小组成员：张玉龙 李盛开 宋宇轩 杨卓琛

1、时钟中断的注册过程

1. 中断控制器

中断在进入CPU前，首先会进入一个被称为 Advanced Programmable Interrupt Controller (APIC) 的控制器中。每个CPU都有一个APIC，称为该CPU的Local APIC (LAPIC)，每个LAPIC由一系列寄存器组成，这些寄存器控制了LAPIC如何将中断送到处理器中。其中一个中断来源就是 APIC timer generated interrupts，即APIC可以通过编程设置某个counter，在固定时间后向处理器发送中断。即时钟中断。

*2. LAPIC是如何工作的

LAPIC通过一系列寄存器工作。

在xv6中，LAPIC的寄存器被映射在内存中，通过内存读写的方式进行访问。其中与时钟中断有关的最重要的一些寄存器，包括 Local Vector Table (LVT)，TICR,TDCR,TCCR

LVT

LAPIC能处理的中断种类有限，LVT表的表项记录了其对每一种中断的处理。举个栗子

```
#define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
  #define X1         0x0000000B   // divide counts by 1
  #define PERIODIC   0x00020000   // Periodic
#define PCINT   (0x0340/4)   // Performance Counter LVT
#define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
#define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
#define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
```

> TIMER：负责发送由APIC Timer产生的中断，即时钟中断
>
> LINT0：负责转发来自LINT0引脚的中断
>
> LINT1：负责转发来自LINT1引脚的中断
>
> ERROR：负责发送由于APIC内部错误产生的中断

具体来说，每一个表项的结构都不同，（图片来源：https://zhuanlan.zhihu.com/p/113365730）

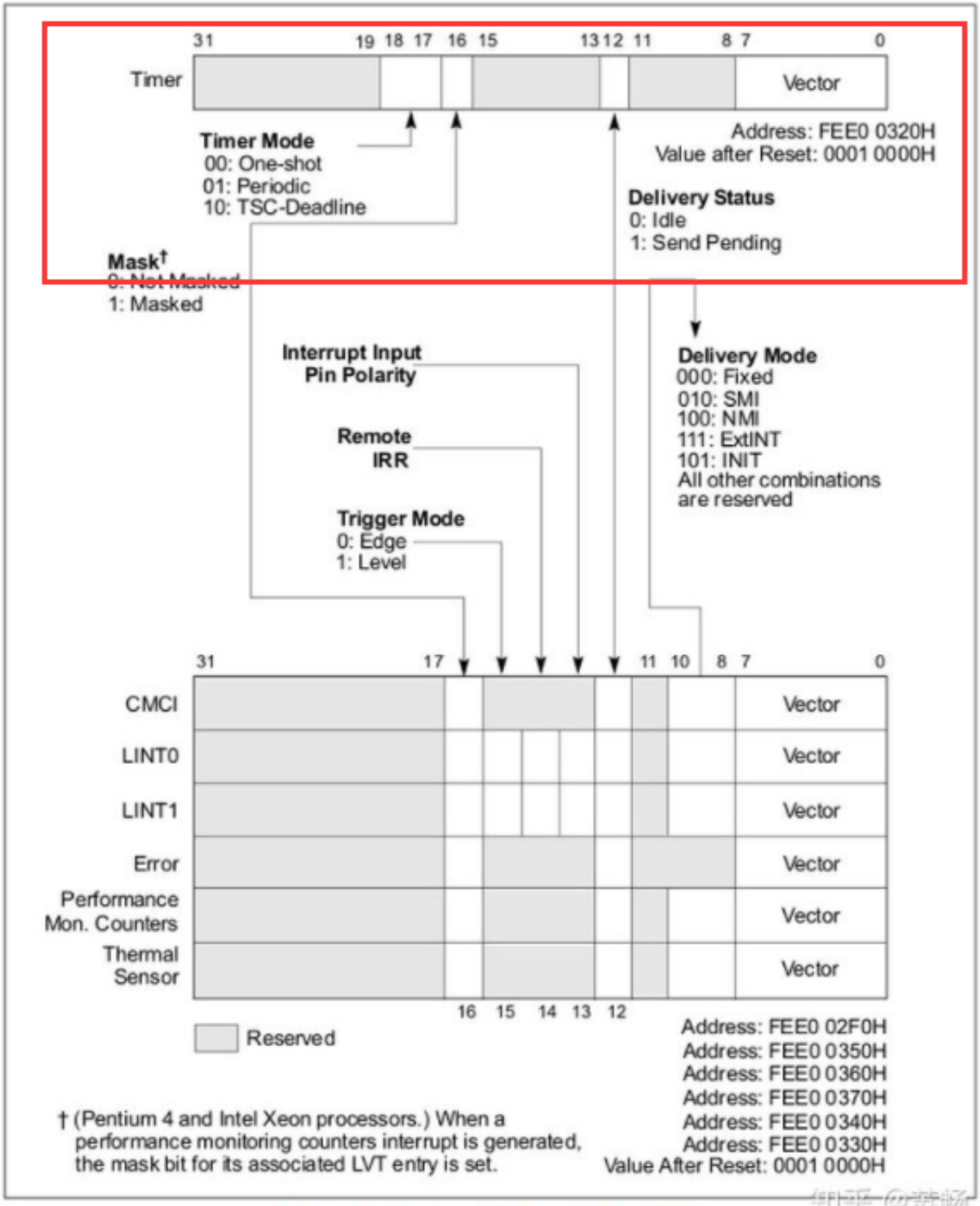
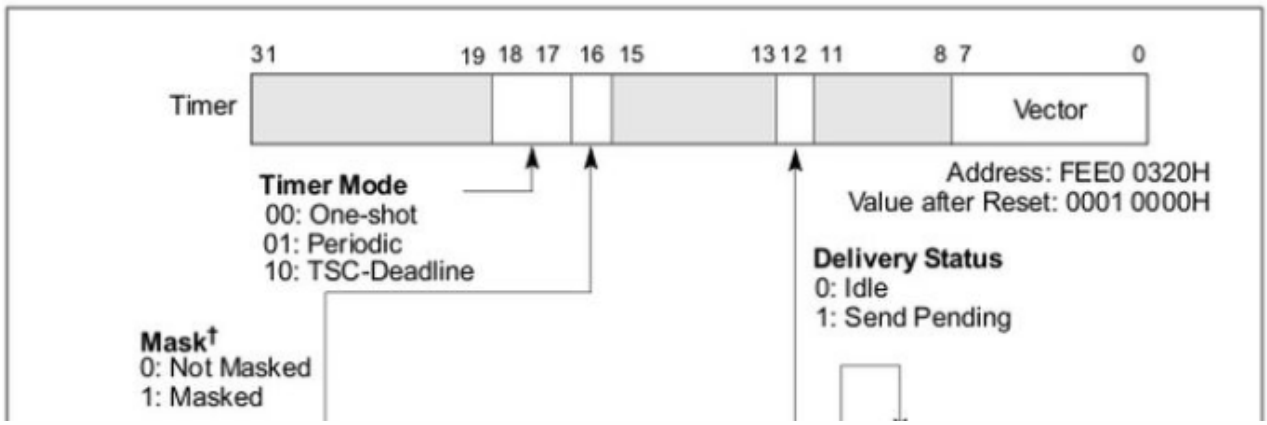


