CE 440 Introduction to Operating System

Lecture 7: Semaphores and Monitors
Fall 2025

Prof. Yigong Hu

BOSTON UNIVERSITY

Slides courtesy of Manuel Egele, Ryan Huang and Baris Kasikci

Administrivia

Project group member sign up due this Sunday

- Due this Sunday

Lab 1 overview session this Friday

- 2:30 - 4:00 PM, PHO305

Recap: Synchronization

Problem: concurrent threads accessed a shared resource without any synchronization

- Know as a race condition

The execution of the two threads can be interleaved

```
balance = get_balance(account);
balance = balance – amount;
```

```
balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);
```

```
put_balance(account, balance);
```

Context switch

Recap: How to Protect Shared Resource?

1. **Mutual exclusion (mutex)**
   - If one thread is in the critical section, then no other is

2. **Progress**
   - If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section
   - A thread in the critical section will eventually leave it

3. **Bounded waiting (no starvation)**
   - If some thread T is waiting on the critical section, then T will eventually enter the critical section

Recap: How to Protect Shared Resource?

4. Performance

- The overhead of entering and exiting the critical section is small with respect to the work being done within it

In summary:

- **Safety property**: nothing bad happens
  - Mutex
- **Liveness property:** something good happens
  - Progress, Bounded Waiting
- **Performance requirement**
  - Performance

Note: correctness of concurrent is guarantee by design

Recap: Lock

Code that uses mutual exclusion to synchronize its execution is called a critical section

A lock is an object in memory providing two operations

- acquire(): wait until lock is free, then take it to enter a C.S
- release(): release lock to leave a C.S, waking up anyone waiting for it

Recap: Higher-Level Synchronization

We looked at using locks to provide mutual exclusion

Locks work, but they have limited semantics

- Just provide mutual exclusion
- Wasteful

Instead, we need synchronization mechanisms that

- Block waiters, leave interrupts enabled in critical sections
- Provide semantics beyond mutual exclusion

Look at two common high-level mechanisms

- Semaphores: binary (mutex) and counting
- Monitors: mutexes and condition variables

Semaphores

Semaphores have a non-negative integer that supports the two operations:

- Semaphore::P() decrements, blocks until semaphore is open, a.k.a **wait()**
- Semaphore::V() increments, allows another thread to enter, a.k.a **signal()**
- That's it! No other operations – not even just reading its value
  - Both P and V are after the Dutch word “Proberen” (to try), “Verhogen” (increment)

Semaphore safety property: the semaphore value is always greater than or equal to 0

Using Semaphores to Fix Banking Problem

Use is similar to our locks, but semantics are different

```
struct Semaphore {
    int value;
    Queue q;
} S;

withdraw (account, amount) {
   P(S);
   balance = get_balance(account);
   balance = balance – amount;
   put_balance(account, balance);
   v(S);
   return balance;
}
```

Critical Section

Threads block

```
P(S);
balance = get_balance(account);
balance = balance – amount;

P(S);

P(S);

put_balance(account, balance);
v(s);

…
v(s);

…
v(s);
```

It is undefined which thread runs after a signal

Semaphores

Semaphores have a non-negative integer that supports the two operations:

- Semaphore::P() decrements, blocks until semaphore is open, a.k.a **wait()**
- Semaphore::V() increments, allows another thread to enter, a.k.a **signal()**
- That's it! No other operations – not even just reading its value
    - Both P and V are after the Dutch word “Proberen” (to try), “Verhogen” (increment)

Semaphore safety property: the semaphore value is always greater than or equal to 0

Semaphores are a kind of generalized lock

- First defined by Dijkstra in the “THE” system in 1968
- Main synchronization primitive used in original UNIX

Semaphores Implementation

Associated with each semaphore is a queue of waiting threads

When P() **is called by a thread:**

- If semaphore is open, thread continues
- If semaphore is closed, thread blocks on queue

