进程调度

第十五组

汪铭煜 盛子轩 曾锦鸿 甘铠荣

2023/10/11

Table of Contents

总体概览

两次swtch,两种context

流程概览

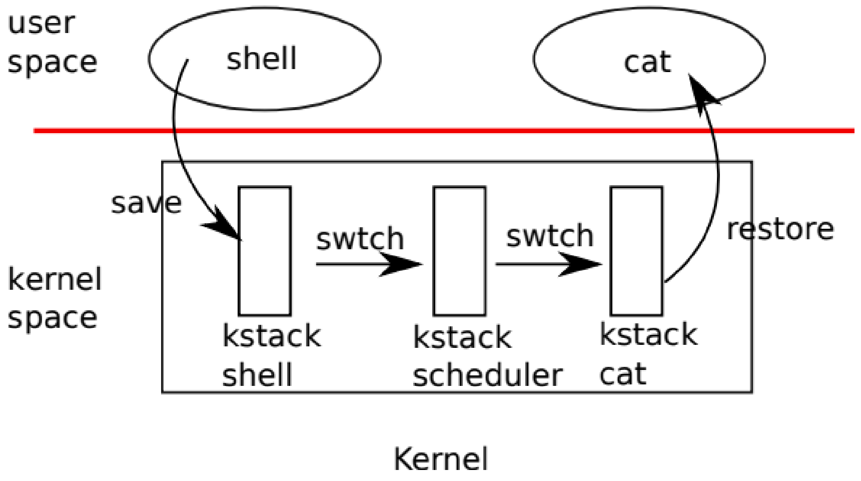
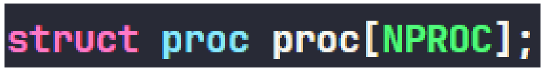


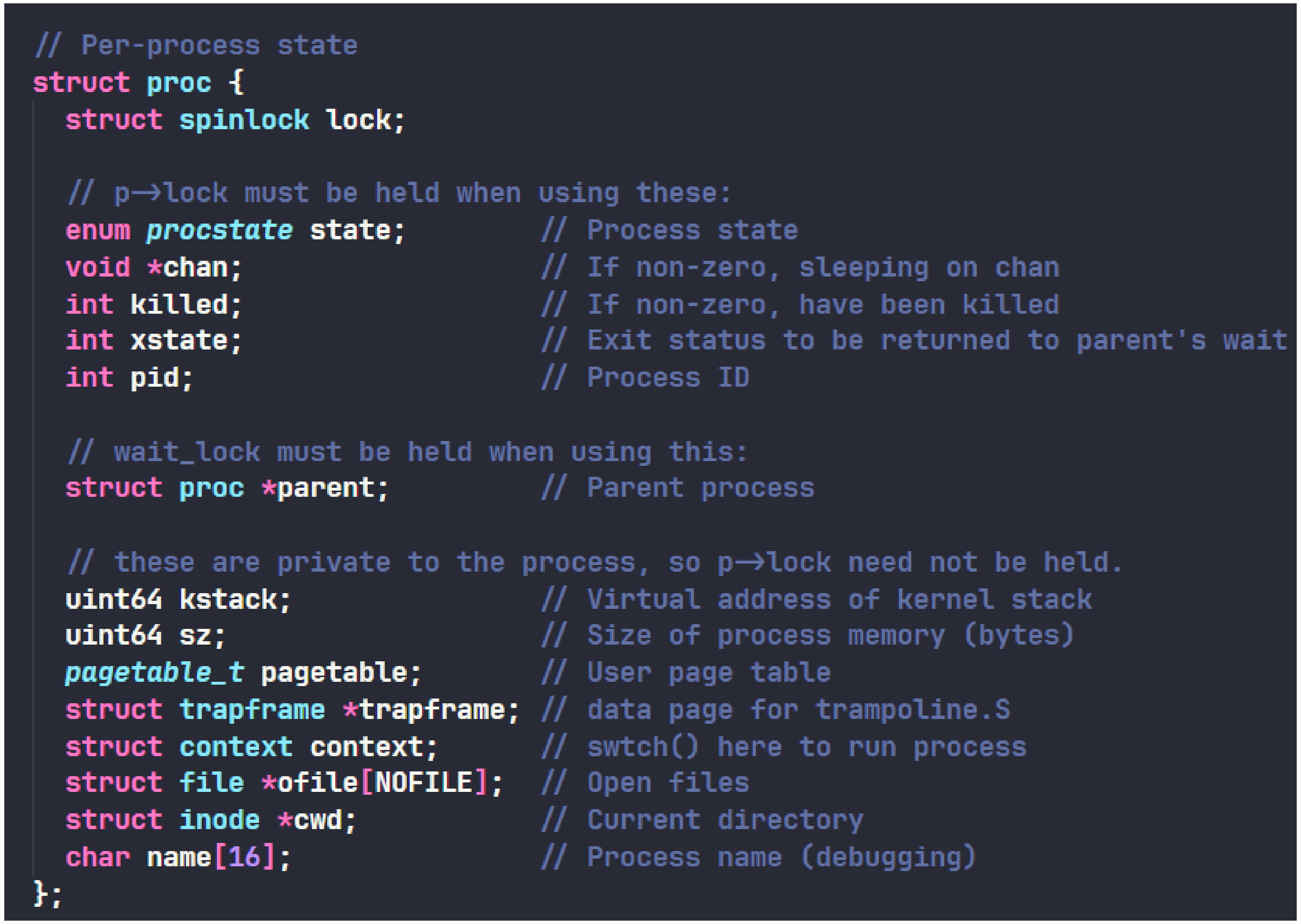
Figure 7.1: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

调度对象：进程

xv6建立一个线程池proc[NPROC](NPROC = 64)



```c
struct proc proc[NPROC];
```



```c
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

调度对象：进程

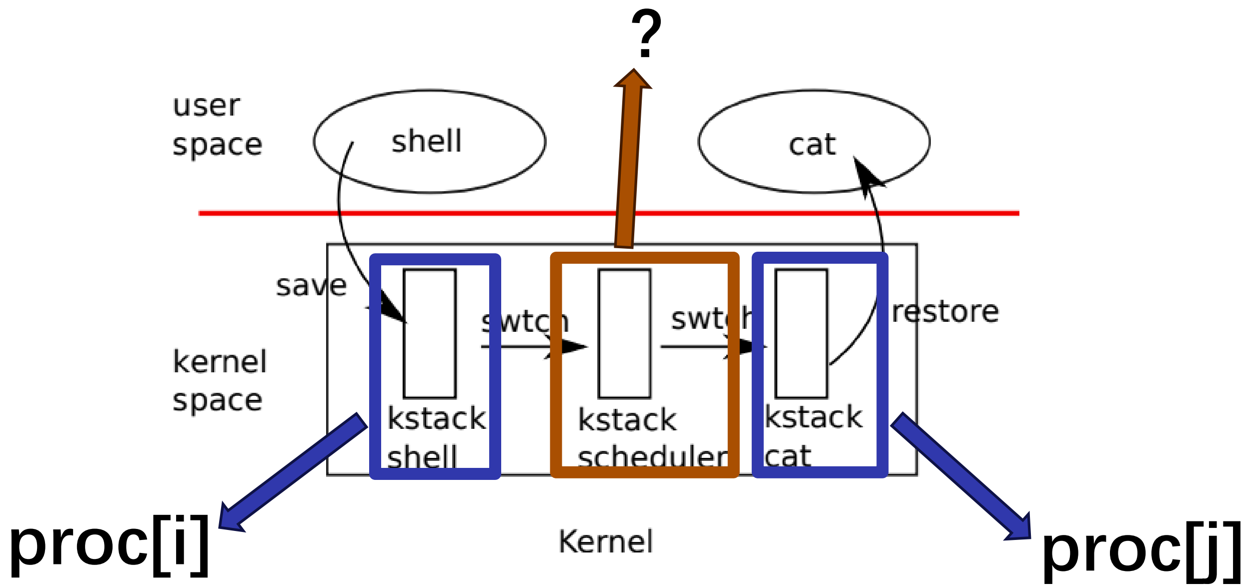
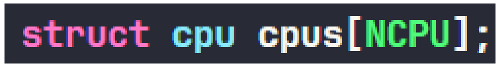


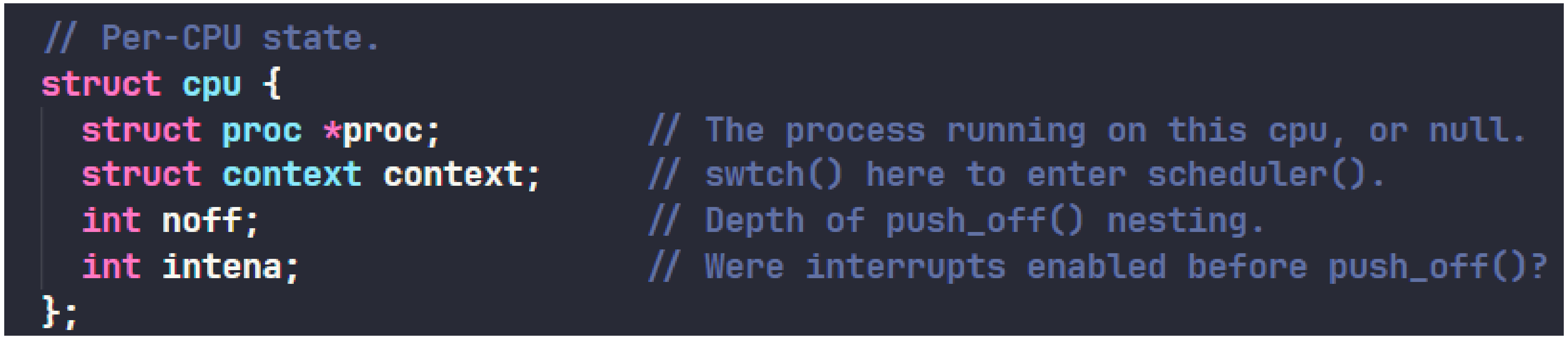
Figure 7.1: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

调度对象：进程

- xv6支持多核，其用cpu结构体来管理cpu。
- 注意到cpu结构体中也存在上下文存储结构体context。
- 这个context存储的就是在该cpu上运行的scheduler()的上下文。



