Operating Systems (Honor Track)

Synchronization 1: Concurrency

Xin Jin

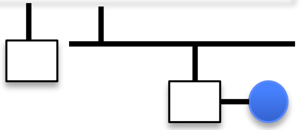
Spring 2022

Acknowledgments: Ion Stoica, Berkeley CS 162

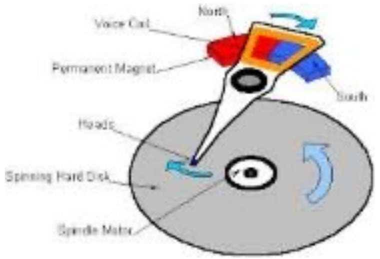
Recap: I/O and Storage Layers

Application / Service

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



Disks, Flash, Controllers, DMA



Recap: Why Buffer in Userspace? Overhead!

- Syscalls are more expensive than function calls
- `read`/`write` a file byte by byte? Max throughput of **~10MB/second**
- With `fgetc`? Keeps up with your SSD

Recap: Why Buffer in Userspace? Functionality!

- System call operations less capable
  - Simplifies kernel

- Example: No “read until new line” operation in kernel
  - Why? Kernel *agnostic* about formatting!
  - Solution: Make a big read syscall, find first new line in userspace
    - » i.e. use one of the following high-level options:

```
char *fgets(char *s, int size, FILE *stream);
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

Recap: State Maintained by the Kernel

- Recall: On a successful call to open():
  - A file descriptor (int) is returned to the user
  - An open file description is created in the kernel

- For each process, kernel maintains mapping from file descriptor to open file description
  - On future system calls (e.g., read()), kernel looks up open file description using file descriptor and uses it to service the system call:

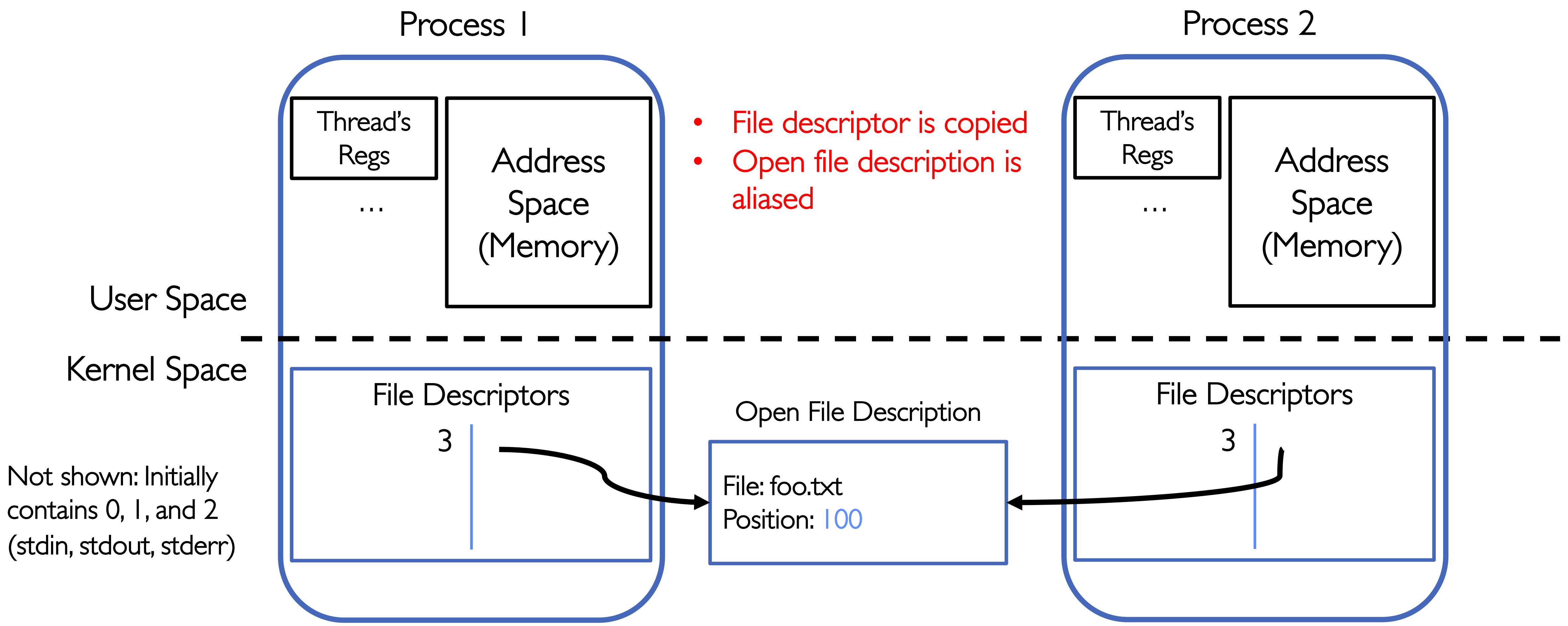
```
char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);
```

The kernel remembers that the int it receives (stored in fd) corresponds to foo.txt

The kernel picks up where it left off in the file

Recap: Instead of Closing, let's `fork()`!

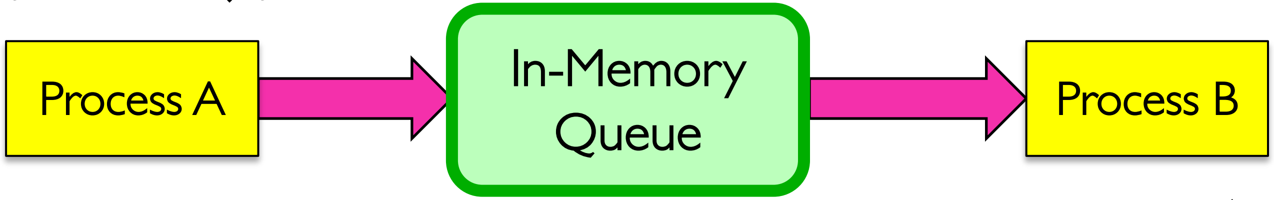
- File descriptor is copied
- Open file description is aliased



Recap: 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);
```



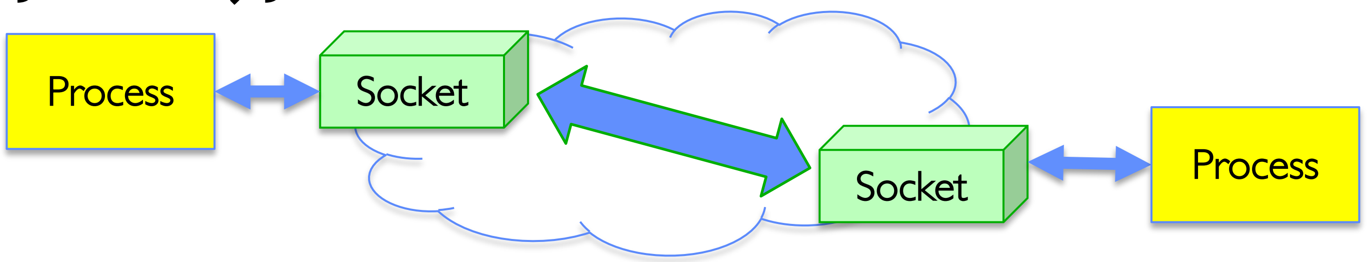

```
n = read(rfd, rbuf, rmax);
```

- 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?

