# Sistemas Operativos: Gestão de Memória

Pedro F. Souto (pfs@fe.up.pt)

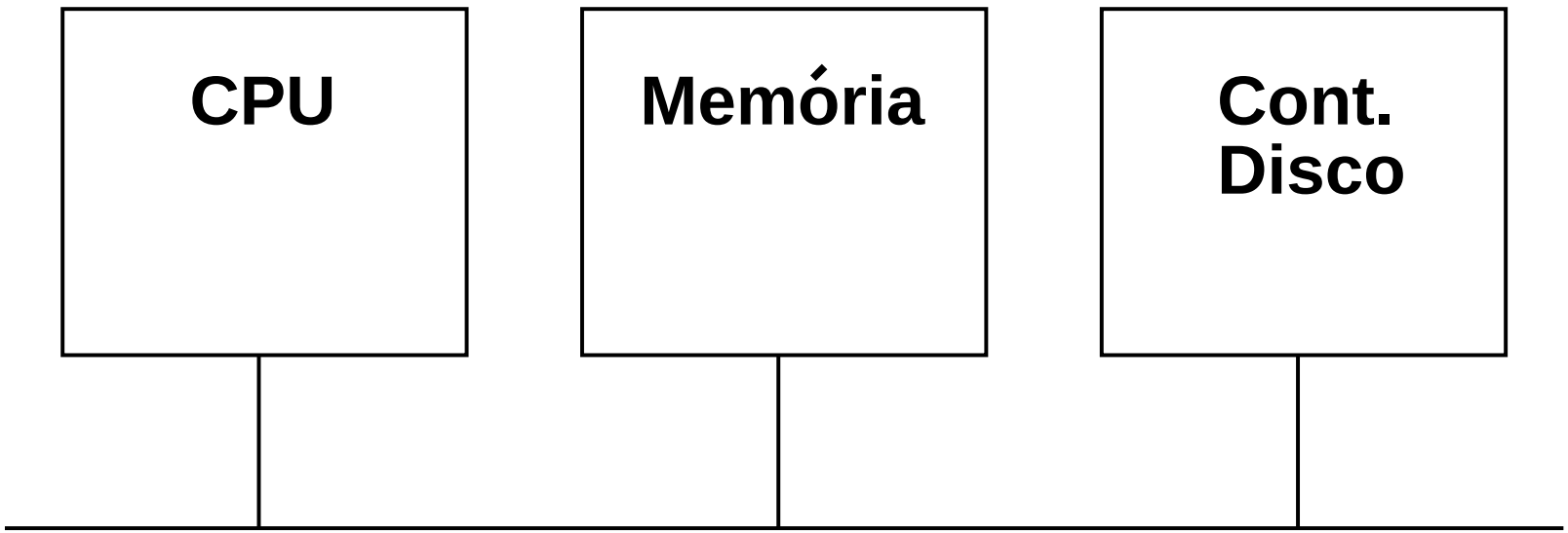
May 17, 2011

# Sumário

# Sumário

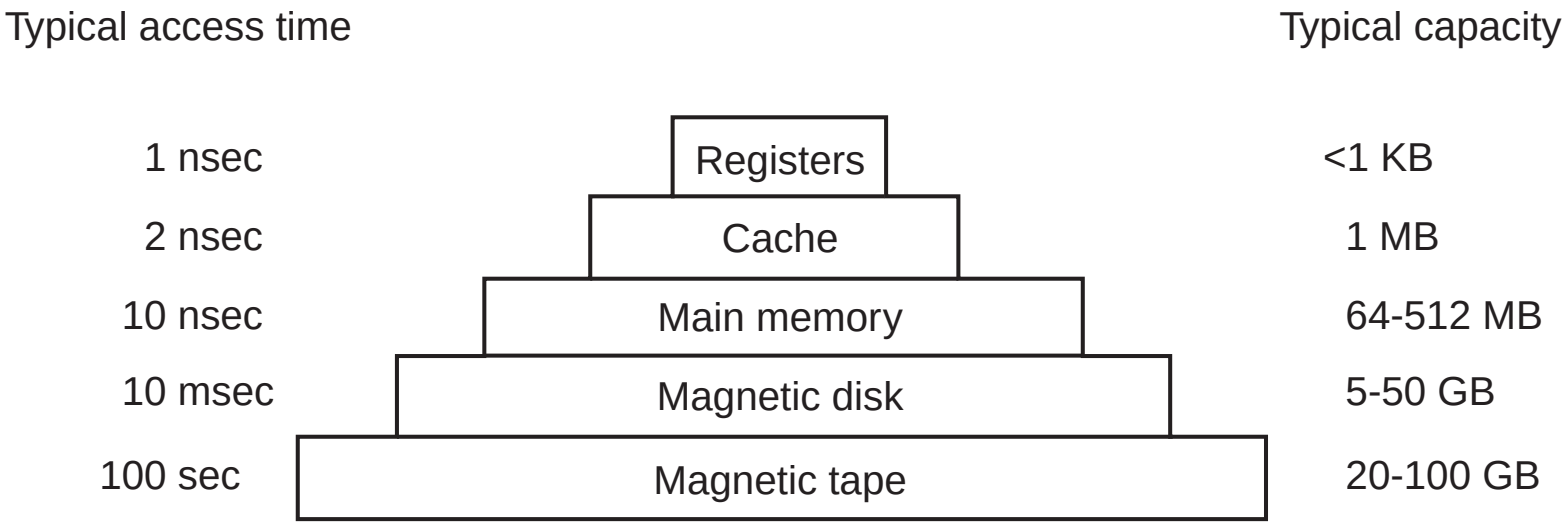
# Gestão de Memória

- Processadores são concebidos para:
  - executar instruções residentes em memória semicondutora;
  - transferir dados de e para memória semicondutora.



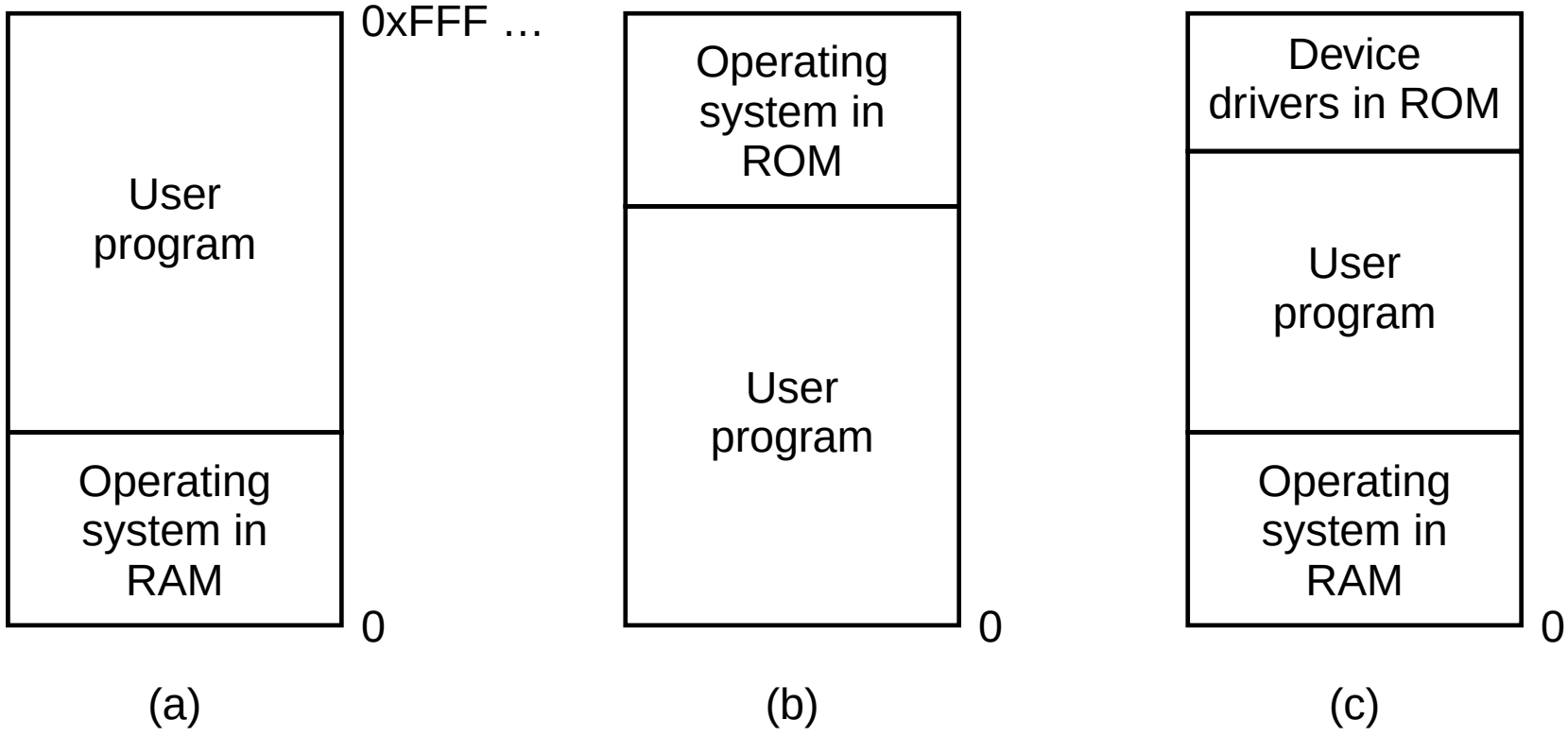
# Hierarquia de Memória

- Praticamente qualquer programador gostaria de dispôr de memória:
  - em grande quantidade;
  - rápida;
  - não volátil.
- SO e os compiladores exploram a hierarquia de memória para satisfazer estes “desejos”:



# Gestão de Memória em Monoprogramação

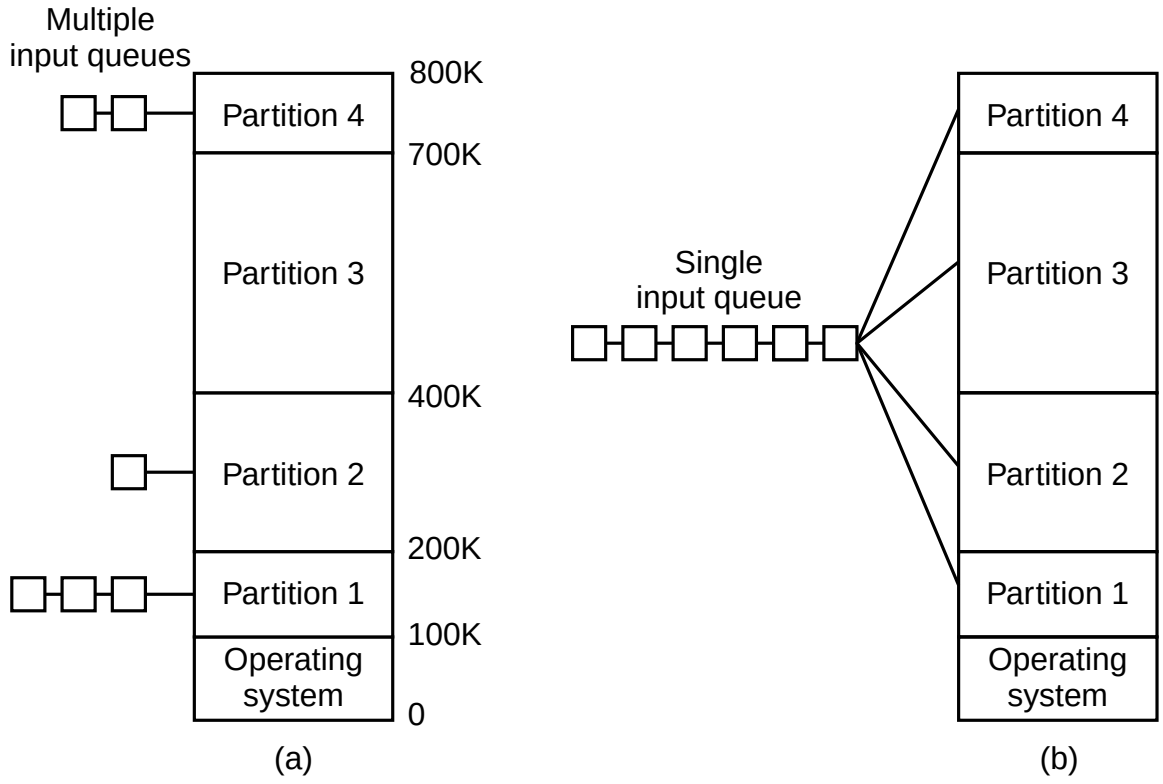
- Executa apenas um processo de cada vez.




b) Disposição típica em *palmtops* ( $\sim$ sistemas embebidos).

c) Disposição em MS-DOS (A parte em ROM é conhecida por BIOS.)

