Concurrent Programming

CSE4100: Multicore Programming

Sungyong Park (PhD)
Data-Intensive Computing and Systems Laboratory (DISCOS)
https://discos.sogang.ac.kr
Office: R908A, E-mail: parksy@sogang.ac.kr

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 error prone and almost impossible**

Why is Concurrent Programming 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-tasks from progressing
    - Example: people always jump in front of you in line
- **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

Echo Server (Main): Revisited

```
#include "csapp.h"
void echo(int connfd);

int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr; /* Enough room for any addr */
    char client_hostname[MAXLINE], client_port[MAXLINE];

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

echoserveri.c

Echo Server (echo Function): Revisited

- **The server uses RIO to read and echo text lines until EOF (end-of-file) condition is encountered**
  - EOF condition caused by client calling `close(clientfd)`

```
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        Rio_writen(connfd, buf, n);
    }
}
```
echo.c

Echo Client (Main): Revisited

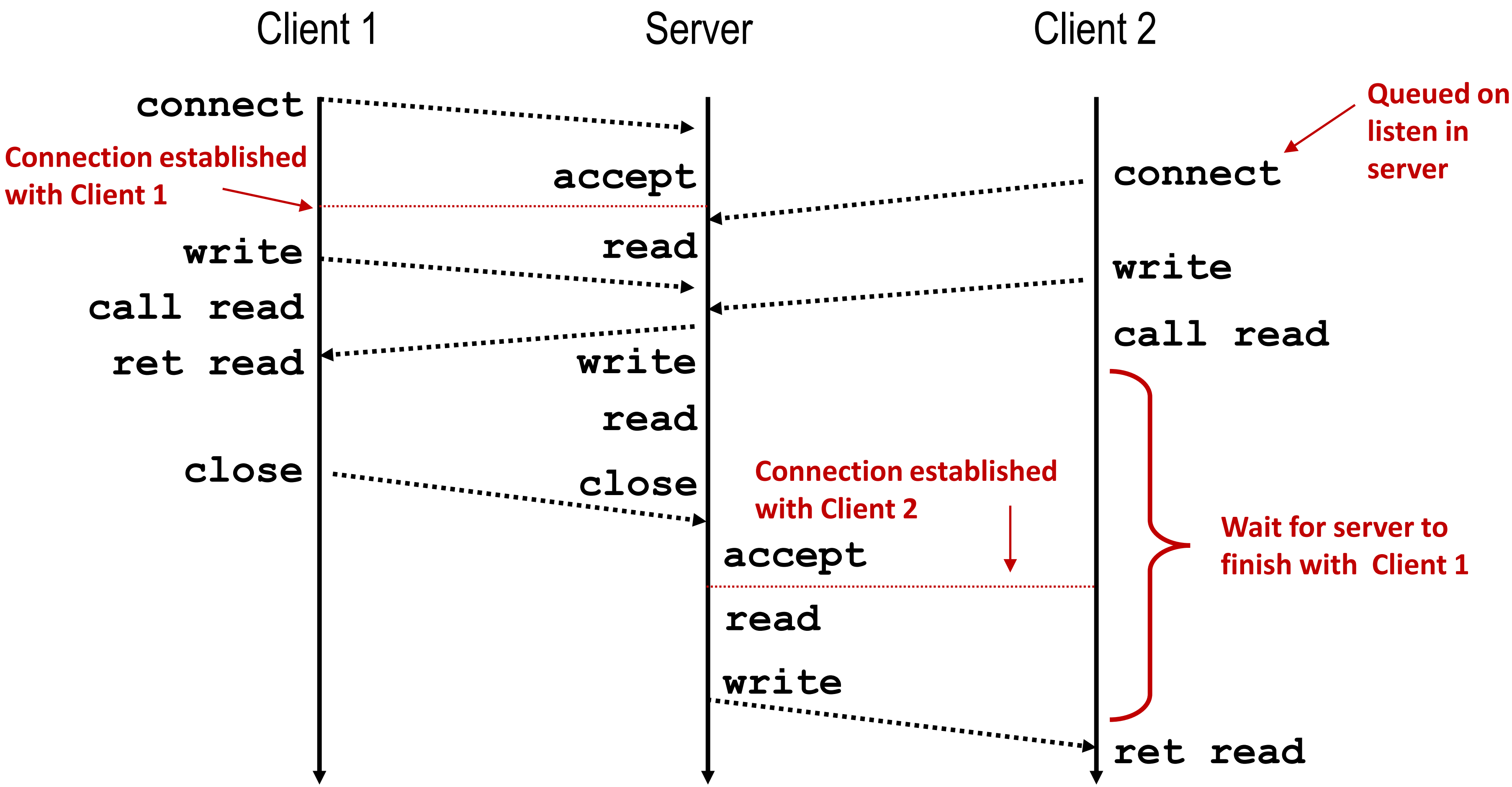
```
#include "csapp.h"

int main(int argc, char **argv)
{
    int clientfd;
    char *host, *port, buf[MAXLINE];
    rio_t rio;

    host = argv[1]; port = argv[2];

    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);

