

# Linux device drivers



Embedded Labworks

Por Sergio Prado. São Paulo, Março de 2017

® Copyright Embedded Labworks 2004-2017. All rights reserved.



# SOBRE ESTE DOCUMENTO

- × Este documento é baseado no material de treinamento disponibilizado pela Free Electrons em:  
<http://free-electrons.com/doc/training/linux-kernel>
- × Este documento é disponibilizado sob a Licença Creative Commons BY-SA 3.0.  
<http://creativecommons.org/licenses/by-sa/3.0/legalcode>



cc creative  
commons

- × Os fontes deste documento estão disponíveis em:  
<http://e-labworks.com/treinamentos/drivers/source>





# SOBRE O INSTRUTOR

- ✗ Sergio Prado tem mais de 20 anos de experiência em desenvolvimento de software para sistemas embarcados, em diversas arquiteturas de CPU (ARM, PPC, MIPS, x86, 68K), atuando em projetos com Linux embarcado e sistemas operacionais de tempo real.
- ✗ É sócio da Embedded Labworks, onde atua com consultoria, treinamento e desenvolvimento de software para sistemas embarcados:  
<http://e-labworks.com>
- ✗ Mantém um blog sobre Linux e sistemas embarcados:  
<http://sergioprado.org>





# AGENDA DO TREINAMENTO

- ✗ DIA 1: Introdução ao kernel Linux, configuração e compilação do kernel, módulos do kernel, driver de dispositivo de caractere, gerenciamento de memória, hardware I/O.
- ✗ DIA 2: Frameworks do kernel, gpiolib, driver model, device tree.
- ✗ DIA 3: Gerenciamento de processos, trabalhando com interrupções, mecanismos de sincronização, kernel debugging.





# PRÉ-REQUISITOS

- ✗ Usuário de distribuições GNU/Linux.
- ✗ Terminal de comandos do Linux (ls, cp, cat, grep, find, vi, etc).
- ✗ Conhecimentos intermediários em linguagem C.
- ✗ Conhecimento intermediários de hardware e sistemas embarcados (GPIO, PWM, I2C, SPI, etc).
- ✗ Conhecimentos básicos de sistemas com Linux embarcado.





# DURANTE O TREINAMENTO

- ✗ Pergunte...
- ✗ Expressse seu ponto de vista...
- ✗ Troque experiências...
- ✗ Ajude...
- ✗ Participe!





# AMBIENTE DE LABORATÓRIO

/opt/labs/  
dl/

docs/  
guides/  
hardware/  
training/  
ex/  
tools/

Ambiente de laboratório  
Aplicações e pacotes open-source  
que serão usados durante as  
atividades de laboratório  
Documentação  
Guias e livros de consulta  
Documentação do hardware  
Slides e atividades de laboratório  
Exercícios de laboratório  
Ferramentas de uso geral





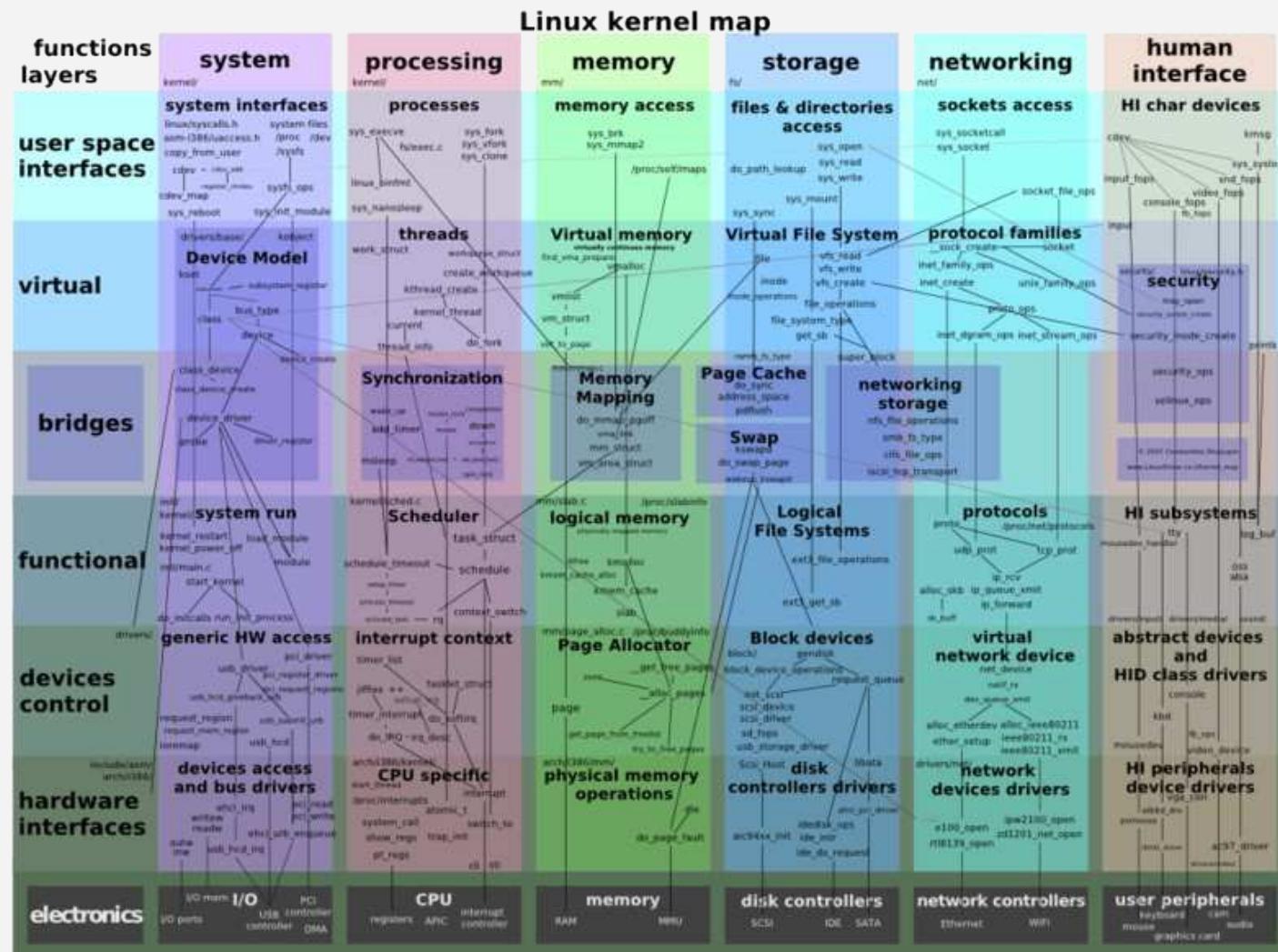
Embedded Labworks

# Linux device drivers

Introdução ao kernel Linux



# LINUX, SIMPLES ASSIM... :)





# HISTÓRICO

- ✗ O kernel Linux é um dos componentes do sistema operacional, que requer bibliotecas e aplicações para prover funcionalidades aos usuários.
- ✗ Foi criado em 1991 por um estudante finlandês, Linus Torvalds, e começou a ser usado rapidamente como sistema operacional em projetos de software livre.
- ✗ Linus foi capaz de criar uma comunidade grande e dinâmica de desenvolvedores e usuários ao redor do projeto.
- ✗ Atualmente, centenas de pessoas e empresas contribuem com o Linux.





# COLABORAÇÃO

**Most active 4.2 employers****By changesets****By lines changed**

Intel	1665	12.3%	AMD	438094	36.8%
Red Hat	1639	12.1%	Intel	96331	8.1%
(Unknown)	884	6.5%	Red Hat	62959	5.3%
(None)	884	6.5%	(None)	46140	3.9%
Samsung	681	5.0%	(Unknown)	41886	3.5%
SUSE	496	3.7%	Atmel	34942	2.9%
Linaro	449	3.3%	Samsung	29326	2.5%
(Consultant)	412	3.0%	Linaro	22714	1.9%
IBM	391	2.9%	Cisco	21170	1.8%
AMD	286	2.1%	SUSE	18891	1.6%
Google	246	1.8%	Code Aurora Forum	18435	1.5%
Renesas Electronics	203	1.5%	Mellanox	18044	1.5%
Free Electrons	203	1.5%	(Consultant)	15234	1.3%
Texas Instruments	191	1.4%	IBM	15095	1.3%
Facebook	176	1.3%	Cavium Networks	14580	1.2%
Oracle	163	1.2%	Free Electrons	13640	1.1%
Freescale	156	1.2%	Unisys	13428	1.1%
ARM	145	1.1%	Linux Foundation	12617	1.1%
Cisco	142	1.0%	MediaTek	11856	1.0%
Broadcom	138	1.0%	Google	11811	1.0%





# PRINCIPAIS CARACTERÍSTICAS

- ✗ Livre de royalties.
- ✗ Muitos recursos disponíveis na Internet.
- ✗ Extremamente portável: suporte para mais de 25 arquiteturas e milhares de dispositivos de hardware.
- ✗ Modular e escalável: o mesmo kernel roda em relógios, celulares e servidores da bolsa de valores!





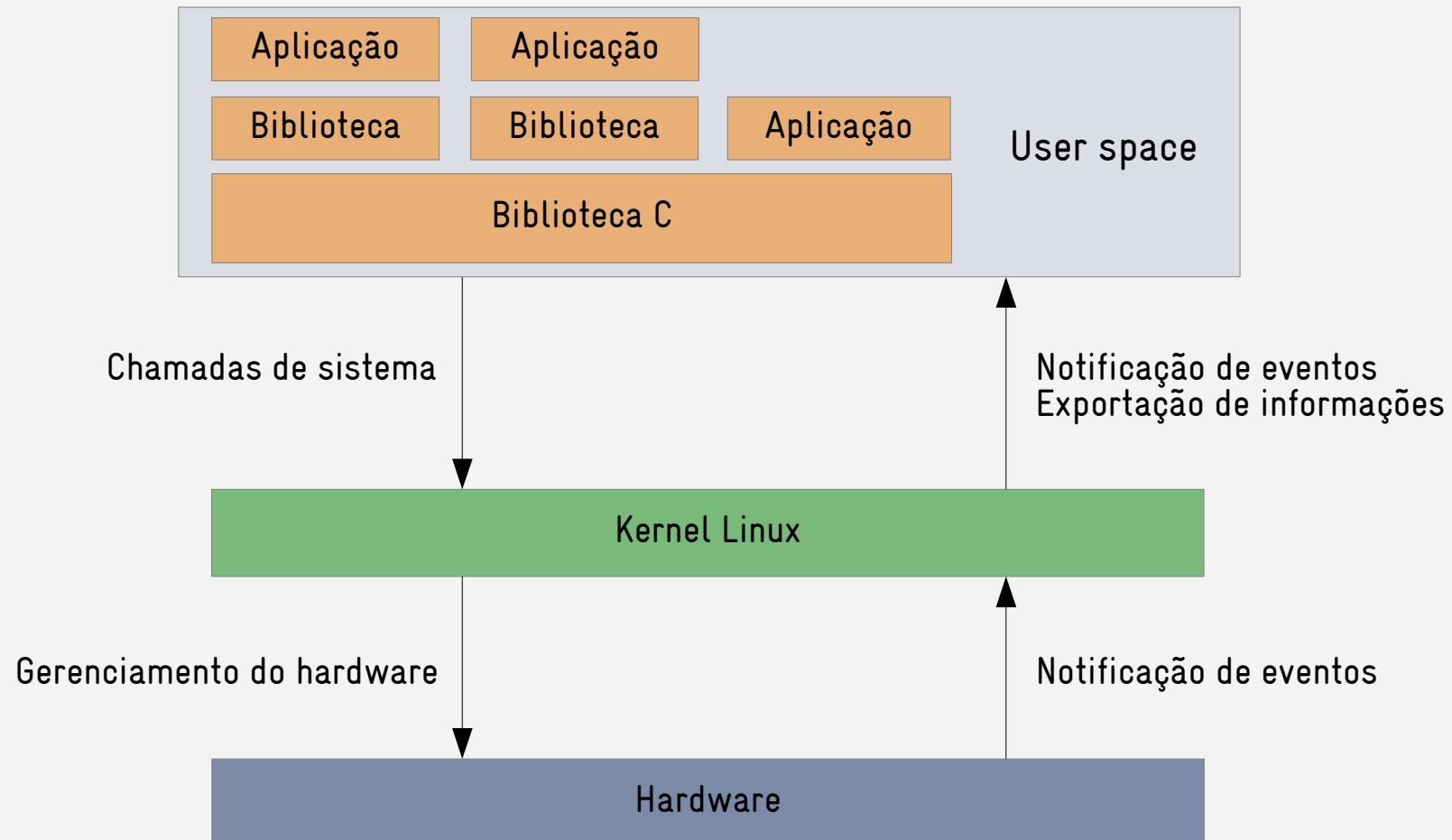
## PRINCIPAIS CARACTERÍSTICAS (cont.)

- ✗ Estável: capaz de rodar por muito tempo sem precisar de um único reboot.
- ✗ Seguro: sistema aberto, revisado por muitos especialistas, não tem como esconder falhas.
- ✗ Compatível com padrões de mercado.





# ARQUITETURA GERAL





# KERNEL SPACE x USER SPACE

- ✗ Existe uma separação bem definida entre o kernel (kernel space) e as bibliotecas e aplicações do usuário (user space).
- ✗ O kernel roda em modo privilegiado, com total acesso à todas as instruções da CPU, endereçamento de memória e I/O, enquanto que os processos do usuário rodam em modo restrito, com acesso limitado aos recursos da máquina.
- ✗ Por isso, existe uma interface de comunicação, baseada chamadas de sistema (system calls), para que as bibliotecas e aplicações tenham acesso aos recursos da máquina.





# CHAMADAS DE SISTEMA

- ✗ O Linux possui em torno de 350 chamadas de sistema, definidas em <asm/unistd.h>.
- ✗ Operações em arquivos, operações de rede, comunicação entre processos, gerenciamento de processos, mapeamento de memória, timers, threads, mecanismos de sincronização, etc.
- ✗ As chamadas de sistema são abstraídas pela biblioteca C padrão. As aplicações normalmente não precisam fazer uma chamada direta. Tudo é feito através da biblioteca C padrão.
- ✗ A interface de chamadas de sistema é bem estável. Em novas versões do kernel, normalmente apenas novas chamadas de sistema são adicionadas.





# FONTES DO KERNEL

- × A versão oficial do código-fonte do kernel liberada por Linus Torvalds encontra-se em:

<http://www.kernel.org>

- × Baixando os fontes por http:

```
$ wget http://www.kernel.org/pub/linux/kernel/v3.0/linux-3.6.tar.bz2  
$ tar xjfv linux-3.6.tar.bz2
```

- × Baixando os fontes por git:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```





# FONTES DO KERNEL (cont.)

- ✗ Muitas comunidades e fabricantes de hardware podem manter versões alternativas do kernel:
  - ✗ Comunidades podem manter versões do kernel voltadas à arquiteturas específicas (ARM, MIPS, PPC), sub-sistemas (USB, PCI, network), sistemas de tempo-real, etc.
  - ✗ Fabricantes de hardware podem manter versões específicas do kernel com suporte às suas plataformas de referência.





# VERSIONAMENTO

- ✗ Antes da versão 2.6:
  - ✗ Uma árvore de versões estáveis (1.0, 2.0, 2.2, 2.4)
  - ✗ Uma árvore de versões de desenvolvimento (2.1, 2.3, 2.5)
- ✗ A partir de 2003, apenas uma árvore: 2.6.X
- ✗ Em 2011, a versão mudou para 3.0.
- ✗ Em 2015, a versão mudou para 4.0.





# CICLO DE RELEASE

- ✗ Processo de desenvolvimento aproximadamente a cada 3 meses:
  - ✗ Merge window: 2 semanas (até sair 4.X-rc1).
  - ✗ Bug fixing: 6 a 10 semanas (4.X-rc2, 4.X-rc3, etc).
- ✗ Em aproximadamente 3 meses temos a liberação do release final 4.X.
- ✗ Para acompanhar as mudanças no kernel:  
<http://wiki.kernelnewbies.org/LinuxChanges>  
<http://lwn.net>





# LICENÇA

- ✗ Todo o código-fonte do Linux é software livre e liberado sob a licença GPLv2.
- ✗ Isso significa que:
  - ✗ Quando você receber ou comprar um equipamento com Linux, você tem o direito de requisitar os fontes, alterá-los e redistribuí-los.
  - ✗ Quando você desenvolver um produto com Linux, você precisa liberar os fontes do kernel sob as mesmas condições, sem restrições.





## LICENÇA (cont.)

- ✗ Portanto, é ilegal distribuir um binário do kernel com módulos compilados estaticamente.
- ✗ Os módulos do kernels são uma área cinza: é um trabalho derivado do kernel ou não?
  - ✗ A opinião geral da comunidade é de que drivers de código fechado são ruins. Veja "Kernel Driver Statement" no link abaixo:  
<http://j.mp/fbyuuH>
  - ✗ Sob um ponto de vista legal, cada driver é provavelmente um caso diferente. Ex: Nvidia.
  - ✗ É realmente útil manter um driver proprietário?





# VANTAGENS DE DRIVERS GPL

- ✗ Você não precisa escrever um driver do zero, podendo reusar o código de outros drivers.
- ✗ Você pode integrar o seu driver na árvore oficial do kernel, e não se preocupar com qualquer alteração em APIs internas do Linux. Custo zero de manutenção e melhorias no driver!
- ✗ Com drivers abertos você tem suporte da comunidade, com mais pessoas revisando e colaborando com seu código.
- ✗ Os usuários e a comunidade tem uma visão positiva da empresa.





# LINGUAGEM DE PROGRAMAÇÃO

- ✗ Implementado em linguagem C, como todos os sistemas UNIX.
- ✗ Um pouco de assembly é usado no código dependente de arquitetura em /arch (código de inicialização, tratamento de exceções, bibliotecas críticas, etc).
- ✗ Nada de C++!

<http://www.tux.org/lkml/#s15-3>





# LINGUAGEM DE PROGRAMAÇÃO (cont.)

- ✗ Compilado normalmente com o GCC, já que usa algumas extensões específicas deste compilador.
  - ✗ Suporta também alguns compiladores específicos com o da Intel e o da Marvell.
  - ✗ Existe uma iniciativa para usar o Clang do projeto LLVM.  
<http://llvm.linuxfoundation.org>





# BIBLIOTECA C PADRÃO

- ✗ O kernel é uma aplicação standalone (baremetal) e não pode usar ou ser dependente de código que roda em espaço de usuário.
- ✗ Portanto, você não pode usar funções da biblioteca C padrão (`memset()`, `malloc()`, `printf()`, etc).
- ✗ Por este motivo, o kernel tem sua própria implementação de rotinas comuns da biblioteca C (`memset()`, `kmalloc()`, `printk()`, etc).
- ✗ Para uma lista completa da API do kernel, consulte o DocBook:  
<http://free-electrons.com/kerneldoc/latest/DocBook/kernel-api/>





# POR T A B I L I D A D E

- ✗ O kernel Linux é projetado para ser extremamente portável.
- ✗ Todo o código fora de arch/ deve ser portável.
- ✗ Para isso, o kernel provê diversas funções e macros para abstrair os detalhes específicos da arquitetura, por exemplo:
  - ✗ Endianness (cpu\_to\_be32, cpu\_to\_le32, be32\_to\_cpu, le32\_to\_cpu).
  - ✗ Acesso à I/O mapeado em memória.
  - ✗ DMA API.





# API INTERNA DO KERNEL

- ✗ A API interna do Linux pode mudar entre duas versões estáveis do kernel. Isso significa que um driver que foi desenvolvido para uma versão do kernel pode não funcionar na próxima versão. Mais informações em Documentation/stable\_api\_nonsense.txt.
- ✗ Sempre que um desenvolvedor alterar uma API interna do kernel, ele é responsável por atualizar todo o código que acessa esta API, garantindo que nada no kernel vai parar de funcionar.
- ✗ Funciona muito bem para todo código na árvore oficial do kernel (mainline), mas pode quebrar drivers de código fechado ou fora da árvore do kernel.





# FERRAMENTAS

- ✗ Como o código do kernel é bem extenso, é necessário o uso de alguma ferramenta para navegar de forma eficiente em seu código-fonte.
- ✗ Sugere-se um editor de textos leve e estável, como por exemplo o vim, emacs ou kdevelop.
- ✗ Espera-se que o editor seja capaz de indexar os símbolos do kernel Linux (algumas ferramentas como o ctags e o cscope podem ajudar).
- ✗ Uma ferramenta chamada LXR (Linux Cross Referencer), capaz de indexar todos os símbolos de um projeto de software para futura consulta em formato HTML, também pode ser útil.





Embedded Labworks

# LABORATÓRIO

Estudando os fontes do Linux



Embedded Labworks

# Linux embarcado

Toradex Colibri i.MX6DL



# COLIBRI IMX6DL

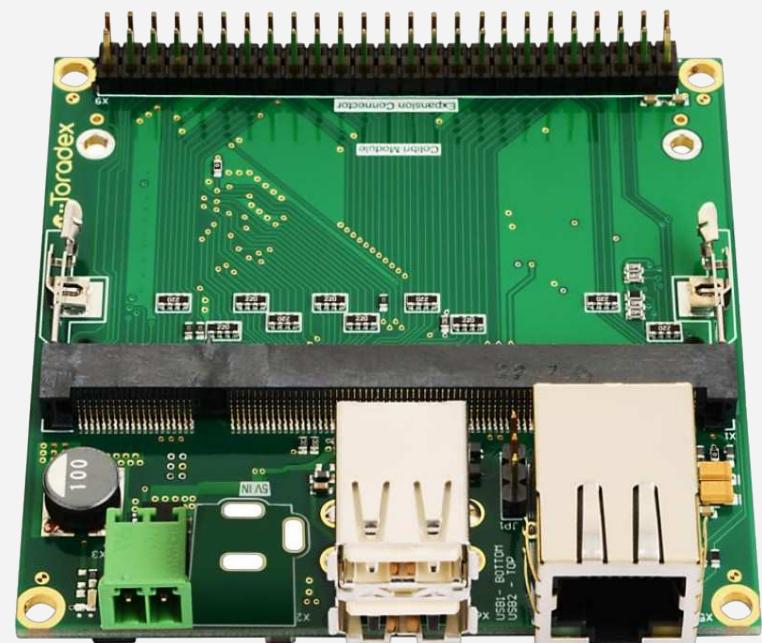
- ✗ System-on-module (SOM) da Toradex.
- ✗ Baseado no SOC i.MX6 DualLite da NXP, um ARM Cortex-A9 rodando a até 996MHz.
- ✗ 512MB de memória RAM.
- ✗ 4GB de armazenamento interno (eMMC NAND Flash).
- ✗ Conector no padrão SODIMM200 (memória DDR1).





# PLACA BASE VIOLA

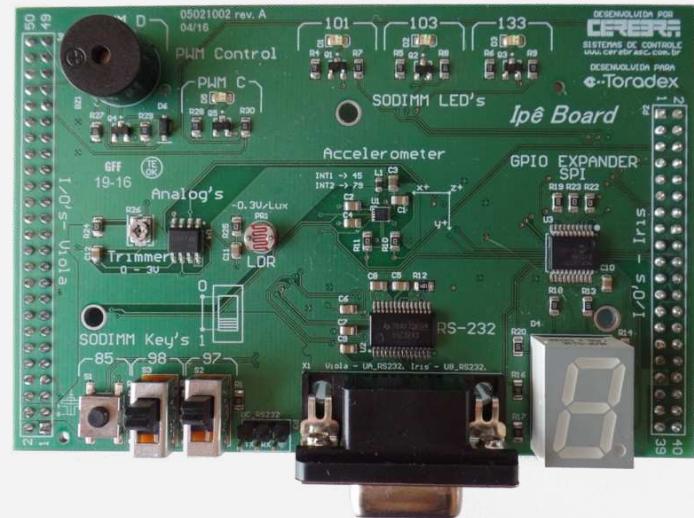
- ✗ Placa base compatível com todos os módulos Colibri da Toradex.
- ✗ 2 conectores USB Host e 1 conector Ethernet.
- ✗ 1 conector de cartão SD.
- ✗ Interface RGB para display LCD.
- ✗ Barramento de 50 pinos que exporta o acesso à diversas interfaces de I/O do SOC (I2C, SPI, UART, GPIO, etc).
- ✗ Alimentação externa com uma fonte de 5V/2A.





# PLACA DE EXPANSÃO IPÊ

- ✗ Placa desenvolvida pela Toradex Brasil.
- ✗ 2 chaves, 1 botão e 3 leds conectados a GPIOs.
- ✗ 1 led e 1 buzzer conectados a canais PWM.
- ✗ 1 porta serial para console.
- ✗ 1 resistor dependente de luz (LDR) e um trimpot conectados a canais A/D.
- ✗ 1 acelerômetro MMA8653.
- ✗ 1 expansor de GPIOs MCP23S08 conectado a um display de 7 segmentos.





# REFERÊNCIAS E DOCUMENTAÇÃO

- ✗ A documentação do hardware está disponível no ambiente de laboratório do treinamento em /opt/labs/docs/hardware:
  - ✗ SOC\_imx6sdl\_datasheet.pdf: datasheet do SOC.
  - ✗ SOM\_colibri\_imx6\_datasheet.pdf: datasheet do SOM.
  - ✗ BASE\_BOARD\_viola\_datasheet.pdf: datasheet da placa base.
  - ✗ EXT\_BOARD\_ipe\_esquemático.pdf: esquemático da placa de expansão.
- ✗ Recursos na internet:  
<http://www.toradex.com/>  
<https://www.toradex.com/community/>  
<http://community.nxp.com>





Embedded Labworks

# LABORATÓRIO

Conectando e testando o target



# Linux device drivers

Configurando e compilando o kernel Linux



# CONFIGURANDO O KERNEL

- ✗ O kernel possui centenas de drivers de dispositivo, diversos protocolos de rede e muitos outros itens de configuração.
- ✗ O kernel é bem modular, são muitas as opções disponíveis para serem habilitadas/desabilitadas.
- ✗ O processo de configuração serve para você configurar o kernel para ser compilado para sua CPU/plataforma.
- ✗ O conjunto de opções que você irá habilitar depende:
  - ✗ Do seu hardware (device drivers, etc).
  - ✗ Das funcionalidades (protocolos de rede, sistemas de arquivo, etc).





# CONFIGURAÇÃO

- As configurações são salvas em um arquivo chamado `.config` no diretório principal dos fontes do kernel, e possuem o formato `key=value`. Exemplo:

`CONFIG_ARM=y`

- Dificilmente você vai precisar editar o arquivo `.config` manualmente. Existem ferramentas de interface gráfica para configurar o kernel e gerar o arquivo de configuração automaticamente:

```
$ make menuconfig  
$ make gconfig  
$ make xconfig  
$ make nconfig
```





# \$ make xconfig

qconf

File Option Help

Option Name

Code maturity level options

General setup

- Configure standard kernel features (for small systems) EMBEDDED

Loadable module support

System Type

- Intel PXA2xx Implementations
  - Toshiba e7xx / e8xx
  - Asus G20/620BT
  - hp iPAQ h1900
  - hp iPAQ h2200**
  - hp iPAQ h3900
  - hp iPAQ h4000
  - hp iPAQ h5400
  - Dell Axim X5
  - Dell Axim X3 (non-functional)
  - RoverP1 (Mitac Mio 336)
  - RoverP5+
- Linux As Bootloader
- Compaq/iPAQ Options

General setup

- PCMCIA/CardBus support
- Generic Driver Options
- Parallel port support

Memory Technology Devices (MTD)

- RAM/ROM/Flash chip drivers
- Mapping drivers for chip access
- Self-contained MTD device drivers
- NAND Flash Device Drivers

Plug and Play support

Option Name

- iPAQ H2200 PCMCIA
- iPAQ H2200 MediaQ 1178 LCD
- iPAQ H2200 battery interface
- iPAQ H2200 touchscreen driver
- iPAQ H2200 hardware audio control

**hp iPAQ h2200 (ARCH\_H2200)**

type: boolean  
prompt: hp iPAQ h2200  
    dep: ARCH\_PXA  
select: PXA25x  
    dep: ARCH\_PXA

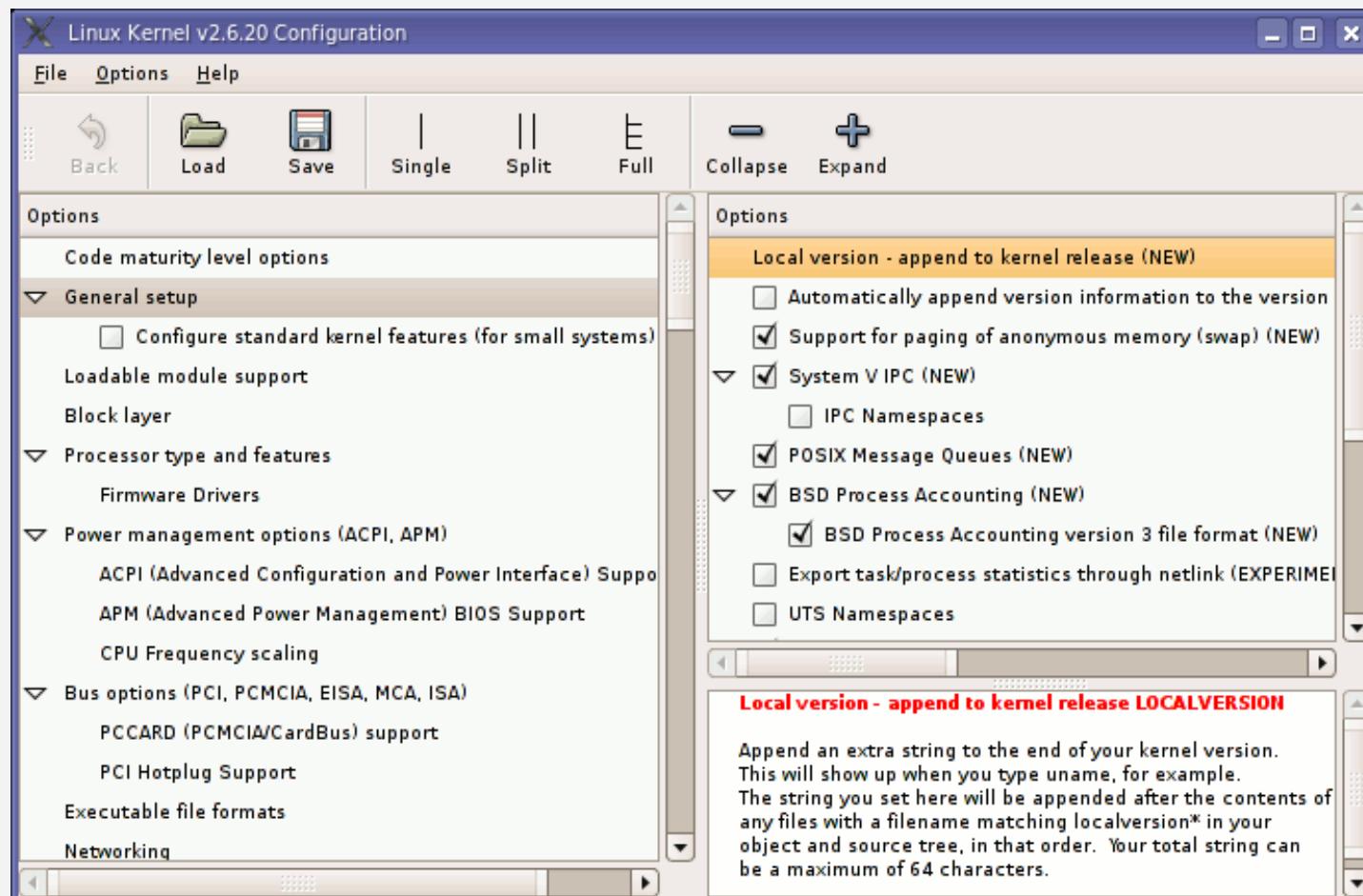
defined at arch/arm/mach-pxa/h2200/Kconfig:1

This enables support for HP iPAQ H22xx series of handhelds.  
There are a number of H22xx-specific drivers under this submenu:  
pcmcia, lcd, battery, touchscreen





# \$ make gconfig





# \$ make nconfig

```
.config - Linux/x86_64 3.7.0-rc4 Kernel Configuration
Linux/x86_64 3.7.0-rc4 Kernel Configuration

        General setup --->
[*] Enable loadable module support --->
[*] Enable the block layer --->
    Processor type and features ---> Processor type and features --->
    Power management and ACPI options --->
    Bus options (PCI etc.) --->
    Executable file formats / Emulations --->
-** Networking support --->
    Device Drivers --->
    Firmware Drivers --->
    File systems --->
    Kernel hacking --->
    Security options --->
-** Cryptographic API --->
[*] Virtualization --->
    Library routines --->

F1 Help F2 Sym Info F3 Insts F4 Config F5 Back F6 Save F7 Load F8 Sym Search F9 Exit
```





# \$ make menuconfig

```
sprado@n1: ~/workspace/build/linux/linux
.config - Linux/x86 4.7.0-rc4 Kernel Configuration
> Processor type and features
    Processor type and features
    Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty
    submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes,
    <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for
    Help, </> for Search. Legend: [*] built-in [ ] excluded <M> module < >
    ^(-)
        [ ] Enable Maximum number of SMP Processors and NUMA Nodes
        (256) Maximum number of CPUs
        [*] SMT (Hyperthreading) scheduler support
        [*] Multi-core scheduler support
            Preemption Model (Voluntary Kernel Preemption (Desktop)) --->
        [*] Reroute for broken boot IRQs
        [*] Machine Check / overheating reporting
        [ ] Intel MCE features
        [ ] AMD MCE features
        <M> Machine check injector support
            Performance monitoring --->
        [ ] Enable support for 16-bit segments
        [ ] Enable vsyscall emulation
        < > Dell i8k legacy laptop support
        [ ] CPU microcode loading support
        < > /dev/cpu/*/msr - Model-specific register support
        < > /dev/cpu/*/cpuid - CPU information support
        [ ] Numa Memory Allocation and Scheduler Support
    ^(+)

    <Select>   < Exit >   < Help >   < Save >   < Load >
```





# BUILT-IN OU MÓDULO

- ✗ O kernel permite que algumas das funcionalidades disponíveis possam ser habilitadas e compiladas de duas formas:
  - ✗ **Estática ou built-in:** a funcionalidade selecionada é linkada estaticamente à imagem final do kernel.
  - ✗ **Dinâmica ou módulo:** é gerado um módulo daquela funcionalidade (arquivo com extensão .ko). Este módulo não é incluído na imagem final do kernel. Ele é incluído no sistema de arquivos e pode ser carregado dinamicamente (em tempo de execução), conforme a necessidade.





# OPÇÕES DE CONFIGURAÇÃO

- ✗ Opções booleanas (verdadeiro/falso)
  - [ ] → Opção desabilitada
  - [\*] → Opção habilitada
- ✗ Opções de 3 estados:
  - < > → Opção desabilitada
  - <\*> → Opção habilitada (built-in)
  - <M> → Opção habilitada (módulo)
- ✗ Números inteiros. Ex: (17) Kernel log buffer size
- ✗ Strings. Ex: (iso8859-1) Default iocharset for FAT





# DEPENDÊNCIAS

- ✗ Na configuração do kernel, podem existir dependências entre funcionalidades:
  - ✗ Exemplo 1: o driver de um dispositivo I2C só pode ser habilitado se o barramento I2C for habilitado.
  - ✗ Exemplo 2: o framework de porta serial do kernel (serial core) é habilitado automaticamente quando um driver de UART é habilitado.





# CONFIGURAÇÃO POR ARQUITETURA

- ✗ Toda a configuração do kernel é dependente da arquitetura.
- ✗ Por padrão, o kernel considera uma compilação nativa, então irá usar a arquitetura da máquina de desenvolvimento no comando abaixo:

```
$ make menuconfig
```

- ✗ Portanto, para configurar para ARM por exemplo, você precisa especificar a arquitetura:

```
$ make ARCH=arm menuconfig
```

- ✗ Ao invés de passar a variável ARCH na chamada do make, você pode também definí-la como variável de ambiente ou alterar o arquivo Makefile no diretório principal do kernel.





# CONFIGURAÇÕES PRÉ-DEFINIDAS

- Arquivos de configuração pré-definidos para diversas plataformas estão disponíveis em arch/<arch>/configs/.

- O uso de arquivos pré-configurados é a forma padrão de configurar um kernel para uma plataforma específica. Por exemplo, para carregar a configuração padrão do módulo Colibri i.MX6 da Toradex:

```
$ make ARCH=arm colibri_imx6_defconfig
```

- Se você mexeu na configuração padrão e deseja salvá-la, pode criar uma cópia conforme exemplo abaixo:

```
$ cp .config arch/arm/configs/myconfig_defconfig
```





# VALIDANDO O ARQUIVO DE CONFIGURAÇÃO

- ✗ O comando abaixo faz a validação e a consistência do arquivo de configuração do kernel:  
`$ make oldconfig`
- ✗ Ele avisa e configura automaticamente parâmetros e dependências que antes não existiam.
- ✗ Deve ser usado sempre que:
  - ✗ Você alterar o arquivo `.config` manualmente.
  - ✗ Você reutilizar o mesmo `.config` em diferentes versões do kernel.





# COMPILEANDO O KERNEL

- ✗ Depois de configurado, para compilar nativamente basta executar:  
`$ make`
- ✗ Não precisa de privilégios de root!
- ✗ Para compilar de forma cruzada, você precisa indicar a arquitetura e o prefixo do cross-compiler. Exemplo:  
`$ make ARCH=arm CROSS_COMPILE=arm-linux-`
- ✗ O comando acima irá gerar uma imagem genérica para ARM (além dos módulos do kernel que estudaremos mais adiante). Se você quiser gerar uma imagem específica, deve adicionar ao fim do comando o nome da imagem.  
`$ make ARCH=arm CROSS_COMPILE=arm-linux- zImage`





# COMPILEANDO OS MÓDULOS

- Para compilar apenas os módulos, basta executar:

```
$ make modules
```

- Para compilar os módulos de forma cruzada, não esqueça de indicar a arquitetura e o prefixo do cross-compiler:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- modules
```

- Estudaremos módulos do kernel em detalhes mais adiante no treinamento.





# COMPILEANDO O KERNEL (cont.)

- ✗ Ao fim do processo de compilação, serão geradas as seguintes imagens:
  - ✗ vmlinuz: imagem do kernel no formato ELF, que não é inicializável, mas pode ser usada para debugging.
  - ✗ \*.ko: módulos do kernel, dentro de seus respectivos diretórios.
  - ✗ Em arch/<arch>/boot/:
    - ✗ Image: imagem final do kernel, inicializável e descomprimida.
    - ✗ \*Image: imagem inicializável e comprimida do kernel (bzImage para x86, zImage para ARM, etc).
    - ✗ uImage: imagem do kernel para o U-Boot (opcional).





# INSTALANDO O KERNEL

- ✗ Para instalar o kernel, basta executar o comando abaixo:  
`$ make INSTALL_PATH=<install_dir> install`
- ✗ Este comando irá instalar os seguintes arquivos no diretório passado (ou no diretório /boot por padrão):
  - ✗ `vmlinuz-<version>` (imagem do kernel comprimida)
  - ✗ `System.map-<version>` (endereços dos símbolos do kernel)
  - ✗ `config-<version>` (arquivo de configuração do kernel)
- ✗ Normalmente não é usado em sistemas embarcados. Em sistemas embarcados, normalmente gravamos o kernel em um dispositivo de armazenamento (cartão SD, memória flash, etc).





# INSTALANDO OS MÓDULOS

- Para instalar os módulos, basta executar o comando abaixo:

```
$ make modules_install
```

- No caso de um ambiente de compilação cruzada, os módulos devem ser instalados no rootfs do target.

- Para isso, devemos passar o parâmetro `INSTALL_MOD_PATH` no comando de instalação:

```
$ make ARCH=<arch> INSTALL_MOD_PATH=<dir> modules_install
```





# DEVICE TREE

- ✗ Muitas plataformas possuem dispositivos de hardware que não podem ser identificados dinamicamente pelo kernel.
- ✗ Nestes casos, é necessário um mecanismo para comunicar ao kernel informações sobre os dispositivos de hardware presentes no sistema.
- ✗ A forma antiga de resolver este problema era descrevendo o hardware através de estruturas de dados, o que deixava o código-fonte do kernel mais confuso e difícil de manter.
- ✗ Atualmente, a maioria das plataformas (incluindo ARM) suportam o mecanismo de device tree.





# DEVICE TREE (cont.)

- ✗ O device tree é uma estrutura de dados capaz de descrever a topologia e a configuração do hardware presente no sistema.
- ✗ Na prática, o device tree é um arquivo texto com extensão .dts que descreve hierarquicamente o hardware através de nós e propriedades.
- ✗ Atualmente, o device tree é suportado por diversas arquiteturas, incluindo ARM, PowerPC, OpenRISC, ARC e Microblaze.





# LISTAGEM DEVICE TREE

- × Cada placa possui um arquivo de especificação do device tree, disponível no código-fonte do kernel em arch/arm/boot/dts:

```
$ ls arch/arm/boot/dts/  
aks-cdu.dts  
alphascale-asm9260-devkit.dts  
alpine-db.dts  
am335x-baltos-ir5221.dts  
am335x-base0033.dts  
am335x-boneblack.dts  
am335x-bone.dts  
am335x-bonegreen.dts  
am335x-chiliboard.dts  
[ ... ]
```





# COMPILEANDO O DEVICE TREE

- ✗ Uma ferramenta chamada Device Tree Compiler (dtc), disponível em `scripts/dtc/`, é responsável por compilar o device tree.
- ✗ Para compilar o device tree de todas as placas habilitadas na configuração do kernel:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- dtbs
```
- ✗ Para compilar o device tree de uma placa específica:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- imx6dl-colibri-ipe.dtb
```
- ✗ Os arquivos DTB são gerados em `arch/arm/boot/dts`.





# BOOT COM DEVICE TREE

- ✗ Durante o processo de boot, o bootloader é o responsável por passar o DTB para o kernel.
- ✗ Caso o bootloader não seja capaz de passar o DTB para o kernel, é possível indicar ao kernel que o DTB está integrado à sua imagem habilitando a opção `CONFIG_ARM_APPENDED_DTB`, e depois gerando uma imagem com o DTB integrado:

```
$ cat zImage board.dtb > zImage_dtb
```





# FAZENDO A LIMPEZA

- × Remove todos os arquivos gerados (imagens, arquivos-objeto, etc).

```
$ make clean
```

- × Remove todos os arquivos de gerados e arquivos de configuração (usado quando pretende-se mudar de plataforma).

```
$ make mrproper
```

- × Além dos arquivos gerados e arquivos de configuração, remove também arquivos de backup (bom para gerar patches).

```
$ make distclean
```





# LINHA DE COMANDOS DO KERNEL

- ✗ Ao ser carregado, o kernel pode receber um conjunto de parâmetros. Chamamos esses parâmetros de linha de comandos do kernel.
- ✗ Esta linha de comandos pode ser passada ao kernel de duas formas diferentes:
  - ✗ Pelo bootloader.
  - ✗ Definido em tempo de compilação na configuração do kernel, através da opção `CONFIG_CMDLINE`.
- ✗ Esta linha de comandos é uma string com diversas opções no formato **chave=valor**.





# LINHA DE COMANDOS DO KERNEL (cont.)

