Project2 A Simple Kernel 设计文档（Part I）

中国科学院大学
王嵩岳
2020 年 10 月 14 日

1. 任务启动与 Context Switch 设计流程

1.1 PCB 设计信息

根据实验框架的设计，PCB 定义为一个结构体，所有 PCB 作为全局变量数组的成员，存放在全局数据区。此结构体中包含的 PCB 信息如下：

Item	Description	type
kernel_sp	Kernel Stack Point for each process	ptr_t
user_sp	User Stack Point for each process	ptr_t
preempt_count	Count the number of disable_preempt	reg_t
list	Doubly linked list, **prod** and **succ** ptr	list_node_t
pid	Process ID	pid_t
type	Enumerate value for differential type of process/thread	task_type_t
status	Enumerate value for process status	task_status_t
cursor_x	Cursor x position	int
cursor_y	Cursor y position	int

Table 1 PCB Structure (defined in sched.h)

对于 part1 实验的非抢占式内核，由于未引入中断，故 preempt_count 未使用。值得注意的几点如下：

（1） 多内核栈设计：对每一个进程，我们都为其分配一个内核栈，故一个进程实际有两个栈（user stack, kernel stack）。故 PCB 中有两个栈指针。

（2） 寄存器上下文保存在内核栈中，而不是 PCB 中。故 PCB 中没有寄存器的信息。

（3） List 是一个通用双向链表，它有两个域，指向前驱和后继。指向的类型也是 list。故我们需要用框架内的 list_entry()函数来根据 list 域的地址寻找其所在 PCB 的起始地址。

1.2 进程的初始化

1.2.1 Task 是什么

Task 的本质是一个函数，OS 的角色就是分时调度不同的 task 使之占用资源运行。当我们想在 OS 上启动一个 task 时，我们需要获得此 task 的一些基本信息。首先我们必须知道 task 的入口地址，否则将无法跳转至相应位置执行，对每个 task，其结构体中的 entry_point 指示了函数的入口地址，这就是我们调度 task 后需要跳转到的地址。其次，为后续准备，task_info 结构体里还有一个 type 域，它是一个枚举类型，指示不同的进程类型。任务一中不做区分，均为内核线程。

```c
/* task information, used to init PCB */
typedef struct task_info
{
    ptr_t entry_point;
    task_type_t type;
} task_info_t;
```

```c
typedef enum {
    KERNEL_PROCESS,
    KERNEL_THREAD,
    USER_PROCESS,
    USER_THREAD,
} task_type_t;
```

Figure 1 Task_info 结构体 (defined in sched.h)

1.2.2 如何初始化一个 task

已经知道，在 OS 中的每一个进程，都需要一个对应的 PCB 来记录其信息。内核本身的 PCB 定义为 pcb0，它的 PID 为 0。对于之后的 task，我们将其视为一个进程，也需为其分配 PCB。

这里的调度方法，我采用了最简单的轮转调度：即维护一个双向链表 ready_queue，里面存放所有处于 READY 状态进程的 PCB。每次调度取队列的头节点 PCB 启动对应进程，进程结束后将其放回队列尾。

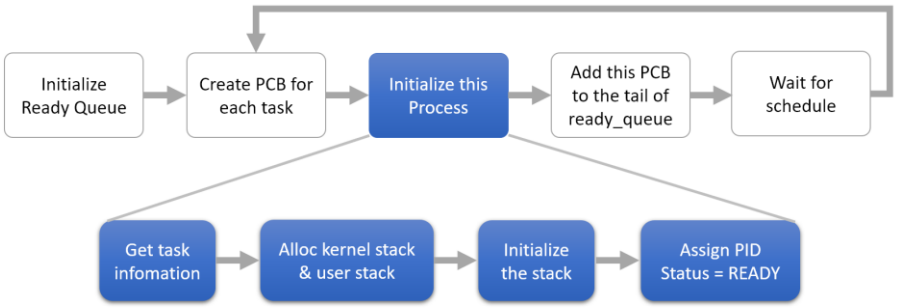

Figure 2 为 task 初始化 PCB 和栈

首先我们需要初始化 ready_queue 等待队列。然后对每一个 task，执行下面的步骤：

(1) 创建 PCB：将 PCB 全局数组的某一个元素 pcb[i]看做是其 PCB。

(2) 初始化 PCB 以及栈：

在函数 init_pcb()中

a) 根据 task 结构体的内容，得到函数入口地址 task->entry_point 和类型 task->type

b) 使用 allocPage() [defined in mm.c]分配一页空间作为内核栈，一页空间作为数据栈，并将两个栈指针分别记录在 pcb[i]->kernel_sp 和 pcb[i]->user_sp

