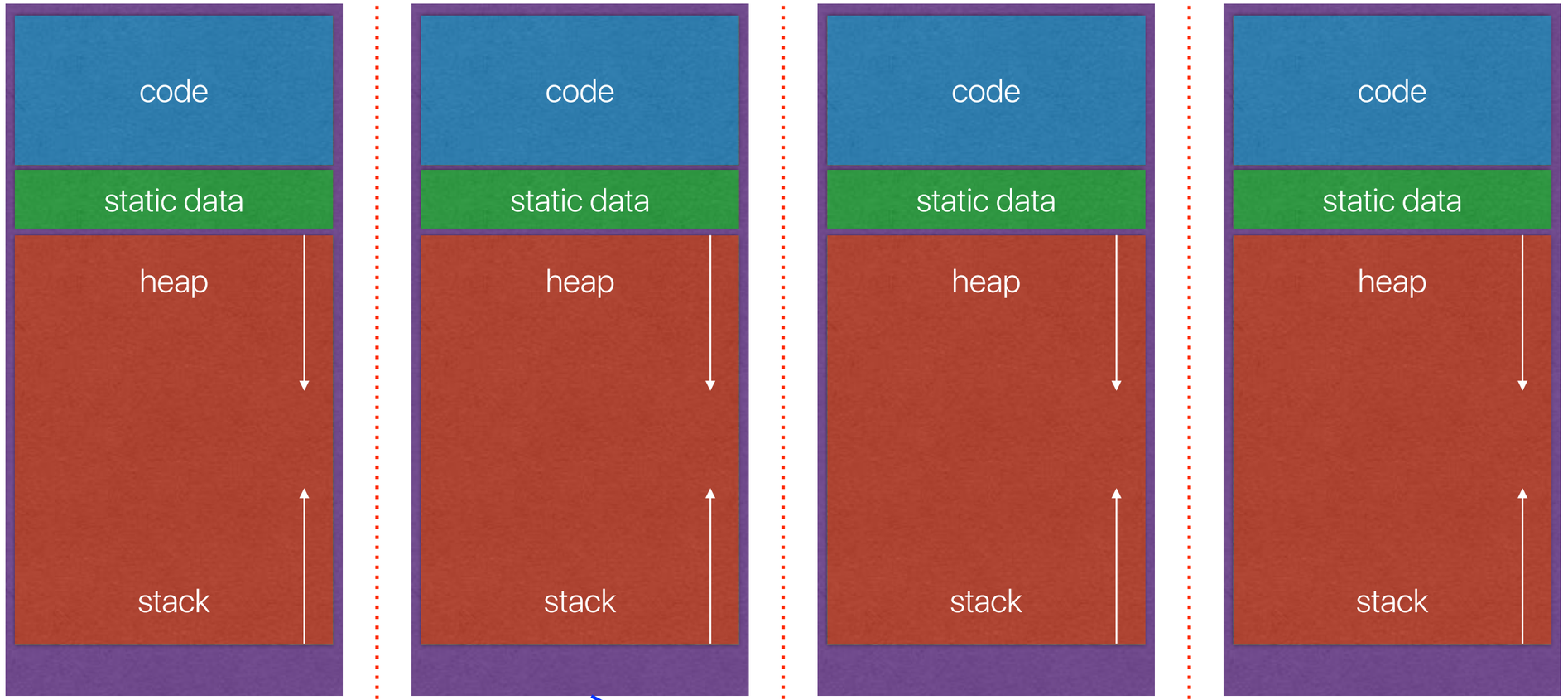
Task Scheduling

Hung-Wei Tseng

Recap: Each process has a separate virtual memory space

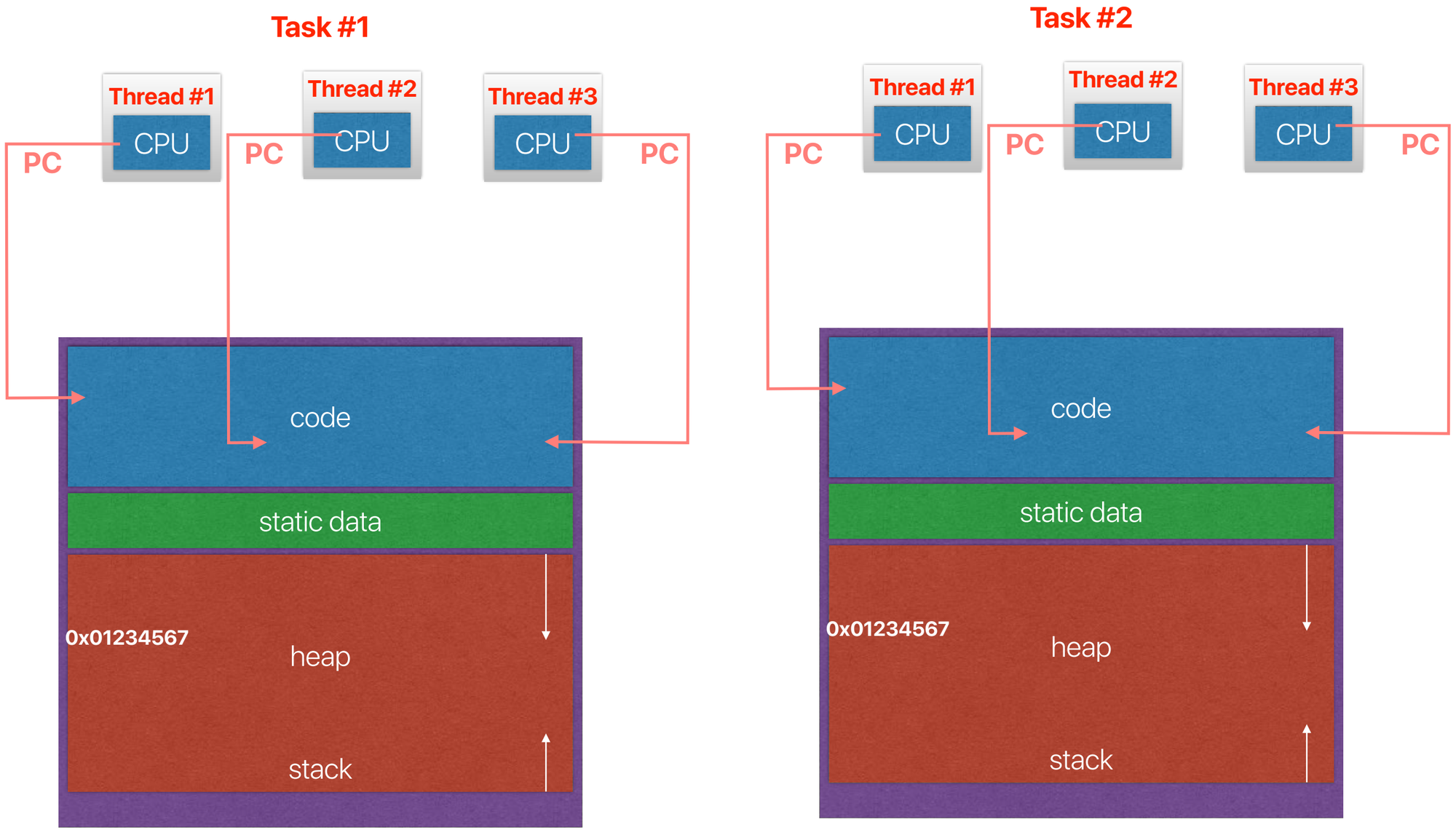


They are isolated from one another. Each of them is not supposed to know what happens to another one

Processor

Virtually, every process seems to have a processor, but only a few of them are physically executing.

Recap: Threads



Recap: Why Threads?

- Process is an abstraction of a computer
