Linux Kernel Synchronization and Timers

Ahmed Ali-Eldin

What we will cover

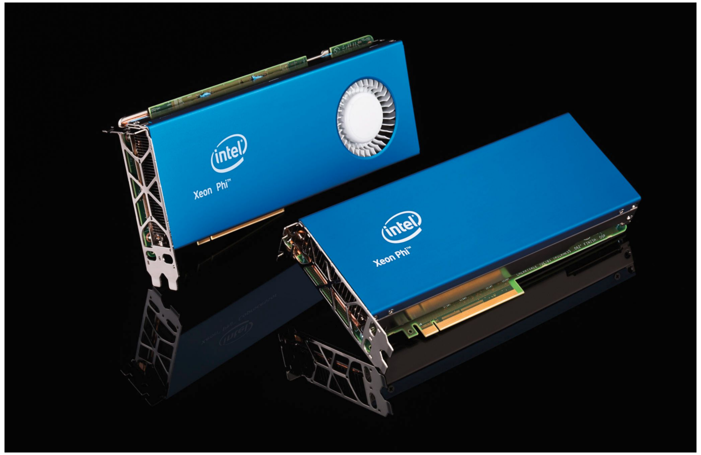
- Kernel Synchronization methods
- Timers in the Kernel

In the beginning

- Hope you are all doing great in these times!
- Will have office hours on zoom at your convenience for the Linux lectures
    - Send an email!
    - Reason: different time zones make it very hard to be fair
    - Just ping me if you need a zoom meeting, send in your time zone, we will figure something out!

Why synchronize?

- Developers must ensure that shared resources are protected from concurrent access
    - Shared resources in the kernel are many, including, the Task list, for example
    - Multiprocessors are everywhere
    - Kernel code is itself mostly preemptive



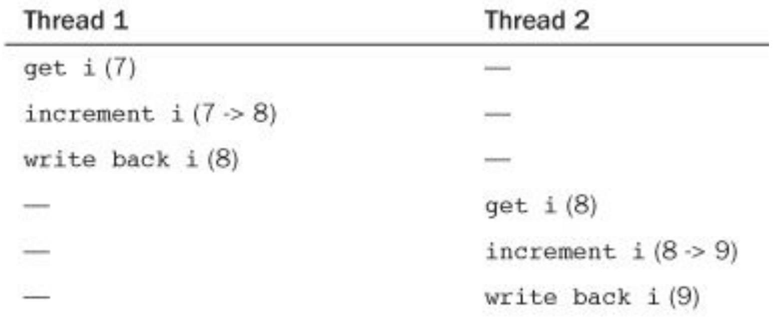
Intel Xeon Phi 72 cores

Remember: Critical regions and Race conditions

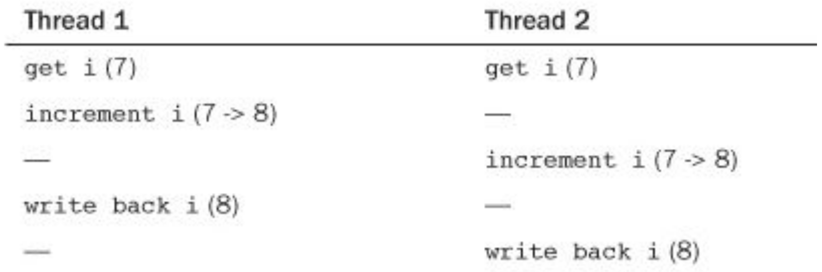
- Code paths that access and manipulate shared data are called critical regions (also called critical sections)
- Critical regions should be executed atomically
  - From the beginning to the end with no interruptions
- Race conditions
  - If it is possible for two threads of execution to be simultaneously executing within the same critical region

Race condition example

Variable has a value of 7



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)



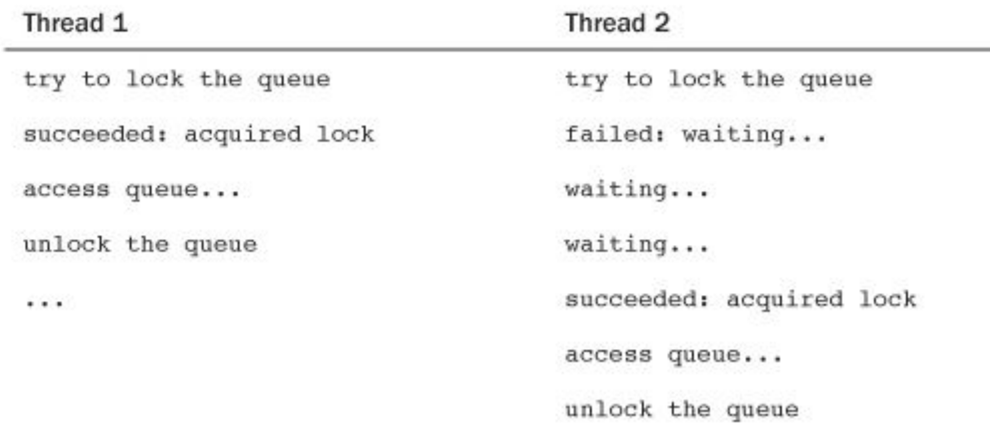
Thread 1	Thread 2
`get i` (7)	`get i` (7)
`increment i` (7 -> 8)	—
—	`increment i` (7 -> 8)
`write back i` (8)	—
—	`write back i` (8)

A Kernel example of Races

- A kernel queue with tasks implemented as a linked list
  - Prevalent in the kernel, e.g., Task queues, network queues, etc
- Two functions manipulate the queue
  - A Queue function that adds new tasks to the end of the queue
  - A Dequeue function that removes the head of the queue for processing
- Imagine two processors trying to queue two new tasks with no atomicity
  - Queuing is not one single instruction

Remember: Locking

- Locks are advisory and voluntary.
- Linux alone implements a handful of different locking mechanisms
- Most significant difference between the various mechanisms is the behavior when the lock is unavailable because another thread already holds it
  - Busy-wait
  - Sleep
- Locks are implemented using atomic operations that ensure no race exists
- Implementation is architecture specific
  - Most use *test-and-set* atomic instruction
  - X86 uses *compare-and-exchange*



Thread 1	Thread 2
`try to lock the queue`	`try to lock the queue`
`succeeded: acquired lock`	`failed: waiting...`
`access queue...`	`waiting...`
`unlock the queue`	`waiting...`
`...`	`succeeded: acquired lock`
	`access queue...`
	`unlock the queue`

Kernel causes of concurrency

- **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.

