# CS-446/646

## Synchronization

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

## *Producer-Consumer* with *Threads*

```c
void* produce(void *arg) {
  int i;
  for(i=0; i<1e7; ++i)
    ++ balance;
}
```

```c
void* consume(void *arg) {
  int i;
  for(i=0; i<1e7; ++i)
    -- balance;
}
```

```c
int balance = 0;
int main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, produce, (void*)1);
  pthread_create(&t2, NULL, consume, (void*)2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("all done: balance = %d\n", balance);
  return 0;
}
```

```
$ gcc -Wall -pthread -o bank bank.c
$ ./bank
all done: balance = 0
$ ./bank
all done: balance = 140020
$ ./bank
all done: balance = -94304
$ ./bank
all done: balance = -191009
```

## *Producer-Consumer* with *Threads*

Why?

- Load – Increment/Decrement – Store

```
$ objdump -d bank
…
08048464 <produce>:
…                                  // ++ balance
8048473: a1 80 97 04 08            mov 0x8049780,%eax
8048478: 83 c0 01                  add $0x1,%eax
804847b: a3 80 97 04 08            mov %eax,0x8049780
…
0804849b <consume>:
…                                  // -- balance
80484aa: a1 80 97 04 08            mov 0x8049780,%eax
80484af: 83 e8 01                  sub $0x1,%eax
80484b2: a3 80 97 04 08            mov %eax,0x8049780
…
```

One possible *Thread Schedule*

| Thread 1 | | Thread 2 |
|---|---|---|
| | balance: 0 | |
| mov 0x8049780,%eax | | |
| | eax0: 0 | |
| add $0x1,%eax | | |
| | eax0: 1 | |
| mov %eax,0x8049780 | | |
| | balance: 1 | |
| | | mov 0x8049780,%eax |
| | eax1: 1 | |
| | | sub $0x1,%eax |
| | eax1: 0 | |
| | | mov %eax,0x8049780 |
| | balance: 0 | |

## ***Producer-Consumer* with *Threads***

Why?

- Load – Increment/Decrement – Store

```
$ objdump -d bank
…
08048464 <produce>:
…                               // ++ balance
8048473: a1 80 97 04 08         mov 0x8049780,%eax
8048478: 83 c0 01               add $0x1,%eax
804847b: a3 80 97 04 08         mov %eax,0x8049780
…
0804849b <consume>:
…                               // -- balance
80484aa: a1 80 97 04 08         mov 0x8049780,%eax
80484af: 83 e8 01               sub $0x1,%eax
80484b2: a3 80 97 04 08         mov %eax,0x8049780
…
```

A more "problematic" *Thread Schedule*

| Thread 1 | | Thread 2 |
|---|---|---|
| | balance: 0 | |
| mov 0x8049780,%eax | | |
| | eax0: 0 | |
| add $0x1,%eax | | |
| | eax0: 1 | |
| | | mov 0x8049780,%eax |
| | eax1: 0 | |
| mov %eax,0x8049780 | | |
| | balance: 1 | |
| | | sub $0x1,%eax |
| | eax1: -1 | |
| | | mov %eax,0x8049780 |
| | balance: -1 | |

- Interrupt can occur before and after any *Instruction* (but not during it)

## *Race Condition*

*Definition:* A timing-dependent error involving *Shared* state

Very bad

- "*Non-Deterministic*"
  - Can't know what the output will be, and it is likely to be different across runs
- Hard to detect
  - Too many possible *Schedules*
- Hard to debug
  - "*Heisen-bug*" : debugging changes timing so it can hide the bugs (vs "*Bohr-bug*")

## Avoiding *Race Conditions*

*Atomic* Operations

- No other *Instructions* can be interleaved
- Entire operation is executed "as a unit" - Guaranteed by Hardware

Possible approach:

- Have a dedicated *Atomic Instruction* for the job:
  - `add $0x1, 0x8049780`

```
// ++ balance
mov 0x8049780,%eax
add $0x1,%eax
mov %eax,0x8049780
```

Problem:

- Can't anticipate every possible way we want *Atomicity*
- Increases Hardware complexity, slows down other *Instructions*

## Layered Approach to *Synchronization*

- Hardware provides simple low-level *Atomic* Operations
  - Upon which we can build high-level *Synchronization* Primitives
    - Upon which we can implement *Critical Sections* and build correct Multi-Threaded/Multi-Processing programs

| Properly synchronized Application |
| --- |
| **High-level *Synchronization Primitives*** |
| **Hardware-provided low-level *Atomic* operations** |

- Example low-level *Atomic* Operations
  - On Uniprocessor, disable/enable *Interrupts*
  - On x86, *Aligned-Load* and *Aligned-Store* of words
  - Special instructions: Test-Set-Lock/Exchange (TSL, XCHG), Compare-and-Swap (lock CMPXCHG)
- Example high-level *Synchronization* Primitives
  - *Lock*, *Semaphore*, *Monitor*

## The Problem with *Threads*

**x** is a global variable initialized with 0

Thread 1

```
void foo()
{
  x++;
}
```

Thread 2

```
void bar()
{
  x--;
}
```

After both *Threads* finish, what is **x** ?

- 0, 1, -1
  - *Assembly*-level *Instruction* sequence + *Time-Slice Interrupt* causing *Thread Switching* in the middle
    - Would run into same situation with pre-increment as well

## The Problem with *Threads*

Global
```
int p = 0, ready = 0;
```

| Process 1 | Process 2 |
|---|---|
| `p = 1000;`<br>`ready = 1;` | `while (!ready);`<br>`use(p);` |

What value of **`p`** is read by *Process 2*?

- 0, 1000
  - Compiler is free to *Reorder* if it can "prove" no side-effects

## *Synchronization* Motivation

*Threads* cooperate in Multithreaded programs

- To **share** resources, access shared data structures
- To **coordinate** their execution

For correct execution, control of this cooperation is required

- *Thread Scheduling* is non-deterministic (i.e. runtime behavior changes on same program re-runs)
  - *Scheduling* is not under the Program's control
    - *Scheduler* is part of OS
  - *Threads* interleave executions arbitrarily and at different rates
- Multi-Word operations are not *Atomic*
- Compiler (e.g. *Instructions*) and/or Hardware (e.g. *Memory*) *Reordering*

## Shared Resources

Initially focus on controlling access to *Shared Resources*

Basic problem

- If two concurrent *Threads* (/*Processes*) are accessing a shared variable, and that variable is read/modified/written by those *Threads*, then access to the variable must be controlled

We need

- Mechanisms to control access to *Shared Resources*
  - *Locks*, *Mutexes*, *Semaphores*, *Monitors*, *Condition Variables*, etc.
- Patterns for coordinating accesses to *Shared Resources*
  - *Bounded-Buffer*, *Producer-Consumer*, etc.

## Example: **Bank Account Balance**

Implement a function to handle withdrawals from a bank account

```
int withdraw (account, amount) {
  balance = get_balance(account);
  balance = balance - amount;
  put_balance(account, balance);
  return balance;
}
```

*Problem:* Suppose 2 people go to separate ATMs and simultaneously initiate withdrawal

- Bank server runs the 2 Threads:

```
int withdraw (account, amount) {
  balance = get_balance(account);
  balance = balance - amount;
  put_balance(account, balance);
  return balance;
}
```

```
int withdraw (account, amount) {
  balance = get_balance(account);
  balance = balance - amount;
  put_balance(account, balance);
  return balance;
}
```

## Example: **Bank Account Balance**

A Bad *Schedule*

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

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

*Thread Context Switching*

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

- Initial balance: **1000**, 2 x Withdrawal amount (each): **100**
- Final balance: **900**

