OS-lab3 report

刘泓尊 2018011446 计84

Chapter3 实现的内容

1. 实现了多道程序和分时多任务OS，包括程序加载、任务切换和基于时钟中断的抢占调度
2. 实现了stride调度算法，以及限制程序运行的最长时间20s
3. 支持sys_yield, sys_gettime, sys_set_priority 系统调用

编程作业：

编程作业

为了完成本章作业，我在任务控制块 `TaskControlBlock` 中添加了如下成员, 同时实现了计算pass值的函数。

```
pub struct TaskControlBlock {
    pub task_cx_ptr: usize,
    pub task_status: TaskStatus,
    pub task_stride: isize, // stride value
    pub task_priority: isize, // priority, default is 16
    pub task_run_duration_ms: usize, // 已经执行的时间
    pub task_last_start_time: usize, // 上一次被调度进CPU的时间(ms)
}
impl TaskControlBlock {
    // 计算 pass 值
    pub fn get_task_pass(&self) -> isize {
        BIG_STRIDE / self.task_priority
    }
}
```

1. **stride调度算法**

   初始化的时候把stride值初始化为0， priority初始化为16。每次任务被调度的时候，将其stride 加上其对应的步长 pass = BigStride / P.priority。其中BigStride设置为 `0x7FFFFFFF` ，保证足够大又不会溢出。每次需要调度的时候，从ready态的进程中找出stride最小的进程进行调度即可（我使用的是线性查找）。注意 stride 和 priority, BigStride等都是无符号整数usize, 比较的时候按照有符号比较，这样就能保证溢出后仍能正确运行。

2. **sys_gettime**

   框架里实现的计时函数其实精确度不高，我自己实现了 `get_time_us()` 来获得以微妙为单位的时间，更方便该系统调用的要求。

   之后利用编译器将寄存器里的值解析为 `*mut TimeVal` 类型，进行赋值即可。

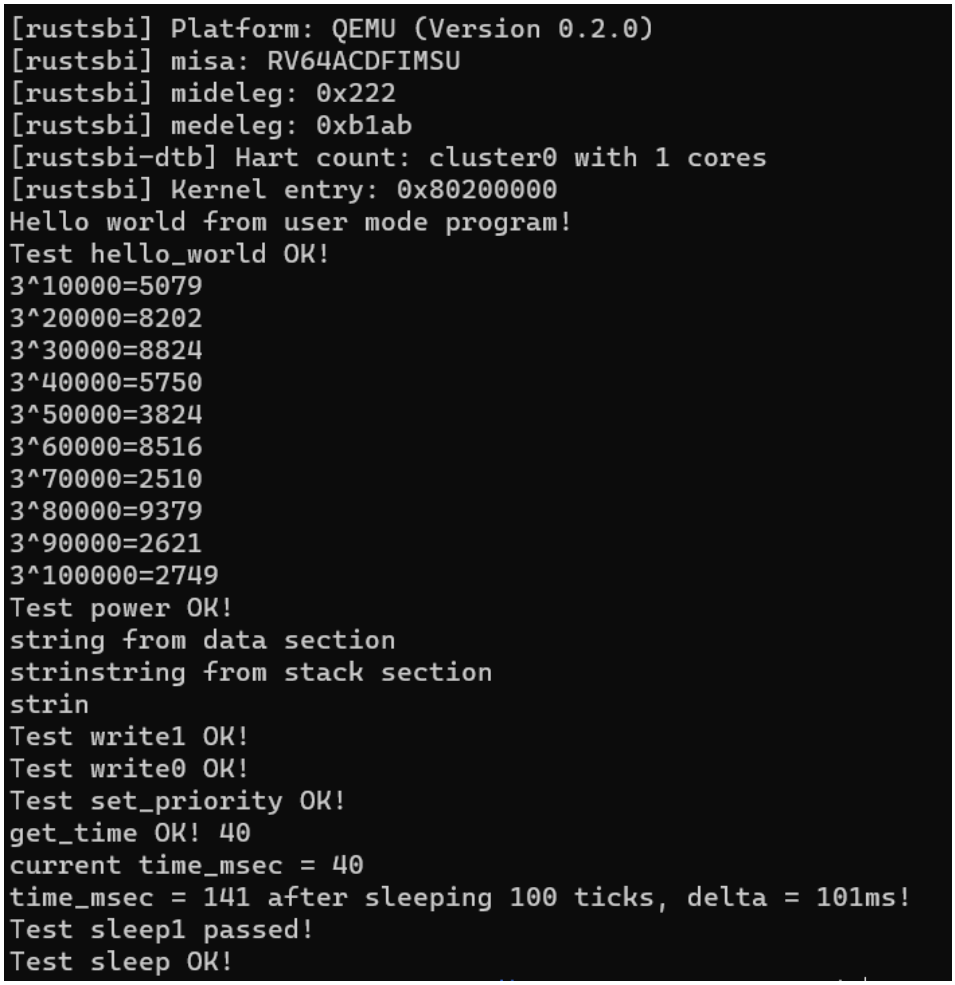
```rust
pub fn get_time_sys(ts: *mut TimeVal, _tz: usize) -> isize {
    unsafe {
        if let Some(ts) = ts.as_mut() {
            (*ts).usec = get_time_us() % USEC_PER_SEC;
            (*ts).sec = get_time_us() / USEC_PER_SEC;
            return 0
        }
    }
    -1
}
```