调用函数 init_pcb_stack()，对栈初始化

c) 将 regs_context (所有寄存器上下文)存放在内核栈的最高位置，并维护栈指针。注意，part1 并不会使用这些上下文，此处可简单地将所有寄存器保存值清零，待以后再进行修改。

d) 将 st_context (switch_to 上下文)存放在内核栈中紧挨着 regs_context 的位置，并维护栈指针。注意，在 part1 中，**首次调度时**，这 14 个寄存器将会在调度时调用 switch_to 并恢复至寄存器，故我们需对某些特殊寄存器做初始化：

i. ra 寄存器，需要初始化为 task->entry_point。这是因为 switch_to 函数最后会执行 jr ra 跳转到 task 的入口。

ii. sp 寄存器，我们需要为进程配置好其要使用的栈。这里我让进程使用用户栈，即将 sp 赋值为 user_sp。

(3) 加入 ready_quene，等待调度

分配栈的过程具体图解如下：

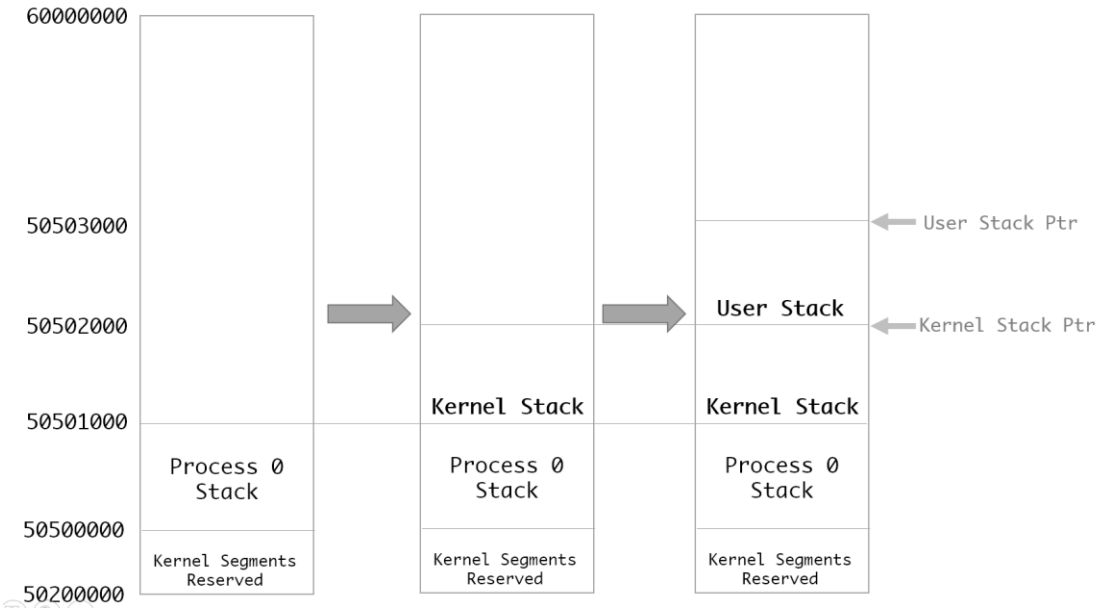

Figure 3 栈空间分配过程图解

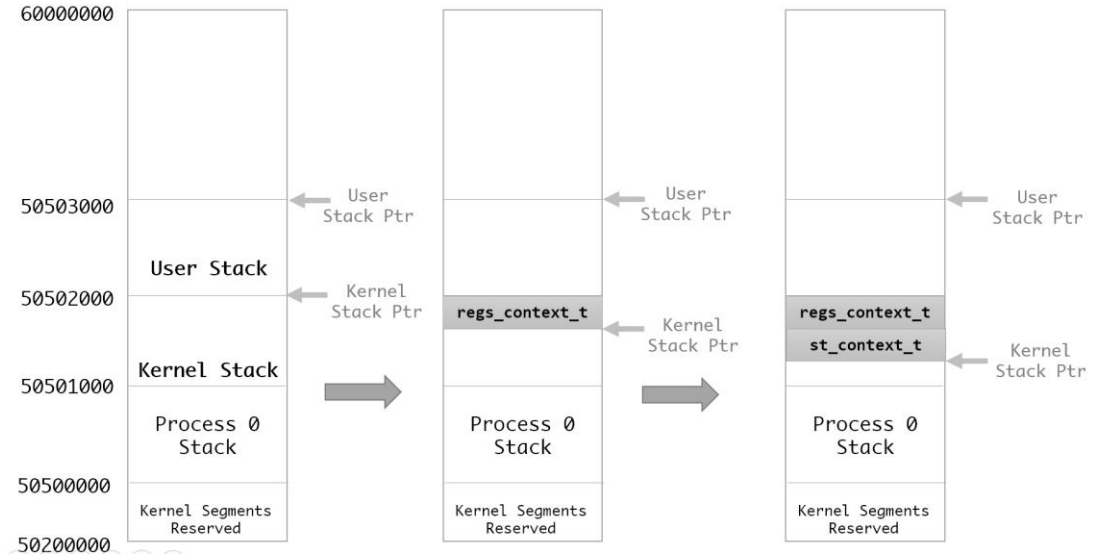

Figure 4 内核栈初始化过程

经过三个进程的加入，最终内存的 layout 如下图：

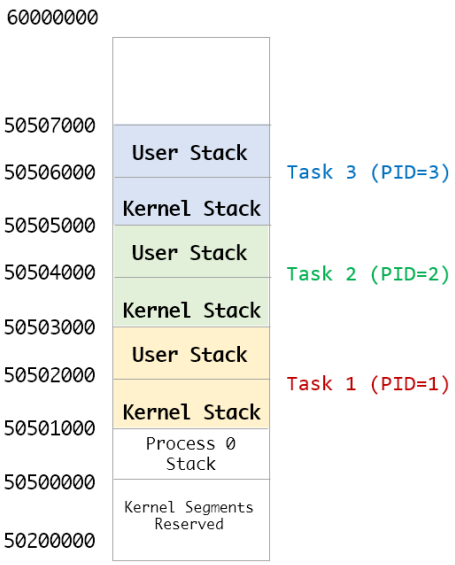

Figure 5 内存 layout

1.3 任务调度

1.3.1 何时调度

Part1 实现的是一个非抢占式内核，这意味着，调度只会发生在两种情况下：内核线程主动调度第一个进程，以及进程自己交出控制权进行调度。请注意下图中 do_scheduler()。

```
while (1) {
  // (QAQQQQQQQQQQ)
  // If you do non-preemptive scheduling, you need to use it
