# STUDIES ON AUGMENTED FAIRNESS SCHEDULING IN GENERAL PURPOSE ENVIRONMENTS

サミホ, ムハマド モサタファ サイド

https://doi.org/10.15017/1807063

出版情報：九州大学, 2016, 博士（学術）, 課程博士
バージョン：

# STUDIES ON AUGMENTED FAIRNESS SCHEDULING IN GENERAL PURPOSE ENVIRONMENTS

**S**amih **M**ohammed **M**ostafa **S**aid

**Kyushu University**

**2017**

# STUDIES ON AUGMENTED FAIRNESS SCHEDULING IN GENERAL PURPOSE ENVIRONMENTS

A Thesis Submitted to Graduate School of Information Science and Electrical Engineering in Partial Fulfilment of the Requirements for the Degree of Doctor of Philosophy in Computer Science

By

**S**amih **M**ohammed **M**ostafa **S**aid

**Kyushu University**

**2017**

# ACKNOWLEDGMENTS

A work of research – a thesis – is a product requiring the knowledge, thinking, encouragement and physical help of many contributors in different ways, over a period of time. Along with many other people, I am proud to express my appreciation to the following names:

Foremost, I would like to thank God for lighting my insight to completing this work.

“Allah does not thank the person who does not thank people” (religious principle).

I would like to express my special appreciation and thanks to my supervisor Assoc. Prof. Hirofumi Amano, you have been a tremendous mentor for me. Your advice on both research as well as on my career have been priceless. I also want to thank you for your brilliant comments and suggestions, thanks to you.

I am deeply indebted to Assoc. Prof. Shigeru Kusakabe, for his help. His advice and insight have been invaluable throughout my entire time of research. Our regular meetings gave me academic guidance and provided the milestones that helped to develop my research skill.

I expose my thanks to Prof. Keijiro Araki, Assoc. Prof. Kenji Hisazumi, Dr. Yoichi Omori, and all laboratory staff and students for their kind help, continuous encouragement, understanding and support. They gave me a lot of their time and effort. I cannot possibly thank them enough.

Furthermore, this dissertation would have never been possible without the support of my family. I would like to thank my mother, brothers and sisters for their encouragement during the very busy and difficult time of work finalization. In addition, I am deeply grateful for my father for his wise treatment. Thank you for your genes, Dad!! I am really grateful to my beloved wife for her deep caring and very strong encouragement that helped me in my “Mission Impossible” especially with kids at home.

# ABSTRACT

An operating system is a set of manual and automatic procedures that enable a group of people to share a computer installation efficiently. The key word in this definition is sharing: it means that people will compete for the use of physical resources such as processor time, storage space, and peripheral devices; but it also means that people can cooperate by exchanging programs and data on the same installation. The sharing of a computer installation is an economic necessity, and the purpose of an operating system is to make the sharing allowable.

It may be useful to distinguish between operating systems and user computations because the former can enforce certain rules of behavior on the latter. However, it is important to understand that each level of programming solves some aspects of resource allocation. Selecting a thread to be executed is called thread scheduling. The most important aspect of thread scheduling is the ability to multiprogramming. A single user cannot, in general, keep either the CPU or the Input/Output devices busy at all times. Multiprogramming increases CPU utilization by organizing processes such that the CPU always has one to execute.

The operating system keeps several processes in memory at a time. This set of processes is a subset of the processes kept in the process pool (since the number of processes that can be kept simultaneously in memory is usually much smaller than the number of processes that can be in the process pool). The operating system picks and begins to execute one of the processes in the memory. Eventually, the process may have to wait for some processes, such as an Input/Output operation to complete. In a non-multi programmed system, the CPU would sit idle. In a multiprogramming system, the operating system simply switches to and executes another process. When that process needs to wait, the CPU is switched to another process, and so on. Eventually, the first process finishes waiting and gets the CPU back. As long as there are always some processes to execute, the CPU will never be idle.

The timeslice is the numeric value that represents how long a process can run until it is preempted. The scheduler policy must dictate a default timeslice, which is not a trivial exercise. The scheduling problem can be stated shortly as: deciding on which process should be executed and moved to where, when and for how long according to

the timeslice, to optimize some system performance criteria. Linux scheduler, Completely Fair Scheduler, assigns processes a proportion of the processor.

Scheduling problems are known to be NP-Complete (i.e., it is believed that there is no optimal polynomial-time algorithm for them) except under a few special situations. The existing heuristics for scheduling processes works either statically or dynamically. This distinction is based on the time at which the scheduling decisions are made. In contrast to static techniques where the complete set of processes to be scheduled is known a priori and the scheduling is done off-line, in dynamic scheduling methods, the timeslice at each round is calculated at run time. Although the principal advantage of the static scheduling is its simplicity, it fails to adjust to changes in the system state. A dynamic scheme is needed because the arrival times of the processes may be random and some processes may go off-line and new processes may come on-line.

The general principle of the scheduler is to provide maximum fairness to each process in the system in terms of the computational power it is given. Or, put differently, it tries to ensure that no process is treated unfairly. Weights of threads and load balancing between processors/cores are the most influential factors on the fairness. The main goals of this work are divided into three parts: firstly; is improving the treatment of execution of running processes, this can be done by adjusting weights of running threads, it is called as local fairness, secondly; is achieving load balancing between cores, this can be done by migrating thread between cores, it is called as global fairness, and thirdly; is enabling the user to adjust Virtual Machine's (VM) weights, this can be done by adding system call, it is called as VM fairness. In addition to achieving these goals, it is important to achieve the scheduling criteria (i.e., minimizing waiting time and turnaround time) without causing defects in fairness and performance.

This research proposes, describes, compares, and discusses an algorithm that can be used for assigning processes to a machine. The processes are assumed to be independent (i.e., no communications between the processes are needed). This scenario is likely to be present, for instance, when many independent users submit their processes to a machine. Furthermore, the algorithm investigated in this study is

preemptive, and assume that the processes have no deadlines or priorities associated with them.

Experimental test studies were performed to compare the scheduling criteria and performance of the proposed scheduler versus default Linux scheduler. The experimental results provide a significant improvement in both waiting time, turnaround time and slightly higher performance.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF ABBREVIATIONS

CFS - Completely Fair Scheduler

CMP - Chip Multicore Processor

CPU - Central Processing Unit

FCFS - First-Come, First-Served

FIFO - First-In-First-Out

GPOS - General-Purpose Operating System

GPS - Generalized Processor Sharing

GUI - Graphic User Interface

I/O - Input/Output

IPC - Inter Process Communication

OS - Operating System

PC - Personal Computer

PCB - Process Control Block

PCPU - Physical CPU

PFS - Process Fair Scheduler

PID - Process Identification

PSS - Proportional Share Scheduling

RR - Round Robin

SFS - Surplus Fair Scheduling

SJF - Shortest-Job-First Scheduling

SRTF - Shortest Remaining-Time First

TFPS - Thread Fair Preferential Scheduler

TGID - Thread Group Identification

TLP - Thread-Level Parallelism

TWRS - Thread Weight Readjustment Scheduler

VCPU - Virtual CPU

VM - Virtual Machine

VMM - Virtual Machine Monitor

# CHAPTER 1:
# Introduction

This chapter summarizes the definition and importance of scheduling when designing OS, the general problem statement of the scheduling, and the most important scheduling criteria which will be stated throughout the thesis. Then, it discusses the underlying OS platform which will be used in the experiments throughout this work, some of its important features, its importance in the society, and the expected problems related to such OS in the future.

**Chapter organization:** Section 1.1 describes the motivation. In Section 1.2, the research objectives is discussed. Thesis overview is discussed in Section 1.3.

## 1.1 Motivation

Scheduling is considered as one of the most widely researched topics in the literature of Computer Science and Engineering. It is also a well-structured and conceptually-demanding problem. The need of efficient CPU scheduling techniques in the area of computer system performance cannot be overstated. Improper or inefficient scheduling may corrupt system performance. One of the main goals for OS design is a highly efficient process scheduler where CPU time is managed so that processes meet their criteria.

Fairness [14, 31, 41] measure is used in designing OS schedulers to determine whether users or applications are receiving a fair share of system resources. In scheduling processes, algorithms attempt to ensure fairness among processes, in addition to fulfillment of the desired criteria. Scheduling algorithms have been found to be NP-complete in general form [37, 49].

Modern OSs nowadays have become more complex than ever before. They have evolved from a single process, single user architecture to a multitasking environment in which processes run in a concurrent manner. Allocating CPU to a process requires careful attention to assure fairness and avoid process starvation for CPU. Scheduling decision try to minimize the scheduling criteria (i.e., average turnaround time, and average waiting time) [45]. Scheduling algorithms are the mechanism by which a

resource is allocated to a client. In this research, the concept of a resource is restricted to CPU usage and clients to processes. A scheduling decision refers to the concept of selecting the next process for execution.

One of the oldest, simplest and most widely used scheduling algorithms is RR [2, 10, 11, 45]. RR is also one of the oldest, simplest and most widely used PSS algorithms, and because of its usefulness, many PSS mechanisms have been developed [8, 18, 19, 21, 25, 33, 35, 43]. In the RR algorithm a small unit of time, called a time quantum or timeslice, is defined and fixed.

As machine architectures become more advanced, many commerce and scientific fields are demanding complex and computationally intensive applications. Usually these applications consist of various components that have different computational requirements. In fact, the applicability and strength of scheduling algorithms are derived from their ability to match computing needs to appropriate resources. Many algorithms profited from the benefits of the RR and tried to improve the performance of the system such as CPU utilization, waiting time, fairness, fair sharing [5, 6, 7], etc.

GPOS, as name implies can be used for various application such as banking, commerce, academic and scientific research, instrumentation, industrial automation and control. Some of the important features of GPOS are, IPC, Multitasking and Multithreading.

IPC: Within the development environment, users often work with many software simultaneously with data common to some or all software. The IPC capabilities of the OS enables to share data with each other. This feature thus facilitates the integration of data sharing and hence interoperability.

Multitasking: Multitasking is the ability to execute more than one application software at the same time. Advanced GPOS have the performance and capability of doing multitasking operations. A switching mechanism forms the basis of multitasking operation. Switching from one program to another is very quick. It gives the appearance of executing all of the programs at the same time although the OS executes them in some order. The OS takes responsibility for managing the CPU so that each application thinks it owns the entire system.

Multithreading: Numerous techniques exist to achieve performance improvements. Multithreading is the ability of an OS to execute different parts of a program, called threads, simultaneously. Multithreading architecture has emerged as one of the most promising technique for exploitation of parallelism. It exhibits a powerful form of concurrency. The technique can be exploited in a system that run on both uniprocessor and multiprocessor. The main difference between the concept of multitasking and multithreading is that in former case the notion of simultaneous operation is in between the application software, whereas in latter case the simultaneity is within the application software itself.

GPOS was developed as a desktop, servers, mainframes, and supercomputers OSs. It is a widely used OS both in research and business and will still be used in the future. No doubt, processes with enormous numbers of threads will be inevitable to execute in such kind of OS. In the same time, single-thread process will still run concurrently with multithreaded processes. Therefore, such simultaneous execution will cause: – 1) single-thread process suffers from starvation, where, multithreaded process will dominate CPU time by spawning more threads and captures CPU time which was supposed to be assigned to single-thread process, – 2) load balance between cores will be worse, where, with the growing number of running threads, some cores will be more loaded than others, and therefore, some processes in the less loaded cores will receive more CPU time than they should be assigned compared to other processes, and – 3) VMs will not be running flexibly, where it is supposed that user can get the services (CPU usage in this context) s/he desires, the VM provider should provide these services to the user, but, the VM provider assigns the same CPU usage for all VMs regardless the number of running threads. Scheduling in such kind of OS must be improved to confront these problems. Although, CPU schedulers fall into two broad categories: real-time and best-effort [4], real-time ones do not work for GPOS, therefore, the remainder of this work will focus on best-effort scheduler, this will be discussed in the next chapter.

## 1.2 Research Objectives

**Changing the behaviour of the current scheduler:** By: – 1) adjusting the weights of threads running in the same core for the purpose of improving the treatment of execution of running processes, this will help in preventing both of the

greedy behavior of multithreaded process and the starvation of single-thread process, – 2) modifying the thread migration for the purpose of improving the load balancing mechanism of the current scheduler, this will help in giving each process CPU time closer to the time assigned by the ideal scheduler, and – 3) adding new system call for the purpose of allowing the user to flexibly control the weight of each VM, this will help in providing services the user would like to obtain.

**Formulating equations:** Determining how to change the behaviour of the scheduler is based upon the equations that were formulated in this work. By using these equations, it will be easy to specify the new weights for running threads and decide which thread and when it should be moved.

**Achieving desired scheduling criteria:** Achieving better scheduling criteria, improving performance and preserving fairness between running processes must be taken into consideration when changing the behaviour of the current scheduler.

**Targeting various platforms:** Enhancing the scalability of the proposed work can be done by considering various types of platforms; single core, multicore, and VM environments.

**Comparing with current scheduler**: Comparing the proposed scheduler with another legacy scheduler from the literature (i.e., CFS) from the point of view of scheduler's behavior, scheduling criteria, and performance.

**Overcoming the previous disadvantages:** Overcoming disadvantages (i.e., greedy behaviour of multithreaded processes, and load imbalance between cores) of related approaches to enhance the preference of the proposed scheduler.

**Developing new features:** Developing some specifications of current scheduler (i.e., pricing framework in VM environment) can be done by allowing the user to get new services (i.e., CPU usage), and on the other hand, allowing the provider to determine the pricing calculations.

## 1.3 Thesis Overview

This thesis is organized in six chapters including this one. Their contents are described briefly in the following lines:

**Chapter 2-** Background Overview: In this chapter, a general description of what OSs do is offered. This chapter provides a more closely overview at processes. It discusses the process concept and goes on to describe various features of processes, including process scheduling algorithms, scheduling criteria, and load balancing.

**Chapter 3-** Local Fairness: In this chapter, TWRS is proposed, in which the weights of threads are adjusted in every round.

**Chapter 4-** Global Fairness: In this chapter, a modification of the CFS migration scheme is proposed to enable most threads to attain CPU time proportional to their weights and the weights of all running threads in all cores.

**Chapter 5-** VM Fairness: This chapter proposes a modification to the current scheduler to allow the user to flexibly control the weight of the running VM by setting it to any arbitrary value. This can be done by adding a system call.

**Chapter 6-** Conclusion and Future Works: This chapter discusses the concluding remarks and future research directions.

# CHAPTER 2:
# Background Overview

This chapter states the definition and importance of OS and summarizes the OS's role in the overall computer system, looks more closely at process and the concept of context-switches. The chapter discusses the thread; its benefits and models. Then it discusses the categories of CPU schedulers. It describes the definitions the scheduling criteria which have been suggested for comparing CPU scheduling algorithms, and illustrates several of scheduling algorithms; their pros and cons. The importance of load balancing in multicore system, and its two general approaches are also discussed in this chapter.

**Chapter organization:** Section 2.1 defines the OS. In Section 2.2, components of computer system and system goals are defined. Process concept is discussed in Section 2.3. Process scheduling is discussed in Section 2.4. Section 2.5 states categories of CPU schedulers. Definitions of CPU usage and capacity are discussed in Section 2.6. Section 2.7 discusses the scheduling criteria and algorithms. Load balancing is illustrated in Section 2.8.

## 2.1 Operating Systems

An OS is a system program that control the hardware and other software of the computer and when it is opened it brings the computer system into a mode from where it is easier to run other applications. OS provides an ‘abstraction layer’ between the application software and the low level hardware by freeing the user from any concern about the details of the underlying hardware of the computer. In summary, the OS manages the resources. The resources are application software, data, files, information and peripheral hardware.

An amazing aspect of OSs is how varied they are in accomplishing some tasks. Mainframe OSs are designed primarily to optimize utilization of hardware. PC OSs support complex games, business applications, and everything in between. OSs for handheld computers are designed to provide an environment in which a user can easily interface with the computer to execute programs. Thus, some OSs are designed to be convenient, others to be efficient, and others some combination of the two.

## 2.2 What Operating Systems Do

This section first summarizes the OS's role in the overall computer system. A computer system can be divided roughly into four components: the hardware, the OS, the application programs, and the users (Figure 2.1) [44].

The hardware, the CPU, the memory, and the I/O devices, provides the basic computing resources for the system.

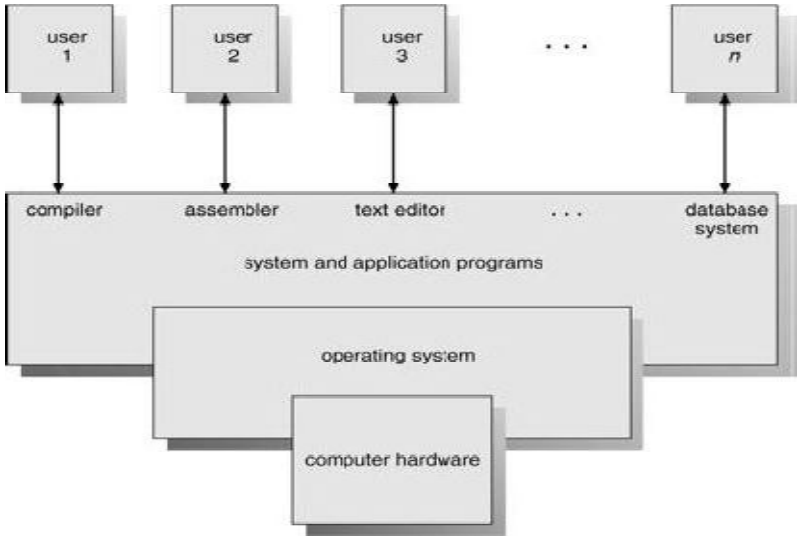



**Figure 2.1.** Abstract view of the components of a computer system.

The application programs, such as word processors, spreadsheets, compilers, and web browsers, define the ways in which these resources are used to solve users' computing problems. The OS controls and coordinates the use of the hardware among the various application programs for the various users.

