Interrupts, Spin Locks, and Preemption

W4118 Operating Systems I

columbia-os.github.io

Credits to Jae

Interrupts

- Hardware interrupts
  - asynchronous
  - e.g. network packet arrival, timer, key press, mouse click
- Exceptions/Faults
  - synchronous
  - e.g. dividing by zero, page fault
- Software interrupts
  - synchronous
  - x86 assembly int: raise software interrupt
  - e.g. syscall (int 0x80), debugger

Kernel Execution: Process Context

- System calls execute kernel code on behalf of a process

- Operations may sleep:
  - Sleeping requires the associated **`task_struct`** to be placed on a wait queue and have **`schedule()`** called to switch to another task

- One kernel stack for each process

Kernel Execution: Interrupt Context

- Interrupt handlers run in interrupt

- Operations cannot sleep – execution does not have an associated task and therefore can't interact with the wait queue and **`schedule()`**
  - e.g. **`kmalloc()`**,**`copy_to/from_user()`** may trigger I/O which causes the caller to sleep until the I/O is satisfied. Can't be called from interrupt context

- All handlers share one interrupt stack per processor:
  - i.e., not the kernel stack of the interrupted task

Interrupt Handling

Key Idea: Defer most work for later

- Only time-critical work should be dealt with in the handler so that we can return to the interrupted task ASAP:
  - Called the ***"top half"*** or sometime the FLIH (First Level Interrupt Handler)
- Push remainder of the work to later
  - Called the ***"bottom half"*** or SLIH (Second Level Interrupt Handler)
  - Several kernel mechanisms available to execute some work at a later time (e.g., softirqs, tasklets, kernel threads)
- Single interrupt will not nest, so handler need not be reentrant
  - … but the handler can be interrupted by a different interrupt

Interrupt Handling Examples

Network Packet Arrival

- **Bottom Half:** acknowledge packet arrival, move packets from NIC to memory, prepare device for further packet arrival
- **Top Half:** propagate packets through kernel networking stack, e.g., TCP/IP processing

Real Time Clock Interrupt (older, simple version)

Requirement for Synchronization

- All data that is accessed in a concurrent manner must be protected
- Users expect a consistent view of the data while accessing it
- This could be
  - a single data item (e.g. integer)
  - a larger data structure (e.g. struct bla_bla)
  - a complex (non-contig) data structure

Critical section

- Critical section: a segment of code that accesses a shared resource
- No more than one thread in critical section at a time

```
// ++ balance
mov    0x8049780,%eax
add    $0x1,%eax
mov    %eax,0x8049780
...
```

```
// -- balance
mov    0x8049780,%eax
sub    $0x1,%eax
mov    %eax,0x8049780
...
```

Mutual Exclusion

- **semaphore**
- **pthread_mutex**

```
pthread_mutex_lock(&balance_lock);
++balance;
pthread_mutex_unlock(&balance_lock);
```

These are **sleeping** locks. The calling task is put to sleep while it waits for the critical section to become available.

Is this always a good idea when waiting?

Spin Lock

Instead of sleeping until the critical section is free, spin locks poll the critical section until it is free.

High-level idea

`lock()` polls until `flag == 0`

then sets `flag == 1`

`unlock()` sets `flag == 0`

```
int flag = 0;

lock() {
    while (flag == 1)
        ;

    flag = 1;
}

unlock() {
    flag = 0;
}
```

Any issues?

Spin Lock: Race Condition

Task 1

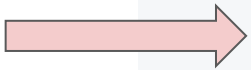
```
int flag = 0;

lock() {
    while (flag == 1)
        ;

    flag = 1;
}

unlock() {
    flag = 0;
}
```

Task 2

```
int flag = 0;

lock() {
    while (flag == 1)
        ;

    flag = 1;
}

unlock() {
    flag = 0;
}
```

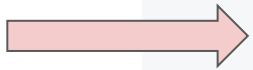
Spin Lock: Race Condition

Task 1



```
int flag = 0;

lock() {
    while (flag == 1)
        ;

    flag = 1;
}

unlock() {
    flag = 0;
}
```

Task 2



```
int flag = 0;

lock() {
    while (flag == 1)
        ;

    flag = 1;
}

unlock() {
    flag = 0;
}
```

Spin Lock

Instead of sleeping until the critical section is free, spin locks poll the critical section until it is free.

