

Lab3 RV64内核线程调度

赵伊蕾：3200104866

1 实验目的

- 了解线程概念, 并学习线程相关结构体, 并实现线程的初始化功能。
- 了解如何使用时钟中断来实现线程的调度。
- 了解线程切换原理, 并实现线程的切换。
- 掌握简单的线程调度算法, 并完成两种简单调度算法的实现。

2 实验内容

在原有Lab2已经完成的时钟中断基础之上, Lab3将模拟完成线程调度。在每次时钟发生中断的时候, 操作系统将判断切换线程还是留在线程继续执行。如果需要切换线程, 则涉及到进程调度算法的使用。在本实验中, 实现了SJF和Priority这两种算法。

2.1 初始化进程

对于每个进程, 我们将为其分发4KB大小的物理页进行存储。其中物理页包括了高地址的栈, 低地址的结构体。中间部分可以通过sp指针存入任何想存入的数据。

由于我们的OS就是初始的idle线程, 但是我们并没有在框架中对其task_struct进行设置, 所以除了要设置task[1]~task[NR_TASKS-1]的这些进程, 还不能忘记idle。

特别需要注意的是, 由于新生成的线程, 其上下文切换的时候没有内容进行切换, 所以要先为其提供一个dummy函数进行跳转。dummy部分不需要自己完成, proc函数中已经给出。

另外, 除了初始化进程之外, 我们先需要初始化分配内存。已经为我们写好了mm_init和task_init两个函数, 只需要在_start中调用就行。

Code

```
void task_init() {
    idle = (struct task_struct *)kalloc();
    idle->state = TASK_RUNNING;
    idle->counter = 0;
    idle->priority = 0;
    idle->pid = 0;
    current = idle;
    task[0] = idle;

    for(int i = 1; i < NR_TASKS; i++){
        task[i] = (struct task_struct *)kalloc();
        task[i]-> state = TASK_RUNNING;
        task[i]-> counter = 0;
        task[i]-> priority = rand();
        task[i]-> pid = i;
        task[i]-> thread.ra = (uint64)__dummy;
        task[i]-> thread.sp = (uint64)task[i] + PGSIZE;
```

```

}

printfk("...proc_init done!\n");
printfk("Hello RISC-V\n");
printfk("idle process is running!\n");
}

```

2.2 dummy函数调用

由于时钟中断的触发会切换进程，为了顺利返回，进程的上下文都将保存在栈上。当上下文再次被调度时，会将上下文从栈上恢复。但是由于我们创造的新线程的栈是空的，所以第一次调度的时候，是没有上下文需要切换的。因此我们设置一个特殊的返回函数 `__dummy` 为第一次调度提供返回。在线程初始化的时候，我们将为每一个线程都设置特殊的返回 `__dummy`，然后由 `__dummy` 跳转到 `dummy` 函数。

在 `entry.S` 中我们需要完成 `__dummy`。因为线程新被分配出来之后，其 `sepc` 值并没有被赋值，所以如果我们不为其赋值返回地址，则很有可能线程跳转到下文之后无法再返回。所以需要先赋值 `sepc` 之后在 `sret`。

```

la s0, dummy
csrw sepc, s0
sret

```

2.3 实现线程切换

在 `switch_to` 函数中设置切换线程的条件，如果当前线程和下一个线程相同(id相同)，则不需要切换，否则需要调用 `__switch_to` 进行线程的上下文切换。`__switch_to` 函数主要是实现将前一个线程的 `thread_struct` 中的相关数据载入到 `ra`, `sp`, `s0~s11` 这些寄存器中，然后再读入后一个线程的 `thread_struct`。

`__switch_to` 上下文切换:

```

sd ra, 5*8(a0)
sd sp, 6*8(a0)
sd s0, 7*8(a0)
...
sd s11, 18*8(a0)
ld ra, 5*8(a1)
ld sp, 6*8(a1)
ld s0, 7*8(a1)
...
ld s11, 18*8(a1)
ret

```

上面从 `5*8` 的位置开始存储 `ra` 寄存器是因为我们可以看到 `proc.h` 中的数据结构 `task_struct`，一共有六个变量。`thread_info` 目前为空不需要考虑，`thread_struct` 是 `ra`、`sp` 这些寄存器存储的地方，所以在 `task_struct` 数据结构中，`ra` 寄存器的位置是第 5 个变量(前面是 `state`、`counter`、`prioirty`、`pid`)，然后每个变量值都是 64 位的 `int`，也就是 8 Bytes，所以是从 `5*8` 的位置开始存储 `ra`。

```

/* 线程状态段数据结构 */
struct thread_struct {