Recap: The Socket Abstraction: Endpoint for Communication

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

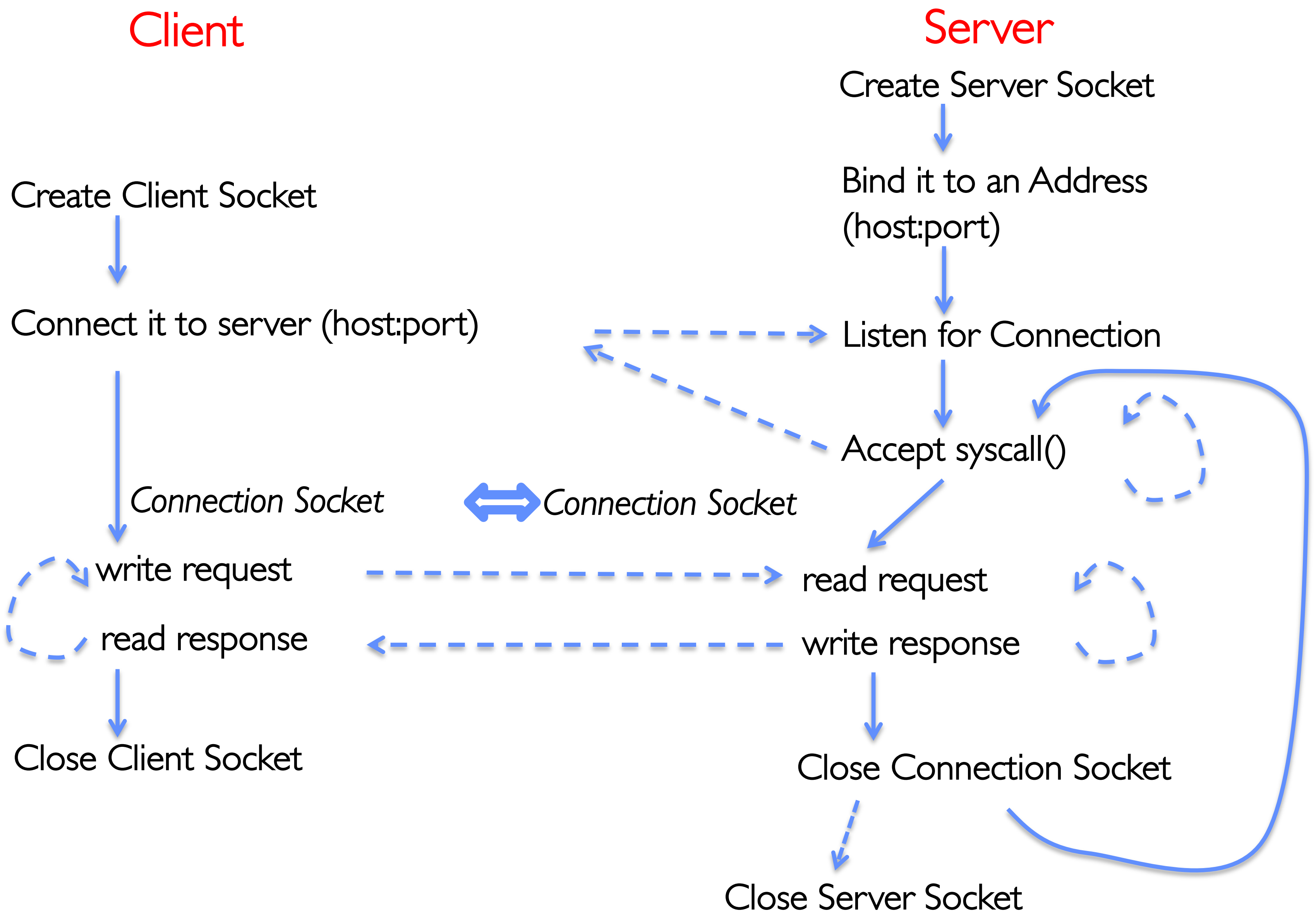
```
write(wfd, wbuf, wlen);
```




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

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

Recap: 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)

