CSCI 3753
Operating Systems

CPU Scheduling (Advanced)

Lecture Notes By

Shivakant Mishra

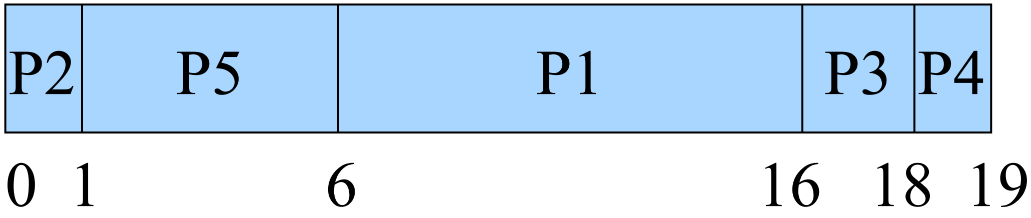
Computer Science, CU-Boulder

Last Update: 03/07/14

Priority Scheduling

- Assign each task a priority, and schedule higher priority tasks first
- Priority can be based on
  - any measurable characteristics of the process, or
  - some external criteria
- Can be preemptive

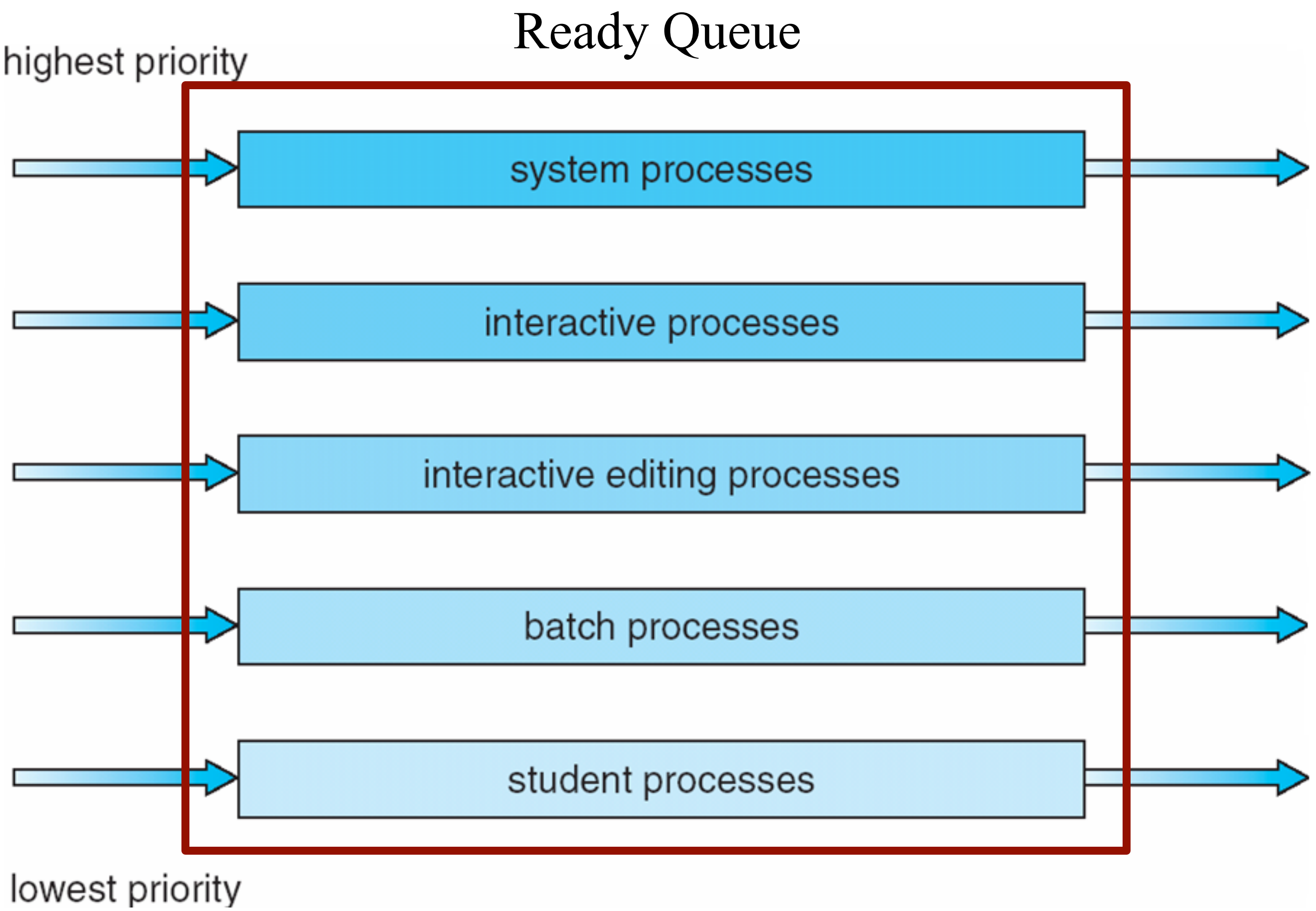
Process	CPU Execution Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



