Operating Systems
CS3500
ProcReapers

Implementing CFS in xv6

Soumith Basina - CS18B005
Chakradhar Chokkaku - CS18B008

Date Submitted: Dec 9th, 2020

Instructor ..... Chester Rebeiro
Teaching Assistant ..... Nikhilesh Singh

1 Introduction

Completely Fair Scheduler (CFS) is a scheduling algorithm which aims at mimicking an ideal fair scheduler by giving all the processes in the runqueue a fair share of CPU time. This allows all processes a chance to execute without causing any starvation. This algorithm is being used in the Linux kernel since ver. 2.6.23 for scheduling normal and batch processes.

It achieves this by tracking process runtime using a virtual clock and having the runqueue in the form of a runtime ordered red-black tree. Interactive processes are also made more responsive because they have less vruntime which leads to the process being picked more often. The priority of a process can also be adjusted using the nice values, lesser nice values lead to the process having more weight which makes the vruntime increase slowly.

2 Why red-black trees?

Red-black trees are self-balancing binary search trees and they ensure insertion and deletion to be O(log n). It provides an efficient way of ordering processes as insertions and deletions are frequent. The smaller the key, the more to the left of the tree a node is. The scheduler always picks the left-most node to run. This can be done in O(1) by maintaining a pointer to the node. This will be discussed in later sections.

3 Scheduler Implementation

The changes are mainly made in the file `proc.c`.

First, the implementation of red-black trees, which is done in `rbtree.h`. The functions we use are `leftmostnode(), insert(), rb_delete()` which take of returning the left-most node, inserting into the tree and deleting from the tree respectively.

A few global variables are also added `sched_min_granularity, sched_latency, sched_period, sched_nice_to_weight[],` which are required in calculating dynamic timeslice for a process.

The runqueue is now `rbtree.h/struct red_black_tree` which has

- `int nproc` - number of processes in the tree.
- `uint64 min_vruntime` - vruntime of the left-most node, which is used to initialise vruntime in new processes.
- `struct proc* root` - pointer to the struct proc that is at the root.

The nodes in the tree are of `struct proc*`. The additions made in `proc.h/struct proc` are

- `enum COLOR color` - denotes the colour of the node, used in balancing the tree.
- `struct proc* par, left, right` - pointers to the parent, left and right nodes in the tree respectively.
- `uint64 vruntime` - keeps track of the vruntime of the process.
- `uint64 starttime` - stores the start time of running the process.
- `uint64 timeslice` - stores the dynamic timeslice allocated in the `scheduler()` function.
- `uint64 prev_vruntime` - stores the vruntime of the last time the process executed in the CPU.

These values are initiated in the `freeproc()` function.

`RUNNABLE` processes are inserted into the tree, those are to be scheduled. The tree is initialized first in `userinit()` alongside the init process with `new_red_black_tree()` function and then the process is inserted.

`fork()` creates a new process that is set to `RUNNABLE`, so that process is inserted.

In the `scheduler()` function, we pick the left-most process in the tree and then delete it from the tree. Then the `sched_period` is calculated by using the inequality `rq.nproc > sched_latency / sched_min_granularity`. If that is satisfied, `sched_period = sched_min_granularity * rq.nproc`, else `sched_period = sched_latency`. Timeslice calculation is done in the scheduler before the context switch. The value is stored in `timeslice`. Set `starttime` to

the current value in the `mtime` register and switch the context.

In the `yield()` function, the CPU gives up the process. As the process is set to `RUNNABLE`, the process is inserted into the tree.

At every timer interrupt, `trap.c/usertrap()` calls `checktimeslice()` which adds `mtime - starttime` multiplied by `1024 / sched_nice_to_weight[p->nice+20]` to `vruntime` and then checks if the execution time exceeded the timeslice allotted. If so, it then calls `yield()` which starts the context switch to the next process. The more the weight of the process, the lesser the time added to the vruntime, giving it more CPU time.

In `wakeup()` and `kill()` functions, if the process was `SLEEPING`, the process is set to `RUNNABLE` and it is then inserted into the tree.

4 Contributions

- **Code** Red-Black trees Implementation by Chakradhar, Scheduler Implementation by Soumith.
- **Report** by Chakradhar and Soumith.
- **Presentation** by Chakradhar and Soumith.