Kernel Synchronization I

Xiaoguang Wang

UIC COMPUTER SCIENCE

Homework

hw6 due tomorrow

proj proposal due tomorrow;

- sign up on the google sheet

paper reading due next Monday

(No homework planned after the spring break)

hw7, hw8 released

- CPU profiler (part 2 & 3)
- due: Feb 23rd, March 1st

Do not share your homework and answers on GitHub, etc.!

Past lectures

Tools: building, exploring, and debugging Linux kernel

Core kernel infrastructure

- syscall, module, kernel data structures

Process management & scheduling

- CFS, scheduler class

Interrupt & interrupt handler

- top half (interrupt handler), bottom halves (softirq, tasklet, work queue)

Agenda

Background on multicore processing

Introduction to synchronization

Atomic operations

Spinlock, reader-writer spinlock (RWLock)

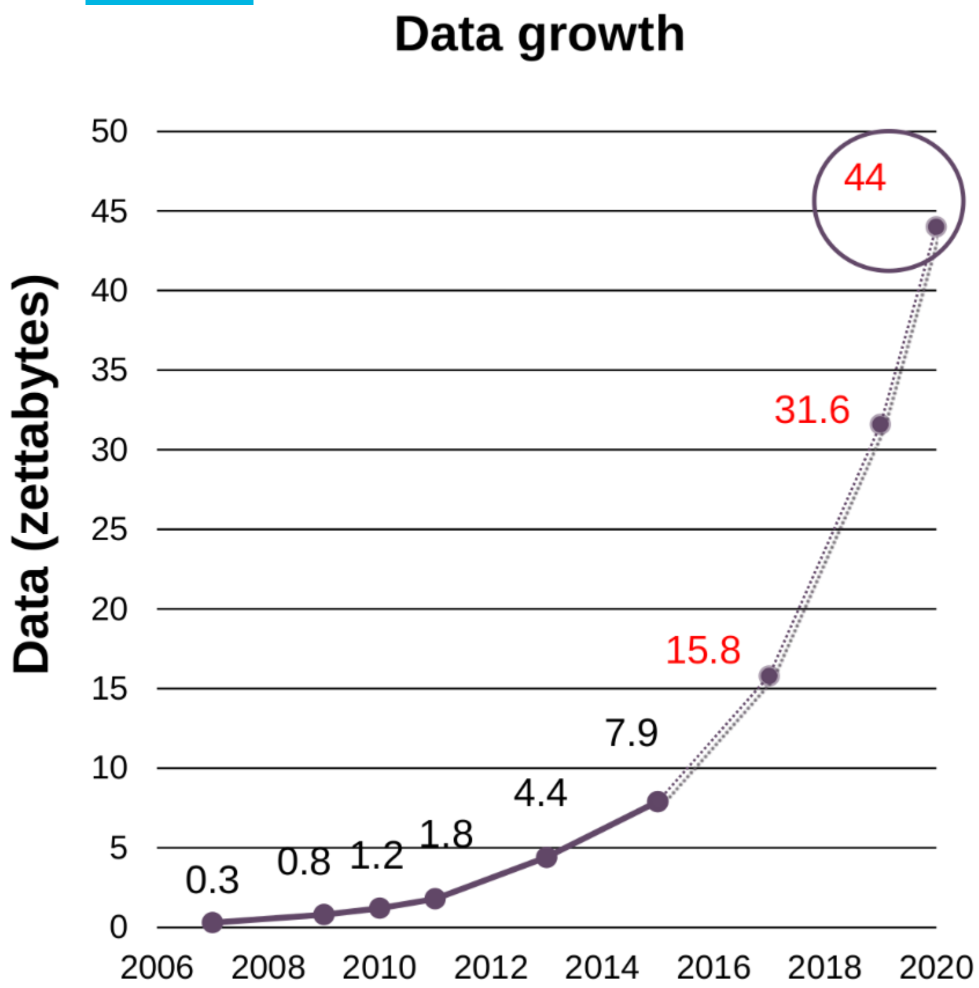
Semaphore, mutex

Sequential lock (seqlock)

