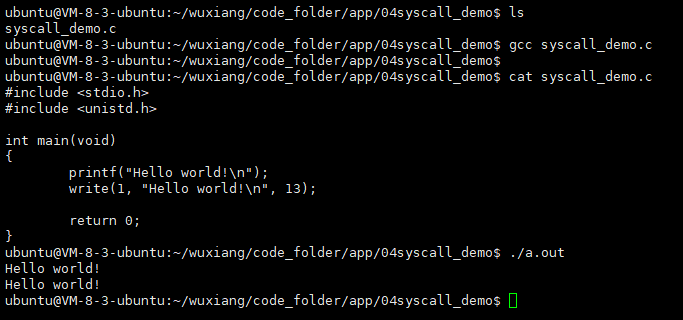
# 系统调用基本概念

## 1.1 一个系统调用的例子



## 1.2 什么是系统调用？软件复用的角度

main.c -- add() -- add.c add.h add.o ==> ar ==> [libmath.so](https://libmath.so) / libmath.a

main.c -- sub() -- sub.c sub.h sub.o ==> ar ==> [libmath.so](https://libmath.so) / libmath.a

glibc -- 默认链接

OS：硬件资源的封装，任务创建、调度，读写磁盘等

如ucos：OSInit -- OSTaskCreate -- 创建一个任务

如Linux：手机、电脑，底层的硬件、OS、上层App

权限管理：内核态、用户态

系统调用：App -- 陷入内核态 -- 访问硬件设备

