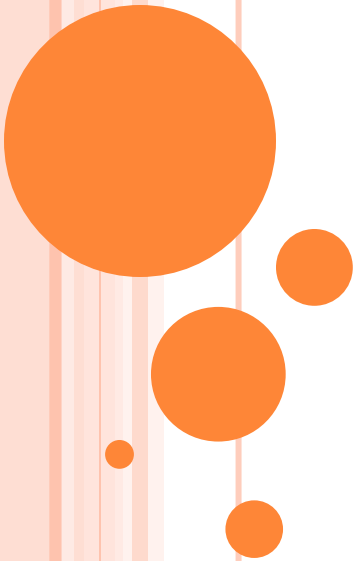

Linux Device Drivers (Kernel Synchronisation) C-DAC, Mumbai

Agenda

- What is Synchronization????
- Sources of Concurrency in the Kernel
- Mechanisms to manage concurrency
- References

What is Synchronization

- In the kernel there can be many tasks that execute pseudo concurrently. This may lead to data inconsistencies in accessing a common resource
- A well defined coordination between tasks in accessing shared data is a must and this coordination leads to synchronization between tasks.

Sources of Concurrency

- In a linux system, there is a possibility that numerous process are executing in the user space, making system calls to the kernel
- SMP systems can access your code concurrently
- Kernel code is pre-emptible.
- Interrupts are asynchronous events that can cause concurrent execution
- Delayed code execution mechanisms provided by the kernel
- Hot pluggable devices can suddenly stop the functioning of the code

Mechanisms to manage concurrency

- Any time a hardware or software resource is shared beyond a single thread of execution, there is a possibility that one thread gets an inconsistent view of that resource
- This calls for some resource access management and is brought about by mechanisms called locking or mutual exclusion making sure that only one resource can manipulate a shared resource at one time.

Semaphores

- At its core, a semaphore is a single integer value combined with a pair of functions that are typically called *up and down*
- To use semaphores, the code must include *asm/semaphore.h*.
- *The semaphore implementation* in the kernel is just a structure semaphore.

```
struct semaphore {
        atomic_t count;
        int sleepers;
        wait_queue_head_t wait;
            };
```

Semaphores

- There are two ways of creating a semaphore. The dynamic way uses the function

  void sema_init(struct semaphore *sem, int val)

- Statically, semaphores may be created by the macro
  - Static DECLARE_SEMAPHORE_GENERIC(name,count)

- The count or val in both cases specifies the initialization value of the semaphore. Setting it to 1 created the semaphore as a binary semaphore or a mutex (mutual exclusion semaphore)

Semaphores

- Semaphores may also be created in the mutex mode by the following functions
  - DECLARE_MUTEX(name);
  - DECLARE_MUTEX_LOCKED(name);
- They may be initialized at runtime by the following
  - init_MUTEX(struct semaphore *sem);
  - void init_MUTEX_LOCKED(struct semaphore *sem)

Semaphores

- Semaphores may be accessed by calling one of the following functions
  - void down(struct semaphore *sem);
  - int down_interruptible(struct semaphore *sem);
  - int down_trylock(struct semaphore *sem);
- Once access to the critical section is completed, the semaphore may be released by the function
  - void up(struct semaphore *sem)

Reader/Writer Semaphores

- Code using rwsems must include *linux/rwsem.h*. *The* relevant data type for rwsem is struct rw_semaphore. An rwsem must be explicitly initialized at run time using
  - void init_rwsem(struct rw_semaphore *sem)
- For read only access,
  - void down_read(struct rw_semaphore *sem);
  - int down_read_trylock(struct rw_semaphore *sem);
  - void up_read(struct rw_semaphore *sem)
- For write only access, the functions are similar
- For requirements wherein a long read is required after a quick write,
  - void downgrade_write(struct semaphore *sem)

Spinlocks

- A spinlock is a mutual exclusion device that can have only two values *locked and unlocked. It is* implemented as a single bit in an integer value. Code wishing to take out a particular lock tests the relevant bit.
- Unlike semaphores, spinlocks may be used in code that cannot sleep.
- If the lock is taken by somebody else, the code goes into a tight loop where it repeatedly checks the lock until it becomes available

Spinlocks

- Spinlocks are intended for use on multiprocessor systems although a uniprocessor workstation running a preemptive kernel behaves like SMP
- If a non preemptive uniprocessor ever went into a spinlock, it would spin forever; on other thread would ever be able to obtain the CPU to release the lock

