Towards Achieving Fairness in the Linux Scheduler

Chee Siang Wong, Ian Tan, Rosalind Deena Kumari
Faculty of Information Technology,
Multimedia University,
63100 Cyberjaya, Selangor, Malaysia
+60-13-3996255, +60-3-8312-5235, +60-3-8312-5242
{cswong,ian,rosalind}@mmu.edu.my

Fun Wey
Intel Penang Design Center,
Bayan Lepas Free Industrial Zone, Phase 3, Halaman Kampong Jawa, 11900 Penang, Malaysia
+604-2530092
funwey@intel.com

ABSTRACT
The Operating System scheduler is designed to allocate the CPU resources appropriately to all processes. The Linux Completely Fair Scheduler (CFS) design ensures fairness among tasks using the thread fair scheduling algorithm. This algorithm ensures allocation of resources based on the number of threads in the system and not within executing programs. This can lead to fairness issue in a multi-threaded environment as the Linux scheduler tends to favor programs with higher number of threads. We illustrate the issue of fairness through experimental evaluation thus exposing the weakness of the current allocation scheme where software developers could take advantage by spawning many additional threads in order to obtain more CPU resources. A novel algorithm is proposed as a solution towards achieving better fairness in the Linux scheduler. The algorithm is based on weight readjustment of the threads created in the same process to significantly reduce the unfair allocation of CPU resources in multi-threaded environments. The algorithm was implemented and evaluated. It demonstrated promising results towards solving the raised fairness issue. We conclude this paper highlighting the limitations of the proposed approach and the future work in the stated direction.

Categories and Subject Descriptors
D.4.1 [**Operating Systems**]: Process Management – *multiprocessing/multiprogramming/multitasking, scheduling, threads.*

General Terms
Algorithms, Measurement, Performance, Experimentation.

Keywords
Linux; Process scheduling; Completely Fair Scheduler; Fairness

1. INTRODUCTION
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 [1]. At any time, there can be more than one runnable threads/processes (collectively known as tasks) demanding service from the Central Processing Unit (CPU). In order to handle the oversubscription of tasks, an operating system scheduler is required to ensure each task obtains its share of CPU time as fair as the scheduling algorithm can guarantee.

Over the last few years, the CPU scheduler in the Linux kernel has been significantly revised so that performance improvement in terms of fairness, interactivity, and scalability can be achieved. Previous work of scheduler designs include the O(1) scheduler which was introduced by Ingo Molnar in the kernel 2.6 to replace the O(n) scheduler offers better support in Simultaneous Multi-processing (SMP) technology, fairness in CPU bandwidth distribution among tasks and better interactivity [2,3]. In the kernel 2.6.23, a new scheduler is introduced, called the Completely Fair Scheduler (CFS) [4].

The main design goal of CFS is to maximize overall CPU utilization while maximizing interactive performance. It also intends to fix deficiencies in the O(1) scheduler. The modular scheduler framework implements Scheduling Classes, which contains extensible scheduler modules. Normal time-sharing tasks are scheduled using the SCHED_OTHER scheduling policy, which is implemented in `sched_fair.c`. The real-time scheduling class is implemented in `sched_rt.c`, which is placed on the top of the hierarchy of scheduler modules. It still maintains the SCHED_FIFO (First In First Out) and SCHED_RR (Round Robin) semantics found in the O(1) scheduler. CFS uses a single time-based red-black tree for each CPU to track all the runnable tasks. Thus, components present in the O(1) scheduler like priority of run queues, interactive processes identification, and priority-based time slices concepts were removed. In the 17th revision of CFS, the scheduler includes scheduling entities (group, container, tasks, users, etc) patch [5], which are used to implement group-fair and user-fair scheduling.

In all of the three schedulers found in 2.4 to 2.6.23 kernels (O(n), O(1), and CFS), attention was mostly paid to designing schedulers that strives for fairness, interactivity, and scalability with Simultaneous Multi-processing (SMP) system. Fairness is the ability to distribute CPU bandwidth equally to all tasks based on their priorities. This is to prevent starvation, which occurs when tasks are left waiting for CPU service. In all the three versions of schedulers mentioned, fairness is implemented at the thread-level. Each thread with similar priorities in the system receives the same amount of CPU bandwidth. In CFS, the distribution of CPU bandwidth can be shared fairly among groups and users. This is to prevent any group or user to dominate the CPU resource at any time. It uses containers to cluster tasks spawned by the same user or group. Under a user's local CPU bandwidth, the distribution is still a thread-fair scheduler. At the time of writing this paper, group/user fair for real-time policy has also been implemented in the current development of CFS.

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. In multi-threaded environments, the starvation-free characteristic of the current scheduler will not be retained when "greedy" multi-threaded applications are run concurrently with "optimal" multi-threaded applications. Here, a multi-threaded application is defined as "optimal" if the number of threads it spawns is equal to the number of CPUs in the system. In contrast, "greedy" multi-threaded application refers to an application that spawns more than the "optimal" number of threads. Due to the task-fairness policy used in the scheduler, "greedy" threaded applications will tend to dominate the CPU resource and this will lead to fairness issues among other running applications. This fairness issue is crucial because software developers can take advantage of it by spawning additional threads so that their programs can dominate the CPU bandwidth when there are other programs executing concurrently. This fairness/starvation issue is particularly exposed in purely CPU-bound multi-threaded applications.

