# Lab: system calls

## 1.Using gdb ([easy](https://pdos.csail.mit.edu/6.828/2024/labs/guidance.html))

1. 实验目的

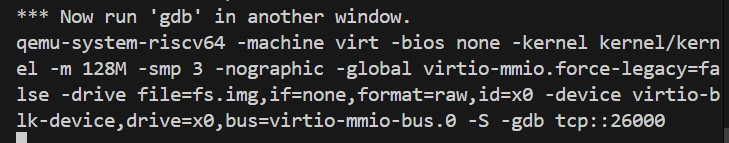
通过GDB单步调试xv6内核中的syscall执行流程，掌握：xv6中系统调用的实验路径，如何使用GDB调试RISC-V内核代码，如何查看函数调用栈、结构体和寄存器等信息，学习定位内核崩溃位置及其原因，掌握通过backtrace追踪进程执行流程。

1. 实验步骤

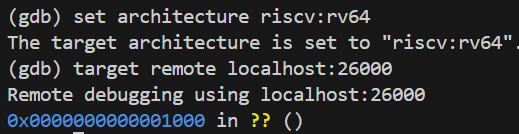
·前期准备

在两个终端A、B中分别调用make qemu-gdb指令、gdb-multiarch kernel/kernel指令来启动GDB调试服务、打开gdb调试器。

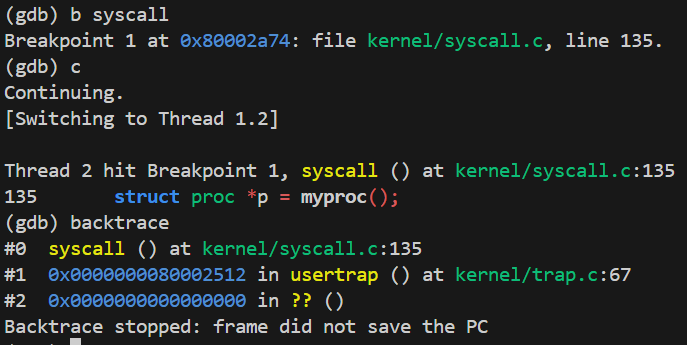
终端A：



终端B：

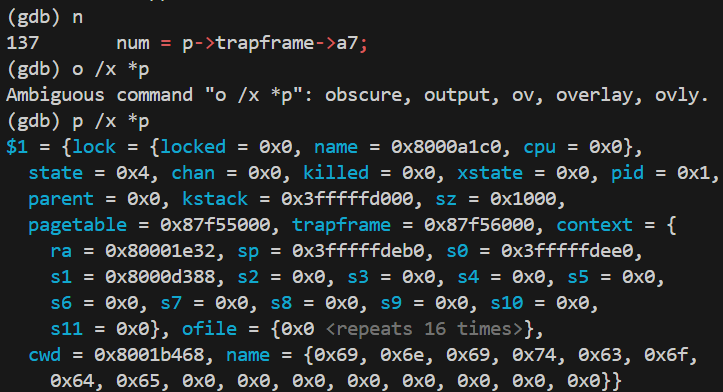


·在syscall函数处设置断点，用c继续运行，使用backtrace查看调用栈：



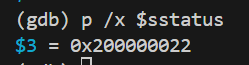
**分析（问题1）：**根据baktrace的输出，syscall()被usertrap()调用

·查看结构体与寄存器



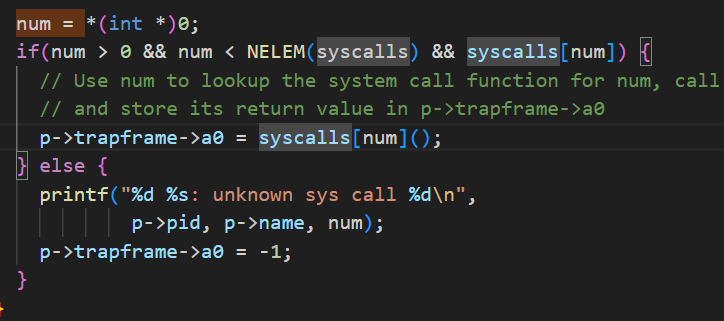
**分析（问题2）**：执行p /x p->trapframe->a7，看到其中的值为0x7，说明该存储器存储的系统调用号为SYS\_exec系统调用

·查看cpu之前的状态，执行：

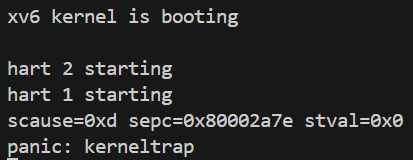


**分析（问题3）：**sstatus表示之前的状态，展开为二进制数：0010 0000 0000 0000 0000 0000 0000 0010 0010 。低第八位为0，说明CPU之前处于用户模式。Xv6的系统调用是在用户程序执行ecall进入到监督模式，因此sstatus记录ecall前的用户模式，符合预期。

