Process/Thread/Task Scheduling

Hung-Wei Tseng

Outline

- Mechanisms of changing processes
- Basic scheduling policies
- An experimental time-sharing system — The Multi-Level Scheduling Algorithm
- Scheduler Activations
- Getting locks done correctly with modern OS scheduling

The mechanisms of changing processes

The mechanisms of changing the running processes

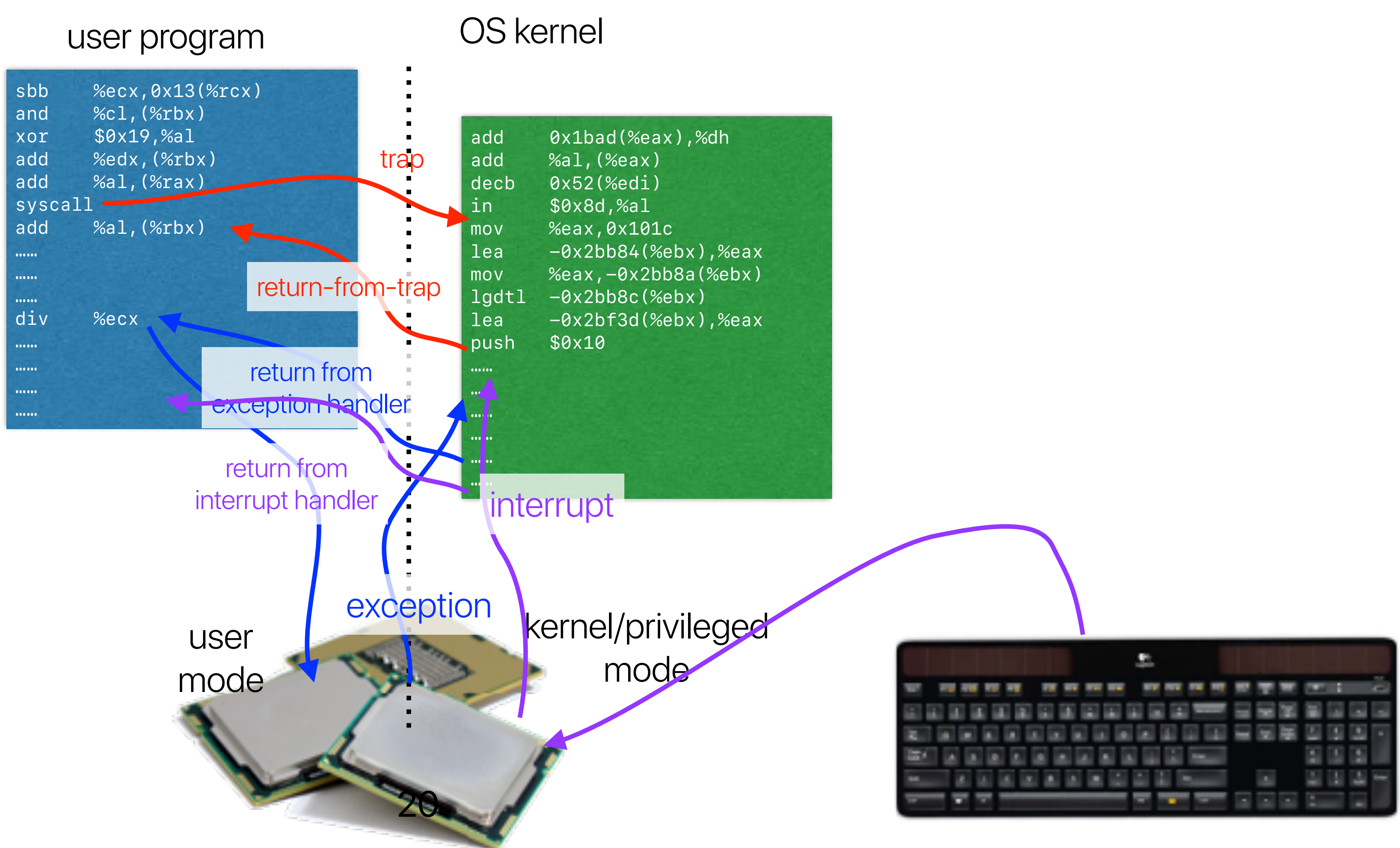
- Cooperative Multitasking (non-preemptive multitasking)
- Preemptive Multitasking

Preemptive Multitasking

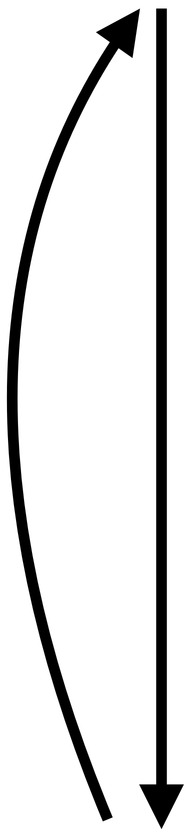
- The OS controls the scheduling — can change the running process even though the process does not give up the resource
- But how?

