CS 477:

Advanced Operating Systems

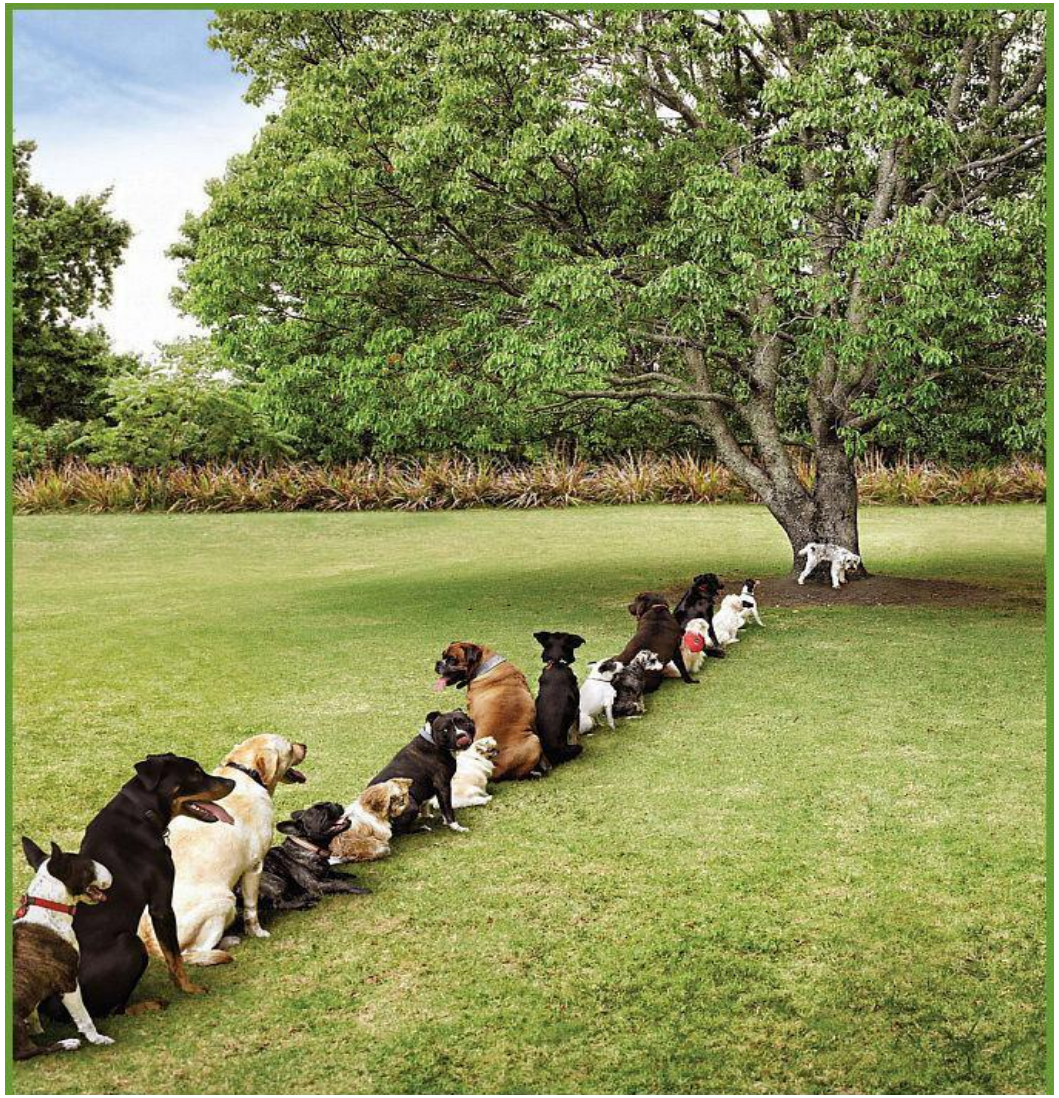
Scheduling I

Administrivia

This week

- Scheduling overview
- Linux CFS
- Issues with Linux CFS

Focus of today's lecture: Scheduling …



PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command (merged)
2887	root	20	0	2520	92	0	S	0.0	0.0	0:00.00	fusermount3 -o rw,nosuid,nodev,fsname=portal,auto_unmount,subtype=portal --
333224	root	20	0	1802M	19440	7348	S	0.0	0.1	0:05.93	snapd
545104	root	20	0	1802M	19440	7348	S	0.0	0.1	0:00.08	snapd
3612896	root	20	0	6558M	54500	10412	S	0.0	0.2	0:07.98	nordvpnd
1721	rtkit	21	1	88564	296	0	S	0.0	0.0	0:06.52	rtkit-daemon
1723	rtkit	20	0	88564	296	0	S	0.0	0.0	0:03.40	rtkit-daemon
1724	rtkit	RT	1	88564	296	0	S	0.0	0.0	0:03.00	rtkit-daemon
2163	sanidhya	20	0	20100	7728	4624	S	0.0	0.0	1:22.00	systemd --user
2167	sanidhya	20	0	168M	4296	0	S	0.0	0.0	0:00.00	(sd-pam)
2174	sanidhya	20	0	103M	5072	1456	S	0.0	0.0	8:39.37	appimagelauncherd
2179	sanidhya	20	0	449M	4512	2680	S	0.0	0.0	0:14.00	gnome-keyring-daemon --foreground --components=pkcs11,secrets --control-dir
2184	sanidhya	20	0	449M	4512	2680	S	0.0	0.0	0:00.00	pool-spawner gnome-keyring-daemon --foreground --components=pkcs11,secrets
2185	sanidhya	20	0	449M	4512	2680	S	0.0	0.0	0:00.00	gmain gnome-keyring-daemon --foreground --components=pkcs11,secrets --contr
2186	sanidhya	20	0	449M	4512	2680	S	0.0	0.0	0:07.08	gdbus gnome-keyring-daemon --foreground --components=pkcs11,secrets --contr
2187	sanidhya	20	0	449M	4512	2680	S	0.0	0.0	0:00.00	timer gnome-keyring-daemon --foreground --components=pkcs11,secrets --contr
2188	sanidhya	20	0	13392	6412	1376	S	0.0	0.0	1:57.66	dbus-daemon --session --address=systemd: --nofork --nopidfile --systemd-act
2192	sanidhya	20	0	226M	376	0	S	0.0	0.0	0:00.00	gdm-x-session --run-script /usr/bin/gnome-session
2195	sanidhya	20	0	226M	376	0	S	0.0	0.0	0:00.00	gdm-x-session --run-script /usr/bin/gnome-session
2256	sanidhya	20	0	226M	376	0	S	0.0	0.0	0:00.00	gdm-x-session --run-script /usr/bin/gnome-session
2257	sanidhya	20	0	226M	376	0	S	0.0	0.0	0:00.00	gdm-x-session --run-script /usr/bin/gnome-session
2258	sanidhya	20	0	363M	1944	0	S	0.0	0.0	0:00.06	gnome-session-binary
2289	sanidhya	20	0	308M	1600	172	S	0.0	0.0	0:00.60	gvfsd
2290	sanidhya	20	0	308M	1600	172	S	0.0	0.0	0:00.00	gvfsd
2291	sanidhya	20	0	308M	1600	172	S	0.0	0.0	0:00.00	gvfsd
2292	sanidhya	20	0	308M	1600	172	S	0.0	0.0	0:00.31	gvfsd
2295	sanidhya	20	0	442M	600	0	S	0.0	0.0	0:00.01	gvfsd-fuse /run/user/1000/gvfs -f
2297	sanidhya	20	0	442M	600	0	S	0.0	0.0	0:00.00	gvfsd-fuse /run/user/1000/gvfs -f
2298	sanidhya	20	0	442M	600	0	S	0.0	0.0	0:00.00	gvfsd-fuse /run/user/1000/gvfs -f
2299	sanidhya	20	0	442M	600	0	S	0.0	0.0	0:00.00	gvfsd-fuse /run/user/1000/gvfs -f
2300	sanidhya	20	0	442M	600	0	S	0.0	0.0	0:00.00	gvfsd-fuse /run/user/1000/gvfs -f
2301	sanidhya	20	0	442M	600	0	S	0.0	0.0	0:00.00	gvfsd-fuse /run/user/1000/gvfs -f
2305	sanidhya	20	0	442M	600	0	S	0.0	0.0	0:00.00	gvfsd-fuse /run/user/1000/gvfs -f
2306	sanidhya	20	0	373M	800	4	S	0.0	0.0	0:00.16	at-spi-bus-launcher
2307	sanidhya	20	0	373M	800	4	S	0.0	0.0	0:00.00	at-spi-bus-launcher
2308	sanidhya	20	0	373M	800	4	S	0.0	0.0	0:00.00	at-spi-bus-launcher
2309	sanidhya	20	0	373M	800	4	S	0.0	0.0	0:00.00	at-spi-bus-launcher
2311	sanidhya	20	0	373M	800	4	S	0.0	0.0	0:00.12	at-spi-bus-launcher
2313	sanidhya	20	0	8912	852	0	S	0.0	0.0	0:12.12	dbus-daemon --config-file=/usr/share/defaults/at-spi2/accessibility.conf --
2324	sanidhya	20	0	363M	1944	0	S	0.0	0.0	0:00.00	gnome-session-binary
2325	sanidhya	20	0	363M	1944	0	S	0.0	0.0	0:00.00	gnome-session-binary
2326	sanidhya	20	0	363M	1944	0	S	0.0	0.0	0:00.00	gnome-session-binary

OS scheduler

- Decides **which** process runs next, **when**, and **for how long**
- Responsible for making the best use of processor (CPU)
  - E.g., Do not waste CPU cycles for waiting processes
  - E.g., Give higher priority to higher-priority processes
  - E.g., Do not starve low-priority processes
- OS scheduler has two parts:
  - **Mechanism** to stop one running process and star running another process
  - **Policy** to pick up which process to run

Context switch

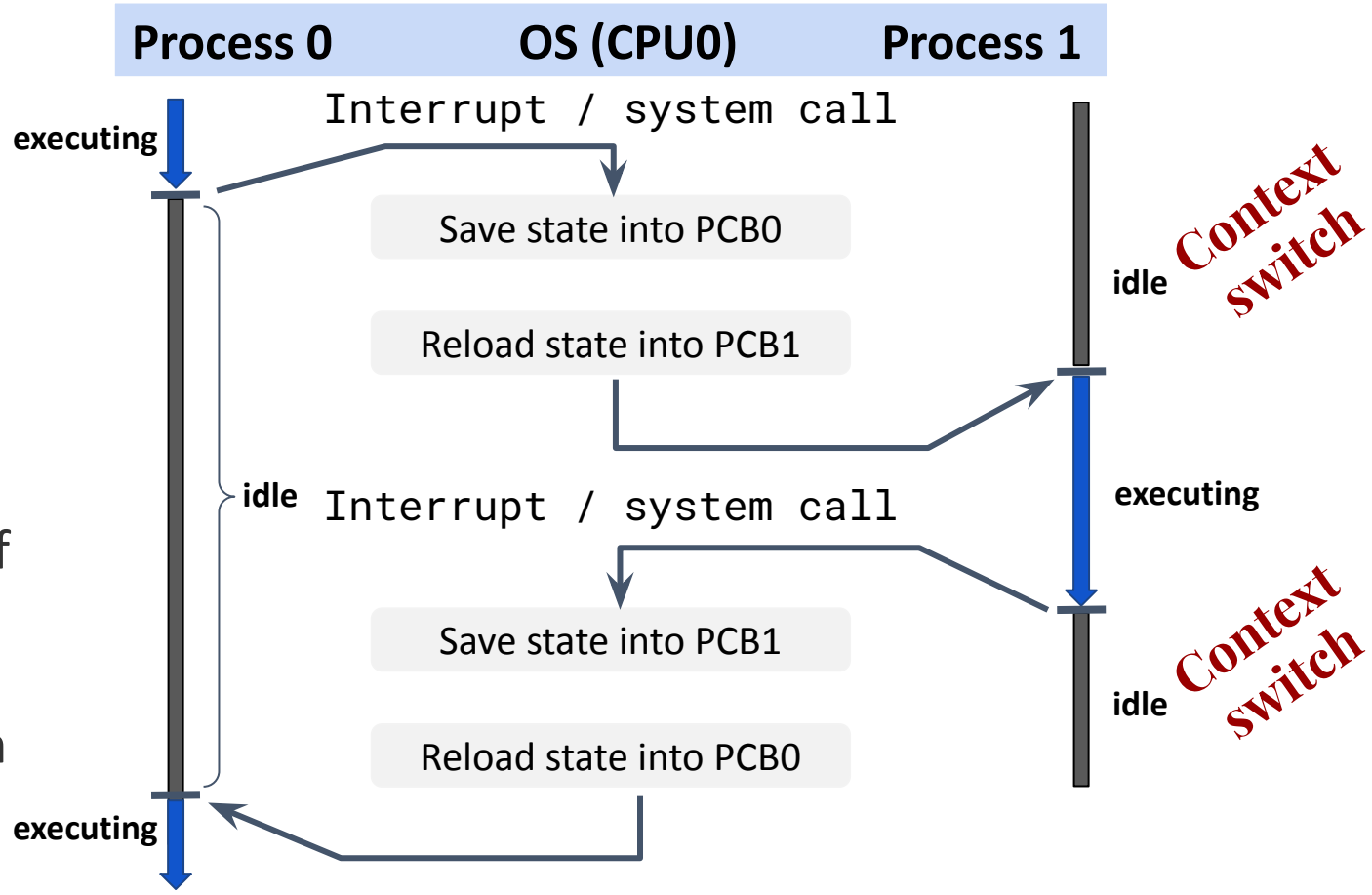
A context switch is a mechanism that allows the OS to store the current process state and switch to some other, previously stored context.

