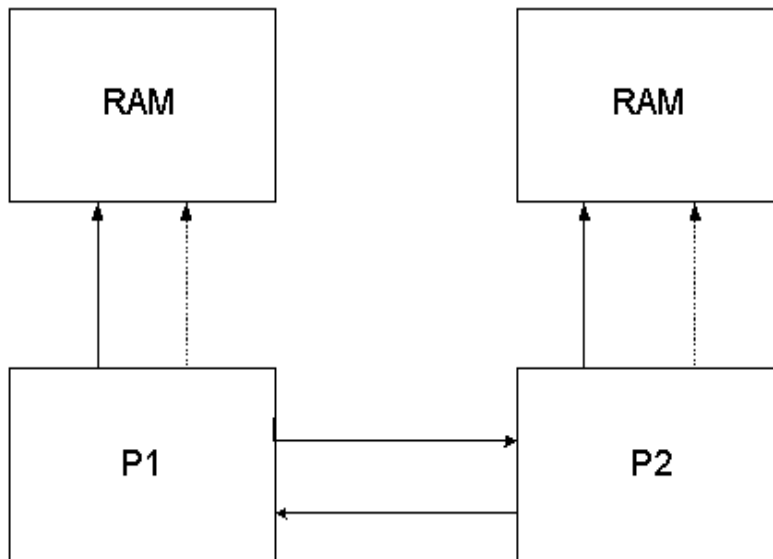


Practical Parallel Programming

»» Shared Memory systems

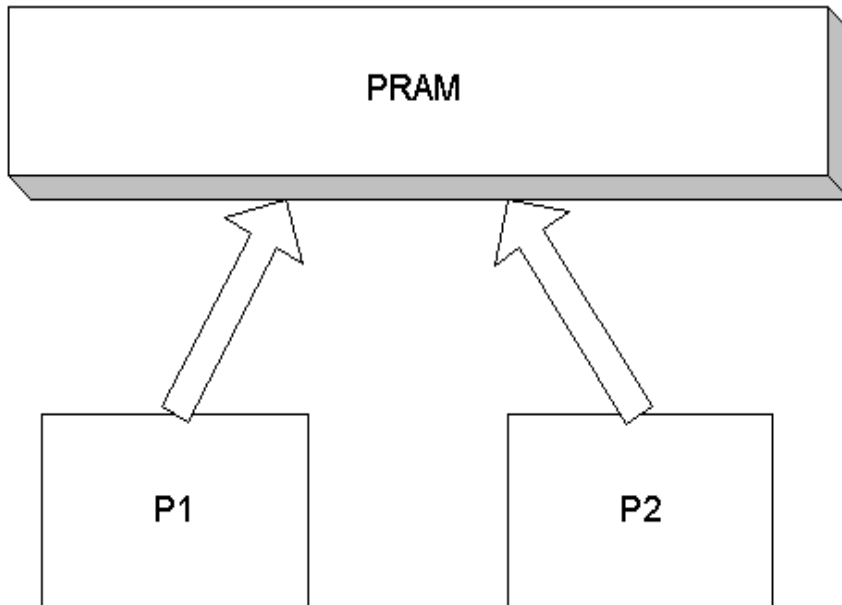
Jan Lemeire
2019–2020

I. Distributed-Memory Architectures



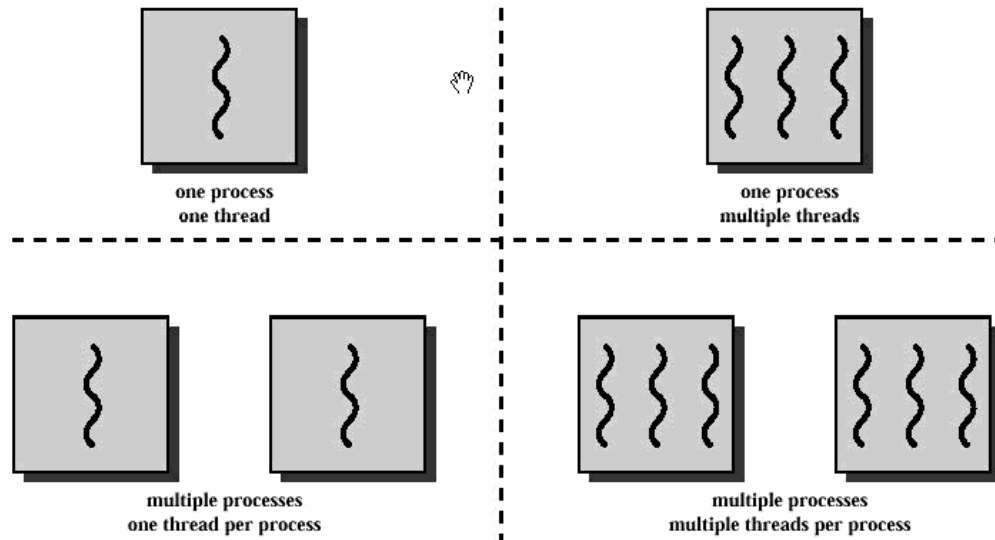
- ▶ Each process got his own local memory
- ▶ Communication through messages
- ▶ Process is in control

II. Shared Address-space Architectures

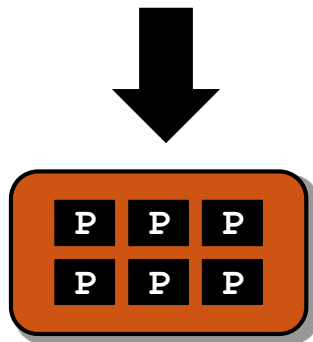


- ▶ Example: multiprocessors
- ▶ **PRAM**: Paralleled Random Access Memory
 - Idealization: No communication costs
- ▶ But, unavoidability: the possibility of *race conditions*

Processes versus Threads



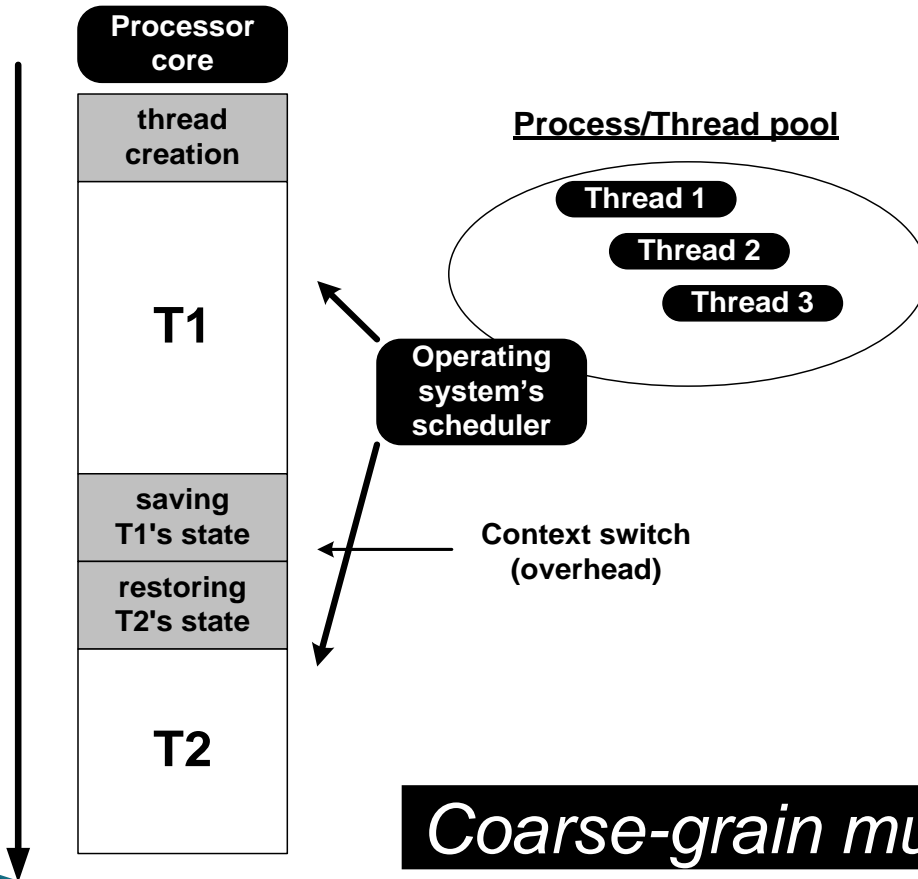
Scheduled by the OS on the available processors



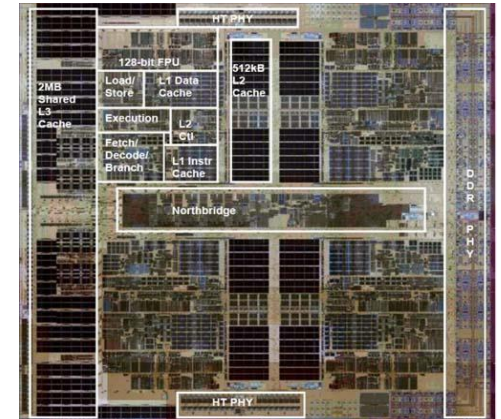
Example: A file server on a LAN

- It needs to handle several file requests over a short period
- Hence, it is more efficient to create (and destroy) a single thread for each request
- Multiple threads can possibly be executed simultaneously on different processors (mapped by Operating System)

Running threads on same core



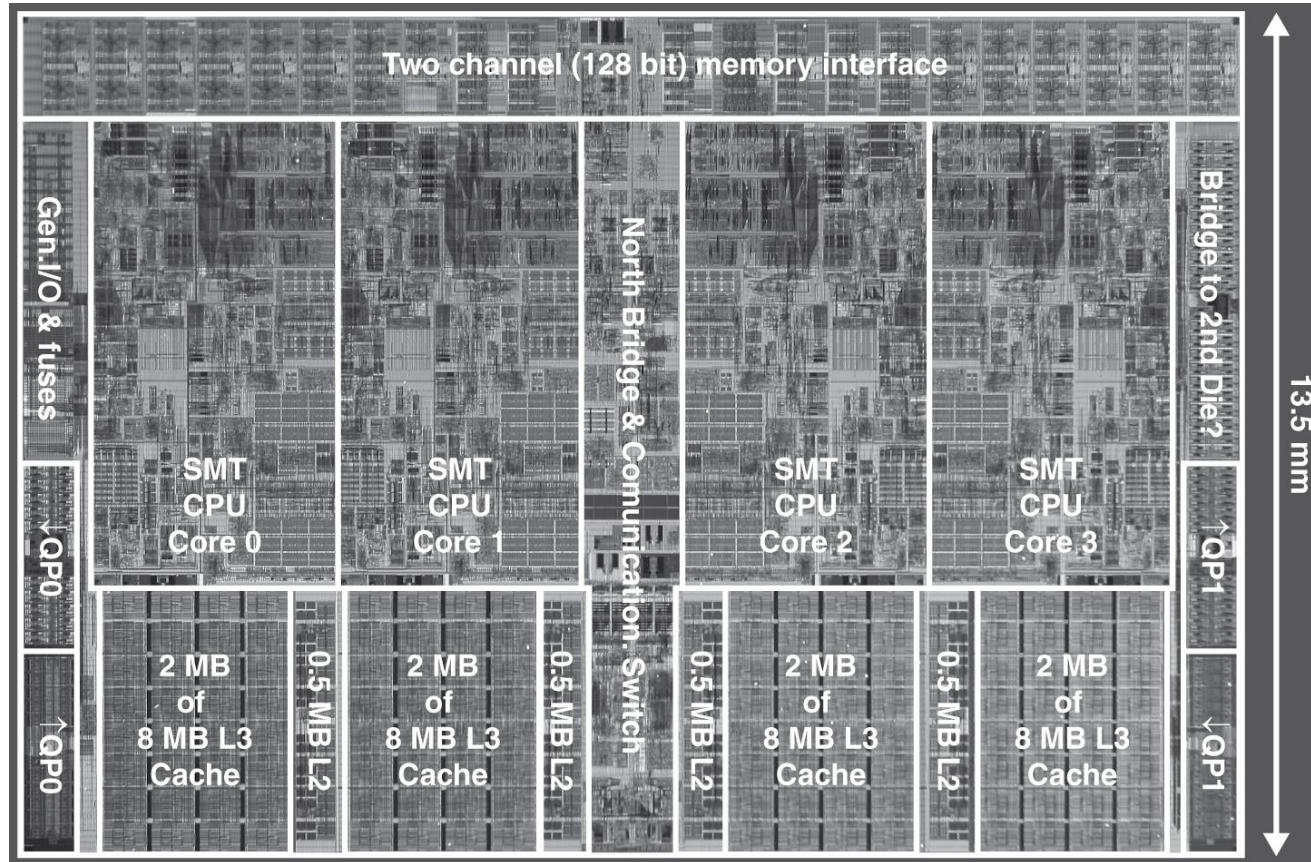
- ▶ Executed one by one
- ▶ Context switch
 - Thread's state in core: instruction fetch buffer, return address stack, register file, control logic/state, ...
 - Supported by hardware
- ▶ Takes time!



1. Architecture

Multilevel On-Chip Caches

Intel Nehalem 4-core processor



Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache

2-Level TLB Organization

	Intel Nehalem	AMD Opteron X4
Virtual addr	48 bits	48 bits
Physical addr	44 bits	48 bits
Page size	4KB, 2/4MB	4KB, 2/4MB
L1 TLB (per core)	L1 I-TLB: 128 entries for small pages, 7 per thread (2x) for large pages L1 D-TLB: 64 entries for small pages, 32 for large pages Both 4-way, LRU replacement	L1 I-TLB: 48 entries L1 D-TLB: 48 entries Both fully associative, LRU replacement
L2 TLB (per core)	Single L2 TLB: 512 entries 4-way, LRU replacement	L2 I-TLB: 512 entries L2 D-TLB: 512 entries Both 4-way, round-robin LRU
TLB misses	Handled in hardware	Handled in hardware

3-Level Cache Organization

	Intel Nehalem	AMD Opteron X4
L1 caches (per core)	L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 9 cycles
L2 unified cache (per core)	256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a
L3 unified cache (shared)	8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a	2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles

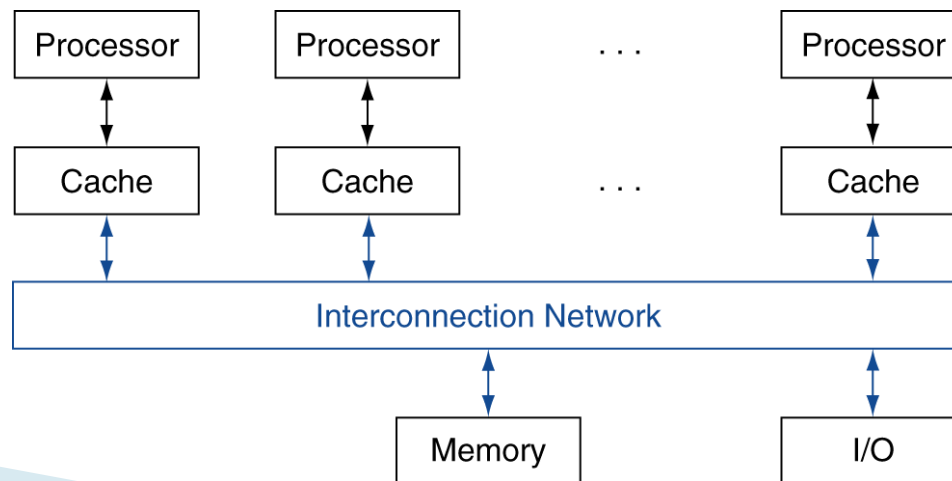
n/a: data not available

Multicores: The following should be provided by hardware and/or OS

- A. **Connect** processors to shared memories (the interconnect)
- B. Address **concurrent** read/writes
- C. **Memory consistency**: cache coherence protocol
- D. Static/dynamic **mapping** of processes/threads to the cores
- E. **Thread synchronization**