# 软中断：系统调用的入口

## 2.1 权限管理

* 程序的用户态、内核态
* 操作系统 + CPU软中断：swi/svc
* CPU的运行级别：普通权限 & 特权

-- ARM32：

普通模式：User

特权模式：FIQ、IRQ、SVC、ABT、UND

-- ARM64：EL0、EL1、EL2、EL3

-- X86：ring0 ~ ring3

## 2.2 系统调用号

* 系统调用接口：read、write
* 内核中的实现：sys\_read、sys\_write
* 系统调用号：

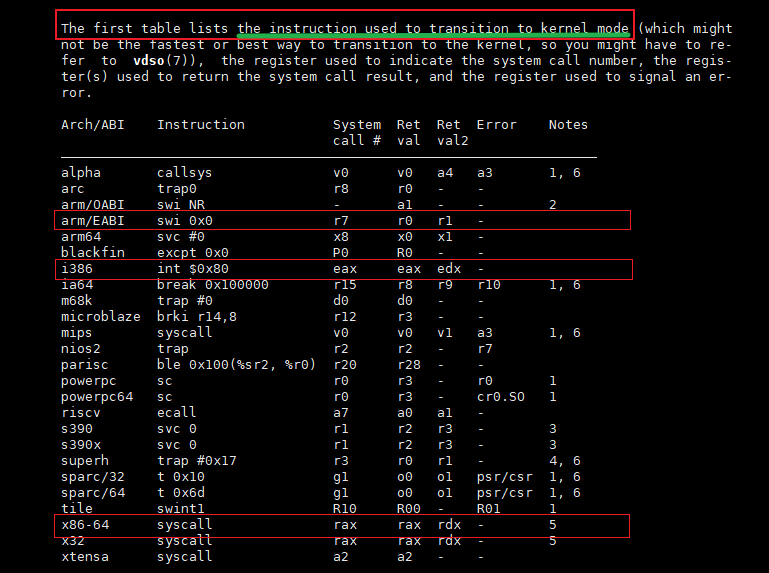
-- 32位ARM：3、4

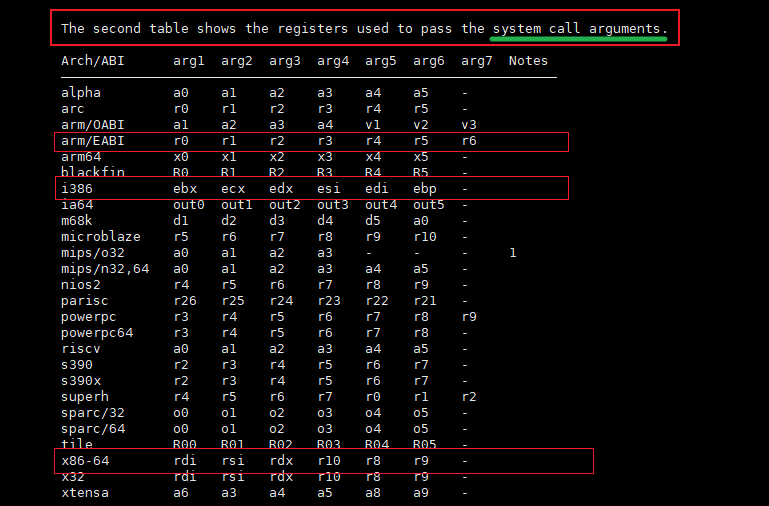
-- 64位ARM：0、1

-- 32位X86：3、4

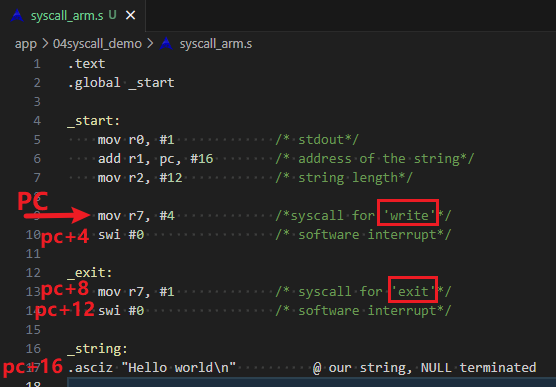
-- 64位X86：0、1

