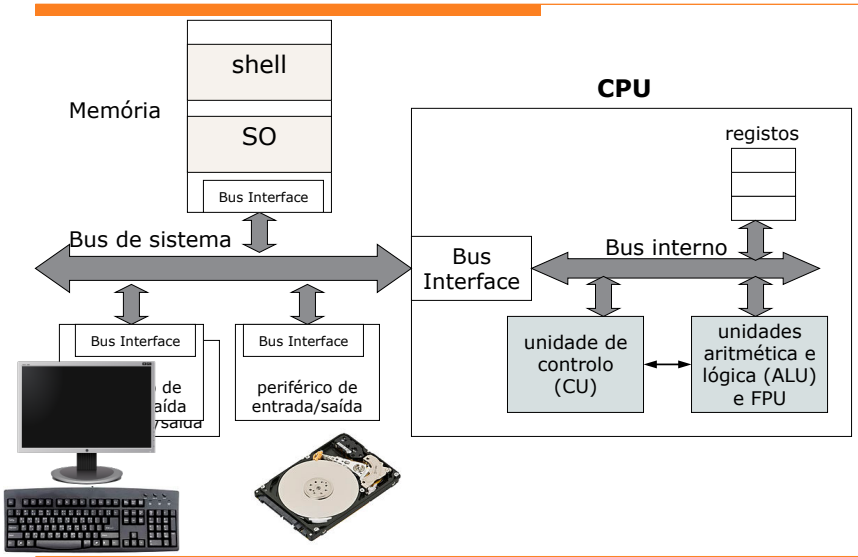
Arquitetura de Computadores

MiEI – 2016/17

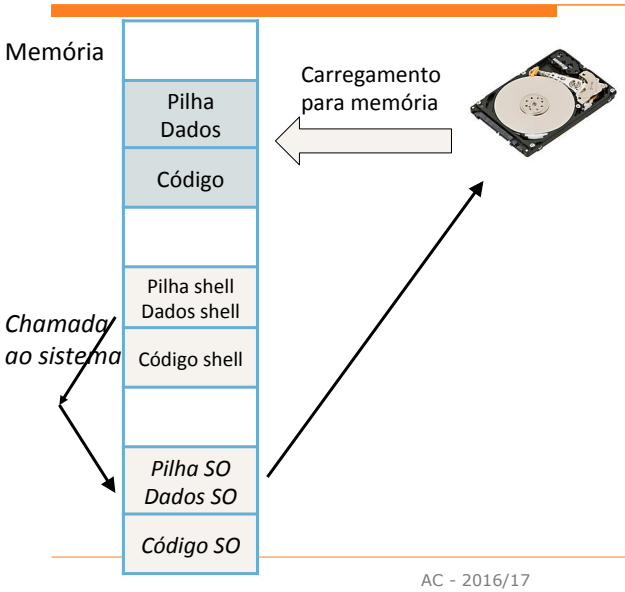
DI-FCT/UNL

Aula 11

Execução de um programa



Execução de um Programa



O que é o SO?

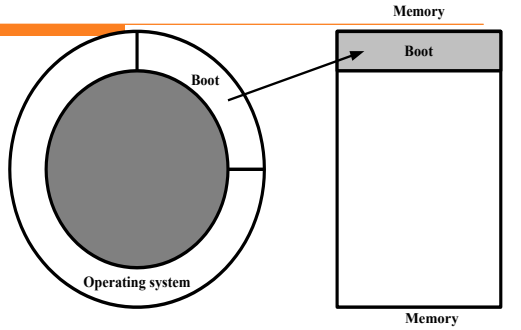
- O gestor de recursos
  - Gere e permite a partilha do *hardware* pelos programas
  - Gere os programas e suas interações
- Uma máquina virtual
  - Oferece abstrações de mais fácil utilização
  - Oferece operações de mais "alto nível" usadas pelo resto do sistema computacional
  - Define uma ABI
    - Exemplo: Linux, intel 32bits
  - Define uma API
    - Exemplo: POSIX, X/OPEN

Na pratica o que é?

