# Server Programming

**Computer Systems**

David Marchant

**Based on slides by:**

Randal E. Bryant and David R. O'Hallaron

# Some reminders…

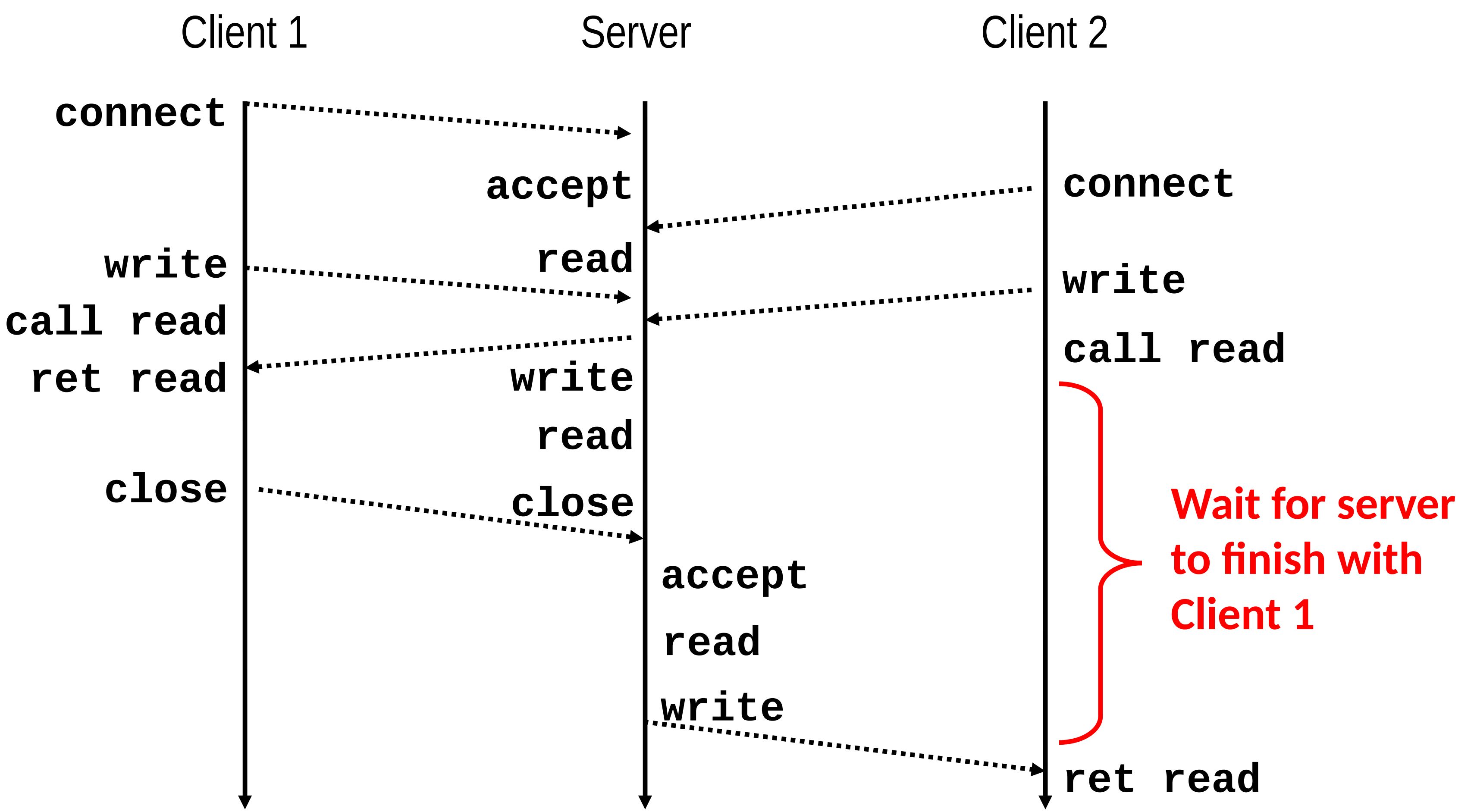
- **Threads & Processes:** Both can mainting concurrent logical control through context switching. Threads are lighter weight and share more data.

- **Concurrency & Parrallel:** Parallel means running different processing at the literal same time. Concurrency can simulate this by interweaving

- **Semaphores & Mutex:** Synchronisation tools to ensure that we don’t encounter concurrency problems such as race conditions or deadlock