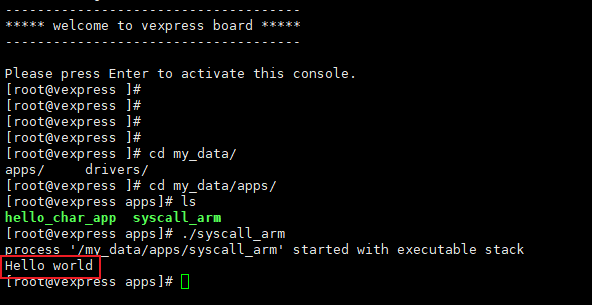
## 2.4 man 2 syscall



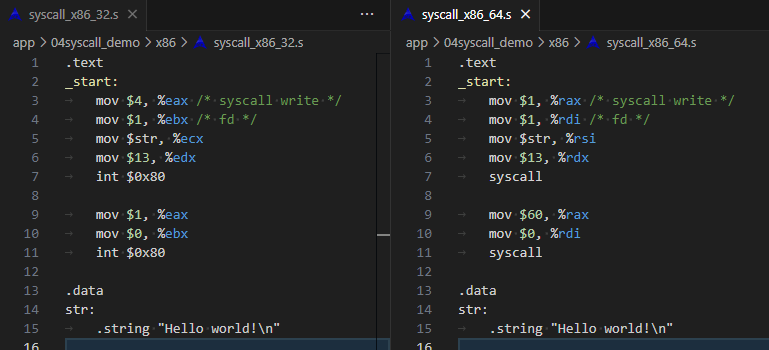


## 2.5 实验：arm系统调用

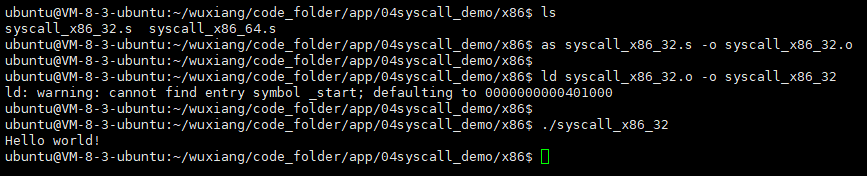




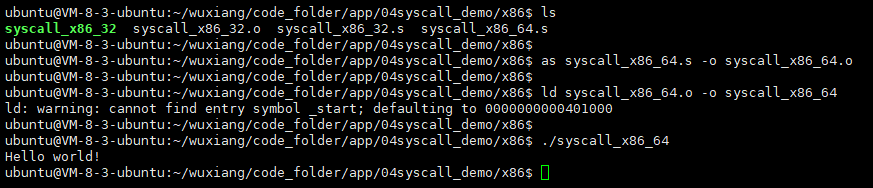
## 2.6 实验：x86系统调用



* x86\_32



* x86\_64



# 系统调用接口的封装

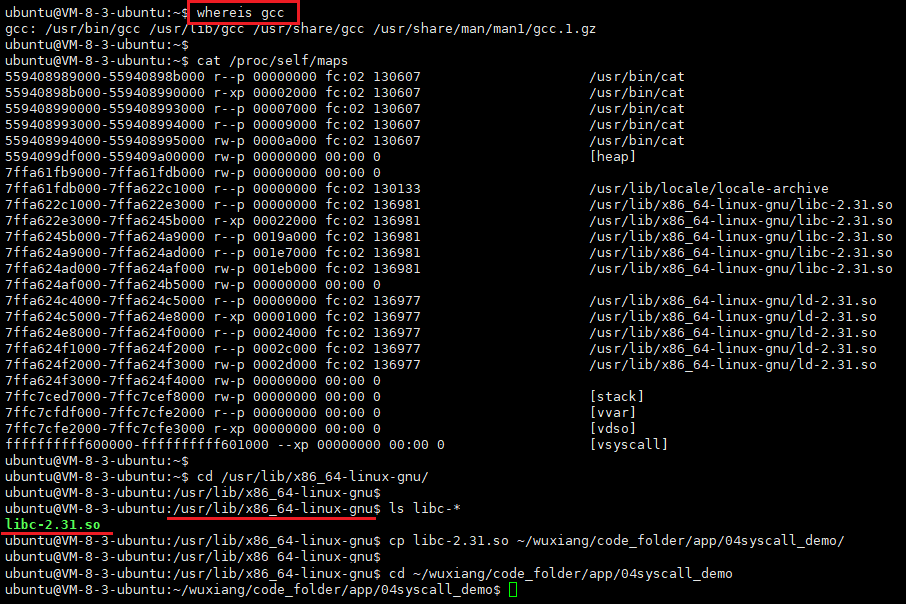
## 3.1 glibc

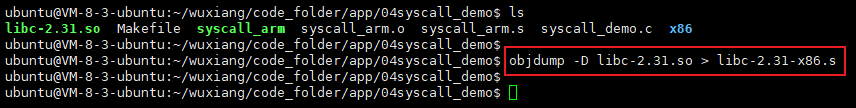
* C标准库

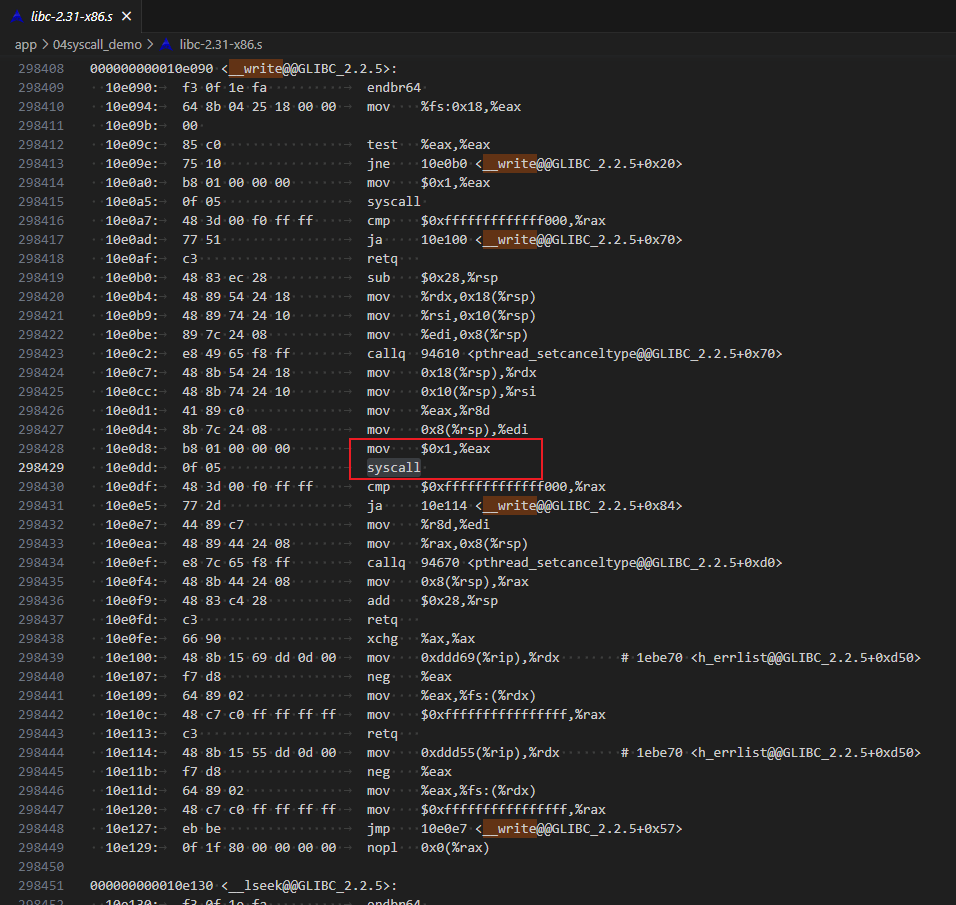
包含一系列系统调用接口的封装

-- read、write、fork、open ......

