# 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

DRAFT
Todos in Yellow

# Introduction

- The thread scheduler is an important system component that is always running
- It manages the runtime that programs receive
- Because it is a system component, it must be as efficient as possible
  - Reduce time spent doing critical managerial tasks (time spent in the “critical section”)
- The problem of thread scheduling has been around since the 1960s and for a while, scheduling implementation was largely unchanged
- Increasing hardware requirements in the early 2000s made the problem more complex

“And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy.”
Linus Torvalds, 2001 [2]

Mhmmm… *sure*… :)

# Overview

- Establish needed concepts
- Thread scheduling and thread load-balancing on Linux
- Bug fixes and two new developments to the Linux thread scheduler

Before we talk about thread scheduling,

let’s get some terms out of the way!

# Outline

- Establish needed concepts
  - Threads, Multithreading Programs
  - Parallel programming
  - Synchronicity and Locks
- Thread scheduling and thread load-balancing on Linux
- Bug fixes and two new developments to the Linux thread scheduler

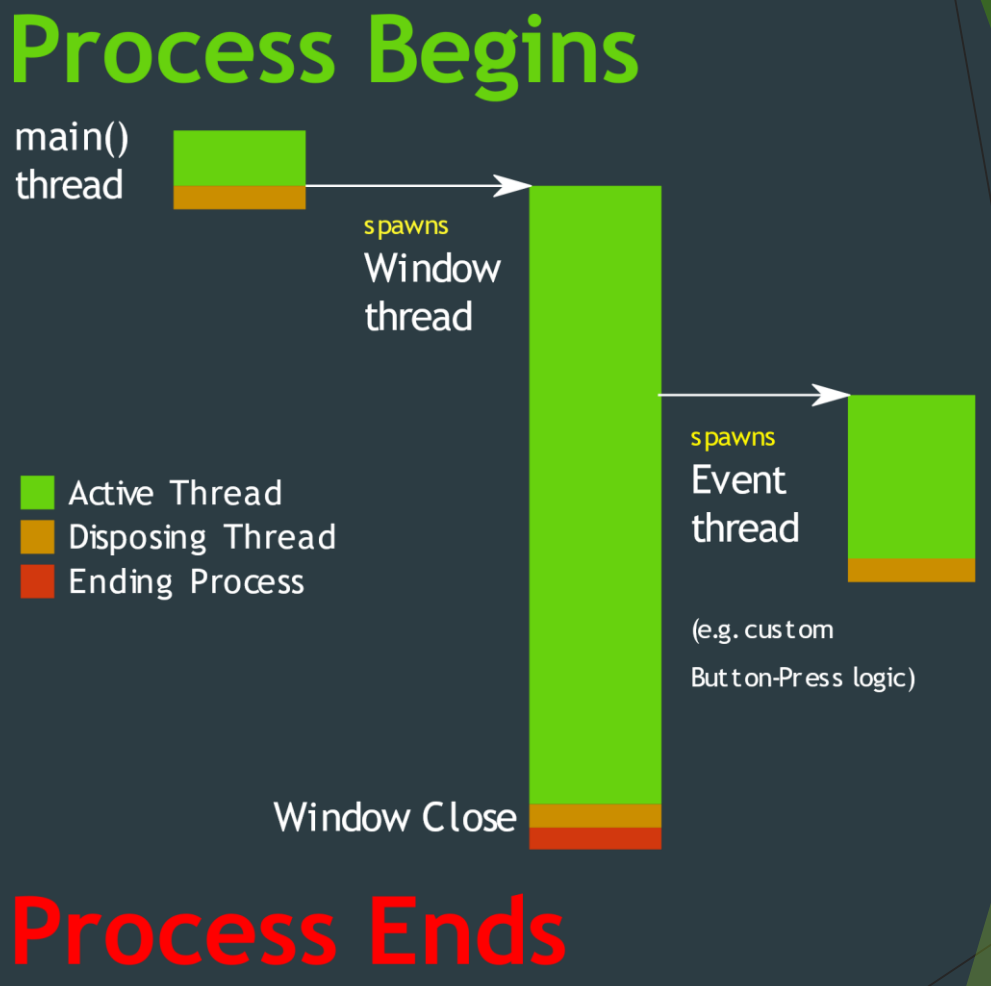
# Using Threads

- Threads allow a program to run more than one independent task at one time

- Useful for…
  - Programs with long, mostly-independent computations
  - Programs with graphical interface

- Example GUI Program (right)

  In this figure three Threads are created within one Process.



