xv6源码实例分析第三部分

付轶凡、肖家睿、伊力哈木江·玉苏扑

时钟中断

1、简述时钟中断的注册过程

xv6系统启动后，main.c调用tvinit函数（trap.c）初始化中断描述符表（IDT）

```
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");
}
```

中断描述符表结构体gatedesc在mmu.h中定义如下：

```
struct gatedesc {
  uint off_15_0 : 16;
  uint cs : 16;
  uint args : 5;
  uint rsv1 : 3;
  uint type : 4;
  uint s : 1;
  uint dpl : 2;
  uint p : 1;
  uint off_31_16 : 16;
};
```

分别定义了：

off_15_0	：	代码段偏移量的低16位
cs	：	代码段选择器索引
args	：	置为0表示中断/例外入口
rsv1	：	保留位，需置为0
type	：	中断类型或陷入类型
s	：	置为0
dpl	：	优先级，数值越大优先级越低，中断的优先级为0
p	：	置为1
off_31_16	：	代码段偏移量的高16位

SETGATE函数定义在mmu.h中，传入的五个参数分别为

gate ： 中断/陷入入口

istrap： 等于1为陷入类型、等于0为中断类型

sel ： 中断/陷入处理程序代码段选择器索引值

off ： 中断/陷入处理程序在代码段的偏移量

d ： 优先级

vectors.pl产生vectors.S文件，用来生成中断处理地址存放到vectors[]数组中。

```
#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;
}
```

中断描述符表初始化完毕后，调用idtinit函数将idt表地址加载到寄存器中。

至此就完成了时钟中断的注册过程。

```
void
idtinit(void)
{
  lidt(idt, sizeof(idt));
}
```

trap.c

```
static inline void
lidt(struct gatedesc *p, int size)
{
  volatile ushort pd[3];

  pd[0] = size-1;
  pd[1] = (uint)p;
  pd[2] = (uint)p >> 16;

  asm volatile("lidt (%0)" : : "r" (pd));
}
```

x86.h

时钟中断的处理过程

中断描述表加载和入口建立完成之后，xv6操作系统通过硬件建立一个任务段描述符表，给寄存器%esp赋值并加载一个栈段选择器，并且函数switchuvm()将内核中的用户进程栈的栈顶值存储在栈段描述符表中。当陷入被击发时，如果处理器处于用户模式，它就从栈段描述符表中加载寄存器%esp和%ss，并把本身的%ss和%esp压入新栈中；如果处理器处于内核模式，则什么也不发生。然后，处理器将eflags,%cs和%eip等压入栈中。最后，处理器从相应的IDT入口处加载%eip和%cs。然后，调用vectors.pl文件击发中断描述符表的入口，并且跳转到这个文件中来执行，然后向栈中压入errorcode和trapno。

在vectors.pl文件中，我们看到所有的异常都要跳转到alltraps函数做处理（trapasm.S），alltraps函数构建一个完整的trapframe结构（在x86.h中有相关定义），它将%ds、%es、%fs、%gs和剩余的通用寄存器全部压入栈中，构成了完整的trapframe结构体，这个结构体包含了陷入程序完成后处理器恢复用户进程寄存器值的全部信息。

