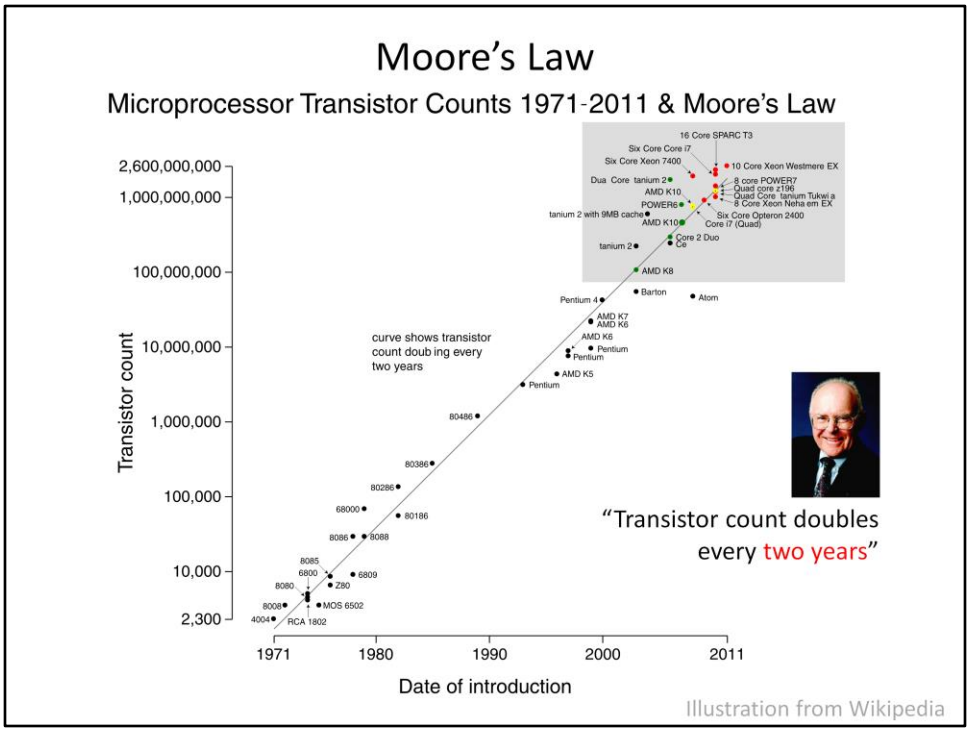
Chapter 11: Shared-memory programming using Pthreads and OpenMP

Chapter 11: outline

- Shared-memory architecture: general considerations
- Shared-memory programming
  - Thread model
  - Programming model: Pthreads
    - Thread creation and joining
    - Mutual exclusion
    - Semaphores (non Pthreads)
    - Condition variables
    - Read-write locks
    - False sharing of cache
  - Programming model: OpenMP
    - Examples

Chapter 11: outline

- Shared-memory architecture: general considerations
- Shared-memory programming
  - Thread model
  - Programming model: Pthreads
    - Thread creation and joining
    - Mutual exclusion
    - Semaphores (non Pthreads)
    - Condition variables
    - Read-write locks
    - False sharing of cache
  - Programming model: OpenMP
    - Examples



While Moore's law is still valid today (and is project to remain valid until at least 2020), the power of a single CPU chip has progressed mainly by incorporating more and more CPU cores, rather than increasing the speed of a single core.

Moore's Law

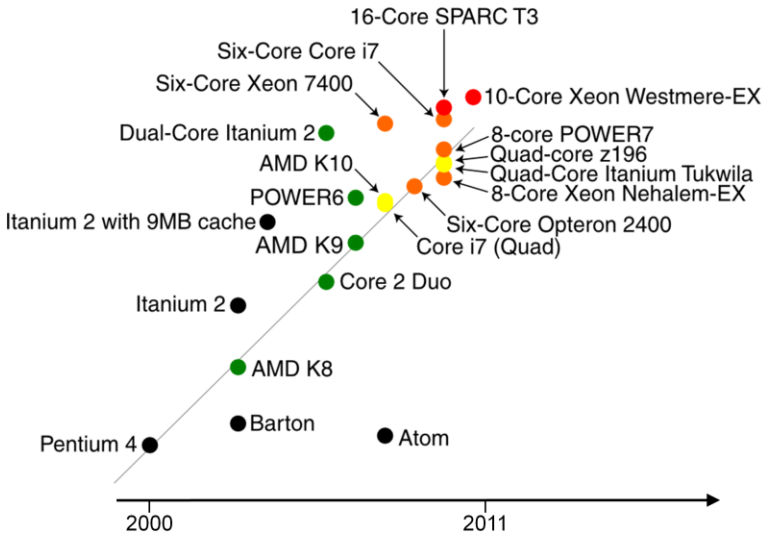
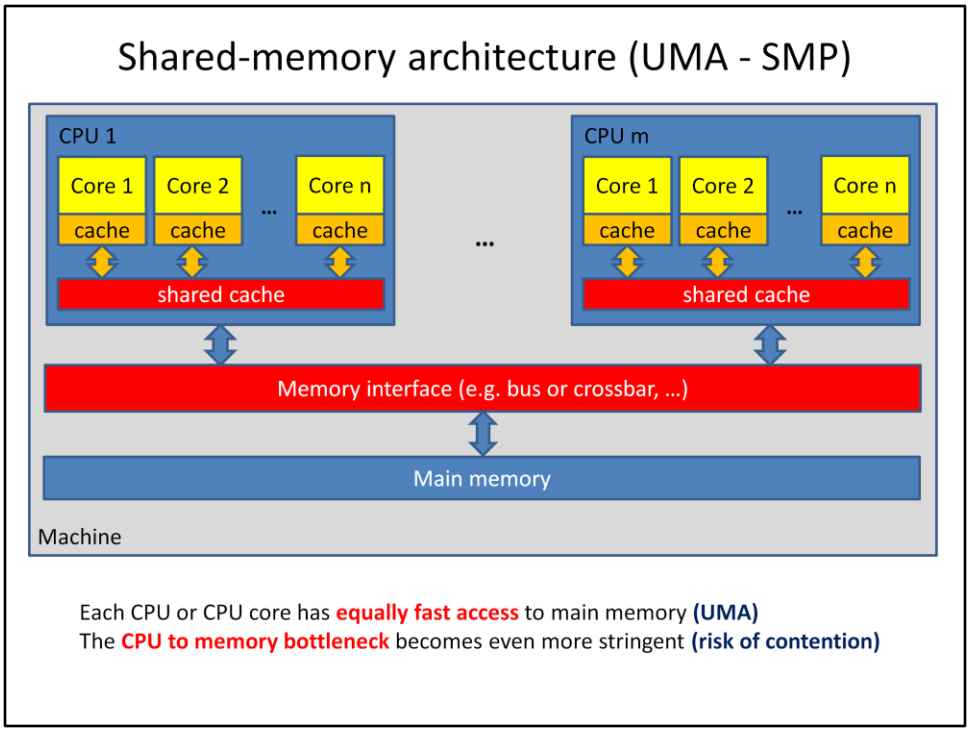


Illustration from Wikipedia



A shared-memory architecture is one where multiple processors or processor cores have access to a single pool of Random Access Memory (RAM). In case of a multi-core CPU, each core can be seen as an independent processing unit, with its own private cache memory (L1 and sometimes also the L2 cache). In order to allow for fast exchange of data between cores on a same CPU chip, the outer level cache (typically L3) is shared between the different CPU cores.

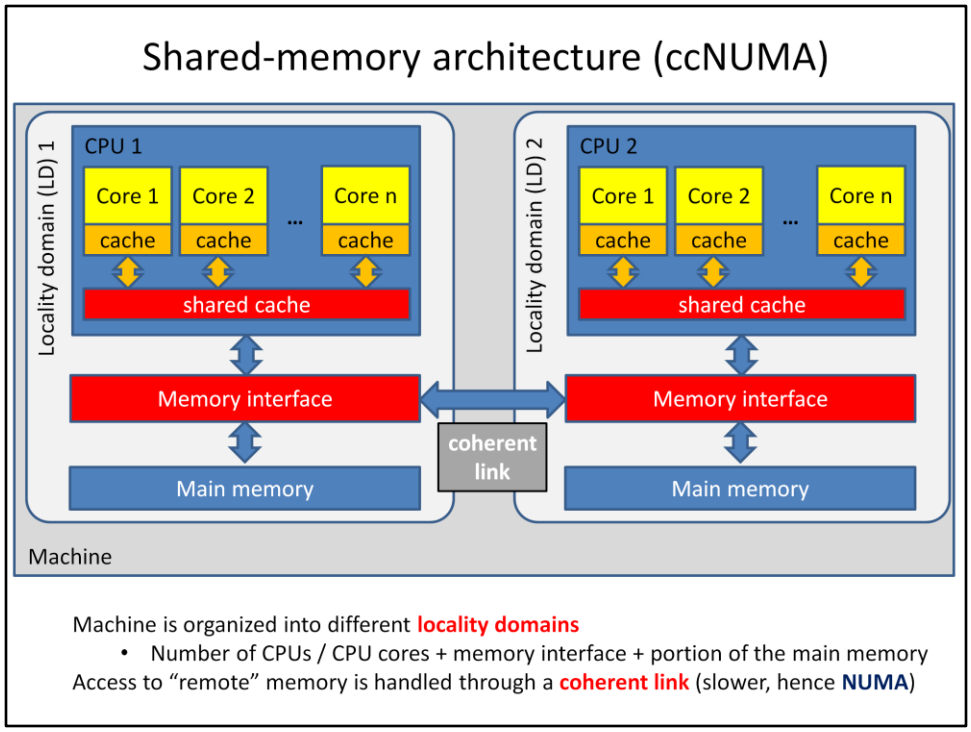
The biggest advantage is that the memory is can be used as an efficient means of transferring data between different processes or threads. Transferring data can be achieved using a **zero-copy protocol**: data written by CPU X can simply be accessed by CPU Y. This is the main difference with a distributed-memory system, where data must be explicitly copied from the local memory of machine X to the local memory of machine Y, before machine Y can access the data.

