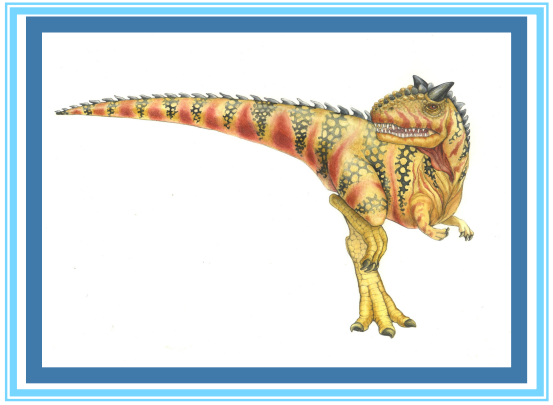
Chapter 5: CPU Scheduling



Scheduling Policies

- Non-preemptive
  - First Come First Served
  - Shortest Job First (aka Shortest Process Next)
- Preemptive
  - Shortest remaining time first
  - Priority
  - Round Robin

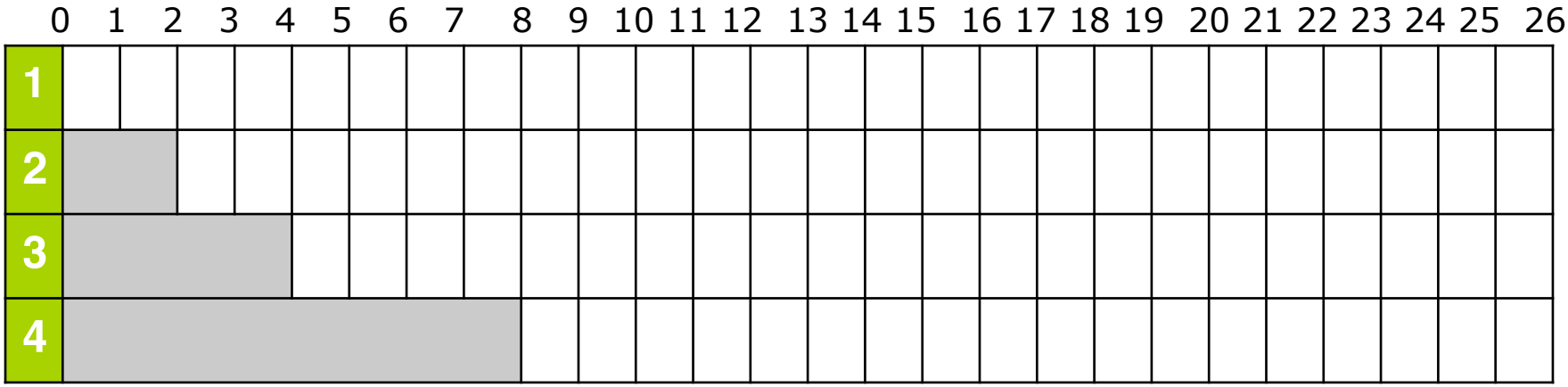
Example Process Arrivals

- Perform the following schedulings
  - FCFS
  - Shortest Job First (SJF)
  - Shortest Remaining-time First (SRTF)
  - Priority
  - Round Robin (RR)

