

# Desenvolvimento de Linux Embarcado

## Técnicas de Programação para Sistemas Embarcados II



UNIVERSIDADE  
FEDERAL DO CEARÁ  
CAMPUS QUIXADÁ

Prof. Francisco Helder

Universidade Federal do Ceará

September 5, 2022



# História

---

- O kernel do Linux é um componente de um sistema, que também requer bibliotecas e aplicativos para fornecer recursos aos usuários.
- O kernel Linux foi criado como hobby em 1991 por um estudante finlandês, Linus Torvalds.
  - O Linux rapidamente começou a ser usado como kernel para sistemas operacionais de software livre
- Linus Torvalds conseguiu criar uma grande e dinâmica comunidade de desenvolvedores e usuários em torno do Linux.
- A partir de 2022, cerca de 2.000 pessoas contribuem para cada versão do kernel, indivíduos ou empresas grandes e pequenas.

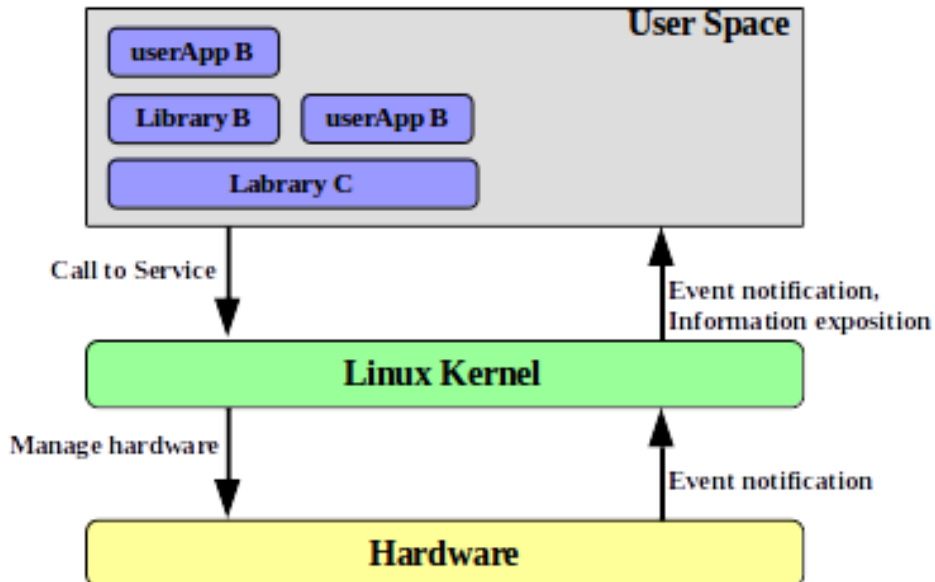


Linus Torvalds 2014

## Funcionalidades:

- Portabilidade e suporte de hardware. Roda na maioria das arquiteturas (veja **arch/** no código fonte).
- Escalabilidade. Pode ser executado em supercomputadores, bem como em dispositivos minúsculos (4 MB de RAM é suficiente).
- Conformidade com padrões e interoperabilidade.
- Suporte de rede exaustivo.
- Segurança. Não pode esconder suas falhas. Seu código é revisado por muitos especialistas.
- Estabilidade e confiabilidade.
- Modularidade. Pode incluir apenas o que um sistema precisa, mesmo em tempo de execução.
- Fácil de programar. Você pode aprender com o código existente. Muitos recursos úteis na rede.

## Kernel Linux: Visão Geral



# Principais Funções do Kernel do Linux

---

- Gerenciar todos os recursos de hardware: CPU, memória, I/O.
- Fornecer um conjunto de APIs, independente da arquitetura e do hardware que permita aplicações em User Space e bibliotecas use os recursos de hardware.
- Trata acessos concorrentes e uso de recursos de hardware a partir de diferentes aplicativos simultâneos.

## Exemplo:

uma única interface de rede é usado por vários aplicativos em User Space através de várias conexões de rede. O kernel é responsável por “multiplex” o recurso de hardware

# Chamada de Sistema

---

- A principal interface entre o espaço de kernel e de usuário é o conjunto de chamadas de sistema.
- Cerca de 400 chamadas de sistema que fornecem os principais serviços do kernel.
  - Operações com arquivo e dispositivos, operações de rede, comunicação entre processos, gerenciamento de processos, mapeamento de memória, timers, threads, sincronização primitivas, etc.
- Esta interface é estável ao longo do tempo: apenas novas chamadas de sistema podem ser adicionados pelos desenvolvedores do kernel.
- Esta interface chamada de sistema é tratada pela biblioteca C, e os aplicações em User Space geralmente nunca fazer uma chamada de sistema diretamente, mas sim usa uma função da biblioteca C correspondente.