# Multiprogramação e Partições Fixas




- A memória é dividida em partições (possivelmente $\neq$s).

a) o SO atribui a partição de menor tamanho capaz de conter o processo;

b) quando uma partição fica disponível, o SO atribui-a ao processo capaz de a usar mais à frente na fila.

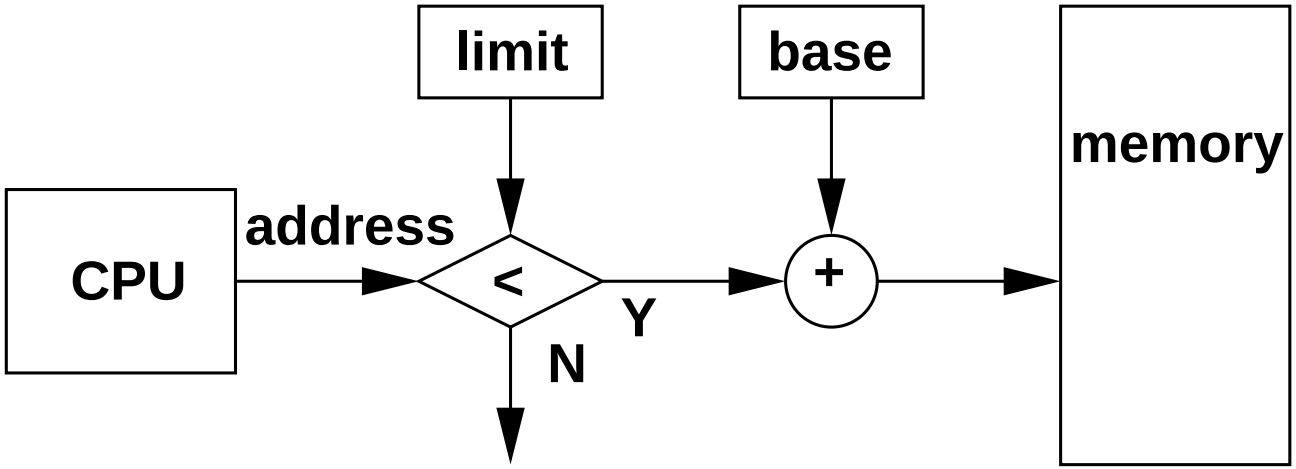
# Problemas Introduzidos por Multiprogramação

**Recolocação:** a posição do código pode variar entre execuções:

- O *loader* pode alterar os endereços absolutos de acordo com a posição onde o código é carregado.
- Usar um *base register* a inicializar com o endereço da partição atribuída ao processo.

**Protecção:** impedir que um processo aceda a código ou a dados de outros processos ou do SO:

- Usar um *limit register* além do *base register*.

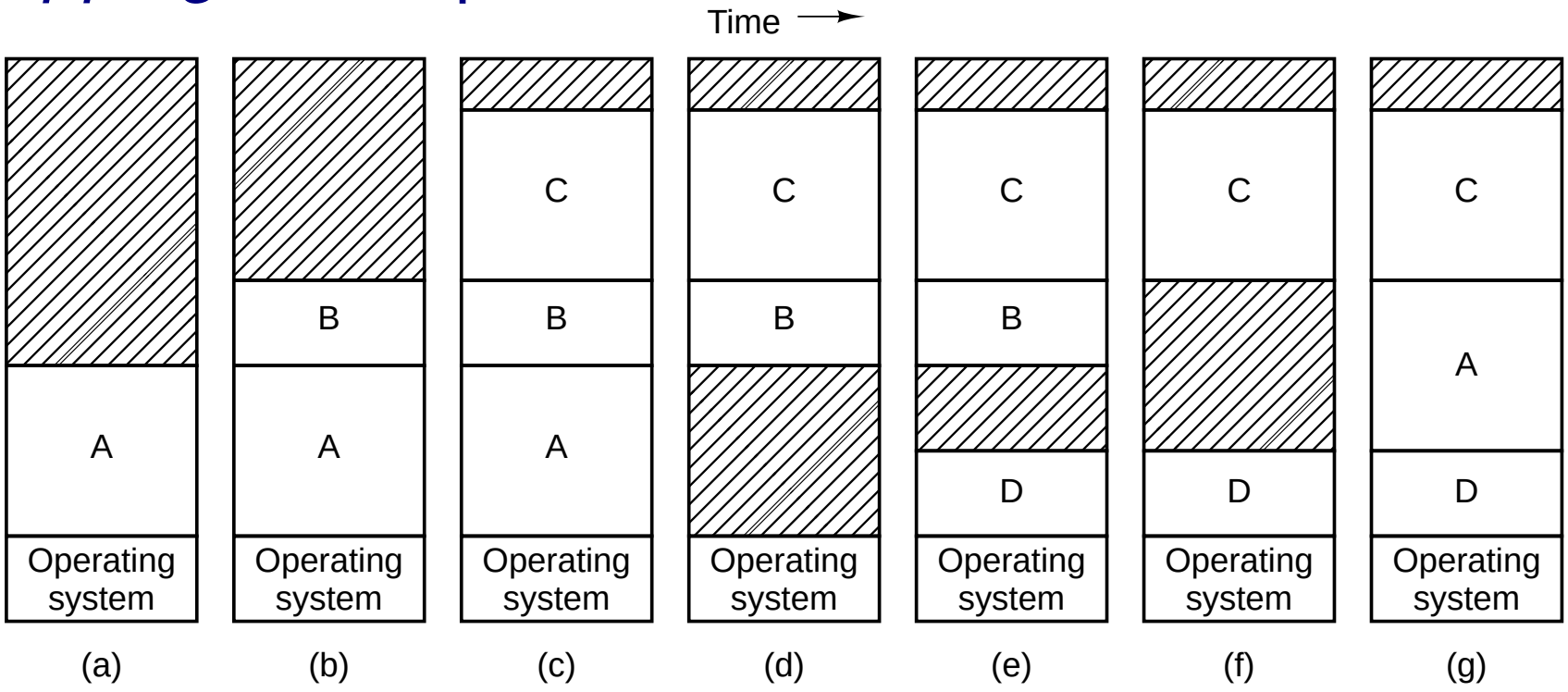


# Sumário

# *Swapping*: Ideia

- Com partições fixas um processo é carregado numa partição quando chega à cabeça da fila correspondente, e depois fica em memória até terminar
- Se as necessidades de memória dos processos forem muito variáveis, a memória física pode tornar-se insuficiente para todos os processos.
- Uma solução possível é o recurso a *swapping*:
  - Passar um processo para o disco (*swap out*) e, posteriormente
  - Transferi-lo de novo do disco para a memória (*swap in*)

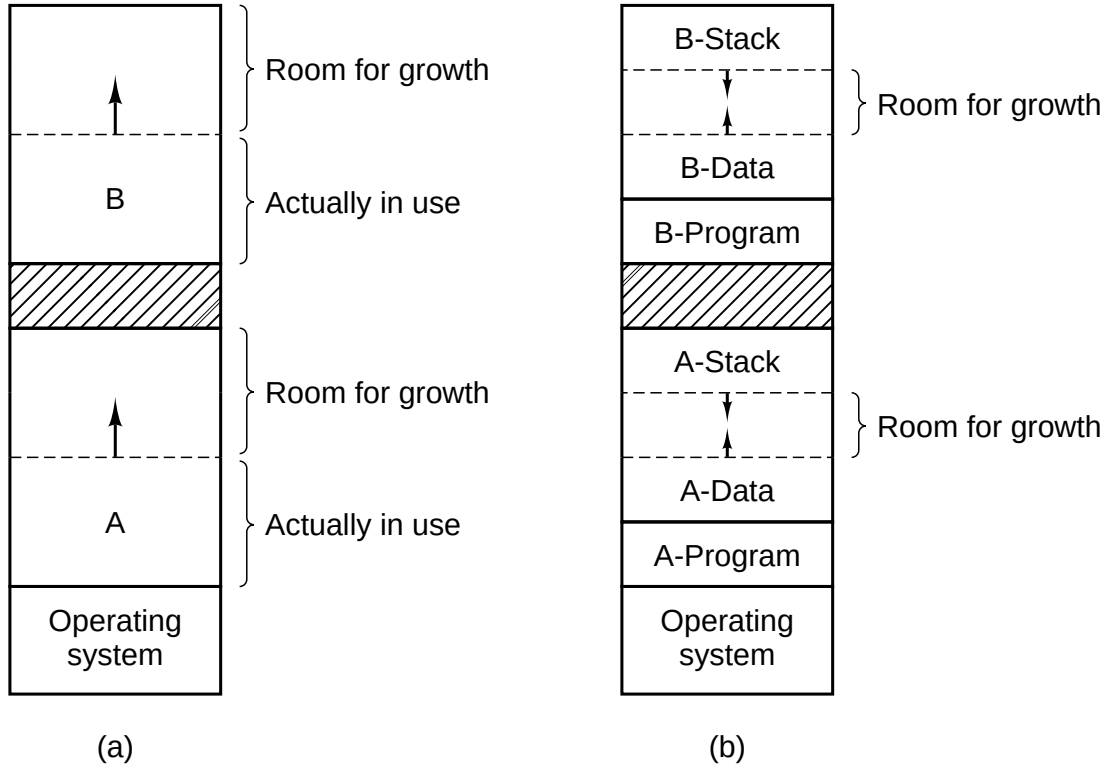
# *Swapping*: Exemplo




- Com *swapping* o número, localização e tamanho das partições varia dinamicamente
  - \+ melhora a utilização da memória;
  - \- torna a gestão de memória mais complexa.
- Contudo *swapping* pode conduzir à fragmentação da memória:
  - pode ser atenuada usando compactação;
  - mas compactação consome bastante CPU

# *Swapping*: Tamanho das Regiões a Alocar

- Qual o tamanho da região de memória a alocar quando um processo é criado ou trazido para memória (*swapped in*)?