Figure 10-8. Local Vector Table (LVT)

这里着重展示一下TIMER表项的构成



其中最重要的就是

> 0~7(vector)：CPU收到的向量号，当作id
>
> 17~18(Timer Mode)：00表示注册时钟中断（具体过程后面介绍），01表示周期性触发

3. 时钟中断的注册过程

TICR, TDCR,TCCR

LVT相当于设置定时器的模式，而TCIR，TDCR，TCCR是真正关系到计数的寄存器

```
#define TICR    (0x0380/4)   // Timer Initial Count
#define TCCR    (0x0390/4)   // Timer Current Count
#define TDCR    (0x03E0/4)   // Timer Divide Configuration
```

xv6的代码注释解释的很清楚，TICR是**初始计数寄存器**，TCCR是当前计数寄存器，TDCR是**频率配置寄存器**。

在xv6中，最关键的代码位于`lapic.c->lapicinit()` 里

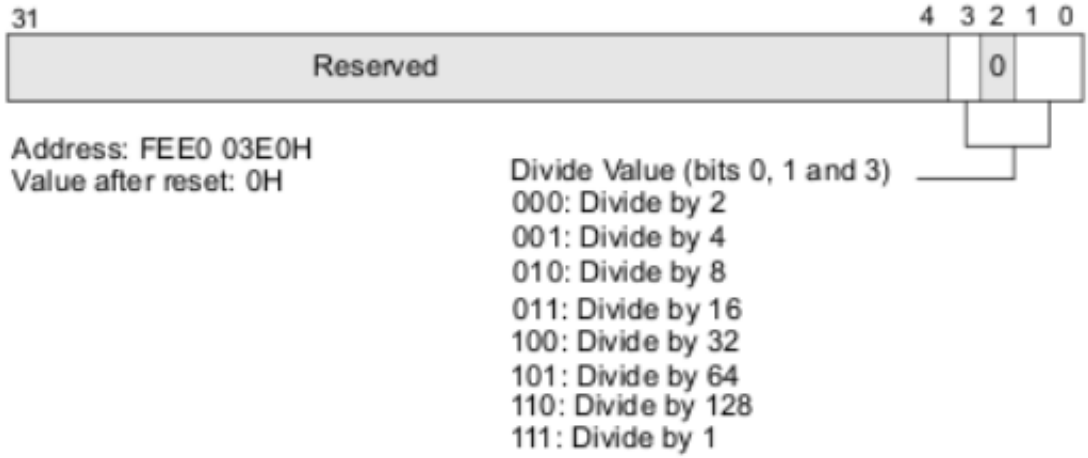
```
  lapicw(TDCR, X1);
  lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
  lapicw(TICR, 10000000);
```

1. 其中，第一行代码设置计数器频率，可以在系统频率上除以一些数得到。*xv6里，向TDCR寄存器写入X1，而X1扩展到0xB，看起来似乎很奇怪，但实际上在x86的定义里，

```
#define             APIC_TDR_DIV_1          0xB
```

0xB就代表系统频率除以1，即使用系统频率计数。（图片来源：https://elixir.bootlin.com/linux/latest/source/arch/x86/include/asm/apicdef.h#L125）

> *关于这一点，可以稍微拓展一下，其实`Divide Configuration Register (TDCR)`的定义如图 （来源：https://github.com/GiantVM/doc/blob/master/interrupt_and_io/IA32_manual_Ch10.md）



因此0xB (1011) 就相当于把第0,1,3位设置为1，即实现`Divide by 1`功能。

2. 第二行设置计数器模式，给时钟中断分配的id（中断向量号）是 `T_IRQ0 + IRQ_TIMER`，同时向17~18位写入01，设置定时器的模式。

> lapicw 函数将第二个参数的值写入第一个参数指示的寄存器中
>
> PERIODIC扩展到0x20000

3. 第三行设置计数次数。**如果软件设置一个非0的32位值到初始计数器TICR中，LAPIC就会把该寄存器的值复制到当前计数寄存器TCCR中，每个时钟计数减1，TCCR为0就触发中断。**

4. 参考文献

1. https://zhuanlan.zhihu.com/p/142345277
2. https://blog.csdn.net/omnispace/article/details/61415994
3. https://zhuanlan.zhihu.com/p/113365730
4. https://elixir.bootlin.com/linux/latest/source/arch/x86/include/asm/apicdef.h#L125
5. https://github.com/GiantVM/doc/blob/master/interrupt_and_io/IA32_manual_Ch10.md

2、时钟中断的处理程序中，ticks变量的作用，tickslock锁的作用

1. ticks变量的作用

在`trap.c`中，对应时钟中断的处理流程如下：

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

可以看到，在获取锁以后，ticks自增，然后调用wakeup唤醒休眠在chan (ticks)上的所有进程，然后释放锁。

```
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;
}
```

因此ticks变量起到了一个计时器的作用，实现依赖时间的进程的休眠与唤醒。

2. 锁的作用

ticks变量是一个全局可见的变量，定义在`trap.c`里，但是在`defs.h`里也有`extern`声明，这意味着这个变量不止在**`trap.c`**中使用。事实上，这个变量还被`sysproc.c`文件里的sys_sleep函数调用

这个函数一直在检查ticks的值，即当前运行时间片总数，那么同一时刻，ticks这个变量是可能被其他CPU上的进程同时访问的。因此属于临界区操作，必须要加上锁来保证对ticks变量的访问正确。

3. 参考文献

1. https://zhuanlan.zhihu.com/p/163797116

3、时钟中断到来后，从硬件读出IDTR寄存器中的IDT地址开始，内核进程切换的调用流程。

1. IDT是个什么东西?

IDT == `interrupt descriptor table (IDT)`，即中断描述符表，中断处理程序的入口就被放在IDT里。

2. 硬件读出idt以后，到跳转到中断处理程序入口前，又发生了什么?

