CPU Scheduling

CSC 256/456 - Operating Systems
Fall 2014

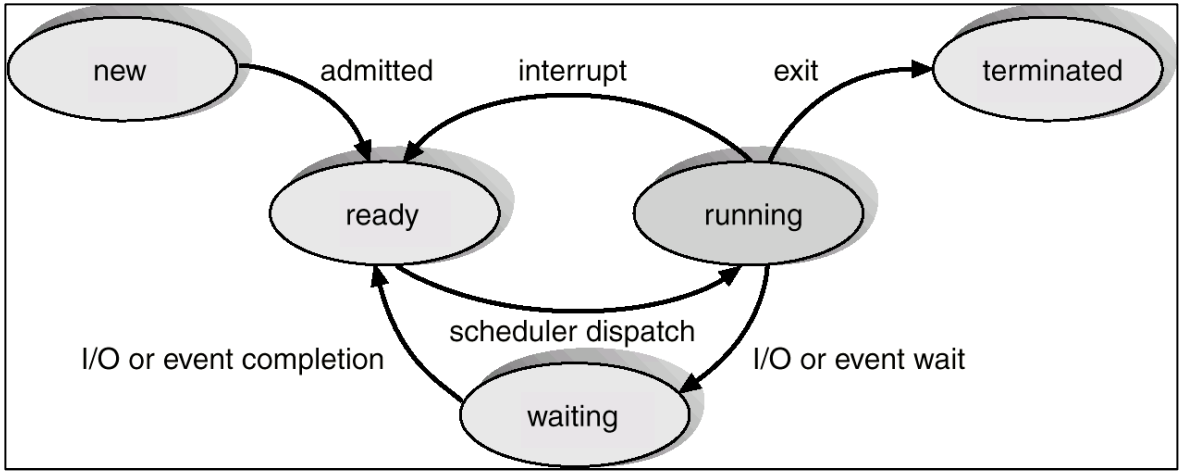
TA: Mohammad Hedayati

Agenda

- Scheduling Policy Criteria
- Scheduling Policy Options (on Uniprocessor)
- Multiprocessor scheduling considerations
- CPU Scheduling in Linux
- Real-time Scheduling