```

```

uint64 ra;
uint64 sp;
uint64 s[12];
};

/* 线程数据结构 */
struct task_struct {
    struct thread_info* thread_info;
    uint64 state;      // 线程状态
    uint64 counter;    // 运行剩余时间
    uint64 priority;   // 运行优先级 1最低 10最高
    uint64 pid;        // 线程id

    struct thread_struct thread;
};

```

switch_to函数

该函数实现了两个线程之间的切换。从 `prev` 线程切换到 `next` 线程。主要实现是通过 `__switch_to` 对寄存器的值进行“调换”，换成现在进程的值，以便于程序后续跳转过去继续执行。

```

void switch_to(struct task_struct* next) {
    /* YOUR CODE HERE */
    if(current->pid != next->pid){
        struct task_struct* prev = current;
        current = next;
        printk("switch to [PID = %d COUNTER = %d PRIORITY = %d]\n", next->pid, next->counter, next->priority);
        __switch_to(prev, next);
    }
}

```

2.4 实现调度入口函数

在时钟中断中调用 `do_timer()` 函数，每个周期都判断一次是否需要重新调度线程。如果当前是 `idle` 线程，也就是最父亲的那个线程，则直接调度，生成 `NR_TASKS-1` 个子线程；如果当前不是 `idle` 线程，则还需判断当前线程运行剩余的时间片是多少，如果即将被运行完，则也需重新调度线程。

Task

1. 如果当前线程是 `idle` 线程 直接进行调度
2. 如果当前线程不是 `idle` 对当前线程的运行剩余时间减1；若剩余时间仍然大于0 则直接返回，否则进行调度。

`do_timer` 函数相当于是在当前时刻判断下一时刻需要执行的进程是什么，如果当前时刻的当前进程剩余时间为 0、或者是 `idle` 初始线程，则需要重新调度，那么就要靠调度算法来决定重新生成一系列线程，然后分析最先进入的是哪个。`schedule` 函数的内容由后续实现。

特别要注意要在 `trap_handler()` 函数中调用 `do_timer()` 函数，系统处理中断的时候去判断是否需要线程的调度。

```

void do_timer(void) {
    /* YOUR CODE HERE */
    if(current->pid == idle->pid)
        schedule();
    else{
        current -> counter --;
        if(current->counter == 0)
            schedule();
    }
}

```

2.5 实现两种线程调度算法

本实验中分别实现了 SJF 和 Priority 算法。

SJF算法

SJF算法的思想很简单。

1. 判断当前的线程是否是全部都运行完了，如果全运行完了，则为他们随机分配counter(运行的时间)。
2. 接下来需要选出所有未运行完的进程中花费时间最少的进程，作为下一个需要执行的进程。

代码如下：

```

void schedule(void) {
    /* YOUR CODE HERE */
    int min = 1;
    int flag = 1; //指示是否需要线程是否全部都进行完了
    for(int i = 1; i < NR_TASKS; i++){
        if(task[i]->state == TASK_RUNNING && task[i]->counter != 0){
            flag = 0;
        }
    }
    if(flag){ //重新rand
        printf("\n");
        for(int i = 1; i < NR_TASKS; i++){
            task[i] -> counter = rand();
            printf("SET [PID = %d COUNTER = %d]\n", task[i]->pid, task[i]->counter);
        }
        for(int i = 2; i < NR_TASKS; i++){
            if(task[i]->counter < task[min]->counter)
                min = i;
        }
    }else{
        min = 1;
        //找到第一个非零的下标
        for(int i = 1; i < NR_TASKS; i++){
            if(task[i]->counter != 0){

```

```

        min = i;
        break;
    }
}
for(int i = min + 1; i < NR_TASKS; i++){
    if(task[i] -> counter != 0 && task[i]->counter < task[min]->counter)
        min = i;
}
}
switch_to(task[min]);
}

```

因为个人习惯，在对线程 `task->counter` 比较的过程中，我是从前往后进行遍历的，所以如果遇到2个运行时间相同的进程，`id` 小的那个会比 `id` 大的进程先被调度。在后面的priority算法中，根据参考的Linux源码的实现，他们是从后往前进行遍历的，所以两个相同运行时间的进程，`id` 大的反而会被先调用。虽然从前往后和从后往前顺序有点区别，但是问题不大。

