
Threads are EVIL
Parallelism meets programming reality

Next slides contain content which may be shocking

Northeast blackout

widespread power outage throughout U.S. and Canada affecting 55 million people

A race condition occurred in General Electric Energy's Unix-based energy management system. Once triggered, the bug stalled FirstEnergy's control room alarm system for over an hour. System operators were unaware of the malfunction. The failure deprived them of both audio and visual alerts for important changes in system state.

Unprocessed events queued up the primary and backup server. The server failures slowed the screen refresh rate of the operators' consoles from 1–3 seconds to 59 seconds per screen. The lack of alarms prevented operators control a simple failure which led to the world's second biggest power outage.

August, 14th 2003

World's second biggest power outage

Source: Wikipedia

Jan Feb Mar Apr May Jun Jul Aug Sept Oct Nov Dec

What ?

What is a thread?

A thread is a programming primitive. It is the smallest sequence of programmed instructions that can be managed independently by the scheduler of the OS.

All memory within the process is shared across the threads.

Sections that need to be executed sequentially need to be protected with a mutex, ...

The problem:

Uncareful programming leads to unpredictable behavior and hard to reproduce bugs.

Non-atomic data needs to be protected with mutexes, semaphores, RW-locks, ...

- Hard for programmers to reach watertight functionality
- Bugs do not show up that easy
  - Eg Heisenbug
- Little support from tools to reveal concurrency issues
- Mutexes, semaphores, … are expensive to use.

Alternatives

Futures and promises

Offloads method into separate thread. Sends a signal when finished.

Implemented by a thread. Securing shared data access is still required.

Uses threadpool: cheaper to construct

Multi-process

Split algorithm into separate process which can be monitored.

To interact with process:

- IPC required
- Cmd line or parsable output is required.

Mutexes	Atomic types	Futures and promises	Single threaded apps
OS primitive	Supported by many CPU architectures	Alternative for apps with event loop.	Easy to program, easy to maintain.
Thread is sleeping while waiting	Cheaper to use than mutex	Be aware of concurrent data access.	Requires IPC to feedback results or interact
Related to semaphores which have less overhead	Be aware of memory re-ordering	Supported by std, Qt, boost	Preferred by Google's and Apple's coding guidelines

Mutexes	Atomic types	Futures and promises	Single threaded apps
OS primitive	Supported by many CPU architectures	Alternative for apps with event loop.	Easy to program, easy to maintain.
Thread is sleeping while waiting	Cheaper to use than mutex	Be aware of concurrent data access.	Requires IPC to feedback results or interact
Related to semaphores which have less overhead	Be aware of memory re-ordering	Supported by std, Qt, boost	Preferred by Google's and Apple's coding guidelines

Mutexes	Atomic types	Futures and promises	Single threaded apps
OS primitive	Supported by many CPU architectures	Alternative for apps with event loop.	Easy to program, easy to maintain.
Thread is sleeping while waiting	Cheaper to use than mutex	Be aware of concurrent data access.	Requires IPC to feedback results or interact
Related to semaphores which have less overhead	Be aware of memory re-ordering	Supported by std, Qt, boost	Preferred by Google's and Apple's coding guidelines

