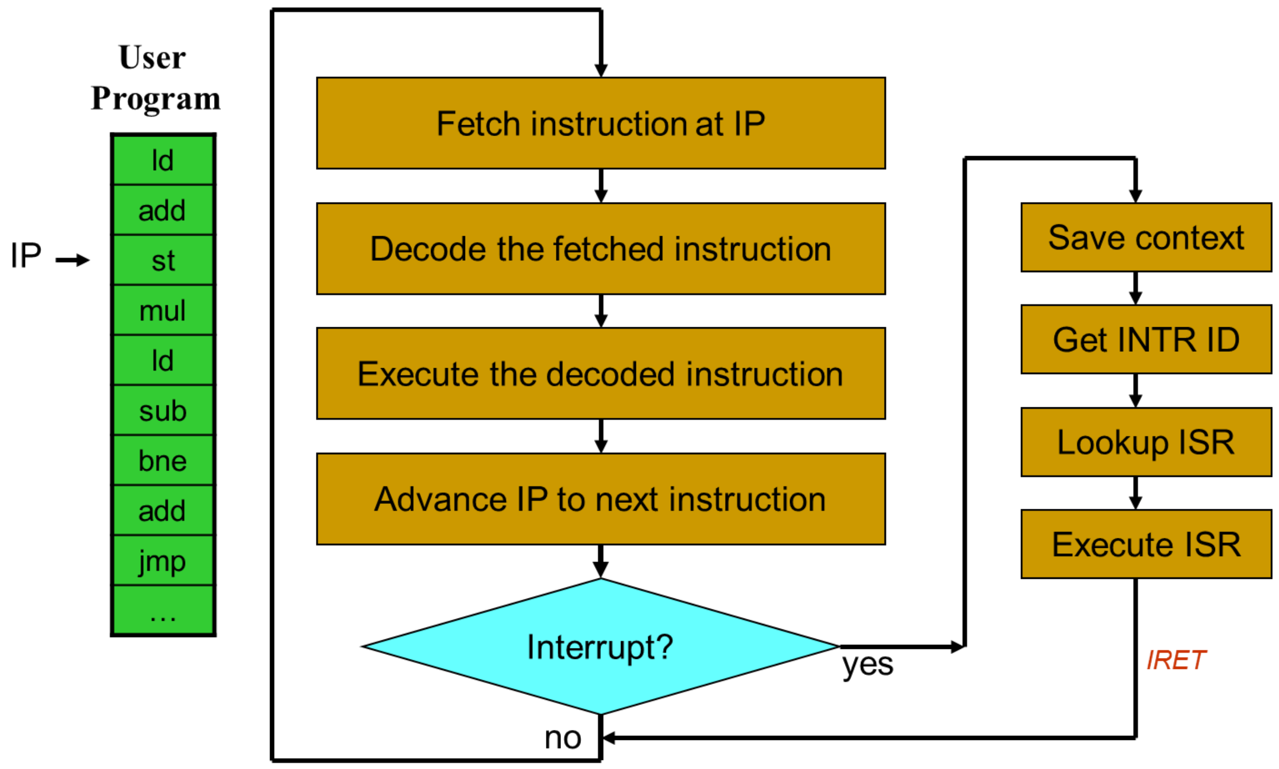
Protected-Mode Interrupt Processing

Introdução

Motivação:

- Utility of a general-purpose computer depends on its ability to interact with I/O devices attached to it (e.g., keyboard, display, disk-drives, network, etc.)
- Devices require a prompt response from the CPU when various events occur, even when the CPU is busy running a program
- Need a mechanism for a device to "gain CPU's attention"
- Interrupts provide a way doing this

CPU's 'fetch-execute' cycle:



Types of Interrupts:

- Asynchronous
  - From external source, such as I/O device
  - Not related to instruction being executed

- Synchronous (also called exceptions)
  - Processor-detected exceptions:
    - Faults — correctable; offending instruction is retried
    - Traps — often for debugging; instruction is not retried
    - Aborts — major error (hardware failure)

  - Programmed exceptions:
    - Requests for kernel intervention (software intr/syscalls)

Interrupção é um mecanismo que permite alterar o fluxo de controle de um programa, similar aos mecanismos já estudados de chamada de procedures e jumps.

Os jumps proporcionam uma transferência de controle sem retorno ao ponto de chamada.

As procedures proporcionam um mecanismo que retorna o controle ao ponto da chamada após o término da procedure chamada.

Interrupções possuem um mecanismo similar ao da chamada de procedure.

A interrupção transfere o controle para uma procedure denominada interrupt service routine (ISR).

Uma ISR é também chamada de $handler$.

Quando a ISR é completada, o programa interrompido retorna a execução como se não tivesse sido interrompida.

Uma das principais diferenças é que as interrupções podem ser iniciadas tanto por software quanto por hardware.

Diferentemente, as procedures são exclusivamente iniciadas por softwares.

O fato de permitir ser iniciada por hardware dá um poder especial para as interrupções.

Esta característica fornece um meio eficiente para que dispositivos externos possam obter a atenção do processador.

Interrupções iniciadas por softwares, ou simplesmente interrupções de software, são geradas pela execução da instrução INT.

Estas interrupções, de modo similar ás procedures, são eventos previstos ou planejados. Um exemplo típico é o $Ctrl - C$ ou $Ctrl - Break$.

O mecanismo de interrupção fornece uma maneira eficiente para lidar com tais eventos imprevistos.

Referindo-se ao exemplo anterior, o $Ctrl - C$ pode fazer com que uma interrupção chame a atenção do processador independente do programa do usuário.

A ISR associada ao $Ctrl - C$ pode encerrar o programa e retornar o controle para o sistema operacional.

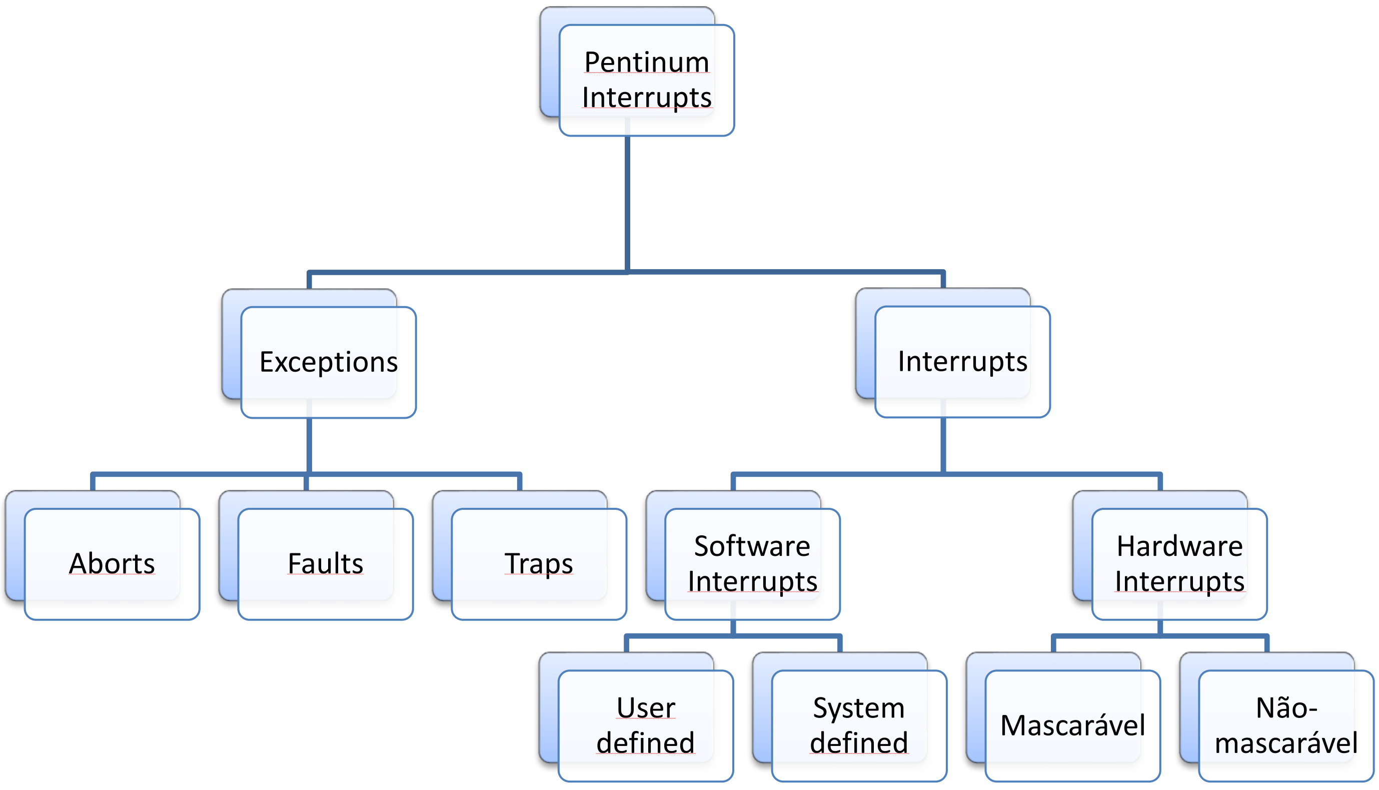
Outra diferença entre procedures e interrupções é que ISRs são normalmente residentes na memória.

