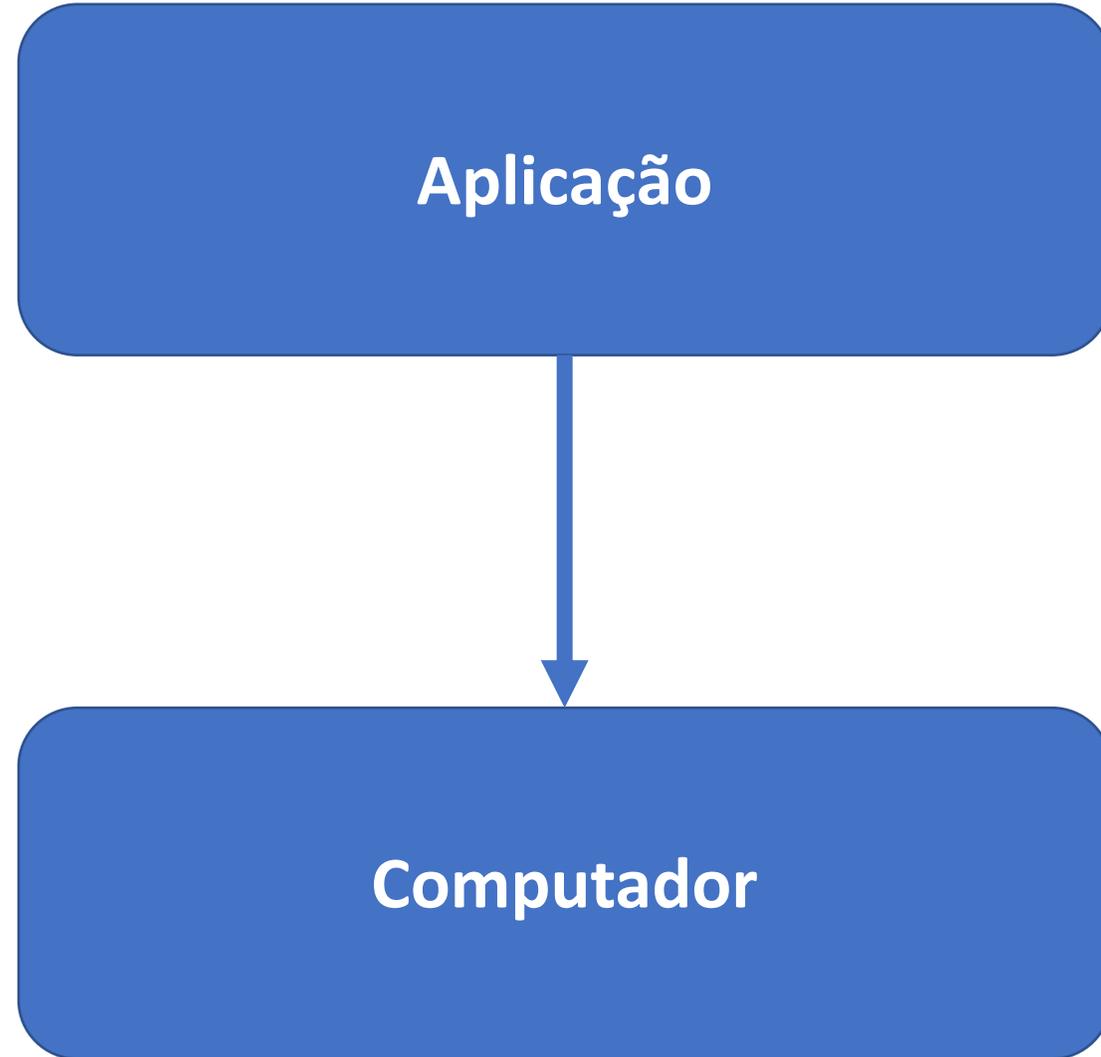
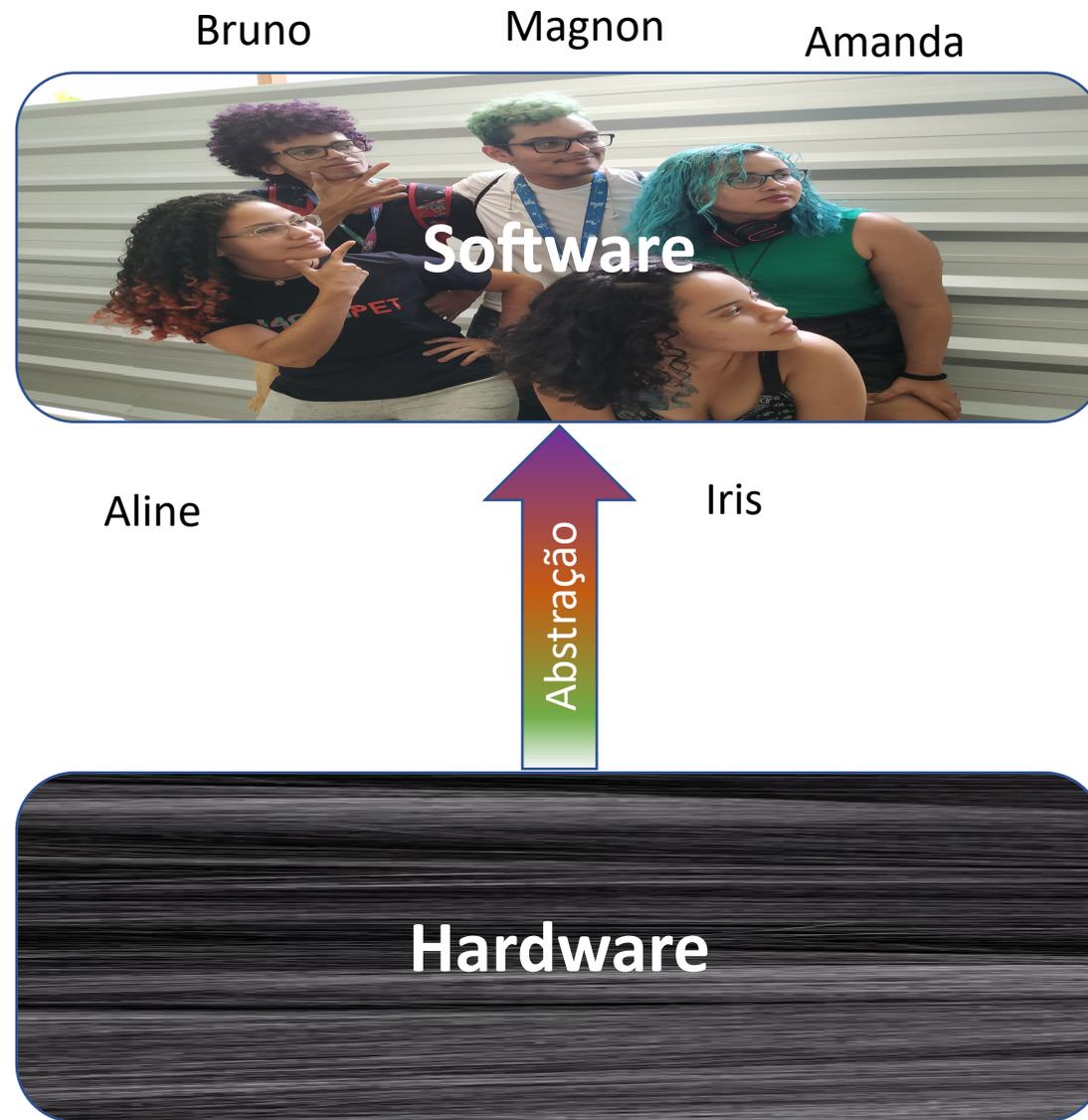


# Introdução

A Máquina Abstrata





**Mais bonito!**  
**Mais agradável!**  
**Mais fácil!**

**Mais “feio”!**  
**Mais complexo!**  
**Mais difícil!**



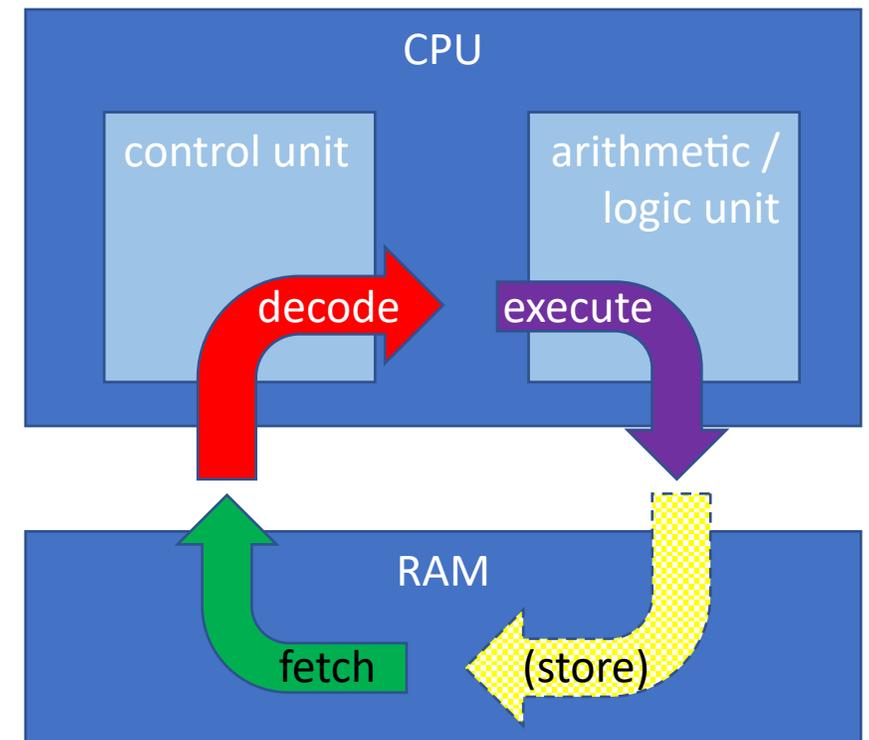


# O que acontece quando um programa executa?

Um programa em execução faz uma coisa **muito simples**: executa **instruções**!

