Concurrent Programming

CSE4100: System Programming

Youngjae Kim (PhD)
https://sites.google.com/site/youkim/home
Distributed Computing and Operating Systems Laboratory (DISCOS)
https://discos.sogang.ac.kr
Office: R911, E-mail: youkim@sogang.ac.kr

Concurrent Programming is Hard!

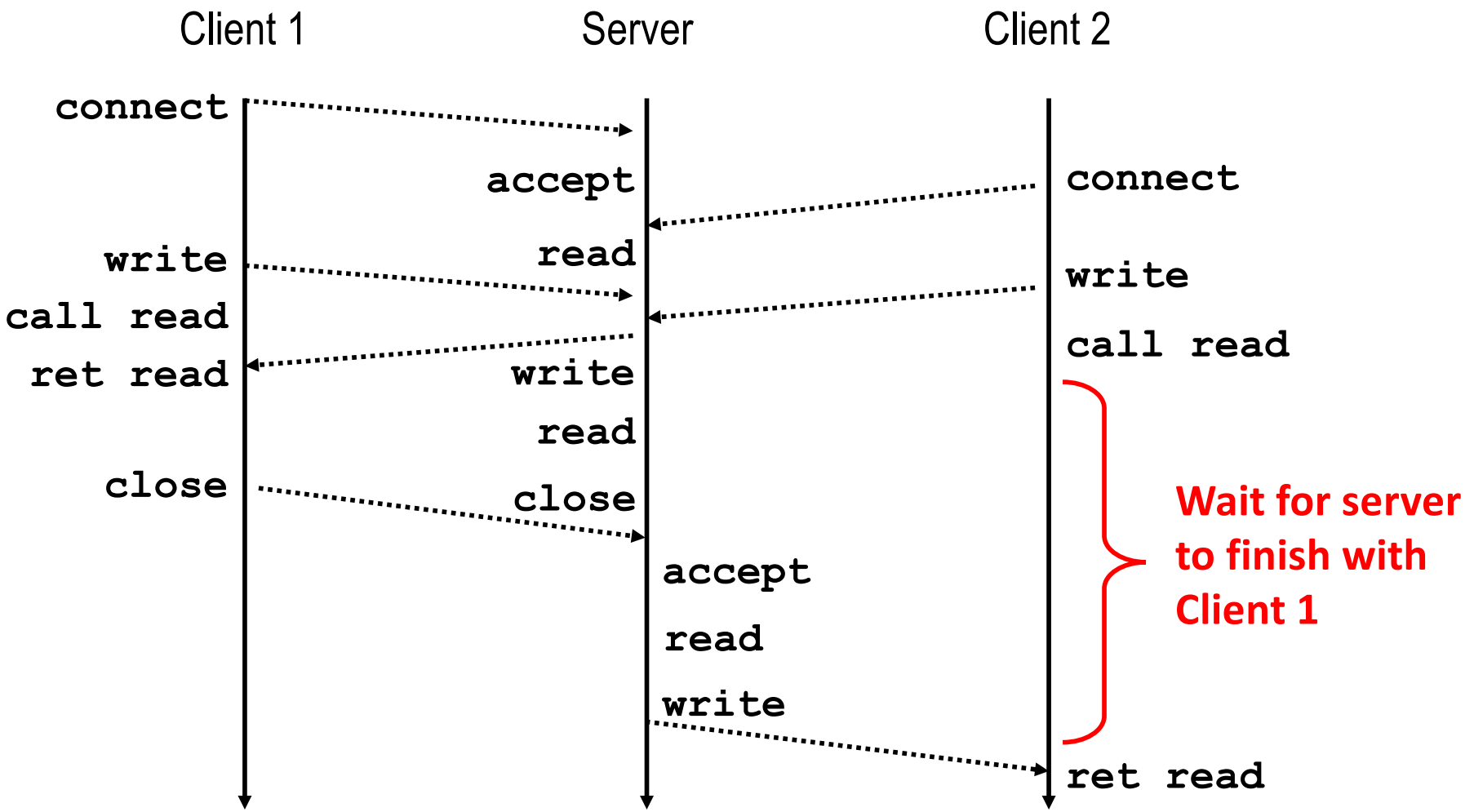
- The human mind tends to be sequential
- The notion of time is often misleading
- Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible

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 our course..**
  - but, not all ☺
  - We'll cover some of these aspects in the next few lectures.

