**Lecture 2 - Handling concurrency**

**Multi-processors with shared memory**

Idea: several CPUs sharing the same RAM. (Note: by CPU, we understand the circuits that process the instruction flow and execute the arithmetic operations; a PC CPU chip contains additional circuits, as well, such as the memory cache, memory controller, etc.)

**Memory caches**

Problems: high memory latency; memory bottleneck

Solution: use per-processor cache

A diagram of a computer hardware system

Description automatically generated

New problem: ensure cache consistency (consider that one CPU modifies a memory location and, immediately afterwards, another CPU reads the same location).

Solution: cache-to-cache protocol for ensuring consistency. (Locking, cache invalidation, direct transfer between caches.) However, this means that:

* if multiple CPU access for both read and write the same memory location, the access is serialized and no speed-up results from multiple cores (moreover, there is a penalty to be paied for the cache ping-pong);
* the same happens if two variables are placed in distinct memory locations, but in the same cache line (false sharing).

Note: see [false-sharing.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/false-sharing.cpp) and play with the *alignof* argument.

**Instruction re-ordering**

In the beginning, the CPU executed instructions purely sequentially, that is, it started one instruction only after the previous one was fully completed.

However, each instruction consists in several steps (fetch the instruction from memory, decode it, compute the addresses of the operands, get the operands from memory, executing any arithmetic operation, etc) and sub-steps (a multiplication, for instance, is a complex operation and takes many clock cycles to complete). Thus, the execution of an instruction takes several clock cycles to complete.

It is possible, however, to parallelize the stages in instruction execution, for instance, to fetch the next instrunction while the previous one is being decoded. The result is a processing *pipeline*, and thus, at each moment, there are several instruction in various stages of their execution. The advantage is that the average execution time per instruction is reduced, but there is a problem if an instruction needs some results from the previous instruction before those results are ready. To solve this problem, the solution is to add wait states or to re-order instructions (so that there is enough time between dependent instructions). Both waits and re-orderings can be done either by the compiler or by the CPU itself.

The result for the programmer is that instructions can be re-ordered without the programmer knowing about that. The reordering is never allowed to change the behavior of a single thread, but can change the behavior in multi-threading contexts. Consider the following code:

bool ready = false;

int result;

Thread 1:

result = <some expression>

ready = true

Thread 2:

while(!ready) {}

use(result)

Because of re-ordering, the above code may not be correct. The compiler or the CPU can re-order the instructions in Thread 1 because the behavior of thread 1 is not change by that. However, this makes thread 2 belive the result is ready before actually being so.

**Processes and threads**

See a [C++ example](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/threads.cpp) and a [Java example](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/threads.java) with threads and performance measurement.

See also a [classical pitfall](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/MisleadingLambda.cs) regarding closures in C#, in a threading context.

A *thread* has a current instruction and a calling stack. In more details, it has the following attributes:

* The pointer to the current instruction (IP register);
* The stack of nested function calls (with the return address from each function to its caller);
* The local variables and temporaries for each active function.

At each moment, a thread can:

* *run* on one of the CPUs,
* *be suspended*, waiting for a CPU to become available,
* *sleep*, waiting for some external operation to complete.

This means that each CPU executes instructions from one thread until it either launches a blocking operation (read from a file or from network), its time slice expires, or some higher piority thread becomes runable. At that point, the operation system is invoked (by the read syscall, by the timer interrupt, or by the device driver interrupt), saves the registers of the current thread, including the *instruction pointer* (IP) and the *stack pointer* (SP), and loads the registers for the next scheduled thread. The last operation effectively restores the context of that thread and jumps to it.

It should be noted that a *context switch* (jumping from one thread to another) is quite an expensive operation, because it consists in some hundreds of instrunctions, and may invalidate a lot of the CPU cache.

Creation and termination of a thread is also expensive.

A *process* can have one or more threads executing for it. The memory and the opened files are per process.

**Mutual exclusion problem**

**The problem**

*Two threads walk into a bar. The bartender says:  
Go I don't away! want a race to get condition last like I time had.*

Consider several threads, where each of them adds a value to a shared sum. For instance, each thread processes a sale at a supermarket, and each adds the sale value to the total amount of money the supermarket has.

Since the addition itself is done in some register of the CPU, it is possible to have the following timeline:  
A diagram of a graph

Description automatically generated  
So, thread B computes the sum based on the original value of S, not the one computed by thread A, and overwrites the value computed by A. What we should have is to execute the addition either fully by A and then fully by B, or viceversa; but not overlapped.

**Atomic operations**

These are simple operations, on simple types (integers, booleans, pointers), that are guaranteed to execute atomically. They have hardware support in the CPU. They need to be coupled with *memory fences* — directives to the compiler and CPU to refrain from performing some re-orderings.

Operations:

* *increment / decrement*;
* *add / substract* a given value;
* *compare-and-exchange*: compare the atomic variable with a given value and, if equal, set it to another value; also, return the old value.

See:

* C++ 11: [atomic<T>](http://en.cppreference.com/w/cpp/atomic/atomic)
* C#: [Interlocked (static class)](https://docs.microsoft.com/en-us/dotnet/api/system.threading.interlocked?view=netframework-4.6)
* Java: [AtomicInteger](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html)

Uses:

* Simple atomic add/substract values;
* Basic block to implement spinlocks;
* Compare and exchange: compute new value, then replace current value with new value if the current value is still the old one.

