Semaphores

Questions answered in this lecture:

Review: How to implement join with condition variables?

Review: How to implement producer/consumer with condition variables?

What is the difference between **semaphores** and condition variables?

How to implement a **lock** with semaphores?

How to implement semaphores with locks and condition variables?

How to implement **join** and producer/consumer with semaphores?

How to implement **reader/writer locks** with semaphores?

Summary: rules of thumb for CVs

- Keep state in addition to CV's
- Always do wait/signal with lock held
- Whenever thread wakes from waiting, recheck state

Condition Variables vs Semaphores

- **Condition variables have no state (other than waiting queue)**
  - Programmer must track additional state

- **Semaphores have state: track integer value**
  - State cannot be directly accessed by user program, but state determines behavior of semaphore operations

Semaphore Operations

- **Allocate and Initialize**

```
sem_t sem;
sem_init(sem_t *s, int initval) {
  s->value = initval;
}
```

  - User cannot read or write value directly after initialization
- **Wait or Test (sometime P() for Dutch word)**
  - Waits until value of sem is > 0, then decrements sem value
- **Signal or Increment or Post (sometime V() for Dutch)**
  - Increment sem value, then wake a single waiter

wait and post are atomic

Join with CV vs Semaphores

CVs:

```
void thread_join() {
    Mutex_lock(&m);// w
    if (done == 0) // x
      Cond_wait(&c, &m); // y
    Mutex_unlock(&m); // z
}
```

```
void thread_exit() {
    Mutex_lock(&m); // a
    done = 1;       // b
    Cond_signal(&c);// c
    Mutex_unlock(&m);// d
}
```

Semaphores:

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

sem_t s;
sem_init(&s, ???);

Initialize to 0 (so sem_wait() must wait…)

```
void thread_join() {
    sem_wait(&s);
}
```

```
void thread_exit() {
    sem_post(&s)
}
```

Equivalence Claim

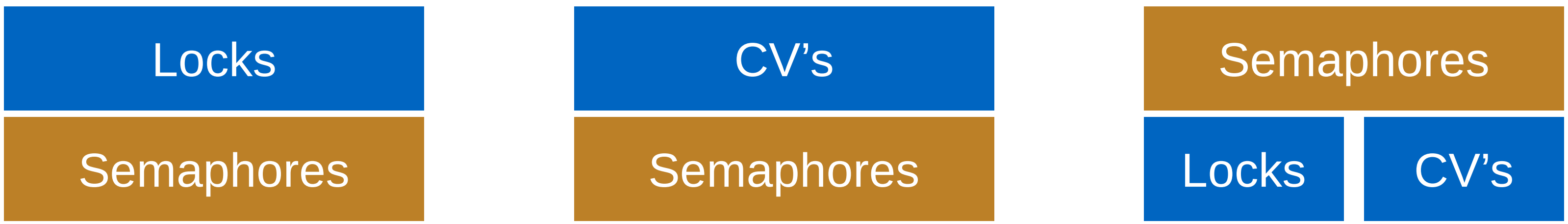
- Semaphores are equally powerful to Locks+CVs
  - what does this mean?

- One might be more convenient, but that's not relevant

- Equivalence means each can be built from the other

Proof Steps

- **Want to show we can do these three things:**



Build Lock from Semaphore

```
typedef struct __lock_t {
// whatever data structs you need go here
} lock_t;

void init(lock_t *lock) {
}

void acquire(lock_t *lock) {
}

void release(lock_t *lock) {
}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Locks

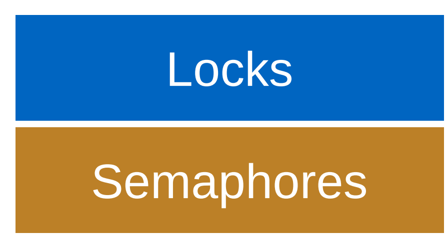
Semaphores

Build Lock from Semaphore

```
typedef struct __lock_t {
        sem_t sem;
} lock_t;

void init(lock_t *lock) {
        sem_init(&lock->sem, ??);
}
void acquire(lock_t *lock) {
        sem_wait(&lock->sem);
}
void release(lock_t *lock) {
        sem_post(&lock->sem);
}
```

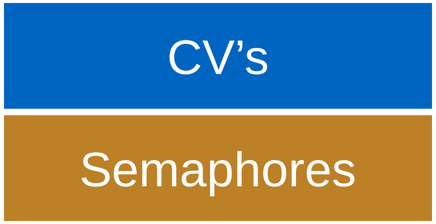
1 → 1 thread can grab lock

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter



Building CV's over Semaphores

- Possible, but really hard to do right



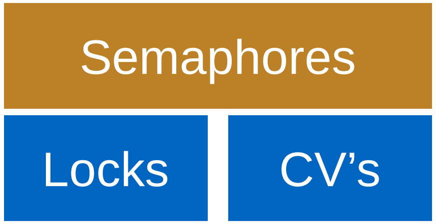
- Read about Microsoft Research's attempts:

http://research.microsoft.com/pubs/64242/ImplementingCVs.pdf

Build Semaphore from Lock and CV

```
Typedef struct {
        // what goes here?


