CFS Scheduler for xv6

Submitted by:
Ritvik Rishi - CS18B057
Vishav Rakesh Vig - CS18B062

Department of Computer Science and Engineering

Operating Systems(CS3500) Course Project

December 12, 2020

OVERVIEW

In the xv6 operating system, the scheduler uses round-robin scheduling. In this project, we import linux's kernel scheduler to xv6. The linux scheduler is the Completely Fair Scheduler(CFS) which uses red-black trees in its implementation. For the whole implementation of CFS in xv6, we have followed the linux default values for min_gran, time_epoch, nice value range and the respective weights associated with them.

METHODOLOGY

The Idea of CFS:

Whenever a timer interrupt occurs, we choose the task with the lowest vruntime (min_vruntime). This is implemented using a **Red-Black Tree**.
We compute the dynamic timeslice of that task and then program the high resolution timer with this timeslice.
Thus, after the timer interrupt, the process begins to execute in the CPU.
When a timer interrupt occurs again, we check the process with the min_vruntime and context switch if that has a smaller vruntime than the running process.

GOALS

1. **Implementing Red-Black Trees for the Completely Fair Scheduler in the xv6 kernel**
2. **Implementing the CFS scheduling policies in the xv6 kernel**
3. **Implementing user-defined priorities (nice values) in the Completely Fair Scheduler**
4. **Addressing the Limitations of xv6**

IMPLEMENTING THE RED-BLACK TREE FOR CFS:

A red-black tree is self-balancing, and no path in a red-black tree is twice as long as any other path. The insertion and deletion of tasks from the red-black tree is quick and efficient.
Time Complexity for Addition of process in R-B Tree = O(log n).
Time Complexity for Deletion of process in R-B Tree = O(log n).
A red-black tree is a binary search tree and the nodes in the tree are ordered according to their vruntimes. The leftmost node is the process with the minimum vruntime and it is also cashed in the variable min_proc.
The implementation details are given below:

Additions made in the proc structure

In our implementation for red black trees we modified the struct proc nodes to hold additional members necessary to implement the red-black tree.

Code Snippet:

```
struct proc {
  ...
  ...
// Added code here
  struct proc *ln;  //pointer to the left node in the tree
  struct proc *rn;  //pointer to the right node in the tree
  struct proc *pn;  //pointer to the parent node in the tree
  int color;  //colour of the node
  int pri;    //priority of the process
  int weight;  //weight based on priority of process
  uint64 vruntime;  //virtual runtime of the process
  uint64 aruntime;  //actual runtime of the process
  uint64 starttime;  //time when the process wakes up and executes
  uint64 interval;  //the scheduler period for the process
};
```

Functions that Interact with Scheduler:

We have 2 functions that are used in the scheduler which provide the interface between the implementation of the Red-Black Trees and the scheduler itself.

Function add_proc() :

Whenever a process is made RUNNABLE, it is inserted into the tree in O(log n) time. This is done in making the first process RUNNABLE in userinit(), forking in fork(), giving up the CPU after a scheduling round in yield(), waking up a process in wakeup(), and while killing a process in kill().

Function get_next_proc() :

When a timer interrupt happens and we have to schedule the process, we obtain the next process with the min_vruntime and set it's state to RUNNING and delete it from the tree in O(log n) time.

Code Snippets:

```
//Calls functions to update the runtime of a process
//and adds the process to the Red-Black tree
void add_proc(struct proc *p, int i, uint64 time_run){
  if(i == 0){  //the process is being added to the tree for the first time
    initial_vruntime(p);
  } else if(i == 1) {  //the process is being added back into the tree
    update_runtime(p,time_run);
  }
  acquire(&rbtree_lock);
  numproc++;
  totweight = totweight + p->weight;

  if(p->prevweight != p->weight){
    totweight = totweight + p->prevweight - p->weight;
    p->weight=p->prevweight;
  }
  insertproc(p);  //inserts the process with updated vruntime in the tree
  release(&rbtree_lock);
}

// This function is called in scheduler to
// remove the process with min vruntime from the
//  process tree and return it
struct proc* get_next_proc(){
    struct proc* current;
    current = min_proc;
    if(eqNIL(current))  return current;
    deleteproc(min_proc);    //deleting min_proc from the RB tree
    min_proc = getminproc();  //finding the next min_proc in tree
    return current;
}
```

