Project 1 A Simple Kernel 设计文档（Part I）

中国科学院大学 徐泽凡 2018K8009929037

任务启动与上下文切换设计流程

(1) 进程控制块 (PCB)

PCB 实现于 `include/os/sched.h` 中的 `pcb` 结构体，包括以下内容：

- `pid`：进程号；
- `name`：进程名，用于调试；
- `type`：进程类型，标记用户态/内核态、进程/线程；
- `status`：进程状态，包括就绪、运行、阻塞等；
- `priority`：进程优先级；
- `in_queue`：指向 PCB 所属队列的指针；
- `prev`、`next`：用于 PCB 队列管理的前驱和后继指针；
- `kernel_context`：内核态上下文，存储现场寄存器值；
- `user_context`：用户态上下文，存储现场寄存器值。

事实上，在 Part I 中，我们并没有使用 `user_context`、`priority` 这两项内容，`kernel_context` 也仅包含主处理器的寄存器。在此后的实验中，PCB 的内容还可能包含更多的内容。

(2) 启动一个 Task

要启动一个 task，需要提供任务名、入口地址、任务类型三项信息，包含在文件 `include/os/sched.h` 内定义的 `task_info` 结构体内。入口地址可通过获取 task 函数的地址获得。

调用定义于 `set_pcb` 函数初始化 PCB，其完成了这些工作：

- 设置基础信息（进程号、类型、状态、优先级、进程名等）；
- 初始化 `prev`、`next` 两个队列指针，将 PCB 放入 `ready_queue`；
- 清空寄存器上下文，而后设置 `$sp` 和 `$ra` 两个寄存器：
  - 将 `$sp` 设置为进程的栈空间地址；
  - 将 `$ra` 设置为进程的入口地址；
- 初始化光标位置。

之后再通过调用调度程序 `do_scheduler`，开始执行就绪队列的任务。

(3) 上下文切换 (Context Switch)

上下文切换由 `arch/mips/entry.S` 中定义的 `do_scheduler` 函数完成，它依次完成了下面的工作：

- 保存现场至 `current_running`；
- 调用调度程序 `scheduler`；
- 恢复现场 `current_running`；
- 恢复执行。

上下文切换时保存了 32 个通用寄存器中的 29 个，没有保存的三个分别是 0 号寄存器 `$zero`、26 号寄存器 `$k0`、27 号寄存器 `$k1`。在 Part I 中，暂时没有保存 CP0 的几个寄存器。

在 MIPS 架构的实验中，寄存器现场是保存在 PCB 中的。具体来说，保存在 `pcb` 中的 `kernel_context` 结构体之中。在 Part I 的任务中，由于是非抢占式调度，程序在调用 `do_scheduler` 交出运行权时，会自动将返回

地址保存到 $ra 寄存器中。因而，可以在进程切换后通过 jr $ra 指令正常运行。

(4) 设计、实现、调试过程中的问题和经验

- 总之要先迭代出一个版本，然后再考虑修改优化；
- 在脑子清醒的时候进行设计、编码；
- gdb 是好文明。

互斥锁设计流程

互斥锁（Mutex Lock）通过阻塞和解除阻塞来实现线程锁的调度。

(1) 无法获得锁时的处理流程

当一个任务请求一个被占用的锁时，将会被阻塞。具体阻塞过程如下：

- 将任务的状态设为阻塞 TASK_BLOCK；
- 将任务放入这个锁的阻塞队列；
- 调用调度程序，交出执行权限。

(2) 被阻塞的 task 何时再次执行

当一个任务释放一个锁时，执行以下动作：

- 若该锁的阻塞队列为空，则将锁的状态修改为 UNLOCKED；
- 若该锁的阻塞队列非空，则解除队列第一个任务的阻塞：
  - 将队列中第一个任务出队；
  - 将其状态修改为就绪 TASK_READY，并将之加入就绪队列 ready_queue。

关键函数功能

set_pcb 函数

set_pcb 函数用于初始化 PCB。该函数没有返回值，包含以下参数：

- pid - 指定的进程号；
- pcb - 指向指定 PCB 的指针；
- task_info - 指向进程信息结构体的指针；

其定义如下：

```
void set_pcb(pid_t pid, pcb_t *pcb, task_info_t *task_info)
{
    // basic info
    pcb->pid = pid;
    pcb->type = task_info->type;
    pcb->status = TASK_READY;
    pcb->priority = 0;
    memcpy(pcb->name, task_info->name, TASK_NAME_LEN);

    // initialize queue
    pcb->prev = NULL;
    pcb->next = NULL;
    queue_push(&ready_queue, pcb);
    pcb->in_queue = &ready_queue;

    // initialize context
    memset(&pcb->kernel_context, 0, sizeof(regs_context_t));
```

```
    // initialize registers
    // ! This part is strong related with architecture
    // $sp(stack pointer)
    pcb->kernel_context.regs[29] = new_kernel_stack();
    // $ra(return addreee)
    pcb->kernel_context.regs[31] = task_info->entry_point;

    // initialize cursor
    pcb->cursor_x = 0;
    pcb->cursor_y = 0;
}
```

其工作流程已经在前面的章节详细说明。此函数直接操作了上下文中部分寄存器的值，与体系结构强相关。我希望在后面对它进行修改，将与体系结构相关的部分代码剥离到 `arch/mips` 目录下。

do_mutex_lock_acquire 函数与 do_mutex_lock_release 函数

`do_mutex_lock_acquire` 函数负责获取一个互斥锁，`do_mutex_lock_release` 函数负责释放一个互斥锁。其均有一个参数 `lock`，为锁的指针。它们的代码如下：

```
// Acquire a mutex lock
void do_mutex_lock_acquire(mutex_lock_t *lock)
{
    if (lock->status == UNLOCKED)
    {
        lock->status = LOCKED;
    }
    else
    {
        do_block(&lock->block_queue);
    }
}

// Release a mutex lock
void do_mutex_lock_release(mutex_lock_t *lock)
{
    if (queue_is_empty(&lock->block_queue))
    {
        lock->status = UNLOCKED;
    }
    else
    {
        do_unblock_one(&lock->block_queue);
    }
}
```

这一部分的工作流程已经在前面的章节详细说明。

目前的 `do_mutex_lock_acquire` 函数中，并没有为进程记录它获取了那些锁。这会导致，当进程被杀死时锁无法释放的问题。在后面的实验中，我希望能够添加这部分功能。

致谢

感谢在 Design Review 时与我交流的老师，与他的交流让我受益匪浅。

感谢在实验过程中与我交流的各位同学。