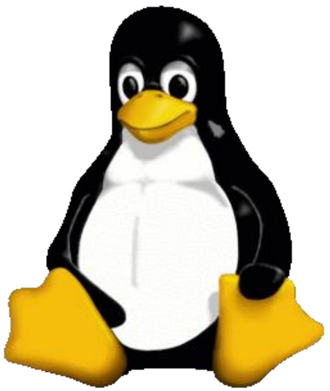
KOÇ UNIVERSITY



Real-Time CPU Scheduling and Linux Scheduler

Hakan Ayral

Lecture 8

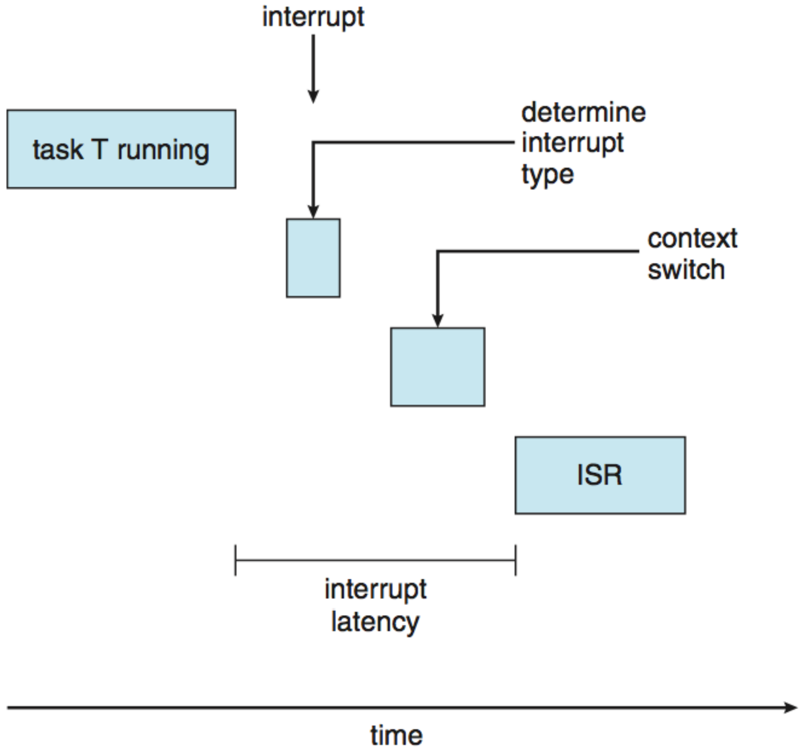
COMP304 - Operating Systems (OS)

Real-Time CPU Scheduler

- **Real-time programs** must guarantee response within strict time constraints, often referred to as deadlines
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled, degrades the system's quality of service
  - Ex: the flight plan updates for an airline, live broadcasting
- **Hard real-time systems** – missing a deadline is a total system failure
  - Mission critical: a real-time deadline must be met, regardless of system load
  - Ex: Anti-lock brakes on a car, heart pacemakers and many medical devices
- Not all the Operating Systems are real-time operating systems

Real-Time CPU Scheduling

- **Event latency:** time that elapses from when an event occurs to when it is serviced
- Two types of latencies affect performance
  1. **Interrupt latency** – time from arrival of interrupt to start of routine that services the interrupt
  2. **Dispatch latency** – time for schedule to take current process off from CPU and switch to another



ISR Stands for "*Interrupt* Service Routine.

