10. Multithreading

Ján Dugáček

November 10, 2017

Table of Contents

Motivation

- So far, our programs can run only on one CPU core
- When calculation is running, we can't use any commands to interact with it (unless the program regularly checks if something happened, react accordingly and return back to calculation, which is not practical)
- Solution exists, but it's not simple - there are large areas of computer science to deal specifically with this problem

Basics

- The simplest way to use multiple CPU cores is to run the program twice, using files to communicate
- This approach is facilitated by *pipes*, special files that are read by one process and written by another, everything read is erased
- Usually, it's more efficient to use more *threads of execution* in a single process, so that they share the address space
- Threads are usually created by launching a function on a new thread, which will run independently of the thread that launched it
- Any thread usage requires adding `-lpthread` to command line arguments

std::async

- Many tasks are so-called *embarrasingly parallel*, splittable into several independent calculations with a short synthesis afterwards
- This can be done with `std::async`, which runs a function on a different thread than the rest of the code and has a method to wait for the result and grab it

```
std::future<int> got[10];
auto sum = [] (int min, int max) {
        int result = 0;
        for (int i = min; i < max; i++) result += i;
        return result;
};
for (int i = 0; i < 10; i++)
        got [i] = std::async(std::launch::async,
                sum, i * 100000, (i + 1) * 100000);
int total = 0;
for (int i = 0; i < 10; i++)
        total += got[i].get();
```

- `future` must be included

Nondeterminism

- Threads are unpredictable, there's no way of telling if one thread will do a specific action before or after that another thread executes some section
- In the recent example, the method `get()` waits for the thread to be finished and will grab the result then, it's the only way to be sure that it will be present at the time when it's read
- During the wait, the main thread is suspended, taking no processor time before the result is present
- Creating threads, pausing them or destroying them takes a lot of time, in the order of tens of thousands of processor cycles, so each thread should do a lot of work before it ends

Exercises

1. Calculate $\int_1^4 e^{\frac{-x^2}{2}} dx$ with high precision, splitting the calculation into multiple threads
2. Calculate $\int_2^7 \int_0^6 \sin x^2 \cos y^2 dxdy$ with high precision, splitting the calculation int multiple threads

Advanced:

1. Check if 4351376251585937029 is a prime, using multiple threads
2. Check if 3437587238752890252973423 is a prime, using multiple threads

std::thread

- std::thread allows a much more precise way to control threads
- Its constructor accepts the function name and its arguments after it
- The thread probably will start running immediately after its construction
- Its join() method will cause the calling thread to wait for it to end
- Its detach() method will cause the thread to run independently and get destroyed when it ends

```
void report(int after) {
        sleep(after);
        std::cout << "Slept" << std::endl;
}
//...
std::thread reporter(report, 1);
std::cout << "Waiting" << std::endl;
reporter.join();
```

Communication between threads

- One thread may use the same variables as another thread (they may be globals, or given to it by reference at creation time, ...)
- It may happen that one thread reads a pointer and begins to use it, another thread replaces the pointer and deletes the objects it points to and the first thread is still reading the deleted data incorrectly
- If the CPU is busy with many threads, threads often get paused and the break between execution of two subsequent lines can be extremely long
- There may be a variable that marks if the pointer is being edited, but one thread may read that it's correct, get interrupted, another thread marks that it's not correct, change it, then the first thread returns and reads incorrect data that has been correct on its previous instruction; it may also happen that both read that it's correct, simultaneously mark they're editing it and overwrite each other's parts
- There is no solution to this problem without special processor instructions

std::mutex

- std::mutex is a special variable that can be set only by one thread
- It has two states, *locked* and *unlocked*, its method try_lock() succeeds only once until it's unlocked again, lock() will keep trying locking it until it succeeds
- It can be used to prevent anything from mutual overwrite, but this approach might have a performance cost

```
bool tryJoin(std::mutex& mutex) {
        if (mutex.try_lock()) {
                for (unsigned int i=0; i < news.size(); i++)
                        entries.push_back(news[i]);
                mutex.unlock();
                return true;
        } else return false;
}
```

- std::condition_variable can be used to make the thread sleep until the mutex is unlocked

Deadlock

- If thread 1 locks mutex A, then thread 2 locks mutex B, then thread 1 tries to lock mutex B and thread 2 tries to lock mutex A, then thread 1 waits for thread 2, but thread 2 waits for thread 1, so they will wait forever; this is an example of a situation called *deadlock*
- Real deadlocks are usually more tangled, but can also be simpler, it may happen that thread 1 locks mutex A and then tries to lock it again, waiting for itself forever
- This may be hard to find because it happens unpredictably (the threads won't meet at the chokepoint so often) and threads have their local data correct

Compare and swap

- It's actually possible to make threads work with shared variables without locking, but it's not necessarily faster and is often much harder to code
- Data types that are not longer than a word (8 bytes usually) are written *atomically*, so that a half-written variable is never read
- Processor operations *compare and swap* simultaneously check if a variable is as expected, overwrite it as expected and return if it was as expected
- It is usually used so that the data is read, something is computed with them, then it's attempted to be written and if it fails (the data are not what they were when the computation begun), it will retry until it succeeds

```
int was, want;
do {
        was = accessed_;
        want = was - 2;
} while (!std::atomic_compare_exchange_weak(
                &accessed_, &was, want));
```

- Special types like `std::atomic_int` have overloaded operators to perform simple arithmetic operations on them atomically

Exercises

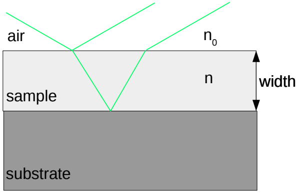
1. Write a program that reads numbers from a file on one thread, multiplies these numbers by 2 on another thread and writes them into another file on yet another thread (they'll need a protected `std::vector` to send data from one into another)

Advanced:

1. Same as the above, but do it without locking (because `std::vector` is not made for concurrent access, use simply a large fixed-width array)

Homework

- Calculate the width of a thin film using the wavelength dependence of its reflectivity under angle 60° from perpendicular
- Assume only first reflection from the substrate, assume the dependence of reflectivity on surface and substrate as $f(x) = ax + b$
- The two reflected beams interfere, constructively and destructively
- The index of refraction is 1.7
- The dependence is one of the files for this lecture
- Use multiple CPU cores to accelerate it



Advanced homework:

- The same, but assume that the reflectivity function is a polynomial of user-set degree