清华大学
Tsinghua University

rCore-Tutorial操作系统中进程调度算法的设计与实现

毕业论文答辩

清华大学计算机系 马思源

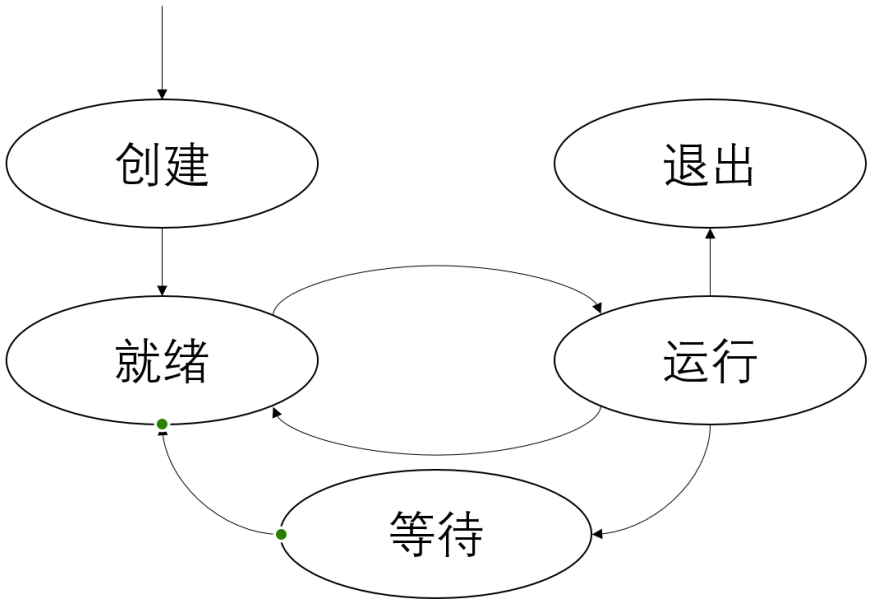
2022年6月10日

指导教师：陈渝

一、引言

课题背景

- 进程（Process）是OS对一个应用程序的一次运行过程的抽象，是OS进行资源分配和调度的基本单位。
- OS需要以适当的方法来管理进程，调整不同进程的状态，选择分配处理机。这一过程便被称为进程调度（Process Scheduling），OS所选择的分配策略则被称为调度算法



- rCore-Tutorial（第三版，全称应为rCore-Tutorial-v3）是面向操作系统初学者使用rust语言编写的教学用单核操作系统，采用时间片轮转调度算法。
- 相关工作：xv6是MIT为其操作系统课程开发的操作系统，采用多路复用、时间片轮转调度策略，使用进程锁保证多核调度的同步互斥。

课题目标

- 本课题的目标主要包括两个方向
- 拓展rCore-Tutorial进程调度，实现并分析多种单核调度算法。
- 将单核rCore-Tutorial拓展为多核操作系统，进行多核进程调度的分析。

二、rCore-Tutorial进程管理模块架构

rCore-Tutorial进程管理模块

- 进程管理模块的主要组成包括:
- 进程控制块（PCB）其中记录了进程从创建到退出所涉及到的各种数据信息。
- 进程管理器（Manager）维护着就绪状态的所有进程的队列，对外开放add和fetch两个接口。
- 处理器管理结构（Processor）是CPU的抽象，记录CPU当前运行状态，并执行控制流的切换

rCore-Tutorial进程管理模块

- 进程的生命周期：
- 创建：操作系统为用户提供了fork和exec系统调用
- 运行：Processor循环地向Manager获取需执行的进程。成功获取后，切换执行流至该进程
- 停止与等待：suspend方法修改进程状态，使之进入就绪队列，切换执行流至idle，回到Processor之内。
- 退出：当进程执行结束后需要调用exit退出，回收掉部分资源，余下资源留待父进程调用waitpid回收。

rCore-Tutorial进程管理模块

- 当前调度策略：时间片轮转（Round-Robin，RR）调度算法。
- Manager中采用的队列方式为简单的先进先出（FIFO）式。
- rCore-Tutorial打开了时钟中断，设置了10ms的定时器，每过去一个时间片则触发时钟中断，暂停当前进程，重新选取进程执行。

三、单核进程调度算法的实现

单核进程调度算法

- 批处理系统：主要耗时为处理器计算，可以预知执行时间。
  - SJF, STCF, HRRN
- 交互式系统：无法再去预知执行时间，需要能够根据进程的动态运行状态进行调整
  - MQ, MLFQ, Lottery, Stride
- 实时计算机系统：要求计算机能够在给定时间内对外部要求作出反应。
  - RMS, EDF