- The context of the process is represented in the process control block (PCB)
- The OS maintains the PCB for each process
- The process **context** includes hardware registers
  - Ex: All x86 registers and the memory registers for process-specific memory region

Context switch procedure

The OS does the following operations during the context switch:

1. **Saves** the running process' execution state in the PCB
2. **Selects** the next thread
3. **Restores** the execution state of the next process
4. **Passes** the control using return from trap to resume next process



PCB: Process control block

IO vs. CPU-bound tasks

- Scheduling policy: a set of rules determining **what runs when**
- **IO-bound processes**
  - Spend most time waiting for IO: disk, network, keyboard, mouse, etc.
  - Runs only for a short duration
  - Response time is important
- **CPU-bound processes**
  - Heavy use of the CPU: MATLAB, scientific computation, etc.
  - Caches stay hot when they run for a long time

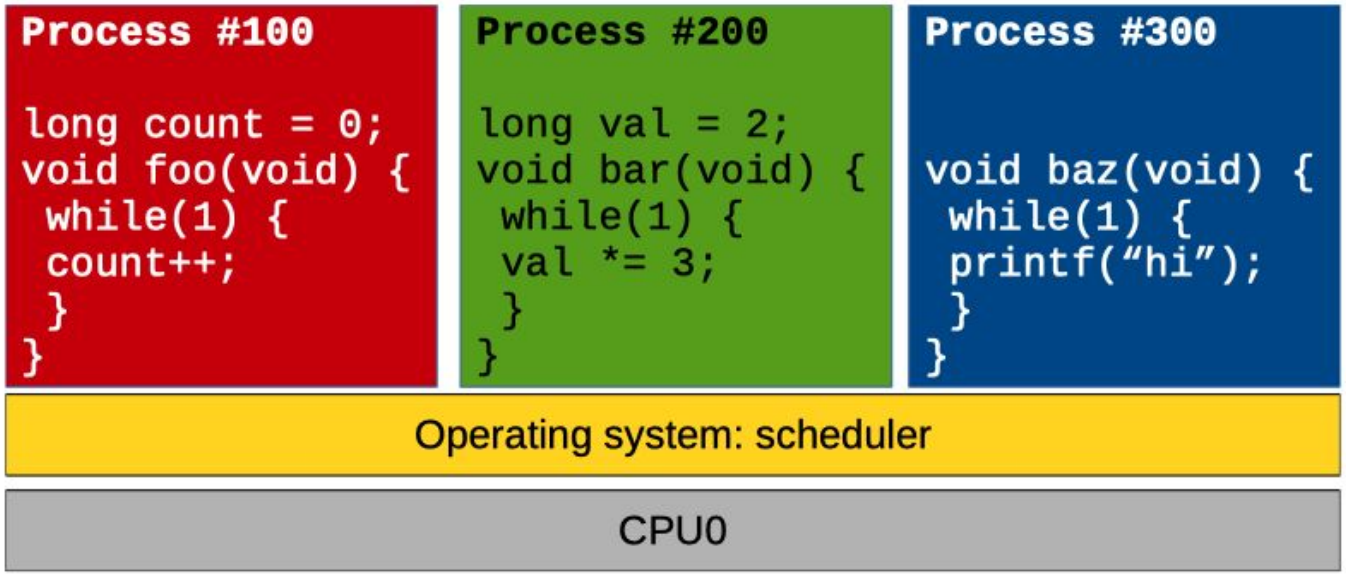
Multitasking

- Simultaneously interleave execution of more than one process
  - Single core
    - The OS scheduler gives illusion of multiple processes running concurrently
  - Multi-core
    - The processor scheduler enables true parallelism

Types of multitasking OS

- **Cooperative multitasking:** old OSes (e.g., Windows 3.1) and few language runtimes (e.g., Go runtime)
  - A process does not stop running until it decides to **yield** CPU
  - The OS cannot enforces fair scheduling
- **Preemptive multitasking:** almost all modern OSes
  - The OS can interrupt the execution of a process (i.e., **preemption**) after the process expires its **timeslice**, which is decided by **process priority**

Types of multitasking OS



Q. How can the preemptive scheduler take the control of infinite loop?

Preemption for process scheduling

- Preemptive scheduler relies on **hardware timer**
- OS sets a timer before scheduling a process
  - Hardware generates an interrupt after the timer expires
  - It interrupts process execution
  - Interrupts lead to switching to the kernel mode
  - OS decides if the process may continue

Process priority

- **Priority-based scheduling**
  - Rank processes based on their worth and need for processor time
  - Processes with a higher priority run before those with a lower priority

Linux process priority

- **Linux has two priority ranges**
  - Nice value: ranges from -20 to +19 (default 0)
    - High values of nice means lower priority
  - Real-time priority: ranges from 0 to 99
    - Higher values mean higher priority
    - Real-time processes always execute before standard process (nice) processes

ps ax -eo pid,ni,rtprio,cmd

Linux process priority

- **Linux has two priority ranges**
  - Nice value: ranges from -20 to +19 (default 0)
    - High values of nice means lower priority
  - Real-time priority: ranges from 0 to 99
    - Higher values mean higher priority
    - Real-time processes always execute before standard process (nice) processes

ps ax -eo pid,ni,rtprio,cmd

Scheduling policy: timeslice

- How much time a process should execute before being preempted
- Defining the default time slice in an absolute way is tricky:
  - Too long → bad interactive performance
  - Too short → high context switch overhead

Scheduling policy: example

- Two tasks in the system:
  - Text editor: IO-bound, latency sensitive (interactive)
  - Video encoder: CPU-bound, background job
- Scheduling goal
  - Text editor: when ready to run, need to preempt the video encoder for good **interactive performance**
  - Video encoder: run as long as possible for better CPU *cache utilization*

Scheduling policy: example in UNIX systems

- Two tasks in the system:
  - Text editor: IO-bound, latency sensitive (interactive)
  - Video encoder: CPU-bound, background job
- Gives higher priority to the text editor
- Not because it needs a lot of processor but because we want it to always have processor time available when it needs

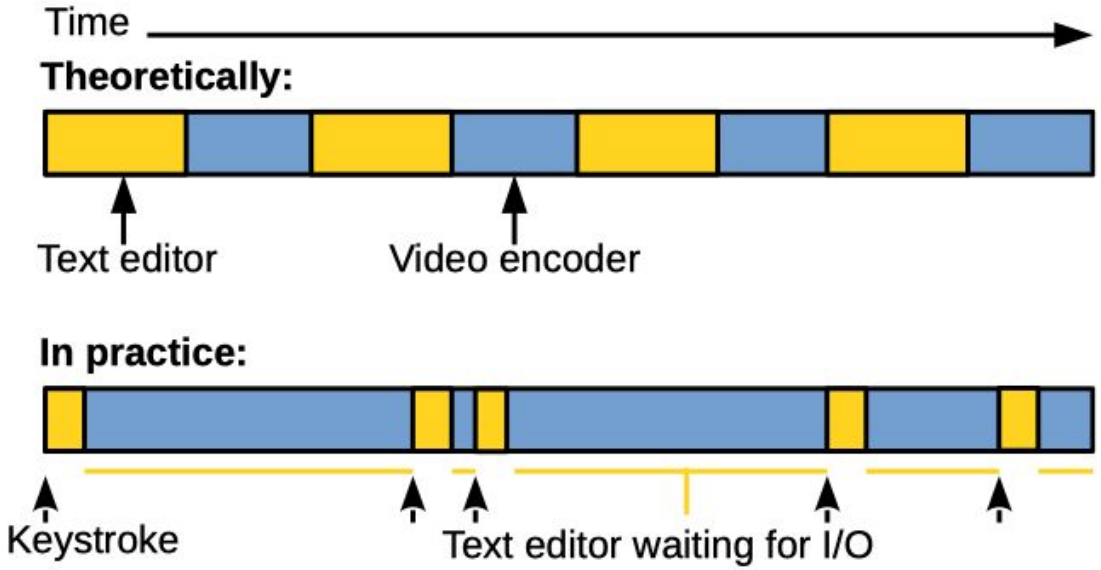
Scheduling policy: timeslice in Linux CFS

- **Linux CFS does not use an absolute timeslice**
  - The timeslice a process receives is a function of the load of the system (i.e., a proportion of the CPU)
  - In addition, that timeslice is weighted by the process priority
  - When a process **P** becomes runnable:
    - **P** will preempt the currently running process **C** if **P** consumed a smaller proportion of the CPU than **C**

Scheduling policy: example in Linux CFS

- CFS guarantees the text editor a specific proportion of CPU time
  - CFS keeps track of the actual CPU time used by each program
- E.g., text editor: video encode = 50% : 50%
  - The text editor mostly sleeps, waiting for user's input and the video encoder keeps running until preempted
  - When the text editor wakes up
    - CFS sees that the text editor actually used less CPU time than the video encoder
    - The text editor preempts the video encoder

Scheduling policy: example in Linux CFS



- Good interactive performance
- Good background, CPU-bound performance

Linux CFS design

- Completely fair scheduler (CFS)
- At each moment, each process of the same priority has received an exact same amount of the CPU time
- If we could run **n** tasks in parallel on the CPU, give each **1/n** of the CPU processing power
- CFS runs a process for some time, then swaps it for the runnable process that has run the least

Linux CFS design

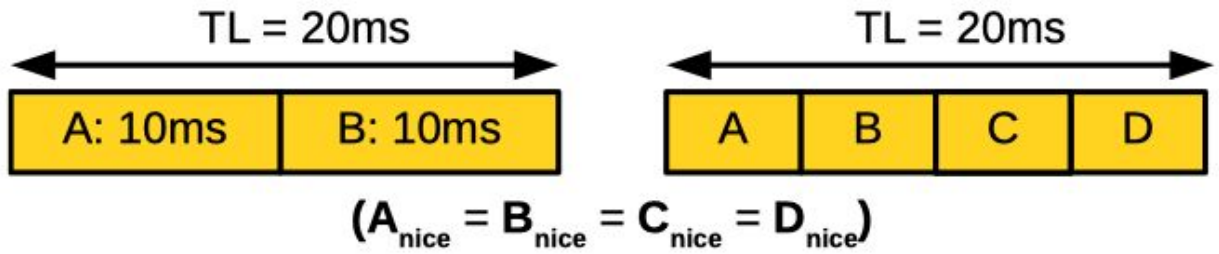
- **Goal:**
  - Fairly divide a CPU evenly among all competing processes with a clean implementation
- **Basic idea:**
  - **Virtual runtime (vruntime):** When a process runs, it accumulates "virtual time." If priority is high, virtual time accumulates slowly. If priority is low, virtual time accumulates quickly
  - It is a "catch up" policy—processes with smallest amount of virtual time gets to run next

Linux CFS design

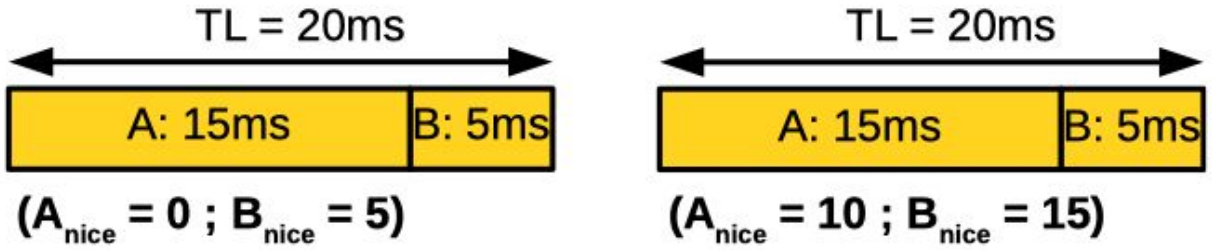
- **No default timeslice**, CFS calculates how long a process should run according to the number of runnable processes
  - That dynamic timeslice is weighted by the process priority (nice)
  - timeslice = weight of the task / total weight of the runnable tasks
- To calculate the actual timeslice, CFS sets a **targeted latency**
  - Targeted latency: period during which all runnable processes should be scheduled at least once
  - Minimum granularity: floor at 1 ms (default)

Linux CFS design

- Example: process with same priority



- Example: process with different priorities



Linux CFS design

- Scheduler maintains a red-black tree where nodes are ordered according to received virtual execution time
- Node with smallest virtual execution time is picked next
- Priorities determine accumulation rate of virtual execution time
  - Higher priority → slower accumulation rate

Linux CFS example

- Three processes A, B, C accumulate virtual time at a rate 1, 2, and 3 respectively
- What is the expected share of the CPU that each process gets?

