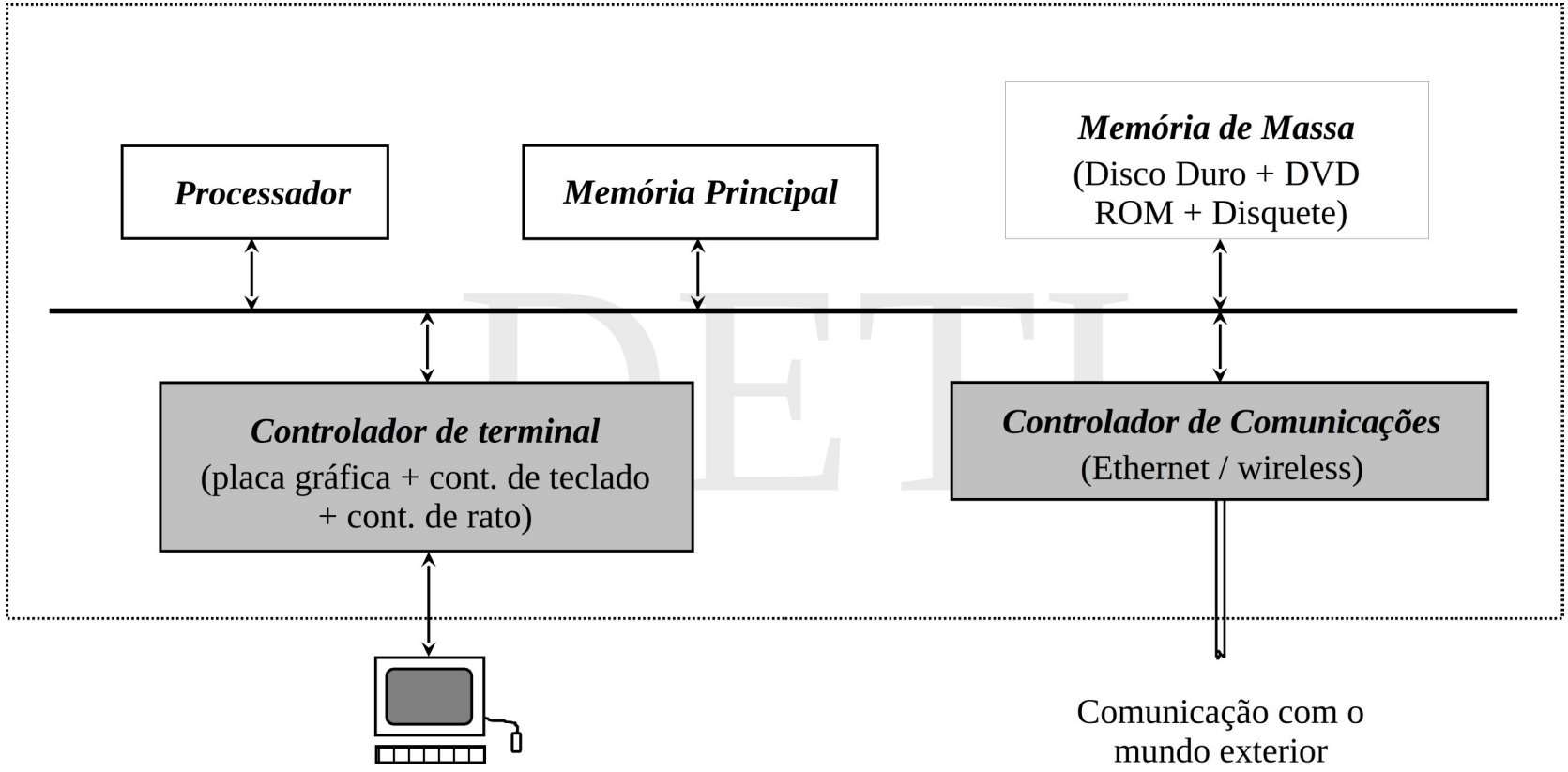
Sistemas de Operação

Entrada / Saída

Artur Pereira / António Rui Borges

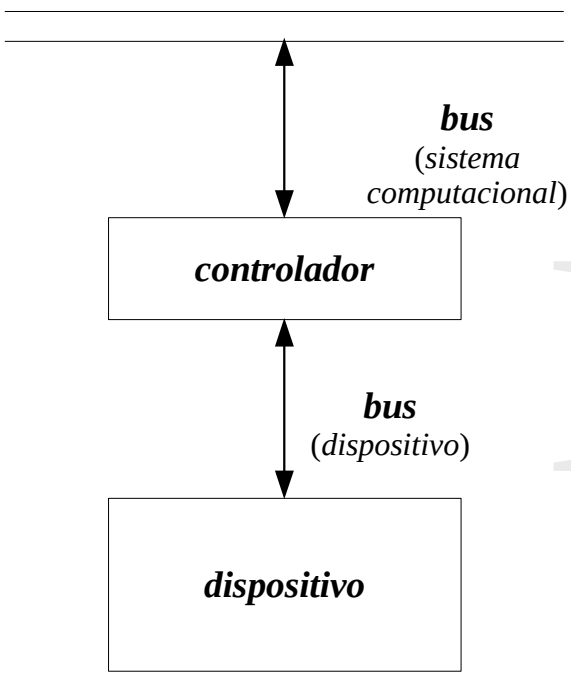
Sistema computacional



Papel do sistema de operação

- São habitualmente consideradas duas perspectivas distintas para enquadrar o papel desempenhado pelo sistema de operação na gestão dos dispositivos de entrada / saída do sistema computacional
  - *perspectiva do utilizador* – fornecer ao programador de aplicações um interface de comunicação com os diferentes dispositivos (*API*) que seja conceptualmente simples, razoavelmente uniforme e tanto quanto possível independente do dispositivo específico;
  - *perspectiva do construtor* – isolar os diferentes dispositivos do acesso direto por parte dos processos utilizador através da introdução de uma funcionalidade intercalar que