Kernel Synchronization
with material from
Understanding the Linux Kernel (O'Reilly)

Synchronization In The Kernel

- Past lectures:
  - Synchronization constructs

- Today’s lecture:
  - What does this stuff look like in an OS?
  - We have source code for Linux...
  - We mostly run on x86 platforms
  - Let’s look at some specifics.
Disabling Interrupts

Key observations:
- On a uni-processor, an operation is atomic if no context-switch is allowed in the middle of the operation.
- Context switch occurs because of:
  - Internal events: system calls and exceptions
  - External events: interrupts
- Mutual exclusion can be achieved by preventing context switch.

Prevention of context switch:
- Eliminate internal events: easy (under program control)
- Eliminate external events: disable interrupts
  - Hardware delays the processing of interrupts until interrupts are enabled.

Lock Implementation: A Naïve Solution

```cpp
Lock::Acquire() { disable interrupts; }
Lock::Release() { enable interrupts; }
```

Will this work on a uni-processor?

What is wrong with this solution?
- Once interrupts are disabled, the thread can’t be stopped ➔ Can starve other threads
- Critical sections can be arbitrarily long ➔ Can’t bound the amount of time needed to respond to interrupts

But this is used all over the place in uniprocessor OSes to do short tasks. It is a large source of bugs. What would typical failure conditions be?
Disabling Interrupts in Linux

- Usually some small number of interrupt levels, statically assigned (e.g., reset = 0, timer = 1, network = 3, disk = 4, software = 7)
  - When you “disable interrupts” you disable them for your level and higher.
  - When you reenable interrupts, you need to do so at the previous level.
  - Where do you store the level in the meantime?
    - A. Local variable
    - B. Global variable
    - C. Hardware register

```c
unsigned long flags;
llocal_irq_save( flags ); // Disable & save
do_whatever;
llocal_irq_restore( flags ); // Reenable
```

Using Locks Correctly

- Make sure to release your locks along every possible execution path.

```c
unsigned long flags;
llocal_irq_save( flags ); // Disable & save
...
if(somethingBad) {
    local_irq_restore( flags );
    return ERROR_BAD_THING;
}
...
llocal_irq_restore( flags ); // Reenable
return 0;
```
**Entering Linux**

- An OS is a server that responds to requests, from user code and from devices.

- How to enter the Linux kernel
  - int 0x80, which is the system call instruction on the x86 by convention.
  - External device sends a signal to a programmable interrupt controller (PIC) by using an IRQ (interrupt request) line, and that interrupt is enabled.
  - A process in kernel mode causes a page fault.
  - A process in a multi-processor system executing in kernel mode raises an inter-processor interrupt.

- Interrupt, exception, or softirq handling can interrupt a process running in kernel mode, but when the handler terminates, the kernel control path of the process resumes.

**Linux Synchronization Primitives**

<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic Operation</td>
<td>Atomic read-modify-write instruction</td>
<td>All CPUs</td>
</tr>
<tr>
<td>Memory barrier</td>
<td>Avoid instruction re-ordering</td>
<td>Local CPU</td>
</tr>
<tr>
<td>Spin lock</td>
<td>Lock with busy wait (readers/writers spin locks)</td>
<td>All CPUs</td>
</tr>
<tr>
<td>Semaphore</td>
<td>Lock with blocking wait (R/W semaphores)</td>
<td>All CPUs</td>
</tr>
<tr>
<td>Local interrupt disable</td>
<td>Forbid interrupt handling on single CPU</td>
<td>Local CPU</td>
</tr>
<tr>
<td>Local softirq disable</td>
<td>Forbid deferrable function handling on a single CPU</td>
<td>Local CPU</td>
</tr>
<tr>
<td>Global interrupt disable</td>
<td>Forbid interrupt and softirq handling on all CPUs</td>
<td>All CPUs</td>
</tr>
</tbody>
</table>
Scope of Synch Primitives

- The test&set instruction is local to a single CPU in a multi-CPU system.
  - A. True
  - B. False

Atomic Operations

- Assembly language instructions that make 0 or 1 aligned memory access (a 32-bit aligned access has the last 2 address bits equal to zero).
- Read-modify-write instructions (like inc and dec) with the lock prefix. Locks the memory bus.
- Linux provides wrappers for these operations.
  - \texttt{atomic\_set(v,i)} sets \texttt{*v=i}
  - \texttt{atomic\_inc\_and\_test(v)} Add 1 to \texttt{*v} and return 1 if value is now zero, 0 otherwise.
Memory Barriers

- Compiler reorders your memory accesses.
- Memory barrier says, “wait here until all outstanding memory operations have completed.”
- `rmb()` expands to
  
  ```
  asm volatile("lock;addl $0,0(%esp):=":"memory");
  ```
  
  - `volatile` – disables compiler reorder of instruction
  - `memory` – forces compiler to assume any RAM can change from this instruction.
  - `lock` prefix locks memory bus, and requires all previous reads to complete.

