Parallel Systems Course: Chapter II

Shared-Memory Paradigm Multithreading

Jan Lemeire
Dept. ETRO
October 2013

Vrije Universiteit Brussel

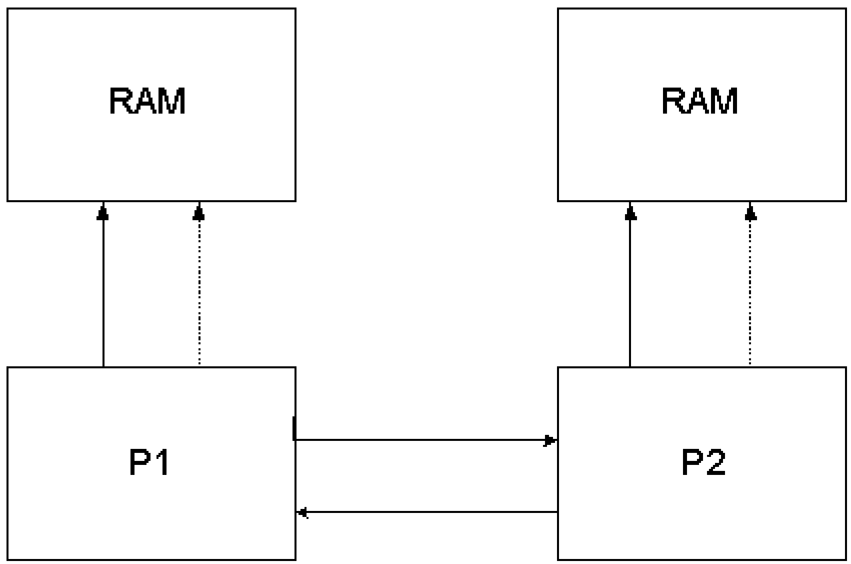
Overview

1. // processors and // instructions sequences
2. Architecture
3. Usage
4. Java Threads
5. POSIX Threads
6. Thread Safety
7. Synchronization Constructs
8. OpenMP and related
9. End Notes

Overview

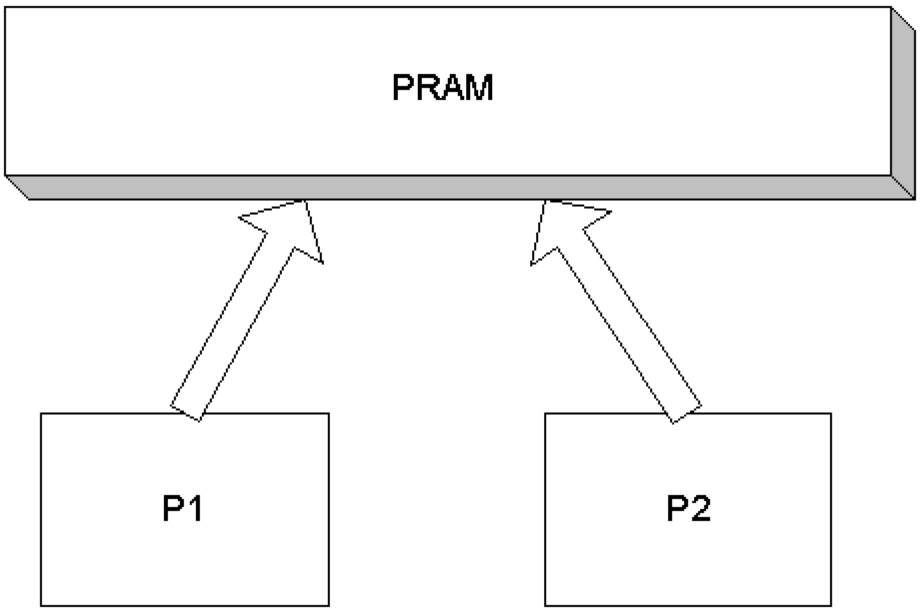
1. **// processors and // instructions sequences**
2. Architecture
3. Usage
4. Java Threads
5. POSIX Threads
6. Thread Safety
7. Synchronization Constructs
8. OpenMP and related
9. End Notes

I. Message-Passing Architectures



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

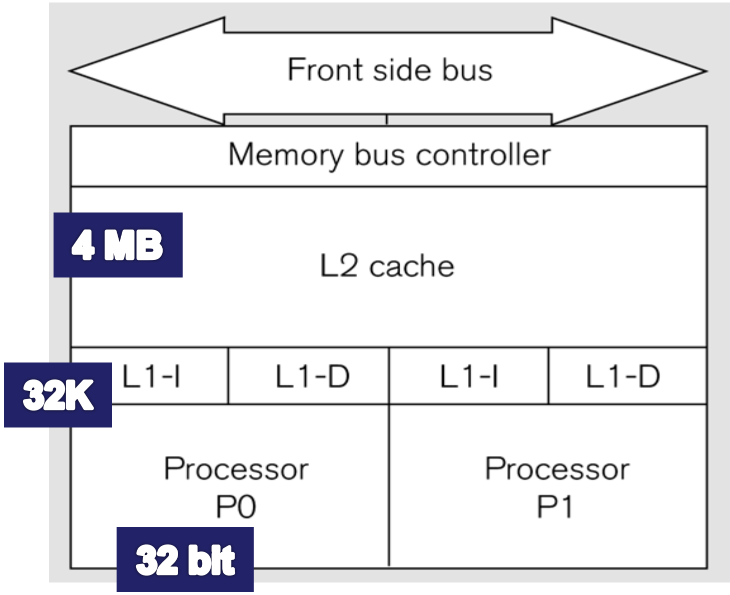
II. Shared Address-space Architectures



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

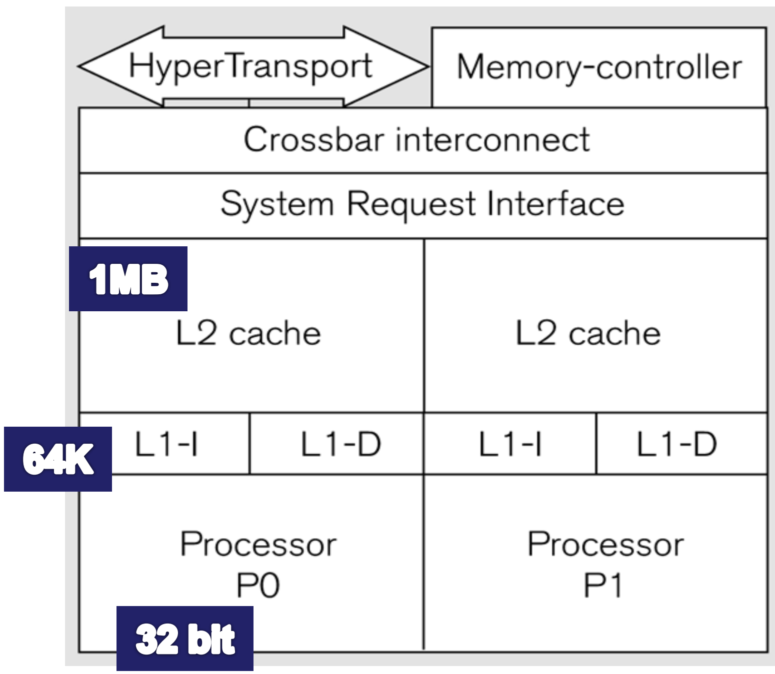
Intel Core Duo

PPP 45-58



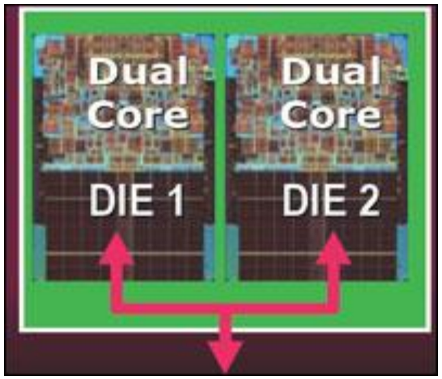
- Doubled memory bandwidth
- MESI cache coherence protocol, see later
  - Introduces overhead
- One processor can utilize the whole L2 cache

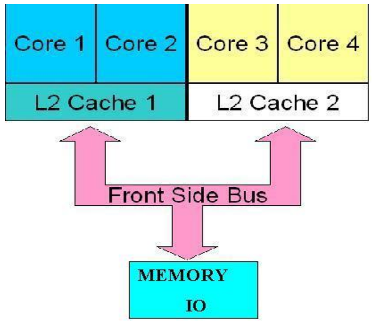
AMD Dual Core Opteron

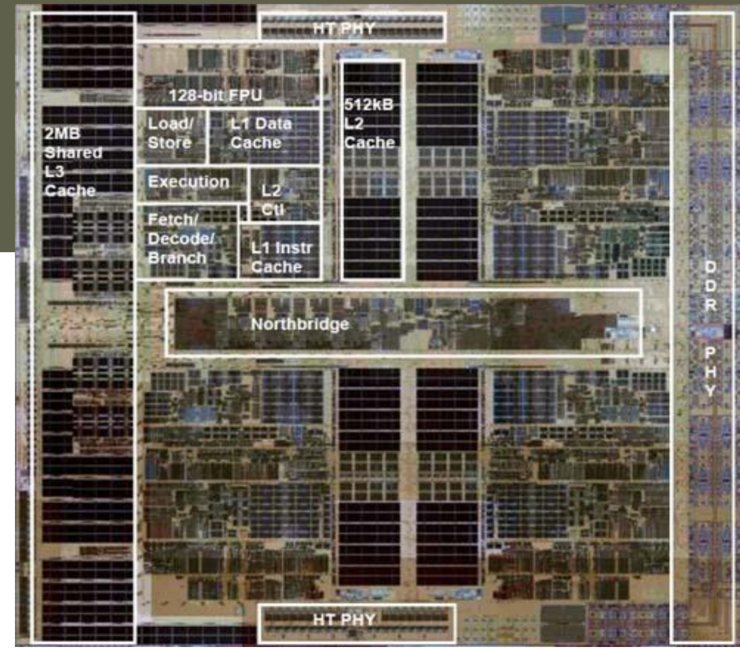


- System Request Interface handles memory coherence
  - MOESI protocol
- HyperTransport: for RAM requests
- Can be combined with that of other processors => SMPs