Then V() **opens the semaphore:**

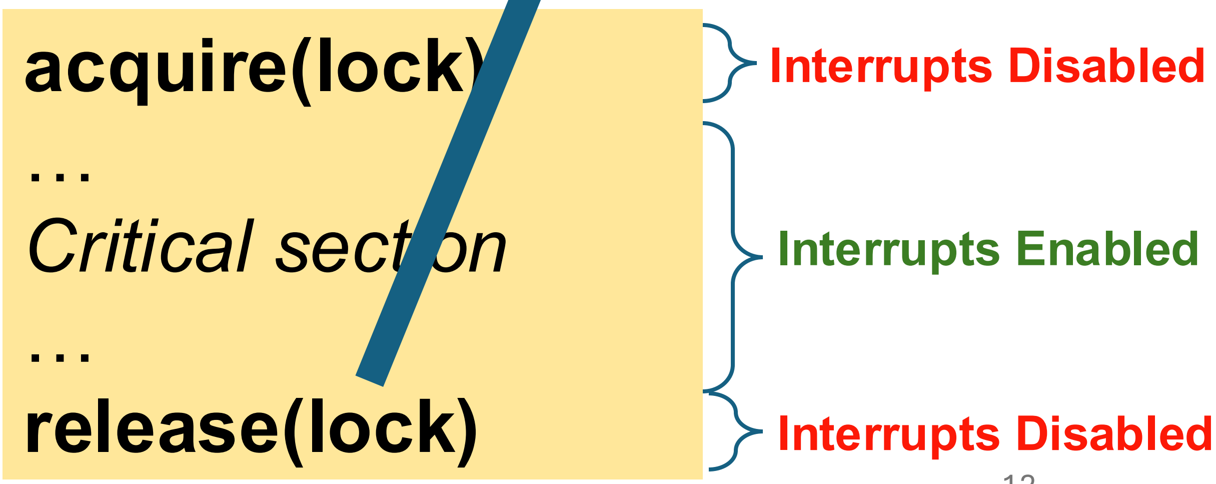
- If a thread is waiting on the queue, the thread is unblocked
- If no threads are waiting on the queue, the signal is remembered for the next thread
    - In other words, V() has "history" (c.f., condition vars later)
    - This "history" is a counter

Recall: Implementing Locks (4)

Block waiters, interrupts enabled in critical sections

```
struct lock {
    int held = 0;
    queue Q;
}
void acquire (lock) {
    Disable interrupts;
    while (lock→held) {
        put current thread on lock Q;
        block current thread;
    }
    lock→held = 1;
    Enable interrupts;
}
```

```
void release (lock) {
    Disable interrupts;
    if (Q) remove waiting thread;
    unblock waiting thread;
    lock→held = 0
    Enable interrupts;
}
```



Pintos threads/synch.c: sema_down/up

Semaphore Types

Semaphores come in two types

Mutex semaphore (or binary semaphore)

- Represents single access to a resource
- Guarantees mutual exclusion to a critical section