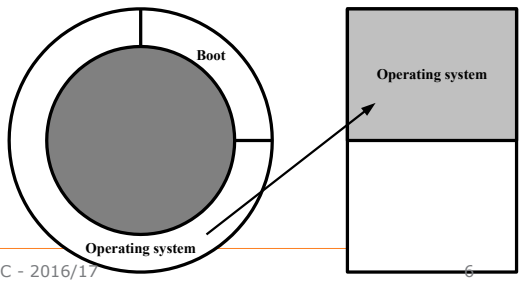
- Um programa permanentemente carregado na memória central (RAM) – também chamado "kernel"
- A sua ação é complementada por
  - Programas de sistema: interpretador de comandos, carregadores, ligadores, serviços de impressão e rede...
- O SO é sempre carregado na RAM quando:
  - O computador é ligado (power up)
  - Quando se reinicia o hardware (reset)

Arranque do SO

O "*bootstrap loader* em ROM" lê o programa de "boot" para memória



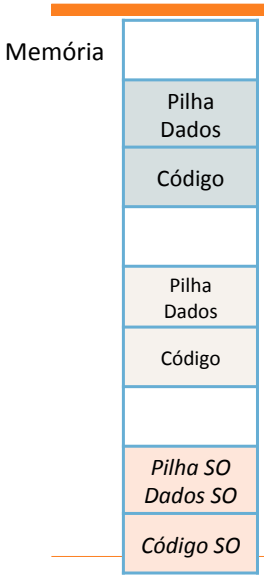
O programa de "boot" lê o resto do sistema para memória



Inicialização

- Inicialização do hardware
  - controladores de entrada /saída
- Inicialização das estruturas de dados que representam os vários recursos de sistema e que suportam os algoritmos de gestão desses recursos.
- Criação dos primeiros processos:
  - processos auxiliares de sistema (rede, sistema de ficheiros, …)
  - Login, interpretador de comandos

Sistema multi-programado ou *multitasking*



- Em memória estão vários programas em execução
  - Podem existir múltiplas instâncias do mesmo programa
- Cada processo tem a ilusão de um sistema dedicado
  - Memória, periféricos e CPU

Um programa num Sist. Operação

- O SO gere a máquina, incluindo pôr os programas em execução
- Num sistema multiprogramado, os programas não podem fazer tudo o que querem:
  - Não podem aceder à memória dos outros programas
  - Não podem alterar diretamente os periféricos (têm de os partilhar com os outros programas)
  - Têm de recorrer a pedidos ao SO para esses tipos de operações, usando as API do SO
- Algumas instruções do CPU estão assim vedadas aos programas "utilizadores"
  - CPU tem pelo menos dois modos de funcionamento

Dois modos de operação

- O hardware tem de suportar pelo menos dois modos de operação do CPU.
  1. *Modo utilizador* – CPU está a executar por conta de um programa normal.
  2. *Modo supervisor* (*kernel mode* or *system mode*) – execução de código do SO

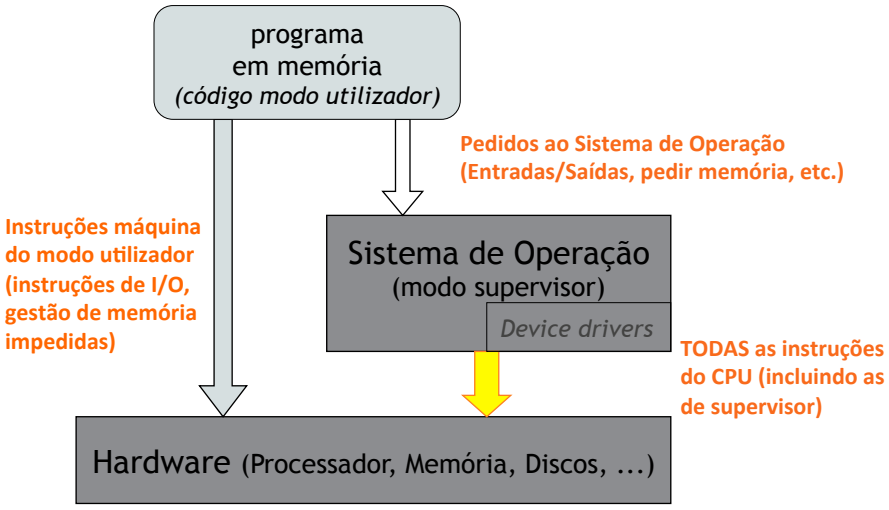
Ciclo de execução:

```
Fetch
Decode
if ( inst previlegiada && modo CPU != supervisor)
                    interrupção por exepção
else Execute
```