```c
struct cpu cpus[NCPU];
```



```c
// Per-CPU state.
struct cpu {
  struct proc *proc;          // The process running on this cpu, or null.
  struct context context;     // swtch() here to enter scheduler().
  int noff;                   // Depth of push_off() nesting.
  int intena;                 // Were interrupts enabled before push_off()?
};
```

调度对象：进程

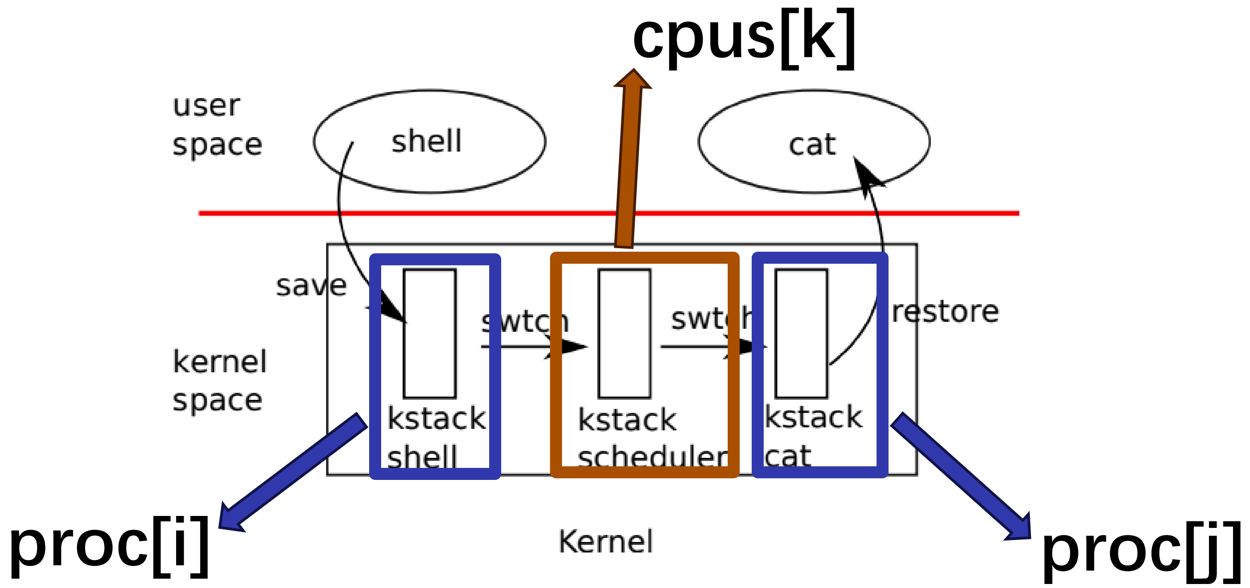


Figure 7.1: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

第一次swtch

yield()和sched()

第一次swtch

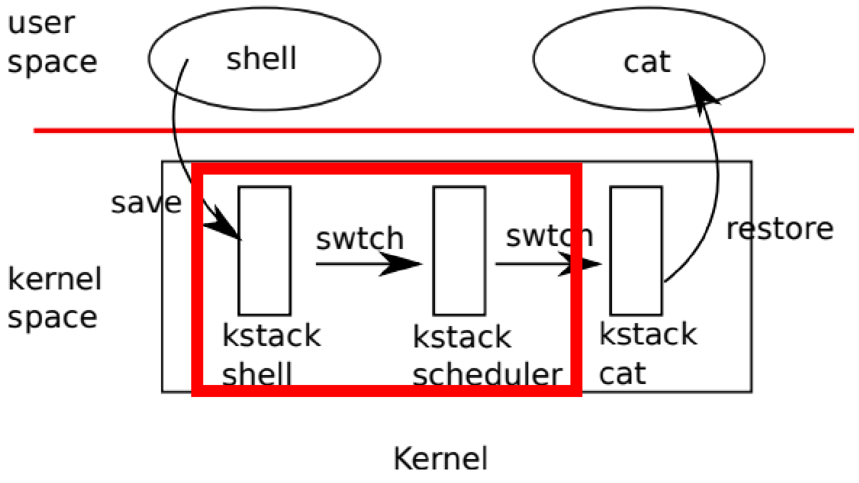
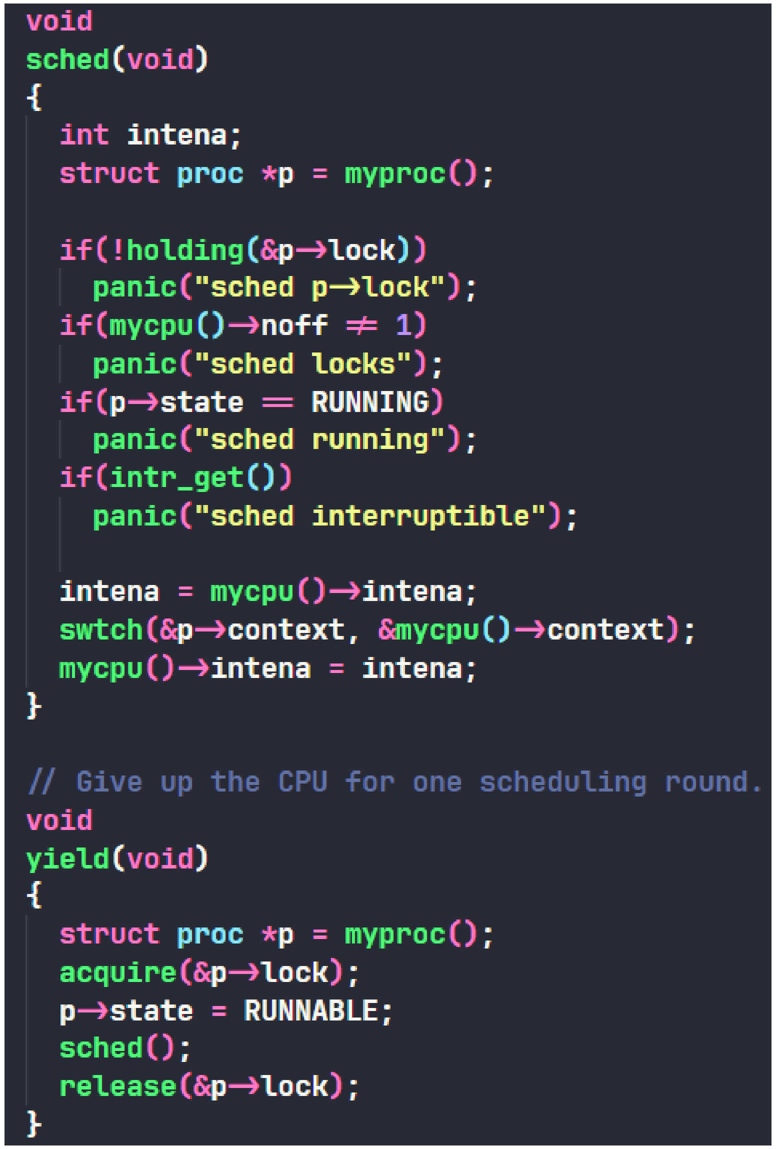


Figure 7.1: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

第一次swtch

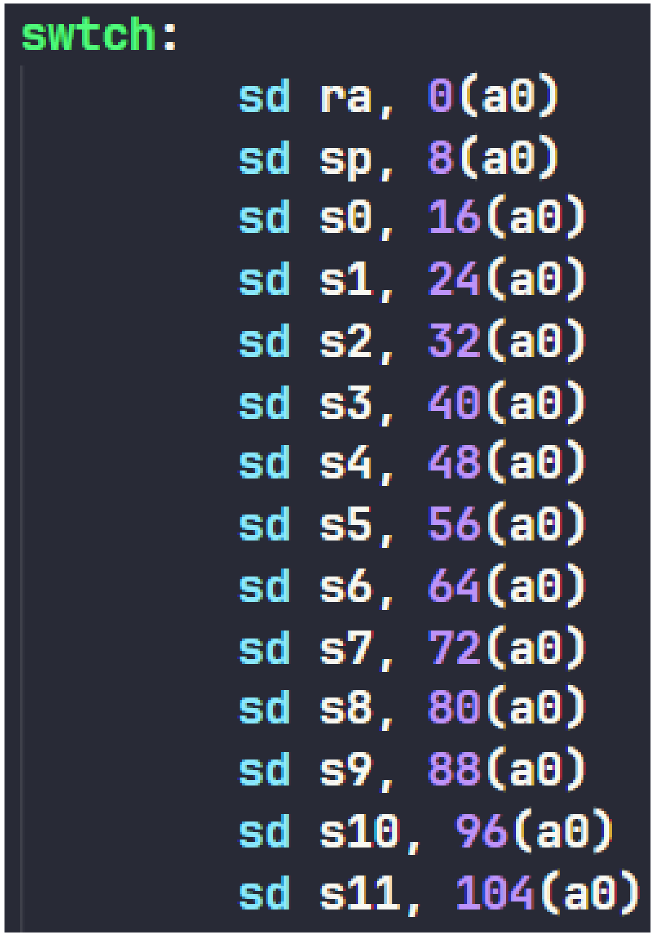


```c
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&p->lock))
    panic("sched p->lock");
  if(mycpu()->noff != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(intr_get())
    panic("sched interruptible");

  intena = mycpu()->intena;
  swtch(&p->context, &mycpu()->context);
  mycpu()->intena = intena;
}

// Give up the CPU for one scheduling round.
void
yield(void)
{
  struct proc *p = myproc();
  acquire(&p->lock);
  p->state = RUNNABLE;
  sched();
  release(&p->lock);
}
```

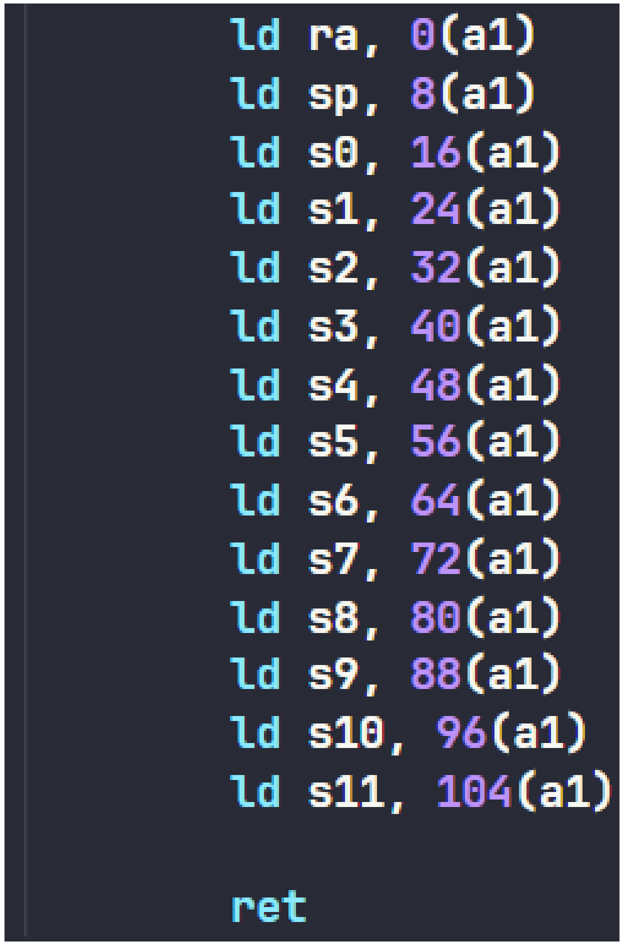
- sched()函数首先检查该进程持有锁，中断嵌套深度为1，不处于RUNNING状态，中断已被关闭(有一个条件不满足就panic，错误返回)；然后记录中断信息，同时从当前进程的上下文swtch到与cpu绑定的调度算法的上下文

- yield()主要负责加锁，并且改掉当前进程的状态，然后调用sched()函数**(这里可以注意一下，yield()函数中的加解锁并不是配套的，而是与schedular中的加解锁配套)**

swtch



