Assignment 1 - Preemption

2025 年 3 月 17 日

1 背景

在当前的 xv6 实现中，我们对内核线程的调度依赖于内核线程主动放弃 CPU 控制权，即调用 `yield` 方法将自己标记为 `RUNNABLE`，并将 CPU 控制权交还给 scheduler。

在该作业的代码包中，`void init()` 函数是内核启动完成后的第一段代码。它会创建 8 个工作线程，分别对共享变量 `count` 进行累加。每个工作线程会先打印 `starting`，然后进行计数，每计数 1000 次会睡眠 500ms 并打印当前进度，最后退出时会打印 `exiting` 并调用 `exit` 退出。init 会等待所有子线程退出后，打印 `count` 的值并退出。

使用 `make run` 运行内核：

```
[INFO  0,-1] bootcpu_init: start scheduler!
[sched 0,-1] scheduler: switch to proc pid(1)
[INFO  0,1] init: kthread: init starts!
[sched 0,1] sched: switch to scheduler pid(1)
[sched 0,-1] scheduler: switch to proc pid(2)
[WARN  0,2] worker: thread 2: starting
[INFO  0,2] worker: thread 2: count 1000, sleeping
[INFO  0,2] worker: thread 2: count 2000, sleeping
...
[INFO  0,2] worker: thread 2: count 9000, sleeping
[INFO  0,2] worker: thread 2: count 10000, sleeping
[WARN  0,2] worker: thread 2: exiting
[sched 0,2] sched: switch to scheduler pid(2)
[sched 0,-1] scheduler: switch to proc pid(3)
[WARN  0,3] worker: thread 3: starting
...
[WARN  0,3] worker: thread 3: exiting
[sched 0,3] sched: switch to scheduler pid(3)
[sched 0,-1] scheduler: switch to proc pid(4)
[WARN  0,4] worker: thread 4: starting
...
[WARN  0,4] worker: thread 4: exiting
[sched 0,4] sched: switch to scheduler pid(4)
[sched 0,-1] scheduler: switch to proc pid(5)
[WARN  0,5] worker: thread 5: starting
...
[WARN  0,9] worker: thread 9: exiting
kthread: all threads exited, count 80000
[INFO  0,1] init: kthread: init ends!
```

我们发现，scheduler 启动后先切换到了 **init** (pid = 1) 线程，随后轮流执行 8 个工作线程，并且只在每个工作线程主动 `exit` 后才切换到下一个线程。

如果我们使用 `make runsmp` 启动有 4 个核心的内核，尽管内核是多核心并发的，但是在工作调度上仍然是 1234 先，5678 后，并且只在线程主动退出时，才切换到其他线程执行。

```
$ make runsmp | grep -E "sched|init"
[INFO  0,-1] bootcpu_init: start scheduler!
[INFO  2,-1] secondarycpu_init: start scheduler!
[INFO  1,-1] secondarycpu_init: start scheduler!
[INFO  3,-1] secondarycpu_init: start scheduler!
[sched 0,-1] scheduler: switch to proc pid(1)
[INFO  0,1] init: kthread: init starts!
[sched 0,1] sched: switch to scheduler pid(1)
[sched 0,-1] scheduler: switch to proc pid(2)
[sched 3,-1] scheduler: switch to proc pid(3)
[sched 2,-1] scheduler: switch to proc pid(4)
[sched 1,-1] scheduler: switch to proc pid(5)
[sched 0,2] sched: switch to scheduler pid(2)
[sched 0,-1] scheduler: switch to proc pid(6)
[sched 3,3] sched: switch to scheduler pid(3)
[sched 3,-1] scheduler: switch to proc pid(7)
[sched 2,4] sched: switch to scheduler pid(4)
[sched 2,-1] scheduler: switch to proc pid(8)
[sched 1,5] sched: switch to scheduler pid(5)
[sched 1,-1] scheduler: switch to proc pid(9)
[sched 0,6] sched: switch to scheduler pid(6)
[sched 0,-1] scheduler: switch to proc pid(1)
[INFO  0,1] init: thread 2 exited with code 114516, expected 114516
[INFO  0,1] init: thread 3 exited with code 114517, expected 114517
[INFO  0,1] init: thread 4 exited with code 114518, expected 114518
[INFO  0,1] init: thread 5 exited with code 114519, expected 114519
[INFO  0,1] init: thread 6 exited with code 114520, expected 114520
[sched 0,1] sched: switch to scheduler pid(1)
[sched 3,7] sched: switch to scheduler pid(7)
[sched 3,-1] scheduler: switch to proc pid(1)
[sched 2,8] sched: switch to scheduler pid(8)
[INFO  3,1] init: thread 7 exited with code 114521, expected 114521
[INFO  3,1] init: thread 8 exited with code 114522, expected 114522
[sched 1,9] sched: switch to scheduler pid(9)
[INFO  3,1] init: thread 9 exited with code 114523, expected 114523
[INFO  3,1] init: all threads exited, count 80000
[INFO  3,1] init: init ends!
```

