线程调度

概述

目标

1. 理解操作系统的调度管理机制
2. 熟悉 XV6 系统的线程调度方法
3. 熟悉 Round-Robin 调度算法
4. 实现多级反馈队列调度算法

简述

任何操作系统都可能运行比 CPU 数量更多的进程，所以需要一个进程间分时共享 CPU 的方案。这种共享最好对用户进程透明。一种常见的方法是，通过将进程多路复用到硬件 CPU 上，使每个进程产生一种错觉，即它有自己的虚拟 CPU。

进程调度是指操作系统在多任务环境下，决定哪个进程能够占用 CPU 的过程。在计算机中，进程是指正在运行的程序，一个进程通常由多个线程组成。

操作系统采用进程调度算法来实现进程的调度。常见的调度算法包括先来先服务（FCFS）、最短作业优先（SJF）、优先级调度、时间片轮转等。

进程调度的目标是使得 CPU 利用率最高，同时保证系统资源的公平分配和响应时间的合理。对于不同的应用场景，需要选择不同的调度算法以达到最优的性能。

在本部分中，我么将会介绍 XV6 系统是如何实现进程调度的。在实验部分，会由浅入深的安排若干实验，观察进程调度过程，在 XV6 上实现更“高级”的调度算法等。

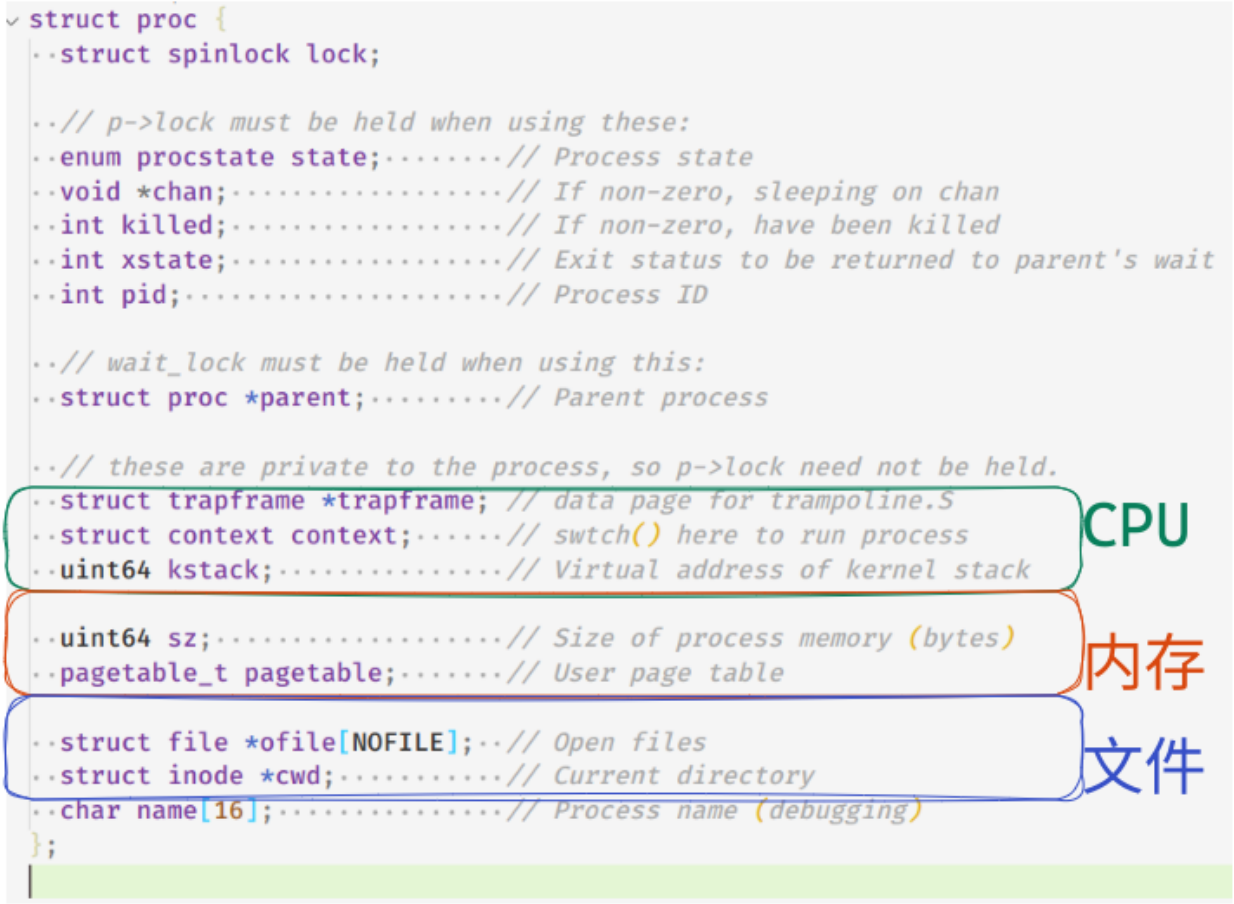
讲解

进程与线程

这个问题，是操作系统里问的最多的问题之一，也是被误解最深的概念之一。

线程是进程的一部分，进程是资源分配的最小单位，线程是 CPU 调度的最小单位。 这句话概括的比较准确，但多少有点难以理解，接下来将会详细的叙述。

先说资源，我们都有什么资源？下图是 XV6 的 PCB 结构体




```
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.
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  uint64 kstack;               // Virtual address of kernel stack

  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table

  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

可以看到在 XV6 上，我们就至少拥有 CPU，内存，打开的文件等资源。

进程，在一定的环境下，把静态的程序代码运行起来，通过使用不同的资源，来完成一定的任务。比如说，进程的环境包括环境变量，进程所掌控的资源，有中央处理器，有内存，打开的文件，映射的网络端口等等。

其中，线程作为进程的一部分，扮演的角色就是怎么利用中央处理器去运行代码。这其中牵扯到的最重要资源的是 CPU, 寄存器和栈（stack）。这里想强调的是，线程关注的是中央处理器的运行，而不是内存等资源的管理。线程间切换简单的说只需要保存当前的寄存器，然后再对这些寄存器赋新值就行了。

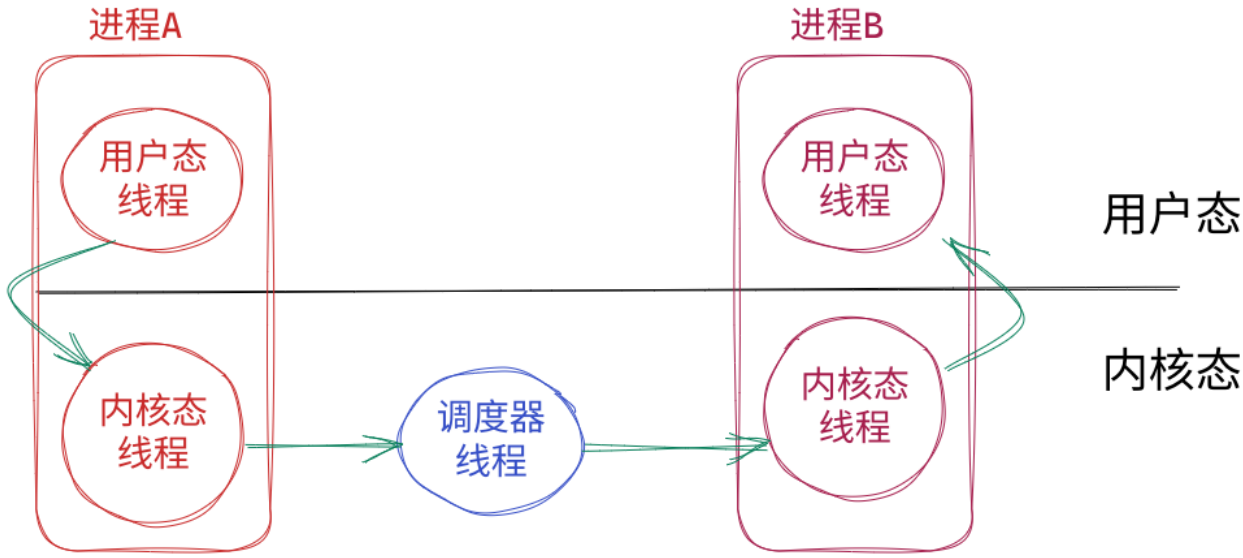
可以说, 进程包括：

- 线程 0，1，2...
- 线程： 寄存器，stack
- 内存
- 文件
- 网络端口等

接下来再说两点：

- 一个 CPU 在某个时刻只会在一个上下文环境里做一件事
- CPU 通电之后就在不停的运行，没法暂停下来，在一段时间什么事都不做，它一直在运行
  - 但因为有多个 CPU，所以可以并行
  - 但因为时间片轮转，所以可以并发
- 在 XV6中，一个进程要么在用户空间执行指令，要么是在内核空间执行指令，要么它的状态被保存在 context 和 trapframe 中，并且没有执行任何指令。
- 可以说，每个进程有两个“线程”，一个用户空间线程，一个内核空间线程，并且存在限制使得一个进程要么运行在用户空间线程，要么为了执行系统调用或者响应中断而运行在内核空间线程，但是永远也不会两者同时运行。

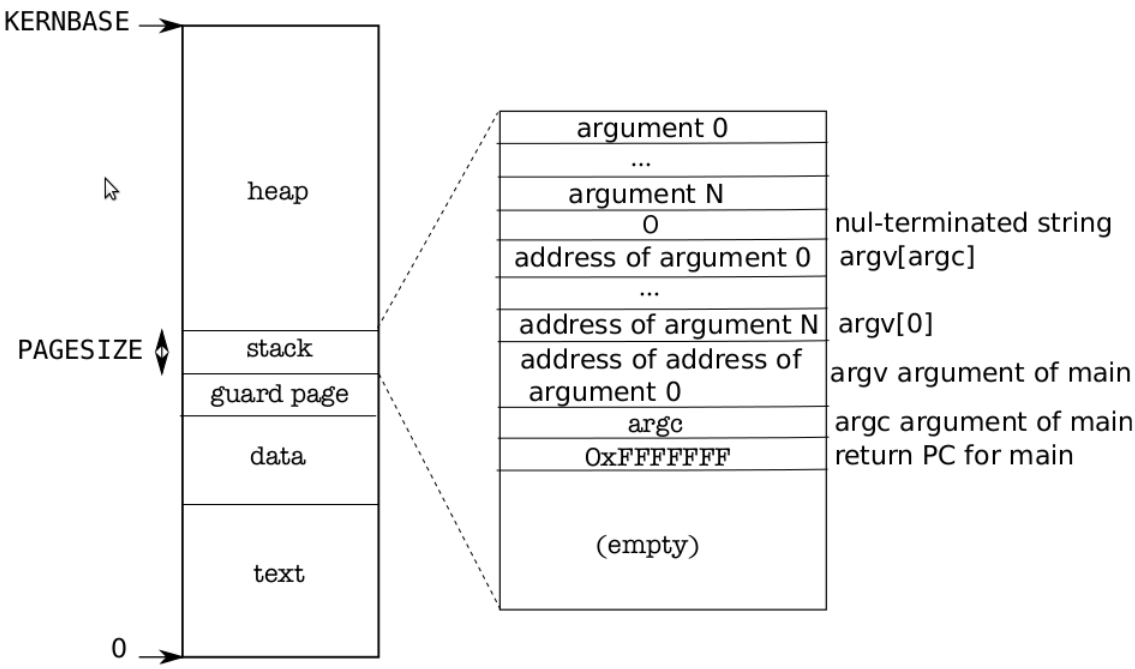
