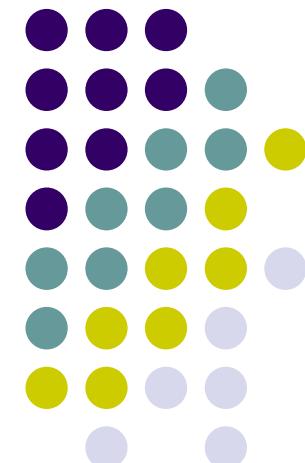
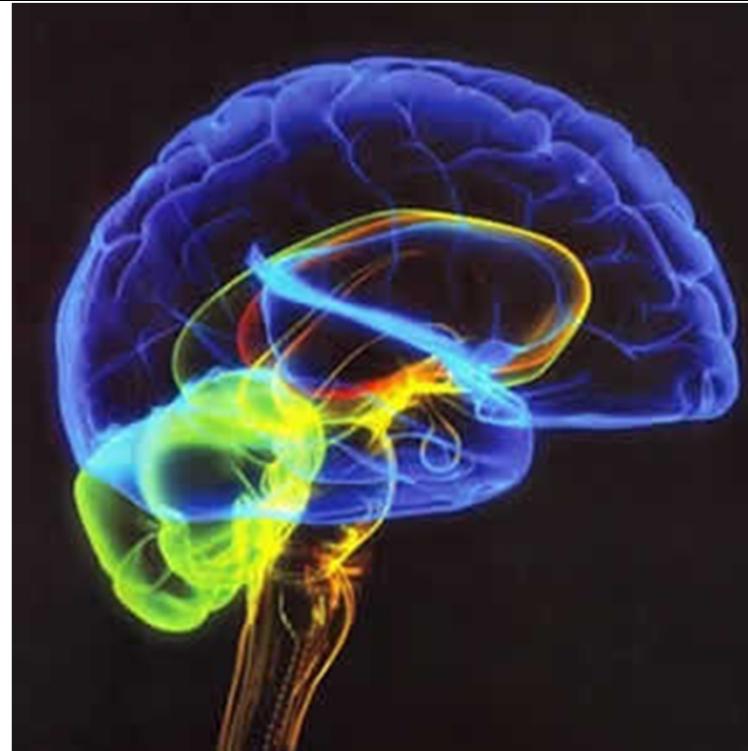


# Gerência de Memória





# Gerência de Memória

## Roteiro:

### 1 - Introdução

Realocação e Proteção

### 2 - Gerência básica de Memória

Registradores base e limite

MMU e troca de contexto

Multiprogramação c/ partições fixas

### 3 - Swapping

### 4 - Paginação (Memória Virtual)

Algoritmos para troca de páginas

Questões de Projeto e Implementação

Tabelas de páginas invertidas

### 5 - Segmentação

# Memória Principal



Tipos básicos:

DRAM: Dynamic random access memory

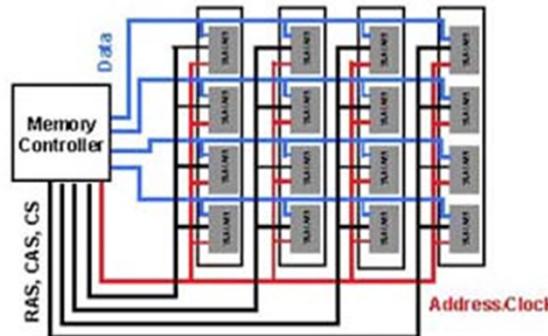
SRAM: Static random access memory chips (não-voláteis)

EPROM: Erasable programmable read-only memory

PROM: apenas escritos 1x

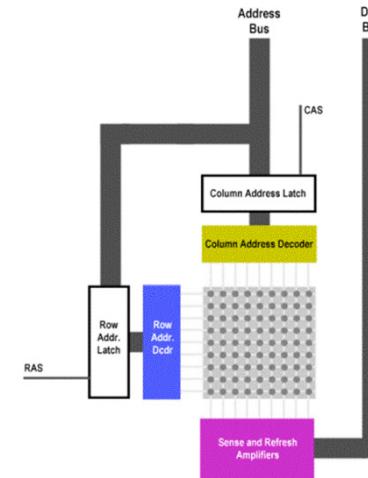
FLASH memory chips offer extremely fast access times, low power consumption, do not need a constant power supply

SDRAM Architecture

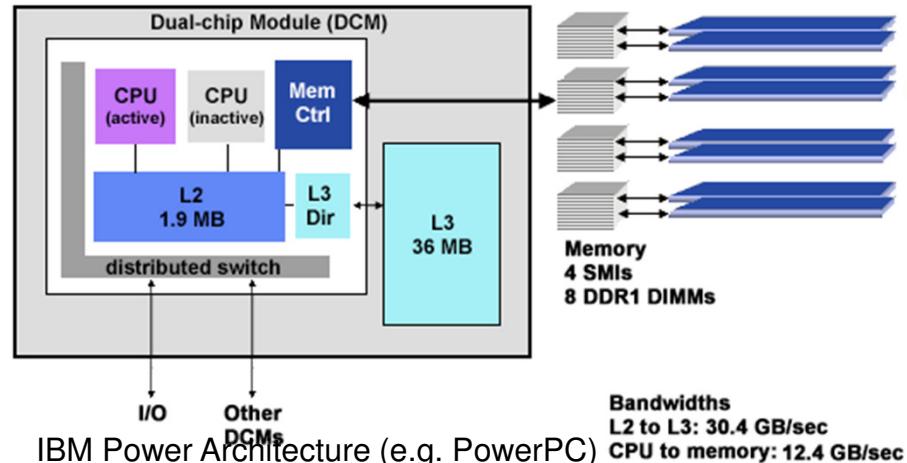


- Variable length wires, different routes
- 66-133Mhz

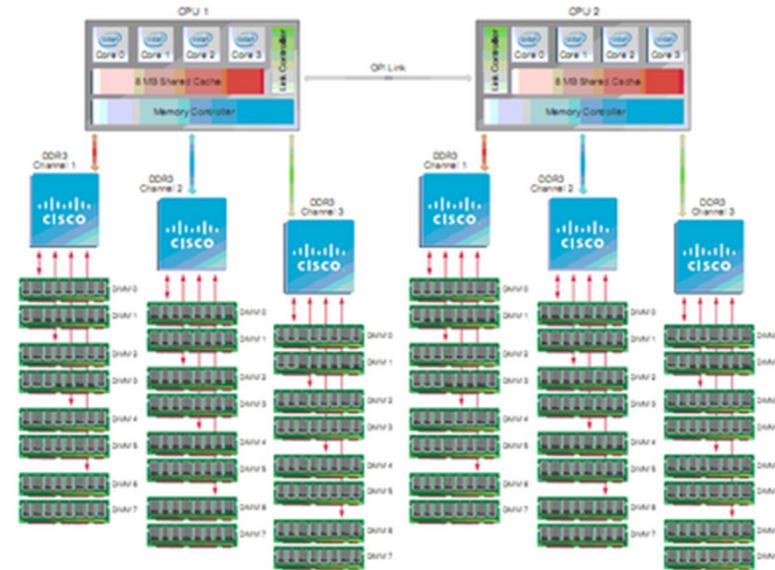
DRAM Architecture



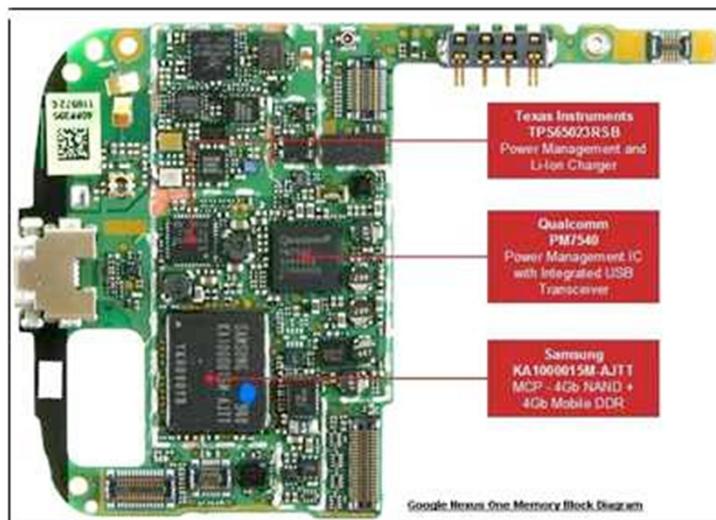
# Memória Principal



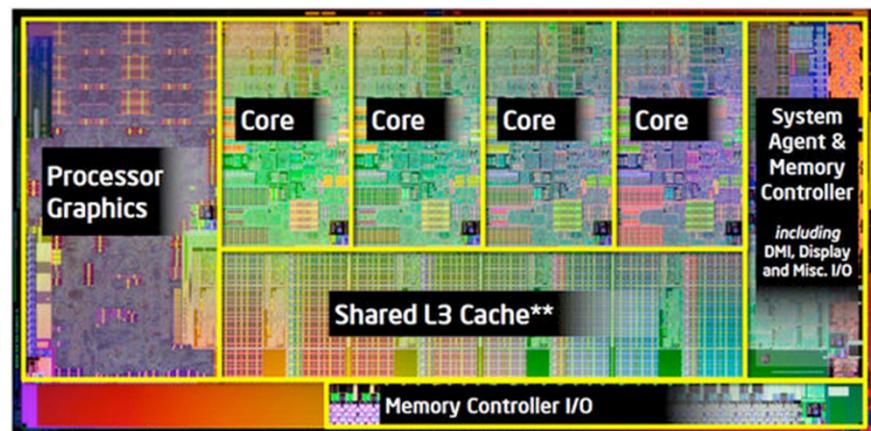
IBM Power Architecture (e.g. PowerPC)



Arquitetura Multi-core com vários controladores



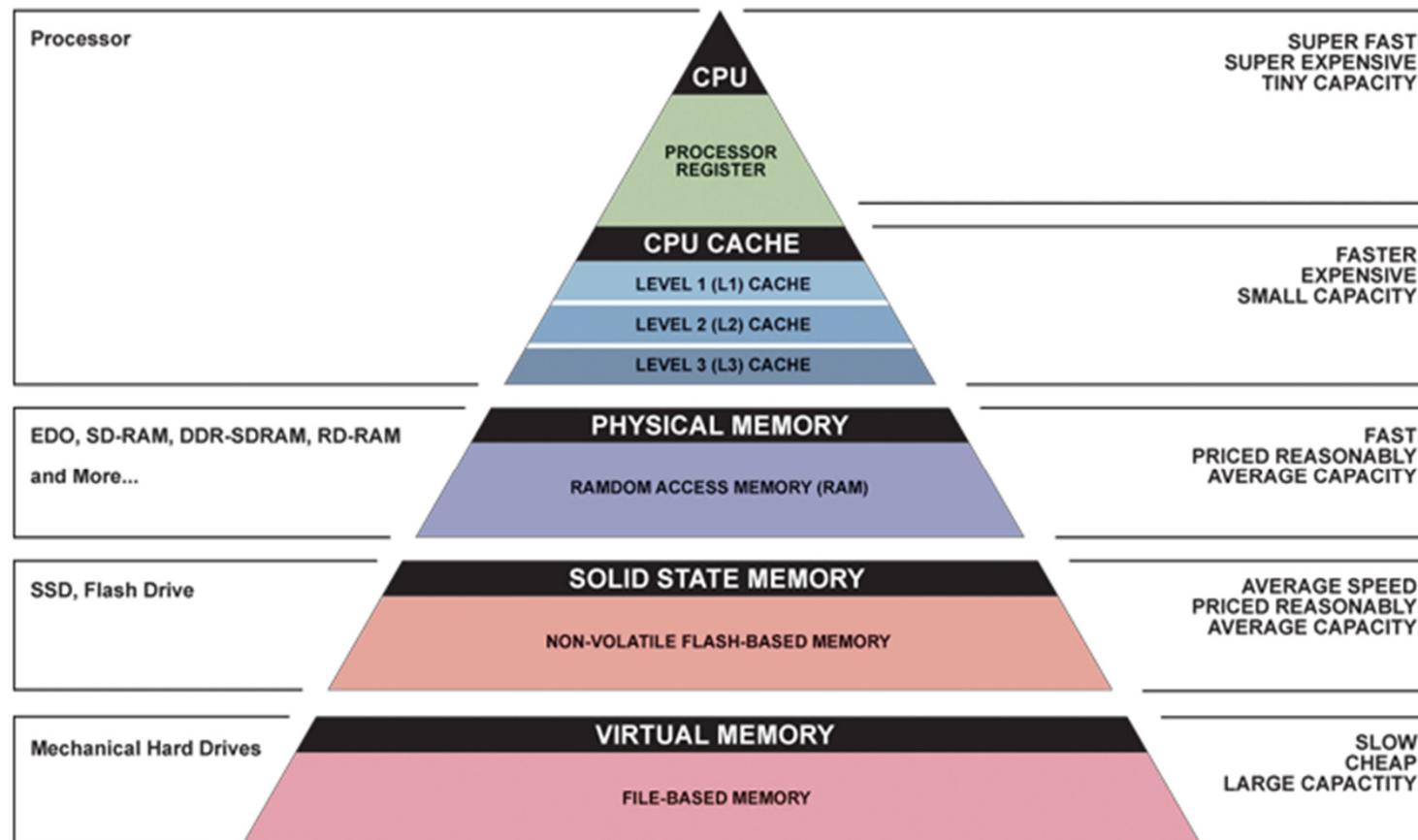
Arquitetura do NexusOne (Google) com memória volátil e não volátil.





# Hierarquia de Memória

O núcleo precisa gerenciar a transferência *transparente* de dados de um nível para outro



▲ Simplified Computer Memory Hierarchy  
Illustration: Ryan J. Leng



# Gerente de Memória: Tarefas básicas

Gerente de Memória opera a hierarquia de memória:

- Quais partes da memória estão em uso, e quais não?
- Alocar memória para novos processos
- Desalocar memória quando processo termina
- Garantir proteção/isolamento entre processos
- Setar os parâmetros para a relocação de endereços de memória
- Quando memória está cheia (sempre!), fazer o swapping transparente entre memória principal e disco

Essas tarefas dependem:

- Do tipo de sistema (p.ex. se suporta multiprogramação)
- Do tamanho da memória e suas características de acesso (latência, acesso concorrente, etc.)
- Das características do Controlador da memória (se possui MMU)
- Do tamanho e quantidade de caches disponíveis

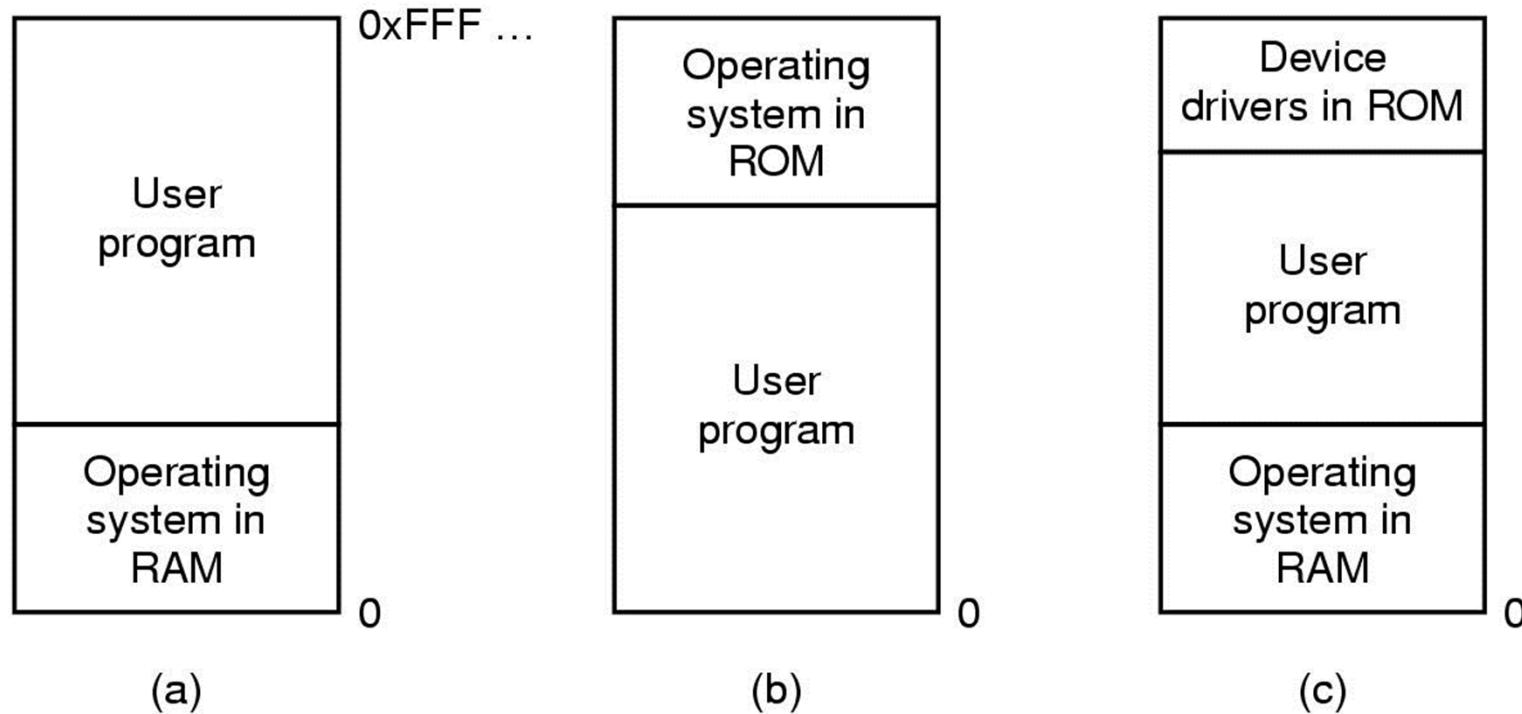
# GM para Monoprogramação

(sem swapping ou paginação)



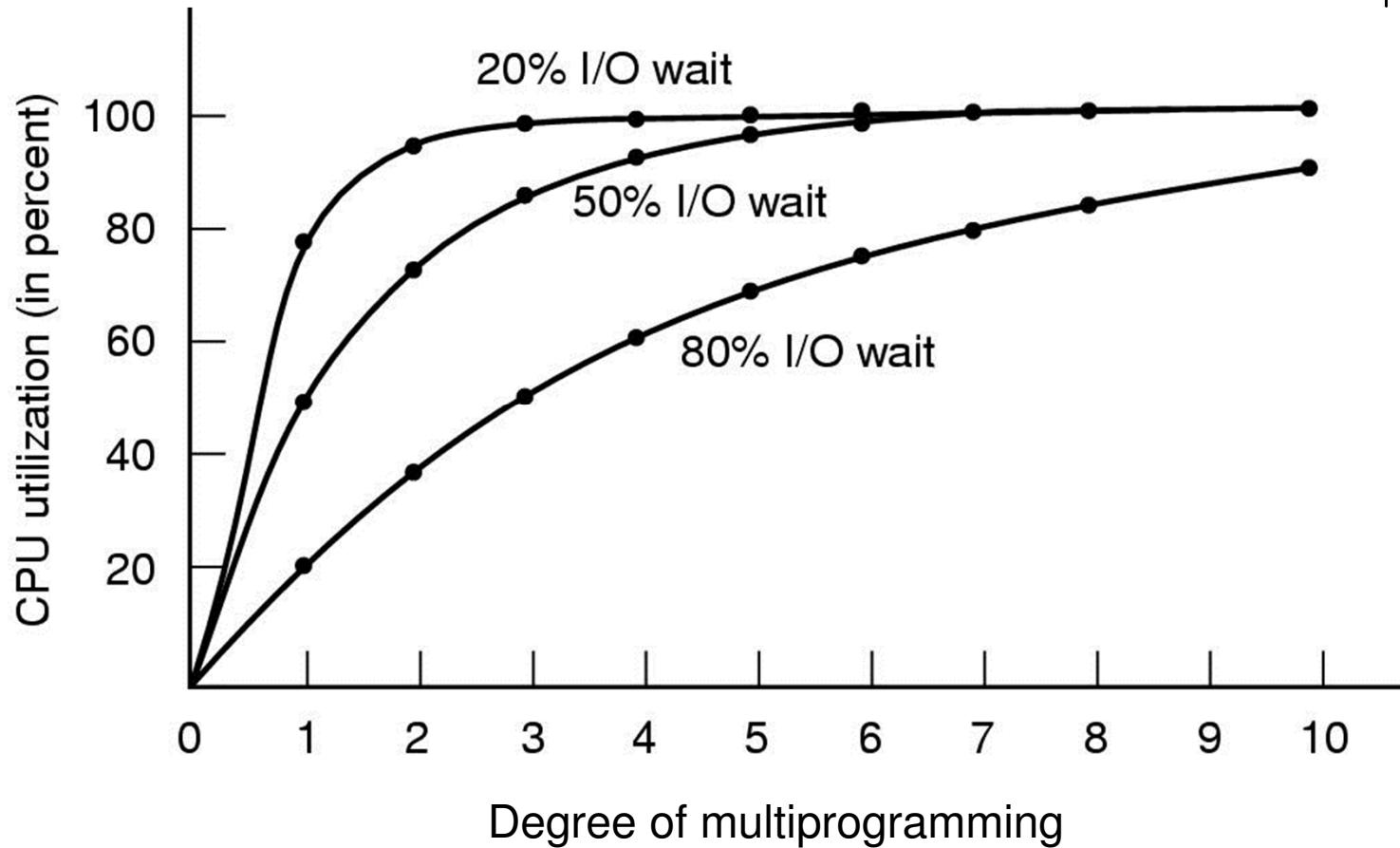
Para S.Os com único usuário e dispositivos simples (embarcados). Execução de 1 programa por vez.

