Process Management
Kernel Level Synchronization

Table of contents

Kernel threads

Kernel threads are **standard processes** that exist only in the **kernel space** and are used by the kernel to perform certain operations in the background (e.g. writing cache blocks to disk, sweeping unused pages to disk).

They are implemented as **ordinary processes** that **share** certain resources with other processes (such as the address space).

The difference between kernel threads and normal processes lies in the fact that for kernel threads the pointer to **memory descriptor** (**mm**) is **NULL**.

They work only in the **kernel space** and never switch context to the user space.

They are subject to **scheduling** and can be **preempted** – just like ordinary processes.

They are created, for example, at the **initial startup** of the system, when **loading** some **modules**, **mounting** some **devices**, **mounting** some **filesystems**.

Usually, they perform their functions in an **infinite loop**. They exist until the system is closed.

Kernel threads

The **kernel_thread()** function creates a **new kernel thread**.

```
pid_t  kernel_thread(int (*fn)(void *), void *arg, const char *name, unsigned long flags)
{
        struct kernel_clone_args args = {
                .flags = ((lower_32_bits(flags) | CLONE_VM | CLONE_UNTRACED) & ~CSIGNAL),
                …
        }
        return kernel_clone(args);
}
```

Simplified version

The ancestor of all processes is **process 0**, also called **idle** or **swapper**. This is the kernel thread **created manually** during the initialization of Linux data structures.

At the end of the system boot the next kernel thread is created, called **init** or **process 1**. The **init** process has **PID=1** and shares all the needed data structures with **process 0**.

After being selected by the **scheduler**, it finishes kernel initialization and tries to load the executable file in the order: /sbin/init, /etc/init, /bin/init and /bin/sh. The **try_to_run_init_process()** function called by it is the wrapper for the system function **sys_execve()**.

A **regular process** (no longer a kernel thread) with its **own data structures** is created, and will run in **user mode**.

```
static int kernel_init(void * unused)
{
  …
   /* We try each of these until one succeeds.   */
                if (!try_to_run_init_process("/sbin/init") ||
                    !try_to_run_init_process("/etc/init") ||
                    !try_to_run_init_process("/bin/init") ||
                    !try_to_run_init_process("/bin/sh"))
                                return 0;
                panic("…");
}
```

Simplified version

Kernel threads

The **init** process works until the kernel stops, monitoring the activity of all processes.

Other kernel threads: *kswapd*, *kworker*, *ksoftirqd*.

They can be easily recognized on the list generated by **ps**, because their names are printed in square brackets.

The shown command was run on 'students'.

```
jmd@students:~$ ps fax | grep kswapd
   610 ?        S      6:29  \_ [kswapd0]
 522775 pts/1    S+     0:00  |          \_ grep kswapd
jmd@students:~$
```

```
jmd@students:~$ ps fax | more
    PID TTY      STAT   TIME COMMAND
      2 ?        S      0:02 [kthreadd]
      3 ?        I<     0:00  \_ [rcu_gp]
      4 ?        I<     0:00  \_ [rcu_par_gp]
      6 ?        I<     0:00  \_ [kworker/0:0H-events_highpri]
      9 ?        I<     0:00  \_ [mm_percpu_wq]
     10 ?        S      0:00  \_ [rcu_tasks_rude_]
     11 ?        S      0:00  \_ [rcu_tasks_trace]
     12 ?        S      0:10  \_ [ksoftirqd/0]
     13 ?        I     12:48  \_ [rcu_sched]
     14 ?        S      0:01  \_ [migration/0]
     15 ?        S      0:00  \_ [cpuhp/0]
     16 ?        S      0:00  \_ [cpuhp/1]
     17 ?        S      0:01  \_ [migration/1]
     18 ?        S      0:05  \_ [ksoftirqd/1]
     20 ?        I<     0:00  \_ [kworker/1:0H-kblockd]
     21 ?        S      0:00  \_ [cpuhp/2]
     22 ?        S      0:01  \_ [migration/2]
     23 ?        S      0:04  \_ [ksoftirqd/2]
     25 ?        I<     0:00  \_ [kworker/2:0H-events_highpri]
     26 ?        S      0:00  \_ [cpuhp/3]
     27 ?        S      0:01  \_ [migration/3]
     28 ?        S      0:04  \_ [ksoftirqd/3]
     30 ?        I<     0:00  \_ [kworker/3:0H-events_highpri]
     31 ?        S      0:00  \_ [cpuhp/4]
...
```

Kernel level synchronization

Because the kernel is **reentrant**, at any time there may be a couple of **active processes** in the **kernel**.

They all share the **same copy** of the **kernel data structures**.

Synchronization is therefore **necessary** when accessing these structures.

Quote of the week (posted June 12, 2019 by Jonathan Corbet)

> ***Implementing a correct synchronization primitive is like committing the perfect crime. There are at least 50 things that can go wrong, and if you are a highly experienced genius, you -might- be able to anticipate and handle 25 of them.***
>
> ***--- Paul McKenney***

The key task when writing kernel code is to recognize **WHICH** parts of the code are vulnerable to **race condition** and require protection.

One should also pay attention to the **granularity** of the locks in the context of system scalability.

Reading: Chapter 5: Concurrency and Race Conditions, Linux Device Drivers, Jonathan Corbet, Greg Kroah-Hartman, Alessandro Rubini, 2005.

Kernel level synchronization

How can **concurrent execution** of the same **kernel** code happen:

- As a result of **interrupt** – an interrupt may appear asynchronously, at any time (interrupt priorities, masking interrupts).
- As a result of **kernel preemption** (from version 2.6 the Linux kernel is **preemptable**). The scheduler can at any time preempt the process executed in **kernel mode** and start executing another one. Kernel preemption can occur:
  - When returning to kernel-space from an interrupt handler.
  - When kernel code becomes preemptible again.
  - If a task in the kernel explicitly calls **schedule()**.
  - If a task in the kernel blocks (which results in invocation of **schedule()**).
- In **multiprocessor systems** (SMP) as a result of executing the **same kernel code** on **different processors**.

Kernel level synchronization

1. **Atomic operations.** These are operations that can be performed using one assembler instruction in an indivisible manner, i.e. without the possibility of interruption during execution. On x86 processors:
   - atomic are the assembler instructions that read the memory contents,
   - assembler instructions such as **inc** or **dec** are atomic if no other processor takes over the memory bus after read, and before saving the argument value,
   - assembler instructions preceded by the byte **lock** are atomic even in a multiprocessor system,
   - assembler instructions preceded by the byte **rep** forcing the CPU to repeat the same instruction several times are not atomic: the CPU checks before each repetition of the iteration if there is an interrupt waiting.

   See: How are atomic operations implemented at a hardware level?

   (Short answer: extra transistors in the chip to implement **special cache** and **memory coherency** and **bus synchronization protocols**. To access **cache line** the other core has to obtain access rights first, and the protocol to obtain those rights involves the current owner.)

Kernel level synchronization

Atomic operations in C on arguments of type int use a special data types:.

```
typedef struct { int  counter; } atomic_t;
typedef struct  {s64 counter; } atomic64_t;
```

Types **atomic_t** and **atomic64_t** are effectively 32-bit and 64-bit numbers, respectively, that can only be operated on using the various **atomic_*()** interfaces. Examples:

```
atomic_read(v)
atomic_set(v, i)
atomic_add(i, v)
atomic_inc(v)
atomic_dec_and_test(v)
```

Examples of the **atomic** operations on a **bitmap**:

```
set_bit(nr, addr)
clear_bit(nr, addr)
test_and_set_bit(nr, addr)
test_and_clear_bit(nr, addr)
```

In some systems, **atomic** operations may be **much slower** than their **non-atomic** counterparts.

Kernel level synchronization

2. **Disabling interrupts.** The kernel can not perform blocking operations with interrupts disabled, because it may cause the system to crash.

   Macros enabling and disabling interrupts in a **uniprocessor** system:

   ```
   spin_lock_irq(lock)
   spin_unlock_irq(lock)
   spin_lock_irqsave(lock, flags)
   spin_unlock_irqrestore(lock, flags)
   ```

   Do not disable interrupts for a long time, because during this time any communication between the processor and controllers of input-output devices is blocked.

3. **Locking**. Linux offers a number of lock types. They could be roughly divided, until recently, into two categories: spinning and sleeping locks:
   - **spinlocks** (used in **multiprocessor** systems),
   - **mutexes** and **system semaphores** (used in **uniprocessor** and **multiprocessor** systems).

Kernel level synchronization

4. **RCU synchronization mechanism** (**Read-Copy Update**). Readings can be made in parallel with writings.
5. **Barriers** that prevent code optimization (its reorganization) by the compiler and processor. Described in detail in Documentation/memory-barriers.txt.
6. **Big kernel lock**, i.e. **BKL** – a mechanism that allows to block the entire kernel. It guarantees that at most one processor is running in kernel mode at a time.

   Functions to support this mechanism are **lock_kernel** and **unlock_kernel**.
   It affects performance very badly.
   It was finally **eliminated** in **2011** (version 2.6.39 kernel) by Arnd Bergmann.

When selecting the synchronization mechanism for a specific problem, one must remember about the **correctness** of the solution and the **efficiency** of the solution.

Spinlocks

The name comes from the fact that waiting for a lock to be released is **active**.

This may seem ineffective, but in reality it can be much **cheaper** than putting the thread to sleep, switching the context, waking it up later when the condition is met.

Spinlocks **depend on architecture** and **are implemented in assembler**. They are used in **multiprocessor systems**.

They should only be used if the expected time to acquire the resource is **short**.

There are **two types of spinlocks**:

- **Regular** – of type **spinlock_t**:

  They ensure that a piece of code surrounded by them will be executed at the same time on only one processor.

- For **readers-writers** – of type **rw_lock_t**:

  They allow to create a critical section of type **read-write**.