Mecanismos Indispensáveis ao SO

- O hardware deve suportar:
  - Proteção do espaço de I/O
  - Proteção de memória
  - Proteção do CPU
  - Modo utilizador/supervisor e interrupções
- Para garantir que os processos não acedem diretamente a esses recursos e que só o SO pode controlar esses recursos

Um programa em execução



Exemplo de troca de modo do CPU

- **`Int`** (interrupt) é uma das instruções que muda o modo de funcionamento (utilizador → supervisor)
  - Mas o SO é o primeiro programa a executar
  - Define para cada "int nº" a rotina a executar (no seu código)
  - Cada código destes termina sempre por voltar a pôr o CPU em modo utilizador
  - O SO inicia a execução dos programas sempre em modo utilizador
  - Conclusão: um programa em modo utilizador nunca consegue executar o seu código em modo supervisor

Protecção do CPU

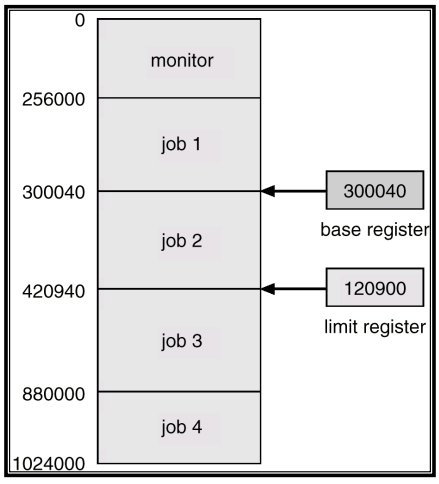
- Como partilhar o CPU entre os processos?
- *Temporizador* – interrompe o processamento ao fim de um tempo pré-determinado.
  - Decrementado a cada "tick" do relógio.
  - Quando chega a 0 lança uma interrupção de hardware.
  - Assegura que o SO nunca perde o controlo do CPU
- *Load-timer* é uma instrução privilegiada.

Protecção de Memória

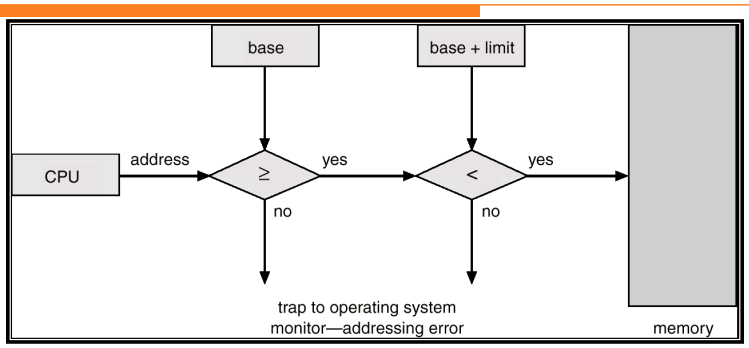
- É necessário proteger a memória entre programas e do Sistema Operativo
- Uma solução simples é ter dois registos no CPU que determinam a faixa legal de endereços:
  - **Registo Base**
  - **Registo Limite**
  - Os endereços efetivos do programa são **virtuais**, ajustados pelo registo base para obter o endereço **real**
- A memória fora dessa zona é proibida ao programa.
  - O CPU não gera endereços for a do intervalo

Registos base e limite

- O SO define para cada programa o valor desses registos



Protecção de memória



- As instruções para alterar os registos *base* e *limite* são privilegiadas.
  - Só o SO as pode usar

Protecção de I/O

- Todas as instruções de I/O são privilegiadas:

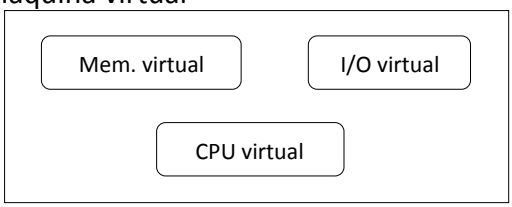
  in src, dst — *src – imediato/reg*

  out src, dst — *dst – endereço (IO)*

- Só o SO pode aceder aos dispositivos de entrada / saída.