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

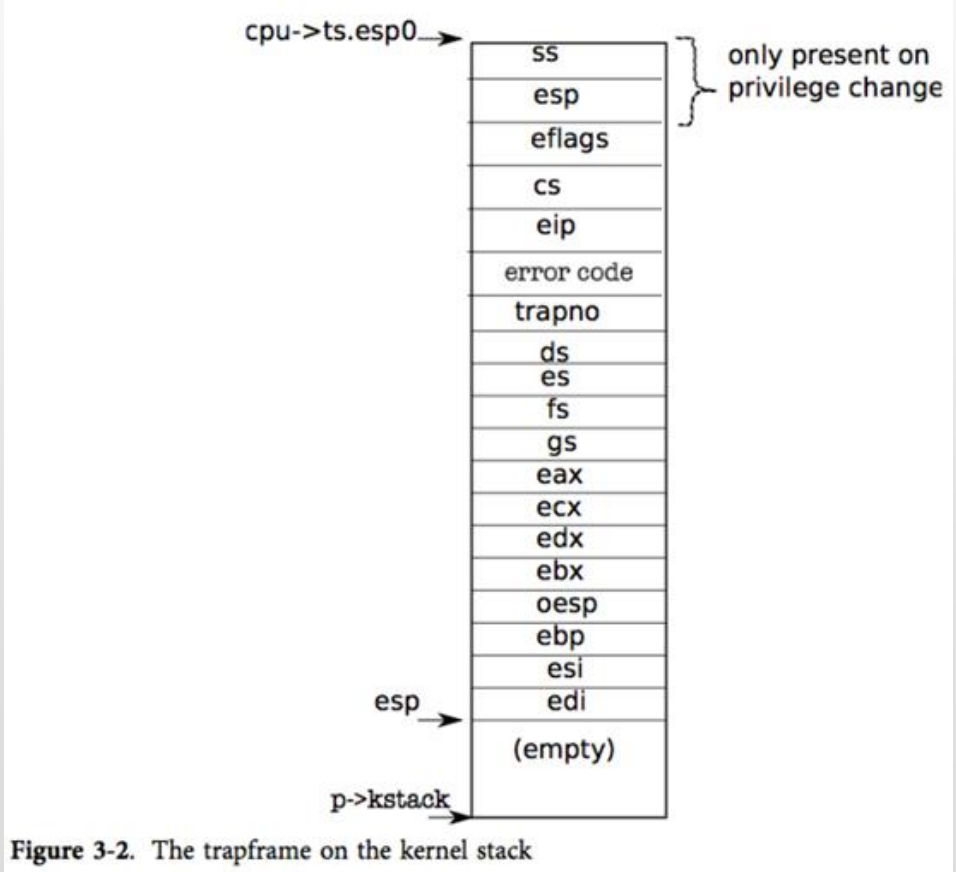


Figure 3-2. The trapframe on the kernel stack

接着，alltraps让寄存器ds,es中含有指向数据段的指针值，建立了数据段。

```
# Set up data segments.
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
```

这时，寄存器esp存储着指向trapframe结构的指针值，它作为trap函数的参数，19行代码表示将esp的值传递给函数trap，20行代码调用trap函数。在trap函数执行完毕返回后，将esp的值加4，此时esp的值指向trapret标签。

```
18    # Call trap(tf), where tf=%esp
19    pushl %esp
20    call trap
21    addl $4, %esp
```

下面就进入了trap函数的调用。对于时钟中断，trap函数处理过程如下：

case T_IRQ0 + IRQ_TIMER是处理时钟中断，如果当前cpu是0号cpu，则将ticks加1，并调用wakeup函数。当中断处理程序在访问ticks变量时，如果另一个CPU上也在进行sys_sleep，同样需要访问ticks，因为tickslock锁的存在，不会导致访问错误。

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

```
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;
}
```

```
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);
}
```

从wakeup和wakeup1函数可以看出ticks的作用是处于睡眠状态的进程，如果它的chan域与ticks相等，就将它从睡眠状态唤醒。

wakeup完成后，返回到trap函数，因为是时钟中断，在后续的检查中，需要调用yield函数切换进程，先由旧进程获得锁，此后yield调用sched，sched调用swtch进行上下文切换。swtch的工作主要包括：1. 保存当前进程的上下文。 2. 加载新进程的上下文到机器寄存器中。这里sched调度的新进程为scheduler调度器。切换到新进程后回到yield，再由新进程释放锁。

```
// 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();
```

```
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