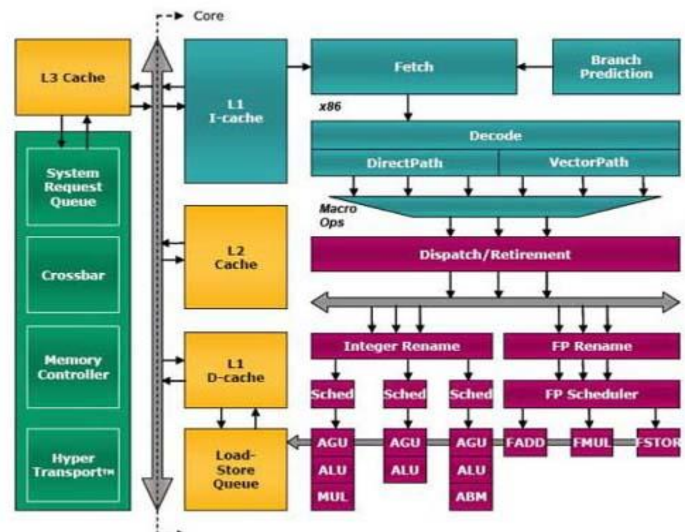
Quadcores



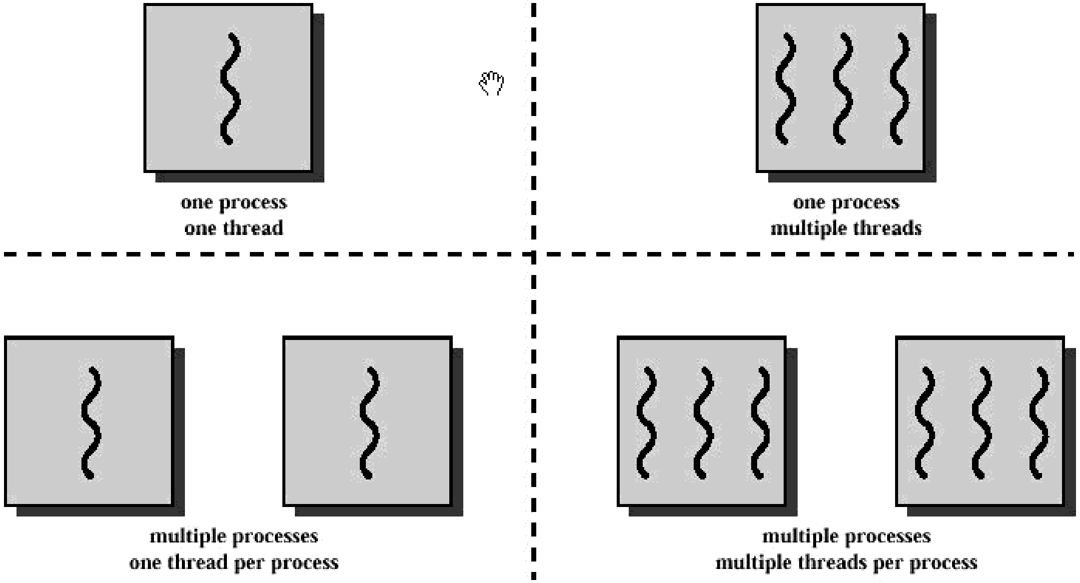




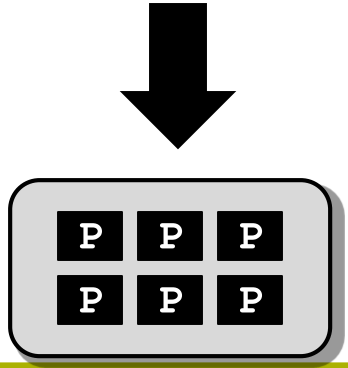
- **More useful than a dual core?**
  - Office applications: NO
  - Special applications such as photoshop: YES
  - Games: IF MULTI-THREADED
  - Scientific applications: IF MULTI-THREADED



Processes versus Threads



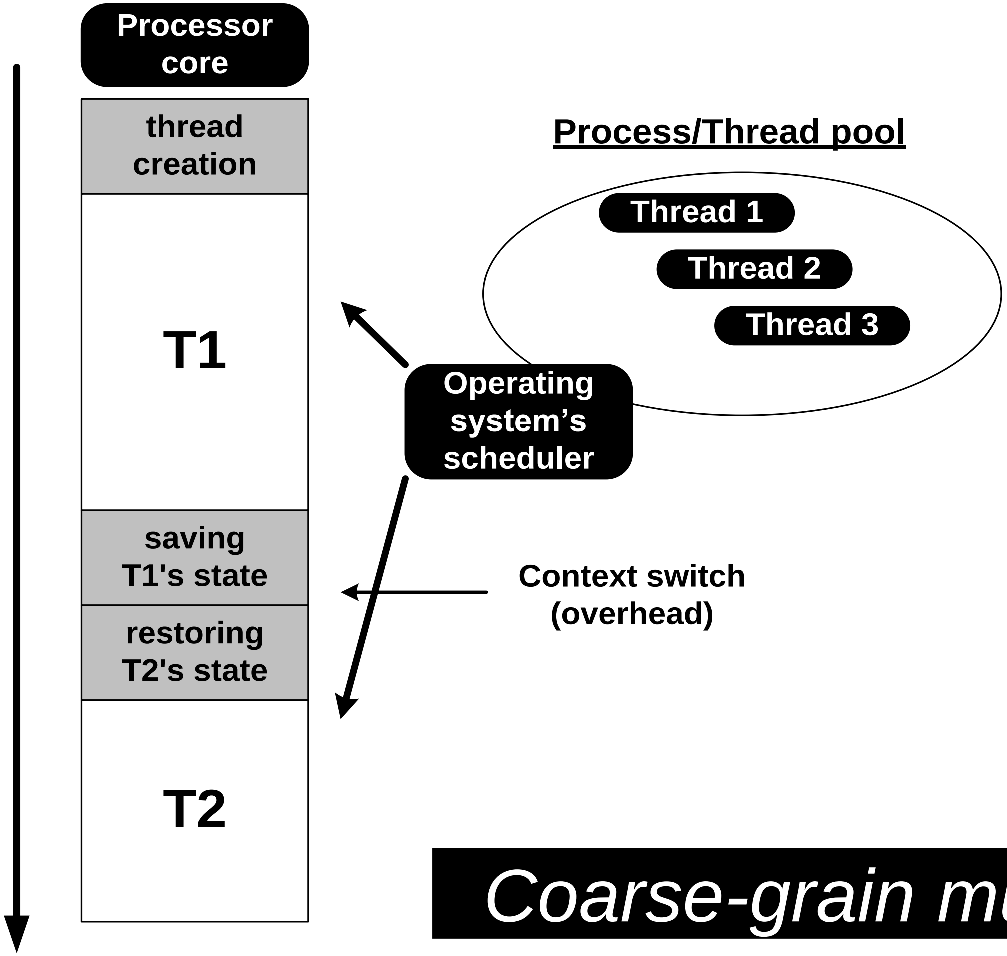
Scheduled by the OS on the available processors



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

Running threads on multiple cores

- Active threads in thread pool
- Scheduled by operating system
- Threads (or processes) can be migrated from 1 core to another

LINK 11

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!

Overview

1. // processors and // instructions sequences
2. **Architecture**
3. Usage
4. Java Threads
5. POSIX Threads
6. Thread Safety
7. Synchronization Constructs
8. OpenMP and related
9. End Notes

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

1. Connect PROCs to MEMs (the interconnect)
2. Address concurrent read/writes
3. Cache coherence

Typical architectures

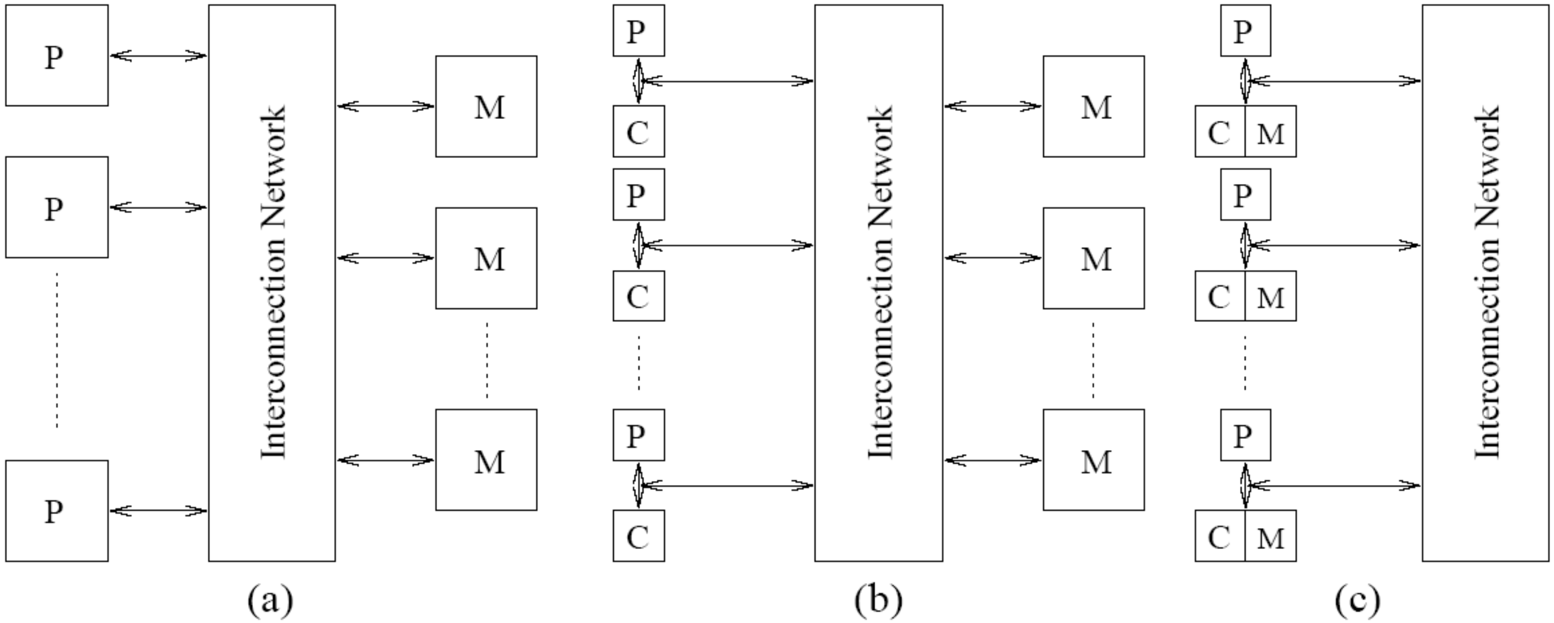
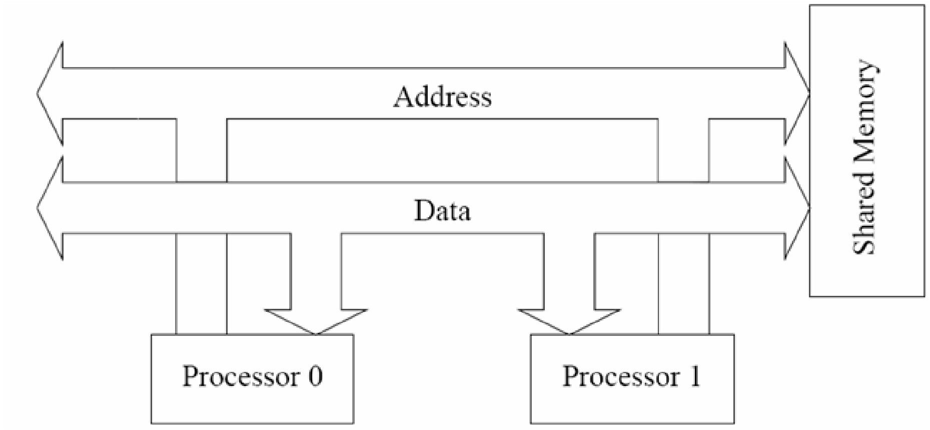
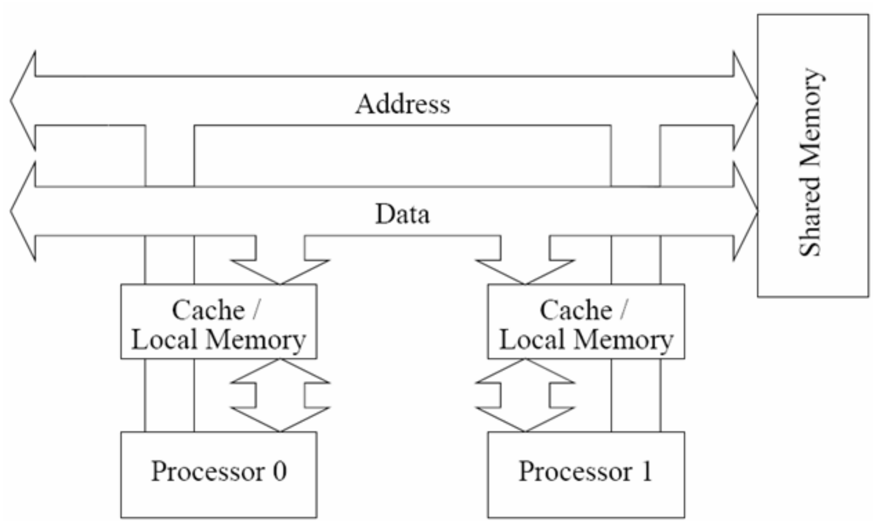


