# OS-lab2-Report

21302010042

侯斌洋

实验一：procnum

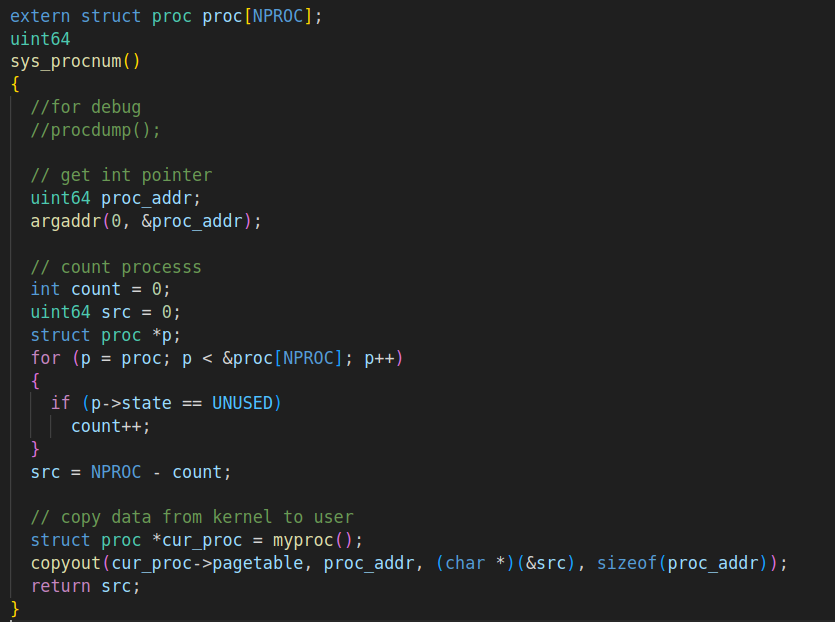
（1）实验思路

先观察proc.c文件的结构，了解xv6中与process相关的信息。在proc.c文件中，发现有个函数procdump()可以打印出当前进程的状态，可以用来debug。然后参考procdump()的函数结构写出统计进程数的代码，即//count process所在的代码块。此时要解决的问题是如何把统计到的值从kernel传到user。因为kernel与user的内存地址不同，所以无法通过直接获取指针并修改的方法传递数据。观察syscall.c文件，可以发现有与系统调用传参相关的代码，这里通过argaddr()函数获取用户内存空间的地址，再通过copyout()函数将值src写入该用户内存空间的地址来实现传递数据，即//get int pointer 和 //copy data from kernel to user 所在的代码块。

（2）实验过程

①实验代码：

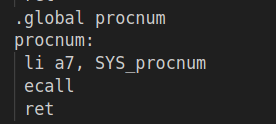
**[kernel/sysproc.c]**



②系统调用流程：

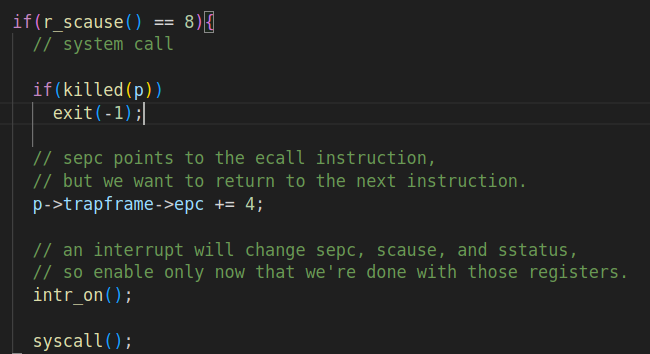
当在shell中输入procnum时，shell在其子进程中运行user/procnum.c中的代码，并在procnum.c中启动系统调用procnum(&num)，(注：在user.h和usys.pl中标记了该系统调用及其入口)。 之后执行usys.S（注：由usys.pl生成）中对应汇编代码，即：

