Design and Analysis of Operating Systems

CSCI 3753

Dr. David Knox
University of Colorado Boulder

Department of Computer Science
UNIVERSITY OF COLORADO **BOULDER**

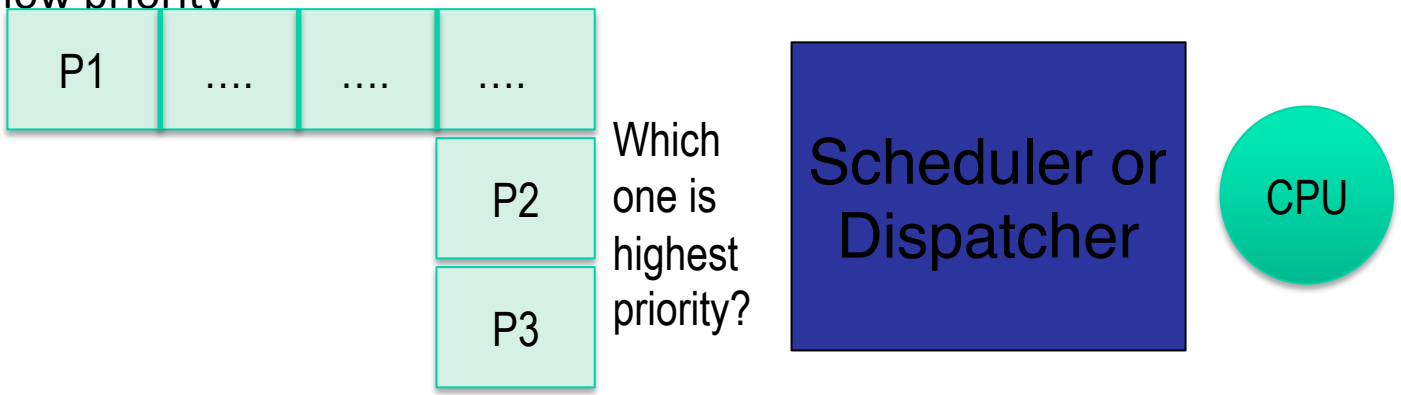
Design and Analysis of Operating Systems CSCI 3753

