Abstract

In this paper, we propose a new approach to significantly optimize some scheduling criteria, context switches and turnaround times in this context, when running multithreaded processes concurrently. Current proportional sharing schedulers allocate CPU time based on the weights of running threads in the system and don’t take into consideration the greedy behavior of multithreaded processes (processes with more threads receive more aggregate CPU time from the scheduler relative to processes with fewer threads). In order to minimize turnaround times and context switches of running multithreaded processes in simultaneous multithreaded (SMT) architectures, we investigate the effect of adjusting the weights of running sibling threads (threads forked from the same process) using novel proportional sharing scheduler, Thread Weight Readjustment Scheduler (TWRS), a proportional share CPU scheduler designed for multithreaded processes, which aims to reduce undesirable events (e.g. context switches) and turnaround times. TWRS provides a practical solution for multitasking operating systems because it operates in concert with existing kernels. We have implemented TWRS in Linux 2.6.24-1, which represents the most prevalent scheduler design (i.e. Completely Fair Scheduler (CFS)). Our evaluation shows that our scheduler minimizes context switches and turnaround time.

Keywords: Multithreaded processes, multitasking, CFS, fairness, turnaround time.

1 Introduction

A. General Overview

Proportional Share Scheduler (PSS), also sometimes referred to as a fair-share scheduler might try to guarantee that each process obtains a certain percentage of CPU time. PSS is a type of scheduling which preallocates certain amount of CPU time to each process. PSS is based upon maintaining fairness between competing processes. PSSs are often primarily evaluated based on the level of fairness that they can provide [28]. Other evaluation criteria depend on what the scheduler is designed for.

Cooperative multitasking and preemptive multitasking are two flavors in which multitasking operating systems come in. Linux, like all Unix variants and most modern operating systems,
implements preemptive multitasking. In preemptive multitasking, the scheduler decides when a process is to cease running and a new process is to begin running. The act of involuntarily suspending a running process is called preemption. The time a process runs before it is preempted is usually predetermined, and it is called the *timeslice* of the process. The timeslice, in effect, gives each runnable process a slice of the processor’s time [4].

**B. Scheduling and its Criteria**

The scheduling problem can be stated shortly as: *which* thread should be moved to *where*, *when* and *for how long* [4][6]. In computer science, a scheduling algorithm is the method by which threads, processes or data flows are given access to system resources (e.g. processor time). This is usually done to load balance a system effectively or achieve a target quality of service [5][6]. Software known as a scheduler and dispatcher carry out this assignment.

Scheduling algorithms have been found to be NP-complete in general form (i.e., it is believed that there is no optimal polynomial-time algorithm for them [12][13]). The give-and-take situation comes from the many definitions of good scheduling performance, such that improving performance in one sense hurts performance in another. Some improvements to the Linux scheduler help performance all-around, but such improvements are getting more and harder to come by [16].

Many criteria have been suggested for comparing CPU scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following: CPU utilization, throughput, fairness, execution time, turnaround time, context switches, completion time, waiting time, and others [7][8][9].

Context switches and turnaround time are two of the most important criteria in designing any operating system scheduler. The context switches time is a pure overhead because the system does no useful work while switching. Reducing such overhead leads to minimizing turnaround time.

**C. Paper Organization**

The rest of this paper is structured as follows: section 2 discusses the contribution of this paper. Current Linux kernel scheduler, CFS, is discussed in section 3. Section 4 presents the problem statement. In section 5, we discuss the related research. Section 6 discusses TWRS. The experimental setup and scheduling modes are given in section 7. Section 8 presents the evaluations.

**2 Research Contribution of this Paper**

In this paper, we proposed and evaluated TWRS, which is a proportional share CPU scheduling, intending to minimize the context switches and turnaround times of processes. In proportional share algorithm every thread has a weight, and thread receives a share of the available resources proportional to its weight [18].

In this work, a modification is implemented to CFS. This modification is based on changing threads’ weights of sibling threads created in the same process and assigning a specific time slice to each of these sibling threads.

We have implemented our scheduler in the Linux kernel and experimentally demonstrated the improvement of our scheduler over the current scheduler, CFS, using multithreaded programs in Sysbench benchmark [27]. Our experimental results show that TWRS scheduler minimizes context switches and turnaround times.
3 Linux Kernel Schedulers

A. Completely Fair Scheduler

Completely Fair Scheduler (CFS) is introduced by Ingo Molnár to replace the O(1) scheduler [17][18][19][20]. This scheduler was designed to provide good interactive performance while maximizing overall CPU utilization [21]. It also strives to provide fairness in every process without sacrificing the interactivity performance [22]. This is done by giving a fair amount of CPU time to each process proportional to its priority. This method is also called proportional share algorithm, where a share is allocated for each process, which is associated with the process’s weight.

