Programação no kernel Linux

Felipe W Damasio

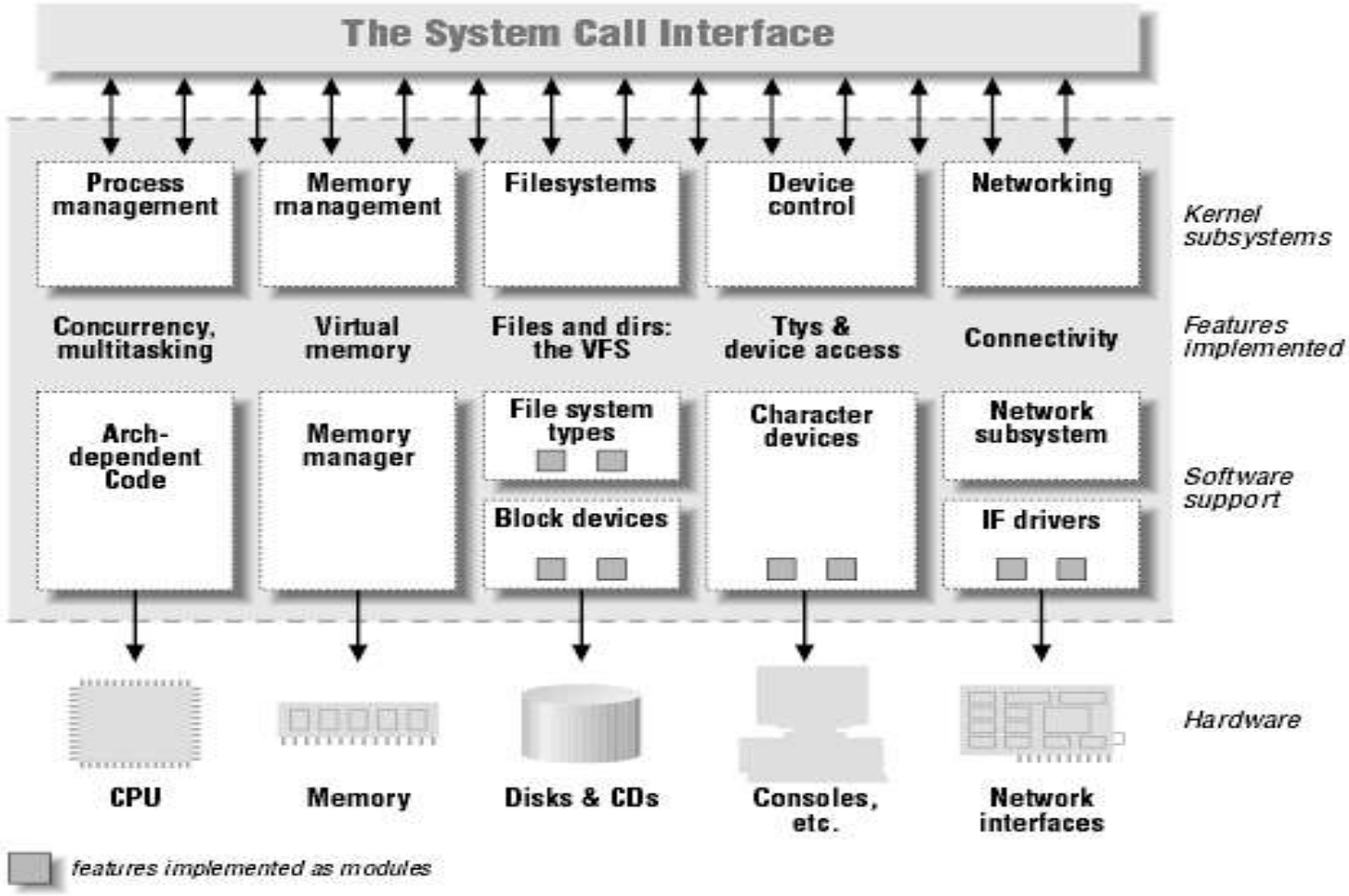
Visão Geral

- Linux é um kernel monolítico
- Símbolos exportados para todo o sistema
- Dificuldade de adição de APIs, hardware
  - Problemas com escalabilidade

Visão Geral

- Kernel modular
  - Permite a carga/ descarga sob demanda
- Facilita o desenvolvimento
- APIs são definidas por sub-sistema