  - When you create a process, you duplicate everything
  - However, you only need to duplicate CPU abstraction to parallelize computation tasks
- Threads as lightweight processes
  - Thread is an abstraction of a CPU in a computer
  - Maintain separate execution context
  - Share other resources (e.g. memory)

Outline

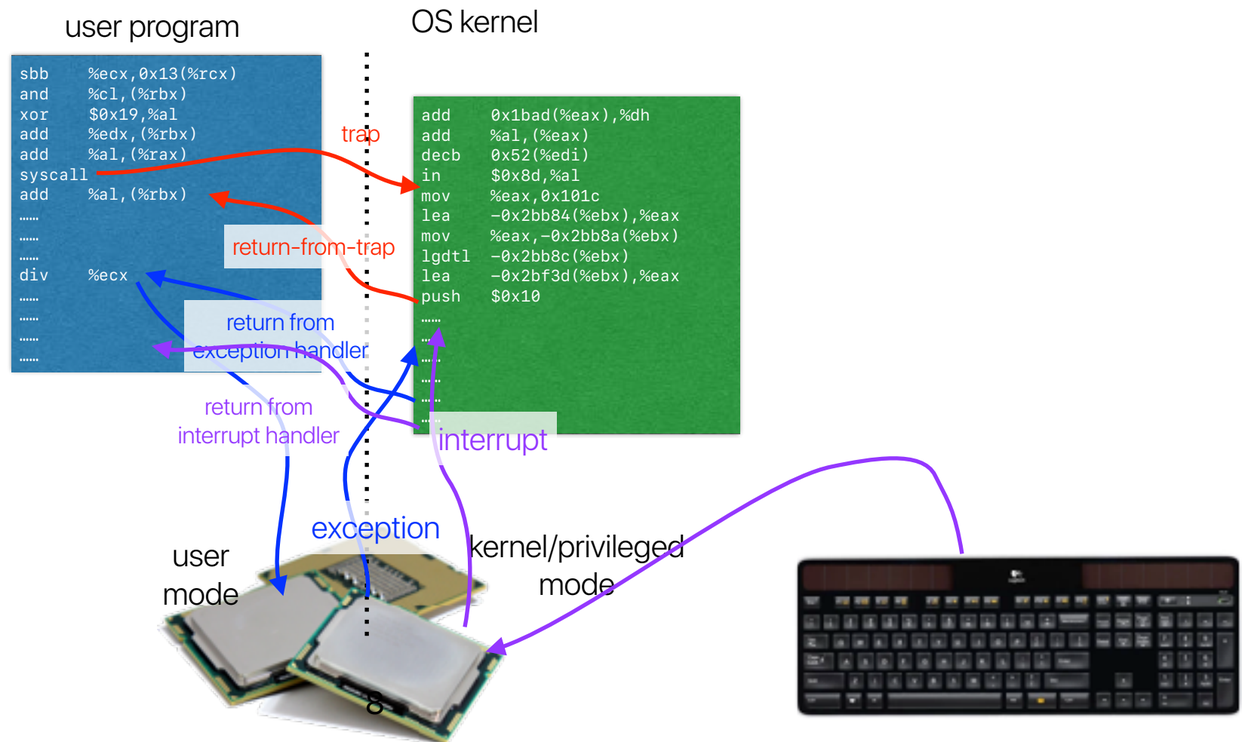
- Mechanisms of changing processes
- Basic scheduling policies
- Linux Scheduling
- An experimental time-sharing system — The Multi-Level Scheduling Algorithm