- Ciclo básico: Comando usuário → carregamento → execução
- Baixa utilização de CPU



Formas (ultrapassadas) de organizar a memória

# Efeito da Multiprogramação



Utilização da CPU como função do grau de multiprogramação (= número de processos na memória)



# ReallocAÇÃO e ProteÇÃO

São dois problemas introduzidos pela Multiprogramação:

- ReallocAÇÃO: não se sabe de antemão em qual região de memória o processo vai ser executado
  - Endereço de variáveis e do código não podem ser absolutos
- ProteÇÃO: evitar que um processo acesse uma região usada por outro processo

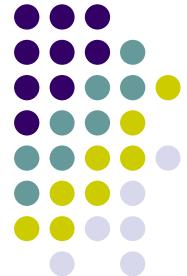
Solução 1: modificar endereços quando processo é carregado (o ligador/carregador precisa ter um bit map sobre quais endereços do programa precisam ser atualizados)

- Como endereços são absolutos, seria possível acessar qualquer endereço de memória, e um programa pode construir dinamicamente instruções

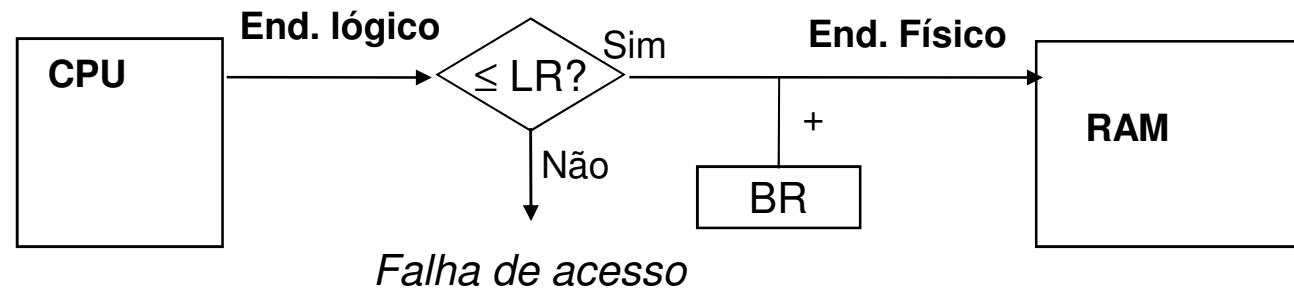
Solução melhor: Mapeamento para a memória física ocorre em tempo de execução e é relativa a dois registradores: base e limite

- Qualquer acesso além do limite é considerado erro e processo é abortado

# Registradores Base e Limite



Para cada processo, existe um par, base register (BR), e limit register (LR), que determinam a região de memória usada pelo processo



- A cada troca de contexto, o (BR/LR) precisa ser trocado



# Técnicas baseadas em realocação

Organização da memória:

- a) com partições fixas (BR e LR só podem assumir determinados valores)

Fila por tamanho de partição vs. Fila única

- b) com divisão/agrupamento de partições

Fila única

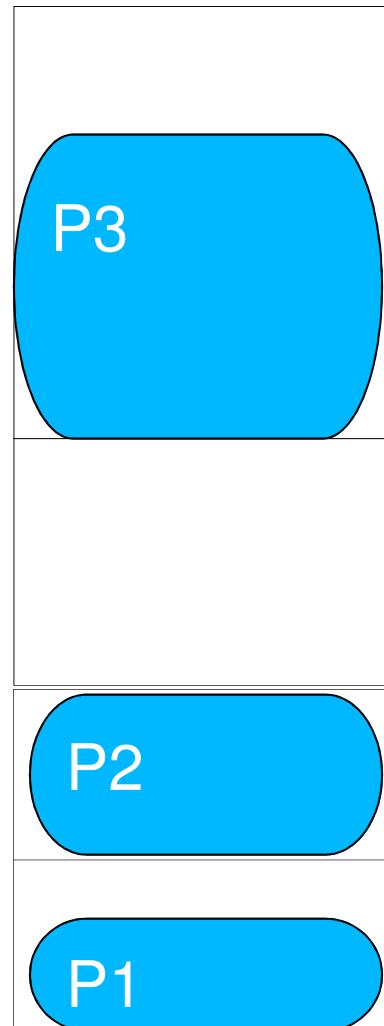
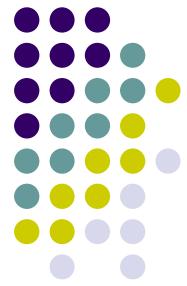
partições tem  $2^x$  palavras (p.ex.:  $2^{10} \leftrightarrow$  2 partições de  $2^9$ )

- c) Memória RAM sem partições (BR e LR assumem qualquer valor)

Outra questão (ortogonal):

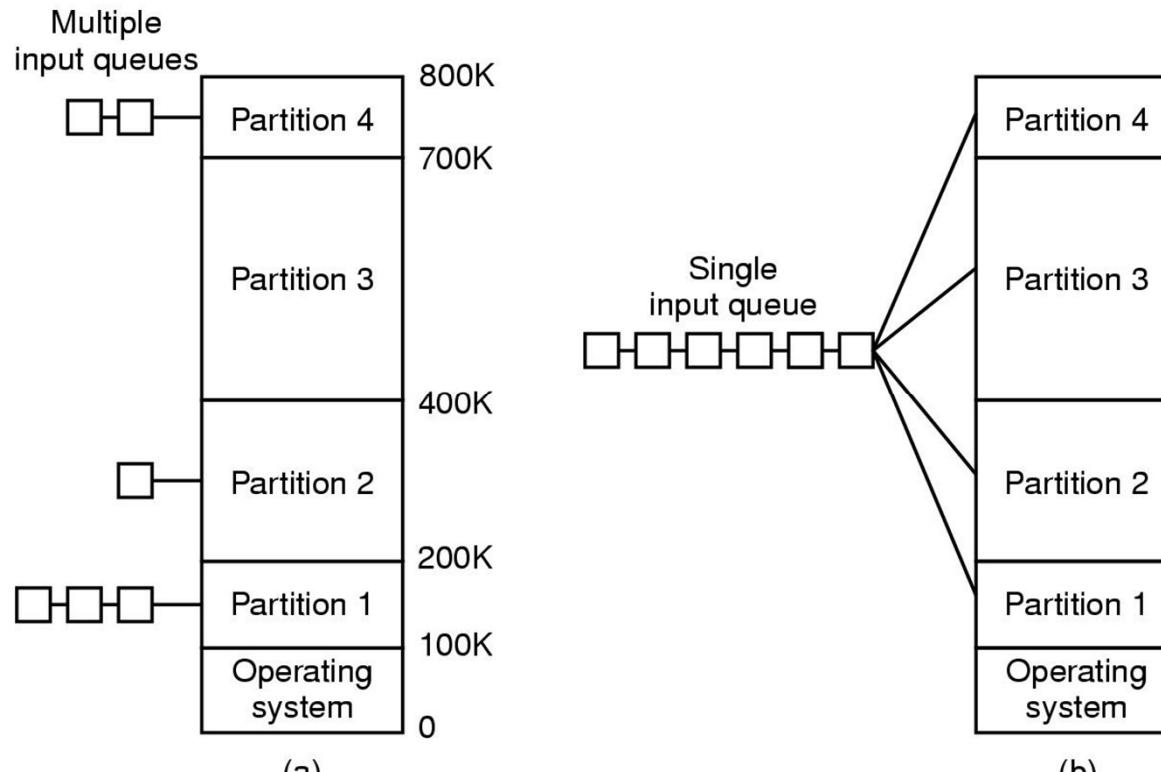
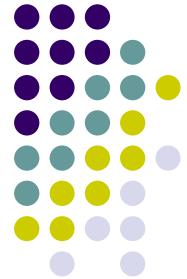
- a) Processos permanecem na memória durante toda a sua execução ? OU
- b) Processos podem ser retirados da memória, e posteriormente voltar à memória em outra região/partição (swapping)?

# Memória com partições fixas



Memória Principal com 4 partições

# Memória com partições fixas



Fila única para todas as partições: (exemplo IBM OS/360)

(+) simples de implementar

Filas de entrada separadas:

(-) processos menores não podem ser colocados em partições grandes disponíveis.

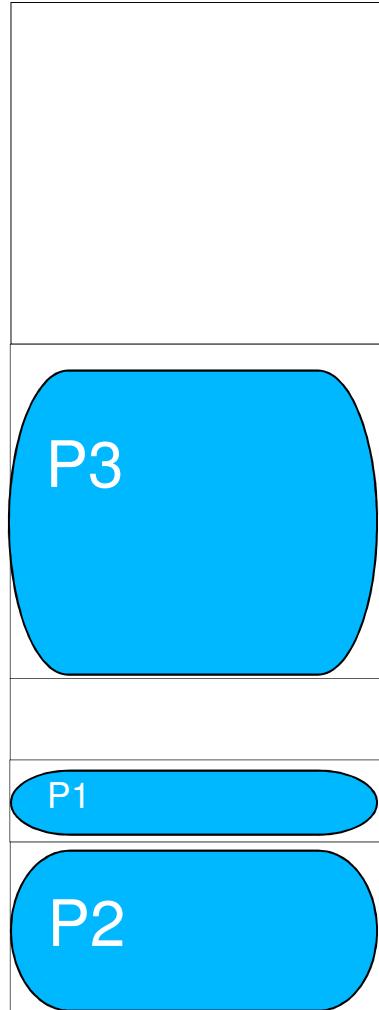
Escolha da partição com Fila única:

- Seleção “First-Fit”, “Best Fit”, “Worst Fit”

# Memória com partições fixas mas variáveis

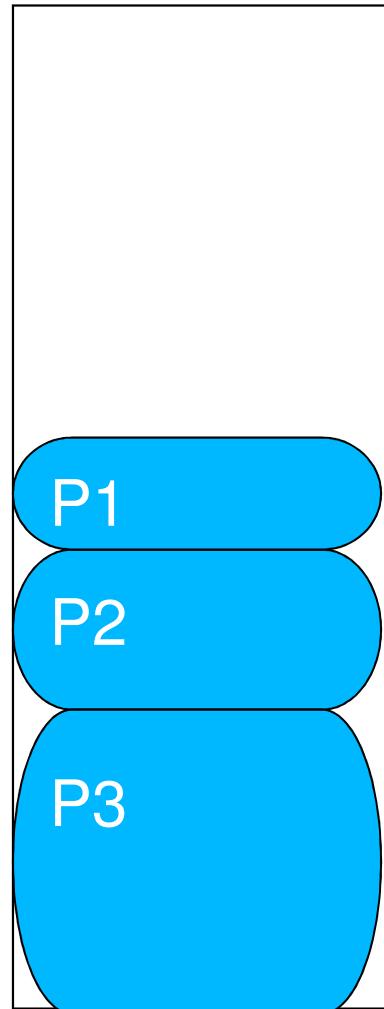
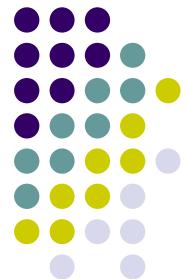


Dependendo da demanda de memória dos processos ingressantes partições são divididas ou agrupadas



Memória com Partições de tamanho variável  $2^i$  bytes

# Memória sem partições



Memória Principal



# Swapping

Em sistemas com compartilhamento de tempo (timesharing) memória principal pode não ser suficiente para todos os processos (e.g. muitos processos interativos de muitos usuários)

- Ideia básica: usar espaço em disco como extensão da memória RAM, e colocar lá os processos enquanto estão bloqueados, carregando-os de volta para a memória assim que são desbloqueados

Duas alternativas:

- Copiar a imagem inteira (swapping)
- Permitir que processo fique parcialmente em memória, e parcialmente em disco (paginação) → Memória Virtual



# Swapping (em mem. sem partição)

Quando um processo é bloqueado (espera por E/S) ele pode ser *swapped out*, e depois *swapped in* para memória principal.

Permite manter um número maior de processos ativos, aumentando a utilização da CPU

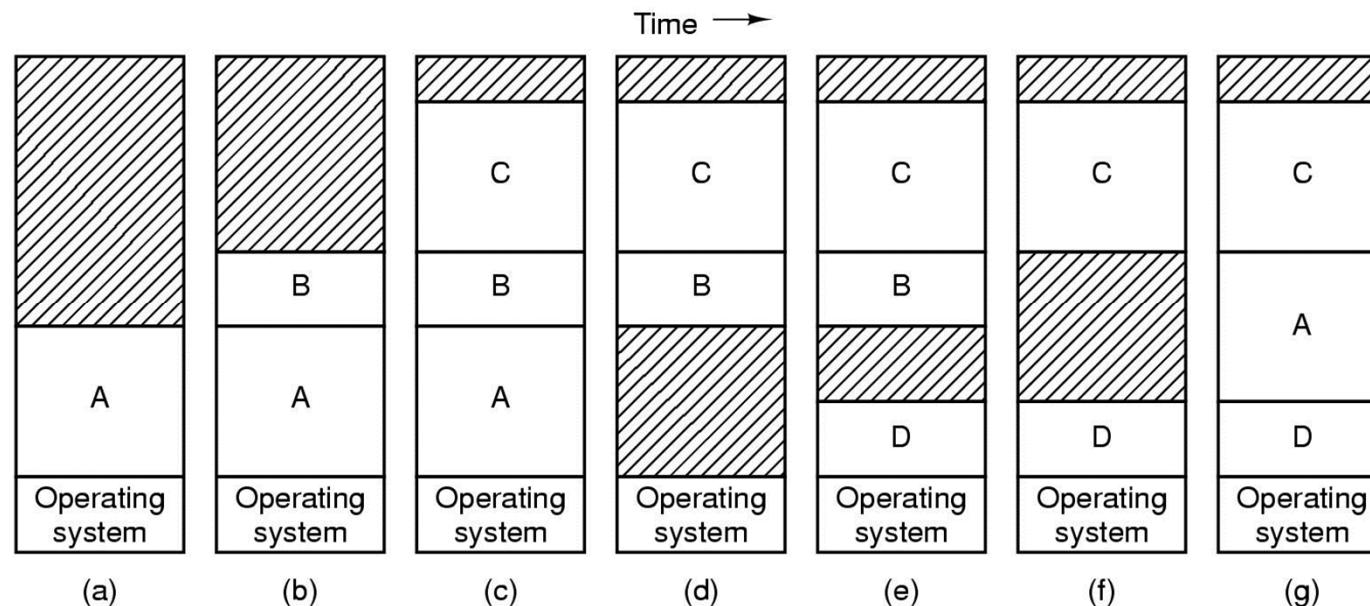


Fig.: Sequência de alocação de memória usando swapping para 4 processos.

Pode acarretar:

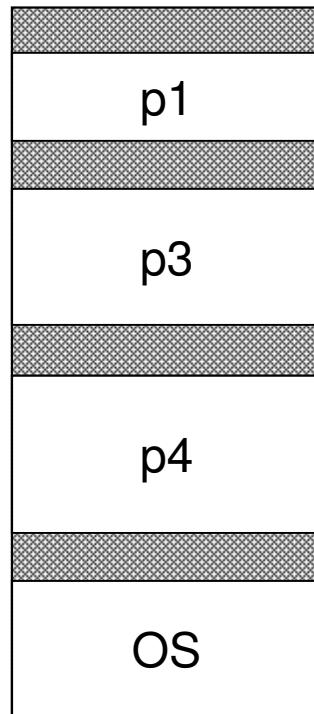
- Buracos de memória não utilizada de tamanho qualquer (fragmentação de memória)
- Um mesmo processo pode ocupar diferentes partições ao longo de sua execução



# Swapping

Principal problema do swapping com partições de tamanho variável:

- Manter a informação sobre espaços não utilizados (livres)
- Evitar uma *fragmentação externa* da memória (= muitos espaços pequenos não utilizados)



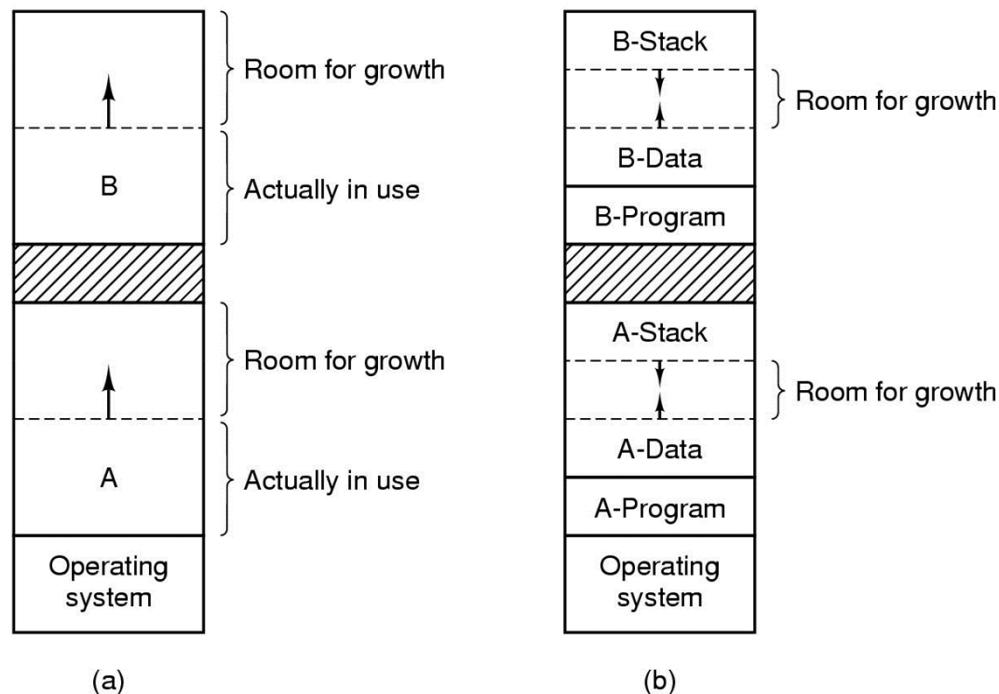
- Compactação de memória é muito cara
  - da ordem de segundos para alguns MBs de RAM



# Swapping

Como lidar com processos que crescem (em demanda de memória)?

