Project3 Preemptive Kernel设计文档

中国科学院大学

袁峥

2017.11.14

# 时钟中断与blocking sleep设计流程

（1）中断处理的一般流程

1、首先通过异常处理入口关闭中断，然后通过exc\_code位判断出是中断，进入中断处理程序

2、保存用户态寄存器

3、判断中断类型，进入相应的中断处理程序

4、恢复用户态寄存器

5、开中断，中断返回

（2）你所实现的时钟中断的处理流程，如何处理blocking sleep的tasks；如何处理用户态task和内核态task

时钟中断处理流程：

1、清时钟中断

2、time\_elapsed加一

3、判断是在用户态还是内核态，若在内核态直接返回

4、关闭中断

5、进入内核态

6、将当前任务放入ready\_queue，并调用scheduler\_entry进行任务切换

7、返回用户态

8、开中断并返回

如何处理blocking sleep的tasks

对于调用sleep的task，在处理时先设置一个deadline（即唤醒时间），然后进入休眠队列，等待到达唤醒时间后被唤醒。具体唤醒的任务是由scheduler函数完成，每次进入scheduler会进行检查是否有休眠任务到达唤醒时间，如果到达那么从休眠队列中取出放入就绪队列。

如何处理用户态task和内核态task

对于用户态task和内核态task，其时钟中断使能信号都打开，即如果发生时钟中断都会进入时钟中断处理程序。

进入处理程序后，首先保存用户态上下文，并将time\_elapsed加一，然后判断是否处于内核态，如果是则直接结束时钟中断处理，恢复上下文并返回。如果当前处于用户态，则进入内核态然后将任务放入ready\_queue，并调用scheduler\_entry切换至新任务，等下次回到该任务时，返回用户态并恢复上下文，结束时钟中断处理。

（3）blocking sleep的含义，task调用blocking sleep时做什么处理？什么时候唤醒sleep的task？

blocking sleep的含义

一个任务主动执行睡眠，将一个任务的状态由运行转换到睡眠，即暂时不调用该任务，等到睡眠时间到了之后再唤醒。

task调用blocking sleep时做什么处理

1. 关闭中断
2. 设置任务的deadline，即唤醒时间
3. 将任务状态由运行改为睡眠
4. 将该任务放入sleep\_wait\_queue队列
5. 进行新的任务调度
6. 打开中断

什么时候唤醒sleep的task

在进行新的任务调度时，先进行check\_sleeping，即检查每个睡眠任务的睡眠时间是否已经结束，如果结束则将其从sleep\_wait\_queue中取出放到ready\_queue，并将状态由修改改为就绪。如果当前没有就绪任务，则不断进行check\_sleeping，直到有任务的睡眠时间结束。

（4）设计或实现过程中遇到的问题和得到的经验

在初始的代码中，enter\_critical和leave\_critical不配对的现象很严重，而且是在一个函数中enter\_critical，在另一个函数中leave\_critical，因此调试时经常出现panic。后来重新通读了一遍代码，并修改了其中的一些开关中断的内容，使其配对。

# 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，如何给task赋予优先级，调度与测试用例如何体现优先级的差别

priority-based scheduler的设计思路是首先在初始化的时候给每个task赋一个权重，在每次在调度的时候，遍历就绪队列中的所有任务，挑选出一个当前权重最高的任务，并将其权重减一，如果权重被减至0，那么将其恢复初始权重。

在调度策略中，优先级即对应初始权重的大小，优先级越高，其初始权重越大。在程序中初始化时，优先级的设置与每个任务的PID相同，所以可以看作越后面的任务优先级越高。

在实际上板测试中，所体现出的结果就是在print\_status函数输出的结果中，PID越大的任务Entries越大，且是成正比的关系，可以认为结果符合设计情况。

（2）设计或实现过程中遇到的问题和得到的经验

一开始设计的优先级调度算法比上述的要复杂一些，大概思路是根据每个任务的Entries和priority的比值进行比较，然后选出最小的。该设计从理论上来说应该也是可行的，但是由于在进行除法时取整后会有误差，因此实际结果并没有完成呈现成正比的关系。因此后来稍微更改了实现方法。

