Concurrent Programming

CS230: System Programming
17th Lecture

Instructor:

Jongse Park

Concurrent Programming is Hard!

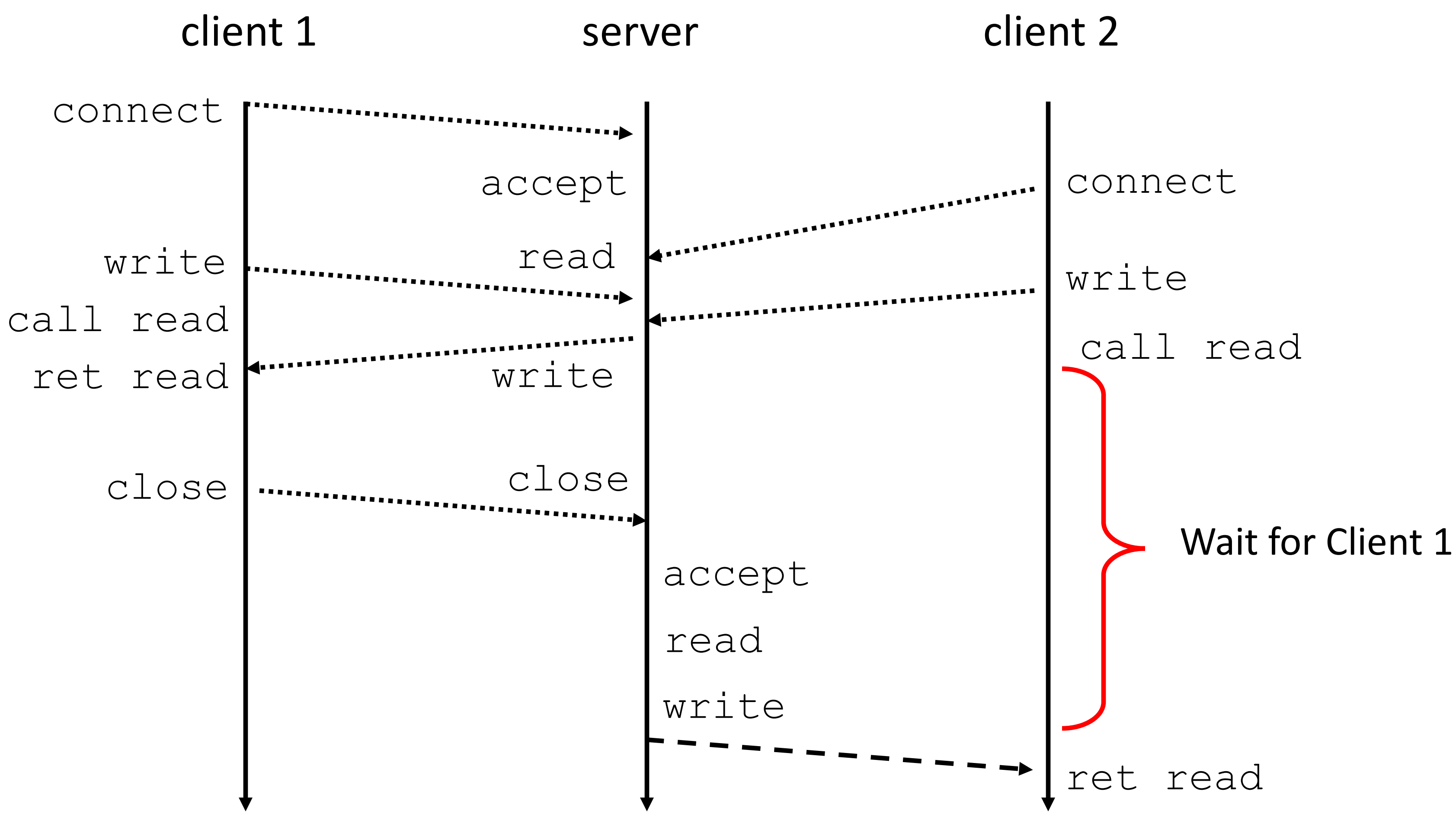
- The human mind tends to be sequential
- The notion of time is often misleading
- Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible

Concurrent Programming is Hard!

- Classical problem classes of concurrent programs:
  - ***Races:*** outcome depends on arbitrary scheduling decisions elsewhere in the system
    - Example: who gets the last seat on the airplane?
  - ***Deadlock:*** improper resource allocation prevents forward progress
    - Example: traffic gridlock
  - ***Livelock / Starvation / Fairness***: external events and/or system scheduling decisions can prevent sub-task progress
    - Example: people always jump in front of you in line
- Many aspects of concurrent programming are beyond the scope of 15-213

