

# Exceções no Fluxo de Execução: Interrupções e Traps

# Fluxo de Controle

- Fluxo de controle de um programa:

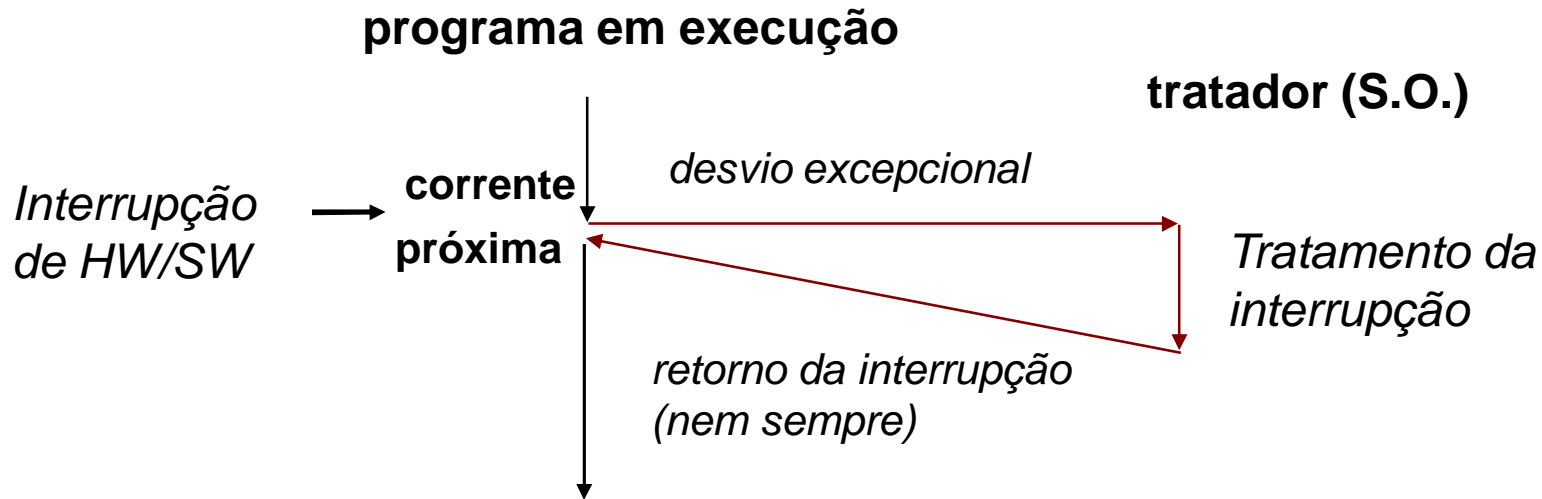
$a_0 \ a_1 \ a_2 \ \dots \ a_n \ \rightarrow$  sequência de endereços  
 $\downarrow \ \downarrow \ \downarrow \ \dots \ \downarrow$   
 $l_0 \ l_1 \ l_2 \ \dots \ l_n \ \rightarrow$  sequência de instruções

- O fluxo mais simples é a **execução sequencial**
- Dois mecanismos alteram esse fluxo de controle:
  - desvios (jumps) e procedimentos (chamada e retorno)
- Um outro mecanismo que altera o fluxo de controle são **exceções** (ou **interrupções**)
  - quando ocorre uma interrupção a CPU interrompe o programa em execução e desvia o controle para um **tratador** (parte do SO)

