第二次实例分析
Part 1-自旋锁

组号 6

小组成员 赵云泽 余秋初 杨以灏

XV6自旋锁相关内容分析

一、自旋锁相关数据结构

```
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

核心域：locked　判断锁是否被占用

其他域：name　锁名

cpu　占用该锁的cpu

pcs　记录栈帧信息，用于调试

二、自旋锁相关重要函数

- 自旋锁的初始化

```
void
initlock(struct spinlock *lk, char *name)
{
  lk->name = name;
  lk->locked = 0;
  lk->cpu = 0;
}
```

二、自旋锁相关重要函数

- 获取锁

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

二、自旋锁相关重要函数

- 释放锁

```
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other cores before the lock is released.
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() tells them both not to.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );

  popcli();
}
```

二、pushcli与popcli函数

- pushcli

```
// Pushcli/popcli are like cli/sti except that they are matched:
// it takes two popcli to undo two pushcli.  Also, if interrupts
// are off, then pushcli, popcli leaves them off.

void
pushcli(void)
{
  int eflags;

  eflags = readeflags();
  cli();
  if(mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}
```

二、pushcli与popcli函数

- popcli

```
void
popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

二、pushcli与popcli函数

Q：mycpu()->intena 变量的作用是什么？

A：

```
// Per-CPU state
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
};
```

cpu结构体中intena域：记录pushcli之前的中断使能标志位

cpu结构体中ncli域：　记录pushcli的调用次数（一个CPU可能占有多个锁）

二、pushcli与popcli函数

- pushcli---readeflags

```
static inline uint
readeflags(void)
{
  uint eflags;
  asm volatile("pushfl; popl %0" : "=r" (eflags));
  return eflags;
}
```

- pushcli---cli, popcli---sti

```
static inline void
cli(void)
{
  asm volatile("cli");
}

static inline void
sti(void)
{
  asm volatile("sti");
}
```

