montavista VISION 2008

EMBEDDED LINUX DEVELOPERS CONFERENCE

WIPRO Applying Thought

# Linux Synchronization Mechanisms in Driver Development

**Dr. B. Thangaraju & S. Parimala**

montavista

# Agenda

- **Sources of race conditions**
  - ~ 11,000+ critical regions in 2.6 kernel
- **Locking mechanisms in driver development**
  - Semaphore
  - Spinlock
  - Atomic bit and integer operations
  - Read-Copy-Update
  - Seqlocks
  - Completions
- **Deadlock**
  - Sources and Prevention

# Vital sources of race conditions

- **Sharing of resources** – Two tasks trying to access the same resource
- **Interrupt handling** – Concurrent access to a critical section by two interrupt handlers
- **Kernel preemption** – Highest priority task preempts a lower priority task which leaves the lower priority task in the decisive state

# Semaphore

- Principal mutual exclusion
- **Suitable if the calling task can sleep**
- Reader-Writer version is available
- Used in exception handler, memory descriptor, page fault handler and system call routines
- Heavy weight solution for small critical sections

## Semaphore Functions

- void sema_init (struct semaphore *sem, int val);
- void down (struct semaphore *sem);
- void down_interruptible (struct semaphore *sem);
- int down_trylock (struct sempahore *sem);
- void up (struct semaphore *sem);
- void init_rwsem (struct rw_semaphore *rw_sem);
- void down_read (struct rw_semaphore *rw_sem);
- void up_read (struct rw_semaphore *rw_sem);
- void down_write (struct rw_semaphore *rw_sem);
- void up_write (struct rw_semaphore *rw_sem);
- void downgrade_write (struct rw_semaphore *rw_sem);

# Spinlock

- Ideal for small critical section for e.g. ISR. Suitable for SMP. Reader/Writer version is also available.
- **Appropriate if the calling task doesn't sleep.**
- Task will be in tight loop till the lock is available – hog the processor.
- Used in file system, vnodes, I/O ports, buffer cache and interrupt handlers.
- Expensive when protecting a global integer variable or manipulating shared hardware bits.

# Spinlock

- Architecture dependent
- Spinlock Functions
  - void spin_lock_init (spinlock_t *lock);
  - void spin_lock (spinlock_t *lock);
  - void spin_unlock (spinlock_t *lock);
  - void spin_lock_irqsave (spinlock_t *lock, unsigned long flags);
  - void spin_lock_irqrestore (spinlock_t *lock, unsigned long flags);
  - void spin_lock_irq (spinlock_t *lock);
  - void spin_unlock_irq (spinlock_t *lock);
  - void spin_lock_bh (spinlock_t *lock);
  - void spin_unlock_bh (spinlock_t *lock);

# Atomic Bit and Integer Operations

- Atomic operations - no locking overhead
- Architecture dependent
- Atomic bit APIs :
  - **void set_bit (nr, void *addr);**
  - **void clear_bit (nr, void *addr);**
  - **void change_bit (nr,void *addr);**
- Atomic integer APIs :
  - **void atomic_set (atomic_t *v, int i);**
  - **int atomic_read (atomic_t *v);**
  - **void atomic_inc (atomic_t *v);**
  - **void atomic_dec (atomic_t *v);**

# Read-Copy-Update

- Best suitable when readers are more and writers are rare
- Orientation towards pointers
- Steps to write access : copy, make changes and change the pointer to a new version
- Freeing the older copy can be done only when there is no other task/thread which has reference to the same resource

# RCU functions

- void rcu_read_lock ( );
- void rcu_read_unlock ( );
- void call_rcu (struct rcu_head *head, void (*func) (void *arg), void *arg);

- func is the one which frees the older copy

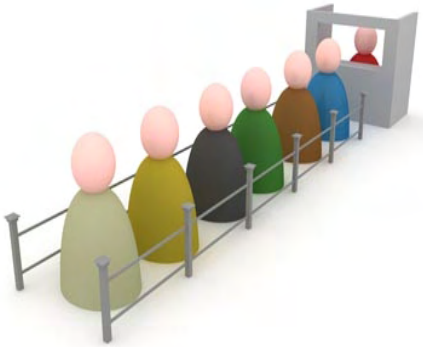
- **Suitable if frequently accessed data structures are available**

- Used in network routing tables, vfs, hash table, scheduler and interrupt handlers

# Seqlocks

- Readers – no locking but need to check for the collisions
- Higher priority for writers
- Internally uses spinlock to control the write access
- Seqlocks functions :
  - **seqlock_init (seqlock_t *lock);**
  - **unsigned int read_seqbegin (seqlock_t *lock);**
  - **int read_seqretry (seqlock_t *lock, unsigned int seq);**
  - **void write_seqlock (seqlock_t *lock);**
  - **void write_sequnlock (seqlock_t *lock);**
- **Suitable to protect few bytes of data**
- Used to update system time