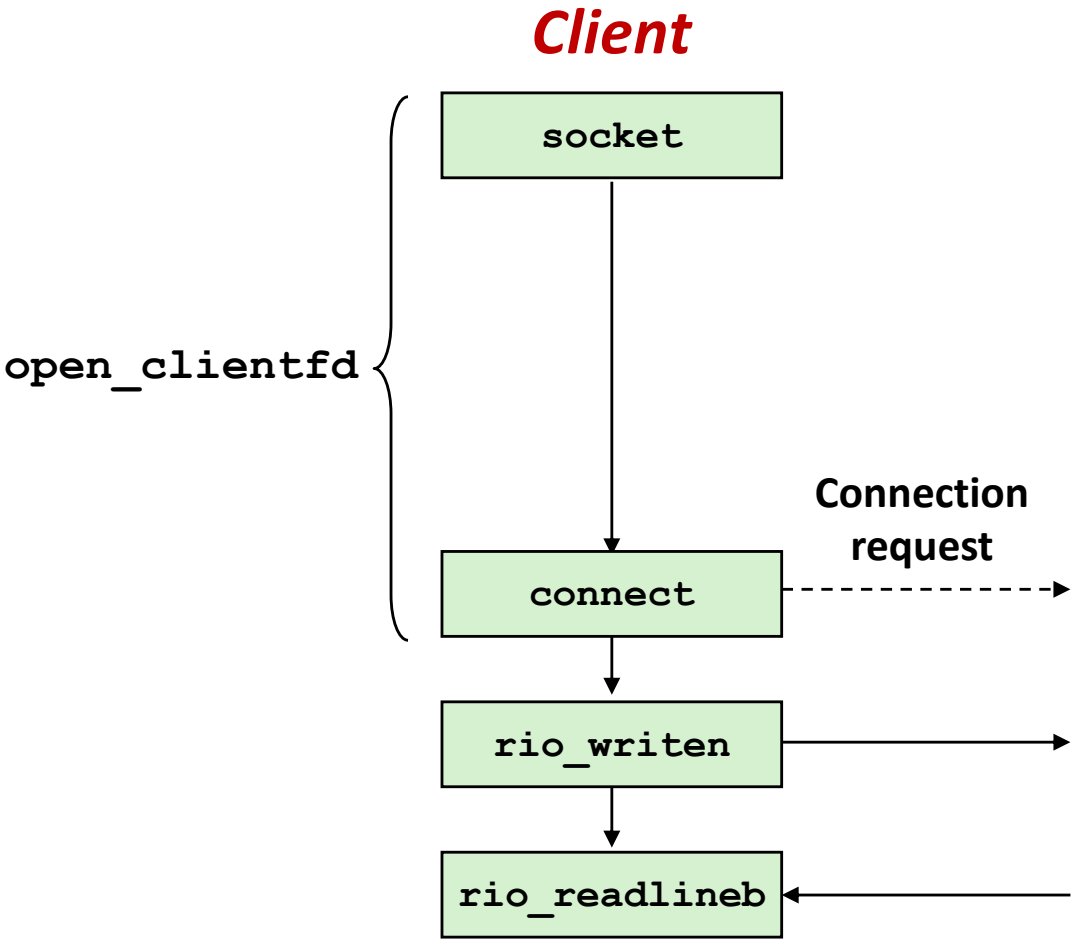
Iterative Servers

- Iterative servers process one request at a time



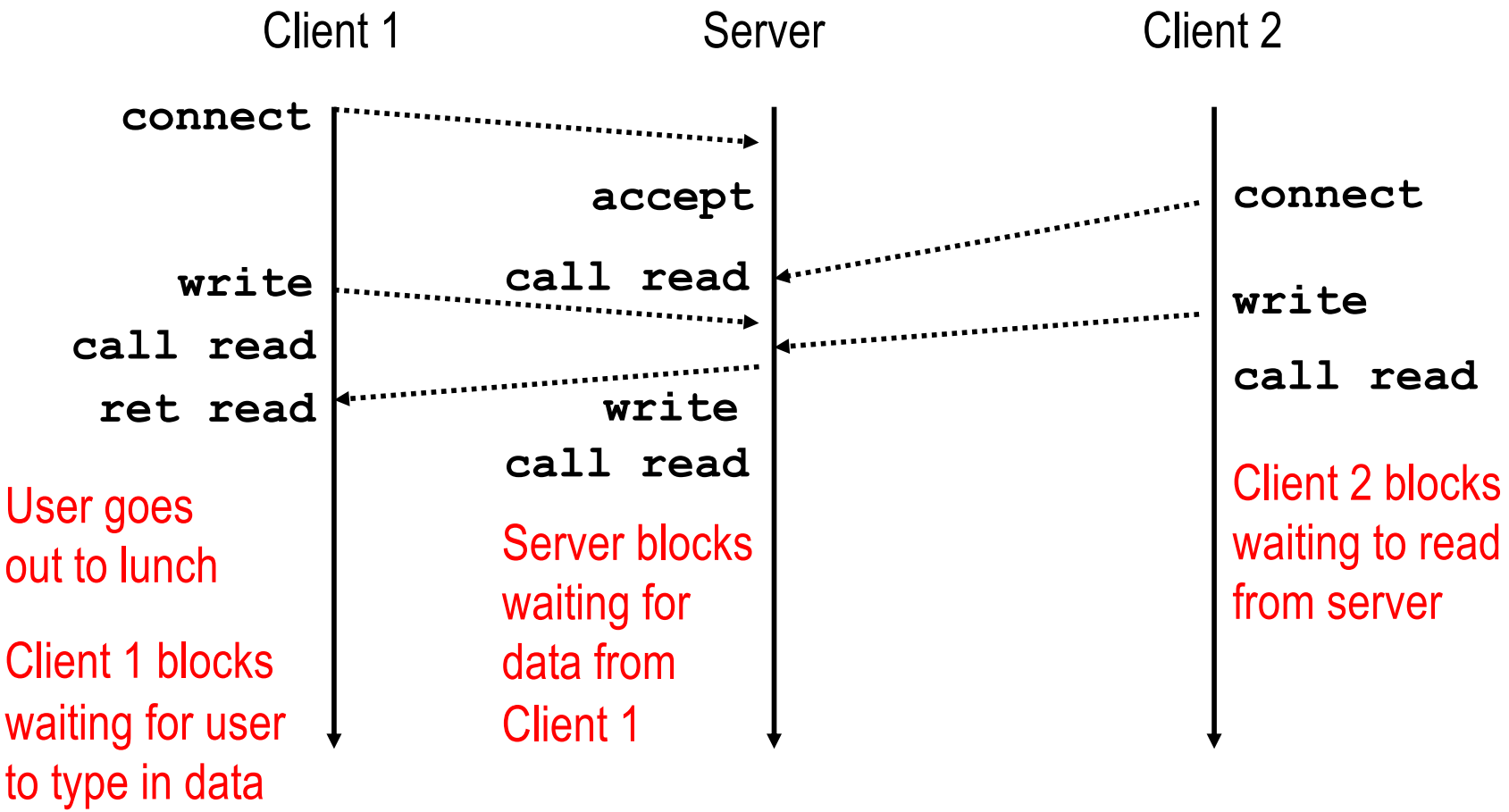
Where Does Second Client Block?

- **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

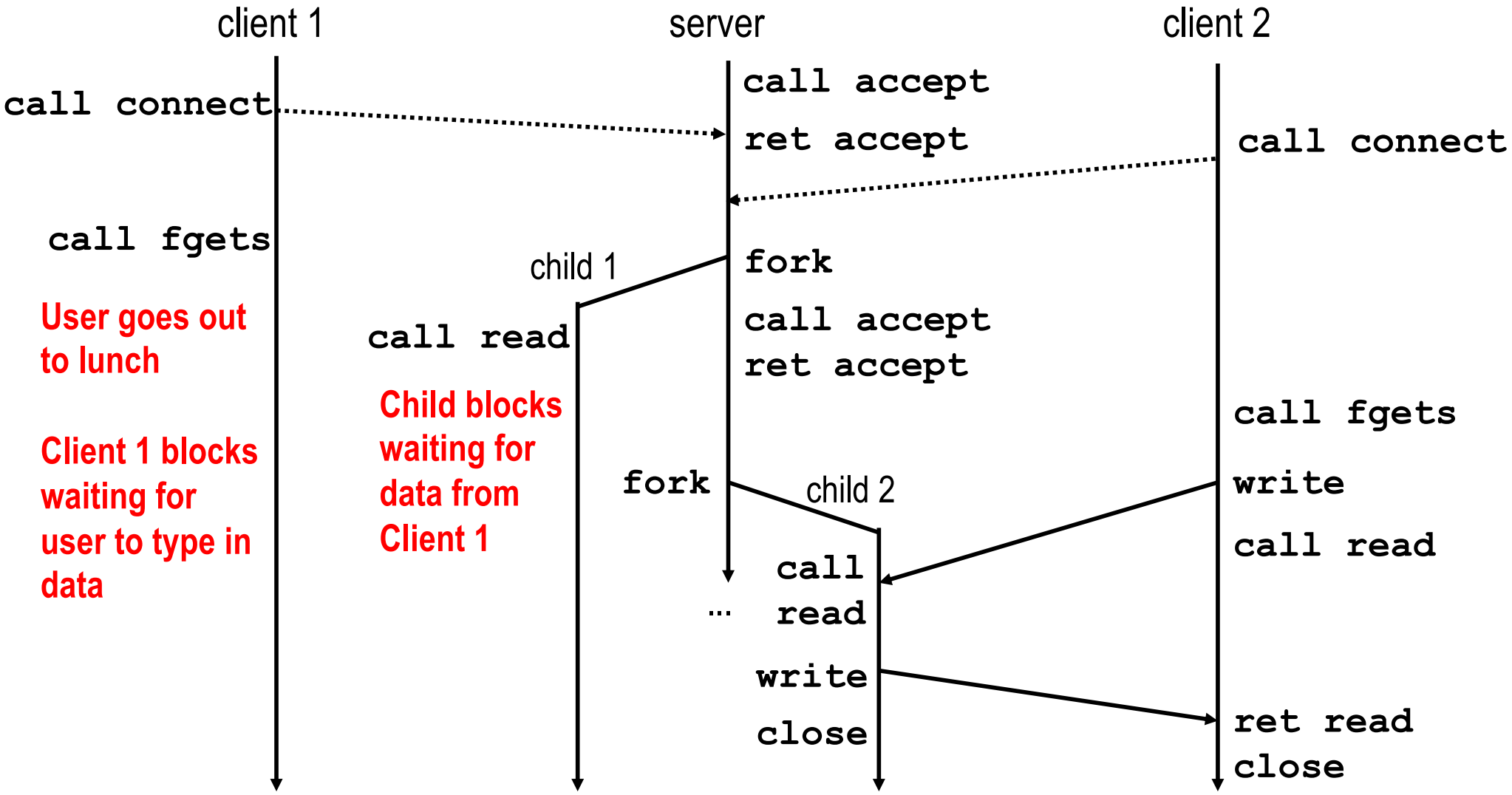
- Programmer manually interleaves multiple logical flows
- All flows share the same address space
- Uses technique called *I/O multiplexing*.

