

Lab4.进程调度

在进行本实验之前，你需要进行分支管理，请在工程目录下输入以下命令：

```
git fetch
git checkout lab4
make clean
```

概述

任何操作系统都可能运行比CPU数量更多的进程，所以需要有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。而在这里，xv6会实现多路复用技术。对于xv6而言，一个用户进程中只包含一个用户线程，所以在这里几乎可以忽略进程和线程的区别，进程和线程都代指同一事物，且不同的用户进程之间由于页表不同而不会共享内存。每一个用户线程都是通过trap技术进入内核使用内核线程来完成用户线程的工作，而且在xv6中内核线程共用一个大页表，所以内核线程是共享内存的。但是这只是最简单的模型，在真实世界的Linux系统中，一个用户进程会包含多个用户线程，且用户线程共享进程的地址空间。

xv6在两种情况下会将每个CPU从一个进程切换到另一个进程：

1. 当进程处于等待设备或者管道I/O完成、等待子进程退出、在sleep系统调用中等待的情况时，会使用睡眠（sleep）和唤醒（wakeup）机制切换。
2. 周期性地强制切换长时间计算而不睡眠的进程，这就是简单的轮转调度。

在本实验中，我们主要是处在在第二种情况下，用最简单的方式探究xv6在进程调度过程中做了什么。

xv6中的进程调度

在进行调度时，xv6需要区分下面几类进程：

- RUNNING（运行态）：当前在某个CPU上运行的进程
- RUNNABLE（就绪态）：一旦CPU有空闲时间就在CPU上运行的进程
- SLEEPING（阻塞态）：在等待I/O或者其他事件的进程

在本实验的情况下，我们主要关注RUNNING和RUNNABLE两种状态的进程即可。实际上，xv6的进程调度过程中就是在定时器中断时将一个RUNNING状态的进程让出CPU，将其转换为RUNNABLE状态，然后将另一个进程转换为RUNNING状态运行。其中还会涉及到很多复杂的操作，称为上下文切换，这个会在后面讨论。

我们会使用一个示例来让你们探索xv6的进程切换过程。

首先，你需要阅读 `kernel/proc.h` 中的结构体 `proc`，相信此前实验中你们也已经多次阅读该结构体，这个结构体就是PCB。下面我们需要关注其中一些变量：
`trapframe`字段保存了用户线程的寄存器，`context`字段保存了内核线程的寄存器，`kstack`字段保存了当前进程的内存栈，`state`字段保存了当前进程状态。

我们准备了一个用户程序 `spin`（在 `user/spin.c` 中可以看到其实现），该程序会创建两个进程，其中一个进程是另一个进程的子进程，两个进程都会进入一个死循环，且它们都不会主动让出CPU。但是在xv6中运行该程序，你可以发现两个进程之间会相互切换。我们会先跳过用户空间的执行，看看在内核中处理定时器中断时，进程如何进行切换。

接下来需要打开gdb，运行用户程序 `spin` 的，然后中断程序的运行，使其在执行 `addiw s1,s1,1` 时中断（可以多执行几次）。在 `kernel/trap.c` 的 `devintr` 函数的207行打一个断点，这里会识别出当前是在响应定时器中断。

开始调试这个过程

你可以在gdb中输入一个命令来显示了当前正在执行的函数以及调用该函数的函数，相信经过前面的实验过程，找到这个正确的命令对你来说不算困难。

因为在 `devintr` 函数中处理定时器中断基本上没有什么内容，我们可以用 `finish` 命令来返回到调用它的函数，和前面得到的显示进行相互验证。并且我们可以看到 `devintr` 函数的返回值是2，这代表了定时器中断。继续向下执行，你会发现，该值为2时才能进入让出CPU的 `yield` 函数。

注意要分辨此时是从用户空间陷入的还是内核空间陷入的。

接下来，可以在gdb中打印当前进程的名字和PID等信息，看看进程切换过程中各种信息的变化。

把上述过程完整记录下来并写在实验报告之中，用自己的话总结发生了什么。

`yield` 函数会调用 `sched` 函数，`sched` 函数在完成一系列检查工作后会调用 `swtch` 函数进行上下文切换，这个过程是由汇编代码完成的，接下来我们会讨论上下文切换。

上下文切换

我们将目光聚焦于 `sched` 函数，在此函数中，我们可以看到在对一系列条件进行检查之后，将执行 `swtch` 函数进行上下文切换。函数 `swtch` 为内核线程切换执行保存和恢复操作。`swtch` 对线程没有直接的了解；它只是保存和恢复寄存器集，称为上下文（contexts）。此前说过 `context` 字段是我们重点关注的字段，其保存了内核线程的寄存器。