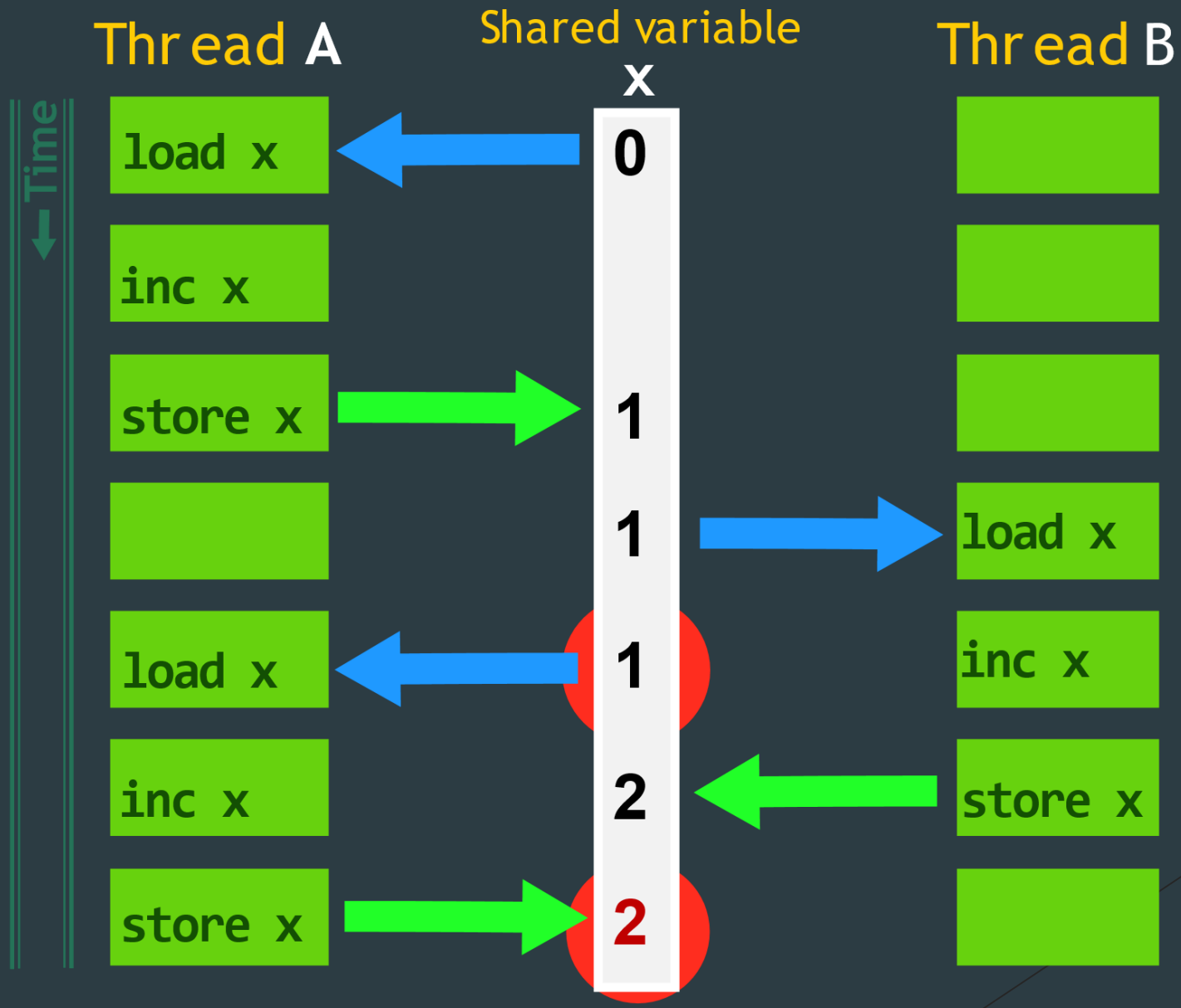
# Using Threads

- 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

# Parallel Programming

- [may not need this slide] probably best not, considering length

# Race Condition (example)



# Synchronicity and Locks

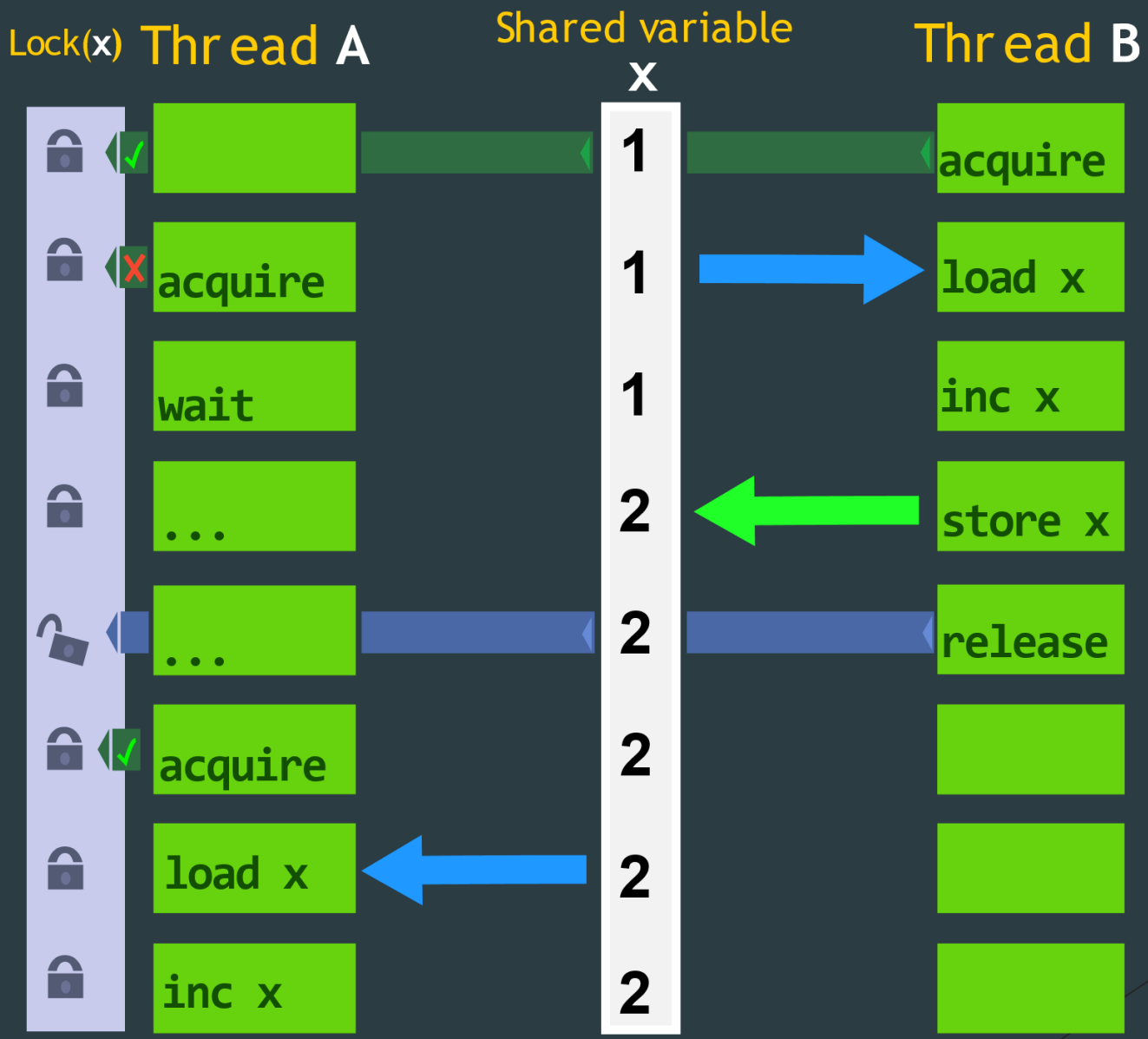
- Synchronicity can achieved by employing locks
- When locks are used properly, they fix race conditions

- 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

Let's fix the race condition in the previous example using locks

# Lock Example



# Outline

- Establish needed concepts
- Thread scheduling and load-balancing on Linux
  - Completely Fair Scheduler (CFS)
  - Cache and Scheduling Domains
  - Load Balance algorithm for the CFS
- Bug fixes and two new developments to the Linux thread scheduler

# 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 by saving and restoring thread and processor state information.
- Switching active threads and processes are called context switches.

