Arquitetura de Computadores

MiEI – 2016/17

DI-FCT/UNL

Aula 17

Desempenho dos sistemas

- Um sistema tem melhor desempenho que outro se produzir resultados em menos tempo
- A melhoria introduzida num sistema (***speedup***) é dada por:

  S = tempo antigo / novo tempo

  - exemplo: um programa passou a executar em 5s quando antes executava em 6s:

    S = 6/5 = 1,2 (corresponde a 1,2x mais rápido)
  - *S* = 1 significa que não há alteração
  - S = 2 significa que passou a metade do tempo

Influência dos componentes

- Componentes software:
  - Programa (algoritmos e estruturas de dados), linguagens, bibliotecas, S.O., etc…
- Componentes hardware:
  - CPU, Memória, Buses, Periféricos, etc…
- Então… exemplo:
  - Um programa não executa 2x mais rápido só porque usamos um CPU 2x mais rápido!
  - O *speedup* obtido será proporcional ao tempo de execução do CPU no tempo total do programa…
  - os tempos de espera pela memória, periféricos, etc. mantém-se

Exemplo – o que é melhor?

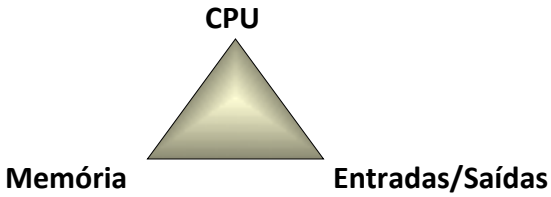
- Considere um sistema que distribui o tempo total de execução nas actividades A e B:

- Se actividade B é tornada 5x mais rápida:

- Se actividade A é tornada 2x mais rápida:

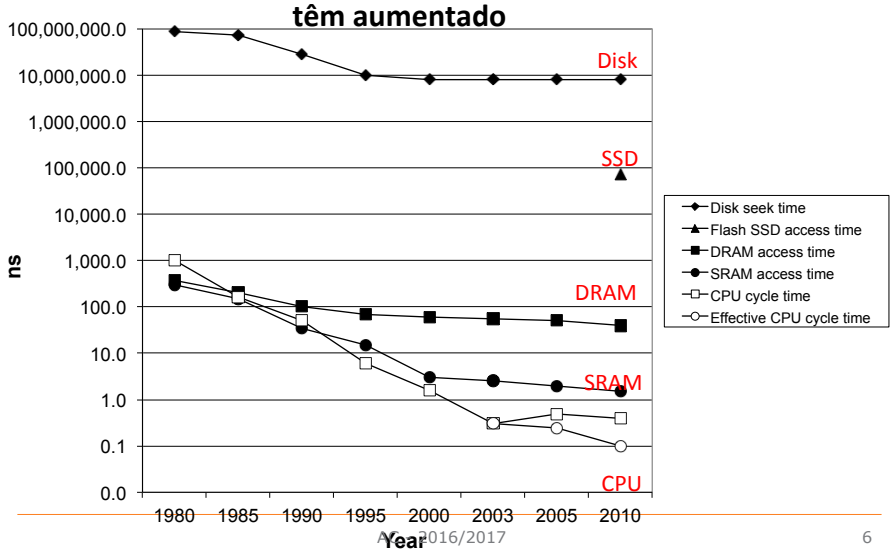
Na arquitectura de computadores

- Procura-se optimizar os vários componentes na medida do respectivo peso nos tempos de execução dos nossos sistemas