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

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

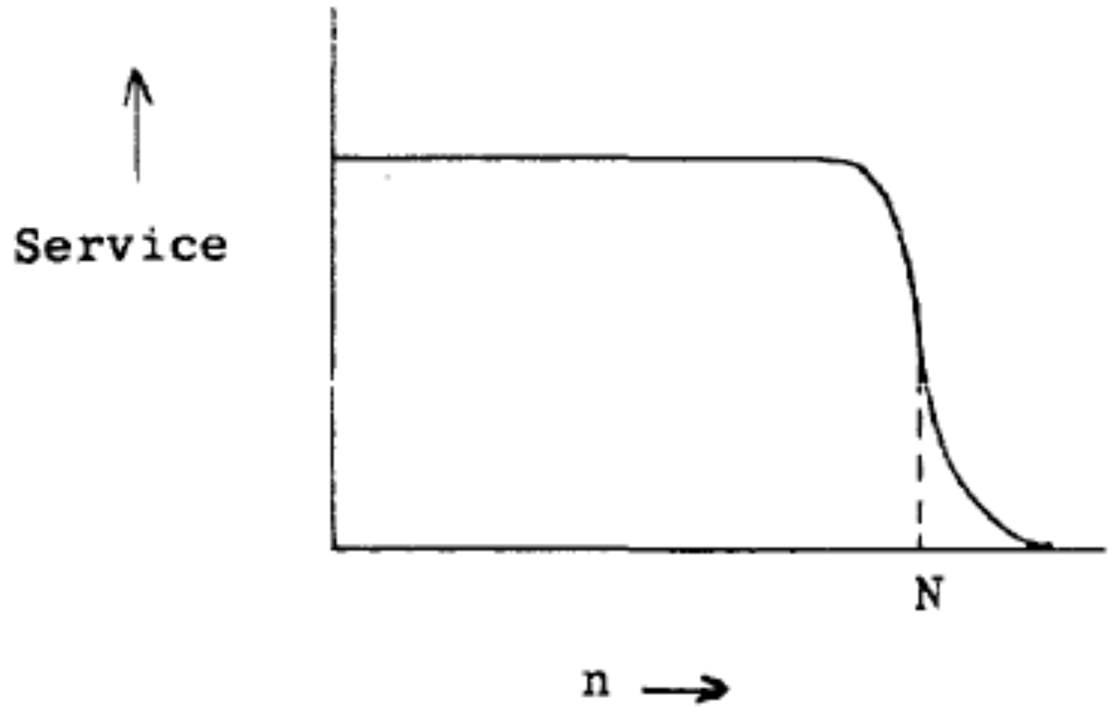
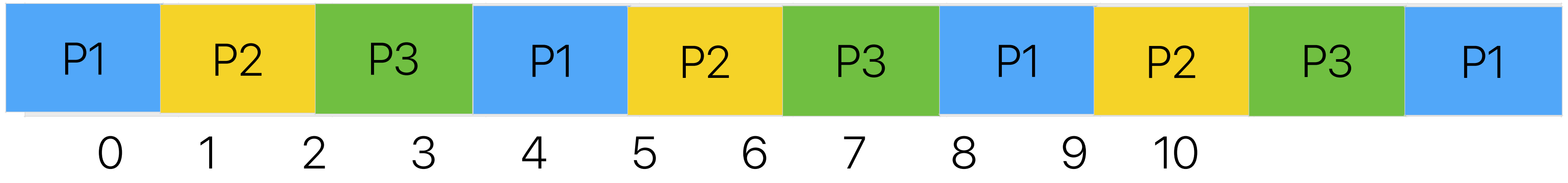


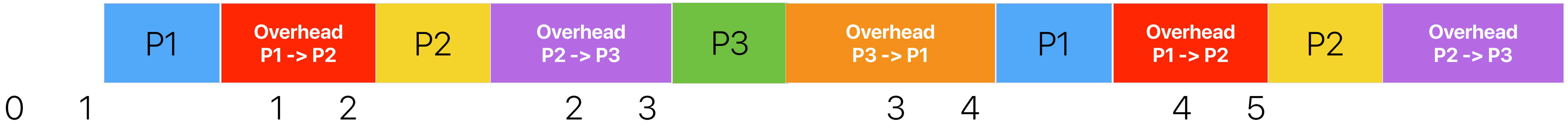
Figure 1. Service vs. Number of Active Users