- Tentar alocar uma partição para o processo que é vizinha de uma partição não usada (nem sempre é possível)
- Alocar uma partição conjunta para a pilha e o *heap*, e fazê-los crescer em sentidos opostos.
- Se processo usa todo espaço de memória disponível, fazer um *swap out*, e um *swap in* em uma partição maior (mas, se disco de swap está cheio, processo precisa ser terminado)





# Gerenciamento do espaço de memória livre

Idéia: dividir a memória em *unidades de alocação* de n bytes<sup>1</sup> e representar a ocupação (livre/ocupado) de lotes de unidades usando um bit map ou uma lista encadeada

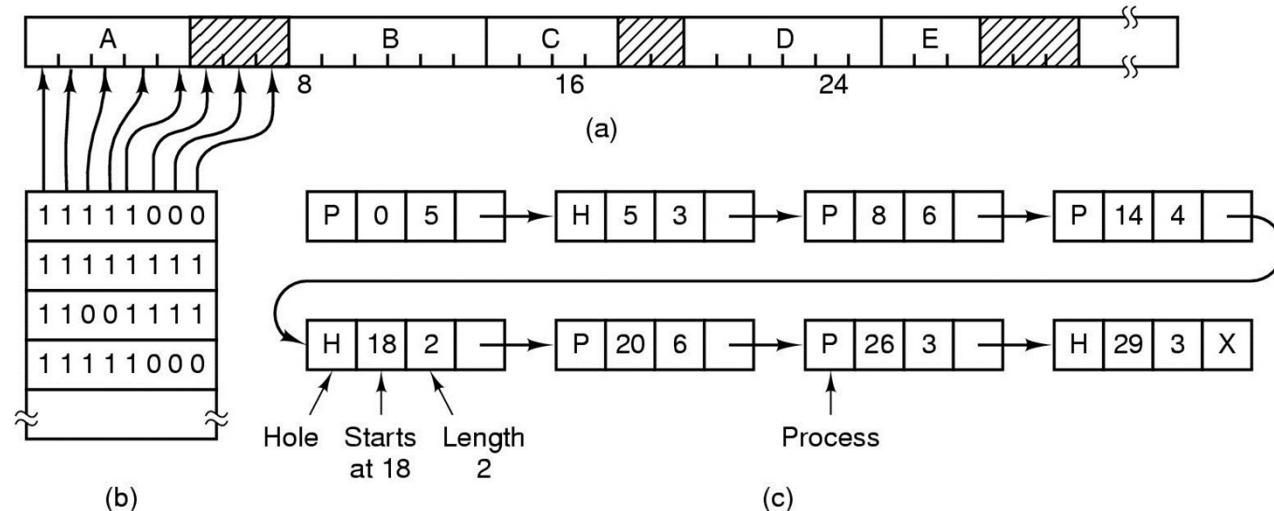


Fig: Representação da ocupação da memória com 5 processos e 5 lacunas em Bit Map (b) ou Lista encadeada (c)

Bit Map: armazenamento compacto e simples, mas busca por determinado tamanho de lote livre pode envolver análise de várias palavras

Lista ligada : cada nó contém o endereço inicial e o tamanho de uma partição ocupada ou livre

(1) Um Minix ‘Click’ = 1KB



# Gerenciamento de Memória com Listas

- Quando o processo é swapped out, a lacuna correspondente precisa ser combinada com espaços vizinhos livres.
- Quando processo é swapped in, percorre-se a lista buscando um espaço livre suficientemente grande (lista geralmente ordenada por endereços de memória)

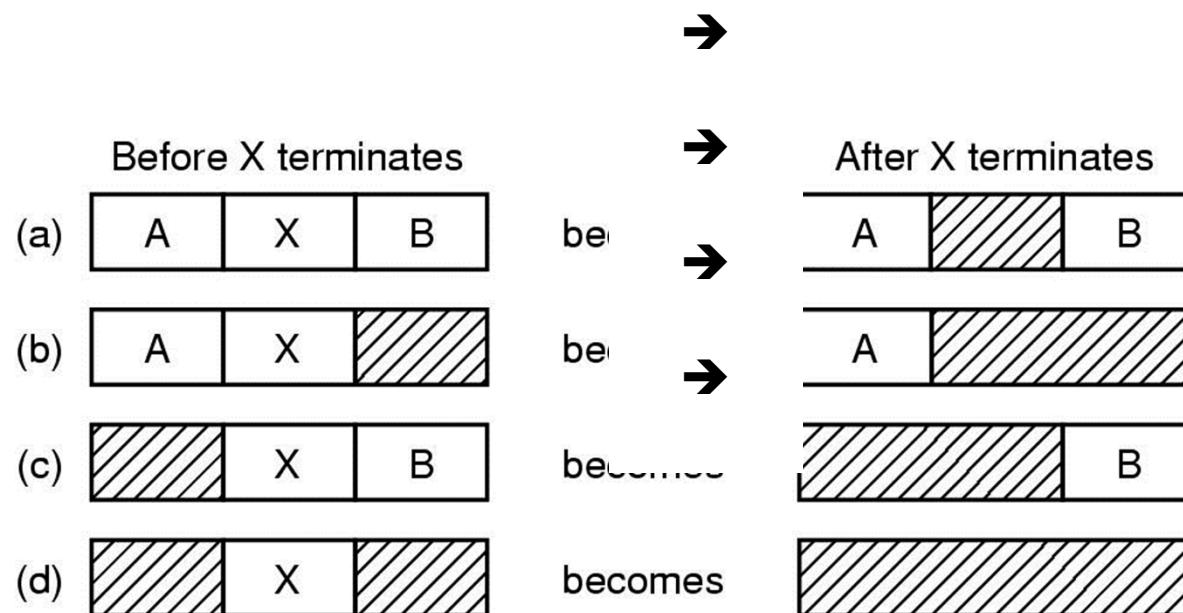


Fig: Quando X é swapped out: quatro combinações de nós na lista



# Gerenciamento de Memória com Listas

Possíveis algoritmos de seleção de espaço livre:

- **First Fit** – percorre a lista e aloca o primeiro espaço encontrado
- **Next Fit** – como first Fit, só que a partir da posição na lista onde foi feita a alocação anterior
- **Best Fit** – percorre toda a lista e aloca o menor possível espaço → pode deixar fragmentos muito pequenos para alocação para outros processos
- **Worst Fit** – percorre toda a lista e aloca o maior possível espaço

Em muitos sistemas, o overhead adicional exigido pelos Best/Worst Fit não valem a pena para obter uma alocação mais efetiva.



# Gerenciamento de Memória com Listas

Qualquer um dos algoritmo anteriores é mais eficiente se:

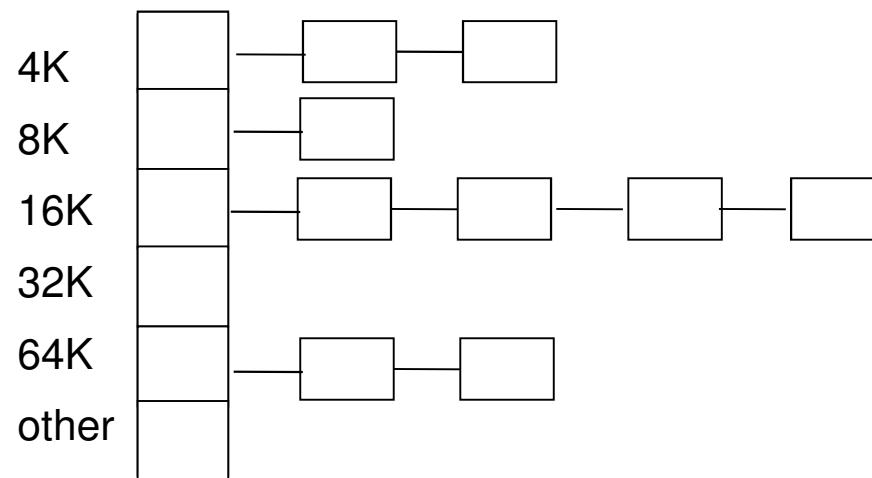
- houverem 2 listas: lista de partições usadas + lista de espaços livres
- Listas são mantidas ordenadas por tamanho (nr. de unidades de alocação)

Problema: há uma complexidade extra quando ocorre uma liberação de memória (precisa-se verificar se há espaços adjacentes livres e inserir o novo espaço na posição correta da lista).

Alternativa:

**Quick Fit:** mantém listas separadas por tamanho do espaço livre (2K, 4K, 8K, etc.)

- Problema: ao liberar memória, o novo espaço criado precisa ser inserido na fila correspondente (possivelmente, após combinação com áreas vizinhas)





# Memória Virtual

É necessária, quando o total de memória utilizada para um conjunto de processos excede o tamanho da memória física. Também aqui, usa-se parte do disco como extensão da memória RAM.

A grande maioria dos SO's (exceto alguns para tempo real), implementam Memória Virtual.

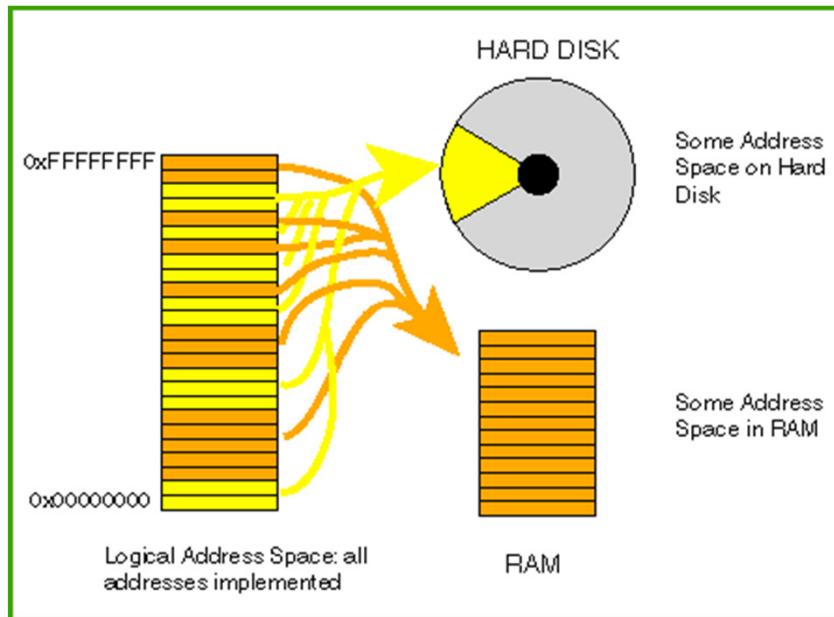
MV usa a técnica de paginação:

- Memória física e espaço de endereçamento lógico de cada processo são divididos em partições de mesmo tamanho:
  - Espaço do processo é dividido em *páginas*
  - Memória é dividida em quadros de página
- Em vez de fazer o swap in/out de uma imagem inteira de processo, cada página pode ser movida do disco para a memória e vice-versa.



# Memória Virtual

Portanto, cada processo pode não estar completamente na memória, mas apenas as partes (paginas) que as instruções em execução estão usando.



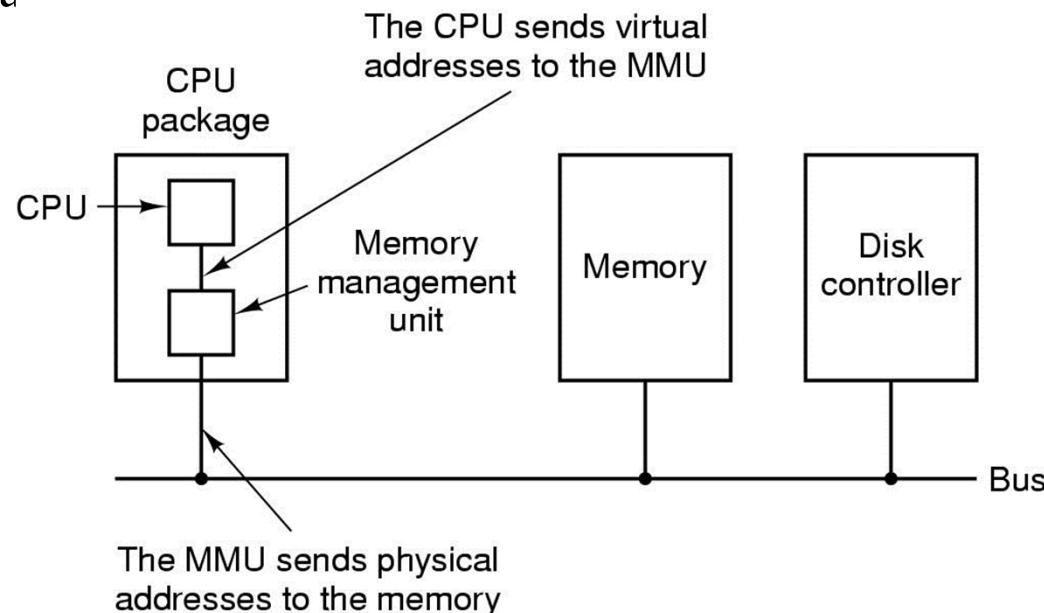
- A cada acesso à memória verifica-se se a página correspondente está na memória; se não estiver, ocorre uma interrupção e um tratador de páginas faltantes (page-fault) a copia para a memória.
- A espera pela cópia para a memória de uma página é equivalente a um bloqueio por E/S. Processo só volta a ficar pronto quando a página requisitada foi copiada para memória.



# Paginação

Requer da existência de suporte por hardware (Memory Management Unit -MMU)

- MMU intercepta qualquer acesso à memória (p/ instruções e dados)
- Mapeia endereços lógicos para endereços físicos (através de uma *tabela de página*)
- Quando página acessada não está em memória, gera uma interrupção de falta de página (*page fault*), que causa a interrupção do processo em execução e o seu bloqueio até que a página tenha sido transferida p/ a memória





# Exemplo Memória Virtual

Exemplo: tamanho página = 4KB (varia de 512 Bytes a 64 KB)

Um espaço de endereçamento virtual de 64K é mapeado em 32KB de memória RAM

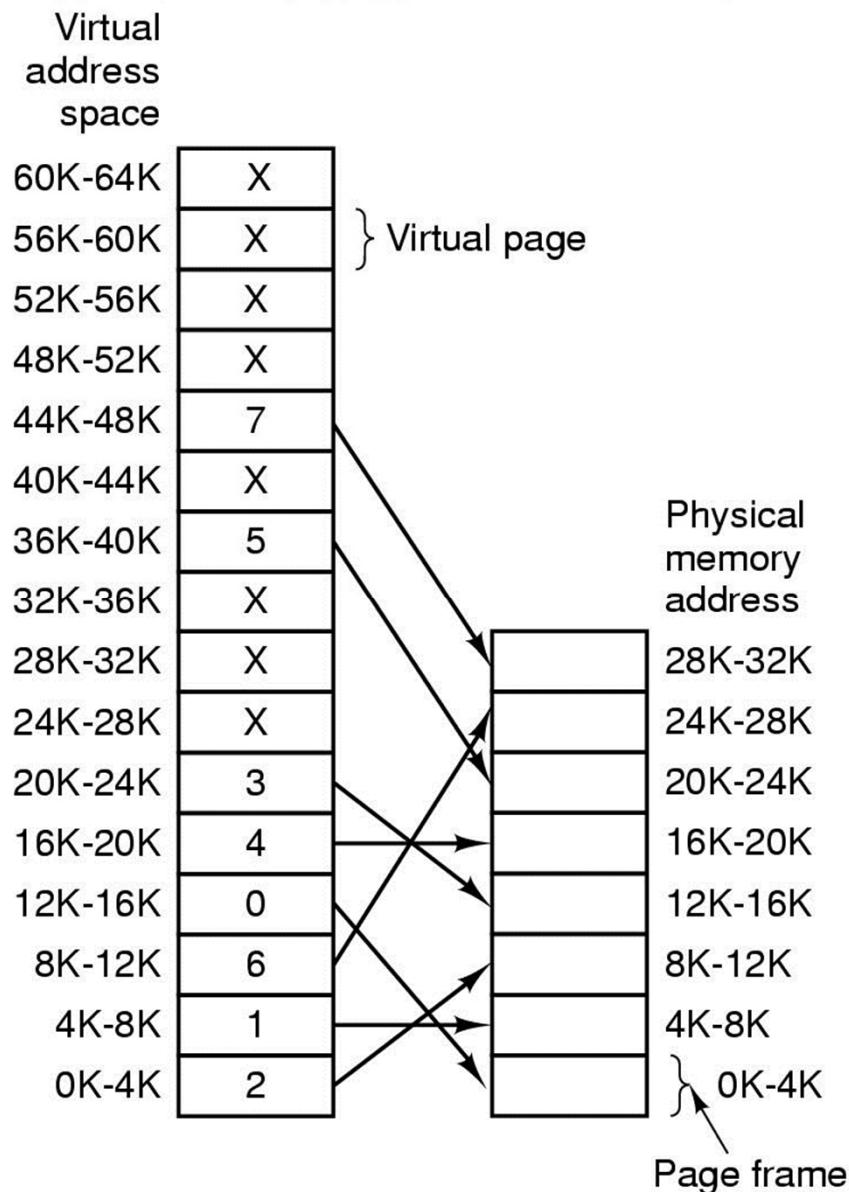
Número da página é usada como índice para a tabela de páginas (TP)

Cada entrada na TP contém, entre outros:

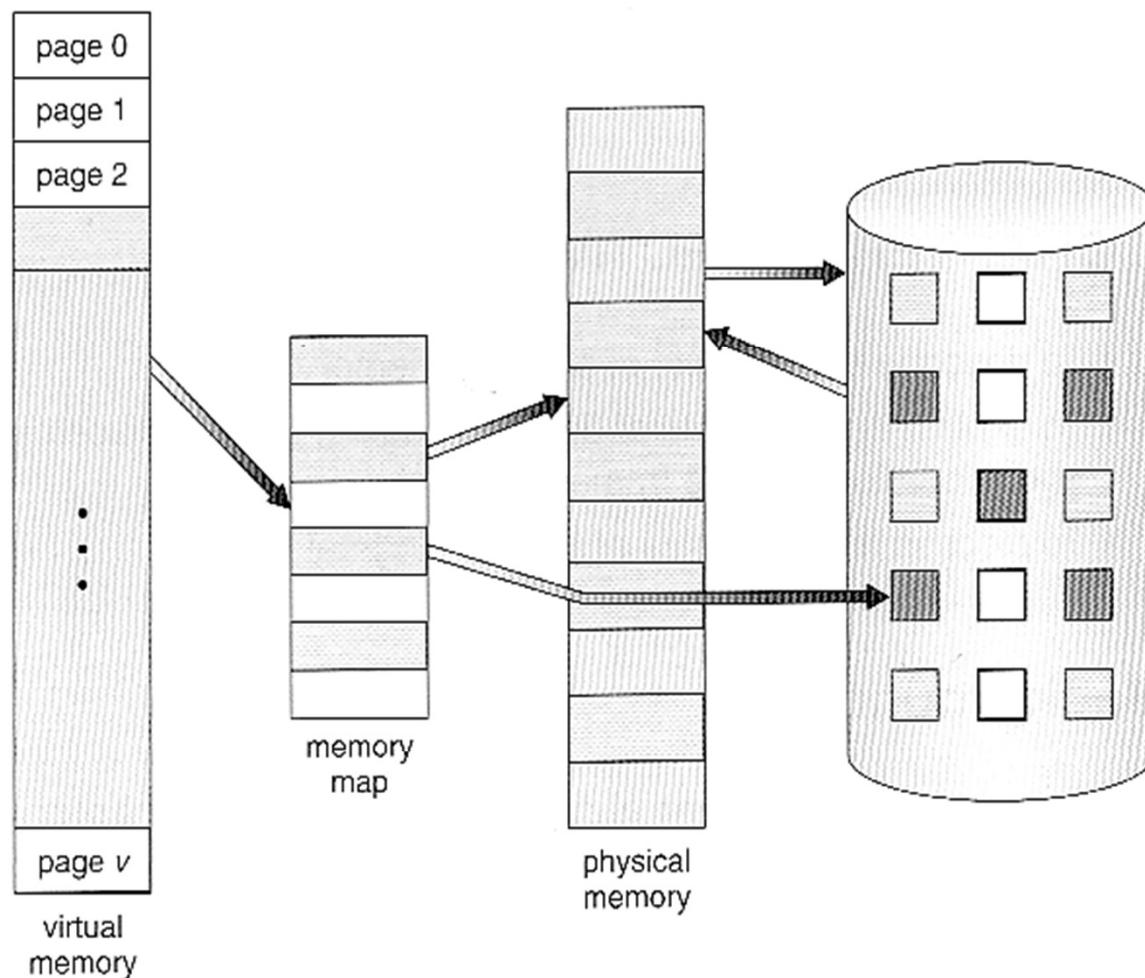
- Bit presença: se página está em memória
- Se estiver, o número do quadro correspondente

Endereço lógico X é mapeado para endereço físico:

$$TP(X/4K) + X \% 4K$$



# Tabela de Páginas



**Figure 9.1** Diagram showing virtual memory that is larger than physical memory.

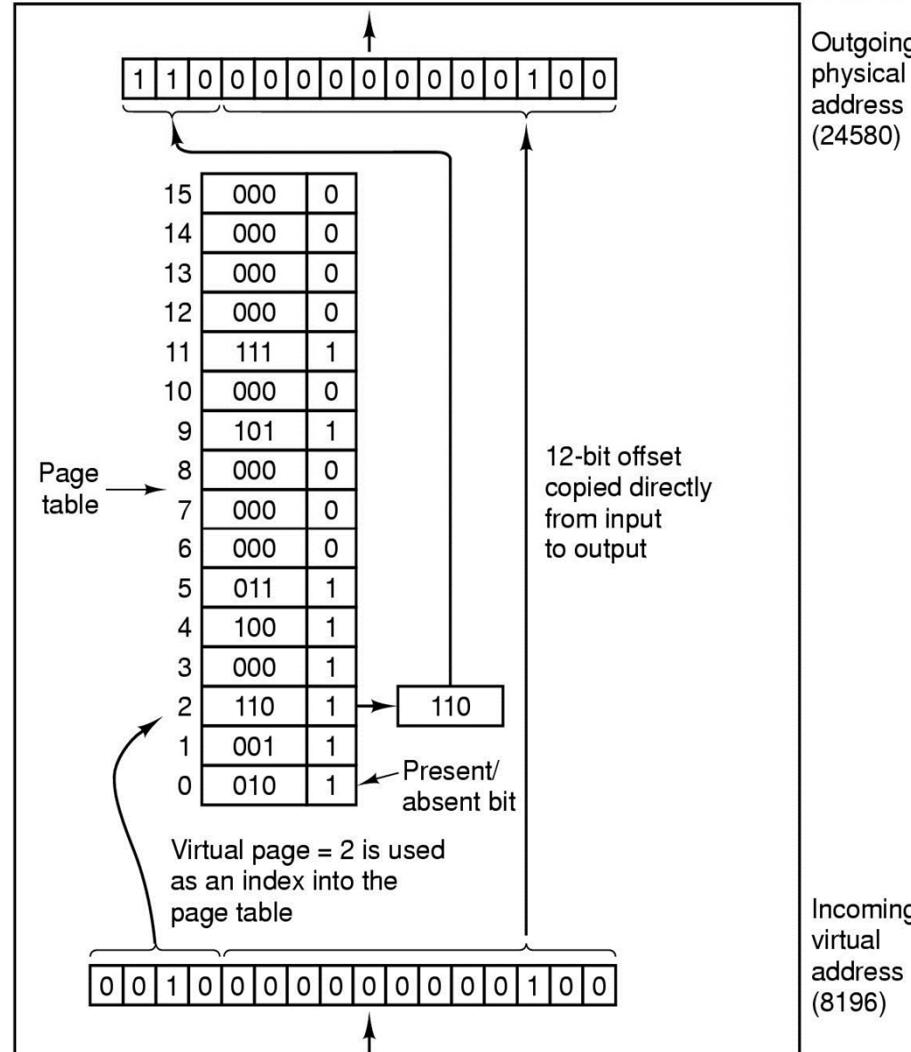


# Tabela de Páginas

Exemplo: Operação interna a uma MMU com 16 páginas de tamanho 4 KB cada

Composição do endereço lógico:

- Bits mais significativos = número da página
- Bits menos significativos = deslocamento do endereço dentro de uma página





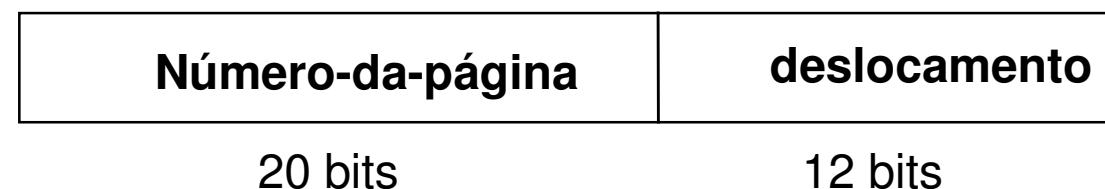
# Paginação

Memória física é partitionada em quadros de página (frames) de tamanho  $2^x$  bytes

Espaço de endereçamento lógico é dividido em páginas de tamanho  $2^x$  bytes

Cada endereço lógico é composto do par (número-da-página, deslocamento), onde número-da-página é usado como índice para uma entrada na tabela de páginas

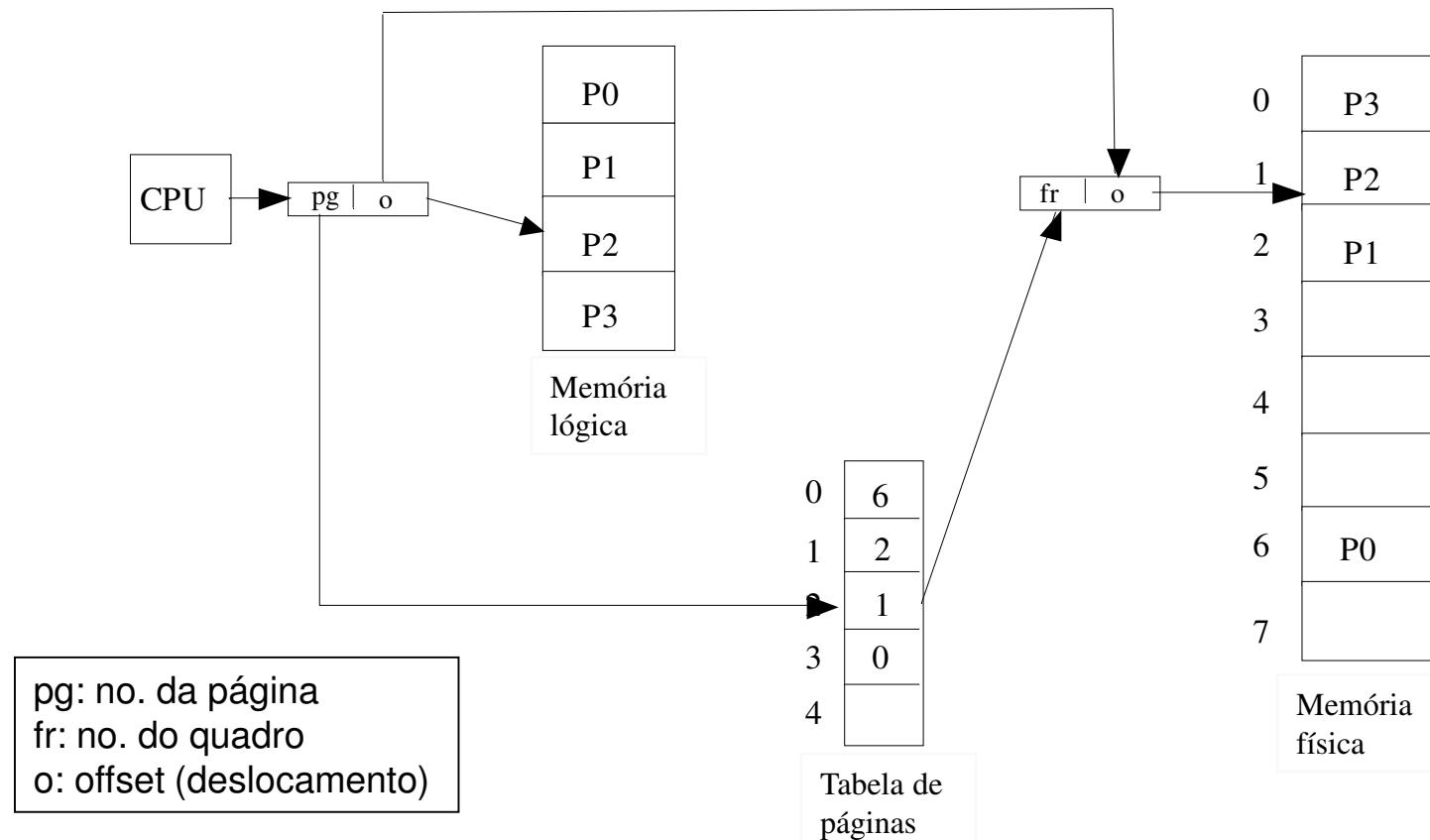
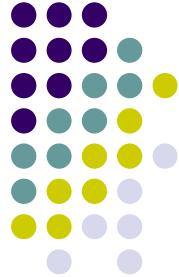
Exemplo: páginas de 4 K para endereços de 32 bits:



**Obs:**

- Em paginação, de certa forma, tem-se um registrador de realocação por página
- Região de memória física ocupada por um processo não precisa ser contígua
- Evita fragmentação externa, mas causa fragmentação interna

# Paginação



Processo pode ser executado contanto que haja um número mínimo de páginas na memória (as páginas sendo acessadas)

O espaço de endereçamento lógico é contíguo, o espaço físico equivalente é distribuído/separado.

A tradução é feita de forma transparente pela MMU, que além da TP, mantém um cache das páginas recentemente consultadas.



# Paginação

Cada processo **P** possui a sua tabela de páginas, que precisa ser carregada na MMU a cada troca de contexto.

Para tal, cada entrada na TabelaProcessos(**P**) contém um ponteiro para a tabela de páginas de **P**. O dispatcher é o encarregado de "carregar" a nova tabela de páginas na MMU.

Como qualquer acesso à memória ocorre através do mapeamento pela TP, **isso fornece também automaticamente um mecanismo de proteção...**

**... contanto que o preenchimento da tabela de página seja feita em modo privilegiado (supervisor)!!**

=> à medida que as páginas vão sendo alocadas, o núcleo preenche as entradas na tabela de página.

Além disto, kernel precisa manter informações sobre o conjunto de quadros livres na memória principal:

Para isso, usa-se uma tabela de quadros (**frame-table**), com uma entrada por quadro, informando se o mesmo está alocado, e para qual processo.

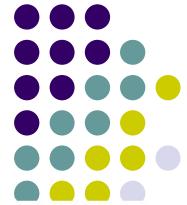
=> De uma forma geral, a paginação aumenta o tempo de troca de contexto. **Por que?**



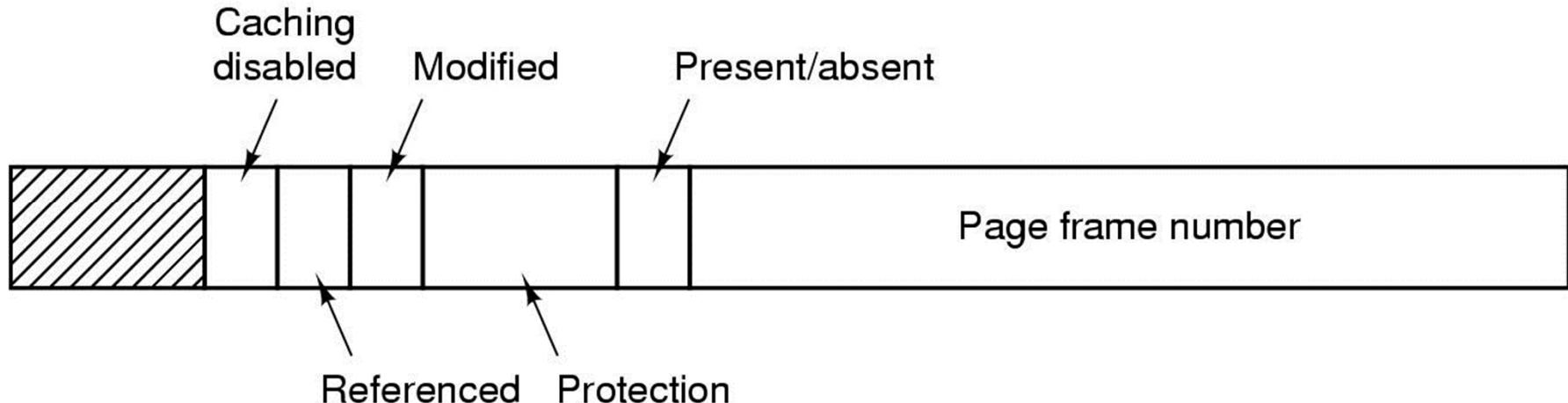
# Controle de acesso

**A paginação pode ser facilmente estendida para incorporar controle de acesso para cada página:**

- além do endereço do quadro da memória, cada entrada da TP contém bits para o tipo de acesso permitido, podendo ser:  
*somente-leitura, leitura-e-escrita ou somente-execução*
- se uma instrução viola o acesso permitido, a MMU gera outra interrupção (violação de acesso de memória)
- a validade de uma operação sob um endereço lógico pode ser testada em paralelo com a obtenção do endereço físico correspondente.



# Entrada da Tabela de Páginas



Resumo dos campos/flags:

- Caching: se faz sentido guardar essa entrada da TP no cache
- Referenced: se houve acesso a algum endereço da página nos últimos  $\Delta t$
- Modified: se houve acesso de escrita a algum endereço da página nos últimos  $\Delta t$
- Protection: Controle de acesso
- Present: se página está na memória
- Número do quadro na memória física



# A tabela de páginas

## Possíveis implementações:

1) TP é mantida em um conjunto de registradores dedicados, que são carregados através de instruções privilegiadas (ex. DEC PDP-11)

Pró: não necessita de MMU e tradução é veloz

Contra: número de entradas é pequeno (tipicamente, de 16 a 64)

2) TP é mantida em memória principal

- mantém-se um registrador com o endereço base da tabela (Process Table Base Register, **PTBR**) e entrada na tabela = PTBR + #página\_virtual

Prós:

- possibilita tabelas de páginas arbitrariamente grandes
- a troca de contexto envolve somente PTBR

Contras:

- tempo de acesso à memória duplica, devido ao acesso à tabela



# A tabela de páginas

Possíveis implementações (cont.):

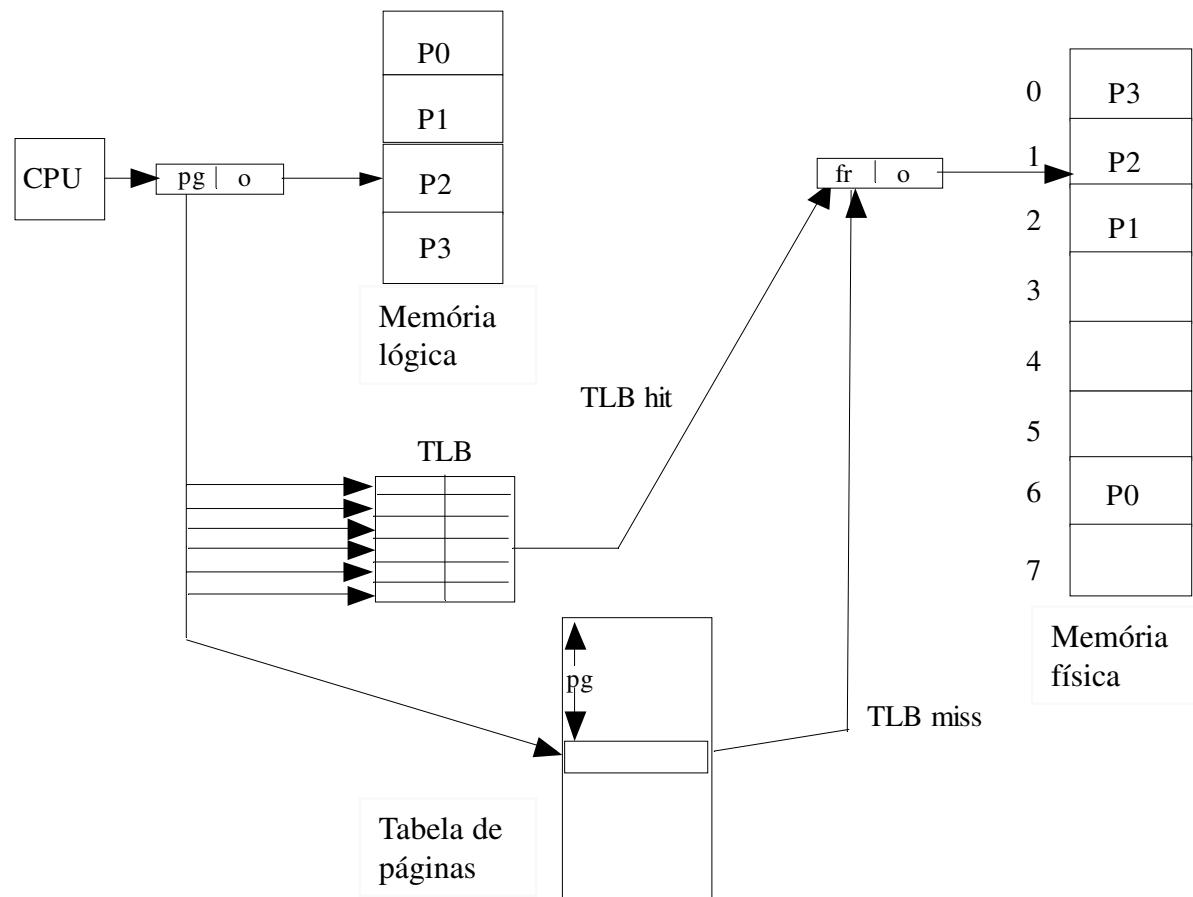
## 3) utilização de um Translation Look-aside Buffer (TLB) na MMU:

- TLB = vetor associativo que permite comparação paralela com suas entradas (de 8 - 128 entradas; Intel 80486 tem 32 entradas)
- mantém-se apenas as entradas da TabPáginas das páginas recentemente acessadas ( $\rightarrow$  princípio de localidade)
- quando uma página lógica é acessada e seu mapeamento não está no TLB (TLB miss) então:
  - \* acessa-se a TabPáginas na memória e substitui-se a entrada no TLB
  - \* dependendo se a página está em memória ou não, continua-se a execução, ou gera-se uma interrupção *Page-fault*
- a cada troca de contexto, o TLB precisa ser limpo

Vantagem: tradução rápida

Desvantagem: requer gerenciamento do conteúdo do TLB (substituição de entradas)

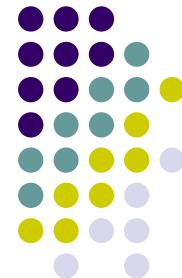
# Translation Look-aside Buffer



Taxa de TLB hits =  $TLB\ hits\ em\ \Delta t / \# \ de\ acessos\ a\ memória\ em\ \Delta t$

É usada para calcular o tempo médio de acesso à memória (eficiência da tradução)

Taxa de TLB hits depende do tamanho do TLB e da estratégia de troca de entradas



# Funcionamento da MMU com TLB

Ao receber um novo no. de página (np) todas as entradas do TLB são comparadas em paralelo:

- Se np é encontrado e tipo de acesso é válido, então usa-se a entrada do TLB (endereço do quadro)
- Se tipo de acesso é inválido, gera-se uma falha de proteção (e processo é abortado)
- Se np não é encontrado na TLB, (*TLB miss*), MMU consulta a TP em memória, obtém o end. do quadro, e copia a entrada completa da TP para o TLB (para agilizar acessos futuros)
- Quando houver acesso p/escrita, copia-se o flag de escrita (Modify bit) de TLB para TP

Obs: Pode-se aumentar a eficiência da paginação, aumentando-se o tamanho do TLB.