二、pushcli与popcli函数

Q：pushcli 函数中的 EFLAGS 和 FL_IF 分别是什么？

A：eflags寄存器，X86系统中的标志寄存器，主要用于提供程序的状态及进行相应的控制

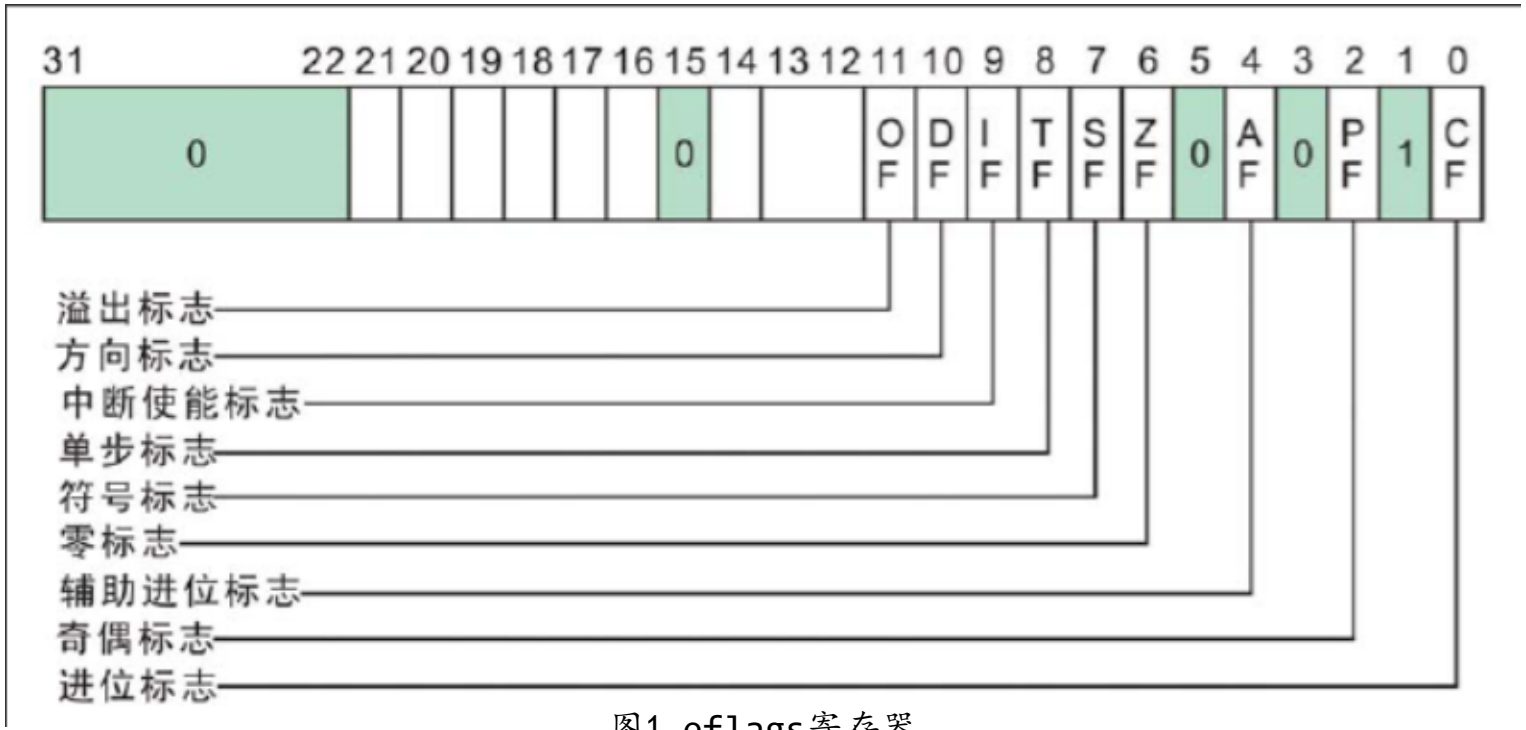


图1 eflags寄存器

Source：https://www.cnblogs.com/Reverse-xiaoyu/p/11397584.html

二、pushcli与popcli函数

Q：pushcli 函数中的 EFLAGS 和 FL_IF 分别是什么？

A：FL_IF是一个宏定义，其位于mmu.h。若将最低位视为第0位，则FL_IF的第九位为1，其余位均为0，通过与运算获得EFLAGS的第九位，即中断使能标志位。

```
// Eflags register
#define FL_IF           0x00000200      // Interrupt Enable
```

二、pushcli与popcli函数

Q：sti 和 cli 内联汇编函数是如何影响前者的？

A：cli汇编指令的作用为关闭硬件对可屏蔽中断的响应，该指令将eflags寄存器的IF位清零。

sti汇编指令的作用为开启硬件对可屏蔽中断的响应，该指令将eflags寄存器的IF位置一。

三、getcallerpcs函数

```
  getcallerpcs(&lk, lk->pcs);

// Record the current call stack in pcs[] by following the %ebp chai
void
getcallerpcs(void *v, uint pcs[])
{
  uint *ebp;
  int i;

  ebp = (uint*)v - 2;
  for(i = 0; i < 10; i++){
    if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff
      break;
    pcs[i] = ebp[1];     // saved %eip
    ebp = (uint*)ebp[0]; // saved %ebp
  }
  for(; i < 10; i++)
    pcs[i] = 0;
}
```

