# Linux Scheduling

Prof. P.H.Dave
DDU, Nadiad;
-
Prof. H.B.Dave
eInfochips, Ahmedabad.

March, 2015

# Default Linux

- Linux is originally designed to be a general purpose OS (GPOS)
  - High functionality
  - Average or peak performance is usually important for GPOS
  - Latency varies widely

# Current Linux and PREEMT_RT extension

- Current mainline Linux has a variety of real-time features
  - Certain level of deterministic behavior by appropriate configurations and tuning
- PREEMPT_RT patchset reduce maximum latency much more
  - Many real-time features in mainline are derived from PREEMPT_RT

# RT Features in Linux

- Mainline
  - Fixed-priority scheduling
  - Kernel preemption
  - High resolution timer (hrtimer)
  - IRQ thread
  - Others
- Out of tree
  - PREEMPT_RT patchset

# Three Types of Schedulers in Linux

- Linux has three types of schedulers
  - Completely Fair Scheduler (CFS)
    - Policy: SCHED_OTHER(Default), _BATCH, _IDLE
    - Task has dynamic-priority based on time slice (non-deterministic)
  - Real-time scheduler
    - Policy: SCHED_FIFO, _RR
    - Task has fixed-priority (deterministic)
  - Deadline scheduler
    - Policy: SCHED_DEADLINE
    - Merged in 3.14

# Linux: Non-Real-Time Scheduling

- Linux 2.6 uses a new scheduler the O(1) scheduler
- Time to select the appropriate process and assign it to a processor is constant
- Regardless of the load on the system or number of processors
- Later kernels use Completely Fair Scheduler (CFS)

# Linux Scheduling

- 3 scheduling classes
  - SCHED_FIFO and SCHED_RR are real-time classes
  - SCHED_OTHER is for the rest
- 140 Priority levels
  - 1-100 : RT priority
  - 101-140 : User task priorities
- Three different scheduling policies
  - One for normal tasks
  - Two for Real time tasks

- Pre-emptive, priority based scheduling.
- When a process with higher real-time priority (`rt_priority`) wishes to run, all other processes with lower real-time priority are thrust aside.
- In SCHED_FIFO, a process runs until it relinquishes control or another with higher real-time priority wishes to run.
- SCHED_RR process, in addition to this, is also interrupted when its time slice expires or there are processes of same real-time priority (RR between processes of this class)
- SCHED_OTHER is also round-robin, with lower time slice

- SCHED_OTHER: Normal tasks
  - Each task assigned a “Nice” value
  - Static priority = 120 + Nice
    - Nice value between -20 and +19
  - Assigned a time slice
  - Tasks at the same priority are round-robined
    - Ensures Priority + Fairness

