Towards Maximizing Throughput for Multithreaded Processes in Linux

Samih M. Mostafa\(^1,2\) and Shigeru KUSAKABE\(^1\)

\(^1\)Graduate School of Information Science and Electrical Engineering, Kyushu University
744 Motooka Fukuoka, 819-0395, Japan.
\(^2\)Math. Dept., Faculty of Science, South Valley University, Qena, Egypt.
E-mail: samih@nanotsu.ait.kyushu-u.ac.jp
kusakabe@ait.kyushu-u.ac.jp

ABSTRACT

CPU scheduler is a very important key concept in operating system which affects scheduling objectives and criteria. Choosing or modifying scheduling policy for running threads depends on predefined and specific objectives and criteria. In this paper, we investigate the effectiveness of thread weight readjustment scheduler (TWRS) for multithreaded processes in multitasking operating systems from the view point of throughput. TWRS is a proportional share CPU scheduler designed for scheduling multithreaded processes. In this work, we make change kernel performance significantly by modifying just few key parameters. We propose TWRS which preallocates certain amount of CPU time to each thread of the multi-threaded processes. The scheduler was implemented and evaluated under specific hardware and software environment. According to our evaluation results, our scheduler is promising to optimize some scheduling criteria.

KEYWORDS

Throughput, Multitasking, Simultaneous Multithreading, CFS Linux scheduler, Thread-Level Parallelism (TLP).

1 INTRODUCTION

1.1 Overview

In order to design a scheduling algorithm, it is necessary to have some idea of what a good algorithm should do. Choosing or modifying scheduling policy for running threads depends on predefined and specific objectives and criteria. Some goals depend on the environment (batch, interactive, or real time), but there are also some that are desirable in all cases. For multitasking systems, different goals apply [29]. The most important one is maximizing throughput (e.g. number of processes that complete their execution per time unit).

A multitasking operating system is one that can simultaneously interleave execution of more than one process. On single processor machines, this gives the illusion of multiple processes running concurrently. On multiprocessor machines, such functionality enables processes to actually run concurrently, in parallel, on different processors. Multitasking operating systems come in two flavors: cooperative multitasking and preemptive multitasking. 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.

Allowing multiple threads to run concurrently in a processor and share resources, CPU time in this context, is a recent architectural technique that improves resource utilization. Simultaneous multithreading (SMT) is one of the architectural techniques has become widespread concern in recent years that improves resource utilization [1]. Simultaneous Multithreading (SMT) architectures execute instructions from multiple streams of execution (threads) each cycle to increase instruction level parallelism [25, 26, 27]. Parallelism is one of the most important issues in multicore systems because it offers great performance, and future computing is progressing
towards use of multicore machines. One of the efficient and scalable way to benefit from multicore architecture is using of multi-threading. Multi-threading is the process of executing multiple threads concurrently on cores and it is a widespread programming and execution model that allows multiple threads to exist within the context of a single process and these threads share the process's resources, but are able to execute independently.

Most software applications that run on modern computers are multithreaded. An application typically is implemented as a separate process with several threads of control. The benefits of multithreading can be broken down into: responsiveness, resource sharing, economy and scalability. Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. A more recent, similar trend in system design is to place multiple computing cores on a single chip. Each core appears as a separate processor to the operating system. Whether the cores appear across CPU chips or within CPU chips, we call these systems multicore or multiprocessor systems. Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency [16].

An operating system is a software application that acts as an interface between user and the computer hardware. Its major responsibilities are to manage and ensure proper operations of the hardware resources. At any time, there can be more than one runnable threads/processes demanding service from the Central Processing Unit (CPU). In order to handle the oversubscription of threads/processes, an operating system scheduler is required to ensure each thread/process obtains its share of CPU time as fair as the scheduling algorithm can guarantee [3].

The need for a scheduling algorithm arises from the requirement for most modern systems to perform multitasking (execute more than one thread/process at a time) and multiplexing (transmit multiple flows simultaneously). Scheduling is a key concept in computer multitasking, multiprocessing operating system and real-time operating system designs. Due to the growing availability of chip multiprocessors (CMP), software applications are encouraged to be designed using multiple threads so that the benefit of thread-level parallelism (TLP) can be exploited [22, 28]. The operating system scheduler is designed to allocate system resources, CPU time, proportionally to all processes.

### 1.2 Motivation

The scheduling problem can be stated shortly as: which thread should be moved to where, when and for how long [5, 9]. 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 [10, 11]. Software known as a scheduler and dispatcher carry out this assignment.