# Tipos de Interrupções

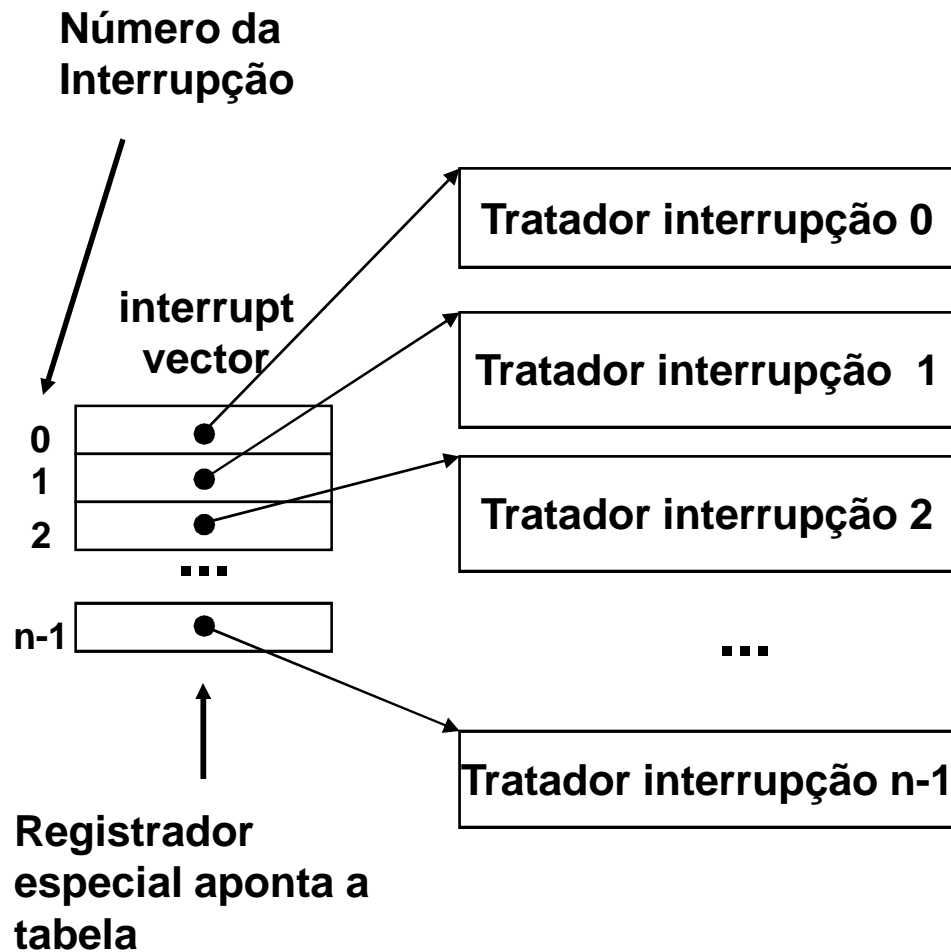
- O mecanismo de interrupções permite que o hardware “chame a atenção” da CPU para um evento
  - esse evento precisa ser tratado rapidamente!
- Interrupções geradas por dispositivos externos são ditas **assíncronas** porque independem da instrução que a CPU está executando no momento da interrupção
  - teclado, controlador de disco, interface de rede, *timer*
- Interrupções **síncronas** (ou **traps**) ocorrem em consequência da instrução sendo executada
  - geradas explicitamente pelo programa: chamadas ao SO
  - geradas pelo hardware: falhas (**faults**)

# Alteração no fluxo de controle



- A CPU interrompe o programa em execução e desvia o controle para um **tratador** (parte do SO)
  - esse tratador executa em modo “kernel” (privilegiado)
- Cada tipo de interrupção é associado a um identificador (número)
- Uma tabela (vetor de interrupções) contém descritores dos tratadores de interrupções (que incluem seu endereço)
  - o número da interrupção é usado como índice nessa tabela`

# Vetor de Interrupções



- Durante a inicialização do sistema o SO preenche a tabela com os descritores dos tratadores e inicializa um registrador que aponta essa tabela
- Em tempo de execução, o hw consulta a tabela para realizar o desvio para o tratador de uma interrupção
- Alguns dos números de interrupção são específicos da plataforma/processador
  - page fault, div. zero, break point