# Completions



- Completions serializes the access to the resource
- Waiting for completion is an uninterruptible wait
- Completion functions :
  - **init_completion (struct completion *c);**
  - **void wait_for_completion (struct completion *c);**
  - **void complete (struct completion *c);**
  - **void complete_all (struct completion *c);**
- **Appropriate if one thread finishes its activity and informs about its completion to another thread**
- Used when the task has to wait for some data or network packet

# Deadlock - a nightmare to embedded develop

## Sources of Deadlock :

- Predominant in preemptive environme
- Improper implementation of locking mechanisms
- e.g. 1 : when there is nesting of functions and all the functions require the same spinlock
- Incorrect order of locking

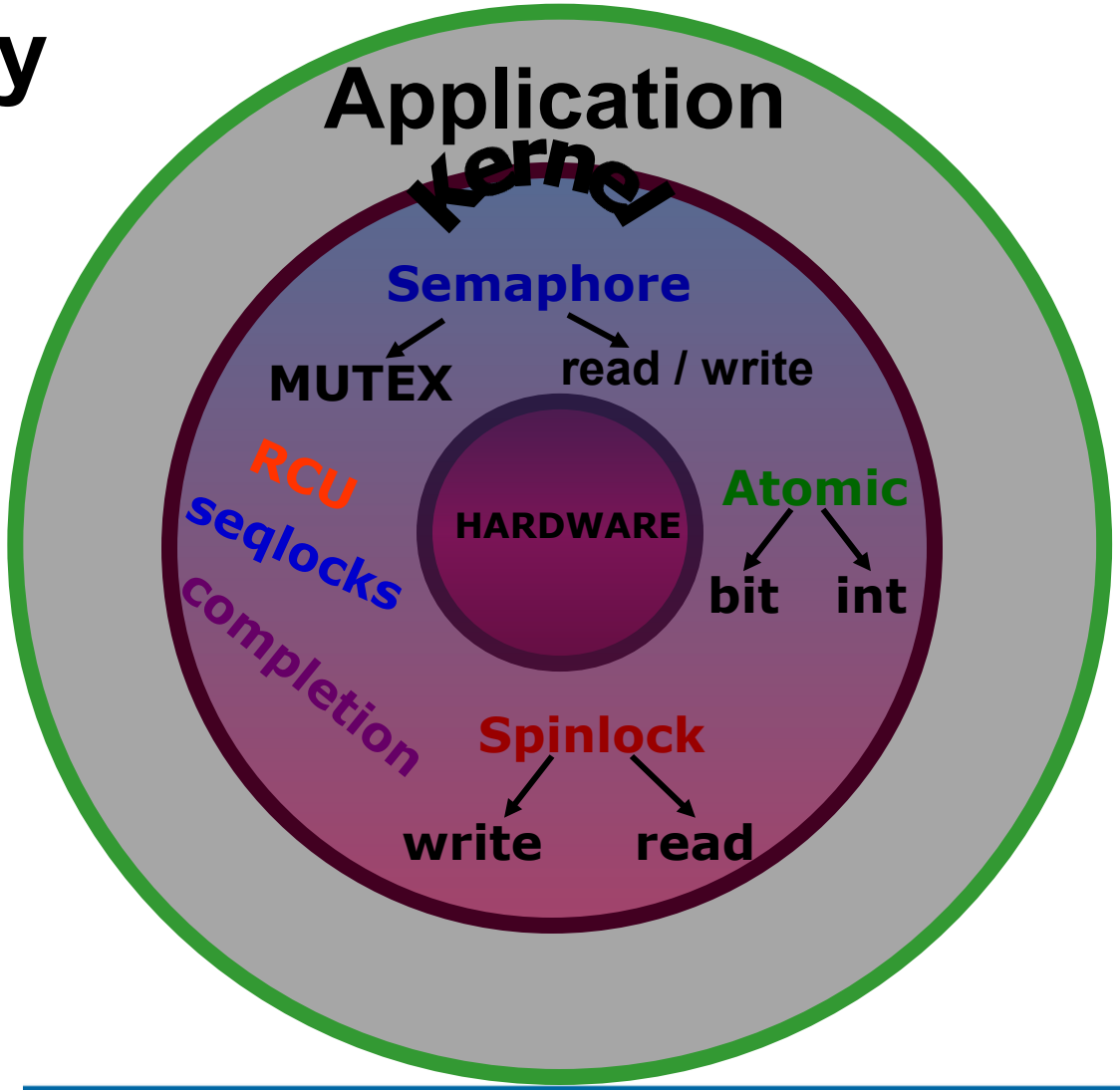
e.g. 2 : task 1 : lock (x) -> lock (y) -> unlock (x) -> unlock(y)

task 2 : lock (y) -> lock (x) -> unlock (y) -> unlock(x)

# Prevention of Deadlock

- Acquire locking in a predefined order
- Hierarchical use of resources
- Not nesting locks, if already a lock is in place
- Not nesting functions, if all are required to access the same resource

# Summary


Application
Kernel
Semaphore
MUTEX
read / write
RCU
seqlocks
completion
HARDWARE
Atomic
bit
int
Spinlock
write
read

# THANK YOU

**Dr. B. Thangaraju - balat.raju@wipro.com**

**S. Parimala - parimala.sathyapramodha@wipro.com**