Linux CFS example

- Three processes A, B, C accumulate virtual time at a rate 1, 2, and 3 respectively
- What is the expected share of the CPU that each process gets?
  - **Strategy**: How many quantums required for

    all clocks to be equal?
    - Lowest common multiple: 6
    - To reach VT = 6 …
      - A is scheduled 6 times
      - B is scheduled 3 times
      - C is scheduled 2 times
    - A => 6 / 11 of CPU time
    - B => 3 / 11 of CPU time
    - C => 2 / 11 of CPU time

Linux CFS example

- Three processes A, B, C accumulate virtual time at a rate 1, 2, and 3 respectively
- What is the expected share of the CPU that each process gets?
  - **Strategy**: How many quantums required for all clocks to be equal?
    - Lowest common multiple: 6
    - To reach VT = 6 …
      - A is scheduled 6 times
      - B is scheduled 3 times
      - C is scheduled 2 times
    - A => 6 / 11 of CPU time
    - B => 3 / 11 of CPU time
    - C => 2 / 11 of CPU time

Q01: A => {A:1, B:0, C:0}

Q02: B => {A:1, B:2, C:0}

Q03: C => {A:1, B:2, C:3}

Q04: A => {A:2, B:2, C:3}

Q05: B => {A:2, B:4, C:3}

Q06: A => {A:3, B:4, C:3}

Q07: A => {A:4, B:4, C:3}

Q08: C => {A:4, B:4, C:6}

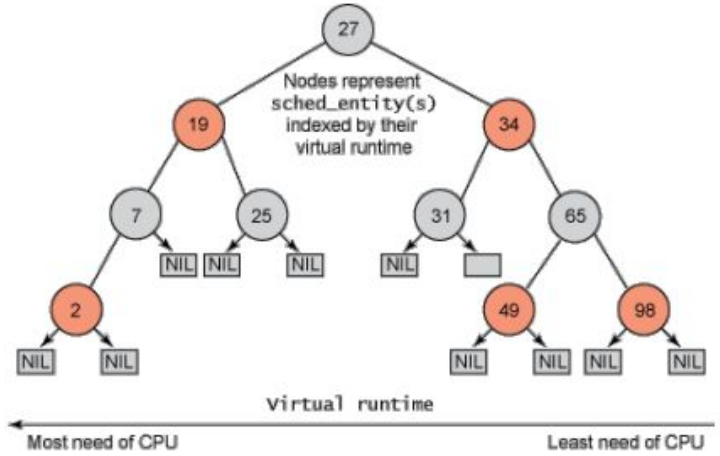
Q09: A => {A:5, B:4, C:6}

Q10: B => {A:5, B:6, C:6}

Q11: A => {A:6, B:6, C:6}

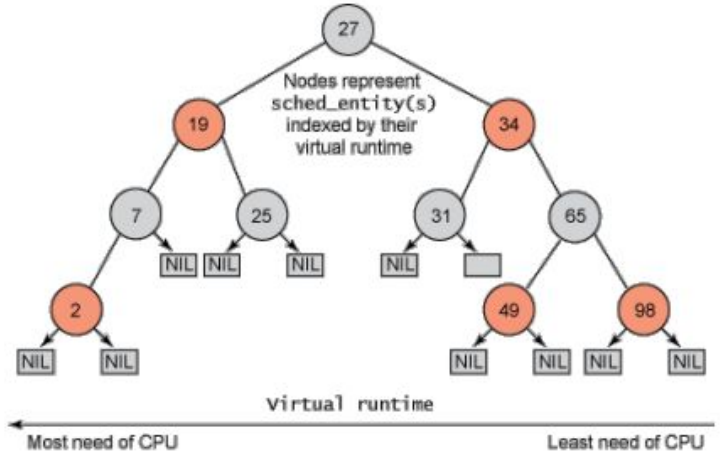
Linux CFS: Red-black tree

- CFS maintains a red-black tree for processes running on a local CPU:
  - An RB tree is a BST with constraints:
    1. Each node is red or black
    2. Root node is black
    3. All leaves (NIL) are black
    4. If node is red, both children are black
    5. Every path from a given node to its descendent NIL leaves contains the same number of black number of black nodes



Linux CFS: Red-black tree

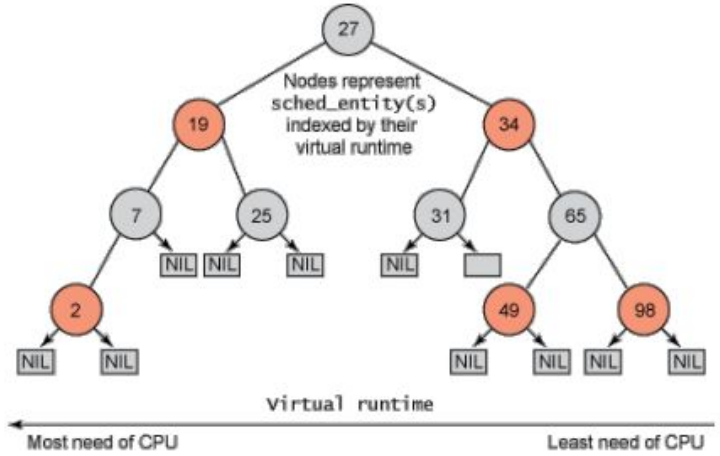
- CFS maintains a red-black tree for processes running on a local CPU:
  - An RB tree is a BST with constraints:
    1. Each node is red or black
    2. Root node is black
    3. All leaves (NIL) are black
    4. If node is red, both children are black
    5. Every path from a given node to its descendent NIL leaves contains the same number of black number of black nodes



In an RB tree, the path from the root to the farthest leafs no more than twice as long as the path from the root to the nearest leaf

Linux CFS: Red-black tree

- Benefits over run queue:
  - O(1) access to the leftmost node
    - Can access process with lowest virtual time
  - O(log n) insert
  - O(log n) delete
  - Self-balancing data structure



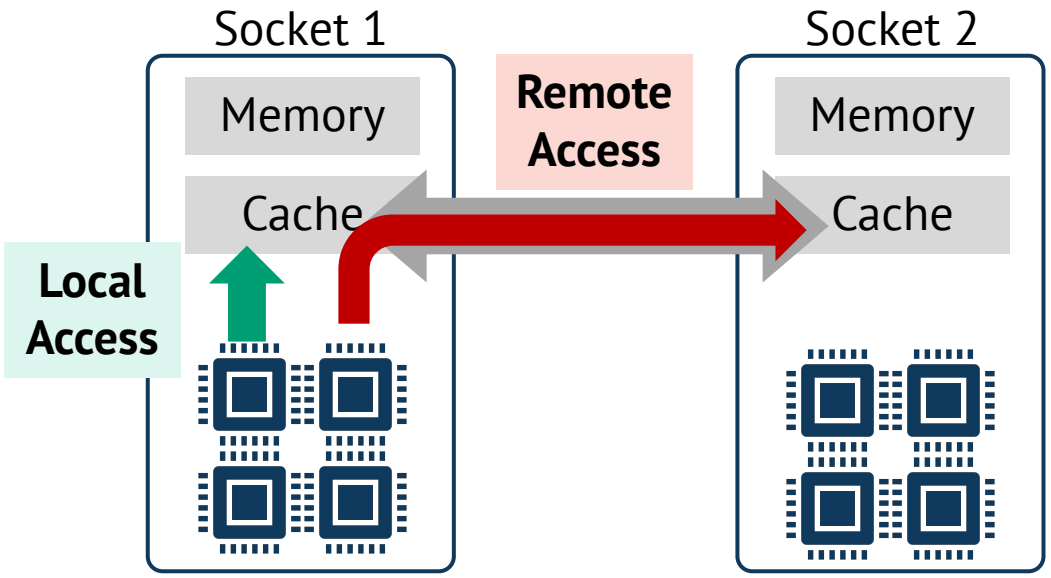
Load balancing

- More like symmetric multi processing: each CPU is self-scheduling
- Two general approaches to balancing:
  - **Push migration:** Process routinely checks the load on each CPU and redistributes processes between CPUs on detecting imbalance
  - **Pull migration:** Idle CPU can actively pull waiting tasks from a busy CPU
- In multi-core environment:
  - Each CPU has its own runqueue

Load balancing

- Load balancing works at the level of **scheduling domains**
  - Creates the hierarchy in a bottom up manner
  - Cores, core pairs, socket, collection of sockets
- Linux creates these domains when it boots up
- The scheduling algorithm works hierarchically, balancing from among cores, to sockets, to other scheduling domains
- Also keeps track of caches when a process is woken up

NUMA: Non-uniform memory access



Accessing local socket data is faster than remote socket data

How / when to preempt?

- Kernel sets the **need_resched()** flag (per CPU variable) at various locations
  - scheduler_tick(): a process used up its timeslice
  - try_to_wake_up(): higher priority process woken up
- Kernel checks need_reched() at certain points, if safe, **schedule()** is invoked
- User preemption:
  - Return to user space from a system call or from an interrupt handler
- Kernel preemption:
  - A process in the kernel explicitly calls schedule()
  - A process in the kernel blocks on events (which internally calls schedule())

Kernel preemption

- In most UNIX-like OSes, kernel code is **non-preemptive**
- In Linux, the kernel code can also be preemptive
  - A process can be preempted in the kernel as long as execution is in a safe state without holding any lock
- Kernel maintains preemption count information as well as thread_info in the task_struct to figure out if a lock is held

Real-time scheduling policies

- Linux also exposes some **soft real-time** scheduling policies:
  - **SCHED_FIFO**, **SCHED_RR**, **SCHED_DEADLINE**
  - Best effort, no guarantee
- Real-time process of any scheduling policy will always run before non-realtime ones (CFS, **SCHED_BATCH**)

Real-time scheduling policies

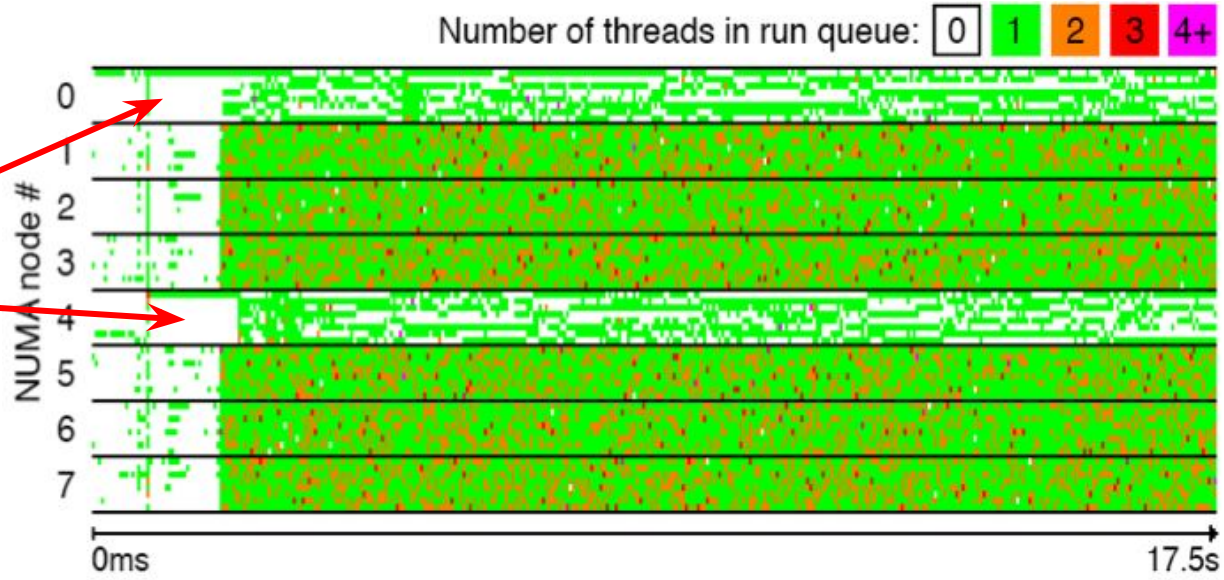
- SCHED_FIFO:
  - Task run until it blocks / yields
  - Only a higher priority RT task can preempt it
  - Round-robin for tasks of same priority
- SCHED_RR:
  - Same as SCHED_FIFO, but with a fixed timeslice

Real-time scheduling policies

- SCHED_DEADLINE:
  - Real-time policies for predictable RT scheduling
  - Early deadline first (EDF) scheduling based on a period of activation and a worst case execution time (WCET) for each task
- SCHED_BATCH:
  - Non-real time, low priority background jobs
- SCHED_IDLE:
    - Non-real time, very low priority background jobs

Consider a case

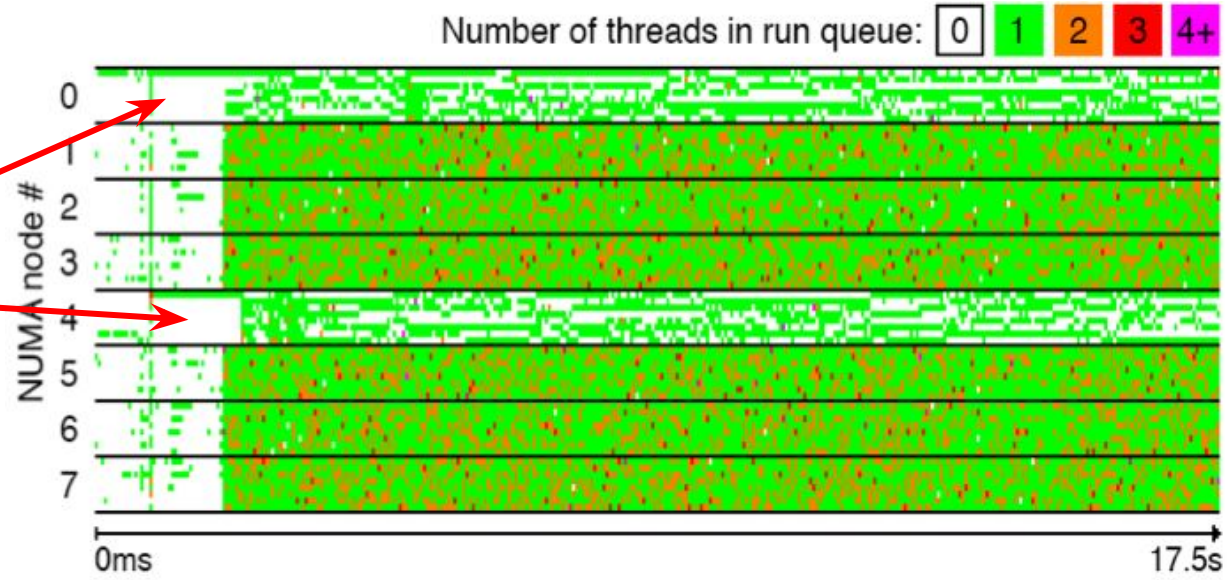
- 64 core machine
- Run 2 CPU intensive processes in two terminals and run kernel compilation
- Two NUMA nodes with many idle cores (white)



Consider a case

- 64 core machine
- Run 2 CPU intensive processes in two terminals and run kernel compilation

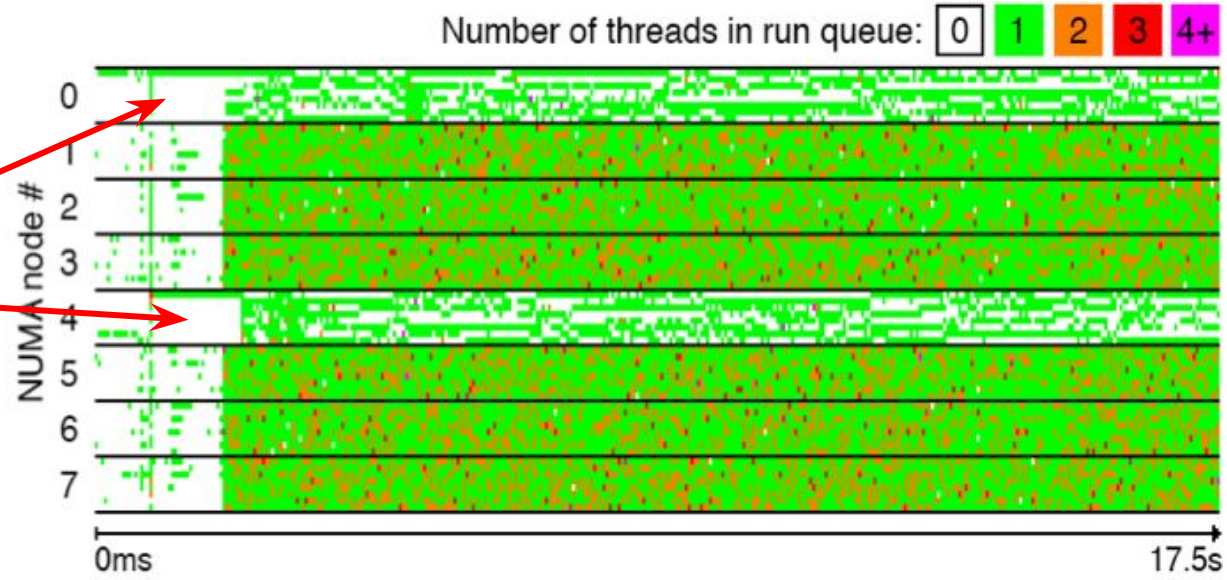
- Two NUMA nodes with many idle cores (white)
- Other NUMA nodes with many overloaded cores (orange, red)



Consider a case

- 64 core machine
- Run 2 CPU intensive processes in two terminals and run kernel compilation

- Two NUMA nodes with many idle cores (white)
- Other NUMA nodes with many overloaded cores (orange, red)



Scheduler is not work conserving!

