

Fundamentos de Sistemas de Operação

Unix Windows NT Netware Mac OS DOS/V/MS Vax/VMS
Linux Solaris HP/UX AIX Mach Chorus

*Subsistema de I/O:
do hardware aos drivers*

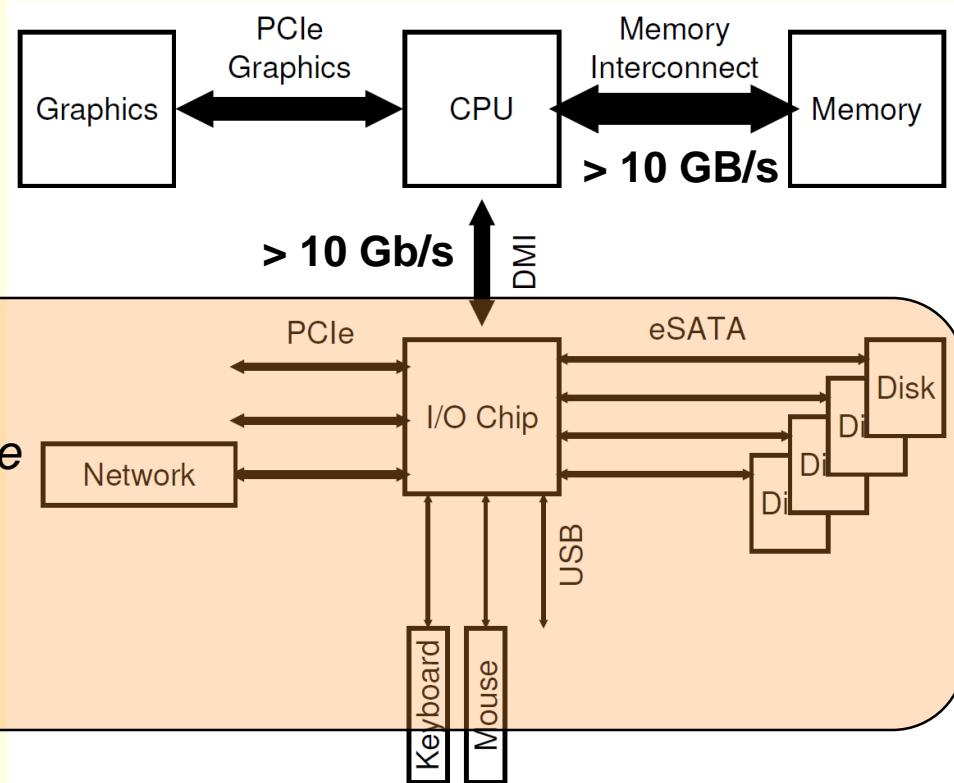
Fundamentos de Sistemas de Operação

Unix Windows NT Netware Mac OS DOS/VMS Vax/VMS
Linux Solaris HP/UX AIX Mach Chorus

*Subsistemas de I/O, Parte I:
o hardware*

Arquitectura de um computador

□ Arquitectura de um computador moderno



Um periférico canónico

Interface

Registers

Status

Command

Data

Internals

Micro-controller (CPU)

Memory (DRAM or SRAM or both)

Other Hardware-specific Chips

*Dispositivo
físico*

e.g., disco, impressora, scanner, mouse, teclado,
...

Os registos da interface

□ **Registo(s) de comandos**

- (*tipicamente*) Só de escrita, *recebe(m)* (do CPU) dados que representam comandos para desencadear acções no dispositivo. Ex: **RESET**, **INIT**, **READ**, **WRITE**, ...

□ **Registo de estado**

- (*tipicamente*) Só de leitura, são agregados de flags que representam os diferentes estados de funcionamento do dispositivo e do resultado de operações. Ex: **IDLE**, **BUSY**, **I/O ERROR**, **OK**, ...

□ **Registo(s) de dados**

- De leitura, escrita, ou leitura/escrita, dependendo do tipo de periférico, *usa(m)*-se para transferência de dados entre o periférico e o resto do sistema.

