Synchronization II

COMS W4118

Prof. Kaustubh R. Joshi

krj@cs.columbia.edu

http://www.cs.columbia.edu/~krj/os

References: Operating Systems Concepts (9e), Linux Kernel Development, previous W4118s

Problems with Locks

- Low level
  - Users must remember to lock/unlock
  - All it takes is one forgetful programmer
    - Unlock without lock: no mutual exclusion
    - Lock without unlock: deadlock
- How to handle multiple resources
  - Have n resources of same type
  - n threads can access concurrently
  - How to co-ordinate?
  - Can't nest locks (one thread, multiple resources?)
- How to enforce ordering?
  - E.g., producers, consumers, pipelines

Higher Level Synchronization Constructs

- Atomic Variables
  - Allow race-free manipulation of simple variables
  - Very useful for kernel programming
- Semaphores
  - Easy coordination for multi-resource and ordering situations
- Monitors
  - Language level constructs
  - Free users from having to worry about synchronization
  - Ensure correct usage
- (Software) Transactional Memory
  - Need compiler support
  - Fine-grained critical sections with compiler support
  - Allow DB-techniques like optimistic execution and rollback

Outline

- Atomic Variables
- Semaphores
- Monitors and condition variables
- Transactional Memory
- Linux Synchronization Primitives

Atomic Operations

- Many instructions not atomic in hardware (smp)
  - Read-modify-write instructions: inc, test-and-set, swap
  - unaligned memory access
- Compiler may not generate atomic code
  - even i++ is not necessarily atomic!
- If the data that must be protected is a single word, atomic operations can be used. These functions examine and modify the word atomically.
- The atomic data type is *atomic_t*.
- Intel implementation
  - lock prefix byte 0xf0 – locks memory bus

Atomic Operations

ATOMIC_INIT – initialize an *atomic_t* variable (integer)
atomic_read – examine value atomically
atomic_set – change value atomically
atomic_inc – increment value atomically
atomic_dec – decrement value atomically
atomic_add - add to value atomically
atomic_sub – subtract from value atomically
atomic_inc_and_test – increment value and test for zero
atomic_dec_and_test – decrement value and test for zero
atomic_sub_and_test – subtract from value and test for zero

64 bit integer and bitwise operations are also available (see LKD 10)

Outline

- Atomic Variables
- Semaphores
- Monitors and condition variables
- Transactional Memory
- Linux Synchronization Primitives

Semaphore motivation

- Problem with lock: ensures mutual exclusion, but no execution order
- Producer-consumer problem: need to enforce execution order
  - Producer: create resources
  - Consumer: use resources
  - bounded buffer between them
  - Execution order: producer waits if buffer full, consumer waits if buffer empty
  - E.g., $ cat 1.txt | sort | uniq | wc

Semaphore definition

- A synchronization variable that contains an integer value
  - Can't access this integer value directly
  - Must initialize to some value
    - sem_init (sem_t *s, int pshared, unsigned int value)
  - Has two operations to manipulate this integer
    - sem_wait (or down(), P())
    - sem_post (or up(), V())

```
int sem_wait(sem_t *s) {
    wait until value of semaphore s
        is greater than 0
    decrement the value of
        semaphore s by 1
}
```

```
int sem_post(sem_t *s) {
    increment the value of
        semaphore s by 1
    if there are threads waiting, wake
        up one
}
```

Semaphore uses: mutual exclusion

- Mutual exclusion
  - Semaphore as mutex
  - Binary semaphore: X=1

```
// initialize to X
sem_init(s, 0, X)

sem_wait(s);
// critical section
sem_post(s);
```

- Mutual exclusion with more than one resources
  - Counting semaphore: X>1
  - Initialize to be the number of available resources

Semaphore uses: execution order

- Execution order
  - One thread waits for another
  - What should initial value be?

```
//thread 0

... // 1st half of computation

sem_post(s);
```