CFS is successor of the O(1) scheduler and one of the most distinct changes from previous scheduler is the policy of setting the priority for each thread. In CFS, the scheduler counts the execution time of each thread and calculates the priority as vruntime (virtual runtime). CFS sets the higher priority for the threads with less vruntime. The run queue of CFS is composed of Red-Black tree, where each node represents the thread and the value of each node represents the vruntime of each thread [14].

4 Problem Statement

This section discusses the main problem the current scheduler faces.

A. Overview

Each process and thread is a task in the eyes of the Linux scheduler. CFS uses thread fair scheduling algorithm, which allocates CPU resources between running threads in the system not between the running processes. In the current scheduler, CFS, when a new process is created, it appears as a thread, where both PID (Process Identification) and TGID (Thread Group Identification) are the same (new) number, although when a thread starts another thread, that new thread gets its own PID, so the scheduler can schedule it independently, and inherits its TGID from its parent as shown in Figure 1.

Therefore, CFS scheduler does not distinguish between threads and processes, and that way, the kernel can happily schedule threads independent of which process they belong to. Each forked thread is assigned a weight which determines the share of CPU bandwidth that thread will receive. Greedy users could take advantage by spawning more additional threads in order to obtain larger CPU resources.

In other words, we can summarize this statement as: the default Linux scheduler is process-agnostic and allows for greedy behavior, where processes with more threads may receive
more aggregate CPU time from the scheduler relative to processes with fewer threads. That is, the
scheduler schedules threads only and does not take process membership into account, or in-
ter-process fairness.

From the point of view of users, one of the important criteria is how long scheduler takes to
eexecute their process. Turnaround time, total time between submission of a process and its com-
pletion, is one of the scheduling criteria that the scheduler is concerned mainly with [16]. Turn-
around time depends on the size of the time slice allotted to running processes, and minimizing
turnaround time is one of the objectives that the scheduler strives to achieve. Previous work fo-
cused on the fairness criterion, however, in our work we focus on turnaround time and context
switches.

5 Related Research

Generalized Processor Sharing (GPS) [1][26] is an idealized scheduling algorithm that achieves
perfect fairness, and all schedulers use it as a reference to measure fairness. A scheduler is per-
fectly fair if it allocates CPU time to threads in exact proportion to their weights.

Chandra [2] presented Surplus Fair Scheduling (SFS), a proportional-share CPU scheduler
designed for symmetric multiprocessors. Chandra first showed that the infeasibility of certain
weight assignments in multiprocessor environments results in unfairness or starvation in many
existing proportional-share schedulers. They presented a novel weight readjustment algorithm to
translate infeasible weight assignments to a set of feasible weights. They showed that weight
readjustment enables existing proportional-share schedulers to significantly reduce, but not
eliminate, the unfairness in their allocations.

Chee [10] proposed an algorithm based on weight readjustment of the threads created in the
same process. This algorithm, Process Fair Scheduler (PFS), is proposed to reduce the unfair
allocation of CPU resources in multithreaded environment. Chee assumed that the optimal num-
ber of threads, best number to create in a process in order to have the best performance in a
multi-processing environment, equals to the number of available cores. PFS changes the weight of
thread according to the equation:

\[
weight(\text{thread}) = \frac{weight(\text{process})}{\alpha}
\]

where \( \alpha \) equals the number of threads created in the process.

A modification of PFS algorithm has been proposed to overcome the limitation of PFS by im-
plementing Thread Fair Preferential Scheduler (TFPS) [23]. TFPS shall give the greedy threaded
process (i.e. process tries to dominate most CPU time) the same amount of CPU bandwidth as
optimally threaded process, and both of their time slices are larger than the single-threaded pro-
cess.