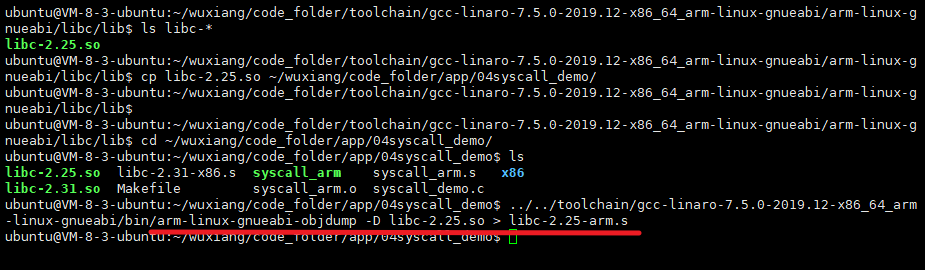
* x86平台

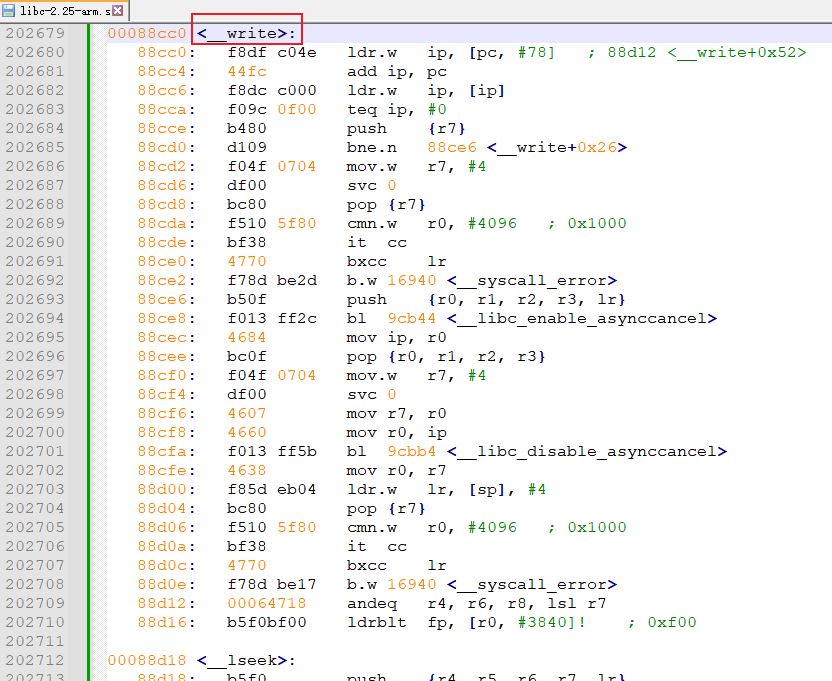






* arm平台





注：\*.s 文件太大，不放到git仓库中了（通过.gitignore）

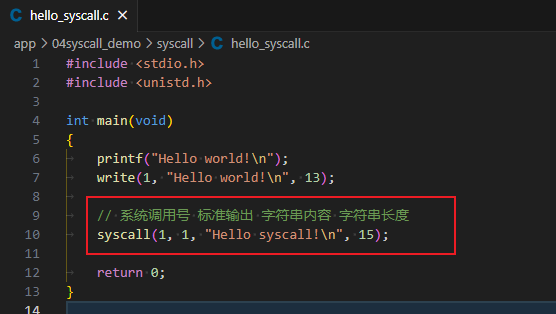
## 3.2 syscall

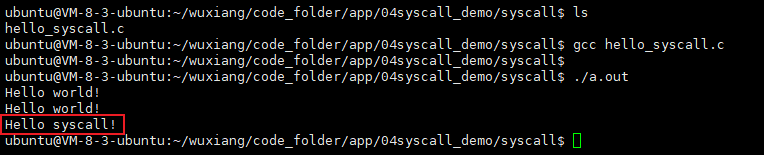
* 在C标准库中没有封装的系统调用
* syscall是一个库函数
* syscall封装了系统调用的汇编接口

