Stochastic Priority Based Task Scheduler

Muhammad Usman Nadeem Talha Ghaffar

March 5, 2017

Abstract

We implement a Stochastic priority based task scheduler on Linux Kernel 2.6.32.27. It uses process priority to determine CPU timeshare per process. It also takes into account fairness and starvation problem. In the end we run it on a variety of loads and different number of processes to evaluate its accuracy and scalability and validate the algorithm.

1 Algorithm description

Our algorithm is based on a simple traditional Lottery based Scheduler. We named it "NEW POLICY" and will refer to it by the same name in this report as well as in the code.

In Linux kernel every process is assigned a priority from range 0 to MAX_PRIO-1 where max priority is 140. The assignable priority depends on whether the process is RT or a normal process (in which case nice values come into account). Low number is equivalent to high priority.

We assign each process a fixed number of tickets whenever it is assigned to NEW POLICY. A high priority process gets a higher number of tickets and vice versa. With zero being the least number of tickets assigned to any process, highest being 140.

At every scheduling decision the scheduling class calculates the total number of tickets all the processes in its run queue contains. It then generates a random number in that range. All the processes are contained in a kernel linked list. We thought that it was better and simpler to use a list instead of a tree because we need to iterate over it regardless of the order.

The scheduler iterates over the list the process who gets the ticket is chosen. A running total of the number of tickets in each process is kept and we stop and choose the process when runningTotal >= randomNumber. In this case we don't need to sort processes by number of tickets to ensure fairness. Although if we keep a sorted list in descending order our pick_next_task_newpolicy function might be faster but there is the added cost of keeping the list in a sorted order. We do not go into further analysis.

There is absolutely no starvation because there is a non-zero probability of getting a lottery ticket for every process in the run queue. Eventually every process will get its turn. The fairness depends on how random and equally distributed the values by our random number generator are. In the ideal case the "number of ticket for a process" to the ratio of "total number of tickets assigned" will be the probability of that process getting selected to run at the next scheduling decision. This is also equal to the CPU time assigned to each process. So a process with 10 tickets will get ten times more CPU time than a process with 10 tickets. Although this must be mentioned that it is the ideal case described above, in practical cases this might not be the same because of early preemption, which could be by a higher priority process like an RT process, or even due to the unfairness/bias in the random number generator. Another factor is the runtime of the process, results are more like to be closer to the ideal case in long running process.

A process running on NEW POLICY will usually run for a full time slice which is defined using the macro DEF_TIMESLICE as (100 * HZ / 1000), unless preempted. A scheduling decision takes place on preemption or whenever the timeslice expires.

2 Scheduler implementation

2.1 Scheduling class

We implement all the regular scheduling functions that are needed. A list is shown in the code snippet below.

```
// sched_newpolicy.c
const struct sched_class newpolicy_sched_class = {
   .next         = &idle_sched_class,
   .enqueue_task    = enqueue_task_newpolicy,
   .dequeue_task    = dequeue_task_newpolicy,

   .check_preempt_curr = check_preempt_curr_newpolicy,

   .pick_next_task    = pick_next_task_newpolicy,
   .put_prev_task   = put_prev_task_newpolicy,

   .set_curr_task        = set_curr_task_newpolicy,
   .task_tick    = task_tick_newpolicy,
   .yield_task   = yield_task_newpolicy,

   .switched_to    = switched_to_newpolicy,
   .prio_changed    = prio_changed_newpolicy,

#ifdef CONFIG_SMP
   .select_task_rq    = select_task_rq_newpolicy,

#endif
};
```

2.2 Changes in sched.h and sched.c

The major changes are shown in the snippet below, macros and logging logic is not shown to save space. Our run queue is based on a list instead of a tree and we only add one new variable to the task_struct.

```
// sched.h
struct task_struct {
   #ifdef CONFIG_SCHED_NEWPOLICY_POLICY
      unsigned long long numTickets;
   #endif
    .
    .
}

// sched.c
struct NEWPOLICY_rq {
   struct task_struct *task;
   struct list_head NEWPOLICY_list_head;
   atomic_t nr_running;
};
```

2.3 init_newpolicy'

We initialize the list head and set the number of processes running on NEW POLICY to zero.