Visão Geral


The System Call Interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs: the VFS
Ttys & device access
Connectivity
Features implemented
Arch-dependent Code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
CPU
Memory
Disks & CDs
Consoles, etc.
Network interfaces
Hardware
features implemented as modules

Obtenção

- Livremente disponível na Internet
- Acesso pode ser feito de vários repositórios
- Link para o projeto
  - http://www.kernel.org
- Existem outros links!
  - http://www.br.kernel.org/
- Repositório central de desenvolvimento
  - Os principais desenvolvedores tem contas neste servidor

2.6 vs 2.4

- 2.6 é mais fácil de escrever drivers;
- Menor latência;
- Maior escalabilidade;
- Escalonador mais “esperto”;
- Suporte a “goodies” para módulos;

Patchsets

- A árvore do Linus ou do mantenedor é considerada “oficial”
- Existem outras!
- MM: Andrew Morton responsável;
  - Finalidade: Testar patches para entrarem depois no 2.6 “vanilla” (oficial)
- AC: Alan Cox responsável;
  - Finalidade:
    - Suporte a novo hardware;
    - Modificações intrusivas no “kernel do kernel” (mm, net e escalonameto);

Patchsets

- CK: Con Kolivas responsável;
  - Finalidade:
    - Escalabilidade;
    - Perfomance;
    - Baixa latência;
    - “Responsiveness”;
- Onde obter?
- http://www.kernel.org/pub/linux/kernel/people/

Configuração

- cd <caminho do kernel/ >
- make menuconfig/ xconfig
- make clean bzImage modules modules_install;

- Vamos configurar um kernel?

Estrutura de diretórios

- arch
- crypto
- Documentation
- Drivers
  - net/
  - Block/
  - ...
- fs
- include
- init

Módulos

Pra que módulos?

Módulos

- Como se usa?
  - insmod
  - depmod
  - modprobe
  - lsmod
  - rmmod (e isso funciona?)

Módulos

- Escrevendo módulos:

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

static init oi_init(void) { printk (KERN_ALERT “Bom Dia!\n”); return 0; }
static void oi_exit (void) { printk (KERN_ALERT “Bom almoço!\n”); }

module_init(oi_init);
module_exit(oi_exit);
MODULE_LICENSE(“GPL”)
```

Módulos

- Como compila?
- No 2.4:

  gcc -D__KERNEL__ -DMODULE -c <arquivo.c> -Wall
- No 2.6:
  - Precisa ser criado um Makefile

Módulos

- Makefile:

```
MODULE:= <nome do módulo>.ko
obj-m:= <nome do módulo>.o
default:
    make -C /usr/src/linux SUBDIRS=`pwd` modules
```

- No shell:

```
$ make
```

Módulos
Parâmetros

- Passados em user-space para o insmod/ modprobe

modprobe meu_modulo.ko inteiros=1 string=”ola!”

insmod meu_modulo.ko inteiros=1 string=”ola!”

Módulos
Parâmetros

- Como se programa isso?

- module_param (nome, tipo, perms);
- modules_param_named (nome, variavel, tipo, perms);

Módulos
Parâmetros

- Tipos de dados padrão:
  - Byte
  - (u)short
  - (u)int
  - (u)long
  - charp
  - Bool

Módulos
Parâmetros

```
static int numero;