## Pseudo Filesystem

---

Linux dispõe informações do sistema e kernel em User Space através do **pseudo filesystems**, também chamado de **virtuais filesystems**.

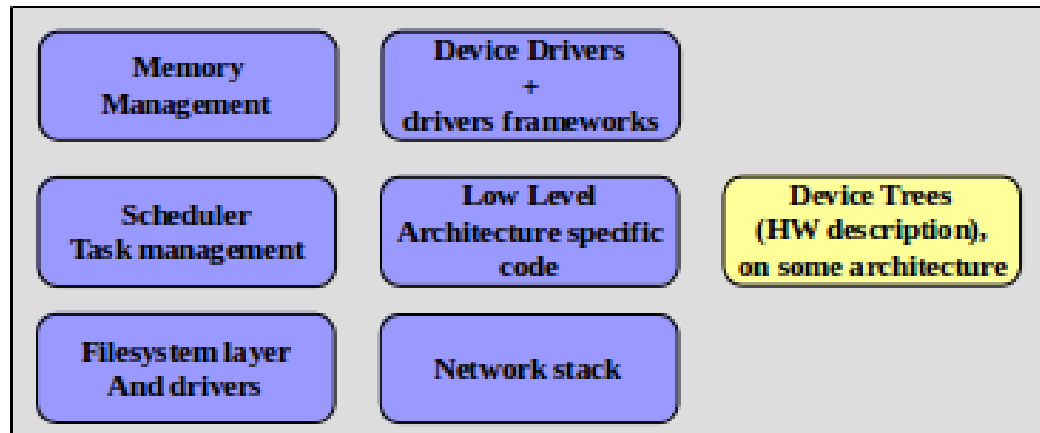
Pseudo filesystems permiti aplicações vejam os diretórios e arquivos que não existem em armazenamento real: eles são criados e atualizados em tempo real pelo kernel.


**Os dois pseudo filesystems mais importantes são:**


- **proc**, geralmente montado em `/proc`:  
Informações relacionadas ao sistema operacional (processos, os parâmetros de gerenciamento de memória ...)
- **sysfs**, geralmente montado em `/sys`:  
Representação do sistema como um conjunto de dispositivos e barramentos.  
Informações sobre estes dispositivos.

## Dentro do Kernel do Linux

---



 Implemented mainly in C,  
A little bit of assembly

 Written in a Device Tree  
specific language



# Licença Linux

---

Todas as fontes do Linux são Software Livre lançado sob a GNU General Public License versão 2 (GPL v2).

Para o kernel Linux, isso basicamente implica que:

- Ao receber ou comprar um dispositivo com Linux, você tem o direito de obter os fontes do Linux, com direito de estudá-los, modificá-los e redistribuí-los.
- Ao produzir dispositivos baseados em Linux, esteja preparado para liberar os fontes para o destinatário, com os mesmos direitos, sem restrições.