Work conserving scheduler

- A scheduler always tries to make full use of system resources by ensuring that if any task that can be executed, it will not let system resources such as CPU to sit idle
- Pros:
  - Better resource utilization
  - No artificial idleness in the system

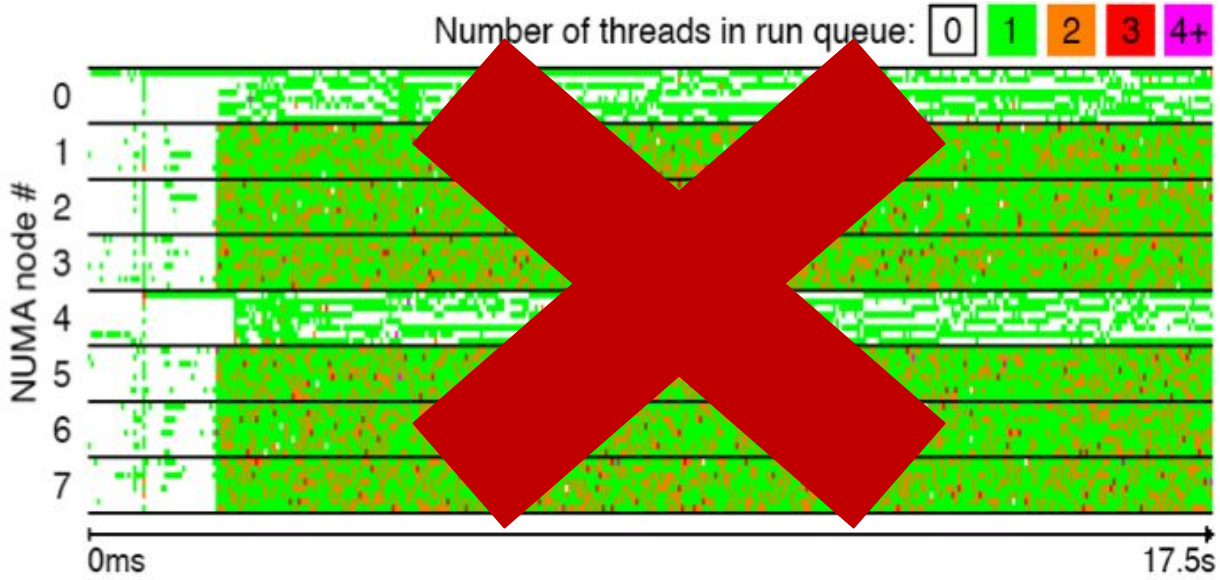
CFS is designed with work-conservation in mind!

CFS is non work conserving

- CFS violates the basic invariant for work conservation:

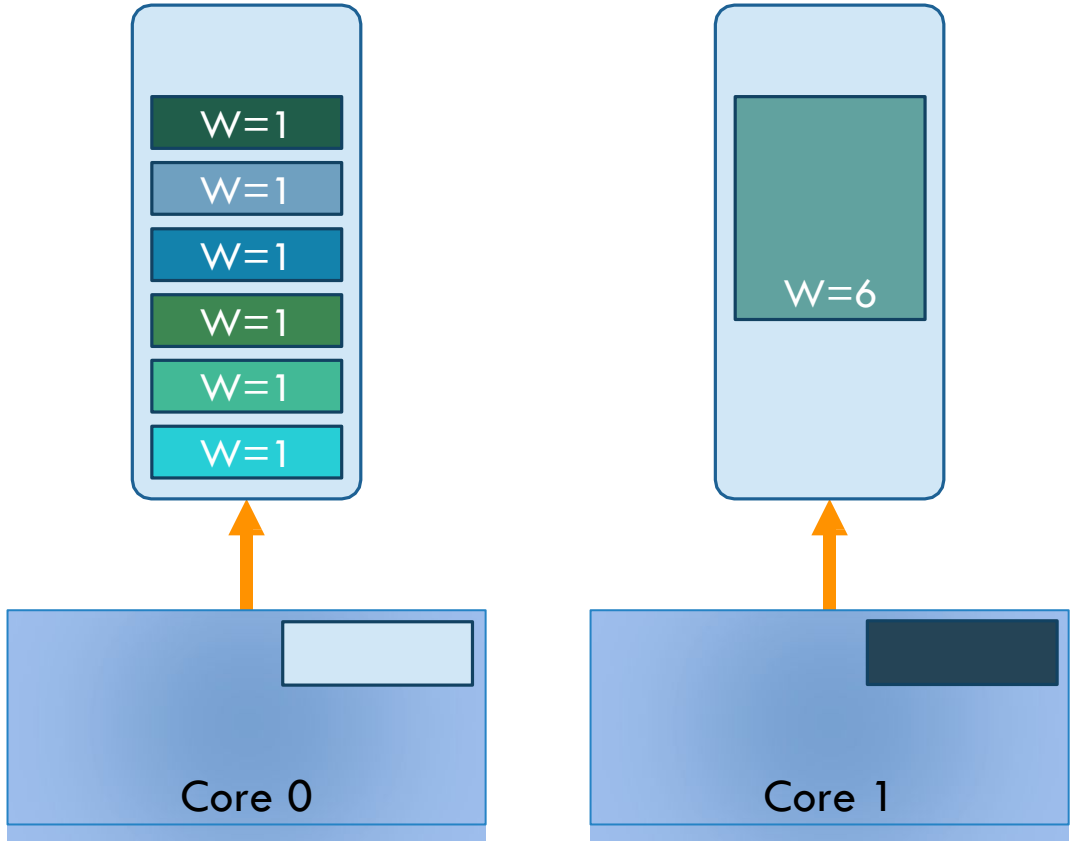
  No idle cores if some cores have several threads in their runqueues

- Can actually happen in transient situations

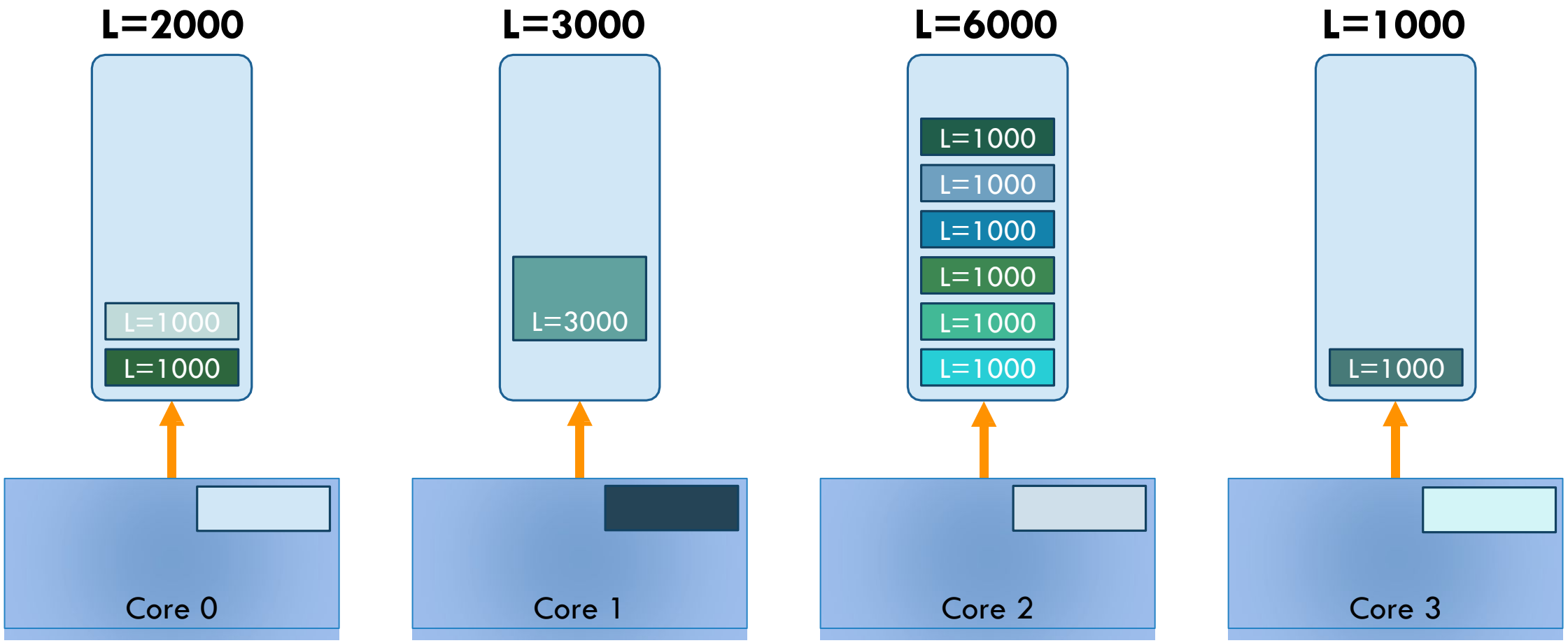


CFS in practice

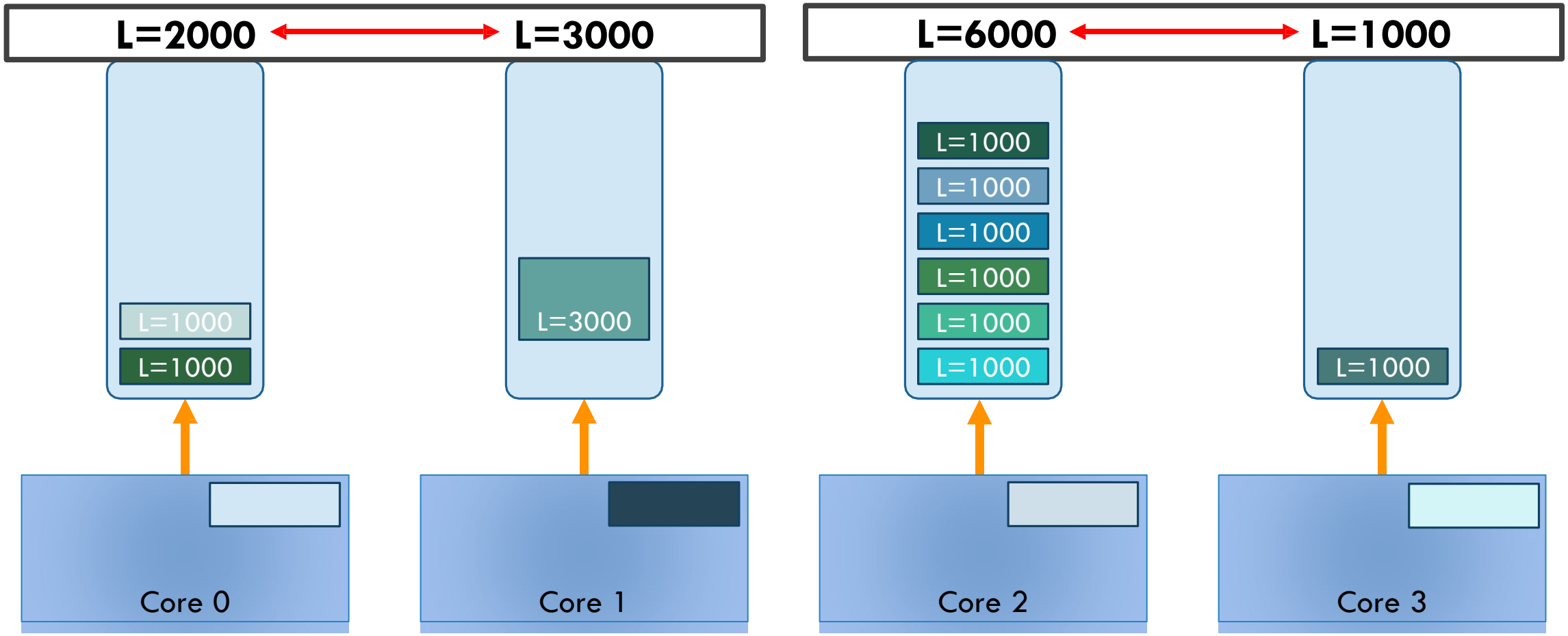
- One runqueue per core to avoid contention
- CFS periodically balances loads:
  - Load(task) = weight[1] x % CPU use[2]
    - 1. Lower niceness = higher weight
    - 2. Prevent high-priority thread from taking whole CPU just to sleep
- Hierarchical approach to balancing tasks



