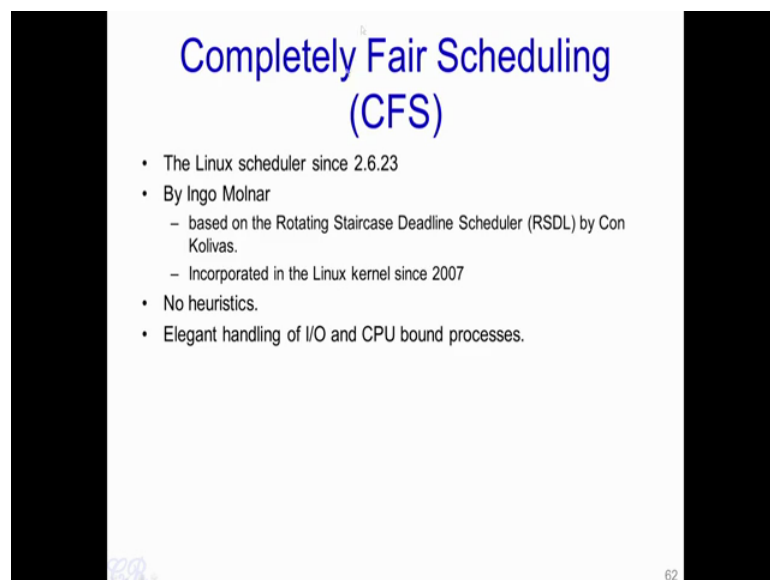


**Introduction to Operating Systems**  
**Prof. Chester Rebeiro**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Madras**

**Week – 05**  
**Lecture – 22**  
**Completely Fair Scheduling**

In this video lecture, we will look at the Completely Fair Scheduler. So, the completely fair scheduler or the CFS scheduler is the default scheduler used in Linux kernels in the latest versions.

(Refer Slide Time: 00:29)



**Completely Fair Scheduling  
(CFS)**

- The Linux scheduler since 2.6.23
- By Ingo Molnar
  - based on the Rotating Staircase Deadline Scheduler (RSDL) by Con Kolivas.
  - Incorporated in the Linux kernel since 2007
- No heuristics.
- Elegant handling of I/O and CPU bound processes.

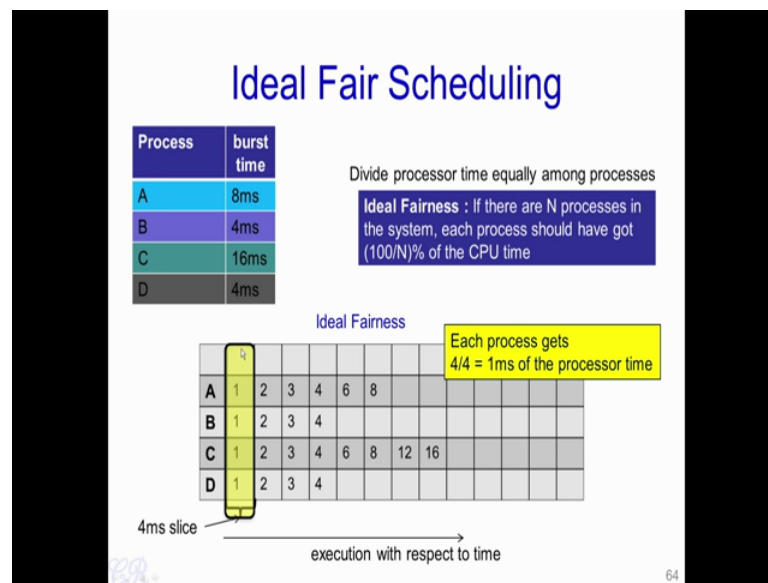
62

So the CFS scheduler has been incorporated in the Linux kernel since version number 2.6.23, and has been used as the scheduling algorithm since 2007. So, it was based on the Rotating Staircase Deadline Scheduler by Con Kolivas. So, the advantage of the CFS scheduler compared to the  $O(1)$  scheduler in particular is that there are no heuristics which are used and there is very elegant handling of I/O bound and CPU bound processes.

Essentially the interactive and non-interactive or batch processes are very easily fit into this particular scheduler. So, we will see a very brief overview of the CFS scheduler.