下图是 XV6 中从进程 A 用完一次时间片调度到进程 B （曾经运行过）的过程：



- 进程 A 用户态线程 --> 进程 A 内核态线程
- 进入内核态（定时器中断）
  - 保存所有寄存器，切换到进程 A 对应的内核栈（kstack）
- 进程 A 内核态线程 --> CPU 调度器线程
- 保存一些寄存器，切换到调度器线程的栈
- CPU 调度器线程 --> 进程 B 内核态线程
- 保存一些寄存器，切换到进程 B 对应的内核栈
- 进程 B 内核态线程 --> 进程 B 用户态线程
- 恢复所有寄存器，切换到进程 B 对应的用户栈
  - 重回用户态

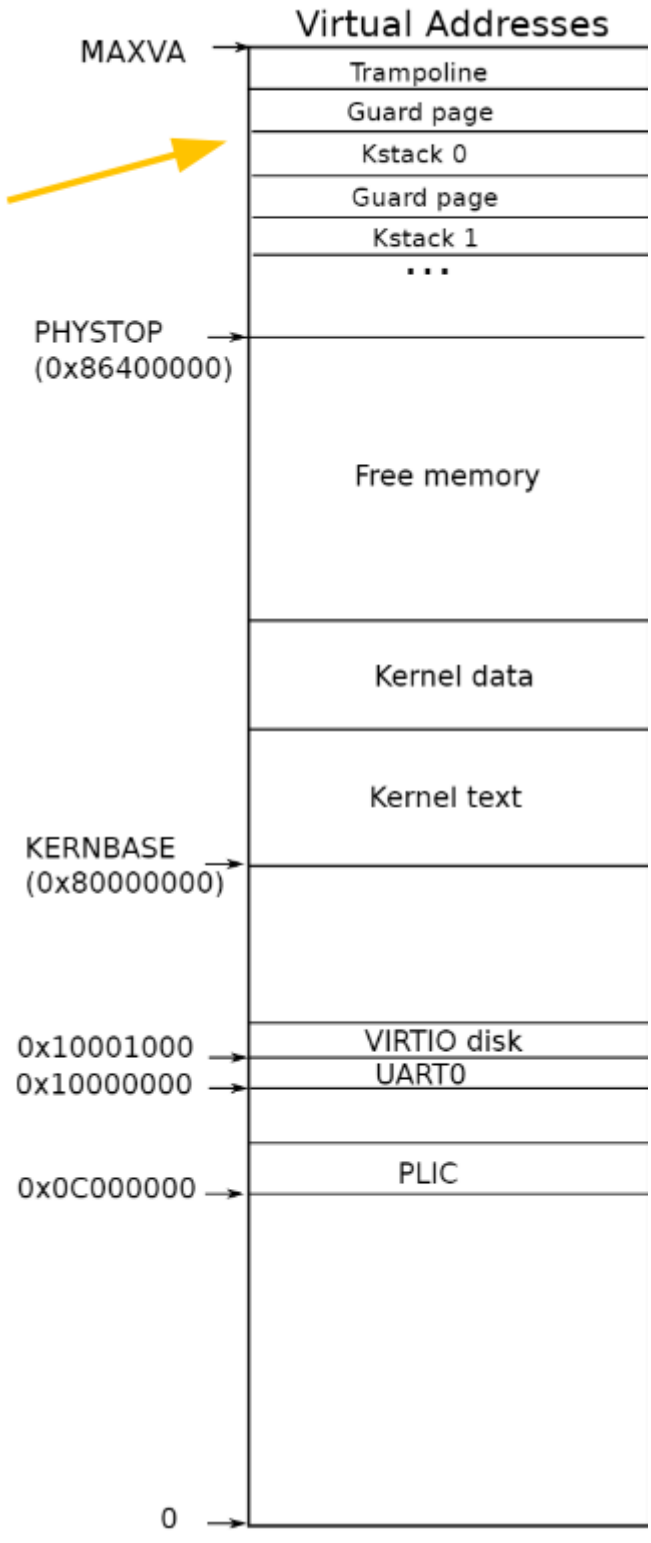
每个进程都有两个栈：

- 用户栈，给用户态线程使用
-



  - 上图中（用户地址空间）的 stack ，在进程创建时开辟

- 内核栈 kstack ，给内核态线程使用
- 每个进程都有其对应内核栈



  - XV6 中位于内核地址空间的顶部

同样的，调度器线程也有其对应栈：

多线程

为什么计算机需要运行多线程？可以归结为以下原因：

- 首先，人们希望他们的计算机在同一时间不是只执行一个任务。在一个单用户的计算机或者在你的iphone上，你会运行多个进程，并期望计算机完成所有的任务而不仅仅只是一个任务。
- 其次，多线程可以让程序的结构变得简单。线程在有些场合可以帮助程序员将代码以简单优雅的方式进行组织，并减少复杂度。
- 最后，使用多线程可以通过并行运算，在拥有多核 CPU 的计算机上获得更快的处理速度。常见的方式是将程序进行拆分，并通过线程在不同的 CPU 核上运行程序的不同部分。如果你足够幸运的话，你可以将你的程序拆分并在4个 CPU 核上通过4个线程运行你的程序，同时你也可以获取4倍的程序运行速度。
- 你可以认为 XV6就是一个多 CPU 并行运算的程序。

所以，线程可以认为是一种在有多个任务时简化编程的抽象。一个线程可以认为是串行执行代码的单元。如果你写了一个程序只是按顺序执行代码，那么你可以认为这个程序就是个单线程程序，这是对于线程的一种宽松的定义。虽然人们对于线程有很多不同的定义，在这里，我们认为线程就是单个串行执行代码的单元，它只占用一个 CPU 并且以普通的方式一个接一个的执行指令。

除此之外，线程还具有状态，我们可以随时保存线程的状态并暂停线程的运行，并在之后通过恢复状态来恢复线程的运行。线程的状态包含了三个部分：

- 程序计数器（Program Counter），它表示当前线程执行指令的位置。
- 保存变量的寄存器。
- 程序的 Stack。通常来说每个线程都有属于自己的 Stack，Stack 记录了函数调用的记录，并反映了当前线程的执行点。

操作系统中线程系统的工作就是管理多个线程的运行。我们可能会启动成百上千个线程，而线程系统的工作就是弄清楚如何管理这些线程并让它们都能运行。

多线程的并行运行主要有两个策略：

- 第一个策略是在多核处理器上使用多个 CPU，每个 CPU 都可以运行一个线程，如果你有4个 CPU，那么每个 CPU 可以运行一个线程。
    - 每个线程自动的根据所在 CPU 就有了程序计数器和寄存器。但是如果你只有4个 CPU，却有上千个线程，每个 CPU 只运行一个线程就不能解决这里的问题了。
- 第二个策略，也就是一个 CPU 在多个线程之间来回切换。

    - 假设我只有一个 CPU，但是有1000个线程，我们接下来将会看到 XV6是如何实现线程切换使得 XV6能够先运行一个线程，之后将线程的状态保存，再切换至运行第二个线程，然后再是第三个线程，依次类推直到每个线程都运行了一会，再回来重新执行第一个线程。

实际上，与大多数其他操作系统一样，XV6结合了这两种策略，

- 首先线程会运行在所有可用的 CPU 核上
- 其次每个 CPU 核会在多个线程之间切换，因为通常来说，线程数会远远多于 CPU 的核数。

