Operating Systems
(Honor Track)

Abstraction 3: IPC, Pipes and Sockets
# A quick, programmer's viewpoint

Xin Jin

Spring 2026

Recap: Key Unix I/O Design Concepts

- Uniformity – everything is a file
  - 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 yielding processor to other task
- Kernel buffered writes
  - Completion of out-going transfer decoupled from the application, allowing it to continue
- Explicit close

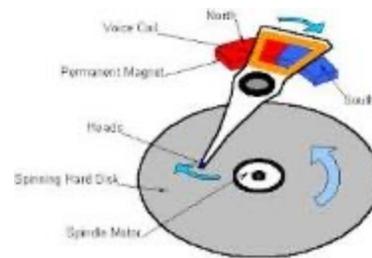# Recap: I/O and Storage Layers

Application / Service
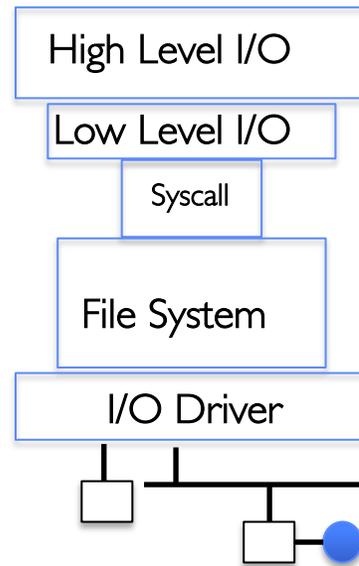
High Level I/O	*Streams*
Low Level I/O	*File Descriptors*
	open(), read(), write(), close(), …
Syscall	
	Open File Descriptions
File System	*Files/Directories/Indexes*
I/O Driver	*Commands and Data Transfers*
	Disks, Flash, Controllers, DMA

Recap: C High-Level File API

```
// 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, ... );
```

Recap: C High-Level File API: Positioning The Pointer

```
int fseek(FILE *stream, long int offset, int whence); // Reposition stream
position indicator
long int ftell (FILE *stream) // Get current position in stream
void rewind (FILE *stream) // Set position of stream to the beginning
```

- For **fseek()**, the **offset** is interpreted based on the **whence** argument (constants in **stdio.h**):
  - **SEEK_SET**: Then offset interpreted from beginning (position 0)
  - **SEEK_END**: Then offset interpreted backwards from end of file
  - **SEEK_CUR**: Then offset interpreted from current position

offset (SEEK_SET)   offset (SEEK_END)

whence

offset (SEEK_CUR)
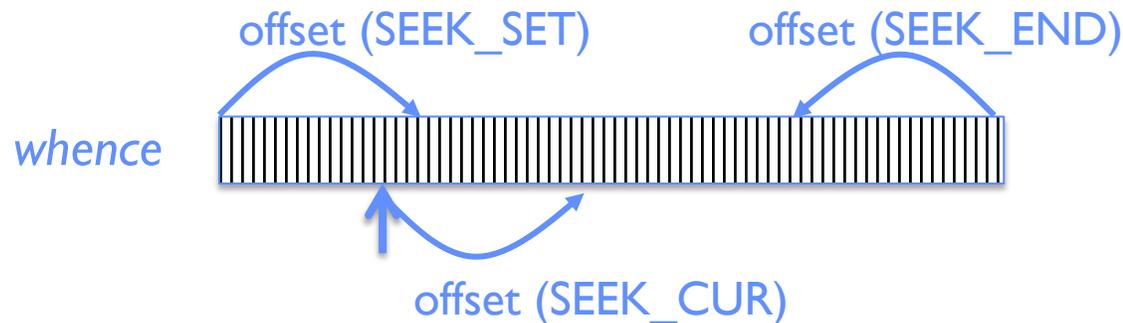
- Overall preserves high-level abstraction of a uniform stream of objects

Recap: Low-Level File API

- Read data from open file using file descriptor:

  ```
  ssize_t read (int filedes, void *buffer, size_t maxsize)
  ```

  - Reads up to `maxsize` bytes – **might actually read less!**
  - returns bytes read, 0 => EOF, -1 => error

- Write data to open file using file descriptor

  ```
  ssize_t write (int filedes, const void *buffer, size_t size)
  ```

  - returns number of bytes written

- Reposition file offset within kernel (this is independent of any position held by high-level FILE descriptor for this file!)

  ```
  off_t lseek (int filedes, off_t offset, int whence)
  ```

Recap: High-Level vs. Low-Level File API

High-Level Operation:

    size_t fread(…) {

      Do some work like a normal fn…

    asm code … syscall # into %eax
    put args into registers %ebx, …
    special trap instruction

> Kernel:
>   get args from regs
>   dispatch to system func
>   Do the work to read from the file
>   Store return value in %eax

    get return values from regs
    Do some more work like a normal fn…
    };

Low-Level Operation:

    ssize_t read(…) {

    asm code … syscall # into %eax
    put args into registers %ebx, …
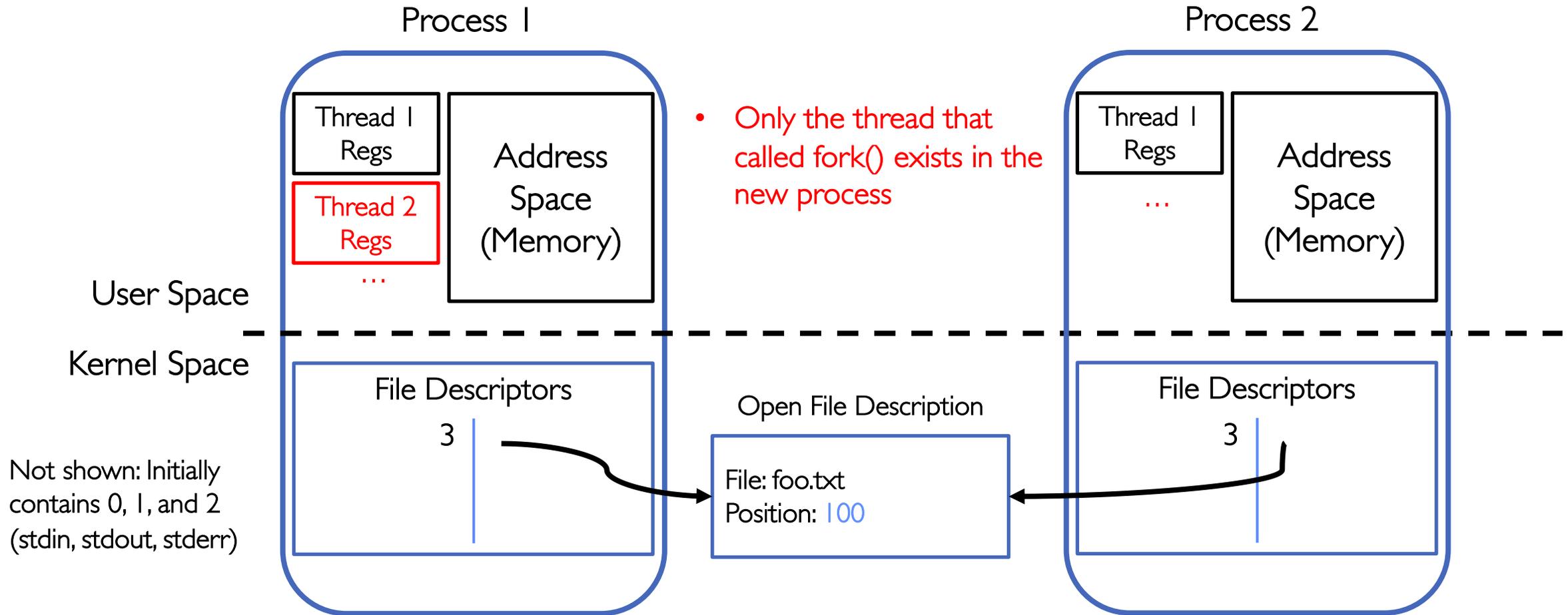   *special trap instruction*

> Kernel:
>   get args from regs
>   dispatch to system func
>   Do the work to read from the file
>   Store return value in %eax

    get return values from regs

    };

Recap: `fork()` in a Multithreaded Processes

Process 1

Thread 1 Regs

Thread 2 Regs

...

Address Space (Memory)

- Only the thread that called fork() exists in the new process

Process 2

Thread 1 Regs

...

