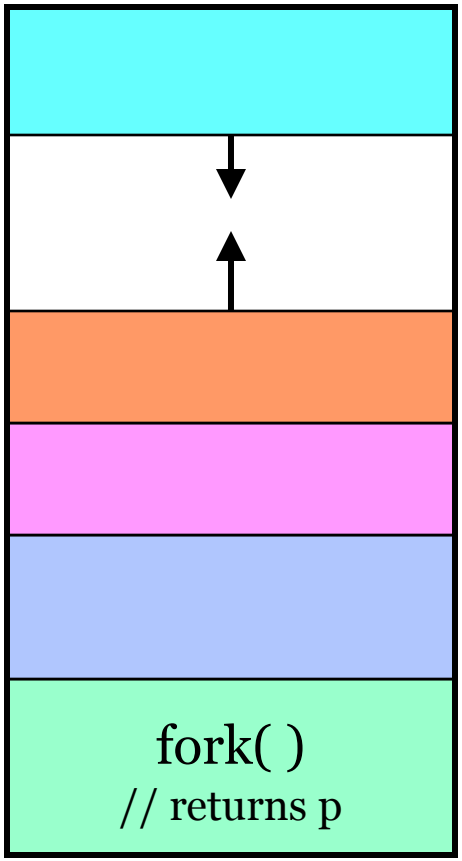


Creating a Process: Before fork()

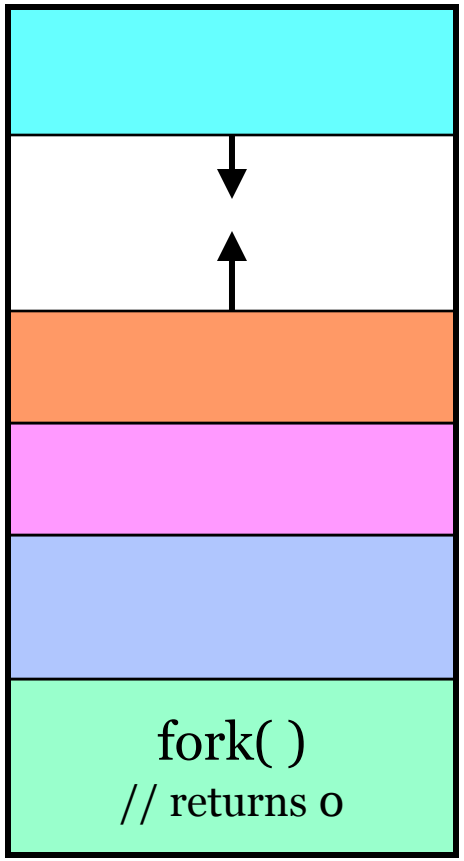


Parent Process

Creating a Process: After fork()

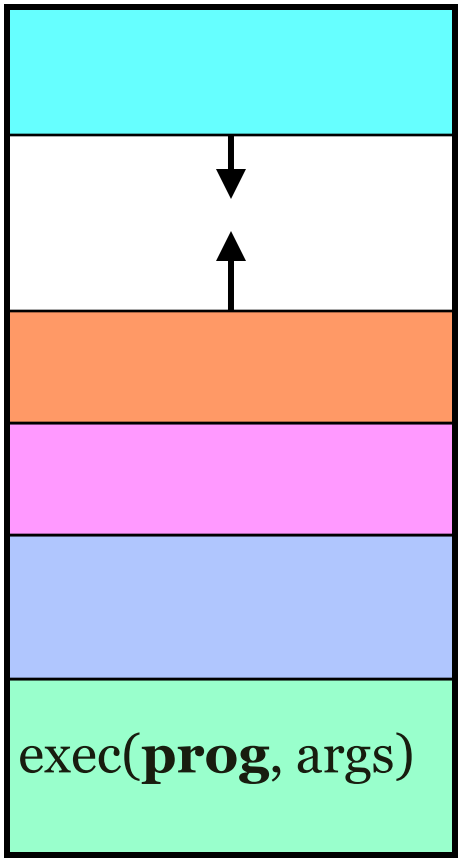


Parent Process

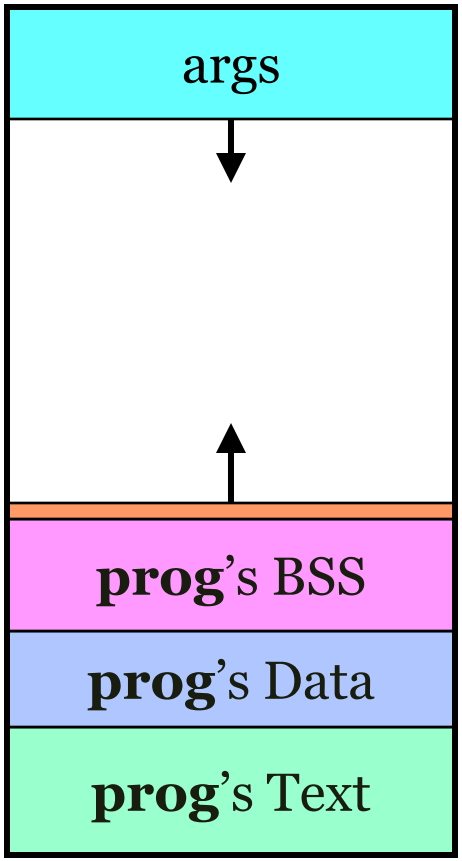


Child Process
(Process id = p)