```
console=ttySAC0 root=/dev/mtdblock3 rootfstype=jffs2
```

- ✗ Exemplo de linha de comandos do kernel, onde:
  - ✗ console = dispositivo que será usado como console
  - ✗ root = dispositivo onde se encontra o sistema de arquivos
  - ✗ rootfstype = tipo do sistema de arquivos (JFFS2)
- ✗ Existem dezenas de outras opções!
- ✗ Documentação disponível em:  
[Documentation/kernel-parameters.txt](#)





Embedded Labworks

# LABORATÓRIO

Compilando o kernel e testando o sistema



Embedded Labworks

# Linux device drivers

Módulos do kernel



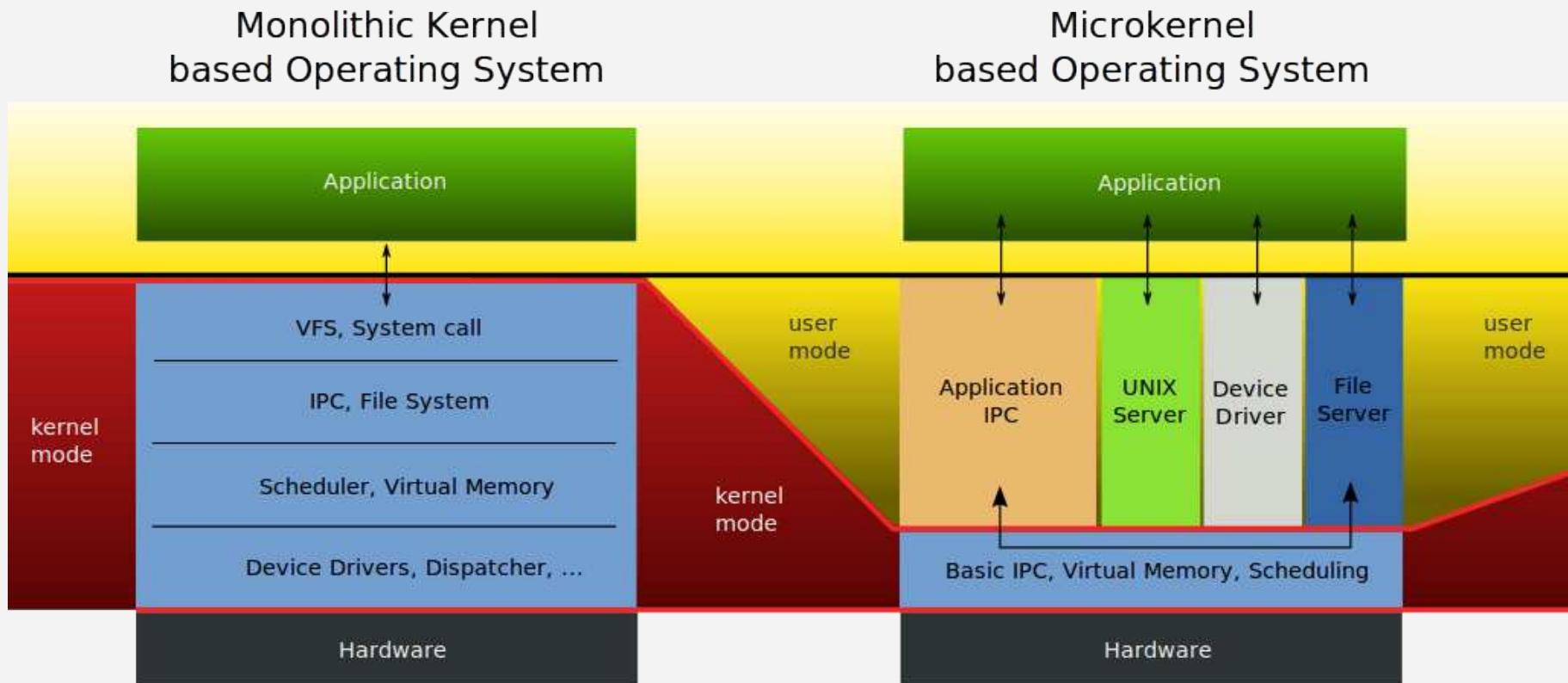
# KERNEL MONOLÍTICO E MICROKERNEL

- ✗ **Kernel monolítico:** o sistema operacional inteiro roda em kernel space, com total acesso aos recursos da máquina (CPU, memória e I/Os), e provê para as aplicações (userspace) uma interface de comunicação através de chamadas de sistema (system calls).
- ✗ **Microkernel:** apenas o núcleo do kernel roda em kernelspace (gerenciamento de memória e processos). O restante roda em userspace, incluindo sistemas de arquivos, device drivers, protocolos de rede, etc!





# MONOLÍTICO X MICROKERNEL





# MÓDULOS

- ✗ O Linux é um kernel monolítico, mas internamente é bem modular. Cada funcionalidade é abstraída em um módulo, com uma interface de comunicação bem definida. Por isso, permite um sistema de configuração onde você pode adicionar ou remover determinada funcionalidade.
- ✗ Além disso, é possível compilar separadamente determinada funcionalidade, e carregar o arquivo gerado em tempo de execução! Chamamos este arquivo compilado separadamente de módulo do kernel.
- ✗ Um device driver pode ser compilado de forma integrada ao kernel (built-in) ou como um módulo do kernel.





# VANTAGENS DOS MÓDULOS

- ✗ Módulos tornam fácil o desenvolvimento do kernel (ex: device drivers) sem precisar reiniciar o target.
- ✗ Ajuda a manter a imagem do kernel bem pequena.
- ✗ Só ocupa memória enquanto estiver carregado.
- ✗ O tempo de boot do kernel fica menor.
- ✗ Cuidado: módulos rodam em kernel space. Uma vez carregados, eles tem total controle do sistema! Por isso só podem ser carregados como root.





# DEPENDÊNCIAS DOS MÓDULOS

- ✗ Alguns módulos dependem de outros módulos, que precisam ser carregados primeiro.
- ✗ Exemplo: o módulo `usb_storage` depende do módulo `usbcore`.
- ✗ As dependências entre os módulos estão descritas no arquivo `/lib/modules/<kernel-version>/modules.dep`.
- ✗ Este arquivo é gerado automaticamente quando você instala os módulos, através da ferramenta `depmod`.





# CARREGANDO UM MÓDULO

- Carrega apenas o módulo passado. É necessário passar o caminho completo do módulo.

```
$ insmod /path/to/module.ko
```

- Carrega o módulo e todas as suas dependências. Deve-se passar apenas o nome do módulo, que deve estar instalado em /lib/modules/.

```
$ modprobe <module_name>
```





# DESCARREGANDO UM MÓDULO

- × Descarrega apenas o módulo passado. Deve-se passar apenas o nome do módulo. Possível apenas se o módulo não estiver mais em uso.

```
$ rmmod <module_name>
```

- × Descarrega o módulo e todas as suas dependências (que não estão sendo usadas). Deve-se passar apenas o nome do módulo.

```
$ modprobe -r <module_name>
```





# LISTANDO INFORMAÇÕES DOS MÓDULOS

- × Lê informações de um módulo, como sua descrição, parâmetros, licença e dependências. Deve-se passar apenas o nome do módulo, que deve estar instalado em /lib/modules/.

```
$ modinfo <module_name>
```

- × Lista todos os módulos carregados.

```
$ lsmod
```





# PASSANDO PARÂMETROS

- × Passando um parâmetro via linha de comando:

```
$ modprobe <module> param=value
```

- × Passando um parâmetro via arquivo de configuração (/etc/modprobe.conf ou /etc/modprobe.d/):

```
options <module> param=value
```

- × Para passar um parâmetro via linha de comandos do kernel:

```
<module>.param=value
```





Embedded Labworks

# Linux device drivers

Desenvolvendo um módulo do kernel



```
#include <linux/module.h>
#include <linux/kernel.h>

/* module initialization */
static int __init mymodule_init(void)
{
    printk("My module initialized.\n");
    return 0;
}

/* module exit */
static void __exit mymodule_exit(void)
{
    printk("Exiting my module.\n");
}

module_init(mymodule_init);
module_exit(mymodule_exit);

MODULE_LICENSE("GPL");
```

`__init` é removido  
após a inicialização  
(built-in ou módulo)

`__exit` é descartado  
caso seja compilado  
estaticamente no kernel  
(built-in).





# O PRIMEIRO MÓDULO

- ✗ A macro `module_init()` declara a função de inicialização que será chamada ao carregar o módulo para a memória.
- ✗ A função de inicialização é nomeada de acordo com o padrão `<modulename>_init()`, é responsável por inicializar o módulo e retornar 0 para OK ou um valor negativo em caso de erro. É removida da memória assim que executada.
- ✗ A macro `module_exit()` declara a função de limpeza, que será chamada assim que o módulo for descarregado da memória. Se o módulo for compilado estaticamente, esta função não é utilizada, e por este motivo, nem é carregada para a memória.





# METADADOS DO MÓDULO

- ✗ Você pode declarar informações (metadados) dos módulos usando as seguintes macros:
  - ✗ `MODULE_LICENSE()`: Declara a licença do módulo.
  - ✗ `MODULE_DESCRIPTION()`: Mensagem descritiva do módulo (apenas informativa).
  - ✗ `MODULE_AUTHOR()`: Autor do módulo (também apenas informativa).
  - ✗ `MODULE_VERSION()`: Versão do módulo.
  - ✗ `MODULE_PARM_DESC()`: Descrição dos parâmetros recebidos pelo módulo.





# EXPORTANDO E USANDO SÍMBOLOS

- ✗ Para que os módulos possam usar algum símbolo do kernel (função ou variável global), estes símbolos precisam ser exportados explicitamente pelo kernel ou por outro módulo.
- ✗ Isso pode ser feito através de duas macros:
  - ✗ EXPORT\_SYMBOL(symbolname), símbolo pode ser usado por qualquer módulo.
  - ✗ EXPORT\_SYMBOL\_GPL(symbolname), símbolo pode ser usado apenas por módulos GPL.





# COMPILEANDO UM MÓDULO

- ✗ Existem duas opções para compilar um módulo:
  - ✗ Fora da árvore do kernel: fácil de gerenciar, mas não permite compilar um módulo estaticamente (built-in).
  - ✗ Dentro da árvore do kernel: permite compilar um módulo estaticamente ou dinamicamente.





# COMPILEANDO FORA DA ÁRVORE DO KERNEL

```
KDIR := /linux/source/code/directory/  
obj-m += mymodule.o  
  
module:  
    $(MAKE) -C $(KDIR) M=$(PWD) modules  
  
clean:  
    $(MAKE) -C $(KDIR) M=$(PWD) clean
```

Obs: em caso de compilação cruzada, não esqueça de definir os parâmetros ARCH e CROSS\_COMPILE.





# VERSÃO DO KERNEL

- ✗ Para ser compilado, um módulo do kernel precisa dos arquivos de cabeçalho (headers) do kernel.
- ✗ Desta forma, um módulo compilado fica dependente de uma determinada versão do kernel.
- ✗ Por este motivo, um módulo compilado com uma versão X dos headers do kernel não é carregado em um kernel com versão Y.
- ✗ As ferramentas insmod ou modprobe irão reportar o erro “Invalid module format”.





# INTEGRANDO NOS FONTES DO KERNEL

- ✗ Para integrar um driver nos fontes do kernel, primeiro adicione o arquivo-fonte em um diretório apropriado. Exemplo:  
`drivers/serial/8250.c`
- ✗ Descreva a configuração do driver no arquivo Kconfig disponível no diretório onde o fonte do driver foi adicionado:

```
config SERIAL_8250
    tristate "8250/16550 and compatible serial support"
    select SERIAL_CORE
    help
        This selects whether you want to include the driver for the standard
        serial ports. The standard answer is Y. People who might say N
        here are those that are setting up dedicated Ethernet WWW/FTP
        servers, or users that have one of the various bus mice instead of a
        ...

```





# INTEGRANDO NOS FONTES DO KERNEL (cont.)

- ✗ Adicione uma linha no Makefile baseado na configuração criada no Kconfig:  
`obj-$(CONFIG_SERIAL_8250) += 8250.o`
- ✗ Execute `make menuconfig` para habilitar a nova opção.
- ✗ Execute `make` ou `make modules` para compilar.
- ✗ Mais informações em `Documentation/kbuild/`.





# PARÂMETROS

```
#include <linux/moduleparam.h>

/* module parameter macro */
module_param(  
    name, /* name of an already defined variable */  
    type, /* either byte, short, ushort, int, uint,  
           long, ulong, charp, or bool.  
           (checked at compile time!) */  
    perm /* visibility in sysfs at  
          /sys/module/<module_name>/parameters/<param>  
          see linux/stat.h */  
);  
  
/* example */  
static int qtd = 5;  
module_param(qtd, int, S_IRUGO);  
MODULE_PARM_DESC(qtd, "Number of buffers to handle");
```





Embedded Labworks

# LABORATÓRIO

O primeiro módulo



Embedded Labworks

# Linux device drivers

Dispositivos de hardware



# DISPOSITIVOS

- ✗ Um papel importante do kernel é prover um mecanismo de acesso ao hardware para as bibliotecas e aplicações.
- ✗ No Linux, o acesso ao hardware é exportado para as aplicações através de 3 principais classes de dispositivos:
  - ✗ Character device (dispositivo de caractere).
  - ✗ Block Device (dispositivo de bloco).
  - ✗ Network device (dispositivo de rede).





# CLASSES DE DISPOSITIVOS

- ✗ **Char device:** pode ser acessado como um stream contínuo de dados (acesso sequencial), sem começo, meio e fim. É acessado através de um arquivo em /dev. Ex: porta serial, impressora, placa de som, etc.
- ✗ **Block device:** trabalha com blocos de dados, pode ser endereçável, tem começo, meio e fim. É acessado através de um arquivo em /dev. Ex: HD, CDROM, DVD, pendrive, etc.
- ✗ **Network device:** dispositivo totalmente diferente, que pode ser representado por uma interface de rede física ou por software (loopback), responsável por enviar e receber pacotes de dados através da camada de rede do kernel. Não possui um arquivo em /dev. A comunicação é feita através de uma API específica.





# ARQUIVOS DE DISPOSITIVO

- ✗ Os dispositivos de caractere e bloco são representados para as aplicações através de arquivos chamados arquivos de dispositivos e armazenados no diretório /dev por convenção.
- ✗ Cada arquivo de dispositivo possui 3 informações básicas, que identificam internamente o dispositivo ao qual o arquivo pertence:
  - ✗ Tipo (caractere ou bloco).
  - ✗ Major number (categoria do dispositivo).
  - ✗ Minor number (identificador do dispositivo).





# ARQUIVOS DE DISPOSITIVO (cont.)

- Exemplos de arquivos de dispositivo:

```
brw-r---- 1 root root 31, 0 Feb 7 2012 /dev/mtdblock0
brw-r---- 1 root root 8, 1 Feb 7 2012 /dev/sda1
crw-rw-rw- 1 root root 4, 64 Feb 7 2012 /dev/ttyS0
crw-rw-rw- 1 root root 4, 65 Feb 7 2012 /dev/ttyS1
crw-rw-rw- 1 root root 29, 0 Feb 7 2012 /dev/fb0
crw-rw-rw- 1 root root 1, 1 Feb 7 2012 /dev/mem
crw-rw-rw- 1 root root 1, 3 Feb 7 2012 /dev/null
```





# CONVERSANDO COM O HARDWARE

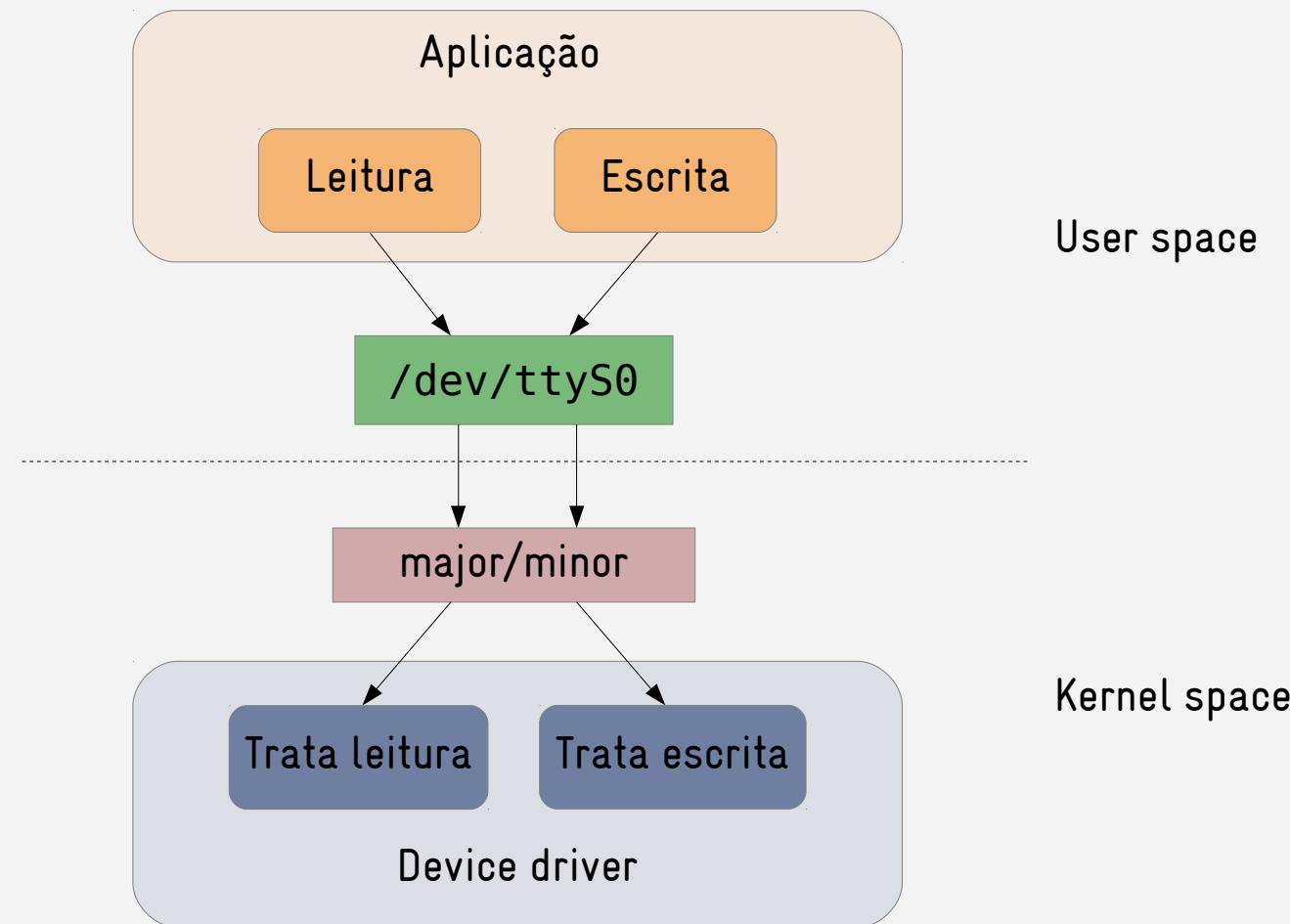
- × Como os dispositivos de hardware são exportados através de arquivos de dispositivo para as aplicações, o acesso ao hardware é abstruído através de uma API comum de acesso à arquivos (open, read, write, close).
- × Exemplo de escrita na porta serial:

```
int fd;  
  
fd = open("/dev/ttyS0", O_RDWR);  
write(fd, "Hello world!", 12);  
close(fd);
```





# ACESSANDO UM ARQUIVO DE DISPOSITIVO





# CRIANDO ARQUIVOS DE DISPOSITIVO

- ✗ Os arquivos de dispositivo podem ser criados manualmente:  
`$ mknod /dev/<device> [c|b] major minor`
- ✗ Neste caso, é preciso conhecer o major e minor definidos no driver do dispositivo, além de ter privilégios de root.
- ✗ Porém, o mais comum é a utilização de mecanismos automáticos de criação dos arquivos de dispositivo:
  - ✗ **udev**: daemon, solução usada em desktop e servidores.
  - ✗ **mdev**: programa disponível no Busybox, versão mais leve do udev.
  - ✗ **devtmpfs**: virtual filesystem do Linux, disponível desde a versão 2.6.32.





# DISPOSITIVOS DE CARACTERE

- ✗ Com exceção dos drivers para dispositivos de armazenamento, a maioria dos drivers são implementados como um driver de dispositivo de caractere.
- ✗ Portanto, a maioria dos drivers que você irá desenvolver será do tipo caractere.
- ✗ Por este motivo, estudaremos no treinamento o desenvolvimento de um driver de dispositivo de caractere.





# IMPLEMENTANDO UM CHAR DEVICE

- ✗ Para implementar um driver de dispositivo do tipo caractere, existem três passos principais:
  1. Reservar o major number e o(s) minor number(s) do seu driver.
  2. Implementar as operações que serão disponibilizadas aos usuários do seu driver (open, read, write, close, etc) e associar estas operações à uma estrutura do tipo `file_operations`.
  3. Associar o major e o minor number alocado à sua estrutura de operações de arquivo e registrar o dispositivo de caractere.





# DEVICE NUMBER

- ✗ O primeiro passo para desenvolver um driver de dispositivo de caractere ou bloco é registrar um device number para o driver do dispositivo. O device number é composto por dois números chamados de **major number** e **minor number**.
- ✗ O kernel armazena as informações de major e minor number no tipo de dados `dev_t`.
- ✗ O tipo de dados `dev_t` está definido em `<linux/types.h>` e atualmente é representado com 32 bits, onde os 12 bits mais significativos representam o major e os 20 bits restantes representam o minor.





# TIPO DE DADOS dev\_t

- Algumas macros são disponibilizadas para gerenciar variáveis do tipo `dev_t`:

```
/* creating a device number */
dev_t mydev = MKDEV(major, minor);

/* extracting major number */
MAJOR(mydev);

/* extracting minor number */
MINOR(mydev);
```





# REGISTRANDO ESTATICAMENTE

- ✗ O major number e o minor number podem ser registrados estaticamente ou dinamicamente.
- ✗ Você pode registrar estaticamente através da função `register_chrdev_region()`.





# FUNÇÃO register\_chrdev\_region()

```
#include <linux/fs.h>

/* allocate device number statically */
int register_chrdev_region(dev_t from,
                           unsigned count,
                           const char *name);

/* example */
static dev_t mydriver_dev = MKDEV(202, 128);

if (register_chrdev_region(mydriver_dev, 4, "mydriver")) {
    pr_err("Failed to allocate device number\n");
    [...]
}
```





# REGISTRANDO DEVICE NUMBER DINÂMICO

- ✗ Como você pode não saber quais dispositivos estão presentes no sistema, pode haver conflito ao tentar alocar estaticamente um major ou minor number já usado por outro dispositivo.
- ✗ Portanto, pode ser melhor alocar dinamicamente com a função `alloc_chrdev_region()`.





# FUNÇÃO alloc\_chrdev\_region()

```
#include <linux/fs.h>

/* allocate device number dynamically */
int alloc_chrdev_region(dev_t *dev,
                        unsigned baseminor,
                        unsigned count,
                        const char *name);

/* example */
static dev_t mydriver_dev;

if (alloc_chrdev_region(&mydriver_dev, 0, 4, "mydriver")) {
    pr_err("Failed to allocate device number\n");
    [...]
}
```





# REGISTRANDO MAJOR/MINOR (cont.)

- Os dispositivos registrados são visíveis em /proc/devices:

```
# cat /proc/devices
Character devices:
  1 mem
  5 /dev/tty
 13 input
 14 sound
[...]
```

```
Block devices:
  1 ramdisk
  7 loop
  8 sd
  9 md
 11 sr
[...]
```





# IMPLEMENTANDO UM CHAR DEVICE (2)

- ✗ Para implementar um driver de dispositivo do tipo caractere, existem três passos principais:
  1. Reservar o major number e o(s) minor number(s) do seu driver.
  2. Implementar as operações que serão disponibilizadas aos usuários do seu driver (open, read, write, close, etc) e associar estas operações à uma estrutura do tipo `file_operations`.
  3. Associar o major e o minor number alocados à sua estrutura de operações de arquivo e registrar o dispositivo de caractere.





# FILE OPERATIONS

- ✗ A estrutura `file_operations` (também chamada de `fops`) contém todas as operações implementadas pelo device driver.
- ✗ Como ela é genérica para todos os arquivos gerenciados pelo kernel, nem todas as operações definidas nesta estrutura são necessárias para um driver de dispositivo de caractere.





# ESTRUTURA file\_operations

```
#include <linux/fs.h>

/* most important file operations for a char device */
struct file_operations {
    [...]
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    [...]
};
```





# FUNÇÃO open()

```
int open(struct inode *i, struct file *f);
```

- ✗ Chamada quando o arquivo de dispositivo é aberto.
- ✗ A estrutura inode é uma representação única do arquivo no sistema (seja ele um arquivo comum, diretório, link, dispositivo de caractere ou bloco, etc).
- ✗ Uma estrutura do tipo file é criada toda vez que um arquivo é aberto, e armazena informações como a posição corrente do arquivo, modo de abertura, etc.





# FUNÇÃO release()

```
int release(struct inode *i, struct file *f);
```

- ✗ Chamada quando o arquivo de dispositivo é fechado.
- ✗ As estruturas inode e file são as mesmas da função open().





# FUNÇÃO read()

```
ssize_t read(struct file *f, char __user *buf,  
            size_t sz, loff_t *off);
```

- ✗ Chamada quando é realizada uma operação de leitura no arquivo de dispositivo.
- ✗ O driver deverá:
  - ✗ Ler até sz bytes do dispositivo e salvar no buffer buf.
  - ✗ Atualizar a posição atual do arquivo na variável off (opcional).
  - ✗ Retornar a quantidade de bytes lidos.





# FUNÇÃO write()

```
ssize_t write(struct file *f, const char __user *buf,  
             size_t sz, loff_t *off);
```

- ✗ Chamada quando é realizada uma operação de escrita no arquivo de dispositivo.
- ✗ O driver deverá:
  - ✗ Ler sz bytes do buffer buf e escrever no dispositivo.
  - ✗ Atualizar a posição atual do arquivo na variável off (opcional).
  - ✗ Retornar a quantidade de bytes escritos no dispositivo.





# TROCANDO DADOS COM USERSPACE

- ✗ Um código do kernel não pode acessar diretamente uma região de memória de espaço de usuário, seja dereferenciando um ponteiro ou usando funções do tipo `memcpy()`.
  - ✗ O ponteiro precisa estar apontando para um endereço de memória virtual que o processo tenha direito de acesso.
  - ✗ Se o endereço passado é inválido, acontecerá erro de segmentação de memória (`segfault`) e o kernel matará o processo.
- ✗ Portanto, para manter o código do driver portável e seguro, o driver precisa usar algumas funções específicas para trocar dados com o espaço de usuário.





# copy\_to\_user()

```
#include <asm/uaccess.h>

/* Copy n bytes from kernel buffer to user buffer.
   Return 0 on success or a negative number on error.
*/
unsigned long copy_to_user(void __user *to,
                           const void *from,
                           unsigned long n);

/* example */
if (copy_to_user(user_buf, kernel_buf, qtd)) {
    pr_err("Error copying to user buffer...\n");
    [...]
}
```





# copy\_from\_user()

```
#include <asm/uaccess.h>

/* Copy n bytes from user buffer to kernel buffer.
   Return 0 on success or a negative number on error.
*/
unsigned long copy_from_user(void *to,
                           const void __user *from,
                           unsigned long n);

/* example */
if (copy_from_user(kernel_buf, user_buf, qtd)) {
    pr_err("Error copying from user buffer...\n");
    [...]
}
```





# OUTROS MÉTODOS

- ✗ Dependendo da quantidade de dados e do fluxo de comunicação, o uso das funções de troca de dados entre kernel e espaço de usuário pode impactar a performance do sistema.
- ✗ Para estes casos, existem soluções alternativas onde não é necessário realizar a cópia dos buffers (**zero copy**):
  - ✗ Implementar a operação `mmap()` para permitir que um código rodando em espaço de usuário tenha acesso direto à memória do dispositivo.
  - ✗ Usar a função `get_user_pages()` para mapear direto uma página de memória do usuário ao invés de copiá-la.





# EXEMPLO read()

```
static ssize_t mydriver_read(struct file *file, char __user *buf,
                           size_t count, loff_t *ppos)
{
    int transfer_size, qtd;
    char bufrx[256];

    transfer_size = min_t(int, sizeof(bufrx), count);

    qtd = read_device(bufrx, transfer_size);

    if (copy_to_user(buf, bufrx, qtd)) {
        return -EFAULT;
    } else {
        return qtd;
    }
}
```





# EXEMPLO write()

```
static ssize_t mydriver_write(struct file *file, const char __user *buf,
                           size_t count, loff_t *ppos)
{
    int transfer_size;
    char buftx[256];

    transfer_size = min_t(int, sizeof(buftx), count);

    if (copy_from_user(buftx, buf, transfer_size)) {
        return -EFAULT;
    } else {
        write_device(buftx, transfer_size);
        return transfer_size;
    }
}
```





# FUNÇÃO unlocked\_ioctl()

```
long unlocked_ioctl(struct file *f, unsigned int cmd,  
                    unsigned long arg);
```

- ✗ Esta operação está associada à chamada de sistema ioctl(), e permite estender as capacidades do driver além da API de leitura e escrita em arquivos.
- ✗ Exemplos de utilização em drivers: configurar o baud rate de uma porta serial, configurar a resolução de uma placa de vídeo, etc.
- ✗ O comando a ser executado é passado no parâmetro cmd, e pode-se usar o parâmetro arg para passar alguma informação adicional.
- ✗ A semântica dos argumentos cmd e arg é específica de cada driver.





# EXEMPLO unlocked\_ioctl()

```
static long phantom_ioctl(struct file *file, unsigned int cmd,
                           unsigned long arg)
{
    struct phm_reg r;
    void __user *argp = (void __user *)arg;

    switch (cmd) {
    case PHN_SET_REG:
        if (copy_from_user(&r, argp, sizeof(r)))
            return -EFAULT;
        /* do something */
        break;

    case PHN_GET_REG:
        if (copy_to_user(argp, &r, sizeof(r)))
            return -EFAULT;
        /* do something */
        break;

    default:
        return -ENOTTY;
    }
    return 0;
}
```





# EXEMPLO APLICAÇÃO COM ioctl()

```
int main(void)
{
    int fd, ret;
    struct phm_reg reg;

    fd = open("/dev/phantom");
    assert(fd > 0);

    reg.field1 = 42;
    reg.field2 = 67;

    ret = ioctl(fd, PHN_SET_REG, &reg);
    assert(ret == 0);

    return 0;
}
```





# DEFININDO FILE OPERATIONS

- Para definir a estrutura de operações em arquivo, basta declará-la e inicializá-la com os ponteiros para as operações implementadas.

```
#include <linux/fs.h>

static struct file_operations mydriver_fops =
{
    .owner      = THIS_MODULE,
    .open       = mydriver_open,
    .release   = mydriver_release,
    .read       = mydriver_read,
    .write      = mydriver_write,
};
```





# IMPLEMENTANDO UM CHAR DEVICE (3)

- ✗ Para implementar um driver de dispositivo do tipo caractere, existem três passos principais:
  1. Reservar o major number e o(s) minor number(s) do seu driver.
  2. Implementar as operações que serão disponibilizadas aos usuários do seu driver (open, read, write, close, etc) e associar estas operações à uma estrutura do tipo `file_operations`.
  3. Associar o major e o minor number alocados à sua estrutura de operações de arquivo e registrar o dispositivo de caractere.





# REGISTRANDO O CHAR DEVICE

- ✗ O kernel representa um dispositivo de caractere com a estrutura cdev.
- ✗ Para registrar um dispositivo de caractere, primeiro declare globalmente uma estrutura do tipo cdev e inicialize-a chamando a função cdev\_init().
- ✗ Depois é só adicionar o dispositivo de caractere ao sistema com a função cdev\_add().
- ✗ Depois desta chamada, o kernel associará o major/minor number registrado com as operações em arquivo definidas. Seu dispositivo está pronto para ser usado!





# EXEMPLO CRIANDO CHAR DEVICE

```
#include <linux/cdev.h>

static struct cdev mydriver_cdev;

static int __init mydriver_init(void)
{
    [...]

    cdev_init(&mydriver_cdev, &mydriver_fops);

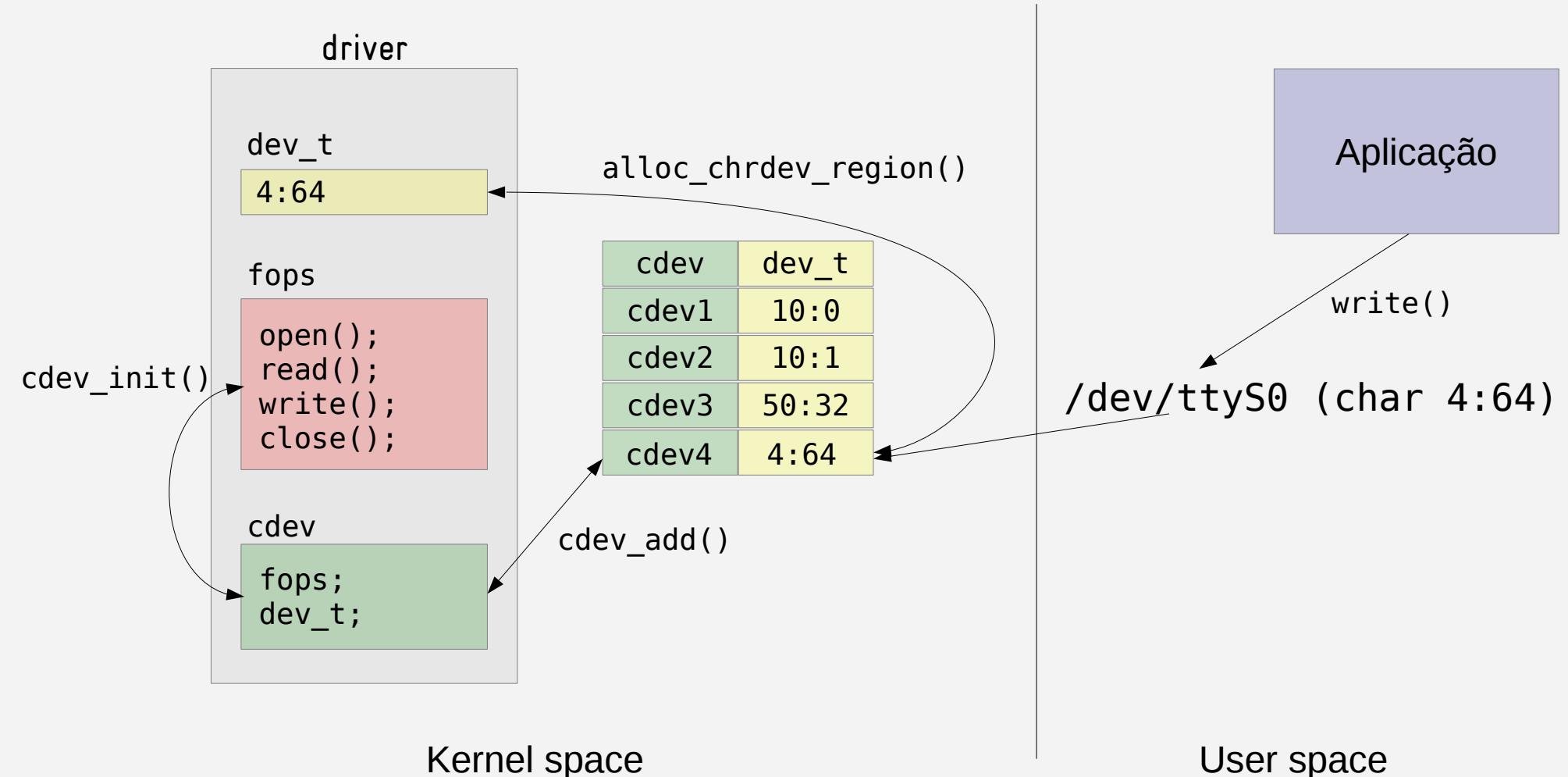
    if (cdev_add(&mydriver_cdev, mydriver_dev, 1)) {
        pr_err("Char driver registration failed\n");
        [...]
    }

    [...]
}
```





# DIAGRAMA CHAR DEVICE





# DESREGISTRANDO

- ✗ Na função de limpeza do driver é necessário desregistrar o dispositivo de caractere.
- ✗ Para isso, primeiro remova o dispositivo de caractere com a função `cdev_del()`.
- ✗ E depois libere o major/minor number alocado com a função `unregister_chrdev_region()`.





# EXEMPLO DESREGISTRANDO CHAR DEVICE

```
#include <linux/cdev.h>

static void __exit mydriver_exit(void)
{
    [...]

    cdev_del(&mydriver_cdev);

    unregister_chrdev_region(mydriver_dev, 1);

