



Concurrent Programming

15-213/18-213/14-513/15-513/18-613: Introduction to Computer Systems
24th Lecture, November 19, 2020

Announcements

- **Final Exam will be Thu, Dec. 17**
 - Three hours. 3 time slots: 8:30 am ET, 1 pm ET, 5:30 pm ET
 - Will post to Piazza a form for you to indicate which slots work for you
 - Then we will assign you a slot
- **Lab 6 (tshlab)**
 - Due Mon, Nov. 23, 11:59pm ET (**Note: Monday!**)
- **Written Assignment 10 peer grading**
 - Due Wed, Nov. 25, 11:59pm ET
- **Written Assignment 11**
 - Due Wed, Nov. 25, 11:59 pm ET
- **Recitation on Networking and Proxy**
 - Mon, Nov. 23
- **No lecture on Tue, Nov. 24**
- **No lecture on Thu, Nov. 26**



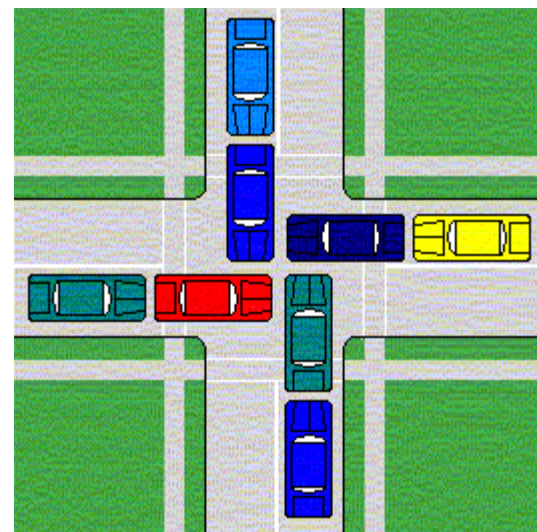
Today

- **Concurrent Programming Basics**
- **Process-based Servers** CSAPP 12.1
- **Event-based Servers** CSAPP 12.2
- **Thread-based Servers** CSAPP 12.3

Concurrent Programming is Hard!

- **The human mind tends to be sequential**
- **The notion of time is often misleading**
- **Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible**

Deadlock



Deadlock

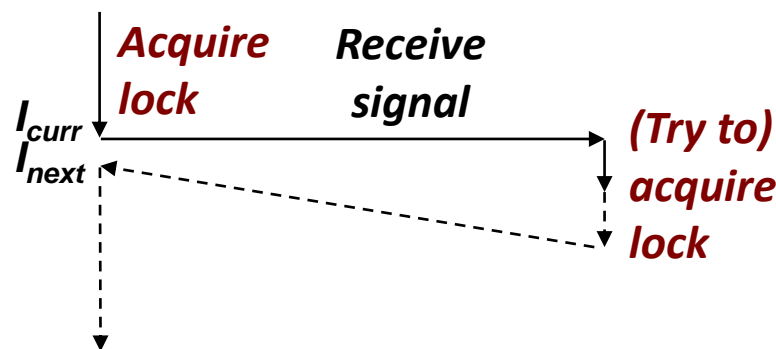


- Example from signal handlers.
- Why don't we use printf in handlers?

```
void catch_child(int signo) {
    printf("Child exited!\n"); // this call may reenter printf/puts! BAD! DEADLOCK!
    while (waitpid(-1, NULL, WNOHANG) > 0) continue; // reap all children
}
```

- **Printf code:**

- Acquire lock
- Do something
- Release lock



- **What if signal handler interrupts call to printf?**

Testing Printf Deadlock

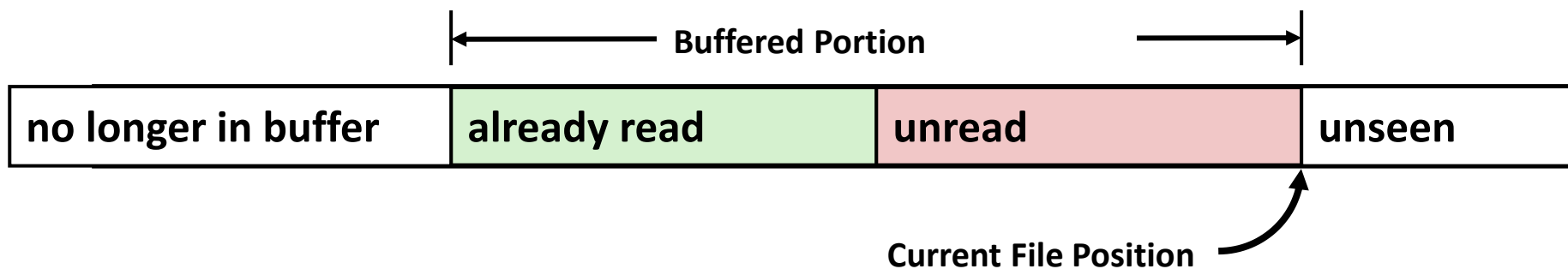
```
void catch_child(int signo) {
    printf("Child exited!\n"); // this call may reenter printf/puts! BAD! DEADLOCK!
    while (waitpid(-1, NULL, WNOHANG) > 0) continue; // reap all children
}

int main(int argc, char** argv) {
    ...
    for (i = 0; i < 1000000; i++) {
        if (fork() == 0) {
            // in child, exit immediately
            exit(0);
        }
        // in parent
        sprintf(buf, "Child #%d started\n", i);
        printf("%s", buf);
    }
    return 0;
}
```

```
Child #0 started
Child #1 started
Child #2 started
Child #3 started
Child exited!
Child #4 started
Child exited!
Child #5 started
.
.
.
Child #5888 started
Child #5889 started
```

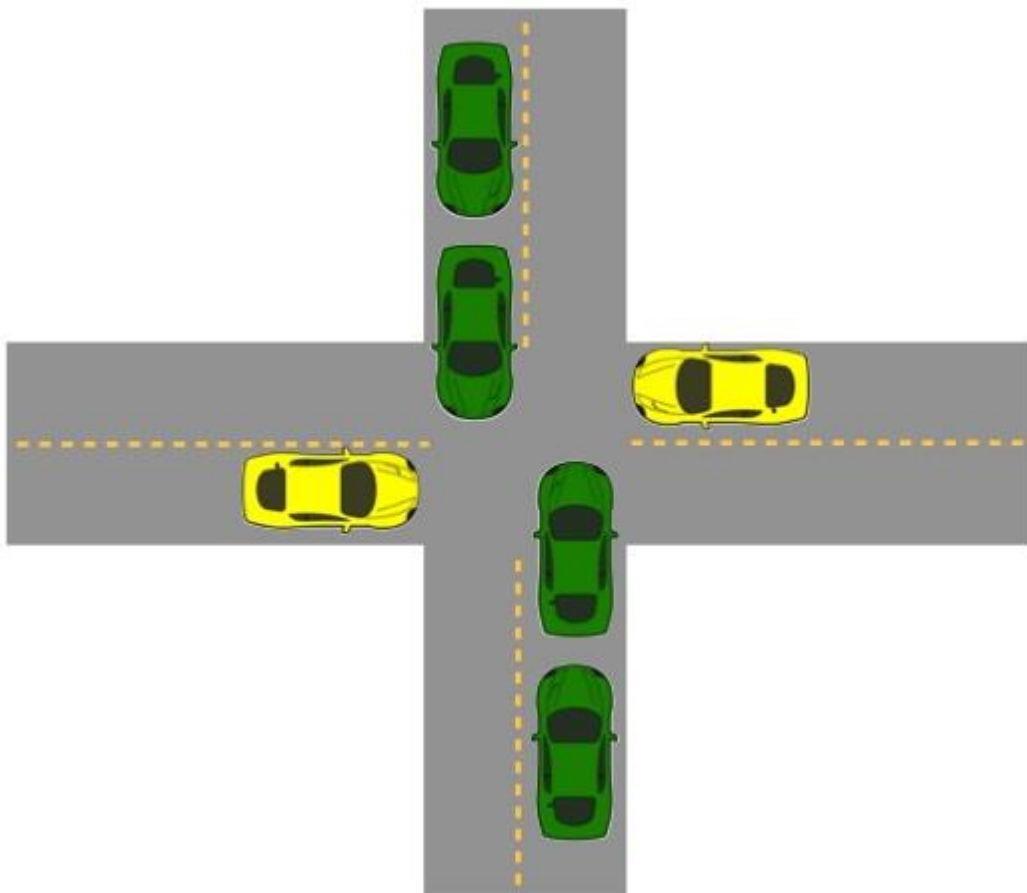

Why Does Printf require Locks?

- Printf (and fprintf, sprintf) implement *buffered* I/O



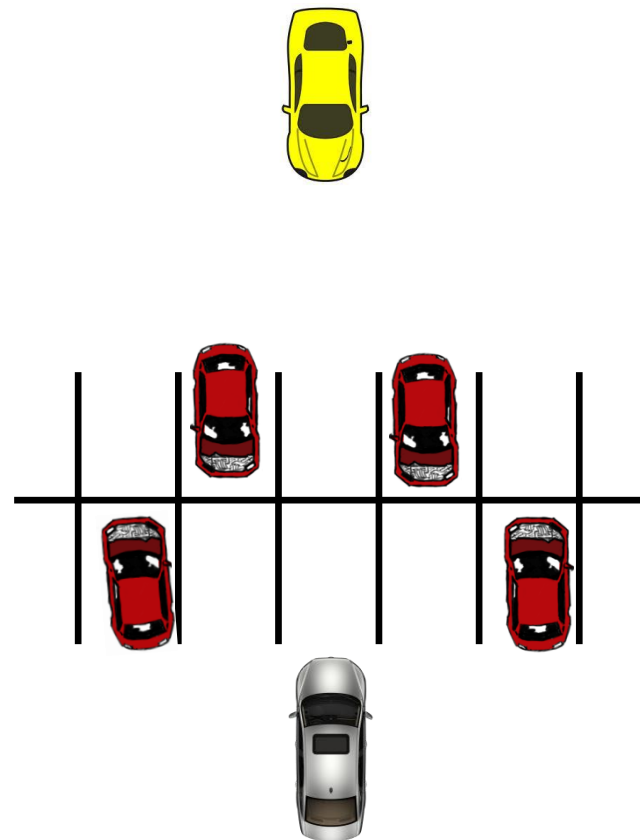
- Require locks to access to shared buffers

Starvation



- Yellow must yield to green
- Continuous stream of green cars
- Overall system makes progress, but some individuals wait indefinitely

Data Race



Concurrent Programming is Hard!

