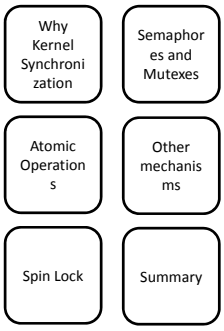
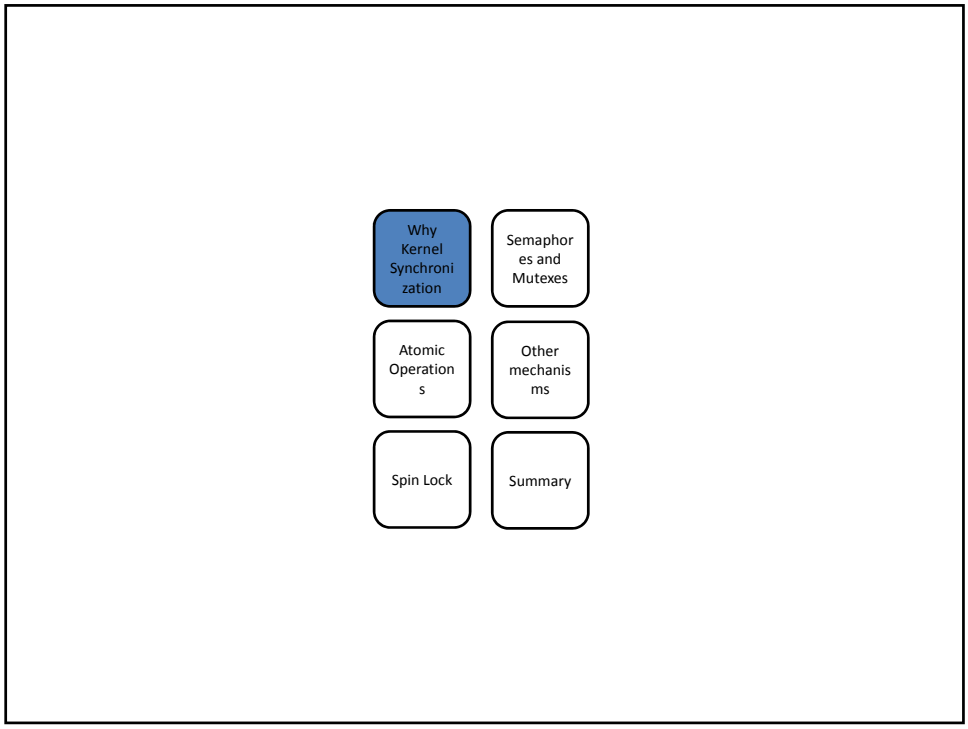
# Operating System Design and Implementation

*Kernel Synchronization*

Shiao-Li Tsao






# Overview

- Shared data
- Critical regions (critical sections)
  - Code paths that access/manipulate shared data

```
993 static int copy_fs(unsigned long clone_flags, struct task_struct *tsk)
994 {
995         struct fs_struct *fs = current->fs;
996         if (clone_flags & CLONE_FS) {
997                 /* tsk->fs is already what we want */
998                 spin_lock(&fs->lock);
999                 if (fs->in_exec) {
1000                         spin_unlock(&fs->lock);
1001                         return -EAGAIN;
1002                 }
1003                 fs->users++;
1004                 spin_unlock(&fs->lock);
1005                 return 0;
1006         }
1007         tsk->fs = copy_fs_struct(fs);
1008         if (!tsk->fs)
1009                 return -ENOMEM;
1010         return 0;
1011 }
```

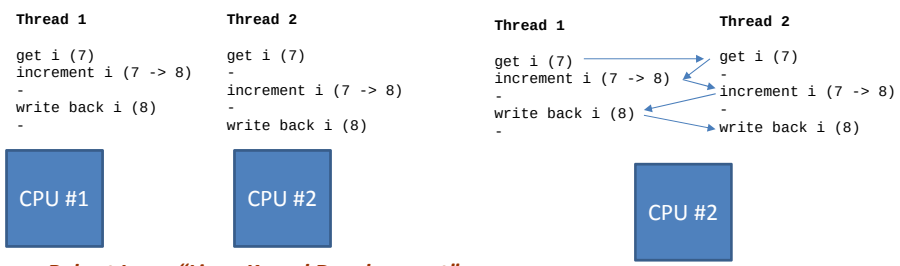
# Overview (Cont.)