    [...]
}
```





# CÓDIGOS DE ERRO DO LINUX

- ✗ A convenção do Linux para tratamento de erros é a seguinte:
  - ✗ Retornar 0 no caso de sucesso.
  - ✗ Retornar um número negativo no caso de erro.
- ✗ Regra geral: sempre trate o retorno das funções e retorne um código de erro compatível com o problema apresentado.
- ✗ Os códigos de erro padrão estão disponíveis em:  
`asm-generic/errno-base.h`  
`asm-generic/errno.h`
- ✗ Na dúvida sobre qual código de erro retornar, consulte os fontes do kernel!





Embedded Labworks

# LABORATÓRIO

O primeiro driver



Embedded Labworks

# Linux device drivers

Gerenciamento de memória



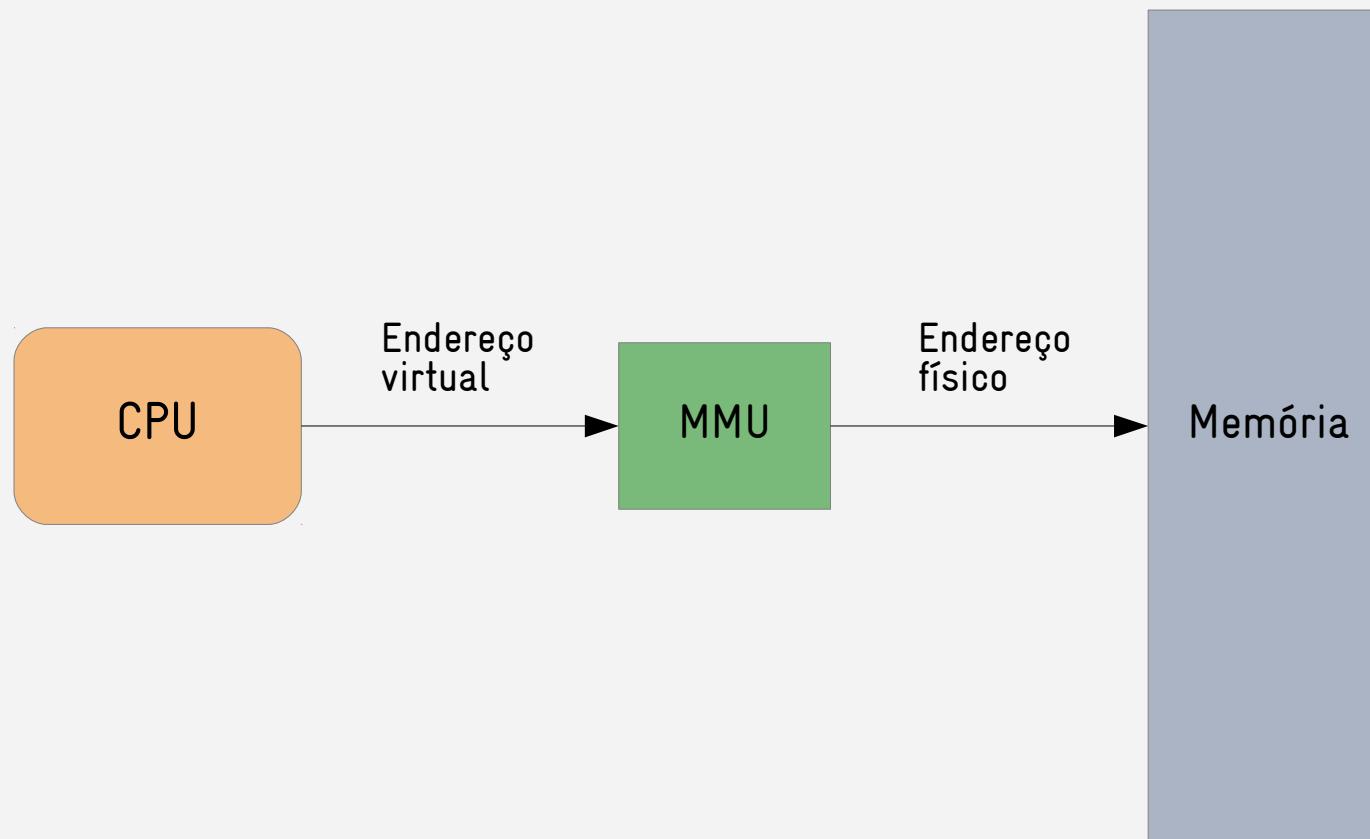
# MEMÓRIA

- ✗ O desenvolvimento de alguns tipos de drivers, principalmente aqueles que precisam de performance, requerem um conhecimento maior de como o sub-sistema de memória virtual do Linux funciona.
- ✗ O sub-sistema de memória virtual no Linux é fortemente apoiado em componente de hardware chamado MMU.
- ✗ A MMU (Memory Management Unit) é o hardware que implementa o mecanismo de memória virtual, gerenciando a memória do sistema e fazendo a conversão entre endereços de memória físicos e virtuais.





# MEMORY MANAGEMENT UNIT





# VANTAGENS DA MMU

- ✗ Um sistema com MMU é capaz de prover:
  - ✗ Maior endereçamento de memória para os processos. Em uma arquitetura de 32 bits, os processos podem ter acesso à um endereçamento linear de até 3G de memória virtual.
  - ✗ SWAP: se faltar memória física, possibilita salvar e recuperar páginas de memória do disco.
  - ✗ Gerenciamento dinâmico da memória do processo (stack, heap, etc).
  - ✗ Compartilhamento: os processos podem compartilhar memória (código, dados, etc), usado por exemplo em mecanismos de IPC.
  - ✗ Proteção: cada processo só enxerga seu espaço de endereçamento, onde um acesso inválido gera uma exceção.





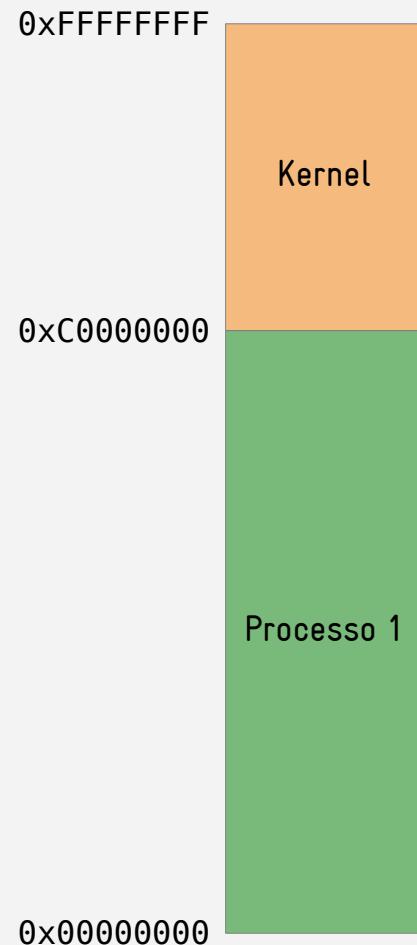
# COMO FUNCIONA NO LINUX?

- ✗ O kernel divide o espaço de endereçamento de memória virtual em duas partes:
  - ✗ Endereçamento de memória virtual do kernel.
  - ✗ Endereçamento de memória virtual dos processos.
- ✗ Esta divisão é configurada em tempo de compilação no kernel em Memory Split, com três opções 1G/3G, 2G/2G e 3G/1G.
- ✗ Por exemplo, um sistema de 32 bits consegue endereçar até 4G de memória virtual. Se configurado com 3G/1G, temos 1G de memória endereçável pelo kernel e 3G de memória endereçável pelos processos.





# ORGANIZAÇÃO DA MEMÓRIA 3G/1G

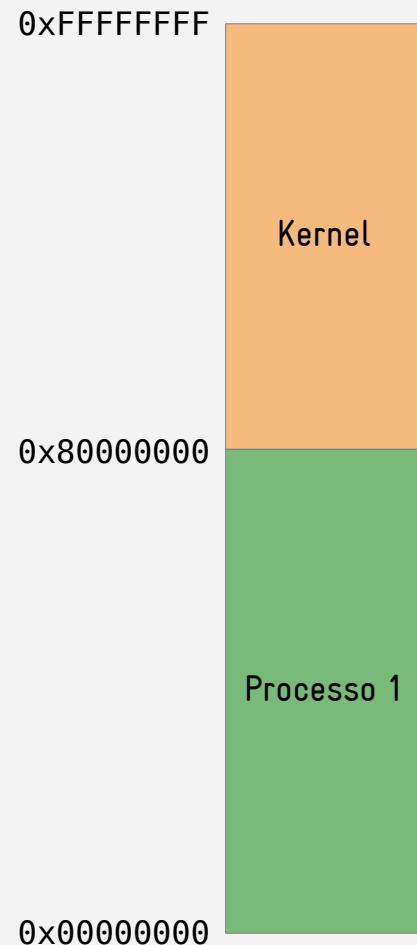


- ✗ 1GB do endereçamento de memória virtual é reservado para o kernel, contendo o código do kernel e suas estruturas de dados principais.
- ✗ 3GB do endereçamento de memória virtual é reservado para cada processo, para armazenar código, dados (stack, heap, etc), arquivos mapeados em memória, etc.
- ✗ Não necessariamente um endereço virtual reservado para um processo pode estar mapeado para um endereço físico!





# ORGANIZAÇÃO DA MEMÓRIA 2G/2G

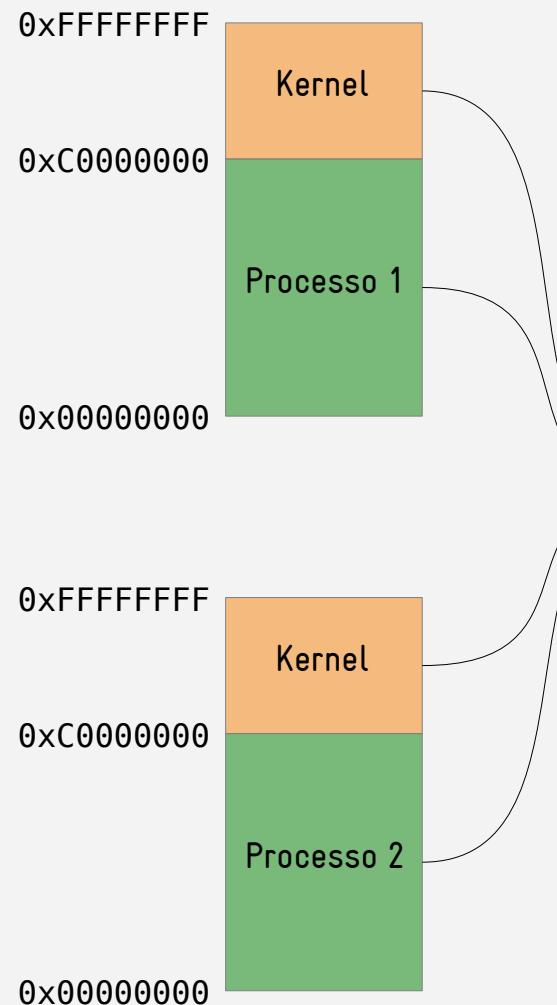


- ✗ 2GB do endereçamento de memória virtual é reservado para o kernel, contendo o código do kernel e suas estruturas de dados principais.
- ✗ 2GB do endereçamento de memória virtual é reservado para cada processo, para armazenar código, dados (stack, heap, etc), arquivos mapeados em memória, etc.
- ✗ Não necessariamente um endereço virtual reservado para um processo pode estar mapeado para um endereço físico!

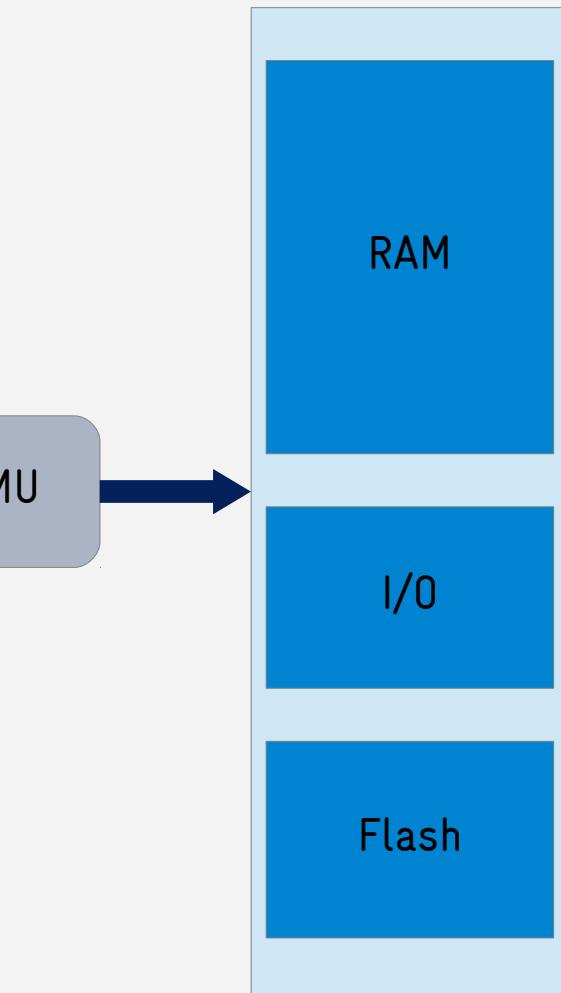




### Endereçamento Virtual de Memória



### Endereçamento Físico de Memória





# MEMÓRIA FÍSICA

- ✗ Apesar da menor unidade de memória endereçável pela CPU ser um byte ou uma palavra (word), o kernel divide a memória física em páginas de memória.
- ✗ Isso porque, para gerenciar a memória e realizar a conversão de endereços virtuais para endereços físicos, a MMU divide a memória em páginas e mantém uma tabela de páginas de memória do sistema.
- ✗ O tamanho de uma página de memória pode variar dependendo da arquitetura (tipicamente 4K em arquiteturas de 32 bits e 8K em arquiteturas de 64 bits).
- ✗ Isso significa que, em um sistema de 32 bits, com páginas de 4K e 1G de memória física, teremos 262.144 páginas físicas de memória.





# ZONAS DE MEMÓRIA

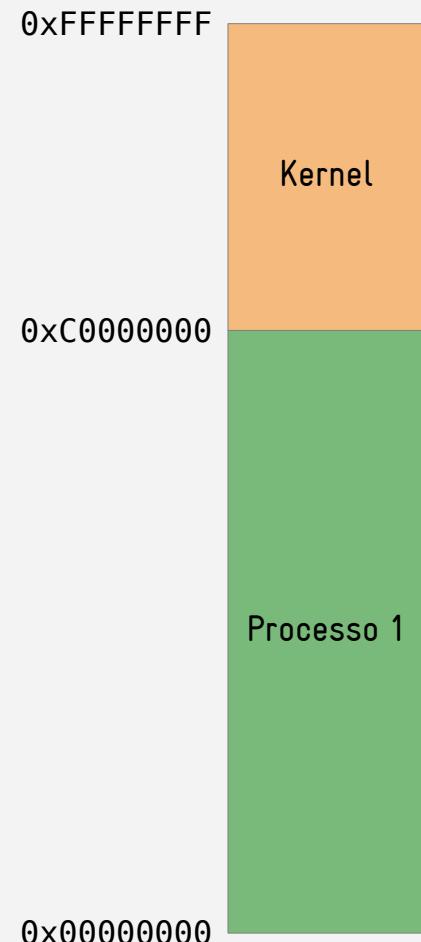
- ✗ O kernel divide as páginas de memória física em diferentes zonas:
  - ✗ ZONE\_DMA e ZONE\_DMA32: páginas de memória para operações de DMA.
  - ✗ ZONE\_NORMAL: páginas de memória mapeadas normalmente no espaço do kernel.
  - ✗ ZONE\_HIGHMEM: páginas de memória que não podem ser mapeadas no espaço de endereçamento do kernel. São alocadas dinamicamente, conforme a necessidade.
- ✗ Desta forma, quando por exemplo um driver precisa alocar memória para realizar uma operação de DMA, ele irá alocar de ZONE\_DMA.
- ✗ O uso e o layout das zonas de memória são totalmente dependentes de arquitetura.



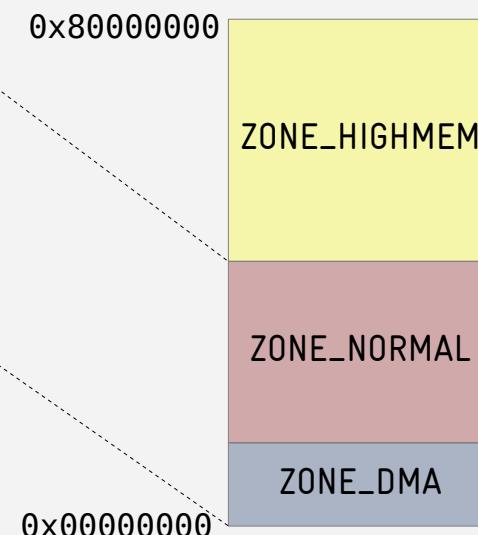


# EXEMPLO DE MAPEAMENTO EM 32 BITS

Espaço de endereçamento virtual (4G)



Espaço de endereçamento físico (2G)





# MEMÓRIA PARA OS PROCESSOS

- ✗ A memória para os processos é alocada das zonas ZONE\_HIGHMEM (se existente) ou ZONE\_NORMAL.
- ✗ Durante uma alocação de memória para um processo, o kernel não necessariamente aloca fisicamente esta página para o processo.
- ✗ Neste caso, o kernel pode usar uma funcionalidade chamada **demand fault paging** para alocar dinamicamente uma página de memória, através da exceção **page fault** gerada pela MMU quando o processo tentar acessar esta região de memória.





# MEMÓRIA PARA OS PROCESSOS (cont.)

- ✗ É permitido à uma aplicação alocar mais memória do que a disponível fisicamente no sistema (mas pode levar à falta de memória).
- ✗ Neste caso, o OOM Killer (Out of Memory Killer) entra em ação e seleciona um processo para matar e ganhar um pouco mais de memória (melhor do que travar!)





# STACK

- ✗ O tamanho do stack de um processo rodando em user space é dinâmico, podendo aumentar ou diminuir conforme a necessidade. Mas o stack de um processo rodando em kernel space é limitado!
- ✗ Dependendo da plataforma, o tamanho do stack do kernel é limitado à uma ou duas páginas de memória (4K/8K). Por exemplo, em ARM o padrão é 8K (procure por THREAD\_SIZE em arch/arm/).
- ✗ Isso significa que você deve evitar chamadas recursivas ou alocações estáticas de buffers muito grandes dentro do kernel!
- ✗ Em contexto de interrupção é usado um stack separado do stack usado em contexto de processo (para cada CPU do sistema).





Embedded Labworks

# Linux device drivers

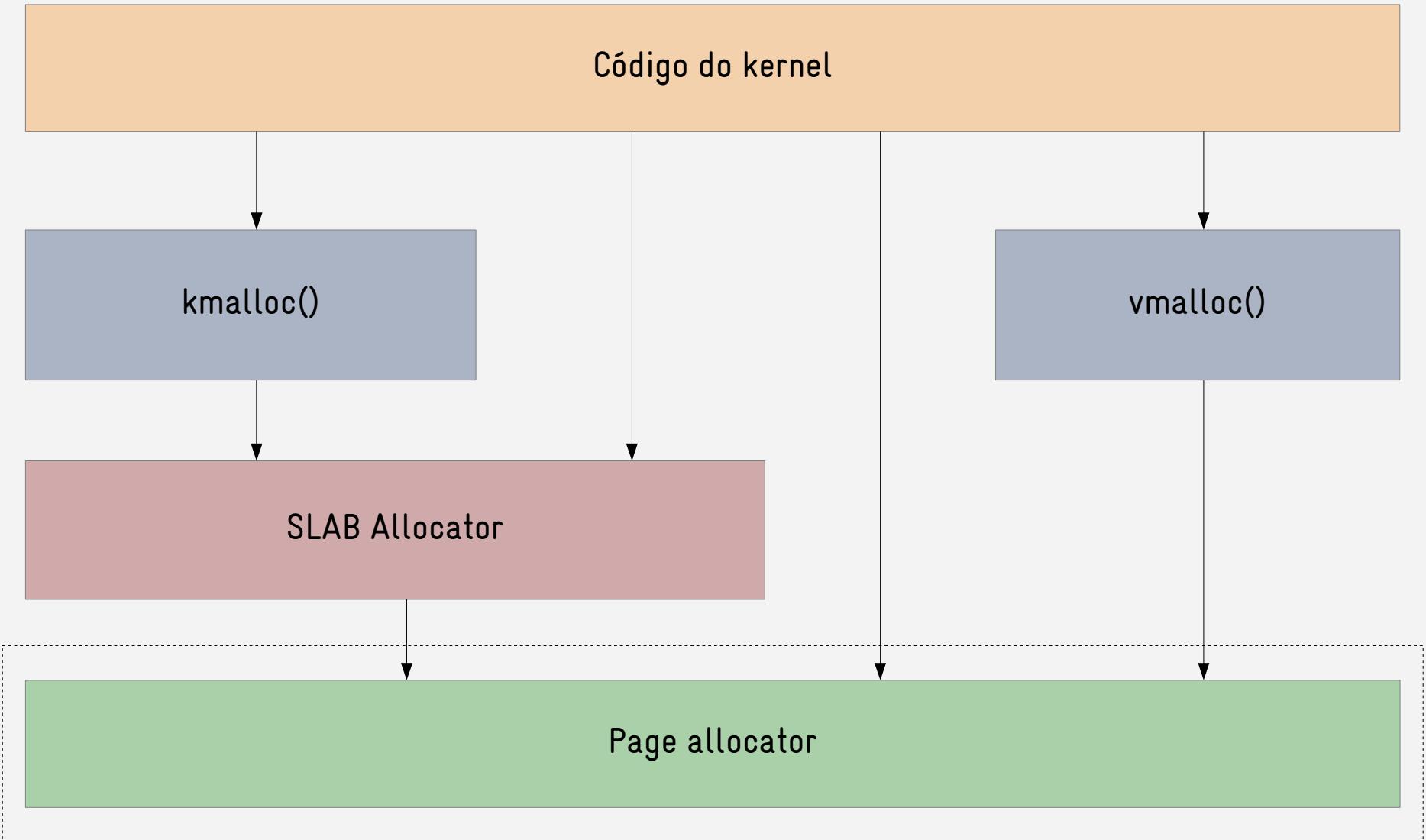
Alocação de memória



# ALOCANDO MEMÓRIA

- ✗ Existem basicamente quatro mecanismos de alocação de memória no kernel:
  - ✗ Page allocator: funções de baixo nível que permitem alocações na granularidade de páginas de memória (normalmente 4KB em 32 bits).
  - ✗ SLAB Allocator: usa as funções do page allocator para criar caches de alocação de memória, permitindo alocar objetos menores que uma página de memória.
  - ✗ kmalloc(): mecanismo mais comum de alocação de memória, que usa as funções do SLAB allocator para alocar memória.
  - ✗ vmalloc(): usa as funções do page allocator, permitindo alocar regiões não-contínuas de memória física.



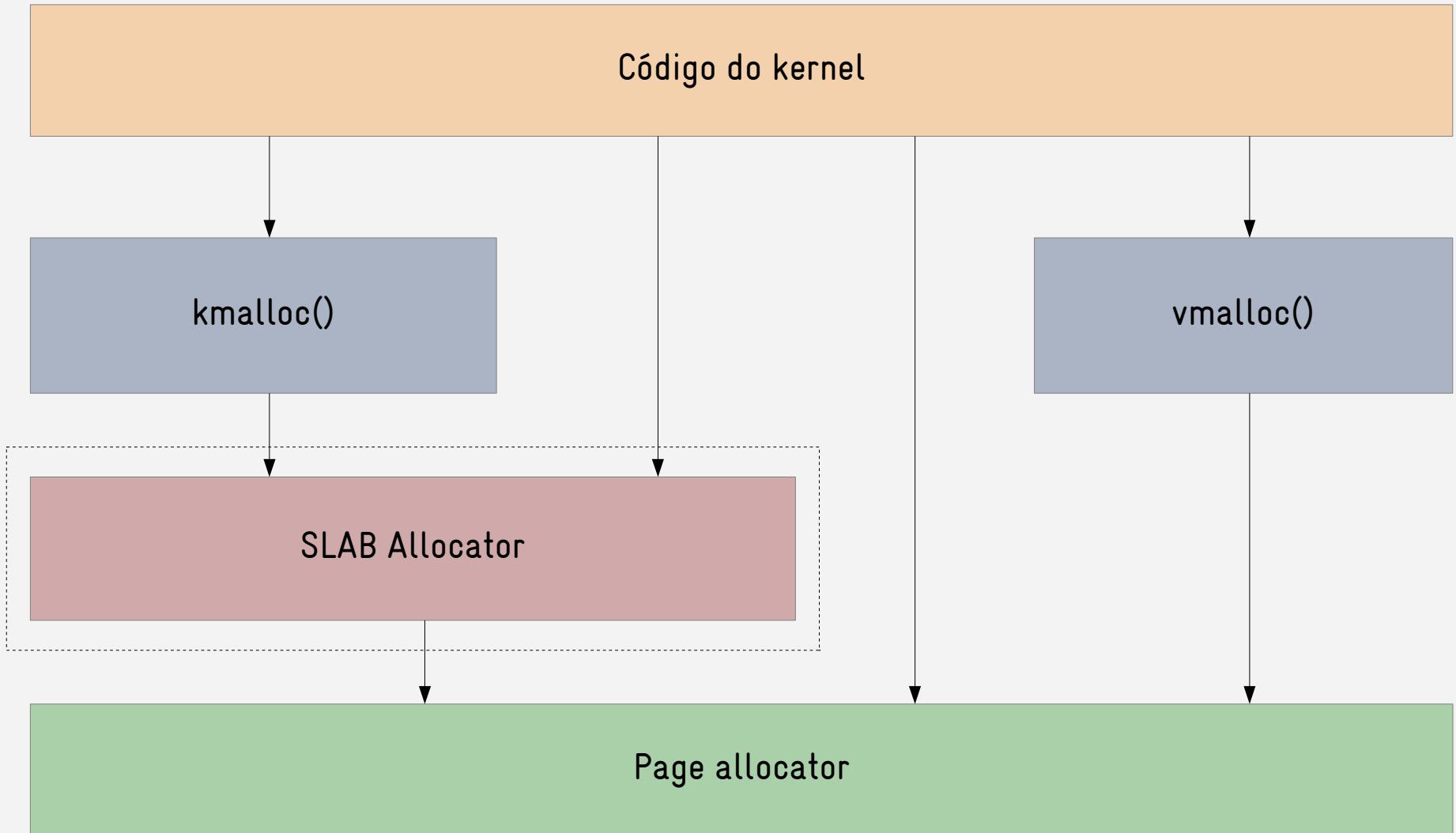




# PAGE ALLOCATOR

- ✗ Trabalha com alocação de páginas de memória (normalmente 4K em 32 bits) e trabalha apenas em potência de 2 (1 página, 2 páginas, 4 páginas, 8 páginas, etc).
- ✗ Apropriado para alocações de grandes regiões de memória (maiores que 128K).
- ✗ Aloca no máximo 8MB (é possível mudar este valor na configuração do kernel).
- ✗ Aloca uma região de memória virtual contínua, e também fisicamente contínua (o que pode ser um problema se a memória estiver muito fragmentada).



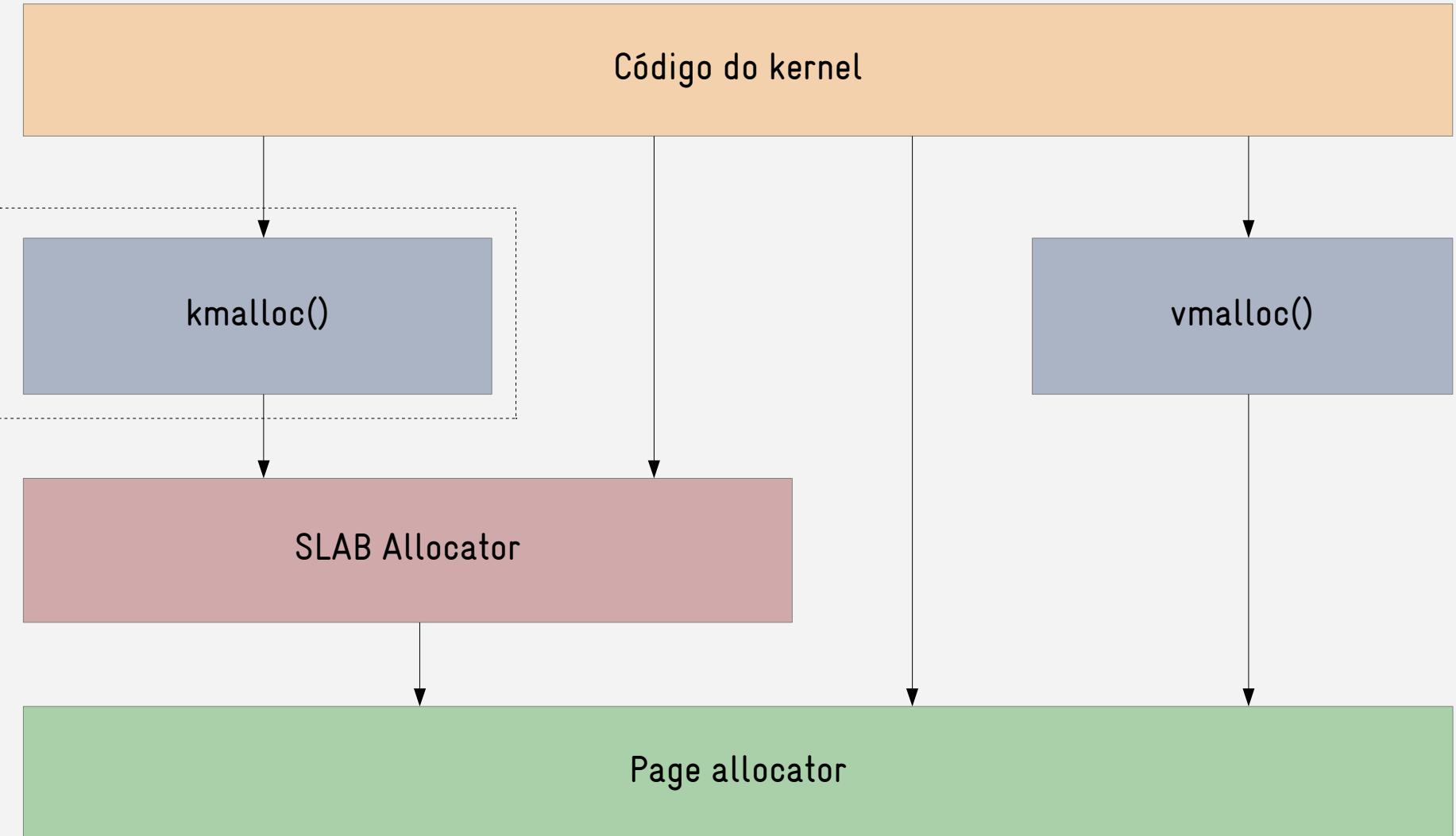




# SLAB ALLOCATOR

- ✗ Camada que usa a API da page allocator para criar um cache de alocação de memória, permitindo alocar objetos menores que o tamanho de uma página de memória.
- ✗ Existem três implementações da camada SLAB (todas implementam a mesma API):
  - ✗ SLAB: versão original.
  - ✗ SLOB: mais simples, economiza espaço, mas não escala bem.
  - ✗ SLUB: mecanismo padrão a partir da versão 2.6.23. Simples e escala melhor que o SLAB, gerando menos fragmentação de memória.
- ✗ Sua API é usada internamente no kernel (vide `<linux/slab.h>`), mas dificilmente será usada em um driver comum.







# KMALLOC

- ✗ Em um driver comum, o mecanismo padrão de alocação é através da função kmalloc().
- ✗ Permite alocar objetos de 8 bytes a 128KB (o valor máximo pode depender da arquitetura).
- ✗ Os caches de alocação de memória podem ser visualizados em /proc/slabinfo (necessário habilitar SLUB\_DEBUG).
- ✗ A área alocada será fisicamente contínua e arredondada à potência de 2.





# KMALLOC API

```
#include <linux/slab.h>

/* allocate size bytes and return a pointer
   to the area (virtual address)
   Return NULL on error */
void *kmalloc(size_t size, int flags);

/* free an allocated area */
void kfree(const void *objp);

/* example */
struct ib_update_work *work;
work = kmalloc(sizeof *work, GFP_ATOMIC);
[...]
kfree(work);
```





# FLAGS

- ✗ Estas são as flags mais comuns em alocação de memória:
  - ✗ GFP\_KERNEL: Flag padrão para alocação de memória, usado na maioria das situações. Pode bloquear o processo se não houver memória disponível.
  - ✗ GFP\_ATOMIC: Usada em situações onde não se pode bloquear (seções críticas ou rotinas de tratamento de interrupção).
  - ✗ GFP\_DMA: Usada quando é necessário alocar memória da zona de DMA.
- ✗ Outras flags estão disponíveis em <linux/gfp.h>.





# KMALLOC API (cont.)

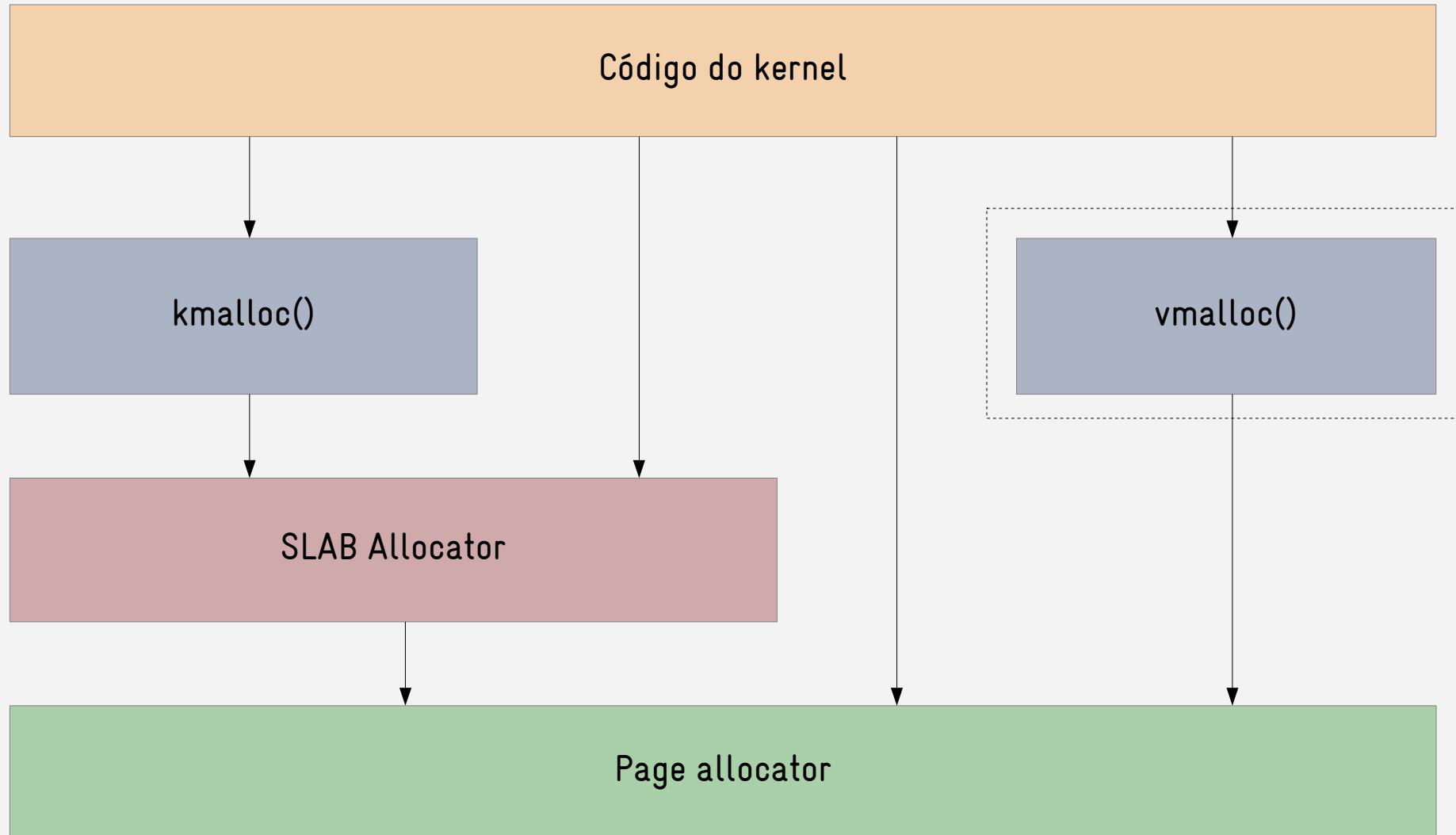
```
#include <linux/slab.h>

/* allocates a zero-initialized buffer */
void *kzalloc(size_t size, gfp_t flags);

/* allocates memory for an array of n elements of size
   size, and zeroes its contents */
void *kcalloc(size_t n, size_t size, gfp_t flags);

/* changes the size of the buffer pointed by p to new_size,
   by reallocating a new buffer and copying the data,
   unless the new_size fits within the alignment of the
   existing buffer. */
void *krealloc(const void *p, size_t new_size, gfp_t flags);
```







# VMALLOC ALLOCATOR

- ✗ Permite alocar um endereço de memória virtual contínuo, mas fisicamente ele pode não ser contínuo.
- ✗ O valor alocado é arredondado em páginas de memória.
- ✗ Quando usar? Em alocações de grandes áreas de memória (maiores que 128K) já que não existe a possibilidade de uma falha na alocação por fragmentação de memória.
- ✗ Desvantagens: a região alocada é arredondada em páginas de memória, baixa performance, não pode ser usado em DMA.





# VMALLOC ALLOCATOR API

```
#include <linux/vmalloc.h>

/* allocate size bytes and returns a virtual address
   (memory is rounded to page size) */
void *vmalloc(unsigned long size);

/* free allocated memory */
void vfree(void *addr);
```





# KERNEL MEMORY DEBUGGING

- ✗ KMEMCHECK: verifica se uma região de memória alocada dinamicamente é acessada antes de ser inicializada. Disponível apenas no x86 (32/64 bits). Documentação em:  
[Documentation/kmemcheck.txt](#)
- ✗ DEBUG\_KMEMLEAK: verifica dinamicamente por memory leaks, disponível em todas as arquiteturas. Documentação em:  
[Documentation/kmemleak.txt](#)
- ✗ Estas funcionalidades adicionam um overhead na execução do kernel. Use apenas em desenvolvimento!





Embedded Labworks

# Linux device drivers

Hardware I/O



# ACESSANDO I/O

- ✗ Existem basicamente dois mecanismos de acesso à I/O, de acordo com a arquitetura da CPU:
  - ✗ Port I/O: quando a CPU utiliza barramentos de endereços diferentes para acessar memória e I/O. Neste caso, o acesso à I/O é realizado através do uso de instruções especiais da CPU (ex: instruções IN e OUT na arquitetura x86).
  - ✗ Memory-mapped I/O: quando a CPU utiliza o mesmo barramento de endereços para endereçar memória e I/O. Neste caso, o acesso é realizado normalmente através de instruções de acesso à memória. É o mecanismo mais usado por diferentes arquiteturas suportadas pelo Linux.





# PORT I/O

- ✗ Antes de usar um port I/O, o driver deve requisitar ao kernel a região de I/O que deseja usar com a função `request_region()`.
- ✗ Isso evita que outros drivers usem a mesma região de I/O (mas é um procedimento puramente voluntário).
- ✗ Você pode listar todos os I/Os registrados pelos drivers no arquivo `/proc/ioports`.
- ✗ Depois de usada, a região de I/O deve ser liberada com a função `release_region()`.





# PORT I/O API

```
#include <linux/ioport.h>

/* request I/O region */
struct resource *request_region(unsigned long start,
                                unsigned long len,
                                char *name);

/* release I/O region */
void release_region(unsigned long start, unsigned long len);
```





# PORT I/O API

```
#include <linux/ioport.h>

/* example: requesting I/O port */
if (request_region(0x03f8, 8, "serial") == NULL) {
    pr_err("Failed requesting I/O region!\n");
    [...]
}

/* example: releasing I/O port */
release_region(0x03f8, 8);
```





# PORT I/O API (cont.)

```
#include <asm/io.h>

/* read/write bytes (8 bits) */
unsigned inb(unsigned long *addr);
void outb(unsigned port, unsigned long *addr);

/* read/write words (16 bits) */
unsigned inw(unsigned long *addr);
void outw(unsigned port, unsigned long *addr);

/* read/write longs (32 bits) */
unsigned inl(unsigned long *addr);
void outl(unsigned port, unsigned long *addr);
```





# PORT I/O API (cont.)

```
#include <asm/io.h>

/* read/write multiple bytes (8 bits) */
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);

/* read/write multiple words (16 bits) */
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);

/* read/write multiple longs (32 bits) */
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```





# PORT I/O API (cont.)

```
#include <asm/io.h>

/* example: read 8 bits */
oldlcr = inb(baseio + UART_LCR);

/* example: write 8 bits */
outb(MOXA_MUST_ENTER_ENCHANCE, baseio + UART_LCR);
```





# MEMORY-MAPPED I/O

- ✗ O primeiro passo para o uso de memory-mapped I/O é requisitar a região de I/O para o kernel através da função `request_mem_region()`, normalmente chamada na inicialização do driver.
- ✗ Esta função irá registrar o uso de determinada região de I/O mapeada em memória.
- ✗ Em `/proc/iomem` temos todas as regiões de I/O mapeadas em memória já reservadas no kernel.
- ✗ Ao descarregar o driver, devemos chamar a função `release_mem_region()` para notificar o kernel de que esta região de I/O mapeado em memória está livre para uso.





# MEMORY-MAPPED I/O API

```
#include <linux/ioport.h>

/* request memory-mapped I/O */
struct resource *request_mem_region(unsigned long start,
                                    unsigned long len,
                                    char *name);

/* release memory-mapped I/O */
void release_mem_region(unsigned long start,
                        unsigned long len);
```





# MEMORY-MAPPED I/O API (cont.)

```
#include <linux/ioport.h>

/* example: requesting memory-mapped I/O */
if (request_mem_region(0x20001000, 0x200, "tec907") == NULL) {
    pr_err("Failed requesting memory-mapped I/O region!\n");
    [...]
}

/* example: releasing memory-mapped I/O */
release_mem_region(0x20001000, 0x200);
```





# MEMORY-MAPPED I/O NA MEMÓRIA VIRTUAL

- ✗ O kernel só trabalha com memória virtual!
- ✗ Portanto, para acessar um endereço de memória correspondente à determinado I/O, precisamos converter o endereço físico desta porta de I/O em um endereço virtual antes de usá-lo.
- ✗ Isso pode ser feito com a função `ioremap()`.





# MEMORY-MAPPED I/O API

```
#include <asm/io.h>

/* get memory-mapped I/O virtual address */
void *ioremap(unsigned long phys_addr, unsigned long size);

/* unmap memory-mapped I/O virtual address */
void iounmap(void *address);
```





# MEMORY-MAPPED I/O API (cont.)

```
#include <asm/io.h>

/* example: mapping memory-mapped I/O address */
static void __iomem *tec907_base;
if ((tec907_base = ioremap(0x20001000, 0x200)) == NULL) {
    pr_err("Failed remapping memory-mapped I/O!\\n");
    [...]
}

/* example: unmapping memory-mapped I/O address */
iounmap(tec907_base);
```





# MEMORY-MAPPED I/O API (cont.)

```
#include <asm/io.h>

/* read/write bytes (little-endian access) */
unsigned readb(void *addr);
void writeb(unsigned val, void *addr);

/* read/write words (little-endian access) */
unsigned readw(void *addr);
void writew(unsigned val, void *addr);

/* read/write longs (little-endian access) */
unsigned readl(void *addr);
void	writel(unsigned val, void *addr);
```





# MEMORY-MAPPED I/O API (cont.)

```
#include <asm/io.h>

/* read/write bytes (raw access) */
unsigned __raw_readb(void *addr);
void __raw_writeb(unsigned val, void *addr);

/* read/write words (raw access) */
unsigned __raw_readw(void *addr);
void __raw_writew(unsigned val, void *addr);

/* read/write longs (raw access) */
unsigned __raw_readl(void *addr);
void __raw_writel(unsigned val, void *addr);
```





# MEMORY-MAPPED I/O API (cont.)

```
#include <asm/io.h>

/* example: little-endian byte read */
tmp8 = readb(tec907_base + RD_REG);

/* example: little-endian long write */
writel(data, tec907_base + WR_REG);

/* example: raw byte read */
sense = __raw_readb(tec907_base + 4);

/* example: raw long write */
__raw_writel(1 << KS8695_IRQ_UART_TX,
             tec907_base + REG_TX);
```





Embedded Labworks

# LABORATÓRIO

Acessando o hardware



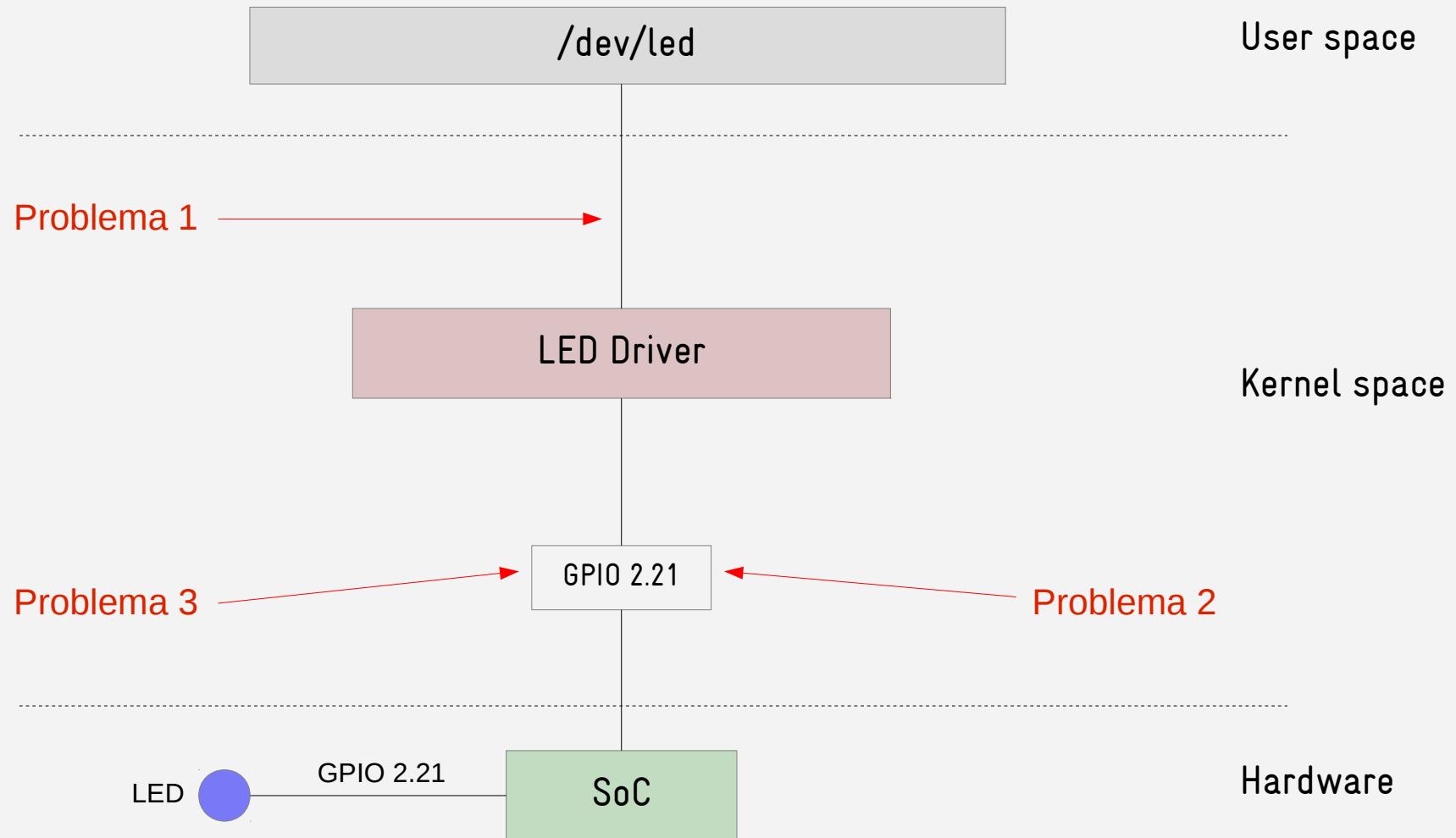
Embedded Labworks

# Linux device drivers

Frameworks



# DRIVER DE DISPOSITIVO DE CARACTERE





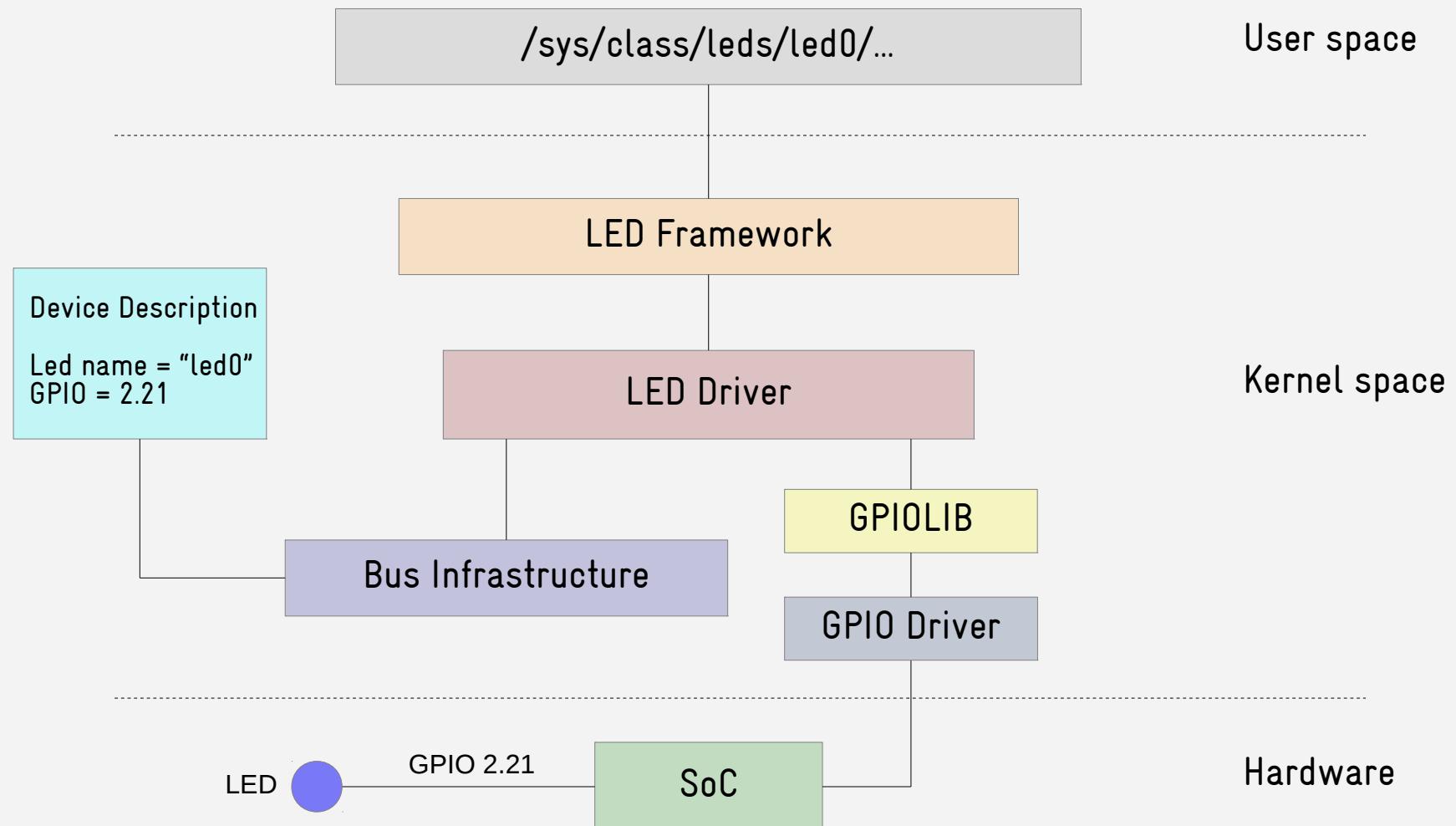
# LINUX KERNEL DRIVER MODEL

- ✗ A solução para estes problemas é o modelo de desenvolvimento de drivers do kernel (Linux Kernel Driver Model).
- ✗ Disponível a partir da versão 2.6 do kernel, provê algumas camadas de abstração para o desenvolvimento de drivers.
- ✗ Contém dois principais componentes:
  - ✗ A interface exportada pelo driver é padronizada através de frameworks.
  - ✗ A separação entre o driver e a descrição do dispositivo de hardware é realizada através de uma infraestrutura de barramento.