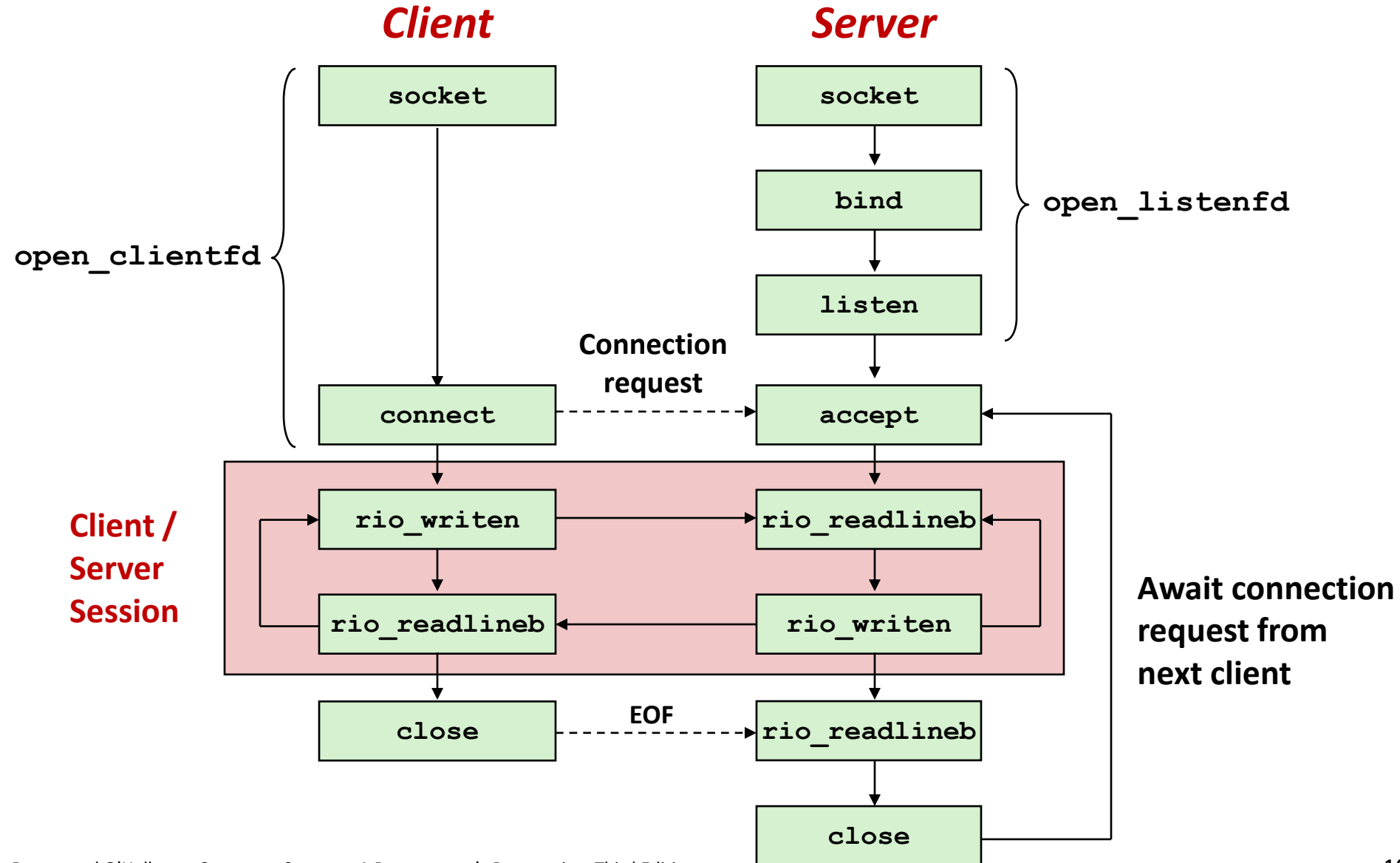
- **Classical problem classes of concurrent programs:**
 - ***Deadlock***: improper resource allocation prevents forward progress
 - Example: traffic gridlock
 - ***Starvation / Fairness***: external events and/or system scheduling decisions can prevent sub-task progress
 - Example: people always jump in front of you in line
 - ***Races***: outcome depends on arbitrary scheduling decisions elsewhere in the system
 - Example: who gets the last seat on the airplane?
- **Many aspects of concurrent programming are beyond the scope of our course...**
 - but, not all 😊
 - We'll cover some of these aspects in the next few lectures.

Concurrent Programming is Hard!

It may be hard, but ...

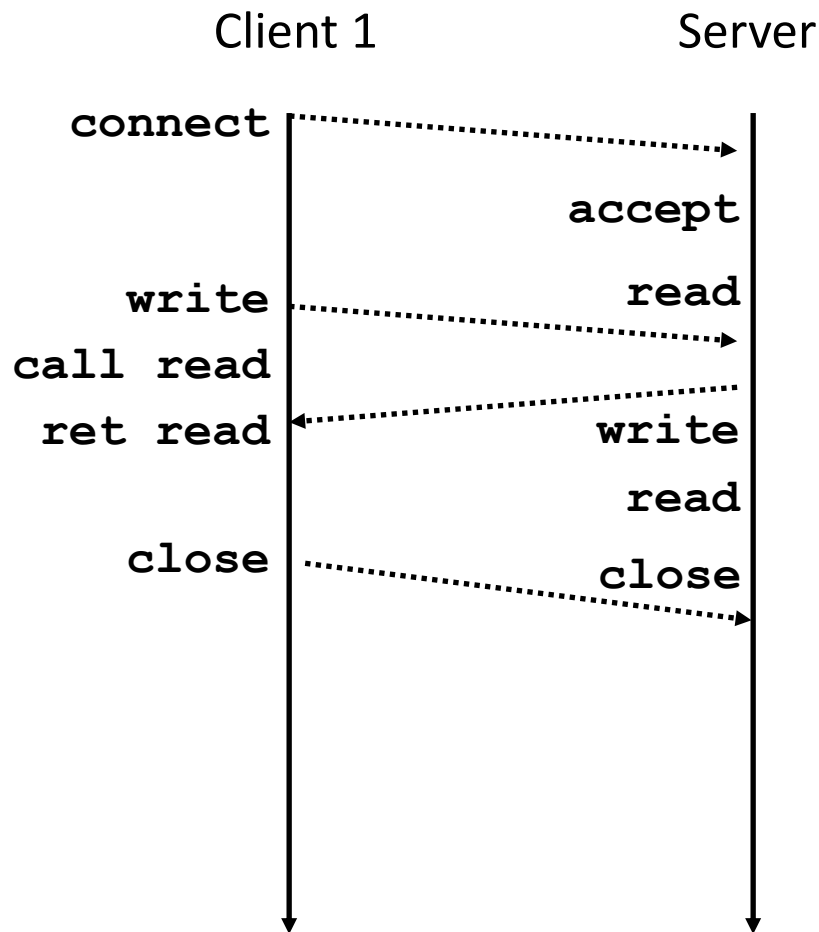
it can be useful and sometimes necessary!

