

# Sistemas Operacionais

## Gestão de memória - Hardware de memória

Prof. Carlos Maziero

DInf UFPR, Curitiba PR

Agosto de 2020

# Conteúdo

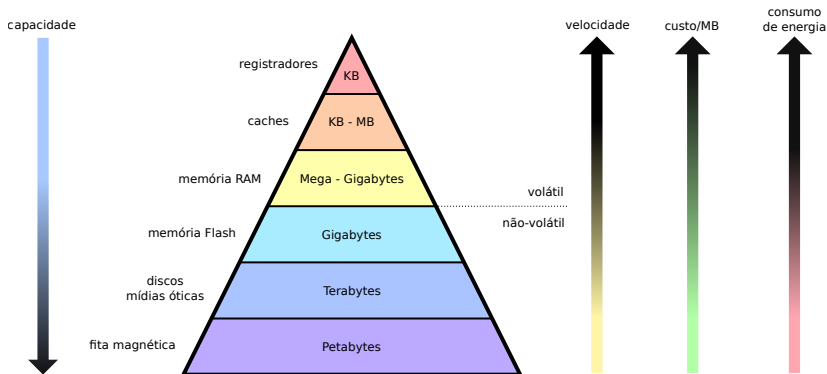
- 1 Armazenamento
- 2 Memória física
- 3 Memória virtual
  - Memória virtual por partições
  - Memória virtual por segmentos
  - Memória virtual por páginas
- 4 Localidade de referências

# Memória = armazenamento

**Memória:** local de armazenamento de informação

Dispositivos	Características
Registradores	capacidade
Caches da CPU	velocidade
RAM	latência
discos SSD, HD	custo por MB
Pendrive	consumo de energia
CD, DVD	volatilidade
Fita magnética	

# Hierarquia de memória



# Velocidades distintas

Meio	Tempo de acesso	Taxa de transferência
Cache L2	1 ns	1 GB/s (1 ns/byte)
Memória RAM	60 ns	1 GB/s (1 ns/byte)
Memória <i>flash</i> (NAND)	2 ms	10 MB/s (100 ns/byte)
Disco rígido SATA	5 ms (desloc. da cabeça de leitura e rotação do disco)	100 MB/s (10 ns/byte)
DVD-ROM	de 100 ms a vários minutos (gaveta aberta ou leitor sem disco)	10 MB/s (100 ns/byte)

**Importante:** Velocidade  $\neq$  Tempo de acesso

# A memória física

Espaço de memória RAM do computador

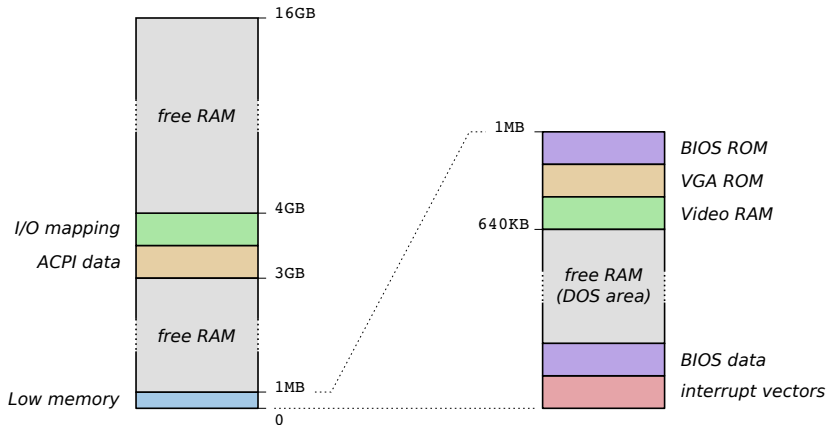
Grande sequência de bytes com endereços individuais

Conteúdo:

- Sistema operacional
- Aplicações em execução
- Buffers de entrada/saída
- Áreas de transferência (DMA)

Dividida em áreas com diferentes finalidades

# Organização da memória em um PC



# Espaço de endereçamento

Faixa de endereços que o processador consegue gerar

Depende da arquitetura e tamanho de barramentos

- 80386: CPU 32 bits, bus 32 bits,  $2^{32}$  endereços
- Core i7: CPU 64 bits, bus 48 bits,  $2^{48}$  endereços

**Não depende** da quantidade de RAM disponível!

- Endereço **válido**: existe um byte de RAM naquela posição
- Endereço **inválido**: não existe RAM naquela posição



# Memória virtual

Separação entre memória RAM e espaço de endereçamento:

- **Endereços físicos:** endereços acessados na RAM.
- **Endereços lógicos:** endereços usados pela CPU.