```
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;
}
```

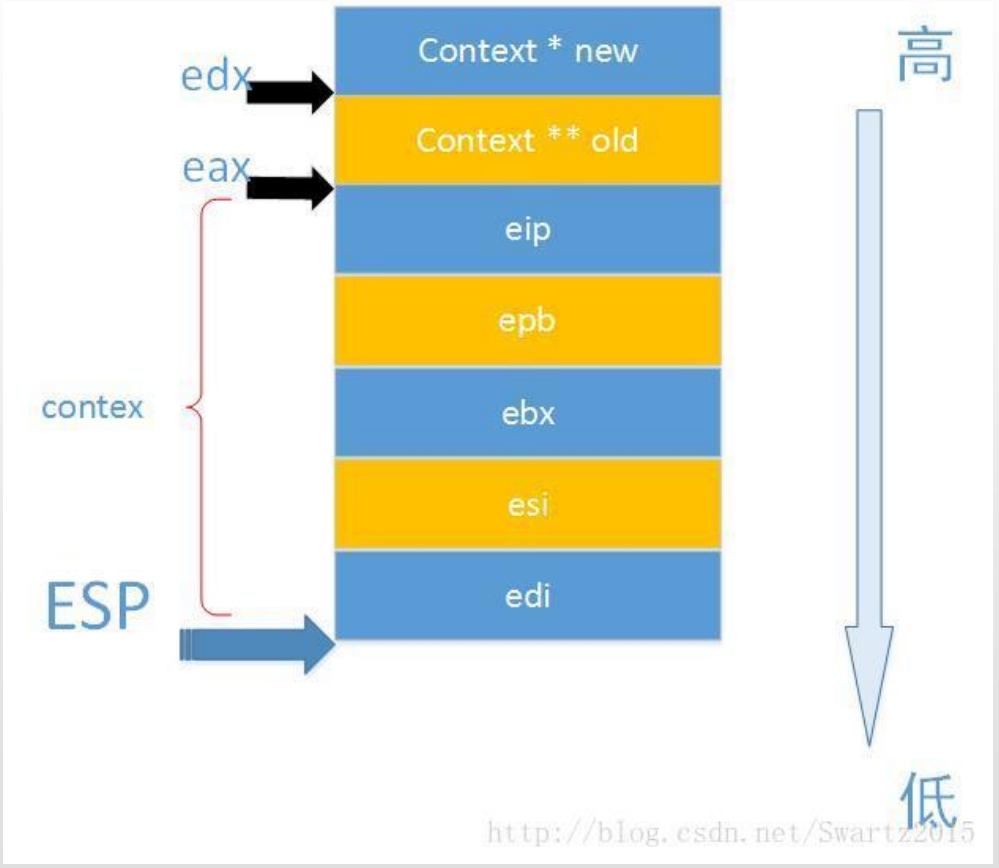
swtch函数执行过程如下：

```
.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切换到scheduler后，由scheduler在进程表中找到一个RUNNABLE的进程，将其状态改为RUNNING，并调用swtch切换到该进程。
scheduler 大部分时间里都持有 ptable.lock，但在每次外层循环中都要释放该锁（并显式地允许中断）。因为当 CPU 闲置（找不到 RUNNABLE 的进程）时，如果一个闲置的调度器一直持有ptable锁，那么其他 CPU 就不可能执行上下文切换或任何和进程相关的系统调用了，也就更不可能将某个进程标记为 RUNNABLE 然后让闲置的调度器能够跳出循环了。而之所以周期性地允许中断，则是因为可能进程都在等待 I/O，从而找不到一个 RUNNABLE 的进程（例如 shell）；如果调度器一直不允许中断，I/O 就永远无法到达了。