- Example use
  
  ```
  new->next = list_element->next;
  wmb();
  list_element->next = new;
  ```

Spin Locks

- CPU “spins,” executing instructions waiting for lock to free.
  - Only useful on multi-processors
  - Insures only one kernel thread runs a routine at a time
- Value of 1 means lock is free
- `spin_lock(spinlock_t slp) {`
  
  ```
  1: lock; decb slp
     jns 3f
  2: cmpb $0, slp
     pause // p4-reduces power
     jle 2b // back compat rep;nop
     jmp 1b
  3:
  ```
R/W Spin Locks – space optimized data structure

- Many locks in kernel, so they should be small.
  - Interpret bit ranges.

<table>
<thead>
<tr>
<th>Unlocked flag</th>
<th>2’s complement reader count</th>
</tr>
</thead>
<tbody>
<tr>
<td>24b 23b</td>
<td>0b</td>
</tr>
</tbody>
</table>
- 0x01000000 - Idle, not locked and no readers
- 0x00000000 - Acquired for writing, no readers
- 0x00FFFFFFE - Acquired by 2 readers
- \textbf{read\_lock()} = lock; subl $1, (rwlp)\n jns 1f
- \textbf{read\_unlock()} = lock; incl rwlp
- \textbf{rw\_lock()} = lock; subl %0x01000000, (rwlp)\n jz 1f
- \textbf{rw\_unlock()} = lock; addl $0x01000000, rwlp

Semaphores

- Kernel semaphores suspend a waiting process, so can’t be called from interrupt handlers and deferrable functions.
  - atomic\_t count; // 1=available 0=busy -1=one waiting
  - wait\_queue; // wait queue list
  - int sleepers; //flag, 1 if sleepers, optimization
- \textbf{down()} – acquire \textbf{up()} – release
- \textbf{down\_interruptable()} 
  - Used by device drivers. Suspend me, but if I get a signal, take me off the wait queue & return error.
- Read/write semaphores. Allows multiple readers.
  - Queues waiters, so it is fair to writers.
- Semaphore implementation only locks when manipulating the sleep queue.
### Linux Use of Semaphores

- Linux uses semaphores in which processing path (primarily)?
  - A. Interrupt Handlers
  - B. System call service routines
  - C. Deferrable kernel functions
  - D. Device drivers
  - E. Kernel threads

### Completions

**Solving race condition with temporary Semaphores**

<table>
<thead>
<tr>
<th>Thread A, CPU 0</th>
<th>Thread B, CPU 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alloc semaphore S</td>
<td>Accept sema S</td>
</tr>
<tr>
<td>Pass sema to B</td>
<td>down(S)</td>
</tr>
<tr>
<td>free(S)</td>
<td>up(S)</td>
</tr>
</tbody>
</table>

- up(S) and free(S) execute concurrently.
  - lock only protects sleep queue, not whole semaphore
- Lock protects entire Completion.
  - Slower, but safer.
Local Interrupt Disable

- `local_irq_disable()/local_irq_enable()`
  - Disables and reenables interrupts
  -Executes cli/sti on x86
- But interrupts can be nested, so what interrupt level to we return to?
  - unsigned long flags;
  - `local_irq_disable(flags);`
  - …Read or update a data structure…
  - `local_irq_enable(flags);`
- Disable clears the IF flag of the eflags register, and saves register in variable. Enable restores previous register value.

Uses of Synchronization in the Kernel

- Short Kernel Critical Sections
  - Disable interrupts (& grab spin lock on MPs)
- Long Critical Sections
  - Separated into “top” and “bottom”
  - top - disable interrupts (& grab spin lock on MPs)
  - bottom – bottom halves don’t interrupt each other, so no sync needed on UP. On MP uses a lock to protect data structures from concurrent access.
- Interrupt Protection levels
  - top-half interrupt handlers
  - bottom-half interrupt handlers
  - kernel-system service routines
  - user-mode programs (preemptible)