**基于内核栈切换的进程切换**

**1．课程设计的内容及要求**

现在的 Linux 0.11 采用 TSS（后面会有详细论述）和一条指令就能完成任务切换，虽然简单，但这指令的执行时间却很长，在实现任务切换时大概需要 200 多个时钟周期。

而通过堆栈实现任务切换可能要更快，而且采用堆栈的切换还可以使用指令流水的并行优化技术，同时又使得 CPU 的设计变得简单。所以无论是 Linux 还是 Windows，进程/线程的切换都没有使用 Intel 提供的这种 TSS 切换手段，而都是通过堆栈实现的。

本次实践项目就是将 Linux 0.11 中采用的 TSS 切换部分去掉，取而代之的是基于堆栈的切换程序。具体的说，就是将 Linux 0.11 中的 switch\_to 实现去掉，写成一段基于堆栈切换的代码。

本次实验包括如下内容：

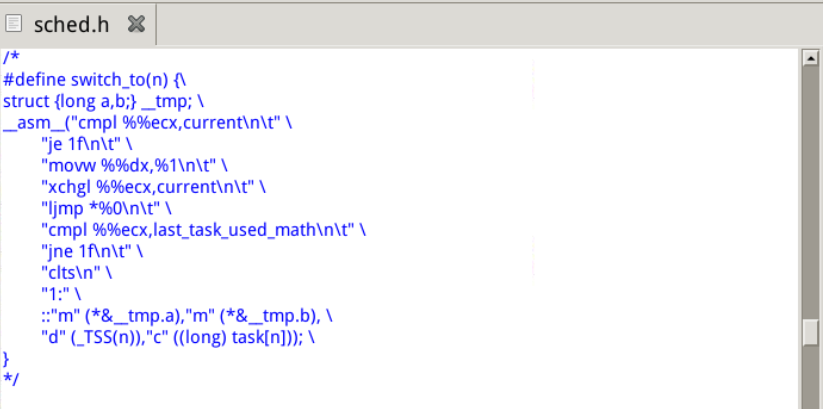
* 编写汇编程序 switch\_to：
* 完成主体框架；
* 在主体框架下依次完成 PCB 切换、内核栈切换、LDT 切换等；
* 修改 fork()，由于是基于内核栈的切换，所以进程需要创建出能完成内核栈切换的样子。
* 修改 PCB，即 task\_struct 结构，增加相应的内容域，同时处理由于修改了 task\_struct 所造成的影响。
* 用修改后的 Linux 0.11 仍然可以启动、可以正常使用。
* 分析实验 5 的日志体会修改前后系统运行的差别。

