

**MC514**  
**Sistemas Operacionais:**  
**Teoria e Prática**  
1s2006

**Gerenciamento de Memória**

# Sonho dos usuários

Idealmente, a memória deveria ser

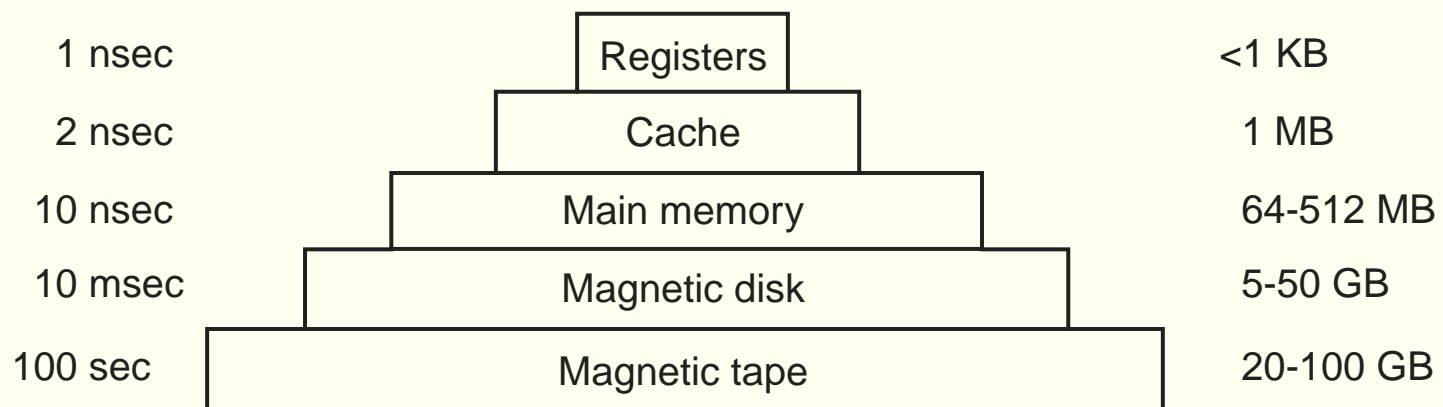
- rápida,
- de custo baixo,
- imensa e
- não volátil.

# Realidade para os usuários

## Hierarquia de Memória

Typical access time

Typical capacity



# Hierarquia de Memória

## Registradores

- Internos à CPU
- Extremamente rápidos
- Otimizações de código podem mover temporariamente variáveis para registradores.
- Programas podem dar palpites sobre o que deve ficar armazenado nos registradores

```
register int r;
```

# Hierarquia de Memória

## Cache

- Internos ou muito próximos à CPU
- Divididos em linhas de cache
- Controlados por hardware
- Cache hit
- Cache miss

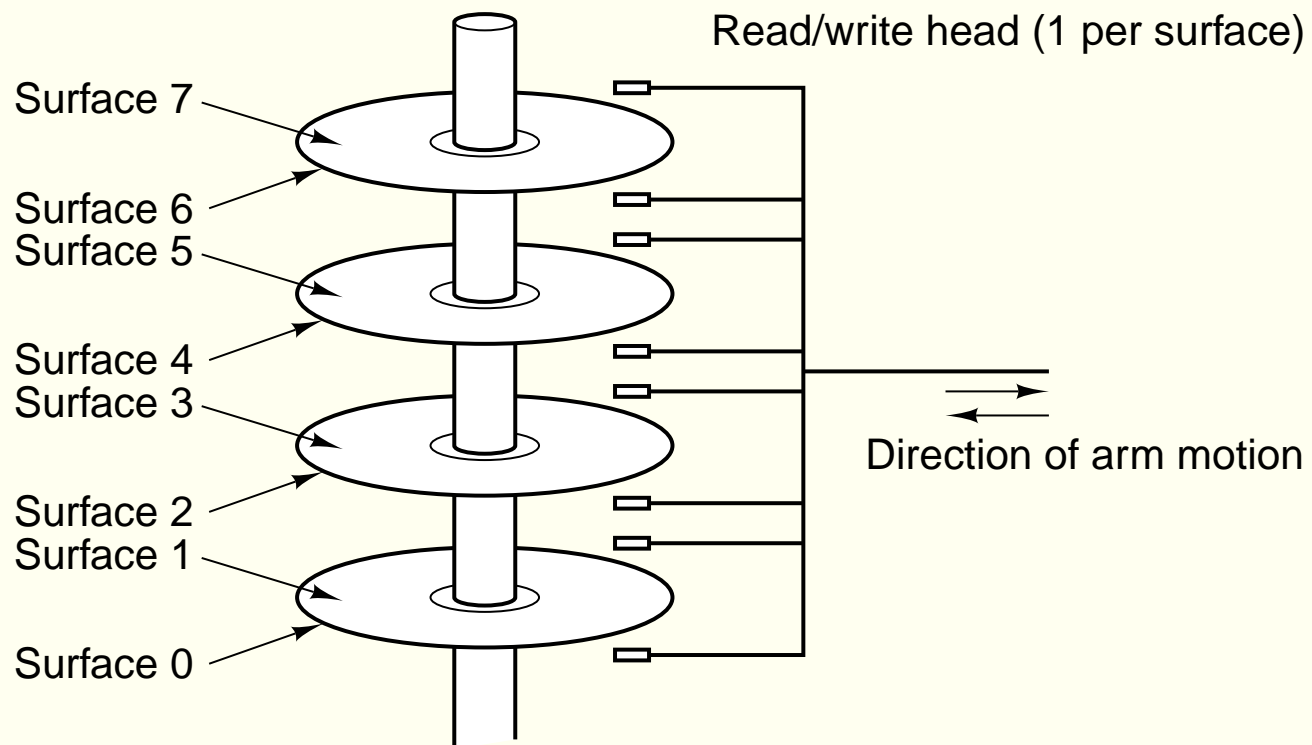
# Hierarquia de Memória

## Memória Principal

- Random Access Memory (RAM)
- Compromisso entre preço e desempenho
- Armazenamento volátil

# Hierarquia de Memória

## Disco



# Hierarquia de Memória

## Fitas magnéticas

- Utilizadas para cópias de segurança (backups)
- Armazenamento de grandes quantidades de dados
- Acesso seqüencial