Multi-level Queue Scheduling

- Use priorities to partition the ready queue into several separate queues
  - Different processes have different needs, e.g. foreground and background
  - If there are multiple processes with the same priority queue, need to apply a scheduling policy within that priority level
  - Don’t have to apply the same scheduling policy to every priority level, e.g. foreground gets EDF, background gets RR or FCFS
- Queues can be organized by priority, or each given a percentage of CPU, or a hybrid combination

Multilevel Queue Scheduling



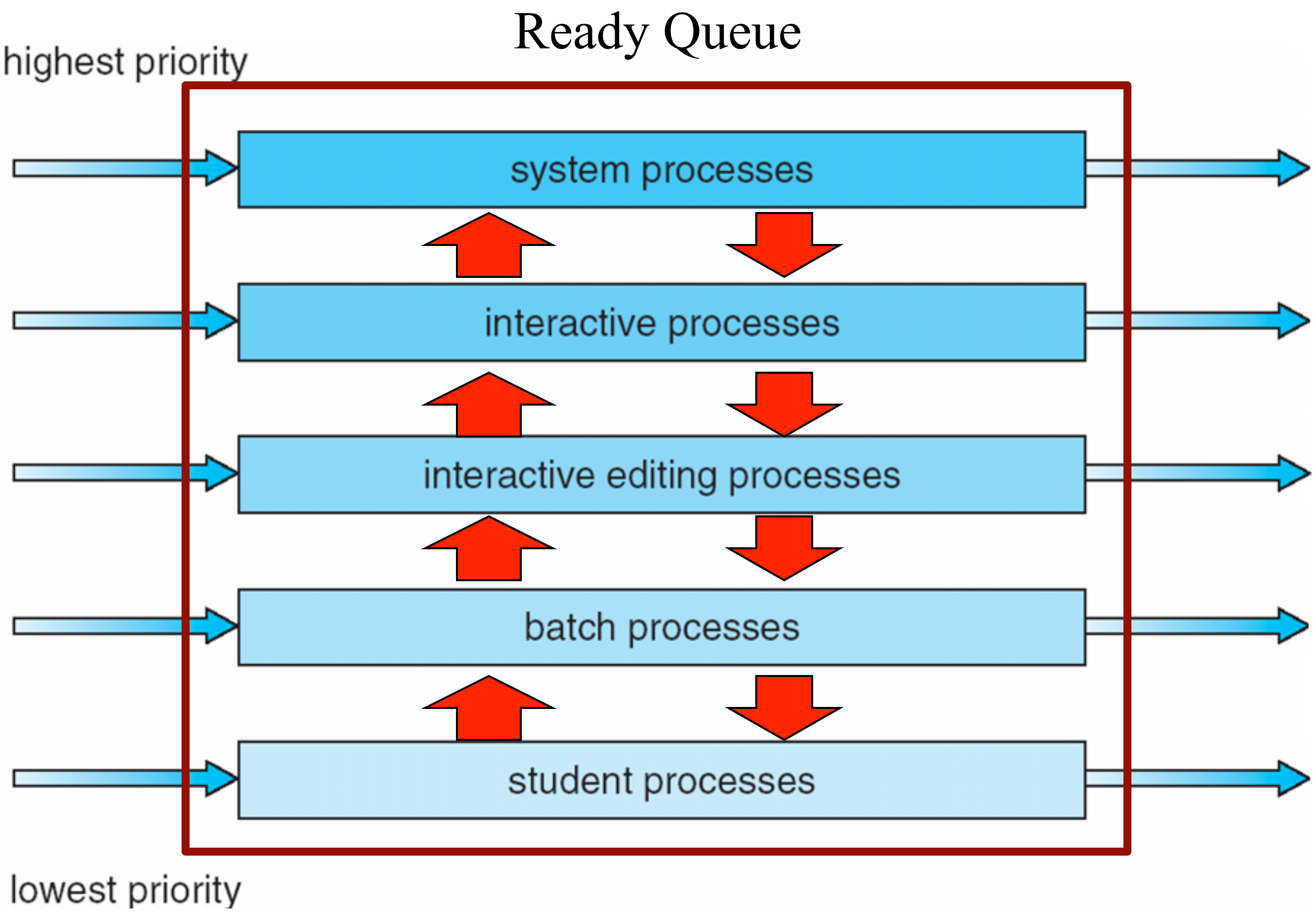
Priority Scheduling

- Preemptive priorities can starve low priority processes
  - A higher priority process always gets served ahead of a lower priority process, which never sees the CPU
- The solution is *multi-level feedback queues* that allow a process to move up/down in priority
  - Avoids starvation

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

Multilevel Feedback Queue Scheduling



Criteria for Process Movement

1. 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

2. Behavior of a process (CPU bound vs I/O bound)
    - Give higher priority to I/O bound processes: allows higher parallelism between CPU and I/O
    - A process typically alternates between bursts of I/O activity and CPU activity
    - Move a process down the hierarchy of queues during CPU burst, 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 uses its time slice completely (CPU burst), it is moved down to the next lowest queue
    - Over time, a process gravitates towards the time slice that typically describes its average local CPU burst