# LINUX KERNEL DRIVER MODEL (cont.)





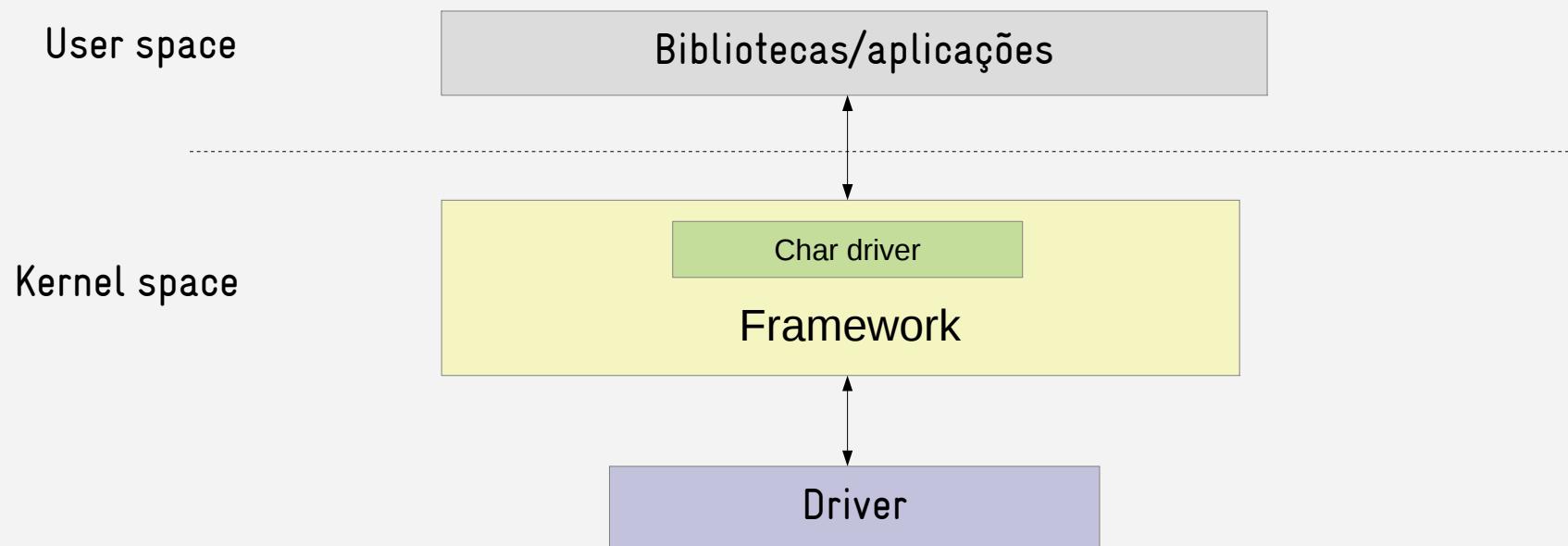
# KERNEL FRAMEWORKS

- ✗ Desenvolver drivers usando diretamente a API de dispositivos de caractere podem trazer alguns problemas, dentre eles:
  - ✗ Falta de padrão para dispositivos do mesmo tipo ou categoria.
  - ✗ Duplicação de código.
- ✗ Por estes motivos, os drivers não são normalmente implementados através de um driver de dispositivo de caractere.
- ✗ Os drivers são normalmente implementados usando um **framework** do kernel.



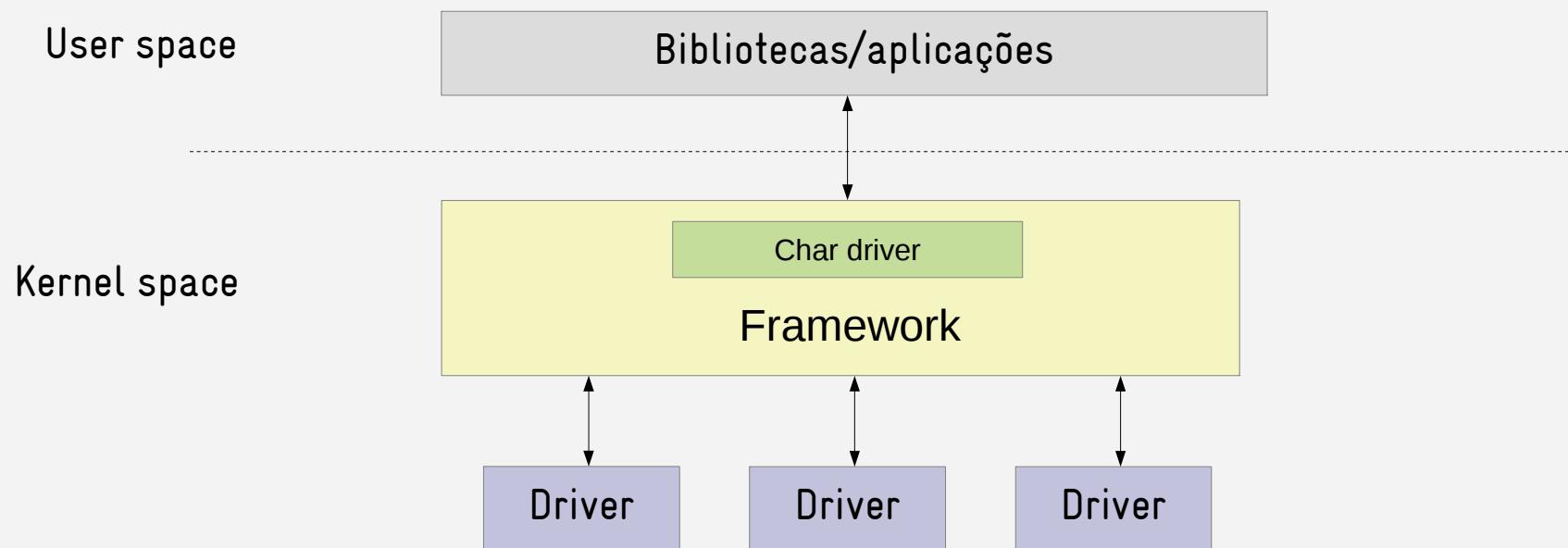


# DIAGRAMA FRAMEWORKS



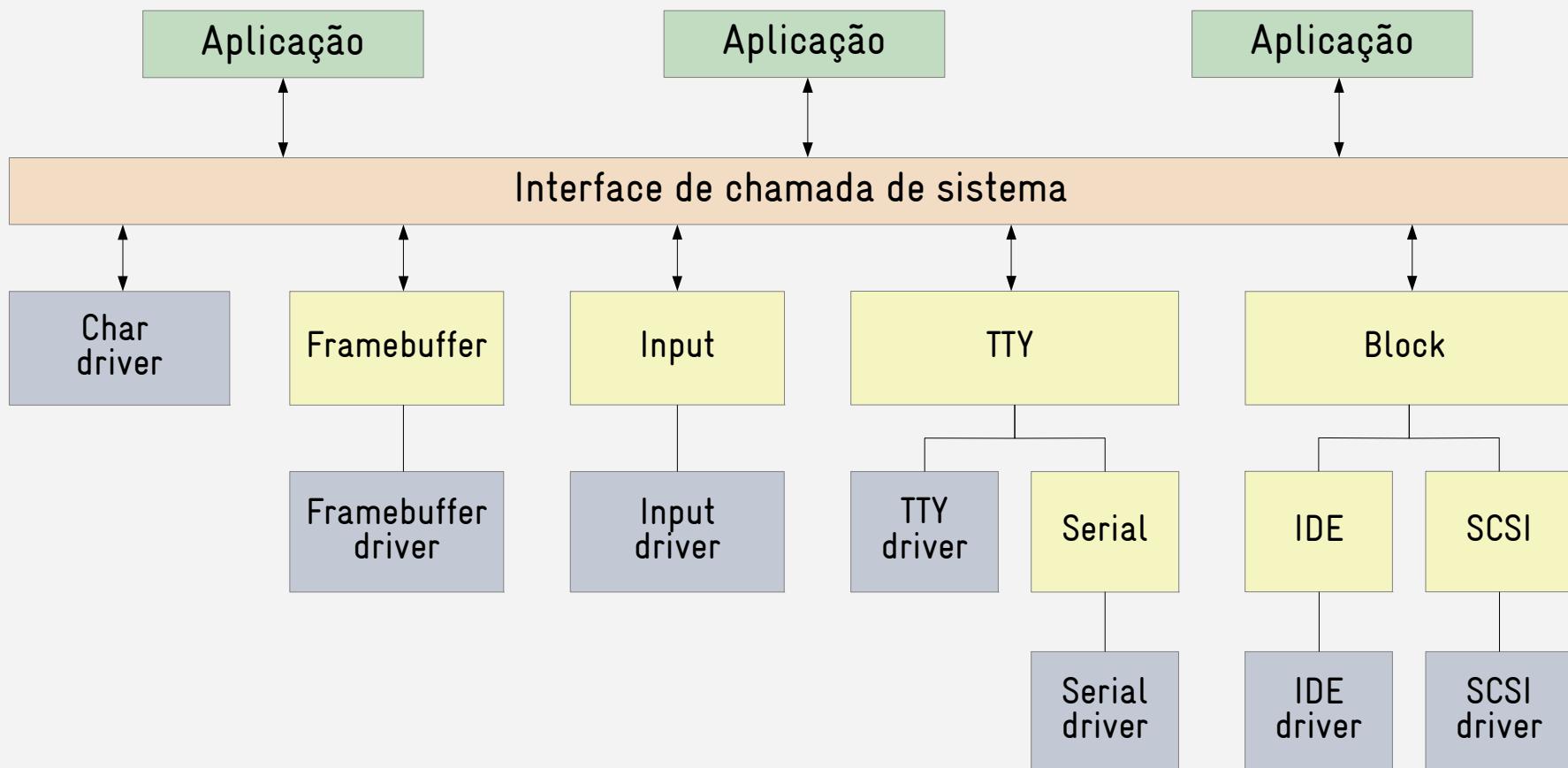


# DIAGRAMA FRAMEWORKS (cont.)





# ALGUNS FRAMEWORKS





# FRAMEBUFFER

- ✗ O framebuffer é um framework para dispositivos de saída de vídeo (monitor, display LCD, etc).
- ✗ Habilitado na opção do kernel `CONFIG_FB`.
- ✗ Fontes disponíveis em `drivers/video/` e definição da API em `<linux/fb.h>`.
- ✗ Implementa um dispositivo de caractere (`/dev/fbX`), onde as aplicações podem ler, escrever e enviar comandos `ioctl()` para o dispositivo.





# FRAMEBUFFER (cont.)

- ✗ Tudo o que um driver de framebuffer precisa fazer é:
  - ✗ Definir e inicializar uma estrutura do tipo `fb_info`.
  - ✗ Prover um conjunto de operações que podem ser realizadas no dispositivo através da implementação da estrutura `fb_ops`.
  - ✗ Registrar o dispositivo de framebuffer com a função `register_framebuffer()`.
- ✗ Existe um modelo de driver de framebuffer disponível em `drivers/video/skeletonfb.c`.





# STRUCT FB\_OPS

```
#include <linux/fb.h>

static struct fb_info *info;

static struct fb_ops xxxfb_ops = {
    .owner        = THIS_MODULE,
    .fb_open      = xxxfb_open,
    .fb_read      = xxxfb_read,
    .fb_write     = xxxfb_write,
    .fb_release   = xxxfb_release,
    .fb_blank     = xxxfb_blank,
    .fb_fillrect = xxxfb_fillrect,
    .fb_copyarea = xxxfb_copyarea,
    .fb_sync      = xxxfb_sync,
    .fb_ioctl     = xxxfb_ioctl,
    .fb_mmap     = xxxfb_mmap,
    [...]
};
```





# REGISTRANDO O FRAMEBUFFER

```
static int __devinit xxxfb_probe(struct pci_dev *dev,
                                 const struct pci_device_id *ent)
{
    [...]

    info = framebuffer_alloc(sizeof(struct xxx_par),
                           device);
    info->fbops = &xxxfb_ops;

    if (register_framebuffer(info) < 0)
        return -EINVAL;

    [...]
}
```





# INPUT

- ✗ A camada input é responsável pelos dispositivos que geram eventos de entrada do usuário (teclado, mouse, touchscreen, joystick, etc).
- ✗ Habilitada na opção do kernel CONFIG\_INPUT.
- ✗ Fontes disponíveis em drivers/input/ e definição da API em <linux/input.h>.





# LENDO OS EVENTOS DE INPUT

- ✗ O framework de input implementa um driver de dispositivo de caractere em /dev/input/eventX, onde X é o número do dispositivo de entrada.
- ✗ As aplicações monitoram os eventos de entrada através da leitura destes arquivos.
- ✗ Cada leitura irá retornar uma estrutura do tipo `input_event`:

```
struct input_event {  
    struct timeval time;  
    unsigned short type;  
    unsigned short code;  
    unsigned int value;  
};
```





# DRIVER DE INPUT

- ✗ Tudo o que um driver de input precisa fazer é:
  - ✗ Definir e inicializar uma estrutura do tipo `input_dev`.
  - ✗ Registrar o dispositivo de input com a função `input_register_device()`.
  - ✗ Reportar os eventos de input com as funções `input_report_key()` e `input_sync()`.
- ✗ Vários exemplos de drivers estão disponíveis em `drivers/input/`.





# REGISTRANDO DISPOSITIVO

```
#include <linux/input.h>

struct input_dev *button_dev;
[...]

button_dev = input_allocate_device();
[...]

input_set_capability(button_dev, EV_KEY, KEY_ENTER);
[...]

input_register_device(button_dev);
```





# REPORTANDO O EVENTO

```
static irqreturn_t button_interrupt(int irq, void *dummy)
{
    input_report_key(button_dev, KEY_ENTER, 1);
    input_sync(button_dev);
    return IRQ_HANDLED;
}
```





# LEDS

- ✗ Existe um framework específico para dispositivos de led.
- ✗ Habilitado na opção do kernel `CONFIG_NEW_LEDS`.
- ✗ Fontes disponíveis em `drivers/leds/` e definição da API em `<linux/leds.h>`.
- ✗ Documentação em `Documentation/leds/leds-class.txt`.





## LEDS (cont.)

- × Implementa um diretório em `/sys/class/leds/` para cada led registrado, disponibilizando alguns arquivos para manipular estes leds.

```
$ ls /sys/class/leds/imx6:blue:power
brightness
device
max_brightness
power
subsystem
trigger
uevent
```





# IMPLEMENTAÇÃO

- ✗ Tudo o que um driver de led precisa fazer é:
  - ✗ Definir e inicializar uma estrutura do tipo `led_classdev`.
  - ✗ Prover uma função de callback para mudar o status do led.
  - ✗ Registrar o dispositivo de led com a função `led_classdev_register()`.
- ✗ Vários exemplos disponíveis em `drivers/leds/`.





Embedded Labworks

# LABORATÓRIO

Integrando com o framework de led



Embedded Labworks

# Linux device drivers

Frameworks Reference



# TTY

- ✗ O que é? Subsistema para dispositivos seriais.
- ✗ Quem usa? Porta serial RS232, porta serial RS485, conversor USB/Serial, etc.
- ✗ Interface: /dev/tty\*.
- ✗ Fontes: drivers/tty/.
- ✗ Documentação: Documentation/serial/.





# INPUT

- ✗ O que é? Subsistema para dispositivos de entrada.
- ✗ Quem usa? Mouse, teclado, touchscreen, joystick, botão, etc.
- ✗ Interface: /dev/input/\*.
- ✗ Fontes: drivers/input/.
- ✗ Documentação: Documentation/input/.





# FRAMEBUFFER

- ✗ O que é? Subsistema de vídeo.
- ✗ Quem usa? Controladores de vídeo em geral para conexão com monitores e displays.
- ✗ Interface: `/dev/fb*` e `/sys/class/graphics/fb*/`.
- ✗ Fontes: `drivers/video/`.
- ✗ Documentação: `Documentation/fb/`.





# DRM

- ✗ O que é? Subsistema responsável por interfacear com GPUs (Direct Rendering Manager).
- ✗ Quem usa? Placas de vídeo e SoCs com GPU.
- ✗ Interface: /dev/dri/\*.
- ✗ Fontes: drivers/gpu/.
- ✗ Documentação: Documentation/DocBook/drm/.





# ALSA

- ✗ O que é? Subsistema de som (Advanced Linux Sound Architecture).
- ✗ Quem usa? Controladores de áudio, placas de som, etc.
- ✗ Interface: /dev/snd/\*.
- ✗ Fontes: sound/.
- ✗ Documentação: Documentation/sound/.





# V4L2

- ✗ O que é? Interface para dispositivos de captura de áudio e vídeo.
- ✗ Quem usa? Webcam, sintonizador de TV, receptor de rádio, receptor de TV digital, etc.
- ✗ Interface: /dev/dvb/\* e /dev/v4l/\*.
- ✗ Fontes: drivers/media/.
- ✗ Documentação: Documentation/video4linux/.





# BLOCK LAYER

- ✗ O que é? Subsistema para dispositivos de armazenamento com capacidade de acesso randômico.
- ✗ Quem usa? Discos rígido, CD/DVD, pendrive, etc.
- ✗ Interface: /dev/sd\*, /dev/sr\*, etc.
- ✗ Fontes: drivers/block/.
- ✗ Documentação: Documentation/block/.





# MTD

- ✗ O que é? Subsistema para memórias flash.
- ✗ Quem usa? Memórias flash NAND e NOR.
- ✗ Interface: /dev/mtd\* e /dev/mtdblock\*.
- ✗ Fontes: drivers/mtd/.
- ✗ Documentação: Documentation/mtd/.





# NETWORK

- ✗ O que é? Subsistema de rede TCP/IP.
- ✗ Quem usa? Placas de rede em geral.
- ✗ Interface: Socket.
- ✗ Fontes: net/ e drivers/net/.
- ✗ Documentação: Documentation/networking/.





# NFC

- ✗ O que é? Subsistema para dispositivos NFC.
- ✗ Quem usa? Dispositivos NFC em geral.
- ✗ Interface: Socket.
- ✗ Fontes: net/nfc/ e drivers/nfc/.
- ✗ Documentação: Documentation/nfc/.





# HWMON

- ✗ O que é? Subsistema para dispositivos de monitoramento do hardware.
- ✗ Quem usa? Sensores em geral (temperatura, corrente, tensão, etc), ventoinha, etc.
- ✗ Interface: `/sys/class/hwmon/hwmon*/.`
- ✗ Fontes: `drivers/hwmon/.`
- ✗ Documentação: `Documentation/hwmon/.`





# IIO (INDUSTRIAL I/O)

- ✗ O que é? Subsistema para dispositivos conversores analógico/digital.
- ✗ Quem usa? Conversor A/D e D/A, acelerômetro, sensor de luz, sensor de proximidade, compasso, giroscópio, magnetômetro, etc.
- ✗ Interface: /sys/bus/iio/devices/\*.
- ✗ Fontes: drivers/iio/.
- ✗ Documentação: drivers/staging/iio/Documentation/.





# WATCHDOG

- ✗ O que é? Subsistema para dispositivos watchdog.
- ✗ Quem usa? Qualquer chip com funcionalidade de watchdog.
- ✗ Interface: /dev/watchdog.
- ✗ Fontes: drivers/watchdog/.
- ✗ Documentação: Documentation/watchdog/.





# RTC

- ✗ O que é? Subsistema para relógios de tempo real.
- ✗ Quem usa? RTCs em geral.
- ✗ Interface: `/dev/rtc*` e `/sys/class/rtc/rtc*/`.
- ✗ Fontes: `drivers/rtc/`.
- ✗ Documentação: `Documentation/rtc.txt`.





# PWM

- ✗ O que é? Subsistema (recente) para dispositivos PWM.
- ✗ Quem usa? Qualquer chip com funcionalidade de PWM.
- ✗ Interface: /sys/class/pwm/\*.
- ✗ Fontes: drivers/pwm/.
- ✗ Documentação: Documentation/pwm.txt.





# PINCTRL (PIN CONTROL)

- ✗ O que é? Subsistema para gerenciar pinos de I/O.
- ✗ Quem usa? Toda e qualquer CPU ou SoC que possuir pinos de I/O para serem gerenciados, como o MUX presente nos SoCs atuais.
- ✗ Interface: -
- ✗ Fontes: `drivers/pinctrl/`.
- ✗ Documentação: `Documentation/pinctrl.txt`.





# GPIO

- ✗ O que é? Subsistema para gerenciar GPIOs.
- ✗ Quem usa? Toda e qualquer CPU ou SoC com GPIOs disponíveis, expansor de I/O, shift register, etc.
- ✗ Interface: /sys/class/gpio/\*.
- ✗ Fontes: drivers/gpio/.
- ✗ Documentação: Documentation/gpio.txt.





# LEDS

- ✗ O que é? Subsistema para gerenciar leds.
- ✗ Quem usa? Todo e qualquer led disponível no hardware.
- ✗ Interface: /sys/class/leds/\*.
- ✗ Fontes: drivers/leds/.
- ✗ Documentação: Documentation/leds/.





# PARPORT

- ✗ O que é? Subsistema para portas paralelas.
- ✗ Quem usa? Qualquer dispositivo de porta paralela.
- ✗ Interface: /dev/lp\*.
- ✗ Fontes: drivers/parport/.
- ✗ Documentação: Documentation/parport.txt.





# CLOCK

- ✗ O que é? Subsistema para gerenciar as fontes de clock do sistema.
- ✗ Quem usa? Todo e qualquer device driver que dependa de um clock para seu funcionamento (Ex: UART).
- ✗ Interface: -
- ✗ Fontes: drivers/clk/.
- ✗ Documentação: Documentation/clk.txt.





# CPUFREQ

- ✗ O que é? Subsistema para gerenciamento de energia.
- ✗ Quem usa? Sistema (CPU, SoC, etc).
- ✗ Interface: /sys/devices/system/cpu/cpu\*/cpufreq/.
- ✗ Fontes: drivers/cpufreq/ ou arch/<arch>/\*.
- ✗ Documentação: Documentation/cpu-freq/.





# POWER MANAGEMENT

- ✗ O que é? Subsistema para gerenciar o estado de gerenciamento de energia do sistema (standby, suspend-to-RAM, suspend-to-disk).
- ✗ Quem usa? Toda plataforma que deseje este tipo de suporte.
- ✗ Interface: /sys/power/.
- ✗ Fontes: drivers/base/power/.
- ✗ Documentação: Documentation/power/.





# CPU IDLE

- ✗ O que é? Subsistema para gerenciar o modo idle da CPU.
- ✗ Quem usa? Qualquer CPU que possua um ou mais modos idle.
- ✗ Interface: /sys/devices/system/cpu/cpu\*/cpuidle/.
- ✗ Fontes: drivers/cpuidle/.
- ✗ Documentação: Documentation/cpuidle/.





# POWER SUPPLY

- ✗ O que é? Subsistema para fontes de energia.
- ✗ Quem usa? Dispositivos de controle de bateria, fontes de alimentação, etc.
- ✗ Interface: `/sys/class/power_supply/*`.
- ✗ Fontes: `drivers/power/`.
- ✗ Documentação: `Documentation/power/power_supply_class.txt`.





# REGULATOR

- ✗ O que é? Subsistema para reguladores de corrente e tensão.
- ✗ Quem usa? Qualquer chip regulador de corrente e tensão.
- ✗ Interface: /sys/class/regulator/\*.
- ✗ Fontes: drivers/regulator/.
- ✗ Documentação: Documentation/power/regulator/.





# MFD

- ✗ O que é? Subsistema para chips multifuncionais.
- ✗ Quem usa? Qualquer chip com múltiplas funções.
- ✗ Interface: -
- ✗ Fontes: drivers/mfd/.
- ✗ Documentação: -





# SEM FRAMEWORK?

- ✗ São raros os casos onde um dispositivo não se encaixa em nenhum destes frameworks, mas estes casos ainda existem. Exemplos:
  - ✗ Hardwares específicos ou proprietários (fieldbus cards, industrial I/O cards, etc).
  - ✗ Displays mais simples (LCD 2x20, 7 segmentos, etc).
  - ✗ Memórias EEPROM.
  - ✗ Chips de criptografia.
- ✗ Para estes casos, temos algumas soluções:
  - ✗ Criar um char driver ou misc driver com interface no /dev.
  - ✗ Criar uma interface específica no /sys.
  - ✗ Escrever um driver em espaço de usuário.





Embedded Labworks

# Linux device drivers

Gpiolib



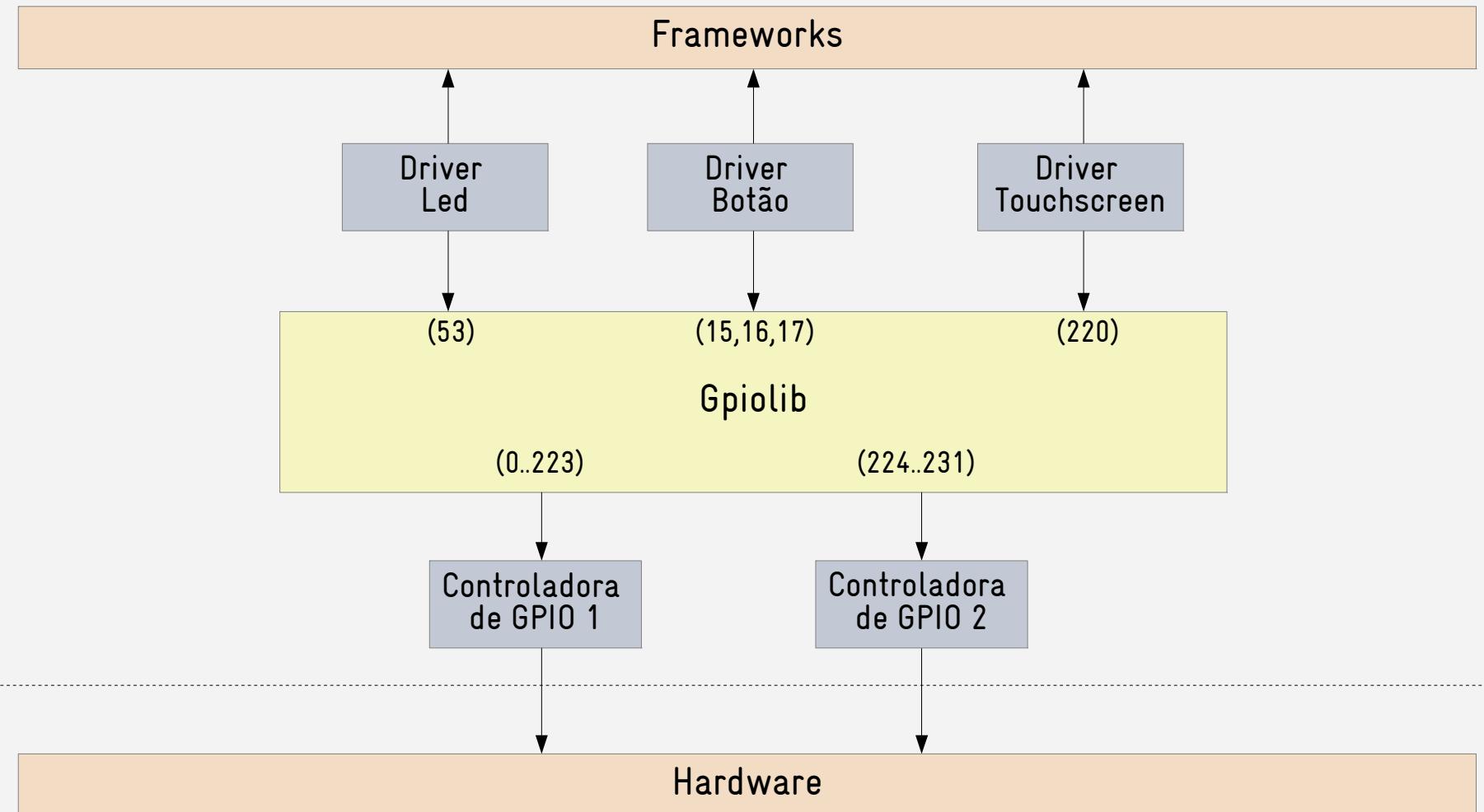
# ACESSANDO O GPIO

- ✗ Para acessar o GPIO conectado ao led da placa, o driver desenvolvido nos exercícios anteriores precisou alocar o uso de registradores da CPU que controlam 32 pinos de I/O.
- ✗ Isso pode ser um problema se algum driver precisar usar outro GPIO disponível neste mesmo registrador.
- ✗ Para estes casos, o kernel implementa uma camada adicional para gerenciar e controlar o acesso aos pinos de I/O chamada de **gpiolib**.





# GPIOLIB





## GPIOLIB (cont.)

- ✗ A definição da gpiolib está disponível em <linux/gpio.h>.
- ✗ Uma documentação completa sobre esta API está disponível nos fontes do kernel em Documentation/gpio.txt.





# USANDO A GPIOLIB

- ✗ O primeiro passo para usar a gpiolib é descobrir qual o número associado ao GPIO que deseja-se usar.
- ✗ A gpiolib atribui um número sequencial para cada GPIO registrado no sistema. Por exemplo (considerando-se que cada porta de I/O possui 32 GPIO's):
  - ✗ GPIO 1.10 = GPIO 10
  - ✗ GPIO 2.5 = GPIO 37
  - ✗ GPIO 3.1 = GPIO 65
- ✗ Com o número identificado, pode-se usar as funções da gpiolib para requisitar, configurar e usar o GPIO.





# ALGUMAS FUNÇÕES DA GPIOLIB

- ✗ `gpio_request()`: requisitar o uso de um GPIO.
- ✗ `gpio_direction_input()`: configurar o GPIO como entrada.
- ✗ `gpio_direction_output()`: configurar o GPIO como saída.
- ✗ `gpio_get_value()`: ler o GPIO.
- ✗ `gpio_set_value()`: escrever no GPIO.
- ✗ `gpio_free()`: liberar um GPIO requisitado.





Embedded Labworks

# LABORATÓRIO

Usando a camada de GPIO do kernel



Embedded Labworks

# Linux device drivers

Bus Infrastructure



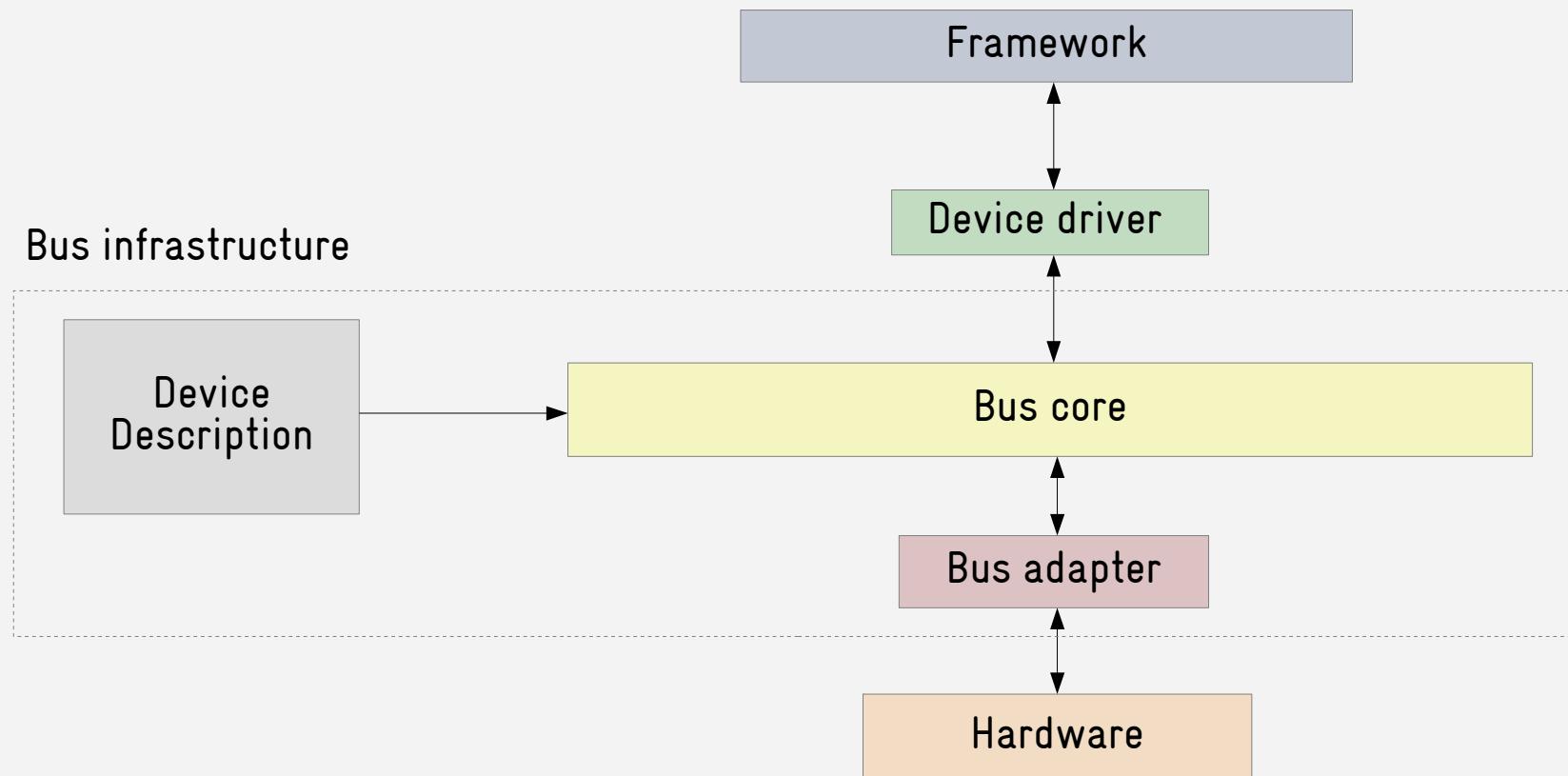
# BUS INFRASTRUCTURE

- ✗ A infraestrutura de barramento (bus infrastructure) faz parte do Driver Model do kernel Linux.
- ✗ Seus dois principais objetivos são:
  - ✗ Separar a descrição do hardware do driver de dispositivo.
  - ✗ Separar a implementação do driver do dispositivo da implementação da controladora do barramento.





# BUS INFRASTRUCTURE (cont.)





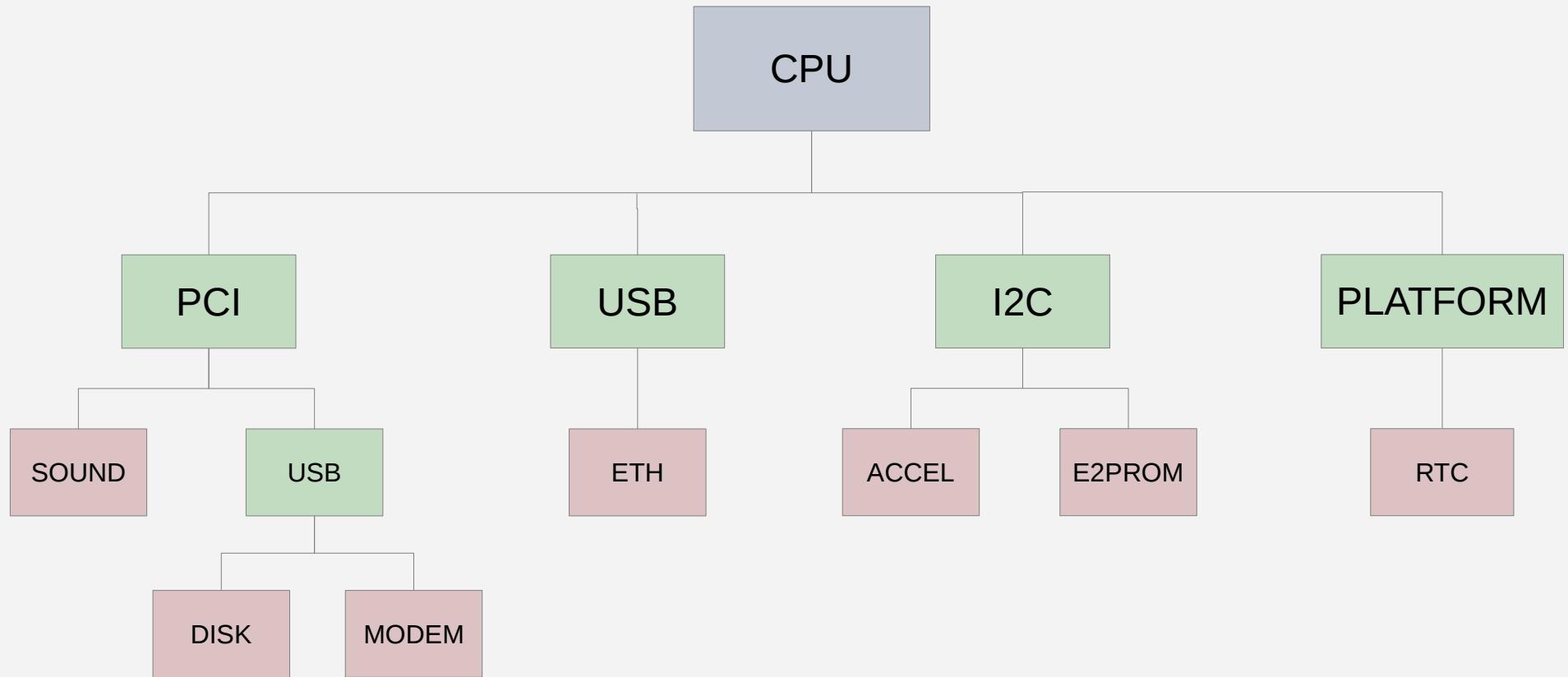
# ALGUMAS VANTAGENS

- ✗ Possibilidade de separar o código do driver da descrição do dispositivo de hardware.
- ✗ Centralizar o acesso à determinado barramento do sistema.
- ✗ Capacidade de identificar hierarquicamente todos os dispositivos e barramentos conectados ao sistema, facilitando o gerenciamento de energia.





# GERENCIAMENTO DE ENERGIA





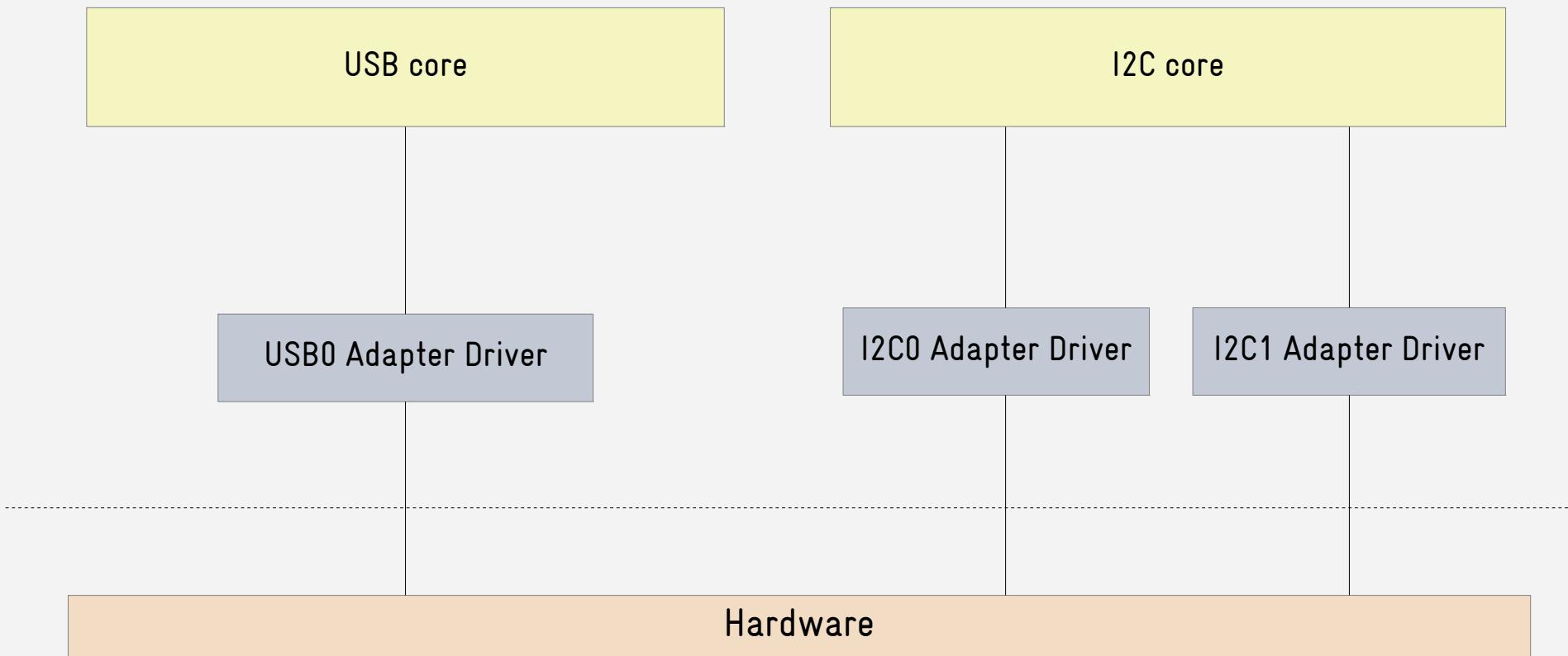
# COMPONENTES

- ✗ A infraestrutura de barramento é composta por dois componentes: bus core e bus adapter.
- ✗ O driver de barramento (bus core) implementa a API para determinado barramento. Existe um bus core para cada tipo de barramento (USB core, SPI core, I2C core, PCI core, etc).
- ✗ Um driver adaptador (bus adapter) implementa o driver da controladora de barramento. Podem existir um ou mais drivers adaptadores, um para cada controlador de barramento existente no hardware.





# COMPONENTES (cont.)





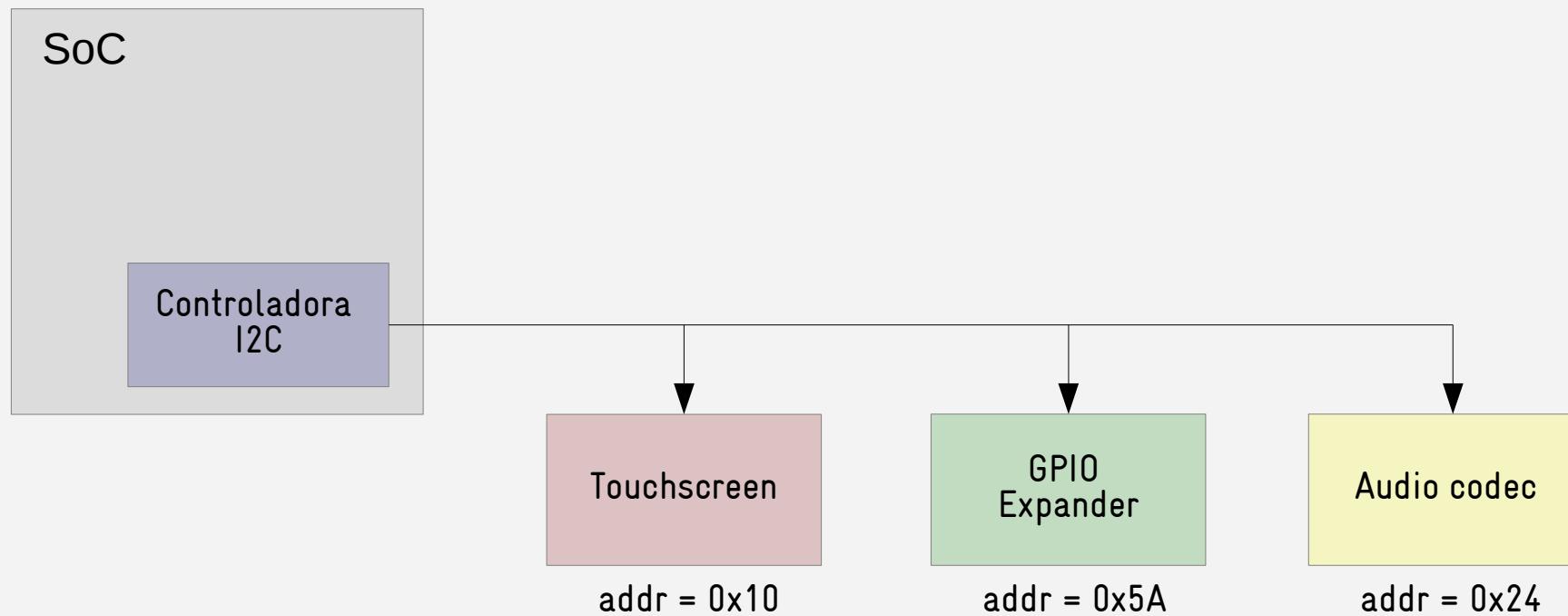
# EXEMPLO: BARRAMENTO I2C

- ✗ Barramento de baixa velocidade para conectar dispositivos próximos ao processador (na mesma placa).
- ✗ Comunicação através de dois fios: SDA (dado) e SCL (clock).
- ✗ Barramento do tipo master/slave:
  - ✗ Em um sistema Linux, o processador é normalmente o master.
  - ✗ Cada slave tem um endereço associado à ele.





