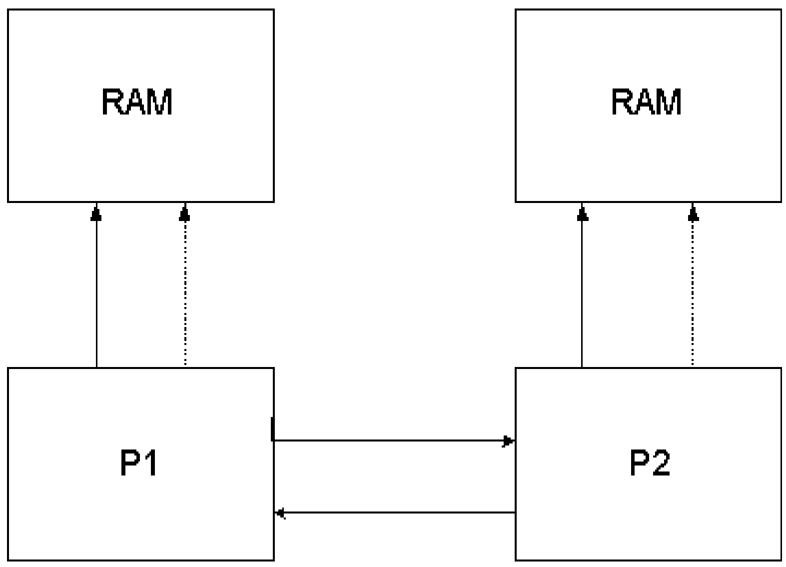
Practical Parallel Programming II

Shared Memory systems

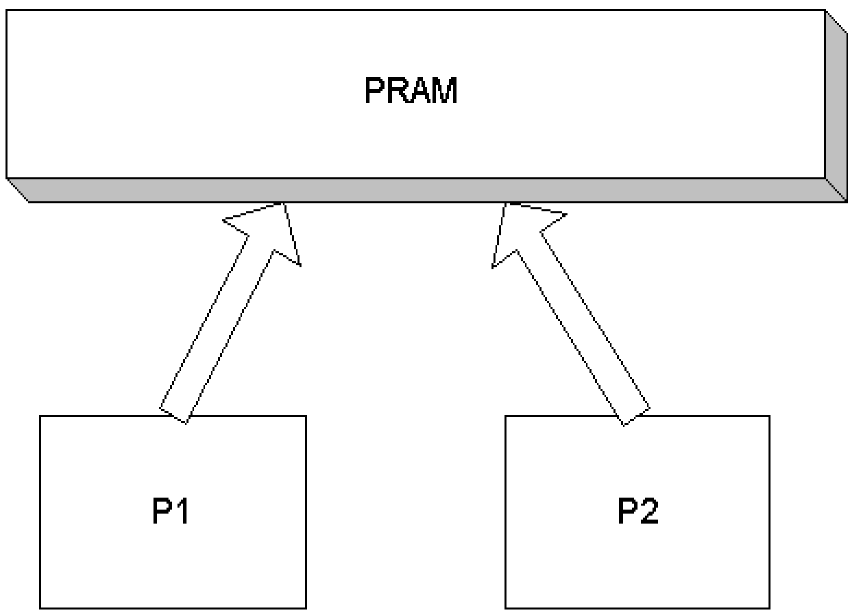
Jan Lemeire

I. Distributed-Memory Architectures



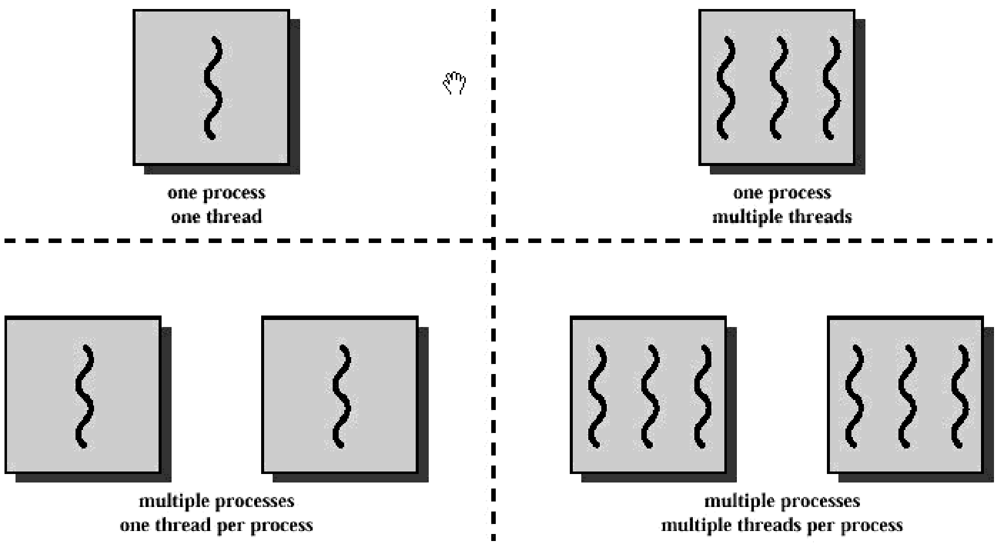
- Each process got his own local memory
- Communication through messages
- Process is in control

II. Shared Address-space Architectures

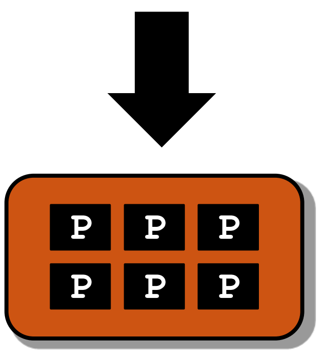


- Example: multiprocessors
- **PRAM**: Paralleled Random Access Memory
  - Idealization: No communication costs
- But, unavoidability: the possibility of *race conditions*

Processes versus Threads



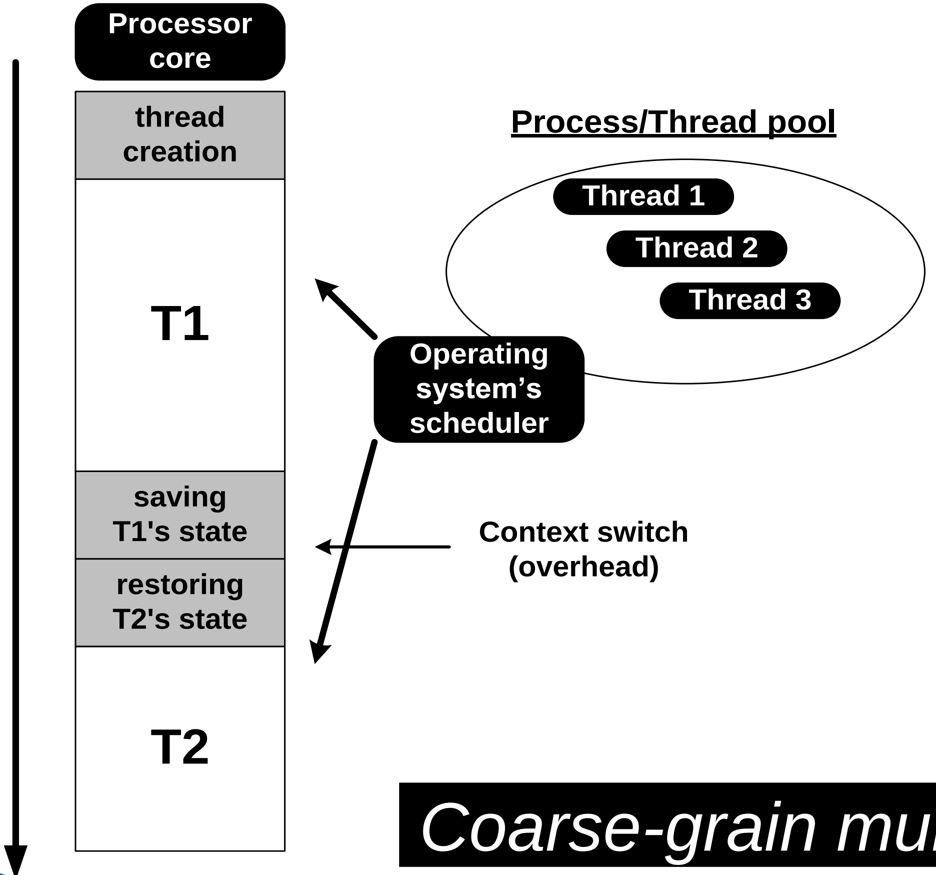
Scheduled by the OS on the available processors



Example: **A file server on a LAN**

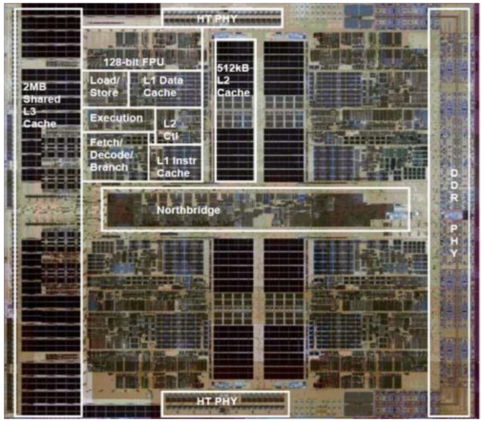
- It needs to handle several file requests over a short period
- Hence, it is more efficient to create (and destroy) a single thread for each request
- Multiple threads can possibly be executed simultaneously on different processors (mapped by Operating System)

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!