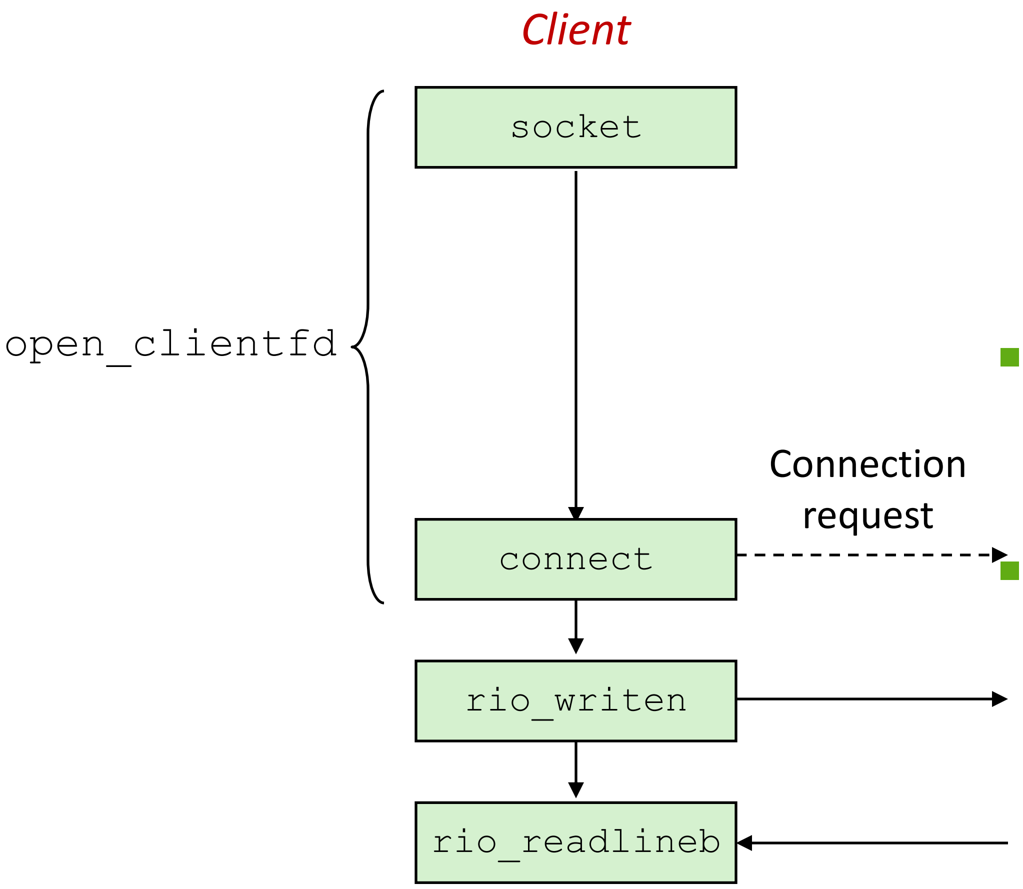
Iterative Servers

- Iterative servers process one request at a time



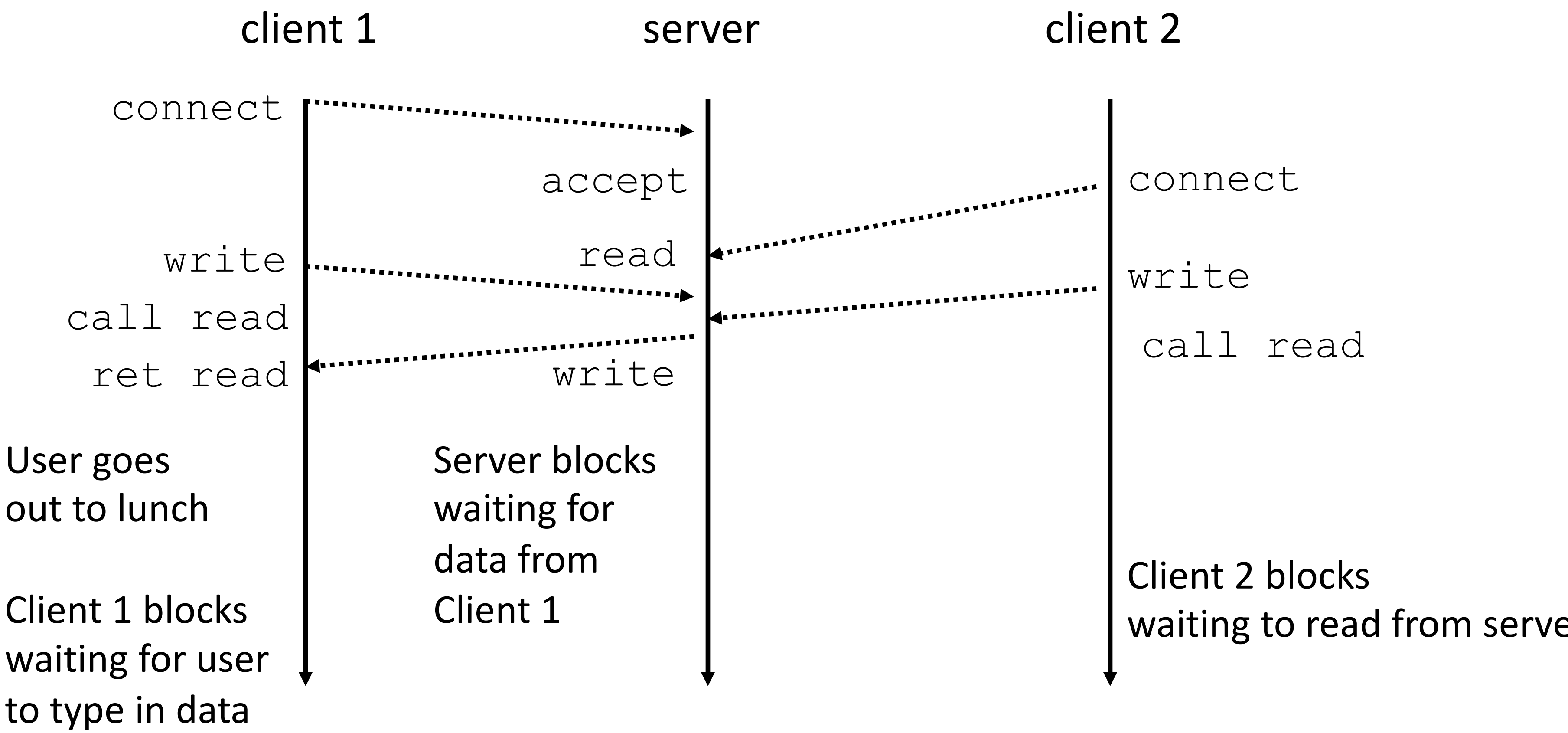
Where Does Second Client Block?

- 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



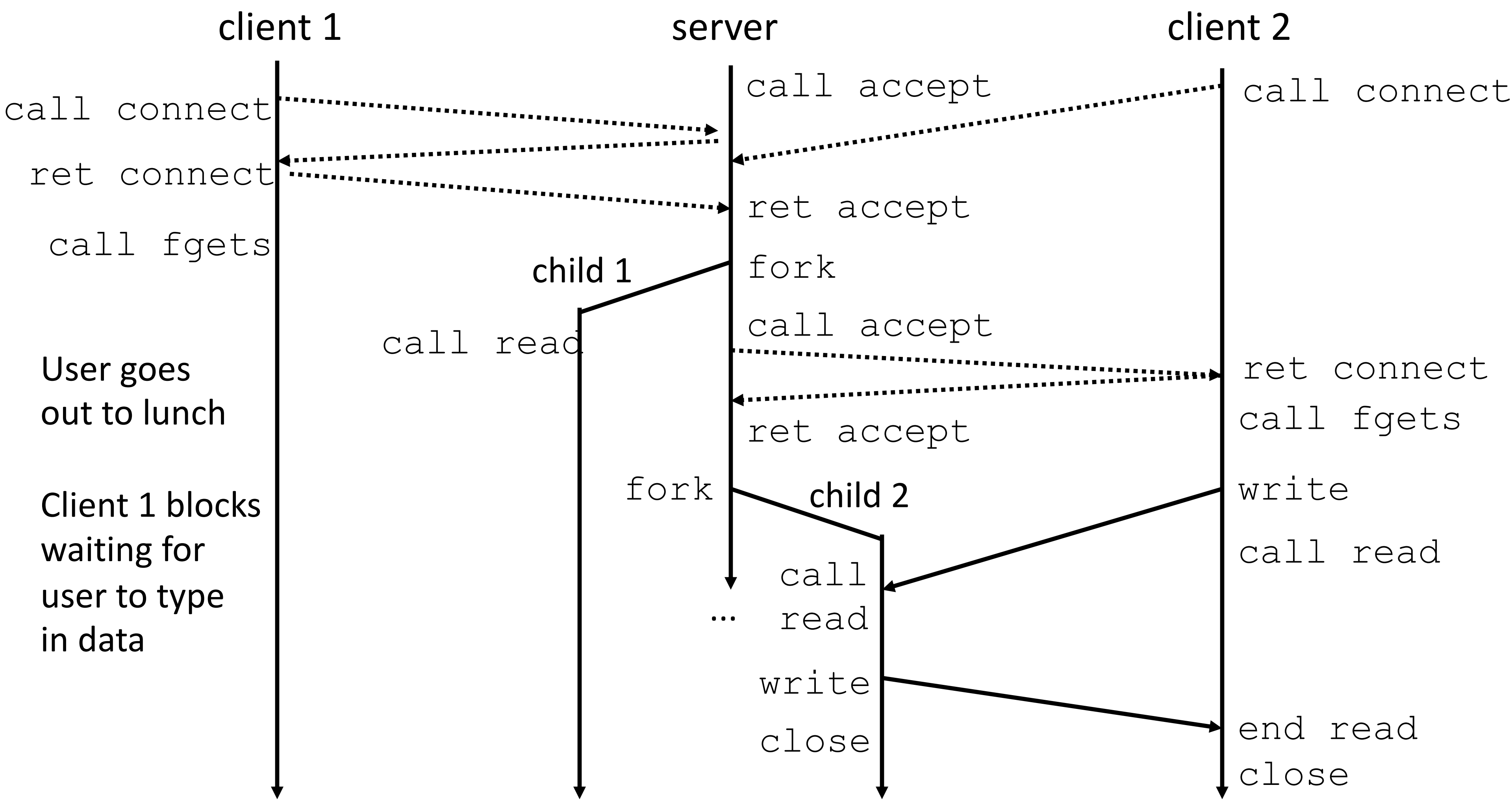
- Solution: use *concurrent servers* instead
  - Concurrent servers use multiple concurrent flows to serve multiple clients at the same time