Internal functions:

Given below are the pseudocodes and descriptions for the various internal functions used in **add_proc()** & **get_min_proc()** which are a part of the implementation of the Red-Black Tree.

Function insertproc():

Finds the location where the node can be inserted in the tree, and inserts it while maintaining the Red-Black Tree property.

Function deleteproc():

This function deletes a node from the red-black treewhile maintaining the Red-Black Tree's property.

Code Snippets:

```
//allocate node for process and insert in tree
void insertproc(struct proc *x) {
    vruntime = x->vruntime;
```

```
    // find where node belongs
    current = root;
    while (current != NIL) {
        parent = current;
        current = compLT(vruntime, current->vruntime) ?
        current->ln : current->rn;
    }
    // setup the node for insertion
    x->pn = parent;
    x->color = 1;
    if(parent) {  //insert node in tree
        if(vruntime < parent->vruntime))
            parent->ln = x;
        else
            parent->rn = x;
    } else {
        root = x;
    }
    insertFixup(x); // Restore Red-Black Tree Property
    // Internally calls helper functions Leftrotate() & Rightrotate()
    min_proc = getminproc();
    return;
}

//delete node z from tree
void deleteproc(struct proc *z) {
    struct proc* y, x;
    if (z is NIL) return;
    if (z has a NIL node as a child) {  //y has a NIL node as a child
        y = z;
    } else {    // find tree successor with a NULL node as a child
        y = z->rn;
        while (!eqNIL(y->ln)) y = y->ln;
    }
    x := only child of y
    x->pn = y->pn;     // remove y from the parent chain
    Replace y with x
    if (y != z) swapval(y,z); // Swaps proc structs
    //Now, x is such that it has at-most one non-leaf child
    if (y->color == 0)
        deleteFixup (x); //Restores Red-Black Tree Property
        // Internally calls helper functions Leftrotate() & Rightrotate()
}
```

HANDLING USER-DEFINED PRIORITIES:

User defined priority of processes is implemented using nice values. Nice value of a process is the priority associated with the process. We implemented this using a system call that sets the nice value in the proc structure.

System Call Implementation:

```
uint64
sys_nice(void){
  int nice_value;
    if(argint(0,&nice_value) < 0)
      return -1;
    if(nice_value < -20 || nice_value > 19 )
      return -1;
  myproc()->pri = nice_value;
  myproc()->prevweight = pri_to_weight[nice_value+20];
  return 1;
}
```

Usage of priority:

The user defined-priority (using nice values) is used to **weigh the increment in the vruntime** of a process. The priorities are also used to **weigh the timeslice** for which the process is scheduled in the CPU.

Code Snippets:

```
//Updates the virtual runtime (vruntime) and the
//actual runtime(aruntime) of a process
void update_runtime(struct proc *p, uint64 time_run){
  p->vruntime = p->vruntime + (100 + ((time_run*1024)/p->weight) * 100 );
  p->aruntime = p->aruntime + time_run;
}

//This function gets the dynamic timeslice for a process
uint64 get_timeslice(struct proc *p){
  uint64 timeslice;
  if(p->weight<=0)  panic("weight negative");
  timeslice = (get_timeepoch()*p->weight)/totweight;
  return timeslice;
}
```

IMPLEMENTING CFS SCHEDULING POLICIES IN XV6

How the CFS achieves fair scheduling

Since we cannot run all the processes simultaneously, to achieve fair scheduling the CFS scheduler schedules each process a part of its scheduling time epoch, so that it gives an illusion that the processes are running simultaneously on the system. The time interval for which the process is scheduled is the dependent on its nice value (user defined priority) and decreases with increasing nice values.