Address Space (Memory)

User Space

Kernel Space

File Descriptors

3 |

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

Open File Description

File: foo.txt
Position: 100

File Descriptors

3 |

Recap: Avoid Mixing FILE* and File Descriptors

```
char x[10];
char y[10];
FILE* f = fopen("foo.txt", "rb");
int fd = fileno(f);
fread(x, 10, 1, f); // read 10 bytes from f
read(fd, y, 10); // assumes that this returns data starting at offset 10
```

- Which bytes from the file are read into y?
  - A. Bytes 0 to 9
  - B. Bytes 10 to 19
  - C. None of these?
- Answer: C! None of the above.
  - The `fread()` reads a big chunk of file into user-level buffer
  - Might be all of the file!

IPC and Sockets

- **Key Idea:** Communication between processes and across the world looks like File I/O
- Introduce Pipes and Sockets
- Introduce TCP/IP Connection setup for Webserver


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

Process ⟷ Socket ⟷ Socket ⟷ Process
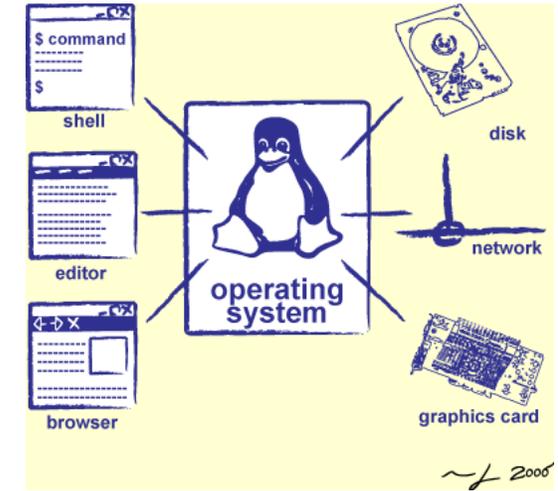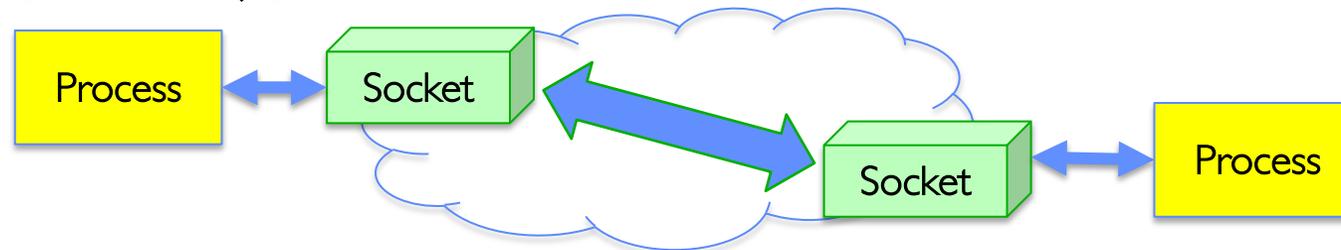
```
n = read(rfd, rbuf, rmax);
```

Communication Between Processes

- What if processes wish to communicate with one another?
  - Why? Shared Task, Cooperative Venture with Security Implications
- Process Abstraction Designed to Discourage Inter-Process Communication!
  - Prevent one process from interfering with/stealing information from another
- So, must do something special (and agreed upon by both processes)
  - Must "Punch Hole" in security
- This is called "Interprocess Communication" (or IPC)

Communication Between Processes

- Producer (writer) and consumer (reader) may be distinct processes
  - Potentially separated in time
  - How to allow selective communication?

- Simple option: use a file!
  - We have already shown how parents and children share file descriptions:
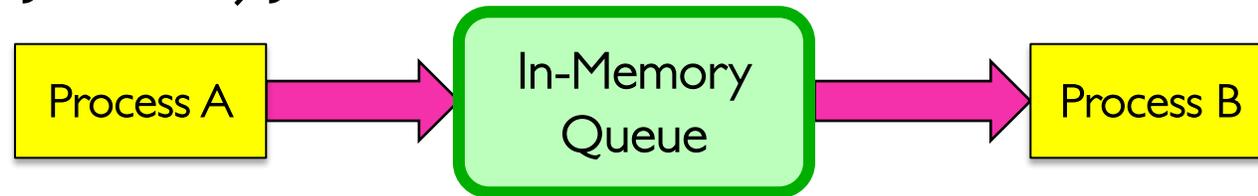
```
write(wfd, wbuf, wlen);
```
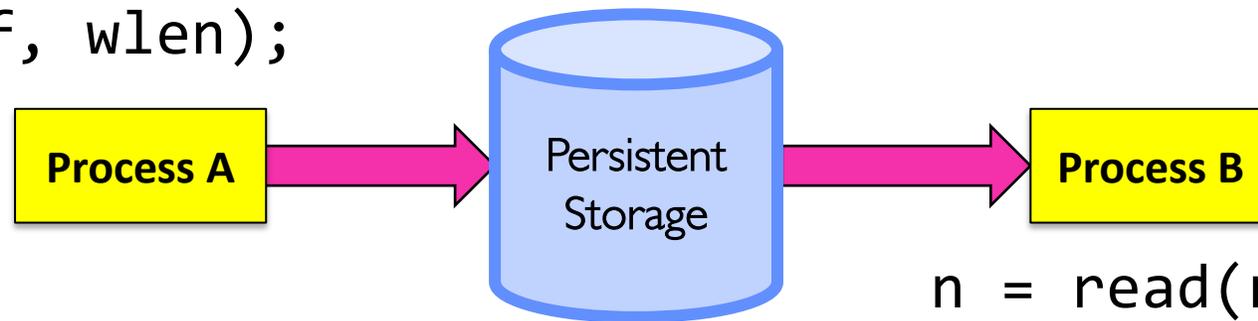


```
n = read(rfd, rbuf, rmax);
```

- Why might this be wasteful?
  - Very expensive if you only want transient communication (non-persistent)

Communication Between Processes

- Suppose we ask Kernel to help?
  - Consider an in-memory queue
  - Accessed via system calls (for security reasons)

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



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

- Data written by A is held in memory until B reads it
  - Same interface as we use for files!
  - Internally more efficient, since nothing goes to disk
- Some questions:
  - How to set up?
  - What if A generates data faster than B can consume it?
  - What if B consumes data faster than A can produce it?

One example of this pattern: POSIX/Unix PIPE

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



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

- Memory Buffer is finite:
  – If producer (A) tries to write when buffer full, it *blocks* (Put sleep until space)
  – If consumer (B) tries to read when buffer empty, it *blocks* (Put to sleep until data)