Creating Concurrent Flows

  - Allow server to handle multiple clients simultaneously
- 1. Processes
  - Kernel automatically interleaves multiple logical flows
  - Each flow has its own private address space
- 2. Threads
  - Kernel automatically interleaves multiple logical flows
  - Each flow shares the same address space
- 3. I/O multiplexing with `select()`
  - Programmer manually interleaves multiple logical flows
  - All flows share the same address space
  - Relies on lower-level system abstractions

Concurrent Servers: Multiple Processes

- Spawn separate process for each client



Review: Iterative Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    int port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen = sizeof(clientaddr);

    listenfd = open_listenfd(port);
    while (1) {
       connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);
       echo(connfd);
       close(connfd);
    }
    exit(0);
}
```

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

Process-Based Concurrent Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    int port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen=sizeof(clientaddr);

    signal(SIGCHLD, sigchld_handler);
    listenfd = open_listenfd(port);
    while (1) {
        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!) */
    }
}
```

Fork separate process for each client

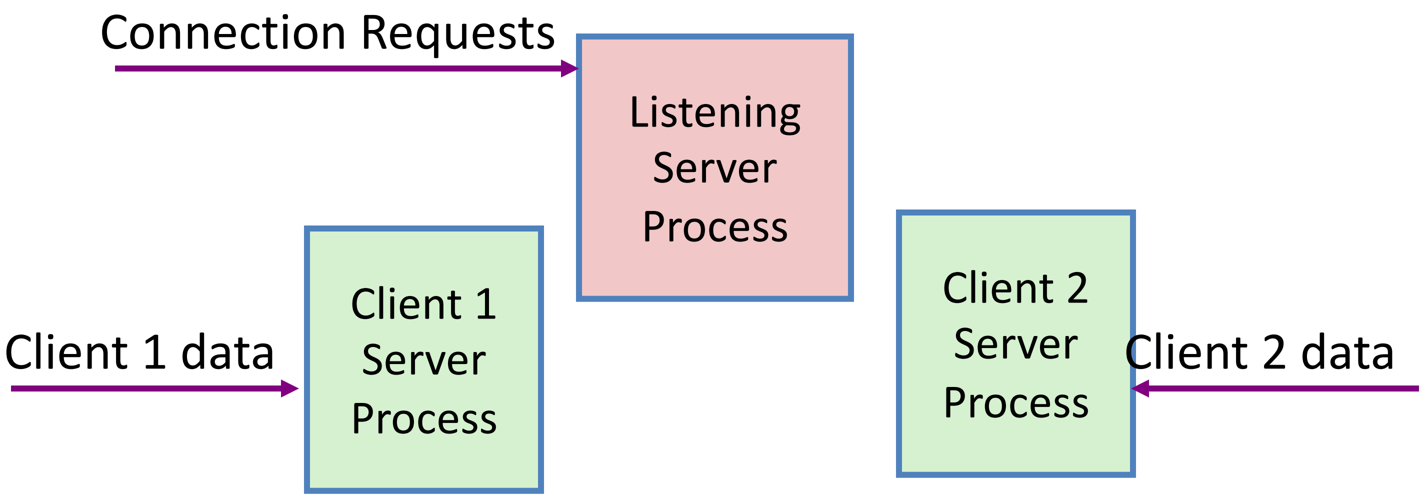
Does not allow any communication between different client handlers

Process-Based Concurrent Server (cont)

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

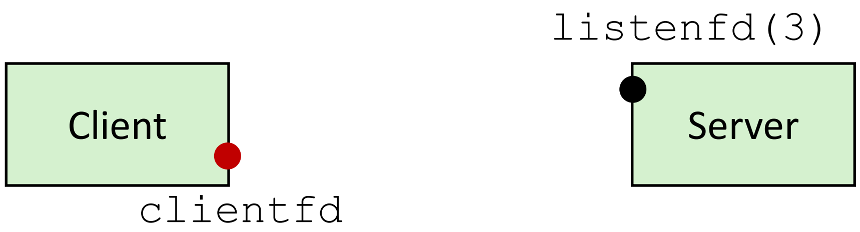
- Reap all zombie children

Process Execution Model

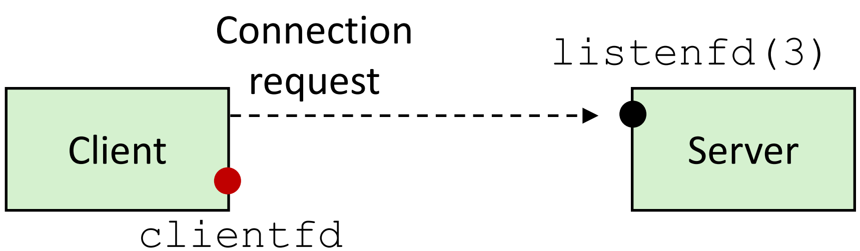


- Each client handled by independent process
- No shared state between them
- Both parent & child have copies of listenfd and connfd
  - Parent must close connfd
  - Child must close listenfd

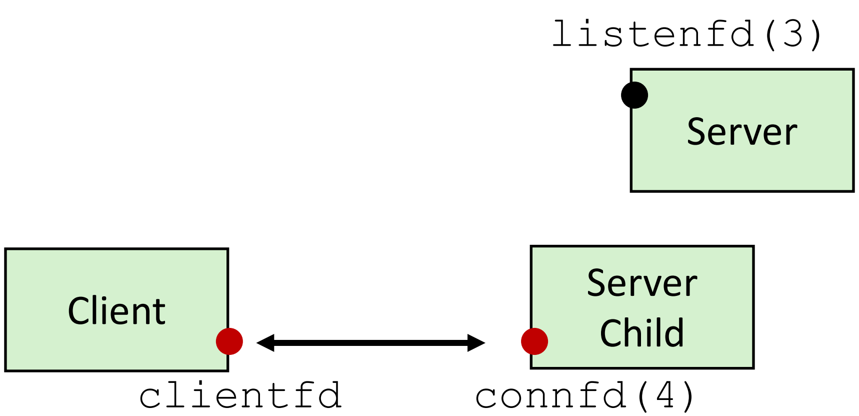
Concurrent Server: `accept` Illustrated



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



2. Client makes connection request by calling and blocking in `connect`



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

Implementation Must-dos With Process-Based Designs

- Listening server process must reap zombie children
  - to avoid fatal memory leak
- Listening server process must `close` its copy of `connfd`
  - Kernel keeps reference for each socket/open file
  - After fork, `refcnt(connfd) = 2`
  - Connection will not be closed until `refcnt(connfd) == 0`