```
swtch:
        sd ra, 0(a0)
        sd sp, 8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        sd s2, 32(a0)
        sd s3, 40(a0)
        sd s4, 48(a0)
        sd s5, 56(a0)
        sd s6, 64(a0)
        sd s7, 72(a0)
        sd s8, 80(a0)
        sd s9, 88(a0)
        sd s10, 96(a0)
        sd s11, 104(a0)
```

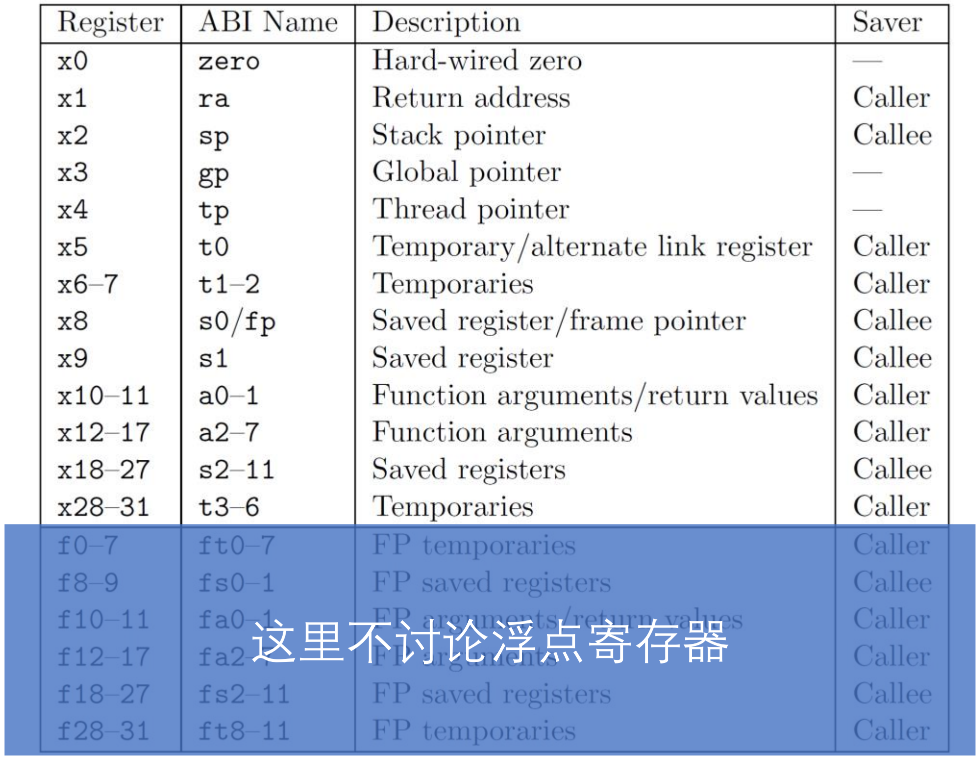


```
        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ld s2, 32(a1)
        ld s3, 40(a1)
        ld s4, 48(a1)
        ld s5, 56(a1)
        ld s6, 64(a1)
        ld s7, 72(a1)
        ld s8, 80(a1)
        ld s9, 88(a1)
        ld s10, 96(a1)
        ld s11, 104(a1)

        ret
```

swtch()

- swtch()保存了旧进程的上下文，恢复了新进程的上下文。
- 通过上下文的切换使得旧进程调用swtch()，返回的时候已经是新进程在执行了。
- swtch()保存了ra, sp, s0-s11
- **为什么只要保存这些寄存器?**

RISC-V寄存器



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

这里不讨论浮点寄存器

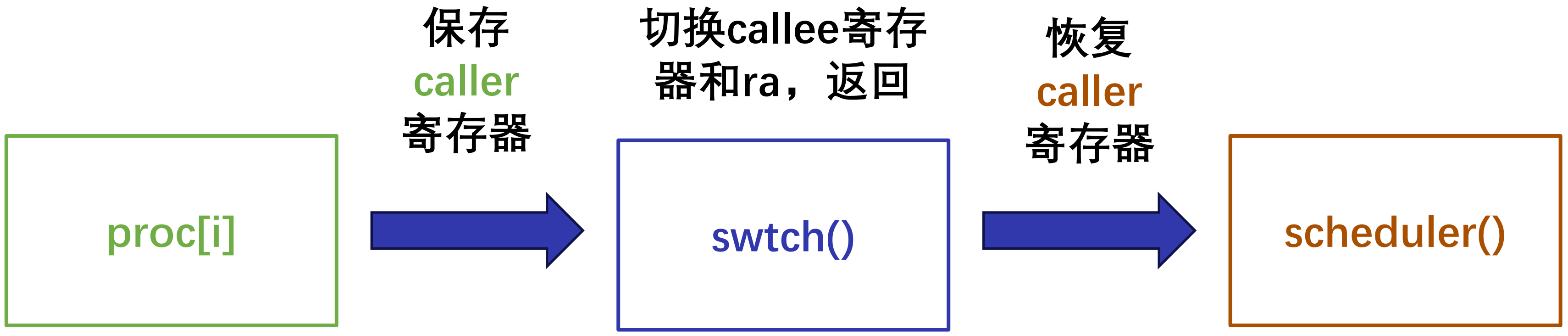
Table 25.1: Assembler mnemonics for RISC-V integer and floating-point registers, and their role in the first standard calling convention.

RISC-V寄存器

- caller saved寄存器： 函数调用中caller需要保存的寄存器（默认callee会使用这些寄存器）。
- callee saved寄存器： 在caller视角中调用前后不变的寄存器， 如果callee需要使用， 需要要提前保存并在返回前恢复

- swtch()保存了ra和callee saved寄存器， 按照函数调用规范， 我们在c代码中调用swtch()的时候编译器就自动保存了caller寄存器，并在swtch()返回后自动恢复caller寄存器。

寄存器行为图


proc[i]
保存 caller 寄存器
切换callee寄存器和ra，返回
swtch()
恢复 caller 寄存器
scheduler()

第一次swtch

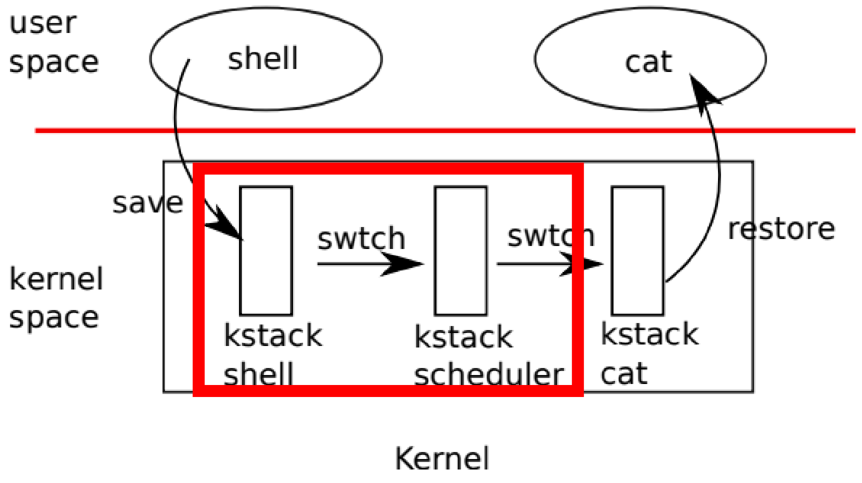


Figure 7.1: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

第二次swtch

scheduler()

第二次swtch

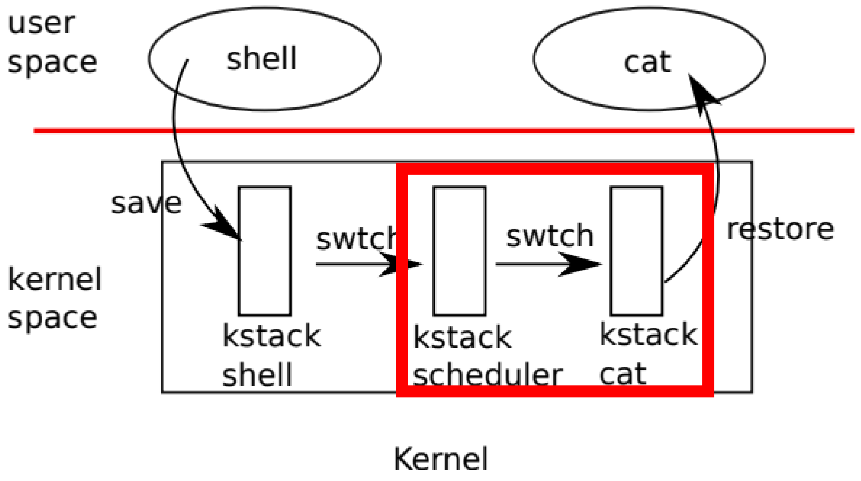
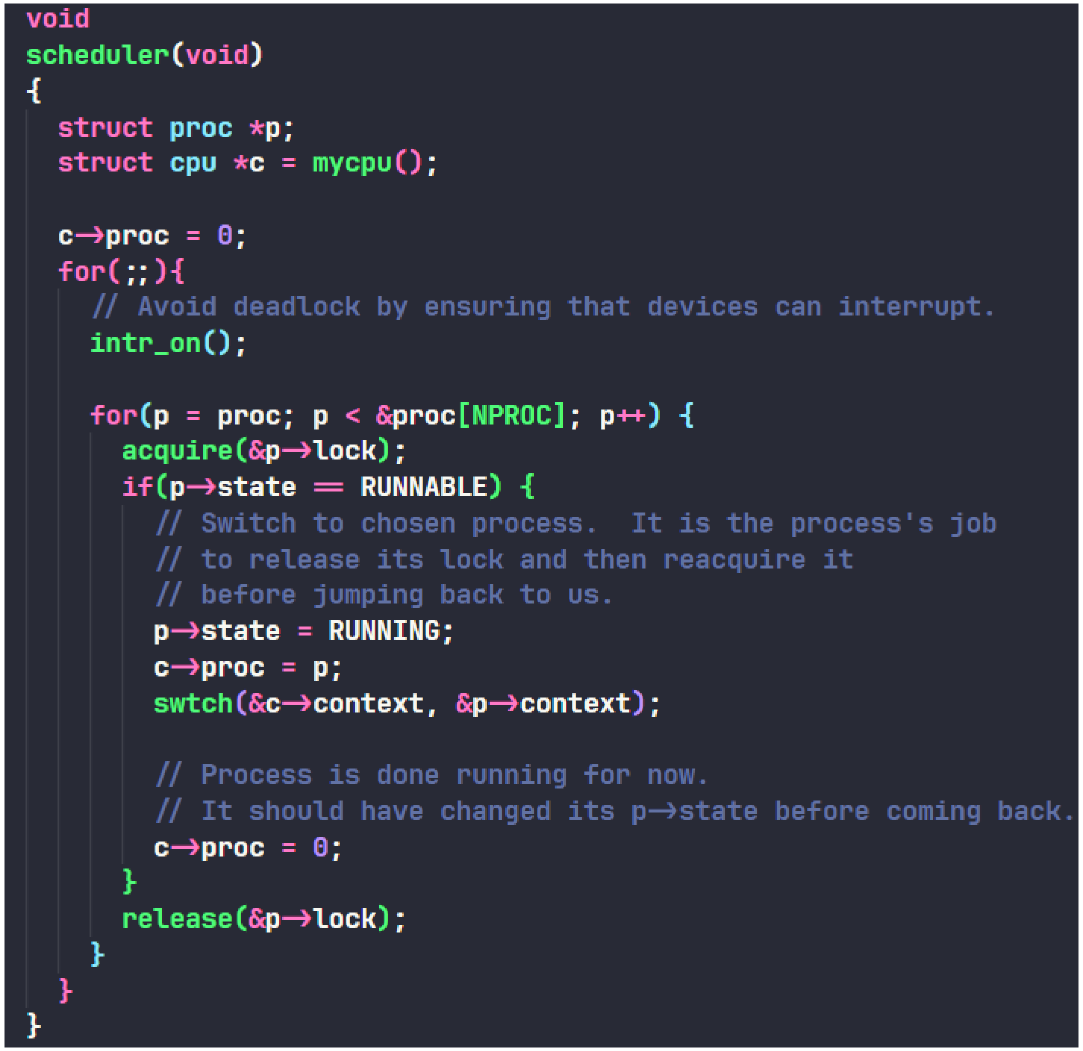


Figure 7.1: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

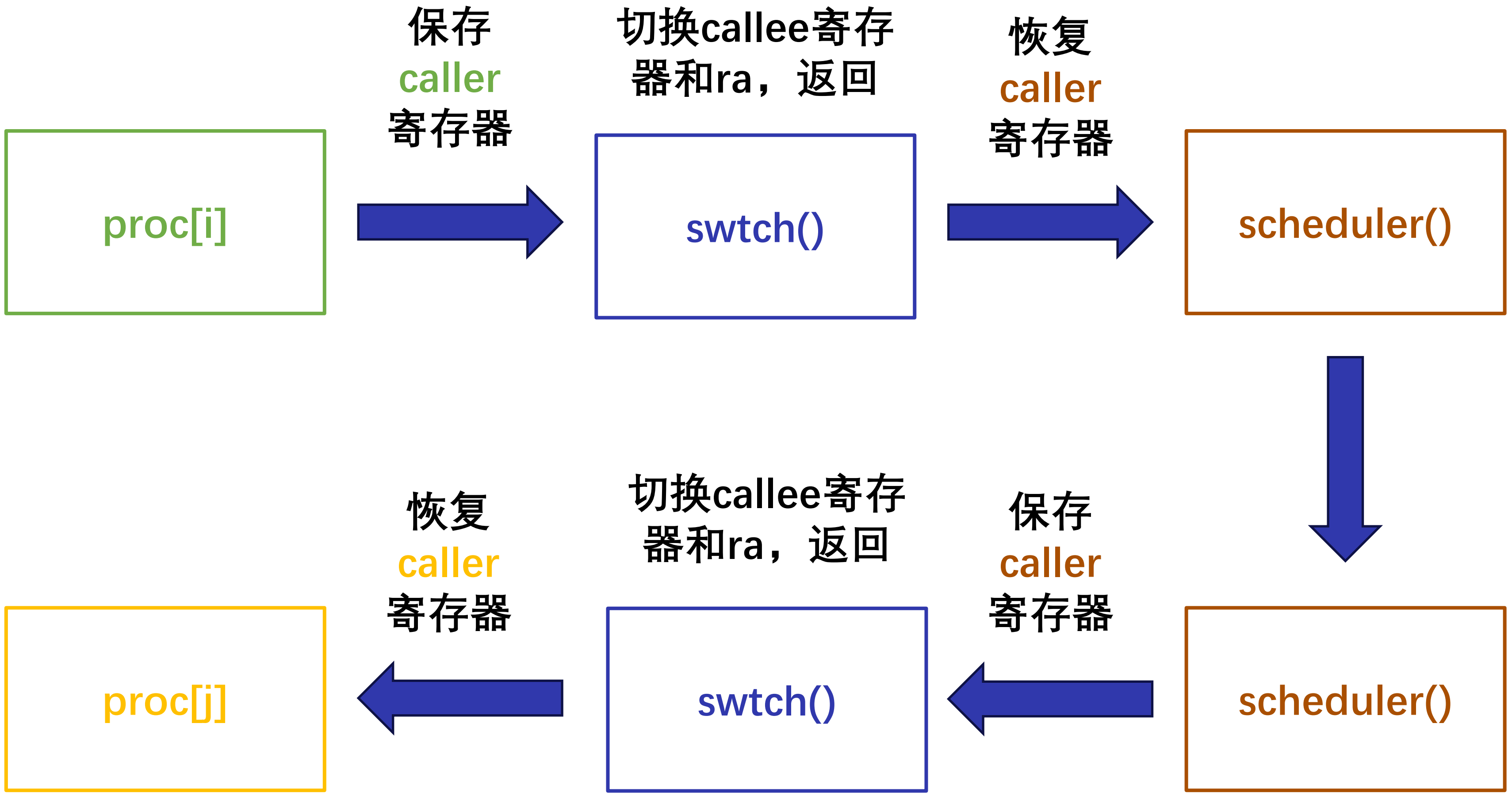
scheduler()



