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*: system component that manages the processing programs receive in a given time
- Always running, so it must be efficient
- Pre-2000 single-core era, scheduling was easy
- Near end, problem considered solved by Linux community

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

Linus, Torvals, 2001 [2]

Introduction

- Popular hardware changed rapidly throughout the 2000s and those developments made thread scheduler implementation much more complex
- Increasing affordability and adoption of multicore systems
- Unfortunately, this increasing complexity led to bugs

A Decade of Wasted Cores

- Lozi et al. found four bugs in the Linux thread scheduler and fixed them [2]
- Previously undetected, required the development of new tools to notice them
- The Scheduling Group Construction bug
- The Group Imbalance bug
- The Overload-on-Wakeup bug
- The Missing Scheduling Domains bug

https://goo.gl/3wsfVU

NAS Parallel Benchmark (NPB)

- Set of 9 computational applications that heavily depend on threading
- Names: *bt, cg, ep, ft, is, lu, mg, sp, ua*
- Executes problems within scheduler, records execution times
- Will refer to this benchmark later on

Scheduling Group Construction bugfix results

Application	With bug (sec)	Fixed (sec)	Improvement
is	271	202	1.33x
mg	49	24	2.03x
lu	1040	38	27x

Table: "NPB applications with least, median and most improvement between a bugged and fixed scheduler" [2]

Group Imbalance and Overload-on-Wakeup bugfixes

Bug Fixes	Full Database benchmark
No bugfix	542.9s
Group Imbalance	512.8s (1.05x)
Overload-on-Wakeup	471.1s (1.13x)
Both	465.6s (1.14x)

Table: "Impact of these bug fixes on a popular commercial database (values averaged after five runs.)" [2]

- They disabled and re-enabled a core before each run of the benchmark

Missing Scheduling Domain bugfix results

- Occurs rarely, but easily reproducible

Application	With bug (sec)	Fixed (sec)	Improvement
ep	72	18	4.0x
mg	81	9	9.03x
lu	2196	16	137.59x

Table: NPB applications with least, median and most improvement between a bugged and fixed scheduler [2]

Outline

Concepts

Thread Scheduling on Linux

Two New Schedulers

Conclusions

Outline

Concepts

- Threads
- Synchronicity and Locks
- Thread State and Cache

Thread Scheduling on Linux

Two New Schedulers

Conclusions

Processors

- Responsible for executing code
- Contain a number of cores:
  - *Single-core processor* (one processing unit)
  - *Multicore processor* (two or more processing units)
  - *Manycore processor* (20 or more processing units)
- A processor with multiple cores allows it to perform tasks concurrently on each core

Using Threads

- Imagine you're using photoshop but it has one thread
- You load a large image and perform an expensive filter operation

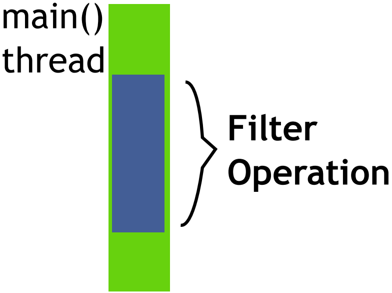


Figure: Running a filter operation in a hypothetical single-threaded version of photoshop

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

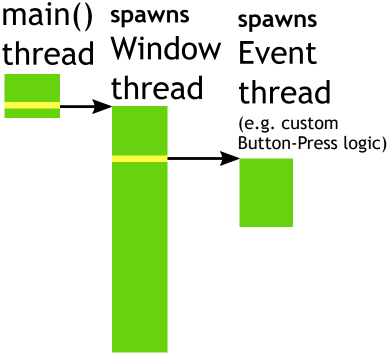


Figure: Example GUI Program. **Three** threads are created within **one** process