Milhões/bilhões de vezes por segundo (MHz/GHz), o processador:

- Busca/captura (***fetch***) uma instrução da memória
- Decodifica-a (***decode***), ou seja, descobre que instrução é esta (de um conjunto de instruções)
- Executa-a (***execute***)



Isto descreve o básico do **modelo de computação** de ***Von Neumann***

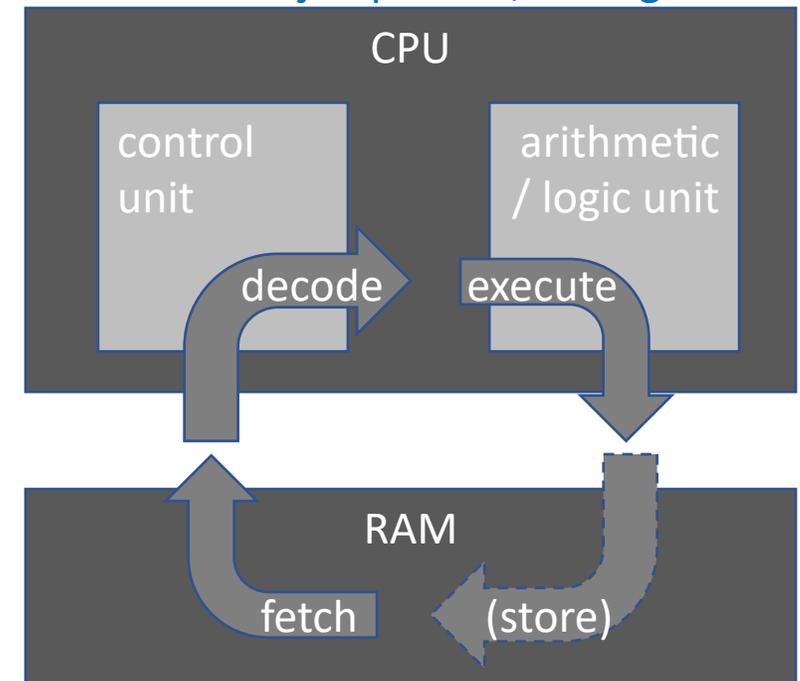


# O que acontece quando um programa executa?

**Obs.:** processadores modernos fazem coisas bizarras e assustadoras, por baixo dos panos, para fazer programas rodarem mais rápido! Executam múltiplas instruções de uma vez, entregam-nas fora de ordem etc. Mas vamos nos preocupar com o modelo simples que a maioria dos programas assumem: uma instrução por vez, entrega em ordem (sequencial).

Milhões/bilhões de vezes por segundo (MHz/GHz), o processador:

- Busca/captura (**fetch**) uma instrução da memória
- Decodifica-a (**decode**), ou seja, descobre que instrução é esta (de um conjunto de instruções)
- Executa-a (**execute**)



Isto descreve o básico do **modelo de computação de Von Neumann**



# Tornando tudo mais fácil

Há um **conjunto** de software responsável por

- Tornar fácil rodar programas (incl. vários ao mesmo tempo...) na CPU
- Fazer programas compartilharem memória
- Fazê-los interagir com dispositivos de Entrada/Saída (E/S)
- Etc....

Este conjunto de software é o **Sistema Operacional**



Máquina Abstrata

Mais fácil de usar



# Tornando tudo mais fácil

Há um **conjunto** de software responsável por

- Tornar fácil rodar programas (incl. vários ao mesmo tempo...) na **CPU**
- Fazer programas compartilharem **memória**
- Fazê-los interagir com **dispositivos de Entrada/Saída (E/S)**
- Etc....

Este conjunto de software é o **Sistema Operacional**

Máquina Abstrata

Mais fácil de usar

Gerenciador de Recursos

Operação correta e eficiente



# Virtualização

- SO transforma um **recurso físico** em uma **forma virtual** mais geral (não específica – ex. **SSD Samsung 860 EVO** → simplesmente **disco**) e fácil de usar
- Por isso, às vezes o SO é também chamado de **máquina virtual**
- O SO oferece **interfaces (APIs)** para aplicações
  - **System Calls**
  - Biblioteca padrão



# O X da questão: como virtualizar recursos?

- Por que? **Virtualização** faz o sistema ser **mais fácil** de usar
- Mas **como?**
  - Quais mecanismos e políticas são implementados pelo sistema operacional para obter virtualização?
  - Como o sistema operacional faz isso de forma eficiente?
  - Que suporte de hardware é necessário?

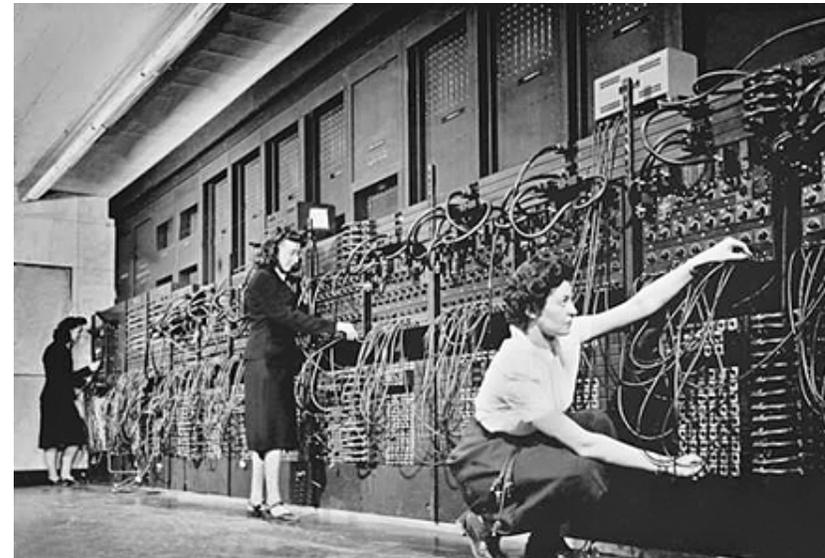
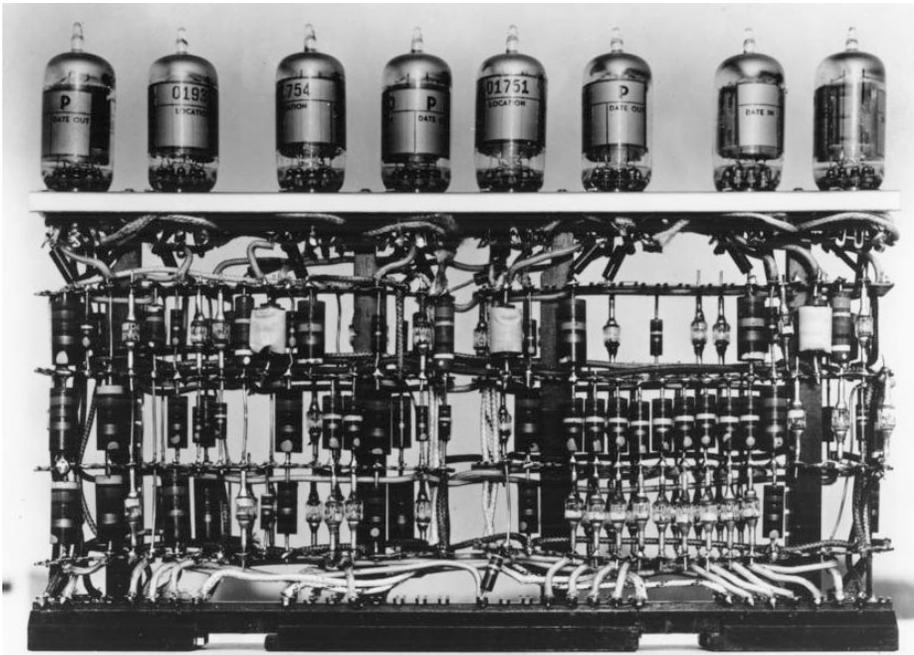


# INTRODUÇÃO

Um pouco de HISTÓRIA

# História dos Sistemas Operacionais

- Primeira geração: 1945 - 1955
  - Válvulas, painéis de programação



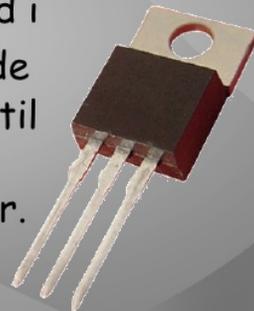
# História dos Sistemas Operacionais

- Primeira geração: 1945 - 1955
  - Válvulas, painéis de programação
- Segunda geração: 1955 - 1965
  - **transistores**, sistemas em lote

## Second Generation - 1956-1963: Transistors

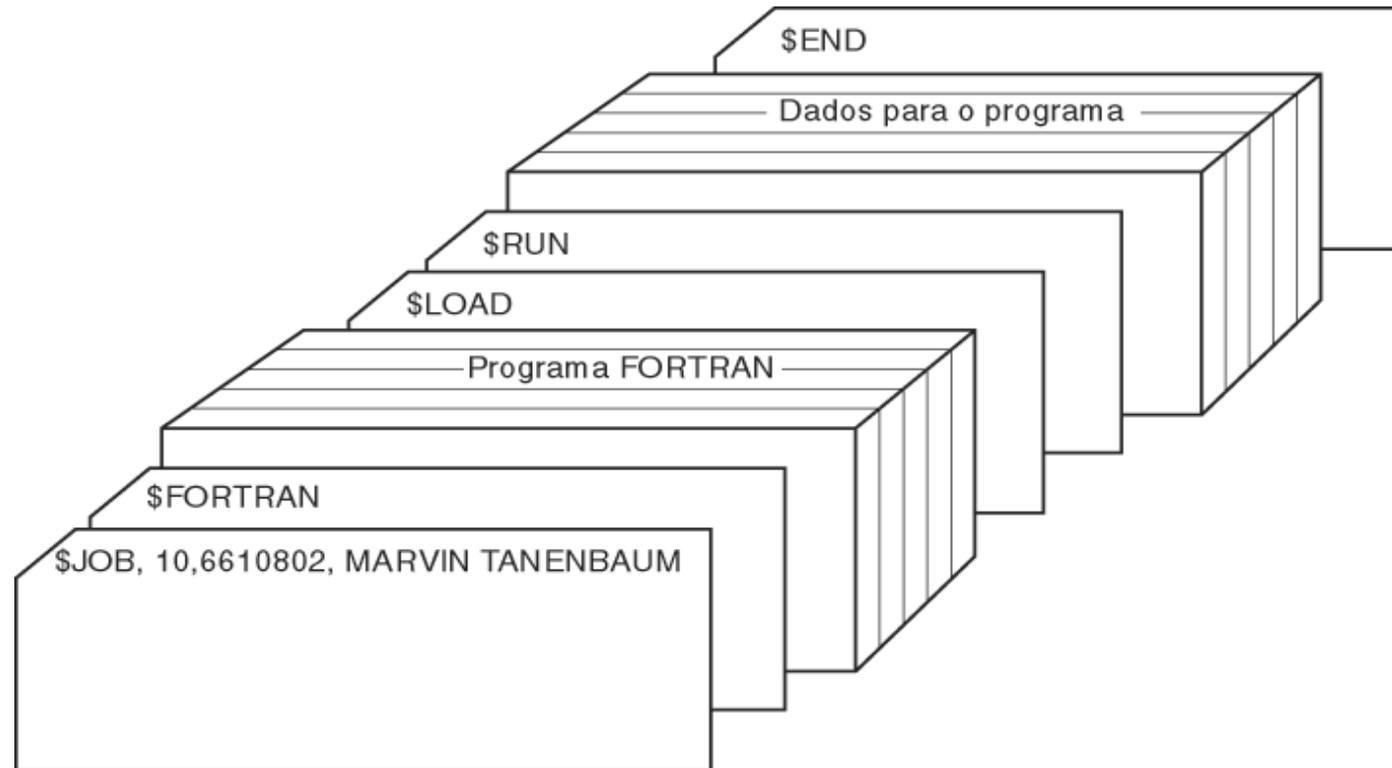
**Transistors** replaced vacuum tubes and ushered in the second generation of computers. The transistor was invented in 1947 but did not see wide spread use in computers until the late 50s.