# BARRAMENTO I2C





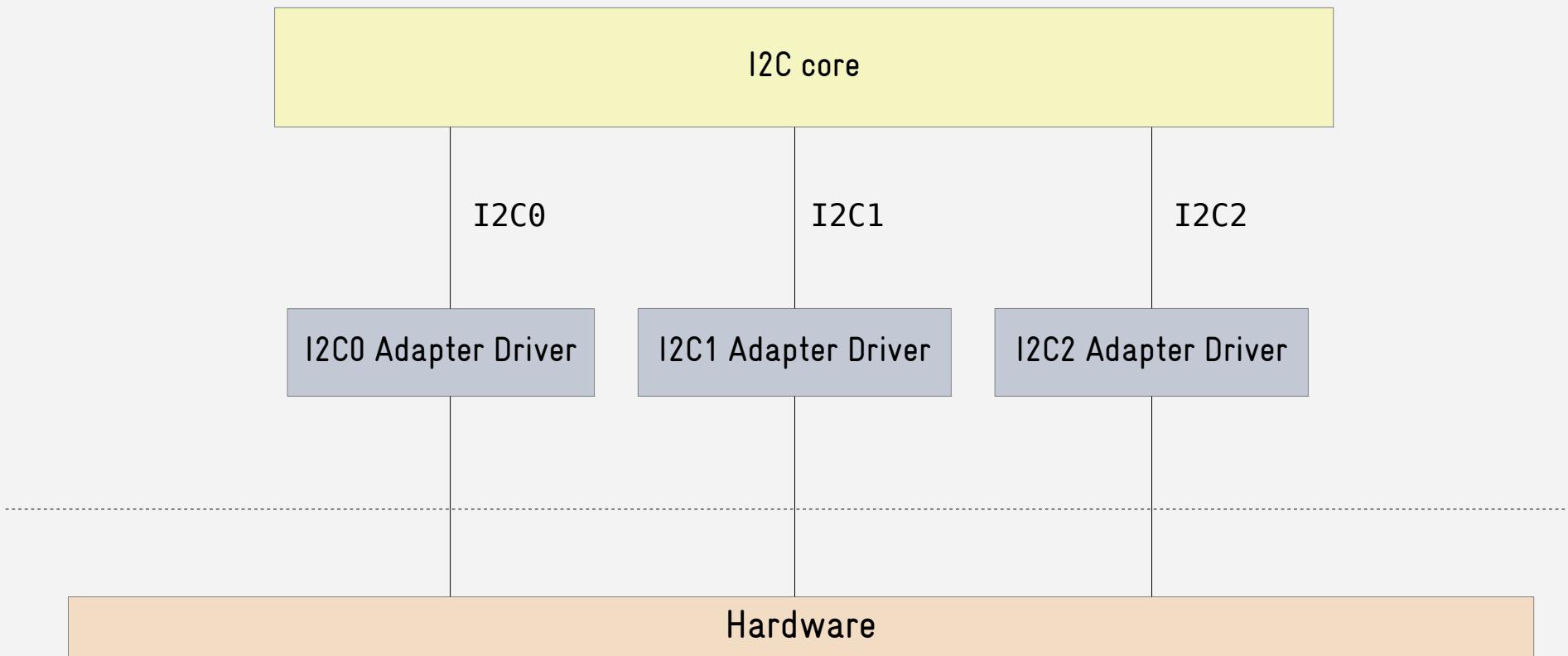
# CORE I2C

- ✗ O código fonte do core I2C está disponível em `drivers/i2c/`.
- ✗ Os drivers de controladoras de barramento I2C estão disponíveis em `drivers/i2c/busses/`.
- ✗ Durante o boot do sistema, os drivers adaptadores das controladoras I2C se registram no core I2C.



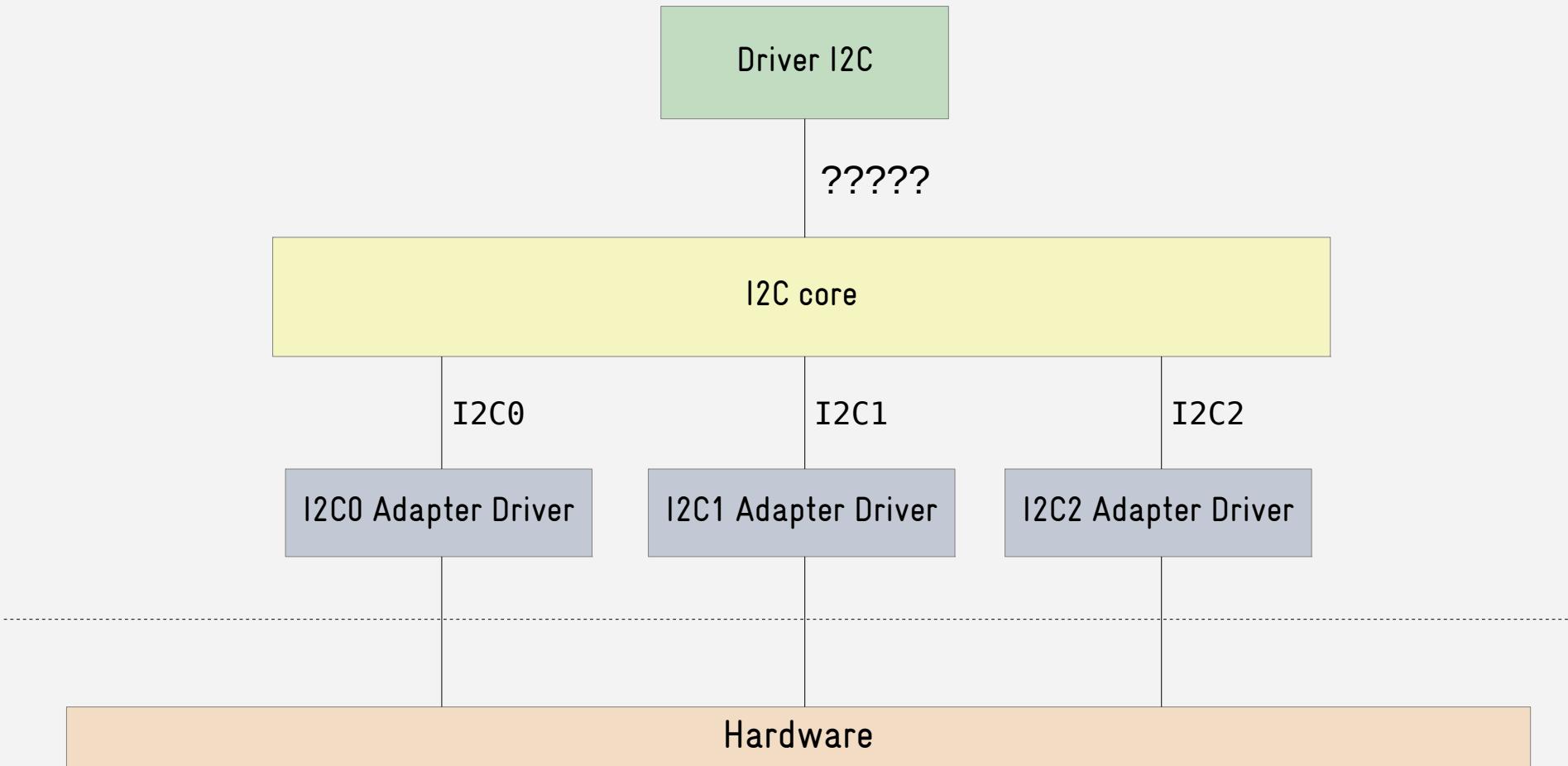


# REGISTRANDO OS BARRAMENTOS





# E0 DRIVER?





# IMPLEMENTANDO O DRIVER

- ✗ Para implementar um driver conectado ao barramento I2C, é necessário:
  - ✗ Definir uma estrutura do tipo `i2c_driver` com informações gerais do driver.
  - ✗ Definir uma estrutura do tipo `i2c_device_id` com a lista de dispositivos tratados por este driver.
  - ✗ Registrar o driver i2c com a função `i2c_add_driver()` na inicialização do driver.





# ESTRUTURA I2C

```
static const struct i2c_device_id mma8450_id[] = {
    { "mma8450", 0 },
    {},
};

static struct i2c_driver mma8450_driver = {
    .driver = {
        .name    = "mma8450",
        .owner   = THIS_MODULE,
    },
    .probe     = mma8450_probe,
    .remove   = mma8450_remove,
};
```





# REGISTRANDO O DRIVER

```
static int __init mma8450_init(void)
{
    i2c_add_driver(&mma8450_driver);
}

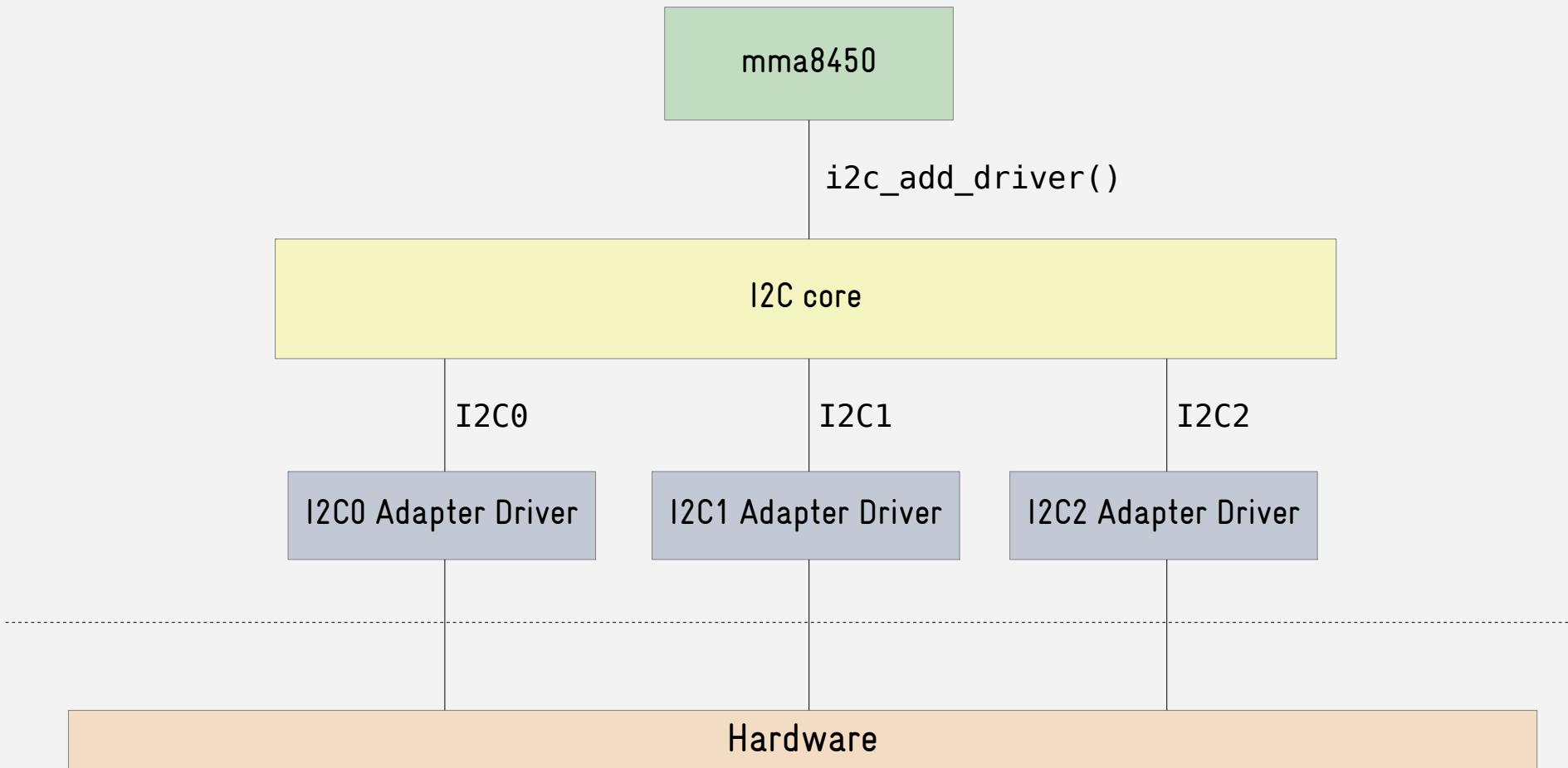
static void __exit mma8450_exit(void)
{
    i2c_del_driver(&mma8450_driver);
}

module_init(mma8450_init);
module_exit(mma8450_exit);
```





# REGISTRANDO O DRIVER





# REGISTRANDO O DISPOSITIVO

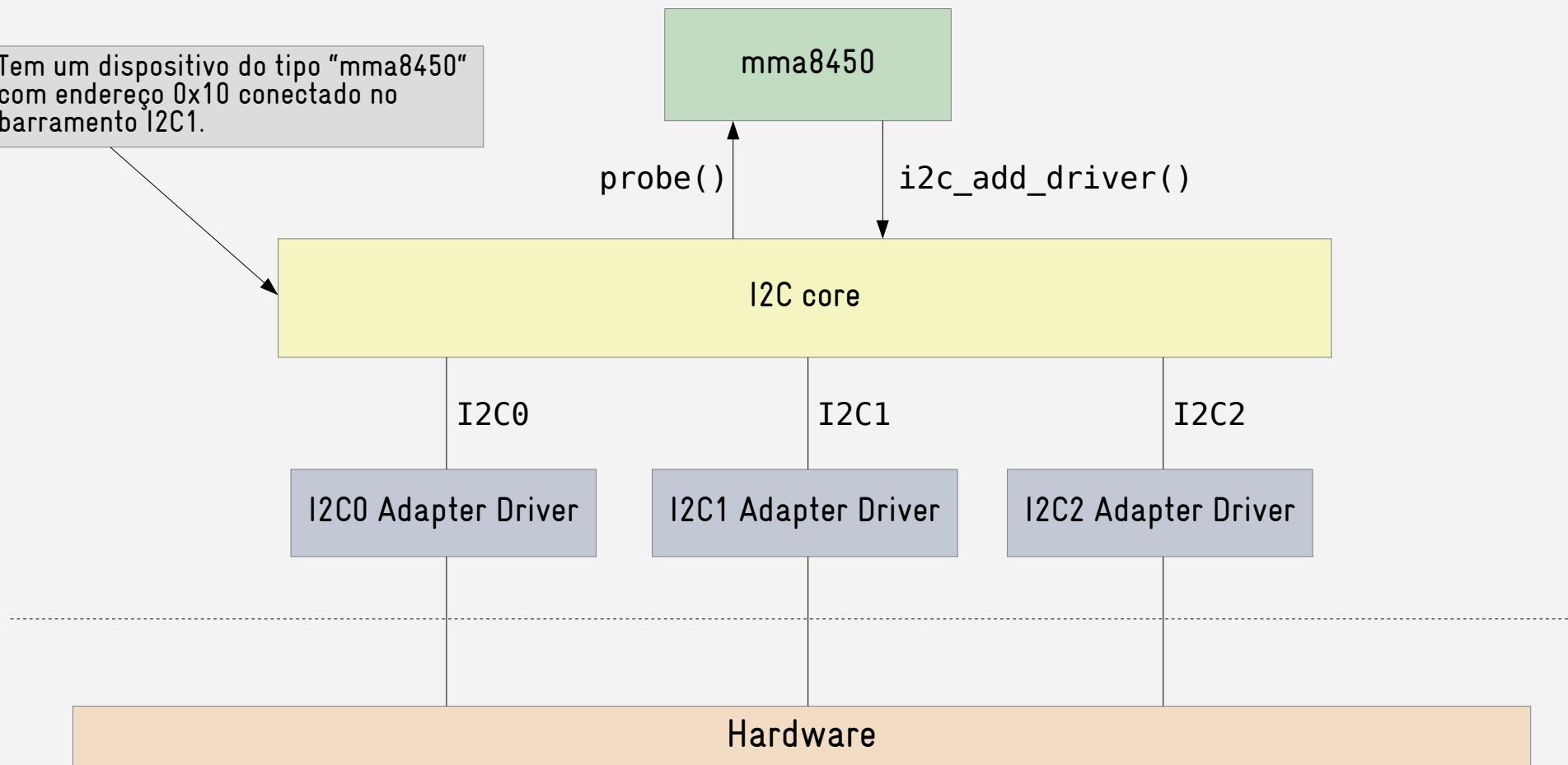
- ✗ A partir deste momento, o barramento I2C sabe que existe um driver para tratar dispositivos do tipo mma8450.
- ✗ Mas ele não sabe (ainda) que existe um dispositivo mma8450 conectado ao barramento.
- ✗ Para isso, você deve **register o dispositivo no barramento**.
- ✗ Quando você registrar o dispositivo no barramento, o kernel irá instanciar o driver chamando sua função `probe()`.





# PROBE

Tem um dispositivo do tipo "mma8450" com endereço 0x10 conectado no barramento I2C1.





# O MÉTODO PROBE

- ✗ O método probe() é responsável por:
  - ✗ Inicializar o hardware, mapear I/O em memória e registrar ISRs.
  - ✗ Registrar o dispositivo no framework correto do kernel.
- ✗ O método probe() recebe como argumento uma estrutura descrevendo o dispositivo de acordo com o barramento (i2c\_client, pci\_dev, usb\_interface, etc).
- ✗ O driver deve utilizar esta estrutura para se comunicar com o barramento.





# MMA8450 PROBE

```
static int mma8450_probe(struct i2c_client *c,
                         const struct i2c_device_id *id)
{
    [...]
    client_id = i2c_smbus_read_byte_data(c, MMA8450_WHO_AM_I);

    [...]
    idev = input_allocate_polled_device();

    [...]
    err = input_register_polled_device(idev);
    [...]
}
```





# I2C API

```
#include <linux/i2c.h>

s32 i2c_smbus_read_byte(const struct i2c_client *client);

s32 i2c_smbus_write_byte(const struct i2c_client *client,
                         U8 value);

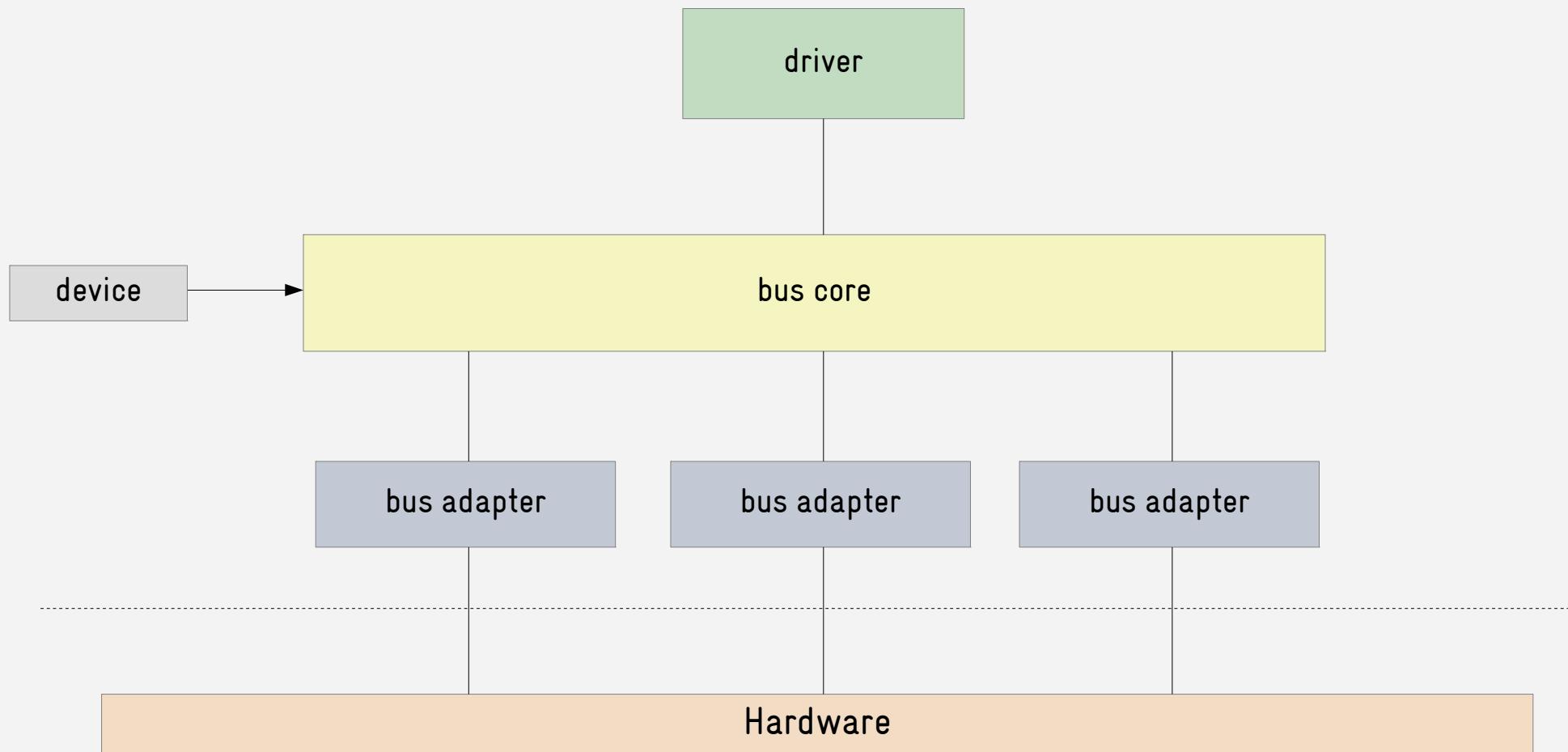
int i2c_master_send(struct i2c_client *client,
                     const char *buf, int count);

int i2c_master_recv(struct i2c_client *client,
                     char *buf, int count);
```





# RESUMO DRIVER MODEL





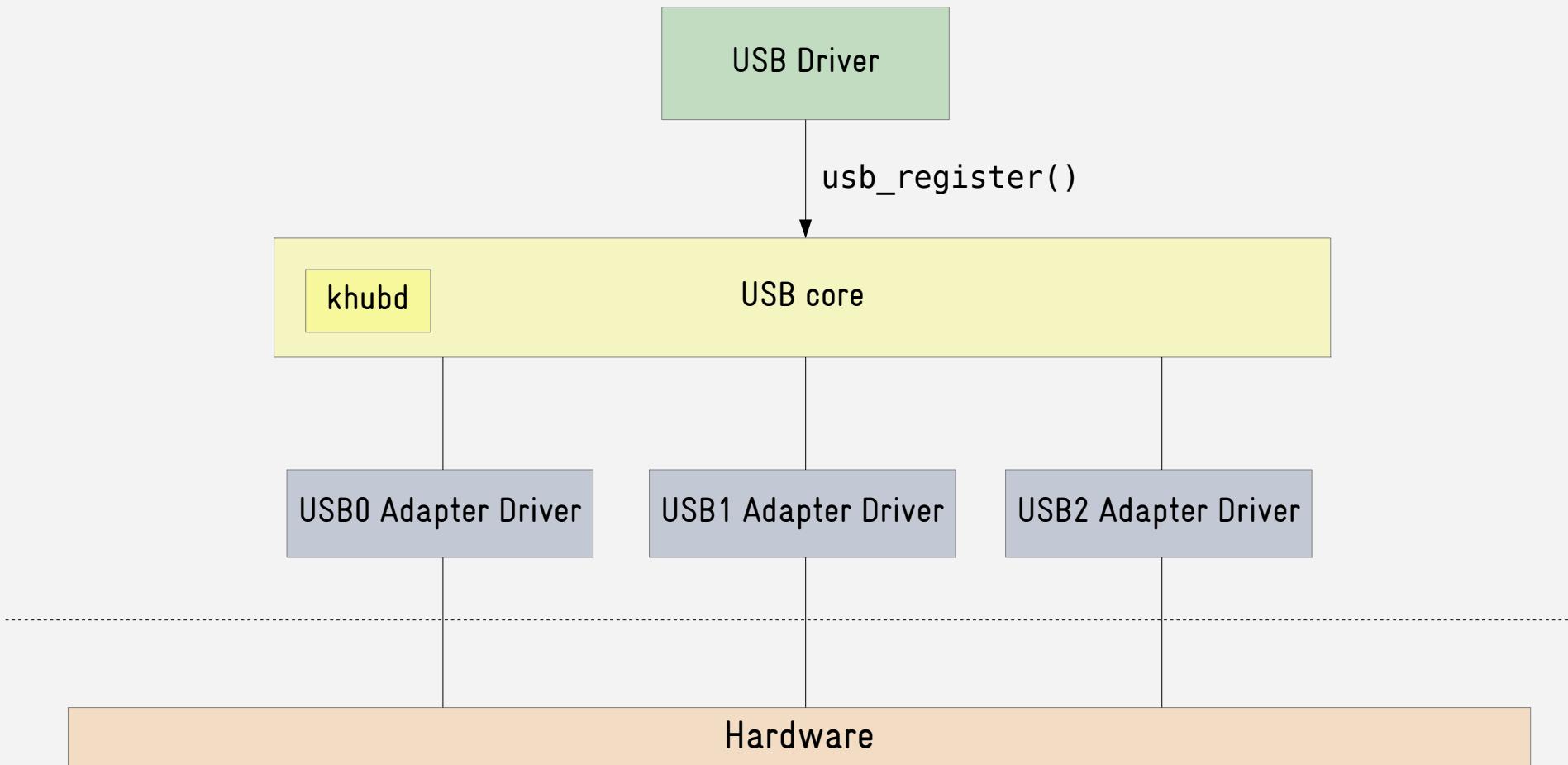
# REGISTRANDO O DISPOSITIVO (cont.)

- ✗ Como registrar um dispositivo?
  - ✗ Um dispositivo pode ser registrado dinamicamente pelo barramento se este possuir suporte à enumeração de dispositivos (ex: USB, PCI, etc).
  - ✗ Um dispositivo pode ser registrado estaticamente através de uma função disponibilizada pela infraestrutura do barramento. Ex: `i2c_register_board_info()`.
  - ✗ Usando o mecanismo de device tree, disponível em algumas arquiteturas, incluindo ARM (mecanismo padrão nas versões mais atuais do kernel).





# REGISTRANDO DINAMICAMENTE





# REGISTRANDO ESTATICAMENTE

```
/* arch/arm/mach-mx5/board-mx53_loco.c */

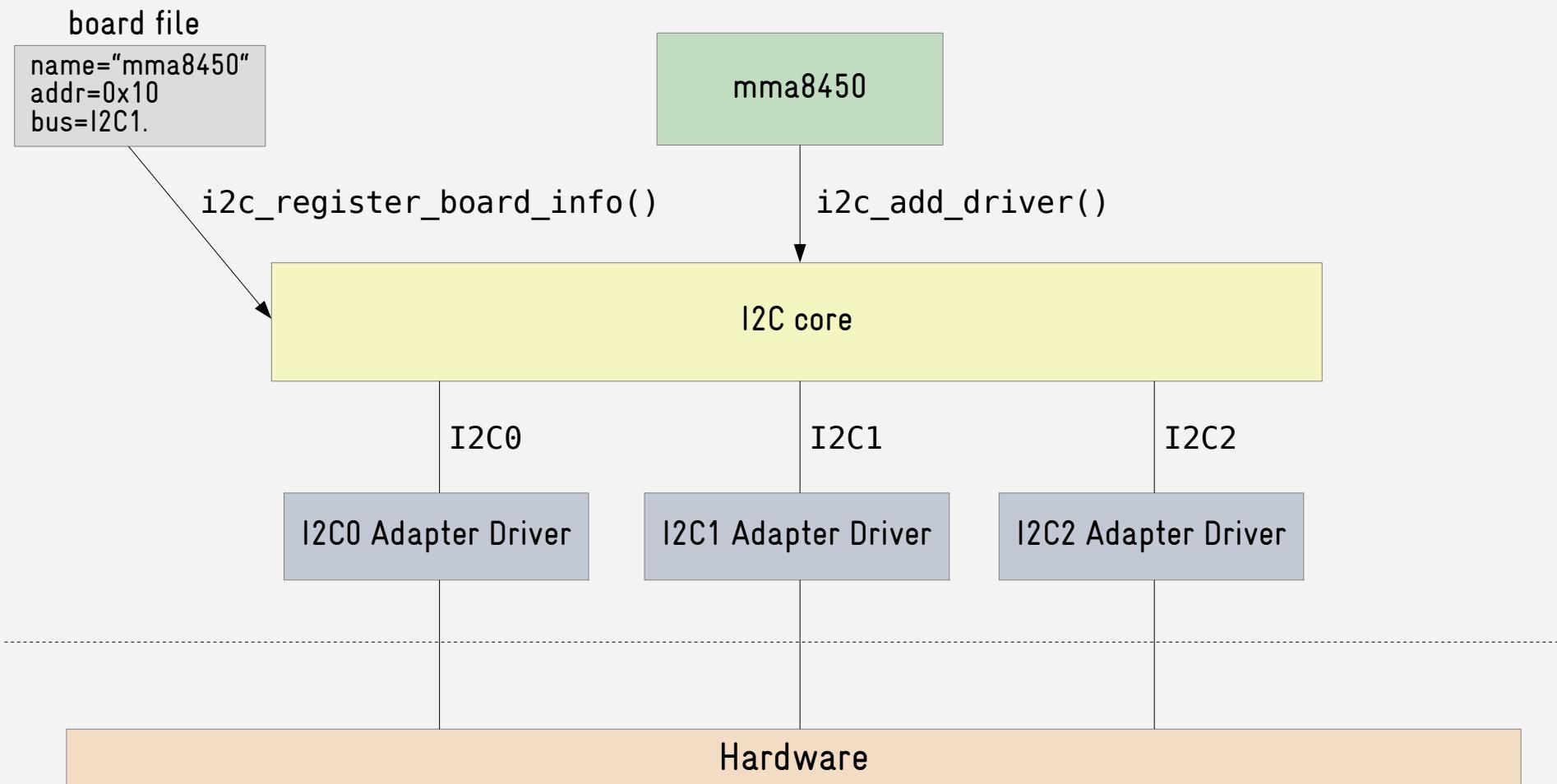
static struct i2c_board_info mxc_i2c1_board_info[] __initdata = {
    {
        .type = "mma8450",
        .addr = 0x10,
    },
};

static void __init mx53_loco_board_init(void)
{
    [...]
    i2c_register_board_info(1, mxc_i2c1_board_info,
                           ARRAY_SIZE(mxc_i2c1_board_info));
    [...]
}
```



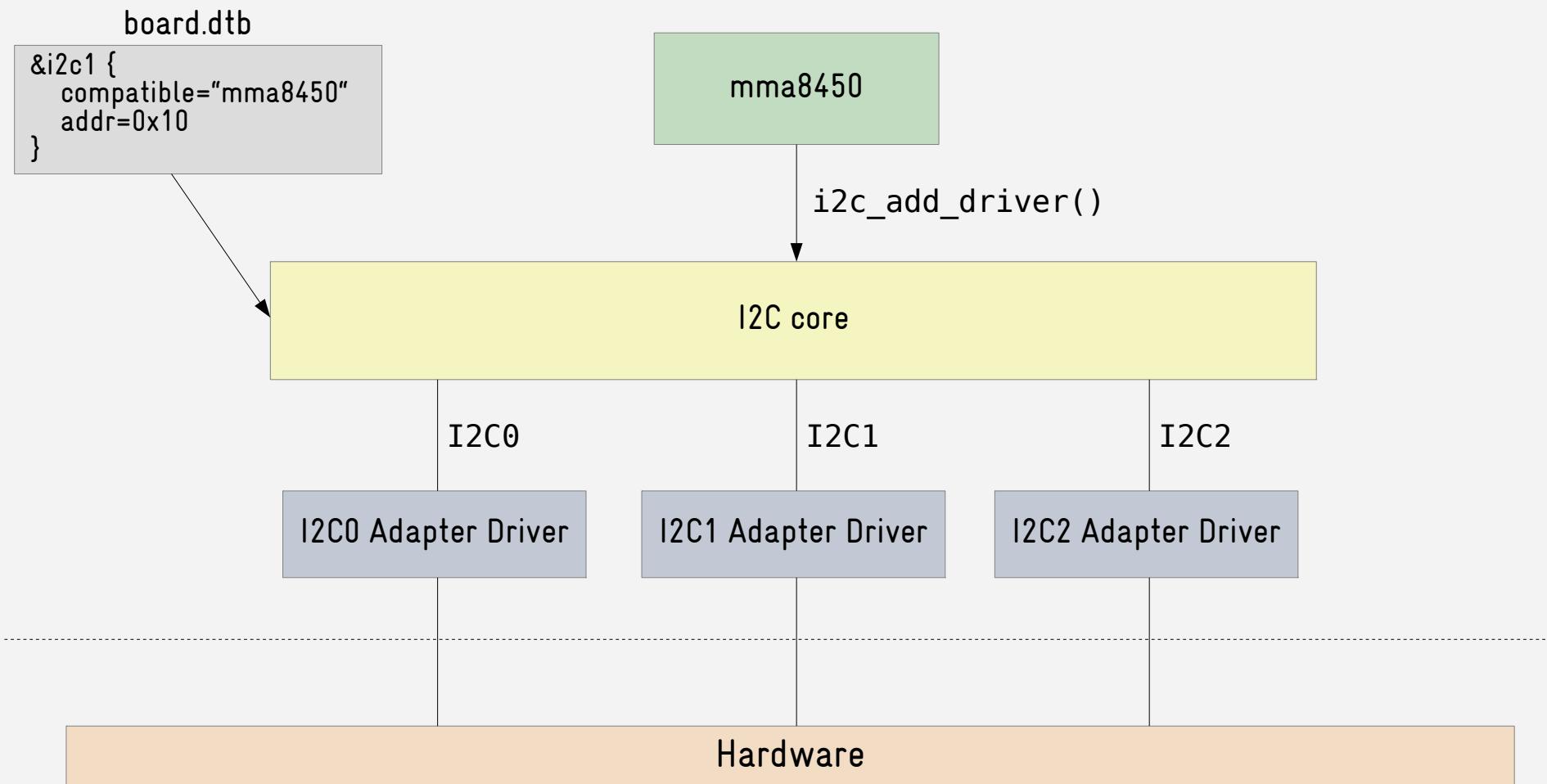


# REGISTRANDO ESTATICAMENTE (cont.)





# REGISTRANDO VIA DEVICE TREE





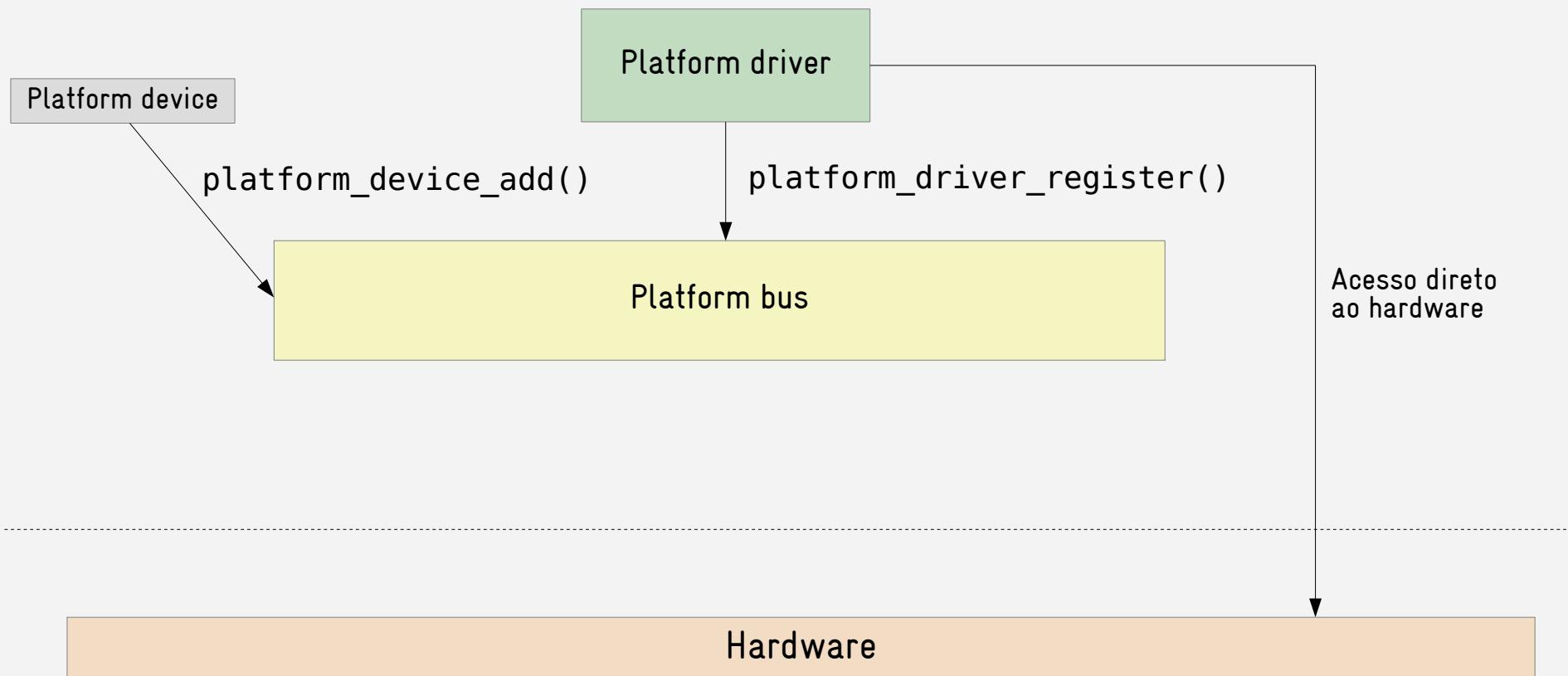
# PLATFORM BUS

- ✗ Alguns dispositivos podem não estar conectados em um barramento, como por exemplo uma UART ou algum dispositivo conectado a um GPIO.
- ✗ E como prover a mesma solução de infraestrutura de barramento sem um barramento físico?
- ✗ Através de um barramento de plataforma!





# PLATFORM BUS (cont.)





# IMPLEMENTANDO PLATFORM DRIVERS

- ✗ Um dispositivo de hardware é representado por um platform device e o driver para tratar este dispositivo é representado por um platform driver.
- ✗ Para implementar um platform driver, é necessário:
  - ✗ Definir uma estrutura do tipo `platform_driver`.
  - ✗ Implementar uma estrutura do tipo `platform_device_id`.
  - ✗ Registrar o platform driver com a função `platform_driver_register()` na inicialização do driver.





# EXEMPLO PORTA SERIAL

```
static struct platform_device_id imx_uart_devtype[] = {
    { .name = "imx1-uart",
        .driver_data = (kernel_ulong_t) &imx_uart_devdata[IMX1_UART],
    }, {
        .name = "imx21-uart",
        .driver_data = (kernel_ulong_t) &imx_uart_devdata[IMX21_UART],
    },
    {},
};

static struct platform_driver serial_imx_driver = {
    .probe      = serial_imx_probe,
    .remove     = serial_imx_remove,
    .id_table   = imx_uart_devtype,
    .driver     = {
        .name      = "imx-uart",
        .owner     = THIS_MODULE,
    },
};
```





# EXEMPLO PORTA SERIAL (cont.)

```
static int __init imx_serial_init(void)
{
    platform_driver_register(&serial_imx_driver);
}

static void __exit imx_serial_exit(void)
{
    platform_driver_unregister(&serial_imx_driver);
}
```





# REGISTRANDO UM PLATFORM DEVICE

```
static struct platform_device imx_uart1_device = {
    .name = "imx1-uart",
    .id = 0,
    .num_resources = ARRAY_SIZE(imx_uart1_resources),
    .resource = imx_uart1_resources,
    .dev = {
        .platform_data = &mxc_ports[0],
    }
};

static void __init mx53_loco_board_init(void)
{
    [...]
    platform_device_add(&imx_uart1_device);
    [...]
}
```





# USANDO RECURSOS

- ✗ Todo driver normalmente usa um ou mais recursos de hardware, como portas de I/O, linhas de interrupção ou canais de DMA.
- ✗ Estas informações podem ser representadas em uma estrutura do tipo struct resource, e um vetor de estruturas deste tipo estão associadas à um platform device.
- ✗ Este tipo de mecanismo permite que determinado driver possa ser instanciado para gerenciar múltiplos dispositivos, que usam recursos de hardware diferentes, sem que uma linha de código seja alterada.





# USANDO RECURSOS (cont.)

```
static struct platform_device imx_uart1_device = {  
    .name = "imx1-uart",  
    .id = 0,  
    .num_resources = ARRAY_SIZE(imx_uart1_resources),  
    .resource = imx_uart1_resources,  
    .dev = {  
        .platform_data = &mxc_ports[0],  
    }  
};
```

Definição dos recursos  
usados pelo dispositivo  
de hardware





# USANDO RECURSOS (cont.)

```
static struct resource mxc_uart_resources1[] = {
{
    .start = UART1_BASE_ADDR,
    .end = UART1_BASE_ADDR + 0x0B8,
    .flags = IORESOURCE_MEM,
},
{
    .start = MXC_INT_UART1,
    .flags = IORESOURCE_IRQ,
},
};
```





# USANDO RECURSOS (cont.)

```
static int serial_imx_probe(struct platform_device *pdev)
{
    struct resource *res;

    [...]

    /* get and map memory mapped I/O */
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    base = ioremap(res->start, PAGE_SIZE);

    [...]

    /* get ISR number */
    sport->rxirq = platform_get_irq(pdev, 0);

    [...]
}
```





# PLATFORM DATA

- ✗ Além dos recursos de hardware comuns que podem ser alocados para um determinado dispositivo, muitos dispositivos requerem informações específicas que não estão disponíveis em outros dispositivos de hardware (ex: modos de vídeo de um display).
- ✗ Estas informações podem ser passadas para um driver em uma outra estrutura chamada de platform\_data.
- ✗ Esta estrutura nada mais é do que um ponteiro para void dentro da estrutura platform\_device.





# USANDO RECURSOS (cont.)

```
static struct platform_device imx_uart1_device = {  
    .name = "imx1-uart",  
    .id = 0,  
    .num_resources = ARRAY_SIZE(imx_uart1_resources),  
    .resource = imx_uart1_resources,  
    .dev = {  
        .platform_data = &mxc_ports[0],  
    }  
};
```

Definição de informações  
específicas do dispositivo  
de hardware





# PLATFORM DATA (cont.)

```
static uart_mxc_port mxc_ports[] = {
    [0] = {
        [...]
        .ints_muxed = 1,
        .mode = MODE_DCE,
        .ir_mode = NO_IRDA,
        .enabled = 1,
        .cts_threshold = UART1_UCR4_CTSTL,
        .dma_enabled = UART1_DMA_ENABLE,
        .dma_rdbuf_size = UART1_DMA_RXBUFSIZE,
        .rx_threshold = UART1_UFCR_RXTL,
        .tx_threshold = UART1_UFCR_TXTL,
    [...]
}
```





# PLATFORM DATA (cont.)

```
static int serial_imx_probe(struct platform_device *pdev)
{
    [...]
    mxc_ports[id] = pdev->dev.platform_data;
    [...]

    if (mxc_ports[id]->enabled == 1) {
        [...]
    }
}
```





Embedded Labworks

# LABORATÓRIO

Integração com o driver model do kernel



Embedded Labworks

# Linux device drivers

Device Tree



# REGISTRANDO DISPOSITIVOS NO CÓDIGO

- ✗ Registrar os dispositivos de hardware no código traz muitas desvantagens, dentre elas:
  - ✗ Muito código duplicado (diferentes implementações para diferentes plataformas/placas).
  - ✗ Qualquer alteração de hardware (ex: adicionar um novo dispositivo SPI ou alterar o endereço de um dispositivo I2C) requer alteração do código e recompilação do kernel.
  - ✗ Fica difícil liberar uma mesma imagem do kernel que rode em diferentes plataformas de hardware.





# SOLUÇÃO

- ✗ Precisamos então de um mecanismo para identificar a topologia e a configuração de hardware (CPU, memória e dispositivos de I/O) em tempo de execução.
- ✗ A solução completa para este problema em plataformas ARM veio na versão 3.7 do kernel, com a funcionalidade de device tree.