Process	Arrival	CPU	Priority
P1	0	8	4
P2	2	4	3
P3	4	9	2
P4	8	5	1

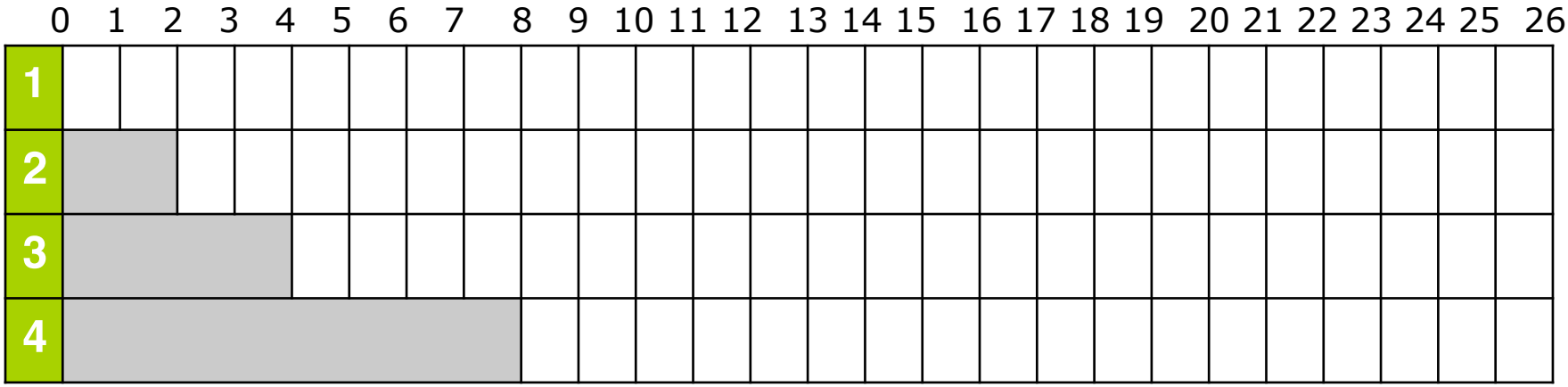
FCFS

Process	Arrival	CPU	Priority
P1	0	8	4
P2	2	4	3
P3	4	9	2
P4	8	5	1



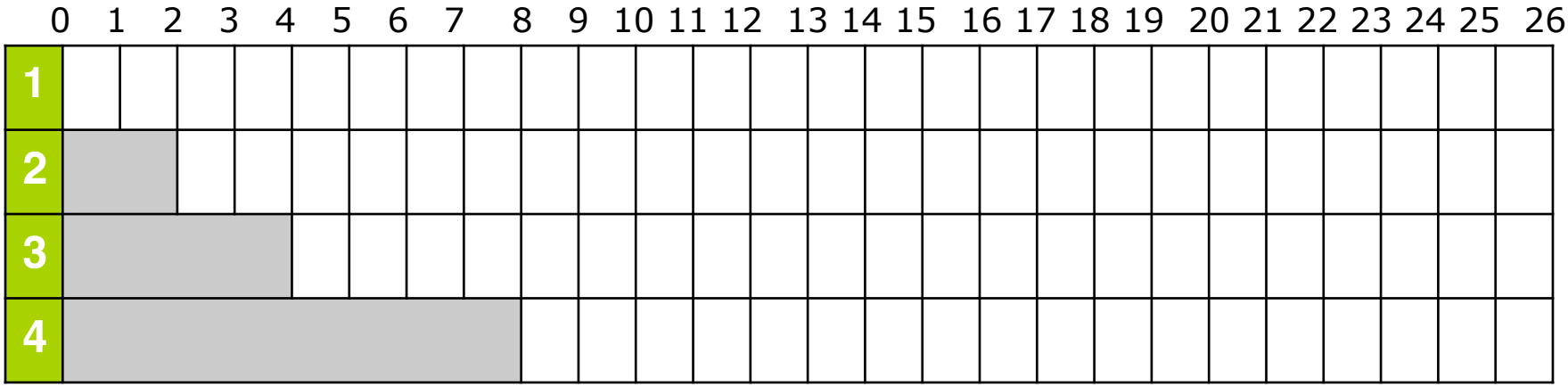
SJF

Process	Arrival	CPU	Priority
P1	0	8	4
P2	2	4	3
