Project2 A Simple Kernel 设计文档

中国科学院大学
段江飞
2018 年 10 月 31 日星期三

1. 任务启动与 Context Switching 设计流程

(1) PCB 包含的信息

```
typedef struct pcb
{
    /* register context */
    regs_context_t kernel_context; //内核上下文和用户上下文
    regs_context_t user_context; //上下文包括 32 个通用寄存器和部分 CP0 寄存器
                                 //status、cause、epc、badvaddr 和 hi、lo 寄存器
    uint32_t kernel_stack_top;     //内核栈帧基址
    uint32_t user_stack_top;       //用户栈帧基址

    /* sleep time*/
    uint32_t begin_time;   //睡眠开始时间
    uint32_t sleep_time;   //要进行的睡眠时间

    /* cursor position */
    int cursor_x;          //打印光标的位置信息
    int cursor_y;

    /* previous, next pointer */
    void *prev;            //在就绪队列、阻塞队列里的链表链接信息
    void *next;

    /* process id */
    pid_t pid;             //进程号

    /* kernel/user thread/process */
    task_type_t type;      //任务类型，是内核还是用户，是线程还是进程

    /* BLOCK | READY | RUNNING */
    task_status_t status;  //任务状态，阻塞、就绪或者正在执行

    /* Priority */
    int priority;          //任务的权衡优先级，随着等待时间变化而增加
    int task_priority;     //任务的固定优先级
} pcb_t;
```

（2）如何启动第一个 task，例如如何获得 task 的入口地址，启动时需要设置哪些寄存器等

非抢占是调度时，第一个 task 启动是在 main 里调用 do_scheduler 函数，进行任务调度，先保存现场，然后修改 current_running 指针，指向第一个任务，再恢复现场，将 31 号寄存器恢复为函数入口地址，利用 jr ra 返回，启动第一个 task。任务的入口地址就是任务函数名的地址，在 task 的信息里，启动时需要将寄存器全部清零，29 号 sp 寄存器初始化为分配的栈帧基址，31 号寄存器初始化为函数入口地址，这些都是在初始化 pcb 的时候初始化进 pcb 表里，在恢复上下文的时候直接恢复。

抢占式调度时，第一个 task 启动是在 main 里进入时钟中断之后，在处理时钟中断的 irq_timer 里调用 do_scheduler 进行任务调度，这时候因为是处于时钟时钟中断，在任务调度之后要继续时钟中断的执行，完成整个时钟中断的流程，因此需要伪造一个任务的第一次执行是从时钟中断处理结束的部分开始的，因此需要将内核和用户的 29 号 sp 寄存器初始化为栈帧基址，内核的 31 号寄存器初始化为时钟中断的结束部分地址，用户的 31 号寄存器、内核和用户的 cp0_epc 寄存器初始化为函数入口地址，在时钟中断结束 eret 返回函数入口执行，同时，因为处于中断之中，内核和用户的 cp0_status 寄存器初始化为 0x10008003，其他的寄存器初始化为 0。

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

上下文转换需要保存和恢复除了 k0 和 k1 的 30 个通用寄存器，以及 cp0_status、hi、lo、cp0_badvaddr、cp0_cause 和 cp0_cause 寄存器。

（4）设计、实现或调试过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

初始化的时候，将 cp0_status 寄存器初始化为 0x10008000，导致进入时钟中断之后，一直陷入中断，无法退出，因为 EXL 为置为了 0，在 eret 的时候，出现了问题，一直陷入中断。

2. 时钟中断、系统调用与 blocking sleep 设计流程

（1）时钟中断处理的流程，请说明你认为的关键步骤即可

首先关中断，然后保存用户上下文，然后进行第一级例外处理，跳到中断处理函数，然后根据中断状态位进入时钟中断处理函数，进行任务调度（保存和恢复内核上下文），然后恢复用户上下文，开中断，eret 退出时钟中断。

（2）你所实现的时钟中断的处理流程中，如何处理 blocking sleep 的任务，你如何决定何时唤醒 sleep 的任务？

阻塞睡眠的任务在其睡眠队列里，每次进入时钟中断说明已经经过了一个时间片，这时

候 time_elapsed 会增加一定的时钟周期数，用来记录已经经过了多少个时钟周期，在时钟中断里会进行任务调度，每次任务调度都会检查睡眠队列，利用 get_timer 计算已经经过的时间，减去睡眠开始时间，和睡眠的长度比较，将已经达到睡眠时间的任务从睡眠队列移除，加入都就绪队列。

（3）你实现的时钟中断处理流程和系统调用处理流程有什么相同步骤，有什么不同步骤？

相同的步骤：硬件跳到相同的例外处理入口，然后关中断、保存用户上下文，根据例外码分级处理，之后会恢复用户上下文和开中断，eret 返回。

不同的步骤：时钟中断会进行任务调度，保存打印光标位置，重置 compare 和 count 寄存器。系统调用会将 epc 加 4，调用其他的内核函数完成操作。

（4）设计、实现或调试过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

时钟中断的处理流程没有搞清楚，最初是按照只保存恢复一次上下文，返回的时候通过修改 epc 实现任务调度，但是时钟中断之后发现系统调用涉及到内核和用户态的转变，难以处理，而且修改 epc 寄存器还出现了玄学 bug，就和学长交流之后换了思路。

