Conceitos de arquitetura e organização de computadores

Gonzalo Travieso[1]

2018

[1] gonzalo@ifsc.usp.br

Outline

1 Estrutura básica

2 Tecnologia

3 Hierarquia de memória

4 Entradas e saídas

5 Multiprogramação

6 Gerenciamento de memória

7 Pipeline de instruções

8 Execução superescalar

Tópico

1 Estrutura básica

2 Tecnologia

3 Hierarquia de memória

4 Entradas e saídas

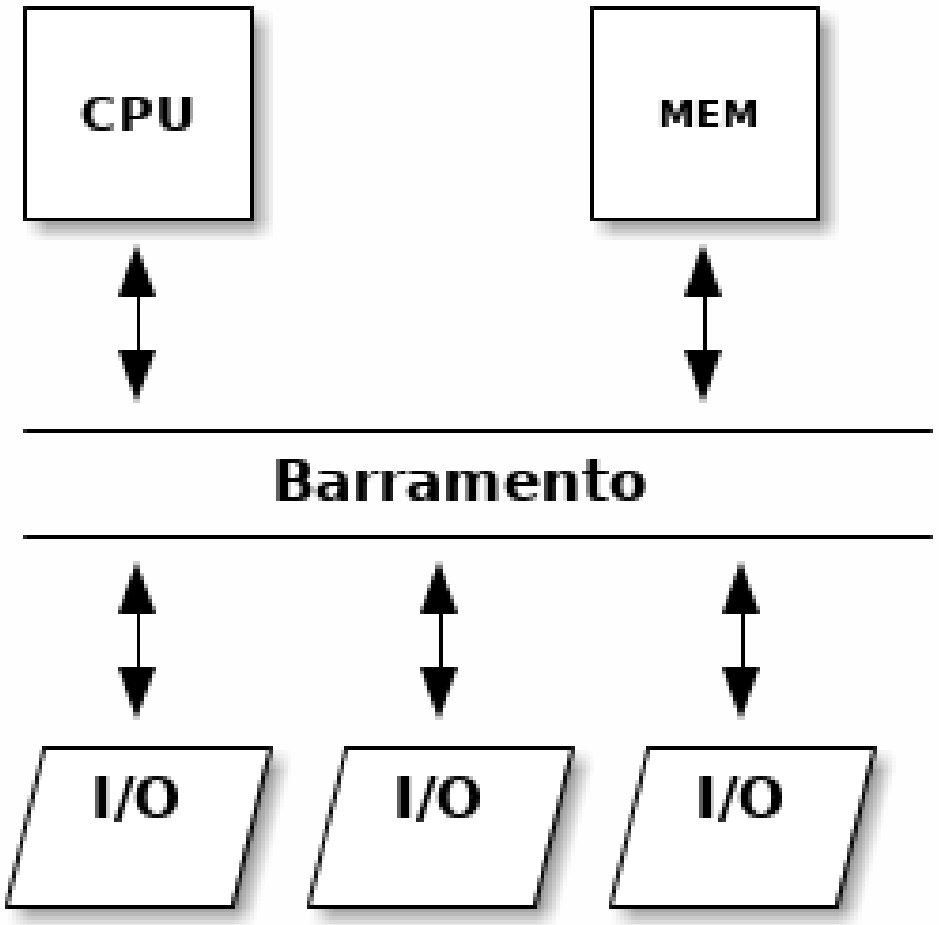
5 Multiprogramação

6 Gerenciamento de memória

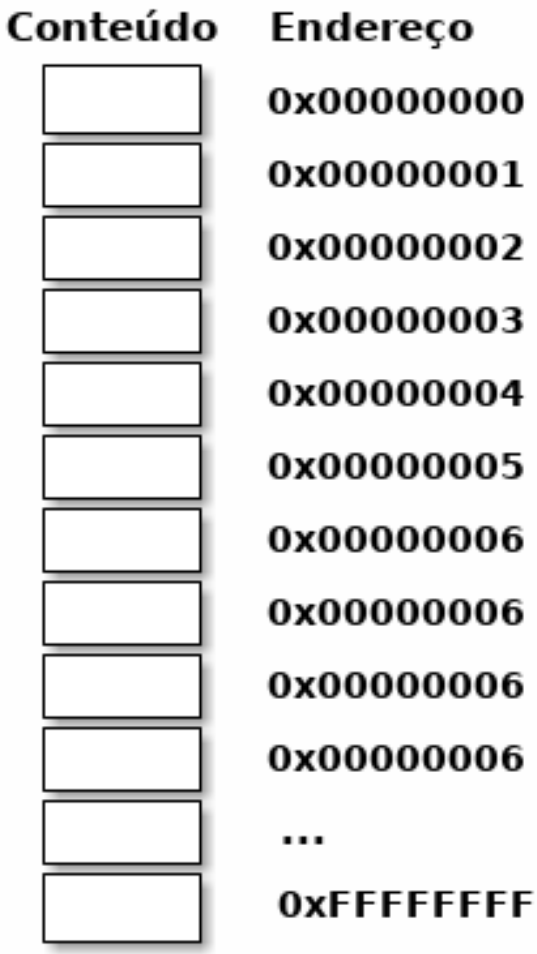
7 Pipeline de instruções

8 Execução superescalar

Estrutura básica



Endereçamento



CPU

- Unidades lógicas aritméticas (operações básicas)
- Controlador
- Registradores
  - Registradores gerais
  - Registradores especiais
  - *Program counter* ou *Instruction pointer*

Exemplo: `x86_64`

- rax, rbx, rcx, rdx
- rbp, rsp, rbi, rdi
- r8 a r15

Tópico

1 Estrutura básica

2 Tecnologia

3 Hierarquia de memória

4 Entradas e saídas

5 Multiprogramação

6 Gerenciamento de memória

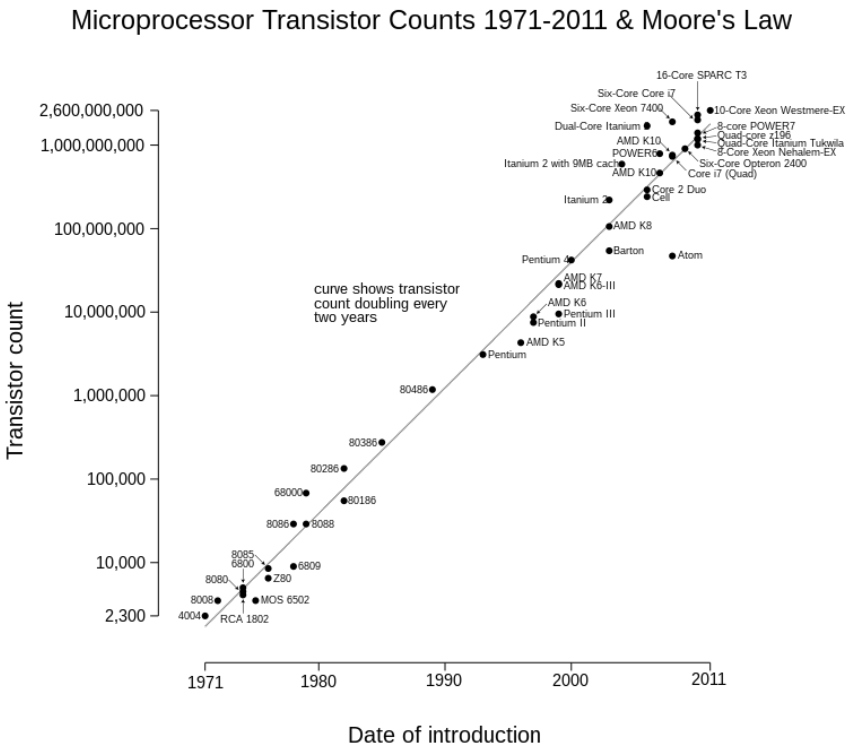
7 Pipeline de instruções

8 Execução superescalar

Tecnologia

- Válvulas/relés
- Transistores
- Circuitos integrados
  - SSI (small scale)
  - MSI (medium)
  - LSI (large)
  - VLSI (very large)
  - ULSI (ultra large)
  - ...