A. Shared Memory

- ▶ SMP: shared memory multiprocessor
 - Hardware provides single physical address space for all processors
 - Synchronize shared variables using locks
 - Memory access time
 - UMA (uniform) vs. NUMA (nonuniform)



Typical architectures

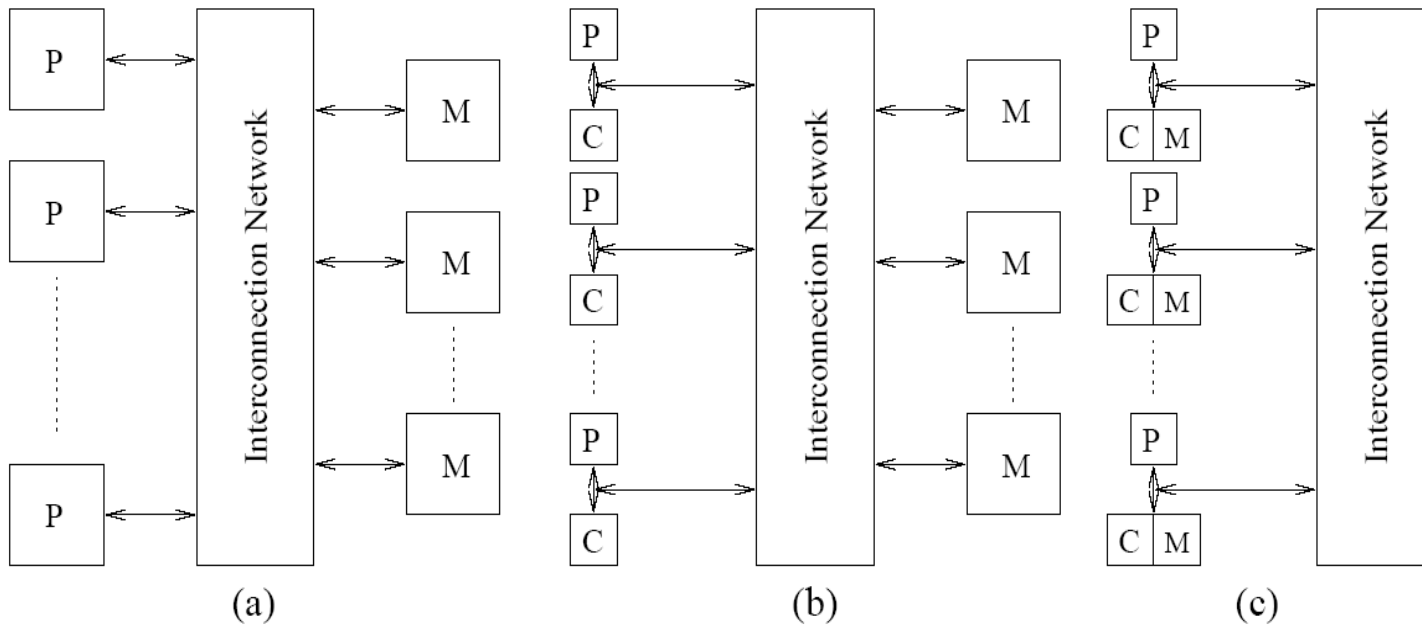
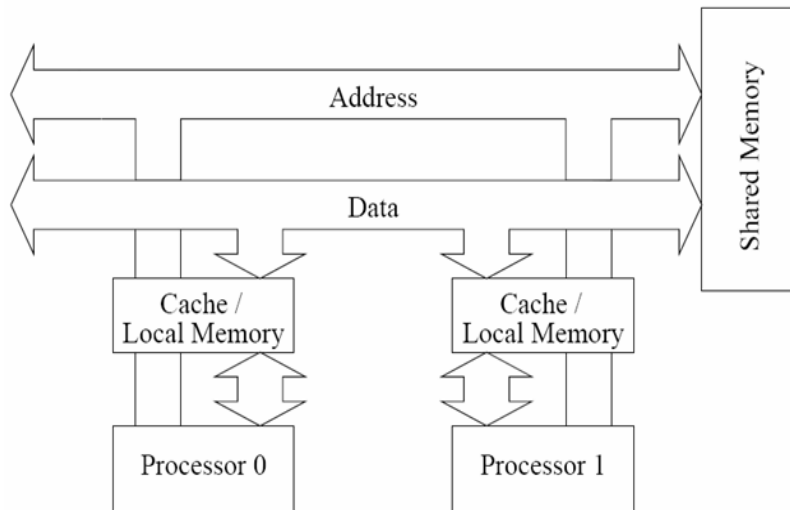
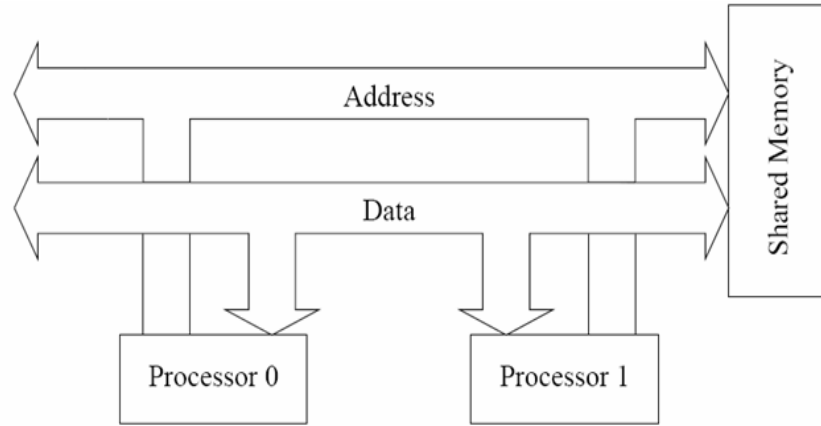


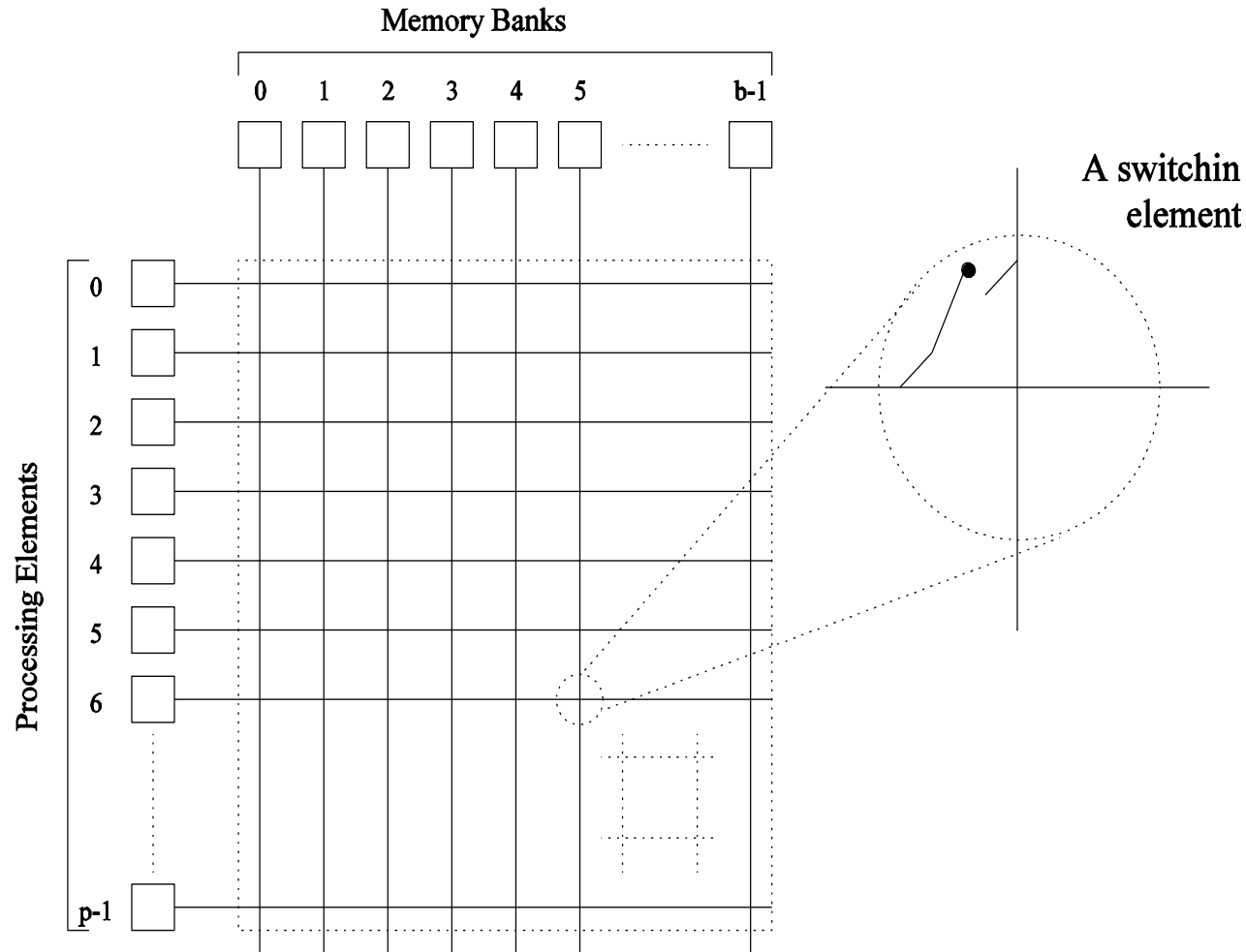
Figure 2.5 Typical shared-address-space architectures: (a) Uniform-memory-access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.

Bus-based Interconnects

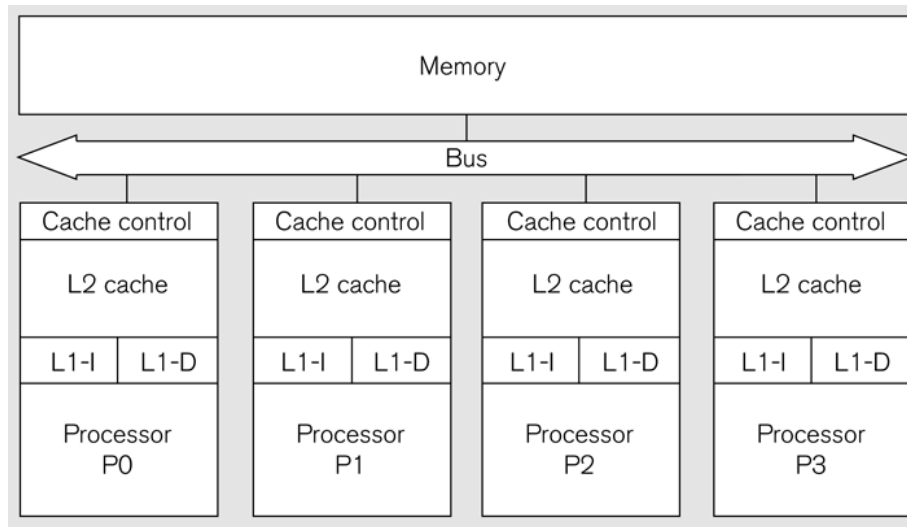


With local memory/cache

Crossbar switches



Symmetric Multiprocessor Architectures (SMPs)



- ▶ Cf AMD architecture
- ▶ Bus is potential bottleneck
- ➔ Number of SMPs is limited

B. PRAM Architectures

- ▶ Handling of simultaneous memory accesses:
 - Read operation
 - Exclusive-read, concurrent-read
 - Write operation
 - Exclusive-write, concurrent-write
- ▶ 4 implementations:
 - EREW: access to a memory location is exclusive
 - CREW: multiple write accesses are serialized
 - ERCW
 - CRCW: most powerful PRAM model

Concurrent Write Access Requires Arbitration

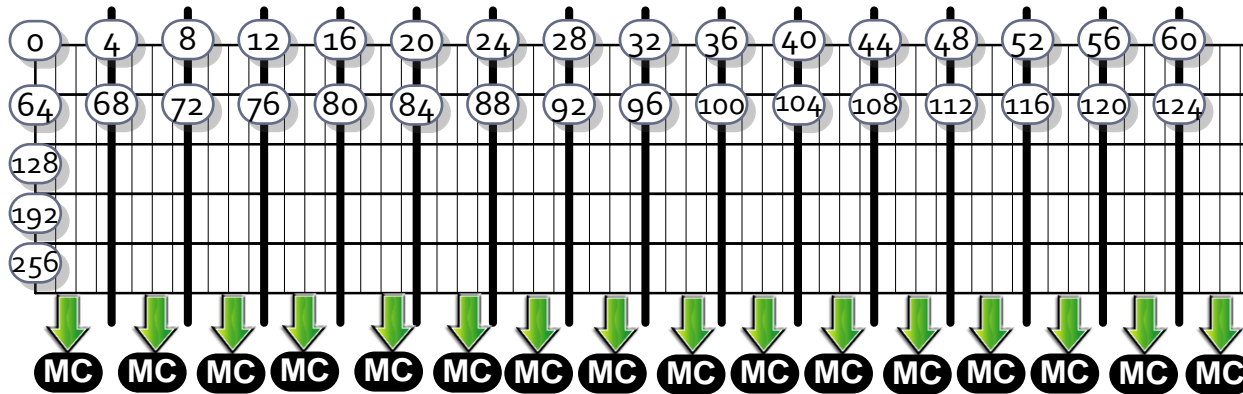
- ▶ **Common:** write is allowed if the new values are identical
- ▶ **Arbitrary:** an arbitrary processor is allowed to write, the rest fails.
- ▶ **Priority:** processor with the highest priority succeeds
- ▶ **Sum:** the sum of the values is written. Any other operator can be used.

Memory banks / partitions

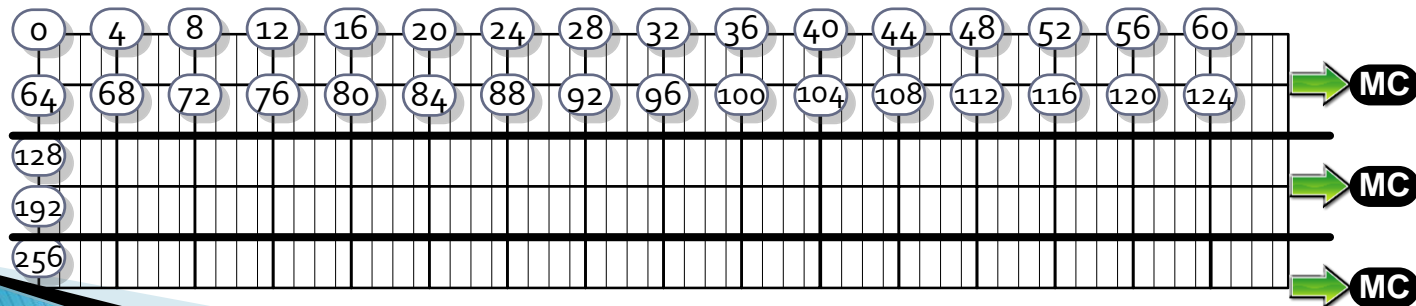
- ▶ Each memory bank / partition has one or two ports (read/write)
 - Vertical partitioning (banks)
 - Horizontal partitioning
- ▶ Each port can serve one memory request at the same time
 - ▶ Multiple requests are handled one-by-one

Memory Controllers (MCs)

Memory banks

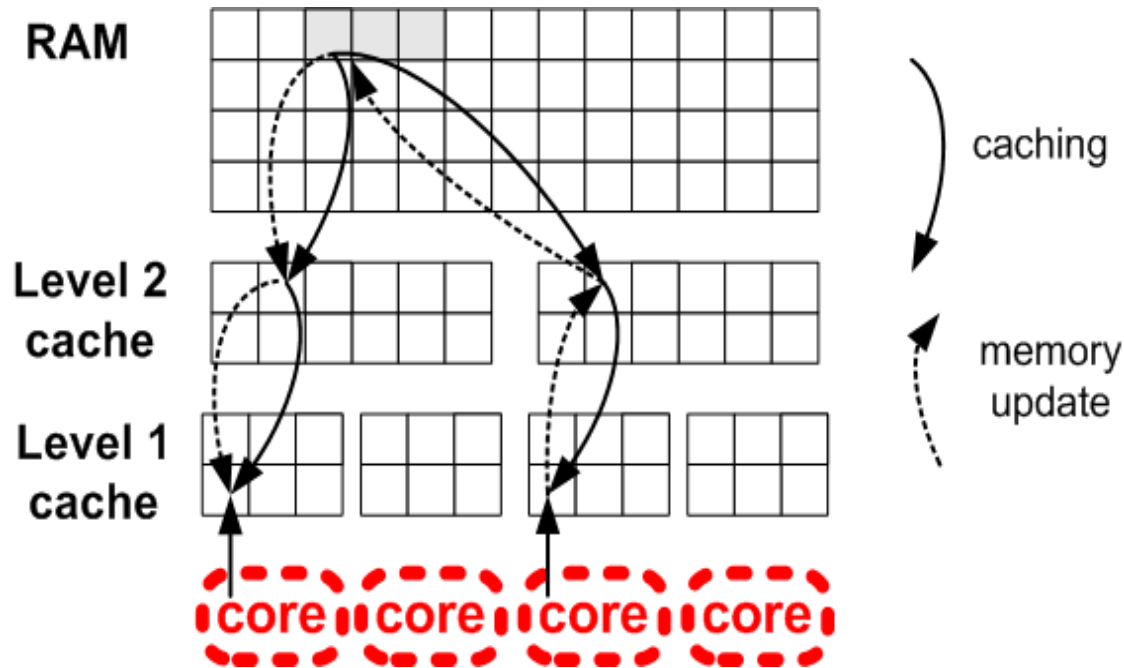


Memory partitions



...

