MC504 - Sistemas Operacionais

Atomicidade e Consistência

Islene Calciolari Garcia

Primeiro Semestre de 2017

Sumário

Atomic (computer science)

From Wikipedia, the free encyclopedia

In concurrent programming, an operation (or set of operations) is **atomic**, **linearizable**, **indivisible** *or* **uninterruptible** *if it appears to the rest of the system to occur instantaneously.*

Necessidade de isolamento

Atomicity is a guarantee of isolation from concurrent processes.

- Decremento/incremento simples não garantem isolamento

```
c++;              c--;
mov rp,c          mov rc,c
inc rp            dec rc
mov c,rp          mov c,rc
```

- Veja os código:
  - `inc.c` e `inc.s`
  - `atomic-inc.c` e `atomic-inc.s`
  - `erro-incremento.c` e `erro-incremento-gdb.c`

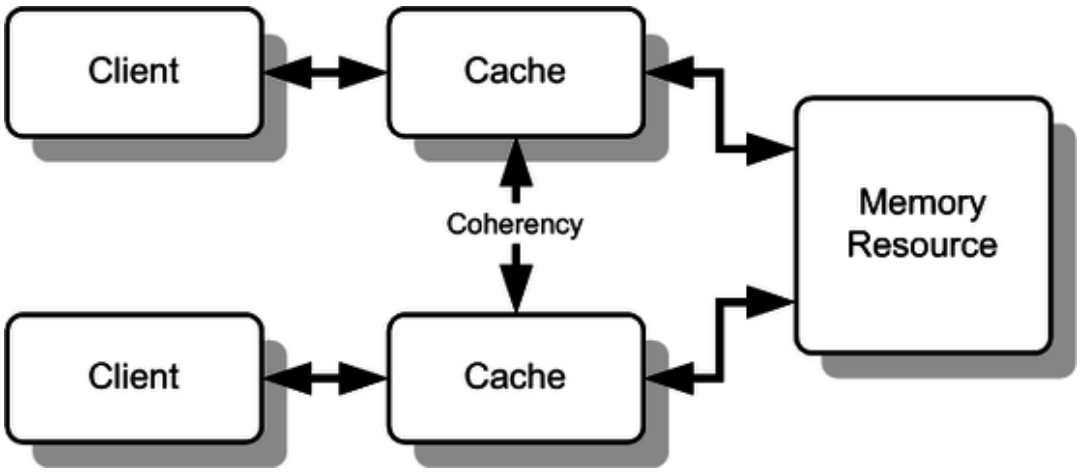
Como obter atomicidade?

Atomicity is often enforced by mutual exclusion, whether at the hardware level building on a cache coherency protocol, or the software level using semaphores or locks.

Atomic (computer science), from Wikipedia, the free encyclopedia.

Cache coherence

From Wikipedia, the free encyclopedia



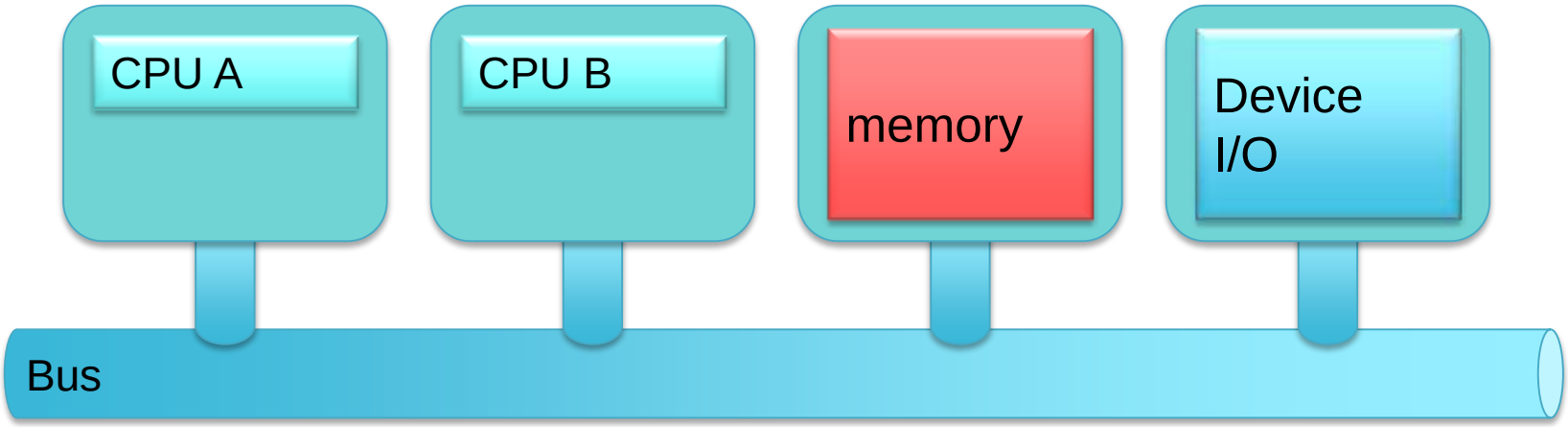
Transaction Serialization : Reads/Writes to a single memory location must be seen by all processors in the same order.