A memória RAM é acessada por **endereços físicos**.

Ao executar, a CPU e os processos só veem **endereços lógicos**.

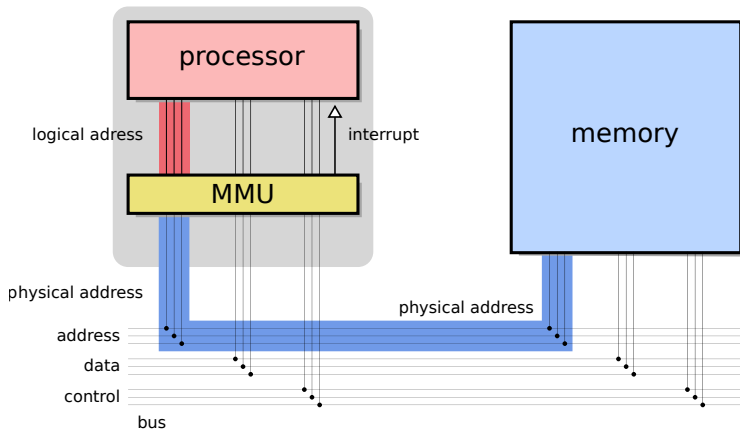
Simplifica o uso da memória pelo SO e processos.

# Conversão de endereços

A conversão *Lógico*  $\rightarrow$  *Físico* é feita pela MMU:

- MMU: *Memory Management Unit*
- Hardware entre a CPU e a memória RAM
- Pode ser gerenciada pelo núcleo do SO
- Converte endereços lógicos em físicos a cada instrução
- Gera IRq se a conversão  $L \rightarrow F$  não for possível

# Unidade de gerência de memória



# Memória virtual

Memória virtual:

- Forma como a CPU e o software vêem a memória
- Corresponde aos endereços lógicos
- Mapeamento  $L \rightarrow F$  é feito pela MMU

Várias formas de implementar:

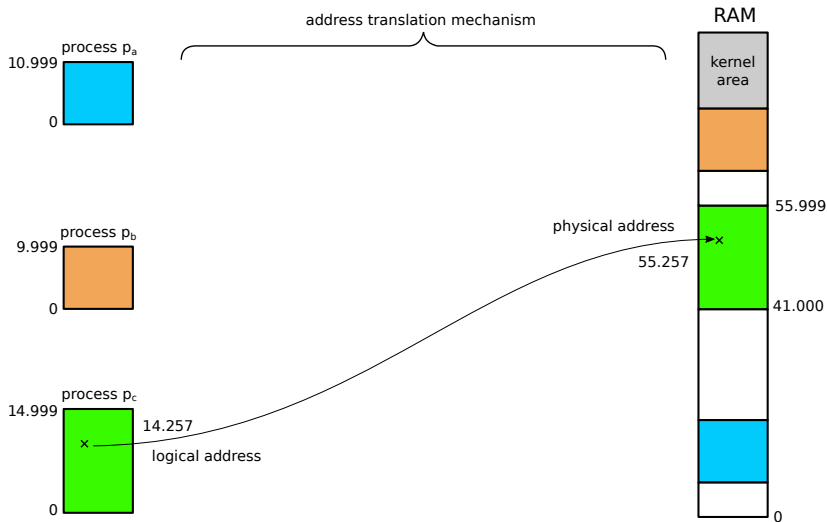
- Por partições
- Por segmentos
- Por páginas
- Por segmentos + páginas

# Memória virtual por partições

Uma partição de tamanho  $T$ :

- Área de RAM contígua (sem buracos) com  $T$  bytes
- Endereços lógicos no intervalo  $[0 \dots T - 1]$
- As partições podem ser tamanhos iguais ou distintos
- As partições podem ser tamanhos fixos ou variáveis
- Cada partição é ocupada por um processo

# Memória virtual por partições



# Memória virtual por partições

A implementação da MMU usa dois registradores:

- Base (B): endereço físico inicial da partição ativa
- Limite (L): tamanho em bytes dessa partição

