第二次实例分析
同步、并发、IPC

中国科学院大学计算机与控制学院
中国科学院计算技术研究所

2018-11

outline

- 自旋锁
- 条件变量
- 时钟中断
- wait & exit & kill 系统调用
- 管道
- 睡眠锁

outline

- **自旋锁**
- 条件变量
- 时钟中断
- wait & exit & kill 系统调用
- 管道
- 睡眠锁

自旋锁

- spinlock.h

```
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?
                     //  0：不持有锁
                      //  1：持有锁
  // 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.
};
```

acquire函数 [spinlock.c]

```
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);
}
```

关中断（pushcli）的作用

- 假设请求锁（acquire）不进行关中断操作，某个应用申请到了锁
  - 例如，磁盘驱动程序idrew[4354]持有了idelock[4365]
- 此时来了一个中断信号，OS在对应的core调用了中断处理函数
  - 例如，磁盘驱动器发出了一个中断，示意读请求已经完成，OS调用ideintr[4304]
- 中断处理函数申请相同的锁，但已经被应用占有，应用需要等待中断处理函数执行完才能释放，死锁！
  - 例如，磁盘中断处理程序申请idelock[4309]

关中断（pushcli）的作用

- 假设请求锁（acquire）不进行关中断操作，某个应用申请到了锁
  - 例如，磁盘驱动程序idrew[4354]持有了idelock[4365]
  - 此时来了一个中断信号，OS在对应的core调用了中断处理函数
  - 例如，磁盘驱动器发出了一个中断，示意读请求已经完成，OS调用ideintr[4304]
  - 中断处理函数中请相同的锁，但已经被应用占有，应用需要等待中断处理函数执行完才能释放，死锁！
    - 例如，磁盘中断处理程序申请idelock[4309]

- 理论上：当某个锁可能被某个中断处理函数获得时，`acquire` 这个锁之前需禁用中断
- `xv6` 实现：每次 `acquire` 都禁用中断

xchg函数

- while(xchg(&lk->locked, 1) != 0);
  - 返回lk->locked在执行xchg之前的值
    - 执行之前，lk->locked == 0: 存入1，返回0，跳出循环，成功
    - 执行之前，lk->locked == 1: 存入1(不变)，返回1，继续自旋
- xchg指令的原子性保证了，两个进程先后执行acquire，只可能分别出现上述两种情况，且只有一个进程获得锁
- **xchg的成功即标志着临界区的开始**

如果xchg不是原子指令

while(xchg(&lk->locked, 1) != 0);　(参数1位于%eax)

进程1

- Load lk->locked, %eax (0,1)
- Store lk->locked, %eax (1,0)

进程2

- Load lk->locked, %eax (0,1)
- Store lk->locked, %eax (1,0)

两个进程都获得了锁，将导致出错

xchg函数

```
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;
}
```

LOCK — Assert LOCK# Signal Prefix

Opcode	Instruction	Clocks	Description
F0	LOCK	0	Assert LOCK# signal for the next instruction

Description

The LOCK prefix causes the LOCK# signal of the 80386 to be asserted during execution of the instruction that follows it. In a multiprocessor environment, this signal can be used to ensure that the 80386 has exclusive use of any shared memory while LOCK# is asserted. The read-modify-write sequence typically used to implement test-and-set on the 80386 is the BTS instruction.

Cite: intel 80386 programmer's reference manual

acquire函数的反汇编结果

```
 asm volatile("lock; xchgl %0, %1" :
801042f7:       ba 01 00 00 00          mov    $0x1,%edx
801042fc:       eb 05                   jmp    80104303 <acquire+0x23>
801042fe:       66 90                   xchg   %ax,%ax
80104300:       8b 5d 08                mov    0x8(%ebp),%ebx
80104303:       89 d0                   mov    %edx,%eax
80104305:       f0 87 03                lock xchg %eax,(%ebx)
  while(xchg(&lk->locked,  1) != 0)
80104308:       85 c0                   test   %eax,%eax
8010430a:       75 f4                   jne    80104300 <acquire+0x20>
  __sync_synchronize();
8010430c:       f0 83 0c 24 00          lock orl $0x0,(%esp)
```

内存屏障

```
// 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();

8010430c:       f0 83 0c 24 00      lock orl $0x0,(%esp)
```