**[user/usys.S]**



这里li指令将SYS\_pronum（注：记录在kernel/syscall.h中）的系统调用号保存在 a7 寄存器中。 然后调用ecall陷入内核，运行uservec保存用户寄存器状态到trapframe中（详细操作在kernel/trampoline.S中），运行usertrap判断trap类型并处理（详细操作在kernel/trap.c中）。如果是系统调用的话，就将pc指向ecall的下一条指令，然后交给syscall函数处理，即：

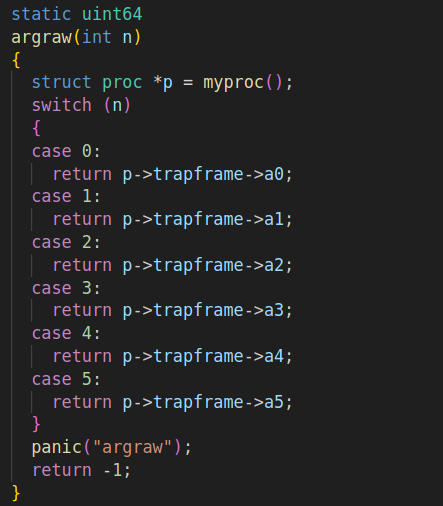
**[kernel/trap.c]**



syscall获取 a7 寄存器中的系统调用号并执行相对应的系统调用，然后就到了sysproc.c中由我们编写的sys\_procnum函数，完成该调用后将返回值保存在trapframe->a0中。最后调用usertrapret()返回到用户空间，trap结束，至此系统调用的过程结束。

在上面的过程中系统调用的参数是通过uservec保存的trapframe来传递的，例如使用argraw()获取参数：

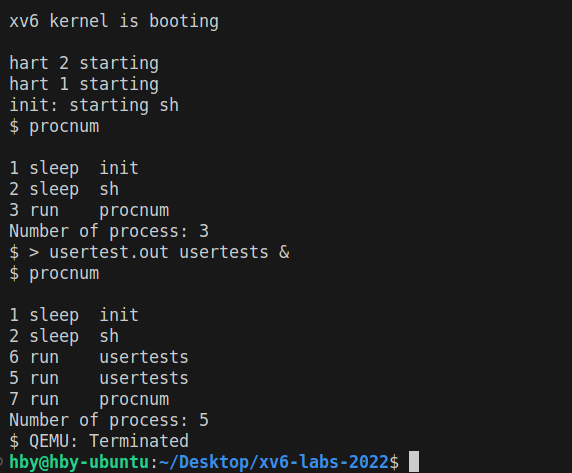
**[kernel/syscall.c]**



如果要在内核空间中读写用户空间某个地址的数据，则要使用专用的copyin和copyout函数（在kernel/vm.c中）通过当前进程对应的页表来进行操作。

（3）实验结果

【注：这里为了体现程序的正确性使用了procdump()函数来打印当前正在运行的进程。】



实验二：freemem

（1）实验思路

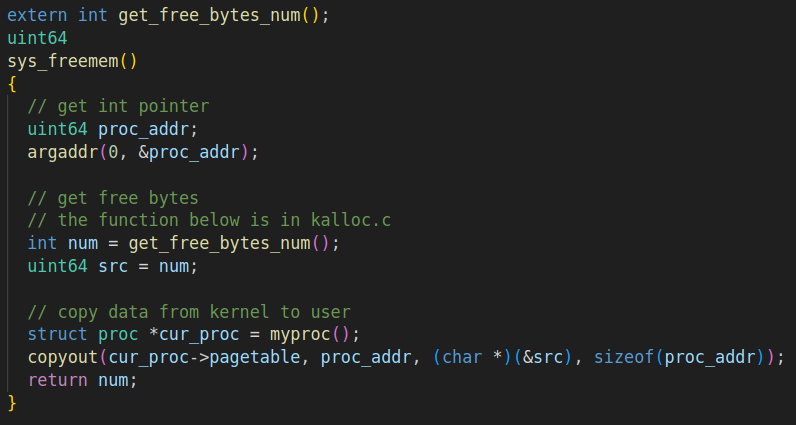
先观察kalloc.c文件的结构，发现内存是用kmem结构管理的，其中freelist记录了空闲的内存块，一个节点代表一页，每页的大小为4096 Bytes。这样只需获取链表的长度即可计算出空闲字节数。即//my function 所在的代码块。

