**实验四 增加Linux系统调用**

实验日期：9月30日 地点：求是楼207

1. **实验目的**

学习如何产生一个系统调用以及怎样同过往内核中增加一个新函数从而在内核空间中实现对用户空间的读/写。

1. **实验内容**
2. 添加系统调用
   1. 修改系统调用表：
      1. 在arch/x86/entry/syscalls/syscall\_64.tbl文件中添加新的系统调用号和名称。
   2. 声明与实现系统调用：
      1. 在include/linux/syscalls.h中声明新的系统调用号。
      2. 在kernel/sys.c中实现新的系统调用函数。
3. 编译内核
   1. 编译内核：
      1. 在内核源码目录中，执行make -jN（N为CPU核心数）开始编译内核。
      2. 编译完成后，会生成vmlinux、bzImage等内核文件。
   2. 安装内核：
      1. 使用make modules\_install安装内核模块
      2. 使用make install或手动复制内核文件到/boot目录，并更新GRUB引导配置。
4. 测试新的系统调用
   1. 重启系统：重启计算机，并在启动时选择新编译的内核。
   2. 编写测试程序：编写一个简单的C程序，使用syscall函数
   3. 编译与运行测试程序：
      1. 使用gcc编译测试程序。
      2. 以root权限运行测试程序，并观察输出结果。
   4. 验证：dmesg命令查看内核日志，确认新的系统调用是否已成功执行并返回了预期的结果。
5. **实验过程记录**
6. 添加系统调用

首先需要以root进入Linux内核源码的根目录，这是所有后续操作的基础，因为后续对系统调用表的修改、系统调用的声明与实现以及内核的编译等操作都要在内核源码环境中进行。



图4- 1 进入内核目录

系统调用表维护了系统调用号与相应内核函数之间的映射关系。当用户空间程序发起一个系统调用时，内核会根据系统调用号查找系统调用表，以确定要执行的内核函数。因此，需要在arch/x86/entry/syscalls/syscall\_64.tbl文件中添加新的系统调用号和名称，以便内核能够识别并处理新的系统调用。



图4- 2 进入系统调用表

在文件中找到一个空的系统调用入口，通常为sys\_ni\_syscall使用命令 /sys\_ni\_syscall 搜索，找到以后回车，然后点击 i 即可修改内容.系统调用编号是156

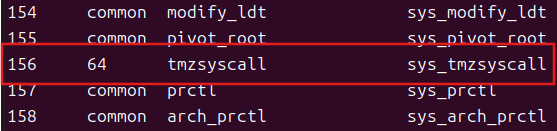


图4- 3 添加新内核系统调用

在Linux内核中，系统调用的声明通常位于include/linux/syscalls.h文件中。这个文件包含了所有可用的系统调用及其参数和返回类型的声明。为了使得内核能够识别并处理新的系统调用，需要在该文件中添加新的系统调用函数的声明

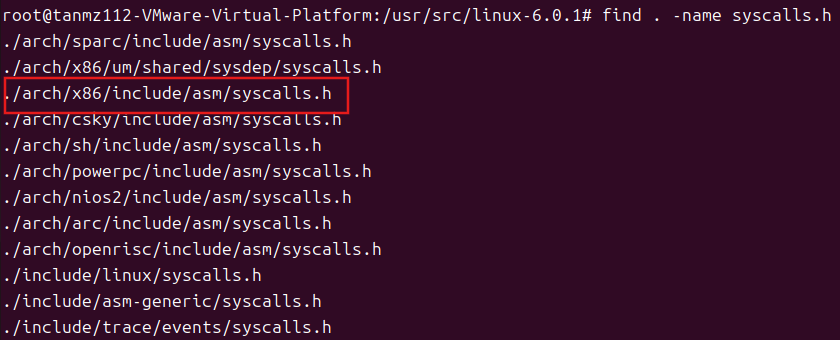


图4- 4 找到系统调用声明表



图4- 5 进入syscall.h添加函数声明

asmlinkage是一个GCC编译器指令，用于指定函数应该从栈中提取其参数，而不是通过寄存器。这是因为在用户态和内核态之间切换时，参数的传递方式可能有所不同。在用户态，函数调用通常通过寄存器传递参数（这取决于具体的CPU架构），但在内核态，由于内核函数可能由中断或异常触发，因此使用栈来传递参数更为可靠。