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)

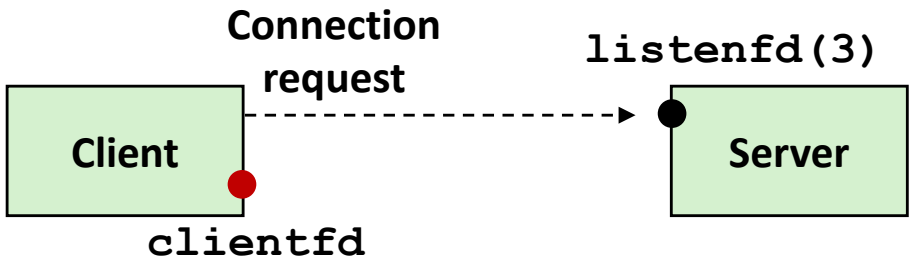
- Reap all zombie children

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

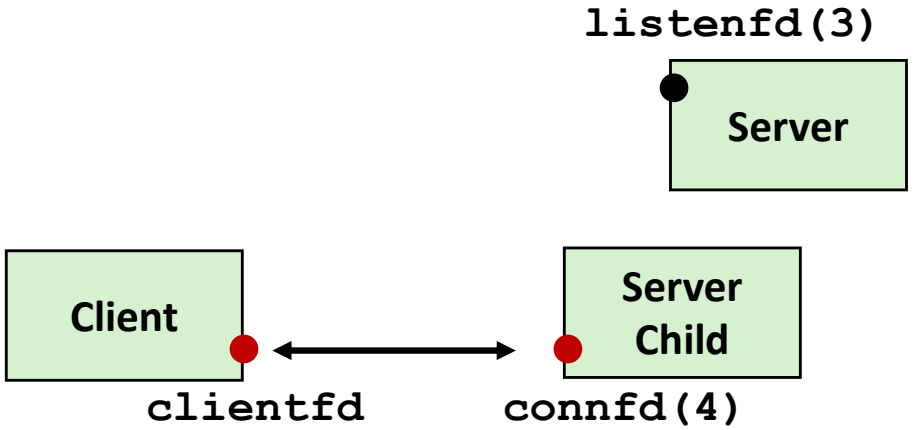
echoserverp.c

Concurrent Server: `accept` Illustrated

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

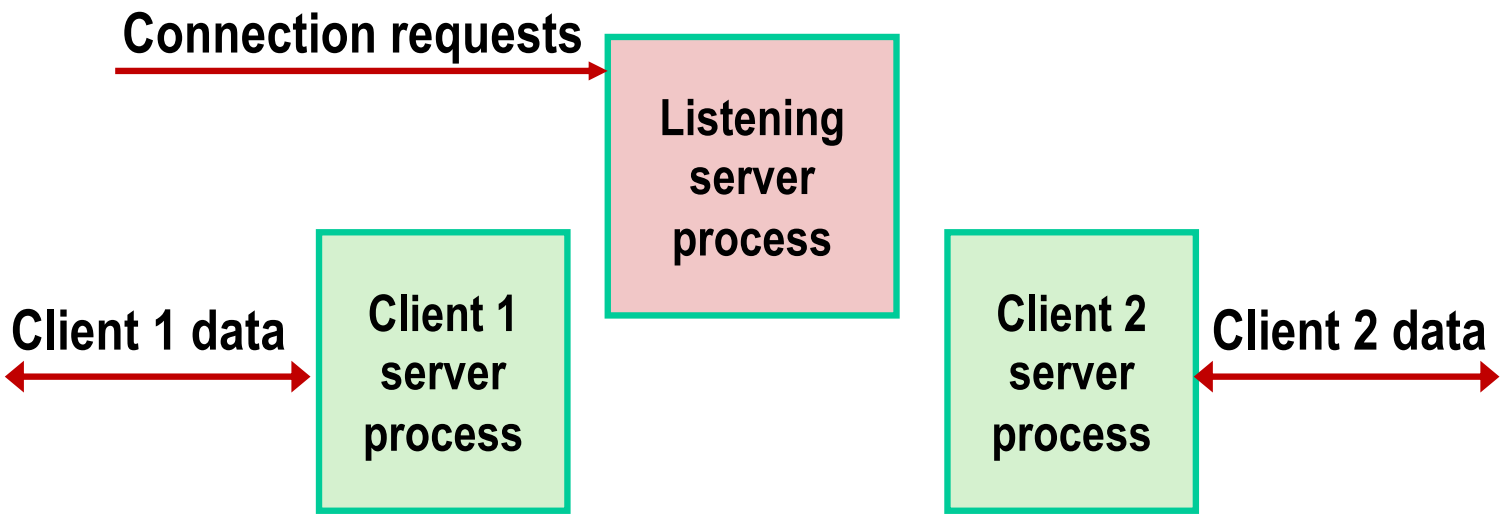


2. Client makes connection request by calling `connect`



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`

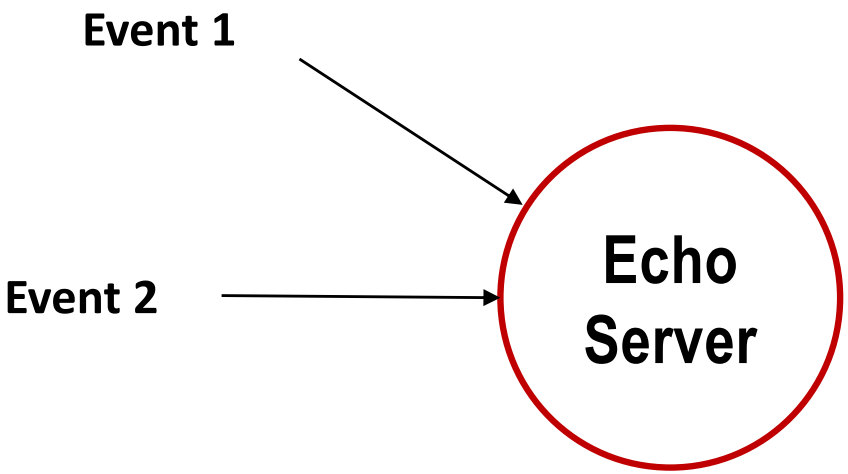
Pros and Cons of Process-based Servers

- **+ 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

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



- Event 1: a network client making a connection request
- Event2: a user typing a command line at a keyboard

