|  |
| --- |
|  |
| Lab1/3/6实习报告 |
|  |

|  |
| --- |
| 姓名：吴悦欣 学号：1900012946  日期：2021/10/16 |

目录

[内容一：实验总结 3](#_Toc84842190)

[内容二：遇到的困难以及收获 11](#_Toc84842191)

[内容三：对课程或Lab的意见和建议 12](#_Toc84842192)

[内容四：参考文献 12](#_Toc84842193)

## 内容一：实验总结

1. Lab1

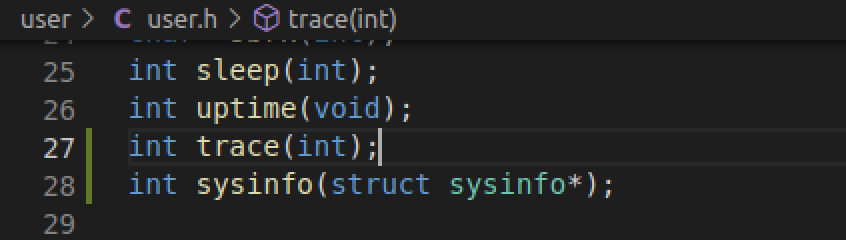
本次实验共有两个部分，都是针对系统调用的实现展开的。

第一部分是实现**trace系统调用**，该函数的功能是：当给出的mask参数指定的系统调用被调用时，会输出一行包含当前进程的pid、对应系统调用名称以及该调用的返回值的字符串。原代码提供了trace的用户调用代码。该过程的系统调用过程如下：

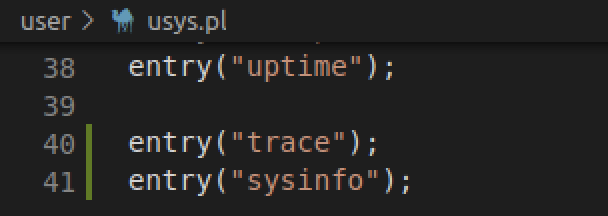
1. 在用户程序中调用int trace(int mask)函数后，会把该进程的a7寄存器的值赋为对应的系统调用号SYS\_trace（定义在syscall.h中），函数调用需要的参数存放在a0寄存器中
2. 通过ecall指令进入内核【该过程在usys.pl中实现】
3. 进程切换后，内核开始执行syscall，查看a7中断的调用号【该过程在syscall.c的syscall函数中】调用对应的函数sys\_trace()，通过argint函数sys\_trace函数可以获得对应的参数【从trapframe结构的a0中获得参数】，并返回需要的值，放在a0寄存器中返回。

因此如果需要打印对应的结果，只需要在syscall的函数中判断函数调用号是否对应mask之后输出相关信息即可。下面是完成该函数的具体步骤（可以根据git版本管理在左侧显示的颜色条分辨我进行的改动）。

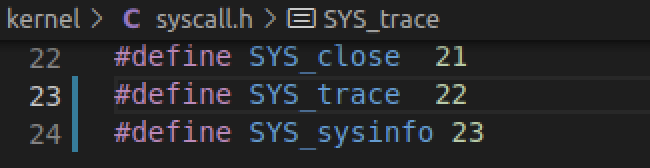
1. 在user/user.h中添加trace函数的声明



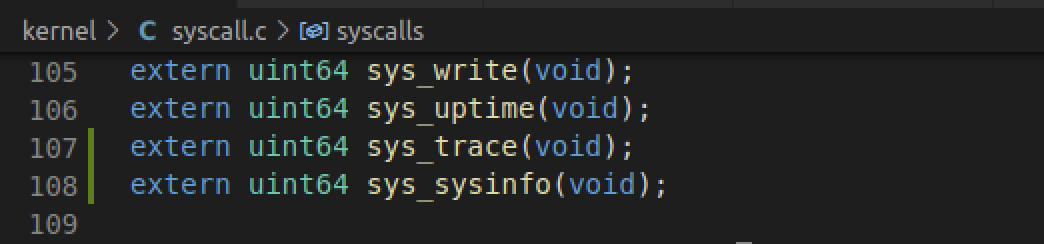
1. 在user/usys.pl中添加trace函数系统调用的入口

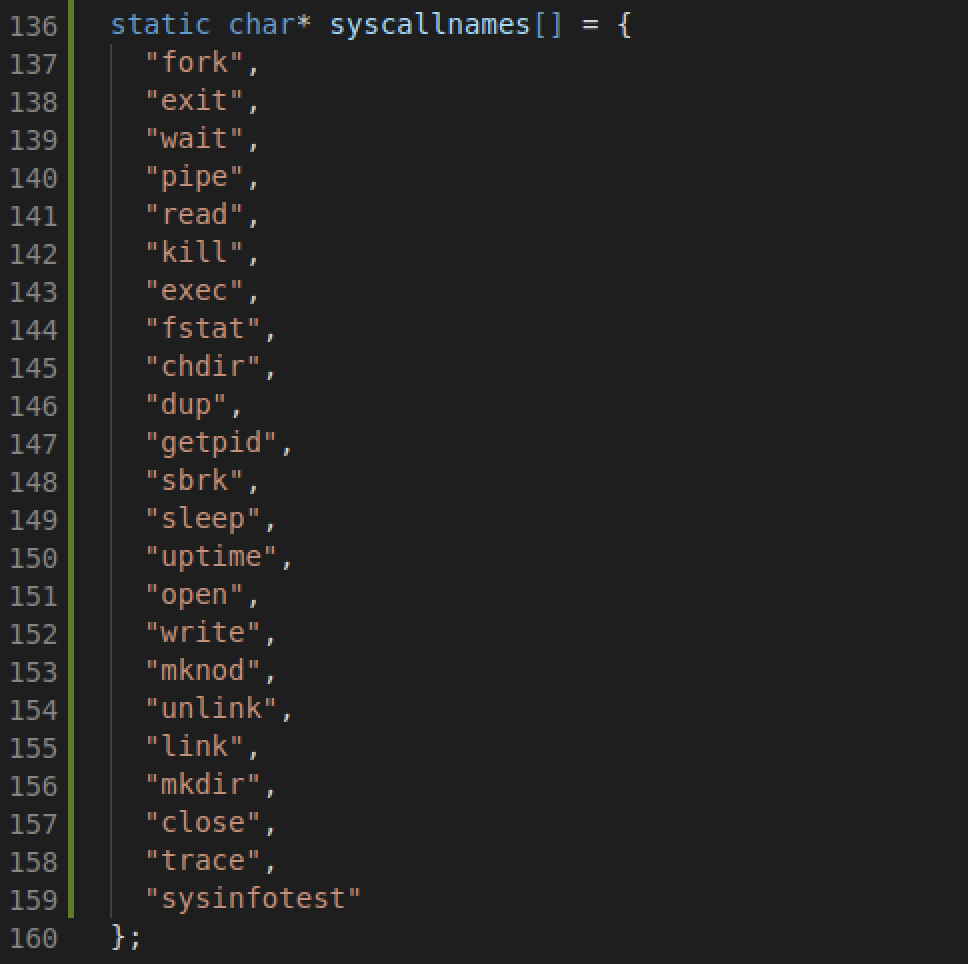
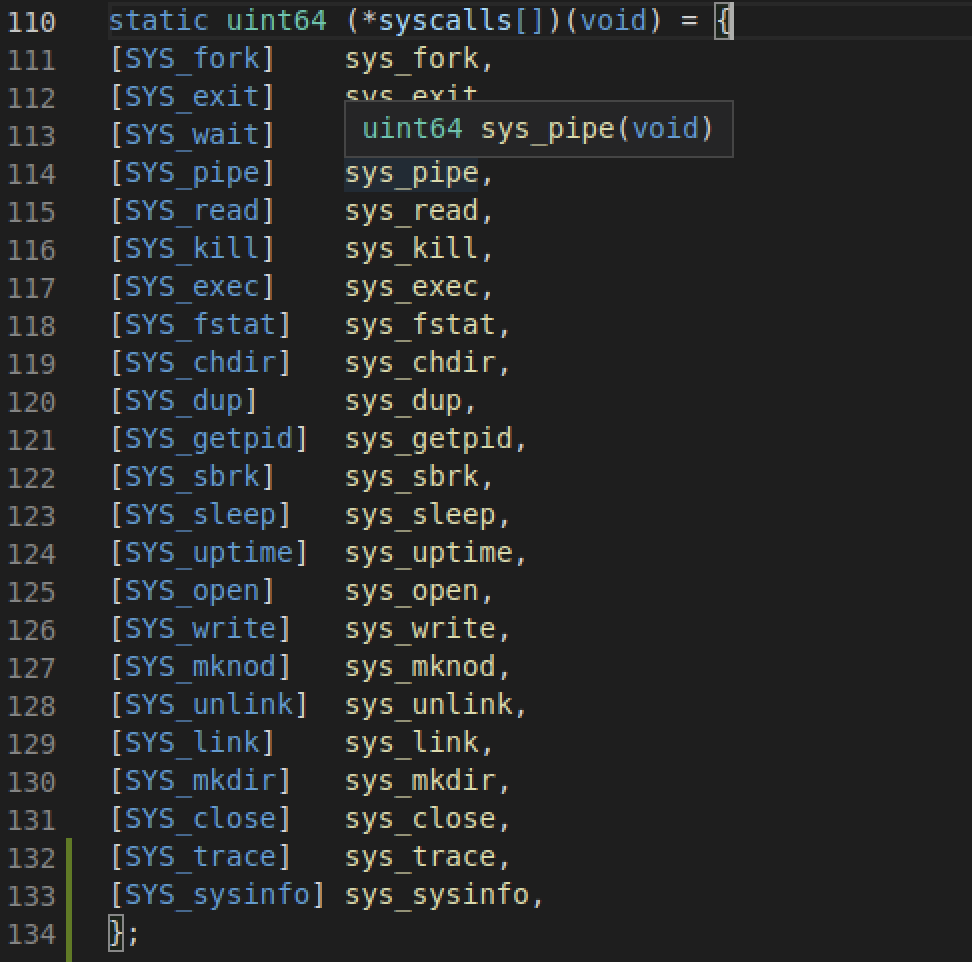