Figure 2.5 Typical shared-address-space architectures: (a) Uniform-memory-access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.

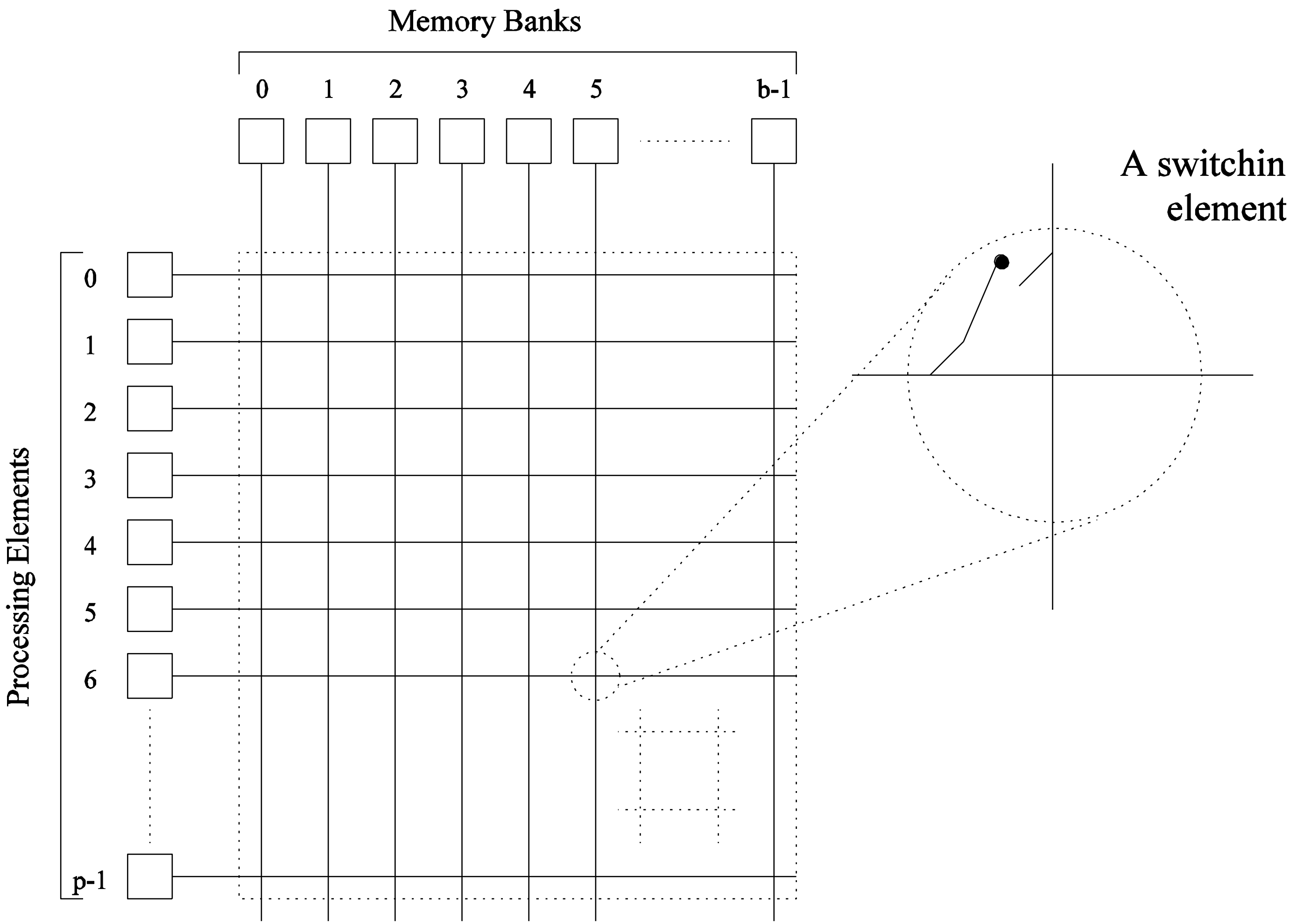
Bus-based Interconnects



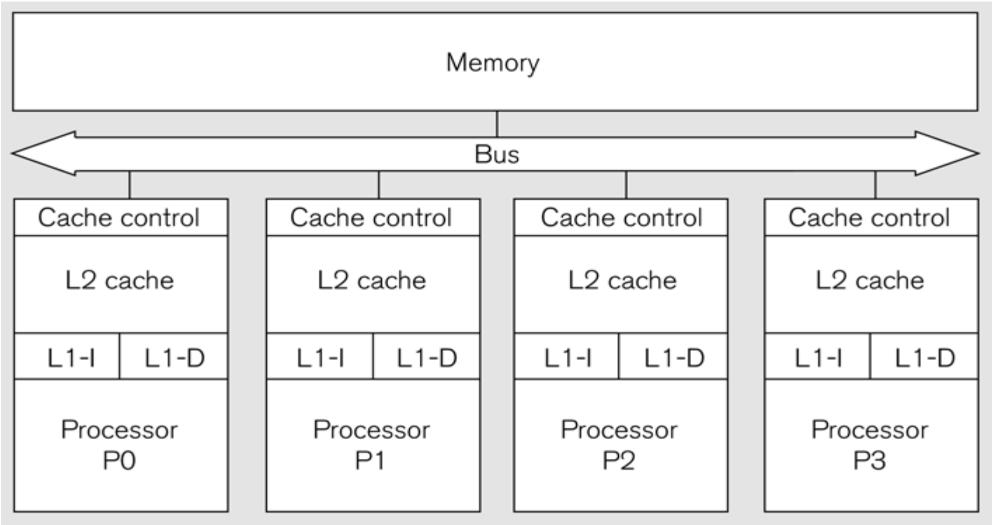


With local memory/cache

Crossbar switches

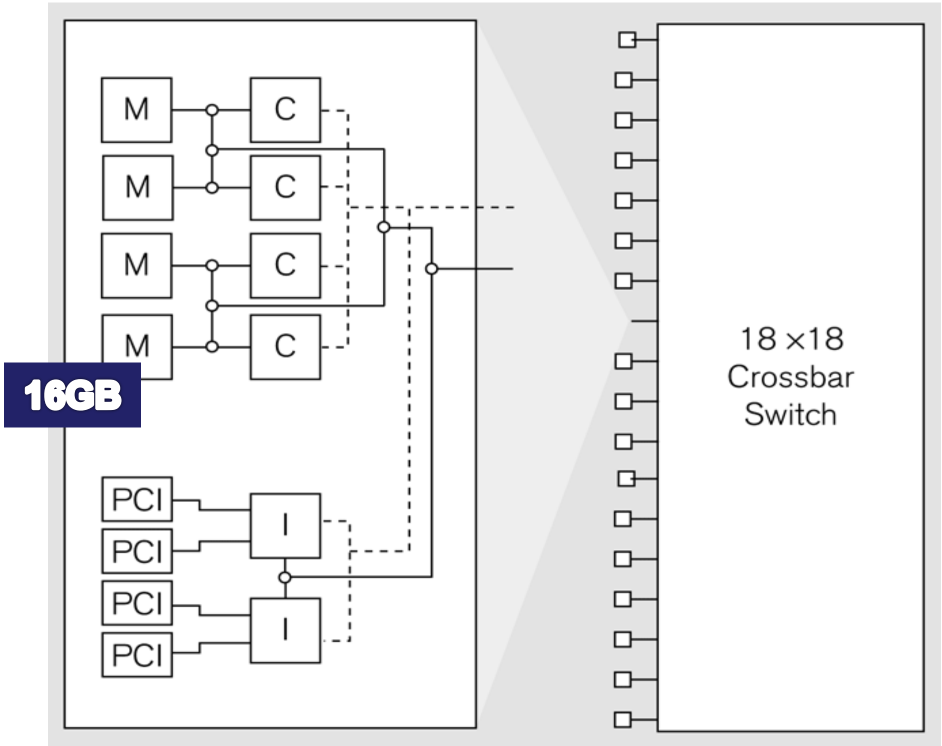


Symmetric Multiprocessor Architectures (SMPs)

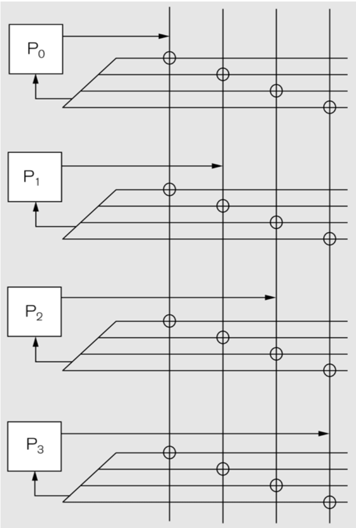


- Cf AMD architecture
- Bus is potential bottleneck
- ⇒ Number of SMPs is limited