int pipe(int fileds[2]);
  – Allocates two new file descriptors in the process
  – Writes to `fileds[1]` read from `fileds[0]`
  – Implemented as a fixed-size queue
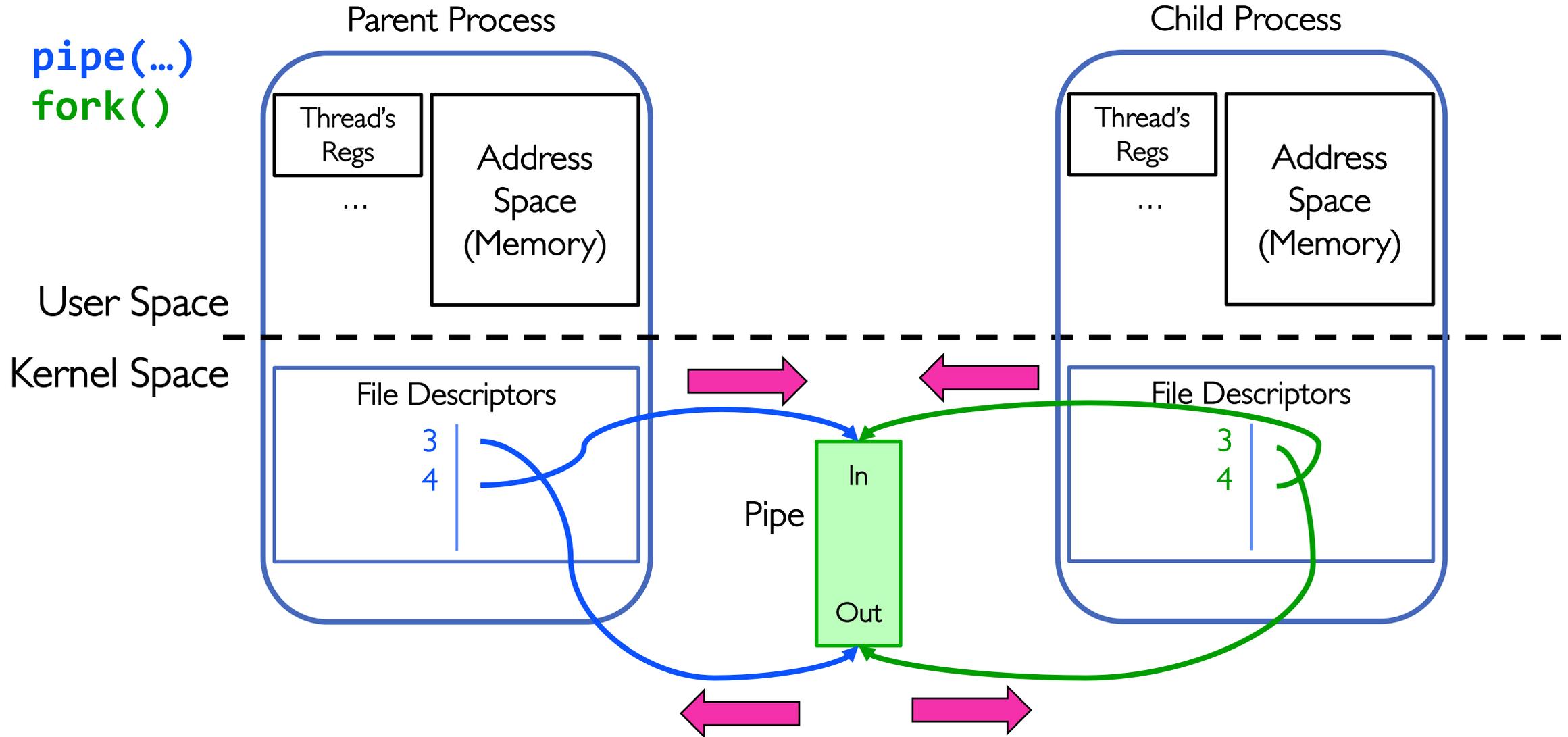
Single-Process Pipe Example

```c
#include <unistd.h>
int main(int argc, char *argv[])
{
  char *msg = "Message in a pipe.\n";
  char buf[BUFSIZE];
  int pipe_fd[2];
  if (pipe(pipe_fd) == -1) {
    fprintf (stderr, "Pipe failed.\n"); return EXIT_FAILURE;
  }
  ssize_t writelen = write(pipe_fd[1], msg, strlen(msg)+1);
  printf("Sent: %s [%ld, %ld]\n", msg, strlen(msg)+1, writelen);

  ssize_t readlen  = read(pipe_fd[0], buf, BUFSIZE);
  printf("Rcvd: %s [%ld]\n", buf, readlen);

  close(pipe_fd[0]);
  close(pipe_fd[1]);
}
```

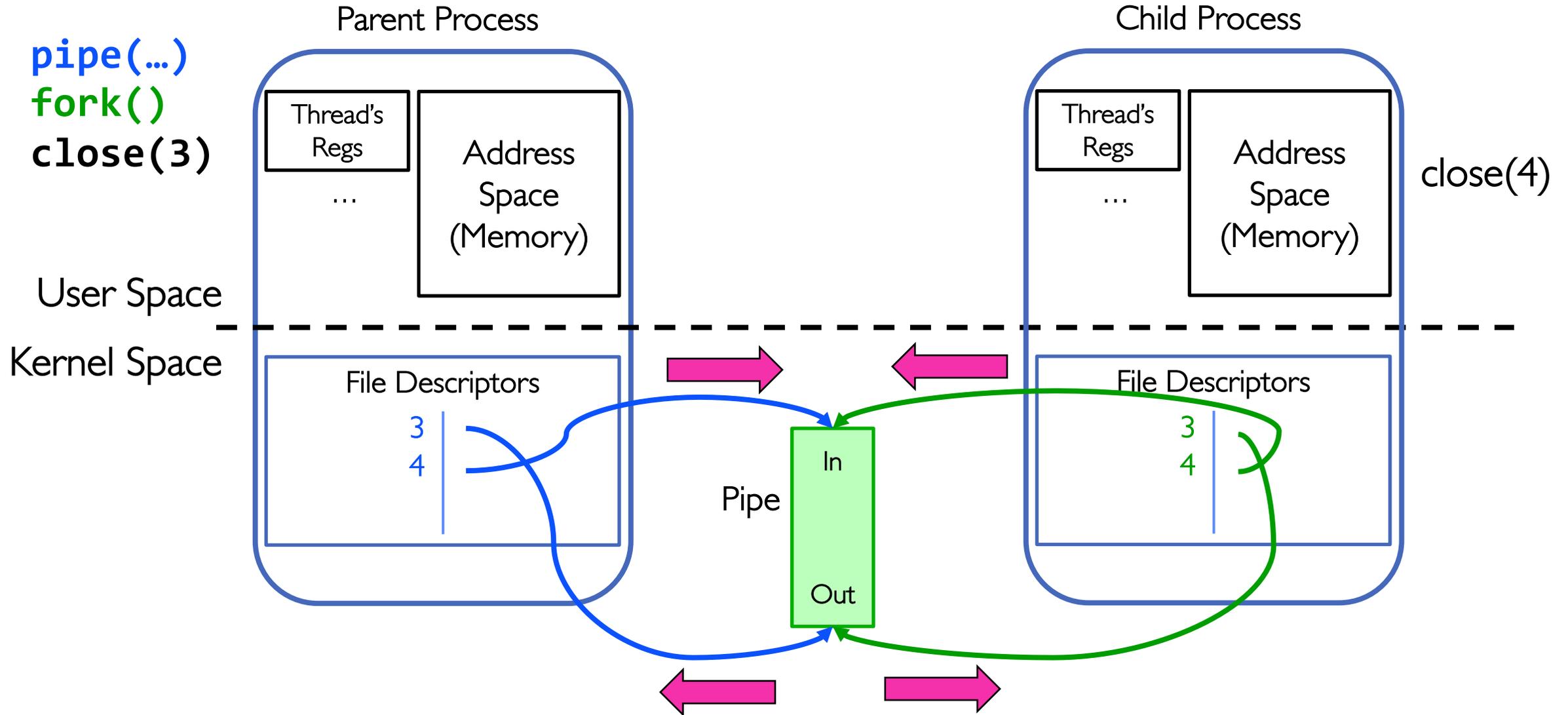# Pipes *Between* Processes

pipe(…)
fork()

Parent Process

Thread's Regs
…

Address Space (Memory)

User Space

Kernel Space

File Descriptors

3
4

Pipe

In

Out

Child Process

Thread's Regs
…

Address Space (Memory)

File Descriptors

3
4

```c
// continuing from earlier
pid_t pid = fork();
if (pid < 0) {
  fprintf (stderr, "Fork failed.\n");
  return EXIT_FAILURE;
}
if (pid != 0) {
  ssize_t writelen = write(pipe_fd[1], msg, msglen);
  printf("Parent: %s [%ld, %ld]\n", msg, msglen, writelen);
  close(pipe_fd[0]);
} else {
  ssize_t readlen  = read(pipe_fd[0], buf, BUFSIZE);
  printf("Child Rcvd: %s [%ld]\n", buf, readlen);
  close(pipe_fd[1]);
}
```

Channel from Parent $\Rightarrow$ Child

pipe(…)
fork()
close(3)

Parent Process

Thread's Regs
…

Address Space (Memory)

Child Process

Thread's Regs
…

Address Space (Memory)

close(4)