Bus-based multiprocessors

SMP: Symmetric Multi-Processing

All CPUs connected to one bus (backplane)

Memory and peripherals are accessed via shared bus. System looks the same from any processor.

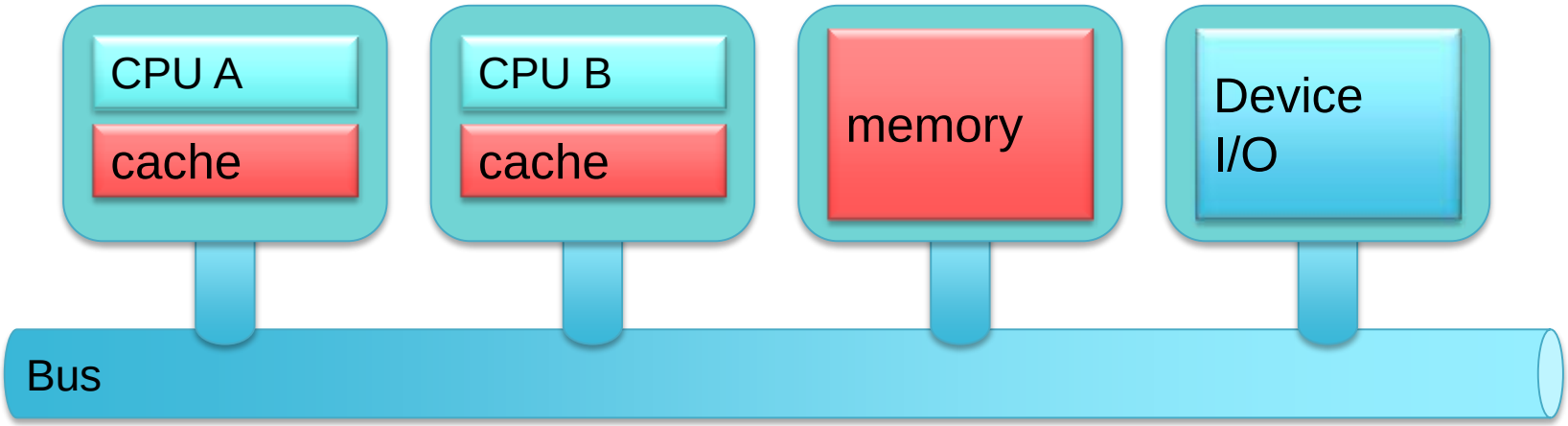


The bus becomes a point of congestion … limits performance

Fonte: Prof. Paul Krzyzanowski

Bus-based multiprocessors

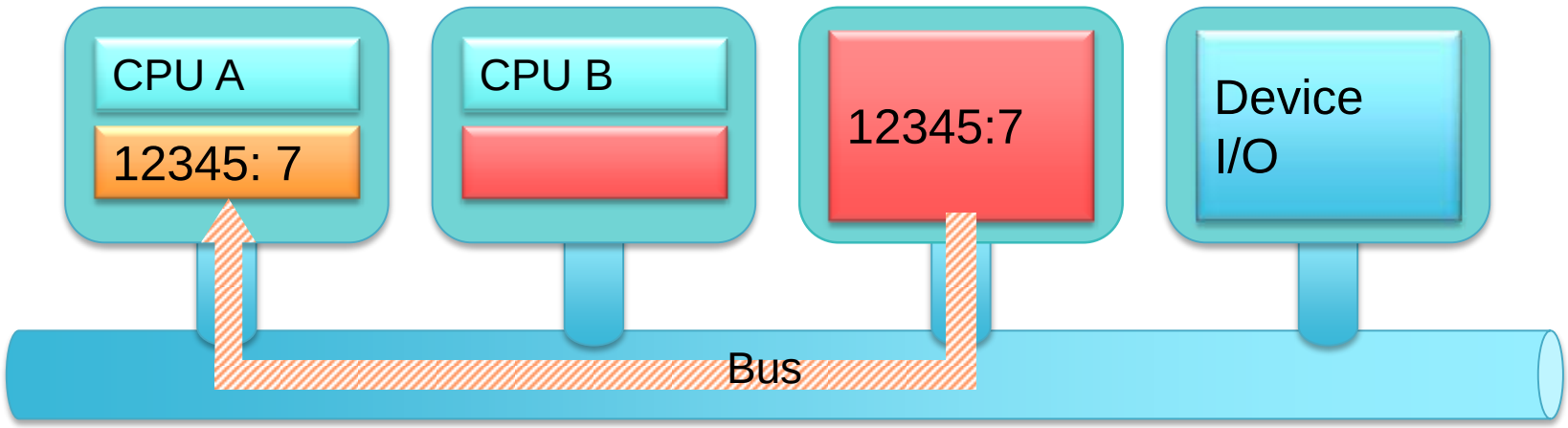
- The cache: great idea to deal with bus overload & memory contention
  - Memory that is local to a processor
- CPU performs I/O to cache memory
  - Access main memory only on cache miss