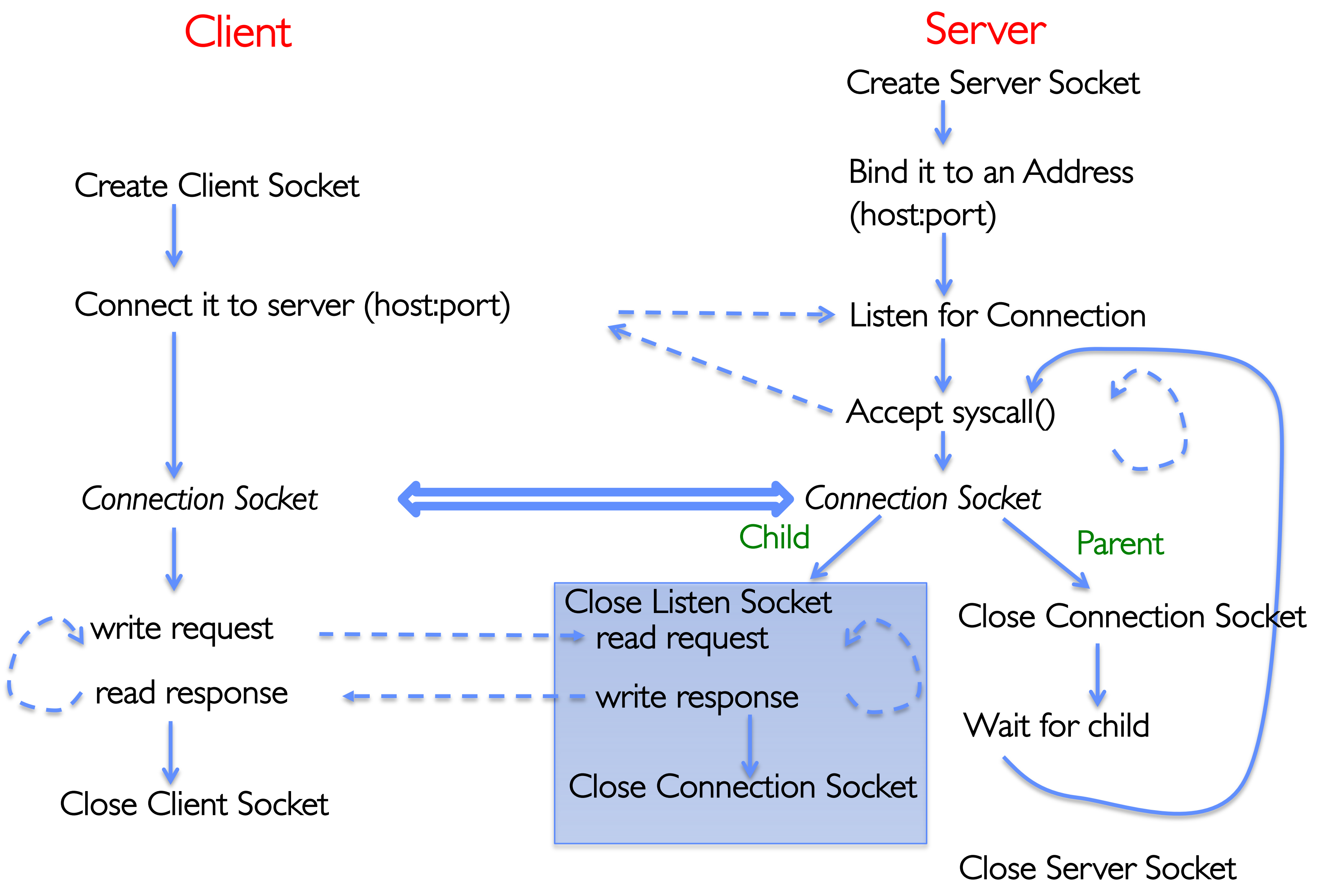
```
// Create socket to listen for client connections
char *port_name;
struct addrinfo *server = setup_address(port_name);
int server_socket = socket(server->ai_family,
      server->ai_socktype, server->ai_protocol);
// Bind socket to specific port
bind(server_socket, server->ai_addr, server->ai_addrlen);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



Server Protocol (v2)

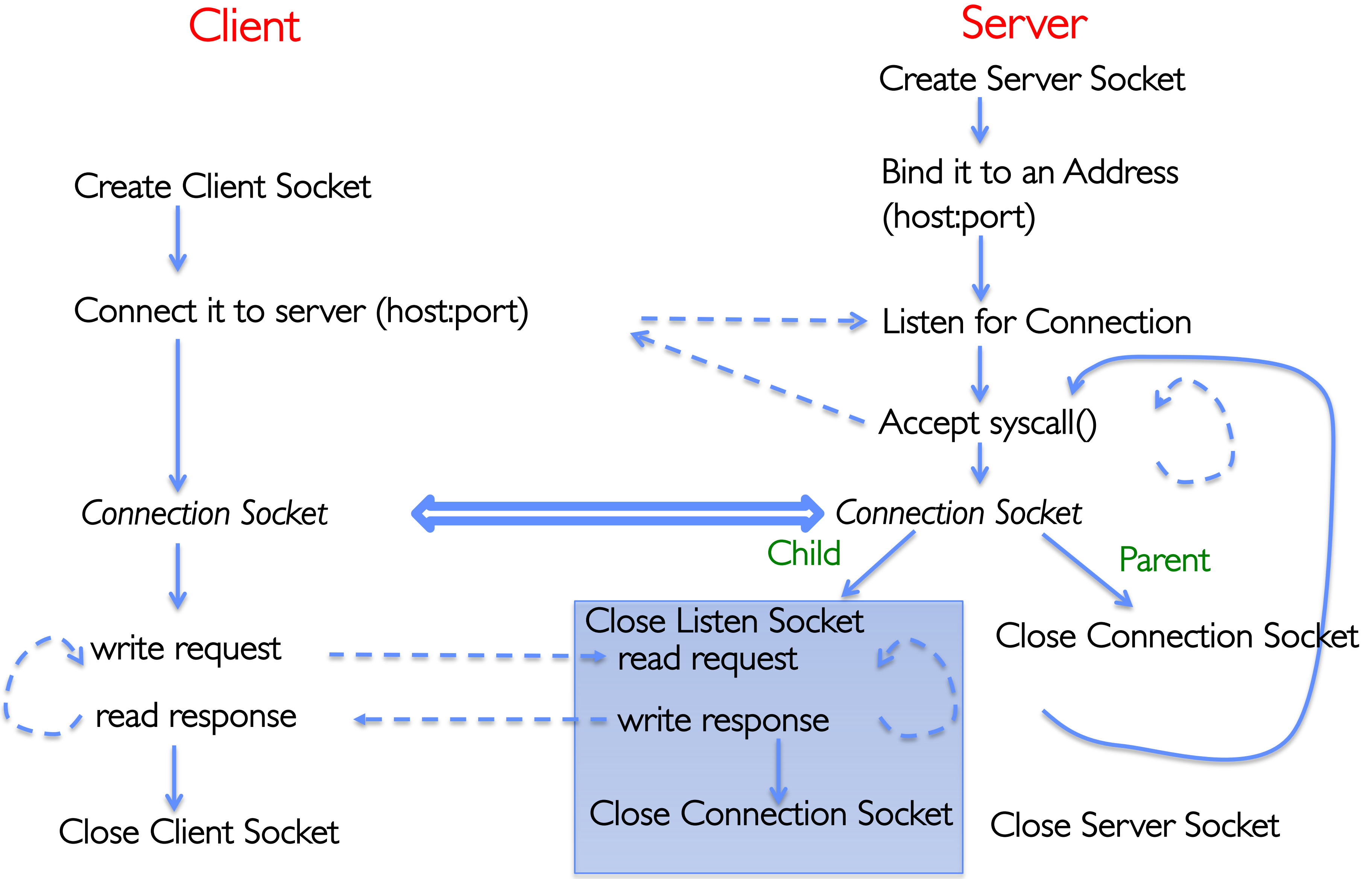
```
// Socket setup code elided…
while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  pid_t pid = fork();
  if (pid == 0) {
    close(server_socket);
    serve_client(conn_socket);
    close(conn_socket);
    exit(0);
  } else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