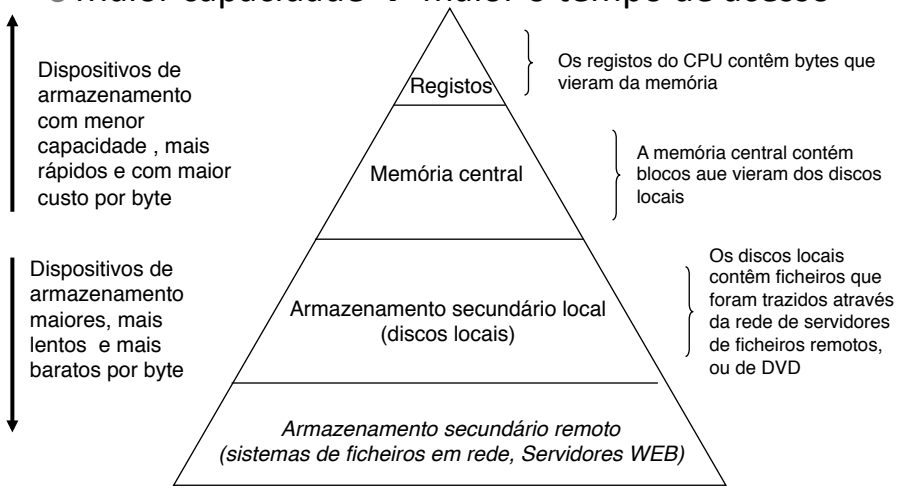
CPU-Memory Gap

As diferenças de velocidade entre RAM ou disco e o CPU têm aumentado



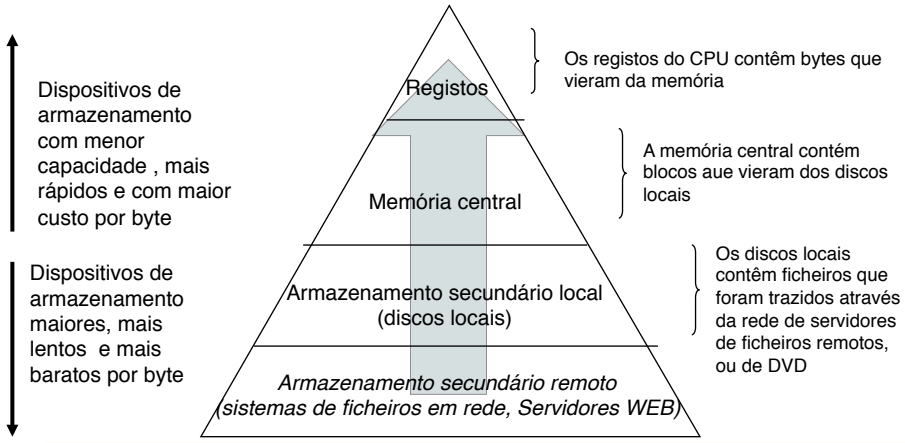
Hierarquia de níveis de memória

- Maior capacidade → maior o tempo de acesso



Hierarquia de níveis de memória

- Procura-se ter os dados a usar na memória mais rápida → mais "perto" do CPU



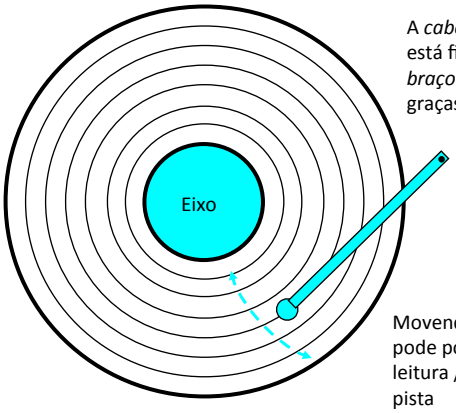
Migração dos dados para "cima"

- Porquê de usar registos? Porque não executar sempre em memória?
- Porquê trazer os programas para memória? Porque não executar diretamente do disco?
- Desempenho vs. possibilidade tecnológica e custos

- O SO tenta trazer para memória o que está no disco
  - As aplicações e os dados que as aplicações leem
- O código nos programas procura tirar o melhor partido dos registos
  - Variáveis são copiadas para os registos

Operação do disco (Visão de 1 só prato)

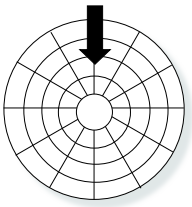
A superfície do disco gira a velocidade constante



A *cabeça de leitura e escrita* está fixa à ponta de um *braço* e voa sobre a superfície graças a uma fina almofada de ar.

