



# Scheduling in Kernel 2.6

---

Mahendra M

*Mahendra\_M@infosys.com*

<http://www.infosys.com>



# Agenda

---

- \* Scheduling
  - o Process Scheduling
    - O(1) scheduler – design, performance
    - Pre-emption
  - o I/O Scheduling
    - Dealine I/O scheduler
    - Anticipatory I/O scheduler
    - Comparison



## Considerations in Scheduler design

---

- \* Fairness
  - o Prevent starvation of tasks
- \* Scheduling latency
  - o Reduction in delay between a task waking up and actually running
  - o Time taken for the scheduler decisions
- \* Interrupt latency
  - o Delay in processing h/w interrupts
- \* Scheduler decisions



# Process Scheduler

---

## \* Goals

- o Good interactive performance during high load
- o Fairness
- o Priorities
- o SMP efficiency
- o SMP affinity
  - Issues of random bouncing taken care
  - No more 'timeslice squeeze'
- o RT Scheduling

( From /usr/src/linux-2.6.x/Documentation/sched-design.txt )



## Goals ( contd .... )

---

- \* Full  $O(1)$  scheduling
  - o Great shift away from  $O(n)$  scheduler
- \* Perfect SMP scalability
  - o Per CPU runqueues and locks
  - o No global lock/runqueue
  - o All operations like wakeup, schedule, context-switch etc. are in parallel
- \* Batch scheduling ( bigger timeslices, RR )
- \* No scheduling storms
- \*  $O(1)$  RT scheduling



