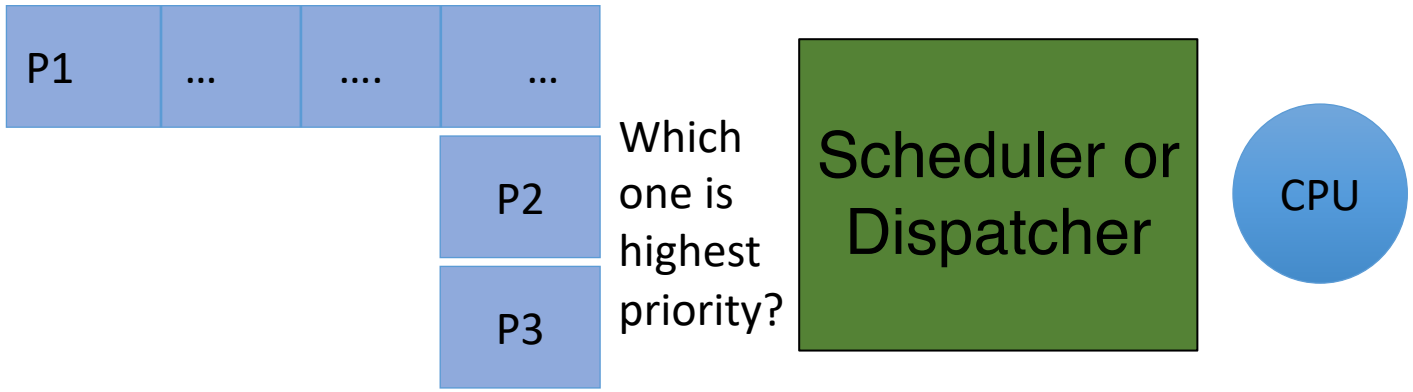
# Lecture 15
# More Scheduling Policies

# Priority-based Scheduling

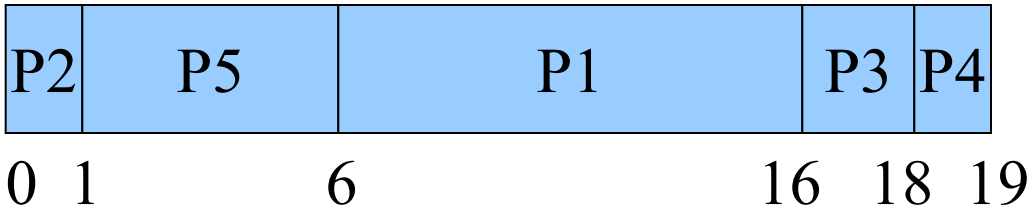
- **Assign each task a priority, and schedule higher priority tasks first, before lower priority tasks**

- **Any criteria can be used to decide on a priority**
  - measurable characteristics of the task
  - external criteria based on the “importance” of the task
  - example: foreground processes may get high priority, while background processes get low priority



# Priority-based Scheduling

| Process | CPU Execution Time | Priority |
|---|---|---|
| P1 | 10 | 3 |
| P2 | 1 | 1 |
| P3 | 2 | 4 |
| P4 | 1 | 5 |
| P5 | 5 | 2 |



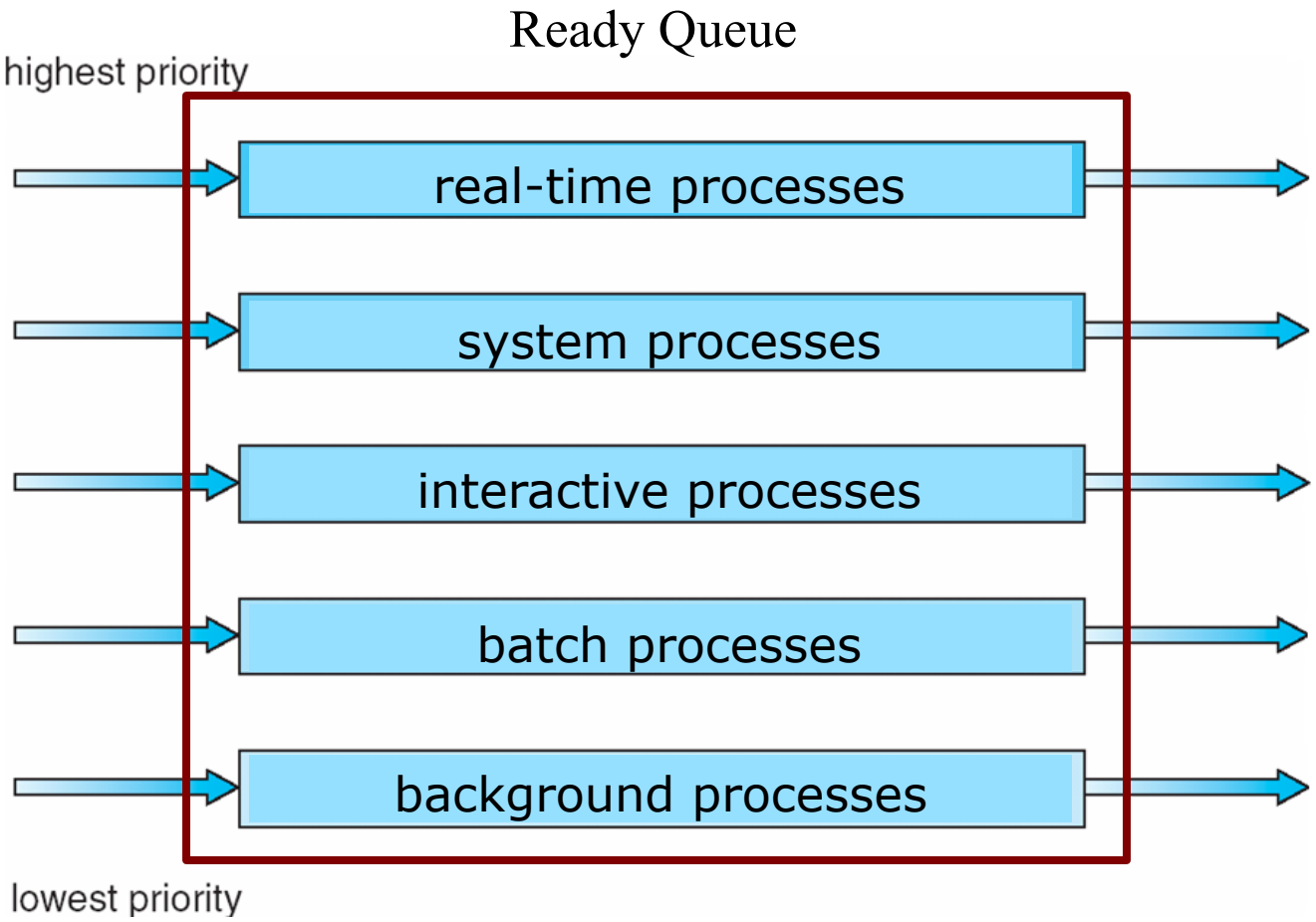
# Priority-based Scheduling

- Can be preemptive:
  - A higher priority process arriving in the ready queue can preempt a lower priority running process
  - Switch can occur if the lower priority process:
    - Yields CPU with a system call
    - Is interrupted by a timer interrupt
    - Is interrupted by a hardware interrupt
  - Each of these cases gives control back to the OS, which can then schedule the higher priority process

# Priority-based Scheduling

- **Multiple tasks with the same priority are scheduled according to some policy**
  - FCFS, round robin, etc.
- **Each priority level has a set of tasks, forming a *multi-level queue***
  - Each level's queue can have its own scheduling policy
- **We use priority-based scheduling and multi-level queue scheduling interchangeably**