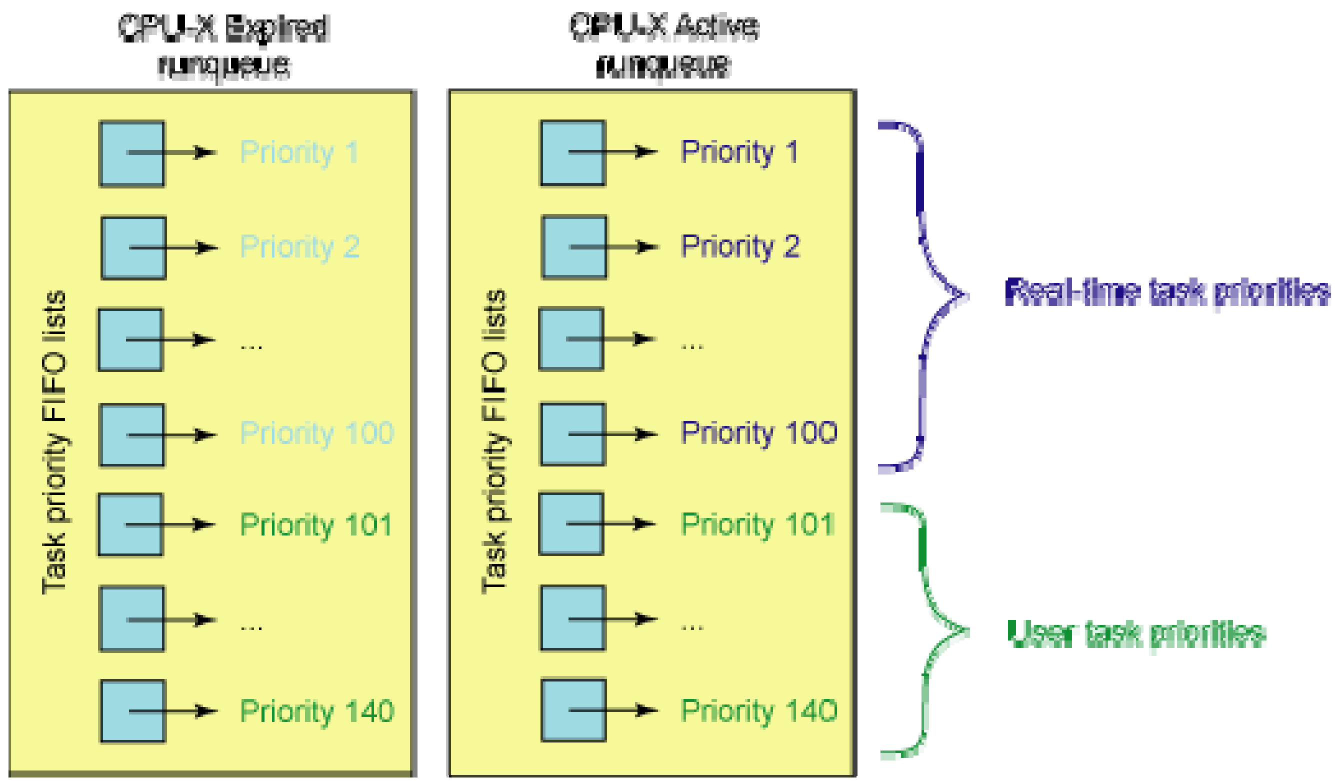
# Basic Philosophies

- Priority is the primary scheduling mechanism
- Priority is *dynamically adjusted* at run time
  - Processes denied access to CPU get increased
  - Processes running a long time get decreased
- Try to distinguish interactive processes from non-interactive
  - Bonus or penalty reflecting whether I/O or compute bound
- Use large quanta for important processes
  - Modify quanta based on CPU use
- Associate processes to CPUs
- Do everything in O(1) time

# The Runqueue

- 140 separate queues, one for each priority level
- Actually, two sets, active and expired
- Priorities 0-99 for real-time processes
- Priorities 100-139 for normal processes; value set via nice()/setpriority() system calls