Process Scheduling Priority and Multi-Level

Dr. David Knox
University of
Colorado Boulder

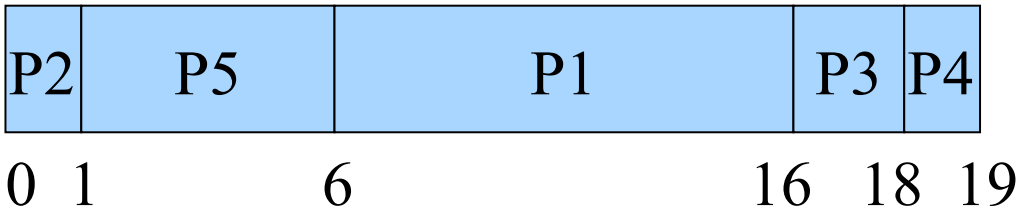
Priority-based Scheduling

- **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



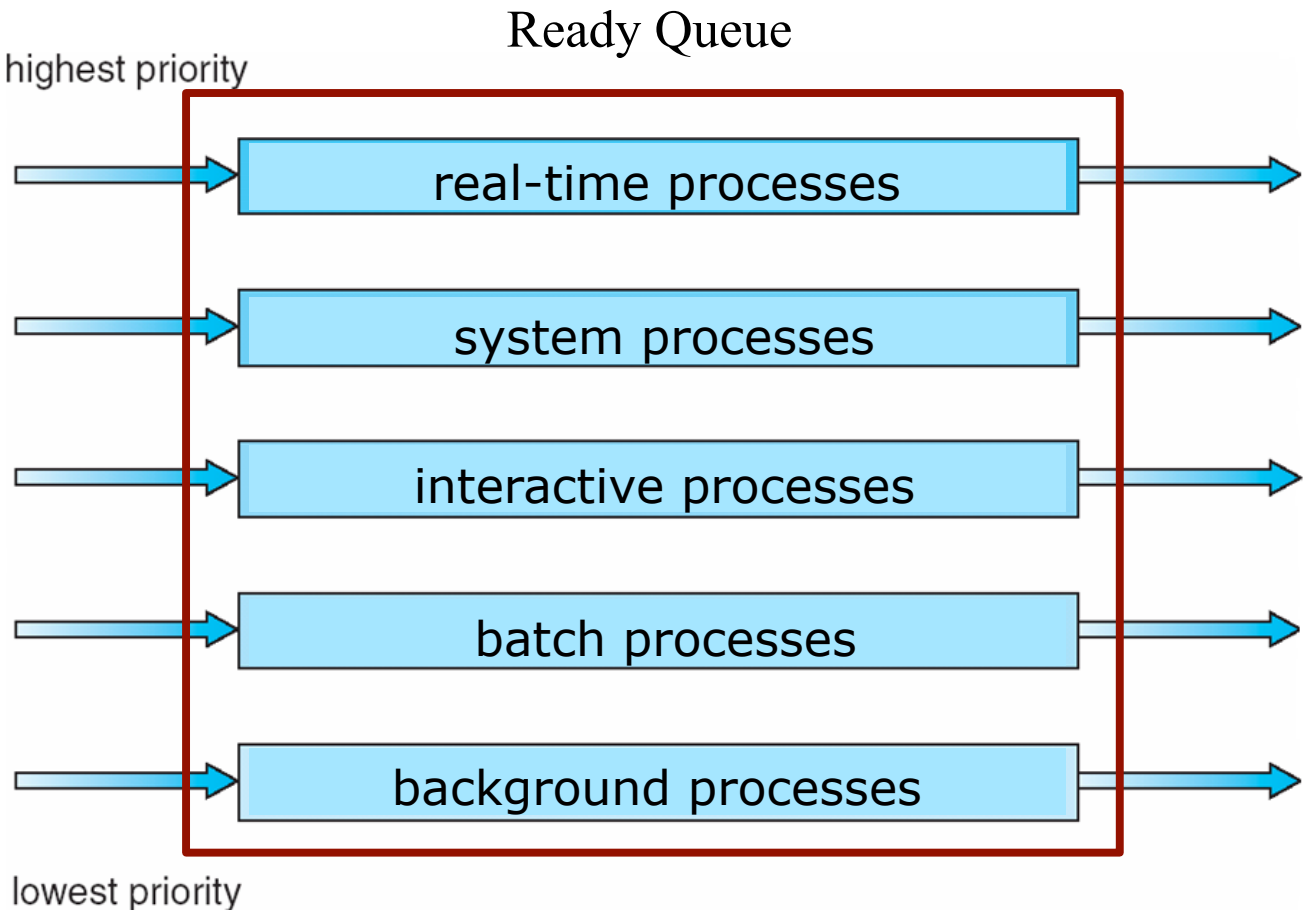
Priority-based Scheduling

- **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

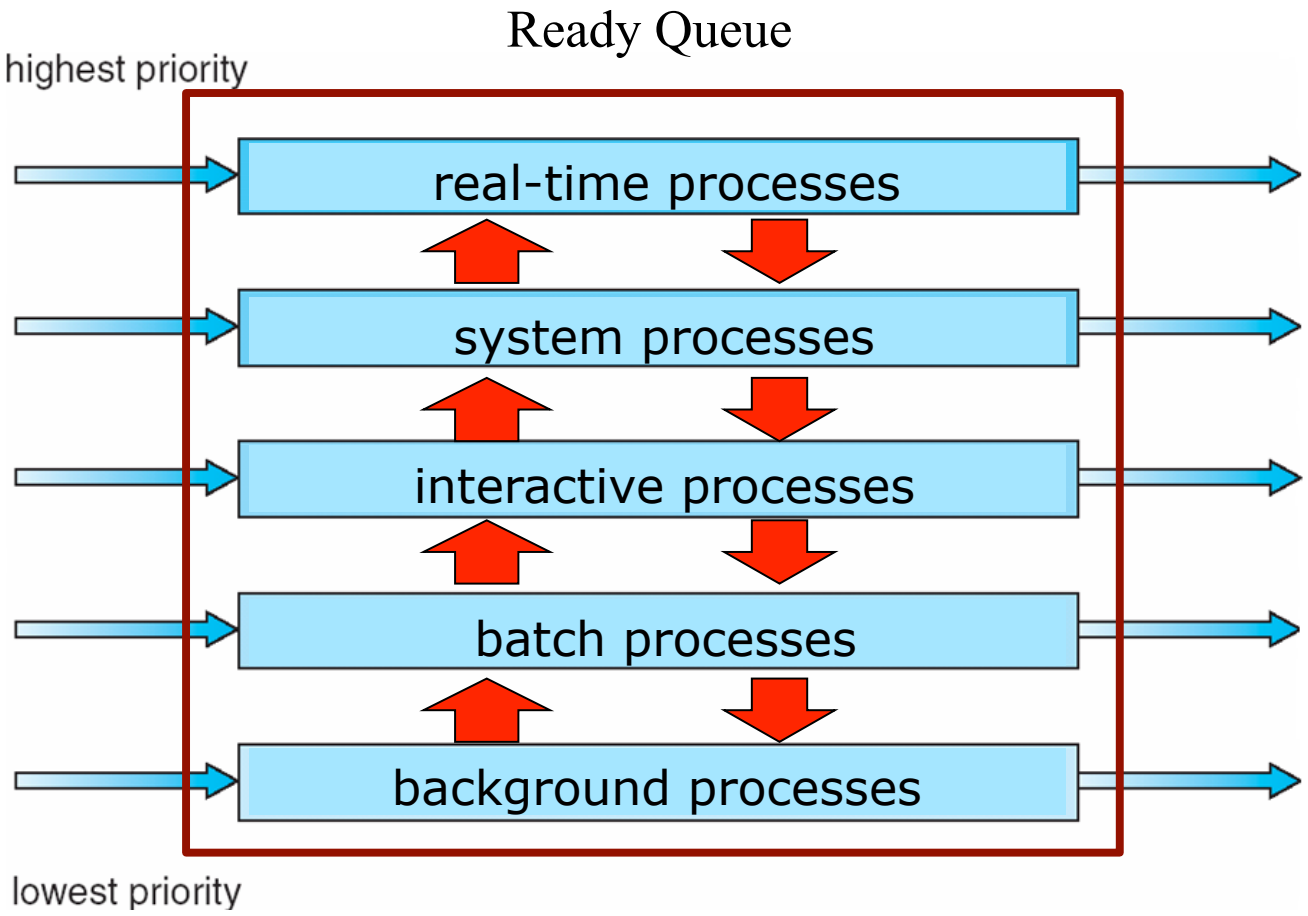