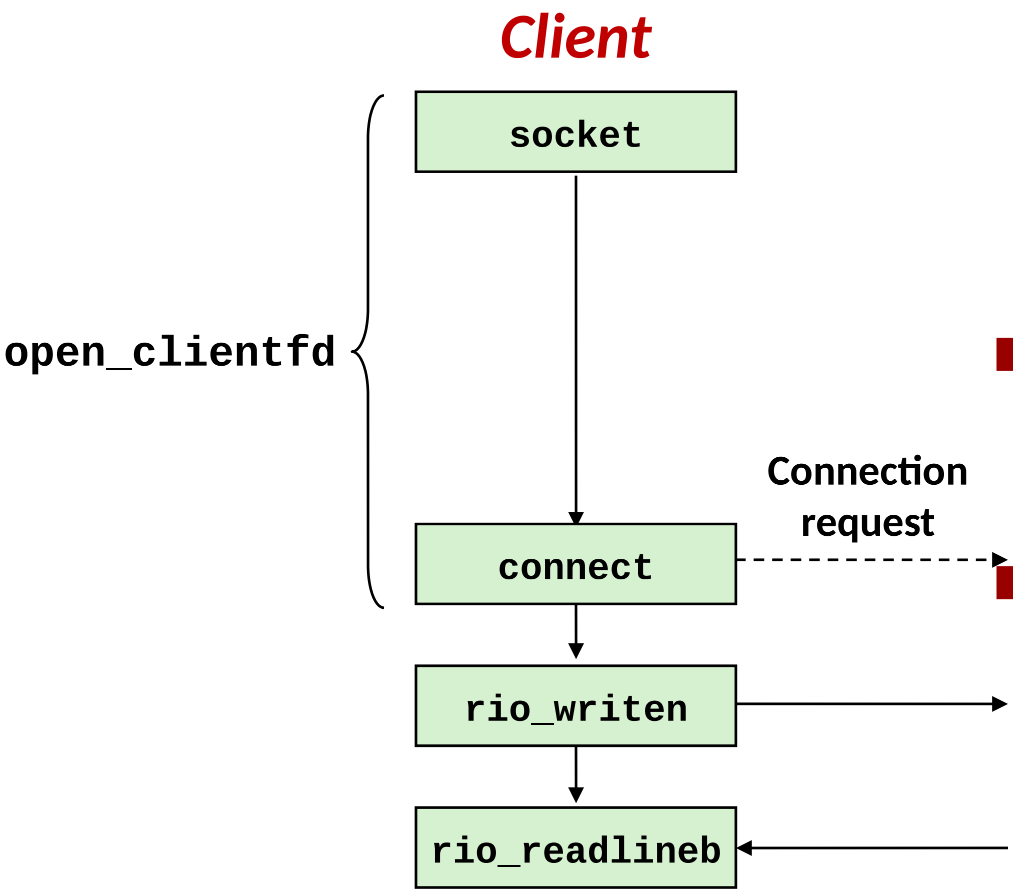
# Iterative Servers

- Iterative servers process one request at a time



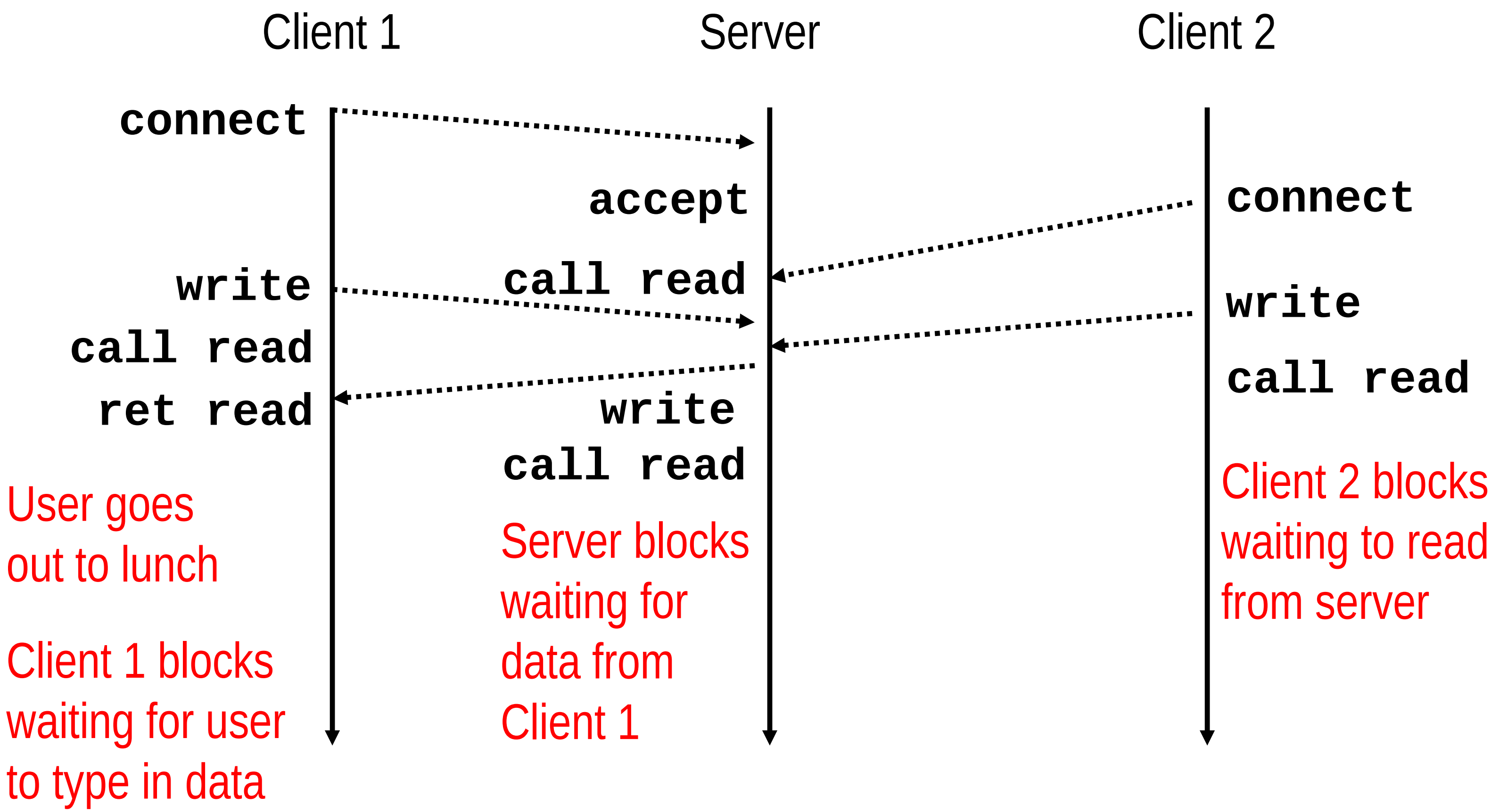
# Where Does Second Client Block?