View from Server's TCP Manager

Client 1 Client 2 Server

```
srv> ./echoserverp 15213
```

```
cl1> ./echoclient greatwhite.ics.cs.cmu.edu 15213
```

```
srv> connected to (128.2.192.34), port 50437
```

```
cl2> ./echoclient greatwhite.ics.cs.cmu.edu 15213
```

```
srv> connected to (128.2.205.225), port 41656
```

Connection	Host	Port	Host	Port
Listening	---	---	128.2.220.10	15213
cl1	128.2.192.34	50437	128.2.220.10	15213
cl2	128.2.205.225	41656	128.2.220.10	15213

View from Server's TCP Manager

Connection	Host	Port	Host	Port
Listening	---	---	128.2.220.10	15213
cl1	128.2.192.34	50437	128.2.220.10	15213
cl2	128.2.205.225	41656	128.2.220.10	15213

- Port Demultiplexing
  - TCP manager maintains separate stream for each connection
    - Each represented to application program as socket
    - New connections directed to listening socket
    - Data from clients directed to one of the connection sockets

Pros and Cons of Process-Based Designs

- + 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
  - Requires IPC (interprocess communication) mechanisms
    - FIFO’s (named pipes), shared memory, and semaphores

Approach #2: Multiple Threads

- Very similar to approach #1 (multiple processes)
  - but, with threads instead of processes

Traditional View of a Process

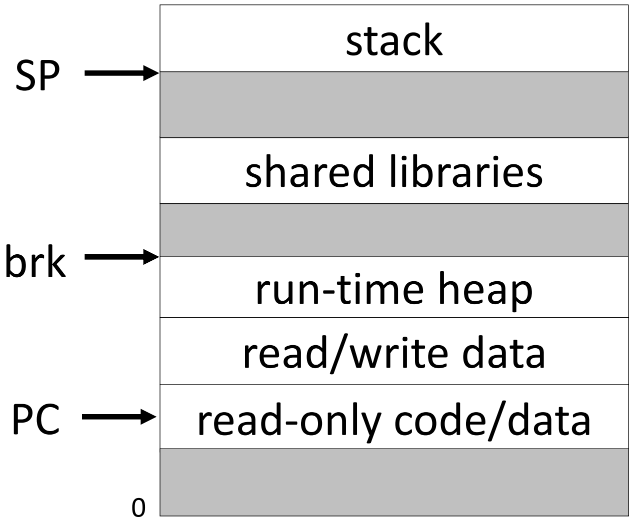
- Process = process context + code, data, and stack

Process context

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

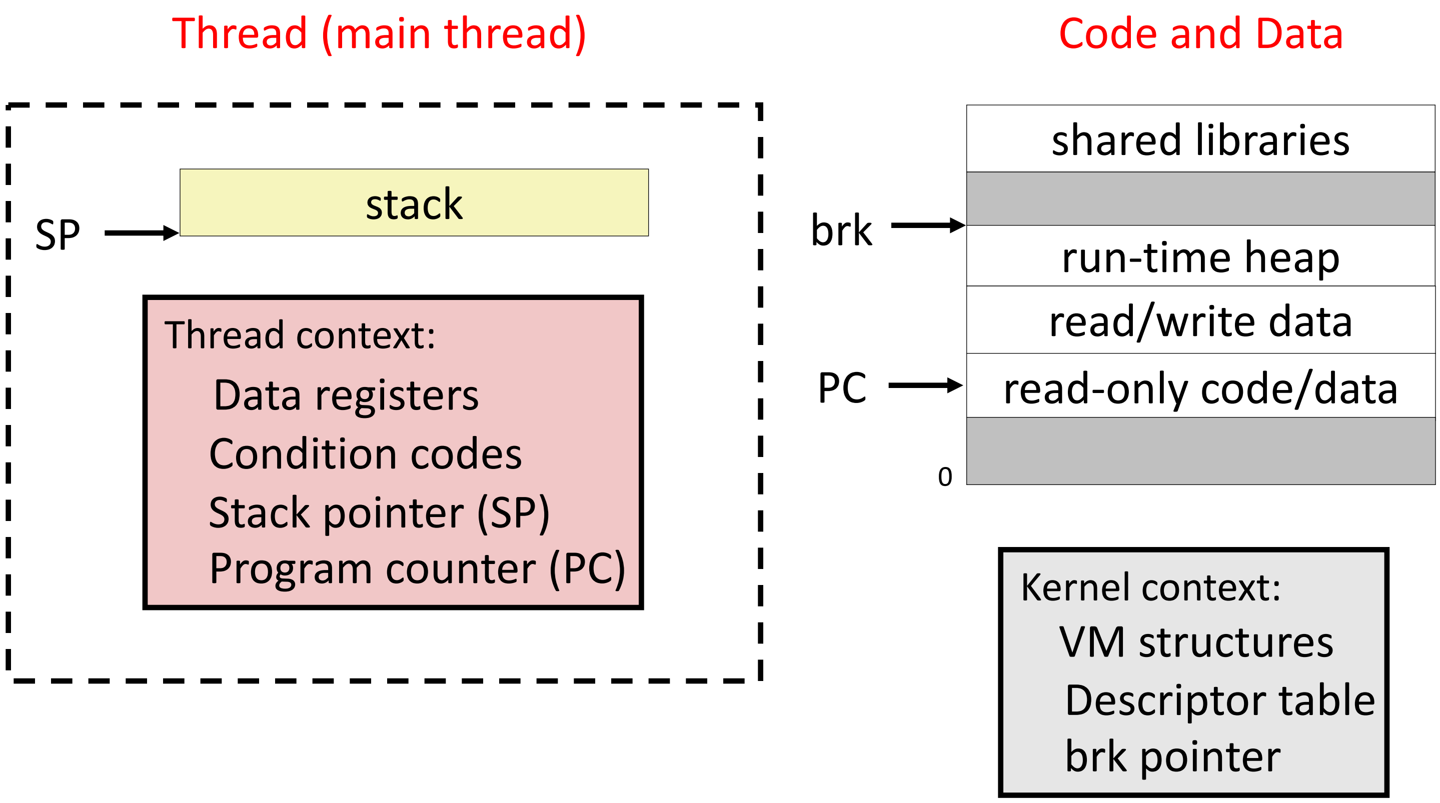
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Code, data, and stack