## Example: **Bank Account Balance**

Can get a lot more interleaved

- *Remember*: Case of *Producer-Consumer Assembly*-level *Thread Schedule*

Assumptions:

- We have to assume that the only *Atomic* operations are *Instructions*
  - e.g. reads and writes of Words
    - even for that, the Hardware has to explicitly provide such support
- A *Context Switch* can happen at any time
- A *Thread* can be delayed indefinitely as long as it is not forever
  - no *Real-Time* guarantee

## Shared Resources

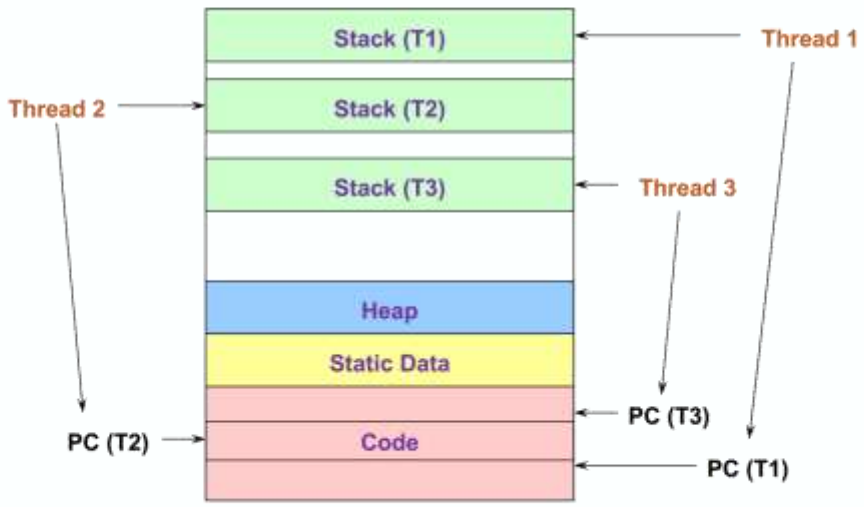
The previously demonstrated problem cam from accessing a *Shared Resource* without proper *Synchronization*

- *Race Condition*

Controlled-access mechanisms to *Shared Data Structures* (bank account, queues, lists, hash tables, etc.) are required to deal with *Concurrency*, so we can ensure a degree of determinism in program execution.

What is *Shared:*

- Local variables are not shared
  - Refer to data on the *Stack*
  - Each *Thread* has its own *Stack*
  - Potentially dangerous to pass/share/store a pointer to a local variable on the *Stack* of one *Thread* to another
- Global and **`static`** variables are shared
  - Stored in the *Static Data Segment*, accessible by any *Thread*
- Dynamic and other *Heap* data are shared
  - Allocated from *Heap* with **`malloc`**/**`free`**



# Synchronization

### ***Mutual Exclusion***

We use *Mutual Exclusion* to *Synchronize* access to *Shared Resources*

- Allows us to build larger *Atomic* blocks

### ***Critical Section***

Code that uses *Mutual Exclusion* to *Synchronize* its execution

- Only one *Thread*'s execution at a time can enter-or-be in the *Critical Section*
- All other *Threads* are forced to wait on entry
- When a *Thread* leaves a *Critical Section*, another can enter

## *Critical Section* Requirements

*Mutual Exclusion (Mutex)*
- If one *Thread* is in the *Critical Section*, then no other is

*Liveness (Progress)*
- If some *Thread T* is not in the *Critical Section*, then *T* cannot prevent some other thread *S* from entering
  - If multiple *Threads* simultaneously request to enter *Critical Section*, one must be allowed to proceed
  - A *Thread*'s operations outside the *Critical Section* should not be able to prevent another one to proceed
- A *Thread* in the *Critical Section* will eventually leave it

*Bounded Waiting (Starvation-free)*
- If some *Thread T* is waiting on the *Critical Section*, then *T* will eventually enter the *Critical Section*

