PSCF 02

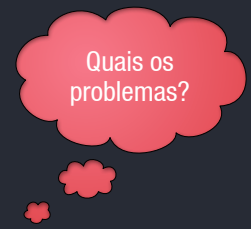
Memória

Performance em Sistemas Ciberfísicos
Escola Politécnica - PUCPR

© Prof. Luiz Lima Jr.

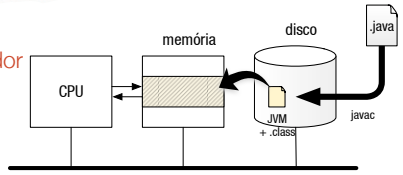
Memória

```java
static class A {
    private int m = 100;
    public void imprime() { System.out.println(m); }
}

static public void main(String [] args) {

    A a = new A();
    A b = null;

    int [] v = new int[100];
    for (int i=0; i<=100; ++i)
        System.out.println(i + " ->" + v[i]);

    a = b;
    a.imprime();

    f(100, v);

    System.out.println("FIM");
}
```



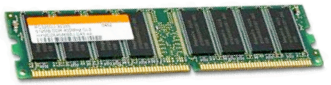

```java
static void f(int n, int [] v) {
    if (n > 0) {
        int [] v2 = new int[v.length * 2];
        for (int i=0; i<2*n; ++i)
            v2[i] = v[i/2];
        f(n-1,v2);
    }
}
```

Introdução



- Memória:
  - um dos mais importantes recursos do computador ➔ requer cuidados especiais.
  - O processo precisa estar na memória para ser executado:
    - **"arquitetura von Neumann"**
- A quantidade de memória exigida pelos processos tem crescido rapidamente:
  - década de 80:
    - **universidades usavam sistema de tempo compartilhado com 4 MB**
  - hoje:
    - **Microsoft Office: 4GB (recomendado, 64 bit)**

Introdução

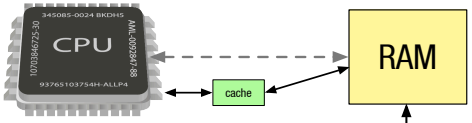


- Memórias têm **capacidades**:
  - expressas em *número de bytes*
- **"Palavras"**:
  - 8, 16 ou 32 bits
    - **(e.g., bits para 1 instrução)**
- **Endereços** de memória:
  - unidades endereçáveis:
    - **palavras**
    - **bytes, mais tipicamente**
  - 2^k endereços ➔ endereços de k bits
- **Acesso**:
  - aleatório (qualquer posição diretamente)
- **Desempenho**:
  - taxa de transferência:
    - **(e.g., bytes por segundo)**

Hierarquia de Memórias

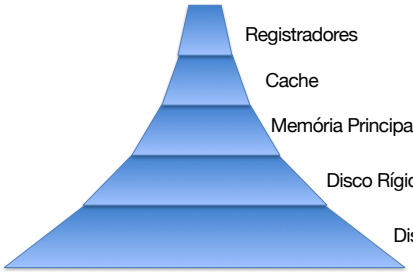
- Para desempenho:
  - memória deve acompanhar desempenho da CPU
  - "*trade-off*": **capacidade**, **taxa** de transferência, **custo**
- "Hierarquia" de memórias:
  - grande quantidade de **memória principal** (RAM)
    - **volátil, velocidade e custo médios**
  - pequena quantidade de memória **cache** ⟹ gerenciamento ao nível de HW
    - **volátil, muito rápida, cara (ex: i7: 24MB de cache)**
  - **memória secundária** (e.g., discos, memória *flash*, *solid-state drives*)
    - **não volátil, dezenas de centenas de GB, velocidade e custo baixos**
- O Sistema Operacional deve:
  - Coordenar a utilização destas memórias
  - Abstrair esta hierarquia em um modelo útil

localidade de referência



CPU é ordens de magnitude mais rápida que memória RAM!

Hierarquia de memória



Registradores	desconsiderado
Cache	~1 ns latência (~25.000.000MB/s)
Memória Principal	~12.5 ns latência (~25.600MB/s)
Disco Rígido	~7.000.000 ns latência (~70MB/s)
Disco Ótico, Pendrive, ...	