# Multilevel Queue Scheduling

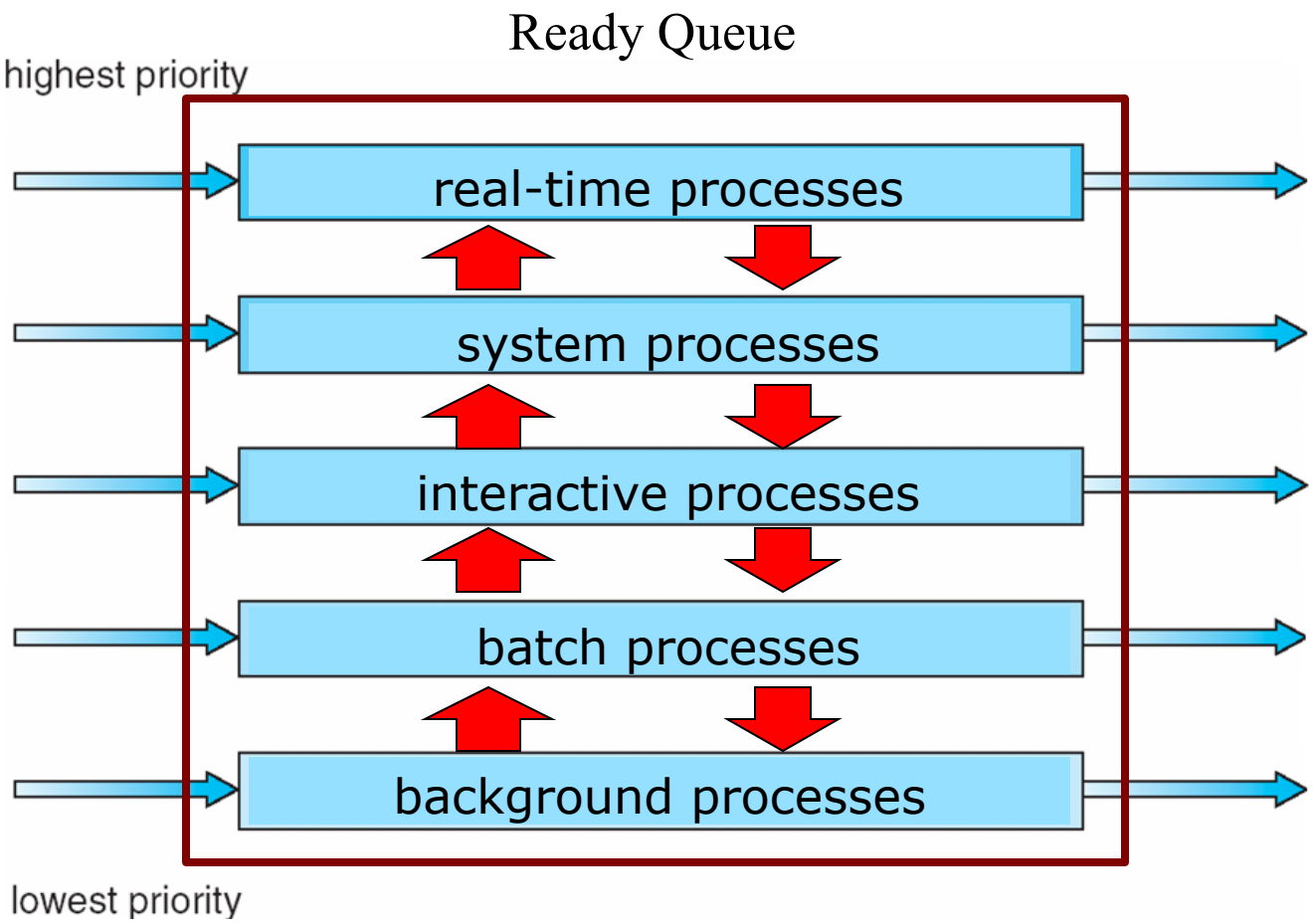


# Priority-based Scheduling

- **Preemptive priorities can starve low priority processes**
  - A higher priority task always gets served ahead of a lower priority task, which never sees the CPU

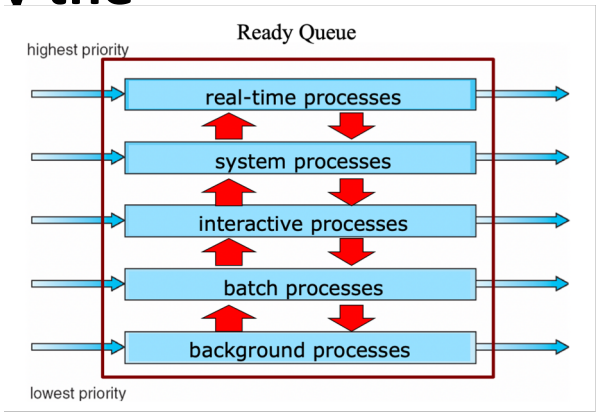
- **Some starvation-free solutions**:
  - Assign each priority level a proportion of time, with higher proportions for higher priorities, and rotate among the levels
    - Similar to weighted round robin, except across levels
  - Create a *multi-level feedback queue* that allows a task to move up/down in priority
    - Avoids starvation of low priority tasks

# Multilevel Feedback Queue Scheduling



# Multilevel Feedback Queue

- **Multilevel-feedback-queue scheduler defined by the following parameters:**
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service



# Multi-level Feedback Queues

- **Criteria for process movement among priority queues could depend upon age of a process:**
  - old processes move to higher priority queues, or conversely, high priority processes are eventually demoted
  - sample aging policy: if priorities range from 1-128, can decrease (increment) the priority by 1 every T seconds
  - eventually, the low priority process will get scheduled on the CPU

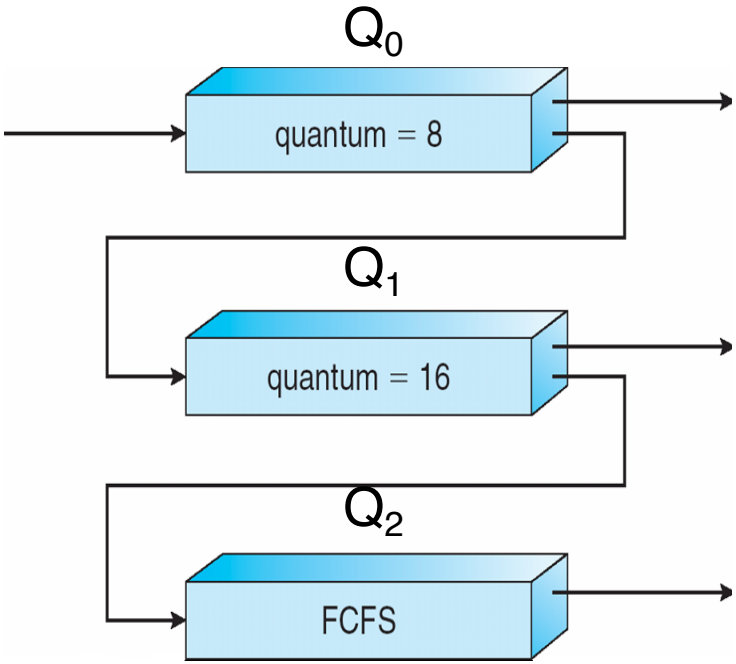
# Multi-level Feedback Queues

- **Criteria for process movement among priority queues could depend upon behavior of a process:**
  - CPU-bound processes move down the hierarchy of queues, allowing interactive and I/O-bound processes to move up
  - give a time slice to each queue, with smaller time slices higher up
  - if a process doesn't finish by its time slice, it is moved down to the next/lower queue
  - over time, a process gravitates towards the time slice that typically describes its average local CPU burst