```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);

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

- 读取当前的cpu并初始化
- 进入死循环，同时允许中断避免锁。
- **获取锁**。
- 遍历进程池，找到RUNNABLE的进程，修改其状态为RUNNING，设置当前cpu的进程为该进程。swtch到该进程开始运行。
- **从进程返回**之后修改当前cpu进程为0。
- **释放锁**。

寄存器行为图


proc[i]
保存 caller 寄存器
切换callee寄存器和ra，返回
swtch()
恢复 caller 寄存器
scheduler()
scheduler()
保存 caller 寄存器
切换callee寄存器和ra，返回
swtch()
恢复 caller 寄存器
proc[j]

流程概览

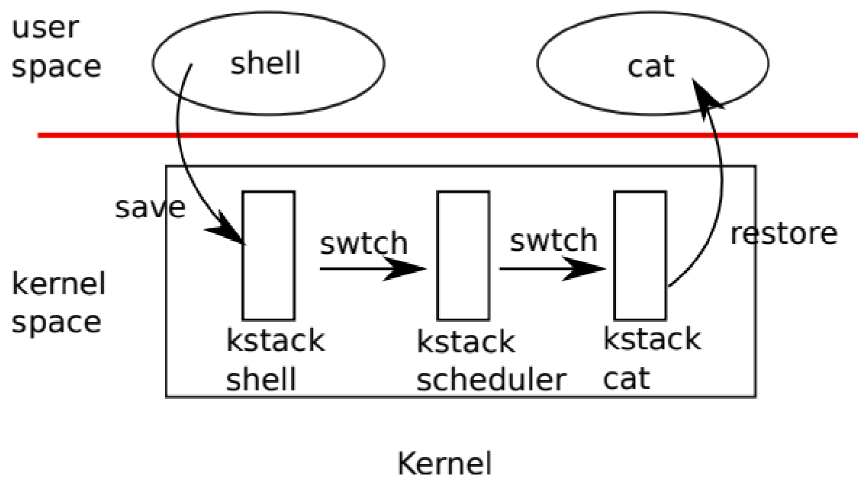


Figure 7.1: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

细化分析

单步跟踪

流程概览

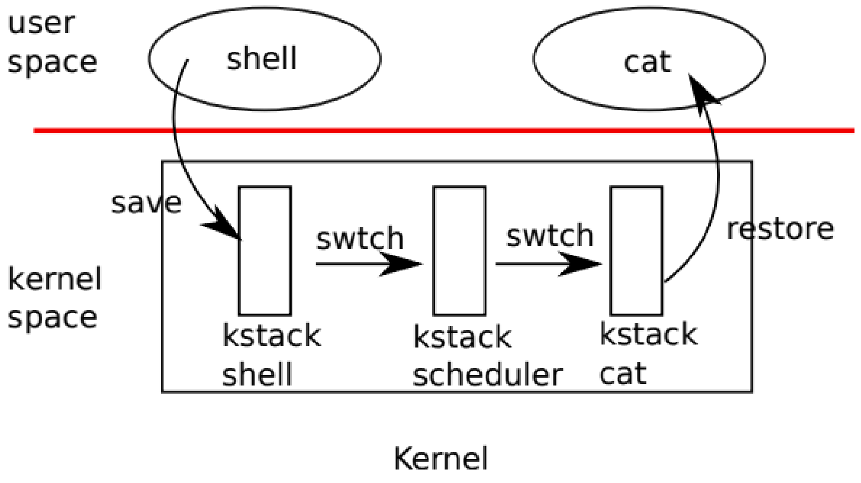
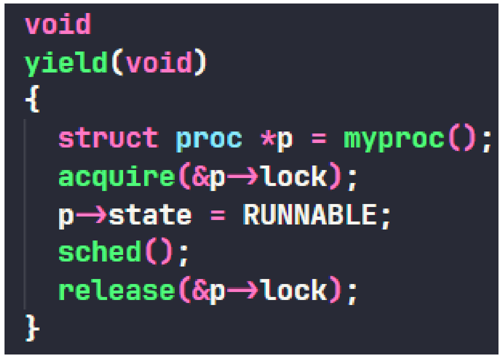