1. 首先应该取出对应的中断描述符表项，检查当前 %cs 寄存器中的CPL是否小于等于中断描述符表项的DPL域（即当前运行级别与中断的运行等级）。

   > 这里可以展开解释一下，x86有4个保护等级，从0到3，等级依次递减，即0为最高特权等级。当前特权等级的信息存放在 %cs 寄存器的CPL域。中断描述符表项的DPL参数规定了对应中断的运行等级，目的是限制用户程序和内核的操作。如果该DPL为0，表示只有最高特权等级，即内核程序才能触发这个例外门。

2. 硬件保存寄存器旧值，加载新的寄存器

   清空 %eflags 的 `IF (interrupt enable flag)`位。加载的寄存器包括%**ss（存放栈顶的段地址），%esp（存储栈指针），%cs（内存代码段区域的入口地址）和%eip（CPU要读取指令的地址）**

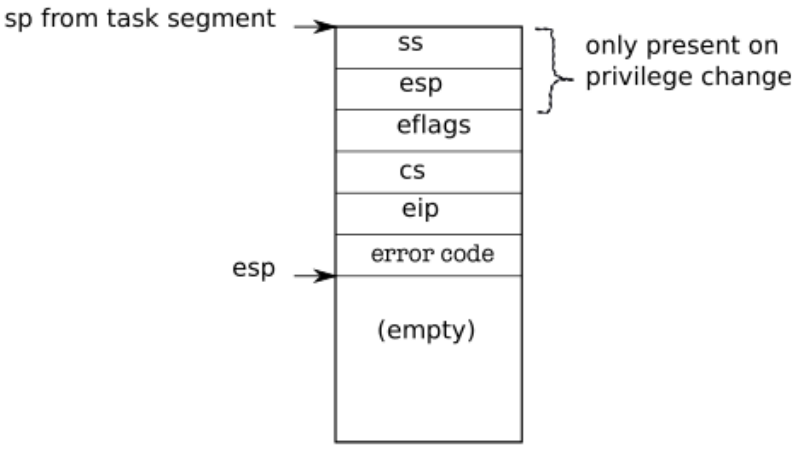
   之后处理器进入IDT entry，开始执行指令。

   > 换句话说，硬件完成了用户栈和内核栈的切换，并且将程序放在了IDT entry入口。

3. 中断处理程序的入口主要包括以下代码：（该代码由 .pl 文件自动生成 .S 文件）

```
.globl vector
vector i:
  pushl $0
  pushl $i
  jmp alltraps
```

中断处理程序将错误码、中断码压入栈，然后跳转到alltraps继续运行。在进入alltraps以前，内核栈应该是这样子的：（图：xv6配套电子文档）



3. 之后软件又做了什么?

1. alltraps 保存中断返回时需要恢复的寄存器（trapframe的结构体）；

   alltraps继续保存寄存器，最终的结果是，内核栈包含了一个trapframe的结构体，里面存储了中断发生时的寄存器，保证了内核返回时，寄存器可以回到中断发生时的样子。

```
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal
```

2. 之后，alltraps设置trap()函数的参数并调用。

```
  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
```

3. trap函数查看中断号 `tf->trapno` 进入相应中断处理。时钟中断的中断号是 `T_TROQ0+TRQ_TIMER`。之后处理函数会获取tickslock，自增ticks，并且检查唤醒进程，之后给LAPIC写EOI（`lapiceoi()`），表示处理完成。

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

4. wakeup，在时钟中断时唤醒ptable.proc数组中依赖chan休眠的所有进程。

   wakeup函数，主要是上锁，调用wakeup1函数，然后释放锁，因此我们只分析wakeup1函数：

   而wakeup1函数就是把chan上的所有SLEEPING的进程状态重设位RUNNABLE。

```
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;
}
```

5. 之后，trap函数还会做别的检查：

   如果当前进程被killed，而且中断发生在用户态，则调用exit函数退出此进程；

   如果当前进程正在运行 RUNNING，而且是时钟中断，则调用yield函数，让当前进程放弃CPU，调度其他的进程运行。

```
  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();

  // 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();
```

6. 在yield函数中，主要操作为获取锁，修改进程状态，调用sched函数切换进程。

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

7. 在sched函数中，主要检查了几个事情：

   - ptable.lock 是否被当前CPU获取，如果不是，可能会出现进程共用一个栈的情况
   - 检查当前CPU的ncli
   - 检查进程的运行状态，不能在RUNNING
   - 检查中断控制位，即eflags的IF位是否被关闭

   这些操作是为了保证原进程被切换过程的正确性。

   同时，sched函数还保存了当前进程的intena，这个值用于判断系统中断的恢复（注释中提到这个性质属于内核线程），而切换进程后会被新进程的intena替代，因此应该提前存储，切换后恢复。

   之后sched函数就会调用swtch函数，**切换到scheduler，进行进程切换**

```
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;
}
```

8. 在swtch函数里，切换上下文，通过ret进入第二个参数指向的进程，时钟中断过程顺序执行到现在，swtch将进入scheduler

   首先复制栈内的两个参数（放到caller-saved寄存器里），然后保存callee-saved寄存器（根据函数调用的约定），之后将栈顶保存到传入的一个参数的地址里（即**&p->context，从而修改p->context），然后将第二个参数（mycpu()->scheduler**）加载到%**esp，即作为栈顶**。之后不断弹出寄存器，最后执行ret指令。

   > 这里需要展开解释一下，参考struct context的定义，会发现这里压栈和弹栈的操作正好与结构体一一对应。
   >
   > ```
   > struct context {
   > uint edi;
   > uint esi;
   > uint ebx;
   > uint ebp;
   > uint eip;
   > };
   > ```
   >
   > ret指令的作用是，处理器从栈中弹出一个字到%eip中，如果注意到上面结构体的定义，就会发现我们弹出的应该是结构体里的eip，而%eip指向的是CPU下一条指令的地址，这样我们就可以进入到新进程的指令里执行。
   >
   > 不过你可能注意到，swtch代码并没有保存eip，这是因为调用swtch函数的指令call会自动将eip压入栈。因此**swtch完成进程切换最重要的步骤就是上文标粗的部分，完成了栈的切换**。

```
.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
```

9. scheduler负责调度进程。

   这个函数不返回，而是一直循环，找到一个RUNABLE的进程进程并调用swtch切换。

   - 循环首先开中断sti()，开中断的原因在下一个问题解答。
   - 然后获取锁，选择一个处于RUNNABLE的进程p，让当前CPU运行进程p，切换page table和p的运行状态（RUNNING），并切换到p的上下文去执行（这时进入到p的指令里执行），scheduler函数只需要等待进程返回就可以。

     ```
     swtch(&(c->scheduler), p->context);
     ```

   从这个切换函数可以看到，第7步的swtch函数事实上会返回到scheduler函数，然后切换到内核的page table，并且继续挑选RUNNABLE的进程切换并运行。

```
  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      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);

  }
```

10. 在切换到新进程之后，（新进程的）函数依次返回，完成调度。

4. 参考文献

1. https://zhuanlan.zhihu.com/p/113696818
2. https://blog.csdn.net/jn1158359135/article/details/7761011
3. https://blog.csdn.net/weixin_42369181/article/details/105463734