Priority-based Scheduling

- **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

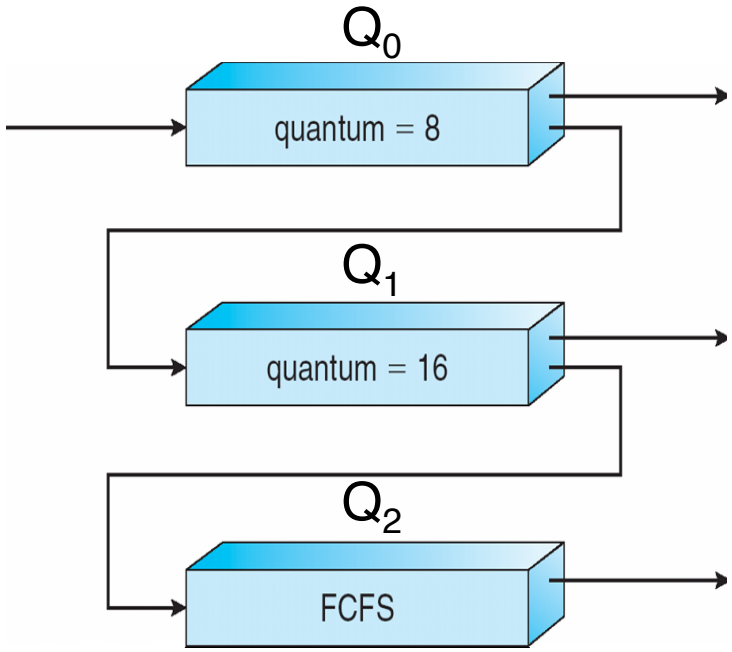
Multi-level Feedback Queues

- **Criteria for process movement among priority queues could depend upon behavior of a process:**
  - could be 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 lowest 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