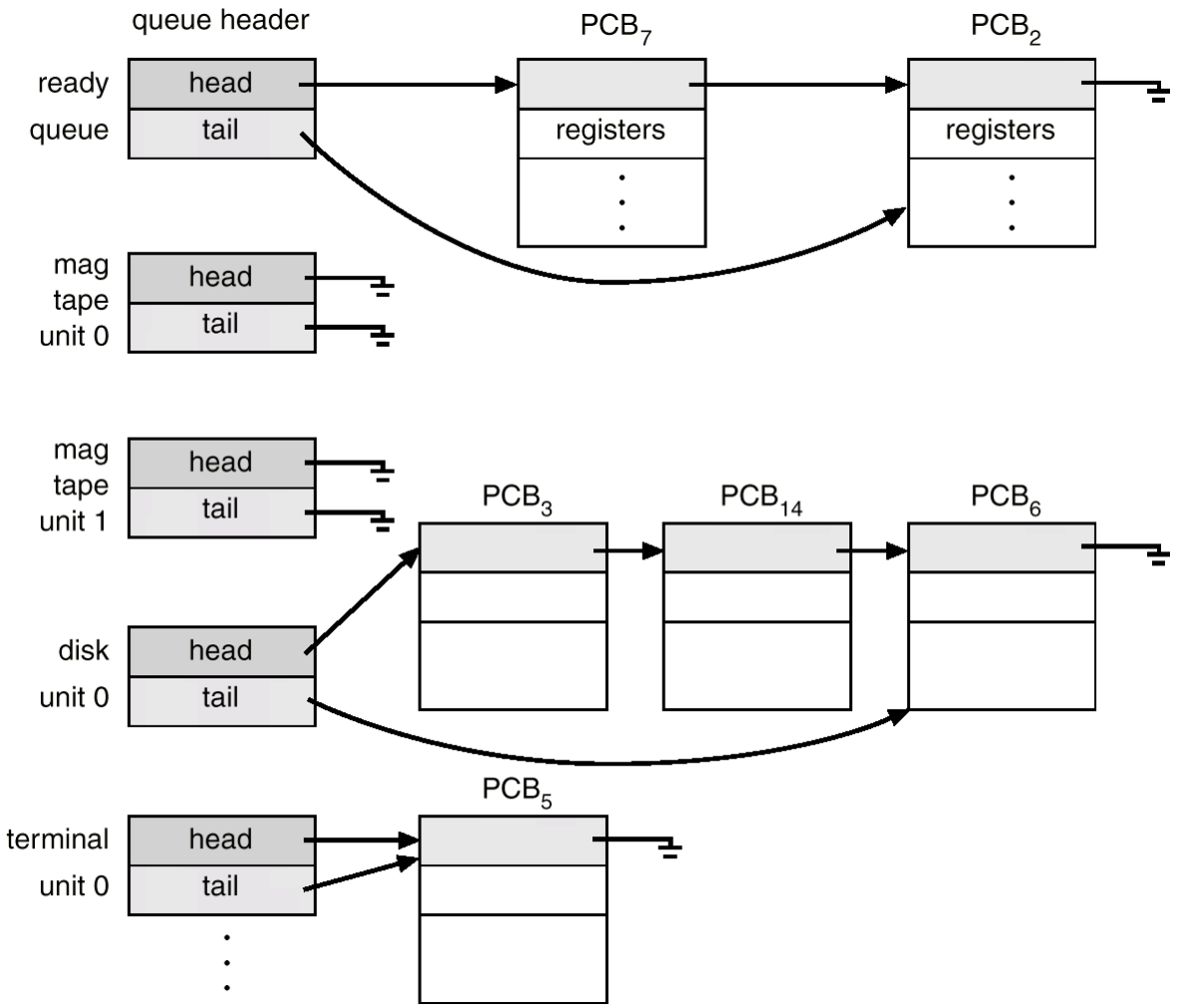
Process States

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a process
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process has finished execution



Queues of PCBs

- **Ready Queue**: set of all processes ready for execution.
- **Device Queue**: set of processes waiting for IO on that device.
- Processes migrate between the various queues.



Scheduling

- **Question**: How the OS should decide which of the several processes to take of the queue?
  - We will only worry about ready queue, but it is applicable to other queues as well.
- Scheduling Policy: which process is given the access to resources?

CPU Scheduling

- Selects from among the processes/threads that are ready to execute, and allocates the CPU to it

- CPU scheduling may take place at:
  - hardware interrupt
  - software exception
  - system calls

- Non-Preemptive
  - scheduling only when the current process terminates or not able to run further (due to IO, or voluntarily calling `sleep()` or `yield()`)

- Preemptive
  - scheduling can occur at any opportunity possible

Some Simplifying Assumptions

- For the first few algorithms:
  - Uni-processor
  - One thread per process
  - Programs are independant

- Execution Model: Processes alternate between bursts of CPU and IO

- **Goal**: deal out CPU time in a way that some parameters are optimized.

CPU Scheduling Criteria

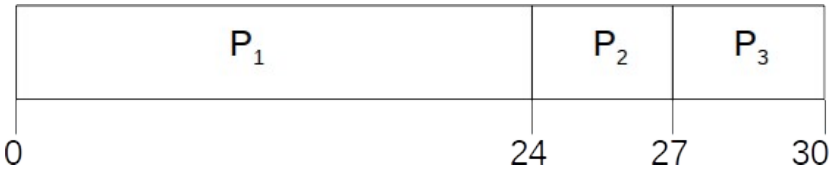
- *Minimize* **waiting time**: amount of time the process is waiting in *ready queue*.
- *Minimize* **turnaround time**: amount of time that the system takes to execute a process (= waiting time + execution time in absence of any other process).
  - **response time**: amount of time that the system takes to produce the first response (interactivity). e.g. echo back a keystroke on editor
- *Maximize* **throughput**: # of processes that complete their execution per time unit (usu. in a long run)
- *Maximize* CPU **utilization**: the proportion of time that CPU is doing useful job.
- **Fairness**: avoid starvation

First-Come, First-Served (FCFS)

Process	Arrival	CPU Time
P1	0	24
P2	0	3
P3	0	3

- Suppose that the processes arrive in the order: P1 , P2 , P3.