当某个进程要主动让出CPU时，该进程的内核线程调用 `swtch` 来保存自己的上下文并返回到调度程序的上下文。阅读源码 `kernel/proc.h` 我们可以看到，每个上下文都包含在一个 `struct context` 中，这个结构体本身包含在一个进程的 `struct proc` 或一个CPU的 `struct cpu` 中。

`swtch` 函数接受两个参数：

```
void swtch(struct context *old, struct context *new);
```

其中当前进程的寄存器会保存在 `old` 中，然后从 `new` 中加载即将要运行的进程的寄存器。

`swtch` 函数的实现在文件 `kernel/swtch.s` 中，这是由汇编代码完成的操作。在 `context` 中，最有趣的是 `ra` 寄存器的值，因为 `ra` 寄存器保存的是当前函数的返回地址，所以在 `swtch` 函数返回后，新切换到的线程会返回到它的 `context` 中 `ra` 寄存器的值对应的地址。调度器线程的 `context` 的 `ra` 对应到 `scheduler` 函数。

另外，`swtch` 函数并没有保存和恢复 `pc` 寄存器。程序计数器并没有有效信息，我们现在知道我们在 `swtch` 函数中执行，所以保存程序计数器并没有意义。但是我们关心的是我们是从哪调用进到 `swtch` 函数的，从 `swtch` 函数返回时，我们希望能够从 `swtch` 函数的调用点继续执行。这里保存的 `ra` 寄存器就保存了 `swtch` 函数的调用点，所以我们将要返回到 `scheduler` 函数中。这一步，通过之前的gdb调试和输出相应的调试信息，你也可以发现其中的奥妙。

还有就是，实际上RISC-V有32个寄存器，而在这里只需要保存和恢复14个寄存器，这是因为在此前我们可以看到，`swtch` 函数被当作一个普通函数来调用，如果不打开它本身我们甚至不知道它是由汇编代码写成的，对于有些寄存器，`swtch` 函数的调用者默认 `swtch` 函数会做修改，所以调用者会在自己的栈上保存这些寄存器，函数恢复时也会自动恢复。所以 `swtch` 函数只需要保存被调用方保存的寄存器 (callee-saved registers) 。

还有一个重要的寄存器是 `sp`，它是当前进程的内核栈地址。在我们从 `swtch` 函数返回之后，`sp` 寄存器的值现在在内存中的 `stack0` 区域中。或许你们对 `stack0` 区域有点陌生，但是其实他在xv6启动的时候就已经出现了，这几乎是你接触过最早的xv6代码之一。

完成 `sp` 寄存器的设置后，其实我们现在已经在调度器线程中了，如果你在整个过程中打印 `ra` 寄存器的值，会发现它已经指向 `scheduler` 函数。虽然我们身处在 `swtch` 函数，但是现在我们实际上位于调度器线程调用的 `swtch` 函数中。调度器线程在启动过程中调用的也是 `swtch` 函数。接下来通过执行 `ret` 指令，我们就可以返回到调度器线程中。

现在我们正运行在CPU拥有的调度器线程中，但其实我们现在返回的 `swtch` 函数并不是刚刚让出CPU的进程的 `swtch` 函数的调用，返回的是之前的调用，所以在 `scheduler` 函数中，我们还需抹去刚刚运行的进程的记录，可以看到 `scheduler` 函数中 `swtch` 函数的下一句是：

```
c->proc = 0;
```

就是将当前CPU运行的进程数设为0，就是让出了。

💡 注意！

这里需要反复思考明白，调度器线程调用了 `swtch` 函数，但是我们从 `swtch` 函数返回时，实际上是返回到了对于 `switch` 的另一个调用，而不是调度器线程中的调用。我们返回到的是另一个进程在很久之前对于 `switch` 的调用。这一点可能会有点让人困惑，但是这就是线程切换的核心。

那么，第一次调用 `swtch` 函数的时候，没有另一个了 `swtch` 函数的调用，这里该怎么办？我们应该伪造一个进程对 `swtch` 函数进行调用。事实上回忆一下之前lab中的 `xv6` 启动，CPU0会设置好其他东西以及调用 `userinit` 函数创建第一个用户进程。之后CPU1和CPU2也会启动，之后就进入 `scheduler` 函数的无限循环中。而 `userinit` 函数会调用创建普通用户进程的 `allocproc` 函数，然后进行一些特殊处理。