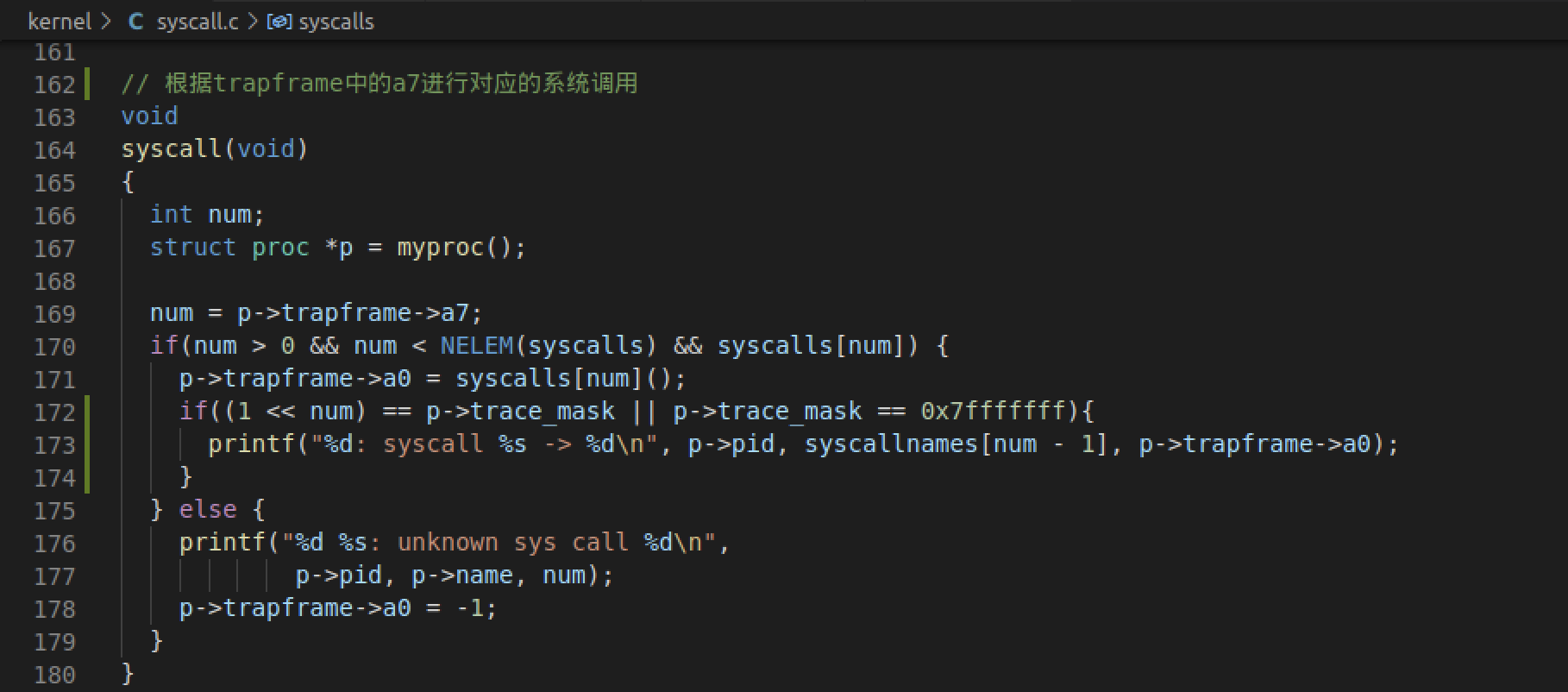
1. 在kernel/syscall.h中添加对应的系统调用号



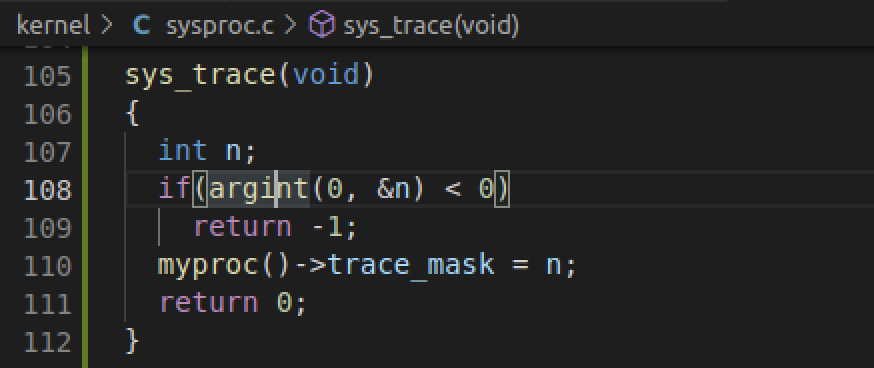
1. 在kernel/syscall.c中添加对应的函数声明，函数指针，用来输出函数名称的字符串数组以及修改syscall实现trace要求的功能