Fonte: Prof. Paul Krzyzanowski

Working with a cache

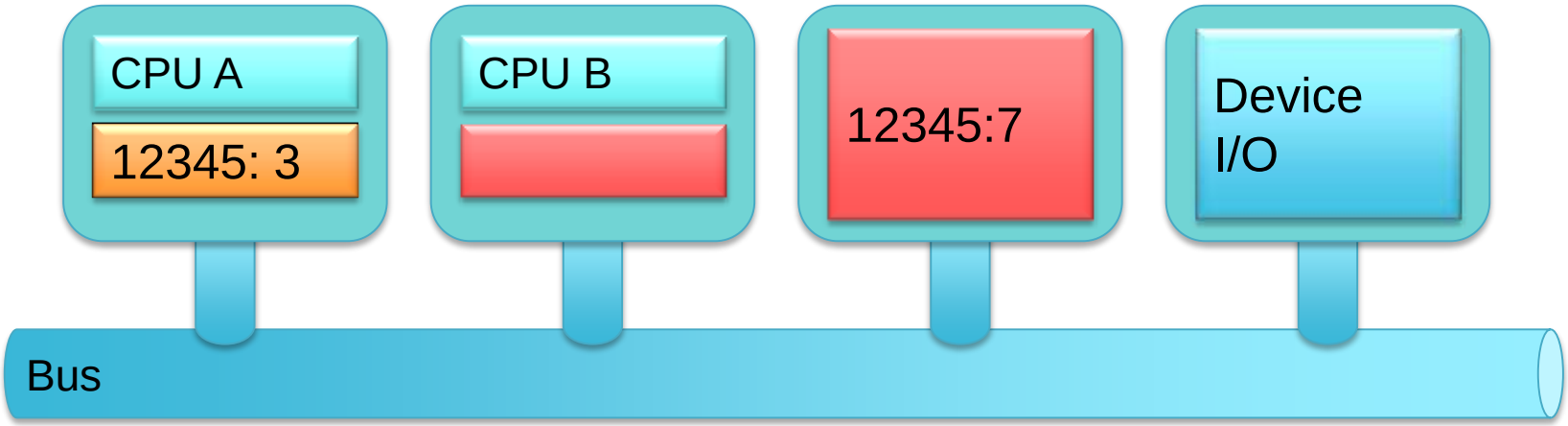
CPU A reads location 12345 from memory



Fonte: Prof. Paul Krzyzanowski

Working with a cache

CPU A modifies location 12345

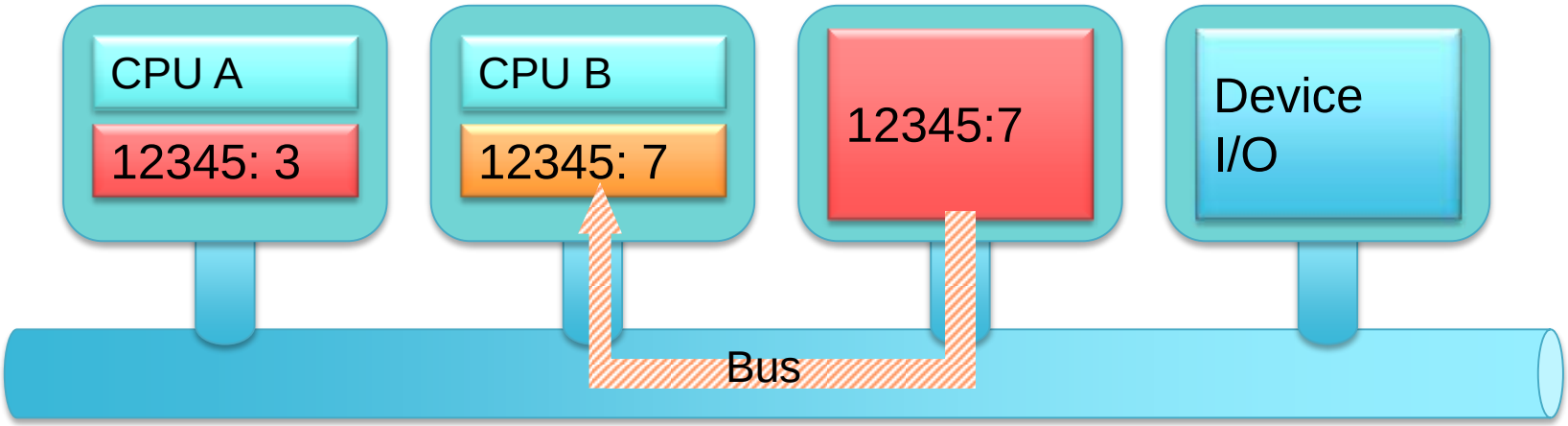


Fonte: Prof. Paul Krzyzanowski

Working with a cache

Gets old value

Memory not coherent!