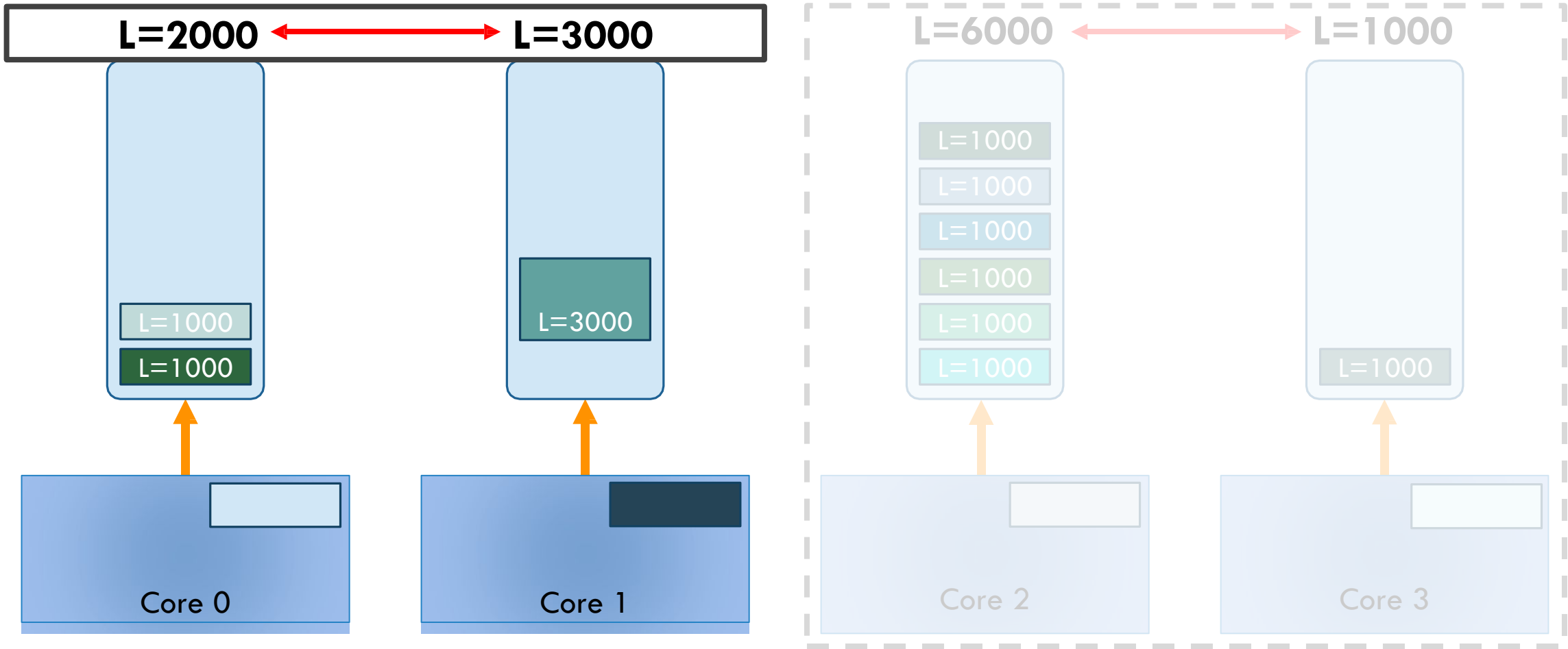
CFS load balancing



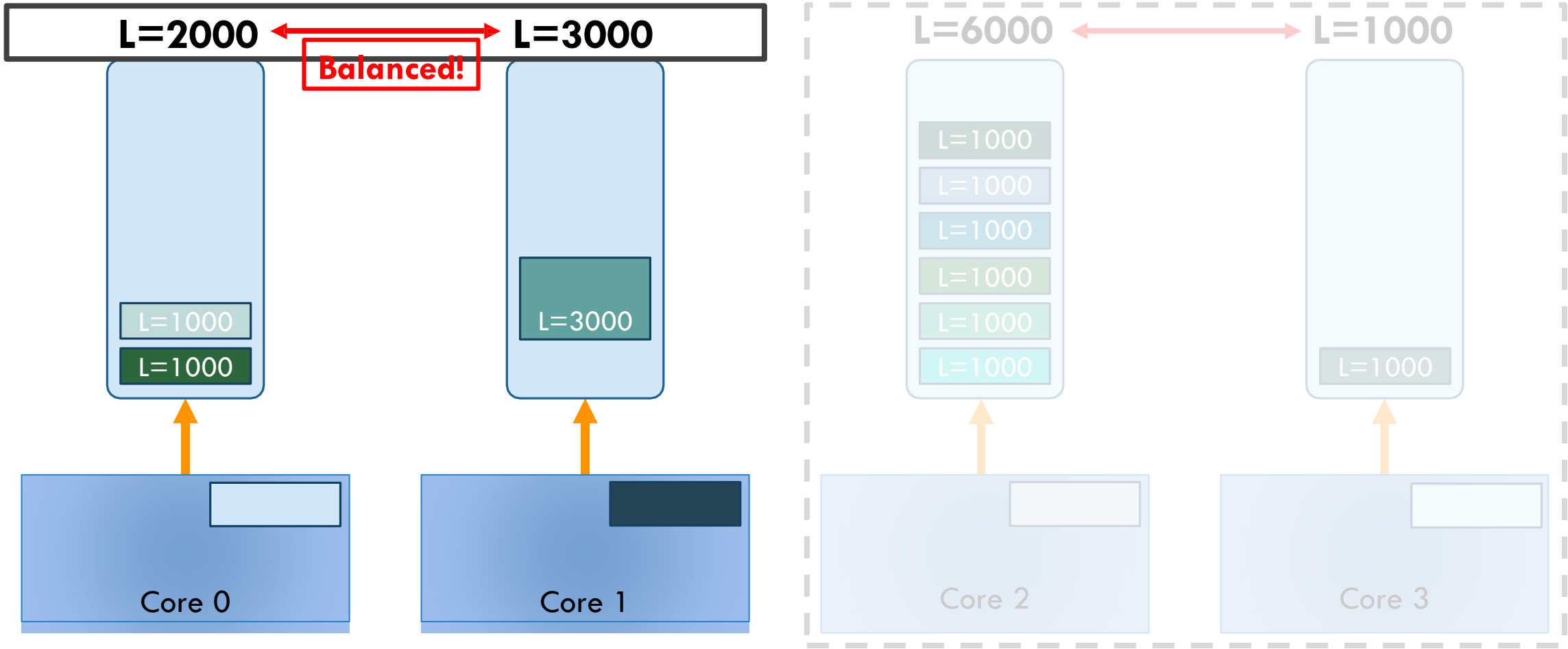
CFS load balancing



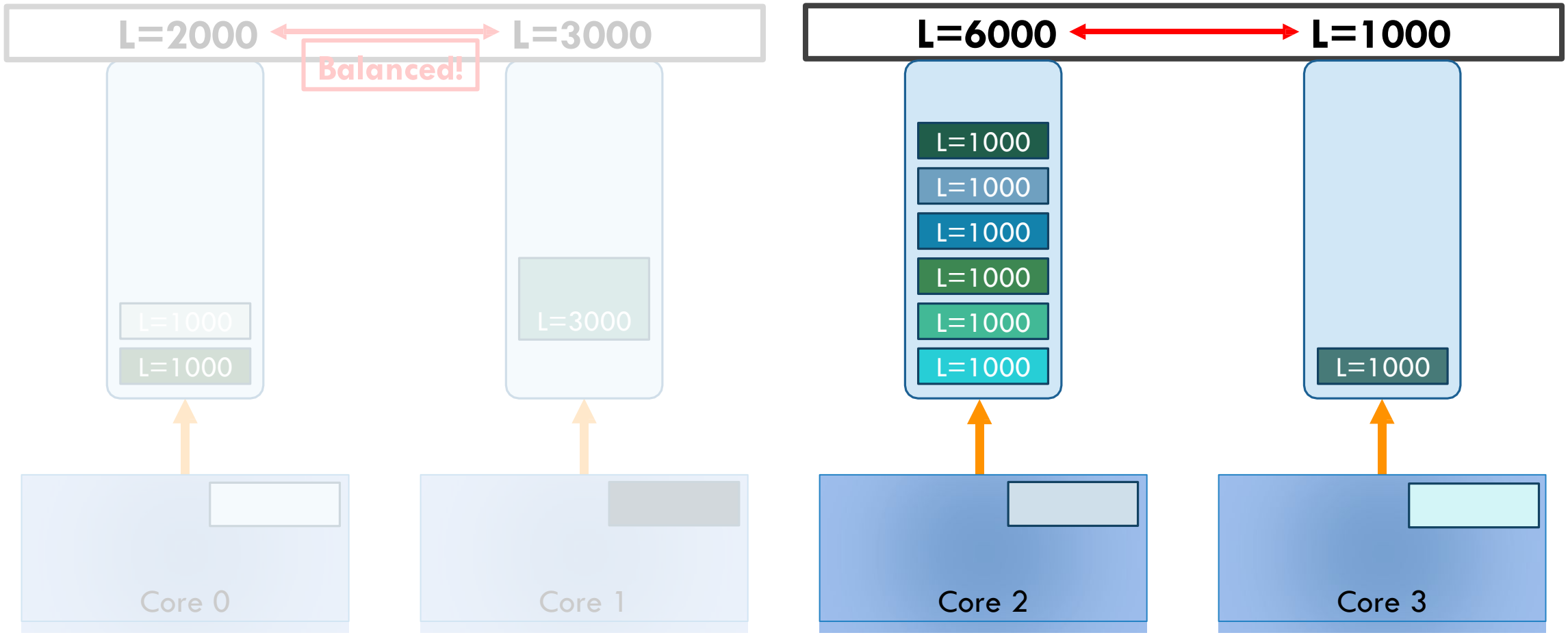
CFS load balancing



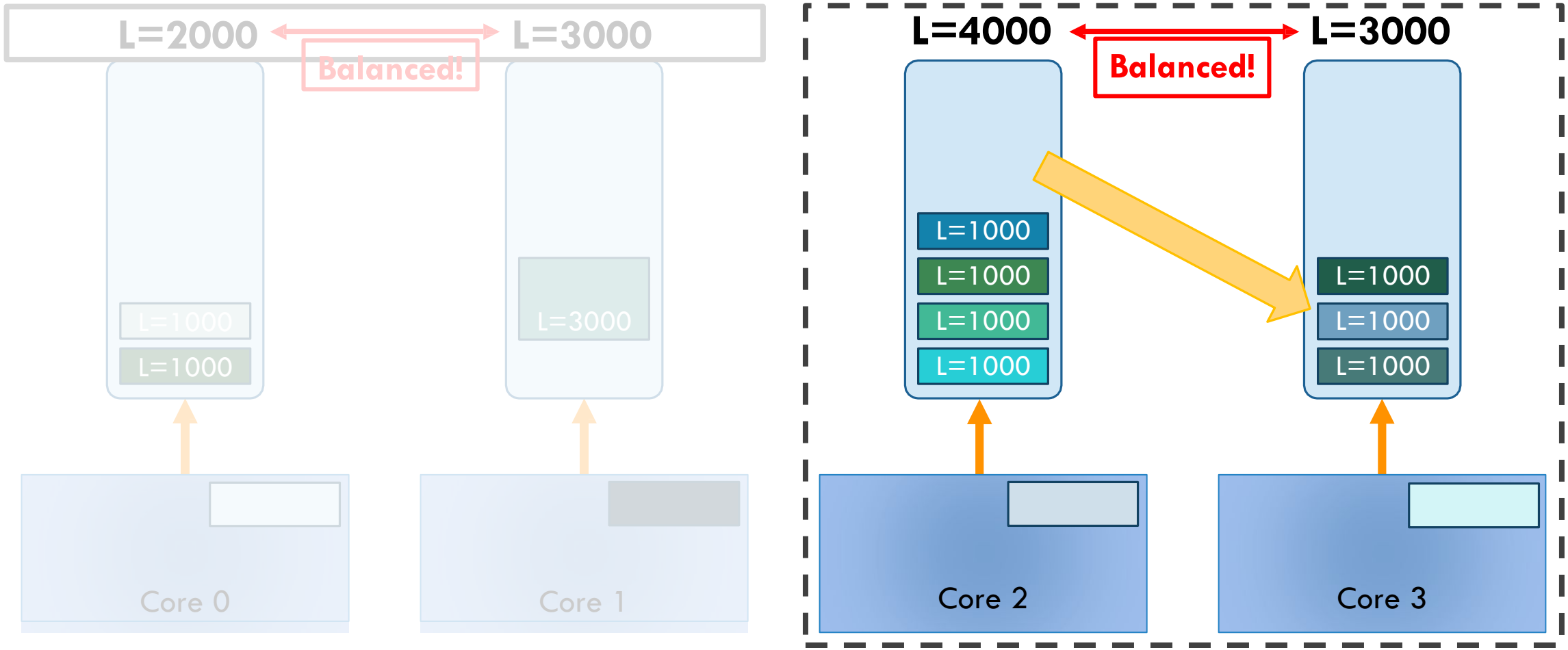
CFS load balancing



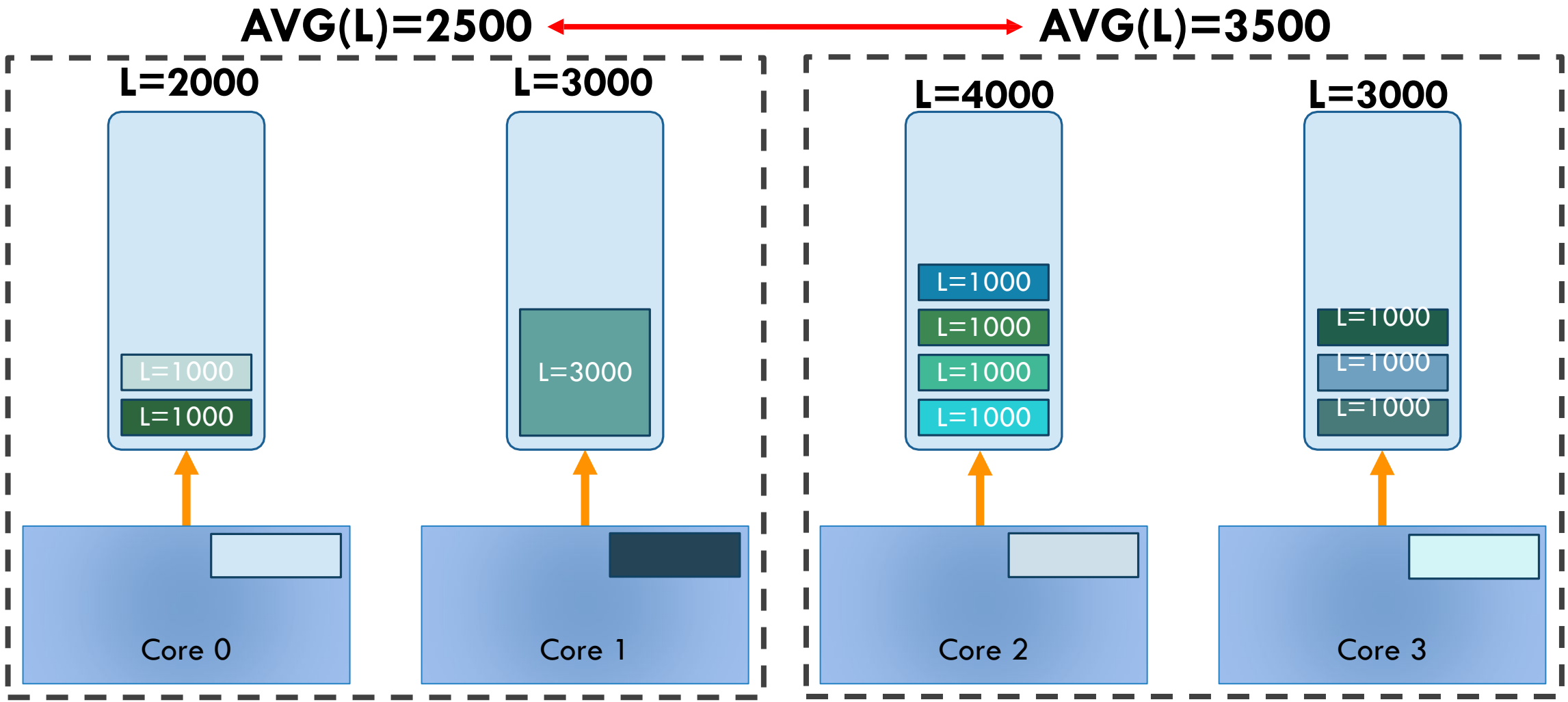
CFS load balancing



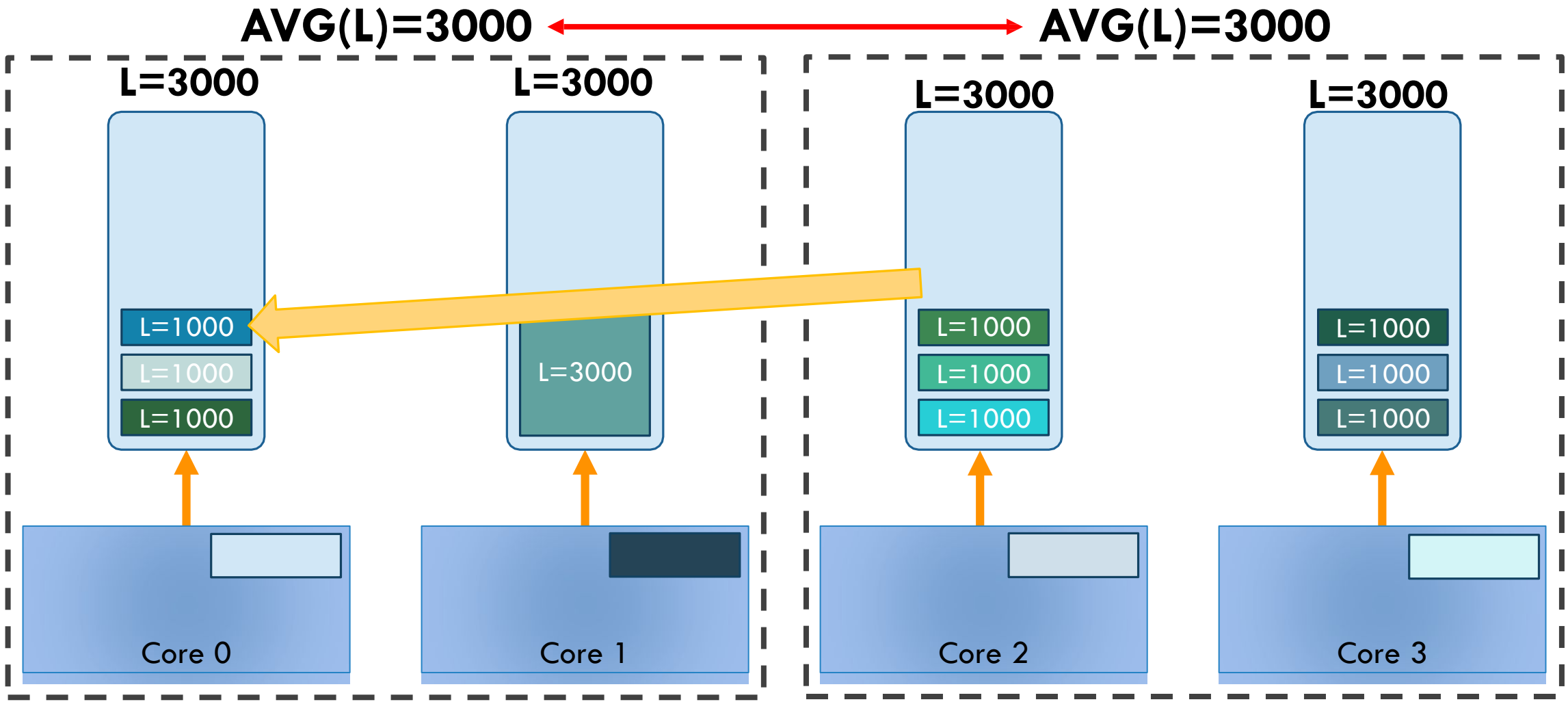
CFS load balancing



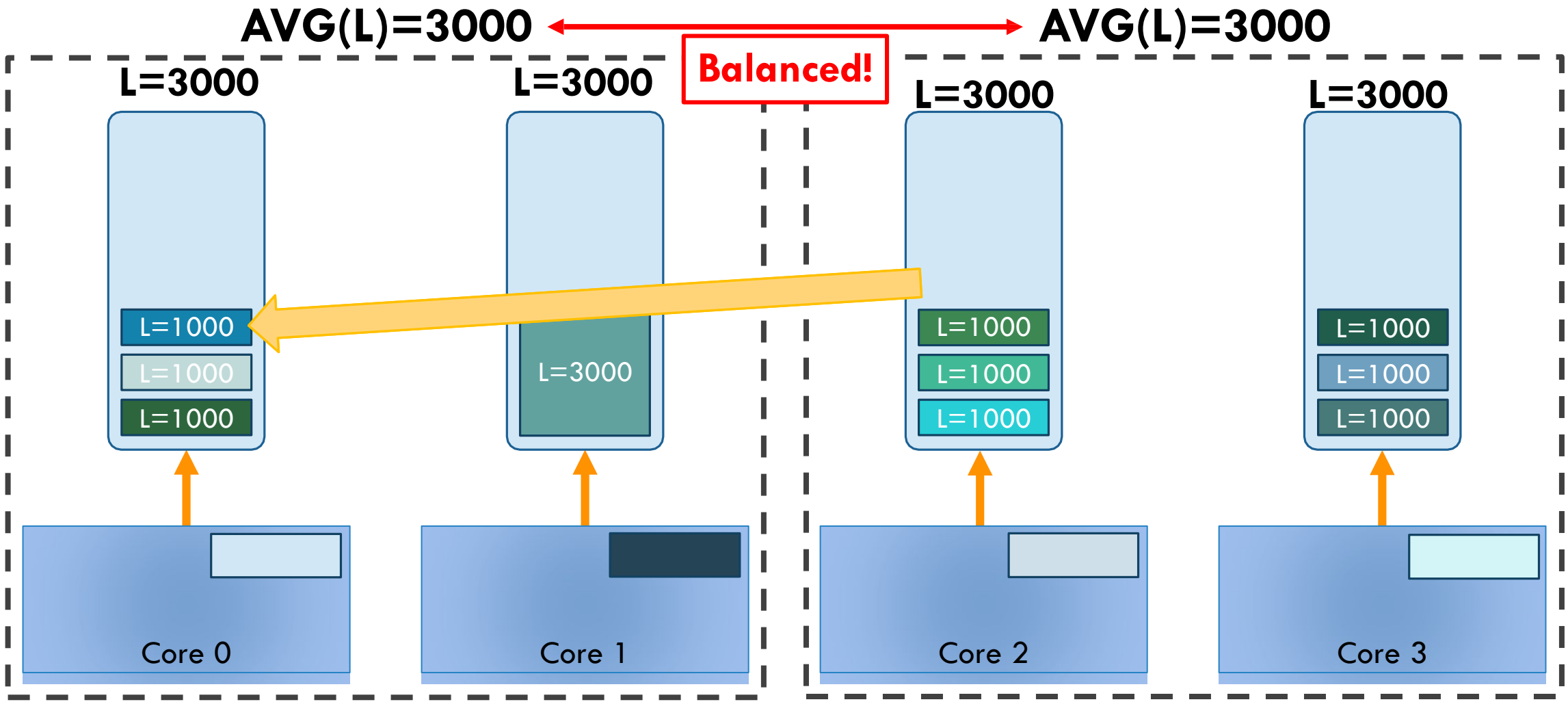
CFS load balancing



CFS load balancing



CFS load balancing

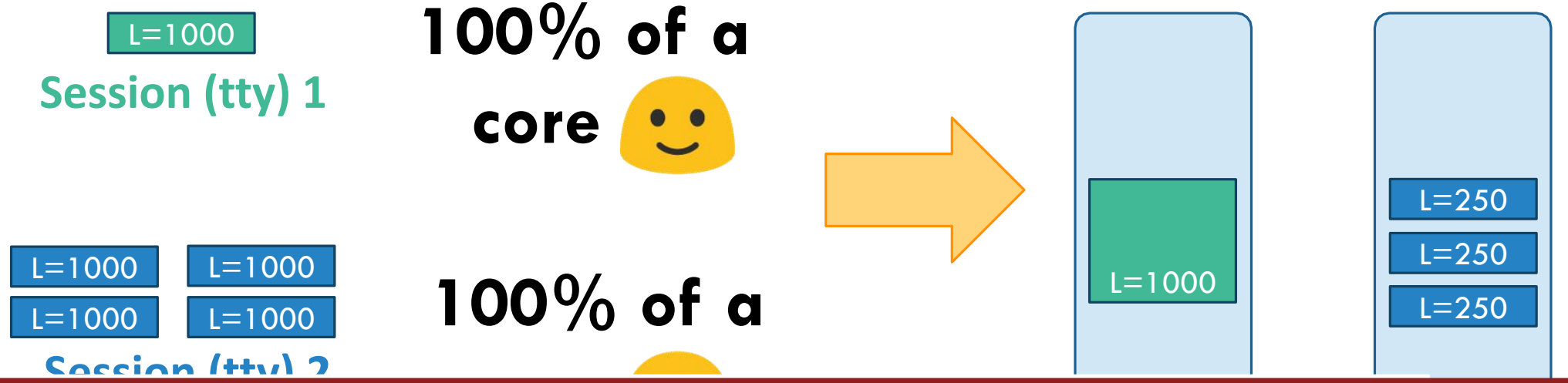