Now as the name suggest the CFS scheduler or the completely fair scheduler aims at dividing the processor time or the CPU time fairly or equally among the processes.

(Refer Slide Time: 01:34)



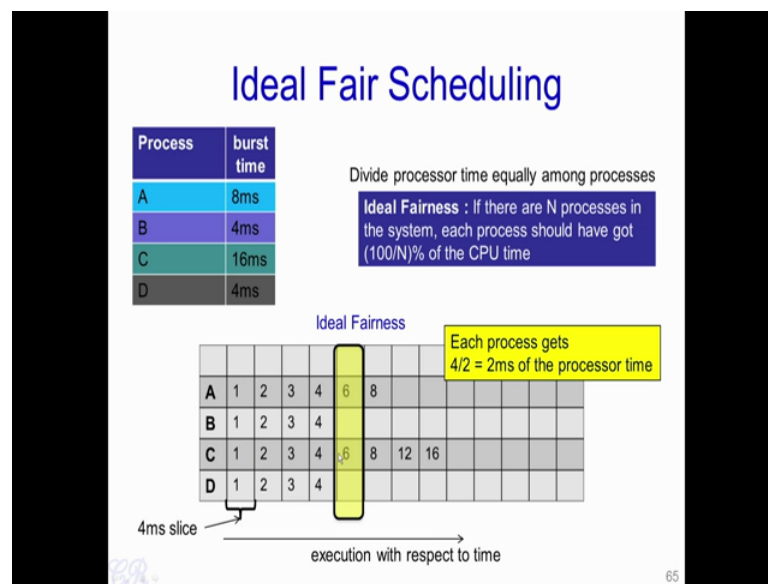
In other words, if there are N processes present in the system or present in the ready queue and waiting to be scheduled, then each process will receive  $(100/N)\%$  of the CPU time. So, this is the Ideal Fairness. Let us take a small theoretical example for this ideal fair scheduling.

Let us consider the four processes A, B, C and D, and having the burst time 8 milliseconds, 4 milliseconds, 16 milliseconds and D has the 4 milliseconds respectively (refer above slide). So, what we will do is let us just divide the time into quanta of 4 milliseconds slices and what we will now see is how the ideal fair scheduling should take place. So, in an ideal fair scheduling, at the end of say this 4 milliseconds epoch, all processes which are in the ready queue should have executed for the same amount of clock cycles.

For instance, if we look at this particular first epoch (first cycle), so it has 4 milliseconds, and we have four processes are present in the ready queue; therefore each process should get 4 divided by 4 that is 1 milliseconds of processor time (refer above Ideal fairness

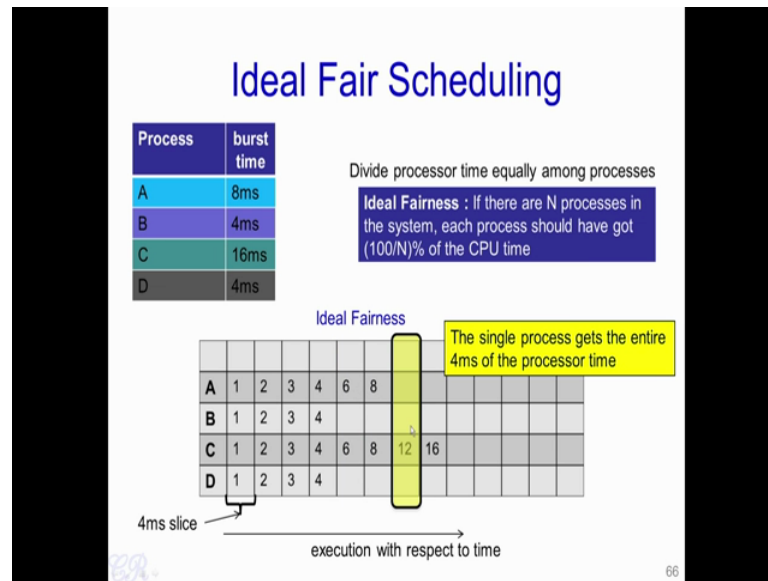
block). Therefore, A, B, C and D will execute for 1 millisecond (each). In a similar way for the 2<sup>nd</sup> cycle, there are four processes again, and therefore, these four processes get an equal share of the slice. So, each process executes for 1 millisecond again. So, therefore, in all A has executed for 2 milliseconds, B for 2, C for 2, and D for 2 milliseconds. Similarly, for 3 and for 4 (same cycle as first and second), so at the end of the 4<sup>th</sup> epoch, we see that processes B and D have completed (4ms of both processes are over). So, what happens next?