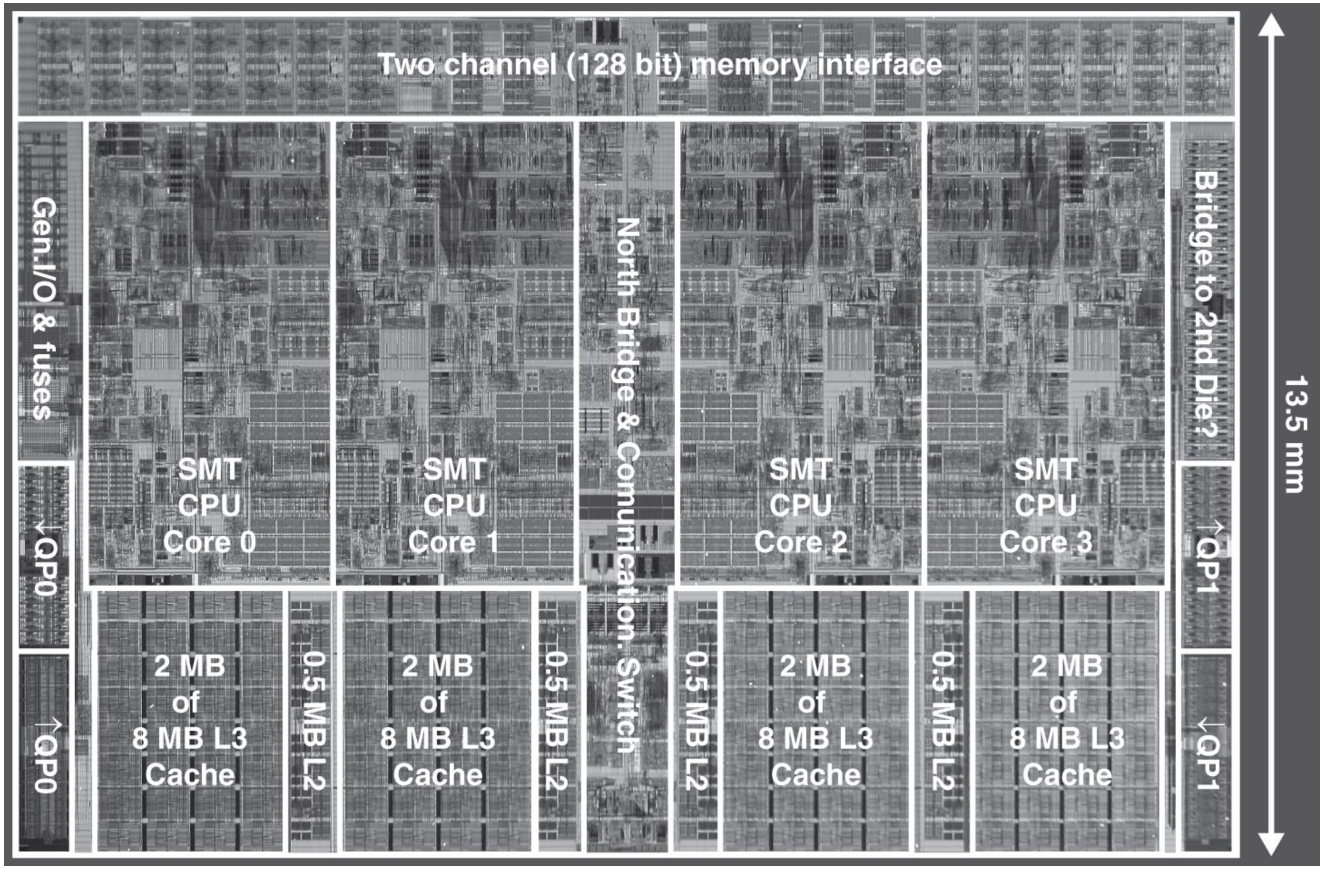
Coarse-grain multithreading



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 (2×) 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

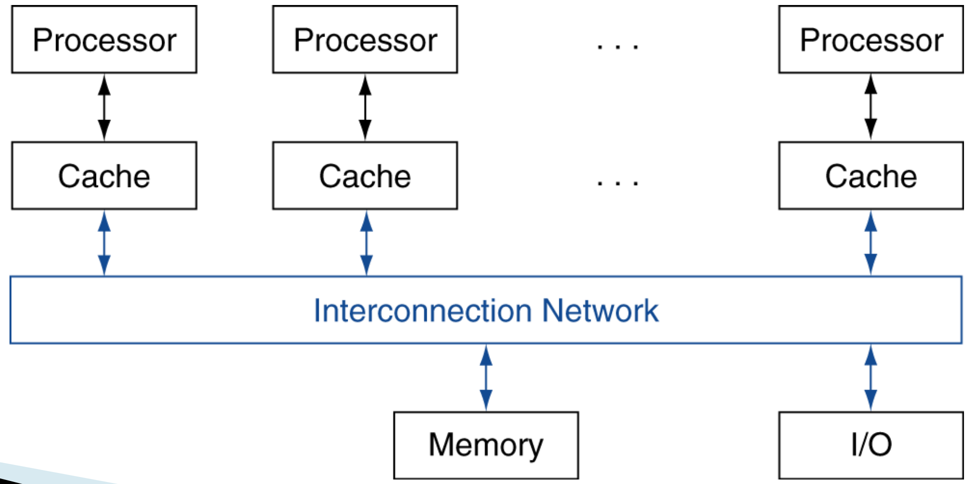
n/a: data not available

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. **Mapping** of 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

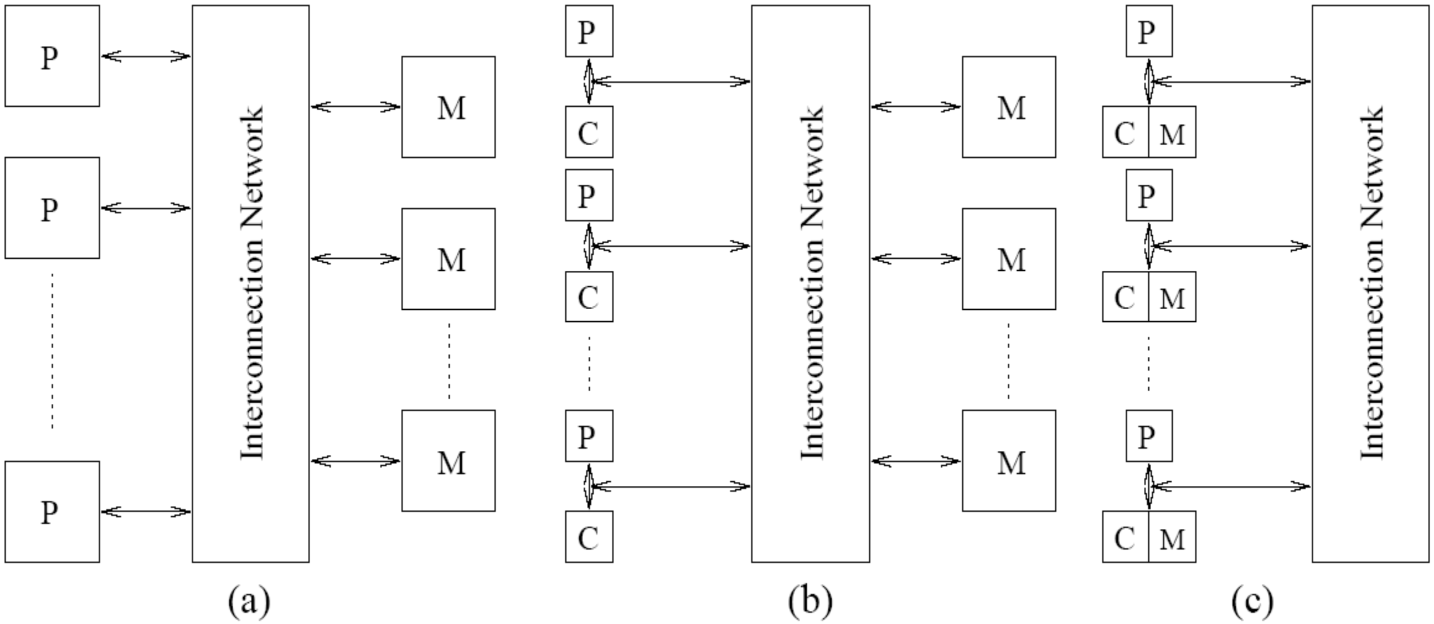
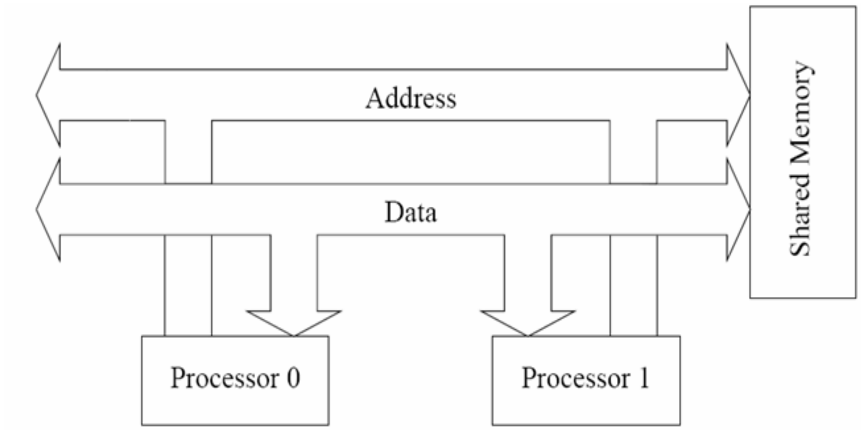
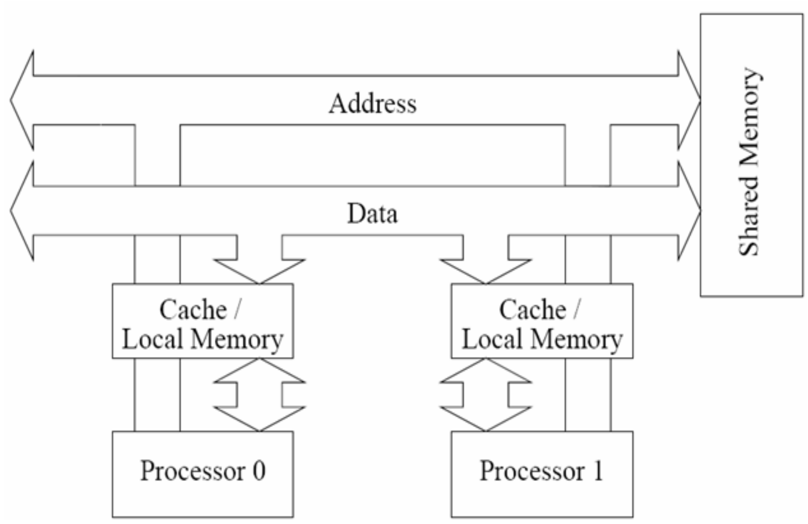


Figure 2.5 Typical shared-address-space architectures: (a) Uniform-memory-access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.