In its simplest form, the different CPUs access the main memory through a shared interconnection. If all CPUs or CPU cores have equally fast access (same latency and bandwidth) to each portion of the RAM, we say that the shared-memory machines is of the **Uniform Memory Access** (**UMA**) type. This is also called **Symmetric Multi-Processing** (**SMP**).

As was discussed in chapter 1, access to the main memory is already a bottleneck in the case of a uniprocessor (von Neumann bottleneck). This becomes even more

stringent in the case of UMA multiprocessor machines, since the available bandwidth is to be shared among the different CPUs. Also, different CPUs trying to access the same memory interface at the same time may serialize these request. This is called **contention**.

It is extremely difficult to scale SMP beyond 10-12 CPUs.



As mentioned before, UMA systems do not scale well beyond a dozen of CPUs / CPU cores. So-called **cache-coherent Non-Uniform Memory Access (ccNUMA)** machines again have access to the complete memory available in the machine (every processor sees the memory as a single addressable space), however, the access times and/or bandwidth is non uniform. In ccNUMA, a machine is organized in **locality domains;** these comprise a set of CPUs / CPU cores with their own memory interface and "**local**" memory. The different memory interfaces are connected through **coherent links**, which allow for transparent access to the memory from a different locality domain (so-called "**remote access**"). Access to memory in the same locality domain is typically faster than accessing memory in a different locality domain. From a programmer's perspective, the ccNUMA setup is completely transparent, i.e., an application does not need to worry about locality domains, as the aggregated memory appears as a single addressable space. It is the task of the operating system (OS) and hardware, to place the data into the "correct" locality domain. Usually, a **"first touch" policy** is used, i.e., the data is physically placed into the memory corresponding to the locality domain of the CPU core that first accessed that data location.

Most recent multi-socket (more than one (multi-core) CPU chip) are of the ccNUMA type. In AMD systems, the coherent link uses the HyperTransport standard. On Intel systems, Quick Path Interconnect (QPI) is used. Nowadays, memory controllers are integrated on the CPU chip.

ccNUMA systems can scale to a much higher number of CPUs / CPU cores compared to UMA / SMP systems, i.e., a few dozen up to hundreds on high end systems.

Cache coherence

Cache coherence = maintaining consistency between several copies of the same data in local caches of a shared memory system.

For example MESI protocol: four states (= 2 bits) for **every cache line**

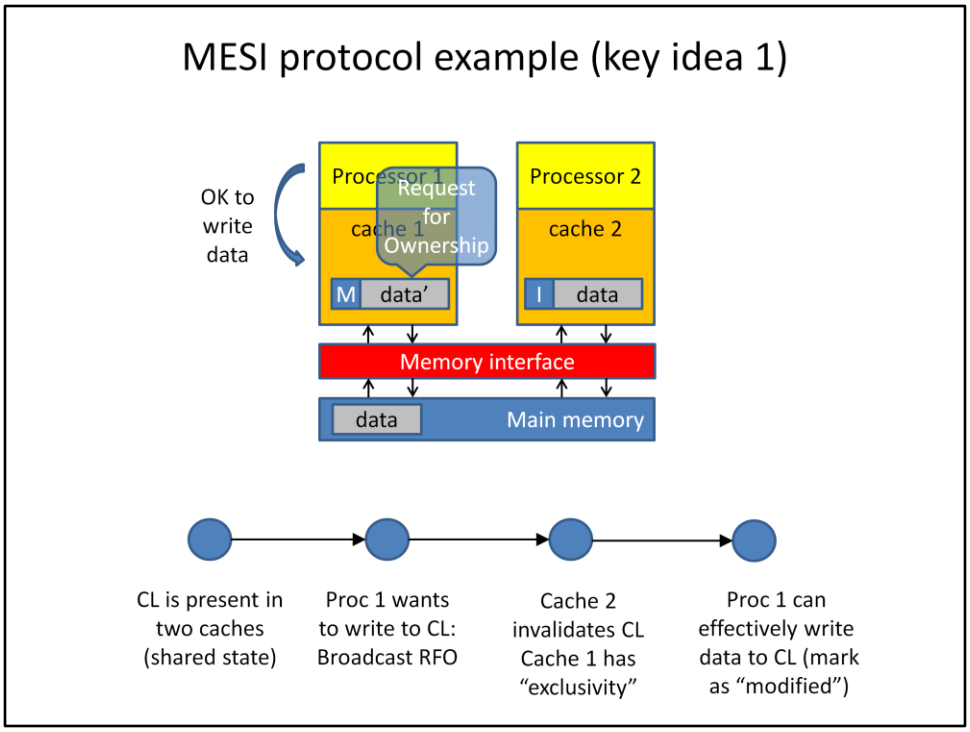
- **M (modified)**: Cache line is modified; not present in any other cache.
- **E (Exclusive)**: Cache line is not modified; not present in any other cache.
- **S (Shared)**: Cache line is not modified; may be present in other caches.
- **I (Invalid)**: Cache line is invalidated (= no longer usable)

Key ideas:

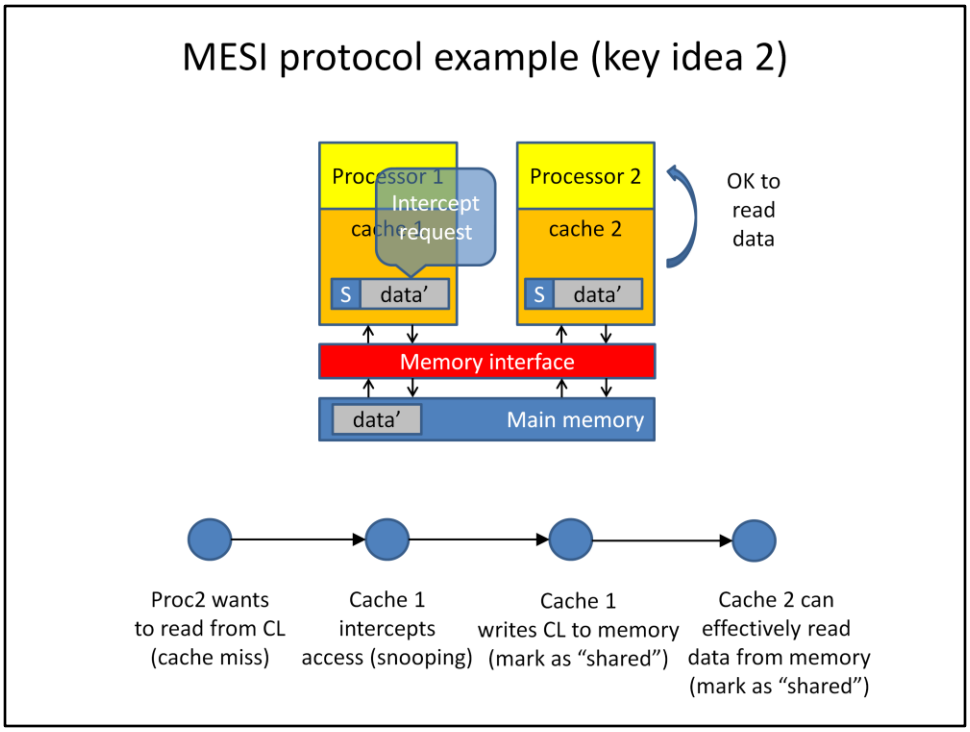
1. Writing is only possible when cache line is in "Exclusive" or "Modified" mode
   - If not, broadcast a "request for ownership" (RFO) first, invalidating other copies.
2. Cache holding "Modified" cache lines must intercept accesses by other caches
   - Modified cache line is first evicted, status is changed to "Shared"
   - Other cache can now access the updated value.

A second issue with shared-memory architectures, is that the data stored in the caches of the different CPU cores must be kept coherent. This is called **cache coherence**. For example, suppose core 1 of CPU 1 writes datum to its L1 cache. If we assume that the cache is of the write-back type, this datum is not updated in the higher-level caches or main memory. A subsequent read by e.g. core 3 of CPU 2 must return the updated value, and not the old value which is, at that moment, present in main memory. Several hardware solutions exists that keep track of which memory locations are altered by a CPU. Rapid writing to or reading from such locations can therefore severely impact performance. We will come back to this issue later.

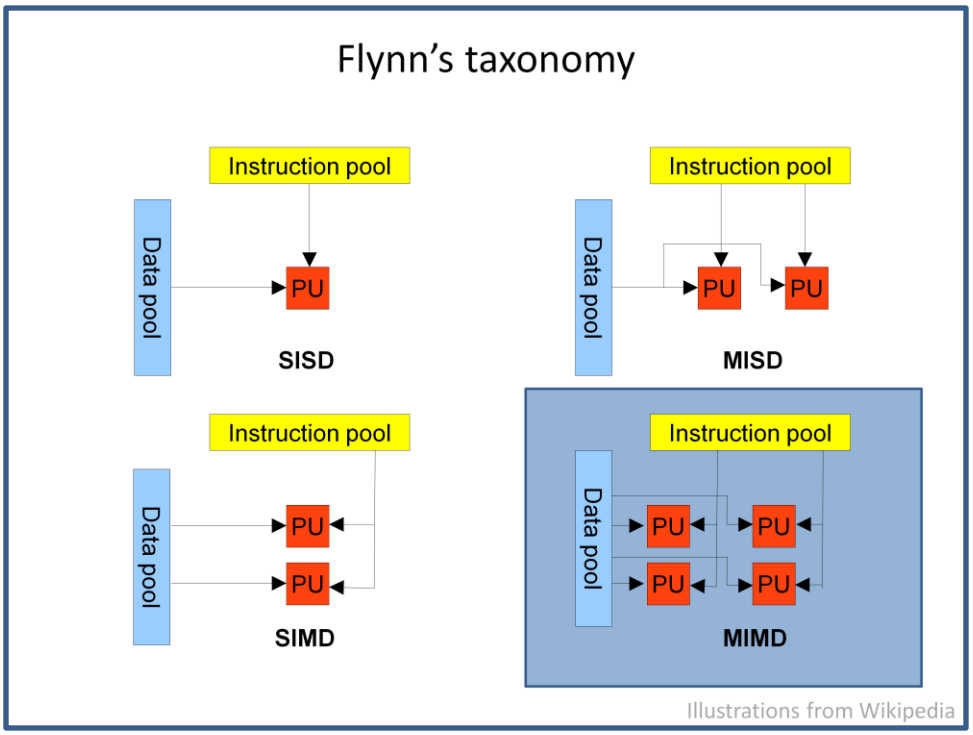
Cache coherence protocols are required for both the UMA and NUMA systems. Systems without cache coherency are excessively difficult to program and therefore no longer used.