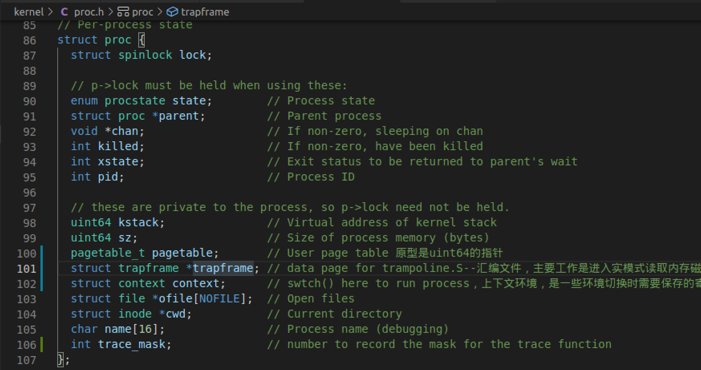




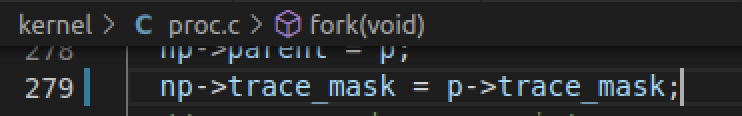


1. 在kernel/sysproc.c中添加对应被调用的函数sys\_trace()，同时在proc的进程结构体（kernel/proc.h）中增加一个变量来记录标志了需要打印的函数对象的mask



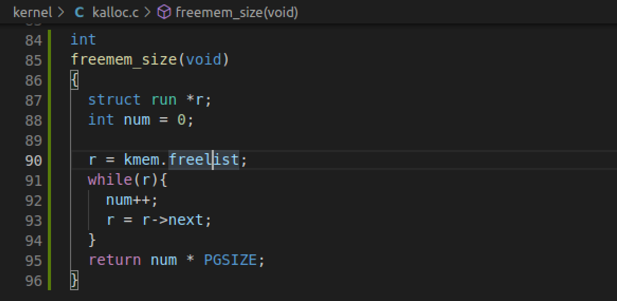


1. 修改fork()函数（kernel/proc.c）使得子进程继承父进程的trace\_mask，打印对应的调用过程

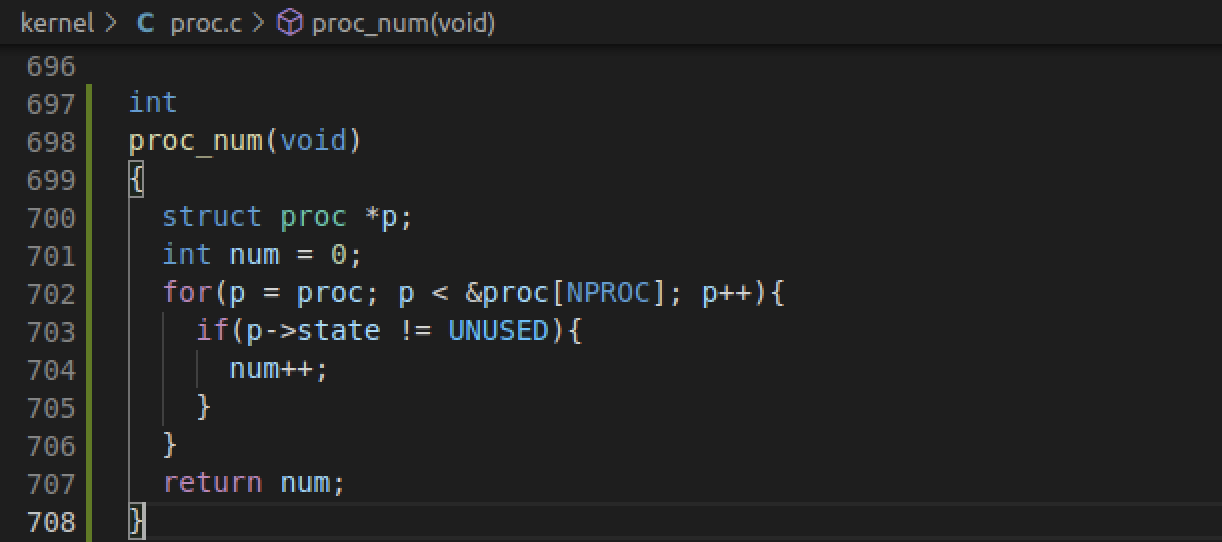


在第二个任务中是需要完成**sysinfo**系统调用，该函数的主要功能是：将当前进程剩余空间的字节数以及非空闲的进程数赋值给对应结构体sysinfo的属性。在实现系统调用的过程类似于第一个trace的实现，需要添加对应的入口、系统调用号、声明以及对应的系统调用函数sys\_sysinfo，不同的在于需要另外实现对剩余空间和非空闲进程的计数：

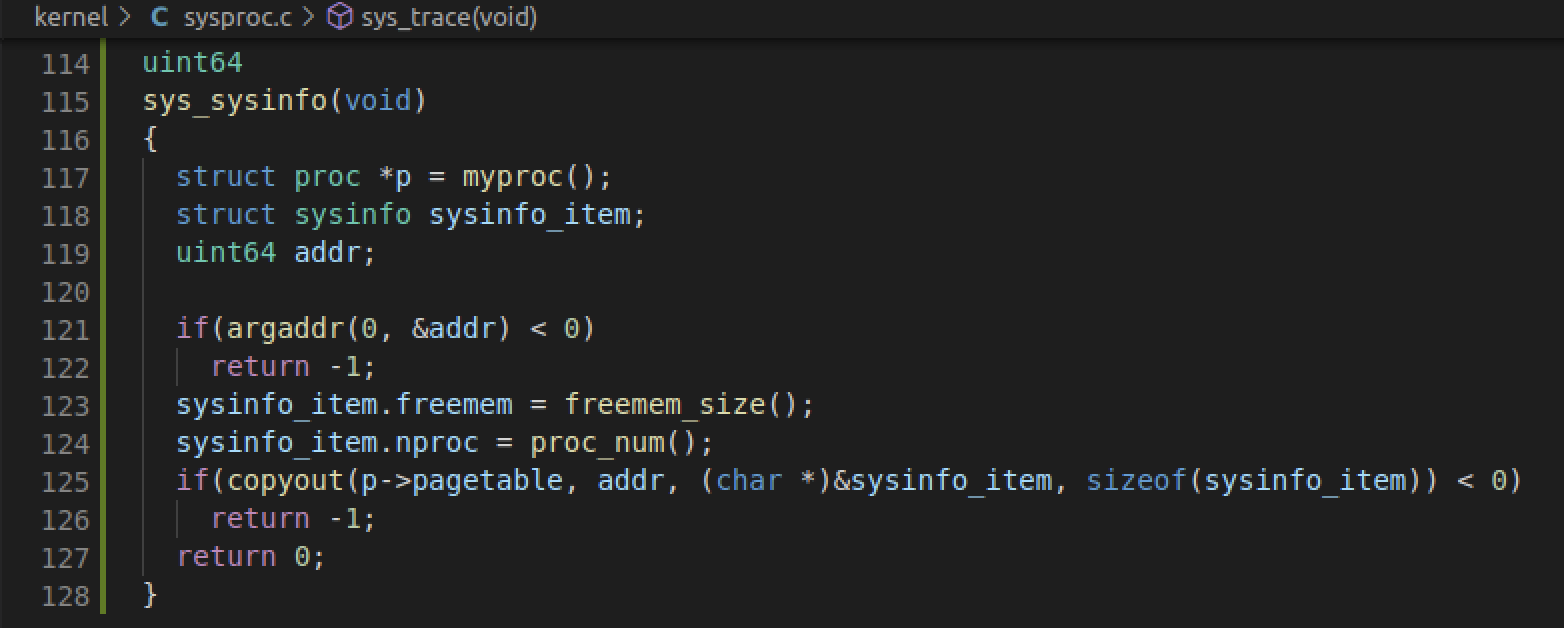
1. kernel/kalloc.c中对剩余空间的字节数：kmem是内核内存的结构体，其中freelist属性指向第一个空闲的页，因此只需要计算剩下的空闲页的个数\*页大小即可。