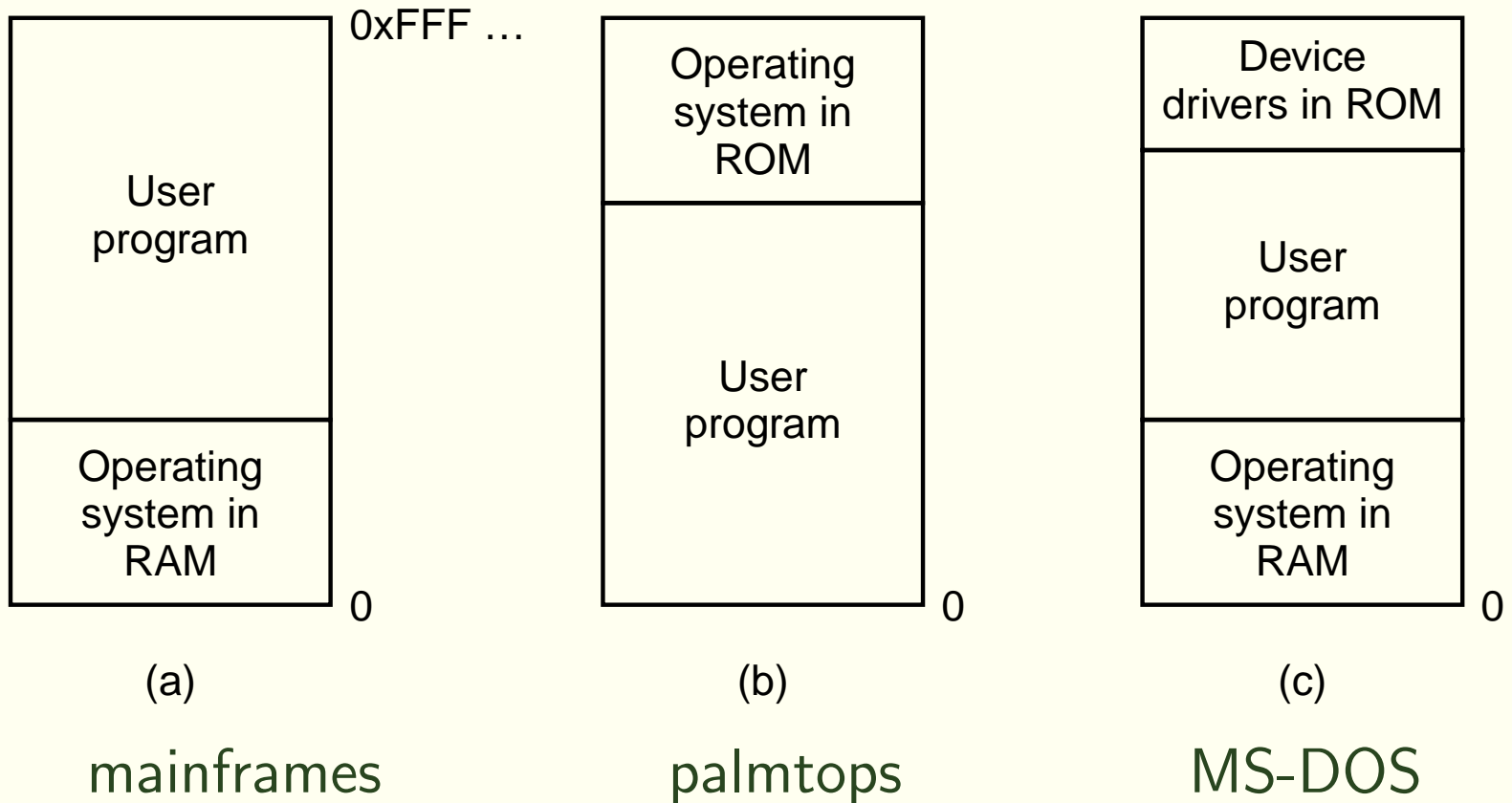


# Hierarquia de Memória

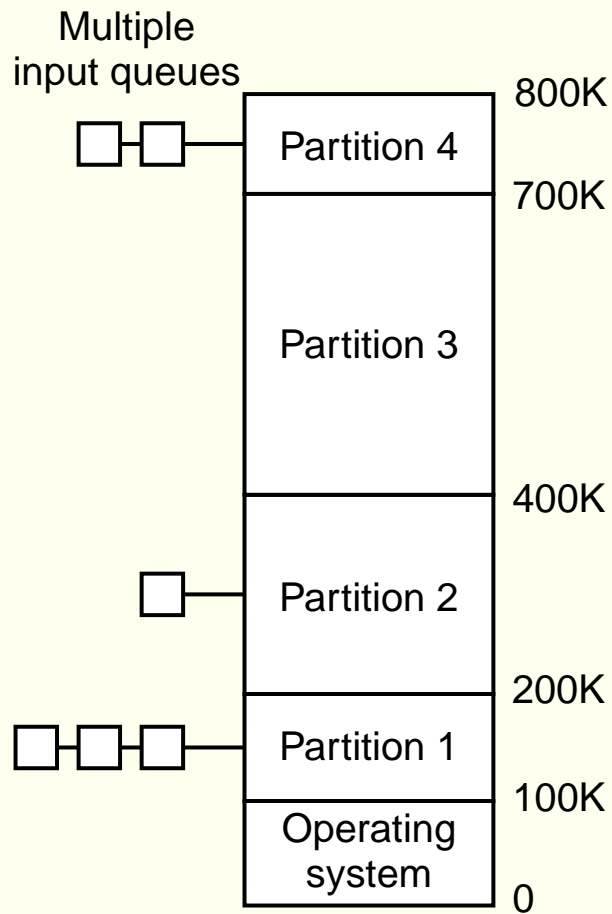
## Outros tipos de memória

- ROM (Read Only Memory)
  - rápida e barata
  - bootstrap loader está gravado em ROM
- EEPROM (Electrically Erasable ROM)
  - podem ser apagadas (erros podem ser corrigidos)
- CMOS
  - dependem de uma bateria
  - armazenam relógio e configurações

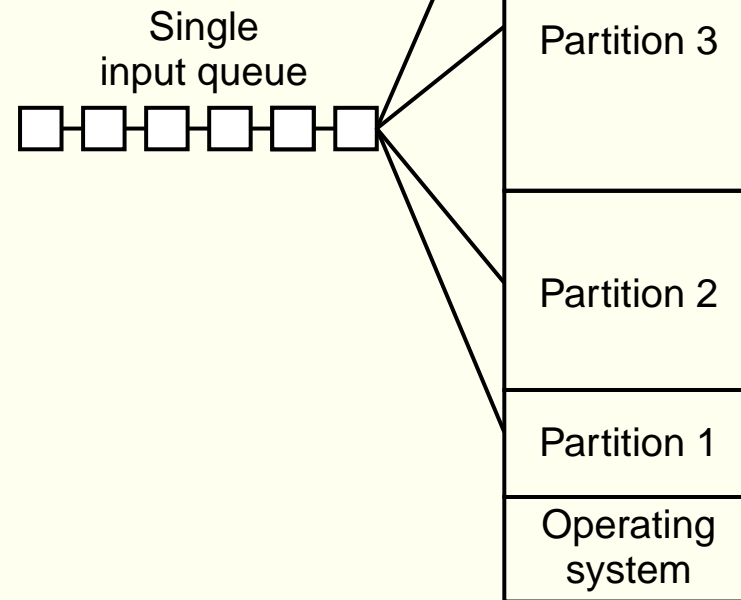
# Monoprogramação



# Multiprogramação e partições fixas



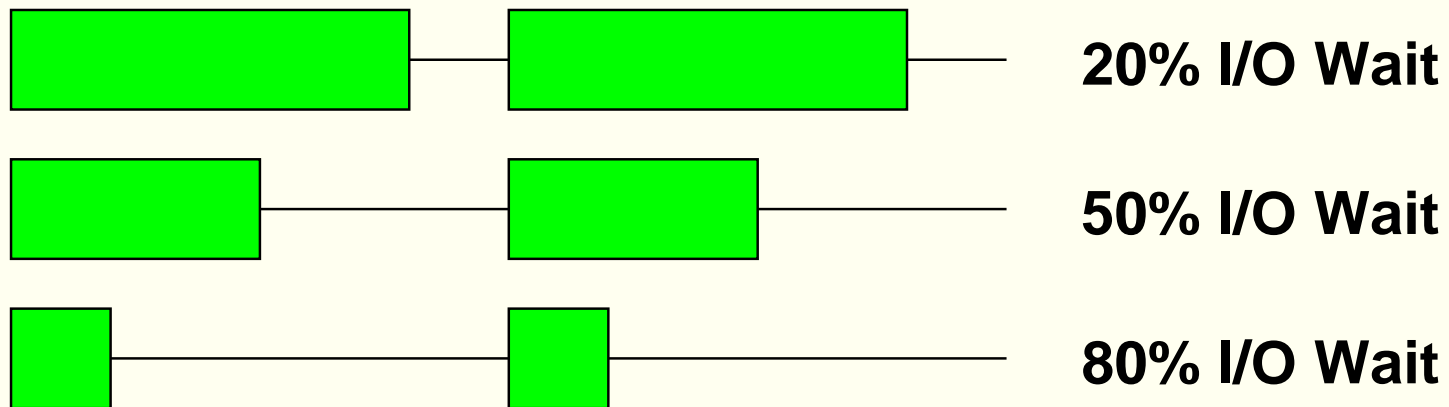
(a)



