Lista - Confecção de uma chamada de sistema

Nome: João Pedro Gavassa Favoretti

No USP: 11316055

Rodando o kernel padrão:

Para implementar minha própria chamada do sistema, primeiro foi necessário decidir qual versão do kernel do Linux utilizar, como foi recomendado a v5.14, foi a que eu utilizei primeiro. (Link para download).
Depois disso, pensei que fosse ser mais fácil se eu utilizasse um emulador de hardware para rodar a versão do sistema operacional, ao invés de instalá-lo na minha própria máquina. Como eu já tinha uma noção de como o qemu funciona, foi o que eu utilizei.
Utilizei um diretório específico na minha máquina para extrair e compilar o kernel, então vou chamar o diretório raiz do kernel de KERNEL_SRC para melhorar a explicação.

```
$ tar xzf ./linux-5.14
$ export KERNEL_SRC=$(pwd)/linux-5.14
$ cd $KERNEL_SRC
```

Depois disso é necessário compilar o código fonte do kernel para gerar os executáveis do sistema operacional. O Linux possui um modo bem customizável para executar a compilação, ele utiliza um arquivo chamado de `.config` no diretório raiz, para explicitar quais módulos que devem ser compilados. Se você decidir ativar todas as opções de módulos que o .config aceita, provavelmente a compilação vai demorar muito, e se ativar o mínimo de modos, então a compilação vai demorar bem menos.
Além de possuir esse arquivo `.config` para especificar a compilação do kernel o linux possui alguns comando para ajudar a construir o arquivo .config, entre eles: `make defconfig`, `make menuconfig`, `make nconfig`. O segundo e o terceiro comando são alguns modos visuais de você adicionar ou remover módulos na hora da compilação. Mas não é necessário depender desses comandos para configurar o arquivo `.config`, como ele é um arquivo de texto, também é possível abrir o `.config` em um editor de texto e editar na mão.
O comando que eu utilizei para compilar o código fonte foi `make defconfig`, como o nome já diz, ele adiciona a configuração default (padrão) no arquivo `.config`.

No diretório `$KERNEL_SRC`:

```
$ make defconfig
(...)
```

Depois disso já posso ver o arquivo .config no diretório raiz do meu repositório:

```
$ ls -l .config
-rw-rw-r-- 1 joao joao 126753 out  3 12:54 .config
```

Agora está tudo pronto para compilar o código fonte do kernel:

```
$ nproc
8
$ make -j8
(...)
```

Utilizei o comando `nproc` para checar a quantidade de núcleos de processamento que estão disponíveis no meu computador e depois o comando `make` com a flag `-j8` (específica que quero usar 8 unidades de processamento durante a compilação, quando mais núcleos, mais rápido a compilação irá ocorrer) para compilar o código fonte baseado no que está escrito no `.config`.

Com isso a compilação gera um arquivo chamado de bzImage dentro do diretório `arch/x86/boot/bzImage` do diretório raiz. Alias, também gera o arquivo dentro do diretório `arch/x86_64/boot/bzImage`, que é o mesmo arquivo.
Esse arquivo o que foi gerado da compilação, para saber o que ele é, podemos usar o comando file do linux:

```
$ file arch/x86/boot/bzImage
./arch/x86/boot/bzImage: Linux kernel x86 boot executable bzImage,
version 5.14.0 (joao@joao-Lenovo-IdeaPad-S145) #4 SMP Sun Oct 3 16:21:44
-03 2021, RO-rootFS, swap_dev 0x9, Normal VGA
```

Depois de compilado, ingenuamente achei que rodar somente o boot executable bzImage fosse ser o suficiente para prover uma interface linux:

```
$ qemu-system-x86_64 -kernel ./arch/x86/boot/bzImage
```

Mas a única coisa que eu consegui foi gerar um Kernel panic dentro do emulador:

```
(...)
[    10.116888] ---[ end Kernel panic - not syncing: VFS: Unable to
mount root fs on unkown-block(0,0) ]---
```

A partir daí percebi que eu preciso de um root fs (file system) montado antes de rodar o

kernel desse modo. Mas eu não fazia idéia do que significa isso, e foi o mais complicado de descobrir. Parecia que eu estava atirando no escuro para descobrir como fazer isso, além de descobrir o que mais era necessário para executar o kernel.
Depois de pesquisar me deparei com algo chamado `ramdisk` que aparentemente é o primeiro binário que executa ao bootar o computador e ele gera as condições necessárias para rodar o binário do kernel. Mas isso ainda não é um root filesystem.
Parando para pensar faz sentido que precisemos utilizar um filesystem para rodar um sistema operacional, visto que utilizamos o terminal em algum diretório e sem diretório deveria ser possível utilizar o sistema operacional.
Depois de tentar bootar somente com um ramdisk, o kernel também não funcionou como eu gostaria, ele ainda emitiu um warning que não possuía um root fs e deixava eu rodar uma versão reduzida do terminal disponível no linux.
Pesquisando mais um pouco descobri um repositório chamado syzkaller, que é um fuzzer FOSS feito primeiramente pela Google para descobrir vulnerabilidades no kernel do linux e eles disponibilizam várias ferramentas para qualquer um utilizar. Por sorte eles tinham um script para construir um root fs baseado no Debian e foi o que eu utilizei.