(Refer Slide Time: 03:32)



Now after B and D completes, we see that we have two processes present in the ready queue that is A and C, also the time quanta remains as 4 milliseconds. So, now each process gets 4 (time slice) divided by 2 (remaining process) that is 2 milliseconds of the processor time. Therefore, process A executes for 2 milliseconds, similarly process C executes for 2 milliseconds. Similarly, for the next epoch, A executes for 2 more milliseconds, and C executes for 2 more milliseconds. So, both have executed for 8 millisecond and as a result, A has completed executing.

(Refer Slide Time: 04:16)




Now, the last part we see that only C is present in the ready queue and it is the since it is the only process which is present in the ready queue. So, it is given the entire slot of 4 milliseconds. So, C executes for 4 milliseconds and followed by the final slot where it executes for another 4 milliseconds to complete its burst time. So, what you see in this ideal scheduling is that in each epoch or in each slot, the scheduler is trying to divide the time equally among the processes, so that asymptotically all processes execute for the same amount of time in the CPU. So, you see that all processes execute for 4 milliseconds here, at the end of this all processes execute for 6 milliseconds then 8 milliseconds and so on. How is this ideal fair scheduling incorporated in the CFS scheduler?

(Refer Slide Time: 05:21)

## Virtual Runtimes

- With each runnable process is included a virtual runtime (**vruntime**)
  - At every scheduling point, if process has run for **t ms**, then (**vruntime += t**)
  - **vruntime** for a process therefore monotonically increases


67

So, this is done by what is known as the Virtual Runtimes. In each processes PCB that is in each processes process control block, an entry is present known as the vrun time or the virtual run time. At every scheduling point, if a process has run for t milliseconds then its vruntime is incremented by t. Vruntime for a process, therefore will monotonically increase.

(Refer Slide Time: 05:51)

## The CFS Idea

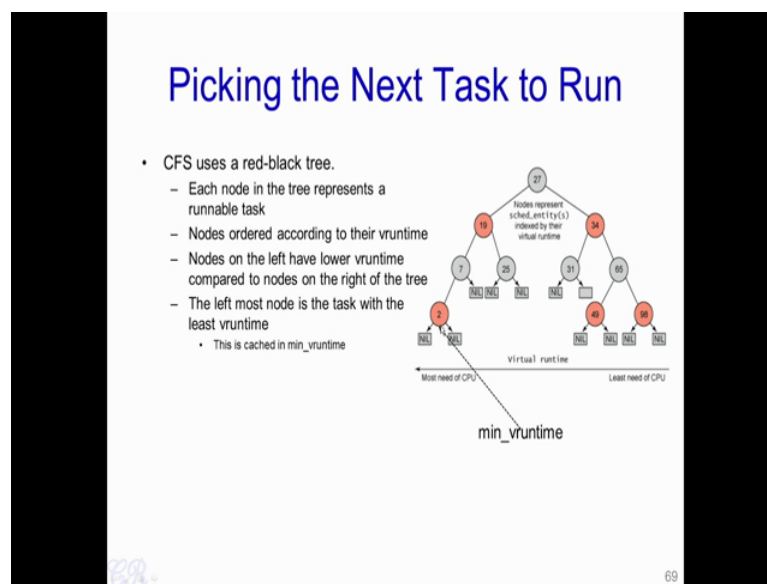
- When timer interrupt occurs
  - Choose the task with the lowest vruntime (**min\_vruntime**)
  - Compute its dynamic timeslice
  - Program the high resolution timer with this timeslice
- The process begins to execute in the CPU
- When interrupt occurs again
  - Context switch if there is another task with a smaller runtime

68

Now, the basic CFS idea is whenever there is a context switch that is required to be done, always choose the task which has the lowest vruntime. So, this is maintained by a variable called `min_vruntime`, this is a pointer to the task having the lowest virtual run time. So, then the time slice required is the dynamic time slice for this particular process is computed and the high resolution timer is programmed with this particular time slice.