Preemptive Multitasking

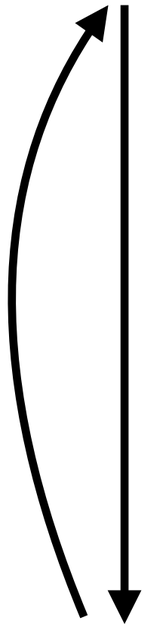
- The OS controls the scheduling — can change the running process even though the process does not give up the resource
- But how?

Three ways to invoke OS handlers

- System calls / trap instructions — raised by applications
  - Display images, play sounds
- Exceptions — raised by processor itself
  - Divided by zero, unknown memory addresses
- Interrupts — raised by hardware
  - Keystroke, network packets

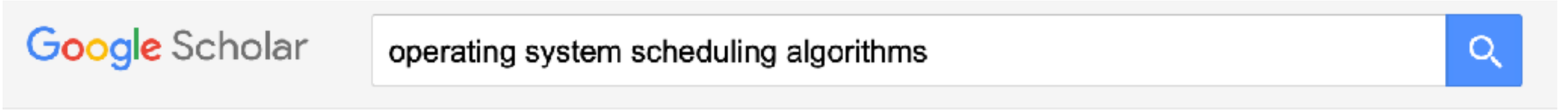


How preemptive multitasking works



- Setup a **timer** (a hardware feature by the processor)event before the process start running
- After a certain period of time, the **timer** generates **interrupt** to force the running process transfer the control to OS kernel
- The OS kernel code decides if the system wants to continue the current process
  - If not — context switch
  - If yes, return to the process

Scheduling Policies from Undergraduate OS classes



Articles

About 2,380,000 results (**0.10** sec)

CPU Scheduling

- Virtualizing the processor
  - Multiple processes need to share a single processor
  - Create an illusion that the processor is serving my task by rapidly switching the running process
- Determine which process gets the processor for how long

What you learned before

- Non-preemptive/cooperative: the task runs until it finished
  - FIFO/FCFS: First In First Out / First Come First Serve
  - SJF: Shortest Job First
- Preemptive: the OS periodically checks the status of processes and can potentially change the running process
  - STCF: Shortest Time-to-Completion First
  - RR: Round robin

An experimental time-sharing system

Fernando J. Corbató, Marjorie Merwin-Daggett and Robert C. Daley
Massachusetts Institute of Technology, Cambridge, Massachusetts

Why Multi-level scheduling algorithm?

- System **saturation** — the demand of computing is larger than the physical **processor** resource available
- Service level degrades
  - Lots of **program** swap ins-and-outs (known as **context switches** in our current terminology)
  - User interface response time is bad — you have to wait until your turn
  - Long running tasks cannot make good progress — frequent swap in-and-out

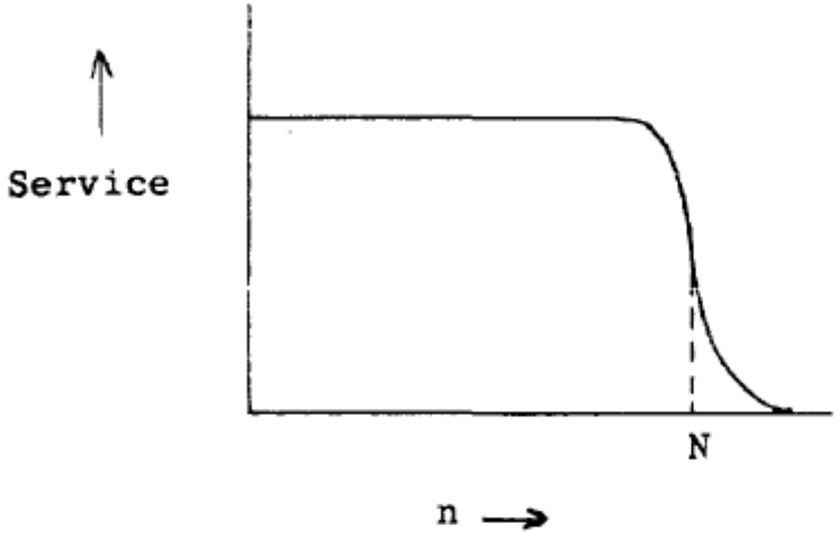
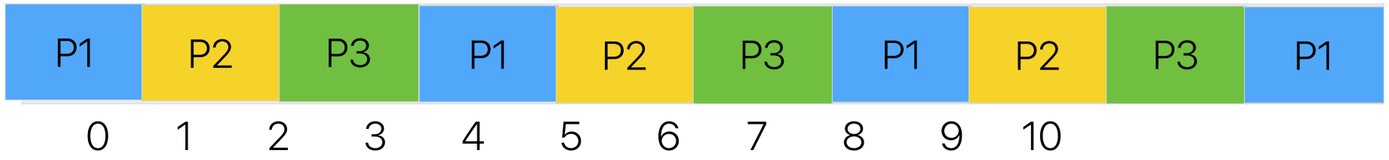