Interacção com periféricos

□ Arquitecturas Memory-Mapped I/O

- Os registos (do periférico) ocupam posições no espaço de endereços, em zonas em que não é instalada RAM (tipicamente em endereços “muito altos”) e mapeadas (as “páginas”) no espaço do kernel.
- A comunicação com o periféricos faz-se usando instruções do tipo *load/store* (ou `mov`, no x86*)
- Exemplos: arquitecturas IBM PowerPC, alguns periféricos no x86*

□ Arquitecturas I/O-Mapped I/O

- Os registos (do periférico) ocupam posições num espaço separado de endereços de I/O (muito menor que o espaço de endereços de RAM).
- A comunicação com o periféricos faz-se usando instruções próprias para I/O (e.g., Intel `in` e `out`)
- Exemplos: arquitecturas x86*

Protocolo de interacção com periféricos (1)

□ Polling / Busy waiting (espera activa)

```
while (STATUS == BUSY)  
    ;
```

Verificar se o dispositivo está livre lendo o reg. status

```
write data to DATA
```

Transferir dados para o dispositivo; comandá-lo
para realizar uma operação

```
write CommandCode to COMMAND
```

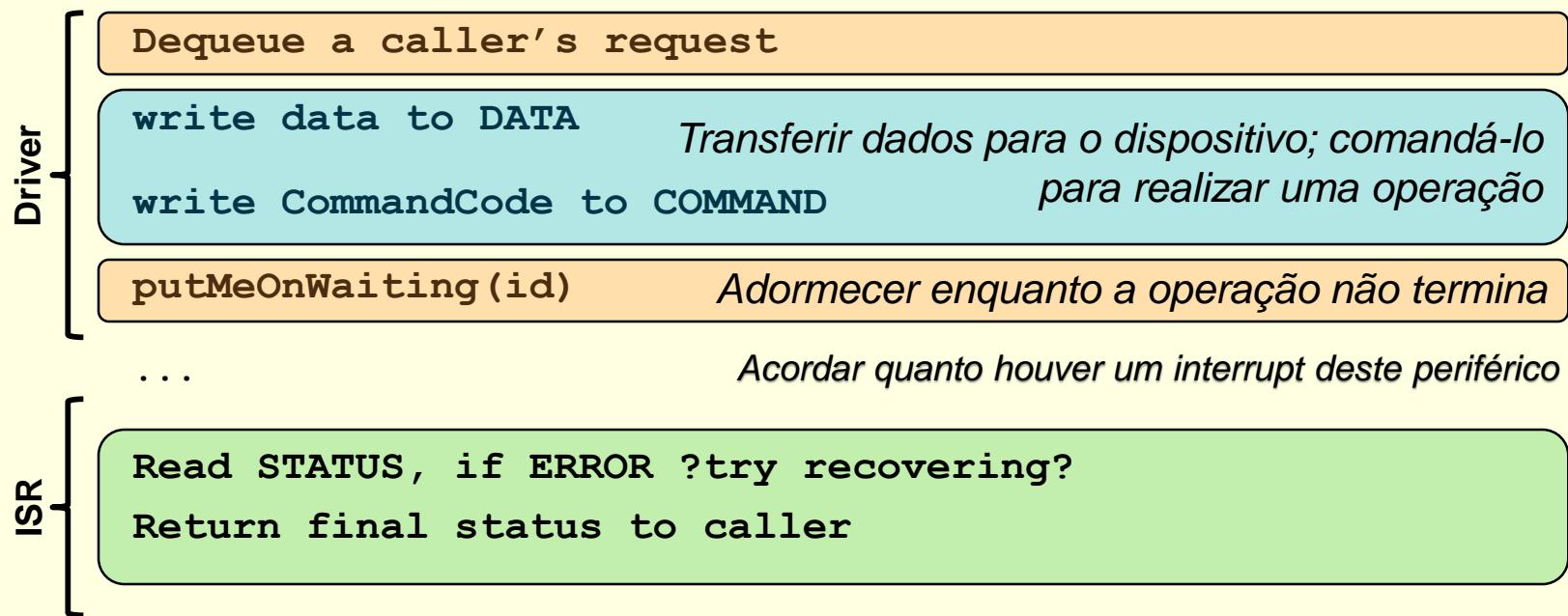
```
while (STATUS == BUSY)  
    ;
```

Verificar se o dispositivo está livre lendo o reg. status

Se o CPU transfere dados para o
dispositivo, é PIO (Programmed I/O)