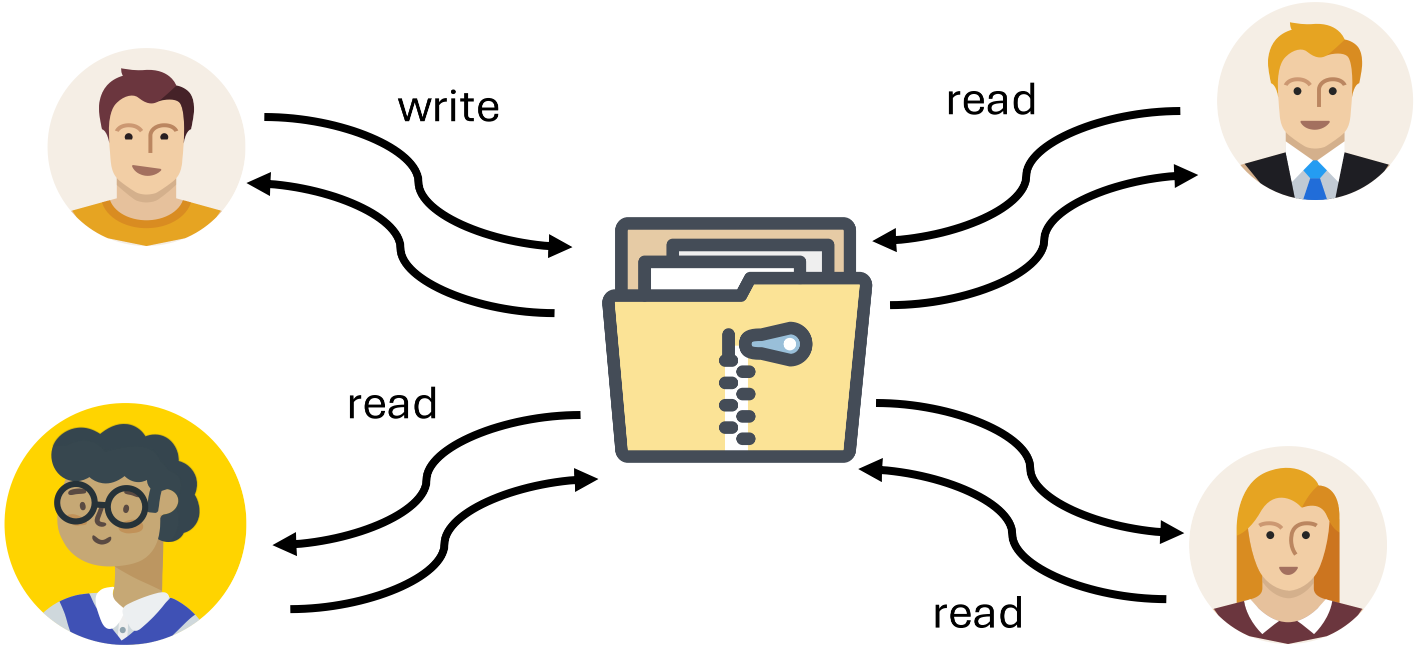
Counting semaphore (or general semaphore)

- Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
- Multiple threads can pass the semaphore
- Number of threads determined by the semaphore "count"
  - mutex has count = 1, counting has count = N

