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
- Always running, so it must be efficient
- Pre-2000 single-core era, scheduling was easy
- Led majority of Linux community to believe problem solved

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

Linus, Torvals, 2001 [1]

Introduction

- Popular hardware changed rapidly throughout the 2000s
- Increasing affordability and adoption of multicore systems
- Hardware changes complicated thread scheduler implementation
- Complexity led to bugs that have been present for a decade

A Decade of Wasted Cores

- In *A Decade of Waster Cores*
- Lozi et al. Found four bugs in Linux thread scheduler, fixed them
- Previously undetected, required the development of new tools

https://goo.gl/3wsfVU

A Decade of Wasted Cores

- Lozi et al. compared performance benchmarks ran on **buggy** and **fixed** Linux scheduler implementations
- Below are average performance improvements

Bug title	Improvement
The Scheduling Group Construction bug	5.96x
The Group Imbalance bug	1.05x
The Overload-on-Wakeup bug	1.13x
The Missing Scheduling Domains bug	29.68x

from Lozi et al. [1]

Outline

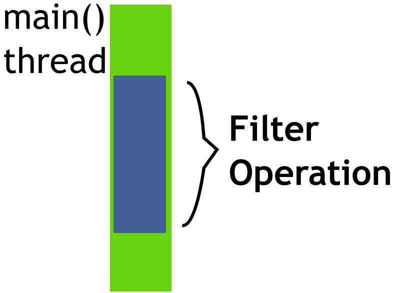
Outline

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)
- Multiple cores allows processor to perform multiple tasks concurrently on each core

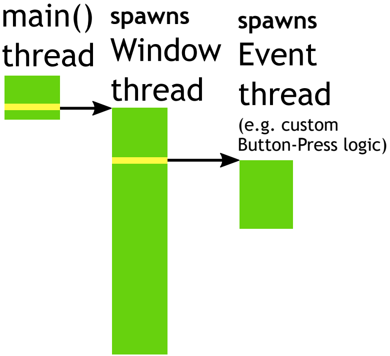
Multithreading Example

- Imagine you're using photoshop, but assume one thread
- Say you load a large image and perform an expensive filter operation



Threading

- *Threads* allow programs to run multiple independent tasks concurrently
- Useful for programs:
  - with long, mostly-independent computations
  - with a graphical interface



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

What if I ask you all a question right now?

What if I ask you all a question right now?

If all answered at once, chaos!

Raise hands to control who gets to talk, this is like locks!

Synchronicity and Locks

- Control achieved by employing locks
- *Locks* secure objects or data shared between threads 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

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

- Note: Process states are much heavier than thread states

Context Switching

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

Cache

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

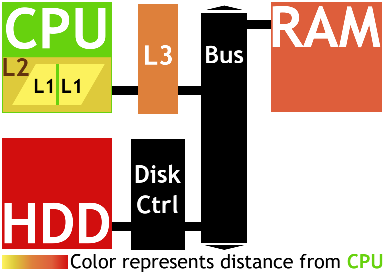


Figure: Distance of various forms of memory from CPU

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

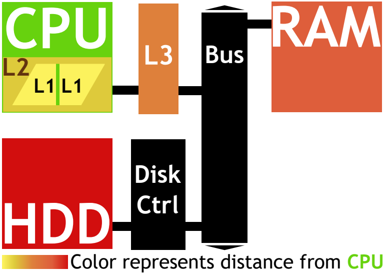


Figure: Distance of various forms of memory from CPU

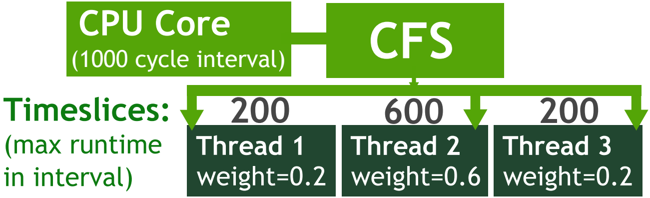
Outline

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 and fairness 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

- Process states heavier than thread states, so 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
- Would slow the system to 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

Shuffler and FLSCHED

- Both schedulers aim to solve the same problem, but for different architectures
- **Problem:** Adding more threads to certain parallel computing applications on CFS makes the application operate slower rather than faster!
- Architectures:

Shuffler	$\rightarrow$	*multiprocessor multicore*
FLSCHED	$\rightarrow$	*single-chip manycore processor*

Shuffler

- Researchers Kumar et al. measured lock times of massively parallel applications
- *Lock times*: amount of time process spends waiting for locks
- Found that massively parallel shared-memory programs experienced high lock times

Lock Contention

- When two threads repeatedly contend for one lock, both threads are frequently waiting for each other to release
- If the two threads are located on separate processors, this problem is compounded by reduced locality
- Further, when both of the threads repeatedly modify the data corresponding to their lock, the cache of both processors must continue to update each other
- High *lock contention*