# Arquiteturas de Hardware Suportadas

---

Veja o diretório **arch/** nas fontes do kernel

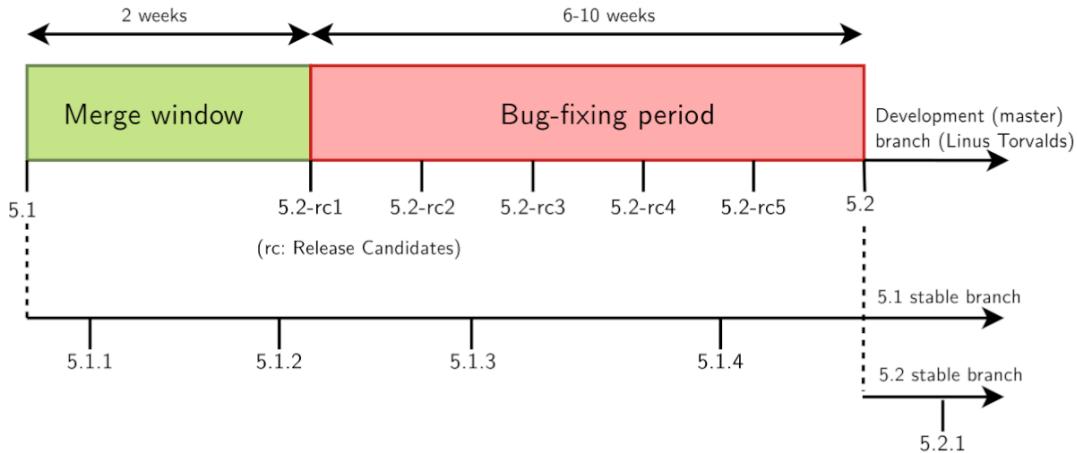
- Mínimo: processadores de 32 bits, com ou sem MMU, suportados por **gcc** ou **clang**
- Arquiteturas de 32 bits (**arch/subdiretórios**)  
Exemplos: **arm**, **arc**, **m68k**, **microblaze** (soft core em FPGA)...
- Arquiteturas de 64 bits  
Exemplos: **alpha**, **arm64**, **ia64**...
- Arquiteturas de 32/64 bits  
Exemplos: **mips**, **powerpc**, **riscv**, **sh**, **sparc**, **x86**...
- Observe que arquiteturas não mantidas também podem ser removidas quando apresentam problemas de compilação e ninguém as corrige.
- Encontre detalhes nas fontes do kernel: **arch/<arch>/Kconfig**, **arch/<arch>/README** ou **Documentation/<arch>/**

## Versionamento Linux

---

- Até 2003, havia um novo ramo de lançamento “estabilizado” do Linux a cada 2 ou 3 anos (2.0, 2.2, 2.4). As ramificações de desenvolvimento levaram de 2 a 3 anos para serem mescladas (muito lentas!).
- Desde 2003, há um novo lançamento oficial do Linux a cada 10 semanas:
  - Versões **2.6** (dezembro de 2003) a **2.6.39** (maio de 2011)
  - Versões **3.0** (julho de 2011) a **3.19** (fevereiro de 2015)
  - Versões **4.0** (abril de 2015) a **4.20** (dezembro de 2018)
  - A versão **5.0** foi lançada em março de 2019.
- Os recursos são adicionados ao kernel de forma progressiva. Desde 2003, os desenvolvedores do kernel conseguiram fazer isso sem ter que introduzir uma ramificação de desenvolvimento massivamente incompatível.
- Para cada versão, há correções de bugs e atualizações de segurança chamadas versões estáveis: 5.0.1, 5.0.2, etc.

# Modelo de Desenvolvimento Linux



## Localização Oficial dos Fontes do Kernel

### As versões principais do kernel Linux, conforme lançadas por Torvalds

- Essas versões seguem o modelo de desenvolvimento do kernel
- Eles podem não conter os últimos desenvolvimentos de uma área específica ainda
- Uma boa escolha para a fase de desenvolvimento de produtos
- <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git>

### As versões estáveis do kernel Linux, mantidas por um grupo de mantenedores

- Essas versões não trazem novidades em relação à árvore de Linus
- Somente correções de bugs e correções de segurança são puxadas para lá
- Cada versão é estabilizada durante o período de desenvolvimento do próximo kernel
- Certas versões podem ser mantidas por muito mais tempo, mais de 2 anos
- Uma boa escolha para fase de comercialização de produtos
- <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git>

# Localização Não-Oficial dos Fontes do Kernel

---

- Muitos fornecedores de chips fornecem suas próprias fontes de kernel
  - Concentrando-se primeiro no suporte de hardware
  - Pode ter um delta muito importante com mainline Linux
  - Às vezes, eles interrompem o suporte para outras plataformas sem se importar
  - Útil nas fases iniciais apenas quando a linha principal ainda não alcançou (muitos fornecedores investem no kernel principal ao mesmo tempo)
  - Adequado para PoC, não adequado para produtos a longo prazo, pois geralmente não são fornecidas atualizações para esses kernels
  - Ficar preso a um sistema obsoleto com software quebrado que não pode ser atualizado tem um custo real no final
- Muitas subcomunidades de kernel mantêm seu próprio kernel, com recursos geralmente mais novos, mas menos estáveis, apenas para desenvolvimento de ponta
  - Comunidades de arquitetura (ARM, MIPS, PowerPC, etc)
  - Comunidades de drivers de dispositivos (I2C, SPI, USB, PCI, rede, etc)
  - Outras comunidades (em tempo real, etc.)
  - Não adequado para uso em produtos