    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
    exit(0);
}
```

Error or EOF (Ctrl+D)

echoclient.c

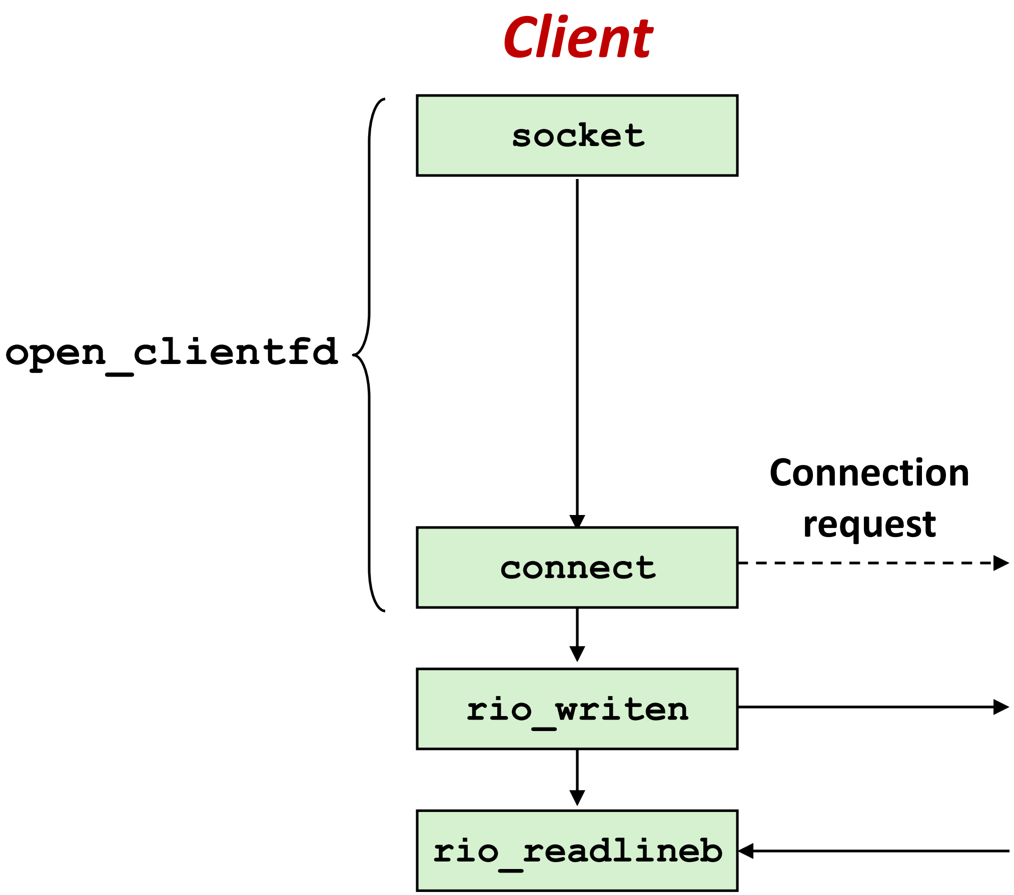
Iterative Servers

- Iterative servers process one request at a time



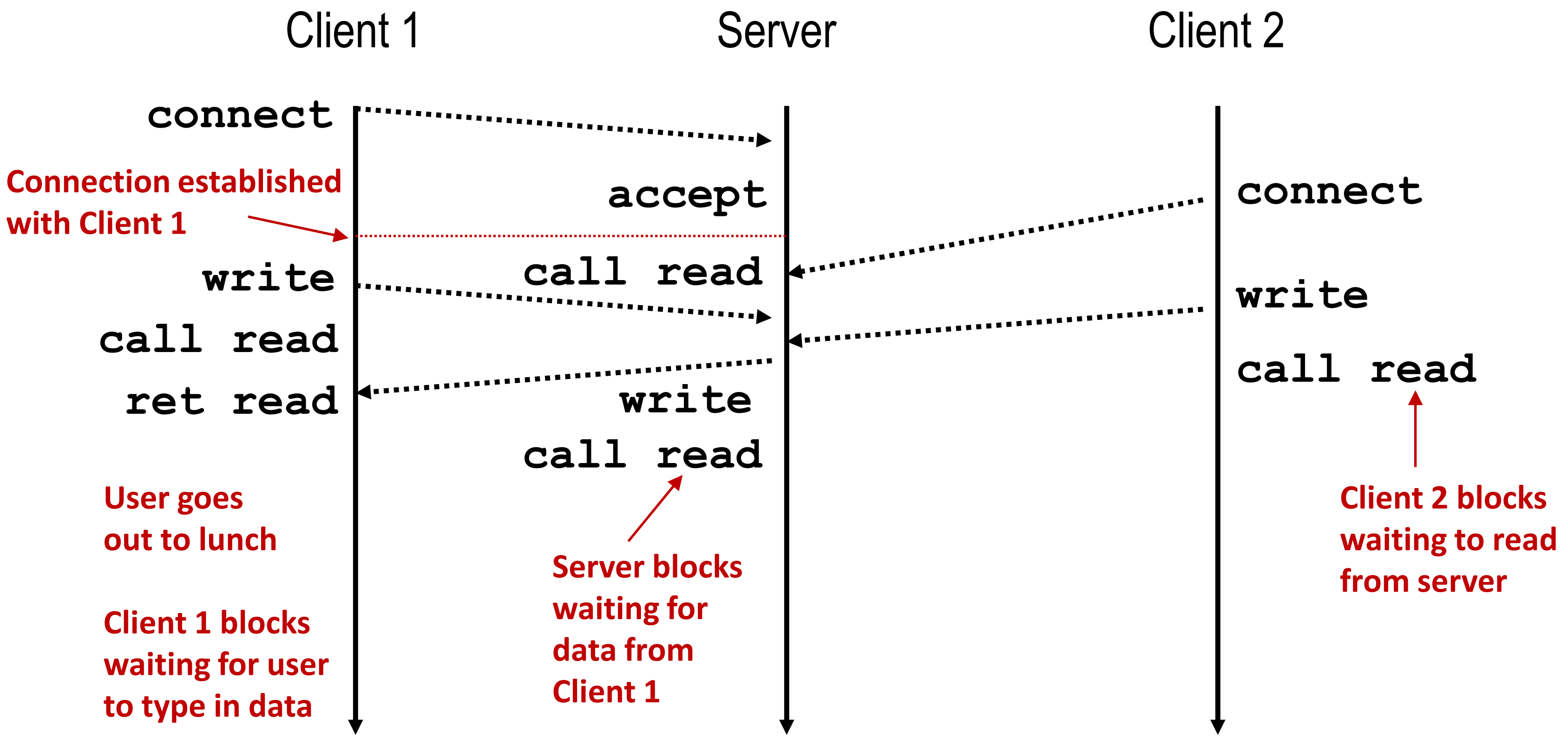
Where Does Second Client Block?

- **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**

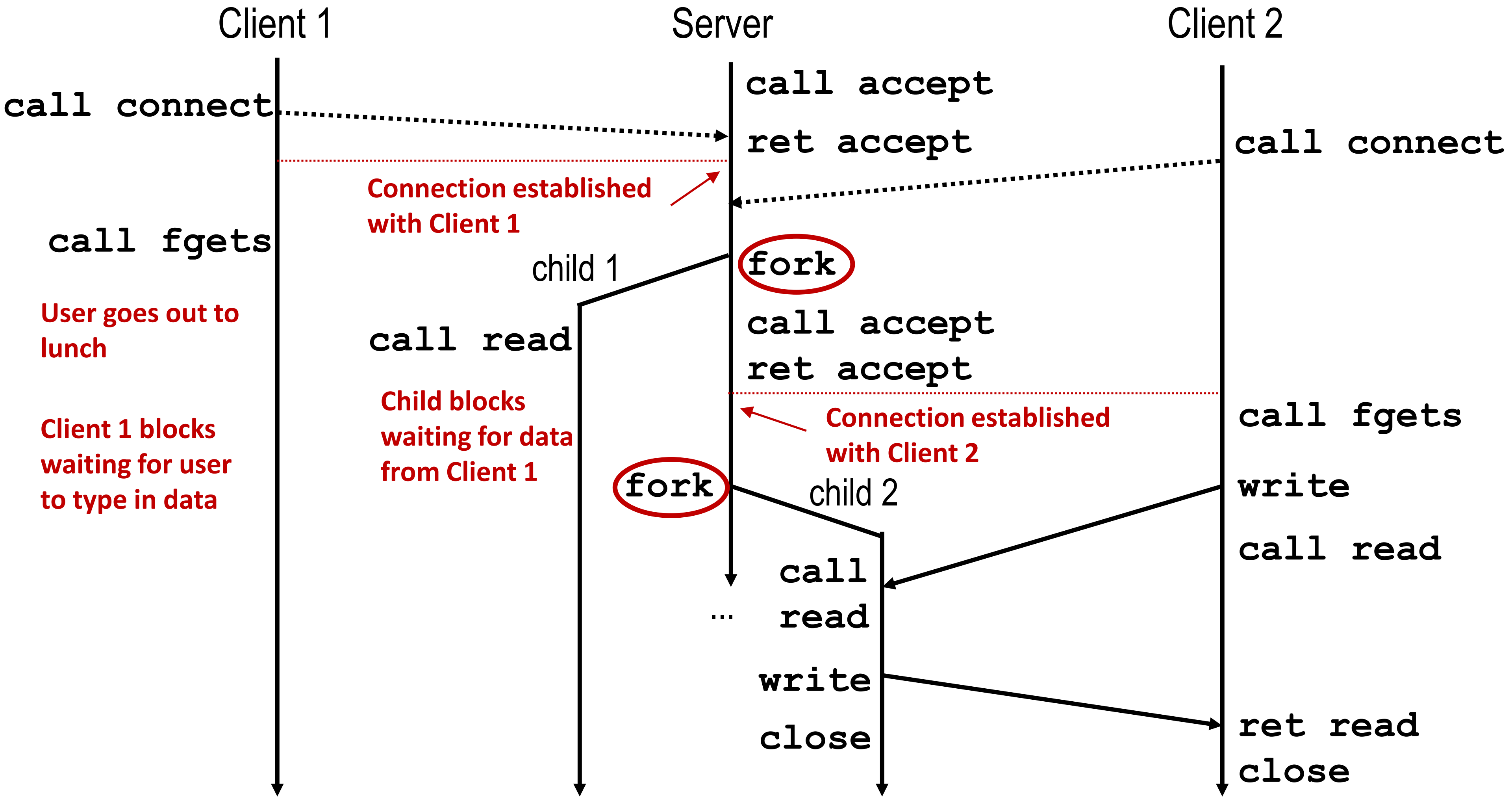
1. Process-based
- Kernel automatically interleaves multiple logical flows
- Each flow has its own private address space

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 of process-based and event-based

Approach #1: Process-based Servers

- Spawn separate process for each client



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)

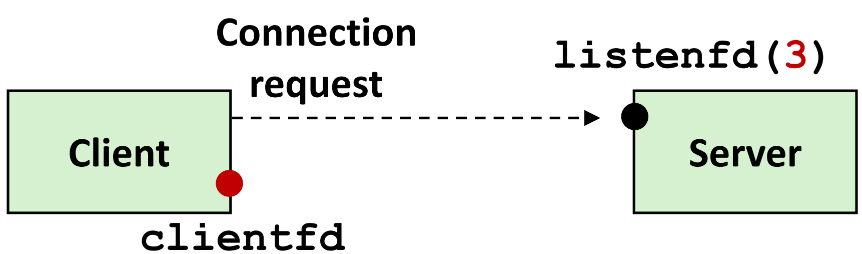
- **Reap all zombie children**

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

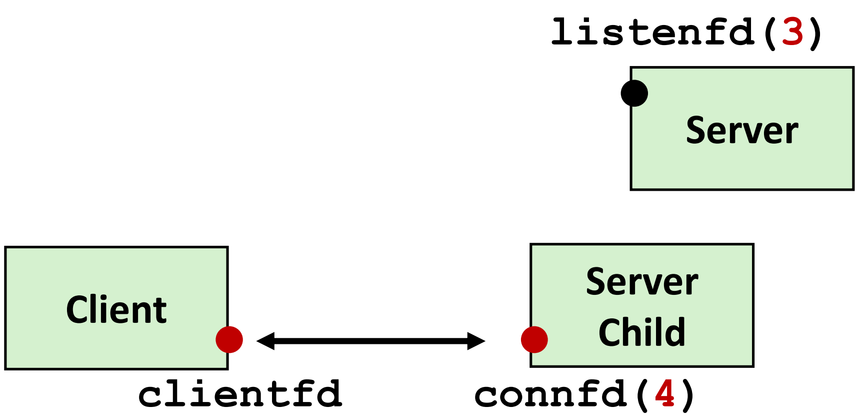
echoserverp.c

Concurrent Server: `accept` Illustrated

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

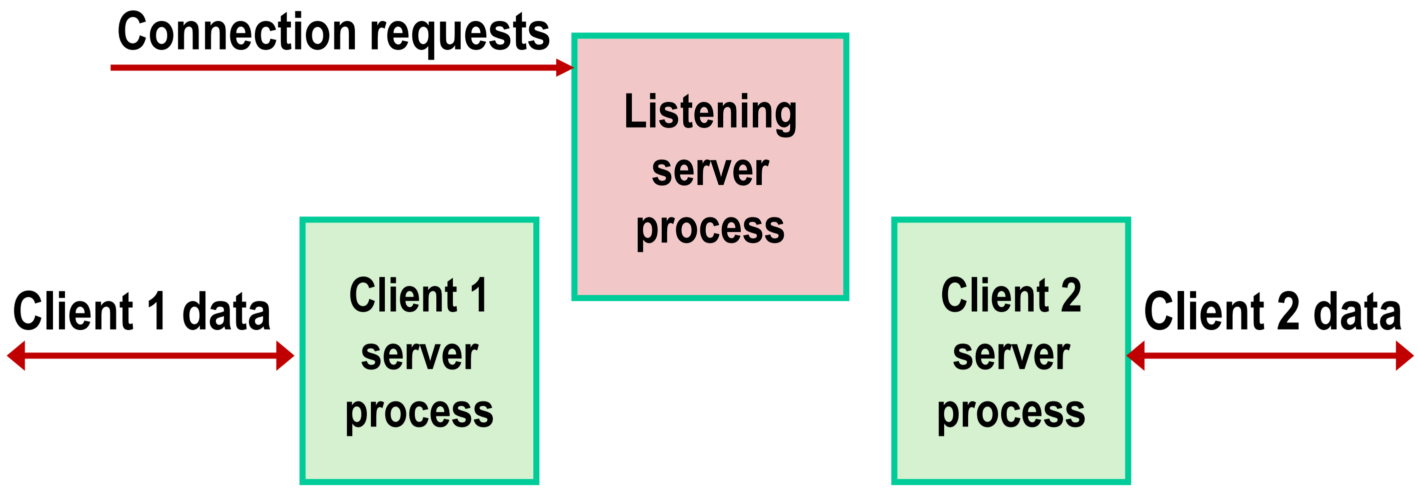


2. Client makes a connection request by calling `connect`



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

Process-based Server Execution Model



- **Each client is 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`**

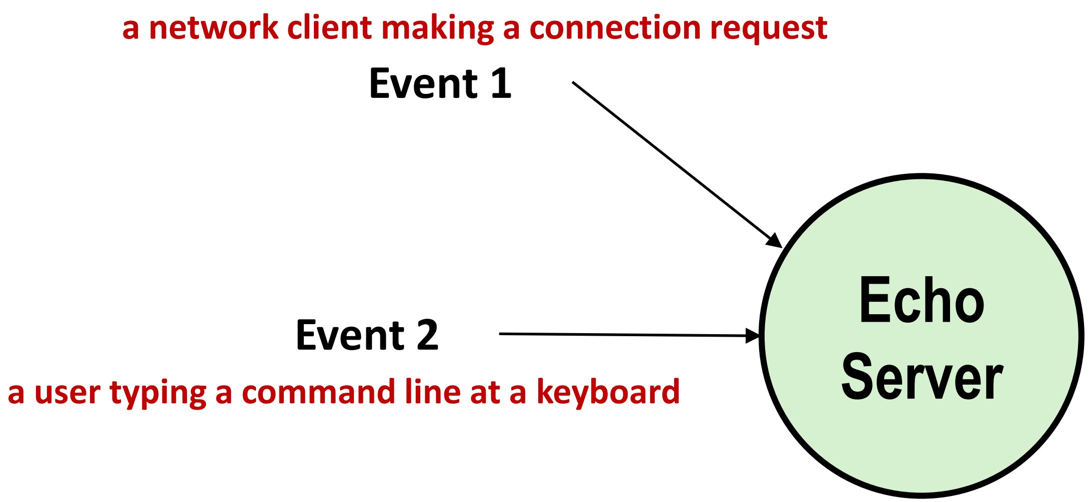
Pros and Cons of Process-based Servers

- **+ 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**
  - Requires IPC (Inter-Process Communication) mechanisms
    - FIFO's (named pipes), System V shared memory and semaphores

Concurrent Programming with I/O Multiplexing

- **Why I/O Multiplexing?**
  - Suppose you are asked to write an echo server that can respond to interactive commands that the user types to standard input in a single process

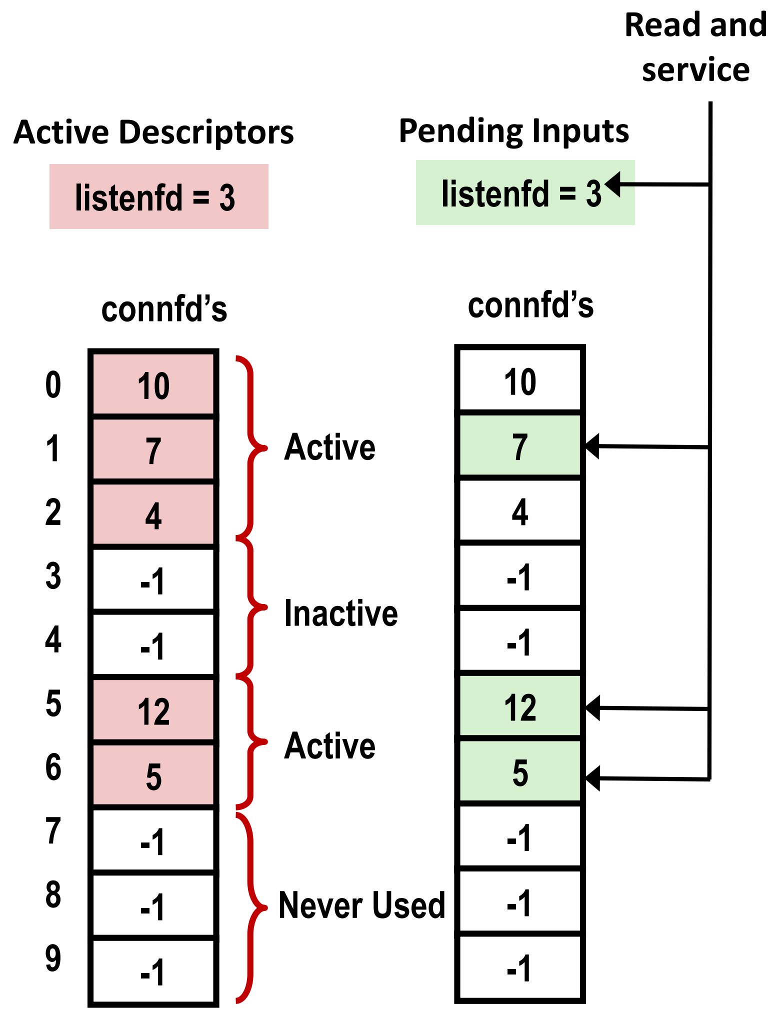


Which event do we wait for first?

Neither option is ideal…

Approach #2: Event-based Servers

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



I/O Multiplexing Using `select`

- **I/O Multiplexing**
  - Use the **`select`** or **`epoll`** functions to ask the kernel to suspend the process, returning control to the application only after one or more I/O events have occurred

max descriptor in descriptor set plus 1

read set

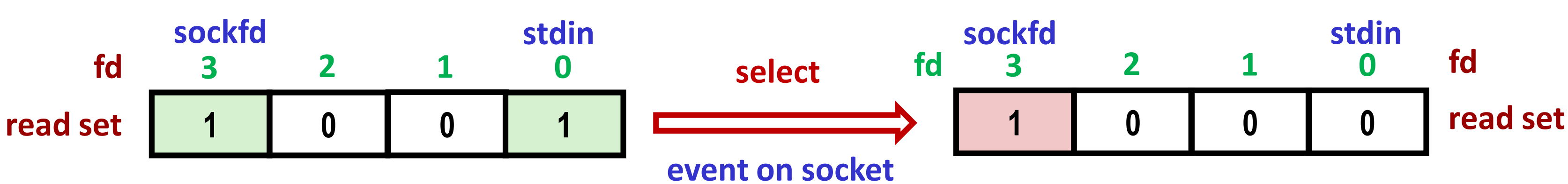
write set (normally NULL)

exception set (normally NULL)