- 处理器和编译器可能会做乱序优化提高性能，而__sync_synchronize告诉编译器和处理器不要做乱序优化。
- 编译之后的汇编指令是lock or，lock前缀锁内存总线，所以带lock前缀指令执行前，会先将未完成的读写操作完成，从而起到内存屏障的功能。

release函数 [spinlock.c]

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

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

  // 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();
}
```

release函数 [spinlock.c]

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

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

  // Both the C compiler and the hardware may
  // stores; __sync_synchronize() tells them
  __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();
}
```

- movl指令：将lk→locked置0，**标志着进程退出临界区**
  - 与acquire的区别：
    - release不需要while循环，因为**release是由持有锁的进程在临界区内执行的，因此必然成功**
    - acquire需要while循环，因为是自旋锁，进程拿不到锁时会进行死循环
- popcli：启用中断

pushcli / popcli 函数 [spinlock.c]

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

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

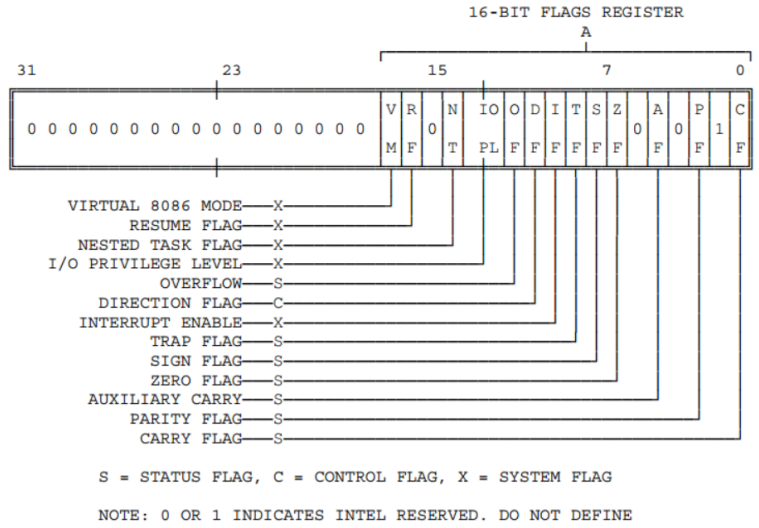
pushcli / popcli 函数 [spinlock.c]

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

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

- EFLAGS：x86 CPU中的控制寄存器
  - 包含运算指令会设置的标志位，如ZF SF
  - 也包含控制标志位，如IF Interrupt enable Flag
  - readeflags()：读取控制寄存器的值到eflags变量

Figure 2-8. EFLAGS Register



Cite: intel 80386 programmer's reference manual

pushcli / popcli 函数 [spinlock.c]

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

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

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

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

CLI — Clear Interrupt Flag

Opcode	Instruction	Clocks	Description
FA	CLI	3	Clear interrupt flag; interrupts disabled

Operation

```
IF ← 0;
```

STI — Set Interrupt Flag

Opcode	Instruction	Clocks	Description
F13	STI	3	Set interrupt flag; interrupts enabled at the end of the next instruction

Operation

```
IF ← 1
```

Cite: intel 80386 programmer's reference manual

pushcli / popcli 函数 [spinlock.c]

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

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

- 为何不直接使用cli()与sti()?
  - 有时内核会同时持有两个锁，又因为cli和sti指令本身不计算次数，会导致第一次release时就启用中断
  - 内核还会在其他场合禁用中断，因此需要保存cli()之前的中断启用情况，以在release时判断是否可以sti()
- cpu->intena：保存当前cpu在pushcli之前的IF状态，以便恢复

outline

- 自旋锁
- **条件变量**
- 时钟中断
- wait & exit & kill 系统调用
- 管道
- 睡眠锁

pthread 条件变量 API

- int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
- int pthread_cond_broadcast(pthread_cond_t *cond);
- int pthread_cond_signal(pthread_cond_t *cond);

- 注意:
  - XV6中的sleep函数，功能与pthread_cond_wait相似，都使得调用线程阻塞在对应的条件变量上。
  - 但是，XV6中的wakeup函数的语义，并不对应pthread_cond_signal函数，而是与pthread_cond_broadcast函数一样会唤醒阻塞在条件变量上的所有线程。

sleep函数 [proc.c]

```
void sleep(void *chan, struct spinlock *lk) {
  struct proc *p = myproc();
  if(p == 0)
    panic("sleep");
  if(lk == 0)
    panic("sleep without lk");
  // Must acquire ptable.lock in order to change p->state and then call sched.
  // Once we hold ptable.lock, we can be guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked), so it's okay to release lk.
  if(lk != &ptable.lock) {
    acquire(&ptable.lock);
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;
  sched();
  // Tidy up.
  p->chan = 0;
  // Reacquire original lock.
  if(lk != &ptable.lock){release(&ptable.lock);
    acquire(lk);
  }
}
```

为什么sleep需要lk参数?