As procedures são carregadas na memória junto com o programa aplicativo.

São usados números para identificar as interrupções em vez de nomes.

Interrupts	**Procedures**
• Initiated by both *software* and *hardware* • Can handle *anticipated* and *unanticipated* internal as well as external events • ISRs or interrupt handlers are memory resident • Use numbers to identify an interrupt service • (E)FLAGS register is saved automatically	• Can only be initiated by *software* • Can handle *anticipated* events that are coded into the program • Typically loaded along with the program • Use meaningful names to indicate their function • Do not save the (E)FLAGS register

Taxonomia das Interrupções no Pentium



Além das interrupções iniciadas por software e por hardware, há também as chamadas $exceptions$.

Um exemplo de $exception$ é a falta causada por erro de divisão por zero (div ou $idiv$ com divisor igual a zero).

Interrupções de Software são inseridas em um programa através do uso da instrução int. O principal uso das interrupções de software é acessar dispositivos de I/O tais como teclado, impressora, tela, disco, etc.

Interrupções de Software podem ser classificadas em $system - defined$ e $user - defined$.

Interrupções de Hardware são geradas por dispositivos de hardware quando precisam da atenção do processador.

Por exemplo, quando uma tecla é pressionada, o hardware do teclado gera uma interrupção externa fazendo com que o processador suspenda sua atividade atual e execute a ISR associada com o teclado a fim de que a tecla seja tratada.

Após encerrar a ISR do teclado, o processador retoma o fluxo a partir do ponto antes de ser interrompido.

Interrupções de Hardware podem ser tanto $maskable$ or $nonmaskable$ (NMI). O processador sempre atende as NMI imediatamente. Um exemplo de NMI é o erro de paridade da RAM, indicando mal funcionamento da memória.

$Maskable\ interrupts$ podem aguardar até que o fluxo de execução alcance um ponto conveniente.

Hardware Interrupts

- Software interrupts are synchronous events
    - Caused by executing the **int** instruction
- Hardware interrupts are of hardware origin and asynchronous in nature
    - Typically caused by applying an electrical signal to the processor chip
- Hardware interrupts can be
    - Maskable
    - Non-maskable
        - Causes a **type 2** interrupt
        - Pode ser desabilitada através de um flip-flop no bit 0 da porta 70h
        - Erro de paridade na memória
        - Erro no barramento ISA IOCHK#
        - Erro no co-processador
        - Erro de sistema no barramento PCI (SERR#)

How Are Hardware Interrupts Triggered?

- Non-maskable interrupt is triggered by applying an electrical signal to the MNI pin of processor
    - Processor always responds to this signal
    - Cannot be disabled under program control
- Maskable interrupt is triggered by applying an electrical signal to the INTR (INTerrupt Request) pin of Pentium
    - Processor recognizes this interrupt only if IF (interrupt enable flag) is set
    - Interrupts can be masked or disabled by clearing IF

How Does the CPU Know the Interrupt Type?

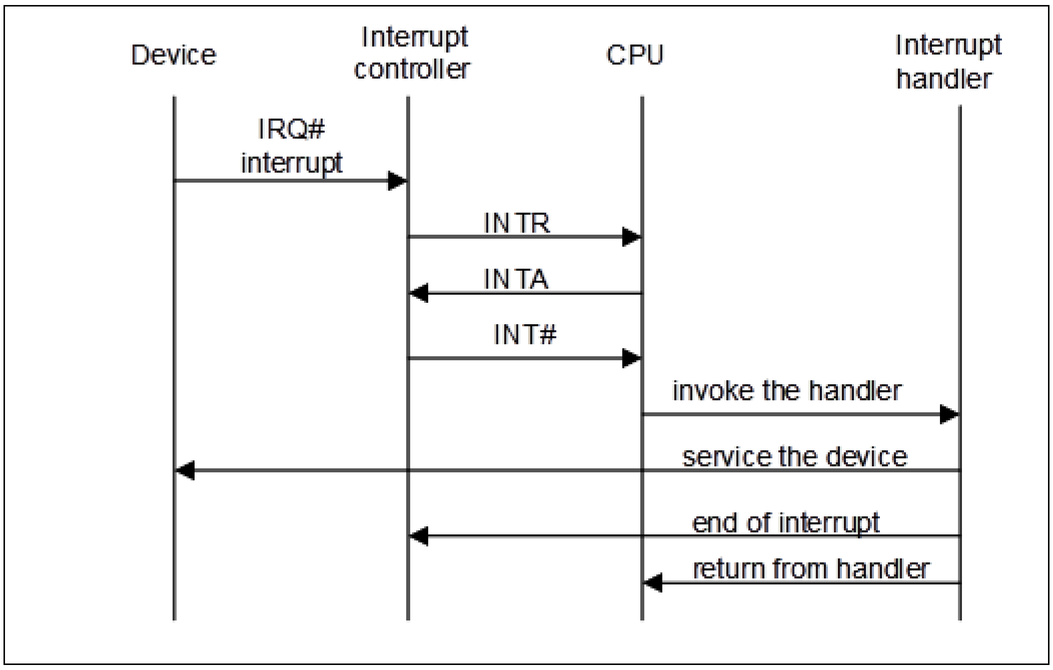
- Interrupt invocation process is common to all interrupts
    - Whether originated in software or hardware
- For hardware interrupts, processor initiates an interrupt acknowledge sequence
    - processor sends out interrupt acknowledge (INTA) signal
    - In response, interrupting device places interrupt vector on the data bus
    - Processor uses this number to invoke the ISR that should service the device (as in software interrupts)

How can More Than One Device Interrupt?

- Processor has only one INTR pin to receive interrupt signal
- Typical system has more than one device that can interrupt --- keyboard, hard disk,

floppy, etc.

