A、本课程设计要求学生在阅读Linux操作系统源代码的基础上完成如下三个主要Project（包括A、B、C三类）:

1. 在Linux2.4的内核中添加新的系统调用，以理解系统调用的工作机制，掌握扩展内核功能的技术。
2. 在Linux2.4中修改调度程序的代码，扩展一个新的调度算法。

**A类：**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 序号 | 实验项目 名称 | 内 容 提 要 | 实验性质 | 实验种类 | 学时分配 | 每组人数 | 备注 |
| 1 | Linux内核代码分析（部分） | 1. Linux内核代码的层次分析 2. 调度程序代码段的分析 3. 系统调用内部数据结构以及执行过程的分析 4. 内核调试基本技术 | 专业基础 | 基本型 | 20 | 1 | 必做 |
| 2 | 新增系统调用 | 1、编写一个新系统调用的响应函数，函数的名称和功能由实验者自行定义。把新的系统调用函数嵌入到Linux内核中  2、编写应用程序以测试新的系统调用并输出测试结果 | 专业基础 | 综合型 | 20 | 1 | 必做（2、3选一） |
| 3 | 进程调度的扩展 | 1. 分析Linux进程调度模块 2. 扩展调度模块，使得新创建的普通进程获得的优先级比一个曾经运行过的优先级高 | 专业基础 | 综合型 | 20 | 1 | 必做（2、3选一） |

linux内核添加系统调用(详细)

<https://blog.csdn.net/zstuyyyyccccbbbb/article/details/121756586>

qemu+gdb调试linux内核

<https://blog.csdn.net/zstuyyyyccccbbbb/article/details/121779027>

系统调用内部数据结构以及执行过程的初步分析

<https://blog.csdn.net/zstuyyyyccccbbbb/article/details/121933648>

# 1 整理稀烂报告（任务A）

## Linux内核编译和系统调用方法

​ 计算机系统的各种硬件资源是有限的，在现代多任务操作系统上同时运行的多个进程都需要访问这些资源，为了更好的管理这些资源进程是不允许直接操作的，所有对这些资源的访问都必须有操作系统控制。也就是说操作系统是使用这些资源的唯一入口，而这个入口就是操作系统提供的系统调用（System Call）。在linux中系统调用是用户空间访问内核的唯一手段。

​ linux内核中设置了一组用于实现系统功能的子程序，称为系统调用。系统调用和普通库函数调用非常相似，只是系统调用由操作系统核心提供，运行于内核态，而普通的函数调用由函数库或用户自己提供，运行于用户态。一般情况进程是不能直接访问内核的。它不能访问内核所占内存空间也不能调用内核函数。但实际情况用户程序是需要使用系统的硬件资源的，为了和用户空间上运行的进程进行交互，内核提供了一组接口。通过该接口，由内核帮助应用程序访问硬件设备和其他操作系统资源。

因此，实现系统调用要满足以下几个问题：

1. 通知内核调用一个哪个系统调用；
2. 用户程序把系统调用的参数传递给内核；
3. 用户程序获取内核返回的系统调用返回值。

内核内部定义了6个宏。分别可以调用参数个数为0～6的系统调用。

|  |
| --- |
| \_syscall0(type,name)  \_syscall1(type,name,type1,arg1)  \_syscall2(type,name,type1,arg1,type2,arg2)  \_syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)  \_syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4)  \_syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5)  \_syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5,type6,arg6) |

接下来采用内核版本为Linux kernel 4.9.76进行int 0x80的**系统调用**：

通过open的系统调用进行查看：

|  |
| --- |
| mov 0x05,$eax  int 0x80 |

在arch/x86/kernel/traps.c中的trap\_init中，定义了各种

set\_intr\_gate/set\_intr\_gate\_ist/set\_system\_intr\_gate。其中

set\_system\_intr\_gate用于在中断描述符表(IDT)上设置系统调用：

文本

描述已自动生成

电脑屏幕截图

描述已自动生成

|  |
| --- |
| #ifdef CONFIG\_IA32\_EMULATION      set\_system\_intr\_gate(IA32\_SYSCALL\_VECTOR, entry\_INT80\_compat);      set\_bit(IA32\_SYSCALL\_VECTOR, used\_vectors);  #endif |

根据/home/ycb/linux-4.9.76/arch/x86/include/asm/irq\_vectors.h得

IA32\_SYSCALL\_VECTOR值为0x80。

文本

描述已自动生成

调用 int 0x80 后，硬件根据向量号在 IDT 中找到对应的表项，即中断描述符，进行特权级检查，发现 DPL = CPL = 3 ，允许调用。然后硬件将切换到内核栈 (tss.ss0 : tss.esp0)。接着根据中断描述符的 segment selector 在 GDT / LDT 中找到对应的段描述符，从段描述符拿到段的基址，加载到 cs 。将 offset 加载到 eip。最后硬件将 ss / sp / eflags / cs / ip / error code 依次压到内核栈。