Spinlocks

The way of using spinlocks:

```
#define DEFINE_SPINLOCK(x) spinlock_t x = __SPIN_LOCK_UNLOCKED(x)
static DEFINE_SPINLOCK(lock);
spin_lock(&lock);
    /* critical section */
spin_unlock(&lock);
```

Macros that support **spinlocks** in a multiprocessor system (some of them):

spin_lock_init(lock)	Initializes the object of type spinlock_t
spin_lock(lock)	Acquires the spinlock
spin_unlock(lock)	Releases the spinlock
spin_lock_irq(lock)	Disables interrupts on the local CPU and acquires the spinlock
spin_unlock_irq(lock)	Enables interrupts on the local CPU and releases the spinlock
spin_lock_irqsave(lock, flags)	Stores the previous interrupt state, disables interrupts on the local CPU and acquires the spinlock
spin_unlock_irqrestore(lock, flags)	Restores the previous interrupt state, enables interrupts on the local CPU and releases the spinlock
spin_trylock(lock)	Tries to obtain a lock, but will not block if it cannot be immediately acquired

Spinlocks

Macros that support **read-write spinlocks** in a multiprocessor system (some of them) :

read_lock_irq(lock)	Disables interrupts on the local CPU and acquires the spinlock for reading
read_unlock_irq(lock)	Enables interrupts on the local CPU and releases the spinlock of type read
write_lock_irq(lock)	Disables interrupts on the local CPU and acquires the spinlock for writing
write_unlock_irq(lock)	Enables interrupts on the local CPU and releases the spinlock of type write

The read-write **spinlocks favor readers over writers** so can **starve pending writers**.

In **uniprocessor systems**:

- When the kernel is compiled with the kernel preemption option **disabled**, spinlocks are defined as **empty** operations.
- When kernel preemption is **enabled**, **spin_lock** is equivalent to **preempt_disable**, and **spin_unlock** is equivalent to **preempt_enable** (kernel preemption is disabled inside the critical section protected by a spinlock). (See also Kernel preemption)

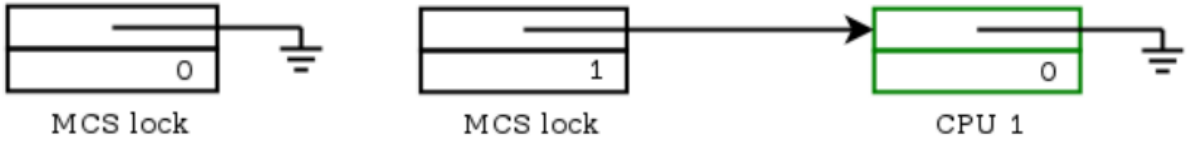
Recommended way of using spinlocks: Documentation/locking/spinlocks.rst.

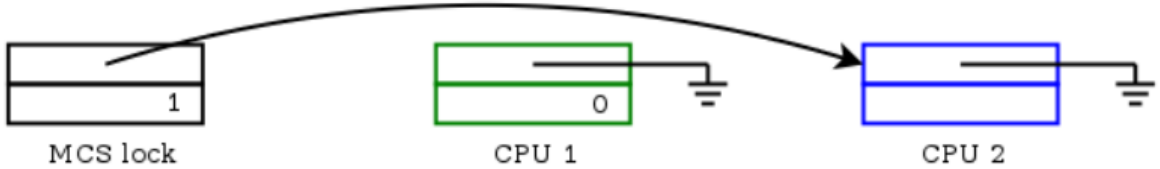
Spinlocks – implementation

– In the 2.6.24 kernel, a spinlock was represented by an **integer value**. A **value of one** indicated that the lock is available, the more **negative the value** of the lock gets, the more processors are trying to acquire it.

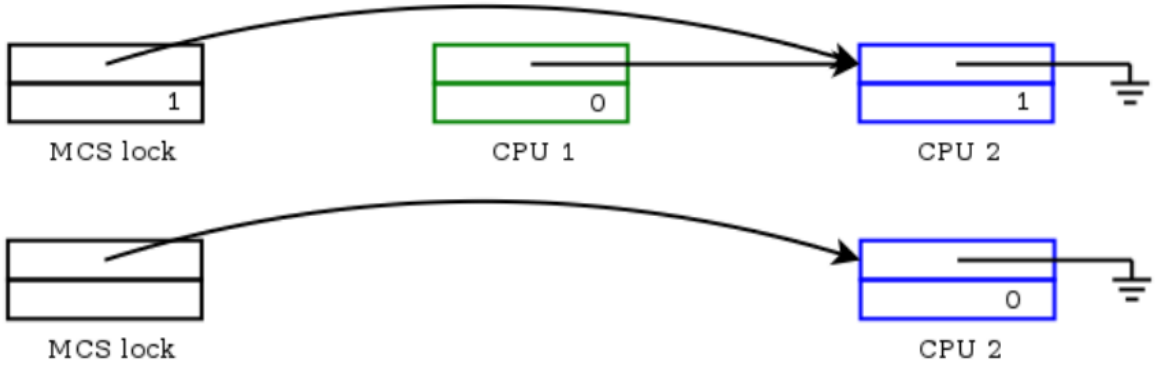
– Ticket spinlocks (2008) added fairness to the mechanism by using 16-bit quantity, split into two bytes. You can think of the "next" field as being the number on the next ticket in the dispenser, while "owner" is the number appearing in the "now serving" display over the counter.

– MCS locks (Mellor-Crummey & Scott, 2014) expand a spinlock into a per-CPU structure, eliminating much of the cache-line bouncing.

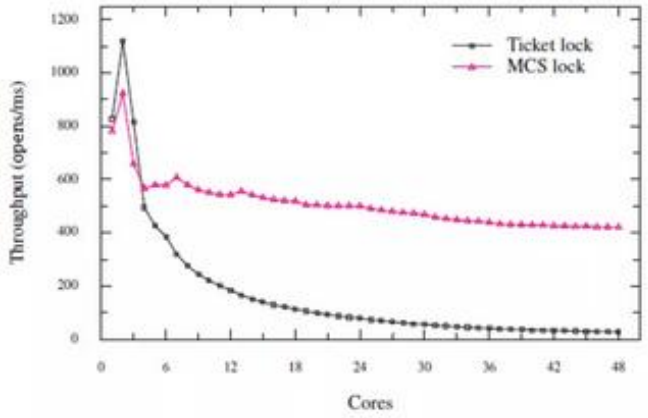




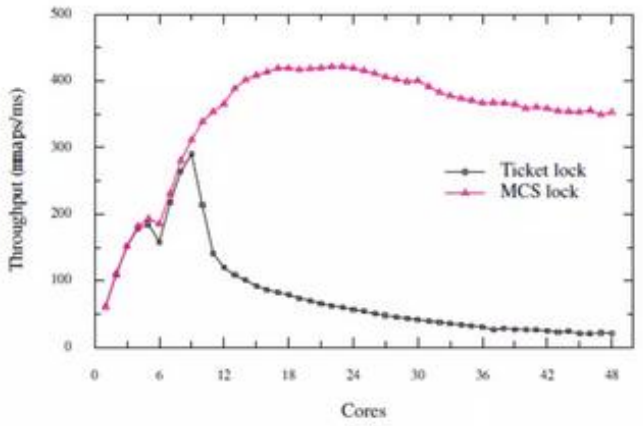

```
struct mcs_spinlock {
    struct mcs_spinlock *next;
    int locked; /* 1 if lock acquired */
};
```



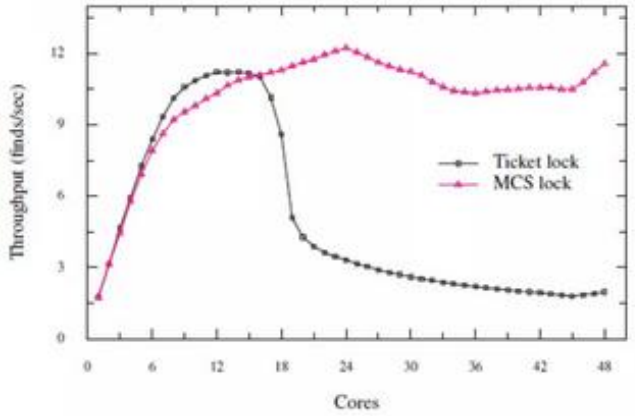
Ticket spinlock vs MCS lock



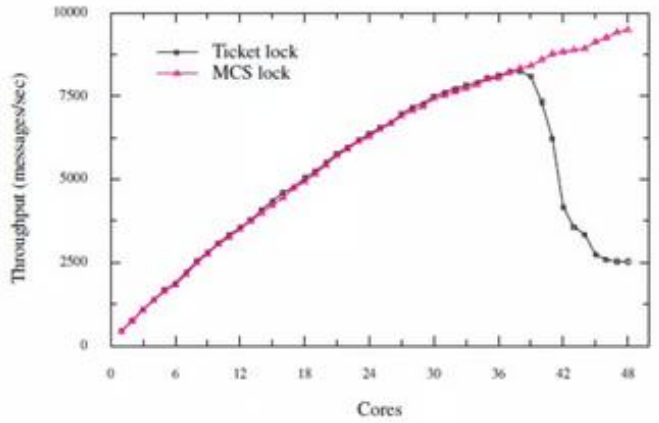
(a) Performance for FOPS.



(b) Performance for MEMPOP.



(c) Performance for PFIND.



(d) Performance for EXIM.

Figure 12: Performance of benchmarks using ticket locks and MCS locks.

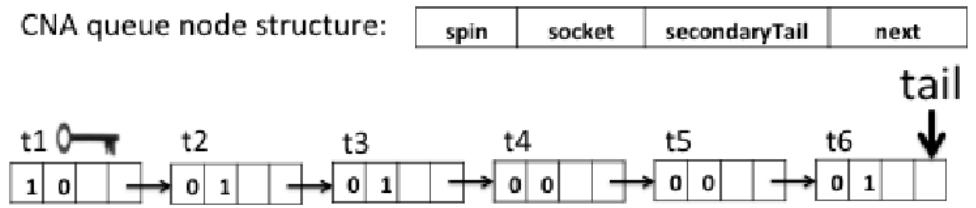
Benchmarks

- FOPS: creates a single file and starts one process on each core. Repeated system calls "open() and close()".
  - ✓ Executing the critical section increases from 450 cycles on two cores to 852 cycles on four cores
  - ✓ The critical section is executed multiple times per-operation and modifies shared data, which incurs costly cache misses
- MEMPOP
  - ✓ One process per core
  - ✓ Each process mmaps 64 kB of memory with the MAP_POPULATE flag, then munmaps the memory
- PFIND: searches for a file by executing several instances of the GNU find utility
- EXIM (mail server): A single master process listens for incoming SMTP connections via TCP and forks a new process for each connection

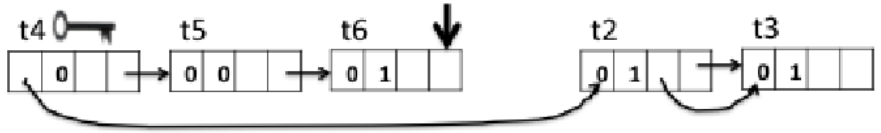
Reference Paper: Boyd-Wickizer, Silas, et al. "Non-scalable locks are dangerous." Proceedings of the Linux Symposium. 2012.

