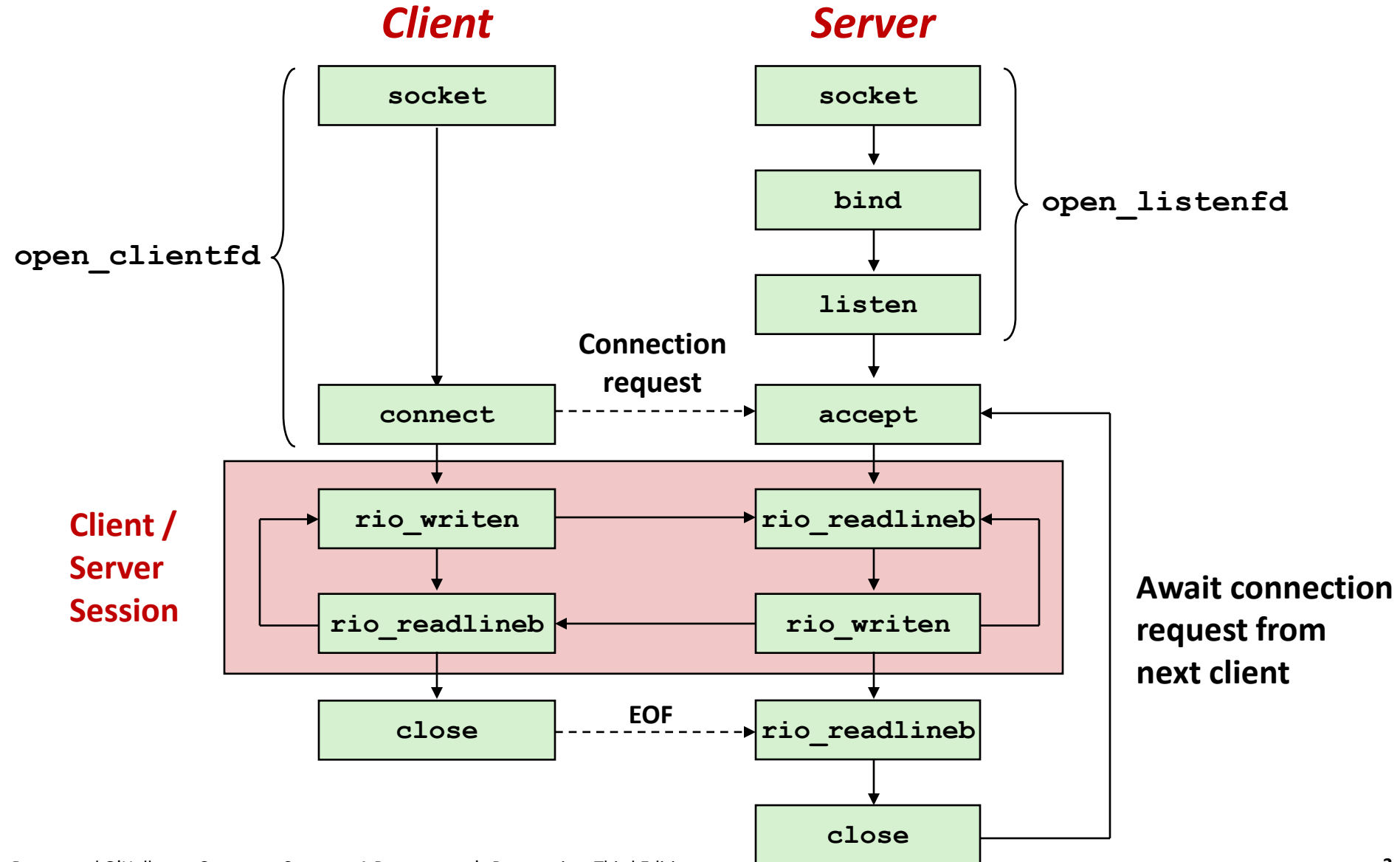




Synchronization: Advanced

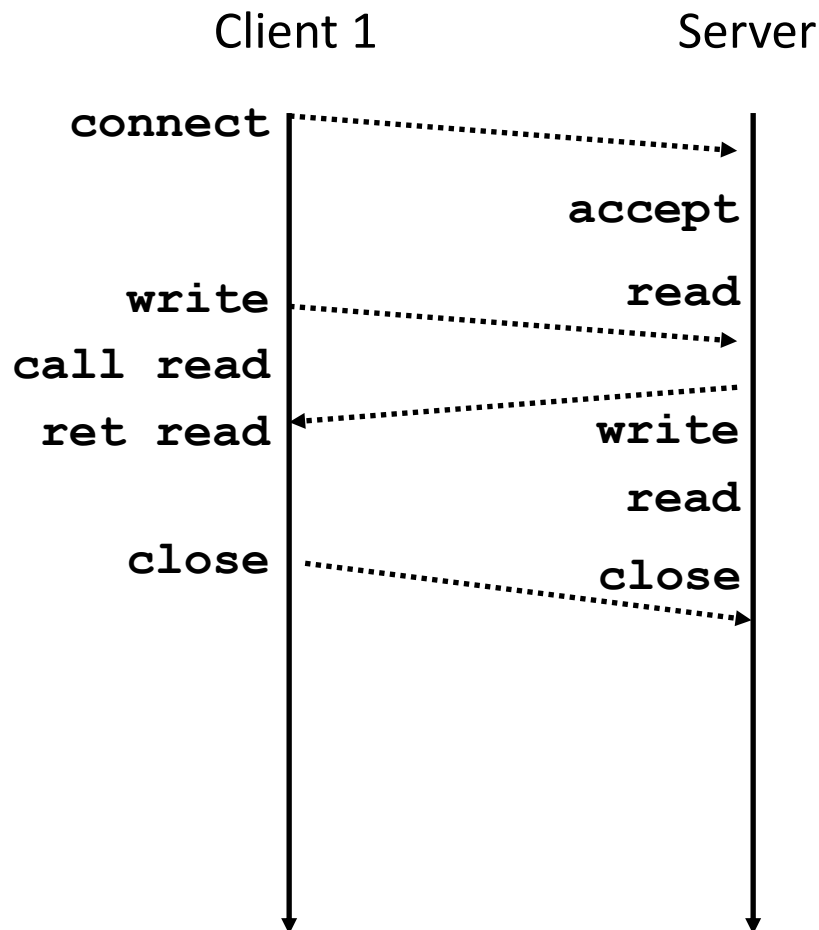
18-213/18-613: Introduction to Computer Systems
25th Lecture, August 1st, 2023