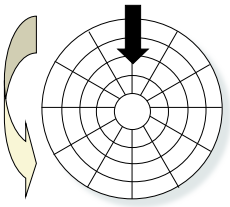
Movendo-se radialment, o braço pode posicionar a cabeça de leitura / escrita sobre qualquer pista

Acesso ao disco



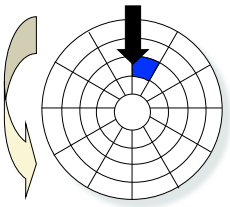
Cabeça em posição sobre uma pista

Acesso ao disco



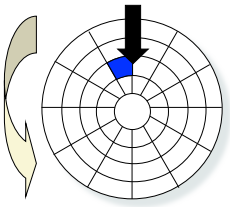
Rotação constante em sentido contrário aos ponteiros do relógio

Acesso ao disco – Leitura



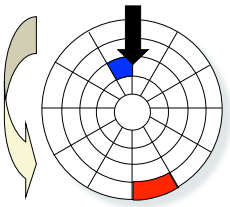
Prestes a ler o sector azul

Acesso ao disco – Leitura



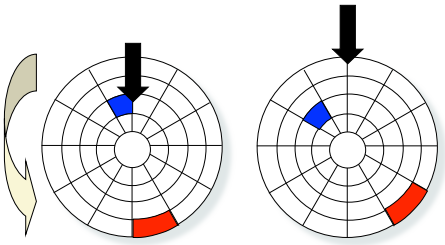
Após ler o sector azul

Acesso ao disco – Leitura



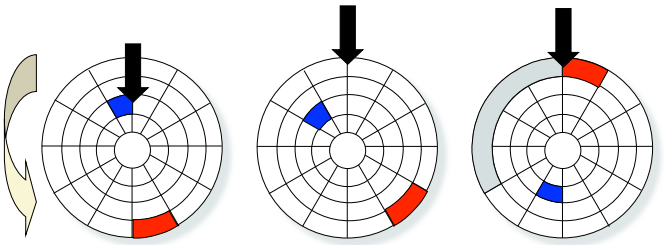
Próximo sector a ler é o sector vermelho (noutra pista)

Acesso ao disco - Seek



Deslocamento (Seek) para a pista do sector vermelho

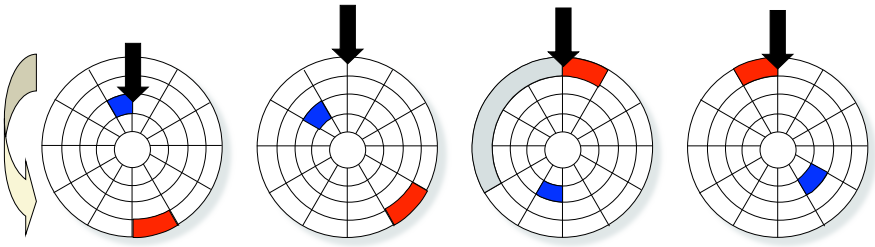
Acesso ao disco – latência de rotação



Depois da leitura BLUE | Seek pelo RED | Latência de rotação

Esperar que o setor vermelho apareça debaixo da cabeça

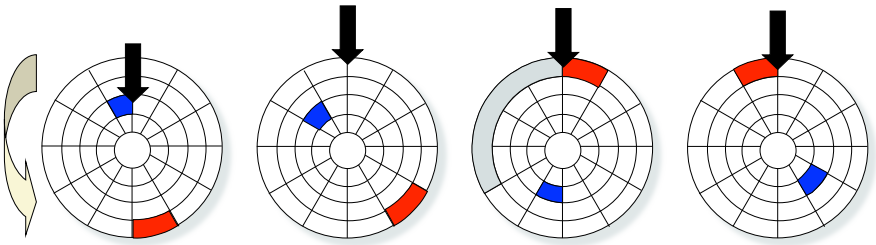
Acesso ao disco - Leitura



Depois de ler o sector AUL | Seek para o VERMELHO | Latência de rotação | Depois de ler o sector VERMELHO

Fim da leitura do sector vermelho

Acesso ao disco – Componentes do tempo de serviço



Depois de ler o sector Azul

Seek pelo Vermelho

Latência de rotação

Depois de ler o sector Vermelho

Transferência de dados

Seek

Latência de rotação

Transferência de dados

Tempo de acesso a um setor

- Tempo médio para acesso a um dado sector pode ser aproximado por:

Tacesso = Tmédio seek + Tmédio de rotação + Tmédio de transferência

- Tempo de seek (Tmédio seek)
  - Tempo para posicionar a cabeça sobre a pista pretendida.
  - Típico: 3 - 9 ms
- Latência de rotação(Tmédio rotação)
  - Tempo de espera até que o 1º bit do sector pretendido passe debaixo da cabeça de r/w.
  - Tmédio de rotação = ½ 60/RPMs
  - Típica velocidade de rotação = 7200 RPMs → 4ms
- Tempo de transferência (Tmédio transferência)
  - Tempo para ler os bits do sector pretendido.
  - Tmédio de transferênca = (1/(média # sectors/pista)) 60/RPM

