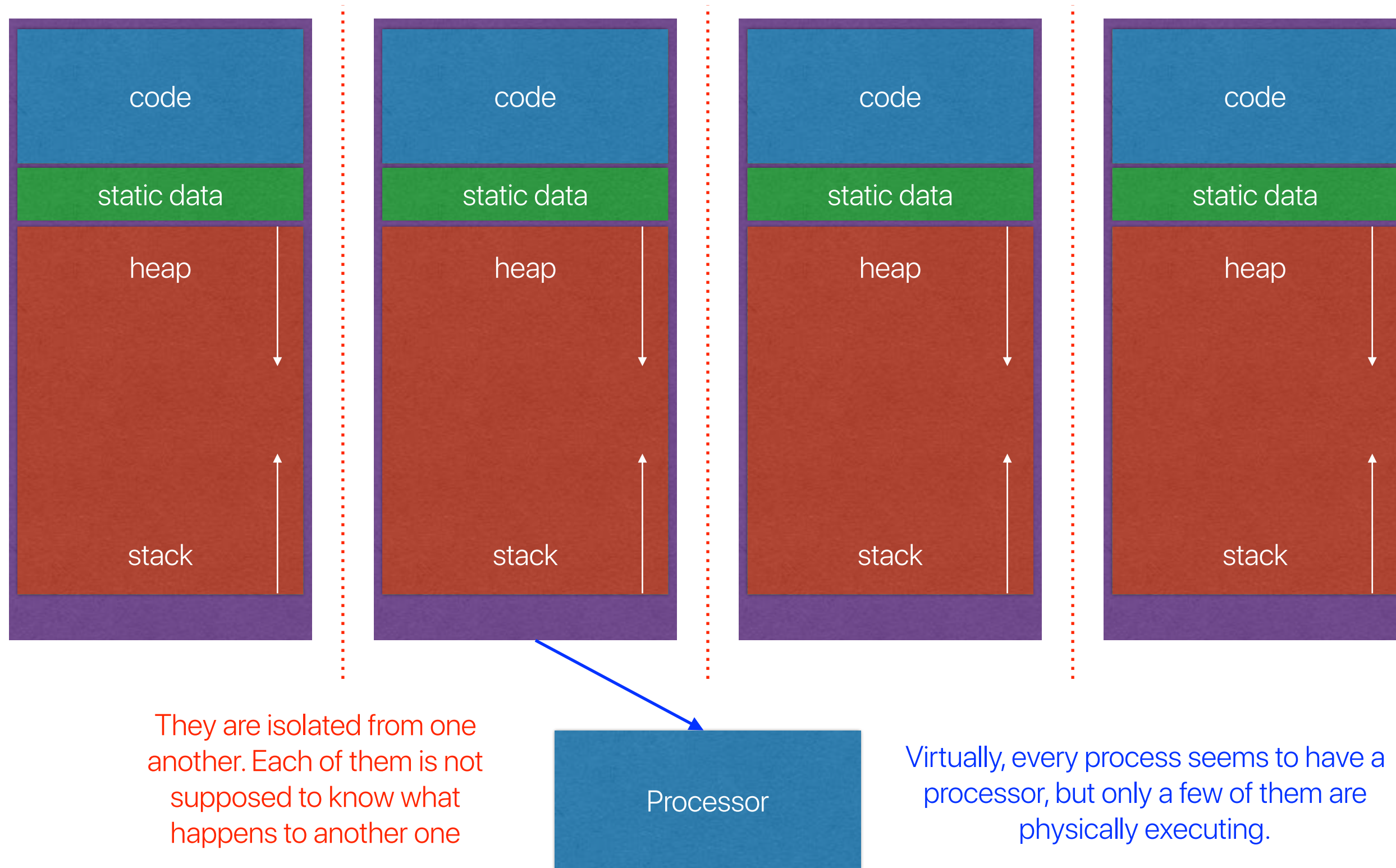


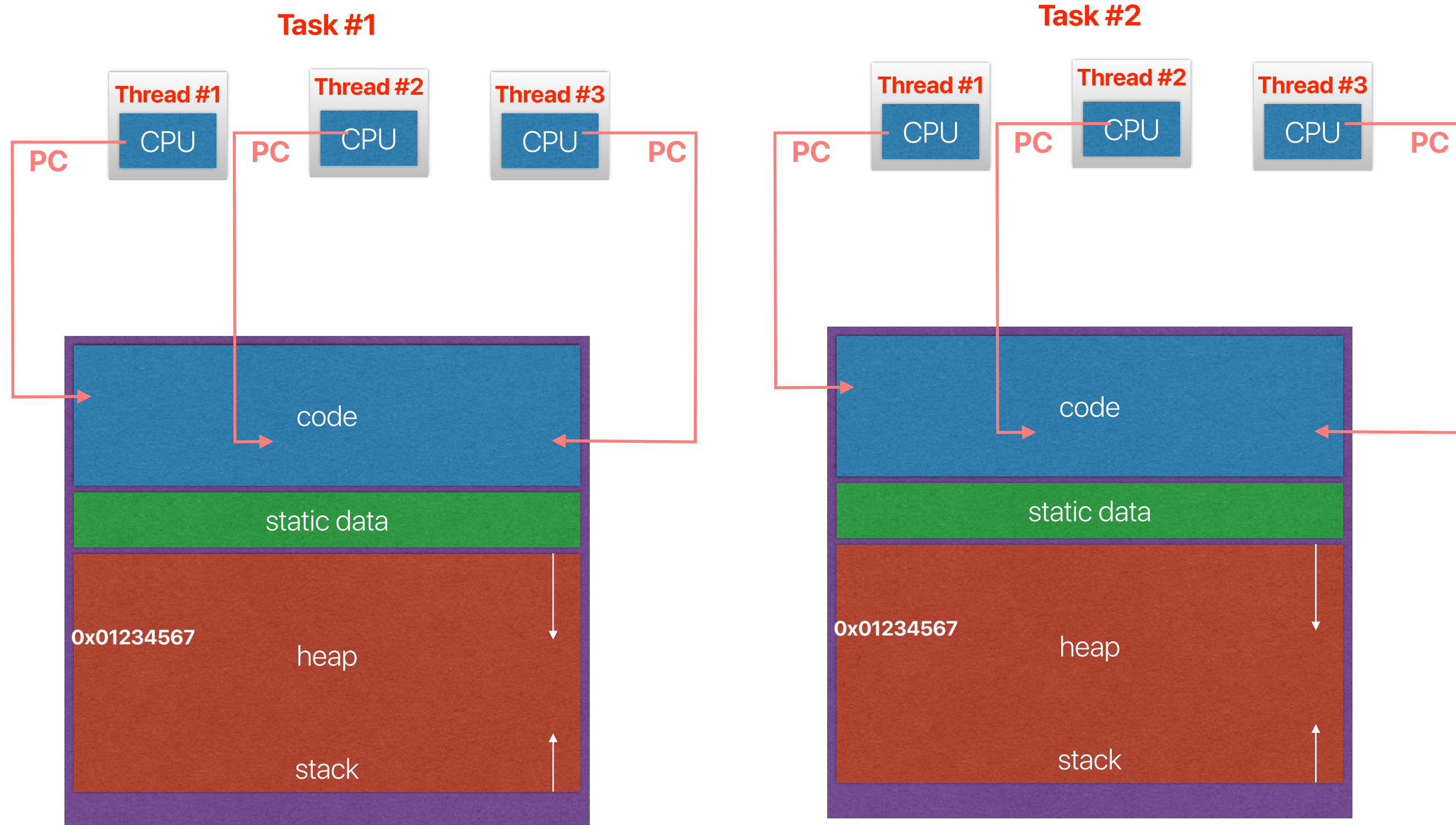
# Task Scheduling

Hung-Wei Tseng

# Recap: Each process has a separate virtual memory space



# 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)

# Recap: Synchronization mechanisms

- Locks
  - Mutual exclusion
  - Progress
  - Fairness
  - Accommodation of nondeterminism/multiple processors
- Semaphores
  - Allow multiple concurrent processes/threads working together
  - Waitlist to reduce CPU load
- RCU
  - Wait-free/Lock-free when reads
  - Used intensively in Linux kernels
- Wait-free synchronization
  - Wait-free regardless reads/writes
- All above needs to be implemented through atomic instructions