Readers/Writers Problem

Consider a shared database

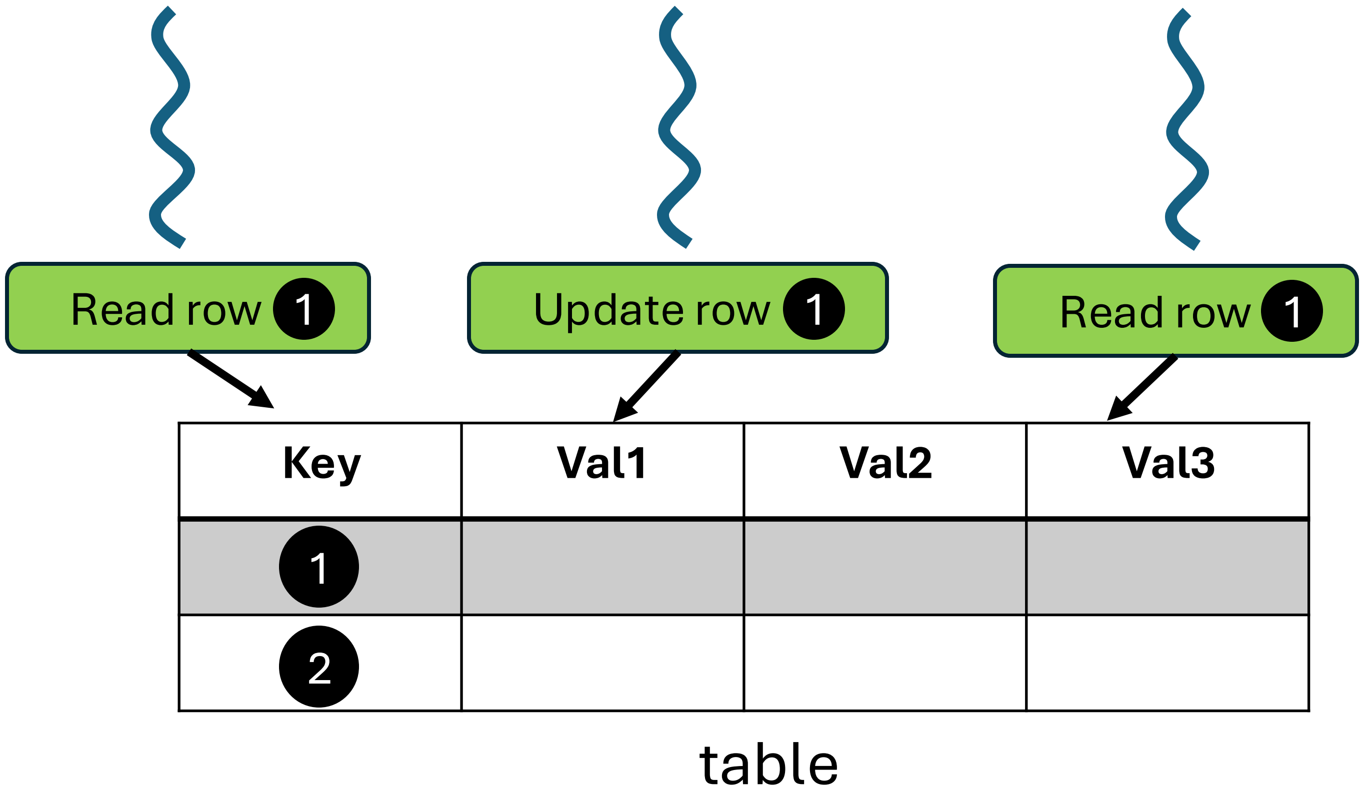
- Two classes of users:
  - Readers – never modify database
  - Writers – read and modify database
- Is using a single lock on the whole database sufficient?
  - Like to have many readers at the same time
  - Only one writer at a time



Readers/Writers Problem

Readers/Writers Problem:

- An object is shared among several threads
- Some threads only read the object, others only write it
- How do we control the access pattern?



Readers/Writers Problem

If we have multiple readers

Get (1,2,3)

Read row 1

Get (1,2,3)