To solve the fairness issue, two main goals are outlined: (1) Limit greedy threaded applications from dominating the CPU resource. (2) The scheduler shall also retain the work-conserving characteristic of CFS. A work-conserving scheduler means the CPU resource cannot be idle when there is at least one runnable task in the system. To address this issue, we present the Process Fair Scheduler, which uses weight re-adjustment algorithm. With the implementation of an automatic weight distribution on threads created in the same process, it can lead to a solution to limit greedy threaded programs from dominating the CPU resource. However, Process Fair Scheduler is not optimized to take advantage of multi-threading because all programs will receive the same amount of time slice regardless of thread counts within the program. This limitation is overcome by Thread Fair Preferential Scheduler, which is the second approach that is set as the future work and direction of this paper.

Related work is discussed in [6], where the fairness among thread groups was suggested. Similarly, the process level fair-sharing algorithm has also been implemented in existing systems such as Aurema ARMTech's Workload Management Solutions for Citrix Metaframe Optimization [7] and Solaris 10 [8].

The contributions of the paper are as follows: (1) Study and analysis of the Linux CFS scheduler in the 2.6.24.2 kernel, which is the latest stable version of kernel at the time of writing. (2) An evaluation of the performance of the CFS scheduler for a multi-threaded application is conducted. The CPU resource allocation scheme of the Linux CFS scheduler is studied through the execution of an application that is multi-threaded to variable degrees. An observation is made about the time taken to complete the various applications and an optimal number of threads is identified. (3) We illustrate the problem statement that the Linux CFS scheduler does not exhibit an appropriate fairness in multi-threaded application execution, thus resulting in fairness or starvation problems. (4) Further, we present a solution, by introducing Process Fair Scheduler (PFS) and demonstrate its efficacy through an experimental evaluation.

The remaining part of this paper is presented as follows: In Section 2 the analysis and algorithm of CFS is presented. Section 3 presents the problem statement, which is the fairness issue that exists among multi-threaded applications. In Section 4, 3 experiments are carried out to illustrate the problem statement: (1) to find out the optimal number of threads required by a multi-threaded application so that maximum performance can be achieved with minimal overhead, (2) to exhibit how different numbers of threads in programs that run concurrently can affect performance of other programs thus leading to unfair CPU resource distribution, and (3) to show that software developers can take advantage of this weakness in the Linux scheduler to get more computational resource. In Section 5, we present a solution to address the problem. The limitation of the proposed solution and future work are also discussed. Finally, a conclusion is made in Section 6.

2. COMPLETELY FAIR SCHEDULER

The CFS scheduler is the successor of the O(1) scheduler. The specific goals that CFS is designed to accomplish are: (1) CFS shall provide good interactive performance while maximizing overall CPU utilization. The system should react and schedule interactive tasks immediately even when the system load is high. (2) It should also ensure fair distribution of CPU time to each entity. Schedule-able

entity is defined as any entity that can be scheduled on the CPU for execution. It can be a group of users, a container, a user or a task. By introducing schedule-able entities, group-fair and user-fair can be implemented. (3) The whole scheduler is implemented utilizing the modular scheduler framework by introducing Scheduling Classes.

2.1 Basic Algorithm of CFS

CFS attempts to accurately model an ideal multitasking CPU. An ideal multitasking uni-processor CPU divides the CPU time quantum equally to each tasks and executes them at equal speed. For example, if there are two tasks running with the same level of priority, they should be processed concurrently and both outputs are expected to be accomplished at the same time. Thus, fairness in CPU bandwidth distribution and good interactivity can be achieved. However, only a single task can be executed at one instant on a practical uni-processor platform. When the CPU is occupied by one task, other waiting tasks are at a disadvantage because that executing task gets an unfair amount of CPU time. To solve this problem, CFS splits up CPU time between runnable tasks as close to the 'ideal case' as possible. This is done by splitting up the time slice in a fine-grain way using high resolution nanosecond-accurate time slices.

2.2 Run-queue

The basic data structure in the scheduler is the run-queue. The run-queue is defined in `kernel/sched.c` as `struct rq` [9]. The run-queue is the list of runnable entities on a given processor and there is one run-queue per processor.

Each run-queue has a CFS-related field (`struct cfs_rq`) and real-time (`struct rt_rq`) related field. The CFS-related field is the data structure that houses runnable schedule-able entities under it. A hierarchy of runnable entities is formed by containers, groups, users or tasks. If group and user-fair scheduling are disabled, the entities equal tasks.

A task can be placed below another scheduling entity [10]. The task is put on the run-queue linked with its parent scheduling entity. When the scheduler decides on the next task to be processed, it first checks the top-level scheduling entities and chooses the one which is considered the most deserving of the CPU bandwidth. If that entity is not a task (i.e. a higher-level entity e.g. container), the scheduler dives into the run-queue contained within that entity (container) and repeats to check if the new selection is a task. The iteration continues down the hierarchy until the scheduling entity found is a task which is then selected for execution.

2.3 Entity Management through Red-Black Trees

All runnable tasks are sorted in a red-black tree by value of a "`se->vruntime - cfs->min_vruntime`" key which is calculated in function `entity_key`. A red-black tree is a self-balancing binary tree where the leftmost leaf has the smallest value because at each node, the smaller value is inserted into the left child, and the larger value is inserted into the right child [11]. There is a CFS-related field containing a red-black tree per CPU.