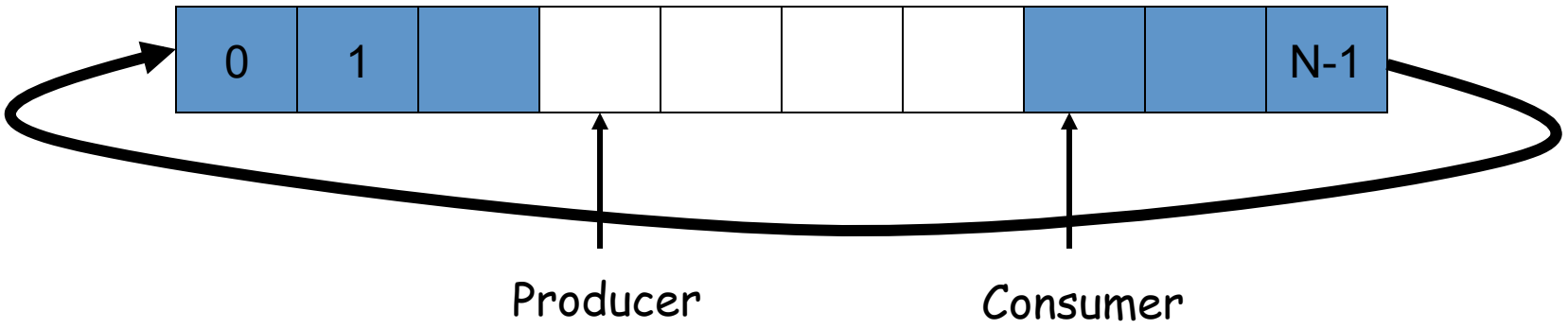
```
// thread 1

sem_wait(s);

... //2nd half of computation
```

How to implement semaphores?

- Exercise
- Q: can we build on top of locks?

Producer-Consumer (Bounded-Buffer) Problem

- Bounded buffer: size N, Access entry 0… N-1, then "wrap around" to 0 again
- Producer process writes data to buffer
- Consumer process reads data from buffer
- Execution order constraints
  - Producer shouldn't try to produce if buffer is full
  - Consumer shouldn't try to consume if buffer is empty



Solving Producer-Consumer problem

- Two semaphores
  - sem_t full; // # of filled slots
  - sem_t empty; // # of empty slots
- What should initial values be?
- Problem: mutual exclusion?

```
sem_init(&full, 0, X);
sem_init(&empty, 0, Y);
```

```
producer() {
    sem_wait(empty);
    … // fill a slot
    sem_post(full);
}
```

```
consumer() {
    sem_wait(full);
    … // empty a slot
    sem_post(empty);
}
```

Solving Producer-Consumer problem: final

- Three semaphores
  - sem_t full; // # of filled slots
  - sem_t empty; // # of empty slots
  - sem_t mutex; // mutual exclusion

```
sem_init(&full, 0, 0);
sem_init(&empty, 0, N);
sem_init(&mutex, 0, 1);
```

```
producer() {
    sem_wait(empty);
    sem_wait(&mutex);
    … // fill a slot
    sem_post(&mutex);
    sem_post(full);
}
```

```
consumer() {
    sem_wait(full);
    sem_wait(&mutex);
    … // empty a slot
    sem_post(&mutex);
    sem_post(empty);
}
```