Lei de Moore

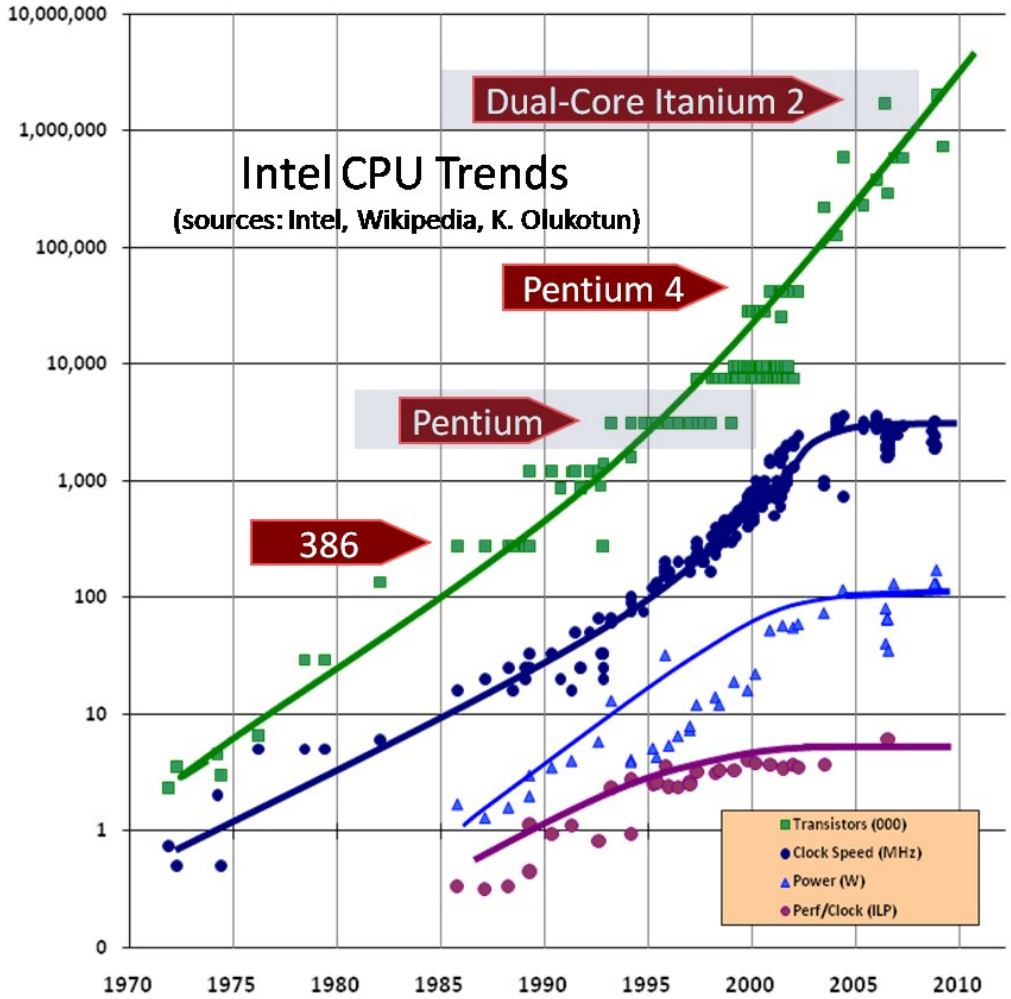


Lei de Moore

"O número de transistores em um *chip* dobra a cada 18 meses"

Fonte: Wikipedia

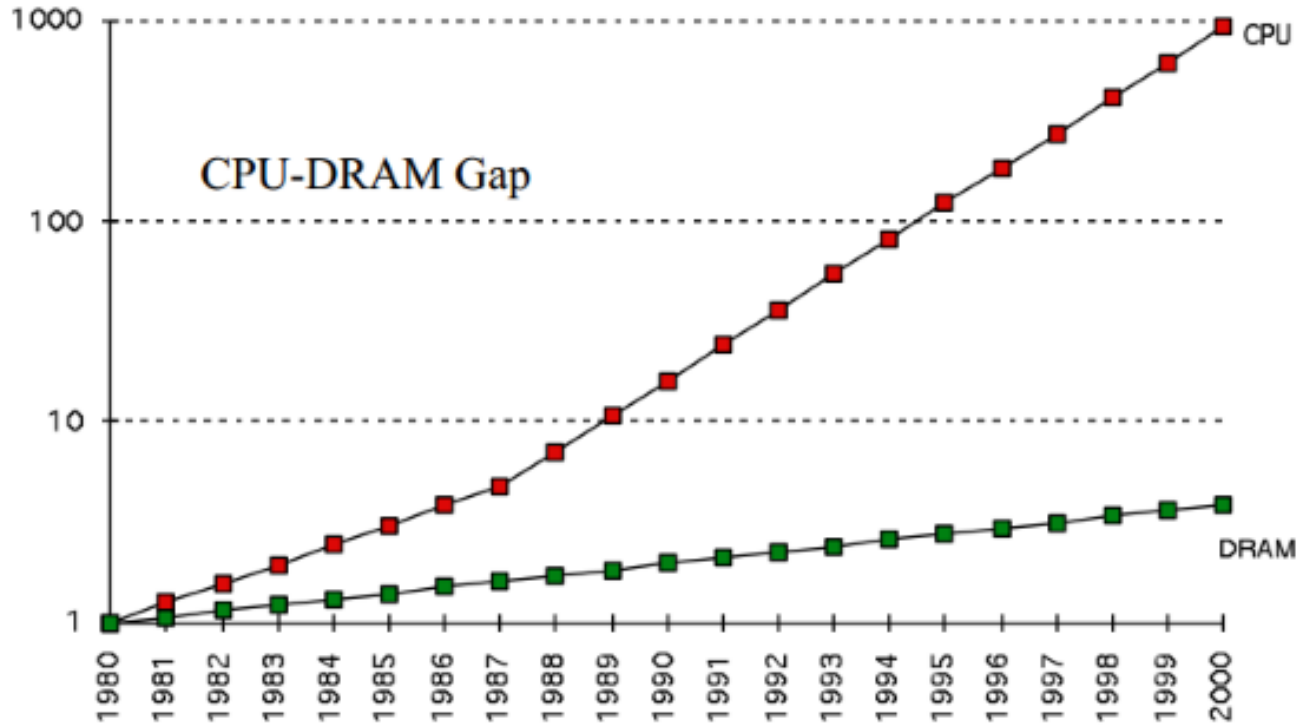
Transistores e desempenho



Fonte: ExtremeTech

CPU x Memória

- Processor vs Memory Performance

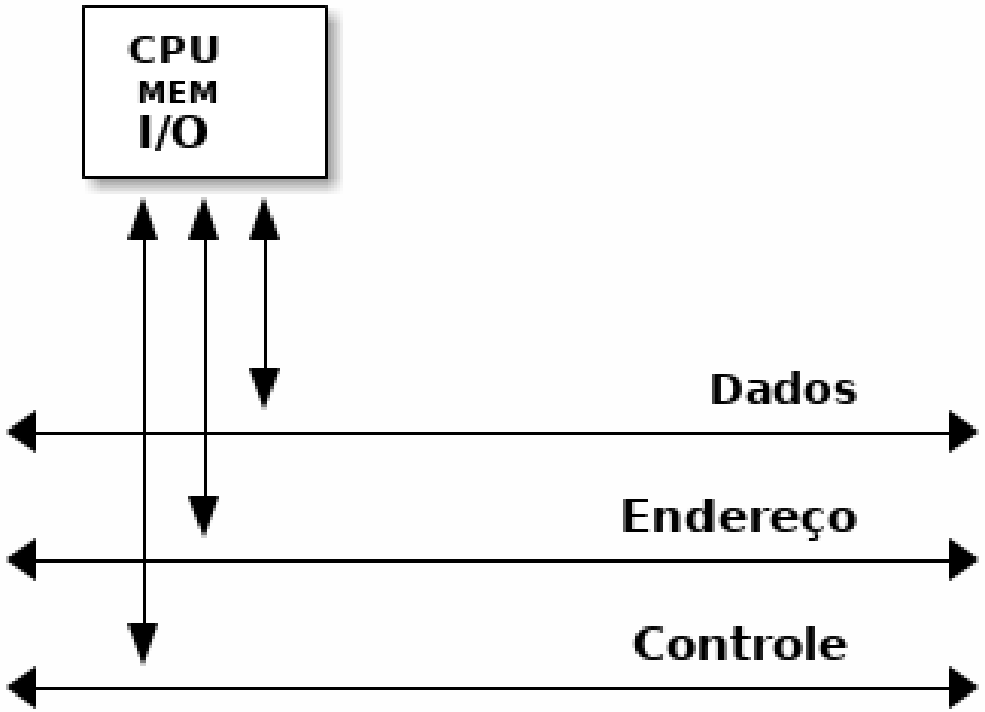


1980: no cache in microprocessor;

1995 2-level cache

Fonte: ExtremeTech

Barramento



Características do barramento

- Largura
- Relógio de operação

Largura de banda

Quantidade de bytes transferidos por segundo no barramento.
A largura de banda total é limitada no barramento