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

$$\ell_0 = \left[\log_2\left(\left[\frac{w_p}{w_q}\right] + 1\right)\right]$$

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

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

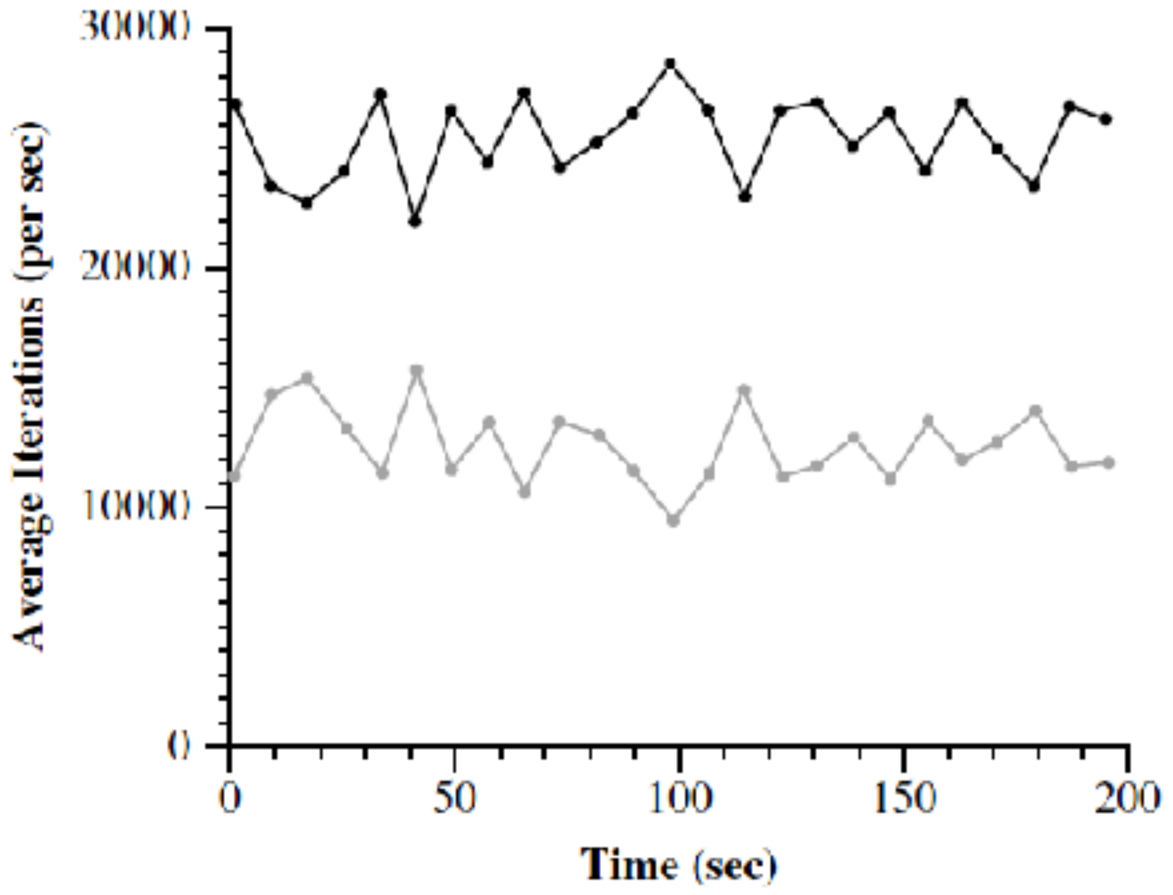


Figure 5: **Fairness Over Time.** Two tasks executing the Dhrystone benchmark with a 2 : 1 ticket allocation. Averaged over the entire run, the two tasks executed 25378 and 12619 iterations/sec., for an actual ratio of 2.01 : 1.

The impact of “lottery”

