# Dealing with Concurrency in the Kernel

Advanced Operating Systems and Virtualization

Alessandro Pellegrini

A.Y. 2019/2020

# Concurrency Properties

- **Safety**: nothing wrong happens
  - It's called **Correctness** as well
  - What does it mean for a program to be correct?
    - What's exactly a concurrent FIFO queue?
    - FIFO implies a strict temporal ordering
    - Concurrent implies an ambiguous temporal ordering
  - Intuitively, if we rely on locks, changes happen in a non-interleaved fashion, resembling a sequential execution
  - We can say a parallel execution is correct only because we can associate it with a sequential one, which we know the functioning of
- **Liveness**: eventually something good happens
  - It's called **Progress** as well
  - Opposed to Starvation

# Correctness Conditions

- The **linearizability** property tries to generalize the intuition of correctness
- A *history* is a sequence of invocations and replies generated on an object by a set of threads
- A *sequential history* is a history where all the invocations have an immediate response
- A history is called *linearizable* if:
  - Invocations/responses can be reordered to create a sequential history
  - The so-generated sequential history is correct according to the sequential definition of the object
  - If a response precedes an invocation in the original history, then it must precede it in the sequential one as well
- An object is linearizable if every valid history associated with its usage can be linearized

# Progress Conditions

- **Deadlock-free**:
  - Some thread acquires a lock eventually
- **Starvation-free**:
  - Every thread acquires a lock eventually
- **Lock-free**:
  - Some method call completes
- **Wait-free**:
  - Every method call completes
- **Obstruction-free**:
  - Every method call completes, if they execute in isolation

# Progress Taxonomy

| | Non-Blocking | | Blocking |
|---|---|---|---|
| **For Everyone** | Wait-Free | Obstruction-Free | Starvation-Free |
| **For Some** | Lock-free | | Deadlock-Free |

Progress conditions on **multiprocessors**:

- Are not about guarantees provided by a method implementation
- Are about the *scheduling support* needed to provide maximum or minimum progress

# Progress Taxonomy

| | Non-Blocking | | Blocking |
|---|---|---|---|
| **For Everyone** | Nothing | Thread executes alone | No thread locked in CS |
| **For Some** | Nothing | | No thread locked in CS |

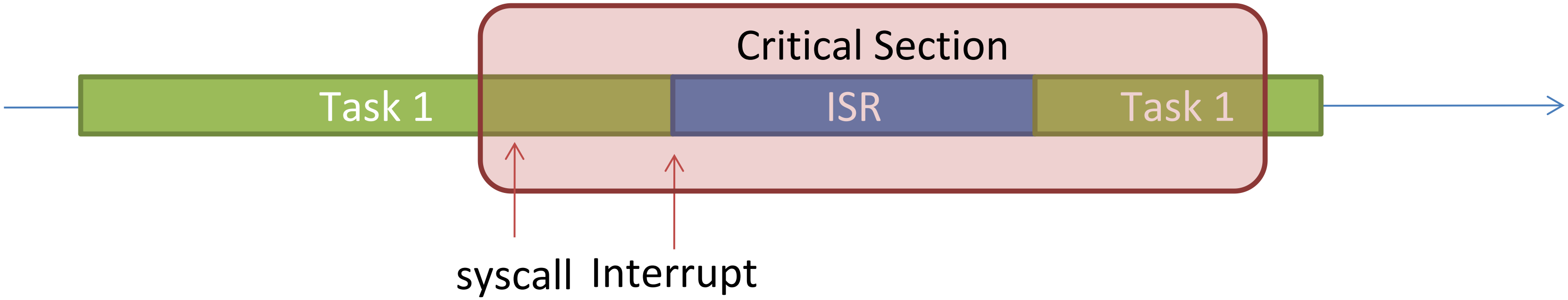
# Concurrent and Preëmptive Kernels

- Modern kernels are preëmptive
  - A process running in kernel mode might be replaced by another process while in the middle of a kernel function
- Modern kernels run concurrently
  - Any core can run kernel functions at any time
- Kernel code must ensure consistency and avoid deadlock
- Typical solutions:
  - Explicit synchronization
  - Non-blocking synchronization
  - Data separation (e.g., per-CPU variables)
  - Interrupt disabling
  - Preëmption disabling