- Use a special chip to prioritize the interrupts and forward only one interrupt to the CPU
  - ∗ 8259 Programmable Interrupt Controller chip performs this function (more details in Chapter 15)



Direct Control of I/O Devices

- Two ways of mapping I/O ports:
  - ∗ Memory-mapped I/O (e.g., Motorola 68000)
    - » I/O port is treated as a memory address (I/O port is mapped to a location in memory address space (MAS))
    - » Accessing an I/O port (read/write) is similar to accessing a memory location (all memory access instructions can be used)
  - ∗ Isolated I/O (e.g., Pentium)
    - » I/O address space is separate from the memory address space
      - – leaves the complete MAS for memory
    - » Separate I/O instructions and I/O signals are needed
    - » Can't use memory access instructions
    - » Can also use memory-mapped I/O and use all memory access instructions

Pentium I/O Address Space

- Pentium provides 64 KB of I/O address space
  - ∗ Can be used for 8-, 16-, and 32-bit I/O ports
- Combination cannot exceed the total I/O space
  - ∗ 64K 8-bit I/O ports
    - » Used for 8-bit devices, which transfer 8-bit data
    - » Can be located anywhere in the I/O space
  - ∗ 32K 16-bit I/O ports (used for 16-bit devices)

    - » 16-bit ports should be aligned to an even address
  - ∗ 16K 32-bit I/O ports (used for 32-bit devices)
    - » Should be aligned to addresses that are multiples of four
    - » Pentium supports unaligned ports, but with performance penalty
  - ∗ A combination of these for a total of 64 KB

Pentium I/O Instructions

- Pentium provides two types of I/O instructions:
  - ∗ Register I/O instructions
    - » used to transfer data between a register (accumulator) and an I/O port
    - » in - to read from an I/O port
    - » out - to write to an I/O port
  - ∗ Block I/O instructions
    - » used to transfer a block of data between memory and an I/O port
    - » ins - to read from an I/O port
    - » outs - to write to an I/O port

Register I/O Instructions

- Can take one of two forms depending on whether a port is directly addressable or not
  - – A port is said to be directly addressable if it is within the first 256 ports (so that one byte can be used specify it)
- To read from an I/O port

  in accumulator,port8 -- direct addressing format

  port8 is 8-bit port number

  in accumulator,DX -- indirect addressing forma

  port number should be loaded into DX

  accumulator can be AL, AX, or EAX (depending on I/O port)
- To write to an I/O port

  out port8,accumulator -- direct addressing format

  out DX,accumulator -- indirect addressing format

Block I/O Instructions

- Similar to string instructions
- ins and outs do not take any operands
- I/O port address should be in DX
  - ∗ No direct addressing format is allowed
- ins instruction to read from an I/O port
  - ∗ ES:(E)DI should point to memory buffer
- outs instruction to write to an I/O port
  - ∗ DS:(E)SI should point to memory buffer
- rep prefix can be used for block transfer of data as in the string instructions

What Happens When An Interrupt Occurs?

- Push the EFLAGS register onto the stack
- Clear interrupt enable and trap flags

    - This disables further interrupts
    - Use **sti** to enable interrupts
- Push CS and EIP registers onto the stack
- Load CS with the 16-bit segment selector from the interrupt gate
- Load EIP with the 32-bit offset value from the interrupt gate

Interrupt Enable Flag Instructions

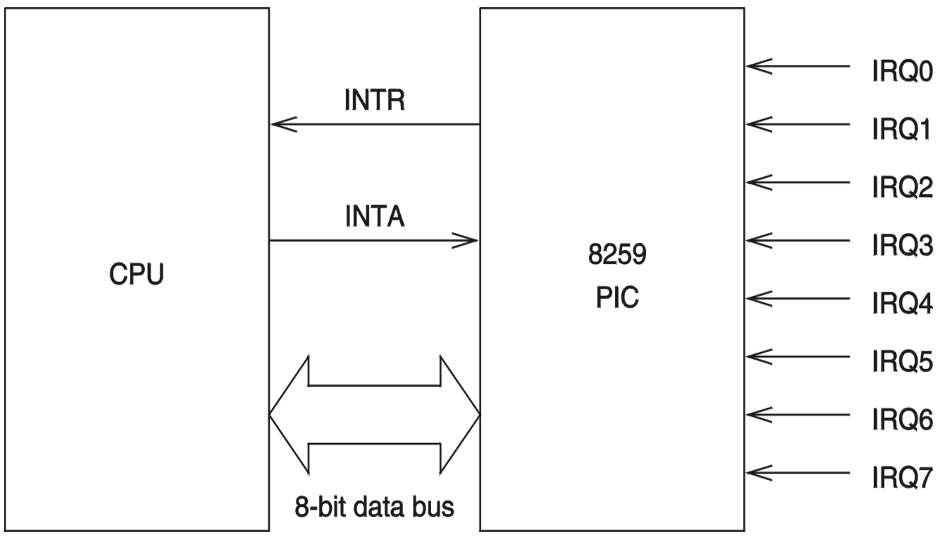
- Interrupt enable flag controls whether the processor should be interrupted or not
- Clearing this flag disables all further interrupts until it is set
    - Use **cli** (clear interrupt) instruction for this purpose
    - It is cleared as part interrupt processing
- Unless there is special reason to block further interrupts, enable interrupts in your ISR
    - Use **sti** (set interrupt) instruction for this purpose

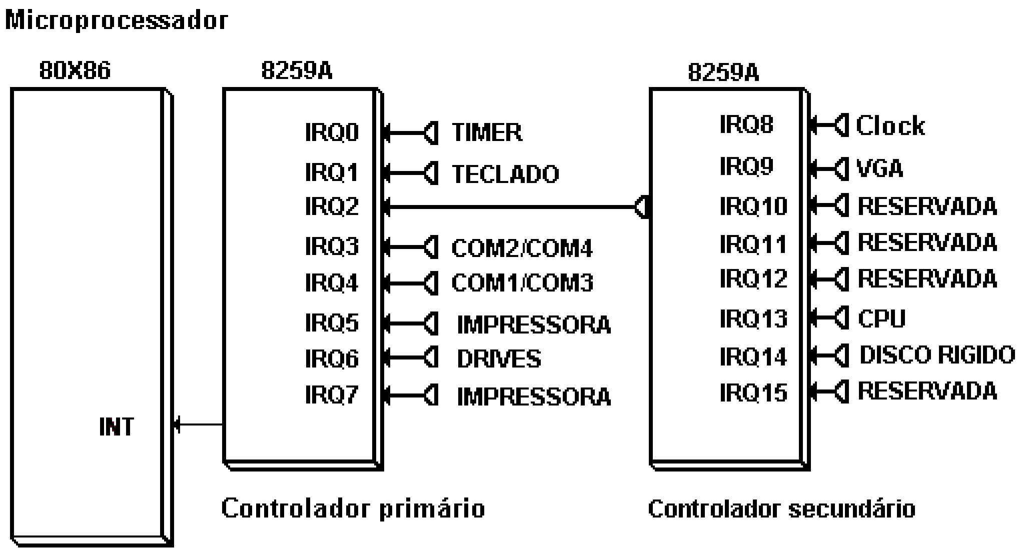
Returning From an ISR

- As in procedures, the last instruction in an ISR should be **iret**
- The actions taken on **iret** are:
    - pop the 32-bit value on top of the stack into EIP register
    - pop the 16-bit value on top of the stack into CS register
    - pop the 32-bit value on top of the stack into the EFLAGS register
- As in procedures, make sure that your ISR does not leave any data on the stack
    - Match your push and pop operations within the ISR

8259 Programmable Interrupt Controller

- 8259 can service up to eight hardware devices
    - Interrupts are received on IRQ0 through IRQ7
- 8259 can be programmed to assign priorities in several ways
    - Fixed priority scheme is used in the PC
        - IRQ0 has the highest priority and IRQ7 lowest
- 8259 has two registers
    - Interrupt Command Register (ICR)
        - Used to program 8259
    - Interrupt Mask Register (IMR)





- Mapping in a single 8259 PIC systems

IRQ#	Interrupt type	Device
0	08H	System timer
1	09H	Keyboard
2	0AH	reserved (2nd 8259)
3	0BH	Serial port (COM1)
4	0CH	Serial port (COM2)
5	0DH	Hard disk
6	0EH	Floppy disk
7	0FH	Printer (LPT1)

- Interrupt Mask Register (IMR) is an 8-bit register
  - ∗ Used to enable or disable individual interrupts on lines IRQ0 through IRQ7
    - » Bit 0 is associated with IRQ0, bit 1 to IRQ1, . . .