- Race condition
  - Two threads execute simultaneously within the same critical region
- Synchronization
  - Prevent unsafe concurrency and avid race conditions

# Example

- i: shared data
- i++

| Thread 1 | Thread 2 |
|---|---|
| get i (7) | - |
| increment i (7 -> 8) | - |
| write back i (8) | - |
| - | get i (8) |
| - | increment i (8 -> 9) |
| - | write back i (9) |




*Source: Robert Love, "Linux Kernel Development"*

# Solution

- Atomic instruction
  - X86 instruction, compare and exchange (CMPXCHG)

| Thread 1 | Thread 2 |
|---|---|
| increment i(7 -> 8) | - |
| - | increment i (8 -> 9) |

or

| Thread 1 | Thread 2 |
|---|---|
| - | increment i (7 -> 8) |
| increment i(8 -> 9) | - |

*Source: Robert Love, "Linux Kernel Development"*

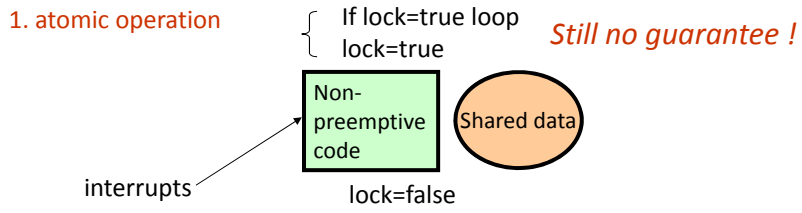
# Locking

- Using atomic instruction to handle shared variable is OK, how about data structure?

| Thread 1 | Thread 2 |
|---|---|
| try to lock the queue | try to lock the queue |
| succeed: acquired lock | failed: waiting … |
| access queue … | waiting … |
| unlock the queue | waiting … |
| ... | succeed: acquired lock |
| | access queue … |
| | unlock the queue |
| | ... |

*Source: Robert Love, "Linux Kernel Development"*

# Locking




2. disable interrupts vs. prevent preemption

# Causes of Concurrency

- Pseudo concurrency
  - Two threads do not actually execute simultaneously but interleave with each other (in critical region)
- True concurrency
  - Two threads execute simultaneously on two processors (SMP)

## Causes of Concurrency (Cont.)

- Interrupts
  - An interrupt can occur asynchronously at almost any time, interrupting the currently executing code
- Softirqs and tasklets
  - The kernel can raise or schedule a softirq or tasklet at almost any time, interrupting the currently executing code
- Kernel preemption
  - Because the kernel is preemptive, one task in the kernel can preempt another
- Sleeping and synchronization with user-space
  - A task in the kernel can sleep and thus invoke the scheduler, resulting in the running of a new process
- Symmetrical multiprocessing
  - Two or more processors can execute kernel code at exactly the same time

## Causes of Concurrency (Cont.)

- Interrupt-safe
  - Code that is safe from concurrent access from an interrupt handler
- SMP-safe
  - Code that is safe from concurrency on symmetrical multiprocessing machines
- preempt-safe
  - Code that is safe from concurrency with kernel preemption

# What do you protect?

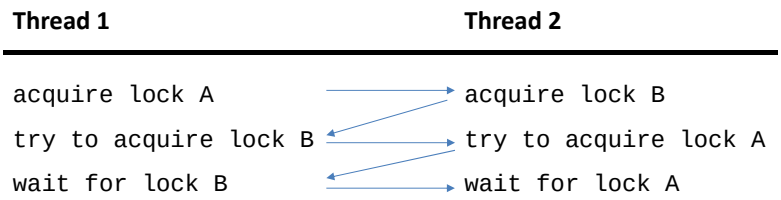
- Protect data (fs_struct), not the code

```
993 static int copy_fs(unsigned long clone_flags, struct task_struct *tsk)
994 {
995         struct fs_struct *fs = current->fs;
996         if (clone_flags & CLONE_FS) {
997                 /* tsk->fs is already what we want */
998                 spin_lock(&fs->lock);
999                 if (fs->in_exec) {
1000                         spin_unlock(&fs->lock);
1001                         return -EAGAIN;
1002                 }
1003                 fs->users++;
1004                 spin_unlock(&fs->lock);
1005                 return 0;
1006         }
1007         tsk->fs = copy_fs_struct(fs);
1008         if (!tsk->fs)
1009                 return -ENOMEM;
1010         return 0;
1011 }
```