- The CFS implementation for single-core systems is simple
- So for now, let's assume single-core

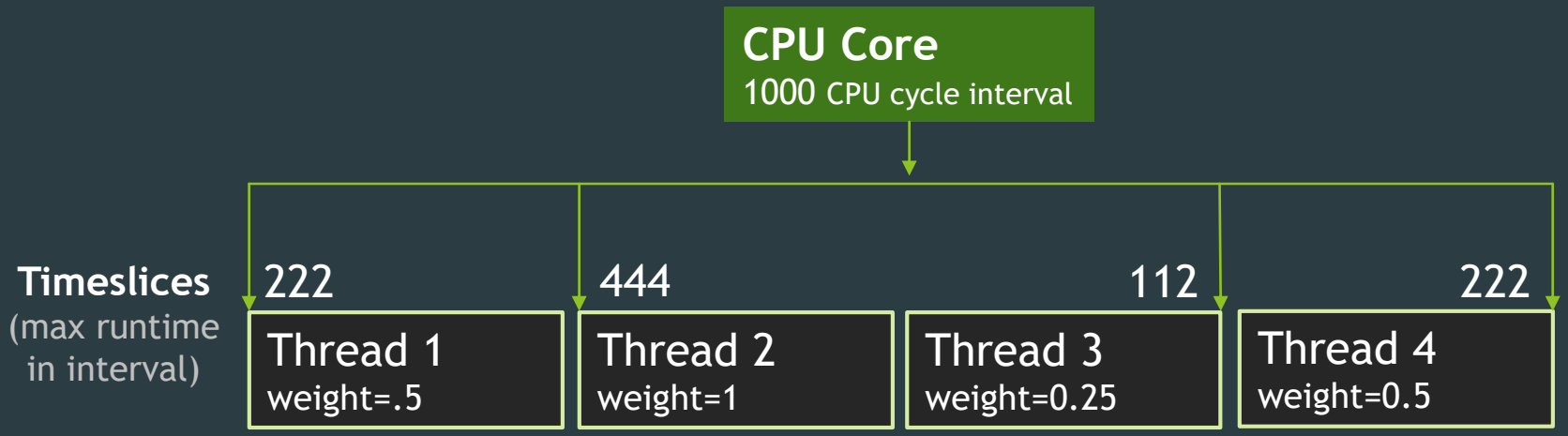
We will cover what thread and process state information consist of after covering the CFS.

# Completely Fair Scheduler (CFS)

Implementation of the weighted fair queueing (WFQ) scheduling algorithm

**Goal:**

- Make sure all threads run at least once within an arbitrary interval of CPU cycles
- Timeslice CPU cycles evenly amongst threads, prioritizing higher weights*




Each instance of the CFS uses a runqueue to chose which of these threads run.

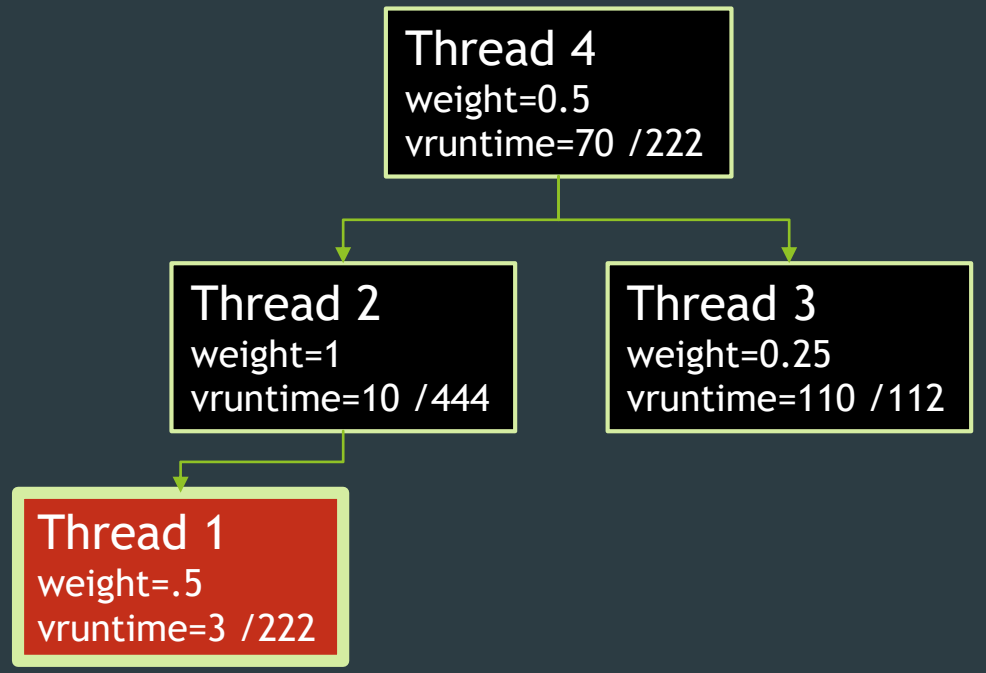
* Disclaimer: priority (PR) and niceness (NI) values are responsible for determining process priority (and weights) on Linux. We won't get into that here.