系统调用之后，epc 需要手动加 4，这个问题卡了挺久，一直陷入系统调用出不来。

3. 基于优先级的调度器设计

（1）priority-based scheduler 的设计思路，包括在你实现的调度策略中优先级是怎么定义的，何时给 task 赋予优先级，测试结果如何体现优先级的差别

优先级定义在 pcb 的结构体中，由两部分组成，，第一部分是固定的任务优先级，是最初初始化任务确定的，另一部分是总的优先级，这是和等待时间权衡的，在每次进行任务调度的时候，等待队列里的任务优先级会加 1，入队的时候是按照优先级队列入队，如果优先级一样，则是按照等待时间，插入到同优先级的最后一个任务的后面，这样能够保证低优先级的任务不会饿死。测试时，优先级的差别通过运行次数体现，优先级高的运行次数比较多，优先级低的运行次数少，甚至可能很久才会运行一次。

（2）设计、实现或调试过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

优先级调度最初时考虑按照等待时间去看，如果一个任务等待时间超过多少就直接运行它，或者给他修改优先级，但这样的话，按照前者，可能形成最后比较多的任务等待时间过长，造成直接按照等待时间去执行，而不是优先级，按照后者，每次修改优先级需要移除和入队，很麻烦，效率感觉不高。后来从姚依航同学得知可以每次在任务调度里面对就绪队列的优先级加 1，这样整体队列还是按照优先级排序，而且低优先级会获得执行机会，本质上，

可以看做是一种对不同优先级赋予不同的时间片。

4. Mutex lock 设计流程

（1）spin-lock 和 mutual lock 的区别，能获取到锁和获取不到锁时各自的处理流程

自旋锁的实现是设置一个变量，当一个线程要进入临界区的时候，首先检查这个变量，如果这个变量状态为临界区，并且此时没有其他冲突线程在访问这个变量，那么该线程就进入临界区，并且将这个变量置为有线程访问状态。如果这个变量状态为有其他冲突线程在访问，那么就不断的使用while 循环重试，直到可以进入临界区。自旋锁每次没获得锁，会进入忙等待，占用CPU时间，比较浪费资源，单线程处理器上如果不直接打断，就会一直等待下去。互斥锁的实现方法为一旦线程请求锁失败，那么该线程会自动被挂起到该锁的阻塞队列中，不会被调度器进行调度。直到占用该锁的线程释放锁之后，被阻塞的线程会被占用锁的线程主动的从阻塞队列中重新放到就绪队列，并获得锁。互斥锁可以节约CPU资源。

（2）被阻塞的 task 何时再次执行

互斥锁释放之后，会查询相应的阻塞队列，若不为空，则将第一个任务出队，设为拿到锁，修改任务为就绪状态，然后进入就绪队列，等待下次调度到的时候执行。

5. Bonus 设计思路

（1）如何处理一个进程获取多把锁

对每一个锁设置一个锁的 id 域，对应的 id 会有一个阻塞队列，试图获取这把锁的任务在阻塞之后会进入相应的阻塞队列，unblock 的时候也是从对应的阻塞队列 unblock。一个进程获取多把锁，如果获得了，就正常执行，如果没有获得，将任务加入相应的阻塞队列，由于任务获取锁的一次获取的，不可能同一时刻请求获得多把锁，因此任务获取锁失败后被加入的阻塞队列也只有一个。

（2）如何处理多个进程获取一把锁

多个进程获取一把锁就是只有一个进程能够获得，其他的进程都会进入对应的等待队列，并且按照优先级排序。等锁释放之后，最高优先级的任务 unblock，获得锁。同时，为了避免低优先级的任务无法获得到锁，一直阻塞的状态，在每次 unblock 的时候，会对相应的阻塞队列的任务的优先级全部加 1，实现优先级的改变，避免低优先级任务阻塞死。

（3）你的测试用例和结果介绍

测试用例有三个任务，第一个任务先去获取锁 1，再去获取锁 2，然后释放锁 2，释放锁 1。第二个任务先去获取锁 1，然后释放锁 1，然后去获取锁 2，释放锁 2。第三个任务获取锁 1，然后释放锁 1。在获取每个锁之后，都会进入一个 for 循环执行一段时间。这样实

现了一个进程获取多把锁，和多个进程获取一把锁。

结果也符合预期，在第一个任务获得锁 1,之后，另外两个任务都进入阻塞状态，然后任务 1 依次执行获得锁 1 和锁 2，释放锁 1 之后，阻塞的任务优先级高的那个获得锁 1，继续执行。

6. 关键函数功能

请列出你觉得重要的代码片段、函数或模块（可以是开发的重要功能，也可以是调试时遇到问题的片段/函数/模块）

1. do_scheduler 函数和保存恢复上下文的宏
2. 开关中断宏
3. 初始化 pcb
4. Scheduler 函数
5. do_block 和 do_unblock 的函数

具体代码见文件