*Performance*
- The overhead of entering and exiting the *Critical Section* is small with respect to the work being done within it

## *Critical Section* Desired Properties

*Safety* : Nothing bad should happen (**#1 Priority**)

- *Mutex*

*Liveness* : Something useful should be happening

- *Progress, Bounded Waiting*

*Performance* :

- *Efficiency*: Don't consume too many Resources while waiting
  - Don't *Busy-Wait* (*Spin-Wait*). Better to relinquish CPU and let another *Thread* run
- *Fairness*: Don't make one *Thread* wait longer than others.
  - Hard to do efficiently
- *Simplicity*: Should be simple to use

- Properties hold for each run, while *Performance* is quantified by all runs

## *Critical Section* Implementation – using `pthread_mutex_t`

`int pthread_mutex_lock(pthread_mutex_t *mutex);`

Acquire *Mutex* (/*Lock*) exclusively; wait if not available

- The *Mutex* object referenced by **`mutex` shall be Locked** by calling `pthread_mutex_lock()`. If the *Mutex* is already locked, the calling *Thread* **shall Block** until the **`mutex`** becomes available.

`int pthread_mutex_unlock(pthread_mutex_t *mutex);`

Release exclusive access to *Mutex* (/*Lock*)

- **Shall Release** the *Mutex* object referenced by **`mutex`**.

```
pthread_mutex_t l = PTHREAD_MUTEX_INITIALIZER;
```

```
void* produce(void *arg) {
  int i;
  for(i=0; i<1e7; ++i)
    pthread_mutex_lock(&l);
    ++ balance;
    pthread_mutex_unlock(&l);
}
```

```
void* consume(void *arg) {
  int i;
  for(i=0; i<1e7; ++i)
    pthread_mutex_lock(&l);
    -- balance;
    pthread_mutex_unlock(&l);
}
```

Critical Section

## Implementing *Locks* – v1

On a Uniprocessor we can cheat

- Implement *Mutual Exclusion* by disabling/enabling *Interrupts*

```
void lock()
{
  disable_interrupts();
}
```

```
void unlock()
{
  enable_interrupts();
}
```

*Good*:

- Simple

*Bad:*

- Both operations are *Privileged*, *User-Level* program not allowed to use them
- Doesn't work on Multiprocessor
  - On multi-core architectures, enabling/disabling *Interrupts* is on a per-core basis. One *Thread* might be running on a different core, so we would either have to disable *Interrupts* on all cores, or have an architecture-dependent implementation using *Inter-Processor Interrupts* (*IPIs*).

## Implementing *Locks* – v2

Software-based *Lock*

Desired specifications for a Software-based *Lock* algorithm:

*Good:*
- Shouldn't require much from hardware

Only assumptions:
- *Loads* and *Stores* are *Atomic*
- They execute *In-Order*
  - (vs *Out-of-Order* execution)
- Does not require special hardware Instructions

## Implementing *Locks* – v2

Software-based *Lock* – 1st Attempt

```
// 0: lock is available, 1: lock is held by a thread
int flag = 0;
```

```
void lock() {
  while (flag == 1); // spin wait
  flag = 1;
}
```

```
void unlock() {
  flag = 0;
}
```

Idea: Use one *Flag*, *Test* then *Set*; if unavailable *Spin-Wait*

Problem?

- Not *Safe*: Both *Threads* can be in *Critical Section*
  - Both can execute the *Test* before one proceeds to execute the line that does the *Set*
- Not *Efficient*: *Busy-waiting*, particularly bad on Uniprocessor (will address this later)

## Implementing *Locks* – v2

Software-based *Lock* – 2nd Attempt

```
// 1: a thread wants to enter critical section, 0: it doesn't
int flag[2] = {0, 0};
```

```
void lock() {
  flag[self] = 1; // I need lock
  while (flag[1- self] == 1); // spin wait
}
```

```
void unlock() {
  // not any more
  flag[self] = 0;
}
```

Idea: Use per-*Thread Flags*, *Set* then *Test*, to achieve *Mutual Exclusion*

Problem?

