****



**MIT xv6实验报告**

**——lab2 : syscall**

**学生姓名 胡轶然**

**学 号 3019244355**

1. **实验目的**
2. 理解系统调用的工作原理。
3. 熟悉xv6系统内核代码。
4. 掌握xv6系统系统调用的实现方法。
5. **前期准备**
6. 切换git分支。
7. 阅读指导书第2章和第4章的第3、4小节。
8. 阅读kernel文件夹下的部分代码。
9. **实验内容及实现步骤**

**任务1 trace**

1. **问题描述**

本任务要求添加系统调用”trace”，实现对系统调用的跟踪，当系统执行被跟踪的系统调用后，会在控制台输出一条包含进程ID、系统调用的名称和返回值消息。为后续实验的调试提供帮助。

1. **思路与实现步骤**

将问题分为4部分分析，分别是：添加系统调用，实现对系统调用的追综，输出进程号、系统调用名称和返回值，子进程继承父进程的跟踪信息。

各步骤的具体思路与实现如下：

1. 添加sys\_trace系统调用，需修改MAKEFILE、user.h、syscall.h、usys.pl等文件。

具体内容包括：

* 1. 添加系统调用“sys\_trace”。
  2. 在makefile中添加trace项。
  3. 修改user/user.h，添加sys\_trace的原型。
  4. 新增宏定义，添加sys\_trace的编号。
  5. 在usys.pl中添加函数入口。

1. 实现对系统调用的追踪，**阅读下文发现**，题目要求使用独热编码(one-hot code)来存储要追踪的系统调用。独热编码可最大化数据存储效率，并实现对多个系统调用的同时追踪。
   1. 考虑到要同时追踪多个系统调用，需通过“或”操作将独热码聚集到一个uint型变量中，通过“与”操作判定某个调用是否被追踪。
   2. 设计系统调用与独热编码的映射关系，阅读题目发现，原程序已通过宏为系统调用做了编号，即题中所述的SYS\_fork（SYS\_\*\*\*）。

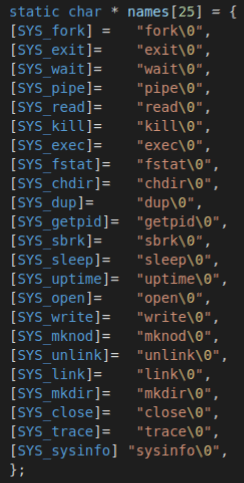
综上，在proc结构体中添加“uint trace\_mask;”变量，收集要追踪的系统调用独热编码。

在sysproc.c文件的sys\_trace函数，使用“或”运算将mask独热码加入trace\_mask中。

修改syscall函数，使用“与”操作判断某系统调用是否被追踪，并打印被追踪的系统调用的信息。

1. 输出进程号、系统调用名称和返回值。
   1. 阅读代码可知，进程号对应proc结构体中的pid元素；
   2. 要输出系统调用名称，需构建系统调用名和系统调用编号的对应关系，考虑到编号是1~23的整型数据，可构建一个字符串数组实现对应关系；
   3. 查阅文档和资料，根据RISCV的C规范，函数的返回值保存在a0寄存器中。

综上，在syscall.c文件中添加一个字符串数组。

****

在syscall.c文件的syscall函数中添加如下代码：



1. 根据题意，子进程要继承父进程的跟踪信息，因此需修改fork函数，令子进程复制父进程监测的所有系统调用。

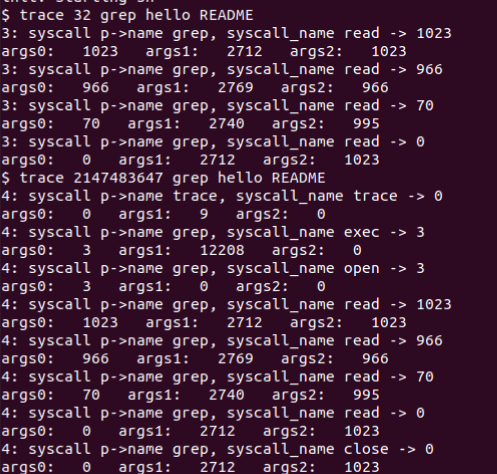
****

1. **问题与解决方法**

**困难一**：在阅读指导书时，不理解为什么要使用移位（“1 << SYS\_\*\*\*”）来表示系统调用。

**解决方法**：在查阅资料后回想起了上学期学过的独热编码。**这样编码可以同时监测最多32个系统调用(uint位数)，且无论要监测几个系统调用，都只需要传入1个uint型参数**。

**困难二**：在打印被追踪系统调用的名字时，错误地认为proc结构体中的name元素保存着系统调用的名字，但在运行过程中，发现proc->name元素的取值始终不变（下图偶数行第2列）。



**解决方法**

* 1. 调试发现proc->name存储的是命令行中输入的命令的名字，不会在执行过程中改变，**更不会在执行完某个系统调用后变为其函数名。**
  2. 阅读代码发现原代码中没有存储系统调用名称的数据结构，但每个系统调用都有一个1~23的整数作为编号，**可以定义一个字符串数组，使用系统调用的编号作为下标，在数组中查找其函数名**并输出。