# TLBs – Translation Look-aside Buffer

Possível configuração do TLB para um programa com:

- Um loop cujas instruções internas estão armazenadas nas páginas 19-21, acessando um vetor nas páginas 129-130. O índice do vetor está na pagina 140 e a pilha em 860-861.

<b>Valid</b>	<b>Virtual page</b>	<b>Modified</b>	<b>Protection</b>	<b>Page frame</b>
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75



# TLBs – Translation Look-aside Buffers

Em arquiteturas RISC(\*), TLB misses são tratados pelo Sistema Operacional e não pelo hardware (MMU).

Quando a MMU gera um *TLB-miss*, o núcleo faz o troca da entrada da TLB.

Desvantagem: gerenciamento do TLB é muito mais lento do que se for feito por hardware

Vantagem: Torna o circuito da MMU bem mais simples, e permite a implementação em muitas arquiteturas;

Possíveis otimizações:

- Pré-carregar as entradas TLB para um processo que vai ser escalonado, e não apenas preencher o TLB à medida que a TP vá sendo usada
- Manter em algumas entradas especiais do TLB o número das páginas onde estão as tabelas de páginas dos processos

(\*) RISC= Reduced Instruction Set Computers, arquitetura com conjunto de instruções reduzido, de baixo nível de complexidade, e que levam aprox. mesmo tempo para serem executadas.

Exemplos: MIPS, Alpha, PowerPC, etc. (obs: x86 é CISC (Complex...))



# Tabelas de página

- Principal problema é que a própria TP precisa estar em memória.

Arquitetura de 32 bits.

Tamanho da página = 4K

Offset = 12 bits

Tabela de página →  $2^{20}$  entradas!

- E pior, precisa-se manter uma TP por processo em execução.
- Então, dependendo do número de entradas de cada TP (que depende do tamanho da página e do número de bits da arquitetura), pode ser inviável manter todas elas na memória do núcleo.
- O que fazer?

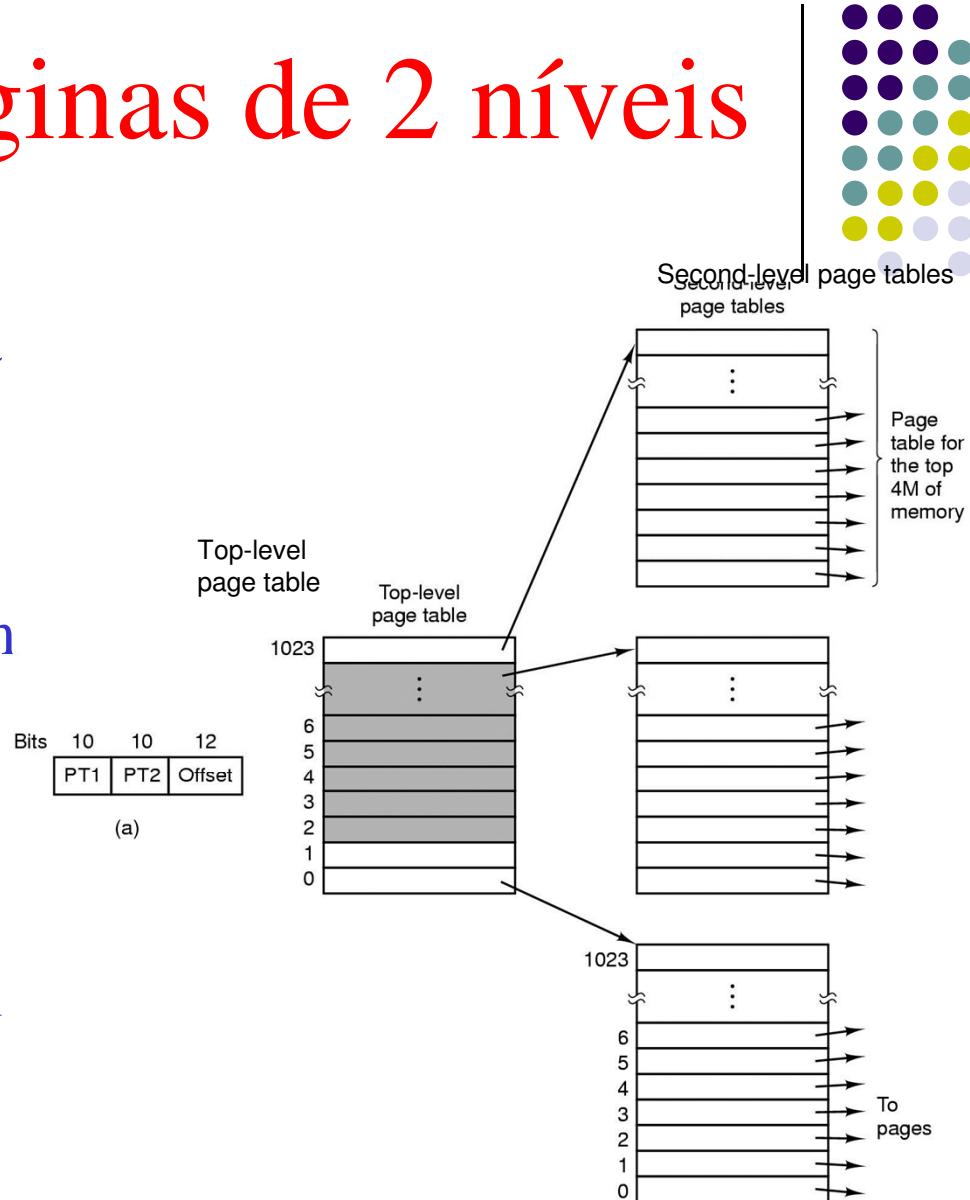
# Tabela de Páginas de 2 níveis



Resolve o problema de manter grandes tabelas de página na memória (para todos os processos).

Com TP de 2 níveis, pode-se partitionar a TP principal em páginas e manter apenas as páginas da TP cujo mapeamento esteja sendo usado.

Uma tabela de 1º. nível contém os ponteiros para essas partes da TP.





# Tabela de páginas Invertidas

Para arquiteturas de 64 bits, o espaço de endereçamento lógico é  $2^{64}$  bytes!

Exemplo: se o tamanho de página for 4KB, isso leva a TP's muito grandes, e.g.  $2^{52} \approx$  alguns Tera Bytes!).

Por isso, em alguns sistemas usa-se uma TP invertida:

- Uma entrada p/ cada quadro de página
- Cada entrada contém uma lista de pares: (processID, nº da página)

Desvantagem: a tradução de endereços lógicos para reais fica bem mais lenta, i.e. para cada página  $p$ , precisa-se fazer uma varredura pela TP invertida para encontrar o quadro correspondente.

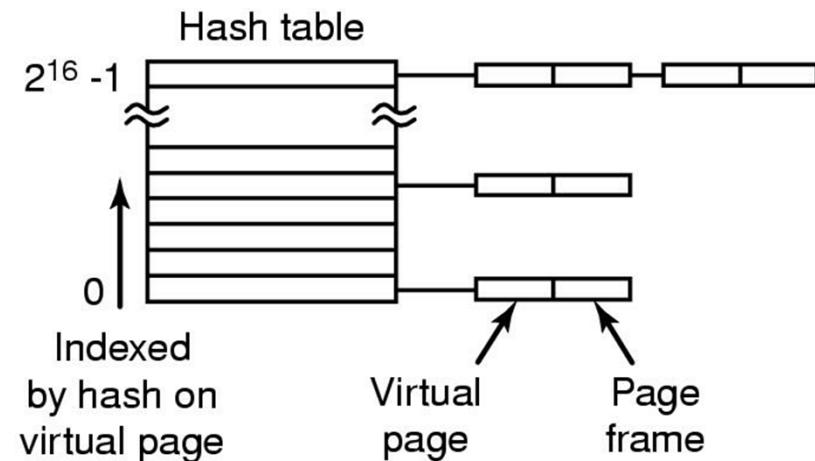
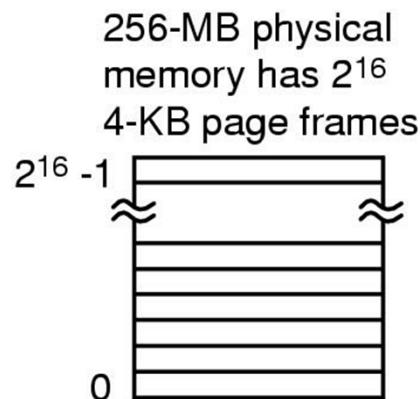
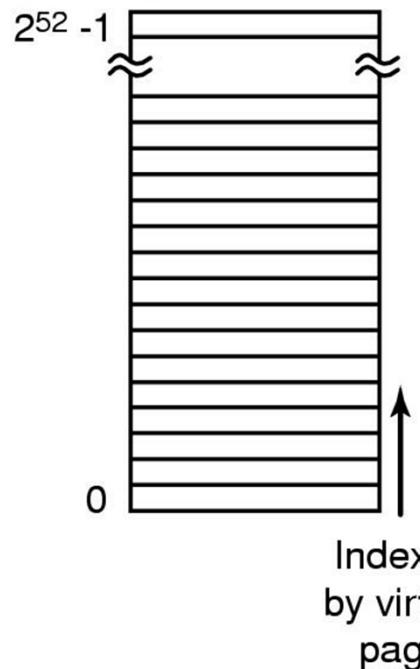
Só é viável, se o TLB for usado para guardar as associações correntes. Somente quando ocorre um *TLB miss*, a TP invertida precisa ser consultada.

Na maioria dos sistemas, usa-se uma função de hash para indexar as entradas da TP invertida.

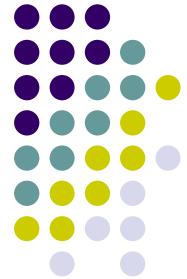


# Tabela de Páginas Invertidas

Traditional page  
table with an entry  
for each of the  $2^{52}$   
pages



Comparação de TP tradicional e TP invertida.



# A questão do tamanho de página

Tamanho de página pequenos:

- **Vantagens**
  - Melhor aproveitamento da memória física: menos fragmentação interna
  - mais programas usando a memória
  - melhor alinhamento de segmentos de código e de estruturas de dados às páginas
- **Desvantagens**
  - programas precisam de mais páginas, tabelas de página maiores, mais tempo para carregar entradas de tabela nos registradores da MMU
  - Transferência de página menor ou maior de/para disco demanda aproximadamente o mesmo tempo (mas é feito de forma mais frequente)



# Algoritmos de Substituição de Páginas

A cada page-fault:

- o gerenciador de memória aloca um quadro de páginas para a página requisitada e
- precisa escolher qual página (de um quadro de páginas) deve ser removida da memória.

O Algoritmo de substituição de páginas determina *qual* página deve ser removida e usa as informações estatísticas contidas nas tabelas de páginas.

Nessa seleção deve-se levar em conta que:

- uma página modificada precisa ser copiada para disco, mas uma não-modificada pode ser sobreescrita
- não é boa idéia tirar uma página que “está em uso”, pois ela terá que ser carregada em breve



# Algoritmo de substituição ideal

- Substitui a página que será acessada no futuro mais remoto
  - Infelizmente, não é viável na prática, pois exigiria um conhecimento sobre todos os acessos futuros
- Usado apenas em simulações para avaliar o quanto os algoritmos concretos diferem do algoritmo ideal
- Que algoritmo usar?

# Algoritmo Página não recentemente Utilizada (Not Recently Used - NRU)



- Cada página tem um bit de acesso (R) e de modificação (M):
  - Bits são setados sempre que página é acessada ou modificada
  - Página é carregada com permissão somente para leitura e M=0.
  - No primeiro acesso para escrita, o mecanismo de proteção notifica o núcleo, que seta M=1, e troca para permissão de escrita
  - A cada interrupção do relógio, seta-se R=0
- Páginas são classificadas em 4 categorias
  1. não referenciada, não modificada
  2. não referenciada, modificada
  3. referenciada, não modificada
  4. referenciada, modificada
- NRU escolhe primeiro qualquer página (em ordem ascendente de categoria)  
→ Razão da prioridade de descarte (#2 antes #3): É melhor manter páginas sendo referenciadas, do que modificadas mas pouco referenciadas.



# Algoritmo FIFO de substituição

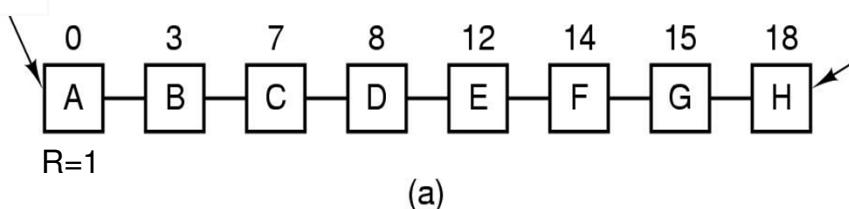
- Manter uma lista ligada de todas as páginas na ordem em que foram carregadas na memória
- A página (mais antiga), no início da fila, é a primeira ser descartada
- Principal Vantagem:
  - Simplicidade de implementação.
- Principal Desvantagem:
  - Não é levado em conta se uma página está sendo frequentemente acessada ou não.
  - Por exemplo: a página mais antiga pode estar sendo acessada com alta frequência.



# Algoritmo da Segunda Chance

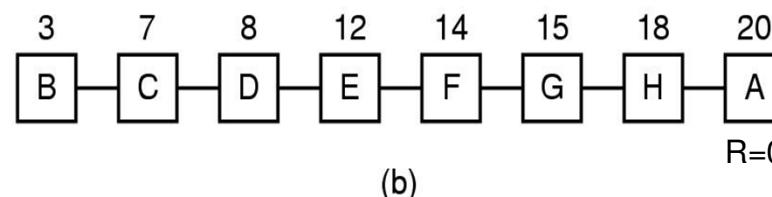
- Uma variante do algoritmo FIFO que leva em conta acessos recentes às páginas:
  - Páginas são mantidas em fila FIFO (ordenada por momento de carregamento)
  - Se página mais antiga possui bit  $R=0$ , ela é removida.
  - Se tiver bit  $R=1$ , o bit é zerado, e a página é colocada no início da fila,. Ou seja: dá se uma 2<sup>a</sup> chance.

Página carregada há mais tempo.



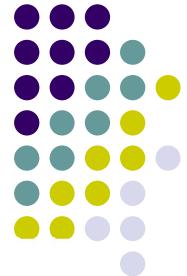
(a)

Página carregada mais recentemente.

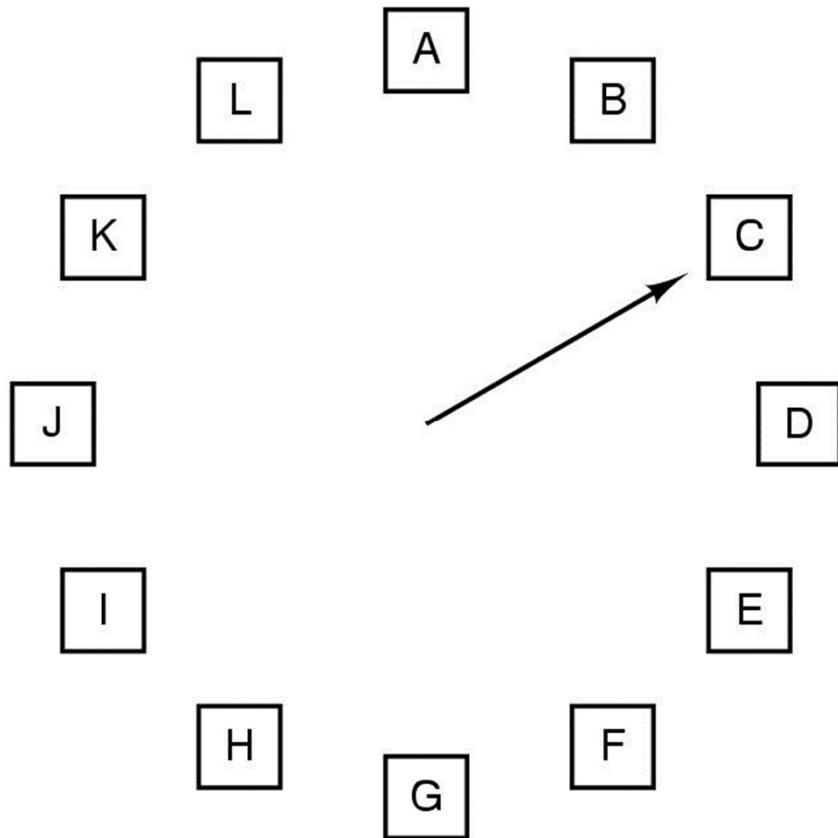


(b)

A página é tratada como se fosse carregada recentemente.