module_param (numero, int, 0);
```

Exercício

- Criem 1 módulo que recebe uma string como parâmetro e um número inteiro
  - Ela imprime essa string 'i' vezes.

Funcionamento do kernel

- Espaço de usuário
  - Enxerga arquivos
  - Nomes, diretórios...

- Como o kernel enxerga os dados?
  - Alguém conhece o **strace**?

  - strace ls /dev/dsp

Funcionamento do kernel

- O kernel lida com números
  - Driver tenta “registrar” um major/minor
  - Se conseguir, realiza comunicação sobre esse dispositivo
- Programa de usuário
  - ls -l /dev/dsp
  - open (“/dev/dsp”, O_RDONLY)
  - Fala com o driver que registrou para si os major/minor 14,3

Classes de dispositivos

- Character devices
  - Acessados byte a byte
- Block devices
  - Transferência em blocos (Hds)
- Ambos apresentam entradas no /dev

Major/ Minor Numbers

- Como o kernel sabe quando um dispositivo é de bloco ou de caractere?
  - Pelo tipo do dispositivo!
- Como o kernel sabe qual driver vai processar aquele dispositivo?
  - Pelo major number!

Classe de Dispositivos

- ls -l / dev/ psaux

crw-r----- 1 root root 10, 1 Dec 31 1969 psaux

Driver de caractere

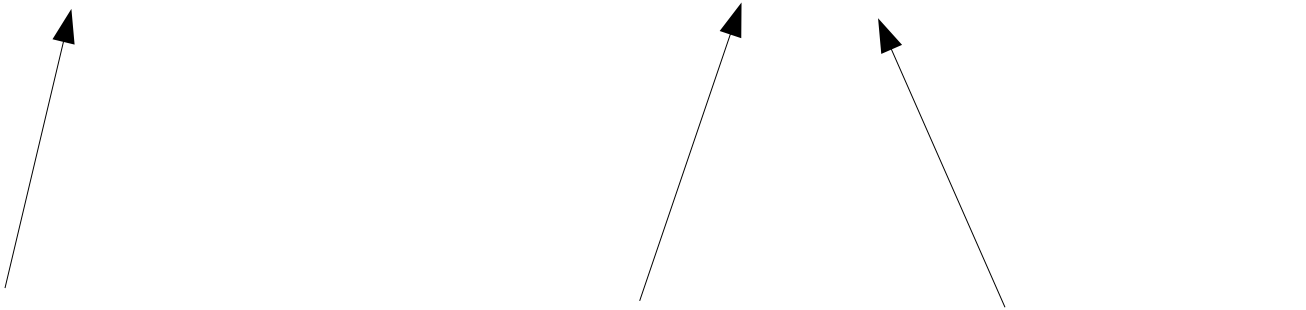
Major number

Minor number

Classe de Dispositivos

- Como se criam aquelas entradas no /dev?
- mknod /dev/psaux c 10 1

crw-r----- 1 root root 10, 1 Dec 31 1969 psaux



Driver de caractere

Major number

Minor number

Major/ Minor Numbers

- E pra que diabos serve o minor number?
- Quem responder ganha uma...

Major/ Minor Numbers

- E pra que diabos serve o minor number?
- Quem responder ganha uma...SALVA DE PALMAS!

Major/ Minor Numbers

- E pra que diabos serve o minor number?
- Para identificar qual dispositivo (manipulado pelo mesmo driver) está sendo executado no momento
- /usr/src/linux/Documentation/devices.txt

Classe de dispositivos

- Implementação de char drivers
  - Registro do major number

```
int register_chrdev(unsigned int major, const char * nome, struct file_operations *fops);
int unregister_chrdev (unsigned int major, const  char *nome);
```

- “name” é a entrada que fica em /proc/devices
  - Com major == 0, a geração dele é dinâmica.

Comunicação kernel/ user-space

- Ioctls: Funções “I/ O Control”
- Permite passagens de parâmetros em tempo de execução, ao invés de **apenas** em tempo de carga

- Ioctls estão disponíveis para ambos drivers de caractere e de bloco

Comunicação kernel/ user-space

- Implementação via file_operations

```
static struct file_operations driver_fops =
{
        .owner = THIS_MODULE,
        .ioctl = driver_ioctl,
        .open = driver_open,
        .release = driver_release,
};
```

Comunicação kernel/ user-space

- Implementação em user-space

```
int ioctl (int fd, int cmd, ...);
```

- O "..." sinaliza que pode receber outro parâmetro além do parâmetro inteiro. Existe uma extensão no GCC para permitir a passagem de estruturas para o kernel

Comunicação kernel/ user-space

- Tratamento no driver

```
int *(ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long
arg);
```

- 'inode' e 'file' são os valores do fd aberto e passado pela ioctl userspace.
- arg chega no kernel como unsigned long, mas pode ser feito um cast para estruturas (geralmente é feito assim).

Comunicação kernel/ user-space

- Funções auxiliares:

  int copy_to_user (void *destino, const void *origem, unsigned long count);

  int copy_from_user (void *destino, const void *origem, unsigned long count);

- Copia estruturas
- Checa os ponteiros (apenas o início)
- Retorna o número de bytes **não** copiados

Exercício

- Implementem um driver pra fazer um modem PCTel funcionar.

Exercício

- Implementem um driver pra fazer um modem PCTel funcionar.

- Desculpa, não resisti :)

Exercício

- Implementar a ioctl “0xcafe”, que troca o ID do dono do processo atual para o parâmetro passado.

- Dicas:
  - current->uid = id_que_o_usuario_passou;