Note: CL = cache line



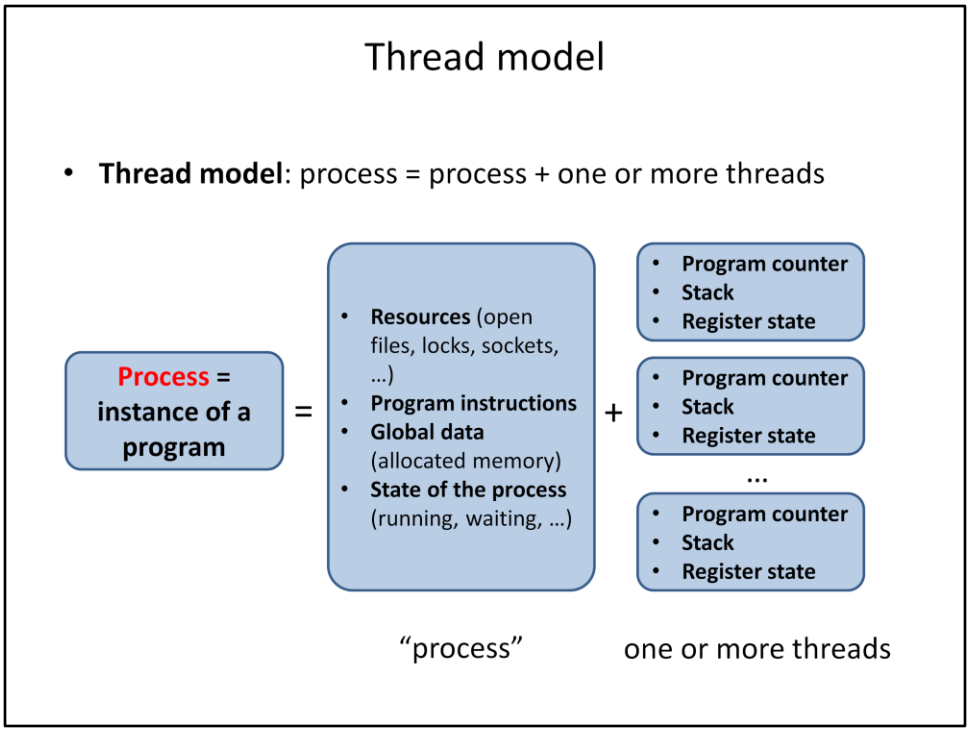
Note: CL = cache line



A shared-memory system is regarded as an MIMD architecture in Flynn's taxonomy.

Chapter 11: outline

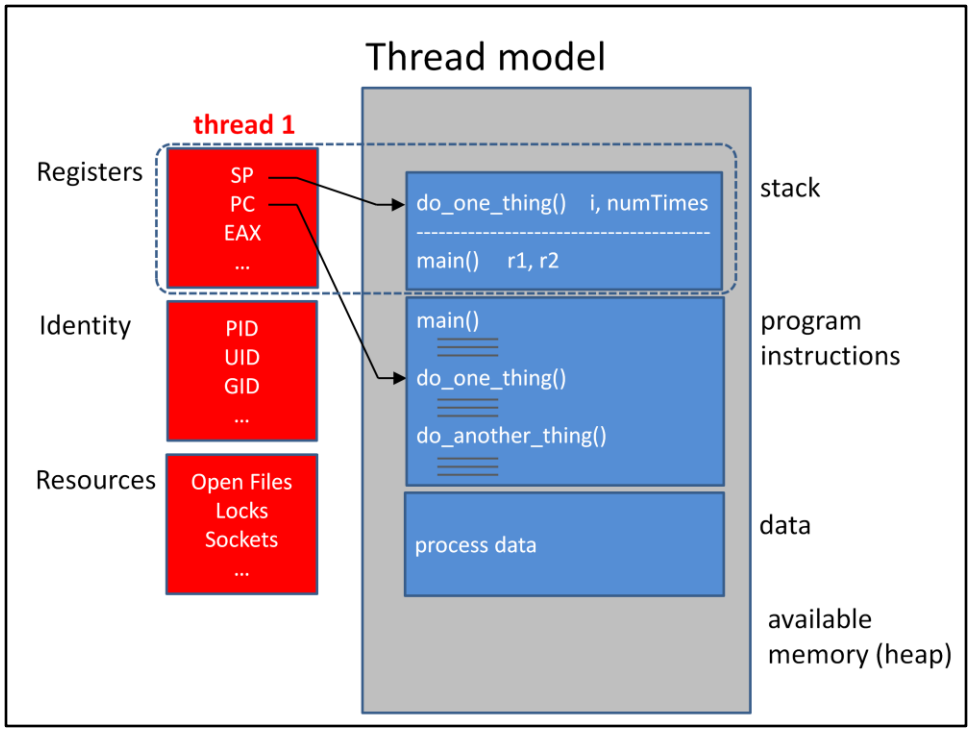
- Shared-memory architecture: general considerations
- Shared-memory programming
  - Thread model
  - Programming model: Pthreads
    - Thread creation and joining
    - Mutual exclusion
    - Semaphores (non Pthreads)
    - Condition variables
    - Read-write locks
    - False sharing of cache
  - Programming model: OpenMP
    - Examples



Program counter = points to the next instruction to be executed

Thread model

```
void do_one_thing(int *numTimes) {
    for (size_t i = 0; i < 10000; i++)
        (*numTimes)++;
}

void do_another_thing(int *numTimes) {
    for (size_t i = 0; i < 10000; i++)
        (*numTimes)++;
}

void do_wrap_up(int oneTimes, int anotherTimes) {
    cout << "Total = " << oneTimes + anotherTimes << endl;
}

void main(int *argc, char **argv) {
    int r1 =0, r2 = 0;
    do_one_thing(&r1);
    do_another_thing(&r2);
    do_wrap_up(r1, r2);
    return EXIT_SUCCESS;
}
```

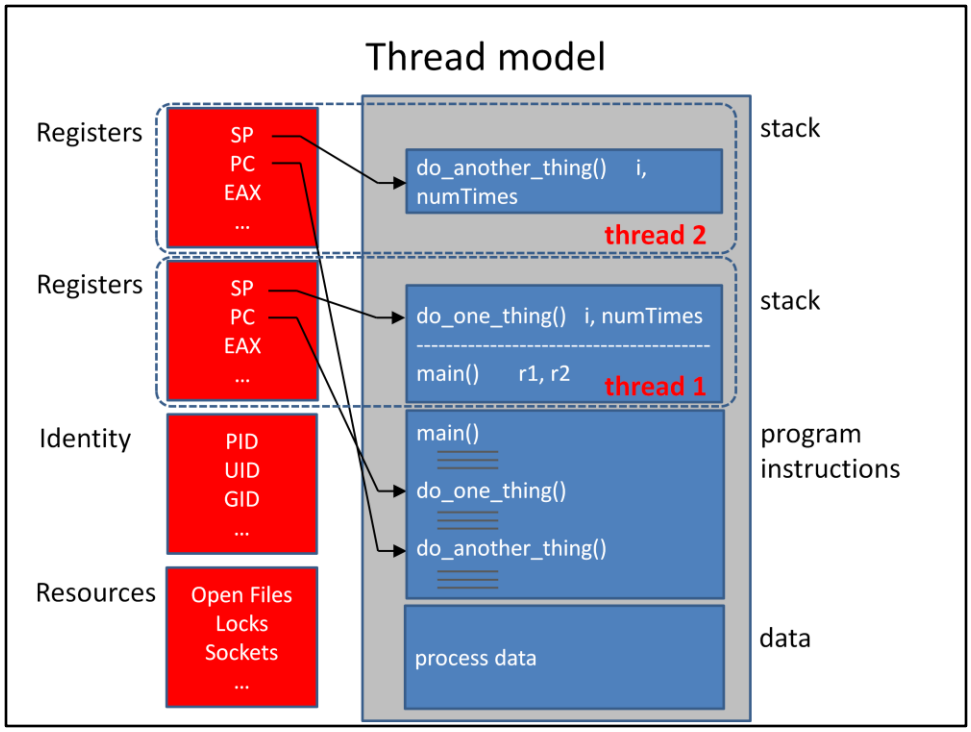


Example of a process with one thread.

SP = Stack Pointer (points to the top of the stack)
PC = Program Counter (points to the current or next instruction to be executed)
EAX = General purpose register

PID = Process identifier (unique ID to identify each process running on the OS)
UID = User identifier (unique ID per user)
GID = Group identifier (unique ID per group of users)

Thread = Register state (including program counter and stack pointer) + stack



Example of a process with two threads.

