Linux Internals
Day 4
The Linux Kernel

Joel Isaacson
Ascender Technologies Ltd

Critical Regions

- Even simple operation like i++ are not immune to data corruption.
- These effects are at times very subtle, difficult, if not impossible, to duplicate.
- This is what make kernel programming so expensive per line.

Race Conditions

- A typical case is:
  - if(cond == 0) do_a(); else do_b();
- Not trivial to fix this problem in all cases.

Locking

```
get_lock();
access_queue();
put_lock();
```

```
get_lock();
failed;
waiting;
succeeded;
access_queue();
put_lock();
```

Causes of Concurrency

- Interrupts.
- Softirqs and tasklets.
- Kernel preemption.
- Blocking and sleeping.
- SMP

What needs Protections

- Let's first figure out what doesn't need protection. It's simpler.

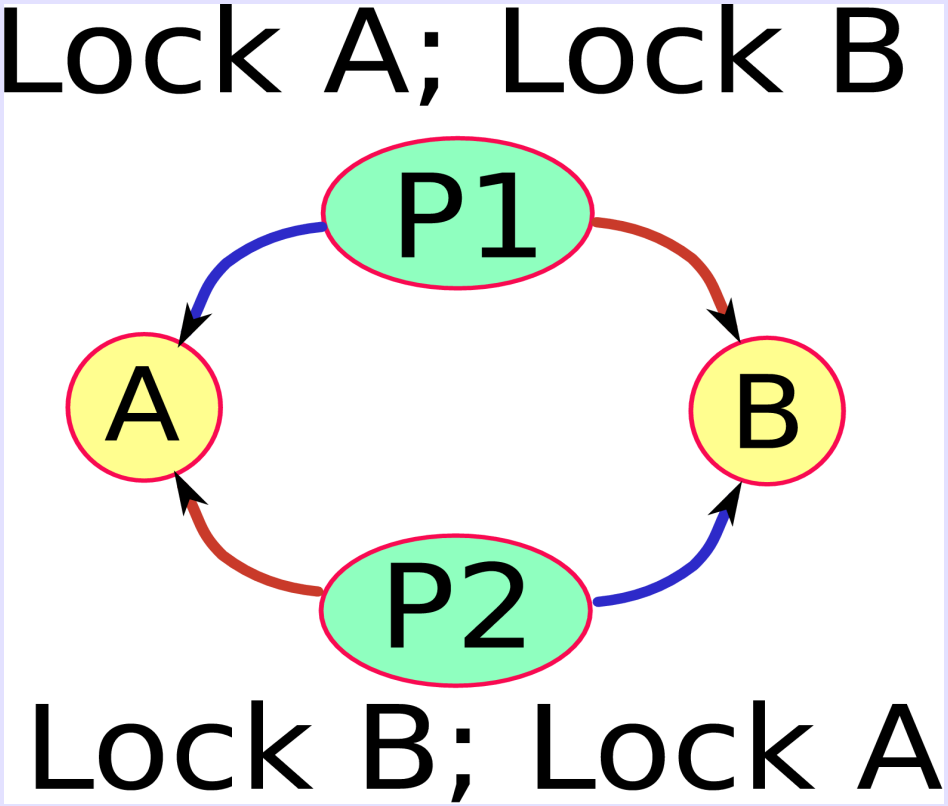
What doesn't need Protections

- Local data.
- Data that can be accessed by one task only (single threaded user applications).
- Private data in non-preemptive non-concurrent applications (tasklets).
- Private per CPU data in non-preemptive concurrent applications (softirqs).

What needs Protections?

- Just about everything else has to be examined carefully.

Deadlock



Deadlock Prevention

- There are a number of standard ideas about preventing deadlock:
  - Try not acquire more than one lock at a time.
  - If you do acquire more than one lock at a time, use a consistent acquisition order.
  - Prevent starvation. Don't acquire locks for unlimited time.
  - Do not reacquire locked locks. Linux doesn't have recursive locks.

Contention and Scalability

- There is a trade off between simplicity and efficiency in designing locking protocols.
- Using few locks, each one for many objects, create highly contended locks, that frequently are unacquirable.
- Linux 2.0 introduced SMP support with just one lock (the Big Kernel Lock).

Contention and Scalability

- The granularity of a lock is a measure of how much data it protects:
  - A coarse grained lock protects many data accesses.
  - A fine grained lock protects few data acceses.
- As Linux developed (2.0 -> 2.2 -> 2.4 -> 2.6) locking has become more and more fined-grained and thus more scalable.

Contention and Scalability

- When we looked at the O(1) scheduler you might have noticed that each CPU has a priority queue that is protected by a per-CPU lock and not one scheduler lock.
- Also notice that when we acquire two runqueue locks we use their addresses to decide the order the locks are acquired.

Contention and Scalability

- The attempt to add a lock to every object increases complexity and inflates code size for no good reason.
- Reason should rule.