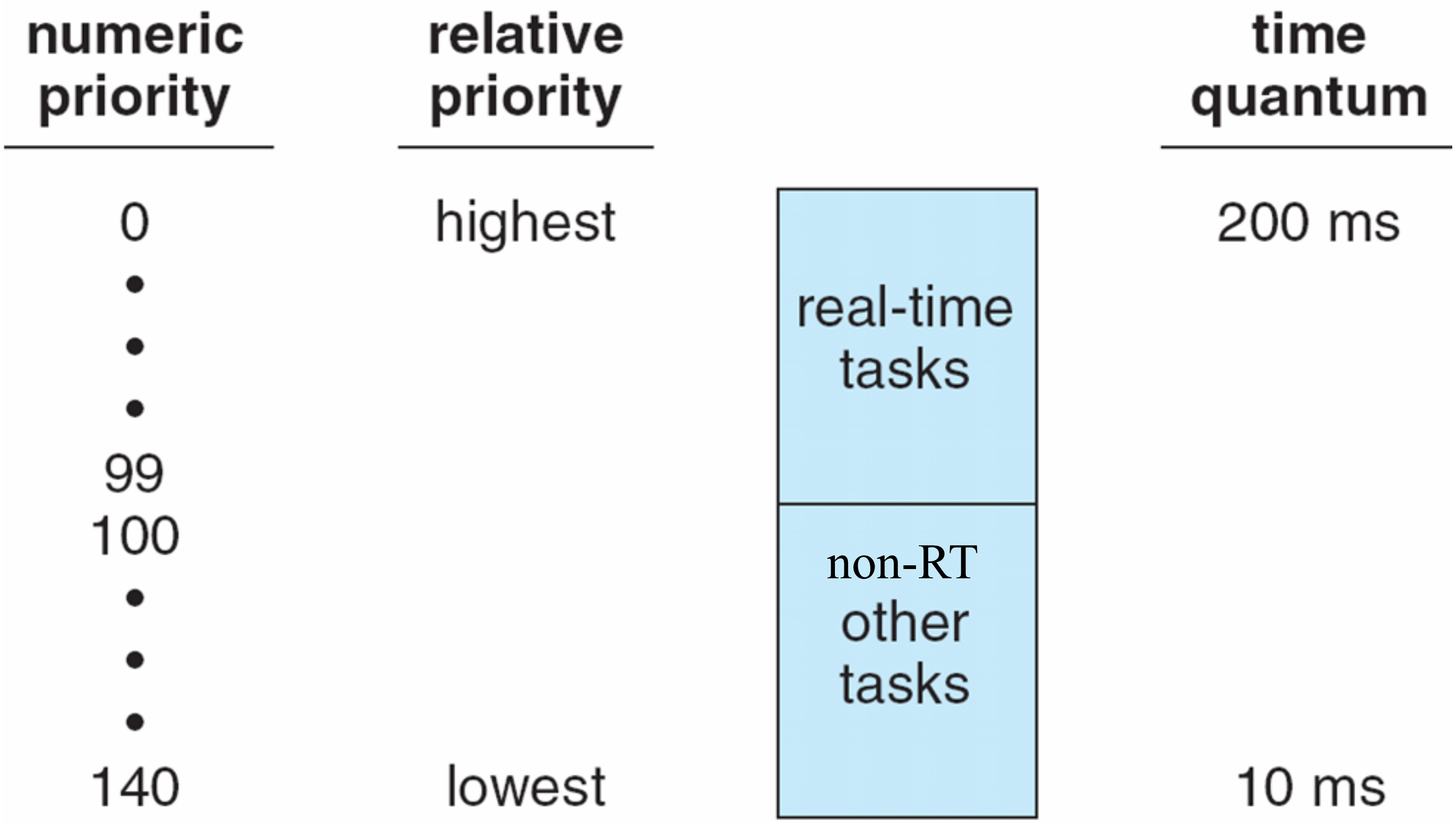
Priority Scheduling

- 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 processes are grouped in a range of priorities that are higher in priority than the range of priorities given to non-real-time processes
  - 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
  - In Linux, priorities 0-99 are for important/real-time processes while 100-139 are for ‘nice’ user processes. Lower values mean higher priorities.
    - Also, longer time quanta for higher priority tasks (200 ms for highest) and shorter time quanta for lower priority tasks (10 ms for lowest).