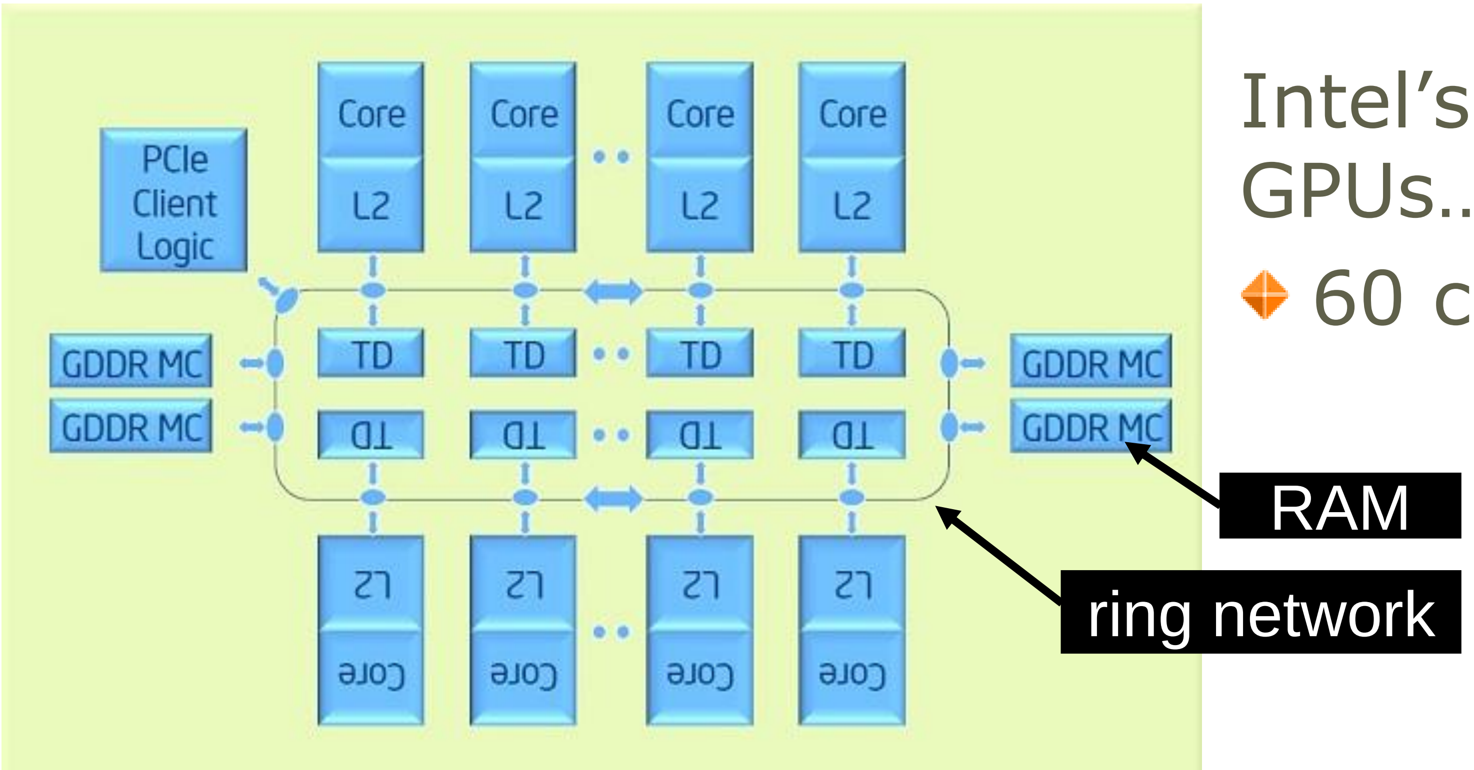
Sun Fire E25K (SMP)



- Up to 72 processors
  - Each can handle 2 **hardware threads**
- Total: 1.15TB
- 3 crossbars
  - **~ n²**



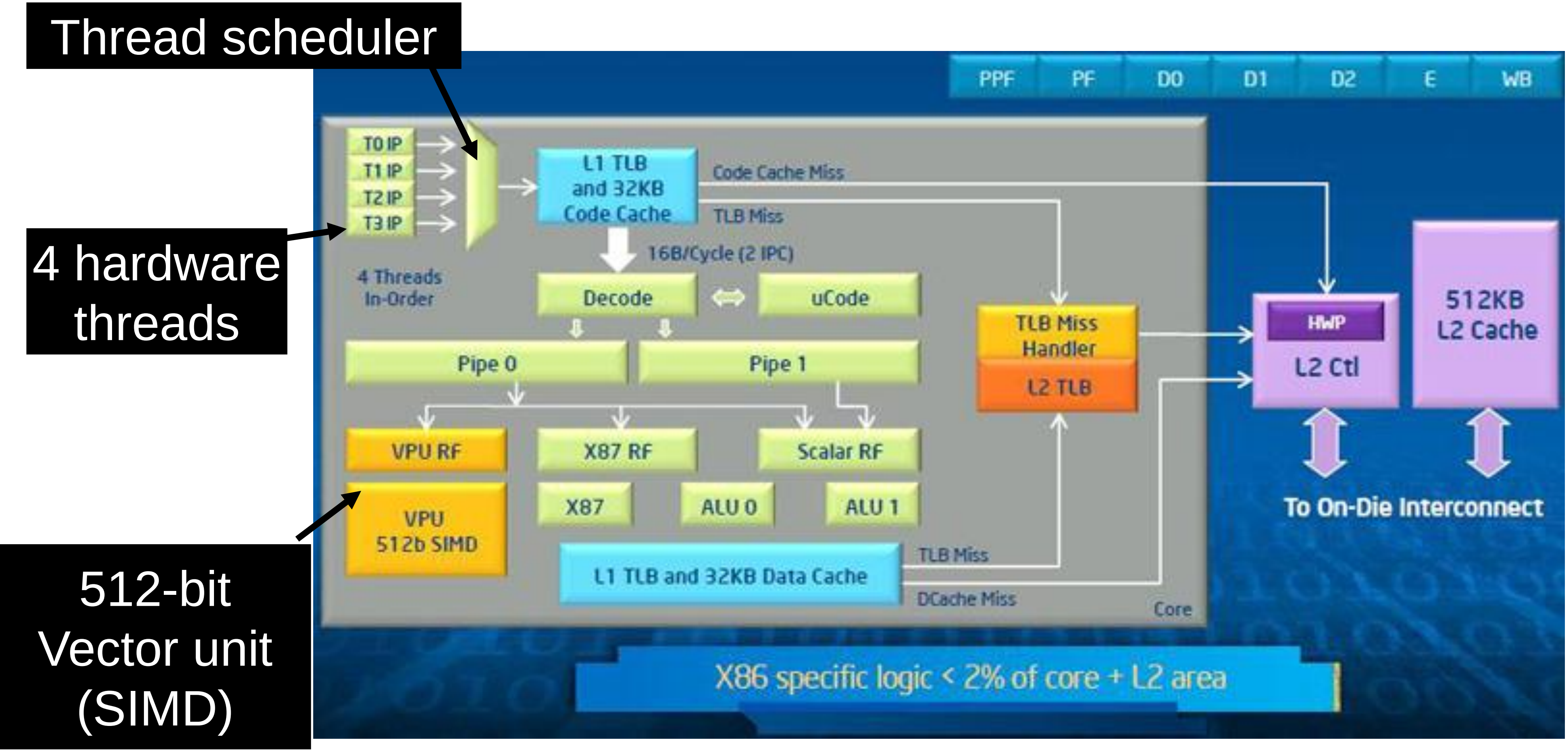
Intel's Xeon Phi coprocessor



Intel's response to GPUs…

- 60 cores

Intel’s Xeon Phi’s core



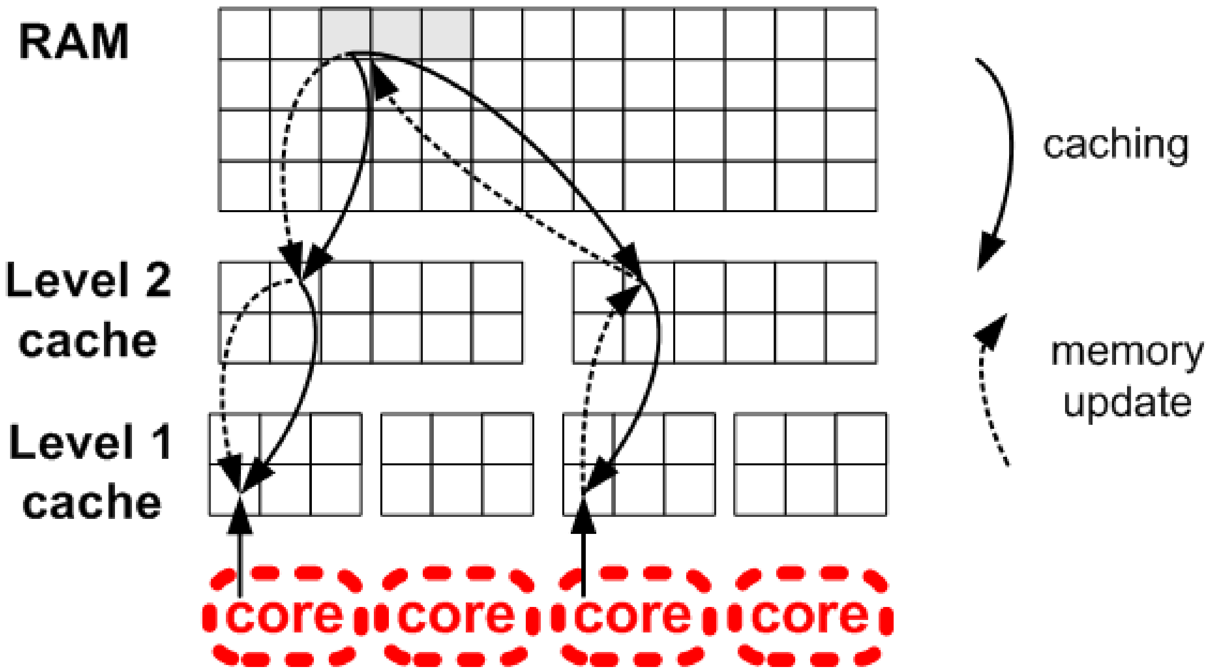
2. PRAM Architectures

- 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.