2.4 Task Fair Allocation of CPU Bandwidth

The CPU bandwidth is distributed proportionally to a task's priority. The ideal runtime for a task is given by Eq. 1:

$$slice = \frac{se \rightarrow load.weight}{cfs\_rq \rightarrow load.weight} \times period \qquad \text{Eq. 1}$$

where `se->load.weight` is the weight of each task mapped from its nice value in `prio_to_weight[ ]` and `cfs_rq->load.weight` is the total weight of all tasks under that CFS run-queue. `period` is the time slice the scheduler tries to execute all tasks. This means all tasks will be executed for the duration of their allocated time slice, which is a fraction of `period`. `period` is not a constant and has a default value of 20ms. This value will be increased when the number of runnable tasks in the runqueue is larger than `nr_latency` where `nr_latency` equals to the ratio of `sysctl_sched_latency` and `sysctl_sched_min_granularity`.

The task that is located at the leftmost node of the red-black tree is the one that is most entitled to run at any given time. `se->vruntime` is the monotonically increasing normalized time slice given to every schedule-able entity. It can be expressed in the Eq. 2:

$$vruntime += \frac{delta\_exec}{se \rightarrow load.weight} \times NICE\_0\_LOAD$$

$$= \frac{period}{cfs\_rq \rightarrow load.weight} \times NICE\_0\_LOAD \qquad \text{Eq. 2}$$

where `delta_exec` is the amount of execution time of that task. `NICE_0_LOAD` is the unity value of the weight, which is a static 10-bit fixed point number (2^{10} = 1024). For example, there are 3 tasks A, B, and C running with nice level 0, 1, and 2 respectively. The corresponding weights are 1024, 820 and 655. So, task A, B, and C have 8.2ms, 6.6ms, and 5.2ms of ideal runtime respectively. Assume this is an ideal case and all 3 tasks start together; the `delta_exec` shall be equal to `ideal_runtime`. Thus, the `vruntime` of task A, B, and C is 8.2ms / unit

weight. The `vruntime` for all tasks are expected to be the same after they have executed for a period of 20ms.

However, in practical cases, all tasks will not start executing at the same instant. So, there will always be one task leading another, and thus be placed on the leftmost leaf of the red-black tree.

3. FAIRNESS ISSUE AMONG MULTI-THREADED PROGRAMS

In previous sections, the Linux scheduler algorithm has been discussed: the CPU resource is distributed fairly among tasks. The major issue faced by task fair schedulers is the inability to allocate CPU resources fairly among programs in systems with multiple processors. The resource allocation is based on equally distributing the CPU resource amongst the number of threads executing in the system instead of amongst programs. This is illustrated in Figure 2.

Assume that 3 programs with the same nice value: A, B, and C are running on a Symmetrical Multi Processor (SMP) system with two CPUs. Program A is a single-threaded program and Program B and C have 2 threads each. When they are run concurrently, each thread will receive T_s amount of time slice, where T_s is equal to T period divided by total number of threads. T_s is inversely proportional to the number of tasks. In the scenario shown in Figure 2(a) (for simplicity, both CPU's run-queues are combined), Program B and C deserve to receive the major portion of CPU resources because they are optimally-threaded programs while Program A does not make use of multi-threading techniques. Optimal numbers of threads are the best number of threads created by a program in order to have the best performance in multi-processing environment, with the assumption that it is the only program running on the system.

In Figure 2(b), the program C receives an unfair amount of time slice due to its excessive number of threads. Here, the program which contains more than the optimal number of threads is termed as "greedy" threaded program. Although each thread receives the same amount of CPU time slice (all threads have the same nice value), Program C dominates most of the bandwidth because the ratio of the number of threads in Program C to Program B and A is 8:2:1. Although Program B is well-threaded, Program C receives a larger portion of CPU bandwidth. This issue needs to be addressed because this could be an advantage to software developers to spawn more additional unwanted threads than required so that their program can obtain a larger share of CPU resources when there are other programs with lower number of threads executing concurrently.

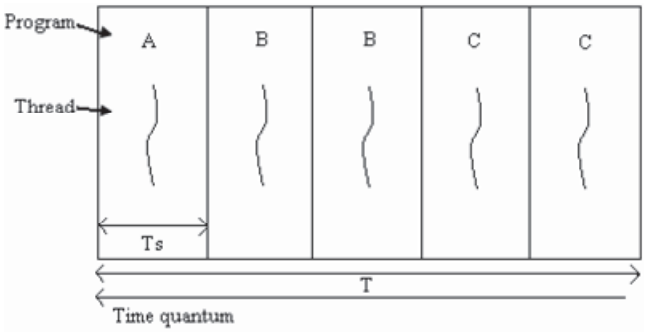


Figure 2(a). Program B and C are well-threaded programs

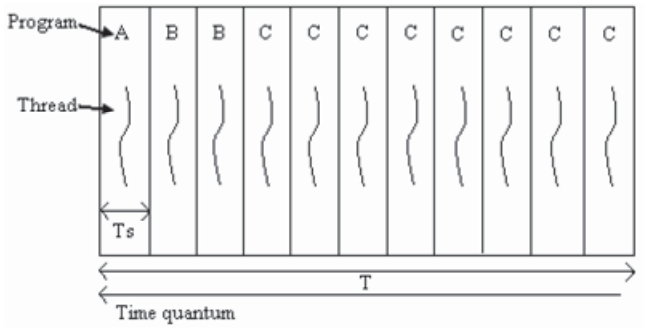


Figure 2(b). Program C is a greedy-threaded program

4. EVALUATION AND ANALYSIS

To illustrate the problem statement, three simulations were done: First, to find the optimal number of threads required by a multi-threaded application so that maximum performance can be achieved with minimal overhead. The CPU resource allocation scheme of the Linux CFS scheduler is studied through the execution of an application that is multi-threaded to variable degrees. An observation is made about the time taken to complete the application and an optimal number of threads is identified. Following the identification of the right degree of threading for the application on our particular hardware, the second simulation is used to validate the problem statement that the Linux CFS scheduler does not exhibit an appropriate fairness in application execution. This is shown by how different number of threads in programs that run concurrently can affect the performance of other programs and lead to unfairness of CPU resource distribution. The third simulation shows how software developers can take advantage of this weakness in the Linux scheduler to get more computational resource and hence be able to accomplish more output.