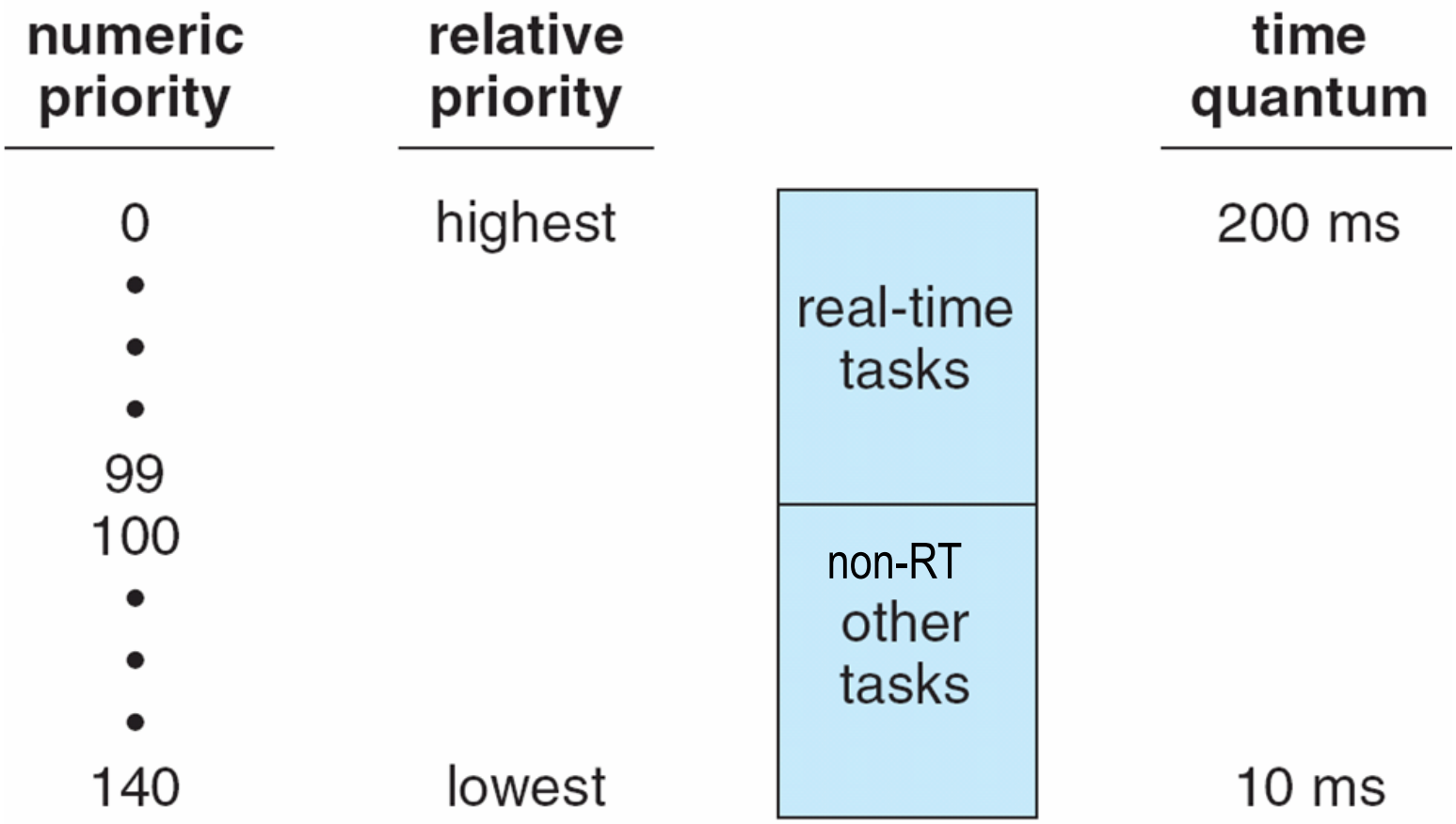
Priority-based Scheduling

- **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
  - Linux 2.4 and above next lecture…