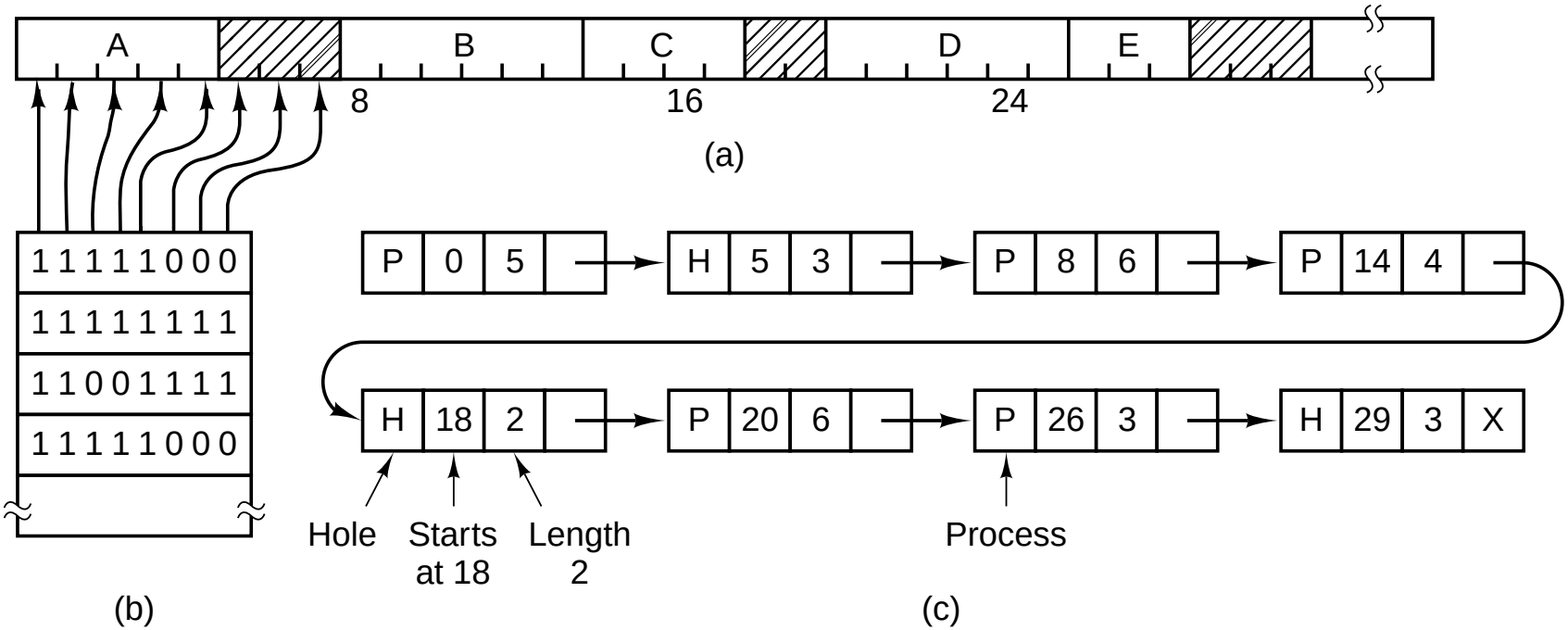



- Se o processo crescer em demasia e não houver mais memória para alocar, pode-se passá-lo para o disco (*swap-out*)
  - Se disco estiver cheio, pode-se bloquear processo ... mas nesse caso o processo ocupará memória

# Sumário

# Alocação Dinâmica de Memória

- O SO tem que manter informação sobre o espaço de memória disponível
- Tipicamente decompõe-se a memória física em unidades de alocação, cujo tamanho é da ordem dos KB
- Normalmente usa-se uma de 2 estruturas de dados
  1. bitmaps;
  2. listas ligadas.

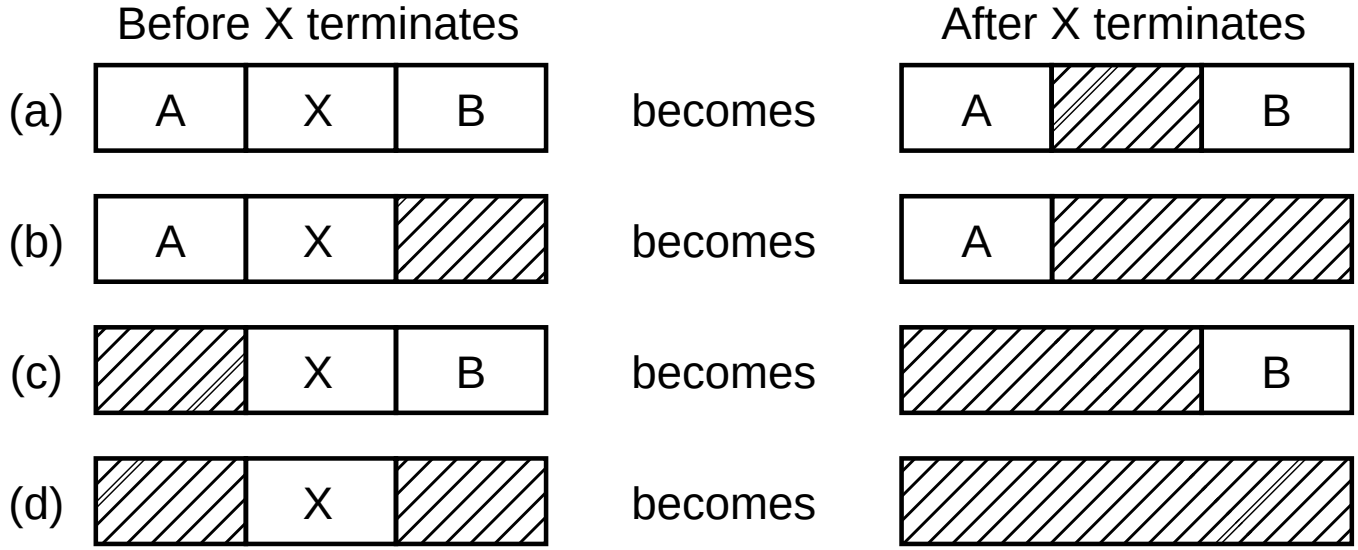


# *Bitmaps vs.* Listas

+ Bitmaps ocupam um espaço de memória fixo que depende do tamanho da unidade de alocação
  - blocos pequenos exigem mais memória para os bitmaps;
  - blocos grandes podem conduzir a fragmentação externa
- Pesquisa de espaço disponível para alocação pode ser lenta

# Algoritmos de Alocação Dinâmica de Memória

- A libertação de memória é relativamente simples:




- A alocação é mais interessante:

  *first fit:* simples e eficiente;
  *next fit:* começa varrimeno onde parou da última vez - de facto pior do que *first fit*;
  *best fit:* tenta evitar fragmentação, mas . . .
  *worst fit:* tenta evitar fragmentos muito pequenos, mas . . .

# Alicação Dinâmica de Memória: Truques

- Manter listas separadas para a memória ocupada e a memória livre
  - \+ facilita alocação;
  - \+ pode-se usar a própria memória livre para implementar os elementos da lista de memória livre;
  - \- penaliza a libertação da memória.
- Manter listas de memória livre, uma para cada um dos tamanhos de blocos mais comuns (*quick fit*)
  - \- problemas análogos aos do uso de listas separadas

# Sumário

# Memória Virtual (MV)

- **Ideia**:
  - Decompôr o espaço de endereçamento dum processo em blocos.
  - Manter em memória apenas alguns desses blocos (código e dados) do processo em execução.
  - Manter os outros blocos no disco:
    - na área de *swap*;
    - no sistema de ficheiros (código, p.ex.).
  - Transferir os blocos entre o disco e a memória, conforme necessário.
- **Mecanismos**
  - Paginação;
  - Segmentação;

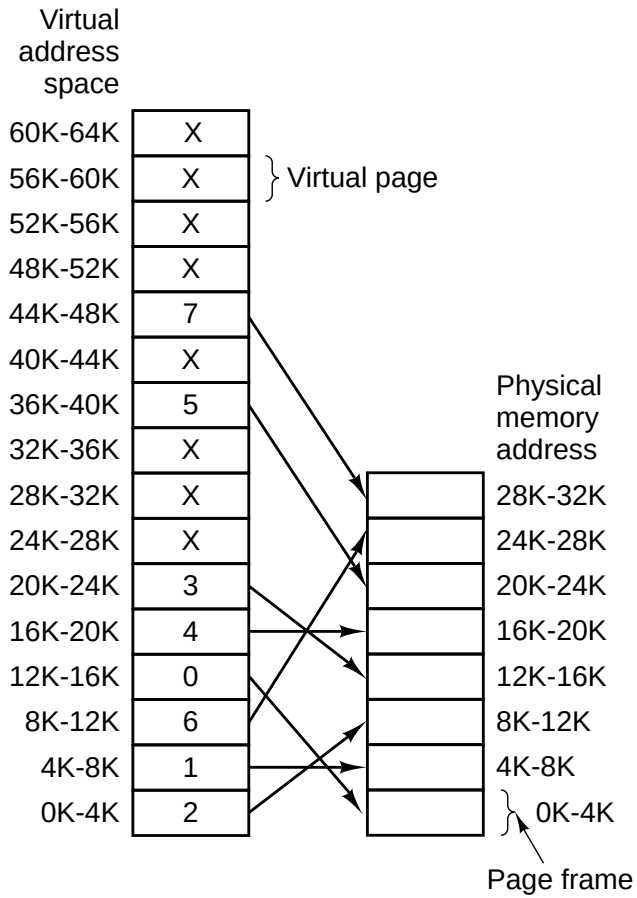
  (ambos requerem suporte do HW).

# MV *vs. swapping*

+ Permite programas maiores do que a memória física disponível;
+ Possibilita ter mais processos em memória (ainda que só parte deles);
+ Reduz tempo de arranque dum processo
- Exige suporte do HW mais sofisticado

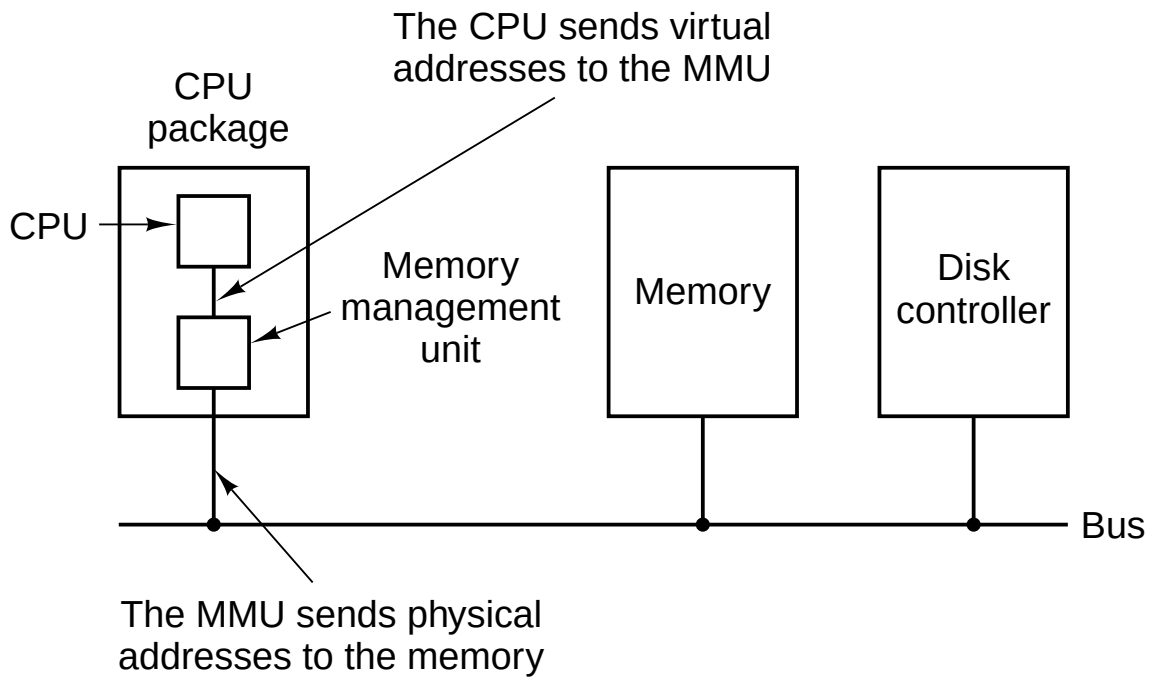
# MV Paginada

- A memória física está dividida em *page frames*, ou *frames*:
  - o tamanho das *frames* é determinado pelo HW (4 Kbyte ou 8 Kbyte).
- O espaço de endereçamento está dividido em *páginas*:
  - tipicamente de tamanho igual ao das *frames*.
- O conteúdo das páginas é transferido de e para *frames*.