} sem_t;

Void sem_init(sem_t *s, int value) {
        // what goes here?


}
```

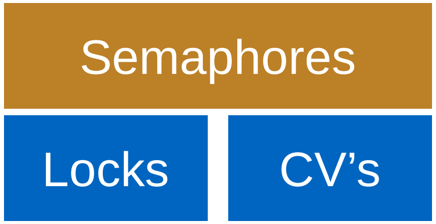
Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter



Build Semaphore from Lock and CV

```
Typedef struct {
        int value;
        cond_t cond;
        lock_t lock;
} sem_t;

Void sem_init(sem_t *s, int value) {
        s->value = value;
        cond_init(&s->cond);
        lock_init(&s->lock);
}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

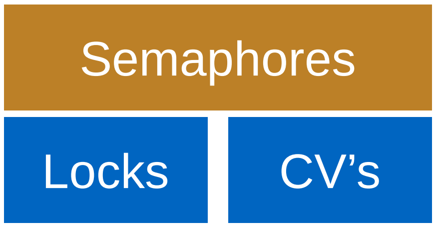


Build Semaphore from Lock and CV

```
Sem_wait{sem_t *s) {
        // what goes here?

}
```

```
Sem_post{sem_t *s) {
        // what goes here?

}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

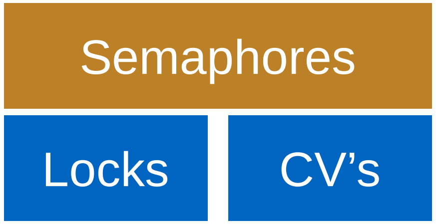


Build Semaphore from Lock and CV

```
Sem_wait{sem_t *s) {
        lock_acquire(&s->lock);
        // this stuff is atomic


        lock_release(&s->lock);
}
```

```
Sem_post{sem_t *s) {
        lock_acquire(&s->lock);
        // this stuff is atomic

        lock_release(&s->lock);
}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

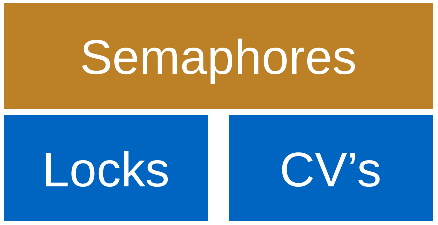


Build Semaphore from Lock and CV

```
Sem_wait{sem_t *s) {
        lock_acquire(&s->lock);
        while (s->value <= 0)
                cond_wait(&s->cond);
        s->value--;
        lock_release(&s->lock);
}
```

```
Sem_post{sem_t *s) {
        lock_acquire(&s->lock);
        // this stuff is atomic

        lock_release(&s->lock);
}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

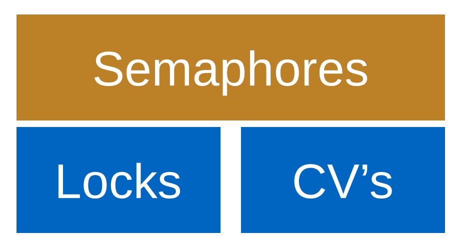


Build Semaphore from Lock and CV

```
Sem_wait{sem_t *s) {
        lock_acquire(&s->lock);
        while (s->value <= 0)
                cond_wait(&s->cond);
        s->value--;
        lock_release(&s->lock);
}
```

```
Sem_post{sem_t *s) {
        lock_acquire(&s->lock);
        s->value++;
        cond_signal(&s->cond);
        lock_release(&s->lock);
}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

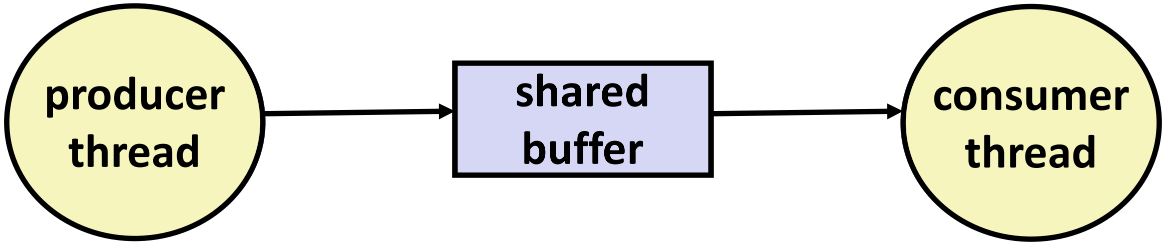


Using Semaphores to Coordinate Access to Shared Resources

- **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.