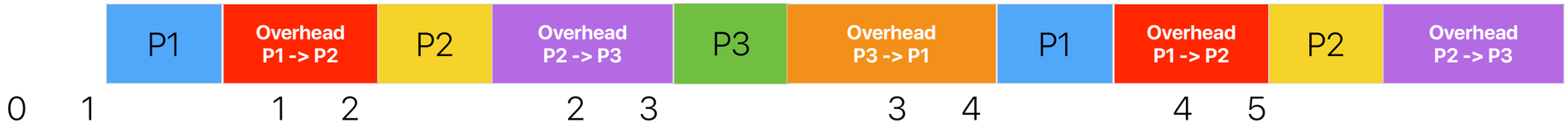
Figure 1. Service vs. Number of Active Users

Context Switch Overhead

You think round robin should act like this —



But the fact is —



- **Your processor utilization can be very low if you switch frequently**
- **No process can make sufficient amount of progress within a given period of time**
- **It also takes a while to reach your turn**

The Multilevel Scheduling Algorithm

- Place new process in the one of the queue
  - Depending on the program size

$$\ell_o = \left\lceil \log_2 \left(\left[\frac{w_p}{w_q} \right] + 1 \right) \right\rceil$$

w_p is the program memory size — smaller ones are assigned to lower numbered queues

Why?

  - **Smaller tasks are given higher priority in the beginning**
- Schedule processes in one of N queues
  - Start in initially assigned queue n
  - Run for 2^n quanta (where n is current depth)
  - If not complete, move to a higher queue (e.g. n +1)
  - **Larger process will execute longer before switch**
- Level m is run only when levels 0 to m-1 are empty
- **Smaller process, newer process are given higher priority**

The Multilevel Scheduling Algorithm

- Not optimized for anything — it's never possible to have an optimized scheduling algorithm without prior knowledge regarding all running processes
- It's practical — many scheduling algorithms used in modern OSes still follow the same idea

The Linux Scheduler: a Decade of Wasted Cores

J-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova

Linux's Completely Fair Scheduler (CFS)

- Real time process classes – always run first (rare)
- Interactive processes: Usually blocked, low total run time, high priority

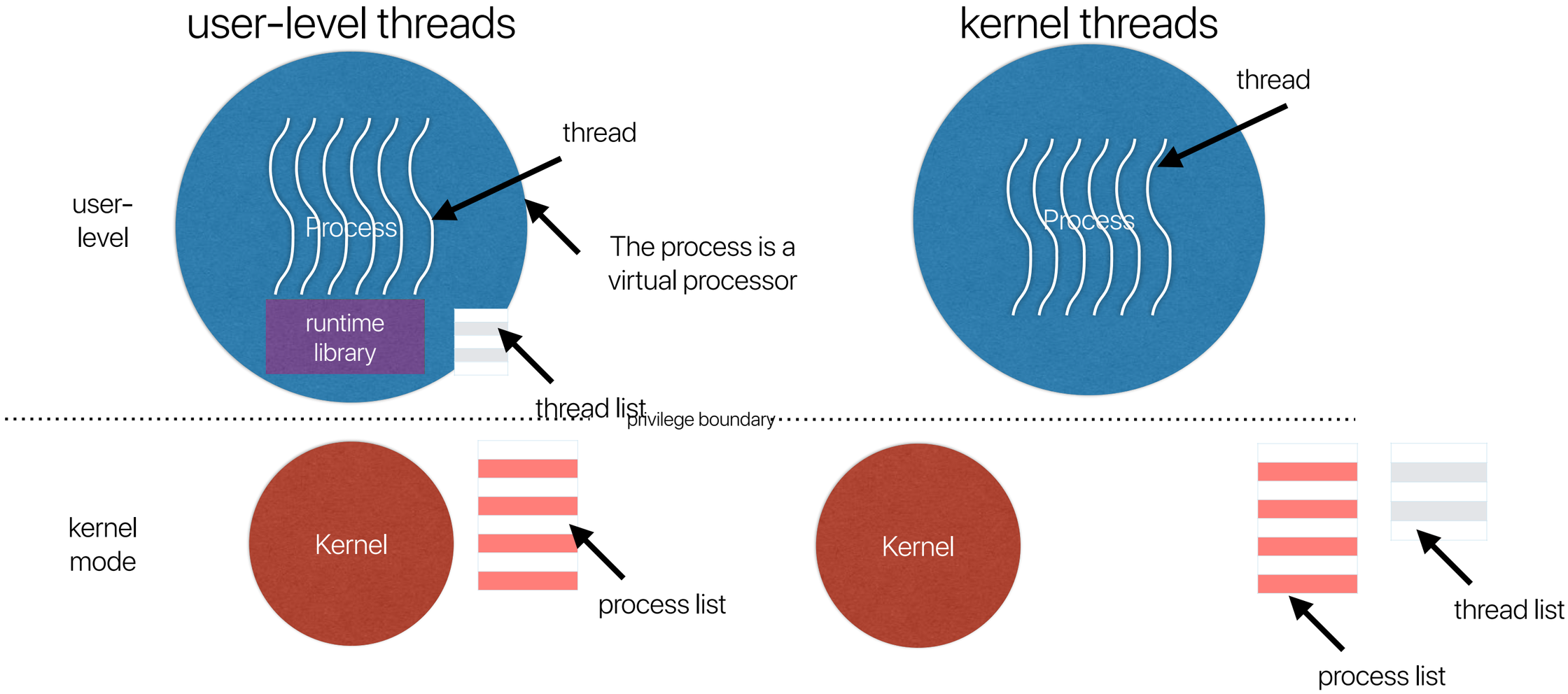
- Other processes:
  - Red-black BST of process, organized by CPU time they've received.
  - Pick the ready process that has run for the shortest (normalized) time thus far.
  - Run it, update it's CPU usage time, add to tree