A computer system can be viewed as consisting of hardware, software, and data. The OS provides the means for proper use of these resources in the operation of the computer system.

### 2.2.1 Defining Operating Systems

In general, there is no completely clear definition of an OS. OSs exist because they offer a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Toward this goal, computer hardware is constructed. Since bare hardware alone is not particularly easy to use, application

programs are developed. These programs require certain common operations, such as those controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the OS.

In addition, there is no universally accepted definition of what must be included in the OS. A simple viewpoint is that it includes everything a vendor ships when you order "the OS". The features included, however, vary greatly across systems. Some systems take up less than 1 megabyte of space and lack even a full-screen editor, whereas others require gigabytes of space and are entirely based on graphical window systems. A more common definition is that the OS is the one program running at all times on the computer (usually called the kernel), with all else being systems programs and application programs. This last definition is the one that this thesis generally follows. The matter of what constitutes an OS has become increasingly important.

### 2.2.2 System Goals

It is easier to define an OS by what it does than by what it is, but even this can be tricky. The primary goal of some OSs is convenience for the user. OSs exist because computing with them is supposedly easier than computing without them. As discussed in the previous subsection, this view is particularly clear when you look at OSs for small PCs. The primary goal of other OSs is efficient operation of the computer system. This is the case for large, shared, multiuser systems. These systems are expensive, so it is desirable to make them as efficient as possible. These two goals, convenience and efficiency, are sometimes contradictory. In the past, efficiency was often more important than convenience. Thus, much of OS theory concentrates on optimal use of computing resources.

OSs have also evolved over time in ways that have affected system goals. For example, UNIX started with a keyboard and printer as its interface, limiting its convenience for users. Over time, hardware changed, and UNIX was ported to new hardware with more user-friendly interfaces. Many GUIs were added, allowing UNIX to be more convenient to use while still concentrating on efficiency.

Designing any OS is a complex task. Designers face many tradeoffs, and many people are involved not only in implementing the OS but also in constantly revising

and updating it. How well any given OS meets its design goals is open to debate and involves subjective judgments on the part of different users.

OSs and computer architecture have influenced each other a great deal. To facilitate the use of the hardware, researchers developed OSs. Users of the OSs then proposed changes in hardware design to simplify them.

### 2.2.3 System Calls

System calls as the interface between the operating system and the process. System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.

The usage of system call can be explained by considering an example. Let us consider writing a program to read data from one file and to copy them to another file. The program will need the naming of two files: the input file and the output file. These names can be specified in many ways, depending on OS-design. The system asks the user to input the two files names, one for input and the other for output. This will now require a sequence of system calls. Firstly, to write a prompting message on the screen, and then to read the data from the keyboard. The characters which are input, defines the two files.

Once the two file names are provided, the program opens the input file and create the output file. Each of these operations requires the set of system calls to execute each step of the program. Sometimes, there may be error condition encountered for each operation. For example if a program tries to open a file, it may find that there is no filename exist or that file may be protected against access. In this case, the program should print the message on console (another set of system calls) and may be terminated abnormally (another set of system calls). If the input file exists, then a new output file is created.

Sometimes, an output file with the same name may be found. When this situation occurs, the old output file with the same filename is deleted (another system call).

Another way is that user prompts the message on the display unit whether to delete the existing file (another system call) or to abort the program (another system call).

Now both the files are present: the input file and the output file, so we can now read the data from input system (a system call) and write it to output file (another system call).

Each read-write operation provides the information regarding various possible error conditions. Finally after entire file is copied, the program closes both file (another system call) and writes a message to the console (another system call) and then terminates normally (last system call) [45].

## 2.3 Process Concept

Early computer systems allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, current-day computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required more control and compartmentalization of the various programs; and these needs resulted in the notion of a process, which is a program in execution. A process is the unit of work in a modern time-sharing system.

The more complex the OS is, the more it is expected to do on behalf of its users. Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are better left outside the kernel itself. A system therefore consists of a collection of processes: operating-system processes executing system code and user processes executing user code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) shared among them. By switching the CPU between processes, the OS can make the computer more productive.

This section looks more closely at processes. It first discusses the process concept and goes on to describe various features of processes, including behavior and scheduling.

A question that arises in discussing OSs involves what to call all the CPU activities. A batch system executes jobs, whereas a time-shared system has user

programs, or tasks. Even on a single-user system such as Microsoft Windows, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package. Even if the user can execute only one program at a time, the OS may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so all of them are called processes. The terms task and process are used almost interchangeably in this thesis.

### 2.3.1 Process Control Block

Each process is represented in the OS by a PCB, also called a task control block. A PCB is shown in Figure 2.2. It contains many pieces of information associated with a specific process, including:

- Process state: The state may be new, ready, running, waiting, halted, and so on.
- Program counter: The counter indicates the address of the next instruction to be executed for this process.
- CPU registers: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.


**Figure 2.2.** Process control block.

- CPU-scheduling information: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Section 2.4 describes process scheduling.)

- Memory-management information: This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the OS.
- Accounting information: This information includes the amount of CPU and real time used, time limits, account numbers or process numbers, and so on.
- I/O status information: This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for any information that may vary from process to process.

### 2.3.2 Threads

The process model discussed so far has implied that a process is a program that performs a single thread of execution. For example, when a process is running a word processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one function at one time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example. Many modern OSs have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one function at a time.

A thread is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID). Traditional (heavy weight) processes have a single thread of control. There is one program counter, and one sequence of instructions that can be carried out at any given time.

As shown in Figure 2.3, multithreaded processes have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.

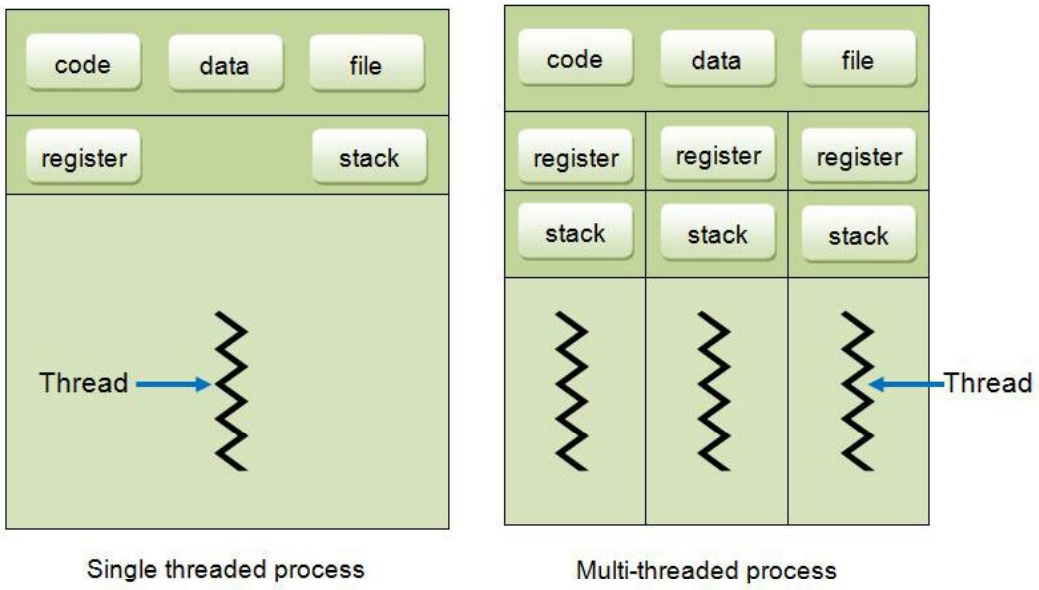



**Figure 2.3.** Single-thread and multithreaded processes.

## I. Benefits

There are four major categories of benefits to multithreading:

1. Responsiveness: One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
2. Resource sharing: By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
3. Economy: Creating and managing threads (and context-switches between them) is much faster than performing the same functions for processes.
4. Scalability, i.e. Utilization of multiprocessor architectures: A single-thread process can only run on one CPU, no matter how many may be available, whereas the execution of a multithreaded application may be split amongst available processors. (Note that single-thread processes can still benefit from multiprocessor architectures when there are multiple processes contending for the CPU, i.e. when the load average is above some certain threshold).

## II. Threads Types

There are two types of threads to be managed in a modern system: user threads and kernel threads. User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their

programs. Kernel threads are supported within the kernel of the OS itself. All modern OSs support kernel level threads, allowing the kernel to perform multiple simultaneous processes and/or to service multiple kernel system calls simultaneously. The user threads must be mapped to kernel threads.

## 2.4 Process Scheduling

The objective of multiprogramming is to have some processes running at all times, to maximize CPU utilization. The objective of time-sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a uniprocessor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

### 2.4.1 Schedulers

A process migrates among the various scheduling queues throughout its lifetime. The OS must select, for scheduling purposes, processes from these queues in some fashion. The selection is carried out by the appropriate scheduler.

Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for execution. The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them.

The primary distinction between these two schedulers lies in frequency of execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast. If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then 10/(100 + 10) = 9 percent of the CPU is being used (wasted) simply for scheduling the work.

The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the degree of multiprogramming (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

It is important that the long-term scheduler make a careful selection. In general, most processes can be described as either I/O-bound or CPU-bound. An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations. A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations. It is important that the long-term scheduler selects a good process mix of I/O-bound and CPU-bound processes. If all processes are I/O-bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. If all processes are CPU-bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.

On some systems, the long-term scheduler may be absent or minimal. For example, time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler.

Some OSs, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This medium-term scheduler is diagrammed in Figure 2.4. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up [45].

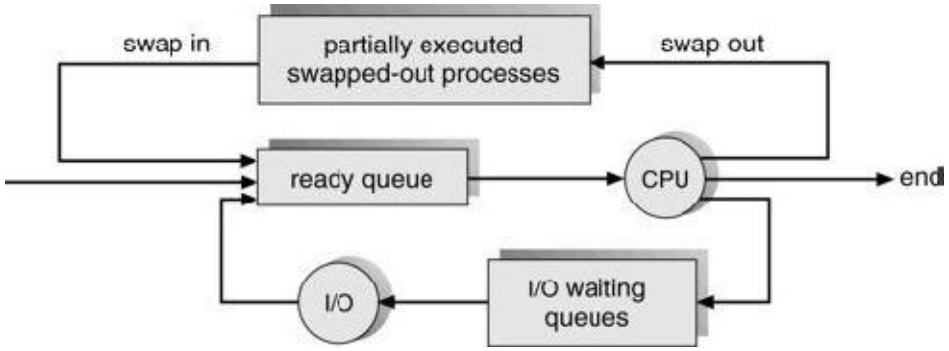



**Figure 2.4.** Addition of medium-term scheduling to the queueing diagram.

### 2.4.2 Context-Switch

Switching the CPU to another process requires saving the state of the old process and loading the saved state of the new process. This task is known as a context-switch. The context of a process is represented in the PCB of the process; it includes the value of the CPU registers, the process state, and memory management information. When a context-switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run (see Figure 2.5). Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be loaded, and the existence of special instructions (such as a single instruction to load or store all registers). Typical speeds are less than 10 milliseconds.

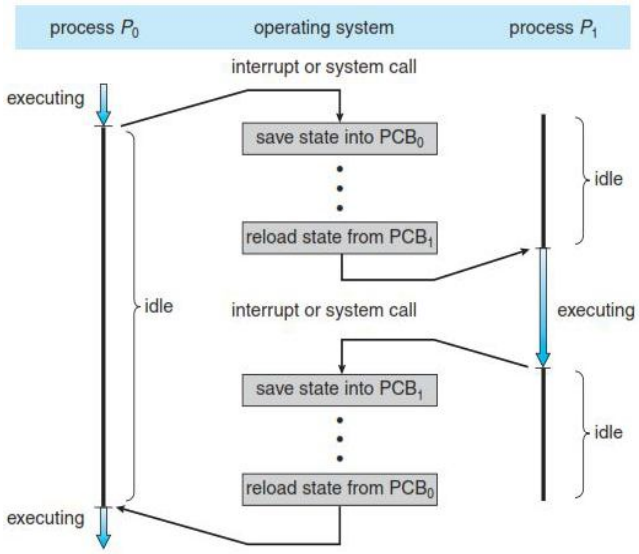



**Figure 2.5.** Diagram showing CPU switch from process to process.

Context-switch times are highly dependent on hardware support. For instance, some processors (such as the Sun ultraSPARC) provide multiple sets of registers. A context-switch simply requires changing the pointer to the current register set. Of course, if there are more active processes than available register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the OS is, the more work must be done during a context-switch. Advanced memory-management techniques may require extra data to be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory-management method of the OS. Context-switching has become such a performance bottleneck that programmers are using alternative structures (threads) to speed it up and possibly even avoid it whenever possible [45].

## 2.5 CPU Scheduling

CPU schedulers provide the illusion of multiple VCPUs to processes; each process appears to have its own CPU. The primary job of a CPU scheduler, then, is to safely and optimally divide CPU resources amongst competing processes. Safety is provided by the kernel's context-switching mechanism and the division of kernel code into portions that allow or disallow context-switching. Optimality is more difficult because the best way to divide CPU resources can vary between processes. Therefore, each CPU scheduler needs a system specific policy that defines how to share the processor. This policy encapsulates the broad scheduling goals of the system, and reflects the system's expectations regarding its workloads.

Scheduling policy is a balancing act between competing goals. Modern scheduling policies make tradeoffs between some primary goals: fairness, utilization, and performance. Other goals exist, but these three are often the most important. The general definition of fairness is, each running process should be assigned a **fair share** of processing time [47]. Utilization criterion dictates us to keep the CPU as busy as possible. Performance measures the amount of work accomplished by the system.

CPU schedulers fall into two broad categories: real-time and best-effort. Schedulers in the real-time category provide guarantees about how long it will take to

respond to an event; these schedulers ensure the process-defined deadlines are always met. Real-time schedulers are typically found in environments requiring deadline time guarantees, like robotics and embedded systems. To provide these guarantees, real-time schedulers need to know the CPU allocation and time requirements of all processes. If the scheduler cannot provide the deadline time guarantees an process requires, the process is not run.

Best-effort schedulers, in contrast, provide no guarantees; their primary goal is ease-of-use. Because they provide only best-effort service, they require no a priori knowledge of process deadlines or allocation requirements. These schedulers are found in all commodity OSs and used by both desktop and server-class machines [4, 45].

Best-effort schedulers are commonly divided into two groups: time-sharing, and proportional-share. The following sections discuss time-sharing and proportional-share in detail.

### 2.5.1 Time-Sharing Schedulers

The primary goal of time-sharing schedulers is to provide low latency for processes. This is accomplished by automatically dividing processes into classes according to the characteristics of their CPU bursts (the amount of time required by the process to finish). Time-sharing schedulers are often implemented using multiple run queues (Multilevel Feedback Queue). for example: Three queues: – 1) RR with time quantum of 4 milliseconds – 2) RR time quantum of 8 milliseconds – 3) FCFS (e.g., processes are dispatched according to their arrival times on the ready queue. Being a nonpreemptive discipline, once a process has a CPU, it runs to completion). If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. Processes are initially assigned a queue and quantum matching their CPU bursts. If the process consumes its quantum without yielding the CPU, it is moved to a worse priority and assigned a large quantum. In this way, each process is assigned a scheduling priority based on its CPU burst behavior and user-assigned priority. Processes can move between queues, this form prevents starvation. Each

queue has its own scheduling algorithm, queues require monitoring, which is a costly activity [44, 45].

### 2.5.2 Proportional-Share Schedulers

The fundamental goal of proportional-share schedulers is to provide fair allocations. Fairness in proportional-share schedulers is defined by the GPS model [35]. Intuitively, the GPS model attempts to provide the illusion that each process has its own CPU. These virtual, per-process CPUs run slower in direct proportion to the number of processes in the system. For example, in a system with three processes and a 3GHz processor, each process would make progress as though it had its own 1GHz processor. This model is, of course, impossible to achieve in the real-world where the processor can only be assigned to a single process at a time and very small scheduling quanta result in poor cache performance. Therefore, this model is interpreted as defining the relative CPU allocations given to processes. If all processes are equal, then all processes should receive the same CPU allocation over a given period of time. For example, one process can be given 70% of the CPU, another 20%, and a third 10%. These schedulers often interpret the GPS model to mean that in the long run all processes should receive their GPS share. Because this is the target environment, the remainder of this work will focus on proportional-share schedulers.

## 2.6 CPU Usage

CPU time (or process time) is the amount of time for which a CPU was used for processing instructions of a computer program or OS. The CPU time is measured in clock ticks or seconds. Often, it is useful to measure CPU time as a percentage of the CPU's capacity, which is called the CPU usage [29]. CPU usage is a term used to describe how much the processor is working at any given time. Capacity is defined as how many bits of information the CPU can process in one cycle. The more bits the CPU can process, the more processes it handles at one time. Rather than using strict rules that associate a relative priority value with the length of a time quantum, the CFS assigns a proportion of CPU processing time to each process. CPU time and CPU usage have two main uses:

- The first use is to quantify the overall busyness of the system.
- The second use, with the advent of multitasking, is to quantify how the processor is shared between computer programs.