```
#include <sys/select.h>

int select(int n, fd_set *readfd, fd_set *writefd, fd_set *exceptfd,
                                  struct timeval *timeout);

- Returns nonzero count of ready descriptors, -1 on error
```

time-out (normally NULL)

```
void FD_ZERO(fd_set *fdset);                /* Clear all bits in fdset */
void FD_CLR(int fd, fd_set *fdset);         /* Clear bit fd in fdset */
void FD_SET(int fd, fd_set *fdset);         /* Turn on bit fd in fdset */
int  FD_ISSET(int fd, fd_set *fd_set);      /* Is bit fd in fdset on? */
```



code/conc/select.c

```
1  #include "csapp.h"
2  void echo(int connfd);
3  void command(void);
4
5  int main(int argc, char **argv)
6  {
7      int listenfd, connfd;
8      socklen_t clientlen;
9      struct sockaddr_storage clientaddr;
10     fd_set read_set, ready_set;
11
12     if (argc != 2) {
13         fprintf(stderr, "usage: %s <port>\n", argv[0]);
14         exit(0);
15     }
16     listenfd = Open_listenfd(argv[1]);
17
18     FD_ZERO(&read_set);               /* Clear read set */
19     FD_SET(STDIN_FILENO, &read_set); /* Add stdin to read set */
20     FD_SET(listenfd, &read_set);      /* Add listenfd to read set */
21
```

line 18:

	listenfd			stdin
	3	2	1	0
read_set (∅):	0	0	0	0

line 19-20:

	listenfd			stdin
	3	2	1	0
read_set ({0,3}):	1	0	0	1

Due to a side effect from `select` which modifies the `fd_set` pointed to by the argument `fdset` to indicate a subset of the read set called the *ready set*

line 24:

	listenfd			stdin
	3	2	1	0
ready_set ({0}):	0	0	0	1

