Project 2 Simple Kernel (Part 1) 设计文档

马月骁

2019K8009915025

一、任务启动与非抢占式调度

1. PCB 设计

根据实验框架，在 include/os/sched.h 中将 PCB 以结构体 pcb_t 的形式定义。在 Part 1 中，pcb_t 结构体的具体信息见表 1。

表 1 pcb_t 结构体信息（Part 1）

item	type	Description
kernel_sp	reg_t	Kernel stack pointer
user_sp	reg_t	User stack pointer
preemt_count	reg_t	Count the number of disable_preemt
list	list_node_t	Doubly linked list pointer (previous/next)
pid	pid_t	Process id
type	task_type_t	Type of process/thread
status	task_status_t	Status of process
cursor_x	int	Cursor x position
cursor_y	int	Cursor y position

由于在 Part 1 中，我们实现的位非抢占式调度，过 preemt_count 事实上并未被使用。而在后续的 Part 2 中，为实现中断等操作，将会使用 preemt_count 并添加必需的信息。

Pcb_t 结构体中 list 指向一个双向链表结点（结构体：包含前驱与后继），双向链表的具体定义与维护函数定义在 include/os/list.h 中。实际上，双向链表即是实验中维护 PCB 信息的主要数据结构。

2. 进程初始化

在 Part 1 中，进程初始化（task）流程见图 1。

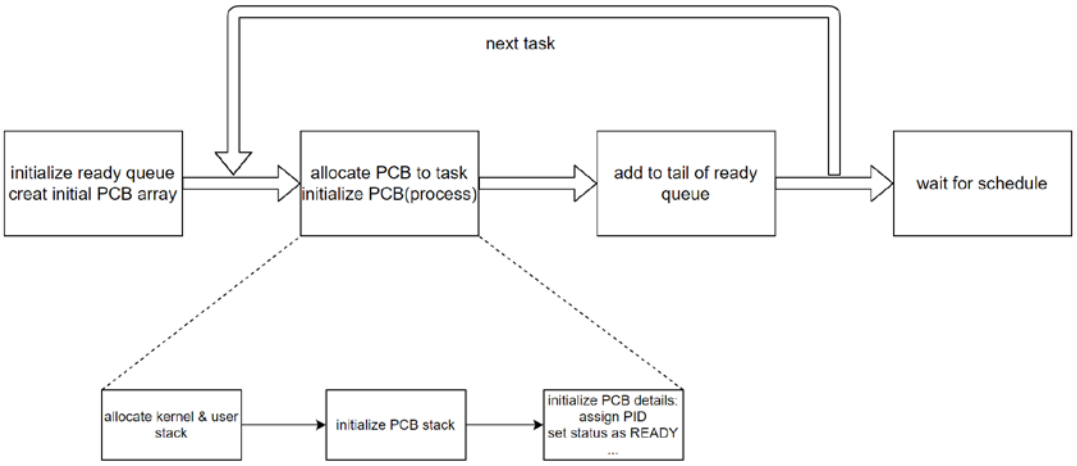

图 1 进程初始化（task）流程图

初始化等待队列以及 PCB 数组（链表）后，对于每一个 task：

（1）分配 PCB 给对应的 task

①分配内存栈以及用户栈空间

②调用 init_pcb_stack()初始化 PCB 栈（Part 1 中主要为初始化 switch_to 上下文，regs 上下文在 Part 1 中不会被使用）

③设置 pid，status，type 等 PCB 信息。

（2）将 PCB 压入等待队列队尾

（3）等待调度、运行

特别的，PID=0 的初始进程在 kernel/sched/sched.c 中初始化。

对于任务 1，三个 task 完成初始化后内存的布局如下：

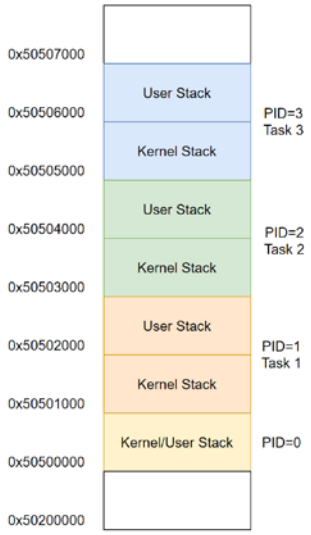

图 2 任务 1 Memory Layout

在任务 1 中，我仅使用了最简单的内存分配方法，后续实验中根据需要改进。

3. 非抢占式调度设计

Part 1 中，由于实现的时非抢占式调度，故进程切换（上下文切换）仅会在上一个进程放弃控制权时发生。

Part 1 中非抢占式调度通过 do_scheduler()实现，具体实现流程图如下：

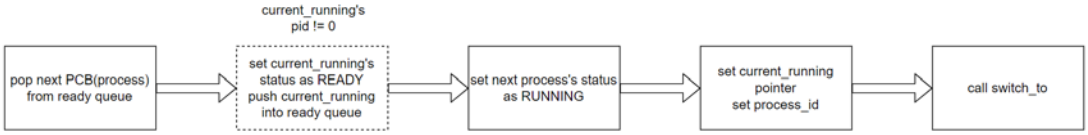

图 3 do_scheduler()流程图

（1）从等待队列中取出下一个下一进程的 PCB

（2）若当前进程的 PID!=0，将当前进程的状态设为 READY，并将其加入等待队列队尾。

（3）将（1）中取出的 PCB 状态设为 RUNNING

（4）根据（1）中的 PCB 修改 current_running 指针和 process_id

（5）调用 switch_to()