# Outline

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

# The mechanisms of changing the running processes

- Cooperative Multitasking (non-preemptive multitasking)
- Preemptive Multitasking

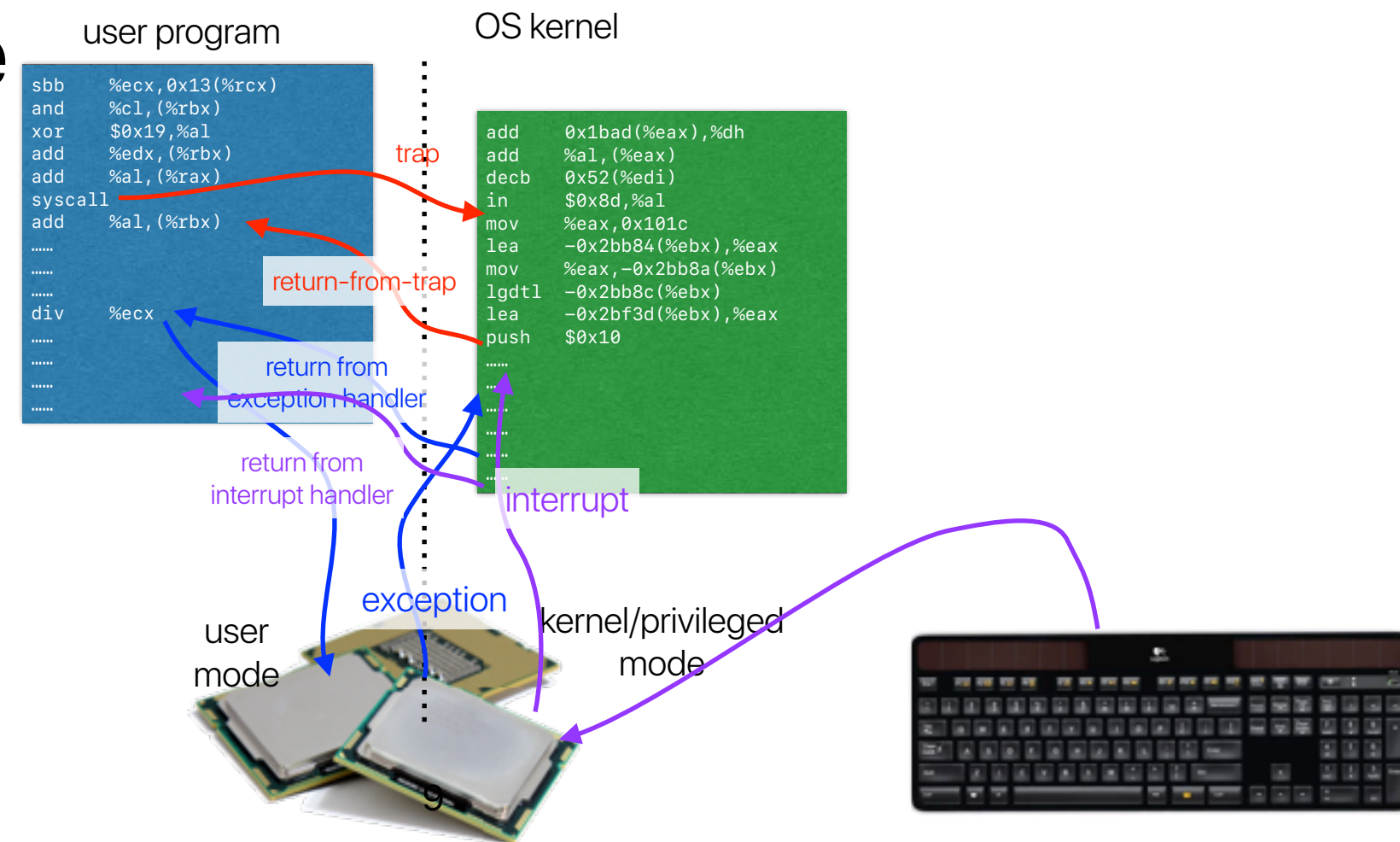
# Preemptive Multitasking

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

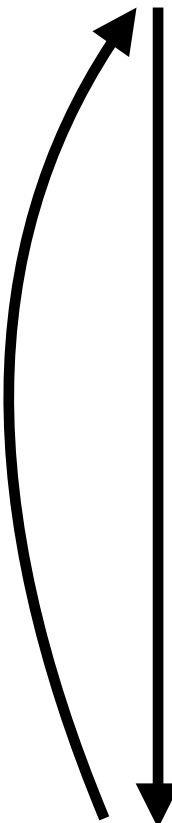


# Three ways to invoke OS handlers

- 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**

Google Scholar

operating system scheduling algorithms



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

# Scheduling Metrics

- CPU utilization — how busy we keep the CPU to be
- Throughput — the **amount** of “tasks/processes/threads” that we can finish **within a given amount of time**
- Turnaround time — the time between **submission/arrival** and **completion**
- Response time — the time between **submission** and **the first time when the job is scheduled**
- Wait time — the time between **the job is ready** (not including the overhead of queuing, command processing) and **the first time when the job is scheduled**
- Fairness — every process should get a fair chance to make progress

# 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

# Parameters for policies

- How many of the following scheduling policies require knowledge of process run times before execution?
  - ① FIFO/FCFS: First In First Out / First Come First Serve
  - ② SJF: Shortest Job First
  - ③ STCF: Shortest Time-to-Completion First
  - ④ RR: Round robin

A. 0

B. 1

C. 2

D. 3

E. 4



# Parameters for policies

- How many of the following scheduling policies require knowledge of process run times before execution?

① FIFO/FCFS: First In First Out / First Come First Serve

② SJF: Shortest Job First

You can never know the execution time before executing them!

③ STCF: Shortest Time-to-Completion First

— These policies are not realistic

④ RR: Round robin

A. 0

B. 1

C. 2

D. 3

E. 4

**The best ones you learned in undergraduate  
OS does not even work in real!  
— forget about them in real implementation**

# **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

- Why MIT's experimental time-sharing system proposes Multi-level schedule algorithm? How many of the followings is it trying to optimize?
  - ① Turn-around time
  - ② Wait time
  - ③ Fairness
  - ④ Response time
- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

# Why Multi-level scheduling algorithm

- Why MIT's experimental time-sharing system proposes Multi-level schedule algorithm? How many of the followings is it trying to optimize?

