Insper

14 e 15 - linux do zero

Igor dos Santos Montagner

Parte 0 - ferramentas

Vamos precisar dos seguintes softwares instalados no sistema. Como o resto do curso, os pacotes abaixo são para o Ubuntu 18.04 LTS,

build-essential flex bison qemu ncurses-dev libssl-dev libelf-dev qemu-system-x86

Parte 1 - compilando o kernel

Vamos primeiro fazer o download do kernel do Linux no site oficial(https://www.kernel.org/). Para este roteiro escolheremos a versão `5.6.8`.

$ tar xvf linux-5.6.8.tar.xz

Isto criará uma pasta `linux-5.6.8` que contém os fontes de todo o kernel. O kernel pode ser compilado com uma quantidade enorme de configurações diferentes, sendo que a configuração atual é salva no arquivo `.config` dentro da pasta do código fonte. Criaremos um kernel com as opções padrão usando o seguinte comando.

```
> $ make defconfig
```

Para ver quais configurações estão disponíveis podemos usar o comando

```
> $ make menuconfig
```

Exercício: Habilite a opção *Linux guest support* dentro de *Processor Type and Features*.

Execute `make -j8` para compilar seu kernel. Isto demora em torno de 10~20 minutos. No fim desta etapa devemos ter um arquivo `bzImage` na pasta `arch/x86_64/boot/`. Este é o arquivo executável contendo o kernel Linux que compilamos.

Parte 2 - a biblioteca padrão - `libc`

A biblioteca padrão *C* é contém uma função em *C* para cada chamada de sistema disponível e interpreta os códigos de erro, deixando-os em um formato (um pouco) mais amigável. Como vimos em aula, realizar chamadas de sistema é uma tarefa que depende do hardware e por isso é diferente em cada arquitetura (ARM vs x86, por exemplo). Logo, a `libc` oferece uma camada de abstração maior acima do sistema operacional, já que programas construídos usando suas função são portáveis em nível de código fonte. Ou seja, necessitando somente a recompilação do executável para funcionar em outras plataformas. Ela também oferece as funcionalidades necessárias para carregar dinamicamente bibliotecas `.so`, como vimos na aula 11. Por outro lado, além do kernel precisamos portar a `libc` também cada vez que trabalhamos com arquiteturas novas.

Neste exemplo iremos usar a `glibc`, implementação feita pela GNU e usada na maioria das distribuições. Desta vez não precisamos compilar nada: ela já está instalada no nosso sistema e podemos simplesmente usá-la na próxima parte.

Exercício: Crie um programa *Hello world (seu nome)* e compile-o com o nome `hello-dyn`. O comando `ldd` é usado para listar todas as bibliotecas dinâmicas usadas por um executável. Use-o no seu *Hello world* e coloque a saída abaixo. Você consegue identificar a `libc` nesta saída?

Todos os arquivos acima são carregados na memória antes da execução de `hello-dyn`. Ou seja, para que `hello-dyn` execute estes arquivos precisam estar presentes no sistema no local listado acima. Por esta razão, cada arquivo `.so` listado é chamado de **dependência** de `hello-dyn`.

Exercício: Vamos agora usar a flag `-static` do `gcc` para embutir todos os arquivos acima em um único executável. Crie um novo executável `hello-static` e cheque que realmente ele não usa nenhuma biblioteca usando `ldd`.

Vamos usar ambos programas mais para a frente do roteiro.

Parte 3 - ferramentas de modo usuário - `busybox`

Agora que já temos uma interface com o kernel via `glibc` precisamos de programas básicos para utilizar nosso sistema. Estamos falando de programas como `cp`, `ls` e até mesmo o nosso shell (`bash`). Assim como o kernel e a libc, o padrão *POSIX* também diz como esses programas deverão funcionar, fazendo com que sua utilização básica seja igual em qualquer sistema compatível. Lembre-se que o kernel não faz nada, ele apenas *intermedia* o acesso ao hardware.

O *busybox*(https://busybox.net/about.html) é um conjunto de ferramentas modo usuário bastante compacto e rápido. Ele contém implementações leves dos executáveis `ash` (shell leve alternativo ao `bash`), `ls`, `vi`, `pwd`, etc. Sua vantagem é o baixo consumo de memória e seu tamanho pequeno após compilado. É muito usado em sistemas embarcados.

Vamos começar baixando os fontes da versão `1.31.1`. A compilação é feita no mesmo esquema do kernel:

```
> $ make defconfig
> $ make menuconfig
```

Exercício: O busy box disponibiliza um grande número de ferramentas. Procure no menu acima o lugar onde são listados os editores de texto disponíveis no *busybox*.