- 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

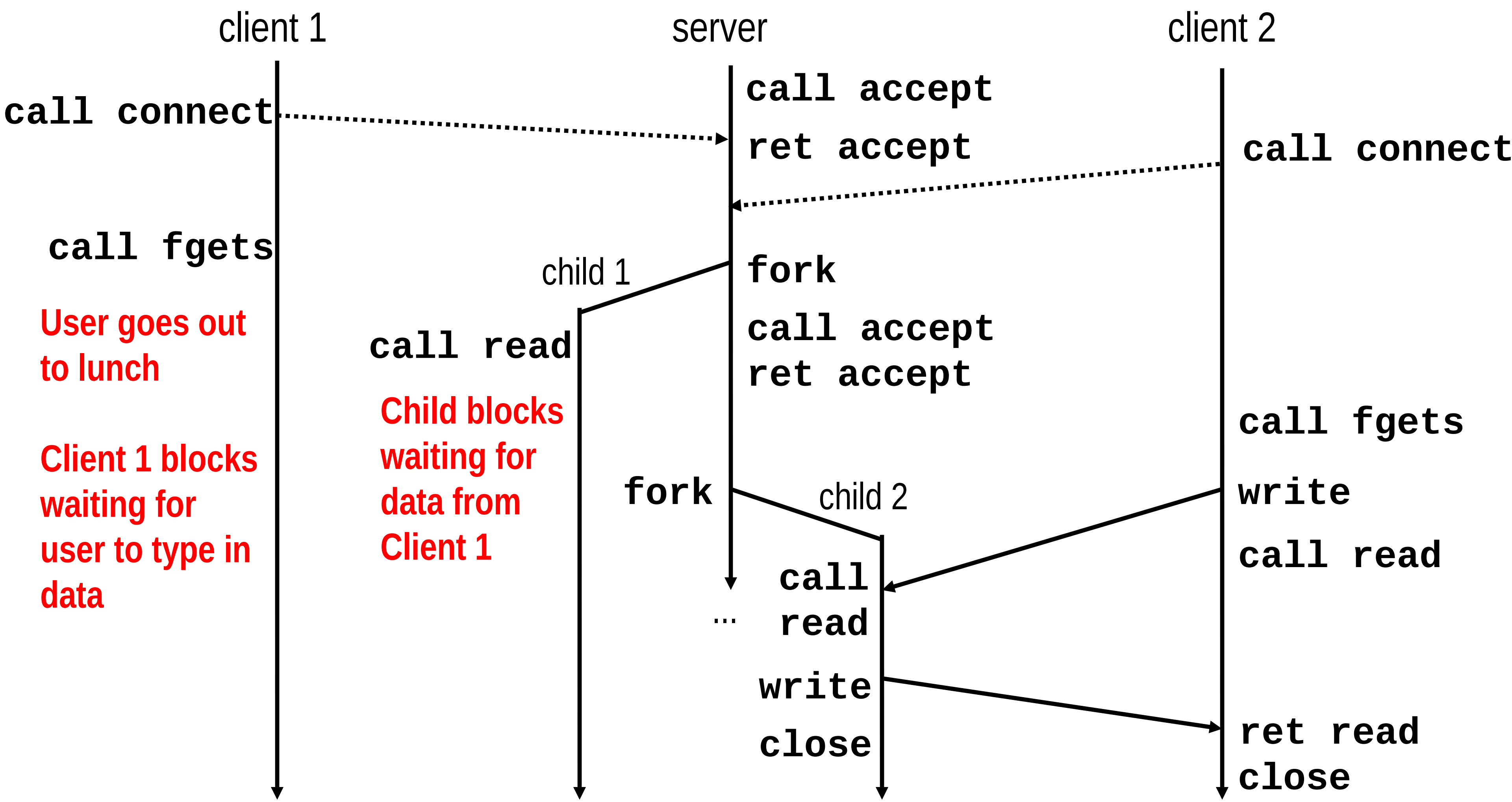
- Programmer manually interleaves multiple logical flows
- All flows share the same address space
- Uses technique called *I/O multiplexing*.

## 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)

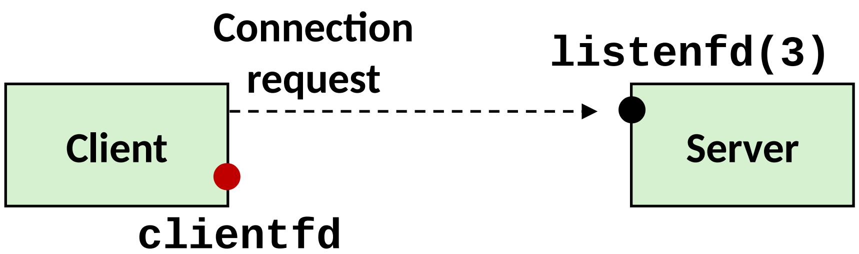
```
void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) > 0)
        ;
    return;
}
```