(b)

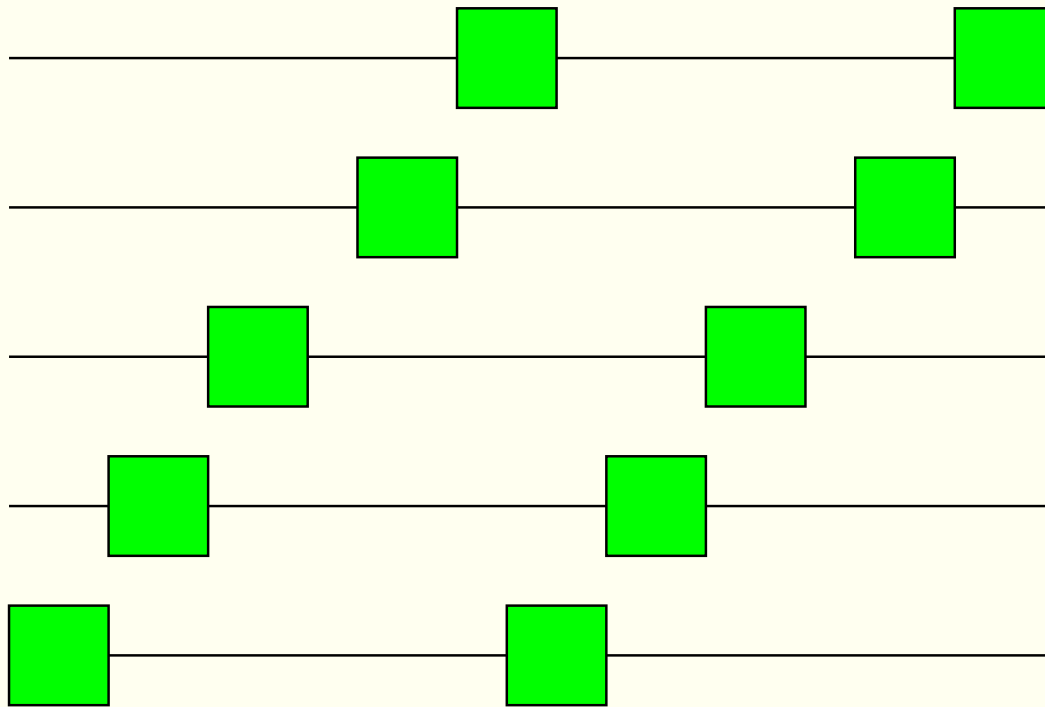
# Modelagem de multiprogramação

- Quantos processos colocar na memória?
- Estimativa de uso da CPU



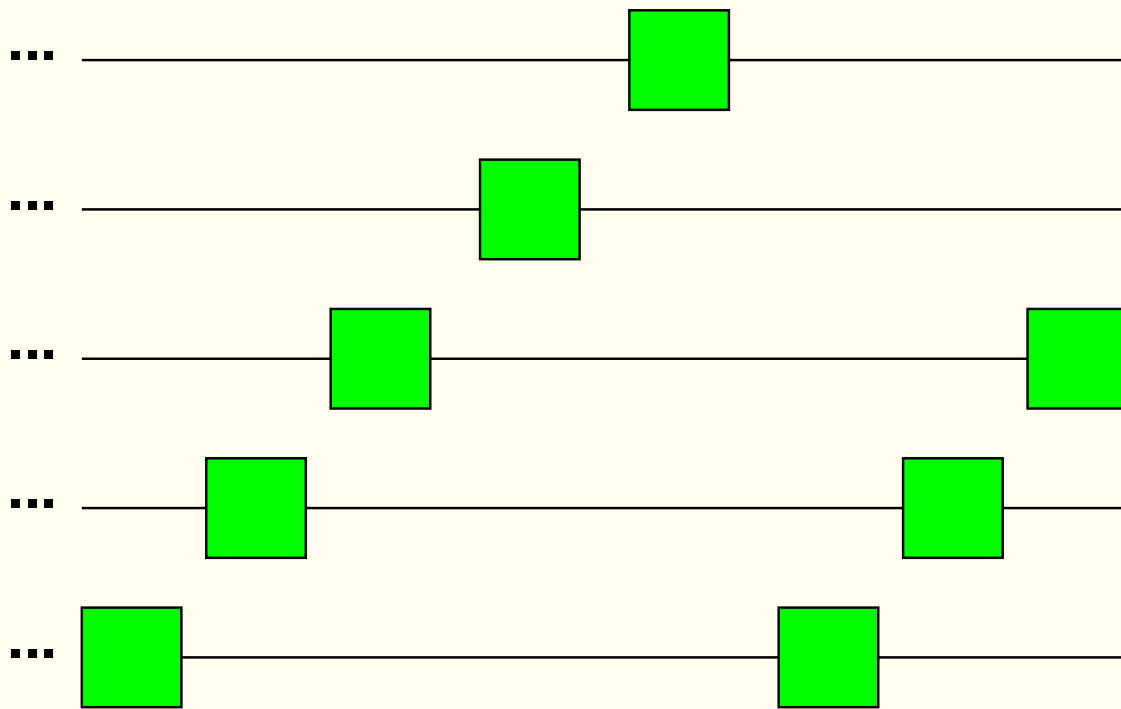
# Modelagem de multiprogramação

- Quantos processos colocar na memória?
- Aposta otimista: 5 processos 80% I/O wait