Protocolo de interacção com periféricos (2)

□ **Interrupt-based** (exemplo de uma escrita no periférico)



Polling versus Interrupts

□ Programar I/O usando Polling

- Simples de programar
- Consumo excessivo de CPU em periféricos lentos (*busy waiting*)
- ... mas pode ser o mais eficiente em periféricos muito rápidos (e.g., placas de rede) capazes de gerar dezenas de milhares de interruptos por segundo

□ Programar I/O usando Interrupts

- Programação mais complexa
- Liberta o CPU, permitindo que este seja usado para ... ☺
- ... mas ver nota sobre periféricos muito rápidos

PIO versus DMA

□ Programed I/O

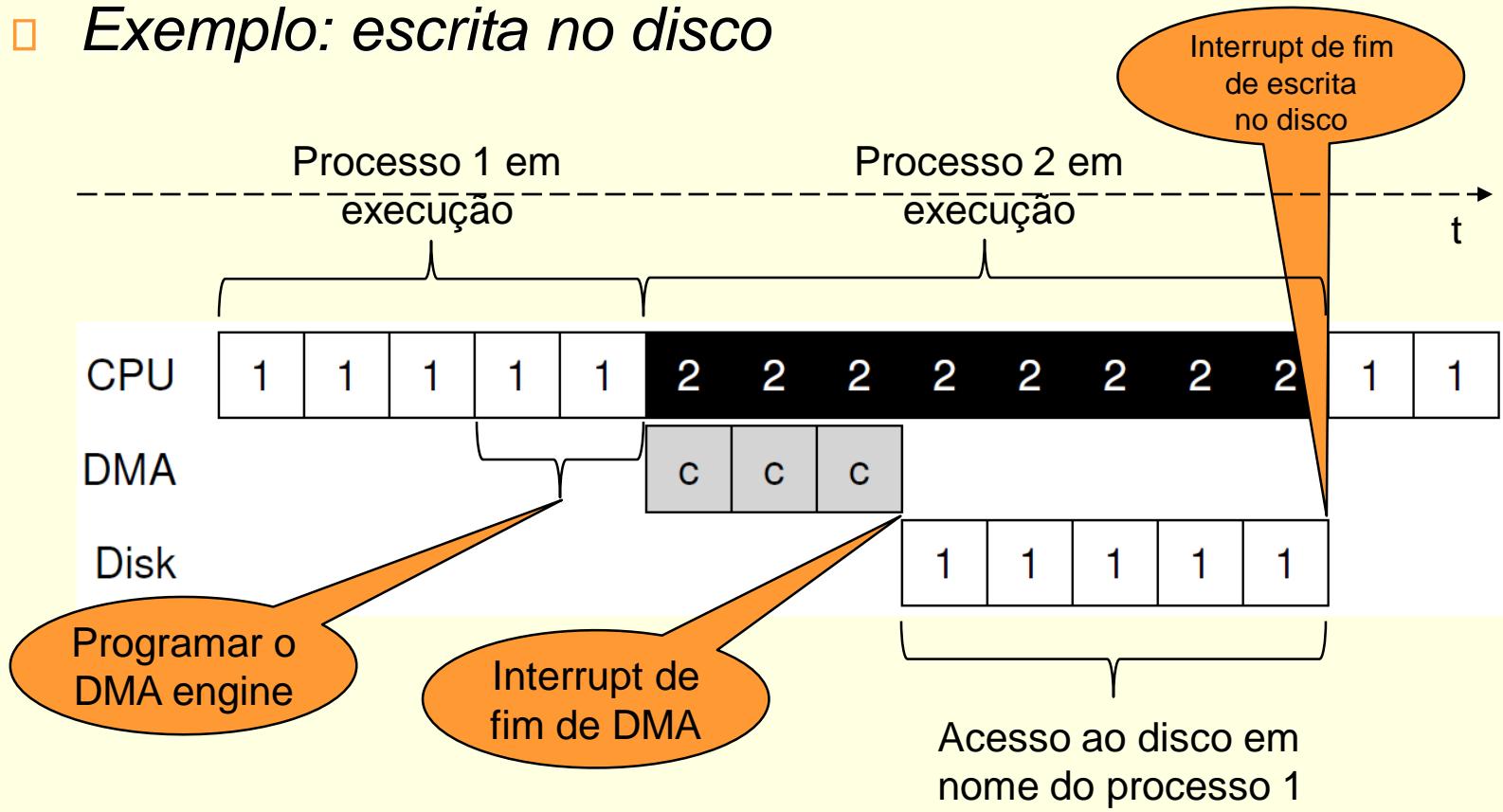
- + Simples de programar: *for/while loops* para transferir os dados entre o periférico e a RAM
- Consumo de CPU (e.g., transferir 4KB, usando registo de 4 bytes, exige 1K instruções *mov*)

