



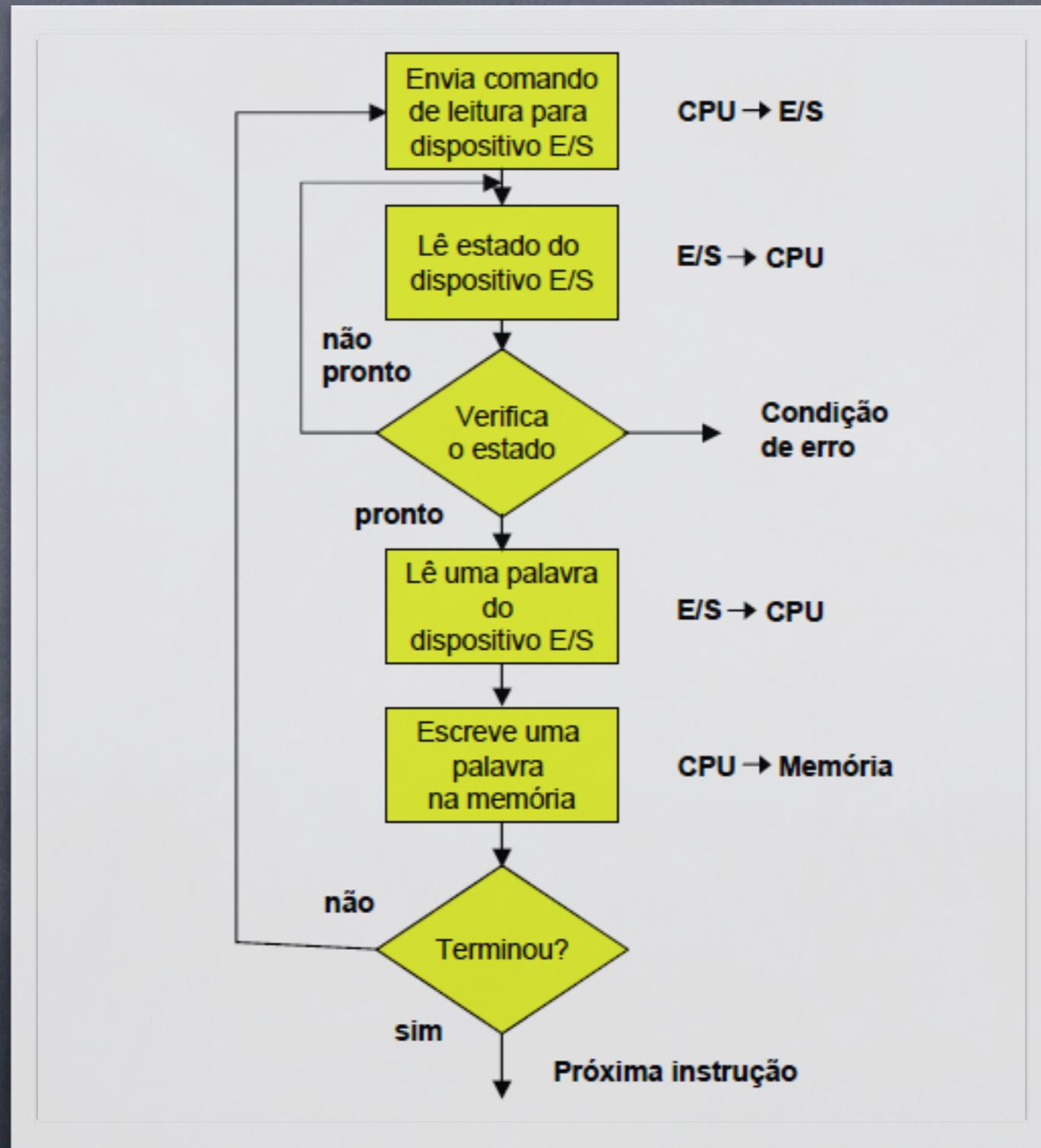
MAC422 Sistemas Operacionais

Prof. Marcel P. Jackowski

Aula #5

Interrupções, traps e syscalls

E/S via “polling”



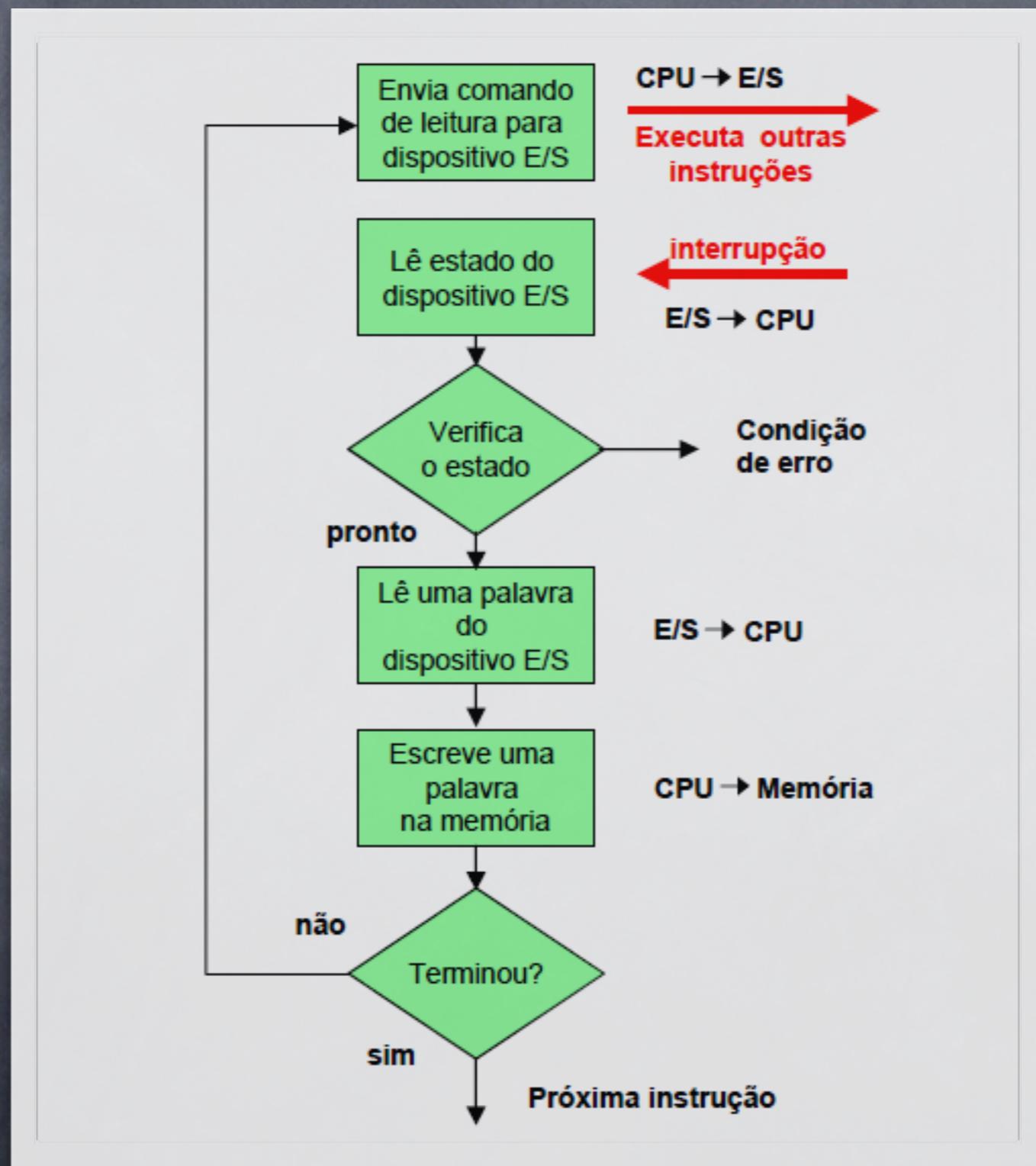
Motivação

- Para controlar entrada e saída de dados, não é interessante que a CPU tenha que ficar continuamente monitorando o status de dispositivos como discos ou teclado.
- Precisamos de um mecanismo que permita que o hardware "chame a atenção" da CPU quando há algo a ser feito.