Concurrent Server

- 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



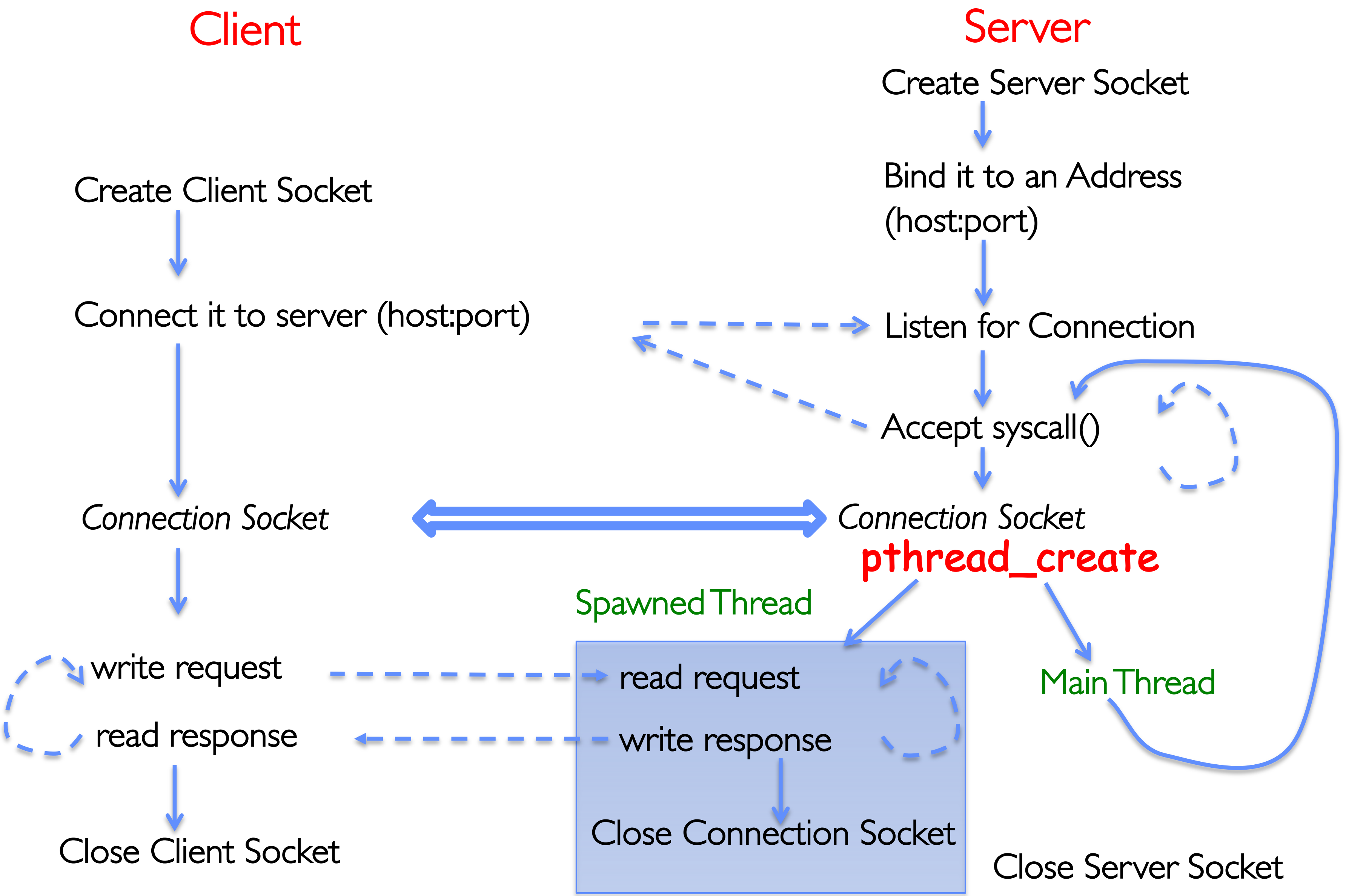
Server Protocol (v3)

```
// 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

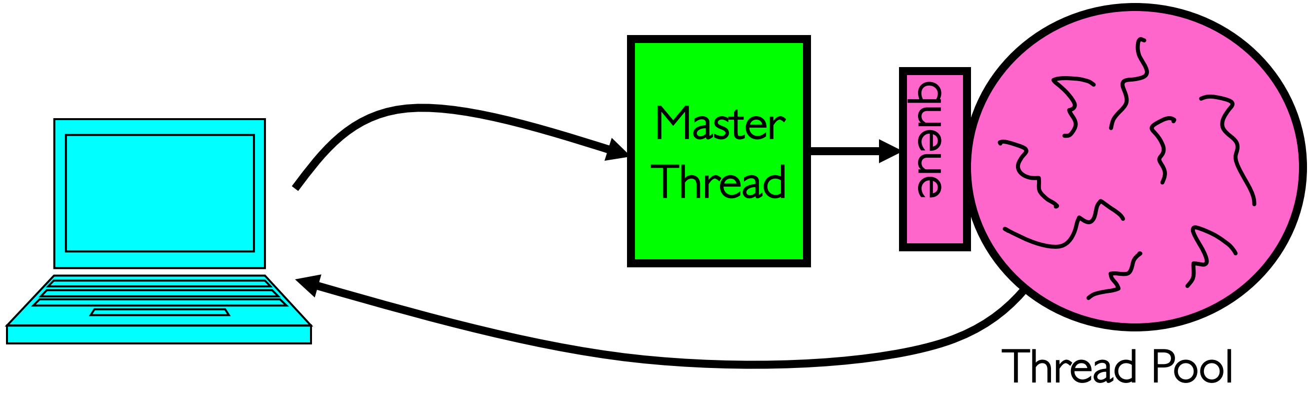
- Spawn a new thread to handle each connection
- Main thread initiates new client connections without waiting for previously spawned threads
- Why give up the protection of separate processes?
  - More efficient to create new threads
  - More efficient to switch between threads

Sockets with Concurrency, without Protection



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() {
   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);
   }
}
```

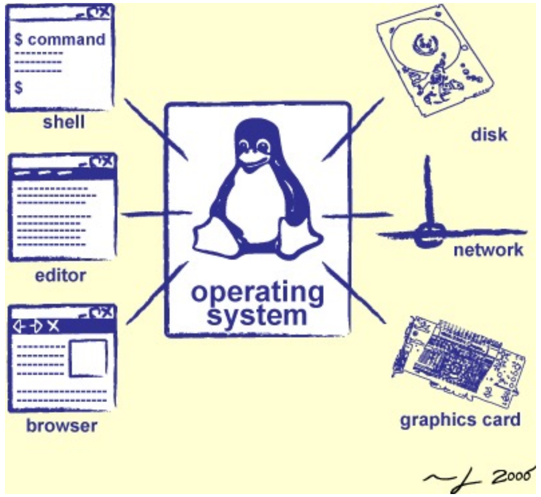
Group Discussion

- Topic: Pipes vs. Sockets
  - What is a pipe? What is a socket?
  - What are similar between pipes and sockets?
  - What are different between pipes and sockets?

- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

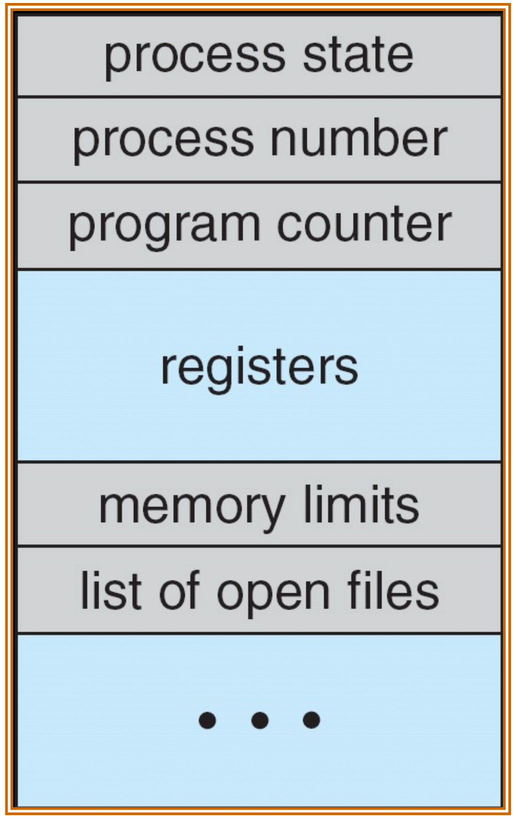
Agenda: Synchronization

- How does an OS provide concurrency through threads?
  - Brief discussion of process/thread states and scheduling
  - High-level discussion of how stacks contribute to concurrency
- Introduce needs for synchronization
- Discussion of Locks and Semaphores



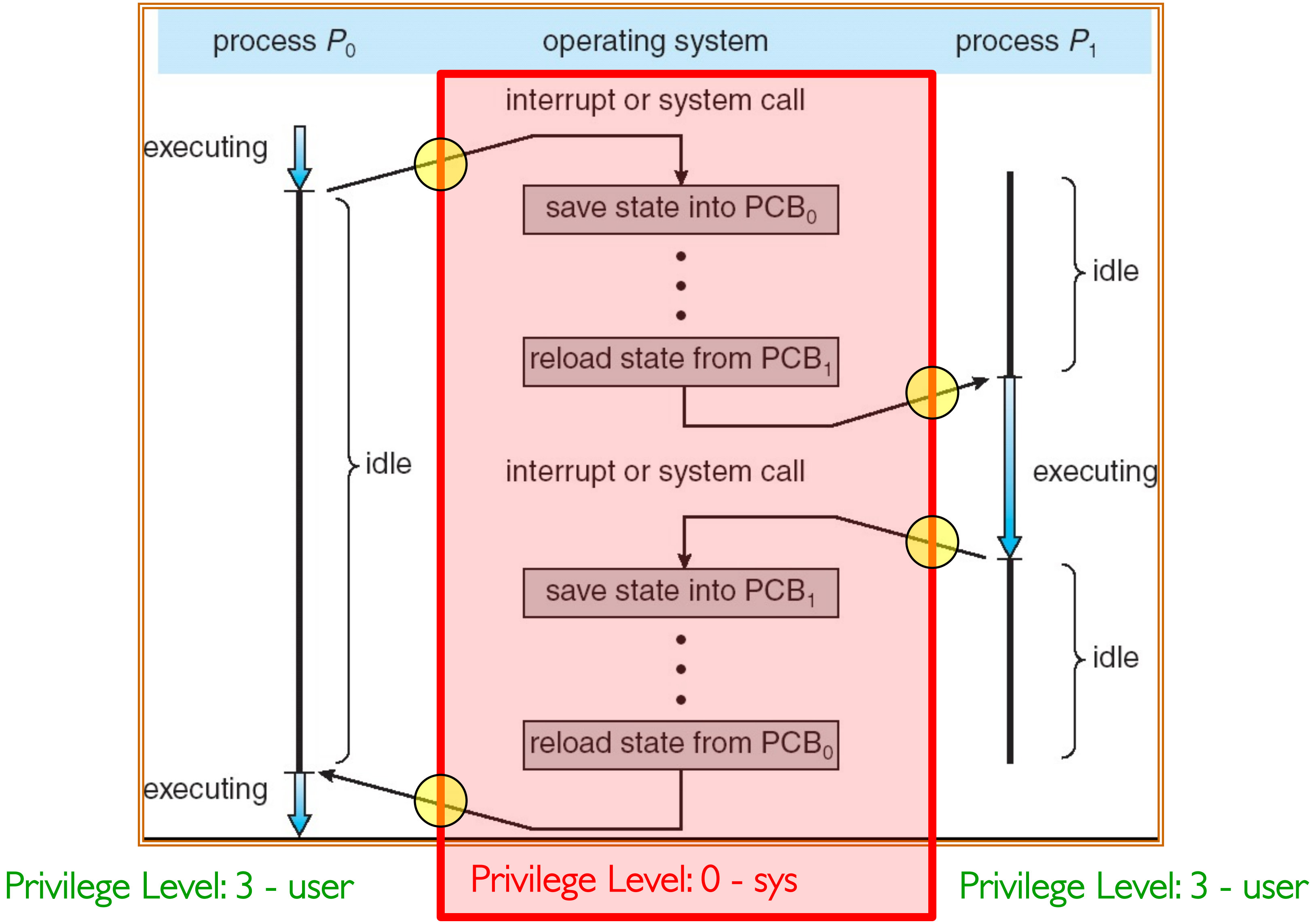
Multiplexing Processes: The Process Control Block

- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, …)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation, …
- Kernel *Scheduler* maintains a data structure containing the PCBs
  - Give out CPU to different processes
  - This is a Policy Decision
- Give out non-CPU resources
  - Memory/IO
  - Another policy decision