echoserverp.c

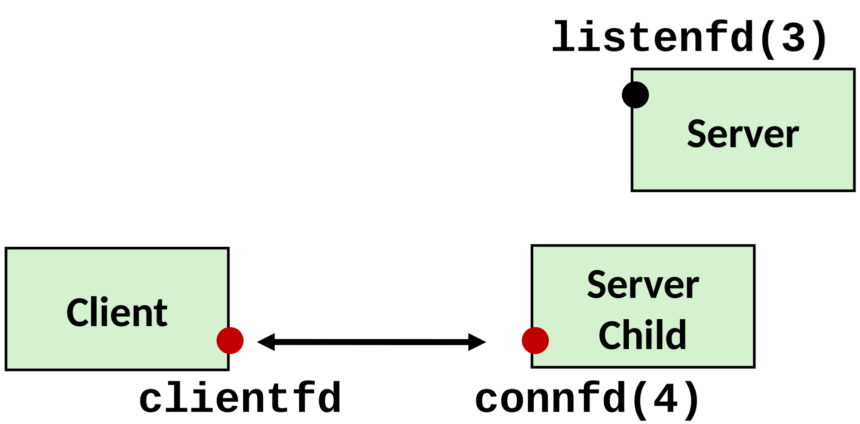
- Reap all zombie children

# Concurrent Server: `accept` Illustrated


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

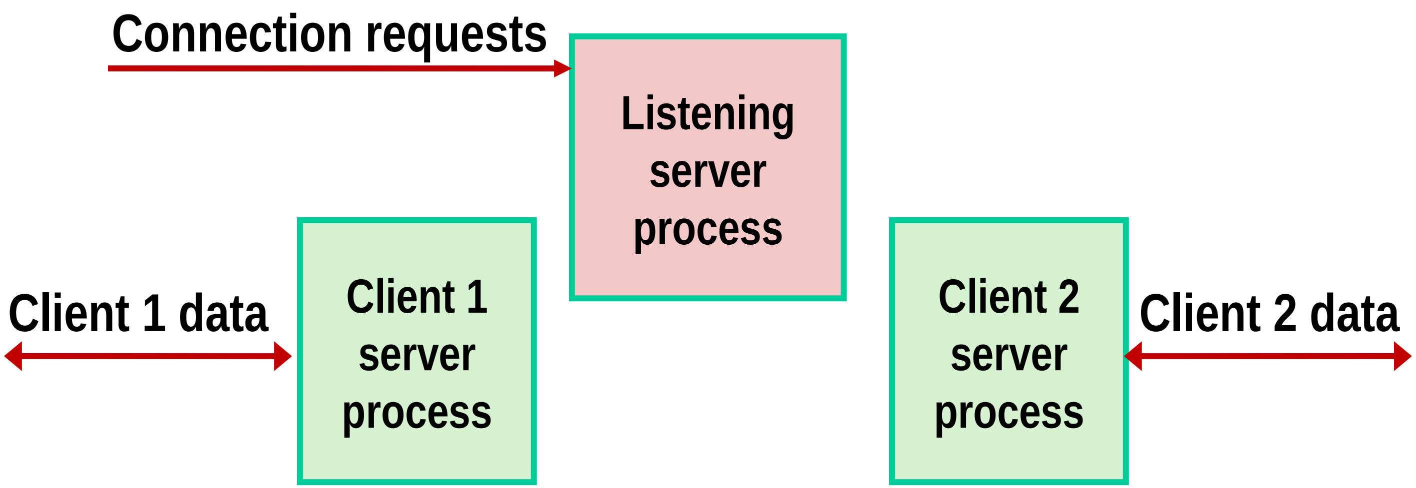



*2. Client makes connection request by calling **`connect`***




*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

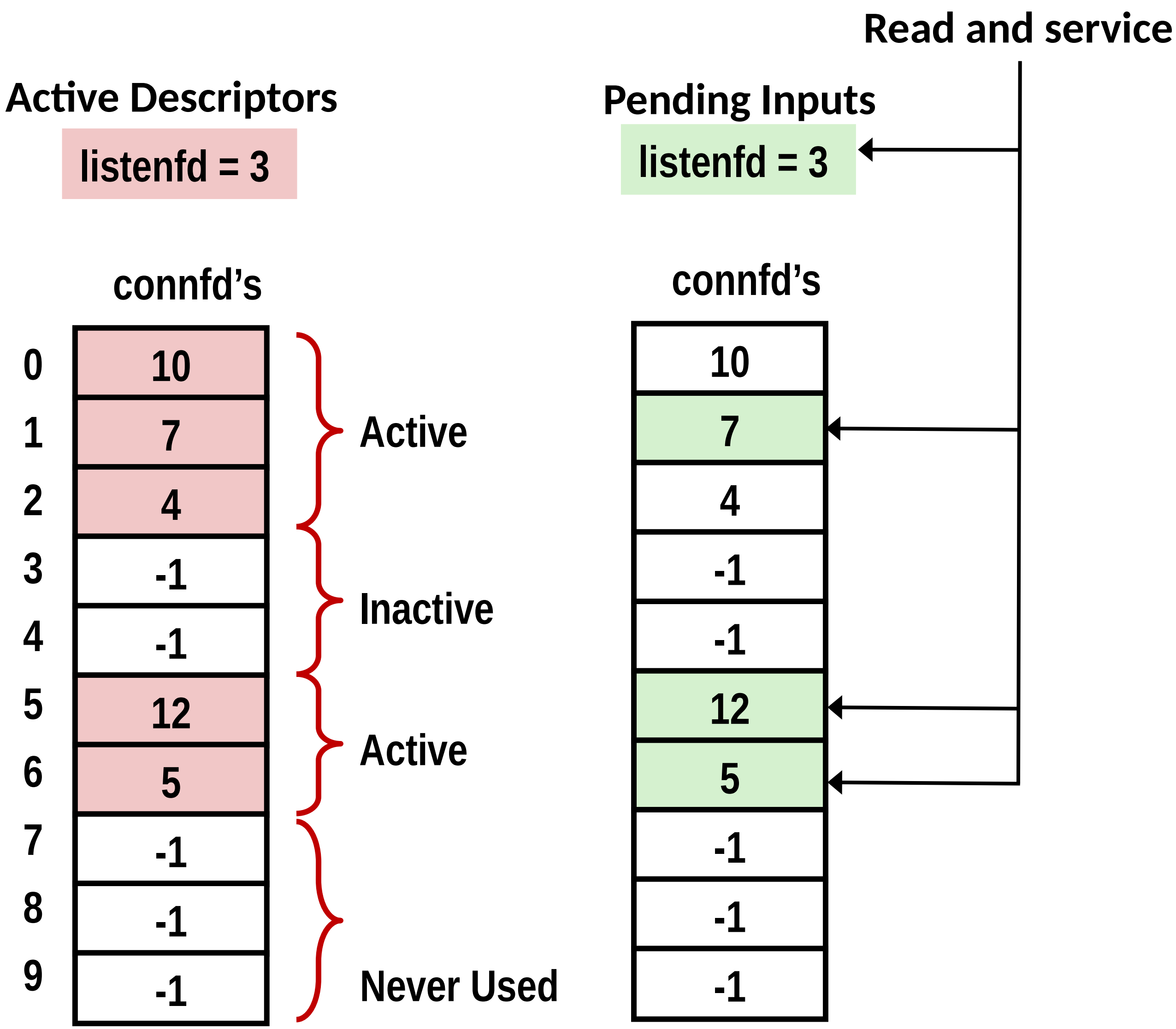
- **+ Handle multiple connections concurrently**
- **+ Clean sharing model**
  - descriptors (no)
  - file tables (yes)
  - global variables (no)
- **+ Simple and straightforward**
- **– Additional overhead for process control**
- **– Nontrivial to share data between processes**
  - Requires IPC (interprocess communication) mechanisms
    - FIFO's (named pipes), System V shared memory and semaphores

