Report

Structures

Struct redblacknode: This is the structure to create a node in the red black tree.

Struct rbtree **:** This structure is used to create the red black tree.

Struct Processor **:** This structure is used to create the processors only in the question 2.

Functions

Function node* createnode:

This function is type of node*, because it returns a node* type node at the end. This function gets two parameters called 'data' and 'tt'. Then it assign memory for a node (simply creates the node at this point), using malloc and assign all the node*type variables to NULL. Then assign 'data' to the 'vruntime' and 'tt' to the 'totaltime'. Then assign 'R' for the 'colour' variable to indicate that it's colour is 'red', because in a red black tree each newly inserted node should be 'red colour'.

Function node*createtree:

This function is to **create the tree** (linking the nodes).First it will assign memory for the tree structure and assign that memory to a pointer variable called 'newtree', and for another node*type 'nillnode'. Then assign all the node* type variables of the 'nillnode' to NULL, its colour to 'B' which indicates 'Black', as all the null nodes are black, and it's virtual runtime(vruntime) to 0 as it's a null node. Finally it will assign the root of the created tree to that above created nill_node to make both of them 'null' and pointing to the same node when the time of creating the tree (as no node is link with each other at first). Finally it will return the newly created tree.

Function void inorder:

In 'inorder function' it will first get the tree and from which node the function should start. Most of the time it will start from the root node. Then it will print the leftnode of the root (selected node), then print the value of root node and at last prints the value of the root node's right child. Most of the times we use this function when we want to print the 'key values' of the nodes of any tree in ascending order.

Function node*insert:

This function gets three parameters, the newly created tree* type newtree and two integer type variables called data and tt. Then initialize a node* type node called **'nn'** which means **'new node'** and pass the values **data ,tt** to the createnode function to create the particular node and assign the node to the **'nn'**. Then created another two node* type variables called **'pn'** for parent node and **'ptr'** for 'pointing'.

Algorithmic explanation for the insert function:

First we should have a pointer variable which points to the root of the tree. Then it checks whether that pointer points to a nil node of the tree. If not then that it save the data of that node (pointer pointing to) in another node type variable and then checks whether the newly created node's virtual runtime is less than the pointed node's virtual runtime. If it is true then it should traverse to the left side of the tree as the left side of the tree holds the data less than the right side's data.

If the newly inserted node's **virtual runtime** is less than or equal to the currently pointed node's virtual runtime, then it should traverse the right side of the tree so the pointer will point to the next right side node. When the pointed node becomes a nil node of the tree this process will stop and it will assign the pn (previously saved pointed node) as the parent node of the current pointed node (the newly created node). If this pn is a nil node then obviously there is no such node so the newly inserted node will become the root node of the tree, else it should check whether the newly inserted node's **virtual runtime** is less than the parent node's virtual runtime. If so the newly inserted node becomes the left child of its parent node 'pn'. At this point the program should check whether any **violation** of red black tree has occurred so far due to the new insertion.

There are **5 rules** in a red black tree which should not be violated:

1. Every node in the tree is either red or black.
2. The root node is always black.
3. Every nil node (leaf) is black.
4. Is a node is red both its children are black (no two consecutive red nodes).
5. The black height of the tree in every path should be the same.

Most of the times the violations occur because, every new node should be **red when it is inserted** to the tree. So this will violate the **4th property** of the red black tree. Then when we are going to fix that it will violate the **5th property** too. So we need to **fix these violations** after inserting. Here it will get a new node type pointer variable to fix up these violations. According to the algorithm while the newly inserted node's parent's colour is red we have to do the fixing up process as I above mentioned. If newly inserted node's parent is the left child of its parent's (**grandparent of new node**), the newly inserted node's **uncle** will be assigned to the pointer node. Then it will check whether that uncle's colour is red. If it is red to fix up the problem, we have to perform a 'colour flip'. This will change the new node's parent's colour to black and its (new node's) uncle's colour to black too. Then new node's grandparent's colour will be changed to red. By this the violated 4th property will be fixed and then it will return the newly inserted node successfully without any violation.

If new node's uncle is black to fix up the violation we have to perform **a rotation**. For that it will first check to where the new node is inserted. If it is inserted as the left child of its parent, new node (nn) will be pointed to its parentnode and as the violation is in the right child's left sub tree which leads us to perform a right rotation along with a left rotation. So it will first perform a right rotation and then change the nn's colour to black then the grand parent of nn will become red and then it will perform a left rotation at the grandparent of the new node, and after fixing up the violation like this it will return the nn(new node) successfully.

Algorithmic explanation for the rightrotate function:

This will first get the node (nnode) that we want to perform the right rotation at. And assigns its leftnode to a new node type pointer (a). Then it will change nnode's left child to a's right child and if a's right node is not null a will become nnode. If nnode's parentnode is

a nil node, the root of the tree will be changed to a, else nnode will be changed to a. And **a's rightnode** will become **nnode**.

Algorithmic explanation for the leftrotate function:

Same as the right rotation **changing the words right to left and left to right**.

