Introdução ao Desenvolvimento de uma Distribuição ARM do GNU/Linux

Tiago Natel de Moura

1 Introdução

Esse documento demonstrará os passos básicos para iniciar o desenvolvimento de uma distribuição Linux embedded.

2 Escolhendo um Emulador

Mesmo que você já possua o hardware que utilizará sua distribuição, é altamente recomendado utilizar uma máquina virtual para realizar os testes. O problema é que todo dia surgem novas arquiteturas embedded (ARM) e muitas vezes não temos emuladores 100% compatíveis.

Se voce necessitar desenvolver algo no nível do kernel só há duas opções: utilizar um emulador 100% compatível ou testar diretamente no seu hardware. Mas para desenvolver o rootfs de um sistema operacional não é necessário ter um emulador completo da arquitetura. Por exemplo, não temos um emulador para ARMv5 WM8505 ou VT8500, mas podemos usar um emulador para outra placa mas que use a mesma versão ARM do nosso processador. Ou seja, o código compilado nesse emulador também roda no nosso hardware se eles implementam a mesma versão do ARM.

Existem muitos emuladores para ARM, neste tutorial utilizarei o `Qemu`.

Para instalar o qemu para ARM é muito simples, se você utiliza uma distribuição debian-like utilize:

Listagem 1: Instalando qemu-system-arm no debian

```
[i4k@i4k ~]$ sudo apt-get install qemu-system-arm
```

Para o fedora, use:

Listagem 2: Instalando qemu-system-arm no fedora

```
[i4k@i4k ~]$ sudo yum install qemu-system-arm
```

Neste tutorial emularemos a placa VersatilePB.

3 Escolhendo o kernel

Essa escolha depende de muitos fatores. Para a maiorias das arquiteturas a escolha da última versão é a melhor alternativa. Mas quando trata-se de sistemas embedded, a escolha da versão é muito importante pois temos grandes limitações de hardware.

Claro que você sempre deve optar por alguma versão do kernel 2.6, pois esse teve importantes melhorias para sistemas embedded, mas a última versão dele nem sempre é uma boa alternativa, depende muito do seu hardware. Por exemplo, para o WM8505 a melhor alternativa é sempre algum fork da versão

2.6.29 disponibilizada pela Wondermedia. Mas para a maioria dos chips, se você pretende iniciar sua distribuição do zero, a melhor escolha é a última versão.

Você pode baixar o kernel diretamente no kernel.org. Nesse tutorial, usarei a versão 2.6.36.1, pois é a última versão estável disponivel.

4 Toolchain

Para compilar um código para outra arquitetura que não seja a sua, você necessita de uma **toolchain**, ou seja, um conjunto de ferramentas compiladas para uma plataforma específica. Essas ferramentas também costumam ser chamadas de **crosstools**. Existem muitas toolchains que voce pode escolher para compilar o seu kernel, aqui algumas das existentes:

- crosstool-ng - Ótima toolchain baseada na crosstool abaixo. Ideal para todas as distribuições Linux e para MACacOS.
- crosstool
- Emdebian Toolchain - Ideal para distribuições Debian-Like.

Neste tutorial usaremos o crosstool-ng. Para saber como configurar e compilar toolchains com o crosstool-ng veja esse outro tutorial:

http://cyberkids.googlecode.com/files/Instalando_e_configurando_Crosstool-ng.pdf

Depois de ter sua toolchain instalada, adicione o diretório contendo o binário das ferramentas ao seu PATH.

5 Compilando seu kernel

O kernel é desenvolvido para suportar a maior quantidade possivel de hardwares, portanto ele é altamente configurável. Quase todo módulo ou feature dele possue uma configuração, que pode ser tanto uma opção de habilitar/ desabilitar como ajuste de valores mais específicos.

Ele possue vários modos de configuração, sendo que o mais utilizado é o **menuconfig**, que nos dá uma interface ncurses para facilitar a configuração. Além desta, tem também o **xconfig** e **gconfig** que possibilitam configurar utilizando uma interface gráfica, o que é uma boa opção para iniciantes.