Setting the scheduler period:

As opposed to having a fixed time interval for which the processes are scheduled, in cfs we schedule the time slice dynamically for each process each time it is scheduled to run.

This time slice is calculated while the process is being scheduled to run and its state is being changed to RUNNING. This timeslice is then stored in the process structure. While returning to resume execution in the user space this timeslice is set as the interval for execution by assigning this value to CLINT_MTIMECMP. We use just_scheduled as a flag for returning from timer interrupts on that CPU. This is done in kernel/trap.c

Code Snippets:

```
//For our implementation, duration of an epoch = 20ms and min granularity = 1ms.

In proc.c
//Finding the scheduling period
uint64 get_timeepoch(){
    uint64 interval = timeepoch;
    if(numproc<0)  panic("numproc negative");
    if((numproc)&&(min_gran>(interval/numproc))){
        interval=min_gran*numproc;
    }
    if(!interval)  panic("interval zero");
    return interval;
}

//This function gets the dynamic timeslice for a process
uint64 get_timeslice(struct proc *p){
  uint64 timeslice;
  if(p->weight<=0)  panic("weight negative");
  timeslice = (get_timeepoch()*p->weight)/totweight;
  return timeslice;
}

In trap.c
//Setting the scheduling period
if(just_scheduled[id] == 1) {
    uint64 interval = p->interval;
    *(uint64*)CLINT_MTIMECMP(id) = r_time() + interval;
    just_scheduled[id] = 0;
}
```

Scheduling a process:

When a timer interrupt happens, we first yield the running process and insert it in the red-black tree containing runnable processes and switch context to the scheduler. We then extract the runnable process with the minimum vruntime from the red-black tree, set it's state to RUNNING, and switch context from the scheduler to that process.

The scheduler code :

```
for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();
    // Choosing the process with the minimum vruntime
    // and removing it from the tree
    acquire(&rbtree_lock);
    p = get_next_proc();
    if(eqNIL(p)){
      release(&rbtree_lock); continue;
    }
    acquire(&p->lock);
    if(p->state==RUNNABLE){    // Found a runnable process.
      // preparing for scheduling
      p->interval=get_timeslice(p);
      p->starttime=r_time();
      just_scheduled[CPUid()]=1;
      // For testing.
      if(!firstexit){
        p->noof++; p->vruntime2=p->vruntime; p->aruntime2=p->aruntime;
      }
      totweight=totweight-p->weight;
      numproc--;
      release(&rbtree_lock);
      p->state=RUNNING;
      c->proc=p;
      swtch(&c->scheduler,&p->context);
      c->proc=0;
    }
    else{
      release(&rbtree_lock);
    }
    release(&p->lock);
  }
```

Updating vruntimes:

We set the start time of a process when its state is set to running using r_time(), and we update the processes' vruntime whenever it goes to sleep or yields and the process is added back into the Red-Black tree. After calculating the process's runtime for that scheduling we update the vruntime by calling the update_runtime() function in add_proc(). The updated vruntime is then used to add back the process in the red-black tree.

```
void update_runtime(struct proc *p, uint64 time_run){
  p->vruntime = p->vruntime + (100 + ((time_run*1024)/p->weight) * 100 );
  p->aruntime = p->aruntime + time_run;
}
```

Elegant handling of CPU and I/O bound processes:

The CFS scheduler doesn't have any complex heuristics to calculate the dynamic priority, CPU and I/O bound processes are handled in a much more inherent way. In the CFS scheduler, even though both the processes are scheduled for the same time interval, due to the inherent nature of the I/O bound processes to have shorter CPU bursts than the CPU bound processes, their vruntime increments are smaller. Thus due to smaller vruntime increments overall their vruntimes remain low and appear on the left side of the tree more and hence these processes are scheduled more. In this way the CFS handles CPU and I/O bound processes without needing to compute complex heuristics like the O(1) scheduler.

