

CS350: Operating Systems

Lecture 5: Synchronization

Ali Mashtizadeh

University of Waterloo

Outline

- 1 Synchronization and memory consistency review
- 2 C11 Atomics
- 3 Cache coherence – the hardware view
- 4 Improving spinlock performance
- 5 MCS Locks
- 6 Deadlock
- 7 OS Implementation

Motivation

$$T(n) = T(1) \left(B + \frac{1}{n}(1 - B) \right)$$

- Amdahl's law
 - ▶ $T(1)$: the time one core takes to complete the task
 - ▶ B : the fraction of the job that must be serial
 - ▶ n : the number of cores
- Suppose n were infinity!
- Amdahl's law places an ultimate limit on parallel speedup
- Problem: synchronization increases serial section size
- Scalable Commutativity Rule: *“Whenever interface operations commute, they can be implemented in a way that scales”* [Clements]

Locking Basics

```
pthread_mutex_t m;  
  
pthread_mutex_lock(&m);  
cnt = cnt + 1; /* critical section */  
pthread_mutex_unlock(&m);
```

- Only one thread can hold a lock at a time
- Makes critical section atomic
- When do you need a lock?
 - ▶ Anytime two or more threads touch data and at least one writes
- Rule: Never touch data unless you hold the right lock

Fine-grained Locking

```
struct list_head *hash_tbl[1024];

/* Coarse-grained Locking */
mutex_t m;
mutex_lock(&m);
struct list_head *pos = hash_tbl[hash(key)];
/* walk list and find entry */
mutex_unlock(&m);

/* Fine-grained Locking */
mutex_t bucket[1024];
int index = hash(key);
mutex_lock(&bucket[index]);
struct list_head *pos = hash_tbl[index];
/* walk list and find entry */
mutex_unlock(&bucket[index]);
```

- Which of these is better?

Memory reordering danger

- Suppose no sequential consistency & don't compensate
- Hardware could violate program order

<u>Program order on CPU #1</u>	<u>View on CPU #2</u>
<pre>read/write: v->lock = 1; read: register = v->val; write: v->val = register + 1; write: v->lock = 0;</pre>	<pre>v->lock = 1; v->lock = 0; /* danger */ v->val = register + 1;</pre>

- If `atomic_inc` called at `/* danger */`, bad `val` ensues!

Ordering requirements

```
void atomic_inc (var *v) {  
    while (test_and_set (&v->lock))  
        ;  
    v->val++;  
    /* danger */  
    v->lock = 0;  
}
```

- Must ensure all CPUs see the following:
 1. `v->lock` was set *before* `v->val` was read and written
 2. `v->lock` was cleared *after* `v->val` was written
- How does #1 get assured on x86?
 - ▶ Recall `test_and_set` uses `xchgl %eax, (%edx)`
- How to ensure #2 on x86?

Ordering requirements

```
void atomic_inc (var *v) {  
    while (test_and_set (&v->lock))  
        ;  
    v->val++;  
    /* danger */  
    v->lock = 0;  
}
```

- Must ensure all CPUs see the following:
 1. `v->lock` was set *before* `v->val` was read and written
 2. `v->lock` was cleared *after* `v->val` was written
- How does #1 get assured on x86?
 - ▶ Recall `test_and_set` uses `xchgl %eax, (%edx)`
 - ▶ `xchgl` instruction always “locked,” ensuring barrier
- How to ensure #2 on x86?

