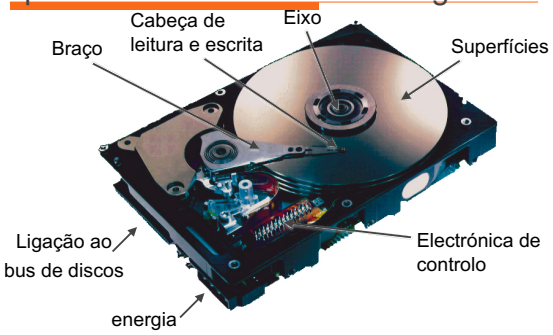
Arquitetura de Computadores

MIEI – 2017/18
DI-FCT/UNL
Aula 18

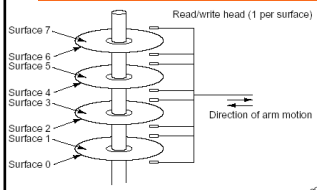
Unidade de transferência: byte-a-byte

- Exemplos: porta série (RS232), teclados, etc
- Um teclado, disponibiliza um byte com o carácter premido; a porta série transfere um único byte
- O software tem de manipular cada um desses bytes individualmente
- Podemos "ver" o teclado como um periférico de leitura que providencia uma sequência (*stream*) de bytes
- Podemos "ver" a porta série como um periférico de leitura e escrita de sequências de bytes

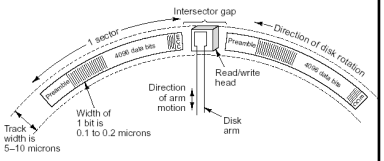
O que há dentro de um disco magnético?



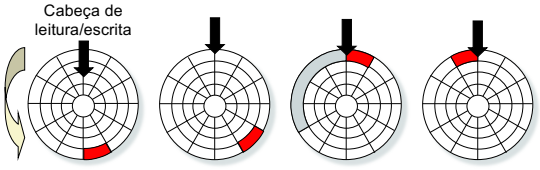
Discos – periféricos orientados ao bloco



- A unidade de transferência é o sector (512bytes, 1K, etc)



Componentes do tempo de serviço



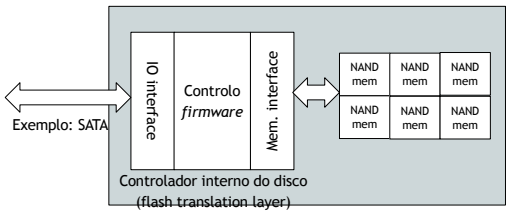
Exemplo de cálculo de tempo de acesso

- Dados:
  - Velocidade de rotação = 7200 RPM
  - Tempo médio de seek = 9 ms.
  - No. médio de setores/pista (spp) = 400.
- Determina-se:
  - Tmédio de rotação = (60/7200 RPM)/2 = 4 ms.
  - Tmédio de transferência = (60/7200 RPM)/400 spp = 0,02 ms
  - Tmédio acesso = 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

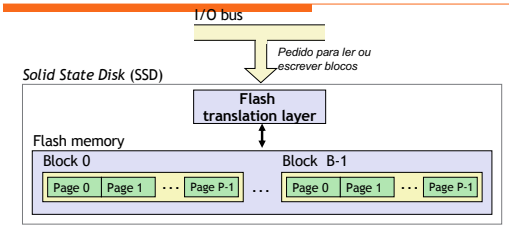
Blocos lógicos no disco

- 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).

Solid State Disks (SSDs)



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.

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)

Unidade de transferência: por blocos

- Exemplo: disco, DVD, pen, etc
- O periférico só permite leitura/escrita de um bloco de bytes (512 bytes, 1K, etc)
- Podemos "ver" estes periféricos como endereçáveis mas orientados a blocos
  - Cada endereço representa um bloco
- Como o software copia esses blocos de memória para o respectivo controlador e vice-versa?
  - Em bytes/words etc por espera ativa?
  - Em bytes/words etc usando interrupções por cada um?