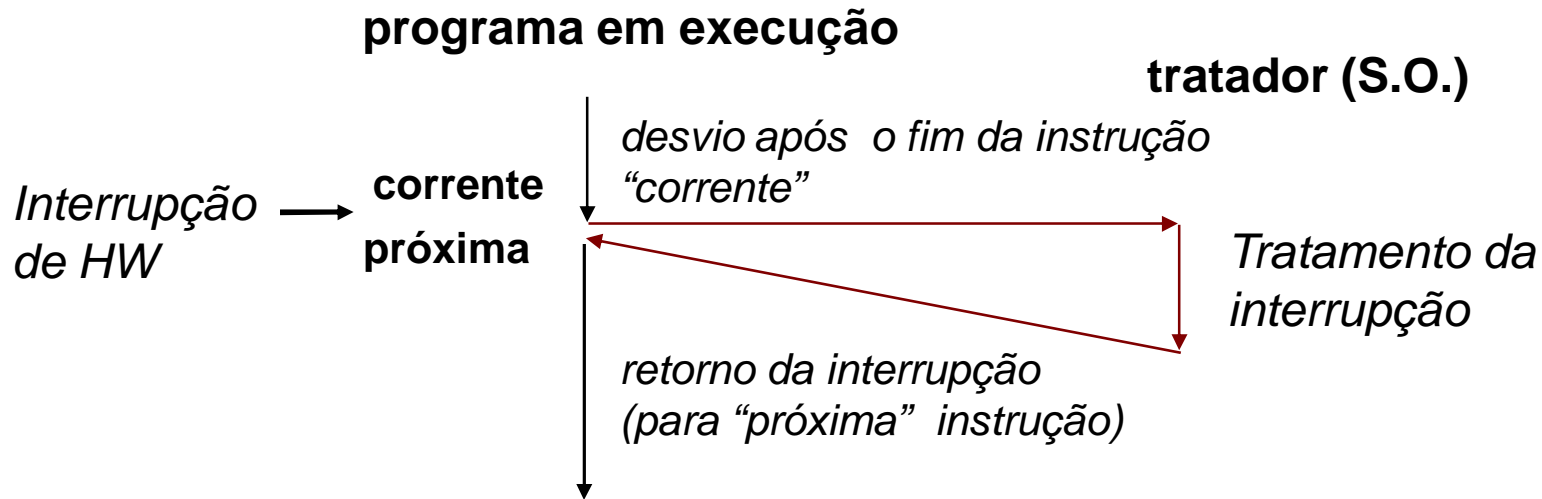
# Salvamento de Contexto

- Para que a execução do programa interrompido possa ser retomada normalmente (transparentemente quando for o caso) é necessário salvar o seu **contexto de execução**
  - o hw salva o **program counter** e outros **registradores de estado** (por ex. EFLAGS) antes de transferir o controle ao tratador
    - em algumas arquiteturas em uma pilha independente !
  - registradores de dados, se usados pelo **tratador da interrupção**, devem ser salvos e restaurados ao final do tratamento
- Ao terminar sua execução, o tratador restaura o contexto de execução do programa interrompido através de uma instrução (privilegiada) especial
  - restaura (pop) os registradores de estado e o *program counter* e transfere o controle de volta ao programa
  - se um código em *user mode* foi interrompido, esse modo de execução é restaurado

# Interrupções Assíncronas

- São geradas por algum dispositivo externo à CPU
  - ocorrem **independentemente** da instrução sendo executada
  - o controlador do dispositivo “sinaliza” a CPU e envia através do barramento o número da interrupção correspondente
- O tratador de interrupção é similar a um procedimento
  - mas sua execução é **transparente** para o programa interrompido e por isso não há qualquer comunicação (parâmetros, retorno) entre eles!
- Exemplos:
  - Interrupção de relógio (*timer*)
  - Interrupção de I/O: fim de leitura no disco, acionamento do teclado, recepção de um pacote pela interface de rede