Easier said than done

- More than 300 race conditions found by Google in 2 days in 2019!
- Using a tool called KTSAN and another one called KCSAN
  - https://github.com/google/ktsan/wiki
  - https://github.com/google/ktsan/wiki/KCSAN

Suggested activity for bored people: See if you can use something similar for Minix (e.g. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual)

Notes

- When doing kernel dev, you should consider concurrency in the design
- Easier to build in design than to detect after implementation
    - Local data for a thread most probably need no locking as it is stored only in that thread's stack
    - Data local to a task is similarly usually only accessed by the specific task on one processor at a time
    - Global kernel data-structures is shared--->lock when accessing
    - Lock data not code!
- Kernel jargon
    - Code that is safe from concurrent access from an interrupt handler is said to be interrupt-safe.
    - Code that is safe from concurrency on symmetrical multiprocessing machines is SMP-safe.
    - Code that is safe from concurrency with kernel preemption is preempt-safe

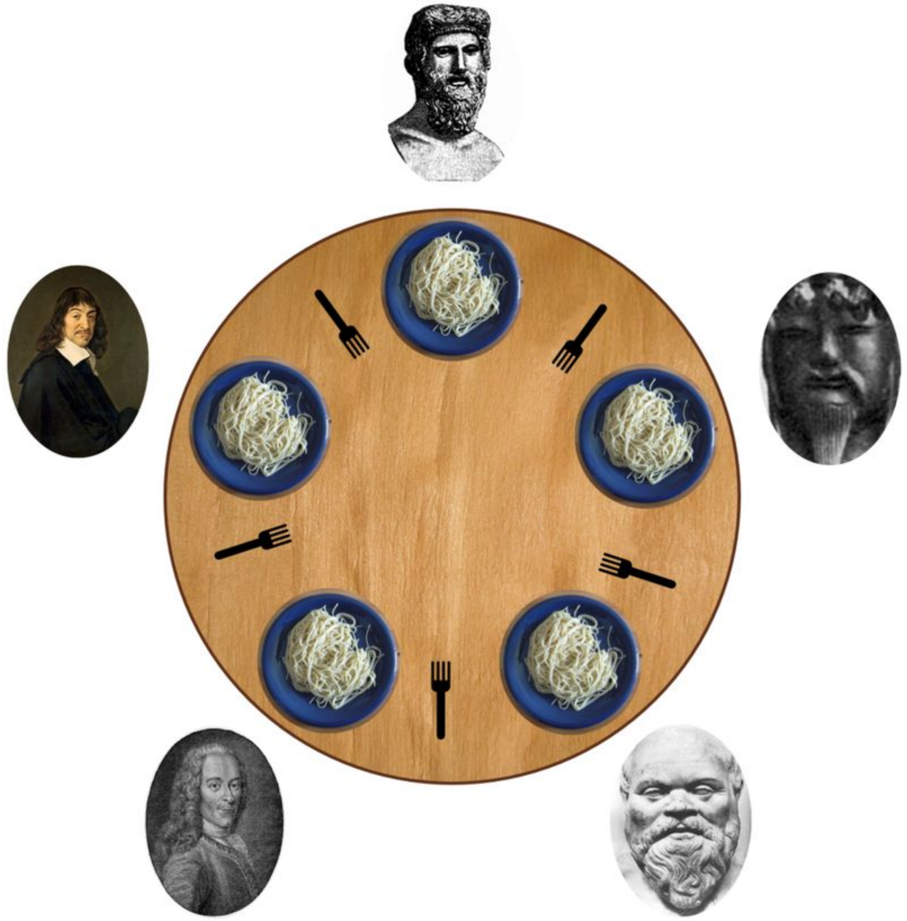
When accessing data

- Ask the following questions
    - 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?
    - What prevents the data from being freed out from under me?
    - What happens if this function is called again on another processor?
    - Given the proceeding points, how am I going to ensure that my code is safe from concurrency?

Remember: Deadlock

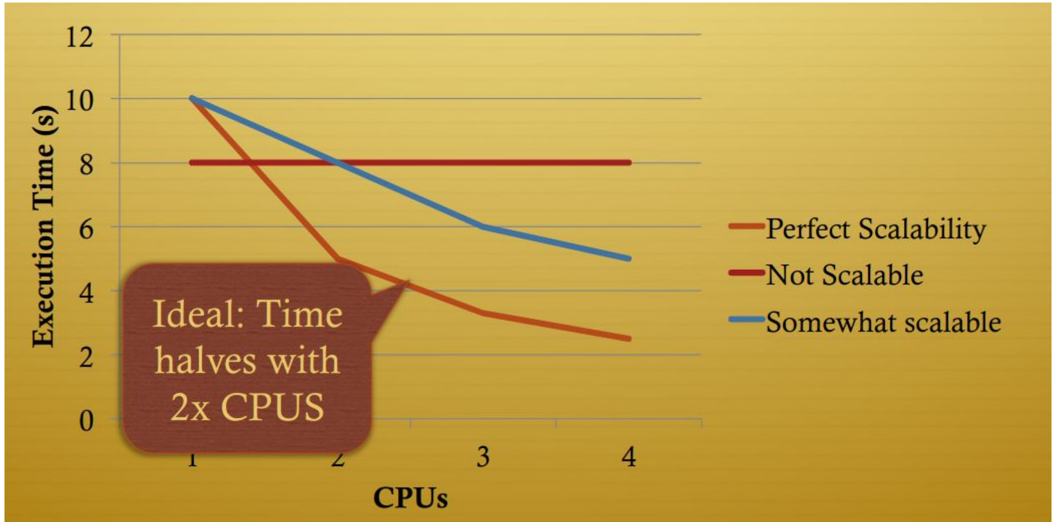
- Lock ordering is important
- Or maybe, locking oneself

```
acquire lock
acquire lock, again
wait for lock to become available
...
```



Lock contention

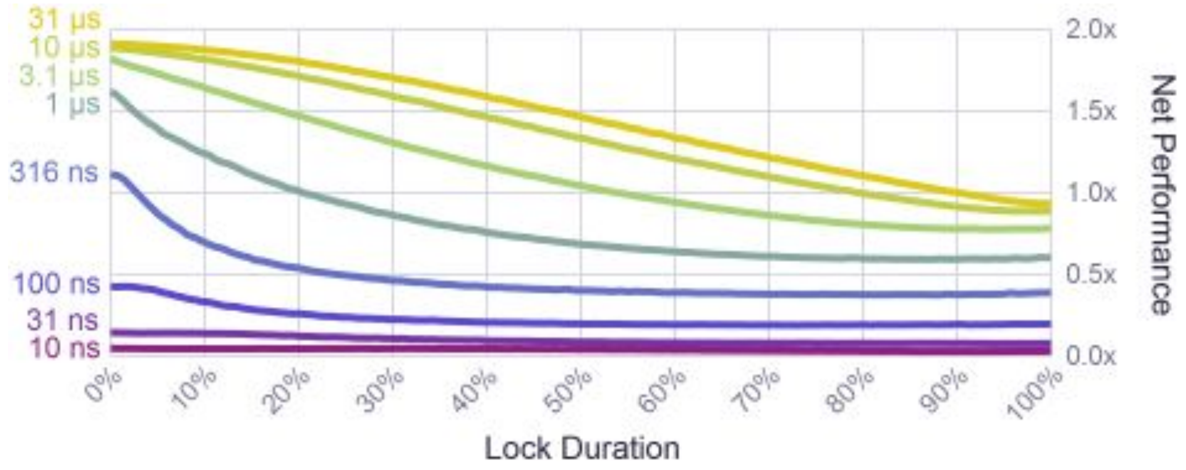
- Popular locks can keep lots of threads waiting
- Kills scalability
- You can measure how bad your locking is in the kernel using https://www.kernel.org/doc/html/latest/locking/lockstat.html



But also lock overheads

This are measured on a windows machine!