- ① Turn-around time
- ② Wait time
- ③ Fairness
- ④ Response time

excessive in size or in time requirements. The predicament can be alleviated if it is assumed that a good design for the system is to have a saturation procedure which gives graceful degradation of the response time and effective real-time computation speed of the large and long-running users.

- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

4. The response time for programs of equal size, entering the system at the same time, and being run for multiple quanta, is no worse than approximately twice the response-time occurring in a single quanta round-robin procedure. If

Several important conclusions can be drawn from the above algorithm which allow the performance of the system to be bounded.

# 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

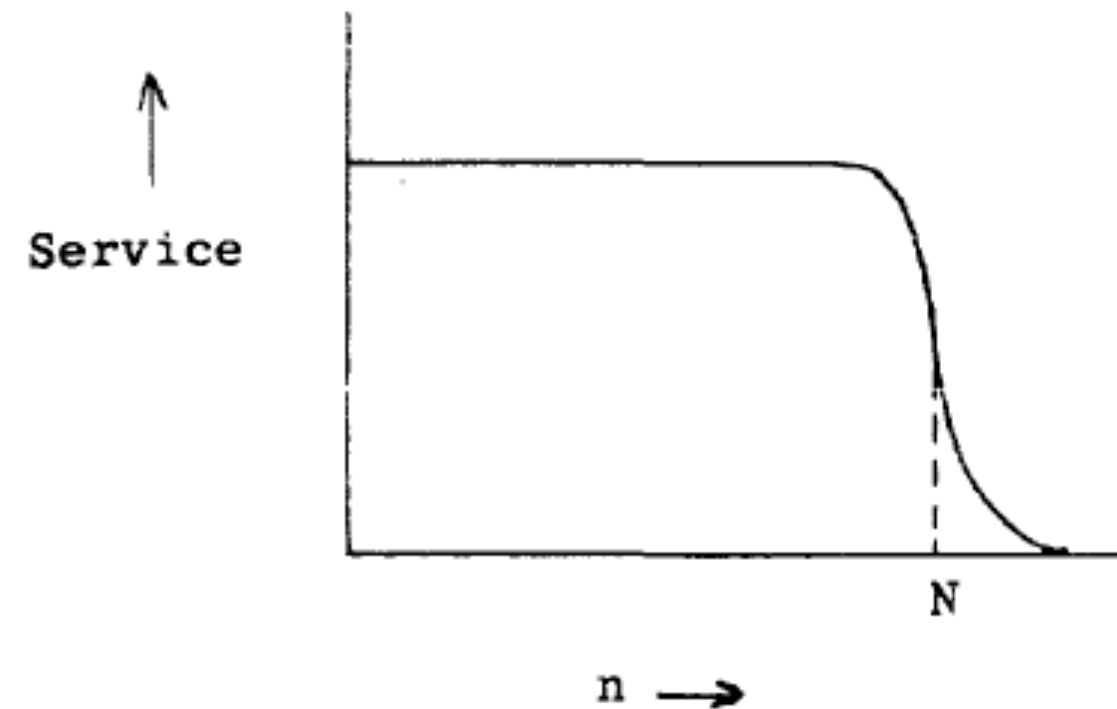


Figure 1. Service vs. Number of Active Users

# What happens during a context switch?

- How many of the followings must occur during a "context switch"?
  - ① Save the current process's PC/registers to its PCB
  - ② Flush/invalidate the cache content of the current process
  - ③ Restore the upcoming process's PC/registers from its PCB
  - ④ Load memory pages of the upcoming processes

A. 0

B. 1

C. 2

D. 3

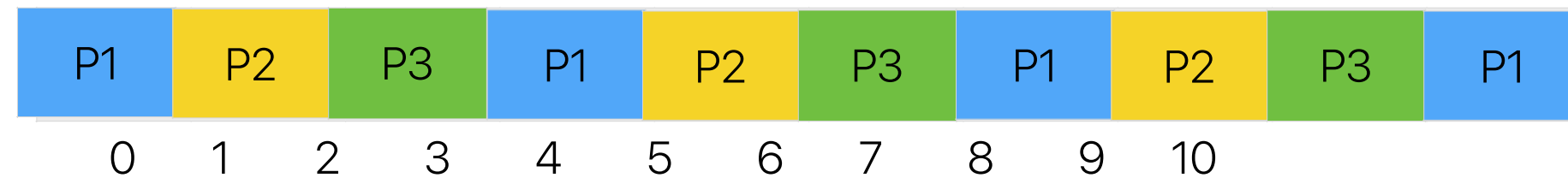
E. 4

# What happens during a context switch?

- How many of the followings must occur during a “context switch”?
  - ① Save the current process's PC/registers to its PCB
  - ② Flush/invalidate the cache content of the current process
  - ③ Restore the upcoming process's PC/registers from its PCB
  - ④ Load memory pages of the upcoming processes
- A. 0
- B. 1
- C. 2
- D. 3**
- E. 4

# 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

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

$w_p$  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

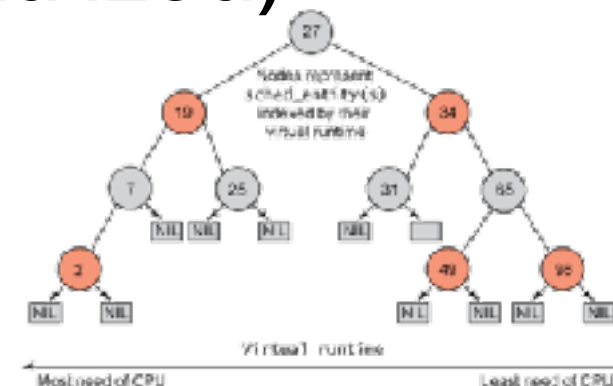
- Not optimized for anything — it's never possible to have an optimized scheduling algorithm without prior knowledge regarding all running processes
- It's practical — many scheduling algorithms used in modern OSes still follow the same idea