批处理系统的调度 SJF&STCF

- 最短作业优先（SJF）调度算法选择预期执行时间最短的进程执行。
- 最短完成时间优先（STCF）是可抢占的SJF版本。当一个比当前运行进程剩余时间更短的任务到来，STCF会抢占当前运行进程。

- 问题：
  - 获得进程的预期执行时间
  - 实现抢占

批处理系统的调度 SJF&STCF

- 获得进程的预期执行时间
  - 利用exec系统调用在创建时由用户传入
  - 剩余时间通过每次执行时间的记录来计算
- 实现抢占
  - 由于fork+exec机制，rCore-Tutorial本来就实现了抢占=>实现不抢占
  - 标记当前运行进程，每次新进程exec后，重排就绪队列时，有标记的排在前面

```
task_prediction: INIT_RUNNING_TIME,
task_last_start_time: 0,
task_complete_time: INIT_RUNNING_TIME as isize,
```

批处理系统的调度 HRRN

- 由于SJF的判断标准比较单调，在其基础上改进得到最高响应比优先调度算法（HRRN），即选择就绪队列中响应比R值最高的进程：

$$R = \frac{W + S}{S}$$

w: 等待时间(waiting time); s: 执行时间(service time).

- 不采用浮点运算，比较

$$w_1 * s_2 > w_2 * s_1$$

交互式系统的调度 MQ

- 多级队列调度（Multi-level Queue）算法将就绪队列分成多个单独队列，根据进程的属性为其分配一个固定的优先级，再按照优先级放入对应的分就绪队列中。
- 每个队列内部单独调度，队列间按照优先级进行抢占式调度。

- 添加新的系统调用sys_set_priority来让用户程序有方法设置进程的优先级。

交互式系统的调度 MLFQ

- 多级反馈队列（MLFQ）调度算法允许进程在队列之间迁移，具有可变优先级。其基本原则是
  - 创建进程并让进程首次进入就绪队列时，设置进程的优先级为最高优先级，进入最高队列。
  - 进程用完其时间配额后，就会降低其优先级。
  - 经过一段时间后，将所有就绪进程设回最高优先级，加回最高队列。

```
if priority == 0{
    self.ready_queue.get_mut( index: 0).unwrap().push_back(task);
}
else{
    self.ready_queue.get_mut( index: priority - 1).unwrap().push_back(task);
}
```

交互式系统的调度 FSS

- 公平共享（FSS）调度：基于每个进程的优先级，分配给该进程同比例的处理器执行时间
- Lottery和Stride

- 彩票调度（Lottery Scheduling）：给每个进程发彩票，进程优先级越高，所得到的彩票就越多；然后每次调度，举行一次彩票抽奖，抽出来的号属于哪个进程，哪个进程就能运行。
- 步长调度（Stride）：每个进程有一个优先级反比的步长（Stride）属性值，操作系统会定期记录每个进程的总步长，即行程（pass），并选择拥有最小行程值的进程运行。

交互式系统的调度 Lottery

- Lottery
  - 使用rust的no_std随机数生成器生成总数范围内的随机数模拟彩票抽取

```
let lucky_dog : usize =  rng.gen_range(0..self.total_counts);
//println!("{}",lucky_dog);
let mut tmp_sum : usize = 0;
for (ind : usize, &val : usize) in self.lottery_array.iter().enumerate(){
    tmp_sum += val * TICKET_X;
    if tmp_sum > lucky_dog{
        self.lottery_array.remove( index: ind);
        self.total_counts -= val * TICKET_X;
        return self.ready_queue.remove( index: ind);
    }
}
```

实时计算机系统的调度 RMS

- 单调速率（RMS）调度算法采用抢占的、静态优先级的策略，调度周期性任务。
- 周期短的进程以高优先级率先被调度

- 由于RMS处理周期性任务，类比STCF，在exec时传入任务的周期，按照周期大小排序就绪队列选择执行。

实时计算机系统的调度 EDF

- 最早截止期限优先（EDF）调度根据截止期限动态分配优先级。截止期限越早，优先级越高

- 将进程到达时间和exec传入的截止时间加和得到deadline，最小者优先被调度。
- 为了体现周期任务的ddl变化，添加新的系统调用sys_cycle，用于执行周期任务，同步增加ddl和定时器。

```rust
pub fn sys_cycle(period: usize) -> isize {
    let task: Arc<TaskControlBlock> = current_task().unwrap();
    let mut task_inner: RefMut<TaskControlBlockInner> = task.inner_exclusive_access();
    let expire_ms: usize = task_inner.task_deadline;
    task_inner.task_deadline += period;
    drop(task_inner);
    add_timer(expire_ms, task);
    block_current_and_run_next();
    0
}
```