exec(): Loading a New Image



Before



After

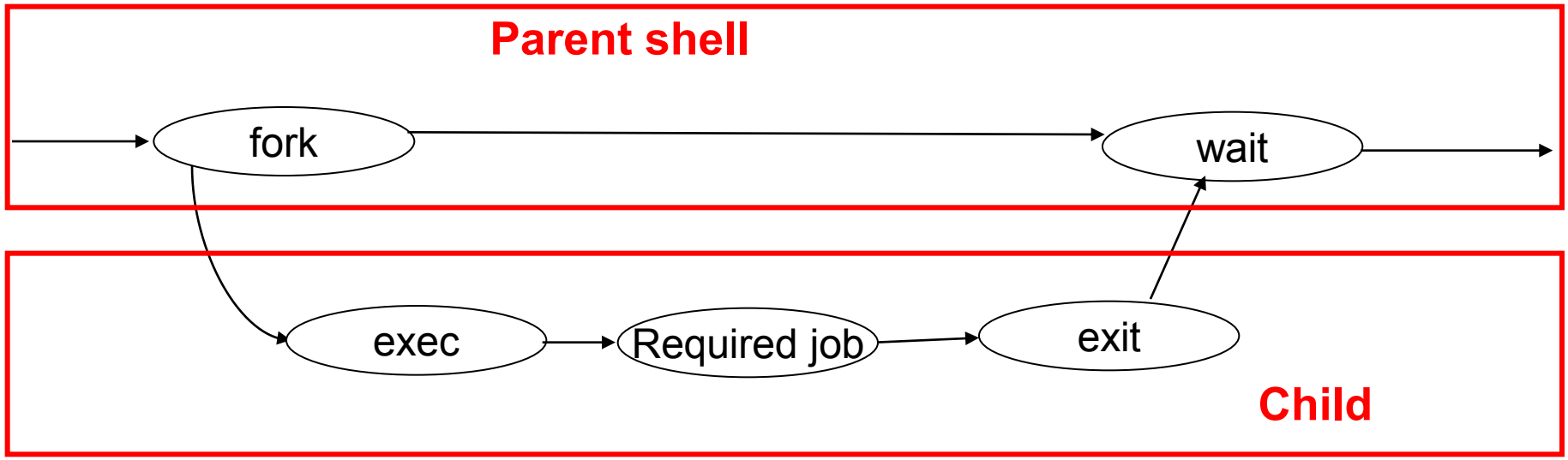
Parent-Child Synchronization

- **exit(status)** - executed by a child process when it wants to terminate. Makes status (an integer) available to parent.
- **wait(&status)** - suspends execution of process until *some* child process terminates
  - status indicates reason for termination
  - return value is process-id of terminated child
- **waitpid (pid, &status, options)**
  - pid can specify a specific child
  - Options can be to wait or to check and proceed

Process Termination

- A process can terminate itself with exit or it can be killed by another process using kill():
- ***kill(pid, sig)*** - sends signal sig to process with process-id pid.
- When a process terminates, all the resources it owns are reclaimed by the system
  - "Process control block" reclaimed
  - Its memory is deallocated
  - All open files closed and open file table reclaimed.
- Note: a process can kill another process only if:
  - Both processes belong to the same user
  - Super user

How shell executes a command

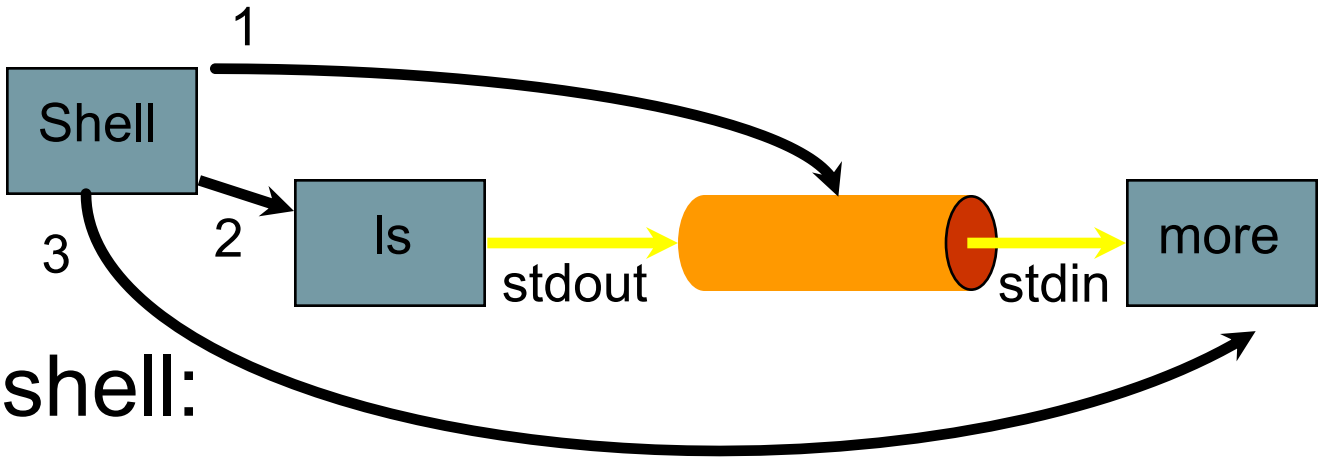


- In response to a command, the shell forks a clone of itself
- The child process makes an **exec()** call, which causes it to stop executing the shell and start executing the entered command
- The parent process, still running the shell, waits for the child to terminate

Pipes

- One process writes, 2nd process reads

```
% ls | more
```



- shell:
  1. create a pipe
  2. create a process for `ls` command, setting `stdout` to write side of pipe
  3. create a process for `more` command, setting `stdin` to read side of pipe

Unix Pipe

- Process inherits file descriptors from parent
  - File descriptor 0: stdin, 1: stdout, and 2: stderr
- Process doesn't know (or care!) when reading from keyboard, file, or process or writing to terminal, file, or process
- System calls:
  - read(fd, buffer, nbytes)
  - write(fd, buffer, nbytes)
  - pipe(rgfd) creates a pipe
    - rgfd array of 2 file descriptors: Read from rgfd[0], write to rgfd[1]

APPLICATION PROGRAMMING INTERFACE

Inter-Process Communication
Sockets

Client Server Model

- Servers are processes running on powerful computers dedicated to managing resources
  - File servers manage disk drives, Print servers manage printers, Web servers manage Web pages
- Clients are PCs or workstations on which users run applications.
- Clients rely on servers for resources, such as files, devices, and even processing power.