Mandatory on multi-core machine

# Kernel Race Conditions

- System calls and interrupts



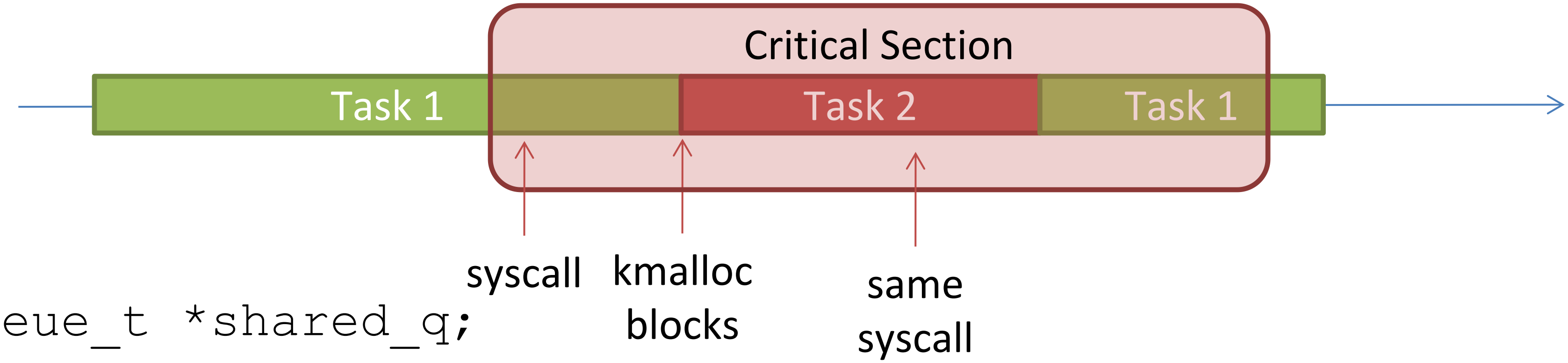

```
queue_t *shared_q;

my_irq_handler() {
        data = io(...);
        push(shared_q, data);
}
```

```
my_syscall() {
    n = length(shared_q);
    if(n > 0) {
        buf = kmalloc(1024);
        pop(shared_q, buf);
    }
}
```