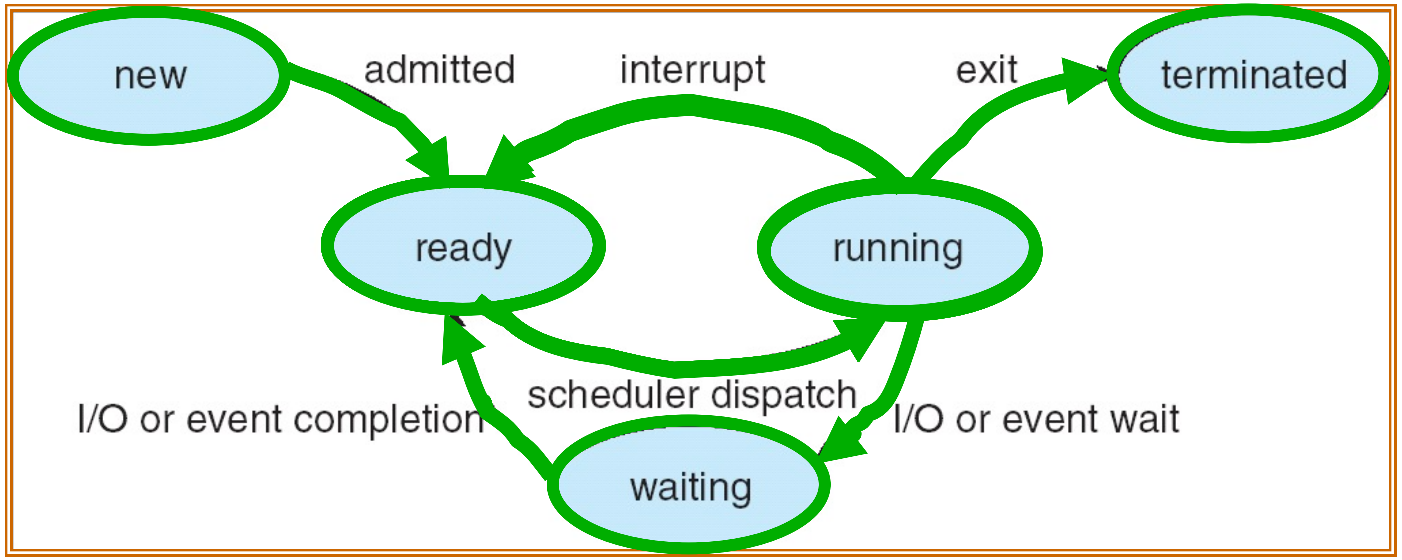


Process Control Block

Context Switch

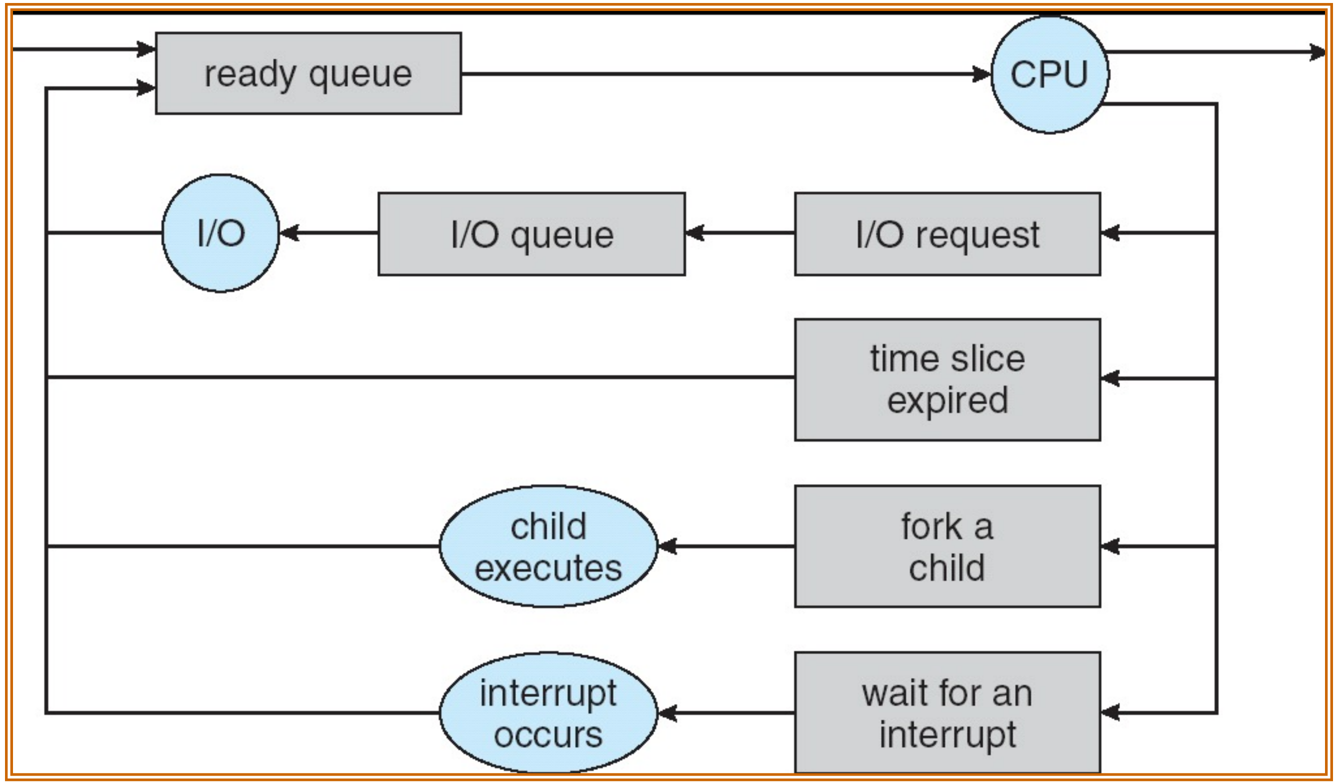


Lifecycle of a Process or Thread



- As a process executes, it changes state:
  - new: The process/thread is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution

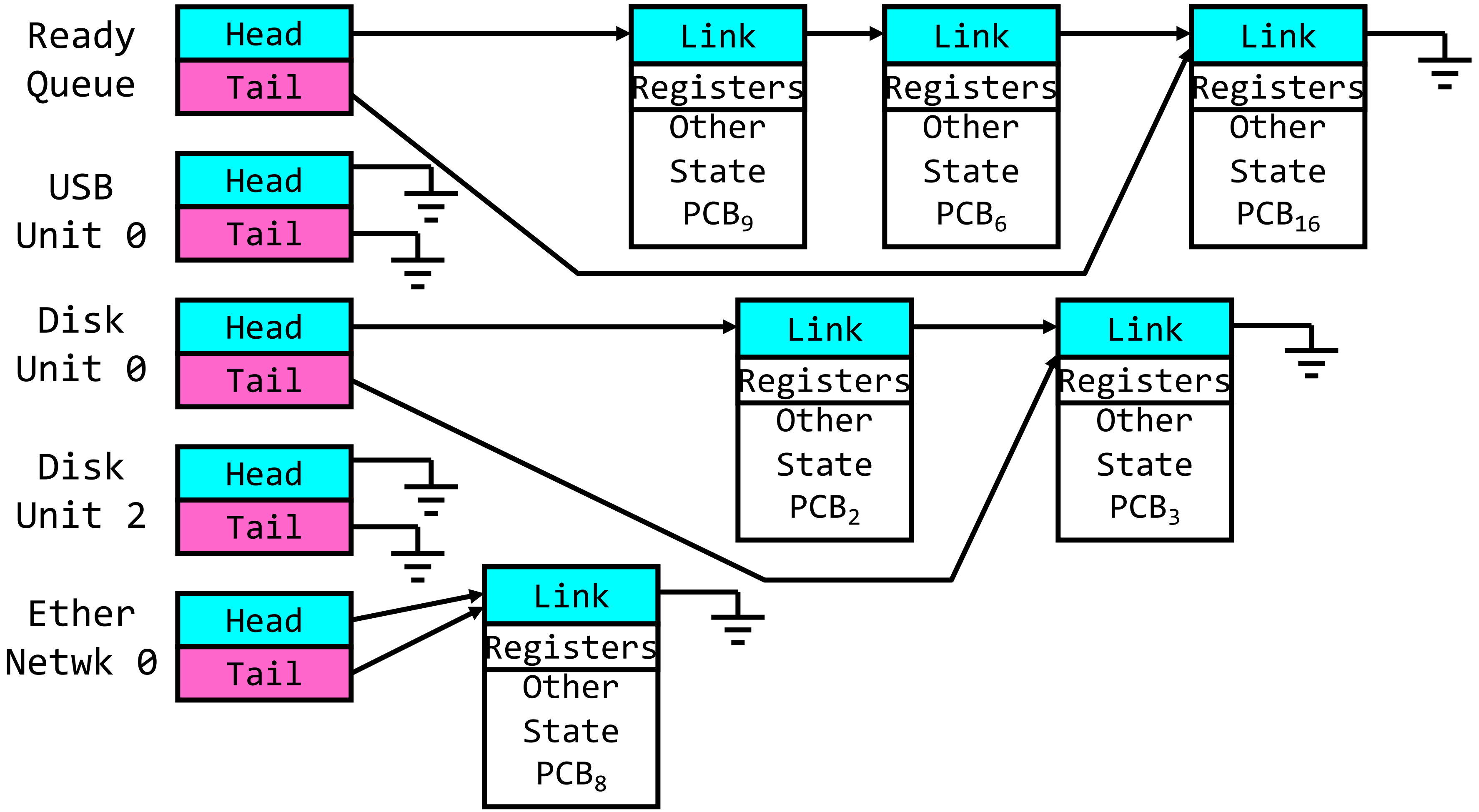
Scheduling: All About Queues



- PCBs move from queue to queue
- **Scheduling:** which order to remove from queue
  - Much more on this soon