□ DMA

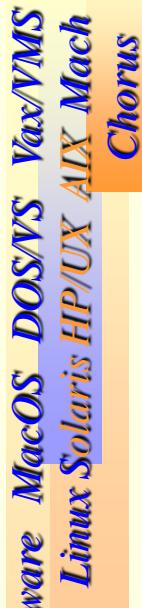
- Programação mais complexa: programa-se a unidade DMA indicando os endereços origem e destino, e número de bytes a transferir)
- O DMA, quando acaba a transferência, gera um interrupt (a programação é mais complexa)
- + CPU livre

Resumo visual: I/O com DMA e interrupções

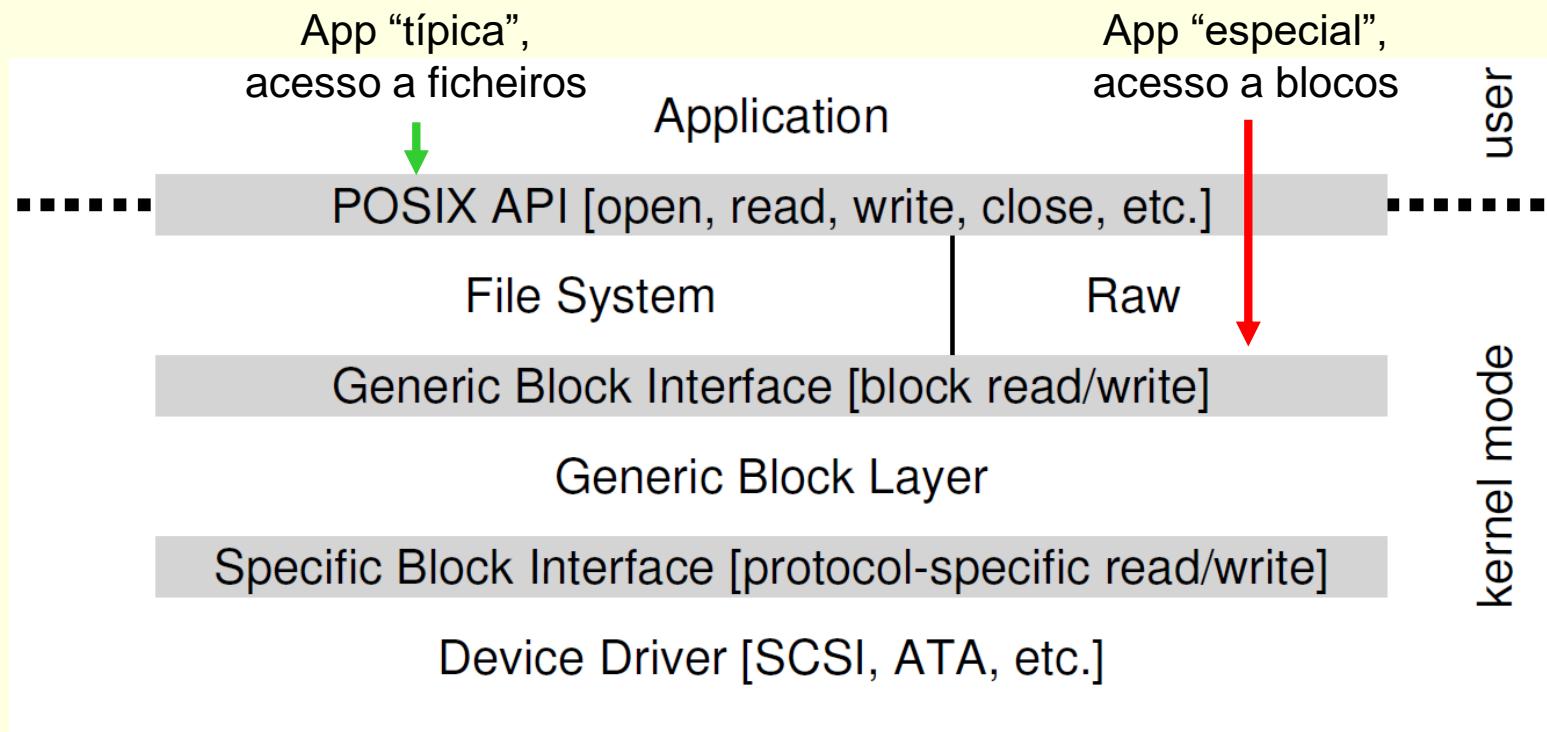
Exemplo: escrita no disco



O I/O stack no Linux (visão parcial, só SF)