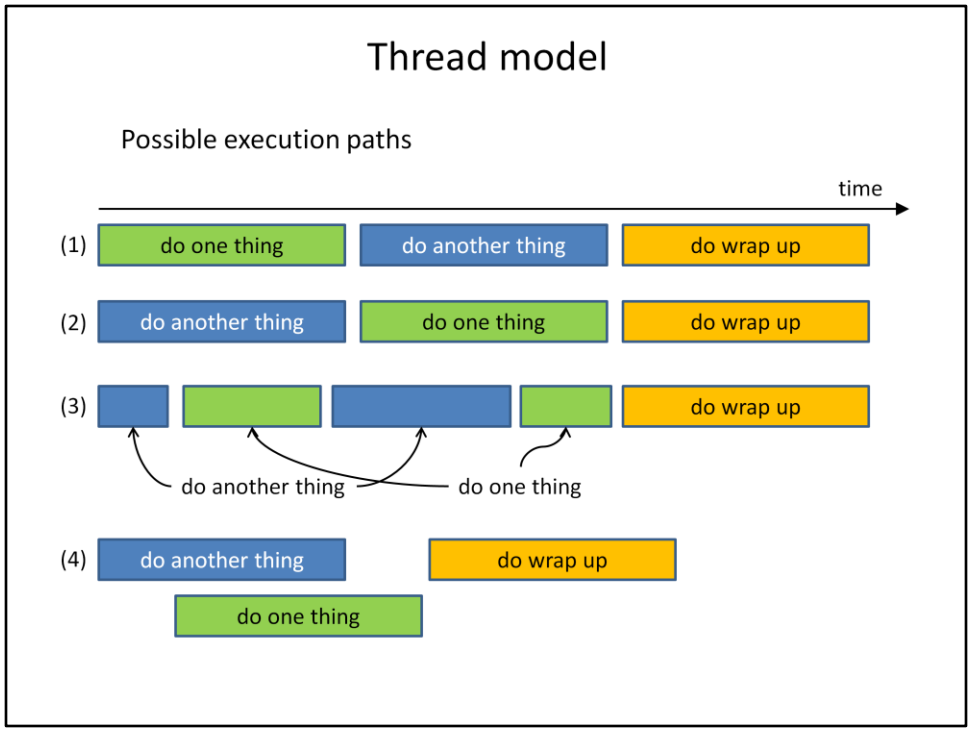
SP = Stack Pointer (points to the top of the stack)
PC = Program Counter (points to the current or next instruction to be executed)
EAX = General purpose register

PID = Process identifier (unique ID to identify each process running on the OS)
UID = User identifier (unique ID per user)
GID = Group identifier (unique ID per group of users)

Thread = Register state (including program counter and stack pointer) + stack

Thread model

- Each **thread**
  - Keeps track of its own **program counter** (PC)
  - Has its own stack containing **stack-frames** to keep track of the routines that are called and their local variables
  - Has **full read/write access** to **all** process data.
  - Has **full access to the process resources** (files, sockets, etc.)
- Scheduling of threads (by the OS)
  - On a **single CPU**: time-division multiplexing
    - Each thread gets a small amount of time before the next one is considered (e.g. round-robin RR).
    - Appears to the user that both threads are progressing simultaneously.
    - Switch of threads is called **context switch**
  - On **multiple CPUs** / CPU cores
    - Threads are scheduled on different CPU / CPU cores
    - Reduction of runtime



In (1) and (2), the order of do_one_thing() and do_another_thing() is interchanged. This will not change the outcome of the do_wrap_up routine.
In (3), so-called time division multiplexing is used to allot a certain amount of time to each of the threads before the other is again considered. On a single processor, it will appear that both threads are progressing simultaneously.
In (4), the execution of do_one_thing and do_another_thing overlaps by scheduling the different threads on different CPUs, thus exploiting parallelism inside the program.

Chapter 11: outline

- Shared-memory architecture: hardware considerations
- Shared-memory programming
  - Thread model
  - Programming model: Pthreads
    - Thread creation and joining
    - Mutual exclusion
    - Semaphores (non Pthreads)
    - Condition variables
    - Read-write locks
    - False sharing of cache
  - Programming model: OpenMP
    - Examples

Programming model: Pthreads

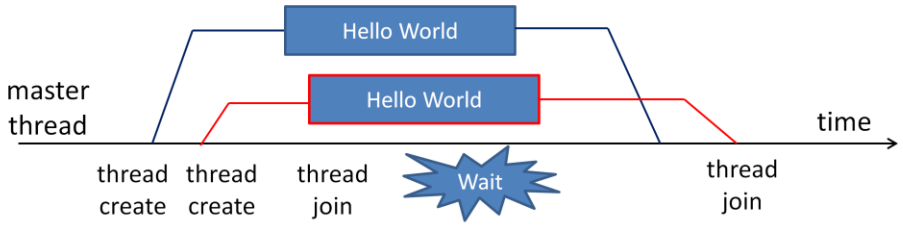
- Pthreads = **POSIX threads**
- Specifies a number of routines to
  - **Create threads** (thread spawning or forking)
  - **Synchronize** threads
    - Sharing process resources
    - Communication: reading and writing to memory location
    - Scheduling
  - **Query** the number of threads and own identifier
  - **Terminate** thread execution
- **Compiling** pthreads programs:
  - g++ input.cpp –o output –lpthread
  - icpc input.cpp –o output –pthread
- **Running** a pthreads program:
  - ./output

Hello world in Pthreads

```
#include <pthread.h>

void* helloWorld(void *)
{
    cout << "Hello world" << endl;
    return NULL;
}

int main(int argc, char** argv)
{
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, helloWorld, NULL);
    pthread_create(&thread2, NULL, helloWorld, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return EXIT_SUCCESS;
}
```

```
john@doe ~]$ ./helloWorld
Hello world
Hello world
```

Hello World in Pthreads



- `int pthread_create(pthread_t *handle, const pthread_attr_t *attr, void*(*routine)(void*), void *arg)`
  - handle = unique identifier to the thread that is created
  - attr = characteristics of the thread to create (input), can be NULL (= default)
  - routine = pointer to the start routine to execute in the newly created thread
  - arg = pointer to the routine parameters, can be NULL (= no arguments)
  - returns zero (success) or non-zero (failed)
- `void pthread_join(pthread_t handle, void **value_ptr)`
  - blocks (waits) until a thread with specific handle terminates
  - return arguments of routine specified in pthread_create

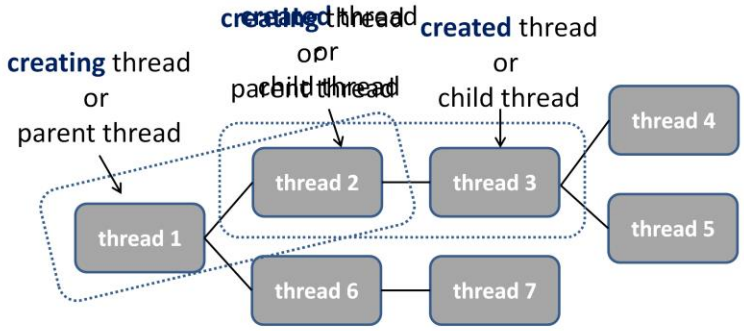
Hello world with thread ID discovery

```
#include <pthread.h>

void* helloWorld(void *arg)
{
    int ID = *((int*)arg);
    cout << "Hello world from thread " << ID << endl;
    return NULL;
}

int main(int argc, char** argv)
{
    pthread_t thread1, thread2;
    int arg1 = 0, arg2 = 1;

    pthread_create(&thread1, NULL, helloWorld, (void*)&arg1);
    pthread_create(&thread2, NULL, helloWorld, (void*)&arg2);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return EXIT_SUCCESS;
}
```

```
john@doe ~]$ ./helloWorldID
Hello world from thread 1
Hello world from thread 0
```

Note that the output in which the treads write “Hello world” to the screen is undetermined as there is no synchronization between threads.

Thread hierarchy

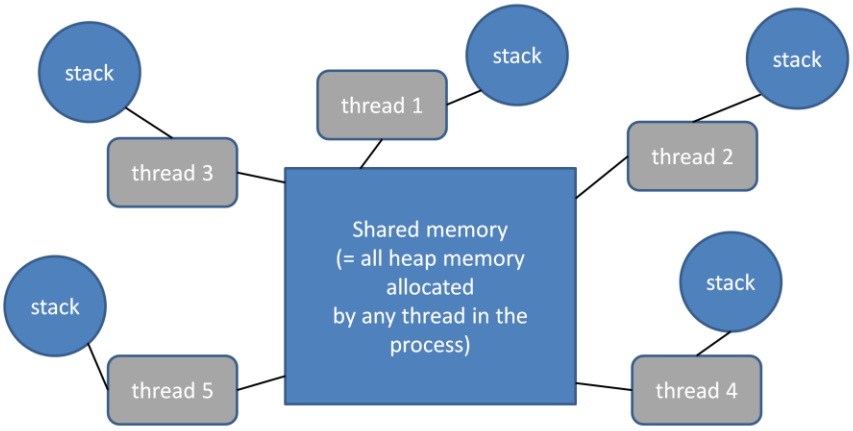


No limitation on **thread hierarchy**: any thread can create (or "spawn") new threads.

Passing arguments to start routine

- Thread **start routine**
  - Can be different for every thread
  - **Input arguments** passed through a pointer (void *)
    - Can point to a **stack** memory location
    - Can point to a **heap** memory location
    - Multiple arguments need to be packed in a class or structure
  - **Return arguments** passed through a pointer (void *)
    - Can point to a **heap** memory location
      - Allocated by the thread itself ? (bad practice – separation of concern).
      - Allocated by the creating thread (pass memory address as input argument).
    - Can point to a **stack** memory location of **creating thread**
    - Can **NOT** point to **stack** memory location of the **created thread**

Shared-memory model with Pthreads



- Threads have **full read/write access** to all memory that is owned by the process, regardless of the thread that allocated this memory.
- That includes the **stack memory** of another thread!

Multithreading versus multiprocessing

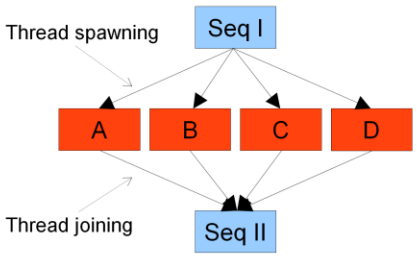
Multi-threading

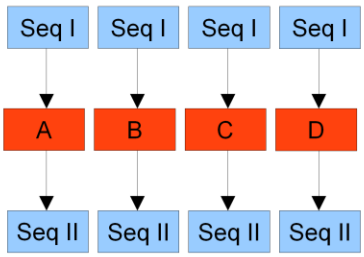
Single process
Shared memory address space
Protect data against simultaneous writing
Limited to a single machine
E.g. Pthreads, CILK, openMP, etc.