# 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` or `epoll` functions
    - 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



# 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
  - We've already seen this in the concurrency section of the course so won't re-iterate here

# 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

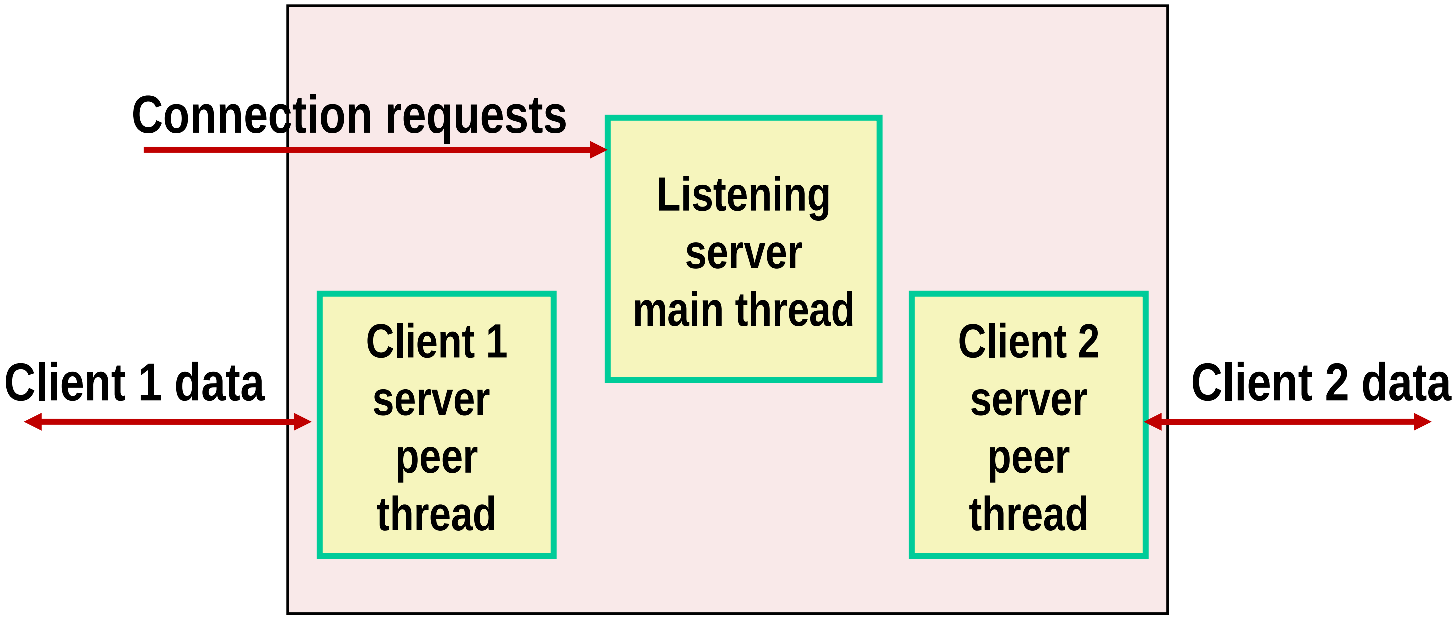
- `malloc` of connected descriptor necessary to avoid deadly race (later)

# 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 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)

# 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

# One worry: Races

- A *race* occurs when correctness of the program depends on one thread reaching point x before another thread reaches point y