4.1 Simulation Environment

In order to demonstrate the problem of unfairness, we are utilizing a multi-threaded program executing in a multi processor environment. The program chosen was a pi (π)

calculation program which utilizes POSIX thread (pthreads) libraries that were available as part of the GNU C (gcc version 4.1.0) compiler distribution. Pthreads provides kernel-level threads through the system call `clone()`. The execution environment is comprised of an Intel Xeon X3210 Quad Core 2.13GHz (Kentsfield) CPU with 2.0 Gigabytes of main memory running the Linux 2.6.24.2 kernel.

The program to calculate π is a simple parallel computation intensive program that does not require communication between the threads and has only one shared variable to update. The π calculation program uses discrete integration [12] where the differential value is greater than the number of threads, and is easily parallelizable. The differential value is made very small to ensure that the π approximation program takes at least a few seconds to complete. This granularity is required as we use the time command to conduct the timing. In our program, we set the differential value to 1/2000000000.

Table 1. Sequence of Events for Timing Accuracy

Parent Process	Child Process	
fork( ) a child		
	Register wait signal	
	Wait for signal	
Send signal to start		
Record start time		
	Create threads	
	Parent thread	Created children threads
	Execute calculation	
Wait for child to end		
Record end time		
Calculate Time		

The π program uses the `clock_gettime()` interface to record the time used to finish the calculation. The main application (parent process) forks a process that manages the multi-threading part of the program according to the required number of thread set by user. This is to make sure the multi-threaded calculation part of the program has a different `tgid` (thread group ID) than the parent process, which only performs the timing mechanism. After the child is created, the child will register a signal handler to listen to the parent using `signal()`. After the parent process has ensured the child process is created, it then signals the child to start calculation using the `kill()` function. The parent registers the time when the child starts threading and performs the pi calculation.

The shared variable is updated using the pthread mutex locking mechanism to ensure correctness and the parent program waits for all the threads to complete before ending. At the completion time of the pi calculation, it will register the time it ends and the difference between the start and the end time is calculated. The sequence of events for the simulation is shown in Table 1. All three simulation tests are using the same multi-threaded pi program at nice value 0.

4.2 Optimal Number of Threads

An optimal number of threads is defined as the best number of threads to create in a program in order to have the best performance in a multi-processing environment. The first simulation's objective is to find out the optimal number of threads. The simulation was run on a test bed containing an Intel Quad Xeon CPU, which has four processing cores. It was performed under normal desktop conditions without other computation intensive programs running. The simulations were done using the pi program with a range of 1 to 8 threads. The average value of the time consumed by each run is taken from 100 simulation runs.

The graph shown in Figure 3 is drawn in a logarithmic scale using the recorded time consumed to execute the program for a variable number of threads spawned. The optimal number of threads is as expected, four threads because the processor used had four processing cores. It is also estimated to behave similarly for systems with other number of cores. There is a minor increase of runtime when the number of threads spawned is greater than the optimal number of threads. This is due to extra overheads introduced from additional context switching as a result of spawning extra number of threads.

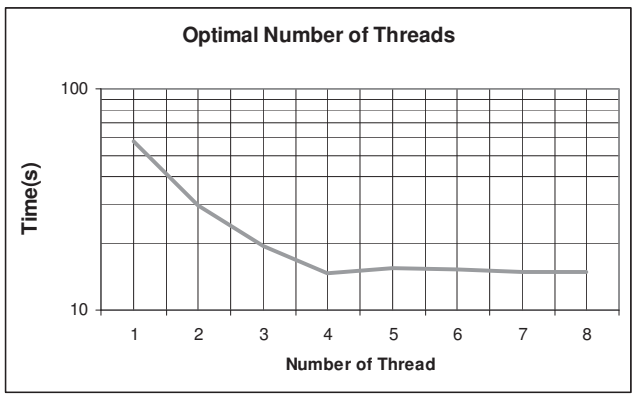


Figure 3. Time taken for the completion of the Pi approximation program in logarithmic scale using different number of threads

4.3 Fairness

In this simulation, we intend to show that two concurrently executing instances of the same program, one threaded

optimally (Program A), and the other threaded variably from 4 to 32 threads (Program B), will lead to unfair execution times. A total of 29 pairs of tests were run; i.e. 4 threads versus *n* threads, where *n* ranges from 4 to 32. This is illustrated in Figure 4.