- 错误实现 1

```
void*
send(struct q *q, void *p)
{
    while(q->ptr != 0)
        ;
    q->ptr = p;
    wakeup(q);      /*wake recv*/
}

void*
recv(struct q *q)
{
    void *p;
    while((p = q->ptr) == 0)
        sleep(q);
    q->ptr = 0;
    return p;
}
```

- 自旋锁的同步控制，进程需要不断循环直到获取到锁，极大地浪费了cpu资源。
- 自然地，会想要让进程在条件不满足（没有申请到锁）的时候，让出cpu从而减少spin开销
- 错误实现1：
  receiver在执行sleep之前，sender可以顺利地率先执行wakeup，导致receiver永远睡眠——lost-wakeup问题
- 需要锁来保护关键区

为什么sleep需要lk参数?

- 错误实现 2

```
struct q {
    struct spinlock lock;
    void *ptr;
};

void *
send(struct q *q, void *p)
{
    acquire(&q->lock);
    while(q->ptr != 0)
        ;
    q->ptr = p;
    wakeup(q);
    release(&q->lock);
}
```

```
void*
recv(struct q *q)
{
    void *p;
    acquire(&q->lock);
    while((p = q->ptr) == 0)
        sleep(q);
    q->ptr = 0;
    release(&q->lock;
    return p;
}
```

- 错误实现2：
  - receiver在睡眠时持有锁，导致sender无法获取到锁，更不能执行wakeup将receiver唤醒
  - 造成死锁

- 需要将锁的信息传给sleep，在进入睡眠之前释放锁，被唤醒之后重新获得锁

为什么sleep需要lk参数?

正确实现

```
struct q {
    struct spinlock lock;
    void *ptr;
};

void *
send(struct q *q, void *p)
{
    acquire(&q->lock);
    while(q->ptr != 0)
        ;
    q->ptr = p;
    wakeup(q);
    release(&q->lock);
}
```

```
void*
recv(struct q *q)
{
    void *p;
    acquire(&q->lock);
    while((p = q->ptr) == 0)
        sleep(q, &q->lock);
    q->ptr = 0;
    release(&q->lock;
    return p;
}
```

sleep实现分析

```
void sleep(void *chan, struct spinlock *lk) {
  // …

  // Must acquire ptable.lock in order to change p->state and then call sched.
  // Once we hold ptable.lock, we can be guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked), so it's okay to release lk.
  if(lk != &ptable.lock) {
    acquire(&ptable.lock);
    release(lk);
  }
  // …
}
```

- sleep首先获得ptable.lock，需要ptable.lock的原因是：
  - sleep本身会对pcb中的进程状态进行修改并调用sched()。
  - sleep需要暂时释放lk以防止死锁，**导致释放lk和修改进程状态之间存在一个“真空区”**，可能会有其他进程修改进程状态。
  - 因此，需要持有ptable.lock保证对于进程状态的修改是在临界区内的。

sleep实现分析

```
void sleep(void *chan, struct spinlock *lk) {
  // …

  // Must acquire ptable.lock in order to change p->state and then call sched.
  // Once we hold ptable.lock, we can be guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked), so it's okay to release lk.
  if(lk != &ptable.lock) {
    acquire(&ptable.lock);
    release(lk);
  }
  // …
}
```

- if语句的作用是防止lk等于&ptable.lock时出现死锁
  - 例如，wait函数中调用sleep时[2707]，传入的lk参数就是ptable.lock
  - 这种情况下，可认为acquire和release相互抵消，因此跳过这一步
- 中间两行的顺序不可交换，否则在release lk和acquire ptable.lock之间仍旧存在一个没有锁保护的“真空区”。