4. 为什么scheduler函数要先开中断sti()？明明后面的acquire又关了中断？

scheduler函数的实现逻辑是利用无限循环来查找RUNNABLE的进程，然后用swtch函数切换到RUNNABLE进程来完成。

如果当前没有RUNNABLE进程而且不进行开中断操作？

scheduler会一直处于关闭中断的状态，并且不断循环查找

考虑极端的情况，所有进程都因为等待I/O中断等被挂起，那么scheduler将永远找不到RUNNABLE的进程。

> 你可能会有疑问，release函数在释放锁的同时不会开启中断吗？
>
> 这个问题可以通过查看源代码解决：release函数中开启中断的子函数为popcli()，这个函数并不是直接调用sti()开启中断，而是进行了必要的检查，同时对CPU的属性有条件限制：
>
> ```
> void
> popcli(void)
> {
> if(readeflags()&FL_IF)
>  panic("popcli - interruptible");
> if(--mycpu()->ncli < 0)
>  panic("popcli");
> if(mycpu()->ncli == 0 && mycpu()->intena)
>  sti();
> }
> ```
>
> 其中一个关键的地方在于，intena属性跟获取锁时CPU的中断开启状态有关，即如果获取锁时CPU就没开启中断，那么释放锁的时候，中断也不会被开启。因此如果进入scheduler函数的时候，中断本身就是没有被开启的，那么该函数会一直处于关闭中断的状态。而在进入scheduler函数前，通过例外进入内核，获取锁之后进入scheduler，此时中断已经被关闭，这是常见的操作。

5. yield()函数中的acquire和release操作执行过程中，与一般线程有什么区别?

1. yield函数中的临界区是什么?

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

从前面的分析中，其实可以注意到，yield的临界区是将进程的状态设置为RUNNABLE，然后调用sched函数完成进程的切换。因此进入临界区后，要经过进程的多次切换才能再返回，这个特点和一般线程是不一样的。

2. yield函数调度过程

yield函数中的锁并不是像常规线程保护临界区的操作一样同一时刻仅被一个线程执行。换句话说，在yied函数首先请求锁，然后进行进程调度，要么在完成swtch的上下文保存后，在scheduler中释放锁，要么在被调度进程返回yield函数时完成锁的释放。

另外，在恢复新进程上下文的过程也保证了锁的持有与释放，保证新进程被正确调度。

6. 一个进程从执行exit开始，到占用空间被彻底回收，进程状态经历了怎样的变化?

7.一般由父进程回收子进程的页表和内核栈，如果父进程先于子进程exit，那么谁来回收子进程的相关数据结构?

1. 一个进程退出（exit）后，它的占用空间，如pcb、内核栈、页表等，该由谁来回收?

在xv6里，回收退出进程的空间这个操作，是由父进程去做的。

如果父进程已经退出，那么由initproc即初始的进程来接替父进程的操作。也就是将initproc当成退出进程的父进程，并且exit函数会进行检查，不允许initproc退出。

```
  if(curproc == initproc)
    panic("init exiting");
```

2. exit函数究竟做了什么?

```
  if(curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  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");
```

3. 父进程是怎么捕获到exited的子进程，并且释放其占用空间的呢?

父进程利用wait函数检测子进程状态，当检测到退出的子进程时，释放子进程的占用空间来彻底杀死子进程。

wait返回只有两种情况，找到一个状态为zombie的子进程，或者没有子进程，否则将父进程休眠。

```
  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){
        // Found one.
        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;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
```

4. 总结

进程状态变化：RUNNING->ZOMBIE->UNUSED

8. 请简述xv6如何利用自旋锁(spinlock)和条件变量(sleep&wake)实现wait以及exit的

wait等待条件满足，exit设置满足条件的信号。

具体地说，wait函数等待处于**ZOMBIE**状态的子进程的出现，而exit函数设置子进程的**ZOMBIE**状态并唤醒父进程。

过程中锁的获取与释放

在进入wait函数后，首先会获取锁(ptable.lock)，然后检查进程表，

如检测到存在一个ZOMBIE状态的子进程或无子进程，释放锁并返回，

否则wait函数继续持有锁并调用sleep将当前进程挂起，在进程调度中释放锁。

当子进程进入exit()，获取锁保证父进程唤醒与子进程状态（条件变量）变化同步。

9. xv6中kill操作是如何实现的？进程是在什么时候被真正杀死的？

1.kill与exit的区别

在xv6的说明文档里，exit是进程杀死自己的函数，而kill是赋予其他进程杀死别人的函数。

kill的操作：针对传入的pid，确定要杀死的进程，**将该进程的killed设置为1**。检查如果该进程被挂起（SLEEPING），就唤醒该进程（RUNNABLE），然后退出。

2. kill函数本身并没有直接“杀死”进程，而是做了一个标记killed，在例外处理函数trap里，函数会不断检查当前进程的killed，如果发现标记，那么就会进入exit。

> 既然进程不是立刻被杀死，那么受害者被杀死前是否不影响正常的操作？关于这一点，xv6说明文档中进行了详细的描述，如果感兴趣可以阅读71-72页，关于kill代码的说明。

```
...
if(tf->trapno == T_SYSCALL){
  if(myproc()->killed)
    exit();
  myproc()->tf = tf;
  syscall();
  if(myproc()->killed)
    exit();
  return;
}
...
// Force process exit if it has been killed and is in user space.
// (If it is still executing in the kernel, let it keep running
// until it gets to the regular system call return.)
if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
  exit();
.....
// Check if the process has been killed since we yielded
if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
  exit();
```

3.回顾问题6、7，进程exit，并且由父进程完成相关数据结构回收之后被真正杀死。

10. 为什么调用mycpu函数的时候，需要先关中断？为什么调用myproc函数的时候不需要关中断？

（关于这一点，xv6说明文档中有详细的介绍（65页））

1. mycpu函数是做什么的，中断会对mycpu造成什么影响?

mycpu返回一个cpu结构体指针，这个结构体记录了当前正在这个处理器上运行的进程，处理器的唯一硬件标识符，以及其他的一些信息。

事实上，如果该函数刚好获取到唯一的apcid后，而时钟中断到来，将当前线程调度给一个不同的处理器，那么返回值就不再正确。（返回的结构体指针就不再是当前处理器的信息，而是调度前处理器的信息）

因此，为了保证mycpu返回正确的处理器信息，需要关中断。

```
// Must be called with interrupts disabled to avoid the caller being
// rescheduled between reading lapicid and running through the loop.
struct cpu*
mycpu(void)
{
  int apicid, i;

  if(readeflags()&FL_IF)
    panic("mycpu called with interrupts enabled\n");

  apicid = lapicid();
  // APIC IDs are not guaranteed to be contiguous. Maybe we should have
  // a reverse map, or reserve a register to store &cpus[i].
  for (i = 0; i < ncpu; ++i) {
    if (cpus[i].apicid == apicid)
      return &cpus[i];
  }
  panic("unknown apicid\n");
}
```