Outline

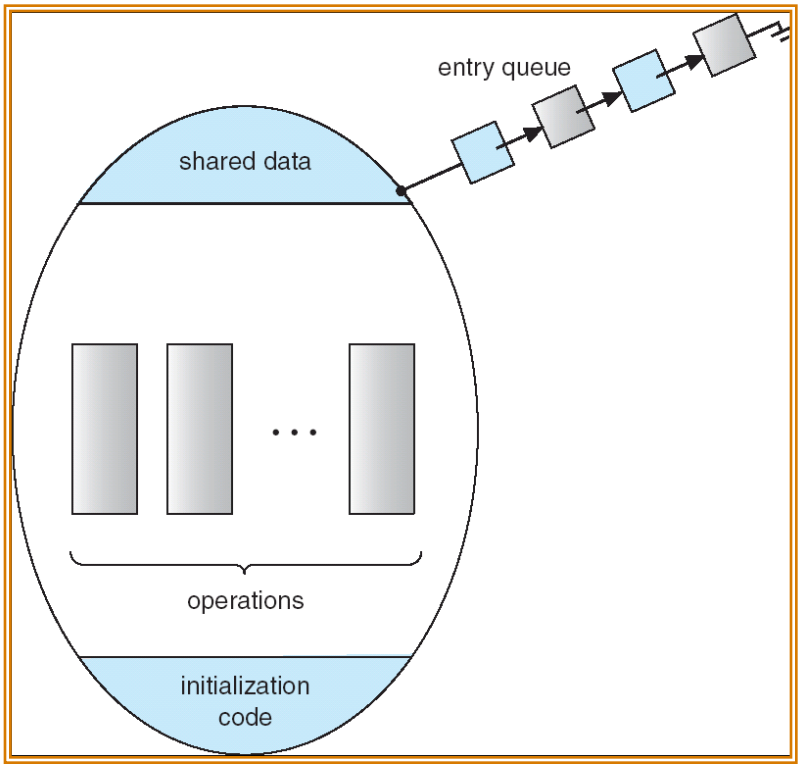
- Atomic Variables
- Semaphores
- Monitors and condition variables
- Transactional Memory
- Linux Synchronization Primitives

Monitors

- Background: concurrent programming meets object-oriented programming
  - When concurrent programming became a big deal, object-oriented programming too
  - People started to think about ways to make concurrent programming more structured
- Monitor: object with a set of monitor procedures and only one thread may be active (i.e. running one of the monitor procedures) at a time

Schematic view of a monitor

- Can think of a monitor as one big lock for a set of operations/ methods
- In other words, a language implementation of mutexes



How to implement monitor?

Compiler automatically inserts lock and unlock operations upon entry and exit of monitor procedures

```
class account {
    int balance;
    public synchronized void deposit() {
        ++balance;
    }
    public synchronized void withdraw() {
        --balance;
    }
};
```

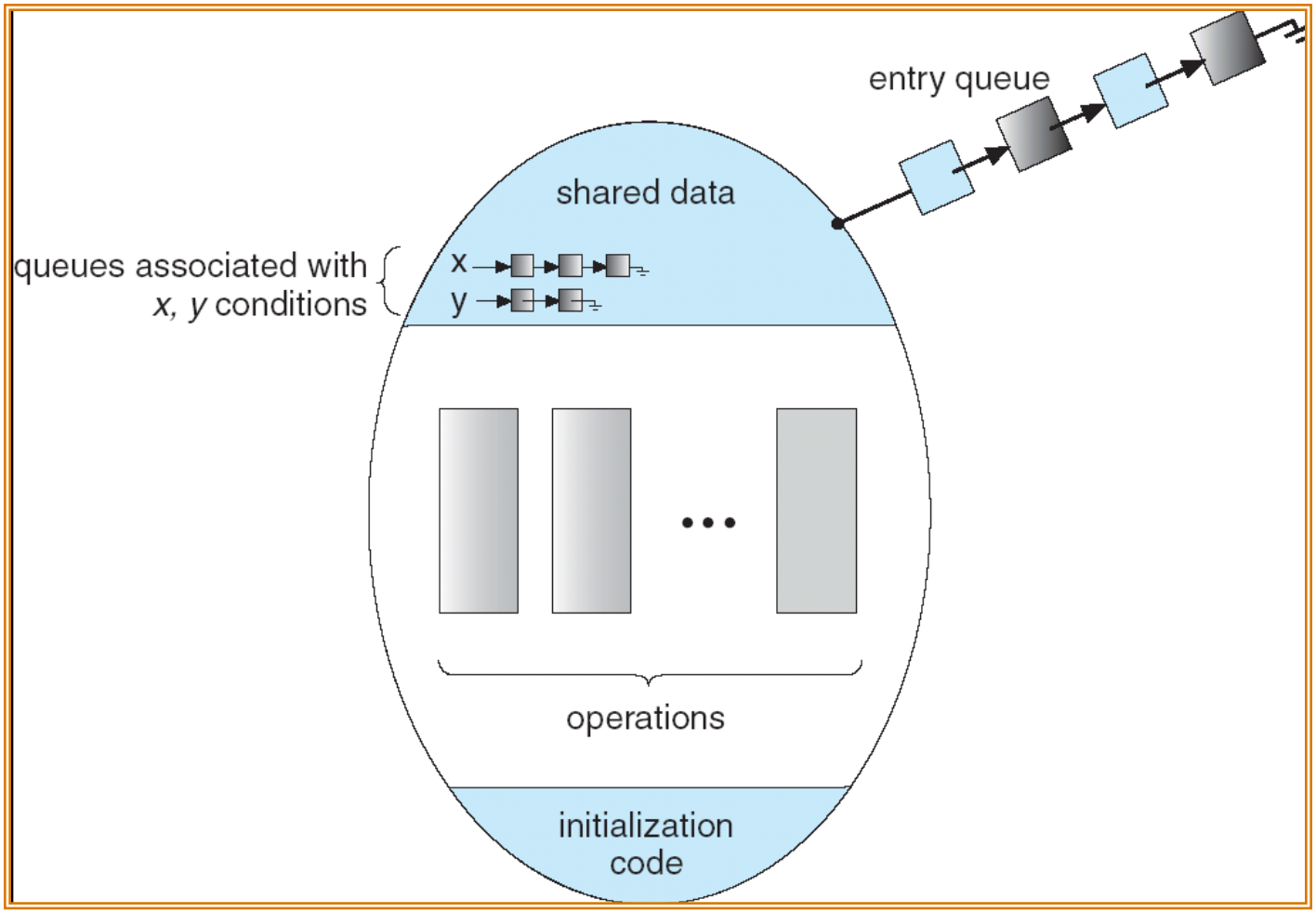
```
lock(this.m);
++balance;
unlock(this.m);
```

```
lock(this.m);
--balance;
unlock(this.m);
```