**2．选择和使用现代工具**

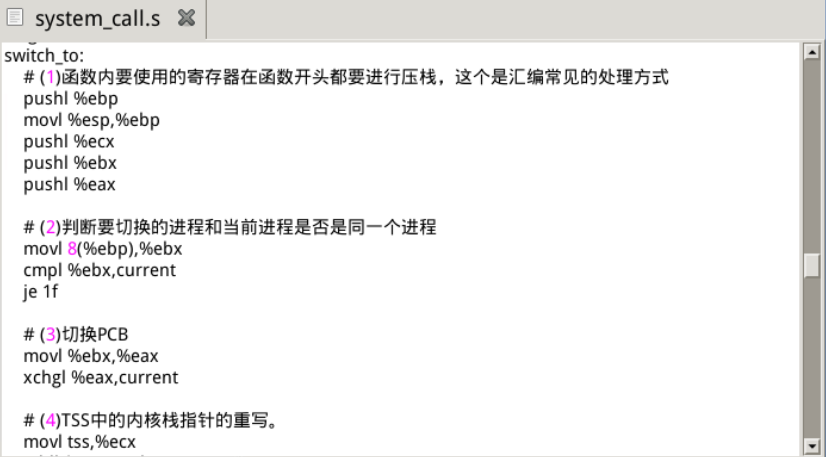
蓝桥云虚拟机

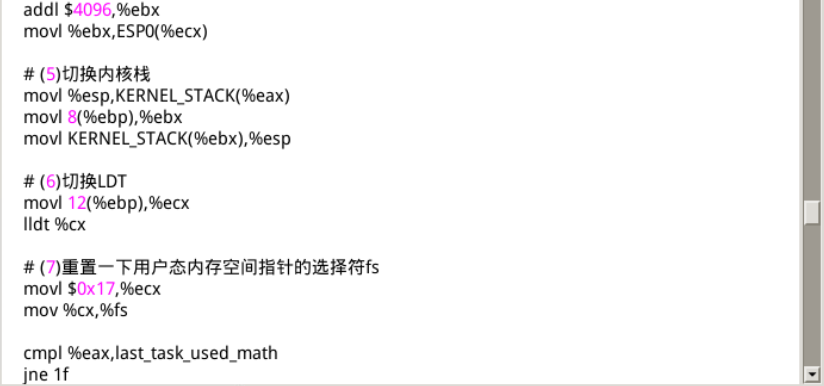
**3．课题实现**

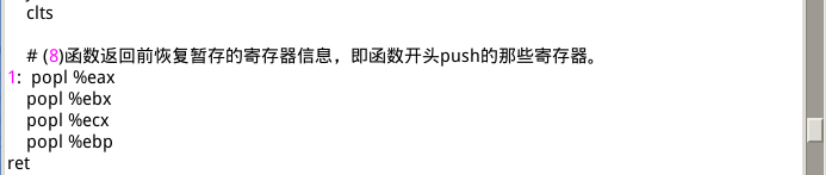
1. 注释掉switch\_to原来在include/linux/sched.h的宏定义