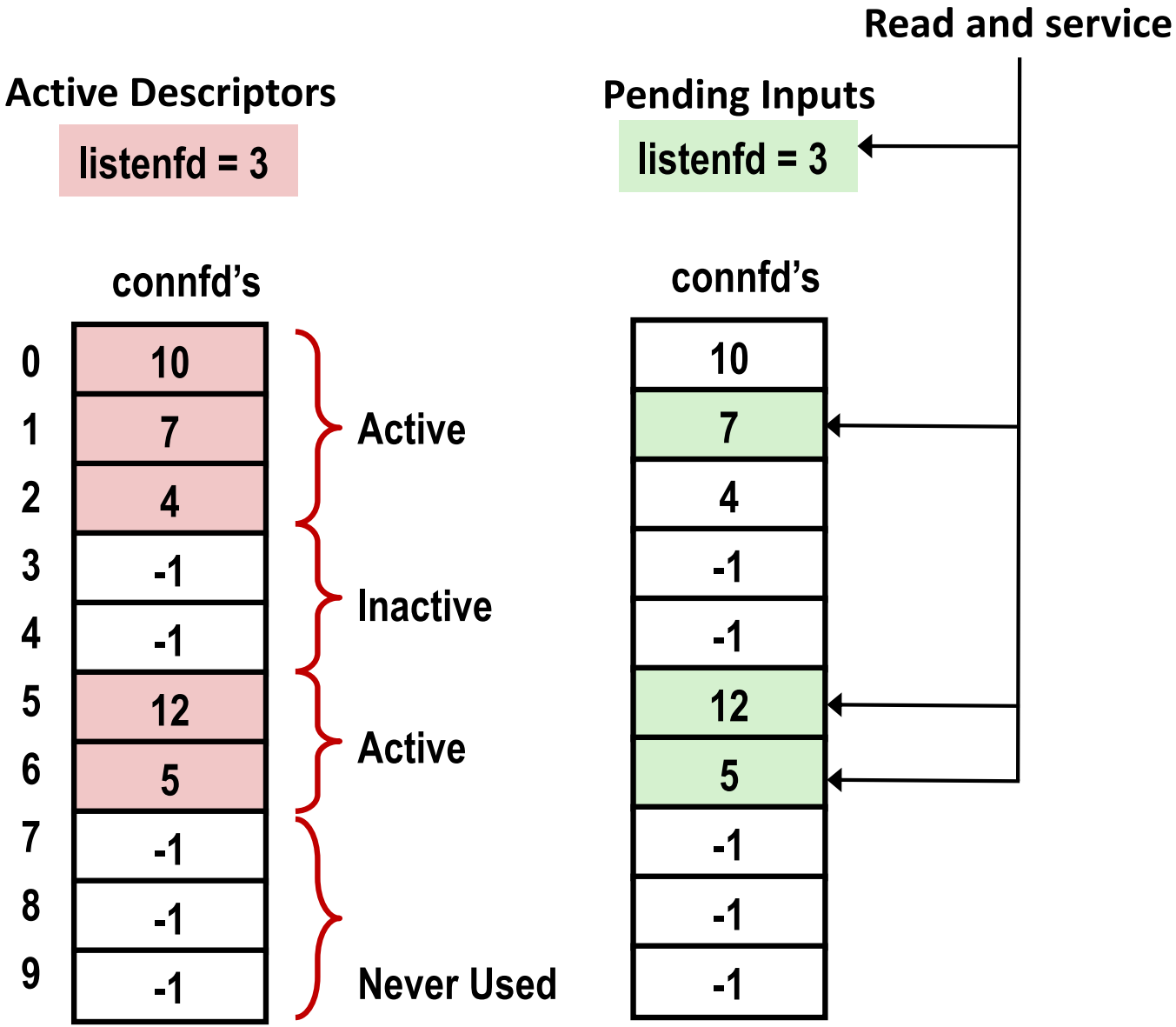
Which event do we wait for first?

Neither option is ideal…

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

I/O Multiplexed Event Processing



I/O Multiplexed Event Processing

- **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
- **Example**
  - Return when any descriptor in the set {0, 4} is ready for reading
  - Return when any descriptor in the set {1, 2, 7} is ready for writing.

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

int select(int n, fd_set *fdset, NULL, NULL, NULL);
                          Returns: nonzero count of ready descriptors, −1 on error

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

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 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`

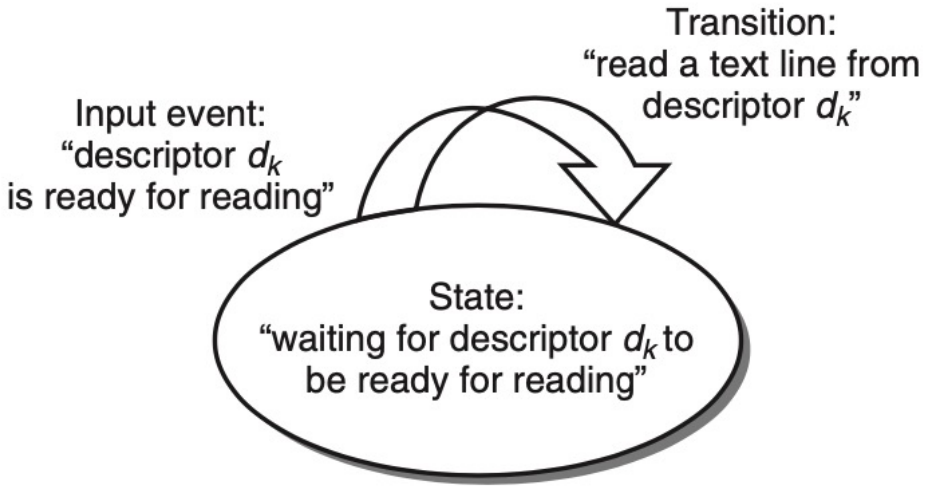
- **Blocking problems**
  - Once the server connects to a client, it continues echoing input lines until the client closes its end of the connection.

    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.

  - Thus, we need a way to multiplex at a finer granualarity, 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 progres 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**



```
1   #include "csapp.h"
2
3   typedef struct { /* Represents a pool of connected descriptors */
4       int maxfd;        /* Largest descriptor in read_set */
5       fd_set read_set;  /* Set of all active descriptors */
6       fd_set ready_set; /* Subset of descriptors ready for reading  */
7       int nready;       /* Number of ready descriptors from select */
8       int maxi;         /* High water index into client array */
9       int clientfd[FD_SETSIZE];    /* Set of active descriptors */
10      rio_t clientrio[FD_SETSIZE]; /* Set of active read buffers */
11  } pool;
12
13  int byte_cnt = 0; /* Counts total bytes received by server */
14
15  int main(int argc, char **argv)
16  {
17      int listenfd, connfd;
18      socklen_t clientlen;
19      struct sockaddr_storage clientaddr;
20      static pool pool;
21
22      if (argc != 2) {
23          fprintf(stderr, "usage: %s <port>\n", argv[0]);
24          exit(0);
25      }
26      listenfd = Open_listenfd(argv[1]);
27      init_pool(listenfd, &pool);
```

```
28
29      while (1) {
30          /* Wait for listening/connected descriptor(s) to become ready */
31          pool.ready_set = pool.read_set;
32          pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);
33
34          /* If listening descriptor ready, add new client to pool */
35          if (FD_ISSET(listenfd, &pool.ready_set)) {
36              clientlen = sizeof(struct sockaddr_storage);
37              connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
38              add_client(connfd, &pool);
39          }
40
41          /* Echo a text line from each ready connected descriptor */
42          check_clients(&pool);
43      }
44  }
```