CFS load balancing

- Load calculations are actually more complicated, relies on heuristics
- **One of them aims to increase fairness between "sessions"**
- Idea: Ensure a **tty** cannot eat up all resources by spawning several threads

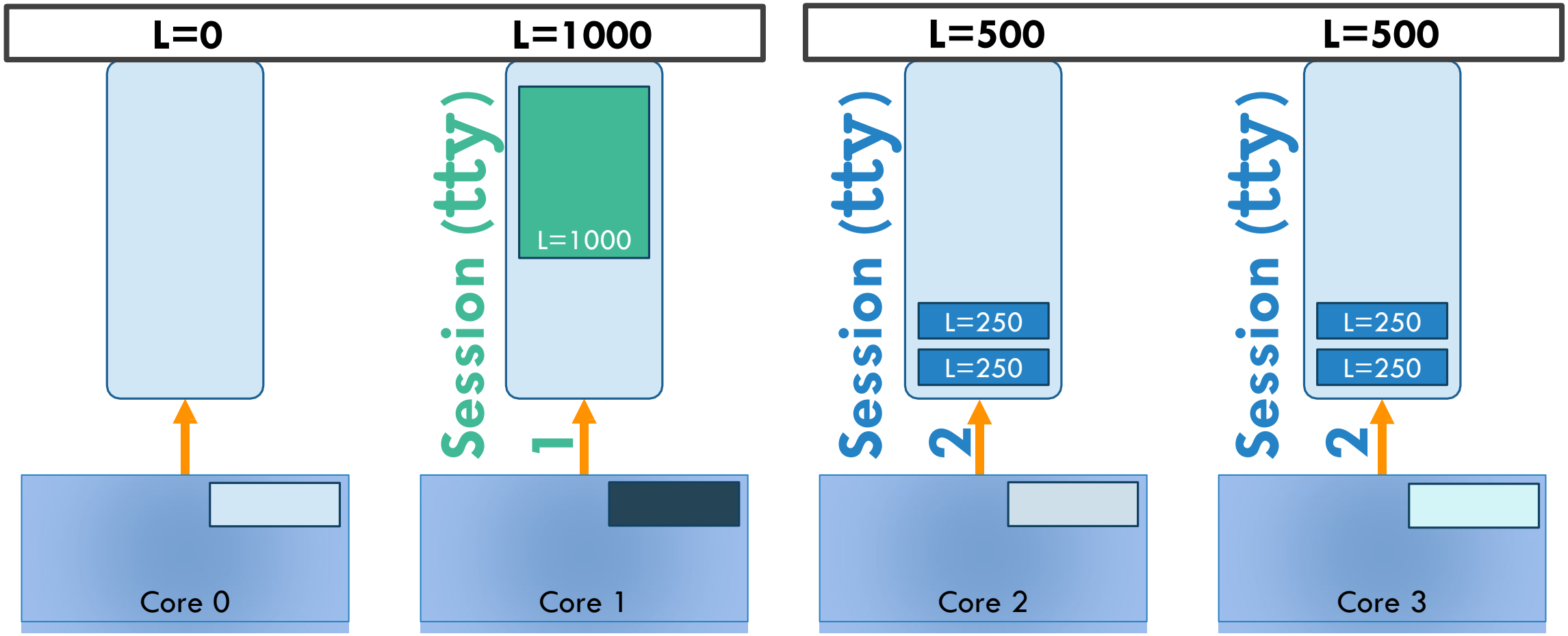
CFS load balancing

- Load calculations are actually more complicated, relies on heuristics
- **One of them aims to increase fairness between "sessions"**
- Solution: Divide the load of a task by the number of threads in **tty**!