\*smaller, faster and cheaper.



# Sistemas em lote

(não necessariamente executa no momento em que é submetido)



- Estrutura de um job típico (**lote** de cartões) – 2a. geração

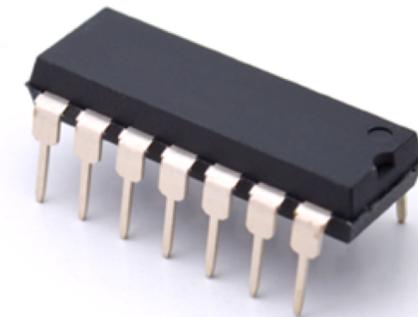
# História dos Sistemas Operacionais

- Primeira geração: 1945 - 1955
  - Válvulas, painéis de programação
- Segunda geração: 1955 - 1965
  - transistores, sistemas em lote
- Terceira geração: 1965 – 1980
  - **ICs (circuitos integrados)** e multiprogramação



Vacuum tubes: slow, expensive, fragile

Transistors: much simpler, much smaller, much cheaper, more reliable, no warm up, much faster.

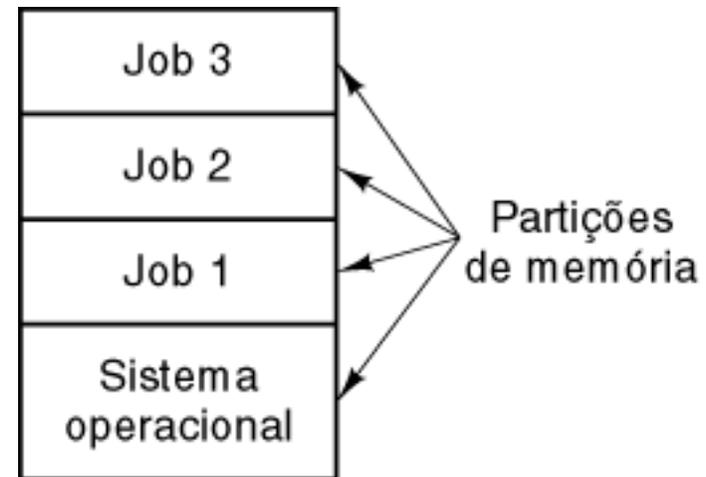


Integrated circuits: miniaturization added to all the existing benefits, enabled unthought-of possibilities

# História dos Sistemas Operacionais

- Primeira geração: 1945 - 1955
  - Válvulas, painéis de programação
- Segunda geração: 1955 - 1965
  - transistores, sistemas em lote
- Terceira geração: 1965 – 1980
  - CIs (circuitos integrados) e multiprogramação

# Multiprogramação



Três jobs na memória – 3a. geração

# História dos Sistemas Operacionais

- Primeira geração: 1945 - 1955
  - Válvulas, painéis de programação
- Segunda geração: 1955 - 1965
  - transistores, sistemas em lote
- Terceira geração: 1965 – 1980
  - CIs (circuitos integrados) e multiprogramação
- Quarta geração: 1980 – presente
  - **Computadores pessoais**
- Hoje: onipresença – **computação ubíqua**

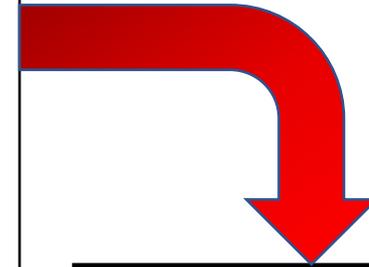


# INTRODUÇÃO

Virtualizando a CPU

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];
    while (1) {
        Spin(1);
        // checa o tempo e retorna depois de rodar por 1 segundo
        printf("%s\n", str);
    }
    return 0;
}
```



```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

Compilando &  
Executando





# Executando várias instâncias do mesmo programa

```
prompt> ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...
```

**Interessante!!!**  
As sequências não se repetem...

- Torna uma única CPU em um número “infinito” de **CPUs virtuais**
- O SO, com alguma ajuda do hardware (interrupções etc.), é o responsável por esta **ILUSÃO!!!**



Dois  
programas  
estão  
prontos  
para rodar;  
qual deve  
rodar  
primeiro?

# SO: Gerenciador de Recursos

- **Políticas** (do SO) são usadas para responder perguntas como estas
  - Definição [Wikipedia]:
    - Uma política é um sistema de *princípios* para orientar decisões e alcançar resultados racionais
    - Uma política é implementada como um *procedimento* ou *protocolo*
- O SO implementa **mecanismos**, como a habilidade de executar múltiplos programas de uma vez
- Neste exemplo, quem decide é o **escalonador**



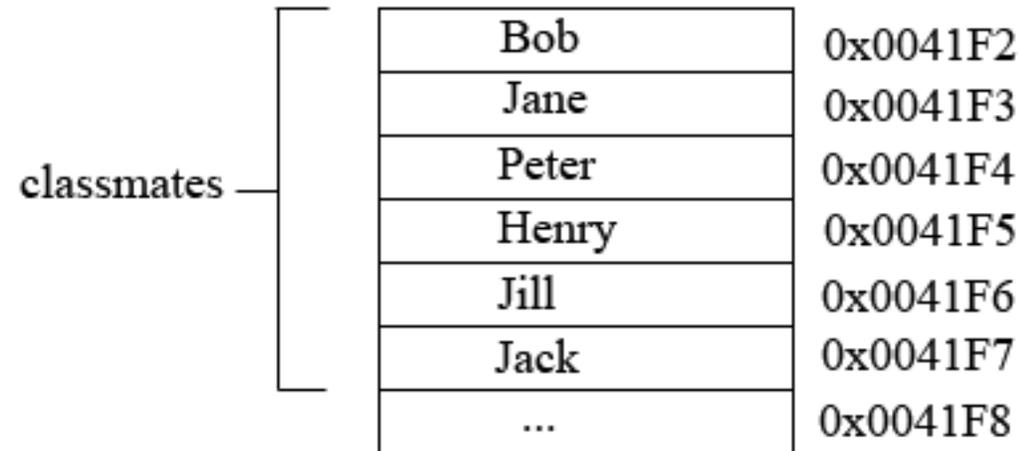
# INTRODUÇÃO

Virtualizando Memória



# Modelo Simples

- Um **vetor** (*array*) de bytes
- Para **ler** especifica um **endereço**
- Para **escrever** (ou atualizar) especifica um **endereço** e os **dados** a serem escritos no endereço dado





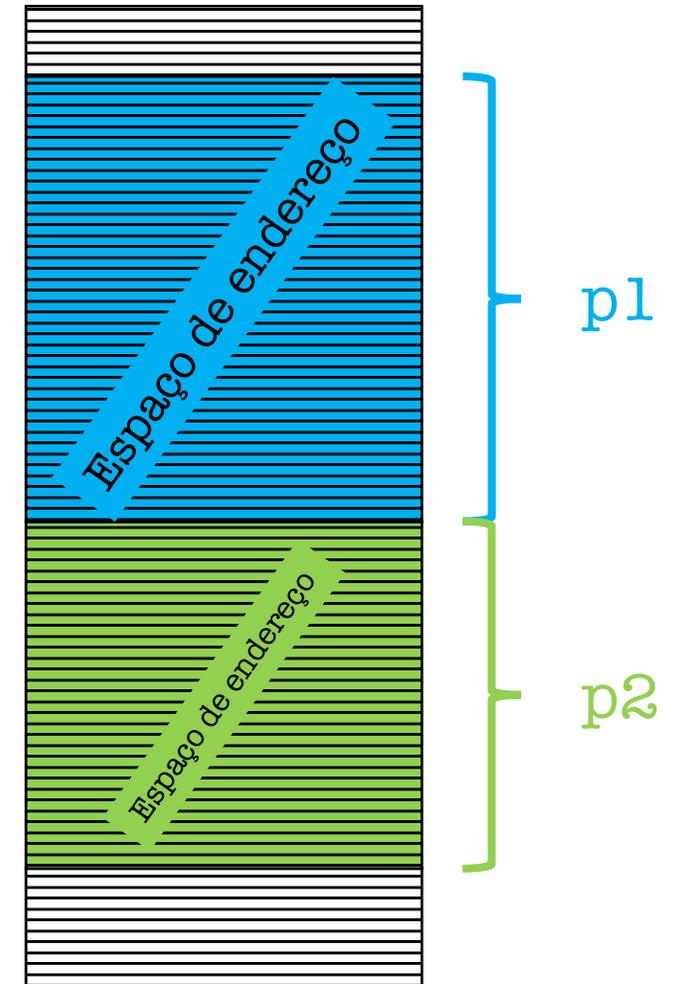
# Dados e Instruções

- Um programa mantém todas as suas **estruturas de dados** na memória
  - Acessa via **instruções**, como `load` e `store`
- As `instruções` do programa também são mantidas na memória



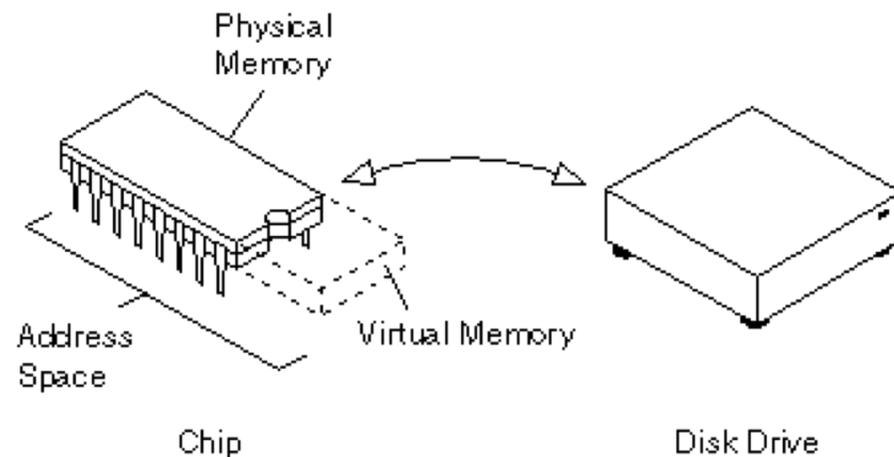
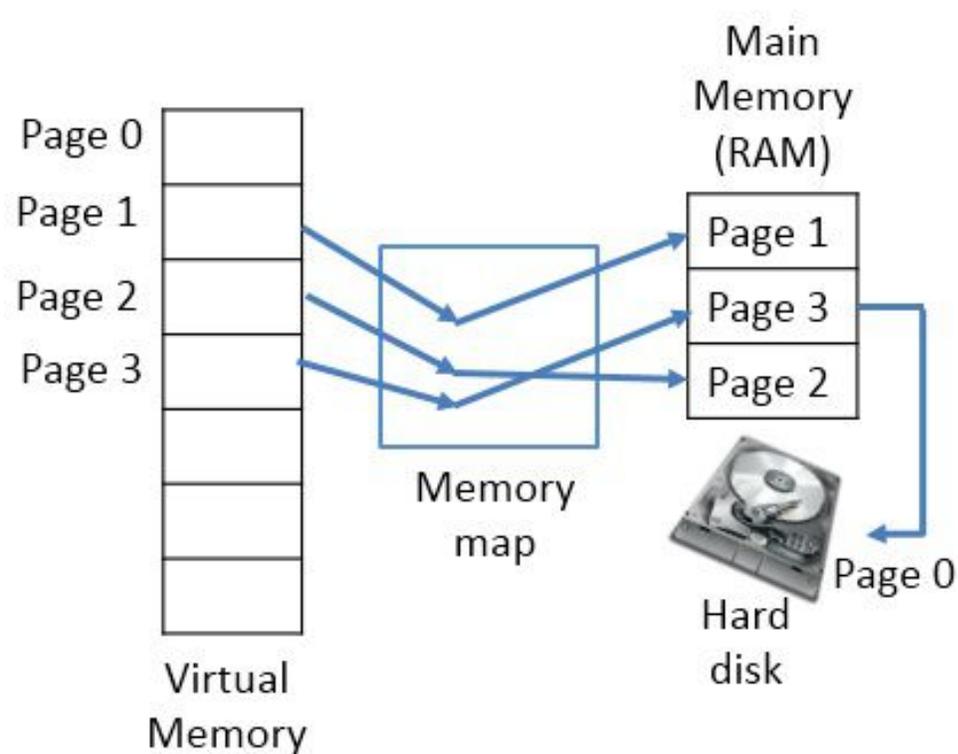
# Virtualizando a memória