Ordering requirements

```
void atomic_inc (var *v) {  
    while (test_and_set (&v->lock))  
        ;  
    v->val++;  
    asm volatile ("sfence" ::: "memory");  
    v->lock = 0;  
}
```

- Must ensure all CPUs see the following:
 1. `v->lock` was set *before* `v->val` was read and written
 2. `v->lock` was cleared *after* `v->val` was written
- How does #1 get assured on x86?
 - ▶ Recall `test_and_set` uses `xchgl %eax, (%edx)`
 - ▶ `xchgl` instruction always “locked,” ensuring barrier
- How to ensure #2 on x86?
 - ▶ Might need fence instruction after, e.g., non-temporal stores

Spinlocks

```
void Spinlock_Lock(Spinlock *lock)
{
    /* Disable Interrupts */
    Critical_Enter();

    while (atomic_swap_uint64(&lock->lock, 1) == 1)
        /* Spin! */;

    lock->cpu = CPU();
}
void Spinlock_Unlock(Spinlock *lock)
{
    ASSERT(lock->cpu == CPU());

    atomic_set_uint64(&lock->lock, 0);

    /* Re-enable Interrupts (if not spinlocks held) */
    Critical_Exit();
}
```

Outline

- 1 Synchronization and memory consistency review
- 2 C11 Atomics
- 3 Cache coherence – the hardware view
- 4 Improving spinlock performance
- 5 MCS Locks
- 6 Deadlock
- 7 OS Implementation

Atomics and Portability

- Lots of variation in atomic instructions, consistency models, compiler behavior
- Results in complex code when writing portable kernels and applications
- Still a big problem today: Your laptop is x86, your cell phone is ARM
 - ▶ x86: Total Store Order Consistency Model, CISC
 - ▶ ARM: Relaxed Consistency Model, RISC
- Fortunately, the new C11 standard has builtin support for atomics
 - ▶ Enable in GCC with the `-std=c11` flag
- Also available in C++11, but not discussed today...

C11 Atomics: Basics

- Portable support for synchronization
- New atomic type: e.g., `_Atomic(int) foo`
 - ▶ All standard ops (e.g., `+`, `-`, `/`, `*`) become sequentially consistent
 - ▶ Plus new intrinsics available (`cmpxchg`, atomic increment, etc.)
- `atomic_flag` is a special type
 - ▶ Atomic boolean value without support for loads and stores
 - ▶ Must be implemented lock-free
 - ▶ All other types might require locks, depending on the size and architecture
- Fences also available to replace hand-coded memory barrier assembly

Example 1: Atomic Counters

```
_Atomic(int) packet_count;  
void recv_packet(...) {  
    ...  
    atomic_fetch_add(&packet_count, 1);  
    ...  
}
```

Example 2: Producer, Consumer

```
struct message msg_buf;
_Atomic(bool) msg_ready;

void send(struct message *m) {
    msg_buf = *m;
    atomic_store(&msg_ready, true);
}

struct message *recv(void) {
    bool ready = atomic_load(&msg_ready);
    if (!ready)
        return NULL;
    return &msg_buf;
}
```

Example 3: A Spinlock

- Spinlocks are similar to Mutexes
- Kernels use these for small critical regions
 - ▶ Busy wait for others to release the lock
 - ▶ No sleeping and yielding to other Threads

```
void spin_lock(atomic_flag *lock) {  
    while (atomic_flag_test_and_set(lock))  
        /* Spin! */;  
}
```

```
void spin_unlock(atomic_flag *lock) {  
    atomic_flag_clear(lock);  
}
```


Outline

- 1 Synchronization and memory consistency review
- 2 C11 Atomics
- 3 Cache coherence – the hardware view
- 4 Improving spinlock performance
- 5 MCS Locks
- 6 Deadlock
- 7 OS Implementation

Overview

- Coherence
 - ▶ concerns accesses to a single memory location
 - ▶ makes sure stale copies do not cause problems
- Consistency
 - ▶ concerns apparent ordering between multiple locations

Multicore Caches

- Performance requires caches
- Caches create an opportunity for cores to disagree about memory
- Bus-based approaches
 - ▶ “Snoopy” protocols, each CPU listens to memory bus
 - ▶ Use write through and invalidate when you see a write bits
 - ▶ Bus-based schemes limit scalability
- Modern CPUs use networks (e.g., hypertransport, UPI)
- Cache is divided into chunks of bytes called *cache lines*
 - ▶ 64-bytes is a typical size