Largura de banda

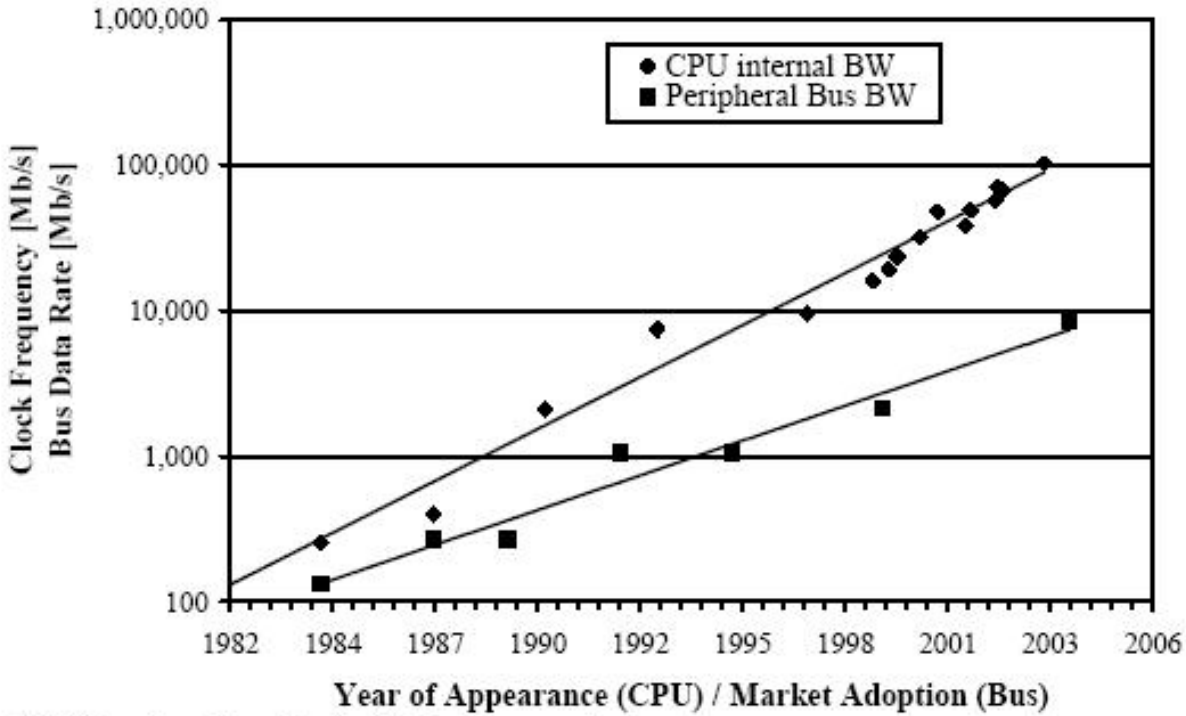
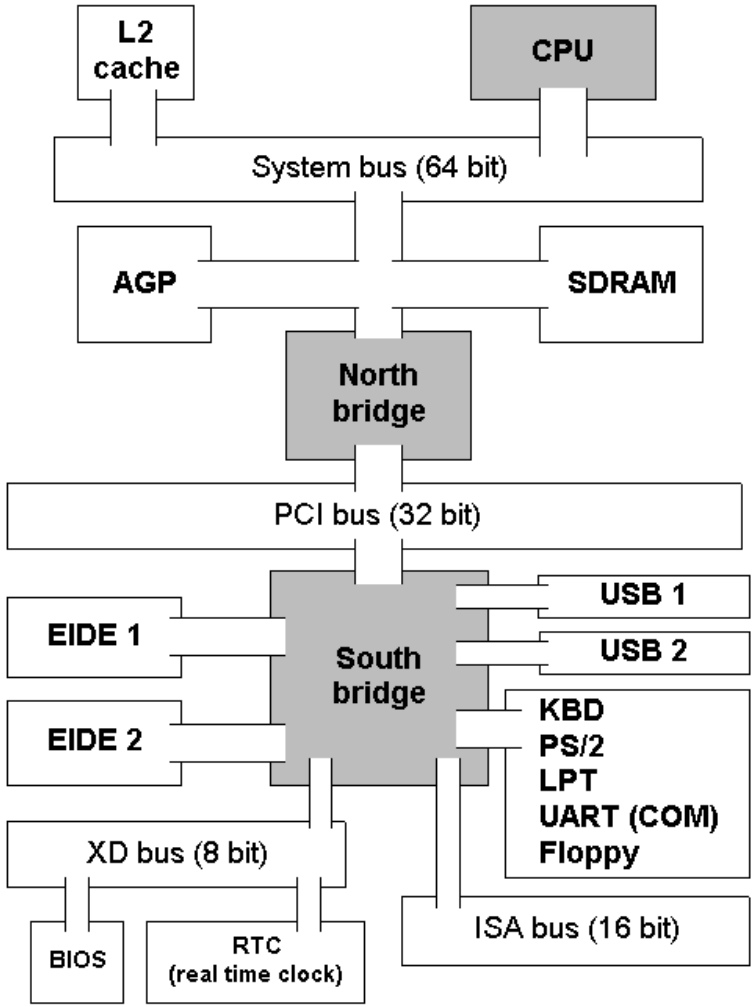


Fig. 1. CPU and peripheral bandwidth increase over time

Fonte: Ünlü et al.

Múltiplos barramentos



Fonte: Karbosguide

Tópico

1 Estrutura básica

2 Tecnologia

3 Hierarquia de memória

4 Entradas e saídas

5 Multiprogramação

6 Gerenciamento de memória

7 Pipeline de instruções

8 Execução superescalar

Locais da memória

- CPU (registradores)
- Chip (cache)
- Interna (RAM)
- Externa (discos, backup)

Tipos de acesso

- Sequencial (ex: fitas)
- Direto (ex: discos)
- Aleatório (ex: RAM)
- Associativo (ex: algumas caches)

Desempenho

Tempo de acesso Tempo desde o pedido do dado pela CPU até sua chegada.

Tempo de ciclo Intervalo necessário entre um acesso e outro a posições associadas.

Tempo de transferência Tempo por byte para transferir uma quantidade de dados entre memória e CPU.

Considerações

- Tamanho
- Velocidade
- Preço

Escolha dois

- Mais baratas por byte são lentas.
- Mais rápidas são mais caras por byte.

Localidade

O que permite sistemas com bastante memória de boa eficiência é a *localidade de referências* existente na maioria dos programas.

```
/* Produto matriz-vetor y = A x */
for (i = 0; i < N; ++i) {
    y[i] = 0.0;
    for (j = 0; j < N; ++j) {
        y[i] += A[i][j]*x[j];
    }
}
```

Tipos de localidade

Localidade temporal Ocorre quando uma mesma posição de memória é acessada diversas vezes em um curto intervalo de tempo. (Exemplo: variáveis `i`, `j` e `N`, ponteiros `A`, `x` e `y`, posição `y[i]`.)

Localidade espacial Ocorre quando posições próximas na memória são acessadas em um curto intervalo de tempo. (Exemplo: posições `A[i][0]`, `A[i][1]`, etc.; posições `x[0]`, `x[1]`, etc.)

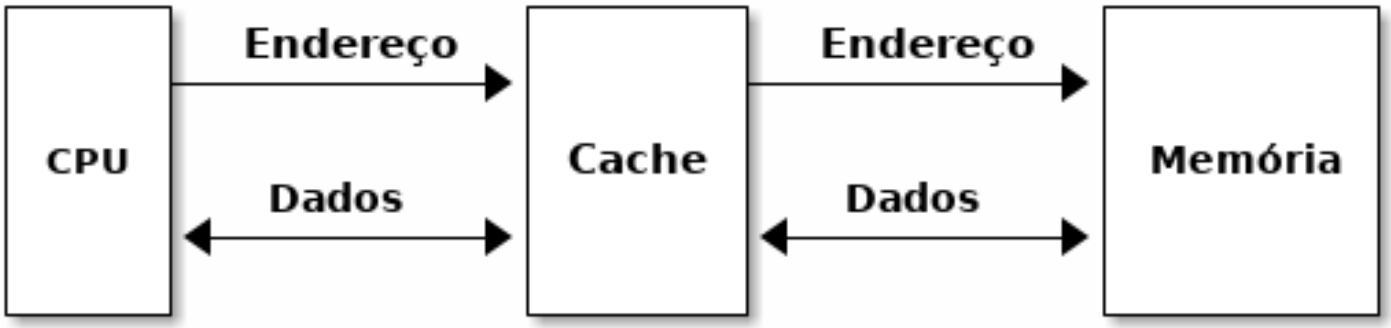
Consequência

- Isso significa que os acessos não são homogeneamente distribuídos entre todas as posições de memória, mas certas posições têm maior probabilidade de serem acessadas em cada instante do que outras.
- Portanto, se conseguirmos acesso mais rápido às posições que vão mais provavelmente ser acessadas, teremos maior eficiência.

Hierarquia de memória

- Registradores
- cache (L1/L2/L3)
- Memória interna
- (SSD? Ex: Intel Optane)
- Disco (com cache interno)
- Memória de backup

Caches



Tamanho da cache

- Caches maiores são mais caras.
- Na prática, ficam no *chip* da CPU e usam transistores que poderiam ser usados pela CPU.
- Caches maiores podem ser mais lentas.

Blocos e linhas

- O organização não é byte a byte.
- Memória é dividida em blocos do mesmo tamanho.
- Cache é dividida em linhas cada uma do mesmo tamanho de um bloco de memória.

Mapeamento

- Nem toda a memória cabe na cache.
- Precisamos decidir em que linha de cache cada bloco da memória pode ser colocado.
- Para isso usamos um método de mapeamento:
  - Mapeamento direto
  - Mapeamento associativo
  - Mapeamento associativo por conjuntos

Mapeamento direto

Cada bloco de memória pode ser colocado em apenas uma linha de cache.
O endereço de memória é dividido em três partes:

Rótulo é um número que identifica o bloco de memória na cache, entre os que podem estar em uma dada linha.