controle diretamente os dispositivos (emita comandos, transfira os dados, gira as interrupções e lide com as condições de erro).

Anatomia de um dispositivo - 1

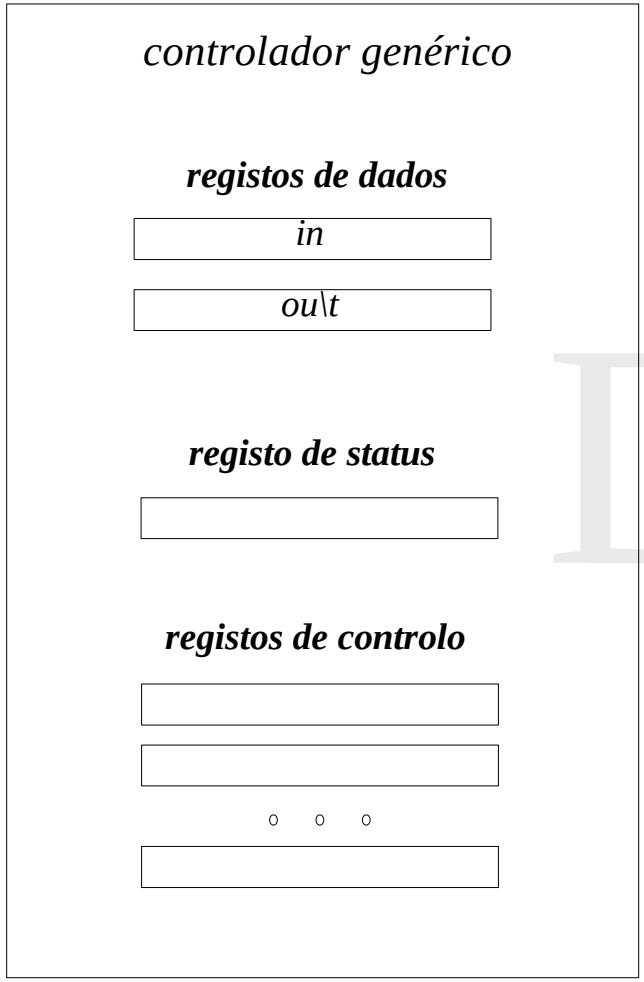


- dois componentes distintos que interagem
  - *dispositivo propriamente dito* – sistema físico, (eletromecânico, ótico-mecânico, ...), que armazena informação e que a converte de, ou para, uma forma acessível do exterior
  - *controlador do dispositivo* – circuito eletrónico, mais ou menos complexo, que funciona como interface entre o *bus* do dispositivo e o *bus* do sistema computacional e que está alojado no interior deste último
- Do ponto de vista do sistema de operação, o controlador constitui o único componente relevante
- Hoje em dia, muito versáteis, minimizam o papel do sistema de operação na sua gestão (programação)

