Introduction to Communication - Sockets

David E. Culler

CS162 – Operating Systems and Systems Programming

Lecture 8

Sept 24, 2019

HW 2 out, Due 10/12
Proj 1

Producer/Consumer Problem

- With multiple threads, each waits for the other to make process
- Scheduling constraints:
  – Consumer waits for producer if buffer is empty
  – Producer waits for consumer if buffer is full
- Mutual Exclusion: Only one thread manipulates the buffer data structure at a time

Condition Variables

- Collection of threads waiting *inside* a critical section
- Operations:
  - **`wait(&lock)`:** Atomically release lock and go to sleep. Re-acquire the lock before returning.
  - **`signal()`:** Wake up one waiting thread (if there is one)
  - **`broadcast()`:** Wake up all waiting threads

- **Rule:** Hold lock when using a condition variable

Producer/Consumer Buffer – 2nd cut

```
mutex buf_lock = <initially unlocked>
Condvar buf_signal = <initially nobody>
```

```
Producer(item) {
  lock buffer
  while (buffer full) { cond_wait(buf_signal, buf_lock) };
  Enqueue(item);
  cond_broadcast(buf_signal);
  unlock buffer
}
```

```
Consumer() {
  lock buffer
  while (buffer empty) { cond_wait(b
  item = queue();
  cond_broadcast(buf_signal, buf_lock);
  unlock buffer
  return item
}
```

Release lock; signal others to run; reacquire on resume

n.b. OS must do the reacquire

Why User must recheck?

Why the `while` Loop?

- When a thread is woken up by **`signal()`** or **`broadcast()`**, it is simply put on the ready queue
- It may or may not reacquire the lock immediately!
  - Another thread could be scheduled first and "sneak in" to empty the queue
  - Need a loop to re-check condition on wakeup

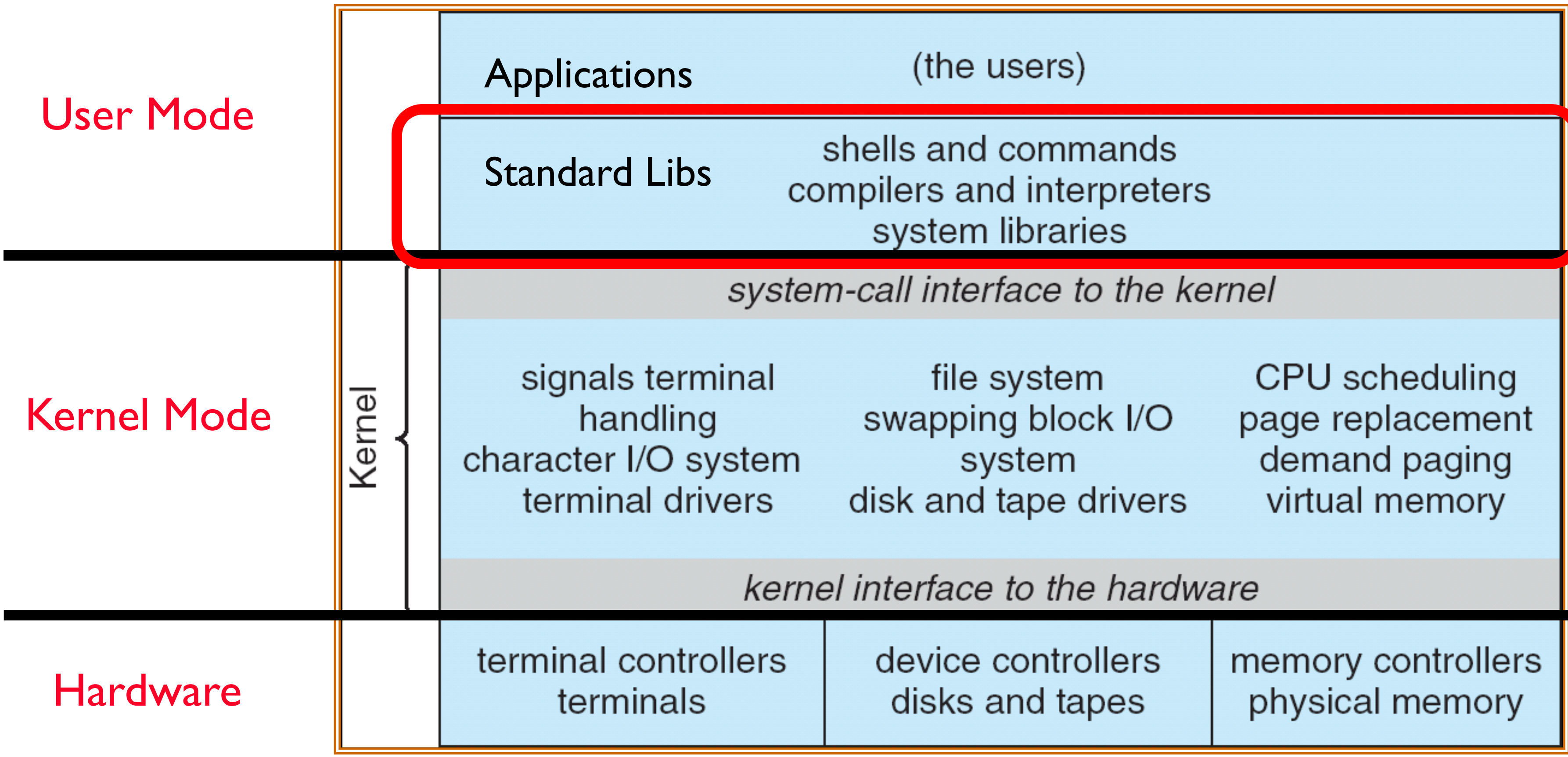
Recall: Semaphore Solution

```
Semaphore fullSlots = 0; // Queue empty to start
Semaphore emptySlots = bufSize; // All slots empty
Semaphore mutex = 1; // No one in critical sect.
```

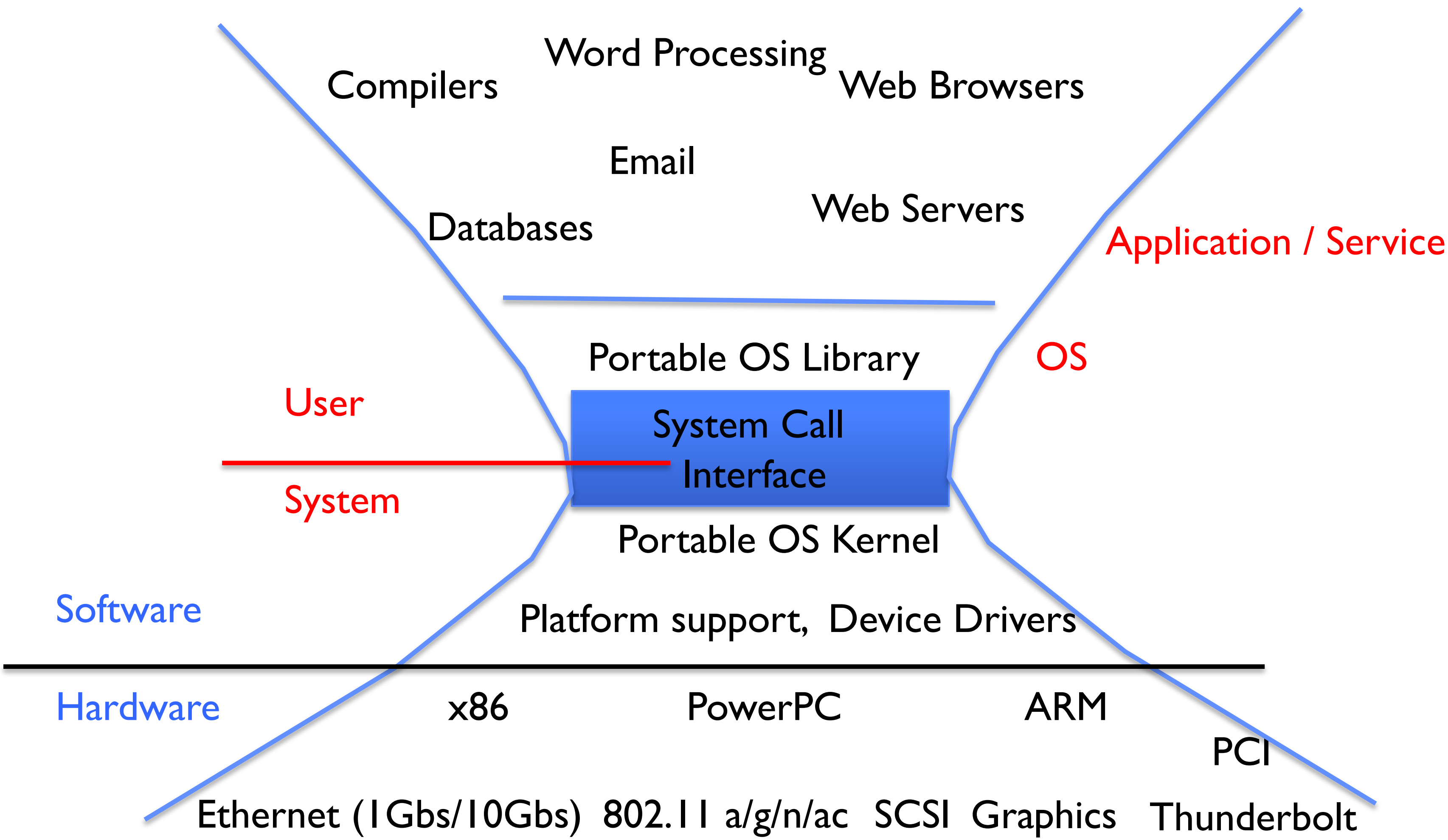
```
Producer(item) {
  emptySlots.P();
  mutex.P();
  Enqueue(item);
  mutex.V();
  fullSlots.V();
}
```

```
Consumer() {
  fullSlots.P();
  mutex.P();
  item = Dequeue();
  mutex.V();
  emptySlots.V();
  return item;
}
```