Reminder: Iterative Echo Server



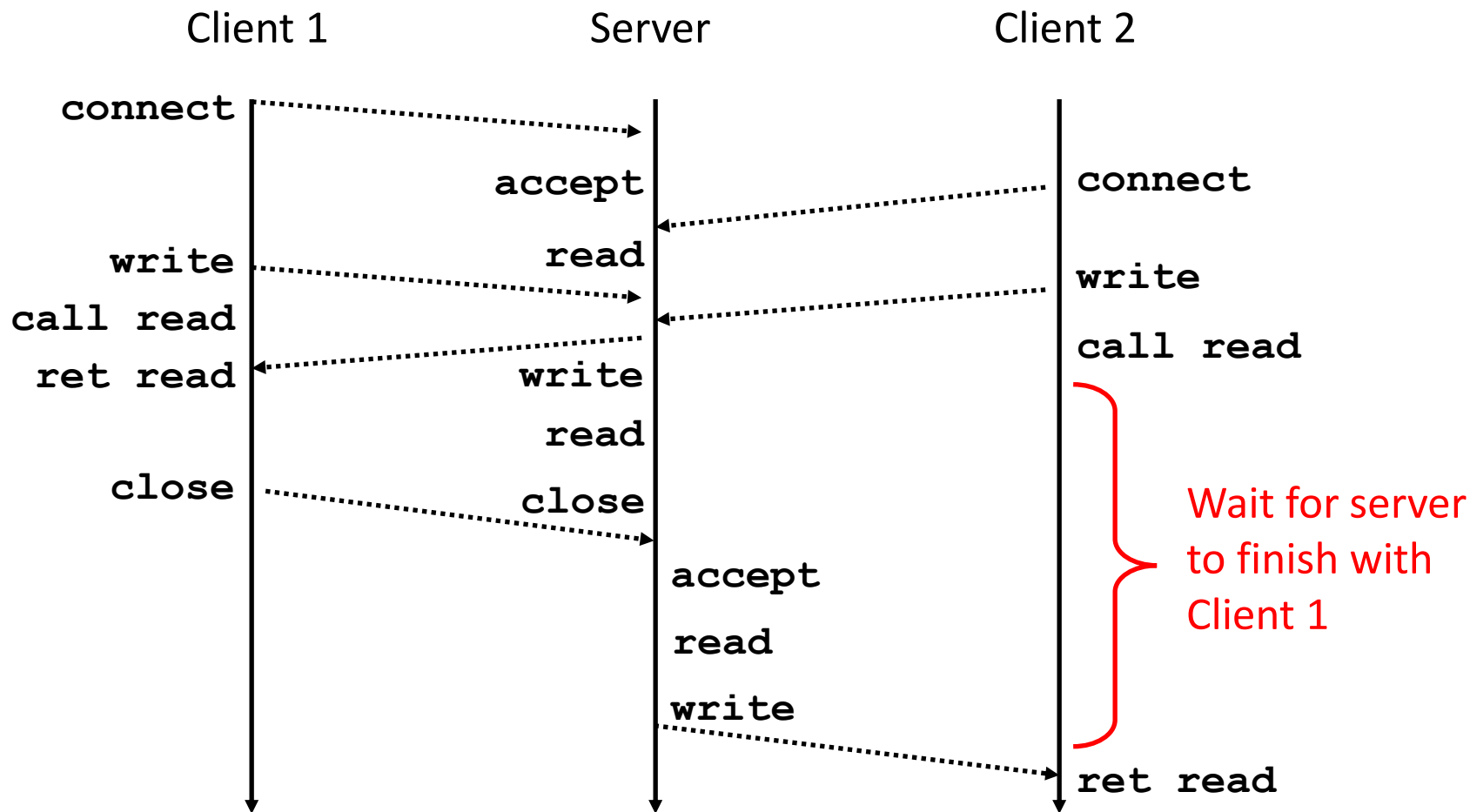
Iterative Servers

- Iterative servers process one connection at a time



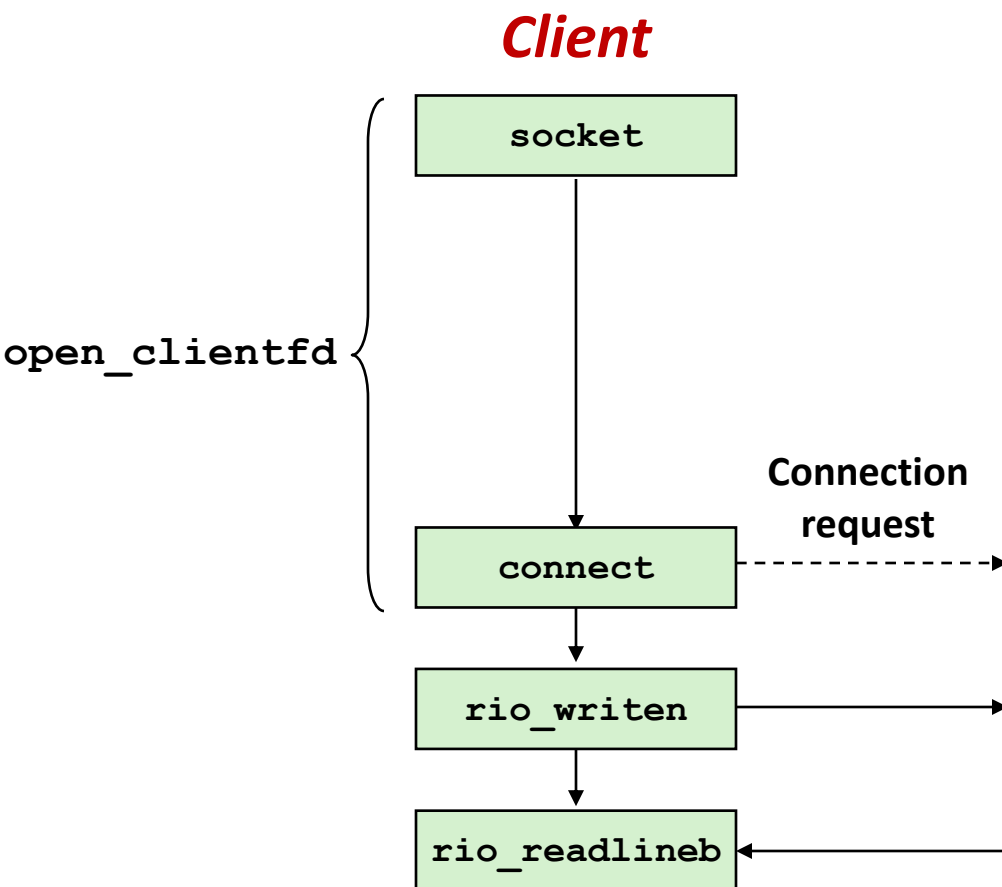
Iterative Servers

- Iterative servers process one request at a time



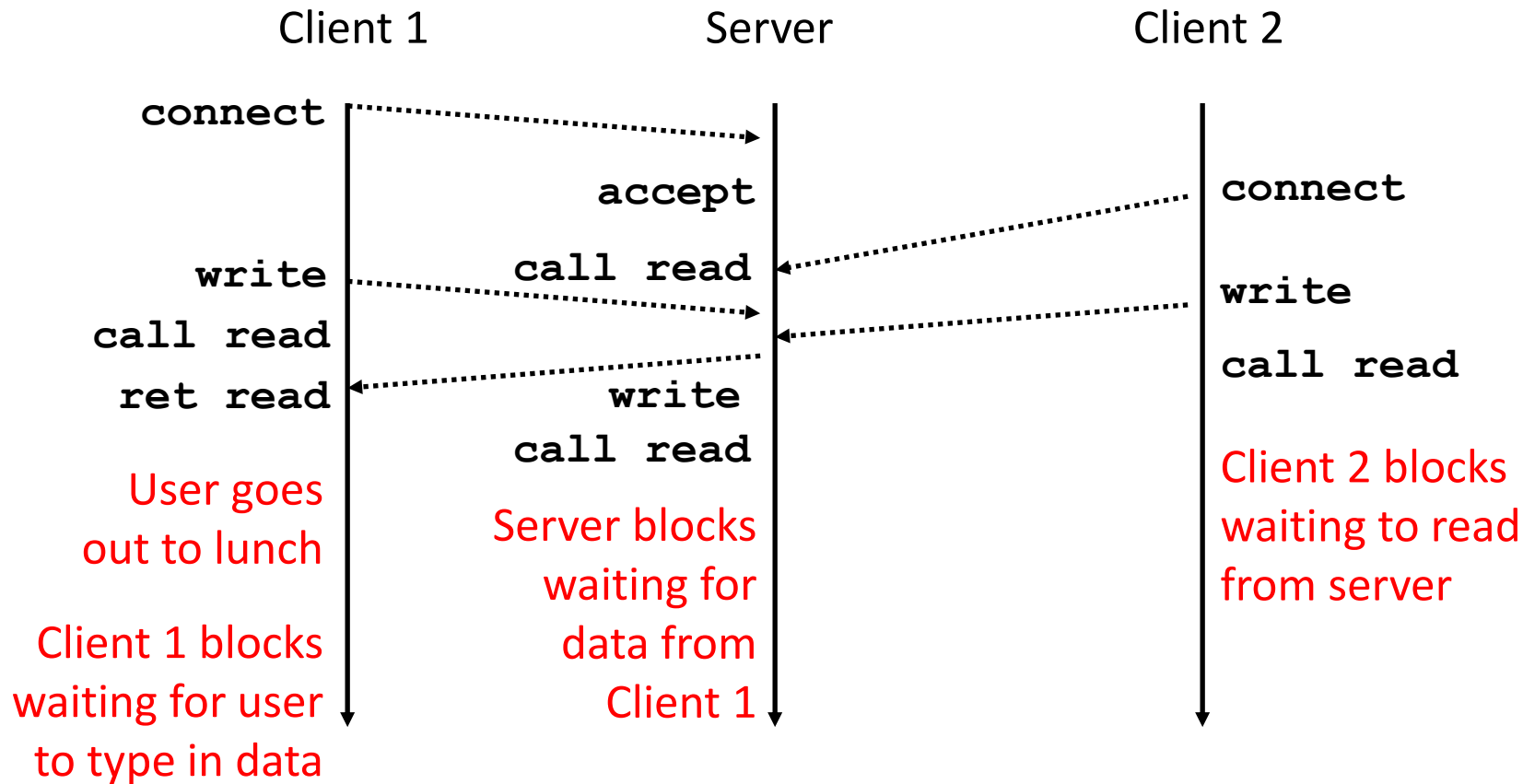
Where Does Second Client Block?

- Second client attempts to connect to iterative server



- Call to `connect` **returns**
 - Even though connection not yet accepted
 - Server side TCP manager queues request
 - Feature known as “TCP listen backlog”
- Call to `rio_writen` **returns**
 - Server side TCP manager buffers input data
- Call to `rio_readlineb` **blocks!**
 - Server hasn't written anything for it to read yet.

Fundamental Flaw of Iterative Servers



■ Solution: use *concurrent servers* instead

- Concurrent servers use multiple concurrent flows to serve multiple clients at the same time

Approaches for Writing Concurrent Servers

Allow server to handle multiple clients concurrently

1. Process-based

- Kernel automatically interleaves multiple logical flows
- Each flow has its own private address space

2. Event-based

- Programmer manually interleaves multiple logical flows
- All flows share the same address space
- Uses technique called *I/O multiplexing*

3. Thread-based

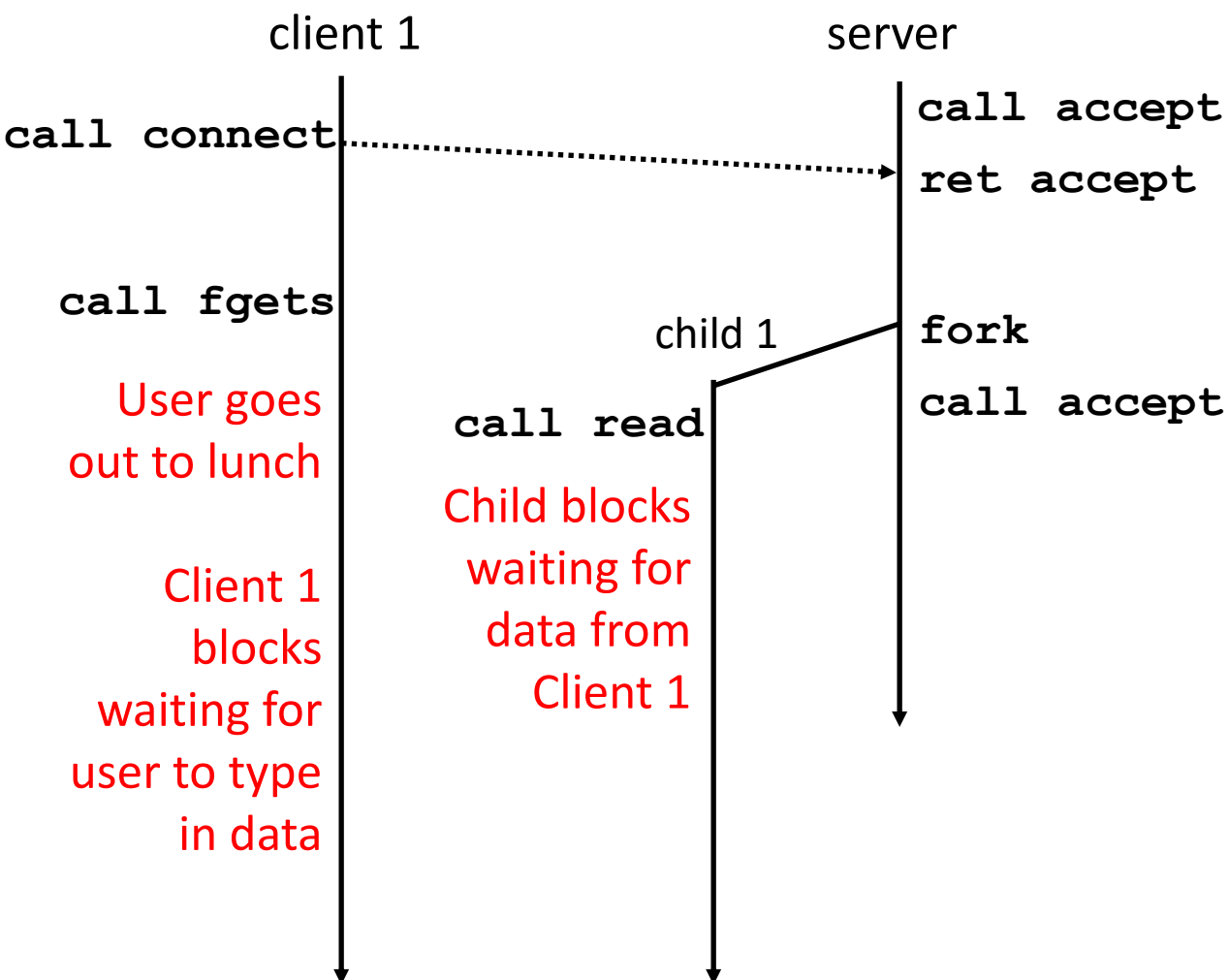
- Kernel automatically interleaves multiple logical flows
- Each flow shares the same address space
- Hybrid of process-based and event-based