Tipos de dispositivos

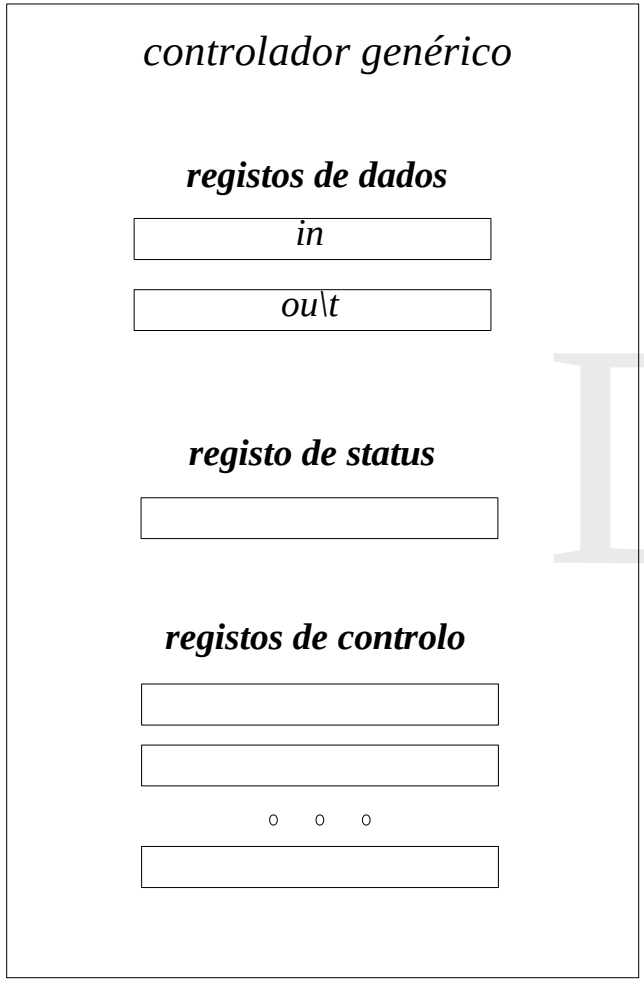
- Em termos de transferência de informação, os dispositivos de entrada / saída dividem-se em duas grandes categorias
  - *dispositivos de tipo carácter* – quando a transferência de informação se baseia num fluxo sucessivo de bytes, cujo número é variável
  - *dispositivos de tipo bloco* – quando a transferência de informação se baseia num número constante e pré-definido de bytes, o *bloco*, (tipicamente, de valor igual a uma potência de 2 entre 512 e 16K); é costume, neste caso, exitir uma restrição temporal à duração máxima da transferência.
- O modo como se materializa a transferência depende fundamentalmente da largura do *bus* de dados, sendo comum ser feita em *bytes* (8 bits), *palavras simples* (16 bits), *palavras duplas* (32 bits) ou *palavras quádruplas* (64 bits).
- A velocidade a que ocorre a transferência depende do tipo de dispositivo, podendo variar desde os poucos B/s (teclado, por exemplo) até algumas centenas ou milhares de MB/s (disco SATA ou USB 3, por exemplo)

Interface com o processador - 1



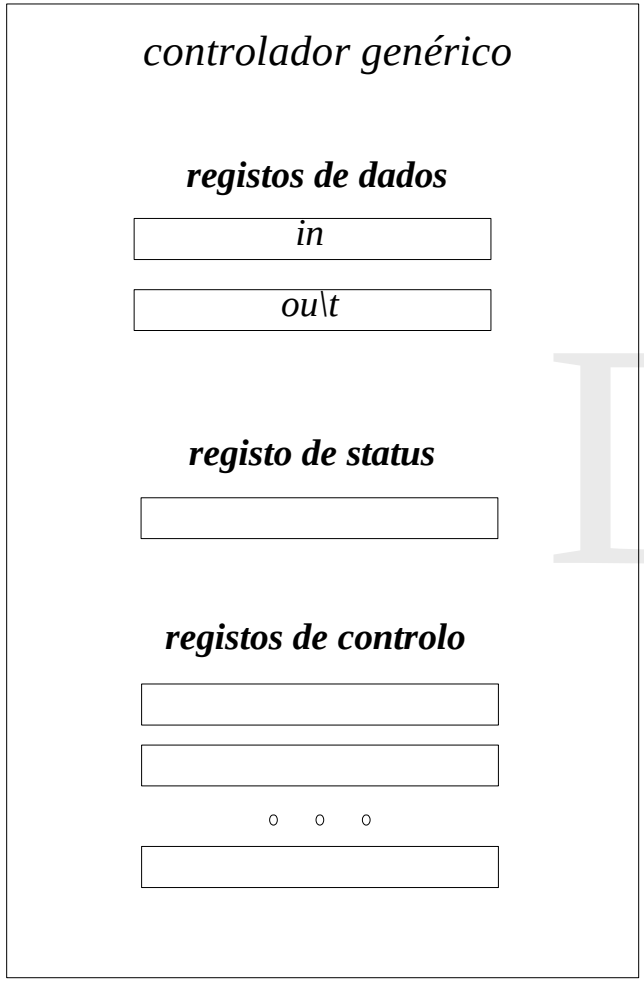
- Um *controlador genérico* é entendido, sob o ponto de vista de programação, como sendo formado por diferentes tipos de registos
  - *Registos de controlo* – desempenham múltiplas funções;
    - Configuração do dispositivo
    - Definição do tipo de transação com o processador (*polled I/O*, *interrupt driven* ou *DMA based*)
    - Em controladores mais complexos, impõe a execução de operações

Interface com o processador - 2



- *Registo de status* – descreve o estado interno atual do controlador
  - pronto a aceitar um novo comando
  - último comando foi realizado com sucesso
  - ocorreram erros (e quais)
  - …
- *Registos de dados* – comunicação propriamente dita com o dispositivo
  - valores escritos no registo *out* são enviados para o dispositivo
  - valores lidos do registo *in* são provenientes do dispositivo e são postos à disposição

Interface com o processador - 3

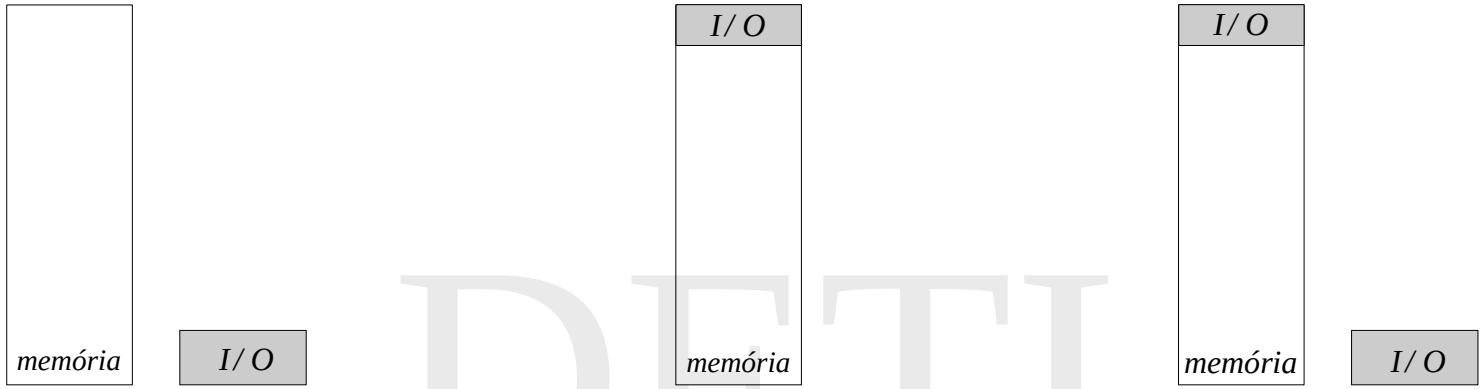


- Nos controladores de *dispositivos de tipo carácter*, os comandos de escrita e de leitura de dados são implícitos
  - um valor escrito no registo *out* é enviado para o dispositivo
  - um valor enviado pelo dispositivo ao controlador é colocado no registo *in*
- Nos controladores de *dispositivos de tipo bloco*, a transferência de dados é posta em marcha pela execução de um comando explícito
  - o registo de dados é normalmente único, *in-out*, e o sentido da transferência é função do comando.

Modos de endereçamento - 1

- Há três formas possíveis do processador aceder aos registos dos controladores
  - *I/O-mapped* – os controladores têm um espaço de endereçamento próprio
    - os seus registos são acedidos através de instruções específicas (*in* e *out*);
  - *memory-mapped I/O* – os controladores usam parte do espaço de endereçamento da memória
    - os seus registos são acedidos através de instruções gerais de acesso à memória (*load* e *store*);
  - *híbrida* – os controladores usam, em princípio, o seu espaço de endereçamento próprio, mas grandes *buffers de dados* são mapeados em memória para facilitar a comunicação.

Modos de endereçamento - 2



- O *Pentium* da Intel tem um espaço de endereçamento de *I/O* de 64 KB. Os PC's compatíveis com IBM usam este espaço prioritariamente para endereçar controladores, mas a região de memória, entre os endereços 640 KB e 1 MB, está também reservada para a implementação de *buffers de dados* de dispositivos.

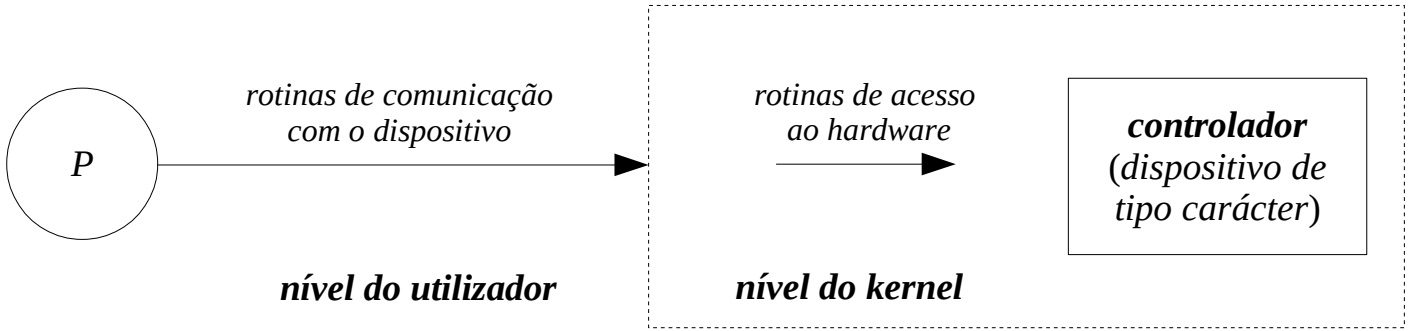
Objectivos da programação de I/O - 1

- O ambiente fornecido pelo sistema de operação na comunicação com os *dispositivos de entrada / saída* deve
  - *ser independente do dispositivo específico* – os programas devem poder ser escritos de modo a aceder a um dispositivo genérico, possibilitando que, ao nível do interpretador de comandos, por exemplo, o *redireccionamento de I/O* seja operacionalizado de uma forma natural;
  - *usar um mecanismo de nomeação uniforme* – o nome dos dispositivos deve ser constituído por sequências de caracteres, ou valores numéricos, sem qualquer significado particular;
  - *efetuar a gestão de erros de uma maneira integrada* – a *política geral* deve ser *só comunicar à camada superior o erro se a camada inferior não puder lidar com ele*; ou seja, a detecção dos erros deve ser realizada tão próxima do dispositivo quanto possível de modo a permitir a sua [eventual] recuperação de uma forma transparente;

Objectivos da programação de I/O - 2

- O ambiente fornecido pelo sistema de operação na comunicação com os *dispositivos de entrada / saída* deve (cont.)
  - *desacoplar os dispositivos dos processos utilizador* – a grande maioria dos *dispositivos de entrada / saída* funciona de uma maneira *assíncrona* (as transferências de dados de, e para, a memória principal são despoletadas por interrupções); na perspetiva do utilizador, contudo, é mais simples conceber a comunicação de uma maneira *síncrona* (o processo bloqueia até que estejam reunidas condições para que a comunicação tenha lugar);
  - *gerir de uma forma externamente uniforme o acesso a dispositivos preemptable e non-preemptable* – a comunicação com dispositivos *preemptable* pode ser *partilhada* por múltiplos utilizadores em simultâneo; com dispositivos *non-premptable*, porém, passa-se exatamente o contrário: a comunicação tem que decorrer em regime de exclusão mútua, ou *dedicado*; o sistema de operação tem, pois, que identificar as diferentes situações e garantir a adequada coordenação.

Polled I/O - 1



- Numa estratégia *polled – I/O*, não há qualquer tipo de desacoplamento e é o próprio *processo utilizador* que se encarrega das comunicações com o dispositivo
  - As *rotinas de comunicação com o dispositivo* constituem *chamadas ao sistema* que implementam diretamente o acesso ao hardware
- Trata-se da solução mais simples, mas é muito pouco eficiente porque o processador é colocado em *busy waiting* aguardando o processamento da informação no controlador

Polled I/O - 2

```
/* rotinas de acesso ao hardware; supõe-se um controlador de tipo carácter */

void control(unsigned short add, unsigned char prog []);

#define  RXRDY      ...     /* há dados para seram lidos */
#define  TXRDY      ...     /* registo de saída vazio */
#define  ERROR      ...     /* sinalização de ocrrência de erro */

unsigned char status(unsigned short add);

unsigned char in(unsigned short add);

void out(unsigned short add, unsigned char val);
```

Polled I/O - 3

```
/* rotinas de comunicação com o dispositivo
   chamadas ao sistema – executadas ao nível do kernel
   supõe-se que já foi estabelecido um canal de comunicação */

/* leitura de N bytes
       dd   --- descritor do dispositivo
       N    --- n.º de bytes a ler
       buff --- ponteiro para a região de armazenamento
    valor devolvido:
       0 --- operação realizada com sucesso
      -1 --- ocorreu um erro (descrito na variável global errno) */

int readNBytes (int dd, int N, unsigned char buff[]);

/* escrita de N bytes
       dd   --- descritor do dispositivo
       N    --- n.º de bytes a escrever
       buff --- ponteiro para a região de armazenamento
    valor devolvido:
       0 --- operação realizada com sucesso
      -1 --- ocorreu um erro (descrito na variável global errno) */

int writeNBytes (int dd, int N, unsigned char buff[]);
```

Polled I/O - 4

```
int readNBytes(int dd, int N, unsigned char buff[])
{
    int add = getAdd(dd);

    for (int n = 0; n < N; n++)
    {
        int stat;
        do
        {
            stat = status(add);
            if ((stat & ERROR) != 0)
                /* error handling */
        } while ((stat & RXRDY) == 0);

        buff[n] = in(add);
    }
    return 0;
}
```

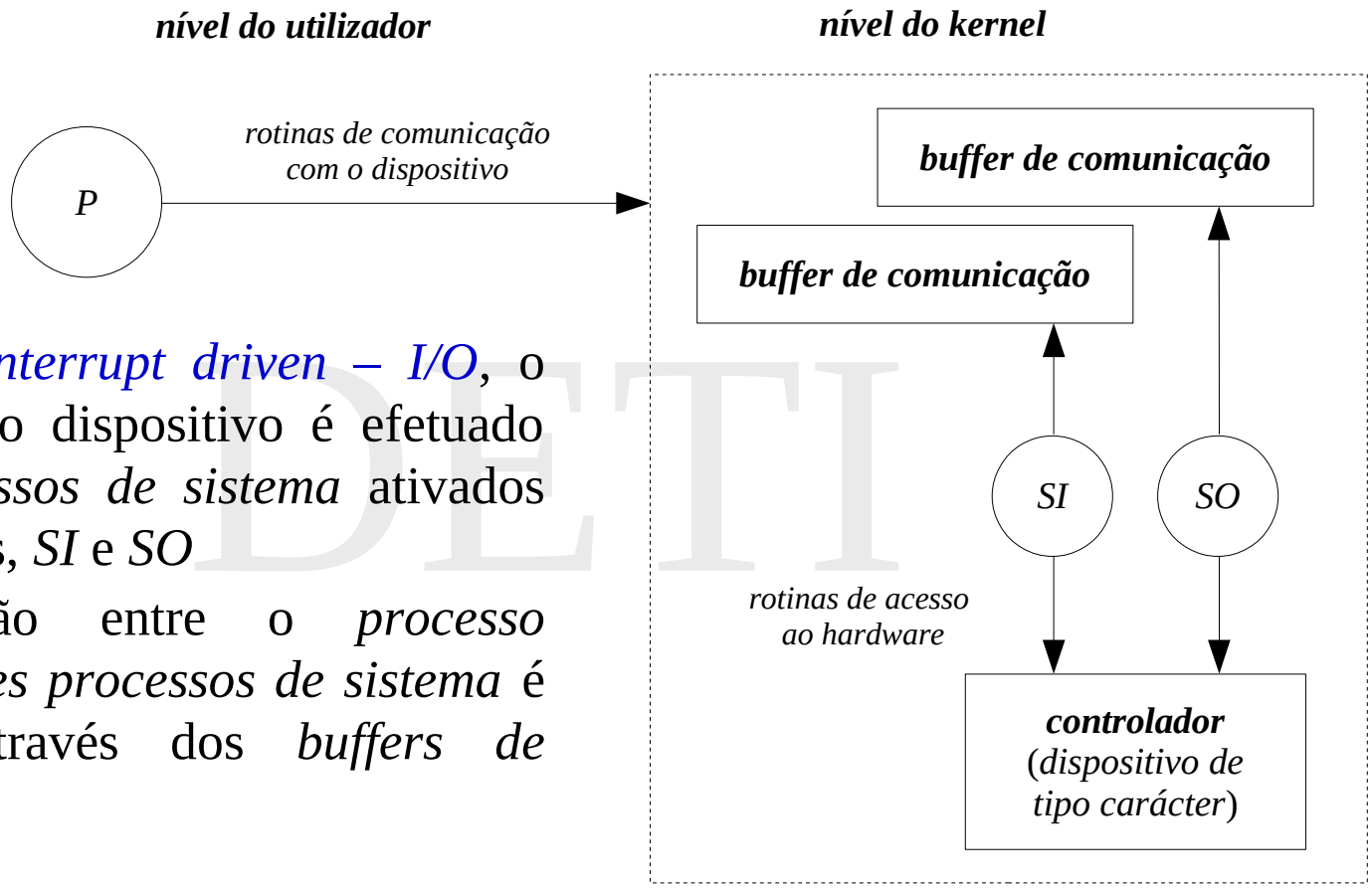
Polled I/O - 5

```
int writeNBytes(int dd, int N, unsigned char buff [])
{
    int add = getAdd(dd);

    for (int n = 0; n < N; n++)
    {
        int stat;
        do
        {
            stat = status(add);
            if ((stat & ERROR) != 0)
                /* error handling */
        } while ((stat & TXRDY) == 0);

        out(add, buff[n]);
    }
    Return 0;
}
```

Interrupt driven I/O - 1



- Na estratégia *interrupt driven – I/O*, o acesso direto ao dispositivo é efetuado por dois *processos de sistema* ativados por interrupções, *SI* e *SO*
- A comunicação entre o *processo utilizador e estes processos de sistema* é estabelecida através dos *buffers de comunicação.*

Interrupt driven I/O - 2

- Os processos *SI* e *SO* representam as rotinas de serviço às interrupções geradas pelo controlador quando, respetivamente, o registo de dados de entrada contém um novo byte, ou o registo de dados de saída está vazio e pronto a receber um novo byte
  - São em qualquer caso processos especiais que executam até ao esgotamento
  - Só perdem o controlo do processador se um outro processo ativado por interrupção, de prioridade mais elevada, for calendarizado para execução
- As *rotinas de comunicação com o dispositivo* implementam o acesso aos *buffers de comunicação*, desacoplando o dispositivo do *processo utilizador*.
- A solução resultante é mais complexa, mas muito mais eficiente. Os *processos utilizador* bloqueiam, libertando o processador, se não se reunirem condições para a transferência de informação.

Interrupt driven I/O - 3

```
/* buffes de comunicação para dispositivos de tipo carácter */

typedef struct
{
    FIFO fifo;          /* região de armazenamento */
    SEMAPHORE wait;     /* semáforo de bloqueio do processo utilizador */
    bool noMoreInt;     /* fim das interrupções (apenas para disp. de saída) */
    int errnumb;        /* indicação de erro na transferência */
} COM_BUFF;
```


Observações

- num *buffer de entrada*, o campo `val` do semáforo `wait` é inicializado a zero, significando que o FIFO está vazio e não há dados a recolher;
- num *buffer de saída*, o campo `val` do semáforo `wait` é inicializado a K, em que K é o tamanho do FIFO, significando que o FIFO está vazio e podem ser depositados K bytes;
- a *flag* `noMoreInt` sinaliza a necessidade de escorvamento do registo de dados de saída do controlador para que sejam geradas de novo interrupções, sendo inicializada a **true**.

Interrupt driven I/O - 4

```
/* rotinas de comunicação com o dispositivo
   chamadas ao sistema – executadas ao nível do kernel
   supõe-se que já foi estabelecido um canal de comunicação */

/* leitura de N bytes
       dd   --- descritor do dispositivo
       N    --- n.º de bytes a ler
       buff --- ponteiro para a região de armazenamento
    valor devolvido:
       0 --- operação realizada com sucesso
      -1 --- ocorreu um erro (descrito na variável global errno) */

int readNBytes (int dd, int N, unsigned char buff[]);

/* escrita de N bytes
       dd   --- descritor do dispositivo
       N    --- n.º de bytes a escrever
       buff --- ponteiro para a região de armazenamento
    valor devolvido:
       0 --- operação realizada com sucesso
      -1 --- ocorreu um erro (descrito na variável global errno) */

int writeNBytes (int dd, int N, unsigned char buff[]);
```

Interrupt driven I/O - 5

```
int readNBytes(int dd, int N, unsigned char buff[])
{
    COMM_BUF * inbuf = getBuff(dd);

    for (int n = 0; n < N; n++)
    {
        if (inbuf->errNumb != 0)
            /* error handling */

        sem_down(inbuf->wait);

        disable_interrupts();
        buff[n] = fifoOut(inbuf->fifo);
        enable_interrupts();
    }
    return 0;
}
```

Interrupt driven I/O - 6

```
int writeNBytes(int dd, int N, unsigned char buff [])
{
{
    COMM_BUF * outbuf = getBuff(dd);

    for (int n = 0; n < N; n++)
    {
        if (outbuf->errNumb != 0)
            /* error handling */

        sem_down(outbuf->wait);

        disable_interrupts();
        fifoIn(outbuf->fifo, buff[n]);
        if (outbuf->noMoreInt)
        {
            writeDReg(getAdd(dd));
            Outbuf->noMoreInt = false;
        }
        enable_interrupts();
    }
    return 0;
}
```

Interrupt driven I/O - 8

```
/* processo SI - leitura de um byte do registo de dados de entrada do controlador
       add --- endereço do controlador */

void readDReg(unsigned short add)
{
    COM_BUFF * inbuf = getBuff(getDesc(add));

    int stat = status(add);              /* leitura do registo de status */
    if ((stat & ERROR) == ERROR)
       inBuff->errnumb = stat & ERROR;   /* reporta erro encontrado */
    if ((stat & RXRDY) == RXRDY)         /* há dados a ler */
    {
        char val = in(add);
        if ((stat & ERROR) != ERROR)
        {
            if (!fifo_full(inbuf->fifo))
            {
                fifo_in(inbuf->fifo, val);    /* armazena o byte no buffer */
                sem_up(inbuf->wait);          /* sinaliza que há dados no buffer */
            }
            else
                inbuf->errNumb = OVERRUN;    /* erro de overrun */
        }
    }
}
```

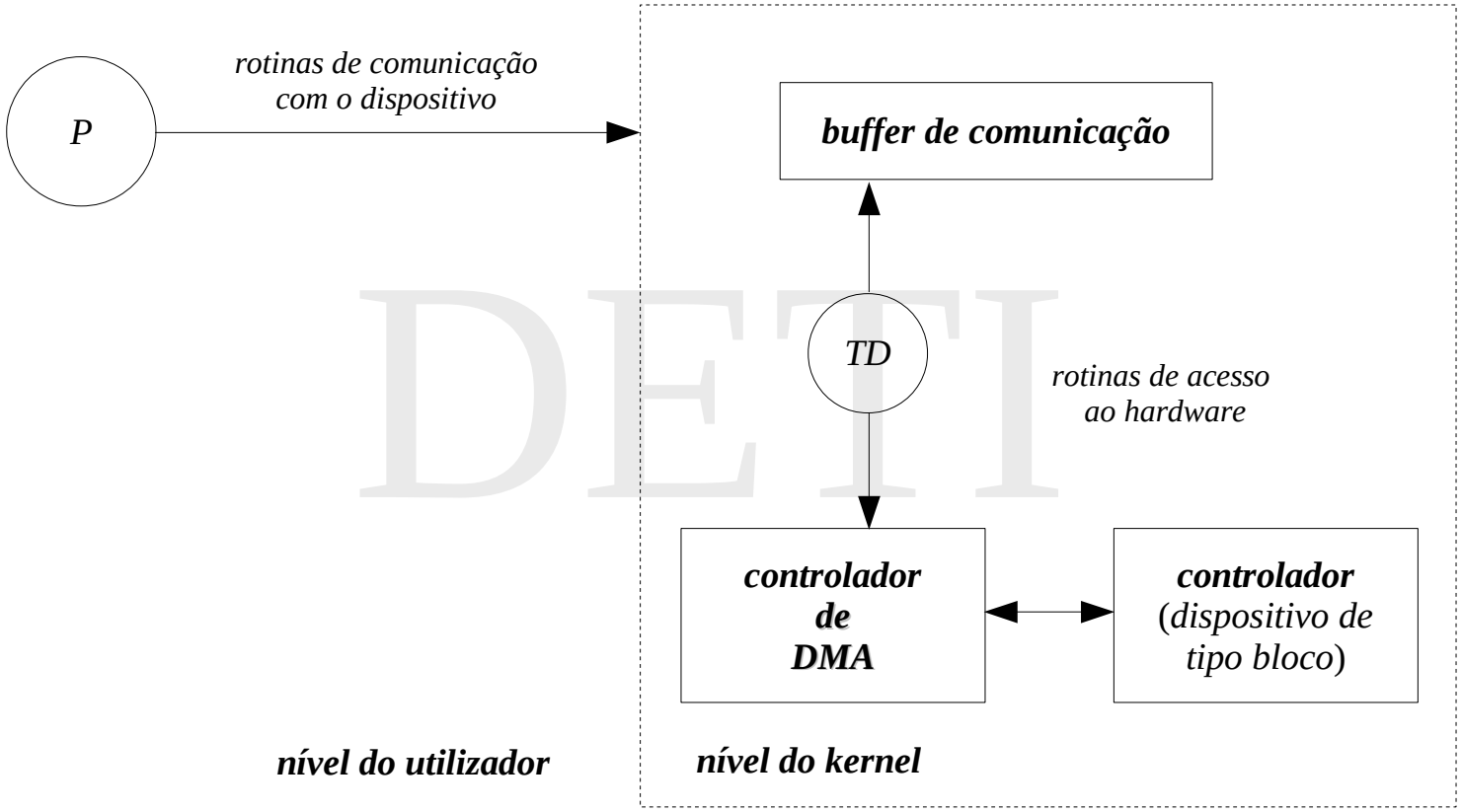
Interrupt driven I/O - 9

```
/* processo SO - escrita de um byte no registo de dados de saída do controlador
       add --- endereço do controlador */

void writeDReg(unsigned short add)
{
    COM_BUFF * outbuf = getBuff(getDesc(add));

    int stat = status(add);              /* leitura do registo de status */
    if ((stat & ERROR) == ERROR)
       outbuf->errnumb = stat & ERROR;  /* reporta erro encontrado */
    if ((stat & TXRDY) == TXRDY)         /* pode escrever */
    {
        if (!fifo_empty(outbuf->fifo))
        {
            char val = fifo_out(outbuf->fifo);  /* retira um byte do buffer */
            sem_up(outbuf->wait);  /* sinaliza que há espaço livre no buffer */
            out(add, val);   /* escrita do byte no registo de saída de dados */
         }
         else
             outbuf->noMoreInt = true;  /* faz o set da flag de sinalização */
     }
}
```

DMA based I/O - 1



DMA based I/O - 2

- Numa estratégia *DMA based – I/O*, o controlador de DMA está diretamente ligado ao controlador do dispositivo, ou está integrado nele
- O princípio subjacente é que quando o controlador do dispositivo pretende transferir informação, ativa uma entrada de *request transfer* no controlador de DMA
- Em resultado disso, este toma controlo do *bus* e procede à realização de uma de duas operações:
  - leitura de um byte ou de uma palavra simples do registo de dados do controlador do dispositivo e subsequente escrita desse valor numa região de memória (*entrada de dados*)
  - leitura de um byte ou de uma palavra simples de uma região de memória e subsequente escrita desse valor no registo de dados do controlador do dispositivo (*saída de dados*)