1. 在 kernel/system\_call.s 中添加以下 switch\_to 汇编代码

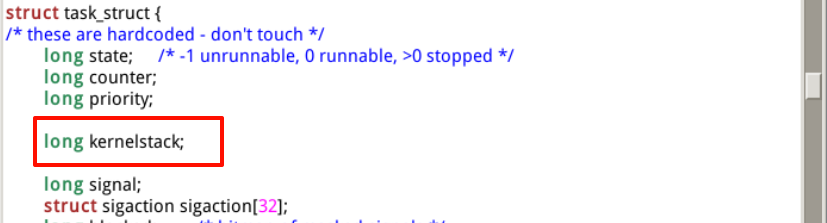




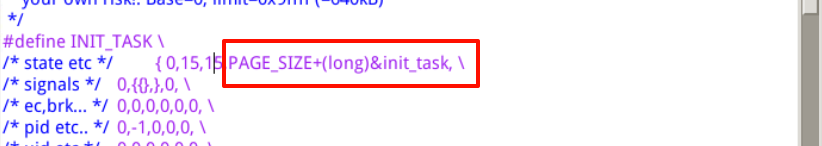


1. 改写task\_struct

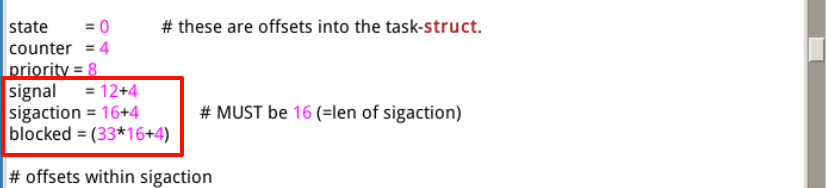
添加kernelstack变量



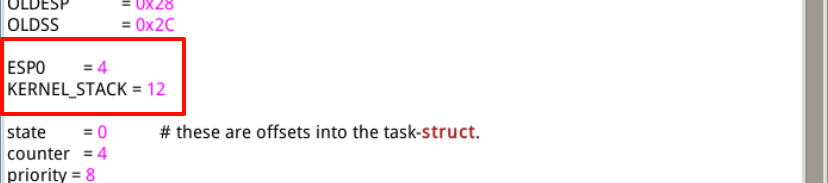
对应的INIT\_TASK修改



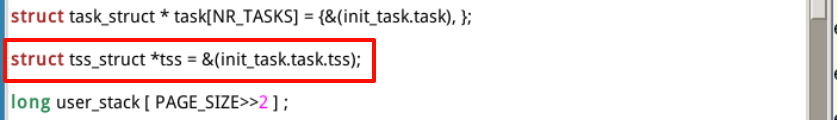
在system\_call.s中的一些硬编码修改



1. 声明ESP0和KERNEL\_STACK

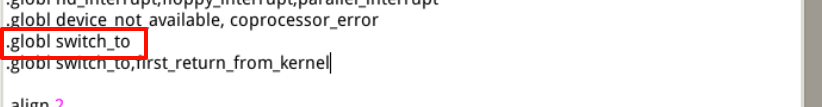


1. 声明进程0的tss

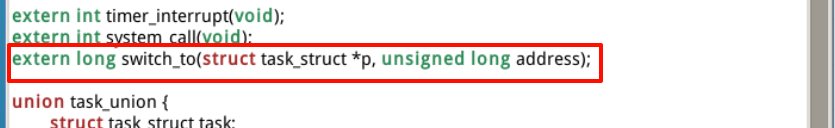


1. 将switch\_to和schedule()接在一起

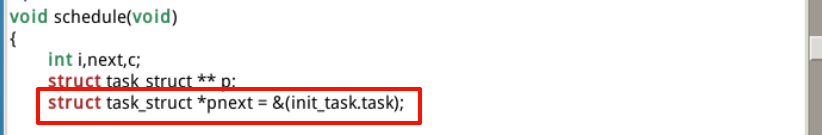
system\_call.s的头部添加代码来开放switch\_to

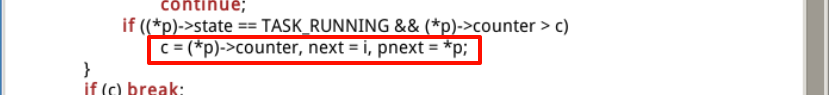


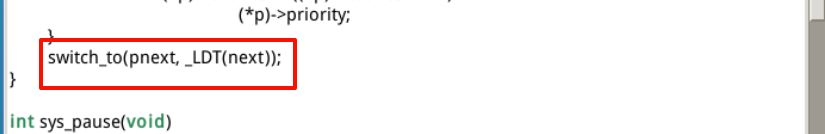
在sche.c声明switch\_to



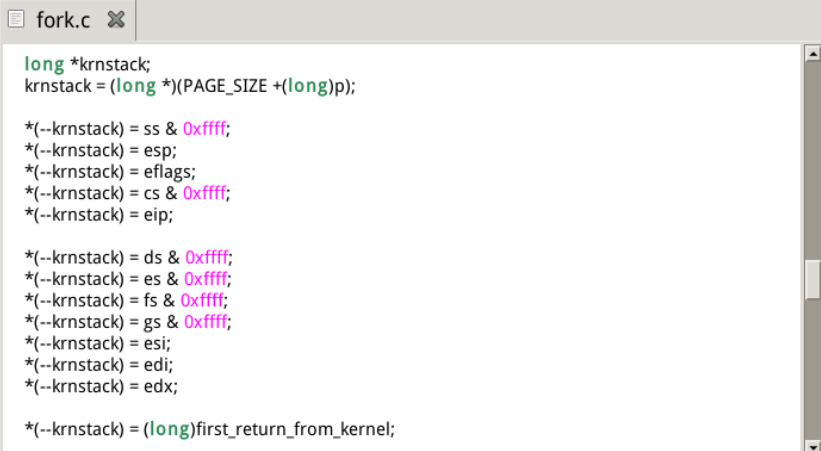