Switch_to()用于将 CPU 控制权从上一进程移交给下一进程（进程切换）。实际上，switch_to()需要完成的工作即为根据 ABI 约定保存上文、恢复上文、跳转至下一进程。具体汇编代码见图 4。

```
ENTRY(switch_to)
  // save all callee save registers on kernel stack
  addi sp, sp, -(SWITCH_TO_SIZE)
  /* TODO: store all callee save registers,
   * see the definition of `struct switchto_context` in sched.h*/
  sd sp,  SWITCH_TO_SP(sp)
  sd ra,  SWITCH_TO_RA(sp)
  sd s0,  SWITCH_TO_S0(sp)
  sd s1,  SWITCH_TO_S1(sp)
  sd s2,  SWITCH_TO_S2(sp)
  sd s3,  SWITCH_TO_S3(sp)
  sd s4,  SWITCH_TO_S4(sp)
  sd s5,  SWITCH_TO_S5(sp)
  sd s6,  SWITCH_TO_S6(sp)
  sd s7,  SWITCH_TO_S7(sp)
  sd s8,  SWITCH_TO_S8(sp)
  sd s9,  SWITCH_TO_S9(sp)
  sd s10, SWITCH_TO_S10(sp)
  sd s11, SWITCH_TO_S11(sp)
  sd sp,  (a0)
  // restore next
  /* TODO: restore all callee save registers,
   * see the definition of `struct switchto_context` in sched.h*/
  ld sp,  PCB_KERNEL_SP(a1)
  ld ra,  SWITCH_TO_RA(sp)
  ld sp,  SWITCH_TO_SP(sp)
  ld s0,  SWITCH_TO_S0(sp)
  ld s1,  SWITCH_TO_S1(sp)
  ld s2,  SWITCH_TO_S2(sp)
  ld s3,  SWITCH_TO_S3(sp)
  ld s4,  SWITCH_TO_S4(sp)
  ld s5,  SWITCH_TO_S5(sp)
  ld s6,  SWITCH_TO_S6(sp)
  ld s7,  SWITCH_TO_S7(sp)
  ld s8,  SWITCH_TO_S8(sp)
  ld s9,  SWITCH_TO_S9(sp)
  ld s10, SWITCH_TO_S10(sp)
  ld s11, SWITCH_TO_S11(sp)
  ld sp,  SWITCH_TO_SP(sp)

  mv tp, a1
  addi sp, sp, SWITCH_TO_SIZE
  jr ra
ENDPROC(switch_to)
```

图 4 switch_to()汇编代码

二、互斥锁的实现

本次实验中互斥锁相关机制如下：

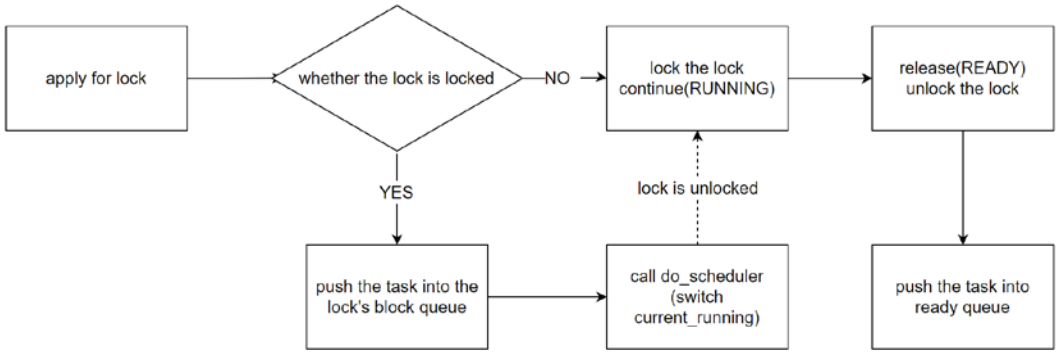

图 5 进程申请互斥锁流程图

对于每个进程：

（1）申请锁，根据锁是否被占用：

（2.1）Mutex Lock：UNLOCKED

a) 调用 do_block()，占用锁，继续运行

b) 调用 do_unblock()，释放控制权，解锁锁

c) 进程进入等待队列

（2.2）Mutex Lock：LOCKED

a) 进程进入锁的等待队列

b) 调用 do_scheduler()，避免资源浪费

c) 锁释放后同（2.1）a) ~ c)