- Not *Live*: Can enter a *Deadlock*
  - Both can execute the *Set* before one proceeds to *Test*, therefore both will forever *Spin-Wait*
- Not *Efficient*: *Busy-waiting*, particularly bad on Uniprocessor (will address this later)

## Implementing *Locks* – v2

Software-based *Lock* – 3rd Attempt

```
// whose turn is it?
int turn = 0;
```

```
void lock() {
  // wait for my turn
  while (turn == 1 - self); // spin wait
}
```

```
void unlock() {
  // I'm done. your turn
  turn = 1 - self;
}
```

Idea: Strict *Alternation* to achieve *Mutual Exclusion*

Problem?

- Not *Live*: Depends on *Threads* operations outside *Critical Section*
  - *Thread* 1 can go into an infinite loop after its *Critical Section* (after it **`unlock`**s)
  - *Thread* 2 will get to execute once, but then *Thread 1* will never again alternate the **`turn`** over to it

## Implementing *Locks* – v2

Software-based *Lock* – *Peterson's* Algorithm – Final Attempt (combine previous ideas)

```
// whose turn is it?
int turn = 0;
// 1: a thread wants to enter critical section, 0: it doesn't
int flag[2] = {0, 0};
```

```
void lock() {
  flag[self] = 1;  // I need lock
  turn = 1 - self; // wait my turn (set NOT my turn)
  while (flag[1-self] == 1 && turn == 1 - self);
    // spin wait while the other thread has intent
    // AND it is the other thread's turn
}
```

```
void unlock() {
  // not any more
  flag[self] = 0;
}
```

- *Safe*
- *Live:* One of the two will have executed the "*Set: Not My Turn*" operation last, before entering the *Spin-Wait* phase, i.e. the other will proceed

## Implementing *Locks* – v3

*Atomic Operation*-based *Lock*

- Problem with Software-based Lock: Hard to implement for > 2 *Threads*
- Also modern CPUs can perform operations *Out-of-Order* (need *Memory Barrier*)

```
// 0: lock is available, 1: lock is held by a thread
int flag = 0;
```

```
void lock() {
  while( test_and_set( &flag ) );
}
```

```
void unlock() {
  unset( &flag );
}
```

*Note:*
Approach better thought of as "Set-and-Test-Previous-Value"
- If previously 1 (by another *Thread*), will just *Spin-Wait*
  - (and set 1, but irrelevant)
- If previously 0, will *immediately* set 1 and proceed

*Remember:* Problem with the *Test*-**then**-*Set* approach is it is not *Atomic*

*Idea:*
- Make *Atomic* *Test-**and**-Set* :

```
int test_and_set (int *lock) {
  int old = *lock;
  *lock = 1;
  return old;
}
```

Should *Atomically* return prior value of `*lock` and set `*lock` to new value of 1

# Implementing *Locks* – v3

## Implementing **test_and_set** on x86

```
long test_and_set(volatile long* lock) {
  int old;
  asm("xchgl %0, %1"
    : "=r"(old), "+m"(*lock) // output
    : "0"(1)                  // input
    : "cc", "memory"          // … the compiler does not assume that any values read from memory
  );  Note: Not required on x86  // before an asm remain unchanged after that asm; it reloads them
                                 // as needed. … Using the "memory" clobber effectively forms a
                                 // read/write memory barrier for the compiler.
  return old;
}
```

*Note:*
The data that **lock** points to is **volatile** : Disable compiler optimizations (for this object) that can result in a variable being assumed that it does not change outside the scope of the current function (e.g. by an *ISR*, by another *Thread*, etc.), and enforce that it is always read from memory afresh (instead of keeping a cached copy in a temporary *Register*)

- Extended Assembly (https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html)