- **Two classic examples:**
  - The Producer-Consumer Problem
  - The Readers-Writers Problem

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: Semaphores #1

- **Simplest case:**
  - Single producer thread, single consumer thread
  - Single shared buffer with one element between producer and consumer
- **Requirements**
  - Consumer must wait for producer to fill buffer
  - Producer must wait for consumer to empty buffer (if filled)
- **Requires 2 semaphores**
  - emptyBuffer: Initialize to ??? **1 → 1 empty buffer; producer can run 1 time first**
  - fullBuffer: Initialize to ??? **0 → 0 full buffers; consumer can run 0 times first**

Producer

```
While (1) {
        sem_wait(&emptyBuffer);
        Fill(&buffer);
        sem_signal(&fullBuffer);
}
```

Consumer

```
While (1) {
        sem_wait(&fullBuffer);
        Use(&buffer);
        sem_signal(&emptyBuffer);
}
```

Producer/Consumer: Semaphores #2

- **Next case: Circular Buffer**
  - Single producer thread, single consumer thread
  - Shared buffer with **N** elements between producer and consumer
- **Requires 2 semaphores**
  - emptyBuffer: Initialize to ??? **N → N empty buffers; producer can run N times first**
  - fullBuffer: Initialize to ??? **0 → 0 full buffers; consumer can run 0 times first**

Producer

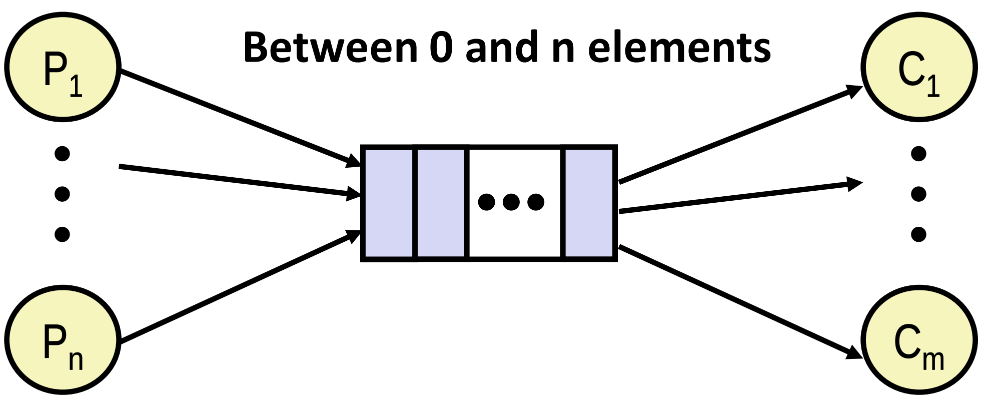
```
i = 0;
While (1) {
      sem_wait(&emptyBuffer);
      Fill(&buffer[i]);
      i = (i+1)%N;
      sem_signal(&fullBuffer);
}
```

Consumer

```
j = 0;
While (1) {
      sem_wait(&fullBuffer);
      Use(&buffer[j]);
      j = (j+1)%N;
      sem_signal(&emptyBuffer);
}
```

Producer/Consumer: Semaphore #3

- **Final case:**
  - **Multiple producer threads, multiple consumer threads**
  - Shared buffer with N elements between producer and consumer
- **Requirements**
  - Each consumer must grab unique filled element
  - Each producer must grab unique empty element
  - **Why will previous code (shown below) not work???**



Producer/Consumer: Semaphore #3

- **Final case:**
  - **Multiple producer threads, multiple consumer threads**
  - Shared buffer with N elements between producer and consumer
- **Requirements**
  - Each consumer must grab unique filled element
  - Each producer must grab unique empty element
  - **Why will previous code (shown below) not work???**

Producer

```
i = 0;
While (1) {
        sem_wait(&emptyBuffer);
        Fill(&buffer[i]);
        i = (i+1)%N;
        sem_signal(&fullBuffer);
}
```