C. Caching & memory coherence



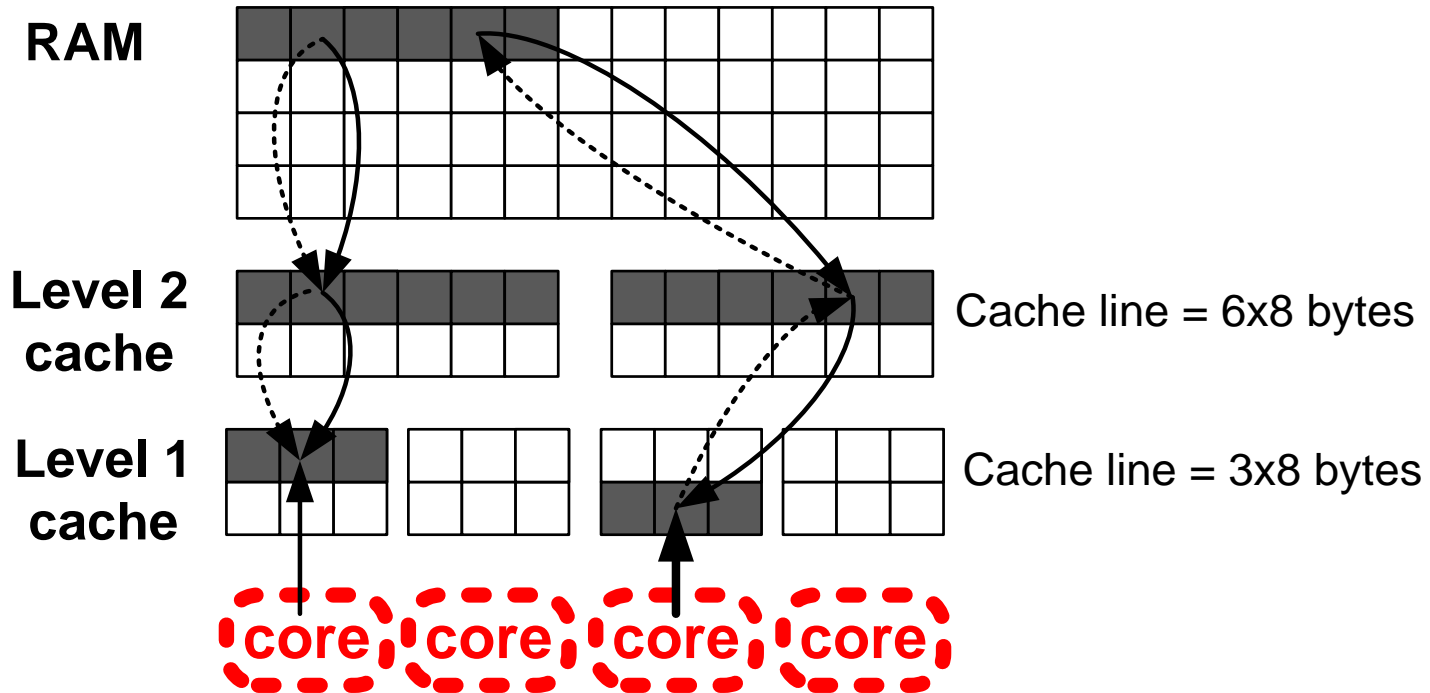
- ▶ *Caching*: copies are brought closer to processor
 - By cache lines of 64/128 Bytes
- ▶ *Cache coherence mechanism*: to update copies

Cache Coherence Problem

- ▶ Suppose two CPU cores share a physical address space
 - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

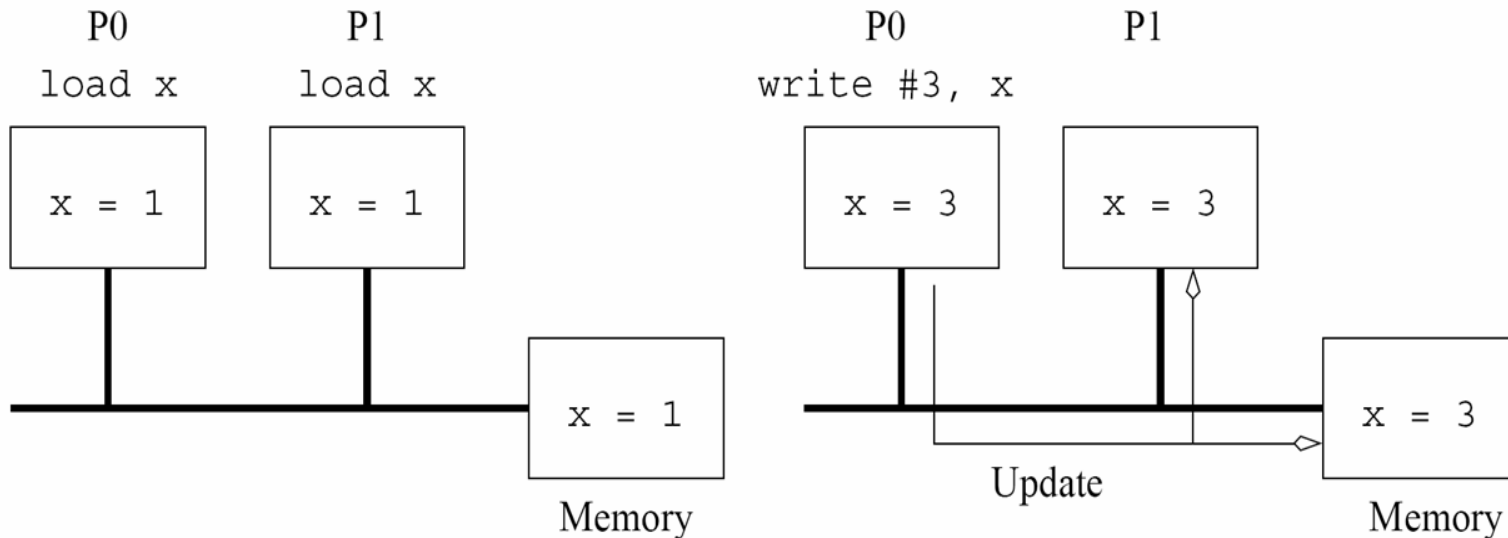
False sharing



- ▶ 2 processors do not share data but share a cache line
 - each processor has some data in the same cache line
 - cache line is kept coherent, *unnecessarily...*

Cache Coherence Mechanisms

Update protocol

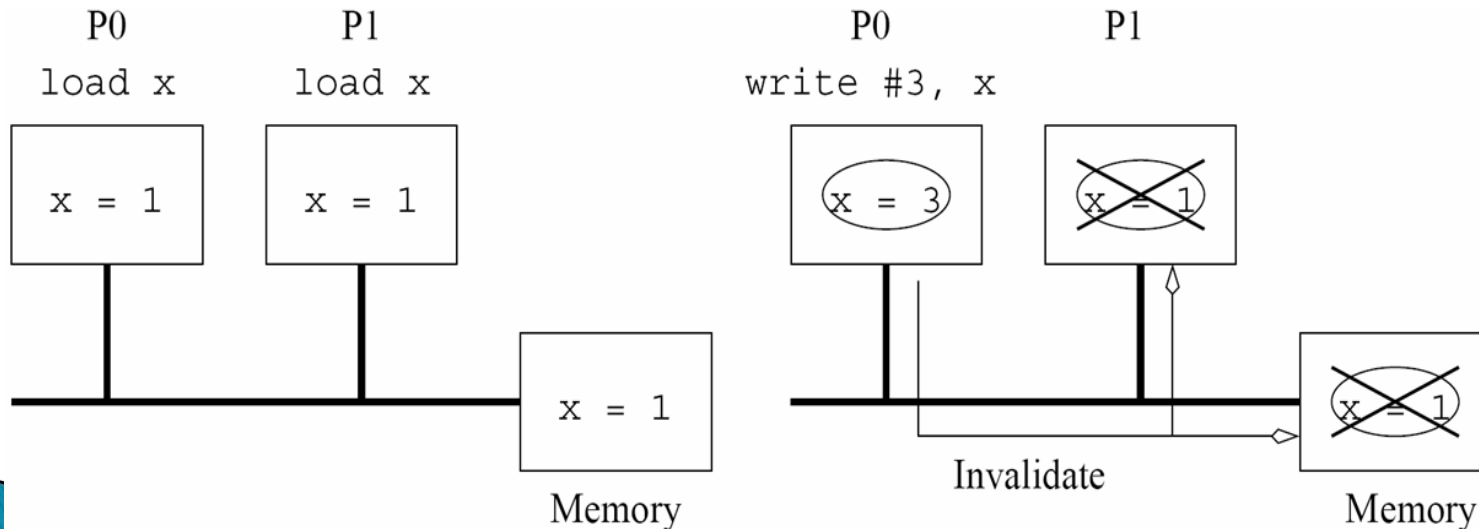


- ▶ Excess in updates if variable is only read once in P1
- ▶ *False sharing*: processes update different parts of same cache line
- ▶ Used nowadays: Invalidate protocols

Cache Coherence Mechanisms

- ▶ To keep copies of data in different memory elements consistent!
 - Is not always performed. Best effort.
 - Or explicit synchronization.

Invalidate protocol



Cache Coherence Protocols

= Operations performed by caches in multiprocessors to ensure coherence

▶ Snooping protocols

- If a cache line has been changed: a notification is put on the snoop bus
- All caches monitor the snoop bus.
 - If a cache line they own is changed by another cache
⇒ cache line is invalidated or update
 - The first cache that can put notification on the snoop bus gets the ownership of the cache line

▶ Directory-based protocols

- Caches and memory record sharing status of blocks in a directory

Invalidating Snooping Protocols

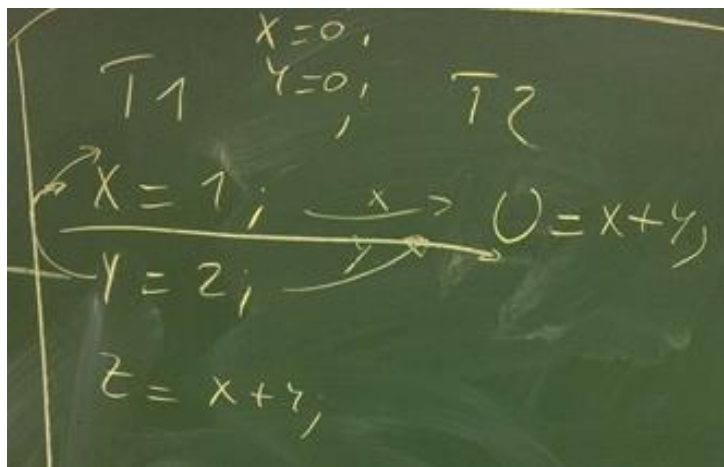
- ▶ Cache gets exclusive access to a block when it is to be written
 - Broadcasts an invalidate message on the bus
 - Subsequent read in another cache misses
 - Owning cache supplies updated value

CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		0
CPU B read X	Cache miss for X	1	1	1

Memory Consistency

- ▶ When are writes seen by other processors
 - “Seen” means a read returns the written value
 - Can’t be instantaneously
- ▶ Assumptions
 - A write completes only when all processors have seen it
 - A processor does not reorder writes with other accesses
- ▶ Consequence
 - P writes X then writes Y
 - ⇒ all processors that see new Y also see new X
 - Processors can reorder reads, but not writes

Memory Consistency



MESI-protocol

Possible states of a cache line:

State	Cacheline Valid?	Valid in memory?	Copy in other cache?	Write access
Modified	Yes	No	No	Cache
Exclusive	Yes	Yes	No	Cache
Shared	Yes	Yes	Possible	Cache/Memory
Invalid	No	Unknown	Possible	Memory

- ◆ Complex, but effective protocol
- ◆ Used by Intel
- ◆ AMD adds an 'owned' state => MOESI-protocol

D. Process/thread scheduler

- ▶ Static mapping
- ▶ Dynamic: Processor switches between processes/threads



Software versus hardware threads

▶ Software threads

- Processor can only execute one program at the same time
- Overhead! Due to *context switch* (saving/restoring of processor state)

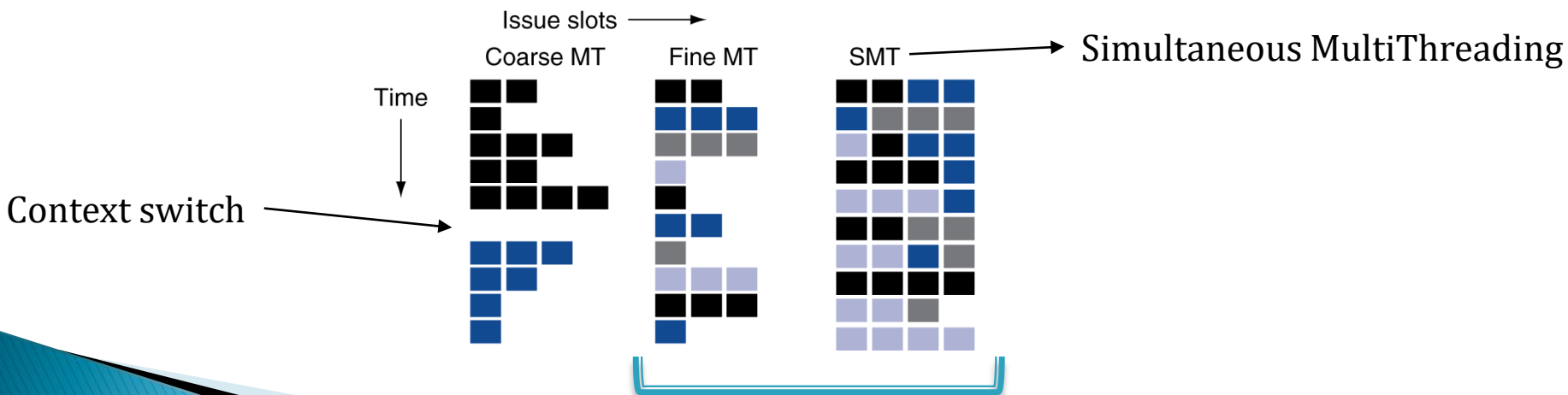
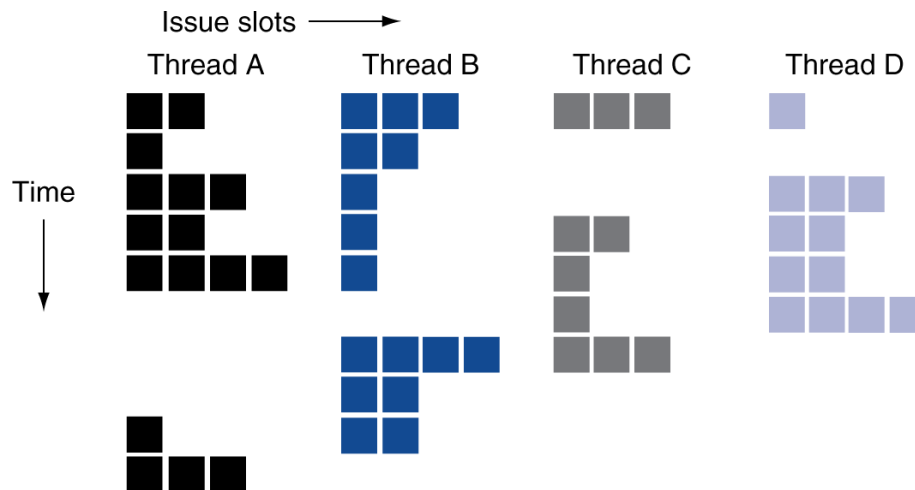
▶ Hardware threads

- Processor can execute several programs simultaneously: instructions of different threads go through pipeline
- No overhead!
- Intel CPUs: *Hyperthreading*

Hardware threads

- ▶ Software threads: scheduling and context switching is performed by Operating System
 - Has a cost (overhead).
- ▶ Hardware thread:
 - Scheduling and context switching done by hardware.
 - Separate registers & logic for each thread.
 - Context switching is cheap.
 - *Each hardware thread appears as a logical processor core to the OS!*
- ▶ In INTEL processors: Hyperthreading
- ▶ In GPUs: 1000s of threads are possible without overhead!