显然，这样的调度实现并不能满足操作系统对多任务调度的要求。现代操作系统会使用抢占式调度 (Preemption)。当系统中有多个就绪进程需要运行时，操作系统需要让它们轮流使用 CPU。即使操作系统只有单个 CPU，也会通过抢占暂停当前进程的执行并开始执行下一个就绪的进程。

Wikipedia: Preemption (computing) In computing, preemption is the act of temporarily interrupting an executing task, with the intention of resuming it at a later time. This interrupt is done by an external scheduler with no assistance or cooperation from the task. This preemptive scheduler usually runs in the most privileged protection ring, meaning that interruption and then resumption are considered highly secure actions. Such changes to the currently executing task of a processor are known as context switching.

2 问题描述

在本次作业中，请你在给定的代码基础上修改，实现对内核线程的抢占调度。你应该在每个时钟中断时调用 **yield** ，使当前进程放弃 CPU 并回到 scheduler。

本次作业一共 4 分，有 3 个 Checkpoint 和一份报告。三个 checkpoint 分别为 2 分、1 分、1 分，报告不占分，但是要求写。你的报告应该符合以下要求：

- PDF 文件格式的报告。
- **每个 checkpoint 运行成功的截图。**
- **记录完成这个作业一共花了多少时间。**
- 对于指引中提出的思考问题，你不需要在报告中回答。
- （可选）描述你在完成这个作业时遇到的困难，或者描述你认为本次作业还需要哪些指引。
- （可选）你可以在报告中分享你完成这个作业的思路。
- 可选部分越简洁越好。

你修改后的操作系统应该能够正确执行到 `infof("init ends!");` 处，并且没有任何 PANIC 消息，除了 *init process exited* 和 *other CPU has panicked* 以外。

2.1 正解提示

如果你的实现正确，8 个子线程应该会**轮流**计数 1000 次，然后开始下一轮循环。你的日志应该长这样：

```
[INFO  0,1] init: kthread: init starts!
[sched 0,1] sched: switch to scheduler pid(1)
[sched 0,-1] scheduler: switch to proc pid(2)
[WARN  0,2] worker: thread 2: starting
[INFO  0,2] worker: thread 2: count 1000, sleeping
[sched 0,2] sched: switch to scheduler pid(2)
[sched 2,-1] scheduler: switch to proc pid(4)
[sched 1,-1] scheduler: switch to proc pid(3)
[sched 3,-1] scheduler: switch to proc pid(5)
[WARN  2,4] worker: thread 4: starting
[WARN  1,3] worker: thread 3: starting
[sched 0,-1] scheduler: switch to proc pid(6)
[WARN  3,5] worker: thread 5: starting
[WARN  0,6] worker: thread 6: starting
[INFO  2,4] worker: thread 4: count 2000, sleeping
[INFO  0,6] worker: thread 6: count 5000, sleeping
[INFO  1,3] worker: thread 3: count 3000, sleeping
[INFO  3,5] worker: thread 5: count 4000, sleeping
[sched 0,6] sched: switch to scheduler pid(6)
[sched 1,3] sched: switch to scheduler pid(3)
[sched 2,4] sched: switch to scheduler pid(4)
[sched 0,-1] scheduler: switch to proc pid(7)
[sched 3,5] sched: switch to scheduler pid(5)
[WARN  0,7] worker: thread 7: starting
[sched 2,-1] scheduler: switch to proc pid(9)
[sched 1,-1] scheduler: switch to proc pid(8)
[WARN  2,9] worker: thread 9: starting
[sched 3,-1] scheduler: switch to proc pid(2)
[INFO  2,9] worker: thread 9: count 7000, sleeping
```

```
[INFO  0,7] worker: thread 7: count 6000, sleeping
[WARN  1,8] worker: thread 8: starting
[INFO  1,8] worker: thread 8: count 8000, sleeping
...
```

3 代码解读

在本次作业的代码中，*main.c* 文件会使用 `create_kthread` 创建一个内核线程 **init**。该方法的原型定义如下：

```
int create_kthread(void (*fn)(uint64), uint64 arg);
```

该方法会申请一个 PCB `struct proc` 结构体。随后将 `context.ra` 设置为 `first_sched_ret` 函数，将其 context 的 s1 和 s2 寄存器分别设为传入的 fn 和 arg 参数。

`first_sched_ret` 是该进程第一次被调度器调度到时执行的方法，它首先释放自己进程的锁，然后打开中断，跳转到 create_kthread 时传入的参数 fn 处开始执行指定的内核线程的代码。arg 将作为一个可选的参数被传给 fn。在执行 worker 方法时，CPU 的中断应该保持为开的状态，允许时钟中断进入 Trap。

```
static void first_sched_ret(void) {
    // s0: frame pointer, s1: fn, s2: uint64 arg
    void (*fn)(uint64);
    uint64 arg;
    asm volatile("mv %0, s1":"=r"(fn));
    asm volatile("mv %0, s2":"=r"(arg));

    release(&curr_proc()->lock);
    intr_on();
    fn(arg);
    panic("first_sched_ret should never return. You should use exit to terminate kthread");
}
```