- Cada processo acessa seu próprio **espaço de endereço [virtual]**, que
- O SO mapeia para a memória física da máquina e
- Gerencia este recurso compartilhado (memória física)





# Não confundir com o conceito de Memória Virtual



**ILUSÃO** de que a memória é muito maior!

Memória virtual > Memória real (física)



# INTRODUÇÃO

Concorrência

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter = counter + 1;
    }
    return NULL;
}
```

## (continua)

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: no_threads <loops>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    printf("Initial value : %d\n", counter);
    worker(NULL);
    worker(NULL);
    printf("Final value  : %d\n", counter);
    return 0;
}
```

**Analise este programa:**  
quais os valores **inicial** e **final**?

```
prompt> gcc -o no_threads no_threads.c -Wall
prompt> ./no_threads 1000
Initial value : 0
Final value : 2000
```



```
prompt> ./no_threads 100000
Initial value : 0
Final value : 200000
```

**Initial = 0**  
**Final = 2 x loops**



```

#include <stdio.h>
#include <stdlib.h>
#include "common.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter = counter + 1;
    }
    pthread_exit(NULL);
}

```

(continua)

```

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <loops>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value : %d\n", counter);
    return 0;
}

```

**Analise este programa:**  
quais os valores inicial e final?

```

prompt> gcc -o threads threads.c -Wall -pthread
prompt> ./threads 1000
Initial value : 0
Final value : 2000

```

```

prompt> ./threads 100000
Initial value : 0
Final value : 113144

```



```

prompt> ./threads 100000
Initial value : 0
Final value : 109904

```





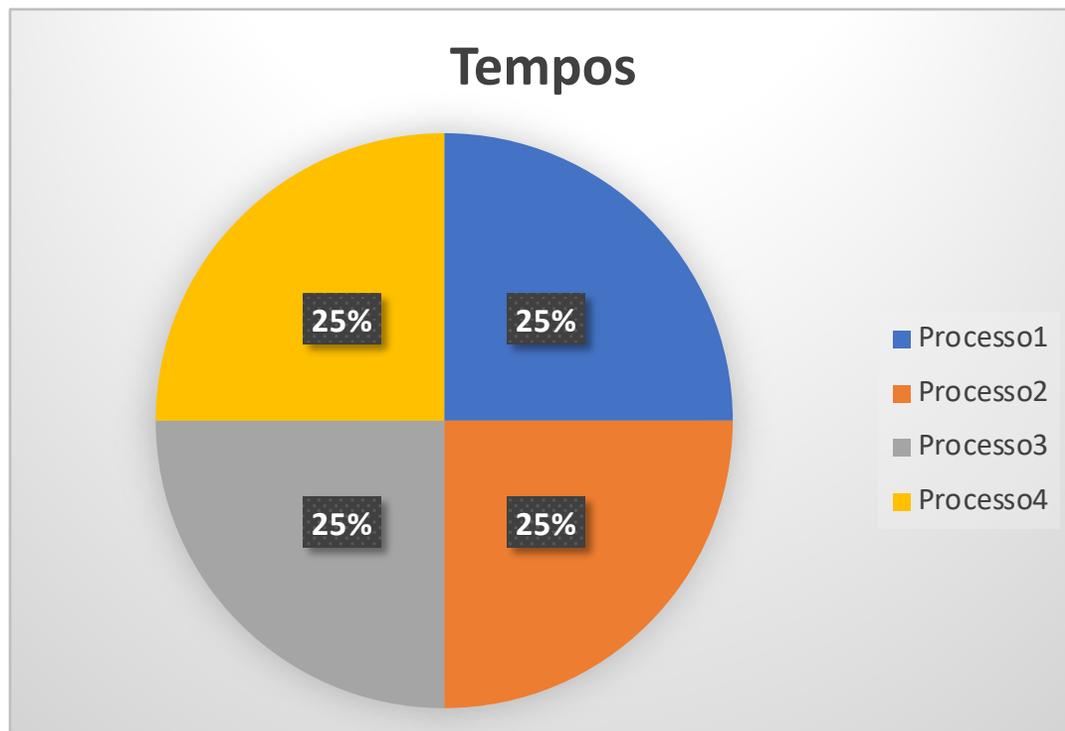
# Explicação

- `counter = counter + 1;`
- **3 instruções** de máquina:
  - `LOAD ...`
  - `ADD ...`
  - `STORE ...`
- que não executam **atomicamente**, ou seja, como se fossem indivisíveis (apenas uma instrução)
- A sequência pode ser **interrompida** a qualquer momento (em qualquer ponto)

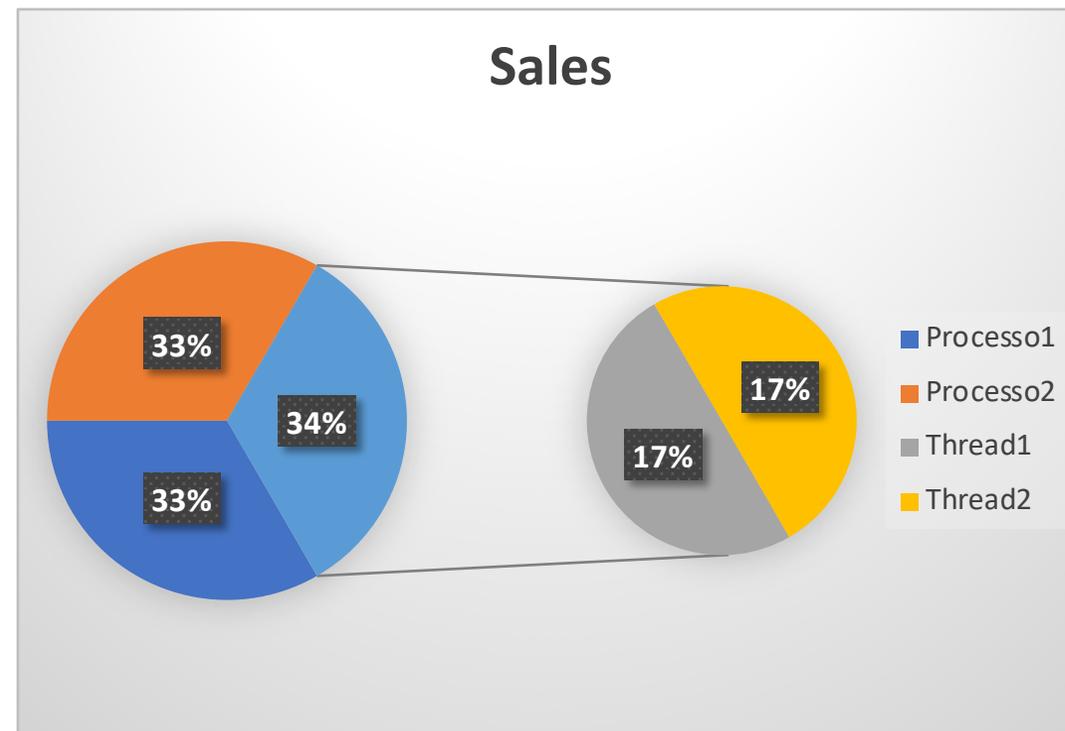


# Interrupções podem ocorrer

## Entre processos



## Entre *threads*



```

#include <stdio.h>
#include <stdlib.h>
#include "common.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter = counter + 1;
    }
    pthread_exit(NULL);
}

```



(continua)

```

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <loops>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value : %d\n", counter);
    return 0;
}

```

Onde está o problema?

Por quê?

```

prompt> gcc -o threads threads.c -Wall -pthread
prompt> ./threads 1000
Initial value : 0
Final value : 2000

```

```

prompt> ./threads 100000
Initial value : 0
Final value : 113144

```



```

prompt> ./threads 100000
Initial value : 0
Final value : 109904

```



Paramos aqui...

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

pthread_mutex_t m;
volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&m);
        counter = counter + 1;
        Pthread_mutex_unlock(&m);
    }
    pthread_exit(NULL);
}
```

(continua)

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <loops>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
    Pthread_mutex_init(&m, NULL);
    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value : %d\n", counter);
    return 0;
}
```



## A Solução: Mutex



# INTRODUÇÃO

Persistência



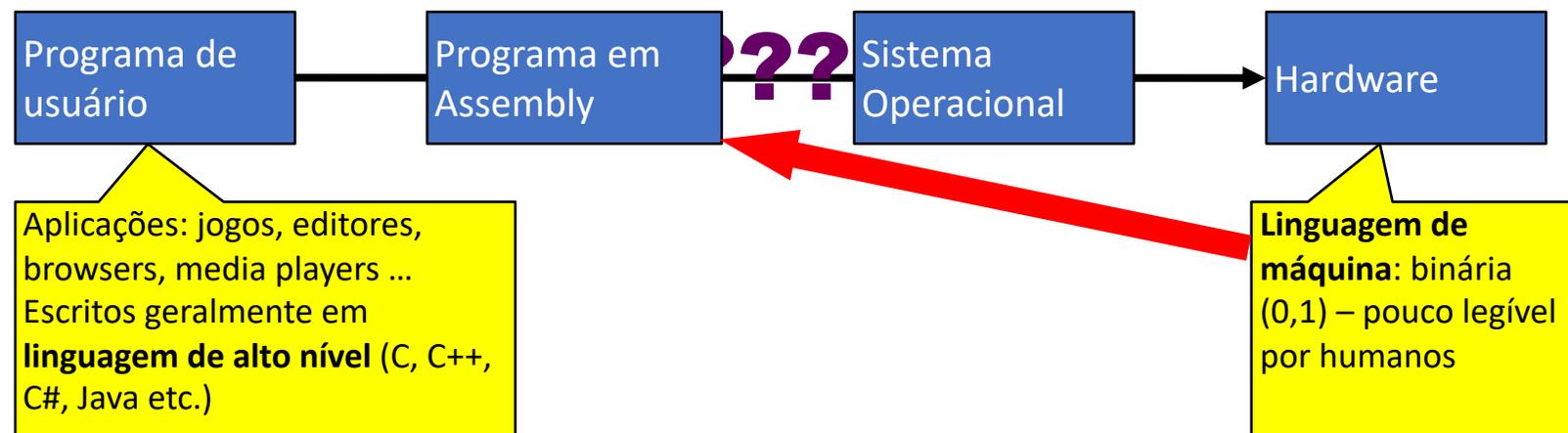
# Após CPU e Memória...