# Linux 2.6 scheduler runqueue structure



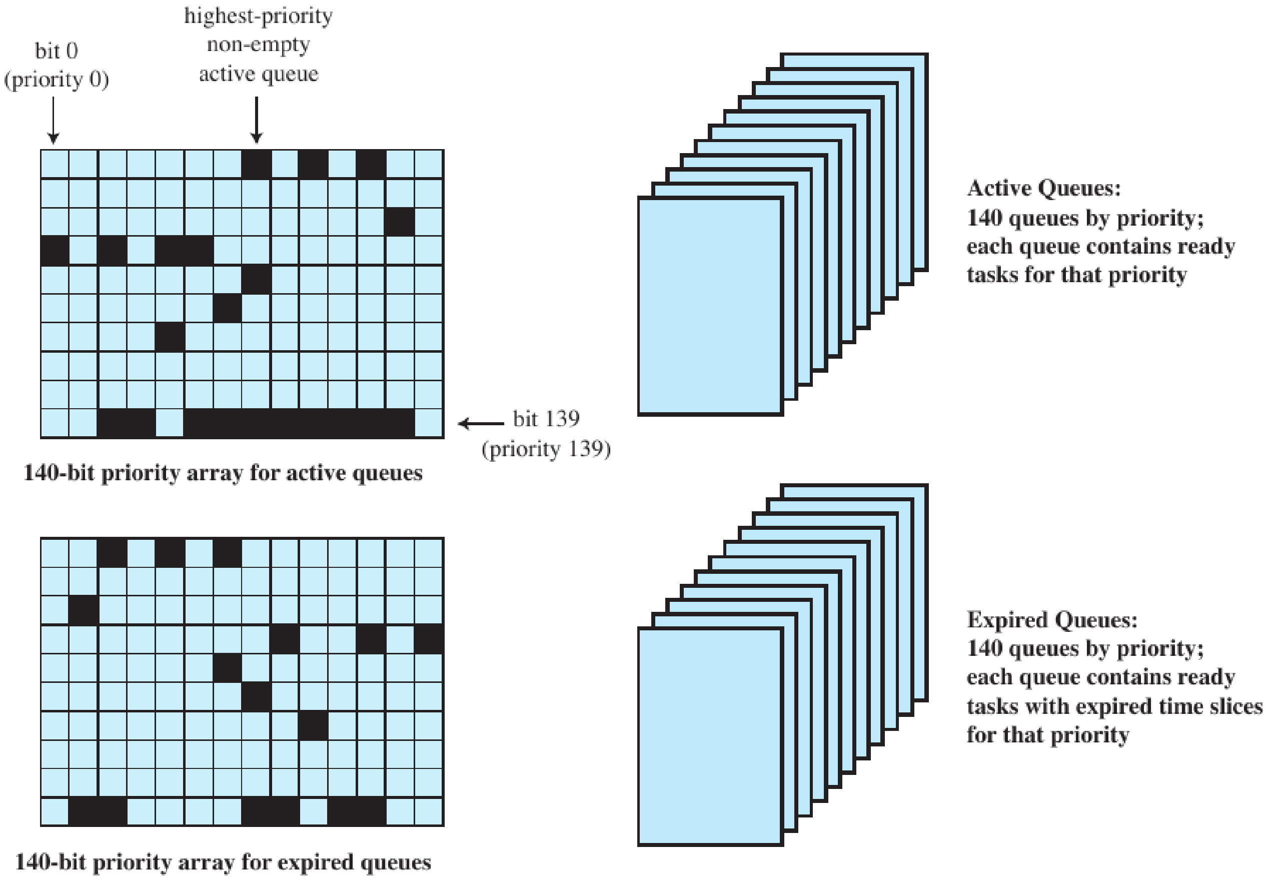
# Scheduler Runqueue

- A scheduler runqueue is a list of tasks that are runnable on a particular CPU.
- A *rq* structure maintains a linked list of those tasks.
- The runqueues are maintained as an array *runqueues*, indexed by the CPU number.
- The *rq* keeps a reference to its idle task
  - The idle task for a CPU is never on the scheduler runqueue for that CPU (it's always the last choice)
- Access to a runqueue is serialized by acquiring and releasing *rq->lock*

# Basic Scheduling Algorithm

- Find the highest-priority queue with a runnable process
- Find the first process on that queue
- Calculate its quantum size
- Let it run
- When its time is up, put it on the expired list
  - Recalculate priority first
- Repeat

# Linux Scheduling Data Structures for Each Processor



# The Highest Priority Process

- There is a bit map indicating which queues have processes that are ready to run
- Find the first bit that’s set:
  - 140 queues → 5 integers
  - Only a few compares to find the first that is non-zero
  - Hardware instruction to find the first 1-bit
    - bsfl on Intel
  - Time depends on the number of priority levels, not the number of processes

# Scheduling Components

- Static Priority
- Sleep Average
- Bonus
- Dynamic Priority
- Interactivity Status

# Static Priority

- Each task has a static priority that is set based upon the nice value specified by the task.
  - *static_prio* in *task_struct*
  - Value between 0 and 139 (between 100 and 139 for normal processes)
- Each task has a dynamic priority that is set based upon a number of factors
  - tries to increase priority of interactive jobs

# Sleep Average

- Interactivity heuristic: sleep ratio
  - Mostly sleeping: I/O bound
  - Mostly running: CPU bound
- Sleep ratio approximation
  - *sleep_avg* in the *task_struct*
  - Range: 0 .. *MAX_SLEEP_AVG*
- When process wakes up (is made runnable), *recalc_task_prio* adds in how many ticks it was sleeping (blocked), up to some maximum value (*MAX_SLEEP_AVG*)
- When process is switched out, *schedule* subtracts the number of ticks that a task actually ran (without blocking)
- *sleep_avg* scaled to a bonus vale