**问题三**：不清楚怎样在系统调用函数中读取传入的参数。

**解决方法**：观察其他系统调用函数，发现可以**使用argint、argaddr等函数，将参数从a0~a7寄存器读取到不同类型的变量中**。

1. **结果**

见make grade

**任务2 sysinfotest**

1. **问题描述**

要求添加一个名为sysinfo的系统调用，实现两个功能：统计剩余空闲内存数量，统计进程数量。

1. **思路与实现步骤**

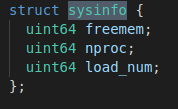
将问题分为5部分分析，分别是：添加系统调用，添加sysinfo结构体，统计剩余空闲内存，统计进程数和将sysinfo结构体从内核空间拷贝到进程空间。

1. 添加sys\_sysinfo系统调用，需修改MAKEFILE、user.h、syscall.h、usys.pl等文件。

具体内容包括：

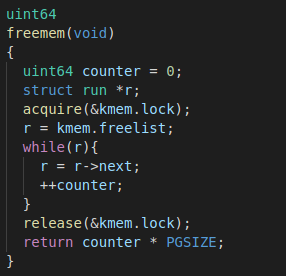
* 1. 添加系统调用“sys\_trace”。
  2. 在makefile中添加trace项。
  3. 修改user/user.h，添加sys\_trace的原型。
  4. 新增宏定义，添加sys\_trace的编号。
  5. 在usys.pl中添加函数入口。

1. 在sysinfo.h中添加sysinfo结构体，其属性有：进程数、剩余空间大小。（load\_num是可选任务的内容）



1. 统计剩余空闲内存，只需统计空闲内存链表的节点数量，再乘以页大小即可。

在kernel/defs.h添加声明，并在kalloc.c文件中添加函数freemem，具体实现如下。



1. 统计进程数，阅读代码可知，proc数组中存储了所有的进程。

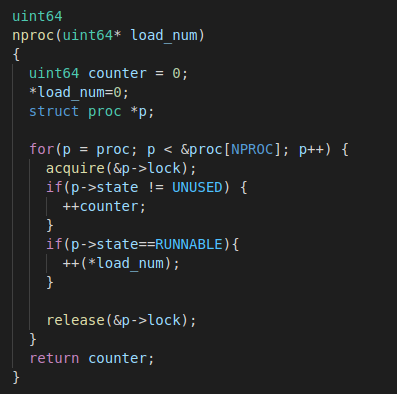
(即struct proc proc[NPROC];）

xv6使用枚举类型procstate标记进程的状态。

(即enum procstate { UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE })

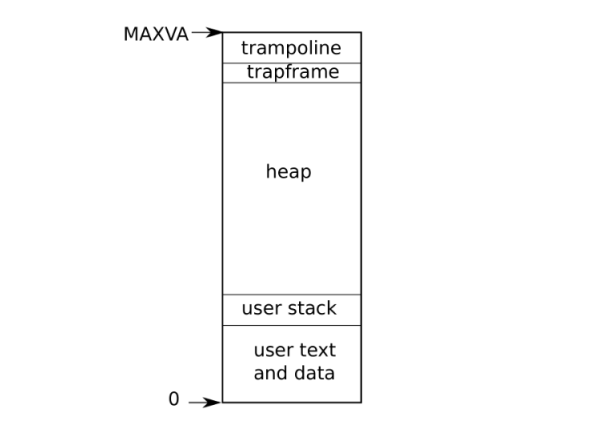
要统计进程数，只需遍历proc数组，统计UNUSED进程的数量即可。

在kernel/defs.h添加函数声明，并在proc.c文件中nproc添加函数。函数的具体实现如下：

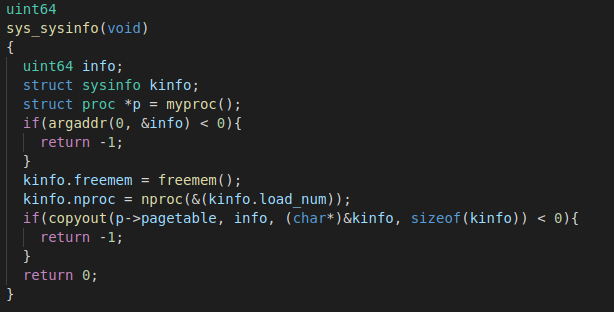


1. 在sysproc.c中实现系统调用sys\_sysinfo，主要内容有：
   1. 统计剩余空闲内存和进程数。
   2. 使用argaddr函数读取目标地址。
   3. 将sysinfo结构体从内核空间复制到进程空间中。