Shuffler

- CFS not mindful of lock contention or parent processes when choosing cores for threads
- Kumar et al. wanted to create a scheduler that did!
- Used Solaris scheduler as base
- **Strategy**: Migrate threads whose locks are contending so they are near each other
- How do you determine which threads' locks are contending?
- Contending threads have similar lock acquisition times

```
input  : N: Number of threads;
         C: Number of Processors.
repeat
    i. Monitor Threads – sample lock times of N threads.
    if lock times exceed threshold then
        ii. Form Thread Groups – sort threads according to
         lock times and divide them into C groups.
        iii. Perform Shuffling – shuffle threads to establish
         newly computed thread groups.
    end
until application terminates;
```

Shuffler Performance

- Kumar et al. compared the efficiency of Shuffler vs Solaris scheduler
- Used programs from four benchmarks to gather data

Program	% Improvement
BT	54.1%
SC	29.0%
RX	19.0%
JB	14.0%
OC	13.4%
AL	13.2%
AS	13.0%
PB	13.0%
VL	12.8%
FS	12.0%

Program	% Improvement
FM	10.7%
AM	9.3%
GL	9.1%
EQ	9.0%
MG	8.8%
FA	6.0%
WW	5.2%
SM	4.7%
GA	4.0%
RT	4.0%

FLSCHED: The Lockless Monster

- Designed by Jo et al. with manycore processors in mind, particularly the Xeon Phi
- The Xeon and Xeon Phi have 24 to 76 cores.
- One processor, so cache looks different than system that would use Shuffler
- With such parallelism, small pauses significantly reduce efficiency
- In the CFS, pauses come from locks necessitated by its features and requirements

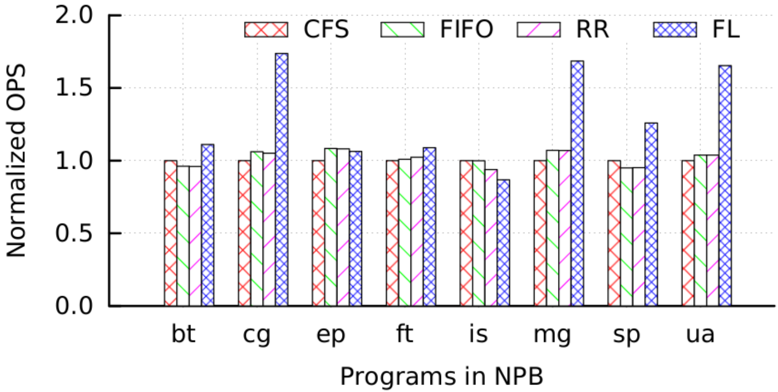
One requirement to rule them all: EFFICIENCY!

- FLSCHED Improves efficiency by removing all locks from the scheduler implementation
- Gutted requirements and features of CFS and simplified
- Requirements they removed were **Fairness** and **Responsiveness**
- Context switches requests delayed to reduce chance another thread steals the core in hope thread reactivates
- Threads never forcefully preempt, instead join runqueue with high priority
- Removed scheduler statistics reporting capabilities

FLSCHED Performance

- Used 8 of 9 programs the NAS Parallel Benchmark (NPB)

Operations per second (OPS) relative to CFS, from Jo et al. [1]



Outline

Conclusion

- Thread scheduling is an important problem and becomes more relevant as number of cores increase
- System architecture can have surprising complexity in its effect on efficiency
- CFS tries to be the go-to scheduler for all problems, but can't
- Does well, but when you need some extra push there are powerful alternatives available

Thanks!

Thank you for your time and attention!

Questions?

References

- Jo, Heeseung and Kang, Woonhak and Min, Changwoo and Kim, Taesoo.
  FLsched: A lockless and lightweight approach to OS scheduler for Xeon Phi.
  In Proceedings of the 8th Asia-Pacific Workshop on Systems *3 APSys '17*, pages 8:1–8:8, Mumbai, India, 2017. ACM.

- K. Kumar and P. Rajiv and G. Laxmi and N. Bhuyan
  Shuffling: A framework for lock contention aware thread scheduling for multicore multiprocessor systems
  In 2014 23rd International Conference on Parallel Architecture and Compilation Techniques *3 PACT* , pages 289–300, 2014.

Lozi, Jean-Pierre and Lepers, Baptiste and Funston, Justin and Gaud, Fabien and Quéma, Vivien and Fedorova, Alexandra
The Linux Scheduler: A Decade of Wasted Cores
In Proceedings of the Eleventh European Conference on Computer Systems *EuroSys '16*, pages 1:1–1:16, London, United Kingdom, 2016. ACM.