Project2 Non-Preemptive Kernel设计文档

中国科学院大学

袁峥

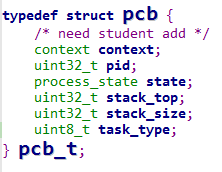
[学号] 2015K8009929008

2017.10.13

# Context Switching设计流程

（1）PCB包含的信息

PCB每个表项中包含进线程切换时31个寄存器的值，以及该进程的pid，该进程的状态（阻塞、就绪、运行、退出）、该进程所分配的栈顶地址、该进程所分配的栈大小和该任务类型（进程还是线程）。特别指出，进程切换时，原进程的运行下一条指令地址被直接保存在31号寄存器，在切换时直接使用寄存器跳转即可。



（2）如何启动第一个task

首先将进行初始化，建好一个就绪进程队列和阻塞进程队列，并初始化每个task的PCB表项，然后逐个压入就绪进程队列。初始化完毕后，启动第一个task任务时，只需调用scheduler\_entry函数，取出就绪进程队列中的一个task，然后装载寄存器并切换到这个task即可。

（3）scheduler的调用和执行流程

scheduler函数是被entry\_mips.S中的scheduler\_entry函数所调用的，scheduler函数首先弹出一个就绪进程队列中的task并赋值到current\_running，并将该task的状态由就绪该为运行。接着scheduler函数结束并回到scheduler\_entry函数，该函数将current\_running所指PCB表项中的保存的寄存器的值全部更新到寄存器中，然后使用jr ra指令跳转到该task上次切换退出时所运行到的指令。

（4）context switching时如何保存PCB，使得进程再切换回来后能正常运行

保存PCB的过程在entry\_mips.S中的save\_pcb函数，该函数将task切换退出时31个寄存器的值保存到该task所在的PCB表项中，以便下次切换回来后能正常运行。其中有两个寄存器的值需要特别说明，由于进行保存PCB时会要调用save\_pcb函数，函数调用时本身就会压栈并更改31号寄存器的值，因此在保存原task切换时寄存器的值时，sp和ra寄存器的值不能直接保存。通过查看反汇编的代码，发现在运行save\_pcb函数前sp指针先被减去了24，因此在保存真正task切换时的sp时，需要将save\_pcb函数中的sp加上24，而task切换时真正需要保存的ra寄存器的值在进入save\_pcb已经被压入栈中，通过反汇编可以看到偏移是20(sp)，因此需要将20(sp)中的数据存入PCB中保存的ra寄存器。

（5）任何在设计、开发和调试bootblock时遇到的问题和解决方法

该部分遇到的问题首先是初始化两个队列时，没有仔细查看队列定义中的元素，所以没有初始化capacity项，导致在后面队列操作时出现模0，从而出现错误。

其次是在save\_pcb函数中保存真正的ra寄存器的值时，从老师课件中看到的是16(sp)，编写代码的时候就先这样写了，后来上板运行时发生错误，发现PC跳转到了0x800XXXXXX，而这个PC地址在这次的实验中是不可能出现的，因此我猜测可能是保存ra时出现了错误，然后检查了反汇编代码，发现在自己的程序中被压入栈的ra值的偏移是20(sp)，修改后最终解决了这个错误。

# Context Switching开销测量设计流程

（1）如何测量线程切换到线程时的开销

该任务中task的执行顺序是thread4、thread5、process3三者不断循环，因此测量线程间切换的开销，只需在切换离开thread4前进行计时，然后切换进入thread5时，再次计时，并将两者作差就是线程间切换的开销。

（2）如何测量线程切换到进程时的开销

考虑到thread4和thread5是在一个文件中，在文件中使用全局变量较为方便，因此在process3中没有做任何事，一进去就切换离开，而测量线程和进程间切换的开销时，在切换离开thread5前进行计时，然后在切换进入thread4时再次计时，两者之间的时间包含从thread5切换到process3，再从process3切换到thread4，也就是两次进程与线程间的切换，因此将该时间除2就是所要求的线程与进程间切换的开销。

（3）任何在设计、开发和调试bootblock时遇到的问题和解决方法

该任务在实现输出显示的时候发现print\_location和printint函数中输入的行列坐标和实际是相反的，因此在观察结果后发现了这个问题并进行了修改。

# Mutual lock设计流程

（1）spin-lock和mutual lock的区别

spin-lock在lock\_acquire时首先先无限循环检测当前锁是否被占用，如果获取锁失败那么该线程切换离开，进入就绪队列队尾，下次切换进入后继续检测锁是否被占用，一直如此循环。如果获取成功，那么修改锁的状态为被占用，然后退出lock\_acquire函数。

mutual lock在lock acquire时也是先检测当前锁是否被占用，如果获取锁失败，那么将此线程放入阻塞队列，将该线程阻塞，等到其从阻塞队列中回到就绪队列并切换到时，再次检测是否可以获得锁，如果仍旧失败则继续放入阻塞队列并不断如此循环，直到获取成功，修改锁的状态为被占用，然后退出。