3. **sys_set_priority**

   这个系统调用十分简单，在`TASK_MANAGER`里提供一个接口用来设置当前正在运行进程的priority即可。注意如果priority < 2或者priority > isize::MAX就将其判断为非法，返回-1.

测试结果截图

`make run CHAPTER=3_0`:



`make run CHAPTER=3_1`:

```
[rustsbi] Platform: QEMU (Version 0.2.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000
AAAAAAAAAA [1/5]
BBBBBBBBBB [1/5]
CCCCCCCCCC [1/5]
AAAAAAAAAA [2/5]
BBBBBBBBBB [2/5]
CCCCCCCCCC [2/5]
AAAAAAAAAA [3/5]
BBBBBBBBBB [3/5]
CCCCCCCCCC [3/5]
AAAAAAAAAA [4/5]
BBBBBBBBBB [4/5]
CCCCCCCCCC [4/5]
AAAAAAAAAA [5/5]
BBBBBBBBBB [5/5]
CCCCCCCCCC [5/5]
Test write A OK!
Test write B OK!
Test write C OK!
```

`make run CHAPTER=3_2`:

```
[rustsbi] Platform: QEMU (Version 0.2.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000
priority = 6, exitcode = 43968000
priority = 9, exitcode = 66828800
priority = 8, exitcode = 58421200
priority = 5, exitcode = 38644000
priority = 7, exitcode = 49919200
priority = 10, exitcode = 76350400
```

问答作业

1. **简要描述这一章的进程调度策略。何时进行进程切换？如何选择下一个运行的进程？如何处理新加入的进程？**

   OS在初始化的时候设置了 `sie.stie` 使得S特权级时钟中断被开启。每当时钟中断 `Interrupt::SupervisorTimer` 到来的时候，Trap分发会调用 `suspend_current_and_run_next()` 来暂停当前应用并切换到下一个。该函数会先将当前进程设置为 `TaskStatus::Ready`，然后调用 `run_next_task` 尝试切换到下一个应用。

   接下来，首先是调用调度算法(Round-Robin或Stride)，找到下一个被调度的进程号。然后设置该进程状态为 `TaskStatus::Running`，将当前进程stride值加上BigStride / priority (=pass), 然后将current_task设置为该进程号（同时还可以记录被换出进程和被调入进程的执行时间, 以限制最大运行时间）。最后调用ch2中的__switch进行 进程上下文和执行流的切换。

   至于调度算法，如果用示例中的RR算法，则从当前进程号后开始，循环一圈寻找第一个status == Ready的进程即可，相当于循环队列。如果是Stride调度算法，则从ready进程中找stride值最小的进程即可。

2. **C版代码使用一个进程池（也就是一个 struct proc 的数组）管理进程调度，当一个时间片用尽后，选择下一个进程逻辑在 chapter 3 相关代码，也就是当第 i 号进程结束后，会以 i -> max_num -> 0 -> i 的顺序遍历进程池，直到找到下一个就绪进程。C 版代码新进程在调度池中的位置选择见 chapter5相关代码，也就是从头到尾遍历进程池，找到第一个空位。**

    1. **在目前这一章（chapter3）两种调度策略有实质不同吗？考虑在一个完整的 os 中，随时可能有新进程产生，这两种策略是否实质相同？**

       因为ch3不涉及中途加入新进程参与调度的情况，所以C和Rust两版都是顺序循环查找 当前进程号之后，第一个遇到的Ready 的程序，所以不会有区别。

       如果随时有新进程产生，C是将任务加到数组中第一个空位，而Rust是加到队列里。这就会产生不同：即先加入的进程不一定是先执行的。

       考虑这样的执行流：

       C uCore:

       p1 p2 p3 -- -- -- [run p1]

       -- **p2** p3 -- -- -- [finish p1, run p2]

       p4 -- **p3** p5 -- -- [p2 spawn p4, p5, finish p2]

       p4 -- -- **p5** -- -- [finish p3]

       p4 -- p3 **p5** -- -- [run p5]

       p4 -- p3 p5 -- -- [run p4]