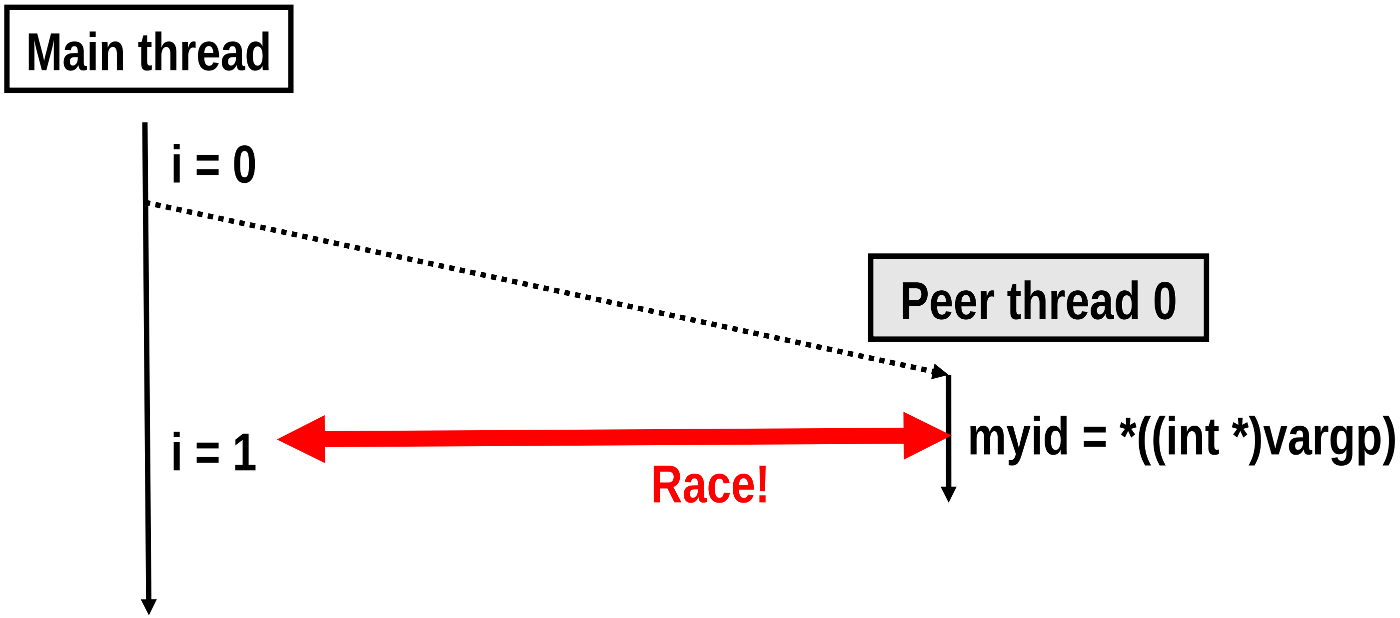
```
/* A threaded program with a race */
int main()
{
    pthread_t tid[N];
    int i;

    for (i = 0; i < N; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

N threads are sharing i

race.c

# Race Illustration

```
for (i = 0; i < N; i++)
    Pthread_create(&tid[i], NULL, thread, &i);
```




- **Race between increment of i in main thread and deref of vargp in peer thread:**
  - If deref happens while i = 0, then OK
  - Otherwise, peer thread gets wrong id value

# Could this race really occur?

**Main thread**

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

**Peer thread**

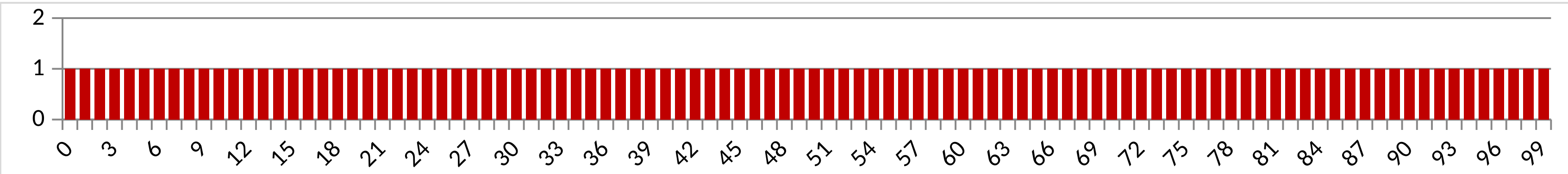
```
void *thread(void *vargp) {
   Pthread_detach(pthread_self());
   int i = *((int *)vargp);
   save_value(i);
   return NULL;
}
```

race.c

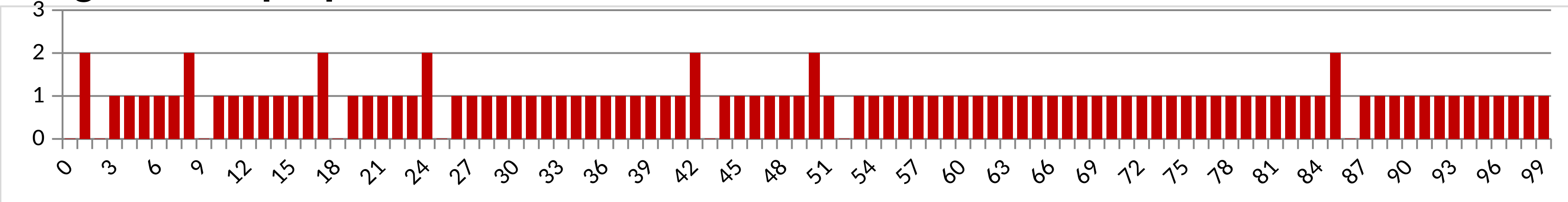
- **Race Test**
  - If no race, then each thread would get different value of i
  - Set of saved values would consist of one copy each of 0 through 99

# Experimental Results

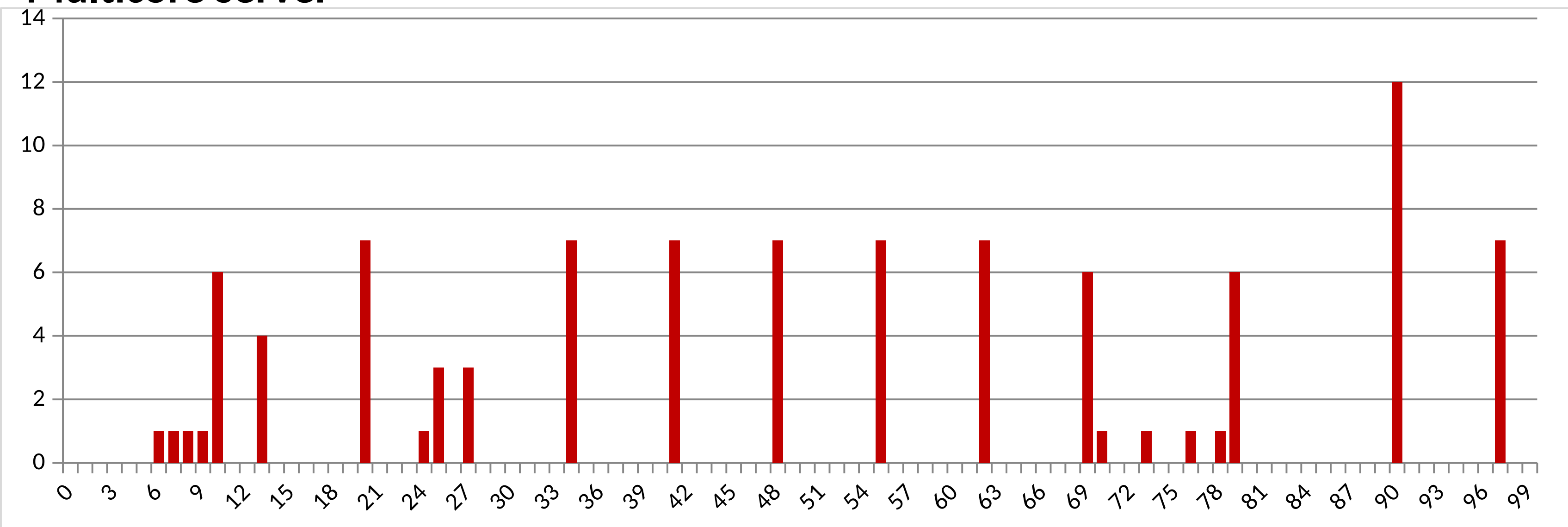
### No Race




### Single core laptop




### Multicore server




- **The race can really happen!**

# Race Elimination