Priority算法

整体的思路 and SJF 算法基本一致，都需要判断是否所有进程都完成执行了。但是不同之处在于，`Priority` 算法是先选出优先级最高的进程执行，最后如果没找到任何一个进程是剩余时间 `>0` 的，则重新生成一堆进程，然后切换到 `idle` 进程返回。

```

void schedule(void) {
    /* YOUR CODE HERE */
    int next = 0;
    while (1){
        int c = 0;
        int i = NR_TASKS;
        struct task_struct **p = &task[NR_TASKS];
        while(--i){
            if(!*--p)
                continue;
            //find max priority, index is next
            if((*p)->state == TASK_RUNNING && (*p)->counter != 0 && (*p)->priority > c
){
                c = (*p)->priority;
                next = i;
            }
        }
        if(c) break;
        printk("\n");
        for(p = &task[1]; p < &task[NR_TASKS]; p++){
            if(*p){
                (*p)->counter = ((*p)->counter >> 1) + (*p) -> priority;
                printk("SET [PID = %d PRIORITY = %d COUNTER = %d\n", p-&task[0], (*p)->priority, (*p)->counter);
            }
        }
    }
}

```

```

}
switch_to(task[next]);
}

```

3 思考题

1. 在 RV64 中一共用 32 个通用寄存器, 为什么 `context_switch` 中只保存了14个?

一共保存的14个寄存器除了 `ra` 其他都是 `callee saved registers`。需要保存 `callee saved registers` 是因为系统采用被调用者保存策略的话, 被调用的过程必须保存它要用的寄存器, 保证不会破坏过程调用者的程序执行环境。保存 `ra` 是因为我们需要记录下上下文切换之后的返回的地址。

2. 当线程第一次调用时, 其 `ra` 所代表的返回点是 `__dummy`。那么在之后的线程调用中 `context_switch` 中, `ra` 保存/恢复的函数返回点是什么呢? 请同学用 `gdb` 尝试追踪一次完整的线程切换流程, 并关注每一次 `ra` 的变换 (需要截图)。

第一次进行 `switch_to` 时候的返回点应该是 `__dummy`。如下图所示, 我在 `ld sp, 40(a1)` 语句上设置断点, 这样就可以观察 `ra` 寄存器的值在这一语句运行前后的变化。第一次执行该线程的时候, 由于没有下文可以跳转, 所以会先进入 `__dummy`。

```

Register group: general
zero    0x0      0
ra       0x80200174    0x80200174 < dummy>
sp       0x87ffceb0    0x87ffceb0
gp       0x0        0x0
tp       0x80018000    0x80018000
t0       0x80200508    2149582088
t1       0x0        0
t2       0x1000      4096
fp       0x1        0x1

0x802001a8 < __switch_to+36>    sd     s7, 112(a0)
0x802001ac < __switch_to+40>    sd     s8, 120(a0)
0x802001b0 < __switch_to+44>    sd     s9, 128(a0)
0x802001b4 < __switch_to+48>    sd     s10, 136(a0)
0x802001b8 < __switch_to+52>    sd     s11, 144(a0)
B+ 0x802001bc < __switch_to+56> ld     ra, 40(a1)
> 0x802001c0 < __switch_to+60>  ld     sp, 48(a1)
0x802001c4 < __switch_to+64>    ld     s0, 56(a1)
0x802001c8 < __switch_to+68>    ld     s1, 64(a1)

remote Thread 1.1 In:  switch to                                L118  PC: 0x802001c0
Continuing.

Breakpoint 2, __switch_to () at entry.S:101
(gdb) b *0x802001bc
Breakpoint 3 at 0x802001bc: file entry.S, line 117.
(gdb) c
Continuing.

Breakpoint 3, __switch_to () at entry.S:117
(gdb) n

```

但在之后返回的话, 会先返回到 `switch_to` 函数。这里没有显示 `switch_to` 的地址而是显示的 `trap_handler` 是因为: `switch_to` 函数在执行完 `__switch_to` 之后就返回了 `schedule` 函数, 而 `switch_to` 函数又恰好是 `schedule` 函数最后一句, 所以 `schedule` 函数也需要跳转回上一级函数 `do_timer`, `schedule` 又恰好是 `do_timer` 的最后一句。所以, 一级一级跳转回去, 则返回了最先调用的 `trap_handler` 函数。

调用关系如下:

```

+-----trap_handler
|----->do_timer()
|----->schedule()
|----->switch_to()
|----->__switch_to

```

所以读入下一个进程的 `ra` 时，直接返回了最初的 `trap_handler`，只是中间过程的函数调用层数太多了，且都结束了，并不是错误。

```

Register group: general
zero      0x0      0
ra        0x802008fc    0x802008fc <trap_handler+32>
sp        0x87ffdeb0    0x87ffdeb0
gp        0x0      0x0
tp        0x80018000    0x80018000

0x802001b8 <__switch_to+52>    sd    s11,144(a0)
B+ 0x802001bc <__switch_to+56>    ld    ra,40(a1)
> 0x802001c0 <__switch_to+60>    ld    sp,48(a1)
0x802001c4 <__switch_to+64>    ld    s0,56(a1)
0x802001c8 <__switch_to+68>    ld    s1,64(a1)

remote Thread 1.1 In:  switch to
(gdb)

```