Motivação

- CPU requer mecanismo para reagir a mudanças no estado do sistema. Exemplos:
 - Dados chegam do disco ou na interface de rede
 - Instrução resulta em divisão por zero
 - Usuário tecla CTRL-C
 - Temporizador expira
- Sistema requer um mecanismo para “fluxo de controle excepcional” (onde o HW chama a atenção da CPU quando algo precisa ser feito)

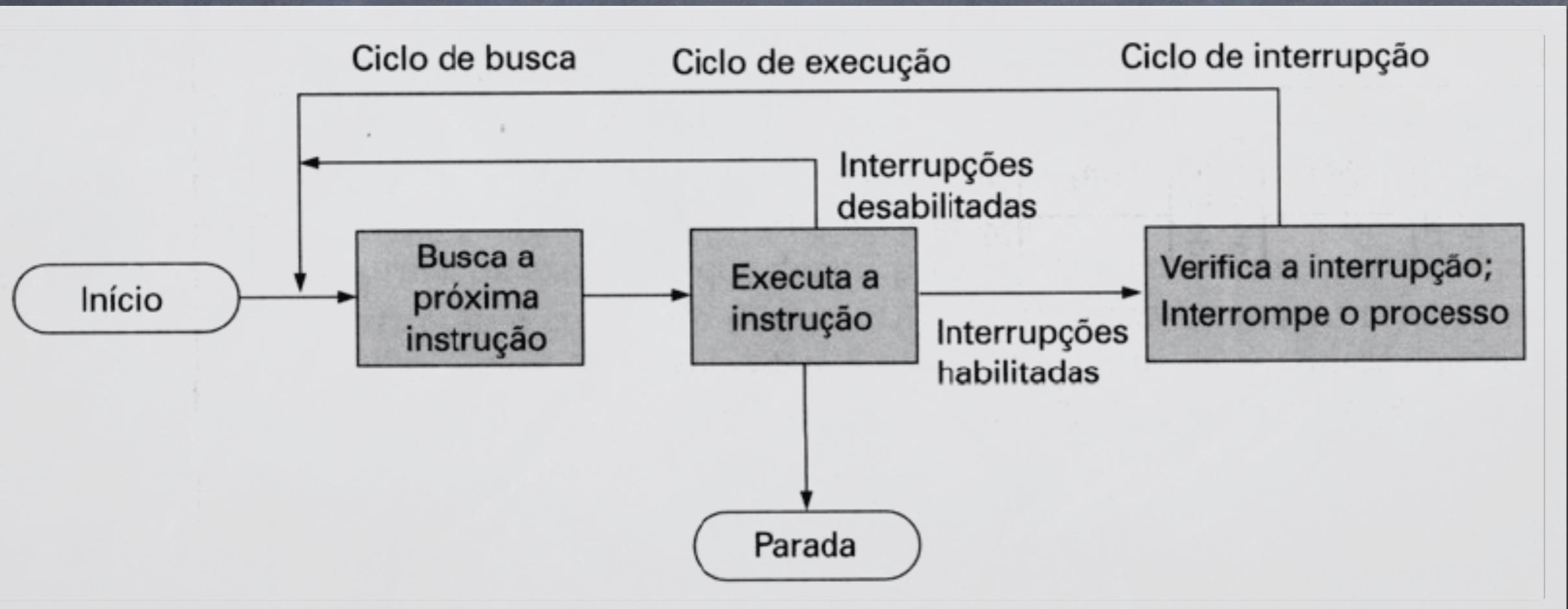
E/S via interrupções



Interrupção: desvio de fluxo

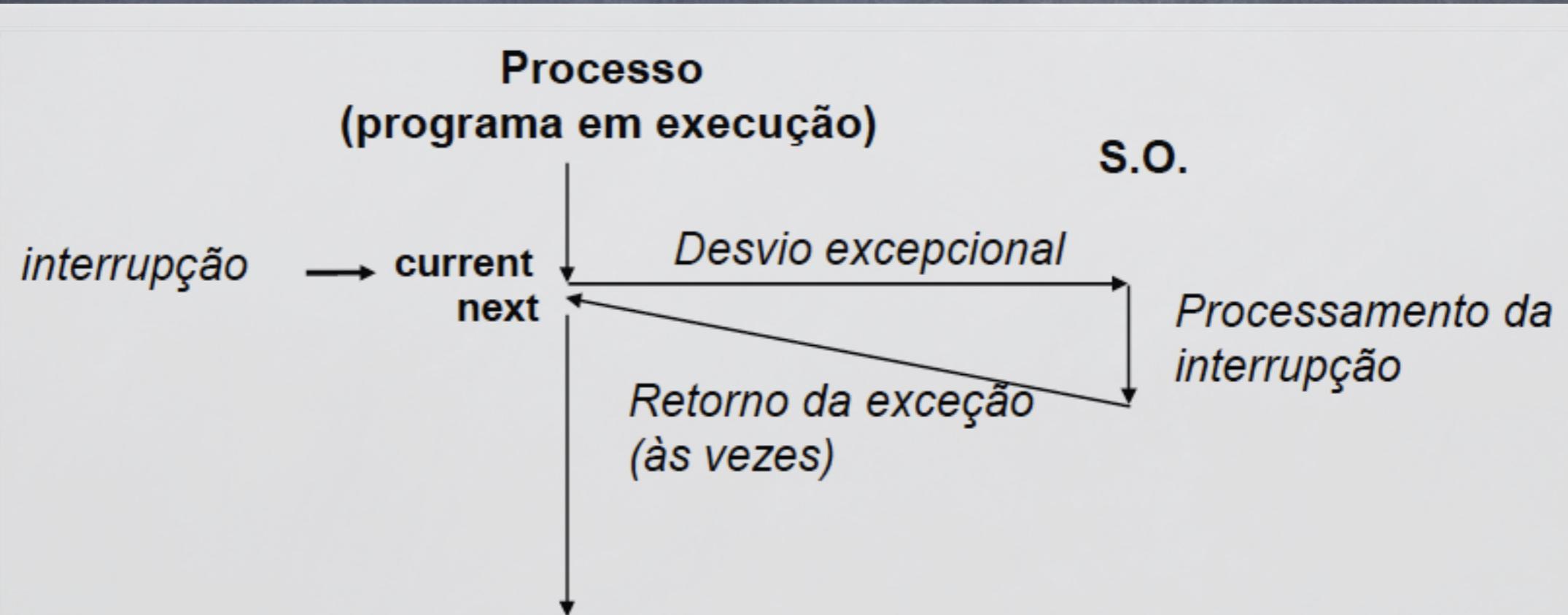
- Existem dois mecanismos principais para alterar o fluxo de controle:
 - Desvios incondicionais e condicionais
 - Chamada (e retorno) de procedimento usando a pilha
- Ambos reagem a mudanças no estado do programa, mas insuficientes para sistemas com entrada/saída.