ADDRESSING THE LIMITATION OF XV6:

Limitation:

The xv6 operating system does not have much support for I/O bound processes. I/O bound processes are processes that execute for short CPU bursts and then sleeps waiting for an I/O to wake them. But since there is no support for traditional input like scanf and even printf get written to a buffer, we cannot implement the I/O bound processes normally.

Work-Around:

Due to not being able to implement basic I/O functions we try to simulate them in xv6. We simulate the I/O behaviour using the sleep() system call. We use the sleep system call to imitate the act of waiting for an input. This has been used in our testcases so that we can test our scheduler against I/O bound processes.

RUNNING AND TESTING

Testing the scheduler:

We will now test the scheduler to verify that it is working policy and the CFS scheduling policy is followed. For this, we run the following tests :

1. Booting
2. usertests
3. Custom testcases

Booting:

After implementing all the changes shown above, we boot xv6 with "$ make qemu". The successful booting and subsequent successful running of shell indicates that the init process is getting scheduled properly and is running properly.

Running usertests:

The user program usertests.c tests xv6 system calls. After booting, we run usertests. Passing all the tests in usertests implies that all the system calls in xv6 are running, and processes are getting scheduled and running properly.

We now move to testing the scheduler and it's efficiency.

Running Custom Testcases:

To check if our CFS implementation does follow the CFS scheduling policies we will run our own test-cases, which check the I/O vs CPU bound Process Behaviour and also User Defined Priorities (Nice Values) implementation.

TEST 1 - I/O AND CPU BOUND PROCESSES:

In this test case we analyse the behaviour of I/O bound and CPU bound processes. I/O bound processes execute in short bursts and should get scheduled more as compared to CPU bound processes.

Test Code Explanation:

The custom test case forks multiple CPU and I/O bound processes (15 each) and then outputs each process's turnaround time, actual-runtime and average-vruntime increment and no of times it was scheduled. We are only calculating the last 2 values for before the first exit because we don't want any change in no of processes since that may cause a change in vruntime increments and hence to keep the conditions fair to both CPU and I/O bound processes we calculate the first 2 values over the course of execution of the processes and the last 2 values from their time they were forked to the time the first exit occurred.

Code:

```
for (i = 0; i < 30; i++) {
    pid = fork();
    if (pid == 0) {  //child
        j = (getpid()) % 2; // ensures independence from the first son's pid when gathering the↩
             results in the second part of the program
        switch(j) {
            case 0: // simulate CPU bound process (CPU):
                for (k = 0; k < 100; k++){
                    for (j = 0; j < 1000000; j++){}
                }
                break;

            case 1:// simulate I/O bound process (IO):
                for(k = 0; k < 400; k++){
                    sleep(1);
                }
                break;
        }
        fexit();  //system call to toggle firstexit
        exit(0); // children exit here
    }
    else  continue; // father continues to spawn the next child
}
```

Hypothesis:

Since the I/O bound processes have smaller CPU burst times, they should have smaller average vruntime increments and should be scheduled more no of times than the CPU bound processes.
Also since the I/O bound processes utilize the CPU for less % of turnaround time, for comparable turnaround times the actual runtimes of the I/O bound processes will be way less than those for CPU bound processes.

Running the test case:

We run this user program in xv6-shell as "$ tc1"