  // to surrender control do_scheduler();
  // enable_interrupt();
  // __asm__ __volatile__("wfi\n\r":::);
  do_scheduler();
};
```

```
void printk_task2(void)
{
    int i;
    int print_location = 2;

    for (i = 0;; i++)
    {
        vt100_move_cursor(1, print_location);
        printk("> [TASK2] This task is to test scheduler. (%d)   ", i);
//        printk("> [TASK2] (%d)   ", i);
        do_scheduler();
    }
}
```

Figure 6 何时调度（左：内核主动调度，右：进程主动交出控制权）

1.3.2 如何调度

调度通过 do_scheduler()实现。它的工作流程如下流程图所示。

Figure 7 do_scheduler()流程图

需要特别注意的有如下几点：

（1） 对于第一个进程，它从 process 0 (内核线程)切换而来，不能执行虚线框的操作，也就是不能把内核线程加入等待队列。

（2） 对于当前进程状态为 RUNNING 的进程，属正常切换，将当前进程状态改为

READY，下一进程改为 RUNNING 即可。

（3） 对当前进程状态为 BLOCK 的进程，它在申请锁的时候已经加入了某个锁的阻塞队列，因此不要执行虚线框（再将其加入等待队列）。

Switch_to 函数的作用就是真正把 CPU 控制权交给下一进程。在非抢占式内核中，由于是进程自己交出的控制权，因此我们只需遵守 ABI 约定，为其保存 13 个由被调用者保存的寄存器和 ra 寄存器即可。Ra 寄存器用于记录返回地址，必须保存。

