Linux Kernel Synchronization Methods

www.embeddedshiksha.com

Agenda

- Atomic Operations
- Spin Locks
- Reader Writer Spinlocks
- Semaphores
- Reader Writer Semaphores

Atomic Operation

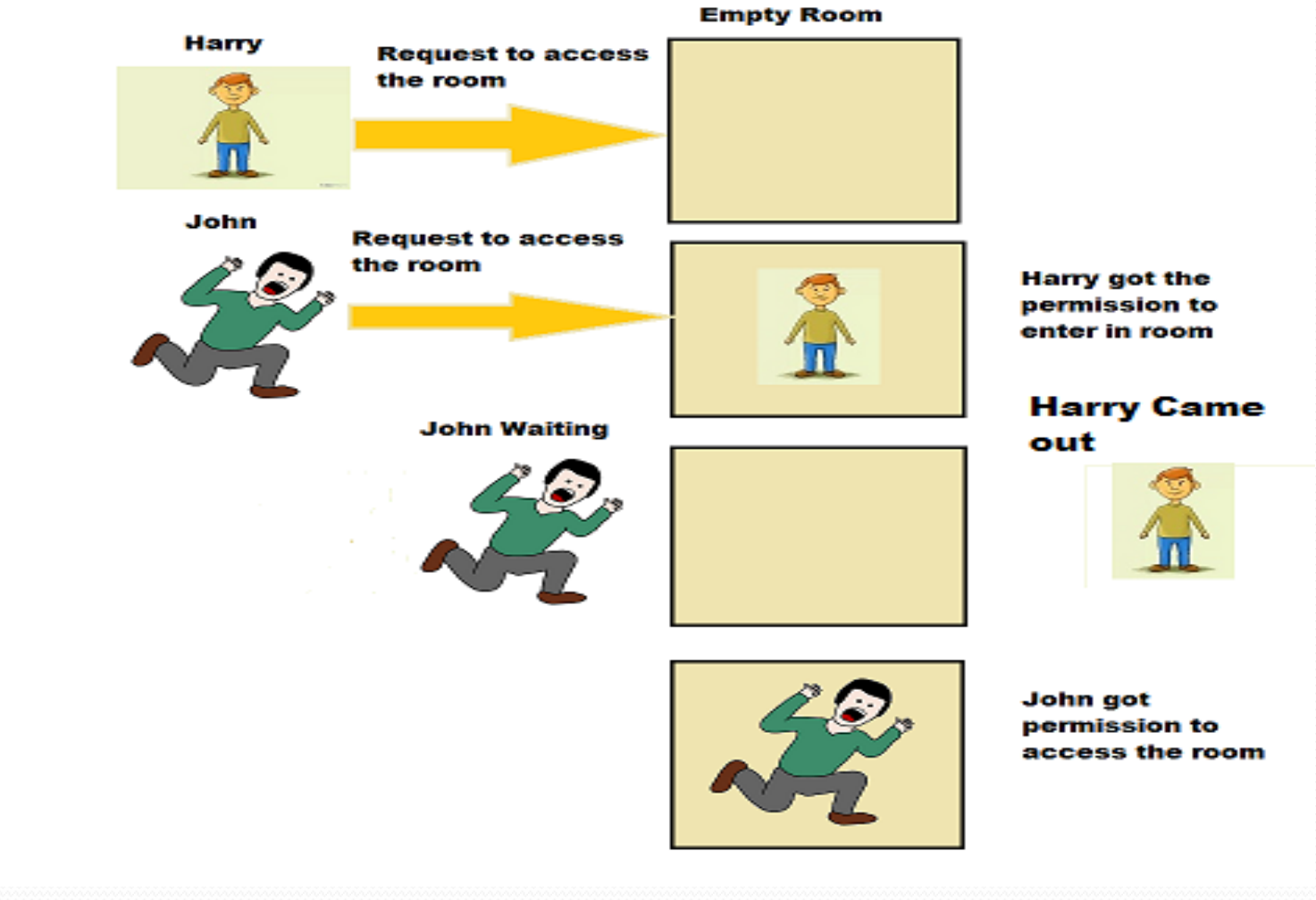
Atomic Integer Operation	Description
`ATOMIC_INIT(int i)`	At declaration, initialize to `i`.
`int atomic_read(atomic_t *v)`	Atomically read the integer value of `v`.
`void atomic_set(atomic_t *v, int i)`	Atomically set `v` equal to `i`.
`void atomic_add(int i, atomic_t *v)`	Atomically add `i` to `v`.
`void atomic_sub(int i, atomic_t *v)`	Atomically subtract `i` from `v`.
`void atomic_inc(atomic_t *v)`	Atomically add one to `v`.
`void atomic_dec(atomic_t *v)`	Atomically subtract one from `v`.