## Design

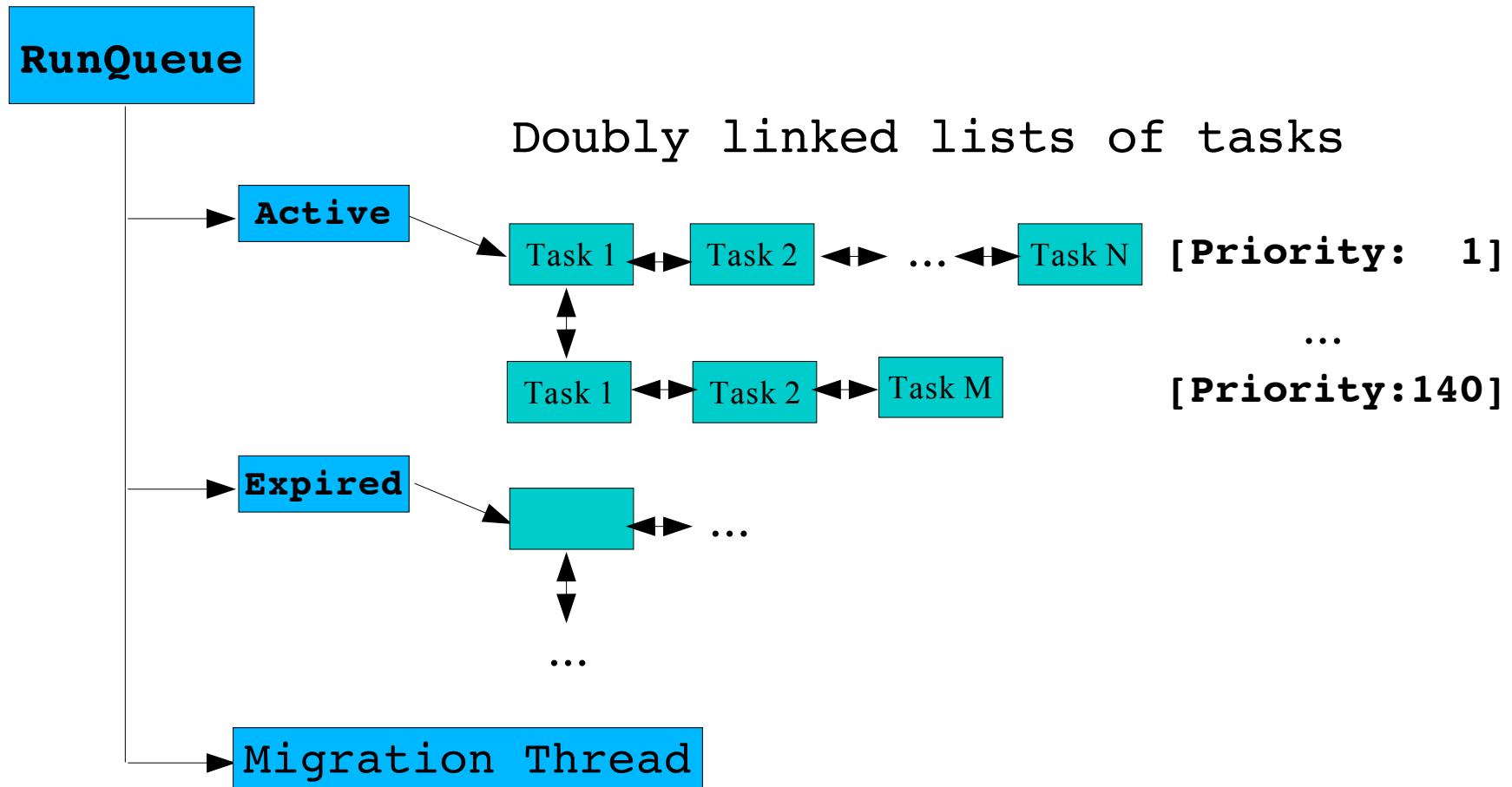
---

- \* 140 priority levels
  - o The lower the value, higher is the priority
  - o Eg : Priority level 110 will have a higher priority than 130.
- \* Two priority-ordered 'priority-arrays' per CPU
  - o 'Active' array : tasks which have timeslices left
  - o 'Expired' Array : tasks which have run
  - o Both accessed through pointers from per-CPU runqueue
- \* They are switched via a simple pointer swap



# Per-CPU runqueue

Each CPU on the system has it's own RunQueue





## O(1) Algorithm ( Constant time algorithm )

---

- \* The highest priority level, with at-least ONE task in it, is selected
  - o This takes a fixed time ( say  $t1$  )
- \* The first task ( head of the doubly-linked list ) in this priority level is allowed to run
  - o This takes a fixed time ( say  $t2$  )
- \* Total time taken for selecting a new process is
  - o  $t = t1 + t2$  ( Fixed )
- \* The time taken for selecting a new process will be fixed ( constant time irrespective of number of tasks )
- \* Deterministic algorithm !!





## 2.6 v/s 2.4

---

Kernel 2.4 had

- \* A Global runqueue.
  - o All CPUs had to wait for other CPUs to finish execution.
- \* An  $O(n)$  scheduler.
  - o In 2.4, the scheduler used to go through the entire “global runqueue” to determine the next task to be run.
  - o This was an  $O(n)$  algorithm where 'n' is the number of processes. The time taken was proportional to the number of active processes in the system.
- \* This lead to large performance hits during heavy workloads.



## Scheduling policies in 2.6

---

- \* 140 Priority levels
  - o 1-100 : RT prio (  $MAX\_RT\_PRIO = 100$  )
  - o 101-140 : User task Prio (  $MAX\_PRIO = 140$  )
- \* Three different scheduling policies
  - o One for normal tasks
  - o Two for Real time tasks
- \* Normal tasks
  - o Each task assigned a “Nice” value
  - o  $PRIO = MAX\_RT\_PRIO + NICE + 20$
  - o Assigned a time slice
  - o Tasks at the same prio are round-robined.
    - Ensures Priority + Fairness



## Policies ( contd ... )

---

- \* RT tasks ( Static priority )
  - o FIFO RT tasks
    - Run until they relinquish the CPU voluntarily
    - Priority levels maintained
    - Not pre-empted !!
  - o RR RT tasks
    - Assigned a timeslice and run till the timeslice is exhausted.
    - Once all RR tasks of a given prio level exhaust their timeslices, their timeslices are refilled and they continue running.
    - Prio levels are maintained
- \* The above can be unfair !! - Sane design expected !!



## Interactivity estimator

---

- \* Dynamically scales a tasks priority based on it's interactivity
- \* Interactive tasks receive a prio bonus [ -5 ]
  - o Hence a larger timeslice
- \* CPU bound tasks receive a prio penalty [ +5 ]
- \* Interactivity estimated using a running sleep average.
  - o Interactive tasks are I/O bound. They wait for events to occur.
  - o Sleeping tasks are I/O bound or interactive !!
  - o Actual bonus/penalty is determined by comparing the sleep average against a constant maximum sleep average.
- \* Does not apply to RT tasks



## Recalculation of priorities

---

When a task finishes it's timeslice :

- \* It's interactivity is estimated
- \* Interactive tasks can be inserted into the 'Active' array again.
- \* Else, priority is recalculated
- \* Inserted into the NEW priority level in the 'Expired' array.



## Re-inserting interactive tasks

---

- \* To avoid delays, interactive tasks may be re-inserted into the 'active' array after their timeslice has expired.
- \* Done only if tasks in the 'expired' array have run recently.
  - o Done to prevent starvation of tasks
- \* Decision to re-insert depends on the task's priority level.



## Finegrained timeslice distribution

---

- \* Priority is recalculated only after expiring a timeslice.
- \* Interactive tasks may become non-interactive during their LARGE timeslices, thus starving other processes.
- \* To prevent this, time-slices are divided into chunks of 20ms.
- \* A task of equal priority may preempt the running task every 20ms.
- \* The preempted task is requeued and is round-robin in its priority level.
- \* Also, priority recalculation happens every 20ms.