Completion variable

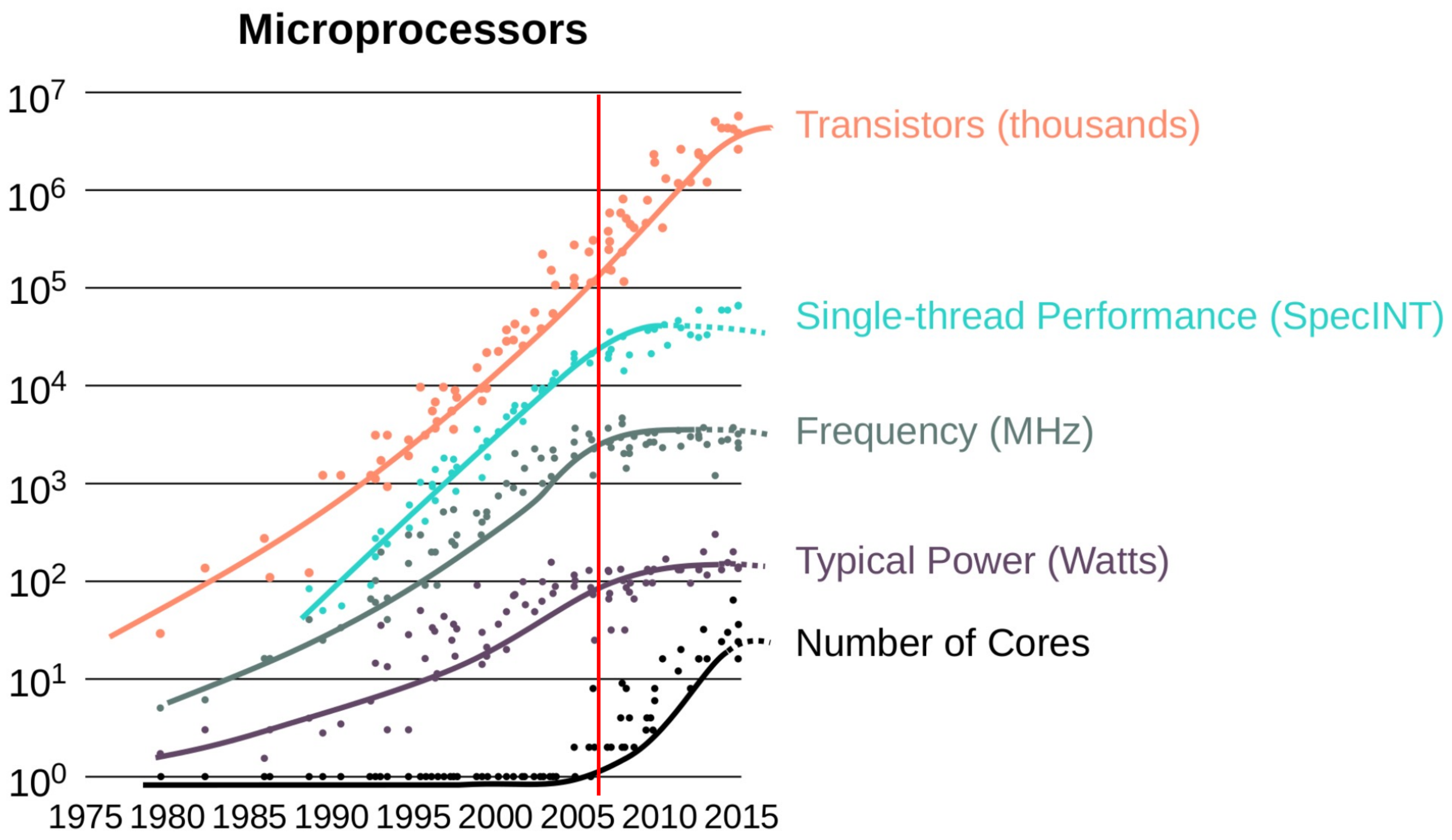
RCU

Data growth is already exponential



1 zettabytes = 109 terabytes

Single-core scaling stopped



Single-core scaling stopped

Increasing clock frequency is not possible anymore

- Power consumption: higher frequency → higher power consumption
- Wire delay: range of a wire in one clock cycle

