第 14 讲: Concurrency in OS Kernel

第一节：Introduction

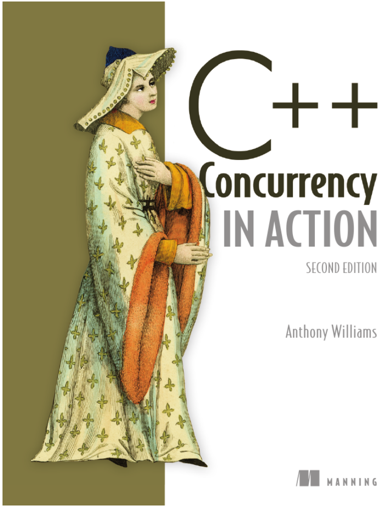
陈渝

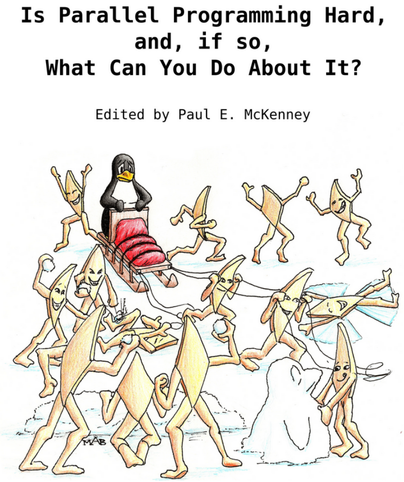
清华大学计算机系

yuchen@tsinghua.edu.cn

2020 年 5 月 18 日

Resource





Reference:
"Is Parallel Programming Hard, And, If So, What Can You Do About It?",Paul McKenney;
"C++Concurrency in Action", ANTHONY WILLIAMS;
"CS510 - Advanced Topics in Concurrency", Jonathan Walpole

- Why do we need locking in the kernel?
- Which problems are we trying to solve?
- What implementation choices do we have?
- Is there a one-size-fits-all solution?

Concurrency in Linux

- Linux is a symmetric multiprocessing (SMP) preemptible kernel
- Its has true concurrency
- And various forms of pseudo concurrency

Software-based preemption

- Voluntary preemption (sleep/yield)
- Involuntary preemption (preemptable kernel)
- And various forms of pseudo concurrency
- Solutions: don't do the former, disable preemption to prevent the latter

Hardware preemption

- Interrupt/trap/fault/exception handlers can start executing at any time
- Solutions: disable interrupts

Uniprocessor Example

```
preempt disable;
    r1 = x;
    r2 = x;
preempt enable;
assert (r1 == r2)
```

Uniprocessor Example

```
preempt disable;
    r1 = x;
    yield();
    r2 = x;
preempt enable;
assert (r1 == r2)
```

Uniprocessor Example

```
interrupt disable;
    r1 = x;
    r2 = x;
interrupt enable;
assert (r1 == r2)
```

True Concurrency in Linux

Solutions to pseudo-concurrency do not work in the presence of true concurrency

Alternatives include atomic operators, various forms of locking, RCU, and non-blocking synchronization

Locking can be used to provide mutually exclusive access to critical sections

- Locking can not be used everywhere, i.e., interrupt handlers can't block
- Locking primitives must support coexistence with various solutions for pseudo concurrency, i.e., we need hybrid primitives

Multiprocessor Example

```
interrupt disable;
    r1 = x;
    r2 = x;
interrupt enable;
assert (r1 == r2)
```

Atomic Operators in Linux

Simplest synchronization primitives

- Primitive operations that are indivisible

Two types

- methods that operate on integers
- methods that operate on bits

Implementation

- Assembly language sequences that use the atomic read- modify-write instructions of the underlying CPU architecture

Memory Invariance Example

```
r1 = atomic read x;
r2 = atomic read x;
assert (r1 == r2)
```

Atomic Integer Operators

```
atomic_t v;
atomic_set(&v, 5); /* v = 5 (atomically) */
atomic_add(3, &v); /* v = v + 3 (atomically) */
atomic_dec(&v);    /* v = v - 1 (atomically) */

printf("This will print 7: %d\n", atomic_read(&v));
```

"atomic_add(3,&v);" is NOT the same as "atomic_add(1,&v); atomic_add(2,&v);"

Mutual exclusion for larger (than one operator) critical sections requires additional support