Real-Time OSs

- **Event driven systems** switch between tasks based on their priorities while time sharing systems switch the task based on clock interrupts.
- Design goal is not high throughput, but rather a guarantee of service for a high priority job
- Real-time OS is more frequently dedicated to a narrow set of applications.
  - Targeted usage is typically embedded systems, robotics etc.
  - Supporting industrial, automotive, smart city and smart home
- Some open-source real-time OSs:
  - Zephyr : https://www.zephyrproject.org/
  - uKOS
  - FreeRTOS
- http://www.wikiwand.com/en/Comparison_of_real-time_operating_systems

Zephyr (an example RTOS)

- **Extensive suite of Kernel services**:
  - *Multi-threading Services* for cooperative, priority-based, threads with optional round robin time-slicing. Includes POSIX pthreads compatible API support.
  - *Interrupt Services* for compile-time registration of interrupt handlers.
  - *Memory Allocation Services* for dynamic allocation and freeing of fixed-size or variable-size memory blocks.
  - *Inter-thread Synchronization Services* for binary semaphores, counting semaphores, and mutex semaphores.
  - *Inter-thread Data Passing Services* for basic message queues …
  - *Power Management Services* such as tickless idle and an advanced idling infrastructure.
- **Multiple Scheduling Algorithms**:
  - Preemptive Scheduling
  - Earliest Deadline First (EDF)
  - Timeslicing: Enables time slicing between preemptible threads of equal priority
  - Multiple queuing strategies:
    - Simple linked-list ready queue
    - Red/black tree ready queue
    - Traditional multi-queue ready queue

https://docs.zephyrproject.org/latest/introduction/index.html

Linux Scheduler

History

Linux Version	Scheduler
Pre 2.5	Multi-level Feedback Queue
Pre 2.6.23	O(1) Scheduler
Post 2.6.23	Completely fair scheduler

Basic Philosophies in Linux

- Priority is the primary scheduling mechanism
- Priority is *dynamically adjusted* at run time

- Try to distinguish interactive processes from non-interactive ones
- Use large time quanta for important processes
  - Modify quanta based on CPU usage for the next run
- Associate processes to CPUs in a multicore systems
  - Process affinity

Priority

- Each task has a static priority that is set based upon the nice value specified by the task.
  - *static_prio* in *task_struct*
  - *Default is 120*
- For normal tasks, the static priority is 100 + nice.

- Each task has a dynamic priority that is set based upon a number of factors
  - *prio* in *task_struct*

Niceness

- Niceness
  - a process is nicer to others if it has a higher nice value
  - Default is inherited from its parent (usually 0)
  - Ranges from -20 to +19
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139

Value can be set via **nice**() system call or **nice** command

```
bash$ nice -n 19 tar cvzf archive.tgz largefile
```

Prior to Kernel 2.5

In the 2.4 kernel, this was the scheduling algorithm:

- Each task got a number of CPU ticks (*jiffies*) made available to the task each scheduling interval, or epoch.

- The number of new ticks given was determined from the nice value for the task. It was roughly:

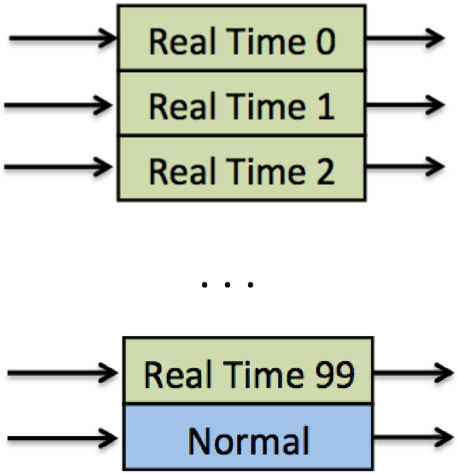
  ((20-nice)*HZ/800) + 1.

- Each task had a *counter*, which was the number of CPU ticks still left for the task to use in the current epoch.

- Unused ticks in a particular epoch decayed by 50% for use in the next interval.