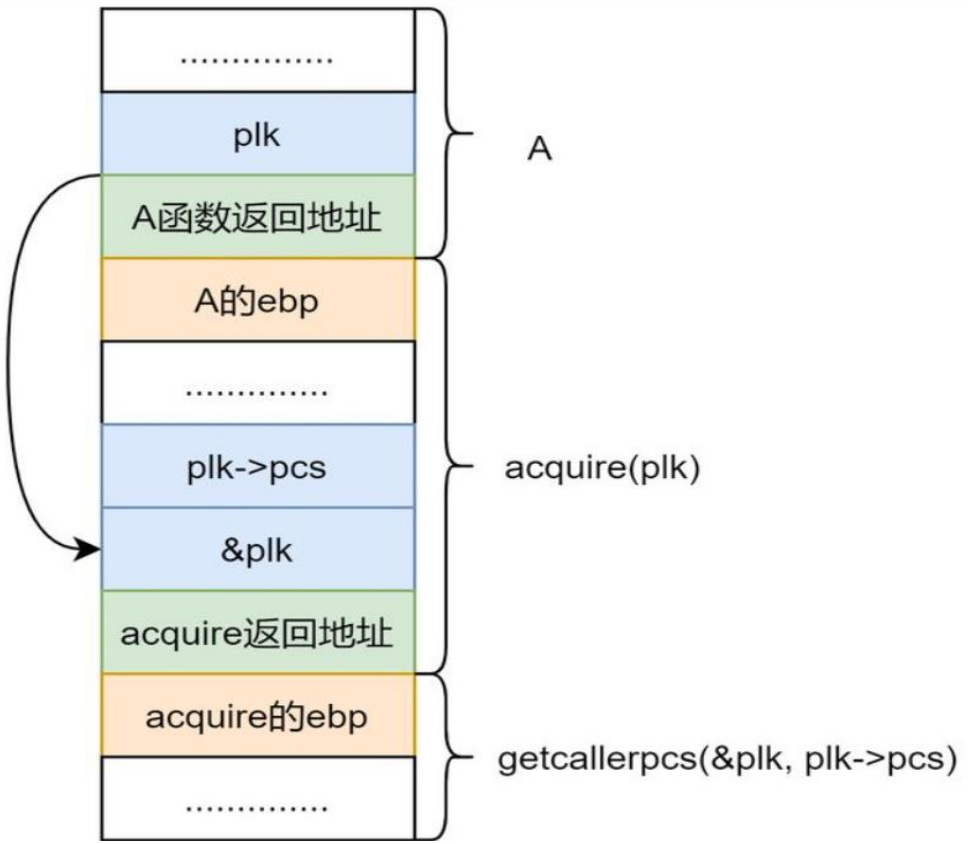


图2 调用栈结构图

Source：https://zhuanlan.zhihu.com/p/399615176

四、关于acquire函数的一些问题

Q: acquire 函数获取锁的时候，为什么首先要执行pushcli “关中断”操作？

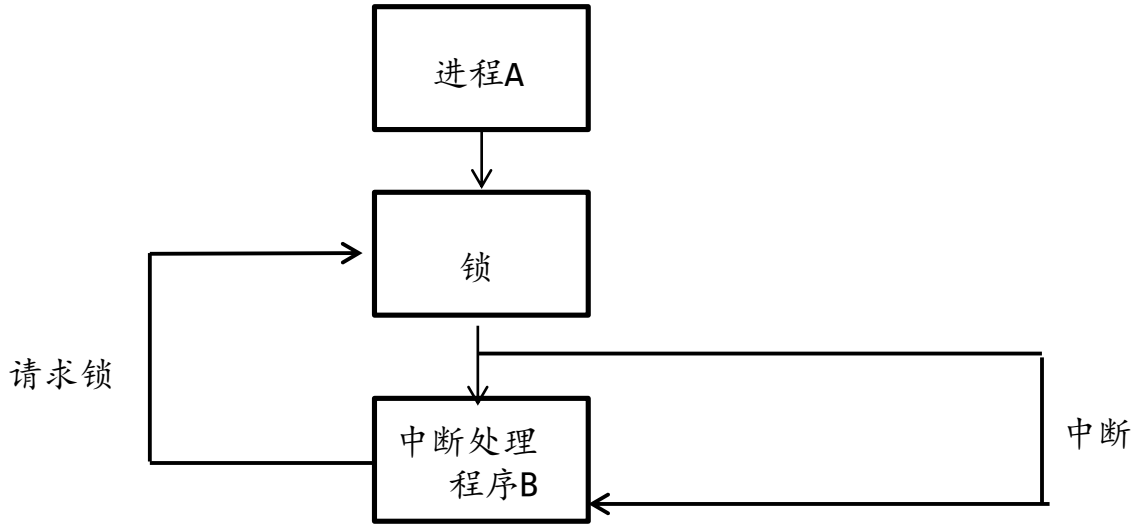
A: 防止产生“死锁”。（持有锁的进程等待中断处理程序，中断处理程序等待锁）

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```



四、关于acquire函数的一些问题

Q：内联汇编函数xchg 及xchgl指令是原子指令的重要性

A：考虑两个core上的分别有一个进程先后执行 xchgl ，返回值分别为0和1，即第一个进程持有锁，第二个进程进入等待。

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

四、关于acquire函数的一些问题

Q: acquire函数中， xchg 函数和它周围的while循环被编译成了什么汇编代码？

A:

```
1b0:   8b 5d 08                mov    0x8(%ebp),%ebx
1b3:   89 d0                   mov    %edx,%eax
1b5:   f0 87 03                lock xchg %eax,(%ebx)
1b8:   85 c0                   test   %eax,%eax
1ba:   75 f4                   jne    1b0 <acquire+0x30>
```

四、关于acquire函数的一些问题

Q: acquire 和 release 函数中的 __sync_synchronize(); 语句的作用是什么？在汇编程序中是如何体现的？

A: 实现内存屏障，防止该指令前后的访存指令越过重排，使得对临界区的内存访问在拿到锁之后才能进行。

```
1bc:   f0 83 0c 24 00          lock orl $0x0,(%esp)
```

```
1b0:   8b 5d 08                mov    0x8(%ebp),%ebx
1b3:   89 d0                   mov    %edx,%eax
1b5:   f0 87 03                lock xchg %eax,(%ebx)
1b8:   85 c0                   test   %eax,%eax
1ba:   75 f4                   jne    1b0 <acquire+0x30>
```