code/conc/echoservers.c

code/conc/echoservers.c

```
1   void init_pool(int listenfd, pool *p)
2   {
3       /* Initially, there are no connected descriptors */
4       int i;
5       p->maxi = -1;
6       for (i=0; i< FD_SETSIZE; i++)
7           p->clientfd[i] = -1;
8
9       /* Initially, listenfd is only member of select read set */
10      p->maxfd = listenfd;
11      FD_ZERO(&p->read_set);
12      FD_SET(listenfd, &p->read_set);
13  }
```

code/conc/echoservers.c

code/conc/echoservers.c

```
1   void add_client(int connfd, pool *p)
2   {
3       int i;
4       p->nready--;
5       for (i = 0; i < FD_SETSIZE; i++) /* Find an available slot */
6           if (p->clientfd[i] < 0) {
7               /* Add connected descriptor to the pool */
8               p->clientfd[i] = connfd;
9               Rio_readinitb(&p->clientrio[i], connfd);
10
11              /* Add the descriptor to descriptor set */
12              FD_SET(connfd, &p->read_set);
13
14              /* Update max descriptor and pool high water mark */
15              if (connfd > p->maxfd)
16                  p->maxfd = connfd;
17              if (i > p->maxi)
18                  p->maxi = i;
19              break;
20          }
21      if (i == FD_SETSIZE) /* Couldn't find an empty slot */
22          app_error("add_client error: Too many clients");
23  }
```

code/conc/echoservers.c

```
1  void check_clients(pool *p)
2  {
3      int i, connfd, n;
4      char buf[MAXLINE];
5      rio_t rio;
6
7      for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
8          connfd = p->clientfd[i];
9          rio = p->clientrio[i];
10
11         /* If the descriptor is ready, echo a text line from it */
12         if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
13             p->nready--;
14             if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
15                 byte_cnt += n;
16                 printf("Server received %d (%d total) bytes on fd %d\n",
17                        n, byte_cnt, connfd);
18                 Rio_writen(connfd, buf, n);
19             }
20
21             /* EOF detected, remove descriptor from pool */
22             else {
23                 Close(connfd);
24                 FD_CLR(connfd, &p->read_set);
25                 p->clientfd[i] = -1;
26             }
27         }
28     }
29 }
```

Pros and Cons of Event-based Servers

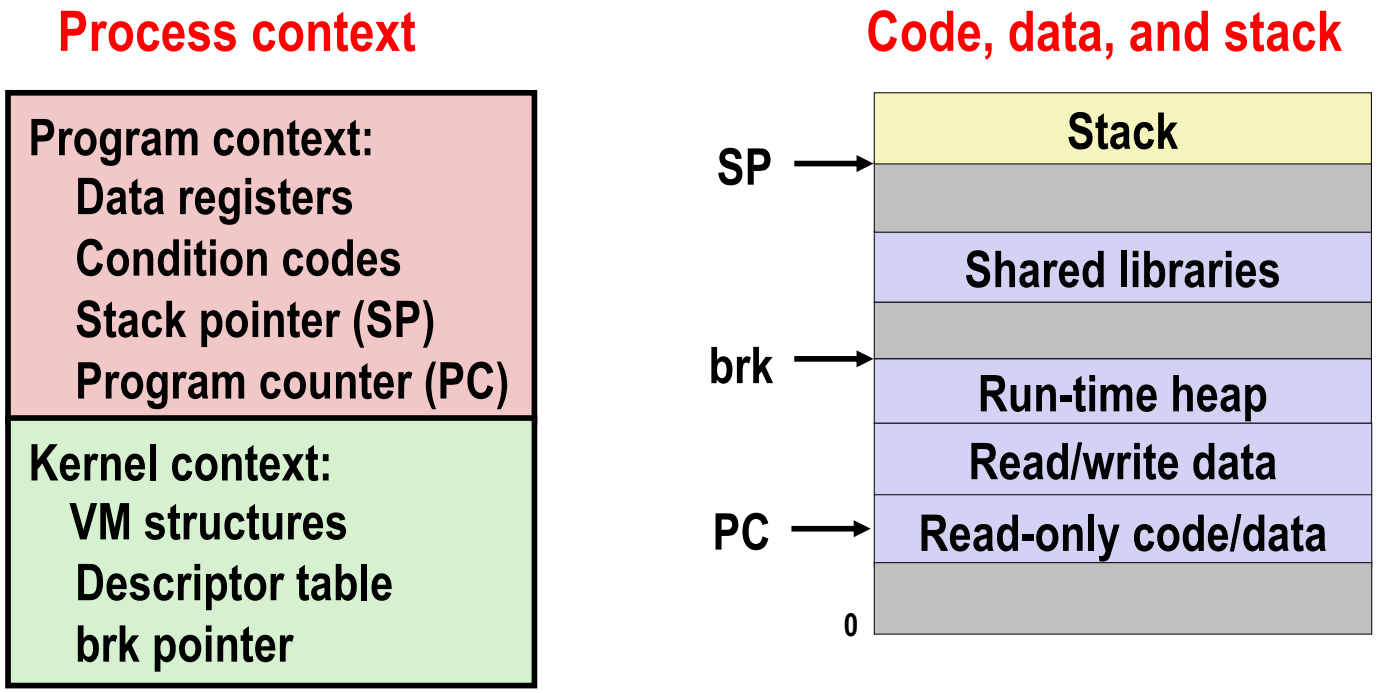
- **+ One logical control flow and address space.**
- **+ Can single-step with a debugger.**
- **+ No process or thread control overhead.**
  - Design of choice for high-performance Web servers and search engines. e.g., Node.js, nginx, Tornado
- **– Significantly more complex to code than process- or 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