Reminder: Iterative Echo Server



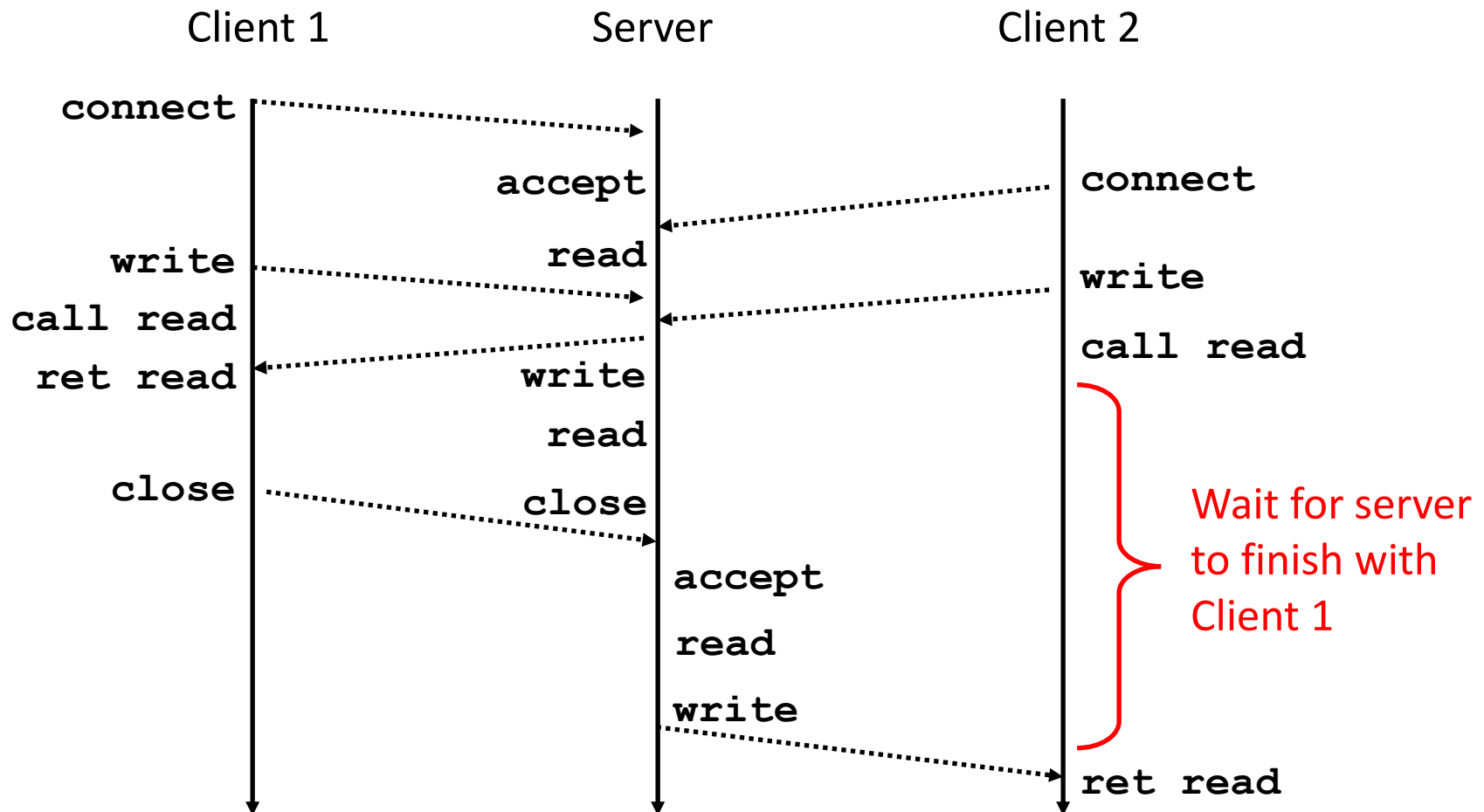
Iterative Servers

- Iterative servers process one connection at a time



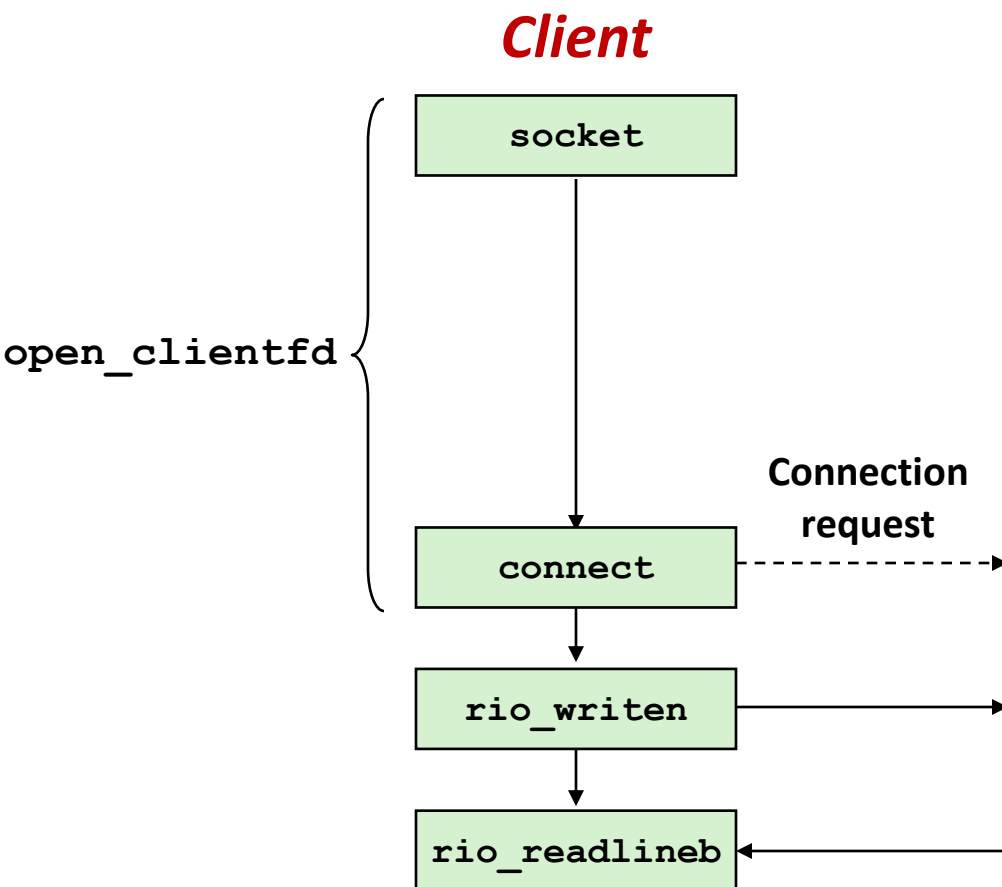
Iterative Servers

- Iterative servers process one request at a time



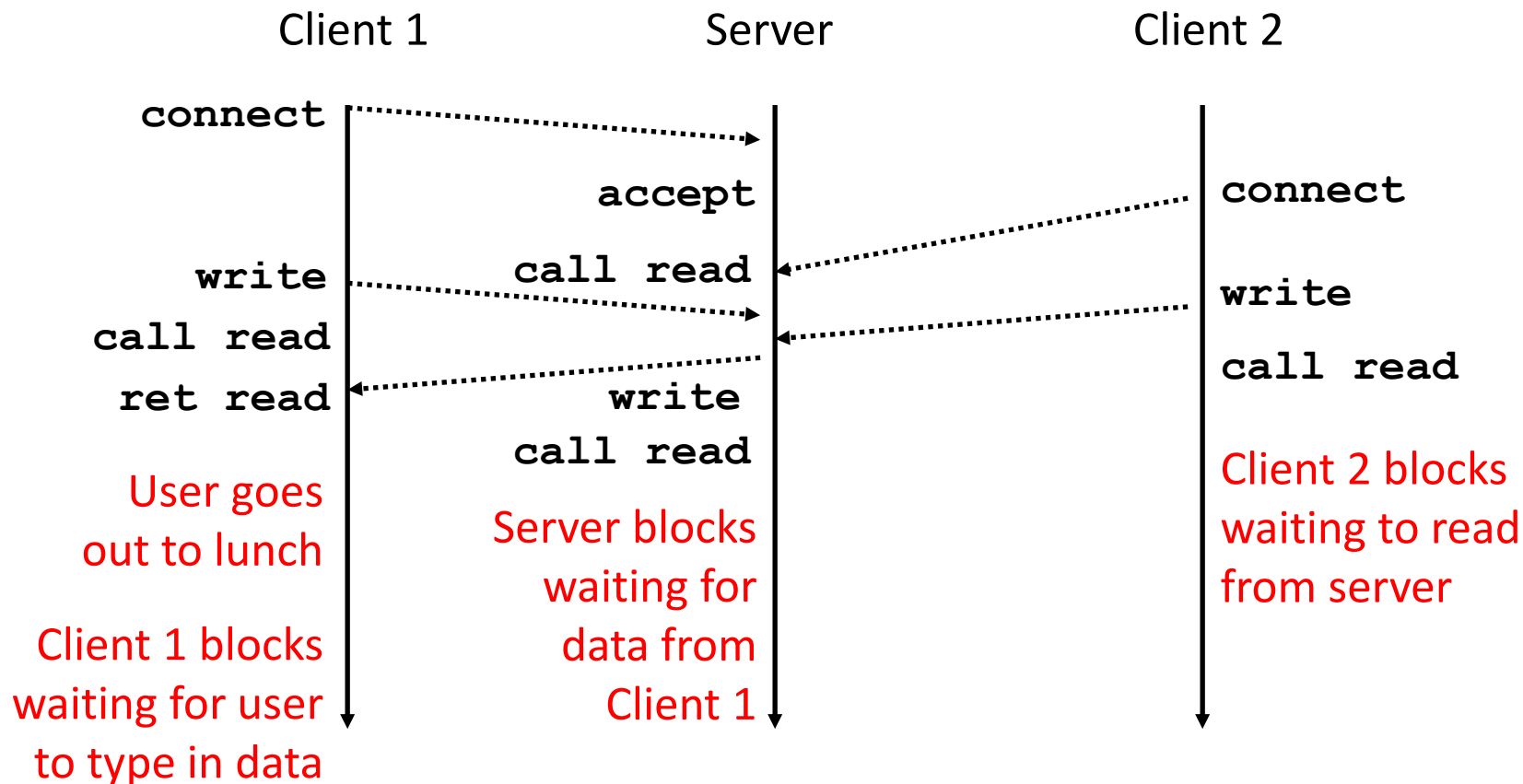
Where Does Second Client Block?

- 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

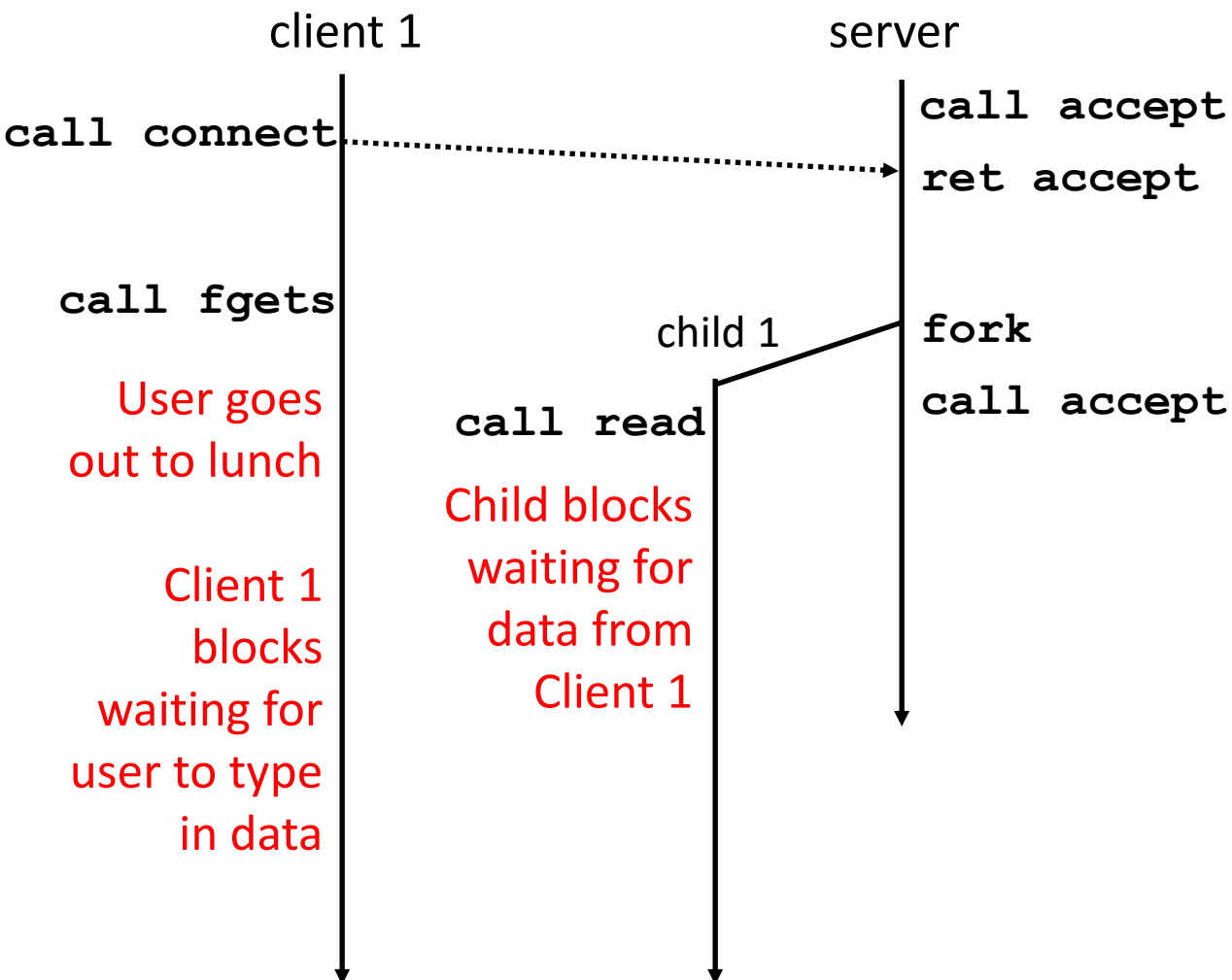
- Kernel automatically interleaves multiple logical flows
- Each flow shares the same address space
- Hybrid of process-based and event-based