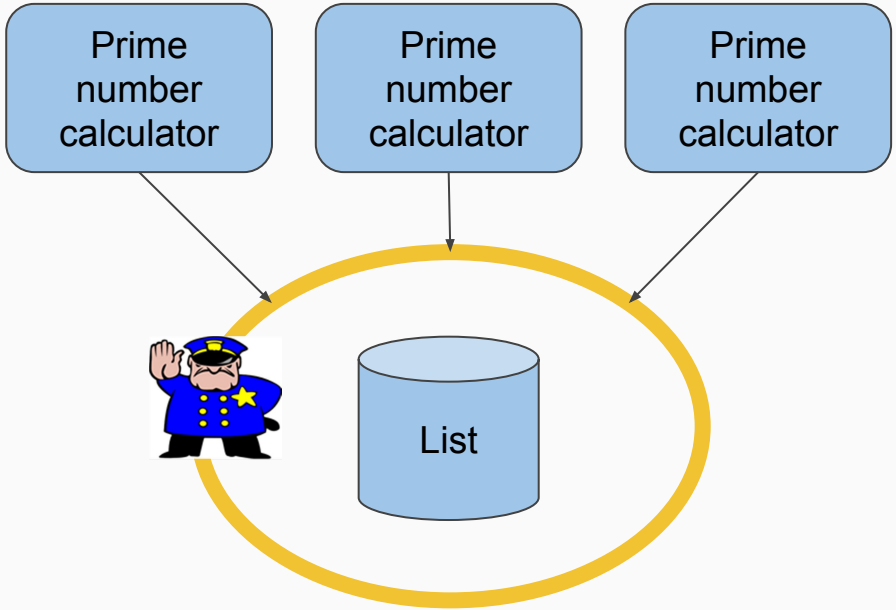
Mutexes	Atomic types	Futures and promises	Single threaded apps
OS primitive	Supported by many CPU architectures	Alternative for apps with event loop.	Easy to program, easy to maintain.
Thread is sleeping while waiting	Cheaper to use than mutex	Be aware of concurrent data access.	Requires IPC to feedback results or interact
Related to semaphores which have less overhead	Be aware of memory re-ordering	Supported by std, Qt, boost	Preferred by Google's and Apple's coding guidelines

Mutexes	Atomic types	Futures and promises	Single threaded apps
OS primitive	Supported by many CPU architectures	Alternative for apps with event loop.	Easy to program, easy to maintain.
Thread is sleeping while waiting	Cheaper to use than mutex	Be aware of concurrent data access.	Requires IPC to feedback results or interact
Related to semaphores which have less overhead	Be aware of memory re-ordering	Supported by std, Qt, boost	Preferred by Google's and Apple's coding guidelines

What have we learned?

Benchmark

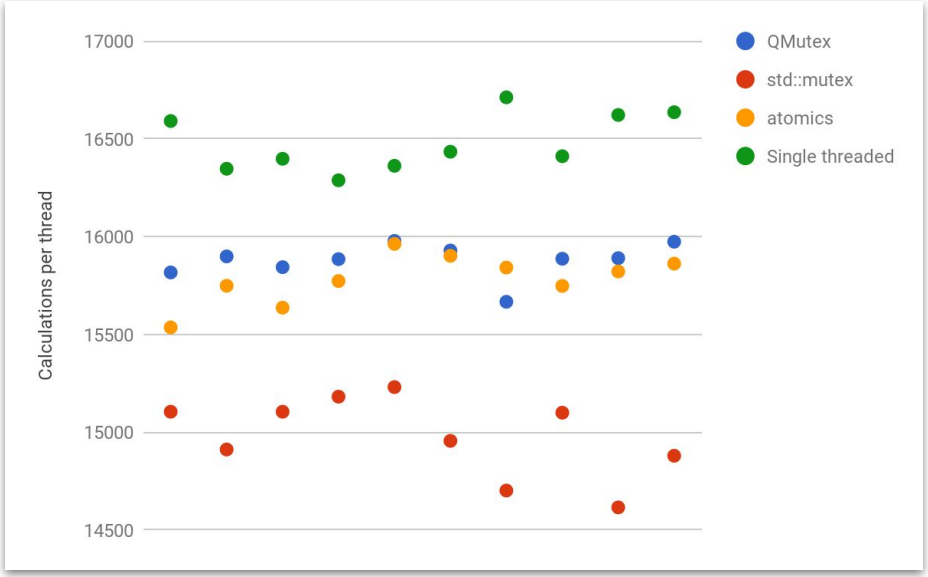
Demo app calculating prime numbers and adding results to a list during 30 seconds