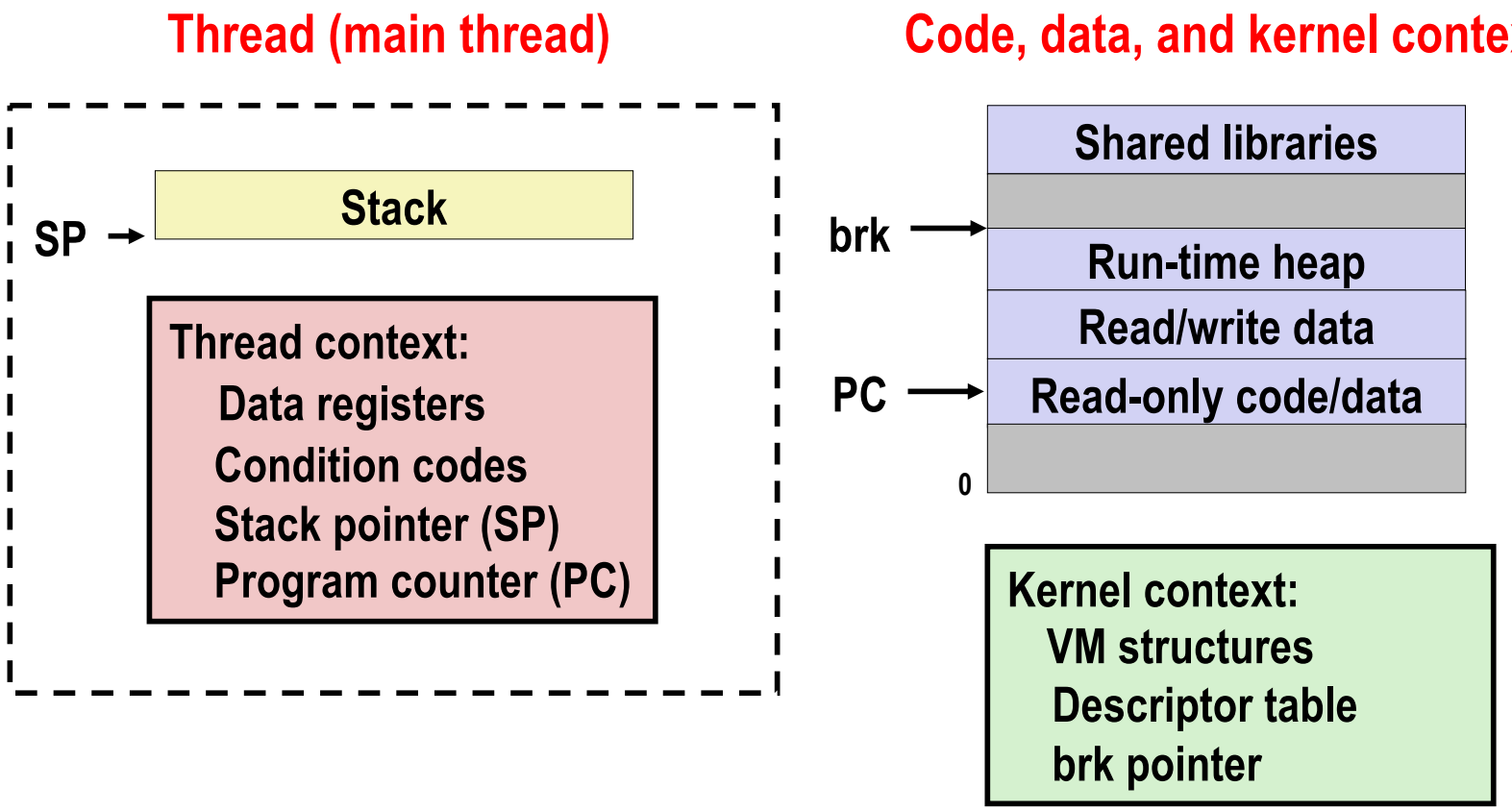
Traditional View of a Process

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



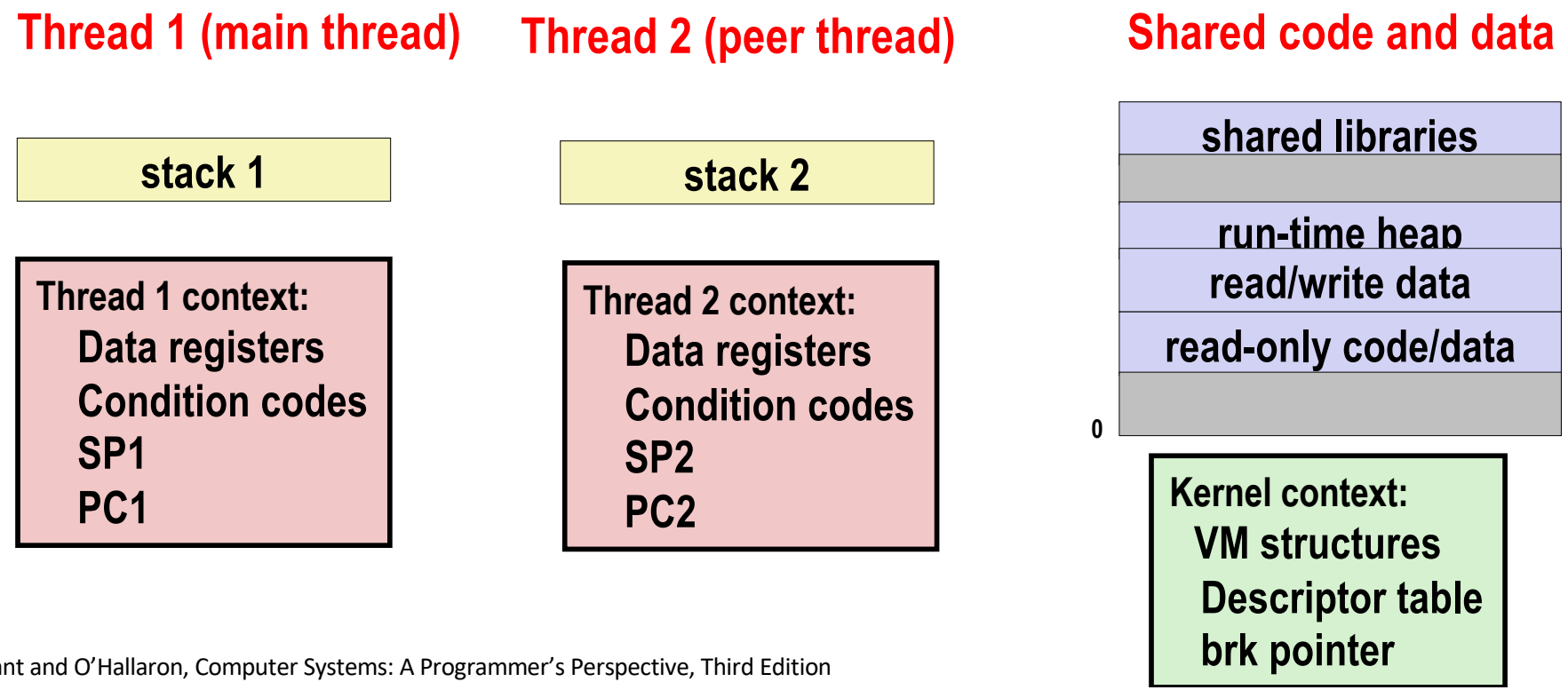
Alternate View of a Process

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



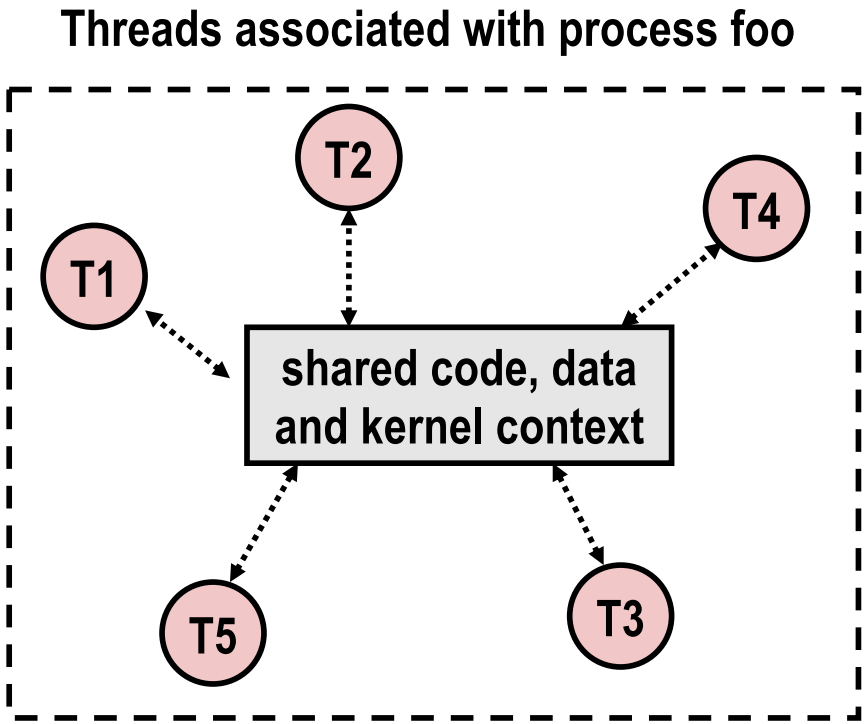
A Process With Multiple Threads

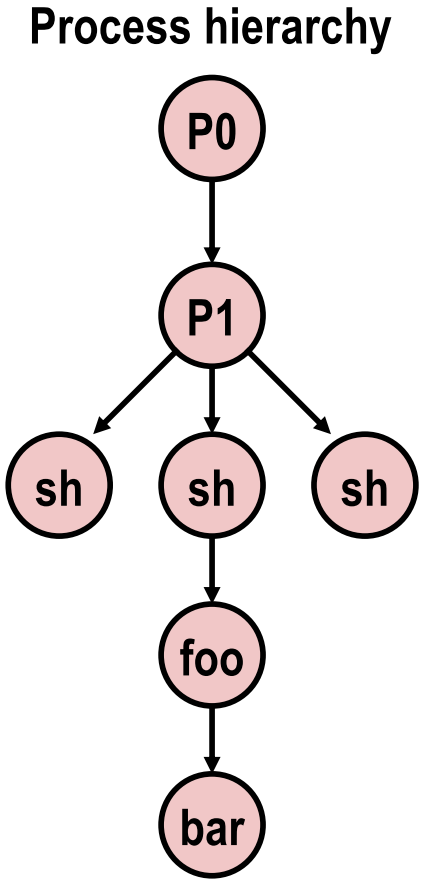
- **Multiple threads can be associated with a process**