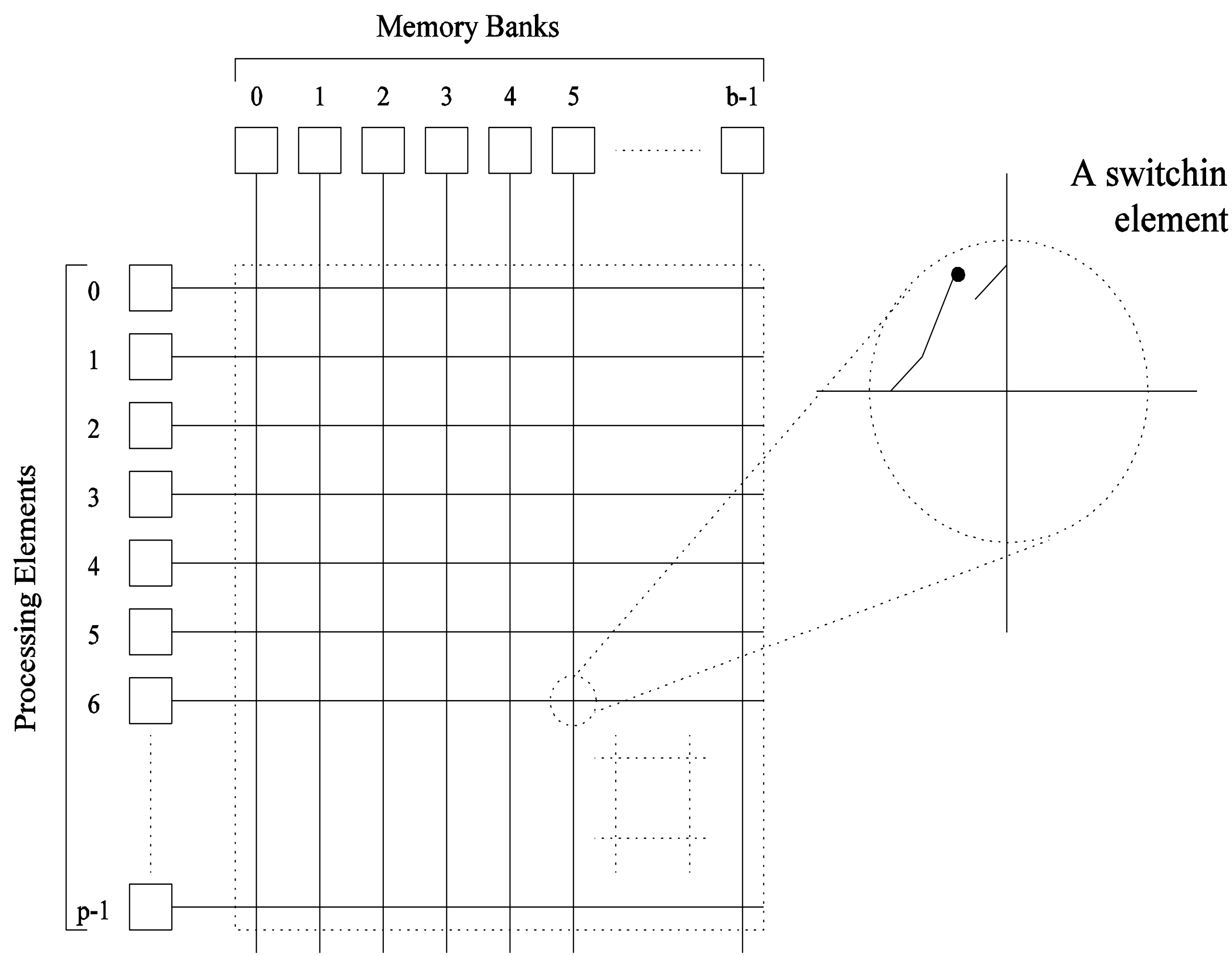
Bus-based Interconnects



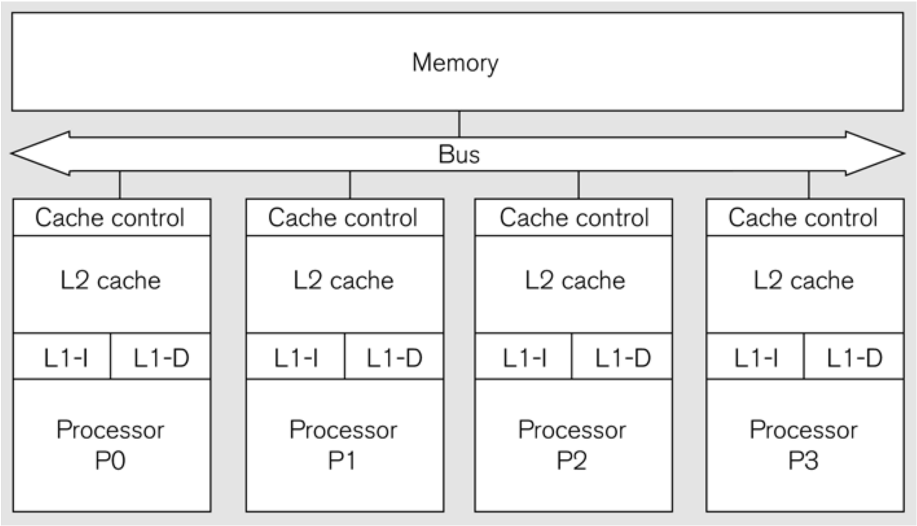


With local memory/cache

Crossbar switches



Symmetric Multiprocessor Architectures (SMPs)



- Cf AMD architecture
- Bus is potential bottleneck

⇒ Number of SMPs is limited

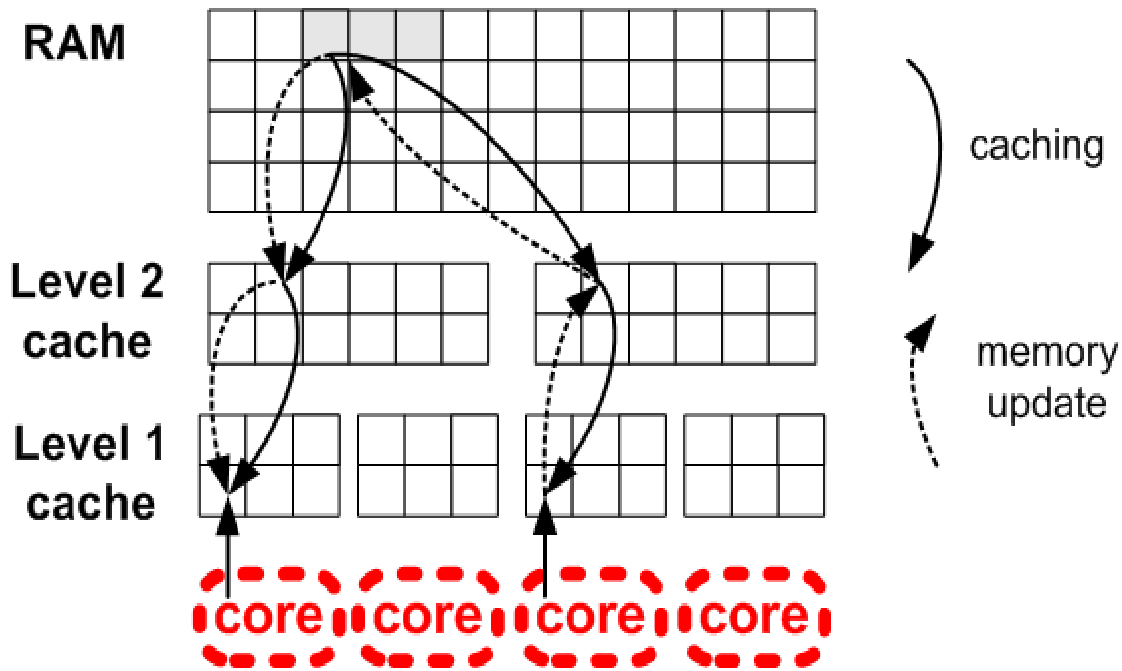
B. PRAM Architectures

- 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.

C. Caching & memory coherence



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

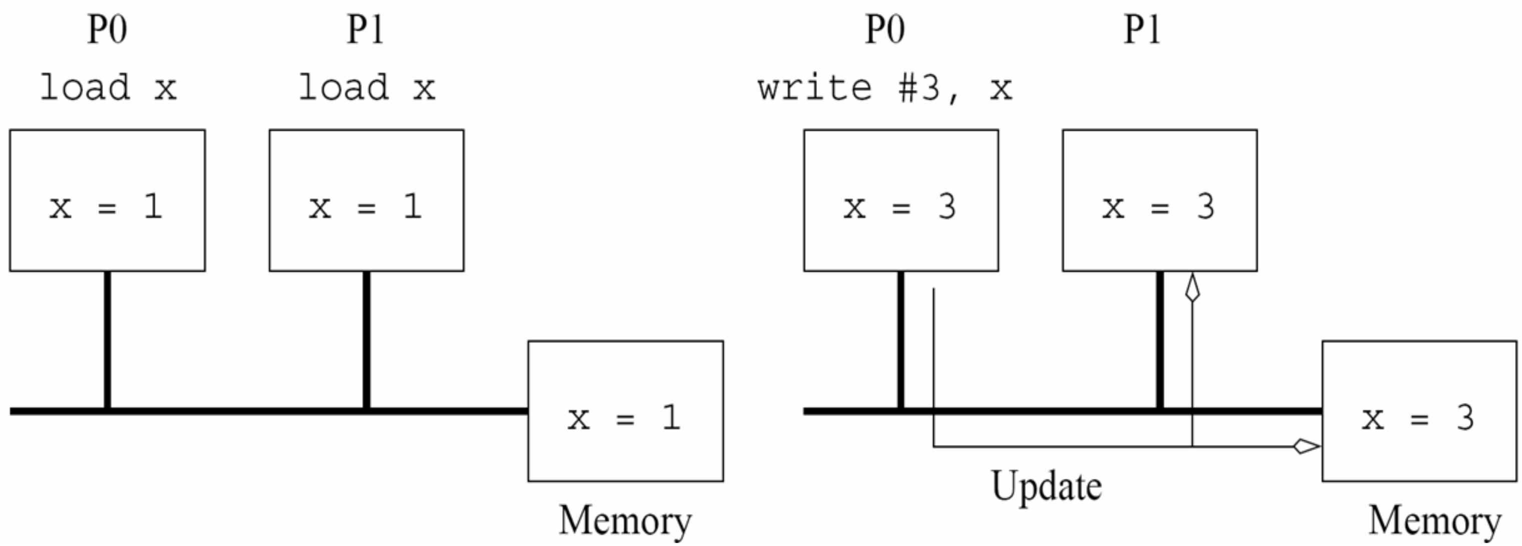
Cache Coherence Problem

- 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

Cache Coherence Mechanisms

Update protocol



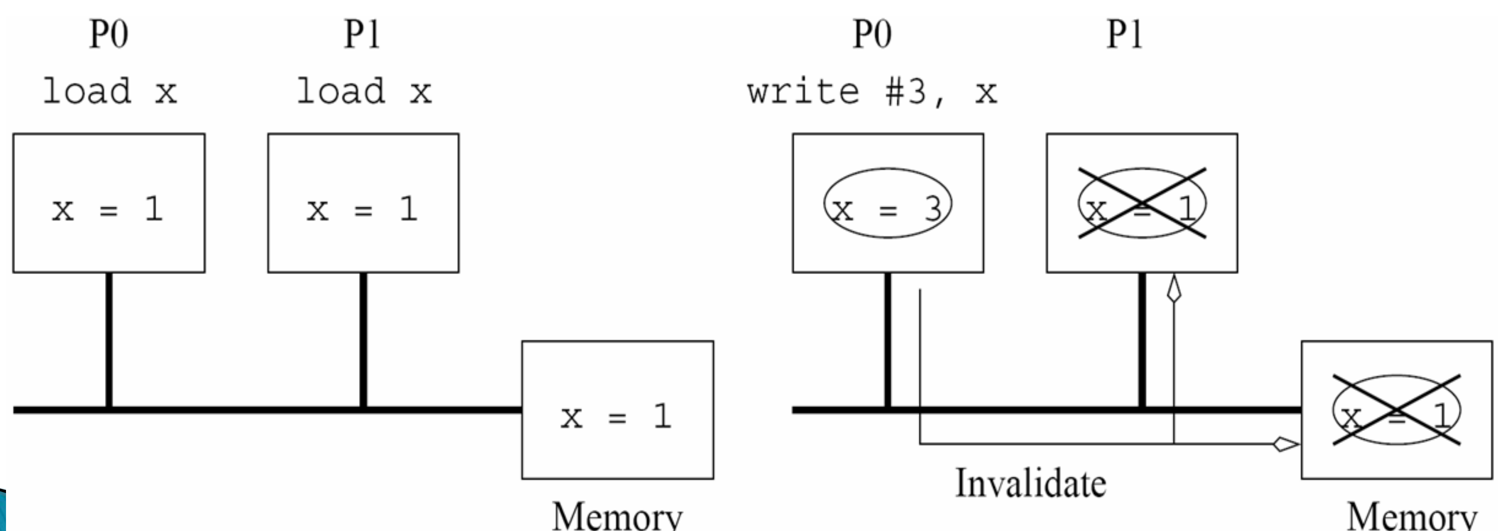
- Excess in updates if variable is only read once in P1
- *False sharing*: processes update different parts of same cache line

➡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 (Hardware!!)