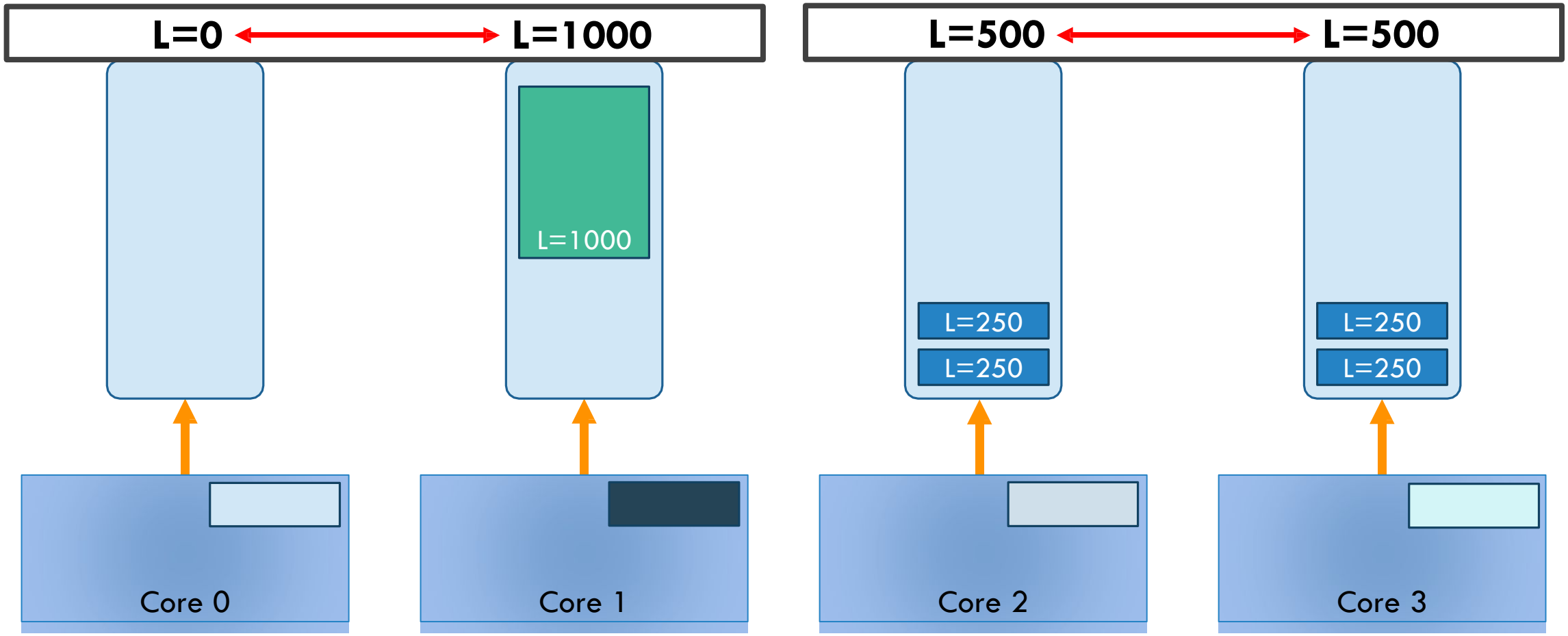


Does this balancing work?