Ciclo de interrupção

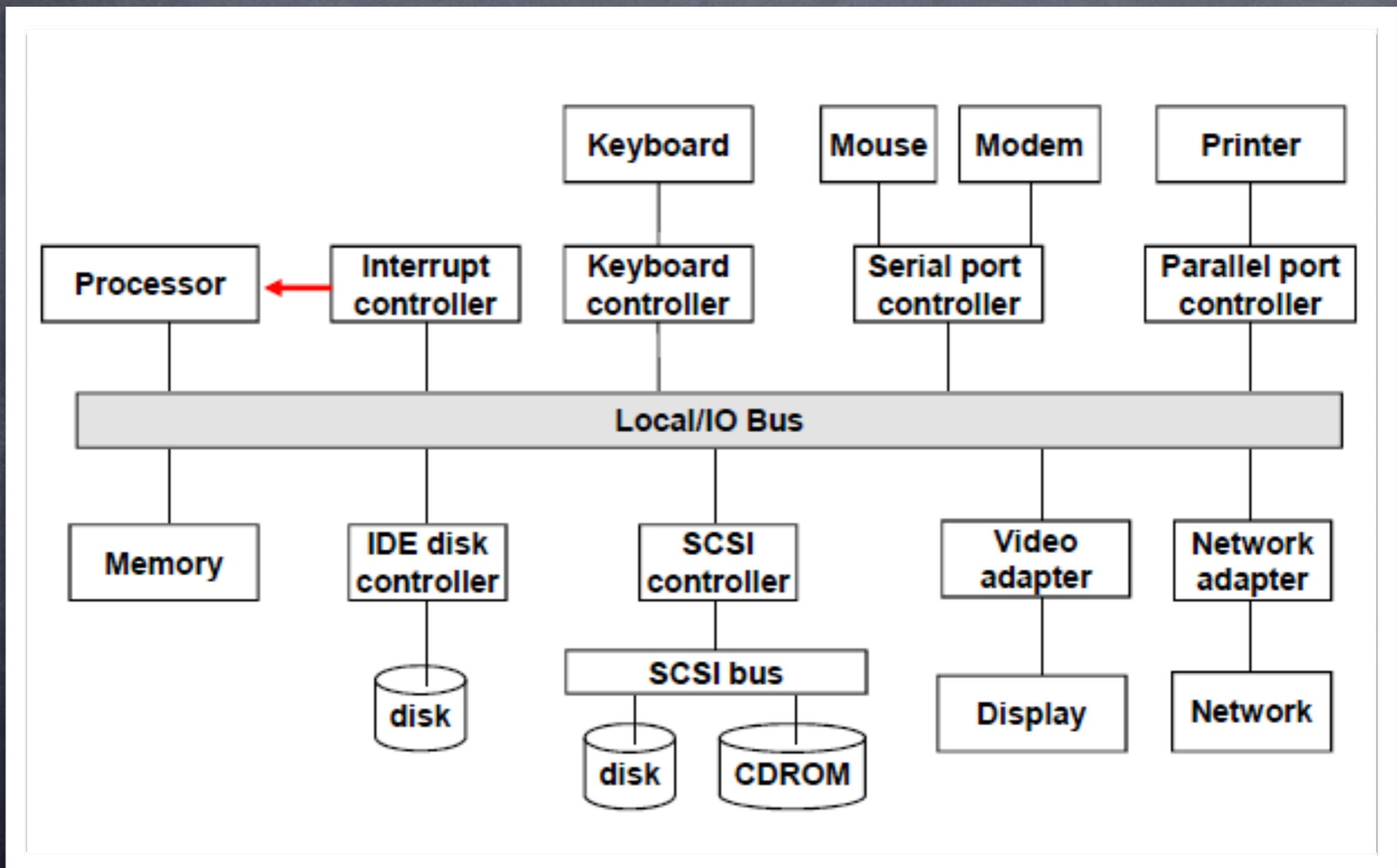


Interrupção

- É uma transferência de controle (e.g. para o SO) em resposta a um evento de hardware ou software.



Arquitetura básica

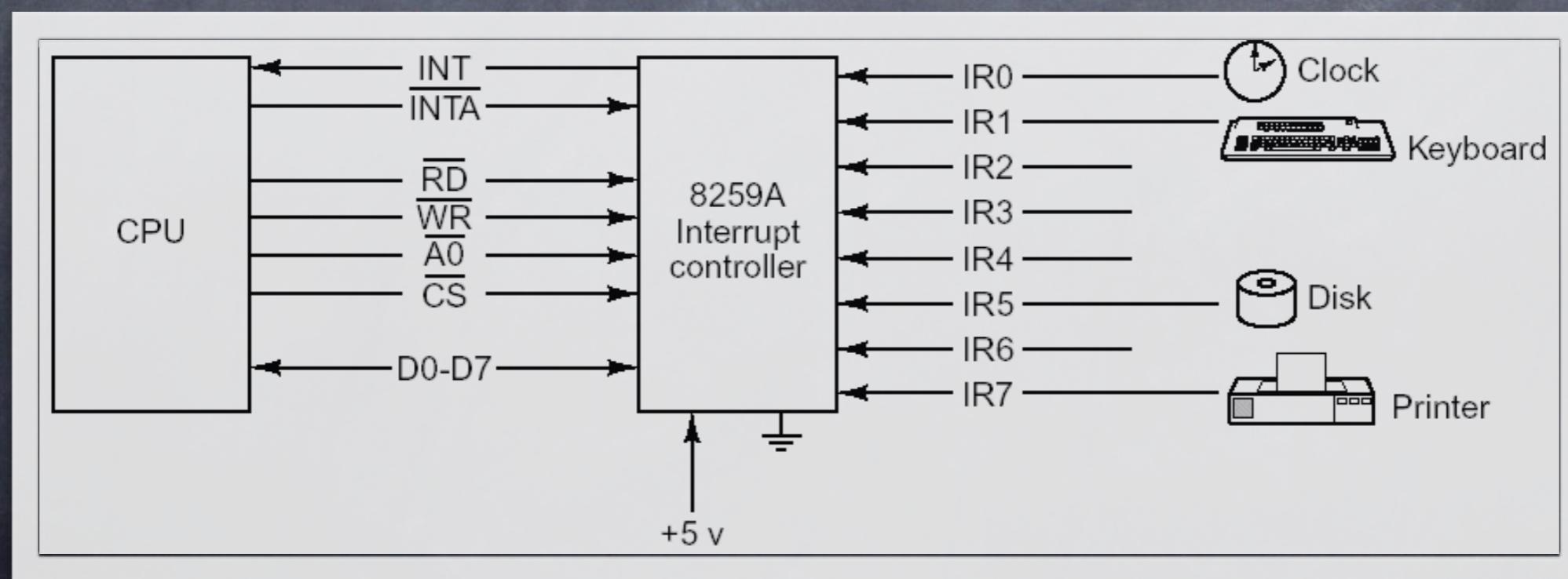


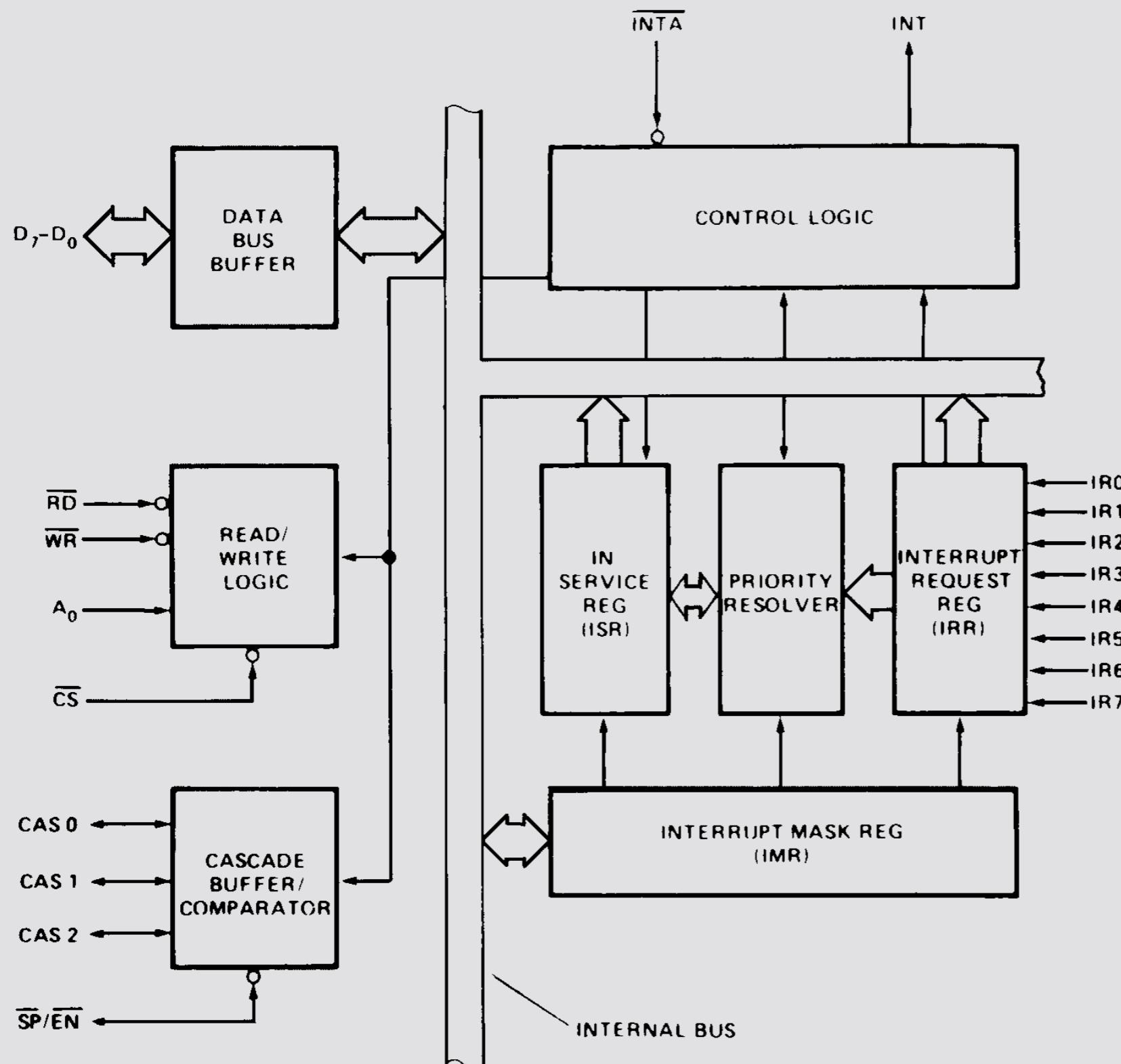
Programmable Interrupt Controller (PIC)