4 指引步骤

4.1 Checkpoint. 1

到下次提示前，只使用 `make run` 运行单 CPU 的内核。

修改 `kernel_trap` 中对 `SupervisorTimer` 中断的处理，调用 *yield* 函数放弃继续执行该进程。

Hint. 1 你可能会遇到空指针的问题，注意 `yield` 函数表示放弃继续执行当前进程，这显然是要求当前有一个进程跑在 CPU 上时才合理。`yield` 方法会对 `curr_proc()` 返回的 `struct proc*` 进行解引用。而当调度器在 scheduler 方法里面等待中断时 (`wfi`)，该值为 NULL。我们应该只对内核进程进行 yield，而不包括 scheduler。

Hint. 2 你会遇到错误 *sched should never be called in kernel trap context.*。这是因为在原先的内核调度模型中，我们不允许抢占内核线程，而只允许抢占用户线程。在每次进入 trap 时，我们会增加 `mycpu()->inkernel_trap` 计数器，并用该计数器来捕捉内核 Trap 环境中遇到的锁或调度的问题。例如我们不期望会在内核的 Trap Handler 中再次遇到 Interrupt，这种行为被称为 Nested Interrupt（嵌套中断）。为了解决这个问题，你只需要在 `yield()` 前将该值清零，`yield()` 返回后复原即可。

Checkpoint Passed. 你应该会看到日志中 8 个工作线程能轮流运行计数。但是，在出现第一个退出的进程时，发生了 kernel panic。此时你应该遇到 *kerneltrap: not from supervisor mode* 的错误，以及 kernel panic。这表示你已经完成了 **Checkpoint. 1**.

4.2 Checkpoint. 2

观察 Kernel Panic 报告中的 `sstatus` 寄存器值，`SPP:U` 表示 `sstatus.SPP` 位是 0，表示进入该 Trap 前，CPU 处于 U-mode。但是，我们从未有代码预期会返回到 U-mode 执行；并且，返回到用户空间的唯一途径是在 `sret` 前清空 `sstatus` 的 `SPP` 位，但是我们的代码从未有主动清空过该标志位。

找到内核为什么会降级到 U-mode，并解决该问题。

Hint. 3 请你翻阅特权级手册 *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*，章节 *4.1.1*，*Supervisor Status Register (sstatus)* 的说明，找到什么时候这个标志位会被 CPU 清空。

Hint. 4 该错误**总是**会在第一个进程 exit 后出现。`exit` 函数会将自己标记为 `ZOMBIE` 并在结束时调用 `sched` 方法切换到 scheduler，scheduler 会寻找下一个 `RUNNABLE` 的进程继续执行。但是，所有的进程均停留在中断处理函数 `kernel_trap` 中的 `yield` 处。

Checkpoint Passed. 你的代码大概率能在单 CPU 上完整跑完 init 函数，并看到以下提示：

```
[INFO  0,1] init: init ends!
[PANIC 0,1] os/proc.c:218: init process exited
```

这表明你已经完成了 **Checkpoint. 2**.

4.3 Checkpoint. 3

该 Checkpoint 需要你通过多 CPU 下的压力测试。

将 *sched.c* 中两条 logf 注释以避免大量输出；将 *timer.h* 中对 `TICKS_PER_SEC` 改为 400，提高中断的频率；将 *nommu_init.c* 中的 `NTHREAD` 改为 15，将 `SLEEP_TIME` 改为 50。

反复使用 `make runsmp` 启动多 CPU 的操作系统，你应该会遇到程序卡死或者 kernel panic 的问题。尝试解决该问题。

Hint. 5 Checkpoint. 2 提示了在 `sched` 前后，有一些 CSR 的值会发生变化。

Hint. 6 参照你解决 Checkpoint. 2 的方法，阅读特权级手册章节 *3.3.2 Trap-Return Instructions* 关于 sret 指令的细节。

Checkpoint Passed. 你能稳定通过 `make runsmp` 测试，连续 10 次以上。在输出的末尾，你应该看到以下日志：

```
[INFO  1,1] init: init ends!
[PANIC 1,1] os/proc.c:218: init process exited
[PANIC 2,-1] os/trap.c:45: other CPU has panicked
[PANIC 0,-1] os/trap.c:45: other CPU has panicked
[PANIC 3,-1] os/trap.c:45: other CPU has panicked
```

5 提交

将你的 PDF 格式的报告放到与 Makefile 文件相同的目录下，运行 `make handin`，这会生成一个 handin.zip 压缩包。上传该文件到 Blackboard 即可。