切换到新进程后，返回到trap函数，判断是否已经切换到新进程，如果已经切换则退出当前进程，退出后返回到trapasm.S的trapret标签，开始恢复现场，最后通过iret指令返回。

```
// Check if the process has been killed since we yielded
if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
  exit();
```

```
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

Linux RCU 产生原因

为了保护共享数据,需要一些同步机制,如自旋锁(spinlock)，读写锁(rwlock)，它们使用起来非常简单,而且是一种很有效的同步机制，在UNIX系统和Linux系统中得到了广泛的使用。但是随着计算机硬件的快速发展，获得这种锁的开销相对于CPU的速度在成倍地增加，原因很简单，CPU的速度与访问内存的速度差距越来越大，而这种锁使用了原子操作指令，它需要原子地访问内存，也就说获得锁的开销与访存速度相关，另外在大部分非x86架构上获取锁使用了内存栅(Memory Barrier)，这会导致处理器流水线停滞或刷新，因此它的开销相对于CPU速度而言就越来越大。

正是在这种背景下，一个高性能的锁机制RCU呼之欲出，它克服了以上锁的缺点，具有很好的扩展性，但是这种锁机制的使用范围比较窄，它只适用于读多写少的情况，如网络路由表的查询更新、设备状态表的维护、数据结构的延迟释放以及多径I/O设备的维护等。

RCU并不是新的锁机制，它只是对Linux内核而言是新的。早在二十世纪八十年代就有了这种机制，而且在生产系统中使用了这种机制，但这种早期的实现并不太好，在二十世纪九十年代出现了一个比较高效的实现，而在linux中是在开发内核2.5.43中引入该技术的并正式包含在2.6内核中。

基本原理

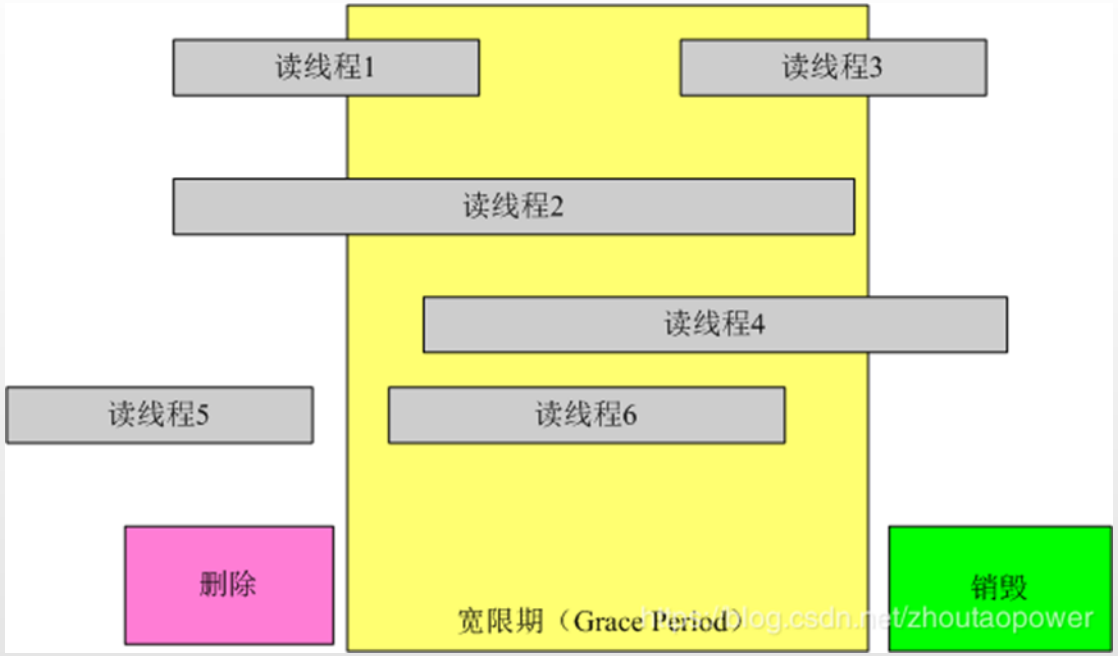
RCU(Read-Copy Update)，顾名思义就是读-拷贝修改，它是基于其原理命名的。对于被RCU保护的共享数据结构，读者不需要获得任何锁就可以访问它，但写者在访问它时首先拷贝一个副本，然后对副本进行修改，最后使用一个回调（callback）机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据。这个时机就是所有引用该数据的CPU都退出对共享数据的操作。

读者在访问被RCU保护的共享数据期间不能被阻塞，这是RCU机制得以实现的一个基本前提，也就说当读者在引用被RCU保护的共享数据期间，读者所在的CPU不能发生上下文切换。写者在访问被RCU保护的共享数据时不需要和读者竞争任何锁，只有在有多于一个写者的情况下需要获得某种锁以与其他写者同步。写者修改数据前首先拷贝一个被修改元素的副本，然后在副本上进行修改，修改完毕后它向垃圾回收器注册一个回调函数以便在适当的时机执行真正的修改操作。等待适当时机的这一时期称为grace period（宽限期），而CPU发生了上下文切换称为经历一个quiescent state（静止状态），grace period就是所有CPU都经历一次quiescent state所需要的等待的时间。垃圾收集器就是在grace period之后调用写者注册的回调函数来完成真正的数据修改或数据释放操作的。

该程序是针对于全局变量 gbl_foo 的操作。假设以下场景。有两个线程同时运行 foo_ read 和 foo_update 的时候，当 foo_ read 执行完赋值操作后，线程发生切换；此时另一个线程开始执行 foo_update 并执行完成。当 foo_ read 运行的进程切换回来后，运行 dosomething 的时候，fp 已经被删除，这将对系统造成危害。为了防止此类事件的发生，RCU里增加了一个新的概念叫宽限期（Grace period）。