User Space

Kernel Space

File Descriptors

3
4

Pipe

In

Out

File Descriptors

3
4

Instead: Channel from Child $\Rightarrow$ Parent

pipe(…)
fork()
close(4)
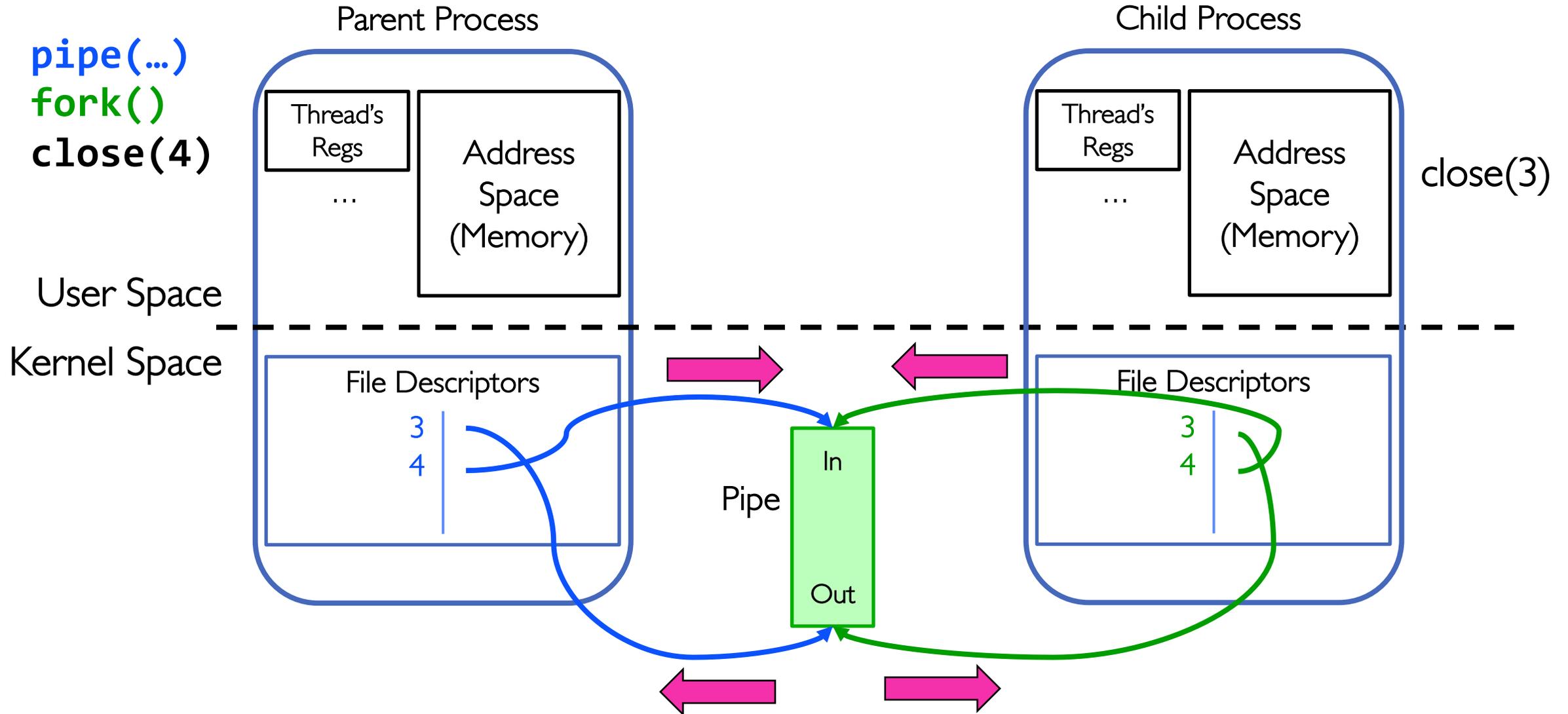
close(3)

When do we get EOF on a pipe?

- After last "write" descriptor is closed, pipe is effectively closed:
  - Reads return only "EOF"
- After last "read" descriptor is closed, writes generate SIGPIPE signals:
  - If process ignores, then the write fails with an "EPIPE" error

EOF on a Pipe

pipe(…)
fork()
close(3)
close(4)

User Space

Kernel Space

Process 1

Thread's Regs
…

Address Space (Memory)

File Descriptors

4

Process 2

Thread's Regs
…

Address Space (Memory)

close(4)

File Descriptors

3

Pipe

In

Out

EOF
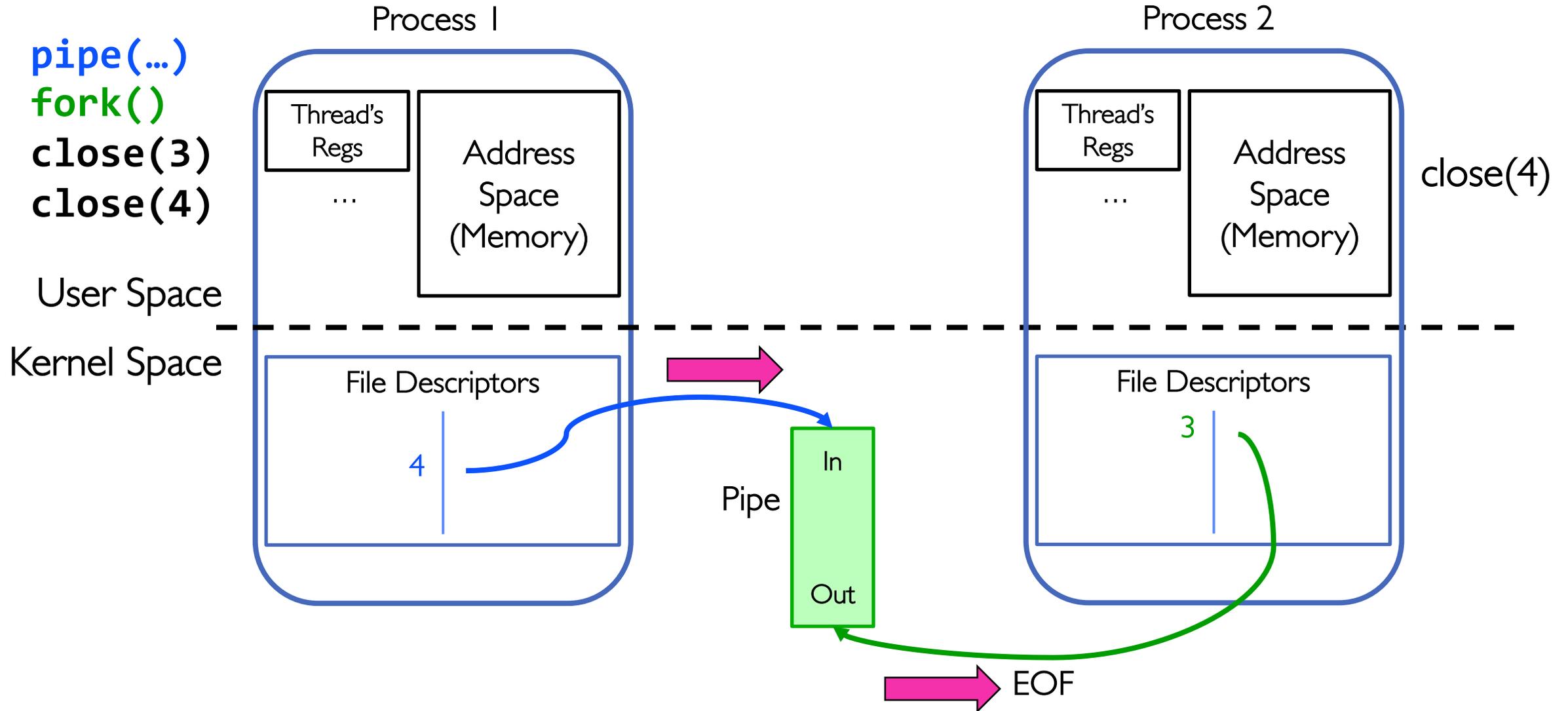
Once we have communication, we need a *protocol*

- 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

- In fact, across network may need a way to translate between different representations for numbers, strings, etc.

