Project2 A Simple Kernel 设计文档（Part I）

中国科学院大学
贾志杰
2020 年 10 月 16 日

1. 任务启动与 Context Switch 设计流程

（1）PCB 的设计

```
/* Process Control Block */
typedef struct pcb
{
    /* register context */
    uint64_t kernel_sp;
    uint64_t user_sp;
    uint64_t entry_point;

    /* previous, next pointer */
    struct pcb_t *prev;
    struct pcb_t *next;
    /* task in which queue */
    uint32_t inqueue;

    /* What tasks are blocked by me, the tasks in this
     * queue need to be unblocked when I do_exit(). */
    pid_t block_task[NUM_MAX_TASK];
    /* holding lock */
    uint32_t holdlock;

    /* priority */
    uint32_t priority;
    /* name */
    char name[32];
    /* process id */
    pid_t pid;
    /* kernel/user thread/process */
    task_type_t type;
    /* BLOCK | READY | RUNNING */
    task_status_t status;
    /* cursor position */
    int cursor_x;
    int cursor_y;
} pcb_t;
```

如上方代码所示，pcb 结构体中包括多项数据。首先第一部分是**内核栈、用户栈和进程入口地址**。栈地址用于确定程序运行使用的栈的位置，内核栈同时也负责保存上下文 register context，进程的入口地址则用于第一次运行进程时跳转到该位置。

第二部分是**调度相关**的数据结构。*prev，*next 是 pcb 类型的指针，在就绪/阻塞队列中用于构建链式结构。inqueue 标记进程位于哪个队列中。block_task[]是该进程所阻塞的进程的列表，目前没有使用。

第三部分是**锁相关**的数据结构。holdlock 是进程持有的锁的数量。为了方便管理和支持同时拥有多把锁，与锁相关的阻塞列表等信息没有设计为存放在 pcb 中，而是存放在锁结构中。

第四部分是**优先级相关**的数据结构。priority 是进程的优先级，目前没有使用。

第五部分是**进程的信息和状态**。name[]是进程的名字字符串。pid 是进程在 pcb 数组里的下标，是识别进程的唯一 ID。type 是进程的类型，标记该进程是内核还是用户进程。status 是进程的状态，标记进程正在运行、就绪或者阻塞等。

最后部分是**光标位置**，记录光标位置可以避免多进程运行时打印混乱。

（2）如何启动一个 task，包括如何获得 task 的入口地址，启动时需要设置哪些寄存器等

想要启动一个 task，需要为该 task 创建一个 pcb 并初始化 pcb 的信息，之后把该 pcb 加入就绪队列，等待内核调度让 task 运行。task 的入口地址是其主函数的起始地址，在 C 语言中可以通过函数名得到。

在启动一个 task 时，在非抢占式调度下，至少需要设定 sp，ra 寄存器。ra 提供程序入口地址，sp 为程序提供栈环境。在抢占式调度下，还需要设置 EPC、CAUSE、STATUS 寄存器。

（3）context switch 时保存了哪些寄存器，保存在内存什么位置，使得进程再切换回来后能正常运行

上下文切换时保存了 32 个全部通用寄存器，和 7 个协处理器，协处理器包括 CAUSE、STATUS、EPC 等。

上下文被保存在内核栈里，由 pcb 的 kernel_sp 指示内核栈的位置。

（4）设计、实现或调试过程中遇到的问题和得到的经验

在最开始设计初始化 pcb 时，对 ra 初始化了程序入口地址，但没有对 sp 初始化栈地址，导致 task 一启动就出错。对于 C 语言程序的运行，一定要先设置栈的位置，有了栈程序才能运行。

2. Mutex lock 设计流程

（1）无法获得锁时的处理流程

当进程尝试获取锁失败时，调用 do_block()函数将自己挂起。do_block()函数首先将该进程的状态改为 BLOCK 阻塞态，然后把该进程的 pid 添加到该锁的阻塞列表中，最后调用 do_scheduler()函数切换进程，让出运行权。

（2）被阻塞的 task 何时再次执行

锁的结构体中有一个阻塞列表，记录被该锁阻塞的进程。当持有锁的进程释放锁时，会调用 do_unblock_all()函数，将锁的阻塞列表中的所有进程改为就绪态，重新添加到就绪队