Desta vez iremos fazer uma modificação nas configurações padrão. Como queremos que esses executáveis pequenos, muito rápidos e que rodem sem qualquer outro tipo de serviço carregado no sistema, iremos compilá-los **estaticamente**.

Exercício: Procure a opção para lincar o `busybox` estaticamente (Submenu *Settings*), habilite-a e faça a compilação.

```
> $ make -j8
```

Isto demorará bem menos que a compilação do kernel e pode ser feito enquanto outras coisas acontecem. Após a compilação um executável `busybox` deverá ter sido gerado na pasta *busybox-1.31.1*.

O `busybox` inclui, em um só executável, todas ferramentas listadas acima. Para executá-las basta passar o nome da ferramenta escolhida como argumento. Veja o exemplo abaixo.

```
> $ busybox ls
```

Se tudo funcionou igual ao `ls` padrão de seu sistema então passe para o próximo passo.

Exercício: Execute `busybox ls --help`. Compare a saída com `ls --help`. Existe diferença? Procure na saída do `ls` do seu sistema qual a implementação utilizada.

Exercício: Você pode obter uma lista completa de todas as ferramentas que o busybox oferece executando `busybox --list-full`. Execute o comando e interprete sua saída. Esses comandos estão disponíveis no seu sistema atual?

Parte 4 - criando o sistema de arquivos

Agora que já temos um kernel, ferramentas de modo usuário e uma `libc` disponível para compilar programas iremos montar a hierarquia de diretórios do Linux. Esta é a última etapa que precisamos cumprir antes de ter um sistema que faz *boot*.

Exercício: Leia a documentação do Debian](https://wiki.debian.org/FilesystemHierarchyStandard)(https://wiki.debian.org/FilesystemHierarchyStandard) sobre um padrão adotado pela grande maioria das distribuições. Note que este não é um padrão POSIX (o macOS faz tudo isso diferente e ainda assim segue a especificação).

> **!** A hierarquia de diretórios não representa (necessariamente) um disco físico, mas sim uma organização das informações disponíveis no sistema. Podemos "pendurar" o conteúdo de um disco em basicamente qualquer diretório.

Agora que você já conhece um pouco melhor como tudo está organizado em um sistema baseado em Linux, vamos começar criando um arquivo vazio de *100Mb* que será usado como nosso diretório raiz `/`.

```
> $ dd if=/dev/zero of=raiz.img bs=1M count=100
```

Antes de continuar, precisamos expor o arquivo acima como um dispositivo de armazenamento para o restante do sistema. Podemos fazer isto usando um *loopback device*. Este tipo de dispositivo se comporta igual a um disco físico, mas modifica os bytes de um arquivo ao invés de interagir com hardware. O comando `losetup` é usado para fazer este serviço. Todo comando cujo prompt começa com `#` deverá ser executado como *root*.

```
> # losetup -P -f --show raiz.img
```

Agora podemos criar uma *tabela de partições*. Esta estrutura, gravada no começo de um disco, contém informações que permitem identificar quais *partições* estão presentes, seus tamanhos e tipo. Em sua essência, uma *partição* é somente uma subdivisão de um disco físico. Isto ajuda, por exemplo, a instalar vários SOs no mesmo disco sem precisar ter um disco separado para cada sistema. Fazemos tudo isto usando o comando `fdisk`:

```
> # fdisk /dev/loop0
```

O `fdisk` trabalha como um prompt de comandos. Digite `m` para conhecer as opções.

Exercício: Crie uma partição neste disco ocupando todo o espaço disponível. As opções padrão do comando de criar partições são adequadas para nosso uso.

Exercício: Use o comando `p` para mostrar o estado atual do disco. Certifique-se de que há uma partição do tipo *Linux* que ocupe o disco todo. Anote o valor do campo *Disk Identifier* abaixo.

> **!** Não se esqueça de usar o comando `w` para salvar as partições criadas.

> **!** Em *nix um arquivo é simplesmente uma sequência de bytes. Se eu pedir para o sistema interpretar esta sequência como um disco formatado no padrão* ext4* isto terá o mesmo efeito que se essa sequência de bytes estivesse armazenada diretamente em um disco físico.

A partir de agora, o dispositivo */dev/loop0* (ou algo similar que tenha sido retornado pelo comando acima) é equivalente a um disco físico. Assim como localizamos a primeira partição de um disco usando `/dev/sda1`, localizamos a primeira partição do nosso *loop device* usando `/dev/loop0p1`.