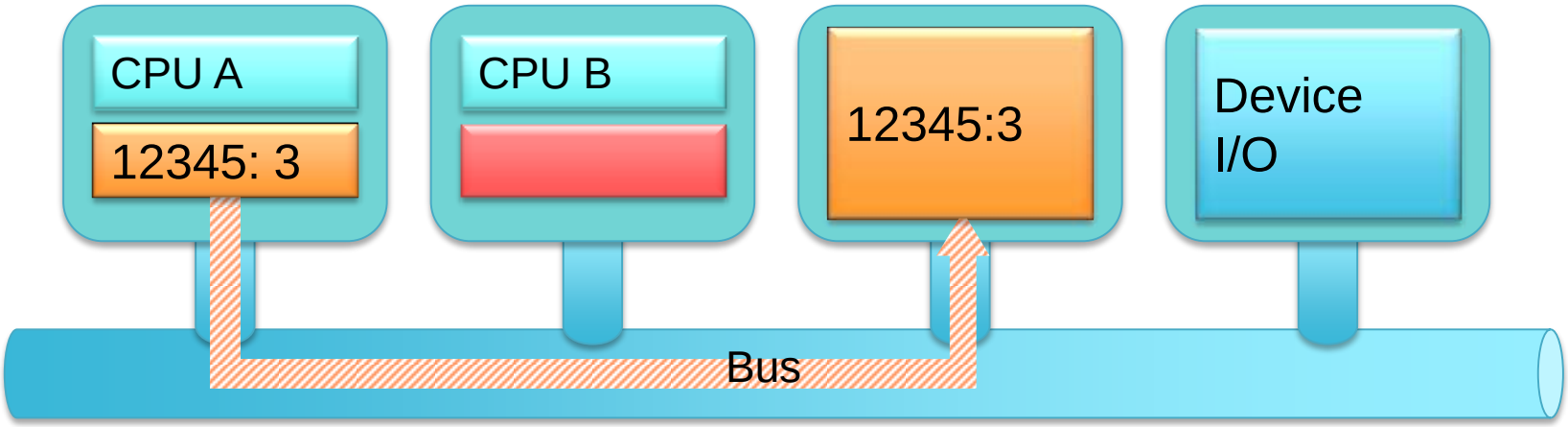


Fonte: Prof. Paul Krzyzanowski

Write-through cache

Fix coherency problem by writing all values through bus to main memory

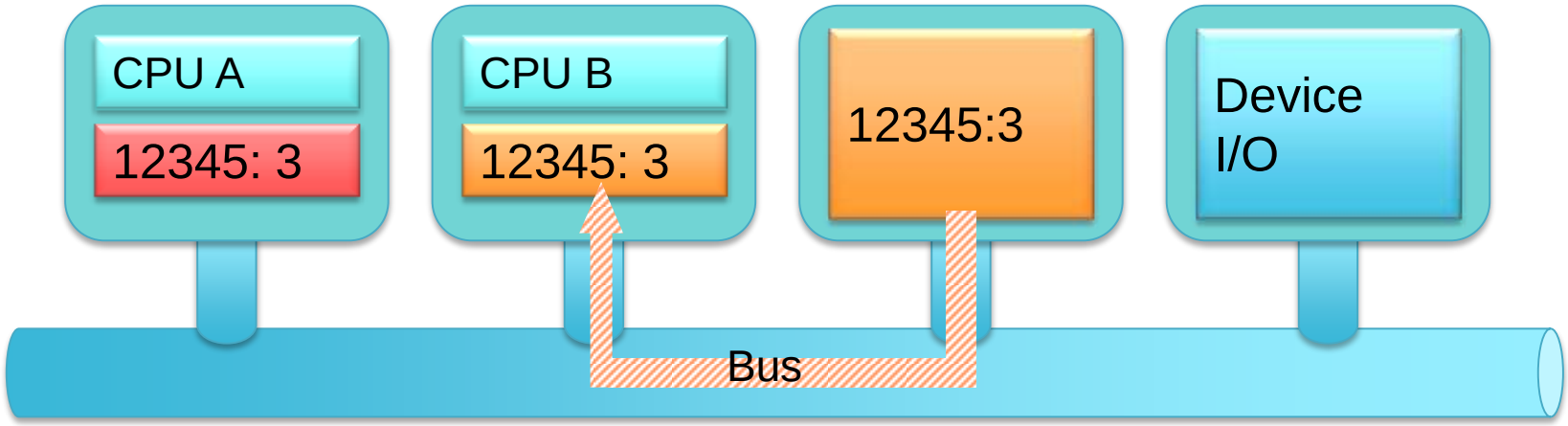
CPU A modifies location 12345 – ***write-through***
main memory is now coherent



