Dealing with Concurrency in the Kernel

Advanced Operating Systems and Virtualization

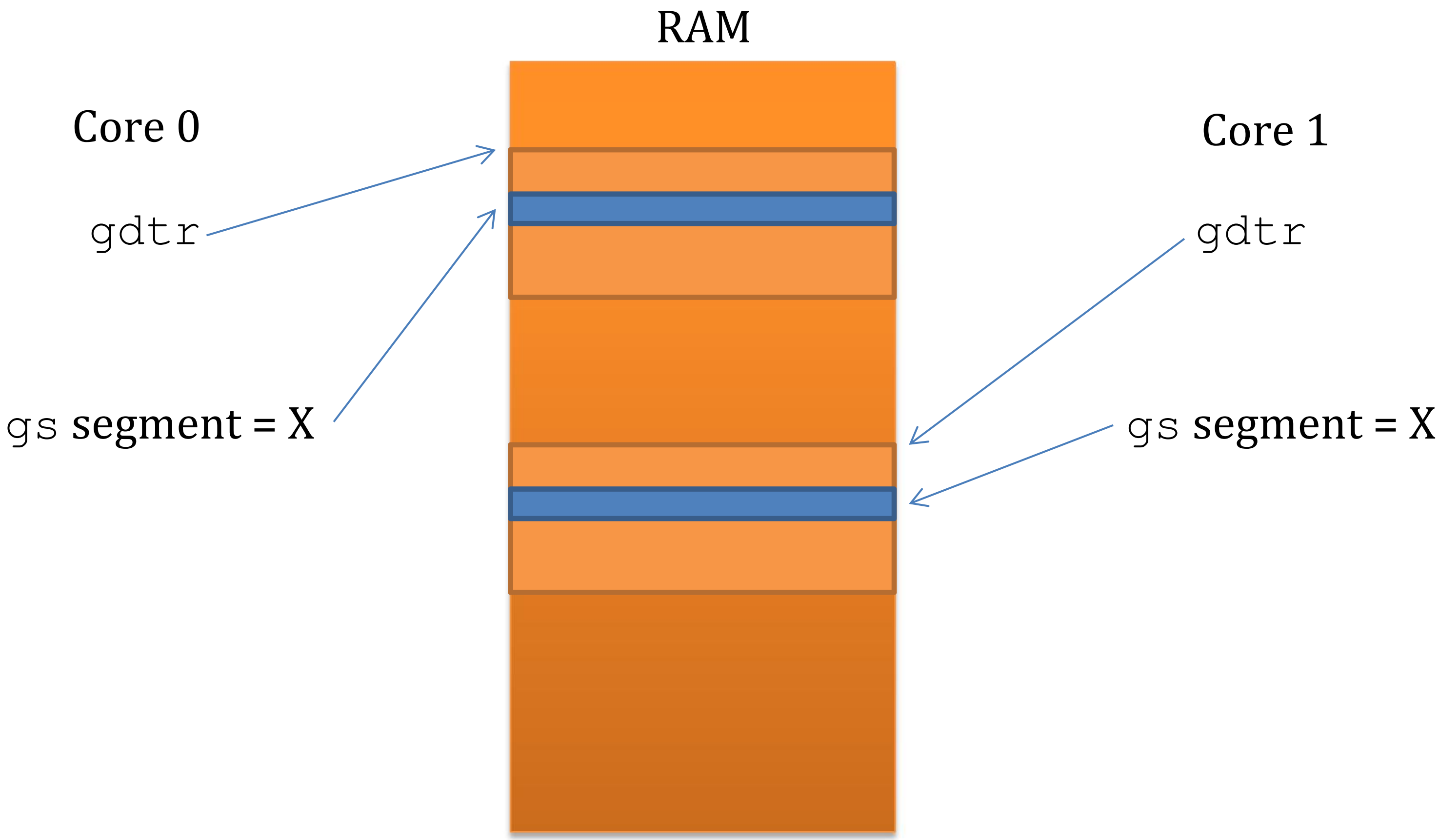
Alessandro Pellegrini

A.Y. 2017/2018

Big Kernel Lock

- Traditionally called a "Giant Lock"
- This is a simple way to provide concurrency to userspace avoiding concurrency problems in the kernel
- Whenever a thread enters kernel mode, it acquires the BKL
  - No more than one thread can live in kernel space
- Completely removed in 2.6.39

Per-CPU Variables



different bases!