# Runqueues in CFS

- While a thread is running it accumulates vruntime
- When vruntime reaches the maximum, the running thread is preempted (replaced) by another thread.
- A runqueue is a priority queue that sorts on vruntime.
- When the scheduler chooses a replacement thread, it selects the thread with the minimum vruntime from the runqueue.

- The priority queue of choice on Linux is a red-black tree. In the example on the right, Thread 1 would be selected.

## Example Runqueue



# Multiple Runqueues in CFS

(no longer assuming single-core implementation)

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

- This is sufficiently vague… Let's take a closer look.
- 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)

# Process and Thread state

Process state

Consists of resources that each of the processes' threads should have access to

- Compiled code and data
- Sockets
- File handles
- Process control block (Logistical information)

Thread state

The scheduler uses the following information for thread execution and in order to pause and resume execution

- Run-time stack (when a thread is made, what's in its stack?)
- Clone of the CPUs registers from when the thread was last *active* (including the instruction pointer, to resume processing)

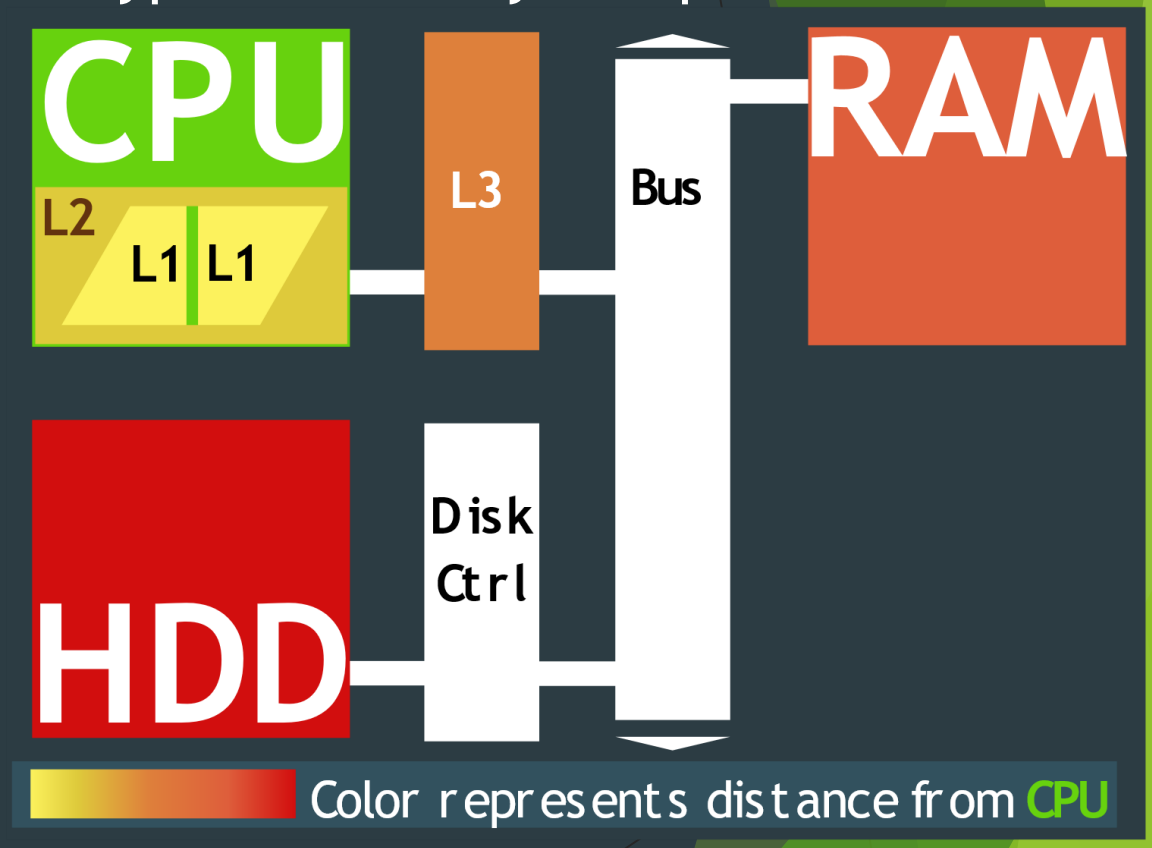
Processor states contain much more information than thread states, thus context switches between threads of different processes are more expensive