> **!** Se você só tem `/dev/loop0/` e não possui `/dev/loop0p1` então houve algo errado com a criação das partições de seu disco. Refaça tudo a partir do comando `dd`.

Vamos agora formatar essa partição e montá-la no diretório *raiz_linux*.

```
> # mkfs.ext4 /dev/loop0p1
> # mkdir raiz_linux
> # mount -t ext4 /dev/loop0p1 raiz_linux
```

Tudo o que for escrito na pasta *raiz_linux* será escrito diretamente no nosso arquivo *raiz.img* da mesma maneira que seria escrito em um disco físico. No nosso caso, tudo o que for colocado nesta pasta estará presente no diretório `/` do nosso sistema Linux. Ou seja, a pasta *raiz_linux/bin* no nosso sistema será somente */bin*.

Vamos agora copiar para *raiz_linux* o mínimo necessário para conseguirmos ligar nosso sistema em um prompt de comando *bash* como *root*. Na parte 6 iremos completar nosso sistema com todo o resto necessário para que ele funcione de maneira plena.

Exercício: O primeiro passo é montar a hierarquia de arquivos descrita no primeiro exercício desta seção. Crie as pastas descritas naquele documento. Ao dar `ls -l` em `raiz_linux` você deverá ver a seguinte saída:

```
drwxr-xr-x 2 root root  1024 abr 30 09:46 bin
drwxr-xr-x 2 root root  1024 abr 30 09:46 boot
drwxr-xr-x 2 root root  1024 abr 30 09:46 dev
drwxr-xr-x 2 root root  1024 abr 30 09:46 etc
drwxr-xr-x 2 root root  1024 abr 30 09:46 home
drwxr-xr-x 2 root root  1024 abr 30 09:46 lib
drwx------ 2 root root 12288 abr 30 09:39 lost+found
drwxr-xr-x 2 root root  1024 abr 30 09:46 proc
drwxr-xr-x 2 root root  1024 abr 30 09:46 root
drwxr-xr-x 2 root root  1024 abr 30 09:46 run
drwxr-xr-x 2 root root  1024 abr 30 09:46 sbin
drwxr-xr-x 2 root root  1024 abr 30 09:46 sys
drwxr-xr-x 2 root root  1024 abr 30 09:46 tmp
drwxr-xr-x 7 root root  1024 abr 30 09:46 usr
drwxr-xr-x 2 root root  1024 abr 30 09:46 var
```

Exercício: Dê permissões totais para a pasta `tmp` e somente para o usuário dono na pasta `root`.

```
> # chmod 777 tmp
> # chmod 600 root
```

Exercício: Copie seu kernel para a pasta `boot` e o executável do *busybox* para a pasta `usr/bin`.

Como vimos anteriormente, o *busybox* contém todas as ferramentas de usuário em um único executável. Porém, não é nada prático digitar *busybox* antes de **todo comando**. Por isso criaremos uma série de links simbólicos que ligam o nome de cada ferramenta oferecida pelo *busybox* ao seu nome "tradicional".

Exercício: execute o comando abaixo na dentro de `raiz_linux`.

```
for util in $(./usr/bin/busybox --list-full); do
 ln -s /usr/bin/busybox $util
done
```

Exercício: Cheque agora que as pastas `bin, sbin` e `usr/bin` estão cheias de executáveis com ferramentas tradicionais de linux. Se não estiverem houve algum problema.

Exercício: copie `hello-static` e `hello-dyn` para a pasta `root`.

Parte 5 - seu primeiro boot

Com isto já temos o mínimo necessário para dar *boot* no sistema, mas não teremos um sistema completamente funcional nem bem montado. A ideia aqui é testar nosso progresso e entender o que falta para esse sistema, que já tem *kernel* e *ferramentas de modo usuário*, funcionar de maneira plena. Faremos todas as melhorias no sistema na parte 6.

Exercício: Pesquise o que é um *boot loader* e cite a opção mais comum usada em sistemas linux.