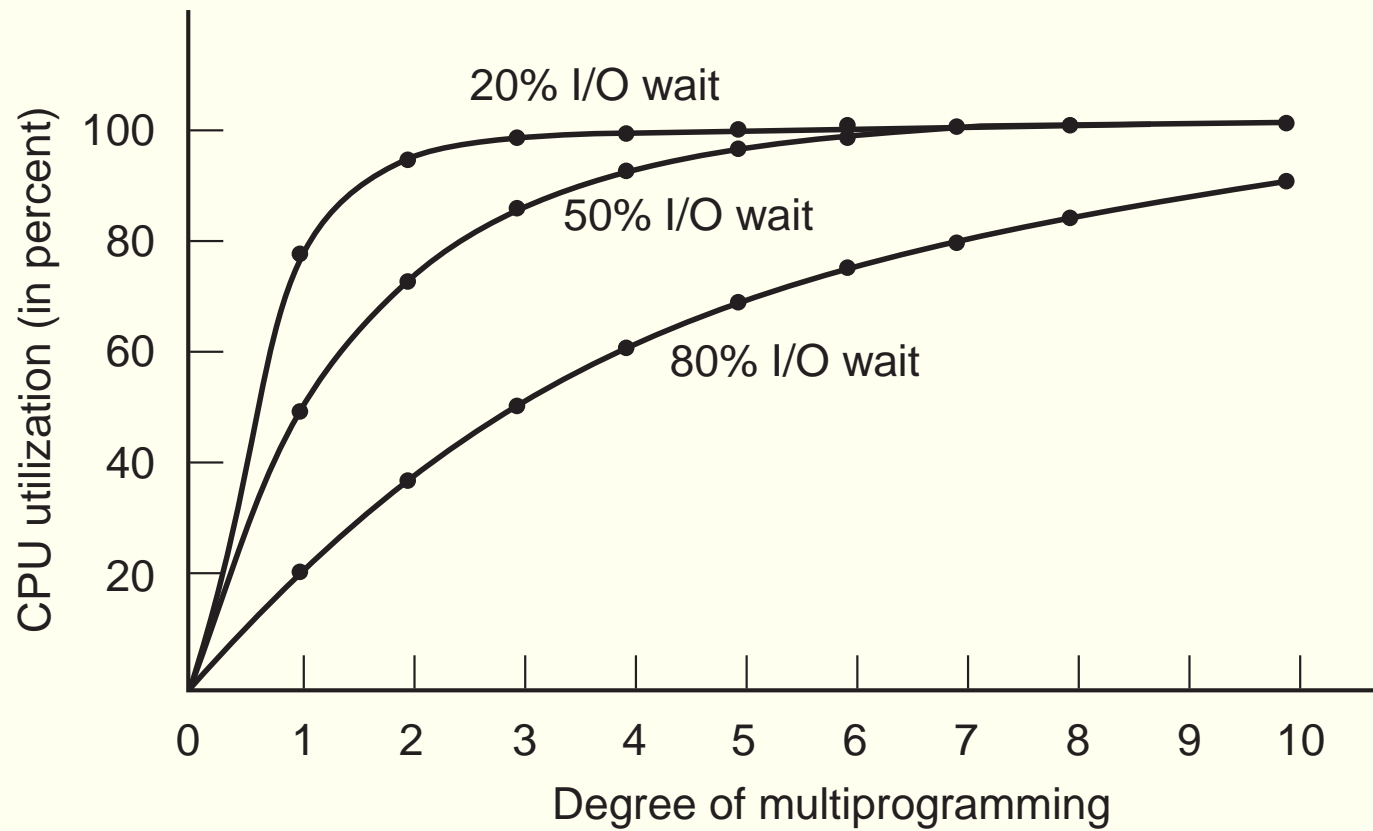


# Modelagem de multiprogramação

- Visão probabilística
- Taxa de utilização da CPU =  $1 - (0.8)^n$



# Modelagem de multiprogramação



$$\text{Utilização da CPU} = 1 - p^n$$

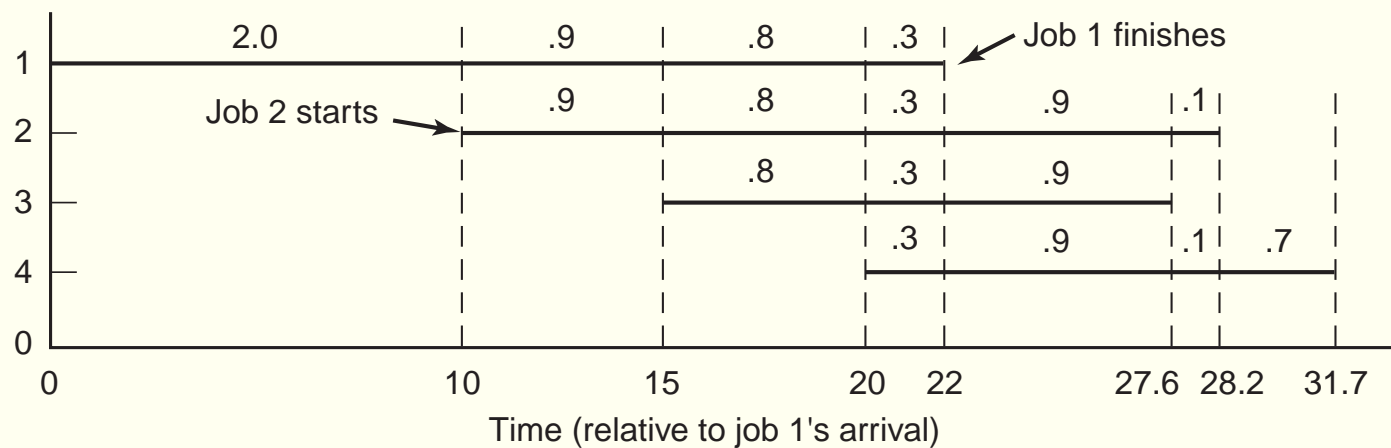
# Análise de Desempenho

Job	Arrival time	CPU minutes needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

(a)

	# Processes			
	1	2	3	4
CPU idle	.80	.64	.51	.41
CPU busy	.20	.36	.49	.59
CPU/process	.20	.18	.16	.15

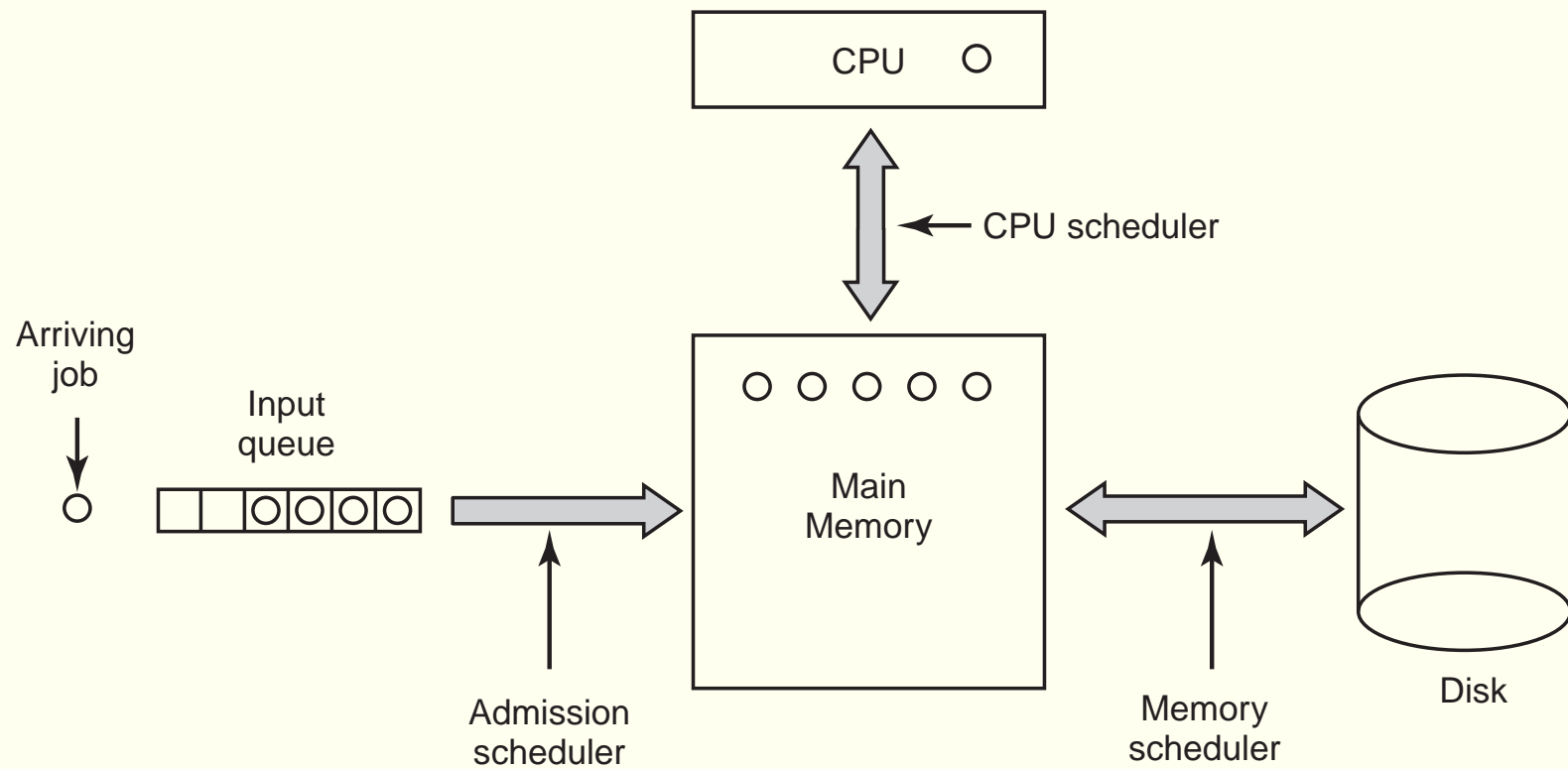
(b)