Fonte: Prof. Paul Krzyzanowski

Write-through cache … continued

CPU B reads location 12345 from memory
- loads into cache

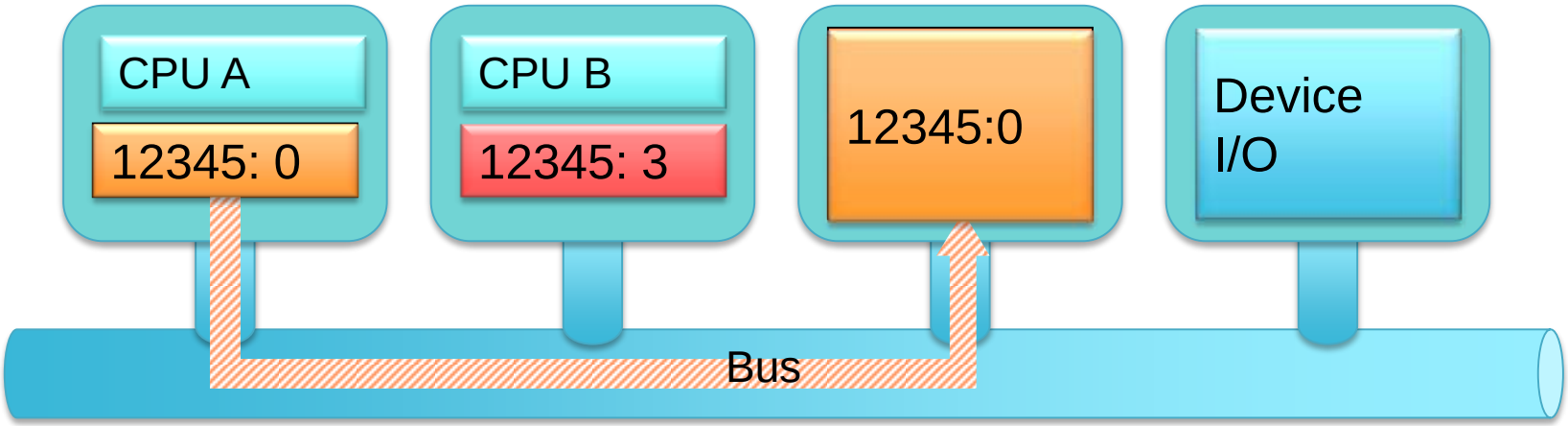


Fonte: Prof. Paul Krzyzanowski

Write-through cache

CPU A modifies location 12345
- write-through

Cache on CPU B not updated
Memory not coherent!