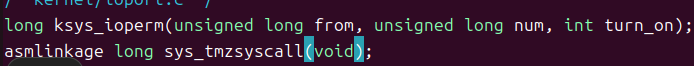


图4- 6 添加系统调用函数声明

虽然系统调用的声明在syscalls.h文件中，但其实现通常位于内核源代码的其他部分。为了完成新的系统调用的添加，需要找到kernel/sys.c源文件，并在其中编写新的系统调用函数的实现代码。但更常见的是，系统调用的实现会分布在多个内核模块或源文件中。



图4- 7 找sys.c位置



图4- 8 进入文件来编写函数

在找到合适的源文件后，需要编写新的系统调用函数的实现代码。这个函数将包含实现新系统调用所需的所有逻辑和代码。通过定义这个函数，可以使得内核在接收到相应的系统调用请求时，能够执行正确的操作并返回预期的结果。

SYSCALL\_DEFINE是一个内核提供的宏，用于简化系统调用的声明。它接受系统调用的名称和参数列表作为输入，并自动生成必要的代码来处理参数、分配系统调用号以及调用实际的系统调用处理函数。我通过使用SYSCALL\_DEFINE0来自动处理。

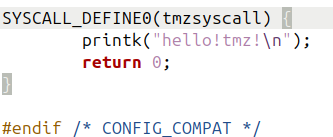


图4- 9定义函数

1. 编译内核

配置内核选项：在编译新的内核之前，需要复制旧内核的配置文件作为新内核配置的基础。这是因为内核配置文件中包含了大量的选项和设置，这些选项和设置决定了内核的功能和特性。通过复制旧内核的配置文件，可以保留大部分的设置，并在此基础上进行必要的修改和调整。