P3	4	9	2
P4	8	5	1



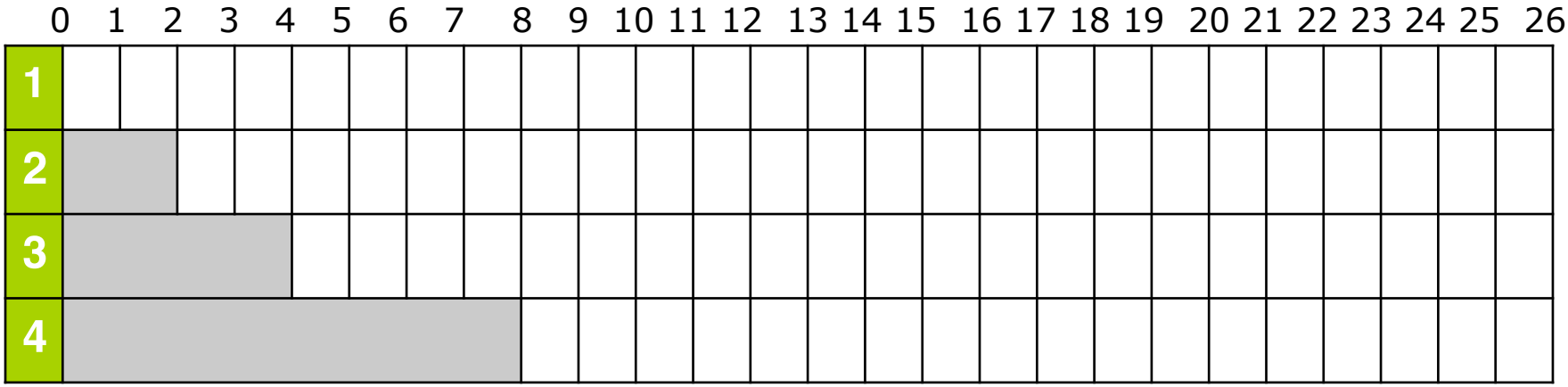
SRTF

Process	Arrival	CPU	Priority
P1	0	8	4
P2	2	4	3
P3	4	9	2
P4	8	5	1



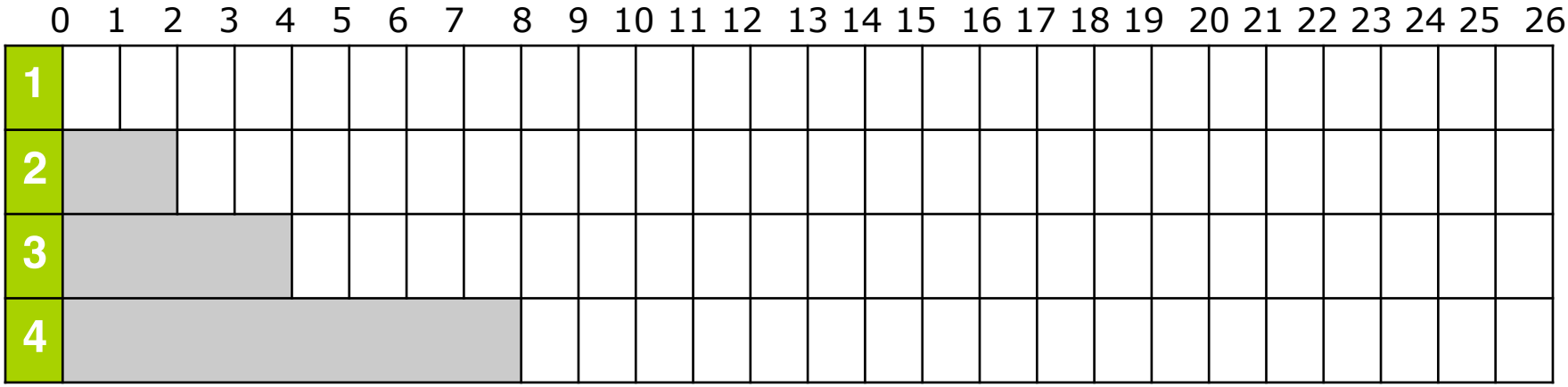
Priority

Process	Arrival	CPU	Priority
P1	0	8	4
P2	2	4	3
P3	4	9	2
P4	8	5	1



Round Robin (q=1)