总而言之，这部分是CPU的中断，在收到中断时，CPU 会把当前那条指令执行完，然后转移到中断处理程序，返回后执行的是当前的下一条指令。

然后此时从 entry\_INT80\_32 开始执行，其定义在 arch/x86/entry/ entry\_32.S：

文本

描述已自动生成

文本

描述已自动生成

|  |
| --- |
| ENTRY(entry\_INT80\_32)      ASM\_CLAC      pushl %eax      /\* pt\_regs->orig\_ax \*/      SAVE\_ALL pt\_regs\_ax=$-ENOSYS  /\* save rest \*/      /\*       \* User mode is traced as though IRQs are on, and the interrupt gate       \* turned them off.       \*/      TRACE\_IRQS\_OFF      movl  %esp, %eax      call  do\_int80\_syscall\_32    .Lsyscall\_32\_done:      restore\_all:      TRACE\_IRQS\_IRET    restore\_all\_notrace:    #ifdef CONFIG\_X86\_ESPFIX32      ALTERNATIVE "jmp restore\_nocheck", "", X86\_BUG\_ESPFIX        movl  PT\_EFLAGS(%esp), %eax   # mix EFLAGS, SS and CS    /\*     \* Warning: PT\_OLDSS(%esp) contains the wrong/random values if we     \* are returning to the kernel.     \* See comments in process.c:copy\_thread() for details.     \*/    movb  PT\_OLDSS(%esp), %ah    movb  PT\_CS(%esp), %al    andl  $(X86\_EFLAGS\_VM | (SEGMENT\_TI\_MASK << 8) | SEGMENT\_RPL\_MASK), %eax    cmpl  $((SEGMENT\_LDT << 8) | USER\_RPL), %eax    je ldt\_ss       # returning to user-space with LDT SS  #endif  restore\_nocheck:    RESTORE\_REGS 4        # skip orig\_eax/error\_code  irq\_return:    INTERRUPT\_RETURN  .section .fixup, "ax" |

将存在 eax 中的系统调用号压入栈中，然后调用 SAVE\_ALL 将其他寄存器的值压入栈中进行保存：

屏幕上有字

描述已自动生成

SAVE\_ALL宏的实现：

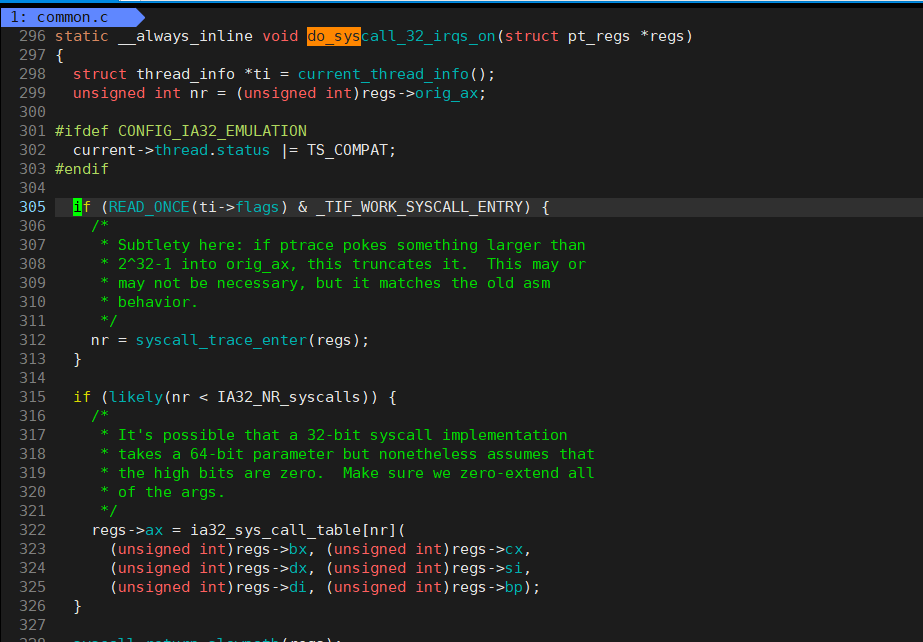
|  |
| --- |
| .macro SAVE\_ALL pt\_regs\_ax=%eax  cld  PUSH\_GS  pushl %fs  pushl %es  pushl %ds  pushl \pt\_regs\_ax  pushl %ebp  pushl %edi  pushl %esi  pushl %edx  pushl %ecx  pushl %ebx  movl $(\_\_USER\_DS), %edx  movl %edx, %ds  movl %edx, %es  movl $(\_\_KERNEL\_PERCPU), %edx  movl %edx, %fs  SET\_KERNEL\_GS %edx  .endm |

保存完毕后，关闭中断，将当前栈指针保存到 eax ，调用 do\_int80\_syscall\_32 => do\_syscall\_32\_irqs\_on。

其中do\_syscall\_32\_irqs\_on在arch/x86/entry/common.c中定义：

文本

描述已自动生成



|  |
| --- |
| static \_\_always\_inline void do\_syscall\_32\_irqs\_on(struct pt\_regs \*regs)  {  struct thread\_info \*ti = current\_thread\_info();  unsigned int nr = (unsigned int)regs->orig\_ax; /\*系统调用号\*/  #ifdef CONFIG\_IA32\_EMULATION  current->thread.status |= TS\_COMPAT;  #endif  if (READ\_ONCE(ti->flags) & \_TIF\_WORK\_SYSCALL\_ENTRY) {  /\*  \* Subtlety here: if ptrace pokes something larger than  \* 2^32-1 into orig\_ax, this truncates it. This may or  \* may not be necessary, but it matches the old asm  \* behavior.  \*/  nr = syscall\_trace\_enter(regs);  }  if (likely(nr < IA32\_NR\_syscalls)) {  /\*  \* It's possible that a 32-bit syscall implementation  \* takes a 64-bit parameter but nonetheless assumes that  \* the high bits are zero. Make sure we zero-extend all  \* of the args.  \*/  regs->ax = ia32\_sys\_call\_table[nr](  (unsigned int)regs->bx, (unsigned int)regs->cx,  (unsigned int)regs->dx, (unsigned int)regs->si,  (unsigned int)regs->di, (unsigned int)regs->bp);  }  syscall\_return\_slowpath(regs);  } |

这个函数的参数 regs(struct pt\_regs 定义见 arch/x86/include/ asm/ptrace.h)就是先前在 entry\_INT80\_32 依次被压入栈的寄存器值。这里先取出系统调用号，从系统调用表(ia32\_sys\_call\_table) 中取出对应的处理函数，然后通过先前寄存器中的参数调用之。