- 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

  $\Rightarrow$ 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

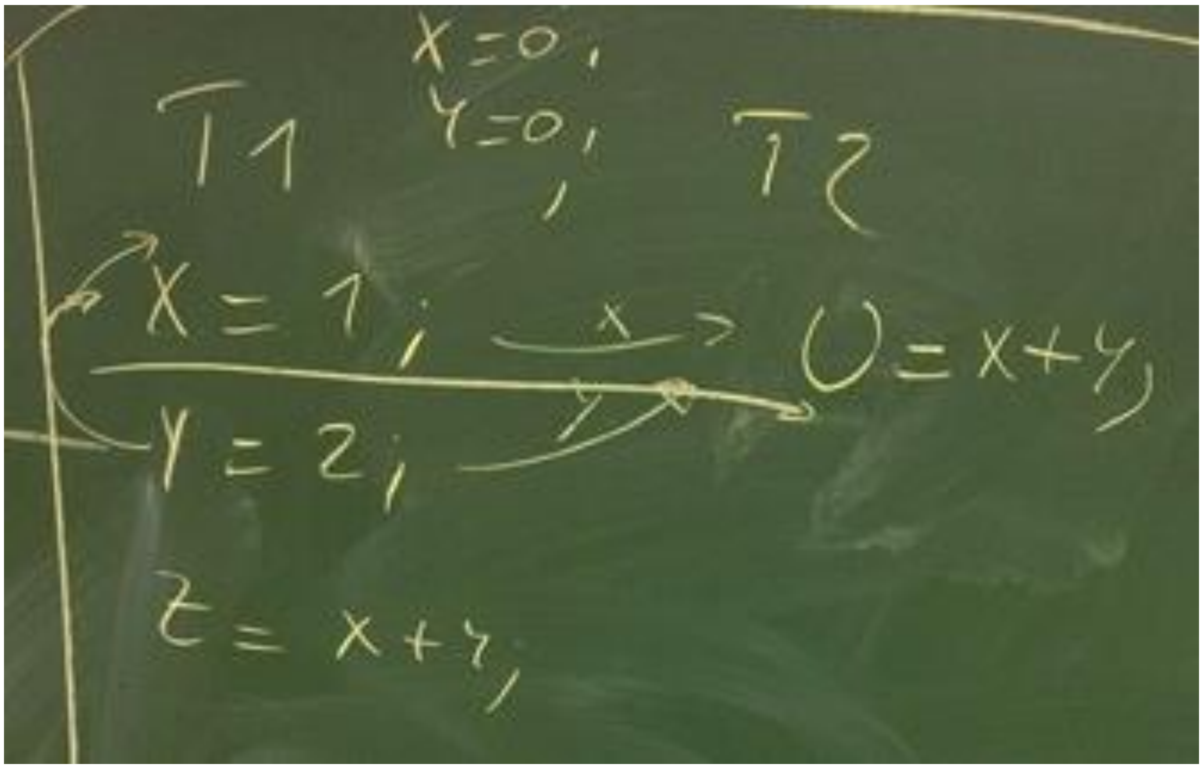
CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		0
CPU B read X	Cache miss for X	1	1	1

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
    $\Rightarrow$ all processors that see new Y also see new X
  - Processors can reorder reads, but not writes

Memory Consistency

- Example



- Order is preserved: U can be 0, 1 or 3, but will never be 2.

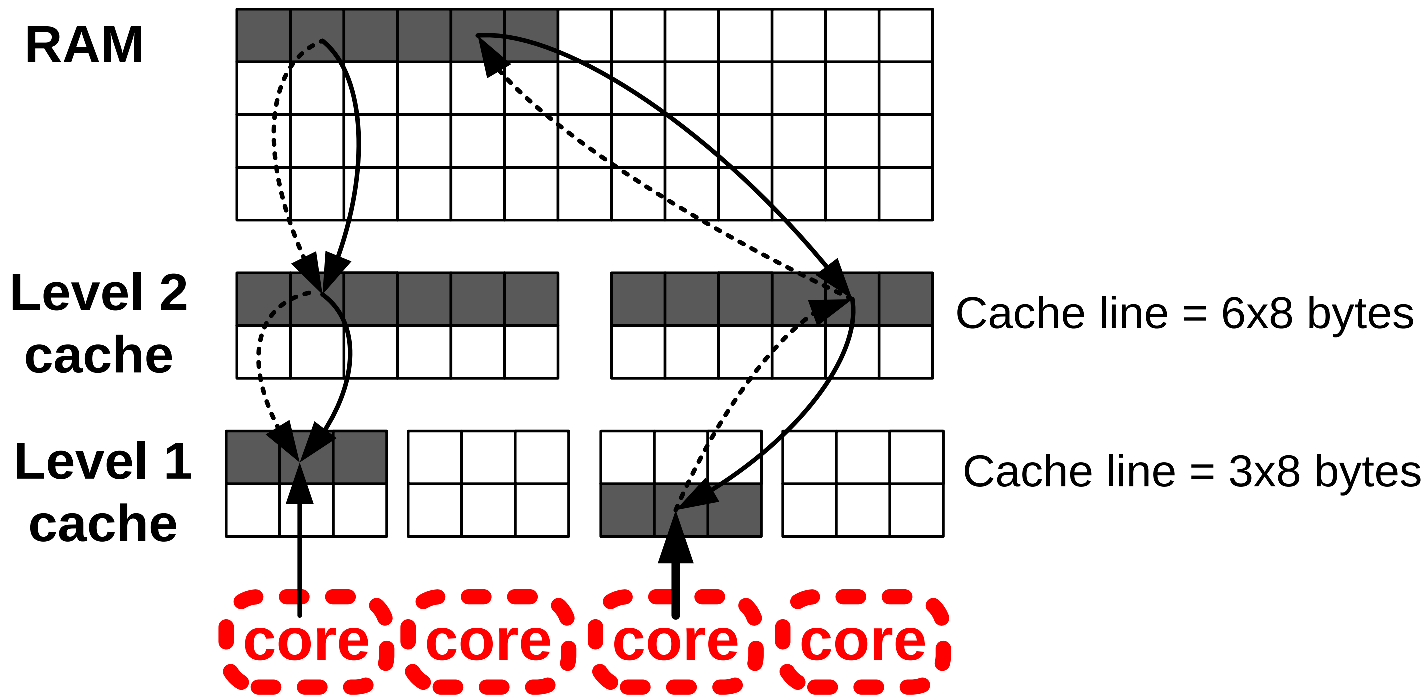
MESI-protocol

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

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*…

D. Mapping of threads on cores.

- **Static mapping**:
  - A thread is dedicated to a specific core on which it is executed until it finishes.
  - Disadvantage: the number of active threads is limited to the number of cores x number of hardware threads
- **Dynamic mapping**:
  - A **scheduler** dynamically assigns threads to the available cores. Each core gets 1 thread (more if hardware threads) at a time. The scheduler can interrupt the thread and replace it with another one. Processor switches between threads.
  - The scheduler is part of the OS.
  - Note that the same happens with the active processes on the system.

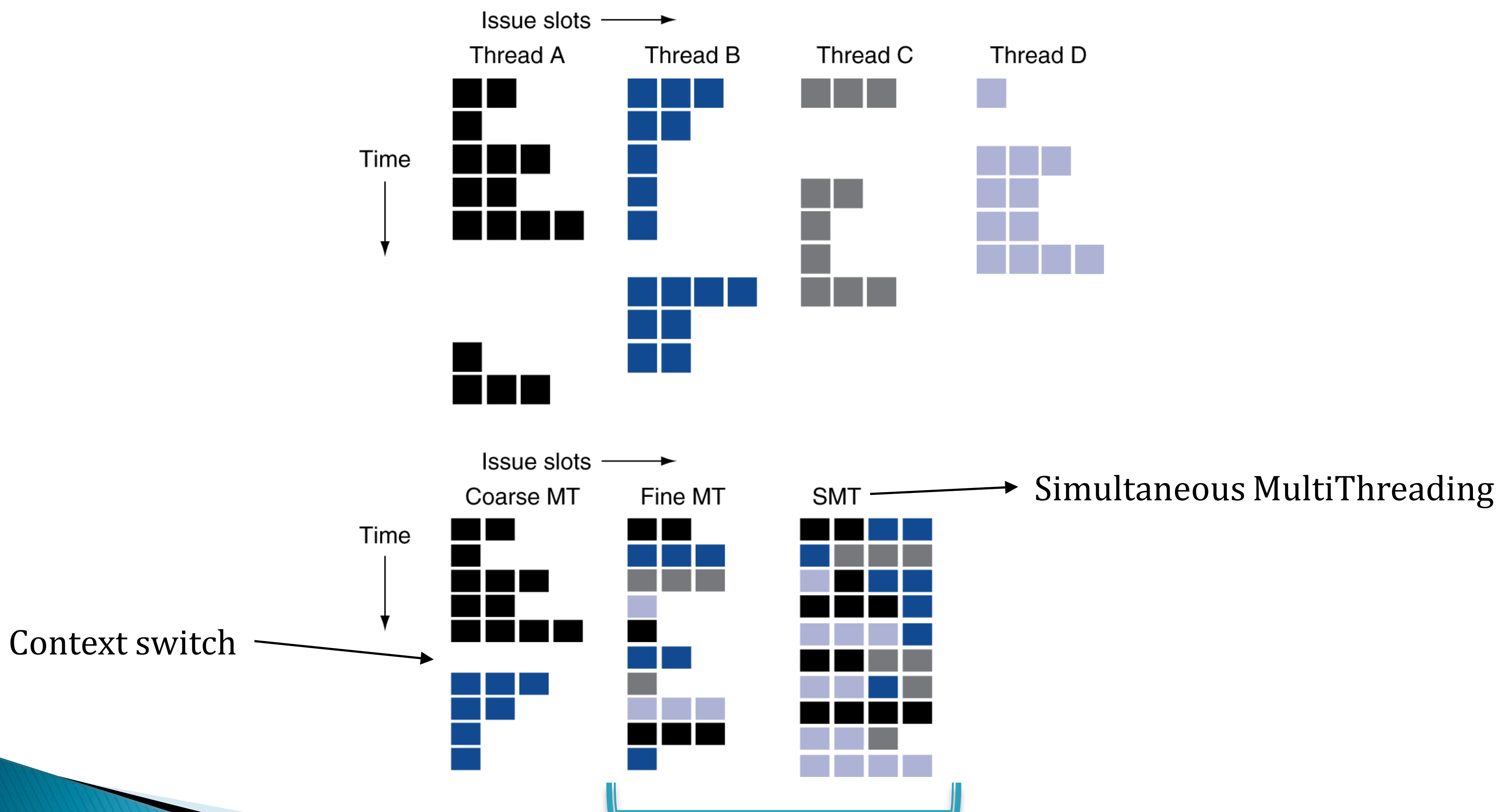
Software versus hardware threads

- **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 hardware threads running simultaneously without overhead!