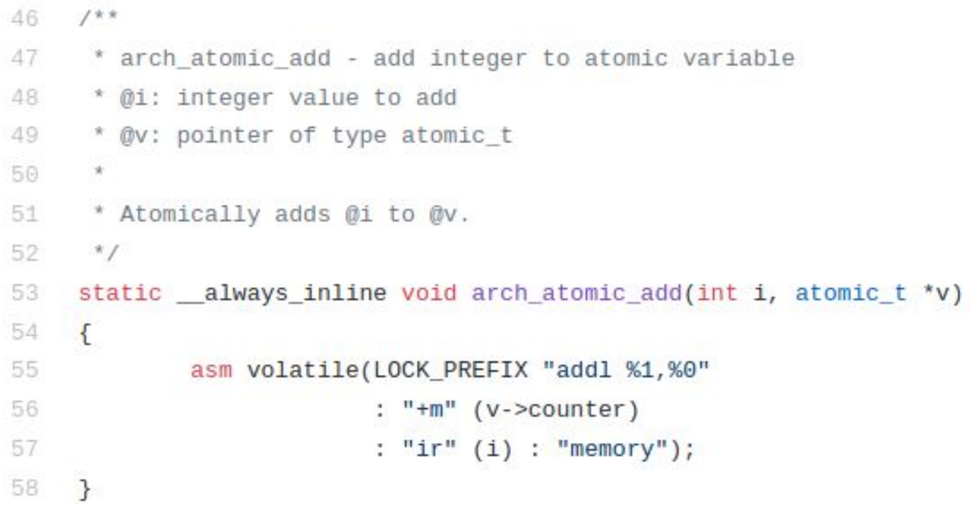
Fine line between coarse-grained locking and fine-grained locking.



Kernel synchronization methods

Integer Atomic Kernel operations

- Atomic integer methods operate on a special data type, *atomic_t* (does not work with *int*)
  - Defined in https://github.com/torvalds/linux/blob/master/tools/include/linux/types.h
  - Atomic operations for the x86 defined in https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/atomic.h, atomic64_32.h, and atomic64_64.h



```
/**
 * arch_atomic_add - add integer to atomic variable
 * @i: integer value to add
 * @v: pointer of type atomic_t
 *
 * Atomically adds @i to @v.
 */
static __always_inline void arch_atomic_add(int i, atomic_t *v)
{
        asm volatile(LOCK_PREFIX "addl %1,%0"
                     : "+m" (v->counter)
                     : "ir" (i) : "memory");
}
```

Some Atomic Integer Methods

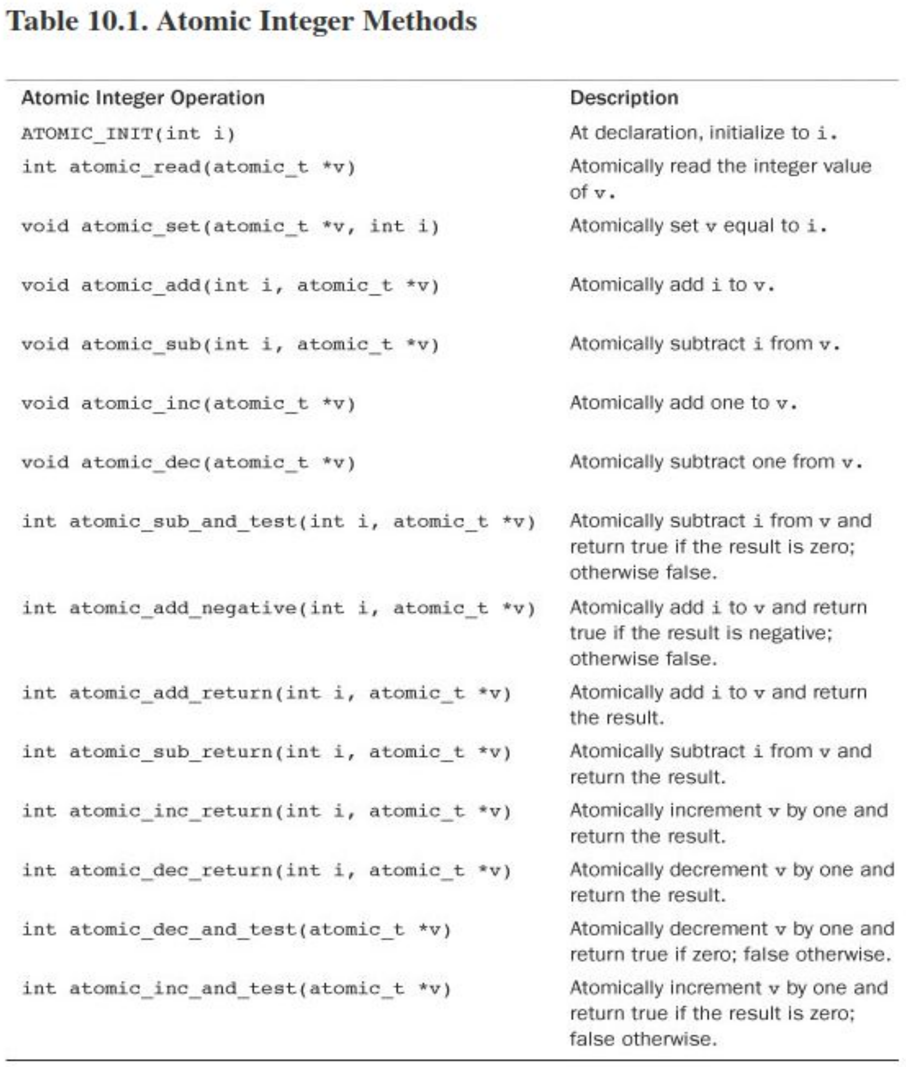
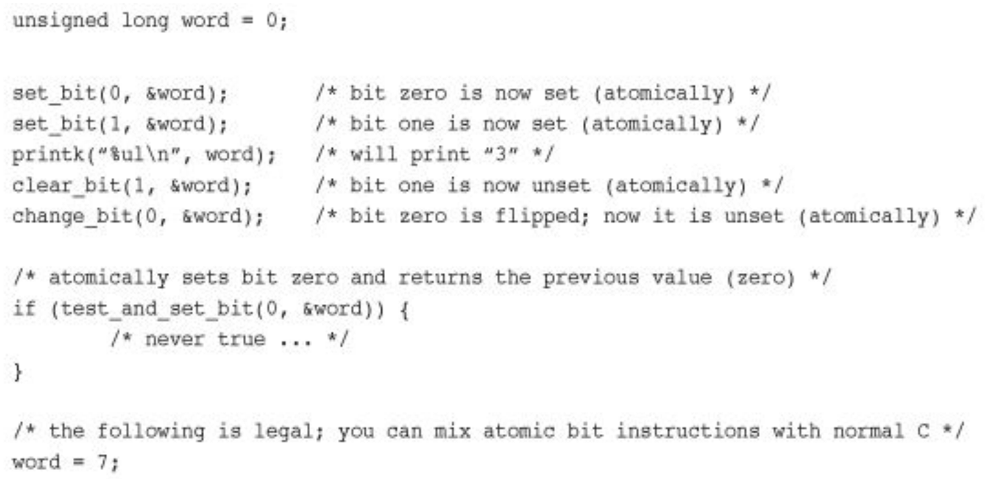