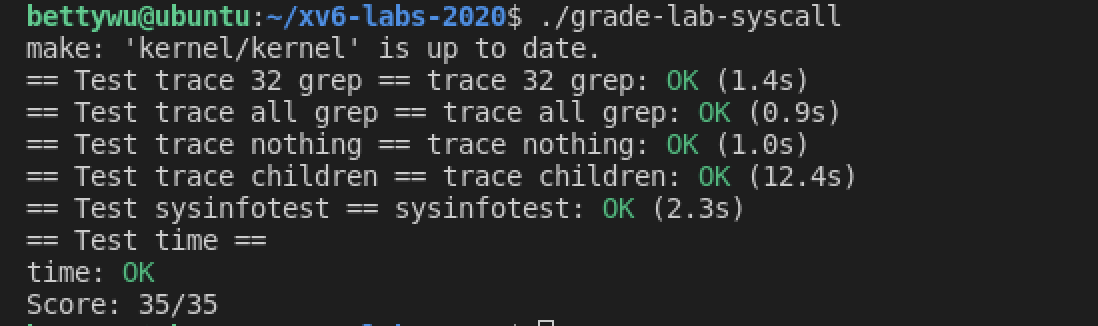
1. kernel/proc.c中对非空闲空间的计数：proc为所有进程的集合，遍历所有进程对象判断其状态进行计数即可。



1. 在kernel/sysproc.c中书写sys\_sysinfo函数，调用1，2中的函数给传入的指针对应的sysinfo结构体赋值，返回进程信息。



根据评分要求添加了time.txt并写入用时5h，lab1最终的测试结果如下：



1. Lab3

本次实验是关于trap陷阱的实现。在做实验前，先根据课本对应内容阅读了kernel/trampoline.S部分的代码，这是一个切换用户/内核空间的汇编文件，其中uservec部分在进入内核空间前保存了用户进程的寄存器状态，并将进程的trapframe中关于内核的部分进行对应的赋值，切换到内核进程；之后调用kernel/trap.c中的usertrap函数，对于陷阱的类别进行区分后进行对应的处理——系统的调用（syscall）或是时钟的中断或是其他的异常（exit）；最后再回到trampoline.S中的userret部分恢复用户空间。通过这一步，我对于trap的机制有了大致初步的了解。

实验的**第一部分**是根据一个测试代码的汇编结果回答对应的问题，以下是问题及对应的回答：

1. Which registers contain arguments to the functions? For example, which register holds 13 in main's call to printf?

函数调用通过a0, a1, a2...a8传递参数，返回值放在a0或a1寄存器。在printf中，由a2传递13这个参数。

1. Where is the call to function f in the assembly code for main? Where is the call to g?

在main函数中，f的调用被优化了，在printf之前直接把12这个结果赋值给了对应的寄存器。而对g的调用是在f函数中，在汇编中也被内联优化，没有调用的过程，直接将g对应的汇编实现加入到f的汇编中。

1. At what address is the function printf located? 0x640
2. What value is in the register ra just after the jar to printf in main? 0x30
3. What is the output? If big -endian, what is the proper value for i? And if the change to 57616 be needed?

Output: He110 World

都不需要改变，57616的十六进制打印不会被大小端影响，而i的值的存储方式被强制类型转换成字符串之后和字符的存储顺序也是一致的，所以不需要变换。

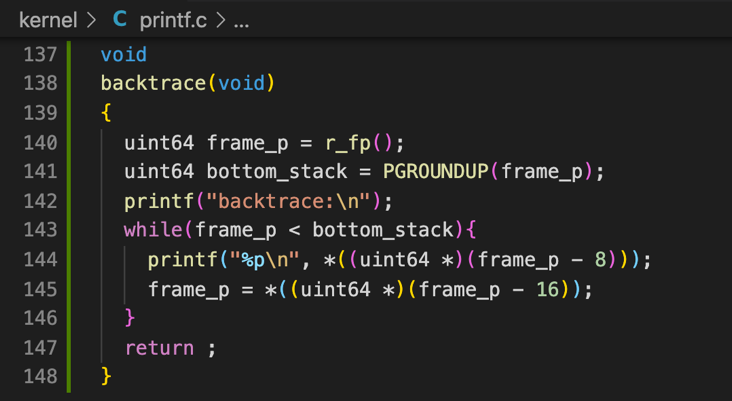
6. What will be printed after 'y='? Why?