# Configuração do Kernel

---

- O kernel contém milhares de drivers de dispositivo, drivers de sistema de arquivos, protocolos de rede e outros itens configuráveis
- Milhares de opções estão disponíveis, que são usadas para compilar seletivamente partes do código-fonte do kernel
- A configuração do kernel é o processo de definir o conjunto de opções com as quais você deseja que seu kernel seja compilado
- O conjunto de opções depende
  - Da arquitetura de destino e em seu hardware (para drivers de dispositivo, etc.)
  - Sobre as capacidades que você gostaria de dar ao seu kernel (capacidades de rede, sistemas de arquivos, tempo real, etc.). Essas opções genéricas estão disponíveis em todas as arquiteturas.

# Configuração do Kernel e Sistema de Build

---

- A configuração do kernel e o sistema de compilação são baseados em vários Makefiles
- Interage apenas com o Makefile principal, presente no diretório superior da árvore de fontes do kernel
- A interação ocorre
  - usando a ferramenta **make**, que analisa o Makefile
  - através de vários **targets**, definindo qual ação deve ser feita (configuração, compilação, instalação, etc.).
  - Execute **make help** para ver todos os alvos disponíveis.

1  
2  
3  
4

```
Exemplo:  
$ cd linux  
$ make <target>
```



# Especificando a Arquitetura

---

Primeiro, especifique a arquitetura para o kernel construir

- 1 Defina ARCH para o nome de um diretório em **arch/**:

```
1 $ export ARCH=arm
2
```

- 2 Por padrão, o sistema de compilação do kernel assume que o kernel está configurado e construído para a arquitetura do host (**x86** em nosso caso, compilação nativa do kernel)
- 3 O sistema de compilação do kernel usará essa configuração para:
  - Uso das opções de configuração para a arquitetura.
  - Compila o kernel com fontes e cabeçalhos para a arquitetura.

## Escolhendo um Compilador

---

O compilador invocado pelo Makefile do kernel é **\$(CROSS\_COMPILE)gcc**

- A especificação do compilador já é necessária no momento da configuração, pois algumas opções de configuração do kernel dependem dos recursos do compilador.
- Ao compilar nativamente
  - Deixe **CROSS\_COMPILE** indefinido e o kernel será compilado nativamente para a arquitetura do host usando **gcc**.
- Ao usar um compilador cruzado
  - Para fazer a diferença com um compilador nativo, os executáveis entre compiladores são prefixados pelo nome do sistema de destino, arquitetura e, às vezes, biblioteca.  
Exemplos: **mips-linux-gcc**: o prefixo é **mips-linux-** **arm-linux-gnueabi-gcc**: o prefixo é **arm-linux-gnueabi-**
- Assim, você pode especificar seu compilador cruzado da seguinte forma:  
**export CROSS\_COMPILE=arm-linux-gnueabi-**

**CROSS\_COMPILE** é na verdade o prefixo das ferramentas de compilação cruzada (**gcc**, **as**, **ld**, **objcopy**, **strip...**).

# Especificando ARCH e CROSS\_COMPILE

- 1 Informe o **ARCH** e **CROSS\_COMPILE** na linha de comando do **make**:

```
1 $ make ARCH=arm CROSS_COMPILE=arm-linux- ...  
2
```

## Desvantagem:

é fácil esquecer as variáveis quando você executa qualquer comando **make**, fazendo com que sua compilação e configuração sejam prejudicadas.

- 2 Defina **ARCH** e **CROSS\_COMPILE** como variáveis de ambiente:

```
1 $ export ARCH=arm  
2 $ export CROSS_COMPILE=arm-linux-  
3
```

## Desvantagem:

só funciona dentro do shell ou terminal atual, podendo colocar essas configurações em seu arquivo `/.bashrc` para torná-las permanentes e visíveis a partir de qualquer terminal.

# Configurações Iniciais

Difícil encontrar qual configuração do kernel funcionará com seu hardware e sistema de arquivos raiz. Comece com um que funcione!