Condition Variables

- Need wait and wakeup as in semaphores
- Monitor uses Condition Variables
  - Conceptually associated with some conditions
- Operations on condition variables:
  - wait(): suspends the calling thread and releases the monitor lock. When it resumes, reacquire the lock. Called when condition is not true
  - signal(): resumes one thread waiting in wait() if any. Called when condition becomes true and wants to wake up one waiting thread
  - broadcast(): resumes all threads waiting in wait(). Called when condition becomes true and wants to wake up all waiting threads

Monitor with condition variables



Condition variables vs. semaphores

- Semaphores are sticky: they have memory, sem_post() will increment the semaphore counter, even if no one has called sem_wait()
- Condition variables are not: if no one is waiting for a signal(), this signal() is not saved
- Despite the difference, they are as powerful
  - Exercise: implement one using the other

Producer-consumer with monitors

```
monitor ProducerConsumer {
    int nfull = 0;
    cond has_empty, has_full;

    producer() {
       if (nfull == N)
           wait (has_empty);
       … // fill a slot
       ++ nfull;
       signal (has_full);
    }

    consumer() {
       if (nfull == 0)
            wait (has_full);
       … // empty a slot
       -- nfull;
       signal (has_empty);
    }
};
```

- Two condition variables
  - has_empty: buffer has at least one empty slot
  - has_full: buffer has at least one full slot
- nfull: number of filled slots
  - Need to do our own counting for condition variables

Condition variable semantics

- Design question: when signal() wakes up a waiting thread, which thread to run inside the monitor, the signaling thread, or the waiting thread?
- Hoare semantics: suspends the signaling thread, and immediately transfers control to the woken thread
  - Difficult to implement in practice
- Mesa semantics: signal() moves a single waiting thread from the blocked state to a runnable state, then the signaling thread continues until it exits the monitor
  - Easy to implement
  - Problem: race! Before a woken consumer continues, another consumer comes in and grabs the buffer

Fixing the race in mesa monitors

```
monitor ProducerConsumer {
    int nfull = 0;
    cond has_empty, has_full;

    producer() {
        while (nfull == N)
            wait (has_empty);
        … // fill slot
        ++ nfull;
        signal (has_full);
    }

    consumer() {
        while (nfull == 0)
            wait (has_full);
        … // empty slot
        -- nfull
        signal (has_empty);
    }
};
```

- The fix: when woken up, a thread must recheck the condition it was waiting on
- Most systems use mesa semantics
  - E.g., pthread
- You should use while!