# Conversão de Endereços em MV Paginada

- Paginação determina dois espaços de endereçamento:
  - virtual ou **lógico** que é o que o CPU conhece - o código executável usa endereços virtuais ou lógicos;
  - físico que é usado no barramento de endereços da memória.
- O mapeamento dos endereços dum espaço no outro é realizado pela *Memory Management Unit (MMU)*:

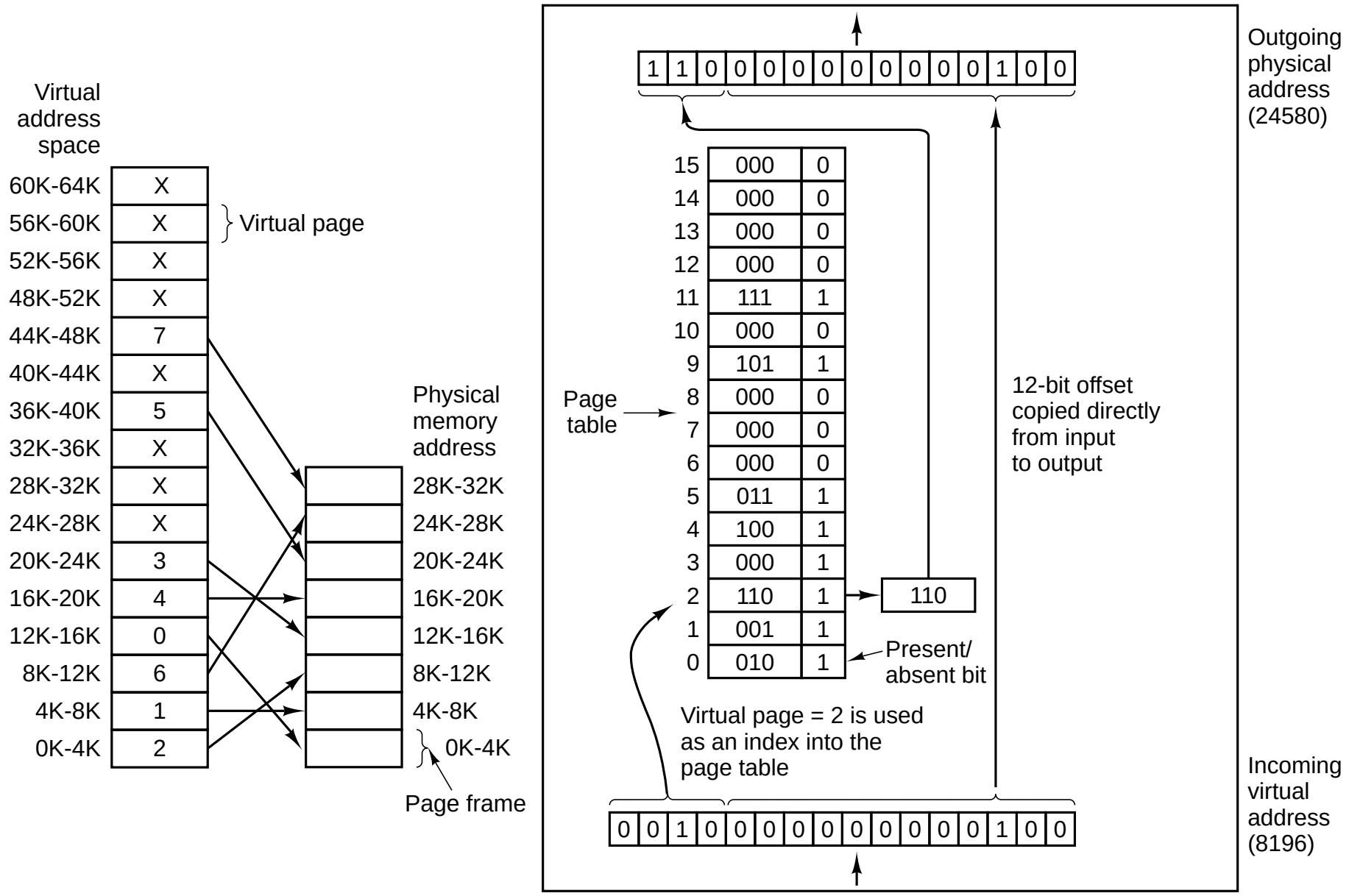


# Conversão de Endereços com uma Tabela (1/2)

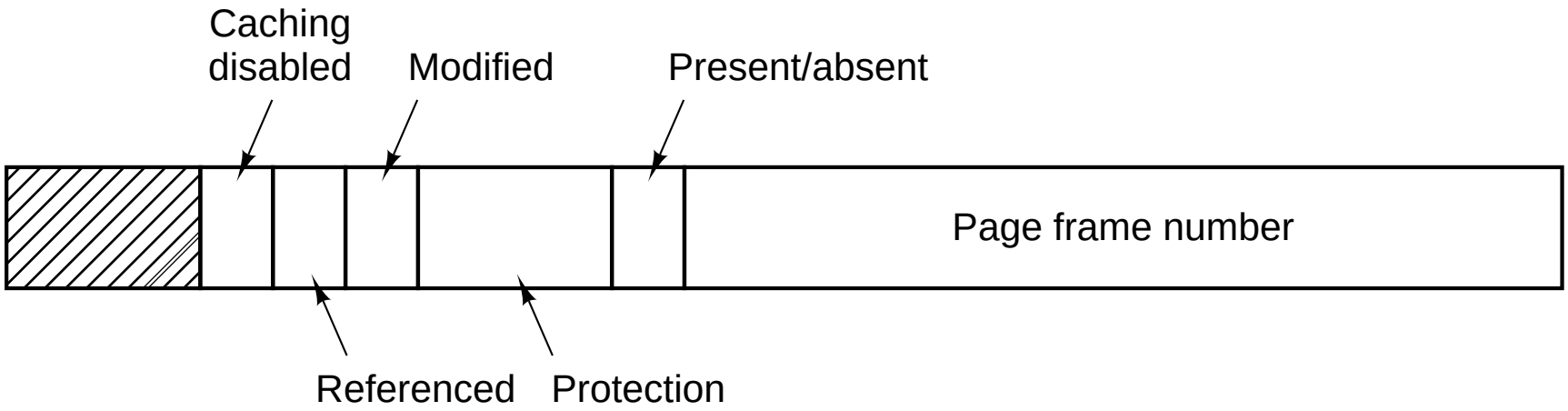
- Em sistemas com MV paginada um endereço tem 2 componentes:


  - O número da página, parte mais significativa do endereço;
  - O deslocamento (*offset*) na página, parte menos significativa.
- Como *frames* e páginas têm o mesmo tamanho, a MMU só tem que mapear o número da página no número da *frame*.
- O mapeamento pode ser feito usando uma tabela, implementada como um vector:
  - o número da página é usado como indíce para obter o número da *frame* que a contém.

# Conversão de Endereços com uma Tabela (2/2)



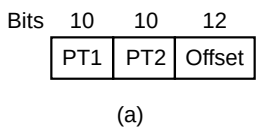
# Page Table Entry

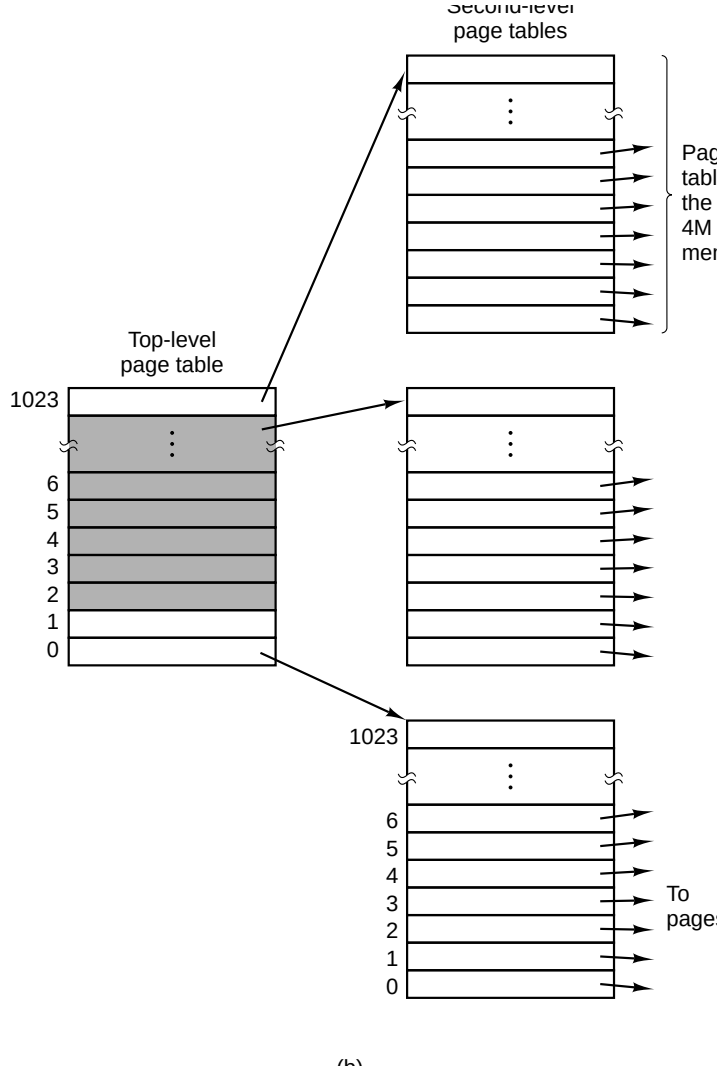



- Se o CPU usar endereços de 32 bits, e páginas de 4 Kbyte (12 bits), a *page table* terá $\sim$4 Mbyte!!!
  - Cada página pode conter até 1024 ($2^{10}$) elementos
  - Sendo necessárias 1024 ($2^{20}/2^{10}$) páginas
- Normalmente, um processo usa apenas uma fracção do seu espaço de endereçamento:
  - A maioria dos elementos da *page table* não são usados
  - Muitas das páginas da *page table* estão vazias

# Tabelas com 2 Níveis

- Permite não usar páginas da tabela vazias
- Exige uma “tabela” adicional para as páginas
  - Elementos correspondentes a páginas vazias tem o bit *Present/absent* a 0
  - Os outros elementos apontam para páginas com conversões
- A conversão é feita em 2 passos:
  - Os 10 bits mais significativos são usados para seleccionar a página de 2º nível que poderá conter o elemento desejado
  - Os 10 bits seguintes são usados como indíce da “tabela” contida nessa página