# **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)

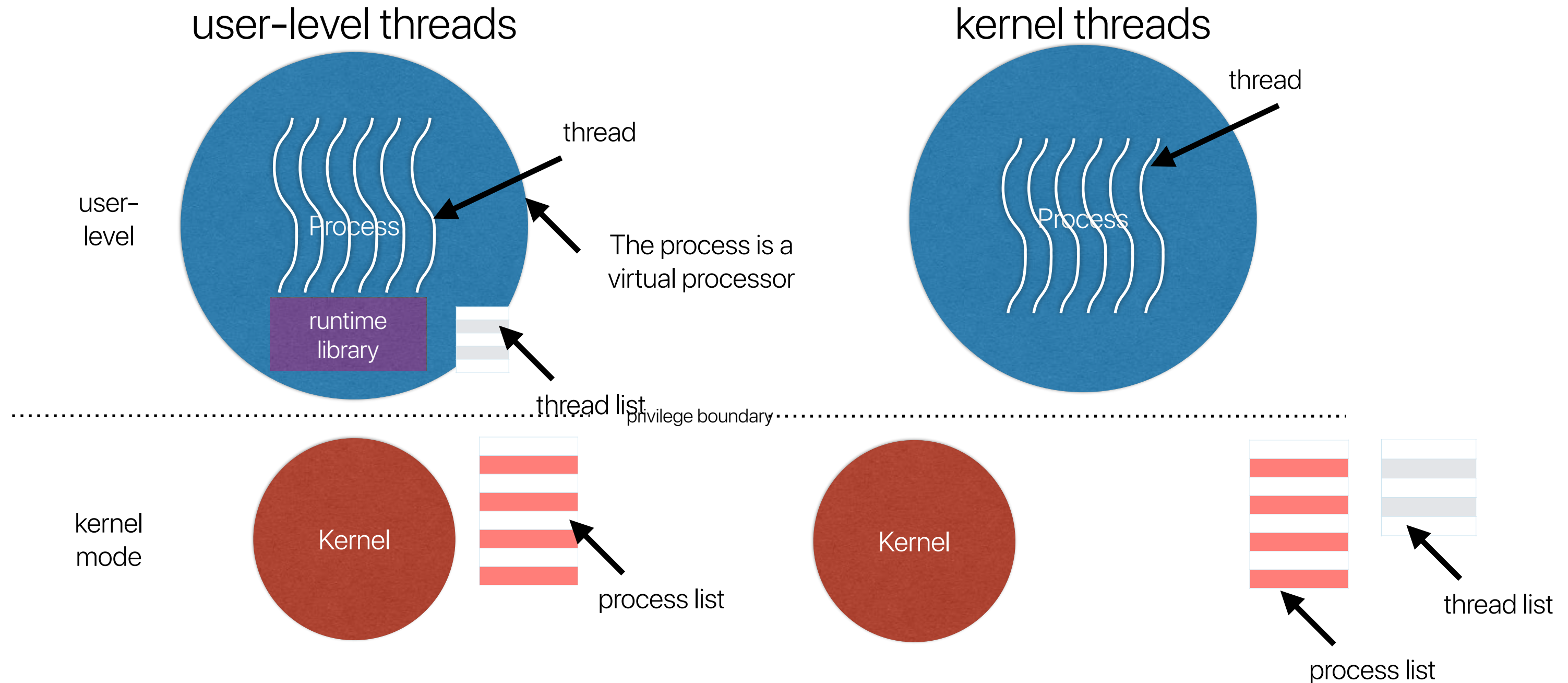
- Real time process classes – always run first (rare)
- Interactive processes: Usually blocked, low total run time, high priority
- Other processes:
  - Red-black BST of process, organized by CPU time they've received.
  - Pick the ready process that has run for the shortest (normalized) time thus far.
  - Run it, update it's CPU usage time, add to tree