Inter-Core Time Aggregation Scheduler (IAS) [1] executes two scheduling policies at the same
time. The first scheduling policy is the Time Aggregation Scheduler (TAS) [3][11], which exec-
cutes sibling threads in a row on a single core. The second scheduling policy is the inter-core
aggregation, which executes sibling threads on different cores at the same time by dividing each
core into master and slave cores. IAS lets every core cooperatively aggregate sibling threads by
making slave cores follow the aggregation on master core. The basic idea of TAS and IAS
schedulers is to dynamically give a priority bonus to the sibling thread of the currently executed
threads. The priority of a thread is higher when the vruntime of the thread is smaller. Therefore,
the priority bonus for the scheduler works to reduce the vruntime of the sibling thread.
6 Thread Weight Readjustment Scheduler

This section gives an overview of TWRS, and discusses its mechanism.

A. Overview

TWRS is a kernel-level thread scheduler to enhance system’s performance running multithreaded processes by readjusting the weights of threads forked from the same process to significantly achieve better some scheduling criteria. TWRS works on top of an existing scheduler that uses run queues for per-CPU, such as Linux 2.6. As its name suggests, TWRS depends on proportionally distributed CPU time between threads by changing their weights. We will explain the policy of allocation CPU time to running threads in the next section.

B. TWRS and Thread Allocation of CPU Time

Each thread is assigned a weight which determines the share of CPU bandwidth that thread will receive [10]. The weight of thread \( i \) is given by,

\[
\text{w}_i = \{ -20 \leq \text{nice} \leq 19 \cdot \text{NICE}_0 \text{ LOAD} \cdot (1.25^{\text{nice}}) \}
\]

nice denotes the thread priority assigned by the user, and the value of ‘nice’ lies within [-20,19]. The default value of ‘nice’ is ‘0’ in which case \( w_i = \text{NICE}_0 \text{ LOAD} \), and its value is assumed to be 1024.

The share in a time interval \( [t_1, t_2] \) of a runnable thread \( i \) is a ratio of its weight to the sum of weights of all active threads in the run queue. This is computed as:

\[
\text{Share}_i(t_1, t_2) = \frac{\sum_{j \in R} w_j}{\sum_{j \in R} w_j} \cdot (t_2 - t_1).
\]  

(1)

\( R \) is the set of all runnable threads that are currently in run queue of a core. \( w_i \) is the weight of the thread, it is mapped from its nice value in \( \text{prio_to_weight}[] \), (defined as a variable in kernel/sched.c file) [24], and each nice value has its respective weight.

The time-slice that a thread \( i \) should receive in a period of time is given by:

\[
\text{slice}_i = \text{share}_i \times \text{period}
\]

(2)

where period is the time quantum the scheduler tries to execute all threads, by default this is set to 20ms [15][25].

If the number of runnable processes does not exceed

\[
\frac{\text{sched\_latency\_ns}}{\text{sched\_min\_granularity\_ns}}
\]

then

\[
\text{period} = \text{sched\_latency\_ns}
\]

otherwise,

\[
\text{period} = \text{number\_of\_running\_tasks} \times \frac{\text{sched\_min\_granularity\_ns}}{\text{sched\_latency\_ns}}
\]

By default,

\[
\text{sched\_latency\_ns} = 20\text{ms}
\]

and

\[
\text{sched\_min\_granularity\_ns} = 4\text{ms}
\]
sched_min_granularity_ns is a scheduler tuneable, this tuneable decides the minimum time a thread will be allowed to run on CPU before being preempted out. sched_latency_ns is a scheduler tuneable. sched_latency_ns and sched_min_granularity_ns decide the scheduler period.

In our consideration, the scheduler counts the number of total threads in the CPU and the number of sibling threads. The weights of the threads will be changed according to the next equation;

\[
\text{new_weight} \rightarrow \text{new_load_weight} = \text{new_weight} \quad (3)
\]

where \(\text{new_weight}\) is the new weight of the current thread and calculated from the equation:

\[
\text{new_weight} = p->\text{se.load.weight} \times (\text{prcsr}->\text{totl_thrds} - \text{curr_proc}->\text{nr_thrds}) \quad (4)
\]

where \(p->\text{se.load.weight}\) is the weight of the current thread, \(\text{prcsr}->\text{totl_thrds}\) is the total number of threads in the current processor and \(\text{curr_proc->nr_thrds}\) is the number of threads in the current process.

7 Experimental Setup

A. Underlying Platform

TWRS can be easily integrated with an existing scheduler based on per-CPU run queues. We implemented TWRS in Linux version 2.6.24-1 which based on CFS. The specification of our experimental platform is shown in Table I.