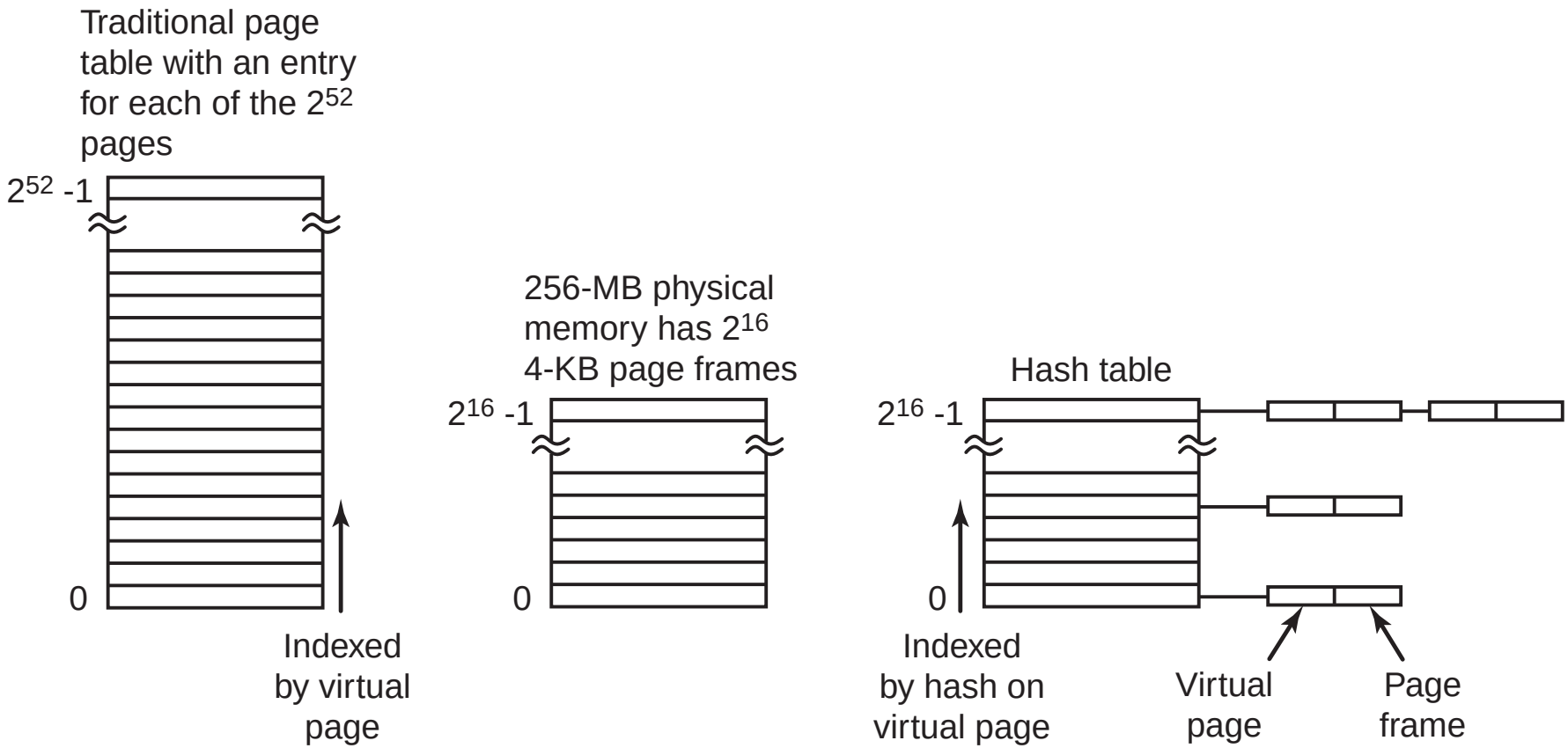



- Este esquema pode ser generalizado a 3 ou mais níveis

# Tabelas Invertidas

- Alguns CPUs de 64 bits usam tabelas invertidas:
  - têm um elemento por *frame*, em vez de ...
  - cada elemento contêm informação sobre (processo, página virtual) contido pela *frame*
- A tabela está organizada como uma *hash table*
  - usa o endereço virtual da página como *key*



# Desempenho da Conversão de Endereços

- Pode traduzir-se um endereço com tantos acessos à memória quantos os níveis da *page table*
  - no caso de tabelas invertidas são necessários 2 acessos no mínimo, mais se houver colisões.
- Obviamente, temos um problema:
  - cada acesso à memória, requereria acessos adicionais para converter endereços.
- A solução é usar uma *cache*, a *Table Lookaside Buffer*, de elementos da *page table* usados em conversões recentes:

| Valid | Virtual Page | Modified | Protection | Page Frame |
|---|---|---|---|---|
| 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 |

# *Table Lookaside Buffer (TLB)*

- A TLB usa memória associativa:
  - cada posição de memória consiste num par *(key, value)*;
  - o endereçamento é feito comparando o valor do *endereço* com o valor de *key*.

  Em termos de implementação, a TLB é semelhante à *cache* de instruções ou dados.
- O número de posições da TLB não precisa ser muito elevado:
  - por causa da *localidade* no acesso à memória.
- Quando da comutação de processos, o conteúdo da TLB tem que ser invalidado, a menos que ...

# *TLB Misses*

- E se o endereço a converter não estiver na *TLB* (*TLB miss*)?
  - Há que consultar a *page table*
- Normalmente, o HW de gestão de memória, a *Memory Management Unit* (MMU), processa automaticamente as *TLB misses*:
  - A MMU precisa conhecer a localização da *page table* do processo em execução.
  - A MMU de alguns CPUs RISC não processam *TLB misses*, deixam essa tarefa a cargo do SO.

# Desempenho da Conversão de Endereços com TLB

- Seja:
  $p$ a TLB *hit ratio*
  $m$ o nº de acessos à memória no caso duma *TLB miss*
- Admitindo que no caso de um *TLB hit*, o custo é zero, o custo efectivo da utilização de MV paginada vem:
  $$(1 - p)m$$
- Seja:
  $p = 0.98$
  $m = 3$
  Então o custo será de: 0.06 acessos à memoria por conversão de endereço

# *Page Fault*

Quando um processo tenta aceder a uma página e essa página não está em memória ocorre uma *page fault*

- A MMU gera uma excepção, que é processada pelo *page fault handler*:
  - O *page fault handler* consulta uma outra estrutura de dados (*address map* do processo) para determinar:
    - se a referência é válida: **protecção**;
    - e, em caso afirmativo, onde se encontra a página.
  - O *handler* obtém uma *frame* livre.
  - Transfere a página do disco para a *frame*.
  - Actualiza a *page table*.
- O CPU reinicia a execução da instrução que causou a *page fault*:
  - mais fácil dizer do que fazer, mas tem a ver com o HW.

# *Address Map*

- É a estrutura de dados do SO que mapeia espaço no disco:
  - ficheiros;
  - partes da área de *swap*

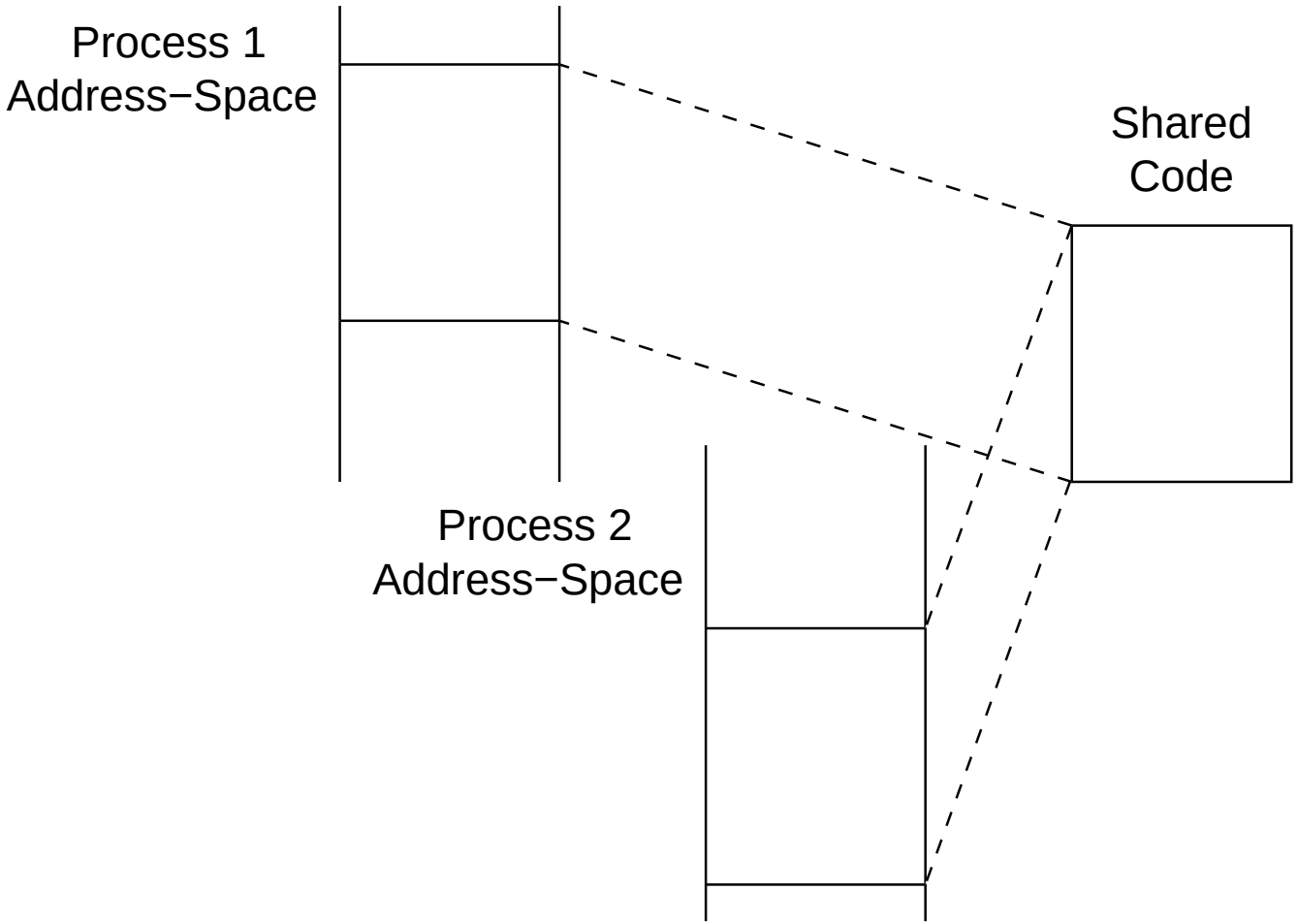
  no espaço de endereçamento dum processo.
- É usada pelo *page-fault handler* para determinar:
  - a validade da referência;
  - em caso positivo, qual a *fonte* da página referenciada.
- Quando o SO cria um processo:
  - mapeia o ficheiro com o código correspondente no espaço de endereçamento desse processo;
    - reduz o tempo de arranque dum processo.
  - transfere algumas páginas desse ficheiro para a memória (*prepaging*):
    - evita uma *page fault rate* excessiva inicialmente.