- Caso de desktop ou servidor:

- Aconselhável começar com a configuração do seu kernel em execução:

```
1 $ cp /boot/config-`uname -r`.config
2
```

- Caixa de plataforma embutida:

- Configurações padrão armazenadas na árvore como arquivos de configuração mínima (apenas listando as configurações que são diferentes dos padrões) em **arch/<arch>/configs/**
- **make help** listará as configurações disponíveis para sua plataforma
- Para carregar um arquivo de configuração padrão, basta executar **make foo\_defconfig** (vai apagar seu .config atual!)
- > No ARM de 32-bits, geralmente há uma configuração padrão por família de CPU
- > No ARM de 64-bits, há apenas uma grande configuração padrão para personalizar

# Crie sua Própria Configuração Padrão

---

- 1 Use uma ferramenta como **make menuconfig** para fazer alterações na configuração
- 2 Salvar suas alterações substituirá seu **.config** (não rastreado pelo Git)
- 3 Quando estiver satisfeito com isso, crie seu próprio arquivo de configuração padrão:
  - Crie um arquivo de configuração mínima (configurações não padrão):

```
1 $ make saveefconfig
2
```

- Salve esta configuração padrão no diretório correto:

```
1 $ mv defconfig arch/<arch>/configs/myown_defconfig
2
```

- 4 Dessa forma, você pode compartilhar uma configuração de referência dentro das fontes do kernel e outros desenvolvedores agora podem obter o mesmo **.config** que você executando **make myown\_defconfig**

## Built-in ou Module?

---

**kernel image** é um **arquivo único**, resultante da vinculação de todos os arquivos objeto que correspondem às funcionalidades habilitadas na configuração

- Este é o arquivo que é carregado na memória pelo bootloader
- Todos os **built-in** estão, portanto, disponíveis assim que o kernel é iniciado

Alguns recursos (drivers de dispositivo, sistemas de arquivo, etc.) podem ser compilados como **módulos**

- Estes são plugins que podem ser carregados/descarregados dinamicamente no kernel
- Cada **módulo é armazenado como um arquivo separado no sistema de arquivo** e, portanto, o acesso a um sistema de arquivos é obrigatório para usar módulos
- Isso não é possível no procedimento de inicialização inicial do kernel, porque nenhum sistema de arquivo está disponível

# Detalhes da Configuração do Kernel

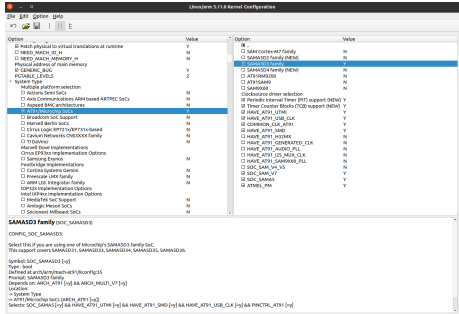
---

- A configuração é armazenada no arquivo **.config** na raiz dos fontes do kernel
  - Arquivo de texto simples, **CONFIG\_PARAM=valor**
  - As opções são agrupadas por seções e são prefixadas com **CONFIG\_**
  - Incluído pelo Makefile principal do kernel
  - Normalmente não editado manualmente devido às dependências

```
1      #
2      # CD-ROM/DVD Filesystems
3      #
4      CONFIG_ISO9660_FS=m
5      CONFIG_JOLIET=y
6      CONFIG_ZISOFS=y
7      CONFIG_UDF_FS=y
8      # end of CD-ROM/DVD Filesystems
9      #
10     # DOS/FAT/EXFAT/NT Filesystems
11     #
12     CONFIG_FAT_FS=y
13     CONFIG_MSDFS_FS=y
14     # CONFIG_VFAT_FS is not set
15     CONFIG_FAT_DEFAULT_CODEPAGE=437
16     # CONFIG_EXFAT_FS is not set
17
```

```
$ make xconfig
```

- A interface gráfica mais comum para configurar o kernel.
- Navegador de arquivos: fácil de carregar arquivos de configuração
- Interface de pesquisa para procurar parâmetros ([Ctrl] + [f])
- Pacotes DebianUbuntu necessários:  
qt5-default (qtbase5-dev no Ubuntu 22.04)