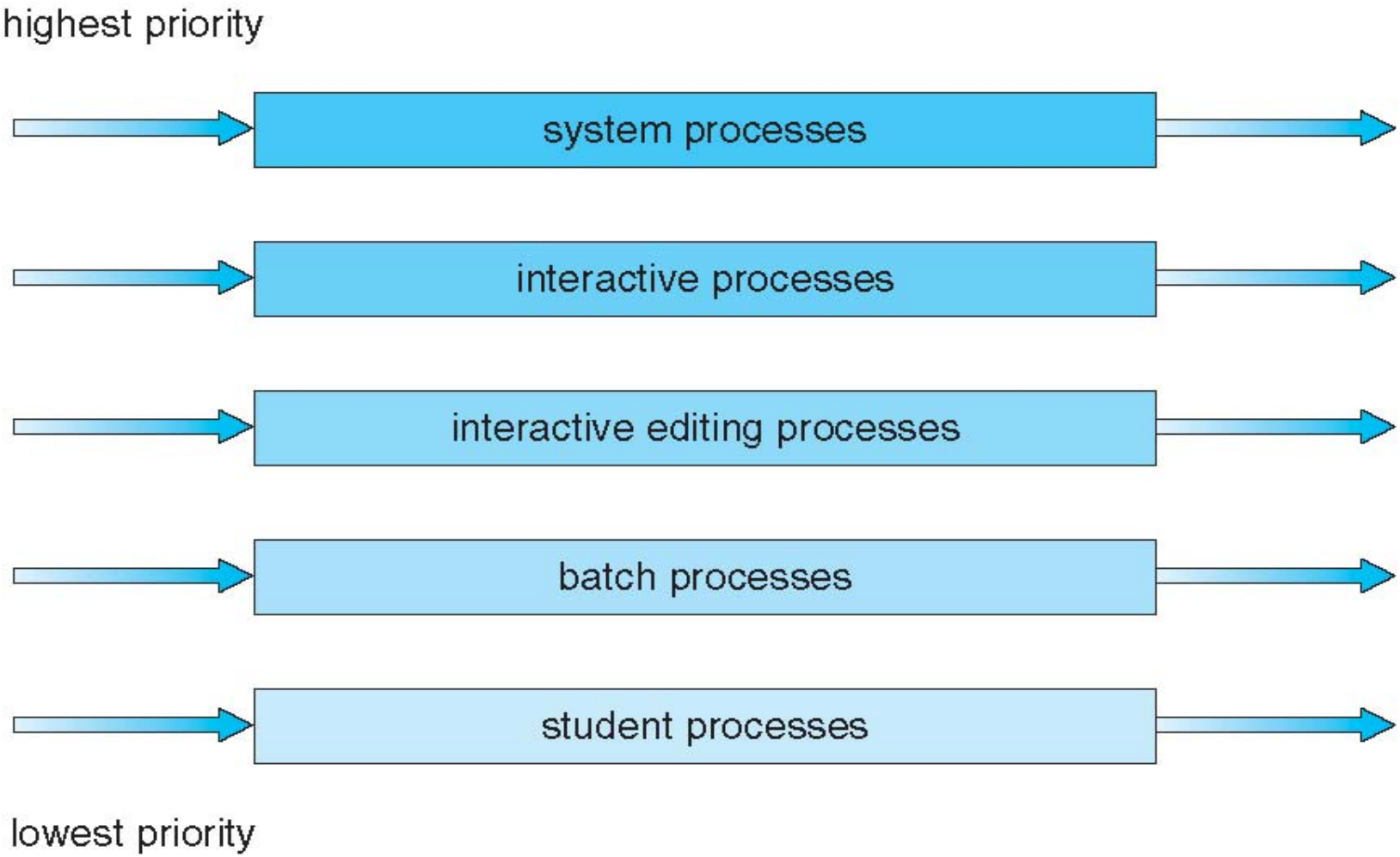
Process	Arrival	CPU	Priority
P1	0	8	4
P2	2	4	3
P3	4	9	2
P4	8	5	1



Multilevel Queue

- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice
    - each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
    - 20% to background in FCFS