Servers

- A machine that runs a **server process** is often referred to as a "Server."
- Servers are long-running processes (daemons).
  - Created at boot-time, typically by the **init** process
  - Run continuously until the machine is turned off.
- Each server waits for requests to arrive for its associated service.

Networking Programming Client-Server Paradigm

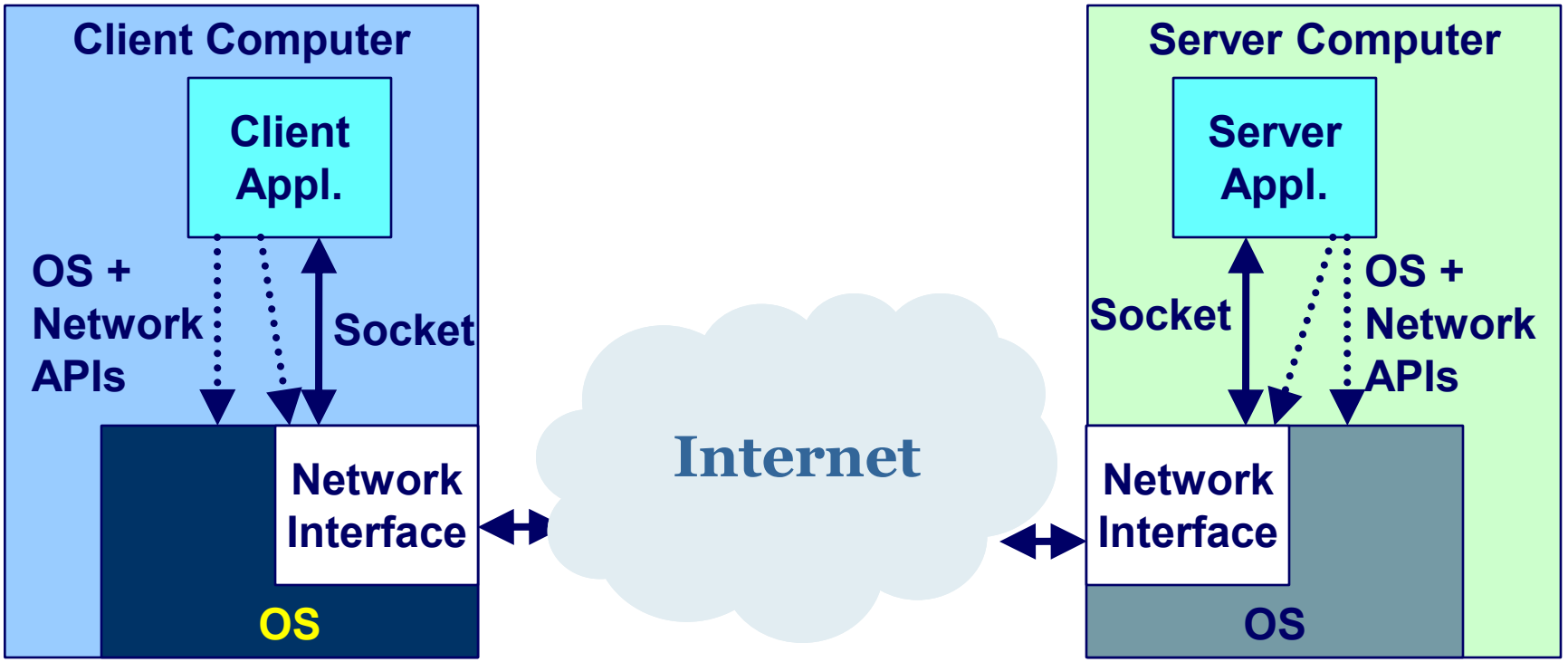
Application Programming Interface Sockets

- Communication is achieved based on the socket abstraction
  - 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.
- Clients and servers communicate with each 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.