Ready Queue And Various I/O Device Queues

- Process not running ⇒ PCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy

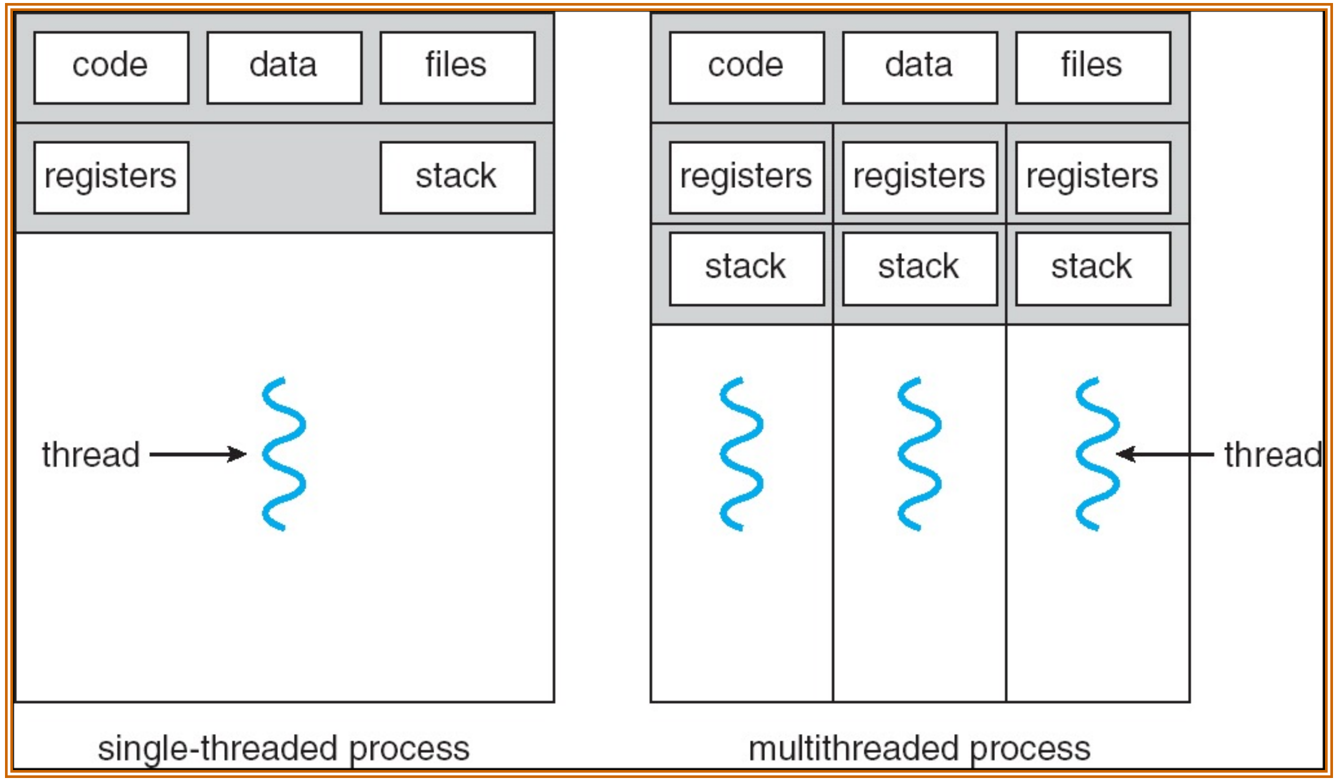


Scheduler

```
if ( readyProcesses(PCBs) ) {
    nextPCB = selectProcess(PCBs);
    run( nextPCB );
} else {
    run_idle_process();
}
```

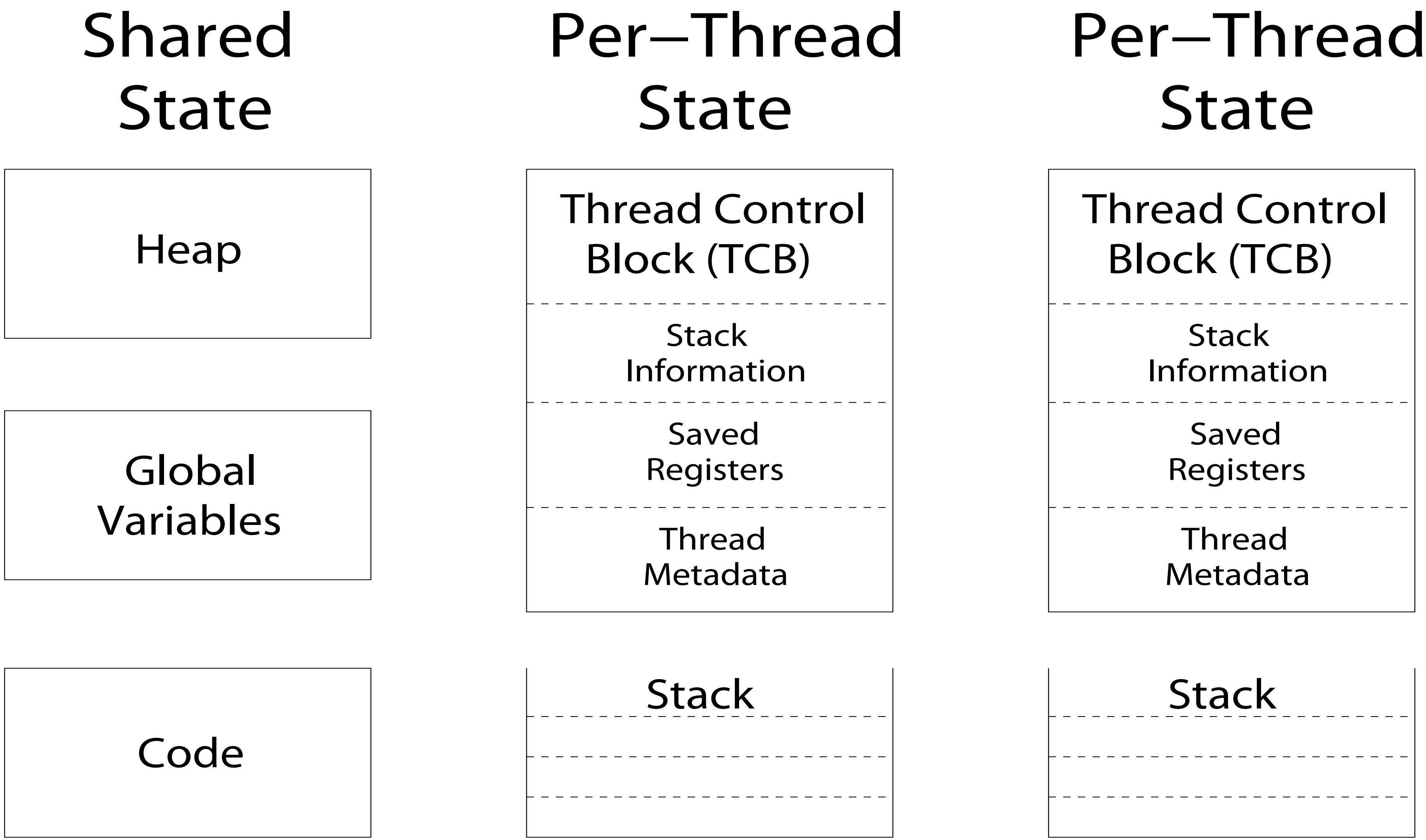
- Scheduling: Mechanism for deciding which processes/threads receive the CPU
- Lots of different scheduling policies provide …
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or ..