The scheduler is the basis of a multitasking operating system such as Linux. By deciding which process runs next, the scheduler is responsible for best utilizing the system and giving users the impression that multiple processes are executing simultaneously.

Managing the timeslice enables the scheduler to make global scheduling decisions for the system. It also prevents any one process from monopolizing the processor. On many modern operating systems, the timeslice is dynamically calculated as a function of process behavior and configurable system policy.

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 many definitions of good scheduling performance often lead to a give-and-take situation, 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.

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a
particular situation, we must consider the properties of the various algorithms. 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, waiting time, and others [16]. Kernel hackers try to make improvements in Linux scheduling algorithm, this is because Linux scheduling algorithm is self-contained and relatively easy to follow [4]. In this work, we make change kernel performance significantly by modifying just a few key parameters. This work investigates the effect of changing weights of sibling threads (threads created in the same process) and evaluates this modification in terms of waiting time, turnaround time and throughput.

1.3 Linux Kernel Scheduler

A) O(1) Scheduler

A version of Linux scheduler was developed with the release of kernel 2.6. This new scheduler is called the O(1) scheduler[4, 5]. The name was chosen because the scheduler's algorithm required constant time to make a scheduling decision, regardless of the number of processes. The algorithm used by the O(1) scheduler relies on active and expired arrays of processes to achieve constant scheduling time. Each process is given a fixed time slice, after which it is preempted and moved to the expired array. Once all the processes from the active array have exhausted their time slice and have been moved to the expired array, an array switch takes place. This switch makes the active array the new empty array, while the expired array becomes the active array. Each array in the run-queue represents 140 different priority levels: from 0 (highest) to 99 (lowest) use real-time policy. Priority levels from 100 (highest) to 139 (lowest) represent time-shared processes under the SCHED_NORMAL scheduling [18, 19].

B) Completely Fair Scheduler

Completely Fair Scheduler (CFS) is introduced by Ingo Molnár to replace the O(1) scheduler [6,7,14,15]. This scheduler was designed to provide good interactive performance while maximizing overall CPU utilization [8]. It also strives to provide fairness in every process without sacrificing the interactivity performance [20]. 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. 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 [4]. Linux’s CFS scheduler does not directly assign timeslices to processes. Instead, in a novel approach, CFS assigns processes a proportion of the processor. On Linux, therefore, the amount of processor time that a process receives is a function of the load of the system. CFS will run each process for some amount of time, round-robin, selecting next the process that has run the least. Rather than assign each process a timeslice, CFS calculates how long a process should run as a function of the total number of runnable processes. Instead of using the nice value to calculate a timeslice, CFS uses the nice value to weight the proportion of processor a process is to receive. Each process then runs for a “timeslice” proportional to its weight divided by the total weight of all runnable threads [5].

1.4 Problem Statement

The Linux Completely Fair Scheduler (CFS) uses thread fair scheduling algorithm, this algorithm allocates CPU resources based on the number of threads in the system not within the running processes. In the current scheduler (CFS), when a new process is created, it appears as a thread where both PID and TGID 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) but it inherits its TID 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 what process they belong to. Each forked thread is assigned a weight which determines the share of CPU bandwidth that thread will receive.

The forked threads from the same process inherit their weights from their parent and consequently the same time slice will be assigned to each thread. Greedy users could take advantage by spawning more additional threads in order to obtain larger CPU resources when there are other processes with lower number of running threads, this has a negative effect on scheduling criteria because these criteria are affected by the time assigned to the additional spawned threads.

**1.5 Research Contribution of this Paper**

The first problem in selecting a CPU-scheduling algorithm for a particular system is defining the criteria to be used. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called *throughput* [16].

To provide a more in depth description of our contribution, we first formulate more precisely the state of the main goal with regard to other criteria and then describe in detail how to achieve that state in the kernel. The following definitions define the relation between the mentioned criteria.

**Definition 1. Turnaround Time**: total time between submission of a process and its completion.

\[
\text{Turnaround Time} = \text{Completion Time} - \text{Arrival Time}
\]  

**Definition 2. Waiting Time**: amount of time a process has been waiting in the ready queue.

\[
\text{Waiting Time} = \text{Turnaround Time} - \text{Execution Time}
\]

Therefore, the problem can be represented as:

\[
\text{Maximize } \text{Throughput} \quad s.t. \quad \text{equations 1 and 2}
\]

Turnaround time and waiting time depend on the size of the time slice assigned to running processes, and minimizing these two criteria is one of the objectives that the scheduler strives to achieve. In this paper, we proposed and evaluated a thread weight readjustment scheduler (TWRS) which is a proportional share CPU scheduling intended to reduce the greedy behaviour of processes in order to minimize waiting time and turnaround time. In proportional share algorithm every thread has a weight, and thread receives a share of the available resources proportional to its weight [17].