而系统调用表 ia32\_sys\_call\_table 在 arch/x86/entry/syscall\_32.c 中定义。

|  |
| --- |
| /\* System call table for i386. \*/    #include <linux/linkage.h>  #include <linux/sys.h>  #include <linux/cache.h>  #include <asm/asm-offsets.h>  include <asm/syscall.h>    #define \_\_SYSCALL\_I386(nr, sym, qual) extern asmlinkage long sym(unsigned long, unsigned long, unsigned long, unsigned long, unsigned long, unsigne d long) ;  #include <asm/syscalls\_32.h>  #undef \_\_SYSCALL\_I386    #define \_\_SYSCALL\_I386(nr, sym, qual) [nr] = sym,    extern asmlinkage long sys\_ni\_syscall(unsigned long, unsigned long, unsigned long, unsigned long, unsigned long, unsigned long);    \_\_visible const sys\_call\_ptr\_t ia32\_sys\_call\_table[\_\_NR\_syscall\_compat\_max+1] = {  /\*  \* Smells like a compiler bug -- it doesn't work  \* when the & below is removed.  \*/  [0 ... \_\_NR\_syscall\_compat\_max] = &sys\_ni\_syscall,  #include <asm/syscalls\_32.h>  }; |

这里会发现系统调用表的内容是 include 进来的。而且syscalls\_32.h 是在编译过程中动态生成的。

编译内核后在arch/x86/include/generated/asm中：

文本

描述已自动生成

|  |
| --- |
| #ifdef CONFIG\_X87\_32  \_\_SYSCALL\_I386(0, sys\_restart\_syscall, )  #else  \_\_SYSCALL\_I386(0, \_\_ia32\_sys\_restart\_syscall, )  #endif  #ifdef CONFIG\_X86\_32  \_\_SYSCALL\_I386(1, sys\_exit, )  #else  \_\_SYSCALL\_I386(1, \_\_ia32\_sys\_exit, )  #endif  #ifdef CONFIG\_X86\_32  \_\_SYSCALL\_I386(2, sys\_fork, )  #else  \_\_SYSCALL\_I386(2, \_\_ia32\_sys\_fork, )  #endif  #ifdef CONFIG\_X86\_32  \_\_SYSCALL\_I386(3, sys\_read, )  #else  \_\_SYSCALL\_I386(3, \_\_ia32\_sys\_read, )  #endif  #ifdef CONFIG\_X86\_32  \_\_SYSCALL\_I386(4, sys\_write, )  #else |

由#define \_\_SYSCALL\_I386(nr, sym, qual) [nr] = sym,可以获得此时ia32\_sys\_call\_table会变成：

|  |
| --- |
| \_\_visible const sys\_call\_ptr\_t ia32\_sys\_call\_table[\_\_NR\_syscall\_compat\_max+1] = {  [0 ... \_\_NR\_syscall\_compat\_max] = &sys\_ni\_syscall,  [0] = sys\_restart\_syscall,  [1] = sys\_exit,  [2] = sys\_fork,  [3] = sys\_read,  [4] = sys\_write,  [5] = sys\_open,  ...  }; |

open调用号是 0x05 ，所以这里调用了 sys\_open ，该定义在 fs/open.c 中：

截图里有图片

描述已自动生成

|  |
| --- |
| SYSCALL\_DEFINE3(open, const char \_\_user \*, filename, int, flags, umode\_t, mode)  {  if (force\_o\_largefile())  flags |= O\_LARGEFILE;  return do\_sys\_open(AT\_FDCWD, filename, flags, mode);  } |

而宏 SYSCALL\_DEFINE3 及相关定义如下：

|  |
| --- |
| #define SYSCALL\_DEFINE3(name, ...) SYSCALL\_DEFINEx(3, \_##name, \_\_VA\_ARGS\_\_)  #define SYSCALL\_DEFINEx(x, sname, ...) \  SYSCALL\_METADATA(sname, x, \_\_VA\_ARGS\_\_) \  \_\_SYSCALL\_DEFINEx(x, sname, \_\_VA\_ARGS\_\_)  #define \_\_SYSCALL\_DEFINEx(x, name, ...) \  asmlinkage long sys##name(\_\_MAP(x,\_\_SC\_DECL,\_\_VA\_ARGS\_\_)) \  \_\_attribute\_\_((alias(\_\_stringify(SyS##name)))); \  \  static inline long SYSC##name(\_\_MAP(x,\_\_SC\_DECL,\_\_VA\_ARGS\_\_)); \  \  asmlinkage long SyS##name(\_\_MAP(x,\_\_SC\_LONG,\_\_VA\_ARGS\_\_)); \  \  asmlinkage long SyS##name(\_\_MAP(x,\_\_SC\_LONG,\_\_VA\_ARGS\_\_)) \  { \  long ret = SYSC##name(\_\_MAP(x,\_\_SC\_CAST,\_\_VA\_ARGS\_\_)); \  \_\_MAP(x,\_\_SC\_TEST,\_\_VA\_ARGS\_\_); \  \_\_PROTECT(x, ret,\_\_MAP(x,\_\_SC\_ARGS,\_\_VA\_ARGS\_\_)); \  return ret; \  } \  \  static inline long SYSC##name(\_\_MAP(x,\_\_SC\_DECL,\_\_VA\_ARGS\_\_)) |

其中SYSCALL\_METADATA保存了调用的基本信息，供调试程序跟踪使用。这里不用管。

而 \_SYSCALL\_DEFINEx用于拼接函数，函数名被拼接为sys##\_##open，参数也通过 \_\_SC\_DECL 拼接，最终得到展开后的定义：

|  |
| --- |
| asmlinkage long sys\_open(const char \_\_user \* filename, int flags, umode\_t mode)  {  if (force\_o\_largefile())  flags |= O\_LARGEFILE;  return do\_sys\_open(AT\_FDCWD, filename, flags, mode);  } |

