Thread and Synchronization

Synchronization Mechanisms (Module 20)

Yann-Hang Lee
Arizona State University
yhlee@asu.edu
(480) 727-7507

Summer 2014

Thread Synchronization -- Semaphore

- **creating a semaphore:**

  *int sem_init(sem_t *sem, int pshared, unsigned int value);*

  - initializes a semaphore object pointed to by sem
  - pshared is a sharing option; a value of 0 means the semaphore is local to the calling process
  - gives an initial value to the semaphore

- **terminating a semaphore:**

  *int sem_destroy(sem_t *sem);*

- **semaphore control:**

  *int sem_post(sem_t *sem);*

  *int sem_wait(sem_t *sem);*

  - sem_post atomically increases the value of a semaphore by 1,
  - sem_wait atomically decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first

Example: Semaphore

```
#include <pthread.h>
#include <semaphore.h>
sem_t semaphore;        // also a global variable just like mutexes
int main()
{
    int tmp;
    tmp = sem_init( &semaphore, 0, 0 );                    // initialize the semaphore
    pthread_create( &thread[i], NULL, thread_function, NULL ); // create threads
    while ( still_has_something_to_do() )
    {
        sem_post( &semaphore );
        ...         }
    pthread_join( thread[i], NULL );
    sem_destroy( &semaphore );
    return 0;
}
```

```
void *thread_function( void *arg ) {
    sem_wait( &semaphore );
    perform_task();
    pthread_exit( NULL );
}
```

Condition Variables

- **A variable of type** *pthread_cond_t*
- **Use condition variables to atomically block threads until a particular condition is true.**
- **Always use condition variables together with a mutex lock.**

```
pthread_mutex_lock();
    while( condition_is_false )
            pthread_cond_wait();
pthread_mutex_unlock();
```

- **Use** *pthread_cond_wait()* **to atomically release the mutex and to cause the calling thread to block on the condition variable**
- **The blocked thread can be awakened by** *pthread_cond_signal(), pthread_cond_broadcast(),* **or when interrupted by delivery of a signal.**

Atomic Operations

- **Atomic operations provide instructions that are**
  - executable atomically;
  - without interruption
  - Not possible for two atomic operations by a single CPU to occur concurrently
- **Atomic 80x86 instructions**
  - Instructions that make zero or one aligned memory access
  - Read-modify-write instructions (inc or dec)
  - Read-modify-write instructions whose opcode is prefixed by the lock byte (0xf0)
- **In RISC, load-link/store conditional (ldrex/strex)**
  - store can succeed only if no updates have occurred to that location since the load-link.
- **Linux kernel**
  - two sets of interfaces for atomic operations: one for integers and another for individual bits

Linux Atomic Operations

- **Uses atomic_t data type**
- **Atomic operations on integer counter in Linux**

Function	Description
atomic_read(v)	Return *v
atomic_set(v,i)	set *v to i
atomic_add(i,v)	add i to *v
atomic_sub(i,v)	subtract i from *v
atomic_sub_and_test(i,v)	subtract i from *v and return 1 if result is 0
atomic_inc(v)	add 1 to *v
atomic_dec(v)	subtract 1 from *v
atomic_dec_and_test(v)	subtract 1 from *v and return 1 if result is 0
atomic_inc_and_test(v)	add 1 to *v and return 1 if result is 0
atomic_add_negative(i,v)	add i to *v and return 1 if result is negative

- **A counter to be incremented by multiple threads**
- **Atomic operate at the bit level, such as**

```
unsigned long word = 0;
set_bit(0, &word);          /* bit zero is now set (atomically) */
```

Spinlock

- **Ensuring mutual exclusion using a busy-wait lock.**
  - if the lock is available, it is taken, the mutually-exclusive action is performed, and then the lock is released.
  - If the lock is not available, the thread busy-waits on the lock until it is available.
  - it keeps spinning, thus wasting the processor time
  - If the waiting duration is short, faster than putting the thread to sleep and then waking it up later when the lock is available.
  - really only useful in SMP systems
- **Spinlock with local CPU interrupt disable**

```
spin_lock_irqsave( &my_spinlock, flags );
   // critical section
spin_unlock_irqrestore( &my_spinlock, flags );
```

- **Reader/writer spinlock – allows multiple readers with no writer**

SpinLock

- ***static inline void spin_lock(spinlock_t *lock)***
- **is defined in**
  - *spinlock_api_up.h*
  - *spinlock_api_smp.h*
- **For up,**

```
#define _raw_spin_lock(lock)                __LOCK(lock)
#define __LOCK(lock)   preempt_disable(); ___LOCK(lock);
#define ___LOCK(lock)  __acquire(lock); (void)(lock);
#define __acquire(x) (void)0                // noop
#define preempt_disable()          barrier()

spin_lock_irqsave(lock, flags) → f = arch_local_save_flags();
                                    arch_local_irq_disable();
```