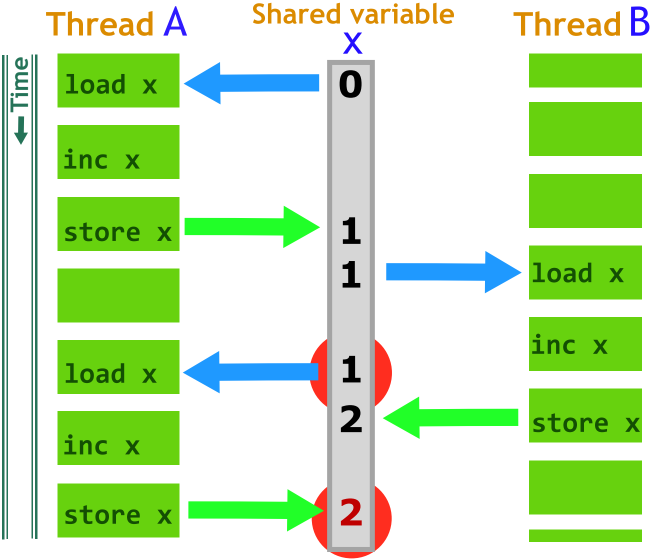
Race Conditions

- Problem in multithreaded systems.

  A timing-dependent error in thread coordination that may result in threads computing incorrect results

  — Saltzer and Kaashoek [W]

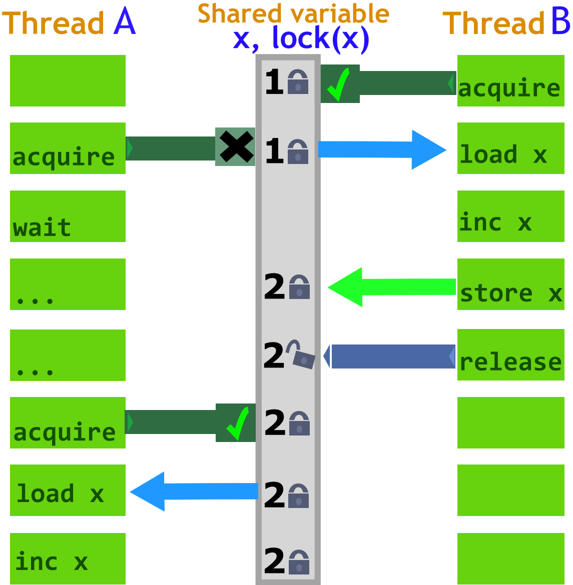
Race Condition Example



Synchronicity and Locks

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

Lock Example



Thread A	Shared variable x, lock(x)	Thread B
	1 🔒 ✓	acquire
acquire	✗ 1 🔒	load x
wait		inc x
...	2 🔒	store x
...	2 🔓	release
acquire	✓ 2 🔒	
load x	2 🔒	
inc x	2 🔒	

Process and Thread State

- Process State

  Resources shared amongst its multiple threads

- Thread State

  Scheduler uses this information to pause and resume a thread's execution

Context Switching

- The scheduler *switches* active threads on cores by saving and restoring thread and processor state information.
- These switches are called *context switches*
- Scheduler performance

Cache

- Local copy of data designed for fast retrieval
- Hierarchical structure
- Placement relative to core:
  - on
  - inside of
  - outside

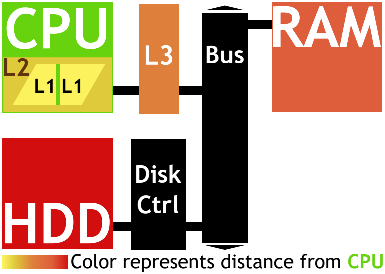


Figure: Distance of various forms of memory from CPU

Cache

- *Locality*: Speed of memory read and writes decrease as distance from CPU increases
- Cache is the fastest form of memory
- *Cache coherence:* Any changes to memory shared by two caches must propogate to the other to maintain correctness