进程状态分析——进入等待状态

```
void sleep(void *chan, struct spinlock *lk) {
  // …
 if(lk != &ptable.lock) {
    acquire(&ptable.lock);
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;
  sched();
  // …
}
```

- 修改进程状态并转入睡眠态
- 注意到进程等待的条件变量地址chan被记录在pcb中
- 修改proc->state=SLEEPING并调用sched()，实现进程进入等待状态并切换执行进程

进程状态分析——唤醒等待进程

```
static void wakeup1(void *chan) {
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}

// Wake up all processes sleeping on chan.
void wakeup(void *chan) {
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

- wakeup函数遍历进程表，将所有等待条件变量chan的进程，都修改成就绪状态。
- wakeup操作并不会保证唤醒的进程立刻执行，因此是符合**Mesa semantics**的。
- 如果没有sleeping的进程，wakeup不会阻塞。

进程状态分析——唤醒等待进程

```
static void wakeup1(void *chan) {
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}

// Wake up all processes sleeping on chan.
void wakeup(void *chan) {
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

- 有时候，一个函数会使用需要被锁保护的数据结构，但这个方法既可能被已经持有锁的函数调用，也可能被不持有锁的函数调用。
  - 一个解决这个问题的方法是，使用两个函数，一个会请求锁，另一个则不会。wakeup1()是其中的代表，在exit()中有被调用。
  - 另一个解决方案，是不管调用函数是否需要持有锁，在调用被调函数的时候统一都先请求锁，sched就是其中的代表。

进程状态分析——进程再次执行

```
void scheduler(void) {
    // …
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
            continue;
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&(c->scheduler), p->context);
        // …
}
```

- wakeup函数唤醒所有等待条件变量chan的进程后，在某一时刻，调度器选择进程表中第一个就绪态的进程（假设为先前调用sleep函数进入等待状态的线程），修改进程状态为运行态，执行swtch进行上下文切换，切换回sleep函数中继续执行。

进程状态分析——进程再次执行

```
void sleep(void *chan, struct spinlock *lk) {
  // …
  sched();
  // Tidy up.
  p->chan = 0;
  // Reacquire original lock.
  if(lk != &ptable.lock){
    release(&ptable.lock);
    acquire(lk);
  }
}
```

- 此时进程已经被wakeup函数唤醒
- 清空chan字段
- sleep退出前再次获取lk

outline

- 自旋锁
- 条件变量
- **时钟中断**
- wait & exit & kill 系统调用
- 管道
- 睡眠锁

时钟中断的注册过程 [main.c]

```
int main(void) {
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

时钟中断的注册过程 [lapic.c]

- 硬件上：大约以每秒100次的速度触发时钟中断。
- 在多核处理器上，时钟芯片是在LAPIC中的，LAPIC是关联在每一个处理器上的，所以每个处理器都可以独立地接收时钟中断。
- 对LAPIC（关于时钟中断）的编程位于lapicinit（7408）中。

```
void lapicinit(void) {
        …
        lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
        …
}
```

告诉LAPIC周期性地在IRQ_TIMER上产生中断信号，也就是IRQ 0

时钟中断的注册过程 [lapic.c]

- 硬件上：大约以每秒100次的速度触发时钟中断。
- 在多核处理器上，时钟芯片是在LAPIC中的，LAPIC是关联在每一个处理器上的，所以每个处理器都可以独立地接收时钟中断。
- 对LAPIC（关于时钟中断）的编程位于lapicinit（7408）中。

```
void lapicinit(void) {
        …
        lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
        …
        lapicw(TPR, 0);

}
```

打开一个CPU上的LAPIC的中断，允许这个LAPIC向处理器传递中断信号

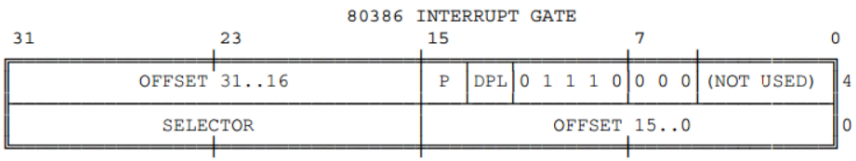
时钟中断的注册过程 [trap.c]

- vectors.pl 脚本生成了一个vectors数组，数组元素是0~255号中断处理程序入口的地址，以0号中断处理程序为例，入口代码片段如下：