不同线程系统之间的一个主要的区别就是，线程之间是否会共享内存。一种可能是你有一个地址空间，多个线程都在这一个地址空间内运行，并且它们可以看到彼此的更新。比如说共享一个地址空间的线程修改了一个变量，共享地址空间的另一个线程可以看到变量的修改。所以当多个线程运行在一个共享地址空间时，我们需要用到上一部分讲到的锁。

XV6内核共享了内存，并且 XV6支持内核线程的概念，对于每个用户进程都有一个内核线程来执行来自用户进程的系统调用。所有的内核线程都共享了内核内存，所以 XV6 的内核线程的确会共享内存。

另一方面，XV6还有另外一种线程。每一个用户进程都有独立的内存地址空间，并且包含了一个线程，这个线程控制了用户进程代码指令的执行。所以 XV6 中的用户线程之间没有共享内存，你可以有多个用户进程，但是每个用户进程都是拥有一个线程的独立地址空间。XV6 中的进程不会共享内存。

在一些其他更加复杂的系统中，例如 Linux，允许在一个用户进程中包含多个线程，进程中的多个线程共享进程的地址空间。当你想要实现一个运行在多个 CPU 核上的用户进程时，你就可以在用户进程中创建多个线程。Linux 中也用到了很多我们今天会介绍的技术，但是在 Linux 中跟踪每个进程的多个线程比 XV6中每个进程只有一个线程要复杂的多。

还有一些其他的方式可以支持在一台计算机上交织的运行多个任务，我们不会讨论它们，但是如果你感兴趣的话，你可以去搜索 event-driven programming 或者 state machine，这些是在一台计算机上不使用线程但又能运行多个任务的技术。在所有的支持多任务的方法中，线程技术并不是非常有效的方法，但是线程通常是最方便，对程序员最友好的，并且可以用来支持大量不同任务的方法。

线程切换

实现内核中的线程系统存在以下挑战：

- 第一个是如何实现线程间的切换。这里停止一个线程的运行并启动另一个线程的过程通常被称为线程调度（Scheduling）。我们将会看到XV6为每个CPU核都创建了一个线程调度器（Scheduler）。
- 第二个挑战是，当你想要实际实现从一个线程切换到另一个线程时，你需要保存并恢复线程的状态，所以需要决定线程的哪些信息是必须保存的，并且在哪保存它们。
- 最后一个挑战是如何处理运算密集型线程（compute bound thread）。对于线程切换，很多直观的实现是由线程自己自愿的保存自己的状态，再让其他的线程运行。但是如果我们有一些程序正在执行一些可能要花费数小时的长时间计算任务，这样的线程并不能自愿的出让 CPU 给其他的线程运行。所以这里需要能从长时间运行的运算密集型线程撤回对于 CPU 的控制，将其放置于一边，稍后再运行它。

接下来，我将首先介绍如何处理运算密集型线程。这里的具体实现你们之前或许已经知道了，就是利用定时器中断。在每个 CPU 核上，都存在一个硬件设备，它会定时产生中断。XV6与其他所有的操作系统一样，将这个中断传输到了内核中。所以即使我们正在用户空间计算π的前100万位，定时器中断仍然能在例如每隔10ms 的某个时间触发，并将程序运行的控制权从用户空间代码切换到内核中的中断处理程序（注，因为中断处理程序优先级更高）。哪怕这些用户空间进程并不配合工作（注，也就是用户空间进程一直占用 CPU），内核也可以从用户空间进程获取 CPU 控制权。

位于内核的定时器中断处理程序，会自愿的将 CPU 出让（yield）给线程调度器，并告诉线程调度器说，你可以让一些其他的线程运行了。这里的出让其实也是一种线程切换，它会保存当前线程的状态，并在稍后恢复。

这里的基本流程是，定时器中断将 CPU 控制权给到内核，内核再自愿的出让 CPU。

这样的处理流程被称为抢占式调度（pre-emptive scheduling）。pre-emptive 的意思是，即使用户代码本身没有出让 CPU，定时器中断仍然会将 CPU 的控制权拿走，并出让给线程调度器。与之相反的是协作式调度（voluntary scheduling）。

有趣的是，在 XV6和其他的操作系统中，线程调度是这么实现的：

- 定时器中断会强制的将 CPU 控制权从用户进程给到内核，这里是抢占式调度
- 之后内核中，用户进程对应的内核线程会代表用户进程出让 CPU，使用协作式调度

