Project2 A Simple Kernel设计文档（Part II）

中国科学院大学

蔡润泽

2019年10月13日

# 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
2. 在初始化操作完成后，当cp0\_compare==cp0\_status时，硬件自动跳转至0x80000180处，此处有之前在初始化阶段搬运过来的exception\_handler\_entry。exception\_handler\_entry负责保存用户态内容，并判断例外的类型，此处例外类型为中断，因此系统跳转至中断处理程序handle\_int。
3. 中断处理程序handle\_int负责取出CP0\_STATUS和CP0\_CAUSE的值，并借助interrupt\_helper程序判断中断类型，此处终端类型为时钟中断。因此，系统跳转至时钟中断处理程序。
4. 时钟中断处理程序负责累加运行时间，并利用do\_scheduler程序，完成上下文切换。
5. 上下文切换完成后，进入新进程的用户态，并恢复用户态内容。
6. 你所实现的时钟中断的处理流程中，何时唤醒sleep的任务？

在进行上下文切换时，scheduler函数首先会利用check\_sleep函数来检查阻塞队列中是否有sleep时间已达到的进程。利用get\_timer函数，该操作系统可以获取当前时间，将当前时间与进程起始睡眠时间相减，若大于等于睡眠时间，则该进程被唤醒，进入到就绪队列。

1. 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？

相同步骤：硬件自动跳转至0x80000180处，此处有之前在初始化阶段搬运过来的exception\_handler\_entry。exception\_handler\_entry负责保存用户态内容，并判断例外的类型。当系统调用处理完毕后，恢复用户态内容。

不同步骤：

1. 触发系统调用的函数是利用syscall指令。
2. 系统调用处理函数是利用syscall表来完成的。
3. 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）

这部分实验主要遇到了两个问题：

1. cp0\_status初始化的问题。在运行task3时，我的cp0\_status寄存器初始化只置为了0x10008001而非0x10008003。这个设置在运行task3的时候还可以进行，但实验运行到task4时，就出现了问题。根据老师上课的提示，我将其初值置为了0x10008003，系统停止运行的bug消失。
2. 打印输出的问题。我发现本次实验中，printf的输出会重叠在之前的输出内容上，导致屏幕出现残影。我通过设置空格和更改print\_location的值解决了这一问题。

# 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，测试时给不同任务赋的优先级是多少，结果如何体现优先级的差别

在优先级设置时，该设计综合考虑了两个因素：任务本身的优先级和考虑等待时间的优先级。进程最开始会按照相应任务本身的优先级进入就绪队列。当一个任务被执行时，就绪队列中其他任务的优先级会增加1，同时当前进程的优先级被还原成任务优先级。同时，对于被阻塞的队列，也会根据等待时间的长短，依次增加其优先级的值。

在任务优先级方面，不同任务的任务优先级如下：（结构中第三个元素为优先级）

1. // test\_lock2.c : User space lock test
2. **struct** task\_info task2\_4 = {(uint32\_t)&lock\_task1, KERNEL\_THREAD,10};
3. **struct** task\_info task2\_5 = {(uint32\_t)&lock\_task2, KERNEL\_THREAD,10};
5. /\* [TASK4] task group to test interrupt \*/
6. // When the task is running, please implement the following system call :
7. // (1) sys\_sleep()
8. // (2) sys\_move\_cursor()
9. // (3) sys\_write()
10. **struct** task\_info task2\_6 = {(uint32\_t)&sleep\_task, USER\_PROCESS,5};
11. **struct** task\_info task2\_7 = {(uint32\_t)&timer\_task, USER\_PROCESS,5};
13. **struct** task\_info task2\_8 = {(uint32\_t)&printf\_task1, USER\_PROCESS,40};
14. **struct** task\_info task2\_9 = {(uint32\_t)&printf\_task2, USER\_PROCESS,40};
15. **struct** task\_info task2\_10 = {(uint32\_t)&drawing\_task2, USER\_PROCESS,40};

# Bonus设计思路（做bonus的同学需要写该节）

1. Bonus设计
   1. 将block\_queue队列由1个更改为NUM\_MAX\_TASK个，并将lock的数据结构增加id值。
   2. 在锁初始化时，将lock->id依次增加，锁的申请和释放的阻塞队列编号，对应lock->id的值
2. 测试用例

本次实验中，我增加了一个lock\_test3文件，利用三个任务，抢占两个锁。任务一先抢占锁1，后抢占锁2；任务而先抢占锁2，后抢占锁1。任务三只抢占锁1。但由于进行本次实验的时间较紧，lock\_id的初始化值出现了不为0的现象，需要在后续更改createimage来修复这一问题。

# 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）

重要函数片段：

本次PCB初始化中，对于ra寄存器等设置：

1. pcb[queue\_id].kernel\_context.regs[31]= (uint32\_t)reset\_cp0;
2. pcb[queue\_id].kernel\_context.cp0\_epc = lock\_tasks[i]->entry\_point;
3. pcb[queue\_id].kernel\_context.cp0\_status=0x10008003;

prioity\_queue\_push（）的根据优先级将任务置入就绪队列的操作。

参考文献

1. project\_2\_partII\_guide\_book