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

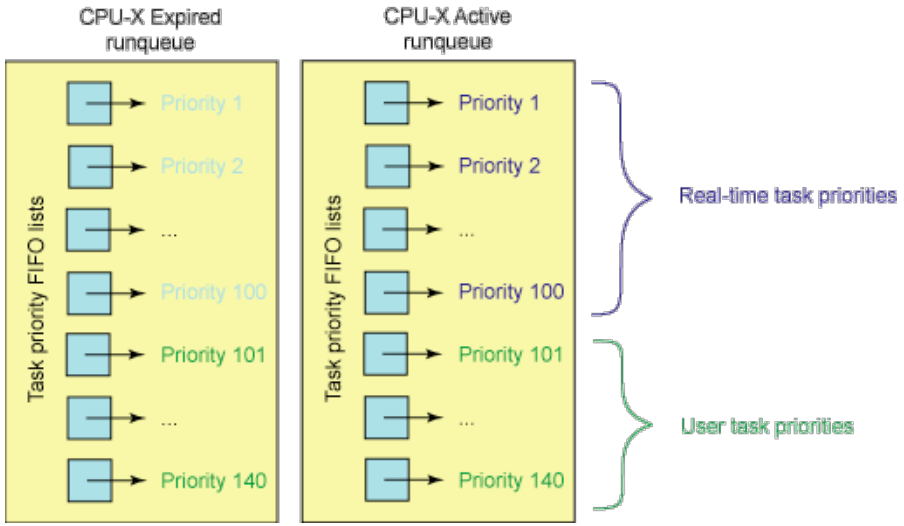
More Linux Scheduler History

- **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

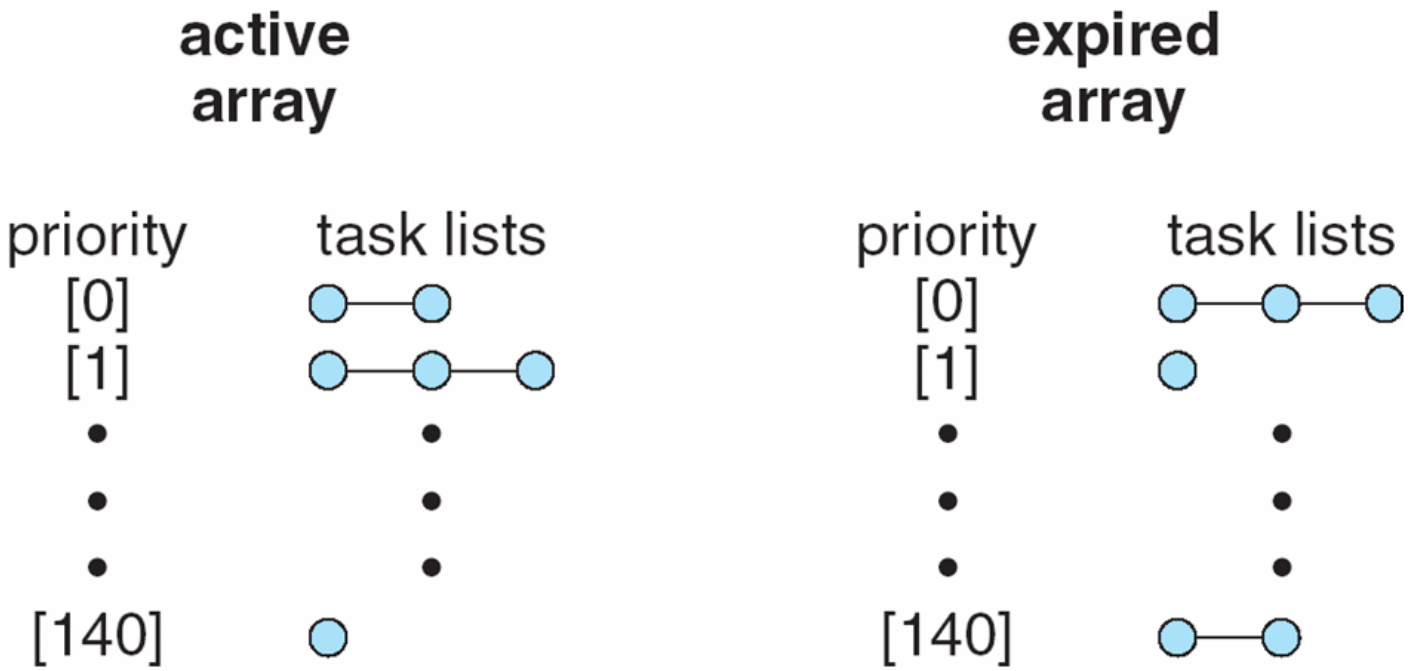
- **Linux maintains two queues:**
  - an active array or run queue and an expired array/queue, each indexed by 140 priorities
- **Active array contains all tasks with time remaining in their time slices, and expired array contains all expired tasks**