Figure 7.1: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

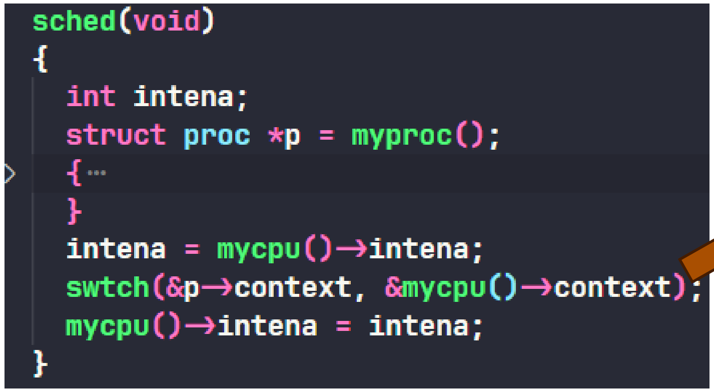
swtch()返回地址

- 前面提到过旧进程调用swtch()，返回的时候已经是新进程在执行了。
- 到底返回到哪里？

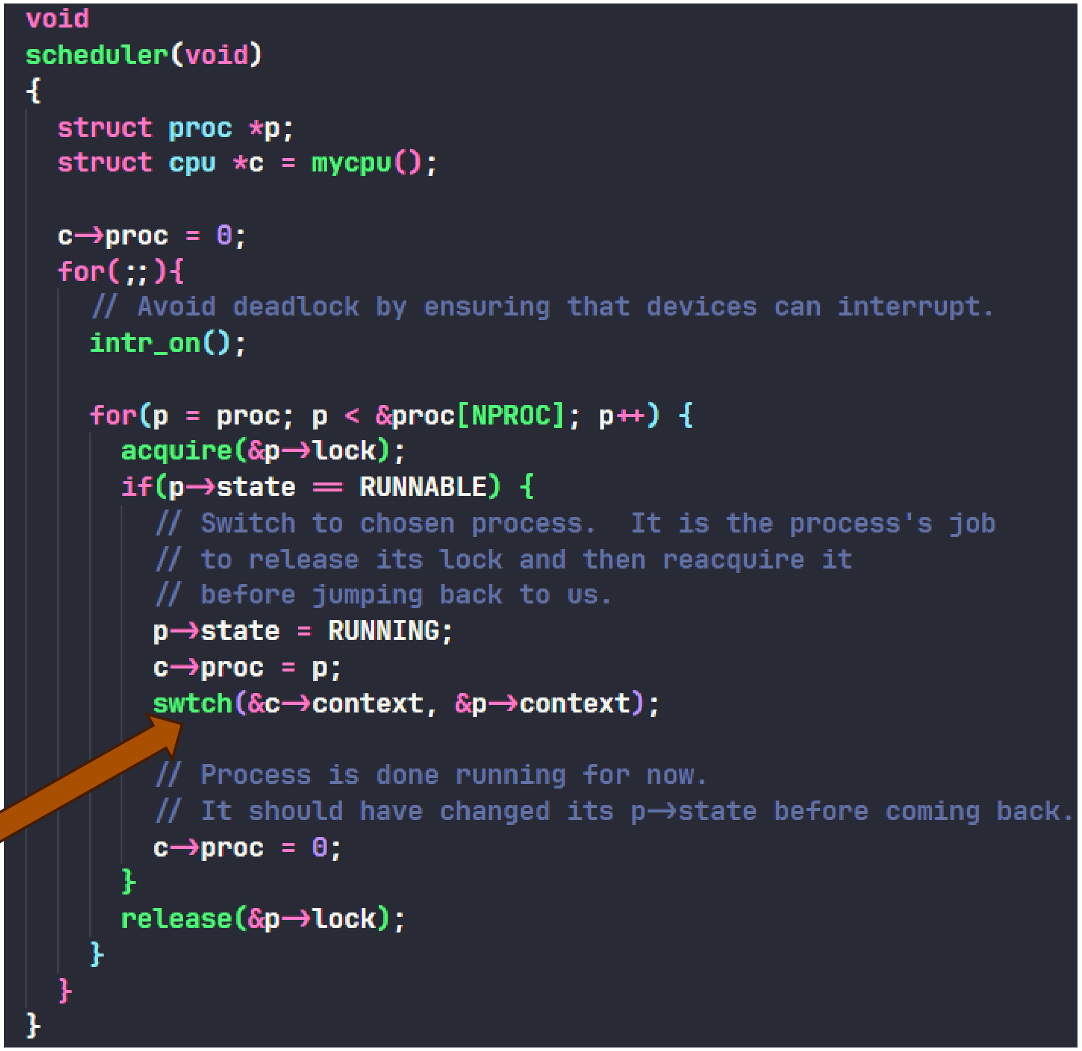
swtch()返回地址



```c
void
yield(void)
{
  struct proc *p = myproc();
  acquire(&p->lock);
  p->state = RUNNABLE;
  sched();
  release(&p->lock);
}
```

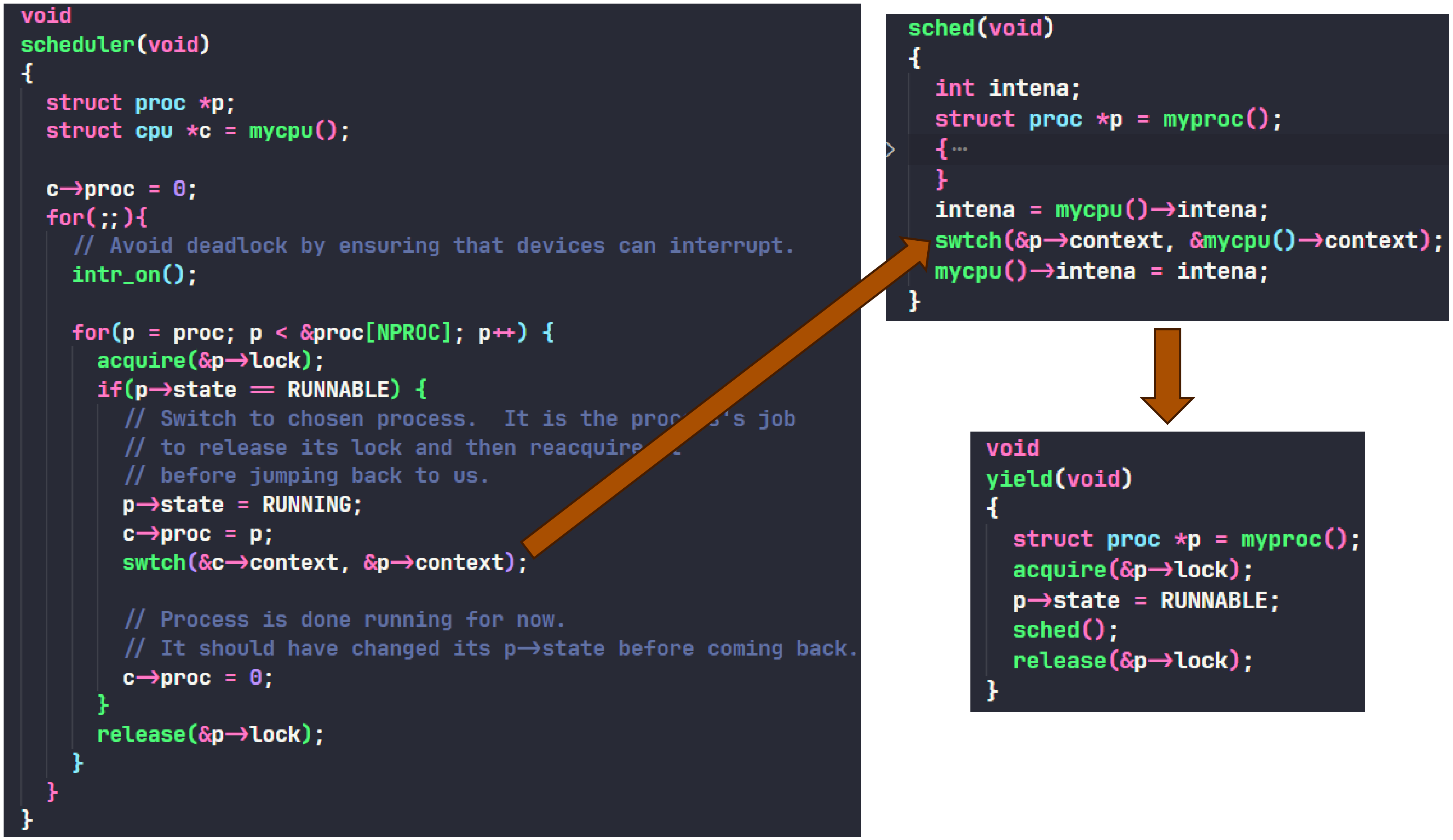


```c
sched(void)
{
  int intena;
  struct proc *p = myproc();
  {…
  }
  intena = mycpu()->intena;
  swtch(&p->context, &mycpu()->context);
  mycpu()->intena = intena;
}
```



```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);

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

swtch()返回地址



```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);

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

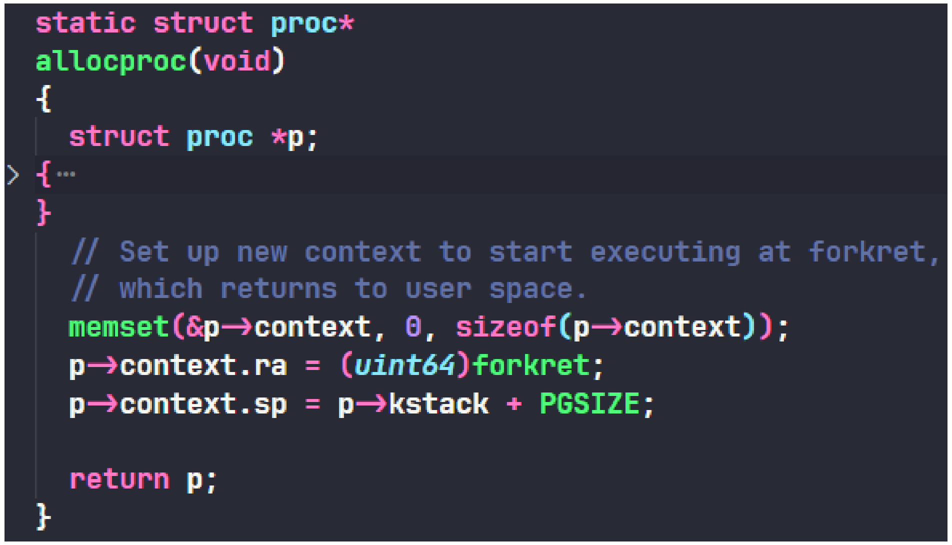
```c
sched(void)
{
  int intena;
  struct proc *p = myproc();
  {…
  }
  intena = mycpu()->intena;
  swtch(&p->context, &mycpu()->context);
  mycpu()->intena = intena;
}
```

```c
void
yield(void)
{
  struct proc *p = myproc();
  acquire(&p->lock);
  p->state = RUNNABLE;
  sched();
  release(&p->lock);
}
```