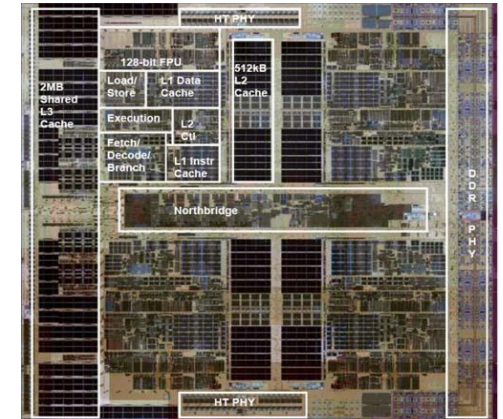
Multi-Threading (MT) possibilities



Fine-grained parallelism: see chapter on GPUs

E. Thread Synchronization

- ▶ For efficiency, OS and hardware should organize this
- ▶ See next part



2. Multicore usage

Example: Sum Reduction

- ▶ Sum 100,000 numbers on 100 processor
UMA
 - Each processor has ID: $0 \leq P_n \leq 99$
 - Partition 1000 numbers per processor
 - Initial summation on each processor

```
sum[Pn] = 0;  
for (i = 1000*Pn; i < 1000*(Pn+1); i++)  
    sum[Pn] += A[i];
```
- ▶ Now need to add these partial sums
 - Reduction: divide and conquer
 - Half the processors add pairs, then quarter, ...
 - Need to synchronize between reduction steps

Example: Sum Reduction

```
half = 100;
repeat
```

```
  synch();
```

```
  if (half%2 != 0 && Pn == 0)
```

```
    sum[0] = sum[0] + sum[half-1];
```

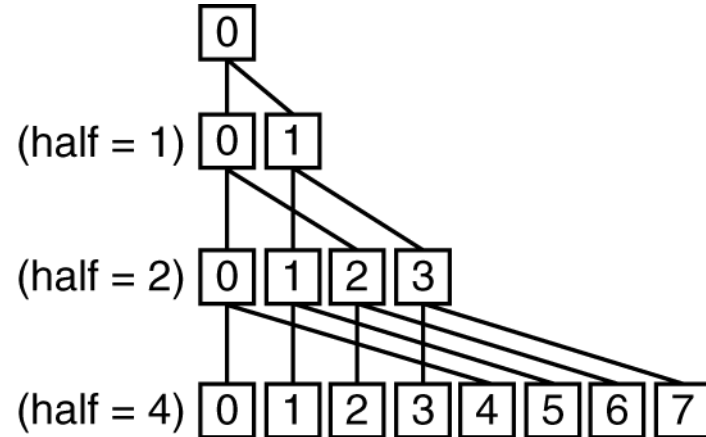
```
    /* Conditional sum needed when half is odd;
```

```
       Processor0 gets missing element */
```

```
  half = half/2; /* dividing line on who sums */
```

```
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
```

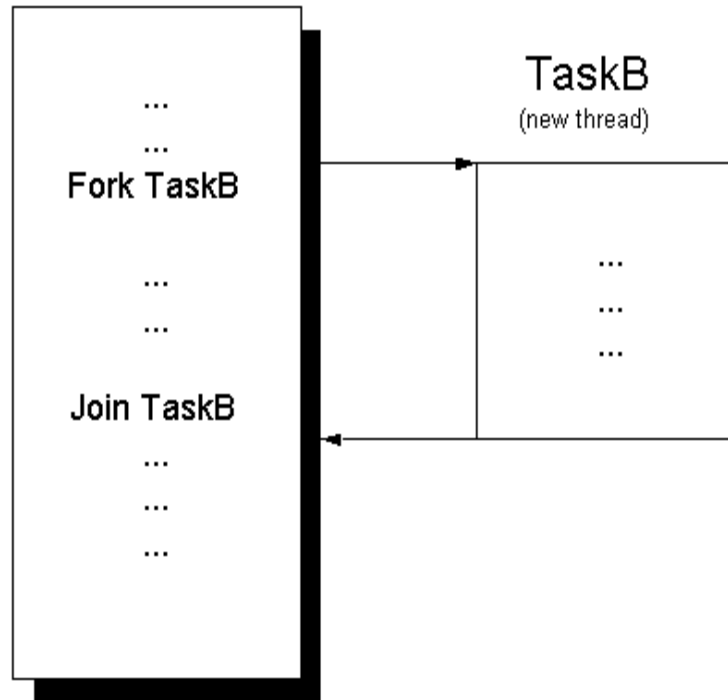
```
until (half == 1);
```



Multi-threading primitives

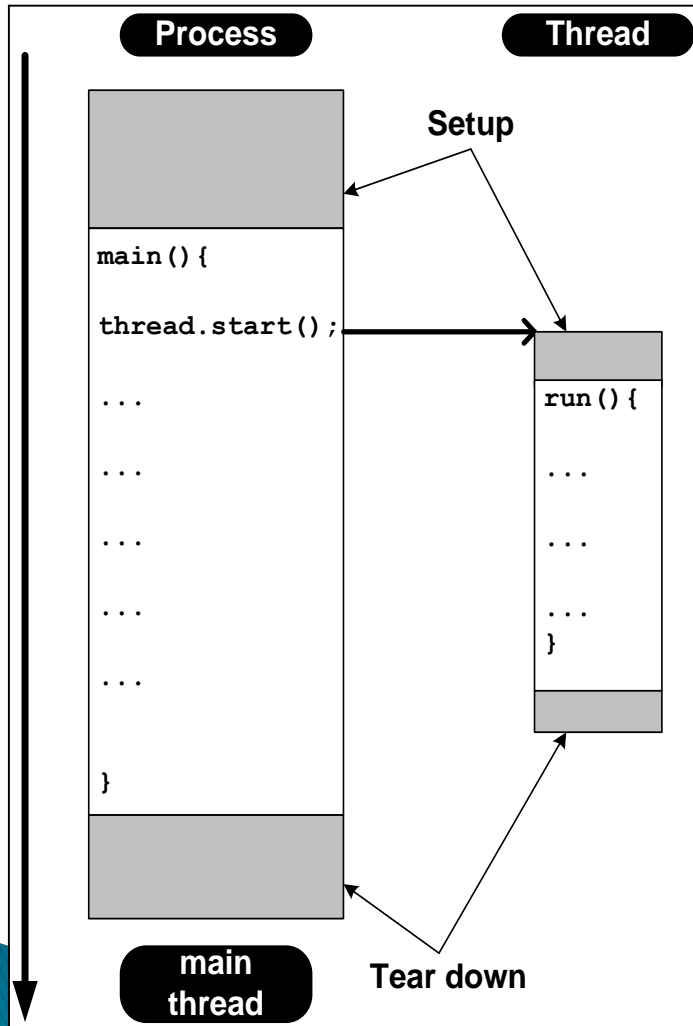
▶ Fork & join

Master process



Parallelism: 2 programs running at the same time

Thread creation



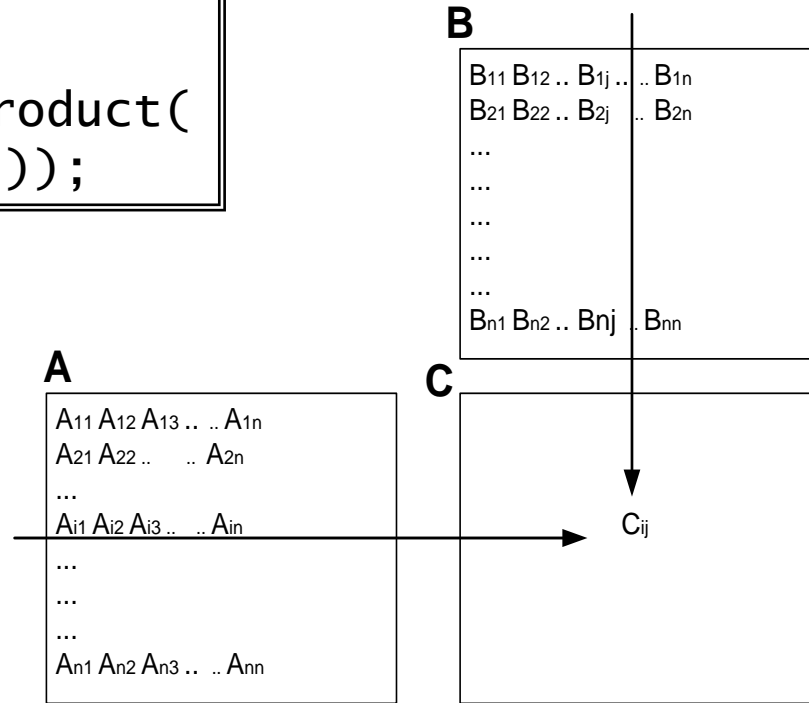
- ▶ A **thread** is basically a *lightweight* process
- ▶ A **process** : unit of resource ownership
 - a virtual address space to hold the process image
 - control of some resources (files, I/O devices...)
- ▶ A **thread** is an execution path
 - Has access to the memory address space and resources of its process. Shares it with other threads.
 - Has its own function call stack.

Example: Matrix Multiplication

```

for (r = 0; r < n; r++)
  for (c = 0; c < n; c++)
    c[r][c] = create_thread(dot_product(
      get_row(a, r), get_col(b, c)));
  
```

- ▶ One thread per C-element
- ▶ Concurrent read must be possible
- ✔ No synchronization necessary
- ✘ Too many threads = a lot of overhead

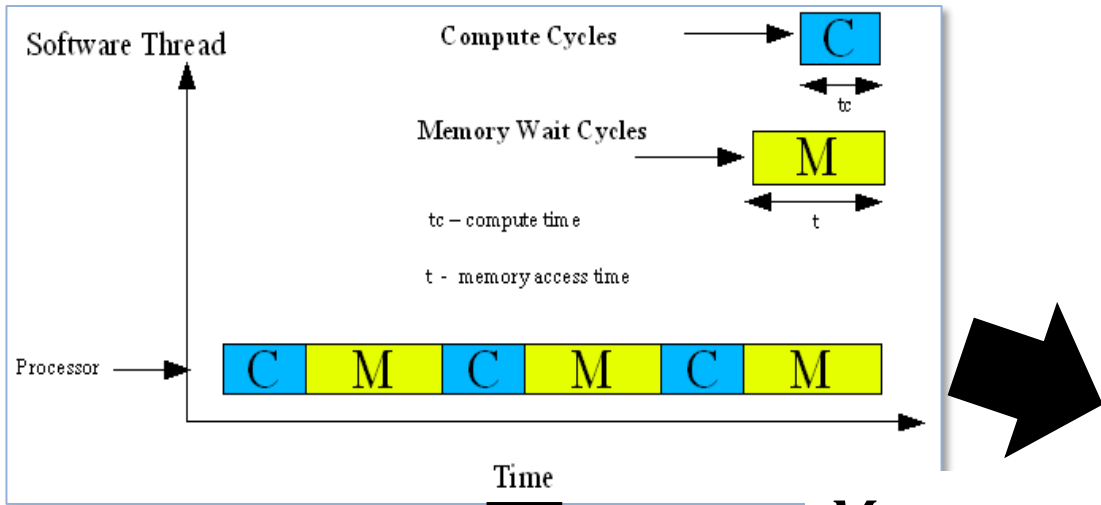


In this case, one may think of the thread as an instance of a function that returns before the function has finished executing.

Why Threads?

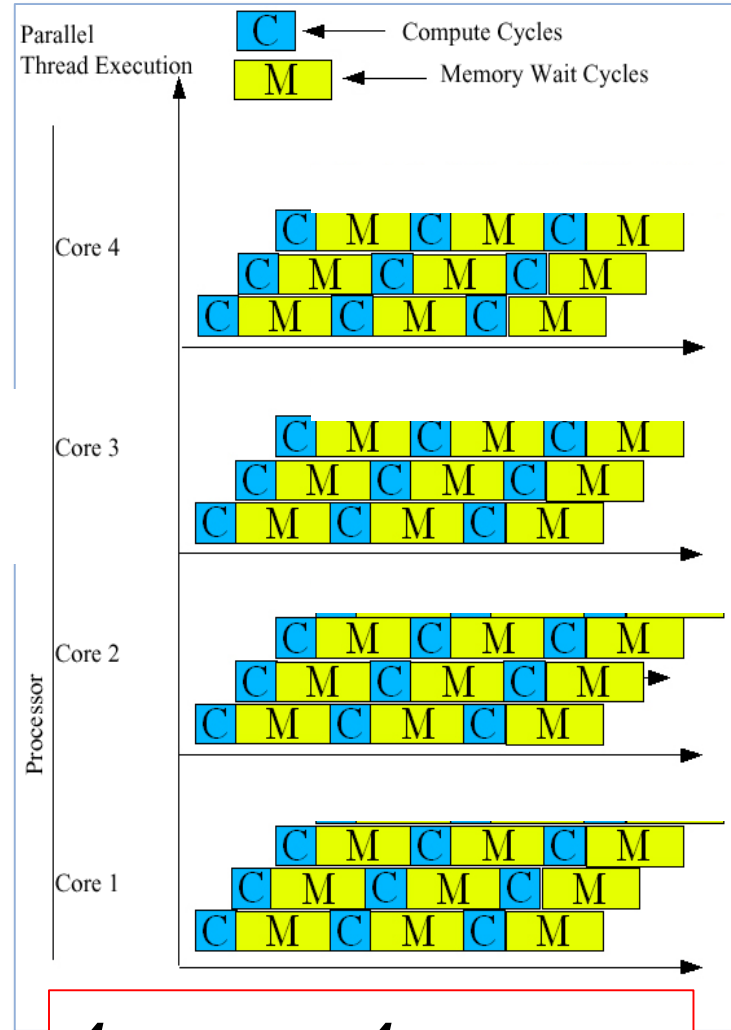
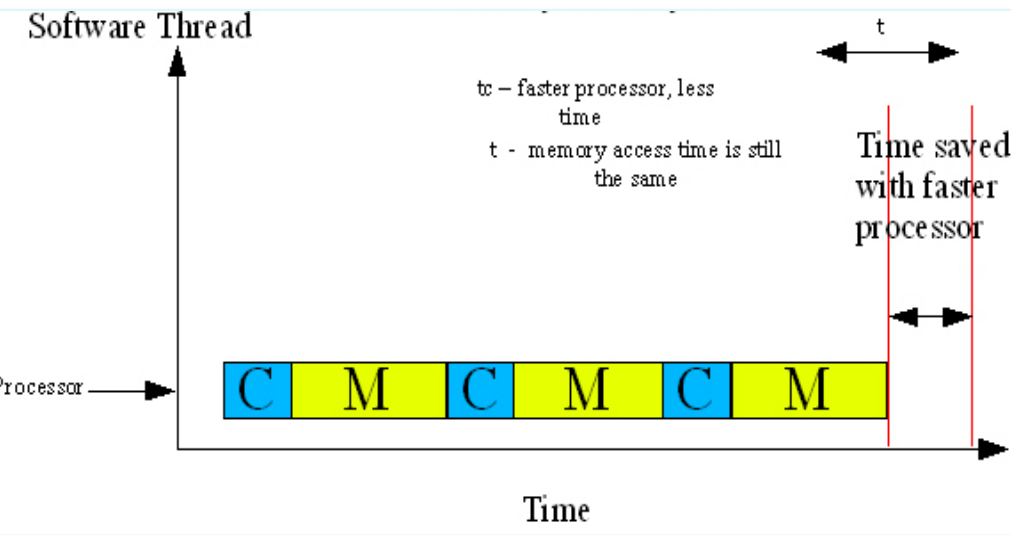
- ▶ **Software Portability**
 - run on serial and parallel machines
- ▶ **Latency Hiding**
 - While one thread has to wait, others can utilize CPU
 - For example: file reading, message reading, reading data from higher-level memory
- ▶ **Scheduling and Load Balancing**
 - Large number of concurrent tasks
 - System-level dynamic mapping to processors
- ▶ **Ease of Programming**
 - Easier to write than message-passing programs (at first sight)