Image source: https://www.ibm.com/developerworks/library/l-completely-fair-scheduler/

CFS on multicore systems

- Each processor has a run-queue — the load within each local queue may not be balanced
- Run load balancing algorithms
  - Cannot invoked often — Expensive computation-wise and communication-wise
  - “Emergency” load-balancing if any core is idle

User-level v.s kernel threads

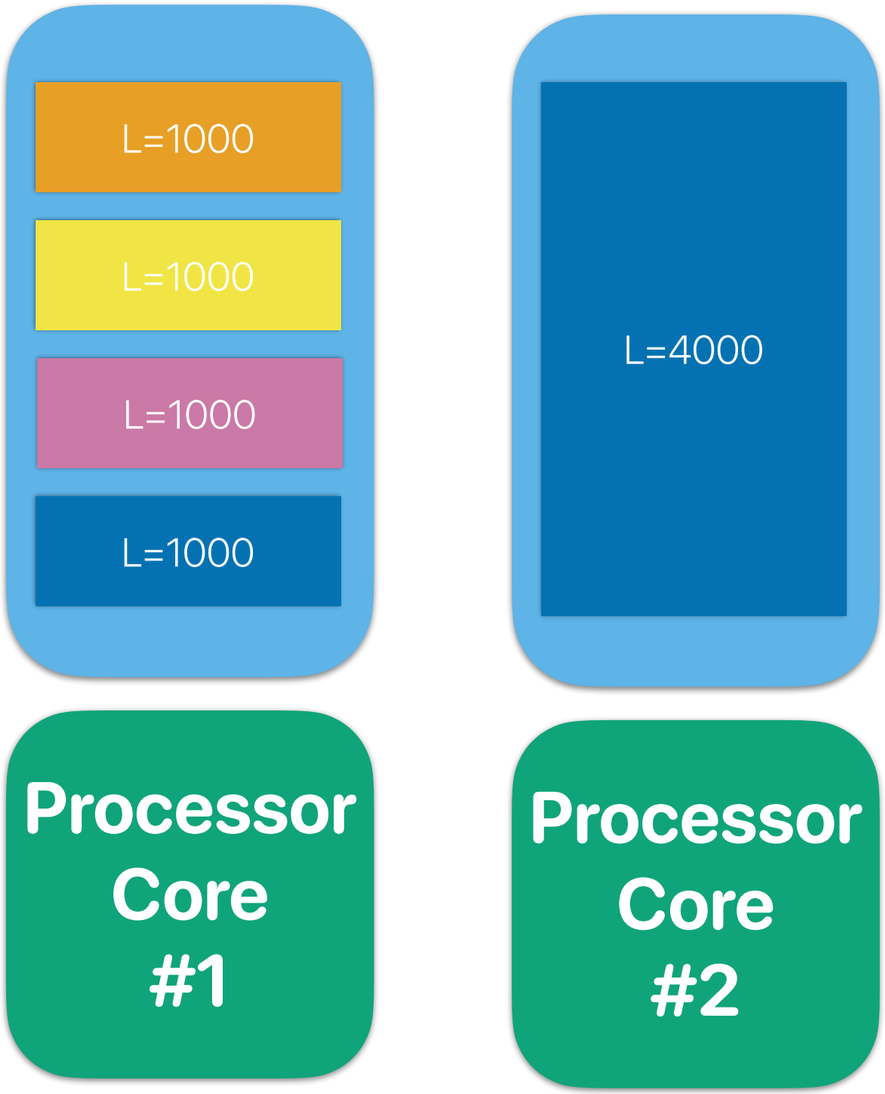


- **The OS kernel is unaware of user-level threads**
- **Switching threads does not require kernel mode operations**
- **A thread can block other threads within the same process**