            » A bit value of 0 enables the corresponding interrupt (1 disables)
- Processor recognizes external interrupts only when the IF is set
- Port addresses:
  - ICR: 20H
  - IMR:21H

Example:

- Disable all 8259 interrupts except the system timer

```
mov   AL,0FEH
out   21H,AL
```

- 8259 needs to know when an ISR is done (so that it can forward other pending interrupt requests)
  - End-of-interrupt (EOI) is signaled to 8259 by writing 20H into ICR

```
mov   AL,20H
out   20H,AL
```

  - This code fragment should be used before **iret**

8255 Programmable Peripheral Interface Chip

- Provides three 8-bit registers (PA, PB, PC) that can be used to interface with I/O devices
- These three ports are configures as follows:

  PA -- Input port
  PB -- Output port
  PC -- Input port
- 8255 also has a command register
- 8255 port address mapping

PA	--- 60H
PB	--- 61H
PC	--- 62H
Command register	--- 63H

Keyboard Interface

- PA and PB7 are used for keyboard interface
  - PA0 -- PA6 = key scan code
  - PA7 = 0 if a key is depressed
  - PA7 = 1 if a key is released
- Keyboard provides the scan code on PA and waits for an acknowledgement
  - Scan code read acknowledge signal is provided by momentarily setting and clearing PB7
    - » Normal state of PB7 is 0
- Keyboard generates IRQ1
    - » IRQ1 generates a type 9 interrupt

I/O Data Transfer

- Three ways

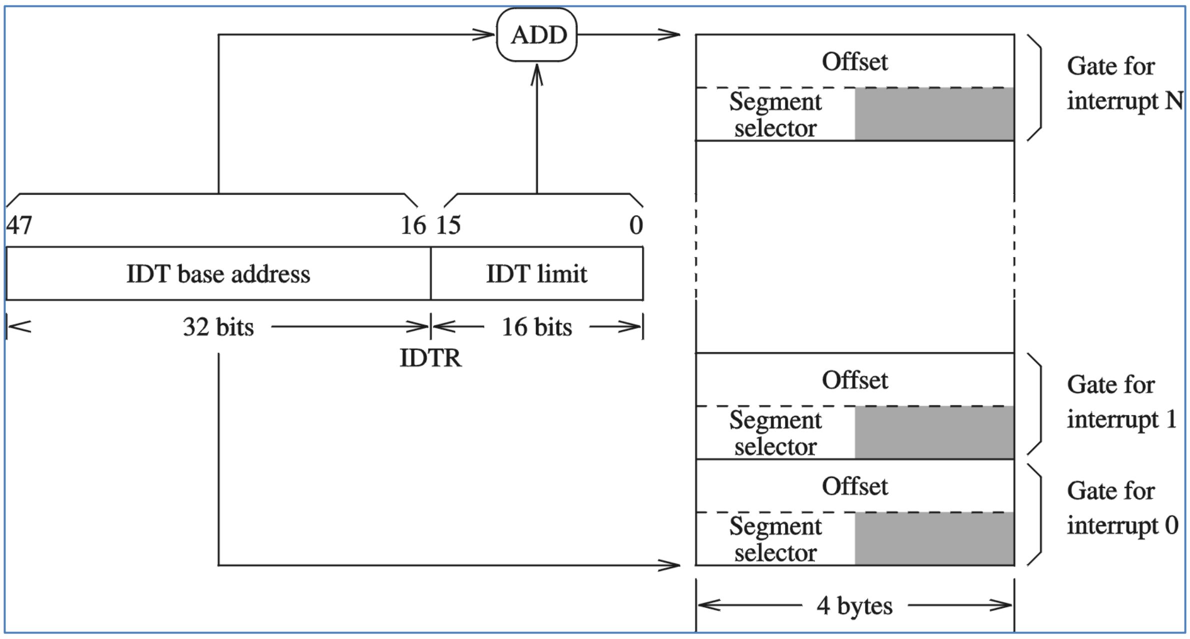
- ∗ Programmed I/O
    - » Repeatedly checks the status of an I/O device (through a status register of the associated I/O controller) until the desired condition is met
    - » This process is called *polling*
    - » Example: KBRD_PIO.ASM
- ∗ Interrupt-driven I/O
    - » Processor gets interrupted when a specified event occurs
    - » Example: KEYBOARD.ASM
- ∗ Direct memory access (DMA)
    - » Relieves the processor of low-level data transfer chore
    - » A DMA controller oversees this task

Polling Versus Interrupt-driven I/O

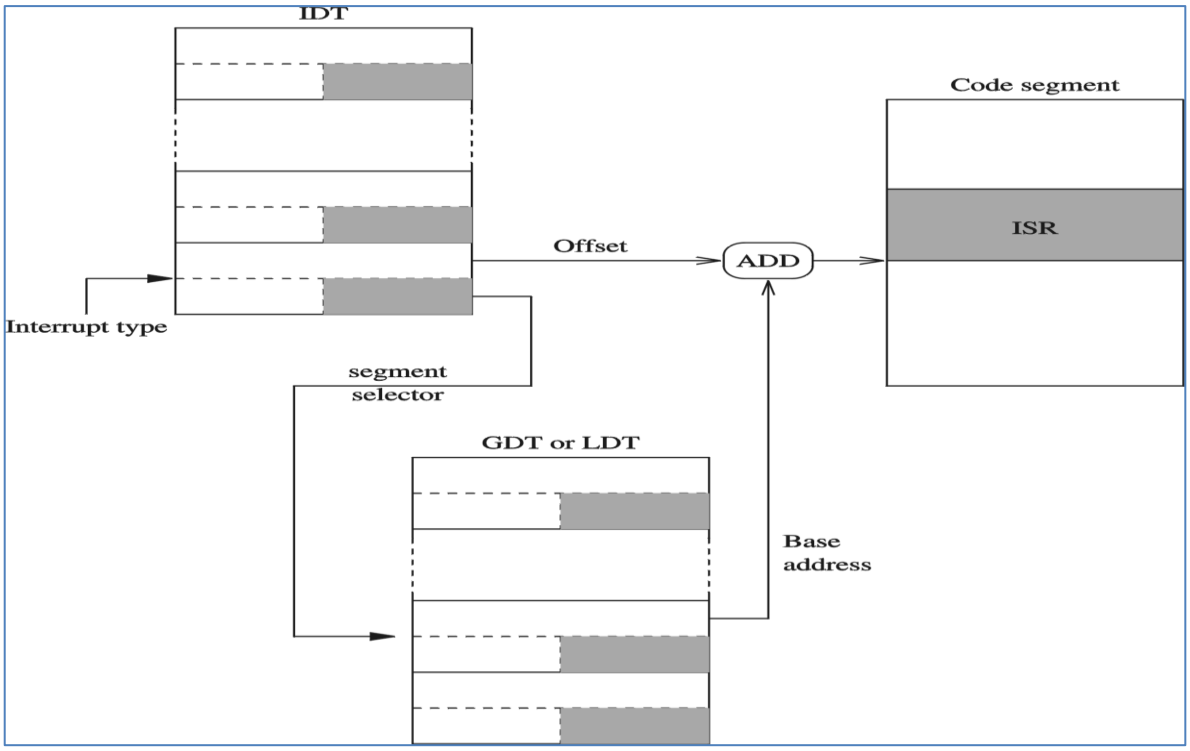
- Interrupt-driven I/O
    - ∗ Very efficient
    - ∗ Can be used to handle unanticipated events
- Programmed I/O
    - ∗ Polling involves overhead
        - » Repeated testing of condition
    - ∗ Can be used to handle only anticipated event