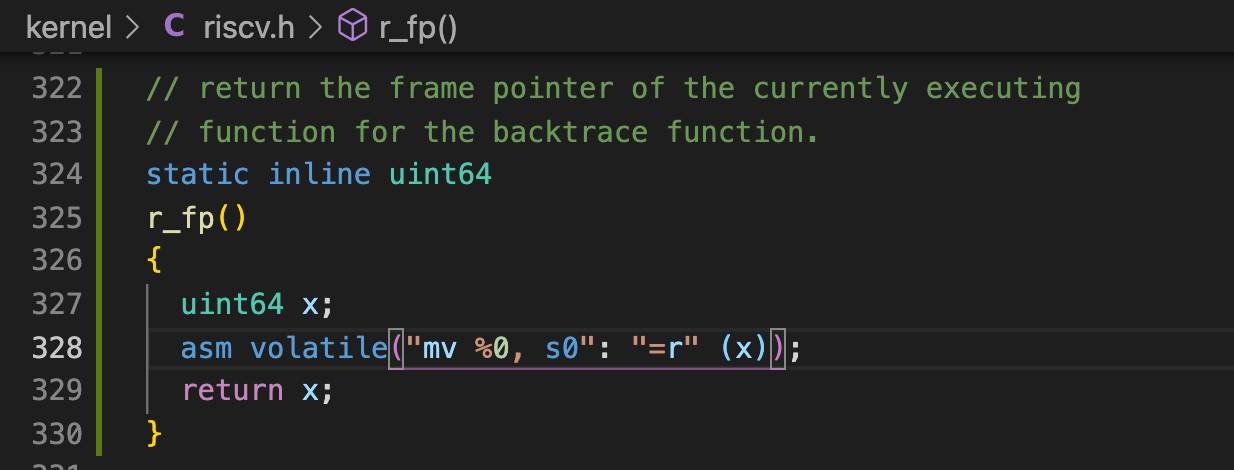
-280324136；应该是寄存器a2中的值，printf会在a2获取y的值

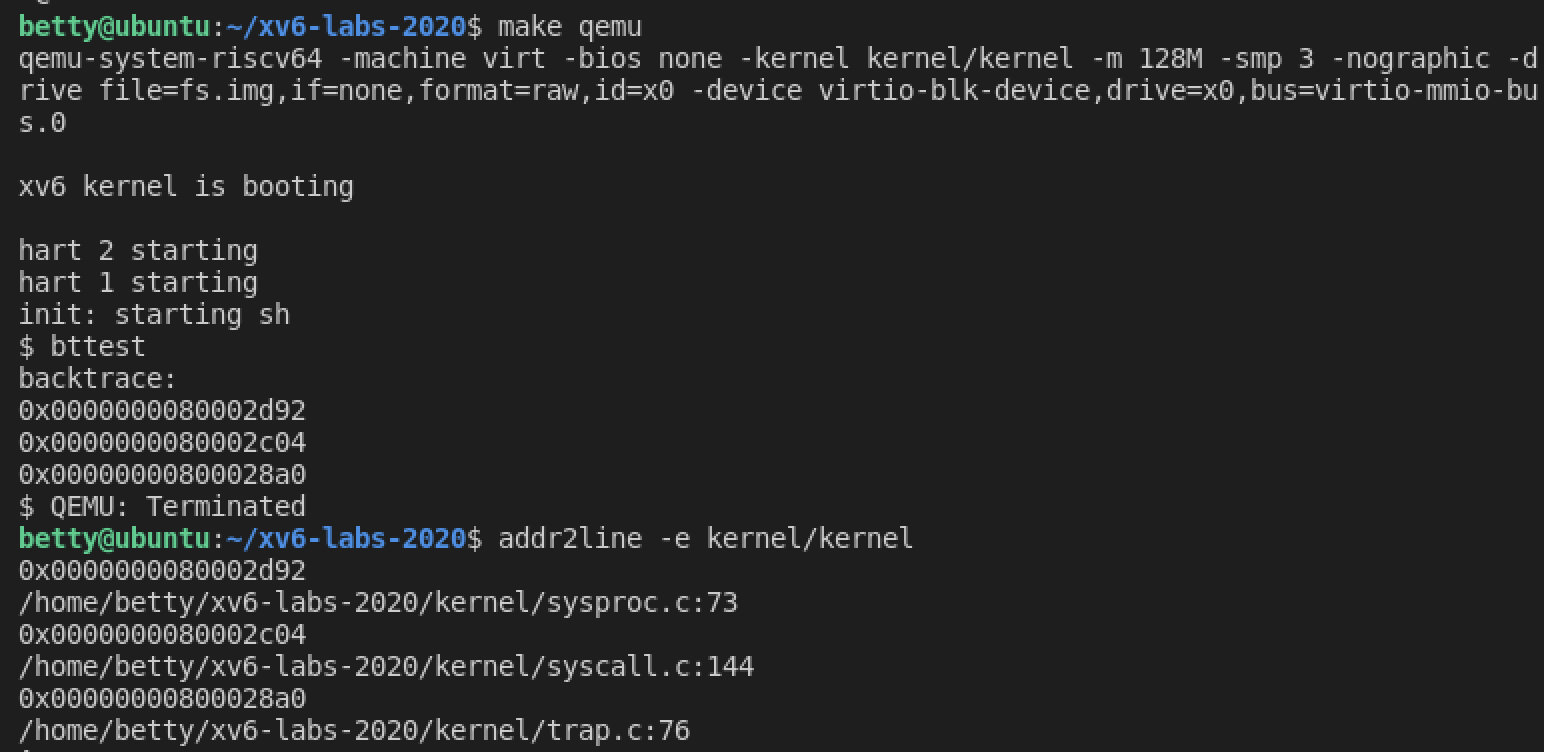
主要解决的问题是函数调用过程中的传参方式、内联优化、函数跳转的定位、大小端的区别。

**第二部分**是实现函数backtrace，该函数在被调用的时候打印在目前调用的函数栈帧地址空间之上的栈中的函数（即经过层层调用到达当前的函数，希望得到整个调用的路径）。

主要的修改是在kernel/printf.c中增加backtrace函数以及在kernel/riscv.h中增加内联汇编将当前栈帧所在寄存器s0的值返回，相关代码和运行结果如下：