Linha é o número da linha de cache onde o bloco será guardado.

Offset é o endereço do byte a ser acessado dentro do bloco.

Exemplo

- Endereço de 32 bits.
- Tamanho dos blocos/linhas de 16 bytes. Ou seja, 4 bits para escolher o byte dentro da linha.
- 4 M (2^{22}) bytes de cache, o que são 2^{18} linhas. Ou seja, 18 bits para escolher a linha do cache.
- Sobram 10 bits, que é o tamanho do rótulo.

Funcionamento

1. A CPU coloca o endereço.
2. O endereço é interceptado pela cache.
3. A cache quebra o endereço nos pedaços da figura anterior.
4. A cache verifica o número da linha no endereço.
5. Verifica se o rótulo fornecido no endereço é igual ao rótulo especificado para essa linha no cache.
6. Se for igual, é um *cache hit* e o dado é acessado diretamente da cache.
7. Se for diferente, é um *cache miss*. A cache então fornece o endereço do bloco de memória que contém o endereço de memória especificado pela CPU e solicita o bloco inteiro à memória.
8. Transfere então o bloco para a linha correspondente e anota seu rótulo.
9. Em seguida, passa o dado solicitado para a CPU.

Localidade

- Se o dado transferido para a cache é acessado novamente (localidade temporal) ele já vai estar no cache.
- Se um dado adjacente é acessado (localidade espacial), ele pode estar no mesmo bloco já acessado, e portanto já estar no cache.

Problema

- Cada bloco só pode ser colocado em uma linha.
- Muitos blocos mapeiam para a mesma linha.
- Se dois blocos que mapeiam para a mesma linha têm endereços acessados frequentemente, eles ficam interferindo um com o outro (ao colocar um, tiramos o outro): thrashing.

Mapeamento associativo

Cada bloco por ser colocado em qualquer linha do cache. O endereço de memória passa a ser dividido apenas em rótulo e offset.

Exemplo

- Endereço de 32 bits.
- Tamanho dos blocos/linhas de 16 bytes. Ou seja, 4 bits para escolher o byte dentro da linha.
- Sobram 28 bits, que é o tamanho do rótulo.

Funcionamento

1. A CPU coloca o endereço.
2. O endereço é interceptado pela cache.
3. A cache quebra o endereço nos pedaços da figura anterior.
4. A cache verifica o rótulo do bloco solicitado e compara com os rótulos dos blocos em todas as linhas.
5. Se o rótulo fornecido no endereço é igual ao rótulo especificado para uma das linhas temos um *cache hit* e o dado é acessado diretamente da cache.
6. Se for diferente do de todas as linhas, é um *cache miss*. A cache então fornece o endereço do bloco de memória que contém o endereço de memória especificado pela CPU e solicita o bloco inteiro à memória.
7. A cache escolhe em seguida uma das linhas para receber o novo bloco (algoritmo de troca).
8. Transfere então o bloco para a linha escolhida e anota seu rótulo.
9. Em seguida, passa o dado solicitado para a CPU.

Problemas

- O rótulo do endereço precisa ser comparado com todos os rótulos de linhas.
  - Se comparamos sequencialmente, leva muito tempo.
  - Se comparamos simultaneamente, precisamos muito *hardware*.
- Cada comparação envolve um grande número de bits.
- O algoritmo de troca não é trivial (e precisa ser implementado em *hardware*).

Mapeamento associativo por conjuntos

- As linhas do cache são agrupadas em conjuntos de linhas.
- Cada conjunto tem um número fixo (e pequeno) de linhas.
- Cada bloco é mapeado diretamente para um conjunto de linhas.
- Dentro do conjunto, o bloco é mapeado associativamente para a linha.

Mapeamento associativo por conjuntos

O endereço de memória é quebrado em três partes:

Rótulo Identifica o bloco de memória entre os que podem estar em um dado conjunto de linhas.

Conjunto Identifica o número do conjunto de linhas onde o bloco deve ser colocado.

Offset do byte dentro da linha.

Exemplo

- Endereço de 32 bits.
- Tamanho dos blocos/linhas de 16 bytes. Ou seja, 4 bits para escolher o byte dentro da linha.
- 4 M (2^{22}) bytes de cache, o que são 2^{18} linhas.
- Conjuntos de 4 linhas, o que significa que temos 2^{16} conjuntos, ou 16 bits para identificar conjunto.
- Sobram 12 bits, que é o tamanho do rótulo.

Funcionamento

1. A CPU coloca o endereço.
2. O endereço é interceptado pela cache.
3. A cache quebra o endereço nos pedaços da figura anterior.
4. A cache verifica o número do conjunto.
5. Em seguida, compara o rótulo fornecido com os rótulos de todas as linhas do conjunto.
6. Se o rótulo fornecido no endereço é igual ao rótulo especificado para uma dessas linhas temos um *cache hit* e o dado é acessado diretamente da cache.
7. Se for diferente do de todas essas linhas, é um *cache miss*. A cache então fornece o endereço do bloco de memória que contém o endereço de memória especificado pela CPU e solicita o bloco inteiro à memória.
8. A cache escolhe em seguida uma das linhas do conjunto para receber o novo bloco (algoritmo de troca).
9. Transfere então o bloco para a linha escolhida e anota seu rótulo.

Vantagens

- Cada bloco pode entrar em mais do que uma linha, diminuindo a chance de ocorrer thrashing.
- O número de rótulos que precisam ser comparados é pequeno.
- O número de bits por rótulo é pequeno.
- Juntando os dois ítens acima, podemos ter implementação eficiente em *hardware* para essas comparações.
- Os algoritmos de troca podem ser mais simples (menos linhas a escolher).

Algoritmo de troca

Similar a algoritmos de troca de página.

Aleatório Retira uma linha aleatoriamente.

First-In First-Out (FIFO) Retira a linha colocada a mais tempo.

Least Recently Used (LRU) Retira a linha que foi usada a mais tempo.

Least Frequently Used (LFU) Retira a linha que foi usada menos frequentemente desde um certo tempo.

Política de escrita

Quando um dado é modificado na cache, o que acontece com o correspondente dado na memória? (Importante em processamento paralelo.)

Write through Quando escreve na cache, escreve o bloco imediatamente na memória.

Write back Ao escrever na cache, apenas marca a linha como suja. Quando o bloco é carregado, a linha é marcada como limpa. Quando a linha for ser retirada, se ela estiver marcada como suja, escreve no bloco de memória correspondente.

Considerações adicionais

Tamanho do bloco Muito grande: disperdício de espaço na cache.
Muito pequeno: maior sobrecarga de gerenciamento.

Número de caches Caches maiores são mais lentas.

- Caches pequenos e rápidos para acesso rápido.
- Caches grandes para muitos dados.
- Usamos mais de um nível de cache (L1, L2, L3).
- Bom para *multicore* (último nível é compartilhado).

Hit ratio

Definimos o *hit ratio* de um dado código em um dado sistema como *a fração dos acessos à memória que são servidos diretamente pela cache* (são *cache hit*).
O *hit ratio* depende de diversos fatores:

- O código sendo executado (sua localidade de acessos).
- A quantidade de cache.
- A eficiência dos algoritmos da cache para o código.

Modelo simplificado

Dado um *hit ratio* h, sabemos que $0 \leq h \leq 1$. Definimos:

Tempo de *hit* t_h é o tempo que um acesso de memória leva quando for um *cache hit*.

Tempo de *miss* t_m é o tempo que um acesso de memória leva quando é um *cache miss*.

Tempo médio de acesso à memória t_a

$$t_a = ht_h + (1 - h)t_m$$

Limites

Da expressão $t_a = ht_h + (1 - h)t_m$ vemos que:

- Se $h \approx 0$ então $t_a \approx t_m$ e a cache não está tendo efeito.
- Se $h \approx 1$ então $t_a \approx t_h$ e o sistema não vê o tempo de acesso à memória.

Exemplo

Num sistema atual típico, um acesso à cache é da ordem de 1 ou 2 tempos de ciclo da CPU, enquanto um acesso à memória é da ordem de centenas de ciclos. Colocando $t_m \approx 100t_h$ ficamos com

$$t_a \approx ht_h + (1-h)100t_h$$

Usando $t_h = 1$ temos:

h	t_a
0.1	90.10
0.25	75.25
0.5	50.50
0.75	25.75
0.9	10.90
0.95	5.95
0.98	2.98
0.99	1.99

Tópico

1 Estrutura básica

2 Tecnologia

3 Hierarquia de memória

4 Entradas e saídas

5 Multiprogramação

6 Gerenciamento de memória

7 Pipeline de instruções

8 Execução superescalar

Entradas e saídas

- Diferentes tipos de dispositivos.
- Cada um requer interações diferentes com a CPU.
- As velocidades podem ser muito diferentes (ordens de grandeza):
  - Linhas seriais: Da ordem de kilo-bits por segundo.
  - Gigabit ethernet: Da ordem de gigabits por segundo
  - Placas gráficas: ???