Interrupt Processing in the Protected Mode

- Up to 256 interrupts are supported (0 to 255)
    - » Same number in both real and protected modes
    - » Some significant differences between real and protected mode interrupt processing
- Interrupt number is used as an index into the *Interrupt Descriptor Table* (IDT)
    - ∗ This table stores the addresses of all ISRs
    - ∗ Each descriptor entry is 8 bytes long
        - » Interrupt number is multiplied by 8 to get byte offset into IDT
    - ∗ IDT can be stored anywhere in memory
        - » In contrast, real mode interrupt table has to start at address 0



- Location of IDT is maintained by IDT register IDTR
- IDTR is a 48-bit register
  - ∗ 32 bits for IDT base address
  - ∗ 16 bits for IDT limit value
    - » IDT requires only 2048 (11 bits)
    - » A system may have smaller number of descriptors
      - Set the IDT limit to indicate the size in bytes
  - ∗ If a descriptor outside the limit is referenced
    - » Processor enters shutdown mode
- Two special instructions to load (**lidt**) and store (**sidt**) IDT
  - ∗ Both take the address of a 6-byte memory as the operand



Interrupção de software e exceção no Linux

A arquitetura Intel identifica diferentes interrupções e exceções por um número variando de 0 a 255 (IDT), chamado de vetor. O Linux usa:

- os vetores de 0 a 31, os quais servem para exceções e interrupções não-mascaráveis,
- vetores de 32 a 47, que são para interrupções mascaráveis, isto é, para interrupções causadas por requisições de interrupção (IRQs),
- e apenas um vetor (128), dos restantes vetores, de 48 a 255, que é usado para interrupções de software.

O Linux usa esse vetor, 128, para implementar uma chamada de sistema, isto é, quando um *opcode* int 0x80 é executado por um processo rodando em user mode, a CPU alterna para kernel mode e inicia a execução da função system_call().

```
set_trap_gate(0,&divide_error);
set_trap_gate(1,&debug);
set_intr_gate(2,&nmi);
set_system_gate(3,&int3);
set_system_gate(4,&overflow);
set_system_gate(5,&bounds);
set_trap_gate(6,&invalid_op);
set_trap_gate(7,&device_not_available);
set_trap_gate(8,&double_fault);
set_trap_gate(9,&coprocessor_segment_overrun);
set_trap_gate(10,&invalid_TSS);
set_trap_gate(11,&segment_not_present);
set_trap_gate(12,&stack_segment);
set_trap_gate(13,&general_protection);
set_intr_gate(14,&page_fault);
set_trap_gate(16,&coprocessor_error);
set_trap_gate(17,&alignment_check);
set_trap_gate(18,&machine_check);
set_trap_gate(19,&simd_coprocessor_error);
set_system_gate(128,&system_call);
```

Exceptions

- Three types of exceptions
  - Depending on the way they are reported
  - Whether or not the interrupted instruction is restarted
    - Faults
    - Traps
    - Aborts
- Faults and traps are reported at instruction boundaries
- Aborts report severe errors
  - Hardware errors
  - Inconsistent values in system tables

Faults

- Instruction boundary before the instruction during which the exception was detected
- Restarts the instruction
- Divide error (detected during **div/idiv** instruction)
- Segment-not-found fault

- Instruction would be illegal to execute
- Examples:
    - Writing to a memory segment marked 'read-only'
    - Reading from an unavailable memory segment (on disk)
    - Executing a 'privileged' instruction
- Detected before incrementing the IP
- The causes of 'faults' can often be 'fixed'
- If a 'problem' can be remedied, then the CPU can just resume its execution-cycle

Traps

- Instruction boundary immediately after the instruction during which the exception was detected
- No instruction restart
- Overflow exception (interrupt 4) is a trap
- User defined interrupts are also examples of traps

- A CPU might have been programmed to automatically switch control to a 'debugger' program after it has executed an instruction
- That type of situation is known as a 'trap'
- It is activated after incrementing the IP

Aborts

- Não permitem o reinício da execução
- Erros de hardware
- Inconsistência nas tabelas de sistema

Dedicated Interrupts

- Several Pentium predefined interrupts --- called dedicated interrupts
- These include the first five interrupts:

interrupt type	Purpose
0	Divide error
1	Single-step
2	Nonmaskable interrupt (MNI)
3	Breakpoint
4	Overflow

- Divide Error Interrupt
    - ∗ CPU generates a type 0 interrupt whenever the div/idiv instructions result in

a quotient that is larger than the destination specified

- Single-Step Interrupt
  - ∗ Useful in debugging
  - ∗ To single step, Trap Flag (TF) should be set
  - ∗ CPU automatically generates a type 1 interrupt after executing each instruction if TF is set
  - ∗ Type 1 ISR can be used to present the system state to the user
- Breakpoint Interrupt
  - ∗ Useful in debugging
  - ∗ CPU generates a **type 3** interrupt
  - ∗ Generated by executing a special single-byte version of **int 3** instruction (opcode CCH)
- Overflow Interrupt
  - ∗ Two ways of generating this type 4 interrupt
    - **int 4** (unconditionally generates a type 4 interrupt)
    - **into** (interrupt is generated only if the overflow flag is set)
  - ∗ We do not normally use **into** as we can use **jo/jno** conditional jumps to take care of overflow

Software Interrupts

- Initiated by executing an interrupt instruction

  int interrupt-type

  interrupt-type is an integer in the range 0 to 255
- Each interrupt type can be parameterized to provide several services.
- For example, Linux interrupt service **int 0x80** provides a large number of services (more than 180 system calls!)
  - ∗ EAX register is used to identify the required service under int 0x80

System calls (chamadas de sistema, syscalls)

Chamadas de sistema fornecem uma camada entre o hardware e os processos em user mode. Essa camada serve para três propósitos:

- fornece uma interface de abstração de hardware
  quando uma aplicação lê ou escreve um arquivo, por exemplo, não há necessidade de se preocupar com o tipo de disco, mídia ou até mesmo sistema de arquivos no qual o arquivo reside.

- asseguram a segurança e estabilidade do sistema
  com o kernel atuando como meio-de-campo entre os recursos do sistema, e o espaço de usuário, o kernel pode arbitrar o acesso baseado nas permissões e outros critérios. Por exemplo, isso previne aplicações de incorretamente usar o hardware, roubar recurso de outros processos, ou fazer qualquer alteração ao sistema.

- uma camada única entre o espaço de usuário e o resto do sistema.
  se as aplicações tivessem acesso livre aos recursos do sistema sem o conhecimento do kernel, seria praticamente impossível implementar

características como multitasking ou memória virtual.