Linux Priorities and Time-slice length



numeric priority	relative priority		time quantum
0	highest	real-time tasks	200 ms
•			
•			
•			
99			
100		non-RT other tasks	
•			
•			
•			
140	lowest		10 ms

Multi-level Feedback Queues

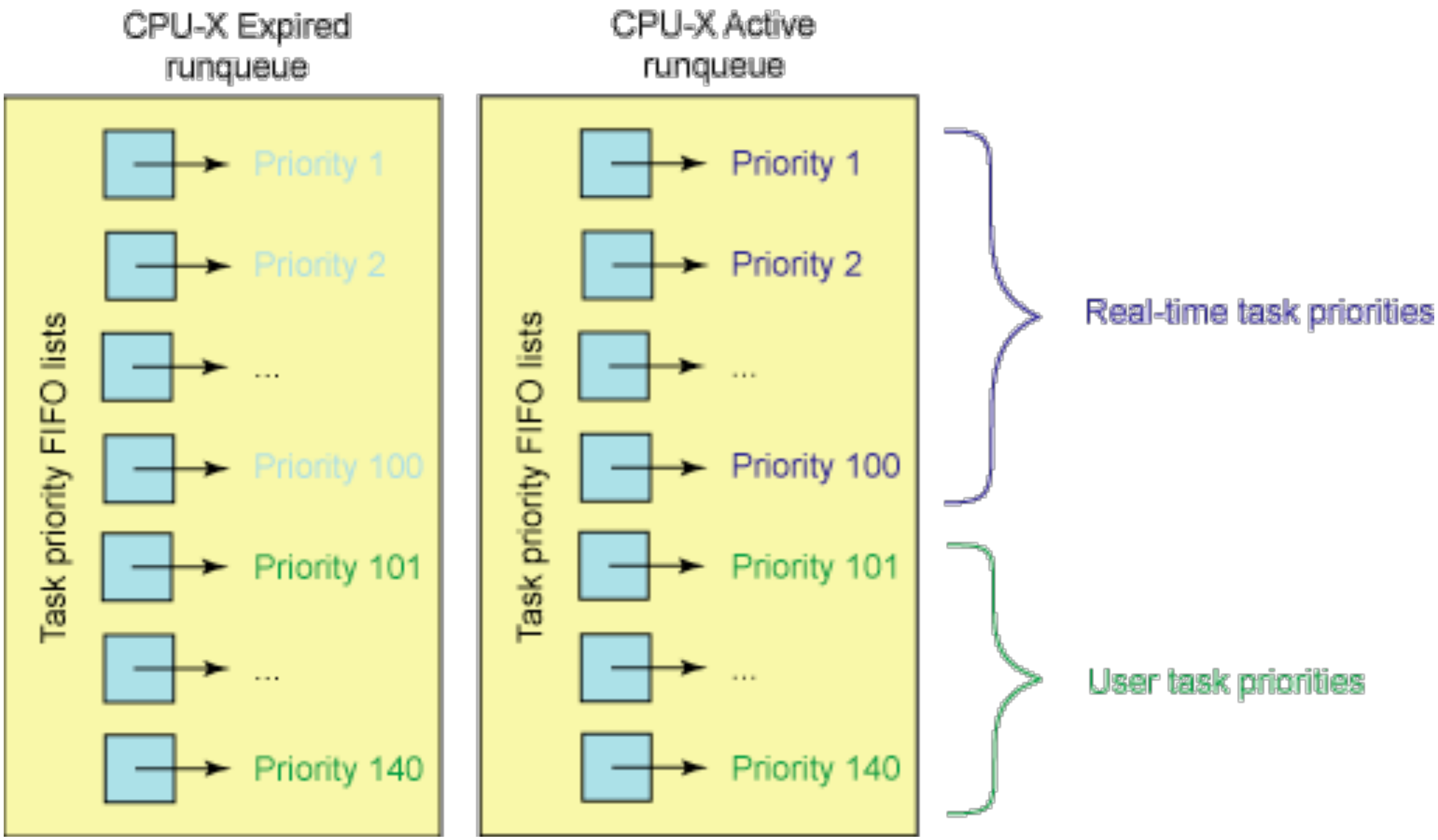
- Most modern OSs use or have used multi-level feedback queues for priority-based preemptive scheduling
  - Windows NT/XP, Mac OS X, FreeBSD/NetBSD and Linux pre-2.6
    - Linux 1.2 used a simple round robin scheduler
    - Linux 2.2 introduced scheduling classes (priorities) for real-time and non-real-time processes and SMP support
    - Linux 2.4 introduced an O(N) scheduler – help interactive processes
  - But Linux 2.6-2.6.23 uses an O(1) scheduler
    - Iterate over fixed # of 140 priorities to find the highest priority task – scales well because larger # processes doesn’t affect time to find best next task to schedule
  - And Linux 2.6.23+ uses a “Completely Fair Scheduler”

O(N) Scheduler

- 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
- SMP: A process could be scheduled on a different processor
  - Loses cache affinity
- SMP: Single runqueue lock
  - The act of choosing a task to execute locked out any other processors from manipulating the runqueues
- Preemption not possible

O(1) Scheduler in Linux

- Linux maintains two queues: an active runqueue and an expired runqueue, each indexed by 140 priorities
- Active runqueue contains all tasks with time remaining in their time slices, and expired runqueue contains all expired tasks.
- Once a task has exhausted its time slice, it is moved to the expired runqueue and is not eligible for execution again until all other tasks have exhausted their time slice