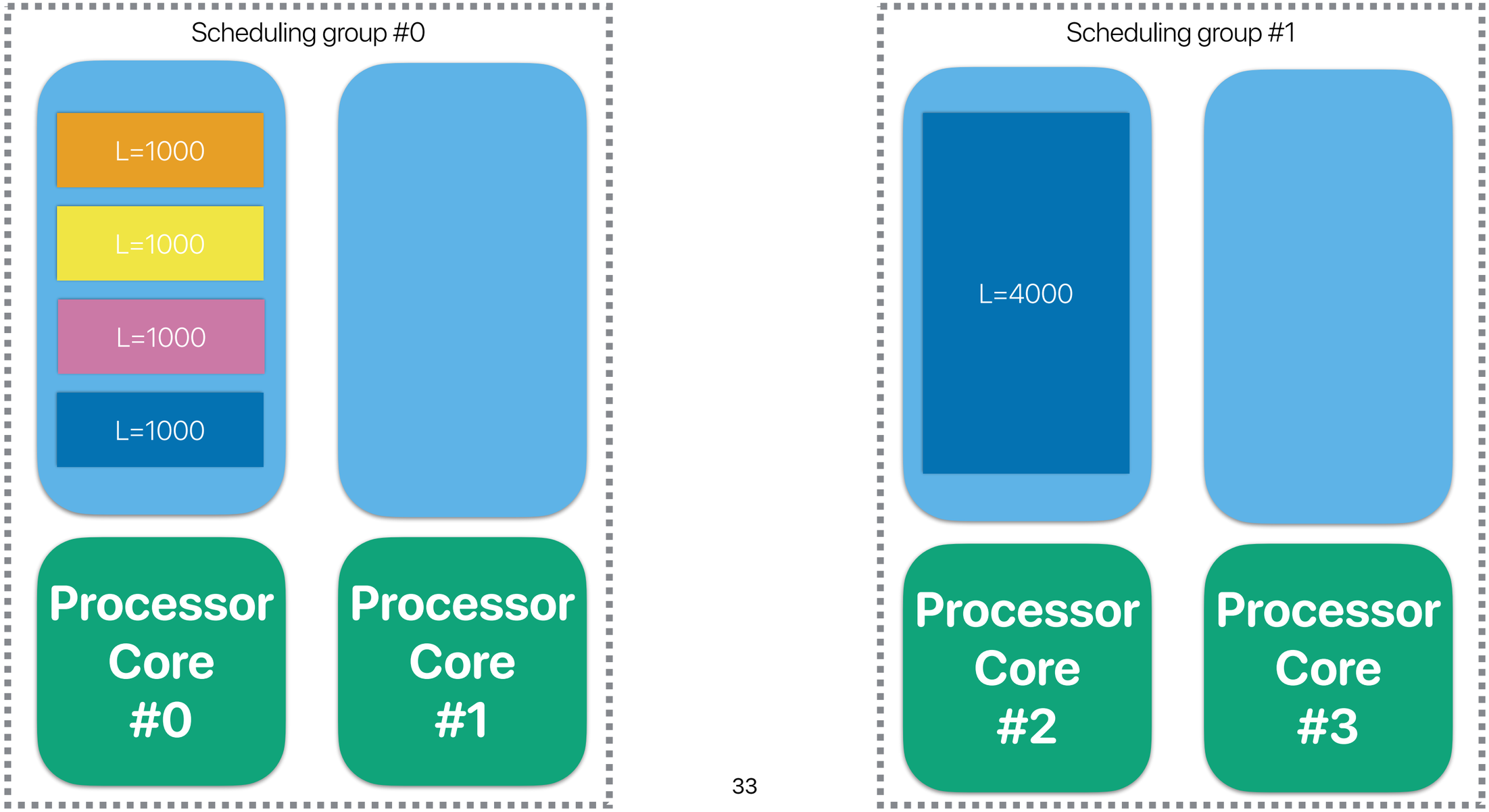
- **The kernel can control threads directly**
- **Thread switch requires kernel/user mode switch and system calls**
- **Thread works individually**

Load balancing

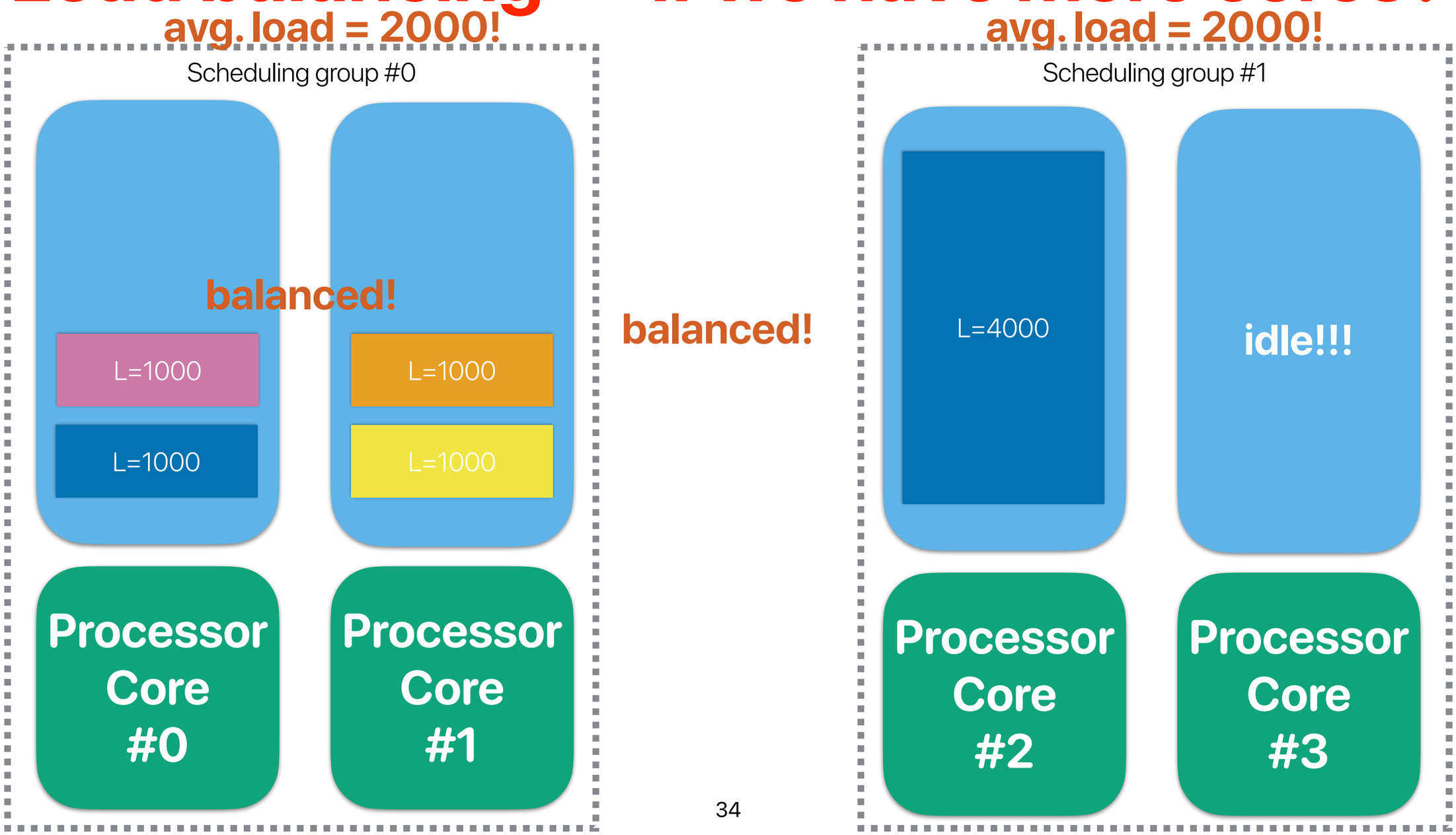
$$task\ load = weight \times \%\ of\ cpu\ use$$