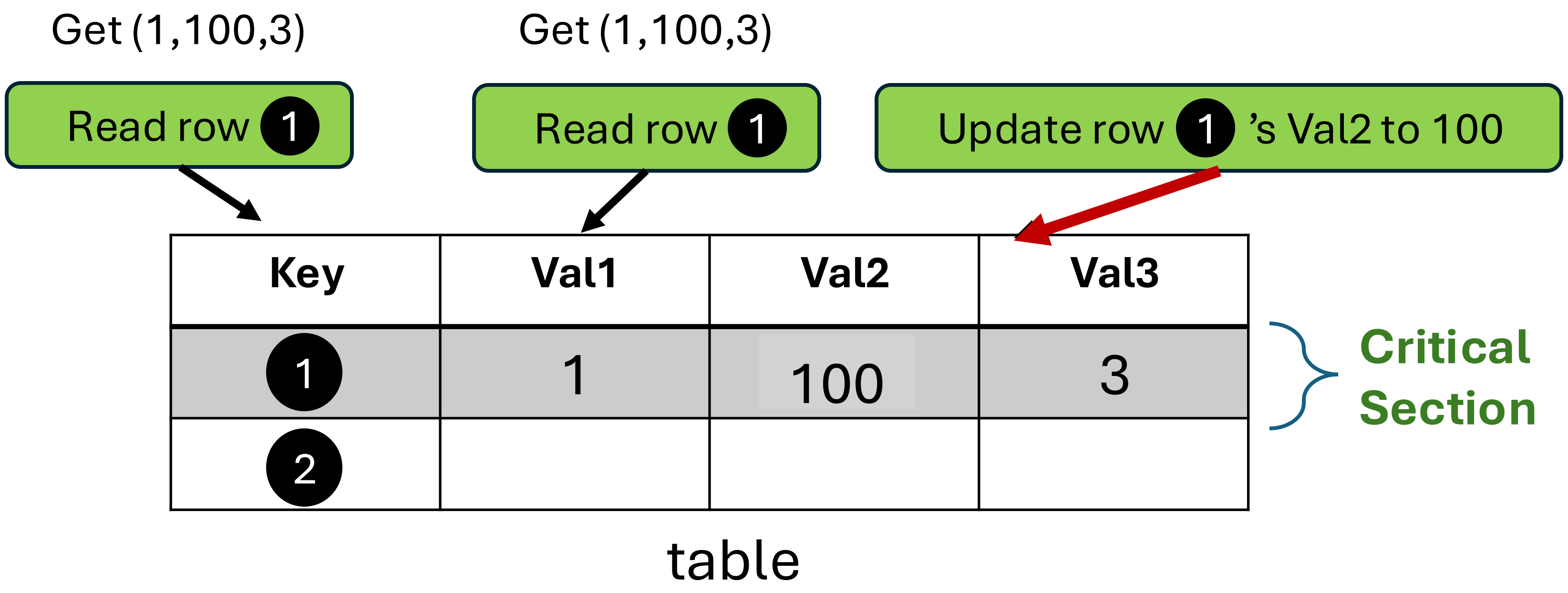
Read row 1

Key	Val1	Val2	Val3
1	1	2	3
2			

table

Readers/Writers Problem

If we have multiple readers and one writer



Readers/Writers Problem

If we have multiple writers

Update row 1

Update row 1

Key	Val1	Val2	Val3
1	1	100	3
2			

Critical Section

table

Readers/Writers Problem

Readers/Writers Problem:

- An object is shared among several threads
- Some threads only read the object, others only write it
- We can allow multiple readers but only one writer
  - Let r be the number of readers, w be the number of writers
  - Safety: $(r \geq 0) \wedge (0 \leq w \leq 1) \wedge ((r > 0) \Rightarrow (w = 0))$

How can we use semaphores to implement this protocol?

Basic Readers/Writers Solution

Safety Constraints:

- Safety: $(r \geq 0) \wedge (0 \leq w \leq 1) \wedge ((r > 0) \Rightarrow (w = 0))$

Basic structure of a solution:

- **Reader()**
  - **Wait until no writers**
  - **Access database**
  - **Check out – wake up a waiting writer**
- **Writer()**
  - **Wait until no active readers or writers**
  - **Access database**
  - **Check out – wake up waiting readers or writer**

Start with...

- Semaphore w_or_r– exclusive writing or reading

Using Semaphores for Readers/Writers

w_or_r provides mutex between readers and writers

- writer wait/signal, reader wait/signal when readcount goes from 0 to 1 or from 1 to 0

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Readers/Writers Notes

Consider the following sequence of operators:

- W1, R3, R4

Why do readers use mutex**?**

Why don't writers use mutex**?**

What if the signal() is above “if (readcount == 1)”**?**

Simulation of Readers/Writers Solution

W1 comes first, R1, R2 come along

w_or_r = 1, mutex = 1, readcount = 0

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 comes first, R1, R2 come along

w_or_r = 0, mutex = 1, readcount = 0

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 comes first, R1, R2 come along

w_or_r = 0, mutex = 0, readcount = 0

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 comes first, R1, R2 come along

w_or_r = 0, mutex = 0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 comes first, R1, R2 come along

w_or_r = 0, mutex = 0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 comes first, R1, R2 come along

w_or_r = 0, mutex = 0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 1, mutex = 0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 0, mutex = 0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 0, mutex = 1, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 0, mutex = 0, readcount =1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 0, mutex = 0, readcount =2

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 0, mutex = 0, readcount = 2

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 0, mutex = 1, readcount = 2

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 0, mutex = 0, readcount = 2

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 0, mutex = 0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 0, mutex = 0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 0, mutex = 0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 0, mutex = 0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 0, mutex = 1, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 0, mutex = 0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 0, mutex = 0, readcount = 0

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 0, mutex = 0, readcount = 0

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 1, mutex = 0, readcount = 0

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 1, mutex = 0, readcount = 0

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Readers/Writers Notes

Consider the following sequence of operators:

- W1, R3, R4

Why do readers use mutex?