# Questions to ask before locking

- Is the data global? Can a thread of execution other than the current one access it?
- Is the data shared between process context and interrupt context? Is it shared between two different interrupt handlers?
- If a process is preempted while accessing this data, can the newly scheduled process access the same data?
- Can the current process sleep (block) on anything? If it does, in what state does that leave any shared data?

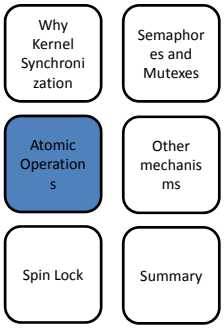
# Deadlocks




*Source: Robert Love, "Linux Kernel Development"*

# How to prevent?

- Atomicity vs. ordering
- Make sure lock orders
  - Implement lock ordering
    - Nested locks must always be obtained in the same order
  - Do not double acquire the same lock
- The order of unlock does not matter with respect to deadlock




# Atomic Operations

- Atomic integer operations
- Atomic bitwise operations

# How to protect critical section?

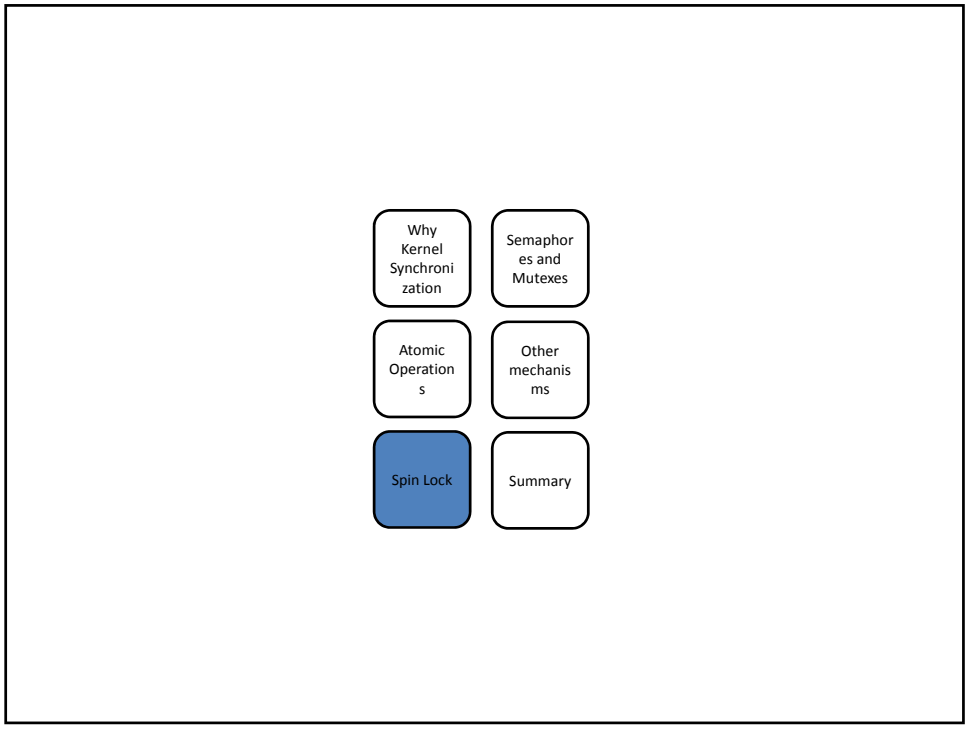
- Atomic Integer Operations

| | |
|---|---|
| ATOMIC_INIT(int i) | At declaration, initialize an atomic_t 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 |
| int atomic_sub_and_test(int i, atomic_t *v) | Atomically subtract i from v and return true if the result is zero; otherwise false |
| int atomic_add_negative(int i, atomic_t *v) | Atomically add i to v and return true if the result is negative; otherwise false |
| int atomic_dec_and_test(atomic_t *v) | Atomically decrement v by one and return true if zero; false otherwise |
| int atomic_inc_and_test(atomic_t *v) | Atomically increment v by one and return true if the result is zero; false otherwise |