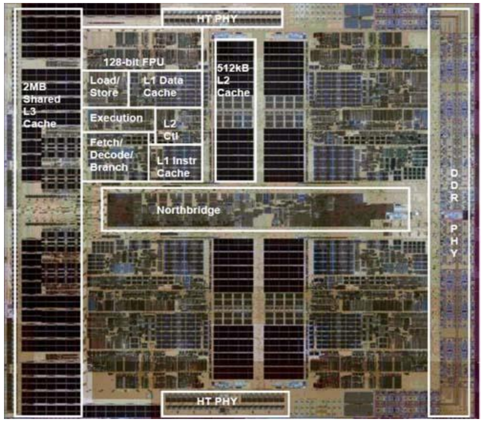
Multi-Threading (MT) possibilities



Fine-grained parallelism: *see chapter on GPUs*

E. Thread Synchronization

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



2. Multicore usage

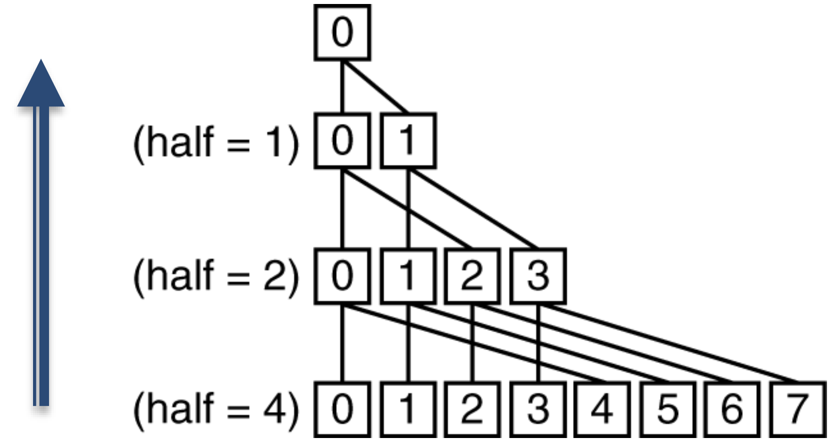
Example: Sum Reduction

- Sum 1000000 numbers with 8 processors
  - Each processor has ID: $0 \leq Pi \leq 8$
  - Array **sum** with 8 elements
  - Partition 125000 numbers per processor
  - Initial local summation on each processor:

```
sum[Pi] = 0;
for (i = 125000*Pi;i < 125000*(Pn+1); i++)
      sum[Pi] += 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

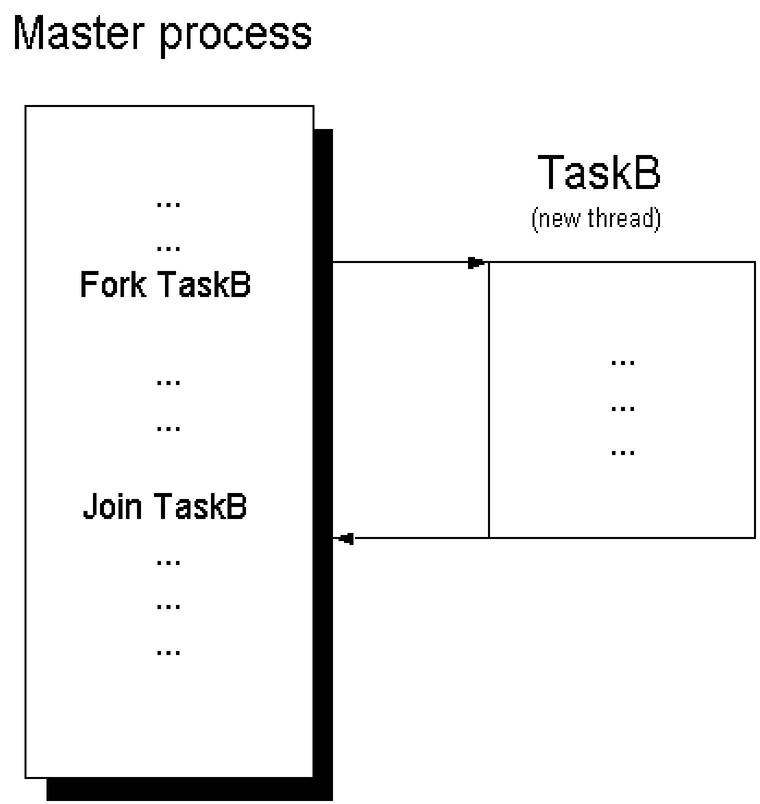
Example: Sum Reduction




```
half = 8;
repeat {
  barrier_synchronization();
  half = half/2; /* dividing line on who sums */
  if (Pi < half)
     sum[Pi] = sum[Pi] + sum[Pi+half];
} until (half == 1);
```

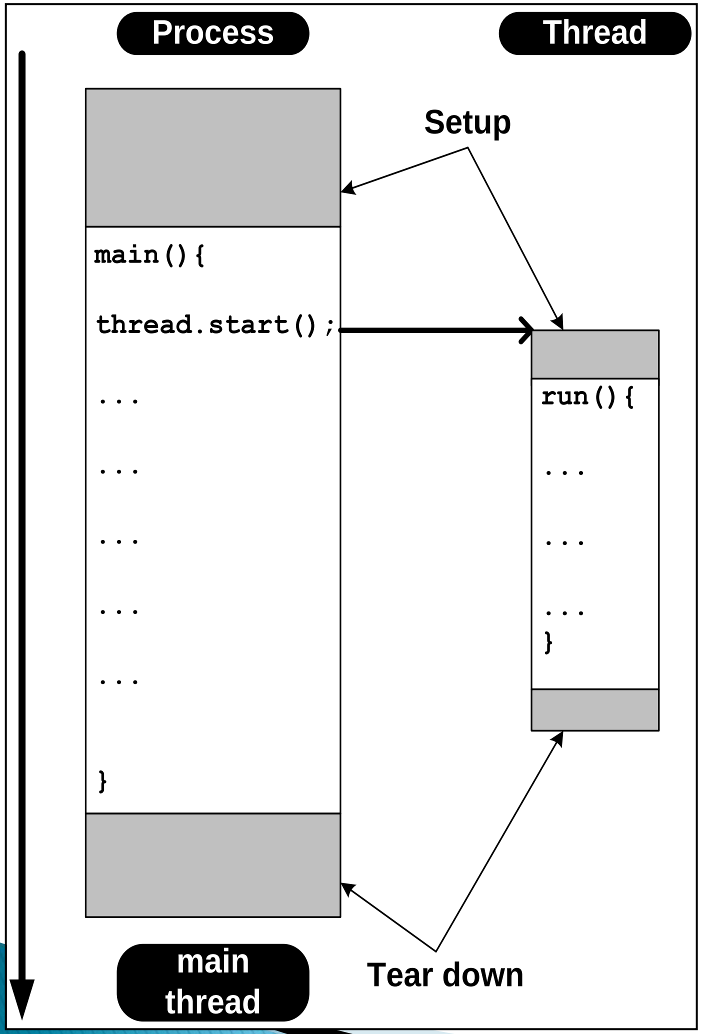
Multi-threading primitives

- Fork & join



Parallelism: 2 programs running at the same time

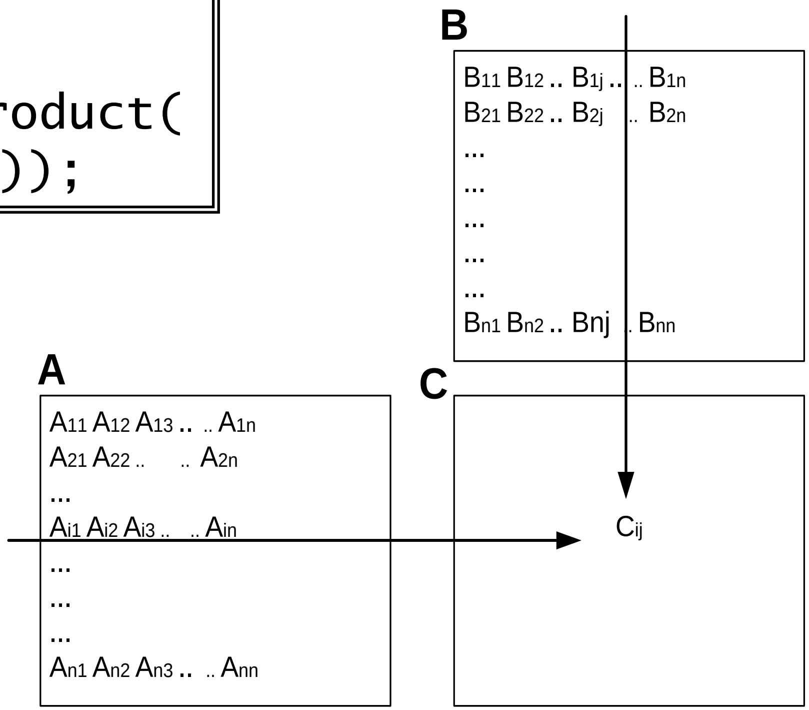
Thread creation



- 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

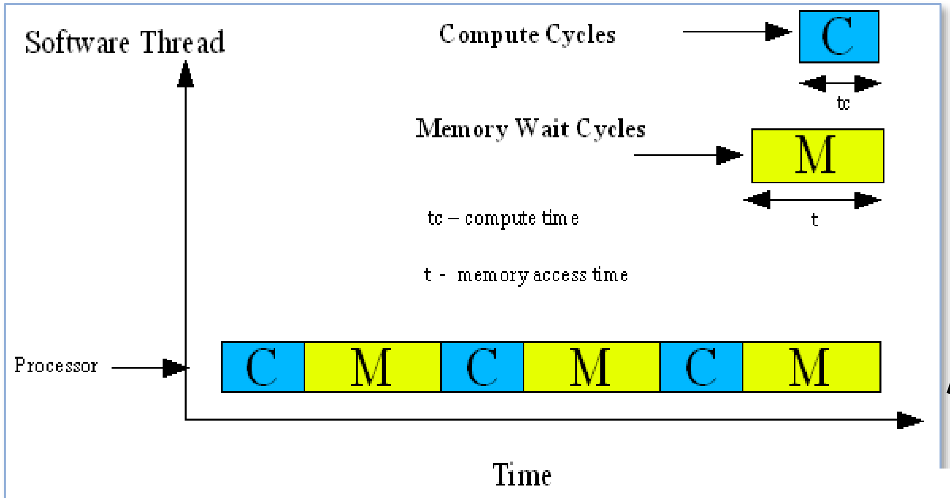
= a lot of overhead

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

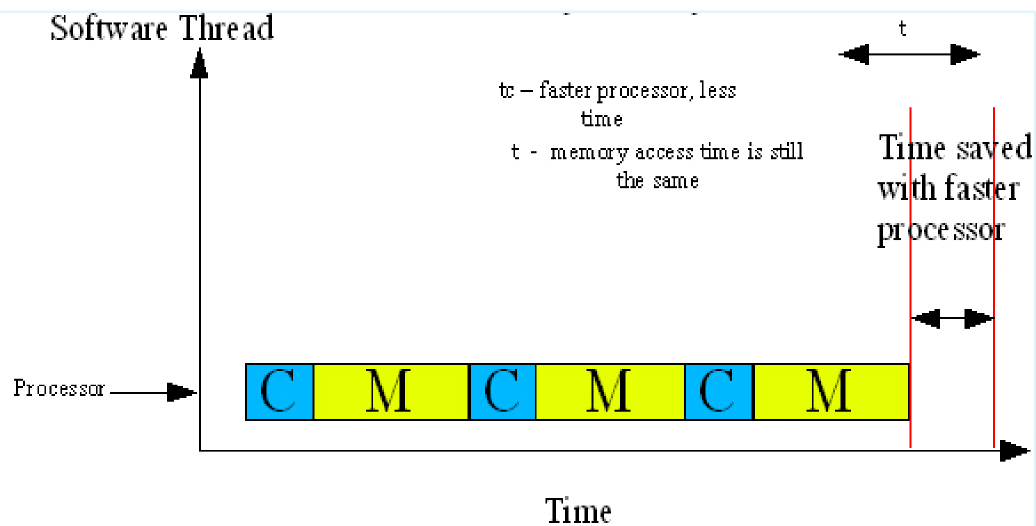
Why Threads?

- 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)

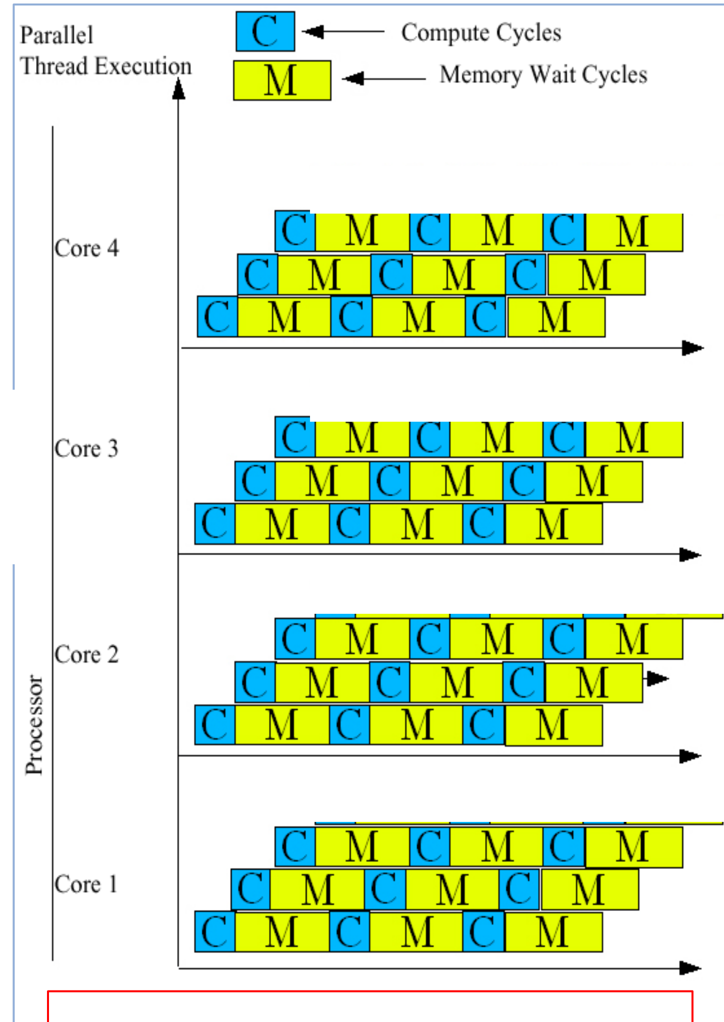
Latency Hiding



Faster CPU



More threads



4 cores: x4

Latency hiding: x3

Multi-threading without speedup

- Webserver: a thread for each client
  - Multi-threading for convenience
  - = 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%!)

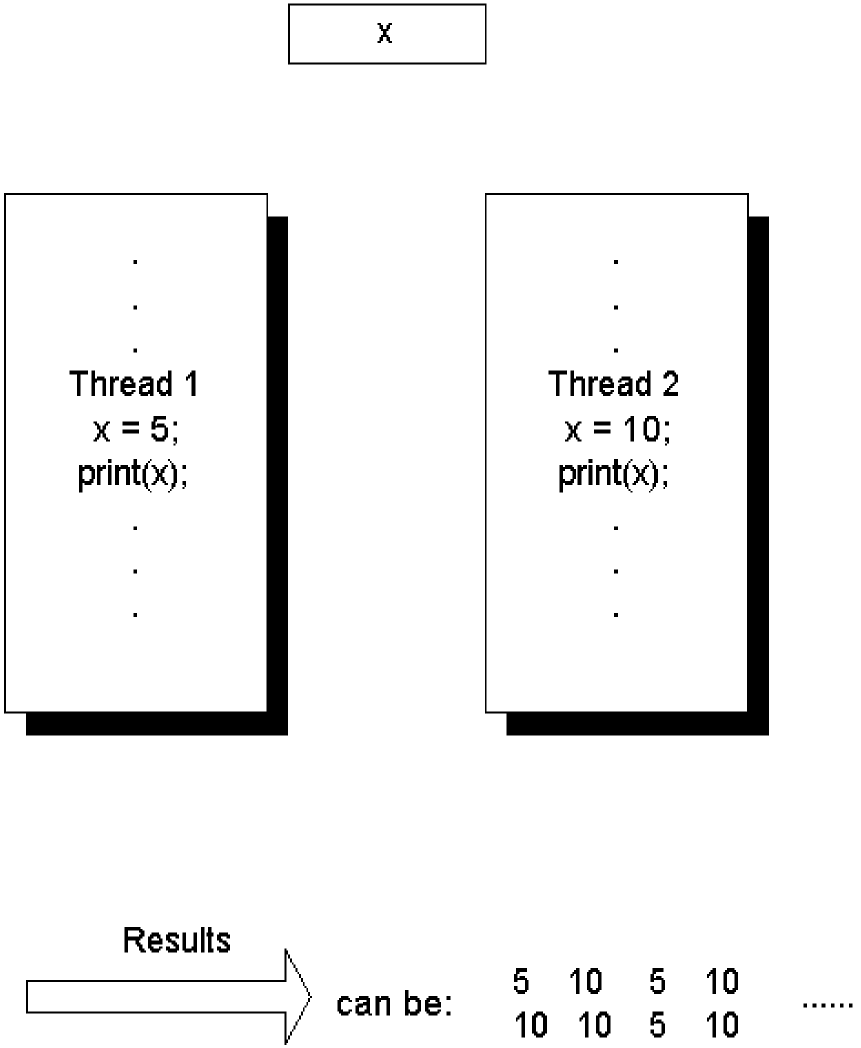
LINK 9

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

- 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.



➔ 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

shared variable

Thread 1