# Example of Multilevel Feedback Queue

- **Three queues:**
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – FCFS

- **Scheduling**
  - A new job enters queue $Q_0$, job receives 8 ms
  - If it does not finish in 8 ms, job is preempted and moved to $Q_1$
  - At $Q_1$ job receives additional 16 ms.
  - If it still does not complete, it is preempted and moved to $Q_2$
  - Interactive processes are more likely to finish early, processing only a small amount of data
  - Compute-bound processes will exhaust their time slice




Interactive processes will gravitate towards higher priority queues, while Compute bound will move to lower priority queues

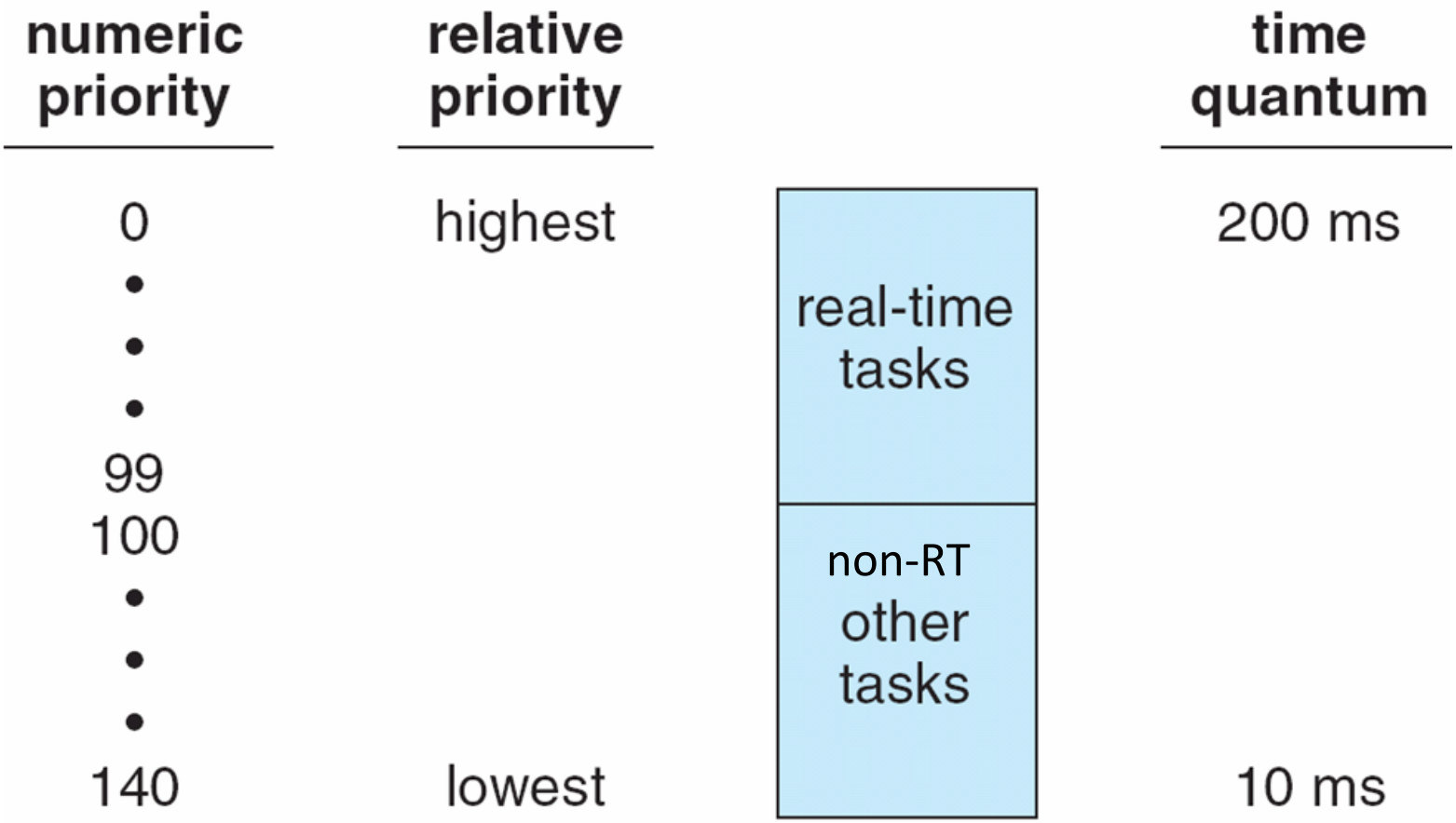
# Priority-based Scheduling

- **In Unix/Linux, you can *nice* a process to set its priority, within limits**
  - e.g. priorities can range from -20 to +20, with lower values giving higher priority, a process with ‘nice +15’ is “nicer” to other processes by incrementing its value (which lowers its priority)
    - E.g. if you want to run a compute-intensive process compute.exe with low priority, you might type at the command line “nice –n 19 compute.exe”
  - To lower the niceness, hence increase priority, you typically have to be root
  - Different schedulers will interpret/use the nice value in their own ways

# Multi-level Feedback Queues

- **In Windows XP and Linux, system & real-time tasks are grouped in a priority range higher than the priority range for non-real-time tasks**
  - XP has 32 priorities
    - 1-15 are for normal processes, 16-31 are for real-time processes.
    - One queue for each priority.
    - XP scheduler traverses queues from high priority to low priority until it finds a process to run
  - Linux has
    - priorities 0-99 are for important/real-time processes
    - 100-139 are for 'nice' user processes.
    - Lower values mean higher priorities.
    - Also, longer time quanta for higher priority tasks
      - 200 ms for highest priority
      - Only 10 ms for lowest priority

# Linux Priorities and Timeslice length



# Multi-level Feedback Queues

- **Most modern OSs use or have used multi-level feedback queues for priority-based preemptive scheduling**
  - e.g. Windows NT/XP, Mac OS X, FreeBSD/NetBSD and Linux pre-2.6
  - Linux 1.2 used a simple round robin scheduler
  - Linux 2.2 introduced scheduling classes (priorities) for real-time and non-real-time processes and SMP (symmetric multi-processing) support

# More Linux Scheduler History

- **Linux 2.4 introduced an O(N) scheduler – help interactive processes**
  - If an interactive process yields its time slice before it's done, then its "goodness" is rewarded with a higher priority next time it executes
  - Keep a list of goodness of all tasks.
  - But this was unordered. So had to search over entire list of N tasks to find the "best" next task to schedule – hence O(N)
    - doesn't scale well

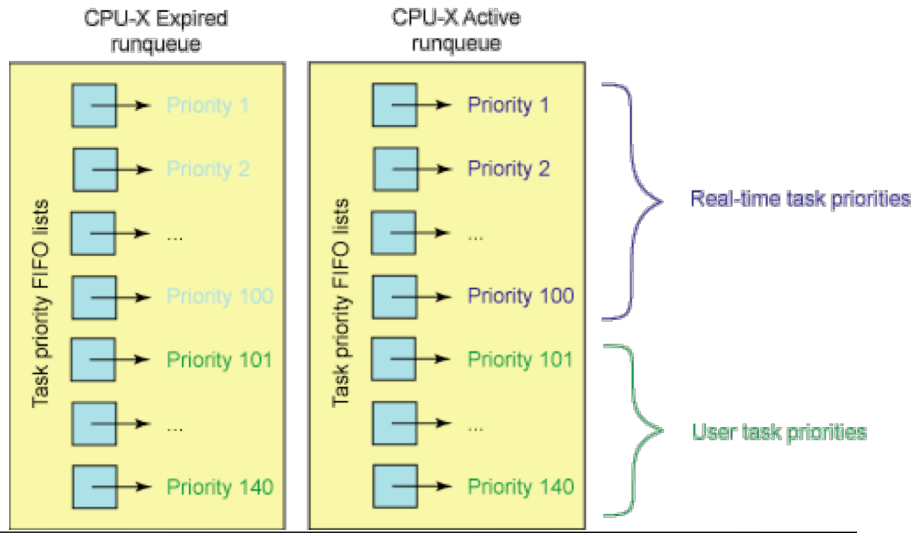
# More Linux Scheduler History

- **Linux 2.6-2.6.23 uses an O(1) scheduler**
  - Iterate over fixed # of 140 priorities to find the highest priority task
  - The amount of search time is bounded by the # priorities, not the # of tasks.
  - Hence O(1) is often called “constant time”
  - scales well because larger # tasks doesn’t affect time to find best next task to schedule