SPINLOCKS

- The required include file for spinlock primitives is *linux/spinlock.h. A spinlock has the type* spinlock_t and has to be initialized before it is used
- The static initialization for a spinlock is done by
  - spinlock_t my_lock = SPIN_LOCK_UNLOCKED
- or at runtime as
  - void spin_lock_init(spinlock_t *lock);
- A spinlock is obtiained and released by
  - void spin_lock(spinlock_t *lock);
  - void spin_unlock(spinlock_t *lock);

Spinlocks and Interrupts

- Spinlocks can be used in interrupt handlers, whereas semaphores may not be used since they sleep.
- If a lock is shared with an interrupt handler, local interrupts must be disabled before acquiring the lock.
- The kernel provides a separate interface for this which disables and enables local interrupts on acquiring and releasing the spinlock respectively

  - void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
  - void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);

Reader/Writer Spinlocks

- Using read/write spinlocks is similar to rwsems
- It is initialized by
  - rwlock_t mylock =RW_LOCK_UNLOCKED //static way
- rwlock_t mylock;
  - rw_lock_init (&my_rwlock); //Dynamic way
- Reader and writer locks may be gained and released by
  - void read_lock(rmlock_t *lock);
  - void read_unlock(rwlock_t *lock);
  - void write_lock(rwlock_t *lock);
  - void write_unlock(rwlock_t *lock);

Semaphores vs Spinlocks

Requirement	Recommended Lock
Low overhead locking	Spinlock
Short lock hold time	Spinlock
Long lock hold time	Semaphore
Need to lock from interrupt context	Spinlock
Need to sleep while holding lock	Semaphore

Completions

- A common phenomenon in kernel programming is the initiation of some activity outside the current execution flow and then wait for that activity to complete.
- Consider the following code snippet
  - struct semaphore sem;
    init_MUTEX_LOCKED(&sem);
    start_external_task(&sem);
    down(&sem);

Completions

- Completions are a simple light weight mechanism with one task: allowing one thread to tell another that the job is done
- A completion can be created with
  - DECLARE_COMPLETION(my_completion);
- Waiting for the completion is by simply calling
  - void wait_for_completion(struct completion *c);
- The actual completion event is signaled by
  - void complete(struct completion *c);
  - void complete_all(struct completion *c);

Atomic Variables

- Atomic variables are special data types that are provided by the kernel, to perform simple operations in an atomic manner.
- The kernel provides an atomic integer type called atomic_t and a set of functions that have to be used to perform operations on the atomic variables.
- The operations are very fast, because they compile to a simple machine instruction whenever possible

Atomic Integer Operations

- Some important integer operations are
  - void atomic_set(atomic_t *v, int i);
  - int atomic_read(atomic_t *v);
  - void atomic_add(int i, atomic_t *v);
  - void atomic_sub(int i, atomic_t *v);
  - void atomic_inc(atomic_t *v);
  - void atomic_dec(atomic_t *v);
  - int atomic_inc_and_test(atomic_t *v);

Atomic Bit Operations

- The atomic data type is good in working around with integers. It does not suffice bitwise operations. The kernel provides necessary functions that act on single bits. These are declared in *asm/bitops.h*

- The available bit operations are:
  - void set_bit(nr, void *addr);
  - void clear_bit(nr, void *addr);
  - void change_bit(nr, void *addr);

SEQLOCKS

- An added feature in the 2.6 kernel that is intended to provide fast, lockless access to a shared resource.
- Seq locks work in situations where write access is rare but must be fast
- They work by allowing readers free access to the resource but requiring those readers to check for collisions with writers and, when collisions occur, retry their access
- Cannot be used to protect data structures involving pointers because the reader may be following a pointer that is invalid while the writer may be changing the data structure

SEQLOCKS

- Seqlocks are defined in *linux/seqlock.h. It may* be initialized by
  - seqlock_t lock1= SEQLOCK_UNLOCKED;
- The write path is obtained by
  - void write_seqlock(seqlock_t *lock);

  /*write lock is obtained….make changes*/
  - void write_sequnlock (seqlock_t *lock);
- Readers may function in this pattern

  do {

  seq = read_seqbegin(&lock);

  read the data here

  }while (read_seqretry(&lock, seq);

Read Copy Update