# Kernel Race Conditions

- System calls and preëmption




```
queue_t *shared_q;
```

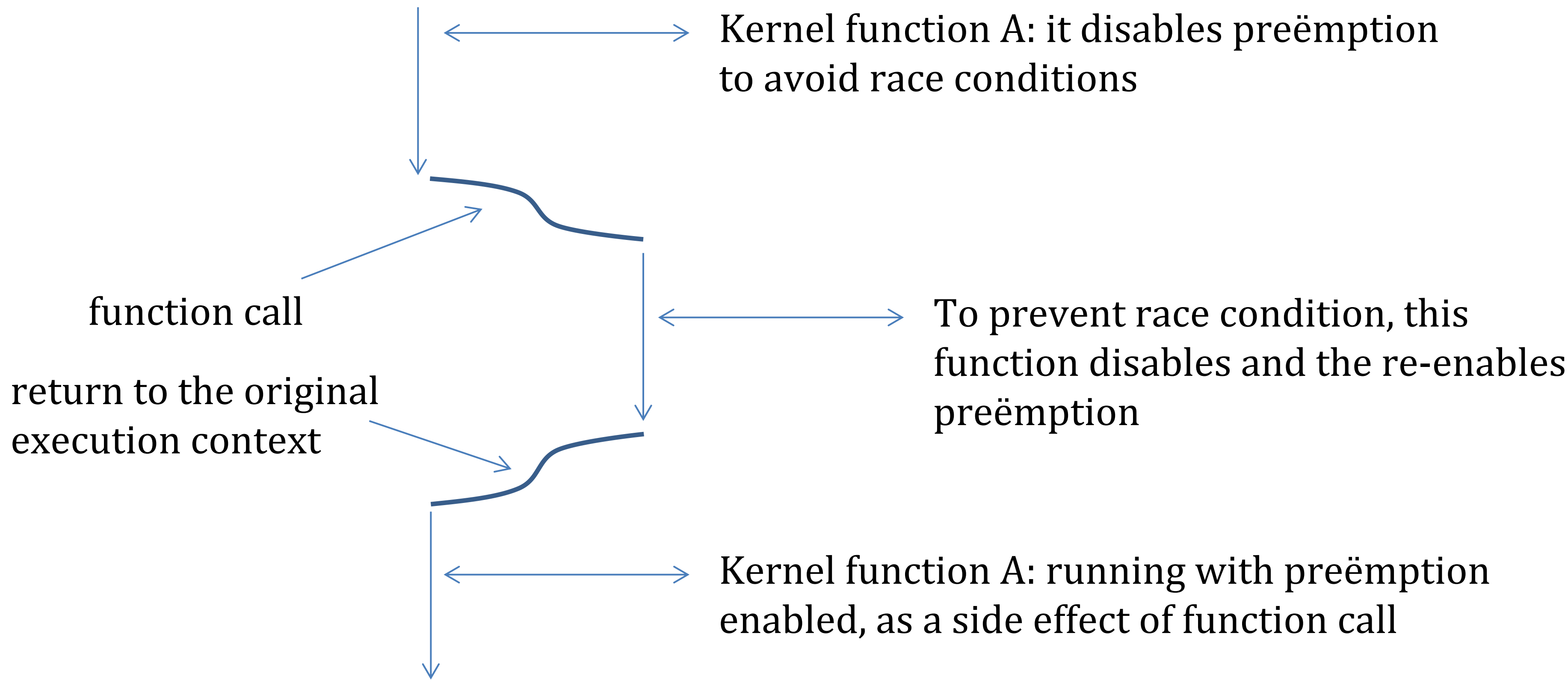
```
my_syscall() {
    n = length(shared_q);
    if(n > 0) {
        buf = kmalloc(1024);
        pop(shared_q, buf);
    }
}
```

# Enabling/Disabling Preëmption

- Kernel preëmption might take place when the scheduler is activated
- There must be a way to disable preëmption
  - This is based on a (per-CPU) counter
  - A non-zero counter tells that preëmption is disabled
- `preempt_count()`: return the current's core counter
- `preempt_disable()`: increases by one the preëmption counter (needs a memory barrier).
- `preempt_enable()`: decreases by one the preëmption counter (needs a memory barrier).

# Why do we need counters?

- In a Kernel with no preëmption counters this is possible:



# Enabling/Disabling HardIRQs

- Given the per-CPU management of interrupts, HardIRQs can be disabled only locally
- Managing the IF flags:
  - `local_irq_disable()`
  - `local_irq_enable()`
  - `irqs_disabled()`
- Nested activations (same concept as in the preëmption case):
  - `local_irq_save(flags)`
  - `local_irq_restore(flags)`

# The `__save` Version

```
#define raw_local_irq_save(flags)                          \
        do {                                               \
                typecheck(unsigned long, flags);           \
                flags = arch_local_irq_save();             \
        } while (0)

extern inline unsigned long native_save_fl(void)
{
        unsigned long flags;
        asm volatile("pushf ; pop %0"
                     : "=rm" (flags)
                     : /* no input */
                     : "memory");
        return flags;
}
```

Why cannot we rely on counters as in the case of preëmption disabling?