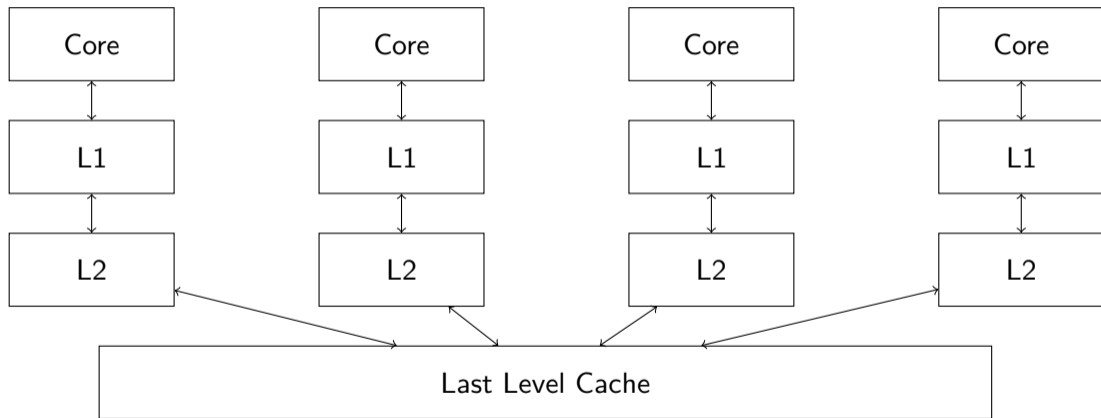
3-state Coherence Protocol (MSI)

- Each cache line is one of three states:
- Modified (sometimes called Exclusive)
 - ▶ One cache has a valid copy
 - ▶ That copy is stale (needs to be written back to memory)
 - ▶ Must invalidate all copies before entering this state
- Shared
 - ▶ One or more caches (and memory) have a valid copy
- Invalid
 - ▶ Doesn't contain any data
- Transitions can take 100–2000 cycles

Core and Bus Actions

- Core has three actions:
- Read (load)
 - ▶ Read without intent to modify, data can come from memory or another cache
 - ▶ Cacheline enters shared state
- Write (store)
 - ▶ Read with intent to modify, must invalidate all other cache copies
 - ▶ Cacheline in shared (some protocols have an exclusive state)
- Evict
 - ▶ Writeback contents to memory if modified
 - ▶ Discard if in shared state
- Performance problem:
 - ▶ Every transition requires bus communications
 - ▶ Avoid state transitions whenever possible

Core-to-Core Communications



Real World Coherence Costs

- See [David] for a great reference, summarized here...
 - ▶ Intel Xeon: 3 cycle L1, 11 cycle L2, 44 cycle LLC, 355 cycle RAM
- Remote core holds modified line state:
 - ▶ load: 109 cycles (LLC + 65)
 - ▶ store: 115 cycles (LLC + 71)
 - ▶ atomic CAS: 120 cycles (LLC + 76)
 - ▶ NUMA load: 289 cycles
 - ▶ NUMA store: 320 cycles
 - ▶ NUMA atomic CAS: 324 cycles
- But only a partial picture
 - ▶ Could be faster because of out-of-order execution
 - ▶ Could be slower because of bus contention or multiple hops

Implications for Multithreaded Design

- Lesson #1: Avoid false sharing
 - ▶ Processor shares data in cache line chunks
 - ▶ Avoid placing data used by different threads in the same cache line
- Lesson #2: Align structures to cache lines
 - ▶ Place related data you need to access together
 - ▶ Alignment in C11/C++11: `alignas(64) struct foo f;`
- Lesson #3: Pad data structures
 - ▶ Arrays of structures lead to false sharing
 - ▶ Add unused fields to ensure alignment
- Lesson #4: Avoid contending on cache lines
 - ▶ Reduce costly cache coherence traffic
 - ▶ Advanced algorithms spin on a cache line local to a core (e.g., MCS Locks)