```
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```

- tvinit 中使用 SETGATE宏，将vectors数组中的入口地址，构造出了中断描述符表idt数组。

```
#define SETGATE(gate, istrap, sel, off, d)  {       \
  (gate).off_15_0 = (uint)(off) & 0xffff;           \
  (gate).cs  = (sel);                               \
  (gate).args  = 0;                                 \
  (gate).rsv1  = 0;                                 \
  (gate).type  = (istrap) ? STS_TG32 : STS_IG32;    \
  (gate).s  = 0;                                    \
  (gate).dpl  = (d);                                \
  (gate).p  = 1;                                    \
  (gate).off_31_16 = (uint)(off) >> 16;             \
}
```



Cite: intel 80386 programmer's reference manual

时钟中断的注册过程

- tvinit 中使用 SETGATE宏，将vectors数组中的入口地址，构造出了中断描述符表idt数组。

```
void tvinit(void) {
  int i;
  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
  initlock(&tickslock, "time");
}
```

- mpmain中调用lidt，将中断描述符数组idt的首地址，保存到IDTR寄存器中，整个中断注册完成。

```
static void mpmain(void) {
  cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
  idtinit();       // load idt register
  xchg(&(mycpu()->started), 1); // tell startothers() we're up
  scheduler();     // start running processes
}
```

时钟中断处理程序

```
void trap(struct  trapframe  *tf) {
  // …
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid()  == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
```

- ticks（计数的时钟）

每次触发时钟中断，如果是0号CPU，ticks都要加1，然后唤醒在ticks上处于睡眠状态的所有进程。

ticks、tickslock的作用

```
int sys_sleep(void) {
  int n;
  uint ticks0;

  if(argint(0, &n) < 0)
    return -1;
  acquire(&tickslock);
  ticks0 = ticks;
  while(ticks - ticks0 < n){
    if(myproc()->killed){
      release(&tickslock);
      return -1;
    }
    sleep(&ticks, &tickslock);
  }
  release(&tickslock);
  return 0;
}
```

- ticks（计数的时钟）

每次触发时钟中断，如果是0号CPU，ticks都要加1，然后唤醒在ticks上处于睡眠状态的所有进程。

- sys_sleep系统调用，会让进程睡眠n个时钟中断。在每次时钟中断到来的时候，wakeup唤醒所有等待在ticks上的进程，这些进程分配到cpu继续执行的时候，通过while语句判断睡眠时间是否已经足够。如果时间不足，会再次调用sleep转为等待状态。
- tickslock 用来维护对共享变量ticks进行操作的临界区。

内核中进程切换的调用流程 1

- 从IDTR寄存器中读出idt的首地址，根据中断号，在idt数组中索引对应的门描述符
- 将门描述符中的offset域组合成目标地址，跳转到目标地址 ---> (即 vectors.pl生成的入口程序)

```
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```

内核中进程切换的调用流程 2

- 进入 alltraps 指令段 [trapasm.S]，建立trapframe，调用trap函数

```
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal
  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp
```

内核中进程切换的调用流程 3

```
void trap(struct trapframe *tf) {
  // …
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  // …
  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
  // ...
}
```

- 进入trap函数[trap.c]后，首先运行时钟中断处理程序，唤醒所有等待ticks的进程，然后调用yield函数让出cpu资源

内核中进程切换的调用流程 4

- 进入yield函数[proc.c]后，将当前进程状态置为就绪态，调用sched函数

```
// Give up the CPU for one scheduling round.
void yield(void) {
    acquire(&ptable.lock);  //DOC: yieldlock
    myproc()->state = RUNNABLE;
    sched();
    release(&ptable.lock);
}
```

内核中进程切换的调用流程 5

- Sched函数[proc.c]调用swtch进行当前进程和调度器进行上下文切换

```
void sched(void) {
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

内核中进程切换的调用流程 6

```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx
  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi
  # Switch stacks
  movl  %esp, (%eax)
  movl  %edx, %esp
  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

- swtch函数[swtch.S]保存当前进程的上下文信息，并且切换到调度器的内核栈上
- ret回到上次调度器执行swtch函数的下一行

内核中进程切换的调用流程 7

```
void scheduler(void) {
  // …
  for(;;){
    sti();
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);
  }
}
```

- swtch函数ret回到上次调度器执行swtch函数的下一行，也就是switchkvm
- 紧接着，调度器进入下一次循环

内核中进程切换的调用流程 8