Dentro do diretório `$KERNEL_SRC`:

```
$ mkdir image/; cd image/
$ wget
https://raw.githubusercontent.com/google/syzkaller/master/tools/create-i
mage.sh
$ chmod u+x ./create-image.sh
$ ./create-image.sh
(...)
```

Então o script cria o filesystem dentro do diretório `$KERNEL_SRC/image/` . Olhando o script conseguir entender que esse script usa outro programa chamado de deboostrap que cria um sistema baseado em debian dentro de um diretório. Ainda não entendi nem 10% de por que precisamos de um filesystem para bootar um sistema operacional e nem como esse deboostrap funciona. Segui isso com base em alguns posts em blogs que achei pela internet.

Podemos ver que também foi criado um arquivo stretch.img dentro desse diretório, que fica claro o que é quando usando denovo o comando file:

Dentro do diretório `$KERNEL_SRC/image`:

```
$ file stretch.img
stretch.img: Linux rev 1.0 ext4 filesystem data,
UUID=0fe821cb-a89f-4401-801a-b1a3ae4523c7 (n)
```

Também não entendi exatamente o que ele é, mas sei que precisamos utilizar ele para explicitar as informações do filesystem que criei.

Além desse script, também usei outro script para para rodar o qemu junto com esse filesystem que acabei de criar:

Dentro do diretório `$KERNEL_SRC`:

```
$ cat ./run.sh
qemu-system-x86_64 \
      -m 2G \
      -smp 2 \
      -kernel ./arch/x86/boot/bzImage \
      -append "console=ttyS0 root=/dev/sda earlyprintk=serial
net.ifnames=0 nokaslr" \
      -drive file=./image/stretch.img,format=raw \
      -net user,host=10.0.2.10,hostfwd=tcp:127.0.0.1:10021-:22 \
      -net nic,model=e1000 \
      -enable-kvm \
      -nographic \
      -pidfile vm.pid \
      2>&1 | tee vm.log
```

Comparado com o script ./create-image.sh, não é muito dificil de entender o que esse script faz, dar uma olhada nas tags do qemu ajuda a entender melhor. Mas basicamente ele usa a imagem do kernel que eu compilei, o arquivo ./image/stretch.img que é a imagem do fs e o diretório /dev/sda como root para o sistema operacional.

Depois de rodar esse script, tudo funcionou corretamente.
Consegui utilizar comandos básicos do linux dentro dos diretórios montados.:

```
(...)
[  OK  ] Started Raise network interfaces.
[  OK  ] Reached target Network.
You are in emergency mode. After logging in, type "journalctl -xb" to
view
system logs, "systemctl reboot" to reboot, "systemctl default" or ^D to
try again to boot into default mode.
Press Enter for maintenance
(or press Control-D to continue):
root@syzkaller:~#
```

É possível ver que o sistema operacional bootou em emergency mode, depois testei compilar outras versões do kernel, como v5.10 e não deu esse problema, não entendi exatamente qual o problema. Mas fora isso, o sistema operacional funcionou corretamente, inclusive o sistema de arquivos possuía o comando apt, como gerenciador de pacotes, então consegui instalar o build-essential e o vim para testar e compilar alguns programas em C.
Depois que conseguimos compilar e rodar o kernel padrão, passamos para as modificações.

Modificações:

Primeiramente, precisamos criar uma definicão da syscall para seguir durante toda a implementação:

```
void mysyscall(char *str);
// Número: 448
```

Seguindo alguns passos baseados nessa versão do kernel, para criar minha própria syscall precisamos declarar o número dela dentro do arquivo `$KERNEL_SRC/arch/x86/entry/syscalls/syscall_64.tbl`. Então assim fiz:

```
(...)
445   common  landlock_add_rule     sys_landlock_add_rule
446   common  landlock_restrict_self  sys_landlock_restrict_self
447   common  memfd_secret          sys_memfd_secret
+448  common  mysyscall             sys_mysyscall
+
#
# Due to a historical design error, certain syscalls are numbered
differently
(...)
```

Depois disso só vamos seguir o padrão das syscalls já existentes. Também precisamos modificar o arquivo `include/linux/syscalls.h` para declarar a syscall que será implementada.

```
 * not implemented -- see kernel/sys_ni.c
 */
asmlinkage long sys_ni_syscall(void);

+asmlinkage long sys_mysyscall(char *str);
+
#endif /* CONFIG_ARCH_HAS_SYSCALL_WRAPPER */
```

E depois dizemos que nossa syscall declarada no header syscalls.h possui o número 448, no arquivo `include/uapi/asm-generic/unistd.h:`

```
__SYSCALL(__NR_memfd_secret, sys_memfd_secret)
#endif

+#define __NR_mysyscall 448
+__SYSCALL(__NR_mysyscall, sys_mysyscall)

#undef __NR_syscalls
```

```
+#define __NR_syscalls 449
```