Esse menu de configuração no final gera um arquivo .config de configuração do tipo NAME=VALUE, muito simples e que pode inclusive ser editado na mão em situações mais avançadas. Para cada arquitetura, o kernel já traz uma grande quantidade de arquivos de configuração genéricos para vários hardwares. O que ajuda na hora de compilar seu kernel para um hw específico. Essas configurações default encontram-se em arch/$ARCH/configs/.

Para iniciar, podemos utilizar uma configuração básica do kernel. Descompacte seu kernel e vamos configurá-lo para o VersatilePB. Dentro de arch/arm/configs tem o arquivo versatile_defconfig, que possui as configurações default para essa placa. Copie ele para o diretório raiz do projeto:

Listagem 3: Pegando a configuração básica do VersatilePB

```
[i4k@i4k linux-2.6.36.1]$ make clean mrproper
[i4k@i4k linux-2.6.36.1]$ cp arch/arm/configs/versatile_defconfig .
    config
```

Abra o menuconfig:

Listagem 4: Configurando

```
[i4k@i4k linux-2.6.36.1]$ make menuconfig ARCH=arm
```

e carregue o nosso arquivo .config indo em "Load an Alternate Configuration File". Se você não fizer isso ele carregará as configurações default e quando você finalizar ele sobrescreverá seu arquivo .config com essa configuração ... Depois, vá em "Kernel Features" e habilite a opção "Use the ARM EABI to compile the kernel".

Para compilar o kernel, simplesmente faça:

Listagem 5: Compilando o kernel

```
[i4k@i4k linux-2.6.36.1]$ make ARCH=arm CROSS_COMPILE=arm-linux-
    gnueabi-
```

* Substitua **arm-linux-gnueabi-** pelo alias que você escolheu na hora de compilar sua toolchain.

Ele deverá compilar sem problemas, visto termos utilizados uma configuração padrão.

Ao final do build, deverá ter sido criado um arquivo **zImage** dentro de arch/arm/boot/. Esse é o nosso kernel compactado.

Para testá-lo, execute:

Listagem 6: Testando o kernel

```
[i4k@i4k linux-2.6.36.1]$ qemu-system-arm -M versatilepb -m 128 \
                          -kernel "arch/arm/boot/zImage" \
                          -append "root=/dev/ram"
```

Ele deverá fazer boot e logo em seguida ocorrer um kernel panic com uma mensagem igual ou parecida com essa:

VFS: Unable to mount root fs on unknown-block(1,0)

Isso é porque ainda não temos um **INITRAMFS**.

6 Initial Ram Disk (A.K.A "INITRD")

Depois que o bootloader carrega o kernel e passa o controle da execução para ele, o kernel inicializa suas estruturas (C struct's), detecta os componentes do hardware, ativa drivers, etc ... Antes dele entrar no ambiente de user-space ele precisa prover um sistema de arquivos.

Mas para montar a raiz do sistema o kernel precisa de duas coisas: Ele precisa conhecer a mídia onde encontra-se o sistema de arquivos e ele precisa de drivers para acessar essa mídia... Isso implica que o kernel seja compilado com suporte a todos os drivers, o que aumenta consideravelmente o tamanho dele e cria várias outras complicações.

Pra resolver esse problema, foi criado o mecanismo de initrd.

Com initrd, o bootloader carrega um disco na memória (RAM disk). O kernel monta esse disco como a raiz do sistema e pode rodar programas dele. Depois, um novo sistema de arquivos raiz (real) é montado de um algum dispositivo. A raiz anterior (initrd) é movida para um diretório e depois pode ser desmontada.

