CPU Scheduling – Continued.

CS 416: Operating Systems Design, Spring 2011

Department of Computer Science
Rutgers University

Scheduling Algorithms

- FIFO is simple but leads to poor average response times. Short processes are delayed by long processes that arrive before them
- RR eliminate this problem, but favors CPU-bound jobs, which have longer CPU bursts than I/O-bound jobs
- SJN and SRT alleviate the problem with FIFO, but require information on the length of each process. This information is not always available (although it can sometimes be approximated based on past history or user input)
- Feedback is a way of alleviating the problem with FIFO without information on process length

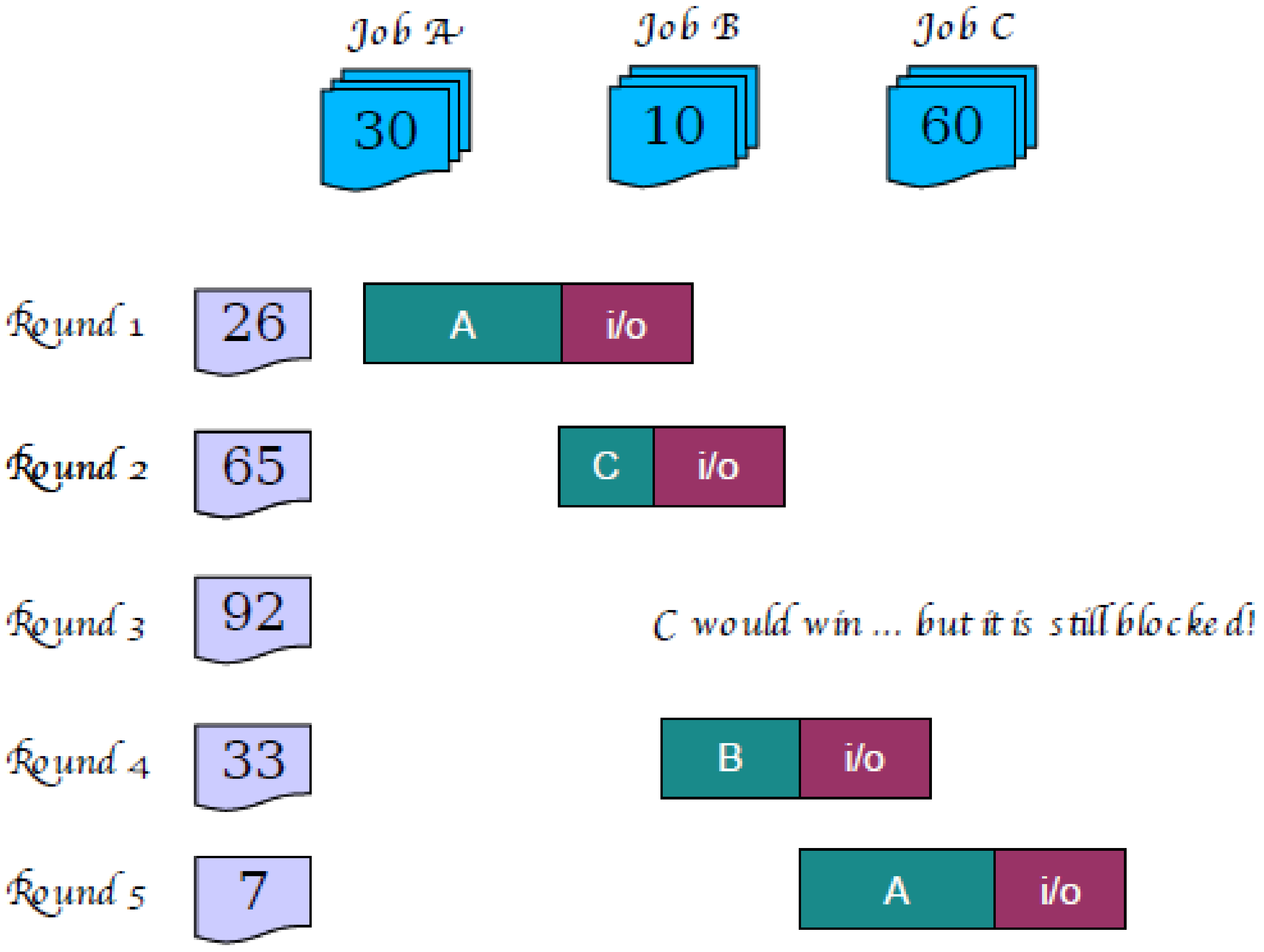
It’s a Changing World

- Assumption about bi-modal workload no longer holds
  - Interactive continuous media applications are sometimes processor-bound but require good response times (Ex. Games)
- New computing model requires more flexibility
  - How to match priorities of cooperative jobs, such as client/server jobs?
  - How to balance execution between multiple threads of a single process?

Lottery Scheduling [OSDI 1994]

- A kind of *randomized priority* scheduling scheme!
- Give each thread some number of "tickets"
  - The more tickets a thread has, the higher its priority
- On each scheduling interval:
  - Pick a random number between 1 and total # of tickets
  - Scheduling the job holding the ticket with this number
- How does this avoid starvation?
  - Even low priority threads have a small chance of running!

Lottery Scheduling Example



Advantages of Lottery Scheduling

- How about Flexible Control?
  - Ex. Monte Carlo Simulations. The first few rounds have larger errors but gives an approximate answer.
  - Later runs have lower answers and try to converge to accurate answer
  - When errors are higher, The application could inflate the number of currencies it holds and be scheduled with high priority to yield a quick answer.
  - When errors go down, the tickets could be lowered and this process could run slow and return an accurate answer at a later time.