Today

- **Process-based Servers** CSAPP 12.1
- **Event-based Servers** CSAPP 12.2
- **Thread-based Servers** CSAPP 12.3

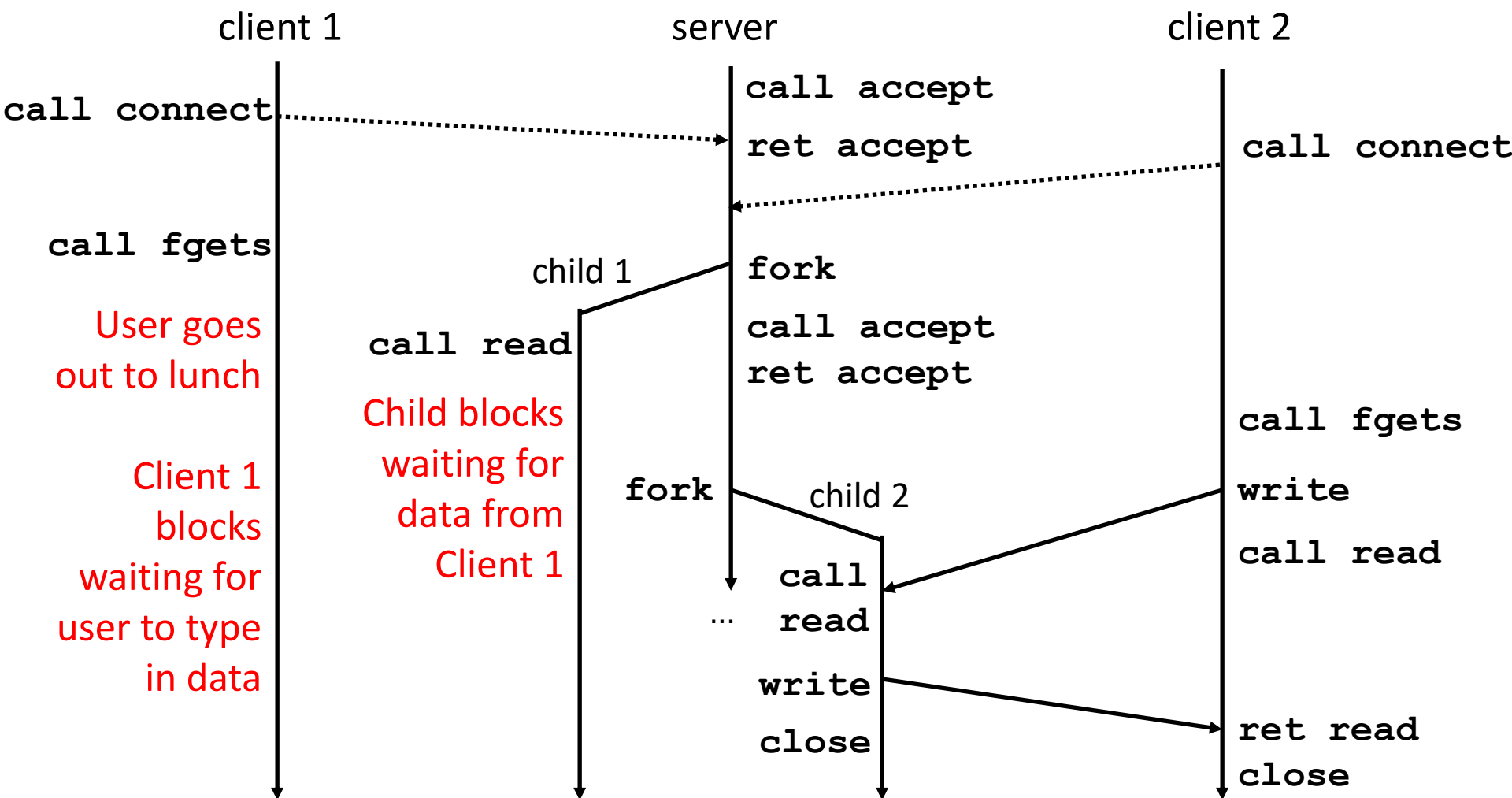
Approach #1: Process-based Servers

- 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)

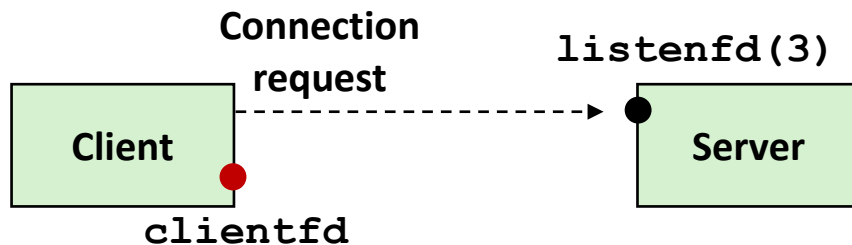
```
void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) > 0)
        ;
    return;
}
echoserverp.c
```

- Reap all zombie children

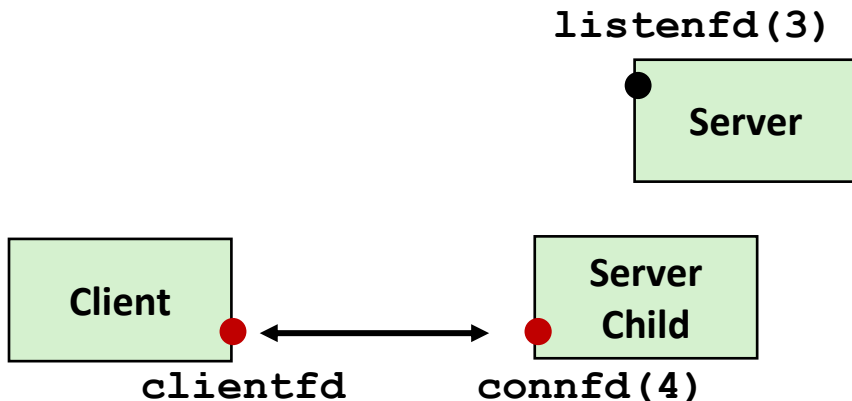
Concurrent Server: `accept` Illustrated



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

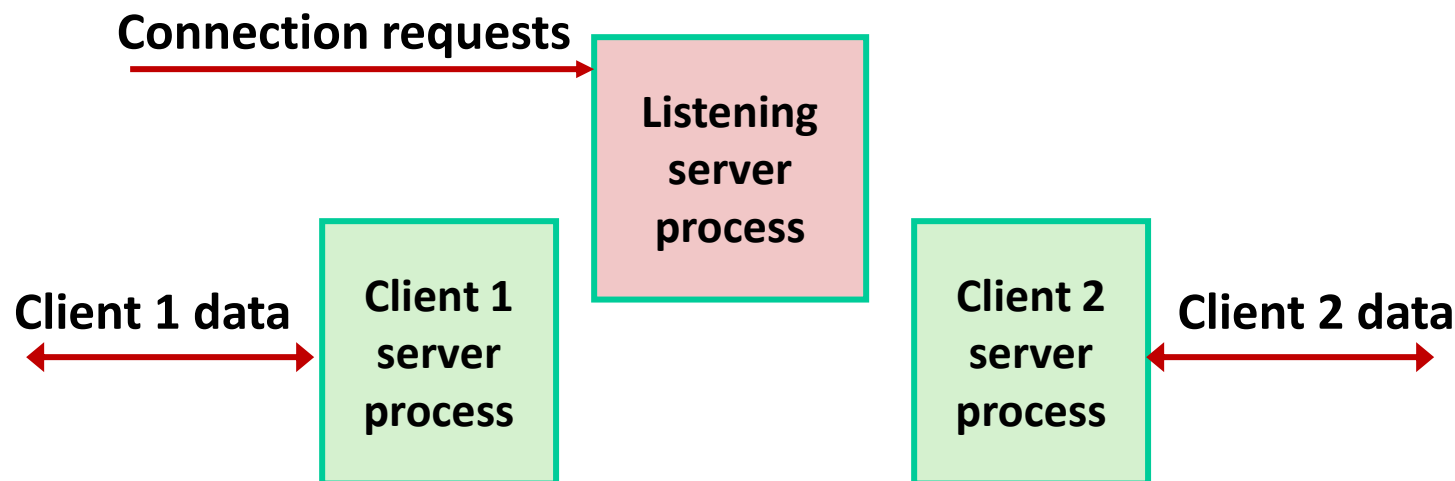


2. Client makes connection request by calling `connect`



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

- Threading Servers
- Process-based Servers CSAPP 12.1
- **Event-based Servers** CSAPP 12.2
- Thread-based Servers CSAPP 12.3