Recall: Single and Multithreaded Processes



- Threads encapsulate concurrency
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

Shared vs. Per-Thread State



The Core of Concurrency: the Dispatch Loop

- Conceptually, the scheduling loop of the operating system looks as follows:

```
Loop {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
}
```

- This is an *infinite* loop
  - One could argue that this is all that the OS does

Running a thread

Consider first portion: `RunThread()`

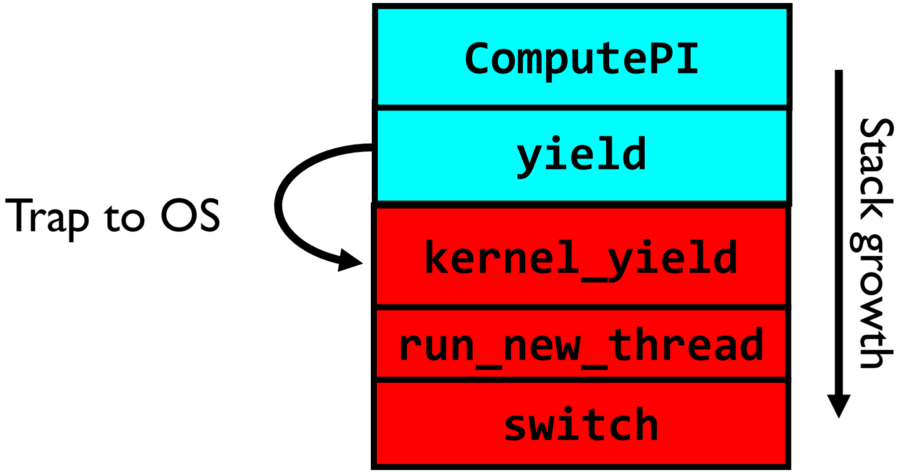
- How do I run a thread?
  - Load its state (registers, PC, stack pointer) into CPU
  - Load environment (virtual memory space, etc)
  - Jump to the PC
- How does the dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets *preempted*

Internal Events

- Blocking on I/O
  - The act of requesting I/O implicitly yields the CPU
- Waiting on a "signal" from other thread
  - Thread asks to wait and thus yields the CPU
- Thread executes a `yield()`
  - Thread volunteers to give up CPU

```
computePI() {
    while(TRUE) {
        ComputeNextDigit();
        yield();
    }
}
```

Stack for Yielding Thread



- How do we run a new thread?

```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* Do any cleanup */
}
```

- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack pointer
  - Maintain isolation for each thread

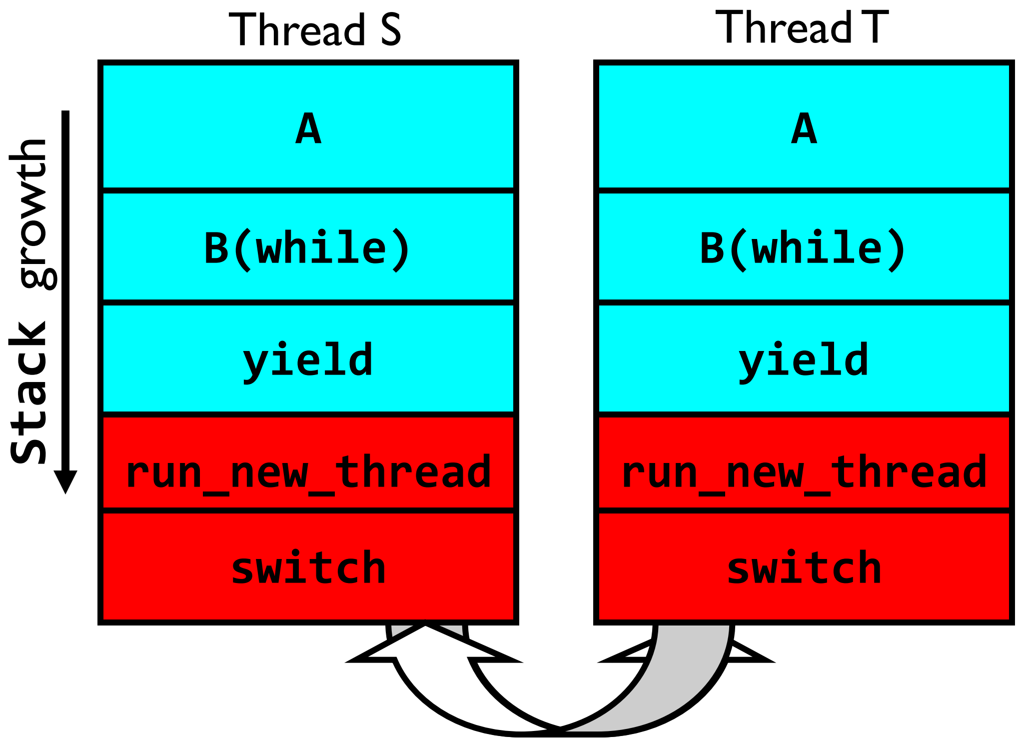
What Do the Stacks Look Like?

- Consider the following code blocks:

```
proc A() {
    B();

}
proc B() {
    while(TRUE) {
        yield();
    }
}
```

- Suppose we have 2 threads:
  - Threads S and T



Thread S's switch returns to Thread T's (and vice versa)

Conclusion

- Concurrency accomplished by multiplexing CPU time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (yield(), I/O) or involuntary (interrupts)
- TCB + Stacks hold complete state of thread for restarting