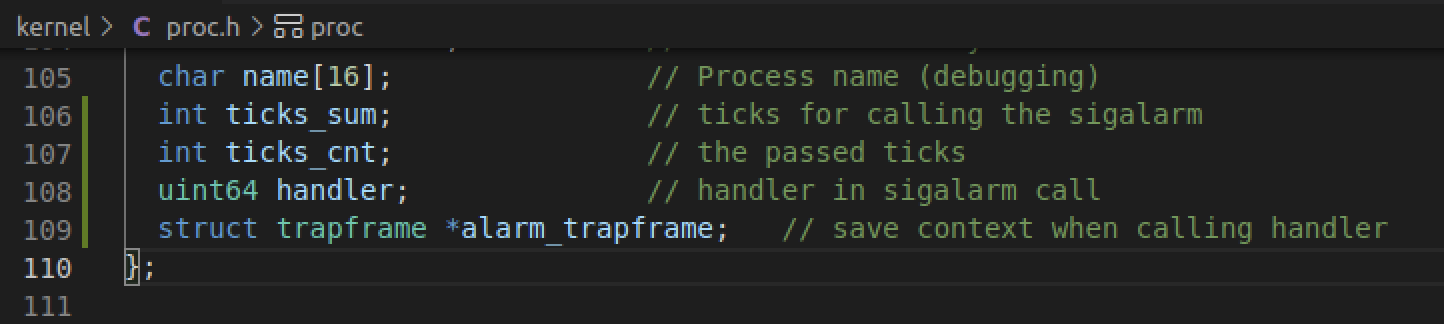




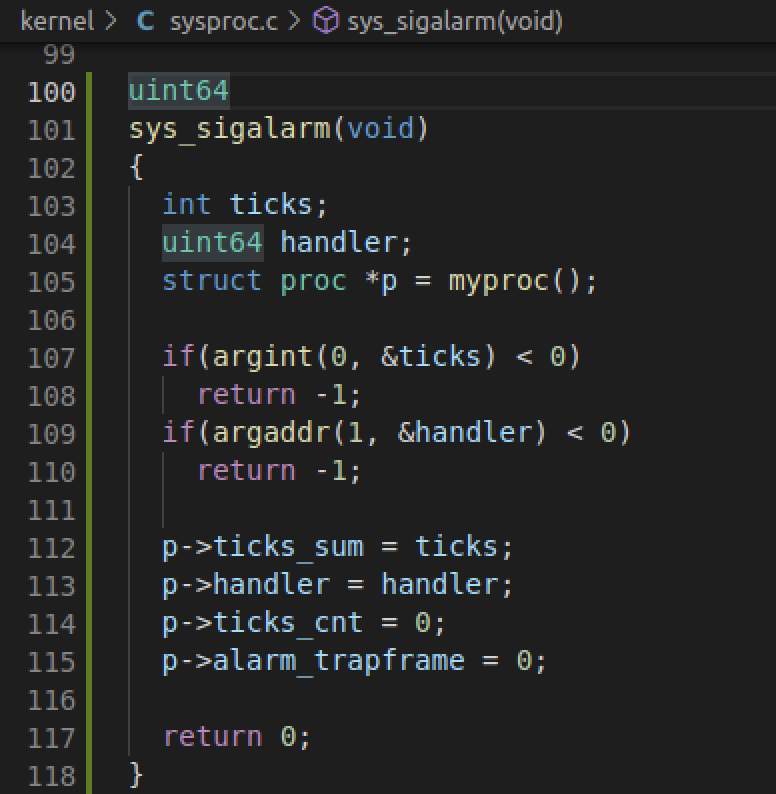


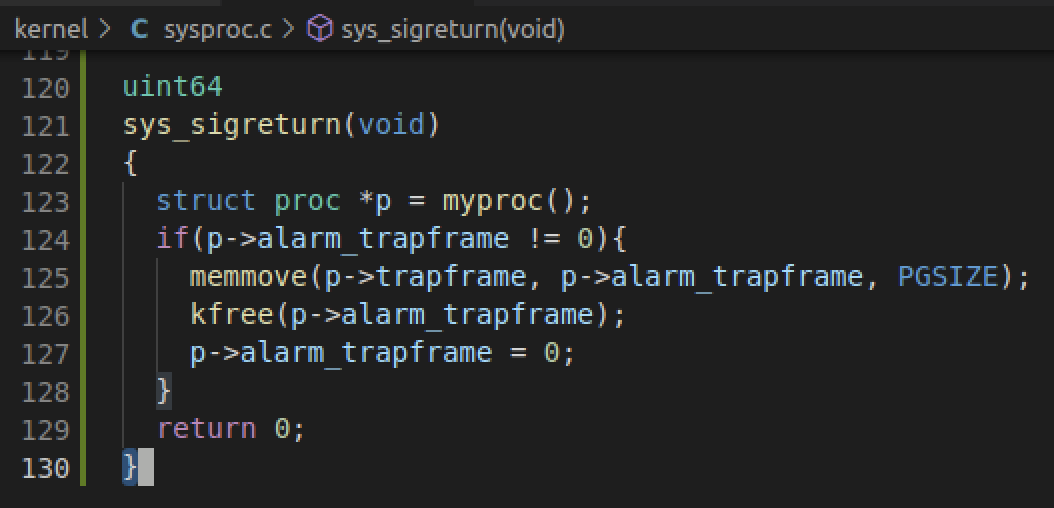
**第三部分**是实现一个系统调用alarm函数，其功能是通过两个参数——整数指定间隔时间，函数指针指定调用的函数——每每隔一段特定的系统的时间就调用一次函数。系统调用的增加和Lab1的过程相同，此处不再赘述，只关注具体的函数实现：

1. 修改kernel/proc.h中proc的结构体，增加成员变量来记录时间间隔、已经过去的时间、函数指针、调用对应函数时保存原进程的寄存器的trapframe。

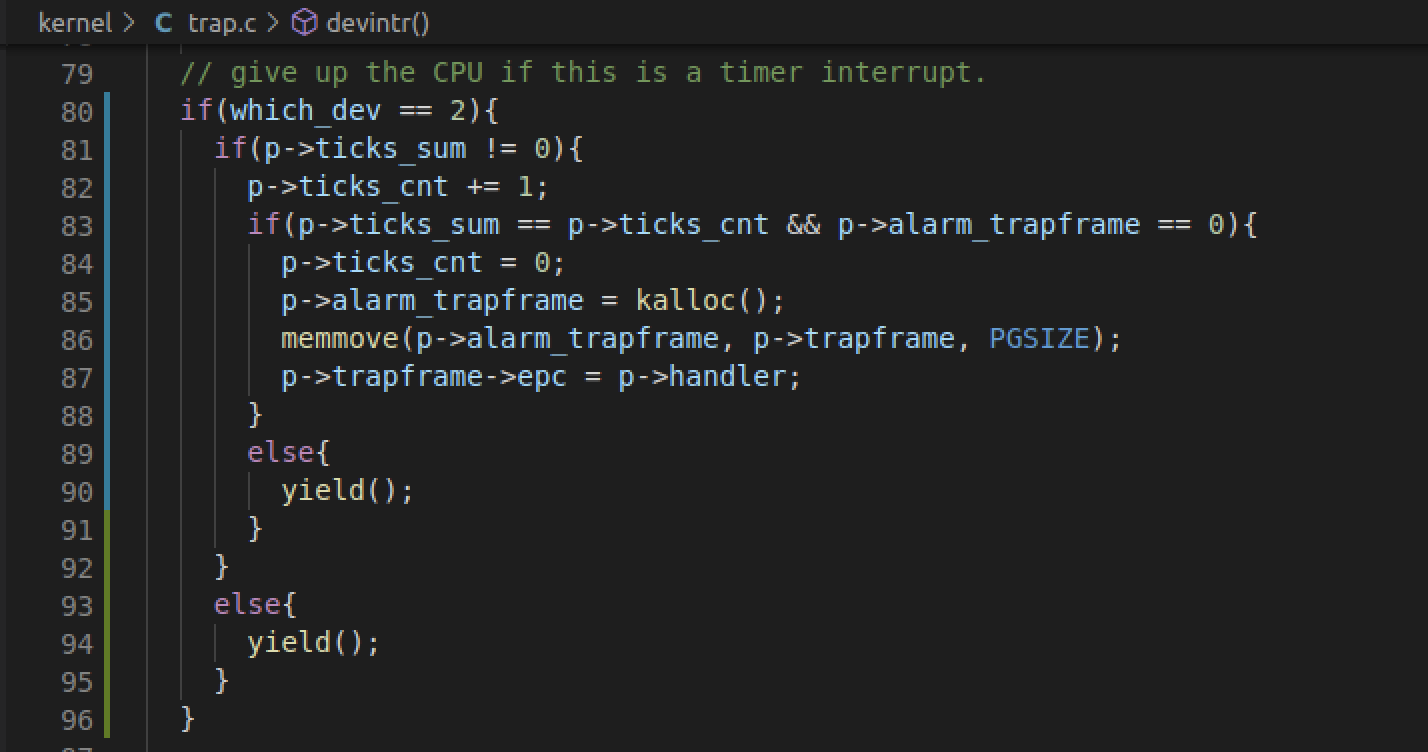


1. 撰写sigalarm和sigreturn对应的系统函数，传参的过程与Lab1相似，同时为对应进程的成员变量赋值。

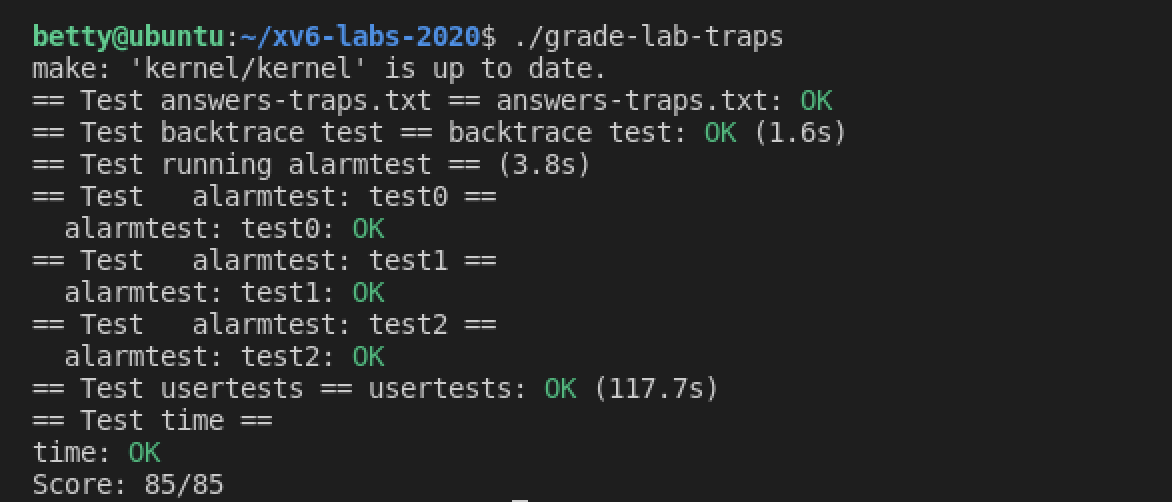




1. 在每个tick（系统的时间单位）的中断中增加当前进程的ticks计数，并且判断是否进行对应的函数调用。这一部分在kernel/trap.c的usertrap函数中实现，时钟中断对应which\_dev（trap原因的序号）为2的情况