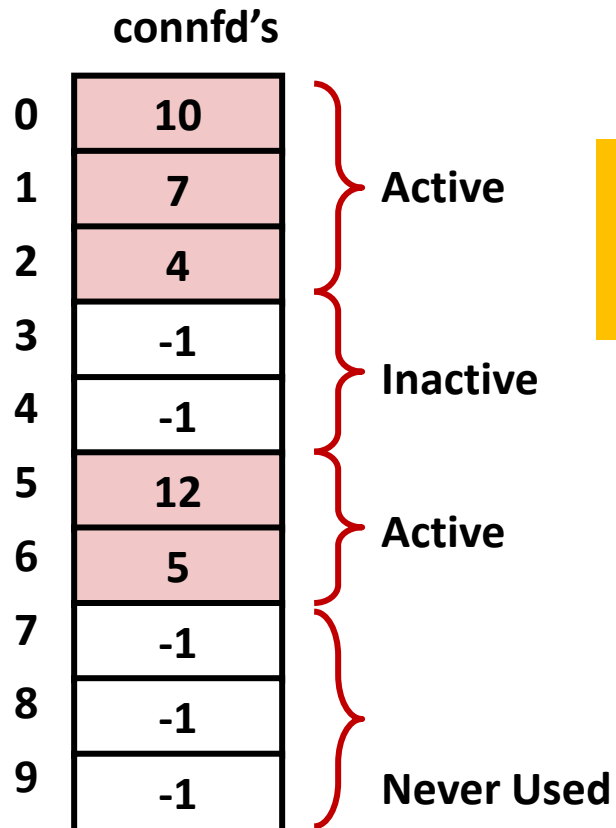
Approach #2: Event-based Servers

- **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



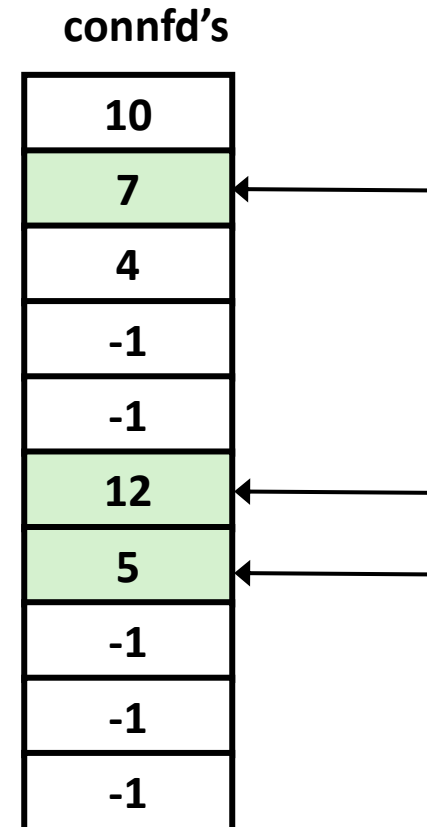
Anything
happened?

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

Today

- Threading Servers
- 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

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

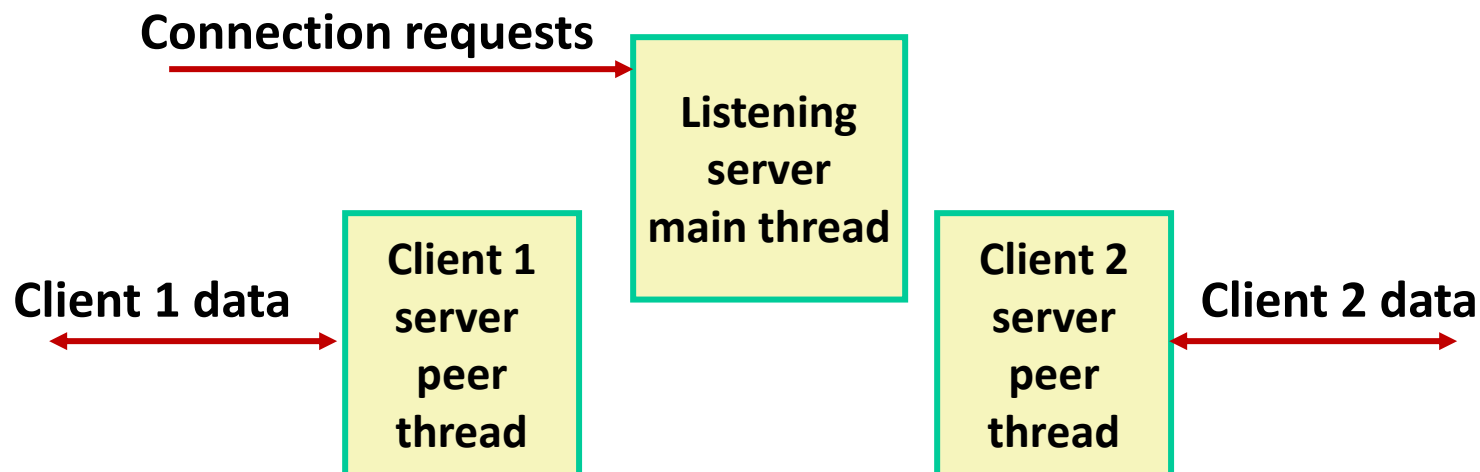
- Spawn new thread for each client
- Pass it copy of connection file descriptor
- Note use of **Malloc ()** ! [but not **Free ()**]

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



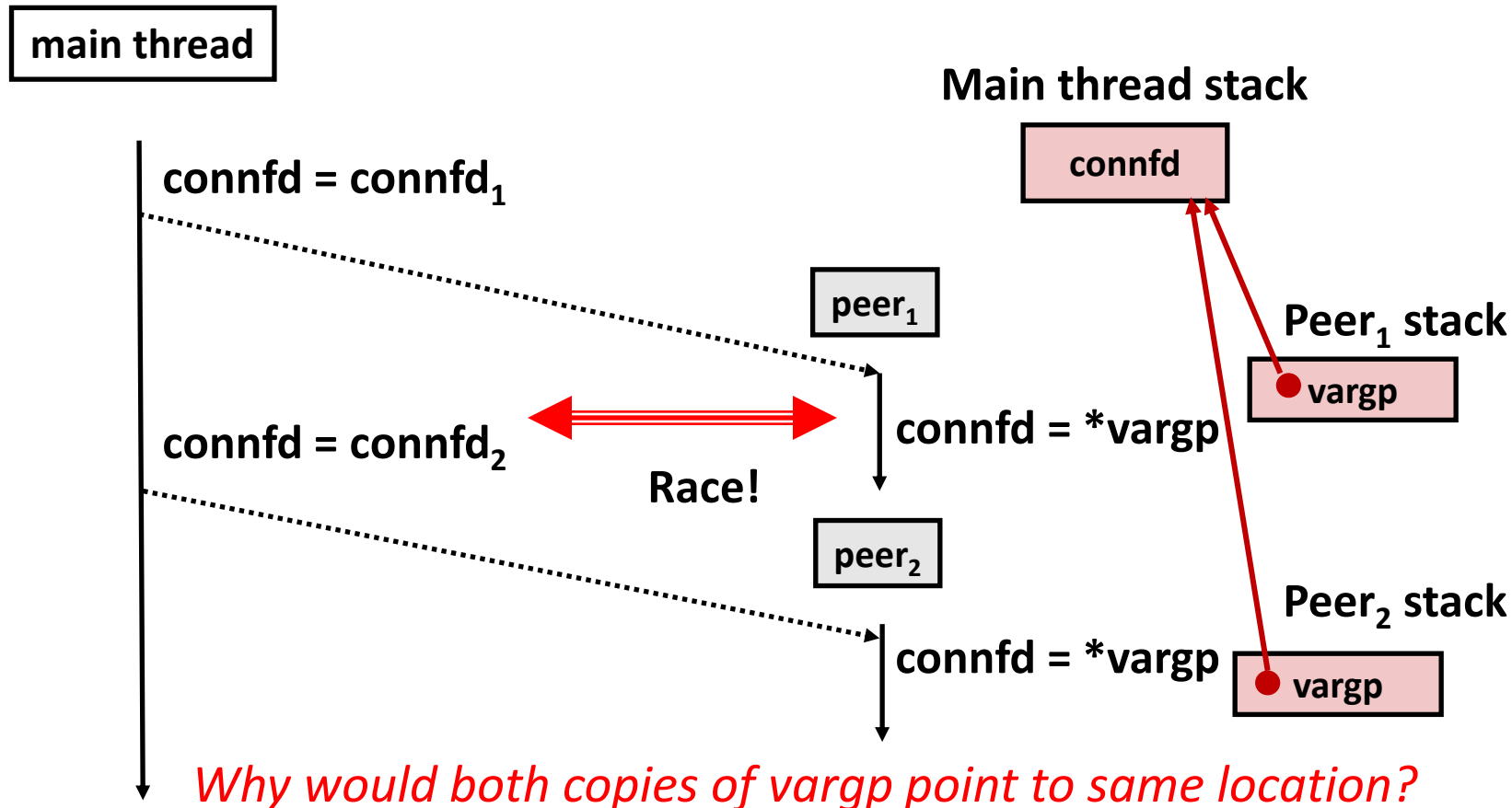
- Each client handled by individual peer thread
- Threads share all process state except TID
- Each thread has a separate stack for local variables