```
best_cost = 100
```

Thread 2

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

```
best_cost = 75
```

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

```
best_cost = 50
```

OK

Synchronization OK

shared variable

Thread 1

`best_cost = 100`

Thread 2

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

```
my_cost = 75;
```

`best_cost = 50`

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

`best_cost = 50`

OK

Synchronization problem!!

<u>*shared variable*</u>

Thread 1 `best_cost = 100` Thread 2

```
my_cost = 75;                              my_cost = 50;
if (my_cost < best_cost)
                                           if (my_cost < best_cost)
                                               best_cost = my_cost;
                        best_cost = 50
    best_cost = my_cost;

                        best_cost = 75
```

NOK

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

Solution: locking of critical sections

<u>*shared variables*</u>

Thread 1

Thread 2

```
best_cost = 100
pthread_mutex_t lock;
```

Thread 1:

```
my_cost = 75;
pthread_mutex_lock(&lock)
if (my_cost < best_cost)
    best_cost = my_cost;
pthread_mutex_unlock(&lock)
```

```
best_cost = 75
```

Thread 2:

```
my_cost = 50;

pthread_mutex_lock(&lock)

if (my_cost < best_cost)
    best_cost = my_cost;
pthread_mutex_unlock(&lock)
```

```
best_cost = 50
```

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.

Multithreaded Counting 3s (C++)

parameters: array arr of size n, NBR_THREADS

```
void count_function(int threadID, int n, int* arr, int* count) {
   for (int i = 0; i < n; ++i)
      if (arr[i] == 3)
         (*count)++;
}
```

this program is still faulty
we will solve it in the exercises

```
vector<thread*> threads; // vector of pointers to threads
int ELEMENTS_PER_THREAD = n / NBR_THREADS, count = 0;

// *** STARTING THE THREADS
for (int t = 0; t < NBR_THREADS; t++)
    // pass the function to be executed and all the necessary parameters
    threads.push_back(new thread(count_function, t,
ELEMENTS_PER_THREAD, arr + t * ELEMENTS_PER_THREAD, &count));

// *** waiting for all threads to finish
for (int t = 0; t < threads.size(); t++) {
     threads[t]->join();
     delete threads[t];
}
```

Counting 3s: experiments

On a dual core processor

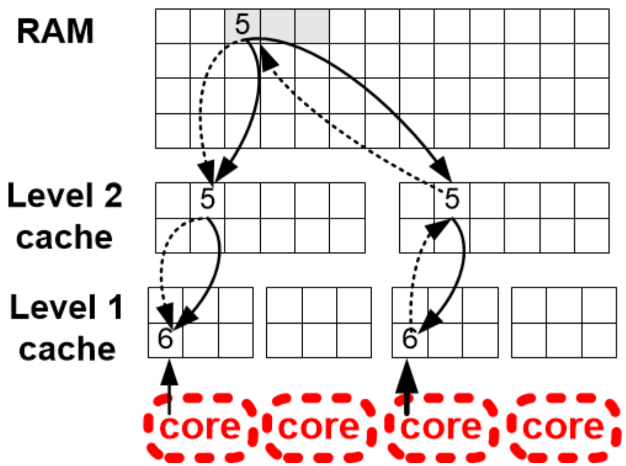
```
Counting 3s in an array of 1000 elements and 4 threads:
 * Seq   : counted 100 3s in 234us
 * Par 1: counted 100 3s in 3ms 615us


Counting 3s in an array of 40000000 elements and 4 threads:
 * Seq   : counted 4000894 3s in 147ms
 * Par 1: counted 3371515 3s in 109ms
```

this program is faulty: parallel result is not the same

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 100% software solution for this: Peterson Algorithm (but not efficient)

Mutex Lock Implementation

= optimized naïve version

```
boolean access_x=true;

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

Shared variable & initialization

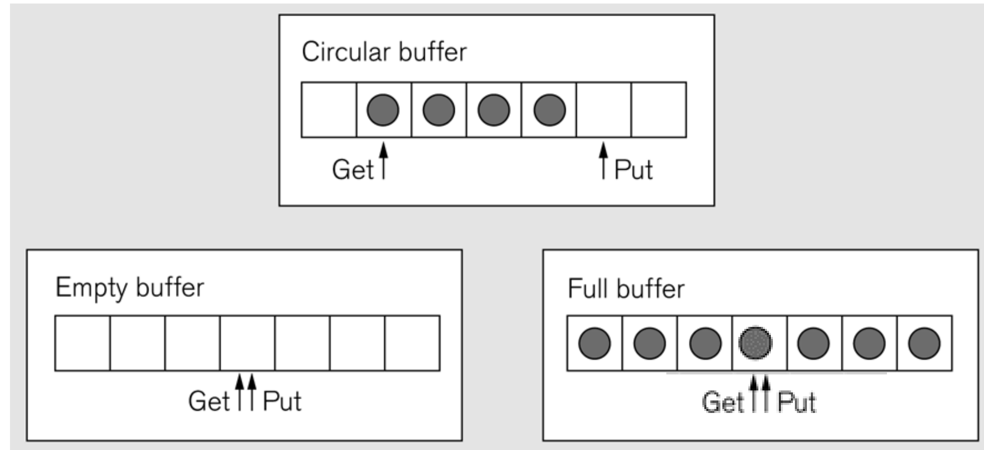
Locking the lock:
- as an atomic operation
- only 1 thread can acquire it
- context switch if loop takes a long time