```
/* Threaded program without the race */
int main()
{
    pthread_t tid[N];
    int i, *ptr;

    for (i = 0; i < N; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], NULL, thread, ptr);
    }
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp);
    Free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

norace.c

- Avoid unintended sharing of state

# Another worry: Deadlock

- **Def: A process is *deadlocked* iff it is waiting for a condition that will never be true**

- **Typical Scenario**
  - Processes 1 and 2 needs two resources (A and B) to proceed
  - Process 1 acquires A, waits for B
  - Process 2 acquires B, waits for A
  - Both will wait forever!

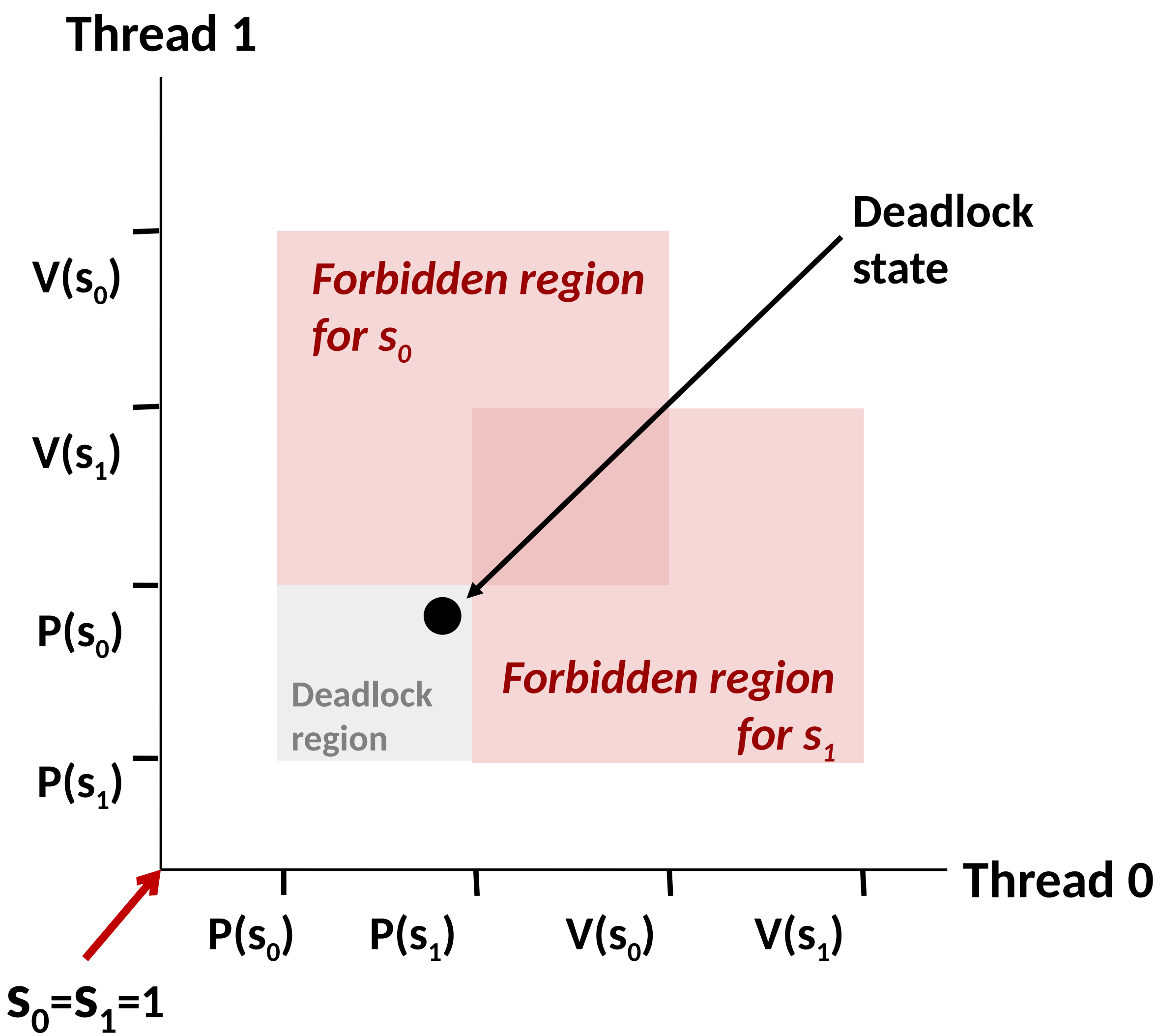
# Deadlocking With Semaphores

```
int main()
{
   pthread_t tid[2];
   Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
   Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
   Pthread_create(&tid[0], NULL, count, (void*) 0);
   Pthread_create(&tid[1], NULL, count, (void*) 1);
   Pthread_join(tid[0], NULL);
   Pthread_join(tid[1], NULL);
   printf("cnt=%d\n", cnt);
   exit(0);
}
```

```
void *count(void *vargp)
{
   int i;
   int id = (int) vargp;
   for (i = 0; i < NITERS; i++) {
      P(&mutex[id]); P(&mutex[1-id]);
      cnt++;
      V(&mutex[id]); V(&mutex[1-id]);
   }
   return NULL;
}
```