进程第一次swtch

- 由此看来swtch()的返回地址和上一次与该进程有关的swtch()相关。
- 那么问题来了，第一次swtch()的时候怎么办，假现场。

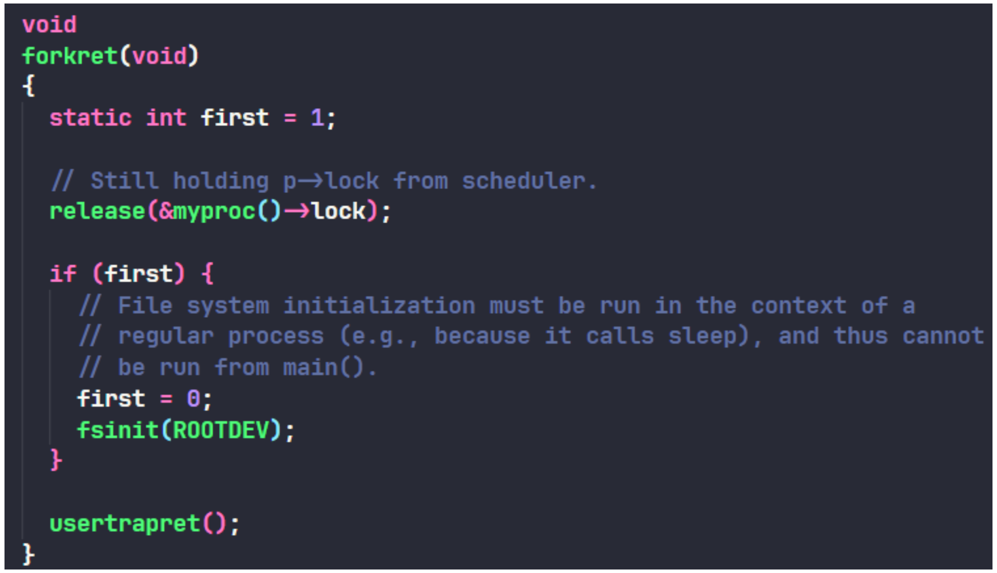
假现场



```c
static struct proc*
allocproc(void)
{
  struct proc *p;
{…
}
  // Set up new context to start executing at forkret,
  // which returns to user space.
  memset(&p->context, 0, sizeof(p->context));
  p->context.ra = (uint64)forkret;
  p->context.sp = p->kstack + PGSIZE;

  return p;
}
```

- 在初始化pcb时已经调用allocproc()函数为每一个PCB做了一个＂假现场＂
- 这使得每个用户进程在初始时能够从“假现场”开始运行也就是forkret()

假现场



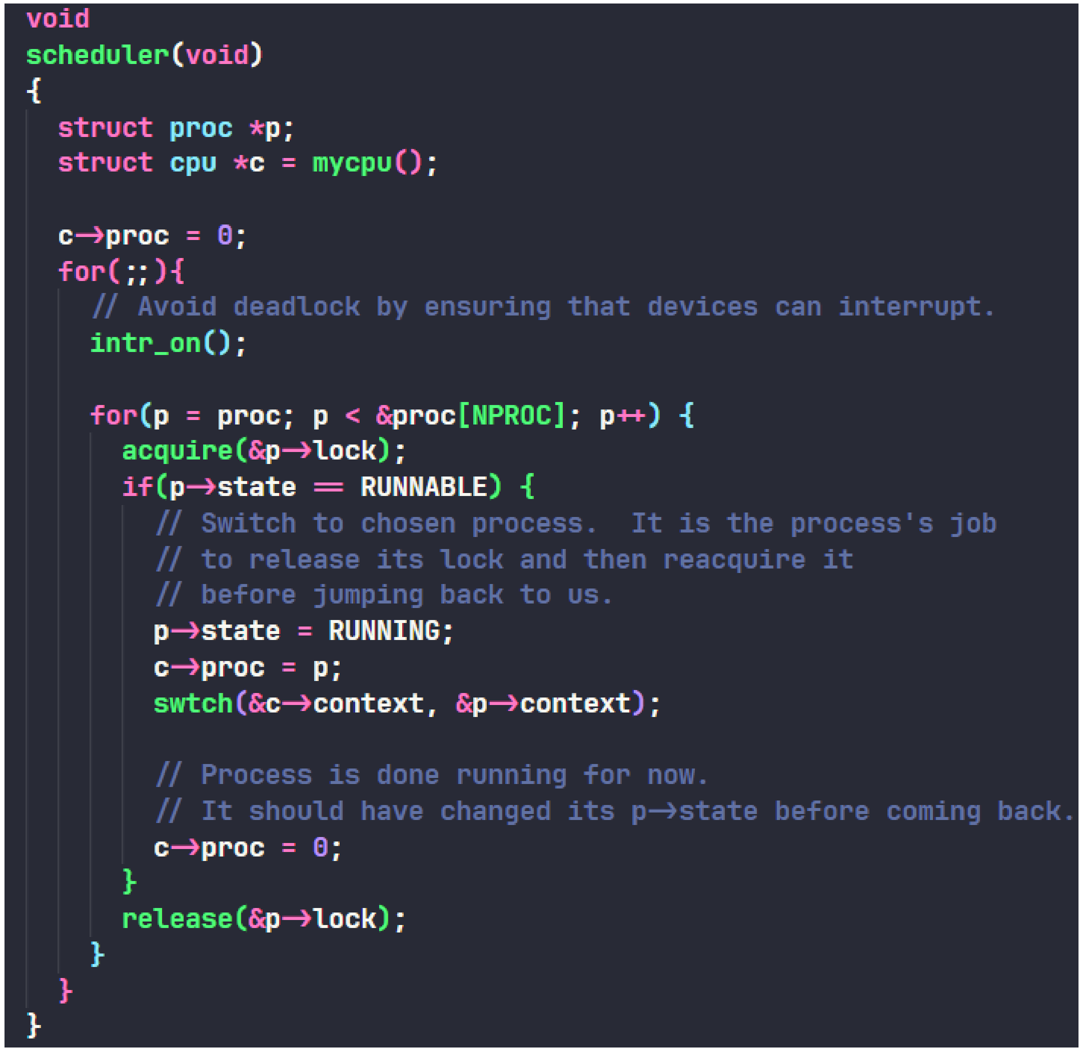
```
void
forkret(void)
{
  static int first = 1;

  // Still holding p->lock from scheduler.
  release(&myproc()->lock);

  if (first) {
    // File system initialization must be run in the context of a
    // regular process (e.g., because it calls sleep), and thus cannot
    // be run from main().
    first = 0;
    fsinit(ROOTDEV);
  }

  usertrapret();
}
```

- 如果是史上第一个进程
  将会初始化文件系统
- 此后调用usertrapret()
  从内核态前往用户态
  开始运行用户进程

竞争与锁

spin lock

竞争

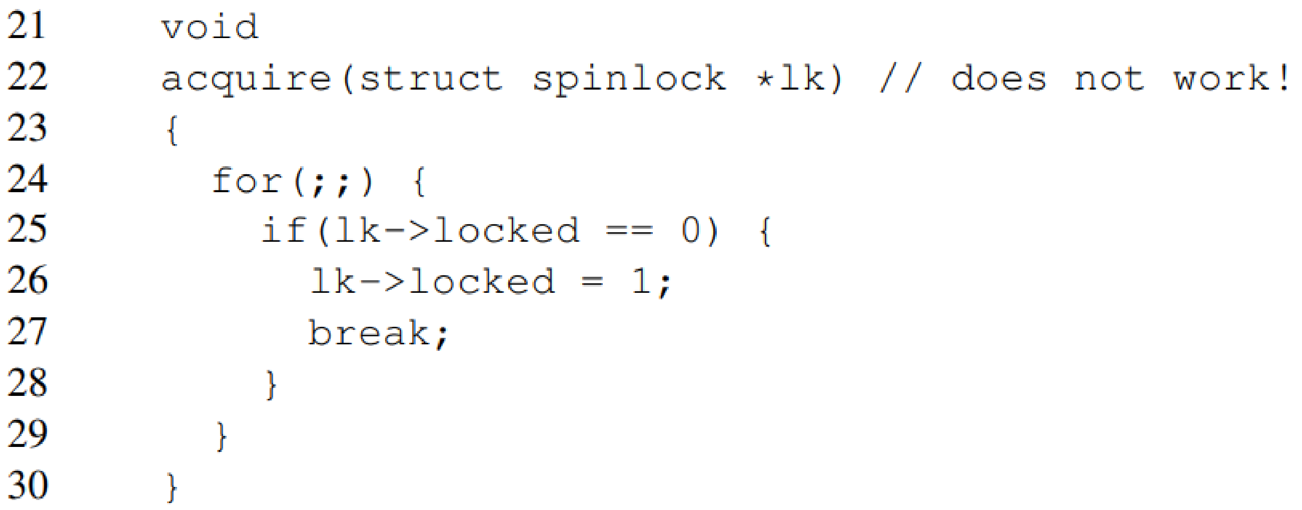


```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);

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

- xv6支持多核。
- 假设两个cpu同时访问p的状态，发现p是RUNNABLE的，然后两个cpu会同时切换到p并运行，这显然出错了。
- 所以需要锁来避免竞争。

申请锁

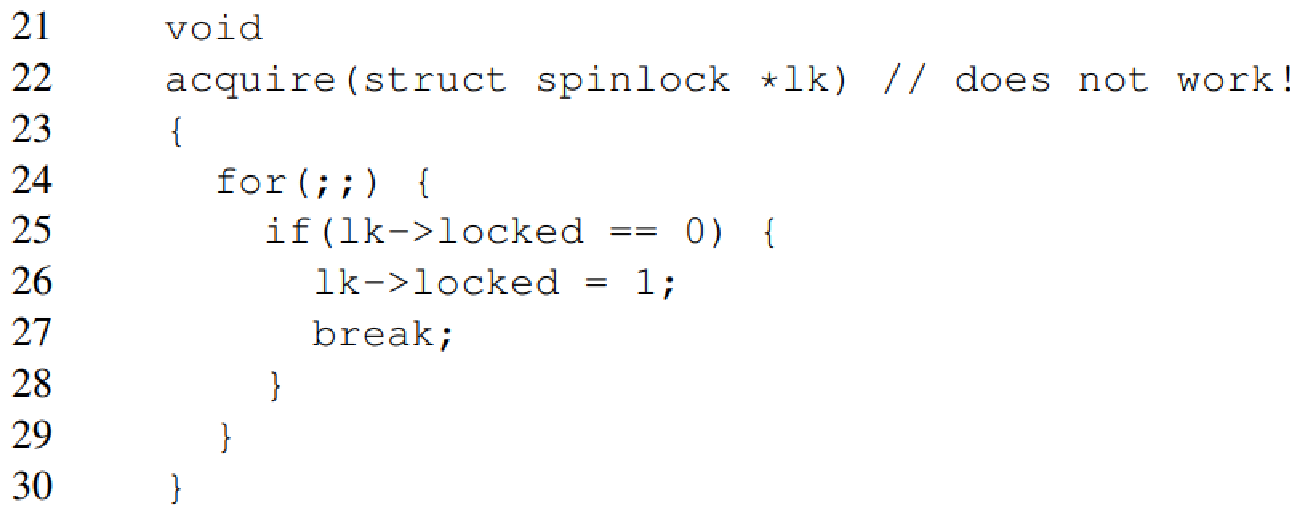
- 锁有两种状态，锁上或解开。
- xv6使用的是spin lock，申请锁的逻辑是如果锁已经锁上就一直申请直到锁解开就获取该锁并把锁锁上。
- 每次只有一个cpu可以申请到锁。
- **逻辑上**如下图程序所示。