在执行线程调度的时候，操作系统需要能区分几类线程：

- 当前在 CPU 上运行的线程
  - RUNNING
- 一旦 CPU 有空闲时间就想要运行在 CPU 上的线程
  - RUNABLE
- 以及不想运行在 CPU 上的线程，因为这些线程可能在等待 I/O 或者其他事件
  - SLEEPING

今天这节课，我们主要关注 RUNNING 和 RUNABLE 这两类线程。前面介绍的定时器中断或者说 pre-emptive scheduling，实际上就是将一个 RUNNING 线程转换成一个 RUNABLE 线程。通过出让 CPU，pre-emptive scheduling 将一个正在运行的线程转换成了一个当前不在运行但随时可以再运行的线程。因为当定时器中断触发时，这个线程还在好好的运行着。

以 `cc` 切换到 `ls` 为例，且 `ls` 此前运行过

1. XV6将 `cc` 程序的内核线程的内核寄存器保存在一个 `context` 对象中
2. 因为要切换到 `ls` 程序的内核线程，那么 `ls` 程序现在的状态必然是 `RUNABLE` ，表明 `ls` 程序之前运行了一半。这同时也意味着：
   a. `ls` 程序的用户空间状态已经保存在了对应的trapframe中
   b. `ls` 程序的内核线程对应的内核寄存器已经保存在对应的 `context` 对象中
   所以接下来，XV6会恢复 `ls` 程序的内核线程的 `context` 对象，也就是恢复内核线程的寄存器。
3. 之后 `ls` 会继续在它的内核线程栈上，完成它的中断处理程序
4. 恢复 `ls` 程序的 trapframe 中的用户进程状态，返回到用户空间的 `ls` 程序中
5. 最后恢复执行 `ls`

这里核心点在于，在 XV6中，任何时候都需要经历：

1. 从一个用户进程切换到另一个用户进程，都需要从第一个用户进程接入到内核中，保存用户进程的状态并运行第一个用户进程的内核线程。
2. 再从第一个用户进程的内核线程切换到第二个用户进程的内核线程。
3. 之后，第二个用户进程的内核线程暂停自己，并恢复第二个用户进程的用户寄存器。
4. 最后返回到第二个用户进程继续执行。

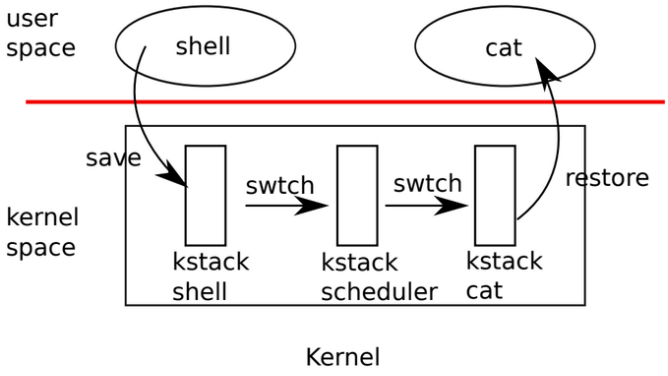


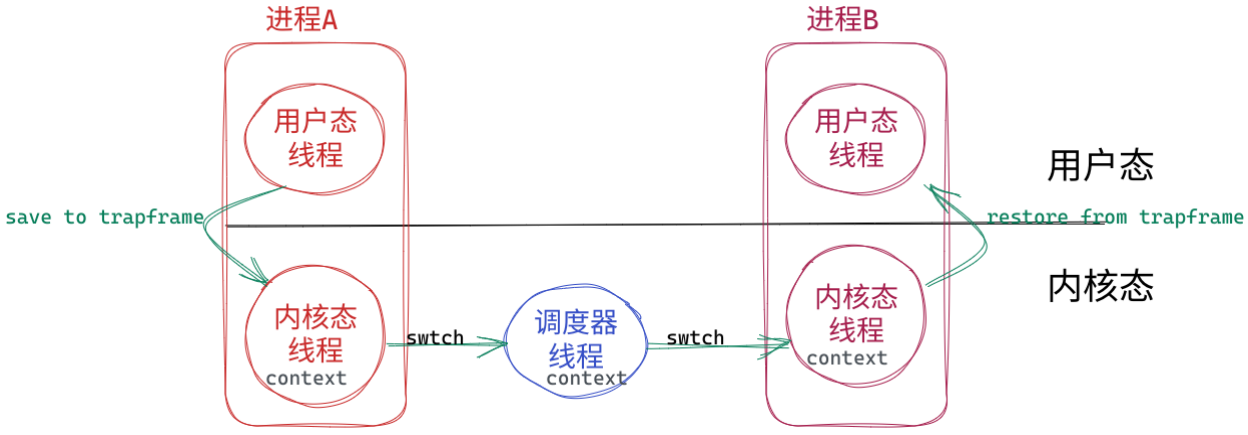
Figure 7.1: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

这里有一个术语需要解释一下。当人们在说 context switching，他们通常说的是从一个线程切换到另一个线程，因为在切换的过程中需要先保存前一个线程的寄存器，然后再恢复之前保存的后一个线程的寄存器，这些寄存器都是保存在 context 对象中。在有些时候，context switching 也指从一个用户进程切换到另一个用户进程的完整过程。偶尔你也会看到 context switching 是指从用户空间和内核空间之间的切换。对于我们这节课来说，context switching 主要是指一个内核线程和调度器线程之间的切换。

这里有一些有用的信息可以记住。每一个 CPU 核在一个时间只会做一件事情，每个 CPU 核在一个时间只会运行一个线程，

- 它要么是运行用户进程的线程，
- 要么是运行内核线程，
- 要么是运行这个 CPU 核对应的调度器线程。

所以在任何一个时间点，CPU 核并没有做多件事情，而是只做一件事情。线程的切换创造了多个线程同时运行在一个 CPU 上的假象。类似的每一个线程要么是只运行在一个 CPU 核上，要么它的状态被保存在 context 中。线程永远不会运行在多个 CPU 核上，线程要么运行在一个 CPU 核上，要么就没有运行。



进程从用户态陷入到内核态时，需要保存寄存器到 trapframe (kernel/trampoline. S)。
在内核态里，从内核态线程与调度器线程切换时，需要保存到 context 结构体。
下面来看这俩的定义。