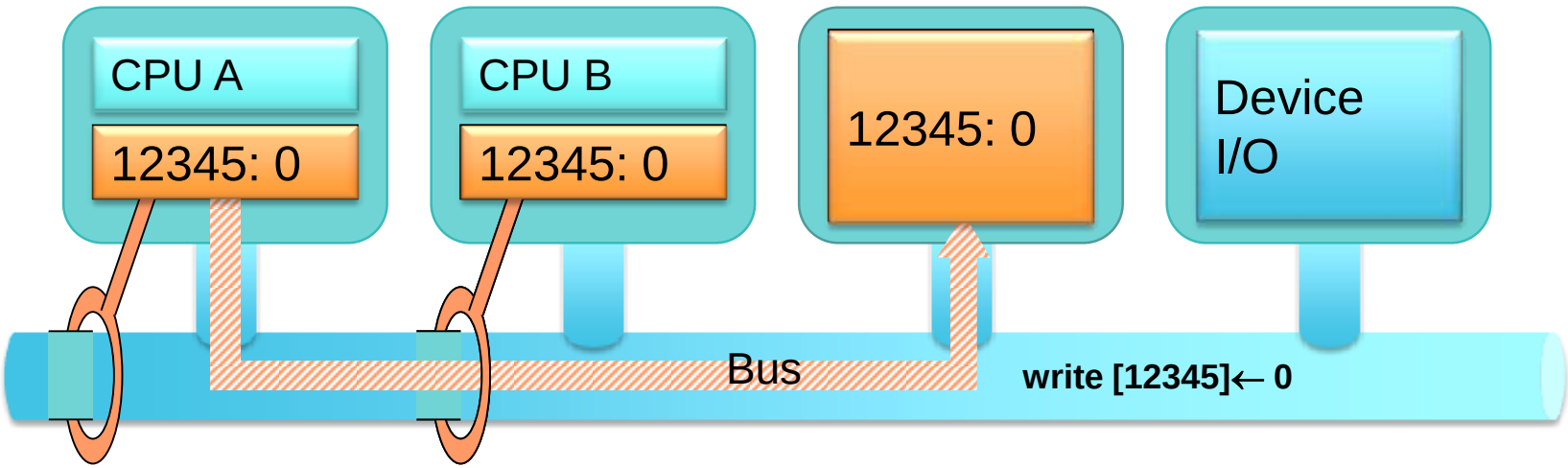
Fonte: Prof. Paul Krzyzanowski

Snoopy cache

Add logic to each cache controller:

monitor the bus for writes to memory

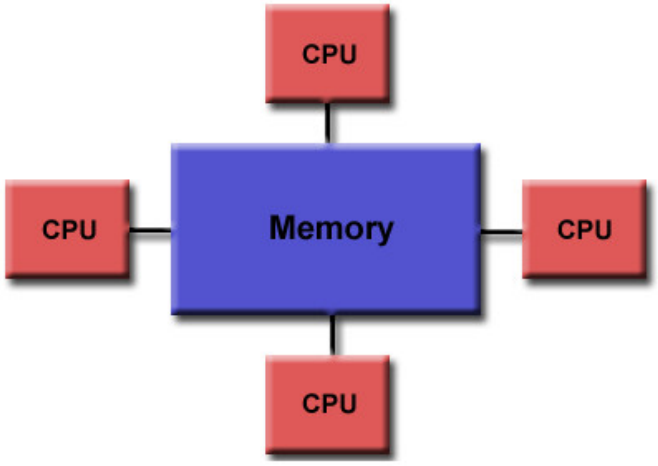
Virtually all bus-based architectures use a snoopy cache