```
22      while (1) {
23          ready_set = read_set;
24          Select(listenfd+1, &ready_set, NULL, NULL, NULL);
25          if (FD_ISSET(STDIN_FILENO, &ready_set))
26              command(); /* Read command line from stdin */
27          if (FD_ISSET(listenfd, &ready_set)) {
28              clientlen = sizeof(struct sockaddr_storage);
29              connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
30              echo(connfd); /* Echo client input until EOF */
31              Close(connfd);
32          }
33      }
34  }
35
36  void command(void) {
37      char buf[MAXLINE];
38      if (!Fgets(buf, MAXLINE, stdin))
39          exit(0); /* EOF */
40      printf("%s", buf); /* Process the input command */
41  }
```

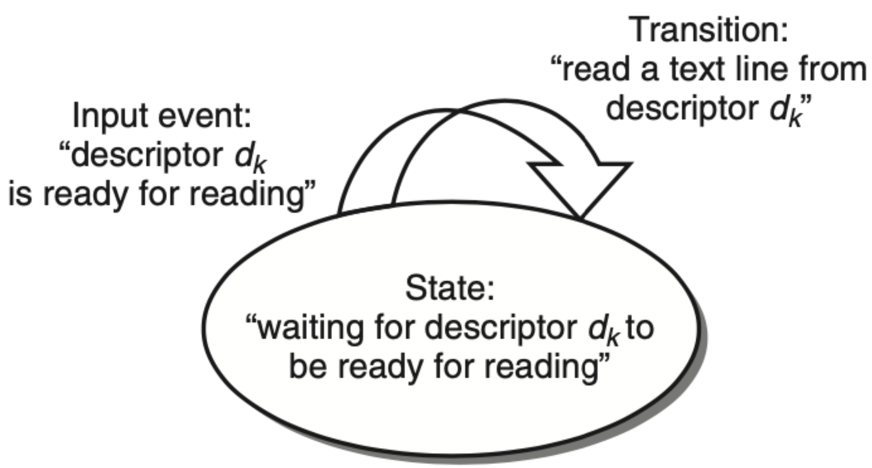
code/conc/select.c

Issues with `select.c`

- **Blocking problems**
  - Once the server connects to a client, it continues echoing input lines *until* the client closes its end of the connection
    - Thus, if a user types a command to standard input, he/she will not get a response until the server is finished with the client
  - We need a way to multiplex at a finer granularity, echoing (at most) one text line each time through the server loop!

I/O Multiplexed Event Processing

- **I/O multiplexing and event-driven programs**
  - I/O multiplexing can be used as the basis for concurrent event-driven programs, where flows make progress as a result of certain events
- **Modeling logical flows as *`state machines`***
  - State machines is a collection of ***`states`***, ***`input events`***, and ***`transitions`*** that map states and input events to states
- **State machine for a logical flow in a concurrent event-driven echo server**



code/conc/echoservers.c

```
#include "csapp.h"

typedef struct { /* Represents a pool of connected descriptors */
    int maxfd;        /* Largest descriptor in read_set */
    fd_set read_set;  /* Set of all active descriptors */
    fd_set ready_set; /* Subset of descriptors ready for reading  */
    int nready;       /* Number of ready descriptors from select */
    int maxi;         /* High water index into client array */
    int clientfd[FD_SETSIZE];    /* Set of active descriptors */
    rio_t clientrio[FD_SETSIZE]; /* Set of active read buffers */
} pool;

int byte_cnt = 0; /* Counts total bytes received by server */

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

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = Open_listenfd(argv[1]);
    init_pool(listenfd, &pool);
```

FD_SETSIZE = 1024

Keep track of the largest index into the `clientfd` array so that we do not need to search the whole array

# of ready descriptors

```
    while (1) {
        /* Wait for listening/connected descriptor(s) to become ready */
        pool.ready_set = pool.read_set;
        pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);

        /* If listening descriptor ready, add new client to pool */
        if (FD_ISSET(listenfd, &pool.ready_set)) {
            clientlen = sizeof(struct sockaddr_storage);
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            add_client(connfd, &pool);
        }

        /* Echo a text line from each ready connected descriptor */
        check_clients(&pool);
    }
}
```

code/conc/echoservers.c

code/conc/echoservers.c

```
void init_pool(int listenfd, pool *p)
{
    /* Initially, there are no connected descriptors */
    int i;
    p->maxi = -1;
    for (i=0; i< FD_SETSIZE; i++)
        p->clientfd[i] = -1;

    /* Initially, listenfd is only member of select read set */
    p->maxfd = listenfd;
    FD_ZERO(&p->read_set);
    FD_SET(listenfd, &p->read_set);
}
```

code/conc/echoservers.c

code/conc/echoservers.c

```
void add_client(int connfd, pool *p)
{
    int i;
    p->nready--;
    for (i = 0; i < FD_SETSIZE; i++) /* Find an available slot */
        if (p->clientfd[i] < 0) {
            /* Add connected descriptor to the pool */
            p->clientfd[i] = connfd;
            Rio_readinitb(&p->clientrio[i], connfd);

            /* Add the descriptor to descriptor set */
            FD_SET(connfd, &p->read_set);

            /* Update max descriptor and pool high water mark */
            if (connfd > p->maxfd)
                p->maxfd = connfd;
            if (i > p->maxi)
                p->maxi = i;
            break;
        }
    if (i == FD_SETSIZE) /* Couldn't find an empty slot */
        app_error("add_client error: Too many clients");
}
```

If empty

code/conc/echoservers.c

code/conc/echoservers.c

```
void check_clients(pool *p)
{
    int i, connfd, n;
    char buf[MAXLINE];
    rio_t rio;

    for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
        connfd = p->clientfd[i];
        rio = p->clientrio[i];

        /* If the descriptor is ready, echo a text line from it */
        if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
            p->nready--;
            if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
                byte_cnt += n;
                printf("Server received %d (%d total) bytes on fd %d\n",
                       n, byte_cnt, connfd);
                Rio_writen(connfd, buf, n);
            }