要把统计到的值从kernel传到user，与实验一中的解决方法相同，使用argaddr()和copyout()函数，不再赘述。即//get int pointer 和 //copy data from kernel to user 所在的代码块。

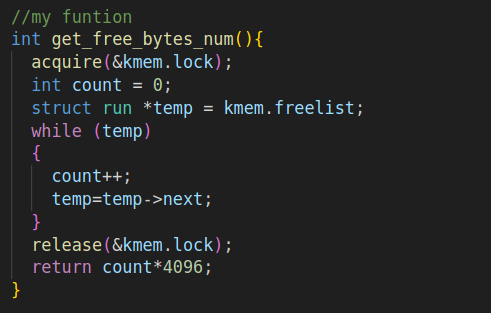
（2）实验过程

①实验代码：

**[kernel/sysproc.c]**

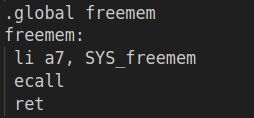


**[kernel/kalloc.c]**



②系统调用流程：(细节在实验一中已说明，下面将尽量简洁地阐述)

当在shell中输入freemem时，shell在其子进程中运行user/freemem.c中的代码，并在freemem.c中启动系统调用freemem(&num)。 之后执行汇编代码：



li指令将系统调用号保存在 a7 寄存器中。 然后调用ecall陷入内核，运行uservec保存用户寄存器状态到trapframe中，运行usertrap判断trap类型并处理。

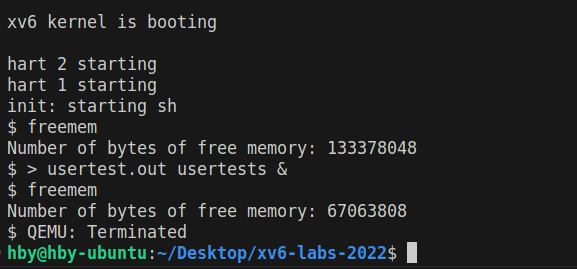
如果是系统调用的话，就将pc指向ecall的下一条指令，然后交给syscall函数处理。syscall获取 a7 寄存器中的系统调用号并执行相对应的系统调用，即sys\_freemem函数。

当syscall执行完成后，调用usertrapret()返回到用户空间，trap结束，至此系统调用的过程结束。

在上面的过程中系统调用的参数是通过uservec保存的trapframe来传递的。且如果要在内核空间中读写用户空间某个地址的数据，则要使用专用的copyin和copyout函数通过当前进程对应的页表来进行操作。

（3）实验结果

由于程序逻辑较为简单，可直接从逻辑上看出对错，故此处只作简单的验证。



可以看出在后台执行usertests后，空闲内存减少。

实验三：trace

（1）实验思路

由于exec不改变进程id，故可通过在当前proc结构中保存trace\_mask来使得之后exec的进程的proc中也有trace\_mask的值，即//add trace\_mask 对应的代码块。其中trace\_mask的值需要通过//get trace\_mask的代码块获得。这样就完成了proc中trace\_mask的标记。

之后在syscall中通过该标志来确定是否需要打印出trace信息。由于trace\_mask是通过对某个或者某些位进行标志来确定跟踪的系统调用的，故可以直接用 & 运算符来确认trace\_mask与 (1 << num) 的某些位是否有交集，如有的话则值不为0，打印trace信息。即//After syscall,trace it if num can match mask所在的代码块。