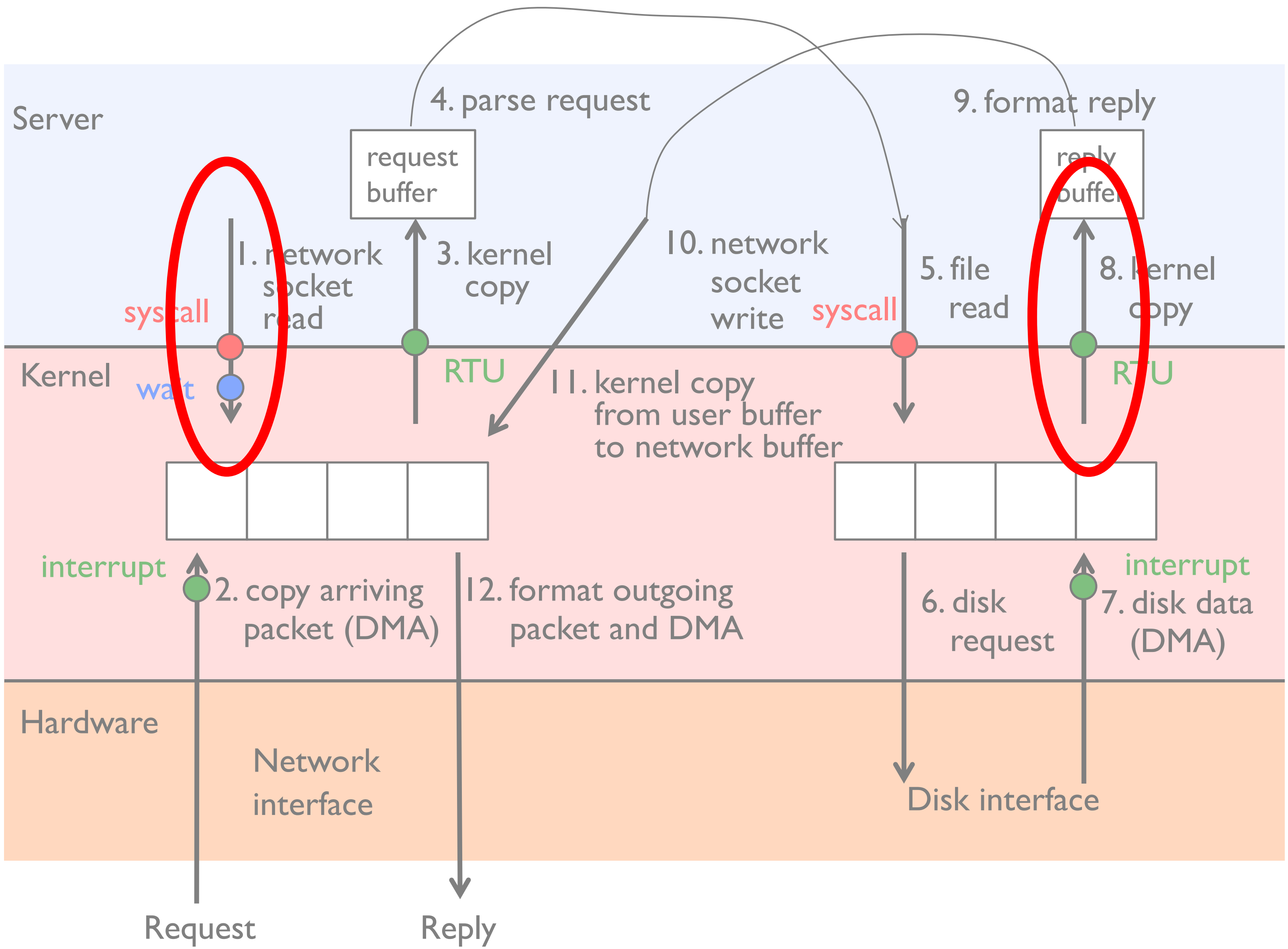
Recall: UNIX System Structure



A Kind of Narrow Waist



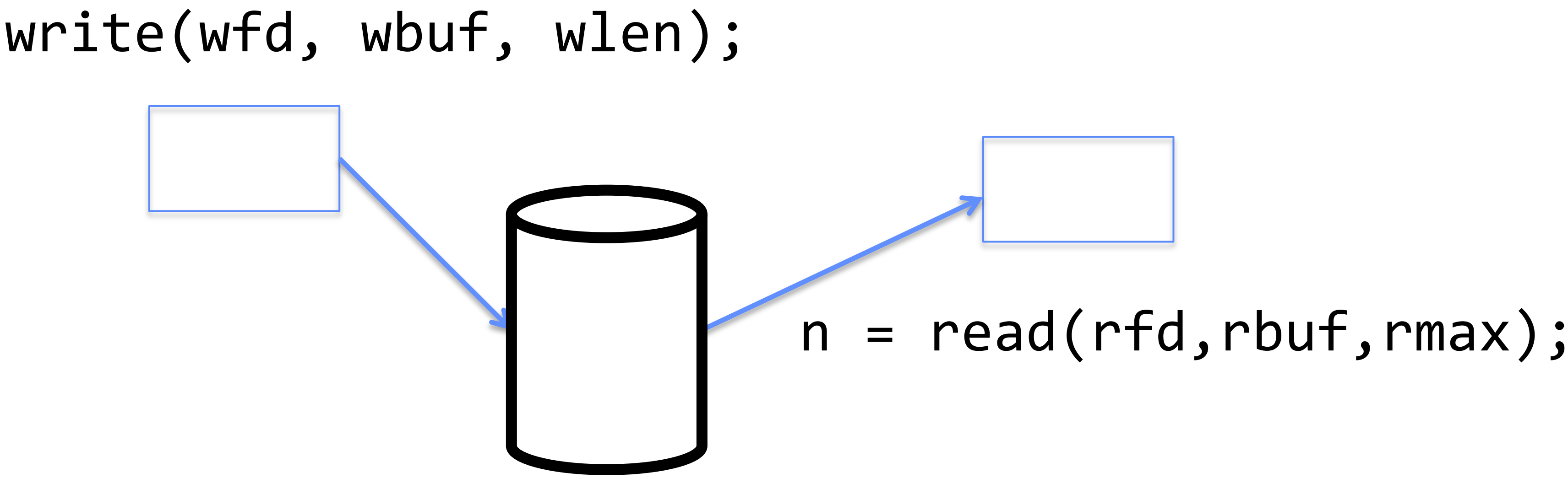
Putting it together: web server



Recall: 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

How can a process communicate with another?

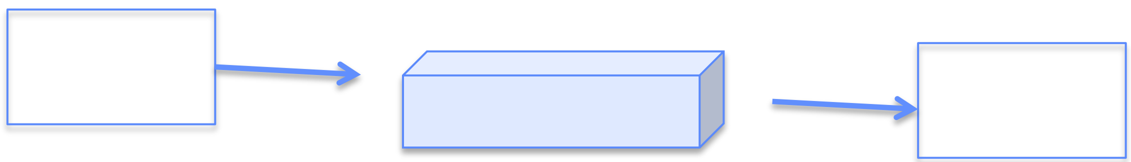


- Producer and Consumer of a file may be distinct processes
  - Also separated in time (one writes and then one later reads)
- However, what if data written once and consumed once?
  - Don't we want something more like a queue?
  - Can still look like File I/O!