# Atomic operation

- Atomic Bitwise Operations

| | |
|---|---|
| void set_bit(int nr, void *addr) | Atomically set the nr-th bit starting from addr |
| void clear_bit(int nr, void *addr) | Atomically clear the nr-th bit starting from addr |
| void change_bit(int nr, void *addr) | Atomically flip the value of the nr-th bit starting from addr |
| int test_and_set_bit(int nr, void *addr) | Atomically set the nr-th bit starting from addr and return the previous value |
| int test_and_clear_bit(int nr, void *addr) | Atomically clear the nr-th bit starting from addr and return the previous value |
| int test_and_change_bit(int nr, void *addr) | Atomically flip the nr-th bit starting from addr and return the previous value |
| int test_bit(int nr, void *addr) | Atomically return the value of the nr-th bit starting from addr |




# Spin lock

- spin_lock
  - Normally, critical section is more than just a variable, spin lock protects a sequence of codes manipulating the critical section
  - Perform busy loops spins for waiting the lock to become available
  - Spin_lock disables the **kernel preemption**
  - On UP, the locks compile away and do not exist; they simply act as markers to disable and enable kernel preemption
  - If kernel preempt is turned off, the locks compile away entirely

# spin_lock implementation on SMP

```
#define BUILD_LOCK_OPS(op, locktype)                                    \
void __lockfunc __raw_##op##_lock(locktype##_t *lock)                   \
{                                                                       \
        for (;;) {                                                      \
                preempt_disable();                                      \
                if (likely(do_raw_##op##_trylock(lock)))                \
                        break;                                          \
                preempt_enable();                                       \
                                                                        \
                if (!(lock)->break_lock)                                \
                        (lock)->break_lock = 1;                         \
                while (!raw_##op##_can_lock(lock) && (lock)->break_lock)\
                        arch_##op##_relax(&lock->raw_lock);             \
        }                                                               \
        (lock)->break_lock = 0;                                         \
}                                                                       \
```

# spin_lock implementation on UP

```
#define ___LOCK(lock) \
  do { __acquire(lock); (void)(lock); } while (0)

#define __LOCK(lock) \
  do { preempt_disable(); ___LOCK(lock); } while (0)

// ....skipped some lines.....
#define _raw_spin_lock(lock)                    __LOCK(lock)
```

## Spin lock

- Principles for using spin lock
  - Lock the data, not the code
  - Don't hold a spin lock for a long time (because someone may wait outside)
  - Don't recursively spin lock in Linux (like you lock your key in your car)
  - Spin lock (instead of using mutex) if you can finish within 2 context switches time
  - In a preemptive kernel, spin lock will perform preempt_disable( )
  - Can be used in interrupt handler (mutex/semaphores cannot) *must disable **"local"** interrupt*

## Spin lock

- When you use spin_lock(x) in the kernel, make sure you don't spend too much time in the lock, and there is no recursive spin_lock(x)
  - No spin_lock(x) in the spin_lock(x) period
- When you use spin_lock(x) in interrupt handler, make sure the interrupt has been disabled. Otherwise, use spin_lock_irq(x) or spin_lock_irqsave(x)

## Considering following situations

- You are writing kernel code (process context)
  - Shared data between processes
- You are writing interrupt handler (interrupt context)
  - Shared data between interrupt handler
- You are writing kernel code or interrupt handler
  - Shared data between interrupt handlers and processes
- *How could you use spin_lock to protect shared data?*