# DEVICE TREE

- ✗ Em linhas gerais, o device tree é uma estrutura de dados capaz de descrever a topologia e a configuração do hardware presente no sistema.
- ✗ Foi criado originalmente pelo projeto **Open Firmware**, um padrão de firmware de boot desenvolvido inicialmente pela Sun.
- ✗ Como o Open Firmware é bastante usado por plataformas PowerPC e SPARC, o suporte à device tree no Linux para estas plataformas é bem antigo.





## DEVICE TREE (cont.)

- ✗ Em 2005, no trabalho de integração do PPC 32 bits e 64 bits, tornou-se obrigatório o uso do DT em todas plataformas PPC.
- ✗ Para isso, criou-se uma variação do device tree chamada **Flattened Device Tree (FDT)**, que transforma o device tree, originalmente um arquivo texto, em um arquivo binário chamado DTB (Device Tree Binary ou Device Tree Blob).
- ✗ Atualmente o device tree é suportado diversas arquiteturas, incluindo ARM, PowerPC, OpenRISC, ARC e Microblaze.





# DEVICE TREE (cont.)

- ✗ Com o device tree é possível descrever:
  - ✗ O tipo e a quantidade de CPUs presentes no sistema.
  - ✗ O endereço base e o tamanho da memória RAM.
  - ✗ Os barramentos presentes no sistema.
  - ✗ Os dispositivos conectados aos barramentos, além de informações específicas de cada dispositivo como linhas de interrupção, endereços de I/O, etc.
- ✗ O kernel Linux é capaz de usar o device tree para identificar em tempo de execução a topologia e a configuração de hardware da placa (CPU, memória e I/O), possibilitando inicializar e configurar o sistema de forma dinâmica.





# NA PRÁTICA

- × Cada placa possui um arquivo de especificação do device tree, disponível em arch/arm/boot/dts:

```
$ ls arch/arm/boot/dts/
```

```
aks-cdu.dts
am335x-bone.dts
am335x-evm.dts
am335x-evmsk.dts
am33xx.dtsi
am3517-evm.dts
am3517_mt_ventoux.dts
animeo_ip.dts
armada-370-db.dts
[...]
```

```
at91sam9x35ek.dts
at91sam9x5cm.dtsi
at91sam9x5.dtsi
at91sam9x5ek.dtsi
bcm11351-brt.dts
bcm11351.dtsi
bcm2835.dtsi
bcm2835-rpi-b.dts
ccu9540.dts
```





# GERANDO O DTB

- × Uma ferramenta chamada Device Tree Compiler (dtc), disponível em `scripts/dtc/`, é responsável por compilar o device tree.

- × Para compilar o device tree de todas as placas habilitadas no arquivo de configuração do kernel:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- dtbs
```

- × Para compilar o device tree de uma placa específica:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- imx6dl-colibri-ipe.dtb
```

- × Os arquivos DTB são gerados em `arch/arm/boot/dts`.





# PASSANDO O DTB PARA O KERNEL

- × Durante o processo de boot, o bootloader é o responsável por passar o DTB para o kernel.
- × Caso o bootloader não seja capaz de passar o DTB para o kernel, é possível indicar ao kernel que o DTB está integrado à sua imagem habilitando a opção `CONFIG_ARM_APPENDED_DTB`, e depois gerando uma imagem com o DTB integrado:

```
$ cat zImage board.dtb > zImage_dtb
```





# SINTAXE DO DEVICE TREE

```
/ {  
    node@0 {  
        string-property = "string";  
        string-list-property = "first string", "second string";  
        byte-string-property = [0x01 0x21 0x76];  
  
        child-node@0 {  
            first-child-property;  
            second-child-property = <1>;  
            reference-to-something = <&node1>;  
        };  
  
        child-node@1 {  
        };  
    };  
  
    node1: node@1 {  
        empty-property;  
        cell-property = <1 2 3 4>;  
    };  
};
```





# DESCREVENDO O HARDWARE NO DT

- ✗ Começando no nó-raiz do device tree, temos os seguintes sub-nós:
  - ✗ model: informação descritiva sobre o modelo do hardware.
  - ✗ compatible: string utilizada para o kernel identificar o hardware e instanciar o código de inicialização correspondente.
  - ✗ cpus: descreve as CPUs presentes no sistema.
  - ✗ memory: descreve a quantidade e o endereço inicial da memória RAM.
  - ✗ chosen: usado para passar a linha de comandos do kernel.
  - ✗ aliases: permite definir apelidos para determinados nós.
  - ✗ Nós adicionais descrevendo os barramentos e dispositivos do hardware.





# EXEMPLO DT

```
/ {  
    model = "Embedded Labworks Super Board";  
    compatible = "labworks,imx6dl-labworks", "fsl,imx6dl";  
  
    memory {  
        reg = <0x10000000 0x80000000>;  
    };  
  
    cpus {  
        cpu@1 {  
            compatible = "arm,cortex-a9";  
            device_type = "cpu";  
            reg = <1>;  
        };  
    };  
  
    aliases {  
        portA = &gpio1;  
        portB = &gpio2;  
    };  
  
    [...]
```





# DT SERIAL IMX

```
uart1: serial@02020000 {  
    compatible = "fsl,imx6q-uart", "fsl,imx21-uart";  
    reg = <0x02020000 0x4000>;  
    interrupts = <0 26 0x04>;  
    clocks = <&clks 160>, <&clks 161>;  
    clock-names = "ipg", "per";  
    dmas = <&sdma 25 4 0>, <&sdma 26 4 0>;  
    dma-names = "rx", "tx";  
    status = "disabled";  
};
```





# DRIVER SERIAL IMX

```
static struct of_device_id imx_uart_dt_ids[] = {
    { .compatible = "fsl,imx6q-uart", .data = &imx_uart_devdata[IMX6Q_UART], },
    { .compatible = "fsl,imx1-uart", .data = &imx_uart_devdata[IMX1_UART], },
    { .compatible = "fsl,imx21-uart", .data = &imx_uart_devdata[IMX21_UART], },
    { /* sentinel */ }
};

static struct platform_driver serial_imx_driver = {
    .probe          = serial_imx_probe,
    .remove         = serial_imx_remove,
    .suspend        = serial_imx_suspend,
    .resume         = serial_imx_resume,
    .id_table       = imx_uart_devtype,
    .driver         = {
        .name   = "imx-uart",
        .owner  = THIS_MODULE,
        .of_match_table = imx_uart_dt_ids,
    },
};
```





# DRIVER SERIAL IMX (cont.)

```
static int serial_imx_probe(struct platform_device *pdev)
{
    [...]

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    [...]

    base = devm_ioremap(&pdev->dev, res->start, PAGE_SIZE);
    [...]

    sport->port.irq = platform_get_irq(pdev, 0);
    [...]

    sport->clk_ipg = devm_clk_get(&pdev->dev, "ipg");
    [...]
}
```





# DRIVER SERIAL IMX (cont.)

```
static int serial_imx_probe_dt(struct imx_port *sport,
                                struct platform_device *pdev)
{
    struct device_node *np = pdev->dev.of_node;
    const struct of_device_id *of_id =
        of_match_device(imx_uart_dt_ids, &pdev->dev);

    if (of_get_property(np, "fsl,uart-has-rtscts", NULL))
        sport->have_rtscts = 1;
    [...]

    if (of_get_property(np, "fsl,irda-mode", NULL))
        sport->use_irda = 1;
    [...]
}
```





# DT SGTL5000

```
&i2c2 {  
    clock-frequency = <100000>;  
    pinctrl-names = "default";  
    pinctrl-0 = <&pinctrl_i2c2_2>;  
    status = "okay";  
  
    codec: sgtl5000@0a {  
        compatible = "fsl,sgtl5000";  
        reg = <0x0a>;  
        clocks = <&clks 201>;  
        VDDA-supply = <&reg_2p5v>;  
        VDDI0-supply = <&reg_3p3v>;  
    };  
};
```





# DRIVER SGTL5000

```
static const struct of_device_id sgtl5000_dt_ids[] = {
    { .compatible = "fsl,sgtl5000", },
    { /* sentinel */ }
};

static struct i2c_driver sgtl5000_i2c_driver = {
    .driver = {
        .name = "sgtl5000",
        .owner = THIS_MODULE,
        .of_match_table = sgtl5000_dt_ids,
    },
    .probe = sgtl5000_i2c_probe,
    .remove = sgtl5000_i2c_remove,
    [...]
};
```





# DRIVER SGTL5000 (cont.)

```
static int sgtl5000_i2c_probe(struct i2c_client *client,
                           const struct i2c_device_id *id)
{
    [...]

    sgtl5000->regmap =
        devm_regmap_init_i2c(client, &sgtl5000_regmap);
    [...]

    sgtl5000->mclk = devm_clk_get(&client->dev, NULL);
    [...]

    ret = regmap_read(sgtl5000->regmap, SGTL5000_CHIP_ID, &reg);
    [...]
}
```





# ORGANIZAÇÃO DO DEVICE TREE

- ✗ O device tree é dividido em vários arquivos para tornar seu código mais modular e reutilizável.
- ✗ O código-fonte final do device tree tem extensão .dts (cada placa tem o seu arquivo .dts).
- ✗ O código comum do device tree que pode ser compartilhado com outras placas é implementado em arquivos com extensão .dtsi.
- ✗ Estes arquivos são incluídos via diretiva #include e processados através de um mecanismo de overlay.





# imx6q-wandboard.dts

```
#include "imx6q.dtsi"
#include "imx6qdl-wandboard.dtsi"

/ {
    model = "Wandboard i.MX6 Quad Board";
    compatible = "wand,imx6q-wandboard", "fsl,imx6q";

    memory {
        reg = <0x10000000 0x80000000>;
    };
};

&sata {
    status = "okay";
};
```





# imx6q.dtsi

[...]

```
sata: sata@02200000 {
    compatible = "fsl,imx6q-ahci";
    reg = <0x02200000 0x4000>;
    interrupts = <0 39 0x04>;
    clocks = <&clks 154>, <&clks 187>, <&clks 105>;
    clock-names = "sata", "sata_ref", "ahb";
    status = "disabled";
};
```

[...]





# imx6q-wandboard.dts + imx6q.dtsci

[...]

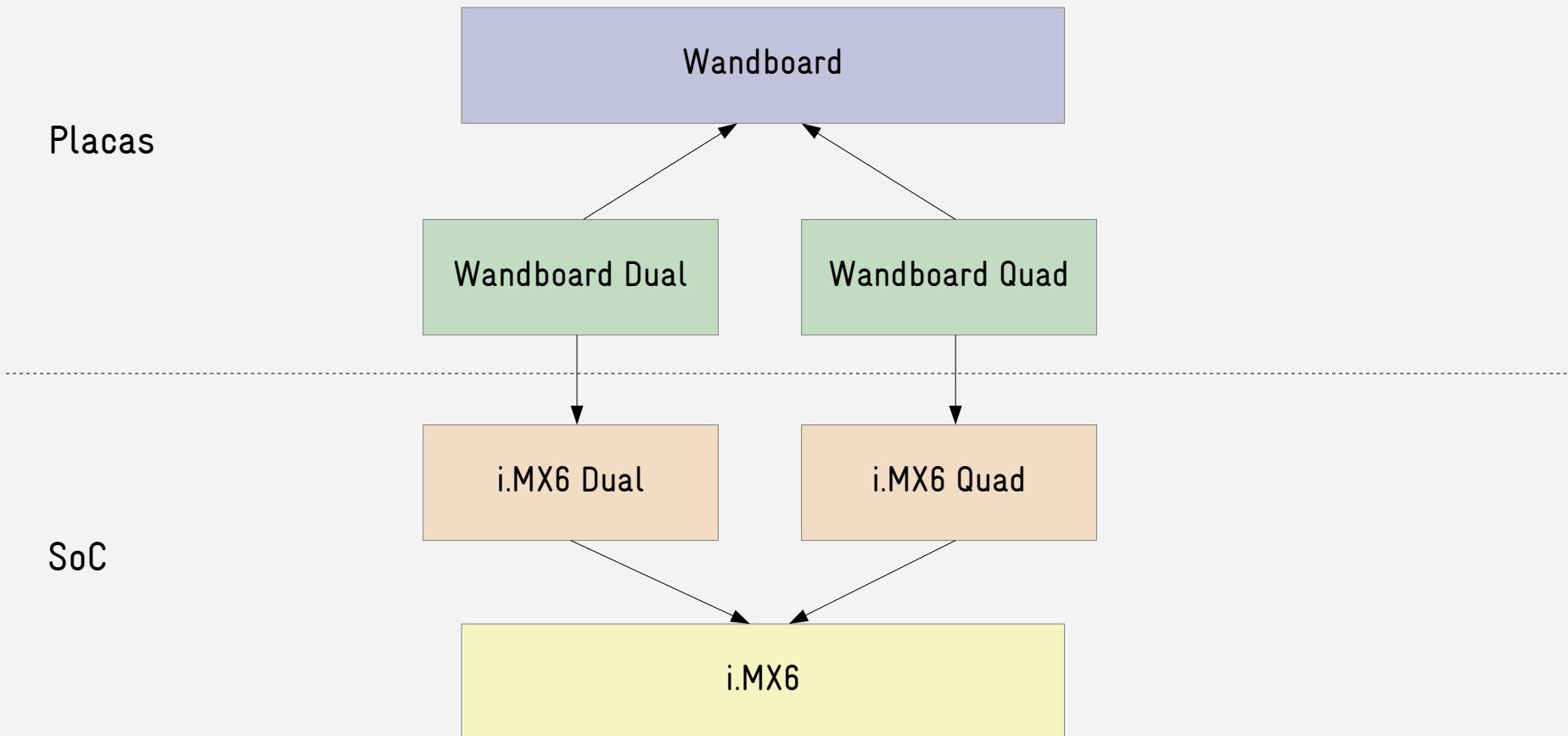
```
sata: sata@02200000 {
    compatible = "fsl,imx6q-ahci";
    reg = <0x02200000 0x4000>;
    interrupts = <0 39 0x04>;
    clocks = <&clks 154>, <&clks 187>, <&clks 105>;
    clock-names = "sata", "sata_ref", "ahb";
    status = "okay";
};
```

[...]



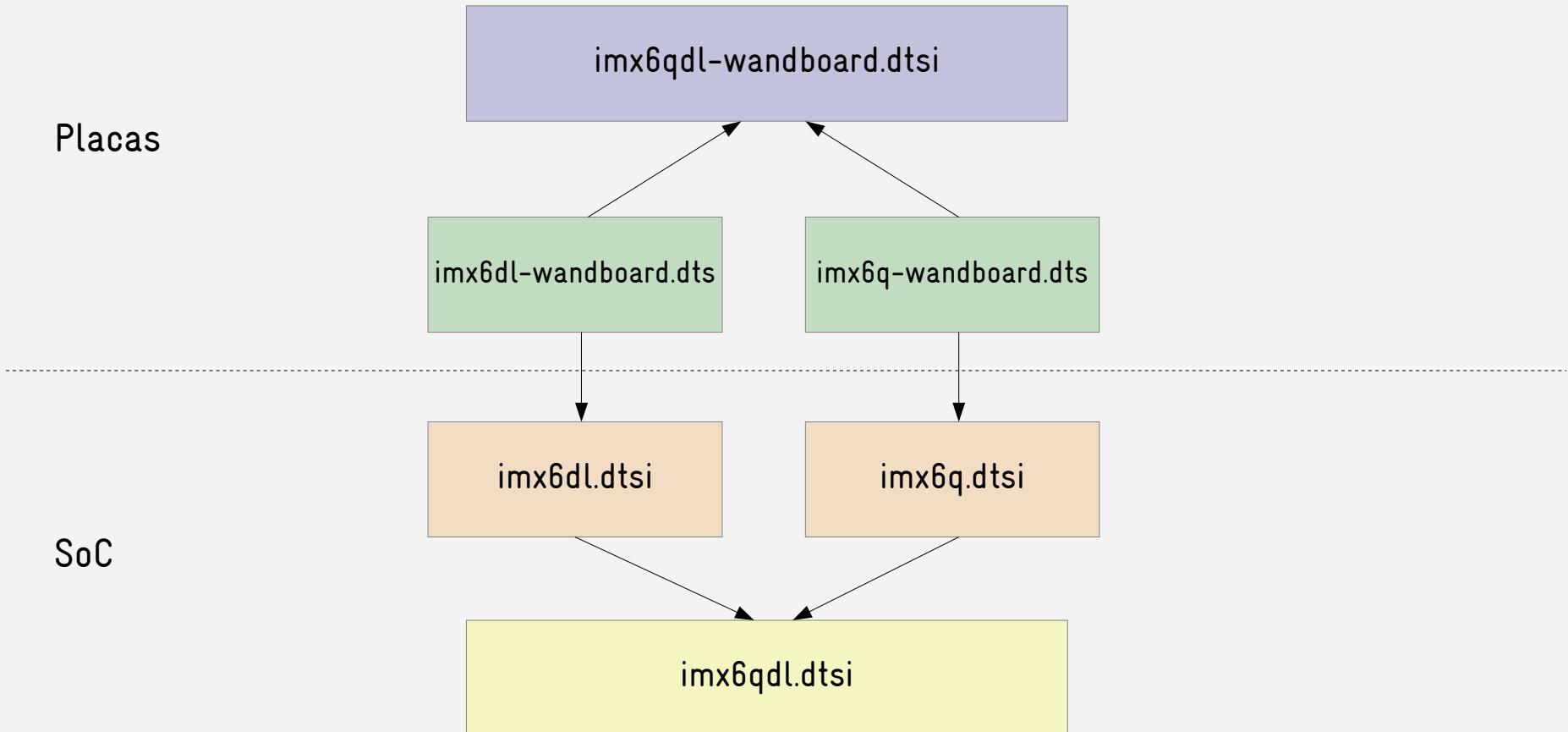


# DIAGRAMA PLACAS E SOC





# DIAGRAMA DEVICE TREES





# DEVICE TREE BINDING

- ✗ A string compatible e as propriedades de um nó são definidas e documentadas em um arquivo chamado device tree binding.
- ✗ É o arquivo de descrição do binding que define como um determinado nó do device tree será tratado pelo kernel.
- ✗ Todos os bindings estão (ou deveriam estar) documentados no kernel em Documentation/devicetree/bindings.





# BINDING SGTL5000

```
/* Documentation/devicetree/bindings/sound/sgtl5000.txt */
```

```
* Freescale SGTL5000 Stereo Codec
```

Required properties:

- compatible : "fsl,sgtl5000".
- reg : the I2C address of the device
- clocks : the clock provider of SYS\_MCLK

Example:

```
codec: sgtl5000@0a {  
    compatible = "fsl,sgtl5000";  
    reg = <0x0a>;  
    clocks = <&clks 150>;  
};
```





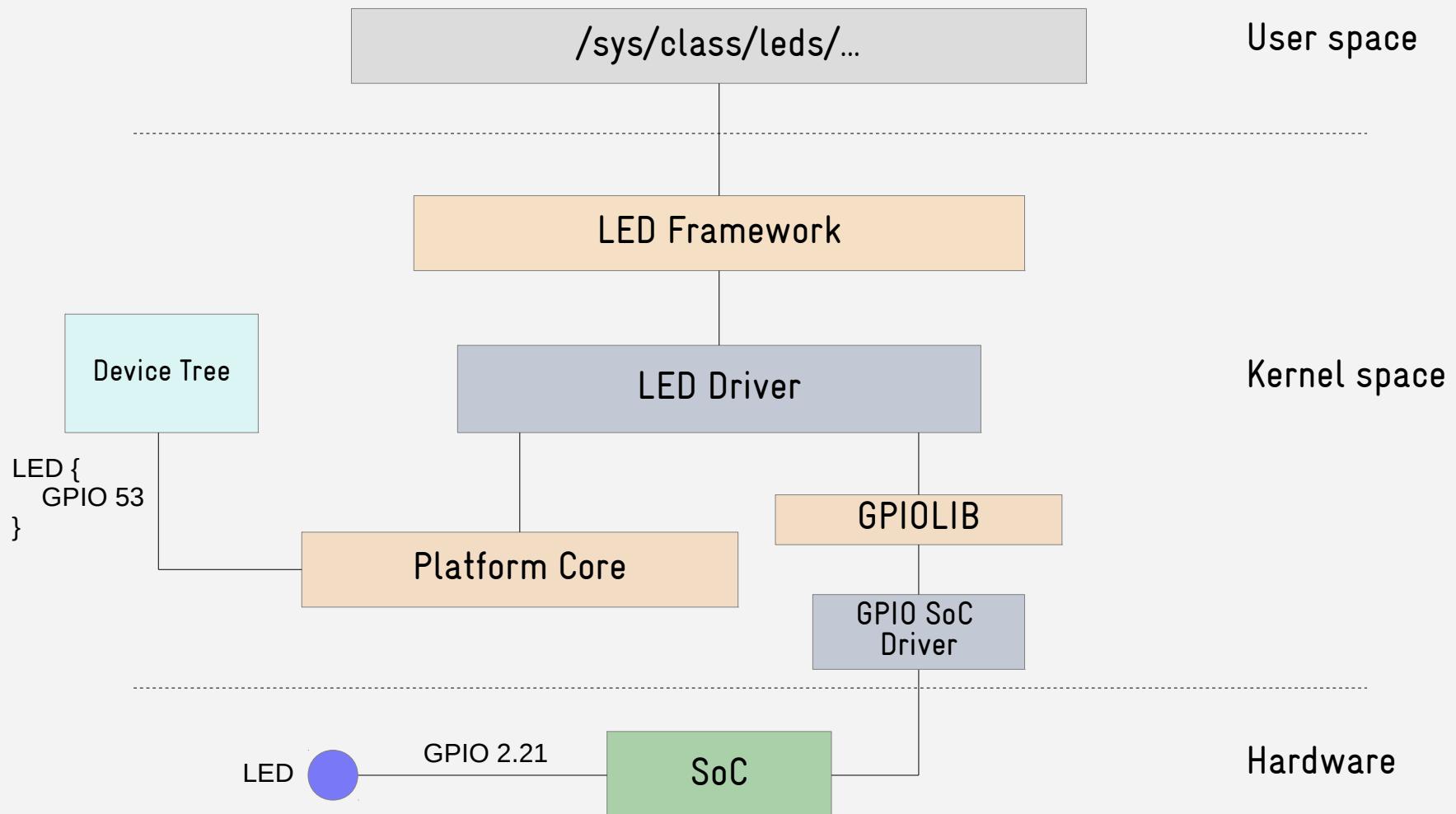
Embedded Labworks

# LABORATÓRIO

Trabalhando com device tree

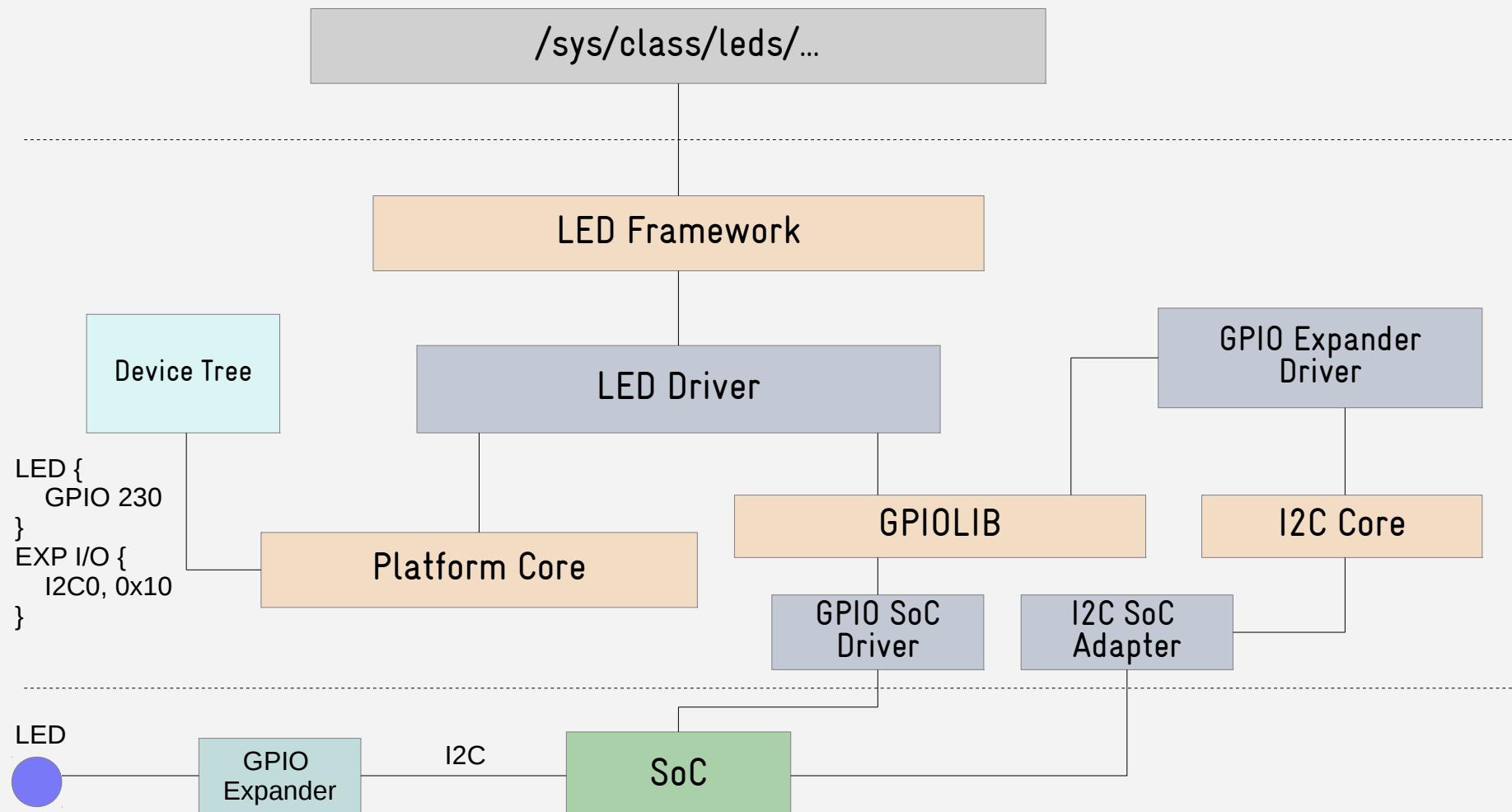


# A FLEXIBILIDADE DO DRIVER MODEL



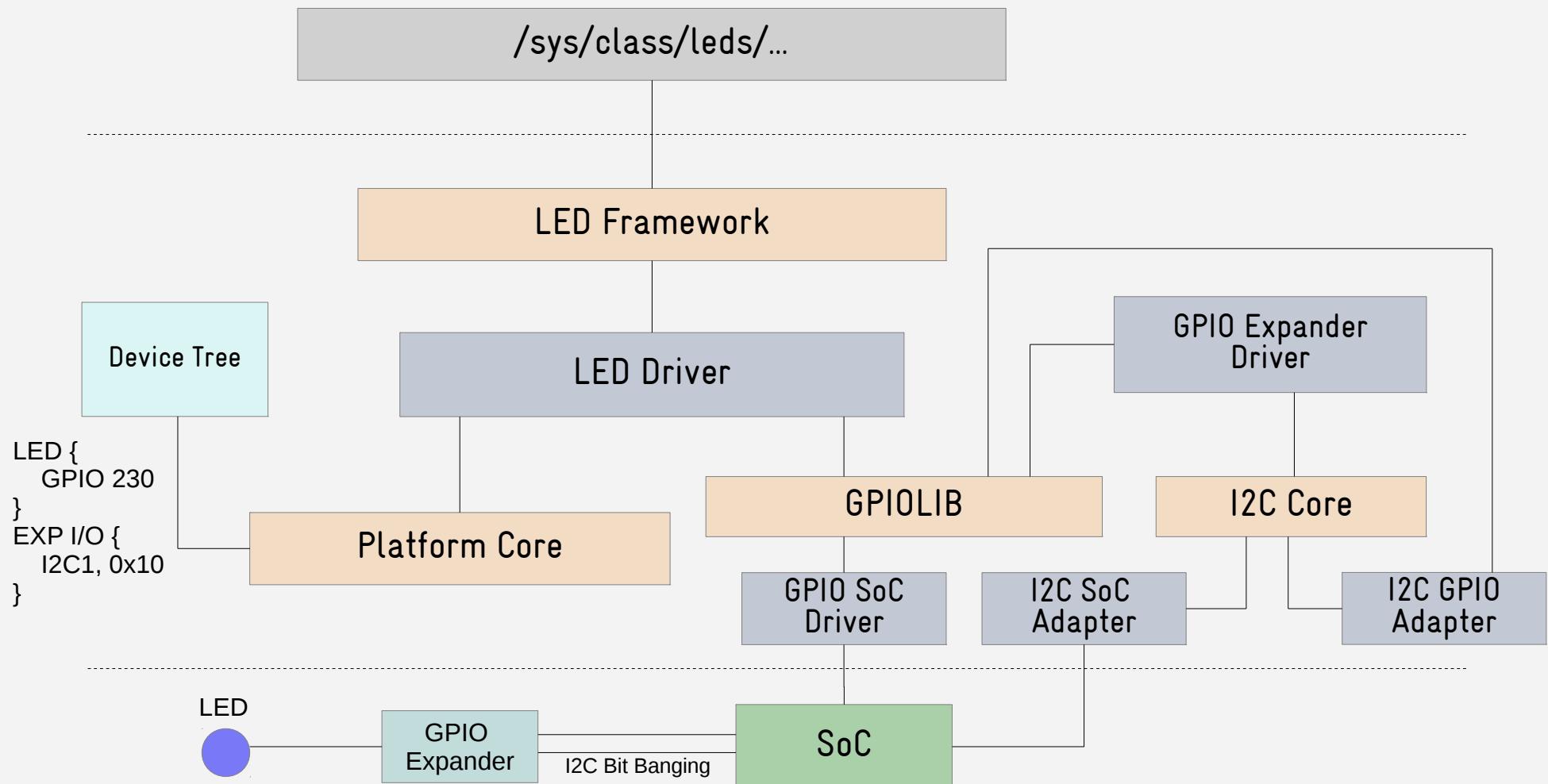


# A FLEXIBILIDADE DO DRIVER MODEL (cont.)





# A FLEXIBILIDADE DO DRIVER MODEL (cont.)





Embedded Labworks

# Linux device drivers

Gerenciamento de processos



# PROCESSOS E THREADS

- ✗ Um processo é um programa em execução.
- ✗ Em sistemas Unix, um processo é criado através da chamada de sistema `fork()`.
- ✗ Um processo é composto por:
  - ✗ Um espaço de endereçamento virtual, que contém código, dados, stack, bibliotecas carregadas dinamicamente, etc.
  - ✗ Uma thread, cujo ponto de entrada é a função `main()`.





# PROCESSOS E THREADS (cont.)

- ✗ Dentro de um processo, threads adicionais podem ser criadas com a função `pthread_create()`:
  - ✗ As threads compartilham o mesmo espaço de endereçamento virtual do processo.
  - ✗ O ponto de entrada da thread é passado como argumento em `pthread_create()`.





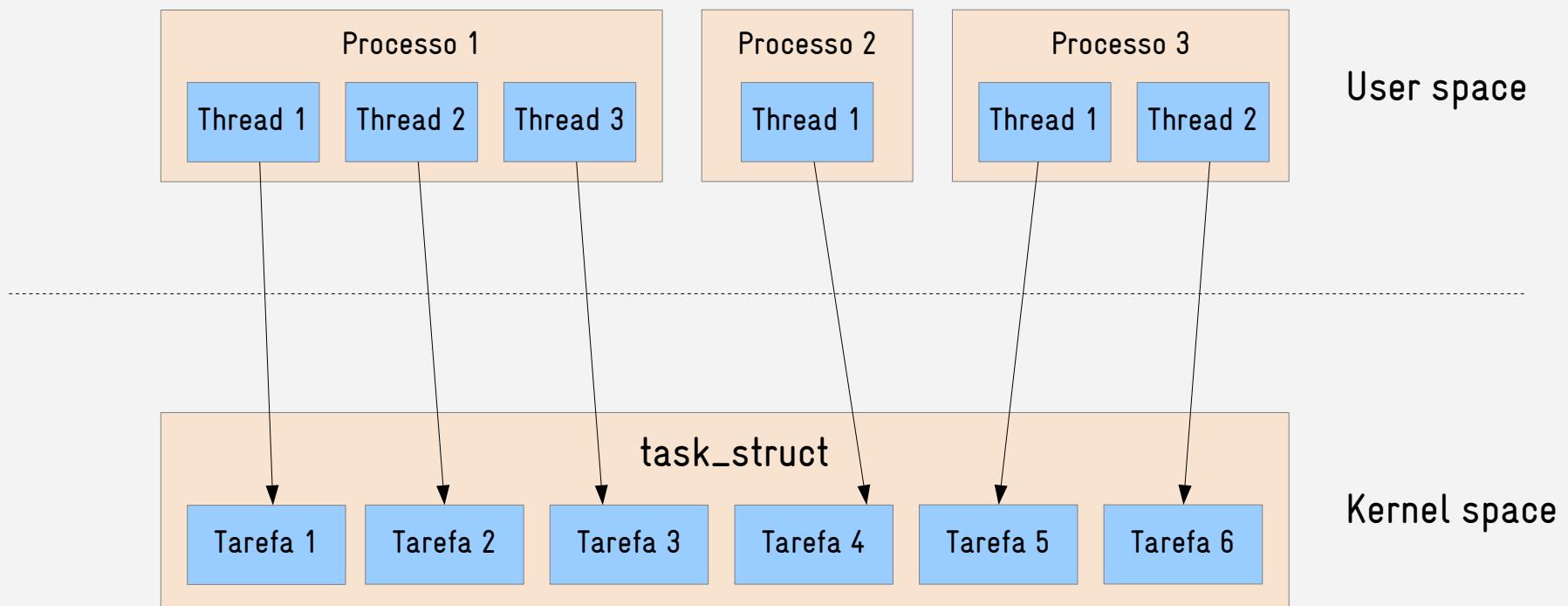
# PROCESSOS E THREADS NO KERNEL

- ✗ Internamente, o kernel não diferencia processos e threads.
- ✗ Uma thread nada mais é do que um processo que compartilha dados com outros processos.
- ✗ Por este motivo, o Linux escalona threads e não processos!
- ✗ A partir de agora, pelo fato do kernel tratar tudo como tarefa (task), vamos nos referir à thread ou processo apenas como tarefa.





# PROCESSOS E TAREFAS





# OS ESTADOS DE UMA TAREFA

- ✗ **TASK\_RUNNING:** A tarefa está em execução ou pronta para ser executada.
- ✗ **TASK\_INTERRUPTIBLE:** A tarefa está bloqueada esperando alguma condição, mas pode ser interrompida se receber um sinal.
- ✗ **TASK\_UNINTERRUPTIBLE:** A tarefa está bloqueada esperando alguma condição, e ignora qualquer sinal.
- ✗ **TASK\_KILLABLE:** A tarefa está bloqueada esperando alguma condição, mas pode ser interrompida com o sinal SIGKILL.

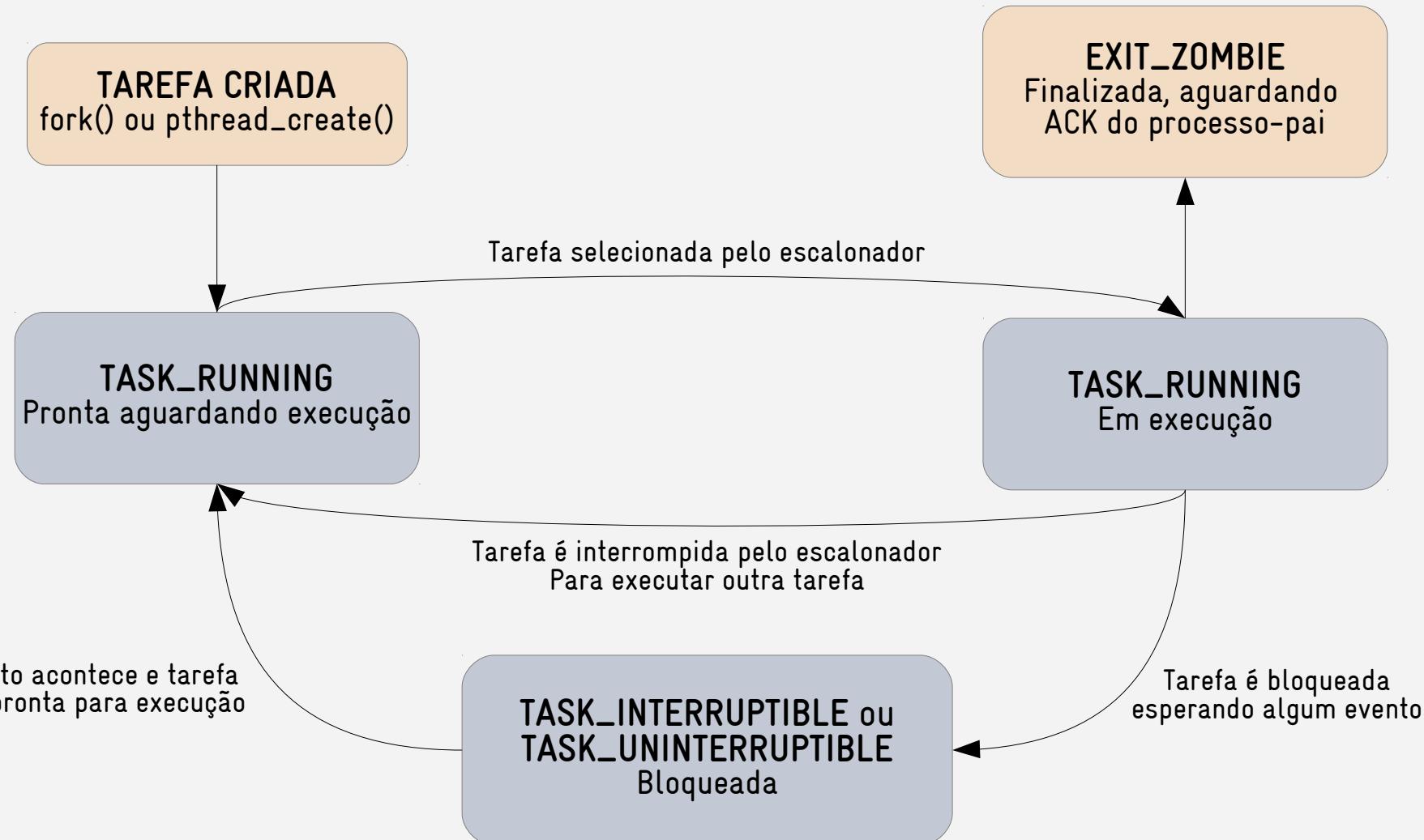




# OS ESTADOS DE UMA TAREFA (cont.)

- ✗ **\_\_TASK\_STOPPED:** A tarefa parou sua execução porque recebeu algum sinal (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) via debugger ou controle de job.
- ✗ **EXIT\_ZOMBIE:** A tarefa terminou, mas o processo-pai ainda não fez a limpeza.
- ✗ **\_\_TASK\_TRACED:** A tarefa está sendo rastreada por outro processo (tracing) via chamada de sistema ptrace( ).







# ESCALONADOR E PREEMPÇÃO

- ✗ O Linux é um sistema multitasking preemptivo.
- ✗ Quando o código de uma tarefa está rodando em user space, a preempção está sempre habilitada, e ela pode acontecer:
  - ✗ No retorno do tratamento de uma interrupção.
  - ✗ No retorno de uma chamada de sistema.
- ✗ Quando o código de uma tarefa está rodando em kernel space (por exemplo dentro de uma chamada de sistema), a preempção só pode acontecer se a opção do kernel `CONFIG_PREEMPT` estiver habilitada.





# CLASSES DE ESCALONAMENTO

- ✗ Como e quando escalar uma tarefa? Trade-off entre capacidade de processamento e tempo de resposta (latência).
- ✗ O kernel Linux possui o conceito de classes de escalonamento, onde cada classe possui um algoritmo que decide qual processo deve ser executado, quando e por quanto tempo.
- ✗ O escalonador padrão do Linux é o CFS (Completely Fair Scheduler), onde cada processo recebe uma "porcentagem justa" da CPU.





# TAREFAS DE TEMPO REAL

- ✗ O Linux possui também duas classes de escalonamento de tempo real (SCHED\_FIFO e SCHED\_RR) , que possuem prioridade sobre o CFS.
- ✗ Mas mesmo assim, o Linux não pode ser considerado um sistema operacional determinístico devido aos tempos de latência para atender aos eventos do sistema.
- ✗ Existe um conjunto de patches (PREEMPT\_RT) que podem ser aplicados ao kernel e melhorar este cenário.
- ✗ Uma opção para o uso do Linux em aplicações hard real-time é a utilização de um kernel de tempo real em conjunto com o Linux (RTLinux, RTAI, Xenomai).





# KERNEL THREADS

- ✗ Kernel threads são tarefas comuns, mas com algumas diferenças com relação às threads que rodam em espaço de usuário:
  - ✗ São executadas sempre em kernel space.
  - ✗ Não possuem um mapa de memória virtual.
- ✗ Muito útil para o kernel ou algum device driver executar operações em background.
- ✗ As threads do kernel aparecem na listagem do programa ps entre colchetes.





# KERNEL THREADS API

- ✗ A API para a criação de threads do kernel está disponível em `<linux/kthread.h>`.
- ✗ Você pode usar a função `kthread_create()` para criar uma thread do kernel. Neste caso a thread é criada de forma bloqueada, e você precisa acordá-la com a função `wake_up_process()`.
- ✗ Você pode também usar a função `kthread_run()` para criar e executar uma thread do kernel. Neste caso a thread é criada e colocada no estado `TASK_RUNNING`, pronta para execução.
- ✗ Para parar uma thread do kernel, você pode usar a função `kthread_stop()`.





# KERNEL THREADS API (cont.)

```
#include <linux/kthread.h>

/* create a kernel thread */
struct task_struct *kthread_create(
    int (*threadfn)(void *data),
    void *data,
    const char namefmt[], ...);

/* create and run a kernel thread */
struct task_struct *kthread_run(int (*threadfn)(void *data),
    void *data,
    const char namefmt[], ...);

/* stop a kernel thread */
int kthread_stop(struct task_struct *k);
```





# KERNEL THREADS API (EXEMPLO)

```
#include <linux/kthread.h>

/* create and start task */
struct task_struct *mytask;
mytask = kthread_run(mythread, NULL, "mythread");

/* kernel thread */
static int mythread(void *unused)
{
    while (!kthread_should_stop()) {
        /* do something */
    }
    return(0);
}

/* stop task */
kthread_stop(mytask);
```





# EVENTOS

- ✗ Boa parte das tarefas que desenvolvemos são baseadas em eventos, ou seja, elas devem esperar um evento para continuar seu processamento. Exemplos:
  - ✗ Eventos temporais ou delay (ex: esperar 100ms).
  - ✗ Operação de I/O (ex: esperar a leitura de um arquivo).
  - ✗ Evento de hardware (ex: esperar uma tecla ser pressionada).
  - ✗ Mecanismos de sincronização (ex: esperar a liberação de um mutex).
- ✗ Uma forma de implementar tarefas que esperam eventos é através de polling. Mas ao usar polling, a tarefa irá monopolizar e desperdiçar ciclos preciosos de CPU.