            /* EOF detected, remove descriptor from pool */
            else {
                Close(connfd);
                FD_CLR(connfd, &p->read_set);
                p->clientfd[i] = -1;
            }
        }
    }
}
```

Search only up to this point

Skip inactive connections

code/conc/echoservers.c

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- 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

Approach #3: Thread-based Servers

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

Traditional View of a Process

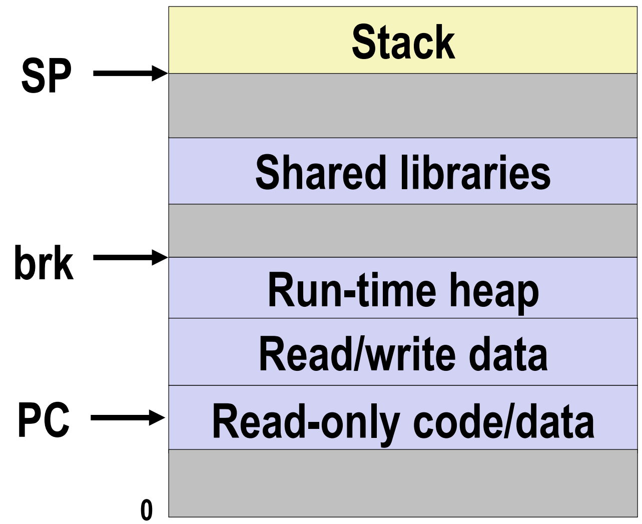
- **Process = process context + code, data, and stack**

Process context

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

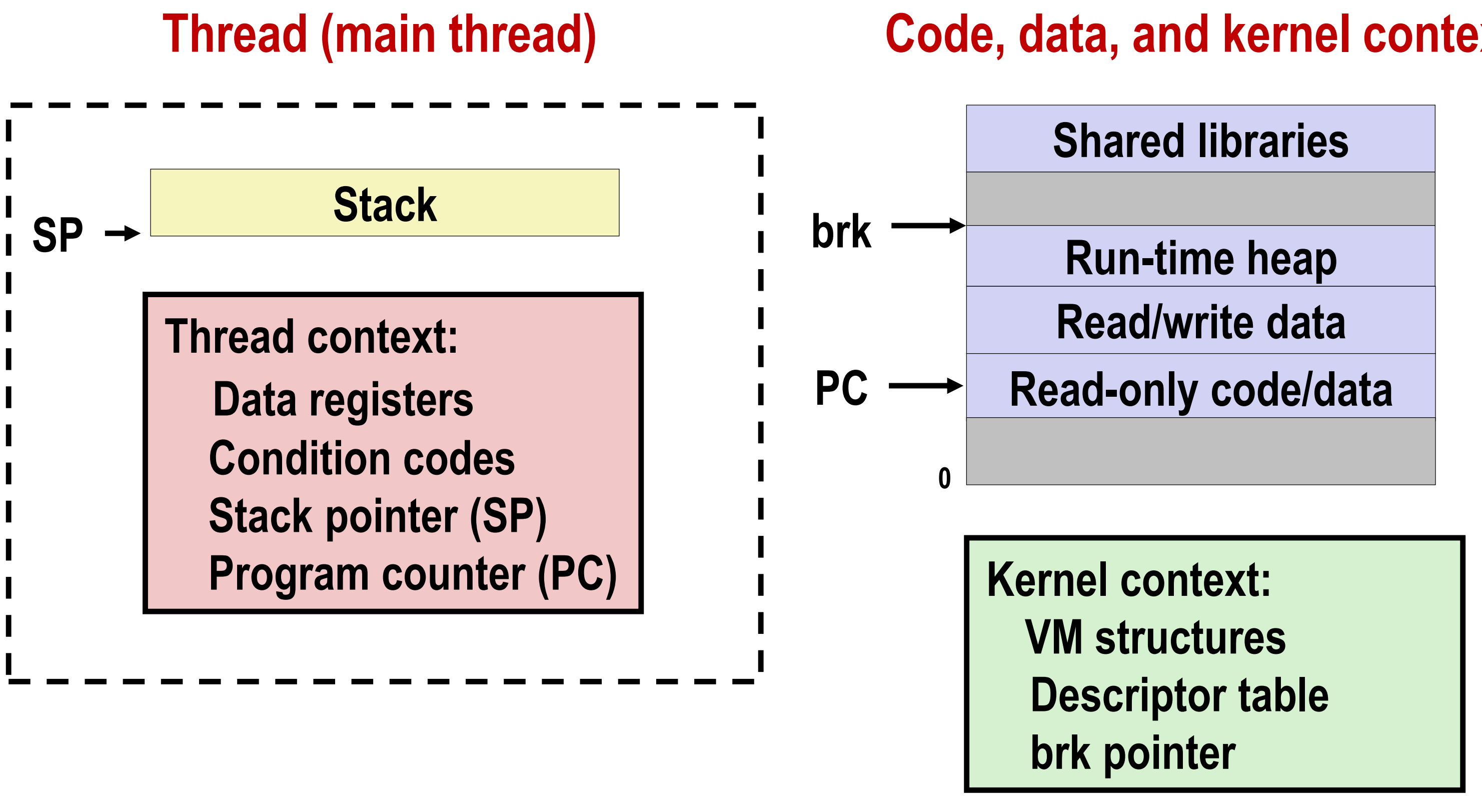
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Code, data, and stack



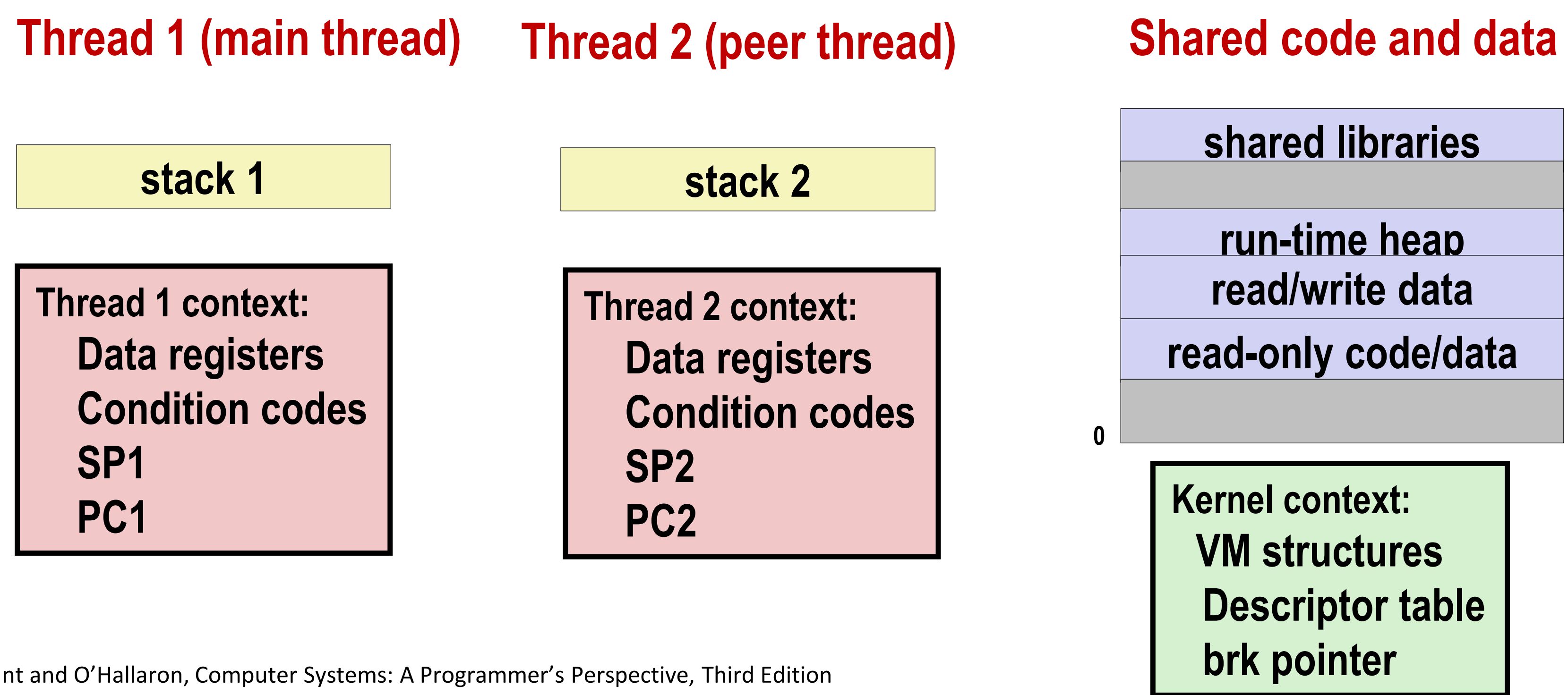
Alternate View of a Process

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



A Process With Multiple Threads

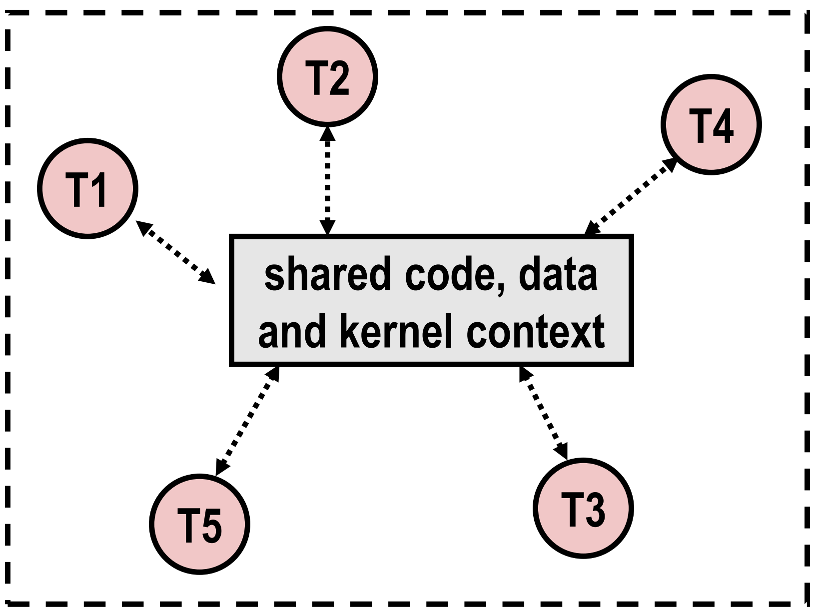
- **Multiple threads can be associated with a process**
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own stack for local variables
    - but not protected from other threads
  - Each thread has its own thread id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



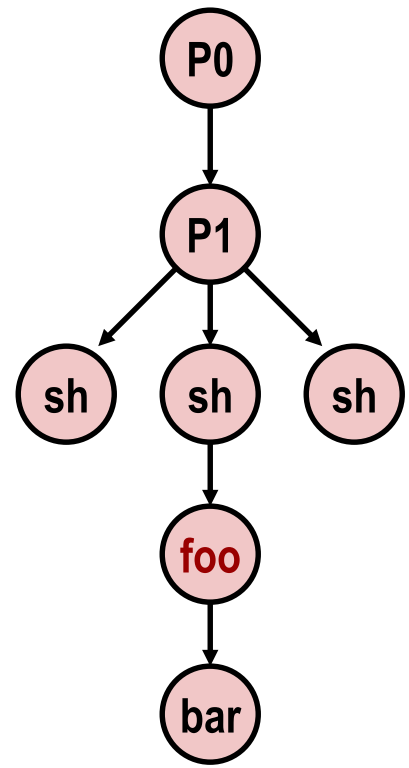
Logical View of Threads

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

Threads associated with a process foo

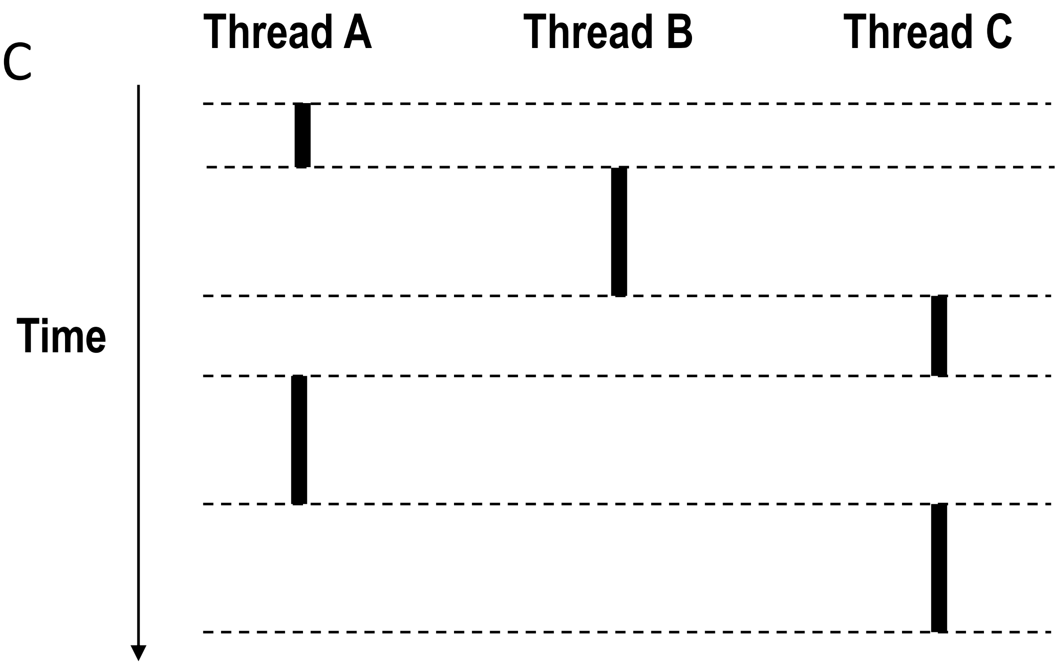


Process hierarchy



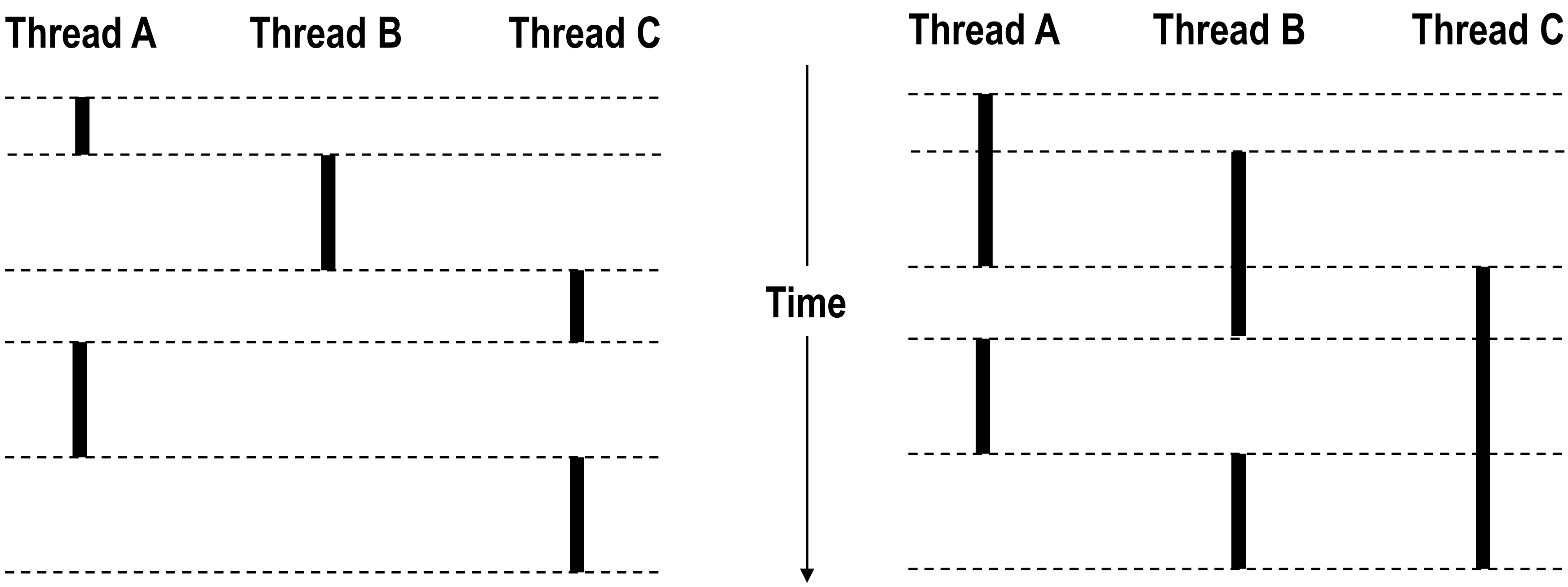
Concurrent Threads

- 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



Run 3 threads on 2 cores

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

Posix Threads (Pthreads) Interface

- *Pthreads:* **Standard interface for ~60 functions that manipulate threads from C programs**

Thread API	Process API analogue
Creating and reaping…	
`pthread_create`	`fork`
`pthread_join`	`waitpid`
Determining your ID…	
`pthread_self`	`getpid`
Terminating execution…	
`pthread_exit`	`exit`
`return` from thread proc	`return` from main
Synchronizing operations…	
`pthread_mutex_lock`	[no exact analogue]
`pthread_mutex_unlock`	

The Pthreads "hello, world" Program