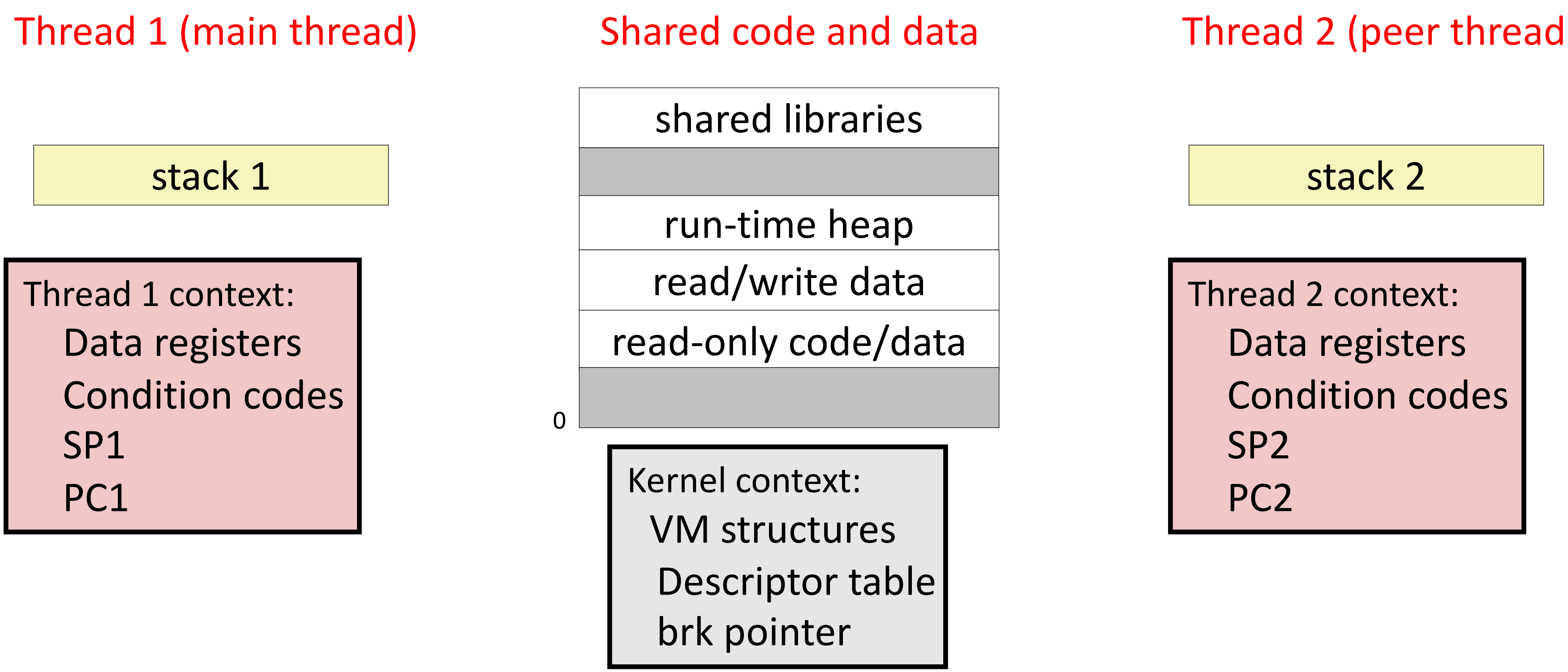
Alternate View of a Process

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



A Process With Multiple Threads

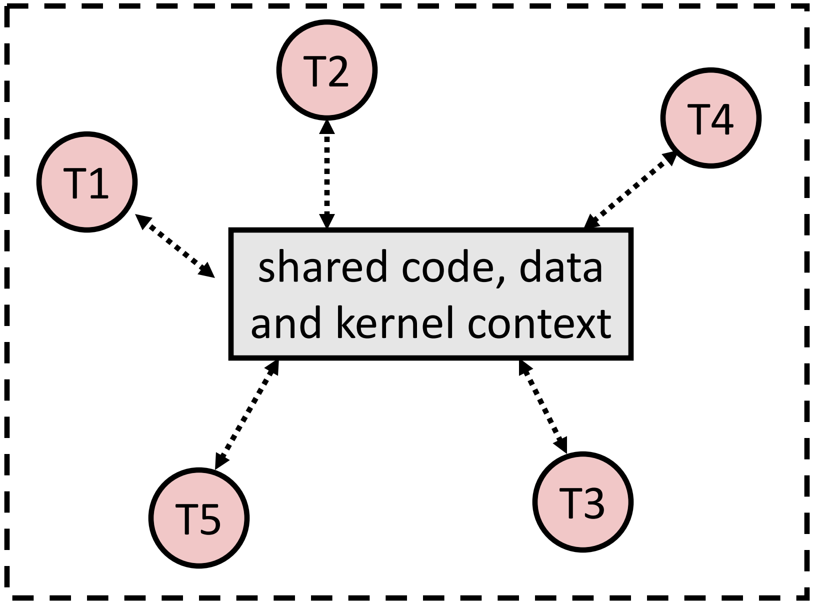
- Multiple threads can be associated with a process
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
    - Share common virtual address space (inc. stacks)
  - Each thread has its own thread id (TID)