Why don't writers use mutex?

What if the signal() is above "if (readcount == 1)"?

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    signal(&mutex); // unlock readcount
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    Read;
```

Simulation of Readers/Writers Solution

W1 finishes, R1, R2 continue

w_or_r = 1, mutex = 1, readcount = 0

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Is this safe?

Readers/Writers Notes

Is It Safe?

- Yes

If readers and writers are waiting, who goes first?

If Writer and Reader Are Waiting for Writer

W1 comes first, R1, W2 come along

w_or_r = 1, mutex = 1, readcount = 0

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

If Writer and Reader Are Waiting for Writer

W1 comes first, R1, W2 come along

w_or_r = 0, mutex = 1, readcount = 0

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

If Writer and Reader Are Waiting for Writer

W1 comes first, R1, W2 come along

w_or_r = 0, mutex = 1, readcount = 0

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

If Writer and Reader Are Waiting for Writer

W1 comes first, R1, W2 come along

w_or_r = 0, mutex = 0, readcount = 0

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

If Writer and Reader Are Waiting for Writer

W1 comes first, R1, W2 come along

w_or_r = 0, mutex = 0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

If Writer and Reader Are Waiting for Writer

W1 comes first, R1, W2 come along

w_or_r = 0, mutex = 0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

If Writer and Reader Are Waiting for Writer

W1 comes first, R1, W2 come along

w_or_r = 0, mutex = 0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Who go first?

Readers/Writers Notes

If a writer is writing, where will readers be waiting?

- **Yes**

If readers and writers are waiting, who goes first?

- If waiting for writers, once a writer exits, all readers/writers can fall through
  - Which reader gets to go first?

If Writer and Reader Are Waiting for Reader

R1 comes first, W1, R2 come along

w_or_r = 1, mutex = 1, readcount = 0

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex);// lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

If Writer and Reader Are Waiting for Reader

R1 comes first, W1, R2 come along

w_or_r = 1, mutex = 0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

If Writer and Reader Are Waiting for Reader

R1 comes first, W1, R2 come along

w_or_r = 1, mutex = 0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

If Writer and Reader Are Waiting for Reader

R1 comes first, W1, R2 come along

w_or_r = 0, mutex =0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

If Writer and Reader Are Waiting for Reader

R1 comes first, W1, R2 come along

w_or_r = 0, mutex =0, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Who go first?

Readers Always Go First

R1 comes first, W1, R2 come along

w_or_r = 0, mutex =1, readcount = 1

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Readers Always Go First

R1 comes first, W1, R2 come along

w_or_r = 0, mutex =0, readcount = 2

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Readers Always Go First

R1 comes first, W1, R2 come along

w_or_r = 0, mutex =1, readcount = 2

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Readers Always Go First

R1 comes first, W1, R2 come along

w_or_r = 0, mutex =1, readcount = 2

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

Readers Always Go First

R1 finishes, R2 blocks W1

w_or_r = 0, mutex =1, readcount = 2

```
// exclusive writer or reader
Semaphore w_or_r(1);

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex(1);

writer() {
    wait(&w_or_r); // lock out others
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex); // unlock readcount
    Read;
    wait(&mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex); // unlock readcount
}
```

What if we have R3,R4,R5… coming now?

Readers/Writers Notes

If a writer is writing, where will readers be waiting?

- **Yes**

If readers and writers are waiting, who goes first?

- If waiting for writers, once a writer exits, all readers/writers can fall through
    - Which reader gets to go first?

Readers/Writers Notes

If a writer is writing, where will readers be waiting?

- **Yes**

If readers and writers are waiting, who goes first?

- If waiting for writers, once a writer exits, all readers/writers can fall through
  - Which reader gets to go first?
- If waiting for readers, possible starvation for the writer

Semaphore Questions

Are there any problems that can be solved with counting semaphores that cannot be solved with mutex semaphores?

- If a system only gives you mutex semaphore, can you use it to implement counting semaphores?

Does it matter which thread is unblocked by a signal operation?

Tips for Pintos: Semaphore Implementation