- É importante ter hardware e software que possam armazenar dados de forma **não-volátil, persistente**
- O hardware: **dispositivo de E/S**, como HD, SSD
- O software que gerencia o disco é o **Sistema de Arquivo**, uma forma confiável e eficiente de armazenar **arquivos – abstrações!!**
  - Característica marcante: não apenas o recurso físico (disco) é **compartilhado**, mas também as informações armazenadas nos arquivos



# Software Básico

[A. Raposo e M. Endler, PUC-Rio, 2008]



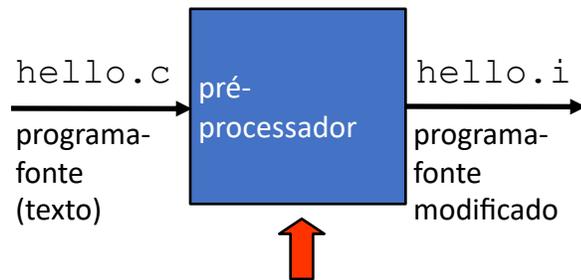
“Conhecendo mais sobre o que está ‘por baixo’ do programa, você pode escrever programas mais eficientes e confiáveis”



# Gerando um arquivo executável

```
unix> gcc -o hello hello.c
```

```
1. #include <stdio.h>
2. int main()
3. {
4.     printf("hello, world\n");
5. }
```



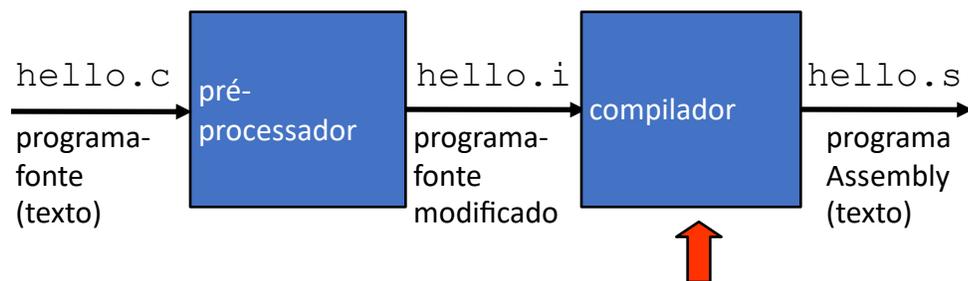
- Modifica o programa em C de acordo com diretivas começadas com #
  - Ex.: `#include <stdio.h>` diz ao pré-processador para ler o arquivo `stdio.h` e inseri-lo no programa fonte
- O resultado é um programa expandido em C, normalmente com extensão `.i`, em Unix



# Gerando um arquivo executável

```
unix> gcc -o hello hello.c
```

```
1. #include <stdio.h>
2. int main()
3. {
4.     printf("hello, world\n");
5. }
```



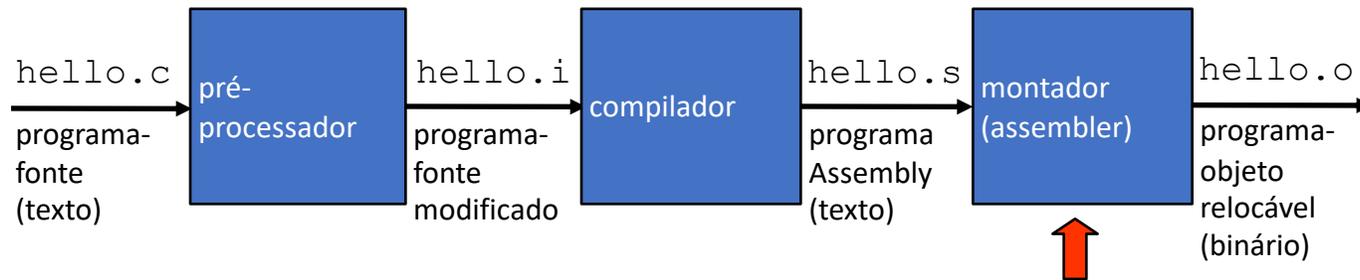
- **Compilador** traduz o programa .i em um programa em Assembly
  - É o formato de saída comum para os compiladores nas várias linguagens de programação de alto nível
    - i.e., programas em C, Java, Fortran, etc vão ser traduzidos para a mesma linguagem Assembly



# Gerando um arquivo executável

```
unix> gcc -o hello hello.c
```

```
1. #include <stdio.h>
2. int main()
3. {
4.     printf("hello, world\n");
5. }
```



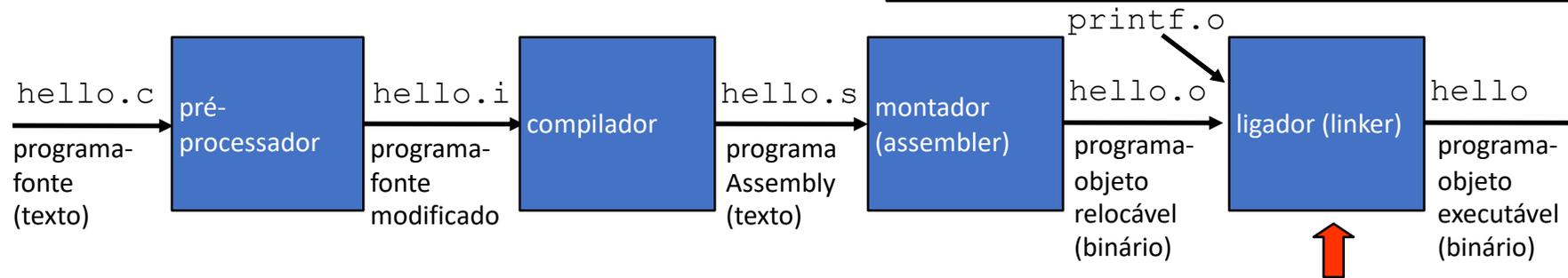
- **Montador** (Assembler) transforma o programa em Assembly em um programa binário em linguagem de máquina (chamado programa-objeto)
  - Os módulos de programas, compilados ou montados, são armazenados em um formato intermediário ("*Programa-Objeto Relocável*" - extensão .o)
- Endereços de acesso e a posição do programa na memória ficam **indefinidos**



# Gerando um arquivo executável

```
unix> gcc -o hello hello.c
```

```
1. #include <stdio.h>
2. int main()
3. {
4.     printf("hello, world\n");
5. }
```



- O **ligador** (linker) gera o programa executável a partir do `.o` gerado pelo assembler
  - No entanto, pode haver funções-padrão da linguagem (ex., `printf`) que não estão definidas no programa, mas em outro arquivo `.o` pré-compilado (`printf.o`)
  - O ligador faz a junção dos programas-objeto (`.o`) necessários para gerar o executável