# O(1) Scheduler in Linux

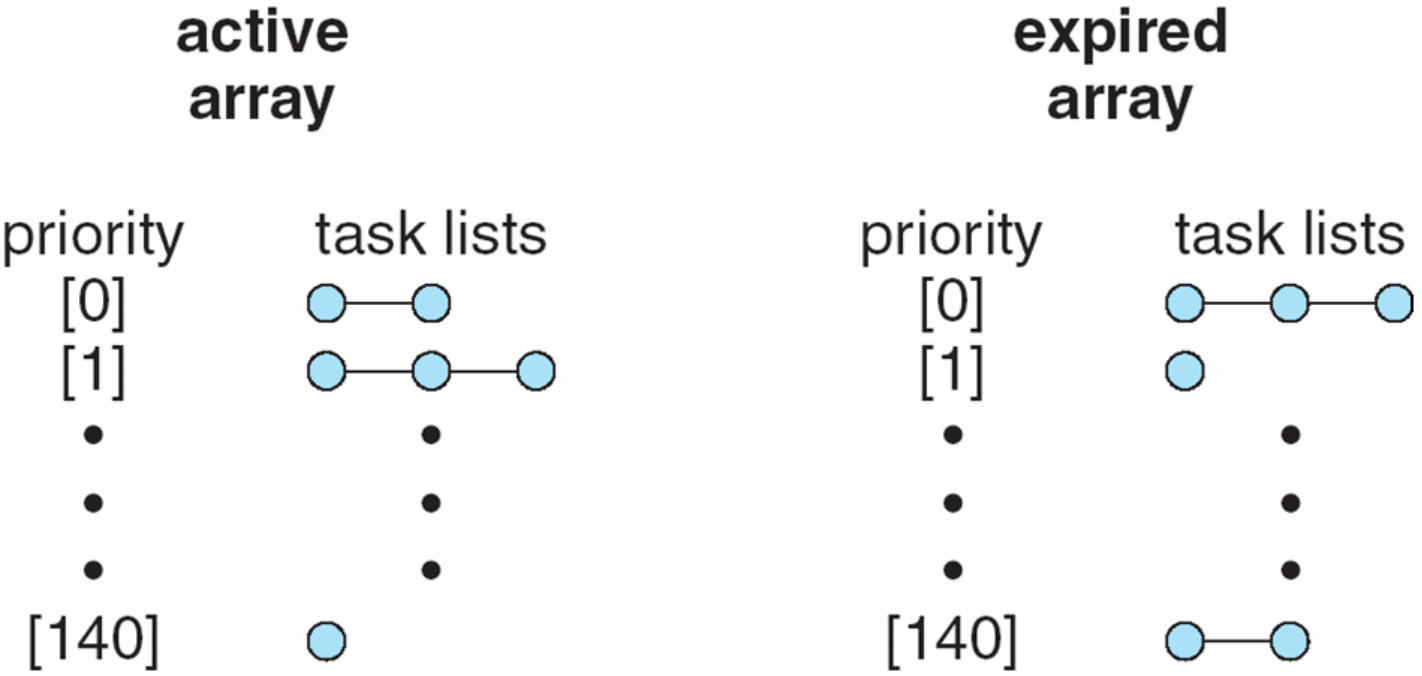
- **Linux maintains two queues:**
  - an active array or run queue and an expired array/queue, each indexed by 140 priorities
- **Active array contains all tasks with time remaining in their time slices, and expired array contains all expired tasks**

- Once a task has exhausted its time slice, it is moved to the expired queue




From http://www.ibm.com/developerworks/linux/library/l-scheduler/

# O(1) Scheduler in Linux

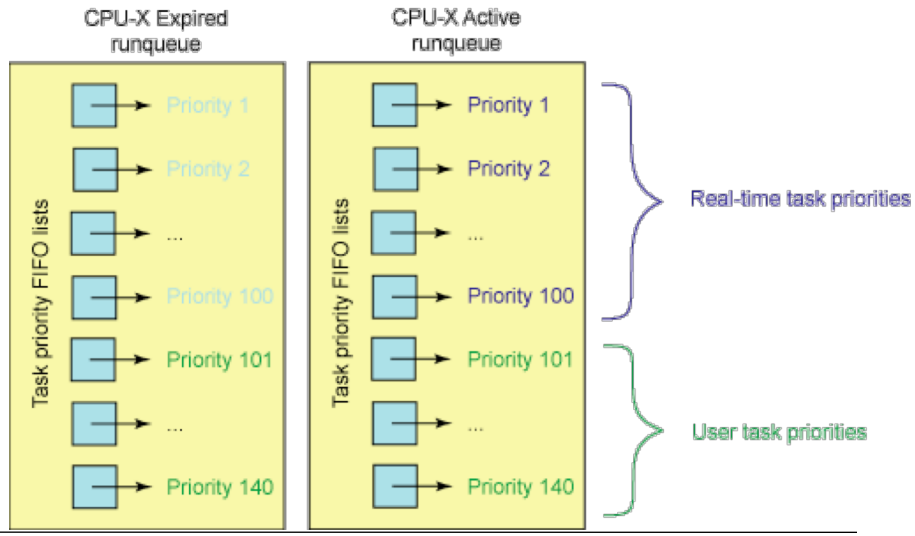


# O(1) Scheduler in Linux

- **An expired task is not eligible for execution again until all other tasks have exhausted their time slice**

- **Scheduler chooses task with highest priority from active array**

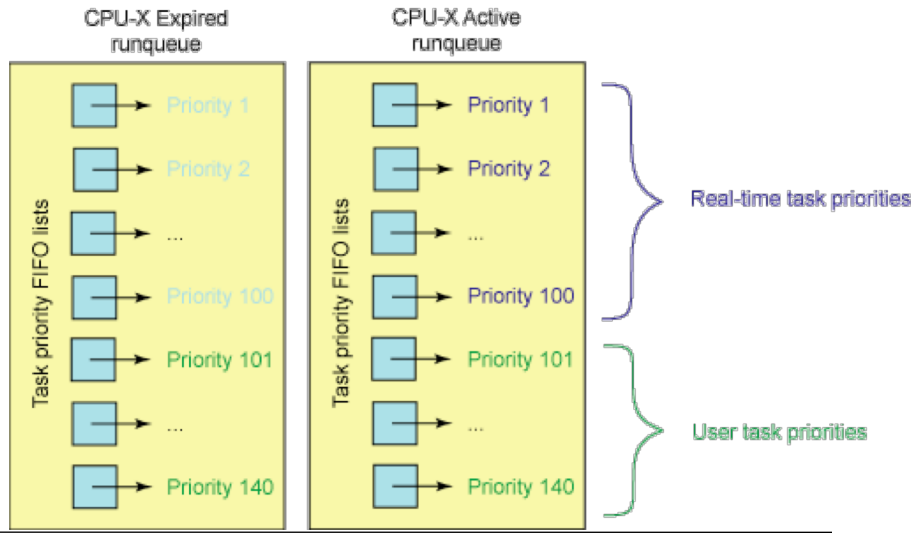
  - Just search linearly through the active array from priority 1 until you find the first priority whose queue contains at least one unexpired task



# O(1) Scheduler in Linux

- **# of steps to find the highest priority task is in the worst case 140**
  - This search is bounded and depends only on the # priorities, not # of tasks, unlike the O(N) scheduler
  - hence this is O(1) in complexity