Outline

- 1 Synchronization and memory consistency review
- 2 C11 Atomics
- 3 Cache coherence – the hardware view
- 4 Improving spinlock performance
- 5 MCS Locks
- 6 Deadlock
- 7 OS Implementation

Improving Spinlocks

- Test-and-set spinlock has several advantages
 - ▶ Simple to implement and understand
 - ▶ One memory location for arbitrarily many CPUs
- But also has disadvantages
 - ▶ Lots of traffic over memory bus (especially when > 1 spinner)
 - ▶ Not necessarily fair (same CPU acquires lock many times)
 - ▶ Even less fair on a NUMA machine
 - ▶ Allegedly Google had fairness problems even on Opterons
- Idea 1: Avoid spinlocks altogether
 - ▶ Lock free algorithms, RCU, transactional memory
- Idea 2: Reduce bus traffic with better spinlocks
 - ▶ Design lock that spins only on local memory
 - ▶ Also gives better fairness

Spinlocks

```
void Spinlock_Lock(Spinlock *lock)
{
    while (atomic_swap_uint64(&lock->lock, 1) == 1)
        /* Spin! */;

    lock->cpu = CPU();
}
```

```
void Spinlock_Unlock(Spinlock *lock)
{
    ASSERT(lock->cpu == CPU());

    atomic_set_uint64(&lock->lock, 0);
}
```

Reduce the Swap/Test-and-set Traffic

- How do we reduce bus traffic?
- Swap/Test-and-Set requires invalidating all copies
- $O(N)$ messages on each attempt
- Waiting cores just generate bus traffic!
- Main idea: spinning on a shared cache line is cheaper

Reduce Bus Traffic with Test Test-and-Set

```
void Spinlock_Lock(Spinlock *lock)
{
    while (1)
        while (lock->lock == 1)
            /* Spin! */;

    if (atomic_swap_uint64(&lock->lock, 1) == 0) {
        lock->cpu = CPU();
        return;
    }
}
```

```
void Spinlock_Unlock(Spinlock *lock)
{
    atomic_set_uint64(&lock->lock, 0);
}
```

How Does Test Test-and-Set Work?

- Spin on a load checking the value
 - ▶ Initial load requires communications to enter shared state
 - ▶ Every core can enter the shared state
 - ▶ Spinning on a load causes no traffic
- Releasing the lock
 - ▶ Stores 0 to the lock, invalidating everyone's copy
- Multiple cores re-issue load and see its free
- One or more attempt to acquire lock
 - ▶ Again, requires invalidating everyone's copy

Comparing with and without the Additional Test

- Increases latency in contended case
 - ▶ Bad if we expect locks to be mostly uncontended
 - ▶ Usually requires upgrading from shared to exclusive
- Traffic on unlock unmodified $O(n)$
- Wait traffic reduced to zero

Outline

- 1 Synchronization and memory consistency review
- 2 C11 Atomics
- 3 Cache coherence – the hardware view
- 4 Improving spinlock performance
- 5 MCS Locks
- 6 Deadlock
- 7 OS Implementation

MCS lock

- Idea 2: Build a better spinlock
- Lock designed by [Mellor-Crummey and Scott](#)
 - ▶ Goal: reduce bus traffic on cc machines, improve fairness
- Each CPU has a qnode structure in local memory

```
typedef struct qnode {  
    struct qnode *next;  
    bool locked;  
} qnode;
```