# 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

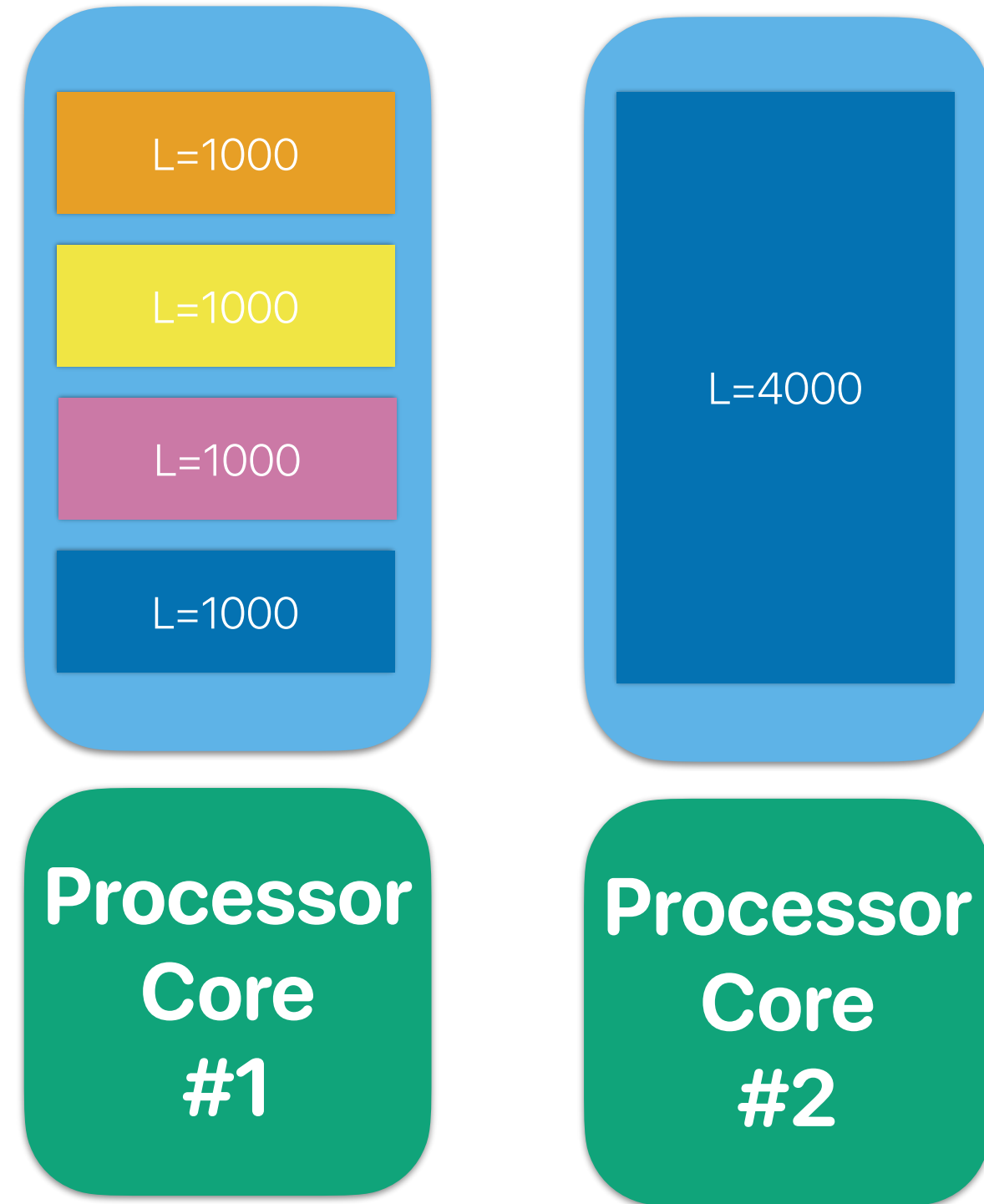
# User-level v.s kernel threads



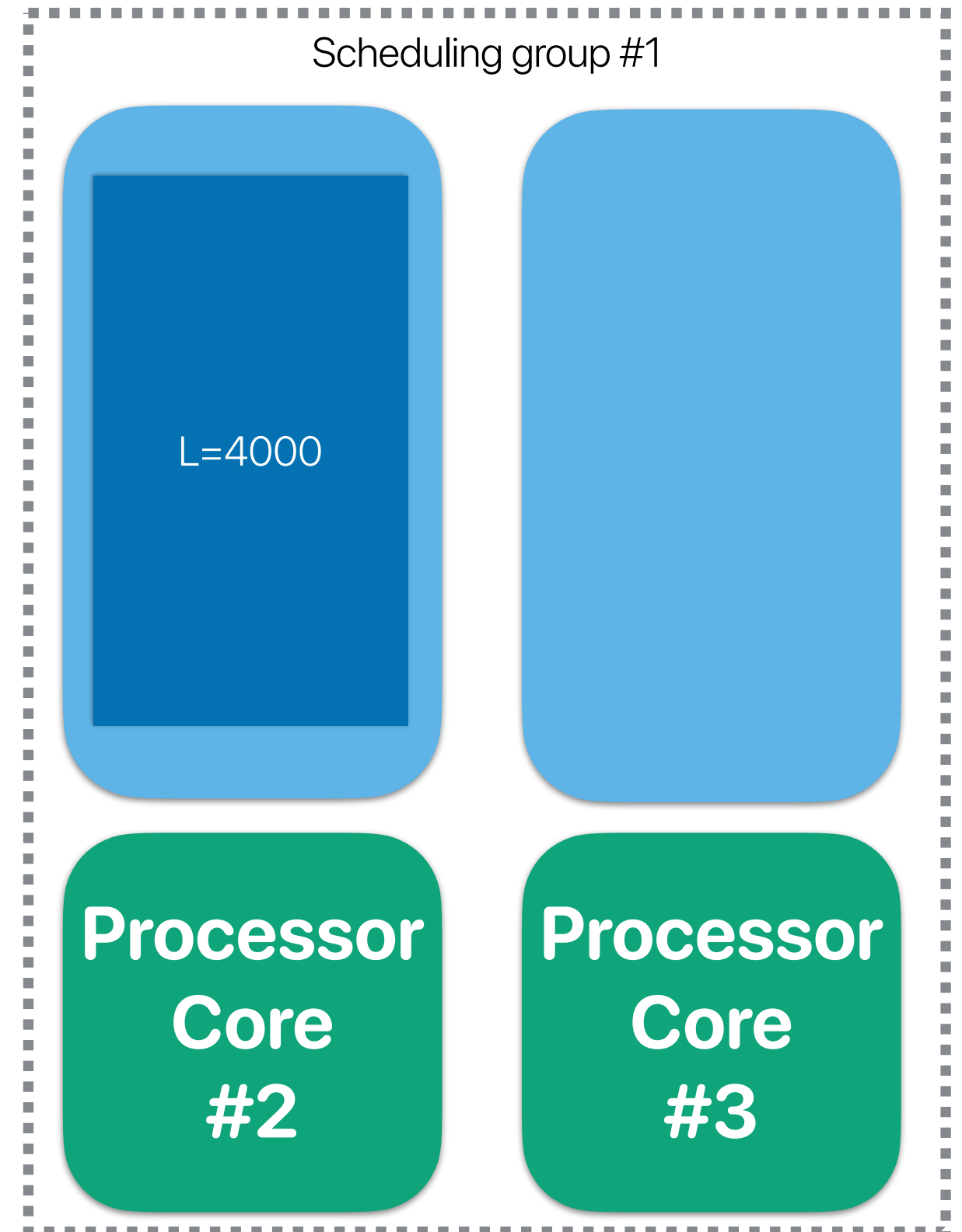
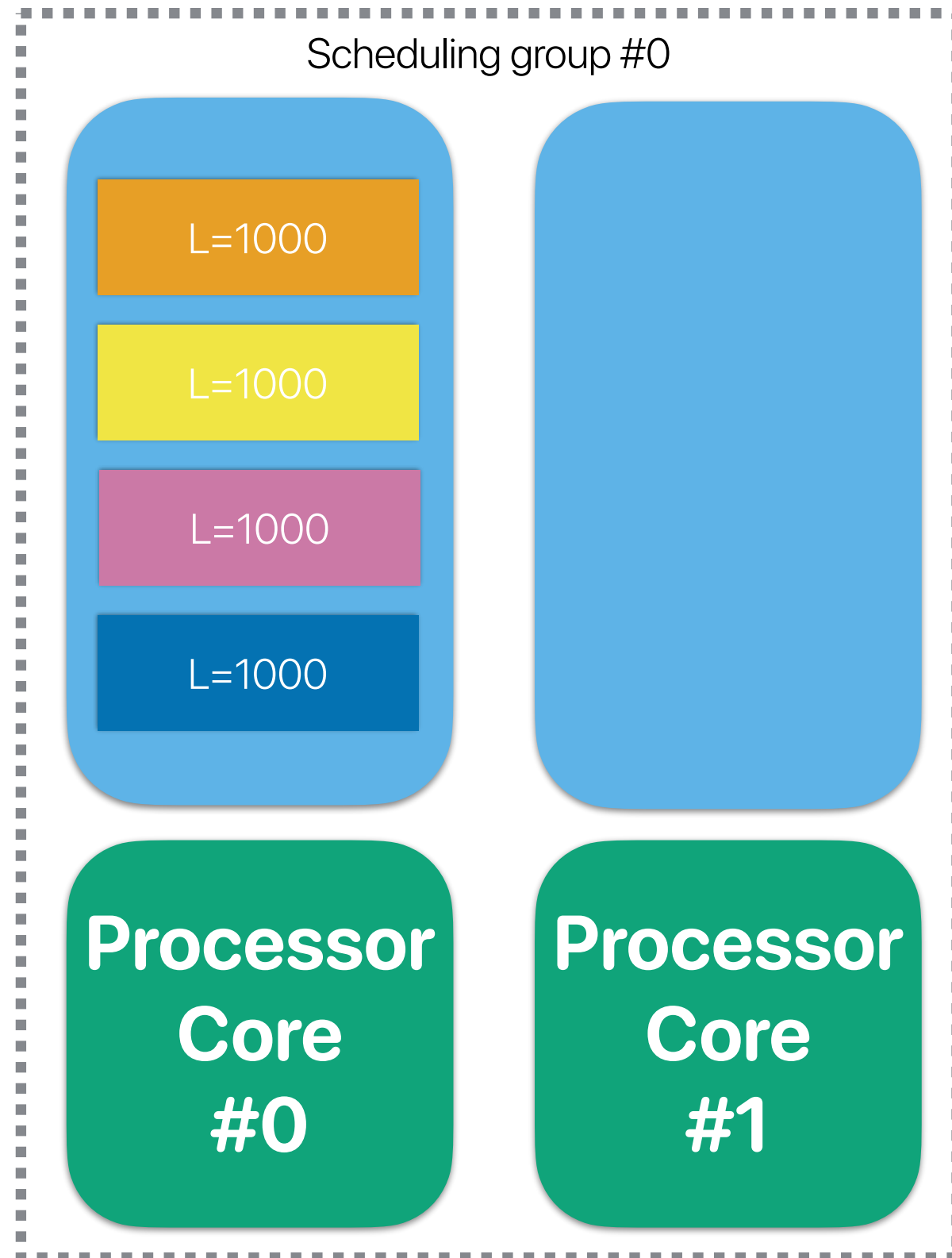
- The OS kernel is unaware of user-level threads
- Switching threads does not require kernel mode operations
- A thread can block other threads within the same process
- The kernel can control threads directly
- Thread switch requires kernel/user mode switch and system calls
- Thread works individually

# Load balancing

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



# Load balancing — if we have more cores?





# Load balancing — if we have more cores?

avg. load = 2000!

avg. load = 2000!