- When all tasks have exhausted their time slices, the two priority arrays are exchanged
  - the expired array becomes the active array



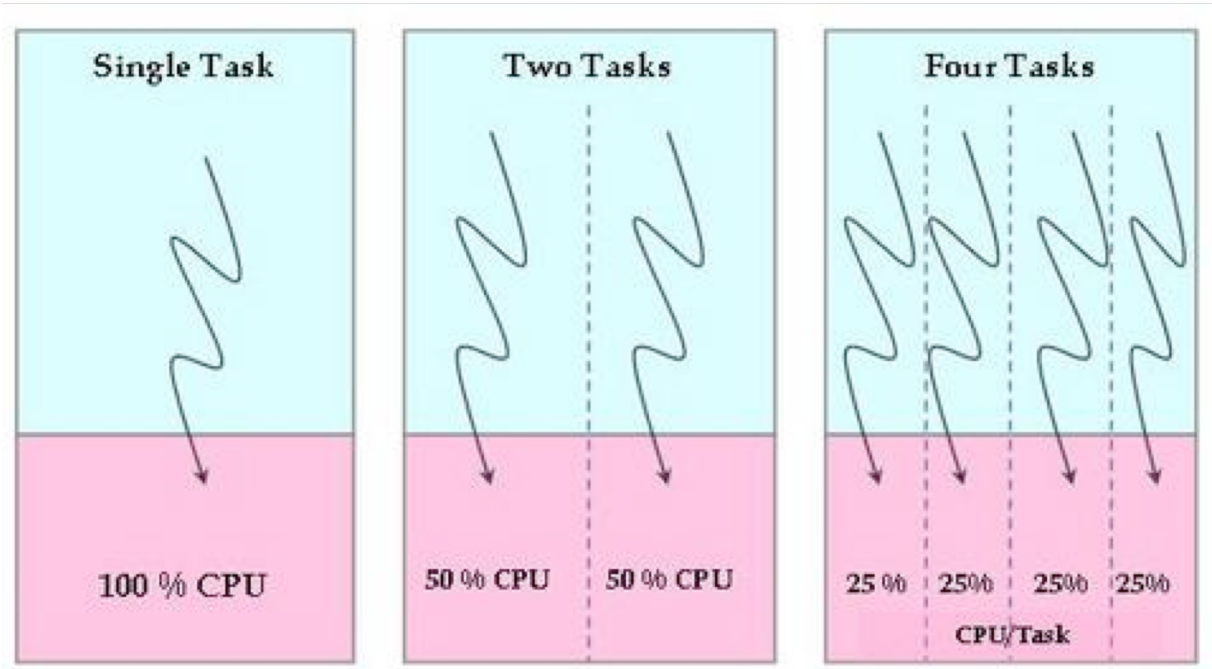
# O(1) Scheduler in Linux

- **When a task is moved from run to expired, Linux recalculates its priority according to a heuristic**
  - New priority = nice value +/- f(interactivity)
    - f() can change the priority by at most +/-5, and is closer to -5 if a task has been sleeping while waiting for I/O
    - interactive tasks tend to wait longer times for I/O, and thus their priority is boosted -5, and closer to +5 for compute-bound tasks
  - This dynamic reassignment of priorities affects only the lowest 40 priorities for non-RT/user tasks (corresponds to the nice range of +/- 20)
  - The heuristics became difficult to implement/maintain

# Linux CFS

# Completely Fair Scheduler (CFS) in Linux

- Linux 2.6.23+/3.* has a "completely fair" scheduler
- Based on concept of an "ideal" multitasking CPU
- If there are N tasks, an ideal CPU gives each task 1/N of CPU *at every instant of time*



# CFS Intuition

- On an ideal CPU, N tasks would run truly in parallel, each getting 1/N of CPU and each executing at every instant of time
  - Example: for a 4 GHz processor, if there are 4 tasks, each gets a 1 GHz processor for each instant of time
  - Each such task makes progress at every instant of time
  - This is “fair” sharing of the CPU among each of the tasks

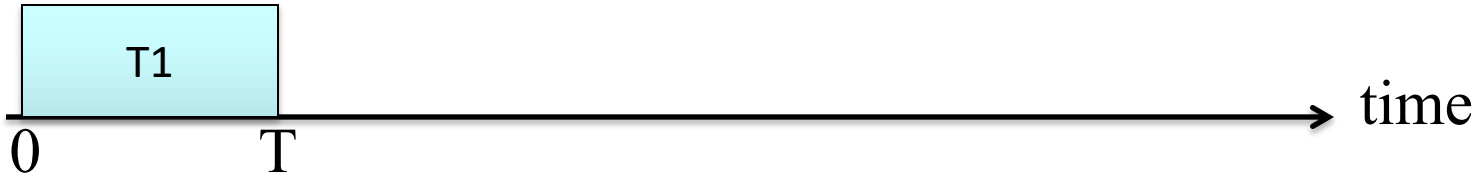
# CFS Intuition

- In practice, we know a real (1-core) CPU cannot run N tasks truly in parallel
  - Only 1 task can run at a time
  - Time slice in/out the N tasks, so that in steady state each task gets ~ 1/N of CPU
  - This gives the illusion of parallelism
  - Thus, what we have is concurrency, i.e. the N tasks run concurrently, but not truly in parallel

# CFS Intuition

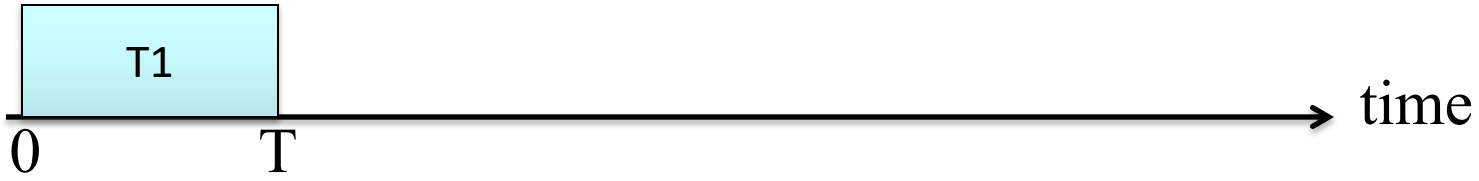
- Ingo Molnar (designer of CFS):
  - “CFS basically models an 'ideal, precise multitasking CPU' on real hardware.”
- So CFS’s goal is to approximate an ideally shared CPU
- Approach: when a task is given T seconds to execute, keep a running balance of the amount of time owed to other tasks as if they all ran on an ideal CPU

# CFS Intuition




- Example:
  - Task T1 is given a T second time slice on the CPU
  - Suppose there are 3 other tasks T2, T3, and T4
  - On an ideal CPU, in any interval of time T, then T1, T2, T3 and T4 would each have had the equivalent of time T/4 on the CPU
  - Instead, on a real CPU
    - T1 is given T instead of T/4, so T1 has been overallocated 3T/4
    - T2, T3 and T4 are owed time T/4 on the CPU, i.e. they have each been forced to *wait* the equivalent of T/4

# CFS Intuition

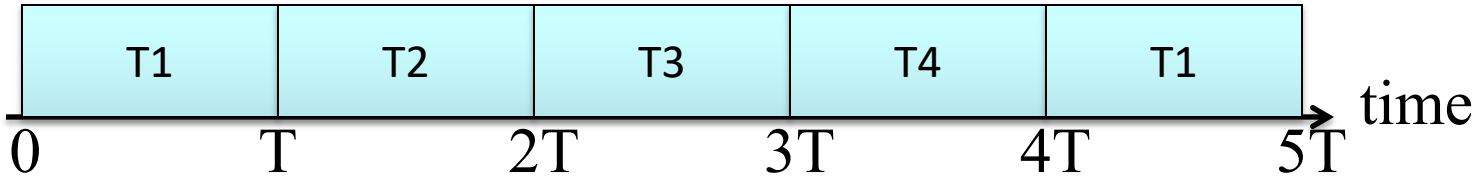



- Example:
  - The current accounting balance is summarized in the table below

| | Time owed to task, i.e. wait time $W_i$: | | | |
|---|---|---|---|---|
| **Giving time T to task:** | T1 | T2 | T3 | T4 |
| T1 | -3T/4 | T/4 | T/4 | T/4 |