# CPU Cache

- Cache and memory in general exist in a hierarchy
- Levels of cache are labelled L1, L2, L3, etc.
- L1 cache is also called last level cache
- How cache is distributed depends on the machine:
  - A cache can exist *on*, *built-in to*, or *outside* a processor
  - Cache can be one *per processor,* or one *per n cores*

- Locality: Speed of memory read and writes decreases as distance from CPU increases
- Cache is the fastest form of memory
- Any memory in $L_i$ cache exists in $L_{i+1}$ cache and RAM
- Any changes to memory shared by two L1 caches must propogate to the other to maintain correctness

A typical memory setup




http://flylib.com/books/en/4.414.1.135/1/

# Multiple Runqueues in CFS

- Considering cache locality and the size of process state information, it would be disadvantageous for the CFS running on each core to consult runqueues external to that core, this is why multiple runqueues are necessary.
- The load on each core's runqueue must stay balanced.
- In order to do so most schedulers, including the CFS, periodically run a load-balancing algorithm.

- My first primary source Lozi, Lepers, Funston, Gaud, Quéma and Fedorova detailed four issues with the CFS and load balancer.
- Some of these bugs have been present for a decade because they were difficult to detect and required new tools to spot. (verify dates, maybe all?)

Let's look at the load balancer so that we can get to these bugs!

# CFS Load Balancer

- Threads are assigned a metric called *load* to best distribute threads to cores
- Defining load is trickier than you might expect, but for the sake of brevity here is how it is defined:

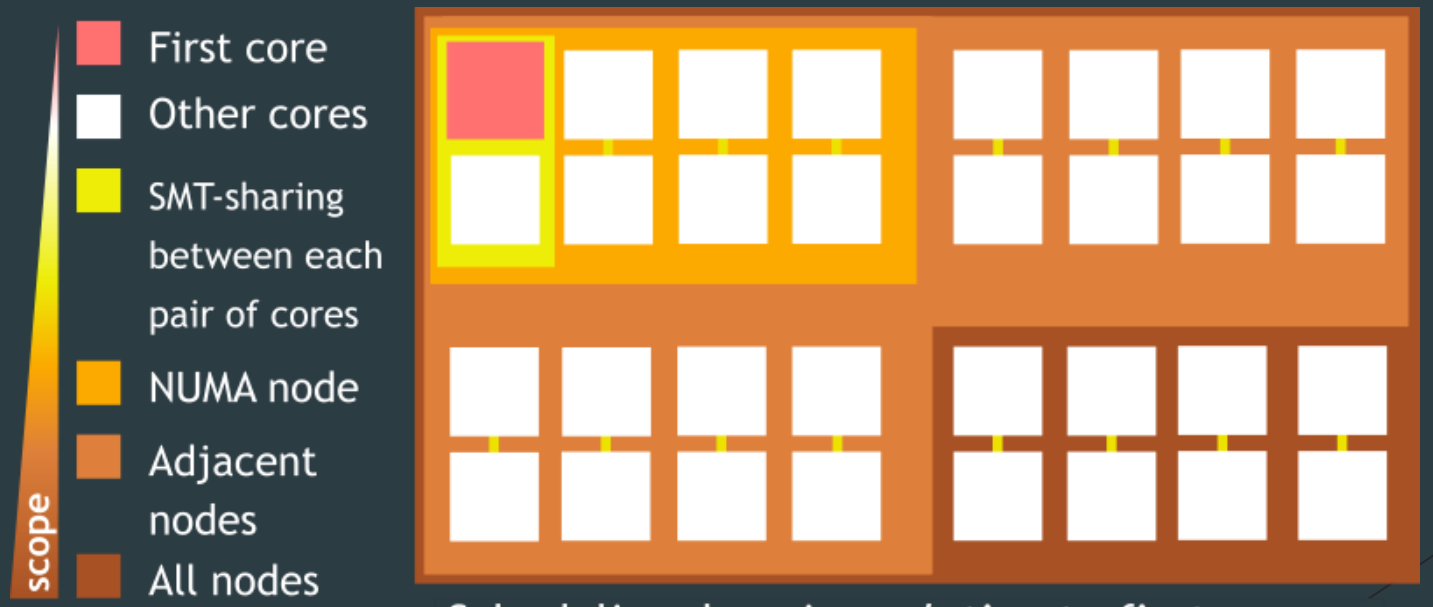
Load metric is a combination* of a thread's weights and average CPU use divided by the number of threads in the parent process.