Scheduling group #0

Scheduling group #1

balanced!

balanced!

L=1000

L=1000

L=1000

L=1000

L=4000

idle!!!

Processor  
Core  
#0

Processor  
Core  
#1

Processor  
Core  
#2

Processor  
Core  
#3

# Loading balancing — grouping?

avg. load = 2000!

balanced!

avg. load = 2000!

Scheduling group #0

Scheduling group #2

Scheduling group #1

L=1000

L=1000

L=1000

L=1000

idle!!!

L=4000

Processor  
Core  
#0

Processor  
Core  
#1

Processor  
Core  
#2

Processor  
Core  
#3

# "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**

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].

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!**

**No body knows how they work...**

# **Solution — Lottery and Tickets**

# What does ticket abstraction promote?

- How many of the following can the **ticket** abstraction in the lottery paper promote?
  - ① Proportional fairness
  - ② Machine-independent implementation of the scheduling policy
  - ③ Generic scheduling policy across different devices
  - ④ Starvation free

A. 0  
B. 1  
C. 2  
D. 3  
E. 4

# 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!



# What does ticket abstraction promote?

- How many of the following can the **ticket** abstraction in the lottery paper promote?

- ① Proportional fairness **Tickets represent the share of a process should receive from a resource**
- ② Machine-independent implementation of the scheduling policy
- ③ Generic scheduling policy across different devices **The ticket abstraction can be independent of machine speeds or detail**
- ④ Starvation free **You may use tickets on everything you would like to share**

A. 0 **Eventually every process with a ticket gets to run**  
**It's also state-free — reduce the overhead**

B. 1

C. 2

D. 3

**E. 4**

# 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

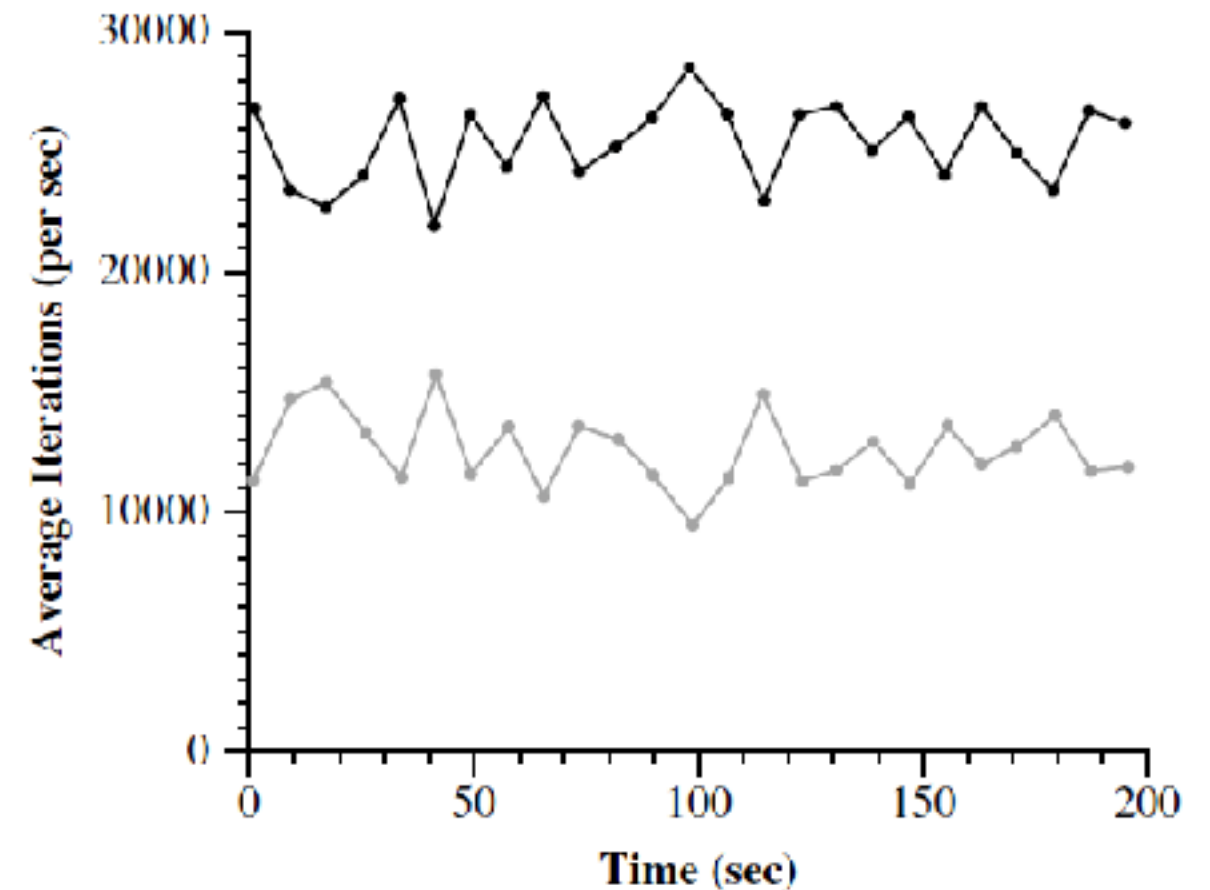


Figure 5: **Fairness Over Time.** Two tasks executing the Dhrystone benchmark with a 2 : 1 ticket allocation. Averaged over the entire run, the two tasks executed 25378 and 12619 iterations/sec., for an actual ratio of 2.01 : 1.