# Tratamento de interrupção

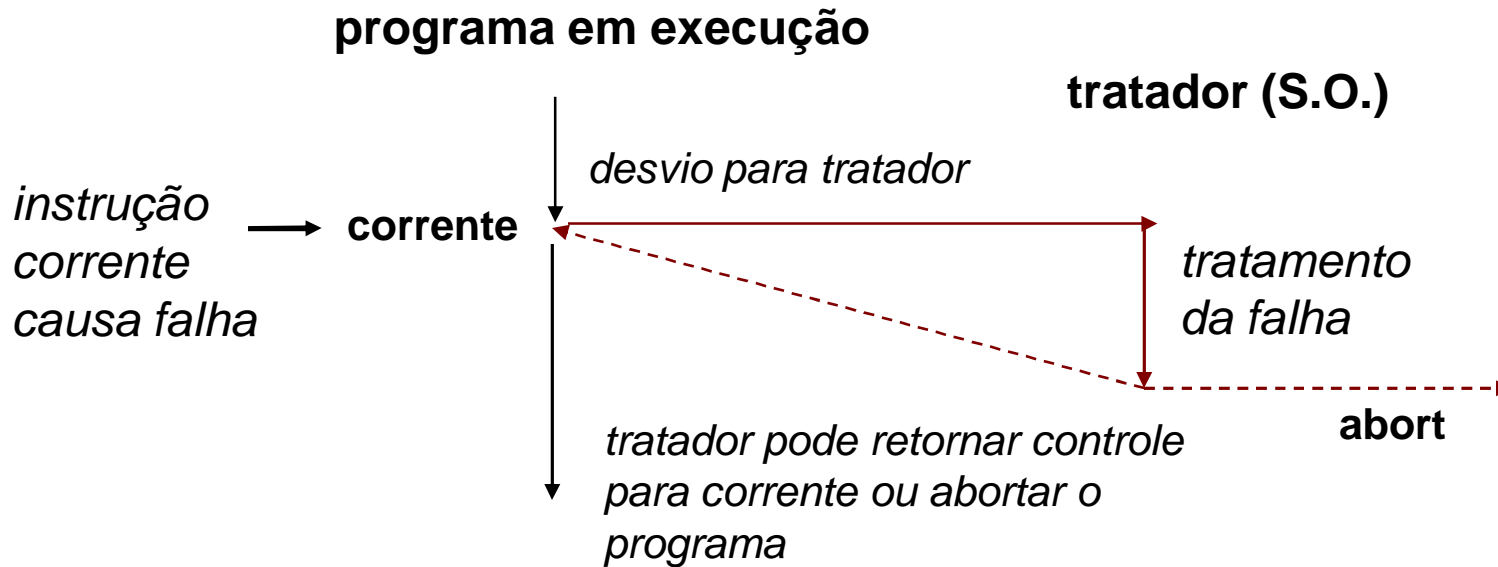


- Após o tratamento da interrupção o programa continua sua execução como se a interrupção não houvesse ocorrido
  - a menos de situações especiais, como o acionamento de ctrl-C!

# Interrupções Síncronas (traps)

- Consequência da instrução sendo executada.
- Geradas pelo hardware (falhas)
  - Erros causados pela execução da instrução
    - o programa não pode prosseguir: SO termina sua execução
    - divisão por zero, acesso não permitido à memória (GPF)
  - Page fault (memória virtual)
    - página acessada (endereço virtual) não está na memória principal
    - o tratador traz a página do disco e retorna o controle à instrução que causou a falha (re-executa a instrução!)
- Geradas explicitamente por instruções do programa
  - A instrução `int n` gera um *trap* (interrupção 'n')
  - Breakpoints: instruções especiais usadas por *debuggers*
  - Chamadas ao Sistema Operacional (`int $0x80`)

# Tratamento de falhas



- Se o tratador pode corrigir a falha (ex. page fault) o controle retorna para a instrução “corrente” (re-execução)
- Se a falha não pode ser corrigida, o tratador transfere o controle para o SO (**abort**), que termina o programa (ex. **seg fault**)