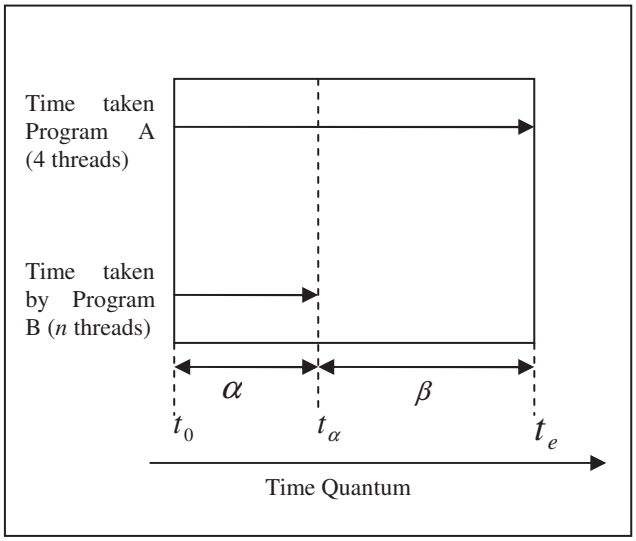


Figure 4. Time taken for the completion of multiple threads applications running concurrently

Both programs were initiated at the same time t_0, with Program B expected to finish executing sooner than Program A due to the higher ratio of the number of greedy threads to the optimal number of threads. At t_α, Program B finishes executing after a period of α. After a time period of α, Program A receives all the CPU bandwidth for the time period of β where it finishes executing at t_e. Thus, the main focus is on the time period α where Program A suffers starvation when Program B is dominating most of the CPU bandwidth. This is explained by Eq. 3:

$$\left(\frac{n}{4+n}\right)\alpha = T_B \Rightarrow \alpha = T_B\left(\frac{4}{n}+1\right) \qquad \text{Eq. 3}$$

where T_B is the time taken by Program B if it were solely executed without other programs running. The equation also exhibits that Program B receives a share of n/(4+n) from the α period of time. As the number of greedy threads *n* increases, the completion time α of the greedy threaded program is decreased. For Program A, it receives the remaining share in the α period of time and executing for the remaining time period of β until completion. This is shown in Eq. 4.

$$\left(\frac{4}{4+n}\right)\alpha + \beta = T_A \qquad \text{Eq. 4}$$

From the result of the test conducted in Section 4.2, $T_A \approx T_B = T$ for $n \geq$ optimal number of threads, with the assumption that context switching penalty on greedy threads are negligible. Thus, solving Eq. 3 and Eq.4 leads to Eq. 5:

$$\beta = \left(1 - \frac{4}{n}\right)T \qquad \text{Eq. 5}$$

where $n \geq$ optimal number of threads. This implies as the number of greedy threads *n* increases beyond the optimal number of threads, β also increases.

The following test was conducted under normal desktop conditions with no other computation intensive applications running. The two programs were initiated through a shell script where the programs were executed with their priority settings fixed to the same value (nice 0). The simulations were done for 2 pi programs executing concurrently with one of the programs fixed at the optimal number of threads (Program A) and the other had its thread count varied from 4 to 32 threads (Program B). Each simulation was done 100 times and the average timing was used.

The simulation results represented by the graph in Figure 5 shows that the timing of the program spawned into 4 threads (Program A) maintained a constant time. The graph for the programs threaded into a higher degree of threads (Program B) shows that the time taken to complete the program tends to fall to a certain extent and then maintains an approximately constant time for highly threaded programs (more number of threads). The result is as suggested in Eq. 3 and Eq. 5. The continuous decrement of α gives an insightful relation of the unfair distribution of CPU bandwidth to Program A which causes Program B to have higher throughput. This is further investigated in Section 4.4.

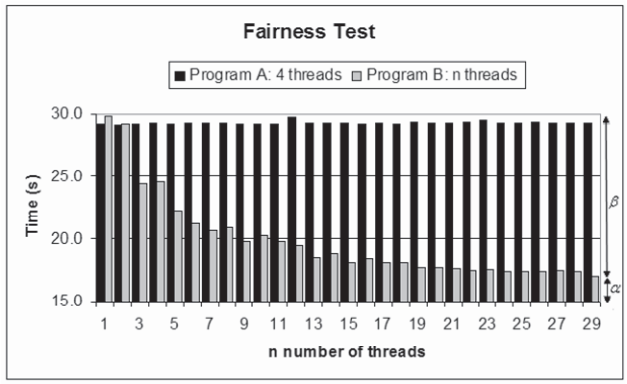


Figure 5. Time taken for the completion of multiple threads applications running concurrently

4.4 Programs Productivity

The third simulation is to demonstrate that software developers can take the advantage of the previously described characteristics of the Linux scheduler to get more computational resource and hence be able to accomplish more output.

Three tests were conducted, where each test used two pi programs that were executed concurrently. The first program is fixed to the optimal number of threads (4 threads) and produced only one output. The second program had 8, 16, or 32 threads producing 2, 4, and 8 outputs respectively. Multiple outputs can be produced by looping the program multiple times. For each simulation, the two programs were run concurrently 100 times and the average values were taken. This is described in Table 2.

Table 2. Each test is run concurrently

Test No.	Program 1		Program 2	
	No. of threads	No. of outputs	No. of threads	No. of outputs
1	4	1	8	2
2	4	1	16	4
3	4	1	32	8

Figure 6 shows that Program 1, which is an optimally threaded program {4} consumes more amount of time to complete its execution as the number of threads in Program 2 {8, 16, 32} is increased. It also shows that more threads in a program yields more outputs given the same time duration; i.e. for Test No. 1, Program 2 (8 threads each) yields 2 outputs compared to only one output in Program 1 (4 threads). In other words, higher threaded programs are relatively more productive when compared to optimally threaded programs. This constitutes an unfair scheme due to the loophole in the method used for resource allocation in the Linux scheduler. The reason of this unfair distribution scheme is due to the high ratio of the number of threads in greedy-threaded programs (in this case, Program 2) with respect to the optimal number of threads.