3. Caching & memory coherence



Several copies of same data reside in memory

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

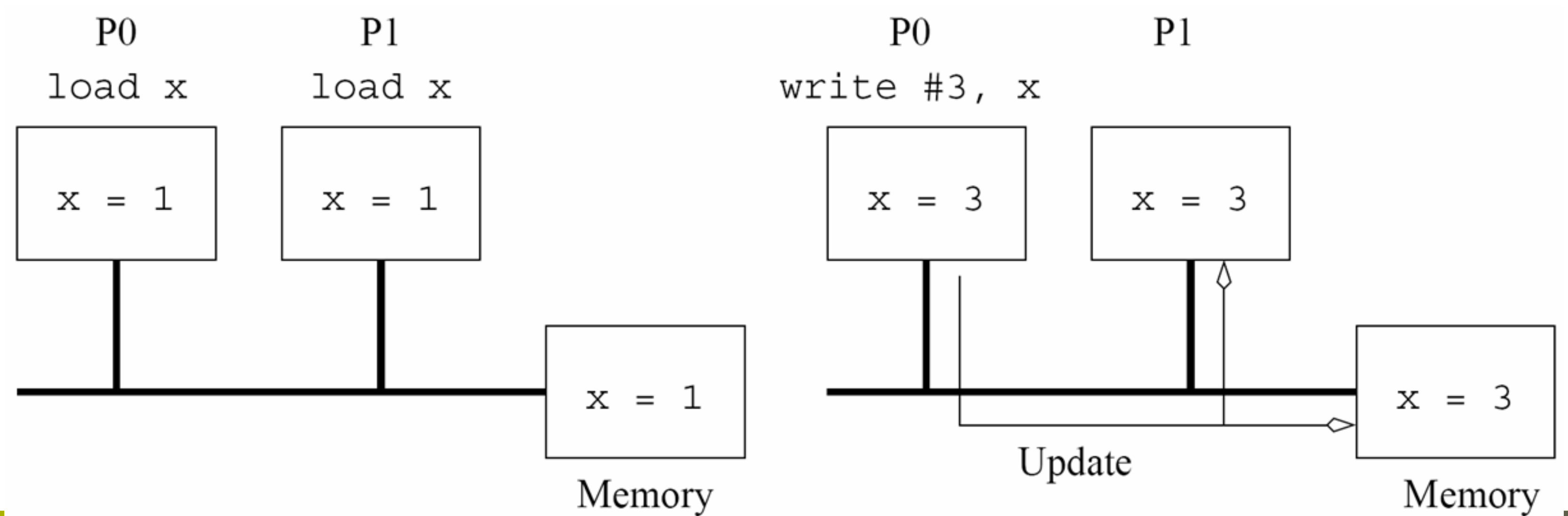
False sharing

- Based on cache line (64 or 128 bytes)
- 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

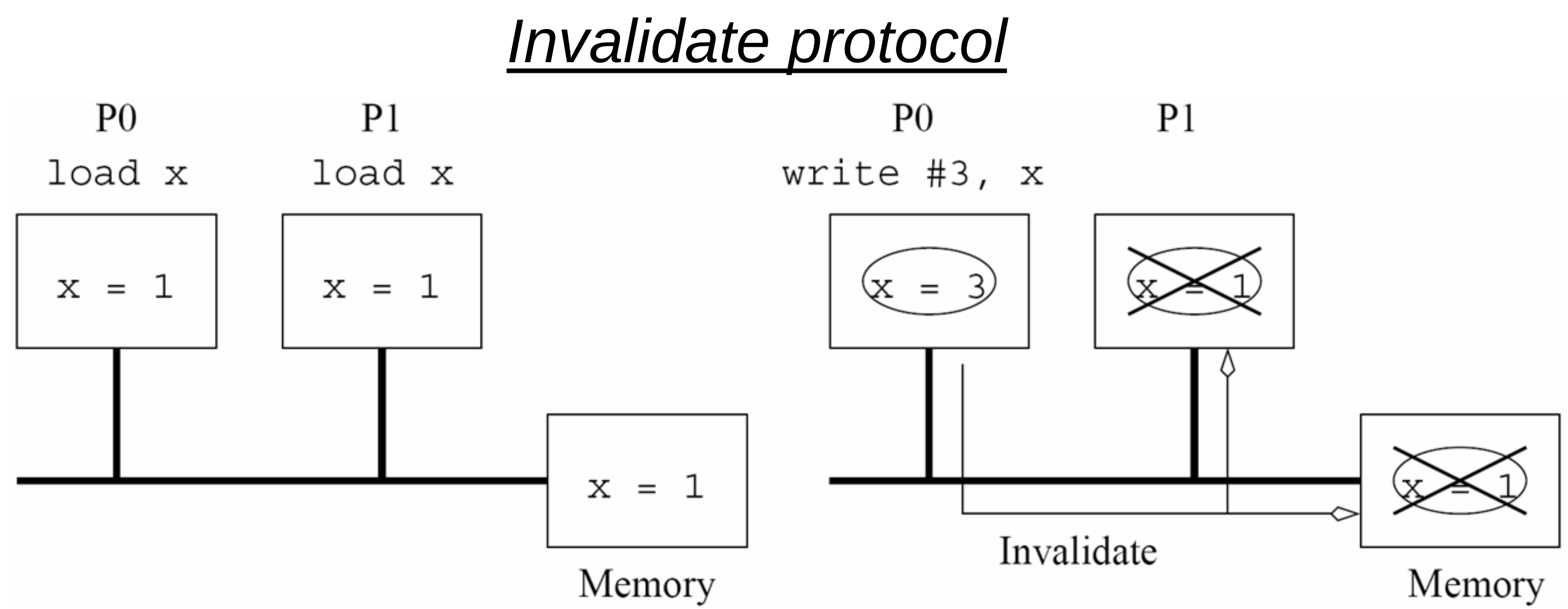
- To keep copies of data in different memory levels consistent!
  - Is not always performed. Best effort.
  - Or by explicit synchronization triggered by software (see later).

<u>Update protocol</u>



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



MESI-protocol

Possible states of a cache line:

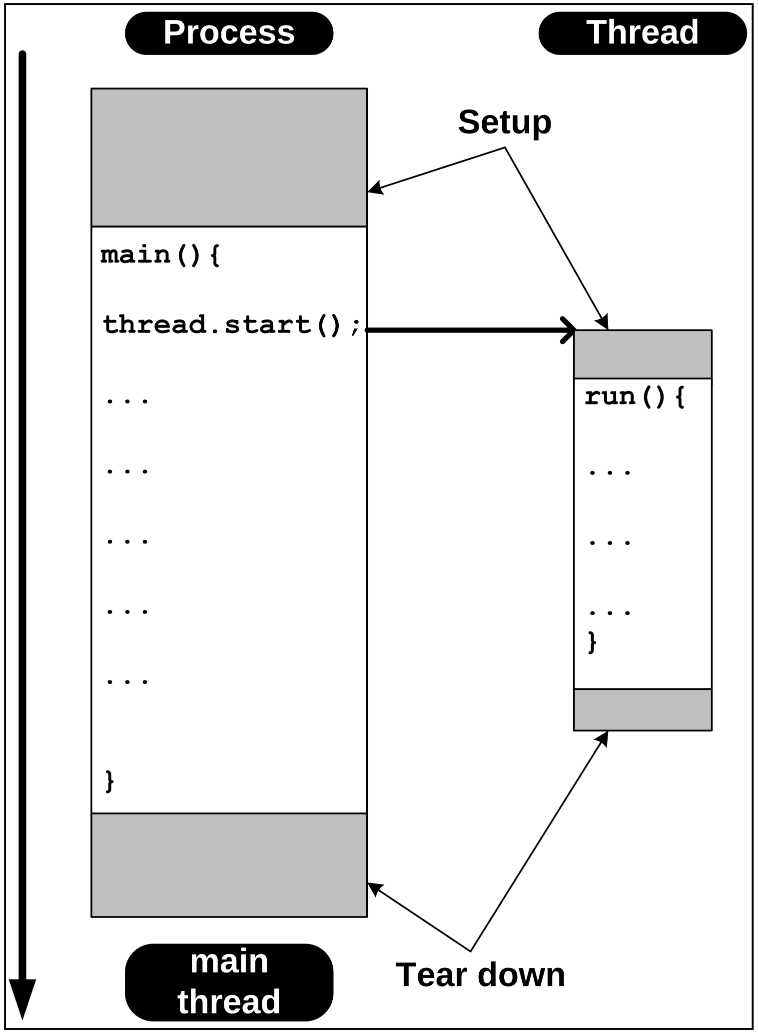
State	Cacheline Valid?	Valid in memory?	Copy in other cache?	Write access
Modified	Yes	No	No	Cache
Exclusive	Yes	Yes	No	Cache
Shared	Yes	Yes	Possible	Cache/Memory
Invalid	No	Unknown	Possible	Memory

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

Overview

1. // processors and // instructions sequences
2. Architecture
3. **Usage**
4. Java Threads
5. POSIX Threads
6. Thread Safety
7. Synchronization Constructs
8. OpenMP and related
9. End Notes

Thread creation

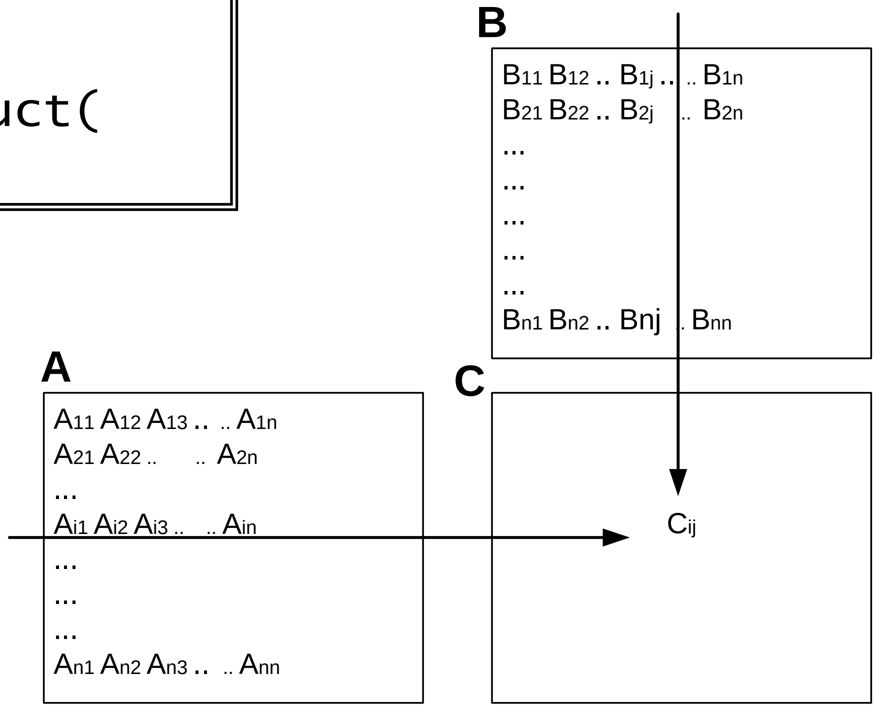


- A **thread** is basically a *lightweight* process
- A process : unit of resource ownership
  - a virtual address space to hold the process image
  - control of some resources (files, I/O devices...)
- A thread is an execution path
  - Has access to the memory address space and resources of its process.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



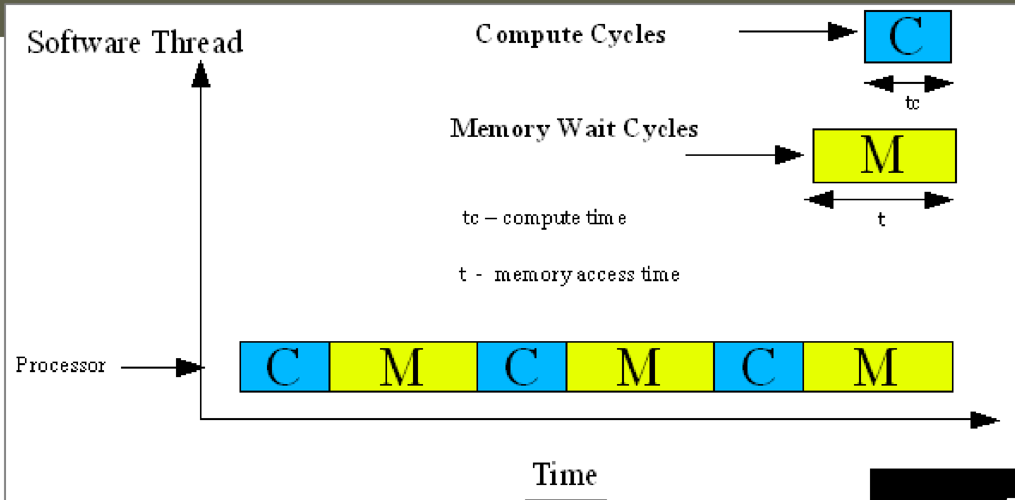
In this case, one may think of the thread as an instance of a function that returns before the function has finished executing.