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own stack for local variables
    - but not protected from other threads
  - Each thread has its own thread id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



Logical View of Threads

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

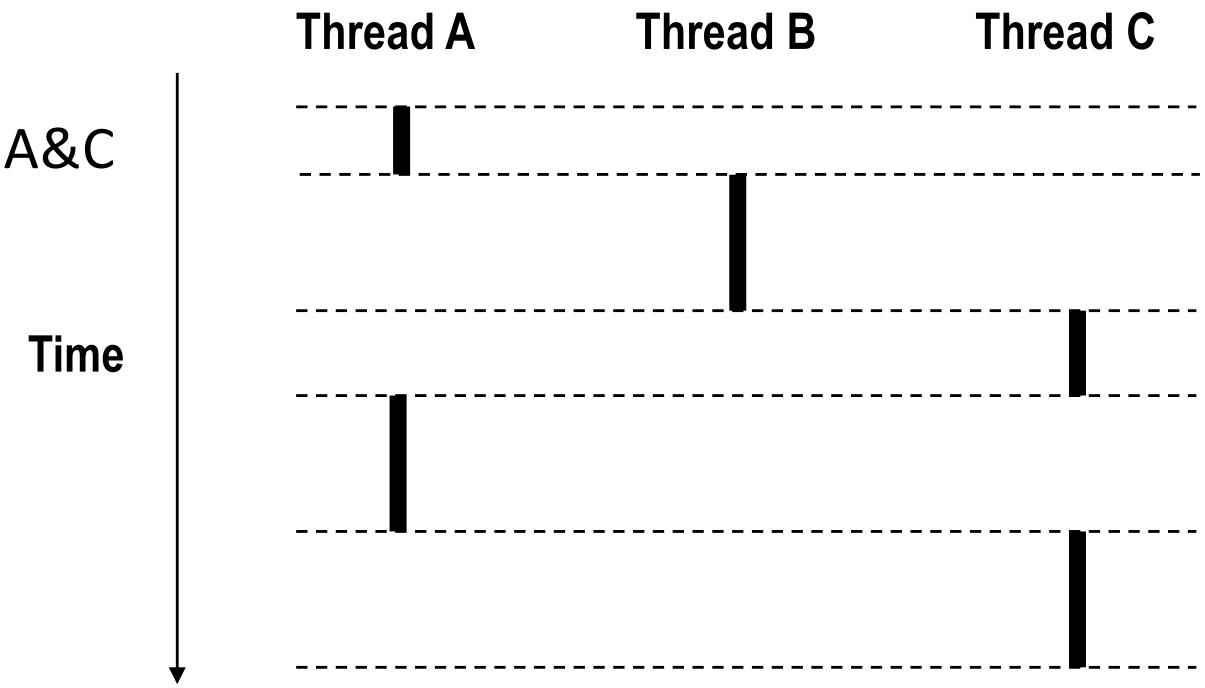




Concurrent Threads

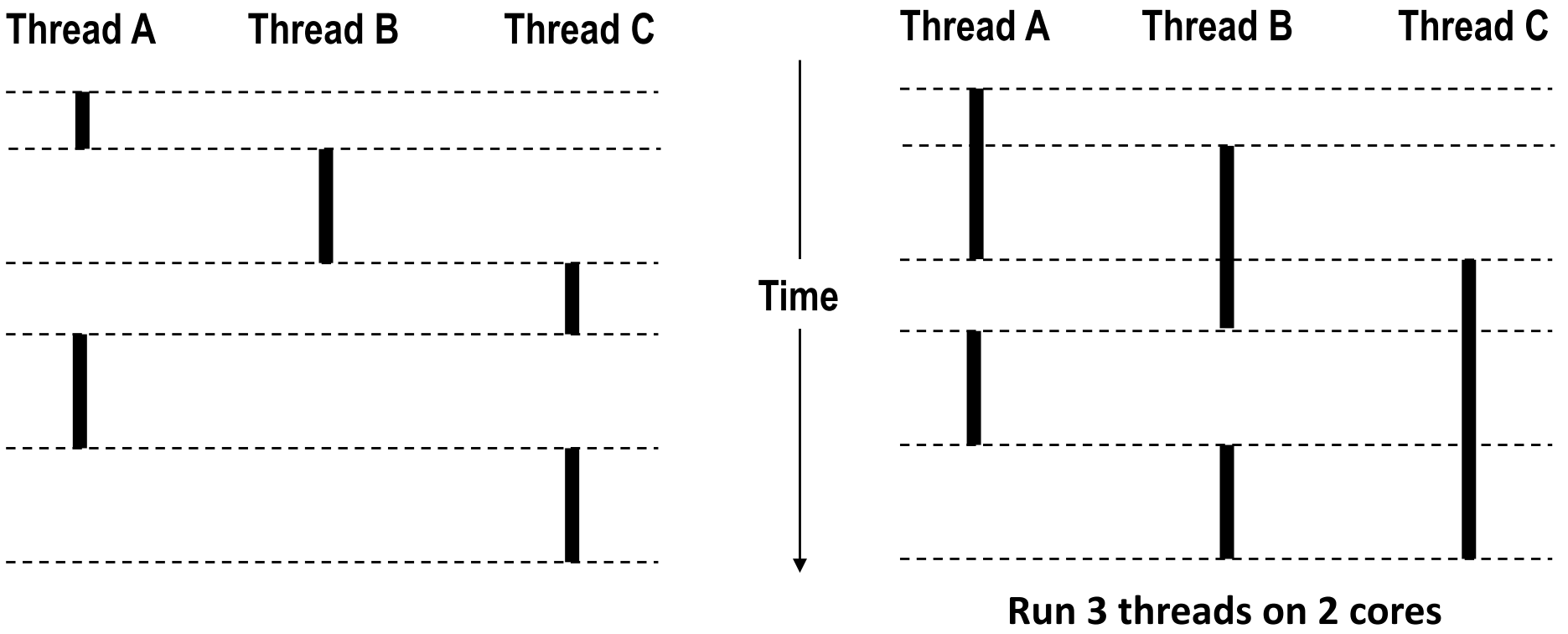
- **Two threads are *concurrent* if their flows overlap in time**
- **Otherwise, they are sequential**