5. PROCESS FAIR SCHEDULER

The evaluation and analysis described in the previous sections demonstrate that the current Linux scheduler only distributes CPU resources fairly among threads in the system, and not among programs with different number of thread counts. This results in fairness/starvation problems for the programs with lower number of threads, especially the optimally threaded programs.

To solve the fairness issue, two main goals are outlined: (1) Limit greedy threaded applications from dominating the CPU resource. (2) Preserve the work-conserving characteristic of the Linux scheduler. A scheduler is said to be work-conserving if the CPU resource cannot be idle when there is at least one active task in the system.

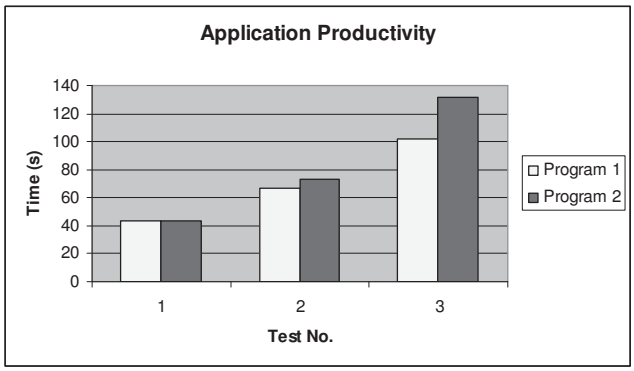
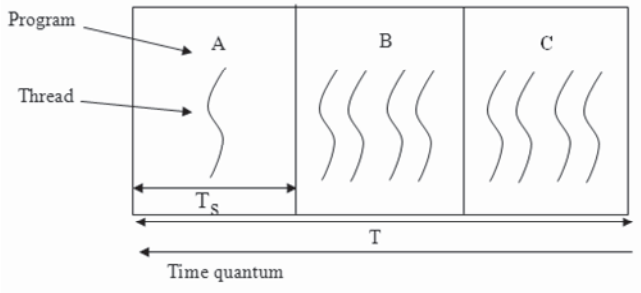


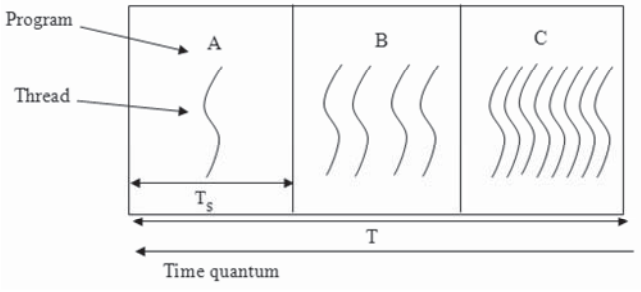
Figure 6. Two programs are run concurrently for each test.

To limit the CPU bandwidth dominance of greedy threaded programs, the Process Fair scheduler is designed. In Figure 7, the Process Fair scheduler prevents the greedy threaded program (Program C) from dominating the CPU resource. In Figure 7(b), the greedy threaded Program C receives T_S amount of time slice. In the Process Fair Scheduler (PFS), each thread within the same program receives the same amount of CPU bandwidth.

The main idea of PFS is to assign a new ideal time slice to threads according to the number of threads in the program. All the threads in the same group shall have the same time



(a) Program B and C are optimally threaded



(b) Program C is greedy-threaded

Figure 7. Process Fair Scheduler prevents greedy threaded program from dominating CPU resource.

slice. In CFS, the ideal time slice is directly proportional to the priority (or weight) of the task, as explained in Section 2.4. Thus, the key point is to downscale the time slice of all threads in the same thread group by dividing their weight by the total number of threads in the same thread group.

The new revised weight equation is given by Eq. 6:

$$se->load.weight = \frac{se->load.weight}{\alpha} \qquad \text{Eq. 6}$$

where α equals the number of threads created in the program.

The chronology of the PFS algorithm is:

1. The current Linux scheduler scheme does not differentiate between processes and threads. Instead, both are treated as tasks and are scheduled in the same way. Hence, an algorithm to identify processes and threads will be required so that they are scheduled accordingly.
2. After threads and processes are identified correctly, the total number of threads spawned under the same process is tracked.
3. New time slice shall be assigned to the identified group of threads so that greedy threaded program will not dominate the CPU resource. This can be done by weighting the ratio of the thread counts in optimal threaded programs and greedy threaded programs. A thread-grouping algorithm is needed to group threads which are descendents of the same process so that all the threads under same thread group have the same time slice.

5.1 Threads Count

The number of threads that exist under the same thread group is tracked for every thread created. Thus, when the `clone()` system call is called and the CLONE_THREAD flag is passed, indicating that a thread rather than a process is spawned, the number of threads count is incremented. All the threads created by the same process shall be grouped by synchronizing the thread count among them. The reason why synchronizing the number of threads is important is due to the nature that threads are created sequentially one by one. This can be done by increasing the `signal->count` in `copy_signal()` (called by `copy_process()`) using atomic locking mechanism.