Exemplo de cálculo de tempo de acesso

- Dados:
  - Velocidade de rotação = 7200 RPM
  - Tempo médio de seek = 9 ms.
  - No. médio de setores/pista = 400.
- Determina-se:
  - Tmédio de rotação = 1/2x(60/7200 RPM) = 4 ms.
  - Tmédio de transferência = 1/400 sets/pista x 60/7200 RPM= 0,02 ms
  - Taccesso = 9 ms + 4 ms + 0,02 ms
- Pontos importantes:
  - Tempo de acesso dominado pelo seek time e pela latência de rotação.
  - 1º bit de um sector é o que demora mais, o resto é de "graça".
  - Acesso a sectores consecutivos poupam no seek e rotação
  - O disco pode ser 100 000x mais lento que a memória central

Blocos lógicos no disco

- Os discos modernos apresentam uma versão abstracta e simplificada da geometria do disco:
  - Os sectores disponívies são vistos como uma sequência de blocos lógicos (0, 1, 2, ...)
  - **LBA - *Logical block addressing***
- Mapeamento entre blocos lógicos e blocos físicos
  - Mantido por *hardware/firmware* no controlador do disco
  - Converte pedidos de blocos lógicos em triplos ordenados (superfície, pista, sector).

Tendências

SRAM

Métrica	1980	1985	1990	1995	2000	2005	2010	2010:1980
$/MB	19 200	2 900	320	256	100	75	60	*320x*
access (ns)	300	150	35	15	3	2	1.5	*200x*

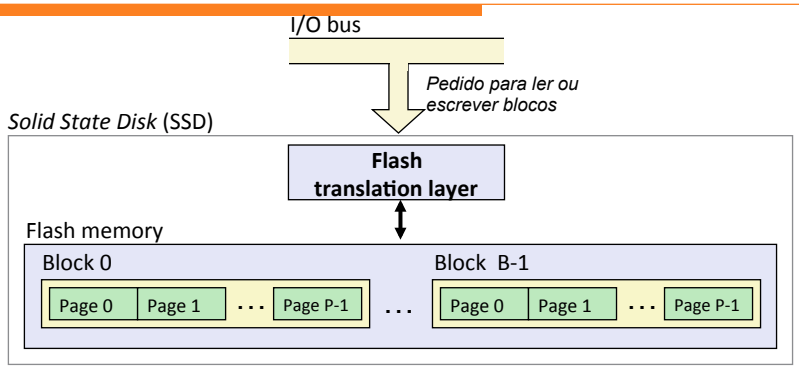
DRAM

Metric	1980	1985	1990	1995	2000	2005	2010	*2010:1980*
$/MB	8 000	880	100	30	1	0.1	0.06	*130 000x*
access (ns)	375	200	100	70	60	50	40	*9x*
typical size (MB)	0.064	0.256	4	16	64	2 000	8 000	*125 000x*

Disk

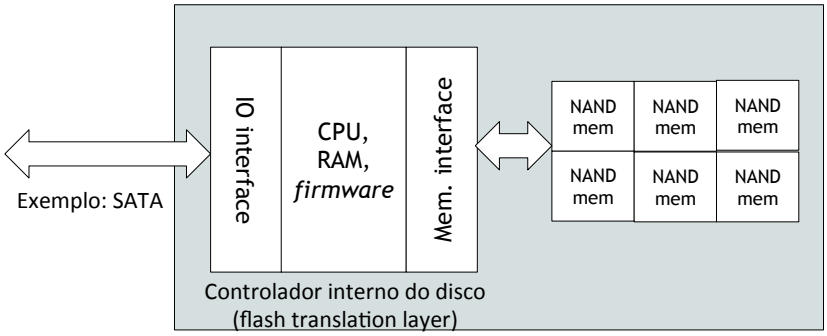
Metric	1980	1985	1990	1995	2000	2005	2010	*2010:1980*
$/MB	500	100	8	0.30	0.01	0.005	0.0003	*1 600 000x*
access (ms)	87	75	28	10	8	4	3	*29x*
typical size (MB)	1	10	160	1000	20 000	160 000	1 500 000	*1 500 000x*

Solid State Disks (SSDs)



- Os dados estão em páginas (eq. setores), dentro de blocos
- As escritas só são possíveis em páginas "apagadas"
- Uma página só pode ser apagada, apagando todo o bloco
- Um bloco fica inutilizado após ser apagado ~100 000 vezes.