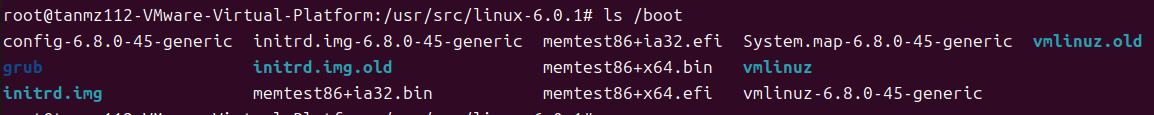


图4- 10 查看旧内核文件名



图4- 11 复制旧内核配置



图4- 12 配置文件

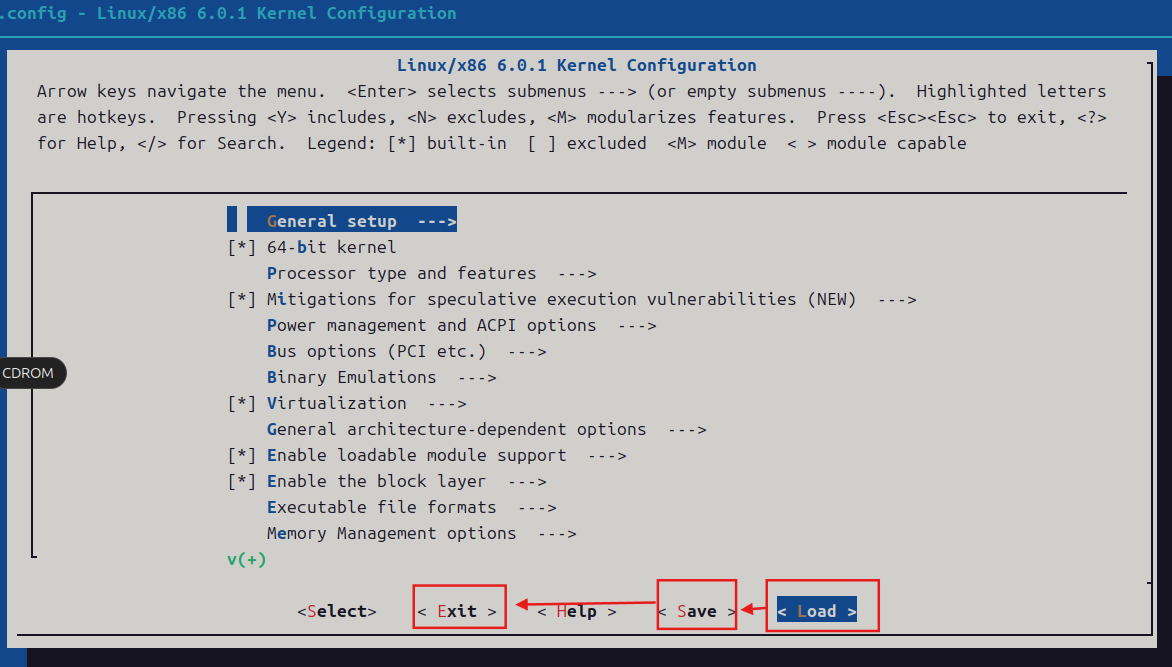


图4- 13 load->save->exit

使用make命令开始编译内核。

图4- 14 编译内核

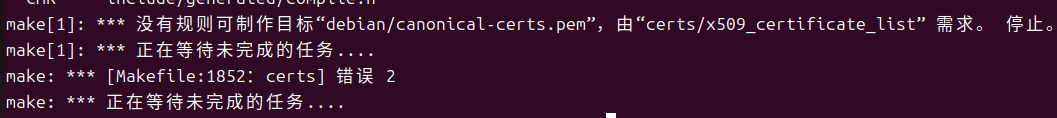


图4- 15 出现错误

为解决错误，编辑.config文件



图4- 16 进入配置文件

修改CONFIG\_SYSTEM\_TRUSTED\_KEYS，将其赋空值，保存并退出

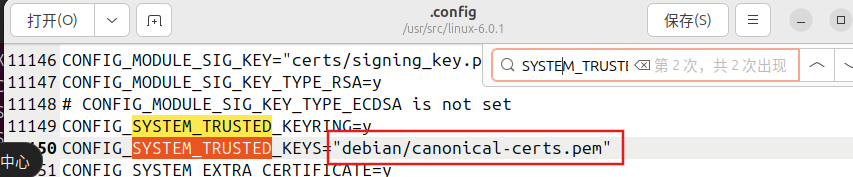


图4- 17 修改配置文件

重新编译，遇到类似错误，将CONFIG\_SYSTEM\_REVOCATION\_KEYS赋空值

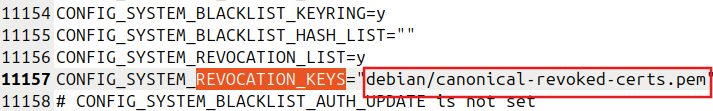


图4- 18 修改配置文件

继续编译：

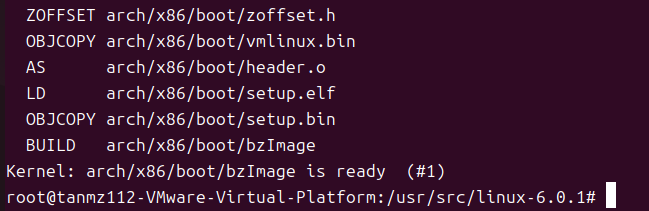


图4- 19 编译成功



图4- 20 编译内核模块

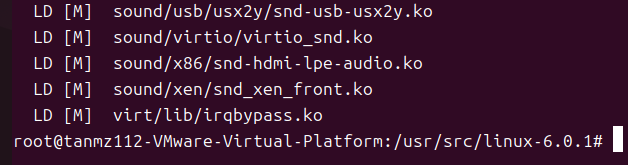


图4- 21 编译成功



图4- 22 安装内核模块

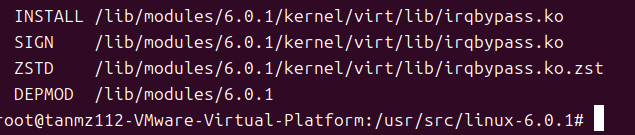


图4- 23 安装成功



图4- 24 安装拷贝镜像文件

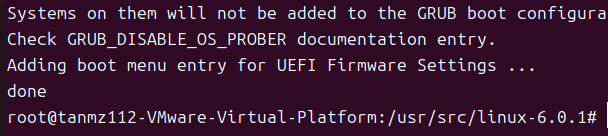


图4- 25 安装成功



图4- 26 重启

1. 测试新的系统调用

重启系统：重启计算机，并在启动时选择新编译的内核。编写测试程序test.c,编写一个简单的C程序，使用syscall函数。



图4- 27 切换内核



图4- 28 编写test.c

在 Linux 中，syscall函数允许用户空间程序直接调用内核提供的服务。这里看到156是我们自定义的系统调用的编号。