- **Interrupt requests (IRQs)**
 - Canais para requisição de interrupções – um canal por periférico
- **Programmable Interrupt controller (PIC)**
 - Um controlador de interrupções é responsável pelo encaminhamento das interrupções dos periféricos para o processador
 - Estabelece um protocolo com o processador, trocando dados necessários para servir a interrupção.

PIC Intel 8259A

- O controlador de interrupções ativa o sinal INT
- O CPU responde com INTA (INTerrupt Acknowledge)
- O controlador de interrupções responde através do bus de dados (D0 a D7) com o número da entrada que produziu a interrupção
- O CPU utiliza esse número para indexar uma tabela de endereços de memória onde estão os conjuntos de instruções que servem cada interrupção.





231468-1

Figure 1. Block Diagram

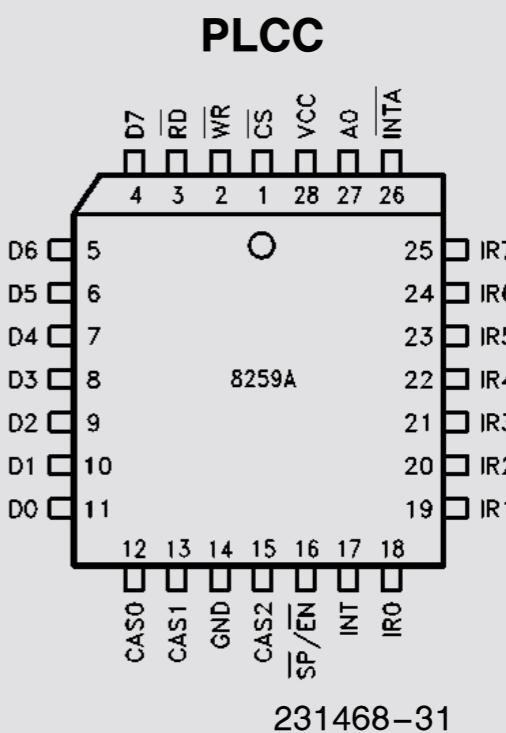
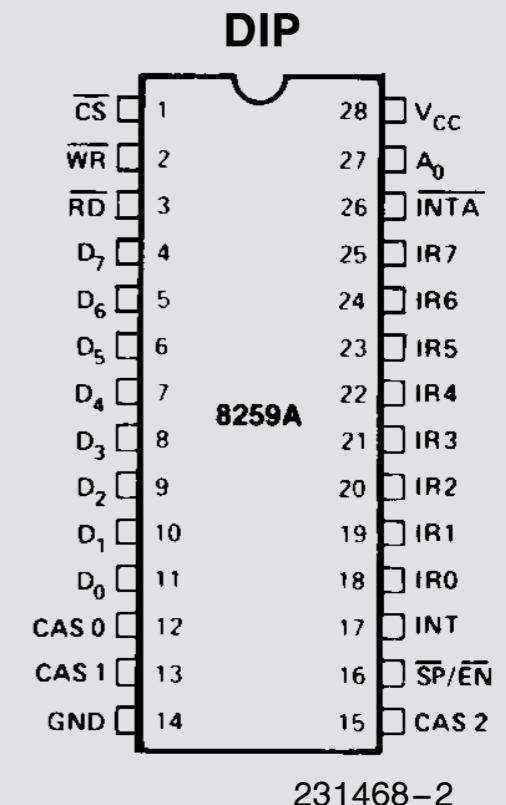


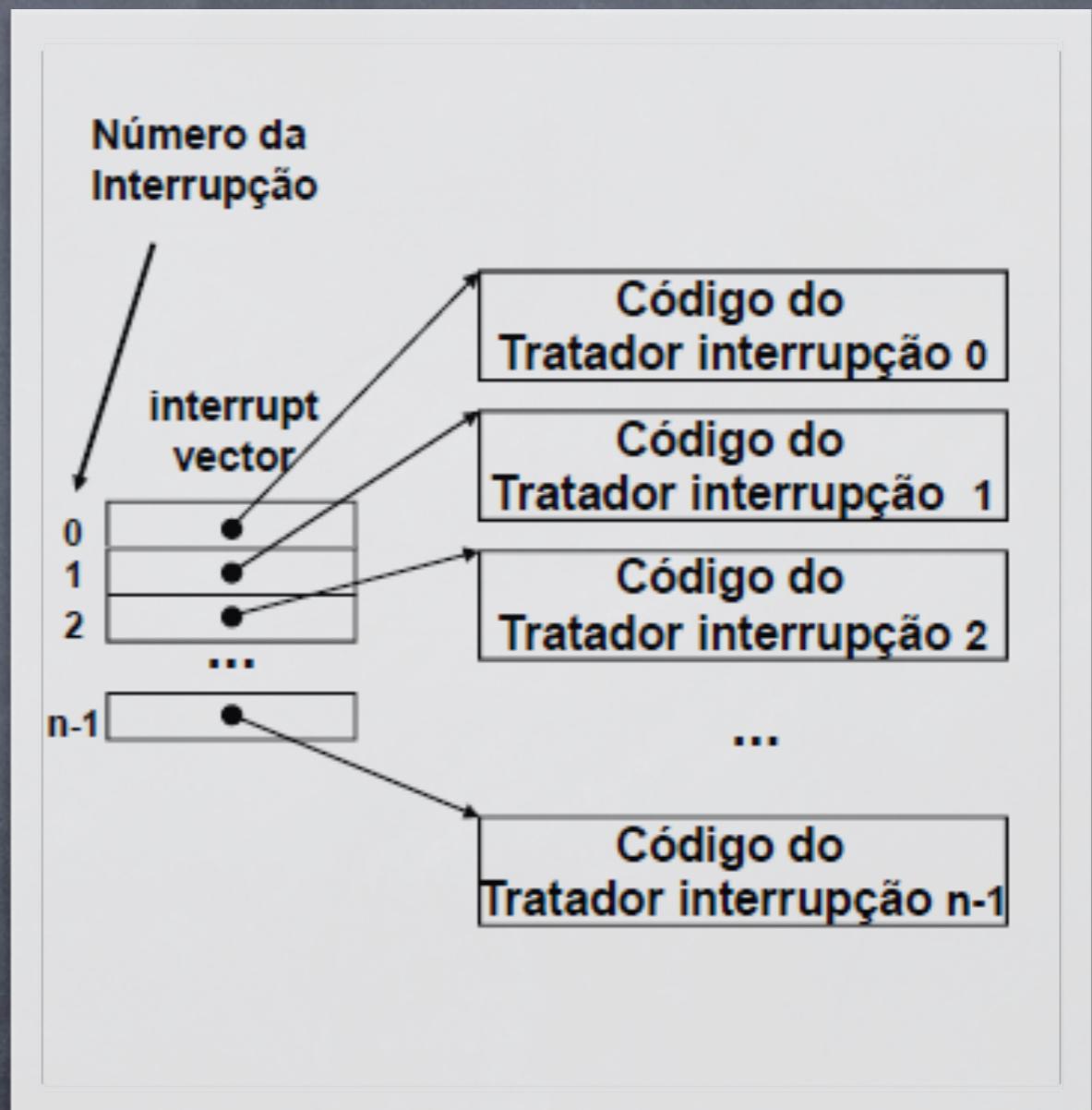
Figure 2. Pin Configurations

Interrupções assíncronas

- São geradas por algum dispositivo externo à CPU
- Ocorrem independentemente das instruções sendo executadas pela CPU
- CPU interrompe o processo; executa um tratador de interrupção; e posteriormente volta a executar o programa interrompido
- Exemplos:
 - Interrupção de relógio
 - Interrupção de I/O:
 - Controlador de disco avisa que leitura ou escrita de um bloco terminou
 - Teclado sinaliza um CTRL-C

Vetor de interrupções

- Cada tipo de interrupção possui um número k , que é o índice para uma entrada do vetor de interrupção
- Cada entrada aponta para o endereço do tratador de interrupção.