Componentes internos ao Disco



Solid State Disks (SSDs)

Leituras sequenciais	300 MB/s	Escritas sequenciais	250 MB/s
Leituras aleatórias	170 MB/s	Escritas aleatéorias	70 MB/s
Tempo de acesso leitura	20µs	Tempo de acesso escrita	200µs

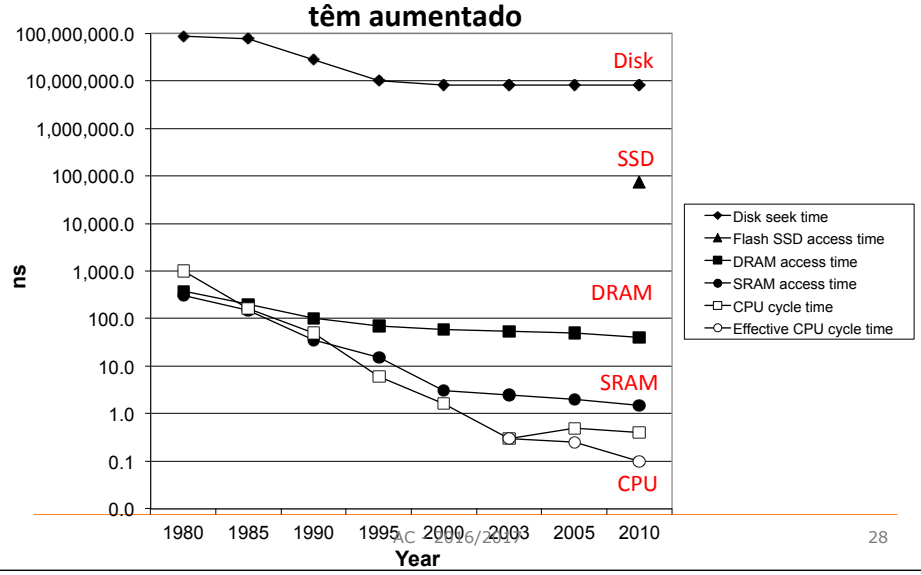
- A escrita de uma página **pode** exigir apagar o bloco
- Apagar um bloco é lento (cerca de 1 ms)
- A escrita de uma página pode desencadear a cópia de todas as páginas em uso no bloco:
  1. Escolher um bloco livre e apagá-lo se necessário
  2. Copiar as páginas a manter do bloco antigo para o novo
  3. Escrever a página nesse bloco
  4. Marcar o bloco antigo como livre
- Esta gestão é feita pelo *firmware* interno ao disco

Comparação entre SSD e discos magnéticos

- Vantagens
  - Sem partes móveis→ maior rapidez e robustez, menos consumo
- Desvantagens
  - Tem o potencial para se gastarem
    - Minimizado pela "*wear leveling logic*" na *flash translation layer*
    - E.g. Intel X25 garante 1 petabyte (1015 bytes) de escritas aletórias até que o disco se torne inútil
  - 100 x mais caro por byte (tem vindo a diminuir)

CPU-Memory Gap

As diferenças de velocidade entre RAM ou disco e o CPU têm aumentado



Hierarquia de níveis de memória

- Maior capacidade → maior o tempo de acesso

Localidade

- Princípio da localidade: Os programas têm padrões conhecidos de acesso a memória
  - **Localidade temporal**: items referenciados recentemente tendem a voltar a sê-lo.
  - **Localidade espacial**: items com endereços próximos tendem a ser referenciados em conjunto.

Exemplo:

- **Dados**
  - Elementos do array são referenciados em sequência: **Localidade espacial**
  - `sum` referenciado em cada iteração: **Localidade temporal**
- **Instruções**
  - Instruções referenciadas em sequência: **Localidade espacial**
  - Ciclo : **Localidade temporal**

```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
return sum;
```

Localidade (1)

- Esta função tem boa localidade?

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Resposta: Sim. Os compiladores de C guardam as matrizes, na memória, linha a linha. Neste caso o percurso na matriz é feito linha a linha. Portanto faz-se sempre acesso a posições de memória contíguas.