Passos (simplificado)

- Escolher o dispositivo (identificador de dispositivo)
- Enviar comando (ex: leitura ou escrita)
- Transmitir os dados

Tipos

I/O Programado A CPU se envolve em cada passo da operação de entrada ou saída, inclusive aguardando o dispositivo ficar pronto para novas operações.

Interrupção Ao invés de aguardar o término da operação, a CPU vai fazer outras atividades, e o dispositivo gera uma interrupção quando a operação anterior terminou. As diversas operações ainda são controladas pela CPU.

DMA Um hardware especial, o *controlador de acesso direto à memória* permite que os dados sejam transferidos entre memória e I/O sem intervenção direta da CPU.

Canais Dispositivos inteligentes que recebem comandos de mais alto nível (ex: operações gráficas complexas em placas gráficas).

Tópico

1 Estrutura básica

2 Tecnologia

3 Hierarquia de memória

4 Entradas e saídas

5 Multiprogramação

6 Gerenciamento de memória

7 Pipeline de instruções

8 Execução superescalar

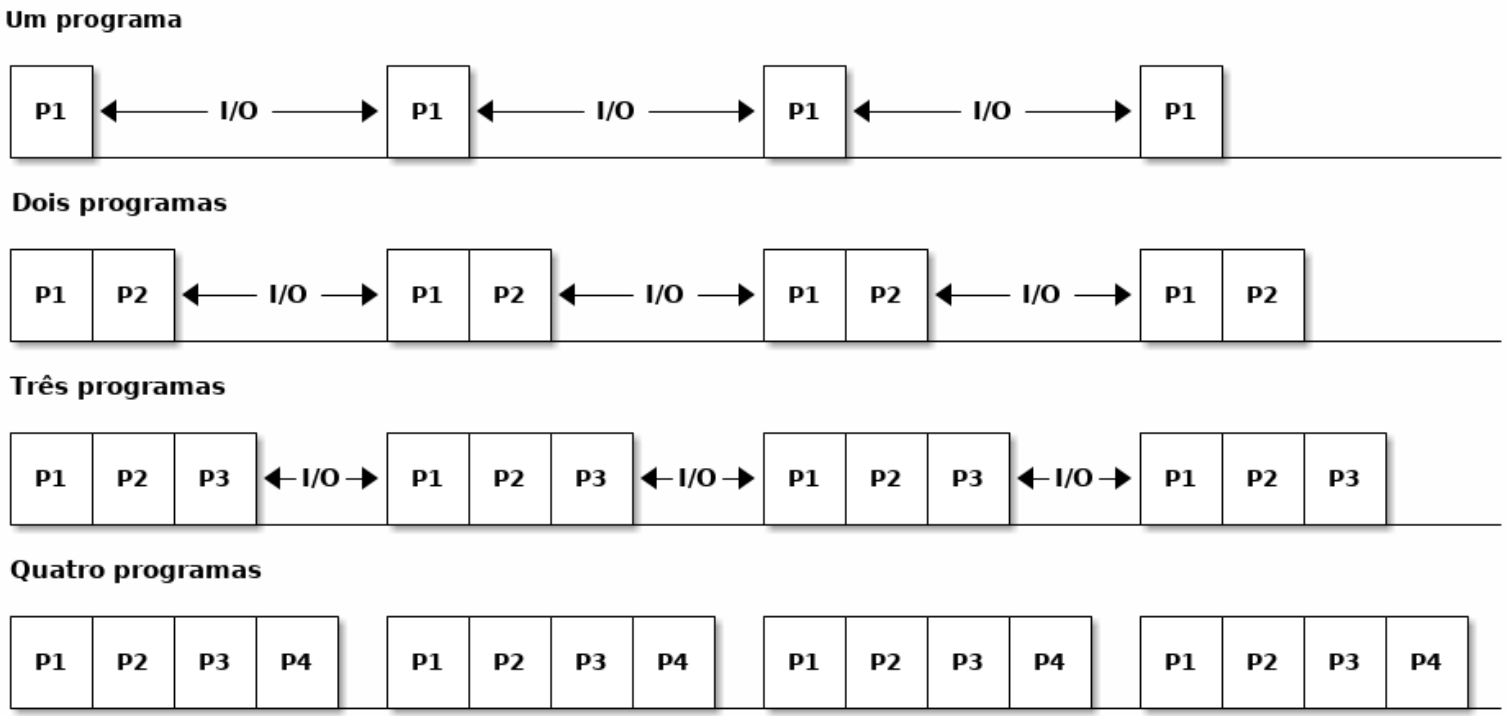
Aproveitamento de CPU

- CPU caras
- Programas precisam fazer I/O frequentemente
- Operações de I/O são demoradas
- Com apenas um programa executando, CPU fica desocupada boa parte do tempo.

Multiprogramação

Manter diversos programas em execução simultaneamente. Quando um deles pára para esperar I/O, outro usa CPU.

Aproveitamento de CPU



Timesharing

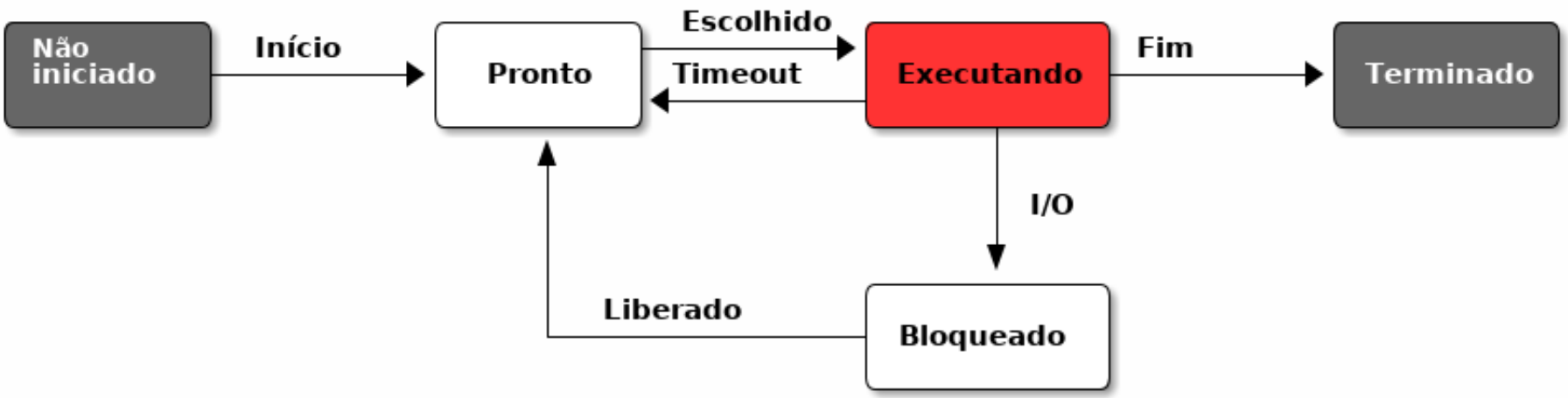
- Já existe multiprogramação.
- Por que não rodar programas de vários usuários?
- Um programa em execução é um processo.
- CPU alterna entre os diferentes processos.
- Cada processo recebe alternadamente uma fatia de tempo (quantum).
- Ao fim do *quantum* o processo é interrompido e outro é escolhido para execução (escalonamento).

Estados dos processos

Processos podem:

- Não ter sido iniciados ainda.
- Já ter terminado.
- Estar executando (usando CPU).
- Estar esperando término de operação de I/O
- Estar pronto para execução, mas sem CPU disponível.

Estados dos processos



Tópico

1 Estrutura básica

2 Tecnologia

3 Hierarquia de memória

4 Entradas e saídas

5 Multiprogramação

6 Gerenciamento de memória

7 Pipeline de instruções

8 Execução superescalar

Uso da memória

- Mais de um processo rodando: precisa dividir a memória entre os processos.
- Precisa também memória para SO.
- Proteção:
  - Proteger memória do SO dos processos de usuários.
  - Proteger memória de um processo dos outros.

Particionamento

A memória precisa ser particionada entre os processos (e SO), com proteção entre as partições.

Particionamento fixo / Tamanhos iguais

- A memória é dividida em pedaços todos do mesmo tamanho.
- Os processos ocupam um desses pedaços.

Particionamento fixo / Tamanhos iguais

- Fácil de implementar.
- Disperdício de espaço ao executar processos que requerem pouca memória.
- Impossibilidade de executar processos que requerem mais memória do que a alocada.

Particionamento fixo / Tamanhos distintos

- A memória é dividida em pedaços de diferentes tamanhos.
- Processos ocupam um desses pedaços.
  - Processos grandes usam pedaços grandes.
  - Processos pequenos usam preferencialmente pedaços pequenos.

Particionamento fixo / Tamanhos distintos

- Melhor uso da memória.
- Possibilita execução de processos maiores.
- Continua havendo disperdício de memória.
- Continua sendo impossível executar processos grandes.