Name	ABI Mnemonic	Meaning	Preserved across calls?
x0	zero	Zero	– (Immutable)
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	– (Unallocatable)
x4	tp	Thread pointer	– (Unallocatable)
x5-x7	t0-t2	Temporary registers	No
x8-x9	s0-s1	Callee-saved registers	Yes
x10-x17	a0-a7	Argument registers	No
x18-x27	s2-s11	Callee-saved registers	Yes
x28-x31	t3-t6	Temporary registers	No

Figure 8 需要保存的 14 个寄存器

保存上文：

由 Figure 4（标号有超链接）可知，我们的 st_reg_context 存放在本进程的内核栈中。由于我们先前已经维护好了栈指针，因此在保存上文时，我们只需先从 pcb 中读取 kernel_sp，再利用 sp 作为基址寄存器，通过偏移访问不同的寄存器存放地址即可。

这里请注意：由于我们需要使用 sp 寄存器作为基址寄存器，因此我们必须首先将其保存。这里我们选用一个无关的寄存器 t0（这是调用者保存的寄存器，根据 ABI 约定，进程切换前若想继续使用它的值，必须自己保存，因此可以随意使用），先将 kernel_sp 加载至 t0，再把 sp 先保存，然后保存剩下的寄存器。

```
158  ENTRY(switch_to)
159    // save all callee save registers on kernel stack
160    ld s0, PCB_KERNEL_SP(a0)
161    sd sp, SWITCH_TO_SP(s0)
162    ld sp, PCB_KERNEL_SP(a0)
163    /* TODO: store all callee save registers,
164     * see the definition of `struct switchto_context` in sched.h*/
165    sd ra, SWITCH_TO_RA(sp)
166    sd s0, SWITCH_TO_S0(sp)
167    sd s1, SWITCH_TO_S1(sp)
168    sd s2, SWITCH_TO_S2(sp)
169    sd s3, SWITCH_TO_S3(sp)
170    sd s4, SWITCH_TO_S4(sp)
```

Figure 9 保存上文

恢复上文的过程也同理。首先读出将要切换到的进程 PCB 中 kernel_sp，再以 sp 作为基址寄存器恢复上文。同理，sp 应当最后恢复。

```
189     ld s6, SWITCH_TO_S6(sp)
190     ld s7, SWITCH_TO_S7(sp)
191     ld s8, SWITCH_TO_S8(sp)
192     ld s9, SWITCH_TO_S9(sp)
193     ld s10, SWITCH_TO_S10(sp)
194     ld s11, SWITCH_TO_S11(sp)
195     ld sp, SWITCH_TO_SP(sp)
196     mv tp, a1
197     jr ra
198 ENDPROC(switch_to)
```

Figure 10 恢复上文

注意，代码框架约定，tp 应与 current_running 保持一致，故需把 a1（下一进程 PCB 的地址）赋给 tp 寄存器。最后通过 jr ra 跳到之前退出的地址。

关于这里保存的 sp 和 ra，在报告的第三部分也会再次讨论。

2. Mutex lock 设计流程

为了避免不同进程对同一临界区的访问，OS 中引入了锁的抽象来解决这个问题。本次实验要求我们使用互斥锁。对于一个进程而言，要访问临界区，首先要先向 OS 申请锁，再获得锁后访问，访问结束后释放锁。当申请锁被占用时，采用互斥锁的实现会将当前进程挂起（BLOCKED），然后加入该锁的阻塞队列（每个锁都要有自己的阻塞队列），再进行调度切换进程，以最大化利用 CPU 资源。