# Algoritmo do Relógio



When a page fault occurs,  
the page the hand is  
pointing to is inspected.  
The action taken depends  
on the R bit:

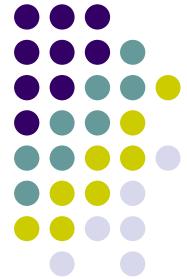
R = 0: Evict the page

R = 1: Clear R and advance hand

Todos as páginas carregadas estão em uma lista circular.

Ponteiro aponta para a próxima a ser testada

Trata-se simplesmente de uma implementação alternativa para o algoritmo da 2a. Chance (sem a necessidade de manipular a fila)



# Algoritmo “Menos recentemente utilizada”

## Least Recently Used (LRU)

Assume que páginas usadas recentemente, deverão ser usadas em breve novamente.

Princípio: descartar a página que ficou sem acesso durante o período de tempo mais longo

- Implementação ideal, mas impraticável:
  - Lista de páginas ordenadas pelo acesso mais recente
  - É inviável pois demandaria uma manipulação da lista a cada acesso de memória!!
- Alternativa (necessita de apoio de hardware):
  - usar um contador que é incrementado por hardware a cada acesso à memória
  - Cada vez que uma página é acessada, atualiza este contador na entrada da TP correspondente
  - Seleciona a página com o menor contador
  - Periodicamente, zera o contador



# LRU com Hardware especial

Manipular uma matriz  $n \times n$  (para  $n$  entradas na TP) com o registro de acessos recentes (linha  $i$  = contador da página  $i$ )

Idéia: Ao acessar página  $i$  preencha toda linha  $i$  com bit 1 e toda coluna  $i$  com bit 0.  
Página mais antiga é a com menor valor.

	Page			
	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

Acesso Pág. 0

	Page			
	0	1	2	3
0	0	0	1	1
1	0	1	1	1
2	0	0	0	0
3	0	0	0	0

Acesso Pág. 1

	Page			
	0	1	2	3
0	0	0	0	1
1	0	0	1	1
2	1	0	0	1
3	0	0	0	0

Acesso Pág. 2

	Page			
	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	1	0	0	0
3	1	1	1	0

Acesso Pág. 3

	Page			
	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	1	0	0	1
3	1	1	0	0

Acesso Pág. 2

	Page			
	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	1	0	0	1
3	1	0	0	0

Acesso Pág. 1

	Page			
	0	1	2	3
0	0	1	1	1
1	0	1	1	1
2	0	0	0	1
3	0	0	0	0

Acesso Pág. 0

	Page			
	0	1	2	3
0	0	1	1	0
1	0	0	1	0
2	0	0	0	0
3	1	1	1	0

Acesso Pág. 2

	Page			
	0	1	2	3
0	0	1	0	0
1	1	1	0	1
2	1	1	0	0

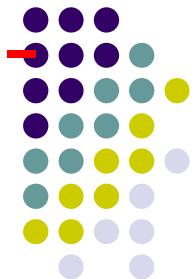
Acesso Pág. 2

	Page			
	0	1	2	3
0	0	1	0	0
1	1	1	0	0
2	1	1	1	0

Acesso Pág. 3

# Algoritmo do envelhecimento

## Simulando LRU em Software



Manter um contador para cada página;

A cada tick de relógio, bit R mantém a informação se página foi acessada durante último período de tempo:

faz-se um shift de 1 bit para a direita no contador de cada página e adiciona-se o R-bit como bit mais significativo no contador

	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
Page	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00100000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000

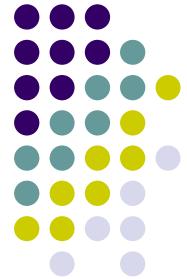
(a) (b) (c) (d) (e)



# Principais limitações do Alg. de Envelhecimento (comparado ao LRU)

- Quando duas páginas possuem o mesmo valor de contador (de idade), não há como saber qual delas foi acessada por último (no último período de tempo)
  - Os contadores de idade têm um número finito de bits. Portanto quando o valor atinge 0, não há como distinguir se a última referência ocorreu a apenas n ticks do relógio ou há muito mais tempo.
- Normalmente, basta contador com 8 bits e períodos de 20 ms entre ticks. Se uma página não foi acessada a 160 ms, provavelmente não é mais relevante.

# Questões de projeto de paginação



Quando um programa gera muitas faltas de página para poucas instruções executadas, ele se encontra em Ultra-paginação (“thrashing”)

Deve-se tentar minimizar a taxa de page faults:

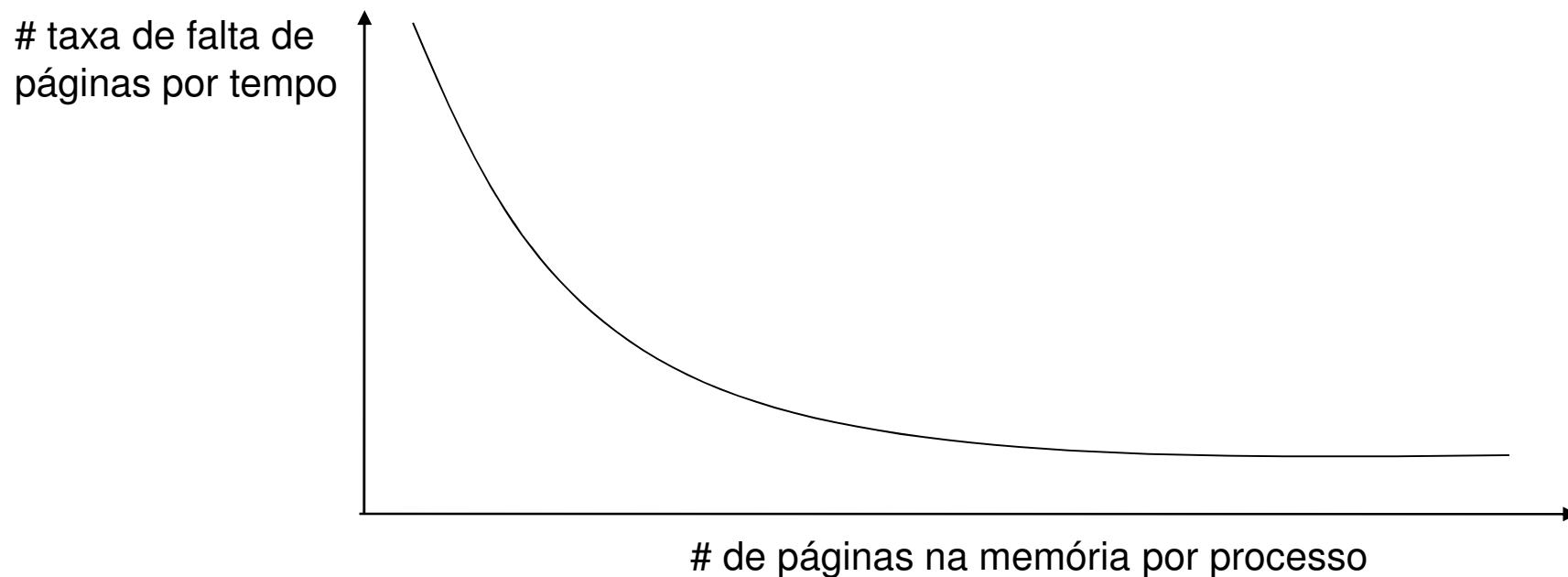
- para cada processo em execução, manter número suficiente de páginas na memória
- para processos que tenham sido movidos para disco e estejam retornando (*swapped-in*), carregar logo várias páginas, evitando que as páginas sejam requisitadas de “forma pingada”, pois isso atrasará ainda mais o processo
- Por isso, precisa-se identificar qual é o conjunto de páginas em uso corrente por cada processo



# Ultra-paginação (Thrashing)

Fenômeno que ocorre quando o gerenciador de memória fica sobrecarregado com cópias de páginas entre memória e disco.

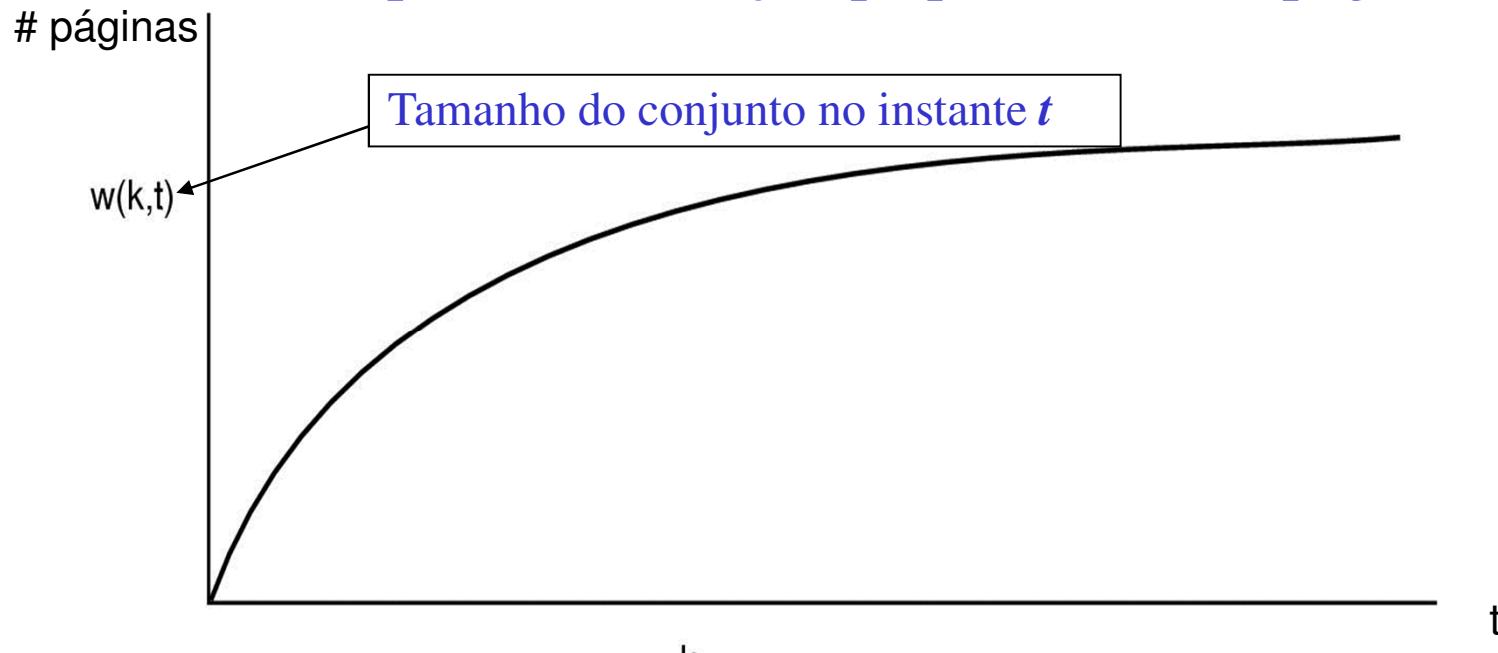
Alta frequência de faltas de página: ocorre quando um processo possui menos páginas na memória do que o seu conjunto de páginas em uso (working-set).





## O Conjunto de páginas em uso (working set)

A maioria dos processos apresenta uma localidade de referência (acessa apenas uma fração pequena de suas páginas).



- O **conjunto de uso** (*working set*) é o conjunto das páginas acessadas pelas  $k$  referências mais recentes. O sistema de paginação deve gerenciar este conjunto usando algoritmo de envelhecimento.
- Quando um processo é escolhido para execução, seu working-set pode ser **pré-paginado** (em vez de fazer paginação por demanda)



## Algoritmo de substituição baseado no working set

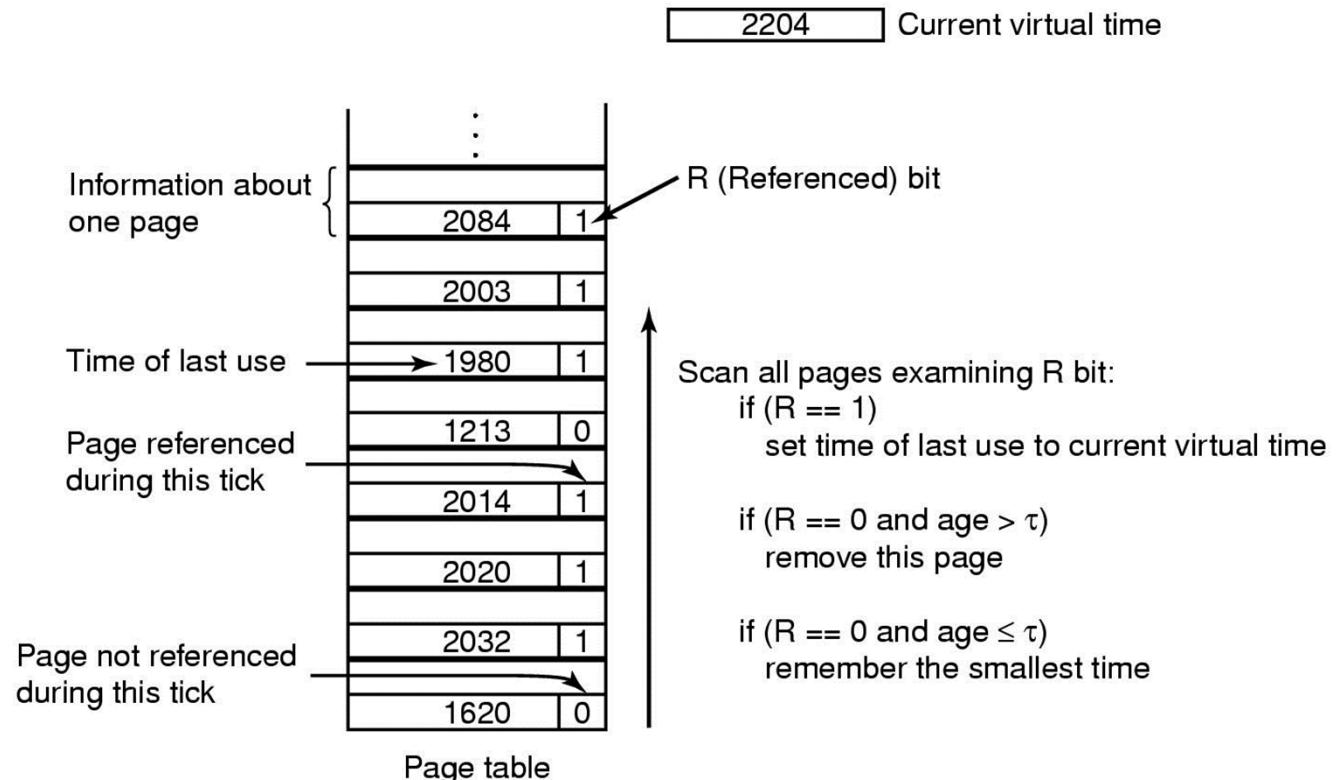
### Ideia Central:

Mantém-se um contador virtual de tempo( $t$ ) e periodicamente, atualiza-se o contador em cada página que tenha o bit  $R=1$ .

Se  $R=0$ , não é feita atualização. Se ( $R==0 \&\& age \leq t$ ) avança com tempo virtual e depois verifica novamente.

Descarta-se páginas com  $R=0$  e mais antigas do que  $t$ .

Working Set =  
páginas com  
contador  
dentro da janela de  
tempo  $[T-t, T]$





# The WSClock Page Replacement Algorithm

Operation of the WSClock algorithm



# Resumo dos algoritmos de substituição

## Algoritmo

Ótimo

NRU (not Recently Used)

FIFO (First-in, First Out)

FIFO second chance

Clock

LRU (Last Recently Used)

NFU (Not Frequently Used)

Aging

Working Set

WSClock

## Eficiência

Não implementável (benchmark)

Muito bruto

Joga fora páginas importantes

Melhoria sobre FIFO

Realista

Excelente mas de difícil implementação

Aproximação simples do LRU

Eficiente, aproxima-se do LRU

Implementação com alto custo

Algoritmo eficiente



# Modeling Page Replacement Algorithms

## Belady's Anomaly

All pages frames initially empty

Youngest page

	0	1	2	3	0	1	4	0	1	2	3	4
0	0	1	2	3	0	1	4	4	4	2	3	3
1	0	1	2	3	0	1	1	1	1	4	2	2
2	0	1	2	3	0	0	0	0	1	4	4	4
P	P	P	P	P	P	P	P	P	P	P	P	9 Page faults

Oldest page

(a)

Youngest page

	0	1	2	3	0	1	4	0	1	2	3	4
0	0	1	2	3	3	3	4	0	1	2	3	4
1	0	1	2	2	2	3	4	0	1	2	3	3
2	0	1	1	1	2	3	4	0	1	2	3	2
3	0	0	0	0	1	2	3	4	0	1	2	1
P	P	P	P	P	P	P	P	P	P	P	P	10 Page faults

Oldest page

(b)

- FIFO with 3 page frames
- FIFO with 4 page frames
- P's show which page references show page faults

# Stack Algorithms



Reference string	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1	
	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1	
	0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4		
	0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3			
	0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7				
	0	2	1	1	5	5	5	5	6	6	6	4	4	4	4	4	4	4	4	4	5	5			
	0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6	6	6	6	6		
	0	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2		
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
Page faults	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P		
Distance string	$\infty$	4	$\infty$	4	2	3	1	5	1	2	6	1	1	4	7	4	6	5							

State of memory array,  $M$ , after each item in reference string is processed



# Política de Substituição Local vs. Global?

- O número de quadros de páginas alocados a cada processo deve variar?
- A página a ser substituída deve ser do processo que causou a falta de página (política local), ou de qualquer processo (política global)?

Processo A  
causou o  
page-fault

	Age
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

(a)

A0
A1
A2
A3
A4
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(b)

A0
A1
A2
A3
A4
A5
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(c)

(a) Configuração original

(b) política local

(c) política global

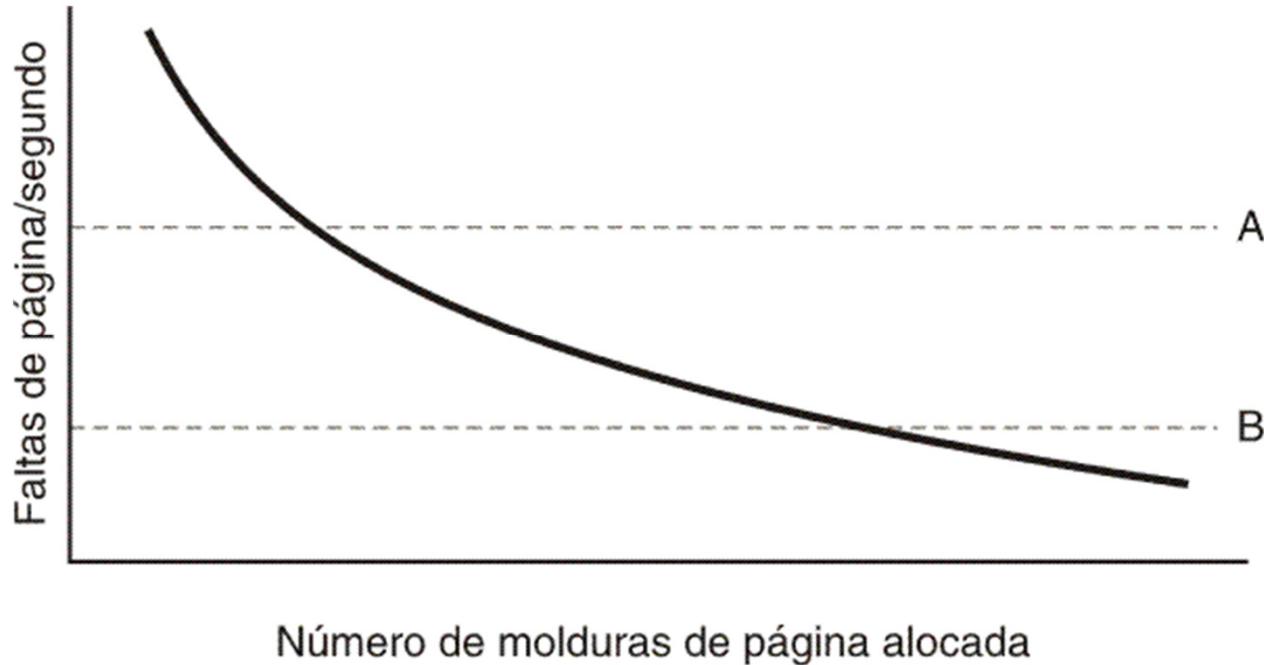


# Política de Alocação Local x Global

Prós e Contras de uma Política Global:

- (+) consegue uma alocação mais eficiente da memória, pois leva em conta flutuações no conjunto de trabalho de todos os processos
- (-) favorece os processos com maior prioridade no escalonamento (geralmente, os mais orientados a E/S), pois eles apresentam mais páginas acessadas recentemente
- (-) alocação é mais complexa, pois precisa monitorar o conjunto de trabalho de todos os processos simultaneamente e comparar a idade de todas as páginas
- (-) o conjunto de trabalho de um processo pode variar muito rapidamente, e o alg. de envelhecimento não reflete isso imediatamente
- (-) pode tornar a execução de um processo inviável se não possuir número mínimo de quadros de página.