- How about Client Server Application ?
  - When Clients wait for server, clients can pass on some tickets to the server and give server a higher of winning.

Traditional UNIX Scheduling

- Multilevel feedback queues
- 128 priorities possible (0-127)
- 1 Round Robin queue per priority
- Lower Number -> Higher Priority
- Every scheduling event the scheduler picks the lowest number non-empty queue and runs jobs in round-robin
- Scheduling events:
  - Clock interrupt
  - Process does a system call
  - Process gives up CPU,e.g. to do I/O

Traditional UNIX Scheduling

- All processes assigned a baseline priority based on the type and current execution status:

swapper	0
waiting for disk	20
waiting for lock	35

- At scheduling events, **all process's priorities** are adjusted based on the amount of CPU used, the current load, and how long the process has been waiting.

UNIX Priority Calculation

- Every 4 clock ticks a processes priority is updated:

$$P = BASELINE + \left\lfloor \frac{utilization}{4} \right\rfloor + 2\, NiceFactor$$

- The utilization of running process is incremented every clock tick by 1.
- The Nice-Factor allows some control of job priority. It can be set from –20 to 20. Users control the Nice Factor.
- Jobs using a lot of CPU increase the priority value. Interactive jobs not using much CPU will return to the baseline.

UNIX Priority Calculation

- Very long running CPU bound jobs will get “stuck” at the highest priority.
- Decay function used to weight utilization to recent CPU usage.
- A process’s utilization at time ***t*** is decayed every second:

$$u_t = \left[\frac{2\,load}{(2\,load + 1)} \right] u_{(t-1)}$$

- The system-wide load is the average number of runnable jobs during last 1 second

UNIX Priority Decay

- 1 job on CPU. load will thus be 1. Assume niceFactor is 0.
- Compute utilization at time N:

- +1 second:

$$U_1 = \frac{2}{3} U_0$$

- +2 seconds

$$U_2 = \frac{2}{3}\left[U_1 + \frac{2}{3} U_0\right] = \frac{2}{3} U_1 + \left(\frac{2}{3}\right)^2 U_0$$