列中。之后由调度器调度再次执行。

（3）设计、实现或调试过程中遇到的问题和得到的经验

在最开始的设计中，锁的阻塞列表被设在持有锁的进程的 pcb 中，但是这样无法满足一个进程拥有多个锁，不同锁阻塞的进程会混在一起。之后改为将阻塞列表设在锁的结构体中，一个锁对应一个阻塞列表，释放一个锁只需要释放该锁的阻塞列表。同时这样的设计也能让一个进程持有多个锁。

3. 关键函数功能

（1）初始化 PCB 的函数

```
/* Process Control Block */
void set_pcb(pid_t pid, pcb_t *pcb, task_info_t *task_info)
{
    pcb[pid].pid = pid;
    pcb[pid].kernel_sp = new_kernel_stack();
    pcb[pid].user_sp = new_user_stack();
    pcb[pid].entry_point = task_info->entry_point;
    init_kernel_stack(pid, pcb);

    pcb[pid].type = task_info->type;
    pcb[pid].status = TASK_READY;
    pcb[pid].inqueue = READY;
    strcpy(pcb[pid].name, task_info->name);

    pcb[pid].priority = 0;
    pcb[pid].prev = NULL;
    pcb[pid].next = NULL;
    pcb[pid].holdlock = 0;
    queue_push(&ready_queue, &pcb[pid]);
}
```

（2）初始化 PCB 的内核栈的函数

```
void init_kernel_stack(pid_t pid, pcb_t *pcb)
{
    regs_context_t *pt_regs;
    pcb[pid].kernel_sp -= sizeof(regs_context_t);
    pt_regs = (regs_context_t *)pcb[pid].kernel_sp;
    memset(pt_regs, 0, sizeof(regs_context_t));

    uint64_t *reg = (uint64_t *)pt_regs;
    reg[29] = pcb[pid].user_sp;
    reg[31] = reg[37] = pcb[pid].entry_point;
    //reg[32] = initial_cp0_status;
```

```
    //reg[33] = initial_cp0_cause;
}
```

（3）调度函数

```
void scheduler(void)
{
    current_running->cursor_x = screen_cursor_x;
    current_running->cursor_y = screen_cursor_y;
    if(current_running->status == TASK_RUNNING){
        current_running->status = TASK_READY;
        queue_push(&ready_queue, current_running);
    }
    if(!queue_is_empty(&ready_queue)){
        current_running = (pcb_t *)queue_dequeue(&ready_queue);
        current_running->status = TASK_RUNNING;
        screen_cursor_x = current_running->cursor_x;
        screen_cursor_y = current_running->cursor_y;
    }
}
```

（4）do_scheduler 汇编函数

```
# function do_scheduler
NESTED(do_scheduler, 0, ra)
    SWITCH_STACK
    SAVE_CONTEXT
    jal scheduler
    RESTORE_CONTEXT
    jr ra
END(do_scheduler)
```

（5）申请锁、释放锁

```
void do_mutex_lock_acquire(mutex_lock_t *lock)
{
    while(lock->status == LOCKED){
        do_block(&block_queue, lock);
    }
    lock->status = LOCKED;
    lock->holder = current_running->pid;
    (current_running->holdlock)++;
}

void do_mutex_lock_release(mutex_lock_t *lock)
{
    do_unblock_all(&block_queue, lock);
```

```
    lock->status = UNLOCKED;
    lock->holder = 0;
    (current_running->holdlock)--;
}
```

（6）do_block 函数

```
void do_block(queue_t *queue, mutex_lock_t *lock)
{
    current_running->status = TASK_BLOCKED;
    current_running->inqueue = BLOCK;
    lock->block_task[++(lock->block_task_count)] = current_running->pid;
    do_scheduler();
}
```

（7）do_unblock 函数

```
void do_unblock_one(queue_t *queue, pcb_t *unblock_one)
{
    unblock_one->status = TASK_READY;
    unblock_one->inqueue = READY;
    queue_push(&ready_queue, unblock_one);
}

void do_unblock_all(queue_t *queue, mutex_lock_t *lock)
{
    pcb_t *unblock_one;
    while(lock->block_task_count){
        unblock_one = &pcb[lock->block_task[lock->block_task_count]];
        do_unblock_one(queue, unblock_one);
        (lock->block_task_count)--;
    }
}
```

参考文献

[1] MIPS® Architecture For Programmers Volume II-A: The MIPS64® Instruction Set Reference Manual, Revision 6.05, 2016

[2] 现代操作系统(第 4 版)[M]. 北京: 机械工业出版社, 2017: 74-75, 90-91