```c
void sema_down(struct semaphore *sema)
{
    enum intr_level old_level;
    old_level = intr_disable();
    while (sema->value == 0) {
        list_push_back(&sema->waiters,
              &thread_current()->elem);
        thread_block();
    }
    sema->value--;
    intr_set_level(old_level);
}
```

```c
void sema_up(struct semaphore *sema)
{
    enum intr_level old_level;
    old_level = intr_disable();
    if (!list_empty (&sema->waiters))
        thread_unblock(list_entry(
              list_pop_front(&sema->waiters),
              struct thread, elem));
    sema->value++;
    intr_set_level(old_level);
}
```

To reference current thread: thread_current()

`thread_block()` **puts the current thread to sleep**

Lab 1 note:

- leverage semaphore instead of directly using thread_block()

Tips for Pintos: thread_block()

```
/* Puts the current thread to sleep. This
function
must be called with interrupts turned off.*/
void thread_block ()
{
ASSERT (!intr_context ());
ASSERT (intr_get_level () == INTR_OFF);
thread_current ()->status = THREAD_BLOCKED;
schedule ();
}
```

Pick another thread to run (← schedule ();)

thread_block() **assumes the interrupts are disabled**

This means we will have the thread sleep with interrupts disabled

Isn't this bad?

- Shouldn't we only disable interrupts when entering/leaving critical sections but keep interrupts enabled during critical section?

Interrupts Re-enabled Right After Context Switch

```
thread_yield() {
    Disable interrupts;
    add current thread to ready_list;
    schedule(); // context switch
    Enable interrupts;
}
```

```
sema_down() {
    Disable interrupts;
    while(value == 0) {
        add current thread to waiters;
        thread_block();
    }
    value--;
    Enable interrupts;
}
```

Thread 1

```
[thread_yield]
Disable interrupts;
add current thread to ready_list;
schedule();
```

Thread 2

```
[thread_yield]
(Returns from schedule())
Enable interrupts;
```

...

Thread 2

```
[sema_down]
Disable interrupts;
while(value == 0) {
    add current thread to waiters;
    thread_block();
}
```

Thread 1

```
[thread_yield]
(Returns from schedule())
Enable interrupts;
```

Semaphore Summary

Semaphores can be used to solve any traditional sync. Problems

However, they have some drawbacks

- They are essentially shared global variables
  - Can potentially be accessed anywhere in program
- No connection between the semaphore and the data controlled by the semaphore
- Used both for critical sections (mutual exclusion) and coordination (scheduling)
  - Note that I had to use comments in the code to distinguish
- No control or guarantee of proper usage

Sometimes hard to use and prone to bugs

Semaphores are good but… Monitors are better!

Semaphores are a huge step up; just think of trying to do the reader/writer with only loads and stores or lock

Problem is that semaphores are dual purpose:

- They are used for both mutex and scheduling constraints

Insight:

- Use locks for mutual exclusion and condition variables for scheduling constraints
- Use programming language support

Monitor

A programming language construct that controls access to shared data

- Synchronization code added by compiler, enforced at runtime
- Why is this an advantage?

A monitor is a module that encapsulates

- Shared data structures
- Procedures that operate on the shared data structures
- Synchronization between concurrent threads that invoke the procedures

```
Monitor account {
 double balance;

 double withdraw (amount) {
   balance = balance – amount;
   put_balance(account, balance);
   return balance;
}
```

Monitor

A programming language construct that controls access to shared data

- Synchronization code added by compiler, enforced at runtime
- Why is this an advantage?

A monitor is a module that encapsulates

- Shared data structures
- Procedures that operate on the shared data structures
- Synchronization between concurrent threads that invoke the procedures

A monitor protects its data from unstructured access

It guarantees that threads accessing its data through its procedures interact only in legitimate ways

Bank Account Problem With Monitor