Function node*deletenode:

Algorithmic explanation for the delete function:

After deleting a node too we have to fix up the violation of rules if any are there in a red black tree as above mentioned. First it will get the node to be deleted as a parameter. If the node to be deleted is a red leaf node nothing to be done. Simply delete it. If the node to be deleted is black then it can affect to the black height property of the RBT. So we have to pass the blackness of the node down to its immediate child. There are **eight different cases** that can be occurred at this moment as **symmetrical** we can only consider 4 of them. When the blackness is passed down the child can be double black or red black according to its original colour. Once the removal of the node is completed the **extra blackness** is passed up the tree until the extra blackness is removed. When that extra blackness reaches to the root it will be ignored. There are **4 cases** to be consider (totally it's eight), when fixing the **double blackness.**

Case1: **Double black node has a black parent with a red sibling**.

This is a nonterminal case which will reduces us to the terminal cases either case 2 or 4. Inhere simply swap the colours of the parent and sibling, and left rotate the sibling.

Case 2: **Double black node has a black sibling with black children.**

This is a terminal case. Here we have to remove one black from the double black node and its sibling and give one black to its parent.

Case 3: **Double black node has a black sibling with sibling's leftchild is red and rightchild is black.**

This is a nonterminal case which will reduces us to the terminal cases either case 2 or 4.Simply swap the colours of sibling and its left child, and perform right rotation on sibling's leftchild.

Case 4: Double black node has a black sibling with sibling's right child is red.

In this case we don't consider the sibling's left child's colour. This is the final terminal case. Simply swap the colours of the node to be deleted's parent and its sibling. Then re colour the double black node to singly black node and finally change the sibling's right child's colour to black.

By performing one or two cases in fix up process we can successfully **rebalance the tree** removing violations.

Function min:

This function will always return the left most node of the red black tree as the left most node's key value is the **minimum key value** of the tree.

Function run:

This function will always decide which process (job) is to be run next in the CPU. According to the algorithm it will choose **the leftmost node of the tree** to run next in the CPU as it has the minimum virtual run time among all the processes in the red black tree.

QUESTION 1

Main function and the algorithm of CFS scheduler:

This program will first create an empty red black tree. Then it will ask the user about the amount of **time quantum** of the particular CPU. The time quantum is the amount of time which the CPU can give for the processes to run. In CFS (Completely Fair Scheduling) if a single process is running, the processor gives the 100% of its power to that process, if two processes are running the CPU gives each process 50% of its power equally. Likewise the processor's power would be equally divided among the number of processes. Therefore this CPU would be **"fair"** to all the tasks running in the system.

The operations of the CFS scheduler:

a) The **left most node** of the scheduling tree is chosen (as it will have the lowest spent execution time (virtual run time)), and **sent for execution**.

b) If the process simply **completes execution**,it is **removed** from the system and scheduling tree.

c) If the process reaches its maximum execution time or is otherwise stopped (voluntarily or via interruption) it is **reinserted** into the scheduling tree based on its new spent execution time.

d) The new left-most node will then be selected from the tree, repeating the iteration.

e) If the process spends a lot of its time sleeping, then its spent time value is low and it automatically gets the priority boost when it finally needs it. Hence such tasks do not get less processor time than the tasks that are constantly running.

- Reference: Completely Fair Scheduler – Wikipedia -

After getting the time quantum of the CPU, it will get the **number of processes** that are going to be in the ready queue (in this case the red black tree), and copy that amount to another float time variable. This program will have a variable called **"rtq"** to save the **remaining time quantum**. That means after the CPU gives a part of its time quantum to a process, each time it will update the remaining time quantum of the CPU and holds that value in this variable. Here it has another time variable which is called as **"ptime"** to save the **amount of time quantum which will be given to each process at a particular moment**. Always the ptime will be **"rtq/num_p"** where **num_p** gives the **number of processes at the moment**. Ptime gets that amount because in **CFS scheduler** it will divide the time quantum among all the processes in the ready queue equally. After that it will get two user inputs for each process such that the **virtual runtime and the total runtime of a particular process**. Then it passes those values with the newly created tree to in the insert function. These steps will be repeated until all of the processes' data are inserted. Then it will call the inorder function just to print the vruntimes of processes in ascending order.

Then as previously mentioned above, it will call the **"run" function** and return the **process which is going to run next**, will call that process as **"j"**. It will update the j's vruntime with the new vruntime as it is running. To do this it will save the j's vruntime and total runtime in two variables, and add the ptime (rtq\process number) to its vruntime as it has run ptime amount at this time. Then it will check whether that process j has got extra time than it actually needed. If the vruntime is larger than or equal to the total time of j that means j is completed so it will be permanently deleted by the system, and If an **extra time** is available that extra amount will be saved to available called **remain_t** so that amount can be assigned

to the next process which is going to run by **adding that remain_t to the rtq** (remaining time quantum of the CPU). As one process is deleted **the number of processes will be reduced by 1**, and the number of processes which are deleted will be increased by one. If the vruntime of j is less that its total time then it will be deleted temporarily in order to update its new vruntime and inserts it back to the red black tree.