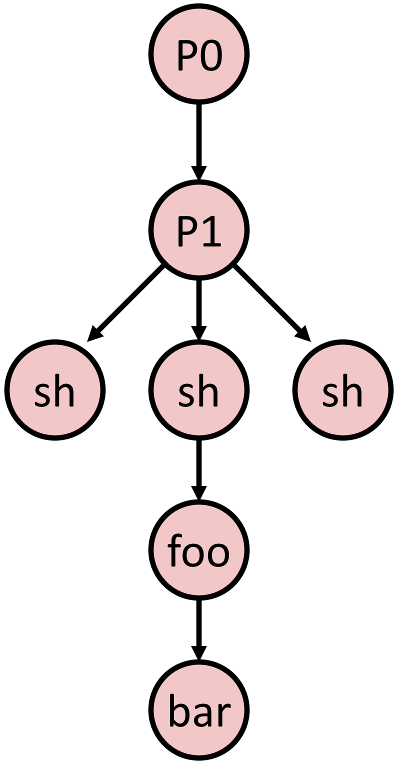
Logical View of Threads

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

Threads associated with process foo

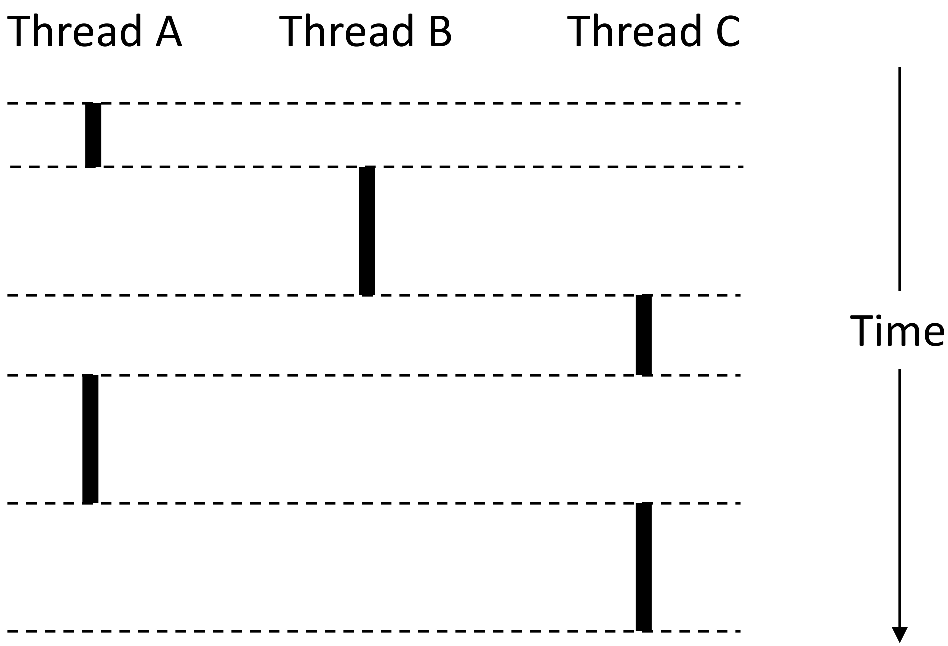


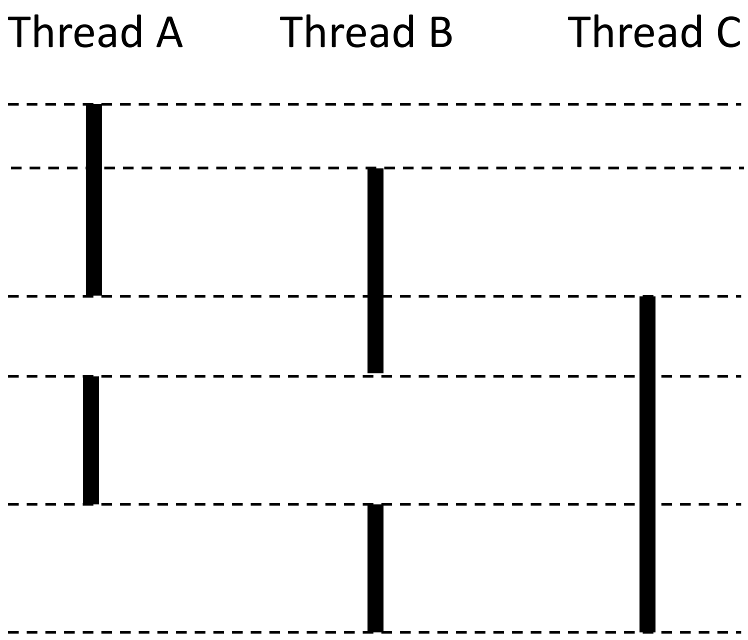
Process hierarchy



Thread Execution

- Single Core Processor
  - Simulate concurrency by time slicing
- Multi-Core Processor
  - Can have true concurrency

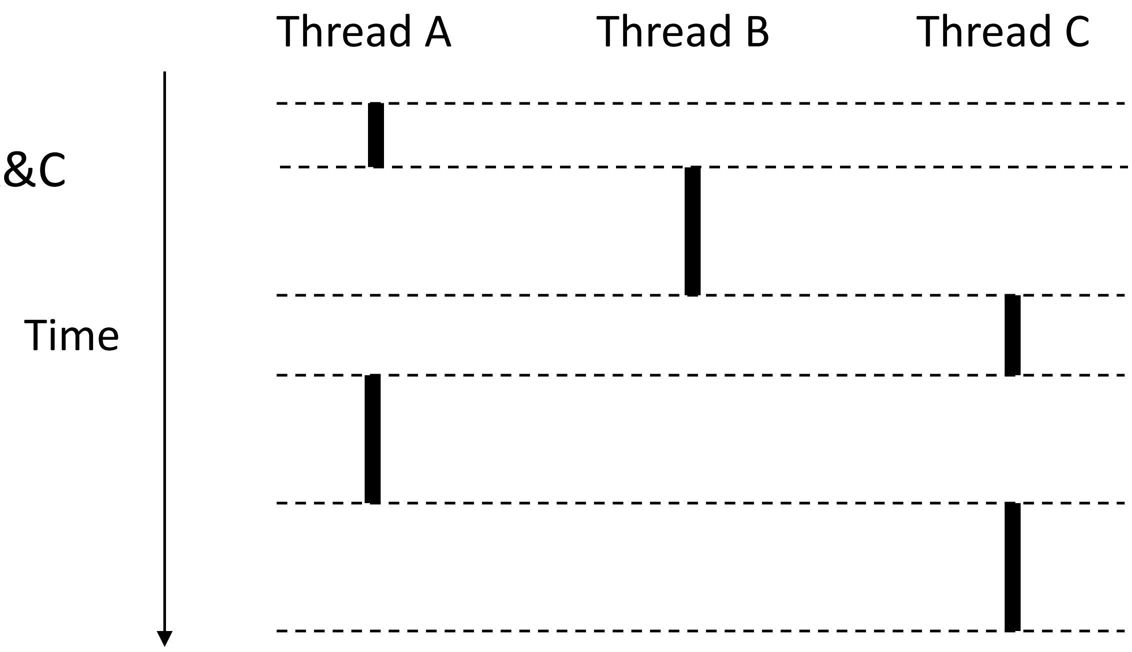




Run 3 threads on 2 cores

Logical Concurrency

- Two threads are (logically) concurrent if their flows overlap in time
- Otherwise, they are sequential
- Examples:
  - Concurrent: A & B, A&C
  - Sequential: B & C



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 code and some data
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) is 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

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] , `RET` [terminates current thread]
  - Synchronizing access to shared variables
    - `pthread_mutex_init`
    - `pthread_mutex_[un]lock`
    - `pthread_cond_init`
    - `pthread_cond_[timed]wait`

The Pthreads "hello, world" Program

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *thread(void *vargp);

int main() {
  pthread_t tid;

  Pthread_create(&tid, NULL, thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}
```

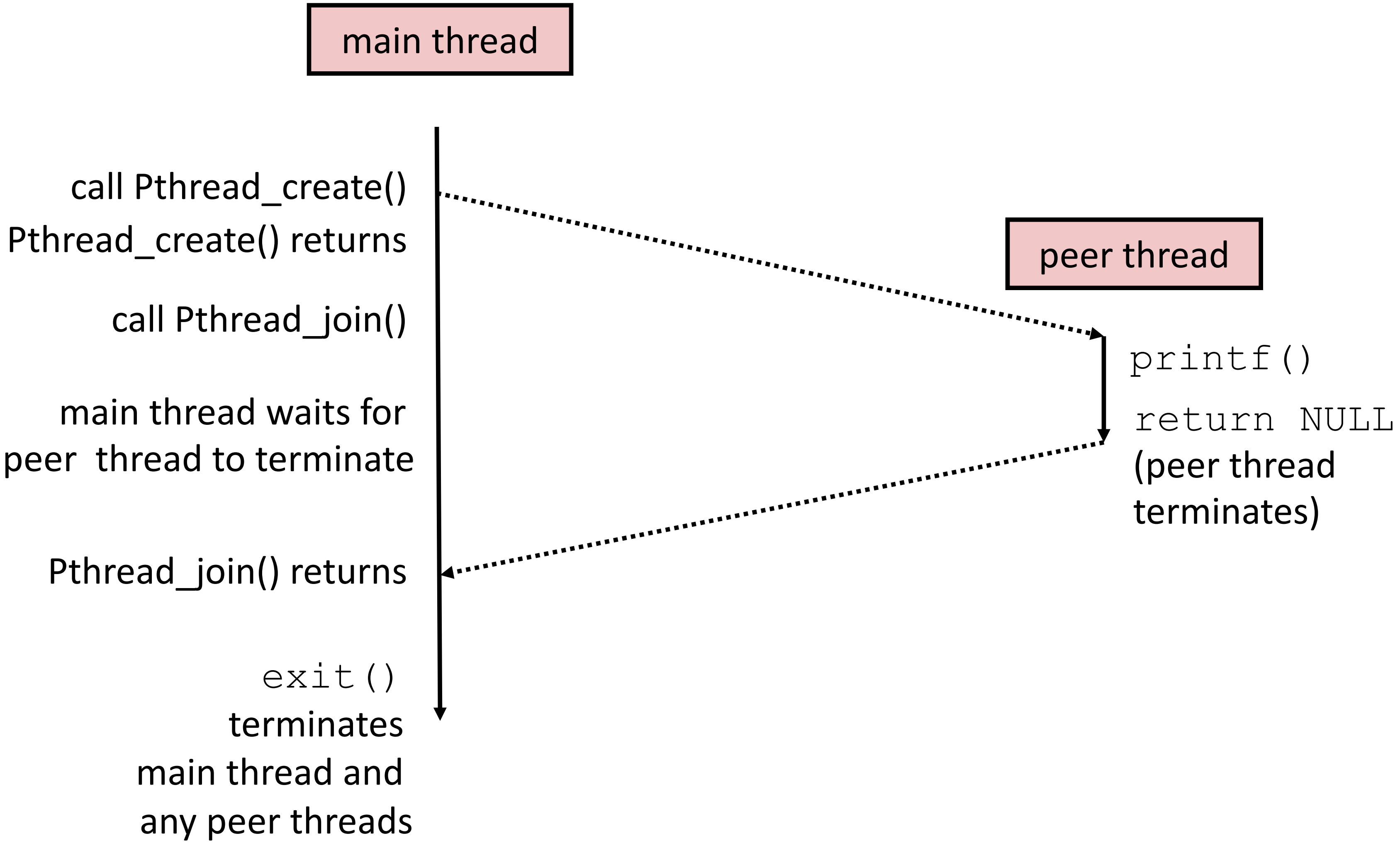
Thread attributes (usually NULL)

*Thread arguments (void *p)*

*return value (void **p)*

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

Execution of Threaded“hello, world”



Thread-Based Concurrent Echo Server