也就是说这里用宏来实现把相应的系统调用转化成具体的函数。

sys\_open 是对 do\_sys\_open 的封装。

|  |
| --- |
| long do\_sys\_open(int dfd, const char \_\_user \*filename, int flags, umode\_t mode)  {  struct open\_flags op;  int fd = build\_open\_flags(flags, mode, &op);  struct filename \*tmp;  if (fd)  return fd;  tmp = getname(filename);  if (IS\_ERR(tmp))  return PTR\_ERR(tmp);  fd = get\_unused\_fd\_flags(flags);  if (fd >= 0) {  struct file \*f = do\_filp\_open(dfd, tmp, &op);  if (IS\_ERR(f)) {  put\_unused\_fd(fd);  fd = PTR\_ERR(f);  } else {  fsnotify\_open(f);  fd\_install(fd, f);  }  }  putname(tmp);  return fd;  } |

getname 将处于用户态的文件名拷到内核态，然后通过 get\_unused\_fd\_flags 获取一个没用过的文件描述符，然后 do\_filp\_open 创建 struct file ， fd\_install 将 fd 和 struct file 绑定(task\_struct->files->fdt[fd] = file)，然后返回 fd。

fd一直返回到 do\_syscall\_32\_irqs\_on，被设置到 regs->ax (eax) 中。接着返回 entry\_INT80\_32 继续执行，最后执行 INTERRUPT\_RETURN 。 INTERRUPT\_RETURN 在 `arch/x86/include/asm/irqflags.h` 中定义为 iret ，负责恢复先前压栈的寄存器，返回用户态。系统调用执行完毕。

### 1.1.1内核调试基本技术

#### 1.编译内核

|  |
| --- |
| make menuconfig // 调整编译选项 |

在内核编译选项中，开启如下"Compile the kernel with debug info"