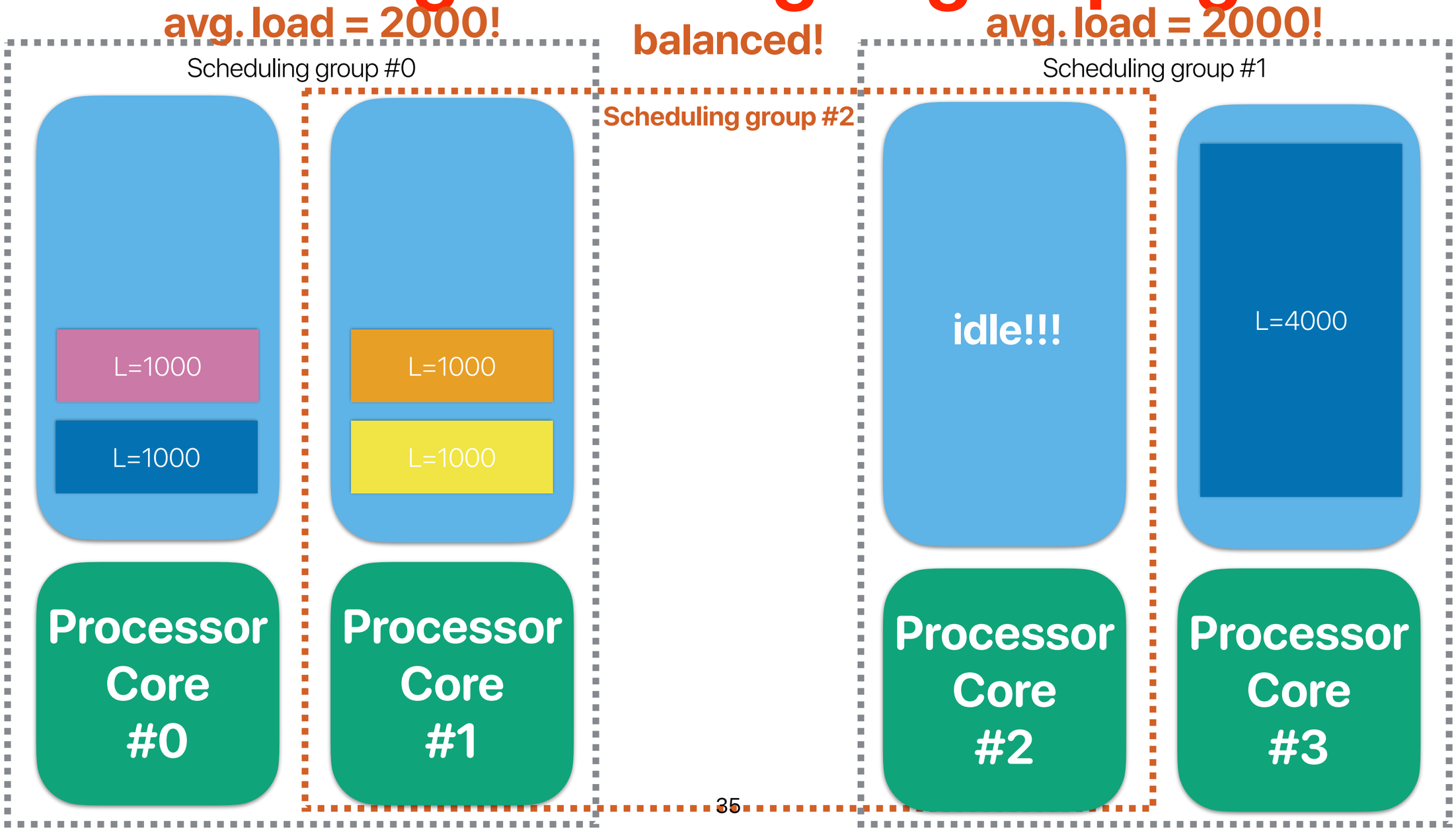
Load balancing — if we have more cores?