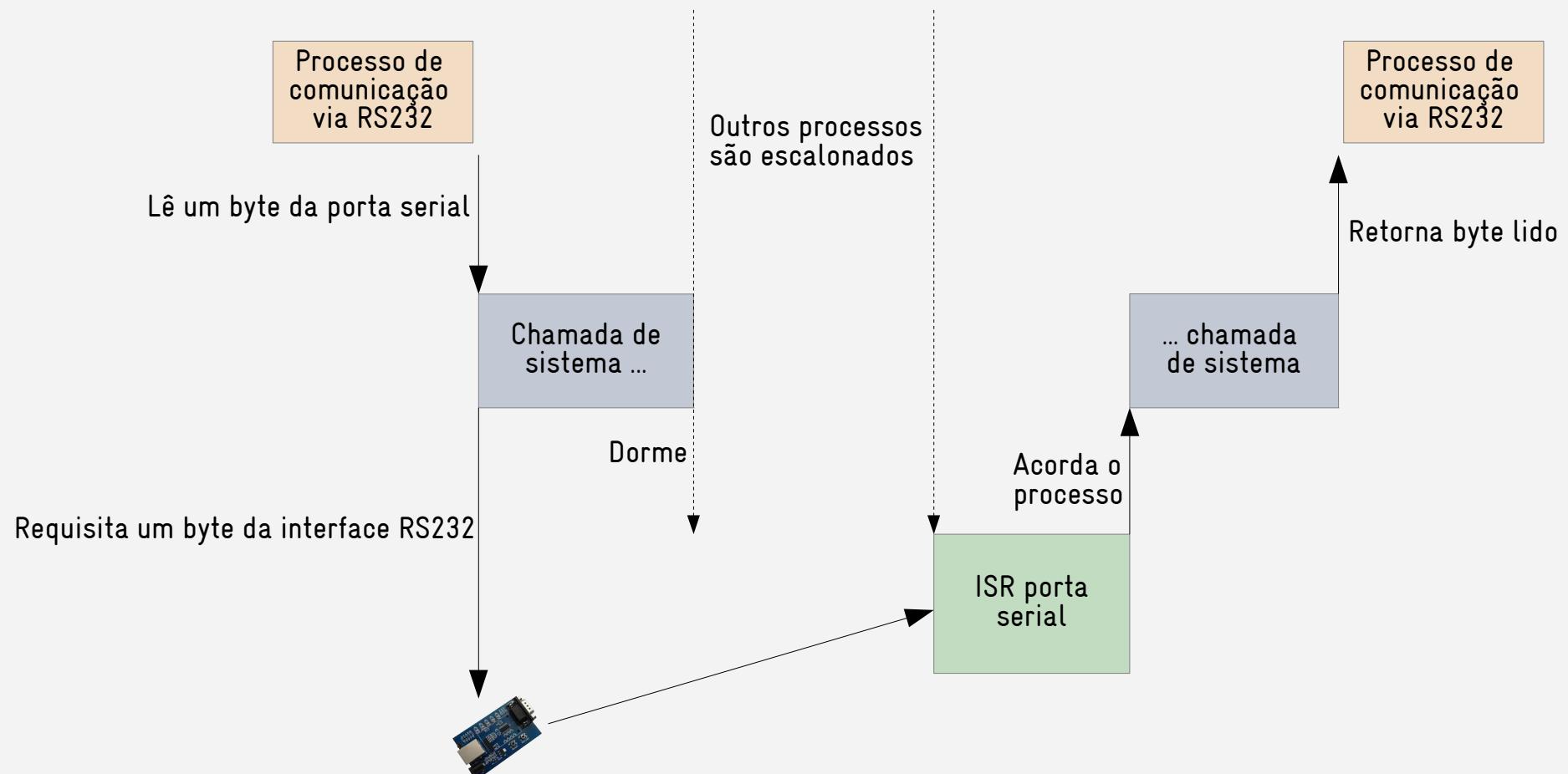
# SLEEPING

- ✗ O ideal nestes casos é trabalhar com interrupção. Ou seja, colocar a tarefa para dormir e pedir para o kernel acordá-la assim que o evento acontecer.
- ✗ Quando uma tarefa dorme, o kernel coloca ela em um dos estados `TASK_INTERRUPTIBLE` ou `TASK_UNINTERRUPTIBLE`, e seleciona outra tarefa para execução.
- ✗ Assim que o evento acontecer, o kernel coloca a tarefa novamente no estado `TASK_RUNNING`, pronta para ser executada.





# SLEEPING (cont.)





# WAIT QUEUES

- ✗ Para dormir, você deve declarar uma `wait queue`, que mantém uma lista de tarefas esperando um evento acontecer.
- ✗ Uma `wait queue` é definida através da variável `wait_queue_head_t`, que pode ser inicializada de suas formas:
  - ✗ Estaticamente com a macro `DECLARE_WAIT_QUEUE_HEAD()`.
  - ✗ Dinamicamente com a função `init_waitqueue_head()`.
- ✗ Os processos devem dormir usando uma das funções do tipo `wait_event*()` e devem ser acordados através das funções do tipo `wake_up*()`.





# SLEEPING API

```
#include <linux/wait.h>
#include <linux/sched.h>

/* declare wait queue statically */
DECLARE_WAIT_QUEUE_HEAD(my_queue);

/* declare wait queue dynamically */
wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);
```





# SLEEPING API (cont.)

```
#include <linux/wait.h>

/* Sleeps until the task is woken up and the given C expression
   is true. Caution: can't be interrupted (can't kill the user-
   space process!) */
wait_event(my_queue, condition);

/* Can be interrupted, but only by a "fatal" signal (SIGKILL).
   Returns -ERESTARTSYS if interrupted. */
ret = wait_event_killable(my_queue, condition);

/* Can be interrupted by any signal. Returns -ERESTARTSYS if
   interrupted. */
ret = wait_event_interruptible(my_queue, condition);
```





# SLEEPING API (cont.)

```
#include <linux/wait.h>

/* Also stops sleeping when the task is woken up and the
   timeout expired. Returns 0 if the timeout elapsed, non-zero
   if the condition was met. */
ret = wait_event_timeout(my_queue, condition, timeout);

/* Same as above, interruptible. Returns 0 if the timeout
   elapsed, -ERESTARTSYS if interrupted, positive value if the
   condition was met. */
ret = wait_event_interruptible_timeout(my_queue, condition,
                                       timeout);
```





# SLEEPING API (cont.)

```
#include <linux/wait.h>

/* wakes up all processes in the wait queue */
wake_up(&my_queue);

/* wakes up all processes waiting in an interruptible sleep on
 * the given queue */
wake_up_interruptible(&my_queue);
```





# SLEEPING API (EXEMPLO)

```
#include <linux/wait.h>
#include <linux/sched.h>

/* declare and initialize wait queue */
wait_queue_head_t wait;
init_waitqueue_head(&wait);

/* wait on a wait queue (called on a task) */
ready = 0;
ret = wait_event_interruptible(wait, ready != 0);

/* wake up all process waiting on the wait queue
   (called inside an interrupt handler for example) */
ready = 1;
wake_up_interruptible(&wait);
```





Embedded Labworks

# LABORATÓRIO

Kernel threads e wait queues



Embedded Labworks

# Linux device drivers

Gerenciamento de interrupções



# INTERRUPÇÃO

- ✗ Um dos principais trabalhos do sistema operacional é gerenciar e se comunicar com os dispositivos de hardware conectados ao sistema.
- ✗ Para isso, o kernel pode checar o status do hardware periodicamente (polling) e atuar de acordo.
- ✗ Mas como os dispositivos de hardware são muito mais lentos que a CPU, este mecanismo de acesso ao hardware desperdiça preciosos ciclos de CPU.
- ✗ Por este motivo, usamos o mecanismo de interrupção para receber eventos de hardware do sistema.





# INTERRUPÇÃO (cont.)

- ✗ O mecanismo de interrupção possibilita ao hardware a capacidade de sinalizar a CPU quando acontecer algum evento (o botão do mouse foi pressionado, uma tecla foi digitada, um pacote foi recebido da interface Ethernet, etc).
- ✗ Diferentes dispositivos estão associados à diferentes números de interrupção, também chamadas de linhas de IRQ (Interrupt Request).
- ✗ Quando a CPU recebe o sinal de interrupção, interrompe a execução atual, e de acordo com a linha de IRQ, executa uma rotina de tratamento de interrupção registrada no sistema (também chamada de interrupt handler ou ISR – Interrupt Service Routine).





# INTERRUPÇÃO NO KERNEL

- ✗ Para um device driver usar uma linha de interrupção, ele deve:
  - ✗ Implementar a rotina de tratamento de interrupção (ISR).
  - ✗ Requisitar o uso da linha de IRQ quando for iniciar o uso do dispositivo de hardware através da função `request_irq()`.
  - ✗ Liberar o uso da linha de IRQ quando não for mais usar o dispositivo através da função `free_irq()`.





# IRQ API

```
#include <linux/interrupt.h>

/* request IRQ, returns 0 on success */
int request_irq(unsigned int irq,
                 irq_handler_t handler,
                 unsigned long irq_flags,
                 const char *devname,
                 void *dev_id);

/* free IRQ */
void free_irq(unsigned int irq, void *dev_id);
```





# IRQ FLAGS

- ✗ Estas são algumas das flags que podem ser passadas para registrar uma IRQ:
  - ✗ IRQF\_SHARED: O canal de IRQ pode ser compartilhado por várias ISRs. Neste caso, o hardware precisa prover um mecanismo para que a ISR possa verificar a origem da interrupção.
  - ✗ IRQF\_ONESHOT: Interrupção não é reabilitada ao finalizar a execução da ISR.
  - ✗ IRQF\_NO\_SUSPEND: Não desabilita a interrupção ao suspender o sistema.
- ✗ Mais informações sobre as flags nos fontes do kernel em `include/linux/interrupt.h`.





# IRQ API (EXEMPLO)

```
#include <linux/interrupt.h>

/* request IRQ */
if (request_irq(irqn, mydriver_isr, IRQF_SHARED,
                 "mydriver", mydriver.dev)) {
    pr_err("Error requesting IRQ!\n");
    [...]
}

/* free IRQ */
free_irq(irqn, mydriver.dev);
```





# IRQS REGISTRADAS NO SISTEMA

- Para verificar as IRQs registradas no sistema, basta listar o arquivo /proc/interrupts:

```
# cat /proc/interrupts
          CPU0
  1:      45      tzic  mmc0
  3:          0      tzic  mmc1
  6:          0      tzic  sdma
 31:      44      tzic  IMX-uart
 39:    5638      tzic  i.MX Timer Tick
 62:          0      tzic  imx-i2c
 87:    6515      tzic  imx25-fec
235:          0  gpio-mxc  mmc1
237:          0  gpio-mxc  mmc0
[...]
```





# PROTÓTIPO DE UMA ISR

```
/**  
 * ISR prototype.  
 * @irq: the IRQ number  
 * @dev_id: the opaque pointer passed at request_irq()  
 *  
 * Should return IRQ_HANDLED if interrupt was recognized  
 * and handled or IRQ_NONE if interrupt wasn't recognized  
 * or managed by module.  
 */  
irqreturn_t foo_interrupt(int irq, void *dev_id);
```





# RESTRIÇÕES DE UMA ISR

- ✗ Como não existe uma garantia de qual espaço de endereçamento o sistema estará quando uma interrupção acontecer, não é possível trocar dados com o espaço de usuário.
- ✗ A execução da rotina de tratamento de interrupção é gerenciada pela CPU, e não pelo escalonador. Ou seja, uma ISR roda em **contexto de interrupção**, e não em contexto de processo!
- ✗ Por este motivo, uma ISR não pode bloquear (dormir), já que ela não roda em contexto de processo, e não pode ser escalonada.





# RESTRIÇÕES DE UMA ISR (cont.)

- ✗ Desde o kernel 2.6.36, quando uma ISR é executada:
  - ✗ A linha de IRQ da interrupção é desabilitada em todas as CPUs (por isso uma ISR não precisa ser reentrante).
  - ✗ Todas as interrupções locais (da mesma CPU) são desabilitadas. Por este motivo, uma ISR deve executar seu trabalho rapidamente e retornar.





# IMPLEMENTANDO UMA ISR

- ✗ Reconhecer a interrupção, por exemplo setando um bit em um registrador de controle do dispositivo.
- ✗ Ler os dados ou evento recebido.
- ✗ Tratar os dados ou evento recebido.
- ✗ Sinalizar o processo esperando dados ou evento do dispositivo, usando por exemplo wait queues:  
`wake_up_interruptible(&mydriver_queue);`





# EXEMPLO ISR

```
irqreturn_t mydriver_interrupt(int irq, void *dev_id)
{
    /* ack interrupt */
    ack();

    /* read device data */
    dev_id->data = getdata();

    /* handle device data */
    handle_data(dev_id->data);

    /* wake up process waiting on wait queue */
    wake_up_interruptible(&mydriver_queue);

    /* return IRQ handled */
    return IRQ_HANDLED;
}
```





# TOP HALF E BOTTOM HALF

- ✗ Em alguns casos, uma interrupção precisa realizar uma quantidade grande de trabalho, ao mesmo tempo que sua execução precisa ser bem rápida. São necessidades conflitantes!
- ✗ Para estes casos, o processamento pode ser dividido em duas partes:
  - ✗ Top Half.
  - ✗ Bottom Half.





# TOP HALF

- ✗ Este é próprio código da ISR, que é executado assim que acontece a interrupção.
- ✗ Deve retornar o mais rápido possível, já que as interrupções estão desabilitadas.
- ✗ Por esse motivo, ele deve delegar o tratamento dos dados ou evento recebido para um mecanismo de bottom half.





# BOTTOM HALF

- ✗ Será executado em algum momento no futuro, com todas as interrupções habilitadas.
- ✗ Desta forma, o impacto é menor se levar mais tempo para realizar o processamento delegado pela interrupção.
- ✗ Existem basicamente dois principais mecanismos de bottom half no Linux: **tasklets** e **workqueues**.





# EXEMPLO BOTTOM HALF

```
/* top half - driver ISR */
irqreturn_t mydriver_interrupt(int irq, void *dev_id)
{
    ack();
    dev_id->data = getdata();
    enable_bottom_half();
    return IRQ_HANDLED;
}

/* bottom half - execute with interrupts enabled */
void bottom_half_function()
{
    /* do the heavy work here! */
    handle_data();

    /* wake up process waiting on wait queue */
    wake_up_interruptible(&mydriver_queue);
}
```





# TASKLETS

- ✗ Tasklets são uma forma de processamento de interrupção em bottom half.
- ✗ São executadas em contexto de interrupção, depois que todas as interrupções forem processadas, normalmente após a execução da ISR.
- ✗ Como são executadas com todas as interrupções habilitadas, diminui o impacto do tratamento de uma ISR no tempo de resposta do sistema.
- ✗ Pelo fato de rodarem em contexto de interrupção, não podem bloquear (dormir).





## TASKLETS (cont.)

- ✗ Foram desenvolvidas em cima da API de softirq do kernel.
- ✗ Pode-se criar tasklets com prioridade alta ou normal. As tasklets com prioridade alta são executadas antes das tasklets com prioridade normal.
- ✗ As tasklets não precisam ser thread-safe, já que não existe a possibilidade do kernel executar a mesma tasklet ao mesmo tempo em CPUs diferentes.
- ✗ Dependendo da carga do sistema, as tasklets podem ser tratadas também em threads do kernel (vide ksoftirqd/X, onde X é o número da CPU).





# TASKLETS API

- ✗ Uma tasklet pode ser declarada estaticamente com a macro `DECLARE_TASKLET()` ou criada dinamicamente com a função `tasklet_init()`.
- ✗ Uma ISR pode delegar trabalho para uma tasklet usando uma das duas funções abaixo:
  - ✗ `tasklet_schedule()`: executa a tasklet com prioridade normal.
  - ✗ `tasklet_hi_schedule()`: executa a tasklet com prioridade alta.





# TASKLET API (cont.)

```
#include <linux/interrupt.h>

/* init tasklet */
void tasklet_init(struct tasklet_struct *t,
                  void (*func)(unsigned long) ,
                  unsigned long data);

/* schedule tasklet */
void tasklet_schedule(struct tasklet_struct *t)

/* remove tasklet */
void tasklet_kill(struct tasklet_struct *t)
```





# TASKLET API (EXEMPLO)

```
#include <linux/interrupt.h>

/* The tasklet function */
static void atmel_tasklet_func(unsigned long data) {
    struct uart_port *port = (struct uart_port *)data;
    [...]
}

/* Registering the tasklet (ex: on an init function) */
tasklet_init(&atmel_port->tasklet, atmel_tasklet_func,
             (unsigned long)port);

/* Triggering execution of the tasklet (ex: on an ISR) */
tasklet_schedule(&atmel_port->tasklet);

/* Removing the tasklet (ex: on a cleanup function) */
tasklet_kill(&atmel_port->tasklet);
```





# WORK QUEUES

- ✗ Work queue é um mecanismo genérico de deferir trabalho, não só limitado ao tratamento de interrupções.
- ✗ A função definida como work queue é executada em uma thread do kernel (kworker) com todas as interrupções habilitadas.
- ✗ Como uma work queue roda em uma thread do kernel, ela é executada em contexto de processo, e por este motivo pode bloquear (dormir).
- ✗ Portanto, se o processamento deferido pela CPU precisa bloquear (dormir), em vez de usar tasklets, você deverá usar work queues.





# WORK QUEUES (cont.)

- ✗ Em sua forma mais comum, a work queue é uma interface para deferir trabalho para uma thread genérica do kernel (events/n ou kworker/n:id, dependendo da versão do kernel, onde n é o número da CPU e id é o id da work queue).
- ✗ Um trabalho a ser delegado (work) pode ser criado com `DECLARE_WORK()` ou `INIT_WORK()`, e é normalmente acionado com `schedule_work()`.





# WORK QUEUE API

```
#include <linux/workqueue.h>

/* create work statically */
DECLARE_WORK(name, void (*func)(void *));

/* create work dinamically */
INIT_WORK(struct work_struct *work, void (*func)(void *));

/* schedule work on default work queue thread */
int schedule_work(struct work_struct *work);

/* schedule delayed work on default work queue thread */
int schedule_delayed_work(struct work_struct *work,
                          unsigned long delay);
```





# WORK QUEUE API (EXEMPLO)

```
void init_function()
{
    INIT_WORK(&gpio->work, pcf857x_irq_demux_work);
    [...]
}

static irqreturn_t isr_function(int irq, void *data)
{
    [...]
    schedule_work(&gpio->work);
    return IRQ_HANDLED;
}

static void pcf857x_irq_demux_work(struct work_struct *work)
{
    /* do the work here! */
}
```





# TASKLETS x WORK QUEUES

Bottom Half	Contexto	Serializado
Tasklet	Interrupção	Sim (na mesma tasklet)
Work queues	Processo	Não





# QUAL BOTTOM HALF USAR?

- ✗ Work queues envolvem um overhead maior pelo uso de threads do kernel e pelas trocas de contexto envolvidas. Você deverá usar work queues se o trabalho for demorado ou se precisar dormir.
- ✗ Tasklets envolvem menos overhead e devem ser o método preferido se seu trabalho for mais rápido e não precisar dormir.
- ✗ Além disso, work queues são melhores para sistemas de baixa latência, e tasklets são melhores para sistemas cujo foco é processamento intensivo.





Embedded Labworks

# LABORATÓRIO

Trabalhando com interrupções



Embedded Labworks

# Linux device drivers

Mecanismos de sincronização



# MULTITHREAD E CONCORRÊNCIA

- ✗ Em um ambiente multithread, recursos compartilhados do sistema precisam ser protegidos de acesso concorrente.
- ✗ Em termos de concorrência, o kernel tem as mesmas restrições de um programa multithread, o seu estado é global e visível a todos os contextos de execução (interrupção e processo).





# CONCORRÊNCIA NO KERNEL

- ✗ Os problemas de concorrência no kernel podem acontecer porque:
  - ✗ A preempção do kernel pode interromper a execução de uma chamada de sistema para executar outra chamada de sistema, e ambas podem estar usando recursos compartilhados.
  - ✗ Uma interrupção pode interromper a execução de um processo, e ambos podem estar usando recursos compartilhados.
  - ✗ Em sistemas SMP (com múltiplas CPUs) podemos ter processos rodando em paralelo em diferentes processadores, que podem estar usando recursos compartilhados.





# CRITICAL SESSION

- × Um trecho de código que acessa recursos compartilhados é chamado de **sessão crítica** (critical session).
- × Para prevenir o acesso concorrente à uma sessão crítica, o acesso deve ser **atômico**, ou seja, a operação em cima do recurso compartilhado deve ser executada do começo ao fim sem interrupção.





# SOLUÇÃO

- ✗ Evite regiões críticas! Sempre que possível, mantenha um estado local dos processos.
- ✗ Caso você precise de recursos compartilhados, identifique quais são estes recursos:
  - ✗ Se o seu recurso compartilhado for um número inteiro, você pode usar as funções atômicas disponibilizadas pelo kernel.
  - ✗ Em outros casos, como estruturas de dados mais complexas ou recursos de hardware, você deverá utilizar um mecanismo de locking.





# ATOMIC OPERATIONS

- ✗ Mesmo uma operação simples em inteiros do tipo `x++` ou `x |= 1` não é garantida que seja atômica em todas as arquiteturas!
- ✗ O recurso de operações atômicas do kernel possibilita proteger um recurso compartilhado do tipo inteiro de acesso concorrente.
- ✗ O Linux fornece duas APIs de operações atômicas:
  - ✗ Operações em número inteiros, definida em `<asm/atomic.h>`.
  - ✗ Operações em bits, definida em `<asm/bitops.h>`.





# ATOMIC OPERATIONS API

```
#include <asm/atomic.h>

/* set or read the variable */
void atomic_set(atomic_t *v, int i);
int atomic_read(atomic_t *v);

/* change variable */
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
```





# ATOMIC OPERATIONS API (cont.)

```
#include <asm/atomic.h>

/* change and return true if result is zero */
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_add_and_test(int i, atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);

/* change and return result */
int atomic_inc_and_return(atomic_t *v);
int atomic_dec_and_return(atomic_t *v);
int atomic_add_and_return(int i, atomic_t *v);
int atomic_sub_and_return(int i, atomic_t *v);
```





# ATOMIC BIT OPERATIONS API

```
#include <asm/bitops.h>

/* set, clear or toggle a given bit */
void set_bit(int nr, unsigned long * addr);
void clear_bit(int nr, unsigned long * addr);
void change_bit(int nr, unsigned long * addr);

/* test bit and return its value */
int test_bit(int nr, unsigned long *addr);

/* set a bit and return its old value */
int test_and_set_bit(int nr, unsigned long *addr);
int test_and_clear_bit(int nr, unsigned long *addr);
int test_and_change_bit(int nr, unsigned long *addr);
```





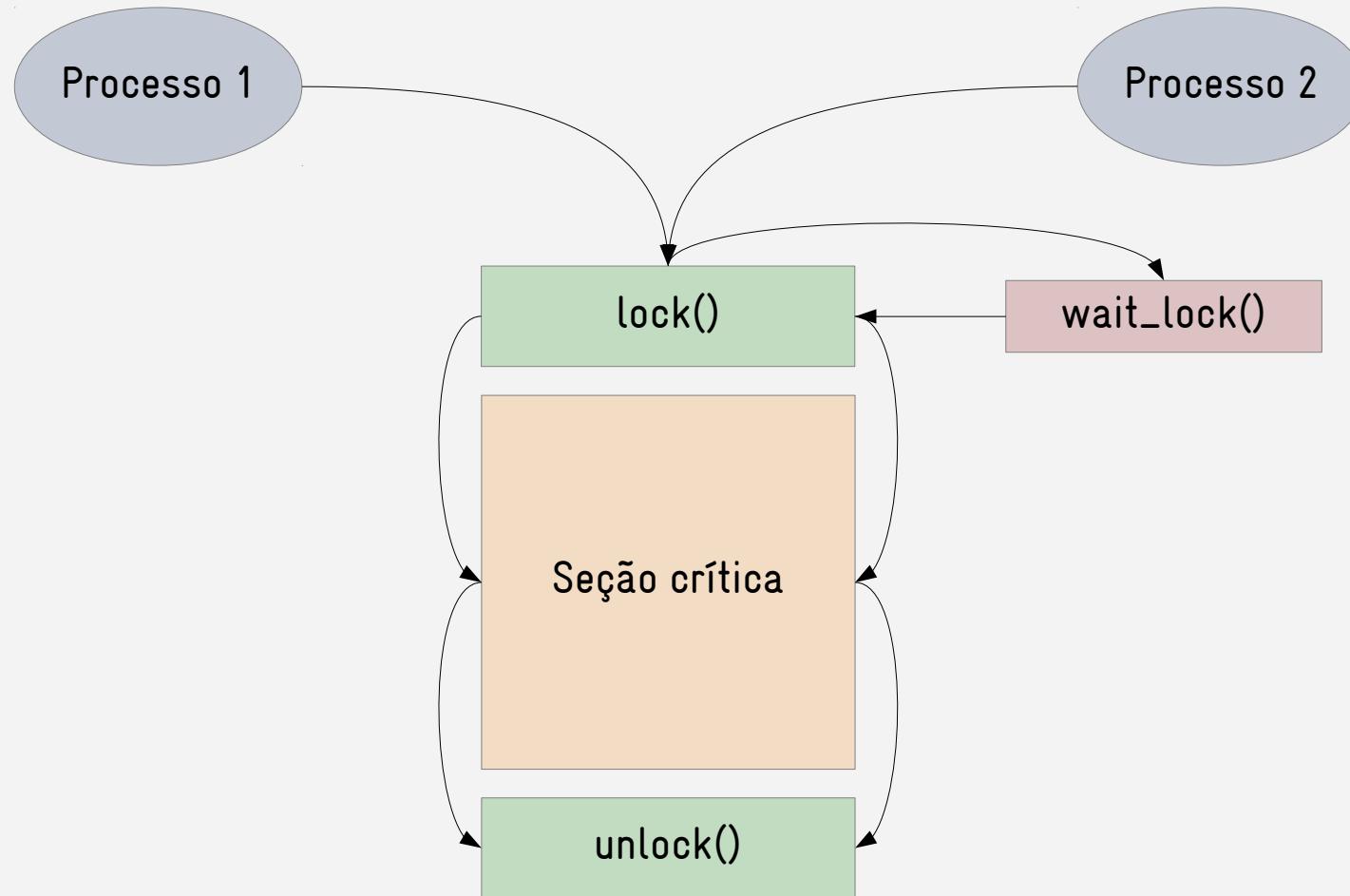
# LOCKING

- ✗ Nem sempre um recurso compartilhado pode ser representado por uma variável do tipo inteiro.
- ✗ Na maioria das vezes, a região crítica que precisamos proteger atua em cima de estruturas de dados mais complexas.
- ✗ Para estes casos, devemos proteger o acesso ao recurso compartilhado através de mecanismos de lock.





# LOCKING (cont.)





# SEMÁFORO

- ✗ O semáforo é um mecanismo de sincronização entre tarefas, podendo ser usado como mecanismo de locking no acesso à recursos compartilhados.
- ✗ A API de semáforos está disponível em <asm/semaphore.h>.
- ✗ O uso de semáforos como mecanismo de locking no Linux foi descontinuado desde a versão 2.6.16.





# MUTEX

- ✗ A partir da versão 2.6.16, a funcionalidade de mutex (abreviação de mutual exclusion) foi implementada e adotada como mecanismo padrão de locking para gerenciar o acesso à recursos compartilhados no kernel.
- ✗ Uma das principais diferenças de um mutex para um semáforo é que, quando usamos um mutex, apenas a tarefa que travou o mutex pode destravá-lo.
- ✗ Além disso, um mutex tem suporte à herança de prioridade.
- ✗ Como o processo que requisita o lock pode bloquear (dormir) se este lock estiver indisponível, um mutex só pode ser usado em contexto de processo.





# MUTEX API

```
#include <linux/mutex.h>

/* initializing a mutex statically */
DEFINE_MUTEX(mutex_name);

/* initializing a mutex dynamically */
void mutex_init(struct mutex *lock);
```





# MUTEX API (cont.)

```
#include <linux/mutex.h>

/* tries to lock the mutex, sleeps otherwise. Can't be
   interrupted, resulting in processes you cannot kill! */
void mutex_lock(struct mutex *lock);

/* releases the lock, do it as soon as you leave the
   critical section */
void mutex_unlock(struct mutex *lock);
```





# MUTEX API (cont.)

```
#include <linux/mutex.h>

/* same, but can be interrupted by a fatal (SIGKILL) signal.
   If interrupted, returns a non zero value and doesn't hold
   the lock. Test the return value! */
int mutex_lock_killable(struct mutex *lock);

/* same, but can be interrupted by any signal */
int mutex_lock_interruptible(struct mutex *lock);

/* never waits, returns a non zero value if the mutex is
   not available */
int mutex_trylock(struct mutex *lock);

/* just tells whether the mutex is locked or not */
int mutex_is_locked(struct mutex *lock);
```





# MUTEX API (EXEMPLO)

```
#include <linux/mutex.h>

static int __devinit ads7846_probe(struct spi_device *spi)
{
    [...]
    mutex_init(&ts->lock);
    [...]
}

static void ads7846_enable(struct ads7846 *ts)
{
    mutex_lock(&ts->lock);
    if (ts->disabled) {
        ts->disabled = false;
        [...]
    }
    mutex_unlock(&ts->lock);
}
```





# DEFICIÊNCIAS DOS MUTEXES

- ✗ Os mutexes devem ser o mecanismo preferido para proteger o acesso concorrente à regiões críticas, mas eles possuem duas principais deficiências:
  - ✗ O fato de bloquear não permite seu uso em rotinas de tratamento de interrupção.
  - ✗ Existe um overhead de pelo menos duas trocas de contexto na sua execução (colocar a tarefa para dormir quando o mutex estiver sendo usado e acordar a tarefa quando o mutex for liberado).





# DESABILITANDO INTERRUPÇÕES

- ✗ Em sistemas com uma única CPU, ao compartilhar dados entre um processo e uma interrupção, você pode simplesmente desabilitar as interrupções.
- ✗ Para isso, você pode usar as famílias de funções que habilitam e desabilitam as interrupções, como por exemplo as funções `local_irq_disable()`, `local_irq_enable()`, `local_irq_save()` e `local_irq_restore()`.
- ✗ Mais informações sobre estas funções em `<linux/irqflags.h>`.
- ✗ Lembre-se de que ao desabilitar as interrupções, você estará aumentando o tempo de latência do sistema.





# DESABILITANDO INTERRUPÇÕES (cont.)

- ✗ Ao desabilitar as interrupções, apenas as interrupções da CPU local são desabilitadas.
- ✗ Como então proteger o acesso concorrente à regiões críticas compartilhadas por processos e interrupções, quando o sistema é SMP?
- ✗ Através de **spinlocks**!





# SPINLOCKS

- ✗ O spinlock é implementado como um loop que fica verificando (spinning) em um loop até que o lock esteja disponível, protegendo desta forma o acesso concorrente em sistemas SMP.
- ✗ Para proteger com um spinlock uma sessão crítica compartilhada entre processos e interrupções, você deve também desabilitar as interrupções através das funções `spin_lock_irqsave()` e `spin_unlock_irqrestore()`.
- ✗ Para evitar que o processo que travou o spinlock seja interrompido por outro processo, o spinlock desabilita a preempção do kernel.





# SPINLOCKS (cont.)

- ✗ Pelo fato do spinlock usar polling como mecanismo de locking, a seção crítica protegida pelo spinlock deve ser executada rapidamente e não pode bloquear (dormir).
- ✗ O lock do spinlock só faz sentido em sistemas multicore. Por este motivo, em sistemas com um único core o tratamento do lock é removido, e só o código que habilita/desabilita a preempção é compilado.
- ✗ Em sistemas com um único core e com a preempção do kernel desabilitada, o código relacionado ao spinlock é removido completamente!





# SPINLOCK API

```
#include <linux/spinlock.h>

/* initializing a spinlock statically */
DEFINE_SPINLOCK(my_lock);

/* initializing a spinlock dynamically */
void spin_lock_init(spinlock_t *lock);
```





# SPINLOCK API (cont.)

```
#include <linux/spinlock.h>

/* doesn't disable interrupts. Used for locking in process
   context (critical sections in which you do not want to
   sleep) */
void spin_lock(spinlock_t *lock);
void spin_unlock(spinlock_t *lock);

/* disables/restores IRQs on the local CPU. Typically used
   when the lock can be accessed in both process and
   interrupt context, to prevent preemption by interrupts */
void spin_lock_irqsave(spinlock_t *lock,
                      unsigned long flags);
void spin_unlock_irqrestore(spinlock_t *lock,
                           unsigned long flags);
```





# SPINLOCK API (EXEMPLO)

```
#include <linux/spinlock.h>

static void serio_init_port(struct serio *serio)
{
    [...]
    spin_lock_init(&serio->lock);
    [...]
}

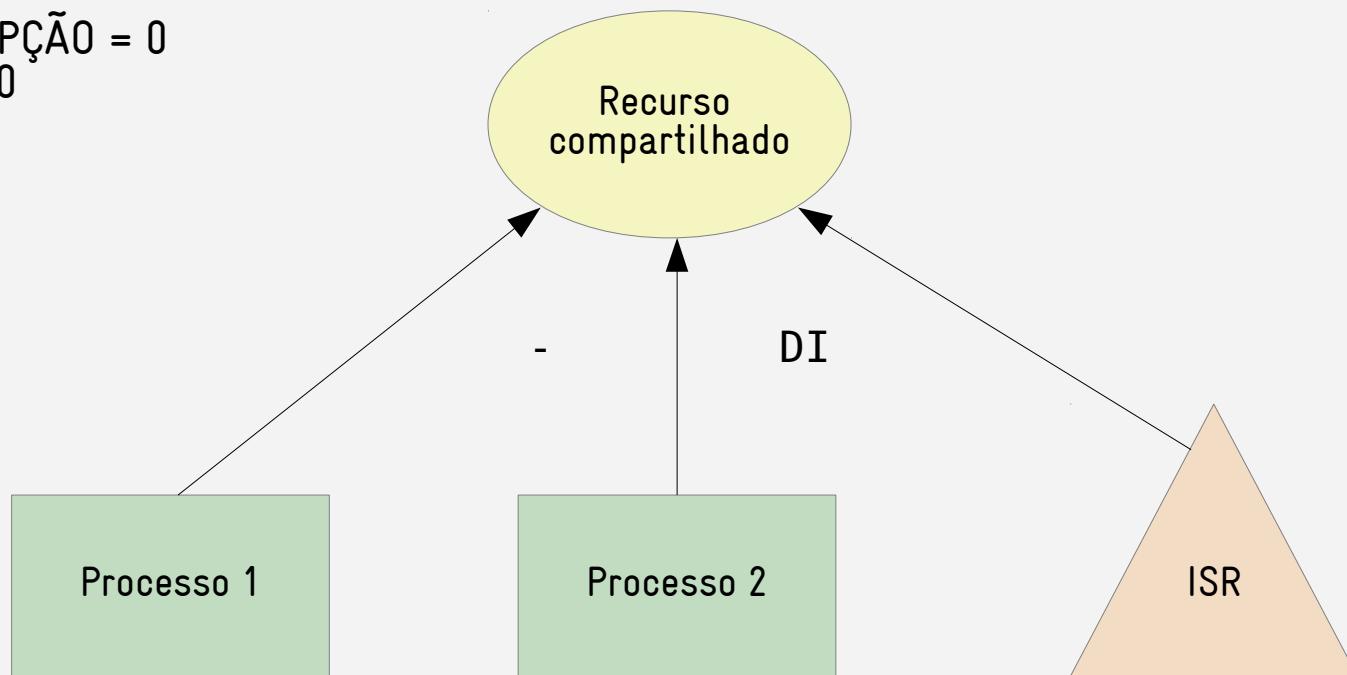
irqreturn_t serio_interrupt(struct serio *serio,
                           unsigned char data, unsigned int dfl)
{
    unsigned long flags;
    spin_lock_irqsave(&serio->lock, flags);
    [...]
    spin_unlock_irqrestore(&serio->lock, flags);
}
```





# CASO 1

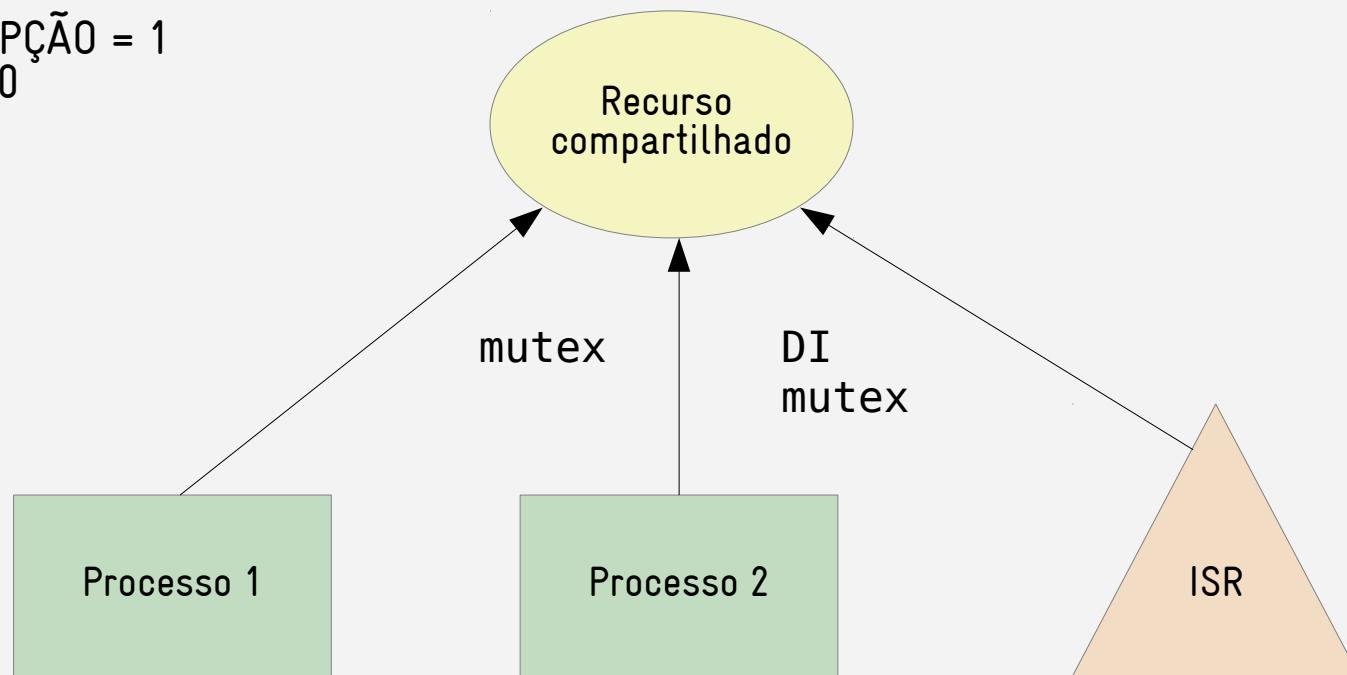
PREEMPÇÃO = 0  
SMP = 0





## CASO 2

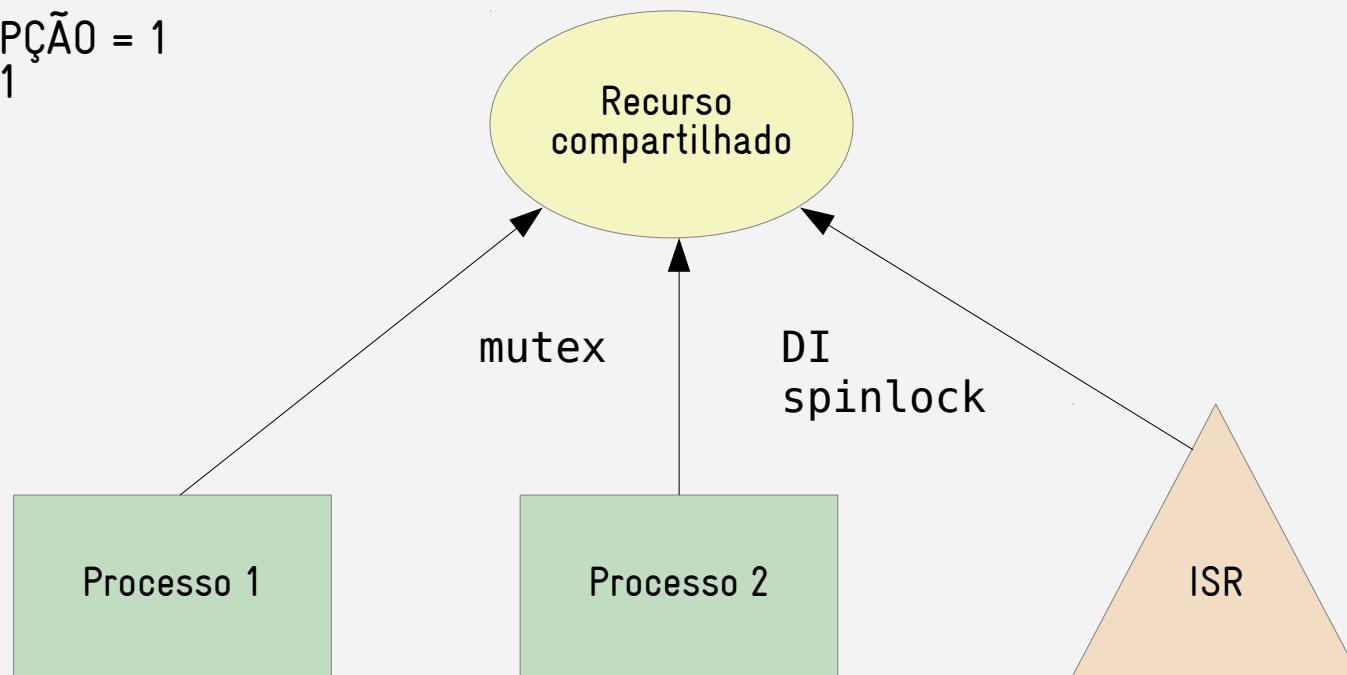
PREEMPÇÃO = 1  
SMP = 0





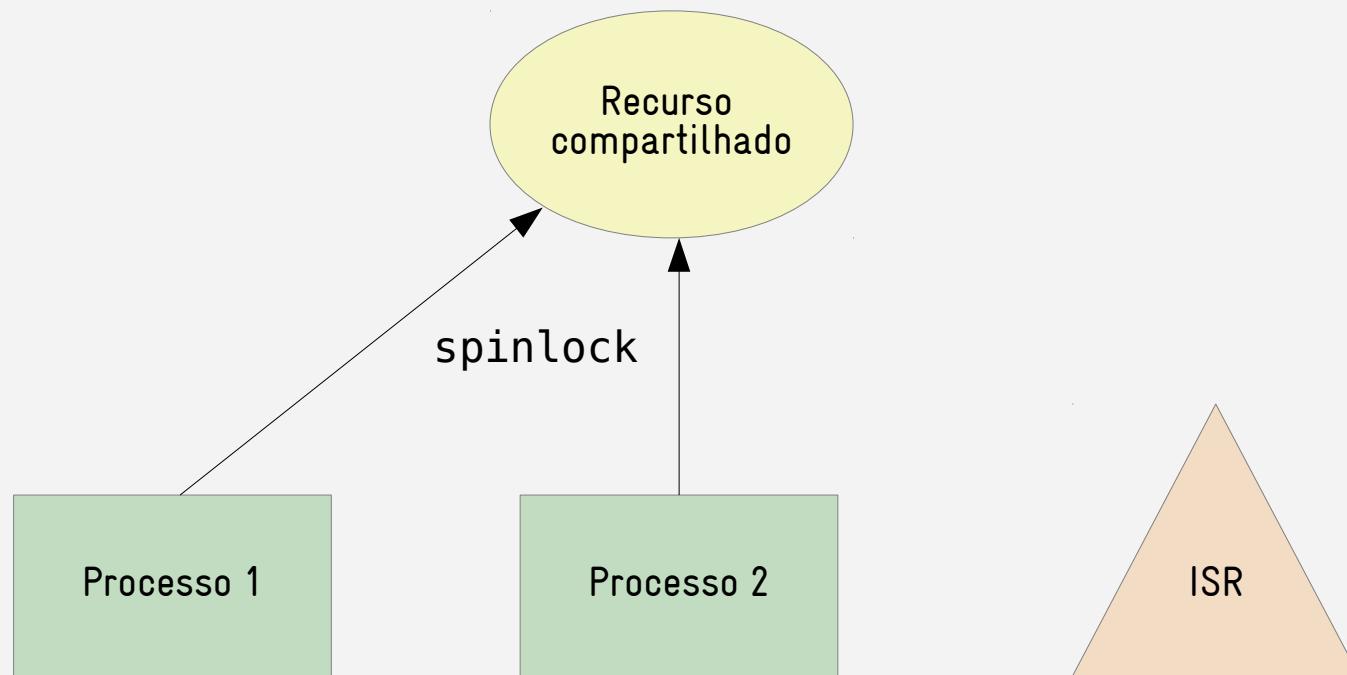
# CASO 3

PREEMPÇÃO = 1  
SMP = 1





# CASO 4





# RESUMO

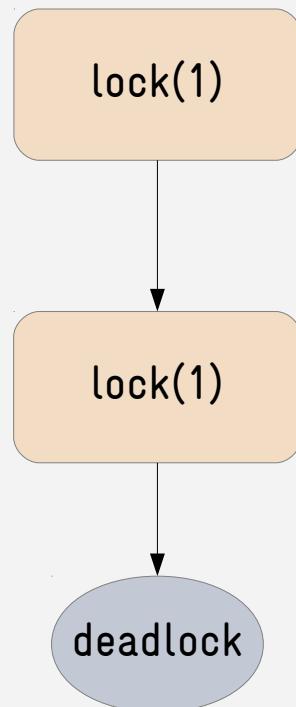
- ✗ Compartilhando dados entre processos:
  - ✗ Como padrão, use mutex.
  - ✗ Caso a performance seja importante, avalie a possibilidade do uso de spinlocks, mas lembre-se de que spinlocks aumentam o tempo de latência do sistema.
- ✗ Compartilhando dados entre processos e interrupções:
  - ✗ Evite, se possível!
  - ✗ Use spinlocks.
- ✗ Lembre-se: desenvolva sempre levando em consideração que ele pode rodar em um sistema SMP e com a preempção do kernel habilitada!