Kernel hacking --->

Compile-time checks and compiler options --->

[ ] Compile the kernel with debug info

示意图如下，利用键盘选中debug选项，然后敲"Y"勾选：

图形用户界面, 文本, 应用程序

描述已自动生成

#### 2.下载qemu

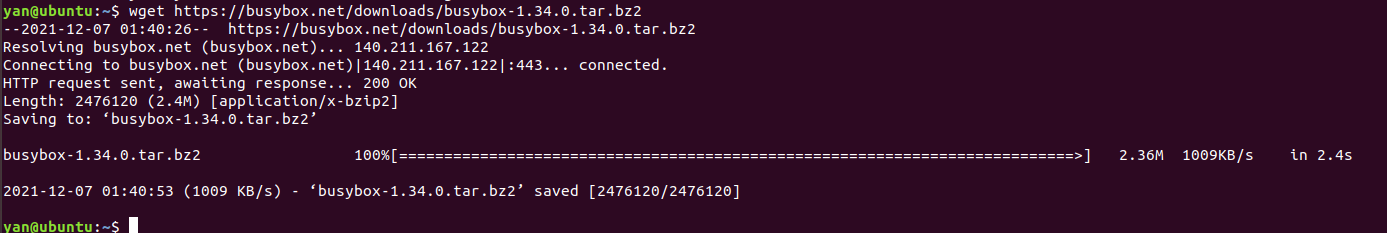
直接安装qemu。

|  |
| --- |
| sudo apt-get install qemu |

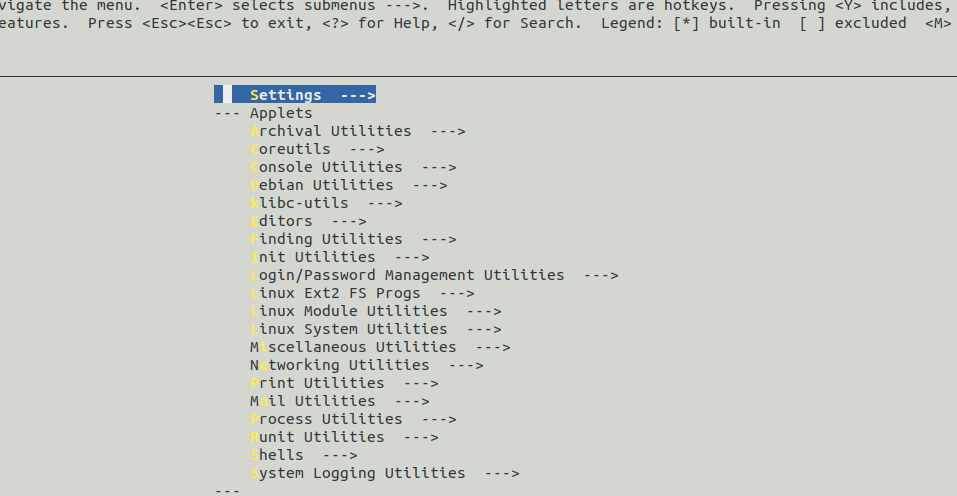
#### 3.构建initramfs根文件系统

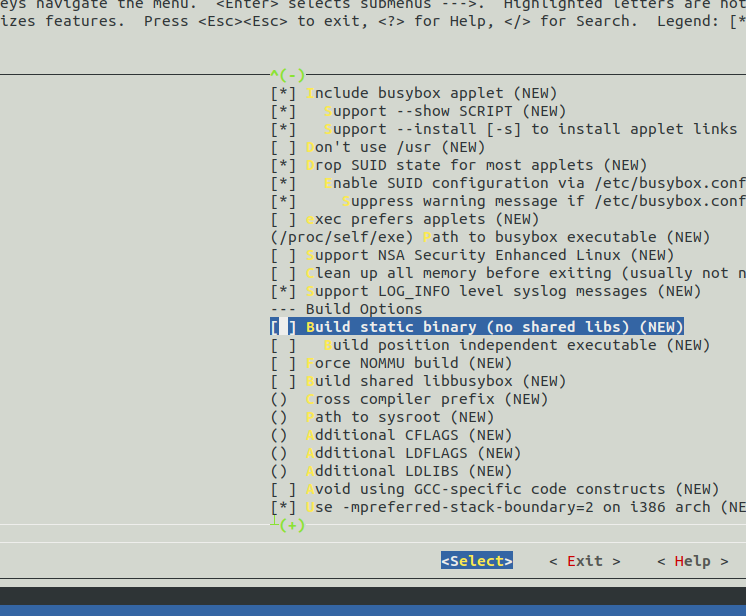
##### 3.1编译Busybox

* <https://busybox.net/> **下载并解压源码**
* wget https://busybox.net/downloads/busybox-1.34.0.tar.bz2



make menuconfig





退出，提示保存，选Yes。

开始编译

make -j4

make install

make 期间会遇到许多的warning，不用管它，对安装影响不大。

此时可以在busybox-1.34.0/中看到生成的\_install目录。

通过下面的命令可以验证：

busybox是否安装正确：./busybox ls

##### 3.2生成initrd

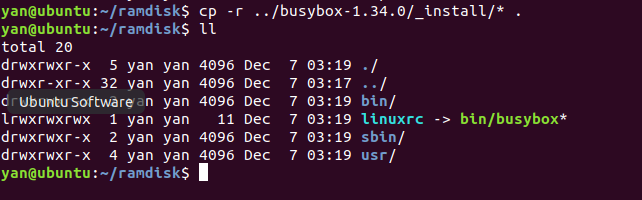
首先将上一步生成的\_install文件夹复制到其他位置。

cd ..

mkdir ramdisk

cd ramdisk

cp -r ../busy-1.34.0/\_install/\* .



设置初始化进程init（建立一个软链接，一定不能直接复制过去）

cd ramdisk

ln -s bin/busybox init

设置开机启动程序

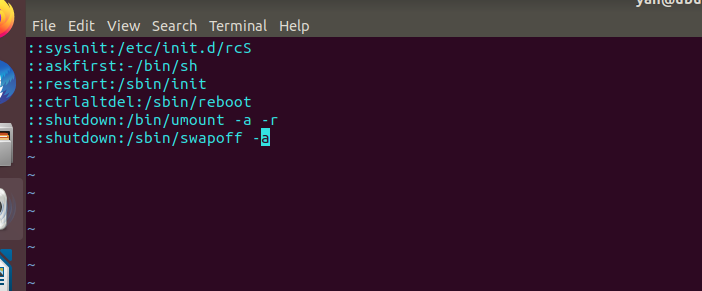
首先，我们需要先设定一些程序运行所需要的文件夹。

  mkdir -pv {bin,sbin,etc,proc,sys,usr/{bin,sbin},dev}

init程序首先会访问etc/inittab文件，因此，我们需要编写inittab，指定开机需要启动的所有程序。

cd etc  
  vim inittab

::sysinit:/etc/init.d/rcS     
  ::askfirst:-/bin/sh      
  ::restart:/sbin/init  
  ::ctrlaltdel:/sbin/reboot  
  ::shutdown:/bin/umount -a -r  
  ::shutdown:/sbin/swapoff -a



直接把这个文件权限拉满 chmod 777 inittab

编写系统初始化命令。

从inittab文件中可以看出，首先执行的是/etc/init.d/rcS脚本，因此，我们生成初始化脚本。

在此新建文件夹并编写脚本

mkdir init.d  
  cd init.d  
  vim rcS

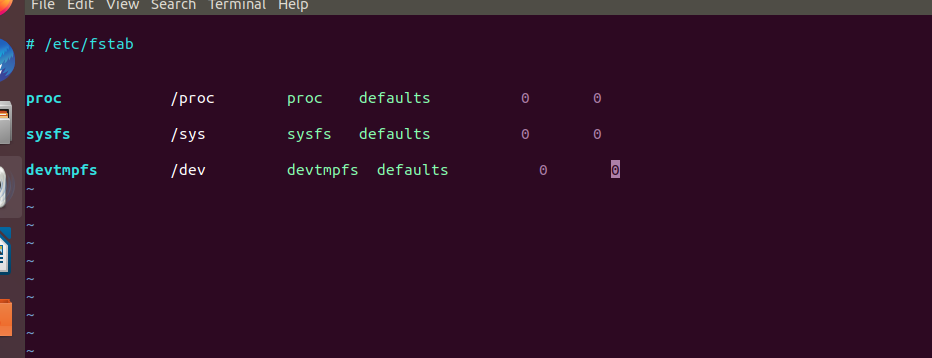
内容如下

|  |
| --- |
| #!/bin/sh  mount proc  mount -o remount,rw /  mount -a      clear                                 echo "My Tiny Linux Start :D ......" |

chmod 777 rcS

在rcS脚本中，mount -a 是自动挂载 /etc/fstab 里面的东西，可以理解为挂在文件系统，因此我们还需要编写 fstab文件来设置我们的文件系统。

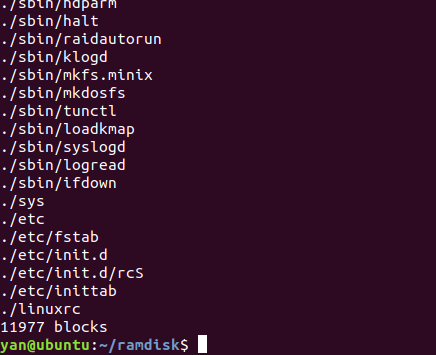
在ramdisk/ect/创建文件fstab



至此，我们已经完成了RAM Disk中相关文件的配置，可以压缩生成文件镜像了。

 cd ramdisk ind . -print0 | cpio --null -ov --format=newc | gzip -9 > ../initramfs.img