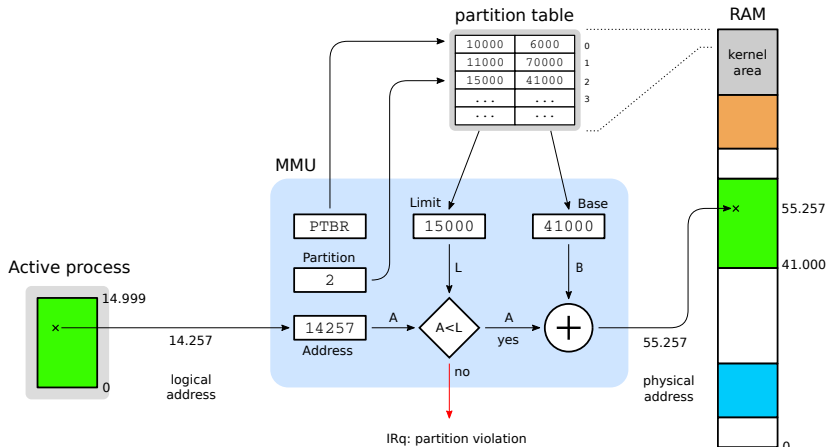
Regra de conversão:

se  $E_L < L$  então  $E_F = B + E_L$  senão *erro*

Os valores de  $B$  e  $L$  são ajustados a cada troca de contexto

Uma tabela contém  $B$  e  $L$  de todas as partições

# Memória virtual por partições





# Memória virtual por partições

## Vantagens:

- Simplicidade
- Rapidez

## Desvantagens:

- Processos podem ter tamanhos distintos das partições
  - Processos pequenos desperdiçam RAM
  - Processos muito grandes não podem executar
- Número de processos  $\leq$  número de partições
- Dificuldade para compartilhar memória

# Memória virtual por segmentos

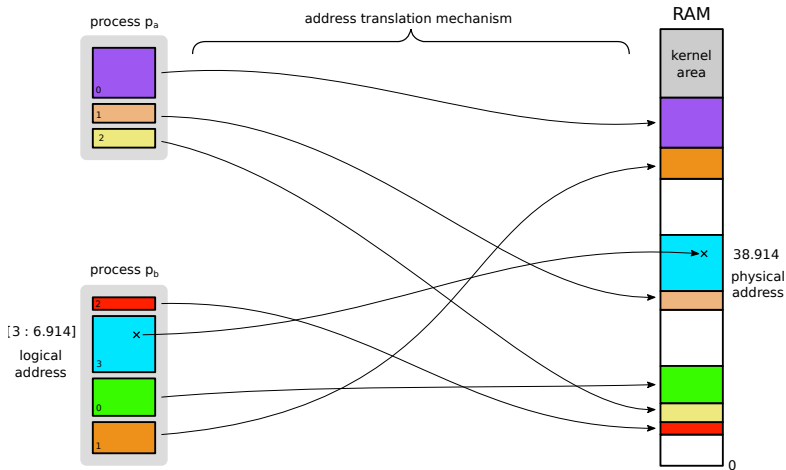
Cada processo tem um conjunto de áreas internas:

- Armazenam as diferentes partes do processo
- Código, dados, pilha, *heap*, ...
- Bibliotecas compartilhadas

Ideia: estender o modelo de partições:

- O espaço de endereçamento é dividido em **segmentos**
- Cada área do processo fica em um **segmento**
- Cada processo tem uma **tabela de segmentos**

# Memória virtual por segmentos



# Memória virtual por segmentos

Os endereços lógicos são **bidimensionais**  $[S : O]$

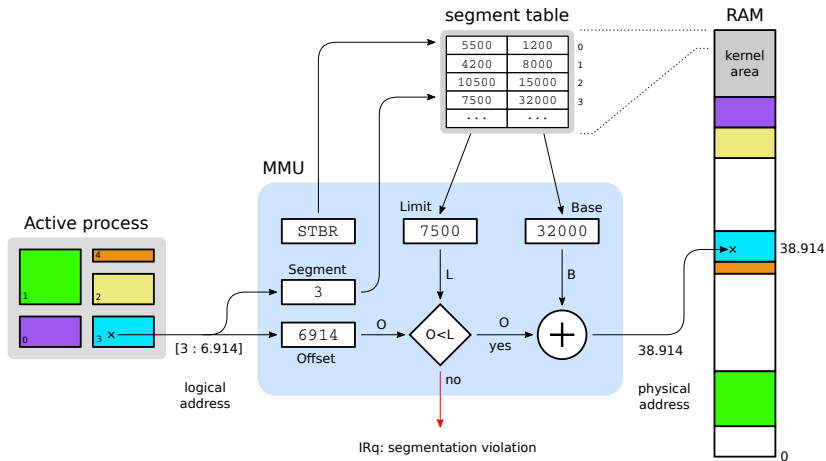
- $S$ : número do segmento daquele processo
- $O$ : *offset* (posição) dentro do segmento  $S$