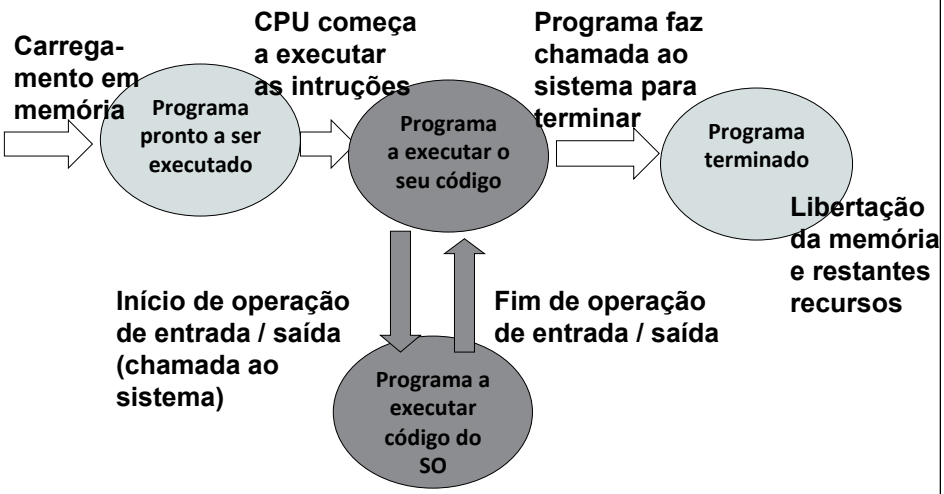
Máquina virtual de um processo

- Processo = programa em execução
  - O SO mantém informação sobre cada processo para garantir a sua máquina virtual



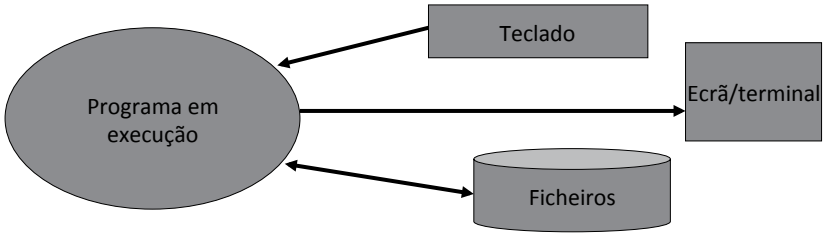
- O SO gere e mapeia na máquina real
  - Mem. Virtual → mapa de memória
  - CPU virtual → time-sharing
  - I/O virtual → exemplo: canais de I/O (streams) ficheiros

Programa em execução: Processo



Canais de entrada/saída

Um programa em execução vê periféricos e ficheiros através de canais de entrada / saída (ou streams)
Esses canais permitem ler e escrever sequências de bytes de forma fiável e mantendo a ordem

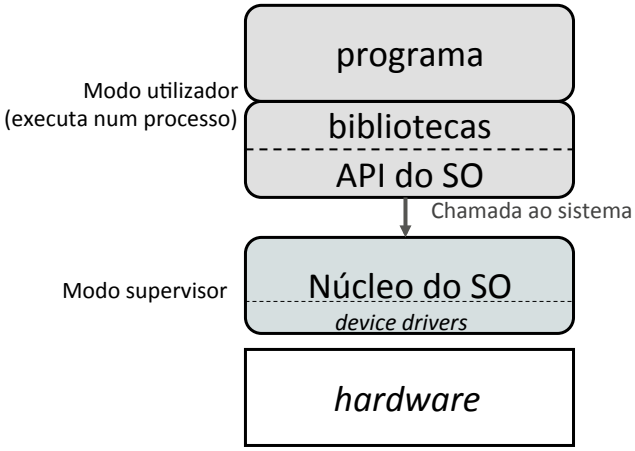


Os canais no SO são identificados por números: começam em 0 e vão crescendo à medida das necessidades

- Os canais 0 (stdin), 1 (stdout) e 2 (stderr) são abertos pelo SO quando o programa começa a executar
- o programa pode abrir mais canais usando a chamada open