The schedule gantt chart is:

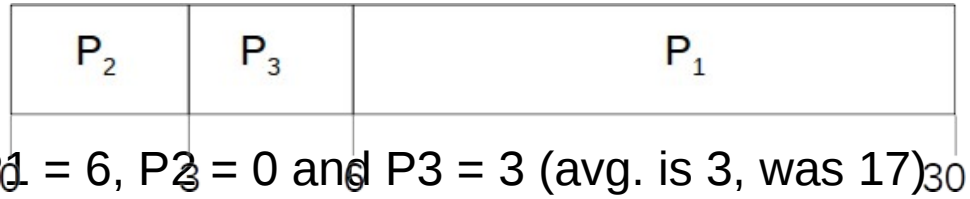


- Waiting time: P1 = 0, P2 = 24 and P3 = 27 (avg. is 17)
- Turnaround time: P1 = 24, P2 = 27 and P3 = 30 (avg. is 27)
- Is it fair?

FCFS (cont.)

- Suppose that the processes arrive in the order: P1 , P2 , P3.

Now, the schedule is:



- Waiting time: P1 = 6, P2 = 0 and P3 = 3 (avg. is 3, was 17)
- Turnaround time: P1 = 30, P2 = 3 and P3 = 6 (avg. is 13, was 27)

- Short process delayed by long process: **Convoy effect**

- Pros and Cons?

Shortest Job First (SJF)

- Associate with each process the length of its CPU time. Use these lengths to schedule the process with the shortest CPU time.

- Two variations:
  - Non-preemptive: once CPU given to the process it cannot be taken away until it completes.
  - Preemptive: if a new process arrives with CPU time less than remaining time of current executing process, preempt (a.k.a Shortest Remaining Job First - SRJF)

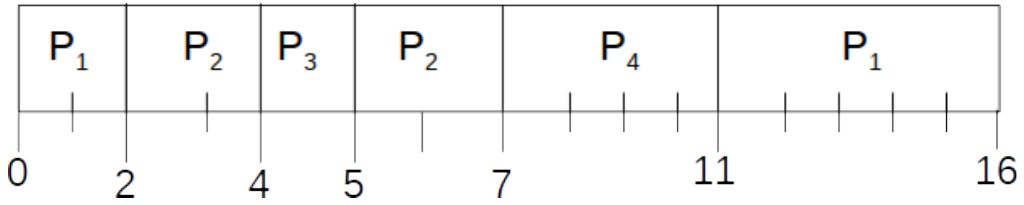
- Preemptive SJF is optimal: gives minimum average turnaround time for a given set of processes

- Problem:
  - don’t know the process CPU time ahead of time
  - is it fair?

Example of Preemptive SJF

Process	Arrival Time	CPU Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

- SJF (Preemptive)



- Waiting time: P1 = 9, P2 = 1, P3 = 0 and P4 = 2 (avg. is 3)
- Turnaround time: P1 = 16, P2 = 5, P3 = 1 and P4 = 6 (avg. is 7)

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority
  - Preemptive vs. Non-Preemptive
- SJF is a priority scheduling where priority is the predicted CPU time.
- Problem: **Starvation** – low priority processes may never execute
- Solution: **Aging** – as time progresses, increase the priority of the process

Round Robin (RR)

- Each process gets a fixed unit of CPU time (time quantum), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue

- If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once. No process waits more than (n-1)q time units

- Performance:
  - q small fair, starvation-free, better interactivity
  - q large FIFO (q = ∞)
  - q must be large with respect to context switch cost, otherwise overhead is too high