## 2.7 Scheduling Criteria

Different CPU scheduling algorithms have different properties and may favor one class of processes over another. In choosing which algorithm to use in a particular situation, the properties of the various algorithms must be considered.

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:** This criterion dictates us to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput:** One measure of work is the number of processes that are completed per time unit, called throughput.
- **Turnaround time:** From the point of view of a particular process, the important criterion is how long it takes to complete that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time:** The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time:** Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, the average values of these measures among processes are optimized. However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize

the maximum response time. Investigators have suggested that, for interactive systems (such as time-sharing systems), it is more important to minimize the variance in the response time than it is to minimize the average response time. A system with reasonable and predictable response time may be considered more desirable than a system that is faster on the average but is highly variable. However, little work has been done on CPU-scheduling algorithms that minimize variance [45].

### 2.7.1 When to Schedule

A key issue related to scheduling is when to make scheduling decisions. It turns out that there are a variety of situations in which scheduling is needed.

First, when a new process is created, a decision needs to be made whether to run the parent process or the child process. Since both processes are in ready state, it is a normal scheduling decision and it can go either way, that is, the scheduler can legitimately choose to run either the parent or the child next.

Second, a scheduling decision must be made when a process is submitted. That process can no longer run (since it no longer exists), so some other processes must be chosen from the set of ready processes. If no process is ready, a system-supplied idle process is normally run.

Third, when a process blocks on I/O, on a semaphore, or for some other reason, another process must be selected to run. Sometimes the reason for blocking may play a role in the choice. For example, if A is an important process and it is waiting for B to exit its critical region, letting B run next will allow it to exit its critical region and thus let A continue.

Fourth, when an I/O interrupt occurs, a scheduling decision may be made. If the interrupt came from an I/O device that has now completed its work, some process that was blocked waiting for the I/O may now be ready to run. It is up to the scheduler to decide if the newly ready process should be run, if the process that was running at the time of the interrupt should continue running, or if some third process should run.

### 2.7.2 Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms. This subsection describes their categories.

Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts. A nonpreemptive scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or until it voluntarily releases the CPU. Even if it runs for hours, it will not be forcibly suspended. In effect, no scheduling decisions are made during clock interrupts. After clock interrupt processing has been completed, the process that was running before the interrupt is always resumed.

In contrast, a preemptive scheduling algorithm picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available). Doing preemptive scheduling requires having a clock interrupt occur at the end of the time interval to give control of the CPU back to the scheduler. If no clock is available, nonpreemptive scheduling is the only option. There are many different CPU-scheduling algorithms. This section describes several of them.

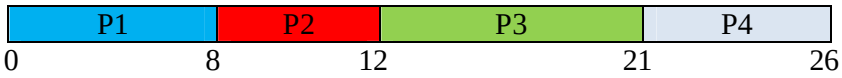
**I. First-Come, First-Served Scheduling**

FCFS [45] is the simplest CPU-scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

**Table 2.1.** Set of processes running under FCFS sheduler.

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 8 |
| P2 | 0 | 4 |
| P3 | 0 | 9 |
| P4 | 0 | 5 |

If the processes arrive in the order P1, P2, P3, P4 and are served in FCFS order, we get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:




The waiting time is 0 milliseconds for process P1, 8 milliseconds for process P2, 12 milliseconds for process P3, and 21 milliseconds for process P4. Thus, the average waiting time is (0 + 8 + 12 + 21) / 4 = 10.5 milliseconds. The average waiting time depend on the arrival order of the processing.

**Pros and Cons:** The implementation of the FCFS policy is easily managed with a FIFO queue. The code for FCFS scheduling is simple to write and understand. However, it is not suitable for time sharing systems where it is important that each user should get the CPU for an equal amount of time interval and poor in performance, as average wait time is high if short requests wait behind the long ones.

### II. Shortest-Job-First Scheduling

SJF [45] associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

As an example of SJF scheduling, consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

**Table 2.2.** Set of processes running under SJF scheduler.

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 8 |
| P2 | 0 | 4 |
| P3 | 0 | 9 |
| P4 | 0 | 5 |

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:


The waiting time is 9 milliseconds for process P1, 0 milliseconds for process P2, 17 milliseconds for process P3, and 4 milliseconds for process P4. Thus, the average waiting time is (9 + 0 + 17 + 4) / 4 = 7.5 milliseconds.

**Pros and Cons:** The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases. The real difficulty with the SJF algorithm is knowing the length of the next CPU request, and can lead to unfairness or starvation (processes with large service times tend to be left in the ready list while small processes receive service).

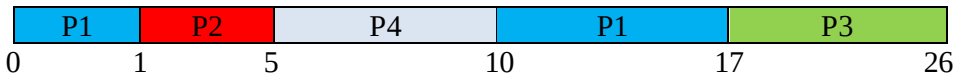
### III. Shortest Remaining-Time First Scheduling

Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling [45]. As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

**Table 2.3.** Set of processes running under SRTF scheduler.

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 8 |
| P2 | 1 | 4 |
| P3 | 2 | 9 |
| P4 | 3 | 5 |

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting SRTF schedule is as depicted in the following Gantt chart:




Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process

P2 is scheduled. The average waiting time for this example is [(10 − 1) + (1 − 1) + (17 − 2) + (5 − 3)]/4 = 26/4 = 6.5 milliseconds.

**Pros and Cons:** short processes returns quickly and it has high yield (processes done per minutes). The real difficulty with the SRTF algorithm is knowing the length a process's remaining time, and the length of the next CPU request.

### IV. Round Robin Scheduling

The RR scheduling algorithm [45] is designed especially for time sharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the OS. A context-switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

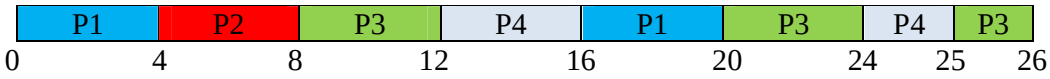
Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

**Table 2.4.** Set of processes running under RR scheduler.

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 8 |
| P2 | 0 | 4 |
| P3 | 0 | 9 |
| P4 | 0 | 5 |

If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds. Since it requires another 4 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2.

Process P2 needs 4 milliseconds, so it quits after its time quantum expires. The CPU is then given to the next processes, process P3 and then process P4. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. Then, the CPU is returned to processes P3, P4 and P3 for additional time quantum. The resulting RR schedule is as follows:




P1 waits for 12 milliseconds (16 - 4), P2 waits for 4 milliseconds, P3 waits for 17 milliseconds (25 – 8), and P4 waits for 20 milliseconds (24 – 4). Thus, the average waiting time is (12 + 4 + 17 + 20) / 4 = 13.25 milliseconds.

**Pros and Cons:** RR is easy to implement, and guarantees fairness since every process gets equal share of CPU. However, important processes may wait in line, average waiting time can be bad. The performance of the RR algorithm depends heavily on the size of the time quantum. if the time quantum is extremely large, the RR policy is the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context-switches.

### V. Lottery Scheduling

With lottery scheduling [45], the goal is to allow a process to be granted a proportional share of the CPU (i.e., a specific percentage). Conceptually, lottery scheduling works by allocating a specific number of “tickets” to each process. The more tickets a process has, the higher its chance of being scheduled.

Consider the following set of processes that arrive at time 0, each with the number of tickets:

**Table 2.5.** Set of processes running under Lottery Scheduler.

| Process | Arrival Time | Tickets |
|---|---|---|
| P1 | 0 | 8 |
| P2 | 0 | 4 |
| P3 | 0 | 9 |
| P4 | 0 | 5 |

The fractions of processor time given to each should be: P1: 8/26 = 30.76%, P2: 4/26 = 15.38% , P3: 9/26 = 34.61%, and P4: 5/26 = 19.23%.

**Pros and Cons:** each process is given a proportional share of the CPU. The difficulty is determining ticket distribution, particularly in an environment where processes come and go and get blocked. This is not a useful algorithm for general-purpose scheduling.

### 2.7.3 Scheduling Algorithm Goals

In order to design a scheduling algorithm, it is necessary to have some idea of what a good algorithm should do. Some goals depend on the environment (batch, interactive, or real time), but there are also some that are desirable in all cases. Some goals are listed as follows:

**I. All systems:**

- Fairness - give each process a fair share of the CPU.
- Policy enforcement - see that stated policy is carried out.
- Balance - keep all parts of the system busy.

**II. Batch systems:**

- Throughput - maximize jobs per hour.
- Turnaround time - minimize time between submission and termination.
- CPU utilization - keep the CPU busy all the time.

**III. Interactive systems**

- Response time - respond to requests quickly.
- Proportionality- meet users' expectations.

**IV. Real-time systems**

- Meeting deadlines - avoid losing data.

  Predictability - avoid quality degradation in multimedia systems.

## 2.8 Load Balancing

On multicoreprocessor/multiprocessor systems, it is important to keep the workload balanced among all cores to fully utilize the benefits of having more than one core. Otherwise, one or more cores may sit idle while other cores have high workloads, along with lists of processes awaiting the CPU. Load balancing attempts to keep the workload evenly distributed across all cores. It is important to note that

load balancing is typically necessary only on systems where each core has its own private queue of eligible processes to execute. On systems with a common run queue, load balancing is often unnecessary, because once a core becomes idle, it immediately extracts a runnable process from the common run queue.

There are two general approaches to load balancing: push migration and pull migration. With push migration, the kernel periodically checks the load on each core and, if it finds an imbalance, evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy cores. Pull migration occurs when an idle core pulls a waiting process from a busy core. Push and pull migration need not be mutually exclusive and are in fact often implemented in parallel on load-balancing systems.

### 2.8.1 Load Balancing Categories

The redistribution of processes among the cores during execution time is called load balancing. The load balancing is performed by transferring processes from the heavily loaded cores to the lightly loaded cores. The load balancing technique is designed specifically for parallel processes running on multicore environment in order to achieve good performance. Load balancing methods are classified into two major groups: dynamic load balancing (the redistribution of processes among the cores during execution time) and static load balancing (assignment of processes to cores is done before program execution begins). This thesis focuses on dynamic load balancing because current GPOS (i.e., Linux) uses this mechanism such that the system need not be aware of the run-time behavior of the applications before execution, and some processes may go off-line and new processes may come on-line. Load balancing can be decomposed into three inherent phases:

- **condition phase**, which determines the conditions under which a process should be transferred;
- **decision phase**, which specifies the amount of load information made available to process positioning decision-makers; and
- **positioning phase**, which identifies the core to which a process should be transferred.

From a system's point of view, the processes distribution choice is considered as a resource management issue and should be considered as an important factor during the design phases of multicore systems [42].

# CHAPTER 3:
# Local Fairness

This chapter summarizes the shortcoming of current Linux scheduler. Current Linux scheduler does not control the greedy behavior of multithreaded process, and therefore, single-thread process suffers from starvation. In this chapter, this shortcoming is illustrated through experimental evaluation thus exposing its weakness. The chapter discusses the advantages and disadvantages of related researches. Then, it proposes a modification of current Linux scheduler, describing its features and mechanism. The chapter compares the proposed scheduler versus current Linux scheduler from the point of view of greedy evaluation, scheduling criteria, and performance evaluation.

**Chapter organization:** Section 3.1 describes the motivation. In Section 3.2, the problem statement is defined. Related research is discussed in Section 3.3. In Section 3.4, the proposed scheduler is presented. Experimental setup is discussed in Section 3.5. Evaluation and analysis are discussed in Section 3.6. Discussion is stated in Section 3.7.

## 3.1 Motivation

The need for a scheduling algorithm arises from the requirement for most modern systems to perform multitasking. Scheduling is a key concept in computer multitasking and multiprocessing OS designs. Due to the growing availability of multicoreprocessor, processes are encouraged to be designed using multiple threads so that the benefit of TLP can be exploited [36, 51]. The OS scheduler is designed to allocate system resources, CPU time, proportionally to all processes.

Scheduling algorithms have been found to be NP-complete in general form. The trade-off 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 harder to come [1].

The lack of accurate fairness can cause poor support for processes, also the shortage of precise fairness leads to inadequate support for the system.

Assigning appropriate timeslice enables the scheduler to make scheduling decisions for the system and prevents any process from monopolizing the processor. On many modern OSs, the timeslice is dynamically calculated as a function of process behavior and configurable system policy. This chapter shows that kernel performance can be improved significantly by modifying just few key parameters. It investigates the effect of changing weights of sibling threads (threads created in the same process) and evaluates this modification in terms of treating the execution of running processes, scheduling criteria and performance.

### 3.1.1 Relationship between sharing and scheduling criteria

From the user point of view, it is important to maximize CPU utilization. CPU utilization depends on scheduling criteria (i.e., turnaround time and waiting time). Sharing contributes in maximizing CPU utilization. The relation between scheduling criteria and sharing can be described as follows.

- Definition 1. Turnaround Time: total time from submission of a process to its completion.

$$Turnaround\ Time = Completion\ Time - Arrival\ Time$$

- Definition 2. Waiting Time: total time a process has spent in the ready queue.

$$Waiting\ Time = Turnaround\ Time - Execution\ Time$$

- The share in a time interval $[t_1, t_2]$ of a running process $i$ is a ratio of its weight to the sum of weights of all active processes in the run queue. This is computed as:

$$share_i(t_1, t_2) = \frac{w_i}{\sum_{\forall j \in R} w_j}(t_2 - t_1)$$

where $R$ is the set of all running processes that are currently in run queue of a core. $w_i$ is the weight of the process, it is mapped from its nice value in `prio_to_weight`[] [30], (defined as a variable in `kernel/sched.c` file), and each nice value has its respective weight.

- The timeslice that a process $i$ should receive in a period of time is given by:

$$slice_i = share_i \times period$$

therefore,

$$share \propto \frac{1}{\sum_{j \in R} w_j}$$

and,

$$slice \propto share$$

then,

$$slice \propto \frac{1}{\sum_{j \in R} w_j}$$

It is known that,

$$waiting\ time \propto \frac{1}{slice}$$

therefore,

$$waiting\ time \propto \frac{1}{share}$$

also,

$$waiting\ time \propto turnaround\ time$$

therefore,

$$turnaround\ time \propto \frac{1}{share}$$

## 3.2 Problem Statement

This section discusses the problem statement illustrating it with examples.

### 3.2.1 Overview

Each process and thread is a task in the eyes of the Linux scheduler. Forked process is assigned a PID and TGID. 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, 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, and inherits its TGID from its parent as shown in Figure 3.1.

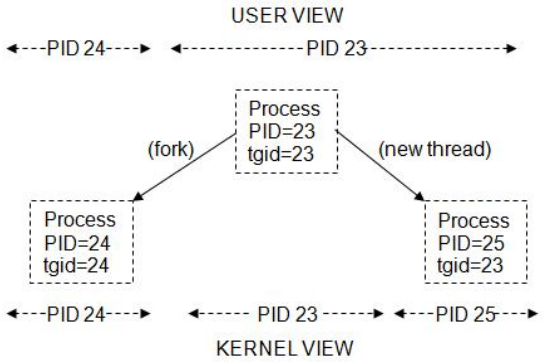



**Figure 3.1.** Identifications of process and thread created from parent process.

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 time that thread will receive. Greedy users could take advantage by spawning more additional threads in order to obtain larger CPU resources.

The situation can be summarized as: the default Linux scheduler is process-agnostic[1] and allows for greedy behavior, where processes consisting of more threads may receive more aggregate CPU time from the scheduler relative to processes with fewer threads. That is, the scheduler does not take process membership into account, or inter-process fairness.

From another view, this has a negative effect on running processes with fewer number of threads, for example, single-thread process will suffer from starvation.

### 3.2.2 Greedy Behavior

To elaborate the problem statement, the following test was done to show the greedy behavior of multithreaded processes. The problem statement is shown by testing how processes with more threads affect other processes with fewer threads and lead to dominating CPU resources. This test, in Figure 3.2, shows two cases: the first case where all running processes possess the same number of threads, and the second case where the running processes have different number of threads, this is done by running three processes (--test=threads) concurrently in SysBench benchmark [48] using shell script. Three processes were executed concurrently with six groups of different numbers of threads. In group 1 (first case), all processes have the same

[1] The word *agnostic* comes from the Greek *a-*, meaning *without* and *gnōsis,* meaning *knowledge.*

number of threads, 8 threads for each. In the other groups (second case), processes A and B with a fixed number of threads, one thread (single-thread process) and eight threads respectively, and the third process, C, has different number of threads, (10, 20, 30, 40, 50). Figure 3.2 shows that processes with the same number of threads in group 1 receive approximately same amount of CPU usage, and in the other groups, as the increasing the threads number of a process is, the CPU usage is increasing. These tests were carried out on Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz.