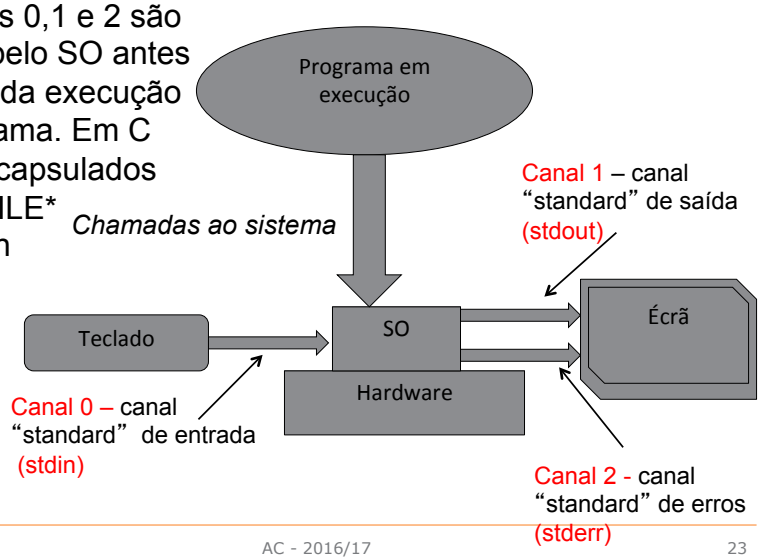
Chamadas entre níveis de software

- Visão de uma aplicação



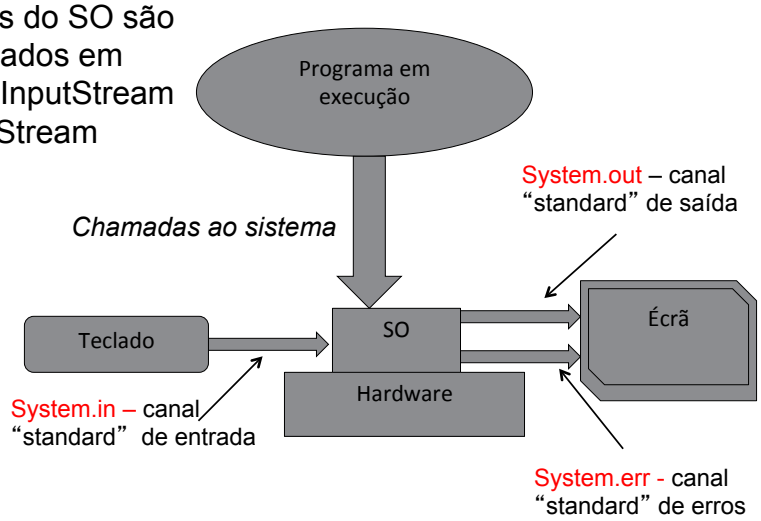
Processo e IO - Unix e C

Os canais 0,1 e 2 são abertos pelo SO antes do início da execução do programa. Em C estão encapsulados no tipo FILE* do stdio.h