这里需用到copyout函数，其作用是将内核空间中的数据复制到进程空间中指定的**虚拟地址（如下图）**中。由于目标地址是虚拟地址，调用时需给出进程的页表；同时还要传入目标地址、原地址、数据长度等参数。



代码实现如下



1. **问题与解决方法**

**问题一**：不理解copyout函数的用途，以及为什么要传入进程页表作为参数。

**解决方法**：阅读sys\_fstat()和filestat() 函数，查阅资料发现，copyout函数可以将内核内存空间中的数据复制到进程内存空间的虚拟地址中。

在原版xv6系统中，所有进程共用一张内核页表。用户进程不能访问内核空间，也不能直接获取物理地址，所以要用**copyout函数将内核中的sysinfo结构体复制到进程可见的虚拟地址中。复制过程依赖于物理地址与虚拟地址的映射关系，所以需要提供进程页表。**

copyout函数的原型是：

copyout(pagetable\_t pagetable, uint64 dstva, char \*src, uint64 len)

其正常返回值为0，小于0的返回值表示发生异常。

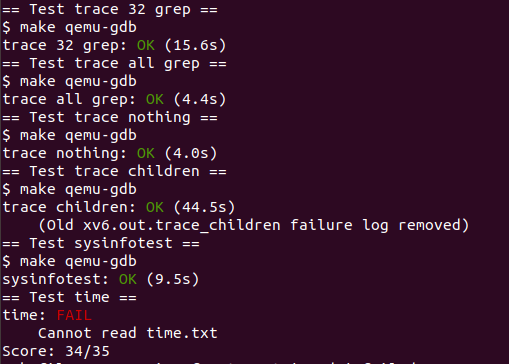
参数依次为：

1. 进程页表
2. 目标虚拟地址
3. 原地址
4. 要复制的数据长度
5. **结果**

见make grade测试。

1. **实验结果**

通过make grade测试



注：因个人虚拟机内存较小，运行较慢，故延长了测试程序的运行时间。此程序在其他虚拟机上可在规定时间内完成测试。

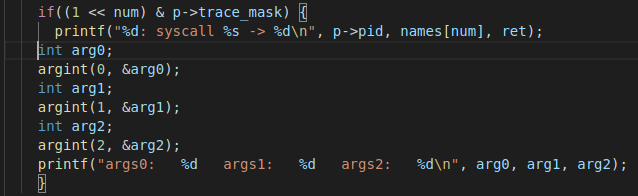
1. **Optional task**
2. **打印syscall参数**
   1. **思路**

上文已提到，xv6系统中使用寄存器a0~a7存储参数，考虑到大部分系统调用的参数不多于3个，所以这里只打印了a0~a2。

注:也可以构建一个数组，存储各系统调用的参数数量，并据此在追踪各系统调用时打印不同数量的参数，其实现方法同任务1的函数名数组，这里不再实现。

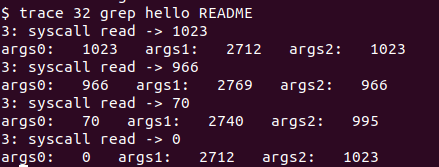
* 1. **实现方法与步骤**

在syscall中添加如下代码：



为防止影响正常测试，提交了两份代码，其中一份注释了optional task的代码。

* 1. **运行结果**



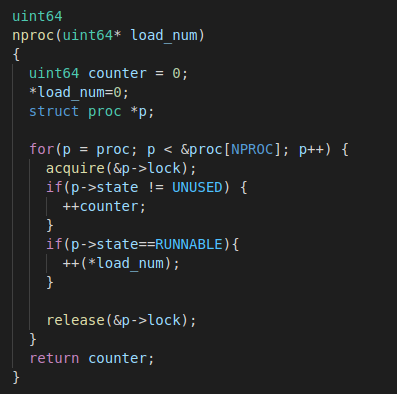
1. **average load**

**2.1思路**

查阅资料，average load的含义为**系统平均负载进程数**，**我将其理解为待执行(runnable)的进程数量**。

**2.2实现方法与步骤**

1. 在nproc函数中添加一个计数器(load\_num)，用于统计runnable进程数量。
2. 由于该函数已经有返回值，我添加了一个引用参数load\_num，并将结果存入其中。



注：仅统计了runnable的进程数量，**由于可选任务没有测试程序，无法判断个人理解是否正确。若要统计其他状态的进程数量，仅需对nproc函数中计数逻辑的触发条件做少量改动即可。**