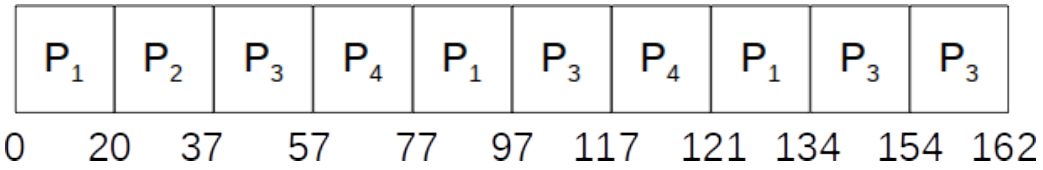
Cost of Context Switch

- Direct overhead of context switch
  - saving old contexts, restoring new contexts, …
- Indirect overhead of context switch
  - caching and memory management overhead

Example of RR with Quantum = 20

Process	CPU Time
P1	53
P2	17
P3	68
P4	24

- The schedule is:



- Typically, higher avg. turnaround time, but better *response time*.

Multilevel Scheduling

- Ready tasks are partitioned into separate classes:
  - foreground (interactive)
  - background (batch)

- Each class has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS

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

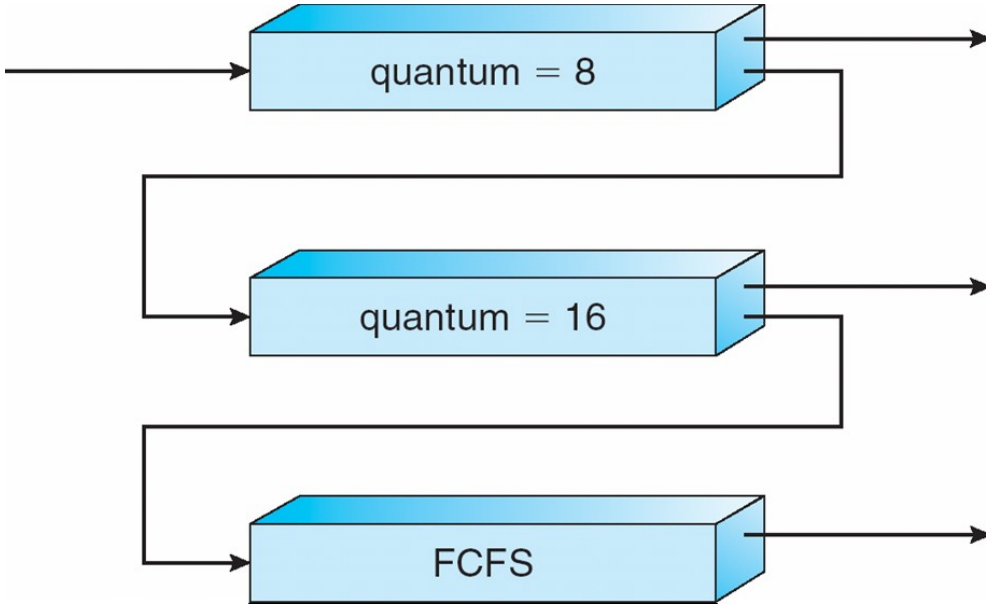
Multilevel Feedback Scheduling

- 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
  - 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

Example of Multilevel Feedback Queue

- Three queues:
  - Q0 – RR with time quantum 8 milliseconds
  - Q1 – RR time quantum 16 milliseconds
  - Q2 – FCFS Scheduling



Solaris Scheduler

- Combines time slices, priority, and prediction

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Lottery Scheduling

- Give processes lottery tickets
- Choose ticket at random and allow process holding the ticket to get the resource
- Hold a lottery at periodic intervals
- Properties
  - Chance of winning proportional to number of tickets held (highly responsive)
  - Cooperating processes may exchange tickets
  - Fair-share scheduling easily implemented by allocating tickets to users and dividing tickets among child processes

Multiprocessor Scheduling

- Given a set of runnable processes, and a set of CPUs, assign processes to CPUs

- Same considerations:
  - response time, fairness, throughput, …

- But also, new considerations:
  - ready queue implementation
  - load balancing
  - affinity
  - resource contention

Ready Queue Implementation

- Option 1: Single Shared Ready Queue (among CPUs)
  - Scheduling events occur per-CPU
    - Local timer interrupt
    - Currently executing thread blocks or yields
  - Scheduling code on any CPU needs to access the shared queue
    - Synchronization is needed.