```
void scheduler(void) {
  // …
  for(;;){
    sti();
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);
  }
}
```

- 调度器选择进程表中第一个处于就绪态的进程，调用swtch进行上下文切换，执行该进程。

调度器中的开关中断

- 为什么scheduler函数的循环体最开始要先开中断sti()？明明紧接着的acquire(&ptable.lock)立刻就关中断了？
- The reason to enable interrupts periodically on an idling CPU is that there might be no RUNNABLE process because processes (e.g., the shell) are waiting for I/O; if the scheduler left interrupts disabled all the time, the I/O would never arrive.

```
void scheduler(void) {
  // …
  for(;;){
    sti();
    acquire(&ptable.lock);
  // ..
}
```

yield函数

- yield()函数中的acquire(&ptable.lock)操作与release(&ptable.lock)操作执行过程中，与一般的线程使用自旋锁来保障临界区有什么区别？ (Hint: sleep同理)
- 一般来说，锁的获取和释放都是在同一个线程中完成的，但是yield是在两个线程中完成的
- xv6 acquires ptable.lock in one thread (often in yield) and releases the lock in a different thread (the scheduler thread or another next kernel thread).

outline

- 自旋锁
- 条件变量
- 时钟中断
- **wait & exit & kill 系统调用**
- 管道
- 睡眠锁

exit函数（进程“自杀”）

```
void exit(void) {
  // …
  acquire(&ptable.lock);
  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);
  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }
  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

- 进程调用exit函数，需要修改进程状态为僵尸态，因此首先需要获取ptable.lock保证对进程状态的互斥修改

wait 函数（父进程“收尸”）

```
int wait(void) {
 // …
 acquire(&ptable.lock);
 for(;;){
  // Scan through table looking for exited children.
  havekids = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
   if(p->parent != curproc)
    continue;
   havekids = 1;
   if(p->state == ZOMBIE){
    pid = p->pid;
    kfree(p->kstack);
    p->kstack = 0;
    freevm(p->pgdir);
    p->pid = 0;
    p->parent = 0;
    p->name[0] = 0;
    p->killed = 0;
    p->state = UNUSED;
    release(&ptable.lock);
    return pid;
   }
  }
  // …
 }
}
```

- 父进程调用了wait函数等待子进程退出：
  - 遍历ptable找到一个僵尸态的子进程
    - 回收内核栈
    - 回收页表
    - pid改为0
    - parent指针悬空
    - 名字name改为空串
    - killed赋0
    - 进程态改为UNUSED
    - 释放锁ptable.lock
    - 返回被清理的子进程原pid
  - 至此进程彻底“消亡”

- 一般由父进程A回收子进程B的页表和内核栈，但如果父进程A先于子进程B执行exit操作，由谁来回收子进程B的相关数据结构？
  - 初始进程 initproc

```
void exit(void) {
  // …
  acquire(&ptable.lock);
  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);
  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }
  // …
}
```

- 进程exit之前，要把该进程的子进程转交给初始进程initproc；
- 如果该进程的子进程中有ZOMBIE，要唤醒initproc去回收相应的“弃子”。

wait & exit 同步控制

```
void exit(void) {
  …
  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }
 // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

- exit需要将退出的进程状态修改为ZOMBIE态，因此需要ptable.lock来保证互斥
- exit通过wakeup操作需要唤醒父进程，通知其可以执行清理工作。
- 由于此时exit的进程已经持有了ptable.lock因此调用wakeup1而非wakeup。并且能够调用sched进行进程切换。

wait & exit 同步控制

```
int wait(void) {
  // …
  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        …
        release(&ptable.lock);
        return pid;
      }
    }
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }
    sleep(curproc, &ptable.lock);
  }
}
```

- wait清理ZOMBIE态的子进程并返回子进程pid；没有孩子或已经被kill则返回-1；否则sleep等待子进程唤醒父进程为其“收尸”。
- wait函数需要将回收的进程状态设置为UNUSED，因此首先需要获取ptable.lock

kill函数（进程“他杀”）

```
int kill(int  pid) {
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

- kill函数杀死指定pid的进程，但是kill函数中并不会回收指定进程的相关数据结构，近近是将PCB中的killed置1.