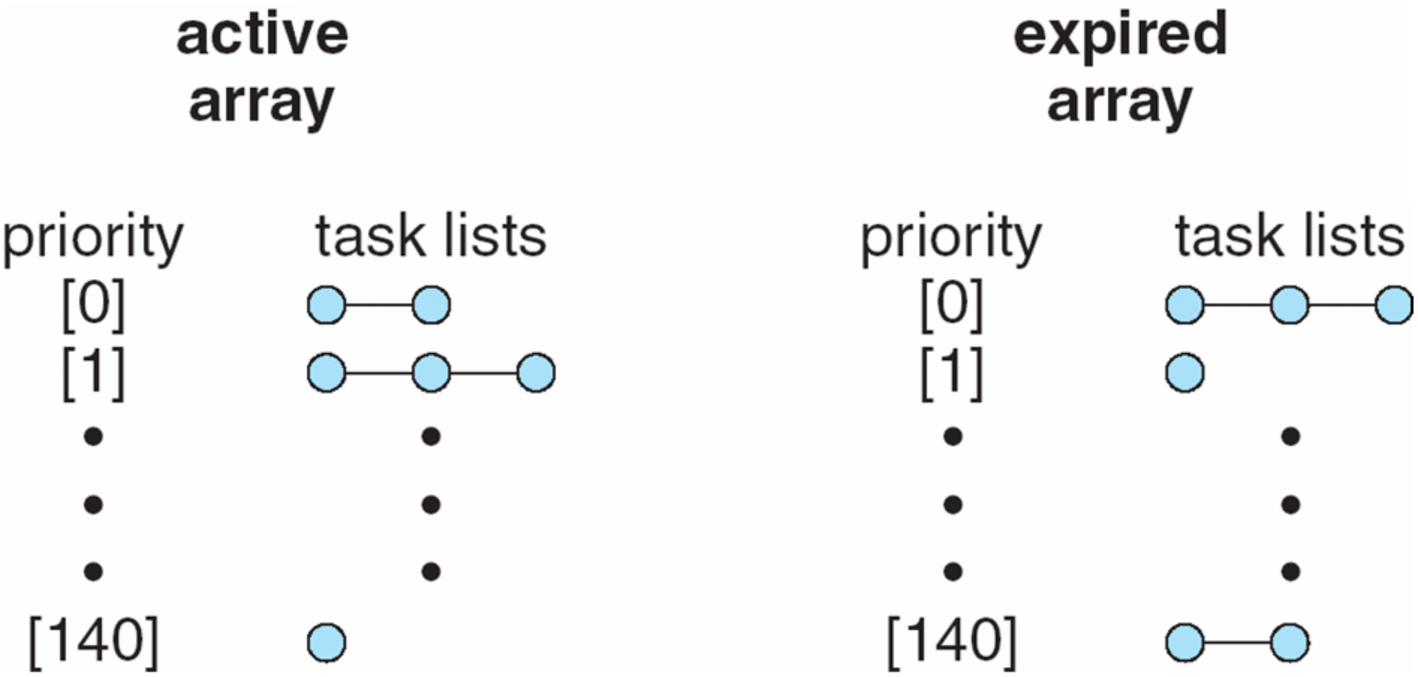


From http://www.ibm.com/developerworks/linux/library/l-scheduler/

O(1) Scheduler

- Scheduler chooses task with highest priority from active runqueue
  - Just search linearly up the active runqueue from priority 1 until you find the first priority whose queue contains at least one unexpired task
  - # of steps to find the highest priority task is in the worst case 140, so it's bounded and depends only on the # priorities, not # of tasks,
  - Hence this is O(1) in complexity