```
int main(int argc, char **argv) {
    int port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen = sizeof(clientaddr);
    pthread_t tid;

    int listenfd = open_listenfd(port);
    while (1) {
        int *connfdp = malloc(sizeof(int));
        *connfdp = accept(listenfd,
                          (SA *) &clientaddr, &clientlen);
        pthread_create(&tid, NULL, echo_thread, connfdp);
    }
}
```

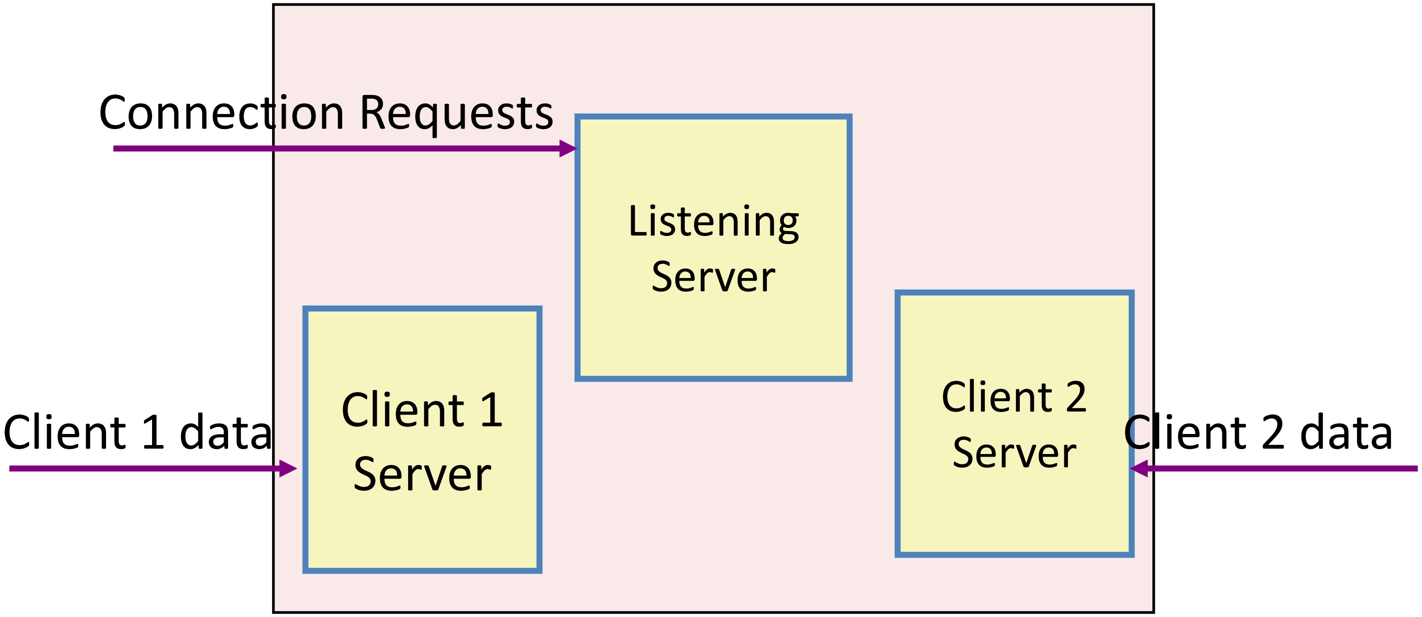
- Spawn new thread for each client
- Pass it copy of connection file descriptor
- Note use of Malloc()!
  - Without corresponding Free()

Thread-Based Concurrent Server (cont)

```
/* thread routine */
void *echo_thread(void *vargp)
{
    int connfd = *((int *)vargp);
    pthread_detach(pthread_self());
    free(vargp);
    echo(connfd);
    close(connfd);
    return NULL;
}
```

- Run thread in “detached” mode
  - Runs independently of other threads
  - Reaped when it terminates
- Free storage allocated to hold clientfd
  - “Producer-Consumer” model

Threaded Server Execution Model



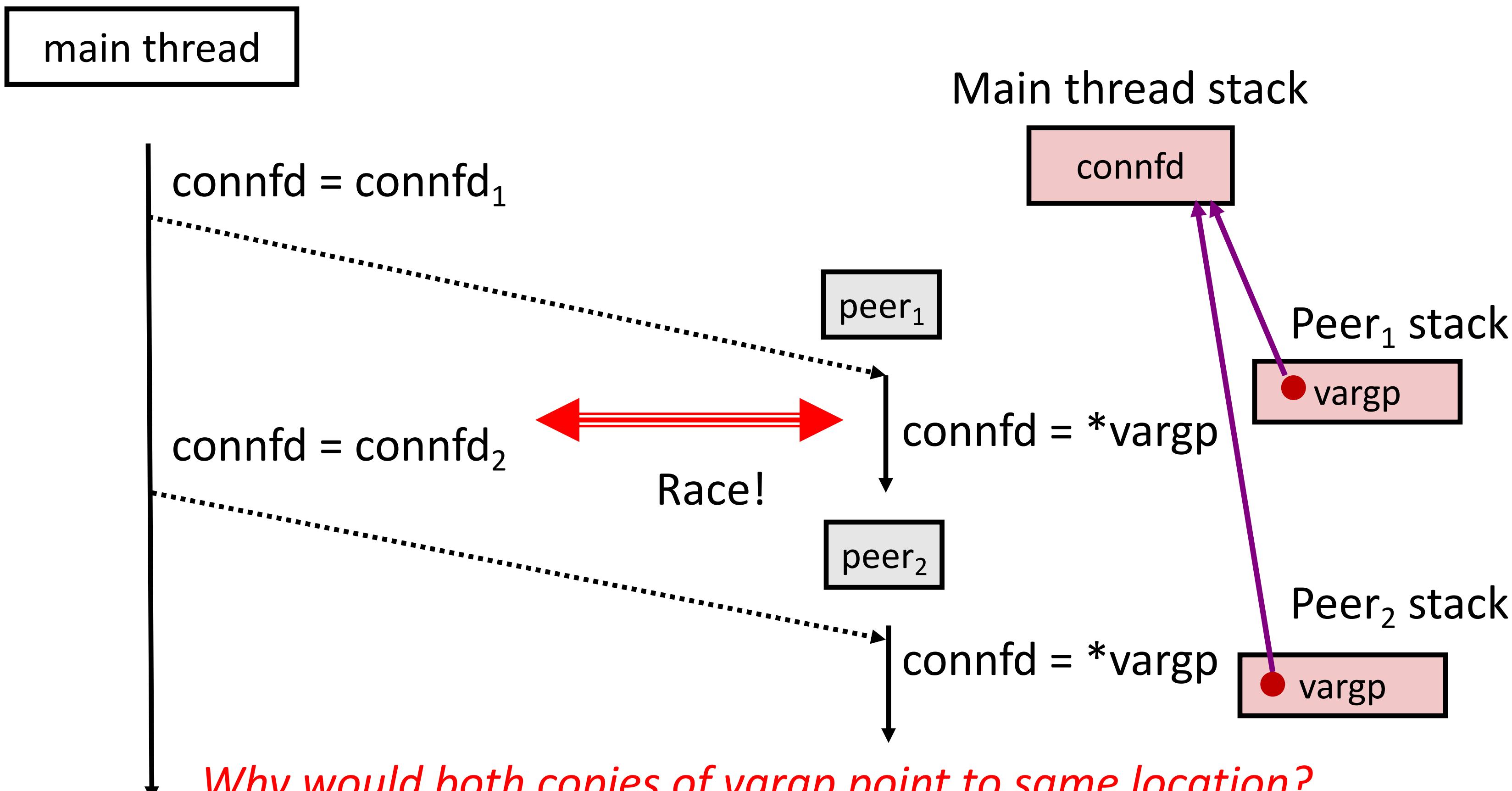
- Each client handled by individual peer thread
- Threads share all process state except TID
- Each thread has a separate stack for local variables

Issues With Thread-Based Servers

- Must run "detached" to avoid memory leak
  - 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, echo_thread, (void *) &connfd);
    }
}
```