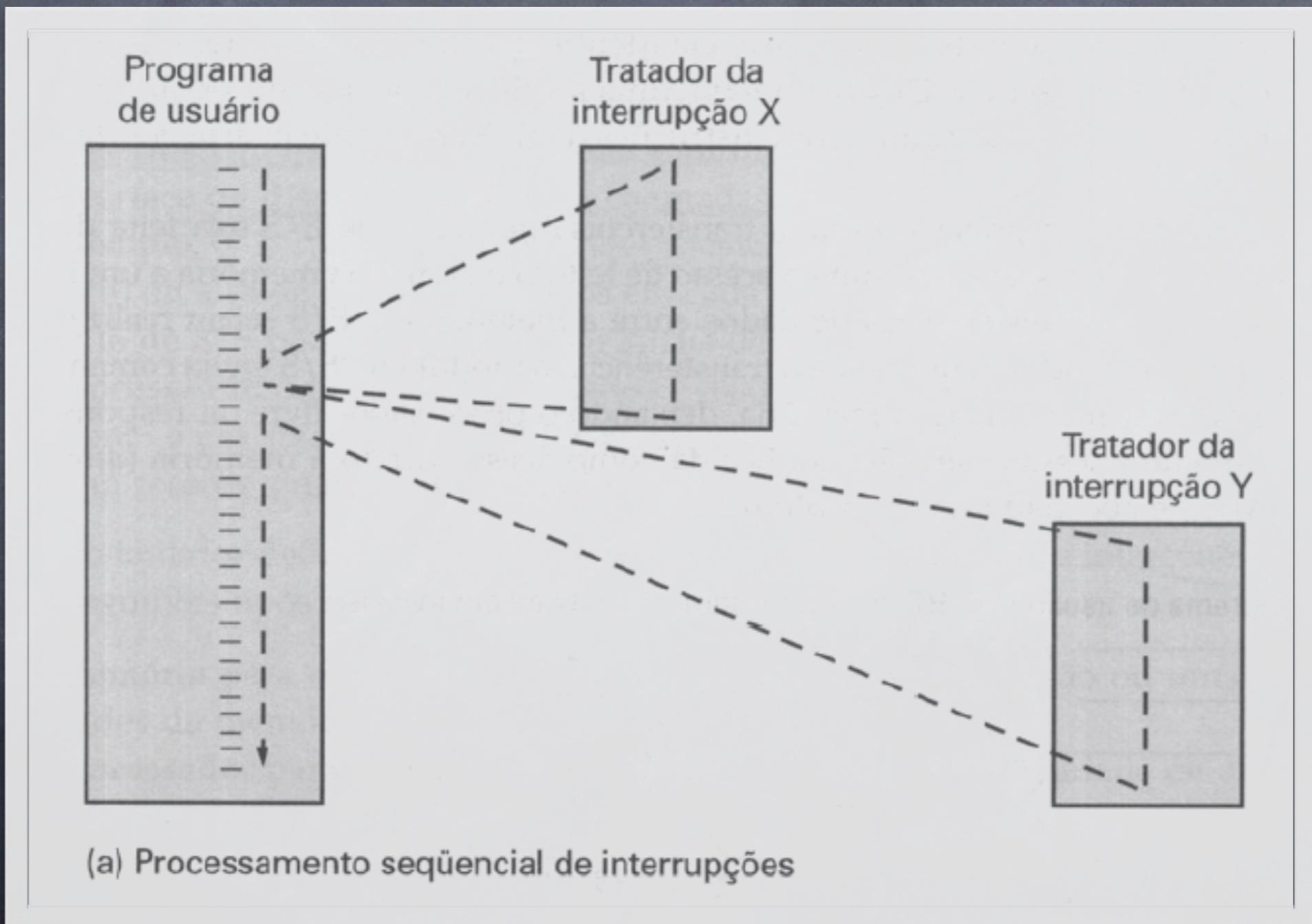
Troca de contexto

- Quando a CPU detecta que ocorreu uma interrupção
 - Aguarda o término da instrução de máquina corrente
 - Salva o contexto de execução do processo (IP, EFLAGS, registradores, pilha de ponto flutuante)
 - O conteúdo dos registradores da CPU são salvos pelo próprio tratador de interrupção
 - Antes de reiniciar o processo, o contexto é restaurado, e o IP recebe o endereço da próxima instrução