In this work, a modification is implemented to CFS. The modification is based on changing sibling threads’ weights 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 TWRS over the current scheduler, CFS, using benchmarks. Our experimental results show that TWRS scheduler provides better results. The rest of this paper is structured as follows: in section II, we discuss the related research. Section III discusses the proposed scheduler TWRS. The experimental setup and scheduling modes are given in section IV. Section V presents the experimental results.
2 RELATED RESEARCH

Chee [22] proposed an algorithm based on weight readjustment of the threads created in the same process. This algorithm, PFS, is proposed to reduce the unfair allocation of CPU resources in multi-threaded environment. Chee assumed that the optimal number of threads, best number to create in a program in order to have the best performance in a multi-processing environment, equals to the number of available cores. A limitation of this algorithm is that all processes executing at the same nice value will receive the same amount of time slice.

A modification of PFS algorithm has been proposed to overcome this limitation by implementing TFPS [24]. TFPS shall give the greedy threaded program (e.g. program tries to dominate most CPU time) the same amount of CPU bandwidth as optimally threaded program, and both of their time slices are larger than the single-threaded program.

According to PFS, all processes executing at the same nice value will receive the same time slice regardless the number of threads, this is considered as a defect in this algorithm. And because each thread will receive the same amount of time slice, the system may suffer from overhead due to the number of context switches resulting from executing greedy program. According to PFS and TFPS, multi-threaded programs are not rewarded; this is because in PFS; the time slice of greedy program is the same as the amount of time slice assigned to the process which has the lowest number of threads, and in TFPS; the time slice of greedy program is the same as amount of time slice allotted to the optimally-threaded process.

The basic idea of TWRS is similar to Chee’s, even though the working environment and the method of readjusting the weights are different. As we will see in section III that our scheduler readjusts the thread’s weight with regard to the number of all running threads in the current CPU and does not restrict the amount of time slice of the multithreaded process with other processes.

3 THREAD WEIGHT READJUSTMENT SCHEDULER

3.1 Overview

TWRS is a kernel-level thread scheduler to enhance performances of multi-threaded programs by focusing on weight readjustment 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.

3.2 TWRS and Thread Allocation of CPU Time

Each thread is assigned a weight which determines the share of CPU bandwidth that thread will receive. The share allocated to a thread is a ratio of its weight to the sum of weights of all active threads in the run queue. This is given by the equation:

\[
\text{share} = \frac{\text{se->load.weight}}{\text{cfs->load.weight}}
\]

se->load.weight is the weight of the thread, it is mapped from its nice value in prioc_to_weight[ ] (defined as a variable in kernel/sched.c file) [21], and each nice value has its respective weight. cfs->load.weight is the total weight of all threads under CFS run queue. The time-slice that a thread should receive in a period of time is given by:

\[
\text{slice} = \frac{\text{se->load.weight}}{\text{cfs->load.weight}} \times \text{period}
\]

period is the time quantum the scheduler tries to execute all threads, by default this is set to 20ms [2]. If number of runnable processes does not exceed (sched_latency_ns / sched_min_granularity_ns), then period = sched_latency_ns, otherwise, period = number_of_running_tasks \times sched_min_granularity_ns. By default, sched_latency_ns = 20ms and sched_min_granularity_ns = 4ms [18].

In our consideration we will count the number of total threads in the CPU and the number of threads forked from the same process. Weights of the
threads will be changed according to the next equation:

\[ \text{thrd} \rightarrow \text{se.load.weight} = \text{new_weight} \]

new_weight is the new weight for the current thread and calculated from the equation:

\[ \text{new_weight} = p \rightarrow \text{se.load.weight} \times \rightarrow \text{totl.thrds} - \text{curr.proc->nr.thrds} \]

p->se.load.weight is the weight of the current thread, prcsr->totl.thrds is the total number of threads in the current processor and curr_proc->nr_thrds is the number of threads in the current process.

