Locking

CIS-4020
Vermont Technical College
Peter C. Chapin

Why Are Threads Difficult?

- In general they are asynchronous. *The actions of one thread interleave arbitrarily with the actions of other threads.*
  - Difficult to test thoroughly.
  - Difficult to reproduce thread related bugs.
  - Difficult to analyze.
- When threads share state, that state can easily become corrupted.
- **Careful design is a must.**

Unexpected Problems

```
long counter;
```

```
void f()
{
  counter++;
}
```

```
void g()
{
  if (counter == 0) {
    ...
  }
}
```

- Assume 16 bit operations
- If `counter == 0x0000FFFF` a *race condition* exists
- Function `g()` might behave unexpectedly.

Is Incrementing Atomic?

- The assembly language for `counter++`
  - Multiple steps are required.

```
        inc low_word
        jnc skip
        inc high_word
    skip:
```

  - If value initially 0x0000FFFF, it temporary becomes 0x00000000.
    - After low word is incremented and before high word is incremented.
  - If thread suspended at that moment, the other thread sees a bad value.

Atomic Operations

- **Defn**: *An atomic operation is one that can't be interrupted.*
  - It either executes completely or not at all.
- On a uni-processor, individual instructions are atomic.
  - Interrupt processing occurs *between* instructions.
  - However: Any operation requiring more than one instruction could be interrupted mid-operation.

SMP

- It's harder on a multiprocessor.
  - Individual machine cycles (memory operations) are atomic (usually).
  - Instructions on two CPUs might interleave.
  - `inc value`

    1)Reads value into the CPU

    2)Increments it

    3)Writes new value back to memory.

    - Two CPUs might read the same value at once.
      - Value ends up getting incremented just once instead of twice.

Memory Locking

- CPUs designed for use in SMP systems...
  - … have some sort of memory locking feature.
  - Allows an instruction to *read-modify-write* memory atomically.
    - Could be specialized instructions.
    - Could be special modes on existing instructions.
  - x86 has a LOCK prefix that can be applied to certain instructions.
    - Causes a special line on the processor to go active.
    - Used by memory hardware to implement locking against other CPUs.

It's Even Worse

- Modern CPUs execute instructions out of order.
  - Processor evaluates data dependencies properly (so no data is used before it is computed).
  - But... not aware of what other threads are doing!
    - Might violate dependency relationship for other threads.
  - Getting this right is complex.
    - Requires compiler assistance.
    - A subject for the parallel programming course.

Summary

- The problem with threads...
  - *As a data structure is manipulated it goes through intermediate, inconsistent states.*
  - *If a thread is suspended while its data is inconsistent... another thread will become confused when it tries to use that data.*
- **Example**
  - Invariant: The count member of a List always contains the number of list elements.
  - BUT... when adding a new member, the count is temporary wrong (invariant invalid).

One Solution

- *Don't let threads execute asynchronously.*
  - *Cooperative multi-threading* occurs when a thread runs as long as it wants and voluntarily *yields* to other threads when ready.
    - Doesn't completely eliminate problems. Care still needed.
- Prevents some of the advantages of threads.
  - If a thread never yields no other threads run.
  - Programs can't easily take advantage of multiple CPUs. (why not?)

Another Solution

- *Lock entire data structures; even large ones.*

  1)Thread tries to acquire lock before using the structure.

  2)If lock already owned by another thread, the second thread is put to sleep.

  3)After data structure is back in a consistent state, the first thread unlocks it.

  4)A thread sleeping on the lock is awakened and allowed to continue.

BKL

- In the old days there was a single “big kernel lock.”
  - Only one thread at a time in the kernel.
  - Must ensure data is consistent *before returning or sleeping*.
- Still some concerns... for example:
  - `kmalloc` might sleep.
  - Thread must put data structures into consistent state before calling `kmalloc`.

Modern BKL

- Modern Linux avoids using the BKL
  - Data structures locked in a fine grained manner.
    - Allows kernel preemption.
    - Improves latency: *A process that wants a kernel service may be able to get it right away, even if another process is in the kernel at the same time.*
    - Allows multiple processors in the kernel.
  - Adds complexity
    - Must be more aware of locking issues.
    - The possibility of *deadlock* arises.

Locking Primitives

- Inside Linux there are several lock primitives. In order of increasing level of abstraction:
  - Atomic operations.
  - Spin locks.
  - Mutexes.
  - Semaphores.
  - Reader/writer locks.