```
void
acquire(struct spinlock *lk) // does not work!
{
  for(;;) {
    if(lk->locked == 0) {
      lk->locked = 1;
      break;
    }
  }
}
```

申请锁

- 下面的代码**并不起作用**，原因是如果两个cpu同时申请同一个打开的锁并且同时运行到25行，那么他们会同时申请到该锁。
- 要想真正实现锁的功能需要使用原子操作。



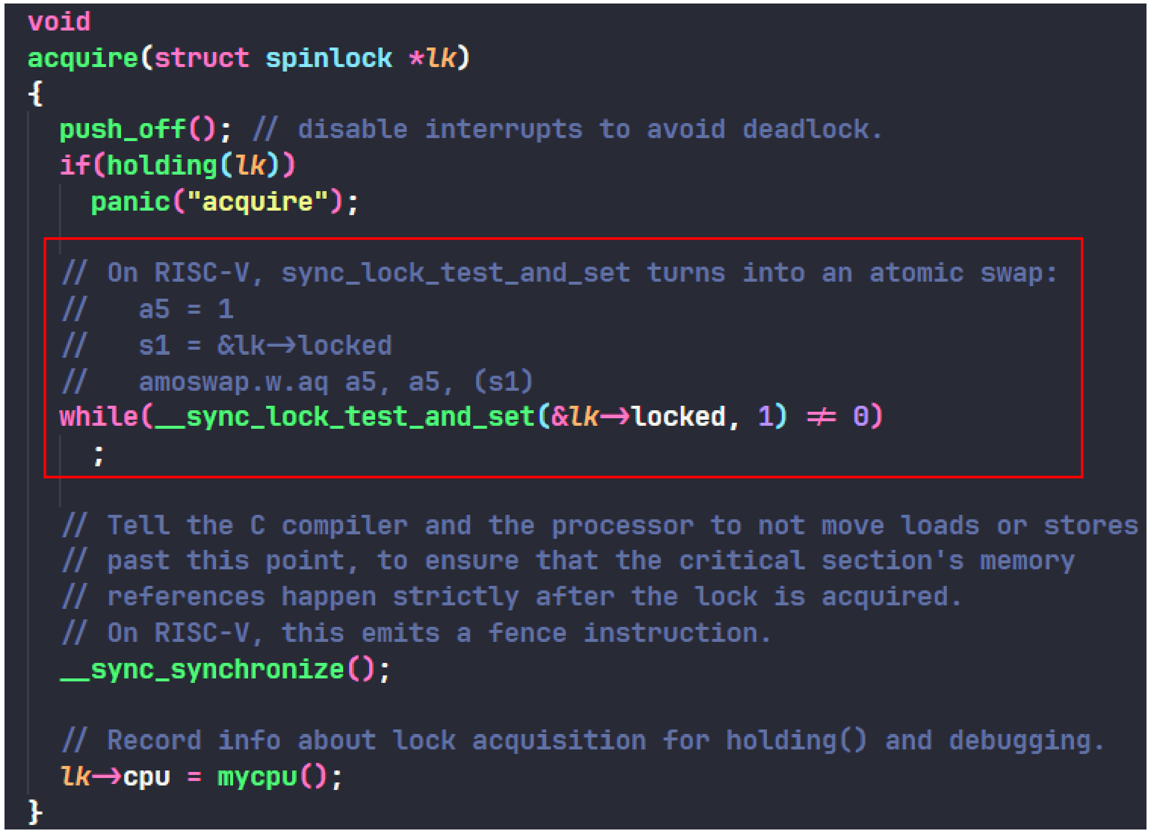
```
21      void
22      acquire(struct spinlock *lk) // does not work!
23      {
24        for(;;) {
25          if(lk->locked == 0) {
26            lk->locked = 1;
27            break;
28          }
29        }
30      }
```

申请锁

- **risc-v**中**amoswap r, a**可以实现25、26行功能。
- **amoswap**读取内存地址**a**处的值，将寄存器**r**中的值写到内存地址**a**处，并且将读取的值写入寄存器**r**中。

```
21    void
22    acquire(struct spinlock *lk) // does not work!
23    {
24      for(;;) {
25        if(lk->locked == 0) {
26          lk->locked = 1;
27          break;
28        }
29      }
30    }
```

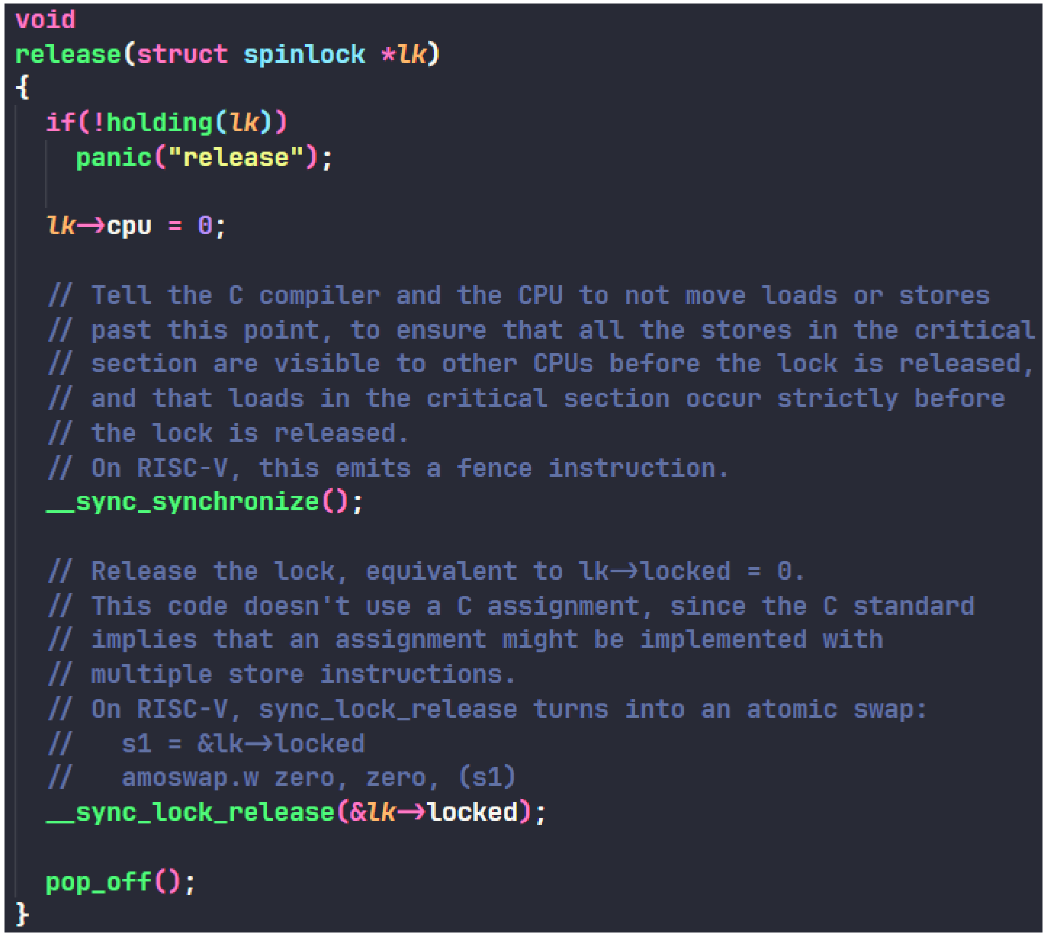
申请锁



```c
void
acquire(struct spinlock *lk)
{
  push_off(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // On RISC-V, sync_lock_test_and_set turns into an atomic swap:
  //   a5 = 1
  //   s1 = &lk->locked
  //   amoswap.w.aq a5, a5, (s1)
  while(__sync_lock_test_and_set(&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 strictly after the lock is acquired.
  // On RISC-V, this emits a fence instruction.
  __sync_synchronize();