·修改kernel/syscall.c文件中的num = p->trapframe->a7，替换为num=\*(int \*)0，故意访问地址0来触发页错误。

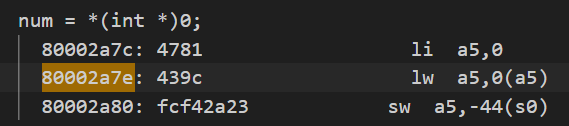


在终端重新执行make qemu，看到这样的输出：



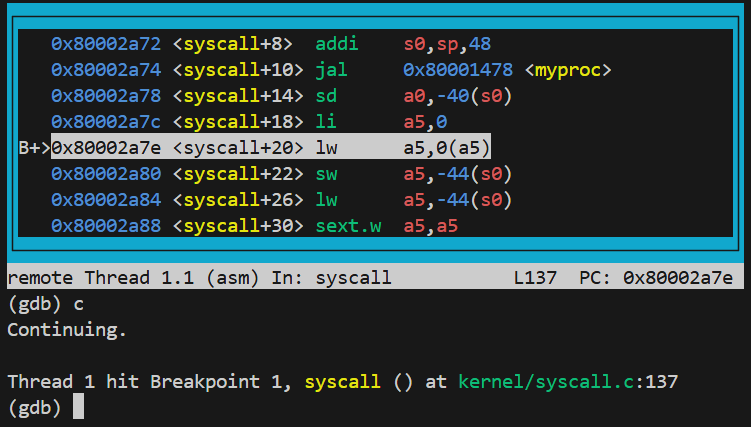
其中scaurse=0xd表示加载页错误，sepc表示出错的指令地址，stval=0x0是刚刚故意设置的非法地址。

·查找出错指令sepc。首先退出QEMU，在kernel/kernel.asm汇编文件中中搜索sepc的值（上面给出的0x80002a7e）看到：



**分析（问题4）：**触发错误的指令是lw a5,0(a5)，也就是load指令。因为上面一条指令li已经把0赋值到寄存器a5中，所以该条指令试图从地址0出加载数据，因此出错。从最后一条store指令来看，num对应的储存位置在栈帧中，为-44(s0)。

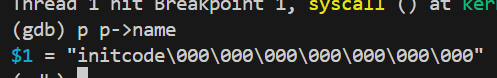
·重新打开gdb调试窗口（步骤与前面相似），并在0x80002a7e处设置断点，通过layout asm指令来开辟窗口观察汇编文件。c运行到断点，就可以看到出现异常时汇编文件的状况了：

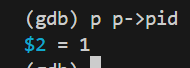


窗口中显示的异常指令和我们之前看到的一样。

**分析（问题5）：**内核中断的原因是0x0到0x1000是未被映射到用户空间或内核空间的保护区域（用于捕获空指针的访问），因此对这个区域的任何访问都会触发页错误。scause值为0xd表示加载页错误（若为0xf则是储存页错误），stval=0x0记录了非法访问的地址，这两个值共同确认了内核因为访问未映射的地址0而崩溃。