Transferência de blocos de bytes

- Periféricos orientados ao bloco. Exemplo: enviar 1000 bytes

```
enviar_bloco()
    por_bloco_num_buffer()
    ligar_transf_intr()
    aguarda_transferencia_p_controlador()
    manda_controlador_escrever_bloco()

RotinaServiço:
    envia prox. byte (ou word ou dword) do buffer p/ controlador
    se último, desliga_transf_intr()
```

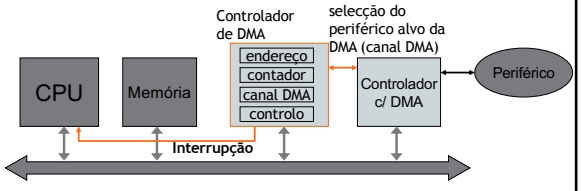
O CPU pode ser interrompido até 1000 vezes para executar a RotinaServiço !

Acesso directo à memória *Direct Memory Access* (DMA)

- As E/S usando interrupções ainda requerem uma intervenção ativa do CPU nestes periféricos
  - Para um bloco de dados o CPU precisa de intervir muitas vezes
  - A resposta a estes problemas é dispensar, nestes casos, a necessidade de interrupções a cada byte/word/dword
- Um novo dispositivo *hardware* que permita transferências entre Memória e Controladores sem intervenção do CPU: Acesso Direto à Memória

Controlador de DMA. Exemplo

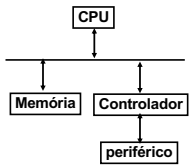
- Registos de um controlador genérico:
  - Endereço – endereço de memória
  - Contador – número de bytes a transferir
  - Canal DMA – *(DMA channel)* identifica o periférico alvo
  - Controlo – comandos, por exemplo: sentido da transferência



Operação com DMA

- O CPU programa o controlador de DMA com a transferência pretendida:
  - Leitura da memória ou escrita em memória
  - Endereço (canal de DMA) do controlador do periférico
  - Endereço do bloco de memória central, onde estão/para onde vão, os dados a transferir
  - Número de bytes a transferir
- O CPU volta ao processamento
- O controlador de DMA trata da transferência com o controlador do periférico
- O controlador de DMA pode "tomar conta" do bus (bus master) e aceder à memória e ao periférico
  - Compete com o CPU pelo uso do BUS
- O controlador de DMA envia uma interrupção quando termina toda a transferência (ou em caso de erro)

Transferência de dados baseada em interrupções sem DMA



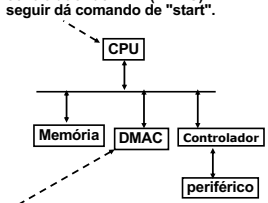
Os processos são parados (o CPU) durante a transferência (CPU a 1GHz e rot. tratamento com 100 instruções a 1 inst/Hz, 1000bytes):

1000 interrupções ➔ 1000 x a rotina de serviço custa ao CPU : 1 000x100 = 100 000 Hz

No melhor dos casos transfere $10^9/100 = 10^7$ = 10 Mbytes/s (mas depende também da velocidade do bus, periférico e memória)

Direct Memory Access

O CPU envia a direcção, end. inicial e nº de bytes ao controlador de DMA (DMAC). A seguir dá comando de "start".



DMAC faz a transferência e interrompe só no fim

Suponhamos que a preparação do DMA também "custa" 100 instr. A transferência de 1000bytes custa ao CPU: 100+100 = 200 instr.

A taxa máxima de transferência depende do bus, memória, do controlador de DMA e do periférico.
O CPU está livre para executar os programas.

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?

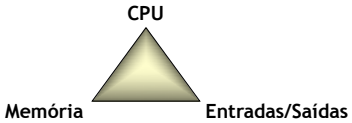
- Considere um sistema que distribui o tempo total de execução nas actividades A e B:

- Se actividade B é tornada 5x mais rápida:

- Se actividade A é tornada 2x mais rápida:

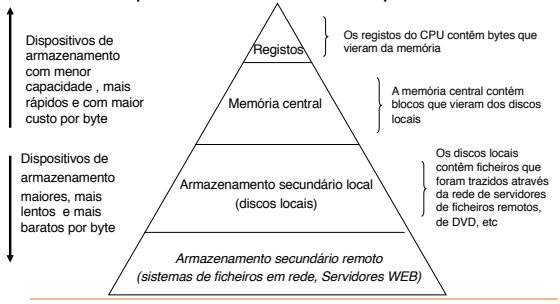
Na arquitectura de computadores

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



Hierarquia de níveis de memória

- Maior capacidade → maior o tempo de acesso



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 periódico 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