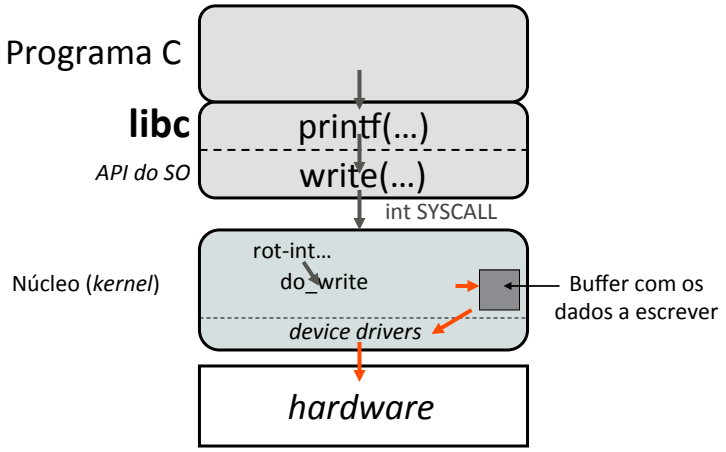
Processo e IO - Java

Os canais do SO são encapsulados em objectos InputStream e OutputStream

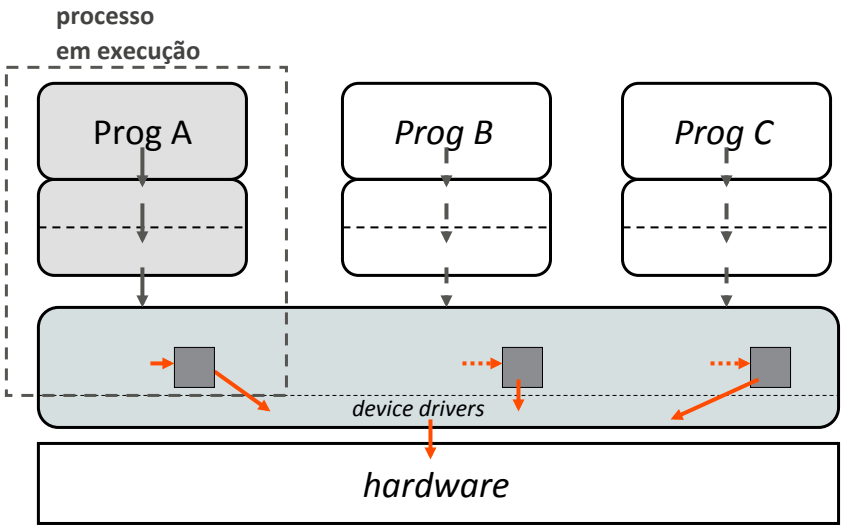


Chamadas ao sistema (I/O)

- Exemplo de saída de dados



Cada processo seu I/O



Helloworld.s

```
EXIT = 1                # usando simbolos para constantes
WRITE = 4
LINUX_SYSCALL = 0x80
.data                   # secção de dados (variáeis)
msg:   .ascii    "Hello, world!\n"  # um vetor de caracteres
len = . - msg           # len representa o tamanho do vetor
.text              # secção de código
.global _start    # exportar o simbolo _start (inicio do programa)
_start:  movl     $len,%edx      # num de bytes a escrever
         movl     $msg,%ecx      # end dos bytes a escrever
         movl     $1,%ebx        # canal onde escrever (1=stdout)
         movl     $WRITE,%eax    # pedir write ao sistema
         int      $LINUX_SYSCALL         # chama o sistema

         movl     $0,%ebx          # codigo de terminacao
         movl     $EXIT,%eax       # pedir o exit ao sistema
         int      $LINUX_SYSCALL         # chama o sistema
```


Biblioteca C

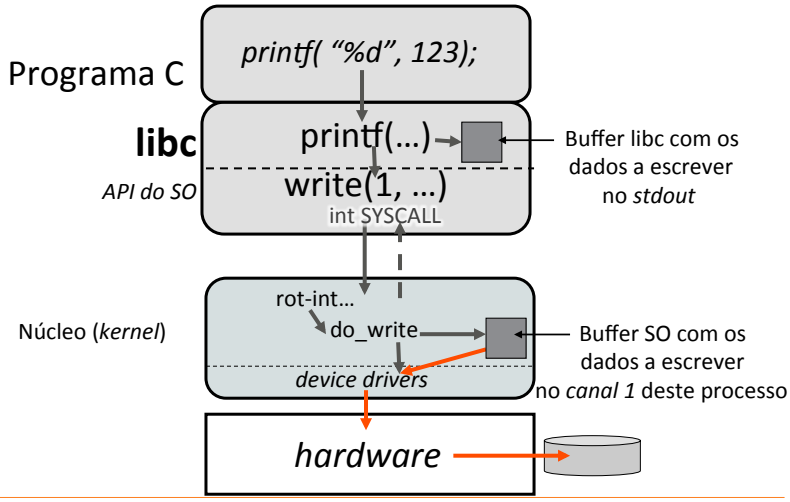
- Canais standard (abertos no inicio):
  - stdin, stdout, stderr
  - (em Java: System.in, System.out, System.err)
- Abrir/fechar:
  - FILE *fopen(char*name, char*mode);
  - fclose(FILE*stream);
- Escrever/Ler:
  - printf() / fprintf() / fwrite()
  - scanf() / fscanf() / fread()

Helloworld.c

```
#include <stdio.h>
int main( int argc, char*argv[]) {
   printf(”Hello world!\n”);
   return 0;
}
```

Dentro da libc:

```
printf(...)
     fprintf( stdout, ... )
          write( 1, ... )  // interface
               int 0x80    // com SO
```


Vários níveis no IO - exemplo

- Níveis e Buffers (exemplo de escrita):