Linux O(1) Scheduler

- Version 2.5 moved to constant order *O*(1) scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **normal** (time-sharing) range from 100 to 139
  - Higher priority gets larger time quantum
  - Scales well with the number of processes



Real-Time Scheduling

- Linux has a soft real-time scheduler
  - No hard real-time guarantees
  - All real-time processes are higher priority than any normal processes
- Processes with priorities [0, 99] are real-time
  - saved in *rt_priority* in the *task_struct*
  - scheduling priority of a real time task is: 99 - *rt_priority*
- A process can be converted to real-time via *sched_setscheduler* system call

Scheduling Policies

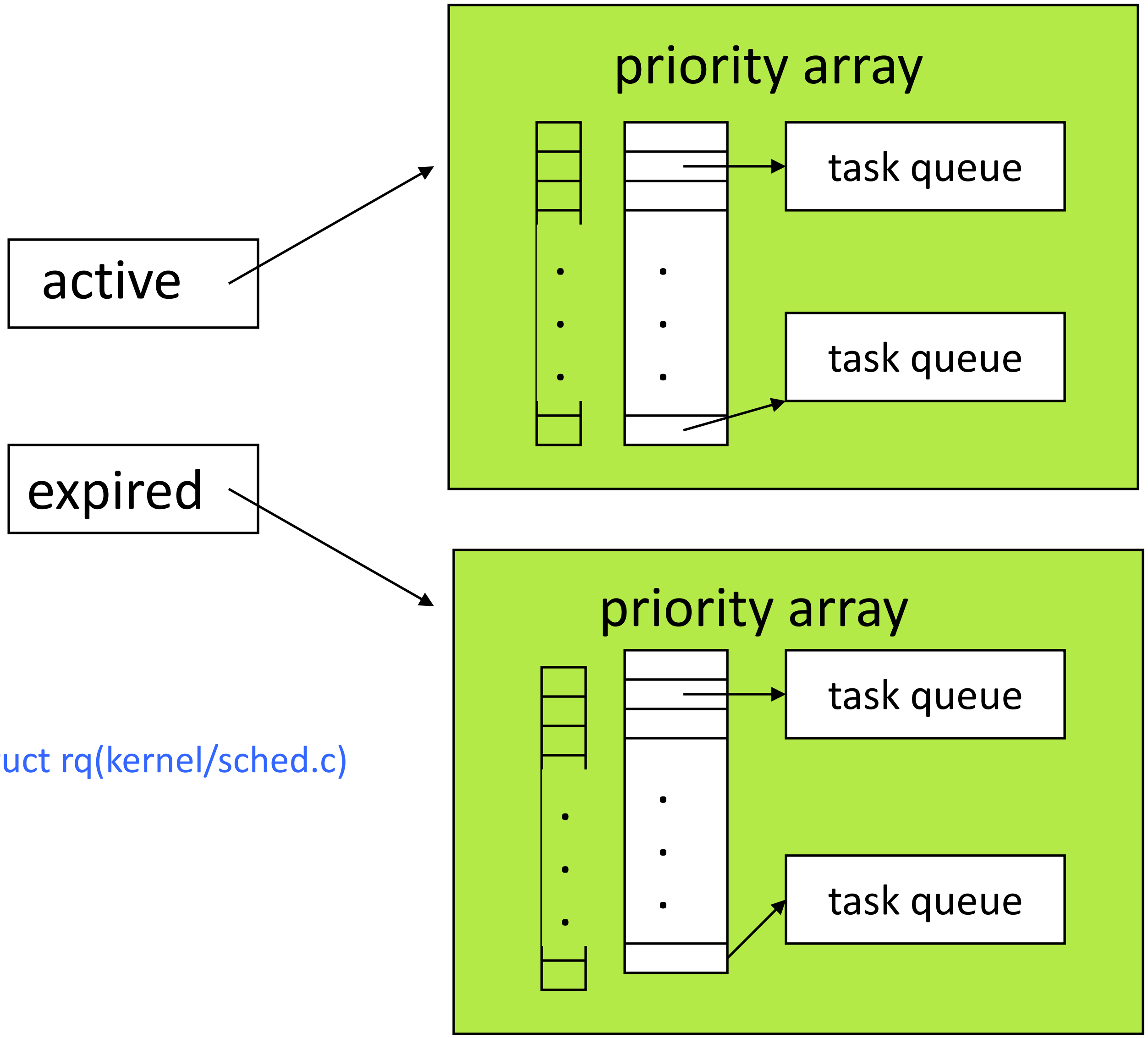
- Real-time processes
  - First-in, first-out: SCHED_FIFO
    - Static priority
    - Process is only preempted for a higher-priority process
    - No time quanta; it runs until it blocks or yields voluntarily
  - Round-robin: SCHED_RR
    - RR within the same priority level
    - A time quanta (800 ms)
- Normal processes have
  - SCHED_OTHER: standard processes
  - SCHED_BATCH: batch style processes
  - SCHED_IDLE: low priority tasks