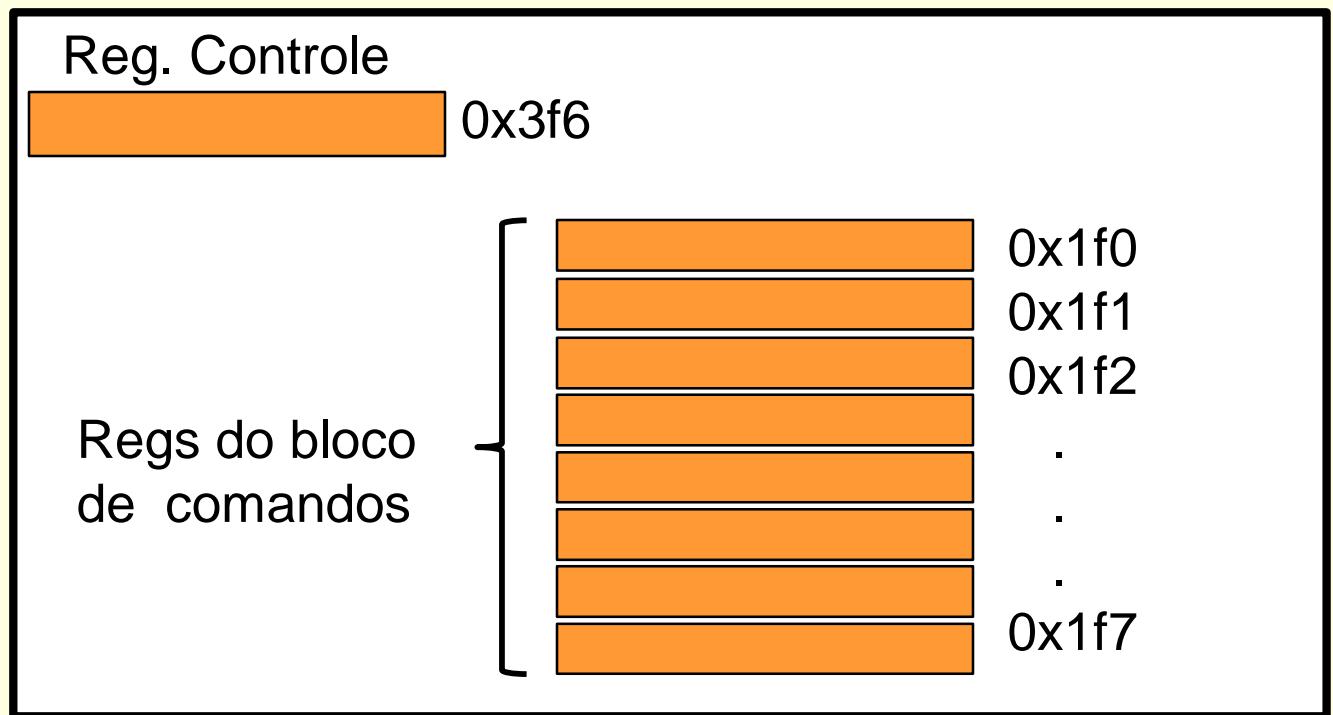


□ Sistema de ficheiros, da aplicação ao