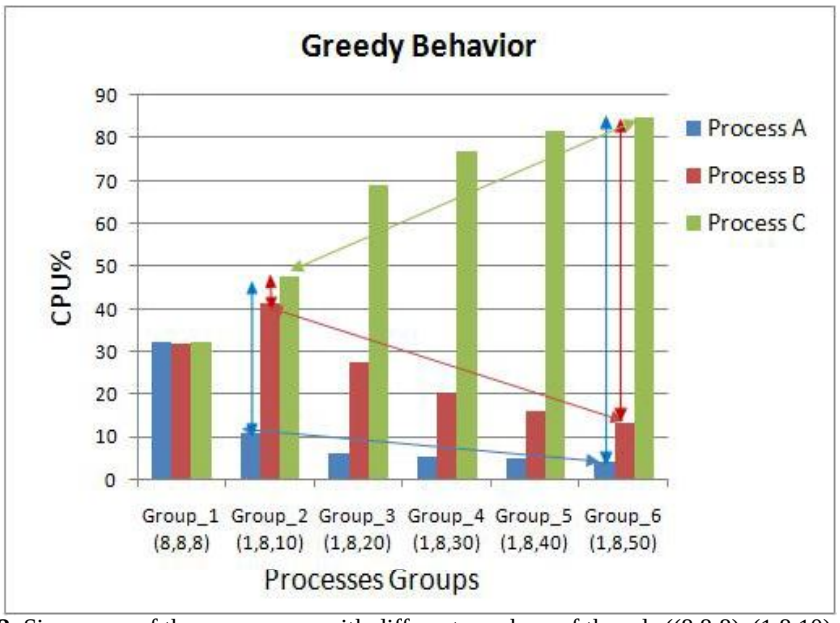



**Figure 3.2.** Six groups of three processes with different numbers of threads ((8,8,8), (1,8,10), (1,8,20), (1,8,30), (1,8,40), (1,8,50)) run concurrently.

Figure 3.2 plots the CPU usage consumed by three processes running on CFS with varying number of threads in process C. This figure reflects an evident relation of the greedy behavior of multithreaded process. This greedy behavior comes from:

- The continuous increase of the gaps of CPU usage between processes in each group (e.g. between process C and process A, and between process C and process B), represented by the vertical arrows in Figure 3.2.
- The continuous increase (of process with more threads, C) and decrease (of processes with fewer threads, A and B) ratios of the CPU usages through all groups, represented by the sloped arrows.

Based on what has been mentioned, the main problem the current scheduler faces is that the process with more threads acquires its CPU usage by deducting from the

CPU usage of processes with fewer threads, therefore, as the number of threads of greedy process increases, the deduction increases. This work contributes in reducing this deduction.

## 3.3 Related Research

PSS has long been studied in OSs. The important property of PSS techniques is that they characterize threads with a single parameter, a share, and consequently, PSSs are often primarily evaluated based on the level of fairness that they can provide [39]. Other evaluation criteria depend on for what the scheduler is designed.

### 3.3.1 Surplus Fair Scheduling

Chandra et al. [12] presented proportional share CPU scheduler, SFS, designed for symmetric multiprocessors. The authors first showed that the infeasibility of certain weight assignments in multiprocessor environments results in starvation or unfairness in many existing PSSs. 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 PSSs to significantly reduce, but not eliminate, the unfairness in their allocations.

The authors introduced the following definition: In any given interval $[t_1, t_2]$, the weight $w_i$ of thread $i$ is infeasible if

$$\frac{w_i}{\sum_{j \in \alpha} w_j} > \frac{1}{P}$$

where $\propto$ is the set of runnable threads that remain unchanged in $[t_1, t_2]$ and $P$ is the number of CPUs.

An infeasible weight represents a resource demand that exceeds the system capability. The authors showed that, in a P-CPU system, no more than $P - 1$ threads can have infeasible weights. They proposed converting infeasible weights into their closest feasible ones.

### 3.3.2 Process Fair Scheduler

Wong et al. [51] 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 multithreaded environment. The authors

assumed that the optimal number of threads, best number a process should be given in order to achieve the best performance in a muti-processing environment, equals to the number of available cores. PFS changes the weight of thread according to the equation:

$$weight(thread) = \frac{weight(process)}{\alpha}$$

where α is the number of threads created in the process.

In PFS, all processes will be assigned the same amount of timeslices regardless the number of threads, therefore, multithreaded processes will not be rewarded which is considered as a defect in this algorithm.

### 3.3.3 Thread Fair Preferential Scheduler

TFPS [52] is a modification of PFS algorithm to overcome the shortcoming of PFS. TFPS shall give the greedy threaded process the same amount of CPU time as optimally threaded process, and both of their timeslices are larger than the timeslice of single-thread process. The new revised weight is given by:

$$w_i' = \left( \frac{n_{op}}{n'_c} \right) w_i$$

where $w_i'$ and $n_c'$ are the updated weight and total number of threads respectively. $n_{op}$ equals to the total number of online processors.

In this algorithm, the multithreaded processes are not rewarded because the timeslice of greedy process is restricted by the amount of timeslice assigned to the process with the optimal number of threads.

## 3.4 Proposed Scheduler

This section proposes a new approach, TWRS, and discusses thread allocation of CPU time, experimental setup, and evaluation from the point of view of scheduling criteria and performance.

### 3.4.1 Overview

TWRS is a kernel-level thread scheduler to improve the treatment of execution of running processes, scheduling criteria, and enhance system's performance by

readjusting the weights of threads forked from the same process and in the same time preserving fairness. TWRS has been integrated with an existing scheduler that uses one run queue for one CPU, such as Linux 2.6. As its name suggests, TWRS depends on proportionally distributed CPU time between threads by changing their weights. The following sections explain the policy for allocating CPU time to running threads.

### 3.4.2 TWRS's Features

The fundamental difference of TWRS lies in the method used for readjusting the weights of the running threads. TWRS tries to reduce the gap of CPU usage between running processes. The main features of TWRS are:

- **Reducing greedy behavior of processes:**
  - TWRS reduces greedy behavior of processes by adjusting the weights of sibling threads, and
  - gives the processes with fewer threads more chance of getting CPU usage by reducing the deduction of CPU usage from the processes with fewer threads in favor of process with more threads.
- **Improving scheduling criteria:**
  - TWRS improves scheduling criteria (i.e., waiting time and turnaround time) by assigning new timeslices to running threads.
- **Preserving performance:**
  - TWRS works in concert with existing scheduler taking into account other system criteria, such as execution time, operations per second and others.

### 3.4.3 TWRS and Thread Allocation of CPU Time

Each thread $i$ is assigned a weight $w_i$. The share of the thread in time interval $[t_1, t_2]$ is a ratio of its weight to the sum of weights of all active threads in the run queue. This is denoted as:

$$Share_i(t_1, t_2) = \frac{w_i}{\sum_{\forall j \in R} w_j}(t_2 - t_1). \qquad (3.1)$$

where $R$ is the set of all threads that are currently in run queue of a core. $w_i$ is the weight of the thread.

The timeslice that a thread $i$ should receive in a period of time is given by:

$$slice_i = share_i \times period \qquad (3.2)$$

where period is the time quantum the scheduler tries to execute all threads.

If the number of threads does not exceed

$$\frac{sched\_latency\_ns}{sched\_min\_granularity\_ns}$$

then

$$period = sched\_latency\_ns$$

otherwise,

$$period = number\_of\_running\_tasks \times sched\_min\_granularity\_ns$$

where 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.

TWRS counts the number of total threads in the CPU and the number of sibling threads. Weights of the threads will be changed according to the next equation:

$$weight(thread_{i,j,k}) = weight(process_{j,k}) \times (processor_k(No\_threads) - process_{j,k}(No\_threads)) \qquad (3.3)$$

where $weight(thread_{i,j,k})$ is the weight of thread i which forked from process j in processor k, $weight(process_{j,k})$ is the weight of process j in processor k, $processor_k(No\_threads)$ is the number of all threads in processor k, and $process_{j,k}(No\_threads)$ is the number of threads of process j in processor k

### 3.4.4 The TWRS's Consideration

TWRS scheduling algorithm can be briefly described in the following steps:

1. Keep the order of processes according to CFS Red-Black tree.
2. If the ready queue is not empty, implement the equations to determine the weights.
3. Run each process for the calculated timeslice in one period.
4. In next period, calculate new weights and timeslices.

5. Repeat the step 3 till there are no processes waiting in the ready queue.

## 3.5 Experimental Setup

This section presents the underlying platform, software, and scheduling modes used in the experiments.

### 3.5.1 Underlying Platform

TWRS can be easily integrated with an existing scheduler based on per-CPU run queues. A TWRS prototype was implemented in Linux version 2.6.24 which based on CFS. The specification of the experimental platform is shown in Table 3.1.

**Table 3.1.** Specification of the experimental platform.

| **H/W** | |
|---|---|
| **Processor** | Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz |
| **CPU cores** | 1 |
| **Memory** | 8186756 kb |
| **S/W** | |
| **Kernel name** | Linux |
| **Kernel version number** | 2.6.24 |
| **Machine hardware name** | x86_64 (64 bit) |
| **version of Linux** | CentOS release 5.10 (Final) |

### 3.5.2 Software for Test

Three types of software were used in the experimental test:

**I. OpenMP API**

- OpenMP is an implementation of multithreading, a method of parallelizing whereby a master thread forks a specified number of slave threads and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors. Each thread has an ID attached to it which can be obtained using a function called omp_get_thread_num().
- The (#pragma omp parallel) is used to fork additional threads to carry out the work enclosed in the construct in parallel [13].

**II. Scientific Benchmark**

- This is a multithreaded scientific benchmark, Pi program to calculate 50,000,000 decimal digits using Chudnovsky Formula [17].

**III. SysBench Benchmark**

- SysBench [48] provides benchmarking capabilities towards Linux. It supports testing CPU, memory, file I/O, mutex performance and even MySQL.
- The idea of this benchmark suite is to quickly get an impression about system performance. Current features allow testing the following system parameters:
  - Memory allocation and transfer speed
  - Mutex performance
  - Scheduler performance
  - CPU performance
  - File I/O performance
  - Database server performance

### 3.5.3 Scheduling Modes

To assess the performance of the modified kernel, the experiment was employed with three multithreaded processes (Pi) and SysBench benchmark. Forking threads is implemented by using OpenMP. The processes run under two distinct scheduling modes: (1) The default scheduling in the Linux kernel, the proportional thread-fair scheduler chosen in the evaluations is the default Linux scheduler (CFS), and (2) The TWRS-augmented kernel.

In the default scheduling mode, the processes run on the original OS 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 timeslices to running threads.

## 3.6 Evaluation

As a proof of concept, two tests were carried out. The first test is to evaluate greedy behavior, it is used to quantitatively measure the usage of CPU received by concurrently running processes, and to evaluate scheduling criteria. The second test is used to evaluate the performance of TWRS.

TWRS was evaluated in two major scenarios, S0 for evaluating reducing greedy behavior of running processes, and scheduling criteria (i.e., waiting time and

turnaround time), and S1 for evaluating performance in each mode. The following tests were conducted with no other computation intensive applications running.

### 3.6.1 Greedy Evaluation

In greedy test, three instances of the same Pi program were executed concurrently to calculate 50,000,000 decimal digits using Chudnovsky Formula. To demonstrate the effectiveness of TWRS, some experimental data is presented for quantitatively comparing TWRS performance against CFS considered on different combinations of number of threads as explained in Table 3.2.

**Table 3.2.** Scenario S0 for greedy evaluation.

| S0 | No. of threads in process (A) "Pi" | No. of threads in process (B) "Pi" | No. of threads in process (C) "Pi" |
|---|---|---|---|
| | 1 | 8 | 10<br>20<br>30<br>40<br>50 |

- In S0, three executing instances (A, B, and C) of the same program Pi were executed concurrently. Two processes, A and B, with fixed numbers of threads, 1 and 8 respectively, and process C varies from 10 to 50 threads. All processes were initiated at the same time through a shell script where these processes were executed with the same nice value 0. This test was repeated fifty times and the average values were taken.
- When comparing the results represented in each group by the graphs in figures 3.3 and 3.4, it is observed that process C with more threads under CFS receives more CPU usage compared to TWRS, and processes A and B with fewer threads under TWRS receive more CPU usage compared to CFS.

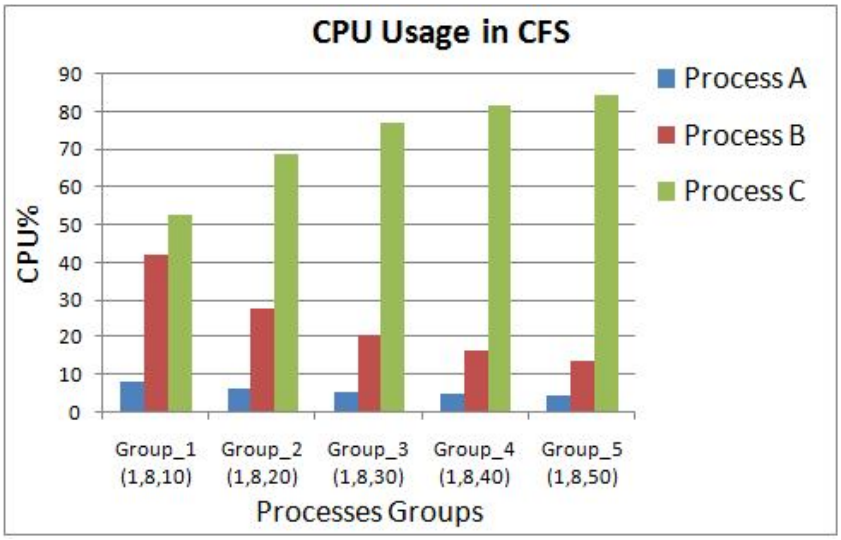



**Figure 3.3**. Five groups of three processes with different numbers of threads ((1,8,10), (1,8,20),….,(1,8,50)) run concurrently under CFS.

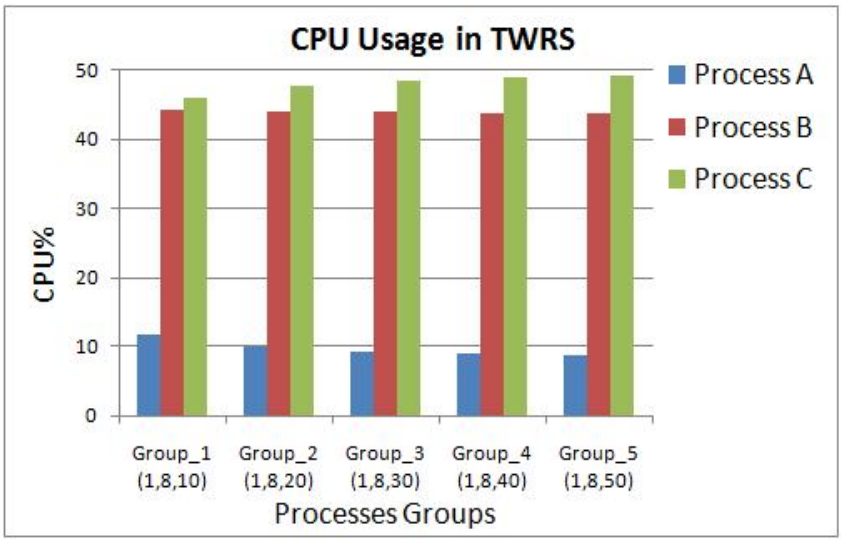



**Figure 3.4.** Five groups of three processes with different numbers of threads ((1,8,10), (1,8,20),….,(1,8,50)) run concurrently under TWRS.

In what follows, the following terminology list was used to measure greedy:

- $CPUU_1(X)$ is the CPU usage of process X in group 1.
- $G_1(Z, Y)$ is the gap of CPU usage in group 1 between process Z and Y, $G_1(Z, Y) = CPUU_1(Z) - CPUU_1(Y)$.

The difference of CPU usage between running processes in every group were calculated under each scheduling mode, the default scheduling and the modified one. Figure 3.5 shows that TWRS reduces the CPU usage gap between multithreaded processes in each group.

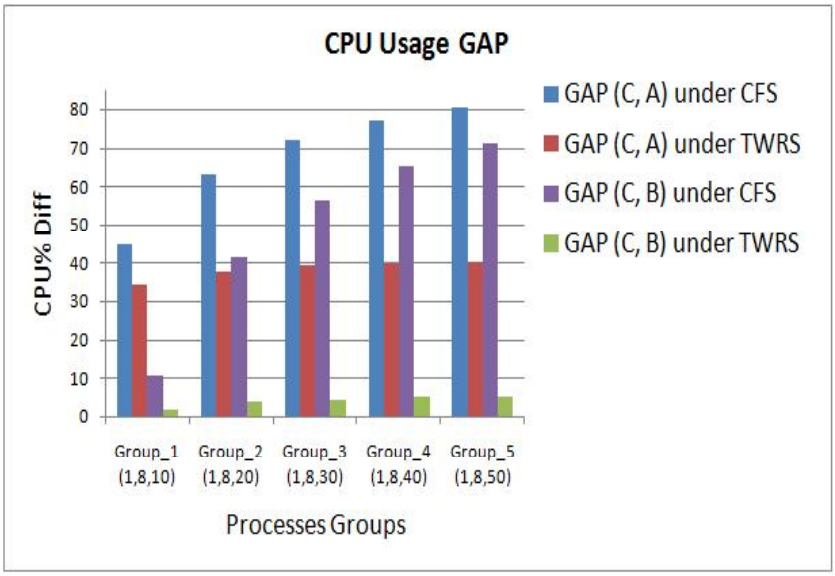



**Figure 3.5.** CPU usage gap comparison between processes in each group under TWRS and CFS.

### 3.6.2 Scheduling Criteria Evaluation

This subsection shows the contribution of adjusting the weights in maximizing the CPU utilization by minimizing waiting time and turnaround time. The average waiting time and turnaround time were analyzed when running the processes in S0. Figures 3.6 and 3.7 show that TWRS reduces the average waiting time and average turnaround time respectively.

