Thread Scheduler Efficiency Improvements for Multicore Systems

Daniel Collin Frazier

Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA

18 November 2017
UMM, Minnesota

Introduction

- *Thread scheduler* is an important system component that manages the processing programs receive in a given time
- Always running, so it must be efficient
- Most computers before 2001 were equipped with one processor containing one core
- At the end of the single-processor single-core era (early 2000s) thread scheduling was largely considered a solved problem by the Linux community

"...not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy."

Linus, Torvals, 2001 [2]

Introduction

Hardware changed rapidly throughout the 2000s and those developments made thread scheduler implementation much more complex.

One of these changes was the development of systems with multiple cores which allows it perform tasks concurrently on each core

Outline

Outline

Using Threads

- *Threads* allow a program to run multiple independent tasks at the same time
- Useful for programs:
  - with long, mostly-independent computations
  - with a graphical interface

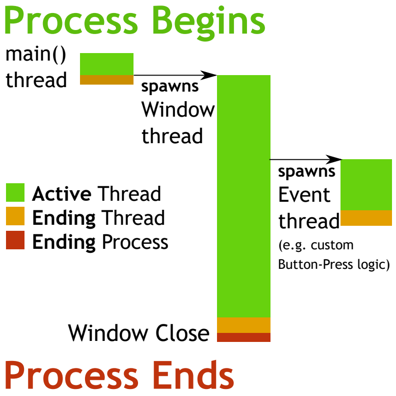


Figure: Example GUI Program. **Three** threads are created within **one** process

Using Threads

- A *multithreaded* program is a program that employs threads
- *Concurrent* computing techniques are techniques that allow many tasks to occur at the same time [W]
- *Parallel* computing techniques are techniques that allow many calculations to occur at the same time [W]
- Problems can be solved or improved using neither, either, or both of these techniques at once

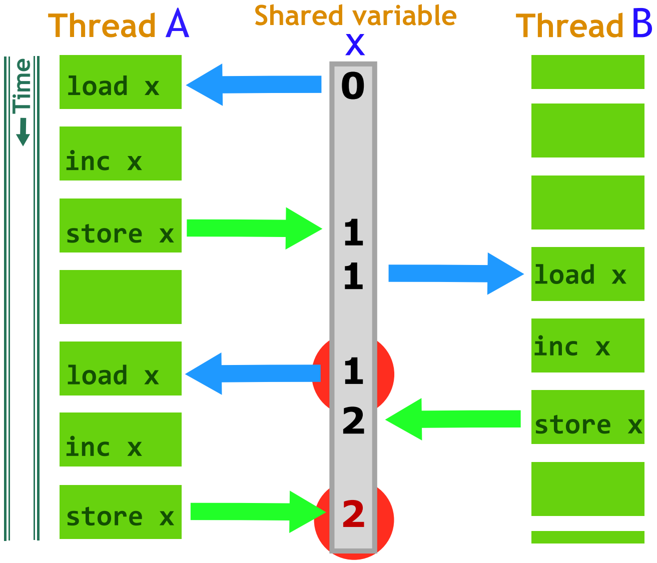
Using Threads

One problem multithreaded programs face are called *Race Conditions*.

Defined in Saltzer and Kaashoek as “A timing-dependent error in thread coordination that may result in threads computing incorrect results.”

Let’s see an example where two threads increment a shared variable.