*Atomic Instruction* **xchg** of **reg** (**old**), **addr**: (***lock**) *Atomically* swaps them

- Most *Spin-Locks* on x86 are implemented using this *Instruction*
  - e.g. xv6 **spinlock.h**, **spinlock.c**, **x86.h**

## Implementing *Locks* – v3

- More modern CPU *Atomic* Instructions unlock more possibilities

*Atomic Compare-and-Swap* (or *Compare-and-Exchange*):

- Checks whether content of memory location matches a value, and if so, modifies it to a new value
- Can now store *Thread ID* of owning *Thread*, instead of just a true/false variable

```
// 0: lock is available, !0: tid of thread holding lock
int tid_lock = 0;
```

```
void lock() {
  while( !compare_and_swap(&tid_lock, 0, gettid()) );
}
```

```
int compare_and_swap (int *addr, int test, int new) {
  if (*addr != test)
    return 0;
  *addr = new;
  return 1;
}
```

*Note:*
MACRO to get current *Thread ID* in Linux
`#define gettid() ((pid_t)syscall(SYS_gettid))`

*Note:*
Can use x86 **`lock cmpxchg`** *Instruction* (the **`lock`** prefix ensures CPU exclusive ownership of Cache line – possibly with a *Memory Bus* Lock)

## *Spin-Waiting* vs *Blocking*

Problem: Waste of CPU cycles

- Worst case scenario: a *Thread* holding a Busy-Wait *Lock* (i.e. is inside the *Critical Section*) gets *Preempted*, while some other *Threads* try to acquire the same *Lock*

On Uniprocessor: Should not use a *Spin-Lock*

- `yield` CPU when *Lock* is not available (need OS support)

On Multiprocessor

- If a *Thread* holding *Lock* gets *Preempted* , the correct action depends on how long before the *Lock* would be released

## *Spin-Waiting* vs *Blocking*

Problem with the simple *Yield*:

```
void lock() {
  while( test_and_set( &flag ) )
    sched_yield();
}
```

- Uncontrollably results in a lot of *Context Switches*
  - *Thundering Herd*
- *Starvation* due to lack of control over which *Thread* gets the *Lock* becomes possible

Why?

- No control over who gets the *Lock* next
- Need explicit control to ensure which *Thread* should get the *Lock*

## Implementing *Locks* – v4

```
// 0: lock is available, 1: lock is held by a thread
int flag = 0;
```

```
void lock() {
  while( test_and_set( &flag ) ) {
    // add myself (back) to wait queue
    sched_yield();
  }
}
```

```
void unlock() {
  unset( &flag );
  if( any_thread_in_wait_queue ) {
    // wake up one thread from wait queue
  }
}
```

*Idea:* Have a *Wait Queue* with those *Threads* that are actually waiting on this specific *Lock*

- (Re-)Add *Thread* to *Wait Queue* while *Lock* remains unavailable
- In `unlock()`, wake up one *Thread* from *Wait Queue*

Problem 1: Lost wakeup

Because it wastes time performing (re-)enqueing itself; in that time another *Thread* reaches the *Test-and-Set* and grabs the *Lock*

- Fix: **Need** the *Spin Lock* to be fast
- *Spin-Wait* will still take place, but should not be expected to be active for too long… *(How?)*

Problem 2: Wrong *Thread* gets *Lock*

- Fix: `unlock()` directly transfers *Lock* to waiting *Thread*
- No other *Thread* should be possible to acquire *Lock*… *(How?)*