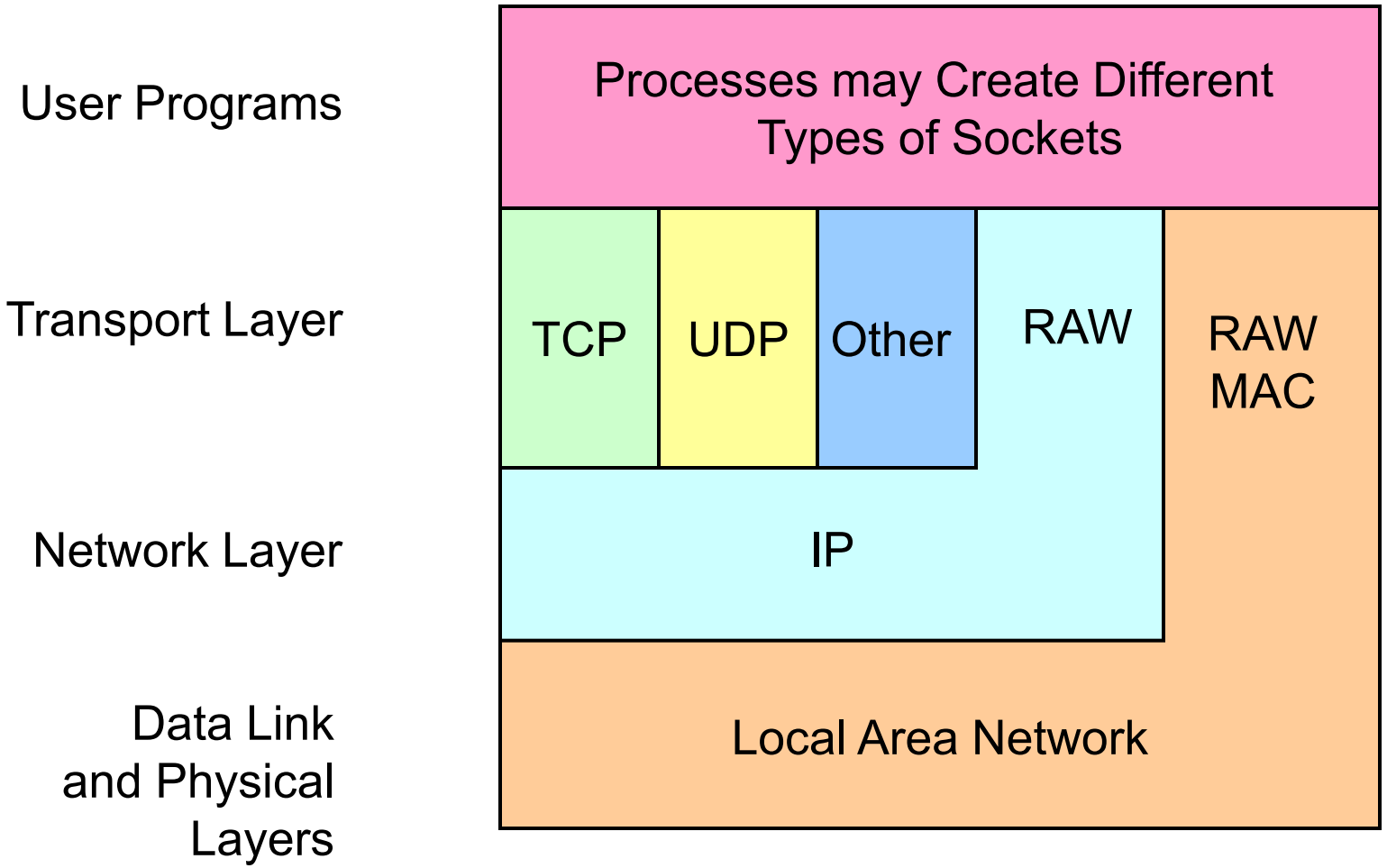
Application Programming Interface Sockets

- A socket lifetime
  - A socket is created,
  - Used for communication, and
  - Terminated