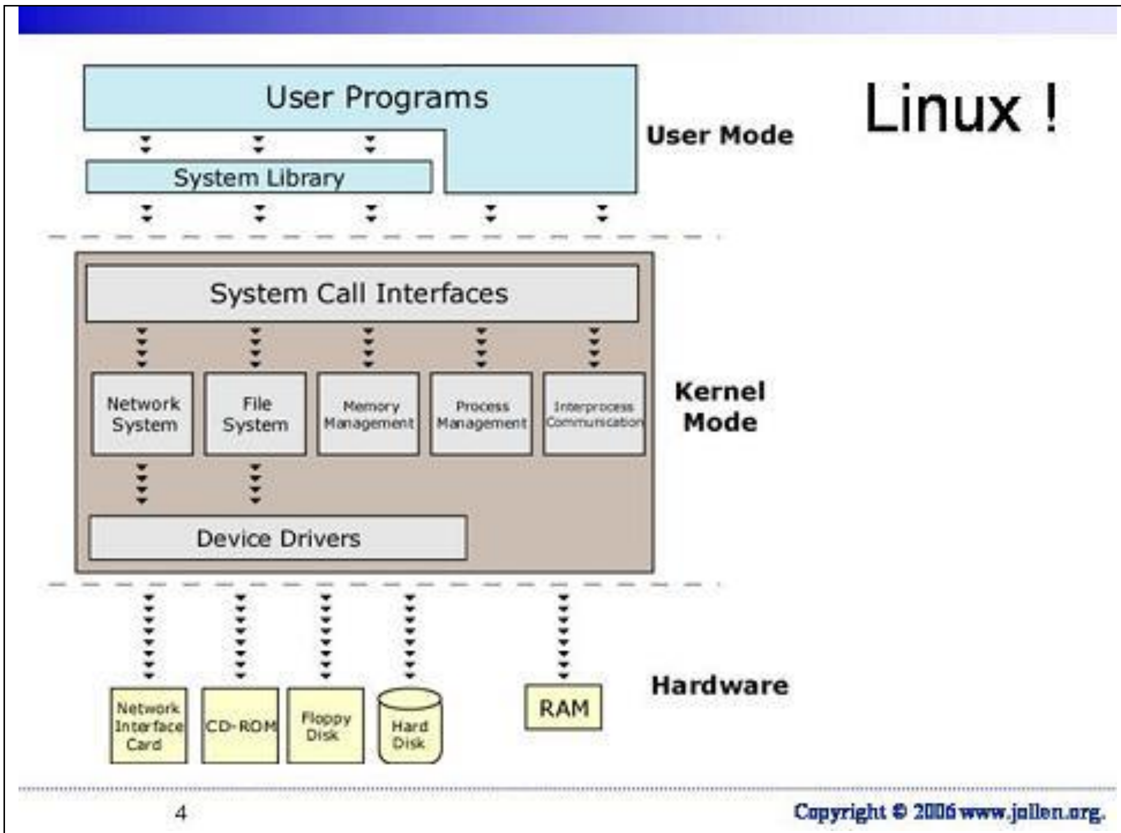


Figura 1: Relacionamento entre as aplicações, o kernel e o hardware

APIs, POSIX e a biblioteca C

Geralmente, aplicações são programadas utilizando uma Interface de Programação de Aplicações (Application Programming Interface, API), e não diretamente *syscalls*. Isso é importante, pois não há uma correlação direta necessária entre a interface que as aplicações fazem uso, e a interface atual fornecida pelo kernel. Uma API define um conjunto de interfaces de programação utilizada por aplicações. Essas interfaces podem ser implementadas como uma chamada de sistema, implementada utilizando múltiplas chamadas de sistemas, ou até mesmo nem usar uma chamada de sistema. De fato, a mesma API pode existir em múltiplo sistemas e fornecer a mesma interface para aplicações, enquanto a implementação da mesma API pode diferir significativamente de um sistema para outro.

Uma das mais populares interfaces de programação de aplicações no mundo Unix é baseado no padrão POSIX. A interface de chamadas de sistema no Linux, assim como na maioria dos sistemas Unix, é fornecida em parte pela biblioteca C. A biblioteca C implementa a API principal nos sistemas Unix, incluindo a biblioteca padrão C e a interface de chamada de sistemas. A biblioteca C também fornece grande parte da POSIX API. Abaixo o relacionamento entre API, a biblioteca C, e o kernel:

```
printf() --> printf() na biblioteca C --> write() na biblioteca C --> write() syscall
Aplicação --------------------------> Biblioteca C ----------------------------> Kernel
```

Usando syscalls

Syscalls são acessadas via uma chamada de função. Elas definem um ou mais argumentos (entradas) e podem resultar em um ou mais resultados (saídas). Syscalls também fornecem um valor de retorno que significa sucesso ou erro. Geralmente, um valor negativo denota um erro, enquanto um valor zero denota sucesso. Por exemplo, a syscall getpid:

```
asmlinkage long sys_getpid(void)
{
     return current->tgid;
}
```

Categories of system calls

System calls can be roughly grouped into five major categories:

1. Process Control
   - load
   - execute
   - create process (for example, fork on Unix-like systems or NtCreateProcess in the Windows NT Native API)
   - terminate process
   - get/set process attributes
   - wait for time, wait event, signal event
   - allocate, free memory
2. File management
   - create file, delete file
   - open, close
   - read, write, reposition
   - get/set file attributes
3. Device Management
   - request device, release device
   - read, write, reposition
   - get/set device attributes
   - logically attach or detach devices
4. Information Maintenance
   - get/set time or date
   - get/set system data
   - get/set process, file, or device attributes
5. Communication
   - create, delete communication connection
   - send, receive messages
   - transfer status information
   - attach or detach remote devices

No Linux, cada syscall tem associada a si um número syscall. Isso é um número único que é usado para referenciar uma syscall em específica. Quando um processo em user mode executa uma syscall, o número da syscall delineia qual syscall foi executada, o processo não se refere à syscall pelo nome.

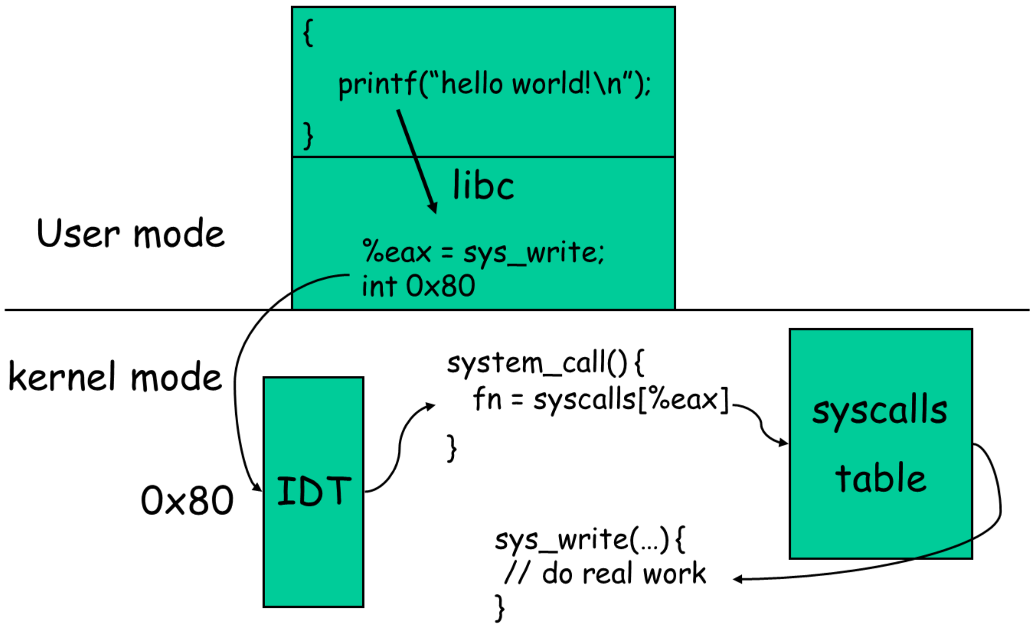
Para utilizar a syscall dentro de uma aplicação, não será possível chamá-la diretamente, pois as chamadas de sistema executam em kernel-mode, e as aplicações em user-mode. Por isso, deve ser utilizado a instrução *int 80h*, que altera a execução para kernel mode.

Os parâmetros para a syscall devem ser passados, conforme abaixo:

- eax: número da system call
- ebx, ecx, edx, esi, edi: no caso da system call requerer mais de um parâmetro, eles devem ser passados nessa ordem antes da chamada do int 80h.