**最后生成的initramfs.img就是根文件系统。**



#### 4.gdb调试内核

|  |
| --- |
| qemu-system-x86\_64 -kernel /home/yan/linux-5.4.1/arch/x86\_64/boot/bzImage -initrd ../initramfs.img -smp 2 -S -s -append nokaslr |

图形用户界面, 文本

描述已自动生成

此时，开启另一个terminal，运行如下命令：

|  |
| --- |
| gdb /home/yan/linux-5.4.1/vmlinux (修改成自己的vmlinux路径)  target remote:1234 (默认端口是1234，进行远程连接)  b start\_kernel (设置断点)  c (continue 运行到断点处) |

文本

描述已自动生成

|  |
| --- |
| (gdb) target remote:1234  Remote debugging using :1234  0x000000000000fff0 in entry\_stack\_storage ()  (gdb) b register\_filesystem  Breakpoint 1 at 0xffffffff812f8ff0: file fs/filesystems.c, line 73.  (gdb) c  Continuing.  Thread 1 hit Breakpoint 1, register\_filesystem (  fs=0xffffffff82720180 <sysfs\_fs\_type>) at fs/filesystems.c:73  73 { |

文本

描述已自动生成

#### 5.新增系统调用

##### 5.1编写系统调用响应函数嵌入内核

1. 开机进入Ubuntu系统，安装gparted软件。

2. 打开软件

3. 图形用户界面, 应用程序

描述已自动生成

4. 图形用户界面

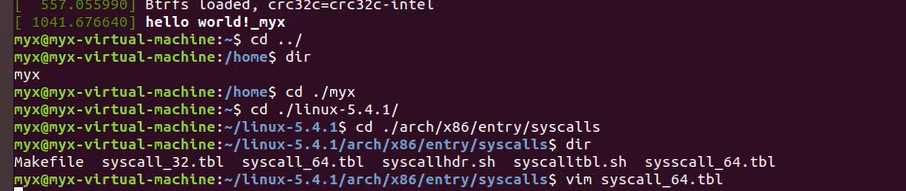
描述已自动生成

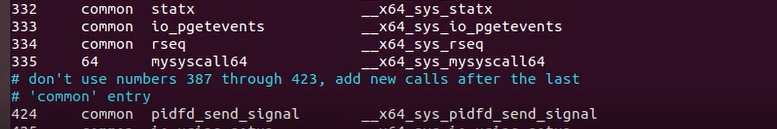
5. 确定后点左上角的√等待操作完成

##### 5.2更新系统调用表，添加系统调用id

修改syscall\_64.tbl

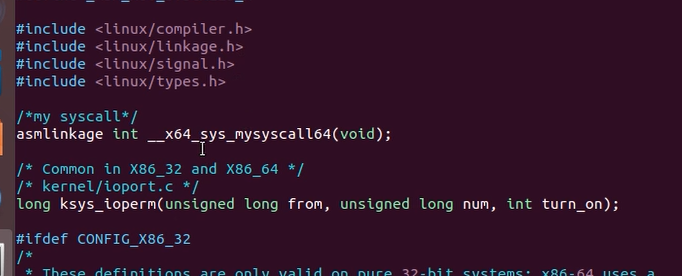
|  |
| --- |
| cd arch/x86/entry/syscalls/ |