Compact NUMA-aware locks

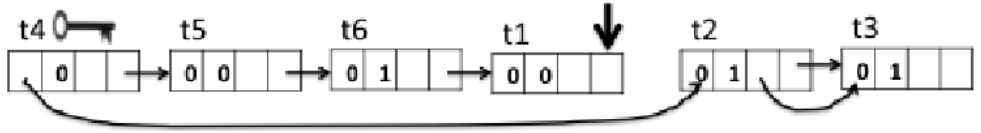
- NUMA-aware qspinlocks, Jonathan Corbet, April 2021.
- Compact NUMA-aware Locks, Dave Dice, Alex Kogan, 2019.



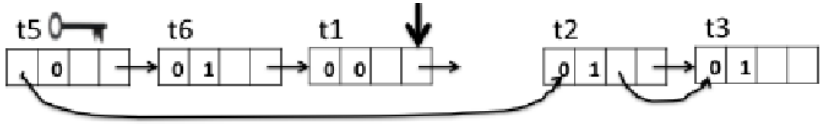
(a) The main queue consists of six threads, with $t1$, $t4$ and $t5$ running on socket 0 and the rest running on socket 1. Thread $t1$ has the lock. The secondary queue is empty.



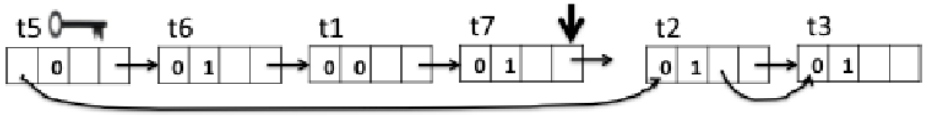
(b) $t1$ exits its critical section, traverses the main queue and finds that $t4$ runs on the same socket. Therefore, $t1$ moves $t2$ and $t3$ into the secondary queue, setting the `secondaryTail` field in $t2$'s node to $t3$. Then, $t1$ passes the lock to $t4$ by writing the pointer to the head of the secondary queue into $t4$'s `spin` field.



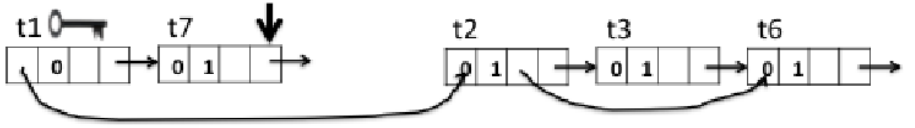
(c) $t1$ returns and enters the main queue.



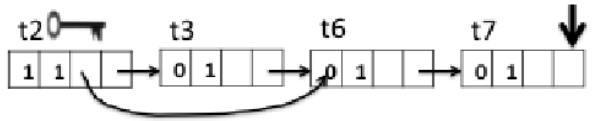
(d) When $t4$ exits its critical section, it discovers that the next thread in the main queue ($t5$) runs on the same socket. $t4$ passes the lock to $t5$ by simply copying the value in $t4$'s `spin` field into $t5$'s `spin` field.



(e) $t7$ running on socket 1 arrives and enters the main queue.



(f) $t5$ exits its critical section, moves $t6$ into the end of the secondary queue (and updates the `secondaryTail` field in $t2$'s node), and passes the lock to $t1$.



(g) $t1$ exits its critical section and finds no threads on socket 0 in the main queue. Thus, $t1$ moves nodes from the secondary queue back to the main one, putting them before its successor $t7$, and passes the lock to $t2$.

A running example for CNA lock handovers on a 2-socket machine.
Empty cells represent NULL pointers

Mutexes and system semaphores

Additional reading:

- Documentation/locking/mutex-design.rst.
- Documentation/locking/rt-mutex-design.rst.
- Documentation/percpu-rw-semaphore.txt.

- Generic Mutex Subsystem, Ingo Molnar , December 2005.
- An Overview of Kernel Lock Improvements, Davidlohr Bueso, Scott Norton, August 2014.
- Reimplementing mutexes with a coupled lock, Jonathan Corbet, September 2016.

Oscar Wilde once famously observed that fashion "is usually a form of ugliness so intolerable that we have to alter it every six months."
Perhaps the same holds true of locking primitives in the kernel.

Mutexes and system semaphores

Mutexes (i.e. **binary semaphores**) are a synchronization mechanism often used in both **uniprocessor** and **multiprocessor** systems.

Mutexes use atomic operations, only one thread can be in the possession of a mutex and only it can release it.

Mutexes are objects of type **struct mutex**:

```
struct mutex {
      atomic_long_t    owner;
      raw_spinlock_t   wait_lock;
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
       struct  optimistic_spin_queue    osq; /* Spinner MCS lock */
#endif
       struct  list_head   wait_list;
       ...
};
```

Simplified version

Drawback:
Among the largest locks in the kernel, which means more CPU cache and memory footprint.

The value of the field **owner** is 64 bits wide, large enough to hold a pointer value.

If the mutex is **available**, there is **no owner**, so the **owner** field contains **zero**.

When the mutex is **taken**, the acquiring thread's **task_struct pointer** is placed there, simultaneously indicating that the mutex is **unavailable** and which thread **owns** it.

Mutexes and system semaphores

In its most basic form it also includes a **wait-queue** and a **spinlock** that serializes access to it. When acquiring a mutex, there are three possible paths that can be taken, depending on the **state of the lock**:

- **fastpath**: tries to atomically acquire the lock by **cmpxchg**()ing the owner with the current task. This only works in the **uncontended case** (cmpxchg() checks against 0UL, so all 3 state bits have to be 0). If the lock is contended it goes to the next possible path.
- **midpath**: aka **optimistic spinning**, tries to spin for acquisition while the lock owner is running and there are no other tasks ready to run that have higher priority (**need_resched**). The rationale is that if the lock owner is running, it is likely to release the lock soon. The mutex spinners are queued up using MCS lock so that only one spinner can compete for the mutex.
- **slowpath**: last resort, if the lock is still unable to be acquired, the task is added to the **wait-queue** and sleeps until woken up by the unlock path. Under normal circumstances it blocks as **TASK_UNINTERRUPTIBLE**.

CMPXCHG — Compare and Exchange

Mutexes and system semaphores

While formally kernel mutexes are **sleepable locks**, it is midpath that makes them more practically a **hybrid type**.

By simply not interrupting a task and busy-waiting for a few cycles instead of immediately sleeping, the performance of this lock has been seen to significantly improve a number of workloads.

This technique is also used for **rw-semaphores**.

Mutex API

```
/* Statically define the mutex */
   DEFINE_MUTEX(name);
/* Dynamically initialize the mutex */
   mutex_init(mutex);
/* Acquire the mutex, uninterruptible */
   void mutex_lock(struct mutex *lock);
   int  mutex_trylock(struct mutex *lock);
/* Acquire the mutex, interruptible */
   int mutex_lock_interruptible(struct mutex *lock);
/* Acquire the mutex, interruptible, if dec to 0 */
   int atomic_dec_and_mutex_lock(atomic_t *cnt, struct mutex *lock);
/* Unlock the mutex */
   void mutex_unlock(struct mutex *lock);
/* Test if the mutex is taken */
   int mutex_is_locked(struct mutex *lock);
```

Mutexes and system semaphores

In addition to mutexes there are also **general semaphores**, which can take any value *n*.

Unlike mutexes, binary semaphores do not have an owner, so **up()** can be called in a different thread from the one which called **down()**. It is also safe to call **down_trylock()** and **up()** from interrupt context.

There are also **read-write semaphores**. They are implemented as **struct rw_semaphore**. These are operations: **down_read**(), **up_read**(), **down_write**(), **up_write**() (only versions *uninterruptible*).

```
/*
 * The ->count variable represents how many more tasks can acquire this
 * semaphore. If it's zero, there may be tasks waiting on the wait_list.
*/

struct semaphore {
    raw_spinlock_t   lock;
    unsigned int     count;
    struct list_head wait_list;
};

static inline void sema_init(struct semaphore *sem, int val)

#define init_MUTEX(sem)             sema_init(sem, 1)
#define init_MUTEX_LOCKED(sem)  sema_init(sem, 0)
```

Mutexes and system semaphores

```
void up(struct semaphore *sem)                          Simplified version
{
    unsigned long flags;
    raw_spin_lock_irqsave(&sem->lock, flags);
    if (list_empty(&sem->wait_list))
        sem->count++;
    else
        __up(sem);
    raw_spin_unlock_irqrestore(&sem->lock, flags);
}
```

The basic semaphore operations are **up** and **down**.

Unlike mutexes, up() may be called from any context and even by tasks which have never called down().

Acquires the semaphore. If no more tasks are allowed to acquire the semaphore, calling this function will put the task to sleep until the semaphore is released.
Use of this function is deprecated, use down_interruptible() or down_killable() instead.

```
void down(struct semaphore *sem)                        Simplified version
{
    unsigned long  flags;
    raw_spin_lock_irqsave(&sem->lock, flags);
    if (sem->count > 0)
        sem->count--;
    else
        __down(sem);
    raw_spin_unlock_irqrestore(&sem->lock, flags);
}
```

Mutexes and system semaphores

```
Int down_interruptible(struct semaphore *sem)
{                                                    Simplified version
      unsigned long flags;
      int result = 0;
      raw_spin_lock_irqsave(&sem->lock, flags);
      if (sem->count > 0)
            sem->count--;
      else
            result = __down_interruptible(sem);
       raw_spin_unlock_irqrestore(&sem->lock, flags);
       return result;
}
```

Operations from the **__down**() family eventually call **schedule()**, which selects the new process to be executed.