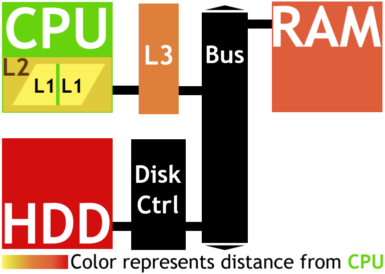


Figure: Distance of various forms of memory from CPU

Outline

Concepts

Thread Scheduling on Linux

- Completely Fair Scheduler

Two New Schedulers

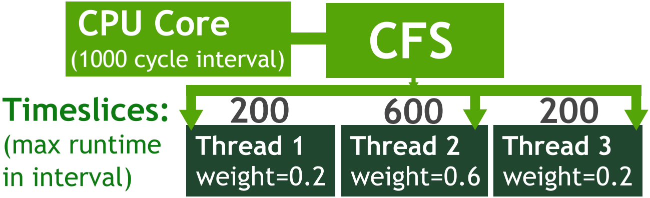
Conclusions

Completely Fair Scheduler (CFS)

- Default Linux thread scheduler (there are others)
- Handles which threads are executed at what times on this core
- Spend a *fair* amount of runtime on all threads
- Designed with Responsiveness, Fairness, and load-balancing in mind.

Single-core Completely Fair Scheduler (CFS)

- Runs on one core
- Ensure all threads run *at least once* within arbitrary interval of CPU cycles
- Distribute *timeslices* (max CPU cycles) among threads
- Threads with higher priority (weights) get larger timeslices



CFS Runqueue

- Data structure containing threads
- Priority queue: sorts threads by number of cycles consumed in current interval
- When thread reaches its maximum cycles, preempted

Runqueues on Multiple Cores

- Since process states are heaver than thread states, context switches between threads of different processes are more expensive
- If cores shared a runqueue, access and changes need to be synchronous and cache-coherent
- This would slow the system to a crawl
- So each core has its own runqueue and threads

- Load on each of the core's runqueues must stay balanced
- CFS periodically runs a load-balancing algorithm

Outline

Concepts

Thread Scheduling on Linux

Two New Schedulers

- Shuffler
- FLSCHED

Conclusions

Shuffler and FLSCHED

Both schedulers aim to solve the same problem, but for different architectures.

Shuffler	→	multiprocessor multicore
FLSCHED	→	single-chip manycore processor

Problem: Adding more threads to parallel computing applications on CFS makes the application operate slower rather than faster!

The term for what these systems face is called *lock contention* and it is to blame for the high acquisition times.

Shuffler

By researchers Kumar et al.

(in short, unfinished)
Shuffler

- Monitors the lock acquisition times of various executing threads
- Isolates groups of threads that have similar lock acquisition times, and migrate those threads to the same processor, assuming that those groups of threads may be contending with eachother for the same lock and would both have faster acquisition times if they were colocated.

FLSCHED: The Lockless Monster

(in short, unfinished)
Designed by Jo et al. with manycore processors (specifically the Xeon Phi) in mind.

The Xeon and Xeon Phi manycore processors come with varying degrees of cores, from 24 to 76 cores. With such potential for parallelism and as the number of cores rise, any small hitch in a critical system component can lead to significant performance degradation.
The hitches that they isolated were due to locks emplaced by features and requirements of the CFS scheduler.

One requirement to rule them all: EFFICIENCY!!!

FLSCHED Aims to improve the efficiency by removing all locks from the scheduler component. In order to do this, they gutted the requirements and features of the CFS scheduler and simplified.

The requirements that they removed are:

- Fairness,
- Responsiveness,

The features that they removed were:

- Group scheduling

FLSCHED is a scheduler implementation without any locks[1].

Lock contention may exist in FLSCHED, but it isn't a problem in the same way as in Shuffler because none of the cores exist on other processors, there is only one processor, so that latency is not a concern.

[1] In Jo et al. they clarify it is still built on top of the *scheduler core* which still locks in it.

Outline

Concepts

Thread Scheduling on Linux

Two New Schedulers

Conclusions

Conclusions

todo

Thanks!

Thank you for your time and attention!

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.