Why Threads?

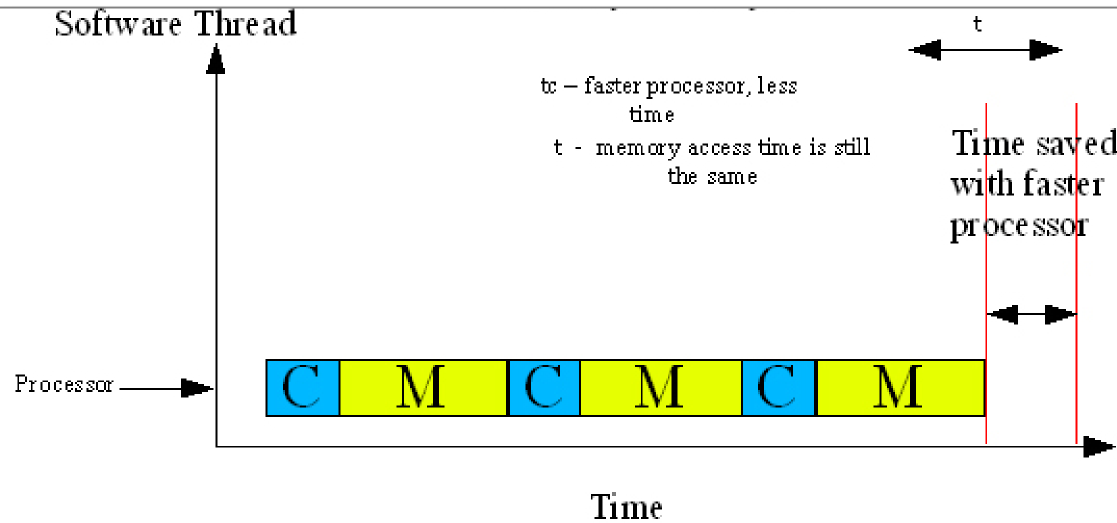
- Software Portability
  - run on serial and parallel machines
- Latency Hiding
  - While one thread has to wait, others can utilize CPU
  - For example: file reading, message reading, reading data from higher-level memory
- Scheduling and Load Balancing
  - Large number of concurrent tasks
  - System-level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

Latency Hiding

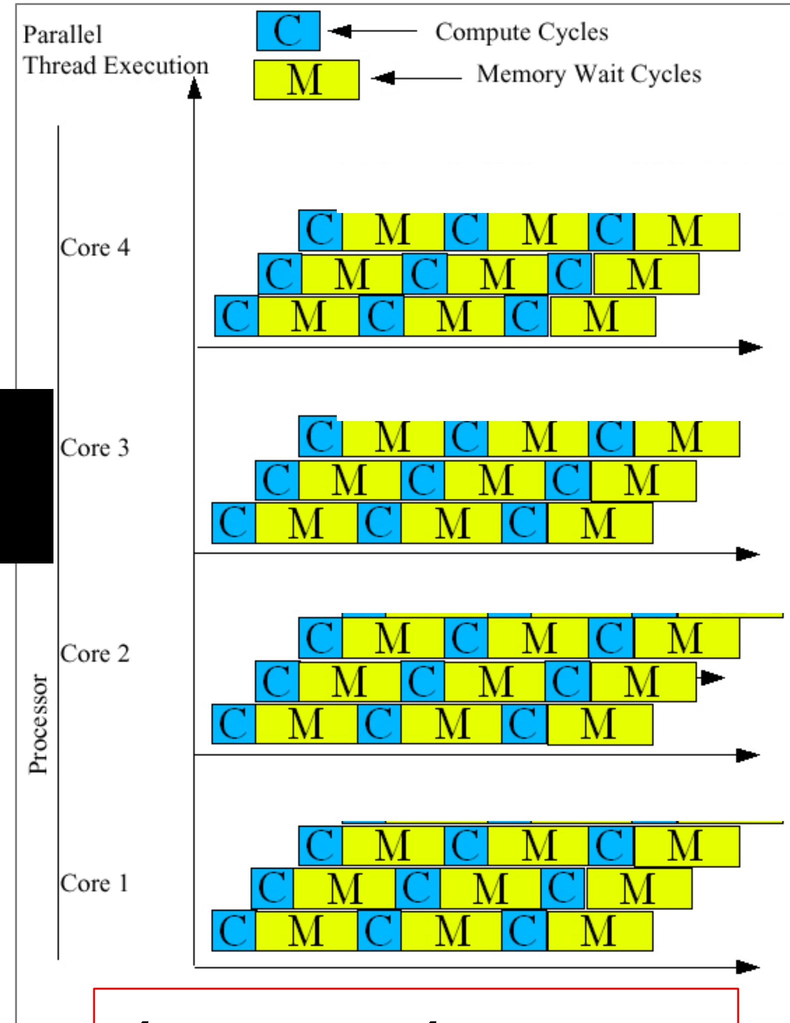
LINK 11



Faster CPU



More threads



4 cores: x4

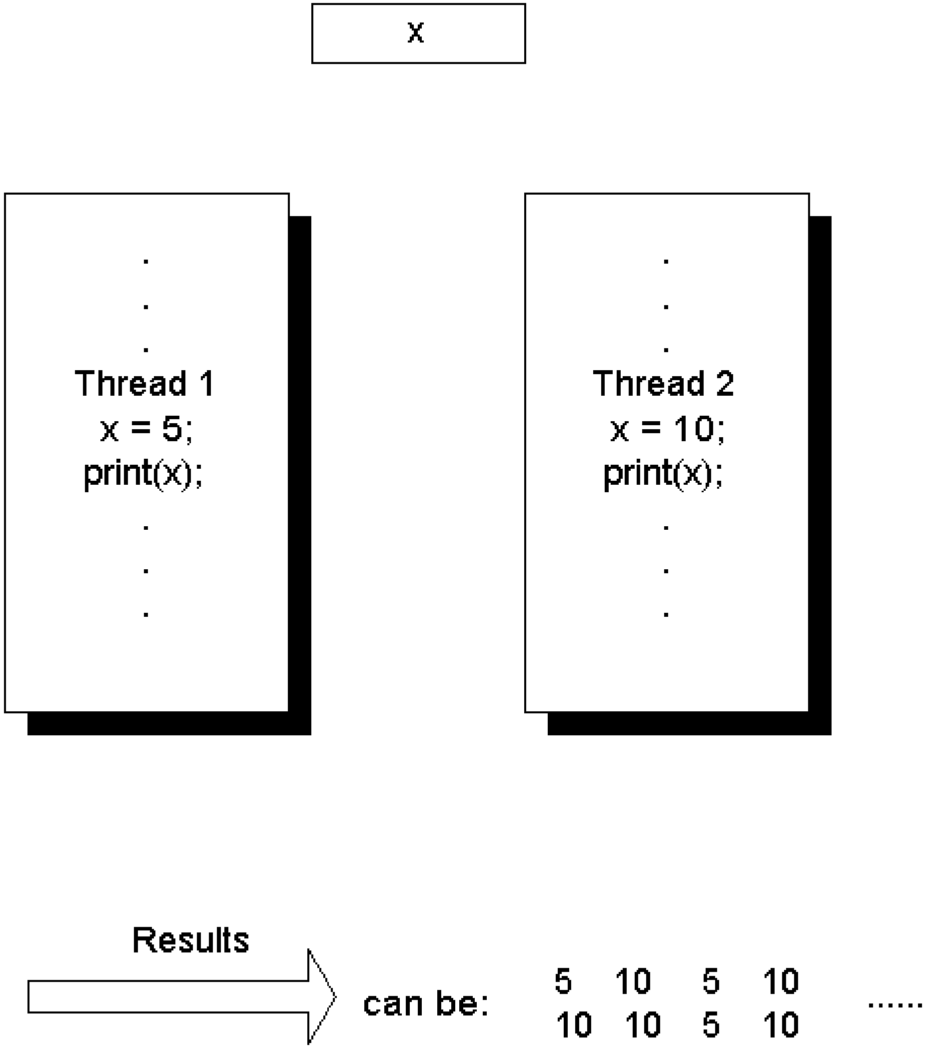
Latency hiding: x3

Example why synchronization is necessary.

- x is initially set to 1
- One thread executes: x = 10; print(x);
- Second thread executes: 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.

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
- ***Operating System support needed!***

Ps. There is a software solution for this: Peterson Algorithm

Critical sections trigger cache coherence

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

LINK 1

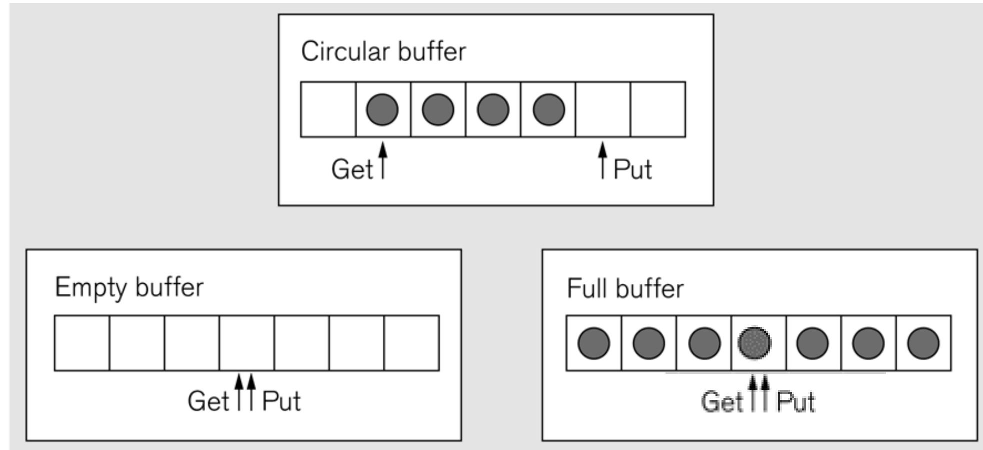
Producers-Consumers Scenario

Producer Threads

```
...
Produce thing
Put in buffer
...
```

Consumer Threads

```
...
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
...
```

***Question**: can synchronization be implemented with only locks?*

2. Also needed: proper locking of critical sections, *see later…*

Multi-threading primitives

Should minimally allow the following:

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

Multi-threading primitives

Should minimally allow the following:

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

!