Table 10.1. Atomic Integer Methods

Atomic Integer Operation	Description
`ATOMIC_INIT(int i)`	At declaration, initialize to `i`.
`int atomic_read(atomic_t *v)`	Atomically read the integer value of `v`.
`void atomic_set(atomic_t *v, int i)`	Atomically set `v` equal to `i`.
`void atomic_add(int i, atomic_t *v)`	Atomically add `i` to `v`.
`void atomic_sub(int i, atomic_t *v)`	Atomically subtract `i` from `v`.
`void atomic_inc(atomic_t *v)`	Atomically add one to `v`.
`void atomic_dec(atomic_t *v)`	Atomically subtract one from `v`.
`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_add_return(int i, atomic_t *v)`	Atomically add `i` to `v` and return the result.
`int atomic_sub_return(int i, atomic_t *v)`	Atomically subtract `i` from `v` and return the result.
`int atomic_inc_return(int i, atomic_t *v)`	Atomically increment `v` by one and return the result.
`int atomic_dec_return(int i, atomic_t *v)`	Atomically decrement `v` by one and return the result.
`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.

Bitwise Kernel Atomic operations

- The Bitwise operations does not require a special atomic data type as it can be used on any memory address
  - Allowing the user to safely perform bitwise operations on memory
  - Defined in https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/bitops.h
  - The arguments are a pointer and a bit number. Bit zero is the least significant bit of the given address.



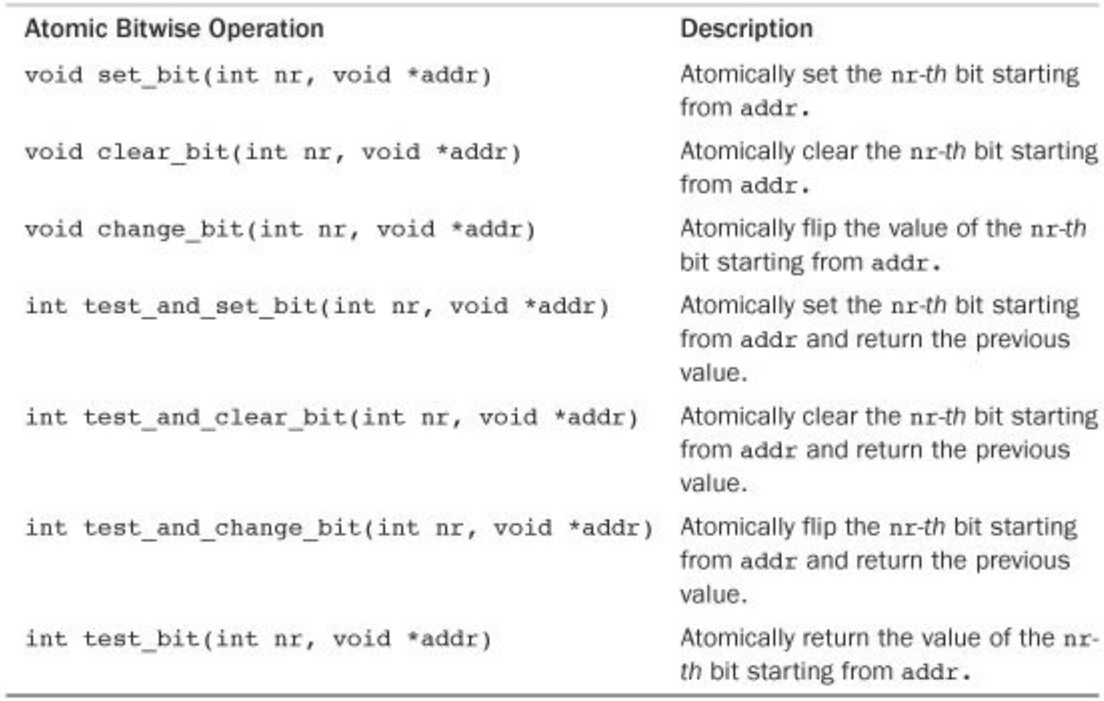
```
unsigned long word = 0;

set_bit(0, &word);        /* bit zero is now set (atomically) */
set_bit(1, &word);        /* bit one is now set (atomically) */
printk(“%ul\n”, word);    /* will print “3” */
clear_bit(1, &word);      /* bit one is now unset (atomically) */
change_bit(0, &word);     /* bit zero is flipped; now it is unset (atomically) */

/* atomically sets bit zero and returns the previous value (zero) */
if (test_and_set_bit(0, &word)) {
        /* never true ... */
}

/* the following is legal; you can mix atomic bit instructions with normal C */
word = 7;
```

Some Bitwise operations



Atomic Bitwise Operation	Description
`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`.

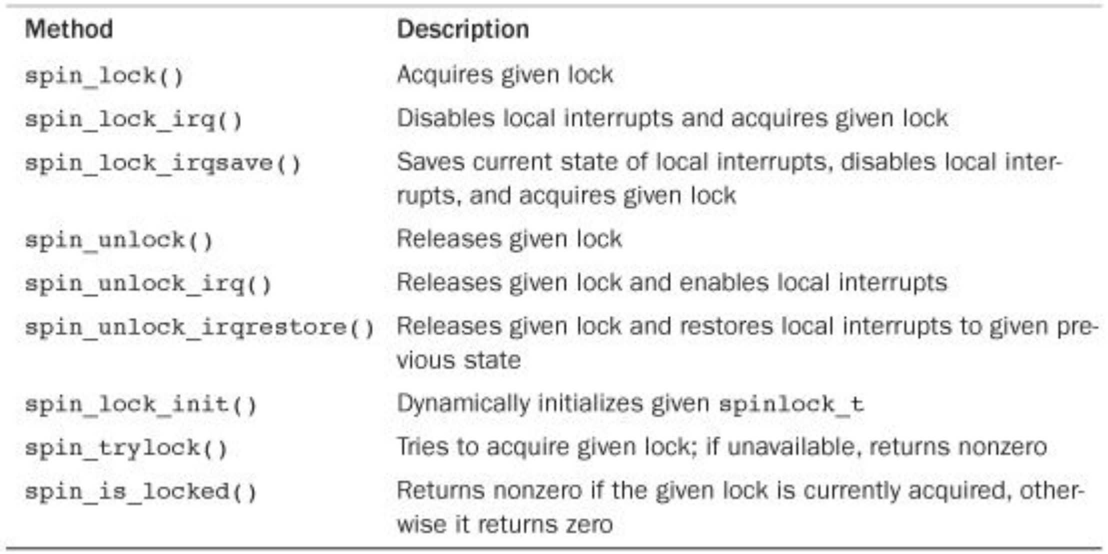
Some notes

- For each atomic bitwise operation, there is a non-atomic equivalent that can be used
- The non-atomic operation starts with double underscores, e.g., test_bit() is __test_bit().
- Use the non-atomic operation if a lock already protects the data
- Non-atomic operation can be faster