四、进程调度算法的测试、分析与比较

SJF和STCF

- 测试STCF的抢占
- 结果
  - sjf4抢占sjf3
  - sjf5未抢占sjf4

```
static TESTS: &[&str] = &[
    "sjf1\0",
    "sjf2\0",
    "sjf3\0",
    "sjf4\0",
    "sjf5\0",
];

static TIMES: [usize;5] = [
    10000,
    100000,
    1000,
    500,
    500,
];
```

```
**************/
sjf1 Arrive at 95
sjf2 Arrive at 96
sjf3 Arrive at 96
I am sjf3
current time_msec = 168
sjf3 running...
sjf4 Arrive at 400
I am sjf4
current time_msec = 425
sjf4 running...
sjf4 running...
sjf4 running...
sjf5 Arrive at 708
sjf4 running...
798059
time_msec = 965, delta = 540ms, sjf4 OK!
I am sjf5
current time_msec = 965
sjf5 running...
sjf5 running...
sjf5 running...
sjf5 running...
798059
time_msec = 1476, delta = 511ms, sjf5 OK!
sjf3 running...
sjf3 running...
sjf3 running...
808111
time_msec = 2266, delta = 2098ms, sjf3 OK!
I am sjf1
I am sjf2
QEMU: Terminated
```

MQ和MLFQ

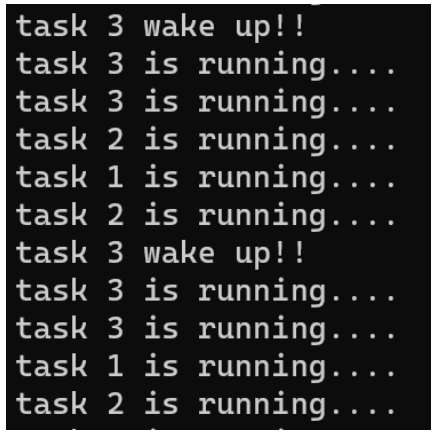
- 测例包括三个任务，利用sleep系统调用采取睡眠唤醒方式模拟IO交互。任务一为单纯的CPU计算型；任务二在前半部分为CPU计算，之后变为I/O交互型；任务三则为交互型
- 按照两个队列的MQ算法，初始任务一二置于后台队列，任务三置于前台队列。
- 即使任务二转为了交互型，但仍和任务一同等优先级。**任务二被唤醒后，既无法抢占任务三，甚至在任务三睡眠后也可能无法优于任务一得到响应**

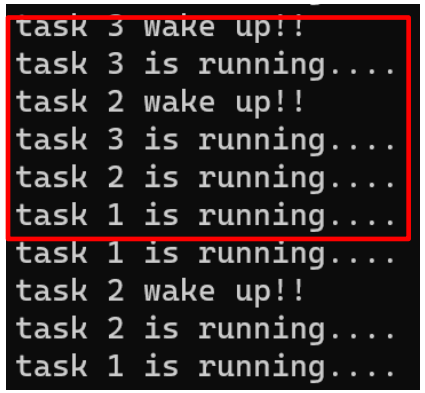
```
task 3 wake up!!
task 3 is running....
task 3 is running....
task 2 is running....
task 1 is running....
task 2 is running....
task 3 wake up!!
task 3 is running....
task 3 is running....
task 1 is running....
task 2 is running....
```

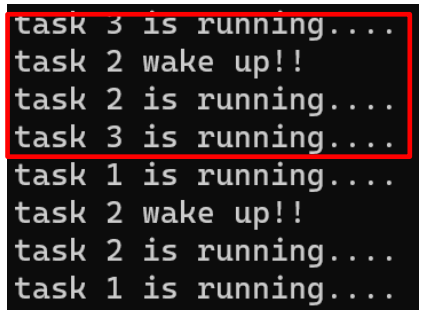
```
task 3 wake up!!
task 3 is running....
task 2 wake up!!
task 3 is running....
task 1 is running....
task 2 is running....
task 1 is running....
task 1 is running....
task 2 wake up!!
task 3 wake up!!
task 3 is running....
task 3 is running....
task 2 is running....
```