Consumer

```
j = 0;
While (1) {
        sem_wait(&fullBuffer);
        Use(&buffer[j]);
        j = (j+1)%N;
        sem_signal(&emptyBuffer);
}
```

Are i and j private or shared? Need each producer to grab unique buffer

Producer/Consumer: Multiple Threads

- **Final case:**
  - **Multiple producer threads, multiple consumer threads**
  - Shared buffer with N elements between producer and consumer
- **Requirements**
  - Each consumer must grab unique filled element
  - Each producer must grab unique empty element

Producer

```
While (1) {
      sem_wait(&emptyBuffer);
      myi = findempty(&buffer);
      Fill(&buffer[myi]);
      sem_signal(&fullBuffer);
}
```

Consumer

```
While (1) {
      sem_wait(&fullBuffer);
      myj = findfull(&buffer);
      Use(&buffer[myj]);
      sem_signal(&emptyBuffer);
}
```

Are myi and myj private or shared? Where is mutual exclusion needed???

Producer/Consumer: Multiple Threads

- Consider three possible locations for mutual exclusion
- Which work??? Which is best???

Producer #1

```
sem_wait(&mutex);
sem_wait(&emptyBuffer);
myi = findempty(&buffer);
Fill(&buffer[myi]);
sem_signal(&fullBuffer);
sem_signal(&mutex);
```

Consumer #1

```
sem_wait(&mutex);
sem_wait(&fullBuffer);
myj = findfull(&buffer);
Use(&buffer[myj]);
sem_signal(&emptyBuffer);
sem_signal(&mutex);
```

Problem?

Deadlock at mutex (e.g., consumer runs first; won't release mutex)

Producer/Consumer: Multiple Threads

- Consider three possible locations for mutual exclusion
- Which work??? Which is best???

Producer #2

```
sem_wait(&emptyBuffer);
sem_wait(&mutex);
myi = findempty(&buffer);
Fill(&buffer[myi]);
sem_signal(&mutex);
sem_signal(&fullBuffer);
```

Consumer #2

```
sem_wait(&fullBuffer);
sem_wait(&mutex);
myj = findfull(&buffer);
Use(&buffer[myj]);
sem_signal(&mutex);
sem_signal(&emptyBuffer);
```

Works, but limits concurrency:
Only 1 thread at a time can be using or filling different buffers

Producer/Consumer: Multiple Threads

- Consider three possible locations for mutual exclusion
- Which work??? Which is best???

Producer #3

```
sem_wait(&emptyBuffer);
sem_wait(&mutex);
myi = findempty(&buffer);
sem_signal(&mutex);
Fill(&buffer[myi]);
sem_signal(&fullBuffer);
```

Consumer #3

```
sem_wait(&fullBuffer);
sem_wait(&mutex);
myj = findfull(&buffer);
sem_signal(&mutex);
Use(&buffer[myj]);
sem_signal(&emptyBuffer);
```

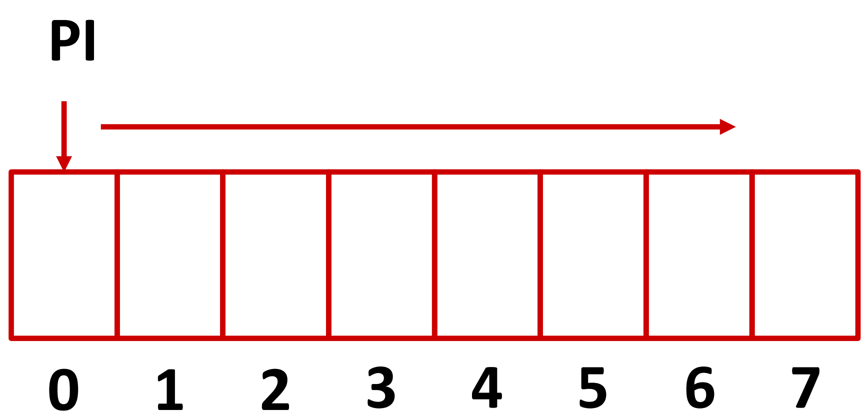
Works and increases concurrency; only finding a buffer is protected by mutex; Filling or Using different buffers can proceed concurrently