At this point if the remaining time quantum is less than or equal to 0 that means the remaining time quantum is over so that we have to update it back to the **initial time quantum of the CPU.**

QUESTION 2

Symmetric Multiprocessor Architecture (SMP)

A symmetric multiprocessing system is a hardware configuration that combines **multiple processors** within a single architecture. In and SMP system, multiple processes can be allocated to different CPUs within the system which makes a **SMP machine** a good platform for coarse-grained **parallelism**.

-https://www.sciencedirect.com/topics/computer-science/symmetric-multi-processing-

Explanation of the Main Function

In this program I have got **two processors** (CPU s), to imitate the **"Symmetric multiprocessing in a CFS scheduler"**, with the **same time quantums**.

So according to my program there are **four processes** and this program will divide those processes into those **two processors** at the same time.

In this program I have made a **structure "processor",** to create two different processors so then I can maintain the time quantums as well as their remaining time quantums for the two processors separately.

Here I have made **four different functions to imitate the four processes**. 1st function to print "Simple letters from 'a' to 'p' in the alphabet. The 2nd function is to print eight "*" (star symbols), the 3rd function to print integers from 1 to 4 and the

4th one to print capital letters from 'A' to 'K'. Here I have assumed that to read a single character the processor takes only 0.5 units of time (Nano seconds).

Now it will start the Multi processing. At first both of the processors are free (not occupied by any process), so while the deleted(completed) process count becomes the total number of processes, if processor 1 is not occupied, then according to the CFS scheduling it will get the node with the minimum virtual runtime as I have mentioned above in the question number 1. Then that process will be allocated to the processor 1. While that process is started running in the processor 1, the scheduler will get the **next leftmost node** of the RBT which has the minimum virtual runtime. Then that process will be allocated to the processor 2. While running if the taken process is finished, the processor will get the next minimum process to run.

When the scheduler chooses a process to run in one of the processors it will **call one of the above mentioned functions** according to the **process's (which is going to run) process number**. For an example, if the scheduler chooses process number 3 as the minimum process which is going to run next, it will call the function 3 and will print the characters according to the function 3.

After finishing every round (every time unit) it will print the message **"ROUND IS OVER",** so we can understand in between two of those messages there is a one round which **"both the processors run their processes parallely"**. As I have made two different processors using a structure, it will simply update the **"rtq", "ptime"** for each process and every time when process is deleted it will update the process count so then the processors can adjust their "ptimes" according to it. The "rtq" and the "ptime" are with the same meaning as mentioned in the **question 1**.

"At the end you can see that the output has both 'o' and 'p', because at that time all the other processes are finished so the processor has the time to complete the process number 1."

When the processes reaches to its maximum needy time to run (the total time), it will automatically deleted by the system and the process count will be decremented. After completing every process in the ready queue (red black tree in this case) it will print a message saying that **"All the processes has been completed"** to indicate that all the processes are finished.

Assumptions

1) **In the first question I have not created processes separately because I thought it is better when the user can input the number of processes, so then as it is not limited I can imitate the functionality of a CFS scheduler successfully.**

2) As an output is needed whenever a process is running, in the **first program** it will print a message when a process is running, for an example "The process 2 is running" .By that we can identify **which process is running at the moment** in the **first question**.

3) All the time values are in Nano seconds and therefore they are stored in **float type** variables, and the **virtual runtime** is the **key value** of a node in the red black tree.

4) The **virtual run time is a user input** in the **first question** because then we can decide if the process is newly arrived or was it already there in the ready queue (RBT).If the initial **vruntime of a process is 0, that means it is a newly inserted process**. The total runtime is the amount of time which the process needs for its completion. In the **second question** I have allocated all the **vruntimes to 0**, assuming that all the processes are newly arrived to the ready queue.

5) If the CPU is able to give time more than a process needs at a moment that **extra time amount** will be added to the remaining time quantum of CPU so that it can also be assigned to another process.

6) In the **2nd program**, the output in between two **"ROUND IS OVER"** messages are the part which two processors run their processes **parallely**.

7) In the **2nd program**, the variable **occupied** stands to save the state of the processor, whether it is running a process or not.

8) **In the 2nd program I assumed that it consumes only 0.5 time units (Nano seconds) to read one character from the given functions.**

References

- Lecture notes of Red Black Trees.
- https://en.wikipedia.org/wiki/Completely_Fair_Scheduler
- https://www.linuxjournal.com/node/10267
- https://www.youtube.com/watch?v=scfDOof9pww
- https://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt
- https://developer.ibm.com/tutorials/l-completely-fair-scheduler/
- http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap14.htm
- https://opensource.com/article/19/2/fair-scheduling-linux
- https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html
- https://www.sciencedirect.com/topics/computer-science/symmetric-multi-processing
- https://www.youtube.com/watch?v=aKjDqOguxjA

Manuli Wanniarachchi
18001841
1st year - 2019