**Mutexes**

A mutex can be hold by a single thread at a time. If a second thread tries to get the mutex, it waits until the mutex is released by the first thread.

A mutex can be implemented as a *spin-lock* (via atomic operations), or by going through the operating system (which puts the thread to sleep until the mutex is freed).

Mutexes are used to prevent simultaneous access to the same data.

Each mutex should have an associate invariant that holds as long as nobody holds that mutex:

* some variables are guaranteed not to change;
* some consistency conditions are guaranteed to hold.

Mutexes in various languages:

* C++ 11: [mutex](http://en.cppreference.com/w/cpp/thread/mutex): lock(), unlock(), try\_lock; also [unique\_lock<mutex>](http://en.cppreference.com/w/cpp/thread/unique_lock)
* C#: [Monitor (static class)](https://msdn.microsoft.com/en-us/library/system.threading.monitor(v=vs.110).aspx) Enter(), Exit(), or lock(){}
* Java: [Semaphore](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html) or synchronized(){}

**Invariants in single-threaded applications**

In a single threaded program, when a function begins execution, it assumes some *pre-conditions* are met. For instance:

// preconditions:

// - a, b and result are valid vectors

// - vectors a and b are sorted in increasing order

// - vector result is not an alias to either a or b

// post-conditions:

// - a, b and result are valid vectors

// - vector result contains all the elements in a and b, each having

// the multiplicity equal to the sum of its multiplicities in a and b

// - vector result is sorted

// - vectors a and b are not modified

// - no other program state is modified

void merge(vector const& a, vector const& b, vector&result);

If the pre-conditions are met, the function promises to deliver the specified *post-conditions*.

If the pre-conditions are not met, the behavior of the function is undefined (anything may happen, including crashing, corrupting other data, infinite loops, etc).

In conjunction with classes, we have the concept of a *class invariant*: a condition that is satisfied by the member data of the class, whenever no member function is in execution.

Any public member function assumes, among its pre-conditions, that the invariant of its class is satisfied. Also, any public member function promises, among its post-conditions, to satisfy the class invariant.

At a larger scale, there are various invariants satisfied by subsets of the application variables. Consider the case of a bank accounts application: an invariant would be that the account balances and history all reflect the same set of processed transactions (the balance of an account an is the sum of all transactions in the account history, and if a transaction appears on the debited account, it appears also on the credited account, and viceversa).

At the beginning of certain functions (for instance, those performing a money transfer, as above), we assume some invariant is satisfied; the same invariant shall be satisifed in the end. Then, sub-functions are invoked, concerned with sub-aspects of the computation to be done; the precise pre- and post-conditions for those functions should be part of their design; however, many bugs arise from a misunderstanding regarding those per- and post-conditions (in other words, the exact responsability of each function).

Note that sometimes the history is not kept as a physical variable in the system; nevertheless, we could think of it as if it were really there.

An implicit assumption in a single-threaded program is that nobody changes a variable unless explicit instructions for that exist in the currently executing function.

**Invariants in multi-threaded applications**

In multi-threaded applications, it is hard to know when it is safe to assume a certain invariant and when it is safe to assume that a certain variable is not modified.

This is the role of mutexes: a mutex protects certain invariants involving certain variables. When a function aquires a mutex, it can rely that:

* the invariants are satisfied at the point the mutex is aquired;
* no other thread will modify the variables before the mutex is released.

The function must re-establish the invariant before releasing the mutex.

The above also implies that, in order to modify a variable, a function must make sure it (or its callers) hold all mutexes that protect invariants concerning that variable.

**Read-write (shared) mutexes**

There are two use cases concerning the invariants:

1. a function changes some variables, it needs to ensure that the invariant holds when it begins, promises to re-establish the invariant, but it will violate the invariant during its execution. Therefore, during the execution, nobody else can be allowed to see the variables involved in the invariant.
2. a function needs to ensure that some invariant is satisfied during its execution, but it does not change any variable involving in that invariant.

A thread doing case 1 above is incompatible with any other thread accessing any of the variables involved in the invariant. A thread doing case 2 above, however, is compatible with any number of threads doing case 2 (but not with one doing 1).

For this reason, we have *read-write mutexes*, also called *shared mutexes*. Such a mutex can be locked in 2 modes:

1. *exclusive lock* or *write lock*, which is incompatible with any other thread locking the mutex;
2. *shared lock* or *read lock*, which is incompatible with any other thread locking in *exclusive* mode the same mutex, but is compatible with any number of threads holding the mutex in *shared* mode.

Caveat: the implementation of a shared mutex must deal with the following dilemma: Suppose several readers hold the mutex in shared mode, and a new (writer) thread attempts to lock it in exclusive mode. What to do if, before all the readers finish, a new reader comes in? If we allow the reader, we run the risk of starving the writer (if we have enough readers to keep at least one active one for a long time). If we deny the reader, we miss a parallelizing opportunity.

**On recursive mutexes**

A recursive mutex allows a lock operation to succeed if the mutex is already locked by the same thread. The mutex must be unlocked the same number of times it was locked.

The problem with recursive mutexes is that, if a function attempting to aquire a mutex cannot determine if the mutex is alreay locked or not, will not be able to determine if the invariant protected by the mutex holds or not immediately after the mutex is aquired. On the other hand, if the function can determine if the mutex is already locked, it has no need for a recursive mutex.