## For programmers

---

From `/usr/src/linux-2.6.x/kernel/sched.c`

\* `void schedule()`

- o The main scheduling function.
- o Upon return, the highest priority process will be active

\* Data

- o `struct runqueue()`
  - The main per-CPU runqueue data structure
- o `struct task_struct()`
  - The main per-process data structure





## For programmers ( contd.... )

---

### Process Control methods

- \* `void set_user_nice ( ... )`
  - o Sets the nice value of task p to given value
- \* `int setscheduler( ... )`
  - o Sets the scheduling policy and parameters for a given pid
- \* `rt_task( pid )`
  - o Returns true if pid is real-time, false if not.
- \* `yield( )`
  - o Place the current process at the end of the runqueue and call `schedule()`.



## Handling SMP ( multiple CPUs )

---

- \* A run-queue per CPU
  - o Each CPU handles it's own processes and do not have to wait till other CPU tasks finish their timeslices.
- \* A 'migration' thread runs for every CPU.
- \* `void load_balance( )`
  - o This function call attempts to pull tasks from one CPU to another to balance CPU usage if needed.
  - o Called
    - Explicitly if runqueues are inbalanced
    - Periodically by the timer tick
- \* Processes can be made affine to a particular CPU.



---

## I/O Schedulers



## I/O Scheduler

---

- \* Kernel 2.4 I/O scheduler
  - o One request queue, which is sorted seek-wise.
  - o Reduces seek-time for the disk head.
  - o This type of sorting can lead to starvation of requests far away from the current seek position.
  - o Write – starving – reads.
    - Write requests are asynchronous and non-blocking
      - Most apps are not bothered about write commits
    - Read requests are blocking, as apps need the data to continue ( synchronous )
    - Read requests have to be prioritised over write requests to improve responsiveness.



## Deadline I/O scheduler

---

- \* Assigns tasks an expiration time
- \* Alongwith a queue sorted seek-wise, two additional queues are implemented.
  - o FIFO read queue with a deadline of 500ms
  - o FIFO write queue with a deadline of 5 seconds.
- \* A request is submitted to the sorted queue and the appropriate deadline queue ( at the tail of the queue )
- \* Requests are scheduled from the sorted queue.
- \* If a request in the FIFO queue expires then the scheduler begins dispatching from FIFO queues.



## Deadline I/O scheduler ( contd... )

---

- \* Ensures that seeks are minimised
- \* At the same time, makes sure that requests are not starved.
- \* Read requests are given a better deadline than write requests ( 10 times )
  - o Interactivity is improved.
  - o Applications are not blocked by read requests.
  - o Improved performance in case of dependent read requests.
    - Eg : `$ cat *`
- \* Can result in a seek-storm, because the sorted queue can get neglected !!



## Anticipatory I/O scheduler

---

- \* Same algorithm, but anticipates future read requests in case of dependent read requests.
- \* After a read-request is completed
  - o Doesn't proceed to the next request
  - o Waits for a few milliseconds ( 6ms ) to see if the application submits another read request.
    - Eg : an app reading an Image file in 1024byte buffers.
  - o If a new read request happens then the scheduler process this request.
- \* This small wait prevents a lot of seek operations.
  - o If the app doesn't issue any read request the waiting period is wasted !!



## Performance Comparison

---

- \* Read operations
  - o Deadline I/O
    - Comparable during streaming writes.
    - Performs 10 times better during streaming reads.
  - o Anticipatory I/O ( w.r.t 2.4 )
    - 10 times better during streaming writes
    - 100 times better during streaming reads.
- \* Write operations
  - o Perform almost same as 2.4 (2.4 may be better)
  - o Deadline I/O scheduler performs slightly better than Anticipatory I/O scheduler.





## Resources

---

- \* Kernel documentation
  - o /usr/src/linux-2.6.x/Documentation/
    - sched-design.txt, sched-coding.txt
    - preempt-locking.txt
    - as-iosched.txt
- \* Scheduler
  - <http://kerneltrap.org/node/view/464>
  - <http://kerneltrap.org/node/view/657>
  - <http://www.arstechnica.com/etc/linux/index.html>
- \* <http://kerneltrap.org>
- \* The Linux Mailing List Archives



# Thankyou

---

- \* Special thanks to
  - o Robert Love & Rusty Russel for their excellent documentation efforts
    - This presentation is based on a paper presented by Robert Love at “LinuxSymposium/2003”
  - o The Linux Kernel Team for commenting the code and giving it a story-book feeling !!
  - o The Infosys OpenSource Community for clarifying most of my doubts.
  - o The Bangalore LUG for giving me a chance to speak at their monthly meet !!