Race Condition Example

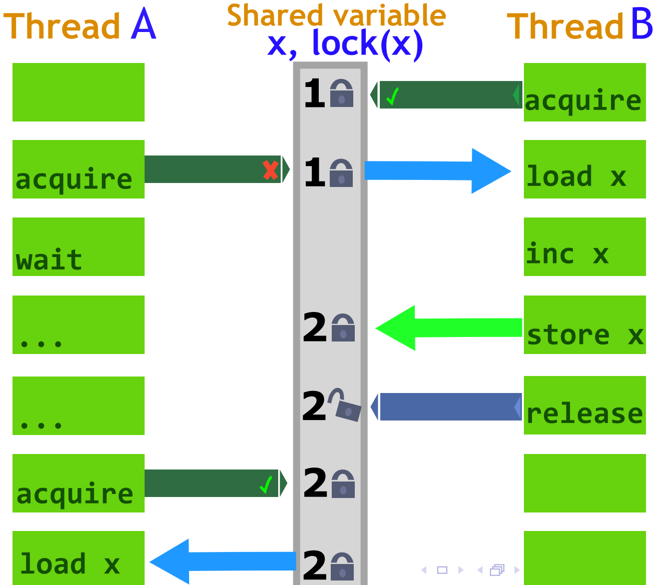


Synchronicity and Locks

- Race conditions can be fixed by controlling access to shared data.
- This control is achieved by employing locks.
- *Locks* secure objects or data shared between threads such that only one thread can read and write to it at one time.
- When a thread *locks* a lock, that thread **acquires** the lock
- When a thread *unlocks* a lock, that thread **releases** the lock

Now, let's fix the race condition in the previous example using locks

Lock Example



Outline

Completely Fair Scheduler (CFS)

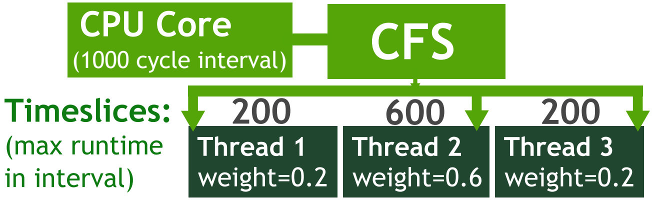
- Default Linux thread scheduler (there are others)
- Handles which threads are executed at what times and on which CPU cores
- Spend a *fair* amount of runtime on all threads
- The scheduler *switches* active threads on cores by saving and restoring thread and processor state information.
- These switches are called *context switches*.

We will continue to cover the CFS, then clarify process and thread state.

Completely Fair Scheduler (CFS)

Goal:

- Makes sure all threads run *at least once* within arbitrary interval of CPU cycles
- *Timeslice* maximum cycles evenly amongst threads, factoring in thread weights[1]
- The scheduler monitors the cycles that threads receive and switches them when they reach their max



[1] Priority (PR) and niceness (NI) values are responsible for determining process priority on Linux. We won't get into that here.

Runqueues

- The data structure within CFS that contains threads is called a runqueue
- A *runqueue* is a priority queue that sorts for threads that have received the least cycles in the current interval
- When a thread reaches its maximum time, the first thread in the runqueue is chosen

Runqueues on Multiple Cores

If each core needs work to do, how do each of them receive threads?

Will show that

- If all cores shared one runqueue, cores would be doing frequent expensive book-keeping in order to search for available work and...
- Each core should have its own runqueue

In order to understand the motivation for multiple runqueues, we need to know some about cache and processor state. (It will also help us later on)

Outline

Outline

Conclusions

- Added developmental plasticity to N-gram GP using Incremental Fitness-based Development (IFD).
- IFD consistently improved N-gram GP performance on suite of test problems.
- “Knocking out” IFD shows it’s valuable in all phases, even if it wasn’t used earlier in a run.
- IFD generates more complex, less converged probability tables.
- IFD generates more modules/loops & uses more low-probability paths.
- Currently exploring applications to dynamic environments.

Thanks!

Thank you for your time and attention!

Contact:

- `mcphee@morris.umn.edu`
- `http://www.morris.umn.edu/~mcphee/`

Questions?

References

H. Jo, W. Kang, C. Min, and T. Kim.
FLsched: A lockless and lightweight approach to OS scheduler for Xeon Phi.
In Günther Raidl, *et al*, editors, *GECCO '09*, pages 1019–1026, Montréal, Québec, Canada, 2009.

R. Poli and N. McPhee.
A linear estimation-of-distribution GP system.
In M. O'Neill, *et al*, editors, *EuroGP 2008*, volume 4971 of *LNCS*, pages 206–217, Naples, 26-28 Mar. 2008. Springer.

See the GECCO '09 paper for additional references.