- Option 2: Per-CPU Ready Queue
  - Scheduling code accesses local queue
  - Load balancing:
    - Infrequent synchronization
  - Per-CPU variables should lie on separate cache-lines

Load balancing

- Keep ready queue sizes balanced across CPUs
  - Main goal: one cpu should not idle while others have processes waiting in their queues
  - Secondary: scheduling overhead may increase w.r.t queue length
- **Push model**: kernel daemon checks queue lengths periodically, moves threads to balance
- **Pull model**: CPU notices its queue is empty (or shorter than a threshold) and steals threads from other queues
- or both

Affinity

- As threads run, state accumulates in CPU cache
- Repeated scheduling on same CPU can often reuse this state
- Scheduling on different CPU requires reloading new cache
  - And possibly invalidating old cache
- Try to keep thread on same CPU it used last

Contention-aware Scheduling I

- Hardware resource sharing/contention in multi-processors
  - SMP processors share memory bus bandwidths
  - Multi-core processors share cache
  - SMT (hyper-thread) processors share a lot more

- An example: on an SMP machine
  - a web server benchmark delivers around 6300 reqs/sec on one processor, but only around 9500 reqs/sec on an SMP with 4 processors

- Threads load data into cache and we can expect multiple threads to trash each others' state as they run on one same CPU
  - Can try to detect cache needs and schedule threads that can share nicely on same CPU