# more /proc/1/maps

```
08048000-0804f000 r-xp 00000000 03:02 8954 /sbin/init
0804f000-08050000 rw-p 00006000 03:02 8954 /sbin/init
08050000-08054000 rwxp 00000000 00:00 0
40000000-40012000 r-xp 00000000 03:02 8862 /lib/ld-2.1.3.so
40012000-40013000 rw-p 00011000 03:02 8862 /lib/ld-2.1.3.so
40013000-40014000 rwxp 00000000 00:00 0
4001b000-400f0000 r-xp 00000000 03:02 8864 /lib/libc-2.1.3.so
400f0000-400f4000 rw-p 000d4000 03:02 8864 /lib/libc-2.1.3.so
400f4000-400f8000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

# Partilha de Código

- O *address map* facilita a partilha de de memória entre processos.
  - Um exemplo muito comum é o código de bibliotecas partilhadas



# E se não houver qualquer *frame* livre?

- Se, quando duma *page-fault*, todas as *frames* estiverem ocupadas, o SO tem que libertar uma:
  - se a *frame* seleccionada tiver sido modificada, terá que ser transferida para o disco.
- Por razões de eficiência, o SO tipicamente executa um processo (*pageout daemon*) que procura manter um certo número de *frames* livres.
- Em qualquer dos casos, põe-se o problema:

  *Que páginas transferir para o disco?*

  É bom que se faça uma boa escolha, doutro modo o desempenho sofre ...
- Note-se que este problema é comum a qualquer tipo de *cache*.

# Algoritmos de Substituição de Páginas

| Algorithm | Comment |
|---|---|
| Optimal | Not implementable, but useful as a benchmark |
| NRU (Not Recently Used) | Very crude |
| FIFO (First-In, First-Out) | Might throw out important pages |
| Second chance | Big improvement over FIFO |
| Clock | Realistic |
| LRU (Least Recently Used) | Excellent, but difficult to implement exactly |
| NFU (Not Frequently Used) | Fairly crude approximation to LRU |
| Aging | Efficient algorithm that approximates LRU well |
| Working set | Somewhat expensive to implement |
| WSClock | Good efficient algorithm |

# Algoritmo Óptimo

- Quando da substituição duma página, expulsa-se a página que será referenciada no futuro depois de todas as outras.

  Problema O SO não consegue prever o futuro
- A **utilidade** deste algoritmo é que pode ser usado para comparar a qualidade de algoritmos que são exequíveis:
  - requer o registo das operações à memória por um processo;
  - note-se que este registo descreve o acesso à memória do programa em causa, para os dados de entrada usados
    - convém que seja representativo

Solução Usar o passado recente para prever o futuro

- Os diferentes algoritmos diferem nos pormenores

# Algoritmo *Not Recently Used (NRU)*

- Cada *page table entry (PTE)* contém 2 bits:

  R-bit referecend bit
  M-bit modified

  actualizados em cada acesso à memória pela MMU.
- O **R-bit** é limpo periodicamente
- As páginas são classificadas em:
  1. não referenciadas, não modificadas
  2. não referenciadas, mas modificadas
  3. referenciadas, não modificadas
  4. referenciadas, modificadas
- NRU selecciona a classe não vazia pela ordem indicada e expulsa uma página desta classe
- NRU é intuitivo, de implementação eficiente e desempenho aceitável para muitas aplicações

# Algoritmo *Least Recently Used (NRU)*

Ideia Assumir que páginas usadas mais recentemente serão usadas em breve

- expulsar a página menos usada recentemente

- Normalmente o HW não fornece os mecanismos necessários para uma implementação eficiente deste algoritmo
- A sua implementação em SW não é viável na prática
  - Porquê?
- A alternativa é tentar aproximar o LRU usando outros algoritmos

# Algoritmo *Not Frequently Used (NFU)*

Ideia Associar a cada página um contador que é incrementado periodicamente, de acordo com o estado do **R-bit**