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);
}
```



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

Today

- Threads review
- Sharing
- **Mutual exclusion**
- Semaphores
- Producer-Consumer Synchronization

Enforcing Mutual Exclusion

- **Question:** How can we guarantee a safe trajectory?
- **Answer:** We must *synchronize* the execution of the threads so that they can never have an unsafe trajectory.
 - i.e., need to guarantee *mutually exclusive access* for each critical section.
- **Classic solution:**
 - Mutex (pthreads)
 - Semaphores (Edsger Dijkstra)
- **Other approaches (out of our scope)**
 - Condition variables (pthreads)
 - Monitors (Java)

MUTual EXclusion (mutex)

- ***Mutex***: boolean synchronization variable
- `enum {locked = 0, unlocked = 1}`
- **lock(m)**
 - If the mutex is currently not locked, lock it and return
 - Otherwise, wait (spinning, yielding, etc) and retry
- **unlock(m)**
 - Update the mutex state to unlocked

MUTual EXclusion (mutex)

- ***Mutex***: boolean synchronization variable *

- **Swap(*a, b)**

```
[t = *a; *a = b; return t;]
```

```
// Notation: what's inside the brackets [ ] is indivisible (a.k.a. atomic)
```

```
// by the magic of hardware / OS
```

- **Lock(m):**

```
while (swap(&m->state, locked) == locked) ;
```

- **Unlock(m):**

```
m->state = unlocked;
```

**For now. In reality, many other implementations and design choices (c.f., 15-410, 418, etc).*

badcnt.c: Improper Synchronization

```

/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
                  thread, &niters);
    Pthread_create(&tid2, NULL,
                  thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}

```

badcnt.c

```

/* Thread routine */
void *thread(void *vargp)
{
    long j, niters =
        *((long *)vargp);

    for (j = 0; j < niters; j++)
        cnt++;

    return NULL;
}

```

How can we fix this using synchronization?

goodmct.c: Mutex Synchronization

- Define and initialize a mutex for the shared variable `cnt`:

```
volatile long cnt = 0; /* Counter */
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL); // No special attributes
```

- Surround critical section with *lock* and *unlock*:

```
for (i = 0; i < niters; i++) {
    pthread_mutex_lock(&mutex);
    cnt++;
    pthread_mutex_unlock(&mutex);
}
```

```
linux> ./goodmct 10000
OK cnt=20000
linux> ./goodmct 10000
OK cnt=20000
```

Function	badcnt	goodmct
Time (ms) niters = 10 ⁶	12.0	214.0
Slowdown	1.0	17.8

Today

- Threads review
- Sharing
- Mutual exclusion
- **Semaphores**
- **Producer-Consumer Synchronization**

Semaphores

- **Semaphore:** non-negative global integer synchronization variable. Manipulated by *P* and *V* operations.
- **P(s)**
 - If *s* is nonzero, then decrement *s* by 1 and return immediately.
 - Test and decrement operations occur atomically (indivisibly)
 - If *s* is zero, then suspend thread until *s* becomes nonzero and the thread is restarted by a *V* operation.
 - After restarting, the *P* operation decrements *s* and returns control to the caller.
- **V(s):**
 - Increment *s* by 1.
 - Increment operation occurs atomically
 - If there are any threads blocked in a *P* operation waiting for *s* to become non-zero, then restart exactly one of those threads, which then completes its *P* operation by decrementing *s*.
- **Semaphore invariant: $s \geq 0$**

Semaphores

- ***Semaphore***: non-negative global integer synchronization variable
- **Manipulated by P and V operations:**
 - $P(s)$: [`while (s == 0) wait(); s--;`]
 - Dutch for “Proberen” (test)
 - $V(s)$: [`s++;`]
 - Dutch for “Verhogen” (increment)
- **OS kernel guarantees that operations between brackets [] are executed indivisibly/atomically**
 - Only one P or V operation at a time can modify s .
 - When `while` loop in P terminates, only that P can decrement s
- **Semaphore invariant: $s \geq 0$**

C Semaphore Operations

Pthreads functions:

```
#include <semaphore.h>

int sem_init(sem_t *s, 0, unsigned int val);} /* s = val */

int sem_wait(sem_t *s); /* P(s) */
int sem_post(sem_t *s); /* V(s) */
```

CS:APP wrapper functions:

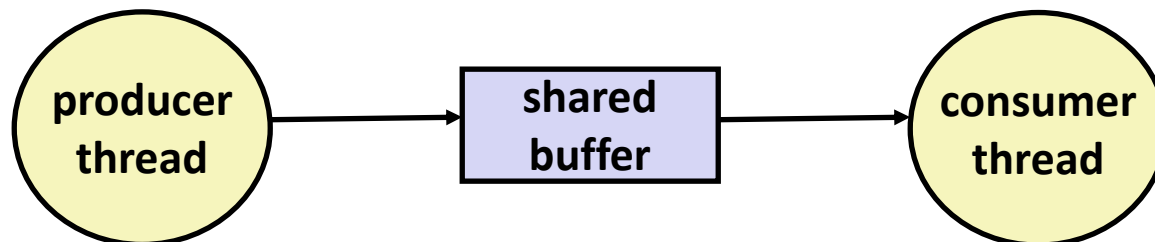
```
#include "csapp.h"

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```

Using Semaphores to Coordinate Access to Shared Resources

- **Basic idea: Thread uses a semaphore operation to notify another thread that some condition has become true**
 - Use counting semaphores to keep track of resource state.
 - Use binary semaphores to notify other threads.
- **The Producer-Consumer Problem**
 - Mediating interactions between processes that generate information and that then make use of that information

Producer-Consumer Problem



■ Common synchronization pattern:

- Producer waits for empty *slot*, inserts item in buffer, and notifies consumer
- Consumer waits for *item*, removes it from buffer, and notifies producer

■ Examples

- Multimedia processing:
 - Producer creates video frames, consumer renders them
- Event-driven graphical user interfaces
 - Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer
 - Consumer retrieves events from buffer and paints the display

Producer-Consumer on 1-element Buffer

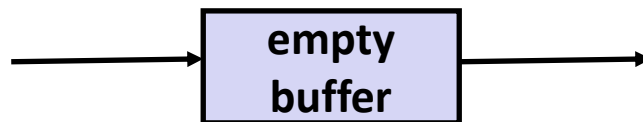
- Maintain two semaphores: `full` + `empty`

`full`

0

`empty`

1

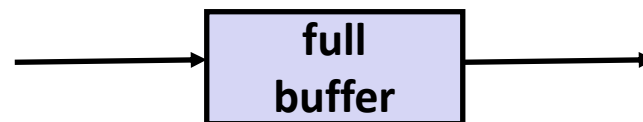


`full`

1

`empty`

0



Producer-Consumer on 1-element Buffer

```
#include "csapp.h"

#define NITERS 5

void *producer(void *arg);
void *consumer(void *arg);

struct {
    int buf; /* shared var */
    sem_t full; /* sems */
    sem_t empty;
} shared;
```

```
int main(int argc, char** argv) {
    pthread_t tid_producer;
    pthread_t tid_consumer;

    /* Initialize the semaphores */ Initial
    Sem_init(&shared.empty, 0, 1); value
    Sem_init(&shared.full, 0, 0);

    /* Create threads and wait */
    Pthread_create(&tid_producer, NULL,
                  producer, NULL);
    Pthread_create(&tid_consumer, NULL,
                  consumer, NULL);

    Pthread_join(tid_producer, NULL);
    Pthread_join(tid_consumer, NULL);

    return 0;
}
```

Producer-Consumer on 1-element Buffer

Initially: `empty==1, full==0`

Producer Thread

```
void *producer(void *arg) {
    int i, item;

    for (i=0; i<NITERS; i++) {
        /* Produce item */
        item = i;
        printf("produced %d\n",
            item);

        /* Write item to buf */
        P(&shared.empty);
        shared.buf = item;
        V(&shared.full);
    }
    return NULL;
}
```

Consumer Thread

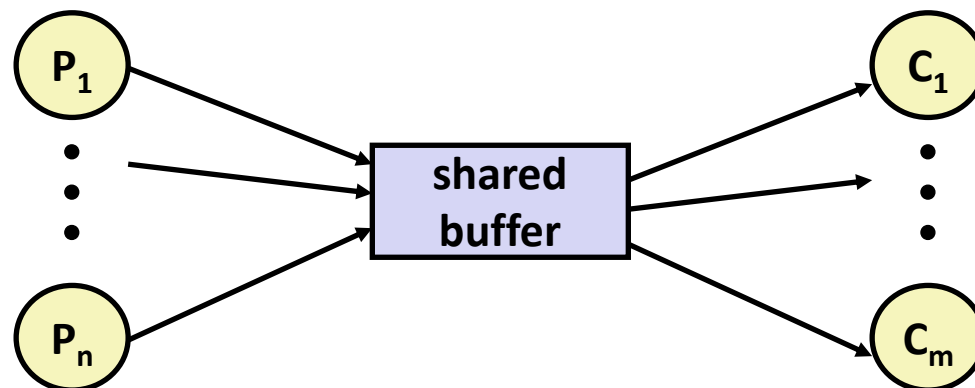
```
void *consumer(void *arg) {
    int i, item;

    for (i=0; i<NITERS; i++) {
        /* Read item from buf */
        P(&shared.full);
        item = shared.buf;
        V(&shared.empty);

        /* Consume item */
        printf("consumed %d\n", item);
    }
    return NULL;
}
```

Why 2 Semaphores for 1-Entry Buffer?