```
struct trapframe {
  /*   0 */ uint64 kernel_satp;   // kernel page table
  /*   8 */ uint64 kernel_sp;     // top of process's kernel stack
  /*  16 */ uint64 kernel_trap;   // usertrap()
  /*  24 */ uint64 epc;           // saved user program counter
  /*  32 */ uint64 kernel_hartid; // saved kernel tp
  /*  40 */ uint64 ra;
  /*  48 */ uint64 sp;
  /*  56 */ uint64 gp;
  /*  64 */ uint64 tp;
  /*  72 */ uint64 t0;
  /*  80 */ uint64 t1;
  /*  88 */ uint64 t2;
  /*  96 */ uint64 s0;
  /* 104 */ uint64 s1;
  /* 112 */ uint64 a0;
  /* 120 */ uint64 a1;
  /* 128 */ uint64 a2;
  /* 136 */ uint64 a3;
  /* 144 */ uint64 a4;
  /* 152 */ uint64 a5;
  /* 160 */ uint64 a6;
  /* 168 */ uint64 a7;
  /* 176 */ uint64 s2;
  /* 184 */ uint64 s3;
  /* 192 */ uint64 s4;
  /* 200 */ uint64 s5;
  /* 208 */ uint64 s6;
  /* 216 */ uint64 s7;
  /* 224 */ uint64 s8;
  /* 232 */ uint64 s9;
  /* 240 */ uint64 s10;
  /* 248 */ uint64 s11;
  /* 256 */ uint64 t3;
  /* 264 */ uint64 t4;
  /* 272 */ uint64 t5;
  /* 280 */ uint64 t6;
};
```