Communication between processes

- Can we view files as communication channels?

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



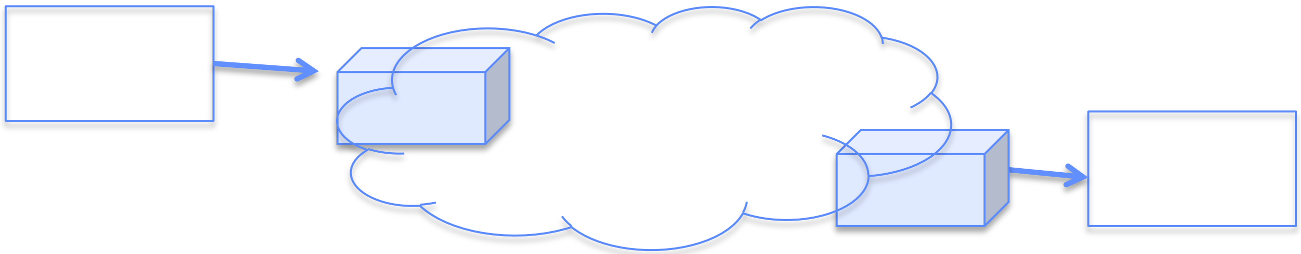
```
n = read(rfd,rbuf,rmax);
```

- We have seen one example – pipes
- Routinely used with the shell

```
>>>  grep list src/*/*.c | more
```

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?

What Is A Protocol?

- A protocol is an agreement on how to communicate
- Includes
  - Syntax: how a communication is specified & structured
    - » Format, order messages are sent and received
  - Semantics: what a communication means
    - » Actions taken when transmitting, receiving, or when a timer expires
- Described formally by a state machine
  - Often represented as a message transaction diagram

Examples of Protocols in Human Interactions

- Telephone
  1. (Pick up / open up the phone)
  2. Listen for a dial tone / see that you have service
  3. Dial
  4. Should hear ringing …
  5. Callee: "Hello?"
  6. Caller: "Hi, it's John…."
     Or: "Hi, it's me" (← what's *that* about?)
  7. Caller: "Hey, do you think … blah blah blah …" **pause**

  1. Callee: "Yeah, blah blah blah …" **pause**
  2. Caller: Bye
  3. Callee: Bye
  4. Hang up

Examples of Protocols in Human Interactions

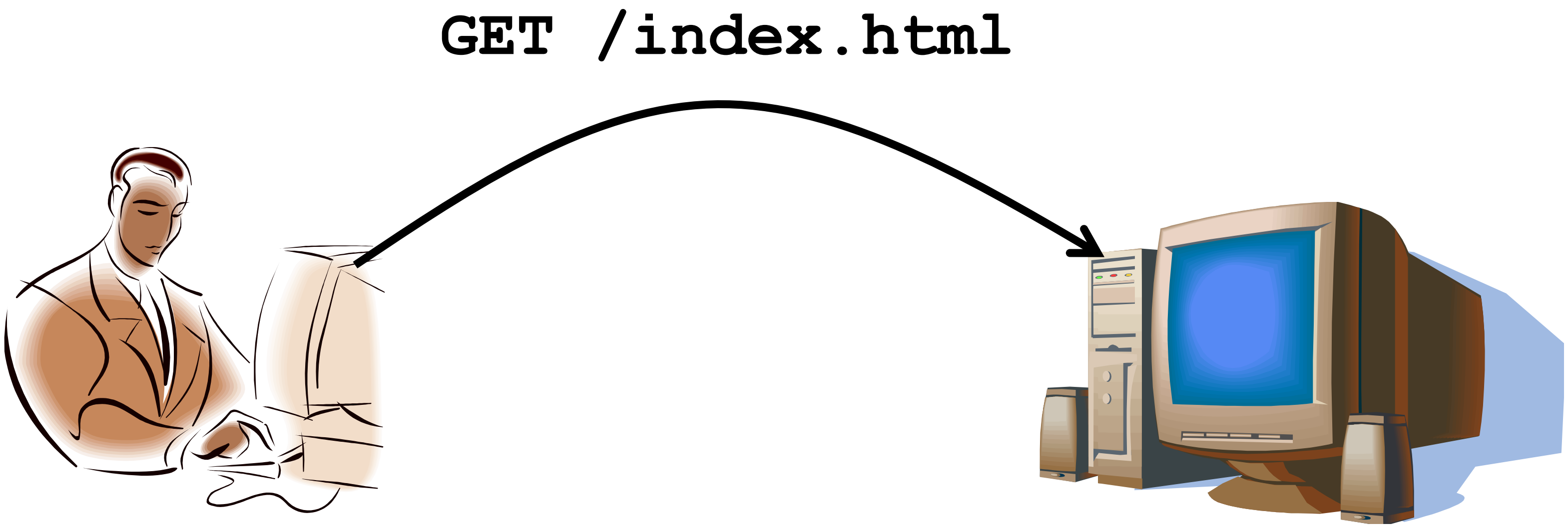
Asking a question

1. Raise your hand
2. Wait to be called on

3. Or: wait for speaker to **pause** and vocalize

Clients and Servers

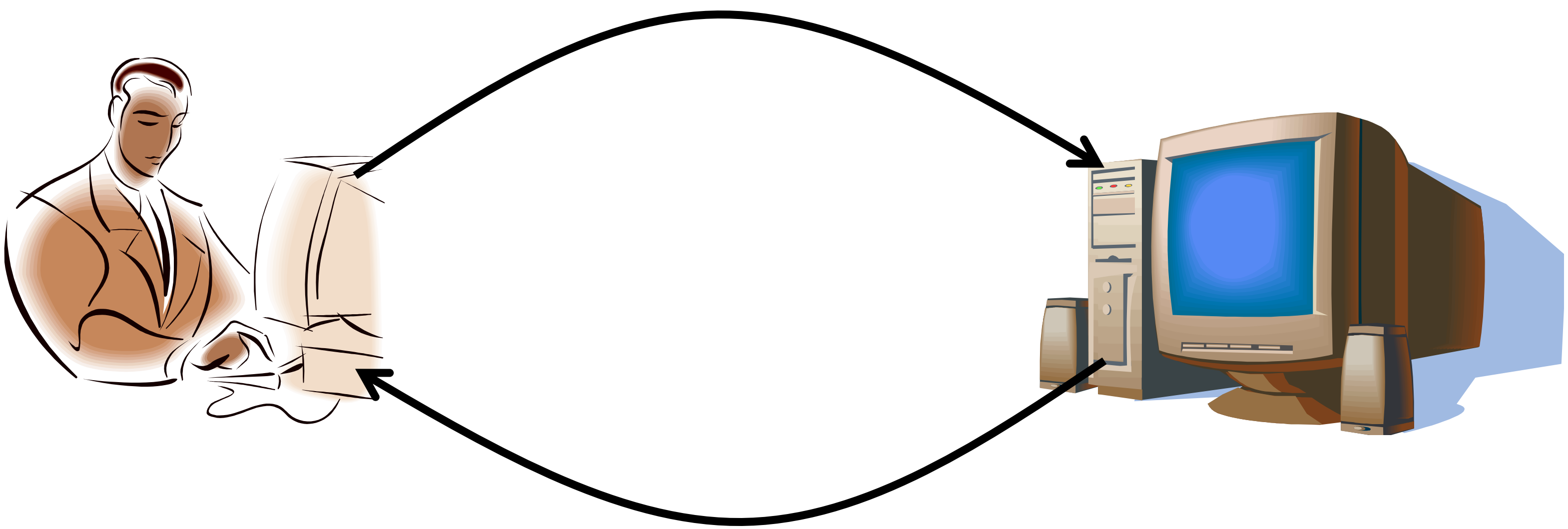
- Client program
  - Running on end host
  - Requests service
  - E.g., Web browser



Clients and Servers

- Client program
  - Running on end host
  - Requests service
  - E.g., Web browser
- Server program
  - Running on end host
  - Provides service
  - E.g., Web server

`GET /index.html`

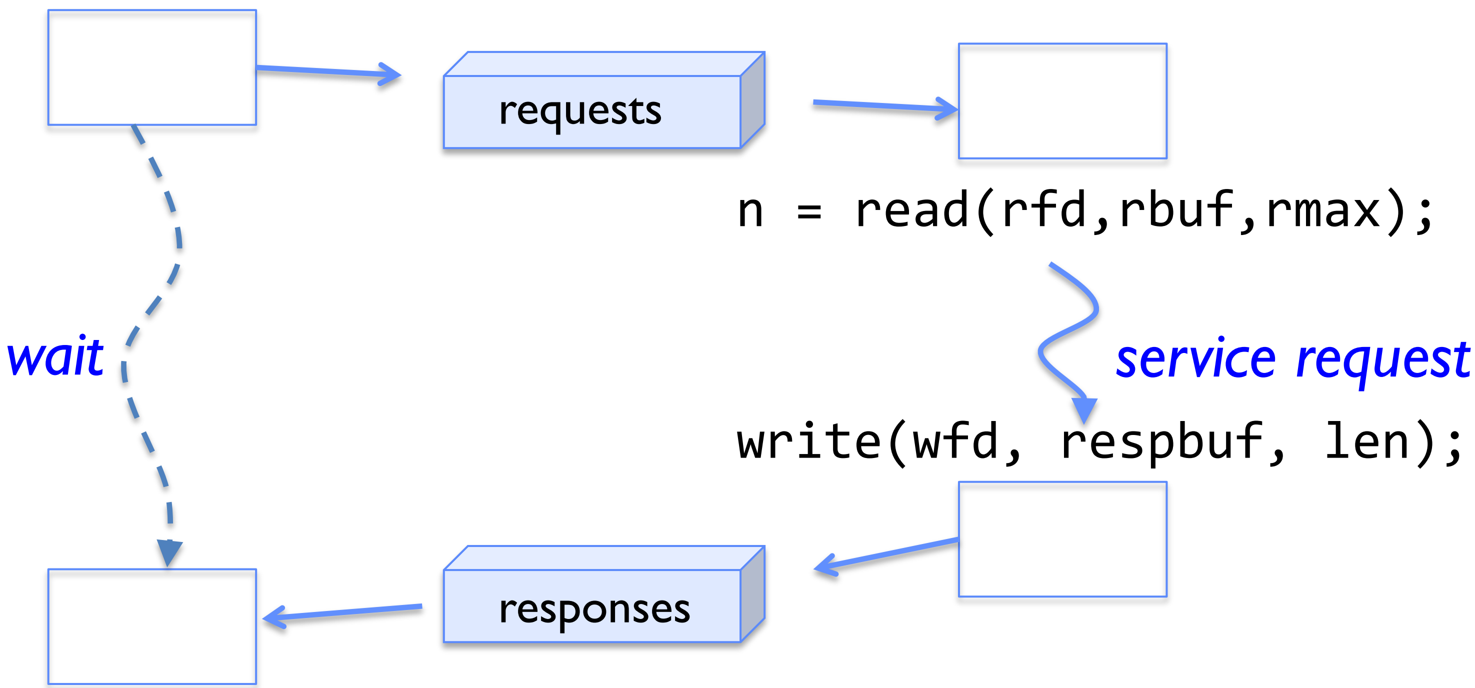


“`Site under construction`”

Request Response Protocol

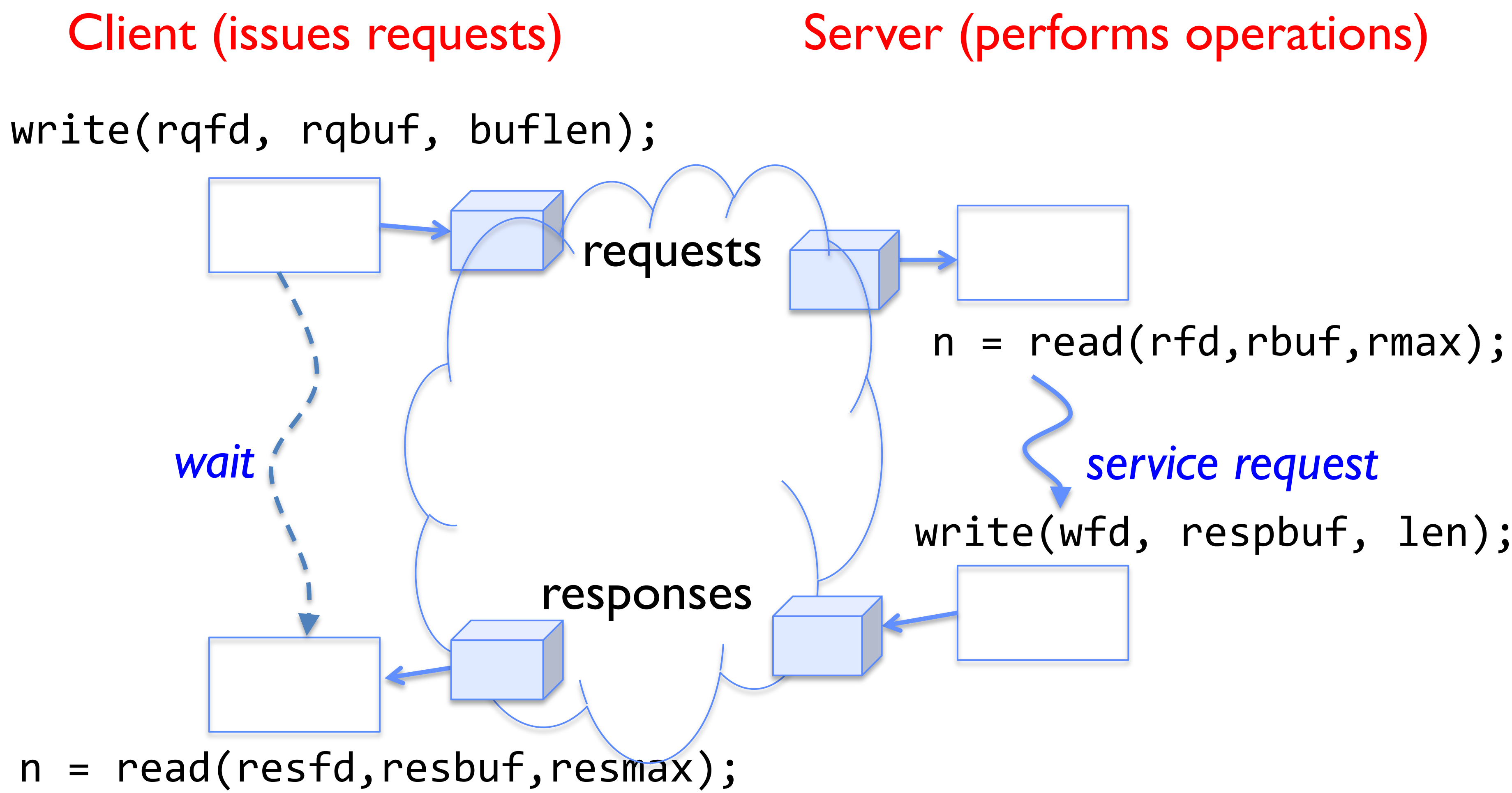
Client (issues requests) | Server (performs operations)

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

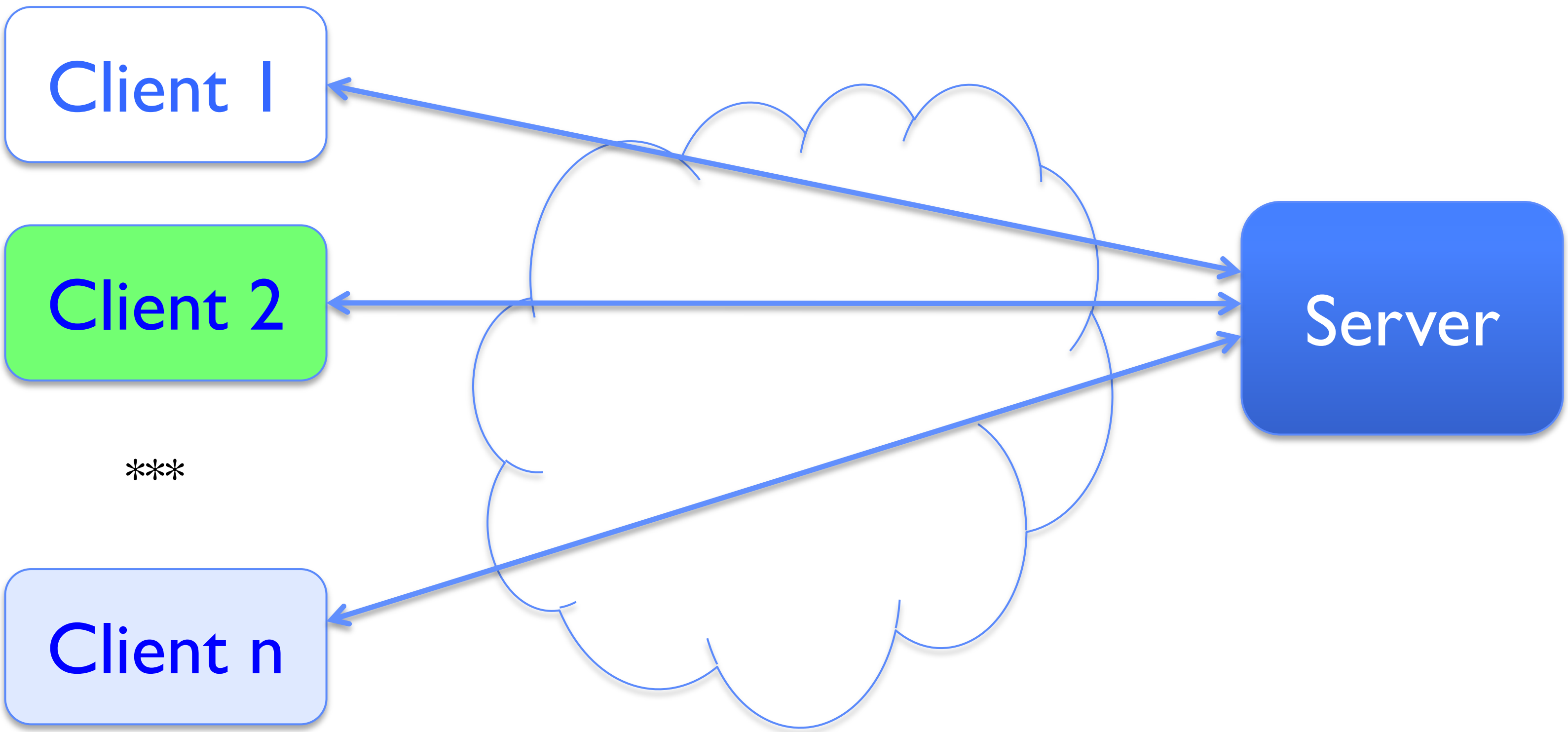



```
n = read(resfd,resbuf,resmax);
```

Request Response Protocol



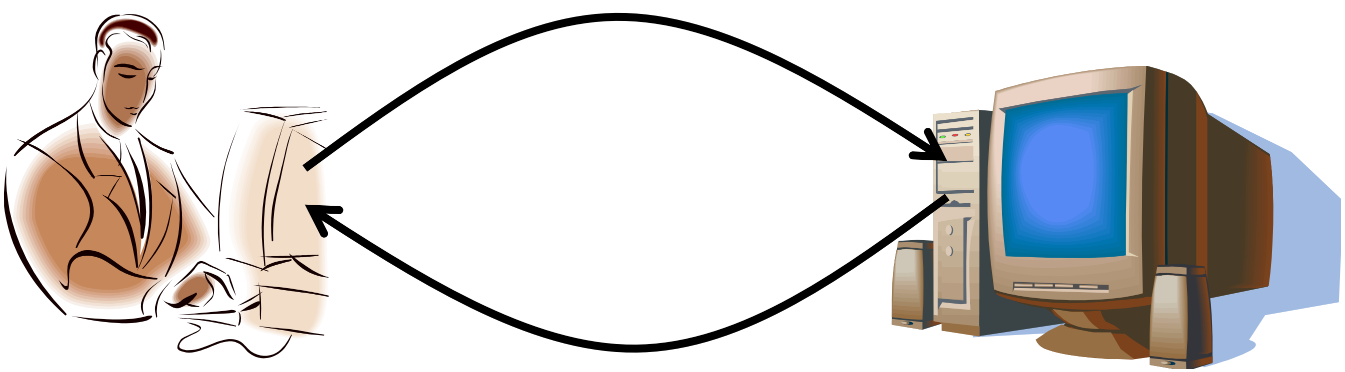
Client-Server Models



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

Client-Server Communication

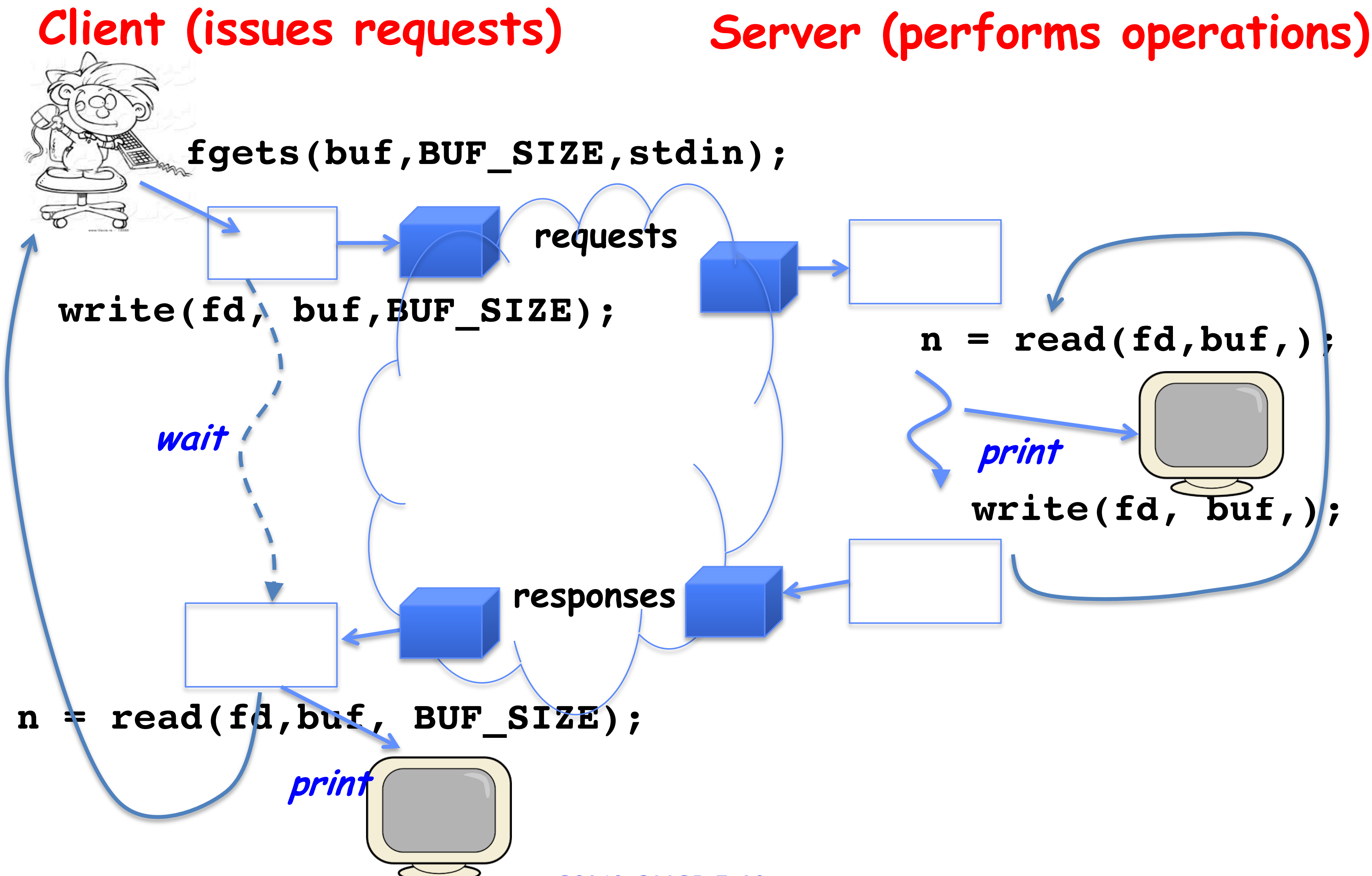
- Client "sometimes on"
  - Initiates a request to the server when interested
  - E.g., Web browser on your laptop or cell phone
  - Doesn't communicate directly with other clients
  - Needs to know the server's address
- Server is "always on"
  - Services requests from many client hosts
  - E.g., Web server for the *www.cnn.com* Web site
  - Doesn't initiate contact with the clients
  - Needs a fixed, well-known address



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

client

```
char buf[BUF_SIZE];
fgets(buf, BUF_SIZE, stdin);          // prompt
write(sockfd, buf, strlen(sndbuf));   // send request
memset(buf, 0, BUF_SIZE);             // clear
read(sockfd, buf, BUF_SIZE-1);        // receive response
printf("%s\n", buf);                  // echo
```

server

```
char buf[BUF_SIZE];
memset(buf, 0, BUF_SIZE);
read(consockfd,reqbuf,MAXREQ-1);    // receive
printf("%s\n", buf);                // echo
write(consockfd, buf, strlen(reqbuf));      // send response
```

What assumptions are we making?

- Reliable
  - Write to a file => Read it back. Nothing is lost.
  - Write to a (TCP) socket => Read from the other side, same.
  - Like pipes
- In order (sequential stream)
  - Write X then write Y => read gets X then read gets Y
- When ready?
  - File read gets whatever is there at the time. Assumes writing already took place.
  - Like pipes!

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
- How do we name them?
- How do these completely independent programs know that the other wants to "talk" to them?

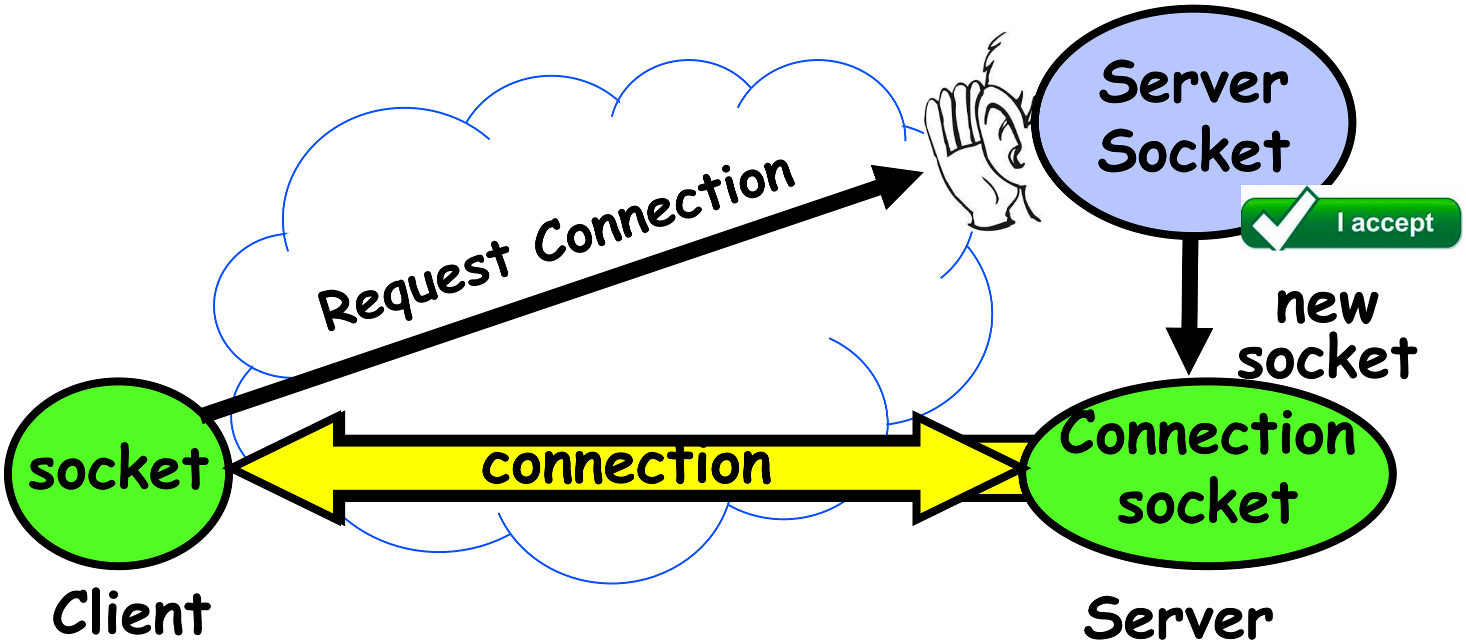
Namespaces for communication over IP

- Hostname
  - www.eecs.berkeley.edu
- IP address
  - 128.32.244.172 (IPv4, 32-bit Integer)
  - 2607:f140:0:81::f (IPv6, 128-bit Integer)
- 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"

How do they “shake hands”?

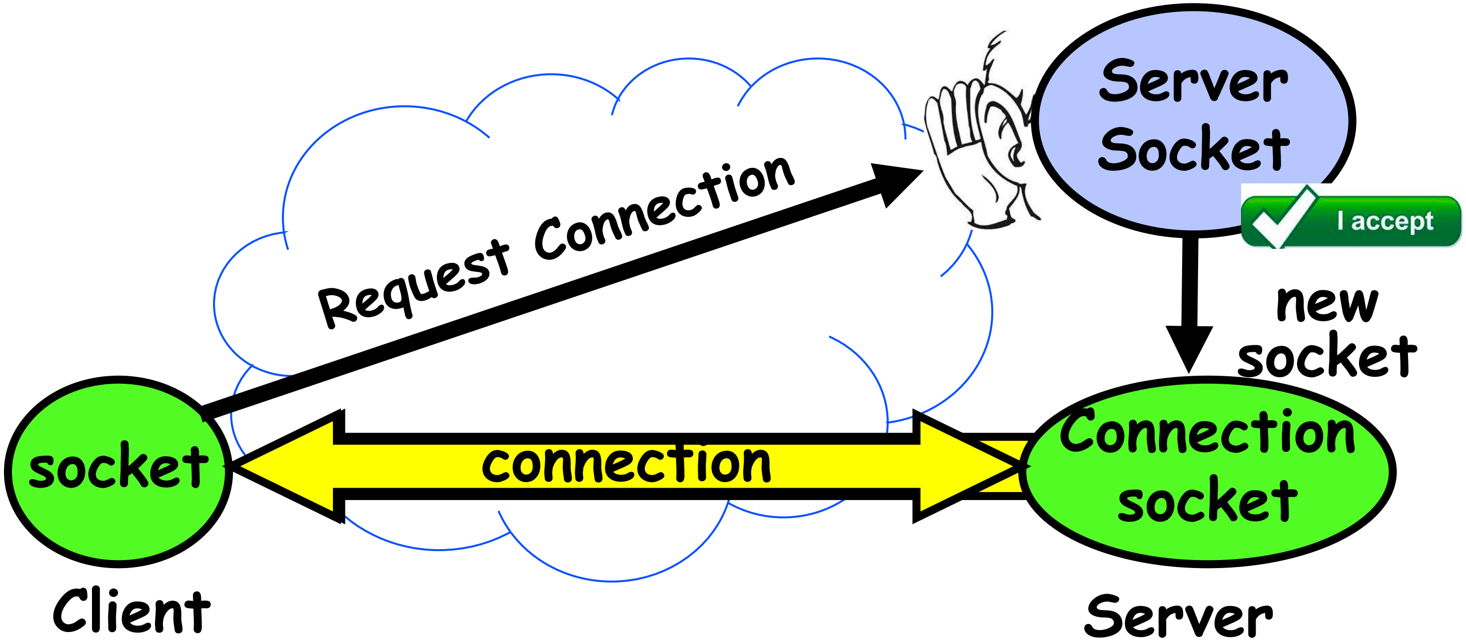
- How does the server know that a client wants to make a request of them?
- How does a client know that the server is accepting requests?

Socket Setup over TCP/IP



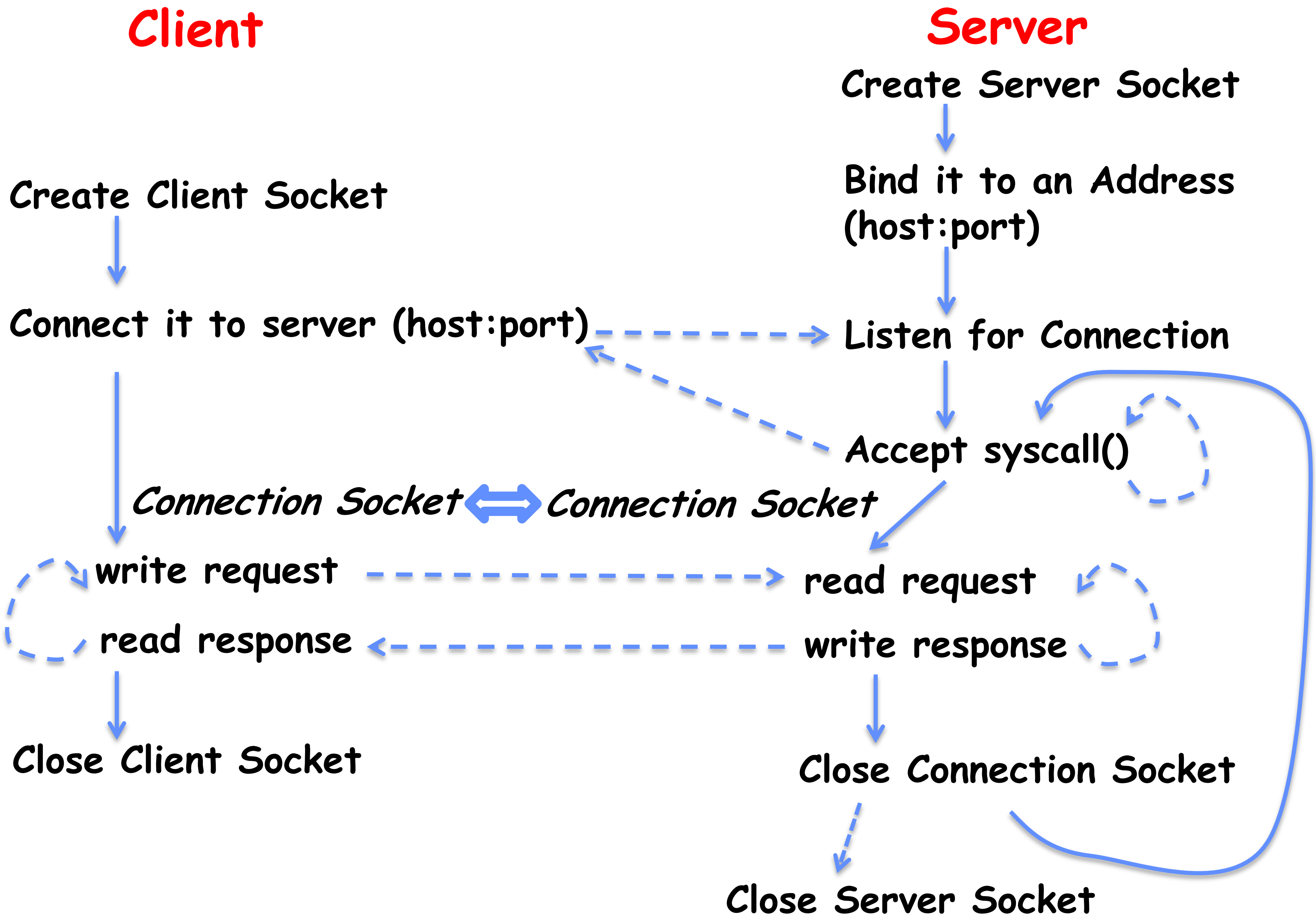
- Special kind of socket: **server socket**
  - Has file descriptor
  - Can't read or write
- Two operations:
  1. **`listen()`**: Start allowing clients to connect
  2. **`accept()`**: Create a *new socket* for a *particular* client connection

Socket Setup over TCP/IP



- 5-Tuple identifies each connection:
  1. Source IP Address
  2. Destination IP Address
  3. Source Port Number
  4. Destination Port Number
  5. Protocol (always TCP here)
- 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 Schematic



Client Protocol

```
char *host_name, port_name;

// Create a socket
struct addrinfo *server = lookup_host(host_name, port_name);
int sock_fd = socket(server->ai_family, server->ai_socktype,
                     server->ai_protocol);

// Connect to specified host and port
connect(sock_fd, server->ai_addr, server->ai_addrlen);

// Carry out Client-Server protocol
run_client(sock_fd);

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

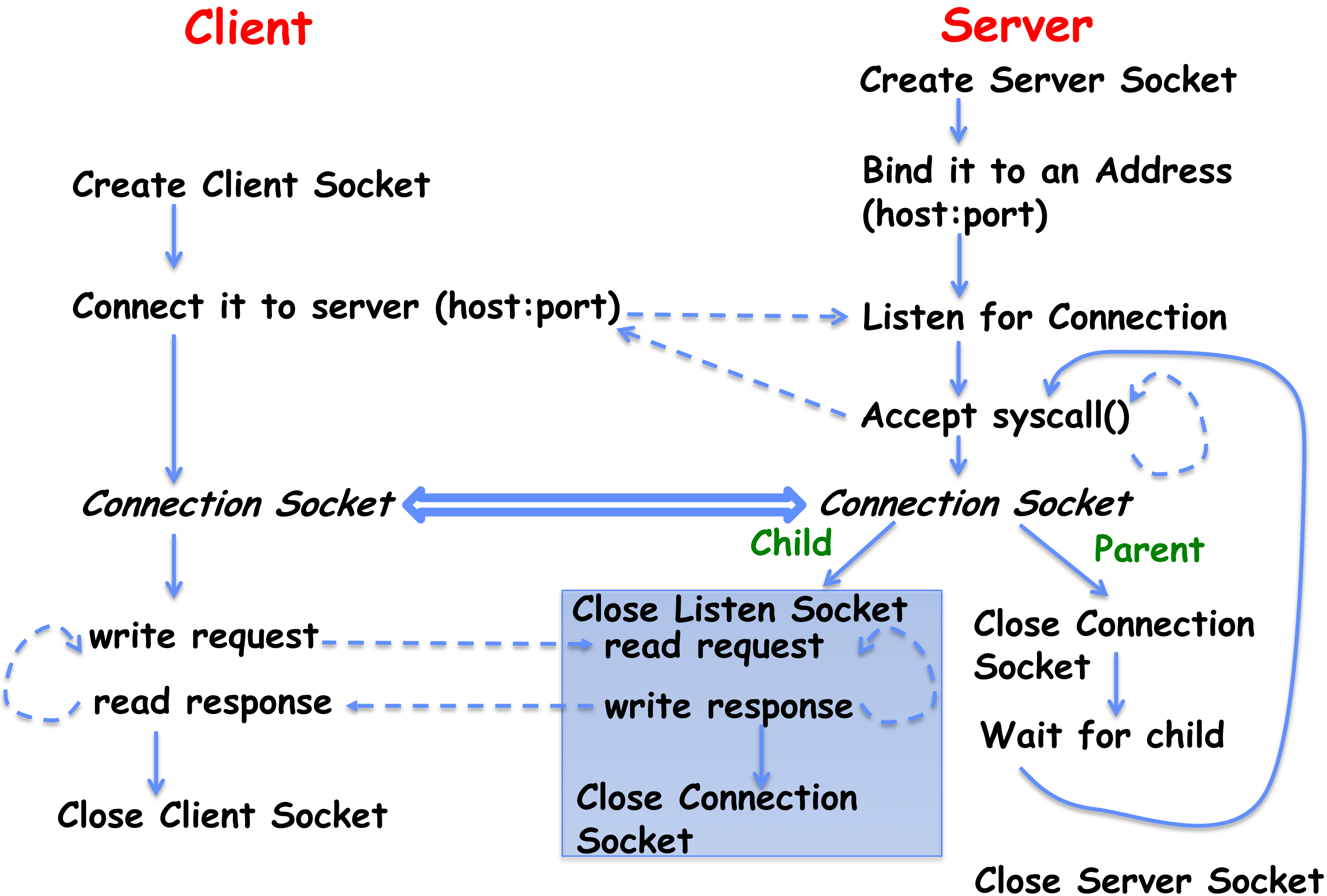
Server Protocol (v1)

```
// Create socket to listen for client connections
char *port_name;
struct addrinfo *server = setup_address(port_name);
int server_socket = socket(server->ai_family,
      server->ai_socktype, server->ai_protocol);

// Bind socket to specific port
bind(server_socket, server->ai_addr, server->ai_addrlen);
// Start listening for new client connections
listen(server_socket, MAX_QUEUE);

while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  serve_client(conn_socket);
  close(conn_socket);
}

close(server_socket);
```

How does the server protect itself?

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

Sockets With Protection



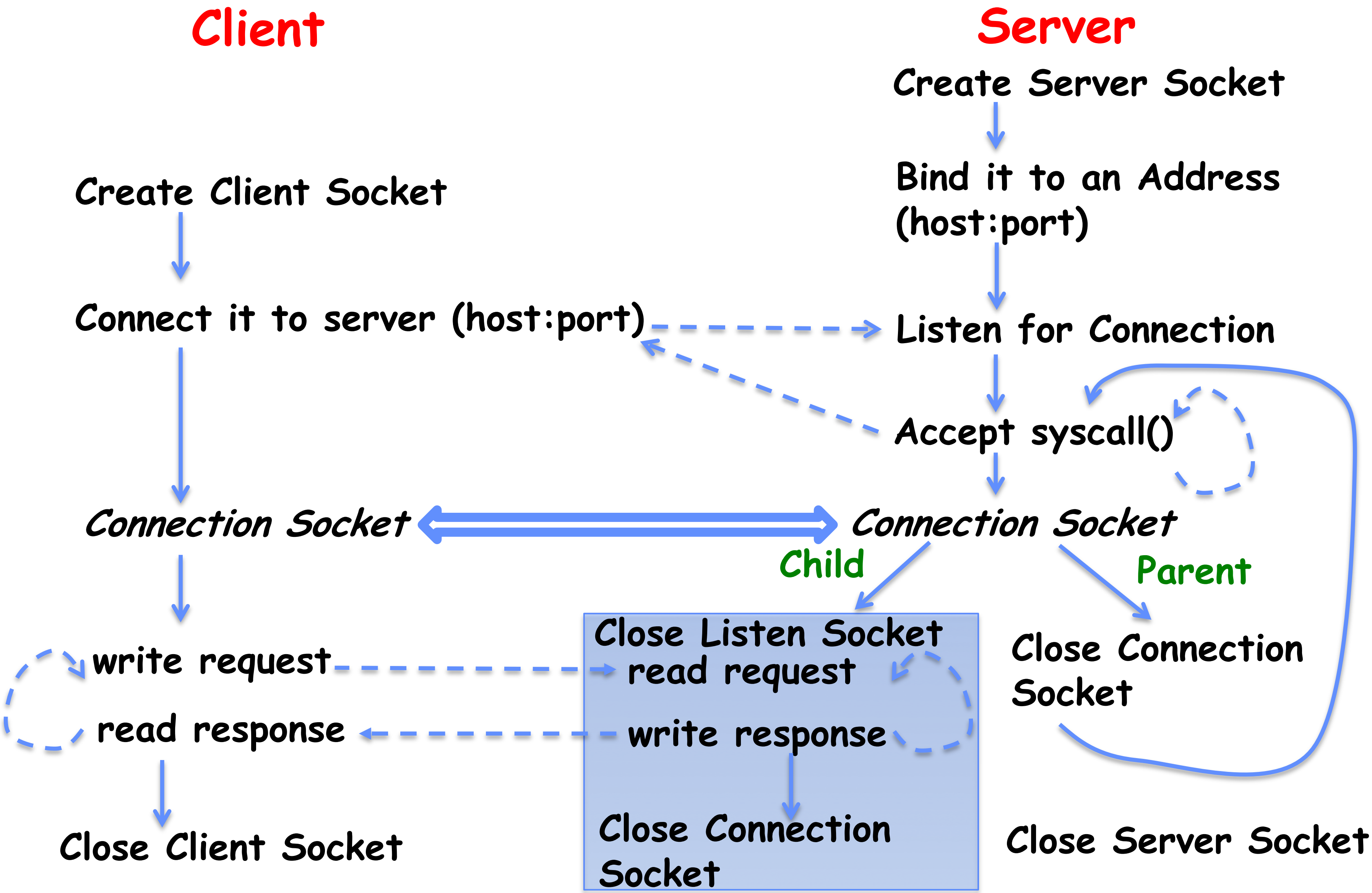
Server Protocol (v2)

```
// Socket setup code elided…

while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  pid_t pid = fork();
  if (pid == 0) {
    close(server_socket);
    serve_client(conn_socket);
    close(conn_socket);
    exit(0);
  } else {
    close(conn_socket);
    wait(NULL);
  }
}

close(server_socket);
```

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



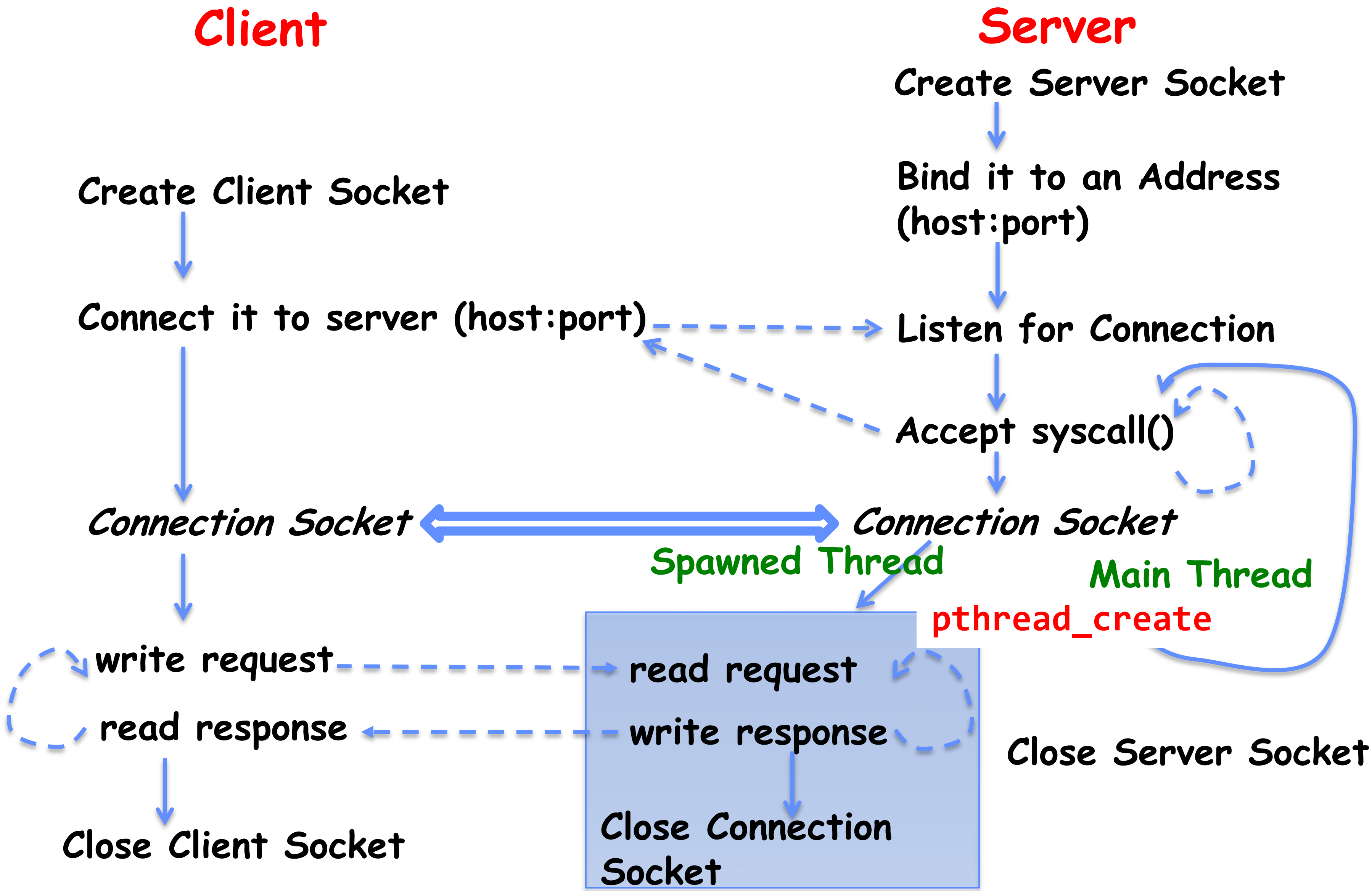
Server Protocol (v3)

```
// Socket setup code elided…

while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  pid_t pid = fork();
  if (pid == 0) {
    close(server_socket);
    serve_client(conn_socket);
    close(conn_socket);
    exit(0);
  } else {
    close(conn_socket);
    //wait(NULL);
  }
}

close(server_socket);
```

Concurrent Server without Protection

- Spawn a new thread to handle each connection
- Main thread initiates new client connections without waiting for previously spawned threads
- Why give up the protection of separate processes?
  – More efficient to create new threads
  – More efficient to switch between threads

Sockets With Parallelism, Without Protection



Server Address - itself

```
struct addrinfo *setup_address(char *port) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;
  getaddrinfo(NULL, port, &hints, &server);
  return server;
}
```

- Simple form
- Internet Protocol, TCP
- Accepting any connections on the specified port

Client: getting the server address

```
struct addrinfo *lookup_host(char *host_name, char *port) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;

  int rv = getaddrinfo(host_name, port_name,
                       &hints, &server);
  if (rv != 0) {
    printf("getaddrinfo failed: %s\n", gai_strerror(rv));
    return NULL;
  }
  return server;
}
```

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 …`

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