- We divide so that two processes with the same priority, but a different number of threads, still receive a fair amount of time
- We factor in the average CPU use so that if you have a program with high priority that isn't busy, then it shouldn't need to be scheduled until it is busy

* Lozi et al. did not mention what kind of "combination"

# CFS Load Balancer

- For efficient balancing, new threads in a process are first considered to be placed on the first core, then nearby cores
- This is accomplished using scheduling domains
- Scheduling domains differ between cores and exist in a hierarchy based on how they share resources.




Scheduling domains *relative to first core* on AMD Bulldozer machine used by Lozi et al.

# CFS Load Balancer

- Much like in picking the load metric, the approach to load balancing is also trickier than you might expect.

Load balancing algorithm:

1. The *first core that is idle*, else the *first core*, in any given scheduling domain is responsible for load balancing that domain.
2. Average load is computed for each scheduling group within each domain where scheduling groups are lower units of scheduling domain
3. If the *busiest group is less busy* than the *group containing this core*, then the threads are considered balanced at this level. (Threads only steal load)
4. If *busiest group is busier*, then this core balances load* between these groups.

Many optimizations were made to this algorithm that introduced bugs.
And we are finally prepared to talk about them!

* Lozi et al. did not clarify how scheduling groups balance load

# Outline

- Establish needed concepts
- Thread scheduling and thread load-balancing on Linux
- Bug fixes and two new developments to the Linux thread scheduler
  - Four bugs found within current implementation of CFS
  - Shuffler: Cache locality improvements via thread migration
  - FLSCHED: The lockless thread scheduler

# Bugs everywhere!

Four bugs:

1. The Group Imbalance bug
2. The Scheduling Group Construction bug
3. The Overload-on-Wakeup bug
4. The Missing Scheduling Domains bug

Performance results were gathered by running the NAS Parallel Benchmark (NPB) developed by NASA

# The Group Imbalance bug

- Lozi et al. ran *program A* with 64 threads at the same time as a single-threaded *program B*
- The load balancer was not balancing *A*'s threads to idle cores
- Because of the division of number of threads in definition of load metric, each of the threads in *A* had 1/64$^{th}$ the load of *B*'s thread
- But the ***group*** containing the 64 *A* threads and ***group*** containing *B* still had the same load so the load balancer considered the ***groups*** balanced!

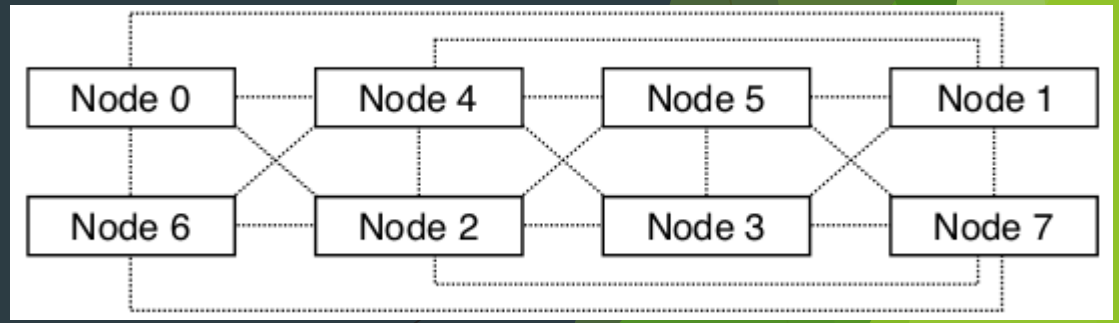
- They fixed the bug by defining the load of a scheduling group as the *minimum load* amongst cores in the group instead of average
- After the fix, program *A* improved by 13%, program *B*'s runtime was not affected
- A certain program in the NAS Parallel Benchmark improved *13 times*