Atomic Operations

- Very primitive.
  - ```
    typedef struct {
            int counter;
      } atomic_t;
    ```
  - An integer wrapped in a structure.
    - The wrapper prevents accidental misuse.
  - `atomic_read(atomic_t *v)`
    - Reads the atomic value `*v`; won't be corrupted by a simultaneous `atomic_set`.
  - `atomic_set(atomic_t *v, int i)`
    - Sets atomic value `*v` to `i`; won't be corrupted by another `atomic_set`.
    ```

Spin Locks

- Used to protect critical sections.
  - Used as follows:

```
DEFINE_SPINLOCK(lock);
...
spin_lock(&lock);
// Critical section. Don't sleep!
spin_unlock(&lock);
```

  - If lock is busy, processor loops (“spins”) until lock is free.
    - Potentially wasteful. Critical section must be short
    - Low overhead.

Spin Locks

- *Only make sense on a multiprocessor.*
  - On a uni-processor, the spinning runs until thread preempted.
    - **Could take many milliseconds!**
    - If preemption disabled, **spinning runs forever!**
  - In Linux if `CONFIG_SMP` is off, spin locks resolve to *nop* operations...
    - … except that they disable kernel preemption if that was on.

Linux Spin Locks

- On an SMP machine...
  - … spin locks spin (but not long... critical section must be short)
- On a UP machine without kernel preemption...
  - … spin locks do nothing. No other thread can be in the critical section anyway.
- On a UP machine with kernel preemption...
  - … spin locks disable preemption (but don't spin). No other thread can preempt the critical section so it executes atomically.

Mutexes

- Semaphores can be used for mutual exclusion.
  - BUT… semaphores have more overhead (and are more flexible).
- Linux provides simple mutex objects.
  -
```
struct mutex {
    atomic_t    count;
    spinlock_t wait_lock;
    struct list_head wait_list;
};
```
- See mutex.h in the kernel source!

Mutual Exclusion

- Used to protect critical sections.
  - Used as follows:

```
struct mutex lock;
mutex_init( &lock );
...
mutex_lock( &lock );
// Critical section. Sleep okay!
mutex_unlocl( &lock );
```

  - Okay to sleep... other threads can run while one thread holds a mutex.
  - More overhead than spinlocks. (*why?*)

Semaphores

- Glorified counters.
  - Defined as:

```
struct semaphore {
  raw_spinlock_t lock;
  unsigned int count;
  struct list_head wait_list;
};
```

  - The core is the count member.
    - Decrementing a zero semaphore puts you to sleep.
    - Incrementing a semaphore might awaken a sleeping thread.

Three Operations

- Semaphore operations
  - `sem_init( struct semaphore *, int )`
    - Initialize the semaphore, including giving it an initial value.
  - `down( struct semaphore * )`
    - Decrement the semaphore, perhaps going to sleep (can't decrement zero).
  - `up( struct semaphore * )`
    - Increment the semaphore, perhaps waking up someone else (if the semaphore was zero).
- See semaphore.h in the kernel source!

Reader/Writer Locks

- Since usually only updates cause corruption...
  - Okay to allow multiple simultaneous reads.
  - Writers must be exclusive.
- Basic facilities
  - `rwlock_t`
    - A spin lock for reader/write applications.
  - `read_lock, read_unlock`
    - Multiple readers okay.
  - `write_lock, write_unlock`
    - Must gain exclusive access.

Reader/Writer Starvation

- What if...
  - There were an unending stream of readers and then a writer comes along?
    - *Does the writer get priority?*
    - *Does the writer starve?*
  - There were an unending stream of writers and then a reader comes along?
    - *Does the reader get priority?*
    - *Does the reader starve?*
    - BUT… writers should be rare (if not, just use "regular" mutual exclusion)

Producer/Consumer

- Classic problem in concurrent programming.
  - One (or more) “producer” threads write data into a buffer.
  - One (or more) “consumer” threads take data out of the buffer.
  - Problems:
    - Make sure no data corruption occurs in the buffer. Provide mutual exclusion.
    - Make sure buffer does not overflow. Count number of free slots and block producer if zero.
    - Make sure buffer does not underflow. Count number of used slots and block consumer if zero.

Producer Pseudo-Code

- The producer looks like this:
  -
```
while( 1 ) {
    produce_item( );
    down( &free_slot_semaphore );
    lock( &buffer_mutex );
    install_item_in_buffer( );
    unlock( &buffer_mutex );
    up( &used_slot_semaphore );
}
```
  - "Reserve" a free slot first; then lock.
    - If you lock first you could sleep holding the lock if no free slots are available!

Consumer Pseudo-Code

- The consumer looks like this:
  - ```
    while( 1 ) {
        down( &used_slot_semaphore );
        lock( &buffer_mutex );
        remove_item_from_buffer( );
        unlock( &buffer_mutex );
        up( &free_slot_semaphore );
        consume_item( );
    }
    ```
  - Doing `up` on the semaphore releases the other thread if it is waiting.
    ```

Priority Inversion

- ***Defn****: A high priority thread is blocked waiting for a thread of lower priority.*
  - How can that happen?
    - Note: a thread can not preempt a mutex, even if it has higher priority than the thread holding it.
    - Doing so risks data corruption!
  - Scenario
    - LP thread locks mutex.
    - HP thread tries to acquire mutex; is blocked.
    - MP thread preempts LP thread; runs indefinitely!

Priority Inheritance

- Solution is *priority inheritance.*
  - Scenario
    - LP thread locks mutex.
    - HP thread tries to acquire mutex; blocked.
    - LP thread “inherits” the priority of the HP thread.
    - MP thread becomes runnable; must wait.
    - LP/HP thread completes critical section. Unlocks mutex.
    - LP/HP thread returns to LP.
    - HP thread acquires mutex.