在schedule()调用switch\_to

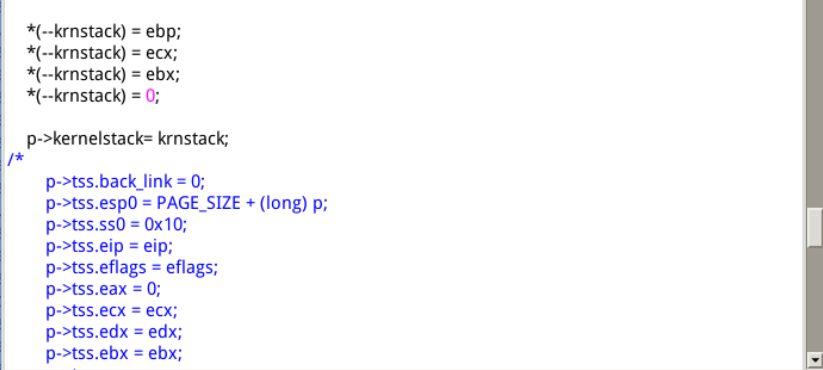


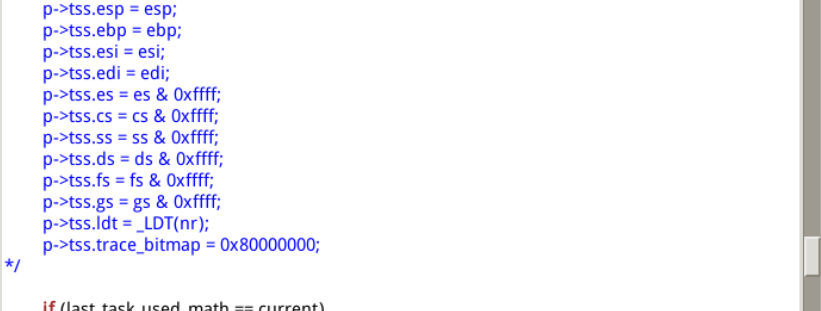




1. 在fork.c中修改copy\_process并且注释掉之前tss相关代码







1. 对first\_return\_from\_kernel进行编写

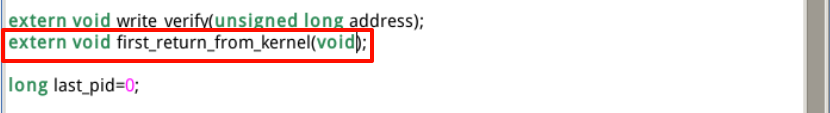
编写first\_return\_from\_kernel



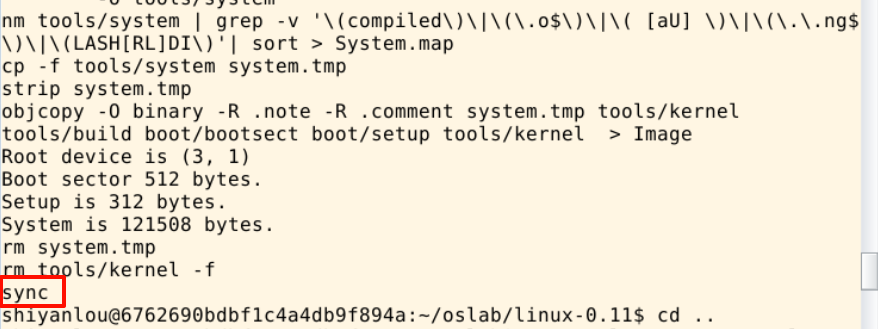
把first\_return\_from\_kernel声明为全局函数



在fork.c中声明要使用这个外部函数

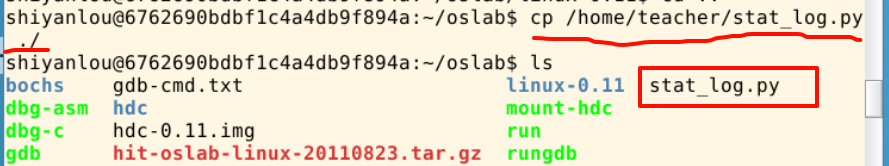


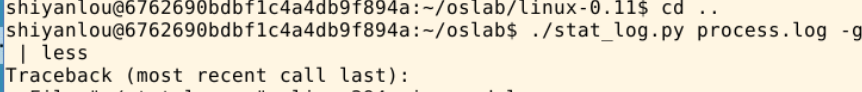
1. 编译内核



将stat\_log.py文件转到shiyanlou/oslab中，添加权限，然后导入实验五中的process.log进行分析，并且将时间片调整为5，进行对比。

将stat\_log.py文件转到shiyanlou/oslab中

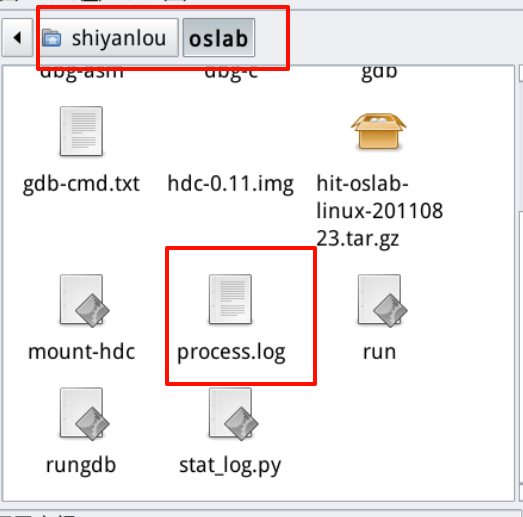




给stat\_log.py增加权限



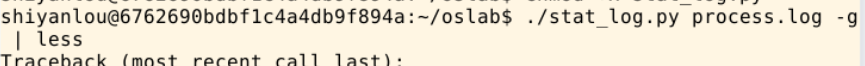
导入实验五中的process.log文件