Exemplo:

```
# void _exit(int status)
mov ebx, 0        # Código do exit
mov eax, 1       # Número da chamada de sistema (exit)
int 80h          # Chamada do kernel
```



Rotina de Tratamento

- Instalação
```
int request_irq(unsigned int irq,
                irqreturn_t (*handler)
                     (int irq,
                      void *dev_id,
                      struct pt_regs *regs),
                unsigned long flags,
                const char *device,
                void *dev_id)
```

- Remoção
```
void free_irq(unsigned int irq, void *dev_id)
```

```
static volatile int count;

irqreturn_t lpirq (int irq, void *dev_id, struct pt_regs *regs)
{
      count++;
      return IRQ_HANDLED;
}
```

File I/O

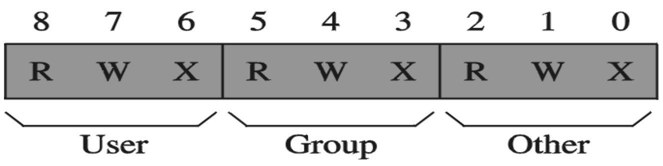
- Focus is on File I/O
    - Keyboard and display are treated as stream files
    - Three standard file streams are defined
        - Standard input (**stdin**)
            - Associated device: Keyboard
        - Standard output (**stdout**)
            - Associated device: Display
        - Standard error (**stderr**)
            - Associated device: Display
- File descriptor
    - Small integer acts as a file id
    - Use file descriptors to access open files
    - File descriptor is returned by the **open** and **create** systems calls
    - Don’t have to open the three standard files
        - Lowest three integers are assigned to these files
            - **stdin** (0)
            - **stdout** (1)
            - **stderr** (2)
- File pointer
    - Associated with each open file
    - Specifies offset (in bytes) relative to the beginning of the file
        - Read and write operations use this location
    - When a file is opened, file pointer points to the firs byte
        - Subsequent reads move it to facilitate sequential access
    - Direct access to a file
        - Can be provided by manipulating the file pointer

File System Calls

- File create call

System call 8 --- Create and open a file

```
Inputs:     EAX = 8
            EBX = file name
            ECX = file permissions
Returns:    EAX = file descriptor
Error:  EAX = error code
```



- File open call

System call 5 --- Open a file

```
Inputs:     EAX = 5
            EBX = file name
            ECX = file access mode
            EDX = file permissions
Returns:    EAX = file descriptor
Error:  EAX = error code
```

- File read call

System call 3 --- Read from a file

```
Inputs:     EAX = 3
            EBX = file descriptor
            ECX = pointer to input buffer
            EDX = buffer size
                    (max. # of bytes to read)
Returns:    EAX = # of bytes read
Error:  EAX = error code
```

- File write call

System call 4 --- Write to a file

```
Inputs:     EAX = 4
            EBX = file descriptor
            ECX = pointer to output buffer
            EDX = buffer size
                    (# of bytes to write)
Returns:    EAX = # of bytes written
Error:  EAX = error code
```

- File close call

System call 6 --- Close a file

```
Inputs:     EAX = 6
```

EBX = file descriptor
Returns: EAX = ---
Error: EAX = error code

- File seek call

System call 19 --- lseek (updates file pointer)
Inputs: EAX = 19
EBX = file descriptor
ECX = offset
EDX = whence
Returns: EAX = byte offset from the beginning of file
Error: EAX = error code

- **whence** value

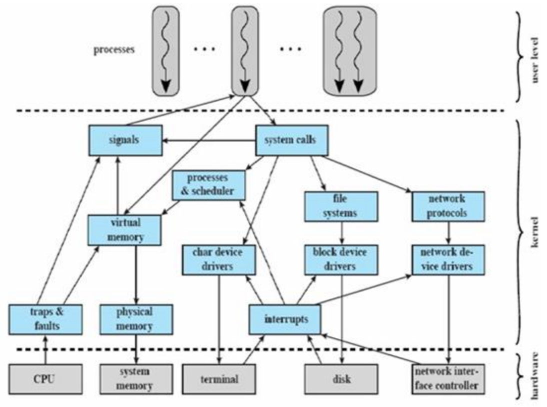
Reference position	**whence value**
Beginning of file	0
Current position	1
End of file	2

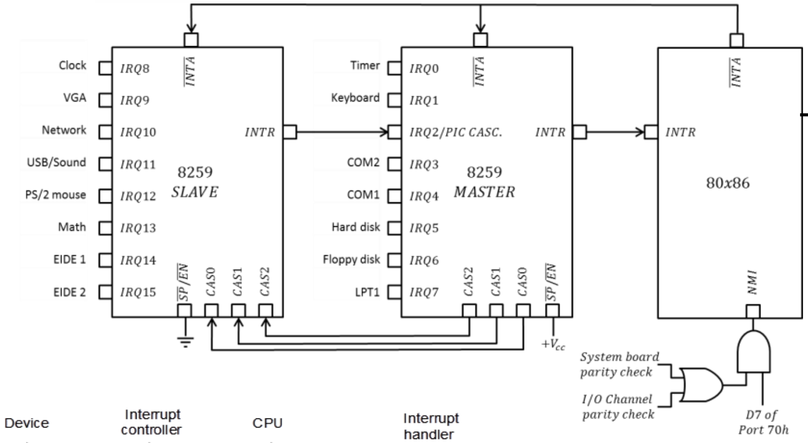
INT 80h		**Create and open a file**	**Open a file**	**Read from a file**	**Write to a file**	**Close a file**
INPUT	**EAX**	8	5	3	4	6
	EBX	filename	filename	file descriptor	file descriptor	file descriptor
	ECX	file permissions	file access mode	pointer to input buffer	pointer to output buffer	
	EDX		file permissions	buffer size (maximum number of bytes to read)	buffer size (number of bytes to write)	
RETURNS	**EAX**	file descriptor	file descriptor	number of bytes read	number of bytes written	
ERROR	**EAX**	error code	error code	error code	error code	error code

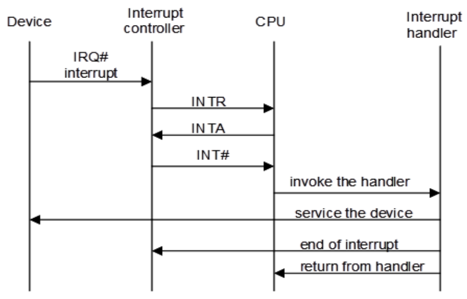
Referência

Sivarama P. Dandamudi, "Guide to Assembly Language Programming in Linux". Publisher: Springer. ISBN: 0387258973. 2005. 539 pages

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define READ                  3
#define WRITE                 4
#define OPEN                  5
#define CLOSE                 6
#define CREATE                8

#define  O_RDONLY 00
#define  O_WRONLY 01
#define  O_RDWR   02

int strlen_ (char *s) {
  __asm__ (
    ".intel_syntax noprefix  \n \t"
    "cld                     \n \t"
    "xor      eax, eax       \n \t"
    "mov      edi, [ebp+8]   \n \t"
    "mov      ecx, -1        \n \t"
    "repne    scasb          \n \t"
    "not      ecx            \n \t"
    "dec      ecx            \n \t"
    "mov      eax, ecx       \n \t"
    ".att_syntax prefix      \n"
    :
    :
  );
};


int printstr (char *msg, int n) {

  __asm__ (
    ".intel_syntax noprefix  \n \t"
    "mov    edx,[ebp+12]     \n \t"      //message length
    "mov    ecx,[ebp+8]      \n \t"      //message to write
    "mov    ebx,1            \n \t"      //file descriptor (stdout)
    "mov    eax,4            \n \t"      //system call number (sys_write)
    "int    0x80             \n \t"      //call kernel
    ".att_syntax prefix      \n"
    :
    :
  );
}


int readstr(char *msg, int n){
  __asm__ (
    ".intel_syntax noprefix        \n \t"
    "mov      eax, 3               \n \t"    //  System Call -- read
    "mov      ebx, 2               \n \t"    //  file descriptor (stdin)
    "mov      ecx, [ebp+8]         \n \t"    //  buffer
    "mov      edx, [ebp+12]        \n \t"    //  size
    "int      0x80                 \n \t"
    "mov      ebx, [ebp+8]         \n \t"
    "mov      BYTE PTR [eax+ebx], 0\n \t"       //  acrescenta o NUL
    ".att_syntax prefix            \n"
    :
    :
  );
};