```
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```

可以看到，trapframe 几乎包含了所有寄存器。
而 context 只包含一些寄存器（callee saved register 与栈指针寄存器 sp ,函数返回地址寄存器 ra）。

感觉上都是进行了线程切换，为什么需要保存的寄存器不同呢?

上文说道：
在 XV6和其他的操作系统中，线程调度是这么实现的：

- 定时器中断会强制的将 CPU 控制权从用户进程给到内核，这里是抢占式调度
- 之后内核中，用户进程对应的内核线程会代表用户进程出让 CPU，使用协作式调度

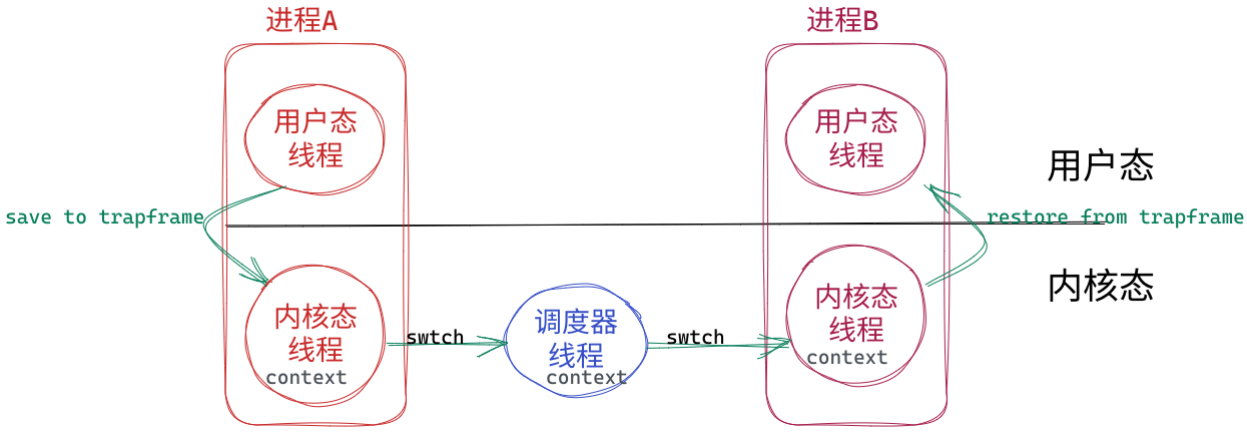
保存的寄存器不同，其实是抢占式调度和协作式调度所带来的。
回到上面的例子，进程 A 正在好好的运行着，定时器中断发生了，所以操作系统要让进程 A 暂停运行，让进程 B 继续运行（时间片轮转），以提高系统的并发。
对于进程 A 而言，它并不知道自己要被剥夺运行，它是被“无知觉地强制”剥夺了运行，为了之后能回到进程 A 现在的运行状态，因为操作系统也不知道进程 A 使用了哪些寄存器，所以需要保存所有的寄存器。

而内存态线程与调度器线程之间的切换是协作式调度。进程 A 的内核态线程切换到调度器线程是“主动进行”的，其调用了 `yield()` 函数切换到调度器线程。对于进程 A 的内核态线程而言，其就像调用了一个普通的 `yield()` 函数，然后函数返回了，继续运行。但实际上，进程 A 的内核态线程调用了 `yield()` 函数，其实际就不在 CPU 上运行了。

协作式调度的上下文切换（内核态的线程切换）就像调用了一个普通的函数一样，需要遵循调用约定（**calling convention**），即保存所有 callee saved register 。此外，需要切换栈，也要保存 sp 寄存器，函数返回地址寄存器 ra。