# Per-CPU Variables

- A support to implement “data separation” in the kernel
- It is the best “synchronization” technique
  - It removes the need for explicit synchronization
- They are not silver bullets
  - No protection againts asynchronous functions
  - No protection against preëmption and reschedule on another core

# Atomic Operations

- Based on RMW instructions
- `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()`

# Memory Barriers

- A compiler might reorder the instructions
  - Typically done to optimize the usage of registers
- Out of order pipeline and Memory Consistency models can reorder memory accesses
- Two families of barriers:
  - *Optimization* barriers
    - `#define barrier() asm volatile(“”:::”memory”);`
  - *Memory* barriers
    - `{smp_}mb()`: full memory barrier
    - `{smp_}rmb()`: read memory barrier
    - `{smp_}wmb()`: write memory barrier

Add fences if necessary

# 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

# 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

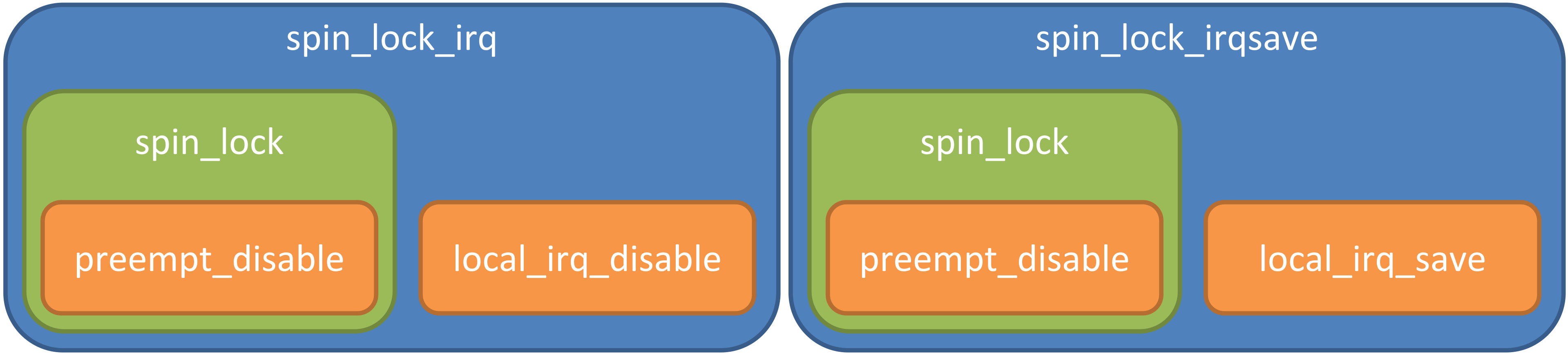
```
#include <linux/spinlock.h>

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);
```

# Linux Spinlocks

```
static inline void __raw_spin_lock_irq(raw_spinlock_t *lock)
{
    local_irq_disable();
    preempt_disable();
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
}
```