Limitation in Instruction Level Parallelism (ILP)

- 1980s: more transistors → superscalar → pipeline
- 1990s: multi-way issue, out-of-order issue, branch prediction

Multi-core processors

Moore's law: the observation that the number of transistors in a dense integrated circuit doubles approximately every two years

~ 2007: make a single-core processor faster

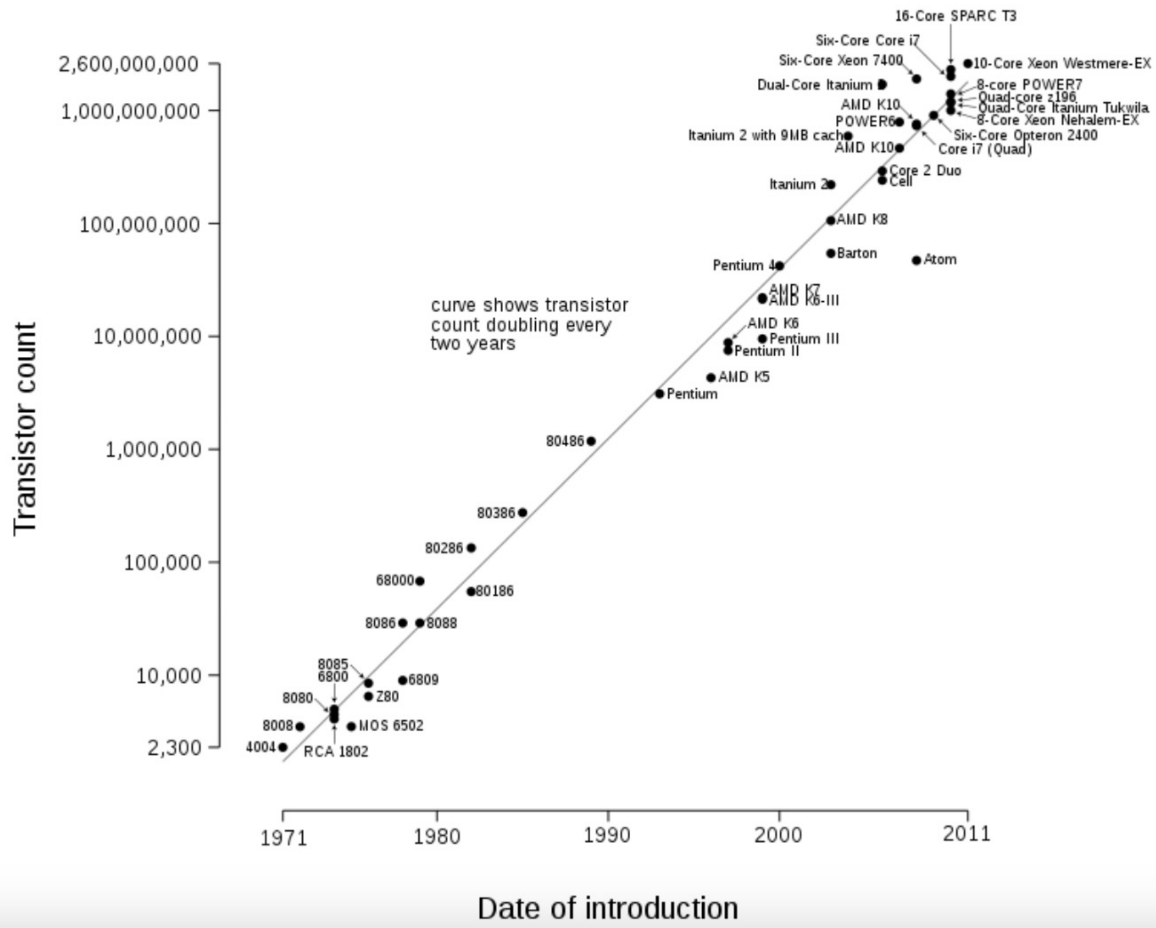
- deeper processor pipeline, branch prediction, out-of-order execution, etc.