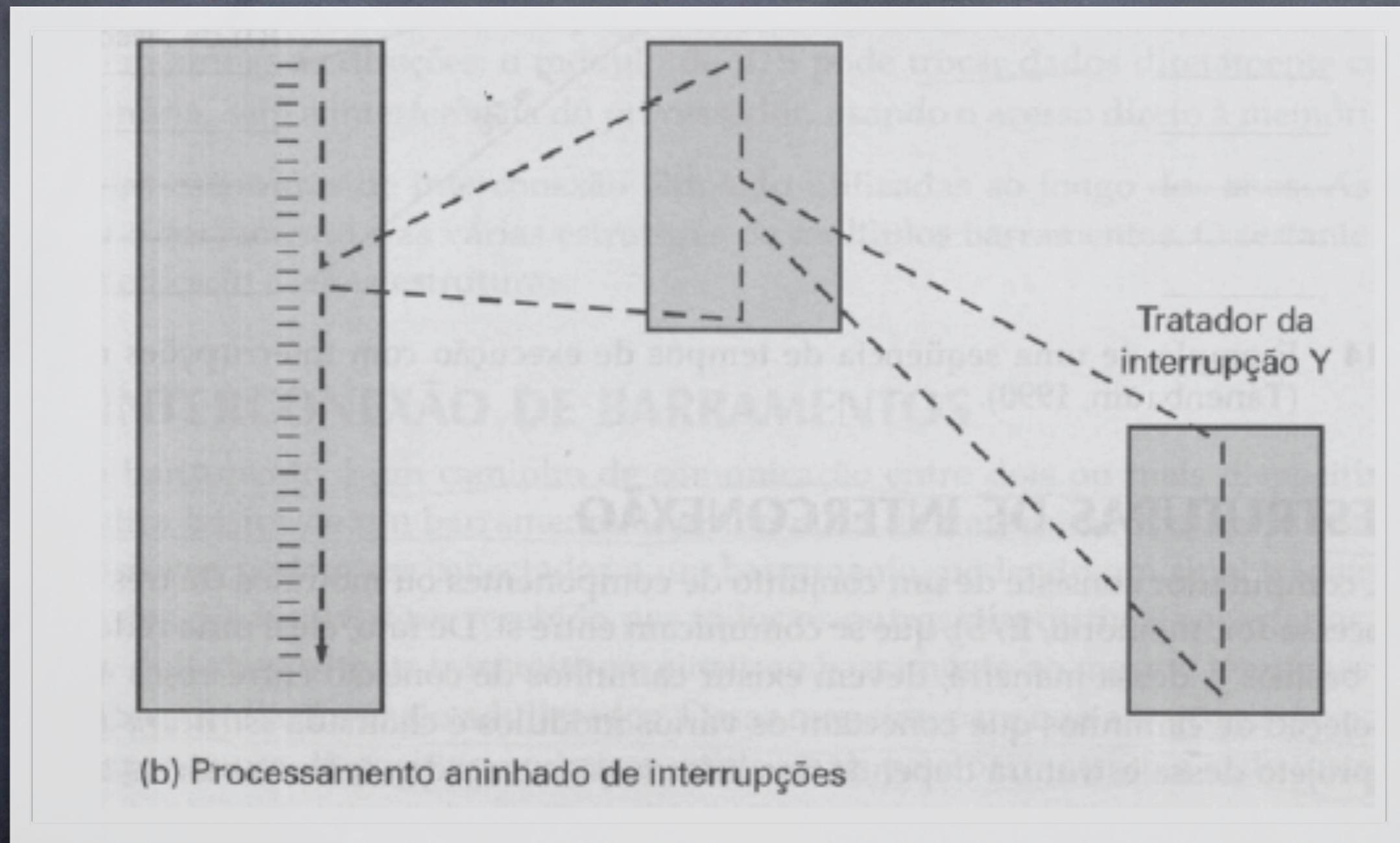
Interrupções múltiplas

- Duas abordagens no tratamento de interrupções múltiplas:
 - A primeira é desligar as interrupções enquanto uma interrupção está sendo processada
 - Quando o tratador termina de ser executado, as interrupções são novamente habilitadas, antes que o controle para o processo do usuário
 - A CPU verifica se ocorreu outra interrupção
- A desvantagem desta abordagem é que ela não leva em consideração prioridades relativas ou requisitos críticos de tempo.

Processamento sequencial de interrupções



Processamento aninhado de interrupções



Interrupções síncronas

- Ocorrem em consequência da instrução sendo executada
- “Traps/Exceptions”
 - Intencionais (controle retorna para próxima instrução)
 - Exemplos: system calls, breakpoints, instruções especiais
- “Faults” (Falhas)
 - Não intencionais, mas possivelmente o sistema poderá tratar a falha (reexecuta a instrução corrente, ou aborta processo)
 - Exemplos: page faults (tratável), falhas gerais de proteção (não tratável).
- “Aborts”
 - Não intencionais e não tratáveis (aborta processo)
 - Exemplos: erros de paridade, overflow aritmético, estouro da pilha.

Memory management Unit (MMU)

- Memory Management Unit (MMU) é um componente de hardware que intercepta qualquer acesso à memória
- Traduz endereços virtuais para endereços físicos de memória (RAM) de acordo com uma tabela de páginas (por processo)
- Quando a página de memória ainda não está em RAM, gera-se um “page-fault” e o gerenciador de memória do S.O. deve trazer a página para a memória.

“Page fault”

- Escrita em parte da memória que não está em RAM

```
int a[2048];
int main()
{
    a[1024]=13;
}
```

```
80483b7: c7 05 10 9d 04 08 0d      movl    $0xd, 0x8049d10
```

- Paginador carrega página correspondente para memória
- Retorna para instrução faltosa e a reexecuta

“Segmentation fault”

- Acesso a endereço de memória que não é válido.
- Paginador detecta endereço inválido de página, e envia sinal **SIGSEG** para o processo
- Processo faz `exit()` e gera erro “Segmentation Fault”

```
int a[2048];
void main()
{
    a[5000] = 13;
}
```

```
80483b7: c7 05 60 e3 04 08 0d      movl    $0xd, 0x804e360
```

Interrupções de software

- Interrupções por software são instruções que chamam as mesmas rotinas de atendimento anteriores
- Nesse caso, a interrupção é solicitada pelo software que está sendo processado pela CPU, em vez de um hardware externo à CPU
- Quando invocada, a rotina de interrupção funciona da mesma forma que uma interrupção vetorizada.
- Exemplos:
 - Comando **INT**, na arquitetura Intel.
 - Comando **SYSCALL**, na arquitetura MIPS
 - Comando **RAISE**, na arquitetura Blackfin.

Interrupções e o kernel

- O kernel deve ter privilégio em suas operações, e **não deve** confiar nos processos do usuário.
- Deve proteger:
 - O kernel de processos do usuário
 - Processos de outros processos
- As interrupções de software (“traps”) são utilizadas como um mecanismo de proteção
 - API de funções para processos do usuário.

Traps como provedores de serviços privilegiados

- Acesso à recursos do Sistema Operacional (DOS, Linux, etc.)
- Para acessar syscalls do Linux, use:
 - int 80h
- Para acessar syscalls do DOS, use:
 - int 21h
- Interrupções da BIOS
 - Para acessar funcionalidades da BIOS, use:
 - int 10h, int 13h etc

Término de processo

- sys_exit()

Linux

```
mov eax, 1 ; sair do programa (sys_exit)
mov ebx, 0 ; código de retorno = 0 (sem erro)
int 80h    ; chama o kernel
```

DOS

```
mov ah, 4Ch ; sair do programa (sys_exit)
mov al, 0   ; código de retorno = 0 (sem erro)
int 21h    ; chama o DOS
```

Mecanismos de proteção usados pelo SO

- ❑ Dual model of operation
 - Privileged (+ non-privileged) operations in kernel mode
 - Non-privileged operations in user mode
- ❑ Memory protection
 - Segmentation and paging
 - E.g., kernel sets **page table** when creating process
- ❑ Timer interrupt
 - Kernel periodically gets back control

Operações privilegiadas

- ❑ Read raw keyboard input
- ❑ Call printf()
- ❑ Call write()
- ❑ Write global descriptor table
- ❑ Divide by 0
- ❑ Set timer interrupt handler
- ❑ Set segment registers
- ❑ Load cr3

Modos de proteção do x86

- Four modes (0-3), but often only 0 & 3 used
 - Kernel mode: 0
 - User mode: 3
 - "Ring 0", "Ring 3"
- Segment has **Descriptor Privilege Level (DPL)**
 - DPL of kernel code and data segments: 0
 - DPL of user code and data segments: 3
- **Current Privilege Level (CPL)** = current code segment's DPL
 - Can only access data segments when $CPL \leq DPL$