Kernel Spin locks

- A spin lock is a lock that can be held by at most one thread of execution
    - If a thread of execution attempts to acquire a spin lock while it is already held, which is called contended, the thread busy loops—spins—waiting for the lock to become available.
    - Rule of thumb, hold a spin lock if waiting is less than the duration of two context switches
    - Implementation is architecture specific, for the x86 https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/spinlock.h
- Used in interrupt handlers after disabling local interrupts

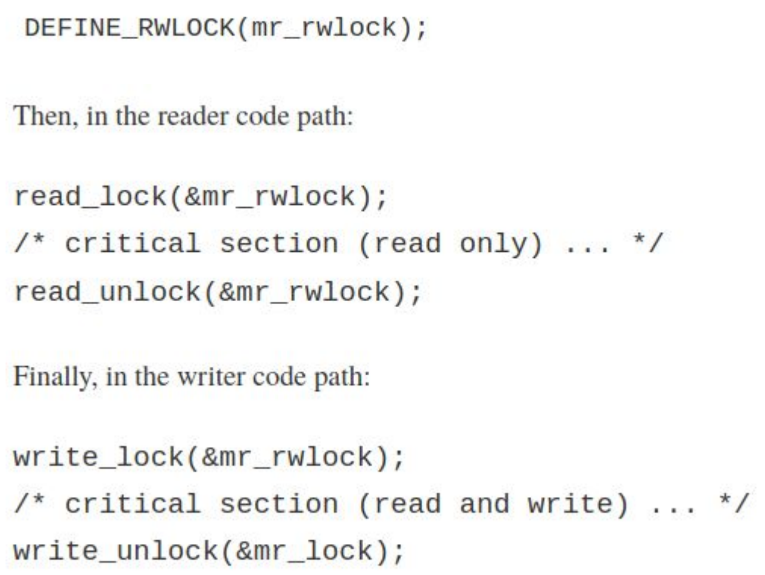
Some Spinlock methods



Method	Description
`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

Reader-Writer Spin Locks

- Sometimes, lock usage can be clearly divided into reader and writer paths
  - Writing demands mutual exclusion, reading does not
  - sometimes called shared/exclusive or concurrent/exclusive locks



```
DEFINE_RWLOCK(mr_rwlock);
```

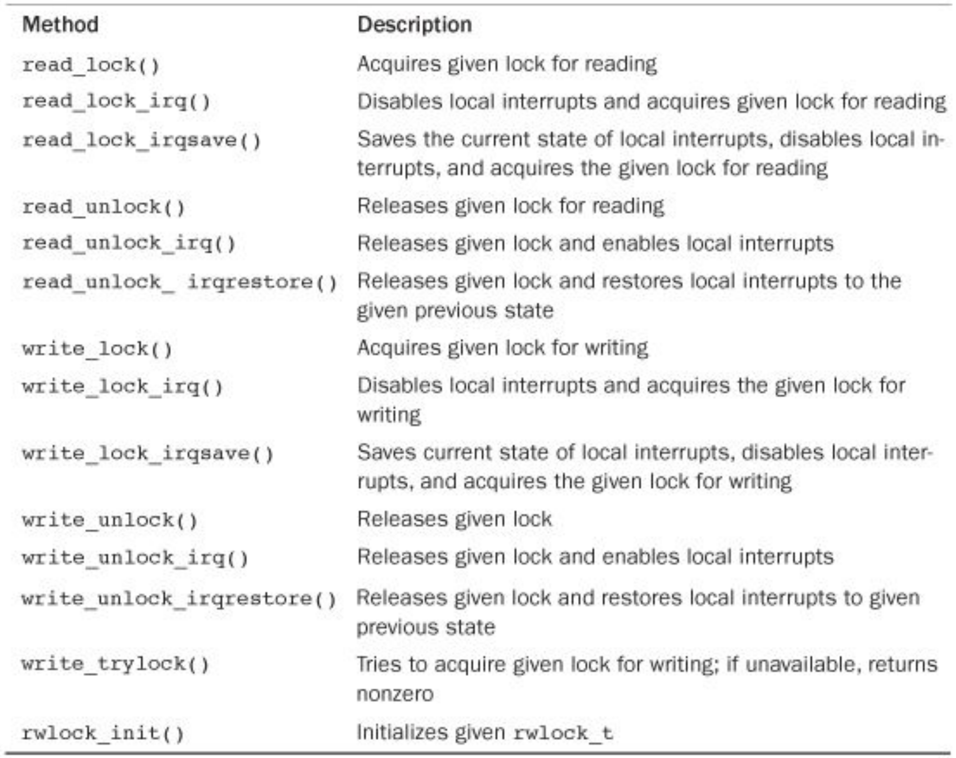
Then, in the reader code path:

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

Finally, in the writer code path:

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

Reader-Writer block methods

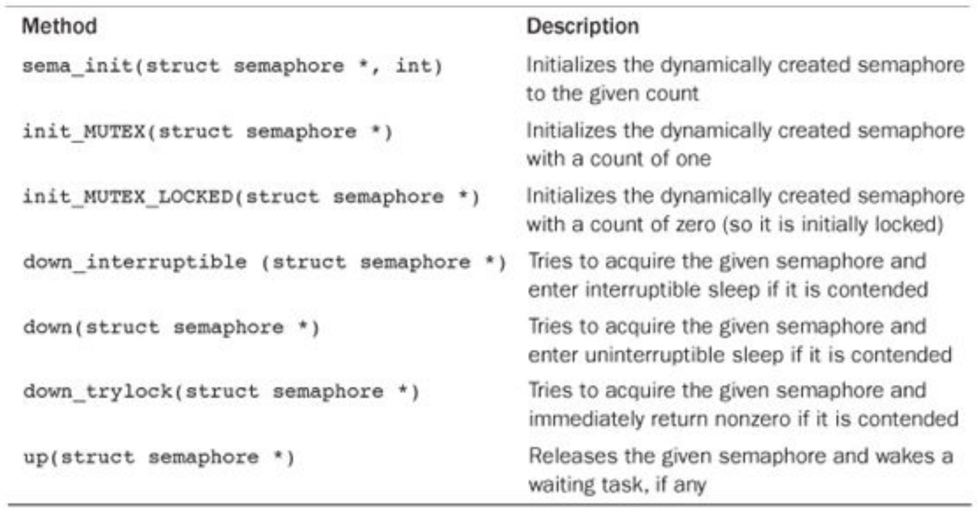


Method	Description
`read_lock()`	Acquires given lock for reading
`read_lock_irq()`	Disables local interrupts and acquires given lock for reading
`read_lock_irqsave()`	Saves the current state of local interrupts, disables local interrupts, and acquires the given lock for reading
`read_unlock()`	Releases 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
`rwlock_init()`	Initializes given `rwlock_t`