Spinlocks

•Spinlock is a lock which can be held by a single thread at a time . If another thread will try to access spinlock while it is being held by some other thread then this thread has to go for busy looping.

•This spinning prevents more than one thread of execution from entering the critical region at any one time.

Spinlocks


Empty Room
Harry
Request to access the room
John
Request to access the room
Harry got the permission to enter in room
Harry Came out
John Waiting
John got permission to access the room

Spinlock Implementation

•Spinlock code is defined in <asm/spinlock.h>

DEFINE_SPINLOCK(mr_lock);.

spin_lock(&mr_lock);

/* critical region ... */

spin_unlock(&mr_lock);

Spinlock Usage in interrupt

Spinlock can be used in interrupt handling code as we never go for sleep in case of spinlock .

```
DEFINE_SPINLOCK(mr_lock);
unsigned long flags;
spin_lock_irqsave(&mr_lock, flags);
/* critical region ... */
spin_unlock_irqrestore(&mr_lock, flags);
```

Reader Writer Spinlock

In Reader Writer spinlock , if one thread is updating the shared data then no other process will get the lock to access the data. But it allows multiple process to read the at same time.

```
DEFINE_RWLOCK(mr_rwlock);
read_lock(&mr_rwlock);
/* critical section (read only) ... */
read_unlock(&mr_rwlock);

write_lock(&mr_rwlock);
/* critical section (read and write) ... */
write_unlock(&mr_lock);
```

Semaphore

Semaphore provides restriction to accessing same data or resource at same time by multiple process by using of locks. A process has to get the semaphore prior to access the resource. If process get the semaphore then only process can access the resource. If semaphore is not available at that time or if the semaphore is already being held by some other process then semaphore places that task to wait queue and put task to sleep state.

When semaphore becomes available one of task on the wait queue is awakened so that it can then acquire the semaphore. Lets understand this by simple example.

Semaphore APIs

```
sema_init(struct semaphore *, int)
```

```
down(struct semaphore *)
/* Critical Region */
up(struct semaphore *)
```

Counting and Binary Semaphore

Counting Semaphore

- N process can access lock at same time .

Binary Semaphore

- Only one Process can acquire lock at a time .

Mutex

Mutex is just like binary semaphore which can be acquired by max one process at a time .

Only difference between mutex and semaphore is Ownership is provided in Mutex lock .

Reader-Writer Semaphore

Reader-Writer semaphore works in the same manner as reader writer spinlocks works.

```
static DECLARE_RWSEM(name);

Dynamic creation
init_rwsem(struct rw_semaphore *sem)

down_read(&mr_rwsem);
up_read(&mr_rwsem);

down_write(&mr_rwsem);
up_write(&mr_sem);
```

When to use spin lock and semaphore

Spinlock :

- Interrupt handler
- Acquiring Lock waiting is short

Semaphore :

- Holding lock time is long

How to protect shared data between process and interrupt handler

- Disable the local interrupt

```
DEFINE_SPINLOCK(mr_lock);

unsigned long flags;

spin_lock_irqsave(&mr_lock, flags);
/* critical region ... */

spin_unlock_irqrestore(&mr_lock, flags);
```

Thank You