两者的区别简言之，前者如果获取锁失败会一直不断检测，直到获取成功。而后者如果获取失败，会先进入阻塞队列，等到再次切换到该task时再次检测。

（2）能获取到锁和获取不到锁时各自的处理流程

如果能获取到锁，那么将锁的状态改为被占用，然后退出lock\_acquire函数。如果取不到锁，那么调用block函数，将当前的task进行阻塞并存入阻塞队列，等待当前使用锁的task在lock\_release时从阻塞队列中取出放入就绪队列，并等待再次切换到，然后再次检测是否可以获取锁，如果还是不能，那么再次进入阻塞队列，并不断循环，直到成功获取锁。

（3）被阻塞的task何时再次执行

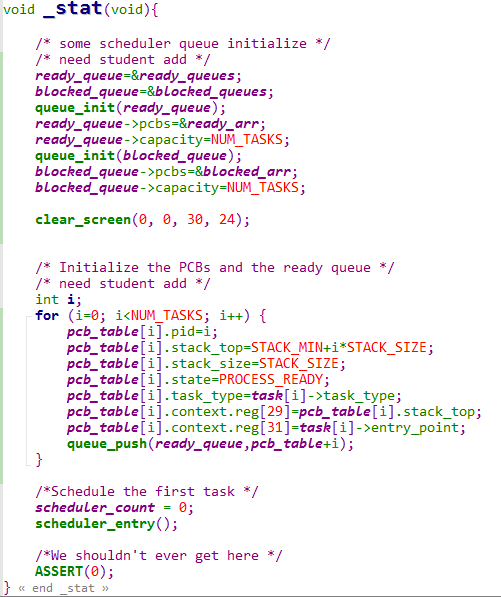
当前占用锁的task进入lock\_release函数释放锁时，检查阻塞队列，如果其中有task被阻塞，那么在其中取一个出来，将其状态从阻塞修改为就绪，然后将其压入就绪队列。在就绪队列中等待，直到切换到该task。

（4）任何在设计、开发和调试bootblock时遇到的问题和解决方法

在关于释放锁时的处理上，一开始的处理方法时，释放锁后同时也将该task切换掉，换新的task运行，但是后来发现，如果采用这种设计，那么需要在unblock函数中调用save\_pcb函数，而在调用前会先准备32的栈空间来保存寄存器，这与yield和block中调用save\_pcb函数前准备24的栈空间不同，这也会导致save\_pcb函数中保存sp和ra寄存器的方法不同。因此后来放弃了该设计，而是释放锁后继续运行当前task。

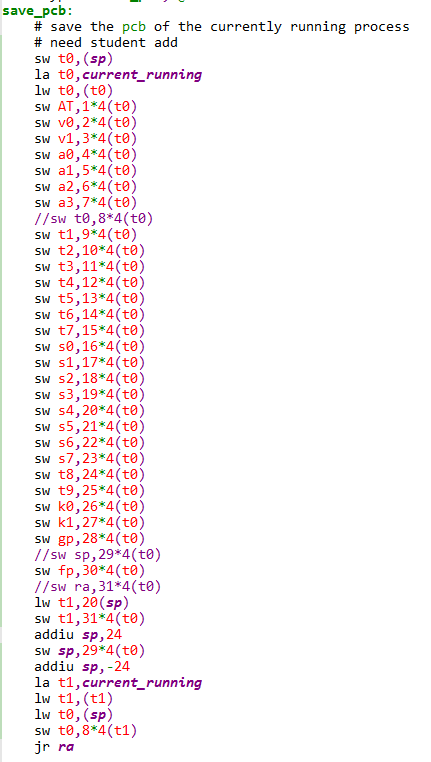
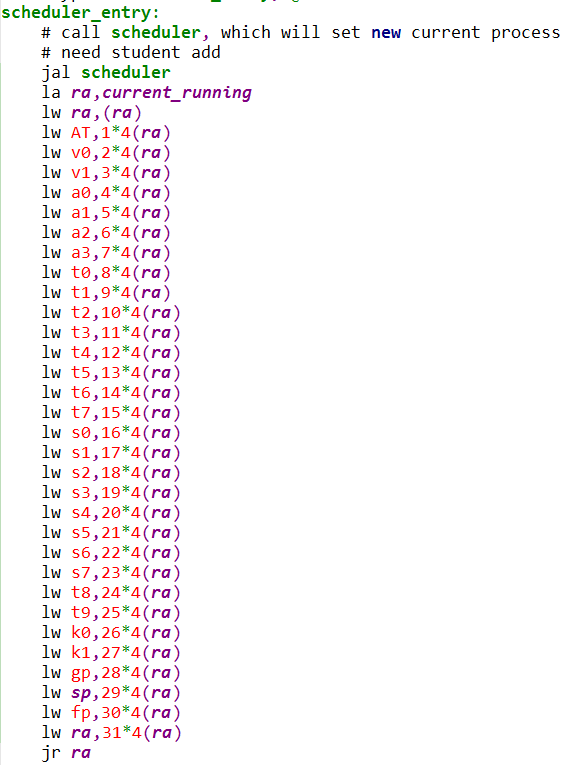
# 关键函数功能