O(1) Scheduler in Linux



O(1) Scheduler in Linux

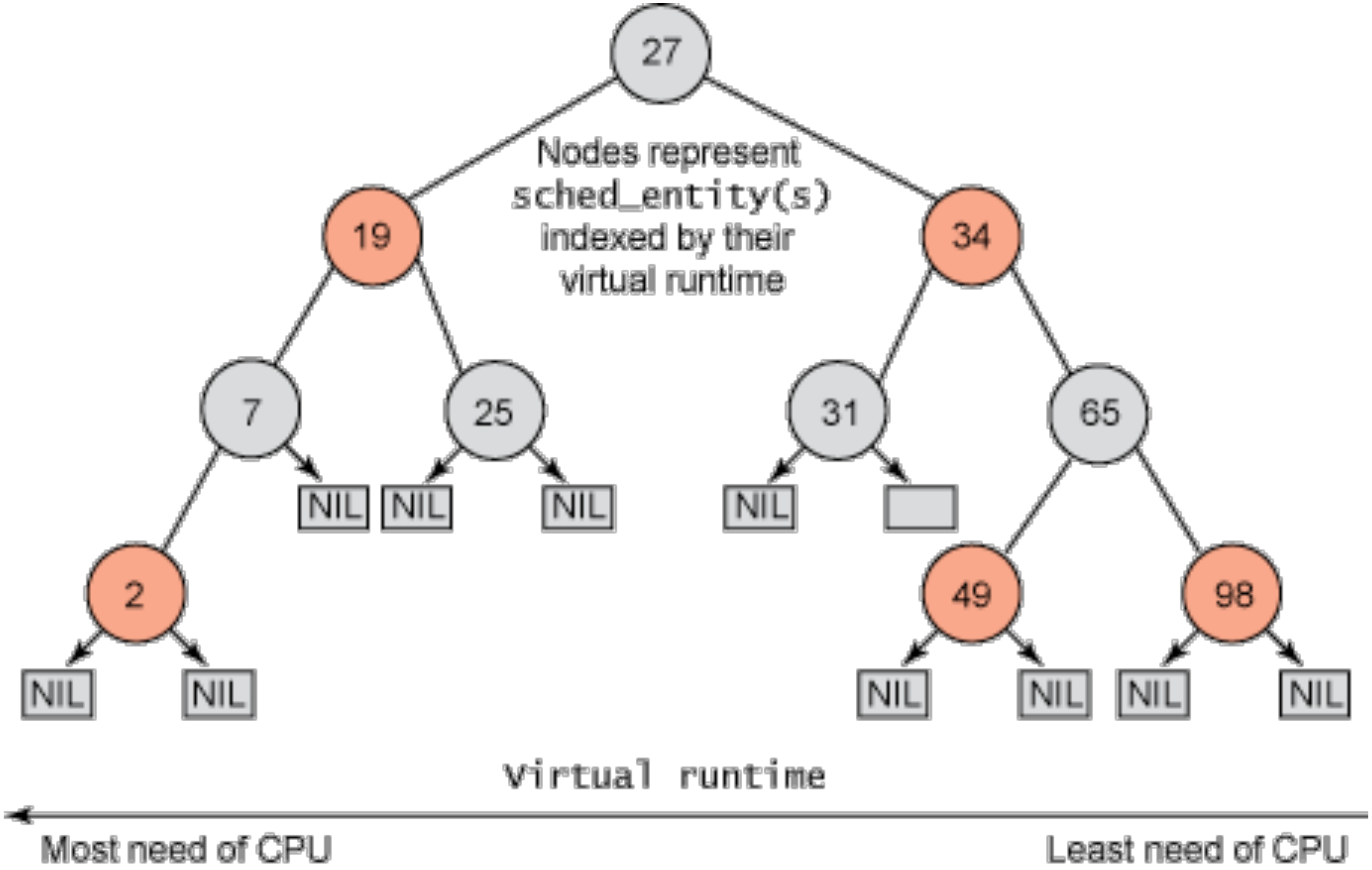
- When all tasks have exhausted their time slices, the two priority arrays are exchanged, and the expired array becomes the active array
- When a task is moved from run to expired, Linux recalculates its priority according to a heuristic
  - New priority = nice value +/- f(interactivity), where f() can 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 O(1) Scheduler was the default scheduler in Linux

Completely Fair Scheduler (CFS)

- Heuristics of O(1) for dynamic reassignment of priorities were complicated and somewhat arbitrary
- Linux 2.6.23+ has a "completely fair" scheduler
- Virtual run time (vruntime): The amount of time a process has used the CPU so far
  - The smaller the virtual run time, the more need for the processor. This is fair.
  - Decay factor: CFS doesn't use priorities directly but instead uses them as a decay factor for the time a task is permitted to execute
    - low priority → high decay factor → the time a task is permitted to execute dissipates more quickly for a lower-priority task than for a higher-priority task
    - elegant solution to avoid maintaining run queues per priority

CFS

- Use (time-ordered) red-black tree instead of queue
  - Self balancing B-Tree: No path in the tree will ever be more than twice as long as any other
  - Insert/delete/search occur in O(log *n*) time
- Lower vruntime processes are on the left side of the tree
- As tasks run more, their virtual run time increases, and they migrate to other positions further to the right in the tree



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

Real Time Scheduling in Linux

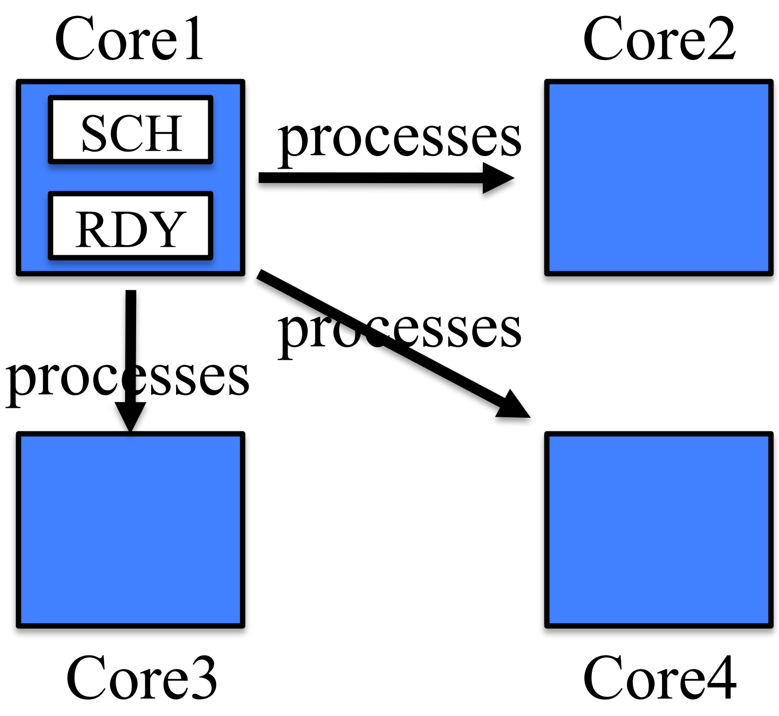
- Linux also includes two real-time scheduling classes:
  - Real time Round Robin
  - Real time FIFO
  - These are soft real time scheduling algorithms, not hard real time scheduling algorithms with absolute deadlines
- Only processes with the priorities 0-99
- "When a Real time FIFO task starts running, it continues to run until it voluntarily yields the processor, blocks or is preempted by a higher-priority real-time task. 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."