kill函数（进程“他杀”）

```
void trap(struct trapframe *tf) {
   if(tf->trapno == T_SYSCALL){
      if(myproc()->killed)
         exit();
      myproc()->tf = tf;
      syscall();
      if(myproc()->killed)
         exit();
      return;
   }
   switch(tf->trapno){…}
   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
      exit()
   if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
      yield();
   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
      exit();
}
```

- 被kill的进程总会通过系统调用或中断（如时钟中断）进入内核。
- trap若检测到进程的killed域被设置了，该进程就会调用exit终结自己。

- 为什么调用mycpu()函数的时候（例如：myproc（2456）函数）中，需要先关中断？为什么调用myproc()函数（例如wait、exit）的时候不需要关中断?

- The return value of mycpu is fragile: if the timer were to interrupt and cause the thread to be moved to a different processor, the return value would no longer be correct. To avoid this problem, xv6 requires that callers of mycpu disable in- terrupts, and only enable them after they finish using the returned struct cpu.
- The return value of myproc is safe to use even if interrupts are enabled: if a timer interrupt moves the calling process to a different processor, its struct proc pointer will stay the same.

outline

- 自旋锁
- 条件变量
- 时钟中断
- wait & exit & kill 系统调用
- **管道**
- 睡眠锁

管道

```
struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;     // number of bytes read
  uint nwrite;    // number of bytes written
  int readopen;   // read fd is still open
  int writeopen;  // write fd is still open
};
```

- lock为pipe使用的自旋锁
- nread为总共已读取的字节数，nwrite为总共已写入的字节数
  - 这两个变量在自增过程中不会对PIPESIZE取模，是为了方便将管道满(nwrite == nread + PIPESIZE)和管道空(nwrite == nread)区分开来

管道实现中的同步控制

- piperead和pipewrite函数中，为什么要使用两个条件变量（p->nread,p->nwrite）？能否只使用一个？如果不能请举例说明。

- 实际上此处可以只使用一个条件变量，与传统使用条件变量解决生产者消费者问题不同，xv6中wakeup会唤醒所有等待进程，因为不会造成所有进程都陷入等待状态。
- 如果使用pthread_cond_signal而不是pthread_cond_broadcast，则不能只是用一个条件变量，留作思考。

管道实现中的同步控制

- 当只有一个读者和写者的时候，pipewrite函数中的6836行以及piperead函数中的6856行的while语句能否使用if代替？多读者多写者呢？

- 当只有单读者和单写者的时候，可以用 if 替代 while，并不会有其他线程来改变buffer的状态；
- 多读者多写者，当然不可以进行替换！
- 实际上，在编写并发程序的时候，有这样一句话，“use while (not if) for conditions”，上述情况下，使用while都是不会错的，因此在使用条件变量的时候，最好都是用while语句。

outline

- 自旋锁
- 条件变量
- 时钟中断
- wait & exit & kill 系统调用
- 管道
- **睡眠锁**

睡眠锁

```
// Long-term locks for processes
struct sleeplock {
  uint locked;       // Is the lock held?
  struct spinlock lk; // spinlock protecting this sleep lock

  // For debugging:
  char *name;        // Name of lock.
  int pid;           // Process holding lock
};
```

- lk为sleeplock使用的自旋锁
- XV6中，sleeplock的应用场景仅出现在文件系统中

acquiresleep & releasesleep 函数

```
void acquiresleep(struct sleeplock *lk) {
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}
```

```
void releasesleep(struct sleeplock *lk) {
  acquire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

- sleeplock的获取和释放，实际上是非常标准的管程模拟
- sleep和wakeup的调用都在sleeplock->lk所保障的临界区之内
- 申请sleeplock的时候，如果已经被其他线程占有，则调用sleep进入等待状态；
- 释放sleeplock的时候，唤醒所有在等待sleeplock->lk的进程

- 为什么sleeplock在获取到锁之后，直到释放锁的这段时间内，并没有像spinlock一样屏蔽中断?

- 如果在获取到sleeplock之后，中断仍旧处于关闭状态，那么进程将IO请求发送给磁盘驱动器并进入睡眠状态，等待IO操作完成。
- IO操作完成之后，发出中断信号，由于此时中断仍旧处于关闭状态，导致该进程不会被唤醒!


纸上得来终觉浅
绝知此事要躬行