## Spin lock irq

- Disables interrupts and acquires the lock
- spin_unlock_irq() unlocks the given lock
- Disable interrupts (spin_lock_irq) and enable interrupts anyway (spin_lock_irqsave) no matter the status of current interrupts
  - Use of spin_lock_irq() therefore is not recommended

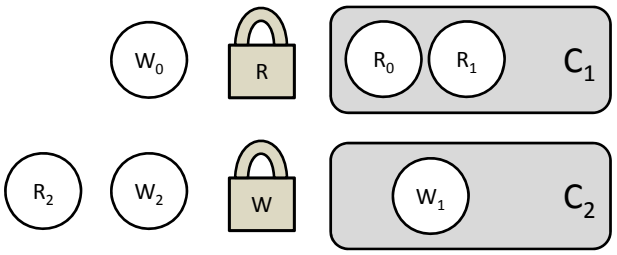
## Spin lock irqsave

- Disables interrupt if it is not disabled and acquires the lock
- spin_unlock_irqrestore () unlocks the given lock and returns interrupts to their previous state

## Spin lock methods

| | |
|---|---|
| spin_lock() | Acquires given lock |
| spin_lock_irq() | Disables local interrupts and acquires given lock |
| spin_lock_irqsave() | Saves current state of local interrupts, disables local interrupts, and acquires given lock |
| spin_unlock() | Releases given lock |
| spin_unlock_irq() | Releases given lock and enables local interrupts |
| spin_unlock_irqrestore() | Releases given lock and restores local interrupts to given previous state |
| spin_lock_init() | Dynamically initializes given spinlock_t |
| spin_trylock() | Tries to acquire given lock; if unavailable, returns nonzero |
| spin_is_locked() | Returns nonzero if the given lock is currently acquired, otherwise it returns zero |

# Read/Write Spin Locks




$R_n$ – Reader kernel control path
$W_n$ –Writer kernel control path
$C_n$ – Critical region

*Source: Understanding the Linux Kernel, Daniel P. Bovet, Marco Cesati*

# Read write spin lock

- Lock usage can be clearly divided into readers and writers (e.g. search and update)
- Linux task list is protected by a reader-writer spin lock

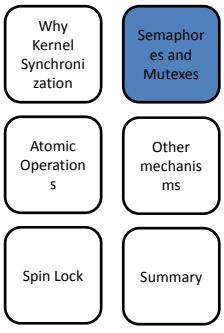
```
read_lock(&mr_rwlock);                  write_lock(&mr_rwlock);
/* critical section (read only) ... */  /* critical section (read and write) ... */
read_unlock(&mr_rwlock);                write_unlock(&mr_lock);
```

- Must split into reader/writer paths
  - You cannot do this

```
read_lock(&mr_rwlock);
write_lock(&mr_rwlock);
```

# Read write spin lock

| Method | Description |
|---|---|
| read_lock() | Acquires given lock for reading |
| read_lock_irq() | Disables local interrupts and acquires given lock for reading |
| read_lock_irqsave() | Saves the current state of local interrupts, disables local interrupts, and acquires the given lock for reading |
| read_unlock() | Releases given lock for reading |
| read_unlock_irq() | Releases given lock and enables local interrupts |
| read_unlock_irqrestore() | Releases given lock and restores local interrupts to the given previous state |
| write_lock() | Acquires given lock for writing |
| write_lock_irq() | Disables local interrupts and acquires the given lock for writing |
| write_lock_irqsave() | Saves current state of local interrupts, disables local interrupts, and acquires the given lock for writing |
| write_unlock() | Releases given lock |
| write_unlock_irq() | Releases given lock and enables local interrupts |
| write_unlock_irqrestore() | Releases given lock and restores local interrupts to given previous state |
| write_trylock() | Tries to acquire given lock for writing; if unavailable, returns nonzero |
| rw_lock_init() | Initializes given rwlock_t |
| rw_is_locked() | Returns nonzero if the given lock is currently acquired, or else it returns zero |



## Semaphores

- sleeping locks
- do not disable kernel preemption
- the mutex/semaphore place the task onto a wait queue and put the task to sleep
- well suited to locks that are held for a long time
- not optimal for locks that are held for very short periods
- can be obtained only in process context, ***not interrupt context***
- can sleep while holding a semaphore
- **cannot hold a spin lock while you acquire a semaphore**
  - **How about hold a semaphore while you acquire a spin lock**