```
1  struct foo {
2      int a;
3      char b;
4      long c;
5   };
6
7  DEFINE_SPINLOCK(foo_mutex);
8
9  struct foo *gbl_foo;
10
11 void foo_read (void)
12 {
13      foo *fp = gbl_foo;
14      if ( fp != NULL )
15         dosomething(fp->a, fp->b , fp->c );
16 }
17
18 void foo_update( foo* new_fp )
19 {
20      spin_lock(&foo_mutex);
21      foo *old_fp = gbl_foo;
22      gbl_foo = new_fp;
23      spin_unlock(&foo_mutex);
24      kfee(old_fp);
25 }
```

图中每行代表一个线程，最下面的一行是删除线程，当它执行完删除操作后，线程进入了宽限期。宽限期的意义是，在一个删除动作发生后，它必须等待所有在宽限期开始前已经开始的读线程结束，才可以进行销毁操作。这样做的原因是这些线程有可能读到了要删除的元素。图中的宽限期必须等待1和2结束；而读线程5在宽限期开始前已经结束，不需要考虑；而3,4,6也不需要考虑，因为在宽限期结束后开始后的线程不可能读到已删除的元素。为此RCU机制提供了相应的API来实现这个功能。



核心API

对于reader，RCU的操作包括：

rcu_read_lock：用来标识RCU read side临界区的开始。

rcu_read_unlock：用来标识reader离开RCU read side临界区

rcu_dereference：该接口用来获取RCU protected pointer。reader要访问RCU保护的共享数据，当然要获取RCU protected pointer，然后通过该指针进行dereference的操作。

对于writer，RCU的操作包括：

rcu_assign_pointer：该接口被writer用来进行removal的操作，在witer完成新版本数据分配和更新之后，调用这个接口可以让RCU protected pointer指向RCU protected data。

synchronize_rcu：writer端的操作可以是同步的，也就是说，完成更新操作之后，可以调用该接口函数等待所有在旧版本数据上的reader线程离开临界区，一旦从该函数返回，说明旧的共享数据没有任何引用了，可以直接进行reclaimation的操作。

call_rcu：当然，某些情况下（例如在softirq context中），writer无法阻塞，这时候可以调用call_rcu接口函数，该函数仅仅是注册了callback就直接返回了，在适当的时机会调用callback函数，完成reclaimation的操作。这样的场景其实是分开removal和reclaimation的操作在两个不同的线程中：updater和reclaimer。

链表操作:

ist_add_rcu ： 该函数把链表项new插入到RCU保护的链表head的开头

list_add_tail_rcu：该函数类似于list_add_rcu，它将把新的链表项new添加到被RCU保护的链表的末尾

list_del_rcu：该函数从RCU保护的链表中移走指定的链表项entry

list_replace_rcu：该函数是RCU新添加的函数，并不存在非RCU版本。它使用新的链表项new取代旧的链表项old

list_for_each_entry_rcu ： 该宏用于遍历由RCU保护的链表head

实例分析

```c
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;
}
```

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

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

可以看出对于读端口的转换十分简单，rcu_read_lock（）和rcu_read_unlock（）分别替换read_lock（）和read_unlock（），链表遍历函数使用rcu版本替换就可以了。

```c
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;
}
```

```c
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. */
    list_for_each_entry(e, list, list) {
        if (!audit_compare_rule(rule, &e->rule)) {
            list_del_rcu(&e->list);
            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;
        list_add_rcu(&entry->list, list);
    } else {
        list_add_tail_rcu(&entry->list, list);
    }
    return 0;
}
```

对于链表删除操作，list_del替换为list_del_rcu和call_rcu，这是因为被删除的链表项可能还在被别的读者引用，所以不能立即删除，必须等到所有读者经历一个quiescent state才可以删除。

另外，list_for_each_entry并没有被替换为list_for_each_entry_rcu，这是因为，只有一个写者在做链表删除操作，因此没有必要使用_rcu版本。

通常情况下，write_lock和write_unlock应当分别替换成spin_lock和spin_unlock，但是对于只是对链表进行增加和删除操作而且只有一个写者的写端，在使用了_rcu版本的链表操作API后，rwlock可以完全消除，不需要spinlock来同步读者的访问。对于上面的例子，由于已经有audit_netlink_sem被调用者保持，所以spin_lock就没有必要了。

wait&kill&exit系统调用