1、kernel.c



这里主要完成了阻塞和就绪队列的初始化，并根据不同任务，把各自所运行的task的PCB表项初始化好，然后分别压入就绪队列，最后调用scheduler\_entry获取第一个task运行。

2、entry\_mips.S

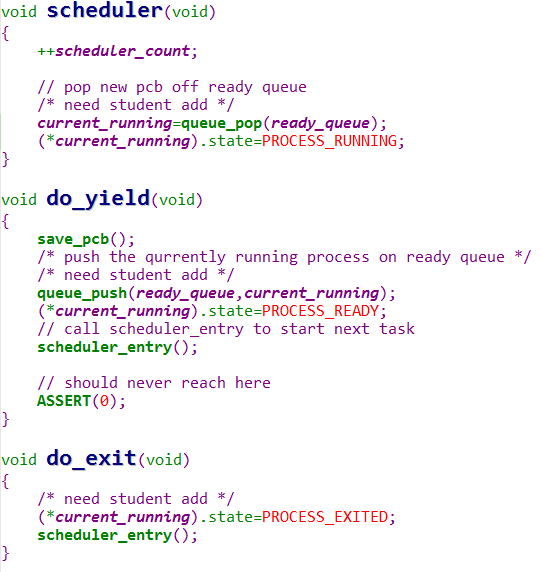


该文件中最重要的两个函数是scheduler\_entry和save\_pcb。

scheduler\_entry先调用scheduler获取新的就绪task，然后将PCB表中该task上次切换离开时保存的寄存器值全部恢复到寄存器中，然后跳转到上次切换时所运行到的地址（即31号寄存器所保存内容）继续运行。

save\_pcb保存进程切换时的寄存器值到对应的PCB表项中，其中sp和ra寄存器的处理需要特别处理，在上面的文档中已经详细介绍。

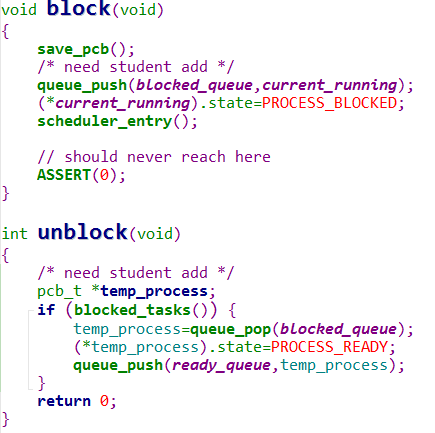
3、scheduler.c



scheduler函数的功能就是从就绪队列中取出一个task并将其状态改为运行。

do\_yield函数首先保存当前task的PCB表项，然后将当前task压入就绪队列队尾，并将其状态从运行改为就绪，接着再调用scheduler\_entry函数获取新的task并运行。

do\_exit函数首先将当前task的状态改为退出，并调用scheduler\_entry函数获取新的task并运行。



block函数首先保存当前task的PCB表项，然后将该task的状态从运行改为阻塞，并放入阻塞队列，然后获取新的task运行。

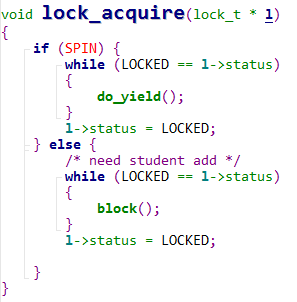
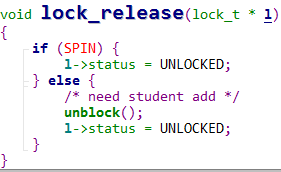
unblock函数首先检查阻塞队列有没有task，如果有，那么取出其中一个task将其状态从阻塞改为就绪并放入就绪队列。

4、th3.c



这两个线程主要是计算线程之间以及线程与进程间切换花销，并输出结果，具体原理已在上述文档中阐述。

5、lock.c



lock\_acquire函数首先循环检测是否锁可以，如果可用则更改锁的状态为被占用然后退出，否则将当前task阻塞。

lock\_release函数将锁的状态更改为空闲，并调用unblock函数将阻塞队列中的task取至就绪队列。