Message passing

⟺ Multiple processes
⟺ Separated memory address space
⟺ Explicitly communicate everything

⟺ Multiple machines possible
⟺ E.g. MPI, Unified Parallel C, PVM





Matrix-vector multiplication

Calculate **y = Ax**, with **A** = m x n matrix, **x** = n x 1 vector, **y** = m x 1 vector

$$y(i) = \sum_{j=1}^{n} A(i,j) * x(j) \qquad \forall\, i = 1 \ldots m$$

sequential algorithm

```
for i=1,m
  y(i)=0
  for j=1,n
    y(i)=y(i)+A(i,j)*x(j)
  endfor
endfor
```

- Assume **T** different threads
- Partition the work of the **outer loop** among different threads:
  - Thread 1 computes values **y[1:m/T-1]**
  - Thread 2 computes values **y[m/T:2m/T-1]**
  - …
  - Thread T computes values **y[m(T-1)/T:m]**

Matrix-vector multiplication

Multithreaded algorithm (A, x, y, m, n, numThreads are **global** variables)

```
void* matvecThreaded(void *arg)
{
       long threadID = (long)arg;
       int start = threadID * m / numThreads;
       int end = (threadID + 1) * m / numThreads;

       for (int i = start; i < end; i++) {
              double temp = 0;
              for (int j = 0; j < n; j++)
                     temp += A[i*n+j]*x[j];
              y[i] = temp;
       }
}
```

Multiple threads read from the same memory location!
This is **allowed**!

Example: computation of π

Use following series to approximate π

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots + (-1)^n \frac{1}{2n+1} + \ldots\right)$$

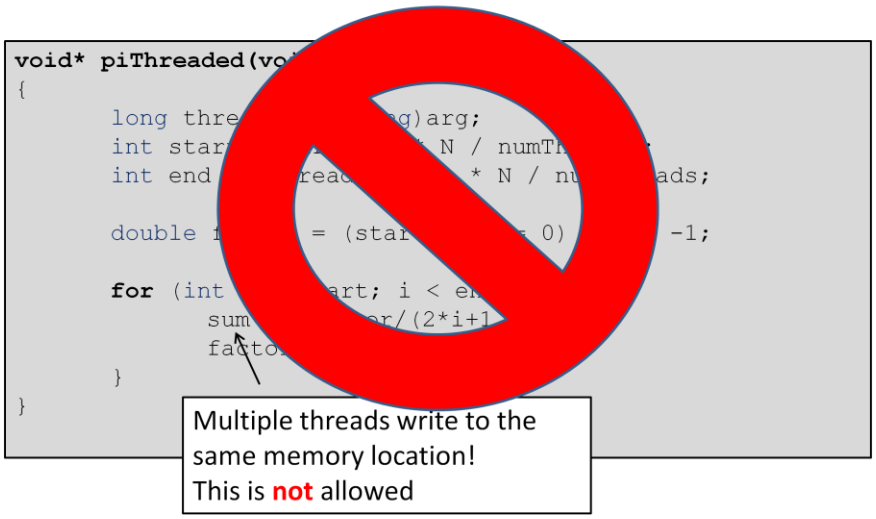
sequential algorithm

```
double factor = 1;
double sum = 0;
for i=0,N
        sum = sum + factor/(2*i+1);
        factor = -factor;
endfor
pi = 4 * sum;
```

- Multithreaded algorithm:
  - Assume **T** different threads
  - Partition the work of the **for loop** among different threads

Computation of π

Multithreaded algorithm (sum, N, numThreads are global variables)



Adding a number to a certain variable in memory is not thread-safe as it involves multiple steps (reading the original value, performing the summation, writing back the result). Each of these steps can also use pipelining and hence, the entire operation will require multiple clock instructions. If other threads write to this memory location while another thread is computing an updated value, this will result in data-corruption (incorrect results). The ultimate outcome will depend on the precise order in which threads are writing to this memory location. This is called a **race condition**.

Critical sections

Assume two threads **simultaneously** execute following code
(variable x is **shared**, y is a **local** (private) variable)

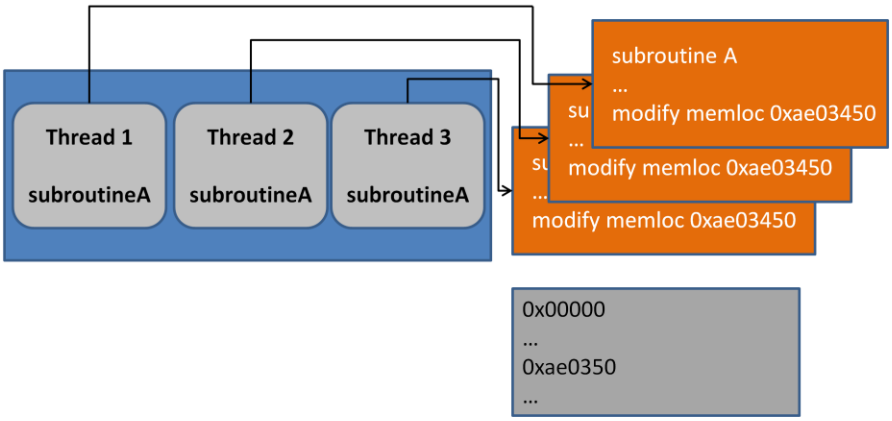
critical section

```
y = local_compute(threadID);
x = x + y;
```

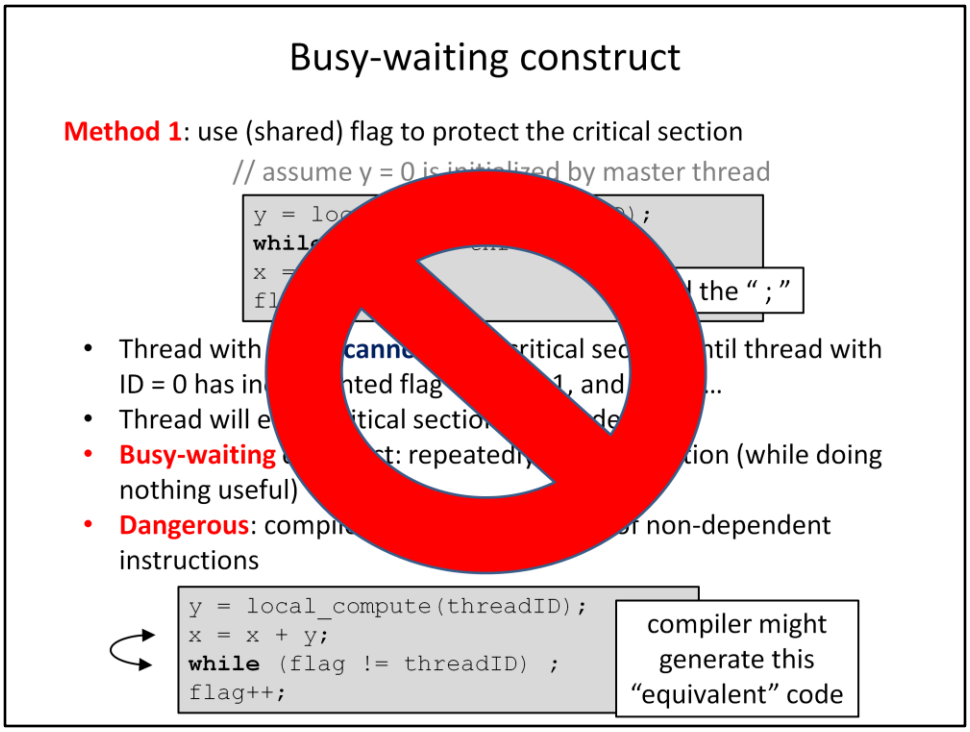
Outcome of x can depend on the order in which instructions are executed by the threads **= race condition**

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call local_compute(0) returns 1	Started by main thread
3	Assign y = 1	Call local_compute(1) returns 2
4	Put x = 0 and y = 1 into registers	Assign y = 2
5	Add 0 and 1	Put x = 0 and y = 2 into registers
6	Store 1 in memory location x	Add 0 and 2
7		Store 2 in memory location x

Data integrity in multithreaded programs



We need to be able to **synchronize threads** such that memory locations are **protected against simultaneous writing**.



Busy-wait constructions are only useful to illustrate certain concepts, but should **never** be used in real code.

Mutexes

Method 2: use Pthreads **mutex** concept

- Mutex = "mutual exclusion", can be **locked** or **unlocked**
- Only a **single thread** can acquire a lock on a mutex
- Mutex **can only be unlock** by thread that **holds the lock**
- Used to **protect critical sections** in code

- `int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t *attr);`
  - Used to initialize a mutex (call this before using the mutex)
  - attr = characteristics of the mutex to initialize (input), can be NULL (= default)
- `void pthread_mutex_destroy(pthread_mutex_t* mutex);`
  - Destroy a mutex (call this when mutex will no longer be used)
- `int pthread_mutex_lock(pthread_mutex_t* mutex);`
  - If mutex is unlocked, calling thread gets a lock on the mutex and continues; if mutex is already locked (by another thread), this function blocks
- `void pthread_mutex_unlock(pthread_mutex_t* mutex);`
  - Unlock the mutex (can only be done by a thread that holds a lock on the mutex).

Computation of π with mutexes

Multi-threaded code (sum, N, numThreads, mutex are global variables)

```
void* piThreaded(void *arg)
{
        long threadID = (long)arg;
        int start = threadID * N / numThreads;
        int end = (threadID + 1) * N / numThreads;

        double factor = (start % 2 == 0) ? 1 : -1;

        for (int i=start; i < end; i++) {
                pthread_mutex_lock(&mutex);
                sum += factor/(2*i+1.0);
                pthread_mutex_unlock(&mutex);
                factor = -factor;
        }
}
```

Code is **thread- safe**, but completely serialized

As most CPU cycles will be spent during the floating point operations in the for-loop, the use of a mutex within this loop will result in a serialization of the code.