Localidade (2)

- Esta função tem boa localidade?

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum
}
```

Resposta: Não. Neste caso o percurso da matriz é feito coluna a coluna; a sequência é a[0][0], a[1][0] …, a[M][0], a[0][1] … Portanto dois acessos consecutivos não fazem acesso a posições de memória contíguas.

Tecnologia de memória

- Todas as memórias rápidas são baseadas em semicondutores
- *Dynamic Random Access Memory* (DRAM)
  - DRAM oferece a melhor relação preço/desempenho, mas necessita de refrescamento periodicamente e após cada leitura
- *Static Random Access Memory* (SRAM)
  - SRAM é mais rápida mas muito mais cara, consome mais energia e ocupa mais espaço
- A memória RAM dos computadores é DRAM

Sumário SRAM vs DRAM

	Transistores por bit	Tempo acesso	Precisa refresh?	Custo	Utilização
SRAM	4 or 6	1X	Não	100x	Memória cache
DRAM	1	10X	Sim	1X	Memória central, frame buffers

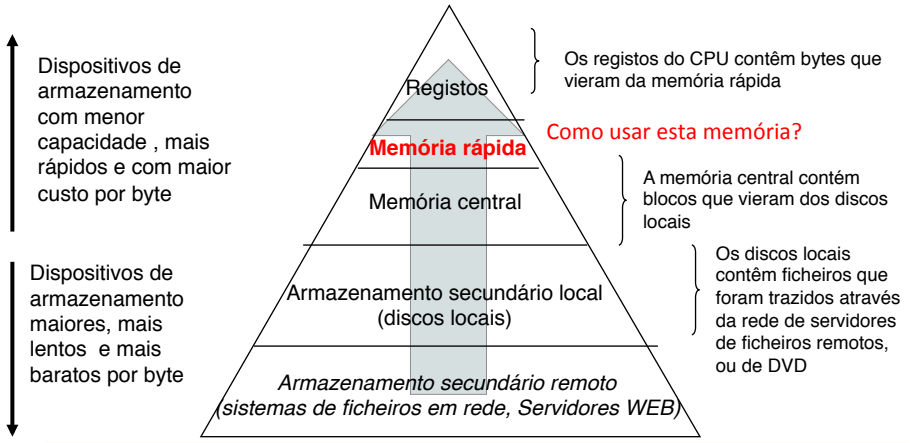
- DRAM e SRAM são memória volátil
  - Perdem informação se a energia é desligada.

Hierarquia de memória

- Propriedades fundamentais:
  - As tecnologias mais rápidas custam mais por byte e têm menor capacidade.
  - O fosso entre o CPU e a memória central está a aumentar.
  - Os bons programas têm boa localidade.
- Estas propriedades suportam:
  - abordagem do armazenamento como uma hierarquia de memória
  - Introduzir mais um nível de memória para tirar partido da localidade: queremos memória central DRAM com o desempenho da SRAM

Hierarquia de níveis de memória