http://www.linuxjournal.com/magazine/real-time-linux-kernel-scheduler

Multi-Core Scheduling

- Multicore processors
  - Multiple processor cores on the same physical chip
  - Each core maintains its architectural state and thus appears to OS as a separate physical processor
- Memory stall: waiting time to get data from memory
  - Cache miss, etc.
- Multithreaded processor cores
  - Two or more hardware threads assigned to a single core
  - UltraSPARC T3: 16 cores per chip, 8 hardware threads per core →appears as 128 logical processors to OS

Multi-core Scheduling

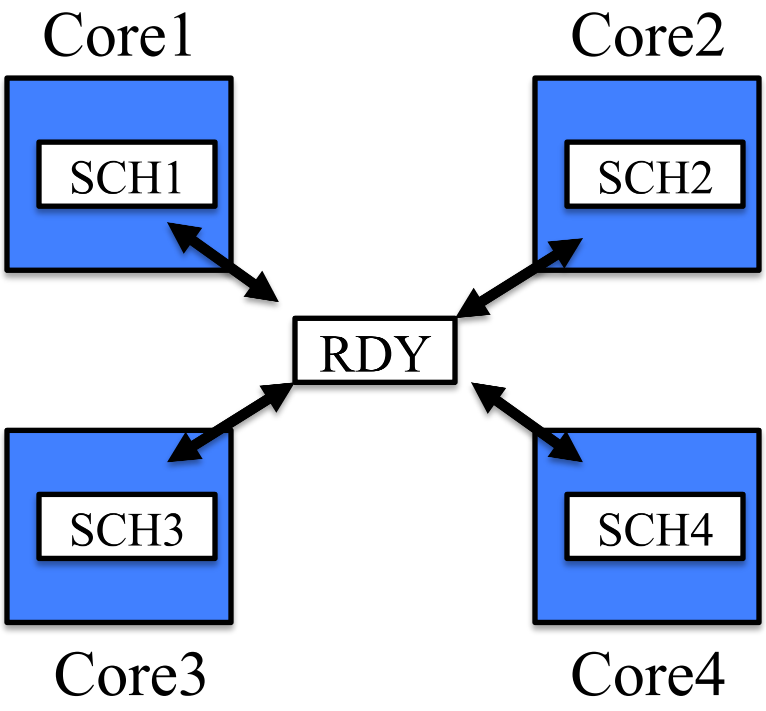


SCH = Scheduler, RDY = Ready Queue

- Variety of multi-core schedulers being tried. We'll just mention some design themes.
- In *asymmetric multiprocessing* (left) – 1 CPU handles all scheduling, decides which processes run on which cores

Multi-core Scheduling

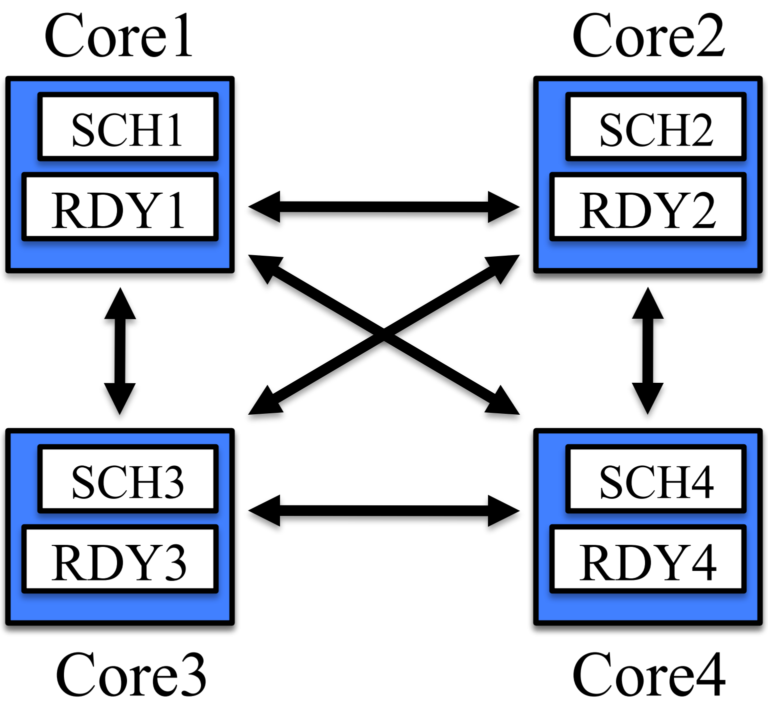
- In symmetric multi-processing (SMP), each process is self-scheduling. All modern OSs support some form of SMP. Two types:
  1. All cores share a single global ready queue.