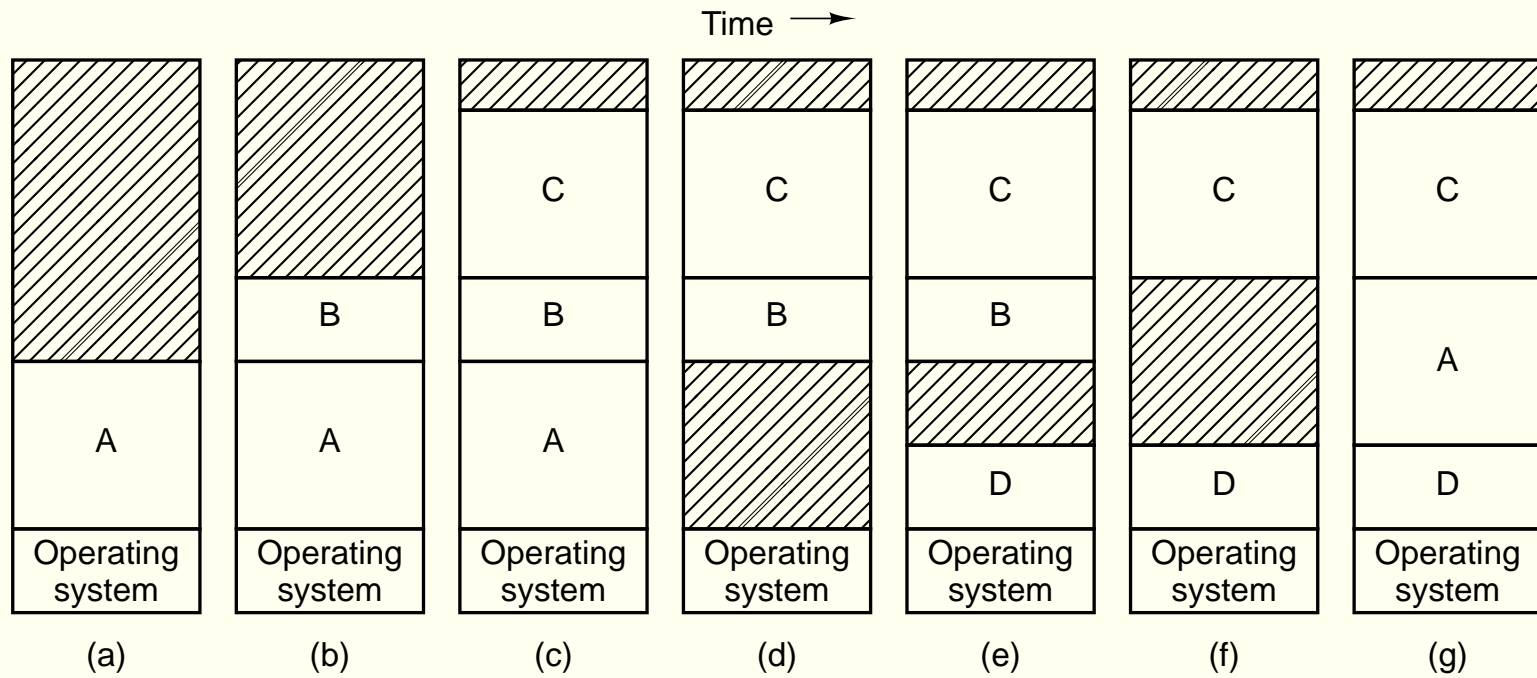
(c)



# Swapping



# Swapping



# Relocação e Proteção

- Relocação: um programa deve poder rodar em endereços físicos distintos.
- Proteção: um programa não pode fazer acesso à área de memória reservada a outro programa.

## Relocação durante a carga

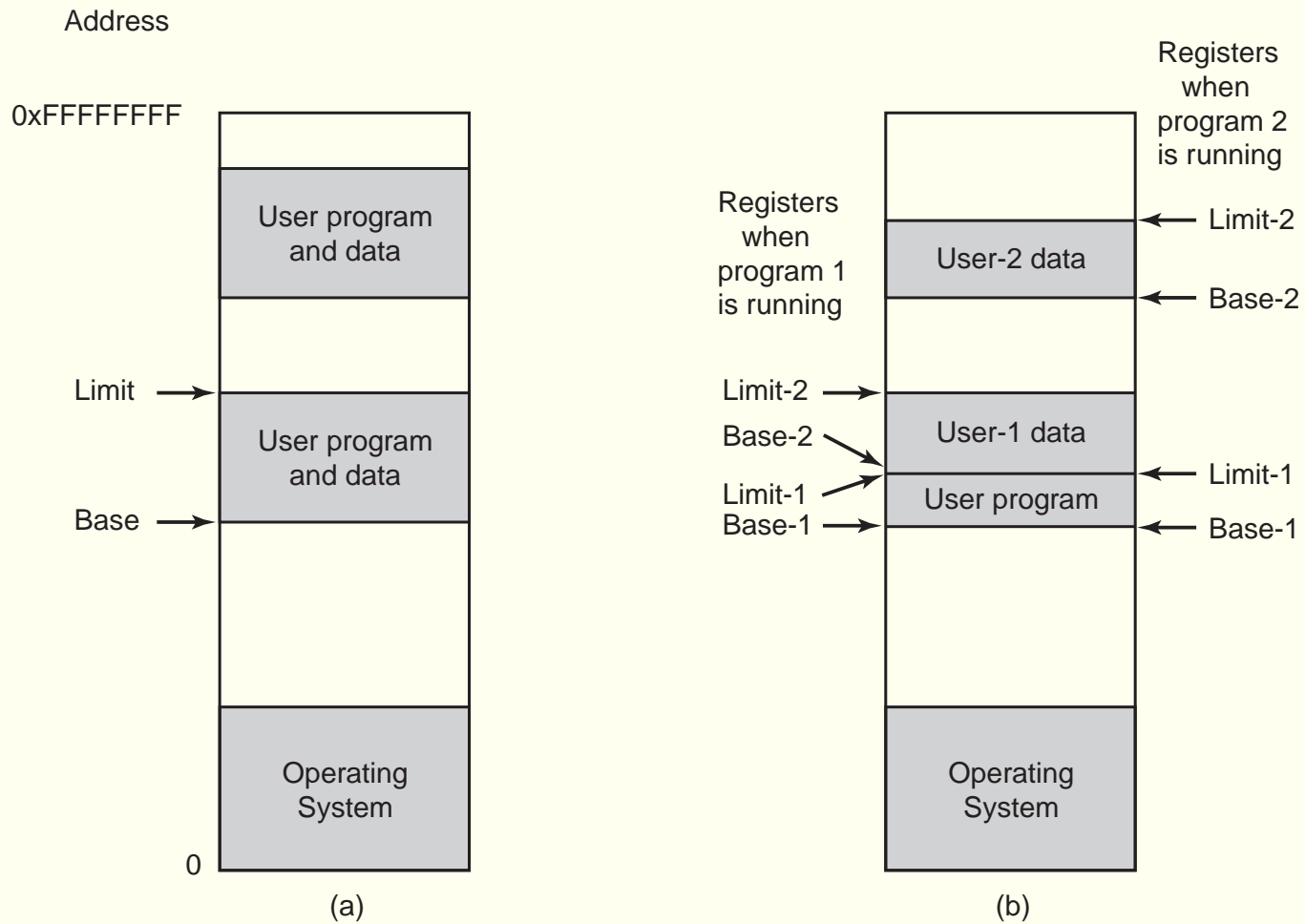
- Todos os endereços precisam ser identificados e alterados
- Não resolve o problema da proteção