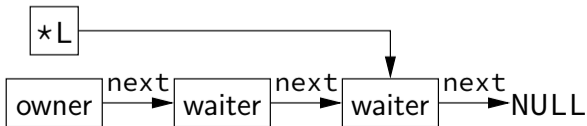
- ▶ Local can mean local memory in NUMA machine
 - ▶ Or just its own cache line that gets cached in exclusive mode
- A lock is just a pointer to a qnode

```
typedef qnode *lock;
```
- Lock is list of CPUs holding or waiting for lock
- While waiting, spin on *your local* locked flag

MCS Acquire

```
acquire(lock *L, qnode *I) {  
    I->next = NULL;  
    qnode *predecessor = I;  
    XCHG(predecessor, *L); /* atomic swap */  
    if (predecessor != NULL) {  
        I->locked = true;  
        predecessor->next = I;  
        while (I->locked)  
            ;  
    }  
}
```

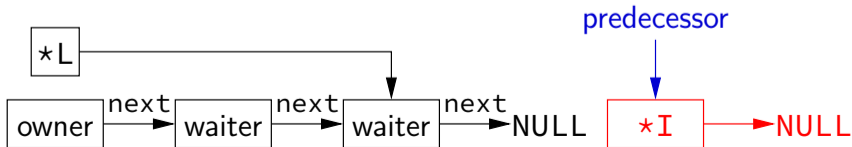
- If unlocked, L is NULL
- If locked, no waiters, L is owner's qnode
- If waiters, *L is tail of waiter list:



MCS Acquire

```
acquire(lock *L, qnode *I) {  
    I->next = NULL;  
    qnode *predecessor = I;  
    XCHG(predecessor, *L); /* atomic swap */  
    if (predecessor != NULL) {  
        I->locked = true;  
        predecessor->next = I;  
        while (I->locked)  
            ;  
    }  
}
```

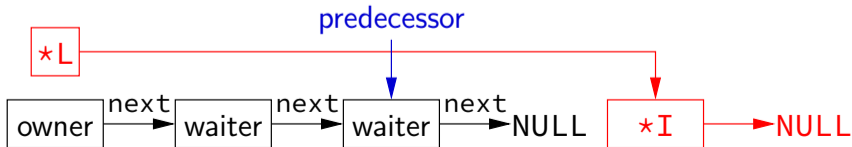
- If unlocked, L is NULL
- If locked, no waiters, L is owner's qnode
- If waiters, *L is tail of waiter list:



MCS Acquire

```
acquire(lock *L, qnode *I) {  
    I->next = NULL;  
    qnode *predecessor = I;  
    XCHG(predecessor, *L); /* atomic swap */  
    if (predecessor != NULL) {  
        I->locked = true;  
        predecessor->next = I;  
        while (I->locked)  
            ;  
    }  
}
```

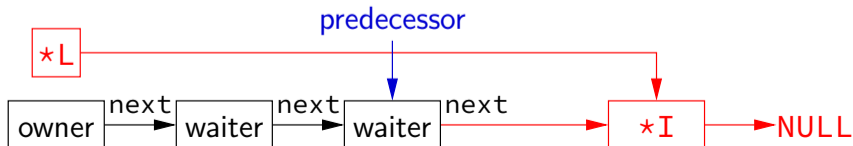
- If unlocked, L is NULL
- If locked, no waiters, L is owner's qnode
- If waiters, *L is tail of waiter list:



MCS Acquire

```
acquire(lock *L, qnode *I) {  
    I->next = NULL;  
    qnode *predecessor = I;  
    XCHG(predecessor, *L); /* atomic swap */  
    if (predecessor != NULL) {  
        I->locked = true;  
        predecessor->next = I;  
        while (I->locked)  
            ;  
    }  
}
```

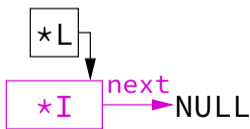
- If unlocked, L is NULL
- If locked, no waiters, L is owner's qnode
- If waiters, *L is tail of waiter list:



MCS Release with CAS

```
release(lock *L, qnode *I) {  
    if (!I->next)  
        if (CAS(*L, I, NULL))  
            return;  
    while (!I->next)  
        ;  
    I->next->locked = false;  
}
```