O(1) Scheduler

- Task runnable as long as time left in time slice (**active**)
- If no time left (**expired**), not runnable until all other tasks use their slices
- All runnable tasks tracked in per-CPU **runqueue** data structure
  - Two priority arrays (active, expired)
  - When no more active, arrays are swapped

Runqueues

140 separate queues, one for each priority level in two sets: active and expired



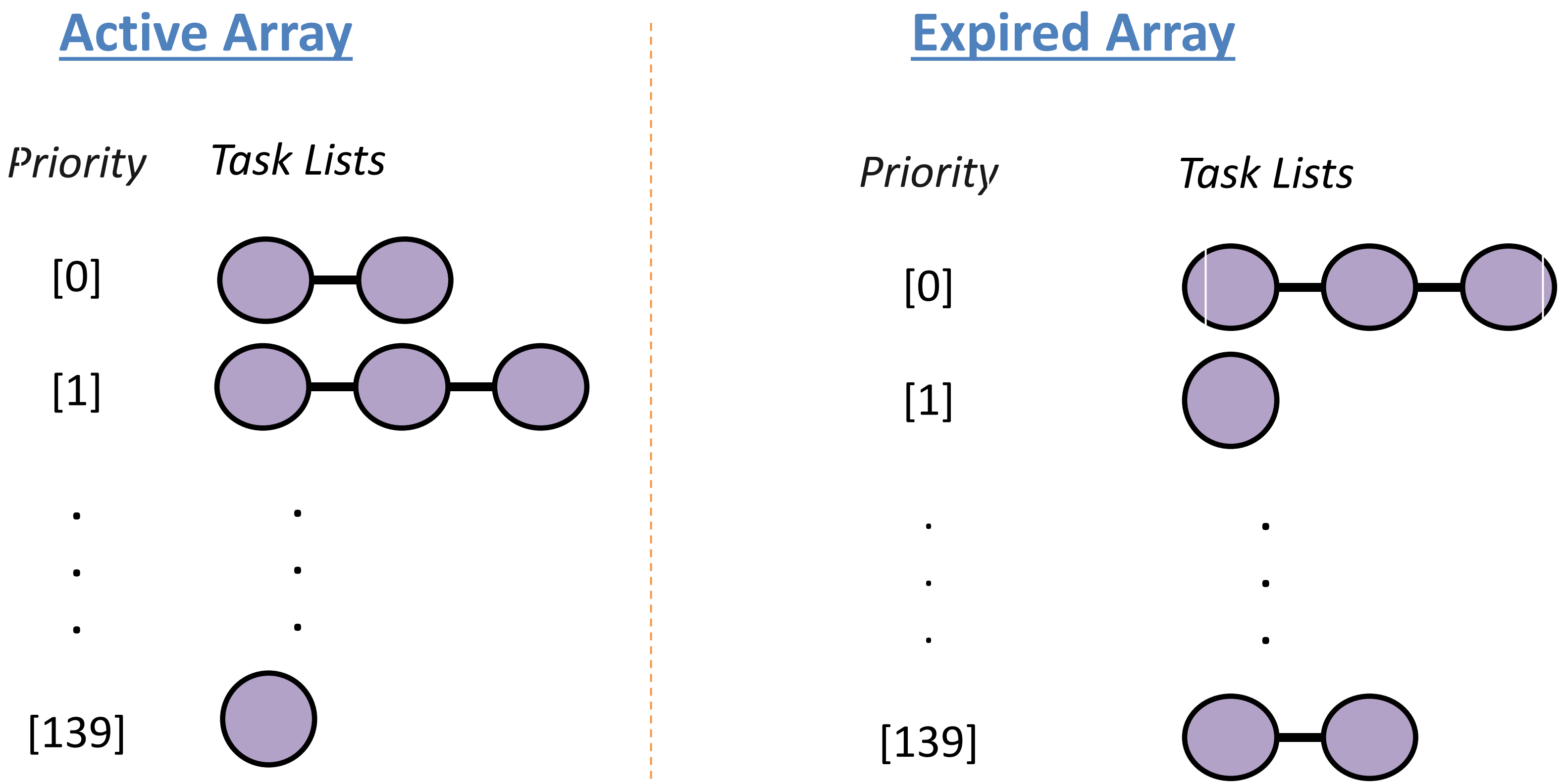
Higher priority
more I/O
800ms quanta

lower priority
more CPU
10ms quanta

Struct rq(kernel/sched.c)

Runqueues

- Two arrays of priority queues: **active** and **expired**
  - Total 140 priorities [0, 140)
  - Smaller integer = higher priority



Scheduling Algorithm for Normal Processes

- Find the highest-priority non-empty queue in rq->active; if none, simulate aging by swapping active with expired
- Next = Find the first process on that queue
- Calculate next's quantum size and its next's priority
- Context switch to next
- Let it run
- When its time is up, put it on the expired list
- Repeat

Simulate Aging

- After running all of the active queues, the active and expired queues are swapped
- There are pointers to the current arrays; at the end of a cycle, the pointers are switched
- Swapping active and expired gives low priority processes a chance to run
- Advantage: O(1)
  - Processes are touched only when they start or stop running

Find highest priority non-empty queue

- Time complexity O(1)
  - Depends on the number of priority levels, not the number of processes
- Implementation: a bitmap for fast look up
  - 140 queues
  - A few comparisons to find the first non-zero bit

Calculating Time Slices

- *time_slice* in the *task_struct*
- Calculate Quantum where
  - If $(SP < 120)$: $Quantum = (140 - SP) \times 20$
  - if $(SP >= 120)$: $Quantum = (140 - SP) \times 5$