## Bits de proteção

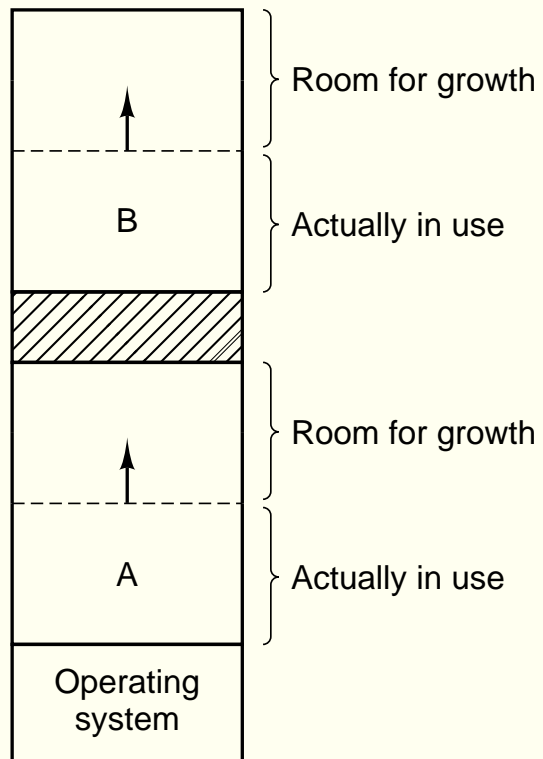
1010	
1010	
1010	
0011	
0011	

O PSW de cada processo deve conter os bits de proteção

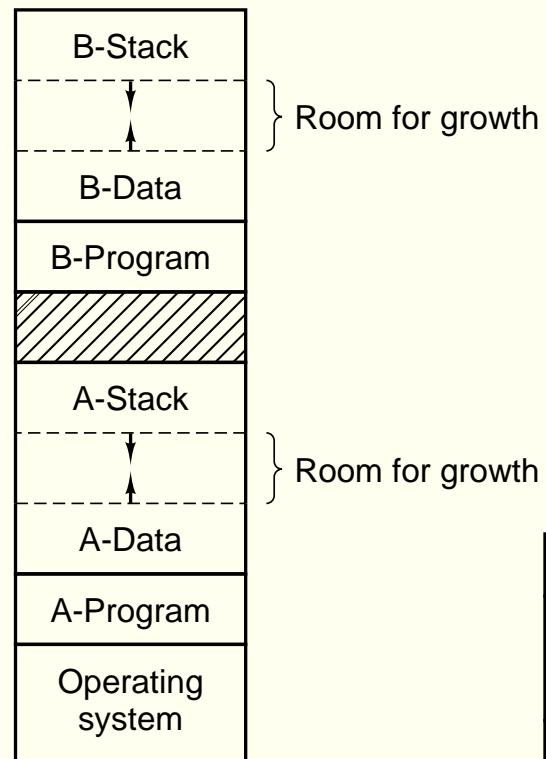
# Registadores base e limite



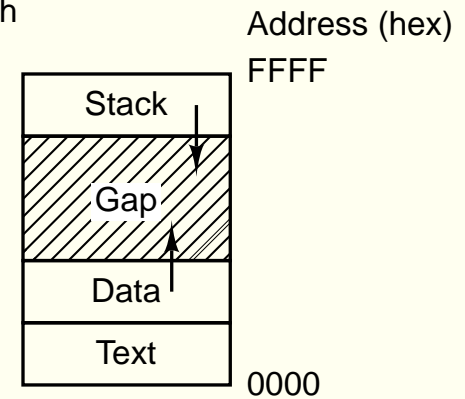
# Espaço para crescimento



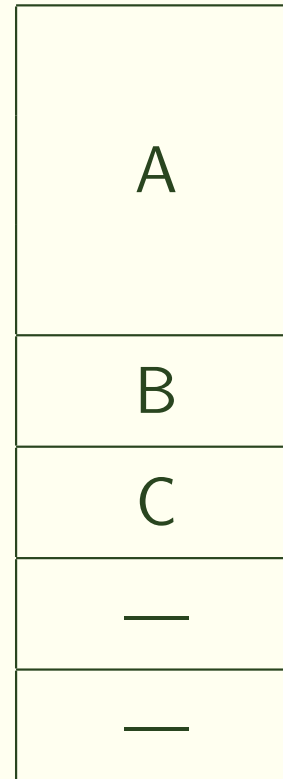
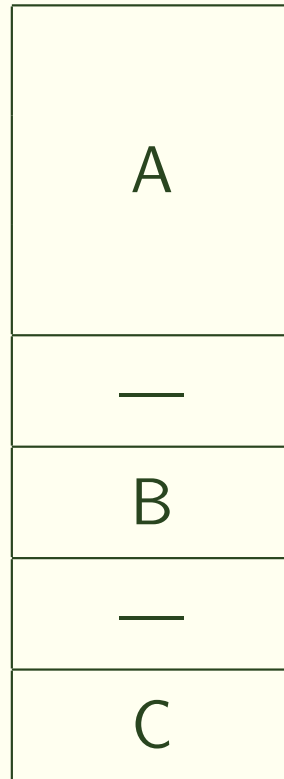
(a)



(b)