Improved computation of π with mutexes

Improved code (sum, N, numThreads, mutex are global variables)

```
void* piThreaded(void *arg)
{
       long threadID = (long)arg;
       int start = threadID * N / numThreads;
       int end = (threadID + 1) * N / numThreads;

       double factor = (start % 2 == 0) ? 1 : -1;
       double my_sum = 0.0;

       for (int i = start; i < end; i++) {
              my_sum += factor/(2*i+1.0);
              factor = -factor;
       }

       pthread_mutex_lock(&mutex);
       sum += my_sum;
       pthread_mutex_unlock(&mutex);
}
```

Code is **thread- safe** and parallel

Semaphore synchronization

At this point, we know how to protect a critical section through a mutex
However, order of entry of threads in mutex is **left to chance...**

```
pthread_mutex_lock(&mutex);
sum += my_sum;
pthread_mutex_unlock(&mutex);
```

OK because addition is commutative

What if we were **multiplying matrices?** (non-commutative)

Need for additional synchronization mechanisms

Semaphores can provide for such functionality

- **Semaphores** take **unsigned** integer values
- Value of **zero (0)** corresponds to **locked** semaphore
- **Non-zero** value corresponds to **unlocked** semaphore
- Not a part of pthreads… but condition variables are (see later)

Semaphore synchronization

```
#include <semaphore.h>
```

- `int sem_init(sem_t* semaphore, int shared, unsigned int init_value);`
  - Used to initialize a semaphore (call this prior to using the semaphore)
  - Non-zero values for shared denote semaphore is shared between processes
- `int sem_destroy(sem_t* semaphore);`
  - Destroy a semaphore (call this when semaphore will no longer be used)
- `int sem_post(sem_t* semaphore);`
  - Increments the semaphore
- `int sem_wait(sem_t* semaphore);`
  - Decrements the semaphore if its value is greater than zero; thread will block otherwise (value is zero) until the value is again greater than zero.

Example 1: "Send" messages in a ring

- Thread 0 writes a message in a buffer and signals thread 1
- Thread 1 writes a message in a buffer and signals thread 2
- …

Note the #include <semaphore.h>. Semaphores are not part of pthreads.

Example 1: Send messages in a ring

First attempt (numThreads, messages are global variables;
messages[i] are initialized to NULL by master thread)

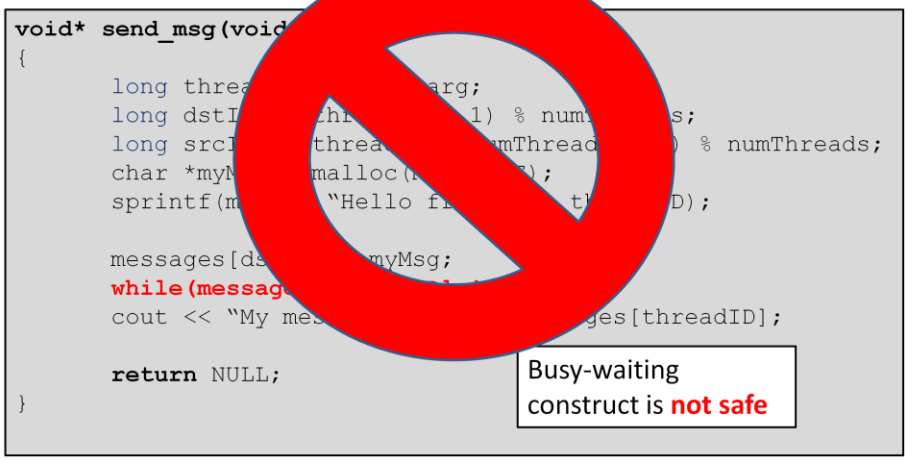
```
void* send_msg(void *
{
        long thread          g)arg;
        long dstID           + 1) % nu      ds;
        long srcI      chr       numThread      ) % numThreads;
        char *myM      mallo          ZE);
        sprintf(m       "Hello          ", th       D);

        messages[d          = myMsg;
        if (message          dID] != NU
                cout <                          essages[threadID];
        else
                cout << "No M          received" << endl;

        return NULL;
}
```

No synchronization, some threads will not "receive" a message

Assume messages is an array containing char* pointers, that has been initialized to NULL by the master thread

Example 1: Send messages in a ring

Second attempt (numThreads, messages are global variables
messages[i] are initialized to NULL by master thread)



Assume messages is an array containing char* pointers, that has been initialized to NULL by the master thread

Example 1: Send messages in a ring

Third attempt (semaphores were initialized to 0 (locked) by master thread)

```
void* send_msg(void *arg)
{
        long myID = (long)arg;
        long dstID = (myID + 1) % numThreads;
        long srcID = (myID + numThreads - 1) % numThreads;
        char *myMsg = malloc(MAX_SIZE);
        sprintf(myMsg, "Hello from %d", threadID);

        messages[dstID] = myMsg;
        sem_post(&semaphores[dstID]);
        sem_wait(&semaphores[myID]);

        cout << "My message is " << messages[threadID];

        return NULL;
}
```

Code is correct and safe !

Assume messages is an array containing char* pointers, that has been initialized to NULL by the master thread.

Example 2: dining philosophers problem

Five philosophers sit at a table

- They alternate between **eating and thinking**

```
think();
get_forks();
eat();
put_forks();
```

- They need **two forks** (left and right) in order to be able to eat.
- There are only five forks (each one is **shared by two philosophers**).

From: "The little book of semaphores":

The Dining Philosophers Problem was proposed by Dijkstra in 1965. It appears in a number of variations, but the standard features are a table with five plates, five forks (or chopsticks) and a big bowl of spaghetti. Five philosophers, who represent threads, come
to the table and execute the following loop

```
While (true) {
   think();
   get_forks();
   eat();
   put_down_forks();
}
```

The forks represent resources that the threads have to hold exclusively in order to make progress. The thing that makes the problem interesting, unrealistic, and unsanitary, is that the philosophers need two forks to eat, so a hungry philosopher might have to wait for a neighbor to put down a fork.

Example 2: dining philosophers problem

Rules:

- Only one philosopher can hold a fork at a time (mutex)
- **Deadlocks** should be impossible
- **Starvation** should be impossible (all philosophers eventually get to eat)
- It should be possible for **more than one** philosopher to eat at a given time.

Q: Derive a **protocol** for the philosophers such that the above rules are satisfied

Example 2: dining philosophers

First try for a protocol:

- **think** until hungry
- **wait** until the left fork is available; when it is, pick it up;
- **wait** until the right fork is available; when it is, pick it up;
- when both forks are held, **eat** until satisfied;
- then, put the right fork down;
- then, put the left fork down;

This protocol can lead to a **deadlock** if each philosopher has e.g. picked up the left fork

Example 2: dining philosophers

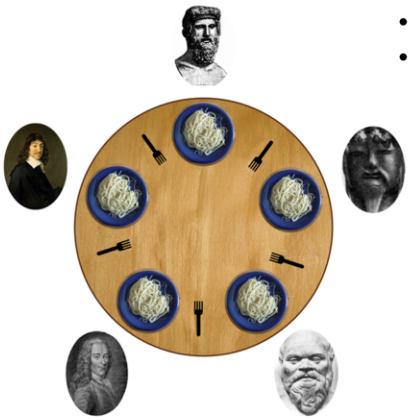
Second try for a protocol:

- **think** until hungry
- **wait** until the left fork is available; when it is, pick it up;
- **wait** until the right fork is available; when it is, pick it up;
  if right fork remains unavailable for some time, put the left fork down again and wait for some time
- when both forks are held, **eat** for a fixed amount of time;
- then, put the right fork down;
- then, put the left fork down;

This protocol can lead to a **livelock** if all philosophers are simultaneously picking up and putting down forks

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphonse moves to his right, while Gaston moves to his left. They're still blocking each other, so...

Example 2: dining philosophers problem



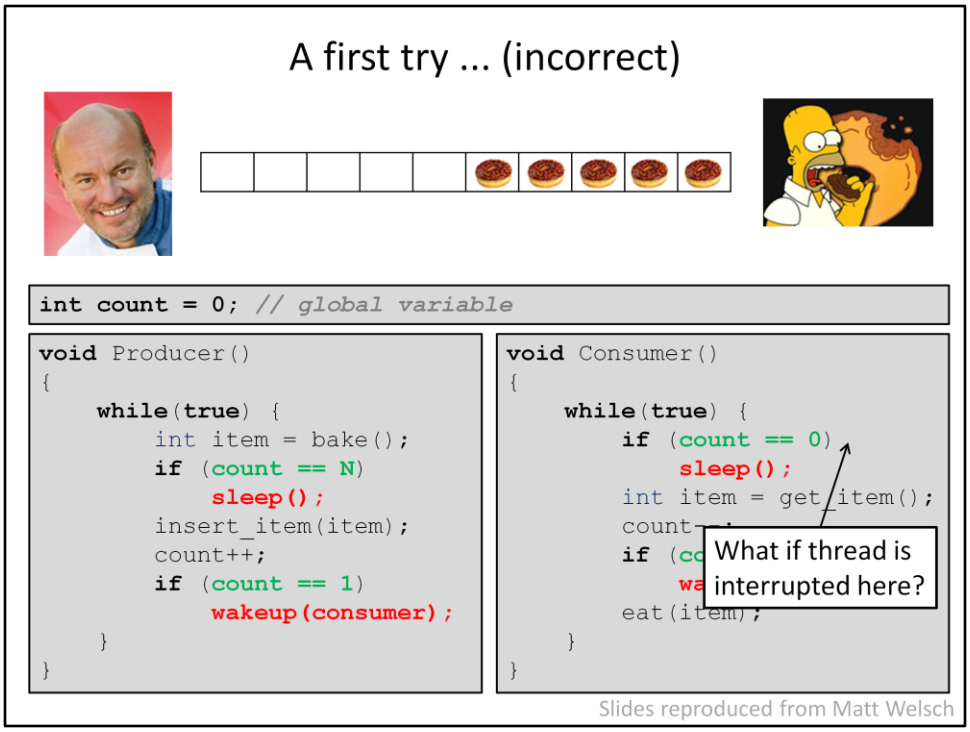
Overcoming the deadlock problem:

- Source of the problem = symmetry
- Breaking this symmetry breaks the deadlock:
  - 1) Allow a maximum of only four philosophers to pick up a fork at the same time: easy to implement using semaphores **(exercise)**.
  - 2) Certain philosophers pick up the **left fork first**, others pick up the **right fork first**
  - 3) Introduce **arbitrage** (e.g. a waiter to whom the philosophers have to ask permission to pick up a fork first.