Network Applications



Protocol Architecture
Sockets

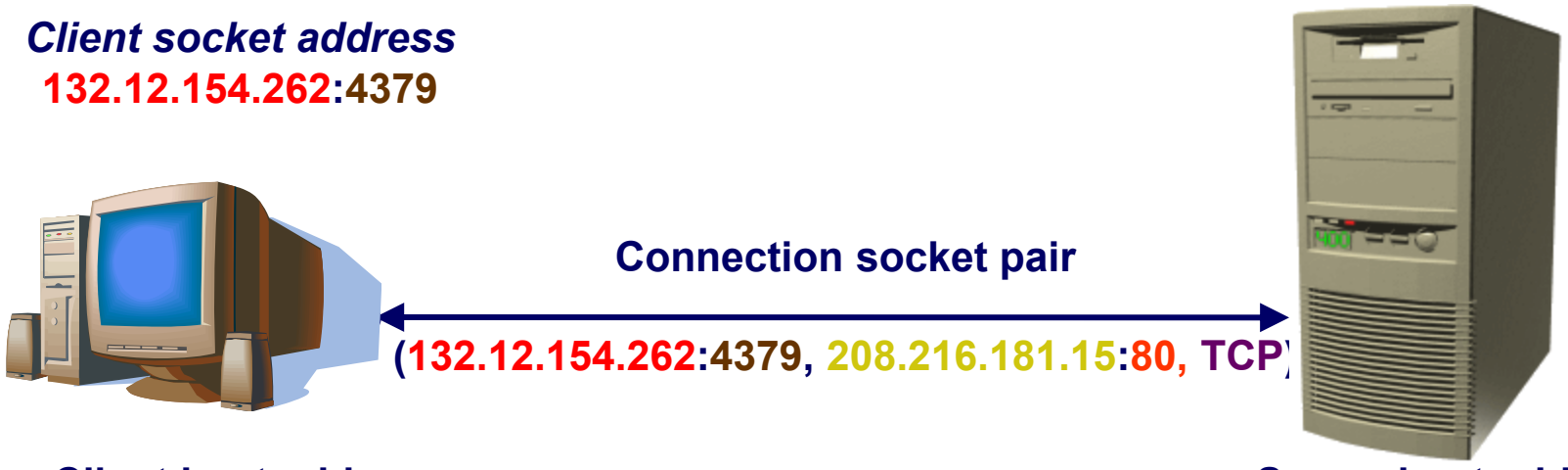


Server Specification – Addressing

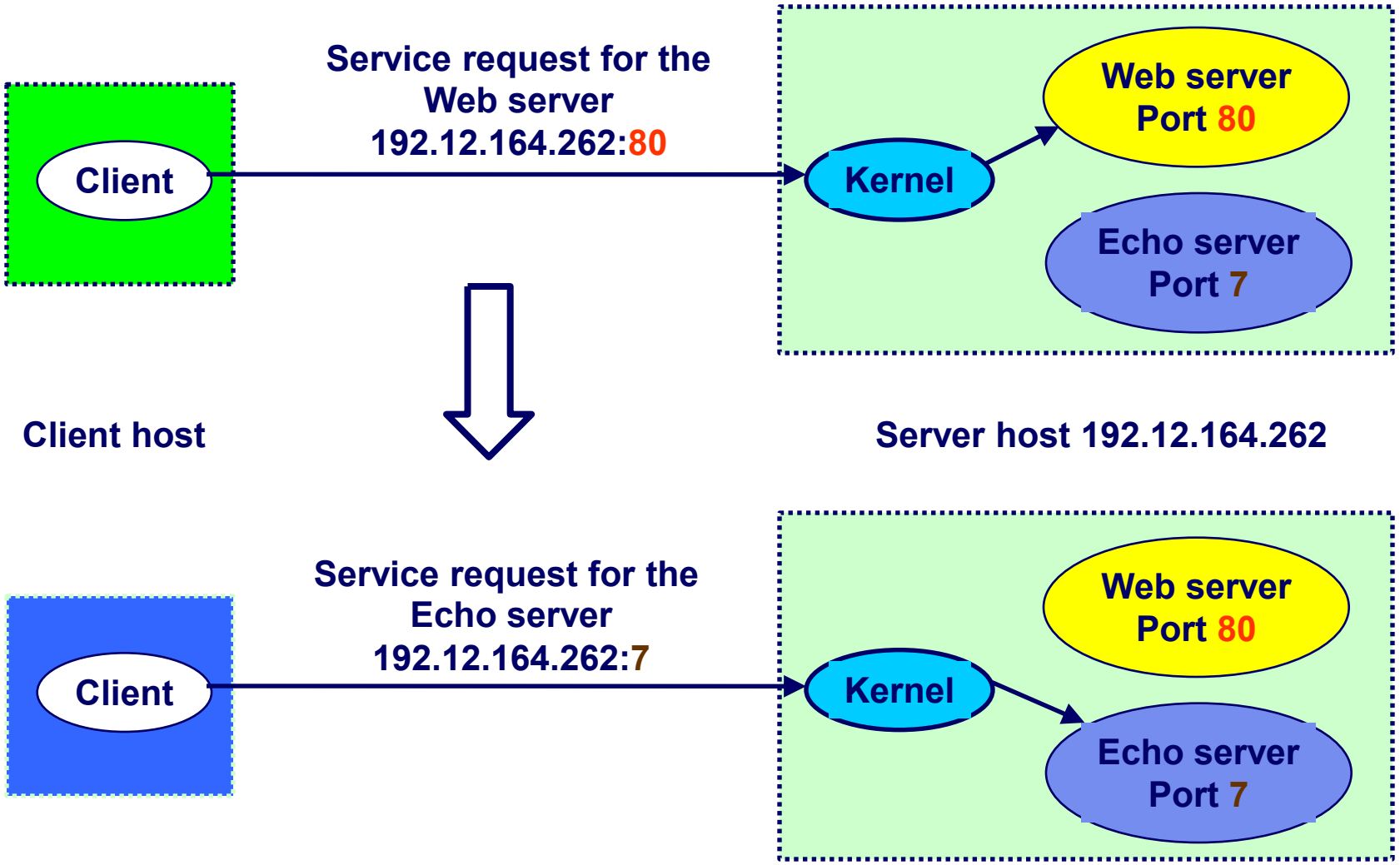
- How does a client specify a server?
  - The IP address in the server socket address identifies the host
    - *An adaptor on the host, more precisely*
  - The well-known port in the server socket address identifies the service,
    - Consequently, it implicitly identifies the server process that performs the requested service
  - Examples of well-known ports
    - Port 7: Echo server; Port 23: Telnet server; Port 25: Mail server; Port 80: Web server