  where SP is the *static priority*
- Higher priority process gets longer quanta
- Basic idea: important processes should run longer

Typical Quanta

Priority:	Static Pri	Niceness	Quantum
Highest	100	-20	800 ms
High	110	-10	600 ms
Normal	120	0	100 ms
Low	130	10	50 ms
Lowest	139	19	5 ms

Issues with O(1) RR Scheduler

- Not easy to distinguish between CPU and I/O bound
  - I/O bound typically needs better interactivity
- Finding right time slice isn’t easy
  - Too small: good for I/O bound but high overhead
  - Too large: good for CPU bound but poor interactivity
- Priority is relative but time slice is absolute
  - Nice 0, 1: time slice 100 and 95 msec: 5% difference
  - Nice 18,19: time slice 10 and 5 msec: 100 % difference

Completely Fair Scheduler (CFS)

- Starting from Linux kernel version 2.6.23 since 2007
- Not based on runqueues as in O(1) scheduler
- Not based on time slices

- Note that CFS is used only for normal processes, for real-time processes, Linux still use priority based FCFS and RR schedulers

Completely Fair Scheduler (CFS)

- Core ideas: dynamic time slice and order
- Don’t use fixed time slice per task
  - Instead, fixed time slice across all tasks
  - Scheduling Latency
- Don’t use round robin to pick next task
  - Pick task which has <u>received the least CPU time</u> so far
  - Equivalent to dynamic priority

CFS

- CFS calculates how long a process should run as a function of the total number of runnable processes.
  - If there are N runnable processes, then each should be afforded 1/N of the processor's time.
  - CFS adjusts the allotment by weighting each process's allotment by its nice value.
  - Small nice value => higher weight
  - Large nice value => lower weight
  - Then process's time slice is proportional to its weight divided by the total weight of all runnable processes.