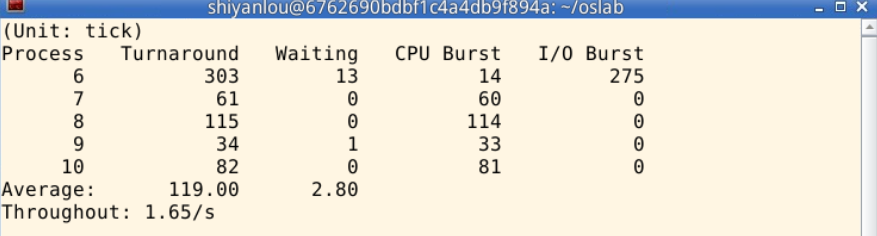
修改时间片



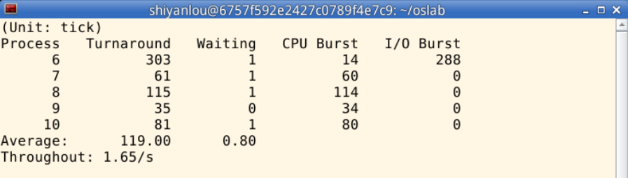
运行stat\_log.py



分析结果如下：

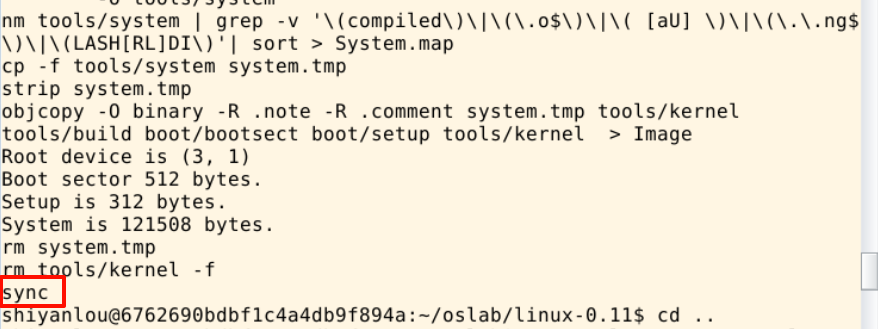


实验五的结果摘录如下：



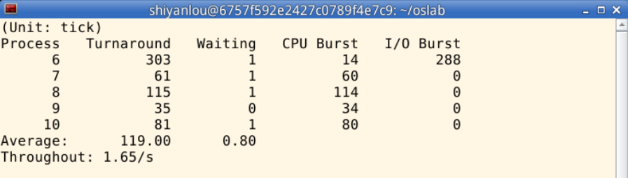
**4．课题实现结果**

1. 编写完全后可以正常的内核编译

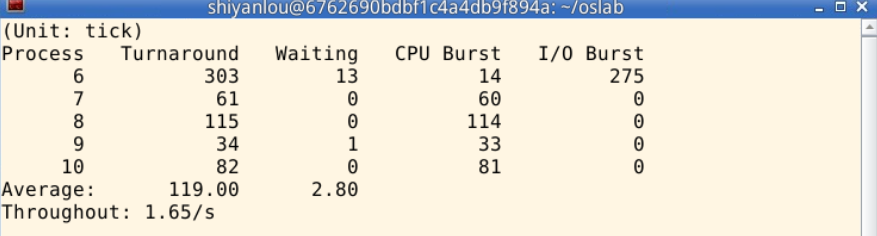


1. 对于实验5的日志体会修改前后系统运行对比。

修改前：



修改后：



通过比较修改前后的截图结果可知，平均周转时间和吞吐量在修改前后保持一致，但等待时间的分布发生了细微变化，特别是进程6的等待时间由1增加到13。这种变化主要是由于采用内核栈切换时，进程上下文保存与恢复的机制发生了改变，导致时间片分配和进程等待队列的调度细节有所不同。

**5．结果分析与心得体会**

5.1 结果分析

本实验通过修改Linux 0.11内核实现了基于内核栈切换的进程切换，替代了原有的TSS（任务状态段）切换机制。修改后的内核能够正常编译和运行。

通过比较修改前后的截图结果可知，平均周转时间和吞吐量在修改前后保持一致，但等待时间的分布发生了细微变化，特别是进程6的等待时间由1增加到13。这种变化主要是由于采用内核栈切换时，进程上下文保存与恢复的机制发生了改变，导致时间片分配和进程等待队列的调度细节有所不同。

5.2 心得体会