Unlocking and activating threads waiting at the lock

Critical sections trigger cache coherence

- System will not perform cache coherence all the time
  - Too costly
- Critical sections indicate shared data
  - Cache coherence is ensured when threads access critical section

Producers–Consumers Scenario



Producer Threads	Consumer Threads

`...`
`Produce thing`
`Put in buffer`
`...`
	`...`
	`Get from buffer`
	`Consume thing`
	`...`

1. Thread synchronization

```
...
Produce thing
If buffer=full
   wait
Put in buffer
Signal non-
emptiness
...
```

```
…
If buffer=empty
  wait
Get from buffer
Consume thing
Signal non-
fullness
...
```

Multi-threading primitives

Should minimally allow the following:

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

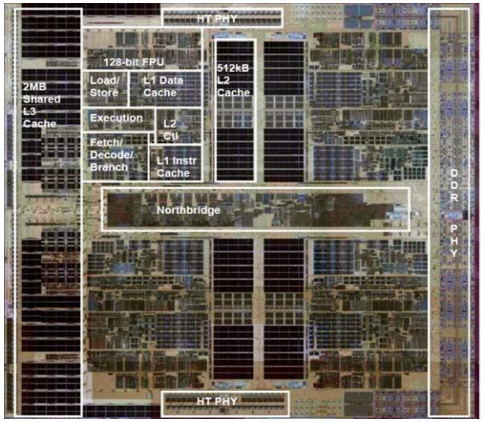
With *primitives* we mean the minimal set of mechanisms (e.g. functions or language constructs) you need to write any multi-threaded program.

Pthreads (C, C++, …) & Java

	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`

Intermezzo: the Operating System

- OS is also a software process
- Context switch necessary to activate it
  - It is not a ‘big brother’ overseeing what’s happening in the processor
  - Takes time! Has to be minimized



3. POSIX Threads

VRIJE UNIVERSITEIT BRUSSEL

KUMAR Chapter 7

PPP Chapter 6

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.

pthreads: 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`.

VRIJE UNIVERSITEIT BRUSSEL

Counting 3s Example

KUMAR

```
#include <pthread.h>
#define NBR_THREADS 16
void count_function(int threadID, int n, int* arr, int* count);

int counting3s(int* totalArray, int arraySize) {
  int count = 0;
  int ELEMENTS_PER_THREAD = arraySize / NBR_THREADS,
   pthread_t p_threads[NBR_THREADS];
   for (i=0; i< NBR_THREADS; i++) {
     pthread_create(&p_threads[i], NULL, count_function, i,
       ELEMENTS_PER_THREAD, totalArray + i * ELEMENTS_PER_THREAD, &count);
   }
   for (i=0; i< NBR_THREADS; i++) {
     pthread_join(p_threads[i], NULL);
   }
   return count;
}
```

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);`

VRIJE UNIVERSITEIT BRUSSEL

Lock critical sections

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

```
pthread_mutex_t costLock;
...
main() {
   ....
   pthread_mutex_init(&costLock, NULL);
   ....
}
void *find_min() {
   ....
   pthread_mutex_lock(&costLock);
   if (my_cost < best_cost)
        best_cost = my_cost;
   pthread_mutex_unlock(&costLock); /* and unlock the mutex */
}
```

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_init(pthread_cond_t *cond,
  const pthread_condattr_t *attr);


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_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.

VRIJE UNIVERSITEIT BRUSSEL

PPP 170

```
pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t nonempty=PTHREAD_COND_INITIALIZER;
pthread_cond_t nonfull=PTHREAD_COND_INITIALIZER;
Item buffer[SIZE];
int put=0;                           // Buff index for next insert
int get=0;                           // Buff index for next remove

void insert(Item x)                         // Producer thread
{
  pthread_mutex_lock(&lock);
  while((put>get&&(put-get)==SIZE-1)||      // While buffer is
         (put<get&&((get-put)==1)))          // full
  {
    pthread_cond_wait(&nonfull, &lock);
  }
  buffer[put]=x;
  put=(put+1)%SIZE;
  pthread_cond_signal(&nonempty);
  pthread_mutex_unlock(&lock);
}

Item remove()                               // Consumer thread
{
  Item x;
  pthread_mutex_lock(&lock);
  while(put==get)                           // While buffer is empty
  {
    pthread_cond_wait(&nonempty, &lock);
  }
  x=buffer[get];
  get=(get+1)%SIZE;
  pthread_cond_signal(&nonfull);
  pthread_mutex_unlock(&lock);
  return x;
}
```

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

Why always a lock with condition variables?

1. The condition and cond_wait form a critical section (lines 10-14 & lines 26-29)

```
while (bufferIsFull()){

  pthread_cond_wait(&nonfull, &lock);
}
```

if consumer refills buffer here, producer is waiting in vain

- The update of the buffer and change of pointer are also a critical section (lines 16-17 & lines 30-31)
- When the thread goes into a wait, the lock that it has at that moment will be released by the cond_wait
- When the waiting thread is activated again, it acquires the lock again (after the notifying thread has released it).

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.

VRIJE UNIVERSITEIT BRUSSEL

Attributes Objects for Mutexes

- Initialize the attrributes 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.

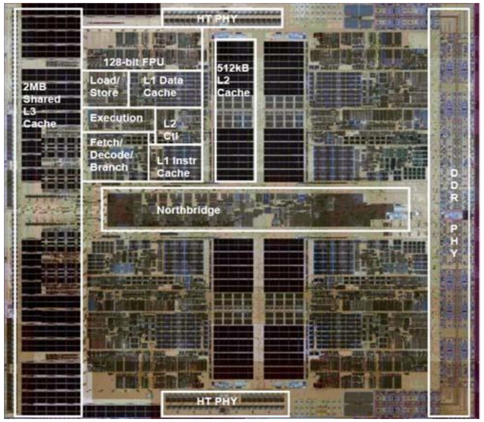
```
pthread_mutexattr_settype_np (
pthread_mutexattr_t *attr,
int type);
```

- 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



4. C++ 11 Multithreading

C++11 = pThreads made easier

- See links on website: practica -> documentation
  - The 3 primitives
  - Three Different ways to Create Threads
  - How to pass arguments to threads
  - **How to return a value from the thread function:**
    - **Pass a result variable by reference**
    - *or* **make it a attribute of your function object**
    - More advanced solutions
- Advanced concepts: see book PPCP

Condition Variables

```
while (!stop_waiting()) {
    cv.wait(lock);
}
```

- Can be written as:

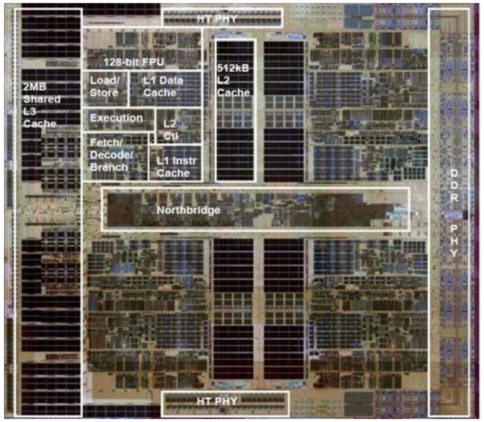
```
cv.wait(lock, []{ return stop_waiting(); });
```

- Second argument is a predicate function, written as a lambda function
- Still another thread has to wake up the waiting thread (is not automatically as soon as the predicate has become true).

C++11 Atomic Objects

= objects that provide the thread-safety and thread-synchronization internally.

- You don't have to worry about locking etc
- Example:
  - atomic<int> ctr=0;
  - Ctr is an integer on which all operations happen atomically.
  - E.g. ctr++; will ensure the read-increment-write critical section happens within a locked section.
- For basic types it is faster than using mutexes!
- For big types, mutexes are used.



5. Thread Safety

Adaptable range-object

- Object specifies a range with a lower and upper attribute.
  - Invariant (should always be true): lower <= upper
  - If multiple threads can modify lower or upper, make thread-safe! Check the invariant.

```
int lower, upper; // shared variables

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;
}
```

Still a critical section present!
We solve this in the exercises

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!