# Política de Alocação Local x Global



Freqüência de faltas de página como função do número de molduras de página alocadas

(A)= taxa inaceitavelmente alta de page-fault,

(B)= taxa muito baixa de page-faults, pode-se diminuir # de quadros do processo.

O Algoritmo Page Fault Frequency (PFF) controla o tamanho dos conjuntos de trabalho de todos os processos de forma a que a frequência de page-faults fique entre A e B.



# Algoritmo Page Fault Frequency (PFF)

- O PFF objetiva evitar a paginação excessiva (*thrashing*)
- O que fazer quando o algoritmo PFF indica que:
  - alguns processos precisam de mais memória
  - mas nenhum processo pode ter menos páginas
- Solução :  
**Reducir o número de processos que competem pela memória**
  - Fazer swap-out de alguns processos e liberar os quadros alocados a eles
  - reconsiderar grau de multiprogramação



# Questões de Implementação

Quatro momentos em que SO trata de paginação:

1. Criação do processo
  - . Criação da Tabela de Páginas
2. Quando processo é escolhido para execução na CPU
  - . Re-setar MMU para novo processo, salvando o conteúdo do TLB
  - . Carregar TLB com novo working set
3. Interrupção por falta de página
  - . Identificar qual é o endereço lógico causador da falta
  - . Escolher página a ser substituída/descartada
  - . Carregar página requisitada para memória
  - . Completar a instrução de máquina
4. Quando processo termina
  - . Desalocar TP e páginas na memória e na área de swaping



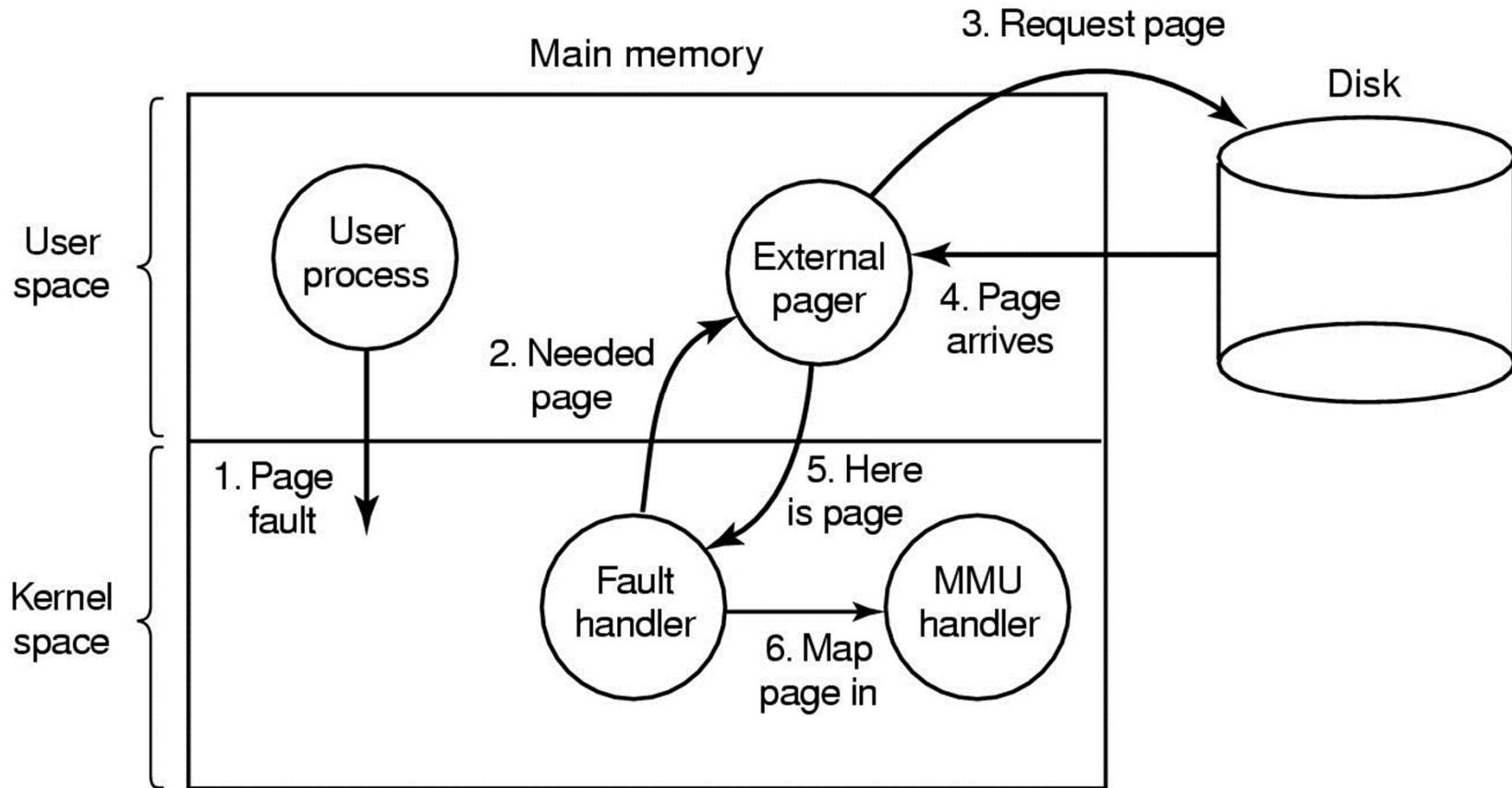
# Bloqueando páginas na memória

Em políticas globais de substituição de páginas, pode haver uma interferência entre paginação e requisições de E/S, que requerem a retenção (bloqueio) temporária de páginas na memória.

Exemplo:

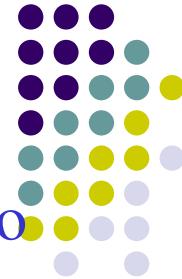
1. Processo1 faz requisição de leitura de dados de um dispositivo para um buffer em uma página na memória
2. Enquanto espera, outro processo 2 causa uma falta de página
3. Por azar, a página do buffer de process01 foi escolhida para substituição
4. Dispositivo não conseguirá copiar dados para o buffer
5. Logo, há necessidade de manter as páginas que são fonte/destino de dados de E/S

# Separação de Política e Mecanismo



- Paginador em espaço do usuário trata os pedidos de falta de página e implementa o algoritmo de substituição.
- O núcleo trata do carregamento de MMU

# Segmentação



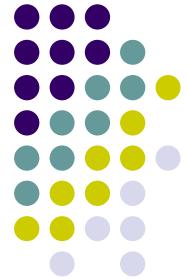
Paginação resolveu o problema da fragmentação externa, mas o espaço de endereçamento (lógico) de um processo é linear.

Segmentação permite visualizar um espaço de endereçamento como uma coleção de espaços de endereçamento (segmentos) independentes e de tamanho variável

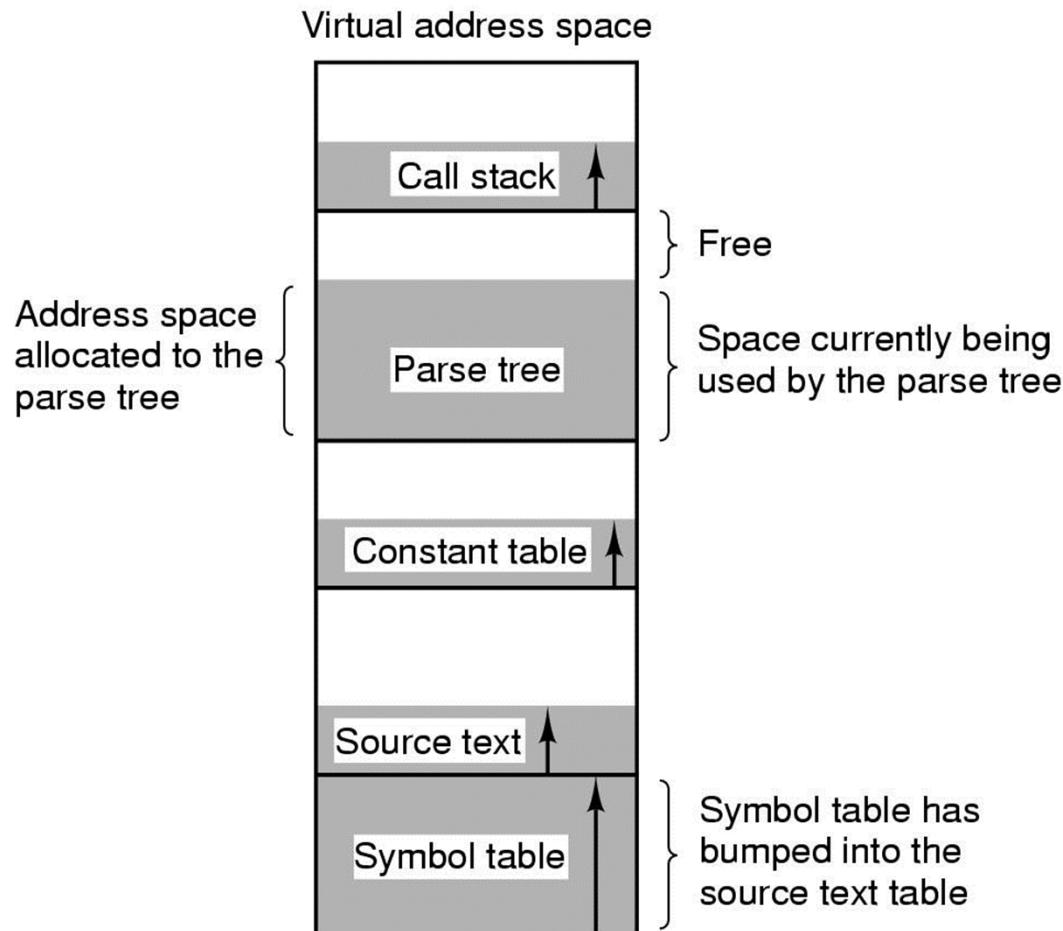
Segmentos são gerados na compilação e/ou estão na forma de bibliotecas

Exemplos:

programa principal, pilha de execução, tabela de símbolos, código de funções, variáveis não inicializadas, bibliotecas compartilhadas, vetores & matrizes, blocos de variáveis globais (common block - Fortran), etc.



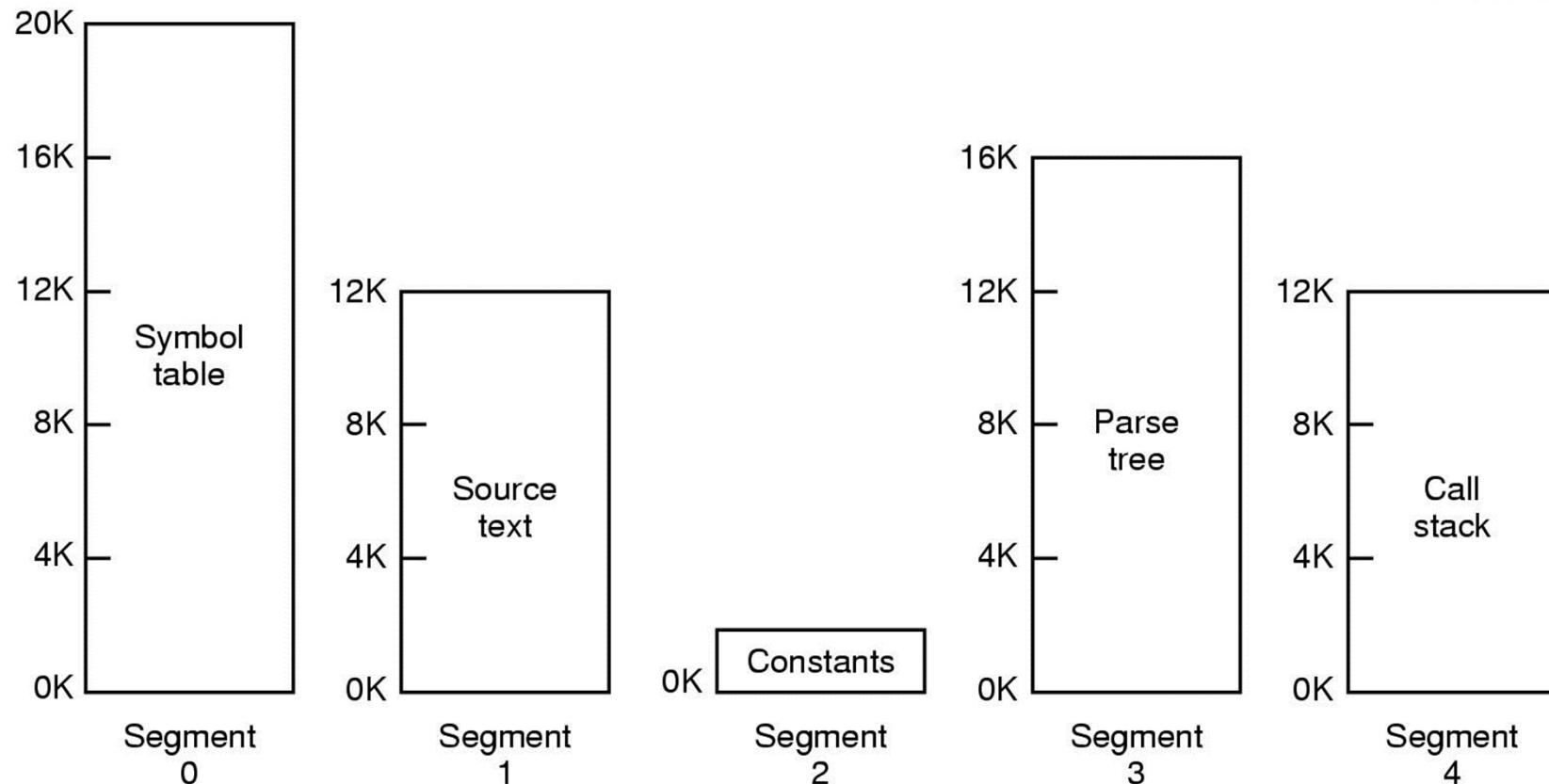
# Segmentação: Exemplo Compilação



- Espaço Virtual com vários segmentos que podem crescer (e acabar usando a memória livre reservada)



# Segmentação



Ideia central: Permitir que cada segmento seja ampliado independentemente.  
→ cada segmento constitui um espaço de endereçamento independente.



# Segmentação

Definição: Segmento = entidade (parte) lógica do programa que contém um único tipo de dados e do qual o programador está consciente.

Exemplos: procedimentos/código, pilha, tabelas, bibliotecas

Vantagens da segmentação:

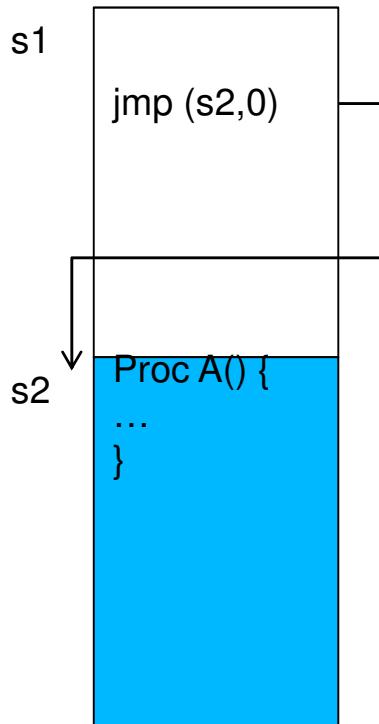
- facilita definir e controlar modos de acesso diferenciados para cada segmento
- segmentos podem crescer independentemente (tabelas dinâmicas, pilhas de threads)
- facilita o compartilhamento de segmentos (p.ex. shared libraries)

Segmentação cria um espaço virtual multi-dimensional.



# Segmentação

Facilita a ligação de código.



Exemplo: quando um procedimento está no começo do segmento (end. 0 do segmento n), pode-se modificá-lo sem que seja necessário alterar qq referência a este procedimento (isto não acontece quando vários procedimentos estão em uma mem. virtual contígua)



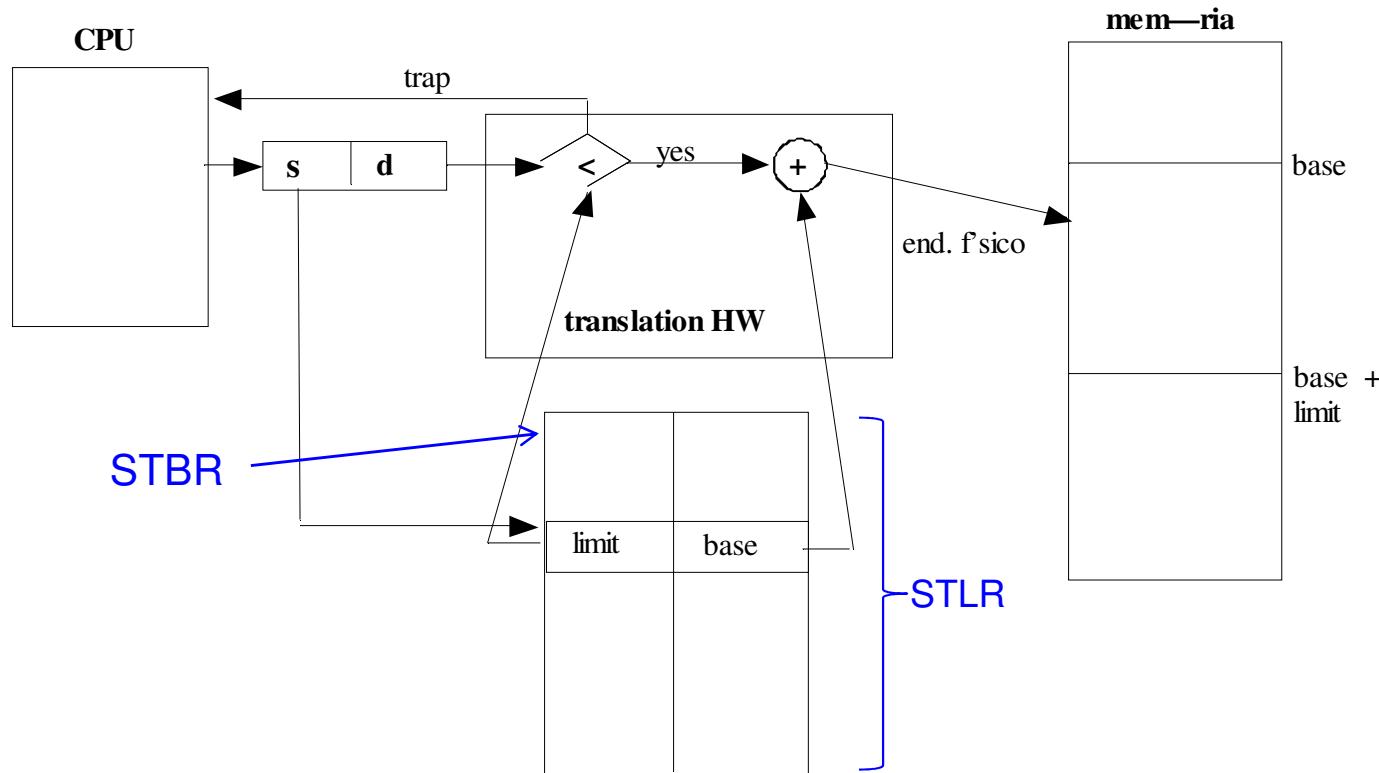
# Segmentação

Alguns sistemas permitem ao programador indicar quais partes do programa devem estar em segmentos independentes.