Timeslice(task) = Timeslice(t) * prio(t) / Sum_all_t'(prio(t'))

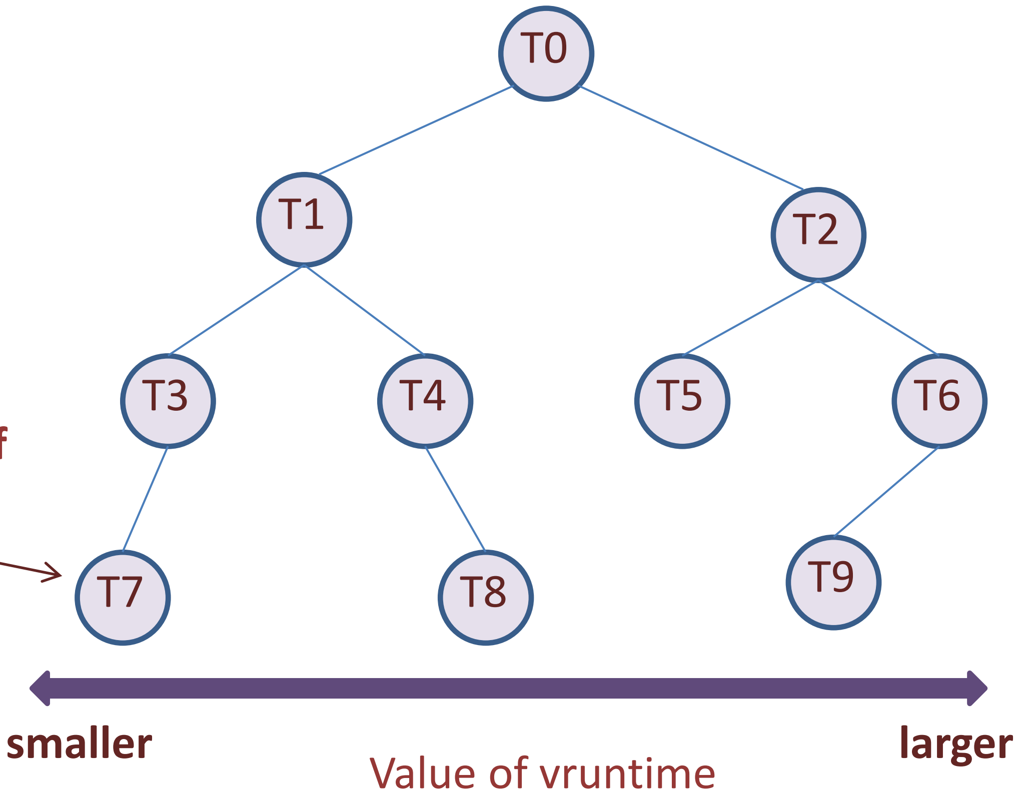
Timeslice (t) = latency / nr_tasks

CFS Tree

- Each runnable task is placed in a red-black tree
  - A balanced binary search tree whose key is based on the value of vruntime (task with min runtime so far)