- expulsar a página cujo contador tem o valor mais baixo

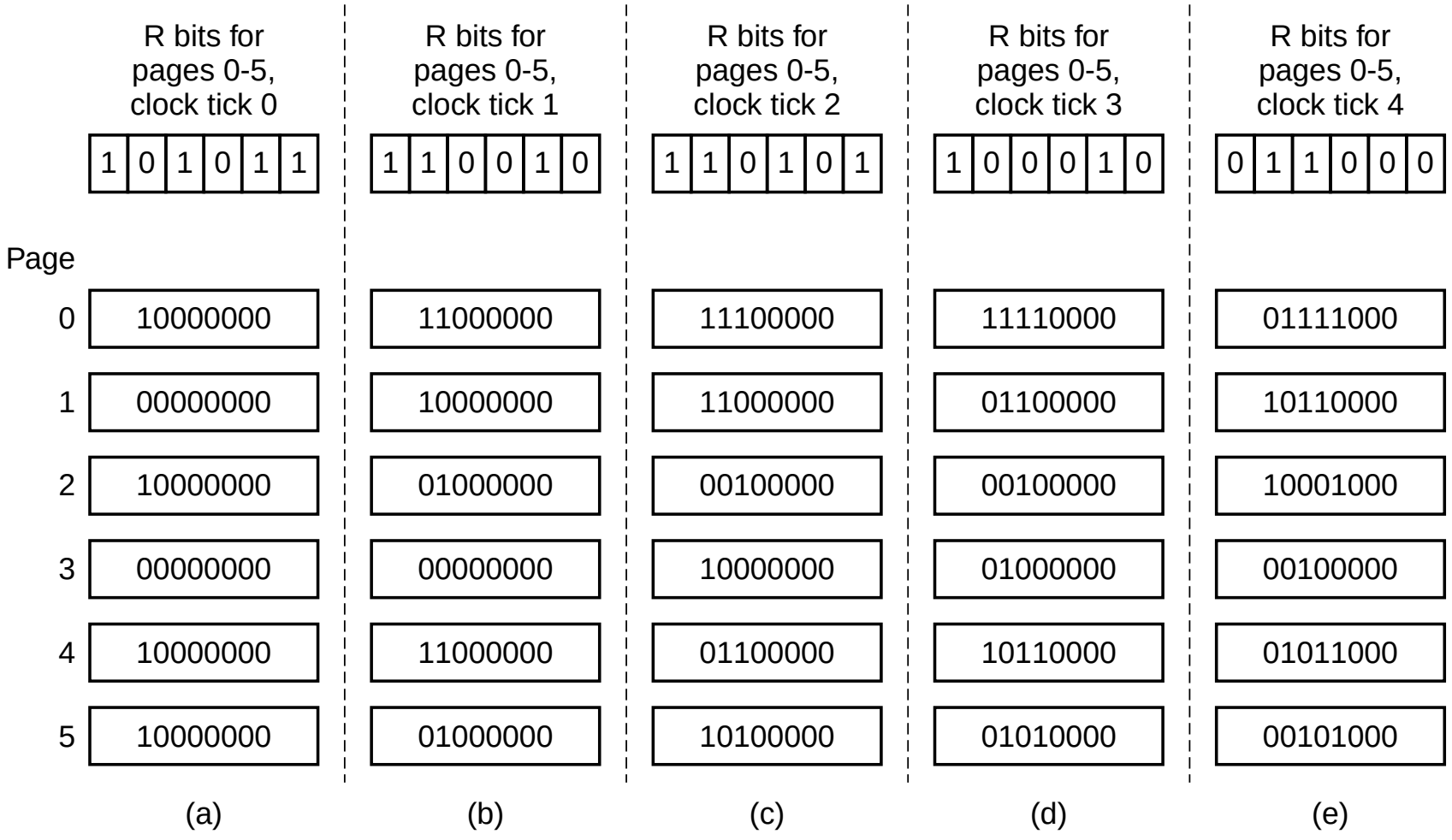
Problema Este algoritmo nunca esquece

- acessos num passado remoto têm o mesmo peso que acessos recentes

Solução Usar *aging*:

- periodicamente, deslocar o valor do contador 1 bit para a direita e inserir o R-bit como MSB

# Algoritmo *NFU with Aging:* Exemplo

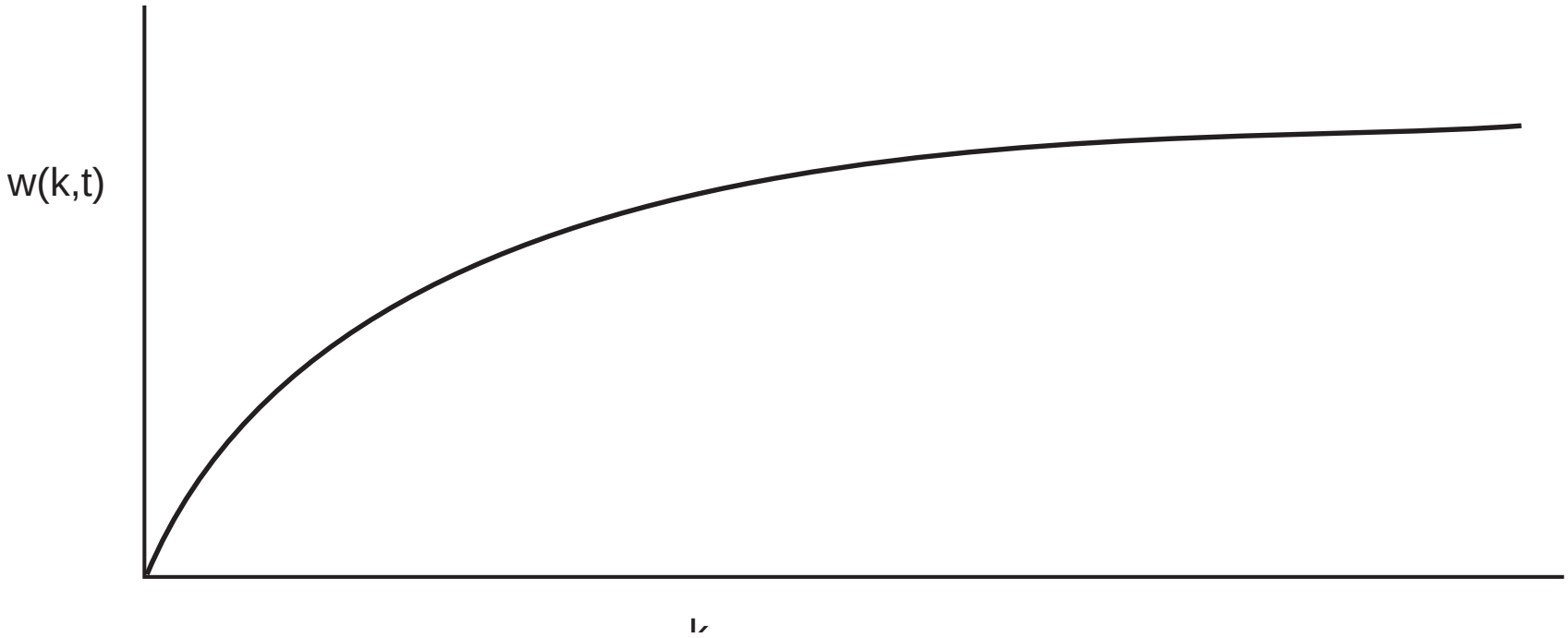


# Conceito de *Working Set*

- Com paginação pura, as páginas dum processo só são trazidas para a memória quando o processo tem uma *page-fault*.
- Quando um processo inicia, tem uma taxa de *page-faults* elevada, até que eventualmente essa taxa diminui:
  - este comportamento deve-se à *localidade de referência*.
- O conjunto de páginas acedidas por um processo num determinado intervalo designa-se por *working set*.
- Se o *working set* dum processo estiver em memória, um processo tem um taxa de *page-faults* muito baixa, reciprocamente ...
- Tipicamente, o *working set* dum processo varia no tempo.

# Definição de *Working Set*

Working set conjunto de páginas acedidas nos $k$ acessos à memória mais receentes




- $w(k, t)$ é o tamanho do *working set* no instante $t$
  - $w(k, t)$ é uma função monótona crescente
  - Contudo $w(k, t)$ não pode nunca ser superior ao nº de páginas do espaço de endereçamento do processo
  - Excepto para valores de $k$ muito baixos, o tamanho do *working set* dum processo é praticamente independente de $k$

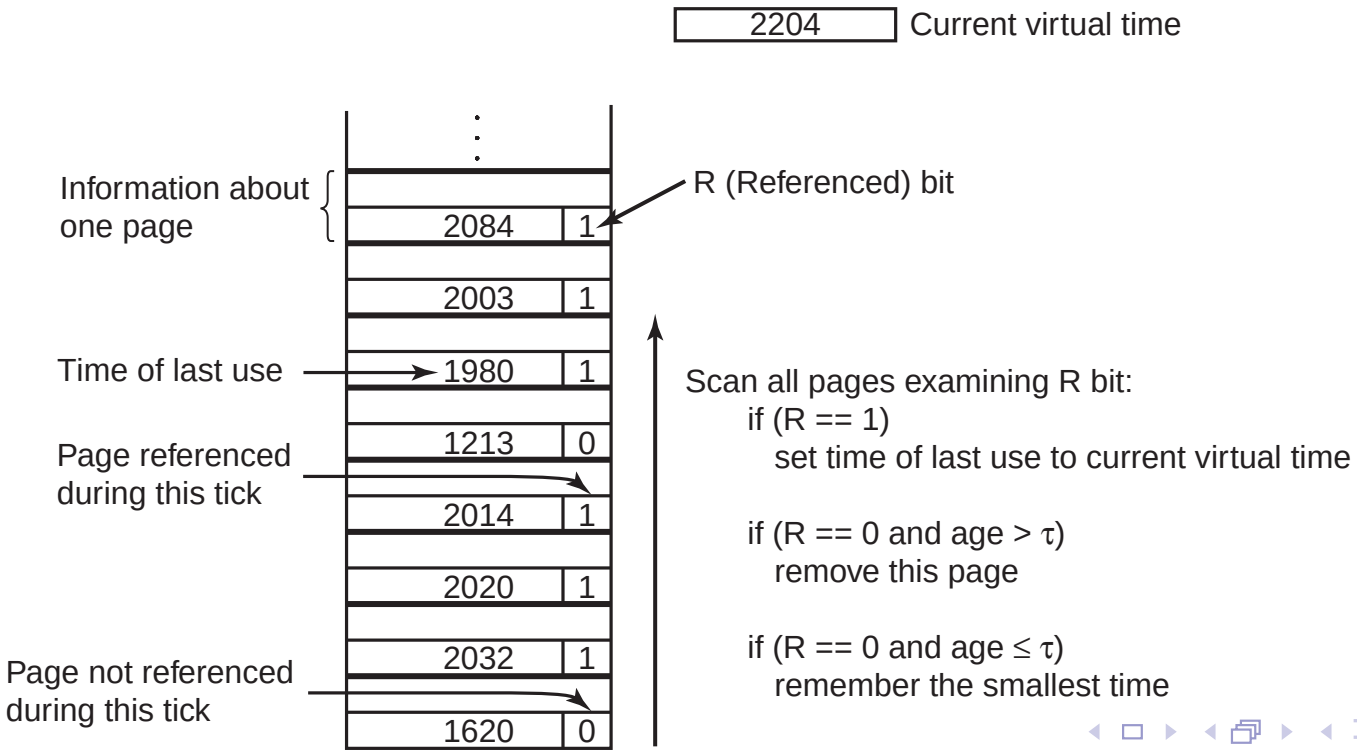
# Algoritmo Baseado no *Working Set*

**Ideia** Expulsar as páginas que não fazem parte do *working set*

**Problema** Como determinar se uma página faz parte do WS?

**Solução** Manter um tempo virtual que é aproximadamente o tempo de execução do processo.

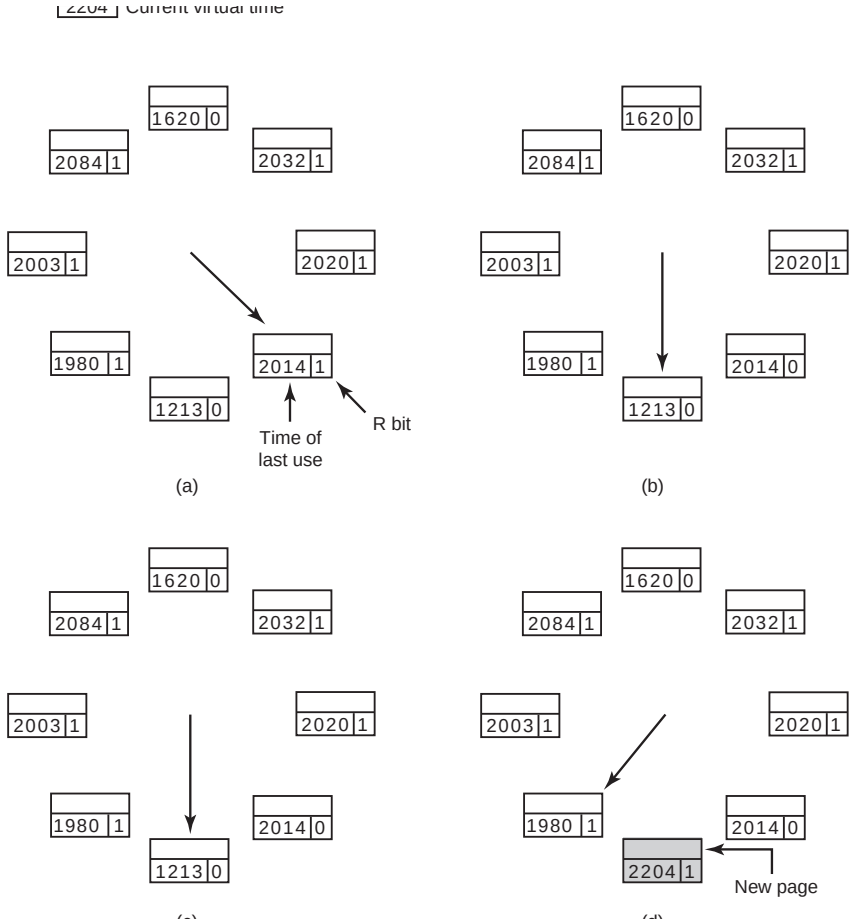
- Periodicamente, actualizar o tempo de acesso de acordo com o R-bit
- Expulsar apenas páginas que não são acedidas há muito tempo



# Algoritmo *WSClock*

- O algoritmo anterior é pouco eficiente pois exige um varrimento de toda a tabela de páginas dum processo sempre que ocorre uma *page fault*

- O tempo do último acesso é mantido como no algoritmo do WS
- Quando ocorre uma *page-fault* o *handler* analisa a página apontada pelo ponterio.
- O algoritmo só expulsa páginas cuja idade é superior a $\tau$

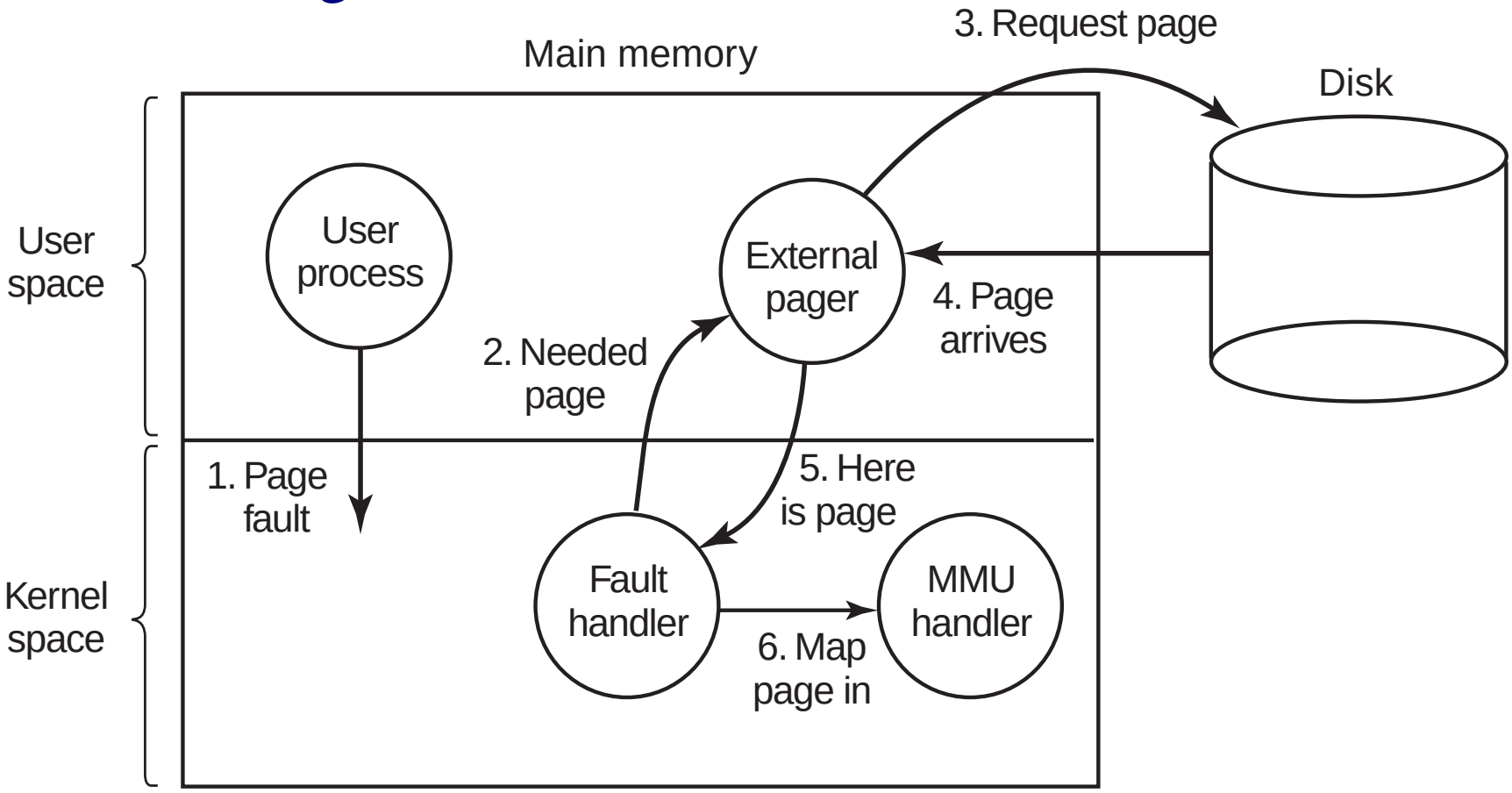


# Algoritmos de Substituição de Páginas

| Algorithm | Comment |
|---|---|
| Optimal | Not implementable, but useful as a benchmark |
| NRU (Not Recently Used) | Very crude |
| FIFO (First-In, First-Out) | Might throw out important pages |
| Second chance | Big improvement over FIFO |
| Clock | Realistic |
| LRU (Least Recently Used) | Excellent, but difficult to implement exactly |
| NFU (Not Frequently Used) | Fairly crude approximation to LRU |
| Aging | Efficient algorithm that approximates LRU well |
| Working set | Somewhat expensive to implement |
| WSClock | Good efficient algorithm |

- O que é que o “utilizador”/programador pode fazer?
  - Pouco a menos que o SO suporte *user-level pagers*

# User-level Pager




- Mais um exemplo de separação entre mecanismos e políticas
- Idealmente, o SO deveria suportar uma API que permitisse expulsar páginas da memória
  - O algoritmo de substituição precisa tipicamente de aceder aos bits R E M.

# Anomalia de *Belady*

- Intuitivamente:
  - quanto maior o número de *page frames*, i.e. memória, menor será a taxa de *page-faults*
- Porém Belady mostrou que nem todos os algoritmos de substituição de páginas garantem que assim é.
  - Alguns algoritmos apresentam casos patológicos nos quais o aumento da memória conduz a mais *page-faults*