Monitor and condition variable in pthread

```
class ProducerConsumer {
    int nfull = 0;
    pthread_mutex_t m;
    pthread_cond_t has_empty, has_full;

public:
    producer() {
      pthread_mutex_lock(&m);
       while (nfull == N)
           pthread_cond_wait (&has_empty, &m);
       … // fill slot
       ++ nfull;
       pthread_cond_signal (has_full);
       pthread_mutex_unlock(&m);
    }
    …
};
```

- C/C++ don't provide monitors; but we can implement monitors using pthread mutex and condition variable
- For producer-consumer problem, need 1 pthread mutex and 2 pthread condition variables (pthread_cond_t)
- Manually lock and unlock mutex for monitor procedures
- pthread_cond_wait (cv, m): atomically waits on cv and releases m

Outline

- Atomic Variables
- Semaphores
- Monitors and condition variables
- Transactional Memory
- Linux Synchronization Primitives

Transactional Memory

- Problem: locks have a fundamentally pessimistic worldview
  - Assume conflict will happen when reading/writing and try to prevent it
  - Leads to poor scalability when lots of cores
- Transactional memory proposes optimistic view
  - Assume that conflict won't usually happen
  - Read/update shared data without locking
  - After operation is done, check if another thread intervened
  - If yes, then retry read/update

Transactional Memory (2)

```
Do {
    Transaction: {
        balance++;
    }
} while (!conflict);
```

- Need mechanism to check if conflict occurred
  - In software, or in hardware
  - Compiler can automatically insert checks for every shared memory access
  - Hardware can use dirty flags whenever updates occur
- Need mechanism to roll back partially executed transaction
  - Modern processors already have support for speculative execution and rollback (for performance reasons)

Transactional Memory Pros and Cons

- Pros:
  - Scalable concurrency at fine granularity, i.e., lots of concurrent small operations
  - Composability - new code can't create problems
  - No blocking
  - No deadlocks
  - No priority inversion
- Cons:
  - Not universal – need operations to be idempotent
  - E.g., what to do with I/O? RPCs?
  - Need hardware support for efficiency (maintain conflict state)
  - Can have substantial overhead if conflict rate is high

Linux Kernel Seq Locks

- Locks that favor writers over readers
  - Lots of readers, few writers, light-weight
  - Programmer invoked transactional memory
  - Limited – doesn't support lock free concurrent writes
- Basic idea:
  - Lock is associated with sequence number
  - Writers increment seq number
  - Readers check seq number at lock and unlock
  - If different, try again
  - Writers synchronize between themselves, never block for readers

Seq Lock Operations

- Operations for manipulating seq locks:

  DEFINE_SEQLOCK – initialize seq lock

  write_seqlock – get the seqlock as writer, incr seq (can block)

  write_sequnlock – release seqlock, incr seq

  read_seqbegin, read_seqretry – define read atomic region, seqretry returns true if op was atomic

Writer

```
write_seqlock(&mr_seq_lock);
  /* update data here */
write_sequnlock(&mr_seq_lock);
```

Reader

```
do {
        seq = read_seqbegin(&mr_seq_lock);
        /* read data here */
} while (read_seqretry(&mr_seq_lock, seq)) ;
```

Outline

- Atomic Variables
- Semaphores
- Monitors and condition variables
- Transactional Memory
- Linux Synchronization Primitives

Linux Kernel Synch Primitives

- Memory barriers
  - avoids compiler, cpu instruction re-ordering
- Atomic operations
  - memory bus lock, read-modify-write ops
- RCU
  - Atomic pointer update, list APIs
- Interrupt/softirq disabling/enabling
  - Local, global
- Spin locks
  - general, read/write, big reader
- Semaphores/Mutex
  - general, read/write, mutex
- Seq Locks
  - provides reader side transactional memory