3.3 Illustrative Example

We show in Figure 2 an example to clarify previous explanation. Figure 2 shows the run queue of CFS, the circle in this figure represents the thread, the color of red or black in Red-black tree is ignored. The threads with same pattern refer to sibling threads; e.g. threads created in the same process. We assume in this example two processes A and B, process A has five threads and process B has two threads and all threads have the same nice value 0 and therefore the same weight. The numbers in the threads show the vruntime, and threads in the run queue are in an ascending order according to their vruntime.

![Figure 2. Two multithreaded processes with different numbers of threads.](image)

Ideally, each thread should get \(2.857\times 20 = 20 \times 1024 / (7 \times 1024)\). But this is less than sched_min_granularity_ns, which decides the minimum time a thread will run. At the time of writing this paper, sched_min_granularity_ns is set to 4000000ns in Linux 2.6.24. So in this case of seven threads; the period becomes 28ms and each thread will run for 4ms before it is preempted. Users could take advantage by spawning many additional threads in order to obtain more CPU resources.

4 EXPERIMENTAL SETUP

4.1 Hardware and Software

TWRS can be easily integrated with an existing scheduler based on per-CPU run queues. To demonstrate its versatility, we have implemented TWRS in Linux version 2.6.24-1 which based on CFS. The specification of our experimental platform is shown in Table 1.

| Table 1. Specification of our experimental platform |
|-----------------|-----------------|
| **H/W**         | **S/W**         |
| Processor       | Processor       |
| Intel(R) Core(TM)2 Duo CPU 7250 @ 2.00GHz | Linux |
| cpu MHz         | 800.000         |
| cpu cores       | 2               |
| Memory          | 2565424 kb      |
| **Kernel name** | Version         |
| Linux           | Linux           |
| **Kernel version number** | CentOS release 5.10 (Final) |

4.2 Scheduling Modes

To assess the performance of the modified kernel, we evaluated using 2 multi-threaded benchmarks in a multitasking 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 [23]. 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-prime option. We evaluated TWRS in two major scenarios, S0 and S1 in each mode as explained in Table 2. The following test was conducted with no other computation intensive applications running.

Table 2. S0 and S1 scenarios

<table>
<thead>
<tr>
<th></th>
<th>S0</th>
<th></th>
<th>S1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of threads in fixed program “threads”</td>
<td>No. of threads in varied program “threads”</td>
<td>No. of threads in fixed program “threads”</td>
<td>No. of threads in varied program “cpu”</td>
</tr>
<tr>
<td>S0.1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>S0.2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>S0.3</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>S1.1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>S1.2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>S1.3</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>

- 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 32 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 repeated 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 32 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 repeated this simulation with a different number in the fixed program (threads).

5 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 the average waiting time, turnaround time and throughput. Our results show a significant improvement in terms of minimizing waiting and turnaround times and maximizing throughput. For each sub-scenario, the two programs were run concurrently 20 times and the average values were taken for 60 sec. Figures 3, 4 and 5 show the results in S0 and figures 6, 7 and 8 show the results in S1.

6 CONCLUDING REMARKS

In this paper, a novel scheduler is proposed to distribute CPU time proportionally between threads according to their weights. We proposed a thread weight readjustment scheduler (TWRS) which preallocates certain amount of CPU time to each thread of the multi-threaded processes. Fairness criterion used in previous work is an important metric; however there are other metrics are used to evaluate operating system scheduling algorithms. In our work, we focus on another scheduling criterion (throughput) because maximizing throughput contributes in optimizing other criteria such as reducing energy consumption which will be our future work. We focused in this work on waiting time, turnaround time and throughput as they are important scheduling criteria the scheduler should meet. 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 is promising to minimize waiting time and turnaround time and maximize throughput.
Figure 3. Turnaround time comparison; fixed number of threads of “threads” program vs. N threads of “threads” program in each sub-scenario in S0.

Figure 4. Waiting time comparison; fixed number of threads of “threads” program vs. N threads of “threads” program in each sub-scenario in S0.

Figure 5. Throughput comparison; fixed number of threads of “threads” program vs. N threads of “threads” program in each sub-scenario in S0.

Figure 6. Turnaround time comparison; fixed number of threads of “threads” program vs. N threads of “cpu” program in each sub-scenario in S1.

Figure 7. Waiting time comparison; fixed number of threads of “threads” program vs. N threads of “cpu” program in each sub-scenario in S1.

Figure 8. Throughput comparison; fixed number of threads of “threads” program vs. N threads of “cpu” program in each sub-scenario in S1.
7 REFERENCES


78