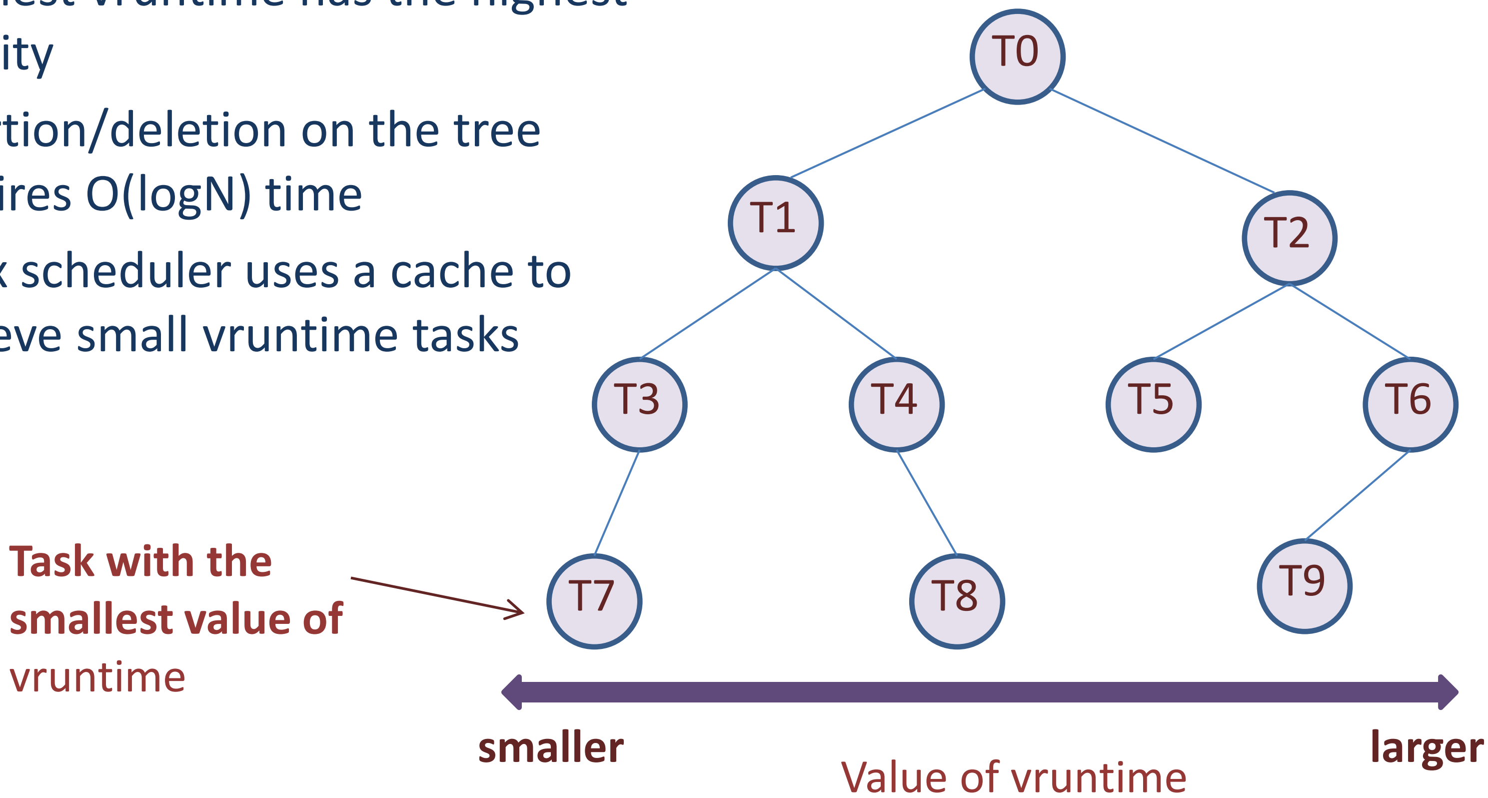
sched_entity in task_struct

Task with the smallest value of vruntime



CFS Tree

- When a task becomes runnable, it is added to the tree (red/black tree)
- Not runnable tasks (e.g. waiting for I/O) are removed from the tree
- Smallest vruntime has the highest priority
- Insertion/deletion on the tree requires O(logN) time
- Linux scheduler uses a cache to retrieve small vruntime tasks



CFS (con.t)

- Two tasks have the same nice values
- One task is I/O bound, other is CPU-bound
  - I/O bound normally runs for a short period before it is interrupted for an I/O operation
  - CPU-bound normally exhausts all its quantum

- Vruntime will eventually be lower for the I/O bound task than for the CPU-bound task
  - Thus I/O bound will get access to CPU more often
  - Vruntime is weighted by process priority

Picking the next process

- Pick task with minimum runtime so far
- Every time process runs for *t* ns
  - Vruntime += t
- How does this impact I/O vs CPU bound tasks?
  - Task A needs CPU for 1 msec every 100 msec (I/O bound)
  - Task B, C need CPU for 80 msec every 100 msec (CPU bound)
  - After 10 times that A, B and C have been scheduled.
    - Vruntime(A) = 10
    - Vruntime(B,C) = 800
    - Overtime task A gets priority, but it quickly releases CPU.