Latency Hiding



Faster CPU

More threads



4 cores: x4
Latency hiding: x3

Multi-threading without speedup

- ▶ Webserver: a thread for each client
 - Multi-threading for convenience LINK 9
 - = distributed computing, not parallel computing
- ▶ But: one can loose performance!
 - 4 requests, each request takes 10 seconds to finish.
 - A single thread: user #1 has to wait 10 seconds, user #2 will wait 20 seconds, user #3 will wait 30 seconds and user #4 will wait 40 seconds.
 - ➔ Average waiting time = 25 seconds
 - Four threads are activated: they must split the available processor time. Each thread will take four times as long. So each request will complete at about 40 seconds.
 - ➔ Waiting time = 40 seconds (+37.5%!)

Example why synchronization is necessary.

- ▶ x is initially set to 1
- ▶ One thread: `x = 10; print(x);`
- ▶ Second thread: `x = 5; print(x);`
- ▶ Both threads are started at the same time
- ▶ What is the output?

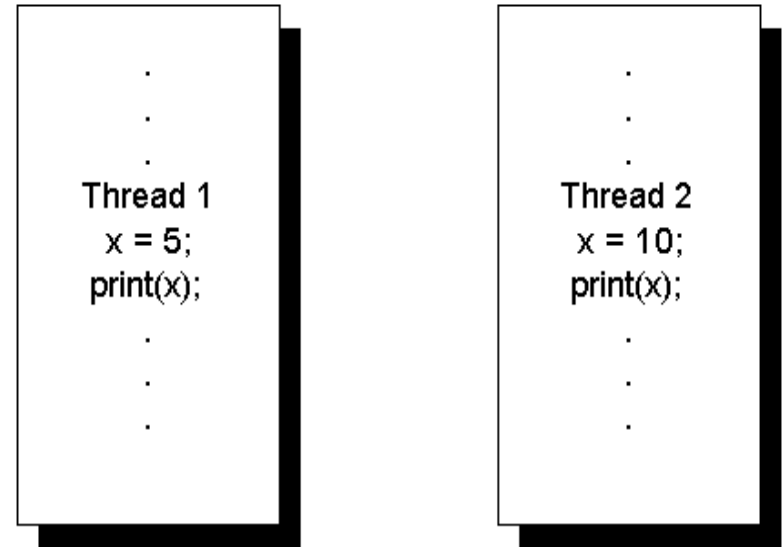
Indeterminism

x

- ▶ When 2 threads run simultaneously, we cannot determine which one is first or which one is faster...

➔ Race condition

- ★ “a flaw in an electronic system or process whereby the output and/or result of the process is unexpectedly and critically dependent on the sequence or timing of other events.”
- ★ The term originates with the idea of two signals *racing each other* to influence the output first.



Results

➔ can be: 5 10 5 10
10 10 5 10

➔ Synchronization necessary

Synchronization of Critical Sections

- ▶ When multiple threads attempt to manipulate the same data item, the results can often be incoherent if proper care is not taken to synchronize them.

- ▶ Example:

critical section

```
/* each thread tries to update variable best_cost */  
if (my_cost < best_cost)  
    best_cost = my_cost;
```

- Assume that there are two threads, the initial value of `best_cost` is 100, and the values of `my_cost` are 50 and 75 at threads `t1` and `t2`.
- Depending on the schedule of the threads, the value of `best_cost` could be 50 or 75!
- The value 75 does not correspond to any serialization of the threads.



Synchronization OK

Thread 1

```
my_cost = 75;  
if (my_cost < best_cost)  
    best_cost = my_cost;
```



shared variable

best_cost = 100

best_cost = 75

best_cost = 50

Thread 2

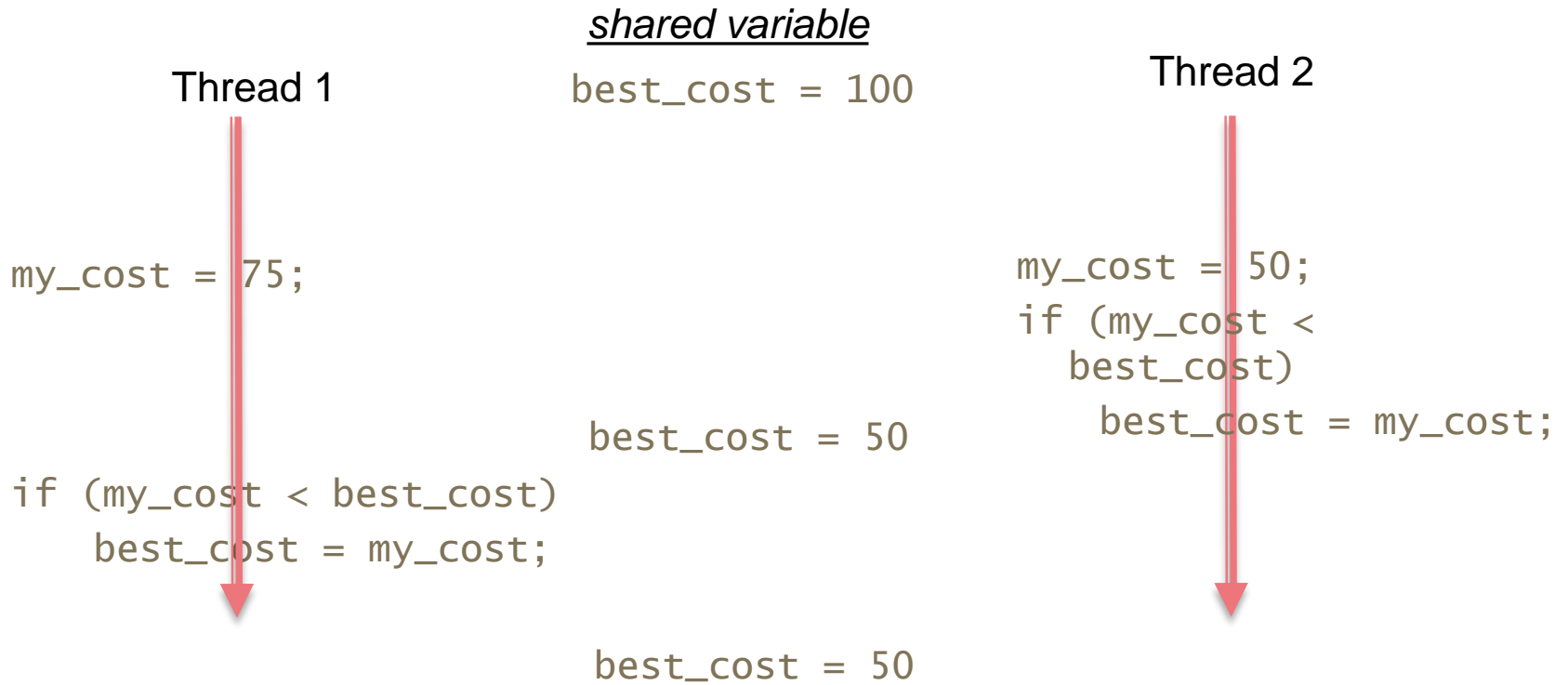
```
my_cost = 50;  
if (my_cost <  
    best_cost)  
    best_cost = my_cost;
```



OK



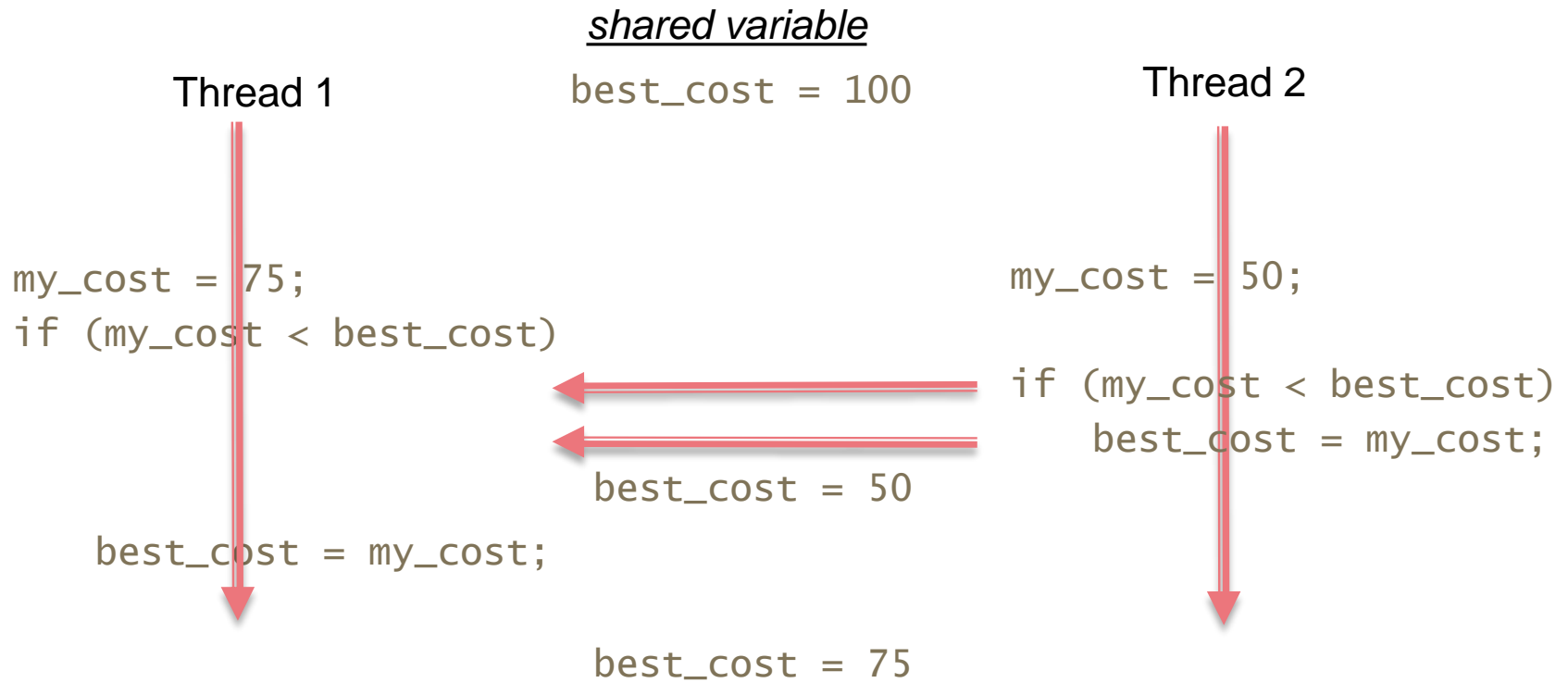
Synchronization OK



OK



Synchronization problem!!



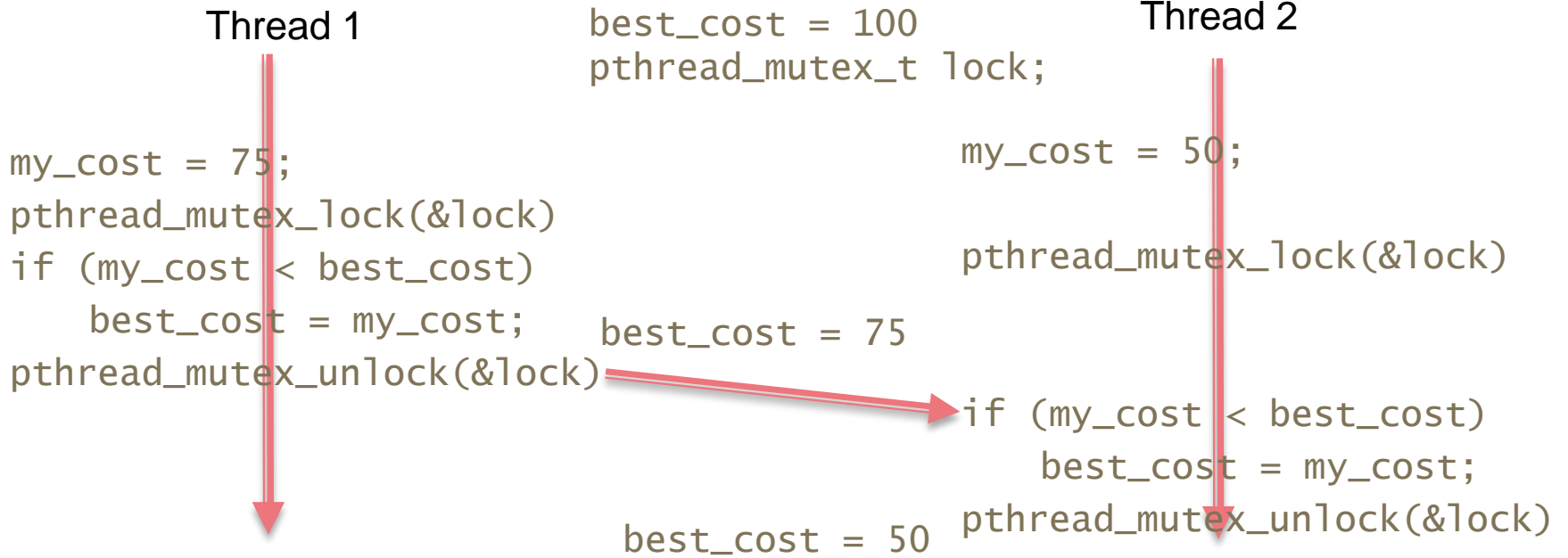
NOK

Happens when the if-then of thread 2 happens in between the if and then of thread 1



Solution: locking of critical sections

shared variables

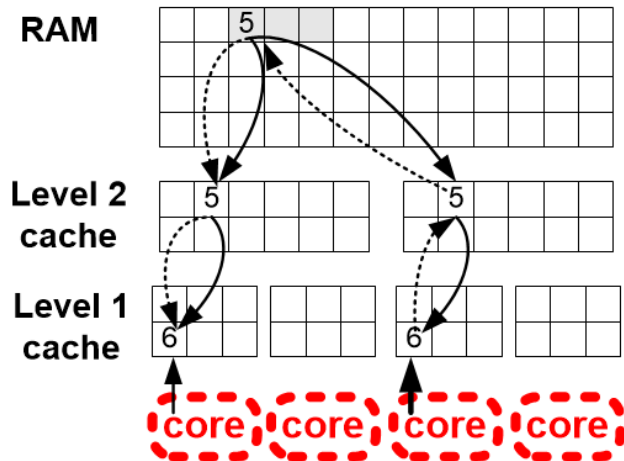


OK

The mutex (mutual exclusion) lock overcomes that 2 threads can simultaneously execute the same critical section, thread 2 is blocked until thread 1 releases the lock.