| Tid[0]: | Tid[1]: |
|---|---|
| P($s_0$); | P($s_1$); |
| P($s_1$); | P($s_0$); |
| cnt++; | cnt++; |
| V($s_0$); | V($s_1$); |
| V($s_1$); | V($s_0$); |

# Deadlock Visualized in Progress Graph




Locking introduces the potential for *deadlock:* waiting for a condition that will never be true

Any trajectory that enters the *deadlock region* will eventually reach the *deadlock state,* waiting for either $S_0$ or $S_1$ to become nonzero

Other trajectories luck out and skirt the deadlock region

Unfortunate fact: deadlock is often nondeterministic (race)

# Avoiding Deadlock

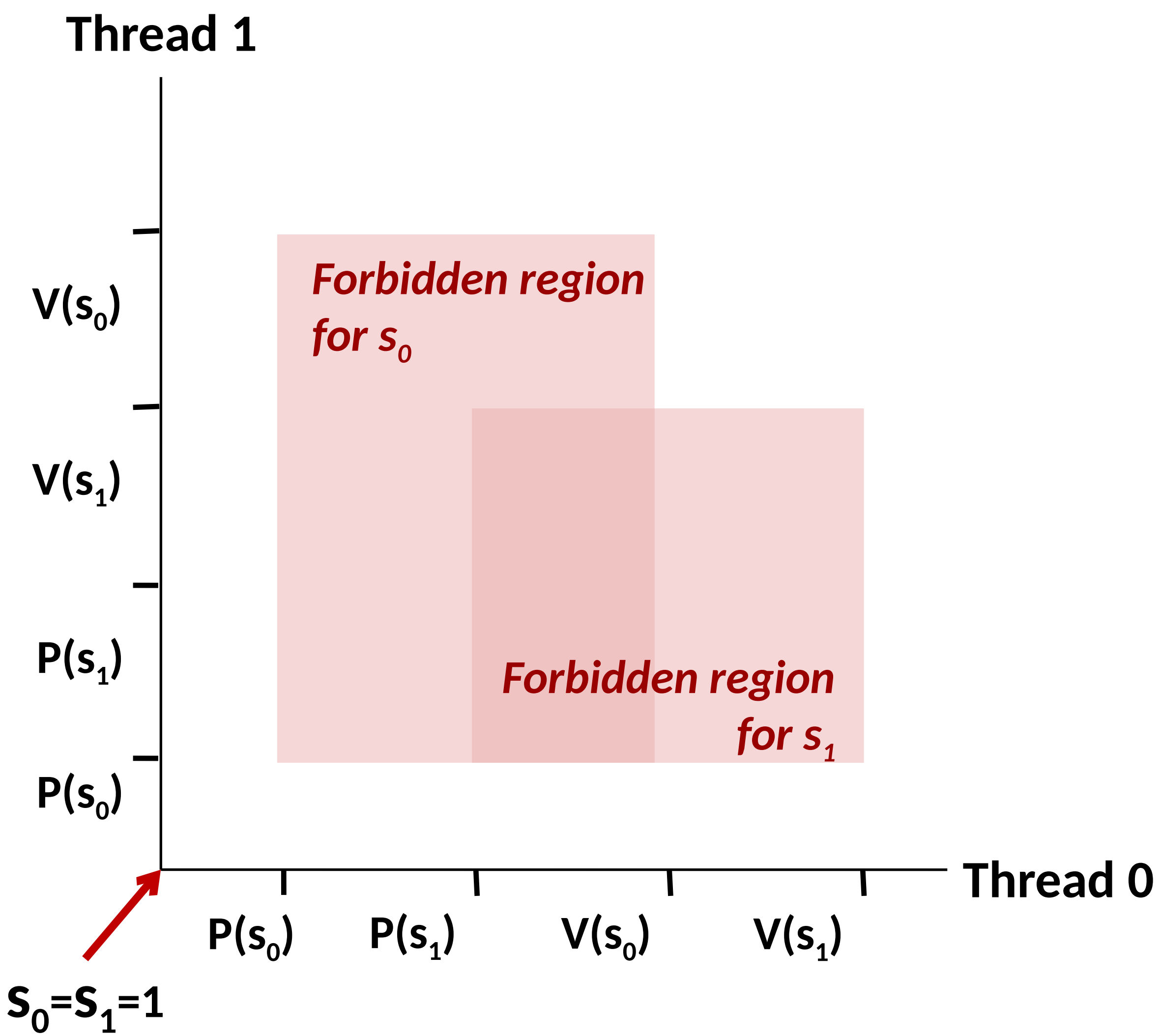
*Acquire shared resources in same order*

```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[0]); P(&mutex[1]);
    cnt++;
    V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

| Tid[0]: | Tid[1]: |
|---|---|
| P(s0); | P(s0); |
| P(s1); | P(s1); |
| cnt++; | cnt++; |
| V(s0); | V(s1); |
| V(s1); | V(s0); |

# Avoided Deadlock in Progress Graph




No way for trajectory to get stuck

Processes acquire locks in same order

Order in which locks released immaterial

# 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