High-level idea

`lock()` polls until `flag == 0`

then sets `flag == 1`

`unlock()` sets `flag == 0`

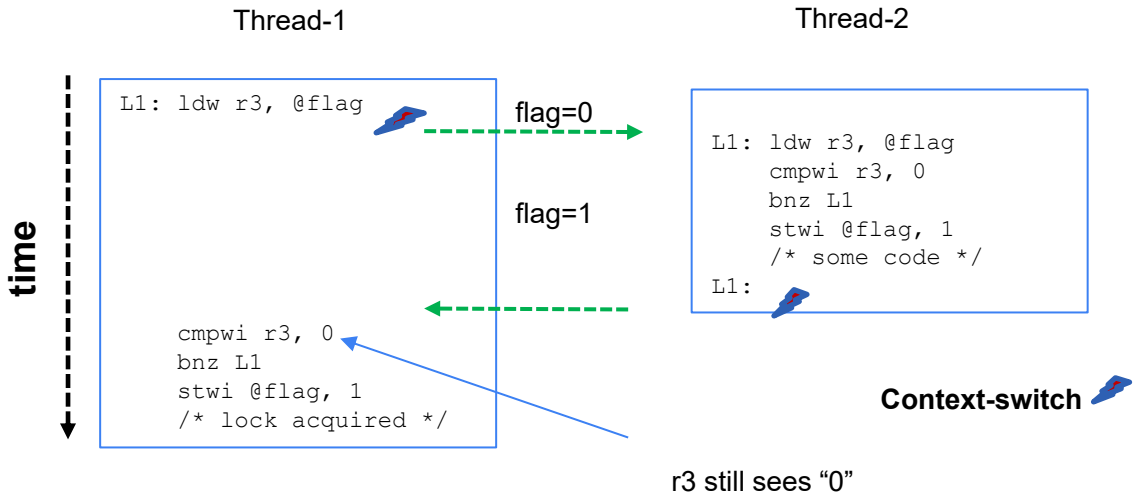
Non-atomic test & set leads to mutual exclusion violation

```
int flag = 0;

lock() {
    while (flag == 1)
        ;

    // This gap between testing and setting the variable
    // creates a race condition!

    flag = 1;
}

unlock() {
    flag = 0;
}
```

Read-Modify-Write Cycles During Locks

- Using a simple Lock Variable <flag>
  we have to go down to the assembler/instruction level to investigate

```
int flag = 0;

lock() {
    while (flag == 1)
        ;

    flag = 1;
}

unlock() {
    flag = 0;
}
```



- Both Threads assume now they hold the lock and proceed

Spin Lock

Instead of sleeping until the critical section is free, spin locks poll the critical section until it is free.

High-level idea

`lock()` polls until `flag == 0`

then sets `flag == 1`

`unlock()` sets `flag == 0`

Correct implementation needs atomic test_and_set hardware instruction

```
int flag = 0;

lock() {
    while(test_and_set(&flag))
        ;
}

unlock() {
    flag = 0;
}
```

Atomic Test and Set

In C pseudocode, **`test_and_set`** hardware instruction looks like:

```
int test_and_set(int *lock) {
    int old = *lock;
    *lock = 1;
    return old;
}
```

Linux Kernel Spin Locks I

- **`spin_lock()` / `spin_unlock()`**
  - keep the critical sections as small as possible
  - must not lose CPU while holding a spin lock
    - other threads will wait for the lock for a long time
  - must NOT call any function that can potentially sleep
    - e.g., **`kmalloc()`**, **`copy_from_user()`**
  - **`spin_lock()`** prevents kernel preemption by **`++preempt_count`**
    - in a uniprocessor, that's all **`spin_lock()`** does
  - hardware interrupt is ok unless the interrupt handler may try to lock this spin lock
    - spin lock is not recursive: same thread locking twice will deadlock

Linux Kernel Spin Locks II

- **`spin_lock_irqsave()` / `spin_unlock_irqrestore()`**
  - save current interrupt state, disable all interrupts on local CPU, lock, unlock, restore interrupts to how they were before
  - need to use this version if the lock is something that an interrupt handler may try to acquire
  - no need to worry about interrupts on other CPUs – spin lock will work normally
  - no need to spin in uniprocessor – just **`++preempt_count`** & disable irq

- **`spin_lock_irq()` / `spin_unlock_irq()`**
  - disable & enable irq assuming it was enabled to begin with
  - should not be used in most cases

When to use?

- **`spin_lock()` / `spin_unlock()`**
  - Used when code never needs to protect against interrupt-level concurrency on the same CPU, e.g.:
    - the lock is only used by process context
    - the lock is only used by a **single** interrupt (this holds because a single interrupt does not nest)

- **`spin_lock_irqsave()` / `spin_unlock_irqrestore()`**
  - Used when an interrupt handler on the same CPU can attempt to acquire the lock while it is held, e.g.:
    - the lock is used by process and interrupt contexts
    - the lock is used by more than one interrupts

Spinning vs. Sleeping Lock

- Sleeping lock incurs cost of context-switch to put caller to sleep
- Spinning lock consumes CPU time by polling
- In interrupt context can only use spin locks
- Can’t sleep while holding spin lock

- Will cover sleeping locks in next set.

Preemption

Sometimes the kernel needs to forcefully reclaim the CPU. It track a per-process **`TIF_NEED_RESCHED`** flag. If set, preemption occurs by calling **`schedule()`** in the following cases:

1. Returning to user space:
    a. from a system call
    b. from an interrupt handler
2. Returning to kernel from an interrupt handler, only if **`preempt_count`** is zero
3. **`preempt_count`** just became zero, right after **`spin_unlock()`**, for example
4. Task running in kernel mode calls **`schedule()`** itself – e.g., blocking syscall