- Consider multiple producers & multiple consumers



- Producers will contend with each to get **empty**
- Consumers will contend with each other to get **full**

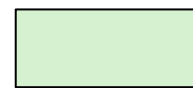
Producers

```
P(&shared.empty);
shared.buf = item;
V(&shared.full);
```

empty



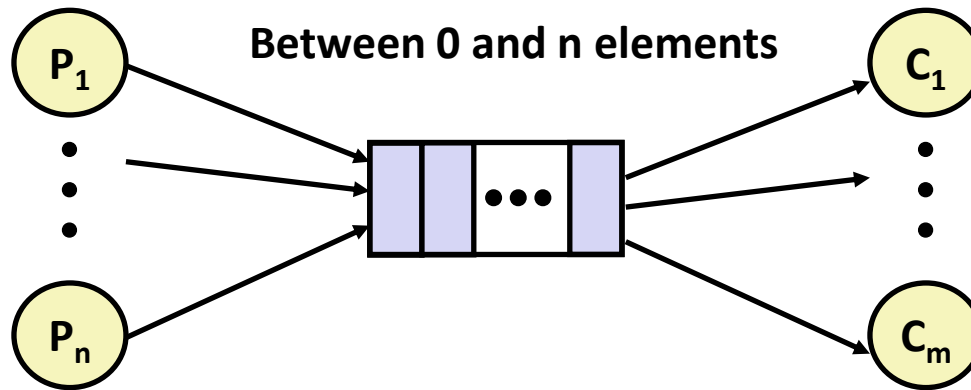
full



Consumers

```
P(&shared.full);
item = shared.buf;
V(&shared.empty);
```

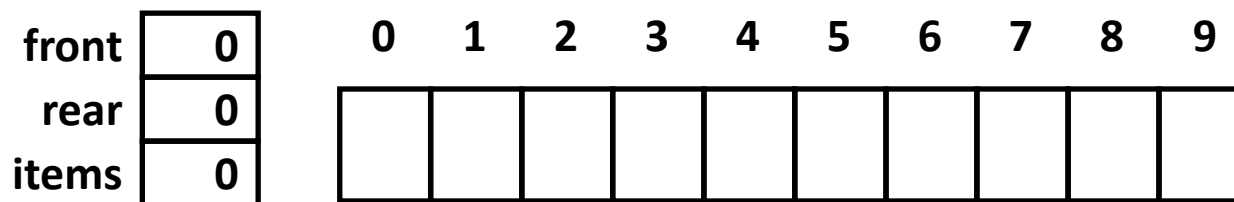
Producer-Consumer on an n -element Buffer



- Implemented using a shared buffer package called `sbuf`.

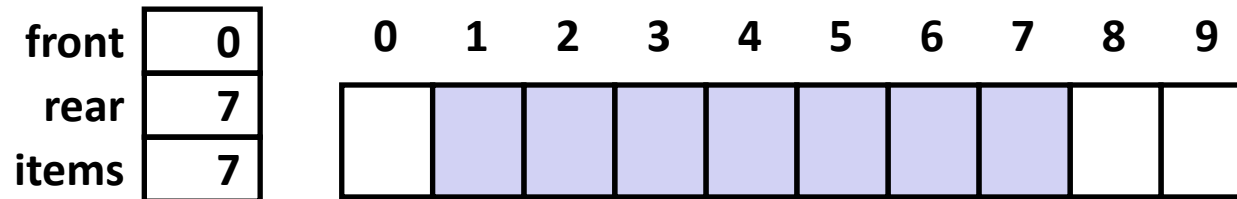
Circular Buffer (n = 10)

- Store elements in array of size n
- items: number of elements in buffer
- Empty buffer:
 - front = rear
- Nonempty buffer
 - rear: index of most recently inserted element
 - front: (index of next element to remove - 1) mod n
- Initially:

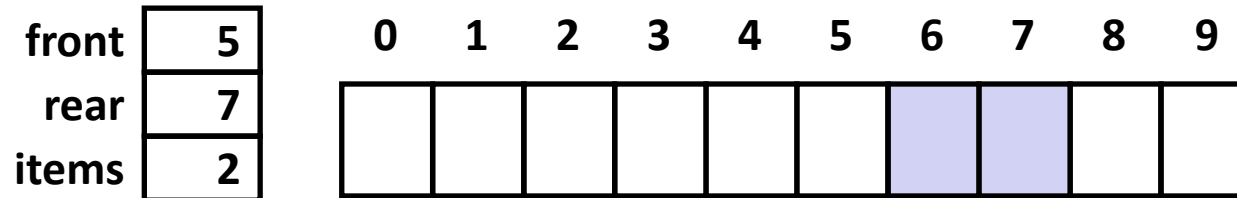


Circular Buffer Operation (n = 10)

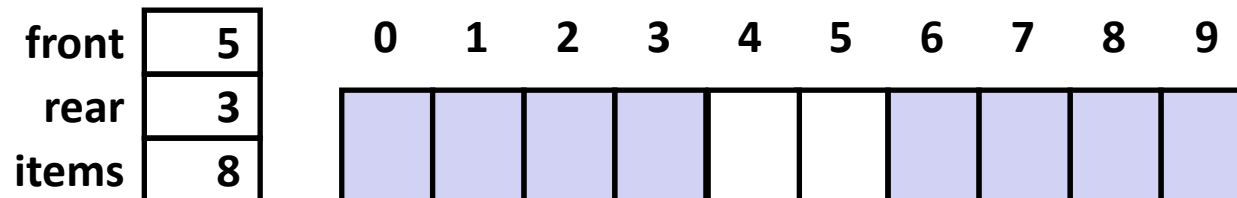
■ Insert 7 elements



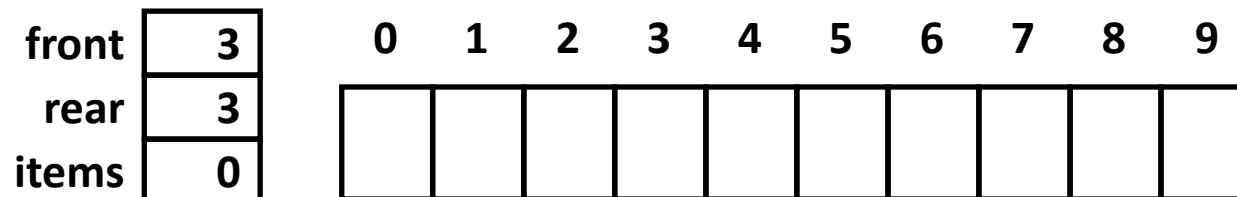
■ Remove 5 elements



■ Insert 6 elements



■ Remove 8 elements



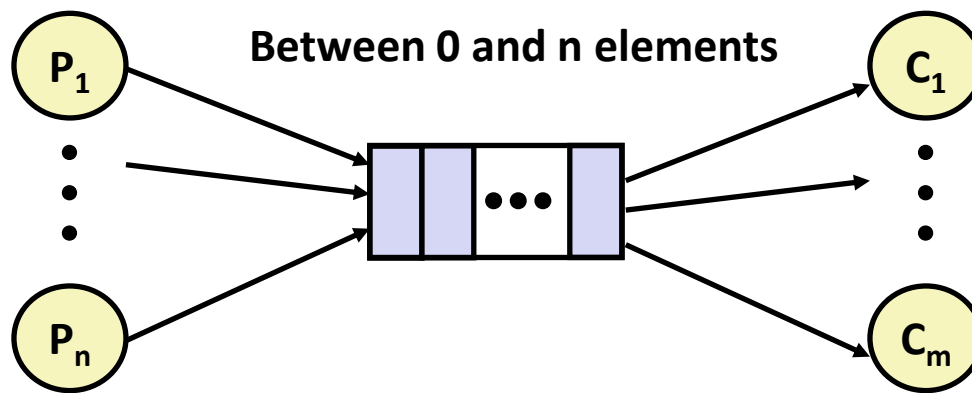
Sequential Circular Buffer Code

```
init(int v)
{
    items = front = rear = 0;
}

insert(int v)
{
    if (items >= n)
        error();
    if (++rear >= n) rear = 0;
    buf[rear] = v;
    items++;
}

int remove()
{
    if (items == 0)
        error();
    if (++front >= n) front = 0;
    int v = buf[front];
    items--;
    return v;
}
```

Producer-Consumer on an n -element Buffer



- **Requires a mutex and two counting semaphores:**
 - `mutex`: enforces mutually exclusive access to the buffer and counters
 - `slots`: counts the available slots in the buffer
 - `items`: counts the available items in the buffer
- **Makes use of general semaphores**
 - Will range in value from 0 to n

sbuf Package - Declarations

```
#include "csapp.h"

typedef struct {
    int *buf;          /* Buffer array */
    int n;            /* Maximum number of slots */
    int front;        /* buf[front+1 (mod n)] is first item */
    int rear;         /* buf[rear] is last item */
    sem_t mutex;      /* Protects accesses to buf */
    sem_t slots;      /* Counts available slots */
    sem_t items;      /* Counts available items */
} sbuf_t;

void sbuf_init(sbuf_t *sp, int n);
void sbuf_deinit(sbuf_t *sp);
void sbuf_insert(sbuf_t *sp, int item);
int sbuf_remove(sbuf_t *sp);
```

sbuf.h

sbuf Package - Implementation

Initializing and deinitializing a shared buffer:

```
/* Create an empty, bounded, shared FIFO buffer with n slots */
void sbuf_init(sbuf_t *sp, int n)
{
    sp->buf = Calloc(n, sizeof(int));
    sp->n = n; /* Buffer holds max of n items */
    sp->front = sp->rear = 0; /* Empty buffer iff front == rear */
    Sem_init(&sp->mutex, 0, 1); /* Binary semaphore for locking */
    Sem_init(&sp->slots, 0, n); /* Initially, buf has n empty slots */
    Sem_init(&sp->items, 0, 0); /* Initially, buf has zero items */
}

/* Clean up buffer sp */
void sbuf_deinit(sbuf_t *sp)
{
    Free(sp->buf);
}
```

sbuf.c

sbuf Package - Implementation

Inserting an item into a shared buffer:

```
/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->slots);          /* Wait for available slot */
    P(&sp->mutex);          /* Lock the buffer          */
    if (++sp->rear >= sp->n) /* Increment index (mod n) */
        sp->rear = 0;
    sp->buf[sp->rear] = item; /* Insert the item          */
    V(&sp->mutex);          /* Unlock the buffer        */
    V(&sp->items);          /* Announce available item */
}
```

sbuf.c

sbuf Package - Implementation

Removing an item from a shared buffer:

```
/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);          /* Wait for available item */
    P(&sp->mutex);          /* Lock the buffer */
    if (++sp->front >= sp->n) /* Increment index (mod n) */
        sp->front = 0;
    item = sp->buf[sp->front]; /* Remove the item */
    V(&sp->mutex);          /* Unlock the buffer */
    V(&sp->slots);          /* Announce available slot */
    return item;
}
```

sbuf.c

Demonstration

- **See program produce-consume.c in code directory**
- **10-entry shared circular buffer**
- **5 producers**
 - Agent i generates numbers from $20*i$ to $20*i - 1$.
 - Puts them in buffer
- **5 consumers**
 - Each retrieves 20 elements from buffer
- **Main program**
 - Makes sure each value between 0 and 99 retrieved once