Os passos do boot usando initrd pode ser visto abaixo:

* Retirado da documentação do kernel em /Documentation/initrd.txt

1. O bootloader carrega o kernel e o initrd.
2. O kernel converte initrd num RAM disk e libera a memória utilizada por initrd.
3. Se o parametro root do kernel *NÃO* for /dev/ram0, é utilizado a função (DEPRECATED) change_root.
4. O dispositivo "root" é montado. Se ele for /dev/ram0, a imagem do initrd é montada como root.
5. /sbin/init é executado (Ele pode ser qualquer executável válido, incluindo shell scripts; Ele roda com uid 0).
6. init monta o sistema de arquivos raiz *real*.
7. init coloca a raiz do sistema de arquivos no diretório raiz usando a chamada de sistemas pivot_root.
8. init executa /sbin/init do novo sistema de arquivos, iniciando a sequencia de boot.
9. O sistema de arquivos do initrd é removido.

Versões recentes do kernel suportam criar uma RAM disk a partir de um arquivo cpio comprimido. Facilitando assim a sua criação. Para utilizar esse formato, simplesmente crie um diretório no seu disco com o conteúdo desejado do initrd e rode:

Listagem 7: Criando uma arquivo cpio compactado.

```
[i4k@i4k linux-2.6.36.1]$ find . | cpio --quiet -H newc -o | gzip
    -9 -n > initrd.img.gz
```

7 Hello World

Vamos criar um initrd básico para vermos isto funcionando, o nosso sistema de arquivos terá apenas um único executável que será chamado pelo kernel quando entrar em user-space. Crie um diretório para o seu initrd e compile o código abaixo com sua toolchain:

Listagem 8: Hello World, Linux

```
#include <stdio.h>

void main()
{
```

```
    printf("\n\n\nHello World\n\n");
    while(1);
}
```

Listagem 9: Compilando

```
[i4k@i4k linux-2.6.36.1]$ cd initrd
[i4k@i4k initrd]$ ls
hello.c
[i4k@i4k initrd]$ arm-linux-gnueabi-gcc -static hello.c -o hello
```

Crie seu initrd:

Listagem 10: Gerando o initrd

```
[i4k@i4k initrd]$ echo hello | cpio -o -H newc | gzip -9 -n >
    initrd.img.gz
```

E pronto, podemos testar:

Listagem 11: Emulando nosso sistema

```
[i4k@i4k linux-2.6.36.1]$ qemu-system-arm -M versatilepb -m 128M \
                            -kernel arch/arm/boot/zImage \
                            -initrd initrd/initrd.img.gz \
                            -append "root=/dev/ram rdinit=/hello"
```

Se tudo correu bem, voce deve ter visto as mensagens de boot comuns e lá embaixo o seu "Hello World".

Se você receber um kernel panic, verifique com atenção os passos anteriores e certifique-se de que compilou o kernel com suporte a EABI.

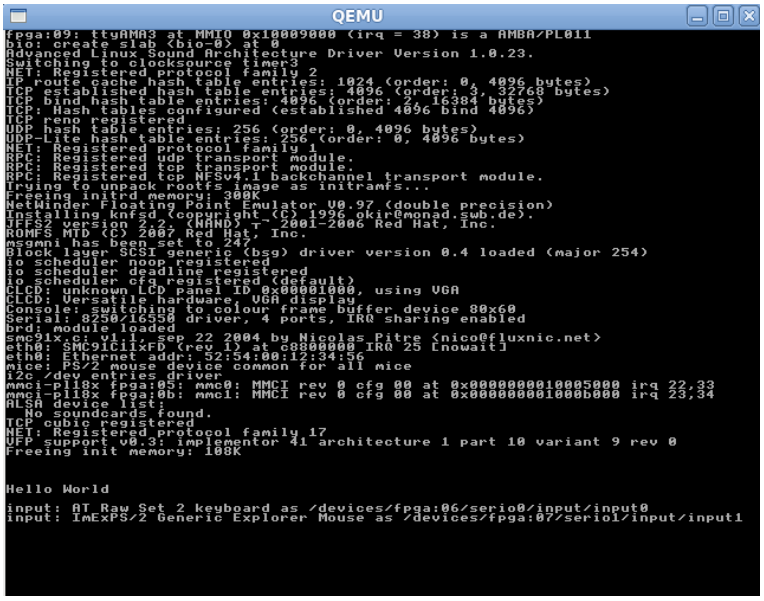

Figure 1: Resultado esperado