-- 系统调用前保存CPU寄存器

-- 从系统调用返回后，恢复寄存器

### 3.2.1 x86\_64下



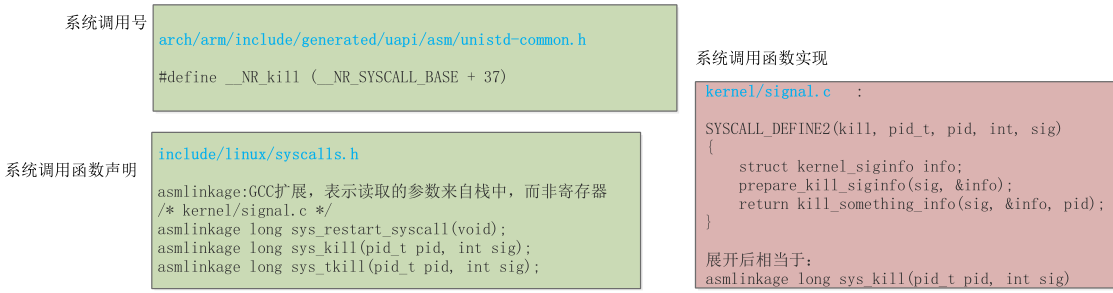


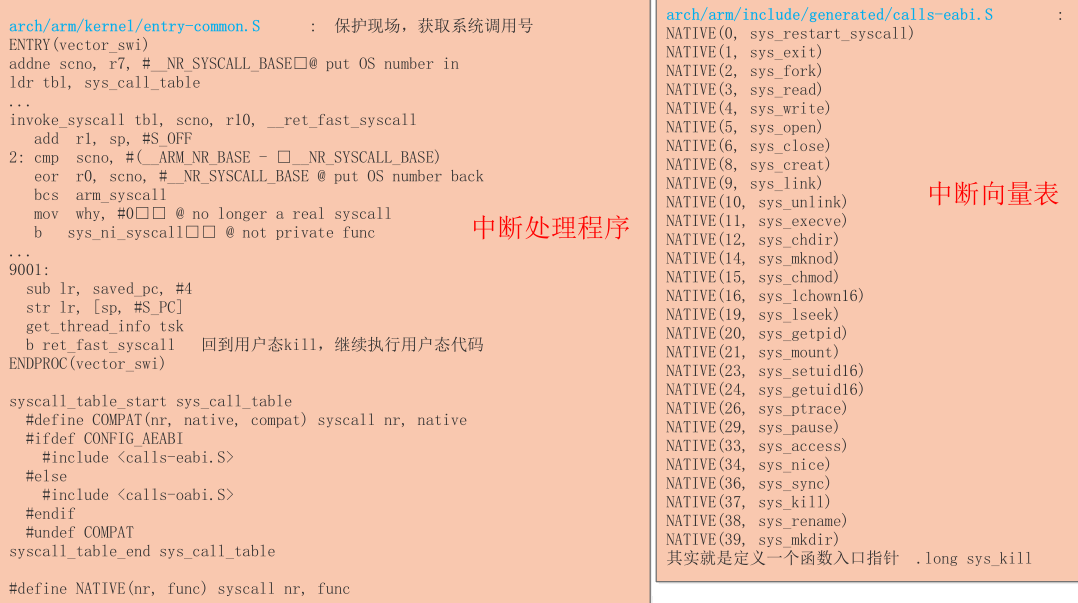
### 3.2.2 arm下

arm平台下，修改对应的系统调用号为4，同样可以运行打印结果

# 系统调用流程分析

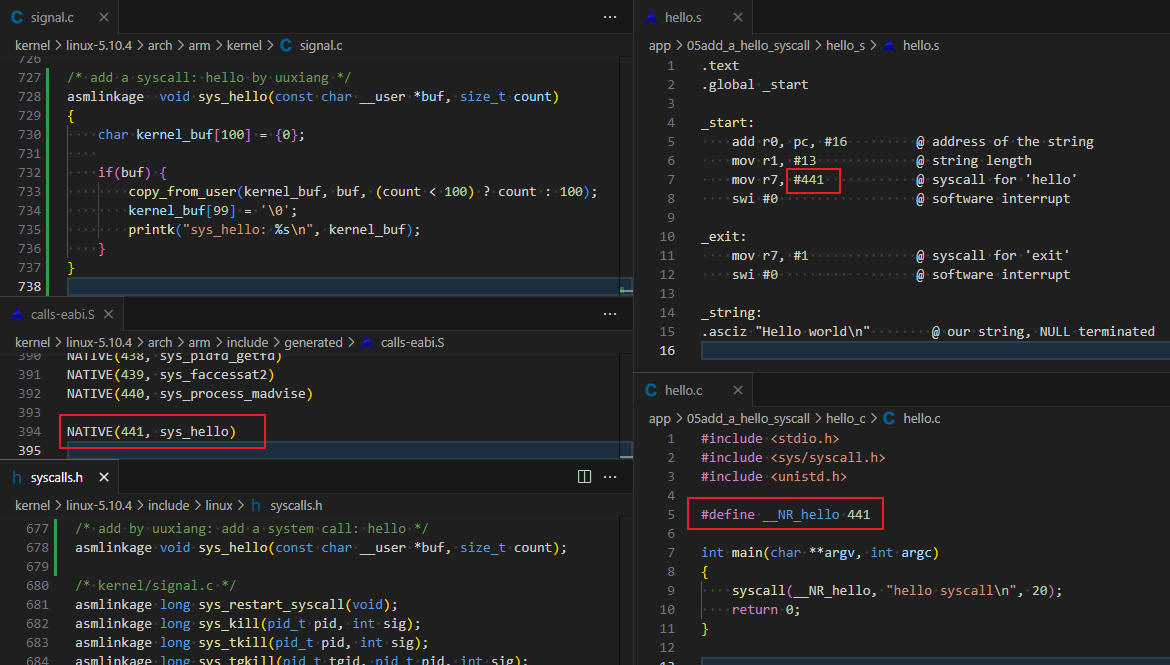
* 示例：kill
* 接口封装： /usr/arm-linux-gnueabi/lib/libc.a
* 系统调用号： arch/arm/include/generated/calls-eabi.S
* 内核实现： kernel/signal.c
* 中断处理： arch/arm/kernel/entry-common.S