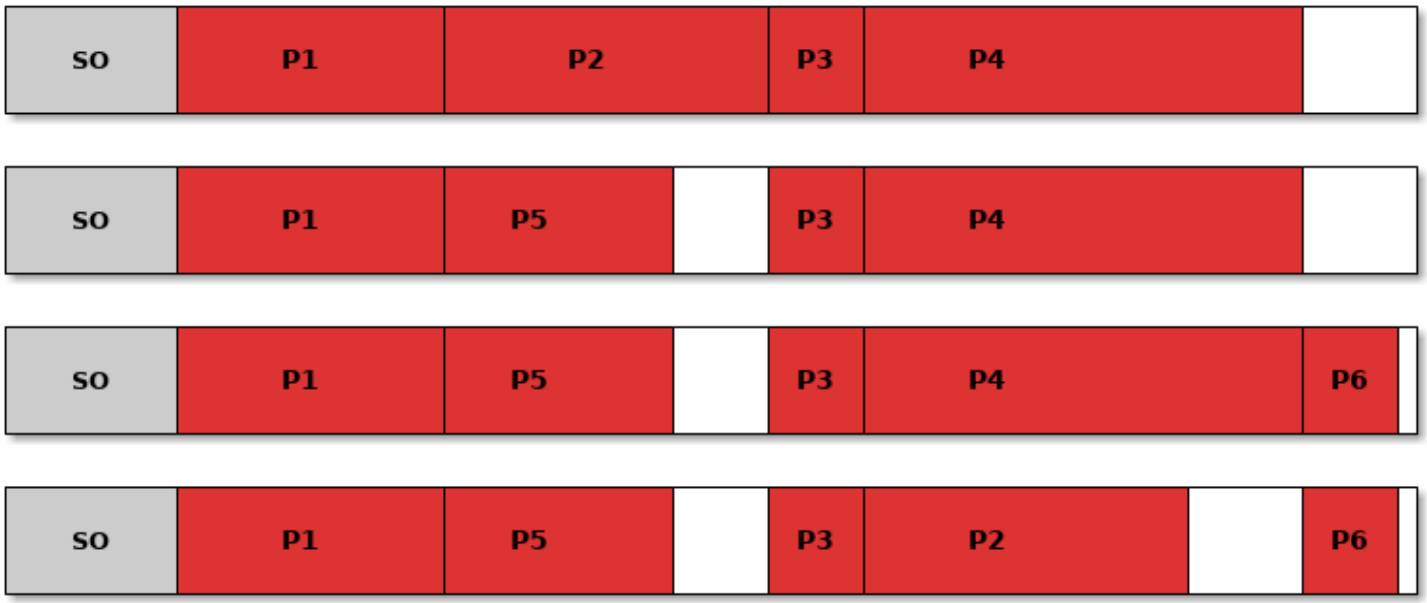
Particionamento variável

- Aloca espaço na memória de acordo com as necessidades do processo.
- Início e final das partições flutua de acordo com as necessidades.

Particionamento variável

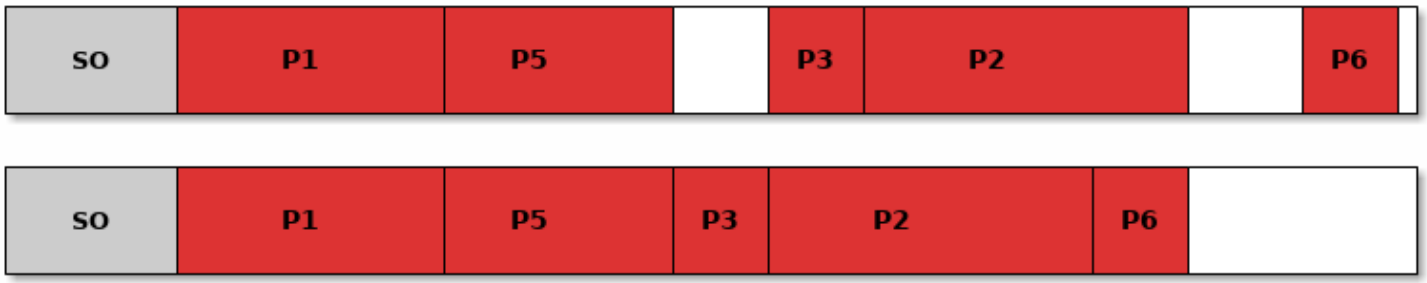
- Evita disperdício (inicialmente).
- Permite rodar processos grandes.
- Gera fragmentação de memória.
- Tem necessidade de relocação de endereços.
- Precisa conhecer o tamanho do processo de antemão.

Fragmentação e relocação



Compactação

Os espaços vazios da memória podem ser compactados regularmente, deslocando os processos para pedaços consecutivos da memória.

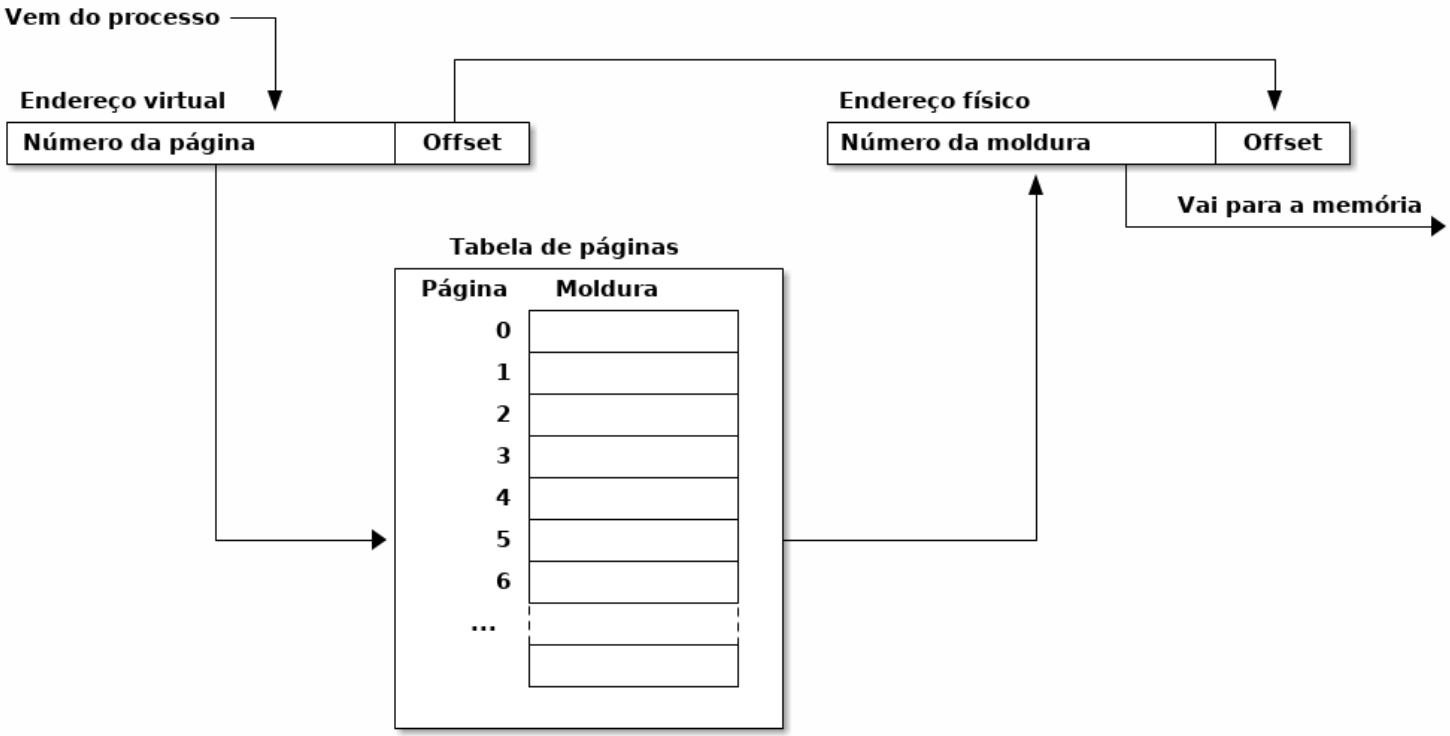


Paginação

- Distinguimos dois tipos de memória:

  Memória física A memória presente no computador.
  Memória virtual A memória referenciada pelos processos (endereços que aparecem no código).
- A memória virtual é dividida em páginas de tamanho fixo.
- A memória física é dividida em molduras de página (do mesmo tamanho que as páginas).
- Cada processo tem sua memória virtual.
- Relação entre página e moldura de página é registrada na tabela de páginas.

Tabela de páginas



Paginação

- O disperdício de memória é pequeno (desde que as páginas não sejam muito grandes).
- Não é necessária realocação no código.
- Molduras associadas a páginas consecutivas não precisam ser consecutivas:
  - Diminui fragmentação (aproveita molduras soltas).
- Processos de qualquer tamanho podem ser executados (desde que haja memória).
- Não precisa saber tamanho de antemão (aloca mais páginas se necessário).
- Não precisa usar espaço de memória para páginas que não são necessárias no momento (paginação sob demanda).
- Permite usar mais memória virtual do que existe memória física.
- Memória virtual excedente precisa ser guardada no disco (swap).