Updating the same variable by different threads

Example: threads are counting something and increment a common counter



Without synchronization, the data is not immediately updated and you might miss some values. The counter increment is called a **critical section**.

Java Solution (synchronized method):

```
synchronized void addOne(){
    count++;
}
```

A naïve critical section solution

```
boolean access_x=true;

while (!access_x)
    ;
access_x=false;
if (my_cost < best_cost)
    best_cost = my_cost;
access_x=true;
```

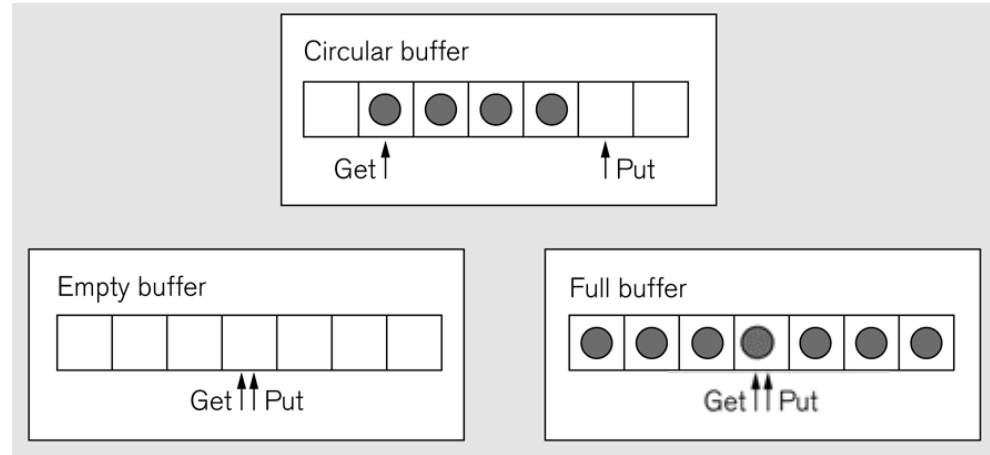
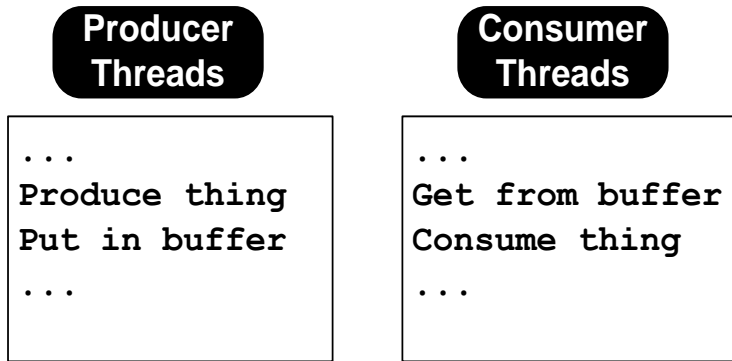
- ▶ Problems:
 - 🗨️ What if `access_x` is accessed at the same time?
 - 🗨️ Thread consumes CPU time while waiting
- ➔ ***Hardware & Operating System support needed!***

*Ps. There is a software solution for this: Peterson Algorithm
(but not efficient)*

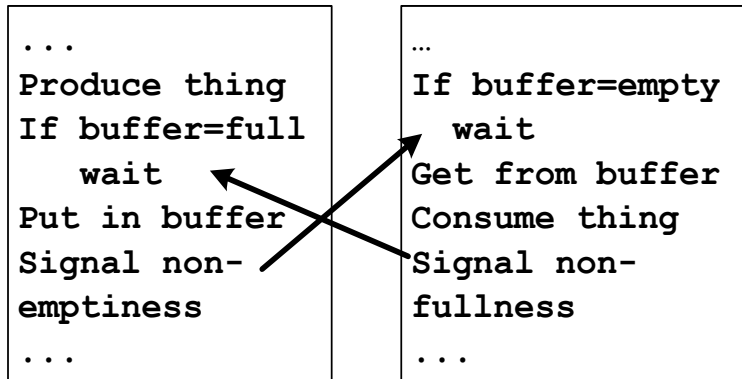
Critical sections trigger cache coherence

- ▶ System will not perform cache coherence all the time
 - Too costly
- ▶ Critical sections indicate shared data

Producers-Consumers Scenario



↓ 1. Thread synchronization



Multi-threading primitives

Should minimally allow the following:

1. Thread creation
2. Locking of critical sections
3. Thread synchronization

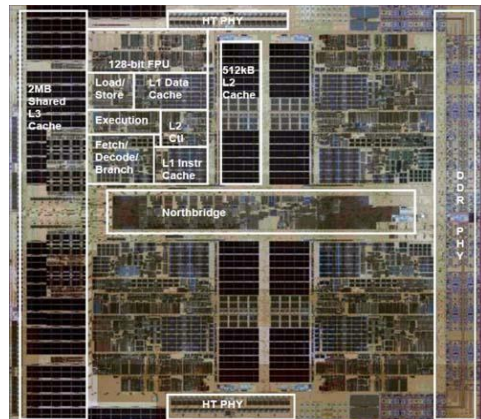
With *primitives* we mean the minimal set of mechanisms (e.g. functions or language constructs) you need to write any multi-threaded program.

Pthreads (C, C++, ...) & Java



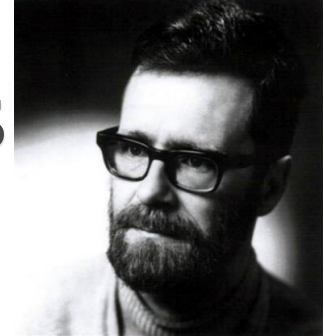
TOT HIER PPP les 1

	Pthreads	Java
How?	library	Built-in language Encapsulation: object manages thread-safety
Thread creation	pthread_create function	Thread class Runnable interface
Critical sections	Locks	Synchronized methods
Thread synchronization	Condition variables	wait & notify



3. Low-level implementation of thread synchronization

A bit of history: Semaphores



1930 – 2002

- ▶ One of the first concepts for both *critical sections* & *thread synchronization*.
- ▶ Invented by Dutch computer scientist *Edsger Dijkstra*.
- ▶ found widespread use in a variety of operating systems as basic primitive for avoiding race conditions.
- ▶ Based on a protected variable for controlling access by multiple processes to a common resource
- ▶ By atomic operations you can decrement or increment semaphores
- ▶ **binary** (flag) or integer (counting)
 - *When binary*: similar to mutexes
 - *When integer*: The value of the semaphore S is the number of units of the resource that have not been claimed.

Problems to solve

1. Atomicity
2. Spinlock versus thread inactivation
3. Cache coherence
4. Instruction reordering

Hardware instructions?

Implementation in hardware?

Role operating system?

Test-and-set

- ▶ to write to a memory location and return its old value as a single atomic (i.e., non-interruptible) operation.
- ▶ If multiple processes may access the same memory location, and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process is done.
- ▶ CPUs may use test-and-set instructions offered by other electronic components, such as dual-port RAM

Implementing mutual exclusion with test-and-set

```
boolean lock = false
function Critical(){
    while TestAndSet(lock, true)
        skip // spin until lock is acquired
    critical section // only one process can be in
this section at a time
    lock = false // release lock when finished with
the critical section
}
```

Synchronization in MIPS

- ▶ Load linked: `ll rt, offset(rs)`
 - `rs`: address
- ▶ Store conditional: `sc rt, offset(rs)`
 - Tries to write value `rt` to address `rs`
 - Succeeds if location not changed since the `ll`
 - Returns 1 in `rt` (side-effect: `rt` = flag to indicate success)
 - Fails if location is changed
 - Returns 0 in `rt`
- ▶ Implemented in hardware with a bit called *LLbit*, which is set to zero if value at location has changed
 - After a store conditional, the LLbit will be set to 0 at the other locations (cache copies) using the same `rs`
 - => their `sc` will fail
 - This is managed by the cache coherence system

See <https://www.cs.auckland.ac.nz/courses/compsci313s2c/resources/MIPSELLSC.pdf>

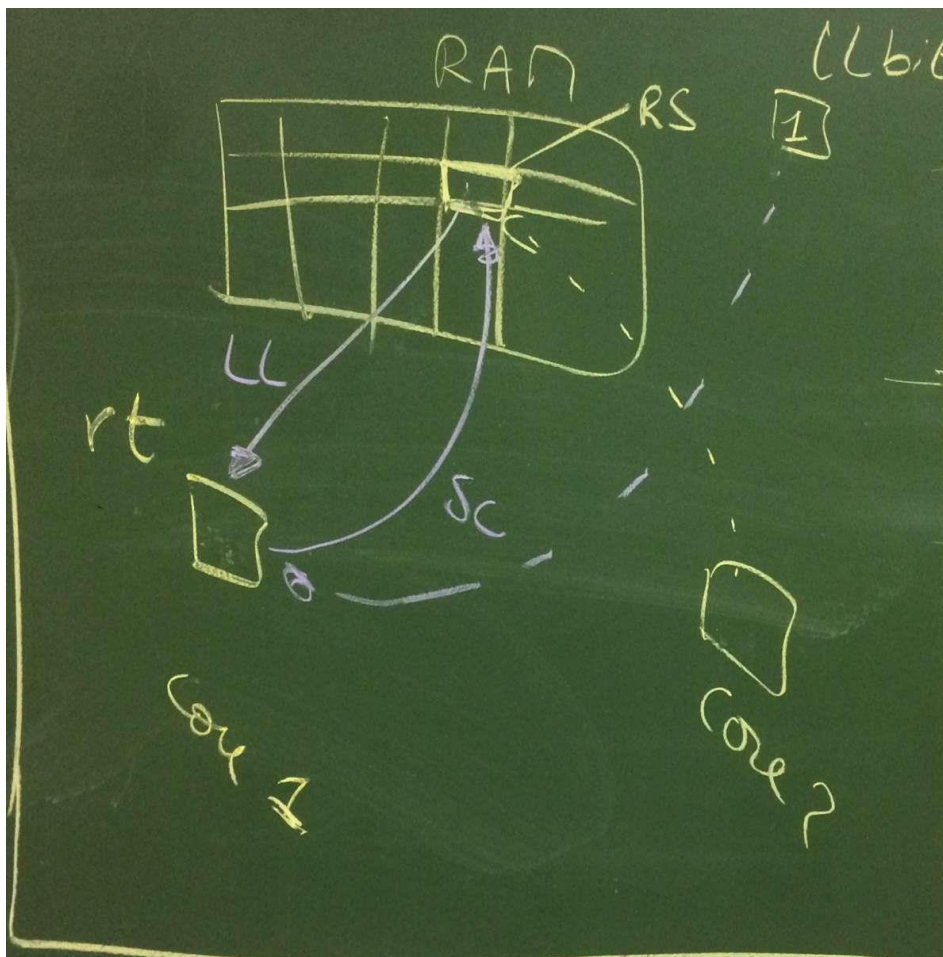
Test-and-set with LL and SC

TestAndSet

argumenten: \$a0 (lock variabele) en \$a1 (waarde)
teruggeefwaarde: register \$v0

```
try: add $t0,$zero,$a1 ;copy set-value
      ll  $t1,0($a0)    ;load linked
      sc  $t0,0($a0)    ;store conditional
      beq $t0,$zero,try ;branch store fails
      add $v0,$zero,$t1 ;put loaded value in $v0
```

Test-and-set with LL and SC



Atomic swap in MIPS

- ▶ Example: atomic swap of \$s4 and \$s1
 - Atomic: \$s4 and \$s1 are not changed by another thread during the swap
 - Application: to test/set lock variable

```
try: add $t0,$zero,$s4 ;copy exchange value
      ll  $t1,0($s1)    ;load linked
      sc  $t0,0($s1)    ;store conditional
      beq $t0,$zero,try ;branch store fails
      add $s4,$zero,$t1 ;put load value in $s4
```

1 a. Atomicity on single cores

- ▶ **Prevent interrupts (thread switching) within critical sections**

1 b. Atomicity on multicores

- ▶ Memory fences (barriers) necessary
 - Memory fence = a type of *barrier instruction* that causes a central processing unit (CPU) or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction.
 - => the operations issued prior to the barrier are guaranteed to be performed before operations issued after the barrier.
- ▶ ***Implemented via cache coherence system***

2. Spinning vs switching

- ▶ Spinning for short locks
- ▶ Switching for long locks
 - Eg: when inactive thread has to release the lock!
- ▶ In practice: spinning with timer interrupt to suspend thread
- ▶ Operating system ensures switching
 - Smart thread scheduling

3. Cache coherence

- ▶ memory fences via snoop bus
- ▶ Claim ownership on cache line
- ▶ Either waiting for response
- ▶ Either tag cache line and check later
 - See LL and SC instructions of MIPS
- ▶ IO must also be connected to the cache coherence system to know whether RAM is up-to-date (e.g. When memory-mapped IO)

4. Thread reordering

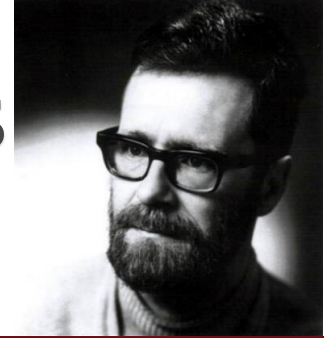
- ▶ Initially, $A = B = 0$

- ▶ Then:

Thread 1	Thread 2
1: $r2 = A;$	3: $r1 = B;$
2: $B = 1;$	4: $A = 2;$

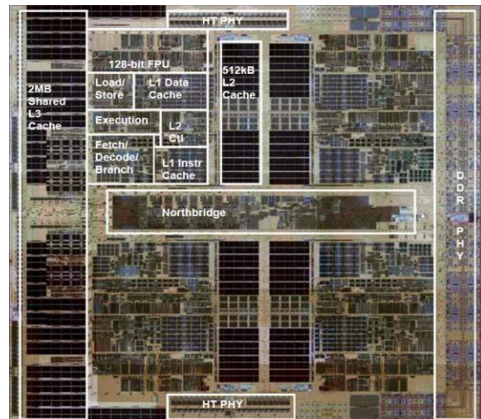
- ▶ Value of $r1$ and $r2$?
- ▶ End result $r2 == 2, r1 == 1$ is possible!!
 - Compilers are allowed to reorder the instructions in either thread, when this does not affect the execution of that thread in isolation (being independent)
 - Reordering instructions might improve performance
- ▶ This can be solved by adding a memory fence, which will act as a barrier

A bit of history: Semaphores



1930 – 2002

- ▶ One of the first concepts for critical sections & thread synchronization.
- ▶ Invented by Dutch computer scientist *Edsger Dijkstra*.
- ▶ found widespread use in a variety of operating systems as basic primitive for avoiding race conditions.
- ▶ Based on a protected variable for controlling access by multiple processes to a common resource
- ▶ By atomic operations you can decrement or increment semaphores
- ▶ binary (flag) or integer (counting)
 - *When binary*: similar to mutexes
 - *When integer*: The value of the semaphore S is the number of units of the resource that have not been claimed.



4. Java threads

The Java Thread Class

```
public synchronized void start()
```

- Starts this Thread and returns immediately after invoking the run() method.
- Throws `IllegalThreadStateException` if the thread was already started.

```
public void run()
```

- The body of this Thread, which is invoked after the thread is started.

```
public final synchronized void join(long millis)  
throws InterruptedException
```

- Waits for this Thread to die. A timeout in milliseconds can be specified, with a timeout of 0 milliseconds indicating that the thread will wait forever.

```
public static void yield()
```

- Causes the currently executing Thread object to yield the processor so that some other runnable Thread can be scheduled.

```
public final int getPriority()
```

- Returns the thread's priority.

```
public final void setPriority(int newPriority)
```

- Sets the thread's priority.