A instalação de um *boot loader* é trabalhosa e cheia de possibilidades de erros. Podemos aproveitar o fato do próprio *QEmu* servir de *boot loader* para facilitar o desenvolvimento deste roteiro. Ao chamar o

Sempre que for usar o `qemu` não se esqueça de desvincular `/dev/loop0p1` de `raiz_linux` usando `umount`.

```
> # qemu-system-x86_64 -enable-kvm \
                       -kernel linux-5.6.8/arch/x86_64/boot/bzImage \
                       -append "quiet init=/bin/sh root=PARTUUID=ac18c11f-01" \
                       /dev/loop0
```

Vamos destrinchar essa chamada:

- `-enable-kvm`: habilita a virtualização por hardware, fazendo com que o sistema *guest* possa executar em velocidade quase real. Pode ser omitido, mas ficará mais lento.
- `-kernel linux-5.6.8/arch/x86_64/boot/bzImage`: instrui o *QEmu* a carregar o kernel presente no caminho passado.
- `-append "quiet init=/bin/sh root=PARTUUID=ac18c11f-01 vga="`: estas opções são passadas para o kernel e configuram sua execução
  - `quiet`: minimiza mensagens de debug
  - `init=/bin/sh`: aqui configuramos o processo de `pid=1`. Ele é o cara que dá `fork+exec` em todos os outros processos do sistema e que dá `wait` nos orfãos. Inicialmente usamos o shell, mas isso não é bom. Na próxima parte veremos por que.
  - `root=PARTUUID=ac18c11f-01`: sistema de arquivos raiz está na partição `01` do disco identificado pelo `UUID` que vocês obtiveram no `fdisk`
- `/dev/loop0`: disco a ser colocado na máquina virtual. O *Disk Identifier* dele (anotado anteriormente) está listado no item acima.

Execute o *QEmu*. Se tudo der certo você estará, em poucos segundos, em um prompt rodando como *root*.

Você conseguiu seu primeiro boot! Chame o professor para ganhar um parabéns e continue o roteiro.

Exercício: localize o programa `hello-static` compilado na seção 2 e execute-o. Se tudo funcionou tire um print ;)

Exercício: tente rodar o programa `hello-dyn`. Funciona? Você tem alguma ideia do por que?

Exercício: Vamos agora explorar um pouco mais as opções de máquinas virtuais criadas com *QEmu*. Escreva abaixo as opções de linha de comando usadas para alterar a quantidade de RAM e cores usados na VM.

Exercício: Qual é o mínimo de memória RAM em que o sistema criado nesta seção ainda liga? Vá testando até que o sistema não ligue mais corretamente.

Parte 6 - sistema `init`

Na última parte trabalhamos com . Se você explorou um pouco o sistema já deve ter notado que várias coisas não funcionam. Não conseguimos, por exemplo, escrever em nenhum arquivo. Vamos explorar dois casos mais interessantes:

Nosso sistema não está finalizado! Várias coisas ainda não funcionam e não deveriam funcionar mesmo!

Exercício: O comando `df` (*disk free*) é usado para listar o espaço livre em todos os discos presentes no sistema. Tente executá-lo no seu sistema. O quê acontece?

Exercício: O comando `lspci` (*list pci devices*) mostra todos os periféricos ligados diretamente na placa mãe do seu PC. Tente executá-lo no seu sistema. O quê acontece?

Exercício: Consulte as pastas apontadas nos itens anteriores. Elas tem conteúdo? Elas *deveriam* ter conteúdo?

Chegamos agora na importância do processo `init` (*pid=1*): ele é responsável por supervisionar a criação de todos os sistemas de arquivos especiais (`/proc, /sys, /dev/`) e por iniciar serviços essenciais para o

funcionamento do sistema. Da mesma maneira, ao finalizar ele é responsável por desligar todos os recursos de hardware de maneira segura.

Exercício: O comandou `mount` é usado para criar os diretórios especiais `/proc` e `/sys`. Rode os seguintes comandos e verifique que agora `df` e `lspci` funcionam corretamente.

```
> # mount -t proc proc /proc -o nosuid,noexec,nodev
> # mount -t sysfs sys /sys -o nosuid,noexec,nodev
```

Estes comandos fazem parte da inicialização normal de um sistema e expõe estruturas do kernel para o resto do sistema via arquivos. O *busybox* já nos fornece um sistema de inicialização bastante simplificado que, entre outras coisas, rodaria estes comandos automaticamente a todo boot. Aproveitaremos ele para três propósitos:

1. executar um script de inicialização que configure todos os diretórios especiais e serviços.
2. adicionar serviços que proveem uma tela de login
3. executar um script de finalização que desliga o hardware quando o PC for desligado.

Primeiro vamos copiar versões padrão de todos os arquivos de configuração necessários. Os seguintes arquivos estão na pasta `configs` do repositório da aula.