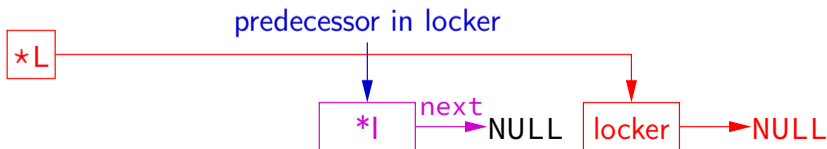
- If $I \rightarrow \text{next} \text{ NULL}$ and $*L == I$
 - ▶ No one else is waiting for lock, OK to set $*L = \text{NULL}$



MCS Release with CAS

```
release(lock *L, qnode *I) {  
    if (!I->next)  
        if (CAS(*L, I, NULL))  
            return;  
    while (!I->next)  
        ;  
    I->next->locked = false;  
}
```

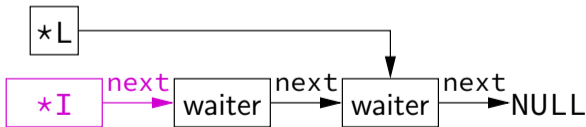
- If $I \rightarrow \text{next}$ NULL and $*L \neq I$
 - ▶ Another thread is in the middle of acquire
 - ▶ Just wait for $I \rightarrow \text{next}$ to be non-NULL



MCS Release with CAS

```
release(lock *L, qnode *I) {  
    if (!I->next)  
        if (CAS(*L, I, NULL))  
            return;  
    while (!I->next)  
        ;  
    I->next->locked = false;  
}
```

- If $I \rightarrow \text{next}$ is non-NULL
 - ▶ $I \rightarrow \text{next}$ oldest waiter, wake up w. $I \rightarrow \text{next} \rightarrow \text{locked} = \text{false}$



Outline

- 1 Synchronization and memory consistency review
- 2 C11 Atomics
- 3 Cache coherence – the hardware view
- 4 Improving spinlock performance
- 5 MCS Locks
- 6 **Deadlock**
- 7 OS Implementation

The deadlock problem

```
mutex_t m1, m2;
```

```
void f1(void *ignored) {  
    lock(m1);  
    lock(m2);  
    /* critical section */  
    unlock(m2);  
    unlock (m1);  
}
```

```
void f2 (void *ignored) {  
    lock(m2);  
    lock(m1);  
    /* critical section */  
    unlock(m1);  
    unlock(m2);  
}
```

- Lesson: Dangerous to acquire locks in different orders

More deadlocks

- Same problem with condition variables
 - ▶ Suppose resource 1 managed by c_1 , resource 2 by c_2
 - ▶ A has 1, waits on c_2 , B has 2, waits on c_1

- Or have combined mutex/condition variable deadlock:

```
mutex_t a, b;
```

```
cond_t c;
```

- `lock(a); lock(b); while (!ready) wait(b, c);
unlock(b); unlock (a);`
- `lock(a); lock(b); ready = true; signal(c);
unlock(b); unlock(a);`

- Lesson: Dangerous to hold locks when crossing abstraction barriers!
 - ▶ I.e., `lock (a)` then call function that uses condition variable

Deadlock conditions

1. Limited access (mutual exclusion):
 - ▶ Resource can only be shared with finite users
2. No preemption:
 - ▶ Once resource granted, cannot be taken away
3. Multiple independent requests (hold and wait):
 - ▶ Don't ask all at once
(wait for next resource while holding current one)
4. Circularity in graph of requests
 - All of 1–4 necessary for deadlock to occur
 - Two approaches to dealing with deadlock:
 - ▶ Pro-active: prevention
 - ▶ Reactive: detection + corrective action