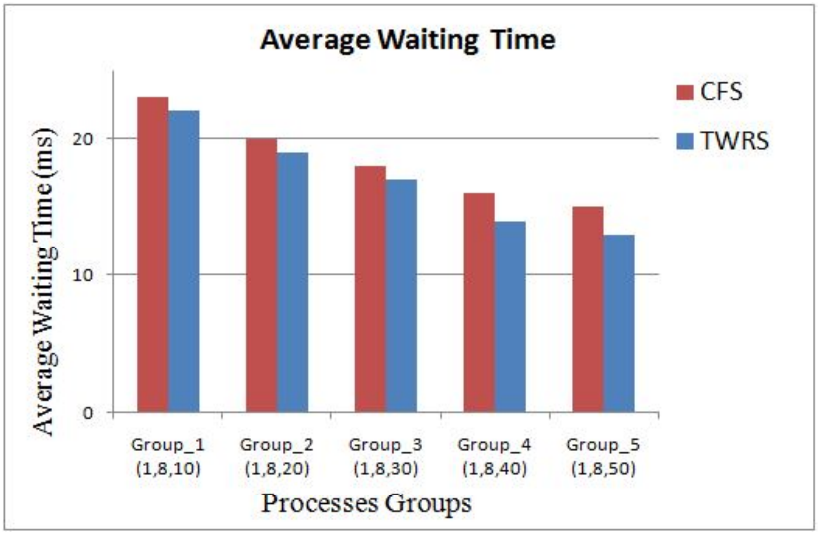



**Figure 3.6.** Average waiting time comparison of running processes in each group under TWRS and CFS.

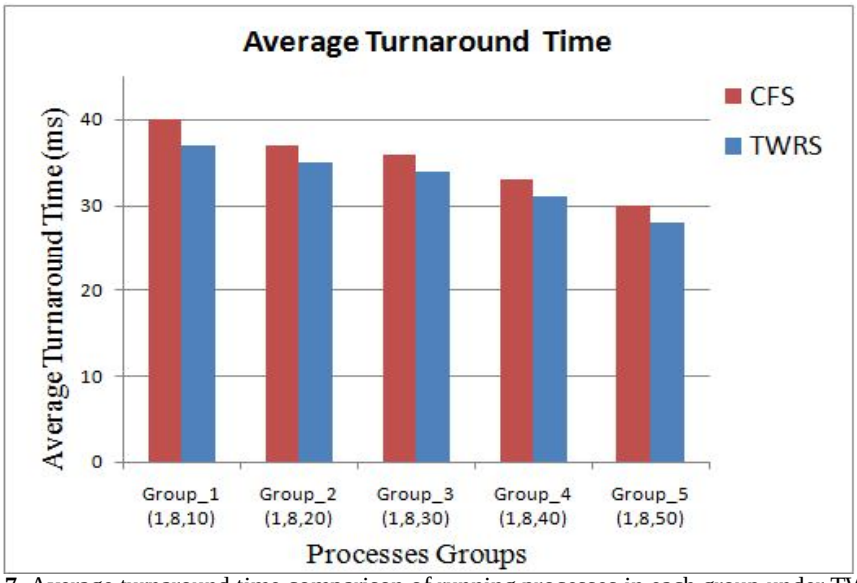



**Figure 3.7.** Average turnaround time comparison of running processes in each group under TWRS and CFS.

### 3.6.3 Performance Evaluation

This subsection evaluates TWRS's performance by showing that it enables slightly higher performance, or at least similar to that of unmodified Linux 2.6.24. Table 3.3 describes the benchmarks used in evaluating the performance.

**Table 3.3.** Benchmarks.

| | |
|---|---|
| **memory** | This test was written to emulate memory allocation and transfer speed. This benchmark application will allocate a memory buffer and then read or write from it. |
| **mutex** | This test mode was written to emulate a situation when all threads run concurrently most of the time, acquiring the mutex lock only for a short period of time. The purpose of this benchmarks is to examine the performance of mutex |
| **threads** | This test mode was written to benchmark scheduler performance, more specifically the cases when a scheduler has a large number of threads competing for some set of mutexes. |
| **cpu** | In this mode, each request performs calculation of prime numbers up to a value specified by the --cpu-max-primes option. |
| **fileio** | This test mode can be used to produce various kinds of file I/O workloads. At the prepare stage SysBench creates a specified number of files with a specified total size, then at the run stage, each thread performs specified I/O operations on this set of files. |
| **oltp** | This test mode was written to benchmark a real database performance. At the prepare stage a table is created in the specified database. |

Table 3.4 describes the second scenario used in the second test. This test ran single-thread process, process with 8 threads, and process with more number of threads (e.g. 20 threads), and each test in SysBench run with an arbitrary number of threads (e.g. 100 threads).

**Table 3.4.** Scenario S1 for performance evaluation.

| S1 | No. of threads in fixed program "Pi" | No. of threads in fixed program "Pi" | No. of threads in fixed program "Pi" | 6 Benchmark programs: memory, mutex, threads, cpu, fileio and oltp |
|---|---|---|---|---|
| | 1 | 8 | 20 | 100 threads for each |

- In S1, three instances of the same program Pi were executed concurrently, each instance has a fixed number of threads. Moreover, six benchmarks were executed to measure the TWRS's performance. All processes were initiated at the same time through a shell script where these processes were executed with the same nice value 0. Table 3.5 shows the results of CFS and TWRS. All the benchmarks achieve nearly identical performance under unmodified Linux and TWRS, demonstrating that TWRS achieves slightly higher performance.

**Table 3.5.** TWRS vs. CFS performance result.

| Performance | Metric | CFS | TWRS | Improvement |
|---|---|---|---|---|
| **Memory allocation and transfer speed** | operations per second (ops/sec) (greater value is better) | 399412.78 | 399495.34 | 0.02% |
| **Mutex performance** | time (sec) (smaller value is better) | 1.8899 | 1.8889 | 0.05% |
| **Scheduler performance** | time per request (sec) (smaller value is better) | 29.4553 | 28.59 | 2.94% |
| **CPU performance** | total time (sec) (smaller value is better) | 29.5 | 28 | 5.08% |
| **file I/O performance** | transferred data per second (kb/sec) (greater value is better) | 7.5 | 7.9 | 5.33% |
| **Database server performance** | transactions per second (trs/sec) (greater value is better) | 680.41 | 700 | 2.88% |

## 3.7 Discussion

CFS does not control greedy behavior of multithreaded processes, and all threads receive same amount of CPU time. PFS, TFPS and TWRS solve the greedy issue. PFS assigns the same amount of CPU time to all processes regardless the number of threads, and considers the number of threads in each process not in all processes. On the other hand, TFPS assigns CPU time slice to processes depending on the number of cores not all running threads in all processes. The main advantages of TWRS are: It gives single-thread process a chance to get more CPU usage, reduces the gap of CPU

usage between running processes, and considers the number of all running threads in the processor.

# CHAPTER 4:
# Global Fairness

This chapter summarizes the shortcoming of current Linux scheduler. Current Linux scheduler can not solve load imbalance in some cases, and therefore, processes running in the less loaded core run more than expected relatively to other running processes in the more loaded core. This shortcoming is illustrated through an illustrative example. The chapter discusses some of related researches. Then, it proposes a modified thread migration to overcome this shortage, describing the mechanism of the proposed scheduler. The chapter compares the CPU time assigned to the running processes, average waiting time, average turnaround time, and performance of the proposed scheduler versus current Linux scheduler.

**Chapter organization:** Section 4.1 describes the motivation. In Section 4.2, the problem statement is defined. Related research is discussed in Section 4.3. In Section 4.4, the proposed scheduler is presented. Experimental setup is discussed in Section 4.5. Section 4.6 evaluates the proposed scheduler. Discussion is stated in Section 4.7.

## 4.1 Motivation

Multicore commodity processors have emerged and are currently the mainstream of general purpose computing [40]. CMPs have emerged as the widespread architecture choice for modern computing platforms [54]. It is expected that the degree of on-chip parallelism will significantly increase and processors will contain tens and even hundreds of cores [9, 22]. CMP executes multiple threads in parallel across the multiple cores [15, 46]. The scheduling criteria must be considered when designing OS scheduler for the purpose of achieving some performance goal(s), such as achieving better fairness, maximizing throughput, minimizing communication delays, minimizing execution time, maximizing resource utilization, and/or others, depending on the purpose of designing the OS [31, 32, 45]. Fairness acquired the maximum importance when designing the scheduler, and many of scheduling algorithms have been studied in order to attain accurate fairness. This can be done by minimizing the gap between their proposed one and the ideal algorithm (i.e., GPS) [28, 35]. Linux 2.6.23 kernel release introduced the first fair scheduler implemented in GPOS to replace earlier RR mechanism. The key idea behind CFS is to assign a

specific weight to a process and provide it with a CPU time proportionally to its weight [23]. CFS does not ensure accurate global fairness for multicore system. CFS endeavors to alleviate this problem by balancing the load among cores, but load imbalance is not avoidable in real world applications. From the view of global fairness between cores, CFS fails to devote CPU time to processes proportionally to their weights [24].

## 4.2 Problem Statement

This section discusses the problem statement. Subsection 4.2.1 elaborates the migration mechanism of the current scheduler. Subsection 4.2.2 illustrates the problem with example.

### 4.2.1 CFS Migration Mechanism

The migration in CFS is handled across cores via its load balancing mechanism [24]. Unfortunately, when migration is triggered in multicore system, fairness is not guaranteed. The following discussion summarizes the most important aspects of this scheme. The CFS defines the load of a core's run queue, $Q_k$ , as:

$$Load_k = \sum_{\tau_i \in S_k} W(\tau_i) \qquad (4.1)$$

where $S_k$ is the set of processes in $Q_k$, and $W(\tau_i)$ is the weight of process $\tau_i$. There are two cases to trigger load balancing, when a run queue becomes empty or at predefined time intervals. The key idea of the load balancing is moving processes from the busiest run queue ($Q_{busiest}$) to $Q_k$, and the amount of load to be moved is defined as:

$$Load_{imbal} = min(min(Load_{busiest}, Load_{avg}); Load_{avg} - Load_k) \quad (4.2)$$

where $Load_{avg}$ is an average system load, and $Load_{busiest}$ is the load of $Q_{busiest}$, the migration is only triggered if

$$Load_{imbal} \geq \min_{\tau_i \subset S_{busiest}} (W(\tau_i))/2 \qquad (4.3)$$

where $S_{busiest}$ is the set of processes in $Q_{busiest}$.

The time share $S_i$ of a process $i$ running on a certain core is calculated as:

$$S_i = \frac{\frac{1024}{1.25^{n_i}}}{\sum_{j=1}^{N} \frac{1024}{1.25^{n_j}}} \qquad (4.4)$$

where $n_i$ is the nice value of process i and N is the total number of running processes in the core. It is worth noting that the time share of a particular process $\tau_i$ is calculated relatively to the nice values of all processes currently assigned to execute at the same core [20].

### 4.2.2 Illustrative Example

It is observed from the motivating example shown in Figure 4.1 that CFS fails to achieve fairness in a multicore system. In this example, five processes are running on two cores. Their nice values and weights are specified on the top of the figure. By following CFS's allocation mechanism, it allocates a process on the core with smallest weights. In this example, T1 which has a weight equals to 1024 is assigned on either one of the cores and the remaining processes are on the other, where the weight of each of them is 335.

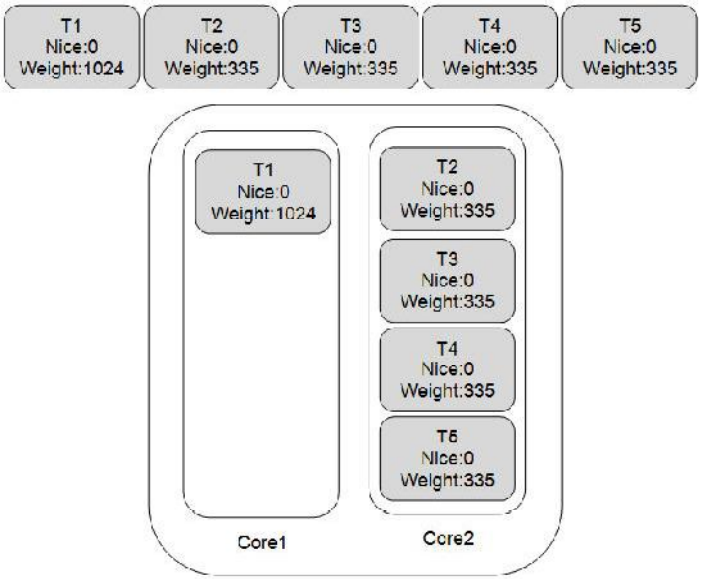



**Figure 4.1.** Five CPU-intensive processes are running on two cores.

According to their weight difference, it is supposed that T1 should run for about 3.06 times longer. By considering the mentioned equations, the load imbalance in this example is not large enough (the imbalance is 158 and it needs to be at least 335/2=167.5 to trigger the balancing), and therefore, all the processes remain running

on the same cores. Therefore, The weights of run queues of core1 and core2 are 1024 and 1340 respectively, and remain constant during any time interval. This leads to that T1 will run for four times longer than processes T2 to T5. This is considered as an unfairness matter from the view of multicore system.

## 4.3 Related Research

PSS has its roots in OSs. This section discusses prior designs.

### 4.3.1 Generalized Processor Sharing

GPS [28, 35] is an idealized scheduling algorithm that achieves perfect fairness, and all schedulers use it as a reference to measure fairness. Its model can be summarized as follows: consider a system with $P$ CPUs and $N$ threads. Each thread $i, 1 \leq i \leq N$, has a weight $w_i$. A scheduler is perfectly fair if it allocates CPU time to threads in exact proportion to their weights. Such a scheduler is commonly referred to as GPS. Let $S_i(t_1, t_2)$ be the amount of CPU time that thread $i$ receives in interval $[t_1, t_2]$. A GPS scheduler is defined as follows:

- If both threads $i$ and $j$ are continuously runnable with fixed weights in $[t_1, t_2]$, then GPS satisfies

$$\frac{S_i(t_1, t_2)}{S_j(t_1, t_2)} = \frac{w_i}{w_j}$$

- During the interval $[t_1, t_2]$, if the set of runnable threads, $X$, and their weights remain unchanged, then, for any thread $i$, GPS satisfies

$$S_i(t_1, t_2) = \frac{w_i}{\sum_{j \in X} w_j}(t_2, t_1)\, P$$

### 4.3.2 Lag-Based Algorithm

Ok et al. [34] proposed a lag-based load balancing scheme to guarantee global fairness in Linux-based multiprocessor systems. The fairness across multiple processors is provided through this approach and the notion of lag is introduced. When task $\tau_i$ is not scheduled during time $T_i$, the lag of task $\tau_i$ will increase by the amount of $\Delta lag_i$.

$$\Delta lag_i = T_i \times \frac{Weight_i}{\sum_{j \in \phi} Weight_j} \times N$$

where $Weight_i$ is the weight of task $\tau_i$, $\phi$ is the set of all the runnable tasks in the entire system, and $N$ is the number of CPUs. As the average load of the entire system is

$$Average\ Load = \frac{\sum_{j \in \phi} Weight_j}{N},$$

$\Delta lag_i$ can also be defined as below.

$$\Delta lag_i = T_i \times \frac{Weight_i}{Average\ Load}$$

Let $laxity_i$ denote the remaining time until task $\tau_i$ exceeds a certain specified lag bound without being scheduled. The $laxity_i$ for any task $\tau_i$ is defined by

$$laxity_i = \left\lfloor \frac{lag\ bound - lag_i}{\Delta lag_i} \right\rfloor$$

Whenever the Linux kernel makes a scheduling decision for each run queue, the proposed algorithm inspects if there exist more than one task that will have zero laxity at some identical time point. If found, only one of those tasks remains in the original run queue and other tasks are moved to less loaded run queues.

### 4.3.3 Progress Balancing Algorithm

Huh and Hong [23] proposed progress balancing algorithm for achieving multicore fairness. It works together with a per-core fair-share scheduling algorithm and runs periodically. Specifically, at every balancing period, it partitions tasks into the same number of task groups as the number of CPU cores in a system and shuffles the tasks to ensure that tasks with larger virtual run times run at a slower pace until the subsequent balancing period. Progress balancing periodically distributes tasks among cores to directly balance the progress of tasks by bounding their virtual run time differences. In doing so, it partitions runnable tasks into task groups and allocates them onto cores so that tasks with larger virtual runtimes run on a core with a larger load and thus proceed more slowly.

### 4.3.4 Virtual Runtime-Based Algorithm

Huh et al. [24] proposed a virtual runtime-based task migration algorithm that bounds the virtual runtime difference between any pair of tasks running on two cores.

The authors have also formally analyzed the behavior of the Linux CFS to precisely characterize the reason why it fails to achieve the fairness in a multicore system. Their algorithm consists of two sub-algorithms: (1) Partition, which partitions tasks into two groups depending on their virtual runtimes and (2) Migrate, which allocates the partitioned groups to dedicated cores while minimizing the number of task migrations. The authors also proved that their algorithm bounds the maximum virtual runtime difference between any pair of tasks.

## 4.4 Proposed Scheduler

To solve fairness issue, alternatives to the CFS migration scheme are proposed in this work to enable most processes to attain CPU time proportional to their weights and the weights of all running processes in all cores during time interval.