  - In general, at any given time t in the system, each task $T_i$ has an amount of time owed to it on an ideal CPU, i.e. the amount of time it was forced to wait, its *wait time* $W_i(t)$,

# CFS Intuition




- Example: let's have round robin over 4 tasks

| | Time owed to task, i.e. wait time $W_i$: | | | |
|---|---|---|---|---|
| **Giving time T to task:** | T1 | T2 | T3 | T4 |
| T1 | -3T/4 | T/4 | T/4 | T/4 |
| then T2 | -T/2 | -T/2 | T/2 | T/2 |
| then T3 | -T/4 | -T/4 | -T/4 | 3T/4 |
| then T4 | 0 | 0 | 0 | 0 |

- After 1 round robin, the balances owed all = 0, so every task receives its fair share of CPU over time 4T

# CFS Intuition

- Suppose a 5th task T5 is added to the round robin
  - Now the amount owed/wait time is calculated as T/5 for each task not chosen for a time slice, and as -4T/5 for the chosen task for a time slice
  - In general, if there are N runnable tasks, then
    - (N-1)T/N is subtracted from the balance owed/wait time of the chosen task
    - T/N is added to the balanced owed/wait time of all other ready-to-run tasks
  - T5 is initially owed no CPU time, so $W_5 = 0$
    - Example: If T5 had arrived just after T2's time slice, then T5's wait time =0 would place it above T1 and T2 but below T3 and T4 in terms of amount of time owed on the CPU

# CFS Scheduler in Linux

- Goal of CFS Scheduler: *select the task with the longest wait time*
  - i.e. choose max $W_i$
    i
  - This is the task that is owed the most time on the CPU and so should be run next to achieve fairness most quickly

# Wait Time Calculation

- Each scheduling decision at time k incurs a wait time $W_i(k)$, either positive or negative, to each task i
- Total accumulated wait time for each task i at time k is:

$$Wtotal_i(k) = \sum_{j=1}^{k} W_i(j)$$

# Wait Time Calculation

- Each wait time $W_i(k)$ =
  - Either a penalty of T/N added to $Wtotal_i$ if task i is not chosen to be scheduled, or
  - (N-1)T/N is *subtracted* from the sum (=T-T/N) if task i is chosen to be scheduled
- So $W_i(k)$= either T/N or –T+T/N
  - note how T/N is added regardless of the case!
- Hence $W_i(k)$ = T/N - execution/run time given to task i at time k, which may be zero
  - Define run time $R_i(k)$ as the execution/run time given to task i at time k, which may be zero
  - So $W_i(k) = T/N - R_i(k)$

# Wait Time Calculation

- In general, each scheduling decision at time k may choose:
  - An arbitrary amount of time T(k) to schedule the chosen task, i.e. it doesn't have to be a fixed time slot T
  - The number of runnable tasks N(k) may change at each decision time k
- So $W_i(k) = T(k)/N(k) - R_i(k)$

# Wait Time Calculation

- Total accumulated wait time for each task i at time k is:

$$Wtotal_i(k) = \sum_{j=1}^{k} W_i(j) = \sum_{j=1}^{k} [T(j)/N(j) + R_i(j)]$$

$$= \sum_{j=1}^{k} T(j)/N(j) + \sum_{j=1}^{k} R_i(j)$$

Global fair clock measuring how system time advances in an ideal CPU with N varying tasks, also called rq->fair_clock in CFS' 1st implementation

Total run time given task i. Let's define it as $Rtotal_i(k)$

# CFS Scheduler in Linux

- Recall: CFS scheduler chooses task with max $Wtotal_i(k)$ at each scheduling decision k
- Maximizing $Wtotal_i(k)$ equivalent to minimizing the quantity [Global fair clock - $Wtotal_i(k)$]
- 1st CFS scheduler:
  - Had to track global fair clock and $Wtotal_i(k)$ for each task i
  - Then would compute the values [Global fair clock - $Wtotal_i(k)$]
  - Then ordered these values in a Red-Black tree
  - Then selected leftmost node in tree (has minimum value) and scheduled the task corresponding to this node

# CFS Scheduler in Linux

- Revised CFS scheduler:
  - We note that [Global fair clock - $Wtotal_i(k)$] = run time $Rtotal_i(k)$ !
  - Minimizing over the quantities [Global fair clock - $Wtotal_i(k)$] is equivalent to minimizing over the accumulated run times $Rtotal_i(k)$
  - 1st CFS scheduler had to track complex values like the global fair clock, and accumulated wait times
    - These both needed the # runnable tasks N(k) at each scheduling decision time k, which keeps changing
  - New approach just sums run times given each task
  - this simple approach still achieves fairness according to our derivation

# Virtual Run Time

- Revised CFS scheduler simply sums the run times given each task and chooses the one to schedule with the minimum sum
  - This is equivalent to choosing the task owed the most time on an ideal fair CPU according to our derivation, and thus achieves fairness
  - Caveat: when a new task is added to the run queue, it may have been blocked a long time, so its run time may be very low compared to other tasks in the run queue
    - Such a task would consume a long time before its accumulated run time rises to a level close to the other executing tasks' total run times, which would effectively block other tasks from running in a timely manner

# Virtual Run Time

- Revised CFS scheduler accommodates new tasks as follows:
  - Define a virtual run time *vruntime*
    - As before, each normally running task i simply adds its given run times to its own accumulated sum $vruntime_i$
  - When a new task is added to the run queue (or an existing task becomes unblocked from I/O), assign it a new virtual run time = minimum of current vruntimes in the run queue
    - This quantity is defined as *min_vruntime*
  - This approach re-normalizes the newly active task's run time to about the level of the virtual run times of the currently runnable tasks

# Virtual Run Time

- Since each newly active task's is given a re-normalized run time, then the run time calculated is not the actual execution time given a task
  - Hence we need to define a new term $vruntime_i(k)$, rather than use the absolute accumulated run time $Rtotal_i(k)$
- *Intuitively, CFS choosing the task with the minimum virtual run time prioritizes the task that been given the least time on the CPU*
  - *This is the task that should get service first to ensure fairness*

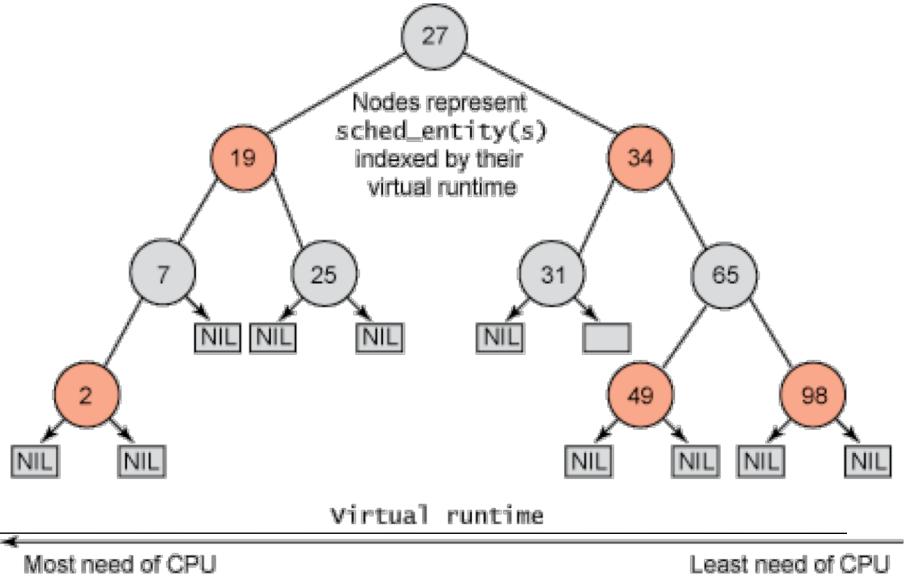
# CFS Scheduler in Linux

- *So revised CFS scheduler chooses the task with the minimum $vruntime_i(k)$ at each scheduling decision time k*
- This approach is responsive to interactive tasks!
  - They get instant service after they unblock from their I/O
  - This is because they are given a re-normalized $vruntime_i(k) = min\_vruntime$,
  - Since CFS chooses the next task to schedule as the one with the minimum vruntime, then the interactive task will be chosen first and get service immediately