Page fault

- Na paginação por demanda, páginas existentes podem não estar na memória física.
- Se forem acessadas pelo processo, ocorre page fault.
- Tabela de páginas deve indicar quais as páginas que não estão presentes.
- Quando há *page fault*:
  - Se existe moldura livre, carrega página nessa moldura (atualiza tabela de páginas).
  - Se não existe livre, deve ser escolhida alguma para substituição (algoritmo de troca; similar a em *cache*).
- Se uma página que foi recentemente substituida for necessária novamente, ela pode deslocar outra página que pode vir a ser necessária em breve, resultando em thrashing.
  - Usar algoritmos adequados.
  - Usar mais memória.
  - Terminar alguns processos.

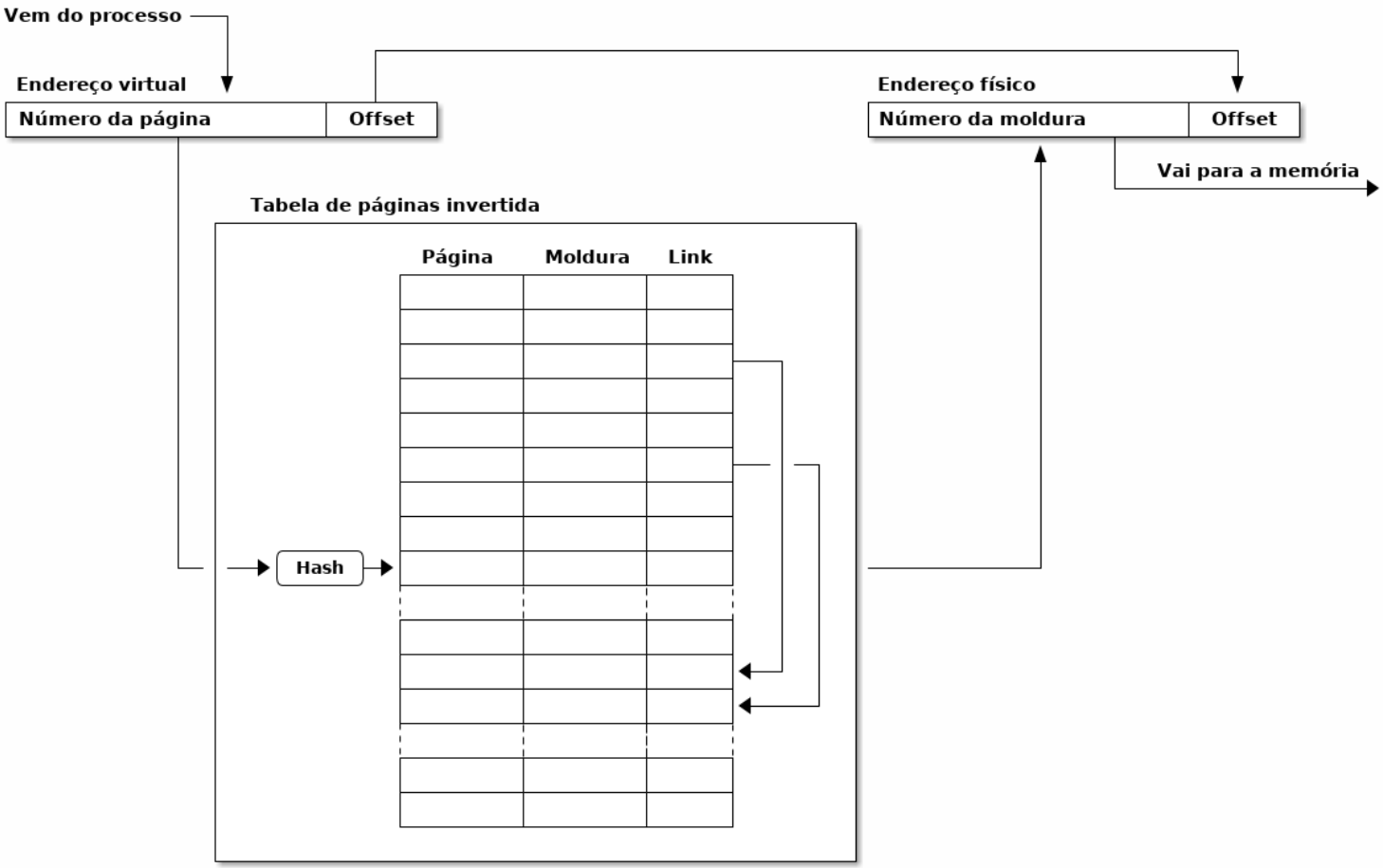
Tabela de páginas invertida

- Representar tabelas de páginas diretamente como indicado não é viável. Exemplos:
  - Endereços de 32 bits.
  - Páginas de 4Kbytes (2^{12} bits para *offset*).
  - Número de página tem 20 bits, portanto total de 2^{20} ou 1 mega-páginas.
  - 20 bits (aproximadamente 3 bytes) para guardar o endereço de cada moldura.
  - Portanto, 3 Mbytes de memória para a tabela de página de cada processo.
  - Refaça agora os cálculos para endereços de 64 bits!
- É usada uma tabela de páginas invertida.

Tabela de páginas invertida

- A tabela de páginas é uma tabela de *hash*.
- Uma função *hash* é aplicada ao número da página (e processo), gerando um número com menos bits.
- Se a entrada correspondente na tabela de *hash* estiver desocupada, isto é um *page fault*.
  - Carrega-se a página solicitada em uma moldura disponível.
  - Marca-se nessa entrada o número da página e o número da moldura correspondente.
- Se a entrada correspondente na tabela de *hash* estiver ocupada, o número da página anotado é comparado com a página solicitada:
  - Se for igual, acessa a moldura indicada.
  - Se for diferente, segue-se um *link* presente nessa entrada para uma outra entrada usada para resolução de colisão no *hash*. Repete até encontrar a página pedida ou chegar ao último *link* de colisão.
    - Se achou a página, acessa a moldura indicada.
    - Se não achou, é um *page-fault*. Carrega-se a página solicitada e cria-se um novo *link* de resolução de colisão para o valor do *hash*.

Tabela de páginas invertida



TLB

- A tabela de páginas é grande.
- Precisa ser armazenada na memória.
- Acesso à memória usa paginação.
- Pode ser necessário acessar mais de uma página para fazer a conversão de endereço virtual para físico.
- Acesso à memória é caro.
- Localidade implica múltiplos acessos à mesma página.
- Usar uma *cache* especial para tabela de páginas.
- *Translation lookaside buffer*, ou TLB.
- Contém os pares página/moldura mais usados recentemente.

TLB e cache

- TLB é usado para converter endereço virtual para físico.
- A cache opera sobre o endereço físico.

Tópico

1 Estrutura básica

2 Tecnologia

3 Hierarquia de memória

4 Entradas e saídas

5 Multiprogramação

6 Gerenciamento de memória

7 Pipeline de instruções

8 Execução superescalar

Execução de instruções

- Instruções são códigos binários (opcodes) armazenados na memória.
- O processador repetidamente busca um *opcode* e executa as instruções contidas nele.
- Normalmente, instruções consecutivas estão armazenadas em posições de memórias consecutivas.
  - Exceções: Desvios (condicionais e repetições) e chamadas de função.

Prefetch

- No prefetch, buscamos o *opcode* da próxima instrução durante a execução da atual.
- Quando a instrução terminar, o *opcode* da próxima já estará disponível (ou faltará menos tempo).
- Limitações de ganho de desempenho:
  - Busca do próximo *opcode* pode colidir com acesso à memória pela instrução corrente.
  - Instrução corrente pode terminar antes da busca do novo *opcode* na memória haver terminado (memória é lenta).
  - Busca do *opcode* pode ser muito mais rápida que execução da instrução corrente (se há cache) e portanto não há muito ganho.
  - O *opcode* buscado pode não ser da instrução que precisa ser executada (desvio).

Pipeline de instruções

- A execução de cada instrução pode ser quebrada em diversas fases. Por exemplo:

  Busca do *opcode* Carrega o *opcode* da memória.

  Decodificação Interpreta o que deve ser executado de acordo com o *opcode*.

  Cálculo de operandos Verifica quais valores devem ser carregados para executar a operação.

  Busca dos operandos Busca esses valores em registradores ou memória.

  Execução da operação Executa a operação desejada.

  Escrita dos resultados Escreve os resultados.

- Essas fases podem ser sobrepostas para instruções consecutivas, num pipeline (linha de montagem).

Pipeline de instruções

- O tempo total da instrução é quebrado nos tempos de cada fase.
- O tempo de ciclo do *pipeline* é o maior tempo das fases.
- A cada tempo de ciclo do *pipeline* as instruções são movidas para a fase seguinte e uma nova instrução é buscada.
- No caso ideal, 6 instruções executadas por tempo de ciclo (neste exemplo).