Today

- Concurrent Programming Basics
- **Process-based Servers** CSAPP 12.1
- Event-based Servers CSAPP 12.2
- Thread-based Servers CSAPP 12.3

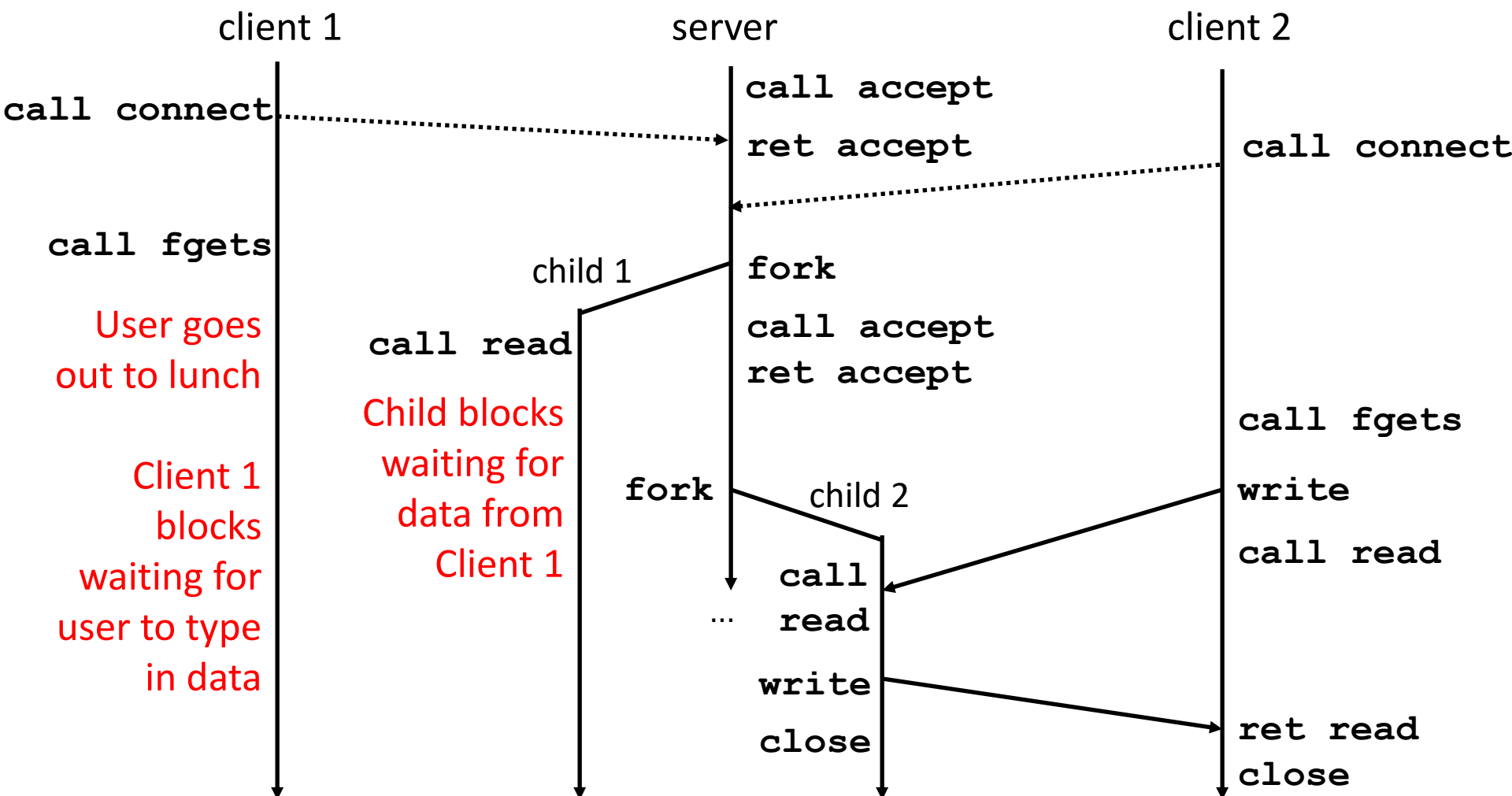
Approach #1: Process-based Servers

- Spawn separate process for each client



Approach #1: Process-based Servers

- Spawn separate process for each client



Iterative Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}
```

- Accept a connection request
- Handle echo requests until client terminates

echoserverp.c

Making a Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {

            echo(connfd);      /* Child services client */
            Close(connfd);    /* Child closes connection with client */
            exit(0);          /* Child exits */
        }
    }
}
```

echoserverp.c


Making a Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {

            echo(connfd);      /* Child services client */
            Close(connfd);    /* Child closes connection with client */
            exit(0);          /* Child exits */
        }
        Close(connfd); /* Parent closes connected socket (important!) */
    }
}
```

echoserverp.c



Why?

Making a Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {
            Close(listenfd); /* Child closes its listening socket */
            echo(connfd);    /* Child services client */
            Close(connfd);  /* Child closes connection with client */
            exit(0);        /* Child exits */
        }
        Close(connfd); /* Parent closes connected socket (important!) */
    }
}
```