# Analizando um programa que cria um arquivo (`/tmp/file`) que armazena a mensagem “hello world”

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <assert.h>
4 #include <fcntl.h>
5 #include <sys/types.h>
6
7 int
8 main(int argc, char *argv[])
9 {
10  int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
11  assert(fd > -1);
12  int rc = write(fd, "hello world\n", 13);
13  assert(rc == 13);
14  close(fd);
15  return 0;
16 }
```

## Chamadas ao Sistema

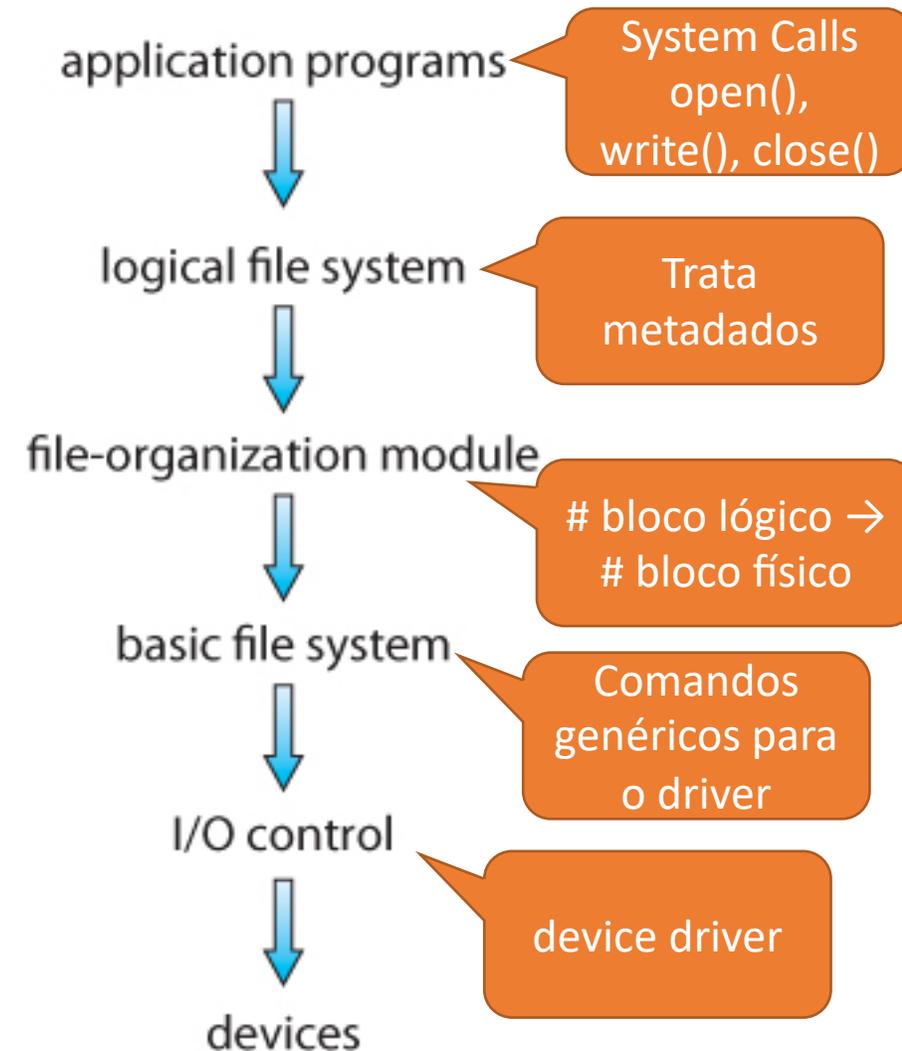
(*system calls*) são roteadas para o **sistema de arquivo**, que lida com os pedidos e retorna algum código (de erro) para o usuário



Por oferecer System Calls, o SO também é visto como uma **biblioteca padrão** (*standard library*)

## O que o SO/Sistema de Arquivos faz para escrever o arquivo no disco?

- Primeiro, é preciso descobrir onde no disco esses novos dados residirão – **alocação**
- Depois, emite solicitações de E/S para o dispositivo de armazenamento subjacente (disco), para ler estruturas existentes ou atualizá-las (gravar)



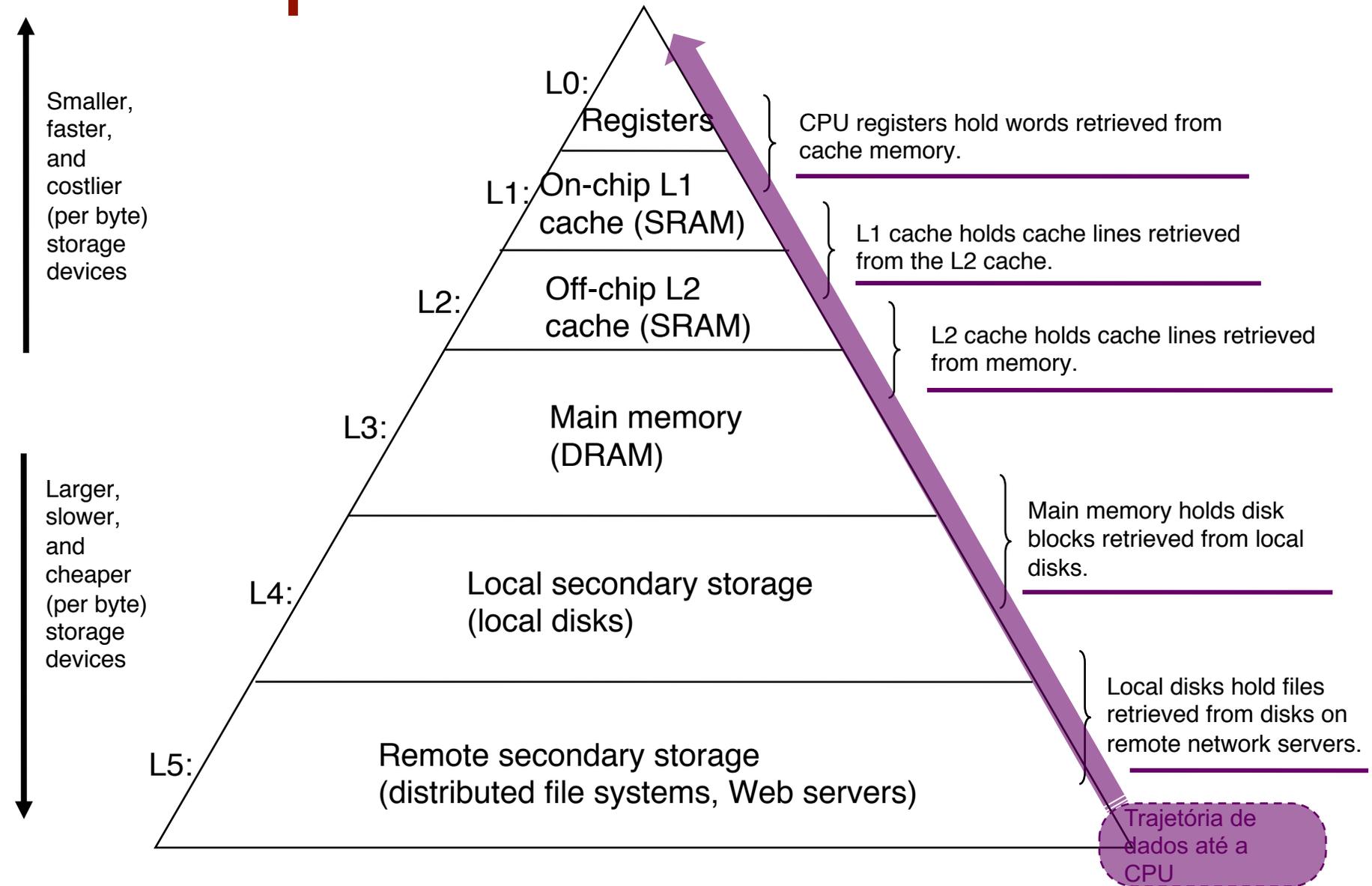


# Algumas curiosidades

- Para melhor **desempenho**, a maioria dos sistemas de arquivos atrasam `writes`, deixando para fazê-los em lotes
- Para lidar com **falhas** (*crashes*) durante `writes`, os mesmos são ordenados para, se uma falha ocorrer durante a sequência de `writes`, o sistema pode se recuperar a partir de um estado anterior ordenado (o último estado correto)