PPP 173-174

```
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);
```

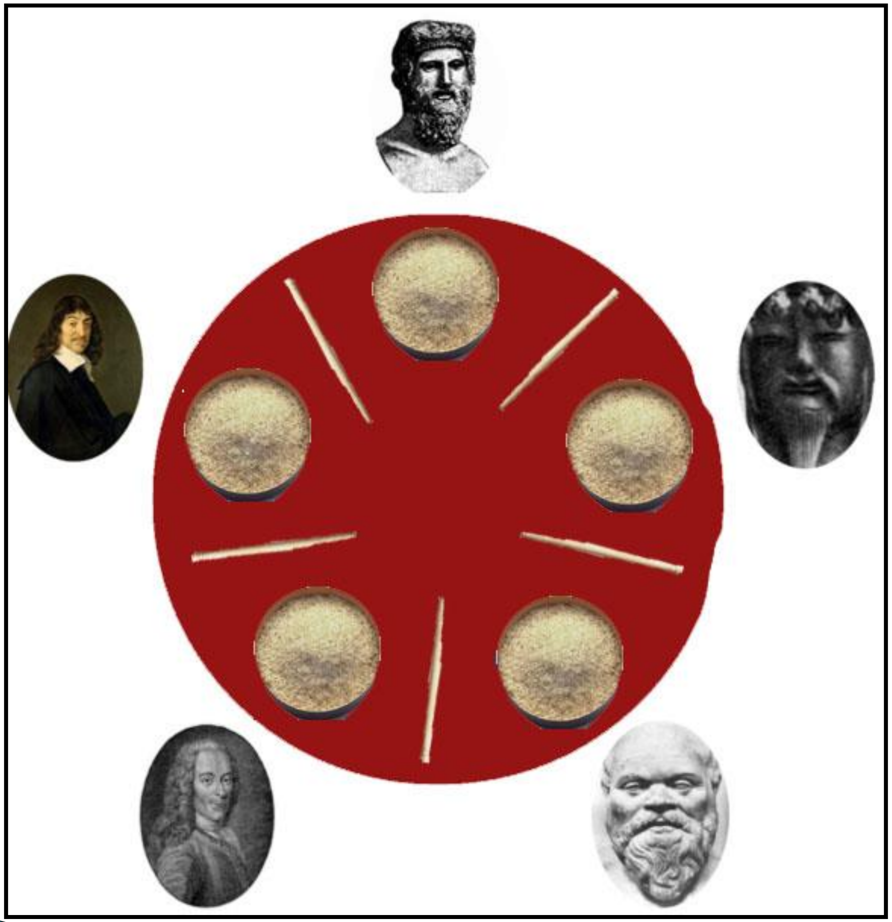
Only 1 cond variable

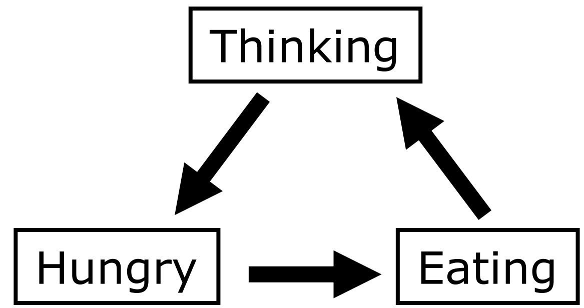
OK

VRIJE UNIVERSITEIT BRUSSEL

LINK 8

The Dining Philosophers



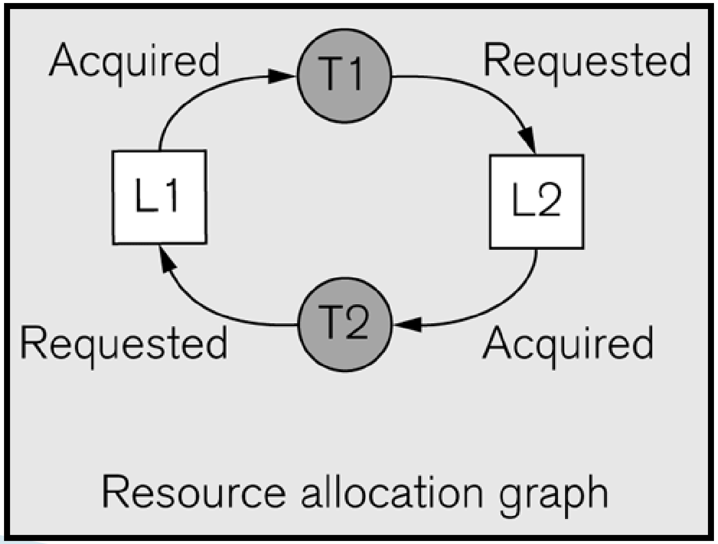


The philosophers are not allowed to speak and there is no arbiter organizing the resources

- ➡ strategy (protocol)?
- ➡ might deadlock or livelock…

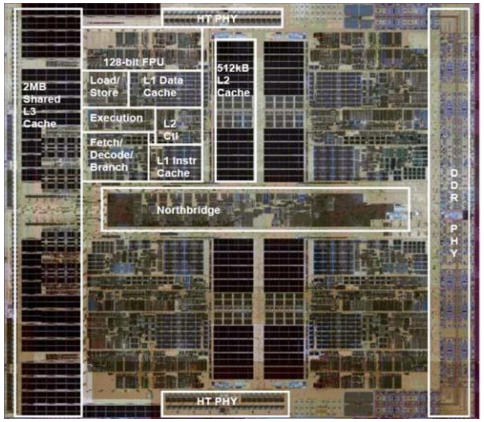
Deadlocks

- 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.



6. OpenMP and related

PPCP 167

PPP 207

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.

OpenMP Hello World

```
#pragma omp parallel
  {
      int i = omp_get_thread_num();
      int n = omp_get_num_threads();
      cout << "Hello world from thread " << i << " of "
  << n << " threads " << endl;
  }
```

- Default number of threads: number of logical cores
- Overwrite with

```
#pragma omp parallel num_threads(8)
```

OpenMP parallel for

```
#pragma omp parallel for
      for (int i = 0; i < n; i++)
            z[i] = x[i] + y[i];
```

- Embarrasingly parallel: all iterations should be independent
  - you have to guarantee that there are no race conditions!
  - Number of loops remains constant
  - Other constraints: see PCPP p. 170
- OpenMP executes this multi-threaded
- Note:

```
#pragma omp parallel for
```

  - Stands for

```
#pragma omp parallel
 {
      #pragma omp for
      for(…){ … }
 }
```

You can add more for-loops here
See PCPP 172-173

VRIJE UNIVERSITEIT BRUSSEL

PPCP 185

PPP 210

OpenMP reduction

- Reduction pragma for computations that combine variables globally

```
accum =0;
#pragma omp parallel for reduction(+,accum)
for(i=0; i<length; i++)
 accum += array[i];
```

  - Reduction operators: +,-,*, max,min, bitwise and logical operations
  - Counting 3s:

PPP 210

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

VRIJE UNIVERSITEIT BRUSSEL

PPCP 173-176

Shared vs private variables

- OpenMP should decide whether variables have to be shared between threads (possibility of race conditions!) or can be considered local to the thread
- Shared = default
  - You can emphasize this with `shared(…)`
- Indicate local variables with `private(…)`
  - See example of counting 3s on next slide

VRIJE UNIVERSITEIT BRUSSEL

PPP 209

Count 3s example with parallel for

```
int count3s()
{
  int i, count_p;
  count=0;
  #pragma omp parallel shared(array, count, length)\
    private(count_p)
  {
    count_p=0;
    #pragma omp parallel for private(i)
    for(i=0; i<length; i++)
    {
      if(array[i]==3)
      {
        count_p++;
      }
    }
    #pragma omp critical
    {
      count+=count_p;
    }
  }
  return count;
}
```

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

VRIJE UNIVERSITEIT BRUSSEL

Sections to express task parallelism

PPCP 209

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

Parallel hint

- Give hints to the auto-parallelizer
- https://docs.microsoft.com/en-us/cpp/parallel/auto-parallelization-and-auto-vectorization?redirectedfrom=MSDN&view=vs-2019

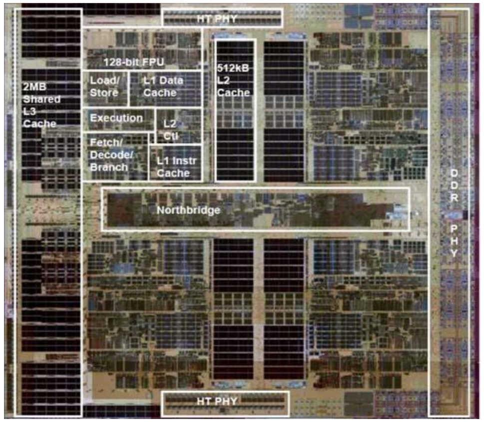
VRIJE UNIVERSITEIT BRUSSEL

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!

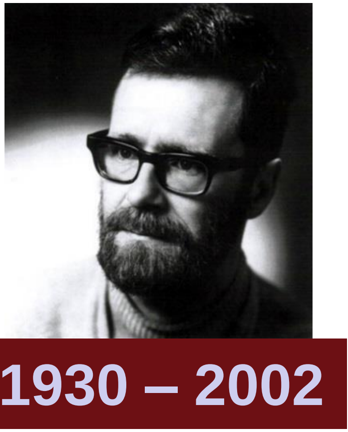
References

- Java Part 1
  https://blogs.oracle.com/javamagazine/post/java-thread-synchronization-raceconditions-locks-conditions
- Java Part 2:
  https://blogs.oracle.com/javamagazine/post/java-thread-synchronization-synchronized-blocks-adhoc-locks
  - https://onlinedisassembler.com/odaweb/
  - https://defuse.ca/online-x86-assembler.htm#disassembly2
- Synchronization in Java, Part 3: Atomic operations and deadlocks:
  https://blogs.oracle.com/javamagazine/post/java-thread-synchronization-volatile-final-atomic-deadlocks



7. Mutex implementation

A bit of history: Semaphores



- 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.

Who can change the shared data?

Two threads want to change the same data at exactly the same time.

- Cache line is changed
- Thread that is first in putting the ‘invalidate’ on the snoop bus
  - The bus arbitration module decides on control over the bus
- Other thread tries but notices it is too late, so the change fails

Thus: we try and test whether success

How a mutex lock should work

```
boolean access_x=true;

while (!access_x)
      ;
access_x=false;
// Critical section
...
access_x=true;
```

Should happen as an atomic operation

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

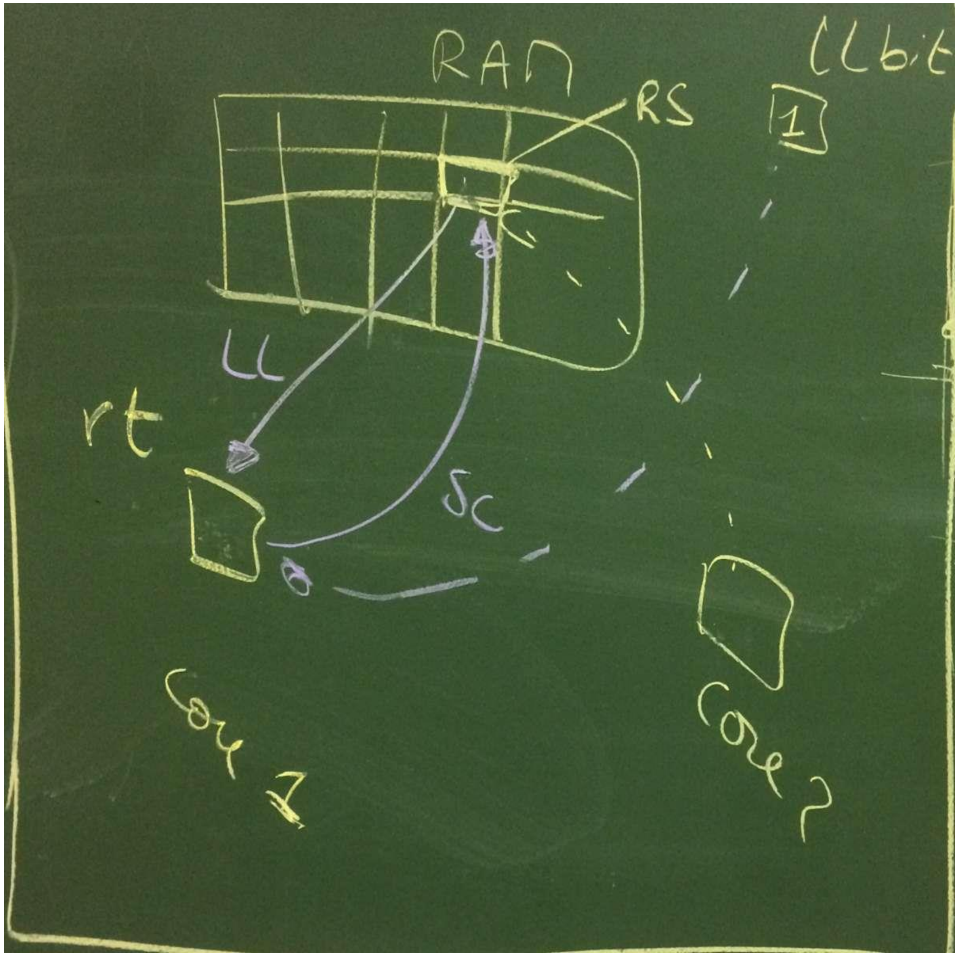
See https://www.cs.auckland.ac.nz/courses/compsci313s2c/resources/MIPSLLSC.pdf

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



Mutex implementation

- Hardware:
  - test-and-set
  - memory fence
- Waiting for lock:
  - 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

Condition variable implementation

- Mutex: hardware + OS
- Inactivation and reactivation of threads: OS