  - Such translation typically part of a Remote Procedure Call (RPC) facility

 - Don't worry about this now, but it is clearly part of the *protocol*

Examples of Protocols in Human Interaction

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

9.   Callee: "Yeah, blah blah blah …" **pause**
10. Caller: Bye
11.   Callee: Bye
12. Hang up

Web Server

Request

Reply

Client

Web Server
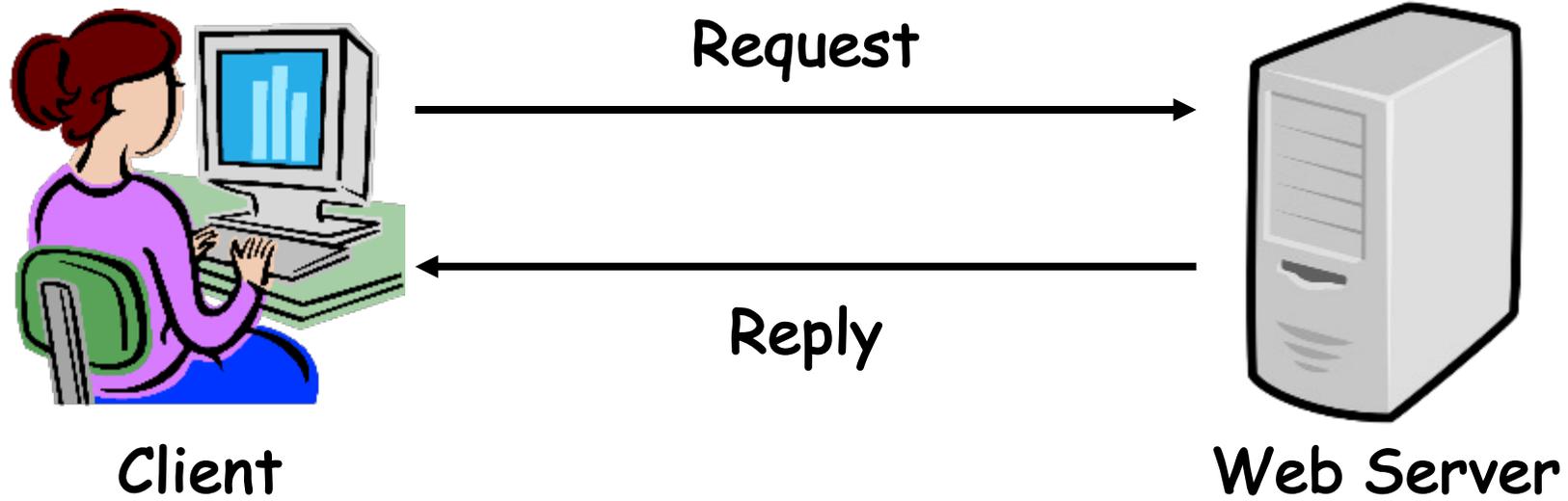
Client-Server Protocols: Cross-Network IPC
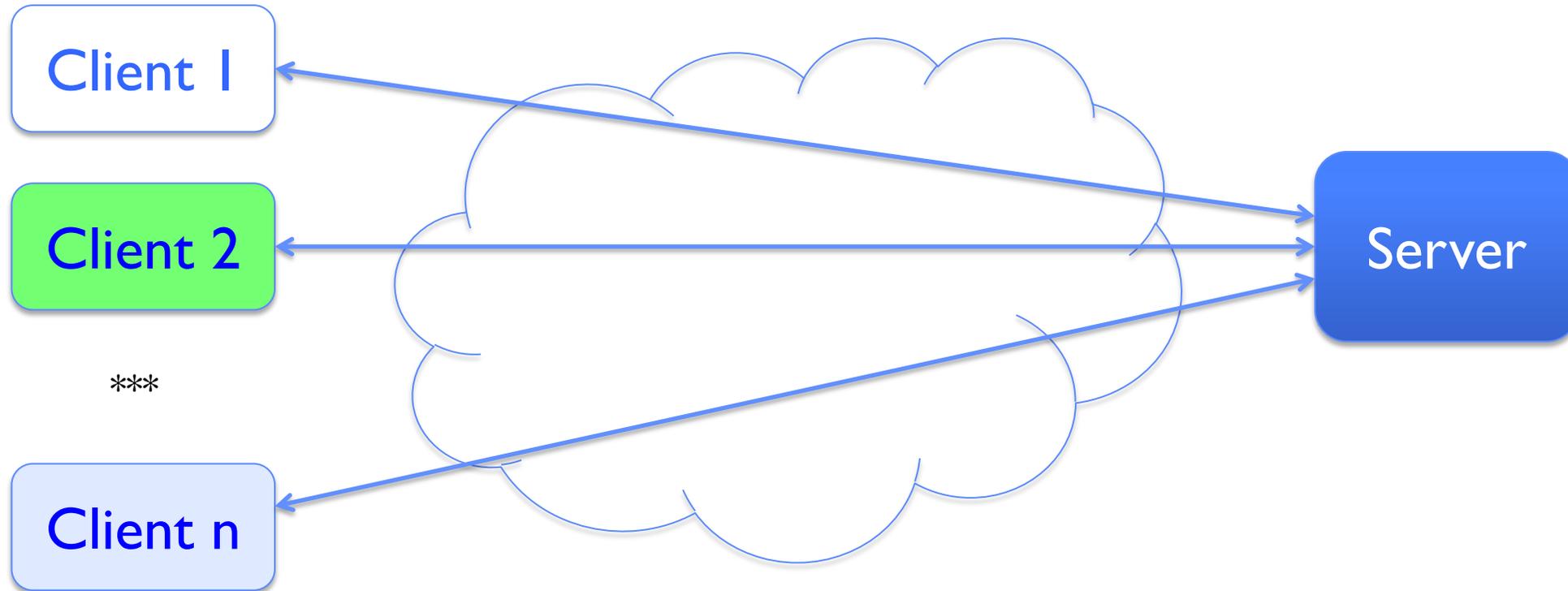
- Many clients accessing a common server
- File servers, www, FTP, databases

Client-Server Communication

- Client is "sometimes on"
  - Sends the server requests for services when interested
  - E.g., Web browser on laptop/phone
  - Doesn't communicate directly with other clients
  - Needs to know server's address

- Server is "always on"
  - Services requests from many clients
  - E.g., Web server for `www.pku.edu.cn`
  - Doesn't initiate contact with clients
  - Needs a fixed, well-known address

`GET /index.html`

"`Site under construction`"
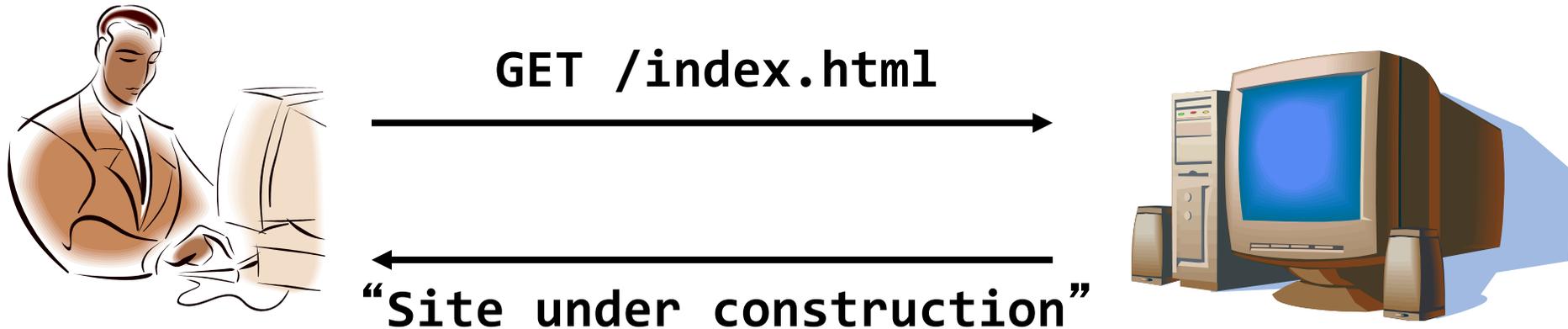
What is a Network Connection?

- Bidirectional *stream* of bytes between two processes on possibly different machines
  - For now, we are discussing "TCP Connections"

- Abstractly, a connection between two endpoints A and B consists of:
  - A queue (bounded buffer) for data sent from A to B
  - A queue (bounded buffer) for data sent from B to A

The Socket Abstraction: Endpoint for Communication

- **Key Idea:** Communication across the world looks like File I/O

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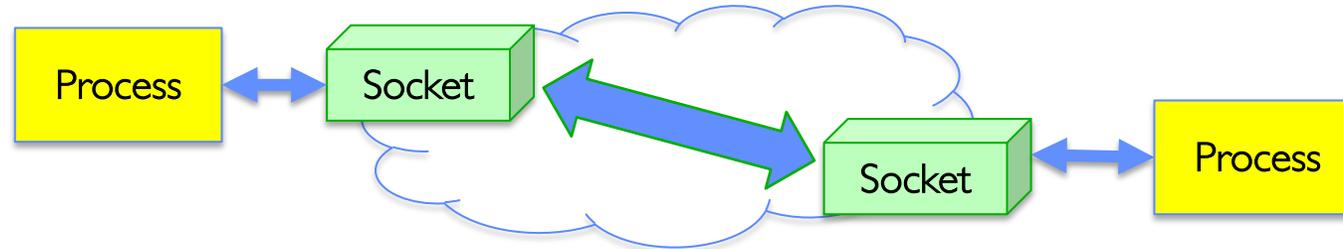


```
n = read(rfd, rbuf, rmax);
```

- Sockets: Endpoint for Communication
  - Queues to temporarily hold results
- Connection: Two Sockets Connected Over the network $\Rightarrow$ IPC over network!
  - How to **open()**?
  - What is the namespace?
  - How are they connected in time?