- 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

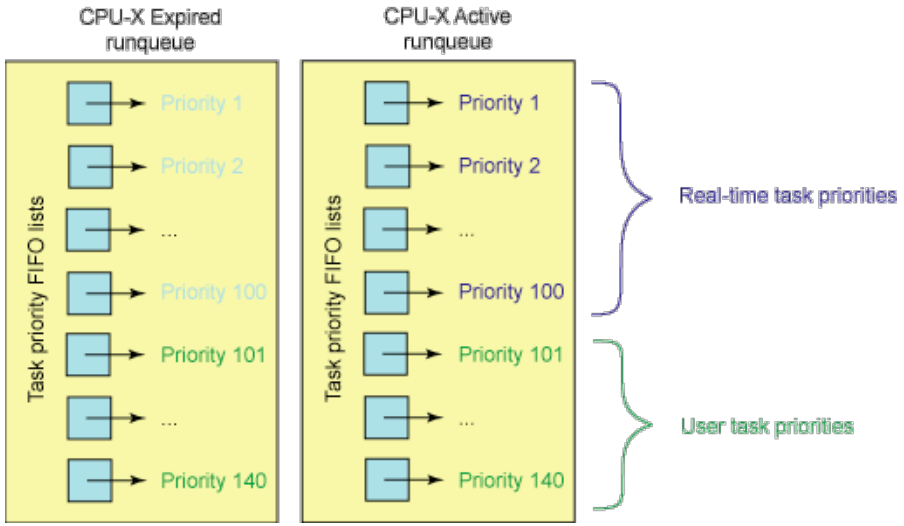


O(1) Scheduler in Linux

- **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**

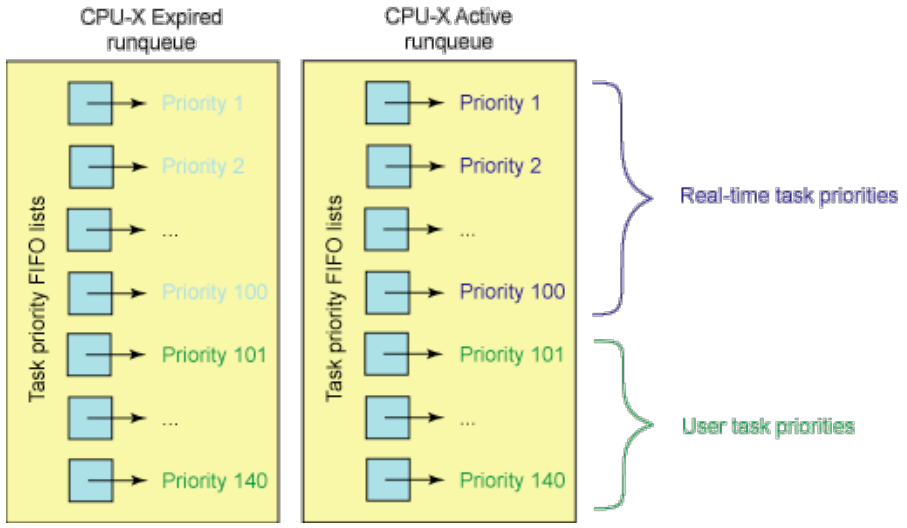
  – Just search linearly through the active array from priority 1 until you find the first priority whose queue contains at least one unexpired task



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

Department of Computer Science
UNIVERSITY OF COLORADO **BOULDER**

Design and Analysis of Operating Systems CSCI 3753

Dr. David Knox
University of
Colorado Boulder