Lab3最终的测试结果如下

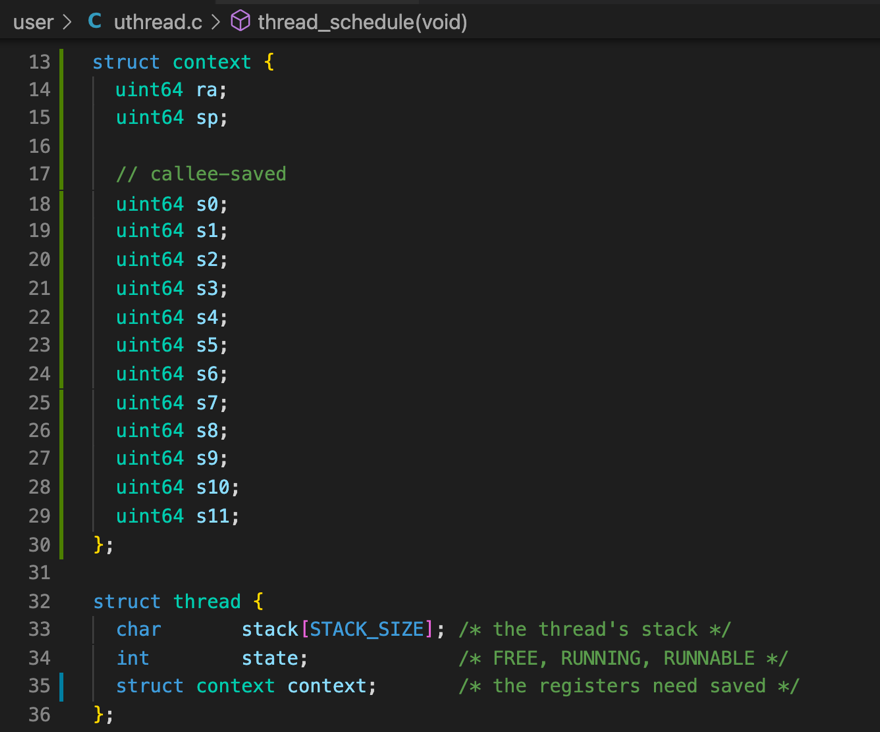


1. Lab6

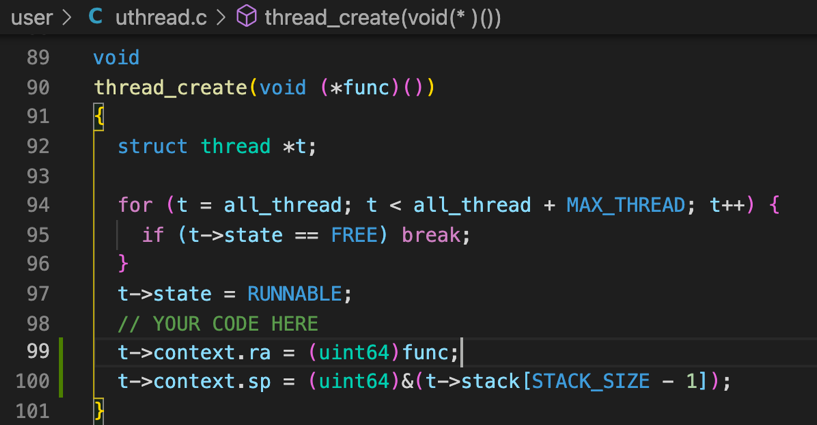
Lab6是关于多线程的实验，在实验之前阅读了xv6说明的第七章和相关的代码，主要是关于xv6中进程切换的实现、sleep和wakeup机制的实现及应用的介绍。其中进程切换主要是依靠kernel/proc.c中的schedular和sched函数的上下文切换、调用swtch函数实现；后续sleep，wakeup函数还有相关的pipe，wait，exit，kill函数的介绍中强调了进程锁的重要性和其中微妙的顺序。

**第一个实验Uthread**是实现用户层面的线程切换，这个task比较简单，但是需要注意的是栈的增长方向应当是高地址向低地址增长，所以在使用数组定义的栈thread->stack的时候要注意栈底是在&thread->stack[STACK\_SIZE]。实验具体的实现如下：

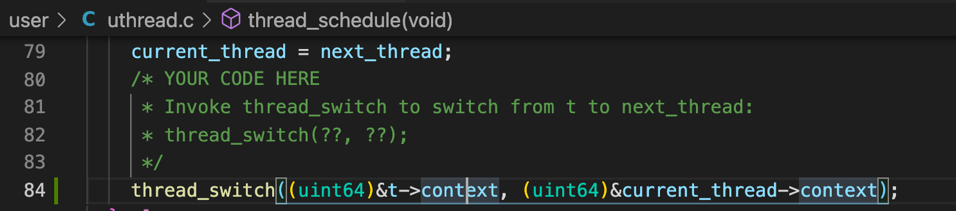
1. 在user/uthread.c完善thread结构体，增加存储上下文（寄存器）的结构context



1. 在user/uthread.c完善thread\_create函数，将参数中的函数指针作为开始该线程时先执行的函数，并让栈指针指向栈底



1. 在user/uthread.c完善thread\_schedule函数，进行线程的上下文切换

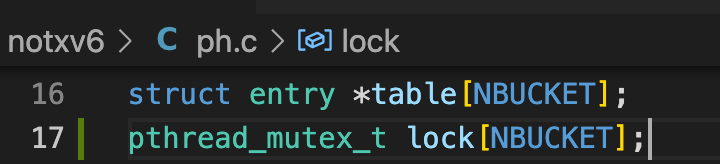


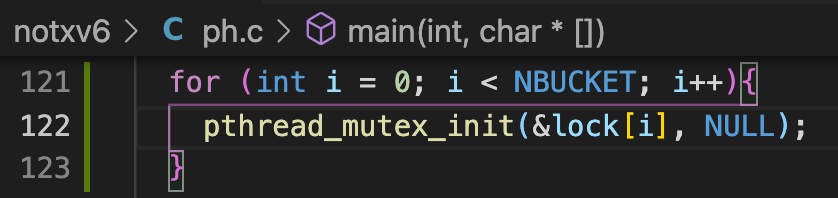
**第二部分**是观察notxv6/ph.c代码，思考多线程可能带来的问题并且通过pthreads提供的pthread\_mutex加锁的机制来避免该错误并保持多核带来的高效率。在原本ph.c文件中多核可能带来问题原因是：

线程共享全局变量，所以它们拥有同一个table。在只有一个线程的时候该线程会按照顺序依次存入表项，但是多个线程（这里以两个为例）时，就有可能出错。按照如下的执行顺序就会丢失表项：thread1执行到put函数中找到了key1可以放置的空项，即将执行insert函数此时cpu调度到thread2，thread2也需要放key2项，而且找到了和thread1一样的空槽，而由于thread1尚未插入表项，因此在thread2看来这是一个可以插入新表项的位置，由此即会导致thread1或者thread2的表项丢失（如果先调度thread1进行insert，那么thread1的entry丢失，否则thread2的丢失）