It is noticed from equation 4.2 that $\mathrm{Load}_{\mathrm{imbal}}$ depends on the load average, $\mathrm{Load}_{\mathrm{avg}}$, of all cores in the system, however, the CPU time assigned to each process, which calculated from equation 4.4, is considered per core and does not consider the total load of all cores. This leads to unfairness during time interval in a multicore system.

This section proposes a modification of CFS's migration mechanism to approximate GPS fairness. The proposed migration mechanism depends on the number of running processes on all cores. This algorithm classifies cores as light or heavy depending on the load of cores. The core with the more load is considered as heavy core and the core with less load is light. The proposed scheduler falls under the push migration approach to load balancing. The main features of the proposed scheduler are:

- the proposed algorithm pursues CFS's assignment of the processes in the first period;
- the difference between number of running processes in heavy and light cores must be greater than one ((nr_running[heavy] – nr_running[light]) > 1);
- the proposed algorithm prioritizes the CFS's migration mechanism;

- migrates the process with the smallest weight (it is called the shuttle process) from the heavy core to the light one in the second period and runs for one period; and
- the shuttle process is returned back to its original core (heavy) in the next period and run for one period, and so forth.

An additional flag is added to the process structure. The flag indicates whether the process is a shuttle and where it exists (e.g. in the light core or in the heavy core). When the processes are submitted for execution in the multicore environment, the flags are assigned to zero. The first assignment pursues the CFS's assignment. The proposed mechanism starts after the first period, and determines the heavy and light cores. As mentioned in Subsection 2.8.1, the three phases of the mechanism is as follows.

1. Condition phase:
   - Number of processes running in the heavy core must be greater than the number in light one by at least two.
   - According to equations 4.1, 4.2 and 4.3, proposed algorithm determines whether the load imbalance found or not.
2. Decision phase:
   - If a load to be transferred is found in the condition phase, the amount of the load to be transferred depends on CFS's migration mechanism.
   - Otherwise, the amount of load to be transferred is only one process with the least weight (shuttle process) in the heavy core.
3. Positioning phase:
   - The flag determines to which core the shuttle process should be moved. If it is 0, then the shuttle process is in the heavy core and should be moved to the light one,
   - and if it is one, the shuttle process is in the light and should be moved to heavy (original core).

The previous phases still continue until the run queue becomes empty. Figure 4.2 summarizes this algorithm in a flowchart.

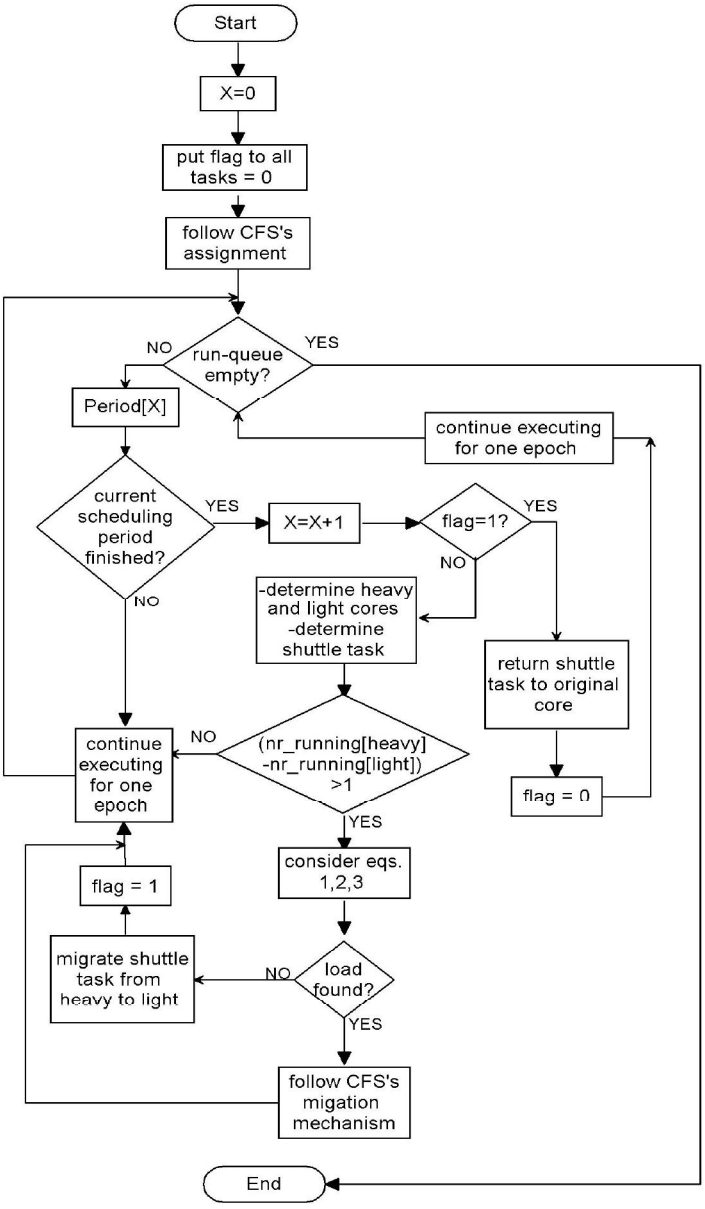



**Figure 4.2.** Flowchart of proposed scheduler.

## 4.5 Experimental Setup

This section presents the underlying platform, software, and scheduling modes used in the experiments.

### 4.5.1 Underlying Platform

Proposed scheduler can be easily integrated with an existing scheduler based on per-CPU run queues. The algorithm was implemented in the Linux kernel 2.6.24 which is based on CFS. The specification of our experimental platform is shown in Table 4.1.

**Table 4.1.** Specification of the experimental platform.

| H/W | |
|---|---|
| **Processor** | Intel(R) Core(TM)2 Duo CPU T7250 @ 2.00GHz |
| **CPU cores** | 2 |
| **Memory** | 2565424 kb |
| **S/W** | |
| **Kernel name** | Linux |
| **Kernel version number** | 2.6.24 |
| **Machine hardware name** | x86_64 (64 bit) |
| **Version of Linux** | CentOS release 5.10 (Final) |

### 4.5.2 Software for Test

Five multithreaded scientific benchmark, Pi, were executed in two cores concurrently. T1 with weight 1024 in core 1, and four processes each of them with 335 of weight in core 2.

### 4.5.3 Scheduling Modes

The processes run under two distinct scheduling modes: (1) The default scheduling in the Linux kernel, and (2) The proposed scheduler. In the default scheduling mode, the processes run on the original OS 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 implements the modified migration mechanism.

## 4.6 Evaluation

This section presents evaluation result for the proposed scheduler to demonstrate its effectiveness. As a proof of concept, two tests were carried out. The first test is to evaluate closeness to the ideal scheduler. The second test is used to evaluate the performance of the proposed scheduler.

### 4.6.1 Idealism Evaluation

Two metrics are used in this work to evaluate the closeness to idealism; the actual runtime assigned to each process after executing for a specific time (e.g. 200ms) compared to the ideal runtime that would have been given to the process under GPS, and lag difference between the GPS and the compared algorithms (proposed and CFS). The experiments were repeated fifty times and show the results in terms of the average of CPU time, lag measurements, and percentage error. Figures 4.3, 4.4 and 4.5 show that the proposed algorithm is closer to ideal one.

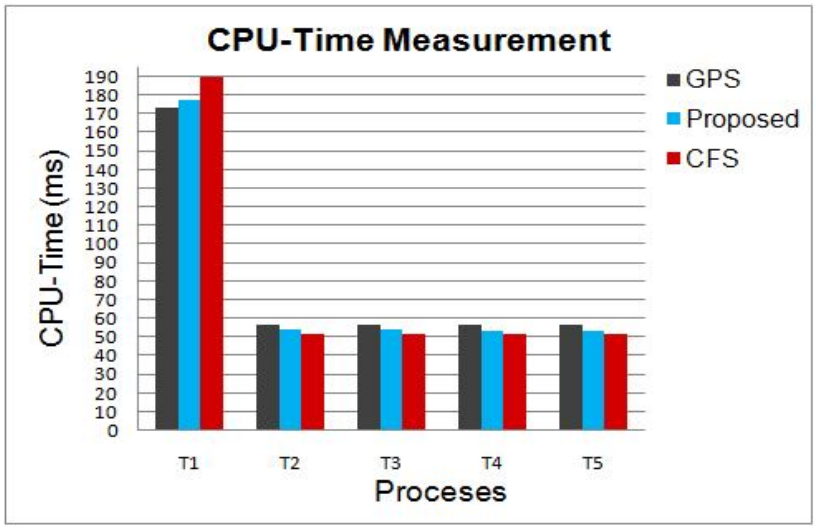



**Figure 4.3.** Comparing the CPU time when running five Pi processes relatively to GPS.

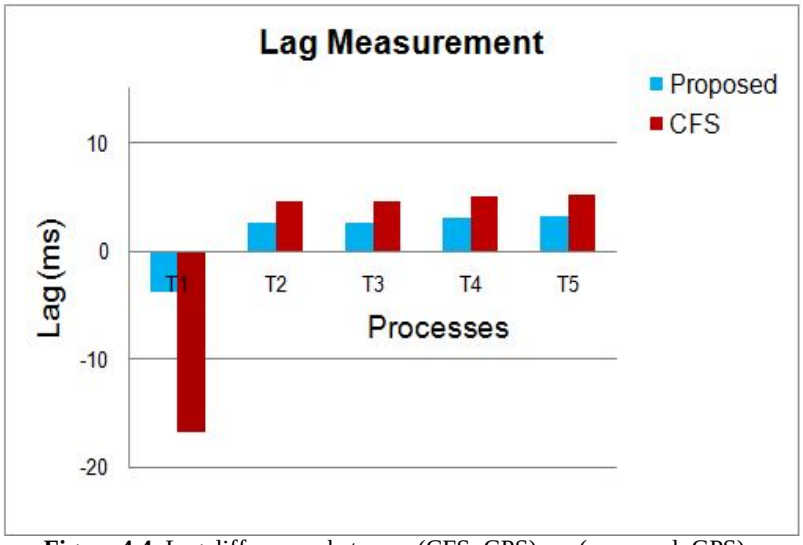



**Figure 4.4.** Lag differences between (CFS, GPS) vs. (proposed, GPS).

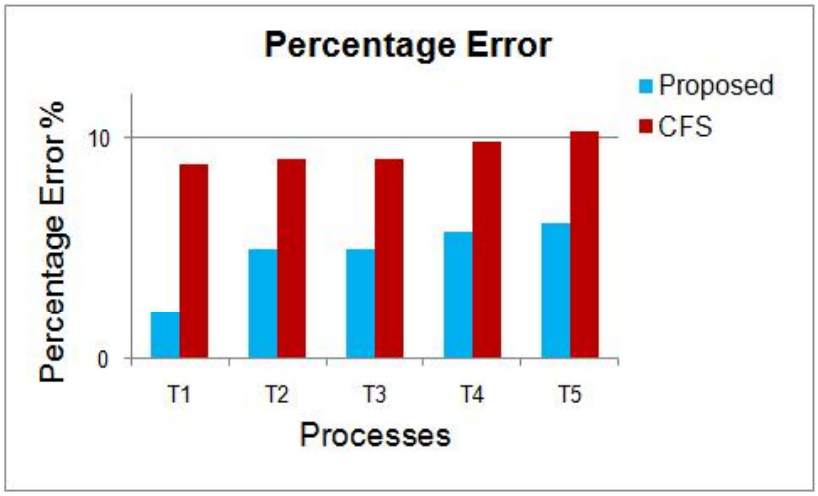



**Figure 4.5.** Percentage error between (CFS, GPS) vs. (proposed, GPS).

### 4.6.2 Scheduling Criteria Evaluation

This subsection shows the contribution of thread migration mechanism in maximizing the CPU utilization by minimizing waiting time and turnaround time. Figures 4.6 and 4.7 shows that the proposed scheduler reduces average waiting time and average turnaround time respectively.

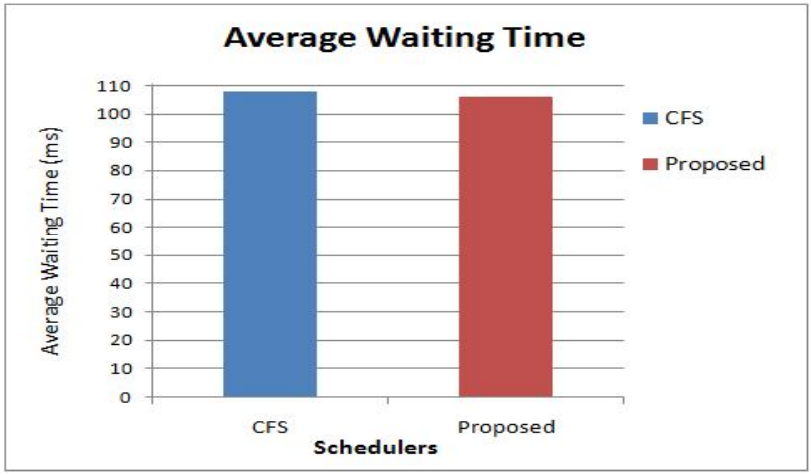



**Figure 4.6.** Average waiting time comparison of running processes under Proposed and CFS.

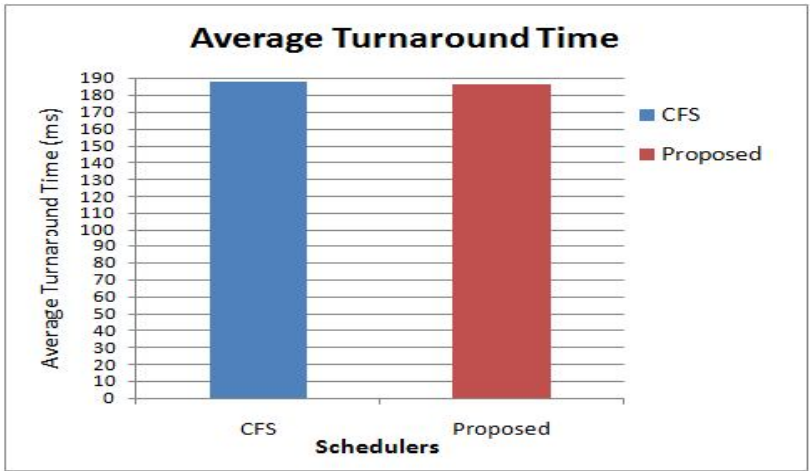



**Figure 4.7.** Average turnaround time comparison of running under Proposed and CFS.

### 4.6.3 Performance Evaluation

This subsection evaluates the performance of the proposed scheduler by showing that it enables higher performance, or at least similar to that of unmodified Linux 2.6.24. The benchmarks described in Table 3.3 were used in this test to evaluate the overall performance of the proposed scheduler.

Table 4.2. shows our results for CFS vs. the proposed scheduler. All the benchmarks achieve nearly identical performance under unmodified Linux and the proposed, demonstrating that our scheduler achieves slightly higher performance.

**Table 4.2.** Proposed vs. CFS performance result.

| Performance | Metric | CFS | Proposed | Improvement |
|---|---|---|---|---|
| **Memory allocation and transfer speed** | operations per second (ops/sec) (greater value is better) | 399525.78 | 399608.34 | 0.021% |
| **Mutex performance** | time (sec) (smaller value is better) | 2.2909 | 2.2899 | 0.044% |
| **Scheduler performance** | time per request (sec) (smaller value is better) | 40.4553 | 39.59 | 2.139% |
| **CPU performance** | total time (sec) (smaller value is better) | 30.8 | 29.3 | 4.870% |
| **file I/O performance** | transferred data per second (kb/sec) (greater value is better) | 8.8 | 9.2 | 4.545% |
| **Database server performance** | transactions per second (trs/sec) (greater value is better) | 757.41 | 777 | 2.586% |

### 4.6.4 Analytical Results

This subsection analyzes formally the properties of our scheduler. As mentioned in the problem statement section, processes running in the light core run more than expected relatively to other running processes in the heavy core. The main idea to help solving this problem is increasing the CPU time for most processes running in heavy core, and decreasing it for processes in light core to be closer to GPS than CFS. The notation is summarized in Table 4.3.

**Table 4.3.** Notation summary.

| Item | Description |
|---|---|
| $TS_{\tau_i,CFS[heavy]}$ | The timeslice of process $\tau_i$ which running in heavy core under CFS |
| $TS_{\tau_i,proposed[heavy]}$ | The timeslice of process $\tau_i$ in heavy core under proposed scheduler |
| $TS_{\tau_i,CFS[light]}$ | The timeslice of process $\tau_i$ running in light core under CFS |
| $TS_{\tau_i,proposed[light]}$ | The timeslice of process $\tau_i$ in light core under proposed scheduler |
| $N$ | Number of periods |
| $T$ | The length of the period |
| $w_{shuttle}$ | The weight of the shuttle process ($w_{shuttle} > 0$) |
| $\beta$ | The set of running processes in the heavy core |
| $\alpha$ | The set of running processes in the light core |

**Lemma:** Under the proposed scheduler, for any set of running processes in the heavy and light cores, then

$$TS_{\tau_i,CFS[heavy]} < TS_{\tau_i,proposed[heavy]}$$
$$and$$
$$TS_{\tau_l,CFS[light]} > TS_{\tau_l,proposed[light]}$$

**Proof**. Induction is employed. Suppose that the processes are running for N periods. According to the proposed scheduler, the migration of shuttle process will occur N/2 times.