Uma tabela de segmentos para cada processo:

- Base e Limite de cada segmento
- Permissões do segmento (*read*, *write*, ...)
- Outros flags

A tradução de endereços é similar ao modelo por partições

# Memória virtual por segmentos



# No Intel 80.x86 (32 bits)

Duas tabelas de segmentos por processo:

- *Local Descriptor Table* (LDT): segmentos locais do processo
- *Global Descriptor Table* (GDT): segmentos compartilhados entre vários processos (DLLs, buffers, etc)
- Cada tabela tem até 8.192 entradas

Registradores para os segmentos mais usados:

- CS: *Code Segment*, código em execução
- SS: *Stack Segment*, pilha do processo
- DS, ES, FS, GS: *Data Segments*, dados usados pelo processo

# Memória virtual por páginas

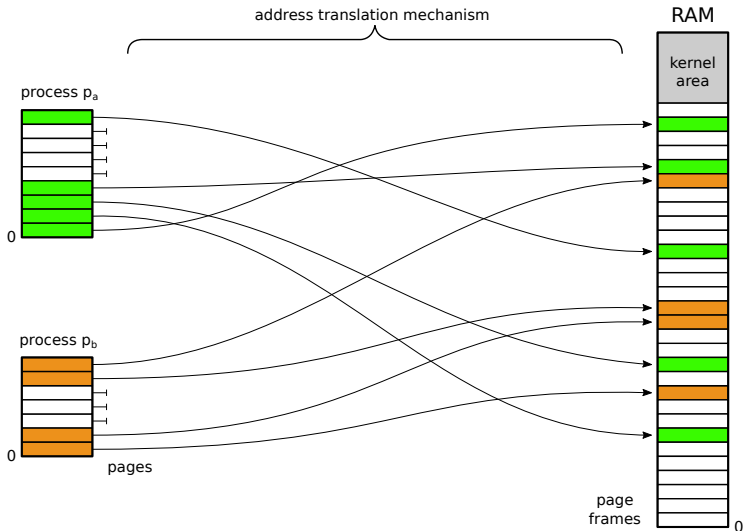
Ideia: dividir RAM e espaço de endereçamento em blocos:

- Blocos de mesmo tamanho (em geral 4.096 bytes)
- Blocos de endereços **lógicos: páginas**
- Blocos de endereços **físicos: quadros**

A tradução  $L \rightarrow F$  consiste em mapear páginas  $\rightarrow$  quadros

O endereços lógicos são lineares (unidimensionais)

# Memória virtual por páginas





# Tradução páginas → quadros

Mapear as páginas dos processos em quadros de RAM

Usa uma **Tabela de Páginas** (*page table*) por processo:

- Cada entrada da tabela corresponde a uma página
- A entrada contém o número do quadro e outros flags
- Cada processo possui sua tabela de páginas
- *Page Table Base Register* (PTBR) indica a tabela ativa

O número de página é extraído do endereço lógico

# Estrutura do endereço lógico

CPU de 32 bits com páginas de 4.096 bytes ( $2^{12}$  bytes)

Conversão do endereço lógico  $01803E9A_h$ :

$$01803E9A_h \rightarrow \overbrace{0000\ 0001\ 1000\ 0000\ 0011}^{\text{page: 20 bits}} \overbrace{1110\ 1001\ 1010}_2^{\text{offset: 12 bits}}$$

$$\rightarrow \overbrace{0000\ 0001\ 1000\ 0000\ 0011}^{\text{page}=01803_h} \overbrace{1110\ 1001\ 1010}_2^{\text{offset}=E9A_h}$$

$$\rightarrow \text{page} = 01803_h \quad \text{offset} = E9A_h$$

# Estrutura do endereço lógico

CPU de 32 bits com páginas de 4 KBytes:

- Endereços lógicos de 32 bits e páginas com 4 Kbytes
- 4 Kbytes são  $2^{12}$  bytes: o deslocamento usa 12 bits
- *Page Number*: 20 bits mais significativos
- *Offset*: 12 bits menos significativos

Os 20 bits mais significativos definem  $2^{20}$  páginas

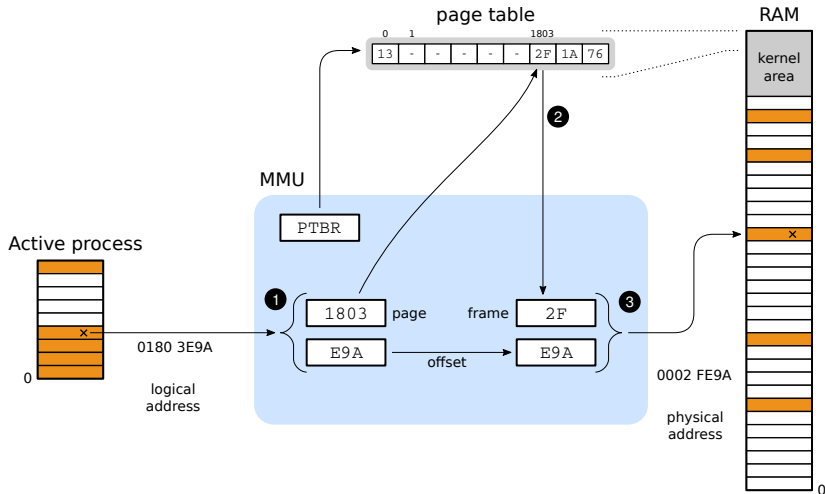
Espaço total: até 1.048.576 páginas com 4.096 bytes cada

# Tradução de endereços lógicos em físicos

Passos da MMU para a tradução  $L \rightarrow F$ :

- 1 Decompor o endereço lógico:
  - número da página (bits mais significativos)
  - *offset* (bits menos significativos)
- 2 Consultar a tabela de páginas ativa:
  - obter o número do quadro correspondente à página
  - se a página não está mapeada, gerar uma **falta de página**
- 3 Construir o endereço físico:
  - número do quadro (bits mais significativos)
  - *offset* (bits menos significativos)

# Tradução de endereços



# Flags de controle das páginas

Para cada entrada da tabela de páginas:

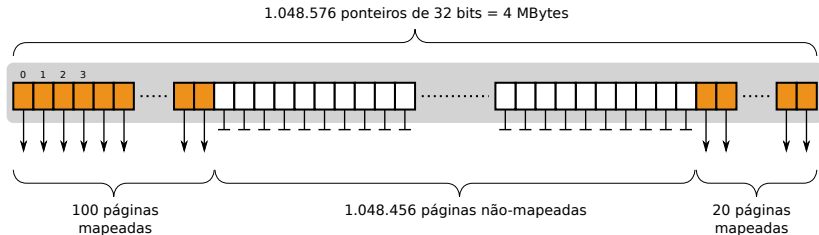
- **Valid:** página é mapeada no espaço do processo
- **Writable:** página pode ser acessada em escrita
- **User:** indica código de usuário ou de núcleo
- **Present:** a página está presente na memória RAM
- **Accessed:** a página foi acessada recentemente
- **Dirty:** página foi modificada recentemente
- Outros flags, para uso do núcleo do SO

Conteúdo depende da arquitetura do processador

# Tamanho da tabela de páginas

Em uma CPU de 32 bits com páginas de 4 Kbytes:

- Uma entrada na tabela: ~ 32 bits (nº do frame + flags)
- Tabela com  $2^{20}$  entradas: 4 Mbytes de RAM
- Desperdício de RAM em processos pequenos



# Tabelas de páginas multiníveis

São estruturadas como árvores:

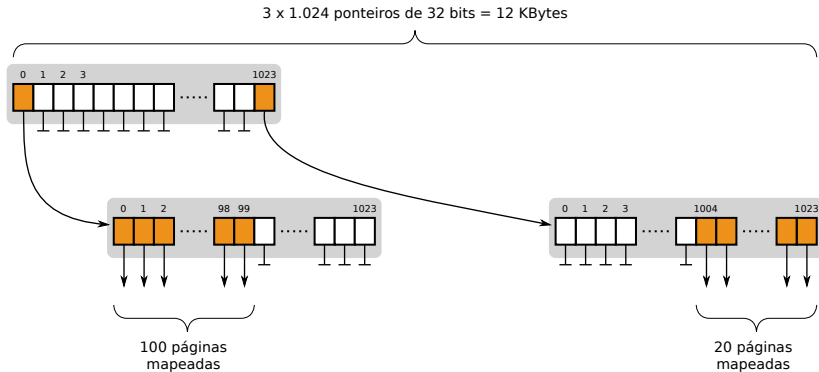
- *tabela de páginas de primeiro nível (diretório de páginas)*
- *tabelas de páginas de segundo nível*
- *tabelas de páginas de terceiro nível*
- ...