Maior custo
Maior banda
Menor latência

Menor custo
Menor banda
Maior latência
Maior armazenamento

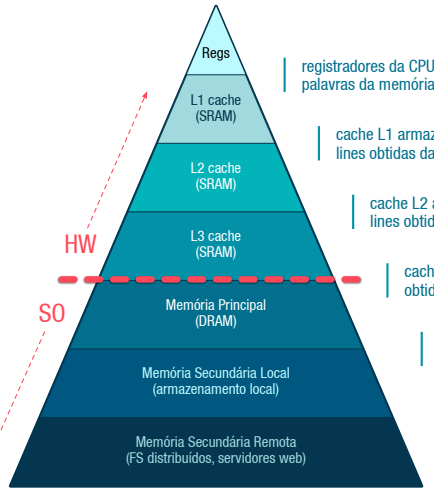
Hierarquia de Memórias

menor, mais rápida, mais cara

maior, mais lenta, mais barata

HW

SO



registradores da CPU armazenam palavras da memória cache

cache L1 armazena cache lines obtidas da cache L2

cache L2 armazena cache lines obtidas da cache L3

cache L3 armazena cache lines obtidas da memória principal

memória principal armazena páginas obtidas do armazenamento local

memória secundária armazena arquivos obtidos

Princípio fundamental em sistemas ciberfísicos:

"Se você entende como o sistema **move dados** na hierarquia de memória, então você pode escrever programas cujos **dados são armazenados mais próximos da CPU** na hierarquia, onde o processador poderá acessá-los mais rapidamente."

R. Bryant, D. O'Hallaron, em Computer Systems: A Programmer's Perspective, 3a Ed. pg. 616.

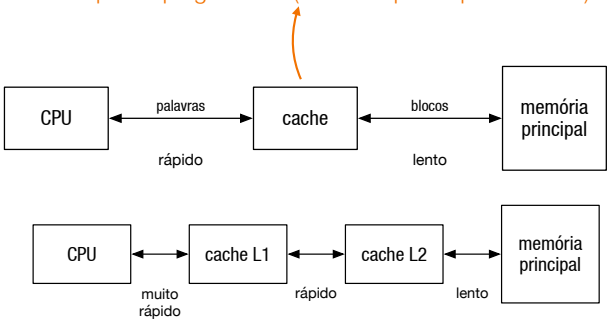
Parte 1

Arquitetura da Memória Cache

A Memória Cache

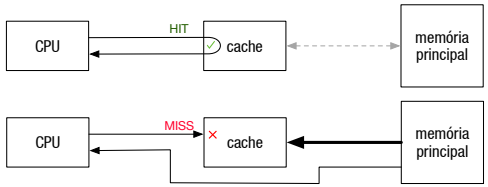
- Contém cópia de **porções** da memória principal.

invisível para o programador (e mesmo para o processador!)



Memória Cache

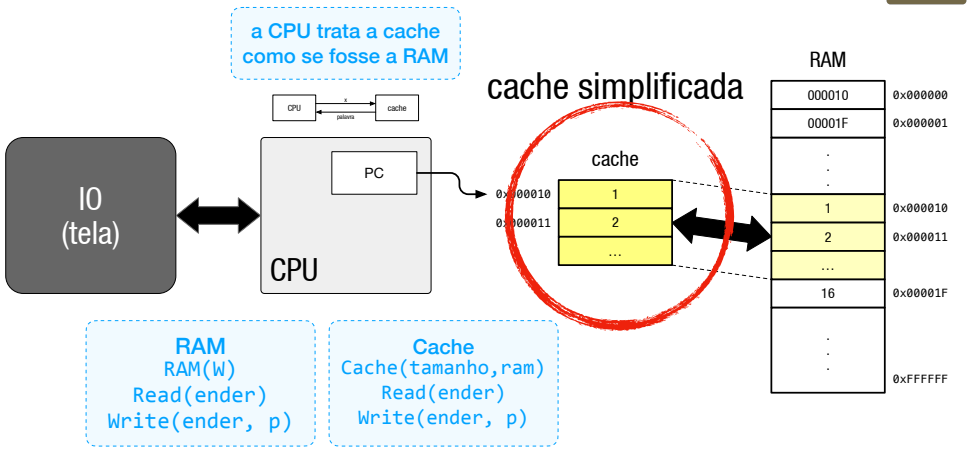
- Quando o conteúdo do endereço solicitado:
  - **está** na cache:
    - **"cache HIT"**
  - **não está** na cache:
    - **"cache MISS"**