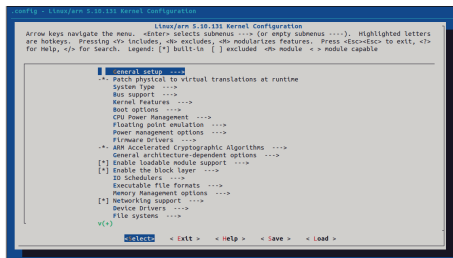




# menuconfig

```
$ make menuconfig
```

- Útil quando não há gráficos disponíveis.  
Interface muito eficiente.
- Mesma interface encontrada em outras ferramentas:  
BusyBox, Buildroot...
- Atalhos numéricos convenientes para pular  
diretamente para os resultados da pesquisa.
- Pacotes Debian/Ubuntu necessários: libncurses-dev



# Opções de Configuração do Kernel

Você pode alternar de uma ferramenta para outra, todas carregam/savam o mesmo arquivo **.config** e mostram o mesmo conjunto de opções

Compiled as a module:  
CONFIG\_ISO9660\_FS=m

Additional driver options:

CONFIG\_JOLIET=y

CONFIG\_ZISOFS=y

Statically built:  
CONFIG\_UDF\_FS=y

☐ ISO 9660 CDROM file system support  
☒ Microsoft Joliet CDROM extensions  
☒ Transparent decompression extension  
☐ UDF file system support

<M> ISO 9660 CDROM file system support  
[\*] Microsoft Joliet CDROM extensions  
[\*] Transparent decompression extension  
<\*> UDF file system support

Values in resulting .config file

Parameter values as displayed by xconfig

Parameter values as displayed by menuconfig

# Compilação Kernel

```
$ make
```

- Funciona apenas no diretório raiz do kernel
- Não deve ser executado como usuário privilegiado
- Execute vários trabalhos em paralelo. Nosso conselho: **n\_cpus \* 2** para usar totalmente a CPU. Exemplo: `make -j8`
- Para recompilar mais rápido (7x de acordo com alguns benchmarks), use o cache do compilador **ccache**:

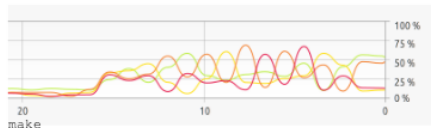
```
$ export CROSS_COMPILE="ccache arm-linux-"
```

## Benefits of parallel compile jobs (`make -j<n>`)

Tests on Linux 5.11 on arm

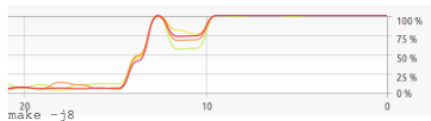
`make allnoconfig configuration`

`gnome-system-monitor` showing the load on 4 threads / 2 CPUs



Command: `make`

Total time: 129 s



Command: `make -j8`

Total time: 67 s

## Resultados da Compilação do Kernel

---

- **arch/<arch>/boot/Image**, imagem de kernel não compactada que pode ser inicializada
- **arch/<arch>/boot/\*Image\***, imagens de kernel compactadas que também podem ser inicializadas
  - **bzImage** para x86
  - **zImage** para ARM
  - **Image.gz** para RISC-V
  - **vmlinux.bin.gz** para ARC
- **arch/<arch>/boot/dts/\*.dtb**, compilado para Device Tree Blobs
- Todos os módulos do kernel, espalhados pela árvore de origem do kernel, como arquivos **.ko** (Kernel Object)
- **vmlinux**, uma imagem de kernel não compactada bruta no formato ELF, útil para fins de depuração, mas geralmente não usada para fins de inicialização

# Kernel Instalação: Caso Nativo

---

A instalação para o sistema host padrão

```
1 $ sudo make install
2
```

- Instalação:

Imagem compactada do kernel, igual ao do **arch/<arch>/boot**

```
1 $ /boot/vmlinuz-<version>
2
```

Armazena endereços de símbolos do kernel para fins de depuração (obsoleto: tal informação é geralmente armazenado no próprio kernel)

```
1 $ /boot/System.map-<version>
2
```