SMP-SMT Multiprocessor

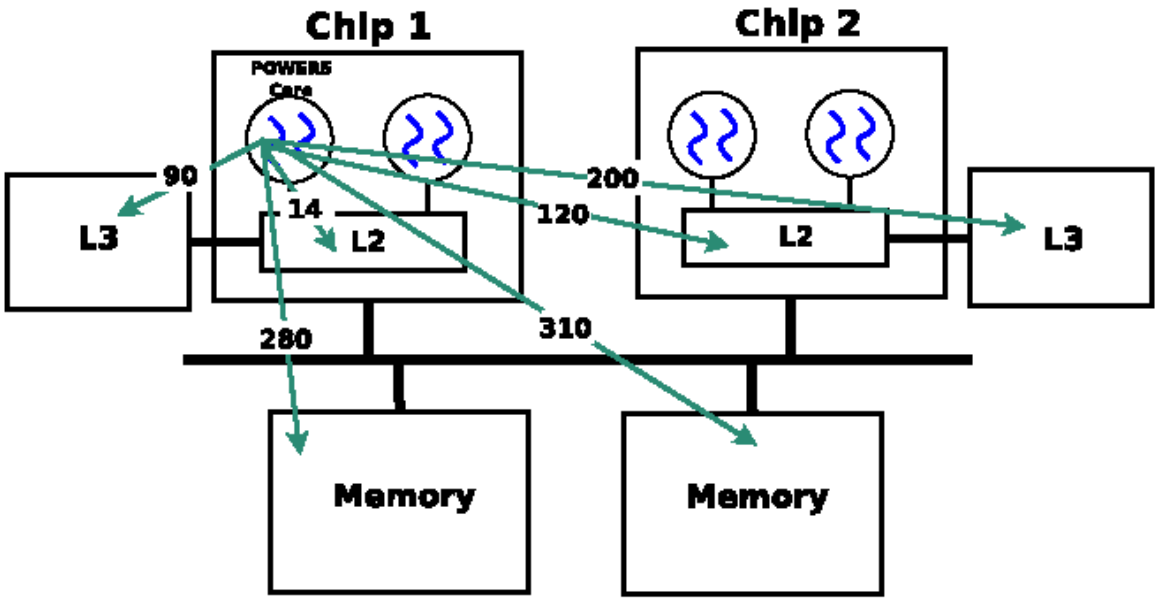


Image from http://www.eecg.toronto.edu/~tamda/papers/threadclustering.pdf

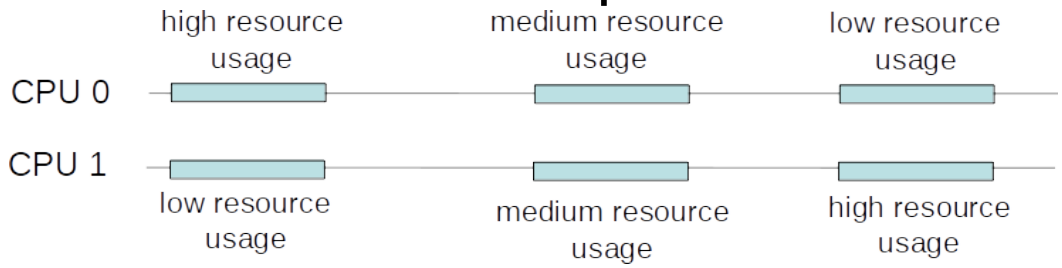
Contention-aware Scheduling II

- Contention-reduction scheduling by co-scheduling tasks with complementary resource needs
  - e.g. a computation-heavy task and a memory access-heavy task
  - e.g. several threads with small cache footprints may all be able to keep data in cache at same time
  - e.g. threads with no locality might as well execute on same CPU since almost always miss in cache anyway

Contention-aware Scheduling III

- What if contention on a resource is unavoidable?
- Two evils of contention
  - high contention ⇒ performance slowdown
  - fluctuating contention ⇒ uneven application progress over the same amount of time ⇒ poor fairness

- [Zhang et al. HotOS2007] Scheduling so that:
  - very high contention is avoided
  - the resource contention is kept stable



Parallel Job Scheduling

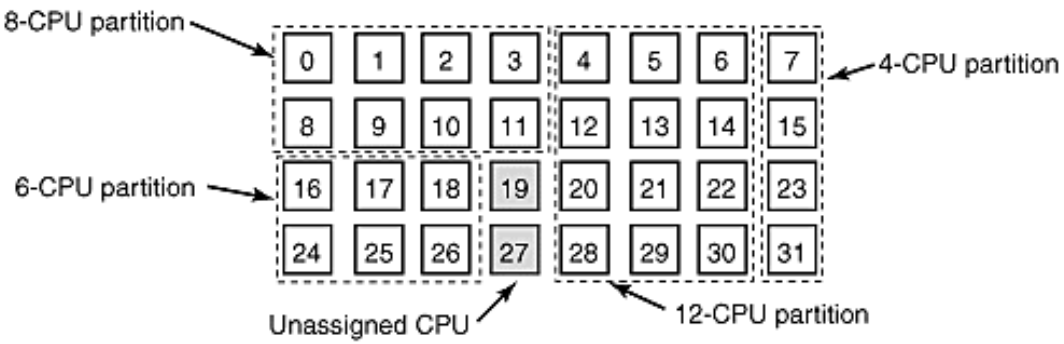
- "Job" is a collection of processes/threads that cooperate to solve some problem (or provide some service)
- How the components of the job are scheduled has a major effect on performance
- Threads in a parallel job are not independent
  - Scheduling them as if they were leads to performance problems
  - Want scheduler to be aware of dependence

Parallel Job Scheduling

- Threads in a processes are not independent
  - Synchronize over shared data
    - Deschedule lock holder, other threads in job may not get far
  - Cause/effect relationships (e.g. producer-consumer problem)
    - Consumer is waiting for data on queue, but producer is not running
  - Synchronizing phases of execution (barriers)
    - Entire job proceeds at pace of slowest thread
- Knowing threads are related, schedule all at same time
  - Space Sharing
  - Time Sharing with *Gang Scheduling*

Space Sharing

- Divide CPUs into groups and assign jobs to dedicated set of CPUs



- Pros
  - Reduce context switch overhead (no involuntary preemption)
  - Strong affinity
  - All runnable threads execute at same time
- Cons
  - CPUs in one partition may be idle while we have multiple jobs waiting to run
  - Difficult to deal with dynamically changing job sizes

Time Sharing

- Similar to uni-processor scheduling – a queue of ready tasks, a task is dequeued and executed when a processor is available

- Gang/Cohort scheduling
  - utilize all CPUs for one parallel/concurrent application at a time