## Semaphores (Cont.)

- binary semaphore vs. counting semaphore (*not used much in the kernel*)
- down_interruptible()
  - attempts to acquire the given semaphore. If it fails, it sleeps in the TASK_INTERRUPTIBLE
- down()
  - If it fails, places the task in the TASK_UNINTERRUPTIBLE state if it sleeps (*normally not used*)

# Semaphores

| | |
|---|---|
| sema_init(struct semaphore *, int) | Initializes the dynamically created semaphore to the given count |
| init_MUTEX(struct semaphore *) | Initializes the dynamically created semaphore with a count of one |
| init_MUTEX_LOCKED(struct semaphore *) | Initializes the dynamically created semaphore with a count of zero (so it is initially locked) |
| down_interruptible(struct semaphore *) | Tries to acquire the given semaphore and enter interruptible sleep if it is contended |
| down(struct semaphore *) | Tries to acquire the given semaphore and enter uninterruptible sleep if it is contended |
| down_trylock(struct semaphore *) | Tries to acquire the given semaphore and immediately return nonzero if it is contended |
| up(struct semaphore *) | Releases the given semaphore and wakes a waiting task, if any |

# Implementation

```
1  void down(struct semaphore *sem)
2  {
3          unsigned long flags;
5          raw_spin_lock_irqsave(&sem->lock, flags);
6          if (likely(sem->count > 0))
7                  sem->count--;
8          else
9                  __down(sem);
10         raw_spin_unlock_irqrestore(&sem->lock, flags);
11 }
```

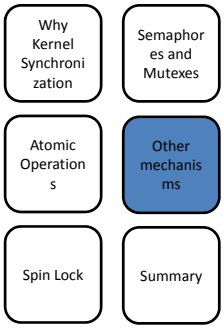
```
1  static noinline void __sched __down(struct semaphore *sem)
2  {
3          __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
4  }
6  static inline int __sched __down_common(struct semaphore *sem, long s
7                                                                  long tim
8  {
9          struct task_struct *task = current;
10         struct semaphore_waiter waiter;
12         list_add_tail(&waiter.list, &sem->wait_list);
13         waiter.task = task;
14         waiter.up = 0;
16         for (;;) {
17                 if (signal_pending_state(state, task))
18                         goto interrupted;
19                 if (timeout <= 0)
20                         goto timed_out;
21                 __set_task_state(task, state);
22                 raw_spin_unlock_irq(&sem->lock);
23                 timeout = schedule_timeout(timeout);
24                 raw_spin_lock_irq(&sem->lock);
25                 if (waiter.up)
26                         return 0;
27         }
29  timed_out:
30         list_del(&waiter.list);
31         return -ETIME;
33  interrupted:
34         list_del(&waiter.list);
35         return -EINTR;
36 }
```

# Mutex

- Behaves similar to a semaphore with a count of one, but it has a simpler interface, more efficient performance

```
mutex_lock(&mutex);
/* critical region ... */
mutex_unlock(&mutex);
```

| Method | Description |
|---|---|
| `mutex_lock(struct mutex *)` | Locks the given mutex; sleeps if the lock is unavailable |
| `mutex_unlock(struct mutex *)` | Unlocks the given mutex |
| `mutex_trylock(struct mutex *)` | Tries to acquire the given mutex; returns one if successful and the lock is acquired and zero otherwise |
| `mutex_is_locked (struct mutex *)` | Returns one if the lock is locked and zero otherwise |




# Other Mechanisms

- Completion variables
  - an easy way to synchronize between two tasks in the kernel when one task needs to signal to the other that an event has occurred
- Sequential locks
  - shortened to seq lock, is a newer type of lock introduced in the 2.6 kernel. It provides a simple mechanism for reading and writing shared data

## Other Mechanisms (Cont.)

- Interrupt disable
  - Allows a kernel control path to continue executing even when hardware issues IRQ signals
- Preemptions disable
  - For per-processor data access. If no spin locks are held, the kernel is preemptive, and it would be possible for a newly scheduled task to access this same variable