# Average Sleep Time and Bonus Values

| Average sleep time | Bonus |
|---|---|
| >= 0 but < 100 ms | 0 |
| >= 100 ms but < 200 ms | 1 |
| >= 200 ms but < 300 ms | 2 |
| >= 300 ms but < 400 ms | 3 |
| >= 400 ms but < 500 ms | 4 |
| >= 500 ms but < 600 ms | 5 |
| >= 600 ms but < 700 ms | 6 |
| >= 700 ms but < 800 ms | 7 |
| >= 800 ms but < 900 ms | 8 |
| >= 900 ms but < 1000 ms | 9 |
| 1 second | 10 |

# Bonus and Dynamic Priority

- Dynamic priority (*prio* in *task_struct*) is calculated in from static priority and bonus
  - = max (100, min( static_priority – bonus + 5, 139) )

# Calculating Time Slices

- *time_slice* in the *task_struct*
- Calculate Quantum where
  - If (SP < 120): Quantum = (140 − SP) × 20
  - if (SP >= 120): Quantum = (140 − SP) × 5

  where SP is the *static priority*
- Higher priority process get longer quanta
- Basic idea: important processes should run longer
- Other mechanisms used for quick interactive response

# Nice Value vs. static priority and Quantum

| | Static Priority | NICE | Quantum |
|---|---|---|---|
| High Priority | 100 | -20 | 800 ms |
| | 110 | -10 | 600 ms |
| | 120 | 0 | 100 ms |
| | 120 | +10 | 50 ms |
| Low Priority | 139 | +19 | 5 ms |

$$\text{Quantum} = \begin{cases} (140 - \text{SP}) \times 20 & \text{if SP} < 120 \\ (140 - \text{SP}) \times 5 & \text{if SP} \geq 120 \end{cases}$$

# Interactive Processes

- A process is considered interactive if

  bonus − 5 >= (Static Priority / 4) − 28

  - (Static Priority / 4) − 28 = interactive delta
- Low-priority processes have a hard time becoming interactive:
  - A high static priority (100) becomes interactive when its average sleep time is greater than 200 ms
  - A default static priority process becomes interactive when its sleep time is greater than 700 ms
  - Lowest priority (139) can never become interactive
- The higher the bonus the task is getting and the higher its static priority, the more likely it is to be considered interactive.

# Using Quanta

- At every time tick (in *scheduler_tick*) , decrement the quantum of the current running process (*time_slice)*
- If the time goes to zero, the process is done
- Check interactive status:
  - If non-interactive, put it aside on the *expired* list
  - If interactive, put it at the end of the *active* list
- Exceptions: don't put on *active* list if:
  - If higher-priority process is on *expired* list
  - If expired task has been waiting more than *STARVATION_LIMIT*
- If there's nothing else at that priority, it will run again immediately
- Of course, by running so much, its bonus will go down, and so will its priority and its interactive status