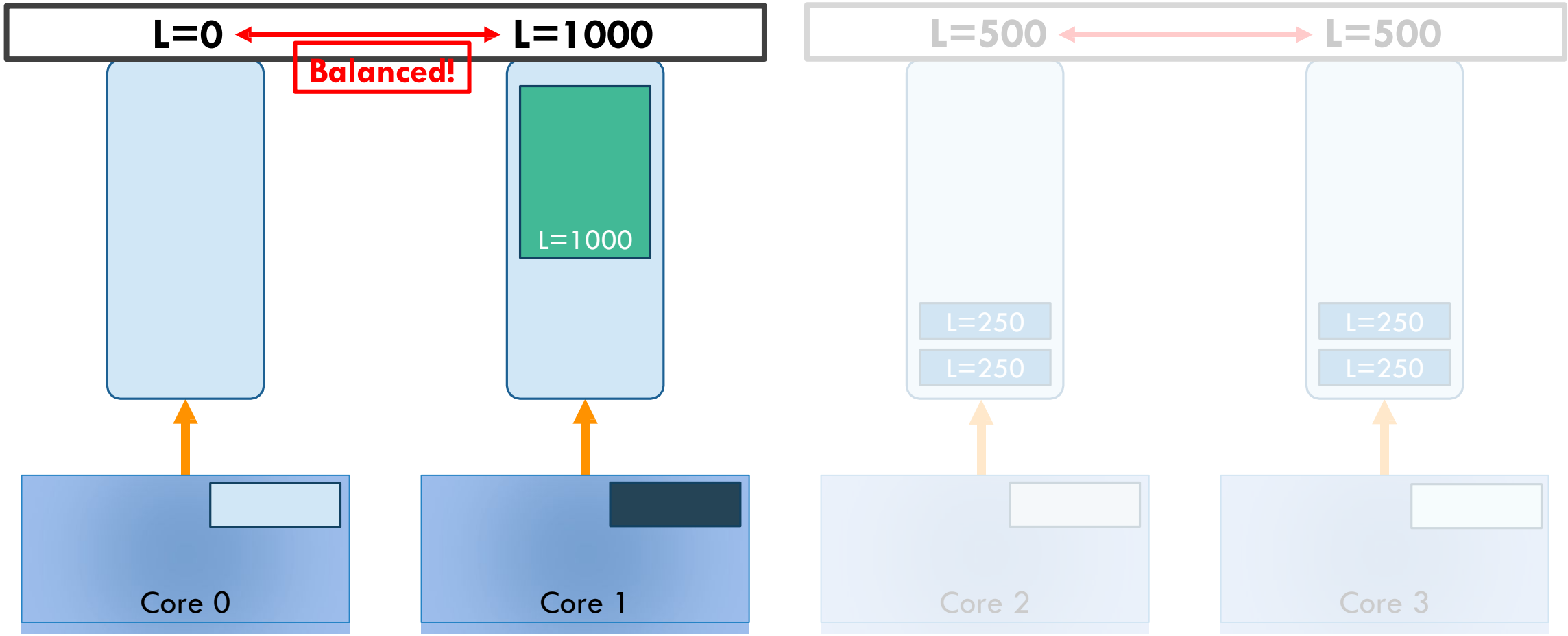
CFS load balancing: BUG #1



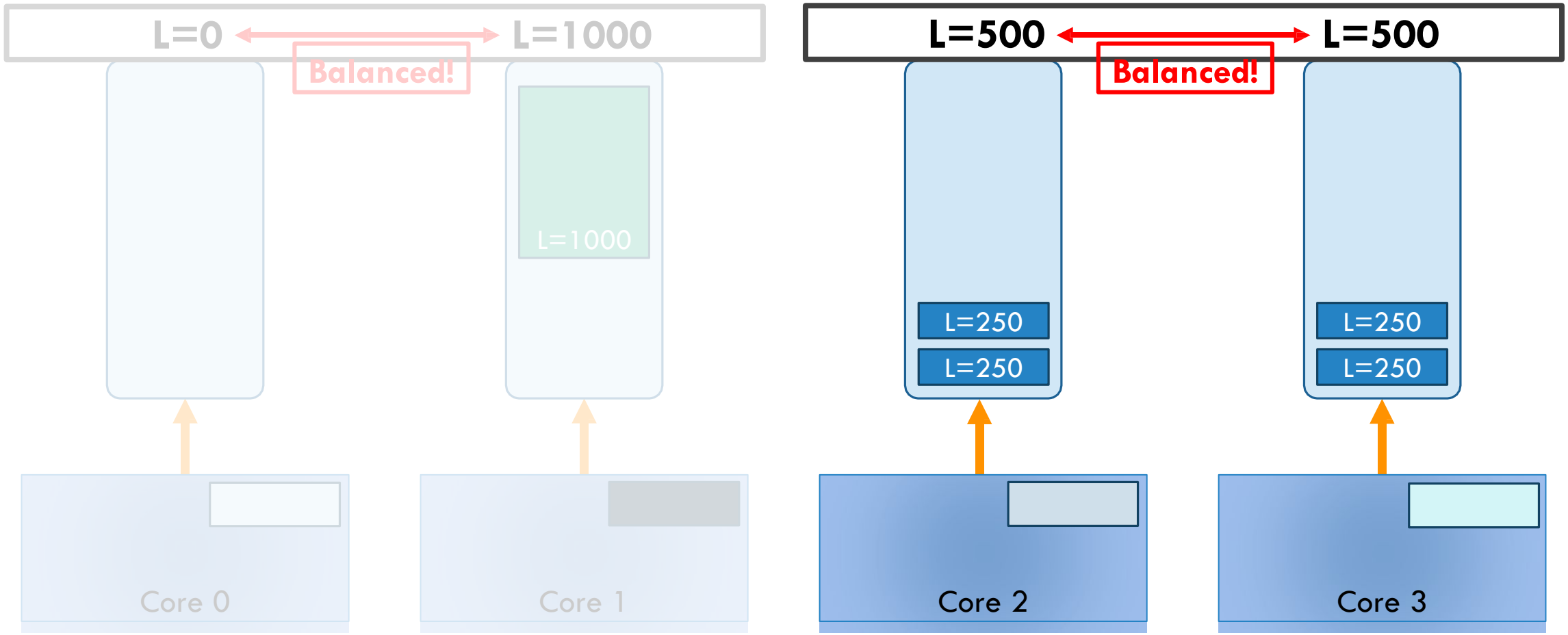
CFS load balancing: BUG #1



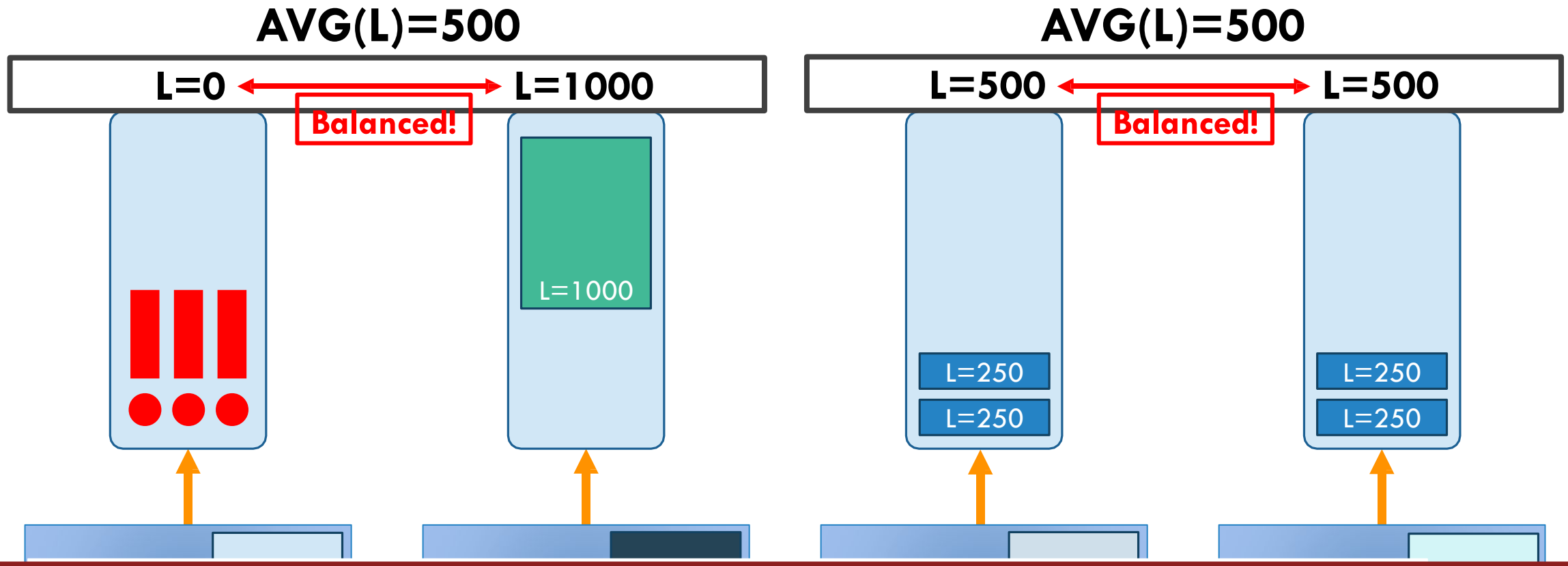
CFS load balancing: BUG #1



CFS load balancing: BUG #1

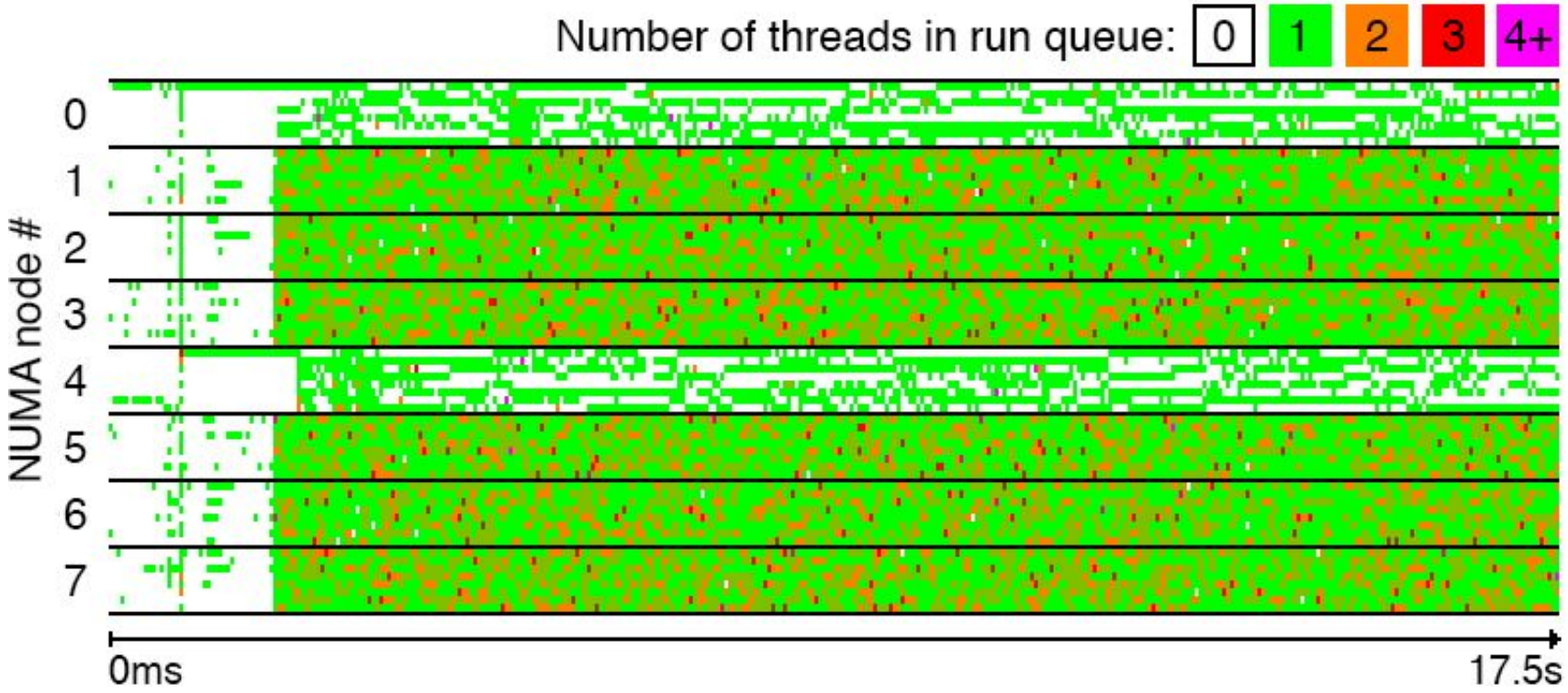


CFS load balancing: BUG #1

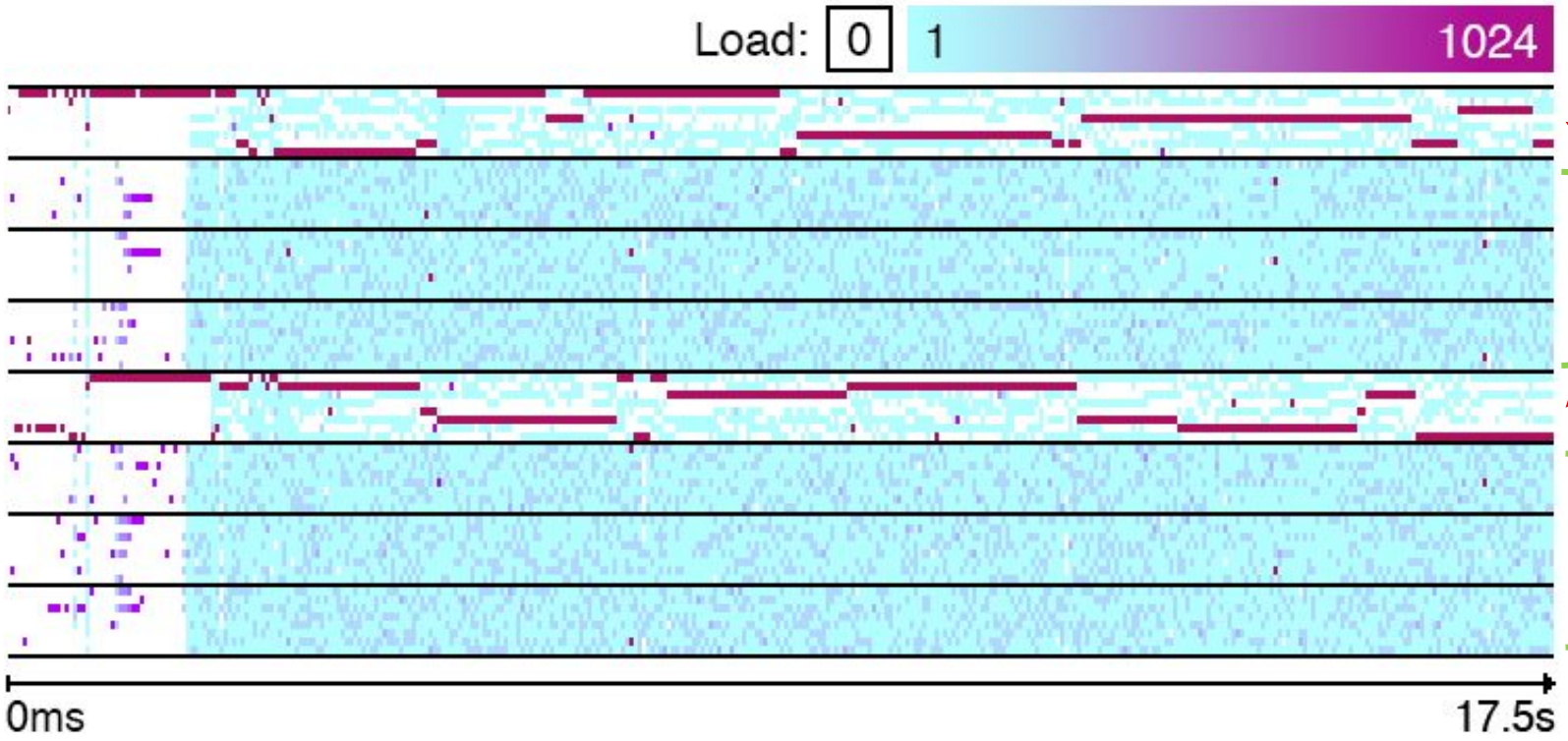


Issue with the load metric!

CFS load balancing: BUG #1



CFS load balancing: BUG #1

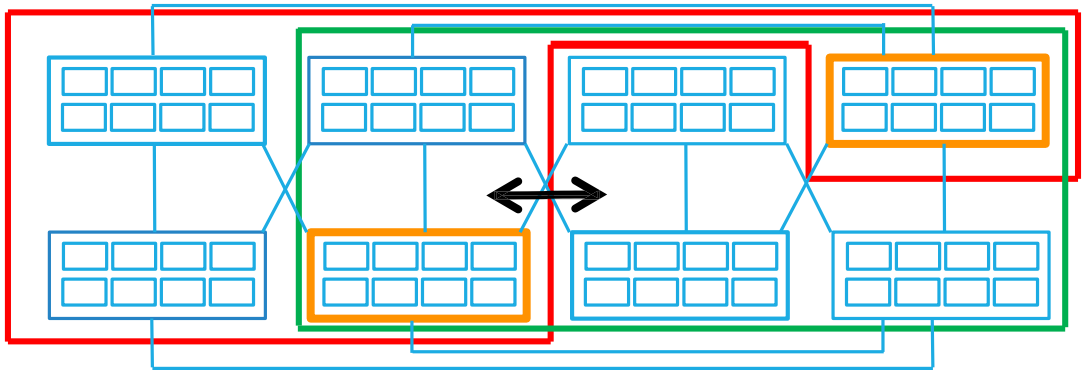


Load 1 = avg(R thread with high load + a few make threads with low load)

Load 2 = avg(many make threads with low load)

More bugs in hierarchical load balancing

- At the level of **cores, pairs of cores, dies, CPUs, NUMA nodes …**
- **Bug #2: on complex machines, incorrect hierarchy representation**
- At the last level, groups in the hierarchy are “not disjoint”
- Can break load balancing: whole app running on a single node



AMD machine hierarchy

- **Bug #3: disabling/reenabling a core breaks the hierarchy completely**

More bugs: wakeups

- **Bug #4: slow phases with idle cores with popular commercial DBs**
  - In addition to periodic load balancing, thread pick where they wake up
  - Only local CPUs are considered for wakeups due to locality “optimization”
  - **Intuition:** periodic load balancing global, wakeup balancing local

What went wrong?

- Scheduling is usually considered to be well-explored and solved problem
- CFS periodically balances, using a metric named **load**

↑ **Issue here…** *appeared with `tty`-balancing heuristic for multithreaded apps*

- CFS balances threads among groups of cores in a **hierarchy**

↑ **Issue here…** *added with support of complex NUMA hierarchies*

- In addition, thread **balance the load by selecting the core where to wake up**

↑ **Issue here…** *added with locality optimization for multicore architectures*

What went wrong?

- Scheduling is usually considered to be well-explored and solved problem
- CFS periodically balances, using a metric named **load**

↑ **Issue here...** *appeared with `tty`-balancing heuristic for multithreaded apps*

- CFS balances threads among groups of cores in a **hierarchy**

↑ **Issue here...** *added with support of complex NUMA hierarchies*

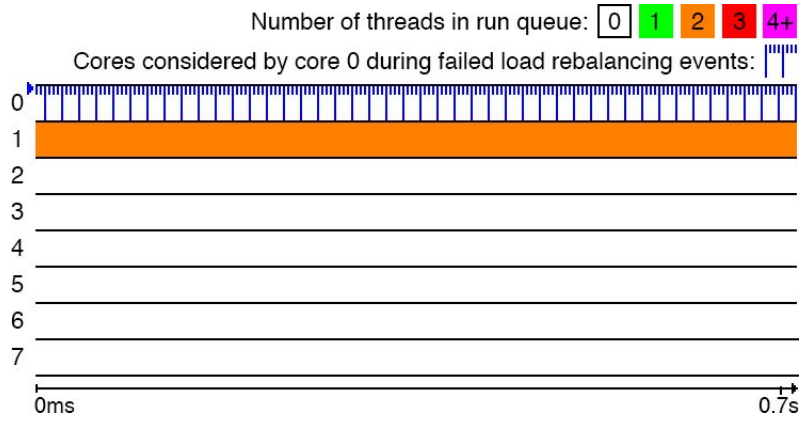
- In addition, thread **balance the load by selecting the core where to wake up**

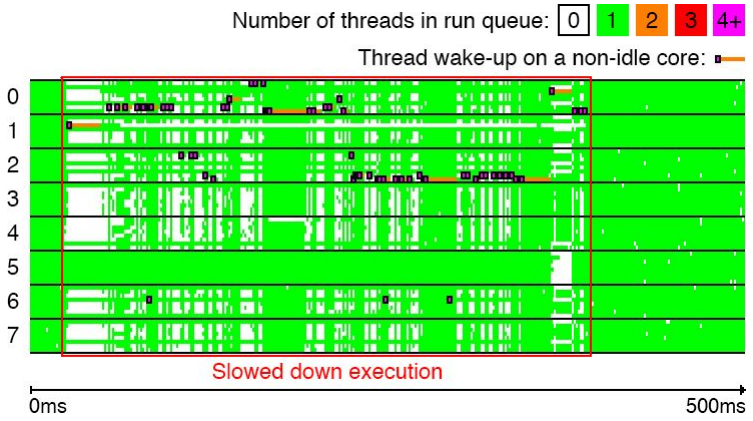
↑ **Issue here...** *added with locality optimization for multicore architectures*

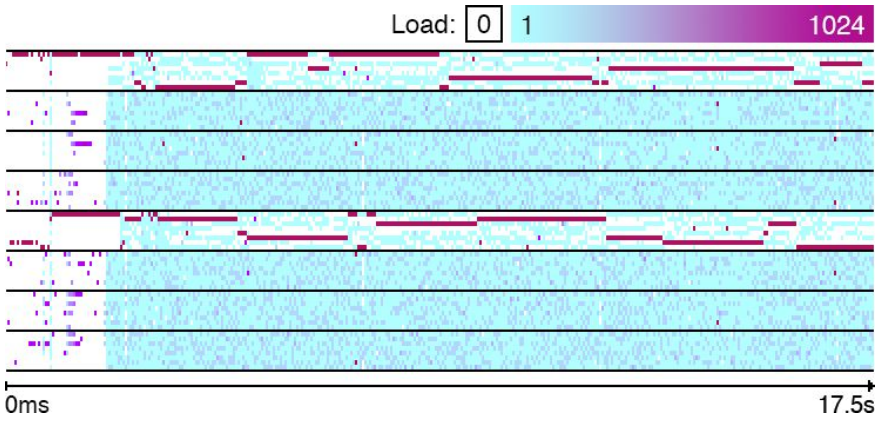
CFS was simple: but became complex due to new hardware and usecases

How to detect such bugs or issues?

- Sanity checker detects invariant violations to find bugs
- Detect suspicious situations, monitor them and produce them if they last
- Exact traces are necessary to understand complex problems







Fixing these scheduling bugs

- **Bug #1:** minimum load balancing than using average (weird!)
- **Bug #2–#3:** Build the hierarchy differently and dynamically (seems to work)
- **Bug #4:** Wake up on cores that are idle for longest time (bad for energy)
- Difficult to gauge the fixes, can worsen performance
  - Scheduler is too complex, many competing heuristics added empirically
- Is redesign possible?
  - EEVDF (next class)
  - Relax work conservation for concurrent case (Ipanema, Eurosys 2020)

Summary

- Simple idea: Logically a queue of runnable tasks, ordered by who has had the least CPU time
- Implemented with a tree for fast lookup, reinsertion
- Timer counts the virtual ticks
- Tweaks to virtual runtime allows us to implement various priorities
- Load balancing is quite difficult in actual scenarios and can lead to non-work conserving situation