TCP Client and Server Model

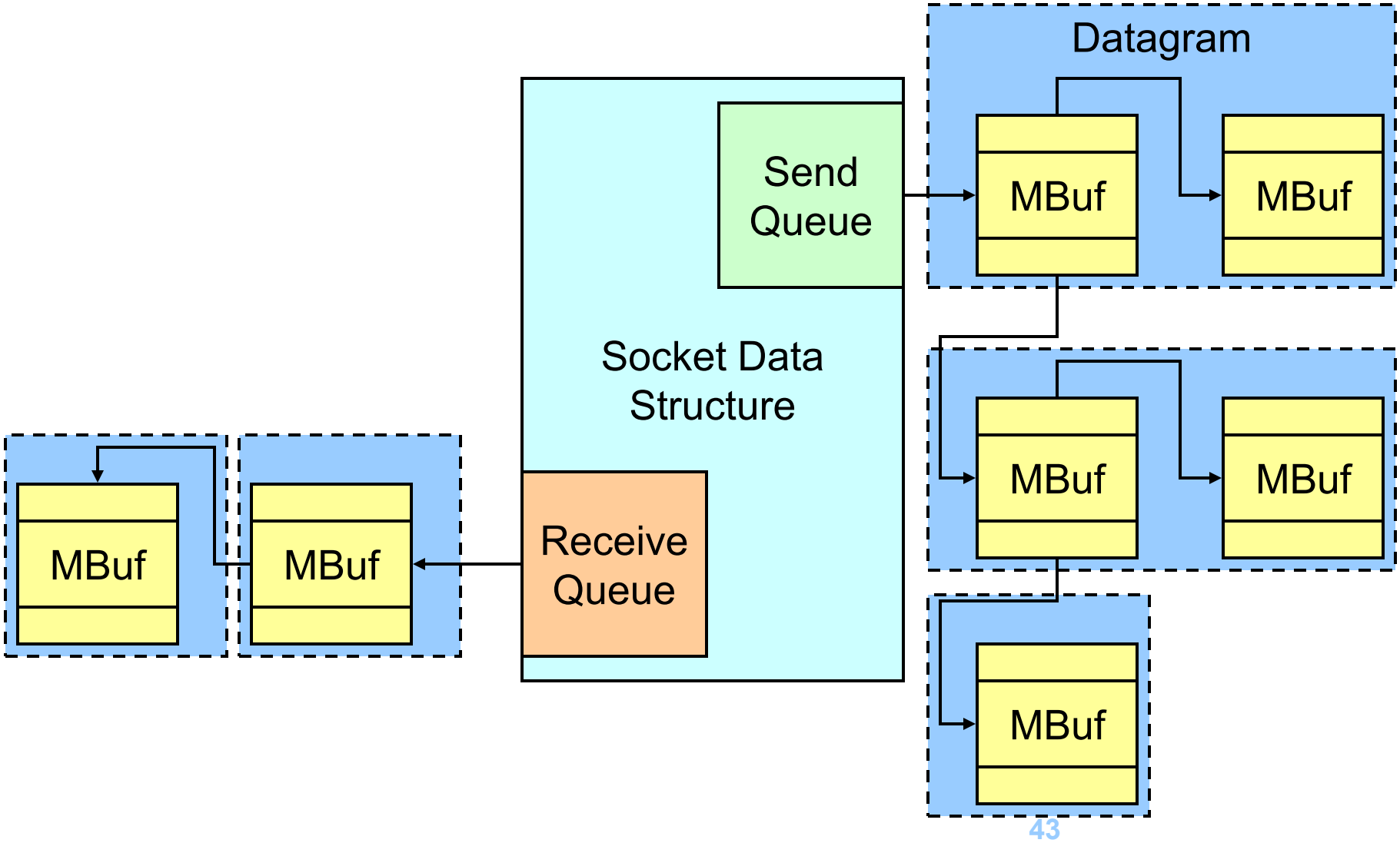
- Clients and servers communicate by sending streams of bytes over *connections*



Using Ports to Identify Services



Socket Structure



Socket API

- **int** mySock = **socket**(**int** prot_family,
  int type,
  int protocol);
- Protocol Family
  - The socket API provides a generic interface is designed to support different protocol families
    - PF_INET specifies a socket that uses protocols from the Internet Family
- Socket Type
  - Type determines the semantics of the socket in terms of the reliability, preservation of message boundaries, etc
    - SOCK_STREAM and SOCK_DGRAM
- Protocol to be used to support the requested socket type

Socket Types

- SOCK_STREAM, for a reliable stream delivery
  - Message boundaries are not preserved
- SOCK_DGRAM, for a datagram service
  - Unreliable delivery of data
- SOCK_RAW
  - raw IP for IP protocols
- SOCK_SEQ_PACKET
  - For a sequenced packet delivery

TCP/IP Socket Protocols

- TCP/IP PF_INET protocols
  - TCP ⟶ SOCK_STREAM
  - UDP ⟶ SOCK_DGRM
    - Defined in IPROTO_TCP Constant and IPPROTO_UDP Constant, respectively

	Family	Type	Protocol
TCP	PF_INET	SOCK_STREAM	IPPROTO_TCP
UDP		SOCK_DGRAM	IPPROTO_UDP

Socket Arguments – Protocol

- The protocol argument refers to the protocol that supports the semantics of the socket

Family	Type	Protocol	Actual Protocol
AF_INET	SOCK_DGRAM	IPPROTO_UDP	UDP
AF_INET	SOCK_STREAM	IPPROTO_TCP	TCP
AF_INET	SOCK_RAW	IPPROTO_ICMP	ICMP
AF_INET	SOCK_RAW	IPPROTO_RAW	raw IP

Socket Addressing

- Applications using sockets need to be able to specify Internet addresses and ports to the kernel
  - Client must specify the address to the server application
  - Sockets layer may need to pass addresses to the applications
- Socket API defines a generic data type:
  - struct sockaddr

```
{
        unsigned short sa_family;    /* Address family (e.g., AF_INET) */
        char sa_data[14];            /* Protocol-specific address information */
};
```

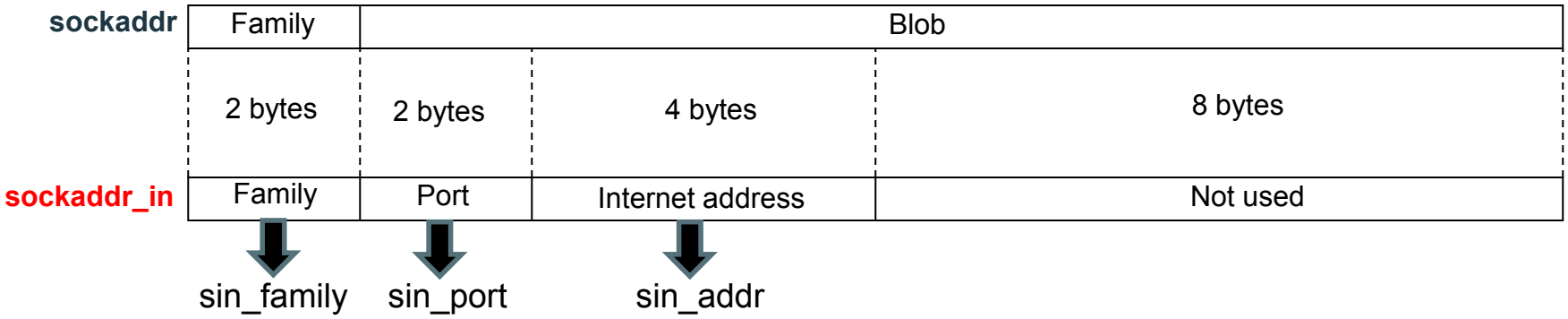
The exact form depends on the address family