Choosing Synch Primitives

- Avoid synch if possible! (clever instruction ordering)
  - Example: RCUs
- Use atomics or rw spinlocks if possible
- Use semaphores or mutexes if you need to sleep
  - Can't sleep in interrupt context
  - Don't sleep holding a spinlock!
- Complicated matrix of choices for protecting data structures accessed by deferred functions

Barrier Operations

- *barrier* – prevent only compiler reordering
- *mb* – prevents load and store reordering
- *rmb* – prevents load reordering
- *wmb* – prevents store reordering

- *smp_mb* – prevent load and store reordering only in SMP kernel
- *smp_rmb* – prevent load reordering only in SMP kernels
- *smp_wmb* – prevent store reordering only in SMP kernels
- *set_mb* – performs assignment and prevents load and store reordering

Interrupt Operations

- Intel: "interrupts enabled bit"
  - cli to clear (disable), sti to set (enable)
- Enabling is often wrong; need to restore
- Services used to serialize with interrupts are:

  local_irq_disable - disables interrupts on the current CPU

  local_irq_enable - enable interrupts on the current CPU

  local_save_flags - return the interrupt state of the processor

  local_restore_flags - restore the interrupt state of the processor
- Dealing with the full interrupt state of the system is officially discouraged. Locks should be used.

Spin Locks

- A spin lock is a data structure (*spinlock_t*) that is used to synchronize access to critical sections.
- Only one thread can be holding a spin lock at any moment. All other threads trying to get the lock will "spin" (loop while checking the lock status).
- Spin locks should not be held for long periods because waiting tasks on other CPUs are spinning, and thus wasting CPU execution time.

Spin Lock Operations

- Functions used to work with spin locks (struct spinlock_t):
  - *DEFINE_SPINLOCK* – initialize a spin lock before using it for the first time
  - *spin_lock* – acquire a spin lock, spin waiting if it is not available
  - *spin_unlock* – release a spin lock
  - *spin_unlock_wait* – spin waiting for spin lock to become available, but don't acquire it
  - *spin_trylock* – acquire a spin lock if it is currently free, otherwise return error
  - *spin_is_locked* – return spin lock state

Spin Locks & Interrupts

- The spin lock services also provide interfaces that serialize with interrupts (on the current processor):

  spin_lock_irq - acquire spin lock and disable interrupts

  spin_unlock_irq - release spin lock and reenable

  spin_lock_irqsave - acquire spin lock, save interrupt state, and disable

  spin_unlock_irqrestore - release spin lock and restore interrupt state

RW Spin Lock Operations

- Several functions are used to work with read/write spin locks (struct rwlock_t):
  - *DEFINE_RWLOCK, rwlock_init* – initialize a read/write lock before using it for the first time
  - *read_lock* – get a read/write lock for read
  - *write_lock* – get a read/write lock for write
  - *read_unlock* – release a read/write lock that was held for read
  - *write_unlock* – release a read/write lock that was held for write
  - *read_trylock, write_trylock* – acquire a read/write lock if it is currently free, otherwise return error

RW Spin Locks & Interrupts

- The read/write lock services also provide interfaces that serialize with interrupts (on the current processor):

  read_lock_irq - acquire lock for read and disable interrupts

  read_unlock_irq - release read lock and reenable

  read_lock_irqsave - acquire lock for read, save interrupt state, and disable

  read_unlock_irqrestore - release read lock and restore interrupt state

- Corresponding functions for write exist as well (e.g., *write_lock_irqsave*).