Output:

```
$ tc1
I/O Bound Process is exiting; actual_runtime = 243092; turnaroundtime = 16368011; ; avg_vruntime_increment = 298; scheduled 801 times
I/O Bound Process is exiting; actual_runtime = 252634; turnaroundtime = 16383657; ; avg_vruntime_increment = 299; scheduled 800 times
I/O Bound Process is exiting; actual_runtime = 243513; turnaroundtime = 16362273; ; avg_vruntime_increment = 298; scheduled 801 times
I/O Bound Process is exiting; actual_runtime = 264312; turnaroundtime = 16448840; ; avg_vruntime_increment = 304; scheduled 799 times
I/O Bound Process is exiting; actual_runtime = 250965; turnaroundtime = 16400352; ; avg_vruntime_increment = 299; scheduled 800 times
I/O Bound Process is exiting; actual_runtime = 261023; turnaroundtime = 16406754; ; avg_vruntime_increment = 299; scheduled 800 times
I/O Bound Process is exiting; actual_runtime = 241948; turnaroundtime = 16472969; ; avg_vruntime_increment = 301; scheduled 799 times
I/O Bound Process is exiting; actual_runtime = 250331; turnaroundtime = 16493064; ; avg_vruntime_increment = 300; scheduled 799 times
I/O Bound Process is exiting; actual_runtime = 262574; turnaroundtime = 16507959; ; avg_vruntime_increment = 305; scheduled 799 times
I/O Bound Process is exiting; actual_runtime = 242381; turnaroundtime = 16465333; ; avg_vruntime_increment = 299; scheduled 799 times
I/O Bound Process is exiting; actual_runtime = 242712; turnaroundtime = 16459952; ; avg_vruntime_increment = 301; scheduled 799 times
I/O Bound Process is exiting; actual_runtime = 242349; turnaroundtime = 16451656; ; avg_vruntime_increment = 299; scheduled 799 times
I/O Bound Process is exiting; actual_runtime = 267444; turnaroundtime = 16440341; ; avg_vruntime_increment = 299; scheduled 799 times
I/O Bound Process is exiting; actual_runtime = 264713; turnaroundtime = 16434290; ; avg_vruntime_increment = 299; scheduled 800 times
I/O Bound Process is exiting; actual_runtime = 241557; turnaroundtime = 16407936; ; avg_vruntime_increment = 299; scheduled 799 times
CPU Bound Process is exiting; actual_runtime = 3501372; turnaroundtime = 19297845; ; avg_vruntime_increment = 15931; scheduled 184 times
CPU Bound Process is exiting; actual_runtime = 3496595; turnaroundtime = 19337467; ; avg_vruntime_increment = 17010; scheduled 172 times
CPU Bound Process is exiting; actual_runtime = 3499260; turnaroundtime = 19294065; ; avg_vruntime_increment = 15842; scheduled 185 times
CPU Bound Process is exiting; actual_runtime = 3511762; turnaroundtime = 19370635; ; avg_vruntime_increment = 15955; scheduled 183 times
CPU Bound Process is exiting; actual_runtime = 3518796; turnaroundtime = 19381336; ; avg_vruntime_increment = 15766; scheduled 185 times
CPU Bound Process is exiting; actual_runtime = 3532300; turnaroundtime = 19414341; ; avg_vruntime_increment = 15889; scheduled 184 times
CPU Bound Process is exiting; actual_runtime = 3544032; turnaroundtime = 19494870; ; avg_vruntime_increment = 17044; scheduled 171 times
CPU Bound Process is exiting; actual_runtime = 3536204; turnaroundtime = 19451269; ; avg_vruntime_increment = 16056; scheduled 182 times
CPU Bound Process is exiting; actual_runtime = 3550096; turnaroundtime = 19435595; ; avg_vruntime_increment = 15994; scheduled 183 times
CPU Bound Process is exiting; actual_runtime = 3543083; turnaroundtime = 19502596; ; avg_vruntime_increment = 16098; scheduled 182 times
CPU Bound Process is exiting; actual_runtime = 3551561; turnaroundtime = 19543919; ; avg_vruntime_increment = 15900; scheduled 184 times
CPU Bound Process is exiting; actual_runtime = 3574668; turnaroundtime = 19538070; ; avg_vruntime_increment = 15958; scheduled 183 times
CPU Bound Process is exiting; actual_runtime = 3555146; turnaroundtime = 19515777; ; avg_vruntime_increment = 16010; scheduled 183 times
CPU Bound Process is exiting; actual_runtime = 3599439; turnaroundtime = 19550033; ; avg_vruntime_increment = 16019; scheduled 183 times
CPU Bound Process is exiting; actual_runtime = 3604279; turnaroundtime = 19585154; ; avg_vruntime_increment = 16063; scheduled 182 times
```

Observations:

1. The average vruntime increment of I/O bound processes is much less than that of CPU bound processes. This is because I/O bound processes are active only for short bursts in the timeslice. This result follows our hypothesis.