# CFS' Red Black Tree

- To quickly find the task with the minimum vruntime, order the vruntimes in a Red-Black tree
  - This is a balanced tree, ordered from left (minimum vruntime) to right (maximum vruntime)

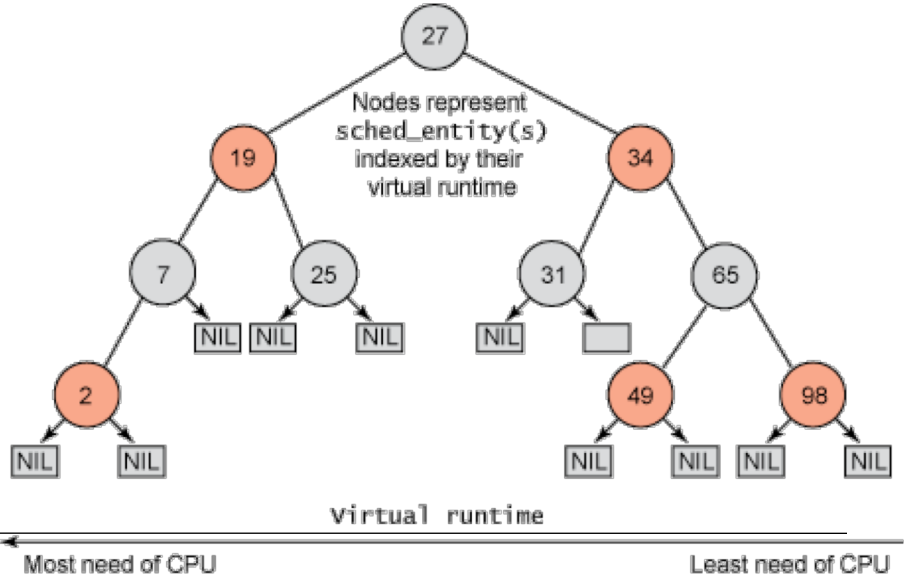
- Finding the minimum is fast, simple and constant time!
  - Choose leftmost task in tree with lowest virtual run time to schedule next



# CFS' Red Black Tree

- As tasks run more, their virtual run time increases
  - so they migrate to other positions further to the right in the tree
  - Must re-insert nodes to tree, and rearrange tree, but the RB tree is self-balancing

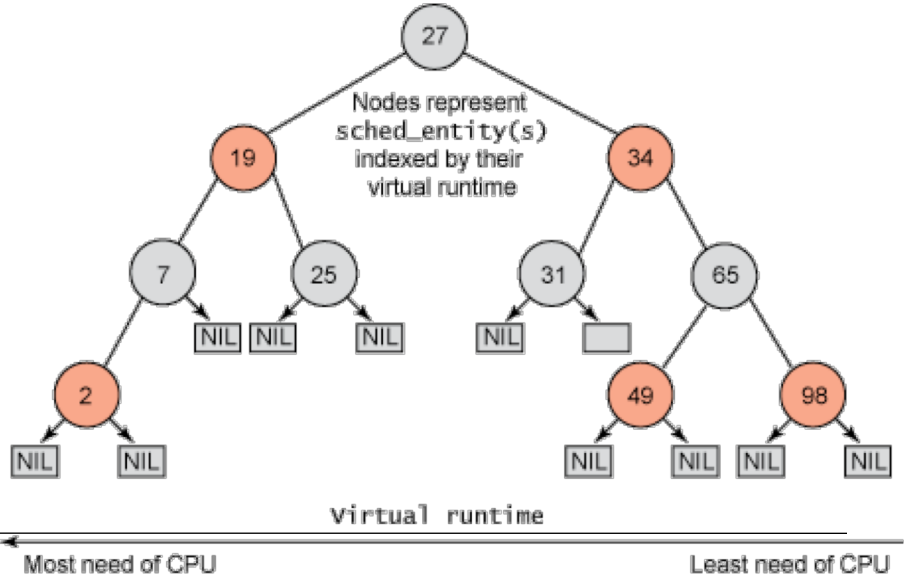
- Inserting nodes is an O(logN) operation due to RB tree
  - This is viewed as acceptable overhead



# CFS' Red Black Tree

- Tasks that haven't had CPU execution in a while will migrate left and eventually get service
  - Intuitively, this eventual migration leftwards makes CFS fair

- Newly active tasks, e.g. interactive ones, will be added to the left of the tree and get service quickly



# CFS Scheduler and Priorities

- All non-Real Time tasks of differing priorities are combined into one RB tree
- don’t need 40 separate run queues, one for each priority - elegant!
- Higher priority tasks get larger run time slices
- lower niceness => higher the priority => more run time is given on the CPU

# CFS Scheduler and Priorities

- Higher priority tasks are scheduled more often
  - *virtual runtime += (actual CPU runtime) * NICE_0 / task's weight*
  - Higher priority
    $\Rightarrow$ higher weight
    $\Rightarrow$ less increment of *vruntime*
    $\Rightarrow$ task is further left on the RB tree and is scheduled sooner

# CFS Scheduler and Priorities

- While CFS is fair to tasks, it is not necessarily fair to applications
  - Suppose application A1 has 100 threads T1-T100
  - Suppose application A2 is interactive and has one thread T101
  - CFS would give
    - A1 100/101 of CPU
    - A2 only 1/101 of the CPU
- Instead, Linux CFS supports fairness across groups:
  - A1 is in group 1 and A2 is in group 2
  - Groups 1 and 2 each get 50% of CPU – fair!
  - Within Group 1, 100 threads share 50% of CPU
- Multi-threaded apps don't overwhelm single thread apps

# Realtime and Multi-core Scheduling

# Real Time Scheduling in Linux

- Linux also includes three **real-time** scheduling classes:
  - Real time FIFO – soft real time (SCHED_FIFO)
  - Real time Round Robin – soft real time (SCHED_RR)
  - Real time Earliest Deadline First – hard real time as of Linux 3.14 (SCHED_DEADLINE)

- Only processes with the priorities 0-99 have access to these RT schedulers

# Real Time Scheduling in Linux

- A real time FIFO task continues to run until it voluntarily yields the processor, blocks or is preempted by a higher-priority real-time task
  - no timeslices
  - all other tasks of lower priority will not be scheduled until it relinquishes the CPU
  - two equal-priority Real time FIFO tasks do not preempt each other