Producer/Consumer: Multiple Threads

- **How to implement with multiple producer and multiple consumers?**

Producer A:
0 <- findempty

An index PI allocate availble slots

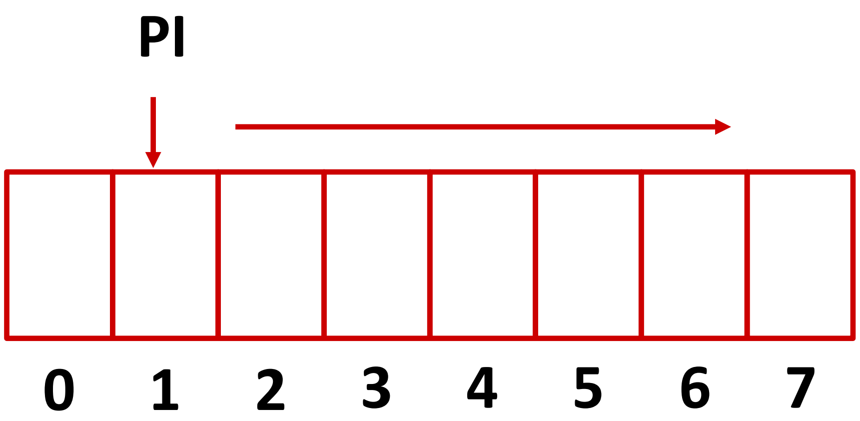


Producer/Consumer: Multiple Threads

- **How to implement with multiple producer and multiple consumers?**

Producer A:
0 <- findempty

Producer B:
1 <- findempty



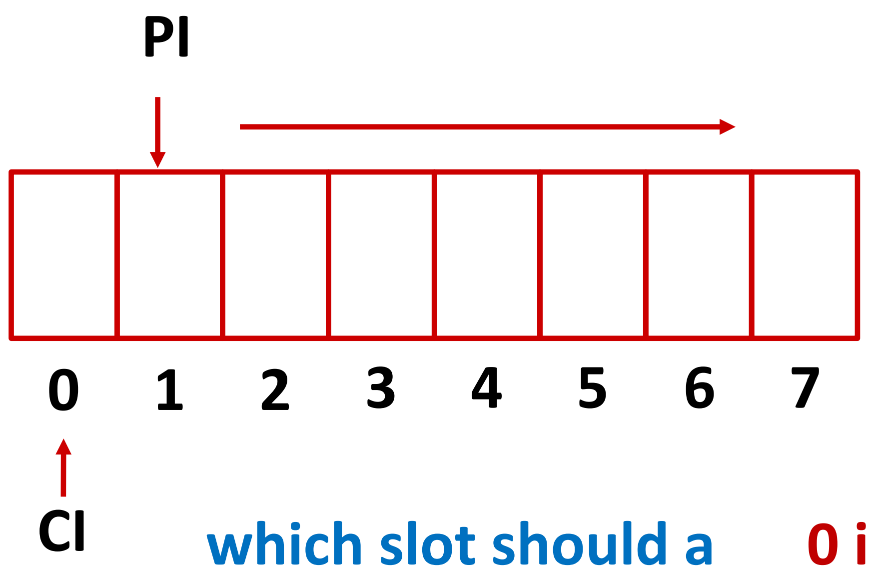
Producer/Consumer: Multiple Threads

- **How to implement with multiple producer and multiple consumers?**

Producer A:
0 <- findempty

Producer B:
1 <- findempty

Producer B completes signal(full_buffer)

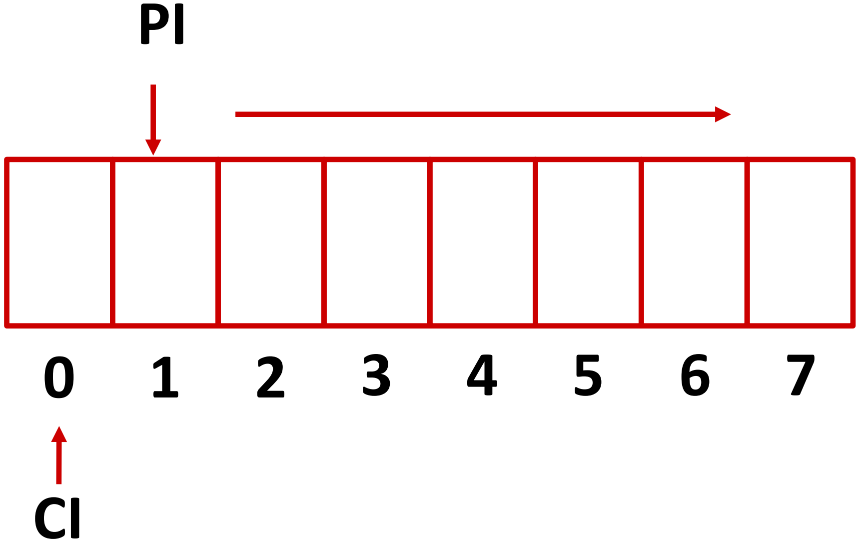


which slot should a consumer get?