* Program *A* was *make*, program *B* was *R*

# The Scheduling Group Construction bug

- The first scheduling group is constructed by adjacent nodes to the first node, then subsequent groups by adjacent nodes of not in any previous group.
- It is possible for a node to be within one hop of each starting node
- Scheduling groups for the below figure would be [0,1,2,4,6] and [1,2,3,4,5,7]
- 1 and 2 are in both scheduling domains and are two hops apart
- If Node 2 should ever steal work from Node 1, it won't because 1 and 2 always contribute to the load sum of each scheduling group and be considered balanced.

They fixed this bug by defining scheduling groups to always consider adjacent nodes from the source node, rather than predefined groups.




From Lozi et al.

# Scheduling Group Construction bugfix Impact

| Application | Time w/ bug (sec) | Time w/o bug (sec) | Speedup factor (×) |
|---|---|---|---|
| bt | 99 | 56 | 1.75 |
| cg | 42 | 15 | 2.73 |
| ep | 73 | 36 | 2 |
| ft | 96 | 50 | 1.92 |
| is | 271 | 202 | 1.33 |
| lu | 1040 | 38 | 27 |
| mg | 49 | 24 | 2.03 |
| sp | 31 | 14 | 2.23 |
| ua | 206 | 56 | 3.63 |

Table 1: Execution time of NAS applications with the Scheduling Group Construction bug and without the bug. All applications are launched using `numactl --cpunodebind=1,2 <app>`.

From Lozi et al.

# The Overload-on-Wakeup bug

- There is an optimization where threads awakened by other threads are placed on the same core to increase cache locality
- But it does this disregarding whether it should start on an idle core instead!

- This problem only occurs in environments where threads sleep frequently, such as database systems
- The fix for this was to modify the wakeup code:
- When a thread awakens, if the same core isn't busy, then it will go there. Otherwise it goes to an idle core

- Results on the right are from the commercial database benchmark TPC-H

| Bug fixes | TPC-H request #18 | Full TPC-H benchmark |
|---|---|---|
| None | 55.9s | 542.9s |
| *Group Imbalance* | 48.6s (−13.1%) | 513.8s (−5.4%) |
| *Overload-on-Wakeup* | 43.5s (−22.2%) | 471.1s (−13.2%) |
| Both | 43.3s (−22.6%) | 465.6s (−14.2%) |

Table 2: Impact of the bug fixes for the Overload-on-Wakeup and Group Imbalance bugs on a popular commercial database (values averaged over five runs).

From Lozi et al.

# The Missing Scheduling Domains bug

- This bug was fixed, but regressed after a refactor on Linux kernel version 3.19+
- Only occurs rarely, when a core is disabled and re-enabled
- Bug causes the system to misrepresent amount of scheduling domains and load balancing would never happen
- Threads and processes would be stuck on the nodes they’re on

- Reintroducing the removed line fixed the problem. Easy peasy.

# Missing Scheduling Domains bugfix Impact

NAS Parallel Benchmark after disabling and re-enabling one core with and without this bug fix

| Application | Time w/ bug (sec) | Time w/o bug (sec) | Speedup factor (×) |
|---|---|---|---|
| bt | 122 | 23 | 5.24 |
| cg | 134 | 5.4 | 24.90 |
| ep | 72 | 18 | 4.0 |
| ft | 110 | 14 | 7.69 |
| is | 283 | 53 | 5.36 |
| lu | 2196 | 16 | 137.59 |
| mg | 81 | 9 | 9.03 |
| sp | 109 | 12 | 9.06 |
| ua | 906 | 14 | 64.27 |

From Lozi et al.

# Lock Contention

# Conclusions

# References


1. K. Kumar, P. Rajiv, 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 (PACT), pages 289–300, Aug 2014.
2. J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova.
   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, New York, NY, USA, 2016. ACM.
3. U. B. Nisar, M. Aleem, M. A. Iqbal, and N. S. Vo.
   Jumbler: A lock-contention aware thread scheduler for multi-core parallel machines.
   In 2017 International Conference on Recent Advances in Signal Processing, Telecommunications Computing (SigTelCom), pages 77–81, Jan 2017.