- Poderemos ter mais um nível para que o programa que está a executar seja mais rápido?



Ideia da *Cache*

- **Cache**: Um dispositivo "invisível" que atua como zona temporária para armazenamento de dados de outro dispositivo mais lento mas maior.
- Ideias fundamentais:
  - No nível L, o dispositivo menor e mais rápido, guarda parte do conteúdo do dispositivo maior e mais lento do nível L+1
  - Baseado na localidade, procura-se ter no nível L os dados de L+1 mais prováveis de ser brevemente usados
  - A atualização dos dados em L deve ser transparente para o utilizador desse nível

Utilização da ideia

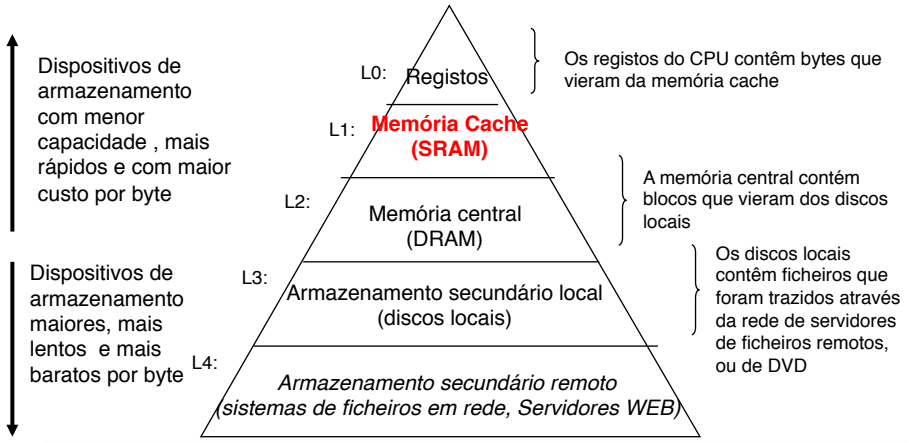
- O mesmo princípio é explorado a outros níveis
- Exemplo dos *browsers Web*:
  - Procuram manter no disco local os dados remotos
  - Mantém em memória os dados que estão a usar ou que podem vir a usar brevemente

Ideia da Cache

- Vantagens esperadas:
  - A localidade leva a que a maior parte dos acessos (leitura/escrita) sejam aos dados já no nível L em vez de ter de ao nível L+1.
  - O armazenamento no nível L+1 pode ser mais lento, e assim mais barato e muito maior.
  - Efeito global: Um grande conjunto de memória que custa o preço da que se encontra no nível mais baixo, mas que permite o acesso aos dados a um ritmo semelhante à do nível mais elevado.

Hierarquia de níveis de memória

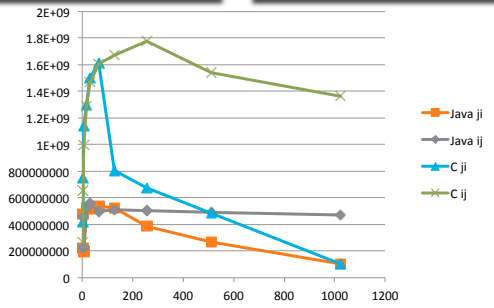
- Poderemos ter um, 2, ou 3 níveis de memória cache, gerida pelo *hardware*



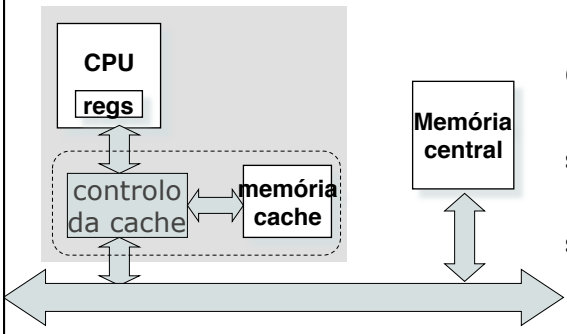
Desempenho (somas/s)