Sockets: More Details

- **Socket:** An abstraction for one endpoint of a network connection
  - Another mechanism for **inter-process communication**
  - Most operating systems (Linux, Mac OS X, Windows) provide this, even if they don't copy rest of UNIX I/O
  - Standardized by POSIX
- First introduced in 4.2 BSD (Berkeley Software/Standard Distribution) Unix
- Same abstraction for any kind of network
  - Local (within same machine)
  - The Internet (TCP/IP, UDP/IP)
  - Things "no one" uses anymore (OSI, Appletalk, IPX, …)

Sockets: More Details

- Looks just like a file with a **file descriptor**
  - Corresponds to a network connection (*two* queues)
  - **write** adds to output queue (queue of data destined for other side)
  - **read** removes from input queue (queue of data destined for this side)
  - Some operations do not work, e.g., **lseek**

- How can we use sockets to support real applications?
  - A bidirectional byte stream isn't useful on its own…
  - May need messaging facility to partition stream into chunks
  - May need RPC facility to translate one environment to another and provide the abstraction of a function call over the network

Simple Example: Echo Server

"hello, world"

"hello, world"

Client

Web Server

Simple Example: Echo Server

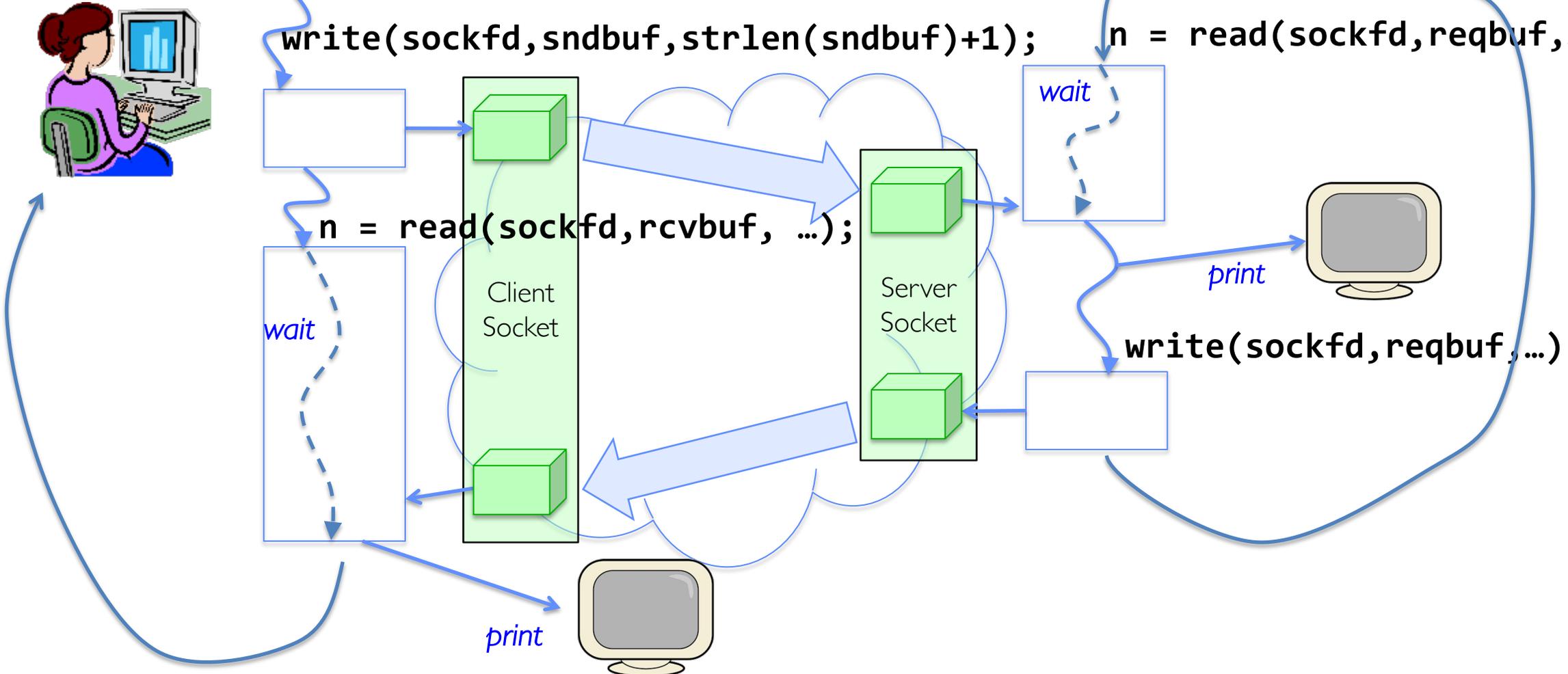**Client (issues requests)**

Server (services requests)

```
fgets(sndbuf,bufsize,stdin);
```

```
write(sockfd,sndbuf,strlen(sndbuf)+1);
```

```
n = read(sockfd,reqbuf,…);
```

wait

```
n = read(sockfd,rcvbuf, …);
```

Client Socket

Server Socket

print

wait

```
write(sockfd,reqbuf,…);
```

print

Echo client-server example

```
void client(int sockfd) {
  int n;
  char sndbuf[MAXIN]; char rcvbuf[MAXOUT];
  while (1) {
    fgets(sndbuf,MAXIN,stdin);                    /* prompt */
    write(sockfd, sndbuf, strlen(sndbuf)+1);  /* send (including null terminator) */
    memset(rcvbuf,0,MAXOUT);                       /* clear */
    n=read(sockfd, rcvbuf, MAXOUT);               /* receive */
    write(STDOUT_FILENO, rcvbuf, n);              /* echo */
  }
}
```

```
void server(int consockfd) {
  char reqbuf[MAXREQ];
  int n;
  while (1) {
    memset(reqbuf,0, MAXREQ);
    n = read(consockfd,reqbuf,MAXREQ);  /* Recv */
    if (n <= 0) return;
    write(STDOUT_FILENO, reqbuf, n);
    write(consockfd, reqbuf, n); /* echo*/
  }
}
```

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
 - Blocks if nothing has arrived yet
  - Like pipes!

Socket Creation

- File systems provide a collection of permanent objects in a structured name space:
  - Processes open/read/write/close them
  - Files exist independently of processes
  - Easy to name what file to `open()`
- Pipes: one-way communication between processes on same (physical) machine
  - Single queue
  - Created transiently by a call to `pipe()`
  - Passed from parent to children (descriptors inherited from parent process)
- Sockets: two-way communication between processes on same or different machine
  - Two queues (one in each direction)
  - Processes can be on separate machines: no common ancestor
  - How do we *name* the objects we are opening?
  - How do these completely independent programs know that the other wants to "talk" to them?

Namespaces for Communication over IP

- Hostname
  - www.pku.edu.cn
- IP address
  - 128.32.244.172 (IPv4, 32-bit Integer)
  - 2607:f140:0:81:e: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"