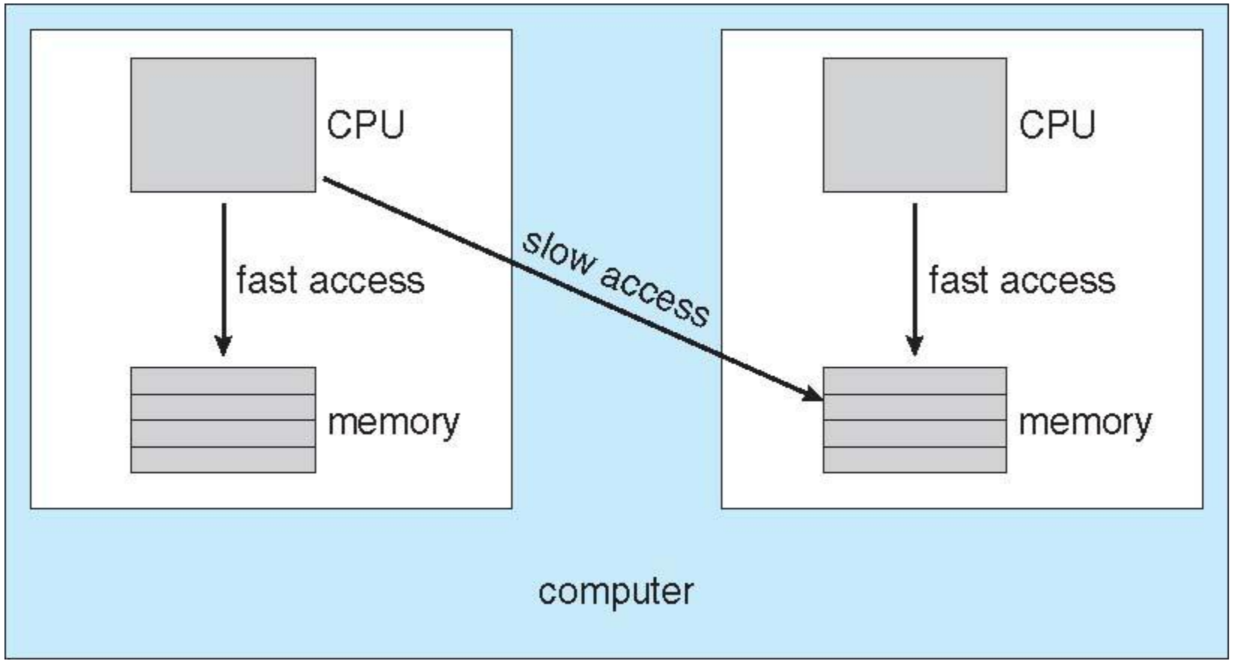
- +N seconds

$$U_n = \frac{2}{3} U_{n-1} + \left(\frac{2}{3}\right)^2 U_{n-2} \ldots$$

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**

NUMA and CPU Scheduling

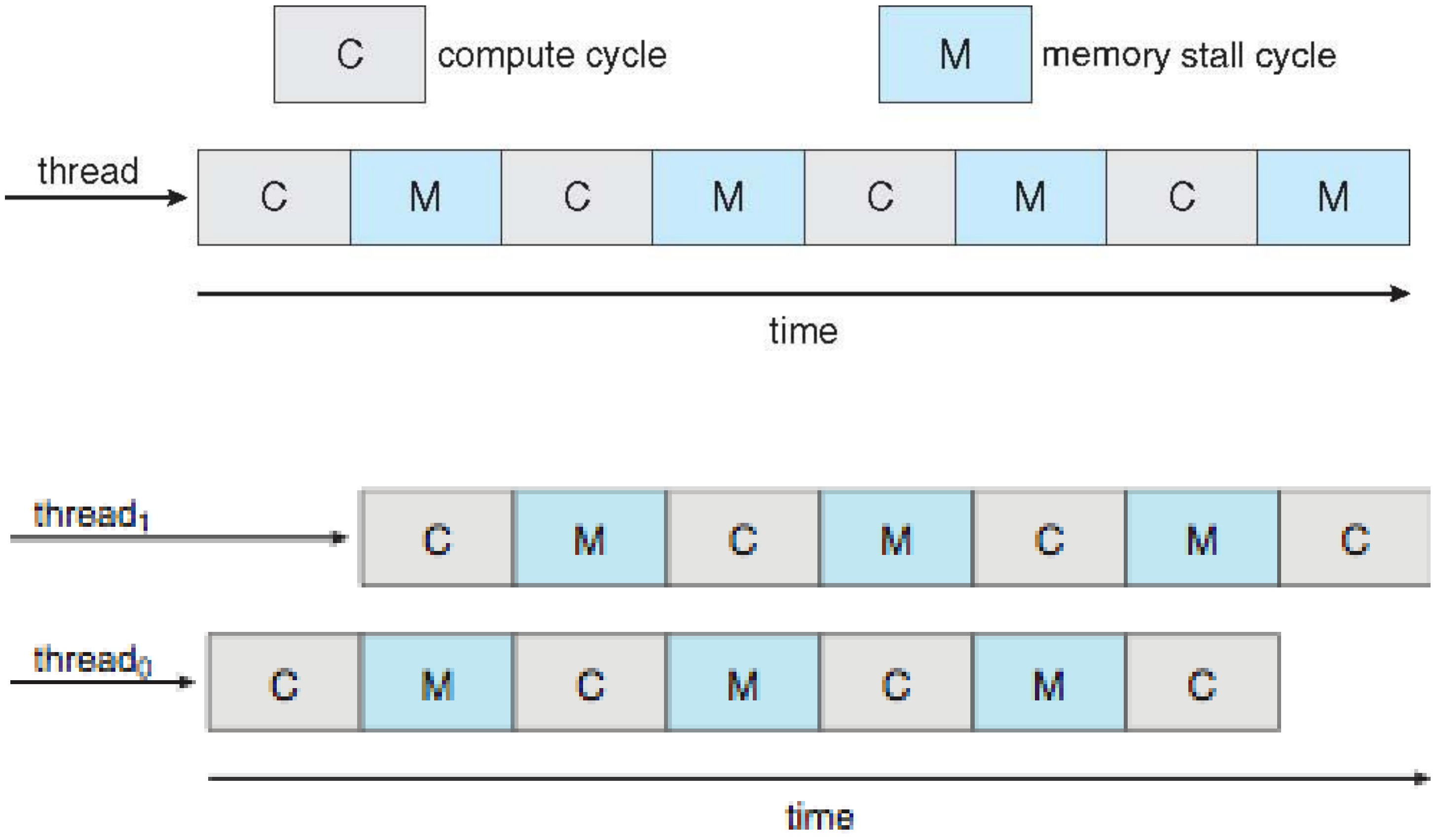


Note that memory-placement algorithms can also consider affinity

Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

Multithreaded Multicore System



Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)
- Each guest doing its own scheduling
  - Not knowing it doesn’t own the CPUs
  - Can result in poor response time
  - Can effect time-of-day clocks in guests
- Can undo good scheduling algorithm efforts of guests

Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

Solaris

- Priority-based scheduling
- Six classes available
  - Time sharing (default)
  - Interactive
  - Real time
  - System
  - Fair Share
  - Fixed priority
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue

$\left.\begin{array}{l}\text{Time sharing (default)}\\ \text{Interactive}\end{array}\right\}$ 60 Priority Queues

$\left.\begin{array}{l}\text{Time sharing, Interactive, Real time,}\\ \text{System, Fair Share, Fixed priority}\end{array}\right\}$ ~169 Priority Queues

Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Higher priority thread gets lower quantum
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

Windows Scheduling

- Windows uses priority-based preemptive scheduling
- 32-level priority scheme
  - **Variable class** is 1-15, **real-time class** is 16-31
  - Priority 0 => Memory management thread (creates free frames)
  - Higher number => Higher Priority
  - Queue for each priority
- Highest-priority thread runs next
- Thread runs until
  - (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time

Windows XP Priorities

Different Priority Classes

Relative Priority

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Base Priority

- If quantum expires, priority lowered, but never below base
- If wait occurs, priority boosted depending on what was waited for
  - Waiting for I/O gets higher boost than disk
- Foreground window given 3x priority boost