Endereço lógico consiste de:

- nº de segmento (s) e deslocamento dentro do segmento (d)
- cada processo tem uma **tabela de segmentos**, que é fornecida à MMU para tradução de endereços.
- cada entrada na tabela consiste de par (base, limite), com endereço base e um limite
- como visto anteriormente:
  - se  $(d \leq \text{limite}[s])$  então end-físico = base[s] + d
  - senão "acesso não autorizado" (TRAP)

# Tabela de Segmentos (TS)



- idealmente, TS deve ser armazenada em registradores rápidos ou em vetores associativos que permitam consulta paralela (tipo TLB)
- quando TS é mantida em memória, cada processo tem um ponteiro indicando o início da tabela (base register - STBR) e um contador do número máximo de entradas na tabela (length register- STLR).
- antes de cada acesso é verificado se  $s \leq \text{STLR}$ , e só então são consultados  $\text{limit}[\text{STBR}+ s]$  e  $\text{base}[\text{STBR}+ s]$

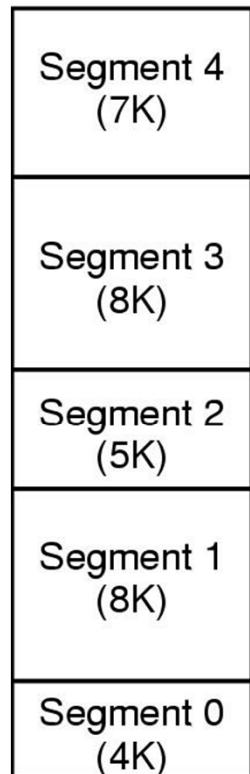
# Implementação de Segmentação



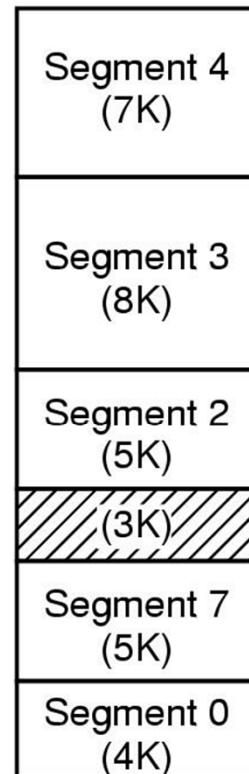
Segmentos podem ser carregados em qualquer espaço livre da memória, logo:

- precisa-se controlar a lista de lacunas disponíveis
- pode causar fragmentação externa (sequencia a-d)

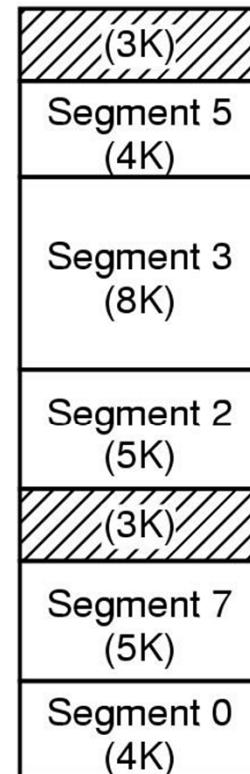
Pode-se fazer compactação de memoria para agrupar segmentos e criar uma lacuna maior.



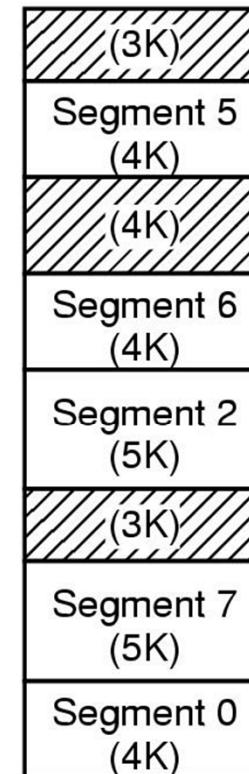
(a)



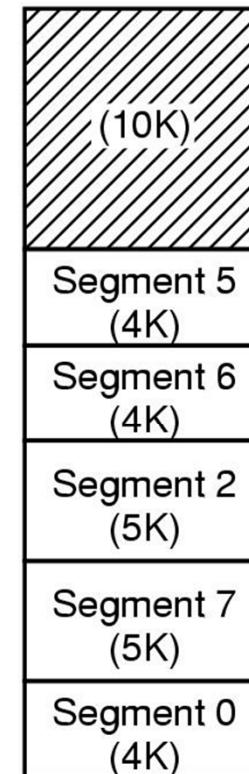
(b)



(c)



(d)



(e)

# Segmentação (3)



Consideração	Paginação	Segmentação
O programador precisa estar ciente de que essa técnica está sendo usada?	Não	Sim
Quantos espaços de endereçamentos lineares existem?	Um	Muitos
O espaço de endereçamento total pode exceder o tamanho da memória física?	Sim	Sim
Os procedimentos e os dados podem ser diferenciados e protegidos separadamente?	Não	Sim
As tabelas com tamanhos variáveis podem ser acomodadas facilmente?	Não	Sim
O compartilhamento de procedimentos entre usuários é facilitado?	Não	Sim
Por que essa técnica foi inventada?	Para fornecer um grande espaço de endereçamento linear sem a necessidade de comprar mais memória física	Para permitir que programas e dados sejam quebrados em espaços de endereçamento logicamente independentes e para auxiliar o compartilhamento e a proteção

## Comparação entre paginação e segmentação



# Segmentação com Paginação

- Principal vantagem: Permite segmentos arbitrariamente grandes
- Cada segmento é um espaço de endereçamento completo, que é paginado da mesma forma como um espaço de endereçamento uni-dimensional
- Somente as partes de cada segmento que estão em uso, precisam estar na memória física
- Cada processo tem sua TabSegmentos, composta de Descritores
- Cada Descritor contém:
  - Ponteiro para endereço da TabPáginas
  - Tamanho do Segmento em páginas
  - Bits de proteção
- Quando um segmento está sendo usado (acessado), sua tabela de páginas precisa estar em memória
- Se o segmento puder ser muito grande ( $2^{32}$  endereços), então a própria tab. de páginas deverá ser paginada
- Endereço Virtual:

s	p	o
---	---	---

s= índice na tab de segmentos  
p= índice na tab de páginas  
o= offset na página

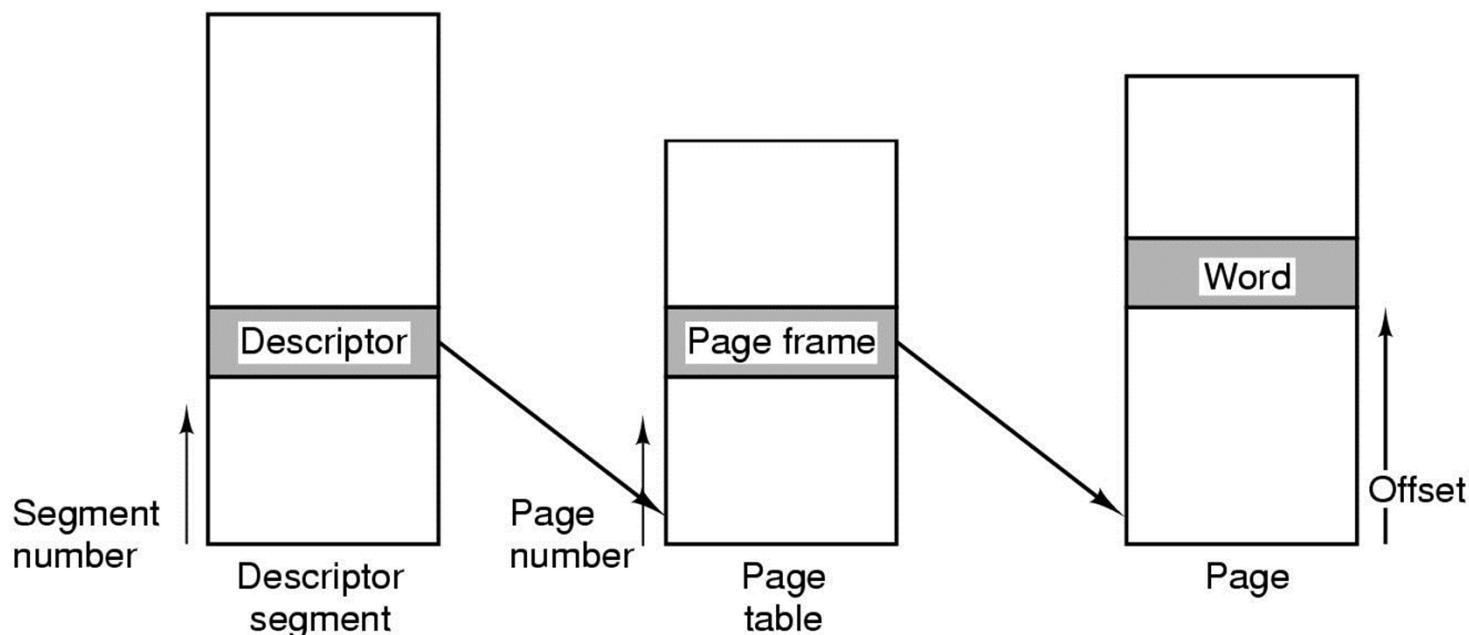
# Segmentação com paginação no MULTICS



MULTICS virtual address



Conversão de um endereço que consiste de 2 partes para um endereço físico de memória.



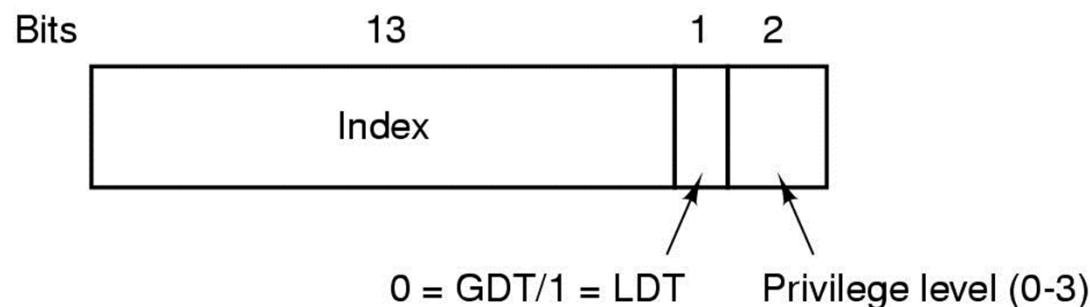
Para agilizar o acesso, MULTICS usa um TLB de 16 palavras, em que um par (s,p) pode ser consultado em paralelo.

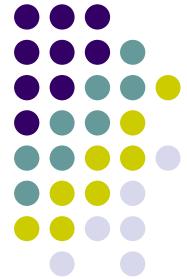


# Segmentação com paginação: Pentium

Memória virtual do Pentium suporta:

- Paginação pura, segmentação pura, ou segmentação paginada
- 16K segmentos independentes cada um com  $2^{32}$  palavras
- 2 tabelas: Local Description Table (uma por processo), e Global Description Table (compartilhada por todos os processos)
- Endereço base (do segmento) é carregado em registrador de CPU: CS (code segment) ou DS (data segment)
- Cada segmento possui um dos 4 níveis de privilégio do Pentium:(kernel, system calls, sh.libraries, user programs)

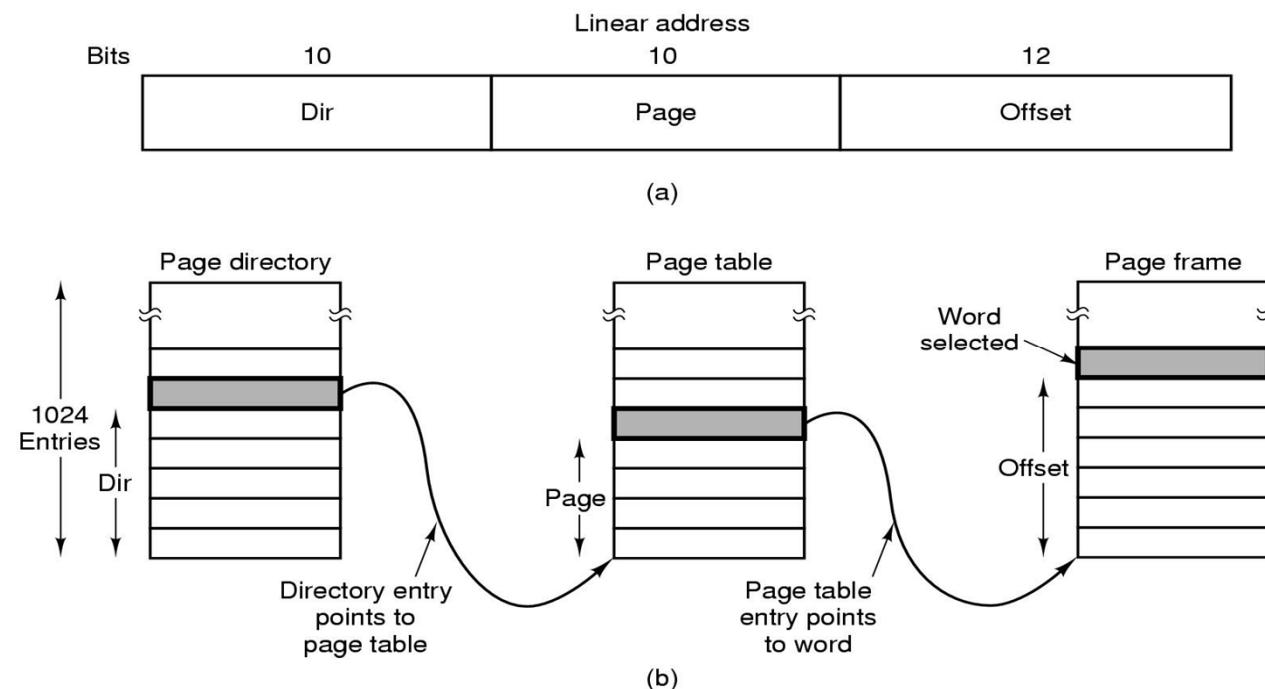




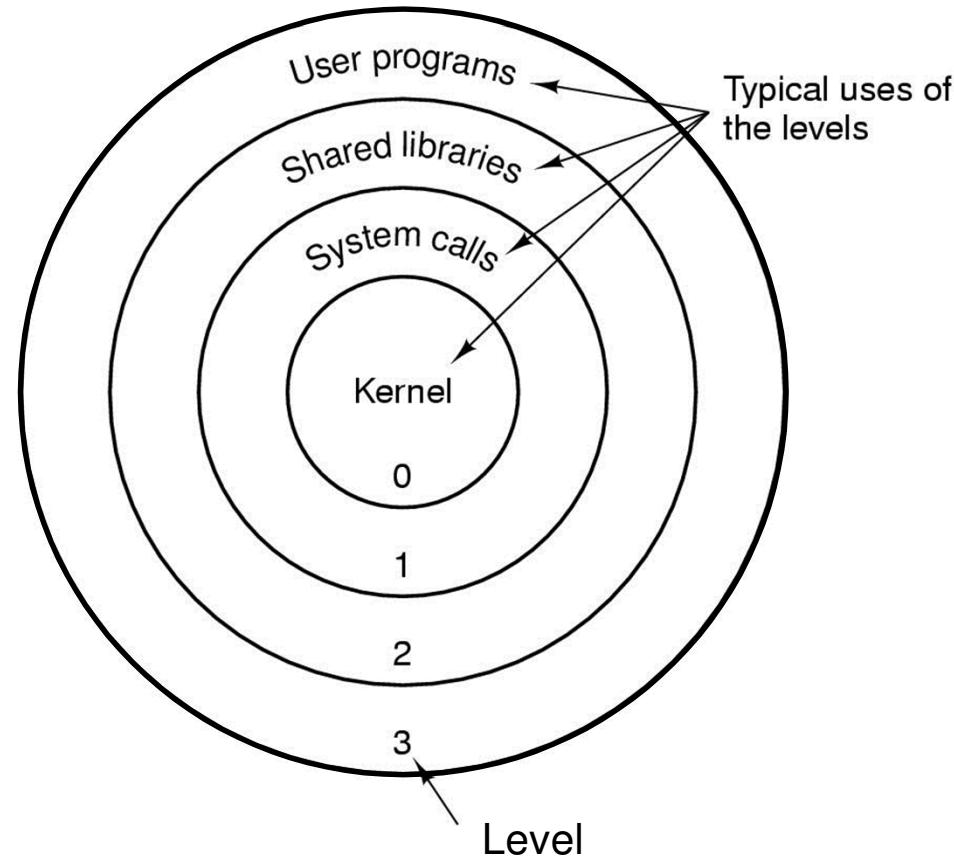
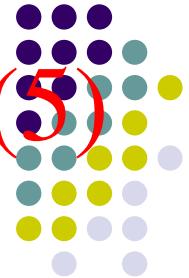
# Paginação em dois níveis no Pentium

Quando usado no modo paginação:

- A tabela de páginas é paginada, e endereço possui dois índices: (dir-index, page-index)
- Pentium possui um pequeno TLB que mantém os pares de índice recentes



# Segmentation with Paging: Pentium (5)



## Protection on the Pentium



# Resumo e Conclusão

Realocação dinâmica de memória é indispensável para multiprogramação;

Cada processo deve ter o seu próprio mapeamento de memória;

Memória Virtual (paginação) é provida na maioria dos S.O. modernos;

Mas necessita de uma MMU (com um TLB), senão torna-se inviável;

A depender o tamanho de página, pode-se precisar paginação em 2 (ou mais) níveis;

Algoritmo de substituição de páginas - implementado no Gerente de Memória - , deve escolher as páginas usadas há mais tempo e não modificadas.

Segmentação permite ao desenvolvedor controlar a gerência de memória.



# Perguntas?