`allocproc` 函数会设置好新进程的context中的 `ra` 和 `sp`：

```
memset(&p->context, 0, sizeof(p->context));  
p->context.ra = (uint64)forkret;  
p->context.sp = p->kstack + PGSIZE;
```

在这里，`ra` 寄存器非常重要，根据上文，可以知道这就是新创建的进程的第一个 `swtch` 函数调用会返回的位置。这里设置的 `forkret` 函数就是进程的第一次调用 `swtch` 函数会返回到的位置。从 `swtch` 函数返回就直接跳到了 `forkret` 函数的最开始位置。`forkret` 函数只会在启动进程的时候以这种奇怪的方式运行，这就是 `xv6` 为每个新创建的进程伪造的它上次调用 `swtch` 函数而停止的地方，当一个新创建的进程第一次被 `scheduler` 函数调度的时候，它就会从 `forkret` 函数开始执行。

🔧 在用户线程中进行上下文切换

接下来你需要实现用户线程的上下文切换，你需要将代码添加到 `user/uthread.c` 中的 `thread_create()` 和 `thread_schedule()`，以及 `user/uthread_switch.s` 中的 `thread_switch`。

一个目标是确保当 `thread_schedule()` 第一次运行给定线程时，该线程在自己的栈上执行传递给 `thread_create()` 的函数。另一个目标是确保 `thread_switch` 保存被切换线程的寄存器，恢复切换到线程的寄存器，并返回到后一个线程指令中最后停止的点。

你必须决定保存/恢复寄存器的位置；可以修改 `struct thread` 以保存寄存器。你需要在 `thread_schedule` 中添加对 `thread_switch` 的调用；可以将需要的任何参数传递给 `thread_switch`，但目的是将线程从 `t` 切换到 `next_thread`。

对于如何实现用户线程的上下文切换，其实和内核空间的机制差不多，阅读了上述指导书的内容和对应的源码，相信你已经可以照猫画虎地实现了。

使用 `python3 grade-lab-4 uthread` 可以测试你的程序。

💡 一个有用的调试方案

使用gdb单步调试通过 `thread_switch`：

```
(gdb) file user/_uthread
Reading symbols from user/_uthread...
(gdb) b uthread.c:60
```

在xv6中键入“uthread”运行程序，在 `uthread.c` 的60行的断点处停住，现在你可以输入一些命令来调试。

查询程序 `uthread` 的状态：

```
(gdb) p/x *next_thread
```

检查内存位置的内容：

```
(gdb) x/x next_thread->stack
```

可以跳到 `thread_switch` 的开头：

```
(gdb) b thread_switch
```

真实世界

在这里我们只是介绍了最简单的进程调度，也就是基于时间片的轮询调度。事实上在真实世界中，调度策略非常多，调度这个问题也相当复杂。

例如，允许进程具有优先级。其思想是调度器将优先选择可运行的高优先级进程，而不是可运行的低优先级进程。这些策略可能变得很复杂，因为常常存在相互竞争的目标：例如，操作系统可能希望保证公平性和高吞吐量。此外，复杂的策略可能会导致意外的交互，例如优先级反转。当低优先级进程和高优先级进程共享一个锁时，可能会发生优先级反转，当低优先级进程持有该锁时，可能会阻止高优先级进程前进。

可以用一个简单的代码示例来解释上面的优先级反转问题：

```
void low() { // 最低优先级
```

```
mutex_lock(&lock);
// 先到先得
}

void medium() { // 中优先级
    while (1) ;
}

void high() { // 高优先级
    sleep(1);
    mutex_lock(&lock);
    ...
}
```

当 `medium` 到来时，持有互斥锁的 `low` 被赶下了CPU，那么 `high` 虽然贵为最高优先级，也没法持有锁进入CPU运行了。

扩展阅读：火星上的优先级反转

事实上这个优先级反转的情况在火星上出现过一次，来自1997年7月4日登陆火星的Sojourner “探路者” (PathFinder)。

这个问题又被称为[The First Bug on Mars](#)，感兴趣的同学可以点击链接查看。

实现更多的调度策略

对调度策略感兴趣的同学可以实现更多的调度策略，推荐从简单轮询策略的设置不同时间片开始，想要挑战自己的同学可以尝试实现优先级调度，当然，实现应对优先级反转的策略就不要挑战了，其实Linux也没有引入解决该问题的机制。