MQ和MLFQ

- 对于MLFQ算法，在任务二是CPU计算型时，和MQ基本一致，优先级下降比较慢的任务三会在每次唤醒时优先执行。
- 当任务二状态转换后，优先级的一般顺序基本是3>2>1，因此任务三仍会优先执行，但任务二则能先于一被响应。
- 当进程优先级重置后，同为I/O型的任务二三回到了同一起跑线，因此或许还会出现任务二能够抢占任务三得到响应的现象，







Lottery和Stride

- 测试
  - 期望一定时间内本进程得到的运行次数与priority成正比
- Lottery
  - 基本合理，但由于随机数的不确定性波动较大
- Stride
  - 4000ms的结果即很符合期望

```
**************/
lottery0 Arrive at 82
lottery1 Arrive at 82
lottery2 Arrive at 83
lottery3 Arrive at 83
lottery4 Arrive at 84
lottery5 Arrive at 84
priority = 5, exitcode = 37896000, ratio = 7579200
priority = 8, exitcode = 49154000, ratio = 6144250
priority = 9, exitcode = 63818400, ratio = 7090933
priority = 10, exitcode = 63311200, ratio = 6331120
priority = 7, exitcode = 49090000, ratio = 7012857
priority = 6, exitcode = 34812400, ratio = 5802066
```

```
**************/
lottery0 Arrive at 87
lottery1 Arrive at 88
lottery2 Arrive at 88
lottery3 Arrive at 89
lottery4 Arrive at 89
lottery5 Arrive at 89
priority = 9, exitcode = 172399600, ratio = 19155511
priority = 10, exitcode = 214190400, ratio = 21419040
priority = 8, exitcode = 145900000, ratio = 18237500
priority = 5, exitcode = 84779200, ratio = 16955840
priority = 6, exitcode = 124425200, ratio = 20737533
priority = 7, exitcode = 137840000, ratio = 19691428
```

```
priority = 5, exitcode = 31210400, ratio = 6242080
priority = 6, exitcode = 39798000, ratio = 6633000
priority = 7, exitcode = 46374400, ratio = 6624914
priority = 8, exitcode = 51412000, ratio = 6426500
priority = 9, exitcode = 60354400, ratio = 6706044
priority = 10, exitcode = 67719600, ratio = 6771960
```

RMS和EDF

- 测试rms1，2两个周期任务，周期为2000，1000；每周期运行时长800，500
- 2对1进行抢占，实现正常调度

```
static TESTS: &[&str] = &[
    "rms1\0",
    "rms2\0",
];

// static TIMES: [usize;2] = [
//     800,
//     500,
// ];

static PERIODS: [usize; 2] = [
    2000,
    1000,
];
```

```
**************/
rms1 Arriving at 78
rms2 Arriving at 78
current time_msec = 1, I am rms2
798059
time_msec = 495, delta = 494ms, rms2 complete
current time_msec = 496, I am rms1
current time_msec = 1010, I am rms2
798059
time_msec = 1502, delta = 492ms, rms2 complete
149523
time_msec = 1768, delta = 1272ms, rms1 complete
current time_msec = 2008, I am rms2
```

RMS和EDF

- 然而单调速率调度有一个限制，调度 N 个进程的最坏情况下的 CPU 利用率为

$$N(2^{\frac{1}{N}} - 1)$$

- 如果超过了这个值，RMS不能保证他们一定能被调度。

- rms2同样要求抢占rms1，但此时1已经无法按时完成。

```
static TESTS: &[&str] = &[
    "rms1\0",
    "rms2\0",
];

// static TIMES: [usize;2] = [
//     800,
//     500,
// ];

static PERIODS: [usize; 2] = [
    1500,
    1200,
];
```

```
**************/
rms1 Arriving at 80
rms2 Arriving at 81
current time_msec = 6, I am rms2
798059
time_msec = 501, delta = 495ms, rms2 complete
current time_msec = 501, I am rms1
current time_msec = 1209, I am rms2
798059
time_msec = 1698, delta = 489ms, rms2 complete
562774
time_msec = 1775, delta = 1274ms, rms1 complete
Panicked at src/bin/rms1.rs:33, rms1 outlimit!
```

RMS和EDF

- EDF调度测试上面提到的rms1，2
- ddl动态变化，1未被2抢占
- 此后则不断变换优先级，最终使得调度成功。

```
rms1 Arriving at 3095
rms2 Arriving at 3096
current time_msec = 21, I am rms2
798059
time_msec = 519, delta = 498ms, rms2 complete
current time_msec = 520, I am rms1
562774
time_msec = 1320, delta = 800ms, rms1 complete
current time_msec = 1320, I am rms2
798059
time_msec = 1815, delta = 495ms, rms2 complete
current time_msec = 1816, I am rms1
562774
time_msec = 2607, delta = 791ms, rms1 complete
```