Example 3: producer - consumer code

Producer creates items and **pushes** them into the buffer
Producer needs to **wait** when the buffer is **full**
Consumer **pulls** items from the buffer
Consumer needs to **wait** when the buffer is **empty**

Slides reproduced from Matt Welsch



The problem with this solution is that it contains a race condition that can lead to a deadlock. Consider the following scenario:

1. The consumer has just read the variable **count**, noticed it's zero and is just about to move inside the **if** block.
2. Just before calling sleep, the consumer is interrupted and the producer is resumed.
3. The producer creates an item, puts it into the buffer, and increases **count**.
4. Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
5. Unfortunately the consumer wasn't yet sleeping, and the wakeup call is lost. When the consumer resumes, it goes to sleep and will never be awakened again. This is because the consumer is only awakened by the producer when **count** is equal to 1.
6. The producer will loop until the buffer is full, after which it will also go to sleep.

A solution using semaphores

```
mutex_t mutex;
semaphore fillCount = 0;    // number of occupied places
semaphore emptyCount = N;   // number of free places
```

```
void Producer()
{
    while(true) {
        int item = bake();
        sem_wait(&emptyCount);
        mutex_lock(&mutex);
        insert_item(item);
        count++;
        mutex_unlock(&mutex);
        sem_post(&fillCount);
    }
}
```

```
void Consumer()
{
    while(true) {
        sem_wait(&fillCount);
        mutex_lock(&mutex);
        int item = get_item();
        count--;
        mutex_unlock(&mutex);
        sem_post(&emptyCount);
        eat(item);
    }
}
```

Slides reproduced from Matt Welsch

FillCount is the number of items already in the buffer and available to be read, while **emptyCount** is the number of available spaces in the buffer where items could be written. **fillCount** is incremented and **emptyCount** decremented when a new item is put into the buffer. If the producer tries to decrement **emptyCount** when its value is zero, the producer is put to sleep. The next time an item is consumed, **emptyCount** is incremented and the producer wakes up.

Condition variables

- Most **blocking synchronization** looks like

```
if (a certain condition is (not) fulfilled)
    thread goes to sleep
```

- The above statement needs to be **atomic**, i.e. condition cannot change before going to sleep
- **Semaphores** implement this atomicity
  - cfr. sem_wait(...)
  - But… only condition that is allowed is a simple counter
- **Condition variables (CV)** allow programmers to define their own condition; a mutex is integrated with the CV to make the condition evaluation atomic.

Condition variables

- Typical use of condition variables:

```
lock mutex
if (condition has (not) occurred) {
    signal (= wake) one or more (sleeping) thread(s);
} else {
    block (= sleep) + unlock the mutex
}
unlock mutex
```

- Mutex is protecting both the signal and block functions
  - Thread cannot be wakened before it actually went to sleep
- The "**block + unlock the mutex**" is an atomic operation
  - When another thread acquires the mutex, the original thread is guaranteed to be in "blocking" state.
- "Block" function needs to be called with mutex locked, "signal" function can also be called with mutex unlocked.

Condition variables syntax

- `int pthread_cond_init(pthread_cond_t* cond_var, const pthread_condattr_t* attr);`
  - Used to initialize a condition variable (call this prior to using the CV)
  - attr = attributes of the condition variable (NULL = defaults)
- `int pthread_cond_destroy(pthread_cond_t* cond_var);`
  - Destroy a condition variable (call this when CV will no longer be used)
- `int pthread_cond_wait(pthread_cond_t* cond_var, pthread_mutex_t* mutex);`
  - This function needs to be called with the mutex locked. This function with atomically unlock the mutex and put the thread to sleep, until awakened by a signal
- `int pthread_cond_signal(pthread_cond_t* cond_var);`
  - Unblock **one** of the threads that has blocked on the condition variable
- `int pthread_cond_broadcast(pthread_cond_t* cond_var);`
  - Unblock **all** of the threads that has blocked on the condition variable

Example: barrier synchronization

Example: create a multi-threaded barrier synchronization

```
pthread_mutex_lock(&mutex);
counter++;
if (counter == numThreads) {
    counter = 0;
    pthread_cond_broadcast(&cond_var);
} else {
    while (pthread_cond_wait(&cond_var, &mutex) != 0);
}
pthread_mutex_unlock(&mutex);
```

Return value of pthread_cond_wait is zero when thread is unblocked by a call to pthread_cond_signal

pthread_cond_wait can be unblocked by some event other than a call to pthread_cond_singal/broadcast. In that case, the return value is non-zero.

Condition variables vs. semaphores

- **Semaphores** vs. **condition variables**
  - Semaphores integrate the condition into an unsigned int counter
  - Condition variables have an **external** condition validation, which is protected by a mutex.
- **sem_post** differs from **cond_signal**
  - signal has no effect if no other thread is waiting on the signal
  - up has the same (ultimate) effect whether or not a thread is waiting (semaphores have "memory effect").
- **sem_wait** differs from **cond_wait**
  - `cond_wait` does not check the condition; this condition is checked externally and is protected by a mutex. Condition needs to be rechecked when thread is awakened.
  - `sem_wait` only waits if the condition (counter) is zero; this condition is limited to a simply counter; there is no need to recheck this condition when thread is awakened.

Read-write locks

Problem: how do we control **access to shared data structures**?
Consider **sorted linked list** of integer elements (elements are unique)

- Three operations: member(), insert(), delete()

```
struct llNode {
    int data;
    struct llNode *next;
}
```

Example with three elements

Case study reproduced from P. Pacheco

Multithreaded linked list

`member()` function implementation:

```
bool member(int value, struct llNode* head) {
   struct llNode* curr = head;
   while (curr != NULL && curr->data < value)
       curr = curr->next;

    if (curr == NULL || curr->data > value)
        return false;
    else
        return true;
}
```

Multiple threads accessing `member()` function: **OK**

Function is said to be **thread-safe**

Case study reproduced from P. Pacheco

Multithreaded linked list

`insert()` function implementation:

```
bool insert(int value, struct llNode** head) {
    struct llNode* curr = *head, *prev = NULL, *temp;
    while (curr != NULL && curr->data < value) {
        prev = curr;
        curr = curr->next;
    }

    if (curr == NULL || curr->data > value) {
        temp = malloc(sizeof(struct llNode));
        temp->data = value;
        temp->next = curr;
        if (prev != NULL)
            prev->next = temp;
        else
            *head = temp;
        return true;
    }
    return false;  // value already in the list
}
```

Multiple threads accessing `insert()` function: **not OK**

Case study reproduced from P. Pacheco

Suppose the linked list contains two elements: 1 and 5.

Simultaneously trying to insert elements 2 and 3 can result in a corrupted linked list (do this as an exercise).

Multithreaded linked list

`delete()` function implementation:

```
int delete(int value, struct llNode** head) {
    struct llNode* curr = *head, *prev = NULL;
    while (curr != NULL && curr->data < value) {
         prev = curr;
         curr = curr->next;
    }

    if (curr != NULL && curr->data == value) {
        if (prev == NULL)
               head = curr->next;
        else
               prev->next = curr->next;
        free(curr);
        return true;
    }
    return false;  // value not in the list
}
```

Multiple threads accessing `delete()` functions: **not OK**

Case study reproduced from P. Pacheco

Suppose the linked list contains four elements: 1, 3, 4 and 5.

Simultaneously trying to delete element 1 can result in free() being called twice.
Simultaneously trying to delete elements 3 and 4 can result in a corrupted linked list.

Multithreaded linked list

- **In general**:
  - **reading + reading** operations: **OK**
  - **writing + writing** operations : **not OK**
  - **reading + writing** operations : **not OK**
- **Solution 1**: use a mutex to restrict access to the list
  - But that also **restricts reading** to just one thread …
- **Solution 2**: protect each individual item with a mutex
  - Much **overhead** because of repeated locking and unlocking…
- **Solution 3**: use pthread **read-write locks**
  - Single lock (similar to a mutex), but effectively combines two locking functionalities:
    - **read-lock**: multiple threads can obtain the lock through this function but only if no other thread has obtained the lock through a write-lock
    - **write-lock**: if locked, no other threads can get a lock on either reading or writing

Read-write locks

- `int pthread_rwlock_init(pthread_rwlock_t* rwlock, const pthread_rwlockattr_t* attr);`
  - Used to initialize a read-write lock (call this prior to using the lock)
  - attr = attributes of the read-write lock (NULL = defaults)
- `int pthread_rwlock_destroy(pthread_rwlock_t* rwlock);`
  - Destroy a read-write lock (call this when rwlock will no longer be used)
- `int pthread_rwlock_rdlock(pthread_rwlock_t* rwlock);`
  - Lock the rwlock for reading. The calling thread will immediately obtain the lock, unless another thread obtained the rwlock for writing; it will block in that case.
- `int pthread_rwlock_wrlock(pthread_rwlock_t* rwlock);`
  - Lock the rwlock for writing. The calling thread will only obtain the lock if no other thread has obtained the lock (either for reading or writing); it will block in the other case
- `int pthread_rwlock_unlock (pthread_rwlock_t* rwlock);`
  - Unlock the lock (reading or writing) thread has obtained

Linked list performance

1000 initial keys; 100000 ops; 99.9% member(); 0.05% insert(); 0.05% delete()