Time slot	CPU 0	1	2	3	4	5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

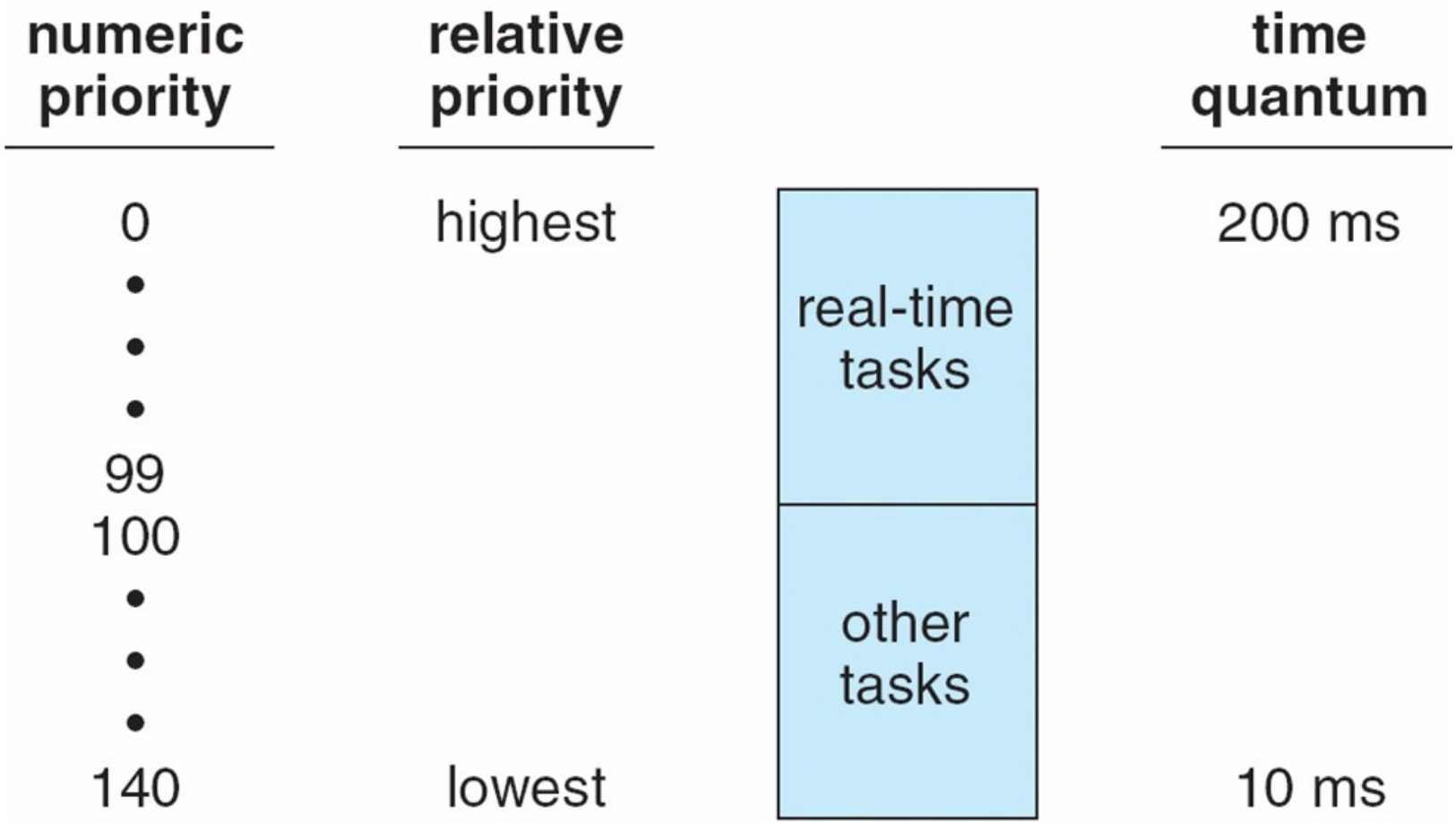
Multiprocessor Scheduling in Linux 2.6

- One ready task queue per processor
  - scheduling within a processor and its ready task queue is similar to single-processor scheduling
- One task tends to stay in one queue
  - for cache affinity
- Tasks move around when load is unbalanced
  - e.g., when the length of one queue is less than one quarter of the other
- No native support for gang/cohort scheduling or resource-contention-aware scheduling