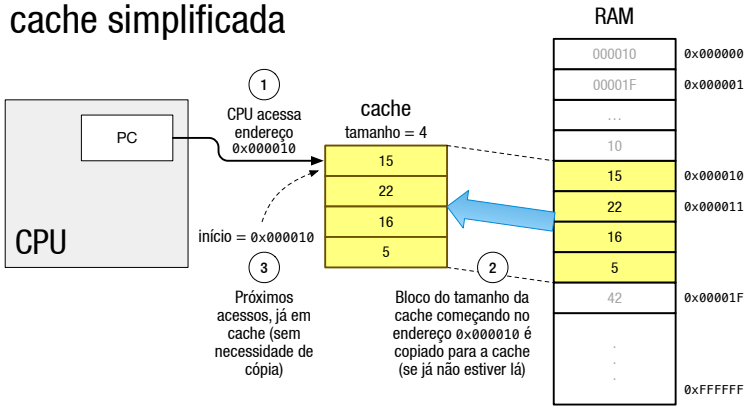
- No caso de "***cache miss***":
  - bloco contendo o endereço precisa ser trazido da memória principal (ou próxima cache)

Exercício Prático

Implementação de Cache "simplificada"



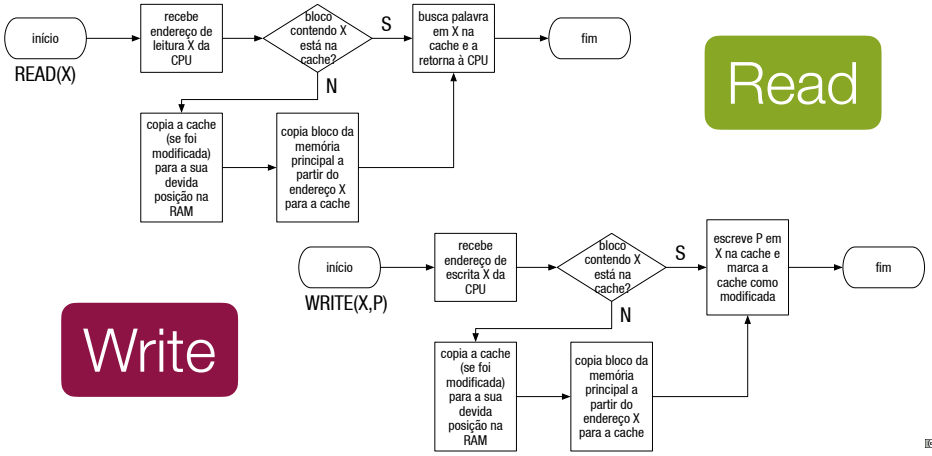
Cache Simplificada



Cache Simplificada

- Implementações de **IO** e **RAM** fornecidas anteriormente *não devem ser alteradas*.
- **CPU** deverá acessar a cache simplificada ao invés da RAM.
- A **cache**:
  - possui um **tamanho** (em número de palavras) indicado no momento da sua instanciação;
  - possui uma **RAM** associada;
  - possui a mesma "***interface***" da RAM (métodos `Read` e `Write`)
    - **(herança: "cache é uma Memória", "RAM é uma Memória");**

Read-Write na Cache Simplificada



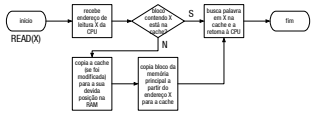
AVA

Cache Simplificada

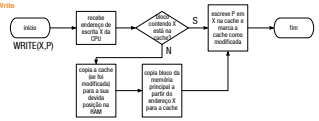
Detalhamento

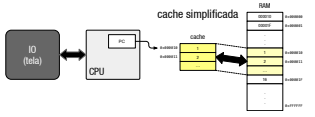
- Faça as modificações na implementação básica da arquitetura von Neumann (disponibilizada anteriormente) de forma a incluir uma **Cache simplificada** entre a CPU e a RAM.
- CPU (e programa principal) deverão ser modificados para **acessar a cache simplificada** ao invés da RAM.
- A *cache*:
  - possui um **tamanho fixo** (em número de palavras) indicado no momento da sua instanciação;
  - possui uma **RAM associada**;
  - possui a mesma "**interface**" da RAM (métodos **Read** e **Write**) (dica: herança "cache é uma memória", "RAM é uma memória");