The process begins to then execute in the CPU. When an interrupt occurs again a context switch will occur if there is another task with a smaller run time. So, you see that this particular process which is selected to run over here (from first point mentioned in above slide) will continue to run until there is another task with a lower run time.

(Refer Slide Time: 06:51)



Now in order to manage this various tasks with various run times, the CFS scheduler with quite unlike the schedulers which we seen so far do not use ready queue; instead, it uses a red black tree data structure (refer above slide). So, in this red black tree or the rb tree data structure, each node in the tree is represented as a runnable task. Nodes are ordered according to their virtual run time.

Nodes on the left have a lower run time or lower vruntime compared to nodes on the right of the tree that is if you see these particular things (rb tree in above slide), so each

node is a task and each node has a number written over here (inside the circle) which is the virtual run time for that particular task. So, you see that each task on the left (left side of the tree) has a lower virtual run time compared to task on the right.

Now, the left most node of this rb tree is the task which has the lowest vruntime or the lowest virtual run time. So, in this particular case (mentioned in above slide), it is this particular node (node with vruntime as 2) which corresponds to the task having the lowest virtual run time, therefore the scheduler should pickup this task to run next.

In order to find this task, there are two ways which are possible; one is you could traverse a tree and go towards the left until you reach a leaf, or the other way is we could directly have a pointer like the `min_vruntime` which points to the left most node of the tree. So, whenever the scheduler needs to make a context switch, it would just need to look into where `min_vruntime` points to and pick out this particular task. This quite naturally will be the lowest or the task with the lowest virtual run time.

(Refer Slide Time: 08:48)

### Picking the Next Task to Run

- At a context switch,
  - Pick the left most node of the tree
    - This has the lowest runtime.
    - It is cached in `min_vruntime`. Therefore accessed in  $O(1)$
  - If the previous process is runnable, it is inserted into the tree depending on its new vruntime. Done in  $O(\log(n))$ 
    - Tasks move from left to right of tree after its execution completes... starvation avoided

Virtual runtime  
Most need of CPU      Least need of CPU

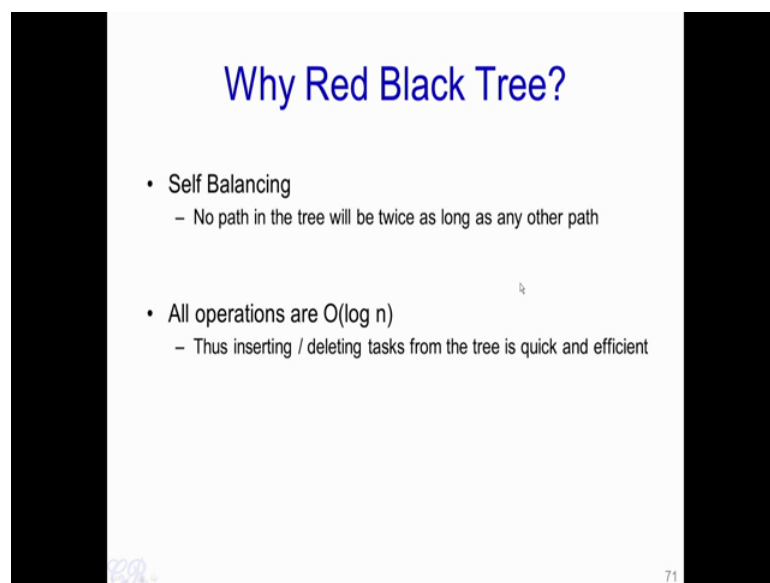
`min_vruntime`

So, this choice of the lowest vruntime is can be done in  $O(1)$  and therefore, independent of the number of processes present in the rb tree. So, at the end of the time slice, if this process which is currently executing is still runnable that is it has not blocked on an I/O

or it has not exited then its new virtual run time is computed based on the amount of time it has executed in the CPU. Then it is inserted back into the tree corresponding to its virtual run time. So, a process in other words would be picked out from the left most part of the tree because it has the lowest virtual run time and then it would execute in the CPU for some time say  $t$  milliseconds.

And at the end of its time slice, its virtual run time would be incremented by  $t$  values, and it would be inserted again into the tree. Now it will not go to the left of the tree, but it will rather be inserted somewhere in the middle towards the right (right side of the tree). Thus as the virtual run times increment, a process moves towards from the left towards the right. This ensures that every process gets a chance to execute because it ensures that at one point or the other, every process is going to have the minimum virtual run time in this particular tree and therefore, will get executed thus starvation is avoided.