This synchronization mechanism does not involve busy waiting, because the process is **suspended** and the **processor** is **passed** to another process.

```
int down_trylock(struct semaphore *sem)
{
      unsigned long flags;
      int count;
      raw_spin_lock_irqsave(&sem->lock, flags);
      count = sem->count - 1;
      if (count >= 0)
            sem->count = count;
      raw_spin_unlock_irqrestore(&sem->lock, flags);
      return (count < 0);
}                                                    Simplified version
```

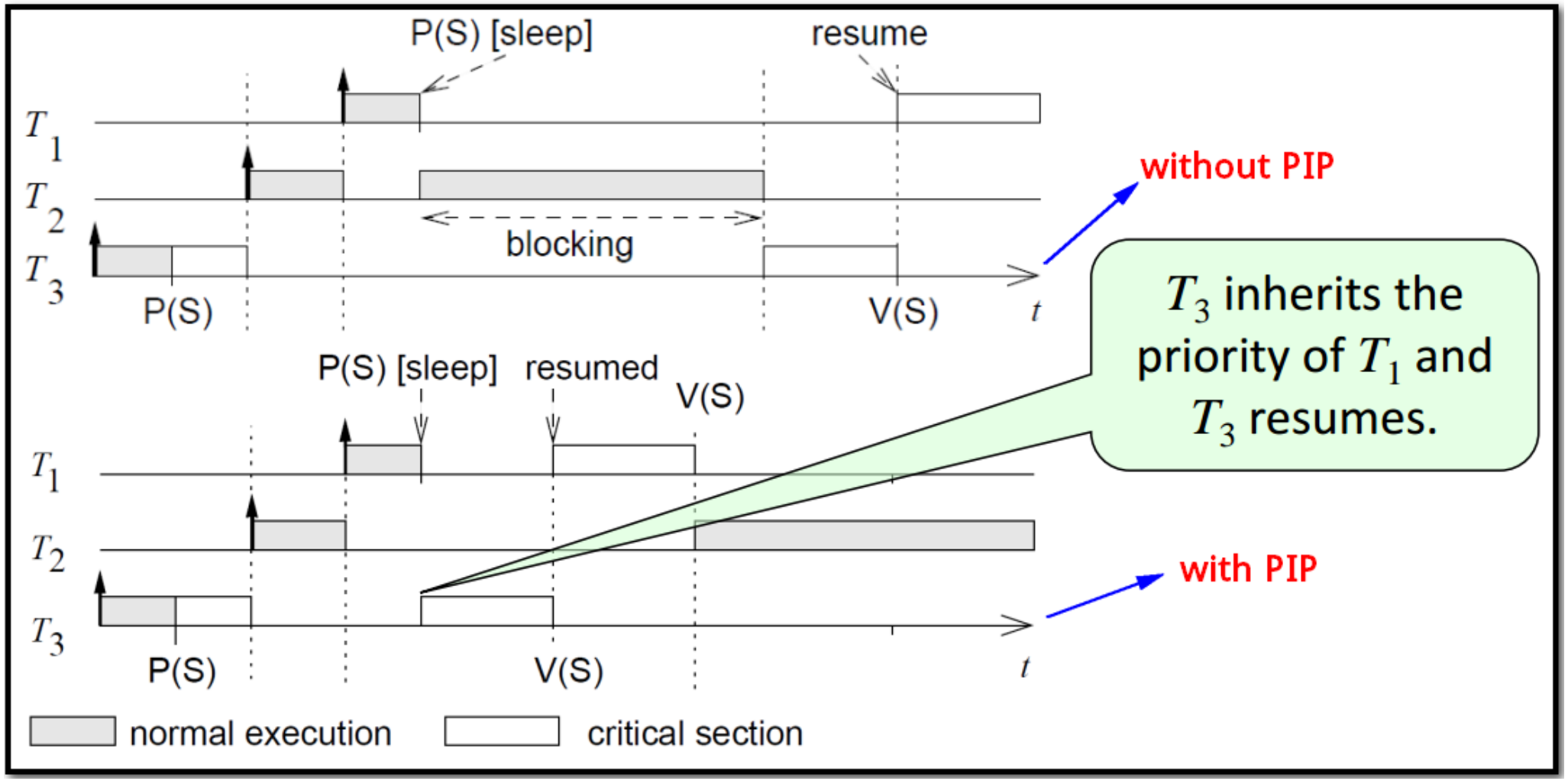
Real time mutexes

There are also **real time mutexes**, which implement **priority inheritence**, and solve the problem of **priority inversion**, which affects **real-time systems**.

Priority inversion is when a **lower priority** process executes while a **higher priority** process wants to run. The example of unbounded priority inversion is where you have three processes, A, B, and C, where A is the **highest** priority process, C is the **lowest**, and B is in **between**. A tries to grab a lock that C owns and must wait and lets C run to release the lock. In the meantime, B executes, and since B is of a higher priority than C, it preempts C, but by doing so, it is in fact preempting A which is a higher priority process.

The problem is solved by **priority inheritance** – process inherits the priority of another process if the other process blocks on a lock owned by the current process. Let's use the previous example. This time, when A blocks on the lock owned by C, C would inherit the priority of A. So now if B becomes runnable, it would not preempt C, since C now has the high priority of A. As soon as C releases the lock, it loses its inherited priority, and A then can continue with the resource that C had.

Priority inversion and priority inheritance



[Solutions for Priority Inversion in Real-time Scheduling]

Real time mutexes and local locks

Real time mutexes are implemented as a structure **rt_mutex**.

```
struct rt_mutex {
        raw_spinlock_t          wait_lock;
        struct rb_root_cached   waiters;   /*rbtree root to enqueue waiters in priority order; */
        struct task_struct      *owner;
        ...
};
```

Simplified version

There are functions available:

rt_mutex_init(), **rt_mutex_lock**(), **rt_mutex_unlock**(), **rt_mutex_trylock**().

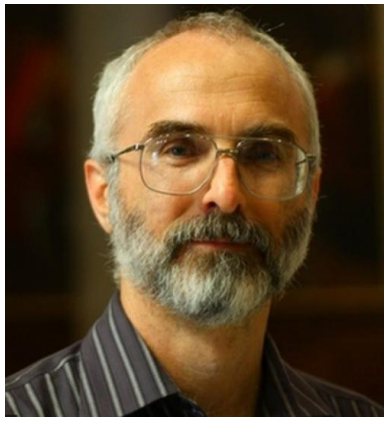
Local locks in the kernel (from v5.8), Marta Rybczyńska, August 2020. **SKIP**

On non-realtime systems, the acquisition of a local lock simply maps to **disabling preemption** (and possibly **interrupts**).

On real time systems, instead, local locks are actually **sleeping spinlocks**; they **do not disable** either **preemption** or **interrupts**. They are sufficient to serialize access to the resource being protected without increasing latencies in the system as a whole.

Interface**: local_lock()**, **local_unlock(), local_lock_irq()**, **local_unlock_irq()** etc.

Read-Copy Update (RCU)



The mechanism was added to the Linux kernel in October 2002.

It is described in detail in Documentation/RCU.

Interesting works on the RCU can be found on the website http://www.rdrop.com/users/paulmck/RCU/ maintained by **Paul McKenney**, who devoted his Ph.D. to this topic (now employed in Meta).

What is RCU?, P. McKenney, November 2018 (good presentation!)

RCU Usage In the Linux Kernel: One Decade Later, P. McKenney, S. Boyd-Wickizer, J. Walpole, 2019.

A series of articles on lwn.net:

- The RCU API, 2019 edition, Paul McKenney, January 2019.
- Requirements for RCU part 1: the fundamentals, Paul McKenney, July 2015.
- RCU requirements part 2 – parallelism and software engineering, Paul McKenney, August 2015.
- RCU requirements part 3, Paul McKenney, August 2015.
- What is RCU, Fundamentally?, Paul McKenney, December 2007.

Wikipedia (compact description with pictures).

Read-Copy Update (RCU)

Information about all **users** of the pointer to the **shared** data structure is **stored**. When the structure is to **change**, a **copy** is created and the change is made on it.

When all **readers** finish reading the old copy, the pointer **changes** to start pointing to the new one. **Readings** can be made in parallel with **writings**. It saves **time** at the expense of slightly higher **memory consumption**.

Suppose that the pointer **ptr** is to be protected by the RCU. You must first call **rcu_dereference()** for it and continue to work on the obtained result.

In addition, the code being executed must be protected by **rcu_read_lock()** and **rcu_read_unlock()**.

*The value returned by **rcu_dereference** is valid only within the enclosing RCU read-side critical section.*

*As with **rcu_assign_pointer**, an important function of rcu_dereference is to document which pointers are protected by RCU.*

```
rcu_read_lock();
p = rcu_dereference(ptr);  // subscribe
if (p != NULL) {
    some_function(p);
}
rcu_read_unlock();
```

Read-Copy Update (RCU)

Modification of the pointer must be done using **rcu_assign_pointer()**. Subsequent **read** operations will see a **new** structure instead of the old one.

```
struct something *new_ptr = kmalloc(...);

new_ptr->field1 = xyz;
new_ptr->field2 = 12;
new_ptr->field3 = 13;

rcu_assign_pointer(ptr, new_ptr);  // publish
```

*RCU **protects readers from writers**, but does not protect **writers from writers**. This must be provided by using other mechanisms, e.g. spinlocks.*

The **rcu_assign_pointer** and **rcu_dereference** primitives contain the architecture-specific **memory barrier** instructions and **compiler directives** necessary to ensure that the **data** is **initialized** before the **new pointer** becomes **visible**, and that any **dereferencing** of the **new pointer** occurs **after** the **data** is **initialized**.

The **old** structure is available to **readers** until the last one finishes reading. Only then can the kernel **removes** it.