Thread creation

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }
    public void run() {
        // compute primes larger
        // than minPrime
        . . .
    }
}
```



```
PrimeThread p = new PrimeThread(143);
p.start();
```

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }
    public void run() {
        // compute primes larger
        // than minPrime
        . . .
    }
}
```



```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

Synchronized methods & blocks

```
synchronized void updateCost(int my_cost){
    if (my_cost < best_cost)
        best_cost = my_cost;
}
```

is identical to

```
Synchronized(object) {
    if (my_cost <
best_cost)
        best_cost =
my_cost;
}
```

```
void updateCost(int my_cost){
    Synchronized(this) {
        if (my_cost < best_cost)
            best_cost = my_cost;
    }
}
```

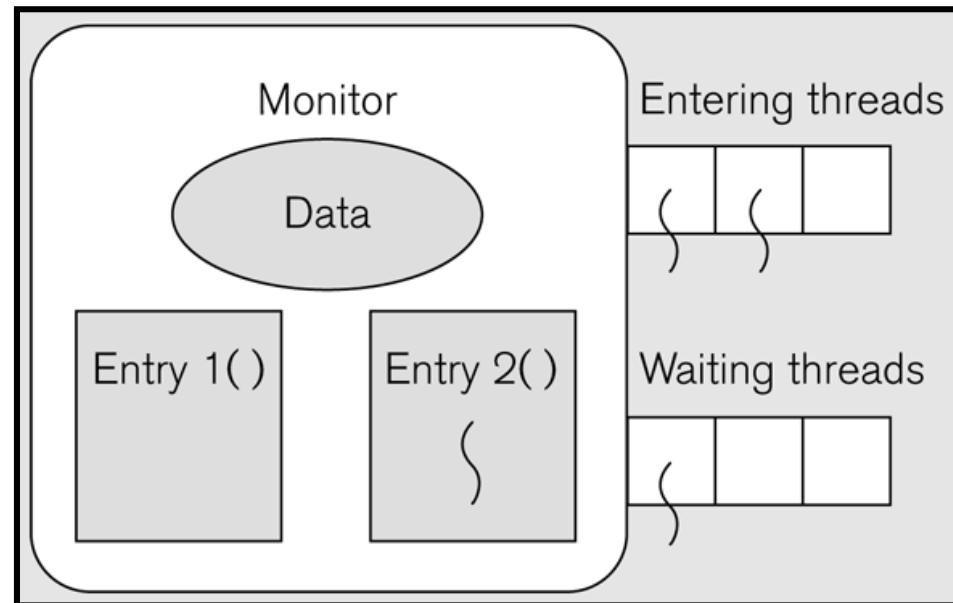
Static methods

```
synchronized static void
method(){
    ...
}
```

synchronized on the associated 'Class' object:
<theClass>.class is used for locking

Java objects act as Monitors

- ▶ When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
 - ▶ When a synchronized method exits, the new state of the object are visible to all threads.
- ➔ *Thread synchronization happens through objects.*



Example: Counting 3s

n data values x_0, \dots, x_n
in array *array*

```
count=0;  
for (int i=0;i<array.length;i++)  
  if (array[i] == 3)  
    count++;
```

- ▶ Parallelism? Yes.
- ▶ Multithreaded solution: divide counting

Parallel Counting 3s (wrong version)

```
count=0;
Thread[] threads = new Thread[nbrThreads];
for(int t=0;t<nbrThreads;t++){
    final int T = t;
    threads[t] = new Thread(){
        public void run(){
            int length_per_thread=array.length/ nbrThreads;
            int start=T*length_per_thread;
            for(int i=start;i<start+length_per_thread; i++)
                if (array[i] == 3)
                    count++;
        }
    };
    threads[t].start();
}
// wait until all threads have finished
for(int t=0;t<nbrThreads;t++)
    try {
        threads[t].join();
    } catch (InterruptedException e) {}
```



Parallel Counting 3s: experiments

On a dual core processor

Counting 3s in an array of 1000 elements and 4 threads:

- * Seq : counted 100 3s in 234us
- * Par 1: counted 100 3s in 3ms 615us
- * Par 2: counted 100 3s in 13ms 83us
- * Par 3: counted 100 3s in 5ms 23us
- * Par 4: counted 100 3s in 3ms 845us

Counting 3s in an array of 40000000 elements and 4 threads:

- * Seq : counted 4000894 3s in 147ms
- * Par 1: counted 3371515 3s in 109ms
- * Par 2: counted 4000894 3s in 762ms
- * Par 3: counted 4000894 3s in 93ms 748us
- * Par 4: counted 4000894 3s in 77ms 14us

Parallel Counting 3s II

```
synchronized void addOne(){ count++; }
```

```
count=0;
final int NBR_THREADS = nbrThreads;
Thread[] threads = new Thread[nbrThreads];
for(int t=0;t<nbrThreads;t++){
    final int T = t;
    threads[t] = new Thread(){
        public void run(){
            int length_per_thread=array.length/NBR_THREADS;
            int start=T*length_per_thread;
            for(int i=start;i<start+length_per_thread; i++)
                if (array[i] == 3)
                    addOne();        }
    };
    threads[t].start();
}
// wait until all threads have finished
for(int t=0;t<nbrThreads;t++)
    try {
        threads[t].join();
    } catch (InterruptedException e) {}
```

- ▶ Problem in previous: access to the same data
- ▶ Solution: synchronized method

Parallel Counting 3s III

PPP 33

```
synchronized void addCount(int n){ count+=n; }
```

```
count=0;
final int NBR_THREADS = nbrThreads;
Thread[] threads = new Thread[nbrThreads];
for(int t=0;t<nbrThreads;t++){
    final int T = t;
    threads[t] = new Thread(){
        int private_count=0;
        public void run(){
            int length_per_thread=array.length/NBR_THREADS;
            int start=T*length_per_thread;
            for(int i=start;i<start+length_per_thread; i++)
                if (array[i] == 3)
                    private_count++;
                    addCount(private_count);
        }
    };
    threads[t].start();
}
// wait until all threads have finished
for(int t=0;t<nbrThreads;t++)
    threads[t].join();
```

- ▶ Problem in previous:
 - locking overhead
 - lock contention
 - cache coherence overhead

- ▶ Solution: Use local subtotals

Parallel Counting 3s IV

PPP 34

```
synchronized void addCount(int n){ count+=n; }

count=0;
final int NBR_THREADS = nbrThreads;
Thread[] threads = new Thread[nbrThreads];
for(int t=0;t<nbrThreads;t++){
    final int T = t;
    threads[t] = new Thread(){
        int private_count=0;
        int p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13, p14, p15;
        public void run(){
            int length_per_thread=array.length/NBR_THREADS;
            int start=T*length_per_thread;
            for(int i=start;i<start+length_per_thread; i++)
                if (array[i] == 3)
                    private_count++;
            addCount(private_count);
        }
    };
    threads[t].start();
}
// wait until all threads have finished
for(int t=0;t<nbrThreads;t++)
    threads[t].join();
```

- ▶ Problem in previous: false sharing (see earlier slide)
- ▶ Solution: padding


Volatile Variables

- ▶ The Java language allows threads to keep private working copies of these variables (= caching). This enables a more efficient execution of the two threads. For example, when each thread reads and writes these variables, they can do so on the private working copies instead of accessing the variables from main memory. The private working copies are reconciled with main memory only at specific synchronization points.
- ▶ **Volatile variables:** Private working memory is reconciled with main memory on each variable access.
= Light-weight synchronization


Only for atomic operations

Which code is thread-safe?


```
volatile int x;
...
X++;
...
```



```
volatile int x;
...
X=5;
...
```




```
volatile int best_cc;
...
if (my_cost <
    best_cost)
    best_cost = my_cost;
...
```



```
volatile int lower, upper;

public void setLower(int value)
    if (value > upper)
        throw new
        IllegalArgumentException(...);
        lower = value;
}

public void setUpper(int value) {
    if (value < lower)
        throw new
        IllegalArgumentException(...);
        upper = value;
}
```



Conditions:

1. Writes to the variable do not depend on its current value.
2. The variable does not participate in invariants with other variables

Incorrectly synchronized programs exhibit surprising behaviors

- ▶ Initially, $A = B = 0$
- ▶ Then:

Thread 1	Thread 2
1: $r2 = A;$	3: $r1 = B;$
2: $B = 1;$	4: $A = 2;$

- ▶ End result
 - *Compilers are allowed to reorder the instructions in either thread, when this does not affect the execution of that thread in isolation (being independent)*
 - *Reordering instructions might improve performance*

The Java Memory Model

- ▶ Describes how threads interact through memory.
- ▶ Specifies the legal behaviors for a multithreaded program.
- ▶ The compiler/virtual machine is allowed to make optimizations.
- ▶ Tries to provide *safety*, but also *flexibility* (allowing optimizations to improve performance).
 - Trade-off!

Thread Synchronization

Via Object class

- ▶ `public final void wait()` throws [InterruptedException](#)
 - Causes the current thread to wait until another thread invokes the [notify\(\)](#) method or the [notifyAll\(\)](#)
 - The current thread must own this object's monitor. The thread releases ownership of this monitor
- ▶ `public final void wait(long timeout, int nanos)` throws [InterruptedException](#)
- ▶ `public final void notify()`
 - Wakes up a single thread that is waiting on this object's monitor.
 - *The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.*
- ▶ `public final void notifyAll()`

Put synchronization in critical section

Producer Threads

```

...
Produce thing
while buffer=full
    wait()
Put in buffer
notify()
...
    
```



Consumer Threads

```

...
while buffer=empty
    wait()
Get from buffer
notify()
Consume thing
...
    
```

OK?

Race condition possible!

```

synchronized void put()
{
    while buffer=full
        wait()
    Put in buffer
    notify()
}
    
```



```

synchronized void get()
{
    while buffer=empty
        wait()
    Get from buffer
    notify()
}
    
```

Lock is released on wait()

Vector versus ArrayList

- ▶ Vector is synchronized, ArrayList is not
- ▶ Only one thread:
 - Reported: Vector is slower <> my tests: no difference
 - ➔ Recent java versions automatically choose best version
- ▶ Multiple threads:
 - Vector OK
 - Use `Collections.synchronizedList(new ArrayList(...)) ;`

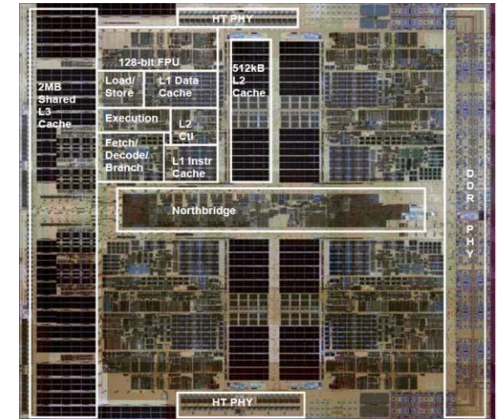
Atomic Objects

<http://java.sun.com/docs/books/tutorial/essential/concurrency/atomicvars.html>

- ▶ Liveness problem:
 - Waiting threads due to (unnecessary) synchronization

More Advanced ...

- ▶ Explicit lock objects
 - tryLock(): provides means to back out of lock
- ▶ Executors: more advanced threads
 - Thread pools: reuse of finished threads
- ▶ Concurrent Collections: concurrent data structures that can be accessed by multiple threads simultaneously
 - BlockingQueues
 - ConcurrentMa



POSIX Threads

The POSIX Thread API

- ▶ Commonly referred to as **Pthreads**, **POSIX** has emerged as the standard threads API (1995), supported by most vendors.
- ▶ The concepts discussed here are largely independent of the API and can be used for programming with other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.

pthread: Creation and Termination

```
#include <pthread.h>
```

```
int pthread_create (pthread_t *thread_handle, const  
pthread_attr_t *attribute, void * (*thread_function)(void *),  
void *arg);
```

```
int pthread_join ( pthread_t thread, void **ptr);
```

- ▶ The function `pthread_create` invokes function `thread_function` as a thread.
- ▶ The function `pthread_join` waits for the thread to be finished and the value passed to `pthread_exit` (by the terminating thread) is returned in the location pointer `**ptr`.

Example

```
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512
void *compute_pi (void *);

main() {
    pthread_t p_threads[MAX_THREADS];
    pthread_attr_t attr;
    pthread_attr_init (&attr);
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
}
```

Executed on a 4-processor SGI
Origin: speedup of 3.91 with 32
threads.

This corresponds to a parallel
efficiency of 0.98!

Mutual Exclusion

- ▶ The code in the previous example corresponds to a *critical segment* or *critical section*; i.e., a segment that must be executed by only one thread at any time.
- ▶ Critical segments in Pthreads are implemented using *mutex locks*.
- ▶ Mutex-locks have two states: locked and unlocked. At any point of time, only one thread can lock a mutex lock. A lock is an atomic operation.
- ▶ A thread entering a critical segment first tries to get a lock. It goes ahead when the lock is granted. Otherwise it is blocked until the lock relinquished.

Mutual Exclusion

- ▶ The pthreads API provides the following functions for handling mutex-locks:
 - `int pthread_mutex_init (pthread_mutex_t *mutex_lock, const pthread_mutexattr_t *lock_attr);`
 - `int pthread_mutex_lock (pthread_mutex_t *mutex_lock);`
 - `int pthread_mutex_unlock (pthread_mutex_t *mutex_lock);`

Lock critical sections

- ▶ We can now write our previously incorrect code segment as:

```
pthread_mutex_t minimum_value_lock;
...
main() {
    ....
    pthread_mutex_init(&minimum_value_lock, NULL);
    ....
}
void *find_min(void *list_ptr) {
    ....
    pthread_mutex_lock(&minimum_value_lock);
    if (my_min < minimum_value)
        minimum_value = my_min;
    /* and unlock the mutex */
    pthread_mutex_unlock(&minimum_value_lock);
}
```

Disadvantages lock

- ▶ Deadlock possible, see later
- ▶ Performance degradation
 - Due to locking overhead
 - Due to idling of locked threads (if no other thread is there to consume available processing time)
- ➔ Alleviate locking overheads
- ▶ Minimize size of critical sections
 - Encapsulating large segments of the program within locks can lead to significant performance degradation.
 - `create_task()` and `process_task()` are left outside critical section!

Alleviate locking overheads

- ▶ Test a lock:
 - `int pthread_mutex_trylock (pthread_mutex_t *mutex_lock);`
 - Returns 0 if locking was successful, EBUSY when already locked by another thread.
- ▶ `pthread_mutex_trylock` is typically much faster than `pthread_mutex_lock` since it does not have to deal with queues associated with locks for multiple threads waiting on the lock.
- ▶ *Example:* write result to global data if lock can be acquired, otherwise temporarily store locally

KUMAR: 'Finding matches in a list'

Condition Variables for Synchronization

- ▶ A condition variable allows a thread to block itself until specified data reaches a predefined state.
- ▶ A condition variable is associated with this predicate. When the predicate becomes true, the condition variable is used to signal one or more threads waiting on the condition.
- ▶ A single condition variable may be associated with more than one predicate.
- ▶ A condition variable always has a *mutex* associated with it. A thread locks this mutex and tests the predicate defined on the shared variable.
- ▶ If the predicate is not true, the thread waits on the condition variable associated with the predicate using the function `pthread_cond_wait`.

Synchronization in Pthreads

- ▶ Pthreads provides the following functions for condition variables:

```
int pthread_cond_wait(pthread_cond_t *cond,  
    pthread_mutex_t *mutex);
```

```
int pthread_cond_signal(pthread_cond_t *cond);
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

```
int pthread_cond_init(pthread_cond_t *cond,  
    const pthread_condattr_t *attr);
```

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Producer–consumer work queues

- ▶ The **producer threads** create tasks and inserts them into a work queue.
- ▶ The **consumer threads** pick up tasks from the queue and executes them.
- ▶ Synchronization!