(Refer Slide Time: 10:24)

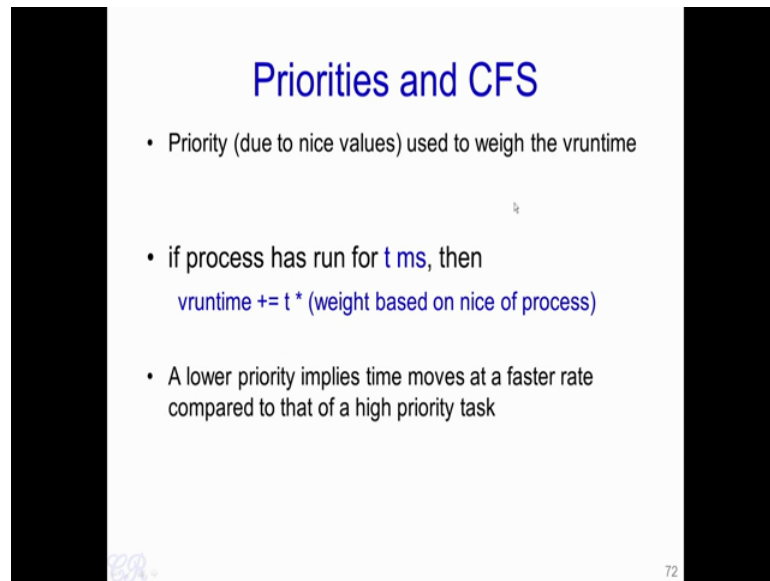


The slide is titled "Why Red Black Tree?" in blue text. It contains two bullet points: "• Self Balancing" with a sub-point "– No path in the tree will be twice as long as any other path", and "• All operations are  $O(\log n)$ " with a sub-point "– Thus inserting / deleting tasks from the tree is quick and efficient". The slide is flanked by two black vertical bars. At the bottom left, there is a small logo, and at the bottom right, the number "71" is visible.

So, why do we choose the red black tree or rather why did the Linux kernel choose the red black tree for the CFS scheduler. So, one obvious reason is the rb tree is self balancing. So, no path in the tree will be twice as long as any other path because of the self balancing nature of the tree. Due to this, all operations will be  $O(\log n)$ , thus inserting or deleting tasks from the tree can be quick and done very efficiently.



(Refer Slide Time: 10:55)



## Priorities and CFS

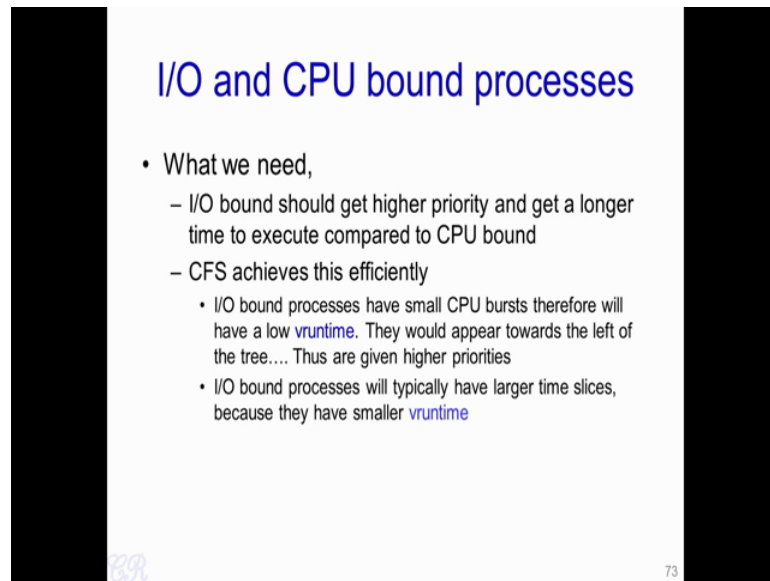
- Priority (due to nice values) used to weigh the vruntime
- if process has run for  $t$  ms, then  
 $vruntime += t * (\text{weight based on nice of process})$
- A lower priority implies time moves at a faster rate compared to that of a high priority task

72

Now how are priorities implemented in the CFS scheduler? So, essentially, CFS does not use any exclusive priority based queues as we seen in the  $O(1)$  scheduler, but rather it uses priorities to only weigh the virtual run time.