Internet Address Family – AF_INET

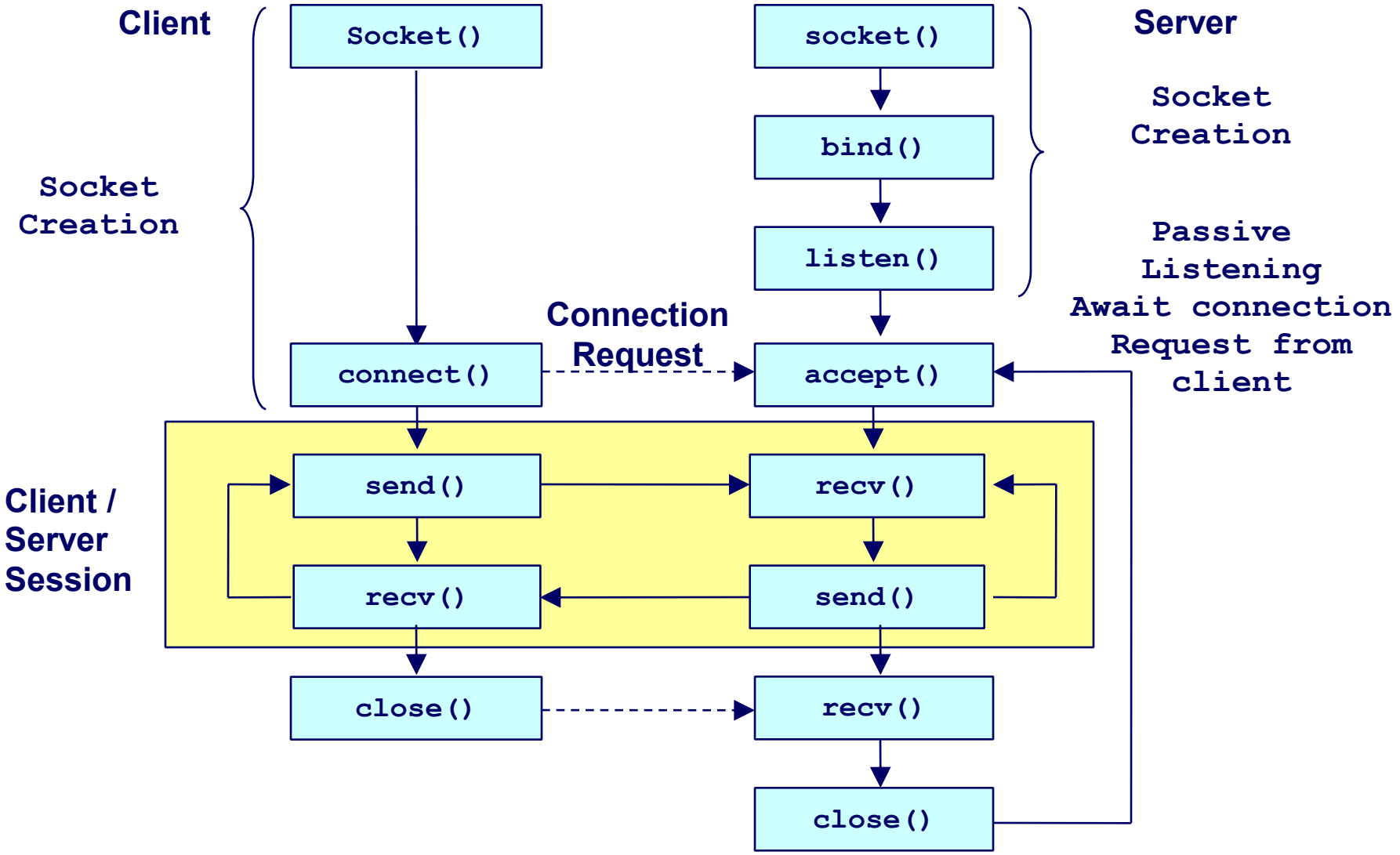
IP Specific

- struct sockaddr_in

```
{
        unsigned short sin_family;      /* Internet protocol (AF_INET) */
                unsigned short sin_port;        /* Port (16-bits) */
                struct in_addr sin_addr;        /* Internet address (32-bits) */
                char sin_zero[8];               /* Not used */
};
struct in_addr
{
        unsigned long s_addr;           /* Internet address (32-bits) */
};
```



Overview of the Sockets Interface



Socket – bind()

- The *bind*() function assigns a local socket address to a socket identified by descriptor *socket* that has no local socket address assigned.
- The command assures a connectionless client that the system has assigned to it some unique address, so that the other end has an address to return replies to

Socket Commands

- connect()
  - The client requests a connection with the server and if it is accepted, will obtain a valid read/write descriptor
- listen()
  - The server signals the system, that it is waiting for connection requests from clients at its “well-known” address
- accept()
  - The server accepts a connection and obtains a unique read/write descriptor for this connection – the connection obtains a different socket – to leave the “well-known” address to other clients requesting for a connection