- `passwd, shadow, groups`: listam os usuários e grupos presentes. `shadows` contém hashes das senha.
- `profile`: é executado logo após um login correto. Pode ser usado para configurar o terminal.
- `issue`: contém o nome do seu sistema mostrado na tela de login.
- `hosts`: associa um nome com alguns IPs. É aqui que associamos *localhost* a `127.0.0.1`
- `hostname`: configura o nome da nossa máquina na rede.
- `fstab`: lista todos os discos que devem ser montados além do *rootfs*.

Exercício: copie estes arquivos para o `etc` do seu sistema.

O sistema de `init` disponibilizado pelo *busybox* lê o arquivo `/etc/inittab` e o interpreta de acordo com as regras mostradas no arquivo *busybox-1.30.1/examples/inittab*. Iremos usar o seguinte arquivo, já disponível em `configs/inittab`.

```
# /etc/inittab
::sysinit:/bin/echo "Iniciando..."
::sysinit:/etc/init.d/startup
tty1::respawn:/sbin/getty 38400 tty1
#tty2::respawn:/sbin/getty 38400 tty2
#tty3::respawn:/sbin/getty 38400 tty3
#::ctrlaltdel:/bin/umount -a -r
::shutdown:/bin/echo SHUTTING DOWN
::shutdown:/bin/umount -a -r
```

A primeira coluna do arquivo mostra o momento em que ela deve rodar. Vemos, por exemplo, que o script `/etc/init.d/startup` rodará ao inicializar o sistema e que toda vez que o processo `/sbin/getty` (terminal com login) terminar ele é reiniciado. Também existem scripts para serem rodados ao desligar o sistema.

Em especial, este arquivo `/etc/init.d/startup` (presente no repositório como `configs/init.d/startup`) contém comandos para configurar o sistema, incluindo os diretórios especiais que mostramos acima. Seu conteúdo é mostrado abaixo por completude.

```
# Monta os sistemas de arquivos especiais
mount -t proc proc /proc -o nosuid,noexec,nodev
mount -t sysfs sys /sys -o nosuid,noexec,nodev

# Configura detector de dispositivos
mkdir -p /dev/pts /dev/shm
mount -t tmpfs shm /dev/shm -o mode=1777,nosuid,nodev
mdev -s
echo /sbin/mdev > /proc/sys/kernel/hotplug

# Configura terminais
mount -t devpts devpts /dev/pts -o mode=0620,gid=5,nosuid,noexec
```

```
# Configura /run, que guarda algumas informações de execução.
mount -t tmpfs run /run -o mode=0755,nosuid,nodev

# Atribui nome ao PC
cat /etc/hostname > /proc/sys/kernel/hostname

# Monta todos os sistemas de arquivos contidos em /etc/fstab
mount -a

mount -o remount,rw /
```

Exercício: Este script requer a criação de um diretório `/run`. Para que ele serve? Crie-o e copie ambos arquivos acima para seu sistema.

Exercício: Agora vamos rodar de novo, desta vez com nosso novo sistema de inicialização configurado. Modifique sua linha de comando do *QEmu* e retire a porção `init=/bin/sh`. Por padrão o kernel buscará o executável `/sbin/init`, que usará os arquivos que criamos para inicializar o sistema.

> **!** Se tudo deu certo você deverá ter um prompt de login. Logue como *root* e continue o roteiro.

Exercício: Crie um arquivo dentro de seu sistema. Você pode usar o editor `vi` ou o comando `touch` para criar um arquivo vazio. Se não deu certo revise se ocorreu tudo certo na execução do seu script `startup` rolando a tela para cima com `Shift+PageUp`.

Exercício: Desligue seu sistema com `poweroff`. Ligue-o novamente e confira se o arquivo ainda está lá.

Parte 7 - bibliotecas e carregamento dinâmico

Todos os executáveis que conseguimos rodar até agora foram compilados estaticamente. Quando tentamos rodar `hello-dyn` tivemos um erro.

Exercício: Reveja, em sua resposta da Parte 2, o resultado do comando `ldd` no `hello-dyn`. Estes arquivos existem no seu sistema?

Ao montar nosso sistema do zero não incluimos nenhuma biblioteca! Logo, o nosso executável não consegue carregar as partes faltantes e não irá rodar. Felizmente, nosso sistema Linux possui a mesma arquitetura do Ubuntu instalado em nossas máquinas e podemos copiar os arquivos necessários para nosso sistema!

Exercício: Faça a cópia das bibliotecas dinâmicas para os locais apontados por `ldd` e rode de novo `hello-dyn`. Funcionou agora?