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`

Overview

1. // processors and // instructions sequences
2. Architecture
3. Usage
4. **Java Threads**
5. POSIX Threads
6. Thread Safety
7. Synchronization Constructs
8. OpenMP and related
9. End Notes

The Java Thread Class

PPP 202

`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.

LINK 2

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();
```

LINK 3

Synchronized methods & blocks

1

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

2

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

is identical to

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

Static methods

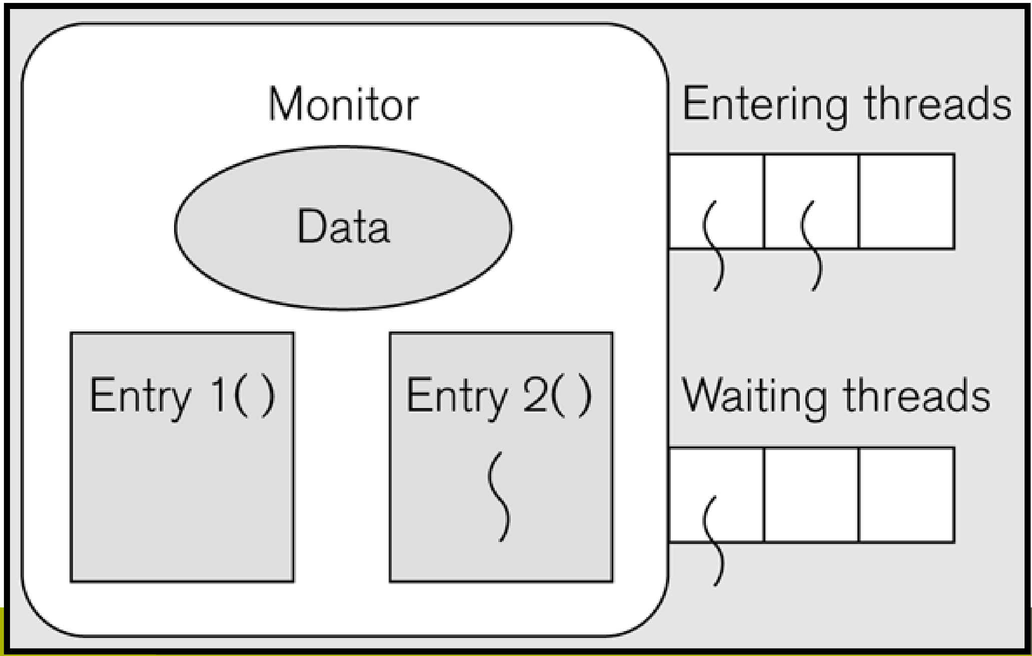
```
synchronized static void method(){
      …
}
```

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

PPP 29

```
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

```
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

PPP 32

```
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

LINK 4

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

LINK 4&5

Only for atomic operations

Which code is thread-safe?

```
volatile int x;
…
X++;
…
```

```
volatile int x;
…
X=5;
…
```

```
volatile int best_cost;
…
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 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*

LINK 4

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!**

LINK 2

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

LINK 6

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()

LINK 7

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(...)) ;`

LINK 3

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

Overview

1. // processors and // instructions sequences
2. Architecture
3. Usage
4. Java Threads
5. **POSIX Threads**
6. Thread Safety
7. Synchronization Constructs
8. OpenMP and related
9. End Notes

The POSIX Thread API

KUMAR Chapter 7

PPP Chapter 6

- 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` is returned in the location pointer `**ptr`.

Example

KUMAR

```
#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.

```
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&&(put+get)==SIZE-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;
}
```

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 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

Overview

1. // processors and // instructions sequences
2. Architecture
3. Usage
4. Java Threads
5. POSIX Threads
6. **Thread Safety**
7. Synchronization Constructs
8. OpenMP and related
9. End Notes

PPP 171-173

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!

PPP 173-174

Mistake in PPP on page 173!!

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

```
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);
```

OK

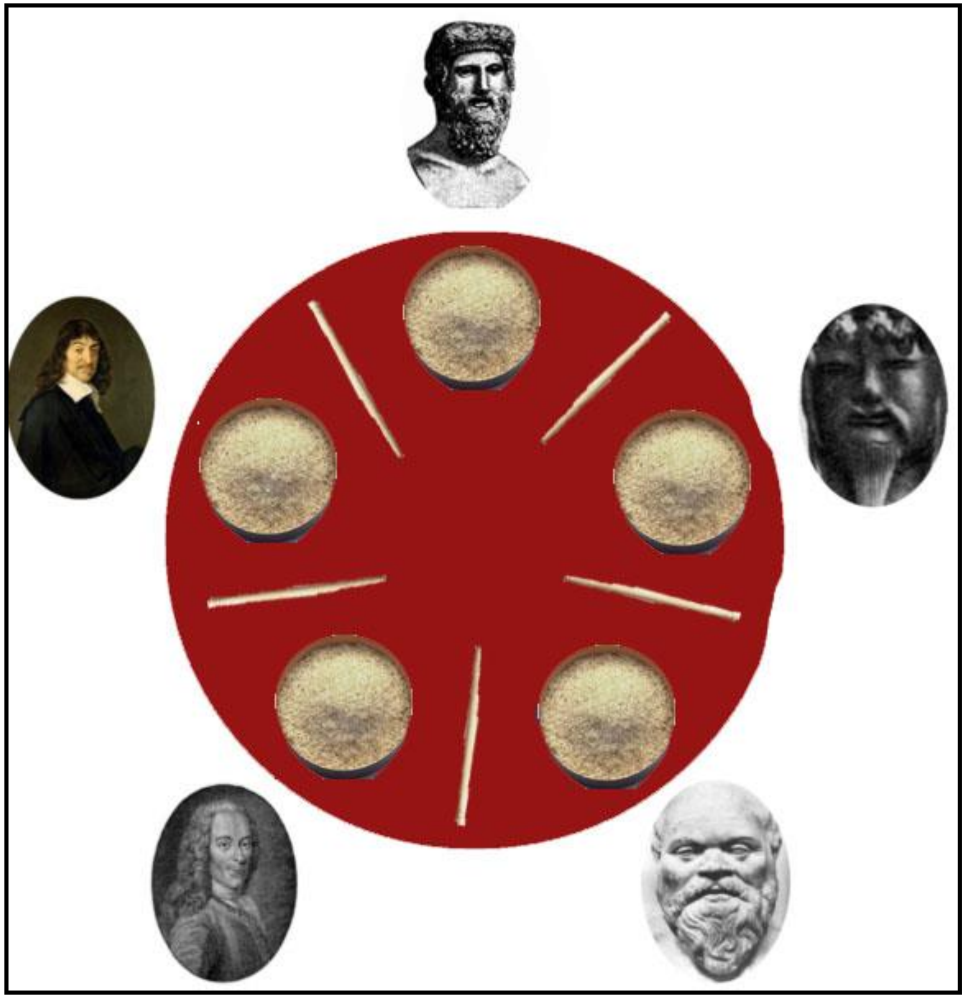
Only 1 cond variable

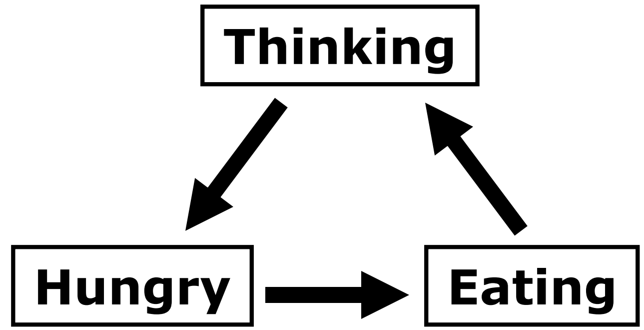
```
pthread_mutex_lock(&lock);
while (apples==0 || oranges==0){
        pthread_cond_wait(&more_apples_or_more
oranges, &lock);
}
// eat apple & orange
pthread_mutex_unlock(&lock);
```

OK

LINK 8

The Dining Philosophers



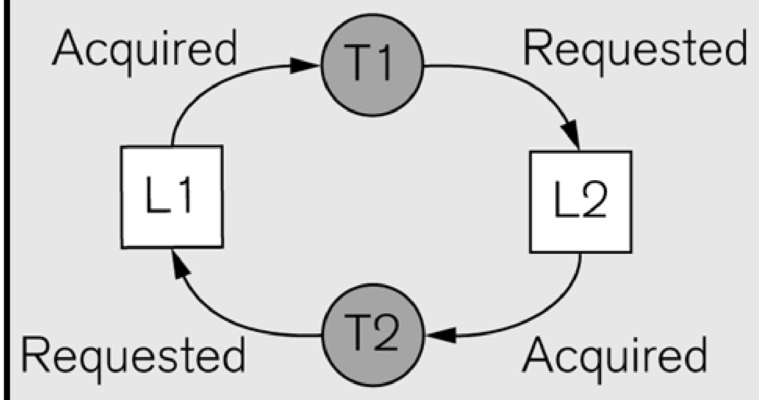


The philosophers do not speak to each other and there is no arbiter organizing the resources

➡ **Deadlock...**

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



Resource allocation graph

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.

Overview

1. // processors and // instructions sequences
2. Architecture
3. Usage
4. Java Threads
5. POSIX Threads
6. Thread Safety
7. **Synchronization Constructs**
8. OpenMP and related
9. End Notes

Composite Synchronization Constructs

KUMAR Chapter 7

- By design, **Pthreads** provide support for a basic set of operations.
- Higher level constructs can be built using basic synchronization constructs.
- We discuss two such constructs - *barriers* and *read-write locks*.

Barriers

- Holds a thread until all threads participating in the barrier have reached it.
- Can be implemented using a counter, a mutex and a condition variable.
  - A single integer is used to keep track of the number of threads that have reached the barrier.
  - If the count is less than the total number of threads, the threads execute a condition wait.
  - The last thread entering (and setting the count to the number of threads) wakes up all the threads using a condition broadcast.
- Release of lock and reactivation of threads must happen atomically
  - Otherwise problematic when barrier is reused…

Barriers

```
typedef struct {
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed;
    int count;
} mylib_barrier_t;

void mylib_init_barrier(mylib_barrier_t *b) {
    b -> count = 0;
    pthread_mutex_init(&(b -> count_lock), NULL);
    pthread_cond_init(&(b -> ok_to_proceed), NULL);
}
```

Barriers

```
void mylib_barrier (mylib_barrier_t *b, int num_threads) {
   pthread_mutex_lock(&(b -> count_lock));
   b -> count ++;
   if (b -> count == num_threads) {
      b -> count = 0;
      pthread_cond_broadcast(&(b -> ok_to_proceed));
   }
   else
      while (pthread_cond_wait(&(b -> ok_to_proceed),
         &(b -> count_lock)) != 0)
         ;
   pthread_mutex_unlock(&(b -> count_lock));
}
```

Barriers

- The barrier described above is called a *linear barrier*.
- The trivial lower bound on execution time of this function is therefore ***O(n)*** for ***n*** threads.
  - Threads are released one by one, since mutex count_lock is passed among them one after the other!
- Can be speeded up using multiple barrier variables organized in a tree.