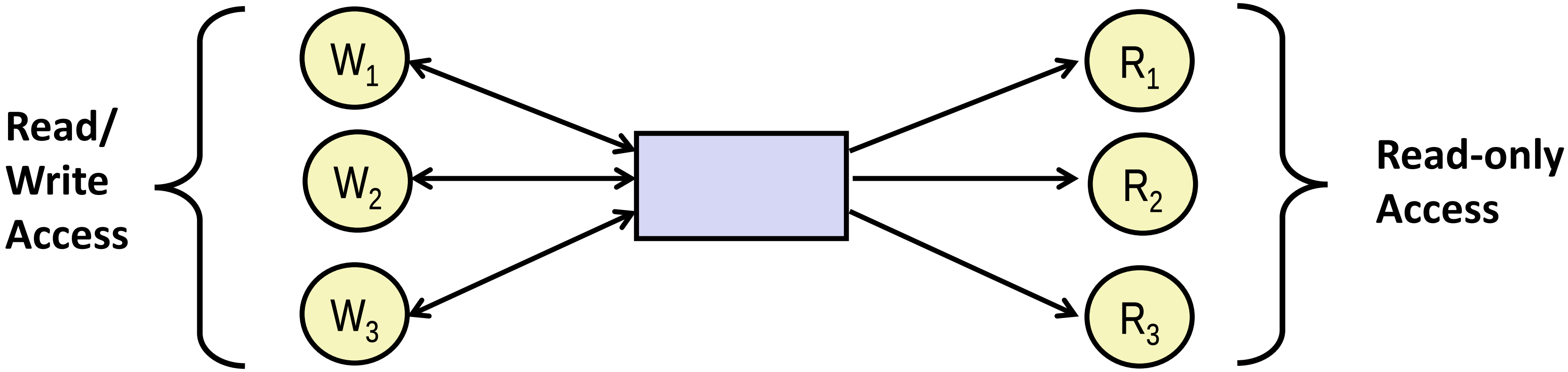
0 is incorrect should return 1

Producer/Consumer: Multiple Threads

- **How to implement with multiple producer and multiple consumers?**
  - Approach 1: In find full, scan the buffer to find an available one
  - Approach 2: Producer insert an item into a list, which is searched by consumer