- **Under CFS:**
  - In the heavy core, the CPU time of process $\tau_i$ is defined as:

$$TS_{\tau_i,CFS[heavy]} = \frac{w_i}{\sum_{j\in\beta} w_j} \times N \times T$$
$$= \left(\frac{w_i}{\sum_{j\in\beta} w_j} \times \frac{N}{2} \times T\right) + \left(\frac{w_i}{\sum_{j\in\beta} w_j} \times \frac{N}{2} \times T\right)$$

  - In the light core, the CPU time of process $\tau_l$ is defined as:

$$TS_{\tau_l,CFS[light]} = \frac{w_l}{\sum_{k\in\alpha} w_k} \times N \times T$$
$$= \left(\frac{w_l}{\sum_{k\in\alpha} w_k} \times \frac{N}{2} \times T\right) + \left(\frac{w_l}{\sum_{k\in\alpha} w_k} \times \frac{N}{2} \times T\right)$$

- **Under proposed scheduler:**
  - In the heavy core:

$$TS_{\tau_i,proposed[heavy]} =$$
$$\left(\frac{w_i}{\sum_{j\in\beta} w_j - w_{shuttle}} \times \frac{N}{2} \times T\right) + \left(\frac{w_i}{\sum_{j\in\beta} w_j} \times \frac{N}{2} \times T\right)$$

  - In the light core:

$$TS_{\tau_l,proposed[light]} =$$
$$\left(\frac{w_l}{\sum_{k\in\alpha} w_k + w_{shuttle}} \times \frac{N}{2} \times T\right) + \left(\frac{w_l}{\sum_{k\in\alpha} w_k} \times \frac{N}{2} \times T\right)$$

  - It is assumptive that:

$$\left(\frac{w_i}{\sum_{j\in\beta} w_j - w_{shuttle}} \times \frac{N}{2} \times T\right) > \left(\frac{w_i}{\sum_{j\in\beta} w_j} \times \frac{N}{2} \times T\right)$$

    and

$$\left(\frac{w_i}{\sum_{j\in\beta} w_j + w_{shuttle}} \times \frac{N}{2} \times T\right) < \left(\frac{w_i}{\sum_{j\in\beta} w_j} \times \frac{N}{2} \times T\right)$$

From previous inequalities, the CPU time of processes in the heavy core under proposed scheduler is greater than CPU time of processes in the heavy core under CFS, and the CPU time of processes in the light core under proposed scheduler is less than CPU time of processes in the light core under CFS.

Therefore, the lemma holds.

## 4.7 Discussion

The proposed scheduler in this chapter overcomes the disadvantages of previous algorithms. CFS assigns the CPU time to each process and does not consider the total load of all cores when assigning the CPU time to each process, however, the proposed scheduler does. The proposed scheduler takes into account the ratio of CPU times assigned to running processes proportional to their weight differences, however, Lag-Based, Progress Balancing, and Virtual Runtime-Based Algorithms do not.

# CHAPTER 5:
# VM Fairness

This chapter summarizes the shortcoming of the current scheduler when running VMs. The problem statement is divided into two parts, fairness and pricing. The problem statement is illustrated through experimental tests thus exposing its drawbacks. Then, the chapter proposes a modified scheduler to overcome this shortage. The chapter discusses the mechanism of the proposed scheduler. It compares the CPU usage assigned to the running VMs, and execution times of the running processes in the VMs when assigning different values of weights to VMs.

**Chapter organization:** Section 5.1 describes the motivation. In Section 5.2, the problem statement is defined. In Section 5.3, the proposed scheduler is presented. Experimental setup is discussed in Section 5.4. Section 5.5 evaluates the proposed scheduler. Discussion is stated in Section 5.6.

## 5.1 Motivation

VMs are really host OS applications. Since each VM is a process in the host OS, it is subject to the host OS scheduling algorithm. Therefore, all VMs that are running will be scheduled like normal applications. The main benefit of VMM is normally allowing a system manager to configure the environment in which a VM will run [26, 27, 53]. Therefore, VMs can have configurations different from those of the real machine. Sharing CPU resources between VMs is done by Time Multiplexing, as shown in Figure 5.1. VM is allowed direct access to resource (e.g., CPU resource) for a period of time before being context switched to another VM.

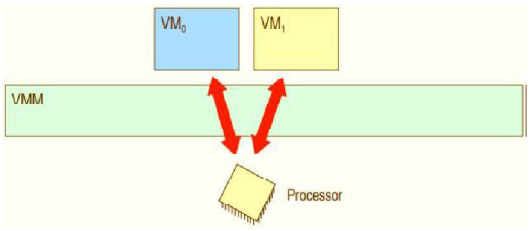



**Figure 5.1.** Sharing CPU resources between VMs.

In VM environment, VMs are available for purchase, each with their own varying levels of services and fees [3]. Current VM schedulers should grant user who pays amount of fee the services s/he wants. One important concept found in most

virtualization options is the implementation of a VCPU. The VCPU does not execute code. Rather, it represents the state of the CPU as the guest machine believes it to be. For each guest, the VMM maintains a VCPU representing the guest's current CPU state. When the guest is context switched onto a CPU, information from the VCPU is used to load the right context, much as a GPOS would use the PCB.

For virtualized systems, such as a public cloud, fairness between tenants and the efficiency of running their applications are keys to success [38]. To enforce fairness between VMs, Linux scheduler assigns equal shares to individual VMs and the shares are further evenly distributed to running threads. Thus, VMs with a smaller number of threads will receive a larger per-thread share. When a thread finishes running, its share is updated based on how long it ran on the PCPU.

Current Linux scheduler requires that the aggregate CPU allocation to all threads of a VM be proportional to its weight [16, 50] in competition with other VMs sharing the same set of pCPUs. If all VMs have the same weight, each VM should receive the same amount of CPU time no matter how many threads a VM has. This leads to that the existing virtualization platforms fail to enforce fairness between VMs with different number of running threads. The host scheduler's unawareness about the attributes (e.g. weight and number of threads) of the running processes in VMs causes the unfairness.

## 5.2 Problem Statement

The problem statement is divided into two parts, fairness and pricing.

### 5.2.1 Fairness

Current host scheduler does not guarantee fairness between VMs because:

- It does not take into consideration the attributes of the running processes in the VM.
- Therefore, it gives all the running VMs the same amount of CPU usage regardless the attributes of the running processes in the VM.

To describe the problem statement, experiments were performed under two scenarios, S0 to run processes in one machine, and S1 to run in VM environment. In the experiments, two instances of the same program (i.e., –test=cpu) in SysBench benchmark were executed. Both programs were initiated at the same time through a

shell script, where these two programs were executed with the same nice value 0 in each scenario as described in Table 5.1.

**Table 5.1.** Two scenarios of problem statement.

| **S0: One Machine** | | | | **S1: VM Environment** | | |
|---|---|---|---|---|---|---|
| | P_1 (No. threads) | P_2 (No. threads) | | | VM_1 (P_1, No. threads) | VM_2 (P_2, No. threads) |
| S0.1 | 6 | 6 | | S1.1 | 6 | 6 |
| S0.2 | 6 | 9 | | S1.2 | 6 | 9 |

- S0 ran two concurrently executing instances of the same program, –test=cpu, in SysBench in the same machine under two sub scenarios as follows:
  - In sub scenario S0.1, both programs have the same number of threads (i.e., 6 threads). Program 1 (P_1) and program 2 (P_2) get same CPU usage. Figure 5.2 shows this result. The execution time of P_1 is similar to the execution time of P_2.

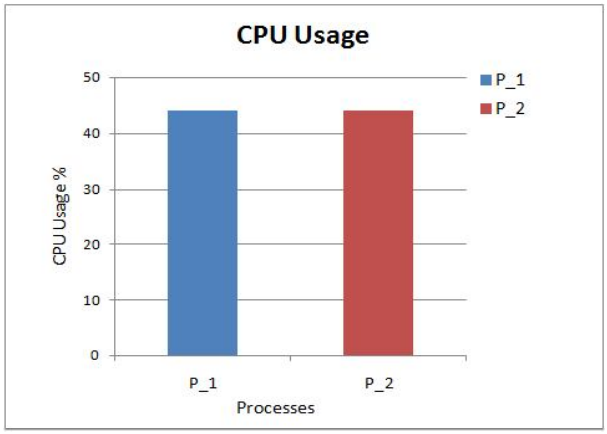



**Figure 5.2.** CPU usage when running two processes with same number of threads in one machine.

  - In sub scenario S0.2, P_1 has 6 threads, and P_2 has 9 threads. P_2 gets more CPU usage (i.e., 50%), and the program with 6 threads gets less CPU usage (i.e., 33%), as shown in Figure 5.3. The execution time of P_2 is less than execution time of P_1.

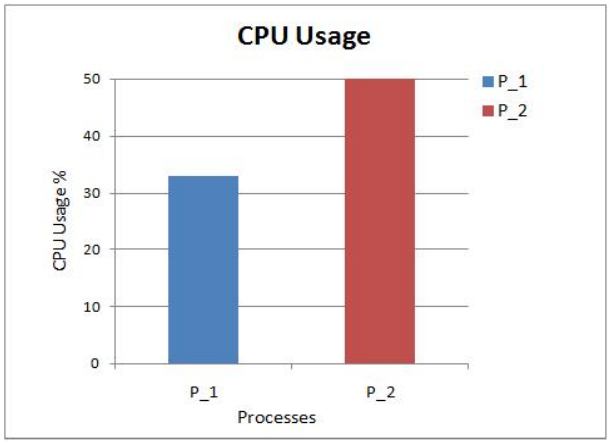



**Figure 5.3.** CPU usage when running two processes with different number of threads in one machine.

- S1 ran two concurrently executing instances of the same program, –test=cpu, in VM environment under two sub scenarios as follows:
  - In sub scenario S1.1, program 1 runs in VM_1, and program 2 runs in VM_2, and both programs have the same number of threads (i.e., 6 threads). Each of VM_1 and VM_2 consumes the same amount of CPU usage, as shown in Figure 5.4. The time which VM_1 uses to execute its running program (P_1) is similar to the execution time of the running program (P_2) in VM_2.

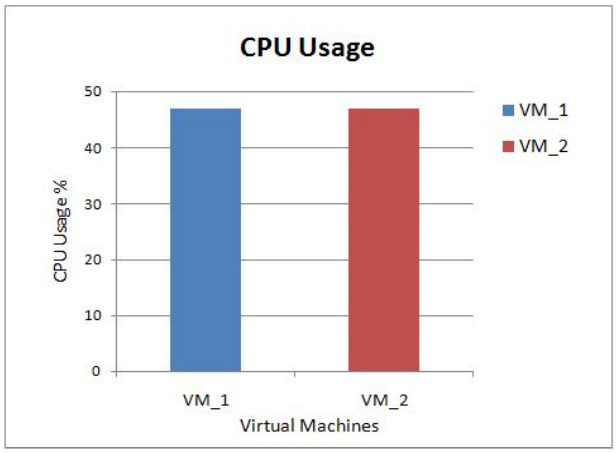



**Figure 5.4.** CPU usage when running two processes with same number of threads in VMs

  - In sub scenario S1.2, program 1 has 6 threads and runs in VM_1, and program 2 has 9 threads and runs in VM_2. Each of VM_1 and VM_2 consumes the same amount of CPU usage (i.e., 48%) as shown in Figure 5.5. The time which VM_1 uses to execute its running program (P_1) is less than the execution time of the running program (P_2) in VM_2.

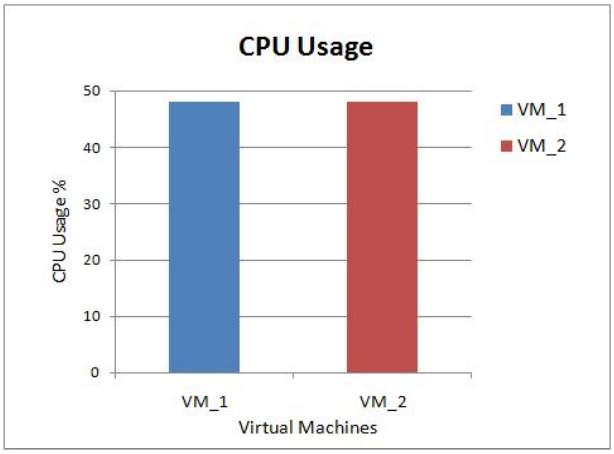



**Figure 5.5.** CPU usage when running two processes with different number of threads in VMs.

Under the concept of multithreading, the derived results from previous experiments are well-known under the scheduling in the first scenario. The problem appears when running same experiments in VMs.

### 5.2.2 Pricing

Creating a scheduling algorithm that captures the user's practical needs and requirements would be extremely useful in VM systems. The requirements vary from user to another, such that some users need to run their processes faster when submitting in VM environment to be executed. Unfortunately, current VM schedulers do not consider some demands (i.e., CPU usage) as services. Therefore, current pricing calculations do not consider CPU usage in their calculations. Integrating CPU usage in the pricing frame work serves both user and provider, user will benefit by receiving more CPU usage and therefore executes processes faster, and the provider will benefit by setting a new pricing rate which depends on the provided service.

## 5.3 Proposed Scheduler

Based on the discussion in the previous subsection, the host scheduler in the VM environment assigns the same CPU usage for all VMs regardless the number of running processes in each VM. To solve this problem, the current Linux scheduler was augmented with an additional system call that allows the users to flexibly control the weight of each VM by setting it to any arbitrary value as described in Algorithm 1.

| **Algorithm 1** | **Customized Scheduler** |
|---|---|
| 1. | **Variables**: the assigned weight of the ith VM is $W_i$; the number of VMs is M; 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. A period is the shortest time interval during which every VM in the system completes at least one of its CPU time. $CPUT(VM_i)$, CPU Time, is defined to be W.S, where W is the VM's weight and S is a timeslice, timeslice = period/M |
| 2. | Procedure Customized(void) |
| 3. | For each period |
| 4. | If $M \leq \frac{sched\_latency\_ns}{sched\_min\_granularity\_ns}$ |
| 5. | Then |
| 6. | $period = sched\_latency\_ns$ |
| 7. | Else |
| 8. | $period = M \times sched\_min\_granularity\_ns$ |
| 9. | End if |
| 10. | For each VM |
| 11. | /*Use system call to assign weight of $VM_i$ (i.e., $W_i$)*/ |
| 12. | $CPUT(VM_i) = W_i \times \frac{period}{M}$ |
| 13. | End for |
| 14. | End for |
| 15. | End procedure |

The user uses the system call to assign the weight's value for the VM (line 11). The assigned CPU time for each VM is determined from the equation in line 12. This algorithm delegates the pricing calculations to the VM environment provider.

## 5.4 Experimental Setup

This section presents the underlying platform, environment, and scheduling modes used in the experiments.

### 5.4.1 Underlying Platform

Customized scheduler can be easily integrated with an existing scheduler based on per-CPU run queues. To demonstrate its efficacy, the customized scheduler was implemented in Linux version 2.6.24 which based on CFS. The specification of the experimental platform is shown in Table 5.2.

**Table 5.2.** Specification of the experimental platform.

| **H/W** | |
|---|---|
| **Processor** | Intel(R) Core(TM)2 Duo CPU T7250 @ 2.00GHz |
| **CPU cores** | 1 |
| **Memory** | 2565424 kb |
| **S/W** | |
| **Kernel name** | Linux |
| **Kernel version number** | 2.6.24 |
| **Machine hardware name** | x86_64 (64 bit) |
| **version of Linux** | CentOS release 5.10 (Final) |

### 5.4.2 Proposed Environment

The host machine consists of VMs and the modified scheduler is integrated with the host kernel. Front End (client) accesses the host kernel in Back End (host machine) via SSH client software (e.g. MobaXterm). Figure 5.6 shows this environment.

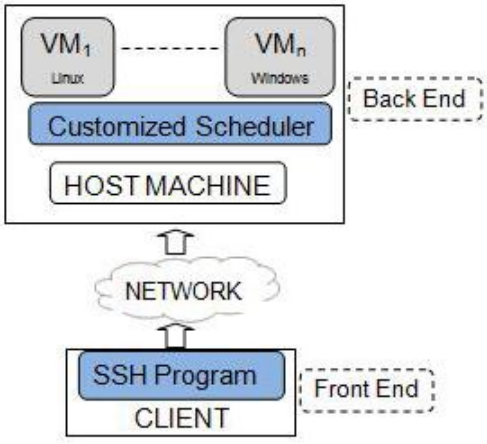



**Figure 5.6.** The proposed environment

Client runs his VMs and has the ability to flexibly control the weights of these VMs by setting them to any value.

### 5.4.3 Scheduling Modes

As a proof of concept, an experiment was performed in one scenario, S1, because the problem appears in this scenario only. The weights of the VMs were changed and the test measured the CPU usage and execution time of running processes. Varied weights are assigned for VMs as shown in Table 5.3.

**Table 5.3.** S1 scenario under VM environment.

| **S1: VM Environment** | | | |
|---|---|---|---|
| **VM_1 (No. threads)** | **Weight** | **VM_2 (No. threads)** | **Weight** |
| 6 | 1 | 6 | 1 |
| | | | 2 |
| | | | 3 |
| | | | 4 |
| | | | 5 |

- In S1, process in VM_1 has 6 threads and its weight is 1, and process in VM_2 has 6 threads and its weight varies from 1 to 5.