2007 ~: increase the number of cores in a chip

- multi-core processor

Multi-core processors

Microprocessor Transistor Counts 1971-2011 & Moore's Law



An example

Product Collection	5th Generation Intel® Xeon® Scalable Processors	5th Generation Intel® Xeon® Scalable Processors	5th Generation Intel® Xeon® Scalable Processors	5th Generation Intel® Xeon® Scalable Processors
Vertical Segment	Server	Server	Server	Server
rocessor Number	8593Q	8592+	8581V	8592V
thography	Intel 7	Intel 7	Intel 7	Intel 7
Use Conditions	Server/Enterprise	Server/Enterprise	Server/Enterprise	Server/Enterprise
Recommended Customer Price	$12400.00	$11600.00	$7568.00	$10995.00

CPU Specifications

Total Cores	64	64	60	64

UIC COMPUTER SCIENCE

Kernel synchronization

The kernel is programmed using the shared memory model

Critical section (also called critical region)

- Code paths that access and manipulate shared data
- Must execute atomically without interruption
- Should not be executed in parallel on SMP → sequential part

Race condition

- Two threads concurrently executing the same critical region → Bug!

Concurrent data accesses in kernel

Q: What kind of kernel code will be considered to concurrently access data? Any scenarios?

Why do we need protection?

```c
int i = 7;
void foo(void) {
  i++;
}
```

Q: What happens if two threads concurrently execute `foo()`?

Q: What happens if two threads concurrently update `i`?

Q: Is incrementing `i` an atomic operation?

Updating a single variable

A single C statement

```
/* C code */
i++;
```

It can be translated into multiple machine instructions

```
/* Machine instructions */
get the current value of i and copy it into a register
add one to the value stored in the register
write back to memory the new value of i
```

Now, check what happens if two threads concurrently update `i`

Updating a single variable

Two threads are running. Initial value of `i` is 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)

As expected, 7 incremented twice is 9

Updating a single variable

Two threads are running. Initial value of `i` is 7

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)

If both threads read the initial value of `i` before it is incremented, both threads increment and save the same value.

➔ variable `i` contains the value 8 when, in fact, it should now contain 9

Solution: atomic instruction

Thread 1	Thread 2
`increment & store i` (7 -> 8)	—
—	`increment & store i` (8 -> 9)

Or conversely

Thread 1	Thread 2
—	`increment & store` (7 -> 8)
`increment & store` (8 -> 9)	—

It would never be possible for the two atomic operations to interleave.

The processor would physically ensure that it was impossible.

Atomic instruction in x86

`XADD DEST SRC`

Operation

TEMP = SRC + DEST

SRC = DEST

DEST = TEMP

`LOCK XADD DEST SRC`

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Atomic operations

Provide instructions that execute atomically without interruption

Non-atomic update: `i++`

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)

Atomic operations

Atomic update: `atomic_inc(&i)`

Thread 1	Thread 2
`increment & store i` (7 -> 8)	—
—	`increment & store i` (8 -> 9)

Or conversely

Thread 1	Thread 2
—	`increment & store` (7 -> 8)
`increment & store` (8 -> 9)	—

Atomic operations

Examples

- `fetch-and-add`: atomic increment
- `test-and-set`: set a value at a memory location and return the previous value
- `compare-and-swap`: modify the content of a memory location only if the previous content is equal to a given value

Linux provides two APIs:

- Integer atomic operations
- Bitwise atomic operations

Atomic integer operations

```c
/* Type definition: include/linux/types.h */
typedef struct {
    int counter;
} atomic_t;

typedef struct {
    long counter;
} atomic64_t;

/* API definition: include/linux/atomic.h */
/* Usage example */
atomic_t v;                     /* define v */
atomic_t u = ATOMIC_INIT(0);    /* define and initialize u to 0 */

atomic_set(&v, 4);              /* v = 4 (atomically) */
atomic_add(2, &v);              /* v = v + 2 == 6 (atomically) */
atomic_inc(&v);                 /* v = v + 1 == 7 (atomically) */
```

Atomic int operations (32-bit)

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.

Atomic int operations (64-bit)

Atomic Integer Operation	Description
`ATOMIC64_INIT(long i)`	At declaration, initialize to `i`.
`long atomic64_read(atomic64_t *v)`	Atomically read the integer value of `v`.
`void atomic64_set(atomic64_t *v, int i)`	Atomically set `v` equal to `i`.
`void atomic64_add(int i, atomic64_t *v)`	Atomically add `i` to `v`.
`void atomic64_sub(int i, atomic64_t *v)`	Atomically subtract `i` from `v`.
`void atomic64_inc(atomic64_t *v)`	Atomically add one to `v`.
`void atomic64_dec(atomic64_t *v)`	Atomically subtract one from `v`.
`int atomic64_sub_and_test(int i, atomic64_t *v)`	Atomically subtract `i` from `v` and return true if the result is zero; otherwise false.
`int atomic64_add_negative(int i, atomic64_t *v)`	Atomically add `i` to `v` and return true if the result is negative; otherwise false.

Atomic int operations: example

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/types.h>

#define PRINT_PREF "[SYNC_ATOMIC] "
atomic_t counter; /* shared data: */
struct task_struct *read_thread, *write_thread;

static int writer_function(void *data)
{
    while(!kthread_should_stop()) {
        atomic_inc(&counter);
        msleep(500);
    }
    do_exit(0);
}
```

Atomic int operations: example

```
static int read_function(void *data)
{
    while(!kthread_should_stop()) {
        printk(PRINT_PREF "counter: %d\n", atomic_read(&counter));
        msleep(500);
    }
    do_exit(0);
}

static int __init my_mod_init(void)
{
    printk(PRINT_PREF "Entering module.\n");

    atomic_set(&counter, 0);

    read_thread = kthread_run(read_function, NULL, "read-thread");
    write_thread = kthread_run(writer_function, NULL, "write-thread");

    return 0;
}
```

Atomic int operations: example

```
static void __exit my_mod_exit(void)
{
    kthread_stop(read_thread);
    kthread_stop(write_thread);
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

Code available on blackboard!

Locking

Atomic operations are not sufficient for protecting **shared data in long and complex critical regions**

- E.g., `page_tree` of an `inode` (page cache)

What is needed is a way of making sure that only one thread manipulates the data structure at a time

- A mechanism for preventing access to a resource while another thread of execution is in the marked region. → **lock**

Deadlocks

Situations in which one or several threads are waiting on locks for one or several resources that will never be freed

- None of the threads can continue

Self-deadlock

- a thread tries to acquire a lock already held by the thread
- NOTE: Linux does not support **recursive locks**

Deadlocks

Deadly embrace (ABBA deadlock)

Thread 1	Thread 2
`acquire lock A`	`acquire lock B`
`try to acquire lock B`	`try to acquire lock A`
`wait for lock B`	`wait for lock A`

Q: How to prevent/detect deadlocks?

Spinlocks

The most commonly-used lock in the kernel

When a thread tries to acquire an already held lock, it spins while waiting for the lock become available.