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

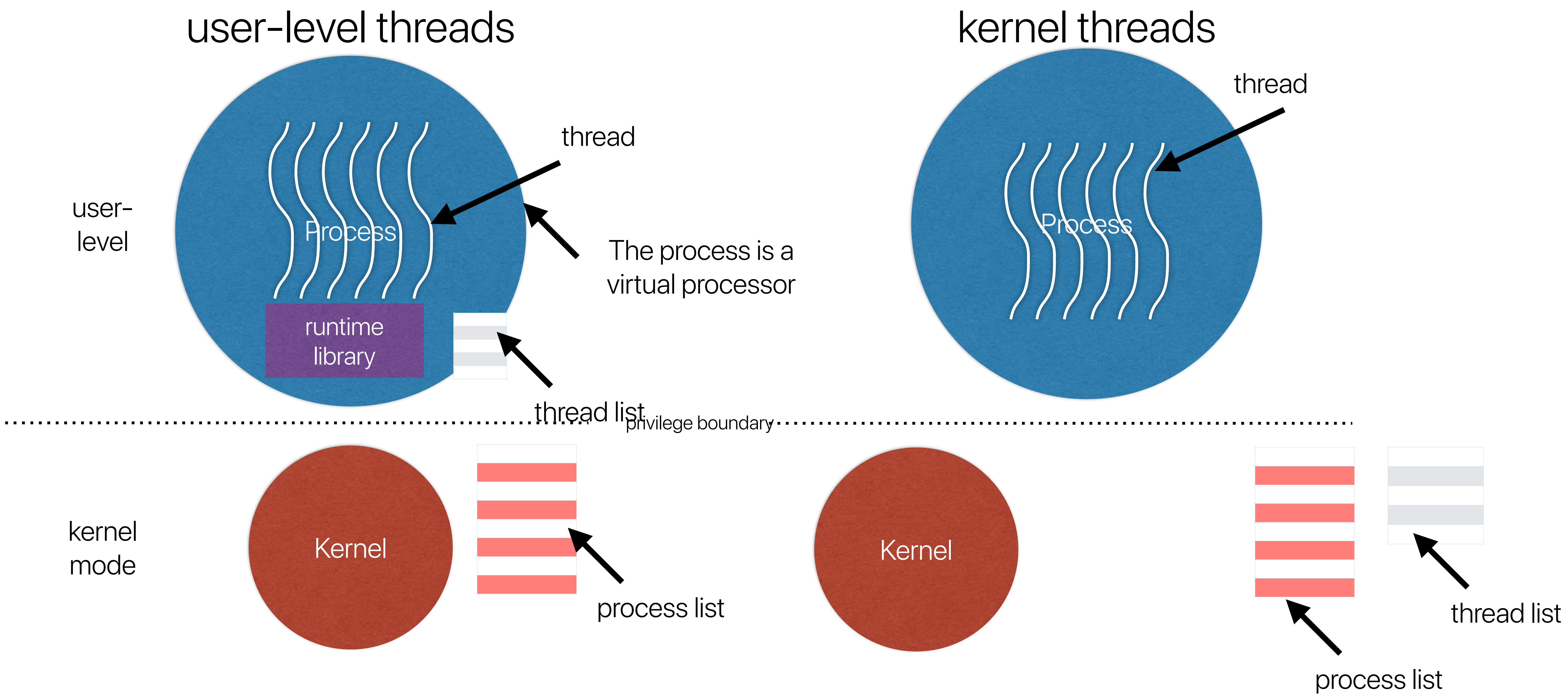
Will you use lottery for your system?

- 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

Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism

Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska and Henry M. Levy
University of Washington

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

Why — the "dilemma" of thread implementations

- User-level threads
  - Efficient, flexible, safer, customizable
- Kernel threads
  - Slower, more powerful
  - Better matches the multiprocessor hardware
- Problems
  - OS is only aware of kernel threads
  - OS is unaware of user-level threads as they are hidden behind each process

What does “Scheduler Activations” propose?

- The OS kernel provides each user-level thread system with its own virtual multiprocessor
- Communication mechanism between kernel and user-level

The virtual multiprocessor abstraction

- The kernel allocates processors to an address space/process
  - An address space is shared by all threads within the same process
  - The kernel controls the number of processors to an address space
- Each process has complete control over the processor-thread allocation
- The kernel notifies the address space when the allocated number of processors changes
- The process notifies the kernel when it needs more or fewer processors
- Transparent to users/programmers

How scheduler activation works?

- Create a scheduler activation when the system create a process on a processor
- Create a scheduler activation when the kernel needs to perform an "upcall" user-level
  - Add a processor
  - Processor has been preempted
  - Scheduler activation has blocked
  - Scheduler activation has unblocked
- Downcalls — hints for kernel to perform resource management
  - Add more processors
  - This processor is idle
- Key difference from a kernel thread
  - Kernel never restarts user thread after it is blocked

Will you use Scheduler activation?

- Once been implemented in NetBSD, FreeBSD, Linux
- A user-level thread gets preempted whenever there is scheduling-related event
  - Overhead
  - You may preempt a performance critical thread
- Blocking system call

Linux's thread implementation

- Linux treat all schedule identities as "tasks" — context of executions
- COEs can share parts of their contexts with each
  - Processes share nothing
  - Threads share everything but the CPU states
- http://www.evanjones.ca/software/threading-linus-msg.html