- **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

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]
  - Synchronizing access to shared variables
    - `pthread_mutex_init`
    - `pthread_mutex_[un]lock`

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

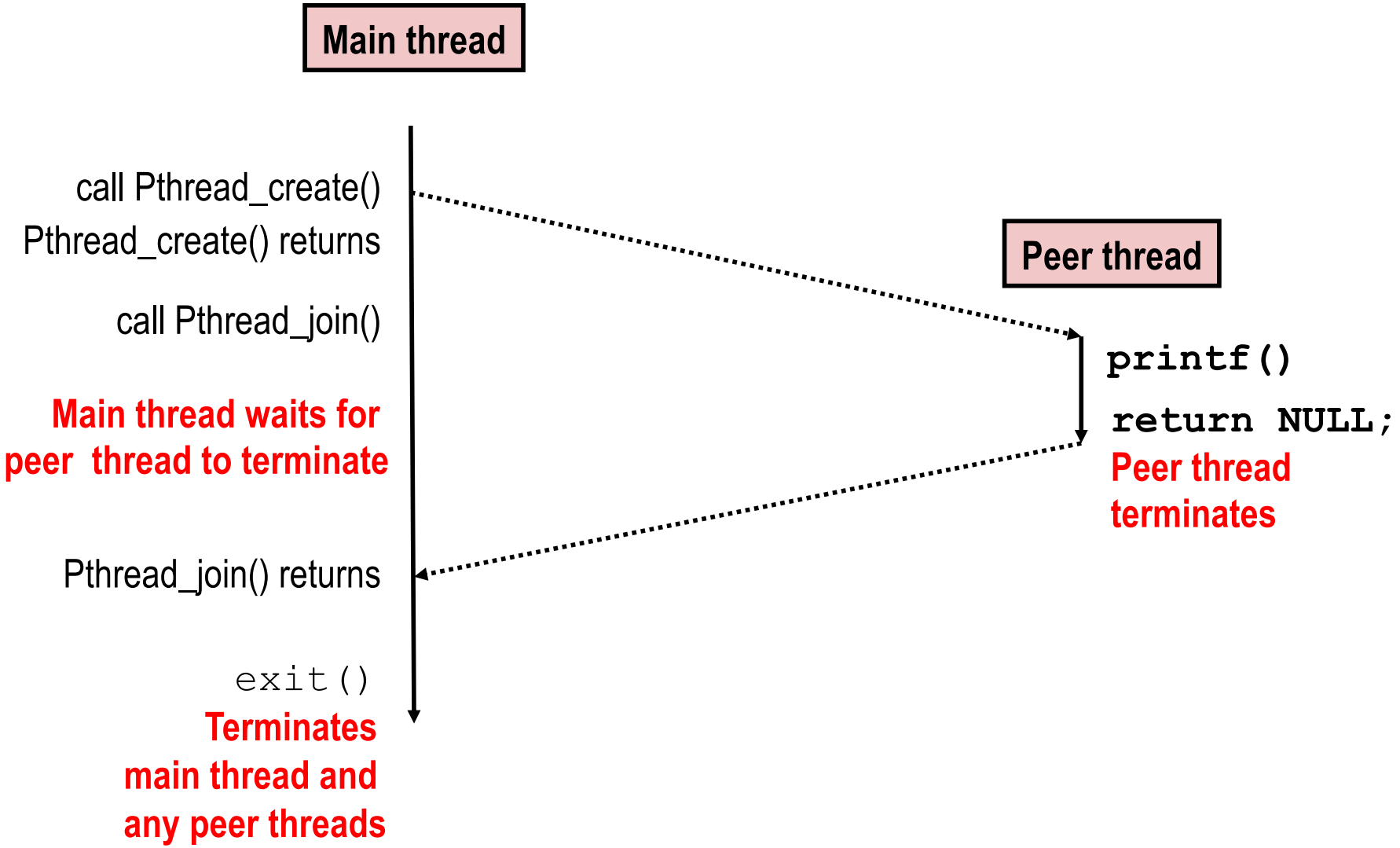
*Thread arguments (void *p)*

*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)

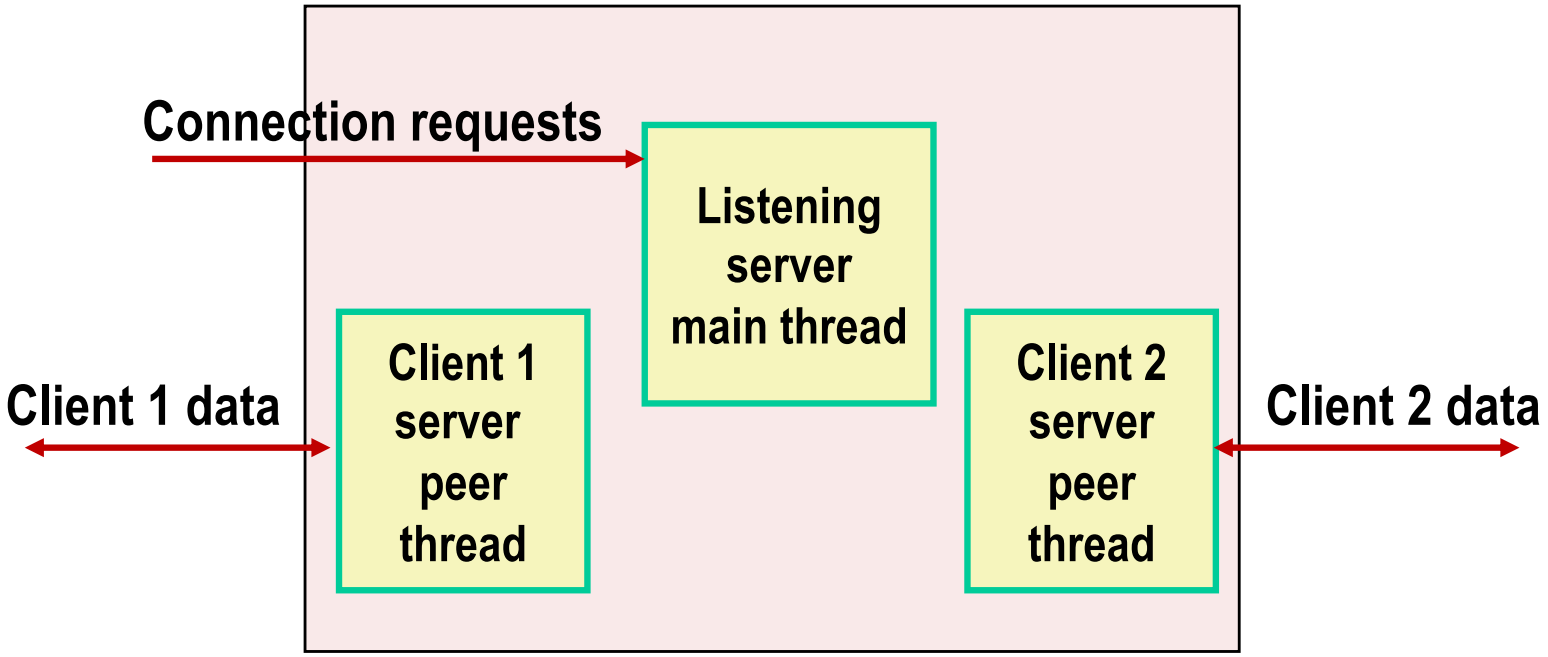
Thread-Based Concurrent Server (cont)

```
/* 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 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