Prevent by eliminating one condition

1. Limited access (mutual exclusion):
 - ▶ Buy more resources, split into pieces, or virtualize to make "infinite" copies
 - ▶ Threads: threads have copy of registers = no lock
2. No preemption:
 - ▶ Physical memory: virtualized with VM, can take physical page away and give to another process!
3. Multiple independent requests (hold and wait):
 - ▶ Wait on all resources at once (must know in advance)
4. **Circularity in graph of requests**
 - ▶ Single lock for entire system: (problems?)
 - ▶ Partial ordering of resources (next)

Cycles and deadlock

- View system as graph
 - ▶ Processes and Resources are nodes
 - ▶ Resource Requests and Assignments are edges
- If graph has no cycles → no deadlock
- If graph contains a cycle
 - ▶ Definitely deadlock if only one instance per resource
 - ▶ Otherwise, maybe deadlock, maybe not
- Prevent deadlock with partial order on resources
 - ▶ E.g., always acquire mutex m_1 before m_2
 - ▶ Statically assert lock ordering (e.g., VMware ESX)
 - ▶ Dynamically find potential deadlocks [[Witness](#)]

Outline

- 1 Synchronization and memory consistency review
- 2 C11 Atomics
- 3 Cache coherence – the hardware view
- 4 Improving spinlock performance
- 5 MCS Locks
- 6 Deadlock
- 7 OS Implementation

Wait Channels

- OS synchronization (except spinlocks) use wait channels
 - ▶ Manages a list of sleeping threads
 - ▶ Abstracts details of the thread scheduler
- `void WaitChannel_Lock(WaitChannel *wc);`
 - ▶ Lock wait channel operations
 - ▶ Prevents a race between sleep and wake
- `void WaitChannel_Sleep(WaitChannel *wc);`
 - ▶ Blocks calling thread on wait channel `wc`
 - ▶ Causes a context switch (e.g., `thread_yield`)
- `void WaitChannel_WakeAll(WaitChannel *wc);`
 - ▶ Unblocks all threads sleeping on the wait channel
- `void WaitChannel_Wake(WaitChannel *wc);`
 - ▶ Unblocks one threads sleeping on the wait channel

Hand-over-Hand Locking

- Hand-over-Hand Locking allows for fine-grained locking
- Useful for concurrent data structure manipulation
 - ▶ Hold at most two locks the previous and next locks
 - ▶ Locks must be ordered in a sequence
 - ▶ You may not go backwards. Why?
- Example: we have locks A, B, C

```
lock(A)
// Operate on A
lock(B)
unlock(A)
// Operate on B
lock(C)
unlock(B)
// Operate on C
unlock(C)
```

Example Semaphores

- Mutexes, CVs and Semaphores use WaitChannel and Hand-over-Hand Locking
- Lock order: sem_lock Spinlock, WaitChannel Lock

```
typedef struct Semaphore {  
    int          sem_count;  
    Spinlock    *sem_lock;  
    WaitChannel *sem_wchan;  
} Semaphore;
```

Example: Semaphores Implementation

```
Semaphore_Wait(Semaphore *sem) {
    Spinlock_Lock(&sem->sem_lock);
    while (sem->sem_count == 0) {
        /* Locking the wchan prevents a race on sleep */
        WaitChannel_Lock(sem->sem_wchan);
        /* Release spinlock before sleeping */
        Spinlock_Unlock(&sem->sem_lock);
        /* Wait channel protected by it's own lock */
        WaitChannel_Sleep(sem->sem_wchan);
        /* Recheck condition, no locks held */
        Spinlock_Lock(&sem->sem_lock);
    }
    sem->sem_count--;
    Spinlock_Unlock(&sem->sem_lock);
}

Semaphore_Post(Semaphore *sem) {
    Spinlock_Lock(&sem->sem_lock);
    sem->sem_count++;
    WaitChannel_Wake(sem->sem_wchan);
    Spinlock_Unlock(&sem->sem_lock);
}
```