echoserverp.c

Process-Based Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    Signal(SIGCHLD, sigchld_handler);
    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {
            Close(listenfd); /* Child closes its listening socket */
            echo(connfd);    /* Child services client */
            Close(connfd);  /* Child closes connection with client */
            exit(0);        /* Child exits */
        }
        Close(connfd); /* Parent closes connected socket (important!) */
    }
}
```

echoserverp.c

Process-Based Concurrent Echo Server (cont)

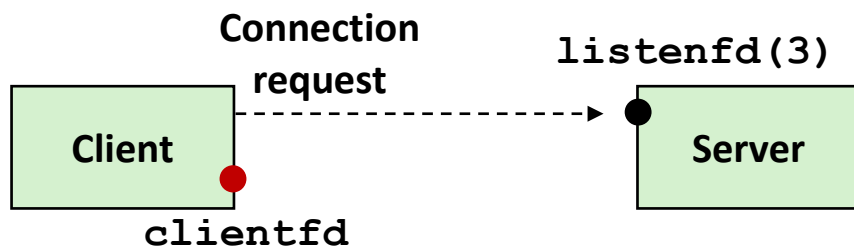
```
void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) > 0)
        ;
    return;
}
echoserverp.c
```

- Reap all zombie children

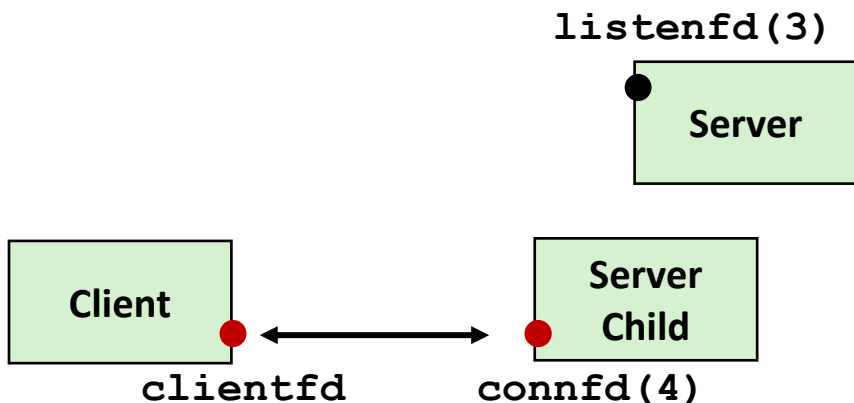
Concurrent Server: `accept` Illustrated



1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`

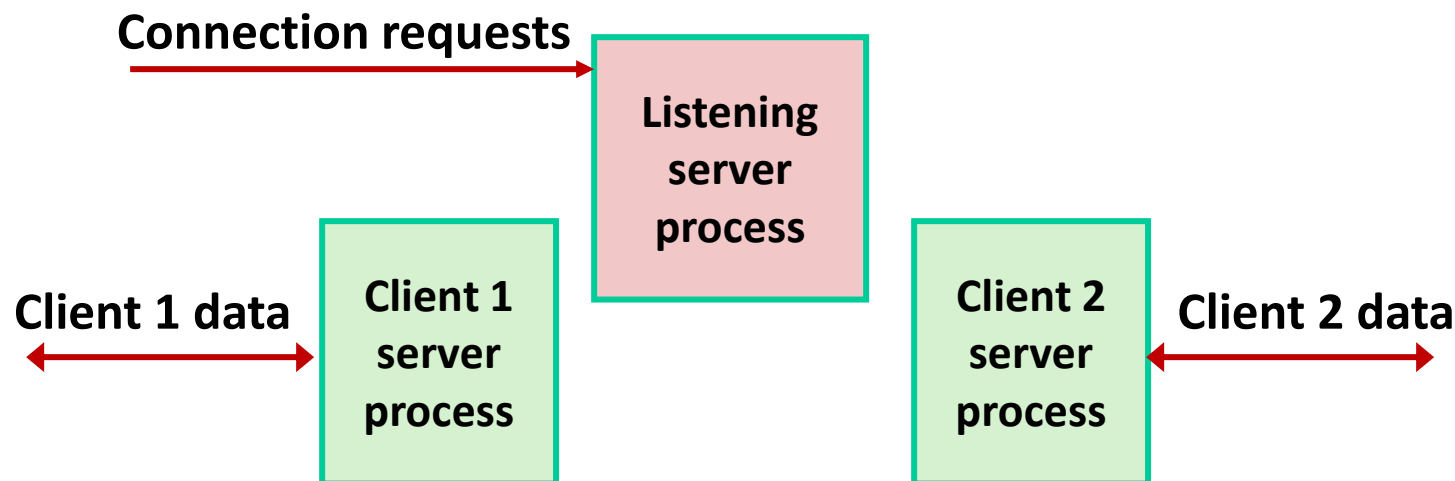


2. Client makes connection request by calling `connect`



3. Server returns `connfd` from `accept`. Forks child to handle client. Connection is now established between `clientfd` and `connfd`

Process-based Server Execution Model



- Each client handled by independent child process
- No shared state between them
- Both parent & child have copies of `listenfd` and `connfd`
 - Parent must close `connfd`
 - Child should close `listenfd`

Issues with Process-based Servers

- **Listening server process must reap zombie children**

- to avoid fatal memory leak

- **Parent process must `close` its copy of `connfd`**

- Kernel keeps reference count for each socket/open file

- After fork, `refcnt(connfd) = 2`

- Connection will not be closed until `refcnt(connfd) = 0`

Pros and Cons of Process-based Servers

- + Handle multiple connections concurrently.**
- + Clean sharing model.**
 - descriptors (no)
 - file tables (yes)
 - global variables (no)
- + Simple and straightforward.**
- Additional overhead for process control.**
- Nontrivial to share data between processes.**
 - (This example too simple to demonstrate)

Today

- Concurrent Programming Basics
- Process-based Servers CSAPP 12.1
- **Event-based Servers** CSAPP 12.2
- Thread-based Servers CSAPP 12.3

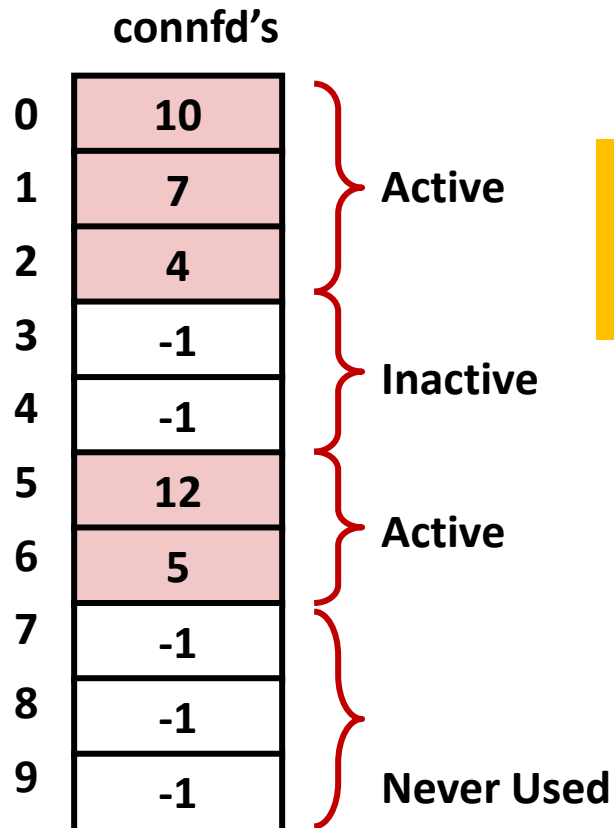
Approach #2: Event-based Servers

- **Server maintains set of active connections**
 - Array of `connfd`'s
- **Repeat:**
 - Determine which descriptors (`connfd`'s or `listenfd`) have pending inputs
 - e.g., using `select` function
 - arrival of pending input is an *event*
 - If `listenfd` has input, then `accept` connection
 - and add new `connfd` to array
 - Service all `connfd`'s with pending inputs
- **Details for select-based server in book**

I/O Multiplexed Event Processing

Active Descriptors

listenfd = 3



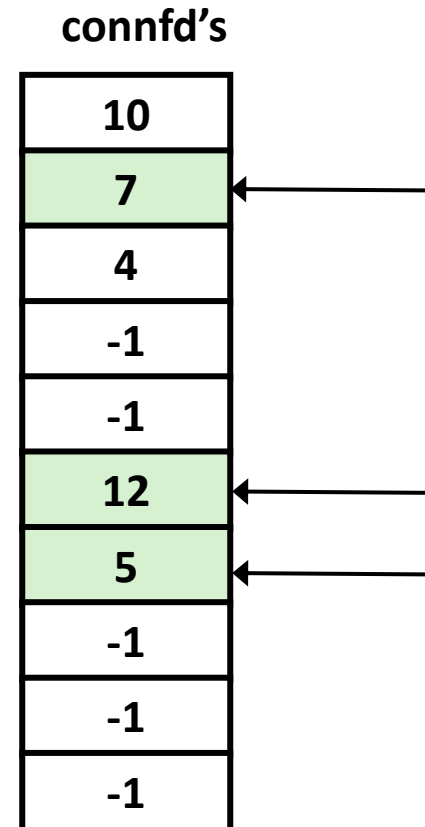
Anything
happened?

Read and
service

Read and service

Pending Inputs

listenfd = 3



Pros and Cons of Event-based Servers

- + One logical control flow and address space.**
- + Can single-step with a debugger.**
- + No process or thread control overhead.**
 - Design of choice for high-performance Web servers and search engines. e.g., Node.js, nginx, Tornado
- Significantly more complex to code than process-based or thread-based designs.**
- Hard to provide fine-grained concurrency.**
 - E.g., how to deal with partial HTTP request headers
- Cannot take advantage of multi-core.**
 - Single thread of control

Quiz Time!

Check out:

<https://canvas.cmu.edu/courses/17808>

Today

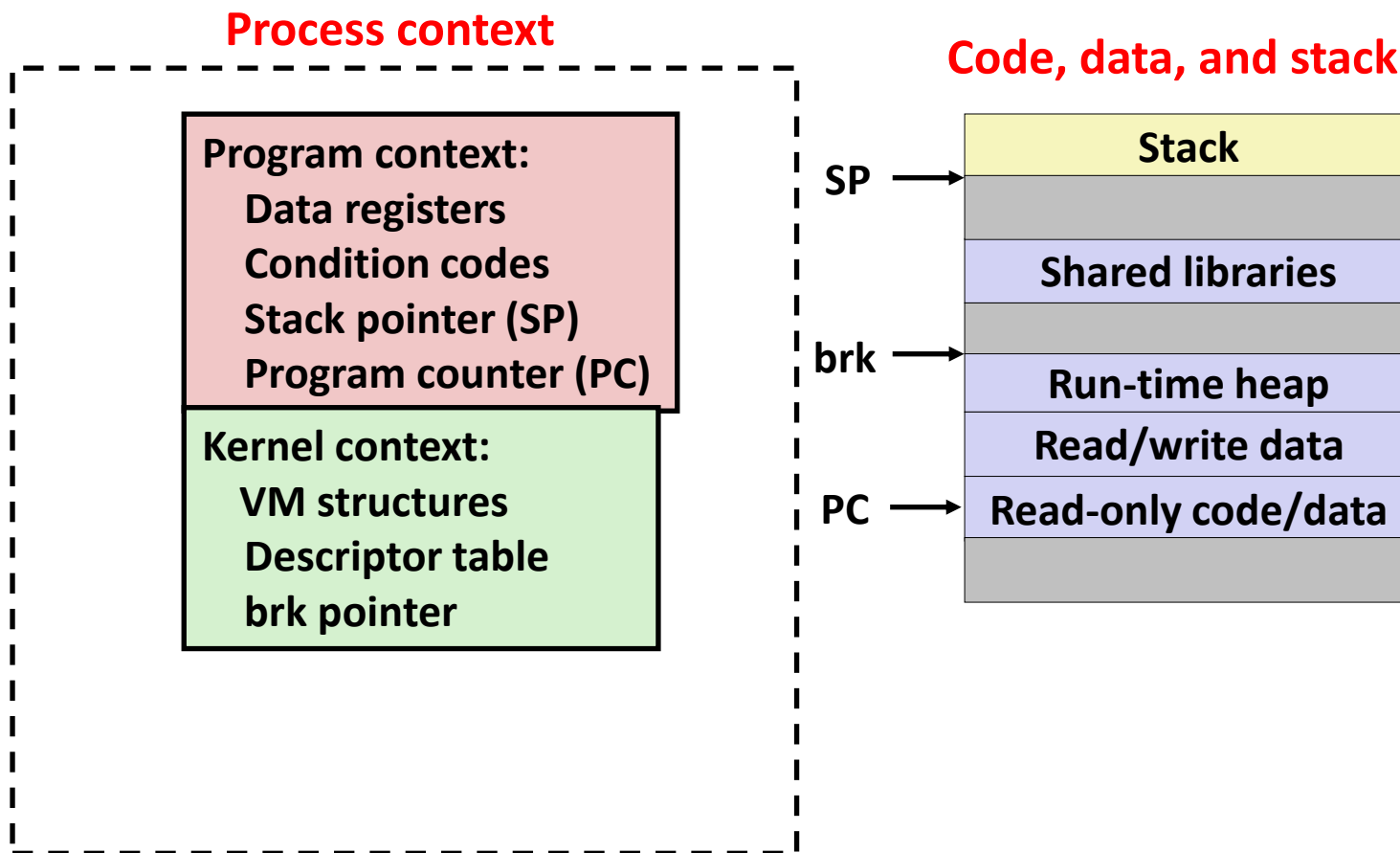
- Concurrent Programming Basics
- Process-based Servers CSAPP 12.1
- Event-based Servers CSAPP 12.2
- **Thread-based Servers** CSAPP 12.3

Approach #3: Thread-based Servers

- **Very similar to approach #1 (process-based)**
 - ...but using threads instead of processes

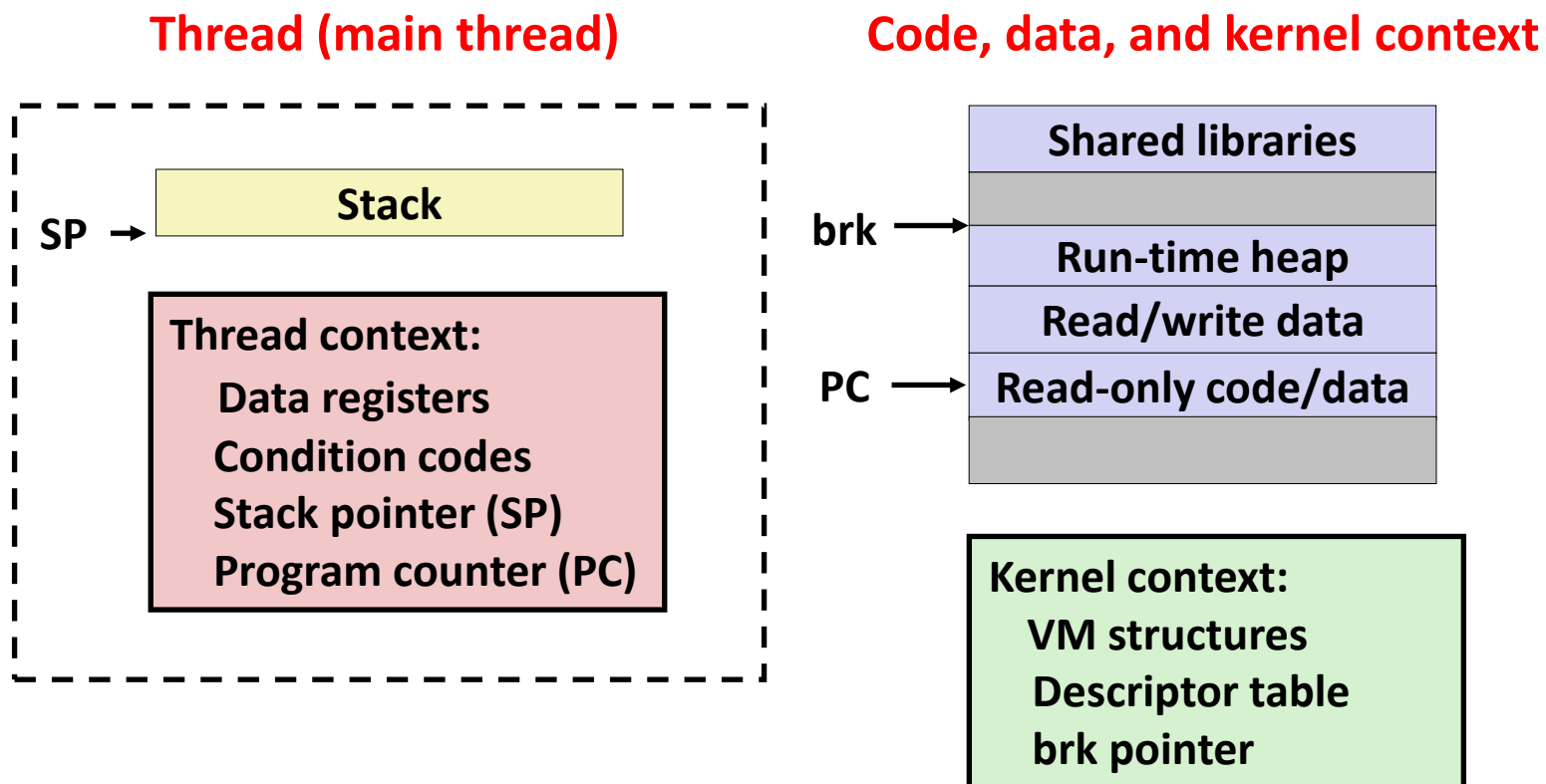
Traditional View of a Process

- Process = process context + code, data, and stack



Alternate View of a Process

- Process = thread + code, data, and kernel context



A Process With Multiple Threads

- **Multiple threads can be associated with a process**
 - Each thread has its own logical control flow
 - Each thread shares the same code, data, and kernel context
 - Each thread has its own stack for local variables
 - but not protected from other threads
 - Each thread has its own thread id (TID)

Thread 1 (main thread)

Thread 2 (peer thread)

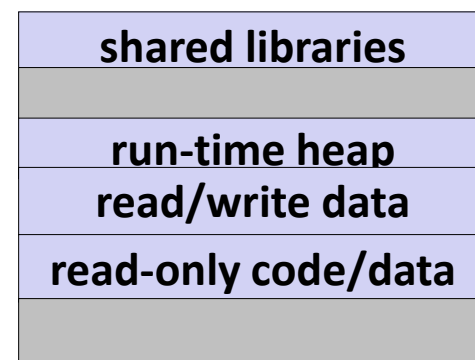
Shared code and data

stack 1

stack 2

Thread 1 context:
 Data registers
 Condition codes
 SP_1
 PC_1

Thread 2 context:
 Data registers
 Condition codes
 SP_2
 PC_2

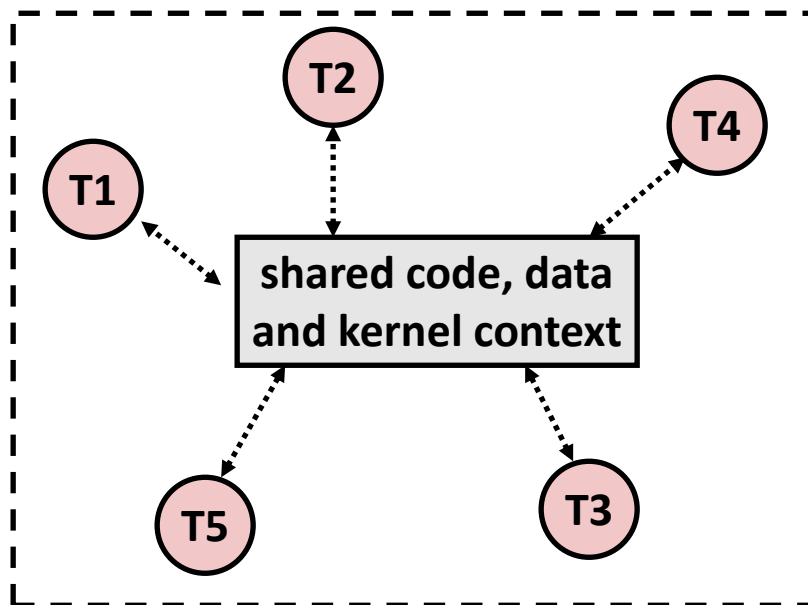


Kernel context:
 VM structures
 Descriptor table
 brk pointer

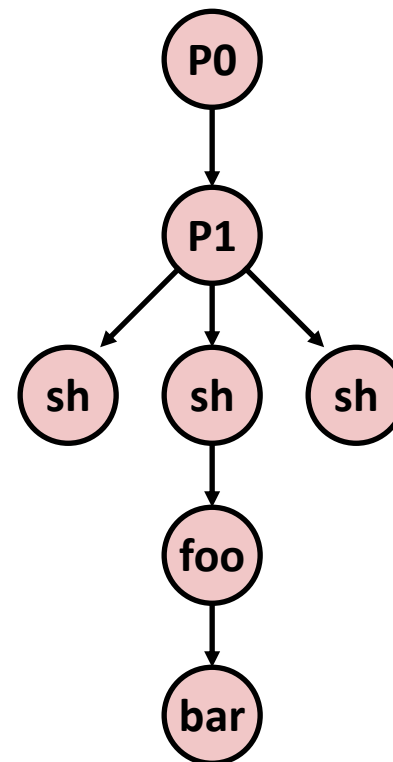
Logical View of Threads

- **Threads associated with process form a pool of peers**
 - Unlike processes which form a tree hierarchy

Threads associated with process foo



Process hierarchy

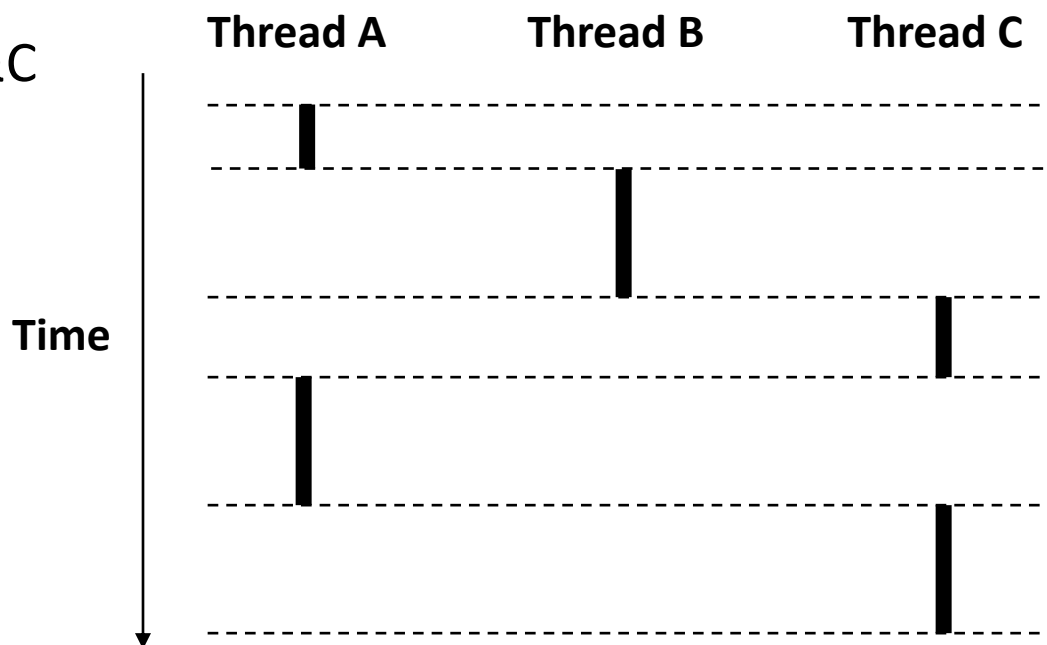


Concurrent Threads

- Two threads are *concurrent* if their flows overlap in time
- Otherwise, they are sequential

- **Examples:**

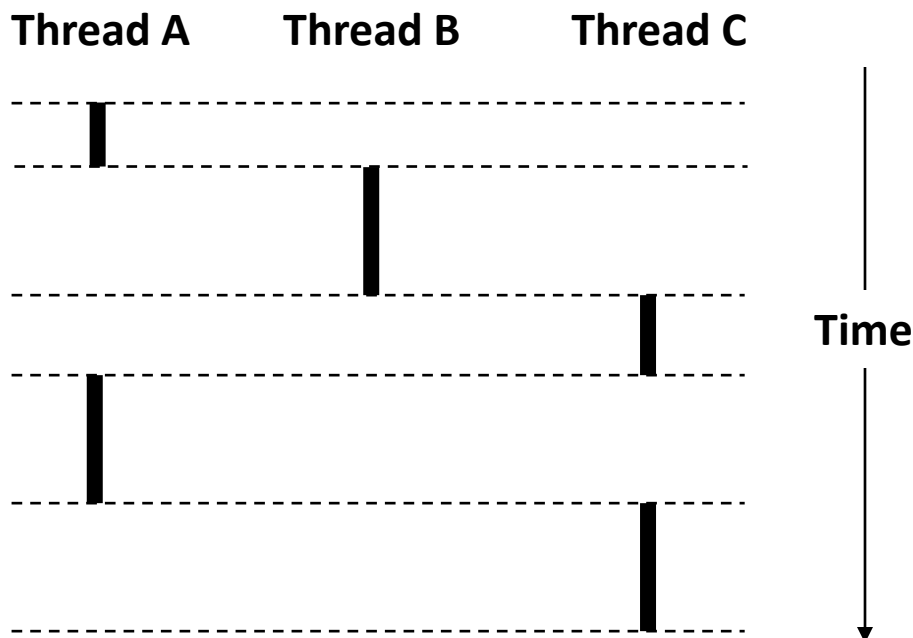
- Concurrent: A & B, A&C
- Sequential: B & C



Concurrent Thread Execution

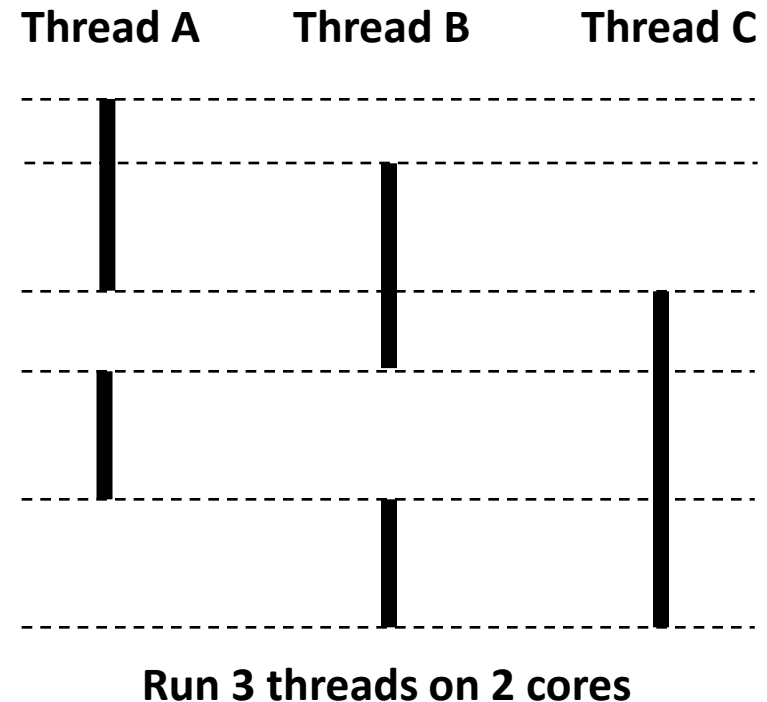
■ Single Core Processor

- Simulate parallelism by time slicing



■ Multi-Core Processor

- Can have true parallelism



Threads vs. Processes

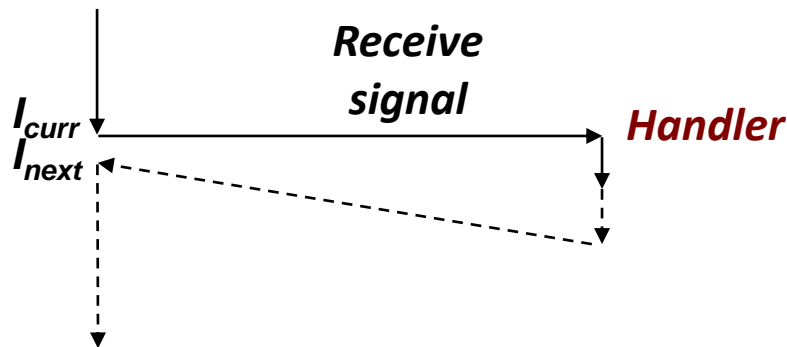
■ How threads and processes are similar

- Each has its own logical control flow
- Each can run concurrently with others (possibly on different cores)
- Each is context switched

■ How threads and processes are different

- Threads share all code and data (except local stacks)
 - Processes (typically) do not
- Threads are somewhat less expensive than processes
 - Process control (creating and reaping) twice as expensive as thread control
 - Linux numbers:
 - ~20K cycles to create and reap a process
 - ~10K cycles (or less) to create and reap a thread

Threads vs. Signals



- **Signal handler shares state with regular program**
 - Including stack
- **Signal handler interrupts normal program execution**
 - Unexpected procedure call
 - Returns to regular execution stream
 - *Not* a peer
- **Limited forms of synchronization**
 - Main program can block / unblock signals
 - Main program can pause for signal

Posix Threads (Pthreads) Interface

- ***Pthreads*: Standard interface for ~60 functions that manipulate threads from C programs**
 - Creating and reaping threads
 - `pthread_create()`
 - `pthread_join()`
 - Determining your thread ID
 - `pthread_self()`
 - Terminating threads
 - `pthread_cancel()`
 - `pthread_exit()`
 - `exit()` [terminates all threads]
 - `return` [terminates current thread]
 - Synchronizing access to shared variables
 - `pthread_mutex_init`
 - `pthread_mutex_[un]lock`

The Pthreads "hello, world" Program