5.2 Threads Grouping and New Time Slice

In order to downscale the time slice of the multi-threaded program, the weight of the newly created thread is divided by α. However, all the threads in the same thread group need to have the same weight due to synchronization. So, the readjusted weight value in the newly created thread needs to be synchronized with all the previous created threads of the same program (except the initial created thread). This is done by traversing through the sibling of tasks, which means children of the parent thread. However, not all traversed siblings belong to the same group of threads. To identify whether the newly created thread is in the same thread group as the traversed sibling, a condition is used to check the `tgid` values. If they have the same `tgid` value, it means the traversed sibling and the new thread are in the same thread group. Then, the `se->weight.load` of the sibling is assigned equal to the newly created thread's weight. `se->inv_load.weight`, which is the inverse of `se->weight.load`, needs to be updated accordingly as well.

Upon updating the weight of an entity, the entity needs to be dequeued from the run-queue first. This is to ensure that the entity will not be accessed concurrently by other run-queues during load balancing. The run-queue load will be reduced and the weight of the entity is updated. Then, it will be enqueued back into the run-queue and the load will be increased according to the updated weight's value. The siblings with the same `tgid` are traversed until all the thread group members have the same newly updated `load.weight` value.

The weight of threads spawned from the same process is guaranteed not to inherit by their descendant children process (as all processes have a parent except `init`). This is done to prevent the child process from suffering starvation when the process is a descendant of a greedy threaded program.

5.3 Fairness and Productivity Evaluation

To evaluate the fairness and the productivity achieved by PFS, we repeated the simulations conducted under Sections 4.3 and 4.4 with the PFS scheduler. Figure 8 shows that both programs A and B finish at approximately the same time regardless of the number of threads they are comprised of. The scheduler prevents greedy threaded programs from dominating the CPU bandwidth. This is due to the fact that

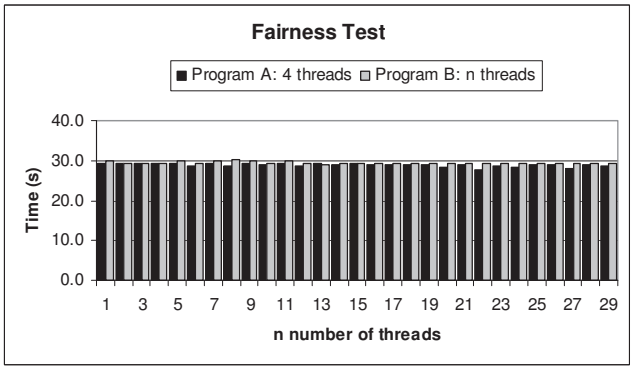


Figure 8. Process Fair Scheduler prevents greedy threaded program from dominating CPU resource.

the total time slice receives by Program A is equal to that by Program B at any given time.

Figure 9 shows that Program 1 maintains a steady productivity even with a concurrently executing greedy threaded Program 2. Program 1 consumes the same amount of execution time regardless of the number of threads existing in greedy threaded Program 2. Program 2 consumes more time to complete because this higher threaded {8, 16, 32} program has more number of outputs {2, 4, 8}.

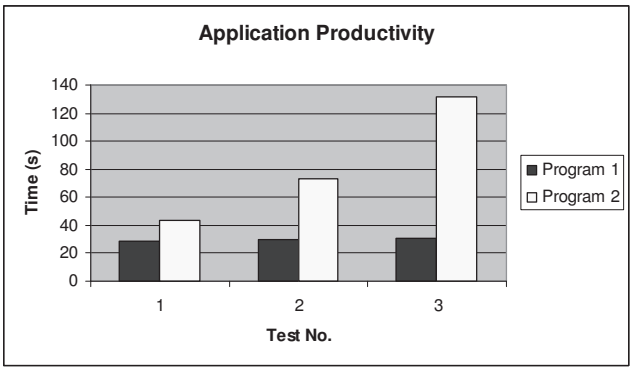


Figure 9. Process Fair Scheduler retains a steady productivity even with the existence of greedy threaded program.

5.4 Limitations and Future Work

One of the limitation of the proposed solution is the impact of threads in multi-threaded processes that exhibits both CPU-bound and I/O-bound behaviors; the latter one frequently leaving (sleep) or joining (wake up) the scheduling subsystem. Due to the start-time fair characteristic of the implementation, the time slice distribution may not be accurate when there are threads that block due to I/O events. To address this, the time slice of the thread that leaves or joins the run-queue shall be evenly distributed among other active threads (`task->state` equals to `TASK_RUNNING`). However, readjusting weights too frequently will introduce a significant overhead and latency to the scheduler. To reduce the overhead, the principle of the load balancing is used, where if a programmer spawns greedy threads, the threads will have similar characteristics (i.e. symmetrical threads). Hence, the system is unlikely to have both blocked threads and active threads at the same time unless they are all waiting for some variables. If the blocked and active threads are waiting for some variables, they are actually not classified as 'greedy' threads and hence weight readjustment is not necessary. Thus, the weight readjustment is only performed during load balancing. If a thread blocks due to an I/O event, each remaining active thread will inherit the gain of time slices through weight readjustment so that the whole thread group receives fair time slices with respect to other processes. The weight of each active thread in the same thread group shall be readjusted using Eq. 7.

$$se->load.weight = \frac{se->load.weight \times \alpha}{\alpha'} \quad \text{Eq. 7}$$

where α' is the updated number of active threads in the process. `se->inv_load.weight`, which is the inverse of `se->weight.load`, needs to be updated accordingly. It is to be noted that this issue is not significant on purely I/O-bound tasks because they tend to block at high frequency.