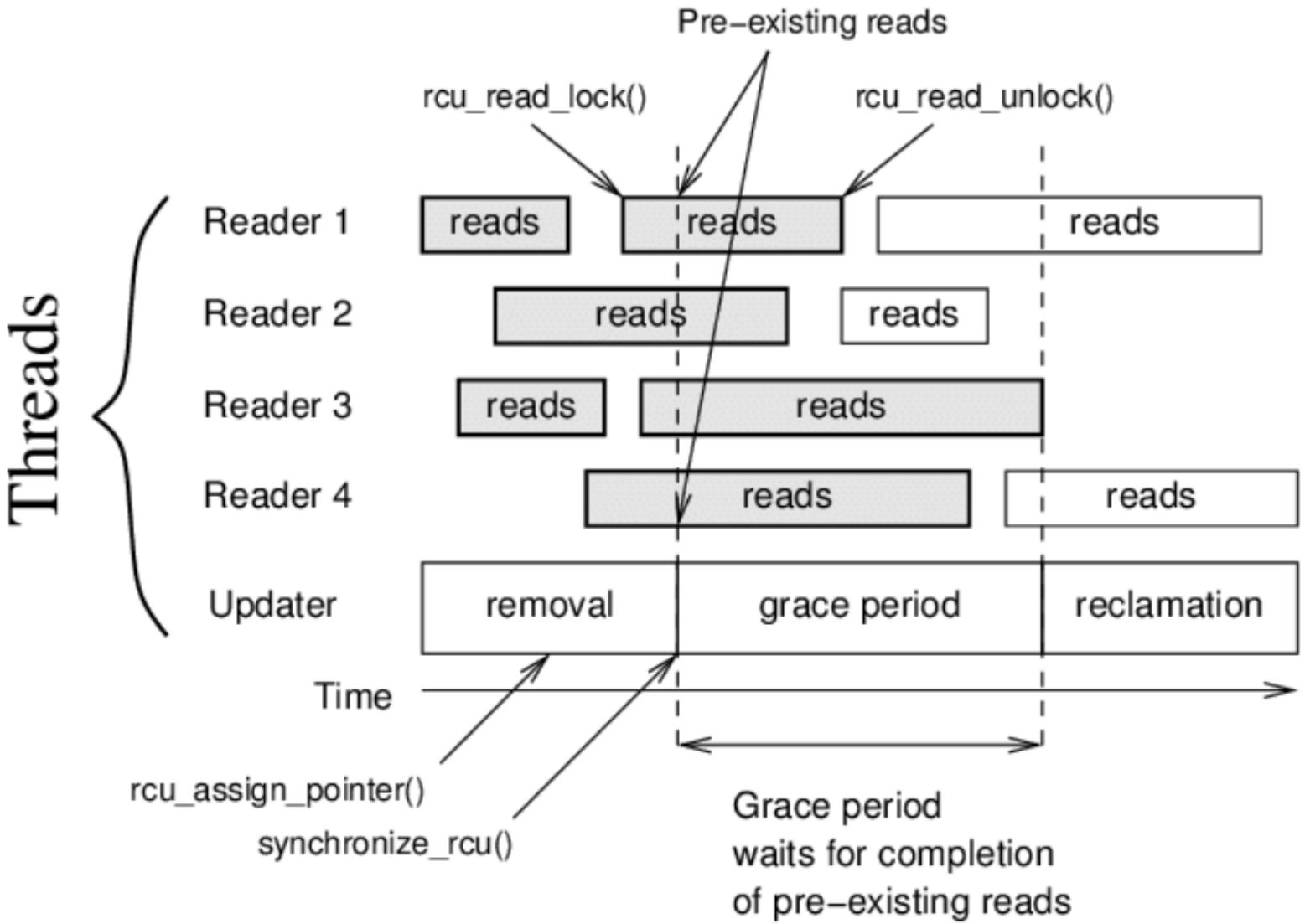
The function **synchronize_rcu()** blocks until all **readers finish** reading. Instead of blocking, synchronize_rcu may register a callback (called **call_rcu()**) to be invoked after all ongoing RCU read-side critical sections have completed.

It is worth using the RCU when there is **much more reading than writing**.

Read-Copy Update (RCU)

Grace period – time period when every thread was in at least one quiescent state.

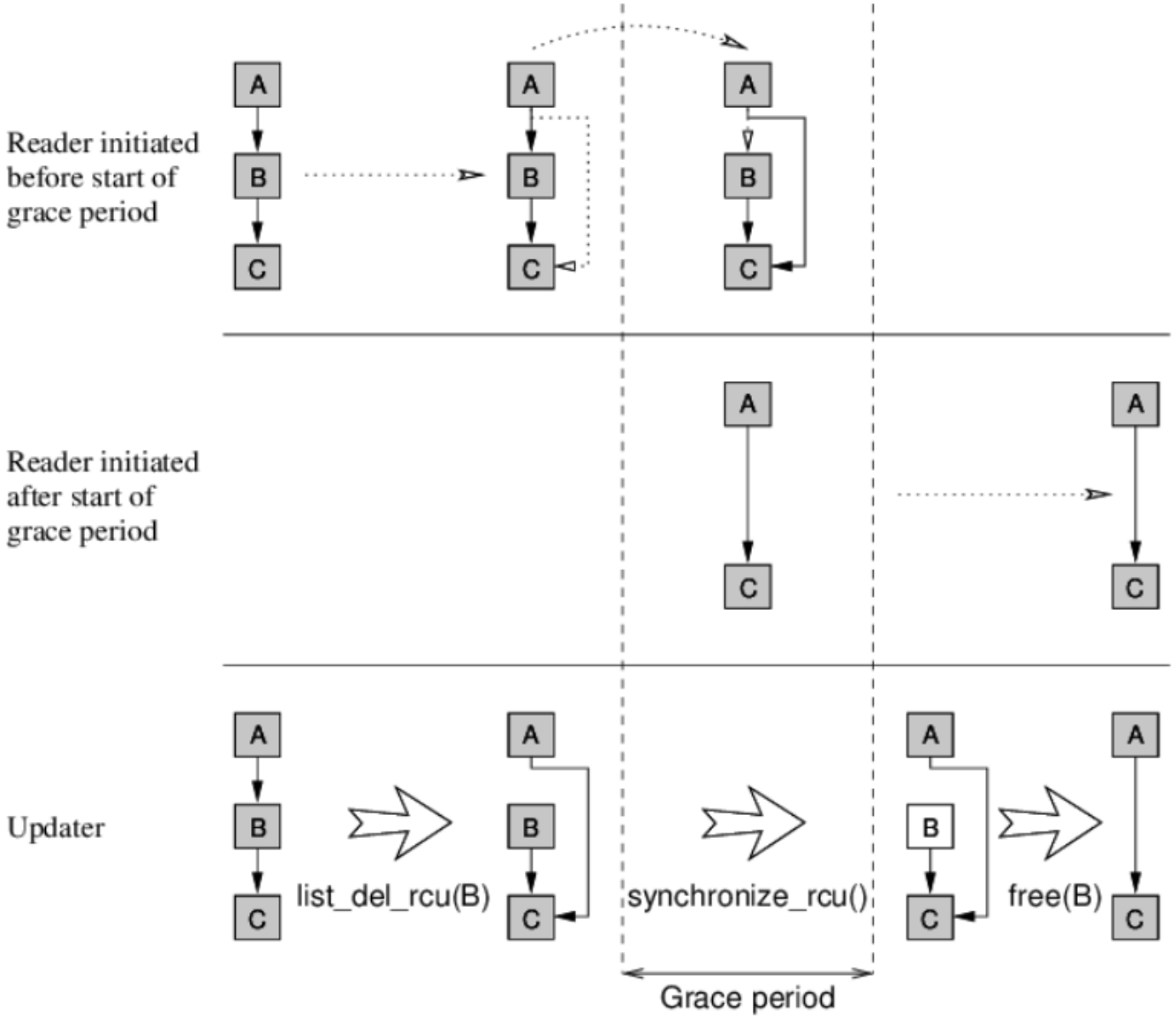
Quiescent state – any point in the thread execution where the thread does not hold a reference to shared memory.



Schematic of RCU Grace Period and Read-Side Critical Sections

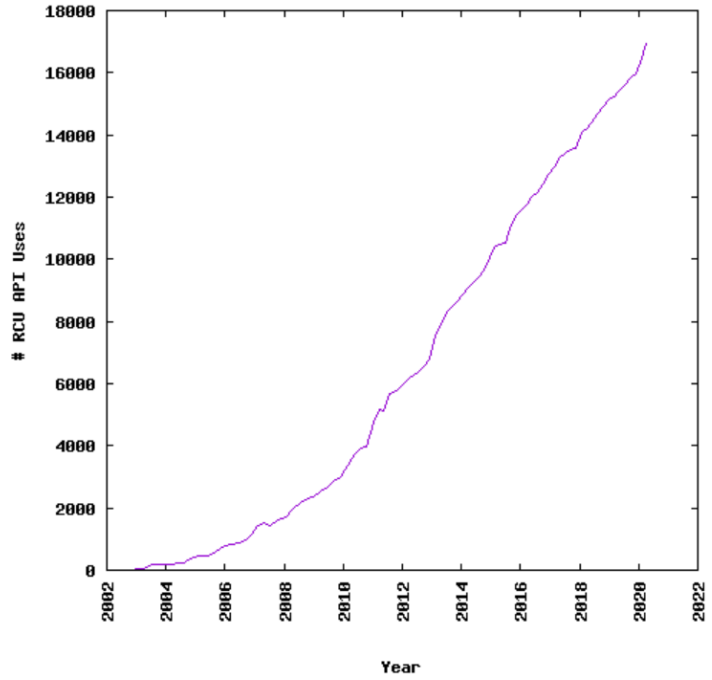
Paul McKenney

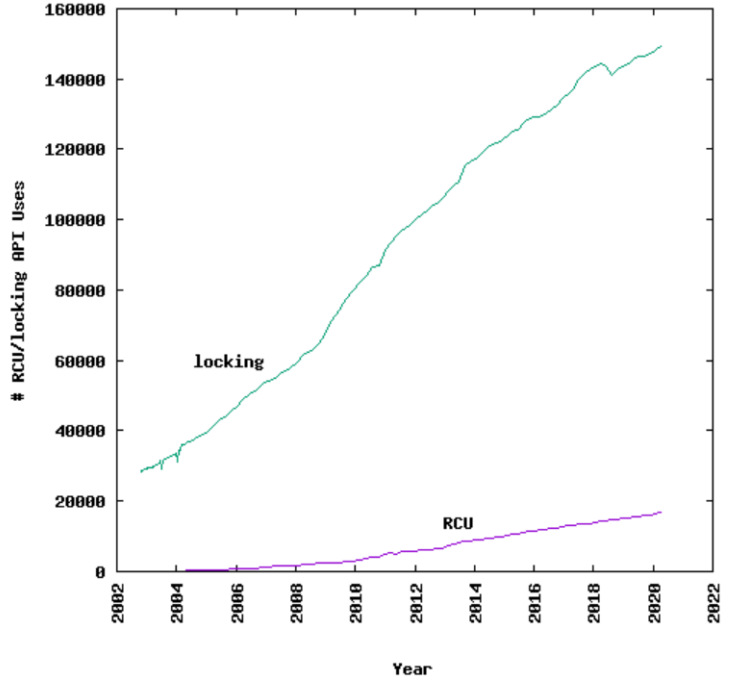
Read-Copy Update (RCU)