Número de níveis depende da arquitetura:

- 2 níveis: *Intel 80.x86*
- 3 níveis: *Sun Sparc, DEC Alpha*
- 4 níveis: *Intel core, Intel i3/i5/i7*



# Tabela de páginas com 2 níveis



Cada subtabela ocupa uma página (4.096 bytes)

# Estrutura do endereço lógico

CPU de 32 bits, páginas de 4 KBytes, dois níveis (10 bits cada):

$$01803E9A_h \rightarrow \overbrace{0000\ 0001\ 10}^{p_2:10\ bits} \overbrace{00\ 0000\ 0011}^{p_1:10\ bits} \overbrace{1110\ 1001\ 1010}^{offset:12bits}$$

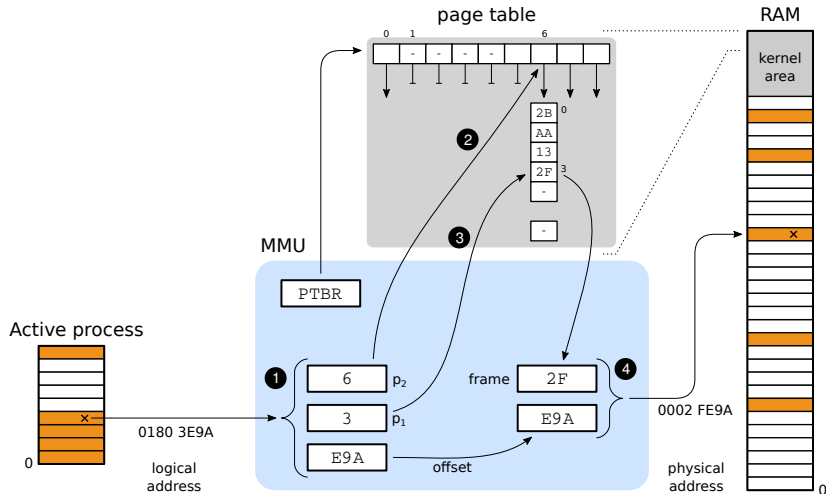
$$\rightarrow \overbrace{0000\ 0001\ 10}^{p_2=0006_h} \overbrace{00\ 0000\ 0011}^{p_1=0003_h} \overbrace{1110\ 1001\ 1010}^{offset=E9A_h}$$

$$\rightarrow p_2 = 0006_h \quad p_1 = 0003_h \quad offset = E9A_h$$

# Tradução de endereços pela MMU

- 1 Decompor o endereço lógico  $L$ :
  - índice externo com 10 bits ( $p_2$ )
  - índice interno com 10 bits ( $p_1$ )
  - *offset* com 12 bits ( $O$ )
- 2 Usar  $p_2$  como índice na tabela externa:
  - obter o endereço da tabela interna
  - se não estiver mapeada, *page fault*
- 3 Usar  $p_1$  como índice na tabela interna:
  - obter o número do quadro  $Q$
  - se não estiver mapeada, *page fault*
- 4 Construir o endereço físico com  $Q$  e  $O$

# Tabelas multi-níveis



# Economia da tabela multi-níveis

Para um processo pequeno (2 páginas):

- Com um nível:  $4 \times 2^{20} = 4 \text{ MBytes}$
- Com dois níveis:  $4 \times (2^{10} + 2 \times 2^{10}) = 12 \text{ KBytes}$

Para um processo grande ( $2^{20}$  páginas):

- Com um nível:  $4 \times 2^{20} = 4 \text{ MBytes}$
- Com dois níveis:  $4 \times (2^{10} + 2^{10} \times 2^{10}) \approx 4,004 \text{ MBytes}$

**CUSTO:** cada tradução precisa mais acessos à RAM (tabelas)

# Cache da tabela de páginas

Traduções *Página*  $\rightarrow$  *Quadro* são repetitivas

TLB - *Translation Lookaside Buffer*:

- cache interno da MMU para traduções recentes
- armazena traduções  $P \rightarrow Q$  recentes
- funciona como uma tabela de *hash* em hardware
- pequeno: entre 16 e 256 entradas

É um cache rápido:

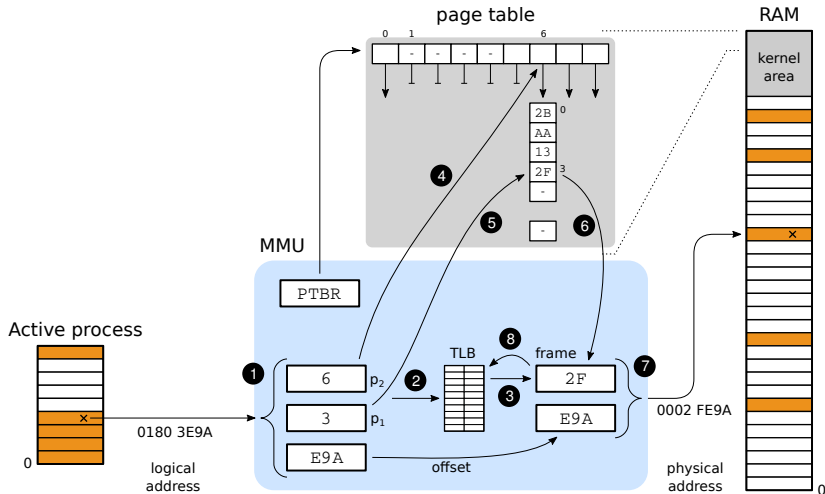
- um acerto (*cache hit*) custa 1 ciclo de *clock*
- um erro custa de 10 a 30 ciclos de *clock*

# Funcionamento do TLB

Para traduzir um endereço lógico  $L$ , a MMU:

- 1 decompõe  $L$  em  $P$  (página) e  $O$  (*offset*)
- 2 verifica se  $P \rightarrow Q$  está no cache TLB
- 3 se estiver (*TLB hit*), pega  $Q$
- 4 caso contrário (*TLB miss*), busca  $Q$  na tabela de páginas (passos 4-6)
- 7 usa  $Q$  e  $O$  para obter o endereço físico  $F$
- 8 a tradução  $P \rightarrow Q$  é adicionada ao TLB

# Cache da tabela de páginas





# Desempenho do TLB

Considerando o seguinte sistema:

- Frequência de CPU de 2 GHz (relógio de 0,5 ns)
- Tempo de acesso à RAM de 50 ns
- Tabelas de páginas com 3 níveis
- TLB com custo de acerto de 0,5 ns e de erro de 30 ns

Calcular tempo médio de acesso à RAM:

- sem TLB
- com TLB e taxa de acerto de 95%

# Desempenho do acesso à RAM

$$\text{Sem TLB: } t_{med} = \overbrace{3 \times 50 \text{ ns}}^{\text{tables}} + \overbrace{50 \text{ ns}}^{\text{address}} = 200 \text{ ns}$$

$$\text{Com TLB (hit): } t_{med} = t_{hit} + t_{address}$$

$$\text{Com TLB (miss): } t_{med} = t_{miss} + t_{address}$$

$$\begin{aligned} t_{med}(95\%) &= 95\% \times (0.5 \text{ ns} + 50 \text{ ns}) \\ &+ (1 - 95\%) \times 30 \text{ ns} + 50 \text{ ns} \\ &= 51.975 \text{ ns} \end{aligned}$$

$$t_{med}(80\%) = 56.40 \text{ ns}$$

# Cache da tabela de páginas

Fatores que influenciam a taxa de acertos do TLB:

- **Tamanho do TLB:** quanto mais entradas, mais acertos
- **Política de substituição** das entradas do TLB
- **Trocas de contexto** exigem a limpeza do TLB
- **Padrão de acessos** à memória (localidade de referências)

# Localidade de referências

Concentração dos acessos à memória no espaço e/ou tempo.