Load balancing — if we have more cores?



Loading balancing — grouping?



“Bugs” in Linux CFS

- Group Imbalance bug
  - Thread load are divided
  - Work stealing based on average load — use minimum load instead
- The Scheduling Group Construction bug
  - Linux spawns threads on the same core as their parent thread
- The Overload-on-Wakeup bug
  - a thread that was asleep may wake up on an overloaded core while other cores in the system are idle
  - promotes cache reuse
- The Missing Scheduling Domains bug
  - When a core is disabled and then re-enabled using the /proc interface, load balancing between any NUMA nodes is no longer performed.

Lottery Scheduling: Flexible Proportional-Share Resource Management

Carl A. Waldspurger and William E. Weihl

Why Lottery

enormous impact on throughput and response time. Accurate control over the quality of service provided to users and applications requires support for specifying relative computation rates. Such control is desirable across a wide spectrum of systems. For long-running computations such as scientific applications and simulations, the consumption of computing resources that are shared among users and applications of varying importance must be regulated [Hel93]. For interactive computations such as databases and media-based applications, programmers and users need the ability

We want Quality of Service

Few general-purpose schemes even come close to supporting flexible, responsive control over service rates.

Most approaches are not flexible, responsive

Existing *fair share* schedulers [Hen84, Kay88] and *microeconomic* schedulers [Fer88, Wal92] successfully address some of the problems with absolute priority schemes. However, the assumptions and overheads associated with these systems limit them to relatively coarse control over long-running computations. Interactive systems require

The overhead of running those algorithms are high!

ware systems. In fact, with the exception of hard real-time systems, it has been observed that the assignment of priorities and dynamic priority adjustment schemes are often ad-hoc [Dei90]. Even popular priority-based schemes for CPU allocation such as *decay-usage scheduling* are poorly understood, despite the fact that they are employed by numerous operating systems, including Unix [Hel93].

No body knows how they work...

Solution — Lottery and Tickets

What lottery proposed?

- Each process hold a certain number of lottery tickets
- Randomize to generate a lottery
- If a process wants to have higher priority
  - Obtain more tickets!

Ticket economics

- Ticket transfers
- Ticket inflation
- Ticket currencies
- Compensation tickets

How good is lottery?

- The overhead is not too bad
  - 1000 instructions ~ less than 500 ns on a 2 GHz processor
- Fairness
  - Figure 5: average ratio in proportion to the ticket allocation
- Flexibility
  - Allows Monte-Carlo algorithm to dynamically inflate its tickets
- Ticket transfer
  - Client-server setup

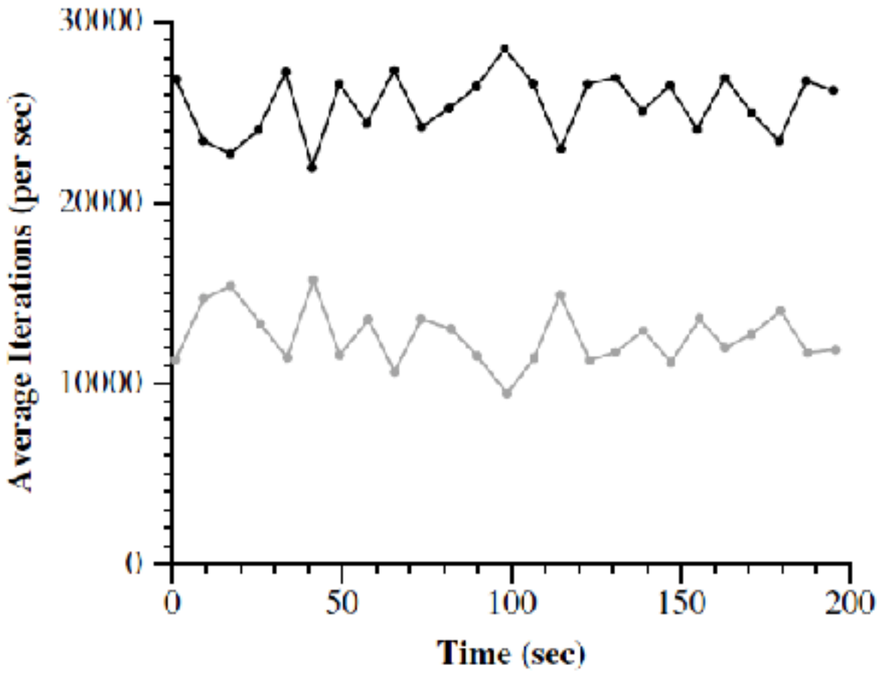


Figure 5: **Fairness Over Time.** Two tasks executing the Dhrystone benchmark with a 2 : 1 ticket allocation. Averaged over the entire run, the two tasks executed 25378 and 12619 iterations/sec., for an actual ratio of 2.01 : 1.

The impact of “lottery”

- Data center scheduling
  - You buy “times”
  - Lottery scheduling of your virtual machine

Will you use lottery for your system?

- Will it be good for
  - Event-driven application
  - Real-time application
  - GUI-based system
- Is randomization a good idea?
  - The authors later developed a deterministic stride-scheduling