# Hierarquia de Memória





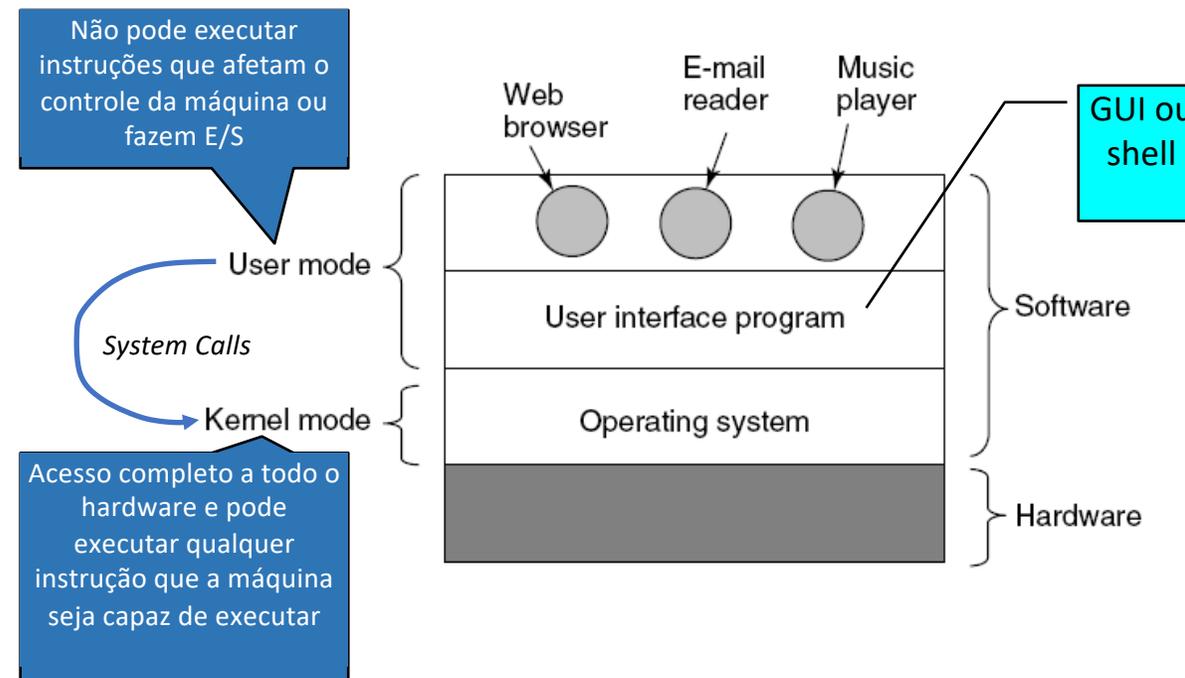
# INTRODUÇÃO

Sistemas Operacionais



# Modos de Operação

- Modo Usuário
- Modo Núcleo





# Diversidade

- Sistemas operacionais de **computadores de grande porte** (*mainframe*)
- Sistemas operacionais de servidores / **redes**
- Sistemas operacionais de **multiprocessadores** (paralelismo)
- Sistemas operacionais de computadores pessoais
- Sistemas operacionais de dispositivos portáteis/ **móveis** (ex. celulares)
- Sistemas operacionais de **tempo-real**
- Sistemas operacionais **embarcados**
- Sistemas operacionais de cartões inteligentes
- Sistemas operacionais de sensores

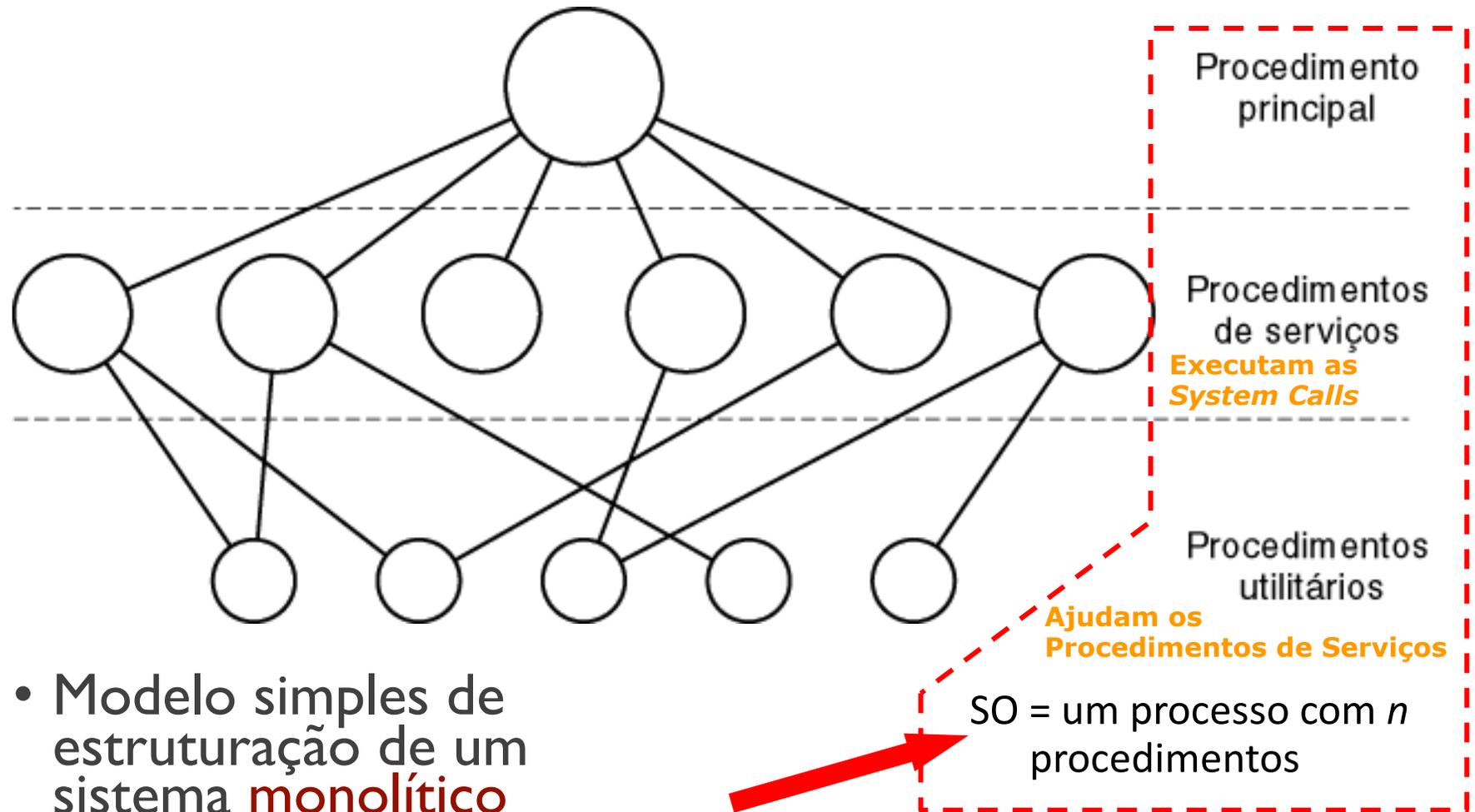


# Estruturas

- Monolítica
- Em camadas
- Cliente-Servidor
- Virtualização



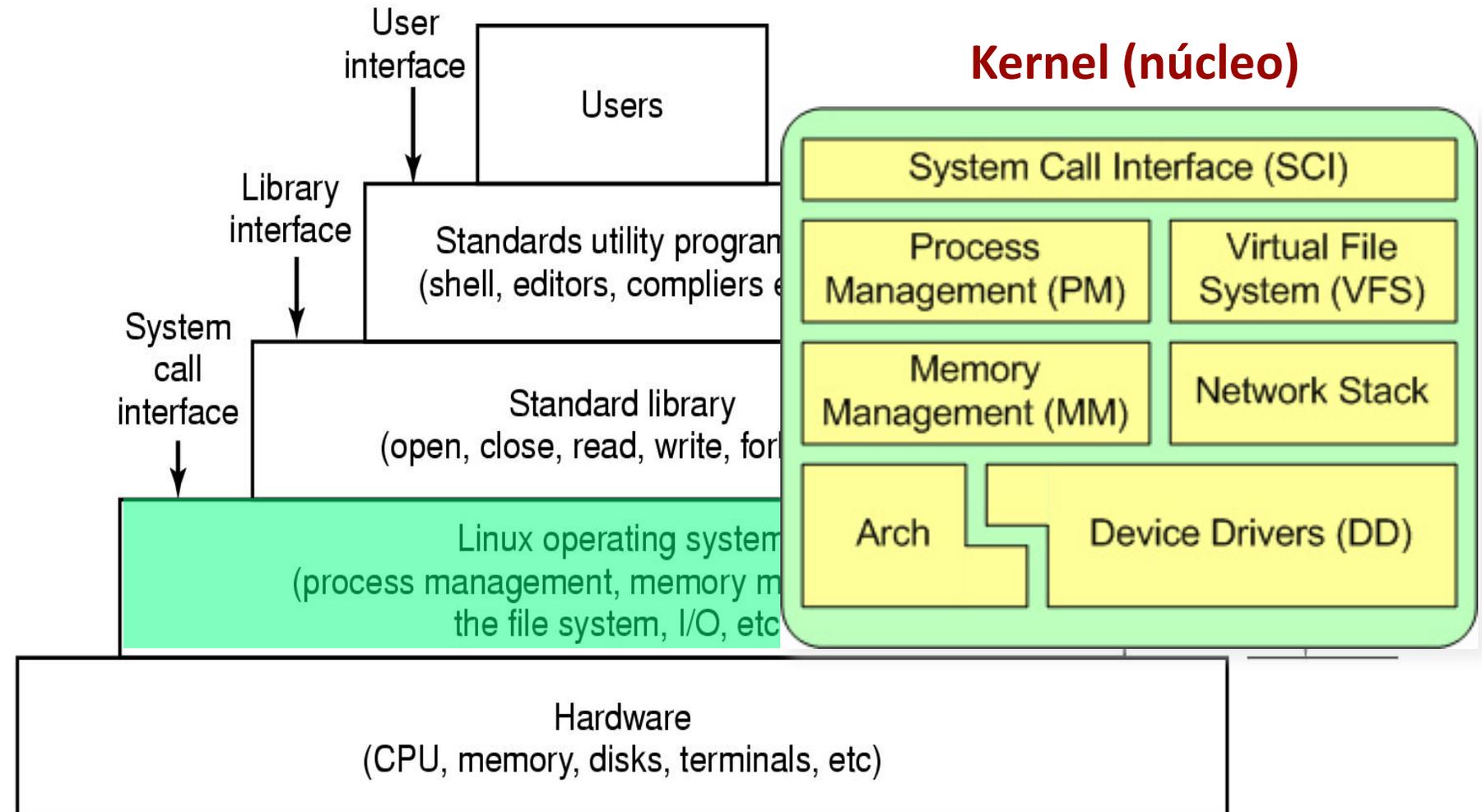
# Estrutura: Sistema Monolítico



- Modelo simples de estruturação de um sistema **monolítico**

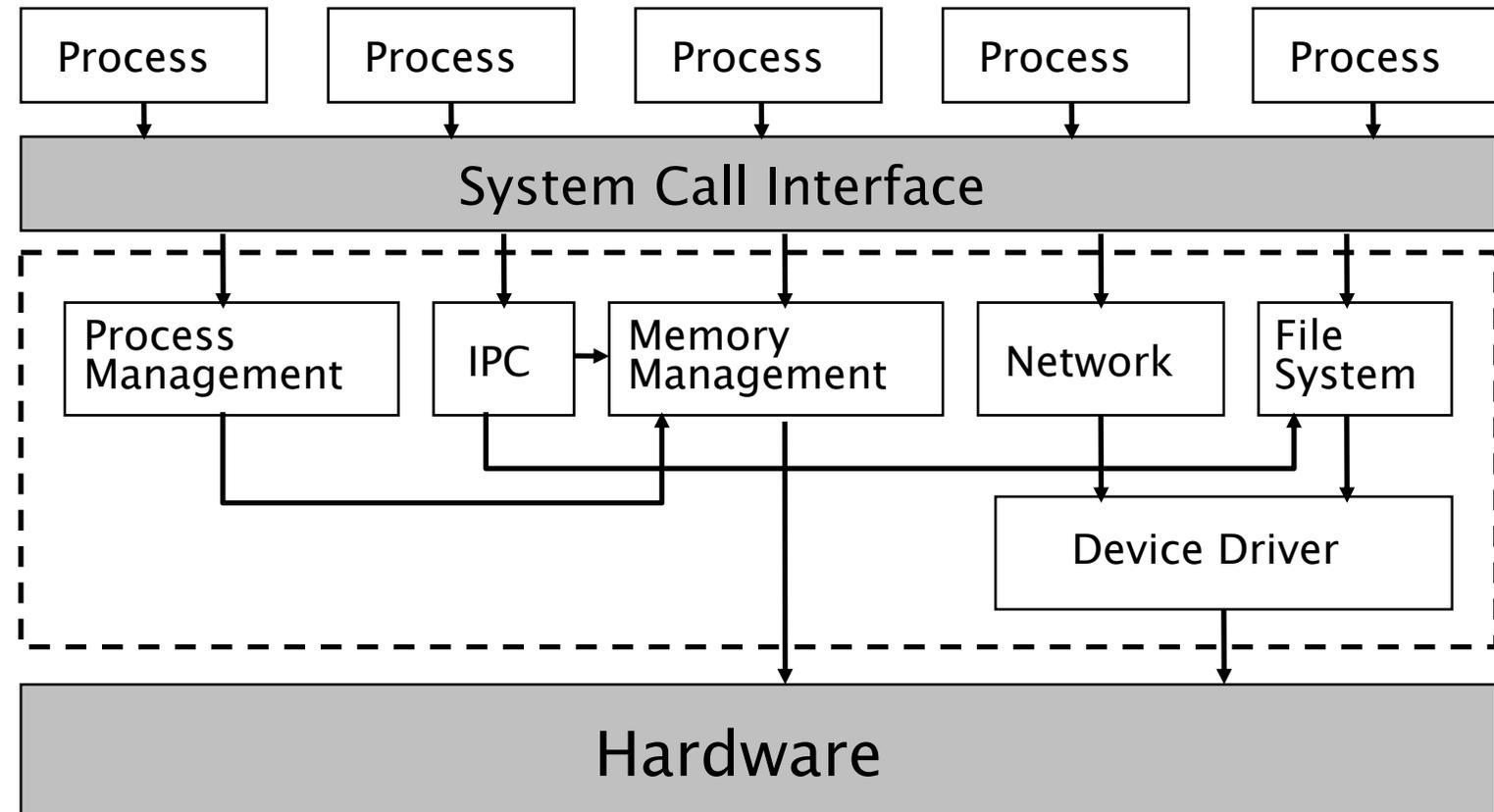


# Camadas em Linux





# Linux Kernel: Relacionamentos



# APPLICATIONS

Home    Contacts    Phone    Browser    ...

# APPLICATION FRAMEWORK

Activity Manager    Window Manager    Content    View    Notification Manager  
Package Manager    **Telephony Manager**    GTalk Service

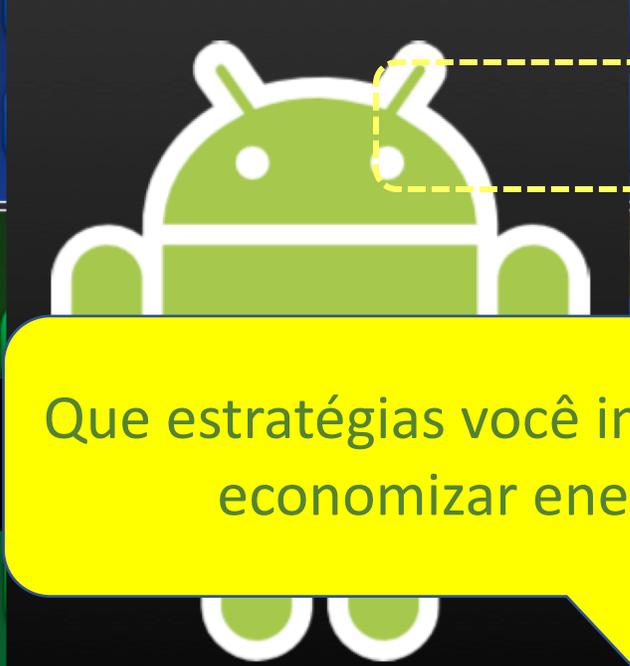