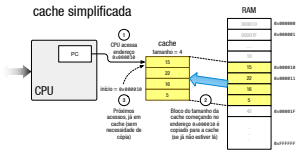
Read



Write







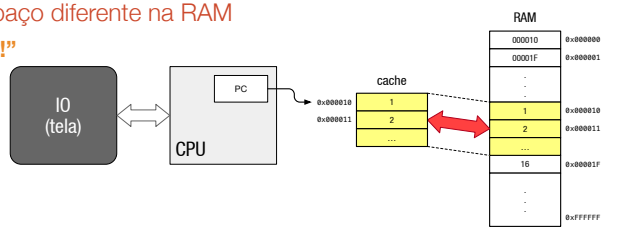
Rode a sua implementação com o seguinte programa principal:

```
io = new IO(System.out);
RAM ram = new RAM(128);
Cache cache = new Cache(8, ram);
CPU cpu = new CPU(cache, io);

try {
    final int inicio = 10;

    // carrega "programa" na RAM
    ram.Write(inicio,118);
    ram.Write(inicio+1, 130);

    cpu(inicio);

} catch (EnderecoInvalido e) {
    System.err.println("Erro: " + e);
}
```


Cache Lines

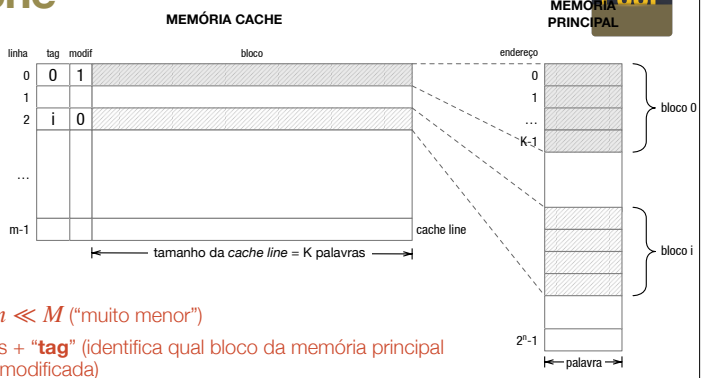
Problemas com "cache simplificada"

- Instruções x dados
  - posições distintas de memória
- "multiprogramação"
  - vários programas rodando "ao mesmo tempo"
  - cada um ocupando um espaço diferente na RAM
    - **muitos "CACHE MISSES!"**
- solução:
  - cache-lines:
    - **várias pequenas porções da memória na cache**



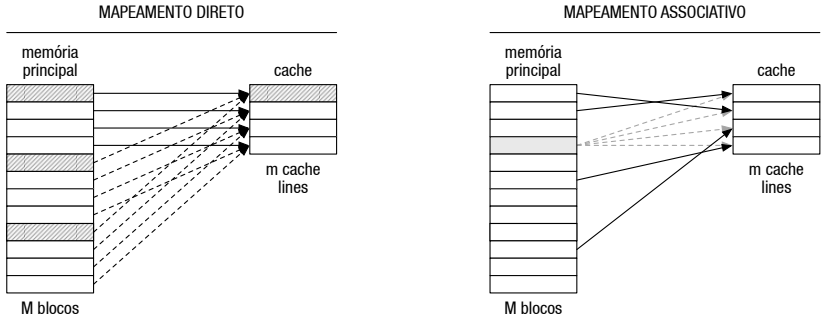
Memória Cache

- Memória principal:
  - 2^n palavras endereçáveis
  - endereços: **n** bits
  - **M** blocos de **K** palavras
    - $M = \frac{2^n}{K}$
- Cache:
  - **m** blocos ("*cache lines*"): $m \ll M$ ("muito menor")
  - cada *cache line*: **K** palavras + "**tag**" (identifica qual bloco da memória principal está na *cache line*) (+ **flag** modificada)
  - tamanho da cache = $m \cdot K$
- A cada instante, **m** blocos da memória principal habitam as **m** cache lines.