Linux 原子操作与自旋锁

一、原子操作

- 原子变量

```
typedef struct {
    int counter;
} atomic_t;
```

- 相应操作---以atomic_add为例

```
/**
 * atomic_add - add integer to atomic variable
 * @i: integer value to add
 * @v: pointer of type atomic_t
 *
 * Atomically adds @i to @v.
 */
static __always_inline void atomic_add(int i, atomic_t *v)
{
    asm volatile(LOCK_PREFIX "addl %1,%0"
                 : "+m" (v->counter)
                 : "ir" (i));
}
```

一、原子操作

- LOCK_PREFIX宏

```
#define LOCK_PREFIX "\n\tlock; "
```

- LOCK前缀的作用：
  - LOCK前缀如何实现原子操作？
  - LOCK前缀如何防止指令重排？

一、原子操作

- LOCK前缀如何实现原子操作？
  - 在Pentium及以前的处理器中，通过在某些特定的指令(read and write指令)前使用lock前缀，会让处理器显式地向总线发出Lock#信号，从而获取总线的控制权。来自其他处理器控制总线的请求将被阻塞。

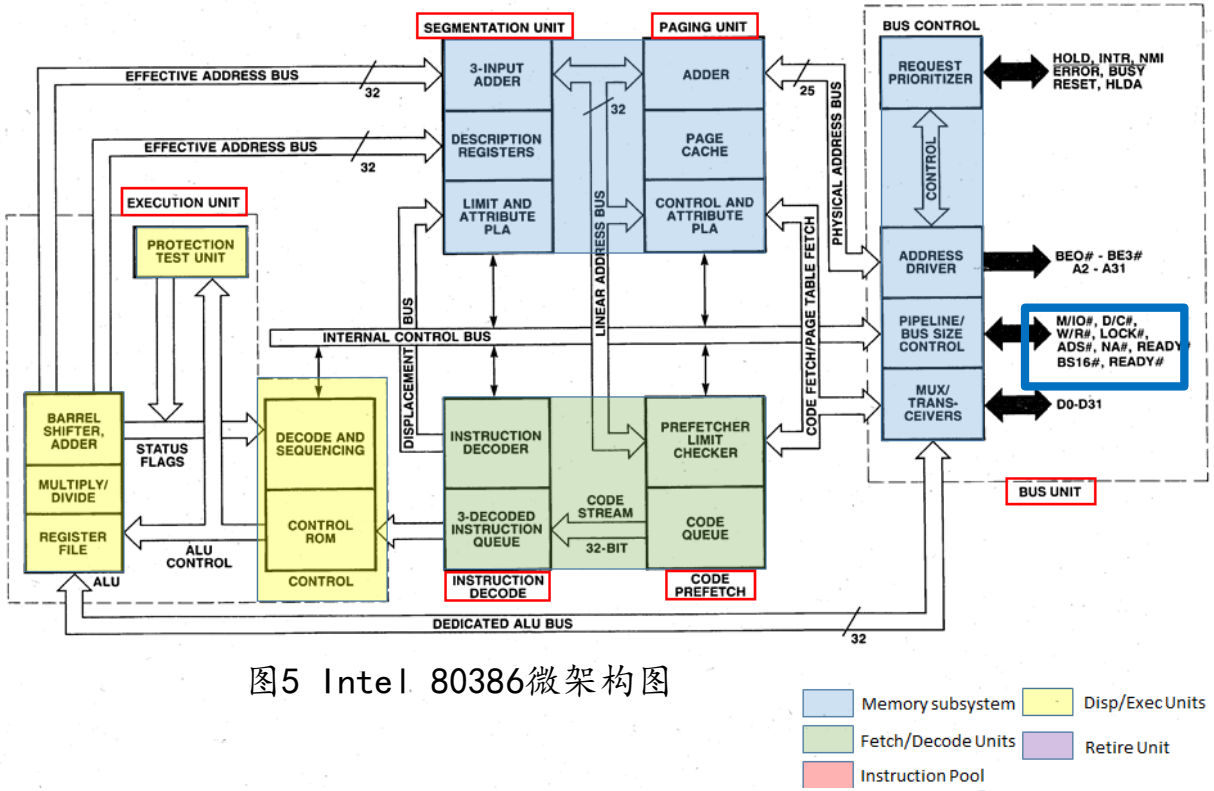


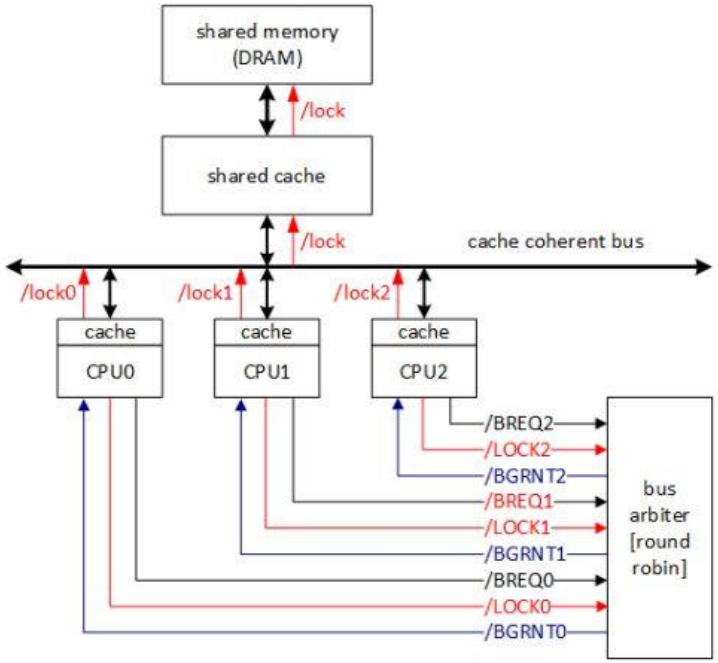
图5 Intel 80386微架构图

Source:
https://blog.csdn.net/qq_43401808/article/details/85231450?utm_medium=distribute.pc_relevant.none-ta

一、原子操作

- LOCK前缀如何实现原子操作？
  - 在Pentium及以前的处理器中，通过在某些特定的指令(read and write指令)前使用lock前缀，会让处理器显式地向总线发出Lock#信号，从而获取总线的控制权。来自其他处理器控制总线的请求将被阻塞。

Bus Arbiter