## Implementing *Locks* – v4 : mutex

```
typedef struct __mutex_t {
  int guard;  // simple guard lock to avoid losing wakeups
  int flag;   // 0: mutex is available, 1: mutex is not available
  queue_t *queue; // queue of waiting threads
} mutex_t;
```

```
void lock(mutex_t *m) {
  //acquire guard lock by spinning
  while (test_and_set(m->guard));
  if (m->flag == 0) {
    m->flag = 1; // mutex acquired
    unset(m->guard);
  } else {
    enqueue(m->queue, self);
    unset(m->guard);
    sched_yield();
  }
}
```

```
void unlock(mutex_t *m) {
  //acquire guard lock by spinning
  while (test_and_set(m->guard));
  if (empty(m->queue))
    // release mutex; no one wants mutex
    m->flag = 0;
  else
    // direct transfer mutex to next thread
    wakeup( dequeue(m->queue) );
  unset(m->guard);
}
```

## Implementing *Locks* – v4 : **mutex**

- Now the **m->guard** *Lock* is an internal property of the **mutex_t**, i.e. it only protects its inner *Critical Sections* of **lock()** and **unlock()** (between the *Spin-Lock* on **m->guard**, and the line unsetting it to **0**)
  - The *Critical Sections* of **lock()** & **unlock()** is now where the actual marshalling of *Threads* happens, by manipulating the *Mutex* state variables **m->flag** and **m->queue**.

```
void lock(mutex_t *m) {
  //acquire guard lock by spinning
  while (test_and_set(m->guard));
  if (m->flag == 0) {
    m->flag = 1; // mutex acquired
    unset(m->guard);
  } else {
    enqueue(m->queue, self);
    unset(m->guard);
    sched_yield();
  }
}
```

```
void unlock(mutex_t *m) {
 //acquire guard lock by spinning
  while (test_and_set(m->guard));
  if (empty(m->queue))
    // release mutex; no one wants mutex
    m->flag = 0;
  else
    // direct transfer mutex to next thread
    wakeup( dequeue(m->queue) );
  unset(m->guard);
}
```

## *Reader – Writer* Problem

- A *Reader* is a *Thread* that needs to look at the shared data but won't change it
- A *Writer* is a *Thread* that modifies the shared data
  - e.g. making an airline reservation
  - Courtois et al 1971: Concurrent Control with "Readers" and "Writers"

Problem: With the regular *Lock* approach, there is unnecessary *Synchronization*
- Only one *Writer* should be active at a time
- However, any number of *Readers* can be active simultaneously

Solution:
- Acquire *Lock* for *Read Mode* and *Write Mode*

## *Reader – Writer Lock*

```
rwlock_t lock;
```

```
void* writer(void *arg) {
  while(true) {
    write_lock(&lock);
    …
    // write shared data
    …
    write_unlock(&l);
  }
}
```

```
void* reader(void *arg) {
  while(true) {
    read_lock(&lock);
    …
    // read shared data
    …
    read_unlock(&lock);
  }
}
```

`read_lock`: Acquires *Lock* in *Read* (*Shared Access*) *Mode*

- If *Lock* is not acquired or in *Read Mode* → Success
- Otherwise, *Lock* is in *Write Mode* → Wait

`write_lock`: Acquires *Lock* in *Write* (*Exclusive Access*) *Mode*

- If *Lock* is not acquired → Success
- Otherwise → Wait

## Implementing *Reader – Writer Lock*

```
struct rwlock_t {
  int nreader;       // init to 0
  lock_t guard;      // init to unlocked
  lock_t datalock;   // init to unlocked
};
```

```
void write_lock(rwlock_t *l) {
  lock(&l->datalock);
}
void write_unlock(rwlock_t *l) {
  unlock(&l->datalock);
}
```

```
void read_lock(rwlock_t *l) {
  lock(&l->guard);
  ++ nreader;
  if(nreader == 1) // 1 reader, no more writing
    lock(&l->datalock);
  unlock(&l->guard);
}
void read_unlock(rwlock_t *l) {
  lock(&l->guard);
  -- nreader;
  if(nreader == 0) // 0 readers, can write
    unlock(&l->datalock);
  unlock(&l->guard);
}
```

Problem:

- *Writer Starvation* is possible

**CS-446/646**

Time for Questions !