```
int
wait(void)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  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
  }
}
```

wait()函数

1.用myproc()获取当前进程，加锁后进入循环

2.查找当前进程的子进程

3.判断子进程是否处于僵尸状态，若是则设置其为UNUSED，并且重置该进程上所有的内容，包括释放栈空间等。

4.返回该子进程的pid

5. 如果没有或者当前进程被杀死了返回-1

6.如果找到了子进程，并且没有一个子进程已经退出，那么就调用 sleep 等待其中一个子进程退出

```
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;
}
```

```
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;
}
```

```
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;
}
```

kill让一个应用程序可以终结其他进程。如果一个进程在进程表中设置被终结的进程 p->killed，如果这个进程在睡眠中则唤醒之。如果被终结的进程正在另一个处理器上运行，它总会通过系统调用或者中断（例如时钟中断）进入内核。当它离开内核时，trap 会检查它的 p->killed，如果被设置了，该进程就会调用 exit，终结自己。
如果被终结的进程在睡眠中，调用 wakeup 会让被终结的进程开始运行并从 sleep 中返回。

```
void
exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  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");
}
```

1.用myproc()获取当前进程，检查是否是初始进程，若是则对控制台输出init exiting再调用exit()自身。

2.然后关闭所有文件。

3.调用iput函数把对当前目录的引用从内存中删除。

4.上锁后唤醒当前进程的父进程，以及以上的进程。

5.把当前进程的所有子进程都划归为用户初始进程initproc的子进程中。

6.把当前进程改为僵尸进程，并从使用进程调度器调度下一个可执行进程。

1.一个进程从执行exit 开始，到最终彻底“消亡”（pcb、内核栈、页表等都被回收），进程状态经历了怎样的变化?

Child process→Zombie→
Parent process释放子进程的页表和内核栈→return 进程pid →
trap 检查pid是否标记为p->killed→kill。

2.一般由父进程A回收子子进程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？

B→Zombie→A→释放子进程B的页表和内核栈
B→Zombie→initproc→释放子进程B的页表和内核栈

```
2653 // Pass abandoned children to init.
2654 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2655 if(p->parent == curproc){
2656 p->parent = initproc;
2657 if(p->state == ZOMBIE)
2658 wakeup1(initproc);
2659 }
2660 }
2661
```

3.xv6中kill 操作是如何实现的？进程是在什么时候真正被“杀死”的？

Tarp根据pid，检查是否为标记成kill，
如果一个进程在进程表中标记
被终结的进程的 p->killed，
如果这个进程在睡眠中则唤醒它。
如果被终结的进程正在另一个处理器上运行，
它总会通过系统调用或者
中断进入内核。
当它离开内核时，trap 会多次检查它的
标记是否为 p->killed，如果被标记了，
该进程就会调用 exit，终结自己。

```
2974 int
2975 kill(int pid)
2976 {
2977 struct proc *p;
2978
2979 acquire(&ptable.lock);
2980 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2981 if(p->pid == pid){
2982 p->killed = 1;
2983 // Wake process from sleep if necessary.
2984 if(p->state == SLEEPING)
2985 p->state = RUNNABLE;
2986 release(&ptable.lock);
2987 return 0;
2988 }
2989 }
2990 release(&ptable.lock);
2991 return -1;
2992 }
```

```
3404 if(myproc()->killed)
3405 exit();
```

```
3468 if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
3469 exit();
```

```
3477 // Check if the process has been killed since we yielded
3478 if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
3479 exit();
3480 }
```

4. 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？

- 如果时间中断了，那就可能引起进程移动到别的CPU进程，返回值可能不是那么准确，因此需要先关掉中断。
- 如果关掉中断，进程会调用mycpu，获取当前进程的指针，从struct CPU里面出去，引起中断，因此不需要关掉中断。

```
2454 // Disable interrupts so that we are not rescheduled
2455 // while reading proc from the cpu structure
2456 struct proc*//获取当前进程
2457 myproc(void) {
2458 struct cpu *c;
2459 struct proc *p;
2460 pushcli();
2461 c = mycpu();
2462 p = c->proc;
2463 popcli();
2464 return p;
2465 }
```

THANK YOU