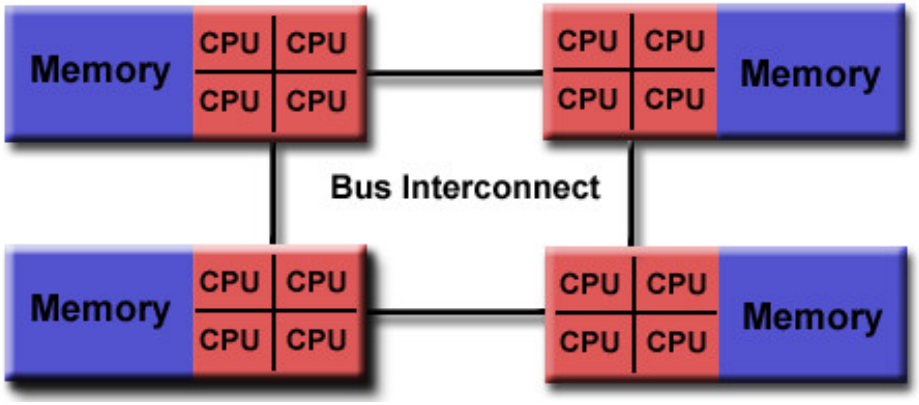
Fonte: Prof. Paul Krzyzanowski

UMA versus NUMA

Uniform Memory Architecture



Non-Uniform Memory Architecture



Fonte:
https://computing.llnl.gov/tutorials/parallel_comp

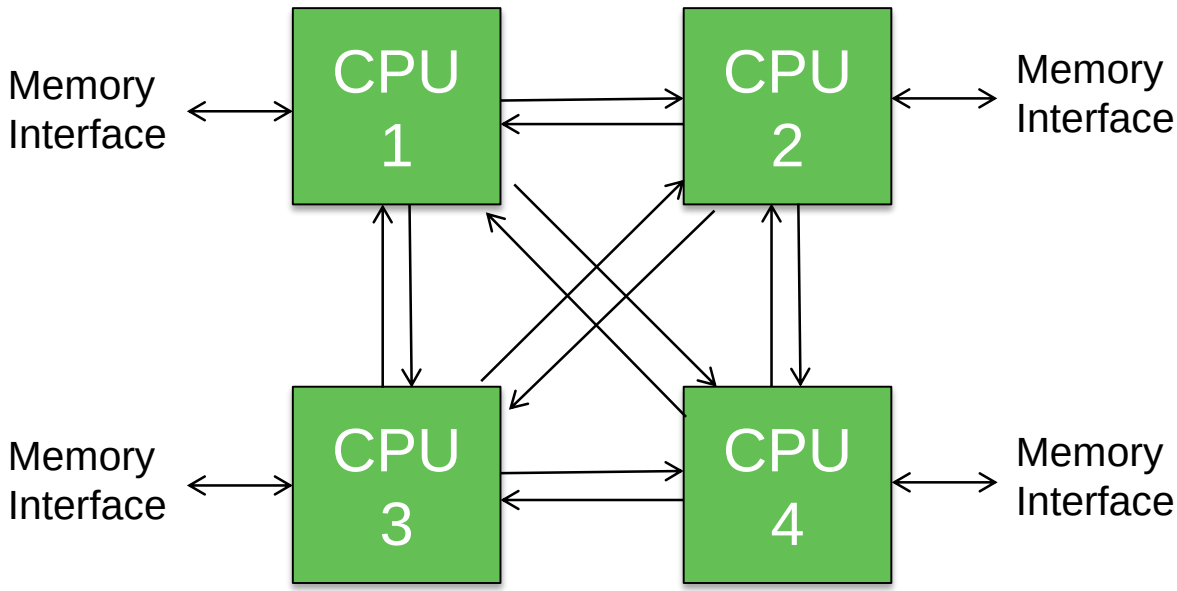
NUMA

- Hierarchical Memory System
- All CPUs see the same address space
- Each CPU has local connectivity to a region of memory
  - fast access
- Access to other regions of memory – slower
- Placement of code and data becomes challenging
  - Operating system has to be aware of memory allocation and CPU scheduling

Fonte: Prof. Paul Krzyzanowski

NUMA Cache Coherence NUMA: Intel Example

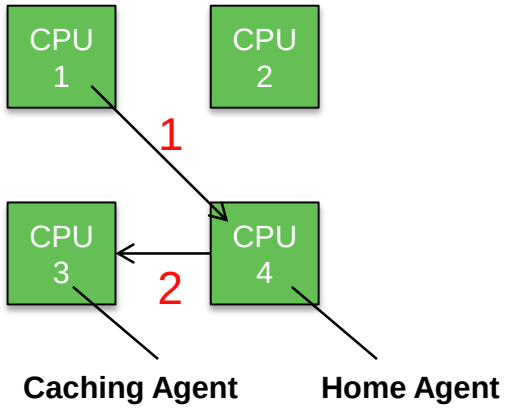
- Home Snoop: *Home-based consistency protocol*
  - Each CPU is responsible for a region of memory
  - It is the "home agent" for that memory
    - Each home agent maintains a directory (table) that track of who has the latest version



Fonte: Prof. Paul Krzyzanowski

NUMA Cache Coherence NUMA: Intel Example

1. CPU sends request to home agent
2. Home agent requests status from the CPU that may have a cached copy (caching agent)