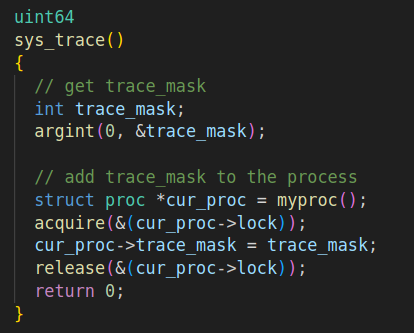
其中进程ID可直接通过p获得，系统调用名称通过定义的syscall\_str数组查询系统调用号获得。返回值通过trapframe->a0获得。

若要同时跟踪其子进程的系统调用，只需在fork函数中将trace\_mask同时复制给子进程的proc结构。即//copy trace\_mask to children 所在的代码块。

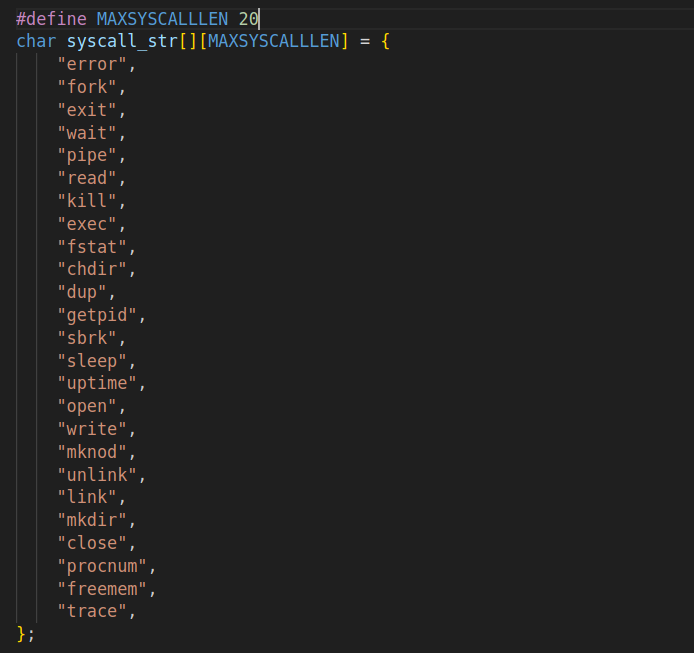
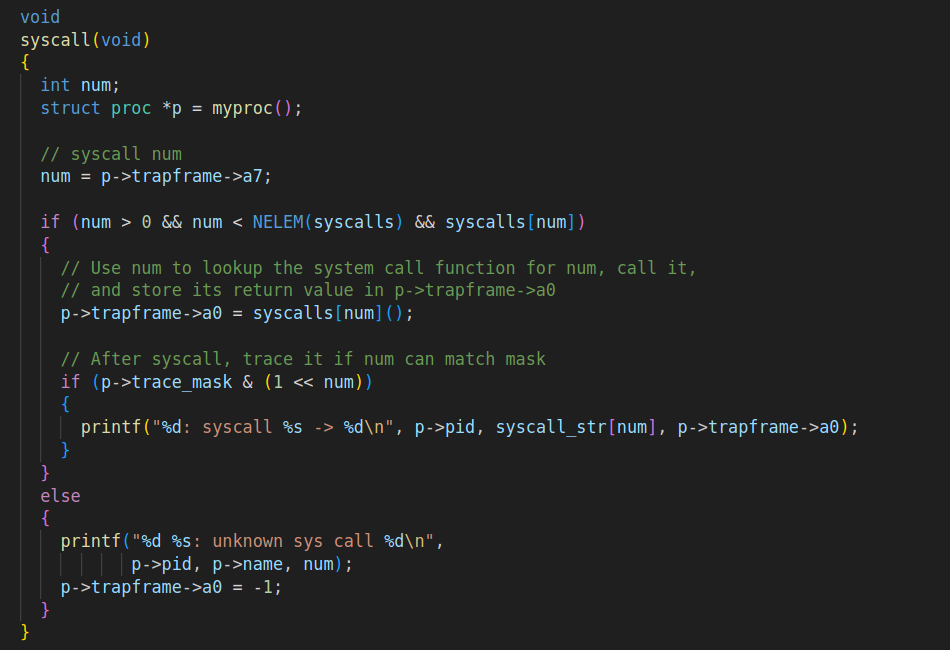
（2）实验过程