·输入p p->name和p p->pid查看内核崩溃时哪一个用户级进程正在运行，以及它的pid是多少：

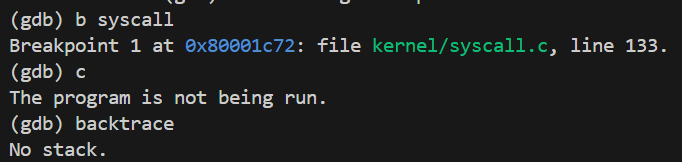




**分析（问题6）**：在内核崩溃时运行的用户级程序名为initcode，pid=1

1. 实验中遇到的问题和解决办法

·gdb-multiarch方式自动连接，必须手动指定所需要连接的gdb服务。一开始没有指定时运行断点指令发现：

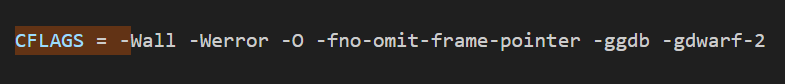


后来使用target remote设置目标服务器之后成功。

·在尝试使用gdb指令进行结构体输出时发现：  
(gdb) p /x \*p

value has been optimized out

这是由于变量p被优化器优化掉了。在Makefile中将找到的这一行进行修改：



将其中的-o改为-o0，取消优化，使得所有变量都保留方便调试。

重新启动make qemu-gdb，成功看到了结构体信息。

### · 最后输出的进程名中除了initcode还出现了乱码，我们输入x/16bx &p->name输出，最终看到了屏幕截图 2025-07-11 212221

前连个字节为initcode，后面是结束符。这样进一步印证了进程名。

1. 实验心得

本次实验学到如何使用ecall指令触发用户态到内核态的切换，以及如何使用gdb调试工具定位和判断内核中断的位置和原因，接触了反汇编代码的分析。进一步加深了对理论课中用户态通过自陷指令进入内核态、进而实现系统调用这一过程的理解。再次认识到对用户空间和内核空间的权限控制，以及非法访问的保护。

## System call tracing ([moderate](https://pdos.csail.mit.edu/6.828/2024/labs/guidance.html))

1. 实验目的

本实验要实现系统调用的追踪功能。添加trace系统调用，通过掩码控制需要追踪的系统调用类型，返回PID、系统调用名称和返回值，并将追踪效果传递给子进程（这一过程通过fork继承实现）。

1. 实验步骤

·添加用户态的trace调用支持

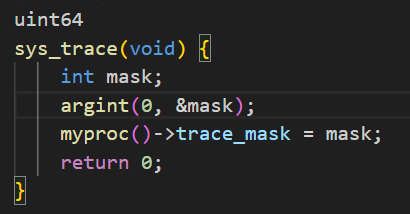
修改Makefile，在UPROGS中添加$U/\_trace

修改user/user.h在其中添加函数原型的声明int trace(int mask);

在user/usys.pl中添加stub，在kernal/syscall.h中添加系统调用号#define SYS\_trace 22（顺延）  
·内核态中实现

修改进程结构体，在kernal/proc.h的struct proc中添加trace\_mask这一记录追踪的掩码

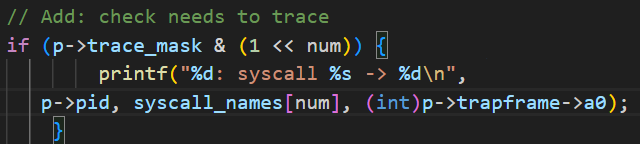
实现sys\_trace()，在kernel/sysproc.c中添加（LINE 30）：



再修改fork()继承掩码，在kernel/proc.c中的fork()里赋值掩码：

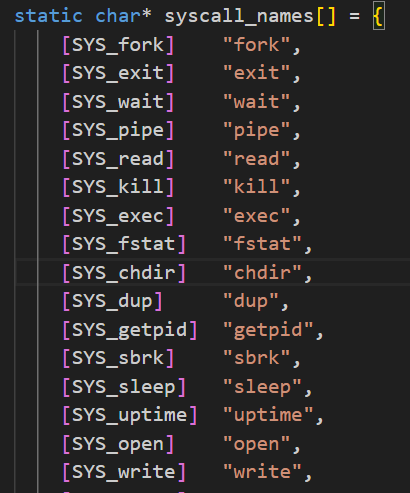
np->trace\_mask = p->trace\_mask; （LINE 312）