int create(const char *pathname, int permission) {
  __asm__ (
    ".intel_syntax noprefix  \n \t"
    "mov      eax, 8         \n \t"    // System Call -- open
    "mov      ebx, [ebp+8]   \n \t"    // Pointer to filename
    "mov      ecx, [ebp+12]  \n \t"    // Flag for Permission
    "int      0x80           \n \t"
    ".att_syntax prefix      \n"
    :
    :
  );
};

int open(const char *fn, int permission, int mode){
  __asm__ (
    ".intel_syntax noprefix  \n \t"
    "mov      eax, 5         \n \t"    //  System Call -- open
    "mov      ebx, [ebp+8]   \n \t"    //  Pointer to filename
    "mov      ecx, [ebp+16]  \n \t"    //  Flag for Permission
    "mov      edx, [ebp+12]  \n \t"    //  Mode
    "int      0x80           \n \t"
    ".att_syntax prefix      \n"
    :
    :
  );
};


int read(int handle, void *buffer, int buffersize){
  __asm__ (
    ".intel_syntax noprefix  \n \t"
    "mov      eax, 3         \n \t"    //  System Call -- read
    "mov      ebx, [ebp+8]   \n \t"    //  handle
    "mov      ecx, [ebp+12]  \n \t"    //  buffer
    "mov      edx, [ebp+16]  \n \t"    //  size
    "int      0x80           \n \t"
    ".att_syntax prefix      \n"
    :
    :
  );
};


int write(int handle, void *buffer, int buffersize){
  __asm__ (
    ".intel_syntax noprefix  \n \t"
    "mov      eax, 4          \n \t"    //  System Call -- write
    "mov      ebx, [ebp+8]    \n \t"    //  handle
    "mov      ecx, [ebp+12]   \n \t"    //  buffer
    "mov      edx, [ebp+16]   \n \t"    //  size
    "int      0x80            \n \t"
    ".att_syntax prefix       \n"
    :
    :
  );
};


int close(int fd) {
  __asm__ (
    ".intel_syntax noprefix   \n \t"
    "mov      eax, 6          \n \t"    // System Call -- close
    "mov      ebx, [ebp+8]    \n \t"    // file handle
    "int      0x80            \n \t"
    ".att_syntax prefix       \n"
    :
    :
  );
};



int copyfile_ (char *fn1, char *fn2) {
//  char *buffer;
//  int h1, h2, n1;
  __asm__ (
    ".intel_syntax noprefix              \n \t"
    "sub      esp, 32                    \n \t"
    "mov      DWORD PTR [esp], 100       \n \t"
    "call     malloc                     \n \t"
    "mov      DWORD PTR [ebp-4], eax     \n \t"
    "mov      eax, [ebp+8]               \n \t"
    "mov      [esp], eax                 \n \t"
    "mov      dword ptr [esp+4], 0x1c0   \n \t"
    "mov      dword ptr [esp+8], 0       \n \t"
    "call     open                       \n \t"
    "mov      [ebp-8], eax               \n \t"

    "mov      eax, [ebp+12]              \n \t"
    "mov      [esp], eax                 \n \t"
    "mov      dword ptr [esp+4], 0x1c0   \n \t"
    "call     create                     \n \t"
    "mov      [ebp-12], eax              \n"
 "ini:                                   \n \t"
    "mov      eax, [ebp-8]               \n \t"
    "mov      [esp], eax                 \n \t"
    "mov      eax, [ebp-4]               \n \t"
    "mov      [esp+4], eax               \n \t"
    "mov      dword ptr [esp+8], 100     \n \t"
    "call     read                       \n \t"
    "mov      [ebp-16], eax               \n \t"
    "cmp      eax, 0                      \n \t"
    "jle      fim                         \n \t"

    "mov      eax, [ebp-12]               \n \t"
    "mov      [esp], eax                  \n \t"
    "mov      eax, [ebp-4]                \n \t"
    "mov      [esp+4], eax                \n \t"
    "mov      eax, [ebp-16]               \n \t"
    "mov      [esp+8], eax                \n \t"
    "call     write                       \n \t"

    "jmp      ini                         \n"
 "fim:                                    \n \t"
    "mov    eax, DWORD PTR [ebp-4]        \n \t"
    "mov    DWORD PTR [esp], eax          \n \t"
    "call   free                          \n \t"
    "mov      esp, ebp                    \n \t"
    ".att_syntax prefix                   \n"
    :
    :
  );

}


int copyfile (char *fn1, char *fn2) {
  char buffer[100];
  int h1, h2, n1;
  h1 = open(fn1, 0x1C0, O_RDONLY); // 700 = 111000000 = 1 1100 0000 = 0x1C0
  h2 = create(fn2, 0x1C0);

  while((n1 = read (h1, buffer, 100))>0)
        write (h2, buffer, n1);

  return 0;
}

int copyfile2 (char *fn1, char *fn2) {
  char *buffer;
  int h1, h2, n1;
  buffer = malloc (100);
  h1 = open(fn1, 0x1C0, O_RDONLY); // 700 = 111000000 = 1 1100 0000 = 0x1C0
  h2 = create(fn2, 0x1C0);

  while((n1 = read (h1, buffer, 100))>0)
        write (h2, buffer, n1);
  free (buffer);
  return 0;
}

// gcc -o syscalls syscalls.c
// gcc -masm=intel -S -o syscalls.S syscalls.c

int main () {
  char msg[] = "press enter\n";
  copyfile_ ("teste.txt", "teste2.bak");
  printstr (msg, strlen_(msg));
  readstr (msg, 10);
  printstr (msg, strlen_(msg));
  return 0;
}
```







IDT - INTERRUPT DESCRIPTOR TABLE

INT		USE
0	19	Nonmaskable interrupts and exceptions
0	Fault	divide_error
1	Fault/Trap	debug
2	Interrupt	nmi
3	Trap	int3
4	Trap	overflow
5	Fault	bounds
6	Fault	invalid_op
7	Fault	device_not_available
8	Abort	double_fault
9	Fault	coprocessor_segment_overrun
10	Fault	invalid_TSS
11	Fault	segment_not_present
12	Fault	stack_segment
13	Fault	general_protection
14	Fault	page_fault
16	Fault	coprocessor_error
17	Fault	alignment_check
18	Abort	machine_check
19	Fault	simd_coprocessor_error
20	31	Intel-reserved
32	127	External interrupts (IRQs)
32	IRQ0	Timer
33	IRQ1	Keyboard
34	IRQ2	PIC cascading
35	IRQ3	Second serial port
36	IRQ4	First serial port
37	IRQ5	Hard disk
38	IRQ6	Floppy disk
39	IRQ7	Printer (LPT1)
40	IRQ8	System clock
41	IRQ9	VGA
42	IRQ10	Network interface
43	IRQ11	USB port, sound card
44	IRQ12	PS/2 mouse
45	IRQ13	Mathematical coprocessor
46	IRQ14	EIDE disk controller's first chain
47	IRQ15	EIDE disk controller's second chain
128		system_call
129	238	External interrupts (IRQs)
239		Local APIC timer interrupt
240		Local APIC thermal interrupt
241	250	Reserved by Linux for future use
251	253	Interprocessor interrupts
254		Local APIC error interrupt
255		Local APIC spurious interrupt

```
PUSH EBP
MOV ...
IN EAX, DX
OUT DX, EAX
POP ...
EOI
IRET
```

```
system_call () {
  fn = syscalls[eax]
}

sys_read (...) {
  // do real work
}
sys_write (...) {
  // do real work
}
```

```
{
  printf ("hello world\n");
}
              (libc)
eax = sys_write;
int 0x80
```

User mode | kernel mode