- ONLY one CPU can access shared memory at a time
- CPUs given access to bus and shared memory, one at a time, by bus arbiter

图6 Bus Arbiter示意图

Source:https://www.scss.tcd.ie/jeremy.jones/CS4021/

一、原子操作

- LOCK前缀如何实现原子操作？
  - 在P6及以后的处理器中，如果被访问的内存区域位于一个cache line中，并且内存访问模式为write-back，处理器通常不会在总线上发出LOCK#信号。在进行访存操作时，只有属于当前处理器的cache被锁住，并且利用cache一致性协议来确保这一操作是原子的，这一操作叫做“cache locking”。Cache一致性协议自动避免了两个缓存了相同内存区域的处理器同时在内存中修改数据数据的情况。
  - 其他情况：内存区域不在缓存中；跨cache line的访存;——锁总线

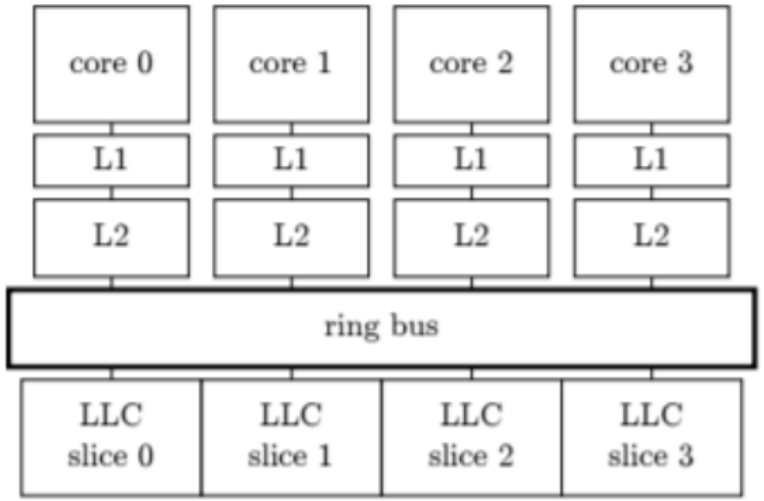


图7 Intel 多核处理器缓存结构示意图

Source:https://zhuanlan.zhihu.com/p/24146167

一、原子操作

Cache一致性协议MESI:

- 一种“窥探”（snooping）协议
- 主要包括Cache line的状态及消息通信机制
- Cache line的状态

Cache Line State	M (Modified)	E (Exclusive)	S (Shared)	I (Invalid)
This cache line is valid?	Yes	Yes	Yes	No
The memory copy is...	Out of date	Valid	Valid	—
Copies exist in caches of other processors?	No	No	Maybe	Maybe
A write to this line ...	Does not go to the system bus.	Does not go to the system bus.	Causes the processor to gain exclusive ownership of the line.	Goes directly to the system bus.