- 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

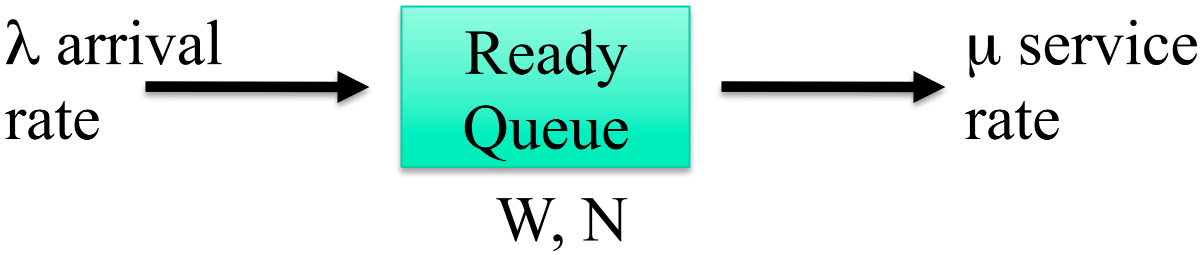
Multi-core Scheduling

- Caching is important to consider
  - Each CPU has its own cache to improve performance
  - If a process migrates too much between CPUs, then have to rebuild L1 and L2 caches each time a process starts on a new processor
    - L3 caches that span multiple processors 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.
  - So processes tend to stick to a given processor – processor affinity
    - Hard vs. soft affinity. In hard affinity, a process specifies via a system call that it insists on staying on a given CPU. In soft affinity, there is still a bias to stick to a CPU, but processes can on occasion migrate.

Multi-core Scheduling: Load balancing

- Goal: Keep workload evenly distributed across processors. Otherwise, some CPUs will be under-utilized.
- When there is a single shared ready queue, there is automatic load balancing, because cores just pull in processes from the ready queue whenever they’re idle.
- For separate ready Q’s,
  - push migration – a dedicated task periodically checks the load on each CPU, and if imbalance, pushes processes from more-loaded to less-loaded CPUs
  - Pull migration – whenever a CPU is idle, it tries to pull a process from a neighboring CPU
  - Linux and FreeBSD use a combination of pull and push

Little’ s Law



λ arrival rate

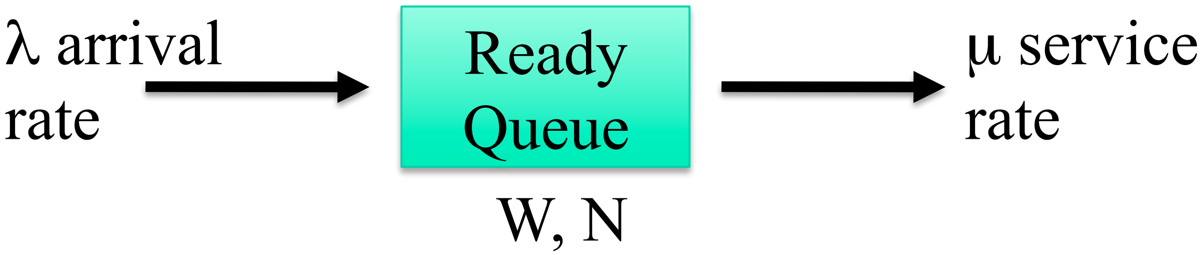
Ready Queue

μ service rate

W, N

- How big of a ready queue do we need?
  - Employ stochastic queueing theory to answer this question
  - Consider an abstract ready queue where λ = arrival rate of processes in the queue, μ = service rate of processes exiting the queue
  - In steady state, λ must be less than or equal to μ, i.e. $\lambda \leq \mu$
  - Let W = average wait time in the queue per process
    - Note W is not the inverse of the service time μ. Analogy: If we have a series of cars that go into a tunnel, eventually they will emerge on the other side, and the steady state service (exit) rate of the cars will equal the arrival (entry) rate. However, the exit rate doesn’ t tell us what is the average time spent by each car in the tunnel. The length of the tunnel (and car speed) tells how long each car spent in the tunnel.
  - Let N = average queue length

Little’ s Law



- Then *N* = λ * *W* (Little’ s Law)
  - In W seconds, on average λ*W processes arrive in the ready queue.
  - This conclusion is valid for any scheduling algorithm and arrival distribution.
  - Useful to know how big on average the queue size N should be for allocation by the OS.