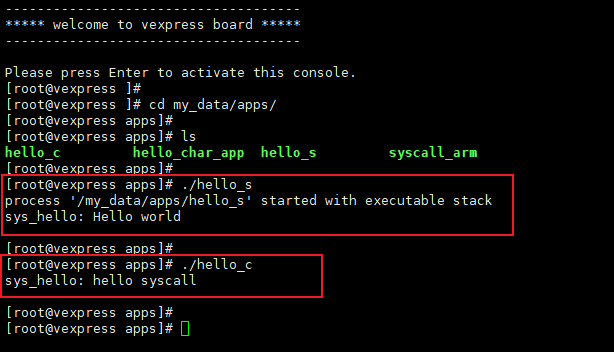




# 添加一个系统调用

1. 增加内核对应的实现函数
2. 在系统调用表中增加一个系统调用号及入口
3. 编写测试程序：在应用层发起系统调用





# 系统调用的开销

## 6.1 主要开销

-- 中断

-- 抢占系统、任务调度

-- 同步

-- IO等待

## 6.2 解决之道

-- 快速系统调用：fast system call

-- 虚拟系统调用：vsyscall

-- VDSO

### 6.2.1 快速系统调用

* x86处理器

-- 专门为系统调用设计的汇编指令

Intel：sysenter、sysexit

AMD：syscall、sysret

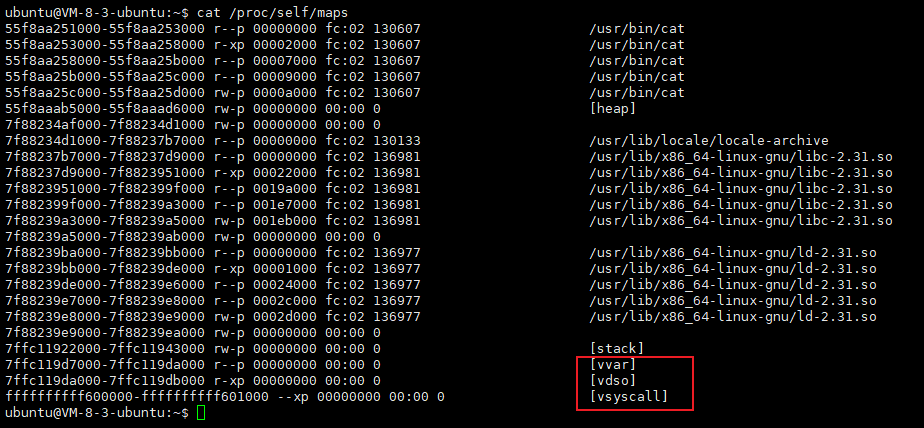
-- 简化了系统调用和返回流程

-- 预加载参数、不做权限检查

-- 不再查表，直接从寄存器取值，实现快速跳转：MSR寄存器

-- 不需要保存地址和返回地址等信息

### 6.2.2 虚拟系统调用：vsyscall



proc/<PID>/maps

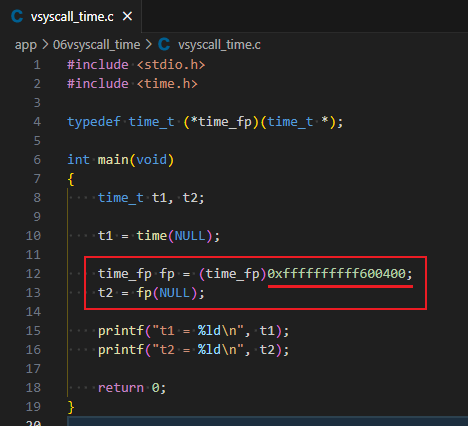
查看**进程的虚拟地址空间**是如何使用的。

* 编程实验：获取当前系统时间

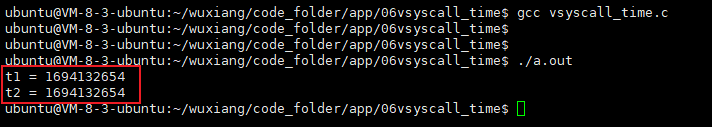
1. 使用系统调用time获取当前时间
2. 使用虚拟系统调用接口获取当前时间
3. 比较结果并分析