E o último local para registrar a syscall é em `kernel/sys_ni.c:`

```
COND_SYSCALL(setreuid16);
COND_SYSCALL(setuid16);

+COND_SYSCALL(mysyscall);
+
/* restartable sequence */
```

Depois disso passamos para implementar a função que será executada:

Em `$KERNEL_SRC/kernel/mysyscall.c:`

```
#ifndef __LINUX_MYSYSCALL_H
#define __LINUX_MYSYSCALL_H
#include <linux/syscalls.h>
#include <linux/printk.h>
#endif

SYSCALL_DEFINE1(mysyscall, char *, str)
{
	printk("[DEBUG]: mysyscall was called");
	printk("%s", str);
	return 0;
}
```

O macro `SYSCALL_DEFINE1` cria o header da minha syscall e 1 se refere à quantidade de argumentos, vemos que só temos realmente 1 argumento `char *str`
Esse macro é importado de `linux/syscalls.h` e o printk é importado de `linux/printk.h`, como o kernel é compilado antes da libc, não existe a função printf, então outra função de print foi implementada dentro do kernel para utilização em logs.

Além de implementar precisamos adicionar esse arquivo ao Makefile para compilar junto com o kernel.

Dentro de `kernel/Makefile`:

```
$(obj)/kheaders_data.tar.xz: FORCE
	$(call cmd,genikh)

clean-files := kheaders_data.tar.xz kheaders.md5
```

```
+obj-y += mysyscall.o
```

Rodando o kernel modificado:

Depois de realizar todas as modificações, o .config e a imagem do root fs já estão gerados. Então só precisamos re-compilar o kernel e rodar novamente:

Dentro de $KERNEL_SRC:

```
$make -j8
(...)
$./run.sh
(...)
```

Dentro do kernel emulado, criei um arquivo para testar a syscall:

```
~/teste_syscall.c
#include <unistd.h>
#include <stdio.h>

#define __NR_mysyscall 449

int main(int argc, char **argv) {
      if (argc < 2) {
            printf("usage: %s <console message>\n", argv[0]);
            return -1;
      }
      syscall(__NR_mysyscall, argv[1]);
      return 0;
}
```

Compilando e rodando com o gcc:

```
root@syzkaller:~# gcc teste_syscall.c
root@syzkaller:~# ./a.out
usage: ./a.out <console message>
root@syzkaller:~# ./a.out "Minha mensagem"
[  274.556947] [DEBUG]: mysyscall was called
root@syzkaller:~# ./a.out "Minha mensagem 2"
[  274.557805] Minha mensagem
[  277.440934] [DEBUG]: mysyscall was called
root@syzkaller:~#
```

Como pode ser percebido, na primeira vez que faço a chamada de sistema ele não exibe a mensagem escrita. Não entendi muito bem o por que ainda, mas ele sempre exibe a mensagem de debug, independente da primeira vez. Provavelmente eu devo ter configurado a syscall errado de algum modo, ou é um problema de endereçamento.

Além de rodar o programa por si só, podemos usar o comando strace para listar todas as syscalls utilizadas pelo programa:

```
root@syzkaller:~# strace ./a.out "Minha mensagem 3"
execve("./a.out", ["./a.out", "Minha mensagem 3"], [/* 10 vars */]) = 0
brk(NULL)                                       = 0x560001e9f000
access("/etc/ld.so.nohwcap", F_OK)              = -1 ENOENT (No such file or
directory)
access("/etc/ld.so.preload", R_OK)              = -1 ENOENT (No such file or
directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=15756, ...}) = 0
mmap(NULL, 15756, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fc684880000
close(3)                                        = 0
access("/etc/ld.so.nohwcap", F_OK)              = -1 ENOENT (No such file or
directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3,
"\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\4\2\0\0\0\0\0"...,
832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1689360, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7fc68487e000
mmap(NULL, 3795296, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3,
0) = 0x7fc6842c2000
mprotect(0x7fc684457000, 2097152, PROT_NONE) = 0
mmap(0x7fc684657000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x0
mmap(0x7fc68465d000, 14688, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 00
close(3)                                        = 0
arch_prctl(ARCH_SET_FS, 0x7fc68487f440) = 0
mprotect(0x7fc684657000, 16384, PROT_READ) = 0
mprotect(0x560001000000, 4096, PROT_READ) = 0
mprotect(0x7fc684884000, 4096, PROT_READ) = 0
munmap(0x7fc684880000, 15756)                   = 0
syscall_448(0x7ffd44e77f0d, 0x7ffd44e76240, 0, 0x560000e007d0,
0x7fc684670ba0, 0x7ffd44e7622m
[  384.036365] [DEBUG]: mysyscall was called
) = 0
exit_group(0)                                   = ?
+++ exited with 0 +++
```

```
root@syzkaller:~#
```

Em negrito podemos ver a syscall de número 448 chamada, que foi a que criei. O que mostra novamente que conseguimos utilizar a syscall criada.