```c
/*
 * 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);
}
```

hello.c

Thread ID

Thread attributes (usually NULL)

Thread routine

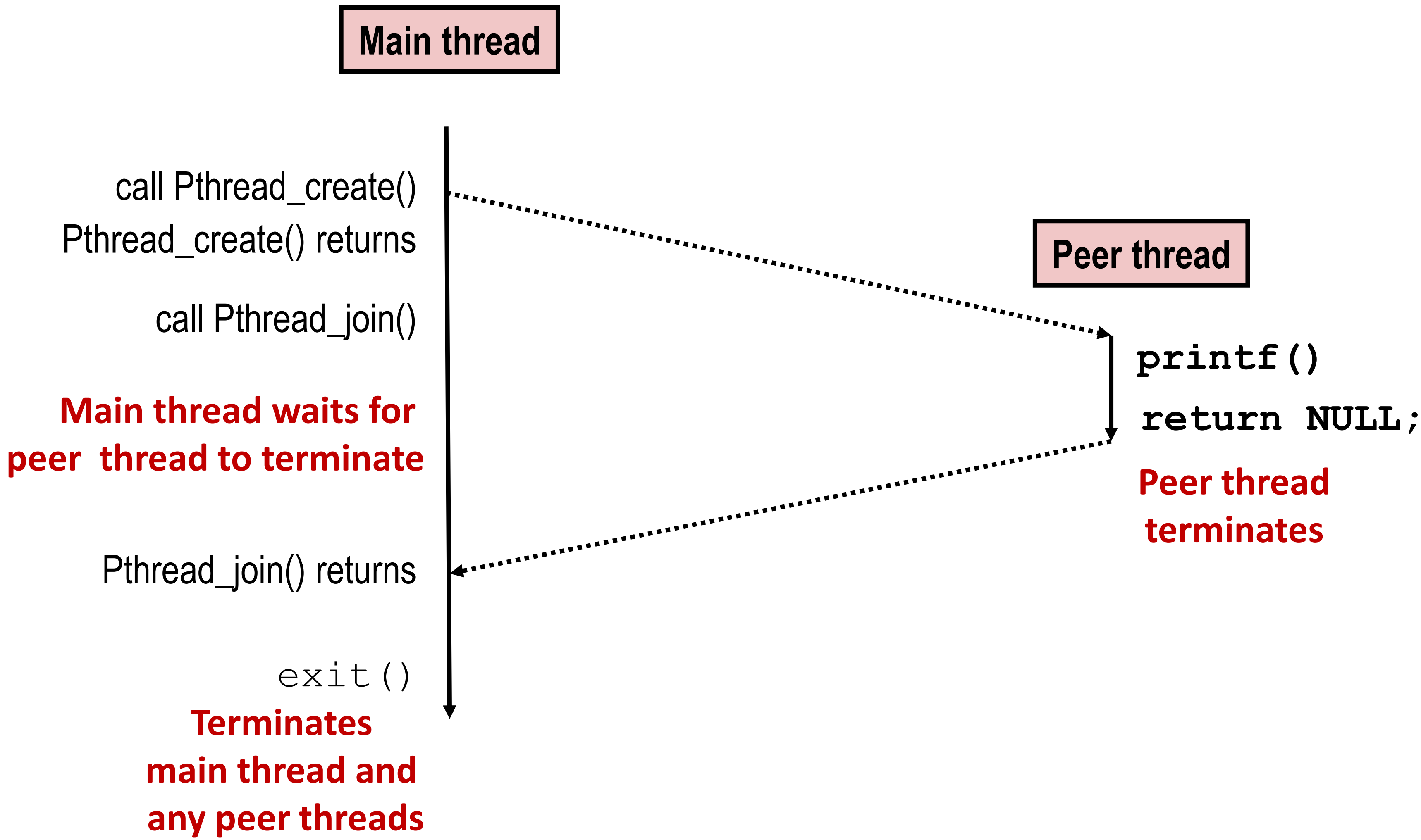
*Thread arguments (void *p)*

*Return value (void **p)*