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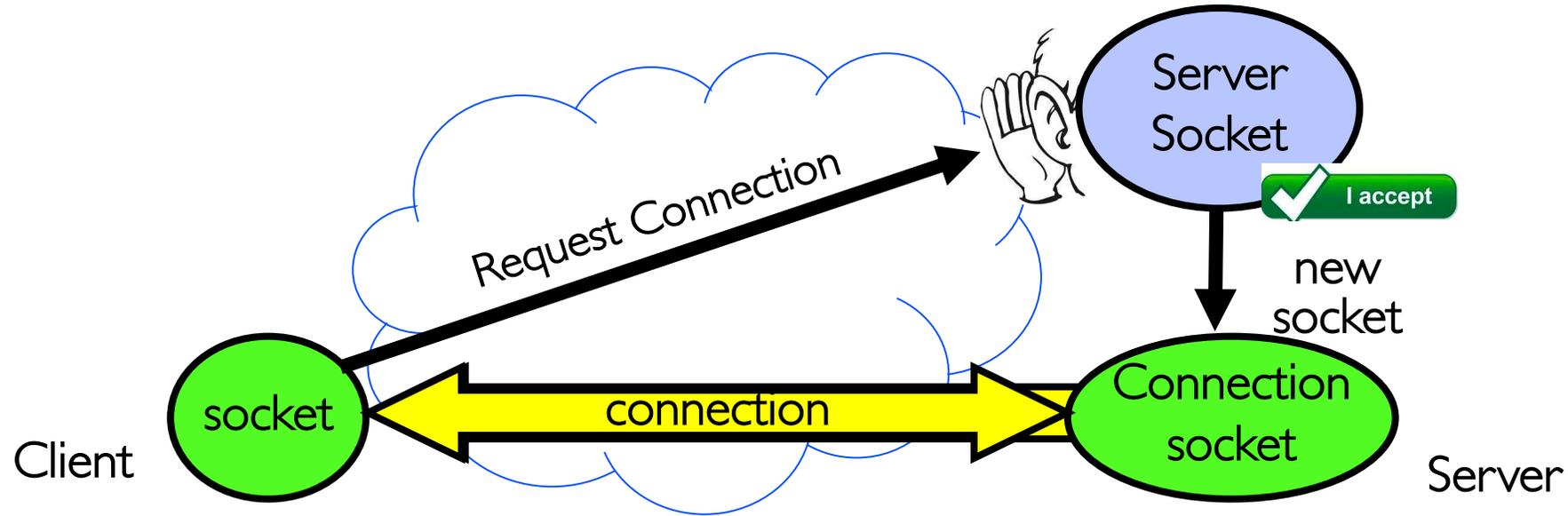
- Special kind of socket: **server socket**
  - Has file descriptor
  - Can't read or write
- Two operations:
  1. **listen()**: Start allowing clients to connect
  2. **accept()**: Create a *new socket* for a *particular* client

Connection Setup over TCP/IP

Request Connection

Server Socket

✓ I accept

new socket

socket

connection

Connection socket
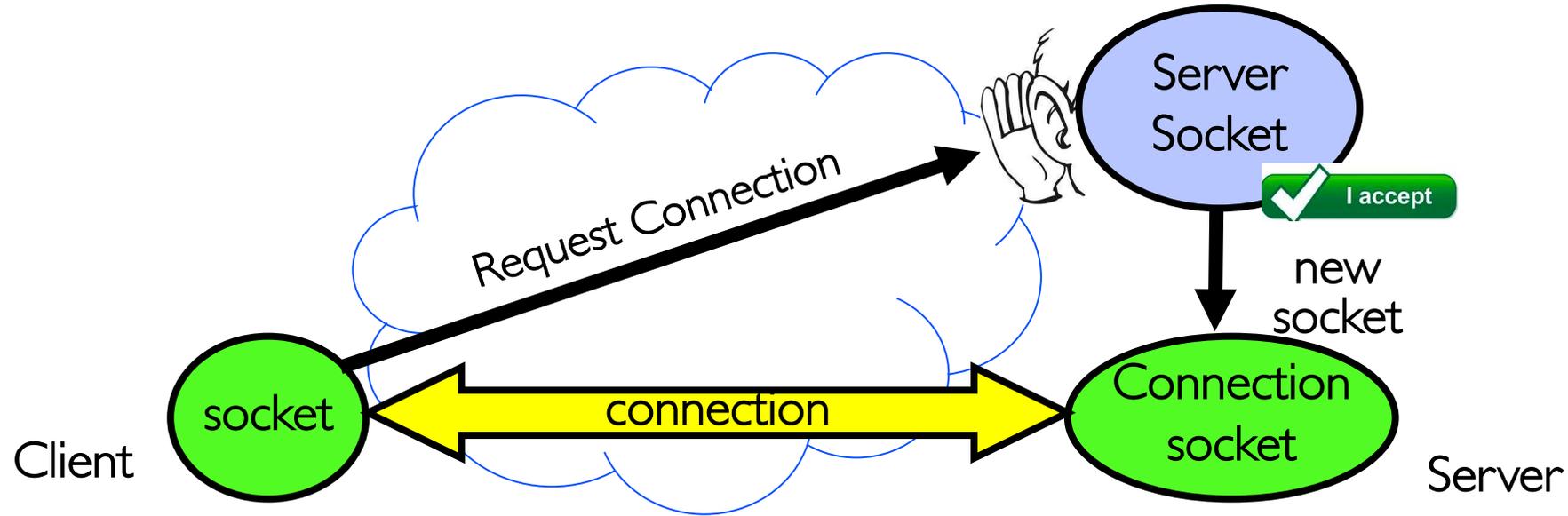
Client

Server

- 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 concept
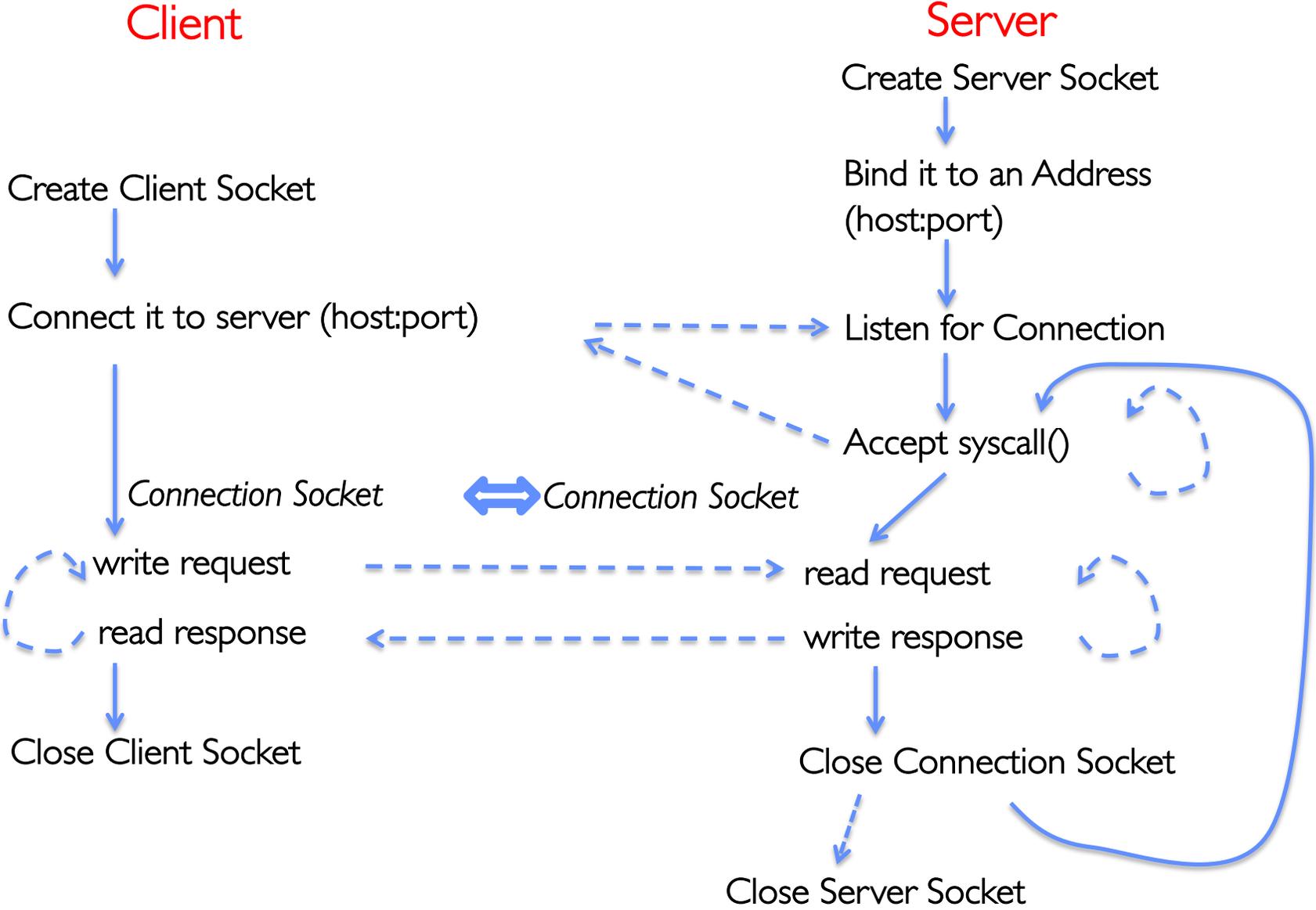
Client Protocol

```
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);
```