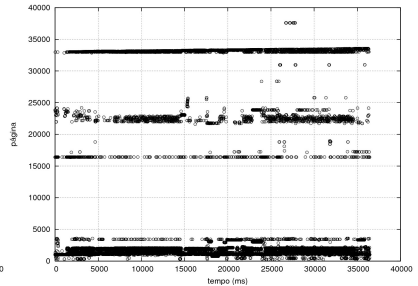
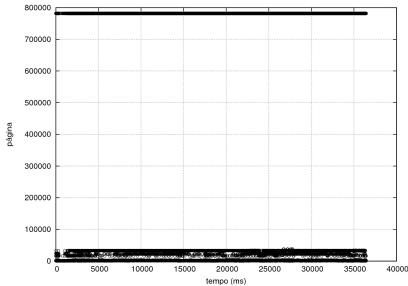
Impacta na eficiência dos mecanismos de gerência de memória.

Três formas básicas:

- **Temporal:** um recurso usado há pouco tempo será provavelmente usado novamente em breve;
- **Espacial:** um recurso será mais provavelmente acessado se outro recurso próximo a ele já foi acessado;
- **Sequencial:** após o acesso a um recurso na posição  $p$ , há maior probabilidade de acessar um recurso em  $p + 1$

# Localidade de referências

Exemplo: visualizador de imagens *gThumb* (ambiente Gnome):



# Localidade de referências

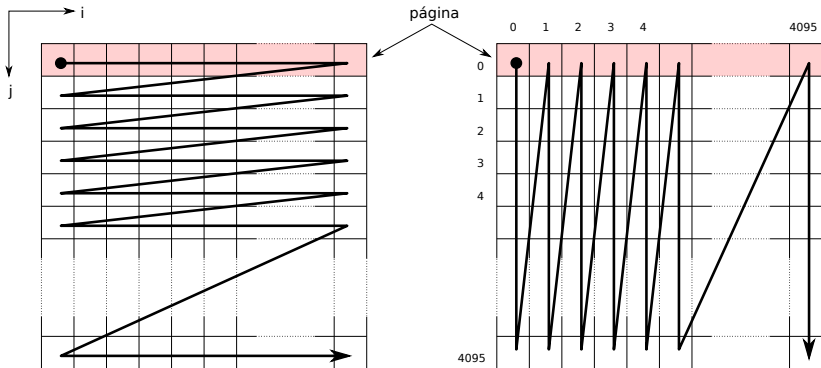
Exemplo: Preenchimento de uma matriz por linhas:

```
1 unsigned char buffer[4096][4096] ;  
2  
3 int main ()  
4 {  
5     for (int i=0; i<4096; i++)      // percorre as linhas do buffer  
6         for (int j=0; j<4096; j++)  // percorre as colunas do buffer  
7             buffer[i][j]= (i+j) % 256 ;  
8 }
```

Ou por colunas:

```
1 ...  
2 for (j=0; j<4096; j++)             // percorre as colunas do buffer  
3     for (i=0; i<4096; i++)         // percorre as linhas do buffer  
4         buffer[i][j]= (i+j) % 256 ;  
5 ...
```

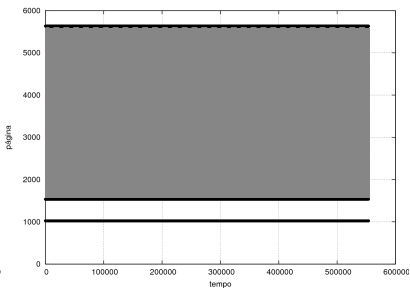
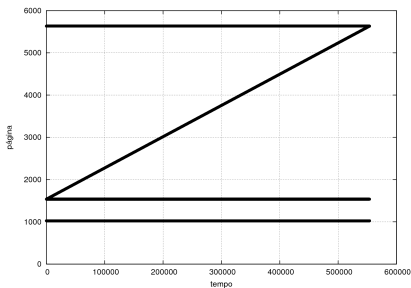
# Padrão de acessos à memória



# Diferenças de desempenho

Por linhas: ~ 5 páginas distintas em 100.000 acessos

Por colunas: ~ 3.000 páginas distintas em 100.000 acessos



Qual o impacto sobre o desempenho do TLB?



# Localidade de referências

Fatores que influenciam a localidade de referências:

- As **estruturas de dados** usadas pelo programa
  - vetores e matrizes têm mais localidade de referências
  - listas encadeadas e árvores **podem ter** baixa localidade
- Os **algoritmos** usados pelo programa
  - acessos por linhas ou por colunas
- A qualidade do **compilador**
  - colocar variáveis usadas juntas nas mesmas páginas
  - alinhar as estruturas de dados nas páginas