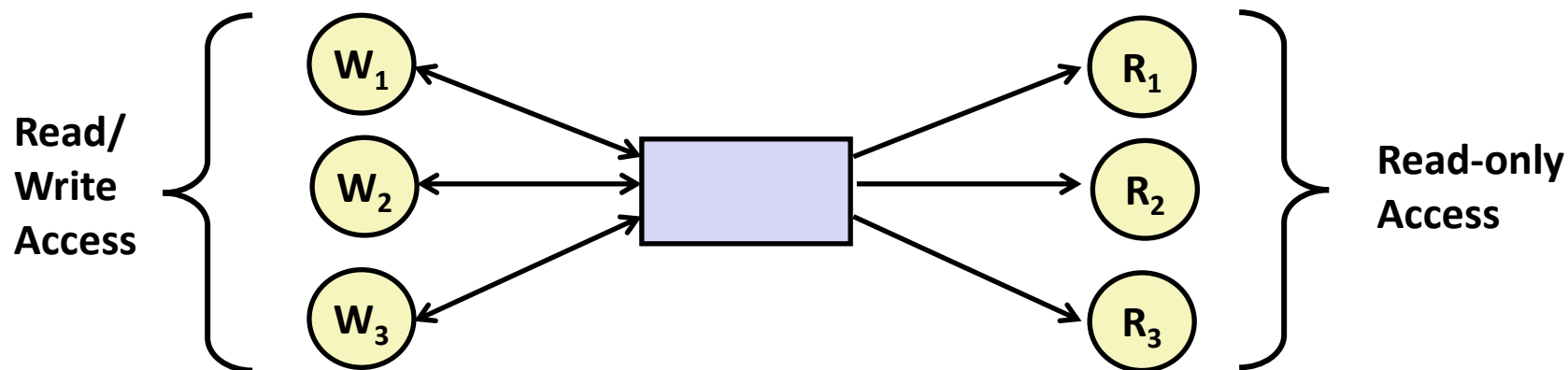
Summary

- **Programmers need a clear model of how variables are shared by threads.**
- **Variables shared by multiple threads must be protected to ensure mutually exclusive access**
 - E.g., using mutex lock and unlock, semaphore P and V
- **Semaphores are a fundamental mechanism for enforcing mutual exclusion**
 - And can also support producer-consumer synchronization

Today

- **Using semaphores to schedule shared resources** **CSAPP 12.5.4**
 - Readers-writers problem
- **Other concurrency issues** **CSAPP 12.7**
 - Races
 - Deadlocks
 - Thread safety
 - Interactions between threads and signal handling

Readers-Writers Problem



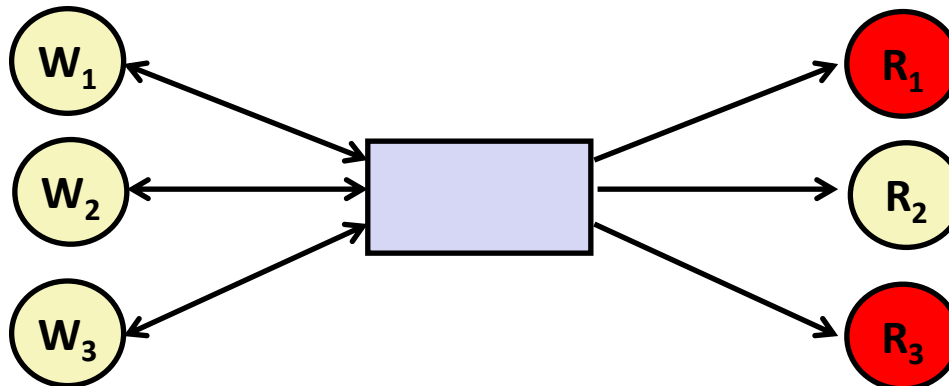
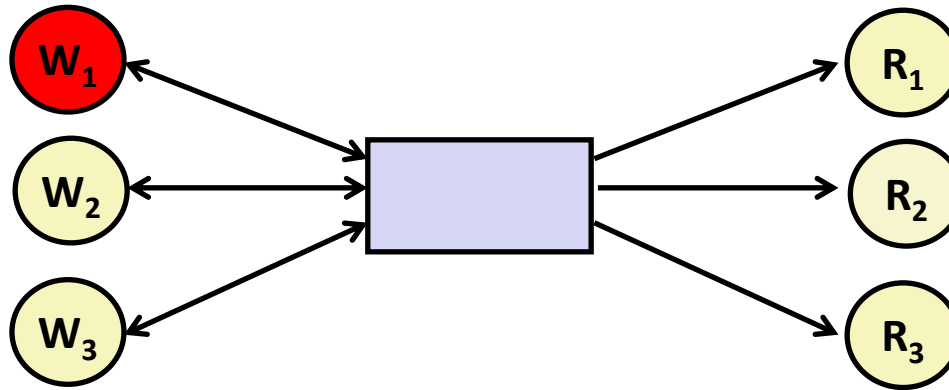
■ Problem statement:

- *Reader* threads only read the object
- *Writer* threads modify the object (read/write access)
- Writers must have exclusive access to the object
- Unlimited number of readers can access the object concurrently

■ Occurs frequently in real systems, e.g.,

- Online airline reservation system
- Multithreaded caching Web proxy

Readers/Writers Examples



Variants of Readers-Writers

- ***First readers-writers problem (favors readers)***
 - No reader should be kept waiting unless a writer has already been granted permission to use the object.
 - A reader that arrives after a waiting writer gets priority over the writer.

- ***Second readers-writers problem (favors writers)***
 - Once a writer is ready to write, it performs its write as soon as possible
 - A reader that arrives after a writer must wait, even if the writer is also waiting.

- ***Starvation (where a thread waits indefinitely) is possible in both cases.***

Solution to First Readers-Writers Problem

Readers:

```

int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);

        /* Reading happens here */

        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}

```

Writers:

```

void writer(void)
{
    while (1) {
        P(&w);

        /* Writing here */

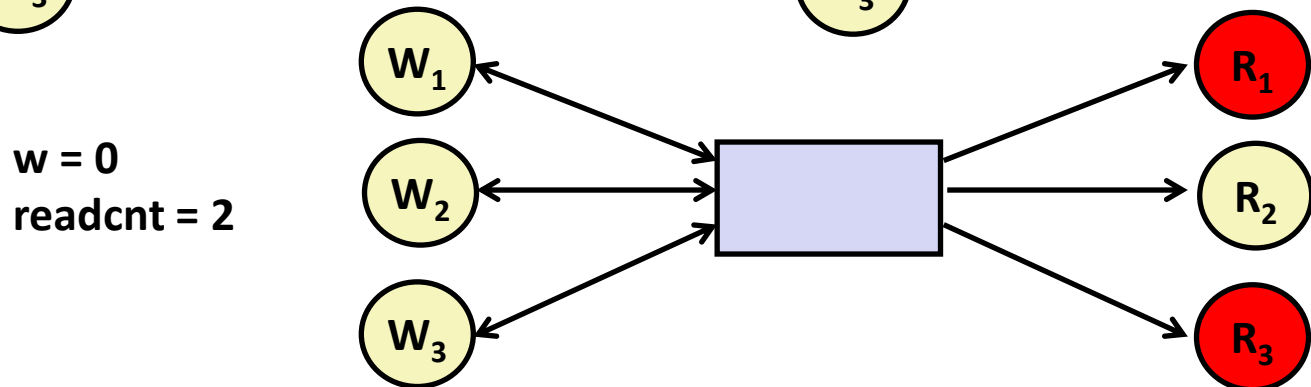
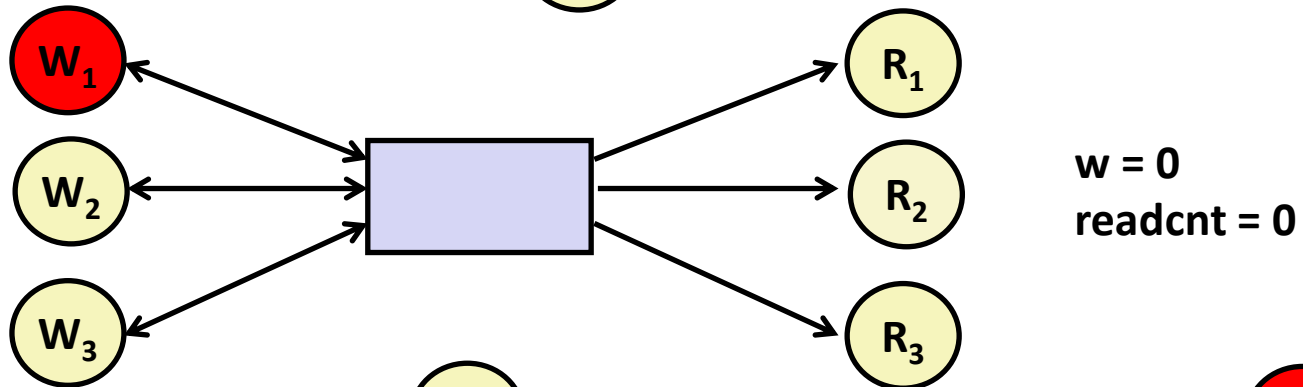
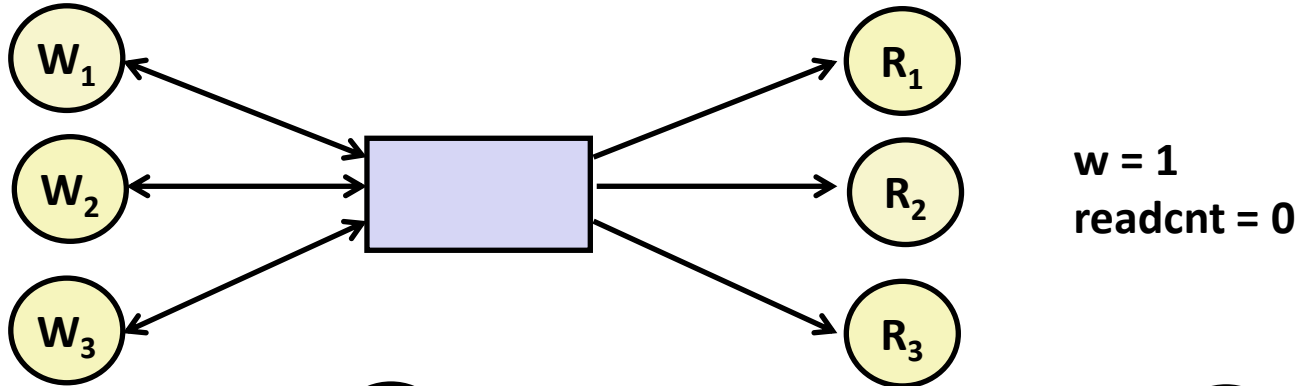
        V(&w);
    }
}