Desvios

- Quando instrução de desvio (durante a fase de execução) indica um desvio do fluxo normal, instruções semi-processadas posteriores a ela no *pipeline* não serão executadas e devem ser descartadas.
- Isso cria uma bolha no *pipeline*, diminuindo sua eficiência média.
- Operações de desvio são freqüentes no código.

Limites de eficiência

- Com n fases, não é n vezes mais eficiente (mesmo no caso ideal), pois o tempo de ciclo é determinado pelo tempo da fase mais lenta.
- Algumas instruções não precisam de todas as fases.
- Algumas instruções têm execuções mais rápidas.
- Bolhas no *pipeline* são freqüentes.

Previsão de desvio

Para reduzir bolhas no *pipeline* podemos usar algum método de previsão de desvio, onde o processador pode ter informação sobre a direção mais provável das instruções de desvio e buscar os *opcodes* seguintes no local apropriado.

Desvios incondicionais Busca instrução indicada pelo desvio.

Prever desvios como nunca seguidos Busca instrução seguinte, desconsiderando desvio.

Prever desvios como sempre seguidos Busca instrução indicada pelo desvio.

Previsão baseada no *opcode* Escolhe instrução a buscar de acordo com o tipo de desvio.

Desvio atrasado Algumas instruções após o desvio sempre serão executadas.

Flag tomado/não-tomado na CPU Prevê que o desvio vai seguir a mesma direção que da última vez que ele foi executado.

Tópico

1 Estrutura básica

2 Tecnologia

3 Hierarquia de memória

4 Entradas e saídas

5 Multiprogramação

6 Gerenciamento de memória

7 Pipeline de instruções

8 Execução superescalar

Unidades funcionais

- Operações determinadas pelos *opcodes* precisam ser executadas (em geral) em hardware.
- Diversos tipos de operações:
  - Movimento de dados
  - Operações lógicas
  - Desvios
  - Operações aritméticas sobre inteiros
  - Operações aritméticas sobre ponto flutuante
- Operações diferentes requerem hardware diferente.
- O hardware usado para cada tipo de operação é denominado uma unidade funcional (FU).

Superescalar

- A CPU dispõe de diversas FU, mas apenas uma[2] está sendo usada por vez para uma instrução.
- Execução de mais de uma instrução simultaneamente é possível.
- Isto é denominado execução superescalar.
- O paralelismo é denominado paralelismo ao nível de instrução.

[2] No caso básico.

Dependência de dados

- Precisa-se encontrar intruções que possam ser executadas simultaneamente sem introdução de erros.
- Instruções não são sempre independentes.
- Dependências de dados
- Também limitante:

  Conflito de recursos Quando duas instruções que poderiam ser executadas simultaneamente requerem a mesma FU.

Dependência de dados

Vários tipos de dependência:

Dependência verdadeira Quando a instrução posterior precisa de dado fornecido pela instrução anterior.

Dependência procedural Quando a execução da instrução posterior pode ocorrer ou não dependendo do resultado de uma instrução anterior.

Dependência de saída Quando duas instruções escrevem resultado no mesmo local.

Antidependência Quando uma instrução posterior escreve em um local acessado por instrução anterior.

Anterior/posterior

Os termos "anterior" e "posterior" usados acima indicam a ordem das instruções no código seqüencial do programa.

Dependência verdadeira

```
... código anterior
[1] r2 = r1 * 3
[2] r3 = r2 - 1
[3] r2 = r1 + r4
[4] r4 = r2 - 10
[5] if r4 == 0:
[6]    r5 = 0
[7] else:
[8]    r5 = 1
... código posterior
```

Existe dependência verdadeira da linha [2] com a linha [1] e da linha [4] com a linha [3]. [1] tem que executar antes de [2] ou o valor lido em `r2` por [2] será errado; [3] tem que executar antes de [4] ou o valor lido em `r2` será por [4] será errado.

Dependência procedural

```
... código anterior
[1] r2 = r1 * 3
[2] r3 = r2 - 1
[3] r2 = r1 + r4
[4] r4 = r2 - 10
[5] if r4 == 0:
[6]    r5 = 0
[7] else:
[8]    r5 = 1
... código posterior
```

Existe dependência procedural entre das instruções [6] e [8] com a instrução [4], pois dependendo do valor resultante em `r4` uma ou outra delas será executada. Se houver mudança de ordem, o valor de `r5` para o código posterior pode ser errado.

Dependência de saída

```
... código anterior
[1] r2 = r1 * 3
[2] r3 = r2 - 1
[3] r2 = r1 + r4
[4] r4 = r2 - 10
[5] if r4 == 0:
[6]    r5 = 0
[7] else:
[8]    r5 = 1
... código posterior
```

Existe dependência de saída entre as instruções [1] e [3]. Se [1] for executada depois de [3], o valor de `r2` lido na instrução [4] (e possivelmente no código posterior) será errôneo.

Antidependência

```
... código anterior
[1] r2 = r1 * 3
[2] r3 = r2 - 1
[3] r2 = r1 + r4
[4] r4 = r2 - 10
[5] if r4 == 0:
[6]    r5 = 0
[7] else:
[8]    r5 = 1
... código posterior
```

Existe antidependência entre as instruções [2] e [3] (por `r2`) e entre as instruções [3] e [4] (por `r4`). Se [3] for executada antes de [2], o valor de `r2` lido por [2] será errado; se [4] for executada entes de [3], o valor de `r4` lido por [3] será errado.

Considerações

Paralelismo ao nível de instrução O código seqüencial deve possuir instruções que possam ser executadas simultaneamente.

- Quem decide são as dependências.
- O compilador pode ajudar.

Paralelismo de hardware Devem haver FU suficientes para tirar proveito das instruções paralelas.

Ordem das instruções

Existem múltiplas ordens:

- Ordem em que as instruções são carregadas.
- Ordem em que as instruções são executadas.
- Ordem em que os resultados das instruções são escritos.

Ordens

- Ordem de execução (comparada com o carregamento): issue order.
- Ordem da escrita (comparada com o carregamento): compĺetion order

Ordens

Combinações de ordens:

In-order issue, in-order completion instruções são enviadas para execução e completadas na ordem do programa.

In-order issue, out-of-order completion instruções enviadas para execução na ordem do programa, mas podem terminar fora de ordem.

Out-of-order issue, out-of-order completion instruções enviadas e completadas fora de ordem.

In-order issue, in-order compĺetion

- As instruções são enviadas para execução na ordem do programa.
- Múltiplas instruções são enviadas para execução simultaneamente.
- Se necessário, uma instrução aguarda para receber dado da outra que está executando.
- As escritas são feitas na ordem do programa, mesmo que instruções posteriores terminem antes.

In-order issue, out-of-order completion

- As instruções podem terminar (resultados escritos) fora de ordem.
- Permite que instrução posterior complete e libere FU antes do término de instrução anterior.
- CPU precisa lidar com dependência de saída (escritas para mesmo local não podem ocorrer fora de ordem).

Out-of-order issue, out-of-order completion

- Permite que as instruções iniciem execução fora de ordem.
- Decodificação da instrução e execução são desacoplados (não ocorrem em sincronia).
- Entre a decodificação e a execução há um buffer de instruções esperando execução.
- Esse buffer precisa guardar informações de dependências entre as instruções, bem como saber quais as instruções que já tiveram todas as suas dependências executadas (e portanto podem ser executadas).
- Instrução executa assim que aquelas de qual depende já foram executadas e que houver FU livre.
- CPU precisa lidar com antidependência (escrita em um local não pode ocorrer antes do uso do valor anterior nesse local).

Renomeação de registradores

- A dependência verdadeira faz parte da lógica do programa (um dado precisa ter sido calculado para podermos calcular um outro que depende de seu valor).
- A antidependência e a dependência de saída não são verdadeiras: elas ocorrem devido ao reuso de registradores (limitação no número total de registradores na arquitetura).
- Elas podem ser resolvidas com o uso de registradores adicionais.
- O método automático usado para isso é a renomeação de registradores.

Renomeação de registradores

- Os nomes de registradores usados no código são apenas indicações.
- A CPU tem uma grande quantidade de registradores físicos disponíveis.
- Quando um novo valor é atribuido a um nome de registrador pelo programa, a CPU aloca um dos registradores físicos disponíveis para ele.
- Todas as leituras posteriores no código usando o mesmo nome de registrador, antes de uma escrita, acessam o mesmo registrado físico.
- Se uma outra atribuição for feita ao mesmo nome de registrador no código, um outro registrador físico é usado.
- Um registrador físico é liberado quando todas as instruções que precisam de seu valor tiverem terminado a execução.

Exemplos

```
... código anterior
[1] r2 = r1 * 3
[2] r3 = r2 - 1
[3] r2 = r1 + r4
[4] r4 = r2 - 10
[5] if r4 == 0:
[6]     r5 = 0
[7] else:
[8]     r5 = 1
... código posterior
```

```
... código anterior
[1] r2a = r1a * 3
[2] r3a = r2a - 1
[3] r2b = r1a + r4a
[4] r4b = r2b - 10
[5] if r4b == 0:
[6]     r5a = 0
[7] else:
[8]     r5a = 1
... código posterior
```