Semaphores

- Semaphores in Linux are sleeping locks
  - If the semaphore is not available, the task sleeps in the wait queue
  - Once semaphore available, one task is woken by the kernel
- Semaphores used for locks held for a long time
- A task can sleep while holding a semaphore
- Two types of Semaphores in the kernel
  - Binary/Mutex
  - Counting (almost always initialized with a count of one)
- Defined in https://github.com/torvalds/linux/blob/master/include/linux/semaphore.h
- Implemented in https://github.com/torvalds/linux/blob/master/kernel/locking/semaphore.c

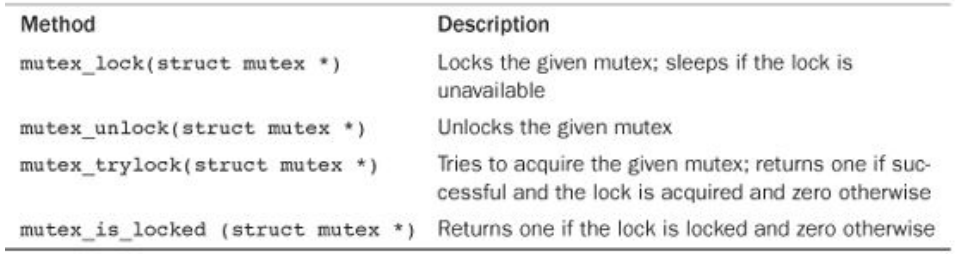
Semaphore methods



Method	Description
`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

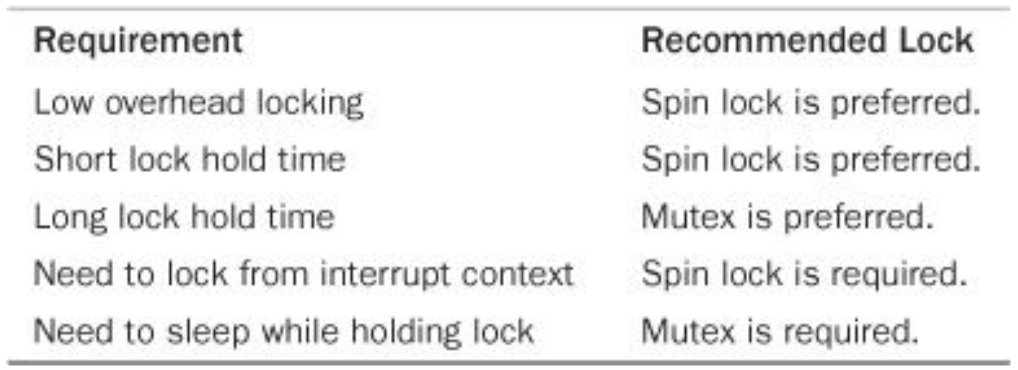
Mutex adds constraints over a semaphore

- Only one task can hold the mutex at a time.
- Whoever locked a mutex must unlock it. You cannot lock a mutex in one context and then unlock it in another. This means that the mutex isn't suitable for more complicated synchronizations between kernel and user-space.
- Recursive locks and unlocks are not allowed. That is, you cannot recursively acquire the same mutex, and you cannot unlock an unlocked mutex.
- A process cannot exit while holding a mutex
- A mutex cannot be acquired by an interrupt handler or bottom half



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

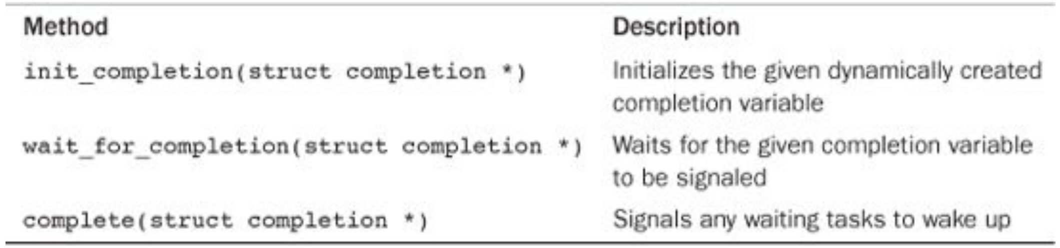
Spinlocks vs Semaphores



Requirement	Recommended Lock
Low overhead locking	Spin lock is preferred.
Short lock hold time	Spin lock is preferred.
Long lock hold time	Mutex is preferred.
Need to lock from interrupt context	Spin lock is required.
Need to sleep while holding lock	Mutex is required.

Completion Variables

- Just another way to build a semaphore like locking mechanism
- One task waits on the completion variable while another task performs some work. When the other task has completed the work, it uses the completion variable to wake up any waiting tasks.



Method	Description
`init_completion(struct completion *)`	Initializes the given dynamically created completion variable
`wait_for_completion(struct completion *)`	Waits for the given completion variable to be signaled
`complete(struct completion *)`	Signals any waiting tasks to wake up

The historical obsolete BKL

- The Big Kernel Lock is no longer part of Linux
- It was a global spin-lock

Sequential Locks

- Useful to provide a lightweight and scalable lock for use with many readers and a few writers
- Designed to not starve writers
- Reader wants a consistent set of information and is willing to retry if the information changes
- Works by maintaining a sequence counter
    - Whenever the data in question is written to, a lock is obtained and a sequence number is incremented. Grabbing the write lock makes the value odd whereas releasing it makes it even because the lock starts at zero
    - Prior to and after reading the data, the sequence number is read. If the values are the same, a write did not begin in the middle of the read
    - If the values are even, a write is not underway

Timers and Time Management

Kernel Notion of Time

- The hardware provides a system timer that the kernel uses to gauge the passing of time.
- This system timer works off of an electronic time source, such as a digital clock or the frequency of the processor.
- The system timer goes off (often called hitting or popping) at a pre-programmed frequency, called the tick rate.
- When the system timer goes off, it issues an interrupt that the kernel handles via a special interrupt handler.
  - a tick and is equal to 1/(tick rate) seconds
- Important for many kernel functions, specially scheduling

The Tick Rate: HZ

- The frequency of the system timer (the tick rate) is programmed on system boot based on a static preprocessor define, **HZ**.
  - Defined in https://github.com/torvalds/linux/blob/master/include/asm-generic/param.h and https://github.com/torvalds/linux/blob/master/include/uapi/asm-generic/param.h
  - Default, tick once every 10 ms
- When writing kernel code, never assume that HZ has any given value as it is architecture dependant