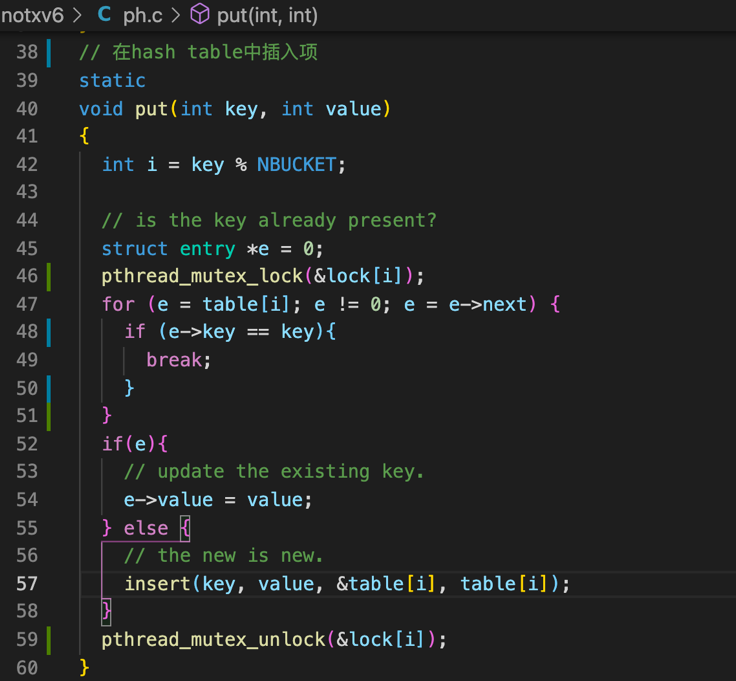
为了解决这个问题添加的代码如下：

1. 为每个桶加锁，并进行初始化

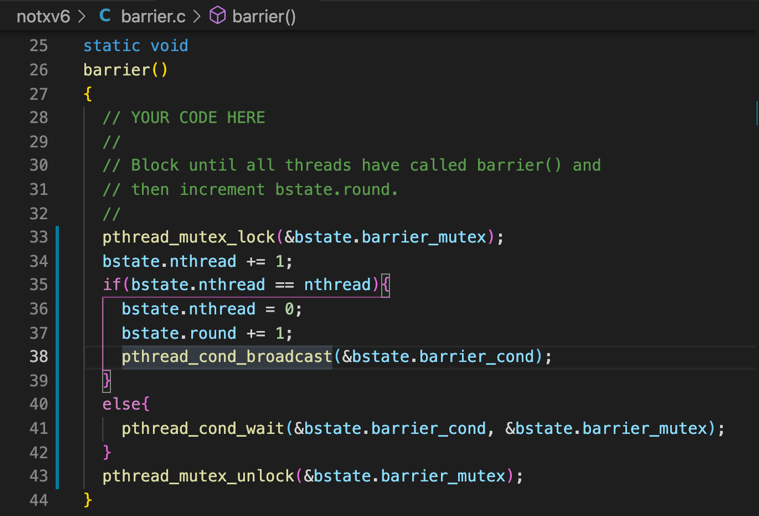




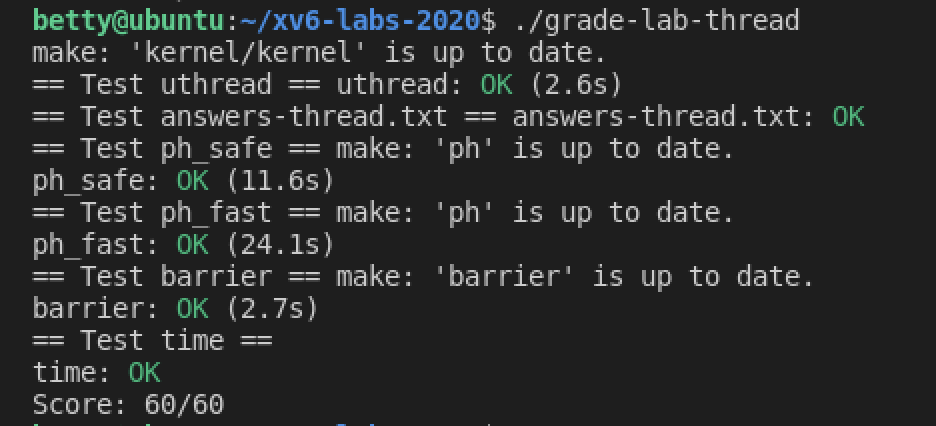
1. 在put中每个线程检查桶是否有对应的表项前加锁，防止其他线程修改该桶的状态；同时在该线程插入/修改项完成后归还锁。



**第三部分**是完善notxv6/barrier.c中的barrier函数，使之完成所有线程都要等到其他线程到达barrier函数之后才能离开的功能。主要的想法是使用pthread\_cond\_wait和pthread\_cond\_broadcast函数，前者的功能是在指定的cond条件上等待，并且释放原本持有的锁，在返回的时候重新取得锁；还是要注意使用互斥量，防止在对bstate.uthread计数增加的时候产生race。下图是函数的具体实现。



Lab6最终的结果如下



## 内容二：遇到的困难以及收获

1. Lab1

在第一个Lab中主要的困难是对于系统调用具体实现的不熟悉，加上内核代码体量较大，因此理解起来比较困难，但是根据官方的指南逐个文件修改，做完第一个trace函数之后，重新梳理了一遍系统调用的顺序，做第二个的时候就明了很多了。

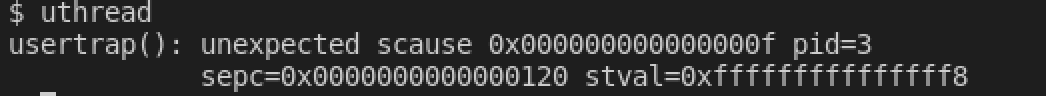
1. Lab3

最开始切换branch的时候遇到了因为没有commit而不能直接切换的问题（从Lab0到Lab1的时候没碰到），查阅了一些资料觉得需要保留之前的一些修改，所以采用的stash和stash pop的方法。但是写完backtrace函数后发现不能运行，因为切换分支的时候覆盖了MakeFile文件中对Lab1设置的命令，而一些文件例如sysinfo.h没有了，所以要手动删掉多余的文件，后来才意识到应该之前直接commit让它覆盖/删除掉不需要的文件，这里折腾了比较久。

在backtrace的实现中，因为在取返回地址的时候没有弄清楚原本得到的只是地址所在的位置的指针，因此打印出来的地址在addr2line中一直不对，也卡了很久。完成试验后对于trap的寄存器的存储过程以及其中重要的一些量的替换（例如trapframe->epc）以及为什么最终的实现可以完成我们以前宏观看到的那种效果有了更多原理上的了解。

1. Lab6

还是碰到了地址和实际的值直接没分太清楚的情况，在第一部分中一开始完全没发现在thread\_create函数中给sp寄存器赋初值时用的是t->stack[STACK\_SIZE-1]，而没有使用取地址符，所以一直报下图的错误。



第三部分的完成过程中主要遇到的是对pthread\_cond\_wait一开始的理解不太正确，对于所有的线程都一刀切的在增加uthread都调用这个函数，后来才理解到说这样会导致所有的进程都wait，就不存在一个可以继续执行uthread判断后进行broadcast的wakeup操作的线程了。

在本次线程相关的实验中，我对线程的运作原理还有xv6中线程切换的过程都有了更多的了解，同时，对互斥量的使用也更加熟悉了；其中感觉理解起来最困难的是sleep和wakeup函数的实现部分。在第一部分其实就有的“pipe如何实现没有可以read的内容就阻塞等待”这个问题，也在后续的实验操作和学习中得到了答案。

## 内容三：对课程或Lab的意见和建议

可以增加一些小问题上的提示，比如说做完之后要commit。有些同学在看到中文的guide之后就不太看英文的，但是原本的英文guide其实要详细很多，纠结在中文的guide上容易浪费时间，也许可以完善一下中文的guide或者在上面注释让大家去原本的guide上查看更多详细的步骤。

## 内容四：参考文献

[1] pthread\_cond\_wait函数的定义：

<https://baike.baidu.com/item/pthread_cond_wait/3011997?fr=aladdin>

[2] 大小端：

<https://www.cnblogs.com/yinheyi/p/5580789.html>

[3] 函数指针：

<https://blog.csdn.net/wangqingchuan92/article/details/77863373>

[4] git commit 报错：

<https://blog.csdn.net/weixin_34248023/article/details/88831183?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_title~default-0.no_search_link&spm=1001.2101.3001.4242>

[5] auipc+jal指令：

<https://zhuanlan.zhihu.com/p/354654279>

[6] 内联汇编：

<https://blog.csdn.net/goaqnfv59125/article/details/70767097>

[7] exec族函数

<https://blog.csdn.net/zhengqijun_/article/details/52852074>