Another limitation of PFS is that all processes executing at the same nice value will receive the same amount of CPU time slice regardless of the number of threads spawned in the process. For example, a single-threaded program will receive the same amount of CPU bandwidth as an optimally threaded program when both are executed concurrently. This is an undesired characteristic because multi-threaded programs are not rewarded and programs with the lower number of threads than optimally threaded applications receive the same amount of CPU bandwidth. This is illustrated in Figure 7 where the single threaded Program A receives the same amount of CPU bandwidth as other multi-threaded programs. Apart from limiting greedy threaded applications' time slice, the scheduler shall also give preference to the optimally threaded applications in case the single-threaded programs are also running concurrently. One way to overcome this problem will be by implementing the Thread Fair Preferential scheduling (TFPS) algorithm. TFPS is similar to PFS except that TFPS shall give preference in allocating CPU bandwidth to the optimally threaded applications and at the same time prevent the greedy threaded program from dominating the CPU bandwidth.

The CPU bandwidth allocation scheme of TFPS is depicted in Figure 10, where the greedy threaded program (Program C) has the same amount of CPU bandwidth as the optimally threaded program (Program B). Both of their time slices are

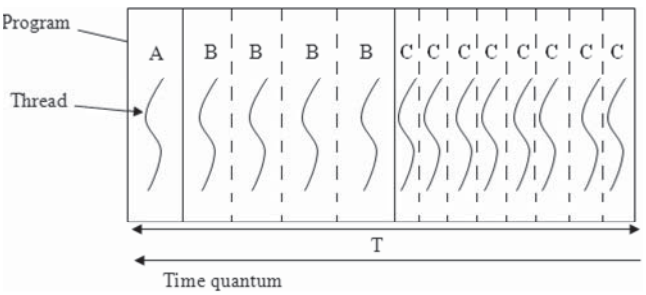


Figure 10. Thread Fair Preferential Scheduler prevents greedy threaded program from dominating CPU resource, but is also optimized for multi-threaded program.

larger than the single-threaded program (Program A). The ratio of the time slice for Program A to Program B to Program C is 1:4:4.

TFPS is an on-going project. It can be derived from the algorithm in PFS we are currently pursuing.

6. CONCLUSIONS

The paper first highlighted that the current Linux thread fair scheduler is not effective in distributing CPU resources in a fair manner when greedy threaded programs are executing. Evaluation and analysis of simulation results have verified and further illustrated the problem statement. We proposed solutions to overcome the fairness issue in order to achieve better fairness in the Linux scheduler. In general, there are two types of solutions.

The first is the Process Fair Scheduler (PFS), which has been implemented and tested in this paper. This scheduler can prevent greedy threaded programs from dominating the CPU bandwidth by allocating all processes with a proportional fair amount of CPU time slice. However, this feature also comes at a cost, where the scheduler also prevents optimally threaded programs from taking advantage of multi-threading technology. This disadvantage will be overcome by the second proposed approach, the Thread Fair Preferential Scheduler (TFPS). TFPS is an on-going project which is a modification of PFS, with the feature that optimally threaded programs will run faster than programs with lower than optimal number of threads and at the same time prevent greedy threaded programs from dominating the CPU bandwidth.

7. ACKNOWLEDGMENTS

We wish to acknowledge Linux kernel developers: Ingo Molnar, Peter Ziljstra, and Srivatsa Vaddagiri for consultation and assistance on CFS. Also, we would like to thank Intel Penang Design Centre for their research funding and hardware sponsor. We would like to thank the anonymous reviewers and our shepherd Hubertus Franke for their comments, feedbacks, and advices.

8. REFERENCES

[1] Silberschatz, A., Gailvin, P.B., Gagne, G., Operating System Concepts, 7th edition.

[2] Bovet, D. P. et al. 2005, Understanding the Linux Kernel, 3rd edition, O'Reilly Press, ISBN 0-596-00565-2.

[3] Love, R. 2005, Linux Kernel Development, 2nd edition, Noval Press, ISBN 0-672-32720-1.

[4] Molnar, I. 2007, Modular Scheduler Core and Completely Fair Scheduler [CFS], http://lwn.net/Articles/230501/

[5] Molnar, I. 2007, CFS Updates, http://kerneltrap.org/Linux/CFS_Updates/,

[6] Torvalds, L. 2007, Re: [Announce][patch] Modular Scheduler Core and Completely Fair Scheduler [CFS], http://kerneltrap.org/mailarchive/linux-kernel/2007/4/18/78624

[7] Aurema ARMTech, Workload Management Solutions for Citrix MetaFrame Optimization, http://www.responseunit.com/amherst/armtech/armtech_citrix_white_paper.pdf

[8] Solaris 10+ Resource Management, http://www.princeton.edu/~unix/Solaris/troubleshoot/resmgmt.html

[9] The Linux Kernel Archives Website: www.kernel.org.

[10] Corbet, J. 2007, Kernel Development, http://lwn.net/Articles/240080/

[11] Turbak, L. 2007, Red-Black Trees, http://cs.wellesley.edu/~cs231/spring01/ps4.pdf, Wellesley College.

[12] Turner, D., Calculating Pi in Parallel, Introduction to Parallel Computing and Cluster Computers, Ames Laboratory, http://cmp.ameslab.gov/cmp/para_comp_intro/mpi_intro/pi