<table>
<thead>
<tr>
<th>H/W</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz</td>
</tr>
<tr>
<td>CPU cores</td>
<td>8</td>
</tr>
<tr>
<td>Memory</td>
<td>8186756 kb</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S/W</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel name</td>
<td>Linux</td>
</tr>
<tr>
<td>Kernel version number</td>
<td>2.6.24</td>
</tr>
<tr>
<td>Machine hardware name</td>
<td>x86_64 (64 bit)</td>
</tr>
<tr>
<td>version of Linux</td>
<td>CentOS release 5.10 (Final)</td>
</tr>
</tbody>
</table>

B. Scheduling Modes

To assess the performance of the modified kernel, we evaluated using 2 multithreaded benchmarks on a SMP system. The benchmarks run under two distinct scheduling modes: (1) The default scheduling in the Linux kernel and (2) The modified kernel.

- In the default scheduling mode, the benchmarks run on the original operating system where the scheduler is allowed to make scheduling decisions. No extra parameter is given to the scheduler to change its native scheduling algorithm.
- The second mode is accomplished in the new modified kernel, where the scheduler operates on the new scheduling policy to give new time slices to running threads.

Because we need to benchmark the scheduler performance, we choose only two test modes, threads and cpu, amongst all test modes available in Sysbench. In threads test a scheduler has a large number of threads competing for some set of mutexes.

In cpu test, each request consists in calculation of prime numbers up to a value specified by
the --cpu-max-primes option.

We evaluated TWRS in two major scenarios, S0 and S1 in each mode as explained in Table II. The following test was conducted with no other computation intensive applications running.

- In S0, we intended to show that two concurrently executing instances of the same program, threads program in Sysbench, one instance with a fixed number of threads and the other variably from N to 64 threads, where N is greater than the number of threads in the fixed instance. Both programs were initiated at the same time and were initiated through a shell script where these two programs were executed with the same nice value 0. We repeat this simulation with a different number of threads in the fixed one.

- In S1, we intended to show that two concurrently executing instances of different programs, threads and cpu programs in Sysbench, threads program run with a fixed number of threads and cpu program variably from N to 64 threads, where N is greater than the number of threads in the fixed instance. Both programs were initiated at the same time and were initiated through a shell script where these two programs were executed with the same nice value 0. We repeat this simulation with a different number in the fixed program (threads).

### TABLE II. S0 AND S1 SCENARIOS

<table>
<thead>
<tr>
<th></th>
<th>S0</th>
<th></th>
<th></th>
<th>S1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of threads in Fixed program &quot;threads&quot;</td>
<td>No. of threads in Varied program &quot;threads&quot;</td>
<td></td>
<td>No. of threads in Fixed program &quot;threads&quot;</td>
<td>No. of threads in Varied program &quot;cpu&quot;</td>
<td></td>
</tr>
<tr>
<td>S0.1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>32</td>
<td></td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>32</td>
<td>64</td>
<td></td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>S0.2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>32</td>
<td></td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>32</td>
<td>64</td>
<td></td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>S0.3</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>32</td>
<td></td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>32</td>
<td>64</td>
<td></td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>

### 8 Experimental Results

To demonstrate the effectiveness of TWRS, we present some experimental data quantitatively comparing TWRS performance against the popular scheduler CFS considered on different combinations of number of threads for each scenario. We repeated this simulation many times for each sub-scenario and show the result in terms of number of context switches and average turnaround time. Our results show a significant improvement in terms of minimizing turnaround time and context switches. We divided our results into six sub-scenarios according to the previous consideration in Table II. For each sub-scenario, the two programs were run concurrently 20 times and the average values were taken. Figures 2 and 3 show the results of turnaround comparison in S0 and S1 respectively, and figures 4 and 5 show the results of context switches comparison in S0 and S1 respectively.

### Concluding Remarks
In this paper a novel scheduler is proposed to distribute CPU time proportionally between threads according to their weights. We proposed TWRS which preallocates certain amount of CPU time to each thread of the multithreaded processes. We focused in this work on context switches and turnaround time. The scheduler was implemented and evaluated under specific hardware and software environment. We used Sysbench benchmark in our test and run under two distinct scheduling modes; the default scheduling in the Linux kernel and the modified kernel. Our results showed that the proposed scheduler achieves better results in minimizing turnaround time and context switches.

Acknowledgments

This work was partly supported by KAKENHI, Grant-in-Aid for Scientific Research(C) 25330084.

References

Effect of Thread Weight Readjustment Scheduler on Scheduling Criteria


Copyright © by IIAI. Unauthorized reproduction of this article is prohibited.