HZ tradeoffs: Pros of Higher values

- Increasing the tick rate means the timer interrupt runs more frequently. Consequently, the work it performs occurs more often.
  - The timer interrupt has a higher resolution and, consequently, all timed events have a higher resolution
  - The accuracy of timed events improves.
  - System calls such as poll() and select() that optionally employ a timeout value execute with improved precision.
  - Measurements, such as resource usage or the system uptime, are recorded with a finer resolution.
  - Process preemption occurs more accurately. Assume a given process is running and has 2 milliseconds of its timeslice remaining. In 2 milliseconds, the scheduler should preempt the running process and begin executing a new process. Unfortunately, this event does not occur until the next timer interrupt

HZ tradeoffs: Cons of Higher values

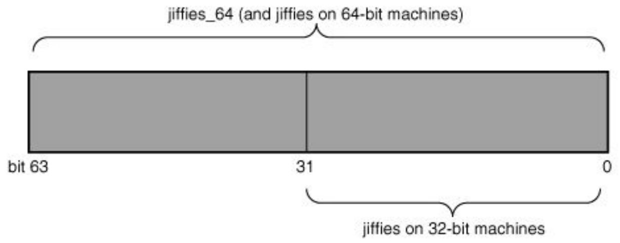
- Linux actually at one point had the default value of HZ set to 1000. Cons of the approach are
  - A higher tick rate implies more frequent timer interrupts, which implies higher overhead,
    - the processor must spend more time executing the timer interrupt handler.
  - The higher the tick rate, the more time the processor spends executing the timer interrupt.
  - More frequent thrashing of the processor’s cache
  - Increase in power consumption

Tickless OS

- Linux supports tickless operation mode
    - The system dynamically schedules the timer interrupt in accordance with pending timers.
    - Instead of firing the timer interrupt every, say, 1ms, the interrupt is dynamically scheduled and rescheduled as needed.
        - If the next timer is set to go off in 3ms, the timer interrupt fires in 3ms. After that, if there is no work for 50ms, the kernel reschedules the interrupt to go off in 50ms.
    - For how to enable this mode: https://github.com/torvalds/linux/blob/e9a83bd2322035ed9d7dcf35753d3f984d76c6a5/Documentation/timers/no_hz.rst
- Saves a lot of power on idle systems
- Many OSes today support or run tickless

Jiffies

- A global variable holding the number of ticks that have occurred since the system booted.
  - System uptime can be calculated easily using *jiffies/HZ* seconds
  - Defined in https://github.com/torvalds/linux/blob/05ef8b97ddf9aed40df977477daeab01760d7f9a/include/linux/jiffies.h
- Jiffies variable has always been an unsigned long
  - 32 bits in size on 32-bit architectures and 64-bits on 64-bit architectures.
    - With a tick rate of 100, a 32-bit jiffies variable would overflow in about 497 days.
    - With HZ increased to 1000, however, that overflow now occurs in just 49.7 days
    - This is important for long running servers!
    - So even for 32 bit architectures, there is a 64-bit jiffie
    - Kernel handles wrap-around correctly


jiffies_64 (and jiffies on 64-bit machines)
bit 63
31
0
jiffies on 32-bit machines

Hardware Clocks and Timers: The RTC

- The real-time clock (RTC) provides a nonvolatile device for storing the system time.
  - RTC continues to keep track of time even when the system is off by way of a small battery included on the system board.
  - On the PC architecture, the RTC and the CMOS are integrated, and a single battery keeps the RTC running and the BIOS settings preserved.
- kernel reads the RTC and uses it to initialize the wall time, which is stored in the xtime variable

Hardware Clocks and Timers: The System Timer

- The system timer provides a mechanism for driving an interrupt at a periodic rate
    - Some architectures implement this via an electronic clock that oscillates at a programmable frequency
    - On x86, the primary system timer is the programmable interrupt timer (PIT).
        - The PIT exists on all PC machines and has been driving interrupts since the days of DOS.
        - The kernel programs the PIT on boot to drive the system timer interrupt (interrupt zero) at HZ frequency.

The Timer Interrupt Handler

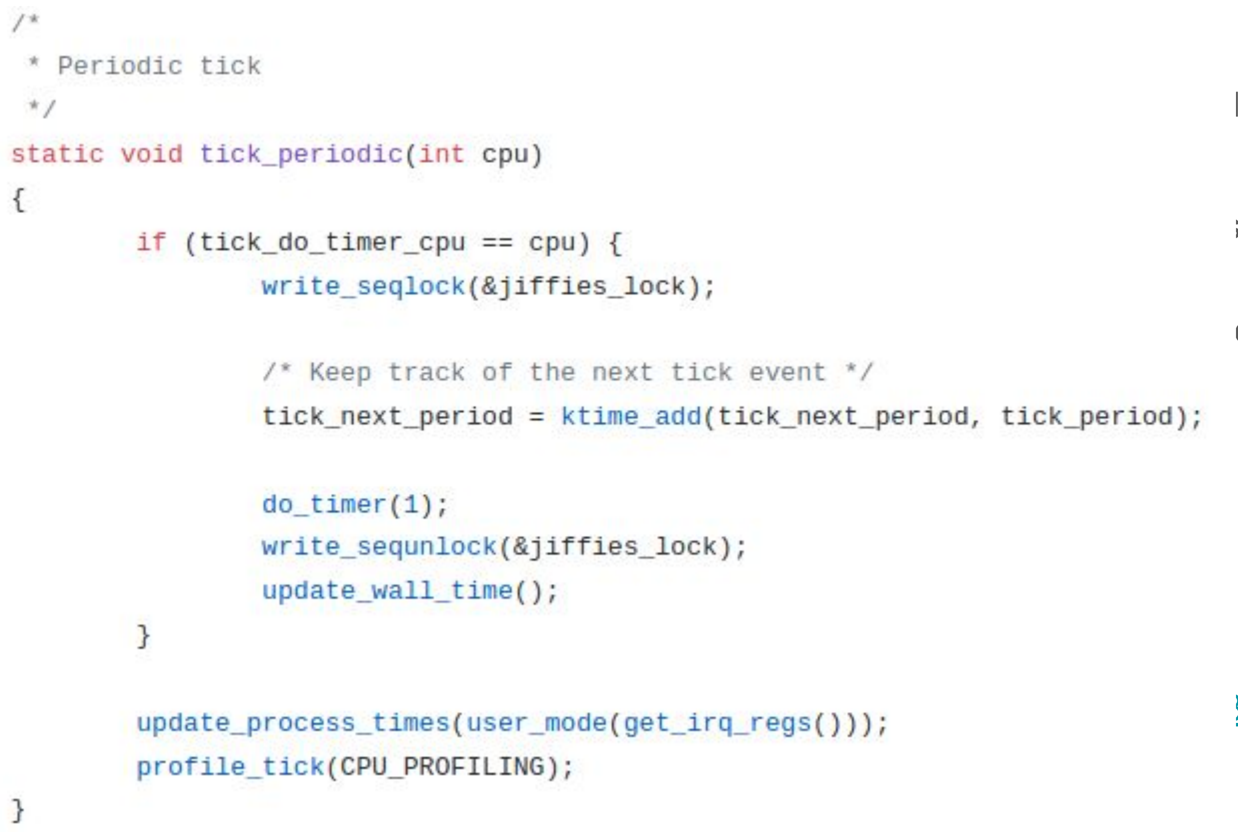
- The timer interrupt is broken into two pieces: an architecture-dependent and an architecture-independent routine
- The architecture dependant part does at least the following functions
  - Obtain the xtime_lock lock, which protects access to jiffies_64 and the wall time value, xtime.
  - Acknowledge or reset the system timer as required.
  - Periodically save the updated wall time to the real time clock.
  - Call the architecture-independent timer routine, tick_periodic()