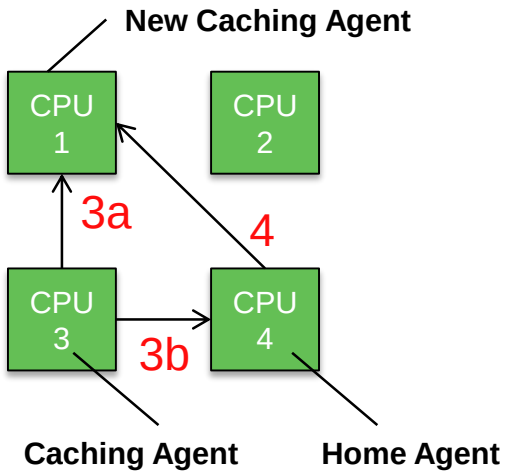


Fonte: Prof. Paul Krzyzanowski

NUMA Cache Coherence NUMA: Intel Example

3. (a) Caching agent sends data update to new caching agent
   (b) Caching agent sends status update to home agent

4. Home agent resolves any conflicts & completes transaction



Fonte: Prof. Paul Krzyzanowski

E tudo pode ser uma grande ilusão...

Thus, an atomic operation does not necessarily actually occur instantaneously. The benefit comes from the appearance: the system behaves as if each operation occurred instantly, separated by pauses. This makes the system consistent.

Atomic (computer science), from Wikipedia, the free encyclopedia.

Instruções atômicas primitivas

- atomic exchange
- compare and exchange
- test and set
- atomic inc

Atomic exchange e spin lock

```
entra_RC:
    MOV RX, #1
    XCHG RX, lock
    CMP RX, #0
    JNE entra_RC
    RET

deixa_RC:
    MOV lock, #0
    RET
```

- Instrução *exchange* troca atomicamente os conteúdos do registrador e da memória;

Compare and exchange

Comportamento básico

```
/* Considere execução atômica das funções */
int val_compare_and_exchange(int *ptr,
                             int oldvalue, int newvalue) {
   int temp = *ptr;
   if(*ptr == oldvalue)
       *ptr = newvalue;
   return temp;
}

int bool_compare_and_exchange(int *ptr,
                              int oldvalue, int newvalue) {
   int temp = *ptr;
   if(*ptr == oldvalue) {
       *ptr = newvalue;
        return 1;
   }
   return 0;
}
```

Mutex lock

Implementação: cmpxchg

`cmpxchg(var, old, new)`

- var $\leftarrow$ new se var == old
- retorna valor de var antes da operação

```
int mutex = 0;  /* mutex livre */

void pthread_mutex_lock(pthread_mutex_t &mutex) {
   while (cmpxchg(&mutex, 0, 1) != 0);
}

void pthread_mutex_unlock(&mutex) {
   mutex = 0;
}
```

Veja o código `spin.c`

Futex (Fast Userspace Mutex)

Prototype

```
long sys_futex (
    void *addr1,
    int op,
    int val1,
    struct timespec *timeout,
    void *addr2,
    int val3);

int syscall(SYS_futex, addr1, FUTEX_XXXX,
            val1, timeout, addr2, val3);
```

FUTEX_WAIT

```
/* Retorna -1 se o futex não bloqueou e
            0 caso contrário */
int futex_wait(void *addr, int val1) {
  return syscall(SYS_futex, addr, FUTEX_WAIT,
                 val1, NULL, NULL, 0);}
```

- Bloqueio até notificação
- Não há bloqueio se `*addr1 != val1`
- Veja o código `ex0.c`

FUTEX_WAKE

```
/* Retorna o número de threads acordadas */
int futex_wake(void *addr, int n) {
    return syscall(SYS_futex, addr, FUTEX_WAKE,
                   n, NULL, NULL, 0);}
```

- Quantas threads acordar?
  - 1
  - x
  - INT_MAX (todas)
- Veja os códigos `ex1.c` e `ex2.c`

Mutex lock

Implementação: cmpxchg e futex

```
int mutex = 0;  /* mutex livre */

void pthread_mutex_lock(pthread_mutex_t &mutex) {
   while (cmpxchg(&mutex, 0, 1) != 0)
    futex_wait(&mutex, 1);
}

void pthread_mutex_unlock(&mutex) {
   mutex = 0;
   futex_wake(&mutex, 1);  /* Como evitar esta   */
                           /* chamada se ninguém */
                           /* estiver esperando? */
}
```

Veja o código `spin-futex.c`

Justiça na obtenção do lock

- Na implementação anterior, threads podem furar a fila e outras podem sofrer starvation...
- Seria possível fazer uma implementação simples que eliminasse esse problema?
- Veja questões de provas anteriores
  - 1s2010: Prova1 questão 2
  - 1s2014: Prova1 questão 4
  - 2s2013: Prova1 questão 3