For instance, if a process has run for  $t$  ms then the virtual run time is incremented by  $t$  into weight based on the nice value of the process i.e  $vruntime += t * (\text{weight based on nice of process})$ , essentially based on the static priority of the process. So, a lower priority implies that the time moves at a faster rate compared to that of a high priority task. So, essentially what we are doing is we are providing a weight for the time that its executes, that is we are either accelerating the time or decelerating the time at which a process runs. So, this weight is used to implement priorities in the CFS scheduling algorithm.

(Refer Slide Time: 12:01)



## I/O and CPU bound processes

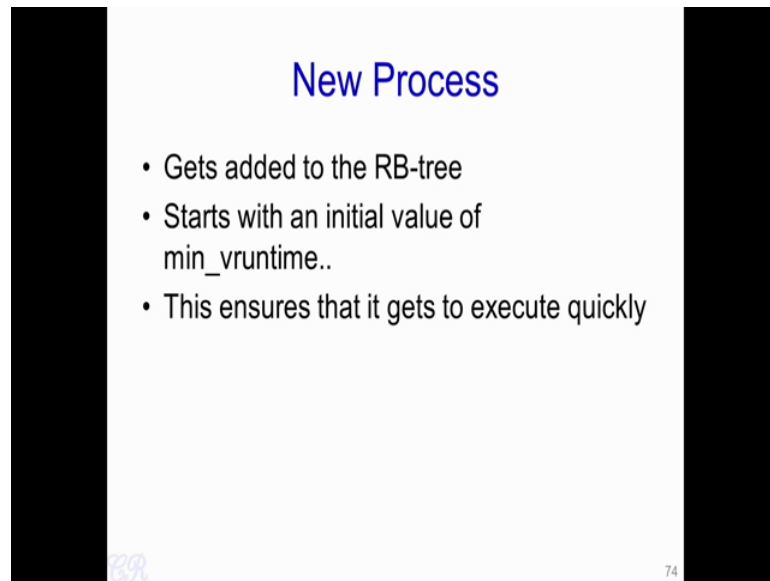
- What we need,
  - I/O bound should get higher priority and get a longer time to execute compared to CPU bound
  - CFS achieves this efficiently
    - I/O bound processes have small CPU bursts therefore will have a low *vruntime*. They would appear towards the left of the tree.... Thus are given higher priorities
    - I/O bound processes will typically have larger time slices, because they have smaller *vruntime*

73

Next we will look at how the CFS scheduler distinguishes between an I/O bound and a CPU bound process. So, essentially this distinguishing is done very efficiently. It is based on the fact that I/O bound processes have a very small CPU burst, and therefore its *vruntime* does not increment very significantly. As a result of this, it is more often than not appearing in the left part of the rb tree. Therefore, it gets to execute more often than other processes. This is because of the fact that as we mentioned as time progresses each process in the CFS scheduler is picked up from the left most nodes and executes and then it is placed on the right; therefore, in general every process moves towards the right part of the rb tree present in the scheduler.

Now, for the I/O bound process, since the *vruntime* does not change too much or increments just by a small margin, it does not move to the extreme right, but rather it's still stays towards the left part of the tree. Thus very soon, it will soon find itself as a process with a lowest *vruntime* and will have a chance to execute again in the CPU. Now as a second effect due to the small *vruntime* or the small virtual run time of the I/O bound processes, it is given a larger time slice to execute in the CPU. Thus we see the I/O bound and CPU bound processes are very well distinguished quite inherently by the CFS algorithm.

(Refer Slide Time: 13:50)



## New Process

- Gets added to the RB-tree
- Starts with an initial value of min\_vruntime..
- This ensures that it gets to execute quickly

74

When a new process gets created, it gets added to the red black tree. Now its starts with a initial value of min\_vruntime therefore, gets placed to the left most node of the tree and this ensures that it gets to execute very quickly. So, as it executes depending on the amount of time it executes whether it is an interactive or a CPU bound process, its position within the rb tree would vary. This was a brief introduction to the CFS scheduler, which is the default scheduler in current versions of the Linux kernel.

Thank you.