The Timer Interrupt Handler: tick_periodic()

- The architecture-independent routine, tick_periodic(), performs much more work:
  - Increment the jiffies_64 count by one. (This is safe, even on 32-bit architectures, because the xtime_lock lock was previously obtained.)
  - Update resource usages, such as consumed system and user time, for the currently running process.
  - Run any dynamic timers that have expired
  - Execute scheduler_tick()
  - Update the wall time, which is stored in xtime.
  - Calculate the load average.
- Defined here https://github.com/torvalds/linux/blob/b0be0eff1a5ab77d588b76bd8b1c92d5d17b3f73/kernel/time/tick-common.c

The Timer Interrupt Handler: tick_periodic()

- The arc ich more work:
  - Incr , because the xtim
  - Upc ently running proc
  - Rur
  - Exe
  - Upc
  - Calc
- Defined h https://gitl 3/kernel/time/tic k-commo



```
/*
 * Periodic tick
 */
static void tick_periodic(int cpu)
{
	if (tick_do_timer_cpu == cpu) {
		write_seqlock(&jiffies_lock);

		/* Keep track of the next tick event */
		tick_next_period = ktime_add(tick_next_period, tick_period);

		do_timer(1);
		write_sequnlock(&jiffies_lock);
		update_wall_time();
	}

	update_process_times(user_mode(get_irq_regs()));
	profile_tick(CPU_PROFILING);
}
```

The Time of Day

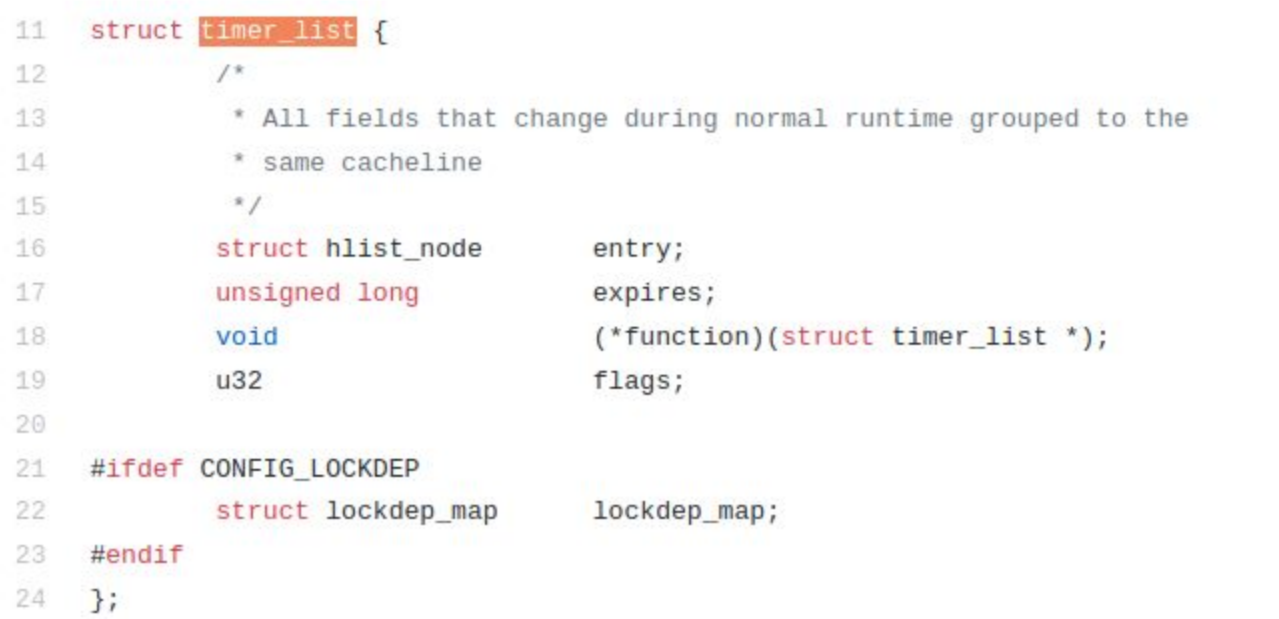
- Defined in https://github.com/torvalds/linux/blob/master/include/uapi/linux/time.h and https://github.com/torvalds/linux/blob/b0be0eff1a5ab77d588b76bd8b1c92d5d17b3f73/kernel/time/timekeeping.c
- The xtime.tv_sec value stores the number of seconds that have elapsed since January 1, 1970 (UTC).
    - This date is called the epoch

Dynamic Kernel Timers

- Timers—sometimes called dynamic timers or kernel timers—are essential for managing the flow of time in kernel code.
  - Kernel code often needs to delay execution of some function until a later time
- Using Timers
  - Initialize timer setup with an expiration time
  - specify a function to execute upon said expiration
  - activate the timer
  - The given function runs after the timer expires
  - Timers are not cyclic and the timer is destroyed after it expires
- Defined and methods to manipulate in https://github.com/torvalds/linux/blob/04cbfba6208592999d7bfe6609ec01dc3fde73f5/include/linux/timer.h

Dynamic Kernel Timers

- Timer sential for
  mana
  - K
- Using
  - In
  - sp
  - ac
  - Tl
  - T
- Define



```
struct timer_list {
        /*
         * All fields that change during normal runtime grouped to the
         * same cacheline
         */
        struct hlist_node       entry;
        unsigned long           expires;
        void                    (*function)(struct timer_list *);
        u32                     flags;

#ifdef CONFIG_LOCKDEP
        struct lockdep_map      lockdep_map;
#endif
};
```

https://github.com/torvalds/linux/blob/04cbfba6208592999d7bfe6609ec01dc3fde73f5/include/linux/timer.h

Timers implementation

- Timers are stored in a linked list
- To optimize insertions and realizing which timers expired
  - the kernel partitions timers into five groups based on their expiration value
  - Timers move down through the groups as their expiration time draws closer
  - The partitioning ensures that, in most executions of the timer softirq, the kernel has to do little work to find the expired timers.

Other methods for delaying execution

- Sometimes there is a need for a short delay, or a simpler way to induce delays
  - The kernel provides three functions for microsecond, nanosecond, and millisecond delays, defined in <linux/delay.h> and <asm/delay.h>, which do not use jiffies
    - void udelay(unsigned long usecs)
    - void ndelay(unsigned long nsecs)
    - void mdelay(unsigned long msecs)
  -