2.4 enqueue_task

When enqueueing a process in the run queue we have to do four major things:
- The timeslice has to be reset.
- The number of tickets have to be reassigned.
- Item has to be added to the list.
- The nr_running variable has to be incremented.

2.5 dequeue_task

We just have to dequeue the task from the list and decrement the nr_running variable.

2.6 check_preempt_curr

We preempt every time a task comes which is not from the SCHED_IDLE and SCHED_BATCH policies, since they have lower priorities.

2.7 pick_next_task

In this function we have pick the task which should be executed next. If the run queue is not empty we count the total number of tickets kept by all the processes. Then we generate a random number in the range from 1 to totalTickets. Then we iterate over the list to find the task which contains the lucky ticket. Instead for storing a starting and ending range for tickets in the task we just keep numTickets and keep on incrementing a running total to find the lucky task. Random number is generated using the get_random_bytes(...) function in the kernel. If no task is found we return null, so that the kernel can go to the next class.

2.8 put_prev_task

This function is called everytime a task is retired. First we have to check if the task is in a RUNNING state, if it is not then we don't need to put it back or do any other calculations.

Then we reset its timeslice and put it back in the run queue lined list using the enqueue function, if it is not already in the list. It might be removed from the list if it slept or went into some other state like interruptible etc.

2.9 set_curr_task

This function accounts for a task changing its policy or group. So if a task came from some other policy we need to assign it some tickets.

2.10 task_tick

On every timer tick we need to decrement the timeslice. If it becomes 0 we need to reschedule it by marking it using the resched_task(task) function.

If there is no other task in the NEWPOLICY_rq then we just need to reset the timeslice and there is no need to reschedule.

2.11 yield_task

This function usually dequeues and enqueues again but we don't need to do anything because our position in the run queue does not matter.

2.12 switched_to and prio_changed

We just reassign the task a number of tickets depending on its new priority.

2.13 select_task_rq

Return CPU ID in case CONFIG_SMP is enabled in the config.

3 Evaluation

3.1 Validation

We ran a large number of tasks sequentially to show that our scheduling algorithm does not crash. Figure 2 shows the result.

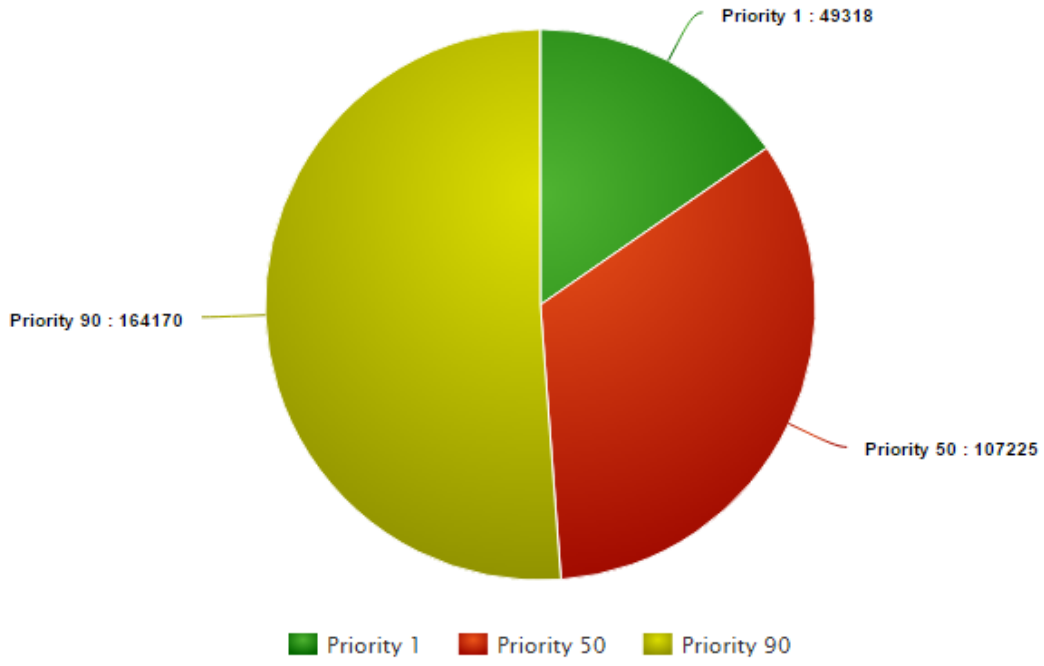


Figure 1: Number of times task is schedules.