# 关键函数功能

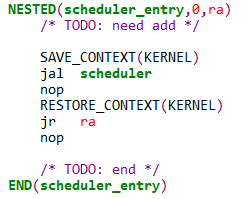
请列出上述各项功能设计里，你觉得关键的函数或代码块，及其作用

1、handle\_int(entry.S)



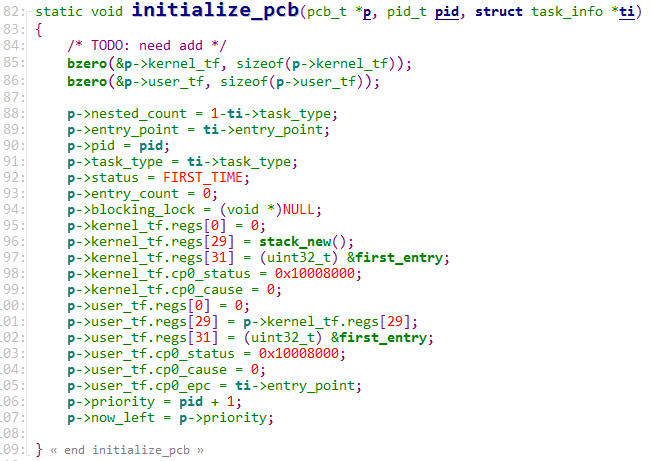
该函数是中断处理程序，首先保存上下文。接着判断中断原因，如果非时钟中断，那么清中断并恢复上下文，退出中断处理程序。如果是时钟中断，那么进入timer\_irq处理时钟中断。在时钟中断处理中，首先将time\_elapsed加1，然后判断当前在什么态，如果是在内核态，那么直接恢复上下文并退出中断处理程序。否则进入内核态，调用scheduler切换任务（保存和恢复内核态上下文在scheduler中完成），再次调度至该任务时，返回用户态并恢复用户态上下文，然后退出时钟中断处理。

2、scheduler\_entry(entry.S)



该函数作用为任务切换调度，首先保存内核态上下文，然后调用scheduler函数获取新的就绪任务，然后恢复新任务的上下文并跳转至上次执行的指令继续。

3、initialize\_pcb(kernel.c)



该函数是进行每个task的PCB初始化，分别初始化内核态和用户态的部分寄存器的值。其中比较重要的是CP0\_STATUS，要将时钟中断打开，其他中断可以关闭。regs[31]中放置的first\_entry函数的地址，该函数会复用中断返回退出时的代码，具体是开中断，然后跳转到EPC的地址，因此将每个task的入口地址放在CP0\_EPC可以完成初始化功能。

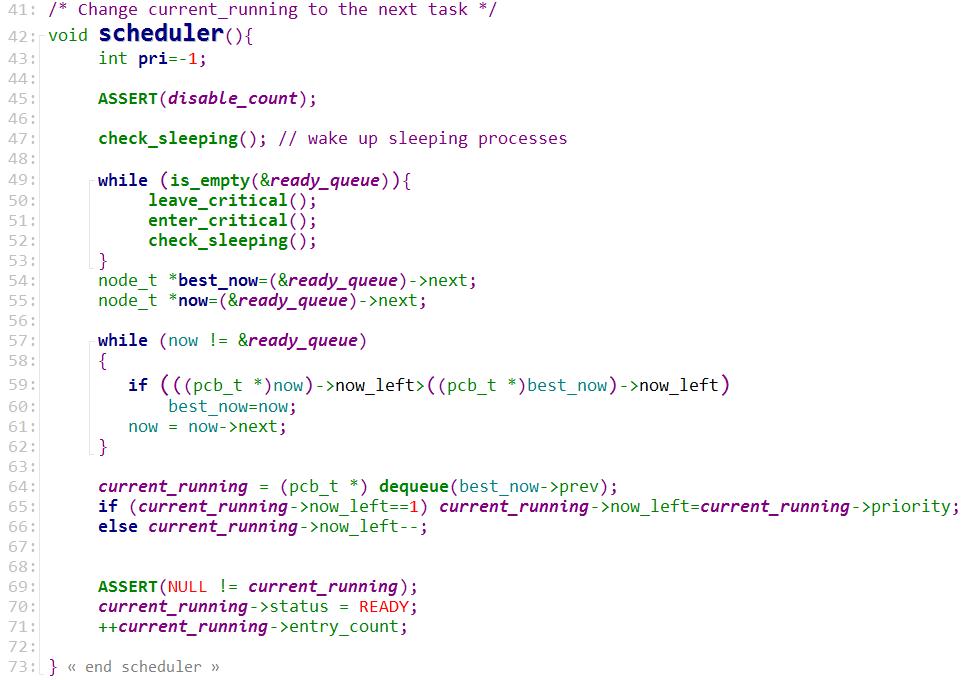
4、SAVE\_CONTEXT(entry.S)