Synchronization Methods

- We will examine various kernel synchronization methods:
  - Atomic operations
  - Spin locks
  - Reader-writer spin locks
  - Semaphores

Synchronization Methods

- We will examine various kernel synchronization methods:
  - Reader-writer semaphores
  - Completion variables
  - Preemption disabling
  - Memory barriers

Atomic Operations

- Simple operations can sometimes be performed without complicated locking.
- As we have mentioned even a simple operation such as “i++” may not be indivisible (atomic).
- The “atomic operations” are typically either #defines or inline functions.

Atomic Operations

- `atomic_t v;`
- `atomic_t v= ATOMIC_INIT(3);`
- `atomic_read(v);`
- `atomic_set(&v,5);`
- `atomic_add(1,&v);`
- `atomic_sub(1,&v);`
- `atomic_inc(&v);`
- `atomic_dec(&v);`
- `atomic_sub_and_test(i,&v);`
- `atomic_add_negative(i,&v);`
- `atomic_dec_and_test(&v);`
- `atomic_inc_and_test(&v);`
-

Atomic Bitwise Operations

- `void set_bit(int n, void *addr);`
- `void clear_bit(int n, void *addr);`
- `void change_bit(int n, void *addr);`
- `int test_and_set_bit(int n, void *addr);`
- `int test_and_clear_bit(int n, void *addr);`
- `int test_and_change_bit(int n, void *addr);`
- `int test_bit(int n, void *addr);`
- `int find_first_bit(unsigned long *addr, int n);`
- `int find_first_zero_bit(unsigned long *addr, int n);`

Spin Locks

- More complicated data structure need more complex systems for guaranteeing data integrity.
- Spin locks are the next in order of complexity and CPU utilization.
- A spin lock can be held by at most one thread of execution.

Spin Locks

- A contended spin lock causes threads to “spin”, essentially wasting processor time, until the owner of the lock unlocks it.
- This waste of CPU cycles means that any spin lock should be acquire for only short periods of time.

Spin Locks

- The duration of time the lock is held should be less than the two context switch times that put the process to sleep and wake it up.
- Otherwise other synchronization primitives (semaphores, process sleeping) should be used.

Spin Locks

- Spin locks are normally implemented in assembly language.
- Spin locks are not needed for single CPU processors. In fact for uniprocessor Linux they compile away and are nullified.
- Carefully implementation is needed to produce a SMP friendly spin lock.

Spin Locks

- Linux spin locks are non-recursive (non-counting).
- Thus if you acquire a spin lock and then attempt to acquire it again you will normally deadlock.
- Spin locks can be used in situations where semaphores can't. For example: interrupts handlers.

Spin Locks

- In many situations we need to combine spin locks with local interrupt masking.
- Why:
  - We acquire a spin lock.
  - An interrupt occurs before the lock is freed.
  - The interrupt handler tried to acquire the lock.
  - **Deadlock!**

Spin Locks and Interrupt Disable

- The kernel provides an interface that:
  - acquire the spin lock
    - and then
  - disable interrupts
- Or is that:
  - disable interrupts
    - and then
  - acquire the spin lock

Spin Locks and Interrupt Disable

- Of course that is:
  - disable interrupts
    - and then
  - acquire the spin lock
- Otherwise we have a race condition.
- The interface is called:
  - spin_lock_irqsave(&lock, flags);
  - spin_unlock_irqrestore(&lock, flags);

Spin Locks Functions

- `spin_lock()`
- `spin_lock_irq()`
- `spin_lock_irqsave()`
- `spin_unlock()`
- `spin_unlock_irq()`
- `spin_unlock_irqrestore()`
- `spin_lock_init()`
- `spin_trylock()`
- `spin_is_locked()`
- `spin_lock_bh()`
- `spin_unlock_bh()`

Reader/Writer Spin Locks

- If we have many readers of the data structure but infrequent writers.
- Many threads can simultaneously acquire read_locks but only one can acquire a write_lock.
- Acquiring a write_lock will require all readers to release their locks.
- Writers might suffer starvation.

Semaphores

- Semaphores are locks that cause a thread to sleep if they can't be acquired.
- They can't be used in interrupt handlers.
- They shouldn't be used for very short locking periods.
- The Linux implementation is a “counting” semaphore rather than a “binary” semaphore.

Semaphores

- You can sleep while holding a semaphore. Eventually you should wake up.
- You cannot sleep while holding a spin lock.
- You therefore cannot acquire a semaphore while holding a spin lock.

Using Semaphores

- sema_init(&sema, i);
- up(&sema);
- down(&sema);
- down_interruptable(&sema);
- down_trylock(&sema);

Reader/Writer Semaphores

- A similar idea to the reader/writer spin lock.
- Binary semaphores only (mutexes)
- Allows multiple readers and single writers.
- Write starvation a possibility.