图8 cache line状态表

Source: Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumn 3A, Chapter11

一、原子操作

消息通信机制：

- Read：cpu发起读数据请求，请求中包含需要读取的数据地址
- Read Response：作为Read消息的响应，该消息可能是内存响应的，也可能是某CPU响应的。
- Invalidate：cpu发起“我要独占一个cache line，其他cpu请失效对应的cache line”的消息，消息中包含了内存地址，所有的其他cpu需要将对应的cache line置为Invalid状态。
- Invalidate ACK：收到Invalidate消息的cpu在将对应cache line置为Invalid后，返回Invalid ACK。
- Read Invalidate：相当于Read消息+Invalidate消息，即取得数据并且独占它，将收到一个Read Response和所有其他cpu的Invalidate ACK。
- Write Back：写回消息，即将状态为Modified的cache line写回到内存

一、原子操作

举个例子：

- 当两个core同时执行针对同一地址的指令时,其实他们是在试图修改每个core自己持有的Cache line。
- 假设两个core都持有相同地址对应cacheline,且各自cacheline 状态为S, 这时如果要想成功修改,就首先需要把S转为E或者M, 则需要向其它core invalidate 这个地址的cacheline。
- 两个core都会向ring bus 发出 invalidate这个操作, 那么在ringbus上就会根据特定的设计协议仲裁是core0,还是core1能赢得这个invalidate。
- 胜者完成操作, 失败者需要接受结果, invalidate自己对应的cacheline,再读取胜者修改后的值, 回到起点。

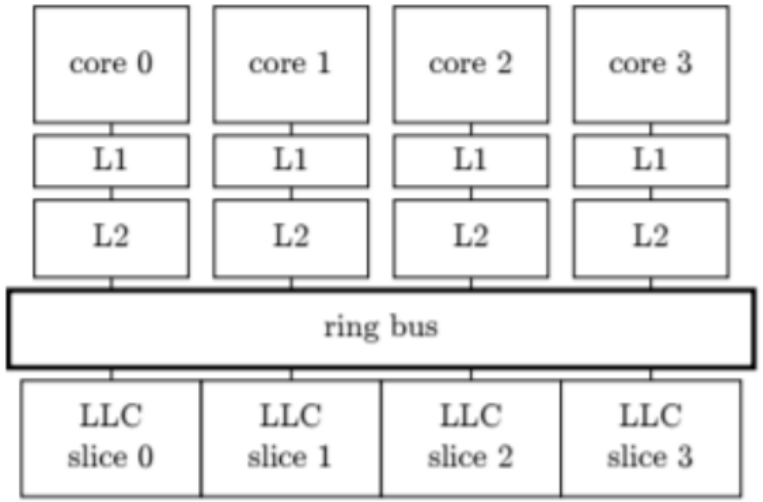


图9 Intel 多核处理器缓存结构示意图

Source:https://zhuanlan.zhihu.com/p/24146167

一、原子操作

LOCK前缀如何防止指令重排？

- MESI协议的缺陷：一种缓慢的总线事务（bus transaction）
- 优化：引入store buffer和invalid queue来加速bus transaction