```

/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main(int argc, char** argv)
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    return 0;
}

```

Thread ID

Thread attributes
(usually NULL)

Thread routine

Thread arguments
(void *p)

hello.c

Return value
(void **p)

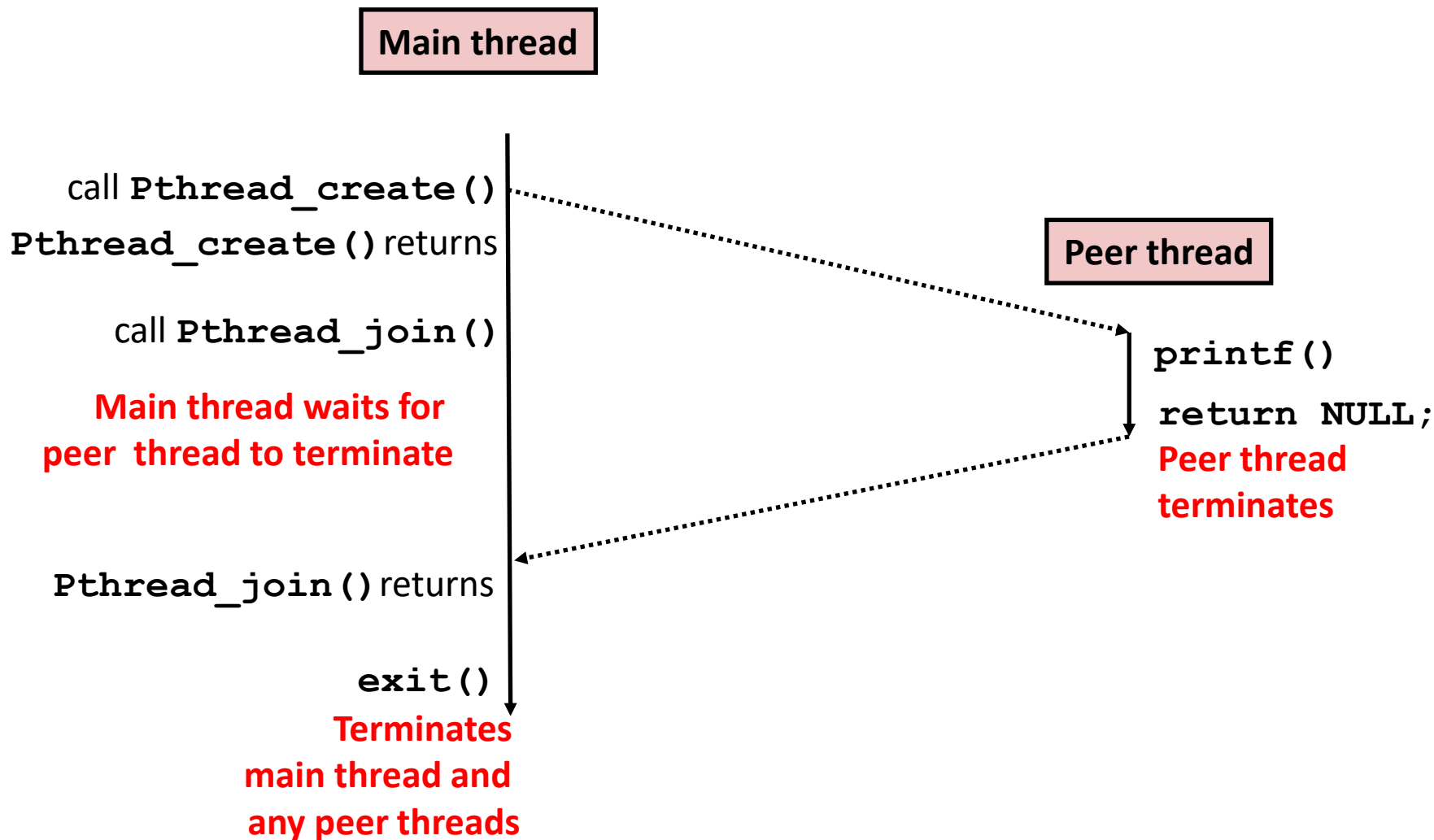
```

void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}

```

hello.c

Execution of Threaded “hello, world”



Thread-Based Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, *connfdp;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen=sizeof(struct sockaddr_storage);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
    return 0;
}
```

echoservt.c

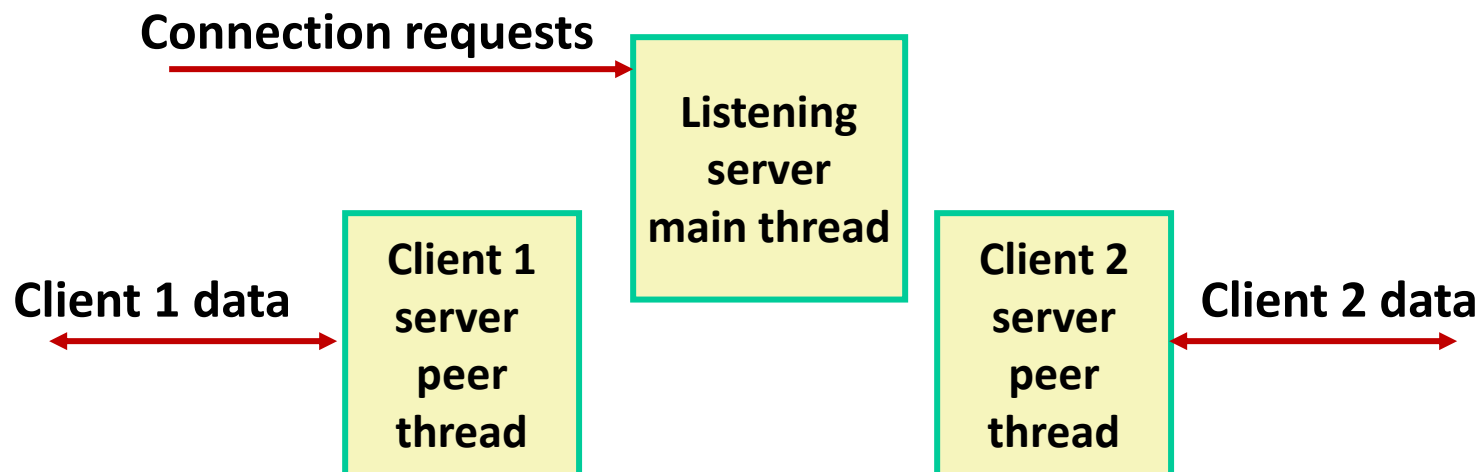
- Spawn new thread for each client
- Pass it copy of connection file descriptor
- Note use of **Malloc ()** ! [but not **Free ()**]

Thread-Based Concurrent Server (cont)

```
/* Thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
    Pthread_detach(pthread_self());
    Free(vargp);
    echo(connfd);
    Close(connfd);
    return NULL;
}
echoserv.c
```

- Run thread in “detached” mode.
 - Runs independently of other threads
 - Reaped automatically (by kernel) when it terminates
- Free storage allocated to hold `connfd`.
- Close `connfd` (important!)