原先：app -- time -- int 80h / syscall -- kernel -- sys\_time -- user space

现在：app -- time -- vsyscall -- user space （效率更高）



可以下载glibc-2.22之前的版本，搜索：grep -nr VSYSCALL\_ADDR\_vtime 查看图中的地址。



### 6.2.3 虚拟动态共享对象：VDSO

* vsyscall的局限性

-- 分配的内存有限

-- 只支持4个系统调用

-- 在进程中的位置是静态的、固定的，易受攻击

* VDSO的改进

-- 提供超过4个系统调用

-- 在进程中的地址是随机的: 如：time

#### 6.2.3.1 VDSO：virtual dynamic shared object

-- # cat /proc/self/maps

-- 源码在**内核中**实现

arch/arm/kernel/vdso.c

关键函数：vdso\_mremap、install\_vvar

-- 速度最快

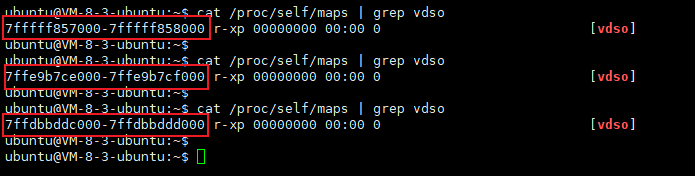
-- 开销最小，基本等价于函数调用开销

#### 6.2.3.2 VVAR：VDSO data page

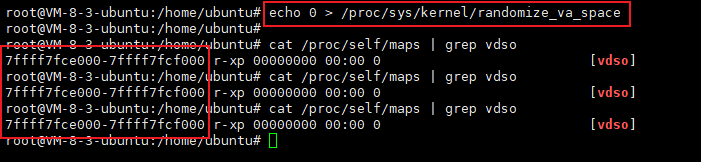
-- struct vdso\_data

#### 6.2.3.3 实验：反汇编VDSO动态库

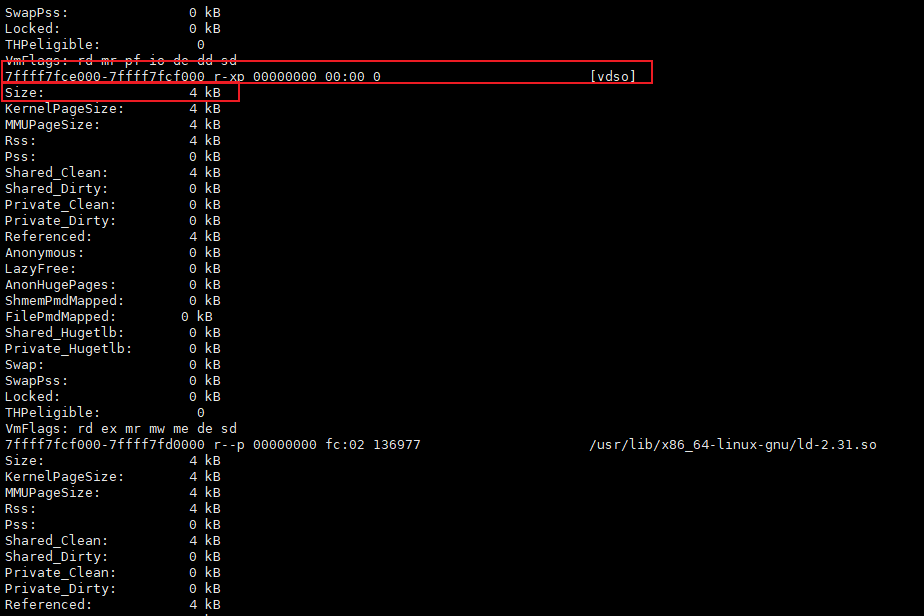
1. 将VDSO指令从内存中dump出来
2. 反汇编为汇编代码
3. 分析汇编代码