```c
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);
    }
}
```

echoservert.c

- `malloc` of connected descriptor necessary to avoid deadly race (later)

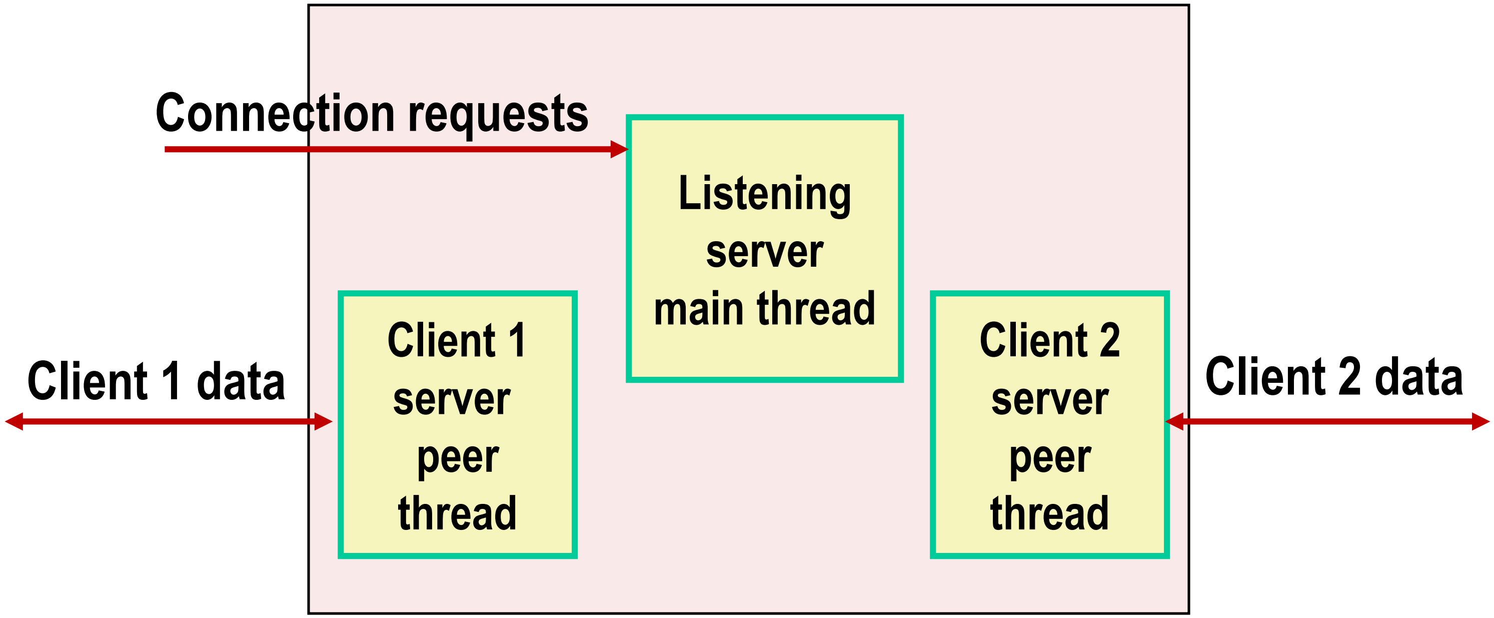
Thread-Based Concurrent Server (Cont)

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

echoservert.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 is 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

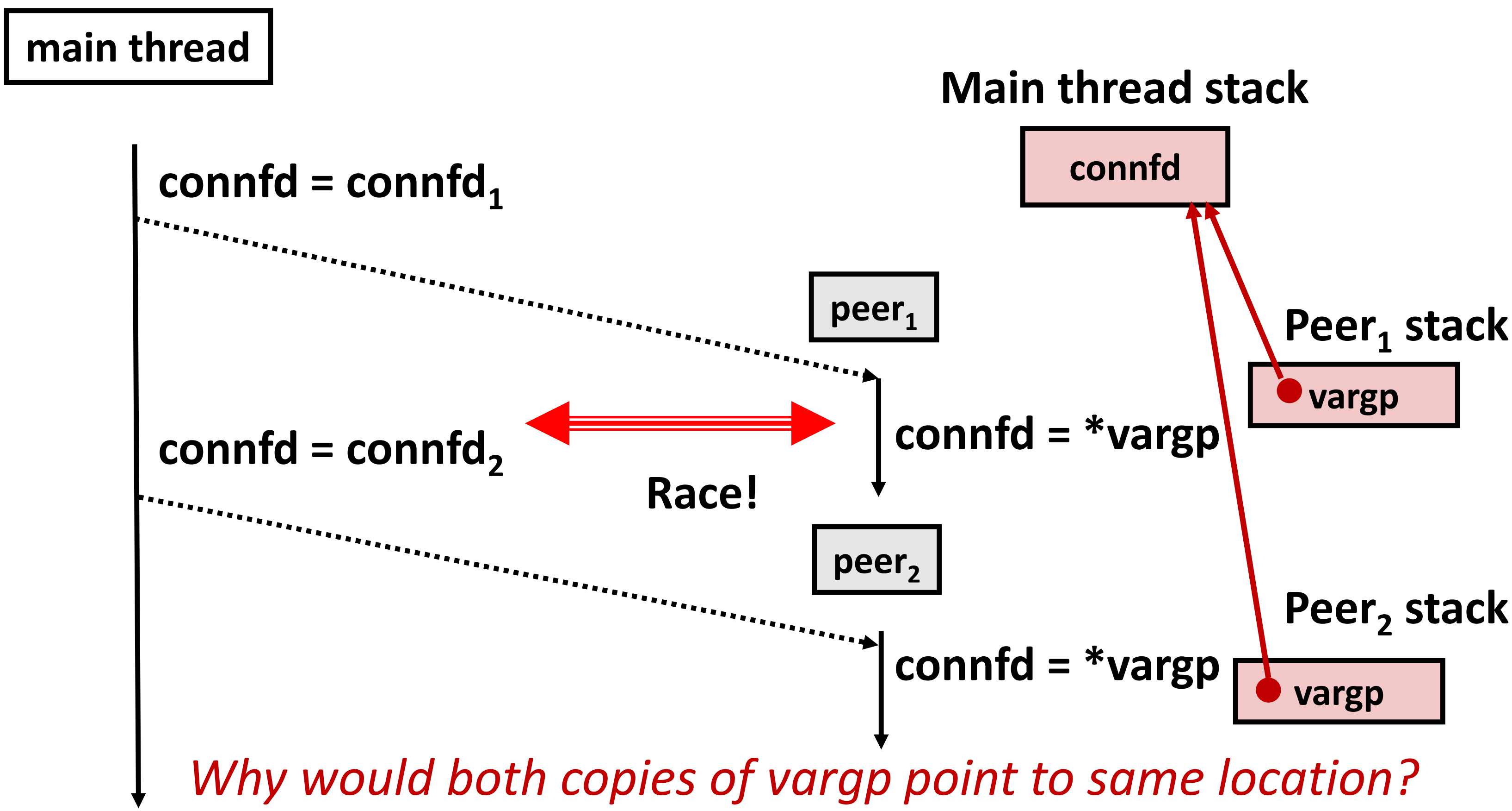
- **Must run "detached" to avoid memory leak**
  - At any point in time, a thread is either *joinable* or *detached*
  - *Joinable* thread can be reaped and killed by other threads
    - must be reaped (with **`pthread_join`**) to free memory resources
  - *Detached* thread cannot be reaped or killed 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***

Potential Form of Unintended Sharing

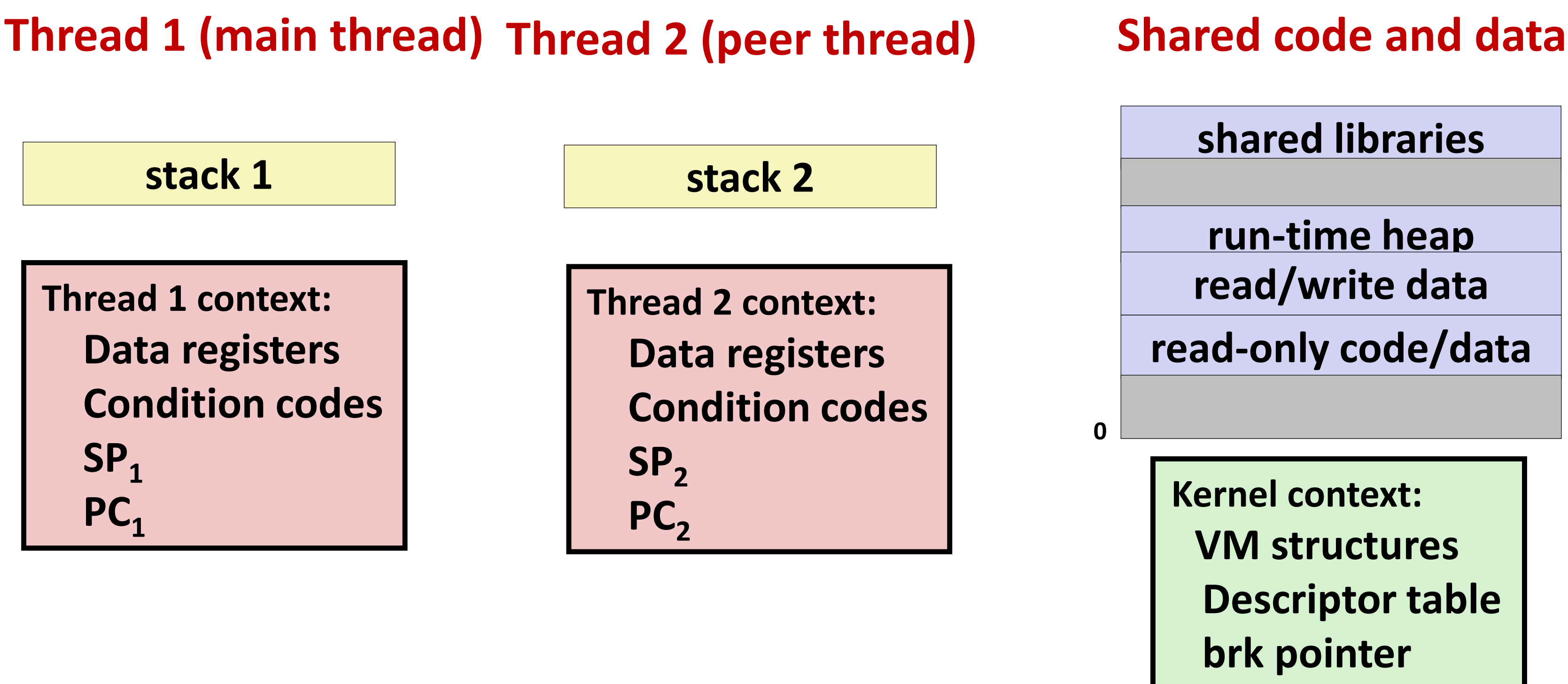
```
while (1) {
  int connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
  Pthread_create(&tid, NULL, thread, &connfd);
}
```



Why would both copies of vargp point to same location?

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)



But ALL Memory is Shared

Thread 1 context:
- Data registers
- Condition codes
- SP_1
- PC_1

Thread 2 context:
- Data registers
- Condition codes
- SP_2
- PC_2

Thread 1 (main thread)

stack 1

Thread 2 (peer thread)

stack 2

shared libraries
run-time heap
read/write data
read-only code/data

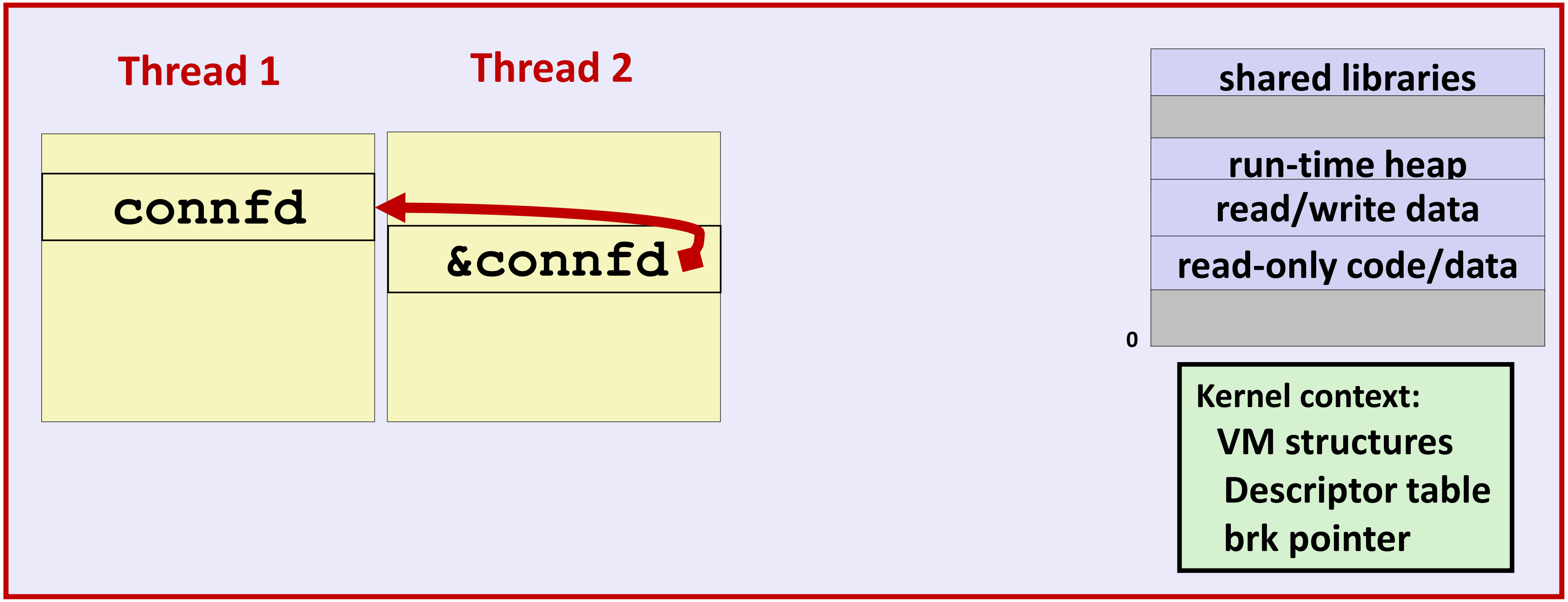
0

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

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

Thread 1 context:
- Data registers
- Condition codes
- SP_1
- PC_1

Thread 2 context:
- Data registers
- Condition codes
- SP_2
- PC_2

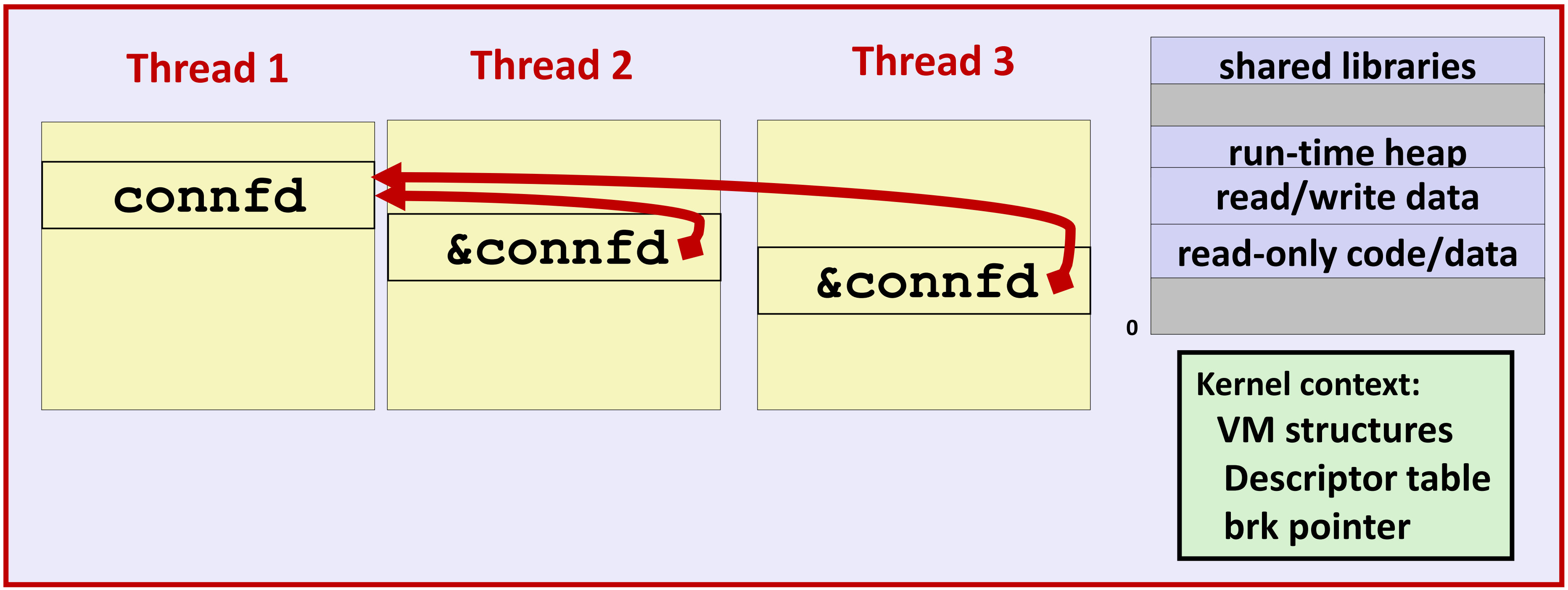