2. myproc函数是做什么的，为什么中断不会对myproc函数产生恶劣影响?

myproc函数返回一个proc结构体指针，指向当前处理器正在运行的进程。

myproc函数中，进程信息的获得与mycpu的调用在同一临界区进行。而进入临界区之时当前cpu运行的进程就是需要的myproc，只要mycpu得到的信息是正确的，myproc的信息就自然正确。

```
// Disable interrupts so that we are not rescheduled
// while reading proc from the cpu structure
struct proc*
myproc(void) {
  struct cpu *c;
  struct proc *p;
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
  return p;
}
```

Linux RCU

这一部分要想解释清楚是比较困难的，建议大家多观看官方文档以及网站。

1. 为什么需要RCU（产生原因）？

RCU（Read-copy update）是针对链表数据对象的一种同步机制，于2002年10月添加到linux内核中。

RCU机制读取数据的时候不对链表进行耗时的加锁操作，而且R允许多个线程同时读取该链表，因此读取数据的效率得到提高，并且同一时刻只允许一个线程对链表进行修改（修改时为了避免多线程同时写入冲突，需要加锁）。

> 即RCU supports concurrency between a single updater and multiple readers. (Paul E. McKenney)

这样的特性使得**RCU适用于需要频繁读取数据，而相应修改数据不多的情景**。

比如在文件系统中，经常需要查找定位目录，但是对目录的修改相对来说并不多，这就是RCU发挥作用的合适场景。

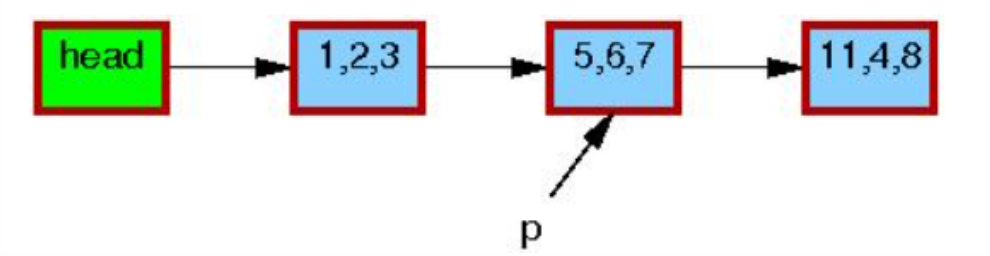
此外，由于读取端不需要加锁，所以RCU对读取端死锁免疫、并且提供了很好的实时延迟。

2. RCU是如何工作的（基本原理）？

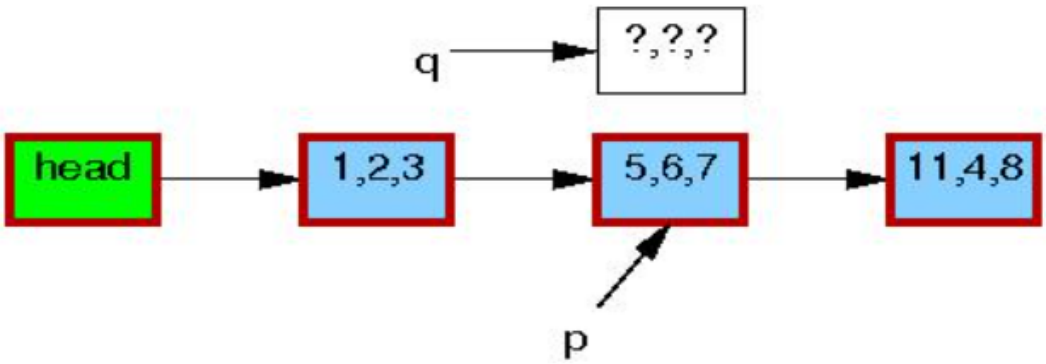
> So the typical RCU update sequence goes something like the following:
>
> a. Remove pointers to a data structure, so that subsequent readers cannot gain a reference to it.
>
> b. Wait for all previous readers to complete their RCU read-side critical sections.
>
> c. At this point, there cannot be any readers who hold references to the data structure, so it now may safely be reclaimed (e.g., kfree()d).
>
> —— 《whatisRCU》

我们以一个对链表的操作代码为例。（下列图片如无特殊说明，均来自https://lwn.net/Articles/262464/）

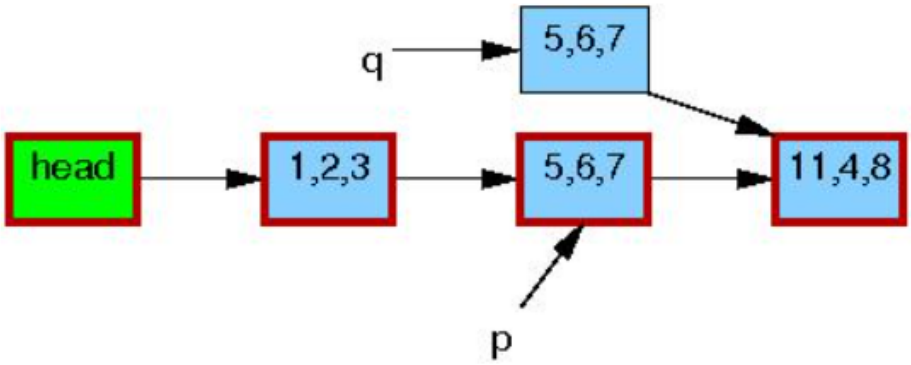
```
1 q = kmalloc(sizeof(*p), GFP_KERNEL);
2 *q = *p;
3 q->b = 2;
4 q->c = 3;
5 list_replace_rcu(&p->list, &q->list);
6 synchronize_rcu();
7 kfree(p);
```



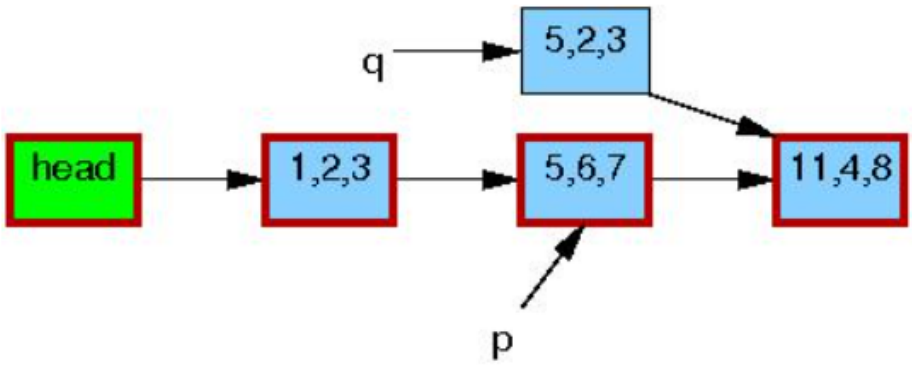

```
1 q = kmalloc(sizeof(*p), GFP_KERNEL);
```