## 5.5 Evaluation

To demonstrate the effectiveness of the modification, some experimental data are presented for quantitatively comparing its performance considered on different combinations of values of weights in scenario S1. This test was repeated fifty times and the results in terms of CPU usage and execution times are shown in figures 5.7 and 5.8. The two programs were run concurrently and the average values were taken for 60 sec.

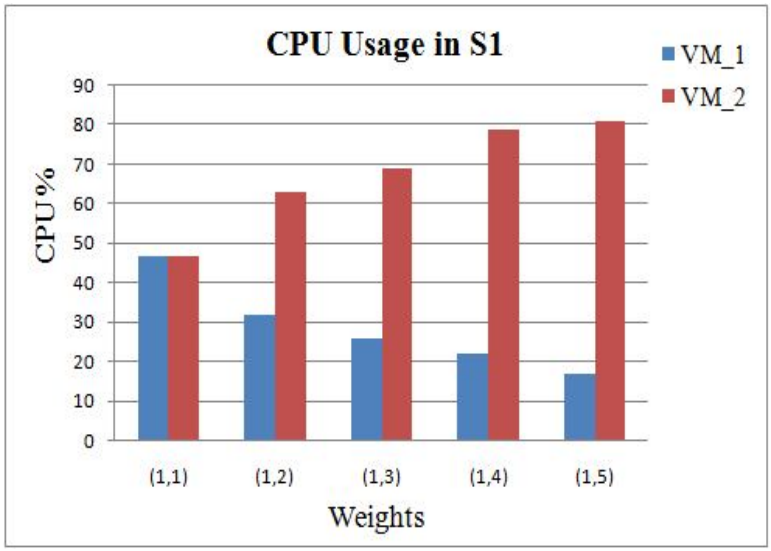



**Figure 5.7.** CPU usage in S1.


**Figure 5.8.** Execution time in S1.

The results show a significant improvement in both execution time and allocating CPU sharing to VM proportional to its weight.

## 5.6 Discussion

Current scheduler in VM environment assigns the same amount of CPU time to all running VMs no matter how many threads a VM has. It is expected that some users need to execute their processes faster, this can be done by receiving more CPU usage. Unfortunately, Current VM schedulers do not consider some demands (i.e., CPU usage) as services. In this work, the current scheduler was augmented with an additional system call enabling the users to flexibly control the weight of each VM by setting it to any arbitrary value, and therefore receives more CPU usage. On the other

hand, the VM environment provider will benefit by setting a new pricing rate which depends on the provided service to the users. I did not find any research related to this matter, therefore, I claim that the proposed modification in this chapter is the first to promote this issue.

# CHAPTER 6:
# Conclusion and Future Works

The purpose of an OS is to share computational resources among competing users. Independent users submit processes with varying resource requirements. An OS is expected to schedule this unpredictable mixture of processes in such a manner that the resources are utilized efficiently.

OS provides an environment in which a user can execute programs in a convenient and efficient manner. The OS must ensure the correct operation of the computer system. I describe the basic computer architecture that makes it possible to write a correct OS. Task/process scheduling in OS acts the major factor in achieving desired criteria. In this research, I focused on changing the behavior of the current Linux scheduler by adjusting the weights of threads running in the same core for the purpose of improving the treatment of execution of running processes, modifying the thread migration for the purpose of improving the load balancing mechanism of CFS and in the same time preserving fairness and performance, and adding a new system call for the purpose of allowing the user to flexibly control the weight of the VM.

Fairness acquired the maximum importance when designing the scheduler, and many of scheduling algorithms have been studied in order to attain accurate fairness. Also, achieving scheduling criteria is an essential goal and important factor in designing scheduler. The definition of fairness depends on the environment in which the processes are running. Better treatment of running processes, scheduling criteria and fairness can be achieved via two approaches, adjusting weights of running processes and load balancing.

The experiments were carried out in general purpose platform because most computers in use today are general purpose computers and are designed to perform a wide variety of functions and operations. Simply by using a general purpose computer and different software, various tasks can be accomplished, including writing and editing (word processing), manipulating facts in a data base, making scientific calculations, and so on. Other types of computers (e.g. special purpose computer) are designed to be task specific and most of the times their job is to solve one particular

problem. They are also known as dedicated computers, because they are dedicated to perform a single task over and over again. I chose Linux as a test-bed OS because it is a widely used OS both in research and business uses. Although Linux was originally developed as a desktop OS experiment, it will be found on servers, mainframes, and supercomputers, and it is a free open source OS. It is possible to modify and create variations of the source code, known as distributions, for computers and other devices. The most common use is as a server, but Linux is also used in desktop computers, smartphones, e-book readers and gaming consoles, etc.

The work of this thesis is divided into three stages, the first stage is: Improving the treatment of the execution of running processes and achieving better scheduling criteria without causing defect on the performance and fairness. In some cases, the process will be executed only on a designated core/CPU rather than any core/CPU this is called Processor affinity, or CPU pinning. Therefore, the goal in the first stage can be done per core by adjusting the weights of running threads (called as local fairness).

The second stage is: Improving load balancing between cores and achieving better scheduling criteria without causing defect on the performance and fairness. Multicore processors have emerged and are currently the mainstream of general purpose computing. Therefore, the goal in the second stage can be done between cores by thread migration (called as global fairness).

The last stage is: Enabling the user to flexibly control the weight of running VM. VM has become a very promising paradigm for both consumers and providers in various fields of endeavor, such as business, science and others. Therefore, the goal in this stage can be done by adding system call to the scheduler (called as VM fairness).

The goal in the first stage can be done for running processes by adjusting their weights in each round. The proposed scheme adjusts the weights by implementing equations giving new weights and preventing any process from monopolizing the processor. This technique provided a significant improvement in the desired scheduling criteria and slightly higher performance, and it can be implemented in any OS.

CFS does not ensure accurate global fairness for multicore system. From the view of global fairness between cores, CFS fails to devote CPU time to processes proportionally to their weights. In the second stage, a novel load balancing mechanism is proposed to redistribute processes between cores in order to approximate accurate fairness presented by ideal scheduler, GPS. The proposed approach determines a specific process with the least weight in the heavy core (core with more load) to be transferred to the light core (core with less load), this process is called shuttle process and assigned a flag to enforce the transferring. The proposed scheduler was implemented under specific hardware platform. The derived results showed an improvement over current scheduler in terms of scheduling criteria and slightly higher performance.

The third stage showed that the current host scheduler assigns the same CPU usage for all VMs regardless the number of running processes in each VM. The current Linux scheduler was augmented with an additional system call that enables the user to flexibly control the weight of each VM by setting it to any arbitrary value. The results showed that allocating CPU sharing to VM proportion to its assigned weight serves the user to execute his submitted processes faster by controlling the weight of each VM by setting it to a desired value. The host provider will benefit by setting a new pricing rate.

The future research directions includes the extension of the algorithm so that it can be applied to a general multicore environment with many cores, and modification of the algorithm so that it can reduce cache misses and number of context switches. Other criteria including energy consumption, carbon emissions, etc are also promising area of further improvements. Beside allowing the user to assign the weights to the running VM, it will be more useful if the kernel became able to change the weights of VMs dynamically according to their attributes.

# BIBLIOGRAPHY


1. Aas J., "Understanding the Linux 2.6.8.1 CPU scheduler," Silicon Graphics, Inc., 2005.
2. Abeni L., Lipari G. and Buttazzo G., "Constant bandwidth vs. proportional share resource allocation," In Proceedings of the IEEE International Conference on Multimedia Computing and Systems, Florence, Italy, June 1999.
3. Ahmed A.E.S., Alsammak A.K. and Algizawy E., "Article: A New Approach to Manage and Utilize Cloud Computing Underused Resources," International Journal of Computer Applications, 76(11), pp. 29-36, 2013.
4. Banachowski S.A. and Brandt S.A., "The BEST Scheduler for Integrated Processing of Best-Effort and Soft Real-Time Processes," In Multimedia Computing and Networking, January 2002.
5. Bashir A., Doja M.N. and Biswas R., "Conceptual Improvement of Priority Based CPU Scheduling Algorithm Using Fuzzy Logic," International Journal of Fuzzy Systems and Rough Systems (IJFSRS), vol. 1, no. 1, June 2008.
6. Bashir A., Doja M.N. and Biswas R., "Improving the Performance of Round Robin Scheduling Using Fuzzy Logic," In proceedings of the International Conference on Advanced Computing and Communication Technologies for High Performance Applications sponsored by IEEE & CSI, Cochin, India, September 24-26, 2008.
7. Bashir A., Doja M.N. and Biswas R., "Improving the Performance of fair Share Scheduling Algorithm Using Fuzzy Logic," Mumbai, Maharashtra, India.ACM 978-1-60558-351-8, January 23–24, 2009.
8. Bennett J. and Zhang H., "WFQ: Worst-case Fair Weighted Fair Queueing," In Proceedings of INFOCOM '96, San Francisco, CA, March 1996.
9. Borkar S., Dubey P., Kahn K., Kuck D., Mulder H., Pawlowski S. and Rattner J., "Platform 2015: Intel processor and platform evolution for the next decade," White Paper, Intel Corporation, 2005.
10. Caprita B., Chan W.C. and Nieh J., "Group Round-Robin: Improving the Fairness and Complexity of Packet Scheduling," Technical Report CUCS-018-03, Columbia University, June 2003.
11. Caprita B., Chan W.C., Nieh J., Stein C. and Zheng H., "Group ratio round-robin: O(1) proportional share scheduling for uni-processor and multiprocessor systems," In USENIX Annual Technical Conference, 2005.
12. Chandra A., Adler M., Goyal P. and Shenoy P., "Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors," In Proceedings of the 4th conference on Symposium on Operating System Design & Implementation, p.4-4, October 22-25, San Diego, California, 2000.
13. Chapman B., Jost G., Vanderpas R. and Kuck D.J., "Using OpenMP – Portable Shared Memory Parallel Programming," MIT Press, Cambridge, Massachusetts, 2007.
14. Chen H.C., Jin H. and Hu K., "Affinity-Aware Proportional Share Scheduling for Virtual Machine System," Proceedings of the 9th International Conference on Grid and Cooperative Computing (GCC), Nanjing, pp. 75-80, November 1-5, 2010.
15. Chen J. and Iii W.W., "Multi-threading performance on commodity multi-core processors," In Proceedings of HPCAsia, 2007.
16. Cherkasova L., Gupta D. and Vahdat A., "Comparison of the three CPU schedulers in Xen," In SIGMETRICS, 2007.
17. Chudnovsky D.V. and Chudnovsky G.V., "Approximations and complex multiplication according to Ramanujan," In Ramanujan Revisited, Academic Press Inc., pp. 375-396 &

pp. 468-472, Boston, 1988.

18. Demers A., Keshav S. and Shenker S., "Analysis and Simulation of a Fair Queueing Algorithm," In Proceedings of ACM SIGCOMM '89, Austin, TX, pp. 1–12, September 1989.

19. Essick R., "An Event-Based Fair Share Scheduler," In Proceedings of the Winter 1990 USENIX Conference, USENIX Berkeley, CA, USA, pp. 147–162, January 1990.

20. Gaspar F.J.F., "Performance and energy-aware real-time scheduling for heterogeneous embedded systems," Thesis to obtain the Master of Science Degree in Electrical and Computer Engineering, 2014.

21. Henry G., "The Fair Share Scheduler," AT&T Bell Laboratories Technical Journal, 63(8), pp. 1845–1857, October 1984.

22. Hofmeyr S., Iancu C. and Blagojević F., "Load balancing on speed," In Proceedings of the 15th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, pp. 147–157, 2010.

23. Huh S. and Hong S., "Providing fair-share scheduling on multicore computing systems via progress balancing," Preprint submitted to The Journal of Systems and Software, September 15, 2015.

24. Huh S., Yoo J., Kim M. and Hong S., "Providing Fair Share Scheduling on Multicore Cloud Servers via Virtual Runtime-based Task Migration Algorithm," Accepted to 32nd International Conference on Distributed Computing Systems, 2012.

25. Kay J. and Lauder P., "A Fair Share Scheduler," Communications of the ACM, 31(1), pp. 44–55, January 1988.

26. Kim H., Lim H., Jeong J., Jo H. and Lee J., "Task-aware virtual machine scheduling for I/O performance," Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, Washington, DC, USA, March 11-13, 2009.

27. King S.T., Dunlap G.W. and Chen P.M., "Operating system support for virtual machines," Proceedings of the annual conference on USENIX Annual Technical Conference, p.6-6, San Antonio, Texas, June 09-14, 2003.

28. Li T., Baumberger D. and Hahn S., "Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin," In Proc. of the 14th ACM Symposium on Principles and Practice of Parallel Programming, February 2009.

29. Love R., "Linux Kernel Development," 3rd edition, Noval Press, ISBN 0-672-32720-1, 2010.

30. Mostafa S.M. and Kusakabe S., "Effect of Thread Weight Readjustment Scheduler on Scheduling Criteria," Information Engineering Express (IEE), vol. 1, no. 2, pp. 1-10, 2015.

31. Mostafa S.M. and Kusakabe S., "Effect of Thread Weight Readjustment Scheduler on Fairness in Multitasking OS," International Journal of New Computer Architectures and their Applications (IJNCAA), pp. 184-192, vol. 4, no. 4, 2014.

32. Mostafa S.M., Rida S.Z. and Hamad S.H., "Finding time quantum of round robin CPU scheduling algorithm in general computing systems using integer programming," International Journal of Research and Reviews in Applied Sciences (IJRRAS) 5 (1), pp. 64–71, 2010.

33. Nieh J., Vaill C. and Zhong H., "Virtual-Time Round-Robin: An O(1) Proportional Share Scheduler," In Proceedings of the USENIX Annual Technical Conference, June 2001.

34. Ok D., Song B., Yoon H., Wu P., Lee J., Park J. and Ryu M., "Lag-Based Load Balancing for Linux-based Multiprocessor Systems," Foundations of Computer Science

(FCS), Las Vegas, USA, July 2013.

35. Parekh A. and Gallager R., "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case," IEEE/ACM Transactions on Networking, 1(3), pp. 344–357, June 1993.

36. Rajagopalan M., Lewis B.T. and Anderson T.A., "Thread scheduling for multi-core platforms," HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems, May 2007.

37. Ramanathan P. and Stankovic J., "Scheduling algorithms and operating systems support for realtime systems," In: Proceedings of the IEEE, vol. 82, no. 1, 1994.

38. Rao J. and Zhou X., "Towards fair and efficient SMP virtual machine scheduling," 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming , pp. 273 -285, 2014.

39. Regehr J., "Some Guidelines for Proportional Share CPU Scheduling in General-Purpose Operating Systems," Work in Progress Session of the Proc. IEEE Real-Time Systems Symp. (RTSS ',01), December. 2001.

40. Rodrigues E., Madruga F., Navaux P. and Panetta J., "Multi-core aware process mapping and its impact on communication overhead of parallel applications," IEEE Symposium on Computers and Communications, pp. 811-817, 2009.

41. Shih C.S., Wei J.W., Hung S.H., Chen J. and Chang N., "Fairness scheduler for virtual machines on heterogonous multi-core platforms," ACM SIGAPP Applied Computing Review 13 (1), pp. 28– 40, March 2013.

42. Shirazi B., Hurson A.R. and Kavi K.M., "Scheduling and Load Balancing in Parallel and Distributed Systems," Wiley, May 14, 1995.

43. Shreedhar M. and Varghese G., "Efficient Fair Queueing Using Deficit Round-Robin," in Proceedings of ACM SIGCOMM '95, 4(3), pp. 231-242, September 1995.

44. Silberschatz A., Galvin P. and Gagne G., "Operating Systems Concepts with Java," John Wiley and Sons. 6Ed 2004.

45. Silberschatz A., Galvin P. and Gagne G., "Operating Systems Concepts," John Wiley and Sons. 9Ed. 2013.

46. Spracklen L. and Abraham S.G., "Chip Multithreading: Opportunities and Challenges," In Proceedings of the International Symposium on High-Performance Computer Architecture, pp. 248-252, 2005.

47. Stallings W., "Operating Systems: Internals and Design Principles," Wiley, Prentice Hall Press, Upper Saddle River, NJ, 2008.

48. SysBench Benchmarks.
http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf

49. Ullman J.D., "Polynomial complete scheduling problems," In Proc. of the fourth ACM symposium on Operating system principles, pp. 96–101, 1973.

50. Weng C.L., Liu Q., Yu L. and Li M.L., "Dynamic Adaptive Scheduling for Virtual Machines," Proceedings of the 20th International Symposium on High Performance Distributed Computing, San Jose, pp. 239- 250, June 8-11, 2011.

51. Wong C.S., Tan I.K.T., Kumari R.D. and Fun W., "Towards achieving fairness in the Linux scheduler," SIGOPS Oper. Syst. Rev. 42, 5, pp. 34-43, July 2008.

52. Wong C.S., Tan I.K.T., Kumari R.D. and Kalaiyappan K.P., "Iterative performance bounding for greedy-threaded process," In: TENCON IEEE Region 10 Conference, Singapore, 2009.

53. Xi S., Xu M., Lu C., Phan L.T.X., Gill C., Sokolsky O. and Lee I., "Real-time multi-core virtual machine scheduling in Xen," In Proc. of ACM International Conference on

Embedded Software (EMSOFT'14), October 2014.

54. Zhuravlev S., Saez J.C., Blagodurov S., Fedorova A. and Prieto M., "Survey of scheduling techniques for addressing shared resources in multicore processors," ACM Computing Surveys, vol. 45, no. 1, 2013.

# INDEX