Per-CPU Variables

```
DEFINE_PER_CPU(int, x);
int z;
z = this_cpu_read(x);
```

- This is mapped to a single istruction:
  - `mov %gs:x,%eax`

```
y = this_cpu_ptr(&x);
```

Linux Mutexes

```
DECLARE_MUTEX(name);
/* declares struct semaphore <name> ... */

void sema_init(struct semaphore *sem, int val);
/* alternative to DECLARE_... */

void down(struct semaphore *sem); /* may sleep */

int down_interruptible(struct semaphore *sem);
/* may sleep; returns -EINTR on interrupt */

int down_trylock(struct semaphone *sem);
/* returns 0 if succeeded; will no sleep */

void up(struct semaphore *sem);
```

Linux Spinlocks

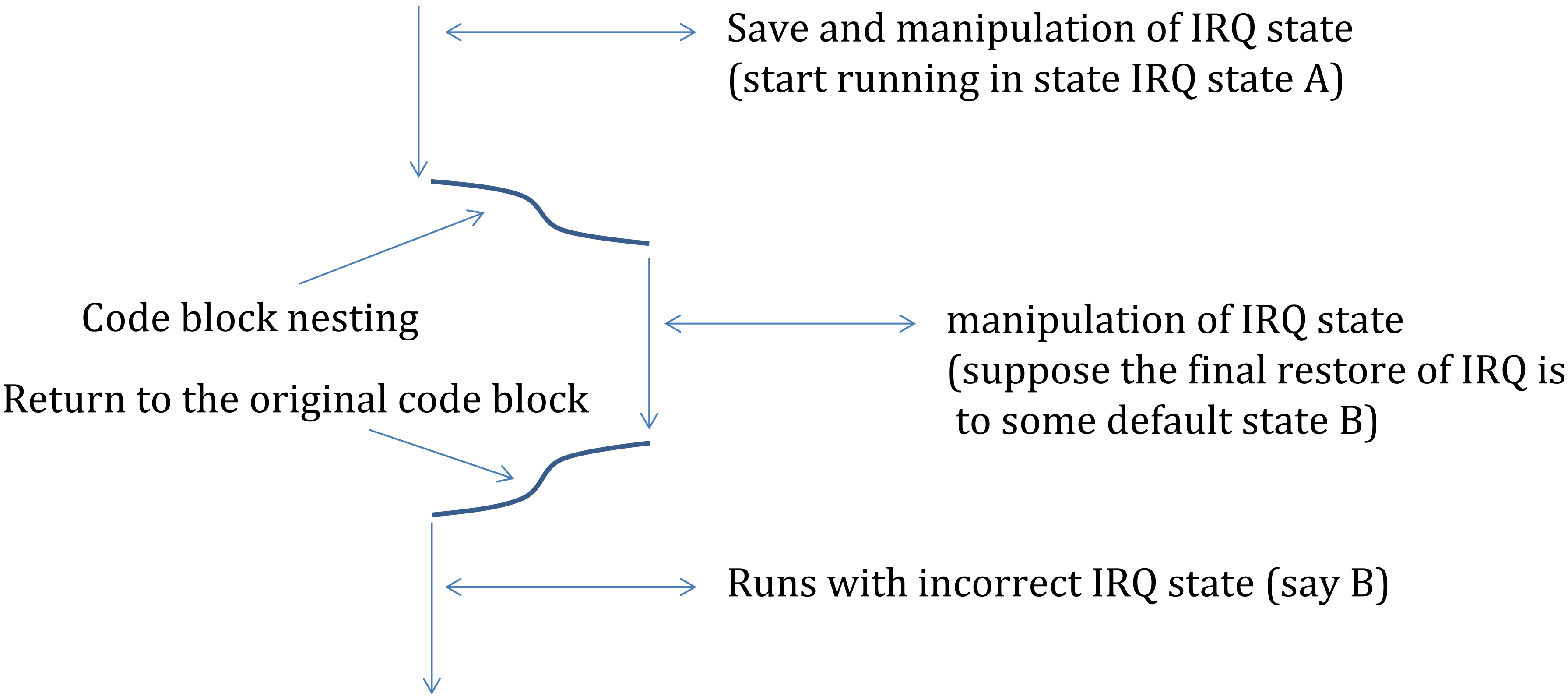
```
#include <linux/spinlock.h>

spinlock_t my_lock = SPINLOCK_UNLOCKED;
spin_lock_init(spinlock_t *lock);
spin_lock(spinlock_t *lock);
spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
spin_lock_irq(spinlock_t *lock);
spin_lock_bh(spinlock_t *lock);

spin_unlock(spinlock_t *lock);
spin_unlock_irqrestore(spinlock_t *lock,
                                 unsigned long flags);
spin_unlock_irq(spinlock_t *lock);
spin_unlock_bh(spinlock_t *lock);
spin_is_locked(spinlock_t *lock);
spin_trylock(spinlock_t *lock)
spin_unlock_wait(spinlock_t *lock);
```