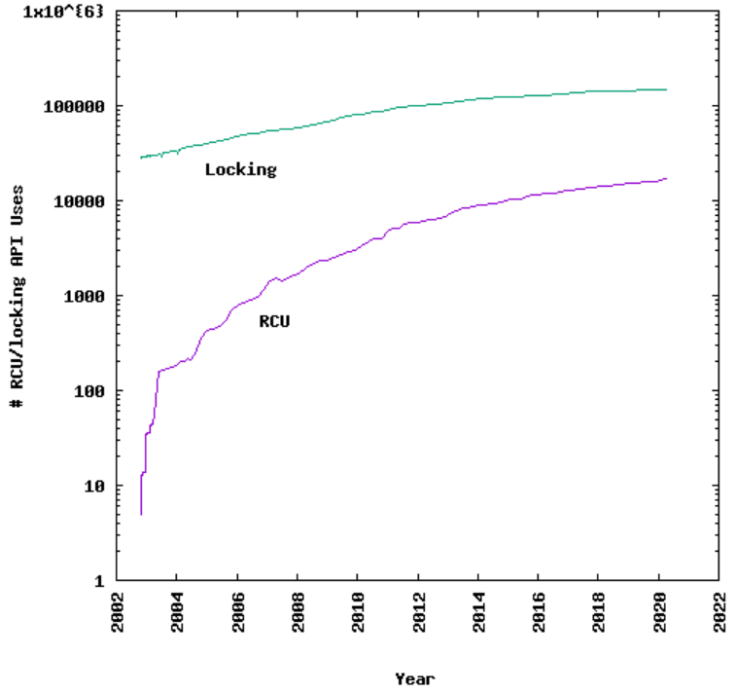


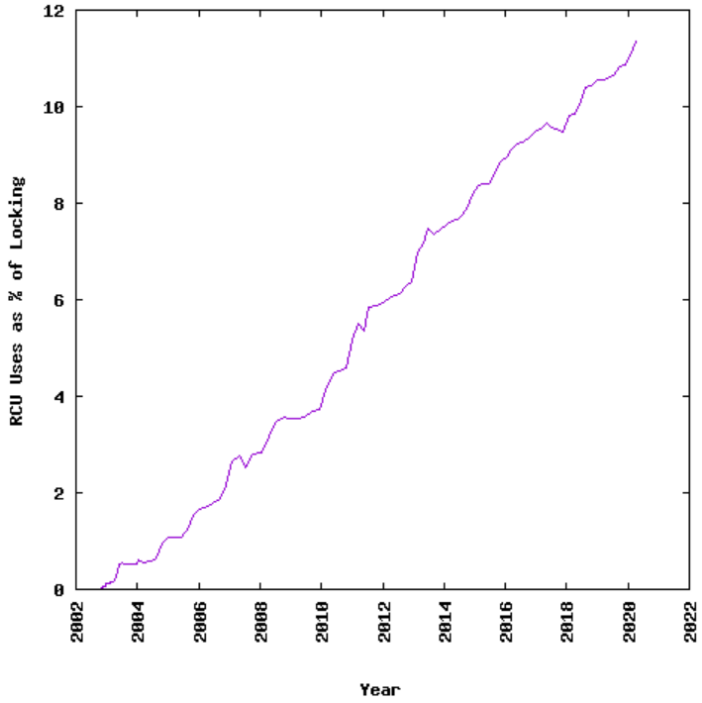
Paul McKenney

http://www.rdrop.com/users/paulmck/RCU/linuxusage.html









Read-Copy Update (RCU)

RCU is a very specialized primitive, and it is exceedingly important to **use the right tool for the job**.

For a great many jobs, normal **locking** remains the **best tool**.

Almost all RCU uses in the Linux kernel use locking to **protect updates**, which does place a hard **upper limit** on **RCU's fraction of synchronization primitives**.

Process states

The **task_struct process descriptor** has a volatile long **state** field that specifies the current state of the process. Possible states are defined as constants:

TASK_RUNNING – process is in execution or ready for execution.

T**ASK_INTERRUPTIBLE** – process is or may be sleeping in an interruptible state, i.e. it will resume execution after the signal arrives or after the wake time has elapsed. The last means that the process can be put to sleep for a certain time (the possibility often used by the device handlers).

TASK_UNINTERRUPTIBLE – process is or will be put to sleep in an uninterrupted state (e.g. waiting for an inode). This process can only be resumed by calling the **wake_up()** function – its status will be changed to TASK_RUNNING.

TASK_STOPPED – process has been stopped, it is neither finished nor ready to be executed. The process will go into this state, e.g. due to receiving a SIGSTOP signal. It can only be resumed by receiving the SIGCONT signal.

EXIT_ZOMBIE – process invoked the **exit()** function, but its parent process has not yet performed **wait()** for it (to retrieve the execution code). The process will remain in the system until **wait()** is done by the parent process.

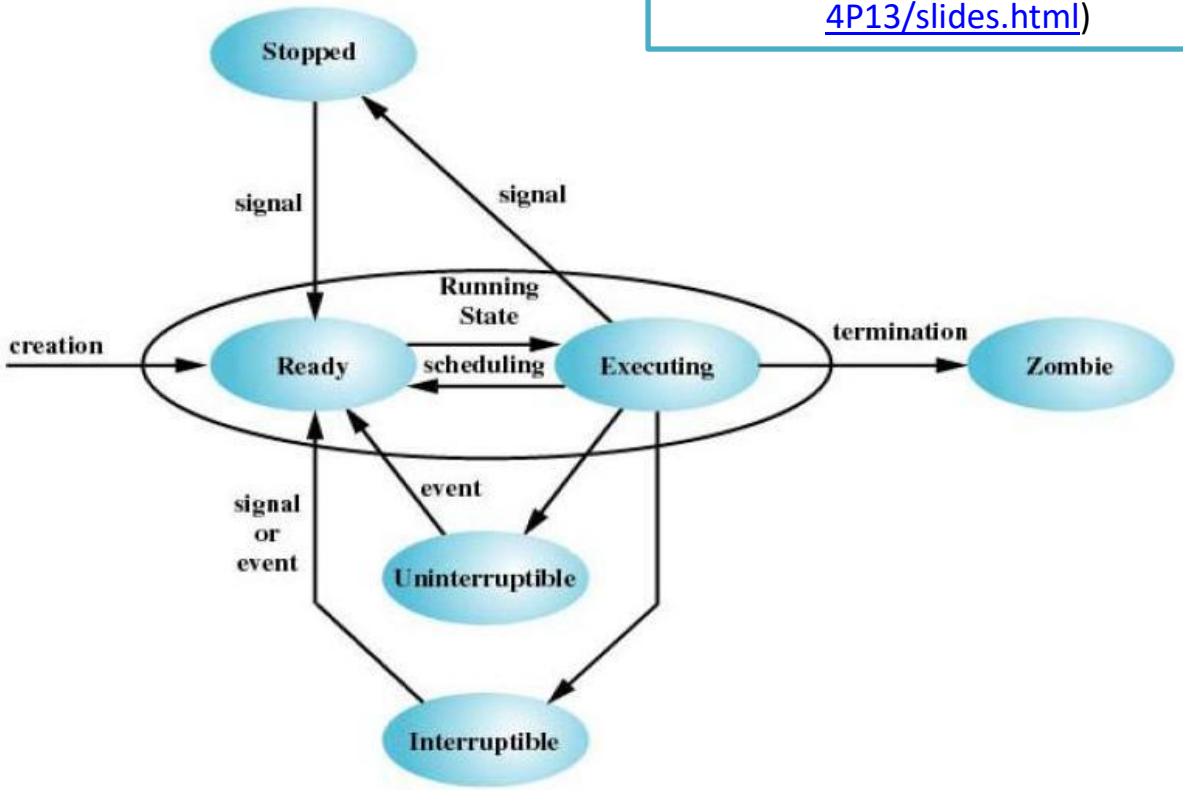
```
/*
 * We have two separate sets of flags:
 * task->state is about runnability,
 * while task->exit_state are about the task exiting.
 * Confusing, but this way modifying one set can't modify
 * the other one by mistake.
 */
  /* in tsk->state */
  #define TASK_RUNNING            0
  #define TASK_INTERRUPTIBLE      1
  #define TASK_UNINTERRUPTIBLE    2
  #define __TASK_STOPPED          4
  #define __TASK_TRACED           8

  /* in tsk->exit_state */
  #define EXIT_ZOMBIE            16
  #define EXIT_DEAD              32

  /* in tsk->state again */
  #define TASK_DEAD              64
  #define TASK_WAKEKILL          12          Many more
```

Wait queues

Process states in Linux (source: http://www.cosc.brocku.ca/Offerings/4P13/slides.html)



All processes in the state **TASK_RUNNING** are in the queue of processes ready for execution.

Processes in the state **TASK_STOPPED**, **EXIT_ZOMBIE** and **EXIT_DEAD** do not have to be in any queue, because they are only referenced via **PID** or through the process **family connections**.

Pending processes in the state **TASK_INTERRUPTIBLE** and **TASK_UNINTERRUPTIBLE** stand in different queues, depending on the event.

These are so-called **wait queues**. The process that is to begin to wait for the event is set in the associated queue and gives control. He will be awakened by the kernel when the event occurs.

Reading:

- Simple wait queues, Jonathan Corbet, December 2013.
- The return of simple wait queues, Jonathan Corbet, October 2015.

Wait queues

The element of the **wait queue** is of the type **wait_queue_entry.**

The **private** field indicates the descriptor of the waiting process. Field **entry** links all processes waiting for the same event. The **func** function is called to wake up the process.