## Other Mechanisms (Cont.)

- Memory barriers
  - May order

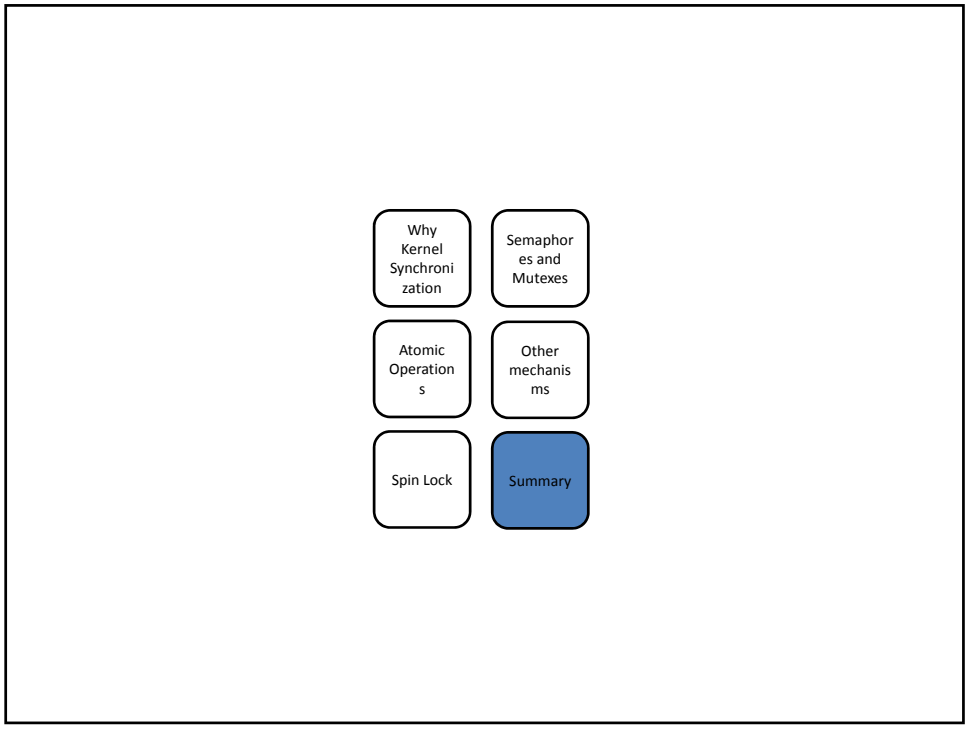
```
a = 1;
b = 2;
```

  - Will not reorder

```
a = 1;
b = a;
```

  - Example, a=1, b=2, no mb(), rmb()

| Thread 1 | Thread 2 |
|---|---|
| `a = 3;` | `-` |
| `mb();` | `-` |
| `b = 4;` | `c = b;` |
| `-` | `rmb();` |
| `-` | `d = a;` |




# Protection of Shared Data in Kernel

| Requirement | Recommended Lock |
|---|---|
| Low overhead locking | Spin lock |
| Short lock hold time | Spin lock |
| Long lock hold time | Mutex |
| Need to lock from interrupt context | Spin lock |
| Need to sleep while holding lock | Mutex |

*Source: Robert Love, "Linux Kernel Development"*

# Protection of Shared Data in Kernel

| Kernel control paths accessing the data structure | UP protection | MP further protection |
|---|---|---|
| Exceptions | Semaphore | None |
| Interrupts | Local interrupt disabling | Spin lock |
| Deferrable functions | None | None or spin lock |
| Exceptions + Interrupts | Local interrupt disabling | Spin lock |
| Exceptions + Deferrable functions | Local softirq disabling | Spin lock |
| Interrupts + Deferrable functions | Local interrupt disabling | Spin lock |
| Exceptions + Interrupts + Deferrable functions | Local interrupt disabling | Spin lock |