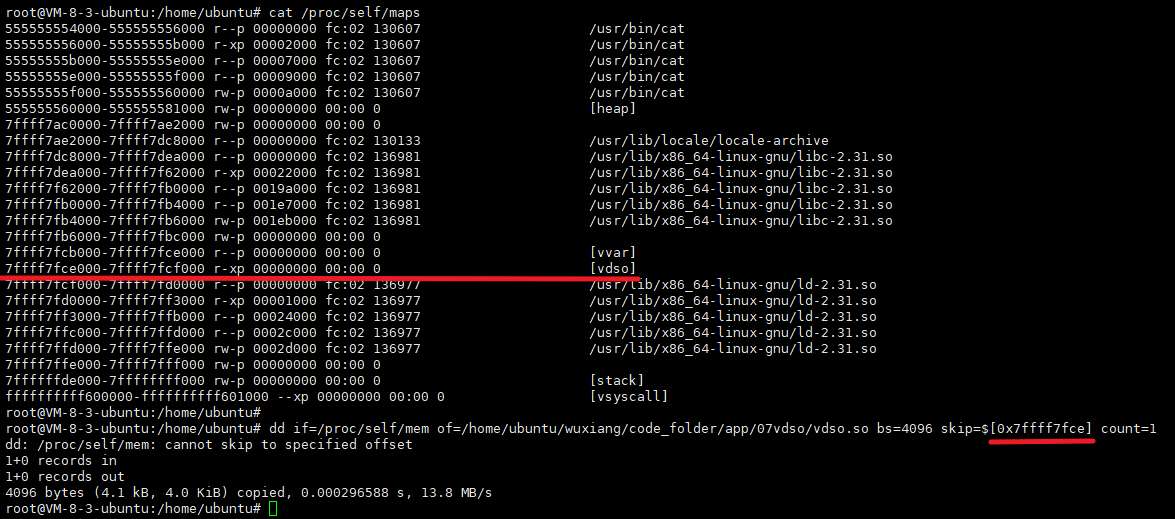
可以关闭随机地址功能：echo 0 > /proc/sys/kernel/randomize\_va\_space



可以通过：cat /proc/self/smaps 查看vdso大小

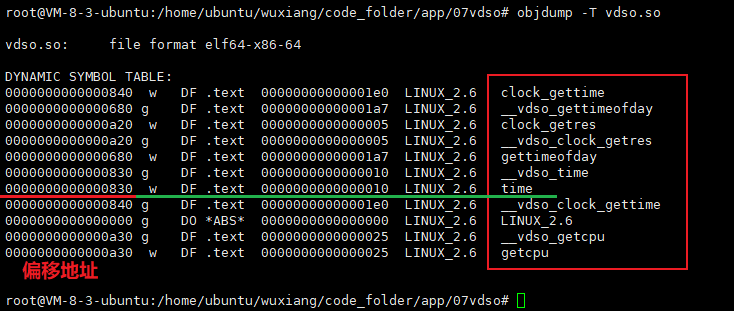


1. 将VDSO指令从内存中dump出来



1. 反汇编查看：

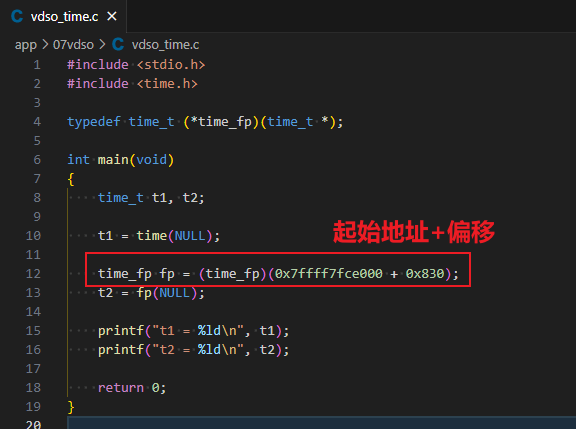
objdump -T 显示文件的动态符号表入口，仅仅对动态目标文件有意义，比如某些共享库

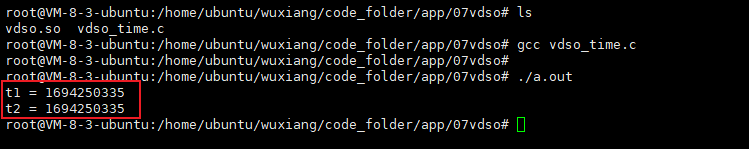


起始地址：0x7ffff7fce000

偏移：0x830

1. 分析





# 文件的读写流程