Semaphores

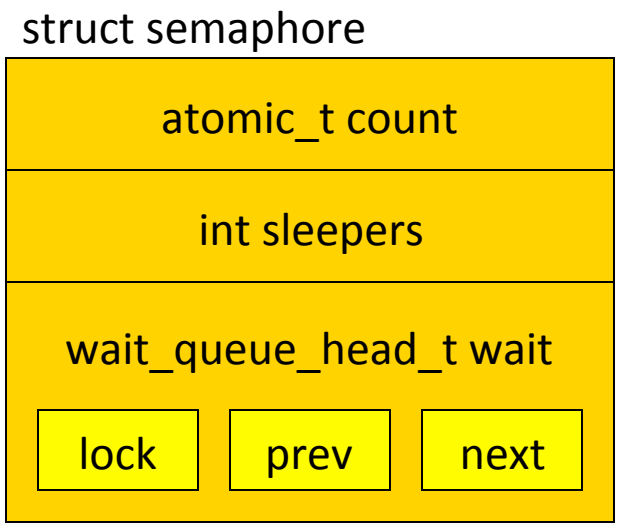
- A *semaphore* is a data structure that is used to synchronize access to critical sections or other resources.
- A *semaphore* allows a fixed number of tasks (generally one for critical sections) to "hold" the semaphore at one time. Any more tasks requesting to hold the *semaphore* are blocked (put to sleep).
- A *semaphore* can be used for serialization only in code that is allowed to block.

Semaphore Operations

- Operations for manipulating semaphores:
  - *up* – release the semaphore
  - *down* – get the semaphore (can block)
  - *down_interruptible* – get the semaphore, but return whether we blocked
  - *down_trylock* – try to get the semaphore without blocking, otherwise return an error

Semaphore Structure

- Struct semaphore
  - count (atomic_t):
    - > 0: free;
    - = 0: in use, no waiters;
    - < 0: in use, waiters
  - wait: wait queue
  - sleepers:
    - 0 (none),
    - 1 (some), occasionally 2
  - wait: wait queue
- Implementation requires lower-level synch
  - atomic updates, spinlock, interrupt disabling



Semaphores

- optimized assembly code for normal case (down())
  - C code for slower "contended" case (__down())
- up() is easy
  - atomically increment; wake_up() if necessary
- uncontended down() is easy
  - atomically decrement; continue
- contended down() is really complex!
  - basically increment sleepers and sleep
  - loop because of potentially concurrent ups/downs
- still in down() path when lock is acquired

RW Semaphores

- A *rw_semaphore* is a semaphore that allows either one writer or any number of readers (but not both at the same time) to hold it.
- Any writer requesting to hold the *rw_semaphore* is blocked when there are readers holding it.
- A *rw_semaphore* can be used for serialization only in code that is allowed to block. Both types of semaphores are the only synchronization objects that should be held when blocking.
- Writers will not starve: once a writer arrives, readers queue behind it
- Increases concurrency; introduced in 2.4

RW Semaphore Operations

- Operations for manipulating semaphores:

  up_read – release a *rw_semaphore* held for read.

  up_write – release a *rw_semaphore* held for write.

  down_read – get a *rw_semaphore* for read (can block, if a writer is holding it)

  down_write – get a *rw_semaphore* for write (can block, if one or more readers are holding it)

More RW Semaphore Ops

- Operations for manipulating semaphores:

  down_read_trylock – try to get a *rw_semaphore* for read without blocking, otherwise return an error

  down_write_trylock – try to get a *rw_semaphore* for write without blocking, otherwise return an error

  downgrade_write – atomically release a *rw_semaphore* for write and acquire it for read (can't block)

Mutexes

- A *mutex* is a data structure that is *also* used to synchronize access to critical sections or other resources, introduced in 2.6.16.
- Why? (`Documentation/mutex-design.txt`)
  - simpler (lighter weight)
  - tighter code
  - slightly faster, better scalability
  - no fastpath tradeoffs
  - debug support – strict checking of adhering to semantics
- Prefer mutexes over semaphores

Mutex Operations

- Operations for manipulating mutexes:
  - *mutex_unlock* – release the mutex
  - *mutex_lock* – get the mutex (can block)
  - *mutex_lock_interruptible* – get the mutex, but allow interrupts
  - *mutex_trylock* – try to get the mutex without blocking, otherwise return an error
  - *mutex_is_locked* – determine if mutex is locked

Completions

- Slightly higher-level, FIFO semaphores
  - Solves a subtle synch problem on SMP
- Up/down may execute concurrently
  - This is a good thing (when possible)
- Operations: complete(), wait_for_complete()
  - Spinlock and wait_queue
  - Spinlock serializes ops
  - Wait_queue enforces FIFO