> 上下文切换不是只能发生在内核态，在用户态也可以发生。所谓的“用户线程”切换就是如此，libc 里有两个函数 `setjmp` 和 `longjmp` ，就是对此的包装。借助这两个函数，你也可以实现“协程”（类似 Python 里的 Generator， `yield` 关键字），其原理是相同的。

接下来看代码。
依旧是进程 A 因定时器中断，切换到进程 B 。



定时器中断 `usertrap`

因发生计时器中断，系统会 trap 陷入到内核，在 trampoline. S 里保存所有寄存器，然后跳到 `usertrap` 函数。

在 `usertrap` 里，会因为本次 trap 是定时器中断而调用 `yield()`

```
75
76  ··if(killed(p))
77  ····exit(-1);
78
79  ··// give up the CPU if this is a timer interrupt.
80  ··if(which_dev == 2)      Robert Morris, 4年前 • yield
81  ····yield();
82
83  ··usertrapret();
84  }
```

> 执行本函数的时候，位于进程 A 的内核态线程的上下文里面。

`yield`

```
// Give up the CPU for one scheduling round.
void
yield(void)      rsc, 17年前 • …
{
  struct proc *p = myproc();
  acquire(&p->lock);
  p->state = RUNNABLE;
  sched();
  release(&p->lock);
}
```

这里，我们将进程 A 的状态改为 RUNNABLE.

XV6 里进程的状态有：

- RUNNING
- 正在运行中
- RUNNABLE
- 不在运行中，可以被调度运行
- SLEEPING
- 等待 IO 事件等
- ZOMBIE
- 进程本身已经终止，但是父进程尚未调用 wait() 或 waitpid() 对它进行清理

`yield` 函数只做了几件事情，它首先获取了进程的锁。实际上，在锁释放之前，进程的状态会变得不一致，例如，yield 将要将进程的状态改为 RUNABLE，表明进程并没有在运行，但是实际上这个进程还在运行，代码正在当前进程的内核线程中运行。所以这里加锁的目的之一就是：即使我们将进程的状态改为了 RUNABLE，其他的 CPU 核的调度器线程也不可能看到进程的状态为 RUNABLE 并尝试运行它。否则的话，进程就会在两个 CPU 核上运行了，而一个进程只有一个栈，这意味着两个 CPU 核在同一个栈上运行代码。

然后调用 `sched()`。

> 执行本函数的时候，位于进程 A 的内核态线程的上下文里面。

`sched`

```
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;
}      rsc, 17年前 • Changes to allow use
```

可以看出，sched 函数基本没有干任何事情，只是做了一些合理性检查，如果发现异常就 panic。为什么会有这么多检查？因为这里的 XV6代码已经有很多年的历史了，这些代码经历过各种各样的 bug，相应的这里就有各种各样的合理性检查和 panic 来避免可能的 bug。我将跳过所有的检查，直接走到位于底部的 `swtch` 函数。

> 执行本函数的时候，位于进程 A 的内核态线程的上下文里面。

`swtch`

`swtch(&p->context, &mycpu()->context);`

```
.globl 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 是一段汇编代码，其实际上做了以下事情：

- 当前 context （一些寄存器）保存到第一个参数的位置
- 从第二个参数的位置加载 context

所谓的 context 实际上就是寄存器。

这里是从进程 P 的 context 切换到 `mycpu()->context` ，也就是调度器线程。（每个 CPU 都有且仅有一个调度器线程）。

> 执行本函数的时候，从进程 A 的内核态线程的上下文切换到了调度器线程的上下文。

`scheduler` 调度器

这里就是调度器代码，也就是线程调度的核心。

从上文的 `swtch` 实际上就会跳到这个函数的 `swtch`，只不过是以汇编的型式，不那么直观。

不过要注意，是从 `swtch` 那里返回的，而不是从函数开始重新运行的。实际是从下图里的箭头位置开始运行的。

```
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);
    }
  }
}
```

这里的过程是：

- 系统启动，第一次运行 `scheduler` 函数
- 在 `swtch` 里，调度器切换到进程运行。（也就是进程 A/B 开始运行了）
- 进程时间片用完（或者其他情况），进行线程调度
- 进程的内核态线程 `swtch` 到调度器的 `swtch`，也就上图的箭头位置。

接下来我将简单介绍一下 `p->lock`。从调度的角度来说，这里的锁完成了两件事情。

- 首先，出让 CPU 涉及到很多步骤，我们需要将进程的状态从 RUNNING 改成 RUNABLE，我们需要将进程的寄存器保存在 context 对象中，并且我们还需要停止使用当前进程的栈。所以这里至少有三个步骤，而这三个步骤需要花费一些时间。所以锁的第一个工作就是在这三个步骤完成之前，阻止任何一个其他核的调度器线程看到当前进程。锁这里确保了三个步骤的原子性。从 CPU 核的角度来说，三个步骤要么全发生，要么全不发生。
- 第二，当我们开始要运行一个进程时，p->lock 也有类似的保护功能。当我们要运行一个进程时，我们需要将进程的状态设置为 RUNNING，我们需要将进程的 context 移到 RISC-V 的寄存器中。但是，如果在这个过程中，发生了中断，从中断的角度来说进程将会处于一个奇怪的状态。

> 比如说进程的状态是 RUNNING，但是又还没有将所有的寄存器从 context 对象拷贝到 RISC-V 寄存器中。所以，如果这时候有了一个定时器中断将会是个灾难，因为我们可能在寄存器完全恢复之前，从这个进程中切换走。而从这个进程切换走的过程中，将会保存不完整的 RISC-V 寄存器到进程的 context 对象中。所以我们希望启动一个进程的过程也具有原子性。在这种情况下，切换到一个进程的过程中，也需要获取进程的锁以确保其他的 CPU 核不能看到这个进程。同时在切换到进程的过程中，还需要关闭中断，这样可以避免定时器中断看到还在切换过程中的进程。

现在我们在 scheduler 函数的循环中，代码会检查所有的进程并找到一个来运行。

然后再次调用了 swtch 函数，切换到另一个进程的内核态线程的上下文。

> 执行本函数的时候，位于调度器线程的上下文，然后切换到进程内核态线程的上下文。

进程调度

XV6 的进程调度算法十分简单，核心就是下面的图：