  // Record info about lock acquisition for holding() and debugging.
  lk->cpu = mycpu();
}
```

- __sync_lock_test_and_set是c库中的原子操作函数，其内部使用了amoswap指令。其将&lk->locked置为1并返回&lk->locked被修改之前的值。
- __sync_synchronize()告诉编译器和处理器不允许移动load和store指令越过该指令。
- 更新所对应的cpu信息。

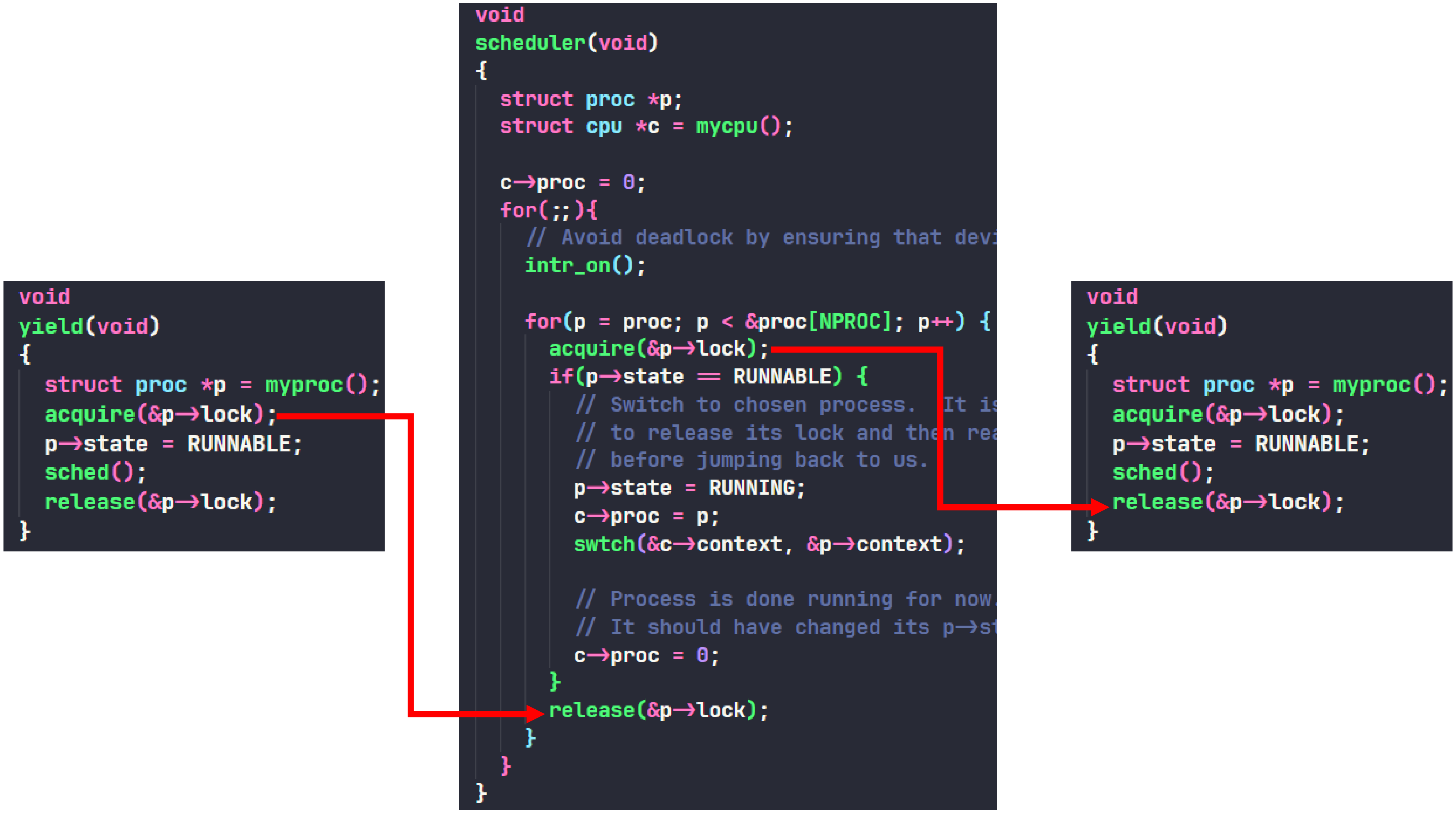
释放锁



```
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->cpu = 0;

  // Tell the C compiler and the CPU to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other CPUs before the lock is released,
  // and that loads in the critical section occur strictly before
  // the lock is released.
  // On RISC-V, this emits a fence instruction.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code doesn't use a C assignment, since the C standard
  // implies that an assignment might be implemented with
  // multiple store instructions.
  // On RISC-V, sync_lock_release turns into an atomic swap:
  //   s1 = &lk->locked
  //   amoswap.w zero, zero, (s1)
  __sync_lock_release(&lk->locked);

  pop_off();
}
```

- 清除锁对于的cpu信息。
- __sync_synchronize()告诉编译器和处理器不允许移动load和store指令越过该指令。
- __sync_lock_release()使用amoswap将锁状态修改为打开。

锁的配套



```
void
yield(void)
{
  struct proc *p = myproc();
  acquire(&p->lock);
  p->state = RUNNABLE;
  sched();
  release(&p->lock);
}
```

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    // Avoid deadlock by ensuring that devi
    intr_on();

    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is
        // to release its lock and then rea
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);

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

```
void
yield(void)
{
  struct proc *p = myproc();
  acquire(&p->lock);
  p->state = RUNNABLE;
  sched();
  release(&p->lock);
}
```

xv6 vs. Linux

xv6

- xv6调度算法就是简单的round robin。

Linux

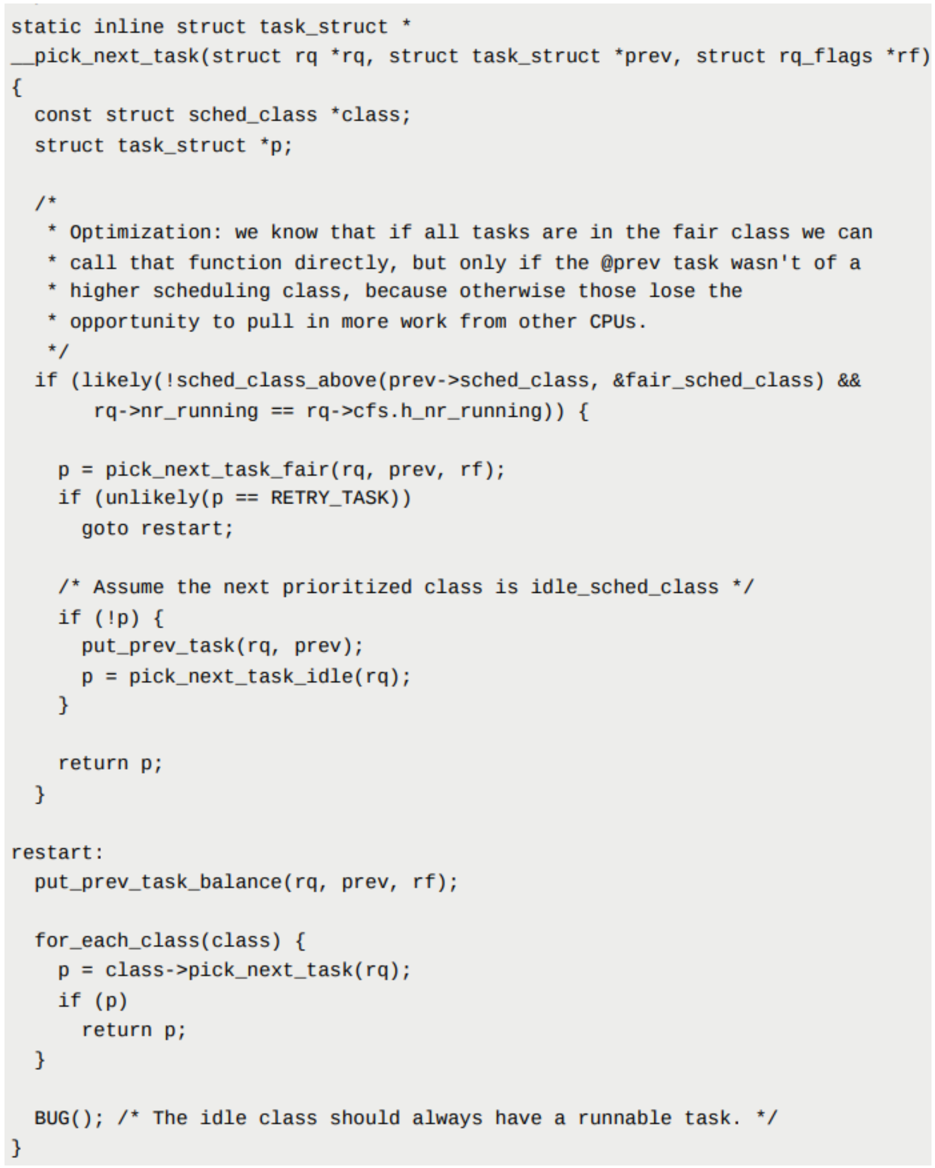
- 分时共享调度（非实时任务）
  - Complete Fairness Schedule (CFS)
  - 根据nice值(-20到19)，设定每个进程的优先级
  - 内核根据进程的nice值计算virtual runtime，nice值为0时，virtual runtime和物理运行时间一致，nice值越大，virtual runtime越大，但实际运行物理时间小，反之，实际运行物理时间大。
  - 通过每个进程的虚拟时间（virtual runtime）来选择被调度的进程，virtual runtime越小的进程越先调度。

CFS

```
static void __sched notrace __schedule(unsigned int sched_mode)
{
        struct task_struct *prev, *next;
        struct rq_flags rf;
        struct rq *rq;
        int cpu;

        cpu = smp_processor_id();
        rq = cpu_rq(cpu);
        prev = rq->curr;

        next = pick_next_task(rq, prev, &rf);

        if (likely(prev != next)) {
                rq = context_switch(rq, prev, next, &rf);
        } else {
                __balance_callbacks(rq);
        }
}
```

- 代码经过大量删减（不知道删的有没有问题>_<）
- 调度入口
- pick_next_task选择要执行的进程。

CFS



```
static inline struct task_struct *
__pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
  const struct sched_class *class;
  struct task_struct *p;

  /*
   * Optimization: we know that if all tasks are in the fair class we can
   * call that function directly, but only if the @prev task wasn't of a
   * higher scheduling class, because otherwise those lose the
   * opportunity to pull in more work from other CPUs.
   */
  if (likely(!sched_class_above(prev->sched_class, &fair_sched_class) &&
       rq->nr_running == rq->cfs.h_nr_running)) {

    p = pick_next_task_fair(rq, prev, rf);
    if (unlikely(p == RETRY_TASK))
      goto restart;

    /* Assume the next prioritized class is idle_sched_class */
    if (!p) {
      put_prev_task(rq, prev);
      p = pick_next_task_idle(rq);
    }

    return p;
  }

restart:
  put_prev_task_balance(rq, prev, rf);

  for_each_class(class) {
    p = class->pick_next_task(rq);
    if (p)
      return p;
  }

  BUG(); /* The idle class should always have a runnable task. */
}
```

- pick_next_task()内调用了__pick_next_task()。
- __pick_next_task进行了一个优化，因为大部分时间系统中主要存在的都是普通进程，所以先检测运行队列的运行数量和公平运行列队中的运行数量，如果相等的话就说明系统中目前只有普通进程，那么就可以直接调用pick_next_task_fair。接着就是主逻辑了，先从高调度类进行选择，如果有可运行的进程就直接返回，如果没有就去查询下一个调度类。最后一定能返回一个进程的，因为idle进程总是可运行的。

CFS

- CFS使用一棵红黑树——平衡树（你可以把其当成一个支持O(logn)插入，删除的优先队列），树上维护的键值是virual runtime。对于每一个进程，系统将nice值换算成weight,每运行完一个时间片，当前进程的virtual runtime += real runtime / weight。每次选择virtual runtime最小的进程运行。
- 对于之前阻塞现在要重新进入ready_queue的进程，如果其virtual runtime小于当前队列最小的virtual runtime，就将其virtual runtime改成当前队列最小的virtual runtime。

欢迎批评指正！