# LIBRARIES

Surface Manager    Media Framework  
OpenGL | **ANDROID**  
SGL    SSL

# ANDROID RUNTIME

# LINUX KERNEL

Display Driver    Camera Driver    Bluetooth Driver    Flash Memory Driver    Binder (IPC) Driver  
USB Driver    Keypad Driver    WiFi Driver    Audio Drivers    **Power Management**

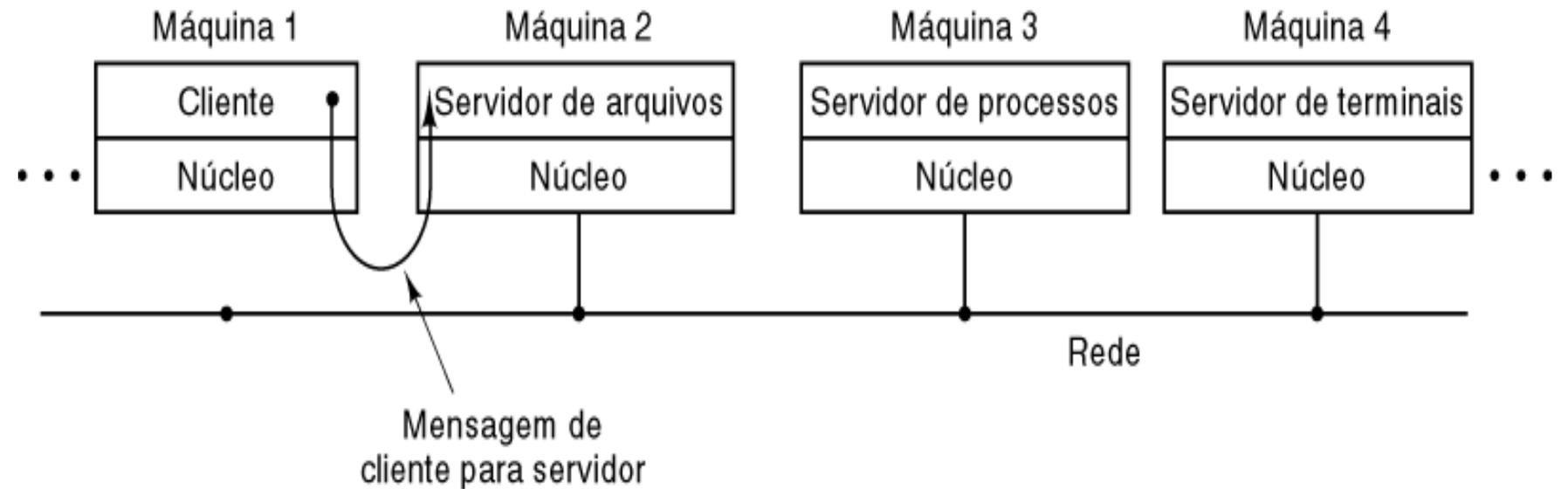


Que estratégias você imagina para economizar energia?





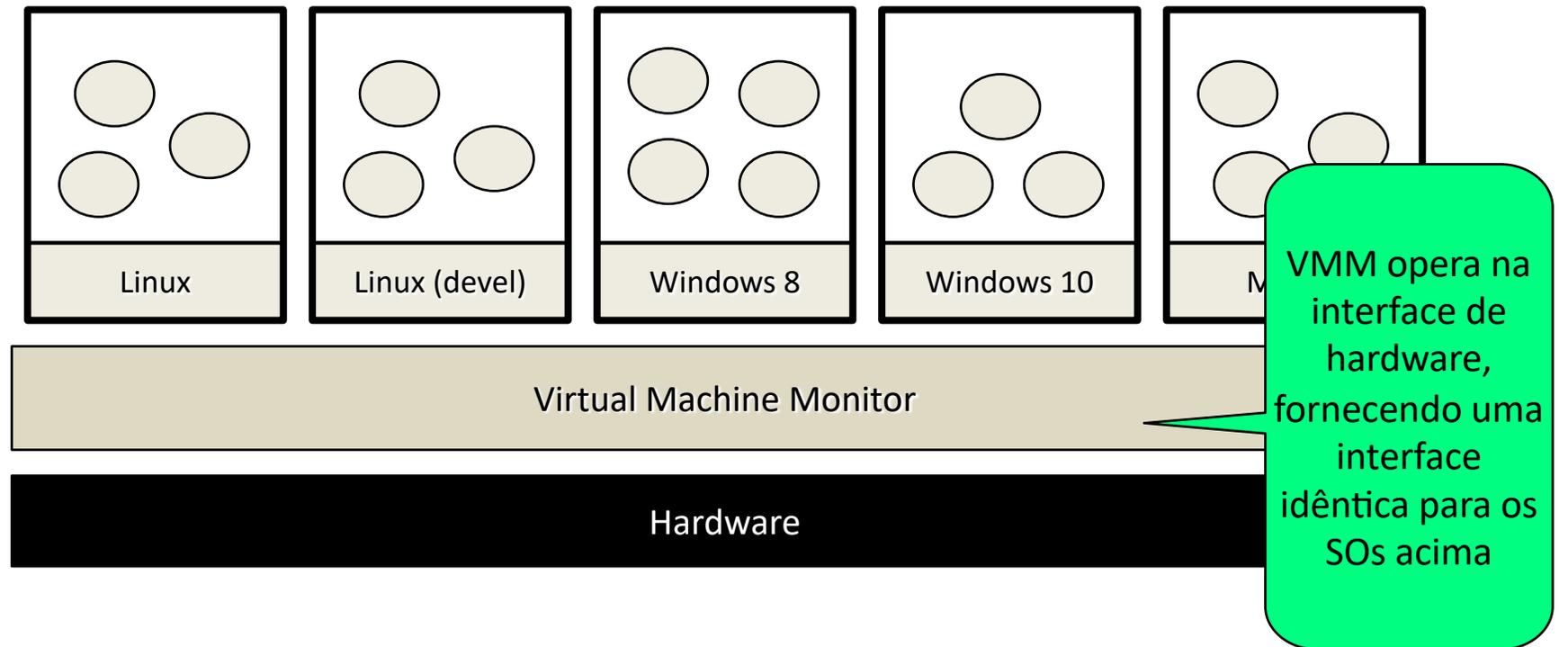
# Estrutura de Sistemas Operacionais: Cliente-Servidor



O modelo **cliente-servidor** em um **sistema (operacional) distribuído**

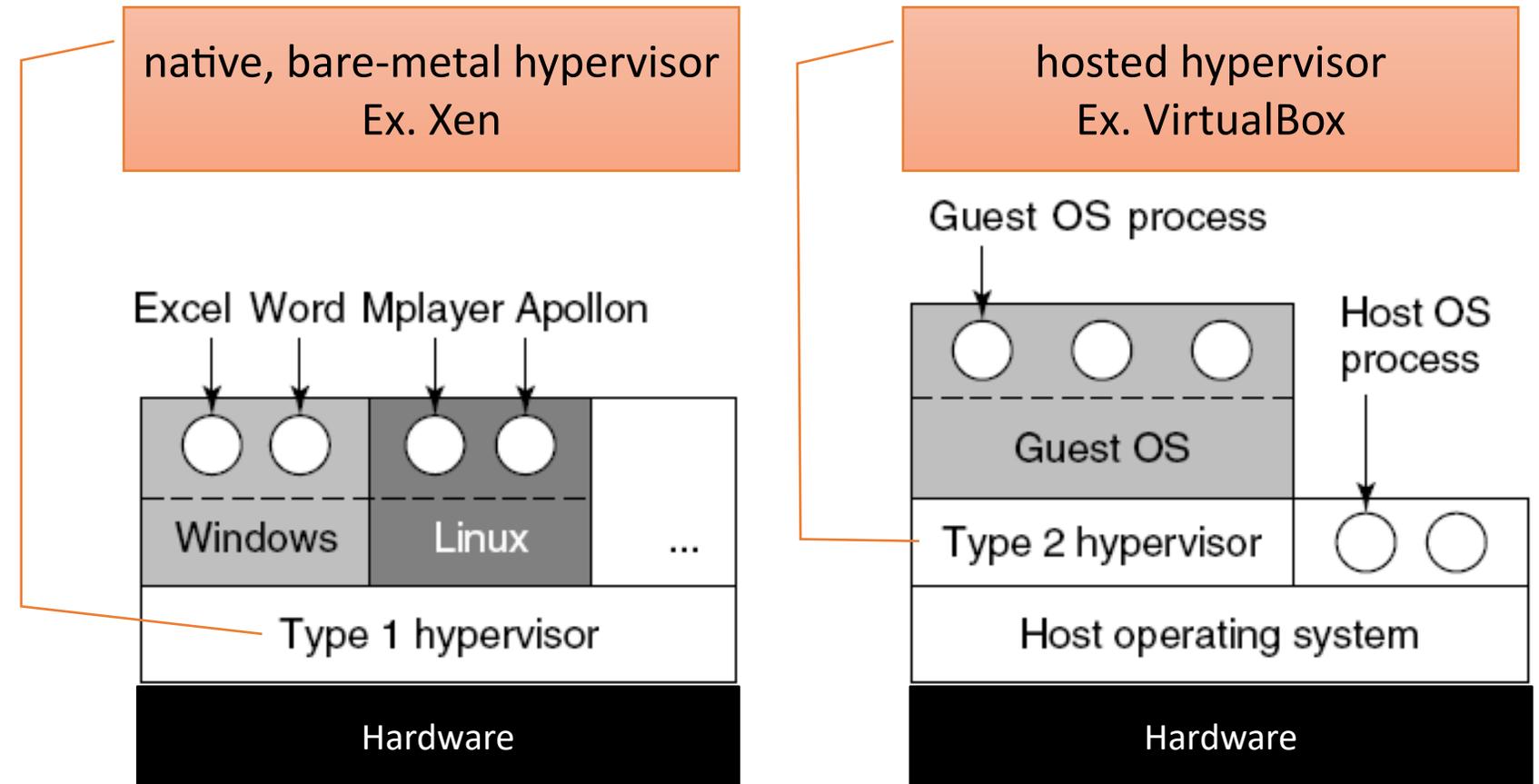


# Estrutura de Sistemas Operacionais: Máquina Virtual (Virtualização)





# Virtual Machines: Tipos (Arquiteturas)



**Hipervisor Tipo 1**

**Hipervisor Tipo 2**



# SISTEMAS OPERACIONAIS

Objetivos de Projeto



# Alguns Objetivos

- Alto **desempenho**: minimização de sobrecargas (tempo e espaço)
  - Ex.: disco – leitura/escrita em bloco, mas qual o tamanho ideal do bloco?
- **Proteção** entre aplicações e entre o SO e aplicações: **isolamento** é um princípio chave
- **Segurança**
- **Confiabilidade**
- **Eficiência energética**
- **Mobilidade**



# Características em destaque

- **Virtualização** de recursos físicos: “ilusões”
- **Concorrência**
- Armazenamento **persistente**
- **Gerência de recursos**: otimização de uso
- **Abstrações**: conveniência e facilidade de uso