Spin locks are one possibility

- Single holder locks
- When lock is unavailable, the acquiring process keeps trying

Basic Use of Spin Locks

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
spin_lock(&mr_lock);
/* critical section ... */
spin_unlock(&mr_lock);
```

spin_lock()

- Acquires the spinlock using atomic instructions required for SMP

spin_unlock()

- Releases the spinlock

Interrupting a spin lock holder may cause problems

- Spin lock holder is delayed, so is every thread spin waiting for the spin lock
- Not a big problem if interrupt handlers are short
- Interrupt handler may access the data protected by the spin-lock
    - Should the interrupt handler use the lock?
    - Can it be delayed trying to acquire a spin lock?
    - What if the lock is already held by the thread it interrupted?

- Should the interrupt handler use the lock?
- Can it be delayed trying to acquire a spin lock?
- What if the lock is already held by the thread it interrupted?

——————————————

- If data is only accessed in interrupt context and is local to one specific CPU we can use interrupt disabling to synchronize
- If data is accessed from other CPUs we need additional synchronization – spin lock
- Normal code (kernel context) must disable interrupts and acquire spin lock

Spin Locks & Interrupt Disabling

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;
spin_lock_irqsave(&mr_lock, flags); /* critical section ... */
spin_unlock_irqrestore(&mr_lock, flags);
```

spin_lock_irqsave()

- Disables interrupts locally
- Acquires the spinlock using instructions required for SMP

spin_unlock_irqrestore()

- Restores interrupts to the state they were in when the lock was acquired
  - Contention for semaphores causes blocking not spinning
  - Should not be used for short duration critical sections!
  - Can be used to synchronize with user contexts that might block or be preempted

Softirqs, tasklets and BHs are deferrable functions

- delayed interrupt handling work that is scheduled
- they can wait for a spin lock without holding up devices
- they can access non-CPU local data

Softirqs –the basic building block

- statically allocated and non-preemptively scheduled
- can not be interrupted by another softirq on the same CPU
- can run concurrently on different CPUs, and synchronize with each other using spin-locks

Bottom Halves

- built on softirqs
- can not run concurrently on different CPUs

Spin Locks & Deferred Functions

spin_lock_bh()

- Implements the standard spinlock
- Disables softirqs
- Allows the softirq to use non-preemption only

spin_unlock_bh()

- Releases the spinlock
- Enables softirqs

Spin Lock Rules

- Do not try to re-acquire a spinlock you already hold!
- Spinlocks should not be held for a long time!
- Do not sleep while holding a spinlock!

Semaphores

Semaphores are locks that are safe to hold for longer periods of time

- Contention for semaphores causes blocking not spinning
- Should not be used for short duration critical sections!
- Can be used to synchronize with user contexts that might block or be preempted

Implemented as a wait queue and a usage count

- wait queue: list of processes blocking on the semaphore
- usage count: number of concurrently allowed holders

down()

- Attempts to acquire the semaphore by decrementing the usage count and testing if it is negative
- Blocks if usage count is negative

up()

- releases the semaphore by incrementing the usage count and waking up one or more tasks blocked on it

Semaphore Implementation

Implemented as a wait queue and a usage count

- wait queue: list of processes blocking on the semaphore
- usage count: number of concurrently allowed holders

down_interruptible()

- Returns −EINTR if signal received while blocked
- Returns 0 on success

down_trylock()

- Attempts to acquire the semaphore
- On failure it returns nonzero instead of blocking

Reader-Writer Locks

No need to synchronize concurrent readers unless a writer is present

Both spin locks and semaphores have reader/writer variants

Reader-Writer Spin Locks

```
rwlock_t mr_rwlock = RW_LOCK_UNLOCKED;

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

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

Reader-Writer Semaphores

```
struct rw_semaphore mr_rwsem;
init_rwsem(&mr_rwsem);

down_read(&mr_rwsem); /* critical region (read only) ... */
up_read(&mr_rwsem);

down_write(&mr_rwsem); /* critical region (read and write) ... */
up_write(&mr_rwsem);
```

Conclusions

Wow! Why does one system need so many different ways of doing synchronization?

- Actually, there are more ways to do synchronization in Linux, this is just “locking”！

Conclusions

One size does not fit all:

- Need to be aware of different contexts in which code executes (user, kernel, interrupt etc) and the implications this has for whether hardware or software preemption or blocking can occur
- The cost of synchronization is important, particularly its impact on scalability