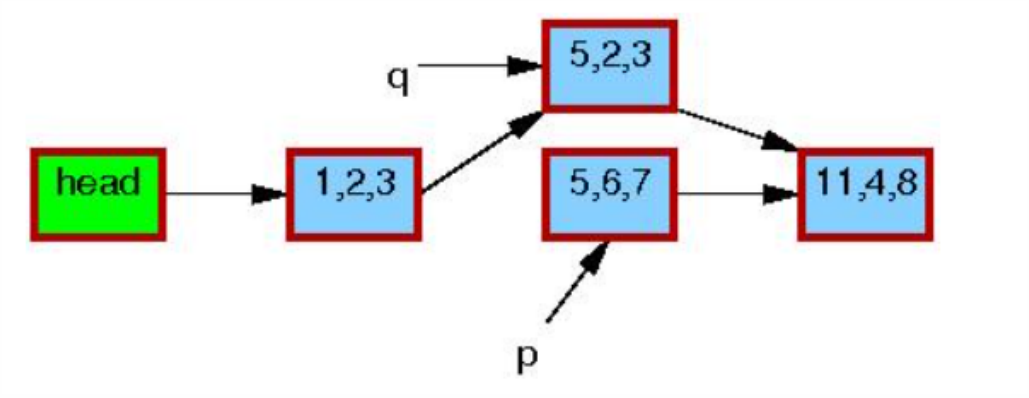

```
2 *q = *p;
```



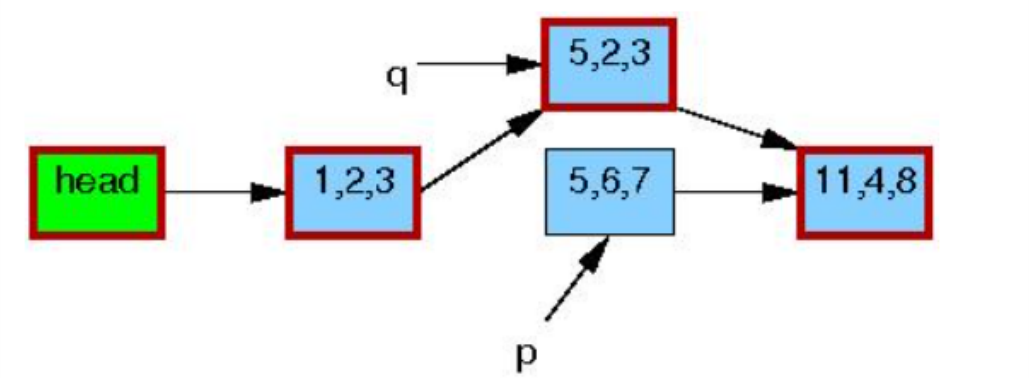

```
3 q->b = 2;
4 q->c = 3;
```



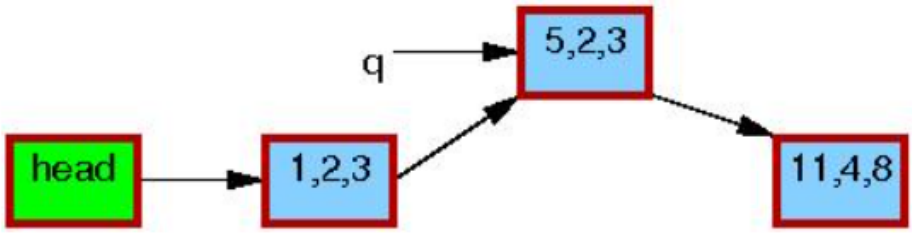

```
5 list_replace_rcu(&p->list, &q->list);
```




```
6 synchronize_rcu();
```