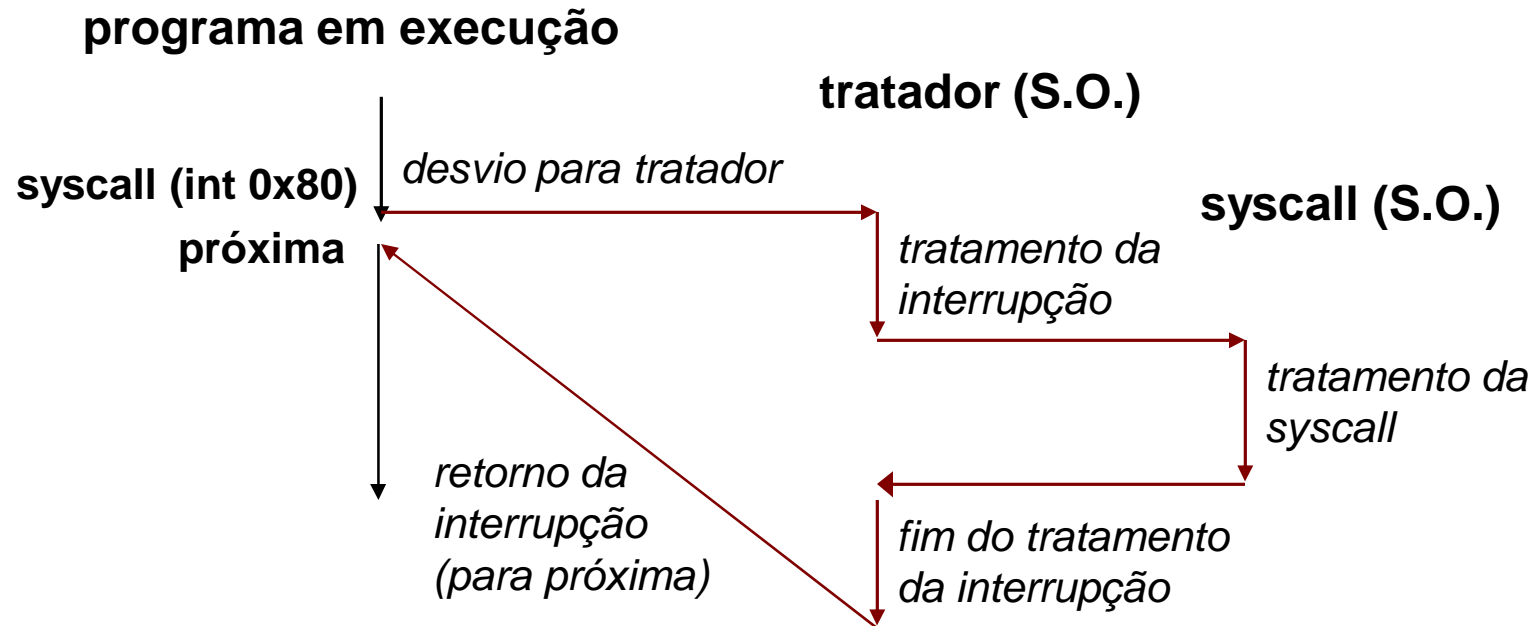
# Chamadas ao Sistema Operacional

- Permitem que um programa “acione” o SO para requisitar um serviço (por exemplo, E/S)
  - programa não pode “chamar” uma função do SO porque o SO executa em seu próprio espaço de endereçamento e em modo privilegiado
- Através de uma interrupção de sw o programa ativa um tratador (da `int $0x80`) que encaminha a chamada (**syscall**) ao SO
  - cada chamada ao sistema (serviço a ser executado) possui um identificador numérico e pode receber parâmetros e retornar um valor
  - a biblioteca padrão de C provê funções *wrapper* para a maioria das system calls

# Interface com syscalls

- O código (número) da chamada deve ser colocado no registrador **%eax**
  - O arquivo **<sys/syscall.h>** tem os códigos das chamadas ao sistema
- Os valores dos argumentos devem ser colocados nos registradores **%ebx, %ecx, %edx, %esi, %edi**
  - não se usa a pilha para passagem de parâmetros !
- O valor de retorno da chamada vem em **%eax**
  - geralmente indica se operação foi bem sucedida (0)
- O controle retorna à instrução seguinte a **int**

# Tratamento de syscall



- Programa envia código da syscall em %eax e parâmetros nos outros registradores
- O controle é transferido de volta ao programa (para “*próxima*”) com o resultado da chamada em %eax

# Exemples de syscalls

<code>%eax</code>	nome	<code>%ebx</code>	<code>%ecx</code>	<code>%edx</code>	retorno ( <code>%eax</code> )
1	<code>sys_exit</code>	<code>int status</code>	-	-	-
3	<code>sys_read</code>	<code>int fildes</code>	<code>void *buf</code>	<code>size_t nbyte</code>	<code>ssize_t</code> ou -1
4	<code>sys_write</code>	<code>int fildes</code>	<code>void *buf</code>	<code>size_t nbyte</code>	<code>ssize_t</code> ou -1
5	<code>sys_open</code>	<code>char *path</code>	<code>int oflag</code>	<code>mode_t mode</code>	<code>int (fildes)</code> ou -1
6	<code>sys_close</code>	<code>int fildes</code>			<code>int (0 ou -1)</code>
19	<code>sys_lseek</code>	<code>int fildes</code>	<code>off_t</code> <code>offset</code>	<code>int whence</code>	<code>off_t</code> ou -1

# Exemplo de chamada

```
int f (...) {  
    ...  
  
    if (...)  
        exit (2);  
  
    ...  
}
```

```
f:  
    pushl %ebp  
    movl %esp, %ebp  
    ...  
  
    {  
        movl $1,%eax /* sys_exit */  
        movl $2,%ebx  
        int $0x80  
    }  
  
    ...
```