The flag **WQ_FLAG_EXCLUSIVE** causes that the process will be awakened by itself, not all processes from the queue at once.

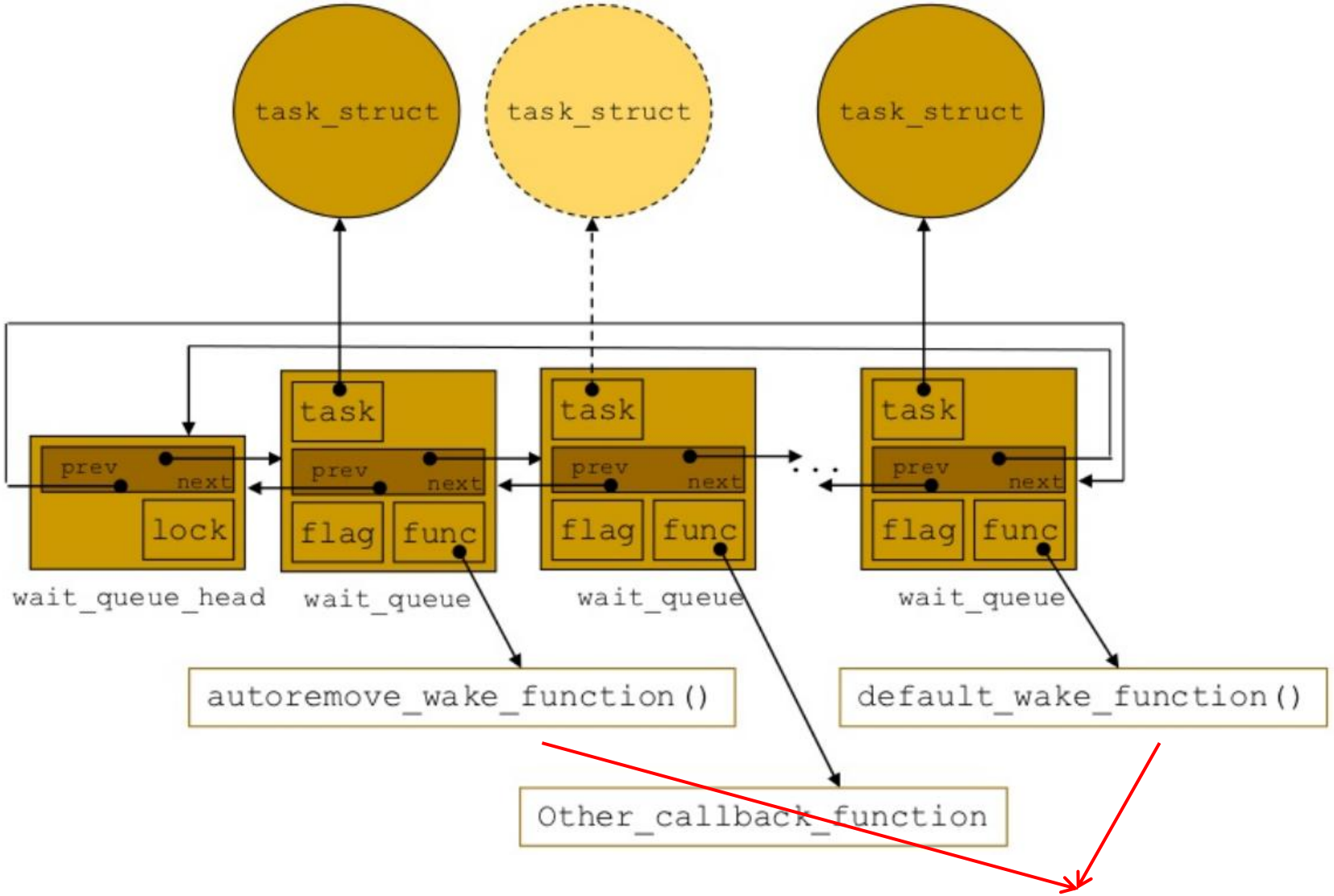
```
#define WQ_FLAG_EXCLUSIVE      0x01

struct wait_queue_entry {
        unsigned int            flags;
        void                    *private;
        wait_queue_func_t       func;
        struct list_head        entry;
}
```

```
struct wait_queue_head {
        spinlock_t              lock;
        struct list_head        head;
}
```

The beginning of the list, **wait_queue_head**, is a distinguished element with a different structure. It is not related to any process, but has a **spinlock** used to ensure the atomicity of operations on the queue.

Wait queues



Wait queue, R. Lee, 2011.

Operations on wait queues

Queues of pending processes are handled both by **kernel functions** and by **interrupts**, so operations on them (inserting and deleting) must be executed with **interrupts disabled**.

Functions to perform the operations of **adding** and **removing** elements from the queue:

1. **Inserts** an element at the **beginning** of the queue without setting the flag WQ_FLAG_EXCLUSIVE.

```
void add_wait_queue(struct wait_queue_head *wq_head,
                                    struct wait_queue_entry *wq_entry)
{
        unsigned long flags;

        wq_entry->flags &= ~WQ_FLAG_EXCLUSIVE;
        spin_lock_irqsave(&wq_head->lock, flags);
        list_add(&wq_entry->entry, &wq_head->head);
        spin_unlock_irqrestore(&wq_head->lock, flags);
}
```

Simplified version

Operations on wait queues

2. Inserts an element at the **end** of the queue setting the flag **WQ_FLAG_EXCLUSIVE**.

```
void add_wait_queue_exclusive(struct wait_queue_head *wq_head,
                              struct wait_queue_entry *wq_entry)
{
        unsigned long flags;

        wq_entry->flags |= WQ_FLAG_EXCLUSIVE;
        spin_lock_irqsave(&wq_head->lock, flags);
        list_add_tail(&wq_entry->entry, &wq_head->head);
        spin_unlock_irqrestore(&wq_head->lock, flags);
}
```

Simplified version

3. Deletes an element from the queue.

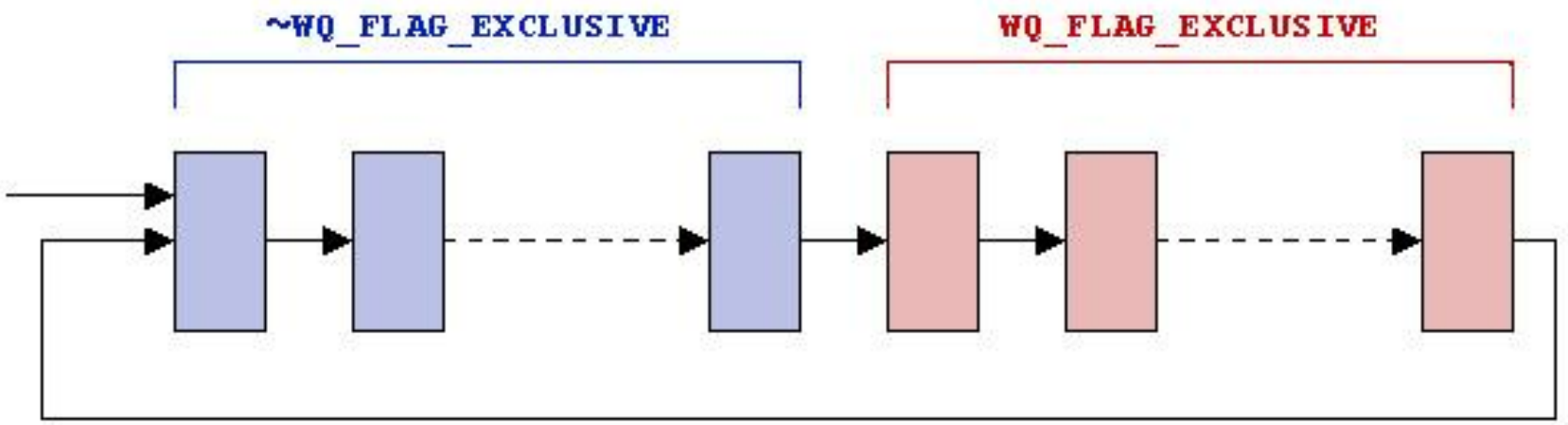
```
void remove_wait_queue(struct wait_queue_head *wq_head,
                       struct wait_queue_entry *wq_entry)
{
        unsigned long flags;

        spin_lock_irqsave(&wq_head->lock, flags);
        list_del(&wq_entry->entry);
        spin_unlock_irqrestore(&wq_head->lock, flags);
}
```

Simplified version

Operations on wait queues

Using **add_wait_queue()** and **add_wait_queue_exclusive()** guarantees a specific layout of processes in the queue:



Processes of type **exclusive**, with a flag value of 1, are awakened by the kernel selectively, and **nonexclusive**, with a flag of 0, are awakened always.

The process waiting for the resource to be allocated exclusively is usually an **exclusive** process. If the event can affect many processes and everyone should be awakened, these are **nonexclusive** processes.

Putting processes to sleep

Functions used to put the process to sleep work on the **current process**. In other words, the **process puts itself to sleep**.

Mainly the **wait_event()** macro is used for putting the process to sleep.

The function **prepare_to_wait_event()** does what **add_wait_queue()**, only that it additionally sets the state of the process (to **TASK_UNINTERRUPTIBLE**).

The function **finish_wait()** does what **remove_wait_queue()**, only that it also sets the process state to **TASK_RUNNING**.

The macro **wait_event()** first makes sure that the **condition** passed as parameter is not actually met yet. Then it puts the process to sleep.

Each time the process is woken up, it checks again whether the **condition** is met and if so, it leaves the loop.

Otherwise, the control is transferred back to the **scheduler** and the process is put to **sleep**.

The process goes to the state **TASK_UNINTERRUPTIBLE** and is inserted in the **waiting queue**.

Procedure **schedule()** chooses another process for execution.

Finishing the execution of the code, i.e. removing the process from the queue, takes place **after the process is resumed** by calling **schedule()** somewhere else in the code.

```
define wait_event(wq_head, condition)   \
do {                                    \
   ...                                  \
   if (condition)                        \
      break;                            \
      do {                              \
        struct wait_queue_entry __wq_entry;   \
        ...                             \
        for (;;) {                      \
            prepare_to_wait_event(&wq_head,  \
           &__wq_entry, TASK_UNINTERRUPTIBLE); \
            if (condition)               \
                  break;                \
            ...                         \
              schedule();                   \
        }                               \
        finish_wait(&wq_head, &__wq_entry);   \
   } while (0)                          \
} while (0)
```

Simplified version

Waking processes up

The **wake_up_process** function wakes up **one process** asynchronously.

```
int wake_up_process(struct task_struct *p)
{
        return try_to_wake_up(p, TASK_NORMAL, 0);
}
```

Many macros are available in the kernel to **wake up** processes waiting in queues.