Log Barrier

- We use ***n/2*** condition variable-mutex pairs for implementing a barrier for ***n*** threads.
- At the lowest level, threads are paired up and each pair of threads shares a single condition variable-mutex pair.
- Once both threads arrive, one of the two moves on, the other one waits.
- This process repeats up the tree.
- This is also called a log barrier and its runtime grows as ***O(log n)***.

Readers-writers problem

- "Many threads must access the same shared memory at one time, some reading and some writing, with the natural constraint that no process may access the share for reading or writing while another process is in the act of writing to it."
- Data structure is read frequently but written infrequently
  - use read-write locks ***instead of traditional locking.***
- A read lock is granted when there are other threads that may already have read locks.
- If there is a write lock on the data (or if there are queued write locks), the thread performs a condition wait.
- **Pending writers get priority over pending readers.**
- If there are multiple threads requesting a write lock, they must perform a condition wait.
- With this description, we can design functions for read locks `mylib_rwlock_readlock`, write locks `mylib_rwlock_writelock`, and unlocking `mylib_rwlock_unlock`.

Read-Write Locks

- The lock data type `mylib_rwlock_t` holds the following:
  - a count of the number of readers,
  - a count of pending writers
  - A boolean specifying whether a writer is present,
  - a mutex `read_write_lock` associated with the shared data structure,
  - a condition variable `readers_proceed` that is signaled when readers can proceed,
  - a condition variable `writer_proceed` that is signaled when one of the writers can proceed

Read-Write Locks

```
typedef struct {
    int readers;
    bool writer;
    pthread_cond_t readers_proceed;
    pthread_cond_t writer_proceed;
    int pending_writers;
    pthread_mutex_t read_write_lock;
} mylib_rwlock_t;

void mylib_rwlock_init (mylib_rwlock_t *l) {
    l->readers = l->pending_writers = 0;
    l->writer = false;
    pthread_mutex_init(&(l->read_write_lock), NULL);
    pthread_cond_init(&(l->readers_proceed), NULL);
    pthread_cond_init(&(l->writer_proceed), NULL);
}
```

Read-Write Locks

```
void mylib_rwlock_readlock(mylib_rwlock_t *l) {
   /* if there is a write lock or pending writers, perform
     condition wait.. else increment count of readers and grant
     read lock */
   pthread_mutex_lock(&(l->read_write_lock));
   while ((l->pending_writers > 0) || l->writer)
      pthread_cond_wait(&(l->readers_proceed),
      &(l->read_write_lock));
   l->readers ++;
   pthread_mutex_unlock(&(l->read_write_lock));
}
```

Read-Write Locks

```
void mylib_rwlock_writelock(mylib_rwlock_t *l) {
    /* if there are readers or a writer, increment pending
      writers count and wait. On being woken, decrement pending
      writers count and set writer */

    pthread_mutex_lock(&(l->read_write_lock));
    while (l->writer || (l->readers > 0)) {
        l->pending_writers ++;
        pthread_cond_wait(&(l->writer_proceed),
            &(l->read_write_lock));
        l->pending_writers --;
    }
    l->writer = true;
    pthread_mutex_unlock(&(l->read_write_lock));
}
```

Error in Kumar: should be inside loop

Read-Write Locks

```
void mylib_rwlock_unlock(mylib_rwlock_t *l) {
/* if there is a write lock then unlock, else if there are read
   locks, decrement count of read locks. If the count is 0 and
   there is a pending writer, let it through, else if there are
   pending readers, let them all go through */
pthread_mutex_lock(&(l->read_write_lock));
if (l->writer) // I'm a writer
   l->writer = false;
else if (l->readers > 0) // I'm a reader
   l->readers --;
pthread_mutex_unlock(&(l->read_write_lock));
if ((l->readers == 0) && (l->pending_writers > 0))
   pthread_cond_signal(&(l->writer_proceed));
else if (l -> readers > 0)
   pthread_cond_broadcast(&(l->readers_proceed));
}
```

WRONG: l->readers is zero when writer leaves

What if pending writers should get access asap

- What if readers unlock, but there are other readers busy (readers > 0) and there are pending writers *and* pending readers
  - Readers should not be released!
  - Should be: `else if (l->pending_writers == 0)`

Conditional wait inside a loop

```
while (!predicate)
      pthread_cond_wait(…)
```

- When waiting on condition variables, the wait should be inside a loop, not in a simple if statement because of **spurious wakeups**.
  - cf http://en.wikipedia.org/wiki/Spurious_wakeup
  - You are not guaranteed that if a thread wakes up, it is the result of a signal or broadcast call.
- Spurious wakeups may occur => the return does not imply anything about the value of the predicate => the predicate should be re-evaluated.
  - http://opengroup.org/onlinepubs/007908799/xsh/pthread_cond_wait.html

Overview

1. // processors and // instructions sequences
2. Architecture
3. Usage
4. Java Threads
5. POSIX Threads
6. Thread Safety
7. Synchronization Constructs
8. **OpenMP and related**
9. End Notes

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

```
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;
}
```

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!

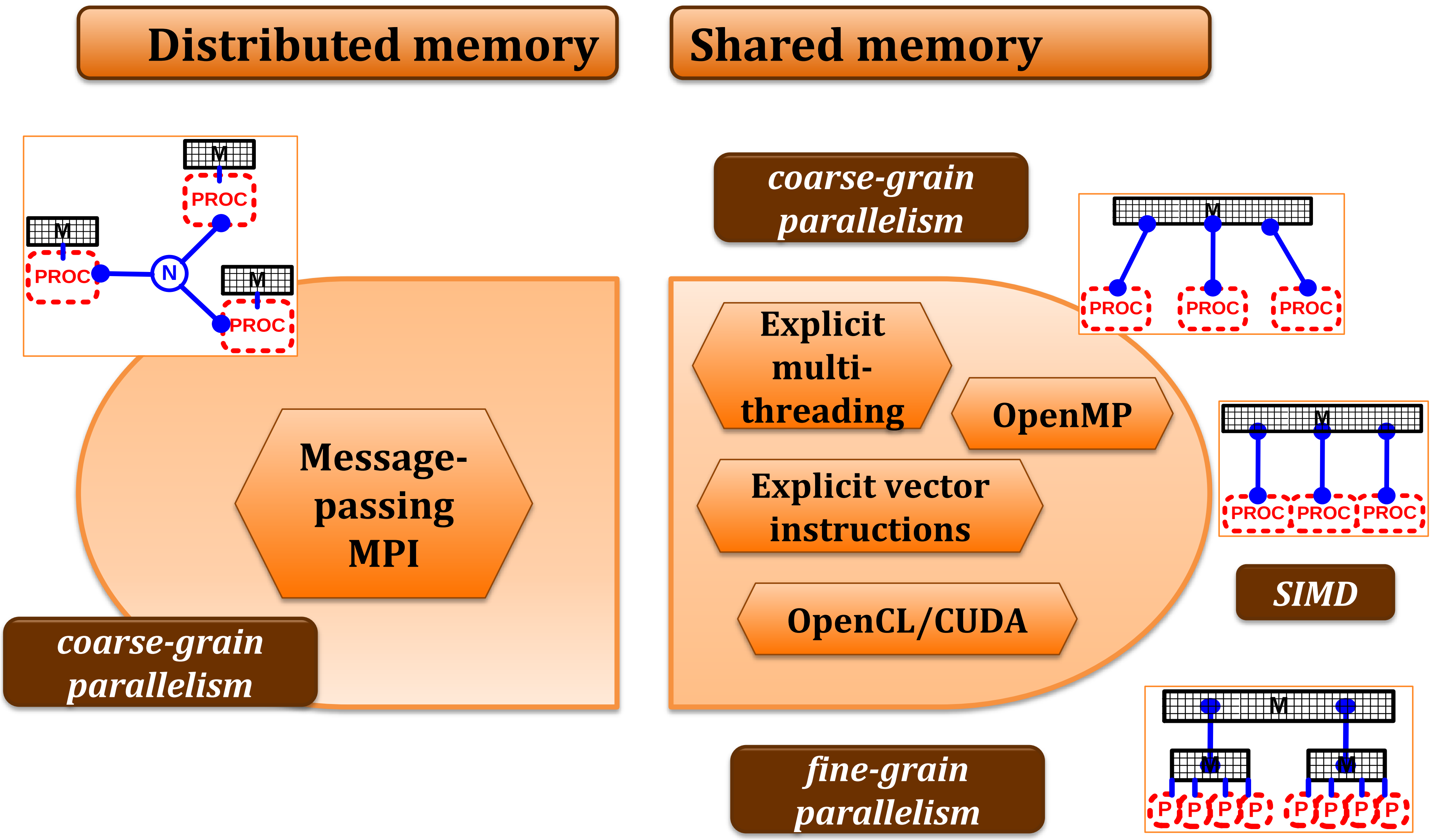
Overview

1. // processors and // instructions sequences
2. Architecture
3. Usage
4. Java Threads
5. POSIX Threads
6. Thread Safety
7. Synchronization Constructs
8. OpenMP and related
9. **End Notes**

Keep in mind when Designing Asynchronous Programs

- Never rely on scheduling assumptions when exchanging data.
- Never rely on liveness of data resulting from assumptions on scheduling.
- Do not rely on scheduling as a means of synchronization.
- Use synchronization mechanisms with mutexes.
- Where possible, define and use group synchronizations and data replication.

Programming Paradigms



Methods for multi-threading

Which one should we prefer?

1. POSIX: low-level
   - Complete
2. Java Threads: integrated in the language
   - Complete, although some things need ‘dirty’ solutions
     - For example: allow multiple synchronized methods of an object to be executed simultaneously.
3. OpenMP (and others): high-level
   - Incomplete, you can’t program everything you want…
4. OpenCL
   - For fine-grain parallelism
   - For algorithms with massive inherent parallelism
   - Thread synchronization is hidden for the user!

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.

Why multithreading

- Performance (speedup):
  - Exploit parallel hardware
  - Latency hiding
    - More than 1 thread per core
  - Load balancing
    - More than 1 thread per core
  - More high-level memory available
- Convenience
  - E.g. one thread per client request
  - Background computations

Disadvantages of multi-threading

- More difficult to understand
- More difficult to debug
- Indeterminism!
  - Finding unsafe constructions through testing is difficult!

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