```
while (1) {
    int connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
    Pthread_create(&tid, NULL, thread, &connfd);
}
```

Thread 1 context:
- Data registers
- Condition codes
- SP_1
- PC_1

Thread 2 context:
- Data registers
- Condition codes
- SP_2
- PC_2

Thread 3 context:
- Data registers
- Condition codes
- SP_2
- PC_2



Could This Race Occur?

Main

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

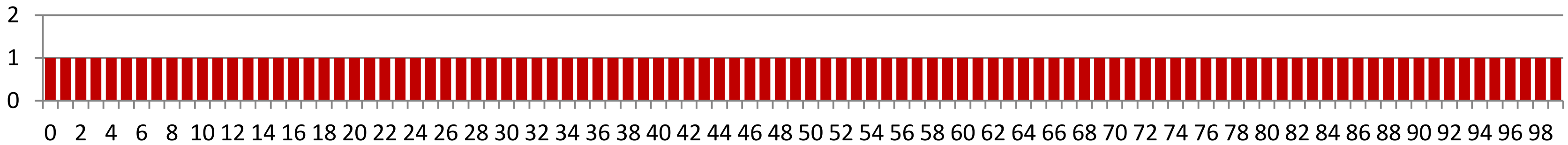
Thread

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

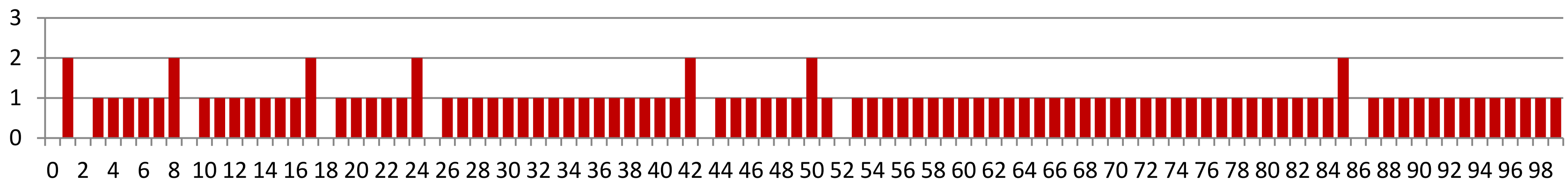
- **Race Test**
  - If no race, then each thread would get different value of `i`
  - Set of saved values would consist of one copy each of 0 through 99

Experimental Results

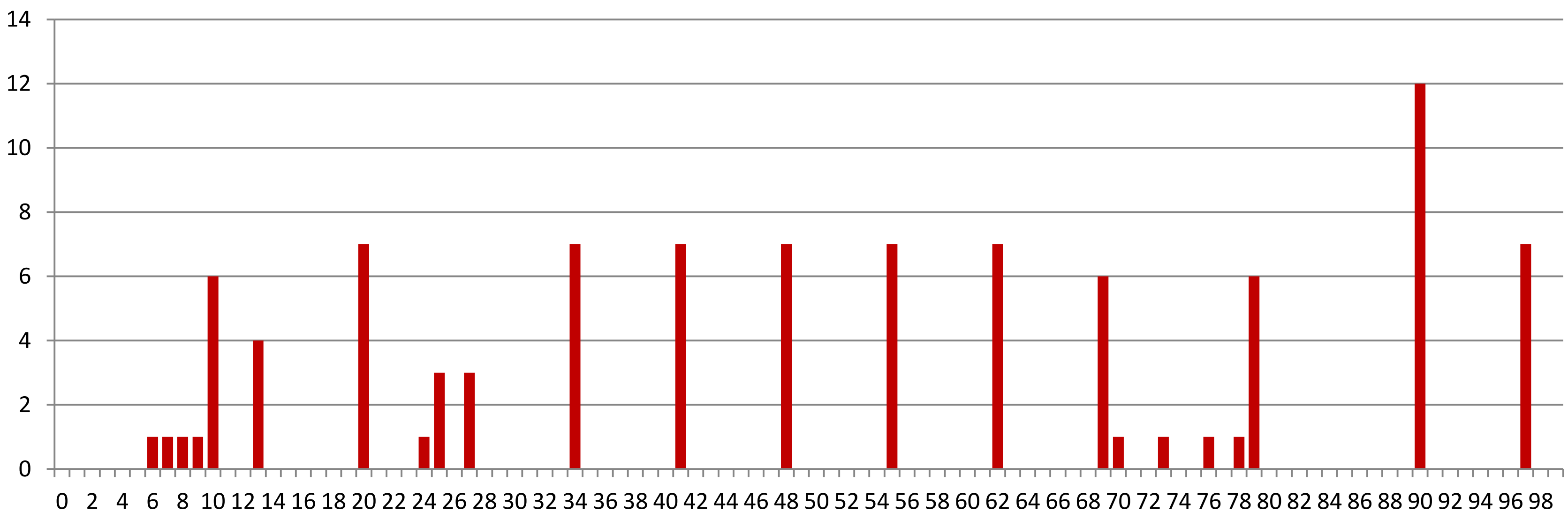
No Race



Single core laptop



Multicore server



- **The race can really happen!**

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