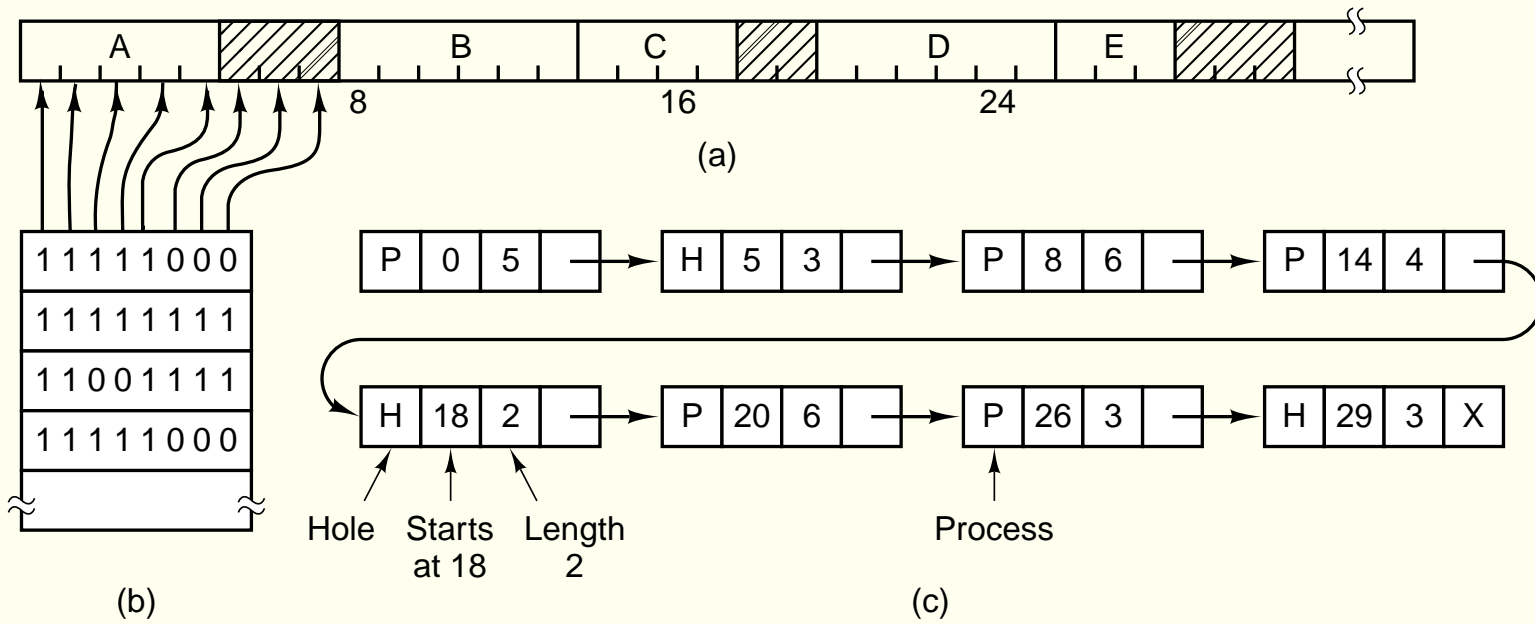


# Compactação de memória

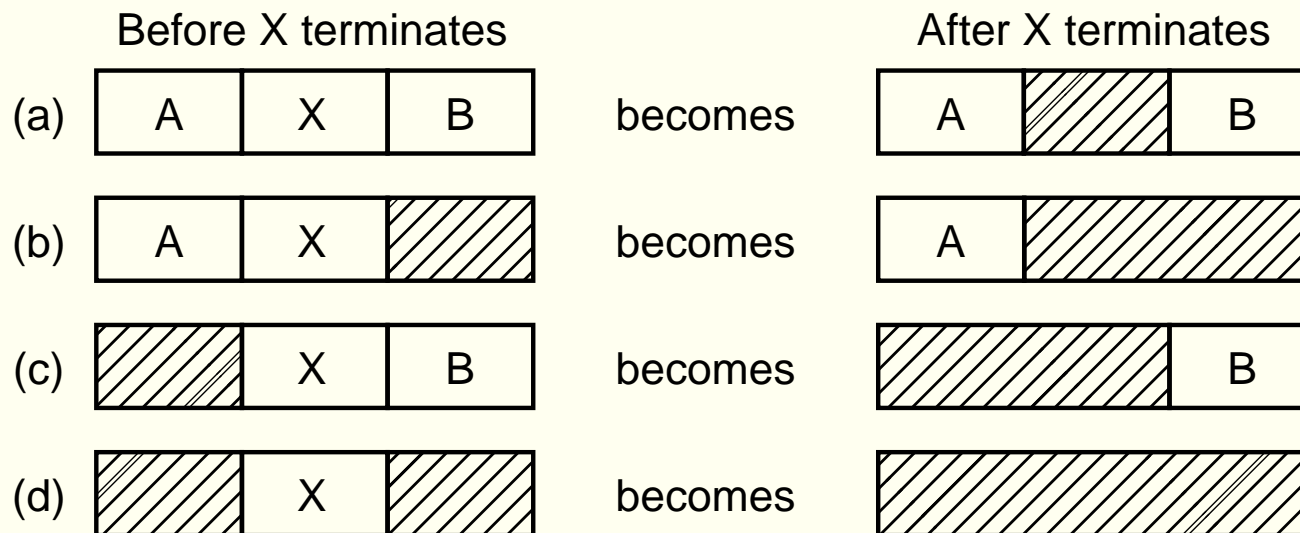




# Bitmaps e lista de livres



# Atualização da lista



# Algoritmos para alocação de memória

- *First fit*
- *Next fit*
- *Best fit*
- *Worst fit*

# Problema relacionado

## Malloc, free e realloc

```
void *malloc(size_t size);  
void free(void *ptr);  
void *realloc(void *ptr, size_t size);
```

A alocação de registros dentro de um processo tem problemas semelhantes:

- administração de uma lista de livres
- fragmentação

# Programas muito grandes

O que fazer se um programa for muito grande para caber na memória?

# Overlays

```
dados d1, d2, d3, d4, d5;
```

```
f1();      g1();      h1();
```

```
f2();      g2();      h2();
```

```
f3();      g3();      h3();
```

```
main() {
```

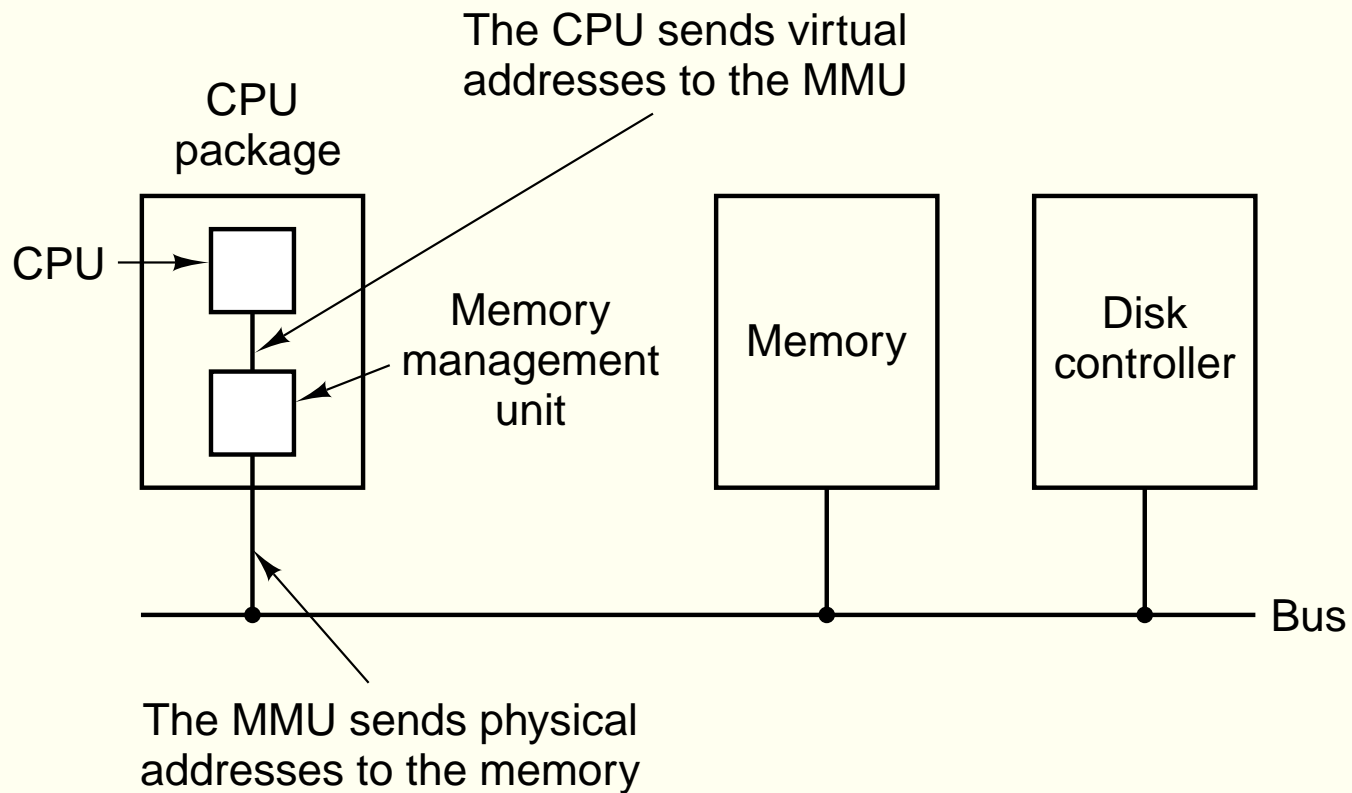
```
    fase_1(); /* funcoes f e dados d1, d2, d3 */
```

```
    fase_2(); /* funcoes g e dados d1, d2, d4 */
```

```
    fase_3(); /* funcoes h e dados d1, d2, d5 */
```

```
}
```