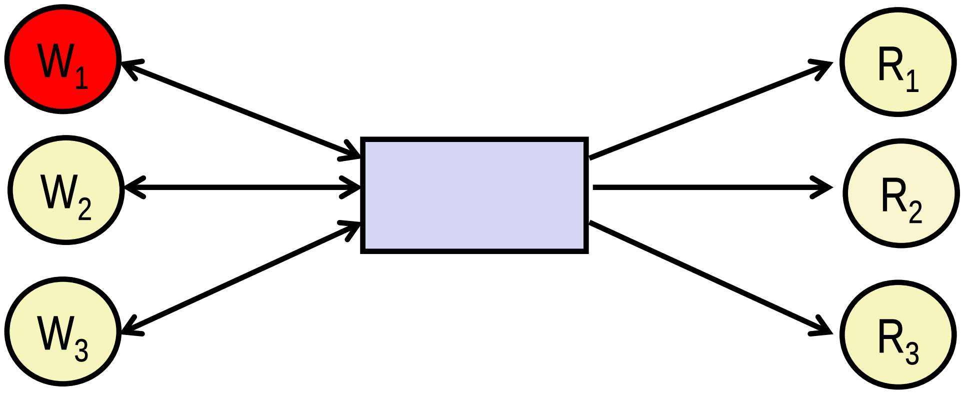


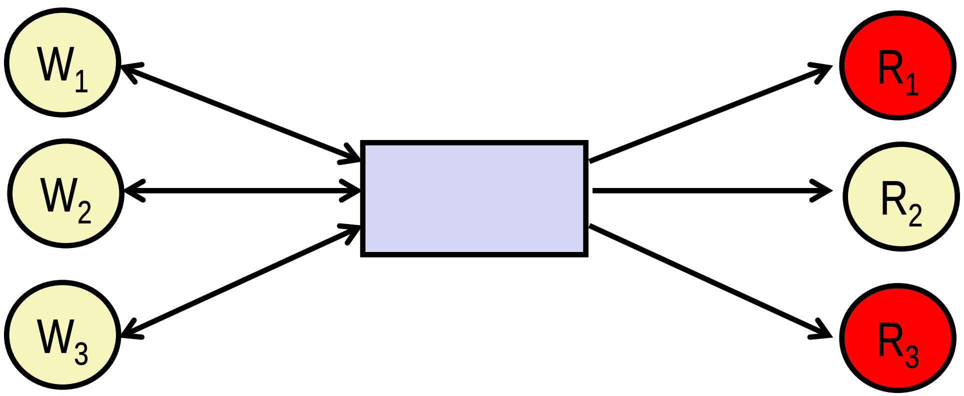
Readers-Writers Problem



- **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
- **Occurs frequently in real systems, e.g.,**
  - Online airline reservation system
  - Multithreaded caching Web proxy

Readers/Writers Examples





Reader/Writer Locks

- **Goal: Let multiple reader threads grab lock (shared)**
- **Only one writer thread can grab lock (exclusive)**
  - No reader threads
  - No other writer threads

- **How to implement reader/writer lock with semaphores**
  - `rwlock_acquire_readlock`
  - `rwlock_release_readlock`
  - `rwlock_acquire_writelock`
  - `rwlock_release_writelock`

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.
  - Web cache (写没那么重要)

- ***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.
  - Ticket reservation

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

Reader/Writer Locks

```
void rwlock_acquire_readlock(rwlock_t *rw) {
              //
}

void rwlock_release_readlock(rwlock_t *rw) {
              //
}

rwlock_acquire_writelock(rwlock_t *rw) {
              //
}

rwlock_release_writelock(rwlock_t *rw) {
              //
}
```

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T2: release_readlock()
T1: release_readlock()
T4: acquire_readlock()
T5: acquire_readlock() // ???
T3: release_writelock()
// what happens???

Reader/Writer Locks

```
typedef struct _rwlock_t {
        sem_t lock;
        sem_t writelock;
        int readers;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
        rw->readers = 0;
        sem_init(&rw->lock, 1);
   sem_init(&rw->writelock, 1);
}

```

Reader/Writer Locks

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T2: release_readlock()
T1: release_readlock()
T4: acquire_readlock()
T5: acquire_readlock() // ???
T3: release_writelock()
// what happens???

```
void rwlock_acquire_readlock(rwlock_t *rw) {
            sem_wait(&rw->lock);
            rw->readers++;
            if (rw->readers == 1)
                sem_wait(&rw->writelock);
            sem_post(&rw->lock);
}
void rwlock_release_readlock(rwlock_t *rw) {
            sem_wait(&rw->lock);
            rw->readers--;
            if (rw->readers == 0)
                sem_post(&rw->writelock); ]
            sem_post(&rw->lock);
}
rwlock_acquire_writelock(rwlock_t *rw) {
            sem_wait(&rw->writelock);
}
rwlock_release_writelock(rwlock_t *rw) {
            sem_post(&rw->writelock);
}
```

Solution to First Readers-Writers Problem

Readers: (读者永远 Block 写者)

```
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);
// 第一个reader阻塞writer

    /* 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:

R2

R1

```
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

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);
R2 →
R1 →  /* 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 == 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);

R2 → /* Reading happens here */

    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 == 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

R2

R1

Writers:

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

    /* Writing here */

    V(&w);
  }
}
```

W1

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);
R3
    /* Reading happens here */

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

Writers:

```
void writer(void)
{
  while (1) {       W1
    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++;
    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);
  }
}
```

W1

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**
  - 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

- **Consider two aspects**
  - Following writes can overtake reads
  - Block multiple reads enter concurrently into the reading area

How to Modify the First Version to the Second

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

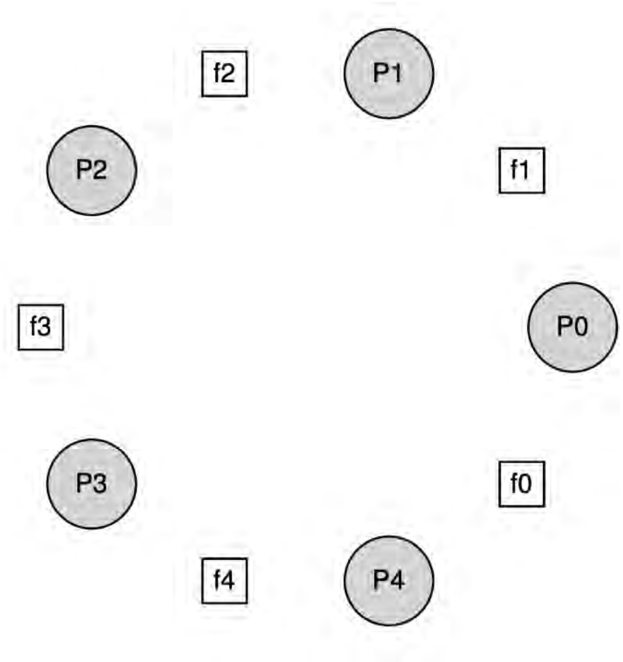
Solution to Second Readers-Writers Problem

