**Lecture 3 - More concurrency issues**

**Finer granularity locks**

**Fighting the bottlenecks**

A mutex is a bottleneck in the program. If multiple threads try to acquire a mutex, their actions will be serialized, thus negating the parallelism.

Reducing the contention over mutexes can be done the following ways:

* making as much of the data read-only; this can be very effective in conjunction with functional-style programming. Functional programming means (mong other things): Use pure functions - functions that depend only on the explicit parameters (not on some global state) and do not modify them. Use variables assigned only once (at the definition time) and only read afterwards.
* reducing the granularity of each mutex;
* using shared mutexes whenever reasonable.

Caveats:

* with many mutexes, it is harder to know exactly what operation requires which mutex;
* with many mutexes, it is harder to avoid deadlocks;
* with many mutexes, it the overhead of aquiring and releasing mutexes can negate the gain with increased parallelism;
* a shared lock is more expensive than an exclusive lock.

**Consistency issues**

Consider the bank accounts problem (lab 1 pb 2):

struct Account {

unsigned balance;

mutex mtx;

};

bool transfer(Account\* a, Account\* b, unsigned amount) {

a.mtx.lock();

if(a.balance < amount) {

a.mtx.unlock();

return false;

}

a.balance -= amount;

a.mtx.unlock();

// ---> what if the audit() occurs here?

b.mtx.lock();

b.balance += amount;

b.mtx.unlock();

return true;

}

An operation running on some other thread between updating the two accounts will see an inconsistent state, with money no longer in the first account and not yet in the second one; so, the audit would fail.

With using locks, the solution is to lock the second mutex before freeing the first one.

The general idea is:

1. There will be, in the code, a reference point where the transaction can be postulated to occur;
2. For every variable involved, there is a mutex that protects it between the time the ideal transaction occurs and the point(s) it is read and/or modified

struct Account {

unsigned balance;

mutex mtx;

};

bool transfer(Account\* a, Account\* b, unsigned amount) {

a.mtx.lock();

if(a.balance < amount) {

a.mtx.unlock();

return false;

}

a.balance -= amount;

// ---> the reference time for the transaction is later, but nobody can see a's balance until then

b.mtx.lock();

// ---> we can consider the whole transaction occurs here

a.mtx.unlock();

// ---> nobody can see that the balance of account b is not yet modified

b.balance += amount;

b.mtx.unlock();

// ---> as we unlock the mutex, b's balance has the right value

return true;

}

However, we've just traded one problem for another...

**Deadlocks**

A deadlock occurs when there is a cycle in the wait dependency between threads (a thread waits on another thread if the first thread tries to aquire a mutex that is already locked by the second thread).

For the account transfer example above, consider thread P executing a transfer from account A to B has aquired the mutex on A and is attempting to aquire the mutex on B. Meanwhile, thread Q executes a transfer from B to A, aquires the mutex for B, and tries to aquire mutex for A. Thread P is put on hold and cannot advance (and release A) until thread Q releases mutex B. Meanwhile, Q cannot advance (and release B) until P releases mutex A.

This can be represented as a dependency graph. Any cycle in such a dependency graph means a deadlock:  
A diagram of a diagram

Description automatically generated

A guaranteed solution to avoid deadlocks is to have a total order relation between mutexes and to allow threads to aquire mutexes only in that order (it is ok to skip mutexes, but not to go back). An special (extreme) case of this is to aquire at most one mutex at any time. This solution is, however, hard to implement if the set of mutexes to be aquired is known only after having aquired other mutexes.

For the bank accounts problem above, the solution would be to have each account have an ID, and lock first the mutex of the account with the lowest ID.

Another solution is to detect deadlocks and to refuse an aquiring attempt when a deadlock would result. This leads to more complex code (the code must threat the case when a lock operation fails) and it is possible to result in a *livelock* - when two or more threads attempt to aquire locks, give up because of a potential deadlock, restart their operations and repeat the same scenario over again.

**Exercices**

1. Design a thread-safe doubly-linked list with per-node mutex.

2. Bank transfer when accounts can be created and deleted/disabled dynamically.

**Producer-consumer communication**

**The problem**

We have a computation that depends on the result of some other computation or on some external data (from the disk, from the network, etc).

A simple solution is to use an atomic variable like in the following example:

atomic\_bool ready = false;

int result;

Producer thread:

result = <some expression>

ready = true

Consumer thread:

while(!ready) {}

use(result)

Note that the default semantic of the atomic variables prevent the compiler and the CPUs from re-ordering the instructions around the atomic variable operations.

However, the problem is that, if the consumer thread gets to the waiting loop long before the producer manages to produce the data, the consumer will busy-wait for the result. This wastes CPU time, which would be much more useful for the producer thread; the operating system cannot know that the consumer does nothing useful and schedule something else.

**Condition variables and their usage**

A *condition variable* is a mechanism, involving the operating system if needed, allowing the consumer to wait until signaled by the producer thread. Waiting consumes no CPU.

Simplifying a bit, it has 2 operations

***wait()***

puts the calling thread on wait, until the condition variable gets signaled

***notify()***

wakes up (ends waiting) any thread waiting in a wait() call at that time.

Actually, *condition variable* is a misnomer: it is simply a communication channel, rather than a variable: it has no state, and the signal passed is transient: it wakes up any consumer waiting at that time, but leaves no trace for

The previous example, re-written using condition variables would be (not entirely correct, yet):

std::mutex mtx;

bool ready = false;

int result;

std::condition\_variable cv;

Producer thread:

int local\_result = <some expression>

{

unique\_lock<mutex> lck(mtx);

result = local\_result;

ready = true;

cv.notify();

}

Consumer thread:

int local\_result;

{

unique\_lock<mutex> lck(mtx);

while(!ready) {

cv.wait();

}

local\_result = result;

}

use(local\_result);

There is a problem with the above code:

* The lock is needed because otherwise, if the producer sets *ready* to true and signals the condition variable just between the time the consumer checks the value of *ready* and the time it enters the *wait()*, then the consumer misses the wake-up call by *signal()* and will wait forever.
* On the other hand, with the lock, we get a deadlock because the consumer will sleep with the mutex locked, therefore not allowing the producer to aquire it in order to set the result and signal the condition variable.

To solve this, the mutex needs to be released for the duration of the sleep, but the release must be atomic with the sleep. The code becomes, on the consumer side:

int local\_result;

{

unique\_lock<mutex> lck(mtx);

while(!ready) {

cv.wait(lck);

}

local\_result = result;

}

use(local\_result);

That is, the *wait()* function gets the mutex holder as an argument. The mutex is release upon entering the sleep, and re-aquired before returning from *wait()*.