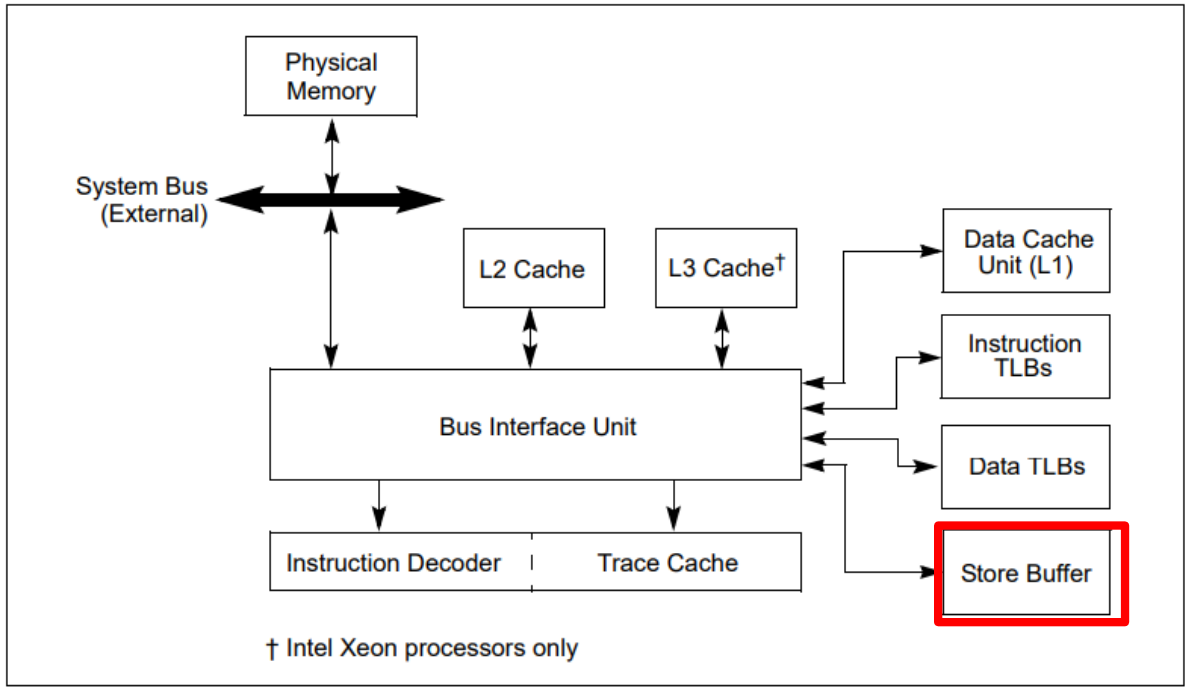


图10 Pentium 4 and Intel Xeon 处理器Cache结构图

Source: Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumn 3A, Chapter11

一、原子操作

为什么要引入Store Buffer？

- 当cpu需要的数据在其他cpu的cache内时，需要请求，并且等待响应，这是一个同步行为。
- 化同步为异步：通过在cpu和cache之间加一个store buffer，cpu可以先将数据写到store buffer，同时给其他cpu发送消息，然后继续做其它事情，等到收到其它cpu发过来的响应消息，再将数据从store buffer移到cache line。

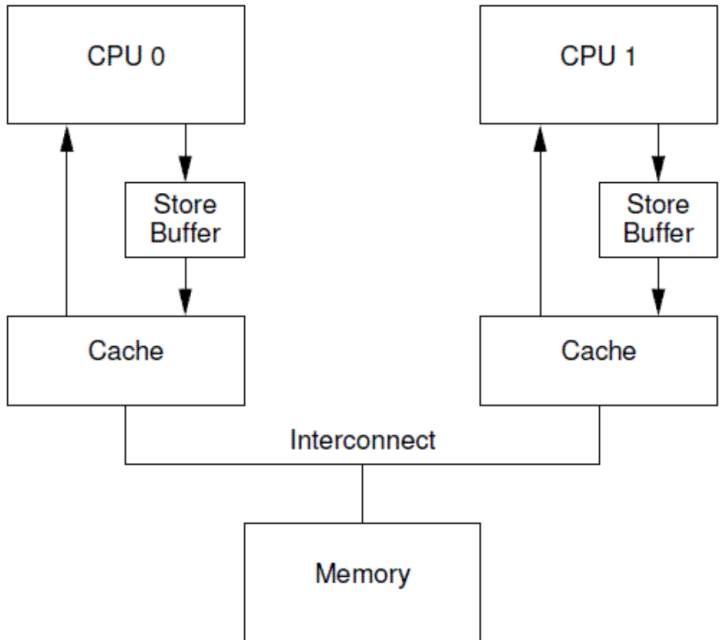


图11 引入store buffer后的结构图

一、原子操作

内存屏障：解决引入store buffer和invalid queue带来的问题

- 引入store buffer可能导致指令执行顺序的变化。要保证这种顺序一致性，需要在软件层面提供支持，即写屏障指令（write memory barrier）：Linux操作系统的smp_wmb()函数，把当前store buffer中的数据刷到cache之后，再执行屏障后的“写入操作”
- store buffer的大小有限，当store buffer满了之后，cpu还是会卡在等对应的Invalidate ACK以处理store buffer中的条目。Invalidate ACK耗时的主要原因是cpu要先将对应的cache line置为Invalid后再返回Invalidate ACK，一个很忙的cpu可能会导致其它cpu都在等它回Invalidate ACK。
- 化同步为异步: cpu不必处理了cache line之后才回Invalidate ACK，而是可以先将Invalid消息放到某个请求队列Invalid Queue，然后就返回Invalidate ACK。CPU可以后续再处理Invalid Queue中的消息，大幅度降低Invalidate ACK响应时间。cpu为防止invalid queue中的数据未处理导致指令执行顺序的变化，还提供了读屏障指令：Linux系统的smp_rmb()函数，把当前invalidate queue中的数据处理掉之后，再执行屏障后的“读取操作”
- x86：sfence、lfence、mfence