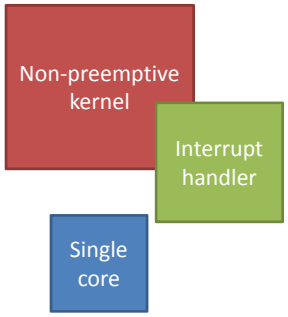
***Will discuss deferrable functions (softirq and tasklet latter)***

***Source: Understanding the Linux Kernel, Daniel P. Bovet, Marco Cesati***

# Protection of Shared Data in Kernel

- Exception handler
  - Such as system call, data can be shared by processes

# Single Core + Non-preemptive kernel



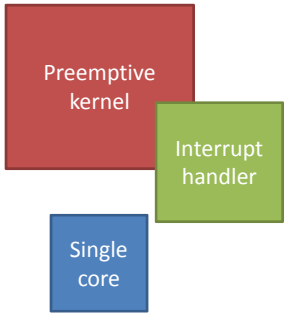

**Non-preemptive kernel**

| Sync methods | Notes |
|---|---|
| local_irq_disable | Lock for short period |
| local_irq_save | Lock for short period, preferred |
| spin_lock | X |
| spin_lock_irq | =local_irq_disable |
| spin_lock_irqsave | =local_irq_save |
| Semaphore/mutex | Lock for long period |

**Interrupt handler**

| Sync methods | Notes |
|---|---|
| local_irq_disable | Lock for short period |
| local_irq_save | Lock for short period, preferred |
| spin_lock | X |
| spin_lock_irq | =local_irq_disable |
| spin_lock_irqsave | =local_irq_save |
| Semaphore/mutex | X |

# Single Core + Preemptive kernel



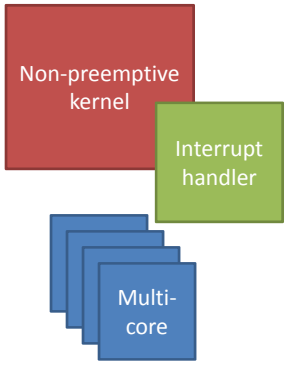

**Preemptive kernel**

| Sync methods | Notes |
|---|---|
| local_irq_disable | OK for UP |
| local_irq_save | OK for UP (preferred) |
| spin_lock | OK, interrupt may occur |
| spin_lock_irq | OK, short period lock |
| spin_lock_irqsave | OK, short period lock (preferred) |
| Semaphore/mutex | OK, long period lock |

**Interrupt handler**

| Sync methods | Notes |
|---|---|
| local_irq_disable | OK for UP |
| local_irq_save | OK for UP (preferred) |
| spin_lock | OK, interrupt may occur |
| spin_lock_irq | OK, short period lock |
| spin_lock_irqsave | OK, short period lock (preferred) |
| Semaphore/mutex | No |

# Multi-Core + Non-preemptive kernel



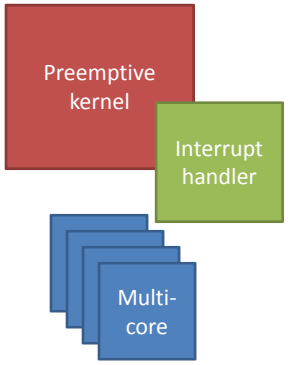

**Non-preemptive kernel**

| Sync methods | Notes |
|---|---|
| local_irq_disable | |
| local_irq_save | |
| spin_lock | |
| spin_lock_irq | |
| spin_lock_irqsave | |
| Semaphore/mutex | |

**Interrupt handler**

| Sync methods | Notes |
|---|---|
| local_irq_disable | |
| local_irq_save | |
| spin_lock | |
| spin_lock_irq | |
| spin_lock_irqsave | |
| Semaphore/mutex | |

# Multi-Core + Preemptive kernel




**Preemptive kernel**

| Sync methods | Notes |
|---|---|
| local_irq_disable | |
| local_irq_save | |
| spin_lock | |
| spin_lock_irq | |
| spin_lock_irqsave | |
| Semaphore/mutex | |

**Interrupt handler**

| Sync methods | Notes |
|---|---|
| local_irq_disable | |
| local_irq_save | |
| spin_lock | |
| spin_lock_irq | |
| spin_lock_irqsave | |
| Semaphore/mutex | |