```
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);
}
```

其遍历所有进程，找到 RUNNABLE 的进程，然后运行。

其使用的是 **轮询调度算法(Round-Robin Scheduling)**，简单说就是顺序遍历，找到一个能被调度的。

算法的优点是其简洁性，它无需记录当前所有连接的状态，所以它是一种无状态调度。

> 关于轮转为什么叫做 Round-Robin 的原因 https://zhuanlan.zhihu.com/p/84799744

实验

实验介绍

实验内容 1 - 观察进程调度

```
> git add .
> git commit -m 'message'    #暂存修改
> git switch lab4-1-sched    #切换到本实验的分支
```

任务：

- 修改 XV6 代码，在进程被调度时，以及进程让出 CPU 时，打印相关信息，以观察进程调度过程
- 在进程要被调度时，输出 `Pid %d is going to run on CPU %d`
- 在进程因用完时间片时，输出 `Pid %d is yield on CPU %d, for running out of time slice`
- 在进程因等待 IO 而让出时，输出 `Pid %d is yield on CPU %d, for waitting I/O`

Tips：

- 修改 `kernel/proc.c` 的 `scheduler` 函数
- 使用 `kprintf` 输出到控制台
- 为了整洁，对于 PID <= 2 的进程不要输出任何东西
- 对于 XV6 而言
  - PID 1 为 init 进程
  - PID 2 为 sh 进程

预期输出：

```
> make qemu
...
xv6 kernel is booting
init: starting sh
$ ./test
Pid 3 is going to run on CPU 0
Pid 3 is yield on CPU 0, for waitting I/O
...
Pid 4 is yield on CPU 0, for running out of time slice
Pid 5 is going to run on CPU 0
Pid 5 is yield on CPU 0, for running out of time slice
Pid 6 is going to run on CPU 0
Pid 6 is yield on CPU 0, for running out of time slice
Pid 4 is going to run on CPU 0
Pid 4 is yield on CPU 0, for running out of time slice
Pid 5 is going to run on CPU 0
Pid 5 is yield on CPU 0, for running out of time slice
Pid 6 is going to run on CPU 0
Pid 6 is yield on CPU 0, for running out of time slice
Pid 4 is going to run on CPU 0
fib == 2178309
...
...
```

```
xv6 kernel is booting

init: starting sh
$ ./test
Pid 3 is going to run on CPU 0
Pid 3 is yield on CPU 0, for waitting I/O
Pid 3 is going to run on CPU 0
Pid 3 is yield on CPU 0, for running out of time slice
Pid 3 is going to run on CPU 0
Pid 3 is yield on CPU 0, for waitting I/O
Pid 3 is going to run on CPU 0
Pid 3 is yield on CPU 0, for waitting I/O
Pid 3 is going to run on CPU 0
Pid 3 is yield on CPU 0, for waitting I/O
Pid 3 is going to run on CPU 0
Pid 3 is yield on CPU 0, for waitting I/O
Pid 3 is going to run on CPU 0
Pid 3 is yield on CPU 0, for waitting I/O
Pid 4 is going to run on CPU 0
Pid 4 is yield on CPU 0, for running out of time slice
Pid 5 is going to run on CPU 0
Pid 5 is yield on CPU 0, for running out of time slice
```

实验内容 2 - 实现多级反馈队列调度算法

```
> git add .
> git commit -m 'message'    #暂存修改
> git switch lab4-2-mlfq    #切换到本实验的分支
```

任务：

- 将 XV6 的调度算法实现为多级（3 级）反馈队列调度

Tips：

- 修改 `kernel/proc.c` 的 `scheduler` 函数
- 可以基于实验内容 1 进行观察。
- 多级反馈队列（Multi-level Feedback Queue，MLFQ）
- https://pages.cs.wisc.edu/~remzi/OSTEP/Chinese/08.pdf
- 进程在两者情况下会让出 CPU
- 用完时间片
  - 等待外部 IO 设备
- 在 XV6 中全局变量 `ticks` 是对时间中断的计数，可以帮助我们度量时间
- 一旦工作用完了其在某一层中的时间配额（无论中间主动放弃了多少次 CPU），就降低其优先级（移入低一级队列）。
  - 要考虑避免饥饿现象
- 对于实现有几个要点
- 避免死锁，多“防御性编程”

    - 即多用 assert, 或者 if then panic
    - 注意获取锁的顺序
-

预期输出：

```
#define NPROCQUEUE 3      You, 56秒钟前 • add MLFQ
#define S_MLFQ 32

struct spinlock procqueue_lock[NPROCQUEUE];
struct proc *procqueue[NPROCQUEUE][NPROC];

int add_to_mlfq(struct proc*,int);
struct proc *get_runnable_from_mlfq(int level);
int requeue_mlfq();
int scheduler_mlfq();
```

```
xv6 kernel is booting

hart 4 starting
hart 3 starting
hart 1 starting
hart 2 starting
hart 5 starting
hart 6 starting
hart 7 starting
init: starting sh
$ forktest
fork test
fork test OK
```

拓展实验内容

> 对于上文的多级反馈队列，其每个队列有其执行时间上限。请自行设计任务，观察任务执行，探索高效的多级反馈队列的参数设置。

Tips：

- https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf
- https://en.wikipedia.org/wiki/Multilevel_feedback_queue