```
Monitor account {
 double balance;

 double withdraw (amount) {
   balance = balance – amount;
   put_balance(account, balance);
   return balance;
}
```

Threads block

```
withdraw(amount)
  balance = balance – amount;
```

```
withdraw(amount)
```

```
withdraw(amount)
```

```
return balance
```

```
balance = balance – amount;
return balance;
```

```
balance = balance – amount;
return balance;
```

When first thread exits, another can enter. Which one is undefined

Monitor Semantics

A monitor guarantees mutual exclusion

- Only one thread can execute any monitor procedure at any time
  - The thread is “in the monitor”
- If a second thread invokes a monitor procedure when a first thread is already executing one, it blocks
  - So the monitor has to have a wait queue…
- If a thread within a monitor blocks, another one can enter

What are the implications in terms of parallelism in a monitor?

A monitor invariant is a safety property associated with the monitor

- It’s expressed over the monitored variables.
- It holds whenever a thread enters or exits the monitor.

Condition Variables

But what if a thread wants to wait for something inside the monitor?

- If we busy wait, it's bad
- Even worse, no one can get in the monitor to make changes now!

A condition variable is associated with a condition needed for a thread to make progress once it is in the monitor.

```
Monitor M {
    ... monitored variables
    Condition c;

    void enterMonitor (...) {
        if (extra property not true) wait(c); waits outside of the monitor's mutex
        do what you have to do
        if (extra property true) signal(c);  brings in one thread waiting on condition
}
```

Condition Variables

Condition variables support three operations:

- Wait – release monitor lock, wait for C/V to be signaled
  - So condition variables have wait queues, too
- Signal – wakeup one waiting thread
- Broadcast – wakeup all waiting threads

Condition variables are not boolean objects

- if (condition_variable) then... does not make sense
- if (num_resources == 0) then wait(resources_available) does
- We will explain the detail in next lecture

Condition Variables != Semaphores

Condition variables != semaphores

- Although their operations have the same names, they have entirely different semantics (such is life, worse yet to come)
- However, they each can be used to implement the other

Access to the monitor is controlled by a lock

- wait() blocks the calling thread, and gives up the lock
  - To call wait, the thread has to be in the monitor (hence has lock)
  - Semaphore::wait just blocks the thread on the queue
- signal() causes a waiting thread to wake up
  - If there is no waiting thread, the signal is lost
  - Semaphore::signal increases the semaphore count, allowing future entry even if no thread is waiting
  - Condition variables have no history

Signal Semantics

Two flavors of monitors that differ in the scheduling semantics of signal()

- Hoare monitors (original)
  - signal() immediately switches from the caller to a waiting thread
  - The condition that the waiter was anticipating is guaranteed to hold when waiter executes
  - Signaler must restore monitor invariants before signaling

Hoare

```
if (!condition)
    wait(cond_var);
```

Condition definitely holds since we just context switched from signal

Signal Semantics

- Mesa monitors (Mesa, Java)
  - signal() places a waiter on the ready queue, but signaler continues inside monitor
  - Condition is not necessarily true when waiter runs again
    - Returning from wait() is only a *hint* that something changed
    - Must recheck conditional case

Mesa

```
while (!condition)
    wait(cond_var);
```

condition might have been changed, if so, wait again

condition holds now

Hoare vs. Mesa Monitors

Tradeoffs

- Mesa monitors easier to use, more efficient
  - Fewer context switches, easy to support broadcast
- Hoare monitors leave less to chance
  - Easier to reason about the program

Summary

Semaphores

- wait()/signal()implement blocking mutual exclusion
- Also used as atomic counters (counting semaphores)
- Can be inconvenient to use

Monitors

- Synchronizes execution within procedures that manipulate encapsulated data shared among procedures
    - Only one thread can execute within a monitor at a time
- Relies upon high-level language support

Condition variables

- Used by threads as a synchronization point to wait for events
- Inside monitors, or outside with locks