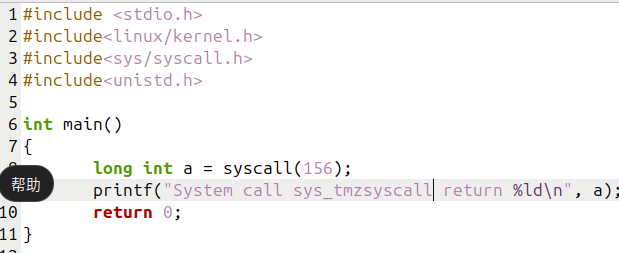


图4- 29 test.c内容

使用 GCC 编译器，读取 test.c 文件，编译它，并将生成的可执行文件命名为a。编译成功后，输入./a（在当前目录下执行名为 a 的可执行文件）来运行编译后的程序。可以看到a的结果和我们定义的test.c函数一致,编译结果正确。

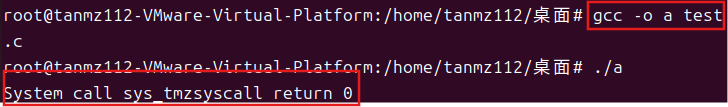


图4- 30 编译结果正确

1. **实验小结**
2. 存在问题
   1. 修改配置文件错误
      1. 报错方式：修改配置文件不能保存
      2. 解决方法：转换超级管理员态
   2. 编译错误
      1. 报错方式：编译卡住
      2. 解决方法：多编译几次，不要输错命令
   3. 内核错误
      1. 报错方式：No rule to make target ‘debian/canonical-certs.pem’, needed by ‘certs/x509\_certificate\_list’
      2. 解决方案：编辑.config文件，修改CONFIG\_SYSTEM\_TRUSTED\_KEYS以及CONFIG\_SYSTEM\_REVOCATION\_KEYS，将他们赋空值。
   4. 编辑错误
      1. 存在问题：将英文“”打成了中文“”
      2. 解决方案：修改代码，重新编译。
3. 注意事项
   1. 注意细节，一个单词的拼错都会导致编译文件产生错误，导致重新编译
   2. 多用快照功能

每次做完一部分实验就使用VMware的系统快照功能在每个关键节点处进行存档，如有问题可及时返回重来

* 1. 边做实验遍记录

完整地记录下实验流程，便于完成后复盘整理，提升自我能力；

* 1. 注意理论与实验的结合

结合Linux理论知识和本次实验，加深了我对Linux系统调用的理解，还提高了我们的动手能力和解决问题的能力。通过实际操作，我更加直观地感受到了Linux系统的强大和灵活性。