Producer–Consumer Using Locks

- ▶ The producer–consumer scenario imposes the following constraints:
- ▶ The producer thread must not overwrite the shared buffer when the previous task has not been picked up by a consumer thread.
- ▶ The consumer threads must not pick up tasks until there is something present in the shared data structure.
- ▶ Individual consumer threads should pick up tasks one at a time.


```

1 pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
2 pthread_cond_t nonempty=PTHREAD_COND_INITIALIZER;
3 pthread_cond_t nonfull=PTHREAD_COND_INITIALIZER;
4 Item buffer[SIZE];
5 int put=0; // Buff index for next insert
6 int get=0; // Buff index for next remove
7
8 void insert(Item x) // Producer thread
9 {
10 pthread_mutex_lock(&lock);
11 while((put>get&&(put-get)==SIZE-1)|| // While buffer is
12 (put<get&&(get-put)==1)) // full
13 {
14 pthread_cond_wait(&nonfull, &lock);
15 }
16 buffer[put]=x;
17 put=(put+1)%SIZE;
18 pthread_cond_signal(&nonempty);
19 pthread_mutex_unlock(&lock);
20 }
21
22 Item remove() // Consumer thread
23 {
24 Item x;
25 pthread_mutex_lock(&lock);
26 while(put==get) // While buffer is empty
27 {
28 pthread_cond_wait(&nonempty, &lock);
29 }
30 x=buffer[get];
31 get=(get+1)%SIZE;
32 pthread_cond_signal(&nonfull);
33 pthread_mutex_unlock(&lock);
34 return x;
35 }

```

Small mistake in PPP on page 170
Thanks to Xuyang Feng, 2014

Controlling Thread and Synchronization Attributes

- ▶ The Pthreads API allows a programmer to change the default properties of entities (thread, mutex, condition variable) using *attributes objects*.
- ▶ An attributes object is a data-structure that describes entity properties.
- ▶ Once these properties are set, the attributes object can be passed to the method initializing the entity.
- ▶ Enhances modularity, readability, and ease of modification.

Attributes Objects for Threads

- ▶ Use `pthread_attr_init` to create an attributes object.
- ▶ Individual properties associated with the attributes object can be changed using the following functions:
 - ✦ `pthread_attr_setdetachstate`,
 - ✦ `pthread_attr_setguardsize_np`,
 - ✦ `pthread_attr_setstacksize`,
 - ✦ `pthread_attr_setinheritsched`,
 - ✦ `pthread_attr_setschedpolicy`,
 - ✦ `pthread_attr_setschedparam`

Threads locks multiple times

```
pthread_mutex_lock(&lock1);  
...  
pthread_mutex_lock(&lock1);  
...  
pthread_mutex_unlock(&lock1);  
...  
pthread_mutex_unlock(&lock1);
```

E.g. happens when in one critical section we call code with also a critical section protected by the same lock

What will happen?

➤ *depends on type of lock*

Types of Mutexes

- ▶ Pthreads supports three types of mutexes – normal, recursive, and error-check.
 - A normal mutex deadlocks if a thread that already has a lock tries a second lock on it. *This is the default.*
 - A recursive mutex allows a single thread to lock a mutex as many times as it wants. It simply increments a count on the number of locks. A lock is relinquished by a thread when the count becomes zero.
 - An error check mutex reports an error when a thread with a lock tries to lock it again (as opposed to deadlocking in the first case, or granting the lock, as in the second case).
- ▶ The type of the mutex can be set in the attributes object before it is passed at time of initialization.

Attributes Objects for Mutexes

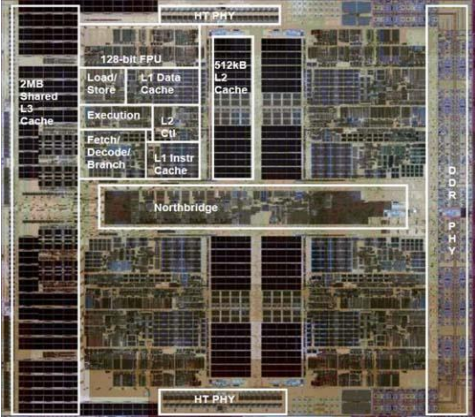
- ▶ Initialize the attributes object using function:
`pthread_mutexattr_init.`
- ▶ The function `pthread_mutexattr_settype_np` can be used for setting the type of mutex specified by the mutex attributes object.

```
pthread_mutexattr_settype_np (  
pthread_mutexattr_t *attr,  
int type);
```
- ▶ Here, `type` specifies the type of the mutex and can take one of:
 - `PTHREAD_MUTEX_NORMAL_NP`
 - `PTHREAD_MUTEX_RECURSIVE_NP`
 - `PTHREAD_MUTEX_ERRORCHECK_NP`

Thread Cancellation

```
int pthread_cancel(pthread_t *thread);
```

- ▶ Terminates another thread
- ▶ Can be dangerous
 - In java: deprecated *suspend()* method. Use of it is discouraged.
 - But sometimes useful, e.g. as long as the user is staying at a certain view in your application, you calculate extra information, as soon as he leaves the view, you stop the calculation.
- ▶ A thread can protect itself against cancellation
- ▶ `pthread_exit`: exit thread (yourself) without exiting the process



Thread Safety

Adaptable range-object

- ▶ Object specifies a range with a lower and upper attribute.
 - Invariant (should always be true): $\text{lower} \leq \text{upper}$
 - Make thread-safe! Check the invariant.

```
volatile int lower, upper;  
  
public void setLower(int value) {  
    if (value > upper)  
        throw new IllegalArgumentException(...);  
    lower = value;  
}  
public void setUpper(int value) {  
    if (value < lower)  
        throw new IllegalArgumentException(...);  
    upper = value;  
}
```

Condition variables & locking

- ▶ Condition variables should be protected by a lock
 - Signal of non-emptiness can happen just between check and when consumer thread goes into waiting
- ▶ Should the signal also be protected by the lock?
 - No

Thread-safe?

Mistake in PPP on page 173!!



```
pthread_mutex_lock(&lock);
while (apples==0)
    pthread_cond_wait(&more_apples, &lock);
while (oranges==0)
    pthread_cond_wait(&more_oranges,
&lock);
// eat apple & orange
pthread_mutex_unlock(&lock);
```

NOK!!



```
pthread_mutex_lock(&lock);
while (apples==0 || oranges==0){
    pthread_cond_wait(&more_apples, &lock);
    pthread_cond_wait(&more_oranges,
&lock);
}
// eat apple & orange
pthread_mutex_unlock(&lock);
```

Still NOK!

Thread-safe!



```
pthread_mutex_lock(&lock);
boolean allConditionsPassed;
do {
    allConditionsPassed = true;
    if (apples == 0){
        pthread_cond_wait(&more_apples, &lock);
        allConditionsPassed = false; }
    if (oranges == 0){
        pthread_cond_wait(&more_oranges, &lock);
        allConditionsPassed = false; }
} while (!allConditionsPassed);
// eat apple & orange
pthread_mutex_unlock(&lock);
```

By the boolean, you can easily add more conditions. Also OK, no boolean: } while(apples == 0 || oranges == 0)

Mistake in PPP on page 173!!

OK

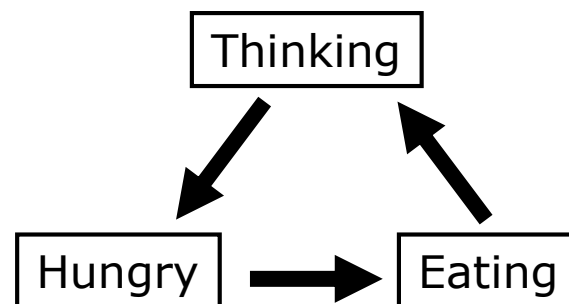
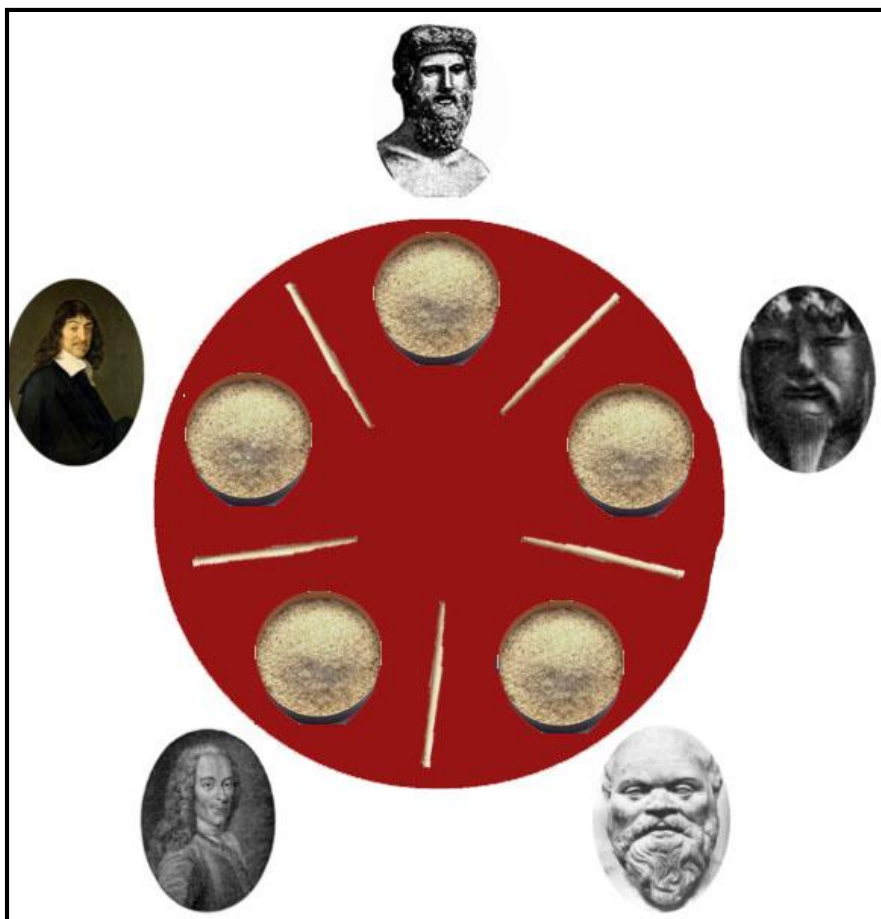


```
pthread_mutex_lock(&lock);
while (apples==0 || oranges==0){
    pthread_cond_wait(&more_apples_or_more
oranges, &lock);
}
// eat apple & orange
pthread_mutex_unlock(&lock);
```

Only 1 cond variable

OK

The Dining Philosophers

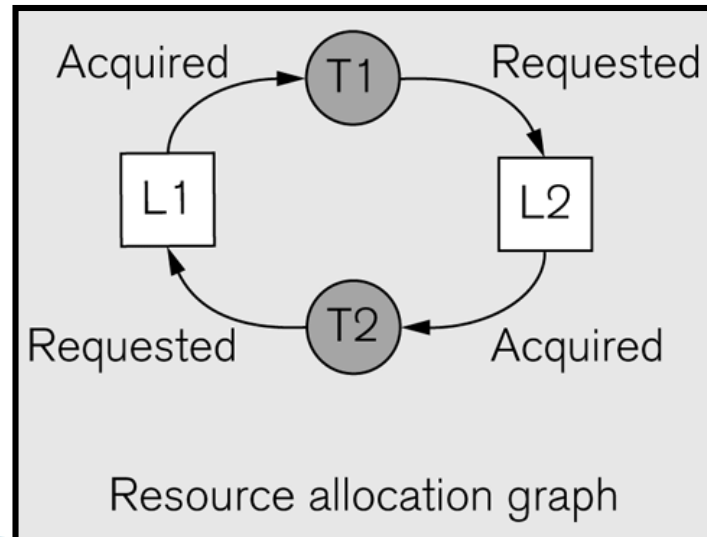


The philosophers are not allowed to speak and there is no arbiter organizing the resources

- ➔ strategy (protocol)?
- ➔ might deadlock or livelock...

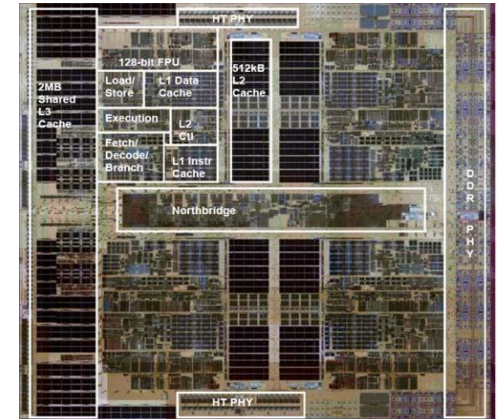
Deadlocks

- ▶ Four conditions
 1. Mutual exclusion
 2. Hold and wait: threads hold some resources and request other
 3. No preemption: resource can only be released by the thread that holds it
 4. Circular wait: cycle in waiting of a thread for a resource of another



Livelocks

- ▶ Similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.
- ▶ *Real-world example*: two people meet in a narrow corridor, each moves aside to let the other pass, but they end up swaying from side to side
- ▶ A risk with algorithms that detect and recover from deadlock.



OpenMP and related

OpenMP Philosophy

- ▶ The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran.
- ▶ Portable, scalable model with a simple and flexible interface for developing parallel applications
- ▶ Augment sequential programs in minor ways to identify code that can be executed in parallel.
 - Simpler to use
 - More restrictive in terms of parallel interactions than Java/POSIX
- ▶ Standardized (Sun, Intel, Fujitsu, IBM, ...)
- ▶ <http://www.openmp.org>

How?

- ▶ Add pragmas to program
 - `#pragma omp <specifications>`
 - The `#pragma` directives offer a way for each compiler to offer machine- and operating system-specific features. If the compiler finds a pragma it does not recognize, it issues a warning, but compilation continues.
- ▶ An OpenMP-compliant compiler will generate appropriate multithreaded code
 - Other compilers simply ignore the pragmas and generate sequential code.

Count 3s example

```
1  int count3s()
2  {
3      int i, count_p;
4      count=0;
5      #pragma omp parallel shared(array, count, length)\
6          private(count_p)
7      {
8          count_p=0;
9          #pragma omp parallel for private(i)
10         for(i=0; i<length; i++)
11         {
12             if(array[i]==3)
13             {
14                 count_p++;
15             }
16         }
17         #pragma omp critical
18         {
19             count+=count_p;
20         }
21     }
22     return count;
23 }
```

parallel for (line 9)

- ▶ The iterations can execute in any order
- ➔ the iterations can execute in parallel.
 - count instead of count_p is wrong!
- ▶ Reduction pragma for computations that combine variables globally

```
count=0;
#pragma omp parallel for reduction(+,count)
for(i=0; i<length; i++)
    count += array[i]==3 ? 1 : 0;
```

Handling data dependencies

```
#pragma omp critical  
{  
    count += count_p;  
}
```

Critical section that
will be protected by
locks

```
#pragma omp atomic  
score += 3
```

Memory update is
noninterruptible

Sections to express task parallelism

```
#pragma omp sections
{
  #pragma omp section
  {
    Task_A();
  }
  #pragma omp section
  {
    Task_B();
  }
  #pragma omp section
  {
    Task_C();
  }
}
```

OpenACC for GPU computing

- ▶ A dialect of OpenACC especially for GPU computing
 - Easier than OpenCL/CUDA
 - The future??
- ▶ Based on OpenHMPP from CAPS enterprise (Bretagne, France)

Matlab: parallel for

- ▶ Parallel computing toolbox provides simple constructs to allow parallel execution
 - Parallel for (when iterations are independent)
 - ...
- ▶ Automatic parallel execution
- ▶ Create pool of computers that will work together
- ▶ Many functions of libraries run in parallel and even (automatically) on GPU!