Tendências

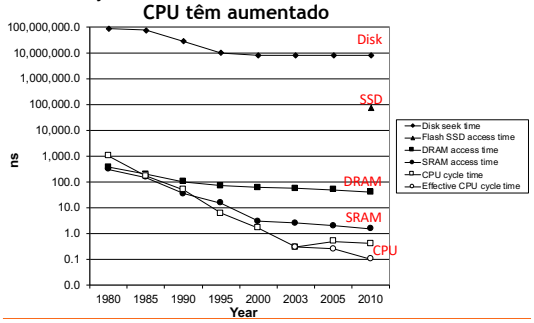
SRAM	1980	1985	1990	1995	2000	2005	2010	2010:1980
$/MB	19 200	2 900	320	256	100	75	60	*320x*
access (ns)	300	150	35	15	3	2	1.5	*200x*

DRAM	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	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*

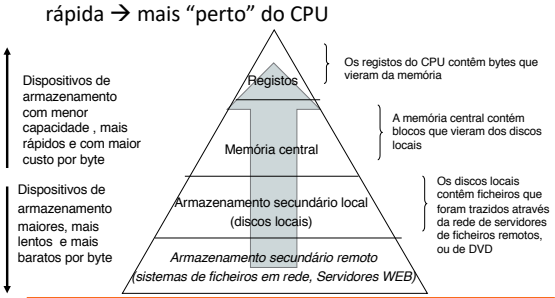
CPU-Memory Gap

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



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

Princípio da Localidade

- Os programas têm padrões 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:

Instruções **Localidade temporal**

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
}
```

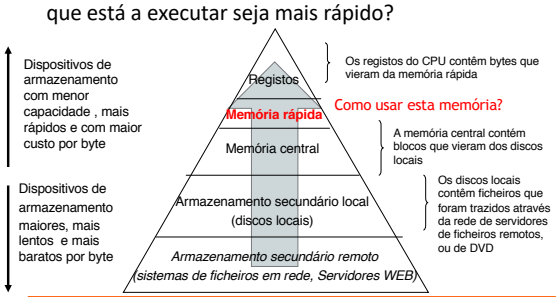
Resposta: Não. Neste caso o percurso da matriz é feito coluna a coluna; a sequência é a[0][0], a[1][0] …, a[M][0], a[0][1] … Portanto dois acessos consecutivos não fazem acesso a posições de memória contíguas.

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 serem brevemente usados
  - A atualização dos dados em L deve ser transparente para o utilizador desse nível

Utilização da ideia

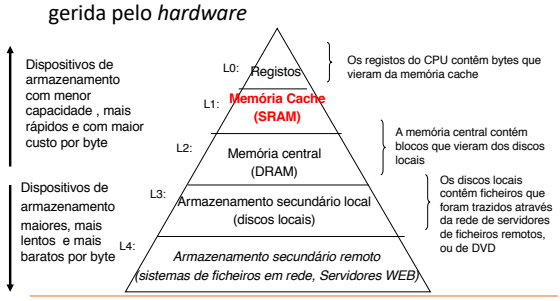
- O mesmo princípio é explorado a outros níveis
- Exemplo dos *browsers Web*:
  - Procuram manter no disco local os dados remotos
  - Mantém em memória os dados que estão a usar ou que podem vir a usar brevemente

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 ir 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 de custo semelhante ao nível mais baixo, mas que permite o acesso a um ritmo semelhante à do nível mais elevado.

Hierarquia de níveis de memória

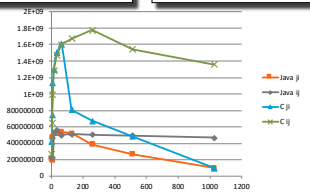
- Poderemos ter um, 2, ou 3 níveis de memória cache, gerida pelo *hardware*



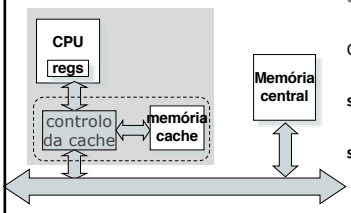
Desempenho (somas/seg)

```
/* 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];
  }
}
```



Cache de memória



- Funcionamento:

CPU referencia memória:

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