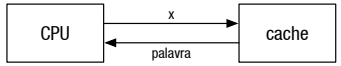
Função de Mapeamento

- Uma vez que $m < M$:
  - necessidade de mapeamento: blocos de memória principal ➔ cache lines



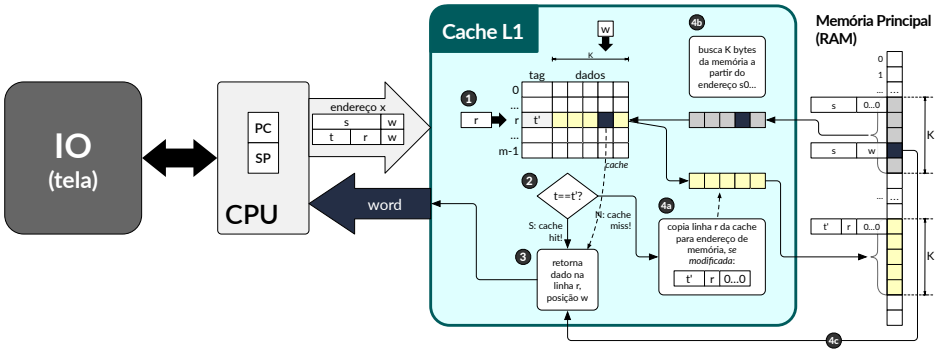
Mapeamento Direto

- CPU solicita acesso ao endereço **x** de memória:
- **x** dividido em bits:
  - **w**: uma das palavras da cache line
    - **se cache line = 64 palavras, w possui 6 bits**
  - **r**: índice da cache line
    - **se cache possui 128 cache lines, r possui 7 bits**
  - **t**: tag formada dos bits restantes de x
    (identifica qual bloco está atualmente na cache line)
    - **se x é de 24 bits, t possui 11 bits**
  - **s**: número do bloco da mem. principal
    (concatenação de **t** e **r**)
    - **18 bits, no exemplo**

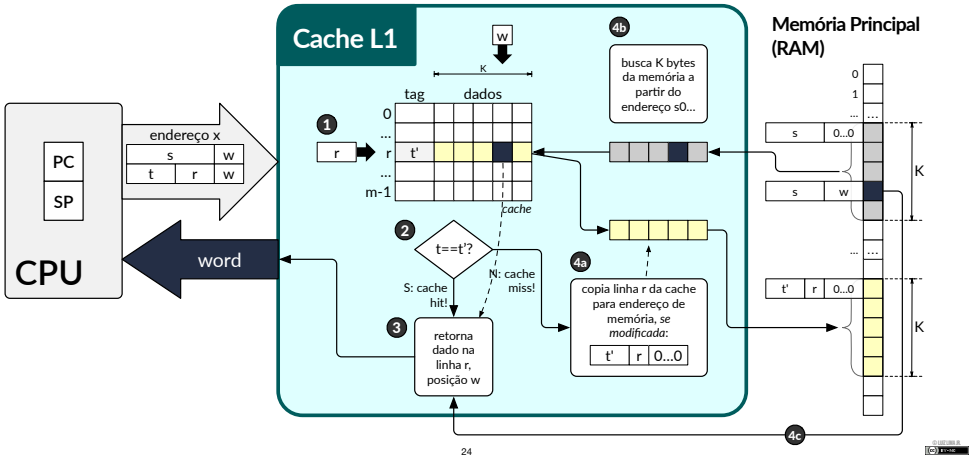


endereço x: 24bits (16M)		
s		w
t	**r**	**w**
tag	# linha	word
0-2047	0-127	0-63
(restante)	m	K
11 bits	7 bits	6 bits