# 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 $w$

# 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);
```

# 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 */`
- `write_sequnlock(&the_lock);`

increment again

# 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

# Read-Copy-Update (RCU)

- Another synchronization mechanism, added in October 2002
- 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
- RCU allow many readers and many writers to proceed concurrently
- RCU is lock-free (no locks nor counters are used)
  - Increased scalability, no cache contention on synchronization variables

# 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)
- RCU scope:
  - Only dynamically allocated data structures can be protected by RCU
  - No kernel control path can sleep inside a critical section protected by RCU

# 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?

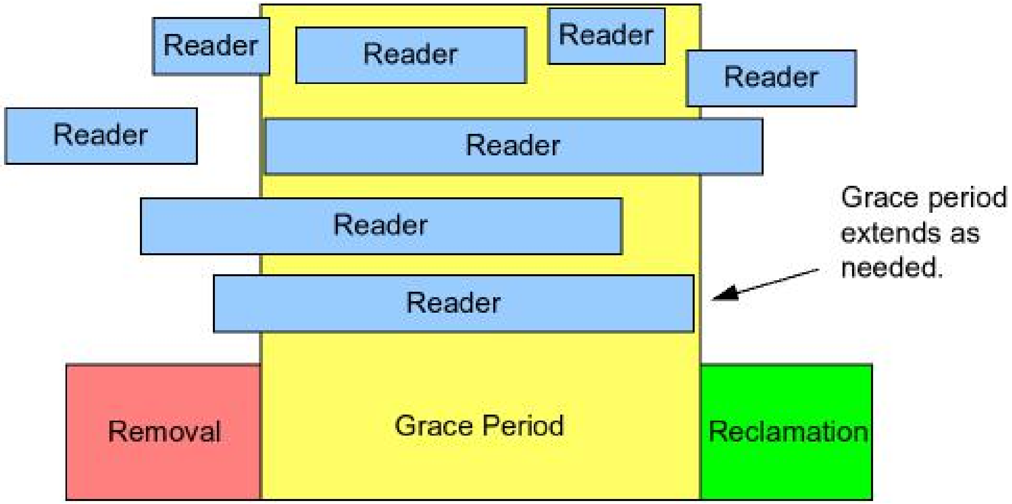
# Reading

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

# Wait Pre-Existing RCU Updates

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

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

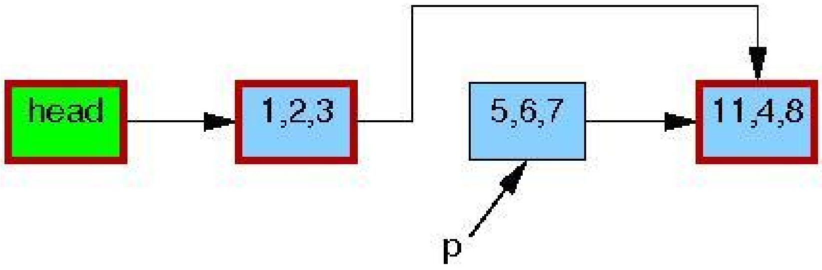


# Multiple Versions: Deletion

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

 /* . . . */

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

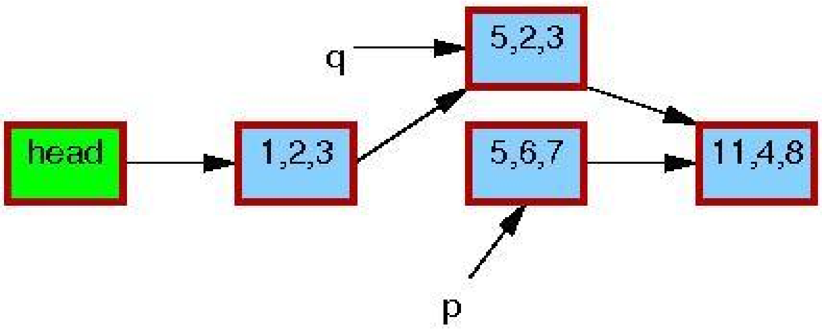


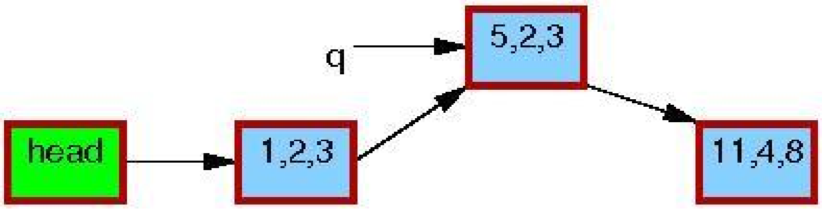
# Multiple Versions: Update

```
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);
```



# RCU Garbage Collection

- An old version of a data structure can be still accessed by readers
  - It can be freed only after that all readers have called `rcu_read_unlock()`
- A writer cannot waste to much time waiting for this condition
- `call_rcu()` registers a callback function to free the old data structure
- Callbacks are activated by a dedicated SoftIRQ action

# RCU vs RW Locks