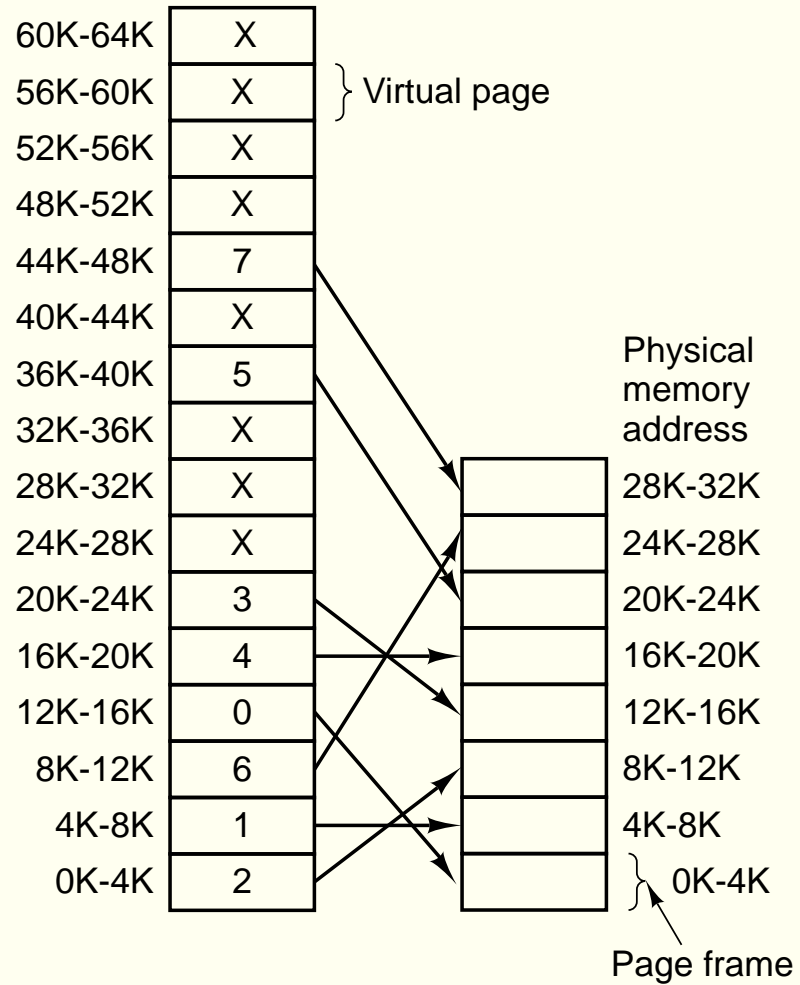
# Memória Virtual



Memory Management Unit (MMU)

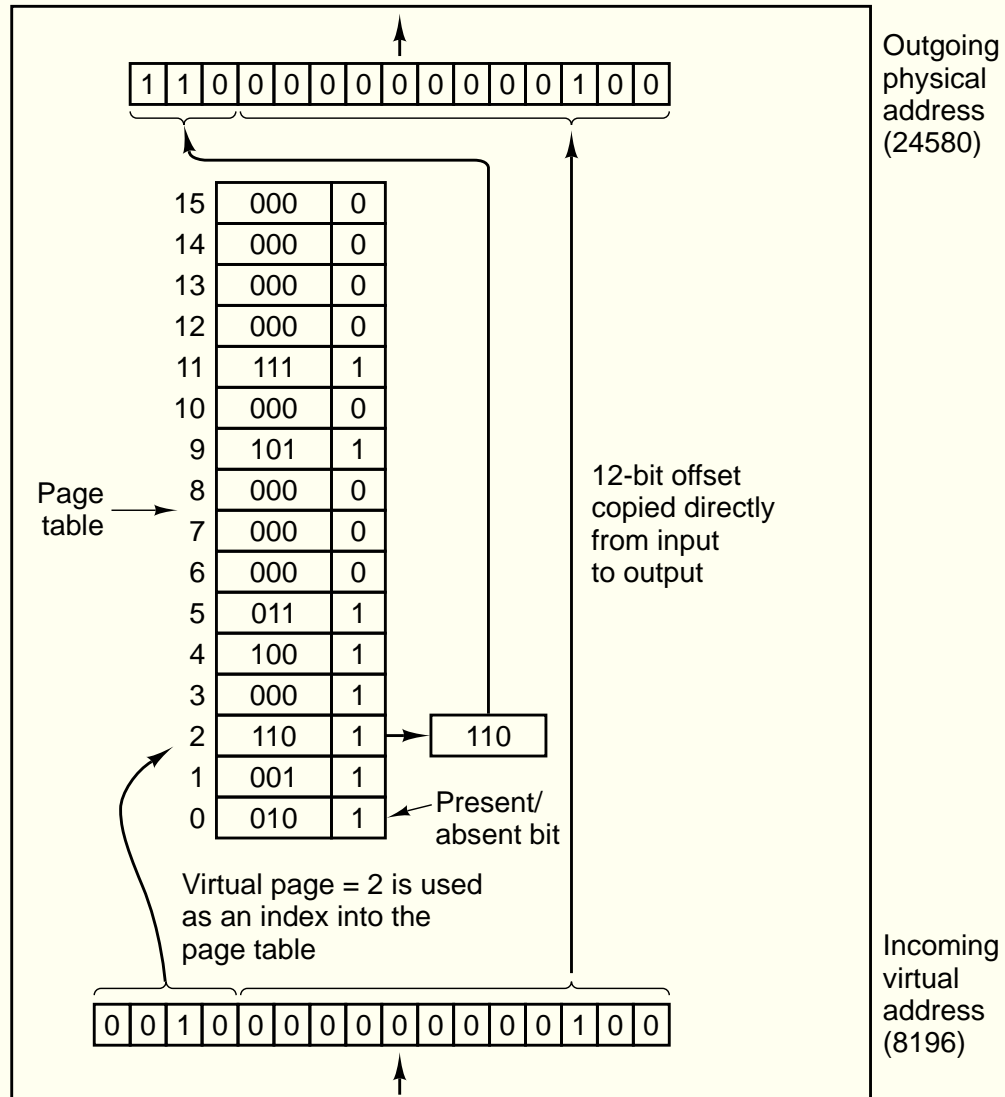
# Paginação

Virtual  
address  
space



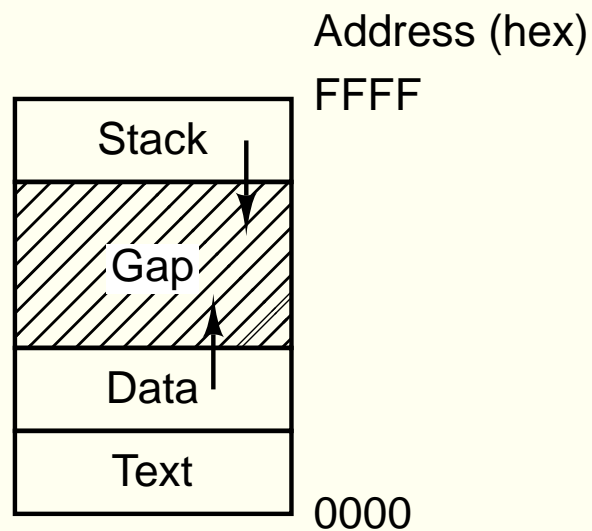


# Mapeamento dos endereços



# Espaço de endereçamento

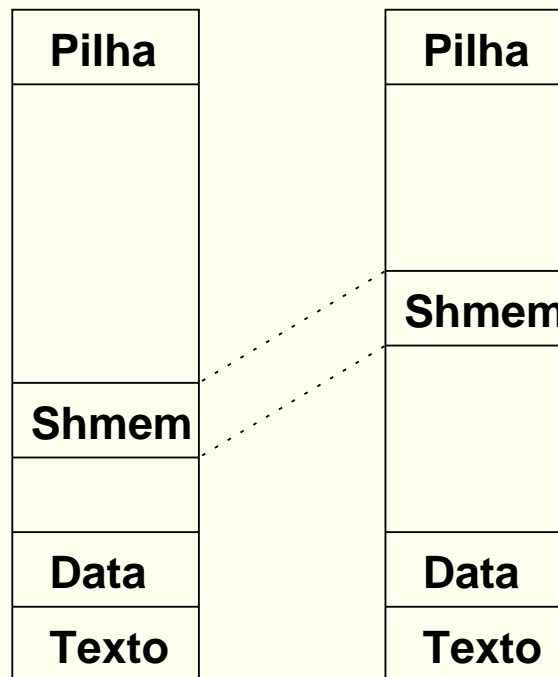
- Apenas as páginas ocupadas precisam ser mapeadas



# Memória compartilhada

Processo A

Processo B



## Memória compartilhada

```
int shmget(key_t key, size_t size, int shmflg);  
void *shmat(int shmid,  
            const void *shmaddr, int shmflg);
```

- Veja os exemplos: sh1.c sh2.c sh\_fork.c