修改syscall()打印追踪信息，在kernel/syscall.c的syscall()函数的末尾添加打印逻辑（LINE 142）：



同时在syscall\_names数组中加入 [SYS\_trace] sys\_ trace,（LINE 155）

而且还要添加一个系统调用名称数组（LINE 10）：

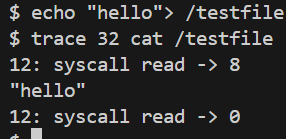


并且要新增声明：extern uint64 sys\_trace(void); （LINE129）

·测试验证，首先运行make qemu

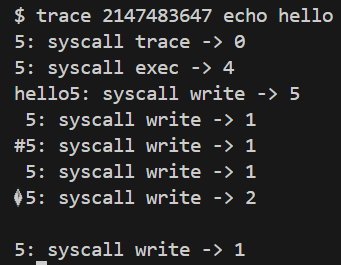
追踪read系统调用：

这里自己创建了一个小文件进行测试。掩码设置为32是因为1<<SYS\_READ对应32



**分析：**输出为读取文件中所执行的所有READ系统调用（读取和文件结束），不包括其他调用（中间夹杂了一个cat本身输出的hello，后面也一样），掩码发挥了作用。

追踪所有的系统调用：



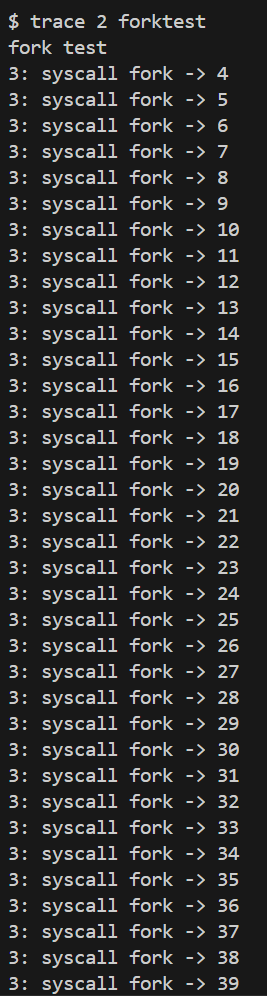
**分析**：这里将掩码设置为底31为全部为1，捕获全部的调用，可以看到包括trace自身、exec、多个write指令全部被捕获，符合预期。

不追踪任何系统调用：



**分析：**只有指令输出的结果hello，没有任何系统调用信息

追踪层级调用（我们用到了系统自带的forktest程序，掩码为2对应fork的系统调用号）：



**分析：**从输出来看，所有的fork调用都被正确追踪，所展示的层级调用过程也十分完整（如子进程PID=4继续调用fork时，生成了PID=5的孙进程）符合forktest的预期行为

1. 实验中遇到的问题和解决办法

添加一个新的系统调用trace最大的麻烦在于需要理解xv6内核中的系统调用源文件的结构，找到在哪里实现，并且要在每个需要声明的地方都进行附加声明，要保证格式的一致性。最初没有添加extern声明导致sycall.c文件无法找到sys\_trace这一在kernel/sysproc.c中实现的函数，经过一番分析才发现需要加入该声明。

于此同时，局域trace功能的特性，我们需要将系统调用的编号映射到人类可以读取的名称，所以不厌其烦地加入了一个新定义的syscall\_names数组，从而能够在输出中清晰地标识出每个系统调用。

还遇到了一些编译器对于默认类型转换warning不允许的情况（在实现打印逻辑时），因此我们加入了强制类型转换。

在测试时一开始尝试打开系统已有的文件但发现太大了：  


因此选择自己手动创建一些小文件进行read追踪的测试。

1. 实验心得

通过实现trace这一新的系统调用，深入理解了系统调用在操作系统源代码中的实现方式与运行逻辑，加强了编程过程中先研究原有代码的文件结构在进行仿照添加的方法意识。

通过观察trace得到的4类系统调用，理解了每一条命令号代码是如何被拆解成多个不同类型的系统调用来实现的，对操作系统的分册设计架构和硬件保护策略有了更深刻的认识。

## 3.Attack xv6 ([moderate](https://pdos.csail.mit.edu/6.828/2024/labs/guidance.html))

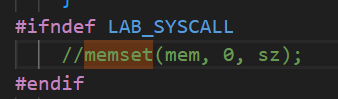
1）实验目的

操作系统本应该把不同的用户级进程以及用户级、内核级进程相互分隔开，以确保数据安全。但本实验利用在系统调用代码中的已经被设计好的一个bug，编写相对应的恶意程序实现对xv6的攻击，从一个用户级进程中查看到另一进程中写入的密码。

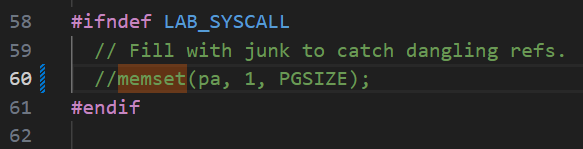
1. 实验步骤

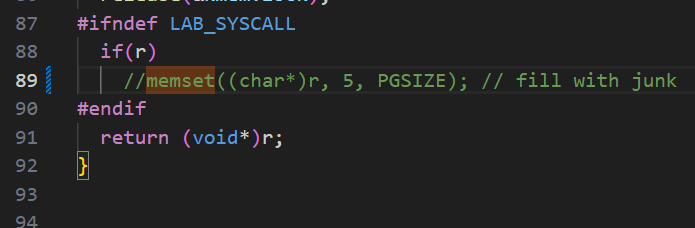
·按照实验的要求在系统文件中设置bug:

注释kernel/vm.c LINE 272



注释kernel/kalloc.c，行数已经显示





·理解系统调用代码中的现存bug

如实验指导中所述，该bug的核心是新分配的内存页没有清零导致敏感数据的泄露。漏洞的代码位置是kervel/vm.c中uvmalloc()省略了memset(mem,0,sz)，同时kervel/kalloc.c中省略了释放页时的memeset。这样新分配的页中会有残留的数据，而且能够被复用，这就造成了数据泄露的风险。