	1 thread	2 threads	4 threads	8 threads
Single mutex	0.211	0.450	0.385	0.457
Mutex per element	1.680	5.700	3.450	2.700
Read-write locks	0.213	0.123	0.098	0.115

1000 initial keys; 100000 ops; 80% member(); 10% insert(); 10% delete()

	1 thread	2 threads	4 threads	8 threads
Single mutex	2.50	5.13	5.04	5.11
Mutex per element	12.00	29.60	17.00	12.00
Read-write locks	2.48	4.97	4.69	4.71

Case study reproduced from P. Pacheco

Cache effects

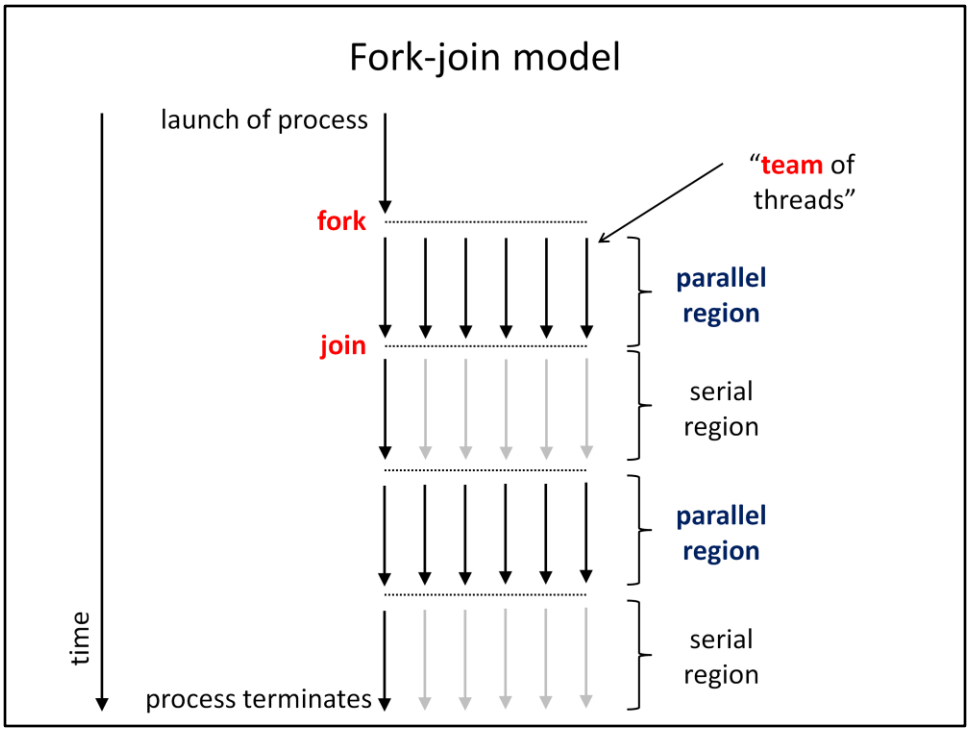
- **Shared caches** (caches shared by different cores, e.g. L3 cache)
  - A certain thread can pull data into the cache that is to be used later by another thread (**positive cache interference**)
  - A certain thread can cause the eviction of another thread's data (**negative cache interference**).
- **Private caches** (cache that is not shared between cores)
  - Can contain data also present in other private caches
  - **Cache coherence protocols** keep data consistent (e.g. MESI)
  - Different cores writing to *different* positions in the *same* cache line will trigger successive cache coherence maintenance (= **false sharing of cache**). This limits performance.
- Scheduling a thread on a different core (**context switch**)
  - Will lead to a high number of initial **cache misses**
  - OS has support to **pin** a thread to a CPU core

Chapter 11: outline

- Shared-memory architecture: hardware considerations
- Shared-memory programming
  - Thread model
  - Programming model: Pthreads
    - Thread creation and joining
    - Mutual exclusion
    - Semaphores (non Pthreads)
    - Condition variables
    - Read-write locks
    - False sharing of cache
  - Programming model: OpenMP
    - Examples

Programming model: OpenMP

- **Pthreads**
  - **Low-level** specification of thread-level parallelism
  - **Library** of functions that can be linked to a C/C++ program
- **OpenMP**
  - Higher-level constructs (**pragma directives**)
  - Instruct compiler to **automatically** parallelize certain code regions
  - Uses the fork-join model
  - Basically exists because Pthreads is considered *too difficult*
  - Allows for the incremental parallelization of an existing sequential programs
- **Alternatives**: Cilk, Intel's TBB (threaded building blocks), UPC


Fork-join model
launch of process
"team of threads"
fork
parallel region
join
serial region
parallel region
serial region
time
process terminates

Hello world with OpenMP

```
#include <omp.h>

int main(int argc, char** argv)
{
    printf("I am the master thread\n");

    # pragma omp parallel num_threads(4) {
        int myRank = omp_get_thread_num();
        int numThreads = omp_get_num_threads();
        printf("I am thread %d of %d\n", myRank, numThreads);
    }

    printf("Bye...");
    return EXIT_SUCCESS;
}
```

parallel section

```
john@doe ~]$ ./helloOpenMP
I am the master thread
I am thread 0 of 4
I am thread 2 of 4
I am thread 1 of 4
I am thread 3 of 4
Bye...
```

Basic OpenMP routines

- `int omp_get_num_threads()`
  - Get the number of thread in a parallel region
  - Returns 1 when invoked in a serial region
- `void omp_set_num_threads(int numThreads)`
  - Sets the number of threads for the subsequent parallel region
- `#pragma omp parallel num_threads(numThreads){ }`
  - Specifies a parallel region between { }
  - Specify the number of threads (optionally)
- `#pragma omp master { ... }`
  - Specifies a region within a parallel section to be executed only by the master thread

- **Compiling** OpenMP programs:
  g++ input.cpp –o output –fopenmp
- **Running** an OpenMP program:
  #export OMP_NUM_THREADS <number of default threads>
  ./output

OpenMP data scoping

- Local variables defined **before** parallel sections are visible by all threads and **shared by default**.
- Local variables defined **before** parallel sections can be made private (= each thread has its local copy)
  - `omp parallel private(varA, varB) { … }`
- Local variables defined **inside** a parallel region are private to each thread, unless variables are defined static.

Shared-memory programming with OpenMP

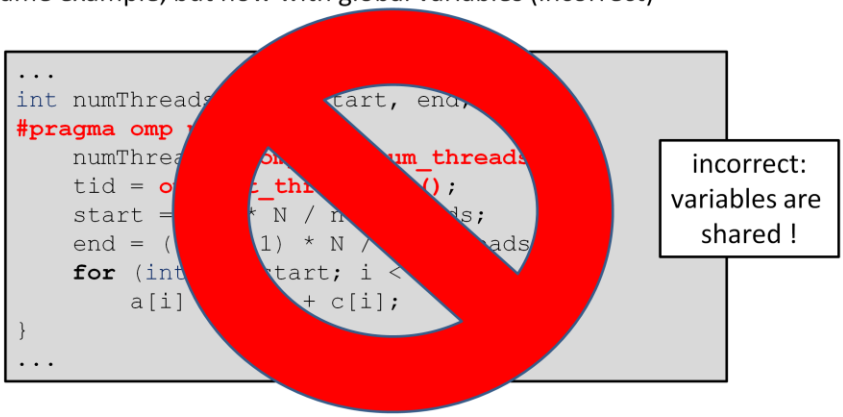
- **Example 1**: perform a[1:N] = b[1:N] + c[1:N] in parallel using OpenMP

```
...
#pragma omp parallel {
    int numThreads = omp_get_num_threads();
    int tid = omp_get_thread_num();
    int start = tid * N / numThreads;
    int end = (tid + 1) * N / numThreads;
    for (int i = start; i < end; i++)
        a[i] = b[i] + c[i];
}
...
```

variables are local to each thread

Shared-memory programming with OpenMP

- Same example, but now with global variables (incorrect)



Shared-memory programming with OpenMP

- Same example, but now with global variables privatized

```
...
int numThreads, tid, start, end;
#pragma omp parallel private(numThreads, tid, start, end) {
    numThreads = omp_get_num_threads();
    tid = omp_get_thread_num();
    start = tid * N / numThreads;
    end = (tid + 1) * N / numThreads;
    for (int i = start; i < end; i++)
        a[i] = b[i] + c[i];
}
...
```

correct: variables are privatized!

Shared-memory programming with OpenMP

- Same example, but now with “automatic” for-loop parallelization

```
...
#pragma omp parallel for {
    for (int i = 0; i < N; i++)
        a[i] = b[i] + c[i];
}
...
```

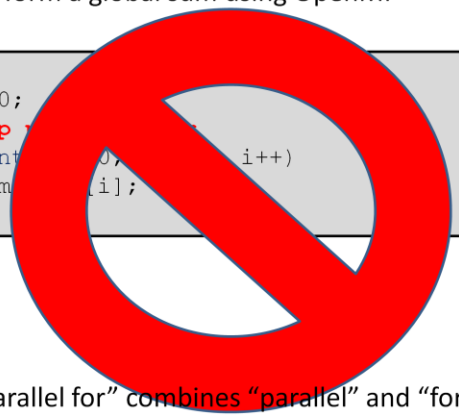
correct: automatic parallelization of for loop

- Compare this code to equivalent Pthreads code...

Shared-memory programming with OpenMP

- **Example 2**: perform a global sum using OpenMP

```
...
int sum = 0;
#pragma omp
    for (int      0;        i++)
        sum       [i];
...
```



simultaneous writing to the a shared variable

"parallel for" combines "parallel" and "for"

Shared-memory programming with OpenMP

- **Example 2**: perform a global sum using OpenMP

```
...
int sum = 0;
#pragma omp parallel {
    int lSum = 0;
    #pragma omp for
        for (int i = 0; i < N; i++)
            lSum += a[i];
    #pragma omp critical {
        sum += lSum;
    }
}
...
```

similar to a mutex: critical section can be executed by a single thread only

The end