- Wasting processor time when spinning is too long

In an interrupt context (a thread cannot sleep)

- Kernel provides special spinlock API for data structures shared in interrupt context

In process context, do not sleep while holding a spinlock

- Kernel preemption is disabled

Spinlocks

```c
/* include/linux/spinlock_types.h */
DEFINE_SPINLOCK(my_lock);

spin_lock(&my_lock);
/* critical region */
spin_unlock(&my_lock);
```

Lock/unlock methods **disable/enable kernel preemption** and **acquire/release the lock**

Lock is compiled away on uniprocessor systems

- Still needs do disabled/re-enable preemption to prevent interleaving of task execution

`spin_lock()` is not recursive! → self-deadlock

Quiz #1: find a deadlock

```c
/* WARNING!!! THIS CODE HAS A DEADLOCK!!! WARNING!!! */
DEFINE_HASHTABLE(global_hashtbl, 10);
DEFINE_SPINLOCK(hashtbl_lock);

irqreturn_t irq_handler(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock(&hashtbl_lock);
    /* access global_hashtbl */
    spin_unlock(&hashtbl_lock);
}

int foo(void)
{
    /* A function running in process context */
    spin_lock(&hashtbl_lock);
    /* access global hashtable */

    spin_unlock(&hashtbl_lock);
}
```

Quiz #2: find a deadlock

```c
/* WARNING!!! THIS CODE HAS A DEADLOCK!!! WARNING!!! */
DEFINE_HASHTABLE(global_hashtbl, 10);
DEFINE_SPINLOCK(hashtbl_lock);

irqreturn_t irq_handler_1(int irq, void *dev_id)
{
      /* Interrupt handler running in interrupt context */
      spin_lock(&hashtbl_lock);
      /* access global_hashtbl */
      spin_unlock(&hashtbl_lock);
}

irqreturn_t irq_handler_2(int irq, void *dev_id)
{
      /* Interrupt handler running in interrupt context */
      spin_lock(&hashtbl_lock);
      /* access global hashtable */

      spin_unlock(&hashtbl_lock);
}
```

Spinlocks in interrupt handlers

Spin locks do not sleep so it is safe to use them in interrupt context

If **a lock** is used in **an interrupt handler**, you must also **disable local interrupts before obtaining the lock**.

- Otherwise, it is possible for an interrupt handler to interrupt kernel code while the lock is held and attempt to reacquire the lock.
- The interrupt handler spins, waiting for the lock to become available. The lock holder, however, does not run until the interrupt handler completes. → **double-acquire deadlock**

Spinlocks in interrupt handlers

Conditional enabling/disabling local interrupt

```
DEFINE_SPINLOCK(mr_lock);
unsigned long flags;

/* Saves the current state of interrupts, disables them locally, and
then obtains the given lock */
spin_lock_irqsave(&mr_lock, flags);

/* critical region ... */

/* Unlocks the given lock and returns interrupts to their previous state */
spin_unlock_irqrestore(&mr_lock, flags);
```

Quiz #1: find a deadlock (How to fix?)

```
/* WARNING!!! THIS CODE HAS A DEADLOCK!!! WARNING!!! */
DEFINE_HASHTABLE(global_hashtbl, 10);
DEFINE_SPINLOCK(hashtbl_lock);

irqreturn_t irq_handler(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock(&hashtbl_lock);
    /* access global_hashtbl */
    spin_unlock(&hashtbl_lock);
}

int foo(void)
{
    /* A function running in process context */
    spin_lock(&hashtbl_lock);
    /* What happen if an interrupt occurs
     * while a task executing here? -> Deadlock */
    spin_unlock(&hashtbl_lock);
}
```

Bug fix for Quiz #1

```c
/* NOTE: BUG-FIXED VERSION OF USAGE #1 */
DEFINE_HASHTABLE(global_hashtbl, 10);
DEFINE_SPINLOCK(hashtbl_lock);

irqreturn_t irq_handler(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock(&hashtbl_lock);
    /* It is okay not to disable interrupt here
     * because this is the only interrupt handler access
     * the shared data and this particular interrupt is
     * already disabled. */
    spin_unlock(&hashtbl_lock);
}

int foo(void)
{
    /* A function running in process context */
    unsigned long flags;
    spin_lock_irqsave(&hashtbl_lock, flags);
    /* Interrupt is disabled here */
    spin_unlock_irqrestor(&hashtbl_lock, flags);
}
```