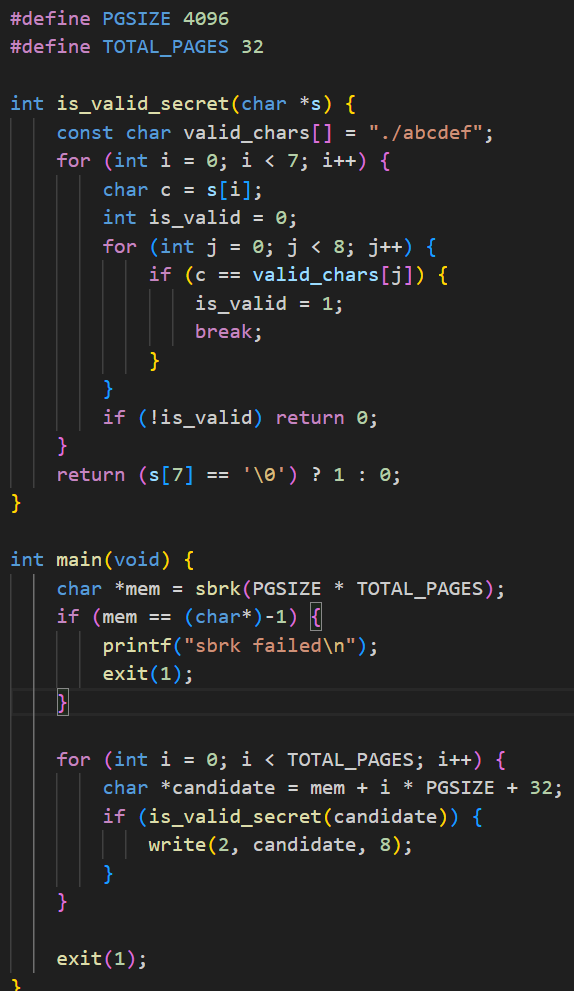
·利用漏洞的策略

首先分析user/secret.c的进程行为，以便攻击：它将申请了32页的内存，8字节的秘钥写入后退出，但没有擦除，并且秘钥的存储位置在分配的第10个页面的+32字节处。因此我们只需要让attack.c分配大小相同的内存（32页），就会高概率复用刚刚被释放的内存（或者至少包含写入密码的那一页）。遍历分配到的所有页，查找+32页内偏移量，只要发现后面的8个字节的格式符合密码的格式，就将该8个字节破译出的密码提交。

·编写user/attack.c

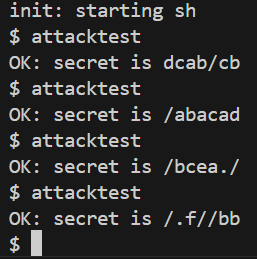
根据上述利用策略，编写代码如下图（LINE 5）。

需要说明：在分析了attacktest.c之后，发现生成的代码都是从{a,b,c,d,e,f,.,/}这8个字符中选取字符组成前面7位，最后一位用尾零填充得到的，因此我们在attack.c中筛选得到的是否为密码时严格按照这个标准来做，从而定义了函数is\_valid\_secret。



·测试程序user/attacktest.c

在shell模式下运行attacktest.c：



**分析：**输出的密码符合7位有效字符且在{a,b,c,d,e,f,.,/}中选取的规格，而且每次都能成功破译，实验成功。

1. 实验中遇到的问题和解决办法

本实验的关键之处在于理解分页式内存管理中偏移量的特点和逻辑页号与物理页号（也称“页框号”）的区别。最开始我机械地按照secret.c中写入代码的逻辑页号9去寻找密码，结果自然是义务所获：**因为每个进程的页表都是不一样的，从页号到页框号的映射是不同的，但是密码信息在页中的偏移量是固定的**。在secret.c中写入密码的物理页框号映射到的页号是9，但在attack.c进程的页表中，即使我们获得了同一个物理页，它的页号只有极小概率仍然是9。因此我们采用（固定偏移量+遍历页号+格式筛选）的方式来确定。因为其他物理页在这个偏移量恰好也写了一个这样的密码的概率很小，我们的代码是有效的。如果把偏移量改掉（32->0，或者偏移量未知），攻击程序自然就无效了**（问题回答）**。

4）实验心得

本实验使我更加深刻地认识到操作系统不同进程之间资源与信息相互隔离的特性，以及页式内存管理和再分配的特点（**特别是深刻体会到物理页框号和页号的区别：二者通过页表来映射，每个程序的页表都是独特的**）。理解了系统调用指令对于隔离软硬件的重大作用，系统调用一旦出现问题，将会导致硬件信息可能暴漏在恶意攻击之下，造成隐私泄露的风险。