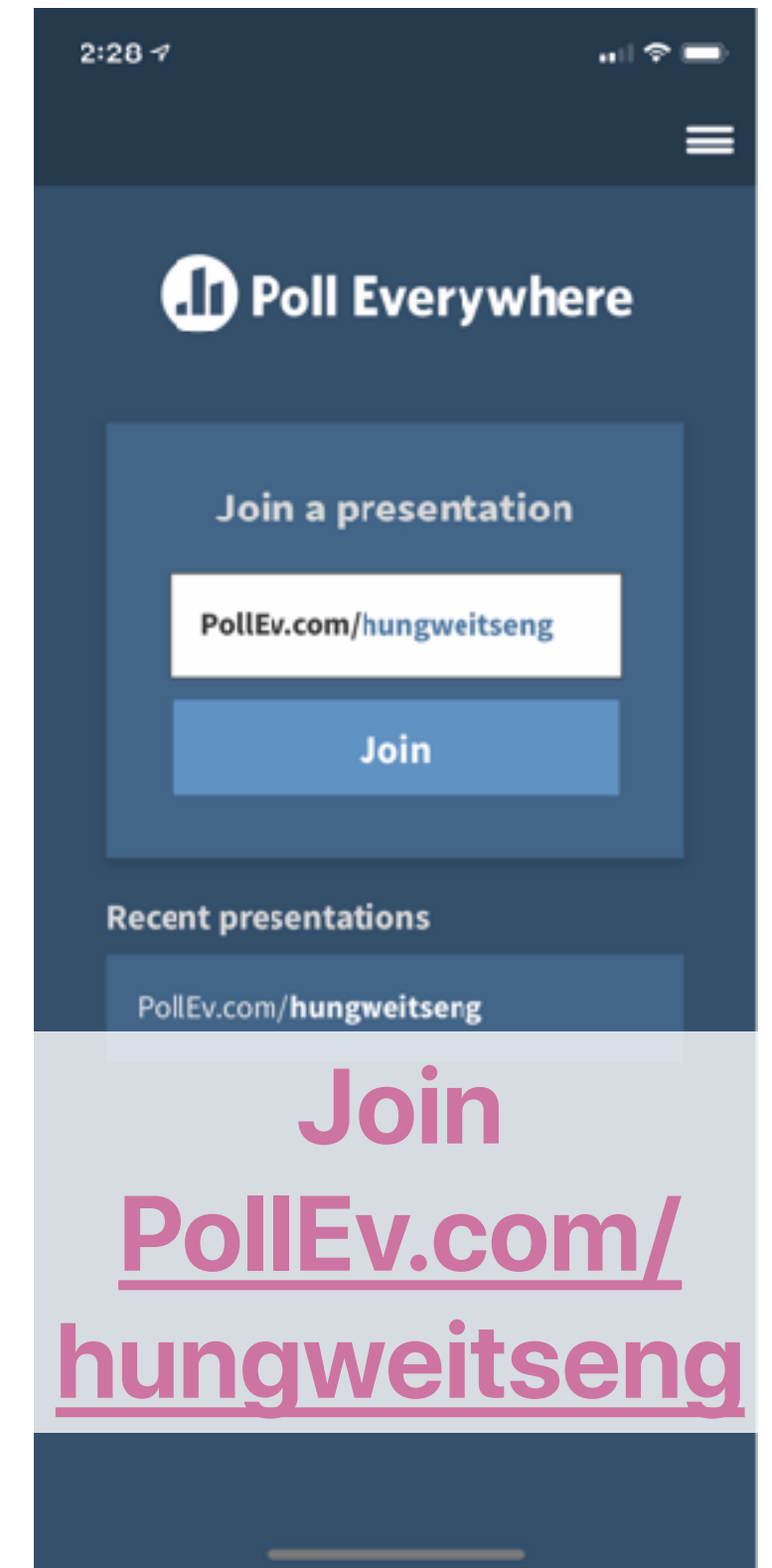
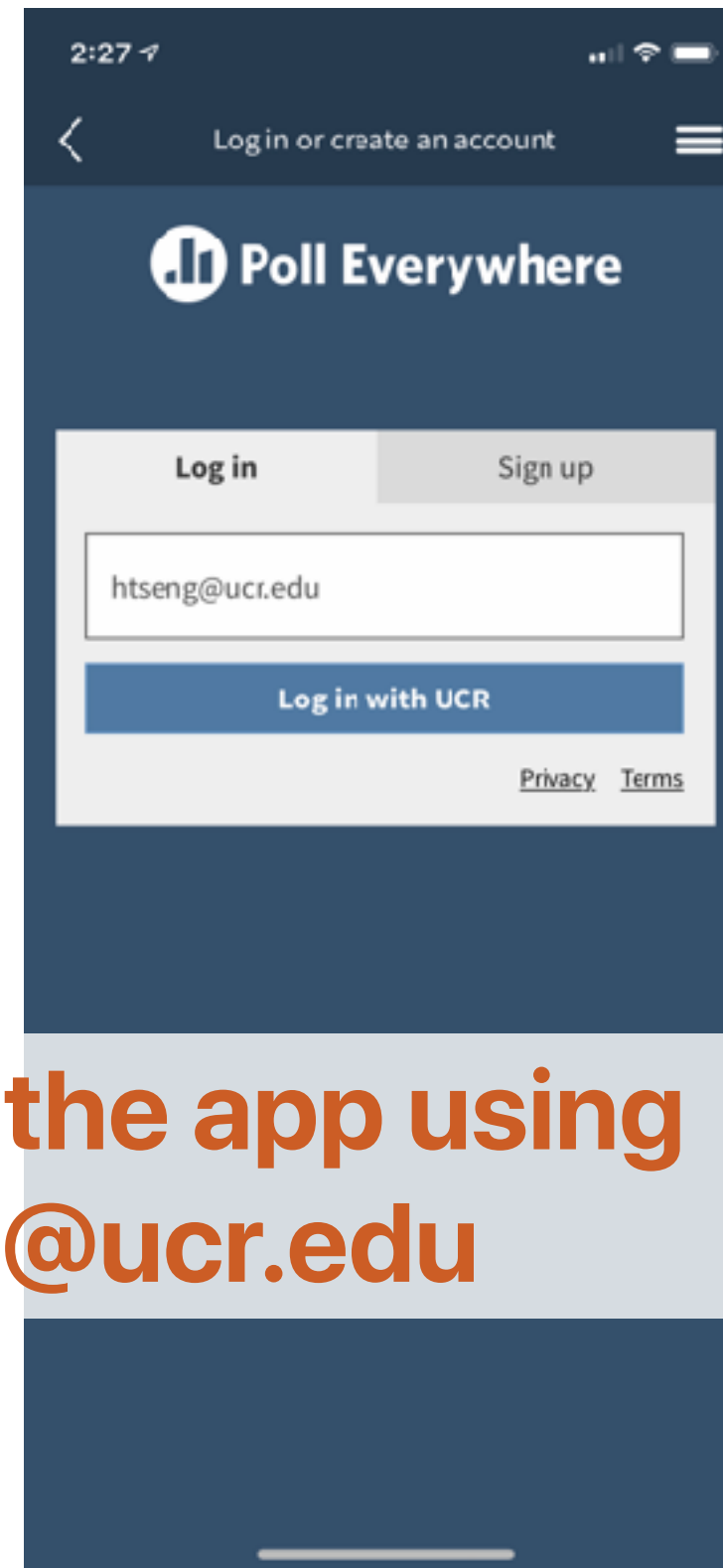
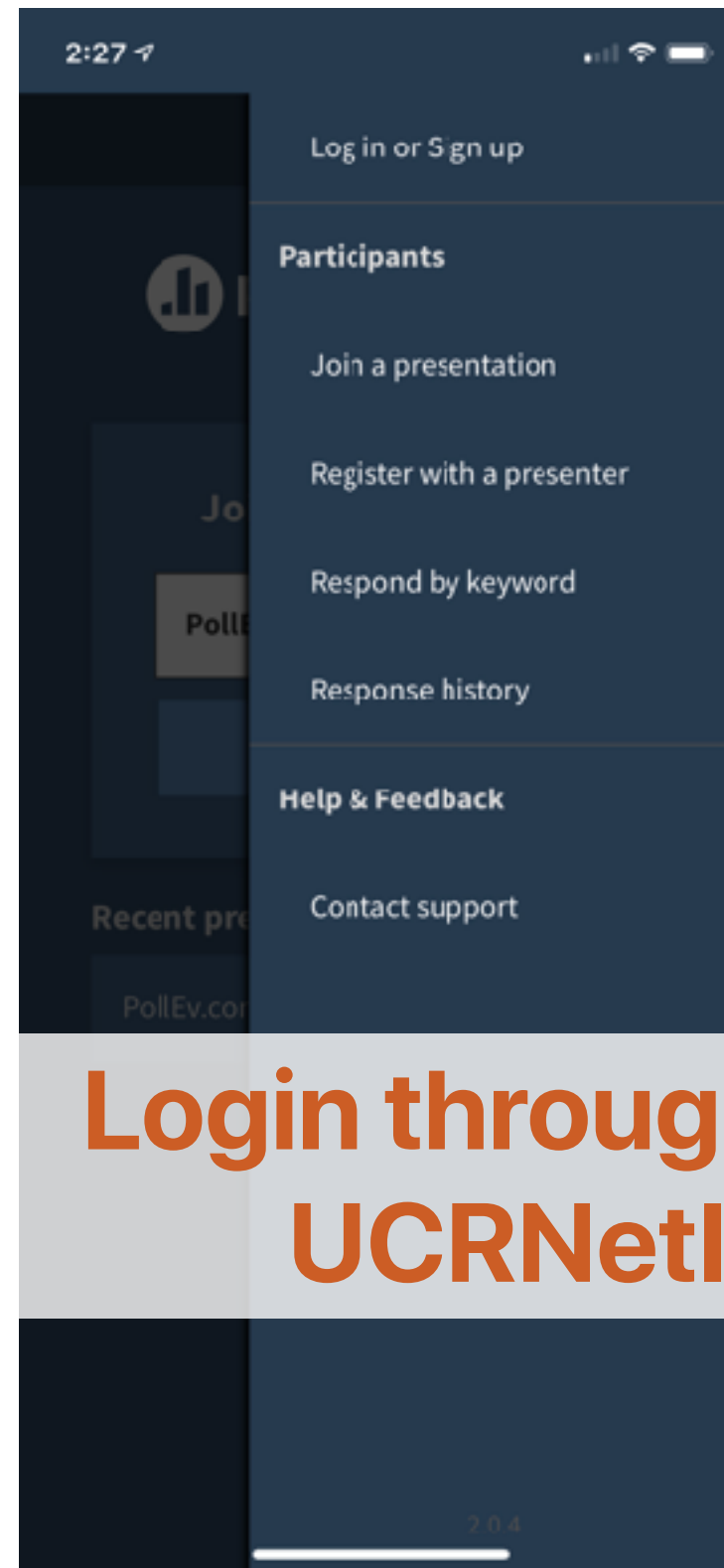
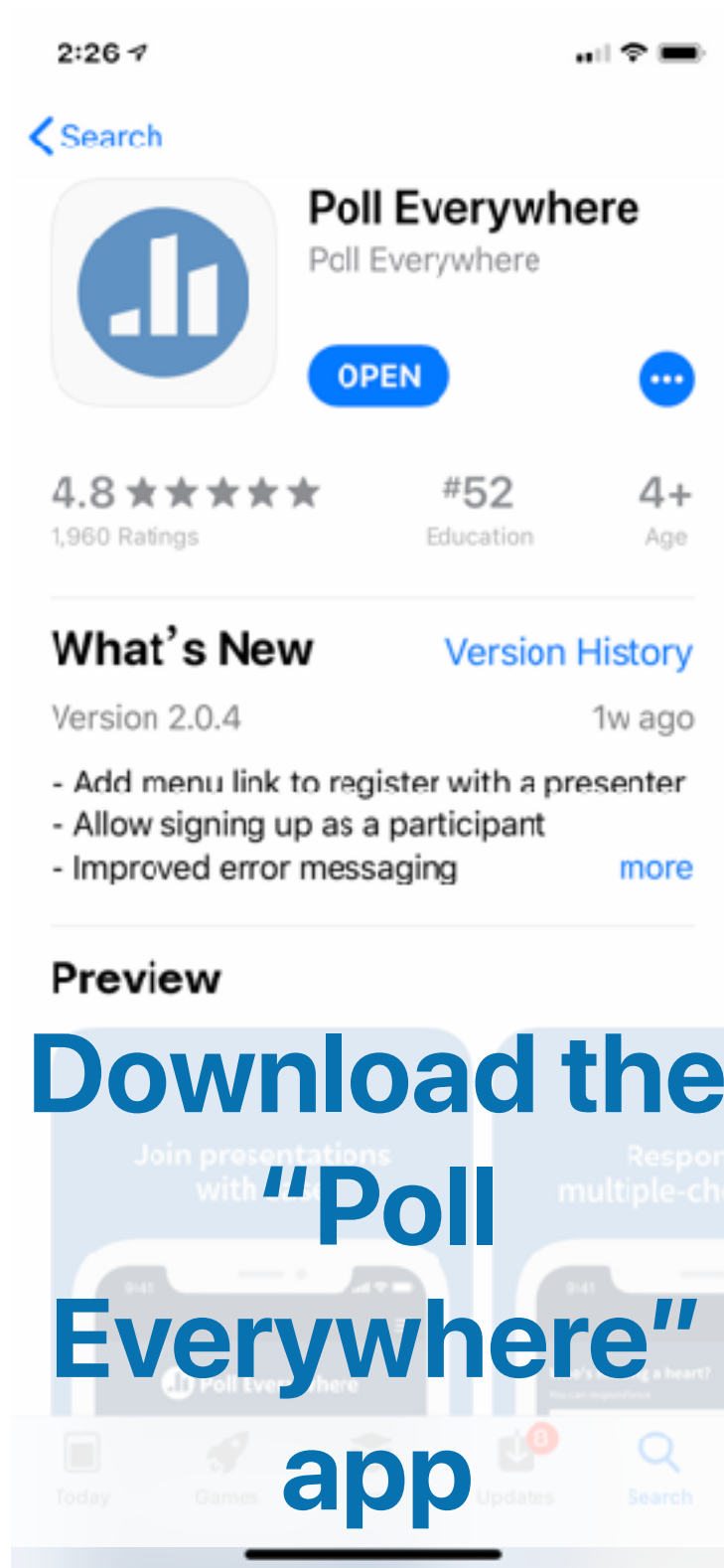
# The impact of "lottery"