# Real Time Scheduling in Linux

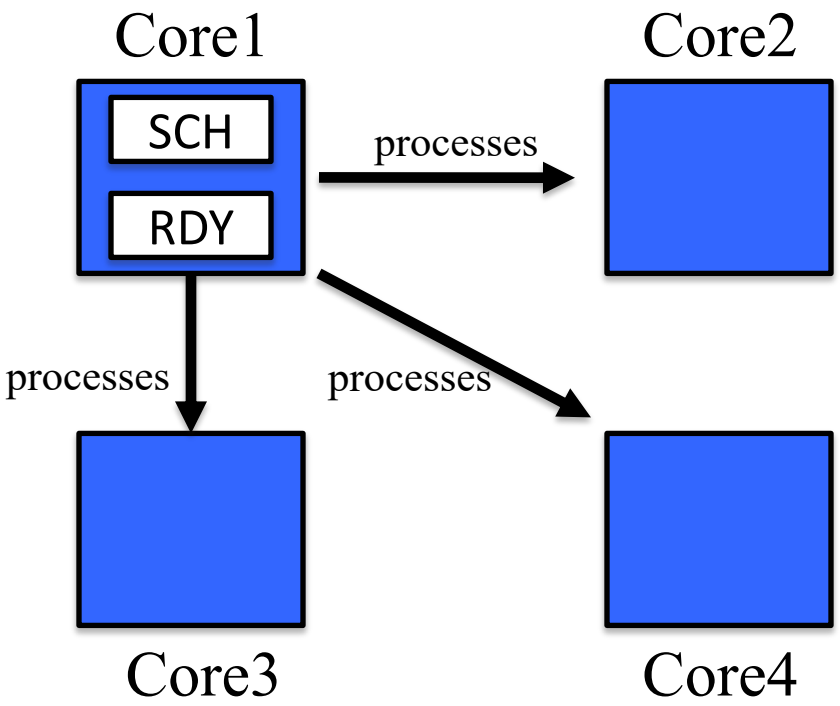
- SCHED_RR is similar to SCHED_FIFO, except that such tasks are allotted timeslices based on their priority and run until they exhaust their timeslice

- Non-real time tasks continue to use CFS algorithm

- SCHED_DEADLINE uses an Earliest Deadline First algorithm to schedule each task.

# Multi-core Scheduling

- Scheduling over multiple processors or cores is a new challenge.
  - A single CPU/processor may support multiple cores




SCH = Scheduler, RDY = Ready Queue

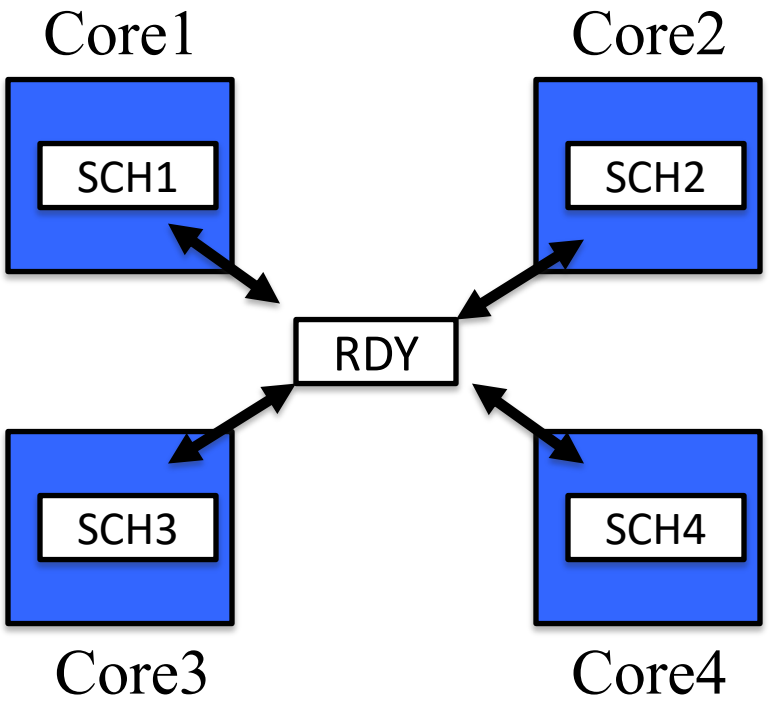
- Variety of multi-core schedulers being tried. We'll just mention some design themes.
- In *asymmetric multiprocessing* - one CPU handles all scheduling, decides which processes run on which cores

# Multi-core Scheduling

- In symmetric multi-processing (SMP), each core is self-scheduling.
  - All modern OSs support some form of SMP

Two types:

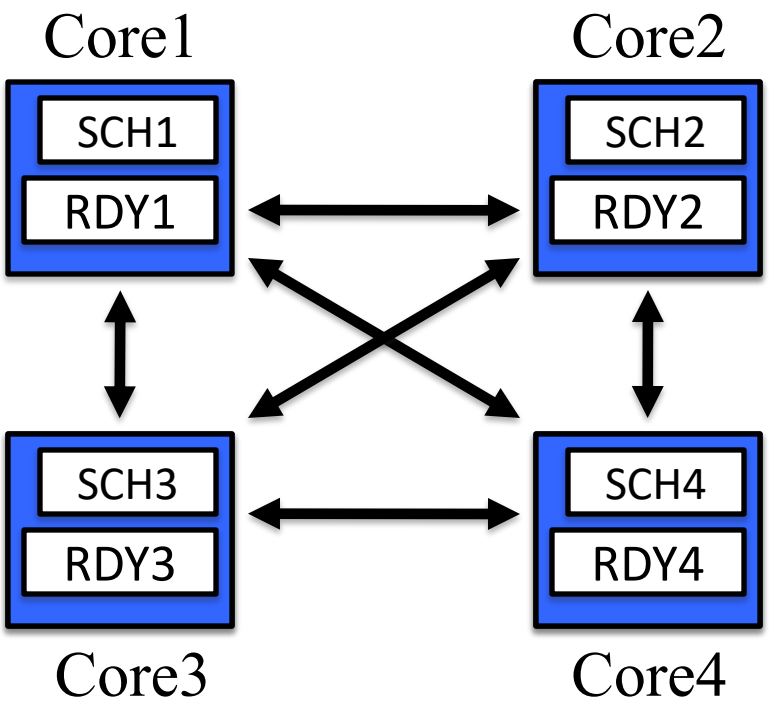
1. All cores share a single global ready queue.




- Here, each core has its own scheduler. When idle, each scheduler requests another process from the shared ready queue. Puts it back when time slice done.
  - Synchronization needed to write/read shared ready queue

# Multi-core Scheduling

2. Another self-scheduling SMP approach is when each core has its own ready queue
   - Most modern OSs support this paradigm




- a typical OS scheduler plus a ready queue designed for a single CPU can run on each core…
- Except that processes now can migrate to other cores/processors
  - There has to be some additional coordination of migration

# Multi-core Scheduling - Cache

- Caching is important to consider
  - Each CPU has its own cache to improve performance
  - If a process migrates too much between CPUs
    - rebuild L1 and L2 caches each time a process starts on a new core/processor
  - L3 caches that span multiple cores can help alleviate this, but there is a performance hit, because L3 is slower than L1 and L2.
  - In any case, L1 and L2 caches still have to be rebuilt.

# Multi-core Scheduling - Affinity

- To maximally exploit caching, processes tend to stick to a given core/processor = processor affinity
  - In hard affinity, a process specifies via a system call that it insists on staying on a given CPU core
  - In soft affinity, there is still a bias to stick to a CPU core, but processes can on occasion migrate.
  - Linux supports both

# Multi-core Scheduling

## Load balancing

- Goal: Keep workload evenly distributed across cores
- Otherwise, some cores will be under-utilized.
- When there is a single shared ready queue, there is automatic load balancing
- cores just pull in processes from the ready queue whenever they're idle.
- push migration – a dedicated task periodically checks the load on each core, and if imbalance, pushes processes from more-loaded to less-loaded cores

# Multi-core Scheduling
# Load Balancing

- Load balancing can conflict with caching
  - Push/pull migration causes caches to be rebuilt
- Load balancing can conflict with power management
  - Mobile devices typically want to save power
  - One approach is to power down unused cores
  - Load balancing would keep as many cores active as possible, thereby consuming power
- In systems, often conflicting design goals