Spinlocks in interrupt handlers

Unconditional enabling/disabling local interrupt

- there is no need to restore previous interrupt’s state

```
DEFINE_SPINLOCK(mr_lock);

/* Disable local interrupt and acquire lock */
spin_lock_irq(&mr_lock);

/* critical section ... */

/* Unlocks the given lock and enable local interrupt */
spin_unlock_irq(&mr_lock);
```

Quiz #2: find a deadlock (How to fix?)

```c
/* WARNING!!! THIS CODE HAS A DEADLOCK!!! WARNING!!! */
DEFINE_HASHTABLE(global_hashtbl, 10);
DEFINE_SPINLOCK(hashtbl_lock);

irqreturn_t irq_handler_1(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock(&hashtbl_lock);
    /* access global_hashtbl */
    spin_unlock(&hashtbl_lock);
}

irqreturn_t irq_handler_2(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock(&hashtbl_lock);
    /* What happen if an interrupt 1 occurs
     * while executing here? -> Deadlock */
    spin_unlock(&hashtbl_lock);
}
```

Bug fix for Quiz #2

```c
/* NOTE: BUG-FIXED VERSION OF USAGE #2 */
DEFINE_HASHTABLE(global_hashtbl, 10);
DEFINE_SPINLOCK(hashtbl_lock);

irqreturn_t irq_handler_1(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock_irq(&hashtbl_lock);
    /* Need to disable interrupt here
     * to prevent irq_handler_2 from accessing the shared data */
    spin_unlock_irq(&hashtbl_lock);
}

irqreturn_t irq_handler_2(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock_irq(&hashtbl_lock);
    /* Need to disable interrupt here
     * to prevent irq_handler_1 from accessing the shared data */
    spin_unlock_irq(&hashtbl_lock);
}
```

Spinlock APIs

Table 10.4 **Spin Lock 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

Spinlock usage example

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/spinlock.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#define PRINT_PREF "[SYNC_SPINLOCK] "

unsigned int counter;    /* shared data: */
DEFINE_SPINLOCK(counter_lock);
struct task_struct *read_thread, *write_thread;

static int writer_function(void *data)
{
    while(!kthread_should_stop()) {
        spin_lock(&counter_lock);
        counter++;
        spin_unlock(&counter_lock);
        msleep(500);
    }
    do_exit(0);
}
```

Spinlock usage example

```
static int read_function(void *data)
{
    while(!kthread_should_stop()) {
        spin_lock(&counter_lock);
        printk(PRINT_PREF "counter: %d\n", counter);
        spin_unlock(&counter_lock);
        msleep(500);
    }
    do_exit(0);
}

static int __init my_mod_init(void)
{
    printk(PRINT_PREF "Entering module.\n");
    counter = 0;

    read_thread = kthread_run(read_function, NULL, "read-thread");
    write_thread = kthread_run(writer_function, NULL, "write-thread");

    return 0;
}
```

Causes of concurrency

Symmetrical multiprocessing (true concurrency)

- Two or more processors can execute kernel code at the same time.

Kernel preemption (pseudo-concurrency)

- Because the kernel is preemptive, one task in the kernel can preempt another.
- Two things do not actually happen at the same time but interleave with each other such that they might as well.

Causes of concurrency

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.

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.

Concurrency safety

SMP-safe

- Code that is safe from concurrency on symmetrical multiprocessing machines

Preemption-safe

- Code that is safe from concurrency with kernel preemption

Interrupt-safe

- Code that is safe from concurrent access from an interrupt handler

What to protect?

Protect the data not code!

Checklist for locking

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

Further readings

Wikipedia: Moore's Law

Wikipedia: Amdahl's Law

Next lecture

Semaphore, mutex

Sequential lock (seqlock)

Completion variable

RCU