①实验代码：

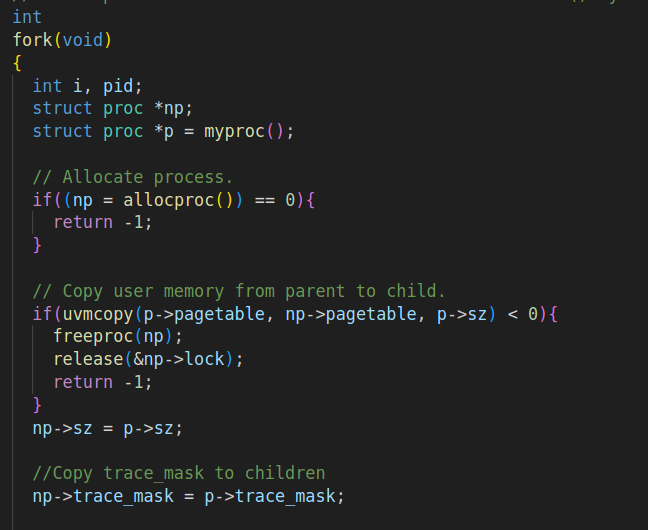
**[kernel/sysproc.c]**



**[kernel/syscall.c]**

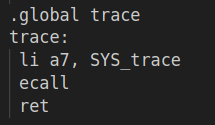


**[kernel/proc.c]**



②系统调用流程：

当在shell中输入trace命令时，shell在其子进程中运行user/trace.c中的代码，并在trace.c中启动系统调用trace(trace\_mask)。 之后执行汇编代码：