The "save" version

- it allows not to interfere with IRQ management along the path where the call is nested
- a simple masking (with no saving) of the IRQ state may lead to misbehavior



Read/Write Locks

```
rwlock_t xxx_lock = __RW_LOCK_UNLOCKED(xxx_lock);
unsigned long flags;

read_lock_irqsave(&xxx_lock, flags);
.. critical section that only reads the info ...
read_unlock_irqrestore(&xxx_lock, flags);


write_lock_irqsave(&xxx_lock, flags);
.. read and write exclusive access to the info ...
write_unlock_irqrestore(&xxx_lock, flags);
```

Read/Write Locks

Read

Get Lock:

- Lock *r*
- Increment *c*
- if *c* == 1
  - lock *w*
- unlock *r*

Release Lock:

- Lock *r*
- Decrement *c*
- if *c* == 0
  - unlock *w*
- unlock *r*

Write

Get Lock:

- Lock *w*

Release Lock:

- Unlock *w*

seqlocks

- A seqlock tries to tackle the following situation:
  - A small amount of data is to be protected.
  - That data is simple (no pointers), and is frequently accessed.
  - Access to the data does not create side effects.
  - It is important that writers not be starved for access.
- It is a way to avoid readers to starve writers

seqlocks

- `#include <linux/seqlock.h>`
- `seqlock_t lock1 = SEQLOCK_UNLOCKED;`
- `seqlock_t lock2;`
- `seqlock_init(&lock2);`

Exclusive access and increment the sequence number

- `write_seqlock(&the_lock);`
- `/* Make changes here */`

increment again

- `write_sequnlock(&the_lock);`

seqlocks

- Readers do not acquire a lock:

```
unsigned int seq;
do {
   seq = read_seqbegin(&the_lock);
   /* Make a copy of the data of interest */
} while read_seqretry(&the_lock, seq);
```

- The call to `read_seqretry` checks whether the initial number was odd
- It additionally checks if the sequence number has changed

Atomic Operations

- `atomic_t` **type**
  - `atomic_fetch_{add,sub,and,andnot,or,xor}()`
- `DECLARE_BITMAP()` **macro**
  - `set_bit()`
  - `clear_bit()`
  - `test_and_set_bit()`
  - `test_and_clear_bit()`
- All based on RMW instructions

Read-Copy-Update (RCU)

- This is a synchronization mechanism added in October 2002
- Scalability is enforced by having readers concurrently perform operations to writers
- RCU ensures that reads are coherent by maintaining multiple versions of objects and ensuring that they are not freed up until all pre-existing read-side critical sections complete

Read-Copy-Update (RCU)

- Three fundamental mechanisms:
  - Publish-subscribe mechanism (for insertion)
  - Wait for pre-existing RCU readers to complete (for deletion)
  - Maintain multiple versions of RCU-updated objects (for readers)

Insertion

```
struct foo {
   int a;
   int b;
   int c;
 };
 struct foo *gp = NULL;

 /* . . . */

 p = kmalloc(sizeof(*p), GFP_KERNEL);
 p->a = 1;
 p->b = 2;
 p->c = 3;
 gp = p;
```

Is this always correct?