2. I/O bound processes are scheduled more often than CPU bound processes. This is following from the above obsevation as since the I/O bound processes have smaller vruntime increments their virtual runtimes stay in the left part of the tree. This implies that our CFS scheduler for xv6 ends up scheduling I/O bound processes more frequently, as it should.

3. We also see that for comparable turnaround times for each type of process, the I/O bound processes utilize the CPU 10 times less than the CPU bound processes, hence successfully simulating I/O and CPU bound processes.

Result:

Thus, we have simulated I/O bound and CPU bound processes and verified that our CFS scheduler schedules them in accordance with our hypothesis, that the I/O bound processes get scheduled more frequently.

TEST 2 - NICE VALUES:

In this test case we wish to see the behaviour of user-defined priorities (nice values). The processes with higher user-priority (lower nice values) will complete first.

Test Code Explanation:

Here we fork processes and the processes are assigned nice values from 19 to -20, in that order. The higher the nice value, the lower is the priority, and the lower is the weight of the process. The processes with lower priority begin execution first. This test case outputs turnaround-time, vruntime and actual-runtime for each process.

Code:

```
for(niceval=19; niceval >= -20; niceval--){
    pid=fork();
    if(pid<0){printf("fork failed\n");}
    if(pid==0){
        nice(niceval);
        for (k = 0; k < 200; k++){
            for (j = 0; j < 5000000; j++){}
        }
        exit(0);
    }
}
```

Hypothesis:

Since the processes with higher priority should complete first, the processes with higher priority (lower nice values) should have a lower turnaround time. The actual runtime for all the processes should be the same as they all do the same processing. Due to the actual runtime being equal, the vruntimes should be lower for higher priority processes as vruntimes are inversely propotional to weight.

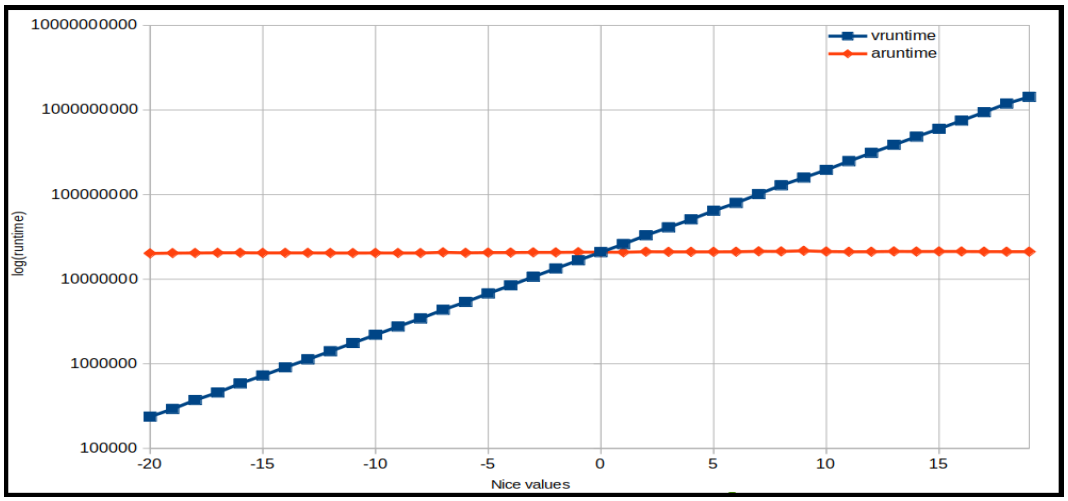
Running the test case:

We run this user program in xv6-shell as "$ tc2"

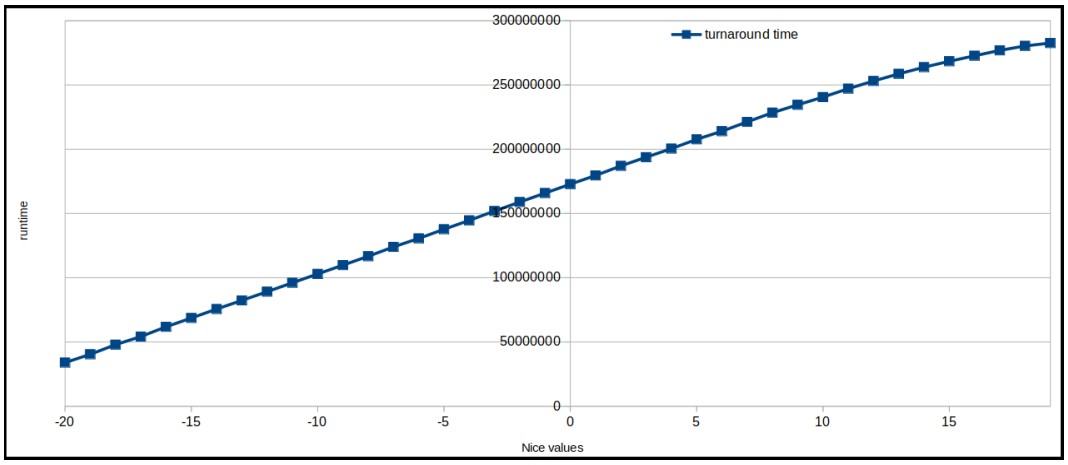
Output:

```
$ tc2
Process with Nice = -20 is exiting; vruntime = 249876; actual_runtime = 21466299; turnaroundtime = 36042503
Process with Nice = -19 is exiting; vruntime = 330362; actual_runtime = 22879467; turnaroundtime = 45363392
Process with Nice = -18 is exiting; vruntime = 407223; actual_runtime = 22301001; turnaroundtime = 52495226
Process with Nice = -17 is exiting; vruntime = 490732; actual_runtime = 22009688; turnaroundtime = 58800290
Process with Nice = -16 is exiting; vruntime = 633508; actual_runtime = 22294451; turnaroundtime = 67278426
Process with Nice = -15 is exiting; vruntime = 790867; actual_runtime = 22331052; turnaroundtime = 74834361
Process with Nice = -14 is exiting; vruntime = 976533; actual_runtime = 22036712; turnaroundtime = 81837013
Process with Nice = -13 is exiting; vruntime = 1260316; actual_runtime = 22890080; turnaroundtime = 90597142
Process with Nice = -12 is exiting; vruntime = 1651152; actual_runtime = 23968307; turnaroundtime = 100143007
Process with Nice = -11 is exiting; vruntime = 2136486; actual_runtime = 24755944; turnaroundtime = 109680814
Process with Nice = -10 is exiting; vruntime = 2751668; actual_runtime = 25559730; turnaroundtime = 119269989
Process with Nice = -9 is exiting; vruntime = 3598442; actual_runtime = 26690156; turnaroundtime = 129981660
Process with Nice = -8 is exiting; vruntime = 4711176; actual_runtime = 27987692; turnaroundtime = 141047329
Process with Nice = -7 is exiting; vruntime = 5845004; actual_runtime = 27932443; turnaroundtime = 150197249
Process with Nice = -6 is exiting; vruntime = 7353077; actual_runtime = 27999870; turnaroundtime = 159783983
Process with Nice = -5 is exiting; vruntime = 9221369; actual_runtime = 28063365; turnaroundtime = 169333767
Process with Nice = -4 is exiting; vruntime = 11504191; actual_runtime = 28069840; turnaroundtime = 178722714
Process with Nice = -3 is exiting; vruntime = 14440189; actual_runtime = 28059258; turnaroundtime = 188298046
Process with Nice = -2 is exiting; vruntime = 18084542; actual_runtime = 27998508; turnaroundtime = 197811050
Process with Nice = -1 is exiting; vruntime = 22505753; actual_runtime = 28061140; turnaroundtime = 207049213
Process with Nice = 0 is exiting; vruntime = 28374089; actual_runtime = 28372605; turnaroundtime = 216846173
Process with Nice = 1 is exiting; vruntime = 35143726; actual_runtime = 28145377; turnaroundtime = 225831363
Process with Nice = 2 is exiting; vruntime = 44131205; actual_runtime = 28233854; turnaroundtime = 235379996
Process with Nice = 3 is exiting; vruntime = 54850018; actual_runtime = 28182106; turnaroundtime = 244420197
Process with Nice = 4 is exiting; vruntime = 67694311; actual_runtime = 27972923; turnaroundtime = 253085093