They all are based on the same function:

```
#define TASK_NORMAL                    (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)

#define wake_up(x)                      __wake_up(x, TASK_NORMAL, 1, NULL)
#define wake_up_nr(x, nr)               __wake_up(x, TASK_NORMAL, nr, NULL)
#define wake_up_all(x)                  __wake_up(x, TASK_NORMAL, 0, NULL)
#define wake_up_locked(x)               __wake_up_locked((x), TASK_NORMAL)
#define wake_up_interruptible(x)        __wake_up(x, TASK_INTERRUPTIBLE, 1, NULL)
#define wake_up_interruptible_nr(x, nr) __wake_up(x, TASK_INTERRUPTIBLE, nr, NULL)
#define wake_up_interruptible_all(x)    __wake_up(x, TASK_INTERRUPTIBLE, 0, NULL)
```

Waking processes up

The function **__wake_up()** wakes up threads blocked on a waitqueue. It enters critical section and delegates the work to **__wake_up_common()**.

The kernel then browses the list of **sleeping processes** and invokes for them the **appropriate function**. It avoids **unnecessary wake-up** of processes.

```
static void __wake_up_common(struct wait_queue_head *wq_head,
            unsigned int mode,  int nr_exclusive, int wake_flags, void *key)
{
      wait_queue_entry_t *curr, *next;

      list_for_each_entry_safe(curr, next, &wq_head->head, entry) {
            unsigned flags = curr->flags;
            int ret;
            ...
            ret = curr->func(curr, mode, wake_flags, key);
            if (ret < 0)
               break;
            if (ret && (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
               break;
      }
      return nr_exclusive;
}
```

try_to_wake_up()

Simplified version

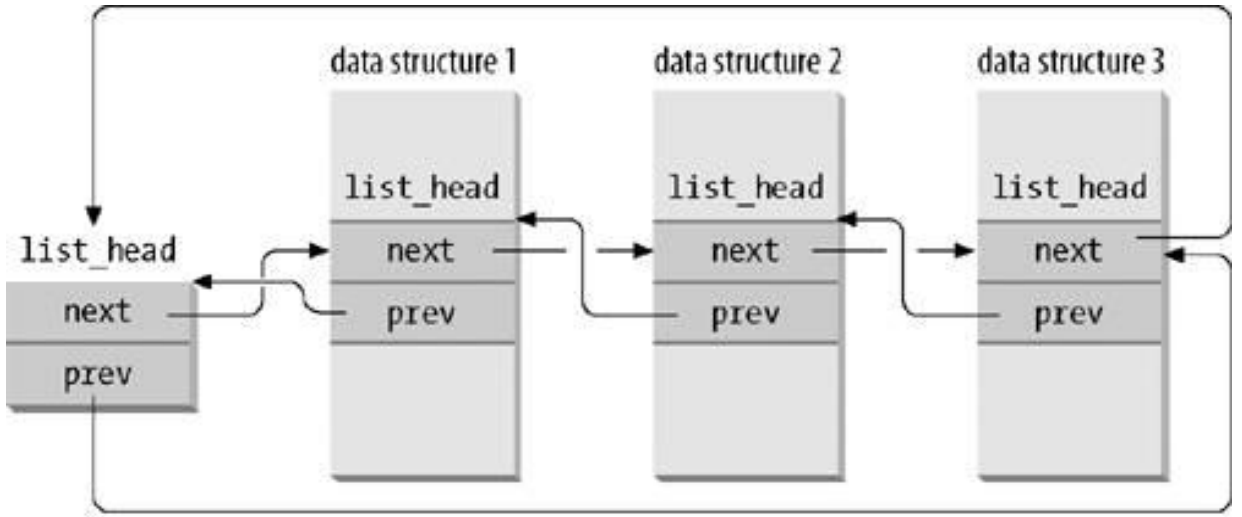
Lists in Linux

The Linux kernel uses **standard doubly linked lists** to implement process queues. All basic operations on the lists are carried out in **O(1)** time.

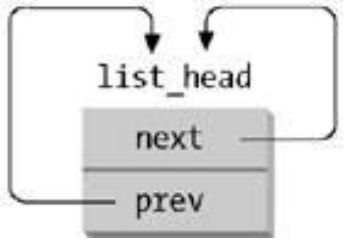
The node in the list contains pointers to the **next** and **previous** list element:

```
struct list_head {
    struct list_head *next, *prev;
};
```

An **empty list** consists of a single element (dummy, pointer to the list) for which the successor (and also the predecessor) point to himself.



(a) a doubly linked listed with three elements



(b) an empty doubly linked list

Doubly linked list
(source: Bovet, Cesati, Understanding the Linux Kernel)

Lists in Linux

The macro **LIST_HEAD_INIT** creates a **list_head** structure with the given **name** and initializes it to create an empty list.

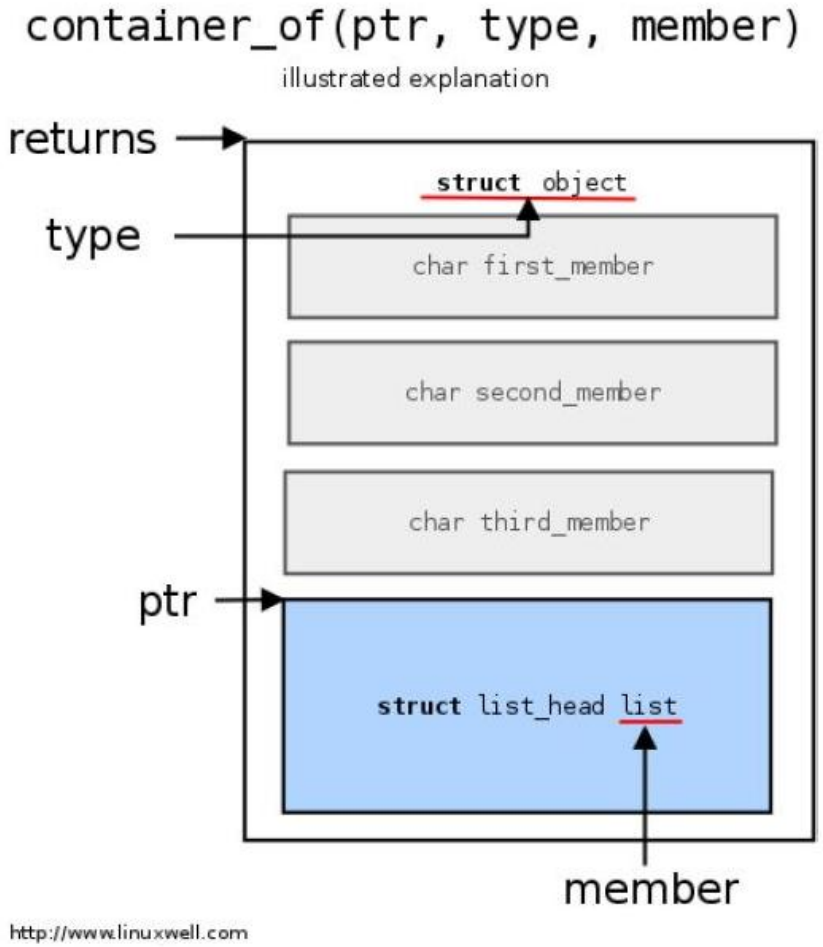
```
#define LIST_HEAD_INIT(name) = { &(name), &(name) }
```

If there is a separate list structure, how can we get to the **object**, which is stored in the list? While wandering through the list, we encounter only elements of the type **list_head**, which in themselves mean nothing.

Since **list_head** is a field in some structure, we can easily track where the structure is in the memory, if only we have information about it.

```
#define list_entry(ptr, type, member)  \
        container_of(ptr, type, member)
```

It passes an object of the type **type**, whose **member** with the given name is of type **list_head**, and contains the list element indicated by **ptr**.



Lists in Linux

Basic functions to operate on such a list:

- **list_add**(new, head) – **inserts** a new element right **after** the head element;
- **list_add_tail**(new, head) – **inserts** a new element right **before** head, that is at the end of the list;
- **list_del**(entry) – **removes** an element from the list;
- **list_empty**(head) – **checks** if the list is **empty**;
- **list_splice**(list, head) – **merges** two lists by inserting the list **list** after the **head** element.

The macro – iterator on all elements of the list:

```
#define list_for_each(pos, head) \
        for (pos = (head)->next; pos != (head); pos = pos->next)
```

Lists in Linux

An example of using a list structure: **task_struct** (process descriptor) contains a couple of them:

```
struct task_struct {
    struct list_head tasks;
    struct list_head children;
    struct list_head sibling;
    ...
  }
```

In **hash tables**, where memory usage is important, and not a fixed access time to the last element, **doubly non-circular linked lists** are used.

The list head is stored in the **hlist_head** structure and is the pointer to the first element.

Each element is represented by a structure of the type **hlist_node**.

What is lost is the ability to access the tail in O(1).

```
struct hlist_head {
    struct hlist_node *first;
  ;

  struct hlist_node {
    struct hlist_node *next, **pprev;
  };
```

Family of processes

Each process in the system has exactly **one parent** and can have one or more **children**. The corresponding links are kept in the process descriptor in the fields **parent** and **children**.

The link to the **parent** of the **current** process:

```
struct task_struct *task = current->parent;
```

Walking through the list of children:

```
struct task_struct *task;
struct list_head *list;

list_for_each(list, &current->children) {
      task = list_entry(list, struct task_struct, sibling);
      /* now the task indicates the next child of the current process */
 }
```

This code will always lead us to the **init_task** process:

```
struct task_struct *task;

for (task = current; task != &init_task; task = task->parent)
  ;
/* now task points to init_task */
```

The **pstree** command shows running processes as a tree.

The tree is rooted at either pid or init (systemd) if pid is omitted (**init** has **pid=1** and is a child of **process 0**).