一、原子操作

LOCK前缀的执行过程

- 对总线/缓存上锁
- 强制所有lock前缀之前的指令，都在此之前被执行，并同步相关缓存
- 执行lock后的指令
- 释放对总线/缓存上的锁
- 强制所有lock前缀之后的指令，都在此之后被执行，并同步相关缓存
- Lock前缀虽然不是内存屏障，但是语义和mfence类似，具有内存屏障的功能

二、自旋锁——Ticket spinlock

- 产生原因：linux较早版本的自旋锁用一个整数表示其是否被占用，该整数初值为1，访问时依次递减。若递减后该值小于零，说明锁被占用。
- 在多个进程申请锁时，出现竞争，无法保证公平性（如无法保证等待时间长的进程先拿锁），尤其是释放锁的进程更容易再次拿到锁（由于cache存在）。
- 因此，Linux在内核版本2.6.25之后，实现了一套名为“FIFO ticked-based”算法的自旋锁机制。

二、自旋锁——Ticket spinlock

算法实现：

进程在初次申请锁时获得一个编号，然后根据当前锁对应的编号与自己是否相同来判断是否占用该锁。

锁的结构体中owner记录了当前占有该锁的进程编号，next记录了若有新进程请求锁，应当分配给该进程的编号，每分配一个新的编号，next加一。

```
typedef struct {
    union {
        u32 slock;
        struct __raw_tickets {
#ifdef __ARMEB__
            u16 next;
            u16 owner;
#else
            u16 owner;
            u16 next;
#endif
        } tickets;
    };
} arch_spinlock_t;
```

二、自旋锁——Ticket spinlock

```
//请求锁伪代码
spin_lock(lock *l){
int n = fetch_and_add(1, l->next);  //获得编号
while(n != atomic_read(l->owner))//查看是否该自己占用锁
      ;
}

//释放锁伪代码
spin_unlock(lock *l)
{
  atomic_add(1, l->owner); //服务完成，叫下一号
}
```

二、自旋锁——Ticket spinlock

请求锁操作：

prefetchw：写预取函数，将内存中的值提前放入缓存中

汇编代码：完成进程获得编号和锁最新编号加一操作

wfe：等待事件唤醒

ACCESS_ONCE：更新当前的占用进程编号

smp_mb：内存屏障

```
static inline void arch_spin_lock(arch_spinlock_t *lock)
{
    unsigned long tmp;
    u32 newval;
    arch_spinlock_t lockval;

    prefetchw(&lock->slock);
    __asm__ __volatile__(
"1: ldrex   %0, [%3]\n"
"   add %1, %0, %4\n"
"   strex   %2, %1, [%3]\n"
"   teq %2, #0\n"
"   bne 1b"
    : "=&r" (lockval), "=&r" (newval), "=&r" (tmp)
    : "r" (&lock->slock), "I" (1 << TICKET_SHIFT)
    : "cc");

    while (lockval.tickets.next != lockval.tickets.owner) {
        wfe();
        lockval.tickets.owner = ACCESS_ONCE(lock->tickets.owner);
    }

    smp_mb();
}
```

二、自旋锁——Ticket spinlock

释放锁操作：

smp_mb：内存屏障

当前占有编号加一

dsb_sev：发送事件，唤醒其他CPU

```
static inline void arch_spin_unlock(arch_spinlock_t *lock)
{
    smp_mb();
    lock->tickets.owner++;
    dsb_sev();
}
```

参考资料

LOCK前缀部分：

[1] https://www.cnblogs.com/Reverse-xiaoyu/p/11397584.html
[2] https://zhuanlan.zhihu.com/p/399615176
[3] https://blog.csdn.net/wll1228/article/details/107775976
[4] https://blog.csdn.net/qq_43401808/article/details/85231450?utm_medium=distribute.pc_relevant.none-ta.
[5] Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumn 2A, Chapter 3.
[6] Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumn 3A, Chapter 3, Chapter 8, Chapter 11.
[7] Intel 80386 Programmers’s Reference Manual 1986, Chapter 11.
[8] http://www.puppetmastertrading.com/images/hwViewForSwHackers.pdf1
[9] https://blog.csdn.net/reliveIT/article/details/90038750
[10] http://www.wowotech.net/kernel_synchronization/Why-Memory-Barriers.html
[11] https://zhuanlan.zhihu.com/p/24146167

参考资料

自旋锁部分：

[1] https://www.cnblogs.com/hehao98/p/10678493.html

[2] https://zhuanlan.zhihu.com/p/67893490

[3] https://blog.csdn.net/wll1228/article/details/107775976

[4] https://blog.csdn.net/zhoutaopower/article/details/86598839

[5] https://blog.csdn.net/reliveIT/article/details/90038750