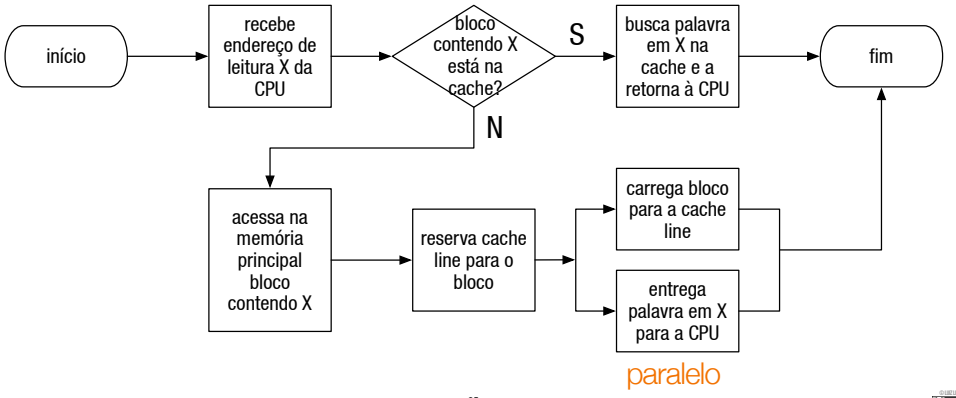
Mapeamento Direto



Mapeamento Direto



Operação de Leitura



Exercício em grupos

Operadores Bitwise em Java

```
int a = 0b00001111;
int b = 0b11110000;
int c = 0b10101010;

System.out.println(Integer.toBinaryString(c & a));  // 00001010
System.out.println(Integer.toBinaryString(c & b));  // 10100000
System.out.println(Integer.toBinaryString(c | a));  // 10101111
System.out.println(Integer.toBinaryString(c | b));  // 11111010
System.out.println(Integer.toBinaryString(c ^ a));  // 10100101
System.out.println(Integer.toBinaryString(c >> 1)); // 01010101
System.out.println(Integer.toBinaryString(a << 1)); // 00011110
```

"e" lógico

&		res
0	0	0
0	1	0
1	0	0
1	1	1

"ou" lógico

|		res
0	0	0
0	1	1
1	0	1
1	1	1

"não" lógico

~	res
0	1
1	0

Precedence	Operator	Operand type	Description
1	++, —	Arithmetic	Increment and decrement
1	+, -	Arithmetic	Unary plus and minus
1	~	Integral	Bitwise complement
1	!	Boolean	Logical complement
1	(*type*)	Any	Cast
2	*, /, %	Arithmetic	Multiplication, division, remainder
3	+, -	Arithmetic	Addition and subtraction
3	+	String	String concatenation
4	<<	Integral	Left shift
4	>>	Integral	Right shift with sign extension
4	>>>	Integral	Right shift with no extension
5	<, <=, >, >=	Arithmetic	Numeric comparison
5	instanceof	Object	Type comparison
6	==, !=	Primitive	Equality and inequality of value
6	==, !=	Object	Equality and inequality of reference
7	&	Integral	Bitwise AND
7	&	Boolean	Boolean AND
8	^	Integral	Bitwise XOR
8	^	Boolean	Boolean XOR
9	|	Integral	Bitwise OR
9	|	Boolean	Boolean OR
10	&&	Boolean	Conditional AND
11	||	Boolean	Conditional OR
12	?:	N/A	Conditional ternary operator
13	=	Any	Assignment

(veja também BitSet em Java)

Exercício

PARÂMETROS

K (palavras na cache line) = 32
Capacidade total da cache = 4K
Capacidade da memória principal = 16M

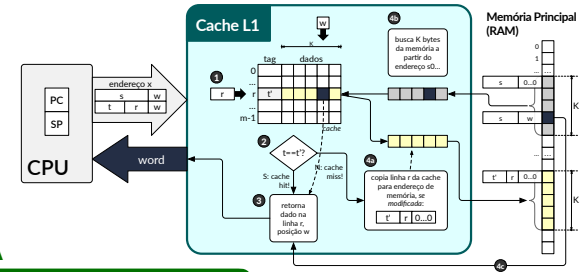
ENTRADA

Endereço x

x = 10.560.325 (em decimal)

1 manual-mente

2 programa Java



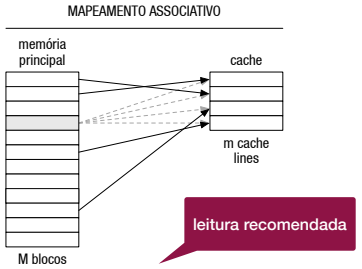
SAÍDA

1. Endereços de início e fim do bloco de x na memória principal (s000…0)
2. Índice da *cache line* de x (r)
3. *Tag* da *cache line* de x (t)
4. Posição de x na *cache line (w)*

x		
s		w
t	r	w

Mapeamento Associativo

- Blocos de RAM podem ser carregados em qualquer cache line.
- “tags”: identificam bloco da RAM
- Para identificar se bloco está na cache:
  - controle lógico examina simultaneamente tags de todas as cache lines.
- Desvantagem:
  - circuitaria complexa (p/ analisar em paralelo)!
- Normalmente adotado:
  - map. direto + associativo = **Mapeamento Associativo por Conjunto**