CFS Algorithm

- The leftmost node of the scheduling tree is chosen (as it will have the lowest spent *execution time*), and sent for execution.
- If the process simply completes execution, it is removed from the system and scheduling tree.
- If the process reaches its *maximum execution time* or is otherwise stopped (voluntarily or via interrupt) it is reinserted into the scheduling tree based on its new spent *execution time*.
- The new leftmost node will then be selected from the tree, repeating the iteration.

Choosing a task can be done in constant time, but reinserting a task after it has run requires O(log N) operations

Multiprocessor Scheduling

- Each processor maintains a red/black tree
- Each processor only selects processes from its own tree to run
- It's possible for one processor to be idle while others have jobs waiting in their run queues
- Periodically, rebalance
  - void load_balance()!
    - Attempts to move tasks from one CPU to another

Processor Affinity

- Each process has a bitmask saying what CPUs it can run on
- Normally, of course, all CPUs are listed
- Processes can change the mask
- The mask is inherited by child processes (and threads), thus tending to keep them on the same CPU

– not allowed to run on the current CPU (as indicated by the *cpus_allowed* bitmask in the *task_struct*)

Adding a new Scheduler Class to Linux

- The Scheduler is modular and extensible
- Each scheduler class has priority within hierarchical scheduling hierarchy
  - Priorities defined in sched.h, e.g. #define SCHED_RR 2
  - Linked list of sched_class sched_class.next reflects priority
- Core functions:
  - kernel/sched.c, include/linux/sched.h
  - Additional classes: kernel/sched_fair.c,sched_rt.c
- Process changes class via
  - sched_setscheduler syscall
- Each class needs
  - New sched_class structure implementing scheduling functions
  - New sched_entity in the task_struct

OS Schedulers

Operating System	Preemption	Algorithm
Amiga OS	Yes	Prioritized round-robin scheduling
FreeBSD	Yes	Multilevel feedback queue
Linux kernel before 2.6.0	Yes	Multilevel feedback queue
Linux kernel 2.6.0–2.6.23	Yes	O(1) scheduler
Linux kernel after 2.6.23	Yes	Completely Fair Scheduler
classic Mac OS pre-9	None	Cooperative scheduler
Mac OS 9	Some	Preemptive scheduler for MP tasks, and cooperative for processes and threads
macOS	Yes	Multilevel feedback queue
NetBSD	Yes	Multilevel feedback queue
Solaris	Yes	Multilevel feedback queue
Windows 3.1x	None	Cooperative scheduler
Windows 95, 98, Me	Half	Preemptive scheduler for 32-bit processes, and cooperative for 16-bit processes
Windows NT (including 2000, XP, Vista, 7, and Server)	Yes	Multilevel feedback queue

A fun read

- The Linux Scheduler: A Decade of Wasted Cores
  - https://people.ece.ubc.ca/sasha/papers/eurosys16-final29.pdf

- Talks about performance bugs in multi-core version of Completely Fair Scheduler and how they fixed them

Reading

- Read Chapter 5.6 Linux Scheduling
- Read Chapter 5
- Read Chapter 4 (Linux Kernel Development)

- Acknowledgments
  - Original slides are by **Didem Unat** which were adapted from
    - Öznur Özkasap (Koç University)
    - Operating System and Concepts (9th edition) Wiley
  - Linux Scheduling
    - Linux Overview. COMS W4118 Spring 2008 slideserve.com
    - Prof. Kaustubh R. Joshi from Columbia
  - http://www.algorithmsandme.com/2014/03/scheduling-o1-and-completely-fair.html#.VPgpbMZLOWc