Completion Variables

- This is a simple wait/signal scheme.
- Usage:
  - init_completion(struct completion *);
  - wait_for_completion(struct completion *);
  - complete(struct completion *);

Preemption Disabling

- Acquiring a spin lock will disable preemption.
- Sometime (if we are only accessing per-CPU variables) it is sufficient just to disable preemption.
- Usage:
  - preempt_disable();
  - preempt_enable();

Memory Barriers

- A computer's architecture may optimize access to memory resources.
- This is especially true of multiprocessors
- Read and especially writes may be delayed or reordered.
- Memory barriers are used to assure memory write ordering and consistency.

Memory Barriers

- It is difficult for people who use Intel computers to understand the strange memory consistency models that other computers might use.
- Linux kernel programmers are expected to write code for the lowest common denominator in computer architecture.

Memory Management

- Memory management in the kernel environment is quite different and more difficult than in user mode.
- Memory allocations possibly require swapping out pages that might cause the requesting thread to block.
- Of course you can not block in all kernel code.

Pages

- The basic building blocks of the memory system is the page.
- The page is typically 4 Kb or possibly 8 Kb.
- We must differentiate between
  - Physical addresses
  - Virtual addresses

Pages

- The memory management unit (MMU) is responsible for mapping physical pages to virtual pages.
- Each page has an integer page number. It is just the base page address divided by the page size.

Zones

- When Linux boots we have to describe the size and structure of memory.
- We divide the physical memory into zone that might have different properties
  - ZONE_DMA < 16MB
  - ZONE_NORMAL 16-896MB
  - ZONE_HIGMEM > 896 MB

Allocating Pages

- The mechanism for allocating a contiguous series of physical pages is the:
  - struct page *alloc_pages(gfp_mask, order);
  - struct page *alloc_page(gfp_mask);

Allocating Pages

- The gfp_mask parameter:
  - __GFP_WAIT
  - __GFP_HIGH
  - __GFP_DMA
  - __GFP_HIGHMEM
  - __GFP_NOFAIL

kmalloc

- The kmalloc routine issimilar to the malloc routine
- kmalloc is like alloc_pages but it can allocate less than a page size.
  - void *kmalloc(unsigned long size, int flags);

vmalloc

- The vmalloc routine allocates memory that isn't necessarily physically and virtually contiguous but it is only virtually contiguous.
  - void *vmalloc(unsigned long size);

The Slab Allocator

- The kernel allocates and frees data frequently
- The slab layer is used to make these operations fast and efficient.
- When a data structure is freed it is places on a free list cache.
- The next allocation will remove the data structure from the free list.

The Slab Allocator

- The slab layer is data type object oriented.
- The structure of the slab layer is:
  - Cache --- For each data type
  - Slab --- A block of memory pages.
  - Object --- The data structure

The Slab Allocator

- Usage:
  - kmem_cache_t *kmem_cache_create(name, size, align, flags, ctor, dtor)
  - void *kmem_cache_alloc(kmem_cache_t *cp, int flags);
  - void kmem_cache_free(kmem_cache_t *cp, void *objp);

Per-CPU Allocations

- Many SMP friendly algorithms use arrays of data that are per-CPU.
- This allows no-lock access to the per-CPU data items:

```
int my_data[NR_CPUS];
int cpu;
cpu= get_cpu(); // get cpu and disable preemption
my_data[cpu]++;
put_cpu();          // put cpu
```

Percpu Interface

- DEFINE_PER_CPU(type, name);
- get_cpu_var(name);
- put_cpu_var(name);
- per_cpu(name);

Virtual Filesystem

- The virtual filesystem is the subsystem that maintains a consistent filesystem user interface over all the various filesystems that Linux supports.
- This idea comes from Bill Joy at Sun Microsystems over 20 years ago.
- It is taken for granted these days.

Virtual Filesystem

- The VFS has the following abstractions:
  - Superblock - A mounted filesystem
  - Dentry - A directory object
  - Inode - A file object
  - File – An open file object
- Each supported file system has initialized structures supporting these abstractions.

Process Address Space

- Linux presents a virtual memory operating system to the user.
- The user has a “flat” address space of 3 Gb into which he can map “memory areas” with mmap().
- The total mapped memory areas is the “memory space” of the first days lecture.

Process Address Space

- A process memory area is defined by the mm pointer in the task_struct.
- This pointer is to a structure of type “struct mm_struct”.
- This structure contains a list of “virtual memory areas” (VMA's) which are the memory areas.

Page tables

- Linux uses a three level page table abstraction.
  - Page global directory (pgd)
  - Page middle directory (pmd)
  - Page table (pte)
- Not all architectures support the three level hierarchy.

Page Cache

- The Linux kernel implements a large dynamic cache in order to minimize I/O.
- The I/O page cache mechanism is fully integrated with the process memory allocation mechanisms.
- The principle results of block I/O is the population of the page cache (previously called the buffer cache).

Pdflush Daemon

- Dirty pages are flushed to the disk.
- There are a number of reasons that the data would be flushed.
  - Free memory gets tight.
  - The dirty page ages past some threshold
- The kernel thread(s) takes care of flushing dirty pages to the disk.