Multilevel Queue Scheduling

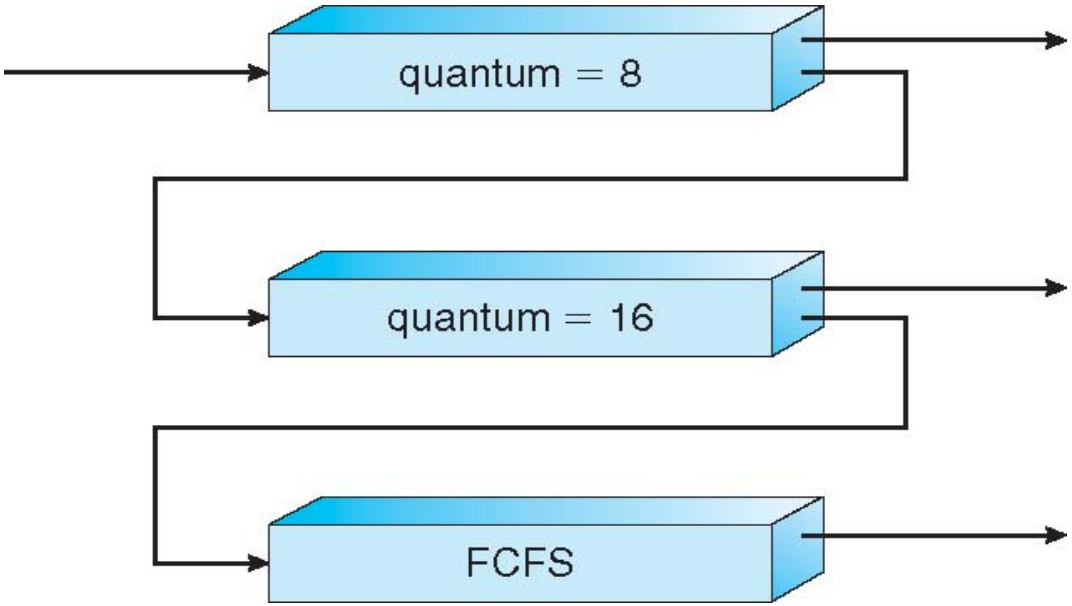


Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - when to upgrade a process
  - when to demote a process
  - which queue a new process will enter

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



Example of Multilevel Feedback Queue

- Scheduling
  - Q_1 is scheduled only when Q_1 is empty
  - Q_2 is scheduled only when Q_1 is empty
  - Process in Q_1 is preempted by process in Q_0.
  - Process in Q_2 is preempted by process in Q_1.
  - A new job enters queue Q_0
    - If it does not finish in 8 milliseconds, moved to Q_1
  - A process in Q_1 takes more than 16 milliseconds
    - moved to queue Q_2

Windows Scheduling

- Windows uses priority-based preemptive scheduling
- *Dispatcher* is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
  - REALTIME_PRIORITY_CLASS
  - HIGH_PRIORITY_CLASS
  - ABOVE_NORMAL_PRIORITY_CLASS
  - NORMAL_PRIORITY_CLASS
  - BELOW_NORMAL_PRIORITY_CLASS
  - IDLE_PRIORITY_CLASS
- A thread within a given priority class has a relative priority
  - TIME_CRITICAL
  - HIGHEST
  - ABOVE_NORMAL
  - NORMAL
  - BELOW_NORMAL
  - LOWEST
  - IDLE

Windows XP Priorities

	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

Windows Priority Classes

- If quantum expires, priority lowered, but never below base
- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x quantum boost

Linux Scheduling

- 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 **nice** value from 100 to 140
- Higher priority gets larger quantum

Priorities and Time-slice length

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

Linux Scheduling

- Each processor maintains a *runqueue*
- Runqueue consists of *active* and *expired* priority array
- 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

List of Tasks Indexed According to Priorities

Linux Scheduling (Cont.)

- Real-time tasks have static priorities
- Other tasks have dynamic priorities, based on *nice* value
  - More interactive (longer I/O-related sleep): -5
  - More CPU-bound (less sleep): +5
  - Priority recalculated when task expired
  - This exchanging arrays implements adjusted priorities