Thread-based Server Execution Model



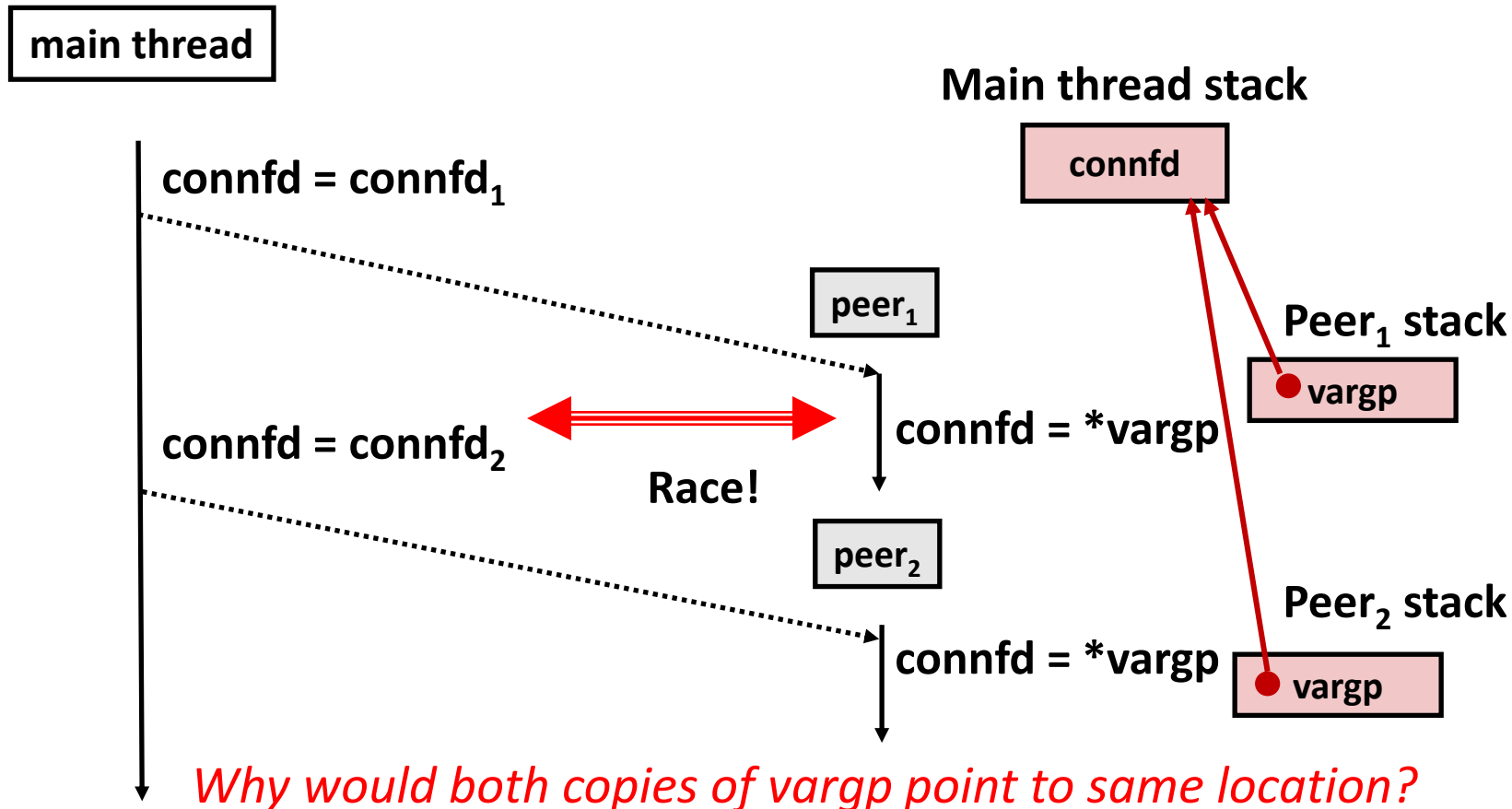
- Each client handled by individual peer thread
- Threads share all process state except TID
- Each thread has a separate stack for local variables

Issues With Thread-Based Servers

- **Run “detached” to automatically reap/cleanup threads**
 - At any point in time, a thread is either *joinable* or *detached*
 - *Joinable* thread can be reaped and killed by other threads
 - must be reaped (with `pthread_join`) to free memory resources
 - *Detached* thread cannot be reaped or killed by other threads
 - resources are automatically reaped on termination
 - Default state is joinable
 - use `pthread_detach(pthread_self())` to make detached
- **Must be careful to avoid unintended sharing**
 - For example, passing pointer to main thread's stack
 - `Pthread_create(&tid, NULL, thread, (void *) &connfd);`
- **All functions called by a thread must be *thread-safe***
 - (next lecture)

Potential Form of Unintended Sharing

```
while (1) {
    int connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
    Pthread_create(&tid, NULL, thread, &connfd);
}
```



Could this race occur?

Main

```
int i;
for (i = 0; i < 100; i++) {
    Pthread_create(&tid, NULL,
                  thread, &i);
}
```

Thread

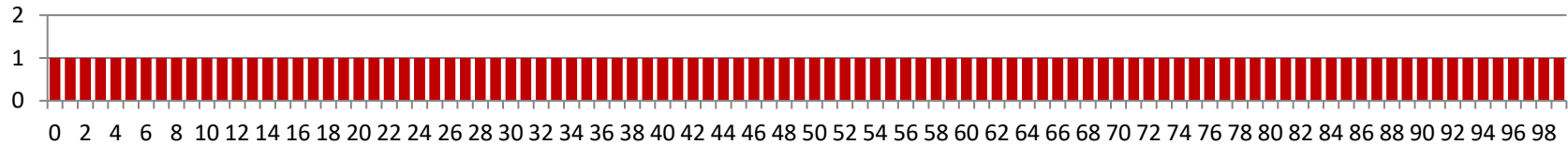
```
void *thread(void *vargp)
{
    int i = *((int *)vargp);
    Pthread_detach(pthread_self());
    save_value(i);
    return NULL;
}
```

■ Race Test

- If no race, then each thread would get different value of **i**
- Set of saved values would consist of one copy each of 0 through 99

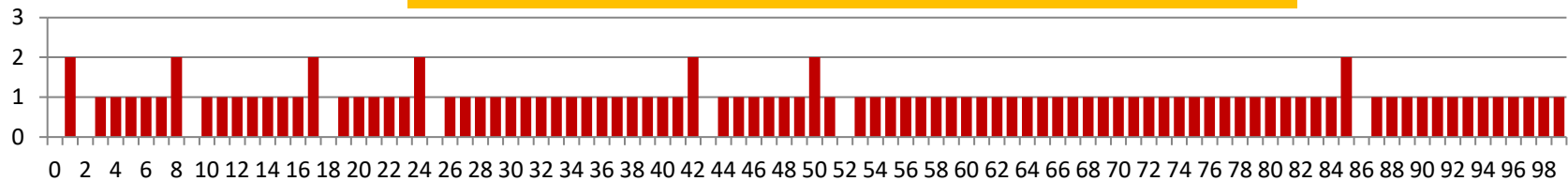
Experimental Results

No Race



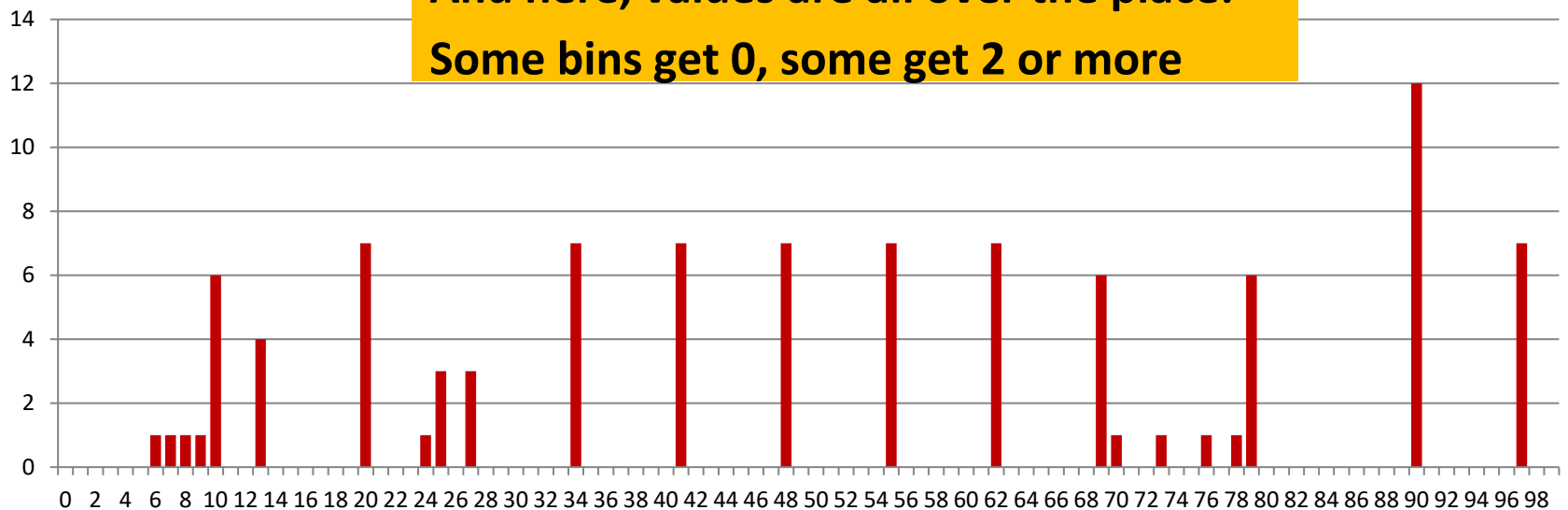
Single core laptop

For each "0" there is some later "2" here



Multicore server

And here, values are all over the place:
Some bins get 0, some get 2 or more



Correct passing of thread arguments

```
/* Main routine */
int *connfdp;
connfdp = Malloc(sizeof(int));
*connfdp = Accept( . . . );
Pthread_create(&tid, NULL, thread, connfdp);
```

```
/* Thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
    . . .
    Free(vargp);
    . . .
    return NULL;
}
```

- Producer-Consumer Model
 - Allocate in main
 - Free in thread routine

Pros and Cons of Thread-Based Designs

- + Easy to share data structures between threads**
 - e.g., logging information, file cache
- + Threads are more efficient than processes**
- Unintentional sharing can introduce subtle and hard-to-reproduce errors!**
 - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
 - Hard to know which data shared & which private
 - Hard to detect by testing
 - Probability of bad race outcome very low
 - But nonzero!
 - Future lectures

Summary: Approaches to Concurrency

■ Process-based

- Hard to share resources: Easy to avoid unintended sharing
- High overhead in adding/removing clients

■ Event-based

- Tedious and low level
- Total control over scheduling
- Very low overhead
- Cannot create as fine grained a level of concurrency
- Does not make use of multi-core

■ Thread-based

- Easy to share resources: Perhaps too easy
- Medium overhead
- Not much control over scheduling policies
- Difficult to debug: Event orderings not repeatable