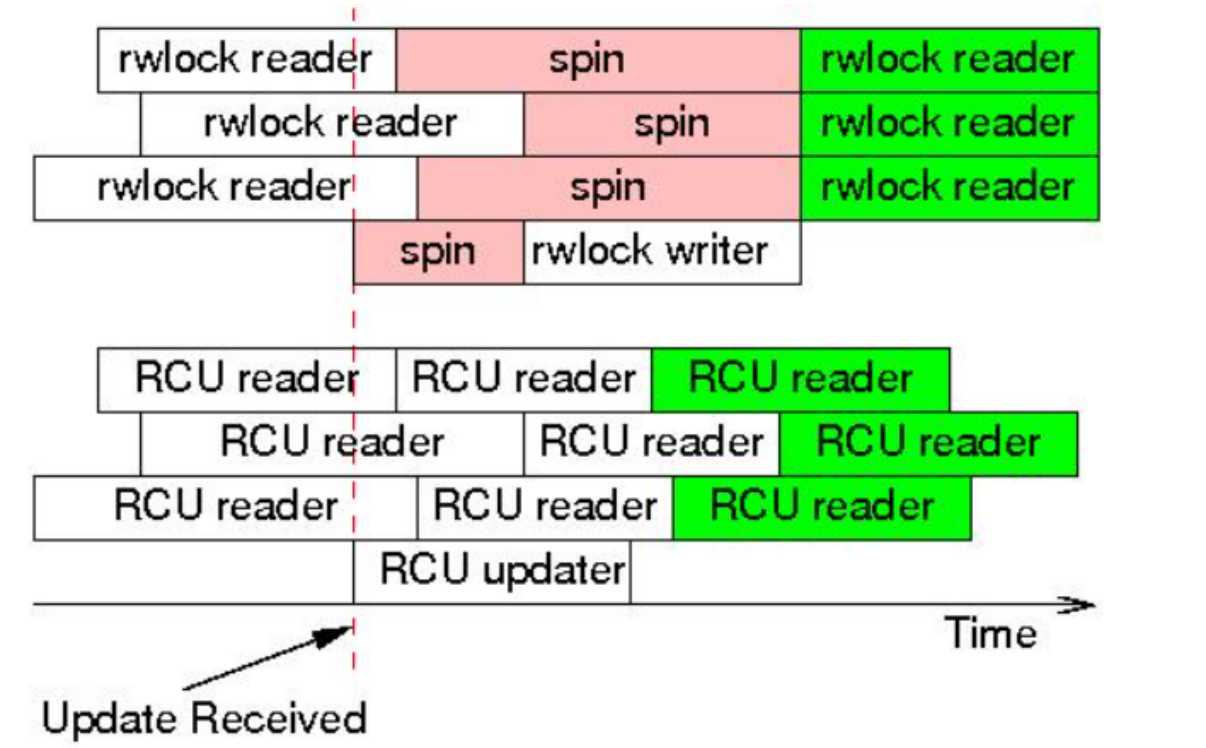



```
7 kfree(p);
```



相比于一般的rwlock，RCU中的reader不需要在写者写数据的时候必须自旋等待了，读者读到的数据的一致性是得到保证的。

> 你可能会有疑惑，读者读到过时的数据，甚至更新前后看到的数据不一致，这是可以接受的吗?
>
> RCU官方文档给出的解答是：“在数量惊人的情况下，不一致和陈旧的数据不是问题。比如网络路由表，路由表更新可能需要相当长时间才能到达给定系统，所以在更新到达时，系统将在相当长的时间内以错误的方式发送数据包。如果使用RCU机制，那么在几毫秒内继续以错误的方式发送更新通常不是问题。此外，RCU读取器很可能比rwlock的方式更早看到更新后的数据：



然而，某些情况下，系统范围内的不一致和陈旧数据是不能容忍的。幸运的是，有很多方法可以避免上述缺点，但是这些讨论超出了本文的范围”

3. RCU的核心API是什么?

根据 `whatisRCU.rst` 的定义，一共有五个核心API，下面我们分别介绍这五个API:

1. `rcu_read_lock()`

   标记数据正在被读者使用，用来告诉回收装置：这个读者正在进入一个`RCU read-side critical section`，保证不被回收装置回收。

2. `rcu_read_unlock()`

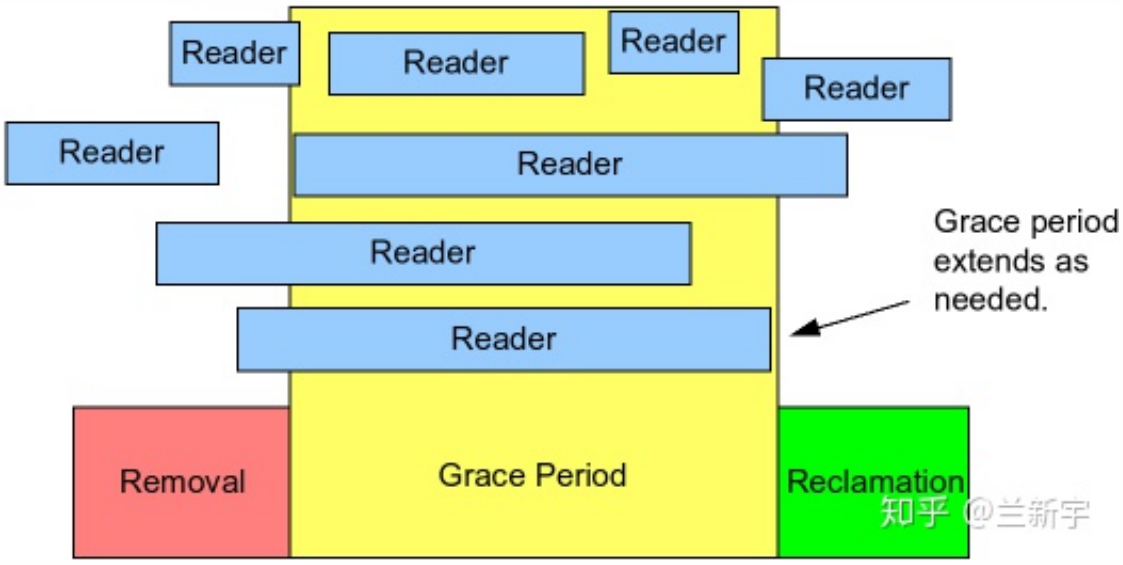
   这个函数也被读者使用，是和第一个API配套的。它告诉回收装置：这个读者正在离开 `RCU read-side critical section` 。

3. `synchronize_rcu()`

   这个API标志着更新者 `updater` 代码的结束以及回收装置 `reclaimer` 代码的开始。它做的事情是，**等待之前的 `RCU read-side critical sections` 完成对旧数据的访问**。

   但是`synchronize_rcu()`的等待方式不够优雅，它是直接阻塞然后等待，类似同步等待回收的效果。
   而`call_rcu`是一种回调函数，当上述等待完成后，该函数会被自动调用，因此CPU可以去做别的事情，类似异步回收的效果。

> 很好理解的是，updater将新的数据链接到链表上（以链表为例），那么之后的读者就直接访问新的数据了；为了保证读者读取数据的一致性，回收装置在开始回收旧数据以前，必须等待正对旧数据进行访问的读者完成自己的工作。`synchronize_rcu`做的正是这样的事情，因此是所谓updater代码的结束和reclaimer代码的开始。
>
> 下面的图片帮助你理解这一同步过程，其中grace period就是等待所有读者对旧数据的访问结束周期（来源：https://zhuanlan.zhihu.com/p/89439043）



4. `rcu_assign_pointer()`

   这个函数用宏实现。`updater`用这个函数来给一个RCU保护的指针赋予新的值，以安全地将这个值的变化从`updater`传递给`reader`。

   通俗来说，`updater`用这个函数来移除旧指针的指向，指向更新后的新节点。

5. `rcu_dereference()`

   这个函数也用宏实现。我们无法给出精确的翻译，因此放出原文：

   > The reader uses rcu_dereference() to fetch an RCU-protected pointer, which returns a value that may then be safely dereferenced. Note that rcu_dereference() does not actually dereference the pointer, instead, it protects the pointer for later dereferencing.
   >
   > ——《WhatisRCU》

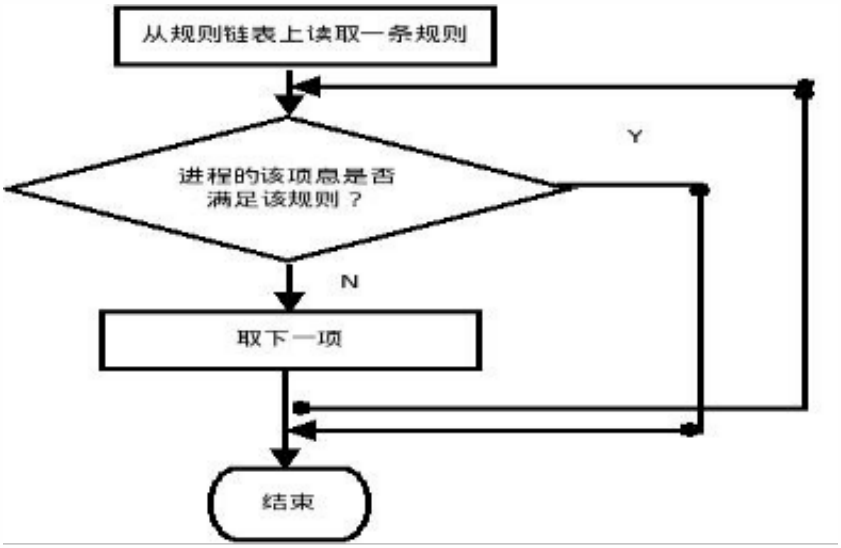
   简单地理解就是，对于RCU保护的指针，需要调用该API才能访问。而且需要在读者临界区内访问，否则回收装置可能将该指针回收，使得访问非法。