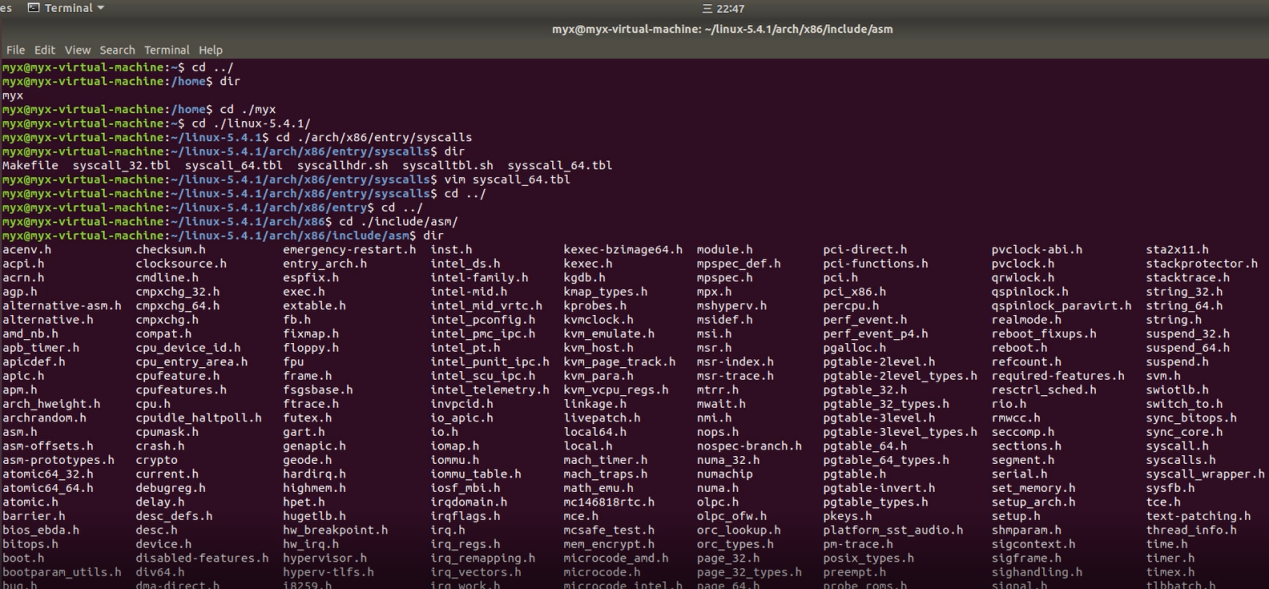
#### 6.添加系统调用函数声明

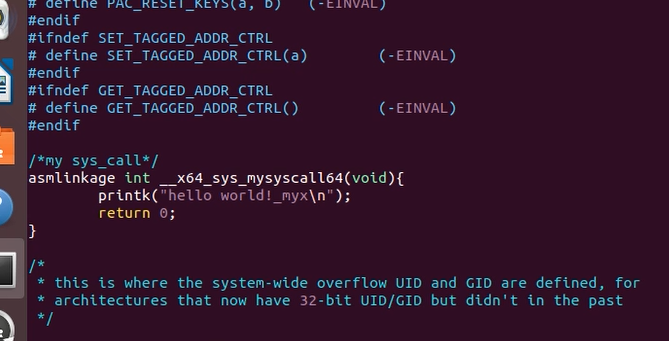
修改arch/x86/include/asm/syscalls.h



#### 7.在内核源代码添加函数定义

linux-5.4.1/kernel/sys.c





#### 8.编译内核

安装所需工具：

|  |
| --- |
| sudo apt-get install gcc make libncurses5-dev openssl libssl-dev  sudo apt-get install build-essential  sudo apt-get install pkg-config  sudo apt-get install libc6-dev  sudo apt-get install bison  sudo apt-get install flex  sudo apt-get install libelf-dev |

图形用户界面, 文本, 应用程序

描述已自动生成

|  |
| --- |
| make menuconfig  make CC='ccache gcc' -j8 |

图形用户界面, 文本

描述已自动生成

文本

描述已自动生成

#### 9.安装模块

|  |
| --- |
| sudo make modules\_install |

文本

描述已自动生成

|  |
| --- |
| sudo make install |

文本

描述已自动生成

#### 10.切换内核

reboot重启虚拟机，按住shift选择进入高级选项,看到linux-5.4.1说明内核。

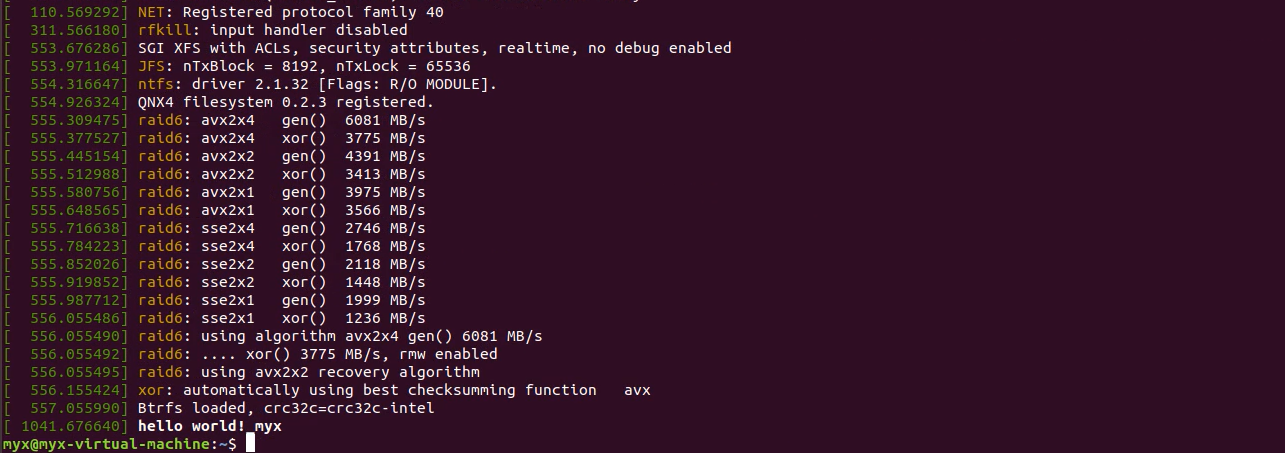
文本

描述已自动生成

#### 11.测试系统调用

|  |
| --- |
| #include<stdio.h>  #include<unistd.h>  #include<sys/syscall.h>  int main(){  syscall(335);  return 0;  } |

|  |
| --- |
| myx@ubuntu:~$ vim test.c  myx@ubuntu:~$ gcc -o test test.c  myx@ubuntu:~$ ./test.out  myx@ubuntu:~$ dmesg |



## Linux内核调试方法（qemu+gdb）

先编译内核：

|  |
| --- |
| # make menuconfig // 调整编译选项 |

在内核编译选项中，开启如下"Compile the kernel with debug info"

Kernel hacking —>

Compile-time checks and compiler options —>

[ ] Compile the kernel with debug info

然后检查自己之前编译的内核有没有该信息。

|  |
| --- |
| # grep CONFIG\_DEBUG\_INFO .config  CONFIG\_DEBUG\_INFO=y |

检查完毕，直接使用命令行安装qemu工具：

# sudo apt-get install qemu

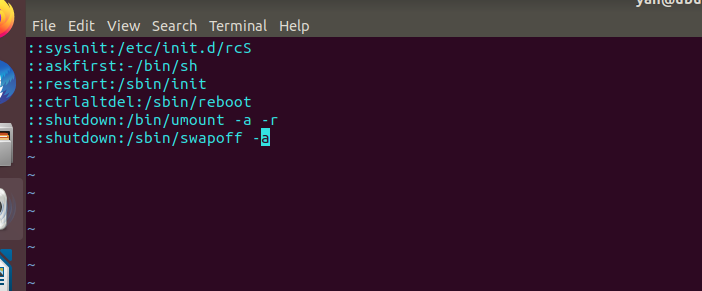
构建initramfs根文件系统，下载并解压Busybox，安装静态依赖进行编译。然后生成initrd，设置初始化进程init和开机启动程序。这部分具体步骤如下：