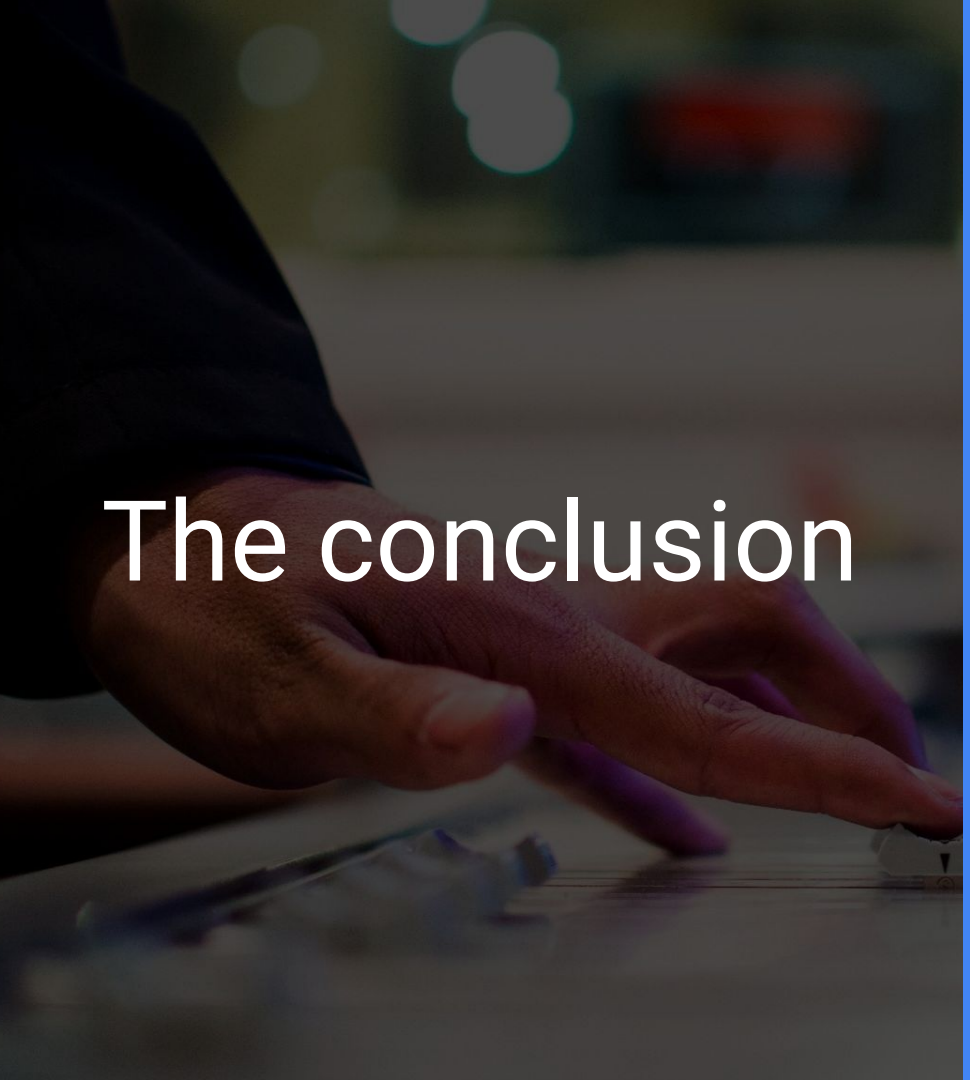


Code on https://github.com/frederikvannoote/becpp

Benchmark

Demo app calculating prime numbers and adding results to a list during 30 seconds





The conclusion

Mutexes are expensive and error prone.

Atomics are cheaper

Multi-process avoids sharing data all together and gives best performance, but require IPC.

More

Advanced threading libraries:

- Threadweaver
- Grand Central Dispatch

Apple coding guidelines:

- https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/Multithreading/AboutThreads/AboutThreads.html