TCP Client/Server Interaction

Server starts by getting ready to receive client connections…

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Create socket for incoming connections */
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
   Exit_if_Error("socket() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. **Create a TCP socket**
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
echoServAddr.sin_family = AF_INET;                    /* Internet address family */
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);  /* Any incoming interface */
echoServAddr.sin_port = htons(echoServPort);          /* Local port */

if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    Exit_if_Error("bind() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Mark the socket so it will listen for incoming connections */
   if (listen(servSock, MAXPENDING) < 0)
      Exit_if_Error("listen() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
for (;;) /* Run forever */
{
    clntLen = sizeof(echoClntAddr);
   if ((clntSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen)) < 0)
       Exit_if_Error("accept() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

Server is now blocked waiting for connection from a client
Later, a client decides to talk to the server…

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Create a reliable, stream socket using TCP */
   if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
      Exit_if_Error("socket() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
echoServAddr.sin_family      = AF_INET;      /* Internet Add family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP); /* Server IP Adrs */
echoServAddr.sin_port        = htons(echoServPort); /* Server port */
 if (connect(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
     Exit_if_Error("connect() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
if ((clntSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen)) < 0)
    DieWithError("accept() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
echoStringLen = strlen(echoString);          /* Determine input length */

/* Send the string to the server */
  if (send(sock, echoString, echoStringLen, 0) != echoStringLen)
     Exit_if_Error("send() sent a different number of bytes than expected");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Receive message from client */
if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
    Exit_if_Error("recv() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

close(sock);

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

close(clntSocket)

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

SOCK_DGRM – Connectionless Protocol

- Server
  - socket()
  - bind()
  - recvfrom()
- Client
  - socket()
  - bind()
  - sendto()

- recvfrom()
- sendto()

Connectionless Sockets – Commands

- sendto() requires the address of the destination (the server) as a parameter
- The server does not have to accept a connection
  - The server uses recvfrom() system call to receive data
    - The call causes the server to block until data is available
    - recvfrom() returns the network address of the client process along with the datagram
  - The server uses sendto() system call to send data

SOCKET INTERNALS

Protocol Control Block

- Client only sends data to {machine, port}
- How does the server keep track of simultaneous sessions to the same {**machine, port**}?
- OS maintains a structure called the **Protocol Control Block** (PCB)

Server: `svr=socket()`

Create entry in PCB table

Local Address	Local Port	Foreign Address	Foreign Port	L?

Client

Server

svr →

Local Address	Local Port	Foreign Address	Foreign Port	L?

Server: `bind(svr)`

- Assign local port and address to socket
  bind(addr=0.0.0.0, port=1234)

Client

Local Address	Local Port	Foreign Address	Foreign Port	L?

Server

svr →

Local Address	Local Port	Foreign Address	Foreign Port	L?
0.0.0.0	1234			

Server: `listen(svr, 10)`

Set socket for listening

Client

Local Address	Local port	Foreign Address	Foreign Port	L?

Server

svr →

Local Address	Local Port	Foreign Address	Foreign Port	L?
0.0.0.0	1234			*

Server: `snew=accept(svr)`

Block – wait for connection

Client

Local address	Local Port	Foreign Address	Foreign Port	L?

Server

svr ⟶

Local address	Local Port	Foreign Address	Foreign Port	L?
0.0.0.0	1234			*

Client: `s=socket()`

Create PCB entry

Local Address	Local Port	Foreign Address	Foreign Port	L?

Client

← **s**

Server

svr →

Local Address	Local Port	Foreign Address	Foreign Port	L?
0.0.0.0	**1234**			*

Client: `s=bind(s)`

Assign local port and address to socket

bind(addr=0.0.0.0, port=7801)

Local Address	Local Port	Foreign Address	Foreign Port	L?
0.0.0.0	7801			

Client

s ←

Server

svr →

Local Address	Local Port	Foreign Address	Foreign Port	L?
0.0.0.0	1234			*

Client: `connect(s)`

Send *connect* request to server

[135.250.68.3:7801] to [192.11.35.15:1234]

Local Address	Local port	Foreign Address	Foreign Port	L?
0.0.0.0	**7801**			

Client

s ←

Server

Local Address	Local Port	Foreign Address	Foreign Port	L?
0.0.0.0	**1234**			*
192.11.35.15	**1234**	**135.250.68.3**	**7801**	

svr →

snew →

Client: `connect(s)`

Server responds with acknowledgement

[192.11.35.15:1234] to [135.250.68.3 :7801]

Local Address	Local Port	Foreign Address	Foreign Port	L?
0.0.0.0	7801	192.11.35.15	1234	

Client

← **s**

Server

Local address	Local port	Foreign Address	Foreign Port	L?
0.0.0.0	**1234**			*
192.11.35.15	**1234**	**135.250.68.3**	**7801**	

svr →

snew →

Communication

- Each message from client is tagged as either *data* or *control* (e.g. *connect*)
- If data – search through table where FA and FP match incoming message and *listen*=false
- If control – search through table where *listen*=true

Server

Local Address	Local Port	Foreign Address	Foreign Port	L?
svr → **0.0.0.0**	**1234**			*
snew → **192.11.35.15**	**1234**	**135.250.68.3**	**7801**	

Conclusion

- Linux Programming Fundamentals
  - Process, Address Space, Descriptors
  - fork(), exec(), and other system calls
- Socket – Abstraction for end-to-end communication
  - Socket generic structure
  - Internet Family Socket Protocol and Address Formats
- Stream Socket Example – Echo Server
- Connectionless Server
- Socket Internals