- 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

# Setup Poll Everywhere — <https://pollev.com/hungweitseng>



Login through the app using  
**UCRNetID@ucr.edu**

Join  
[PollEv.com/hungweitseng](https://pollev.com/hungweitseng)

# Announcement

- Regarding in-person instructions starting from February
  - We will have live sessions in Material Science Building 103 starting **next Tuesday 2p!**
  - Online options remain open — Zoom or youtube
  - Please setup “Poll everywhere” before attending lectures — both in-person and online — by next Tuesday as well
- Midterm
  - Will release on 2/10/2021 0:00am and due on 2/11/2021 11:59:00pm
  - You will have to find a consecutive, non-stop 80-minute slot with this period
  - One time, cannot reinitiate — please make sure you have a stable system and network
  - **No late submission is allowed**
- Project released — <https://github.com/hungweitseng/CS202-MMA>
  - Groups in 2. 3 is acceptable, but not recommended
  - **Pull the latest version — had some changes for later kernel versions**
  - **Install an Ubuntu Linux 20.04 VM as soon as you can!**
  - **Please do not use a real machine — you may not be able to reboot again**
  - Need help? Check for office hours — [https://calendar.google.com/calendar/u/0/r?cid=ucr.edu\\_b8u6dvdvkretn6kq6igunlc6bldg@group.calendar.google.com](https://calendar.google.com/calendar/u/0/r?cid=ucr.edu_b8u6dvdvkretn6kq6igunlc6bldg@group.calendar.google.com)
- Reading quizzes due next Tuesday

# Computer Science & Engineering

# 202

# つづく