Why would both copies of vargp point to same location?

Could this race occur?

Main

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

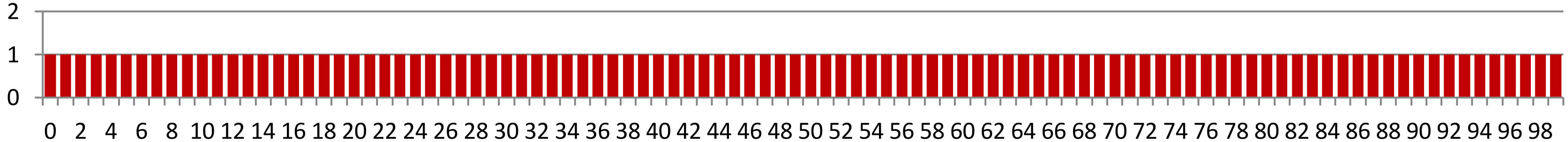
Thread

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

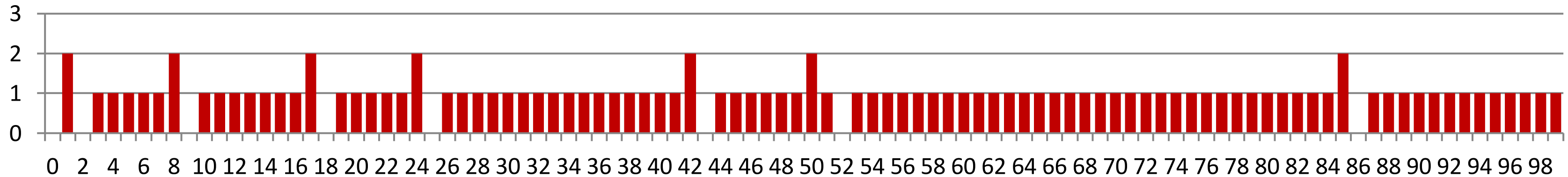
- Race Test
  - If no race, then each thread would get different value of i
  - Set of saved values would consist of one copy each of 0 through 99.

Experimental Results

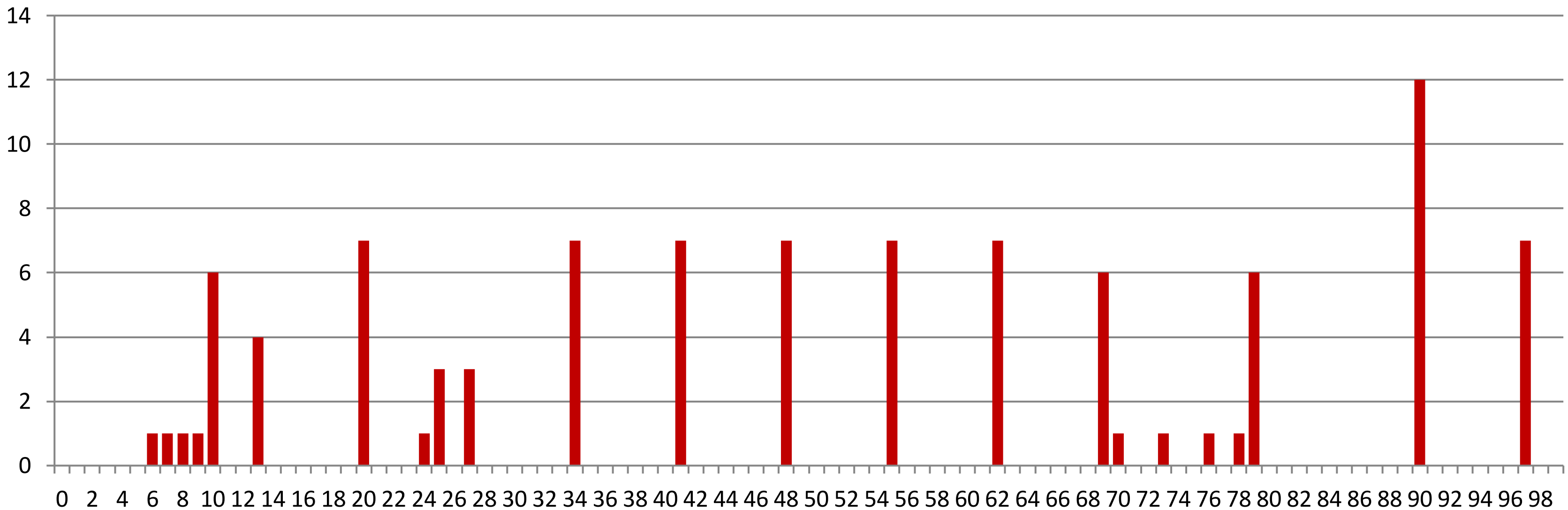
No Race



Single core laptop



Multicore server



- The race can really happen!

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

Approaches to Concurrency

- Processes
  - Hard to share resources: Easy to avoid unintended sharing
  - High overhead in adding/removing clients
- Threads
  - Easy to share resources: Perhaps too easy
  - Medium overhead
  - Not much control over scheduling policies
  - Difficult to debug
    - Event orderings not repeatable
- I/O Multiplexing
  - 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