该函数是保存上下文，主要保存31个寄存器和CP0\_STATUS、CP0\_HI、CP0\_LO、CP0\_BADVADDR、CP0\_CAUSE、CP0\_EPC等CP0协处理器寄存器。由于该函数其实是宏定义，没有进行函数调用，因此此时的SP和RA寄存器的值就是实际应该保存的值，不需要再像实验2中去栈中寻找。offset所对应的是USER或KERNEL，其为USER和KERNEL对应的trapframe\_t结构在PCB中的偏移，这样可以使保存内核态和用户态上下文共用一段程序。

对应的，RESTORE\_CONTEXT与SAVE\_CONTEXT几乎对称，作用是将保存的上下文恢复至寄存器中，在此不再重复罗列。

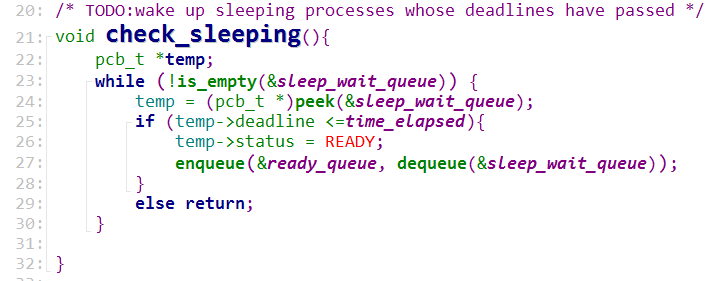
5、scheduler(scheduler.c)



该函数为新任务的调度函数，其中包含了优先级的相关实现。首先调用check\_sleeping检查是否有休眠任务到达唤醒时间，如果有就加入就绪队列。如果就绪队列为空就一直空跑，等待有休眠任务被唤醒。

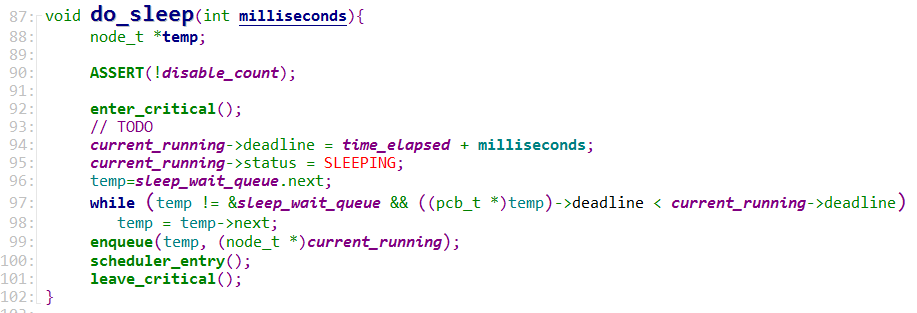
然后遍历整个就绪队列，选择当前权重最高的任务，并将该任务的权重减一，如果为0则恢复默认优先级。

6、check\_sleeping(scheduler.c)



该函数遍历休眠队列，检查任务是否到达唤醒时间，如果到达就从休眠队列中取出至就绪队列，由于在将任务放入休眠队列时已经保证唤醒时间从早到晚排列，因此只需从前往后找到第一个还没到唤醒时间的任务就可以退出。

7、do\_sleep(scheduler.c)



该函数处理任务的休眠请求，由于要修改current\_running，因此首先关中断，然后设置当前任务的唤醒时间，并将其改为休眠态，然后按照唤醒时间从早到晚，将任务放入休眠队列。再进行任务调度执行新的任务。