五、rCore-Tutorial多核扩展

多核操作系统的启动

- 为QEMU的启动添加设置-smp
- 修改入口文件entry.asm，获取当前核id并分配各个核的启动栈
- rust_main中对不同核进行不同的行为设定
  - 全局初始化工作仅一个核进行
  - 其他核等待全局工作结束开始各自的boot
  - 使用原子变量同步信息

```
if cpu_id == CONTROL_CPU{
    println!("Global initialization start...");
    clear_bss();
    ..
    finish_global_init();
}
wait_global_init();
println!("Hello world from CPU {:x}!", cpu_id);
..
boot_finish();
wait_all_booted();
```

多核操作系统的启动

- 创建全局STDOUT实例并为其加锁，实现输出的互斥，避免截断现象。
- 由原有的单核互斥包装改为使用spin库的Mutex互斥锁

多核操作系统的进程管理模块

- 多处理机MultiProcessor
- 顾名思义将单核的Processor实例扩展为Vec<Mutex<Processor>>
- 使用entry.asm获取到的id来访问对应的处理机

- 进程的同时执行带来许多同步互斥问题

多核操作系统的进程管理模块

- switch是执行流切换所需的方法，分为存储当前执行流的寄存器、栈指针等核加载目标执行流的相关数据两个步骤
- 单核rCore-Tutorial在暂停进程时首先将进程放回就绪队列再进行switch
- 多核环境下可能在放回队列后就被另一处理机取出进行另一个switch，若加载快于前者的存储阶段则产生异常。

- 将放回队列挪到switch后，即回到Processor内部再进行add_task

多核操作系统的进程管理模块

- exit中，需要将当前进程的children挂载到INITPROC下，来让当前进程退出后其子进程仍能正常回收。若按照单核rCore-Tutorial的实现逻辑如下，有多个获取锁的步骤。

```
let mut inner = task.inner_exclusive_access();
..
let mut initproc_inner = INITPROC.inner_exclusive_access();
for child in inner.children.iter() {
    child.inner_exclusive_access().parent =
    Some(Arc::downgrade(&INITPROC));
    initproc_inner.children.push(child.clone());
}
```

- 多核支持父子进程同时进入exit处理，因此会出现父进程拿着INITPROC的锁等待子进程，子进程拿着自己的锁等待INITPROC的死锁现象。
- 因此需要调整一下获取锁的顺序，将获取INITPROC放到获取当前任务inner的前面进行，这样同一时间只能有一个处理器获取到INITPROC的锁来进行下面的操作，从而解决的父子锁的冲突。

多核下的调度结果

- 对于单核调度，遵循耗时和程序数目的正比关系，即循环执行每个任务，最终用时为单一时间与个数的乘积。
- 对于四核调度，平均耗时在程序不超过4个时相差不大，符合四核逻辑，且应是在数目为4时才充分利用了CPU资源。
- 当进程数达到5个时，多出一个进程被四个处理器平摊，总耗时应增加了单一进程的1/4与表中数值基本一致。当进程数大于5后，耗时将以线性关系增长，到达10后耗时为5的一倍，与表中数据一致。

表 5.1　样例程序组在单核和多核环境下的执行情况①

样例程序数目	单核平均耗时/ms	四核平均耗时/ms
1	993(1032)	1100(1143)
2	1967(2043)	1108(1155)
3	2956(3080)	1084(1130)
4	3942(4110)	1068(1121)
5	4912(5125)	1318(1426)
10	9800(10217)	2600(2734)

① 括号里的数值为包括启动程序和回收子进程的总用时。

六、总结

总结

- rCore-Tutorial的进程管理模块被大大充实，能为使用者提供更完整的进程调度参考。
- 多处理机的教学系统更能贴合当下的计算机实际；另一方面，在已实现的多核rCore-Tutorial基础上，还可以进行更多的多核功能扩展，为rCore-Tutorial的进一步完善开辟了新的道路。

- 仍有部分单核调度算法没有被本文所实现，而且对于已实现的进程调度的完备性还需要进一步的检验与测试。
- 从单核到多核这一执行环境的巨大转变也为原有的rCore-Tutorial模块提出了新的要求，还有一些潜在的问题等待被发现和解决。

清華大學
Tsinghua University

谢谢!