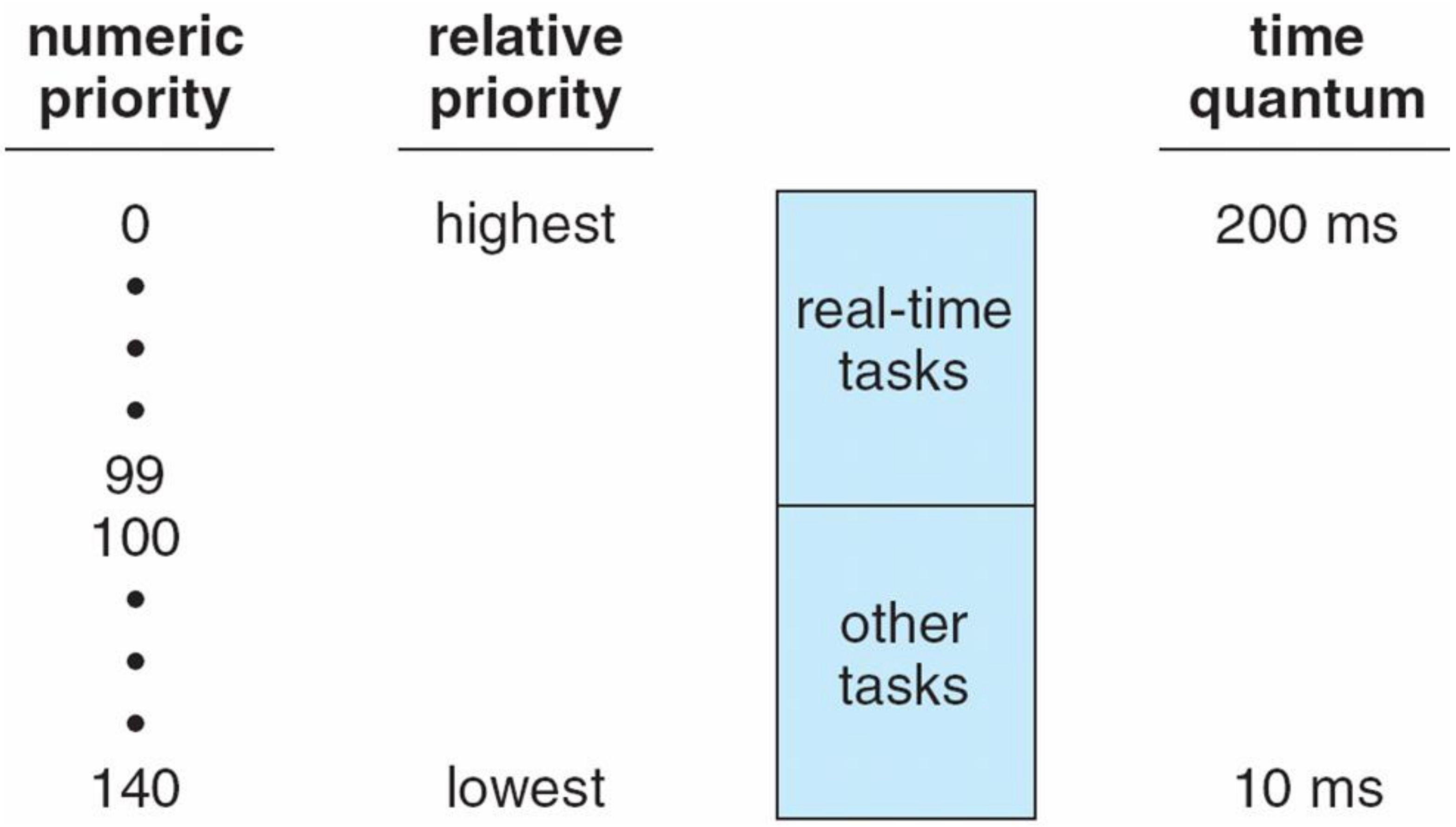
Linux 2.6 Scheduling

- Preemptive, priority based
- Two priority ranges:
  - **Real Time (0 – 99)**
  - **Time Sharing (100-139)**
- Numerically lower values indicating higher priority
- Higher priority gets larger quantum
- Task run-able as long as time left in time slice (**active**)
- If no time left (**expired**), not run-able until all other tasks use their slices
- When no more active, arrays are exchanged