Insertion

```
struct foo {
   int a;
   int b;
   int c;
 };
 struct foo *gp = NULL;

 /* . . . */

 p = kmalloc(sizeof(*p), GFP_KERNEL);
 p->a = 1;
 p->b = 2;
 p->c = 3;
 rcu_assign_pointer(gp, p)
```

the "publish" part

Reading

```
p = gp;
if (p != NULL) {
  do_something_with(p->a, p->b, p->c);
}
```

Is this always correct?

Reading

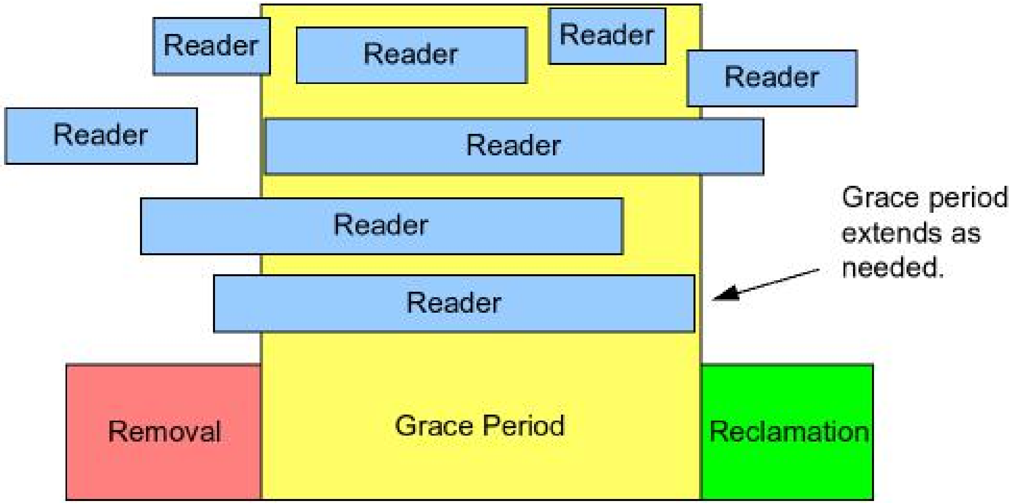
```
rcu_read_lock();
p = rcu_dereference(gp);
if (p != NULL) {
  do_something_with(p->a, p->b, p->c);
}
rcu_read_unlock();
```

Memory barriers here

Wait Pre-Existing RCU Updates

- `synchronize_rcu()`
- It can be schematized as:

```
for_each_online_cpu(cpu)
        run_on(cpu);
```



Wait Pre-Existing RCU Updates

```
struct foo {
   struct list_head list;
   int a;
   int b;
   int c;
 };
 LIST_HEAD(head);

 /* . . . */

 p = search(head, key);
 if (p == NULL) {
   /* Take appropriate action, unlock, and return. */
 }
 q = kmalloc(sizeof(*p), GFP_KERNEL);
 *q = *p;
 q->b = 2;
 q->c = 3;
 list_replace_rcu(&p->list, &q->list);
 synchronize_rcu();
 kfree(p);
```

Multiple Concurrent RCU Updates

```
struct foo {
   struct list_head list;
   int a;
   int b;
   int c;
 };
 LIST_HEAD(head);

 /* . . . */

 p = search(head, key);
 if (p == NULL) {
   /* Take appropriate action, unlock, and return. */
 }
 q = kmalloc(sizeof(*p), GFP_KERNEL);
 *q = *p;
 q->b = 2;
 q->c = 3;
 list_replace_rcu(&p->list, &q->list);
 synchronize_rcu();
 kfree(p);
```

```
p = search(head, key);
if (p != NULL) {
  list_del_rcu(&p->list);
  synchronize_rcu();
  kfree(p);
}
```