```

rw1.c

**A reader that arrives
after a waiting writer
gets priority over the writer**

Readers/Writers Examples



Solution to First Readers-Writers Problem

Readers:

```

int readcnt;      /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);

        /* Reading happens here */

        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}

```

Writers:

```

void writer(void)
{
    while (1) {
        P(&w);

        /* Writing here */

        V(&w);
    }
}

```

rw1.c

Arrivals: R1 R2 W1 R3

Solution to First Readers-Writers Problem

Readers:

```

int readcnt;      /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);

        /* Reading happens here */

        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}

```

R1 →

Writers:

```

void writer(void)
{
    while (1) {
        P(&w);

        /* Writing here */

        V(&w);
    }
}

```

rw1.c

Arrivals: R1 R2 W1 R3

readcnt == 1
w == 0

Solution to First Readers-Writers Problem

Readers:

```

int readcnt;      /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
R2 → if (readcnt == 1) /* First in */
        P(&w);
        V(&mutex);

R1 → /* Reading happens here */

        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}

```

Writers:

```

void writer(void)
{
    while (1) {
        P(&w);

        /* Writing here */

        V(&w);
    }
}

```

rw1.c

Arrivals: R1 R2 W1 R3

readcnt == 2
w == 0

Solution to First Readers-Writers Problem

Readers:

```

int readcnt;      /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);

        /* Reading happens here */

        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}

```

R2
R1



Writers:

```

void writer(void)
{
    while (1) {
        P(&w); ← W1

        /* Writing here */

        V(&w);
    }
}

```

rw1.c

Arrivals: R1 R2 W1 R3

readcnt == 2
w == 0

Solution to First Readers-Writers Problem

Readers:

```

int readcnt;      /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);

        /* Reading happens here */

        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}

```

Writers:

```

void writer(void)
{
    while (1) {
        P(&w); ← W1

        /* Writing here */

        V(&w);
    }
}

```

rw1.c

Arrivals: R1 R2 W1 R3

readcnt == 1
w == 0

Solution to First Readers-Writers Problem

Readers:

```

int readcnt;      /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        R3 → if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);

        /* Reading happens here */

        R2 → P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);

        R1 → }
    }
}

```

Writers:

```

void writer(void)
{
    while (1) { ← W1
        P(&w);

        /* Writing here */

        V(&w);
    }
}

```

rw1.c

Arrivals: R1 R2 W1 R3

readcnt == 2
w == 0

Solution to First Readers-Writers Problem

Readers:

```

int readcnt;      /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);

        /* Reading happens here */

        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}

```

Writers:

```

void writer(void)
{
    while (1) {
        P(&w); ← W1

        /* Writing here */

        V(&w);
    }
}

```

rw1.c

Arrivals: R1 R2 W1 R3

readcnt == 1
w == 0

Solution to First Readers-Writers Problem

Readers:

```

int readcnt;      /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);

        /* Reading happens here */

        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}

```

R3



Writers:

```

void writer(void)
{
    while (1) {
        P(&w);
        /* Writing here */
        V(&w);
    }
}

```

rw1.c

Arrivals: R1 R2 W1 R3

readcnt == 0

w == 1

Other Versions of Readers-Writers

■ Shortcoming of first solution

- Continuous stream of readers will block writers indefinitely

■ Second version

- Once writer comes along, blocks access to later readers
- Series of writes could block all reads

■ FIFO implementation

- See rwqueue code in code directory
- Service requests in order received
- Threads kept in FIFO
- Each has semaphore that enables its access to critical section

Solution to Second Readers-Writers Problem

```

int readcnt, writecnt;           // Initially 0
sem_t rmutex, wmutex, r, w;    // Initially 1
void reader(void)
{
    while (1) {
        P(&r);
        P(&rmutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&rmutex);
        V(&r)

        /* Reading happens here */

        P(&rmutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&rmutex);
    }
}

```

```

void writer(void)
{
    while (1) {
        P(&wmutex);
        writecnt++;
        if (writecnt == 1)
            P(&r);
        V(&wmutex);

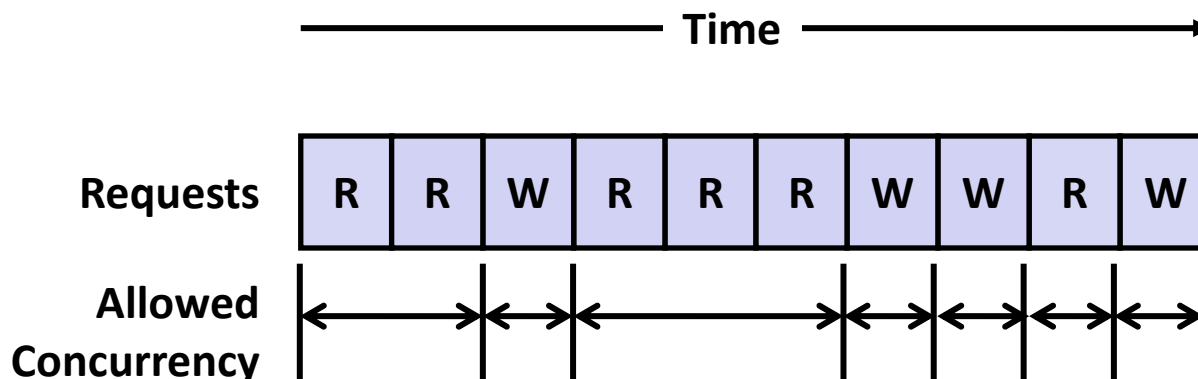
        P(&w);
        /* Writing here */
        V(&w);

        P(&wmutex);
        writecnt--;
        if (writecnt == 0);
            V(&r);
        V(&wmutex);
    }
}

```

A reader that arrives after a writer must wait, even if the writer is also waiting

Managing Readers/Writers with FIFO



■ Idea

- Read & Write requests are inserted into FIFO
- Requests handled as remove from FIFO
 - Read allowed to proceed if currently idle or processing read
 - Write allowed to proceed only when idle
- Requests inform controller when they have completed

■ Fairness

- Guarantee every request is eventually handled

Readers Writers FIFO Implementation

■ Full code in rwqueue.{h,c}

```
/* Queue data structure */
typedef struct {
    sem_t mutex;           // Mutual exclusion
    int reading_count;     // Number of active readers
    int writing_count;     // Number of active writers
    // FIFO queue implemented as linked list with tail
    rw_token_t *head;
    rw_token_t *tail;
} rw_queue_t;
```

```
/* Represents individual thread's position in queue */
typedef struct TOK {
    bool is_reader;
    sem_t enable;         // Enables access
    struct TOK *next;    // Allows chaining as linked list
} rw_token_t;
```


Readers Writers FIFO Use

■ In rwqueue-test.c

```
/* Get write access to data and write */
void iwriter(int *buf, int v)
{
    rw_token_t tok;
    rw_queue_request_write(&q, &tok);
    /* Critical section */
    *buf = v;
    /* End of Critical Section */
    rw_queue_release(&q);
}
```

```
/* Get read access to data and read */
int ireader(int *buf)
{
    rw_token_t tok;
    rw_queue_request_read(&q, &tok);
    /* Critical section */
    int v = *buf;
    /* End of Critical section */
    rw_queue_release(&q);
    return v;
}
```

Library Reader/Writer Lock

- Data type `pthread_rwlock_t`

- Operations

- Acquire read lock

```
pthread_rwlock_rdlock(pthread_rwlock_t *rwlock)
```

- Acquire write lock

```
pthread_rwlock_wrlock(pthread_rwlock_t *rwlock)
```

- Release (either) lock

```
pthread_rwlock_unlock(pthread_rwlock_t *rwlock)
```

- Observation

- Library must be used correctly!
 - Up to programmer to decide what requires read access and what requires write access