```
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);
// 第一个writer阻塞reader

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

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

Dining Philosopher's Problem

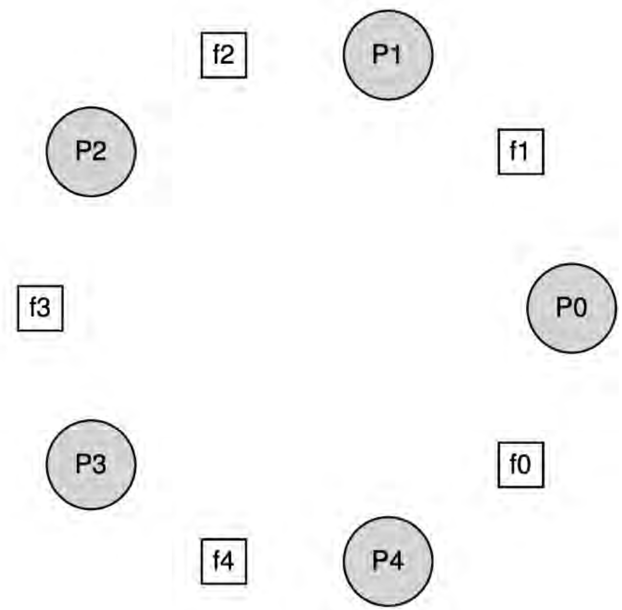
- **Problem Statement**
  - 5 philosophers on a table with 5 forks
  - A philosopher needs 2 forks to eat
  - Sometimes they think, no need for forks
  - **Challenge: no deadlock and no philosopher starves**




```
Each philosopher:

while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```

First Try

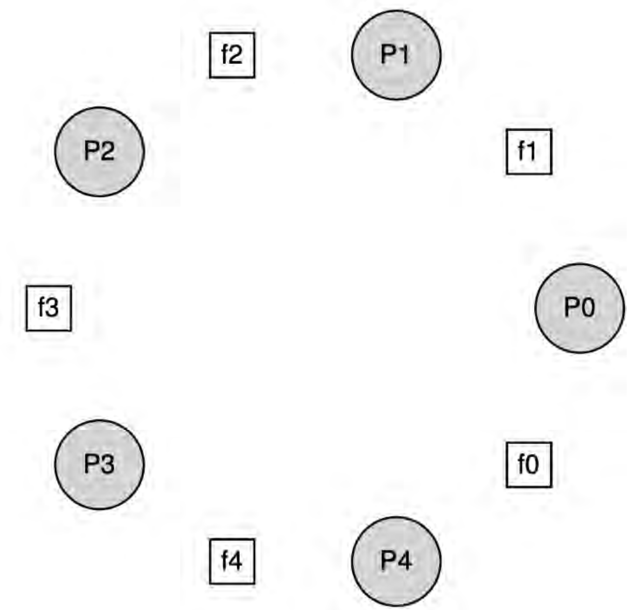
```
void get_forks(int p) {
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
}

void put_forks(int p) {
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
}
```



- **Wrong and cause deadlock**
  - If every philosopher takes their left fork simultaneously, no one can get the right fork
  - Deadlock!

Second Try: break the dependency

```
void get_forks(int p) {
    if (p == 4) {
        sem_wait(&forks[right(p)]);
        sem_wait(&forks[left(p)]);
    } else {
        sem_wait(&forks[left(p)]);
        sem_wait(&forks[right(p)]);
    }
}
```



- **Cycle of waiting is broken!**
  - More deadlocks on next lecture

Semaphores

- **Semaphores are equivalent to locks + condition variables**
  - Can be used for both mutual exclusion and ordering
- **Semaphores contain state**
  - How they are initialized depends on how they will be used
  - Init to 1: Mutex
  - Init to 0: Join (1 thread must arrive first, then other)
  - Init to N: Number of available resources
- **Sem_wait(): Waits until value > 0, then decrement (atomic)**
- **Sem_post(): Increment value, then wake a single waiter (atomic)**
- **Can use semaphores in producer/consumer relationships and for reader/writer locks**