Visão da interrupção pelo SO

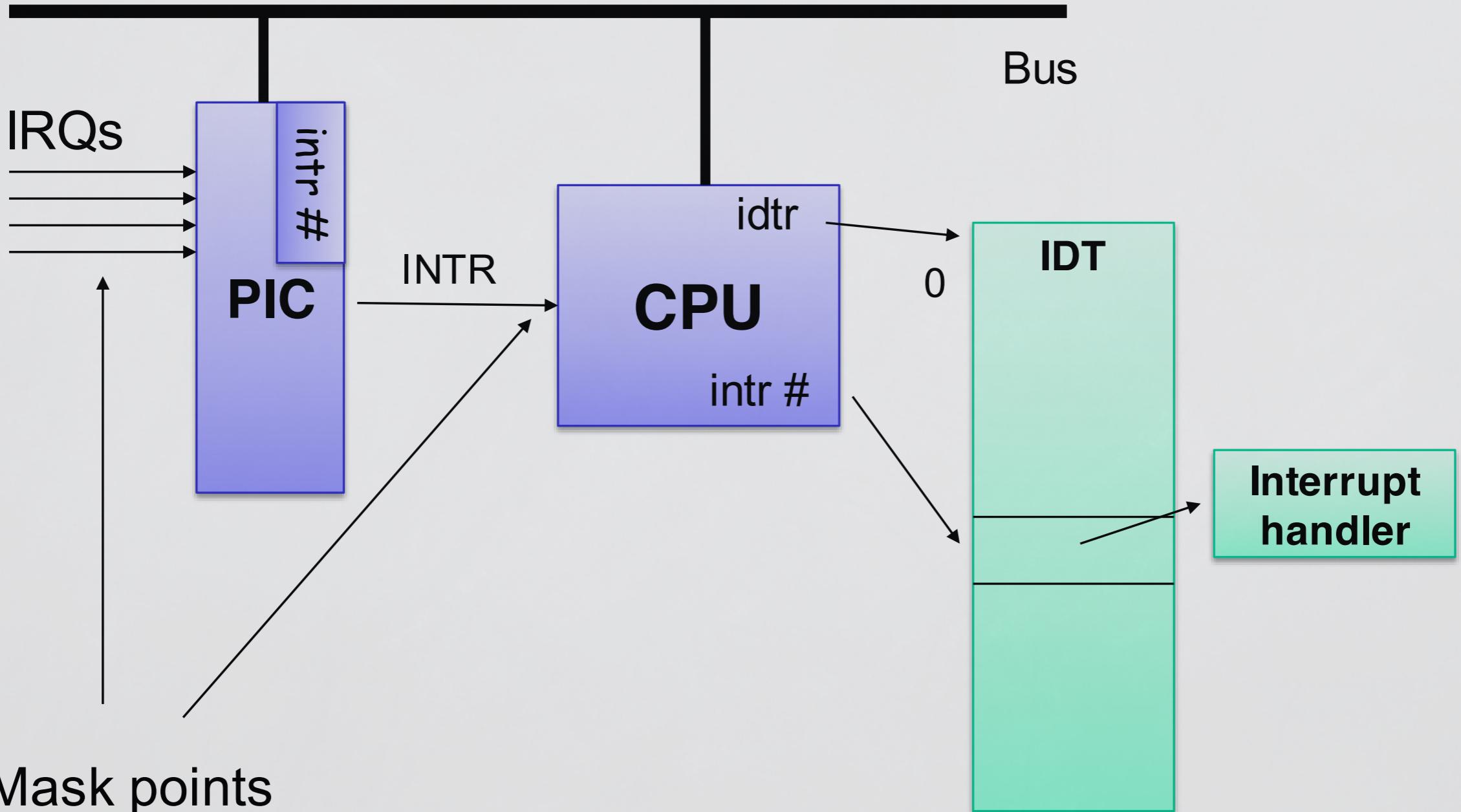
```
while (fetch next instruction) {  
    run instruction;  
    if (there is an interrupt) {  
        switch to kernel stack if necessary  
        save CPU context and error code if any  
        find OS-provided interrupt handler  
        jump to handler  
        restore CPU context when handler returns  
    }  
}
```

- Q1: how does hardware find OS-provided interrupt handler?
- Q2: why switch stack?
- Q3: what CPU context to save and restore?
- Q4: what does handler do?

1) Como achar o tratador da interrupção ?

- Hardware maps interrupt type to interrupt number
- OS sets up Interrupt Descriptor Table (IDT) at boot
 - Also called interrupt vector
 - IDT is in memory
 - Each entry is an interrupt handler
 - OS lets hardware know IDT base
 - Defines all kernel entry points
- Hardware finds handler using interrupt number as index into IDT
 - $\text{handler} = \text{IDT}[\text{intr_number}]$

Interrupções no x86



Enumeração das interrupções

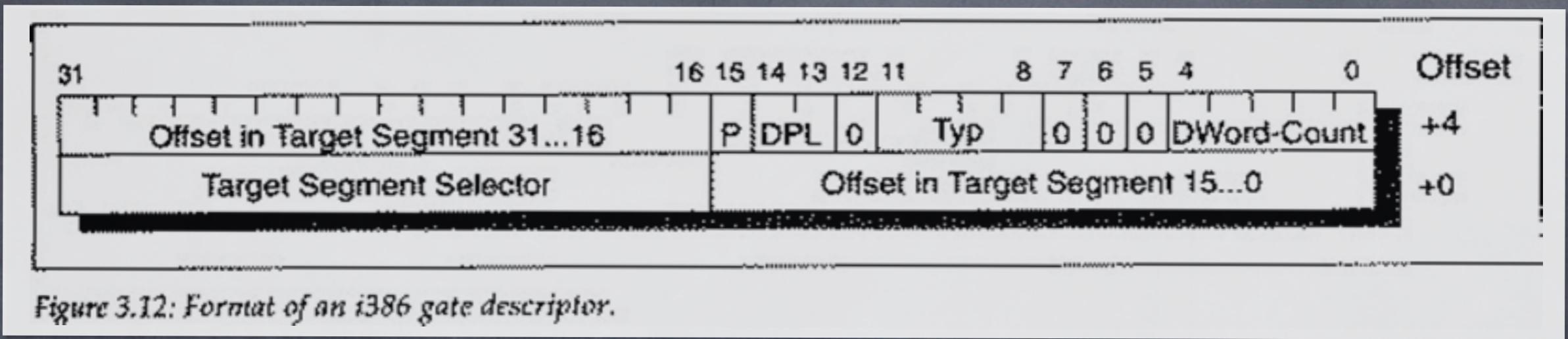
- Total 256 number [0, 255]
- Intel reserved first 32, OS can use 224
 - 0: divide by 0
 - 1: debug (for single stepping)
 - 2: non-maskable interrupt
 - 3: breakpoint
 - 14: page fault

```
// x86 trap and interrupt constants.

// Processor-defined:
#define T_DIVIDE          0      // divide error
#define T_DEBUG           1      // debug exception
#define T_NMI             2      // non-maskable interrupt
#define T_BRKPT           3      // breakpoint
#define T_OFLOW            4      // overflow
#define T_BOUND            5      // bounds check
#define T_ILLOP            6      // illegal opcode
#define T_DEVICE           7      // device not available
#define T_DBLFLT           8      // double fault
// #define T_COPROC          9      // reserved (not used since 486)
#define T_TSS              10     // invalid task switch segment
#define T_SEGNP            11     // segment not present
#define T_STACK             12     // stack exception
#define T_GPFLT            13     // general protection fault
#define T_PGFLT            14     // page fault
// #define T_RES              15     // reserved
#define T_FPERR            16     // floating point error
#define T_ALIGN             17     // alignment check
#define T_MCHK              18     // machine check
#define T SIMDERR           19     // SIMD floating point error

// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL           64     // system call
#define T_DEFAULT            500    // catchall
```

Interrupt descriptor table (IDT)



- Interrupt gate descriptor
 - Code segment selector and offset of handler
 - Descriptor Privilege Level (DPL)
 - To invoke “int x” in software, must have $CPL \leq DPL$
 - Trap or exception flag. If exception, hardware clears the IF flag in EFLAGS to disable further maskable interrupts
- lidt instruction loads CPU with IDT base
- xv6
 - Handler entry points: vectors.S
 - Interrupt gate format: SETGATE in mmu.h
 - IDT initialization: tvinit() & lidt() in trap.c