3.2 Scalability

We ran a large number of tasks in parallel to further validate our implementation and show that our algorithm scales with the number of processes i.e. the run time even with a large number of concurrently running processes remains linear.

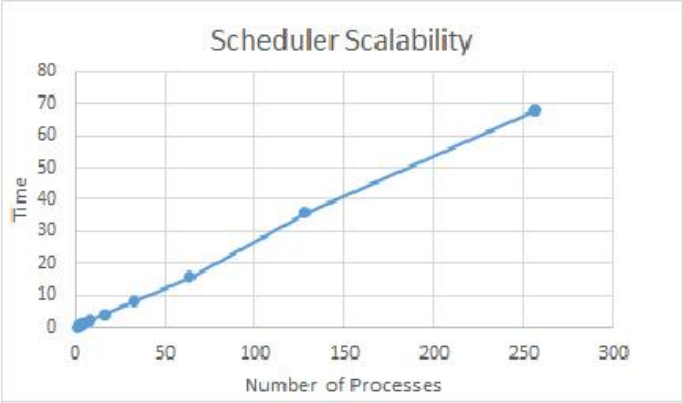
The tasks had a long computation in them and so that they had plenty of time to do context switches and there was also plenty of opportunity to be preempted any higher priority tasks. The results show that the critical functions in our algorithm e.g. pick_next_task_newpolicy, put_prev_task_newpolicy and task_tick_newpolicy etc scale well.

The table below shows the raw data for CONCURRENTLY RUNNING long processes. Please note that the ideal time is extrapolated using the runtime of a single process, since that is low there is a high error rate for more number of processes.

```
Long  |  Iterations=3000000000
--------------------------------
Num     Time       Ideal Time extrapolated using the runtime of a single process
1       7.423      7.423
2       14.671     14.846
4       29.338     29.692
8       61.505     59.384
16      120.917    118.768
32      242.71     237.536
```

3.3 Concordance to the theoretical algorithm

We tested using long running processes (infinite loop) with different priorities: 1 50 and 99. (non inverted - where low number is low priority) We ran them for 30 seconds and counted the number of times pick_next_task_newpolicy was called with each process pid. Figure 1 shows the results. It can clearly be seen that a process with higher priority runs more number of times. In the raw data we could also see interleaved execution showing that preemption and context switches worked.



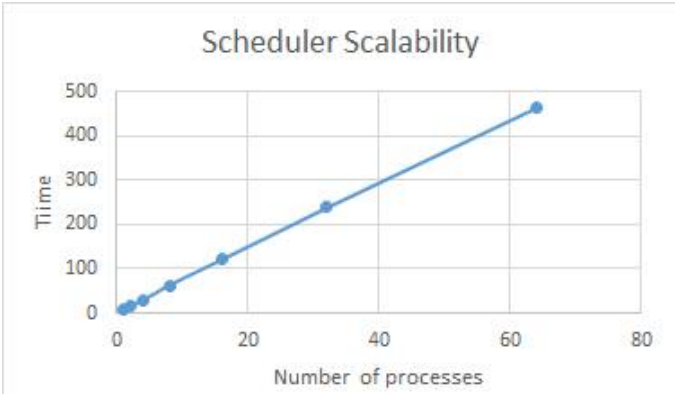


Figure 2: On the left short tasks. On the right long-running tasks.

4 Conclusion

4.1 How to add Citations and a References List

We implement a Stochastic priority based task scheduler on Linux Kernel 2.6.32.27. It uses process priority to determine CPU timeshare per process. It also takes into account fairness and starvation problem in the sense that no process starves and every process gets time share according to priority. In the end we run it on a variety of loads and different number of processes to evaluate its accuracy and scalability and validate the algorithm. We find that the algorithm scales well with a large number of processes and has little core-functionality overhead of only around 2%.