```
/* ij */
sum = 0.0;
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum += a[i][j];
  }
}
```

```
/* ji */
sum = 0.0;
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum += a[i][j];
  }
}
```



Organização da cache de memória

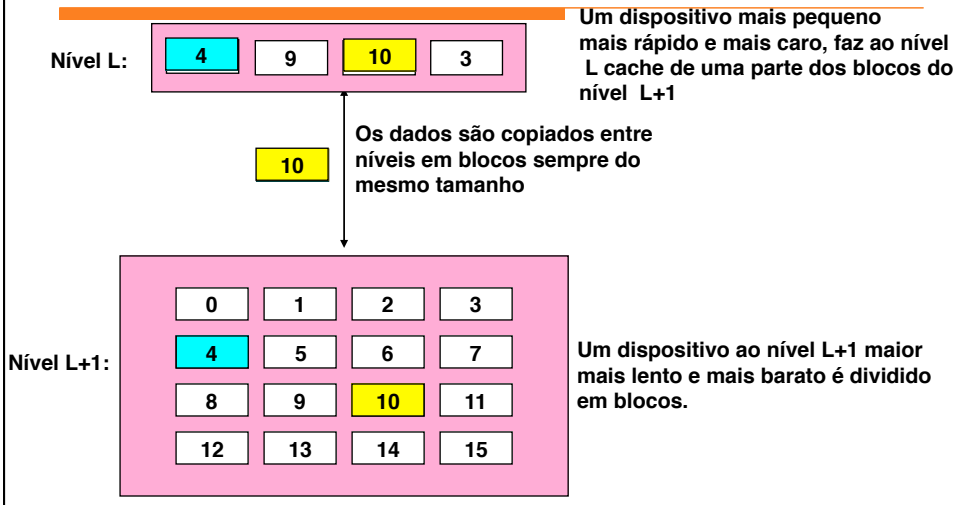


- Funcionamento:

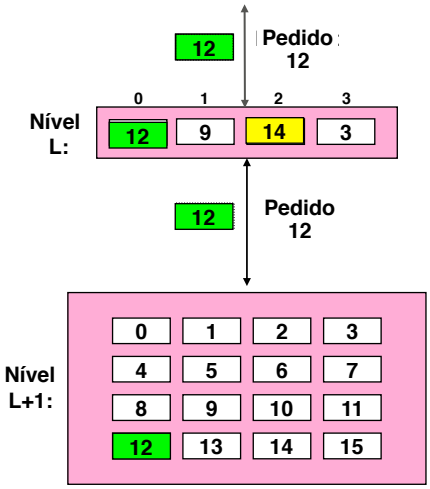
CPU referencia memória:

se *está em cache*
  acede à cache
senão
  acede à memória
  e actualiza cache

Caching na hierarquia de memória



Conceitos gerais da cache



- O programa requer um dado b.
- Cache hit
  - O programa encontra b na cache a nível L. E.g., bloco 14.
- Cache miss
  - b não está no nível L, então a cache de nível L deve obtê-lo no nível L+1. E.g., bloco 12.
  - Se o nível L está cheio, então é preciso escolher uma vítima:
    - Qual o bloco a ser escolhido como vítima?

Conceitos gerais da cache: hit/miss

- ***Cache hit***
  - O programa encontra o dado pretendido na cache (nível L). Não é preciso fazer nada!
- ***Cache miss***
  - Não enconta o que pretende no nível L, então a cache de nível L deve obtê-lo no nível L+1 (e assim sucessivamente).
  - Se o nível L está cheio, então é preciso arranjar espaço, escolhendo uma *vítima*:
    - Qual o bloco a ser escolhido como vítima (ao acaso? LRU - *least recently used* ?)
    - Se o bloco vítima está *limpo*, isto é, **não foi alterado** desde que veio para o nível L, posso carregar o novo bloco por cima
    - Se o bloco vítima está *sujo*, isto é, **foi alterado**, é preciso escrevê-lo no nível L+1 antes de o substituir

Conceitos gerais duma cache

- Taxa de sucesso (*hit ratio*)
  - *h* = percentagem de vezes que o dado pretendido está na cache:

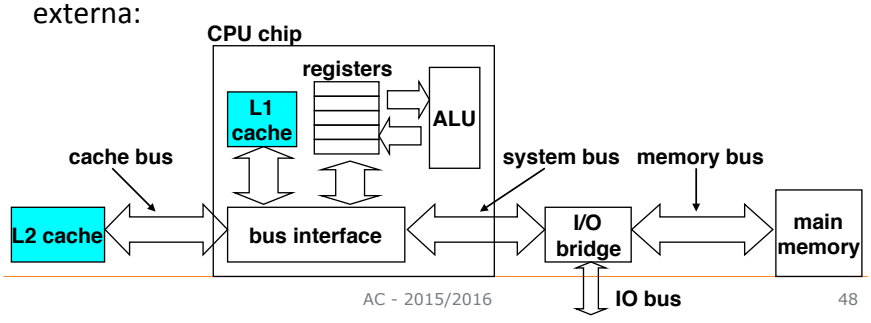
    h = núm. cache hit / núm. total acessos
- Tempo de acesso médio:
  - T_L é o tempo de acesso no nível L
  - T_{L+1} é o tempo de acesso no nível L+1
  - Tempo de acesso médio: Ta

    $Ta = h * T_L + (1 - h) * T_{L+1}$

- Exemplo:
  - *T cache* = 2 ns
  - *T mem* = 8 ns
  - *Hit ratio* = 80%
- Ta = $h * T_L + (1 - h) * T_{L+1}$
  - Ta = 0,8 x 2 + 0,2 x 8 = **3,2 ns**

Cache – Exemplo com dois níveis

- Cache são memórias rápidas (SRAM) geridas automaticamente pelo *hardware*.
- CPU procura em L1, depois em L2, finalmente na memória central
  - Podem introduzir algum *overhead,* mas desprezável
- Exemplo de arquitetura com L1 interna ao chip do CPU e L2 externa:



Hierarquia de níveis de memória