Process with Nice = 5 is exiting; vruntime = 85826809; actual_runtime = 28088100; turnaroundtime = 262749066
Process with Nice = 6 is exiting; vruntime = 111109576; actual_runtime = 29524040; turnaroundtime = 273418566
Process with Nice = 7 is exiting; vruntime = 138580981; actual_runtime = 29108619; turnaroundtime = 282549277
Process with Nice = 8 is exiting; vruntime = 173458578; actual_runtime = 29147176; turnaroundtime = 291696854
Process with Nice = 9 is exiting; vruntime = 215674725; actual_runtime = 28866734; turnaroundtime = 300362408
Process with Nice = 10 is exiting; vruntime = 267642671; actual_runtime = 28763034; turnaroundtime = 308657910
Process with Nice = 11 is exiting; vruntime = 336839535; actual_runtime = 28630703; turnaroundtime = 317268216
Process with Nice = 12 is exiting; vruntime = 416483535; actual_runtime = 28481848; turnaroundtime = 324884918
Process with Nice = 13 is exiting; vruntime = 524478730; actual_runtime = 28693256; turnaroundtime = 332726015
Process with Nice = 14 is exiting; vruntime = 643634655; actual_runtime = 28295725; turnaroundtime = 339209456
Process with Nice = 15 is exiting; vruntime = 799782557; actual_runtime = 28128159; turnaroundtime = 345378901
Process with Nice = 16 is exiting; vruntime = 987987053; actual_runtime = 27990755; turnaroundtime = 350752798
Process with Nice = 17 is exiting; vruntime = 1245195241; actual_runtime = 27978738; turnaroundtime = 356349557
Process with Nice = 18 is exiting; vruntime = 1570196154; actual_runtime = 27799263; turnaroundtime = 360826050
Process with Nice = 19 is exiting; vruntime = 1894314645; actual_runtime = 27950652; turnaroundtime = 363885612
```

Observations:



1. All the processes are same and so have the same actual-runtimes.
2. The processes with smaller nice values (higher priority) have smaller virtual runtimes and the virtual runtime increases with increase in nice value. The virtual runtime and the actual runtime are equal at nice value = 0 (default priority).
3. We see that the [log(vruntime)] vs [nice] graph is a straight line which show that weight is exponential wrt nice values, which is correct as $weight_{nice} = 1024/(1.25)^{(nice)}$ (linux specifications).
4. The turnaround time for the process with nice = -20 is the least, and it increases with increase in nice values. This implies that the processes with smaller nice values finish execution faster, even though they started execution later. This implies that high priority processes are scheduled more frequently and thus finish execution quicker.



Result:

Thus, we have simulated priorities using nice values for different processes and verified that our CFS scheduler schedules processes in accordance with our hypothesis and that the higher priority processes complete execution early.