   > 举个栗子：
   >
   > ```
   > p = rcu_dereference(head.next);
   > return p->data
   > ```
   >
   > 可以插一嘴，因为编译乱序和执行乱序的问题，上述4，5两个宏都是linux为了确保代码的执行顺序而设计的。感兴趣的小伙伴可以自行查阅官方文档了解更多内容。

4. 程序示例分析

> Example 1: Read-Side Action Taken Outside of Lock, No In-Place Updates

对于`audit_filter_task`函数：这个函数是一个典型的多次读取的函数，它的主要功能如下图：（来源：https://blog.csdn.net/weixin_28717969/article/details/116893930）



如果采用锁的方式实现，那么这个函数大概长这样：

```
static enum audit_state audit_filter_task(struct task_struct *tsk)
  {
    struct audit_entry *e;
    enum audit_state   state;

    read_lock(&auditsc_lock);
    /* Note: audit_netlink_sem held by caller. */
    list_for_each_entry(e, &audit_tsklist, list) {
      if (audit_filter_rules(tsk, &e->rule, NULL, &state)) {
        read_unlock(&auditsc_lock);
        return state;
      }
    }
    read_unlock(&auditsc_lock);
    return AUDIT_BUILD_CONTEXT;
  }
```

即进入读取临界区以前获取锁，完成读取后释放锁等。

而如果使用RCU机制实现，这个函数大概长这个样子：

```
static enum audit_state audit_filter_task(struct task_struct *tsk)
  {
    struct audit_entry *e;
    enum audit_state   state;
    //difference here!
    rcu_read_lock();
    /* Note: audit_netlink_sem held by caller. */
    //difference here!
      list_for_each_entry_rcu(e, &audit_tsklist, list) {
      if (audit_filter_rules(tsk, &e->rule, NULL, &state)) {
        //difference here!
                rcu_read_unlock();
        return state;
```

```
      }
    }
      //difference here!
    rcu_read_unlock();
    return AUDIT_BUILD_CONTEXT;
  }
```

通过RCU设置读者临界区，其中list_for_each_entry_rcu也是一种API，大概是实现了RCU保护机制的遍历。

上述代码都是对读者端的操作，而对updater端，也是类似的：用锁来实现规则的删除和插入：

```
static inline int audit_del_rule(struct audit_rule *rule,
           struct list_head *list)
  {
    struct audit_entry  *e;

    write_lock(&auditsc_lock);
    list_for_each_entry(e, list, list) {
      if (!audit_compare_rule(rule, &e->rule)) {
        list_del(&e->list);
        write_unlock(&auditsc_lock);
        return 0;
      }
    }
    write_unlock(&auditsc_lock);
    return -EFAULT;   /* No matching rule */
  }

  static inline int audit_add_rule(struct audit_entry *entry,
           struct list_head *list)
  {
    write_lock(&auditsc_lock);
    if (entry->rule.flags & AUDIT_PREPEND) {
      entry->rule.flags &= ~AUDIT_PREPEND;
      list_add(&entry->list, list);
    } else {
      list_add_tail(&entry->list, list);
    }
    write_unlock(&auditsc_lock);
    return 0;
  }
```

而如果用RCU机制实现：

```
static inline int audit_del_rule(struct audit_rule *rule,
           struct list_head *list)
  {
    struct audit_entry  *e;

    /* Do not use the _rcu iterator here, since this is the only
     * deletion routine. */
```

```
    /* in this case, all callers hold
   audit_netlink_sem, so no additional locking is required.*/

   list_for_each_entry(e, list, list) {
     if (!audit_compare_rule(rule, &e->rule)) {
               //difference here!
       list_del_rcu(&e->list);
               //difference here!
       call_rcu(&e->rcu, audit_free_rule);
       return 0;
     }
   }
   return -EFAULT;   /* No matching rule */
}

static inline int audit_add_rule(struct audit_entry *entry,
         struct list_head *list)
{
   if (entry->rule.flags & AUDIT_PREPEND) {
     entry->rule.flags &= ~AUDIT_PREPEND;
          //difference here!
     list_add_rcu(&entry->list, list);
   } else {
          //difference here!
     list_add_tail_rcu(&entry->list, list);
   }
   return 0;
}
```

上述代码中出现了我们曾介绍过的核心API，但是还有一些对linux中的双向循环链表做操作的API我们没有介绍，比如`list_add_rcu`，这里统一简单介绍一下：

Category	Primitives	Availability	Overhead
List traversal	`list_for_each_entry_rcu()`	2.5.59	Simple instructions (memory barrier on Alpha)
List update	`list_add_rcu()`	2.5.44	Memory barrier
	`list_add_tail_rcu()`	2.5.44	Memory barrier
	`list_del_rcu()`	2.5.44	Simple instructions
	`list_replace_rcu()`	2.6.9	Memory barrier
	`list_splice_init_rcu()`	2.6.21	Grace-period latency
Hlist traversal	`hlist_for_each_entry_rcu()`	2.6.8	Simple instructions (memory barrier on Alpha)
Hlist update	`hlist_add_after_rcu()`	2.6.14	Memory barrier
	`hlist_add_before_rcu()`	2.6.14	Memory barrier
	`hlist_add_head_rcu()`	2.5.64	Memory barrier
	`hlist_del_rcu()`	2.5.64	Simple instructions
	`hlist_replace_rcu()`	2.6.15	Memory barrier
Pointer traversal	`rcu_dereference()`	2.6.9	Simple instructions (memory barrier on Alpha)
Pointer update	`rcu_assign_pointer()`	2.6.10	Memory barrier

这些API主要实现了RCU保护机制，同时添加了必要的内存屏障。

可以看到相对于用锁实现的函数，RCU实现只需要将必要的锁操作换成RCU机制的操作。

5. 参考文献

1. https://blog.csdn.net/xabc3000/article/details/15335131
2. https://zhuanlan.zhihu.com/p/89439043
3. linux/Documentation/RCU/whatisRCU.rst
4. linux/Documentation/RCU/listRCU.rst
5. https://www.cnblogs.com/LoyenWang/p/12681494.html （我们推荐初学者查看此篇文章了解RCU）
6. https://blog.csdn.net/weixin_28717969/article/details/116893930