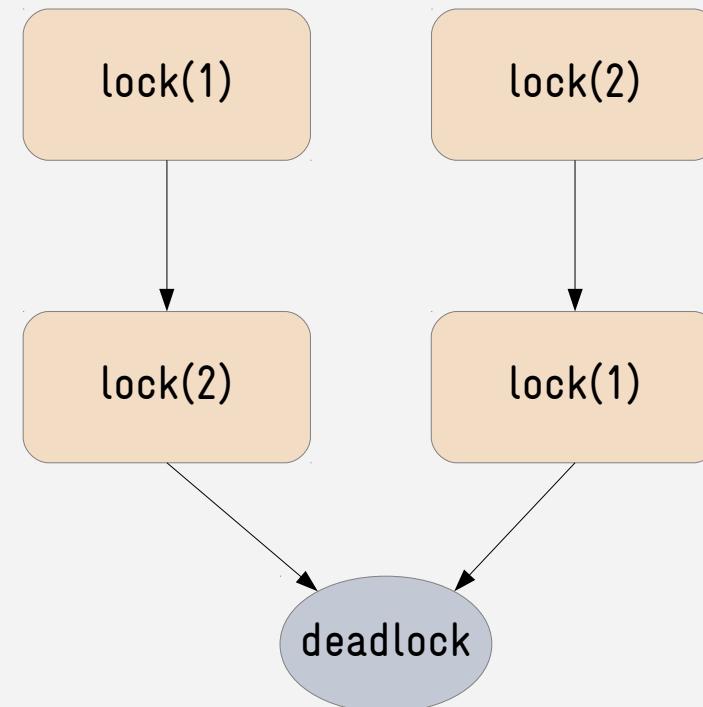


# DEADLOCKS

SITUAÇÃO 1



SITUAÇÃO 2





# READER-WRITE SPINLOCKS

- ✗ Quando um processo pretende escrever em um recurso compartilhado, é importante que nenhum outro processo tenha acesso ao recurso compartilhado.
- ✗ Porém, quando um processo pretende ler de um recurso compartilhado, não existe problema se outro processo também tentar ler deste mesmo recurso compartilhado.
- ✗ Neste caso podemos ter um lock exclusivo para escritas e um lock compartilhado para leituras.
- ✗ O Linux implementa esta funcionalidade através de spinlocks do tipo Reader-Writer.





# READER-WRITE SPINLOCKS API

```
#include <linux/rwlock.h>

/* initialize */
DEFINE_LOCK(mydriver_rwlock);
void rwlock_init(rwlock_t *lock);

/* lock on read */
void read_lock(rwlock_t *lock);
void read_unlock(rwlock_t *lock);

/* lock on write */
void write_lock(rwlock_t *lock);
void write_unlock(rwlock_t *lock);
```





# OUTROS MECANISMOS

- ✗ **Sequencial locks:** também chamado de seqlock, é um mecanismo parecido com o Reader-Write lock, mas prioriza escritas ao invés de leituras. Mais informações em `<linux/seqlock.h>`.
- ✗ **RCU (Read Copy Update):** é um outro mecanismo parecido com o Reader-Write lock, com as rotinas de leitura muito mais rápidas, porém com a rotina de escrita mais lenta. Mais informações em `<linux/rcupdate.h>`.





# COMPLETION VARIABLE

- ✗ A variável `completion` é um exemplo de uso de semáforos como mecanismo de comunicação de eventos (notificação) entre tarefas.
- ✗ Uma tarefa espera (dorme) na variável `completion` enquanto outra tarefa realiza algum processamento.
- ✗ Quando a outra tarefa finalizar o processamento, ela utiliza a variável `completion` para acordar qualquer outra tarefa que esteja esperando.





# COMPLETION VARIABLE API

```
#include <linux/completion.h>

/* initialize completion variable */
DECLARE_COMPLETION(x);
void init_completion(struct completion *x);

/* wait for completion variable */
void wait_for_completion(struct completion *x);

/* wait for completion variable - return on signal */
int wait_for_completion_interruptible(struct completion *x);

/* sinalize completion variable */
void complete(struct completion *x);

/* sinalize completion variable on all threads */
void complete_all(struct completion *x);
```





Embedded Labworks

# LABORATÓRIO

Usando mutex



Embedded Labworks

# Linux device drivers

Kernel Debugging



# KERNEL DEBUGGING

- ✗ Depurar o kernel é bem mais difícil quando comparado ao processo de depuração de aplicações.
- ✗ Um bug em uma aplicação derruba apenas a aplicação, já um bug no kernel pode derrubar todo o sistema!
- ✗ O processo de depuração do kernel exige conhecimentos do sistema operacional e das diversas técnicas de análise que estudaremos nesta seção do treinamento.





# DEBUGGING COM MENSAGENS

- ✗ A função `printk()`, definida em `<linux/printk.h>`, é a responsável por imprimir mensagens no kernel, podendo ser chamada tanto em contexto de processo quanto em contexto de interrupção.
- ✗ Ela tem o mesmo protótipo da função `printf()` usada em user space:  
`int printk(const char *s, ...);`
- ✗ Todas as mensagens do kernel são armazenadas em um buffer circular (ring buffer), cujo tamanho pode ser definido em tempo de compilação na opção `CONFIG_LOG_BUF_SHIFT` ou em tempo de execução no parâmetro de boot `log_buf_len`.
- ✗ As mensagens são normalmente exibidas na console e podem ser emitidas a qualquer momento com a ferramenta `dmesg`.





# NÍVEIS DE LOG

- Por padrão, deve-se passar uma macro indicando a prioridade da mensagem:

```
printf(KERN_WARNING "warning: skipping physical page 0\n");
```

- O kernel define os seguintes níveis de prioridade das mensagens de log:

0 (KERN_EMERG)	system is unusable
1 (KERN_ALERT)	action must be taken immediately
2 (KERN_CRIT)	critical conditions
3 (KERN_ERR)	error conditions
4 (KERN_WARNING)	warning conditions
5 (KERN_NOTICE)	normal but significant condition
6 (KERN_INFO)	informational
7 (KERN_DEBUG)	debug-level messages

- Caso a chamada à função printf() não passe o nível de log, será usado como padrão o nível de log definido na configuração do kernel CONFIG\_DEFAULT\_MESSAGE\_LOGLEVEL (normalmente KERN\_WARNING).





# NÍVEIS DE LOG (cont.)

- ✗ Todas as mensagens são armazenadas, mas por padrão apenas as mensagens com prioridade menor que debug são exibidas na console.

```
#define DEFAULT_CONSOLE_LOGLEVEL 7 /* em kernel/printk.c */
```

- ✗ Esta configuração pode ser alterada no boot passando o parâmetro loglevel para o kernel ou em tempo de execução no arquivo /proc/sys/kernel/printk.
- ✗ Outros parâmetros de boot que podem alterar o comportamento das mensagens de log do kernel:
  - ✗ **debug**: Habilita o nível 7 (KERN\_DEBUG) de mensagens de log.
  - ✗ **ignore\_loglevel**: Habilita todos os níveis de mensagens (também pode ser habilitado em /sys/module/printk/parameters/ignore\_loglevel).
  - ✗ **quiet**: Configura o nível de log como 4 (KERN\_WARNING).





# NOVAS FUNÇÕES

- ✗ Hoje temos outras opções para exibir mensagens de log no kernel, e não é mais comum o uso da função `printk()` para debugging.
- ✗ Pode-se usar a família de funções `pr_*`(), definida em `<linux/printk.h>`:  
`pr_emerg()`, `pr_alert()`, `pr_crit()`, `pr_err()`, `pr_warning()`,  
`pr_notice()`, `pr_info()`, `pr_cont()`, `pr_debug()`.
- ✗ Ou a família de funções `dev_*`(), definida em `<linux/device.h>`. Estas funções recebem como argumento um ponteiro para uma estrutura do tipo `device`.  
`dev_emerg()`, `dev_alert()`, `dev_crit()`, `dev_err()`,  
`dev_warning()`, `dev_notice()`, `dev_info()`, `dev_dbg()`.





# MENSAGENS DE DEBUG

- As funções `pr_debug()` e `dev_dbg()` serão compiladas apenas se a macro `DEBUG` for definida.
- Isso pode ser feito no makefile do arquivo que se deseja habilitar as mensagens de debug.

```
CFLAGS_[filename].o := -DDEBUG
```

- Quando o kernel é compilado com a opção `CONFIG_DYNAMIC_DEBUG`, as mensagens de debug podem ser habilitadas por arquivo, função ou até por linha de código! Veja a documentação do kernel em `Documentation/dynamic-debug-howto.txt`.





# DEBUGFS

- ✗ O debugfs é um sistema de arquivos virtual que exporta informações de debug do kernel para o espaço de usuário.
- ✗ Para utilizá-lo, é necessário habilitar a opção CONFIG\_DEBUG\_FS na configuração do kernel em Kernel hacking → Debug Filesystem.
- ✗ Depois é só montar o debugfs:

```
# mount -t debugfs none /sys/kernel/debug
# ls /sys/kernel/debug/
asoc      bluetooth hid      mmc1
bdi       gpio        mmc0    usb
```

- ✗ A API esta documentada do DocBook do kernel:

<http://free-electrons.com/kerneldoc/latest/DocBook/filesystems/index.html>





# DEBUGFS API

```
#include <linux/debugfs.h>

/* create a sub-directory for your driver */
struct dentry *debugfs_create_dir(const char *name,
                                 struct dentry *parent);

/* expose an integer as a file in debugfs, where u is for
   decimal representation and x for hexadecimal
   representation */
struct dentry *debugfs_create_{u,x}{8,16,32}(
    const char *name, mode_t mode,
    struct dentry *parent, u8 *value);

/* expose a binary blob as a file in debugfs */
struct dentry *debugfs_create_blob(const char *name,
                                   mode_t mode, struct dentry *parent,
                                   struct debugfs_blob_wrapper *blob);
```





# OUTROS MECANISMOS

- ✗ Alguns mecanismos de debugging foram bastante usados, mas agora seu uso não é mais comum:
  - ✗ Usar comandos especiais de `ioctl()` para debugging (use `debugfs`).
  - ✗ Usar entradas especiais no sistema de arquivos `proc` (use `debugfs`).
  - ✗ Usar entradas especiais no `sysfs` (use `debugfs`).
  - ✗ Usar `printf()` (use a família de funções `pr_*`() ou `dev_*`()).





# MAGIC SYSRQ KEY

- ✗ A Magic Sysrq Key é uma combinação de teclas manipulada pelo kernel, que pode ser usada como ferramenta de análise, debug e recuperação do kernel:
  - ✗ No PC: [Alt] + [SysRq] + <caractere>. Em alguns teclados, a tecla SysRq é a tecla Print Screen.
  - ✗ Pela console serial: <break> + <caractere>.
  - ✗ Em todas plataformas: escrever o caractere diretamente no arquivo /proc/sysrq-trigger.
- ✗ Deve ser habilitada no kernel através da opção CONFIG\_MAGIC\_SYSRQ, e pode ser habilitada ou desabilitada em tempo de execução através do arquivo /proc/sys/kernel/sysrq.





# MAGIC SYSRQ KEY (cont.)

- ✗ Exemplos de comandos:
  - ✗ **b**: Reinicia o sistema automaticamente.
  - ✗ **e**: Envia o sinal SIGTERM para todos os processos (menos o init).
  - ✗ **t**: Mostra o stack (dump) de todos os processos em execução.
  - ✗ **w**: Mostra o stack (dump) dos processos bloqueados (dormindo).
  - ✗ **c**: Força um crash do kernel desreferenciando um ponteiro nulo.
- ✗ Mais informações e comandos disponíveis na documentação do kernel em Documentation/sysrq.txt.





# KERNEL OOPS

- ✗ Um oops é um mecanismo de comunicação do kernel para notificar o usuário que um erro aconteceu.
- ✗ Este erro pode acontecer por diversos motivos, como por exemplo acesso ilegal à regiões de memória ou execução de instruções inválidas.
- ✗ Quando acontece um oops o kernel emite uma mensagem na console, exibindo o status atual do sistema no momento em que aconteceu o problema, incluindo um dump dos registradores e um backtrace do stack.
- ✗ Após o oops, o kernel irá tentar se recuperar e resumir a execução, mas dependendo do erro, nem sempre isso é possível.
- ✗ Nestes casos, o kernel pode travar após o oops (kernel panic).





# KERNEL OOPS (EXEMPLO)

```
Internal error: Oops: 817 [#1] PREEMPT
pc : [<80231bb8>]    lr : [<80231fe8>]    psr: 60000093
sp : df4f5f18  ip : df4f5e88  fp : 00000002
r10: 00000000  r9 : 00000000  r8 : 00000007
r7 : 60000013  r6 : 808214c8  r5 : 00000000  r4 : 00000063
r3 : 00000001  r2 : 00000000  r1 : 00000000  r0 : 00000063
Stack: (0xdf4f5f18 to 0xdf4f6000)
5f00:                                     00000002 802320b4
5f20: df4d9400 00000002 000a8d48 00000000 df4f5f80 802320e4 df1f7e00 800fea88
5f40: 00000002 df4d9400 000a8d48 df4f5f80 00000000 00000000 7ec2271c 800bfff0
[<80231bb8>] (sysrq_handle_crash+0x14/0x20) from [<80231fe8>]
(__handle_sysrq+0xdc/0x1a8)
[<80231fe8>] (__handle_sysrq+0xdc/0x1a8) from [<802320e4>]
(write_sysrq_trigger+0x30/0x38)
[<802320e4>] (write_sysrq_trigger+0x30/0x38) from [<800fea88>]
(proc_reg_write+0xb4/0xc8)
[<800fea88>] (proc_reg_write+0xb4/0xc8) from [<800bfff0>]
(vfs_write+0xac/0x154)
[<800bfff0>] (vfs_write+0xac/0x154) from [<800c0144>] (sys_write+0x3c/0x68)
[<800c0144>] (sys_write+0x3c/0x68) from [<80030f80>]
(ret_fast_syscall+0x0/0x30)
Kernel panic - not syncing: Fatal exception
```





# CONFIGURANDO O OOPS

- ✗ No exemplo anterior, os endereços estavam convertidos para os nomes das respectivas funções, facilitando o processo de debugging.
- ✗ Para que esta funcionalidade esteja disponível, habilite a opção `CONFIG_KALLSYMS` na configuração do kernel
- ✗ Em versões mais antigas, o kernel só exibia os endereços e era necessário usar um programa chamado `ksymoops` para fazer a análise.





# NOTIFICANDO PROBLEMAS

- ✗ Se necessário, é possível gerar um kernel oops através das macros BUG() ou BUG\_ON(). Exemplos:

```
if (erro) BUG();
BUG_ON(erro);
```
- ✗ Se quiser gerar direto um kernel panic, use a função panic():

```
if (erro_geral)
    panic("Alguma coisa terrível aconteceu!");
```
- ✗ Já se o que você quer é só imprimir um backtrace do stack, use a função dump\_stack().





# OUTRAS OPÇÕES DE DEBUG

- ✗ O Linux possui diversas funcionalidades de debugging integradas ao kernel.
- ✗ Para usá-las basta habilitar a opção CONFIG\_DEBUG\_KERNEL no menu de configuração, e depois habilitar a funcionalidade de debug desejada no menu "Kernel hacking". Alguns exemplos:
  - ✗ CONFIG\_DEBUG\_MUTEXES: habilita checagem do uso de mutexes.
  - ✗ CONFIG\_DEBUG\_SPINLOCK: habilita checagem de spinlocks.
  - ✗ CONFIG\_SLUB\_DEBUG\_ON: checagem na API de alocação de memória.
  - ✗ CONFIG\_DEBUG\_KMEMLEAK: habilita checagem de memory leak.





# KDB

- ✗ O kdb é um debugger que permite examinar a memória e as estruturas de dados do kernel enquanto o sistema está em execução.
- ✗ Durante um bom tempo era disponibilizado através de um conjunto de patches, mas foi integrado ao mainline na versão 2.6.35.
- ✗ Fornece uma interface de linha de comandos, permitindo realizar operações típicas de um debugger como step, stop, run, colocar breakpoints, disassembly de instruções, etc.
- ✗ A tecla Pause pode ser usada para entrar no modo de debug.
- ✗ Não trabalha no nível do código-fonte, apenas no nível de instruções assembly!





# GDB

- ✗ O GDB (GNU Debugger) é o debugger padrão do projeto GNU, disponível para diversas arquiteturas.  
<http://www.gnu.org/software/gdb/>
- ✗ Interface via console, mas com diversos frontends disponíveis (Eclipse, DDD, GDB/Insight, etc).
- ✗ Para um processo de depuração remota, usa-se um cliente GDB no host e um servidor GDB no target.





# KGDB

- ✗ O KGDB é uma implementação do GDB server no kernel Linux.  
<http://free-electrons.com/kerneldoc/latest/DocBook/kgdb/>
- ✗ Suporta comunicação via porta serial (disponível no mainline) e rede (necessário patch).
- ✗ Esta funcionalidade foi incluída no kernel deste a versão 2.6.26 (x86 e sparc) e 2.6.27 (arm, mips e ppc).
- ✗ Possibilita controle total sobre o processo de execução do kernel Linux no target, incluindo leitura e escrita em memória, execução passo-a-passo e até breakpoints em rotinas de tratamento de interrupção!





# KGDB: CONFIGURANDO O KERNEL

- ✗ Para usar o KGDB, você precisa habilitar e recompilar o kernel com as seguintes opções:
  - ✗ `CONFIG_KGDB`: Habilita o KGDB.
  - ✗ `CONFIG_KGDB_SERIAL_CONSOLE`: Habilita o driver de I/O para a comunicação entre o host e o target via console serial.
  - ✗ `CONFIG_DEBUG_INFO`: Para compilar o kernel com símbolos de debugging.
  - ✗ `CONFIG_FRAME_POINTER`: Ajuda a produzir backtraces de stack mais confiáveis.





# KGDB: CONFIGURANDO O TARGET

- Para configurar os parâmetros de comunicação serial do KGDB, passe o parâmetro kgdboc para o kernel no boot.

```
kgdboc=ttyS0,115200
```

- Para colocar o kernel no modo de depuração no boot, basta passar o parâmetro kgdbwait.
- Se você quiser colocar o kernel no modo de depuração durante a execução do sistema, envie um comando de SysRq:

```
# echo g > /proc/sysrq-trigger
```





# KGDB: DEPURANDO NO HOST

- × Na máquina de desenvolvimento, inicie o cliente GDB passando a imagem ELF do kernel (vmlinux):

```
$ arm-linux-gdb vmlinux
```

- × Lembre-se de usar o gdb do seu cross-compiling toolchain.

- × Depois configure a conexão serial:

```
(gdb) set remotebaud 115200  
(gdb) target remote /dev/ttyUSB0
```

- × Se o servidor GDB estiver em execução no target, o processo de depuração do kernel deverá ser iniciado automaticamente.





# OTIMIZAÇÃO

- × Em alguns casos, otimizações do compilador podem prejudicar o processo de depuração interativa.
- × Neste caso, você pode desabilitar as otimizações incluindo a variável abaixo em um Makefile do kernel:

`EXTRA_CFLAGS += -O0`

- × Você pode também desabilitar as otimizações de um arquivo específico do kernel:

`CFLAGS_[filename] += -O0`





# DEBUGGING COM JTAG

- ✗ JTAG é uma interface física que permite acesso à modulos de debug integrados ao core da CPU, onde podemos colocá-la em halt, inspecionar registradores, memória, colocar breakpoints, etc.
- ✗ É uma interface de debugging necessária quando trabalhamos no porte do bootloader ou do kernel para determinada plataforma (board bring-up).





# DEBUGGING COM JTAG (cont.)

- ✗ Precisamos basicamente de 4 componentes para trabalhar com debugging via JTAG:
  - ✗ Hardware (kit de desenvolvimento) com suporte à JTAG.
  - ✗ Adaptador JTAG (Ex: Flyswatter).
  - ✗ Debugger (Ex: GDB).
  - ✗ Software de interface entre o adaptador JTAG e o debugger (Ex: OpenOCD).
- ✗ Mais informações em:  
<http://sergioprado.org/linux-kernel-debugging-com-jtag/>





# KDUMP

- ✗ O Kdump é um mecanismo capaz de executar um novo kernel para capturar o dump de um kernel travado (Ex: kernel panic).
- ✗ Requer um kernel com algumas funcionalidades habilitadas, como a chamada de sistema kexec e o sistema de arquivos sysfs.
- ✗ A habilitação da funcionalidade deve ser feita com o pacote kexec-tools.
- ✗ Possui suporte em algumas arquiteturas, incluindo x86, PPC e ARM.
- ✗ Para mais detalhes, consulte a documentação do kdump em Documentation/kdump/kdump.txt.





# TRACING

- ✗ Tracing é uma forma especializada de log, normalmente utilizada para depuração, capaz de colher dados sobre o estado e o funcionamento de um programa durante sua execução.
- ✗ É realizado através da inclusão de tracepoints (probes) no código-fonte para instrumentar o software em tempo de execução.
- ✗ A documentação dos mecanismos de tracing suportados pelo Linux está disponível no código fonte em Documentation/trace/.





# KERNEL PROBES

- ✗ Kernel probes são um mecanismo de tracing e instrumentação do kernel Linux.
- ✗ Com esta funcionalidade é possível extrair informações do kernel e até alterar seu comportamento (aplicar patches) em tempo de execução, sem precisar reiniciar o sistema!
- ✗ A implementação genérica de kernel probes é chamada de kprobes, e pode ser usada para instrumentar qualquer instrução do kernel.
- ✗ Os kprobes possuem ainda duas variantes:
  - ✗ jprobes: usados para instrumentar chamadas de função.
  - ✗ return probes: usados para instrumentar retornos de função.





## KERNEL PROBES (cont.)

- ✗ O princípio básico do kprobe é baseado no uso de interrupções de software.
- ✗ Para usar a infraestrutura de kprobes, você precisa desenvolver um módulo do kernel e gerenciar os pontos de instrumentação através de uma estrutura do tipo kprobe.
- ✗ A limitação do uso de kprobes está na dificuldade de associar o código-fonte com o binário gerado (otimizações do compilador, recursos da linguagem C como macros e funções inline, etc).
- ✗ Mais informações sobre kprobes em Documentation/kprobes.txt.





# SYSTEMTAP

- ✗ O SystemTap é uma infraestrutura de tracing que usa kprobes para facilitar o trabalho de instrumentação no kernel.  
<http://sourceware.org/systemtap/>
- ✗ Assim como os kprobes, ele também elimina a necessidade de modificar e recompilar o kernel para investigar um problema funcional ou de performance.
- ✗ Utiliza uma linguagem simples de script (diversos exemplos disponíveis na Internet).
- ✗ Tutorial e exemplos do SystemTap disponíveis no ambiente de laboratório do treinamento em /opt/labs/docs/guides/systemtap.pdf.





# SYSTEMTAP (EXEMPLO)

```
# strace-open.stp

probe syscall.open
{
    printf ("%s(%d) open (%s)\n", execname(), pid(), argstr)
}

probe timer.ms(4000) # after 4 seconds
{
    exit ()
}
```





# SYSTEMTAP (TESTANDO EXEMPLO)

```
# stap strace-open.stp
vmware-guestd(2206) open ("/etc/redhat-release", 0_RDONLY)
halld(2360) open ("/dev/hdc", 0_RDONLY|0_EXCL|0_NONBLOCK)
halld(2360) open ("/dev/hdc", 0_RDONLY|0_EXCL|0_NONBLOCK)
halld(2360) open ("/dev/hdc", 0_RDONLY|0_EXCL|0_NONBLOCK)
df(3433) open ("/etc/ld.so.cache", 0_RDONLY)
df(3433) open ("/lib/tls/libc.so.6", 0_RDONLY)
df(3433) open ("/etc/mtab", 0_RDONLY)
halld(2360) open ("/dev/hdc", 0_RDONLY|0_EXCL|0_NONBLOCK)
...
...
```





# KERNEL TRACEPOINTS

- ✗ Kprobes possibilitam instrumentar qualquer instrução do kernel, mas adicionam um overhead ao processamento pelo fato de trabalharem com interrupções de software.
- ✗ Uma solução mais leve são os tracepoints, que adicionam ponto de trace estáticos no kernel, definidos em tempo de compilação.
- ✗ A implementação é bem simples: se uma função de probe estiver associada à determinado tracepoint, esta função será chamada.
- ✗ Mais informações sobre tracepoints nos fontes do kernel em Documentation/trace/tracepoints.txt.





# LTTng

- ✗ Toolkit que permite coletar e analisar informações de tracing do kernel, baseado em kernel tracepoints.
- ✗ Até então (Linux 3.6), é necessário aplicar patches no kernel para utilizar esta ferramenta de tracing.
- ✗ Capacidade de gerar timestamps extremamente precisos e com muito pouco overhead.
- ✗ Mais informações na documentação do projeto:  
<http://lttng.org/documentation>





# FTRACE

- ✗ Ftrace é um framework genérico de tracing no kernel, e pode ser usado para identificação de bugs e problemas de latência.
- ✗ Utiliza alguns mecanismos de probe providos pelo kernel, incluindo tracepoints e kprobes.
- ✗ Interface com o usuário através do sistema de arquivos virtual debugfs.
- ✗ Documentação disponível no código fonte do kernel em Documentation/trace/.





# USANDO O FTRACE

```
# mount -t debugfs none /sys/kernel/debug
# cd /sys/kernel/debug/tracing/
# cat available_tracers
blk      function_graph      wakeup_rt      wakeup      preemptirqsoff
preemptoff irqsoff function nop
```





# FUNCTION TRACER

```
# echo function > current_tracer
# cat trace
[...]
<idle>-0      [002] d..2 11933.473505: ktime_get <-cpuidle_enter_state
<idle>-0      [002] d..2 11933.473514: clocksource_mmio_readl_up <-ktime_get
<idle>-0      [002] ...2 11933.473520: ns_to_timeval <-cpuidle_enter_state
<idle>-0      [002] ...2 11933.473525: ns_to_timespec <-ns_to_timeval
<idle>-0      [002] ...2 11933.473530: menu_reflect <-cpuidle_idle_call
<idle>-0      [002] ...2 11933.473535: idle_notifier_call_chain <-arch_cpu_idle
<idle>-0      [002] ...2 11933.473539: atomic_notifier_call_chain <-idle_notifier_call_chain
<idle>-0      [002] ...2 11933.473544: __atomic_notifier_call_chain <-atomic_notifier_call_chain
<idle>-0      [002] ...2 11933.473548: __rcu_read_lock <-__atomic_notifier_call_chain
<idle>-0      [002] ...2 11933.473553: notifier_call_chain <-__atomic_notifier_call_chain
<idle>-0      [002] ...2 11933.473557: cpufreq_interactive_idle_notifier <-notifier_call_chain
<idle>-0      [002] ...2 11933.473562: __rcu_read_unlock <-__atomic_notifier_call_chain
<idle>-0      [002] ...2 11933.473567: rcu_idle_exit <-cpu_startup_entry
<idle>-0      [002] d..2 11933.473572: rcu_eqs_exit_common.isra.41 <-rcu_idle_exit
[...]
```





# OPROFILE

- ✗ O principal objetivo de uma ferramenta de profiling é analisar a performance do sistema (consumo de ciclos de CPU).
- ✗ A ferramenta OProfile usa eventos de timer ou contadores de performance providos pelo processador para capturar dados em intervalos regulares.
- ✗ Mais informações sobre o projeto em:  
<http://oprofile.sourceforge.net/>





# OPROFILE (EXEMPLO)

```
$ opreport --exclude-dependent
CPU: PIII, speed 863.195 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not
halted) with a unit mask of 0x00 (No unit mask) count 50000
    450385 75.6634 cc1plus
      60213 10.1156 lyx
      29313  4.9245 XFree86
      11633  1.9543 as
      10204  1.7142 oprofiled
       7289  1.2245 vmlinu
       7066  1.1871 bash
       6417  1.0780 oprofile
       6397  1.0747 vim
       3027  0.5085 wineserver
       1165  0.1957 kdeinit
        832  0.1398 wine
...
...
```





Embedded Labworks

# LABORATÓRIO

Usando ferramentas de debugging



Embedded Labworks

# Linux device drivers

E agora?



# FILOSOFIA LINUX (1)

"Linux is evolution, not intelligent design."

Linus Torvalds





# FILOSOFIA LINUX (2)

"Talk is cheap. Show me the code."

Linus Torvalds





# LEIA A DOCUMENTAÇÃO

- ✗ O kernel tem uma extensa documentação no diretório Documentation/. Consulte sempre!
- ✗ O kernel também possui um conjunto de documentos em Documentation/DocBook, que podem ser compilados com um dos comandos abaixo:  
`$ make pdfdocs`  
`$ make htmldocs`





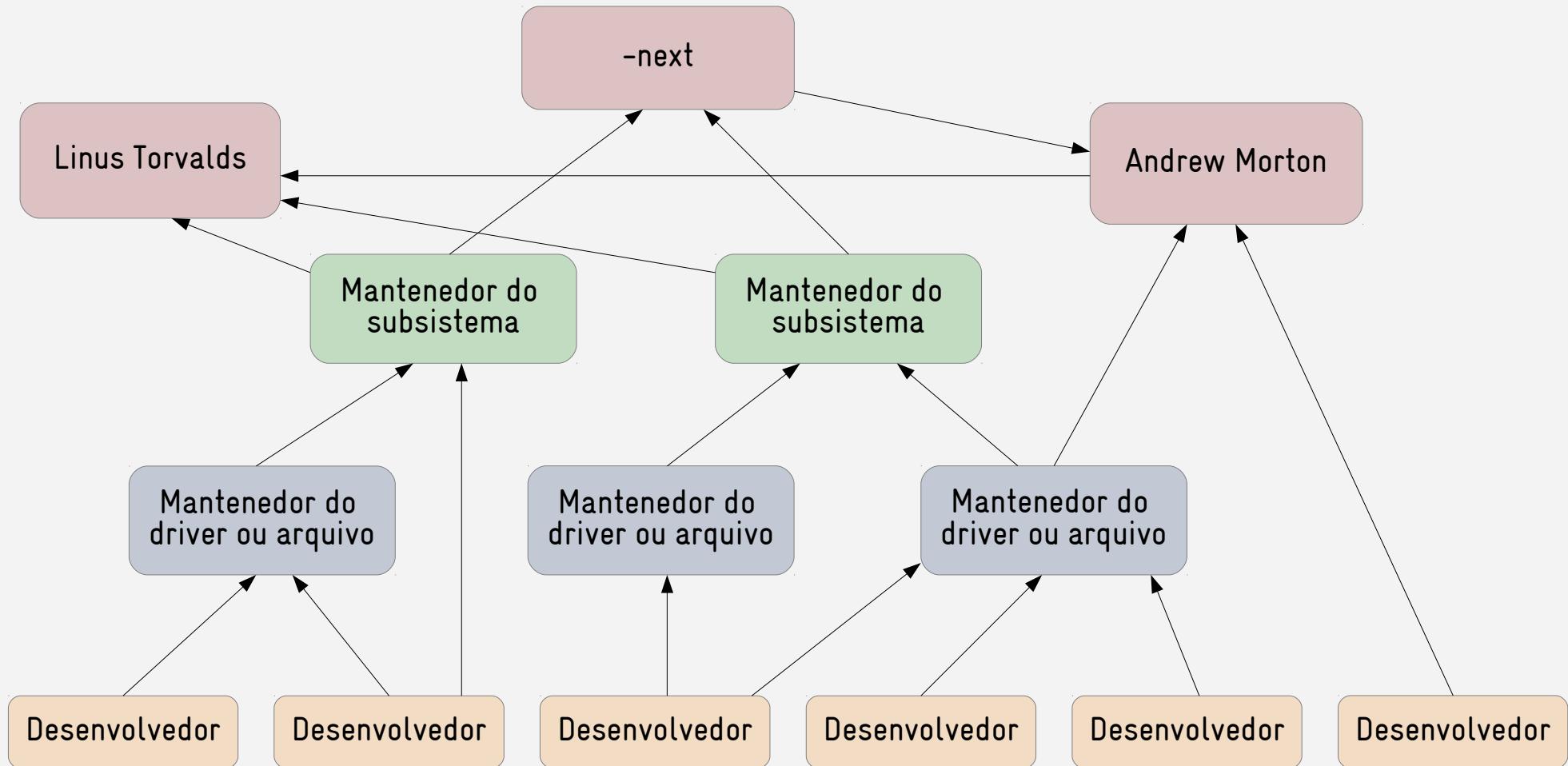
# LEIA A DOCUMENTAÇÃO (cont.)

- ✗ Alguns documentos interessantes:
  - ✗ Documentation/HOWTO: contém instruções gerais sobre como se tornar um desenvolvedor do kernel.
  - ✗ Documentation/ManagementStyle: descreve o estilo de gerenciamento do Linux e um pouco de sua cultura.
  - ✗ Documentation/stable\_api\_nonsense.txt: explica porque o Linux não tem uma interface (API) interna estável.





# PROCESSO DE DESENVOLVIMENTO





# PROCESSO DE DESENVOLVIMENTO (cont.)

- ✗ Boa documentação sobre o processo de desenvolvimento em Documentation/development-process/.
- ✗ A maioria das árvores git dos subsistemas do kernel encontram-se no link abaixo:  
<http://git.kernel.org/>
- ✗ Cada subsistema possui uma lista de discussão e um ou mais mantenedores.





# PROCESSO DE DESENVOLVIMENTO (cont.)

- ✗ O e-mail do mantenedor, bem como a lista de discussão do subsistema, estão disponíveis no arquivo MAINTAINERS no diretório principal do kernel.
- ✗ O script scripts/get\_maintainer.pl pode ajudar a identificar o e-mail da lista de discussão e o nome e e-mail dos mantenedores de determinado arquivo ou patch!





# GET MAINTAINER

```
$ ./scripts/get_maintainer.pl -f init/main.c
Rusty Russell <rusty@rustcorp.com.au> (commit_signer:4/20=20%)
Jim Cromie <jim.cromie@gmail.com> (commit_signer:3/20=15%)
Al Viro <viro@zeniv.linux.org.uk> (commit_signer:3/20=15%)
"H. Peter Anvin" <hpa@linux.intel.com> (commit_signer:3/20=15%)
Andrew Morton <akpm@linux-foundation.org> (commit_signer:2/20=10%)
linux-kernel@vger.kernel.org (open list)
```





# CONTRIBUINDO

- ✗ Todo o processo de contribuição acontece por e-mail através das listas de discussão.
- ✗ A principal lista de discussão é a LKML (Linux Kernel Mailing List):  
<http://lkml.org/>
- ✗ Mas muitos subsistemas possuem listas específicas.  
<http://vger.kernel.org/vger-lists.html>





## CONTRIBUINDO (cont.)

- ✗ Para colaborar, é só preparar o patch e enviar para a lista de discussão e para todos os mantenedores responsáveis pelo subsistema.
- ✗ O gerenciamento dos patches enviados é feito com a ferramenta patchwork.

<https://patchwork.kernel.org/>





# CONTRIBUINDO (cont.)

- ✗ Documentação sobre como submeter um patch:  
[Documentation/SubmittingPatches](#)
- ✗ Checklist adicional para submissão de patches:  
[Documentation/SubmitChecklist](#)
- ✗ Documentação específica sobre como submeter novos drivers:  
[Documentation/SubmittingDrivers](#)





# CONTRIBUINDO (cont.)

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
$ cd linux/
$ git branch smeagol
$ git checkout smeagol
# change Linux, and change the world!
$ scripts/checkpatch.pl -f <changed_files>
$ git commit -a
$ git format-patch master..smeagol
$ scripts/get_maintainer.pl <patch_file>
$ git send-email --to email@email.com --cc email@email.com <patch_file>
```





# AGUARDANDO APROVAÇÃO

- ✗ Ao enviar um patch, você pode esperar por críticas, comentários, pedidos para alterar o código ou pedidos para explicar determinado ponto da alteração.
- ✗ Você deve ser capaz de aceitar as críticas, analisar tecnicamente o problema, alterar o código ou explicar claramente os motivos pelos quais o código não foi alterado, e reenviar o patch.
- ✗ Se não receber resposta, espere alguns dias e reenvie o patch.





# AGUARDANDO APROVAÇÃO (cont.)

- ✗ “Publicly making fun of people is half the fun of open source programming. In fact, the real reason to eschew programming in closed environments is that you can't embarrass people in public.” Linus Torvalds.
- ✗ É um processo que exige muita paciência e perseverança!





# PROCURANDO AJUDA

- ✗ O site Kernel Newbies é um ponto de referência para buscar ajuda sobre o funcionamento interno do Kernel.  
<http://kernelnewbies.org>
- ✗ Possui uma lista de discussão e um canal IRC onde são discutidos qualquer tipo de assunto relacionado ao kernel, do mais básico ao mais avançado.
- ✗ Lembre-se de procurar nos arquivos da lista antes de postar alguma pergunta!





# REPORTANDO PROBLEMAS

- ✗ Um documento descrevendo como reportar bugs chamado REPORTING-BUGS encontra-se no diretório principal do kernel.
- ✗ O kernel também tem um bug tracker no link abaixo:  
<https://bugzilla.kernel.org>
- ✗ Você pode contribuir reportando ou corrigindo problemas!





# PARA COMEÇAR

- Kernel Janitors é um projeto do site Kernel Newbies para quem quiser aprender a desenvolver para o kernel revisando o código, fazendo limpezas, convertendo o uso de APIs e dando manutenção em código antigo.

<http://kernelnewbies.org/KernelJanitors>

- Linux Driver Project é um projeto criado para quem quiser ajudar no desenvolvimento de drivers.

<http://www.linuxdriverproject.org/>





# LINKS

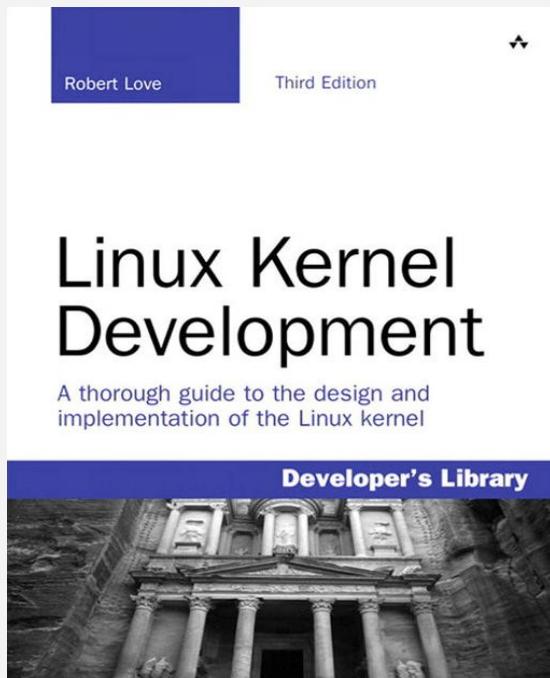
- ✗ Site do kernel Linux:  
<http://www.kernel.org>
- ✗ Linux kernel mailing list:  
<http://www.tux.org/lkml>
- ✗ Acompanhar as mudanças nas novas versões do kernel:  
<http://wiki.kernelnewbies.org/LinuxChanges>
- ✗ Notícias e novidades sobre o desenvolvimento do kernel:  
<http://lwn.net>





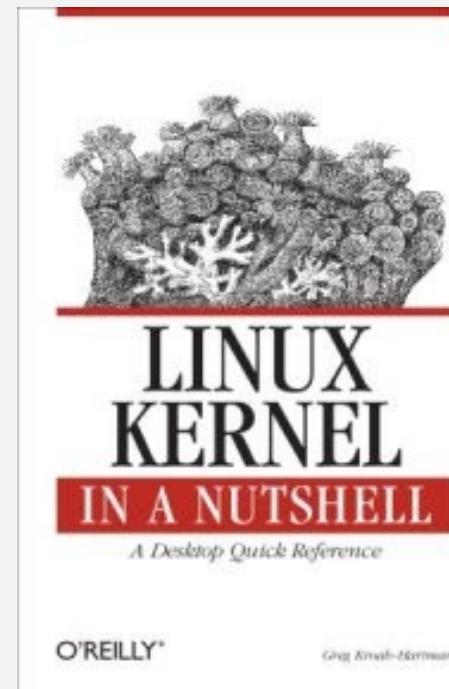
Embedded Labworks

# LIVROS LINUX KERNEL



Linux Kernel Development  
Robert Love

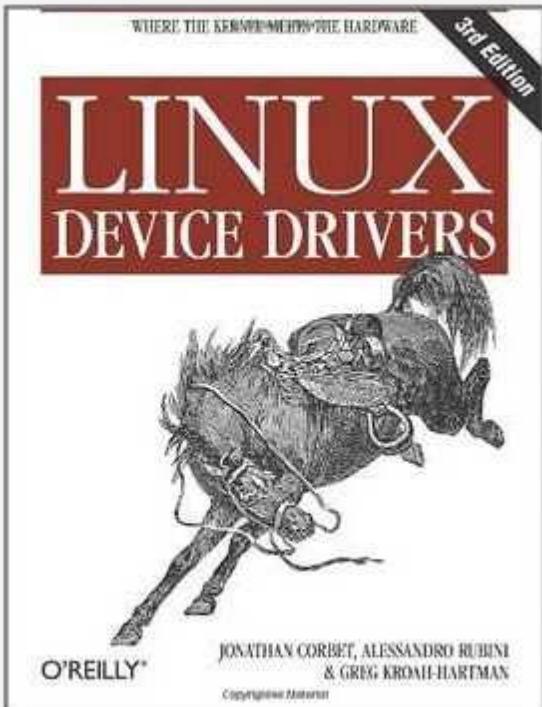
Linux Kernel in a Nutshell  
Greg Kroah-Hartman





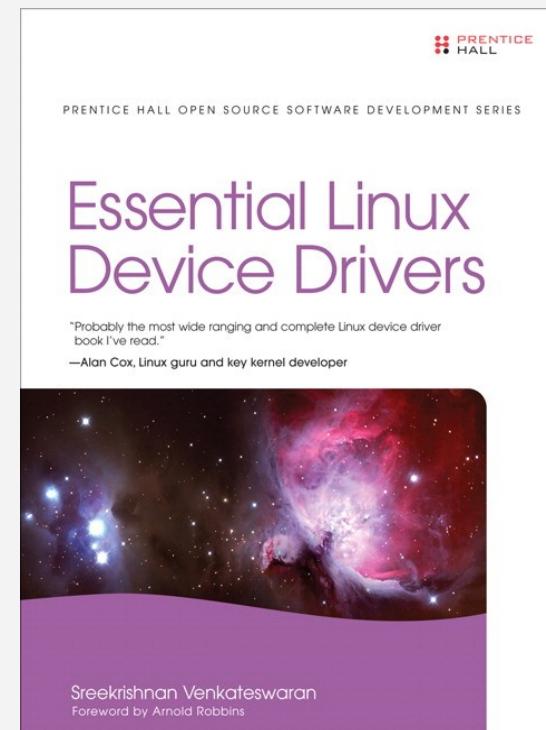
Embedded Labworks

# LIVROS LINUX DEVICE DRIVERS



Linux Device Drivers  
Jonathan Corbet & others

Essential Linux Device Drivers  
Sreekrishnan Venkateswaran





Embedded Labworks

# PERGUNTAS OU COMENTÁRIOS FINAIS?



# OBRIGADO!

E-mail  
Website

[sergio.prado@e-labworks.com](mailto:sergio.prado@e-labworks.com)  
<http://e-labworks.com>



Embedded Labworks

Por Sergio Prado. São Paulo, Abril de 2016

® Copyright Embedded Labworks 2004-2016. All rights reserved.