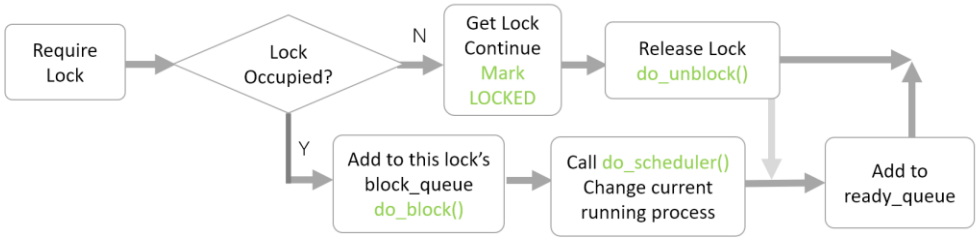

Figure 11 进程锁的生命周期

对互斥锁，当锁被占用时，do_mutex_lock_acquire()方法会调用 do_block()方法：首先将该进程的状态设置为阻塞态（BLOCKED），然后将此进程的 PCB 加入该锁的 block_queue，最后调用 do_scheduler 唤起一个 READY 状态的进程。

当有进程释放锁时，若该锁的 block_queue 不空，则 do_mutex_lock_release()方法会调用 do_unblock()方法：弹出 block_queue 中的一个进程，将其加入 ready_queue，并进行调度。这样就保证了被阻塞的进程当锁释放时还能够继续被唤醒。

3. 思考

下面记录了我设计过程中思考的一些问题

1. **保存的寄存器真的是 task 对应函数退出前一刻的状态吗？sp 指针是谁的栈指针？ra 指向 task 函数代码的某个地址吗？**

实则不然。分析函数调用过程可以发现，函数的调度流程如下：

task() → do_scheduler() → switch_to()

在函数调用时，寄存器值的保存遵守 RISCV64-ABI 约定。因此最终 switch_to 函数内保存的寄存器上下文其实是调用 switch_to()函数进入函数的入口时的状态。

返回地址呢？返回地址记录在 ra 中，其实我们保存的 ra 指向的是 do_scheduler()函数中最后部分的指令。

不禁思考，这是我们需要的上下文吗？岂不保存这么多反而自废武功？

实际上这个想法很多虑，但是确实能彻底搞清函数调用的过程。

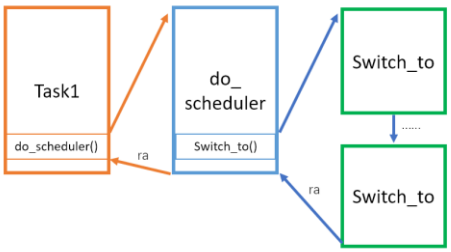

Figure 12 函数调用过程示意图

首先，task1 运行，最后调用 do_scheduler()切换进程。这是一个函数调用，task1 遵守 ABI 约定，会将调用者保存的寄存器保存下来，然后进入 do_scheduler 函数。在调用 switch_to 函数时，返回地址 ra 会被 do_scheduler 函数本身保存至栈中，最后 ra 实际上是 switch_to 函数退出后应跳转至 do_scheduler 函数某处的地址。

恢复时，switch 将 ra 恢复并跳转至 ra 对应位置的指令（do_scheduler 结束部分）。然后 do_scheduler 把先前调用 switch_to 函数时保存的 ra 从栈中 load 出来，再 ret (jr ra)，就跳到了 task1 中 do_scheduler 的下一条指令。栈指针 sp 同理。

通过查看反汇编代码，并结合 QEMU+gdb 调试，可以更清晰地观察到这个过程。

```
1835    50202378:   70a2                ld  ra,40(sp)
1836    5020237a:   7402                ld  s0,32(sp)
1837    5020237c:   6145                addi    sp,sp,48
1838    5020237e:   8082                ret
```

Figure 13 do_scheduler 从栈中恢复 ra 寄存器原值

2. 非抢占式内核中，自旋锁为何一直锁死？

在非抢占式内核测试自旋锁时，会发现当一个进程拿到锁之后，再次调度进入第二个进程，第二个进程就卡死了。这是很好理解的，因为自旋锁的原理就是一直等待锁的释放，而锁的控制权在第一个进程，在第一个进程不被唤起时是不可能主动释放的，因此进程二在申请锁时会陷入死循环。

在后面实现抢占式内核时，对 OS 来说，进程调度由被动变为主动，此时自旋锁可以起到作用。

参考文献

[1] Raspberry Pi OS: Lesson 4.1: Scheduler (https://github.com/s-matyukevich/raspberry-pi-os/blob/master/docs/lesson04/rpi-os.md)

[2] RISC-V Manual for Chinese Reader v2.1, CRVA, ICT

[3] 三十天自制操作系统. 川合秀实