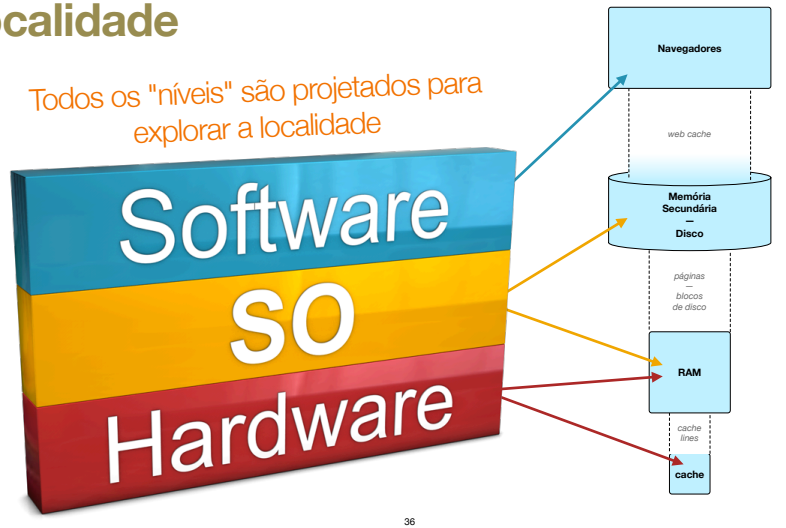
Localidade

Localidade

- Programas bem escritos possuem boa **localidade:**
  - acessam dados próximos na memória a outros dados acessados;
  - "**princípio da localidade**"
    - **grande impacto no projeto e desempenho de sistemas computacionais**
- Localidade:
  - **temporal**:
    - **posição de memória acessada uma vez, será acessada novamente múltiplas vezes em um futuro próximo;**
  - **espacial**:
    - **posição de memória acessada uma vez, então posições próximas serão acessadas em um futuro próximo;**

Localidade

Todos os "níveis" são projetados para explorar a localidade



Localidade de Acessos a Dados de Programas

```java
static int somav(int [] v) {
    int soma = 0;
    for (int i=0; i<v.length; ++i)
        soma += v[i];
    return soma;
}
```

Java

- Boa **localidade**?
  - variável `soma`:
    - **boa localidade temporal**
    - **(localidade espacial não se aplica)**
  - v:
    - **boa localidade espacial**
    - **má localidade temporal**
      - cada elemento é acesso 1 única vez
- Conclusão: **boa localidade**
  - **(pois possui seja localidade espacial ou temporal com relação às variáveis do *loop*)**

Qual possui melhor localidade?

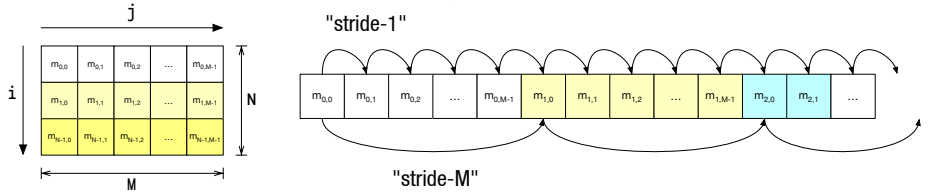
(a)

```java
static int somalin(int [][] m) {
  int soma = 0;
  for (int i=0; i<m.length; ++i)
    for (int j=0; j<m[i].length; ++j)
      soma += m[i][j];
  return soma;
}
```

mesmo resultado numérico

(b)

```java
static int somacol(int [][] m) {
  int soma = 0;
  for (int j=0; j<m[0].length; ++j)
    for (int i=0; i<m.length; ++i)
      soma += m[i][j];
  return soma;
}
```



Localidade: Regras Gerais

1. Programas que **acessam repetidamente as mesmas variáveis** possuem boa localidade.
2. Programas com padrões de acesso de **posições consecutivas** de memória ("*stride-1*") possuem melhor localidade do que aqueles que saltam para posições mais distantes ("*stride-k*", k>1).
3. *Loops* possuem **boa localidade temporal e espacial** na busca de **instruções** na memória.
   - Quanto **menor o corpo** do *loop* e **maior o número de iterações**, melhor a localidade.

Referências

- A. Tanenbaum, “Gerenciamento de Memória” em Sistemas Operacionais Modernos, 4ª edição
- Stallings, W. “Arquitetura e Organização de Computadores”, 10a. Edição, 2018, Pearson.
  - Seções 4.2 e 4.3
- Bryant, R. e O'Hallaron, D, "Computer Systems - A Programmer's Perspective"
  - Localidade de Referências