Configuração do kernel para esta versão

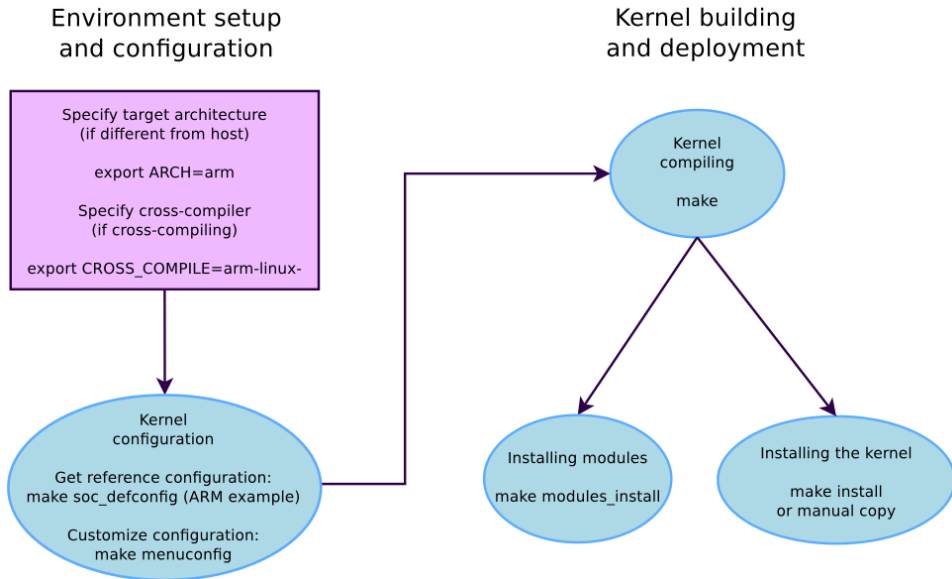
```
1 $ /boot/config-<version>
2
```

## Kernel Instalação: Caso Embedded

---

- **make install** raramente é usado em desenvolvimento embarcado, pois a imagem do kernel é um arquivo único, fácil de manusear.
- Outra razão é que não há uma maneira padrão de implantar e usar a imagem do kernel.
- Portanto, a disponibilização da imagem do kernel para o destino geralmente é manual ou feita por meio de scripts em sistemas de compilação.
- No entanto, é possível personalizar o comportamento `make install` em `arch/<arch>/boot/install.sh`

# Resumo da Construção do Kernel



## Inicializando com U-boot

---

O U-Boot pode inicializar diretamente o binário **zImage**.

Além da imagem do kernel, o U-Boot também deve passar um DTB para o kernel.

O processo de inicialização típico é, portanto:

- 1 Carregar zImage no endereço X na memória
- 2 Carregue <board>.dtb no endereço Y na memória
- 3 Inicie o kernel com bootz X - Y

O “-” no meio indica que não há initramfs



# Linha de Comando Kernel

---

- Além da configuração de tempo de compilação, o comportamento do kernel pode ser ajustado sem recompilação usando o **kernel command line**
- A linha de comando do kernel é uma string que define vários argumentos para o kernel
  - É muito importante para a configuração do sistema
  - **root=** para o sistema de arquivo
  - **console=** para o destino das mensagens do kernel

```
1 Exemplo: console=ttyS0 root=/dev/mmcblk0p2 rootwait
2
```

- Existem muitos mais. Os mais importantes estão documentados em **admin-guide/kernel-parameters** na documentação do kernel.

## Passando comandos para o kernel

---

- U-Boot carrega a string que passa comandos para o kernel em sua variável de ambiente **bootargs**
- Logo antes de iniciar o kernel, ele irá armazenar o conteúdo dos **bootargs** no **chosen** do Device Tree
- O kernel se comportará de forma diferente dependendo de sua configuração:
  - Se **CONFIG\_CMDLINE\_FROM\_BOOTLOADER** estiver definido: O kernel usará apenas a string do bootloader
  - Se **CONFIG\_CMDLINE\_FORCE** estiver definido: O kernel usará apenas a string recebida no momento da configuração em **CONFIG\_CMDLINE**
  - Se **CONFIG\_CMDLINE\_EXTEND** estiver definido: o kernel concatenará as strings

# Prática de lab - Kernel Linux

---



Hora de começar o laboratório prático 03!

- ① Configurar o ambiente de compilação
- ② Configure e faça a compilação cruzada do kernel para uma plataforma ARM
- ③ Nesta plataforma, configure o bootloader para inicializar seu kernel