TSS切换虽然实现简单，但其执行时间较长，内核栈切换通过直接操作栈指针ESP来完成上下文保存和恢复，相比之下更加高效。

灵活性增强： 基于内核栈的切换机制更加灵活，易于与现代处理器的流水线并行优化技术结合，进一步提升系统性能。

通过改写swtich\_to、schedule和fork等核心代码，对进程的上下文切换进行了深入分析，并逐步掌握了内核栈的实现原理。

优点：

* 熟练掌握了进程切换的实现细节，包括PCB（进程控制块）和内核栈的操作。
* 能够独立完成内核代码的修改与调试，并验证修改后的内核运行稳定性。

不足：

* 对进程调度策略的理解还需进一步加强，特别是在分析等待时间和时间片的影响方面，可以结合更复杂的调度算法进行深入优化。
* 对实验结果的对比分析不够深入，尚未结合更大规模的进程调度场景进行验证。

**参考文献**

<https://blog.csdn.net/qq_42718041/article/details/124970546?fromshare=blogdetail&sharetype=blogdetail&sharerId=124970546&sharerefer=PC&sharesource=William_lyx&sharefrom=from_link>

<https://blog.csdn.net/weixin_40909092/article/details/118072123>

<https://blog.csdn.net/lyj1597374034/article/details/111033682?spm=1001.2014.3001.5506>

<https://zhuanlan.zhihu.com/p/11141942147>