# Avoiding Starvation

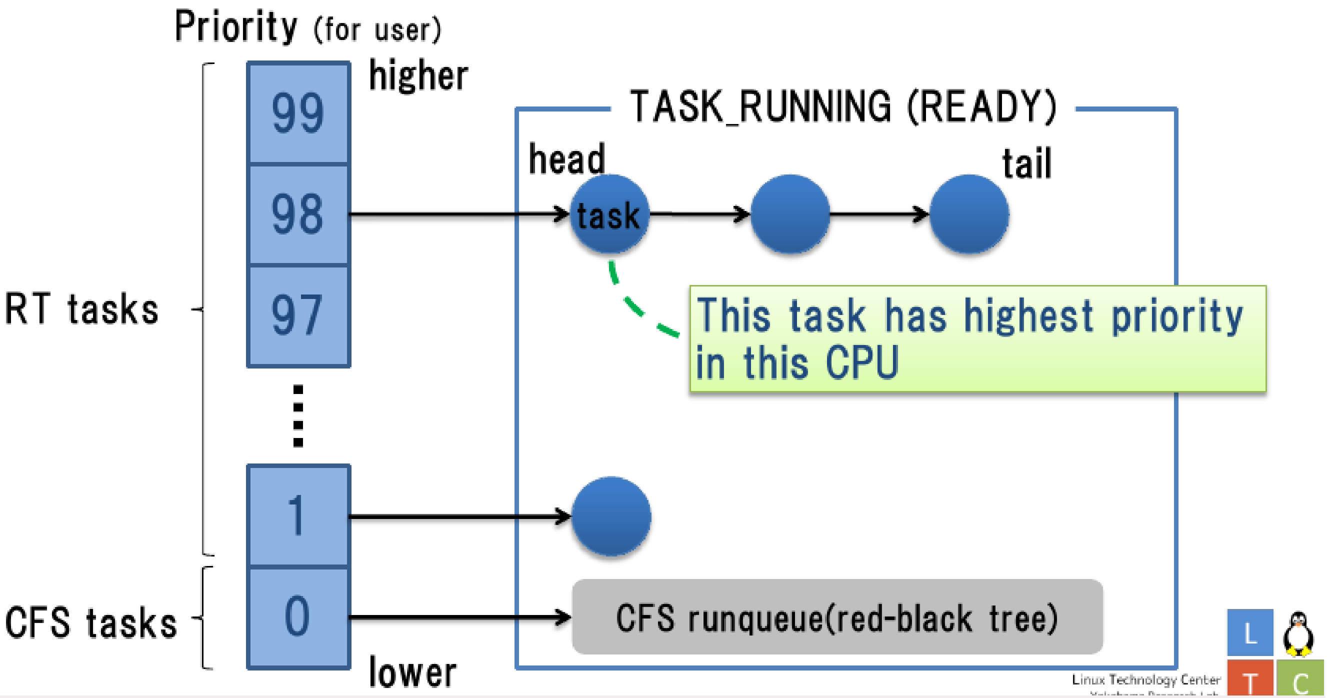
- The system only runs processes from active queues, and puts them on expired queues when they use up their quanta
- When a priority level of the active queue is empty, the scheduler looks for the next-highest priority queue
- After running all of the active queues, the active and expired queues are swapped
- There are pointers to the current arrays; at the end of a cycle, the pointers are switched

# Linux Real-Time Scheduling

- Scheduling classes:
  - SCHED_FIFO: First-in-first-out real-time threads
  - SCHED_RR: Round-robin real-time threads
  - SCHED_OTHER: Other, non-real-time threads, traditional Unix scheduling
- Within each class multiple priorities may be used

# RT and CFS Task Queues

- There are per-cpu runqueues in Linux
  - RT tasks are always dispatched prior to CFS tasks



# Linux is More Efficient

- Processes are touched only when they start or stop running
- That’s when we recalculate priorities, bonuses, quanta, and interactive status
- There are no loops over all processes or even over all runnable processes

# Real-Time Scheduling

- Linux has soft real-time scheduling
  - No hard real-time guarantees
- All real-time processes are higher priority than any conventional processes
- Processes with priorities [0, 99] are real-time
  - saved in *rt_priority* in the *task_struct*
  - scheduling priority of a real time task is: 99 - *rt_priority*
- Process can be converted to real-time via *sched_setscheduler* system call

# Real-Time Policies

- First-in, first-out: SCHED_FIFO
  - Static priority
  - Process is only preempted for a higher-priority process
  - No time quanta; it runs until it blocks or yields voluntarily
  - RR within same priority level
- Round-robin: SCHED_RR
  - As above but with a time quanta (800 ms)
- Normal processes have SCHED_OTHER scheduling policy

# Example of Linux Real-Time Scheduling

| | |
|---|---|
| A | minimum |
| B | middle |
| C | middle |
| D | maximum |

(a) Relative thread priorities

D → B → C → A →

(b) Flow with FIFO scheduling

D → B → C → B → C → A →

(c) Flow with RR scheduling