Address family, e.g.,
- AF_INET (IPv4)
- AF_INET6 (IPv6)

Protocol type, e.g.,
- IPPROTO_TC
- 0 (any protocol)

Socket type, e.g.,
- SOCK_STEAM
- SOCK_DGRAM

Server Protocol (v1)

```c
// 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 Could the Server Protect Itself?

- Handle each connection in a separate process

Sockets With Protection (each connection has own process)

Client

Server

Create Client Socket

Create Server Socket

Bind it to an Address (host:port)

Connect it to server (host:port)

Listen for Connection

Accept syscall()

Connection Socket

Connection Socket

Child

Parent

write request

Close Listen Socket
read request

Close Connection Socket

read response

write response

Wait for child

Close Client Socket

Close Connection Socket

Close Server Socket

Server Protocol (v2)

```c
// 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

- So far, in the server:
  - Listen will queue requests
  - Buffering present elsewhere
  - But server waits for each connection to terminate before servicing the next

- A concurrent server can handle and service a new connection before the previous client disconnects

Sockets With Protection and Concurrency

Client

Server

Create Server Socket

Create Client Socket

Bind it to an Address
(host:port)

Connect it to server (host:port) ---> Listen for Connection

Accept syscall()

Connection Socket <===> *Connection Socket*

Child

Parent

write request ----> Close Listen Socket
read request

Close Connection Socket

read response <---- write response

Close Connection Socket
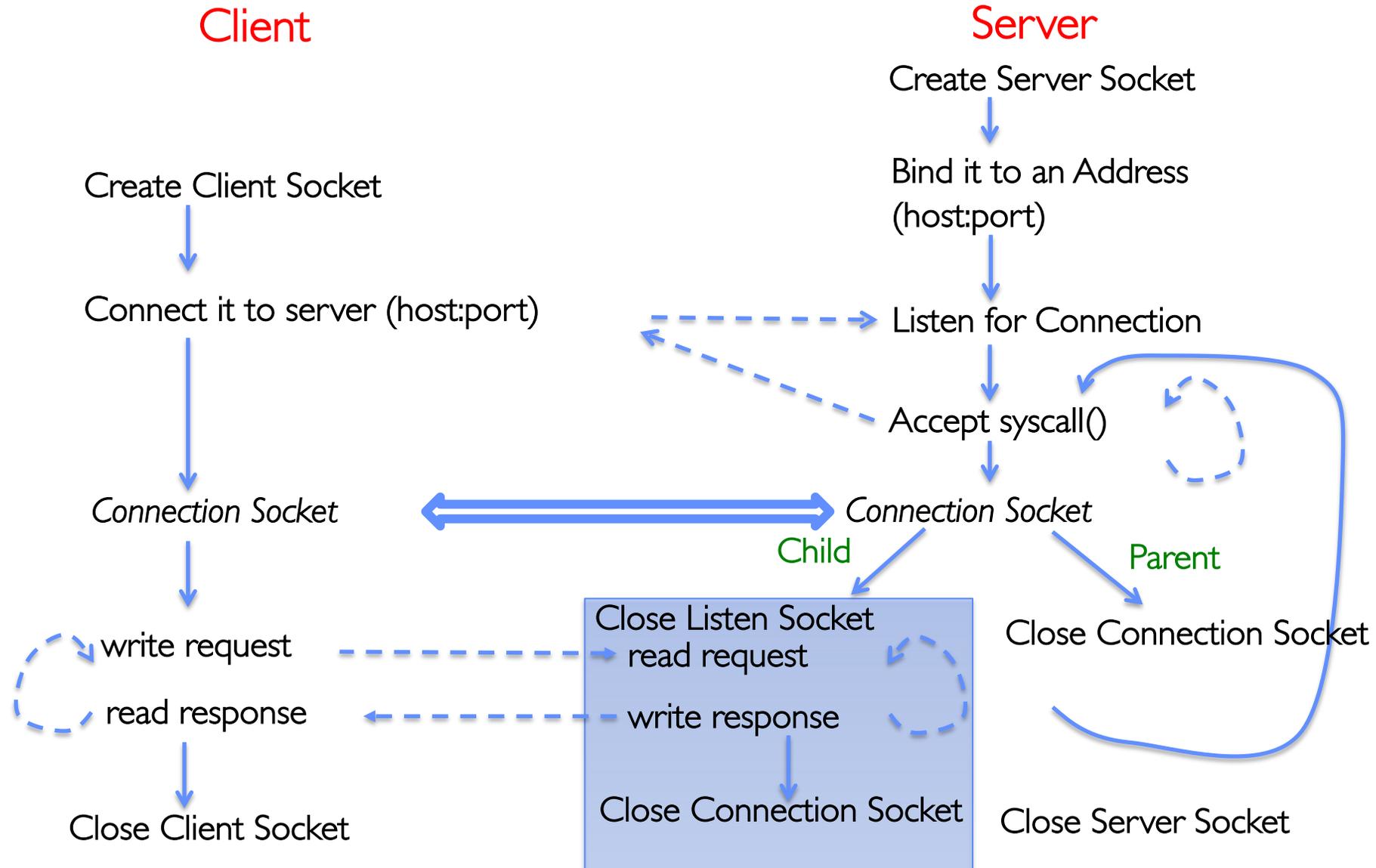
Close Client Socket

Close Server Socket

Server Protocol (v3)

```c
// 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 Concurrency, without Protection

Client

Server

Create Server Socket

Create Client Socket

Bind it to an Address (host:port)

Connect it to server (host:port)

Listen for Connection

Accept syscall()

Connection Socket

Connection Socket

pthread_create

Spawned Thread

write request

read request

Main Thread

read response

write response

Close Client Socket

Close Connection Socket

Close Server Socket

Thread Pools

- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded "pool" of worker threads, representing the maximum level of multiprogramming

Master Thread

queue

Thread Pool
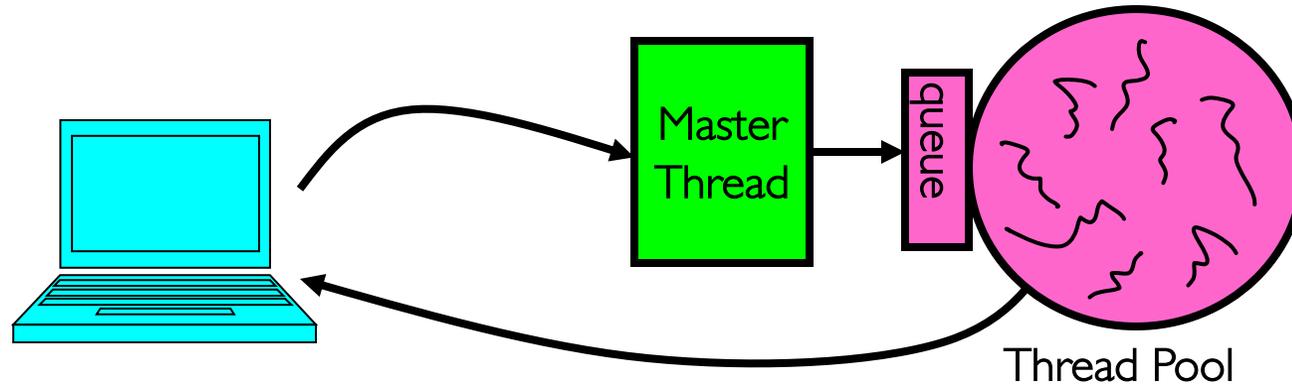
```
master() {
    allocThreads(worker,queue);
    while(TRUE) {
        con=AcceptCon();
        Enqueue(queue,con);
        wakeUp(queue);
    }
}
```

```
worker(queue) {
    while(TRUE) {
        con=Dequeue(queue);
        if (con==null)
            sleepOn(queue);
        else
            ServiceWebPage(con);
    }
}
```

Summary

- Interprocess Communication (IPC)
  - Communication facility between protected environments (i.e. processes)

- Pipes are an abstraction of a single queue
  - One end write-only, another end read-only
  - Used for communication between multiple processes on one machine
  - File descriptors obtained via inheritance

- Sockets are an abstraction of two queues, one in each direction
  - Can read or write to either end
  - Used for communication between multiple processes on different machines
  - File descriptors obtained via socket/bind/connect/listen/accept
  - Inheritance of file descriptors on fork() facilitates handling each connection in a separate process

- Both support read/write system calls, just like File I/O