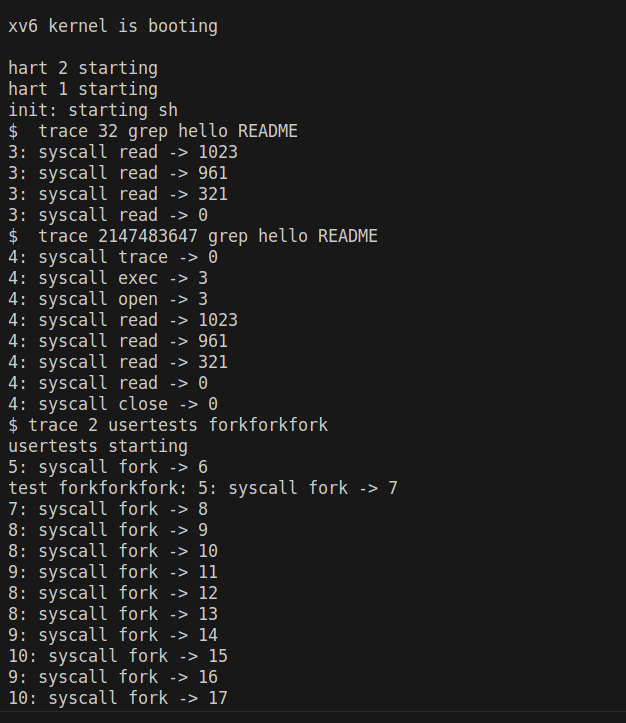


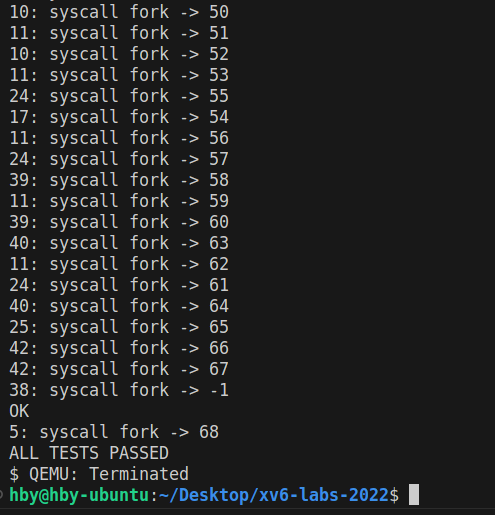
li指令将系统调用号保存在 a7 寄存器中。 然后调用ecall陷入内核，运行uservec保存用户寄存器状态到trapframe中，运行usertrap判断为系统调用，将pc指向ecall的下一条指令，然后交给syscall函数处理。syscall获取 a7 寄存器中的系统调用号并执行相对应的系统调用，即sys\_trace函数，该函数设置当前进程的trace\_mask。

当syscall执行完成后，调用usertrapret()返回到用户空间，trap结束，至此trace系统调用的过程结束。

之后返回用户空间继续执行 exec(nargv[0], nargv); 语句替换当前进程，但trace\_mask的状态依然保留，这样在当前进程中每次执行系统调用时若trace\_mask与系统调用号匹配就会打印trace信息。子进程由于复制了trace\_mask，故trace也会同时跟踪子进程。

（3）实验结果





实验四：流程概述

（1）根据以上xv6的系统调用经验，总结过程如下：

在用户空间内保存了系统调用的声明以及入口，当启动系统调用时，首先保存系统调用号到一个特定的寄存器中（xv6中为a7），之后陷入内核（xv6中通过执行ecall实现，在ecall时中保存了当前pc以从trap中返回），进入内核后首先保存用户寄存器的状态到一个特定的结构中（xv6中为trapframe），注意到在这里保存了系统调用的参数。然后识别trap类型并进行处理，如为系统调用，则交由系统调用处理程序执行，系统调用处理程序通过之前保存的系统调用号（xv6中为trapframe->a7）调用对应的系统调用，并将返回值保存在某处（xv6中为trapframe->a0）。最后设置trapframe，恢复页表，寄存器，并设置pc以从trap返回到用户空间（xv6中通过usertrapret()实现）。

【注：以下资料对理解trap有很大帮助。】

<https://www.cnblogs.com/weijunji/p/14338450.html>

<https://www.cnblogs.com/KatyuMarisaBlog/p/13934537.html>

<https://zhuanlan.zhihu.com/p/462538325>

（2）

一些机制：

1：每个进程有独立的虚拟地址空间，进程访问的是虚拟地址。

2：内存和磁盘中的空间以页为单位进行组织。

3：虚拟地址可通过每个进程上的页表与物理地址进行映射，获得真正物理地址。

4：当访问的虚拟地址对应的物理地址不在物理内存中，则产生缺页中断，分配物理内存。

5：malloc分配的只是虚拟内存，并未分配物理内存。

在malloc执行的过程中，当malloc的空间较小时（如小于128K），malloc函数调用brk系统调用，将\_edata指针（即堆指针）往高地址推对应的空间。之后进程访问该内存时，若发生缺页中断则分配物理内存，否则不会额外分配物理内存。在free时，brk分配的内存需要等到高地址内存释放以后才能释放，故如果高地址的内存未被free，该free内存块会形成一个内存碎片，\_edata指针也不会下移。而free时若发现free后最高地址空间的空闲内存超过一定空间（如128K），则执行内存紧缩操作，\_edata指针下移。

当malloc的空间较大时（如大于等于128K），malloc函数调用mmap（）系统调用来在堆和栈中间找一块空闲的虚拟内存。之后进程访问该内存时，若发生缺页中断则分配物理内存，否则不会额外分配物理内存。在free时，mmap（）系统分配的内存可以直接释放。

参考： <https://www.cnblogs.com/ssezhangpeng/p/10808969.html>

<https://www.cnblogs.com/dongzhiquan/p/5621906.html>