设定程序运行所需要的文件夹：

mkdir -pv {bin,sbin,etc,proc,sys,usr/{bin,sbin},dev}

init程序首先会访问etc/inittab文件，因此，我们需要编写inittab，指定开机需要启动的所有程序：

|  |
| --- |
| cd etc  vim inittab |



赋予权限：

|  |
| --- |
| chmod 777 inittab |

编写系统初始化命令。

从inittab文件中可以看出，首先执行的是/etc/init.d/rcS脚本，因此，我们生成初始化脚本。

在此新建文件夹并编写脚本：

mkdir init.d

cd init.d

vim rcS

内容如下：

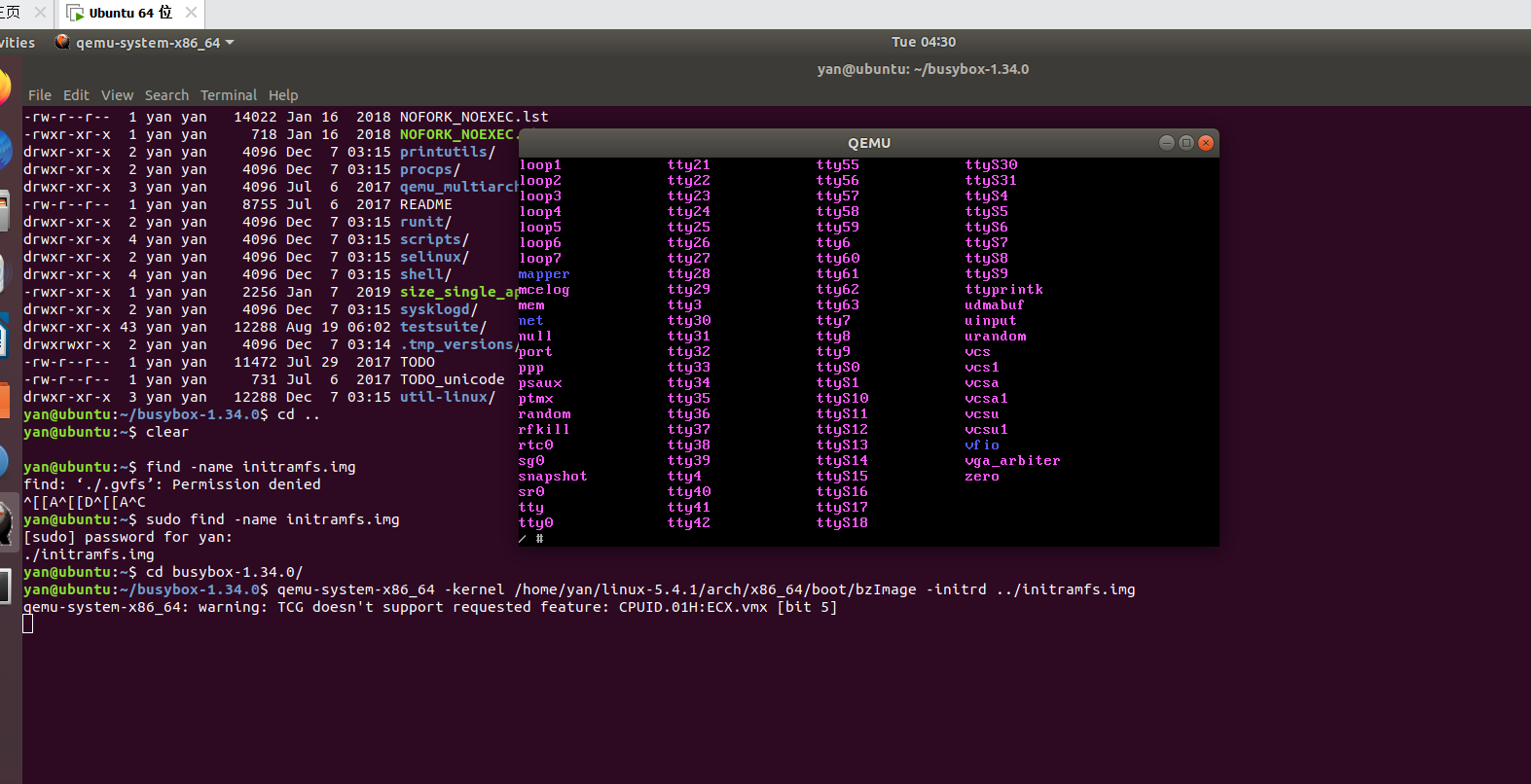
|  |
| --- |
| #!/bin/sh  mount proc  mount -o remount,rw /  mount -a  clear  echo "My Tiny Linux Start :D ......" |

在rcS脚本中，mount -a是自动挂载/etc/fstab里面的东西，可以理解为挂在文件系统，因此我们还需要编写 fstab文件来设置我们的文件系统。

在ramdisk/ect/创建文件fstab

最后生成的initramfs.img就是我们的根文件系统。

可以进入编译完成的内核的文件系统，输入ls /dev查看是否挂载成功。



然后使用gdb调试内核：

启动gdb，然后开启另一个terminal，运行以下命令：

|  |
| --- |
| gdb /home/yan/linux-5.4.1/vmlinux (修改成自己的vmlinux路径)  target remote:1234 (默认端口是1234，进行远程连接)  b start\_kernel (设置断点)  c (continue 运行到断点处) |