- Uma classe de algoritmos de substituição que não sofre deste problema é conhecida por *stack algorithms*
  - Estes algoritmos garantem que:

$$M(n, r) \subseteq M(n+1, r)$$

onde $M(f, r)$ dsigna o conteúdo da memória com $f$ *frames* após $r$ referências

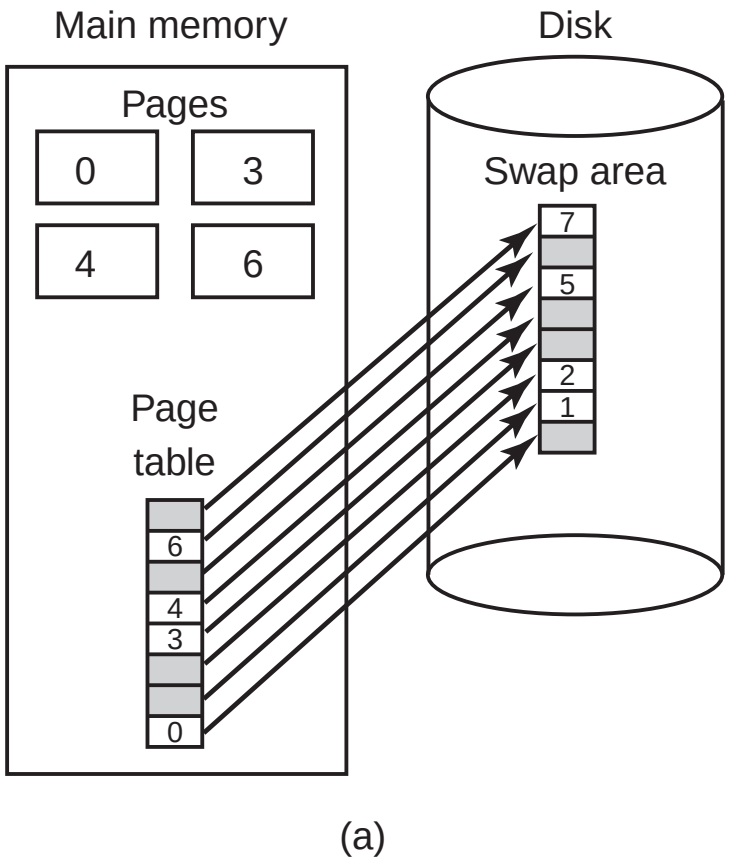
# *Swap Area*

- É uma zona do disco para onde é transferido o conteúdo das páginas modificadas expulsas da memória:
  - páginas de ficheiros não alteráveis, p.ex. código, não precisam ser transferidas para a *swap area*.
- Pode ser implementada:
  - directamente sobre o disco – mais eficiente;
  - sobre ficheiros – fácil de ajustar o seu tamanho.
- Em qualquer caso, o SO tem que gerir o espaço disponível na *swap area*.
- A alocação de espaço na àrea de *swap* pode ser:

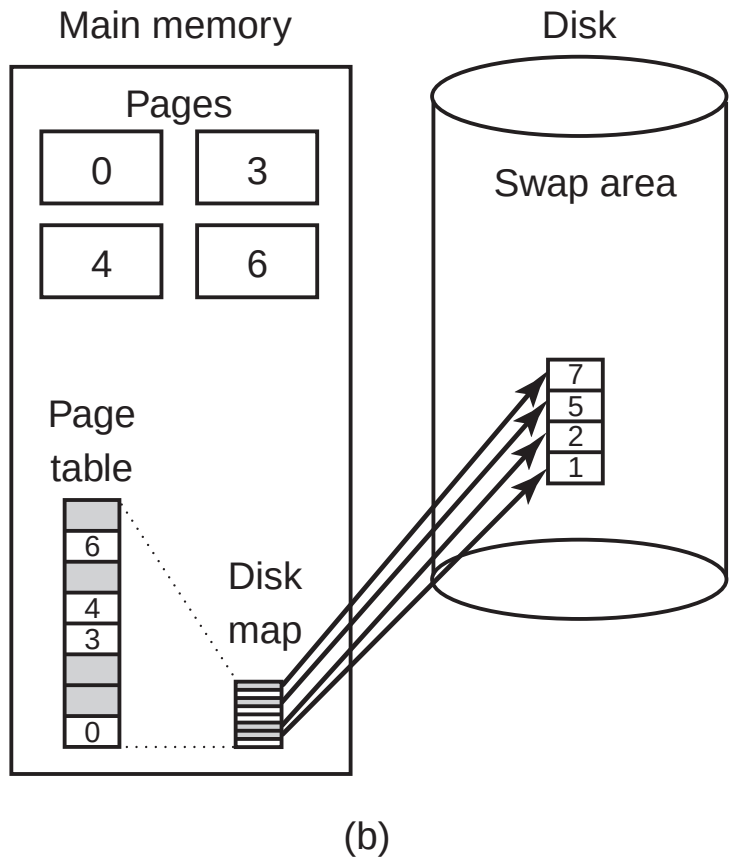
  optimista só é alocada quando necessário – possibilidade de *deadlock*?

  pessimista é reservada – pode não vir a ser usada e, consequentemente, reduzir a utilização dos recursos.

# *Swap Area*: Modos de Alocação.



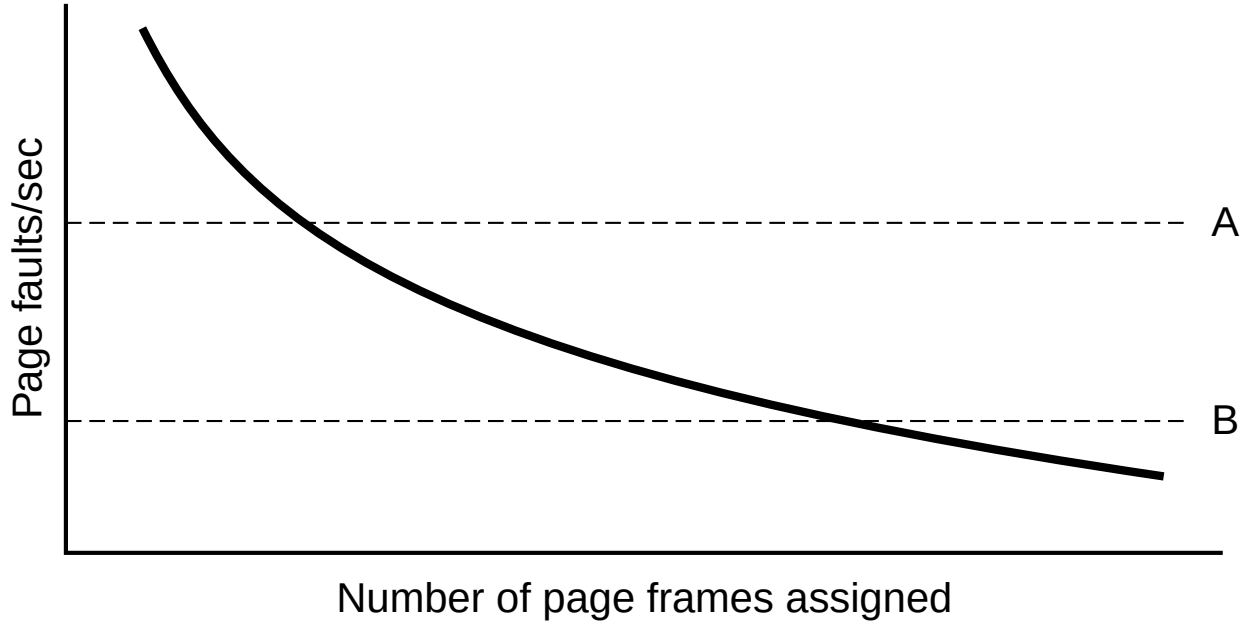

(a)




(b)

# *Working Set e Thrashing*

- Normalmente, a frequência de *page faults* depende do número de *frames* atribuídas a um processo:




- Quando a maioria dos processos não tem um número de páginas suficiente em memória, i.e. o seu *working set*, o computador pode entra num estado de *thrashing*:

  *usa a maioria dos seus recursos para transferir páginas entre a memória principal e o disco*

# Escalonamento Multinível

- Se a frequência de *page faults* aumenta bastante, indicando a possibilidade de *thrashing*, o SO pode "fazer o *swapout* de processos":

  *i.e. transferir todas as páginas dum ou mais processos para o disco.*

- Ou seja, o SO poderá usar dois níveis de escalonamento:
  - O nível superior determina quais os processos em memória.
  - O nível inferior faz o escalonamento dos processos em memória.

# Envolvimento do SO na Gestão de MV Paginada

- Criação dum processo:
  - criar *page table*.
- Comutação de processos:
  - configurar a MMU para o novo processo;
  - invalidar o TLB.
- *Page-fault*:
  - determinar e validar o endereço que a causou;
  - transferir página do disco para a memória, possivelmente com troca.
- Terminação dum processo:
  - libertação da *page table* e das *frames*.
- Outras tarefas:
  - manter número mínimo de *frames* livres;
  - *swap-out* processos, se frequência de *page-faults* excessiva.

# Interacção com E/S

- Converter endereços para realizar E/S:
  - Dispositivos de E/S usam endereços físicos, não lógicos.
- Impedir que as páginas usadas numa operação de E/S sejam expulsas enquanto a operação não terminar:
  - *pin-down*/*lock* páginas na memória.

# Algumas Questões

- Relacionadas com a implementação:
  - Qual deve ser o tamanho duma página?
  - Onde é que as tabelas devem residir?
  - Será que as páginas que constituem a tabela têm que estar sempre em memória?
  - E as páginas com o SO?
- Outras:
  - Quantas *page tables* são necessárias?
  - Como é que a MV paginada assegura protecção entre processos?
  - Que medidas se pode tomar para reduzir a frequência de *page-fault*?
  - Por que razões a MV funciona, i.e. tem um desempenho próximo do dum sistema com capacidade de memória infinita?

# Sumário

# Leitura Adicional

*Modern Operating Systems, 2nd. Ed.*

- Secção 4.1: Fundamentos (4.1.1, 4.1.2 e 4.1.5)
- Secção 4.2: *Swapping*
- Secção 4.3: Memória Virtual
- Secção 4.4: *Page Replacement Algorithms*
- Secção 4.6: *Design Issues* (até 4.6.2 inclusivé).
- Secção 4.7: *Implementation Issues* (excepto 4.7.3).