device driver



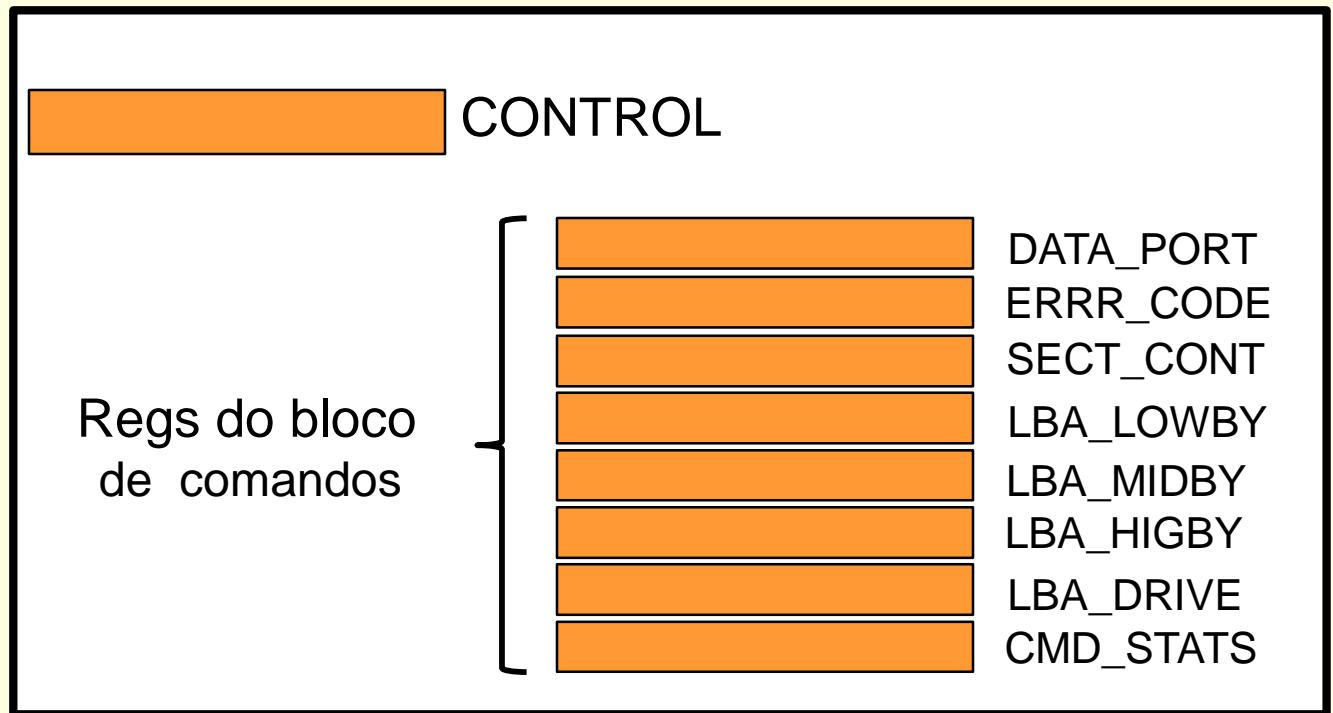
Caso de estudo: disco IDE (1)

Interface programática dum controlador de discos de “tipo IDE”



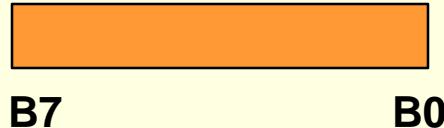
Caso de estudo: disco IDE (2)

Interface programática com “nomes” para os registos



Caso de estudo: disco IDE (3)

Bits do registo de estado

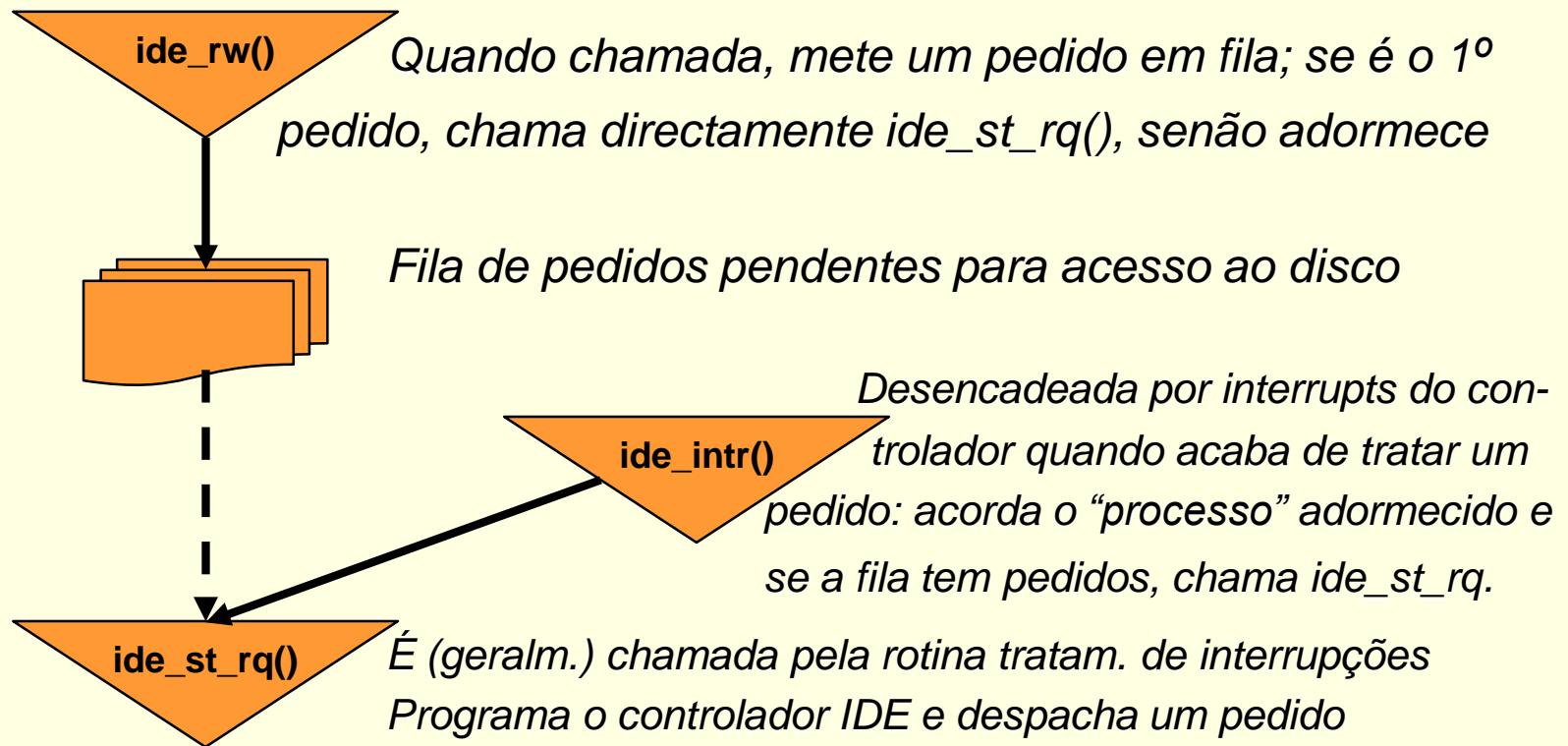


- B0: ERROR
- B1: IDDEX
- B2: CORR
- B3: DRQ
- B4: SEEK
- B5: FAULT
- B6: READY
- B7: BUSY

Quando o B0 indica erro, pode ler-se o registo ERRR_CODE para saber qual foi o erro, indicado também por 8 flags...

Caso de estudo: disco IDE (4)

Fluxo e estruturas de dados do driver



Programming an IDE controller (1)

```
#define CONTROL          0x3f6
#define DATA_PORT         0x1f0
#define ERRR_CODE          0x1f1
#define SECT_CNTR          0x1f2
#define LBA_LOWBY          0x1f3
#define LBA_MIDBY          0x1f4
#define LBA_HIGBY          0x1f5
#define LBA_DRIVE           0x1f6
#define CMD_STATS           0x1f7

#define USE_INTRS          0x08
#define IDE_WRITE            0x?? // não especificado no livro
#define IDE_READ             0x?? // não especificado no livro
```

Programming an IDE controller (2)

```
// Flags para usar no pacote do pedido. Não têm nada a ver
// com o controlador, são flags do "nossa" programa driver

#define B_VALID      ?      // o buffer contém dados válidos
#define B_DIRTY      ?      // o buffer tem de ser escrito
```

Programming an IDE controller (1)

```
// insere um pedido de acesso ao disco (especificado no pacote b) na fila de
// pedidos. Se é o 1º a ser inserido na fila, chama directamente a função para o
// processar

void ide_rw(struct buf *b) {
    acquire(&ide_lock);
    for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
        ; // walk queue up to the end
    *pp = b; // add request at the end
    if (ide_queue == b) // if queue was empty
        ide_st_rq(b); // immediately call the function
    while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
        sleep(b, &ide_lock); // wait for completion
    release(&ide_lock);
}
```

Programming an IDE controller (2)

```
// processa um pedido, programando o controlador IDE para que
// este o execute

static void ide_st_rq(struct buf *b) {
    ide_wait_ready();
    ProgramDiskTransfer(b, 1); // programa endereço e nº de blocos a transferir

    if(b->flags & B_DIRTY) {
        outb(CMD_STATS, IDE_CMD_WRITE); // this is a WRITE
        outsl(DATA_PORT, b->data, 512/4); // transfer data to W
    } else {
        outb(CMD_STATS, IDE_CMD_READ); // this is a READ
    }
}
```

Programming an IDE controller (3)

```
// interrupt routine

void ide_intr() {
    struct buf *b;
    acquire(&ide_lock);
    if (!(b->flags & B_DIRTY) && ide_wait_ready() >= 0)
        insl(DATA_PORT, b->data, 512/4); // if READ: get data
    b->flags |= B_VALID;
    b->flags &= ~B_DIRTY;
    wakeup(b); // wake waiting process
    if ((ide_queue = b->qnext) != 0) // start next request
        ide_st_rq(ide_queue); // (if one exists)
    release(&ide_lock);
}
```

Programming an IDE controller (4)

```
// Program the IDE disk controller to execute the task as
// specified in the b packet
static void PrgramDskTransf(struct buf *b, int blocks) {
    outb(SECT_CNTR, blocks);
    outb(LBA_LOWBY, b->sector & 0xff);
    outb(LBA_MIDBY, (b->sector >> 8) & 0xff);
    outb(LBA_HIGBY, (b->sector >> 16) & 0xff);
    outb(LBA_DRIVE, 0xe0 | ((b->dev&1)<<4) \
                      | ((b->sector>>24)&0x0f));
}

// busy wait until drive isn't busy
static int ide_wait_ready() {
    while (((int r = inb(CMD_STATS)) & IDE_BSY) || !(r & IDE_DRDY))
        ;
}
```