Heap Based Priority Scheduler

121025

121027

121059

# What is Priority Scheduler?

A scheduler, basically, performs the task of scheduling the processes in an operating system. It decides when a process is to be executed and for what time. There are many types of scheduling algorithms in OS. Some of them are, FCFS (first come first serve), SJF (shortest job first), Round Robin and the Priority Based scheduler.

In priority based scheduling, a priority is assigned to each and every process that comes under scheduling. Every process has a priority value based upon which it is processed in the queue. A queue has the FIFO (first in first out) characteristic but in the priority based scheduling, this property is violated because in this priority queue, the process which has the highest priority comes out first though it may not be the first one in the queue. Hence the scheduler maintains the priority and schedules the processes based on the priority.

For e.g. a University has a single printer in the admin office and everyone has access over it through a network. Everyone can print pages from that printer. The page requested to be printed by the Head of Academics should be printed first and then, by the Professors, then by the PG students and the lastly by the UGs. This problem can be solved by priority scheduling.

# Present Priority Scheduling Method:

An “ideal, precise, multitasking CPU” is a hardware CPU that can run multiple processes at the same time , giving each process an equal share of processor power important here is to note that we are not talking about time here.

Ideally we can explain this as ,if a single process is running, it would receive 100% of the processor's power. With two processes, each would have exactly 50% of the physical power (in parallel). Similarly, with four processes running, each would get precisely 25% of physical CPU power in parallel and so on. Therefore, this CPU would be “fair” to all the tasks running in the system .

But we know that there is no such case as ideal in real world, but the CFS we try to implement this in real processor.

On an actual real-world processor, only one task can be allocated to a CPU at a particular time. Therefore, all other tasks wait during this period. So, while the currently running task gets 100% of the CPU power, all other tasks get 0% of the CPU power. This is obviously not fair.

The CFS tries to eliminate this unfairness from the system. The CFS tries to keep track of the fair share of the CPU that would have been available to each process in the system. So, CFS runs a fair clock at a fraction of real CPU clock speed. The fair clock's rate of increase is calculated by dividing the wall time (in nanoseconds) by the total number of processes waiting. The resulting value is the amount of CPU time to which each process is entitled.

This is the basic overview of the scheduler

DATA STRUCTURE -red black tree

CODE IMPLEMENTATION:

we have implemented the basic min heap in our c code with the functionality of adding and removing items.

# Heap Based Priority Scheduling:

In a heap, as per its properties, the parent node has the greater value than its children nodes. Hence implementing the priority scheduler with heaps. The process with highest priority will be the head of the heap. Other all nodes will be set in a descending order of the priority. If a new node comes, it is set on the leaf node with right alignment. Then the node is compared with its parent node for the proper place of the process to be set.

# WHY WE MOVED TO THE KERNEL 2.6?

The move to kernel 2.6 Was a decision made on the basis of the understanding of how CFS was implemented in the kernel 3.19. the CFS implementation in the latest kernel was done with the help of red black tree data structure. This algorithm also includes the time calculation which actual forms the main part of the implementation of CFS. The CFS has the main feature of time slice which is done through run time calculation in the code of fair.c present in the kernel. Now as we know that the heap based priority scheduler that we are implementing does not have such a feature. Rather this is the point at which CFS is better compared to any other present right now. The CFS is better and more efficient. Now for implementing the heap based priority scheduler , the data structure that we use is HEAP. Now the current scheduler is using RB tree. The conversion from RB tree to HEAP is almost not possible. This made us realize that the code for the kernel that we were trying to understand would only make our life more difficult for the implementation but rather we would not be able to do so.

So we shifted to kernel 2.6 in which the data structure used is the simple queue and there is no introduction of the time slice calculation. This makes it easier for us to understand and also implement it. So now we shift to the kernel 2.6 for our focus of implementation of a heap based priority scheduler.

So for this project we take kernel 2.6 as it also includes the parameter of goodness calculation which is also important for us when we want to implement a priority queue scheduler.

# 2.6 kernel scheduler in detail:

Algorithm explaned:

In this scheduler, the scheduler maintains two different queues which is according to the priority as well as per a CPU.

These two queues are :

1. Active : here those processes are included which have their time allotted left.
2. Expired: as the name suggests, those processes that have completed.

This scheduler maintains basically two different scheduling algorithm which are based on either being a real time processes or a normal process.

For any normal process the scheduling algorithm that it follows is as follows:

1. Each and every process is give a value using the nice
2. According to it the process is given a priority number
3. According to the priority that it got the process is give the time for same priority the scheduler round robins it.

This way we ensure that the algorithm for scheduling is prioritized as well as fair for scheduling.

Whenever a process has completed its quantum the algorithm sees the if it is an interactive you place it in an Active queue and if not we recalculate its priority and place it according to its new priority in the Expired queue.

Now we know that the re-insertion can also be in the interactive processes. So as we saw once it is reinserted in Active queue.

We do this only when the expired queue process has at least run recently.

This is done so that the starvation doesn’t occur. This means no matter how many interactive processes come the static or the normal processes would at least be given time to perform their tasks.

This scheduling algorithm runs in O(1) which means it runs for a constant time

# How is the priority decided?

Now when we said that priority of any process has to be decided accordingly for any real time process or whether the process is normal process. This is actually done through how it interacts.

If the process is interactive, it is given more time to execute as compared to the ones which are simply computer bounded. In fact those processes which are only CPU bounded receive a lower time amount.

# Code

The following is the code to be altered in the sched.c:

static void

enqueue\_task(struct rq \*rq, struct task\_struct \*p, int wakeup, bool head)

{

printk("Inside enqueue,head:%d\n",head);

if (wakeup)

p->se.start\_runtime = p->se.sum\_exec\_runtime;

sched\_info\_queued(p);

p->sched\_class->enqueue\_task(rq, p, wakeup, head);

p->se.on\_rq = 1;

}

static void dequeue\_task(struct rq \*rq, struct task\_struct \*p, int sleep)

{

printk("Inside dequeue\n");

if (sleep) {

if (p->se.last\_wakeup) {

update\_avg(&p->se.avg\_overlap,

p->se.sum\_exec\_runtime - p->se.last\_wakeup);

p->se.last\_wakeup = 0;

} else {

update\_avg(&p->se.avg\_wakeup,

sysctl\_sched\_wakeup\_granularity);

}

}

sched\_info\_dequeued(p);

p->sched\_class->dequeue\_task(rq, p, sleep);

p->se.on\_rq = 0;

}

# Attempts:

We attempted to alter the runqueue data structure and applied the logic of heap into it. We compiled the kernel again with the altered code. We wrote suitable printk statements at the start, end and middle of the code. When we compiled and then again restarted, the kernel crashed and it was not able to boot up the system. The printk statements also showed that the pointer is not entering the code. It prints the starting statement but then doesnt go inside the code.

A reattempt was made by altering the logic of the code and this time after rebooting, the GRUB loader was found vulnerable and the laptop started in a grub rescue mode. After solving the grub rescue mode, ubuntu as a whole was not found in the system. Whenever we pressed enter on ubuntu, it entered an infinite loop and screen was shown black.