trap.c

```
void
tvinit(void)
{
    int i;

    for(i = 0; i < 256; i++)
        SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);

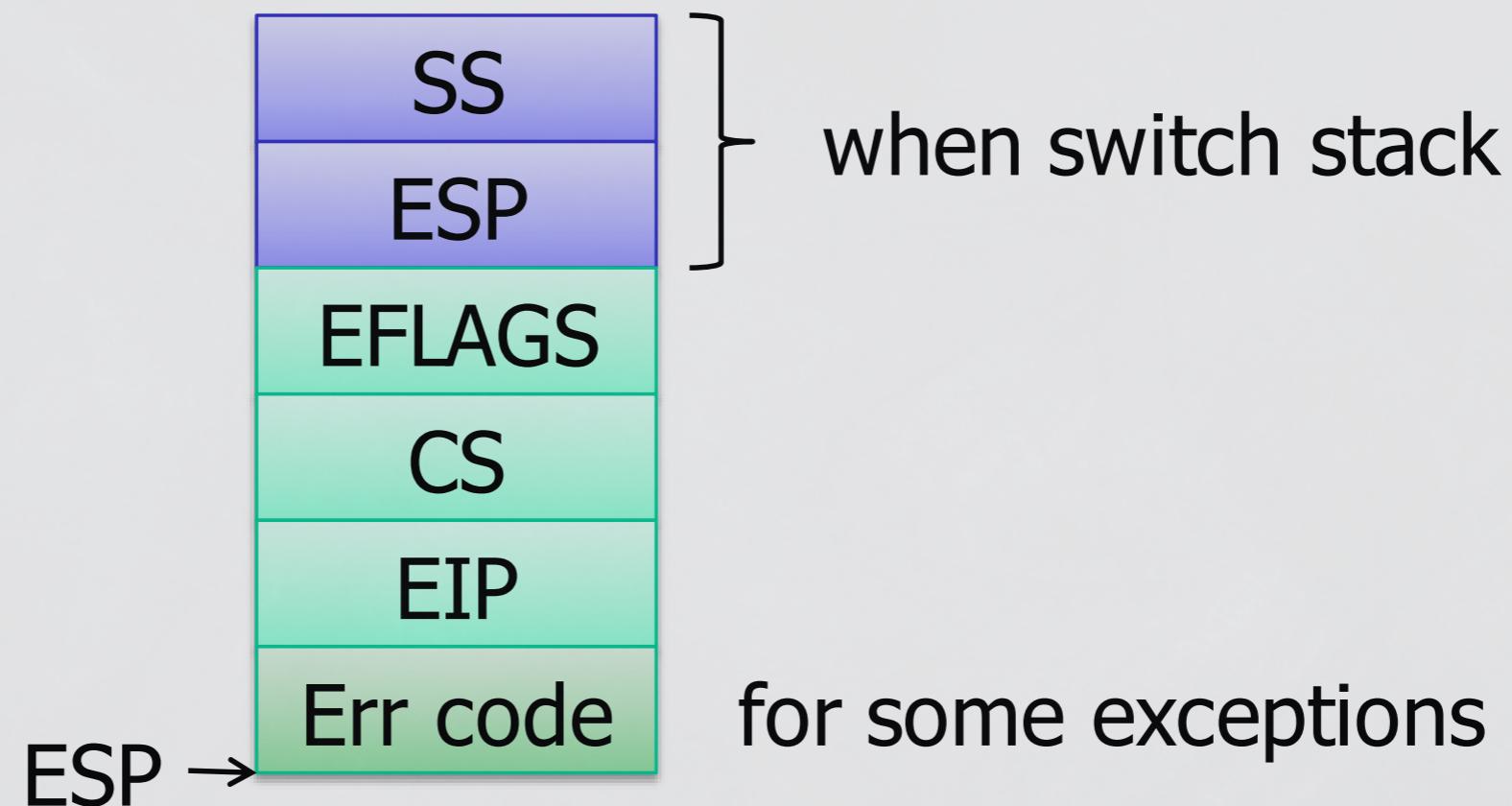
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors
[T_SYSCALL], DPL_USER);

    initlock(&tickslock, "time");
}
```

2) Por que trocar a pilha ?

- Cannot trust stack of user process!
- x86 hardware switches stack when interrupt handling requires user-kernel mode switch
 - That is, when CPL <= DPL of handler's code segment
- Where to find kernel stack?
 - task gate descriptor has SS and ESP for interrupt
 - Itr loads CPU with task gate descriptor
- xv6 uses current process's kernel stack
 - switchuvm() in vm.c

3) Que contexto salvar ?



- x86 saves SS, ESP, EFLAGS, CS, EIP, Err code
- Restored by `iret`
- OS can save more context

4) Qual o papel do tratador da interrupção ?

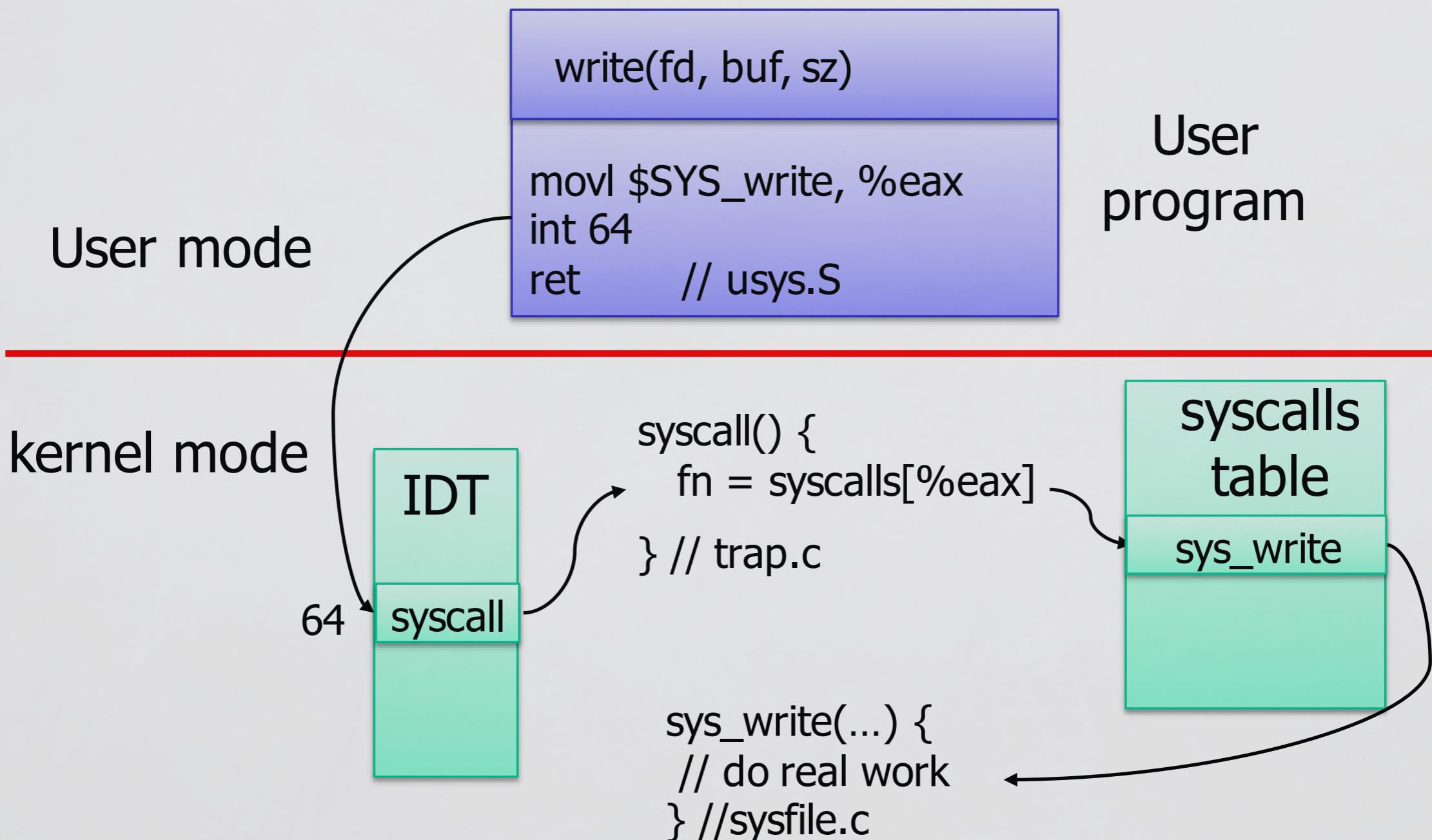
□ Typical steps

- Assembly to save additional CPU context
- Invoke C handler to process interrupt
 - E.g., communicate with I/O devices
- Invoke kernel scheduler
- Assembly to restore CPU context and return

□ xv6

- Interrupt handler entries: `vectors.S`
- Saves & restore additional CPU context: `trapasm.S`
- C handler: `trap.c`, *struct trapframe* in `x86.h`

Despacho de interrupções no xv6



Convenções do xv6

- Usually a library function `foo()` will do some work and then call a system call `sys_foo()`
 - `sys_foo()` implemented in `sys*.c`
 - Often wrappers to `foo()` in kernel
- System call number for `foo()` is `SYS_foo`
 - `syscalls.h`

Parâmetros de syscalls

- Typical methods
 - Pass via registers (e.g., Linux)
 - Pass via user-mode stack (e.g., xv6)
 - Pass via designated memory region
- xv6 system call parameter passing
 - Arguments pushed onto user stack based on gcc calling convention
 - Kernel function uses special routines to fetch these arguments
 - `syscall.c`

Exercício

- No xv6:
 - Mostre todas a cascata de chamdas (interrupções e funções) decorrentes de o uso de um comando “write()”
 - Liste quais os passos necessários para a implementação de uma nova syscall no kernel do xv6.
 - Quais arquivos deverão ser modificados ?