       其中**加粗**的进程是正在执行的进程。可以看到C uCore中，先加入的进程可能后执行，这并不是队列。

       Rust rCore:

       p1 p2 p3 [run p1]

       p2 p3 [finish p1, run p2]

       p3 p4 p5 [p2 spawn p4, p5, finish p2, run p3]

       p4 p5 [finish p3, run p4]

       p4 p5 [run p4, finish p4]

       p5 [run p5]

       可以看到Rust是遵循队列调度的，先加入的进程先被调度。

       所以这两种策略实质不同，是因为插入方法不一致导致的。

    2. **其实 C 版调度策略在公平性上存在比较大的问题，请找到一个进程产生和结束的时间序列，使得在该调度算法下发生：先创建的进程后执行的现象。你需要给出类似下面例子的信息（有更详细的分析描述更好，但尽量精简）。同时指出该序列在你实现的 stride 调度算法下顺序是怎样的?**

       我给出如下例子：

时间点	0	1	2	3	4	5	6	7
运行进程		p1	p2	p3	p5	p1	p4	
事件	p1、p2、p3产生		p2结束	p4, p5 产生，p3结束	p5结束	p1结束	p4结束	

产生顺序：p1、p2、p3、p4、p5。第一次执行顺序: p1、p2、p3、p5、p4。违反了公平性。

这就是因为插入的时候不是在队尾，而是在进程池中的第一个空位。

如果在**rCore中使用Stride调度算法**执行上述序列：

时间点	0	1	2	3	4	5	6	7
运行进程		p1	p2	p3	p4	p5	p1	
事件	p1、p2、p3产生		p2结束	p4, p5 产生，p3结束	p4结束	p5结束	p1结束	

产生顺序：p1、p2、p3、p4、p5。第一次执行顺序: p1、p2、p3、p4、p5。实现了公平性。

这是因为p4, p5第一次被执行前，stride都是0，都是最小的，所以先于被换出的p1来执行。

3. **stride 算法深入**

   stride算法原理非常简单，但是有一个比较大的问题。例如两个 pass = 10 的进程，使用 8bit 无符号整形储存 stride， p1.stride = 255, p2.stride = 250，在 p2 执行一个时间片后，理论上下一次应该 p1 执行。

   1. **实际情况是轮到 p1 执行吗？为什么？**

      不是p1.

      因为p2执行1个时间片后，其stride变成了 (250 + 10) % 256 = 4 < p1的stride, 所以选择最小stride的进程一定不是p1.

我们之前要求进程优先级 >= 2 其实就是为了解决这个问题。可以证明，在不考虑溢出的情况下, 在进程优先级全部 >= 2 的情况下，如果严格按照算法执行，那么 STRIDE_MAX – STRIDE_MIN <= BigStride / 2。

2. **为什么？尝试简单说明**

   简要用一下**反证法**:

   如果进程优先级 >= 2，那么每次stride更新时候的pass值 <= BigStride / 2. 假设在 某个时间片t 以后第一次出现 p1.stride - p2.stride > BigStride / 2, 那么该时间片t 必定调度了p1, 不然上个时间片t-1就满足 p1.stride - p2.stride > BigStride / 2 了。

   在这个时间片前，p1.stride(t-1) = p1.stride(t) - p1.pass >= p1.stride(t) - BigStride / 2 > p2.stride(t). 因为这个时间片执行的是p1, 所以 p2.stride(t-1) = p2.stride(t). 所以 p1.stride(t-1) > p2.stride(t-1) , 所以该时间片t 会调度p2. **矛盾**!

   所以任一时间片都有 STRIDE_MAX – STRIDE_MIN <= BigStride / 2.

3. **请补全如下 `partial_cmp` 函数（假设永远不会相等）**

   从上面的分析可以知道，如果两者差超过BigStride / 2, 那么更小的Stride实际上更大（溢出了）。如果两者差 <= BigStride / 2，那么Stride更大者更大。

   同时注意 rust 的 usize 只能大的减小的，给我们的实现来了一点复杂。

```rust
use core::cmp::Ordering;

struct Stride(u64);

const BigStride: u8 = 255;

impl PartialOrd for Stride {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        let stride_1: u64 = self.0;
        let stride_2: u64 = other.0;
        if stride_1 < stride_2 {
            // less but stride_1 overflow
            if (stride_2 - stride_1) > (BigStride / 2).into() {
                Some(Ordering::Greater)
            } else {
                Some(Ordering::Less)
            }
        } else {
            // greater but stride_2 overflow
            if (stride_1 - stride_2) > (BigStride / 2).into() {
                Some(Ordering::Less)
            } else {
                Some(Ordering::Greater)
            }
        }
    }
}

impl PartialEq for Stride {
    fn eq(&self, other: &Self) -> bool {
        false
```

```
32        }
33    }
```