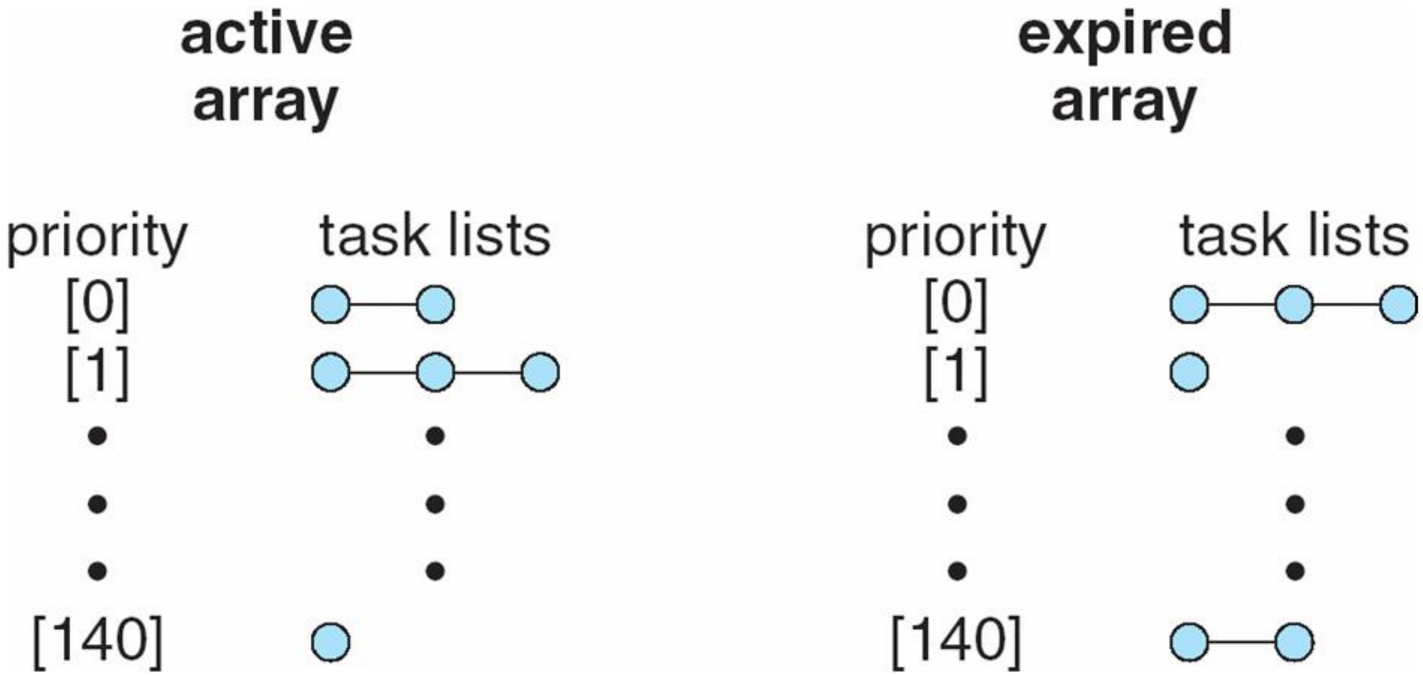
Linux 2.6 Scheduling

- Each time-sharing task has two priorities: **static priority and dynamic priority**
- Static priority
  - Ranges from 100-139, default value is 120
  - Only changed using the nice() system call – change by -20 (higher) to +19 (lower)
- Dynamic priority: This is the value actually used by the scheduler
  - Represents static priority plus a dynamic “bonus”
- How the “bonus” is calculated
  - Bonus range is between -5 (higher priority) to +5 (lower priority)
  - I/O bound tasks given boost of up to -5
  - CPU bound jobs given penalty of up to +5
  - Bonus calculated by taking ratio of task's “wait time” to “running time”
    - *Idea: Task that waits more often is probably I/O bound*

Priorities and Time-slice length



List of Tasks Indexed according to Priorities



active array
priority
task lists
[0]
[1]
[140]
expired array
priority
task lists
[0]
[1]
[140]