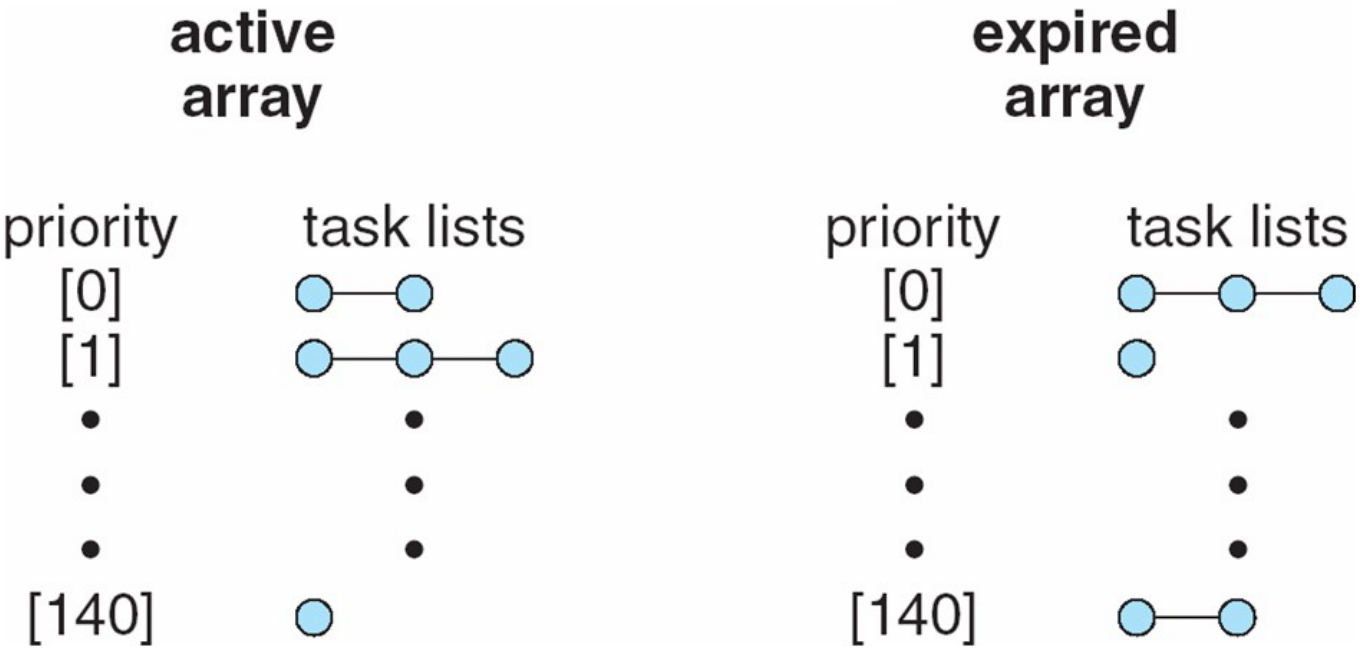
Linux Task Scheduling

- Linux 2.5 and up uses a preemptive, priority-based algorithm with two separate priority ranges:
  - A time-sharing class for fair preemptive scheduling (nice value ranging from 100-140)
  - A real-time class that conforms to POSIX real-time standard (0-99)
- Numerically lower values indicate higher priority
- Higher-priority tasks get longer time quanta (200-10 ms)
- Ready queue indexed by priority and contains two priority arrays – *active* and *expired*
- Choose task with highest priority on active array; switch active and expired arrays when active is empty - in O(1)

Priorities and Time-slice length



List of Tasks Indexed According to Priorities



Real-Time Scheduling

- Deadline
  - Time to react
  - keep pace with a frequent event

- **Hard real-time systems** – required to complete a critical task within a guaranteed amount of time

- **Soft real-time computing** – requires that critical processes receive priority over less fortunate ones

- Earliest Deadline First (EDF)

Disclaimer

Parts of the lecture slides were derived from those by Kai Shen, Willy Zwaenepoel, Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Andrew S. Tanenbaum, Angela Demke Brown, and Gary Nutt. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester.