UNIVERSITY OF OSLO
Department of Informatics

A Comparison of Two Linux Schedulers

Master thesis

Gang Cheng

Network and System Administration

Oslo and Akershus University

College of Applied Sciences

Spring 2012

A Comparison of Two Linux Schedulers

Gang Cheng

Network and System Administration
Oslo and Akershus University College of Applied Sciences

Spring 2012

Acknowledgements

- I would like to express my deep and sincere gratitude to my supervisor, Æleen Frisch, for her kind inspiration, guidance and encouragement. Not only she gave me lots of ideas, helped me through out the process of writing thesis but also she was such an open and direct person, from whom I learned a lot.
- I would like to give my thanks to Kyrre Matthias Begnum, who has opened a new door to the research world to me. He has made great contributions to this master project, and set a good example for us.
- It is also my pleasure to thank Hårek Haugerud for his kind help during the whole period of this master program.
- I would also like to thanks my colleague George Magklaras, who shares his experience and knowledge with me during my work in the Biotechnology Center of Oslo and the Norwegian Center for Molecular Medicine.
- It is always fun to work my classmates, for they not only provide you advices but also funny jokes, which really make life interesting.
- As Donna Hedges said, Having a place to go - is a home. Having someone to love - is a family. Having both - is a blessing. I want to give my deep thanks to my family, for the continuous love from them.
- Last, special thanks to my girlfriend, who not only spend much time assisting me to plot all the data, but also inspired me to get through all the difficulties.

Abstract

The scheduler is the key component of the Linux kernel. What makes the scheduler crucial is that scheduler serves as a mediator between system resources such as CPU, and requests. It schedules and manages processes to have access to resources based on different data structures and algorithms. It is very important for a scheduler to make the best use of system resources.
From the beginning of Linux history, there have been a variety of different schedulers. Some are still in use, while some have become out of date. CFS and O(1) are two common schedulers which are still serving users. Since these two schedulers have different designs and performances, it is necessary to discover the differences between them, because users could then choose different schedulers based on their requirements.
This thesis compares CFS and O(1) with respect to the theoretical differences, by literature survey, and performance differences by testings. Differences then are presented in this thesis, and based on the results from testings, some recommendations are made to users.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Linux has been widely used not only in enterprise system but also in personal environment worldwide. As a multiple processes system, it is critical for Linux to distribute resources to different tasks in the system to process tasks at the same time; therefore, a task scheduler is needed and becomes key component of Linux. So the scheduler is designed to switch different processes in the system to make the best use of resources.
There are two schedulers right now, the older O(1) scheduler in the versions of Linux kernel prior to 2.6.23, and the new CFS (Completely Fair Scheduler) in the versions of Linux kernel 2.6.23 and after that.
The O(1) scheduler was announced on January 4th 2002 after the previous O(N) scheduler. The O(N) scheduler would go through the whole list of tasks in the queue to determine the next task based on certain function, and it is not well designed for multi-processors architecture, which means the O(N) scheduler is relatively inefficient, unstable, and week for real-time systems.[1] The O(1) scheduler then was designed to solve the problem of O(N) scheduler. It schedules tasks based on the priorities of different tasks. The priority of one specific task is determined by the schedule function and is assigned dynamically. The O(1) scheduler does not go through the whole list of all tasks in the queue, however, it separates all the tasks into two different queues: one active and one expired, and chooses the next task from the active queue. Besides, the O(1) supports multi-processors architecture. Those changes improve the performance of O(1) schedule. For example, O(1) is more efficient and scalable with number of task or processor than O(N).
The O(1) scheduler thus is still widely used now, for example, in the Red Hat Enterprise Linux 5.7. However, CFS takes the spot light later.
The CFS(Completely Fair Scheduler) , is quite different from O(1) scheduler. CFS has modular scheduler core, each core is enhanced with scheduling classes, which represent scheduling policies.
The main idea of CFS is to make every task have the "fair" processor time.

When there are tasks that are out of fair processor time, CFS will determine to give them time to execute. This process is called to make a balance. To maintain this balance, CFS uses a concept called red-black tree structure which is quite different from the run queue structure in O(1). This red-black tree is a time-ordered and self-balanced binary tree. The operation time in the tree occurs in O(log n) time, where n is the total number of nodes in the tree. The advantage of red-black tree is that the operation of tree, for example to insert or delete task from tree, is quick and efficient. CFS has been the default scheduler since the Linux kernel version 2.6.23.
As we know, O(1) scheduler and CFS are quite different from each other, and both of them are widely used. So, for system administrators, here comes the question: which has better performance? This project reviews the mechanisms of two schedulers, compares performance of these two schedulers on a variety of system workloads in order to make recommendations to system administrators. All the tests are primarily aimed at the personal computers environment.

1.2 Problem statement

Compare two Linux schedulers: O(1) and CFS

* What are differences between of O(1) and CFS?

* Which scheduler gives better performance under different types system workloads?

Red Hat Enterprise Linux 5.7 and Red Hat Enterprise Linux 6.2 are installed in two different hard disks, and both of them are in one machine sharing the same network, CPU and other devices. Different sets of tests will be executed to benchmark both schedulers, and those tests are classified on different types of processes. Basically, processes are classified based on their purposes and behaviors. In this project, the following types of processes will be executed toward both schedules.

* CPU-bound processes
* I/O-bound processes
* Mixed processes
* Interactive processes

More detailed definitions of these four types of processes will be covered in the next background chapter 2.3.4.

1.3 Thesis Outline

This thesis will be structured as follows:
Chapter 1 gives a general idea of whole project, and presents motivations, problem statements and thesis structure.
Chapter 2 introduces readers to the background, previous literature and other related topics.
Chapter 3 presents the methodology and testing plans of schedulers.
Chapter 4 shows the results from the tests.
Chapter 5 analyzes the results.
Chapter 6 Makes discussion and conclusion, and recommendation are suggested for users.

Chapter 2

Background

2.1 Linux Kernel Overview

The kernel is the key component of operating system. The main responsibility of scheduler is to manage the system resources. The management of resources includes several contexts. First, the kernel provides an interface for applications to have access to the hardware resources. When applications send requests for hardware resources such as address space, the kernel receives requests and uses system calls to communicate with those applications. Second, the kernel allocates resources such as CPU and memory. Third, the kernel software is generally organized into subsystems. Subsystems logically map to resources the kernel is dealing with[2]. Those systems include processes, memory management, file systems, device control and networking. The kernel creates and destroys processes, and schedules those processes for execution using its scheduler subsystem.
The following figure 2.1 [3] illustrates the Linux kernel and its subsystems.

In summary, the kernel's jobs include managing resources, handling requests, tracking processes and allocating resources. Allocating and servicing requests are part of resource management. Kernels also perform internal management that is not directly related to services. The kernel has to track what resources it is using and often collects information about various aspects of the system.

2.2 Linux Scheduler History and Literature

As discussed in the previous section, the scheduler plays a key role within kernel. The Linux kernel operating system is now used for many different versions, such as servers, desktops and embedded systems. The scheduler has been developed and modified along with the kernel. The early Linux scheduler, 1.2 Linux scheduler, was very simple. It used a circular queue of runnable tasks that operated with a round-robin scheduling policy. This scheduler was

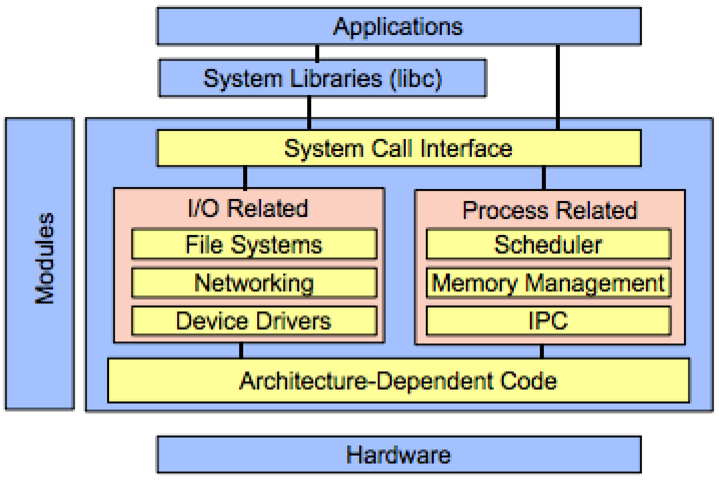


Figure 2.1: Linux Kernel [3]

very efficient in adding and removing processes in the queue (and included a lock to protect the structure) [4].

The 2.4 Linux kernel version introduced the O(N) scheduler, which was much better than the previous scheduler with its simple circular queue. The O(N) scheduler used an algorithm with O(n) complexity to select a task to run. The O(N) scheduler examined the whole list of runnable process in the system, and the time for O(N) scheduler to schedule a process scaled linearly with the number of processes. In case of large number runnable processes, the scheduler spent a great deal time of scheduling, resulting in less time assigned to processes. Obviously, the O(N) had disadvantages handling many simultaneous tasks [5].

On January 4th 2002, the O(1) scheduler was announced as the replacement for the O(N) scheduler. As the name indicates, the O(1) scheduler doesn't need to go through the whole list of process in the system. Instead, the O(1) scheduler keeps two running queues. The first running queue is the list of processes that are active and ready to be executed. The second is the list of processes that are expired. When it must select a process to run, the O(1) scheduler only chooses from the active queue. This makes O(1) much more efficient and scalable than that of O(N).

The latest Linux scheduler is called the Completely Fair Scheduler (CFS). Released on October 9th 2007, CFS uses a completely different approach. As the name indicates, the idea behind the CFS is to provide fairness with respect to all processes, in other words, to ensure that all processes are assigned with a fair share of processing time. Unlike previous scheduler, CFS doesn't use run queues. Instead, it maintains a time-ordered red-black tree to build a timeline of future task execution [6].

As the scheduler is a key component of the kernel, there is a great deal of documentation of Linux kernel that includes the scheduler [7, 4].
Understanding the Linux Kernel, by Daniel P. Bovet, Marco Cesati, published in October 2000, is one of the valuable books that deserves reading. In chapter 10, it covers processes scheduling, including scheduling policy and scheduling algorithms. Besides, there are many articles, documentation with respect to scheduler. For example, M. Tim Jones, who is an embedded firmware architect and the author of many books, wrote a summary of Linux scheduler history [4]. The research towards Linux scheduler differs from interactive processes scheduling to scheduler tuning [7].

This paper, aims to give a theoretical and practical comparison about CFS and O(1). The future research of Linux schedulers could focus on documenting a comprehensive literature about Linux schedulers, from developing history, architecture and performance tuning.

2.3 Process Management

2.3.1 what is a process?

The scheduler makes it possible for the system to run multiple processes, and it decides the most deserving process to run out in the system. So, what is the process?
A process is an instance of execution in the system, which has an address space and communicates with other processes by system signals. A process includes executable code, open handles to system objects, a security context, and a unique process identifier. A program or an application could be divided into different processes when running.
Each process has its own virtual address space, and doesn't intercommunicate with others except by kernel management mechanisms such as Inter-Process Communication (IPC). Thus, if one process crashes, it will not affect other processes [8].

In Linux, all the processes running in the system are managed by a dynamically allocated task_struct structure, which is also called as a process descriptor. In the task_struct, there is information such the PID, and other attributes of the process[9, 10].

2.3.2 Program and Process

A program can be considered as a set of instructions and data that are put together, which is then executed for a specific purpose. A process can be thought of an instantiated instance of a program, or a program in action. It is a dynamic entity, constantly changing as the processor executes the machine code instructions. As well as the program's instructions and data, the process also includes the program counter and all of the CPUs register as well as the process stacks containing temporary data such as routine parameters, return addresses and saved variables. The current executing program, or process, includes all of the current activity in the CPU [11].

2.3.3 Thread and Process

The main difference between a thread and a process is that a process is an independent executable unit. A process has its own ID, virtual address space, and can communicate with other processes via the kernel. Usually, an application consists of different processes, and starts with one main process. This process, may call for more sub-processes when it gets a signal from kernel to accomplish an application.
A thread, in contrast, is not an independent unit and can be thought as a lightweight process. A process can have several threads, and those threads share the same address space and memory in the system. This allows threads to read from and write to the same data structures and variables, and also facilitates communication between threads[12]. Different processes need system signals to talk to each other, while threads in the same process can communicate with each other directly. Usually, a process starts with a thread, which is a coding construct and executable. In the thread, there is information about the stack and address space of the process. The following figure 2.2 [10] shows the relationship between a process and a thread.

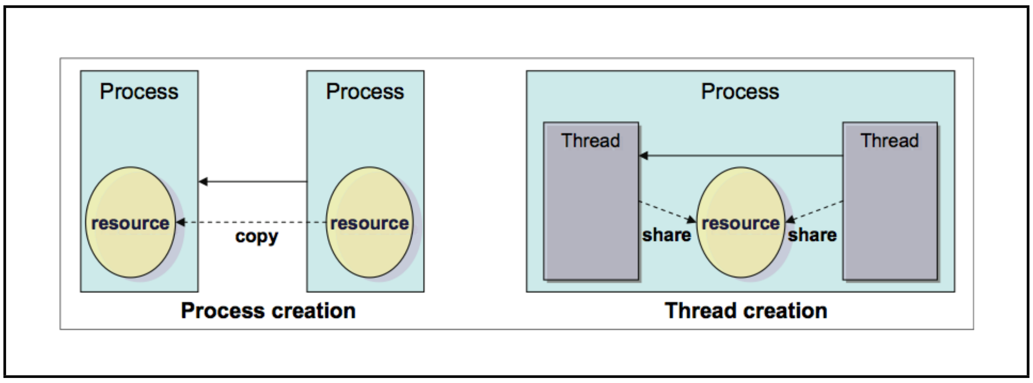


Figure 2.2: [10]

2.3.4 Different types of processes

Generally speaking, process is classified according to its performance and behavior. In this project, processes are divided into the four types:

* CPU-bound Process
  Those processes require a lot of CPU time. The total time to finish such process mostly depends on the performance of central processor, which means it takes less time to run a CPU-bound process in a higher speed CPU than that in a slower speed CPU. A CPU-bound process doesn't require much I/O, and the CPU is the bottleneck. A typical example is a task that performs mathematical calculations or crunches numbers.
* I/O-bound Process
  These processes require a lot of I/O. The total time to finish such process mostly depends on the speed of requesting data. The process spends much time waiting for input/output operations, which means it takes less time to run a I/O-bound process in a faster I/O system than that in a slower I/O system. So, an I/O-bound process doesn't require much CPU, and I/O is the bottleneck. A typical example is a task that processes data from disk.
* Interactive Porcess
  These processes refer to the situation where users have interactions with the system. Usually, users have to spend much time waiting for system response, for example to the keyboard and mouse. Those processes require a small delay in order to meet users' needs. For these processes, the scheduler has to response quickly to the requests from users. For interactive process, it is difficult to identify which particular component of the system is the bottleneck because the bottleneck shifts quickly according to the user's instructions. Typical examples are text editors and command shells.
* Mixed Process
  These processes can be defined a mix of all above processes. Since it consist of different types of processes, there is no clear bottleneck or characteristic of this kind of process. The behavior depends on what are mixed, and how much of them are mixed. In this thesis, the mixed process is a mix of I/O-bound processes and CPU-bound process.

2.3.5 Process Priority and Preemption

In Linux, each process is given a priority assigned by kernel. The schedulers chooses the next process and assign the processing time to this process based on its priority. There are two types of priorities, dynamic priority and static priority. A process is initiated with a priority when the process is created, and this priority is dynamically changed according to the scheduling algorithms. The scheduler works through the list of process, and assigns a priority to each process according to its internal algorithms, often based on process behaviors and charateristics. For example, the scheduler might increase the priority of a process when it has not been running for long time. A process with higher priority is more likely to be selected to run.

Linux allows multiple processes to be running in the system, and from the user's point of view, those processes are running at the same time. However, that is not the case in reality. Only one process can run at a specific time on any given CPU. So, processes must typically be stopped and started several times before they are completely finished. Each time it runs, the process is scheduled to run only for a very short period of time[7]. Thus, processes in the Linux system are preemptive. The Linux scheduler is responsible for choosing processes to be suspended and to be started.

2.3.6 Context Switch

A context switch can also be thought as process switch. When kernel switches from one process or thread to another, the information for the previous process and the next process is stored in the kernel. The information is called the context, which consists of the contents of the CPU registers and program counter[13].

A context switch proceeds as follows:

- The scheduler stores the state information about the current running process in the memory;
- The process is stopped, and the context of this processes stored, for example the location of where it is interrupted;
- The scheduler goes to the next process to run, restoring its context and resuming it;

The following figure 2.3 [13]demonstrate how context switches works:

When the processor switches from one process to another, it has to flush its register by removing the previous context and restoring the new context. Doing so costs a certain amount of time. Thus, scheduler is optimized to avoid context switches as much as possible.

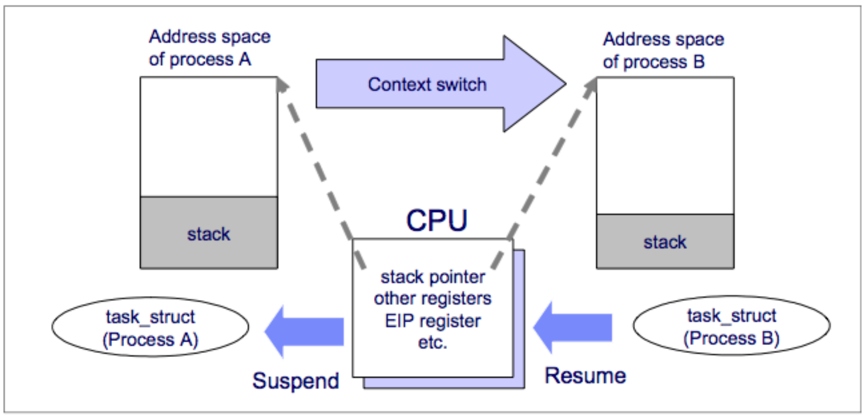


Figure 2.3: Context Switch
[13]

2.4 Process Scheduling

2.4.1 Scheduling Policy

As discussed in the previous section, the CPU can only run one process at one time, so processes have to share processor. The main job for a scheduler is to choose the most appropriate process to run next. The scheduling policy is the key part of a scheduler, since it tells it how and when to switch from one process to another, and how to select the next process to run.
Generally, the scheduling policy is based on ranking the priority of a process. The priority is given to process by scheduler based on its scheduling algorithm, which will be described in the next section 2.4.2. The priority indicates how likely a process is to be executed. A process with higher priority has more chance to be executed. The priorities assigned to processes are dynamic. The scheduler goes through the list of processes in the system and reassigns priorities based on the status of those processes.
When the scheduler finds there is one process that has greater priority than that of current running process, the current process will be interrupted and the next process will be put into processor. In another word, Linux processes are preemptive. However, the preempted process is not suspended by kernel but still waits in the queue until it is executed for the next time.

Timeslice

By default, processes are allotted a quantum of processing time. This quantum of time should be not too long or too short. If it is too long, other processes have to wait too long to be processed. If it is too short, the system spends too much time on switching processes. As Josh Aas said:

> The choice of quantum duration is always a compromise. The rule of thumb adopted by Linux is: choose duration as long as possible, while keeping good system response time.

A process can divide its timeslice into several parts rather than run all of it at one time. When its timeslice is exhausted, the process will be preempted until assigned another. When all the processes used up their timeslice, the system will do a recalculation of the timeslice. The following figure 2.4 [13] shows the recalculation of the timeslice.

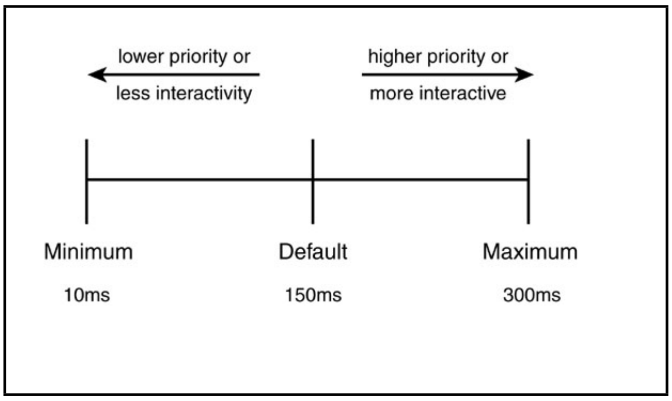


Figure 2.4: Recalculation of Timeslice
[13]

2.4.2 Scheduling Algorithm

The goal of a scheduling algorithm is to produce a "good" schedule, but the definition of "good" will vary depending on the application[14]. There are two key parts of a scheduling algorithm, the quantum of processing time and process priorities.
As discussed in the previous section, each process is assigned a timeslice. During this period of time, the process occupies the CPU. The process will be preempted when it has used up this time quantum, and the next process will occupy the CPU. A process may have been assigned a quantum several times before the process is completely finished. The value of this quantum is defined in the INIT_TASK macro, and may differ among different hardware manufacturer.
There are two types of process priorities: static and dynamic. The static priority is assigned to the process when it is created. The static priority is also called the real-time priority for real-time processes. Only real-time processes with super user privileges can get a static priority the value of which ranges from 0 to 99[15]. Those real-time processes have higher priority than the normal processes in the system, and the priorities of those processes are never changed by the scheduler.
For real-time process, there are two kind of scheduling policies, SCHED_FF

and SCHED_RR, which are based on different scheduling algorithms. SCHED_FIFO and SCHED_RR are intended for special time-critical applications that need precise control over the way in which runnable processes are selected for execution[15].
The SCHED_FIFO, as the name indicates, uses a first-in, first-out scheduling algorithm without timeslices[16]. As long as a SCHED_FIFO process is in the runnable queue, it will preempt any other non-real-time processes. Since the SCHED_FIFO process has no timeslice, scheduler will never stop it before it finishes. Basically, processes that are scheduled by SCHED_FIFO policy have higher priority than normal processes. But what happens when there is another process that is scheduled by SCHED_FIFO policy? Since the static priority ranges from 0 to 99, a SCHED_FIFO process will be preempted if there is a process with higher priority and will resume execution as soon as all processes of higher priority have completed or are blocked [15].
The SCHED_RR policy is the variation of SCHED_FIFO with timeslices. It uses a real-time round-robin scheduling algorithm. When a SCHED_RR process has exhausted its time quantum, it will be put in the end of run queue and wait for the next time to be executed, when processes with higher priorities have finished their quantum time. The length of the time quantum can be retrieved by the sched_rr_get_interval system call[16].
For real-time scheduling policies, the Linux kernel implements so-called soft real-time behavior. The real-time processes will be executed as long as they are in the runnable queue. However, the kernel doesn't guarantee that all processes will be processed within time deadlines. On the other hand, there is so-called hard real-time behavior, which guarantees to fulfill all the needs of all applications within time deadlines[16].
For normal processes (non-real-time processes), there is scheduling policy called SCHED_OTHER, which is used by default. Normal processes will be assigned with a static priority of 0. However, the static priority can be adjusted within the range from -20 to 19. The value of the static priority is also called the nice value, and it can be changed by the *nice()* system call. The nice value controls two things, the static priority and the timeslice. A greater nice value means a lower priority and shorter timeslice. On the other hand, a smaller nice value means a higher priority and longer timeslice. The reason why it is called nice value is probably that if one process is nice to others, then it voluntarily takes lower precidence in order to let other processes running in the system run first.
Running processes also have a dynamic priority. This is the sum of the base time quantum (which is therefore also called the base priority of the process) and of the number of ticks of CPU time left to the process before its quantum expires in the current epoch[17].

2.5 Current Linux Schedulers

2.5.1 O(1) Scheduler

Overview

Released in 2002, O(1) was designed to replace the previous O(N) scheduler. As the name indicates, the O(N) scheduler uses the O(N) algorithm, of which of the execution time is the function of number of process, which is N here. To be precise, the algorithm's time is the linearly function N. When N increases, the time increases linearly. As discussed before, the O(N) scheduler may end up with over head if the N continuously increase. The O(1) scheduler, runs in the constant time, as the name indicates. So, no matter how many processes running in the system, the scheduler can guarantee to finish in a fixed time. This make O(1) scale well with the number of process, which is best know feature of O(1).

Scheduling Policy

Generally speaking, the O(1) scheduler uses priority-based scheduling policy. The scheduler chooses the most appropriate process to run based on process's priority.
The O(1) scheduler is the multiple queues scheduler. The key structure of O(1) scheduler is the runqueue. There are two runqueues in the O(1) scheduler, the active runqueue and the expired runqueue. The kernel can get access to these two runqueues through pointers from the per-CPU pointer. These two runqueues can be swapped by a simple pointer swap. This will be covered later this section.
In the active runqueue, there are 140 priorities levels of processes. All the processes that have the same priority are grouped in a specific priority level. For each priority level, processes are processed in the FIFO, fist-in-first-out algorithm, which means the process that comes first is processed first. Let's take priority 1 level as an example. All the processes with priority 1 will be added to this priority level, and processed by kernel based on the FIFO. The following figure 2.5 shows how scheduler schedules processes that are in the priority 1 level:
As discussed above, the O(1) scheduler choose processes based on their prior-

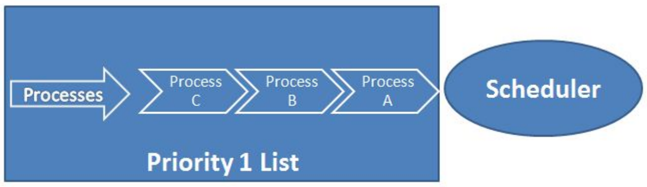


Figure 2.5: Priority 1 Level

ities. Since there are 140 different priorities levels, and for each priority level, there are processes with same priority, the scheduler starts to choose process from the highest priority level, and goes down to the second highest priority level until it go through all the levels in the system. The following figure 2.6 shows how scheduler schedules processes that in different priorities levels.
As discussed before, the O(1) scheduler doesn't go through the whole active

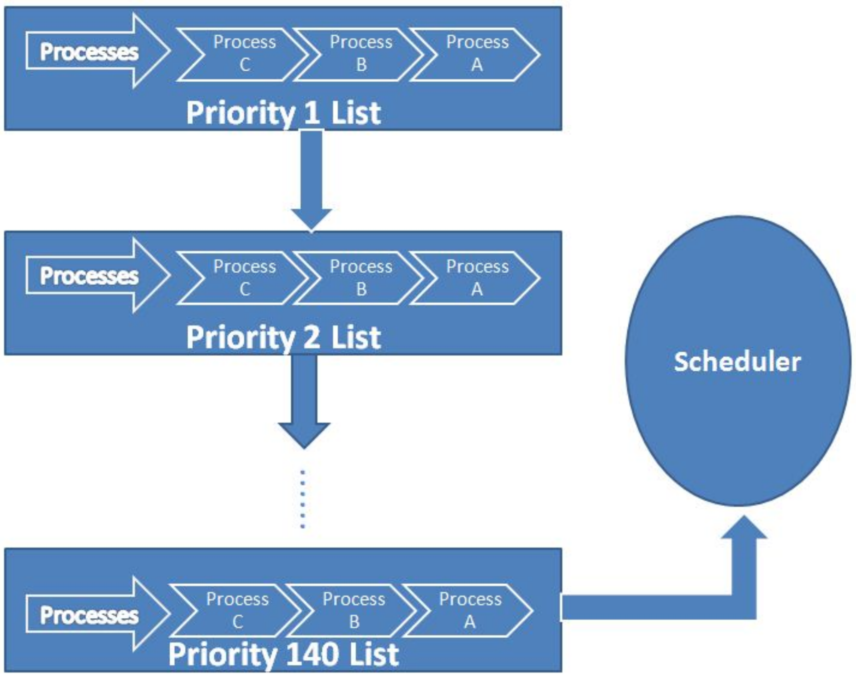


Figure 2.6: Priorities Level

runqueue level to determine the next process to run, as the figure shows, the scheduler always takes the first process from the highest priority level. There are two parts for the O(1) scheduler to choose the next process to run. First, the O(1) scheduler needs to find the highest priority level. Since the number of priority level is fixed, which is 140, so this takes a fixed time, we can call it *t1*. Second, the O(1) scheduler needs to find the first process in the level, this also takes a fixed time, we can call it *t2*. So the total time for O(1) scheduler to choose the process to run is

$$t = t1 + t2 \tag{2.1}$$

Since *t1* and *t2* are constant, the total time *t* is the constant value.
For the 140 priority levels, the first 1 to 100 priority levels are reserved for the real-time processes and the last 101 to 140 are used for user processes.
For a specific process, as discussed before, it will be assigned with priority. When this process ran out of its timeslice, the priority will be lowed, and this process will go to the end of the next lower priority level. On the other hand, after a certain timeslice, those processes which are waiting in the queue get a higher priority, and moved the next higher priority level.

The following figure 2.7 show the how processes are transferred between priority levels.
Let's suppose there are two processes in each priority level. At the time of

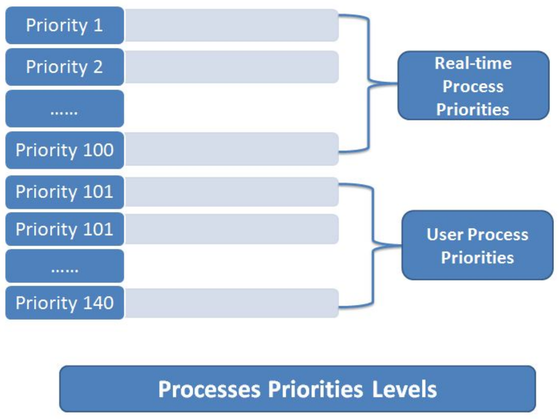


Figure 2.7: Priority

t, the process with the highest priority will start to run. As the shown in the figure, both process E and F are in the highest priority level N, and E is in the front of the level. Based on the FIFO algorithm, E will run first. A process doesn't need to use of its timeslice a time, however, the timeslice can be divided into several small parts. After a part of timeslice, E is finished, and moved the next lower priority level N-1. At the same time, the priorities of other processes will increase, and those processes will be moved to the next level. For example process A and B will be moved to the priority level 2. For process F, since it has the highest priority, it will keep the same. As for process E, since it has been processed for some time, the priority of E is lowed. And E is moved to the level $N-1$. Then F will be processed and moved to $N-1$ level.

A process may be processed for several times before it runs out of timeslice. When a process has run out of its timeslice, it will be moved to another ran queue, called expired runqueue. The timeslice for this process will be recalculated, as well as the priority. After all the processes in the active runqueue have ran out their timeslices, the active runqueue and expired runqueue will be swapped. The following figure 2.8 shows that.

Data Structure

The runqueue is the basic and essential part of O(1) scheduler's data structure. It is defined in the */usr/src/linux-2.6.x/kernel/sched.c*.

```
struct rt_prio_array {
DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /* include 1 bit for delimiter */
struct list_head queue[MAX_RT_PRIO];
};
```

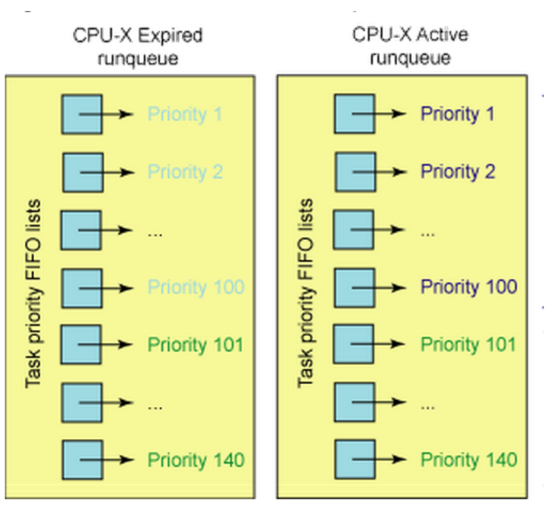


Figure 2.8: Active and Expired Runqueue
[5]

This is the priority-queue data structure of the RT scheduling class:

```
spinlock_t lock
```

This is the lock that protects the runqueue. Only one task can modify a particular runqueue at any given time[8].

```
 struct rq {
  /* runqueue lock: */
spinlock_t lock;
/* nr_running and cpu_load should be in the same cacheline because
* remote CPUs use both these fields when doing load calculation.
*/
unsigned long nr_running;
#define CPU_LOAD_IDX_MAX 5
unsigned long cpu_load[CPU_LOAD_IDX_MAX];
```

This is the main, per-CPU runqueue data structure. Locking rule: those places that want to lock multiple runqueues (such as the load balancing or the thread migration code), lock acquire operations must be ordered by ascending runqueue. The *unsigned long nr_running* defines the number of runnable tasks on the runqueue. The *unsigned long cpu_load* represents the load of CPU. The load is recalculated whenever *rebalance_tick()* is called, and is the average of the old load and the current *(nr_running * SCHED_LOAD_SCALE)*.The latter macro simply increases the resolution of the load average.

```
void schedule()
```

This is the main function for scheduling, which choose process with highest priority.

```
#define NICE_TO_PRIO(nice)      (MAX_RT_PRIO + (nice) + 20)
#define PRIO_TO_NICE(prio)      ((prio) - MAX_RT_PRIO - 20)
#define TASK_NICE(p)            PRIO_TO_NICE((p)->static_prio)
```

Convert user-nice values [-20 ... 0 ... 19] to static priority [MAX_RT_PRIO..MAX_PRIO-1], and back.

```
#define USER_PRIO(p)            ((p)-MAX_RT_PRIO)
#define TASK_USER_PRIO(p)       USER_PRIO((p)->static_prio)
#define MAX_USER_PRIO           (USER_PRIO(MAX_PRIO))
```

'User priority' is the nice value converted to something we can work with better when scaling various scheduler parameters, it's a [0 ... 39] range.

2.5.2 Completely Fair Scheduler

As a scheduler, the most important feature for CFS is, as its name indicates, fairness to processes, which means when processes are scheduled by CFS, they should be given fair share of processing time. Like O(1), processes in CFS are given processing time, whenever a process's processing time is not as fair as to other processes, this process is switched, and another process is going to have get processing time. Generally speaking, this is how CFS maintains fairness.

Red-black Tree

Unlike O(1) which uses priority arrays based algorithms, CFS uses time-based queues. In O(1) each process is given a timeslice, the same with CFS, but the processing time given to a process is called virtual runtime here. In O(1), the higher priority of a process, the sooner it is going to be processed. In CFS, the smaller the virtual runtime is, the sooner it is going to be processed. In order to switch processes, the O(1) basically maintain two priority queues, one active, and expired, however, CFS uses so called red-black tree to manage processes. The next figure 2.9[5] shows the structure of red-black tree. Generally speaking, a red-black tree is self-balancing binary search tree. First, a binary tree is a kind of data structure. As the next figure shows, every node in the binary tree can have no child node or have two, one left child node and one right child node. A node with child node is called parent node, and a node without child node is called leaf. There is one root node at the top of the tree.
Second, a binary search is a binary tree, which for each node n has following features:

- The left sub-tree has nodes with keys smaller than the key of n;

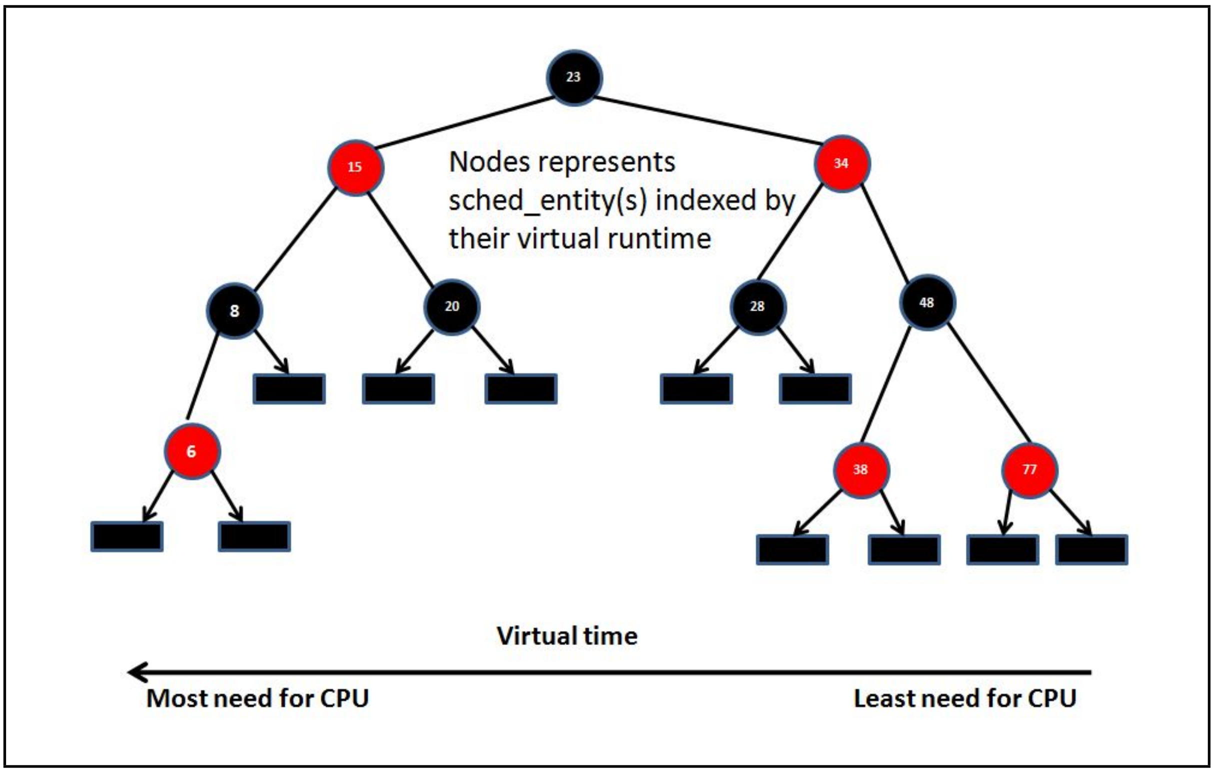

23
15
34
Nodes represents sched_entity(s) indexed by their virtual runtime
8
20
28
48
6
38
77
Virtual time
Most need for CPU
Least need for CPU

Figure 2.9: Red-black Tree
[5]

- The right sub-tree has nodes with keys greater than the key of n;

To search for a specific element in the tree, first, compare the element with the root node. If the element is greater than the node key, then compare the element with the right-child node. If the element is less than the node key, then compare with the left-child node. So the most comparison it will take is the depth of this tree, which is *log n*, where n is the total number of node.
A balancing tree is a binary tree, which has following features:

- The difference between two depths of the two sub trees of every node is no greater than 1;
- The two subtrees are also balancing trees

So, based on the definition of the balancing tree, the depth of a tree with n nodes can be:

$$Depth = m, where 2^m <= n < 2^(m + 1) \tag{2.2}$$

As long the n is given, the depth of balancing tree is defined.

A red-black tree is a binary search tree which has following features [18]:

- Every node is either red or black
- Every leaf is black
- If a node is red, then both its children are black
- Every simple path from a node to a descendant leaf contains the same number of black nodes

The red-black tree has many useful properties. First, to search in the red-black tree will take O(log n) time. Second, it's self-balancing, which means that no path in the tree will ever be more than twice as long as any other[4]. So, back to the red-black tree figure 2.9, as the figure shows, every node represents a task in the system, and the key value of the node represent the virtual runtime of this specific task. As the definition of red-black tree means, the left-most node has smallest key value, which means this task has smallest virtual runtime, so this task is most needed to be processed. On the other hand, the right-most node has greatest key value, which means this task in least needed to be executed. So, every time the CFS only needs to pick the left-most task to the processor. As long as the left-most task is processed, it is deleted from the tree. Since this task has already got some processing time, its virtual runtime is increased. Now, if this task is not finished, it will be inserted back to the red-black tree with the new virtual runtime. And the time for this operation, as discussed before, is O(long n).
The virtual runtime can be considered as a weighted timeslice, which will be

described in the following. The virtual runtime is defined as the below equation:

$$virtualruntime+ = \frac{(delta\_exec)(NICE\_0\_LOAD)}{se(load.weight)} \tag{2.3}$$

In this equation, delta_exec is the amount of execution time of task, NICE_0_LOAD is the unity value of the weight [19].

This virtual runtime can be considered as the unfairness of a process, so the left-most process has the most unfairness. Every time, when a process is finished in the CPU, all the other leaving processes become unfair because they have waited for a period of time. So, their unfairness must be increased as well. In O(1), when a process has waited, it is moved one step forward in the priority list or it is moved to a higher priority list. But in CFS, the position of the process in the red-black tree doesn't change, but the virtual runtime increases. But, how CFS assigns the base virtual runtime to a process when it is created? A newly created process will be assigned with a minimum current virtual runtime. This minimum virtual runtime is maintained and used to avoid overflow of the virtual runtime value so that virtual runtime and min_virtual runtime can be reliably used to sort the red-black tree[20].

After the left-most node is deleted from the tree, the parent node of this node becomes the new left-most node, or becomes the next task to be processed. Every task on the left of the red-black tree is given the processing time, and after that, tasks on on the right of the tree moves to the left. So, in this way, scheduler schedules every runnable task in the system as fairly as possible[4].

In CFS, there is no exact timeslice like O(1), but task does receive CPU share. A weight value is given to a task, and this weight is defined in the structure called *sched_entity*, which will be described later. The CPU share then is defined by following equation:

```
share = se {load.weight} / cfs_rq {load.weight}
```

Here, in this equation, se load.weight is the weight of the entity, structure *cfs_rq* load.weight is the sum of all entities' weights. The structure *cfs_rq* will also be described later. The entity here is not only a task or process; it can also be a group, a user (group scheduling will be discussed later). As long as the CPU share is defined, the time slice is easy to define, as the following equation indicates:

```
timeslice = share * period
```

The period, is the total timeslice that scheduler uses for all the tasks. The minimum of the period is 20ms. As the equation indicates, since the total number of the processes in the system is a variable, the total time slice is also a variable, and the period is a variable. So the timeslice in CFS for a process is not constant, but a changing variable. It is quite different from O(1), where the timeslice is a constant[19].

Data Structure

Unlike O(1) scheduler, there is no *struct prio_array* in CFS, instead, the scheduling entity and scheduling classes is used. They are difined by *struct sched_entity* and *stuct_class*, and both of these two structures are contained in the *task_struct*.

```
struct task_struct {
# Defined in 2.6.23:/usr/include/linux/sched.h */
struct prio_array *array;
struct sched_entity se;
struct sched_class *sched_class;
 ...
}
```

The *task_struct* structure describes all the task information such as state, stack, address, flags and priority. All the tasks in the system are stored in this structure; however, some of the tasks in the system are not runnable. Or in other words, they have nothing to do with scheduler. So *sched_entity* structure is needed to store the information of all runnable tasks. In this structure, there is scheduler-related information, as the following table shows:

```
struct sched_entity {
# Defined in 2.6.23:/usr/include/linux/sched.h
 long wait_runtime;  # Amount of time the entity must run to become completely fair and balanced.
 s64 fair_key;
 struct load_weight   load;  #  for load-balancing
 struct rb_node run_node;  # To be part of Red-black tree data structure
 unsigned int on_rq;
 ....
}
```

One of the most important variables in this structure is the wait_runtime. This variable contains the time that this task has used, and also is the index in the red-black tree.
The next structure is *sched_class*, as shown in the table below.

```
struct sched_class {
      # Defined in 2.6.23:/usr/include/linux/sched.h */
      struct sched_class *next;
      void (*enqueue_task) (struct rq *rq, struct task_struct *p, int wakeup);
      void (*dequeue_task) (struct rq *rq, struct task_struct *p, int sleep);
      void (*yield_task) (struct rq *rq, struct task_struct *p);

      void (*check_preempt_curr) (struct rq *rq, struct task_struct *p);

      struct task_struct * (*pick_next_task) (struct rq *rq);
```

```
        void (*put_prev_task) (struct rq *rq, struct task_struct *p);

        unsigned long (*load_balance) (struct rq *this_rq, int this_cpu,
                struct rq *busiest,
                unsigned long max_nr_move, unsigned long max_load_move,
                struct sched_domain *sd, enum cpu_idle_type idle,
                int *all_pinned, int *this_best_prio);

        void (*set_curr_task) (struct rq *rq);
        void (*task_tick) (struct rq *rq, struct task_struct *p);
        void (*task_new) (struct rq *rq, struct task_struct *p);
};
```

This structure describes how a process is going to be scheduled, which basically consists of three parts, to add a task to the scheduler, to make a task preempted, to pick up next task to run. Those actions are defined by different functions, take *enqueue_task* as an example, when a task becomes runnable, this function is called. A process is then added to the red-black tree with its virtual runtime. The last function as shown above is *task_new*, which is used for group scheduling[21], which will be covered in the next part.
The next figure2.10 shows how CFS makes decisions. CFS usually starts with the top scheduler class, then tries to find available task to run. If there are available runnable tasks, the function *pick_next_task* is called, and task is scheduled. If no available task, CFS will puck the next scheduler class to find other available tasks.

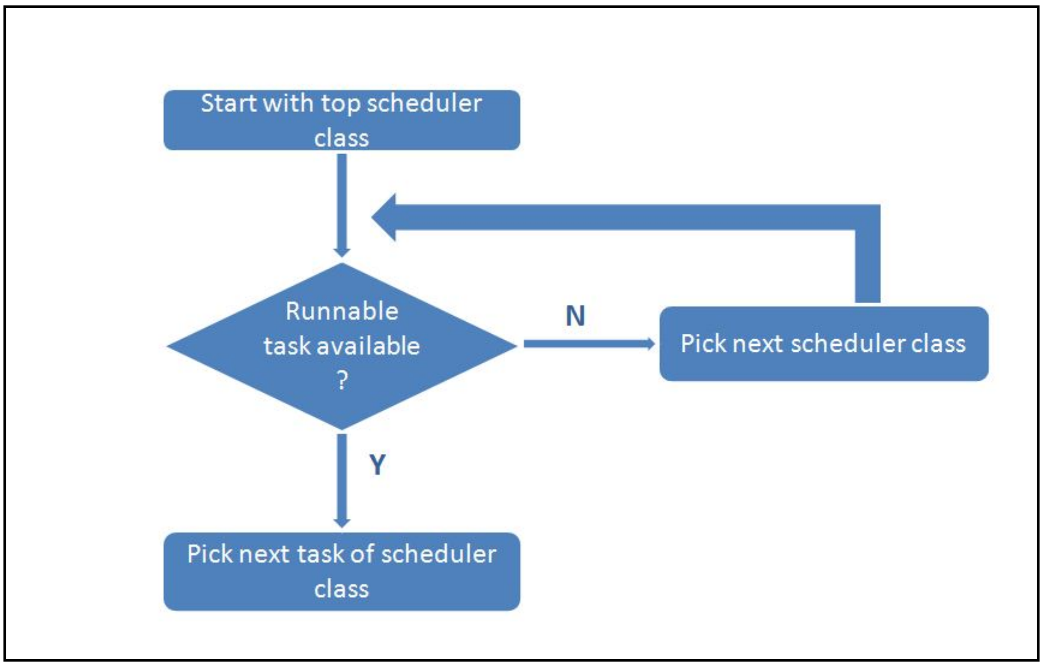

Start with top scheduler class
Runnable task available ?
N
Pick next scheduler class
Y
Pick next task of scheduler class

Figure 2.10: CFS Scheduler Decision

The next important data structure is the *cfs_rq* structure.

```
struct cfs_rq {/* Defined in 2.6.23:kernel/sched.c */
    struct load_weight load;
    unsigned long nr_running;
    s64 fair_clock; /* runqueue wide global clock */
    u64 exec_clock;
    s64 wait_runtime;
    u64 sleeper_bonus;
    unsigned long wait_runtime_overruns, wait_runtime_underruns;
    struct rb_root tasks_timeline; /* Points to the root of the rb-tree*/
    struct rb_node *rb_leftmost; /* Points to most eligible task to give the CPU */
    struct rb_node *rb_load_balance_curr;
#ifdef CONFIG_FAIR_GROUP_SCHED
    struct sched_entity *curr; /* Currently running entity */
    struct rq *rq;      /* cpu runqueue to which this cfs_rq is attached */
    ...
    ...
#endif
}
```

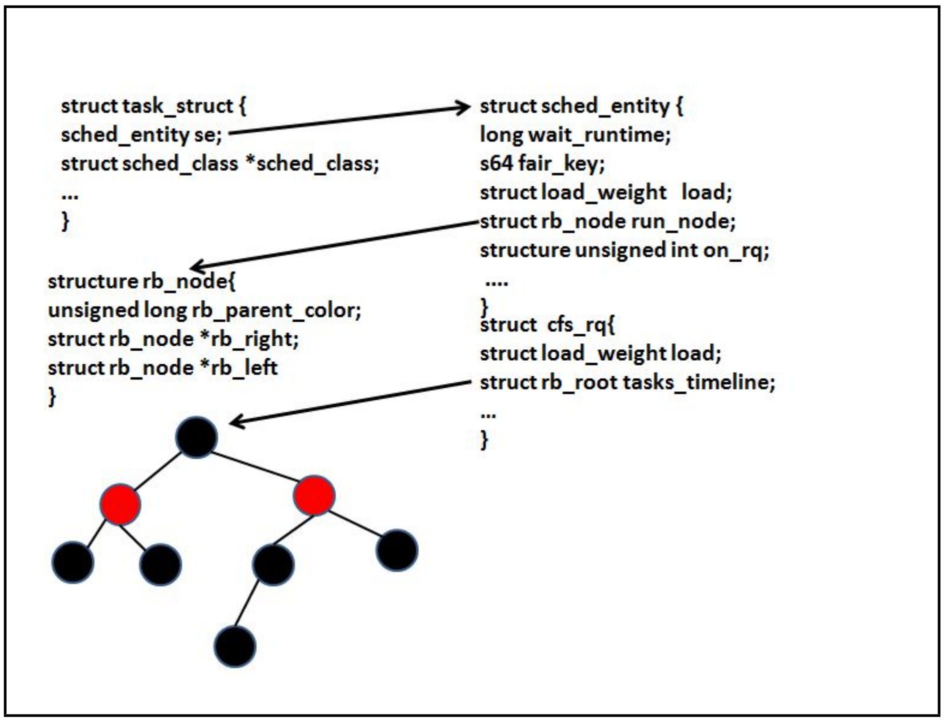
As the name indicates, this structure contains the information of red-black tree. Holding information about the associated red-black tree. For example, the *rb_root* structure defines the root information in the red-black tree.
The next data structure is called *rb_node*, which contains information of every node in the red-black tree, for example the color of a node's parents and the child references. As described before, the structure *sched_entity* containes *rb_node* structure information such as reference, load weight and other statistic data *multipl_processes_cfs*.

```
structure rb_node{
unsigned long rb_parent_color;
struct rb_node *rb_right;
struct rb_node *rb_left
};
```

So far, key data structures are covered with a general description. The relationships among those structures can be shown in the next figure 2.11:

Scheduling Policy

In order to maintain the fairness for process, CFS uses appeasement policy. When a process becomes runnable and is added to the runqueue, the current time is recorded. When a process is waiting for the CPU, the wait_runtime of this process is also increasing. How much is this wait_runtime increased depends on two factors: the total number of current processes and the priorities of those processes. When the processes is processed by CPU, the wait_runtime of this process begins to decrease until the next process becomes the left-most in the red-black tree. Then this current process is preempted by a function called *schedule()* function. The preempted time is not a static value, but a changing variable. As long as this process is not finished, it will be sent back



```
struct task_struct {
sched_entity se;
struct sched_class *sched_class;
...
}

struct sched_entity {
long wait_runtime;
s64 fair_key;
struct load_weight   load;
struct rb_node run_node;
structure unsigned int on_rq;
....
}

structure rb_node{
unsigned long rb_parent_color;
struct rb_node *rb_right;
struct rb_node *rb_left
}

struct  cfs_rq{
struct load_weight load;
struct rb_root tasks_timeline;
...
}
```

Figure 2.11: CFS Data Structure Relationships

to the red-black tree by the function called *put_prev_task*, which is defined in the scheduling class. After this process is preempted, the next process will be picked up through a function called *pick_next_task* function. The function then return the sched_entity references to the scheduler. At last, the process is processed by CPU[21].

Priority in CFS

In O(1), priority is used to determine how appropriate a process is going to be processed. However, though CFS still maintains the same 140 priority levels that are found in O(1) scheduler, in CFS, as discussed in above section, priority is not the key factor that affect a process's states. In CFS, the higher priority a process has, the longer processor can execute it. In other words, in CFS, a process with a lower priority, the time this process is permitted to execute dissipates more quickly[5].

Group Scheduling

The concept of CFS is to maintain fairness to all the individual processes in the system. However, a system may have multiple users. Suppose there are

multiple users in the system running processes, for example user A and user B, user A runs only one process called p, and user B has 49 processes from p1 to p49. Then user A will get only 2% of CPU. CFS tries to make processes fair to each other, but not users. So, in order to make users are fair to each other, CFS has a feature called group scheduling, which means processes belonging to a users are wrapped as a group, and those groups are share fair CPU. In the example, the process p belonging to user A will get 50% CPU, and process from p1 to p49 as a whole belonging to user B will get same 50% CPU.
This feature is especially useful when CFS is applied in the server environment. Consider a server with many different users, each user may have different number of processes running in the system, instead of maintaining fairness to process, it is much better to maintain fairness to users, so that users don't have to wait long time for responses. The group-scheduling feature, however, is not by default set; users need to tune the parameter by themselves. Besides group scheduling CFS also has other tunable features, for example modular scheduler framework.

Chapter 3

Methodology

As addressed in the problem statement part, this thesis is going to compare O(1) scheduler and CFS. The comparison of O(1) and CFS will be in the terms of :

- Configurability
- Performance

The configurability of both schedulers will cover the theoretical design with respect to O(1) and CFS.
The comparison of performance will be under a serial of tests. Those tests, as discussed in the chapter 1 will be classified regarding to their purpose and behavior. Different performance data will be collected later, and those data will be analyzed in different ways.

3.1 Theoretical Design Comparison

The theoretical comparison toward O(1) and CFS will be in the term of following aspects:

- data structure
- scheduling policy

Data Structure Comparison

The main data structure of O(1) scheduler is the runqueue which consists of different priority lists. As described in the background section, processes are grouped in two queues, active queue and expired queue. Both queues uses

First-In-First-Out algorithm for each priority list in the queue. After a process finishes its timeslice, it is moved to the expired queue. When all the process in the active queue are finished, the active queue and expired queue are swapped. Since the scheduler always picks the first process in the highest priority list, it takes O(1) time, which is a constant to schedule a process. The following figure shows the data structure of O(1).

CFS uses quite data structure as compared with O(1). As described in the

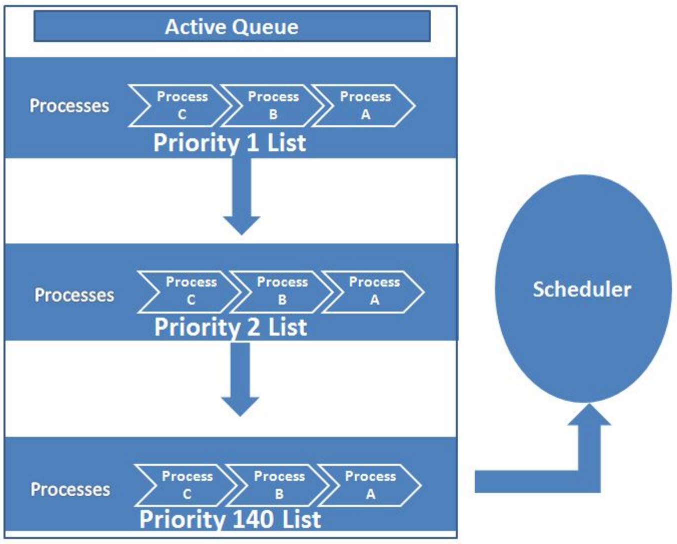


Figure 3.1: O(1) Data Structure

above section, the main data structure in CFS is the red-black-tree, which consists the virtual time of different processes. All the processes or tasks with their unfairness which is the virtual time are in the red-black tree, and CFS always picks the left-most task, sends it to the processor. It takes O(log n) to schedule a process, where n is the total number of the process.

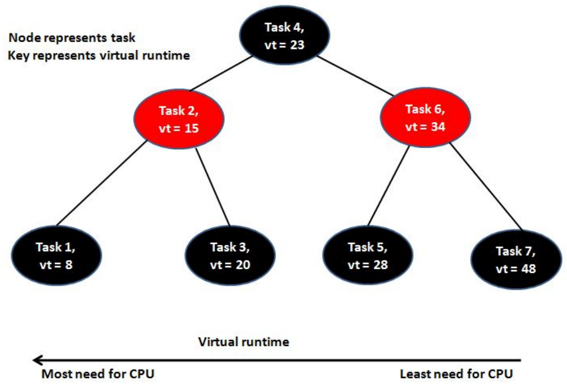


Figure 3.2: CFS Data Structure

Scheduling Policy Comparison

In summary, the scheduling policy for O(1) consists of the following parts:

- 140 priority list, from 1 to 100 for real time processes, from 101 to 140 for normal processes;
- SCHED FIFO and SCHED RR for real time processes, SCHED OTHER and SCHED BATCH for normal processes;
- Each process is assigned with timeslice and a nice value, and the PRIO equals the sum of MAX_RT_PRIO, NICE and 20;
- Processes in the the same priority list are round-robined;
- Interactive processes get extra bonus.

For CFS, the scheduling policy consists of the following parts:

- SCHED NORMAL for normal processes;
- SCHED BATCH for batch processes;

There are four groups in O(1), *SCHED FIFO*, *SCHED RR*, *SCHED OTHER* and *SCHED BATCH*. The first two groups handle the real-time processes, the rest two groups handle other remaining processes. CFS uses approximately the same policies, for example *SCHED FIFO*, *SCHED RR* for the real time processes. Besides that, CFS also has *SCHED NORMAL* and *SCHED BATCH* groups which ensure fairness, and *SCHED IDLE* for idle group.
In general, the goal for O(1) is to achieve fairness and interactive performance,

and the goal for CFS is to achieve complete fairness but still having good interactive performance.
The next part will then cover the performance comparison testing schema to verify and evaluate the theoretical difference presented in the background chapter.

3.2 Performance Comparison Design

3.2.1 Variables in the Test

The comparison of performance will be under a serial of experiments. The first concept in the experiment is the variable. A variable is a factor or a condition that exists in the whole experiment, and has effect on the result of experiment. Generally speaking, there are three kinds of variables in the experiment: independent variable. dependent variable and controlled variable.

The independent variable is the factor that changes with experiment, and dependent variable is the factor that responds to the change of independent variable. The control variable, however, is that factor that is constant during the whole experiment. To make the test as fair and accurate as possible, there must be only one independent variable, so that people can observe the correspondence between dependent variable and independent variable, and also people can make sure the comparison of two different dependent variables is fairly made[22].
In this project, obviously the schedulers themselves are the dependent variables, while processes are independent variables, and other factors like system configurations, network, I/O, CPU are the controlled variables, and should be the same for both schedulers.
All tests are performed using Intel(R) Core(TM)2 CPU with 8 GB of memory. In order to make controlled variables the same for both schedulers, Redhat 6.2 with CFS and Redhat 5.7 with O(1) are installed in two separate disks, while they share the same CPU, memory and other configurations. All tasks are put in one core, so that only one run queue is used.

3.2.2 Performance Interpretation

As discussed in the above section, both schedulers will have performances according to different processes, and since all the other controlled factors are the same, so we can make the comparison of performance as fair as possible. But what is the performance of a scheduler? What are key factors of a scheduler?

- Fairness: The fairness for processes here means all these processes which are the same to each other should have the same execution time or take the same amount of time to run when all of them are running at the same time in the system. However, that is not the case in the real world obviously. In this thesis, the execution time of each process will be grouped in a sample, and from the statistic point of view, the standard deviation of the sample could be a term of how fair each process is to each other.
- Efficiency: One way of showing how busy CPU is will be to record CPU usage. Most CPU monitoring tools reveal the CPU usage by the following terms:
  - real time, is wall clock time - time from start to finish of the call. This is all elapsed time including time slices used by other processes and time the process spends blocked (for example if it is waiting for I/O to complete).
  - user time, is the amount of CPU time spent in user-mode code (outside the kernel) within the process. This is only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.

  - system time, is the amount of CPU time spent in the kernel within the process. This means executing CPU time spent in system calls within the kernel, as opposed to library code, which is still running in user-space. Like 'user', this is only CPU time used by the process. See below for a brief description of kernel mode (also known as 'supervisor' mode) and the system call mechanism.

- Stability:
  For a specific job, the time that it takes to run should be approximately the same when repeated, which makes the system predictable, and means the stability of the scheduler.

- Predictability:
  For the CPU-bound process here, the total execution time for N simultaneous processes is compared with N single sequential process. In idea world, the total time of N simultaneous process should be exactly the same with N single sequential execution time. In the real world, they couldn't be exactly the same. The difference between these two values is compared between schedulers.

- Turnaround Time:
  The total time to finish a process is very important for a user. Total execution time of N processes is compared between two schedulers.

- Context Switch:
  That how many times a process is switched is compared between two schedulers.

- CPU Usage:
  Percentage of the CPU that a process gets, computed as (%U + %S) / %E.

3.2.3 CPU-bound Processes Testing

Based on the testing goals, the testing schema consists two parts, processes with same priority and processes with different priorities. In both parts, a single CPU-bound process used as benchmark is a pi calculation, which is done by a Perl script.

In the first part, all the CPU-bound processes have the same priority, which is set as default. In order to compare the above performances of two schedulers, several sets of simultaneous processes run in the system. In different set, the number of processes increases from 1, 100, 200, 400, 600, 800 and 1000. Each set of processes are executed simultaneously and repeated for 10 times. Besides, another group of tests are executed without simultaneous processes but with sequential processes. The performance of two schedulers are tested by the following ways:

- In order to measure fairness and turnaround time, the values of elapsed time of those simultaneous processes are recorded. For example, for 100 simultaneous processes, the average values of these100 elapsed time of CFS and O(1) are recorded. This average value is considered as the turnaround time. Both average values are compared to show differences of CFS and O(1). Also, as discussed in the performance interpretations section, the standard deviation of these 100 elapsed time of both schedulers are compared to show the differences of them.
- In order to measure stability, the values of elapsed time of sequential processes are recorded. For example, for 100 sequential processes, the standard deviation of these 100 elapsed time of two schedulers are compared to show differences.
- In order to measure predictability, elapsed time of N simultaneous processes and N sequential processes are compared to show differences of two schedulers.
- In order to measure CPU usage, efficiency, context switch, the average of those values are compared between two schedulers.

The following graph 3.3 demonstrates the comparison method in general. In

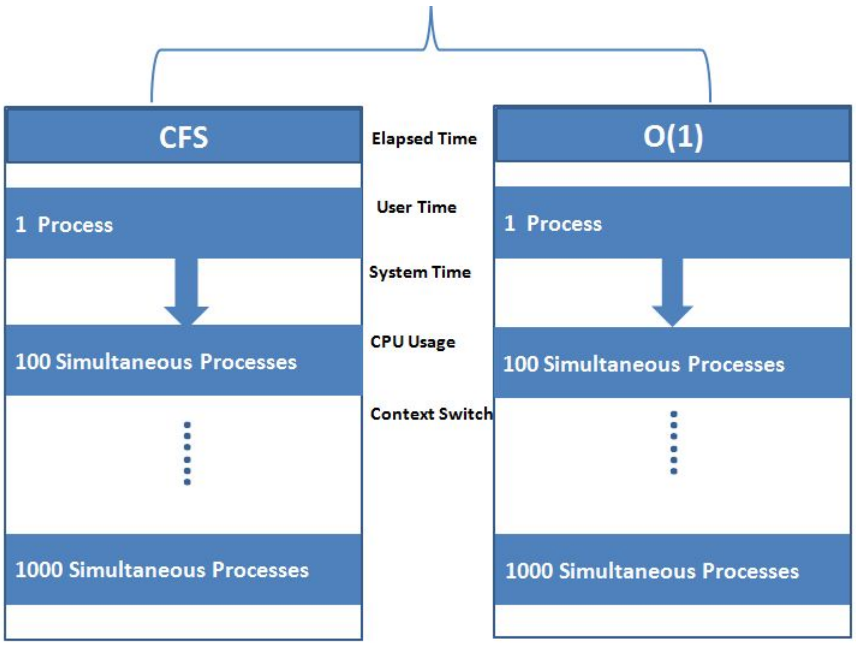

CFS
1 Process
100 Simultaneous Processes
1000 Simultaneous Processes
Elapsed Time
User Time
System Time
CPU Usage
Context Switch
O(1)
1 Process
100 Simultaneous Processes
1000 Simultaneous Processes

Figure 3.3: CPU-bound Process Comparison

the second part, all the CPU-bound processes are divided into two groups with different priorities. The first groups of processes have default priority, and the second group of processes' priority is modified by a nice value of 10(The background section shows how to modify the nice value of a process). Those two

group of processes are called normal group and nice group. Obviously, since the nice group is nice to normal group, the normal group has higher priority.
The testing method is the same with first part. For example, for the scenario of 100 simultaneous processes, they are divided into two groups, 50 nice processes and 50 normal processes. The elapsed time and other information are collected respectively.
The Linux *time* utility is used to collect above information, as the following command shows:

```
/usr/bin/time --format '%e %U %S %P %c %w' ./pi.pl
```

For single process, it lasts for about 4 second, which is long enough for a process to be scheduled many times by scheduler. As discussed in the background part, the timeslice for O(1) and CFS are in the millisecond level.

3.2.4 I/O-bound Processes Testing

I/O -bound process requires much I/O, but not CPU. A disk benchmark tool bonnie++ is used to create I/O-bound process. A bonnie++ process can perform a number of simple tests of hard drive and file system. To measure the scheduler performance, a number of I/O-bound processes run in the system. The number increases from 1 to 20. The same with CPU-bound processes testing, Linux *time* utility is used to collect scheduler related information, as the following command shows:

```
/usr/bin/time --format '%e %U %S %P %c %w' bonnie++ -d /root -s 1024 -m test -r 512 -x 1 -f -u root
```

Besides *time* utility, another system monitor tool *vmstat* is used to collect scheduler related information for example idle of CPU. The following command shows the result from *vmstat*.

```
procs -----------memory---------- ---swap-------io---- --system-- -----cpu----
 r  b  swpd  free   buff   cache   si  so bi bo   in   cs  us   sy  id  wa  st
 0  0  148  7290812 181728  311008  0  0  0  8 1035  158 0 0 100  0  0
```

The output displays a serial of system information, however, not all of them are schedulers related. The last eight columns are of interest for the scheduler testing.
The results from above two tools are compared between CFS and O(1) with a number of I/O-bound processes.

3.2.5 Mixed Processes of CPU-bound and I/O-bound

After measuring pure CPU-bound processes and pure I/O-bound processes, the next interesting thing is to measure a mixed situation, where both kinds of processes run in the system.
50 CPU-bound processes and 10 I/O-bound processes are mixed together, and start at the same time in both schedulers. The elapsed time and other scheduler related information of both kinds of processes are collected by the same tools *time* and *vmstat*.
There are eight dimensions of comparisons in this part:

- comparisons of scheduler related results of CPU-bound processes between CFS and O(1);
- comparisons of scheduler related results of I/O-bound processes between CFS and O(1);
- comparisons of scheduler related results between pure CPU-bound processes and mixed CPU-bound process of CFS;
- comparisons of scheduler related results between pure CPU-bound processes and mixed CPU-bound process of O(1);
- comparisons of scheduler related results between pure I/O-bound processes and mixed I/O-bound process of CFS;
- comparisons of scheduler related results between pure I/O-bound processes and mixed I/O-bound process of O(1);
- comparisons of difference of pure CPU-bound processes and mixed CPU-bound between CFS and O(1);
- comparisons of difference of pure I/O-bound processes and mixed I/O-bound between CFS and O(1);

3.2.6 Interactive Processes Testing

Before measuring how these two schedulers scheduling interactive processes, choosing what's kind of interactive process is first step. As discussed in the background part, a process that its inout and output are interrelated, or a process conducted by user through an interface can be seen as an interactive process. Based on this definition, there are a variety of different kinds of interactive processes. In this project, there are two kinds of simulations.
For editor simulation, a Perl script using Expect package is used to simulate the situation that a number of users editing a file in the system. The script reads data from file, processes them, and prints on the screen. This simulating the situation where users open a file and edit it. A number of this processes run in the system. The number increases from 100 to 1000. And *time* is utility is used to record scheduler related results, which are then compared between two schedulers.

The second interactive process is running a web browser. The web browser in this project is a Firefox process. The process is not started by user clicking the button, but by a Perl script running a Firefox command, as shown below:

```
firefox -chrome www.google.com
firefox -chrome new www.vg.no
```

In total 20 Firefox process run in the system. At the same time, *vmstat* is running in the background to record scheduler related information. The results from *vmstat* are then compared between two schedulers.

Chapter 4

Result

This chapter covers the results from 4 types of testing sections, CPU-bound processes testing, I/O-bound processes testing, Mixed Processes testing and Interactive Processes testing. In each part of them, there are results either from *time* utility or *vmstat* . Each testing section contains a number for testing scenarios, which means there are lots of result data. Only a brief view of results which are typically related with scheduler, is shown in the below.
Before showing results, an explanation of output from *time* and *vmstat* is needed here. As discussed in the approach chapter, the Linux *time* utility is used to get scheduler related data. We get the time information of each individual process with respect to :

- e: Elapsed real time, in seconds.
- U: Total CPU seconds used directly by the process.
- S: Total CPU seconds used by the system on behalf of the process.
- P: Percentage of the CPU that this job got, computed as (
- c: Number of times the process was context-switched involuntarily (because the time slice expired).
- w: Number of waits: times that the program was context-switched voluntarily, for instance while waiting for an I/O operation to complete.

Unlike *time* , which displays information of a single process, *vmstat* , however, displays the information of the whole system. The following output from *vmstat* is presented in this chapter:

- us: Time spent running non-kernel code. (user time, including nice time)
- sy: Time spent running kernel code. (system time)
- id: Time spent idle. Prior to Linux 2.5.41, this includes IO-wait time.

- wa: Time spent waiting for IO. Prior to Linux 2.5.41, shown as zero.
- in: The number of interrupts per second, including the clock.
- cs: The number of context switches per second.

4.1 CPU-bound Process

4.1.1 Single CPU-bound Process

Before starting running multiple processes in the system, an individual process is repeated for 400 times. Then 400 occurrences of elapsed time, user time, system time, CPU percentage, involuntary context switch and voluntary context switch are presented here to give a bottom line of comparison.

Elapsed Time

The following figure 4.1 shows differences of elapsed time of a single CPU-bound process toward O(1) and CFS.

As the graph shows, the elapsed time of CFS is slightly greater than that

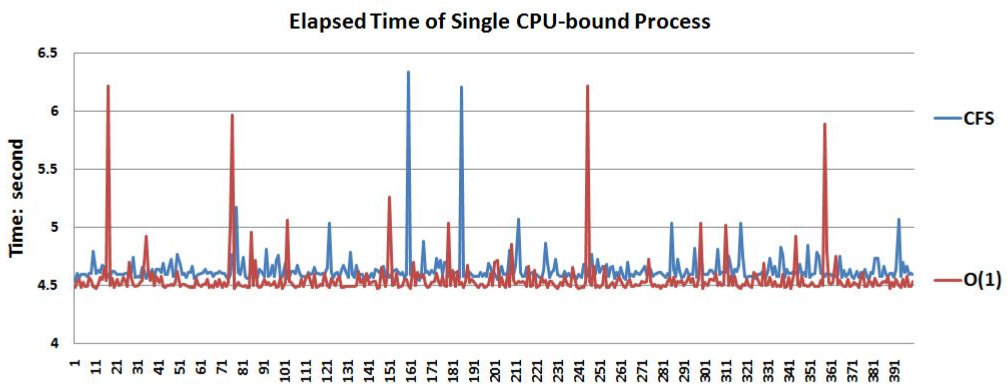


Figure 4.1: Elapsed Time of Single CPU-bound Process

of O(1). To be more precise, the average of elapsed time of CFS is 4.6311, 1.72% greater than that of O(1), which is 4.5529. The difference between this two groups of data is moderate and small. A statistical test should indicate whether these differences could have been produced by chance. In this chapter, t.test is used. The following table is the result from t.test of above two group data.

```
Welch Two Sample t-test
data:  cfs_elapsed_400_seq and o1_elapsed_400_seq
t = 6.8613, df = 755.846, p-value = 1.422e-11
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.05582581 0.10057419
sample estimates:
mean of x mean of y
 4.631175  4.552975
```

The null hypothesis here is that the elapsed time is the same in both groups. As the result shows, the p-value is 1.422e-11, less than 0.05. Then the null hypothesis is rejected which means that the true difference between not equal to 0. The t.test indicates that from the statistical point of view, there is indeed difference between CFS and O(1), and obviously, CFS costs more elapsed time than O(1).
The other thing that can be seen from the figure is that the curve of O(1) has more outliers, which indicates that there are more processes that are not well scheduled. To be more precise, the standard deviation of O(1) is 0.1702, greater than that of CFS which is 0.1408.

User Time

The following figure 4.2 shows differences of user time toward O(1) and CFS.

The figure demonstrates that, the user time of CFS is greater than that of

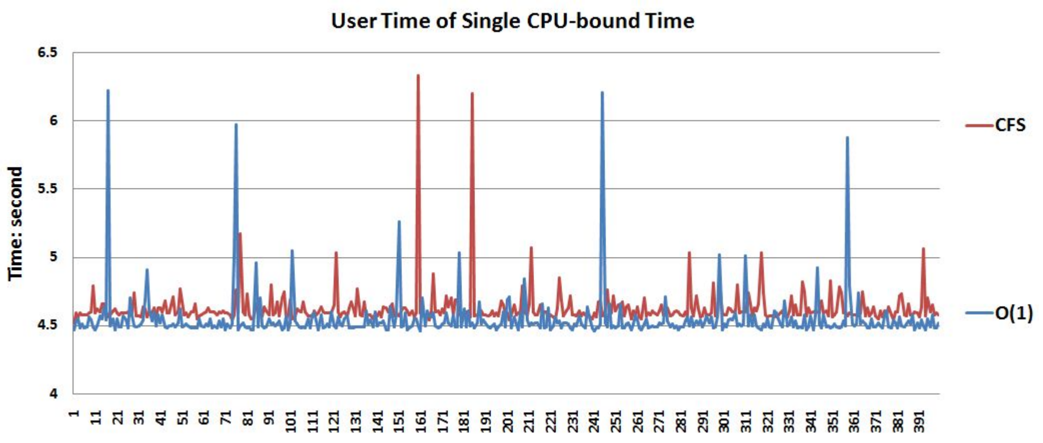


Figure 4.2: User Time of Single CPU-bound Process

O(1). To be more precise, the average of user time of CFS is 4.6252, 1.68% greater than that of O(1), which is 4.5484. Just like that of elapsed time, there is slight difference between two schedulers. Also, the O(1) has more outliers, which indicates that O(1) scheduler is not as stable as CFS. When tests are repeated, both CFS and O(1) show similar user time as this figure demonstrates.So, in the following section, the user time of both schedulers are not presented anymore.

System Time

The following figure 4.3 shows difference of system time toward O(1) and CFS.

As the figure shows, the system time of CFS is a static value, which is 0.

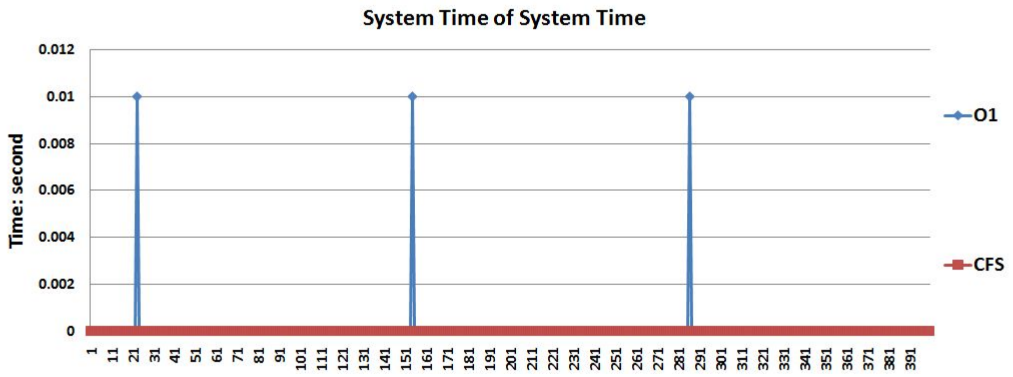


Figure 4.3: System Time of Single CPU-bound Process

That means that the processes spend no CPU time in the kernel model. However, for O(1) scheduler, there is a very small number of processes that cost system time. To prove that this is not caused by chance, the test are repeated, and the result is almost the same. So CPU-bound process does cost system time when it is scheduled by O(1), and cost no system time when it is scheduled by CFS.

As known to all, the user time is the time of process that is spent in user model, which doesn't interfere other processes. The system time is the time of process that is spent in the kernel model, which should be avoided. The same is for the scenarios when there is a number of a process running in the system. The system time of CFS is always 0, while for O(1), the outliers occur. So, in the following section, the system time of both schedulers are not presented anymore.

Involuntary Context Switch

The following figure 4.4 shows the involuntary context switch. As described in the background section, the involuntary context switch is the number of times the process was context-switched involuntarily (because the time slice expired).

As the figure shows, the curve of CFS and O(1) are overlapped with each other. However, the curve of CFS has strong volatility, or it is bustier than that of O(1). This means that the number of involuntarily switched in CFS varies a lot, while this number in O(1) is generally the same. However, there are three spikes in the curve of O(1), which means that O(1) scheduler sometimes scheduled a process more than it should be. To be more precise, the average of CFS is 13.6, while the average of O(1) is 13.4825. The standard deviation of CFS is 3.24, while the standard deviation of

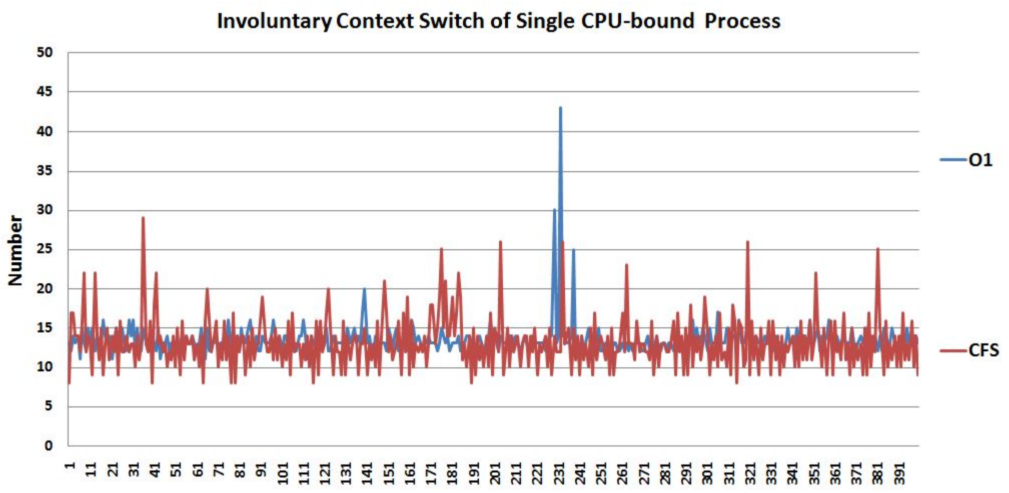


Figure 4.4: Involuntary Context Switch of Single CPU-bound Process

O(1) is 2.12. This means, that for a single CPU-bound process, the number of the times it is switched by CFS and O(1) are almost the same, but the curve of O(1) is relatively flat.

Voluntary Context Switch

The following figure 4.5 shows the the voluntary context switch. As described in the background section, the voluntary context switch is the number of times the process was context-switched voluntarily, for example, waiting for I/O.

As the figure shows, the voluntary context-switch of CFS is a fixed value,

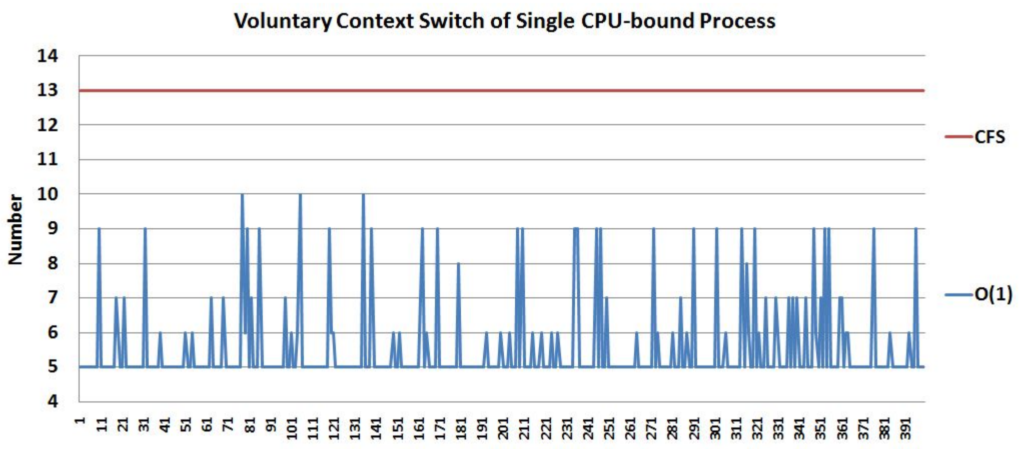


Figure 4.5: Voluntary Context Switch of Single CPU-bound Process

which is 13, while this number of O(1) has strong volatility, which varies from 10 to 5. To be more precise the average of CFS is 13, and standard deviation is 0. The average of O(1) is 5.47, and the standard deviation is 1.11. This means, when a CPU-bound process is scheduled by CFS, it has more stable and greater voluntary context switch, while O(1) has smaller but more unstable voluntary context switch.

CPU Percentage

The following figure 4.6 shows the CPU percentage. As described in the background section, this is the percentage of the CPU that processes got, computed as (%U + %S) / %E. One term of the performance of scheduler

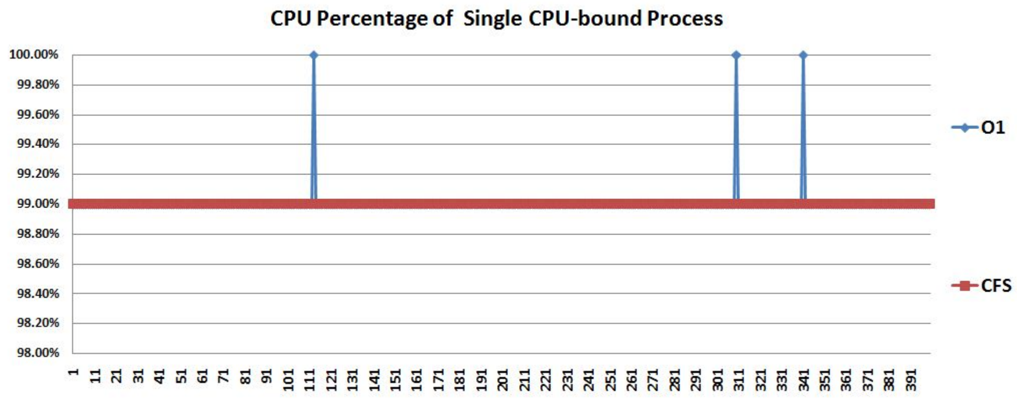


Figure 4.6: CPU Percentage of Single CPU-bound Process

is the efficiency, which means how busy it can make the CPU be. As the figure shows, both schedulers keep the CPU as busy as to 99.0%. This means, both schedulers has good performance to keep CPU busy.

The above figure tells the information about scheduler when there is single CPU-bound process running in the system. The next section will describe the result when there are a number of CPU-bound processes running in the system.

4.1.2 Processes with Same Priority

The results from 1 single process offer a base line for the comparison. The next part presents the results of multiple results. Results from multiple simultaneous processes with same priority are shown in this section. The next section will cover results from multiple simultaneous processes with different priorities. The number of simultaneous processes increases from 100 to 1000.

Elapsed Time

The following figure 4.7 shows the elapsed time of 100 simultaneous CPU-bound processes from one test.

The figure demonstrates that, the elapsed time of CFS is greater than that

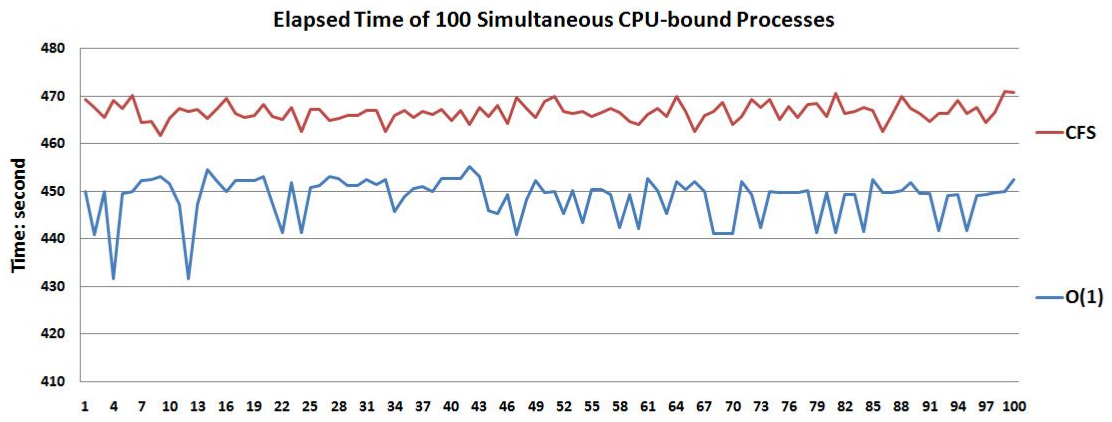


Figure 4.7: Elapsed Time of 100 CPU-bound Processes

of O(1). To be more precise, the average of CFS is 466.66, 4.0% greater than that of O(1), which is 448.65. The figure also shows that the curve of O((1) has strong volatility than that of CFS. To be more precise, the standard deviation of O(1) is 4.45, while this number of CFS is 1.90. This means, this 100 simultaneous CPU-bound processes have more fair CPU share in CFS than that in O(1).

Involuntary Context Switch

The following figure 4.8 is the comparison of involuntary context switch.

As the figure shows, there is huge difference between CFS and O(1) with

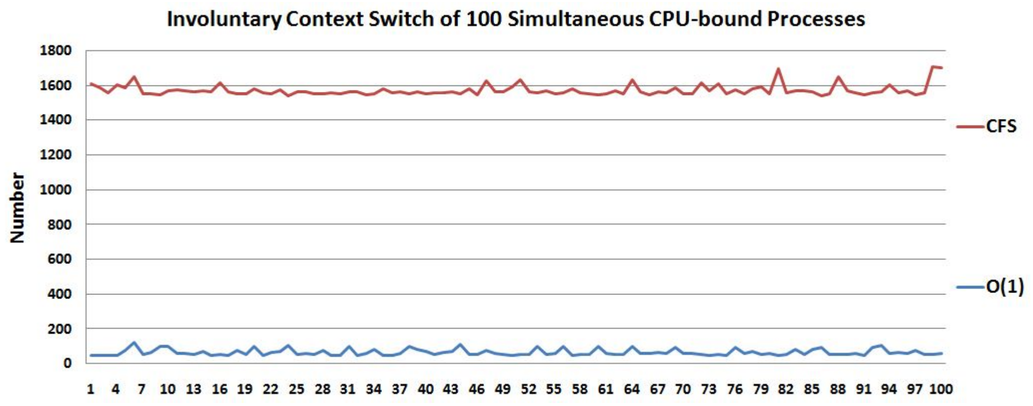


Figure 4.8: Involuntary Context Switch of 100 CPU-bound Processes

respect to involuntary context switch. The average involuntary context switch of CFS is 1573.53, while this number of O(1) is 67.47. This means, when there are 100 simultaneous CPU-bound processes in the system, every process will be switched for 1573.53 times by CFS scheduler, and 67.47

times by O(1) scheduler.
The standard deviation of this number is 31.87, while 18.36 with O(1). Although the standard deviation of CFS is greater than that of O(1), considering the value of average, the data of O(1) have stronger volatility than that of CFS. To measure this, the coefficient of variation is needed here. It measures relative variation, whereas the standard deviation is a measure of absolute variation, and is a way of comparing the variation between different sets of data. The formula of coefficient of variation is:

```
coefficient of variation  = standard deviation / mean(average)
```

So, in this case the coefficient of CFS is:

$$31.87/1573.53 = 2.02\% \tag{4.1}$$

The coefficient of O(1) is:

$$18.36/67.47 = 27.21\% \tag{4.2}$$

So, in spite of that the standard deviation of CFS, also known as the absolute variation, is larger than that of O(1), the coefficient of variation of CFS, also known as relative variation is much less than that of O(1). This means that CFS is really scheduling processes more fairly than O(1).

Voluntary Context Switch

The following figure 4.9 shows the voluntary context switch of above test. As the figure shows, the curve of voluntary context switch of both sched-

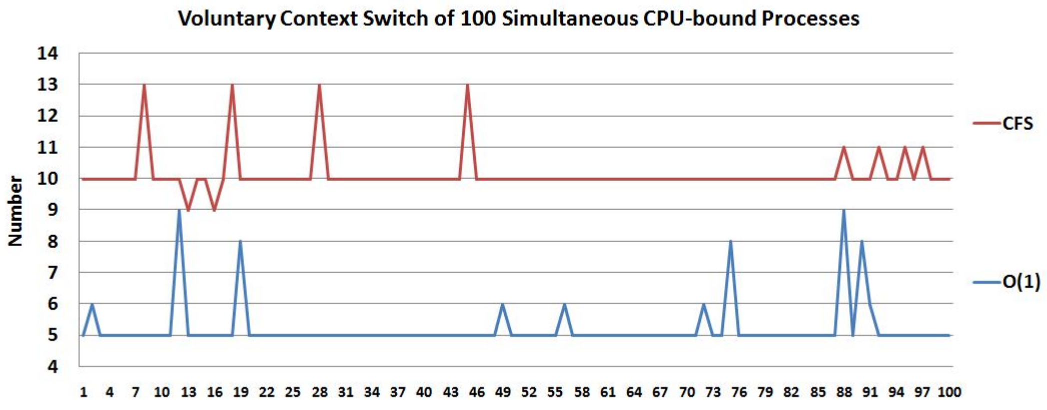


Figure 4.9: Involuntary Context Switch of 100 CPU-bound Processes

ulers keep flat, which means for those 100 simultaneous procession the system, they have similar times of voluntary context switch respectively. Compared with Figure 4.5 which shows the voluntary context switch of one process, the voluntary context switch doesn't change with the number of simultaneous process.

As for other output from *time* utility, such as user time and system, the number of processes doesn't change the value of them. And for the CPU percentage, it goes down to 1% in the case of 100 processes, and becomes 0 when number keeps increases. Since those output from both schedulers are the same, they are not presented.
The above is the results from 100 simultaneous processes, which is the fist group; the last group is the results from 1000 simultaneous processes. Between them, tests of 200, 400, 600 and 800 simultaneous processes are done, however, those results are not shown individually but as a summary at the end of this section.

Elapsed Time

The following figure 4.10 is the comparison of elapsed time of 1000 simultaneous CPU-bound processes.

As the figure shows, there are two main difference between the curve

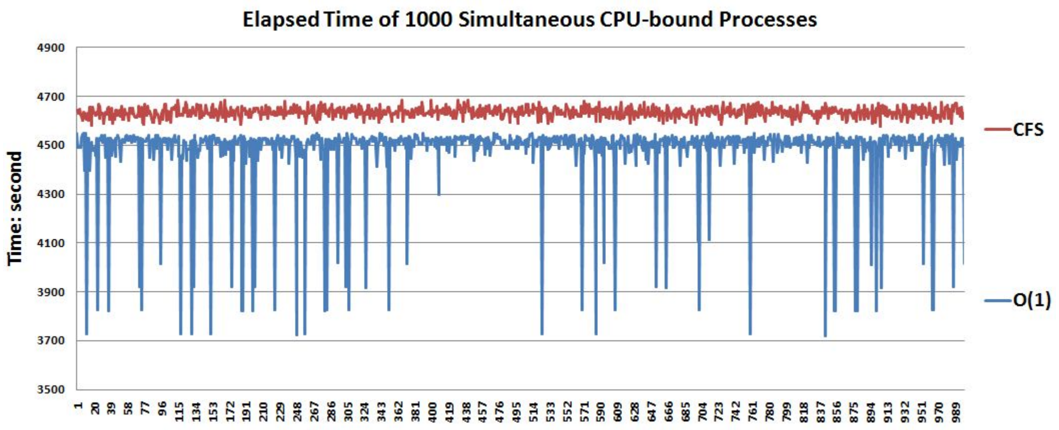


Figure 4.10: Elapsed Time of 1000 CPU-bound Processes

of CFS and O(1). First, the elapsed time of CFS is significantly larger than that of O(1). To be precise, the average of CFS is 4634.90, 156.5 seconds higher than that of O(1), which is 4478.39. This means that for those 1000 simultaneous processes, they take more time to finish when they are scheduled by CFS than by O(1). Second, the curve of O(1) is much more fluctuant than that of CFS. On the other hand, O(1) has much stronger volatility than that of CFS. This means that for those 1000 simultaneous processes, they are more fairly scheduled by CFS than O(1).

Involuntary Context Switch

The following figure 4.11 is involuntary context switch of 1000 simultaneous processes.

As can be seen from the figure, there is huge difference between O(1) and

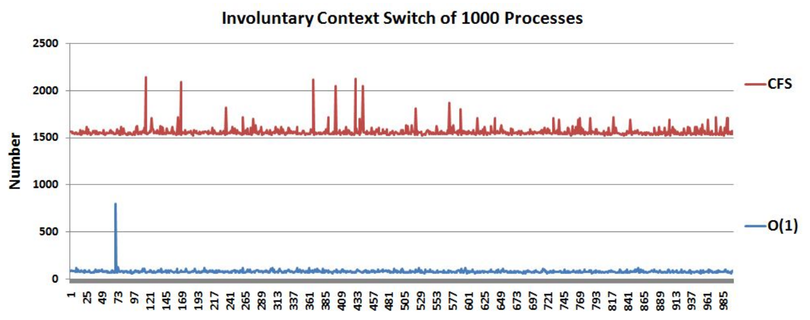


Figure 4.11: Involuntary Context Switch of 1000 CPU-bound Processes

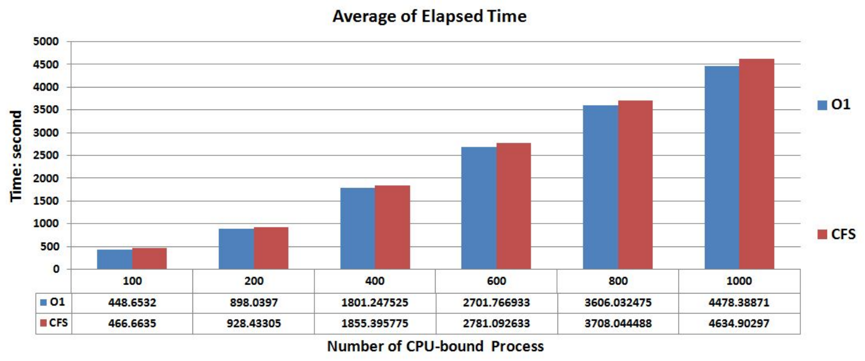
CFS. First, the involuntary context switch of CFS is much greater than that of O(1). This is the same with 100 simultaneous processes scenario as shown in the figure 4.9 . The average of involuntary context switch of CFS is 1564.34, and 78.76 with O(1). The standard deviation of both scheduler is 53.57 and 25.03. The coefficient of variations of both schedulers is:

$$CFS : 53.57/1564.34 = 3.42\% O(1) : 25.23/78.76 = 32.03\% \tag{4.3}$$

So, it shows that CFS has much higher involuntary context switch than that of O(1), but it schedules processes more fairly than that of O(1). As for the voluntary context switch, it doesn't increase with the number of process. So this part of result is not shown here.
The above results are form two scenarios, 100 simultaneous processes and 1000 simultaneous processes. The next part lists all the summaries of results from all the scenarios, 100, 200, 400, 600, 800 and 1000.

Elapsed Time

The following figure 4.12 shows average elapsed time of 100, 200, 400, 600, 800 and 1000 simultaneous processes.
As the figure shows, quite obviously, CFS has greater elapsed time than



	100	200	400	600	800	1000
🟦 O1	448.6532	898.0397	1801.247525	2701.766933	3606.032475	4478.38871
🟥 CFS	466.6635	928.43305	1855.395775	2781.092633	3708.044488	4634.90297

Figure 4.12: Average Elapsed Time of O(1) and CFS

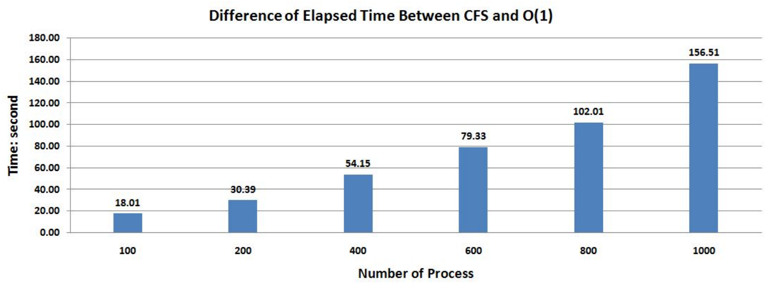


Figure 4.13: Difference of Elapsed Time

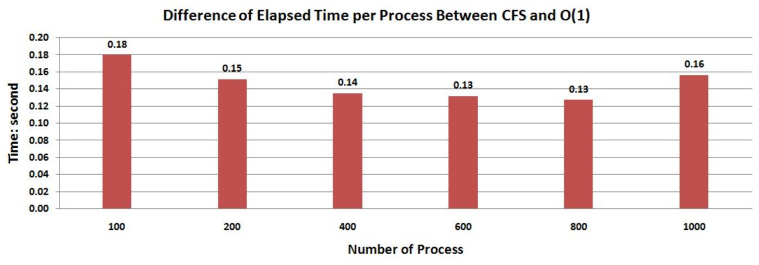
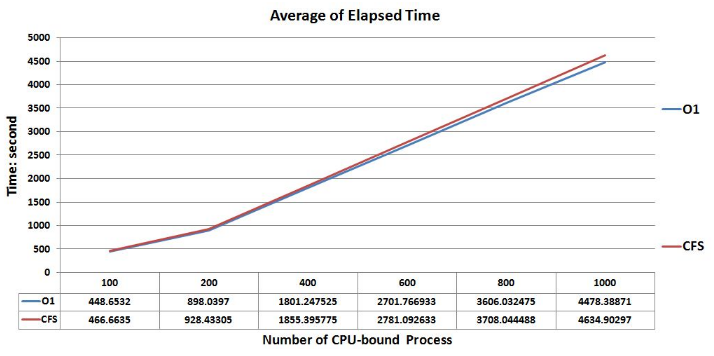


Figure 4.14: Difference of Elapsed Time per Process

O(1) in each scenario. The difference of elapsed time is also increasing with the number of process as the above figure 4.13 show.

However, the difference of elapsed time per process is decreasing when the number of process increases from 100 to 800 as can be seen in the figure 4.14. This means that the CFS is trying to catch up with O(1) to reduce the difference per process. But, this tendency stops at the point of 800, which means that CFS again starts to fall behind of O(1).

Back to the figure 4.12 , it also shows that, both CFS and O(1) follow almost linear distribution, as the following figure 4.15 shows. This means, both sched-

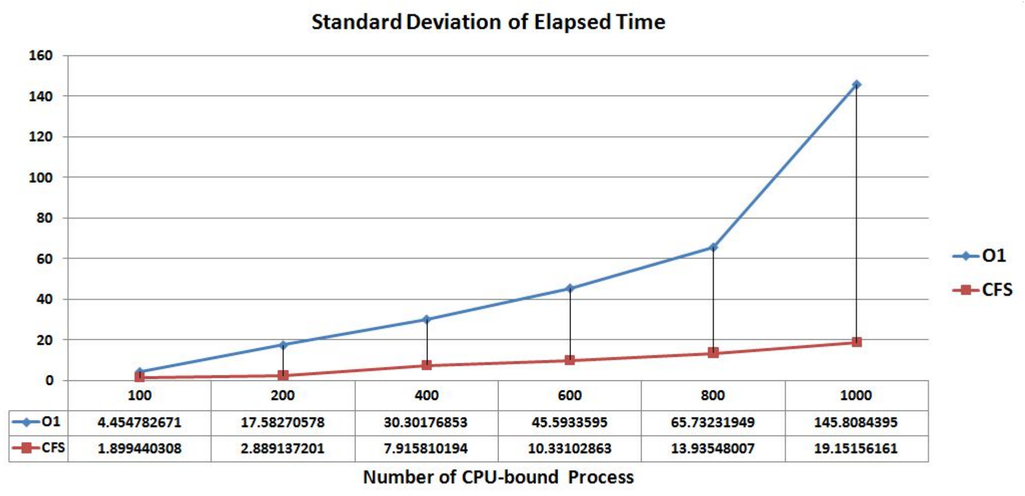


	100	200	400	600	800	1000
O1	448.6532	898.0397	1801.247525	2701.766933	3606.032475	4478.38871
CFS	466.6635	928.43305	1855.395775	2781.092633	3708.044488	4634.90297

Figure 4.15: Average Elapsed Time of O(1) and CFS

uler scale well when the number of processes increases from 100 to 1000.

The next figure 4.16 shows the standard deviation of elapsed time of CFS and O(1).
As the figure shows that, the standard deviations of elapsed time of both



	100	200	400	600	800	1000
O1	4.454782671	17.58270578	30.30176853	45.5933595	65.73231949	145.8084395
CFS	1.899440308	2.889137201	7.915810194	10.33102863	13.93548007	19.15156161

Figure 4.16: Standard Deviation of Elapsed Time of O(1) and CFS

schedulers increase. The differences are, first, the standard deviation of CFS increases slightly with a linear distribution. However, the standard deviation of O(1) elapsed time doesn't follow exactly linear distribution. To be more precise, when the number of process increases from 100 to 600, it follows linear distribution. But when the number of process increases from 800 to 1000, the standard deviation of O(1) jumps sharply. This means that, at the point of 800, the O(1) scheduler begins to suffer from such amount of processes, and can not manage to be make processes fair to each other as well as before.

Involuntary Context Switch

The following figure shows the involuntary context switch.
As can be seen from the graph, the involuntary context switch of CFS is much greater than that of O(1). What is the same for both schedulers is that this involuntary context switch doesn't change with the number of process.

4.1.3 Processes with Different Priorities

In this scenario, all the simultaneous processes in the system are divided into two groups. One group is called normal; the other group is called nice. All the processes in the normal group have default priority, and all the processes in the nice group have a nice value of 10, which means nice processes have a lower priority compared with normal. The same number of nice and normal

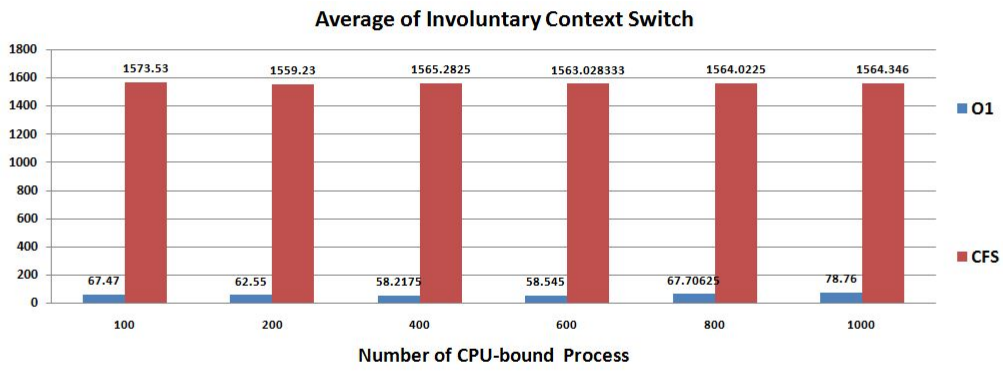


Figure 4.17: Average of Involuntary Context Switch of O(1) and CFS

processes will run in the same time in the system. The total number of simultaneous processes increases from 100 to 1000.

Elapsed Time

The following figure 4.18 shows elapsed time when there are100 nice processes and 100 normal processes, in total 200 simultaneous processes in the system.

The figure shows quite a lot differences between CFS and O(1).

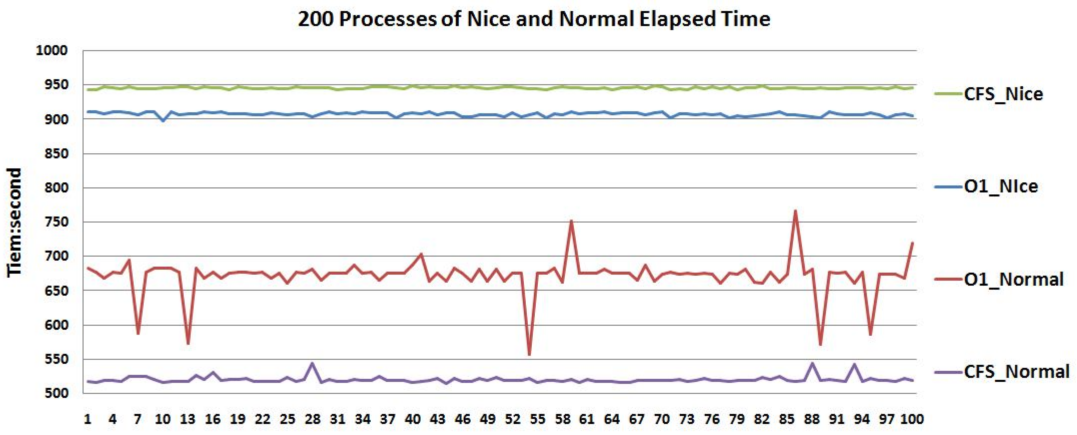


Figure 4.18: 200 Processes of Nice and Normal Elapsed Time

- First, CFS nice and CFS normal are located at the top and at the bottom of the graphic, which means the difference between CFS nice elapsed time and CFS normal elapsed time is greater than that of O(1).
- Second, the CFS nice is greater than that of O(1), and CFS normal is less than that of O(1), which means , with the same nice value of 10, nice processes in CFS are much nicer to normal processes than that in O(1).

- Third, compared with nice elapsed time, both CFS normal elapsed time and O(1) elapsed time have stronger volatility, especially, the curve of O(1) normal elapsed time is the most fluctuant one.

Compared with the scenario which also has 200 processes with same priority, as the following figure 4.19 shows, the CFS nice elapsed time is larger than CFS

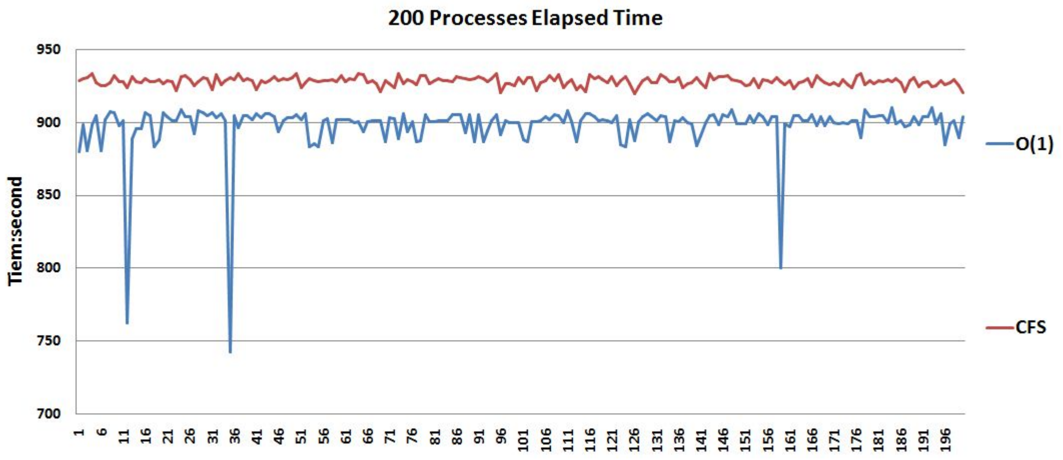


Figure 4.19: Elapsed Time of 200 CPU-bound Process with Same Priority

elapsed time with the same priority of processes. But for O(1), the nice elapsed time is approximately the same with that of with same priority of processes.
The flowing 4.1 table shows the average and standard deviation of elapsed

Table 4.1: 200 Processes Elapsed Time

	CFS			O(1)		
Value	**same**	**nice**	**normal**	**same**	**nice**	**normal**
average	**928.433**	**946.06**	**520.729**	**898.04**	**907.93**	**672.516**
stdv	**2.88914**	**1.38591**	**4.82604**	**17.5827**	**2.66151**	**26.4716**

time with respect to CFS and O(1) both in same priority scenario and different priorities scenario.
The next figure 4.20 shows the involuntary context switch of 200 processes with 2 different groups of priorities. As the figure shows, first, nice processes have larger involuntary context switch, which means that a nice process is switched more times than a normal process for both schedulers. Second, the difference between nice and normal of CFS is large, however, the difference between nice and normal of O(1) is quite small.
Compared with the same scenario which also has 200processes with same priority, as the following figure 4.21 shows, The nice involuntary context switch of CFS with nice processes is larger than that of processes with same priority. Besides, the involuntary context switch of normal process is smaller than that of processes with same priority. While for O(1), involuntary context switch for nice processes, normal processes and processes with same priority are approximately the same.

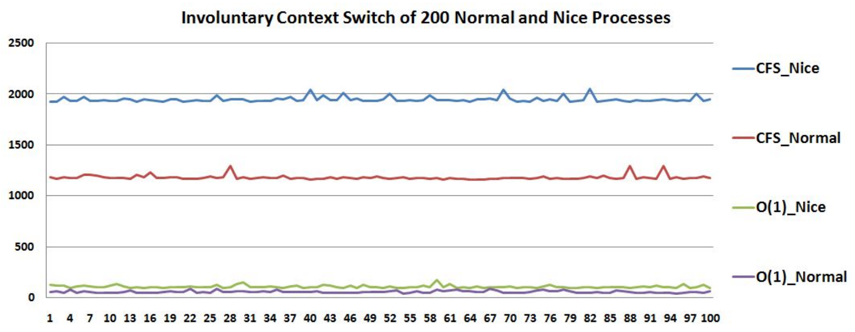


Figure 4.20: Involuntary Context Switch of 200 Nice and Normal CPU-bound Process

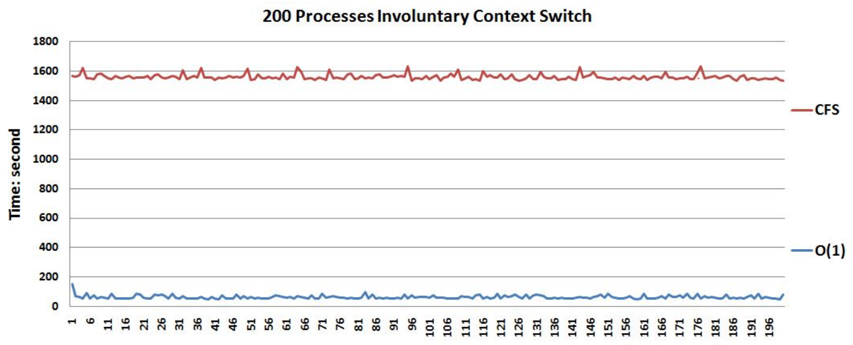


Figure 4.21: Involuntary Context Switch of 200 CPU-bound Processes with Same Priority

The following table 4.2shows the average and standard deviation of involuntary context switch with respect to CFS and O(1) both in same priority scenario and different priorities scenario.

The above results are form the scenarios of 200 processes. The next part will

Table 4.2: 200 Processes Involuntary Context Switch

	CFS			O(1)		
Value	**same**	**nice**	**normal**	**same**	**nice**	**normal**
average	**1559.23**	**1948.86**	**1184.18**	**62.55**	**107.66**	**61.33**
stdv	**19.47**	**25.29**	**2.64**	**12.26**	**12.49**	**9.55**

list all the summaries of results from all the scenario of 100, 200, 400, 6000, 800 and 1000 simultaneous processes with different priorities.

Elapsed Time

The following figure 4.22 shows average elapsed time of 100, 200, 400, 600, 800 and 1000 simultaneous processes with different priorities.

As the figure shows, first, as the number of processes increase, CFS nice

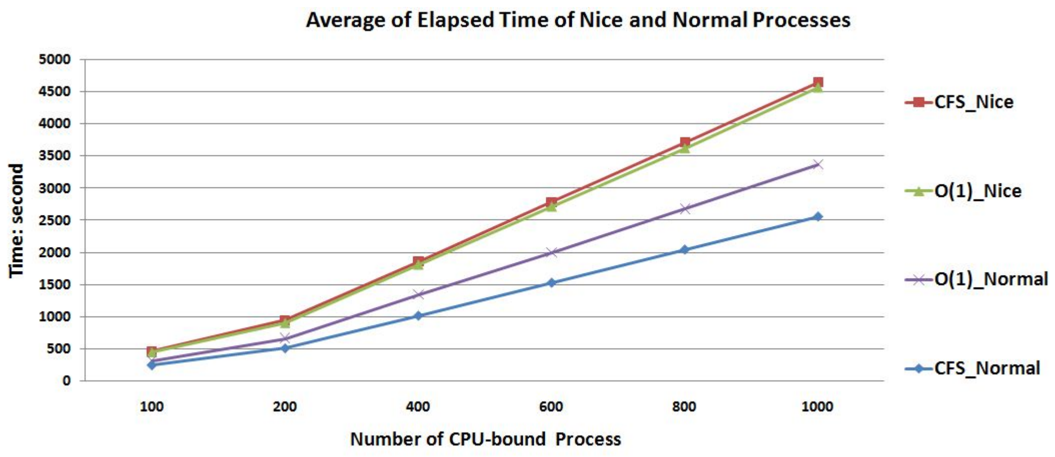


Figure 4.22: Average of Elapsed Time of Nice and Normal Processes

elapsed time has largest value, then O(1) nice, O(1) normal and CFS normal. Second, the difference between nice and normal increases with the number of processes. Third, the difference between nice and normal of CFS is larger than that of O(1). The figure tells that, when the number of process increases, for single process with the same nice value, it is getting nicer to the normal process by giving more CPU share to normal processes.

The next figure 4.23 shows the difference between average nice and normal elapsed time.

As the figure shows, when the number of process increases, the difference of nice and normal also increases. Both CFS and O(1) have approximately

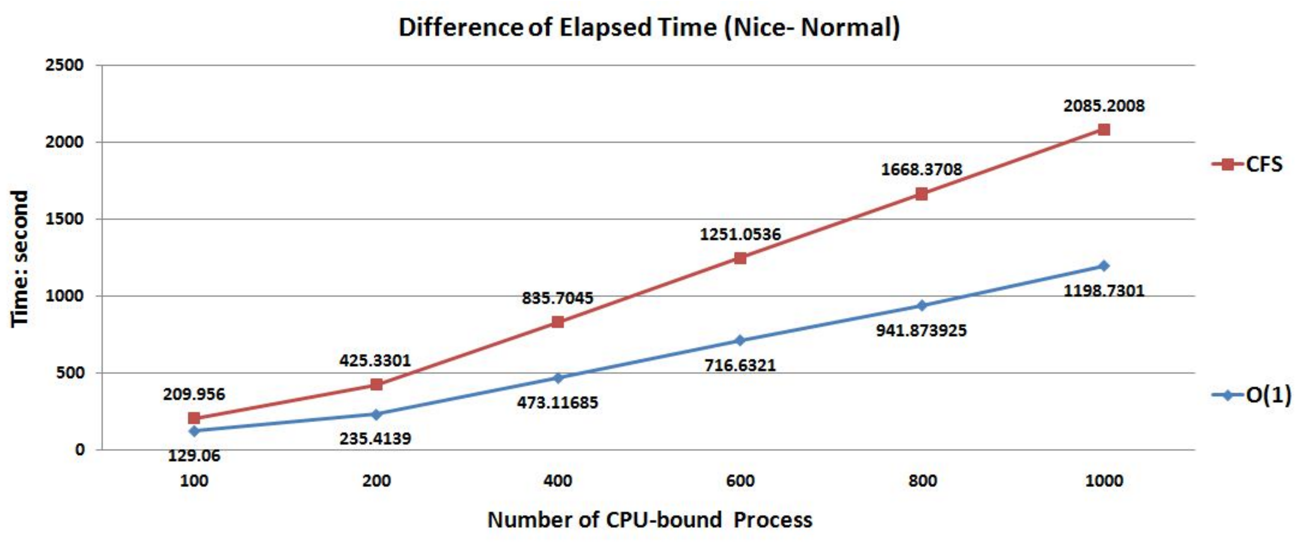


Figure 4.23: Difference Between Average Nice Elapsed Time and Average Normal Elapsed Time

the same trend. However, CFS have larger difference between nice and normal than that of O(1).

The next figure 4.24 shows the standard deviation of elapsed time of all the scenarios.

As the figure shows, the standard deviation of elapsed time with CFS

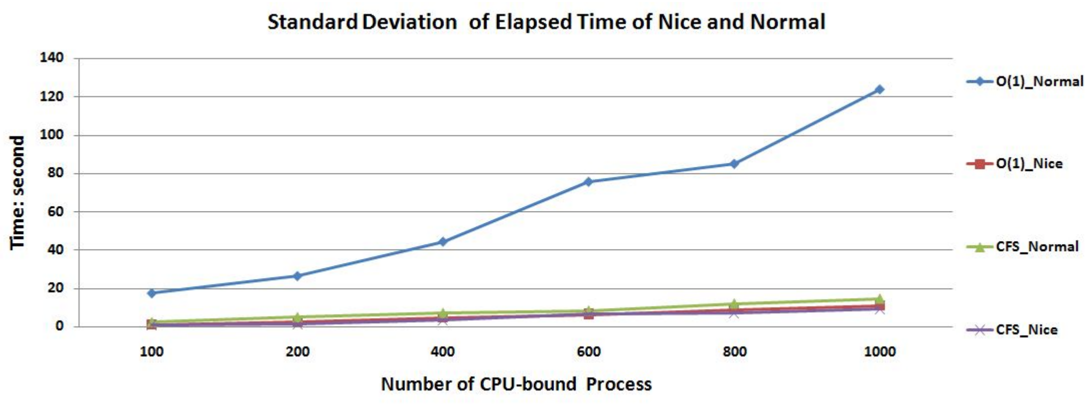


Figure 4.24: Standard Deviation of Elapsed Time of Nice and Normal Processes

keeps approximately the same. But for O(1) normal, the value keeps increasing with the increment of number of process. This means, when the number of process increases, the normal process are scheduled more and more unfairly, because they are getting more unfair share of processing time.

The next figure 4.25 shows the difference of standard deviation of nice elapsed time and normal time. As the figure shows, when the number of process increases, the difference between nice and normal of CFS keeps

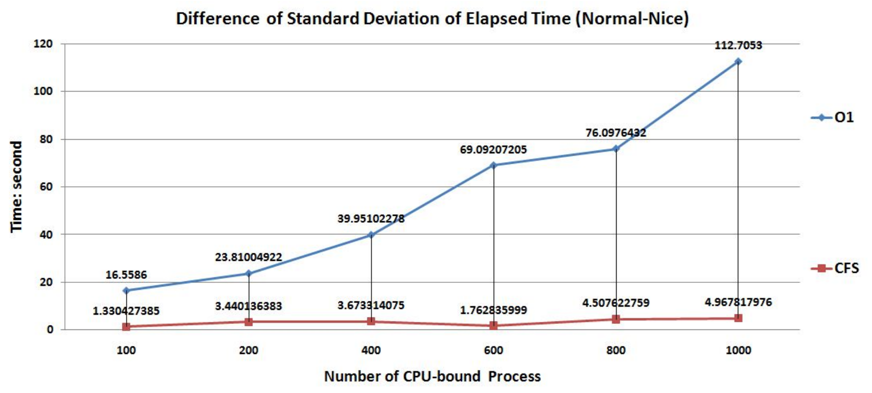


Figure 4.25: Difference of Standard Deviation of Nice Elapsed Time and Normal Elapsed Time

almost the same. However, for O(1), the difference is increasing.
The next figure 4.26 shows the average of involuntary context switch of 100,200,400,600,800 and 100 simultaneous processes with different priorities.
As the figure shows, the involuntary context switch for CFS and O(1) keeps almost the same. This means, no matter how many processes in the system, the number of times of switched by scheduler keeps the same. Both schedulers scale well with the increment of process.
The next figure 4.27 shows the standard deviation of involuntary context switch.
The graph demonstrates that CFS not only has greater value of average involuntary context switch but also standard deviation of involuntary context switch, no matter those processes have the same priority or different priorities.

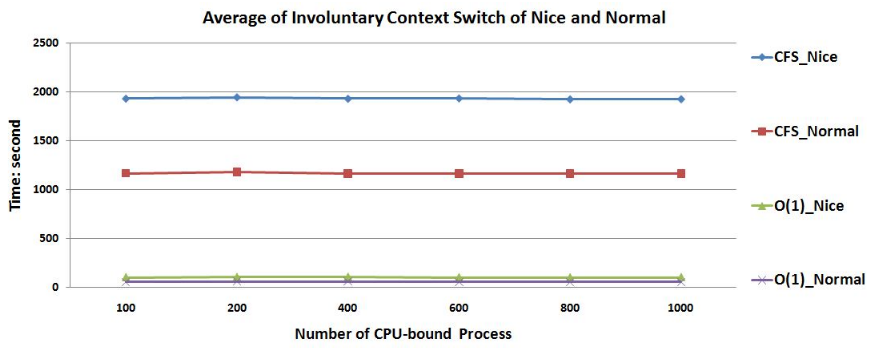


Figure 4.26: Average of Involuntary Context Switch of Nice and Normal Processes

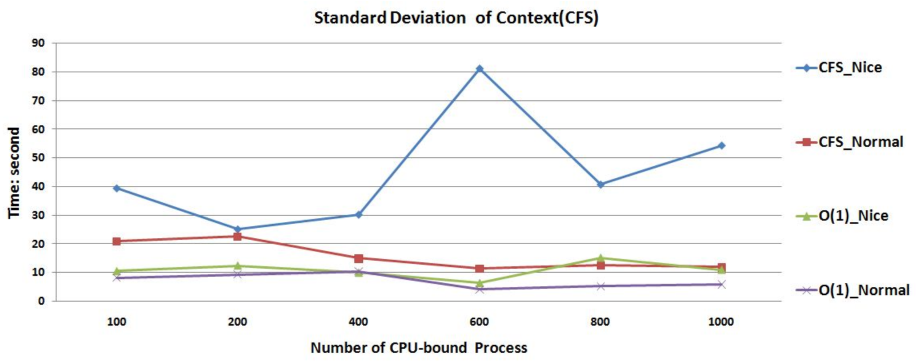


Figure 4.27: Standard Deviation of Involuntary Context Switch of Nice and Normal Processes

4.2 I/O-bound Process

4.2.1 Single I/O-bound Process

First, results from the scenario of single I/O-bound Process are presented in this section. Outputs from *time* utility and *vmstat* are also shown in this section.
The following figure 4.28 shows the elapsed time of 1 *bonnie++* process of CFS and O(1). There are two main differences from the figure toward CFS and O(1).

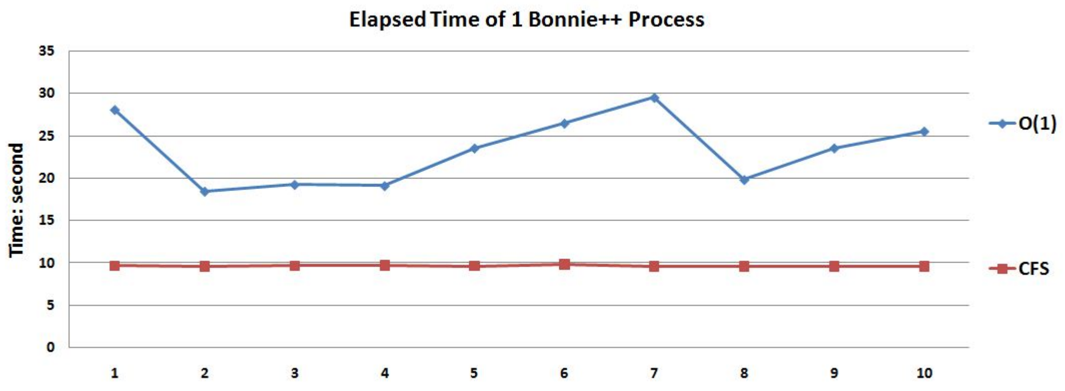


Figure 4.28: Elapsed Time of 1 Bonnie++ Process

First, the elapsed time of O(1) is significantly larger than that of CFS. Second, the curve of O(1) has stronger volatility than that of CFS, which means O(1) can't promise same result from same process, which makes O(1) performance difficult to predict. However, for CFS, the elapsed time keep almost the same, which shows scheduler is much stable than O(1).
The next figure 4.29 shows the user time of both schedulers.
As the figure shows, the user time of bonnie++ process is significantly small compared with CPU-bound process, as shown in the figure 4.2 . However, the system time is just opposite. As the next figure 4.30 shows, the system time of

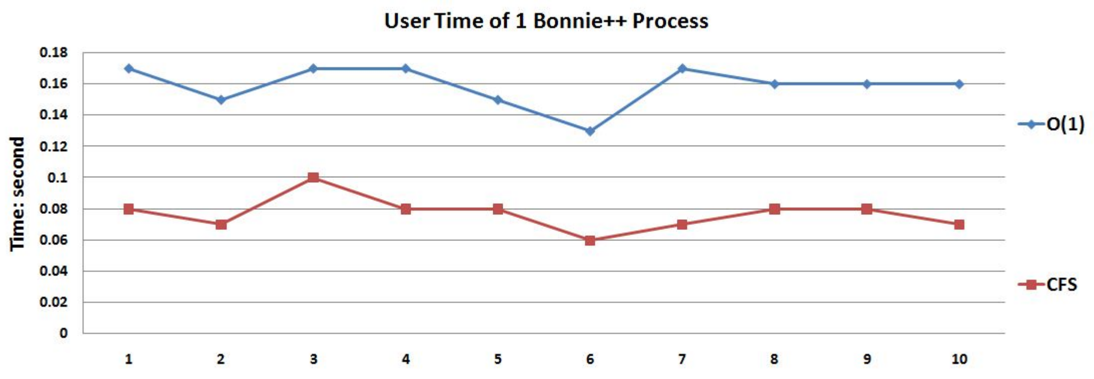


Figure 4.29: User Time of 1 Bonnie++ Process

bonnie++ process is larger than that in the CPU-bound process, as shown in the figure 4.3 .

This means, for pure I/O-bound process, much of the time is spent in the

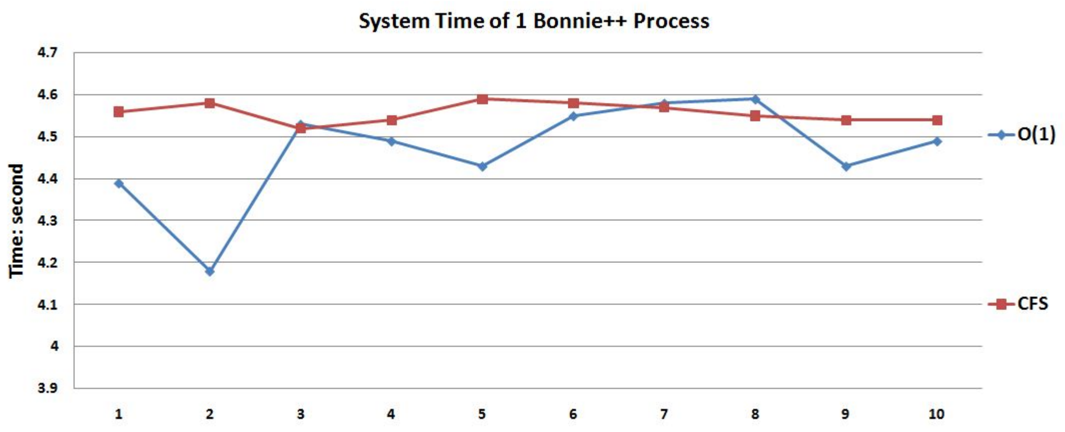


Figure 4.30: System Time of 1 Bonnie++ Process

kernel model.

The next figure 4.31 shows the involuntary context switch of both schedulers.

As the figure shows, CFS has larger involuntary context switch, while the curve tends to be flatter. This means, for the I/O-bound process, it is scheduled more times than that of O(1).

Compared with that involuntary context switch of CPU-bound process, as shown in the figure 4.4 , the involuntary context switch of I/O-bound process is much much higher. This means, for both schedulers, I/O-bound process is more times switched than CPU-bound process. And again, CFS has more involuntary context switch than O(1).

The next figure 4.32 shows the percentage of CPU of both schedulers.

bonnie++ process of CFS and O(1). Unlike CPU-bound process, which takes all the CPU, the I/O-bound process doesn't full CPU. For CFS, the process takes about 48%, and for O(1), process takes about 20%.

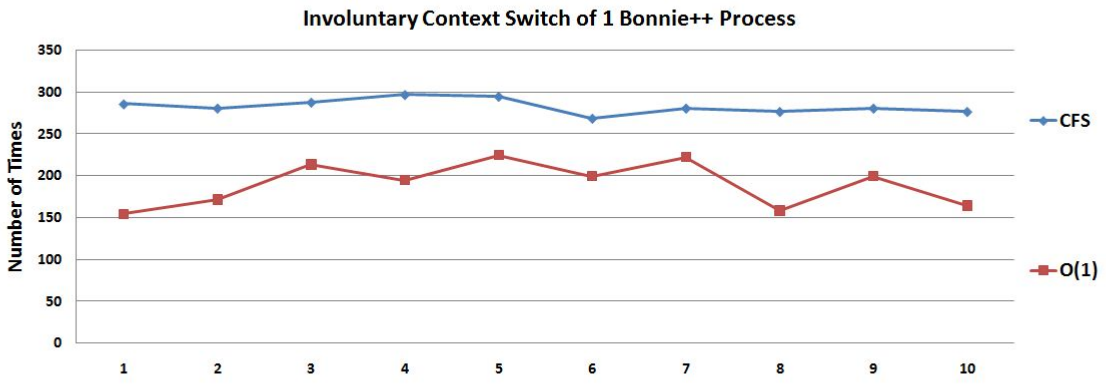


Figure 4.31: Elapsed Time of 1 Bonnie++ Process

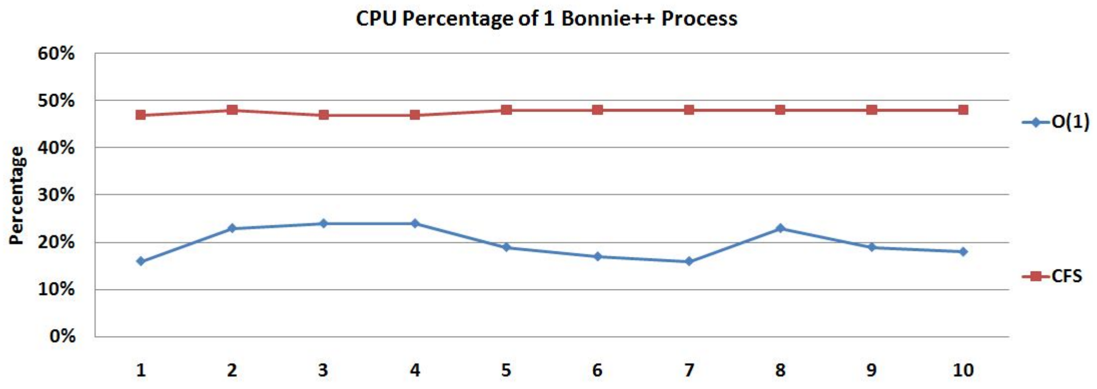


Figure 4.32: Percentage of CPU of 1 Bonnie++ Process

4.2.2 Multiple I/O-bound Processes

The above result is the base line for multiple processes. Results from 2, 4, 6, 8,10,15 and 20 processes are presented as follows.
The following figure 4.33 shows average elapsed time of bonnie++ process.
There are four curves in the graphic, the red curve is the average elapsed time

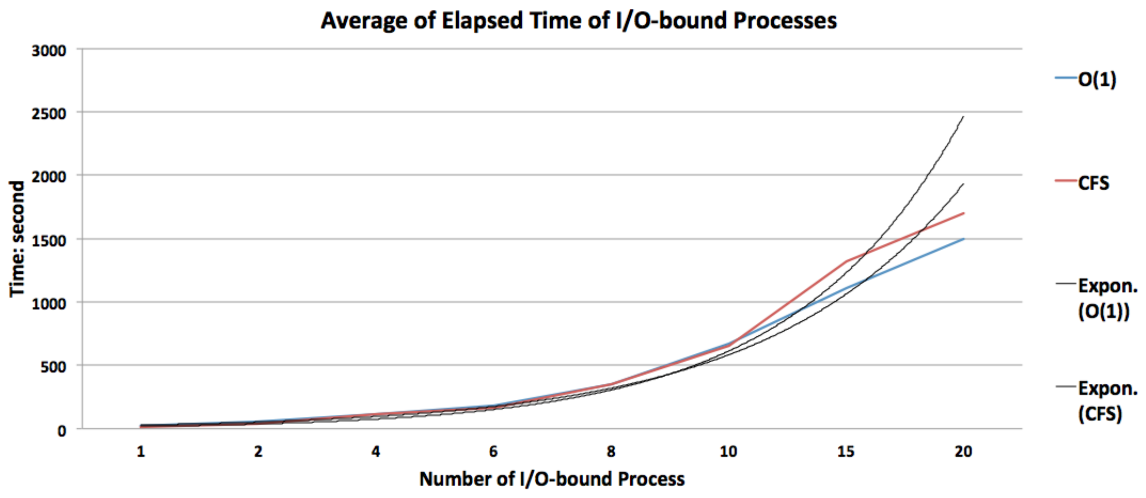


Figure 4.33: Average of Elapsed Time of Bonnie++ Process

of CFS, and the blue one is the elapsed time of O(1). Both curves go up with the increment of number of process. There are also two black lines, they are two exponential trend lines of CFS and O(1). As the figure shows, both CFS and O(1) follow approximate exponential distribution based on the data trend.

The next figure 4.34 shows the standard deviation of elapsed time.
As can be seen from the figure, up to 10 I/O-bound processes, the standard de-

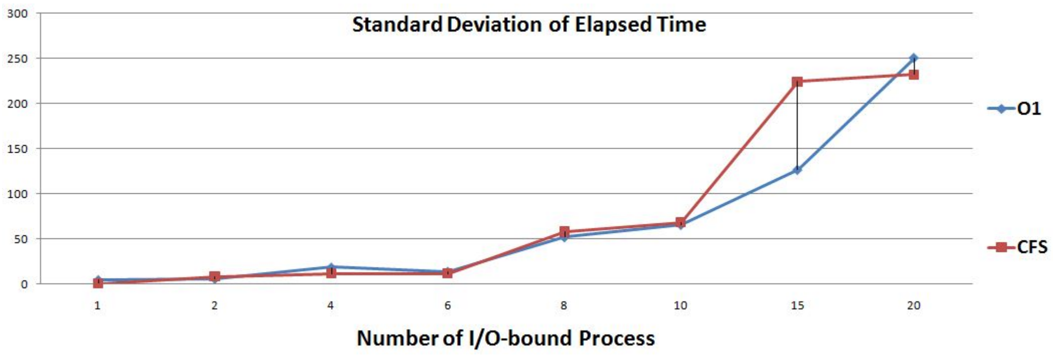


Figure 4.34: Standard Deviation of Elapsed Time of Bonnie++ Process

viations of both schedulers are quite close to each other. After that, CFS shows a higher standard deviation than O(1), which means O(1) schedule I/O-bound process more fairly than CFS! The next figure 4.35 shows the average of involuntary context switch.

The figure demonstrates that CFS has a higher involuntary context switch,

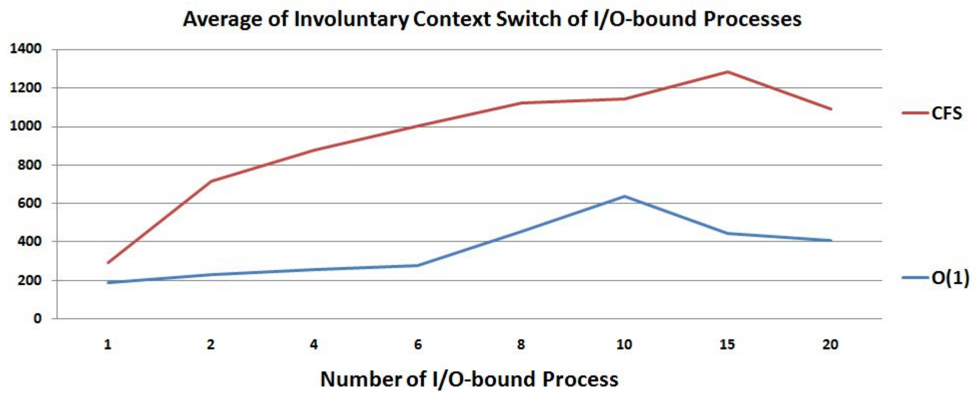
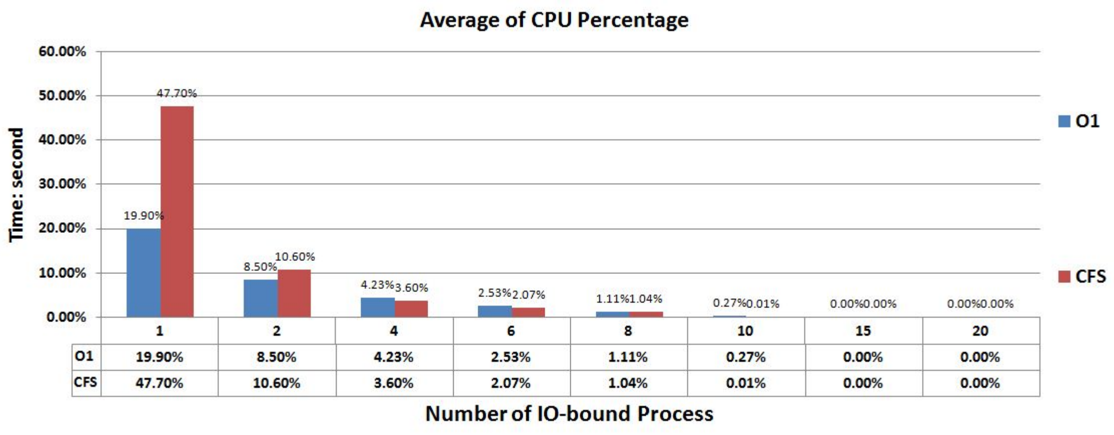


Figure 4.35: Average of Involuntary Context Switch of Bonnie++ Process

which means the I/O-bound prices is scheduled more times in CFS than in O(1). However, the voluntary context switch of both schedulers is a constant of 0, no voluntary context switch for both of them. The next figure 4.36 shows average of CPU percentage.

Compared with figure 4.35 , which shows the value of how much CPU a sin-



	1	2	4	6	8	10	15	20
O1	19.90%	8.50%	4.23%	2.53%	1.11%	0.27%	0.00%	0.00%
CFS	47.70%	10.60%	3.60%	2.07%	1.04%	0.01%	0.00%	0.00%

Figure 4.36: Average of CPU Percentage of Bonnie++ Process

gle I/O-bound process get, this figure shows that this value decreases when the number of process increase. This is reasonable, since the more processes, the less CPU percentage. However, this is quite different from CPU-bound process, as shown in the figure 4.6 , in which, as can be seen that both schedulers consume almost 100% CPU. As discussed in the background chapter, I/O-bound process doesn't consume much CPU. This is the same for both scheduler. The difference is that, CFS consume more CPU than O(1) as the above graph shows.

The next part shows results from *vmstat* , which displays information of the

whole system. The next figure 4.37 shows the average of time spent idle.
There is significant difference between two schedulers as shown in the graph.

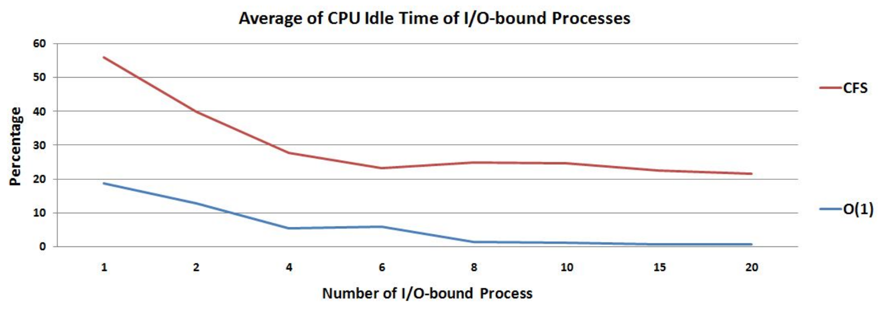


Figure 4.37: Average of Idle of Bonnie++ Process

The idle time of O(1) is quite smaller than CFS, which means there are idle time in CFS. Further more, the idle time of O(1) goes down to almost zero when there are 8 I/O-bound processes, while CFS still has some part of CPU idle. Generally speaking, it is not good to keep CPU idle because people want CPU works as hard as possible. However, even thoungH CFS has much higher idle time, the difference of elapsed time between them are not as impressive as idle time. The above data strongly back up the point that CFS has better performance than O(1).
The next figure 4.38 shows the number of interrupts per second, including the clock.
It is quite clear from the figure that, first, O(1) comes up with more interrupts

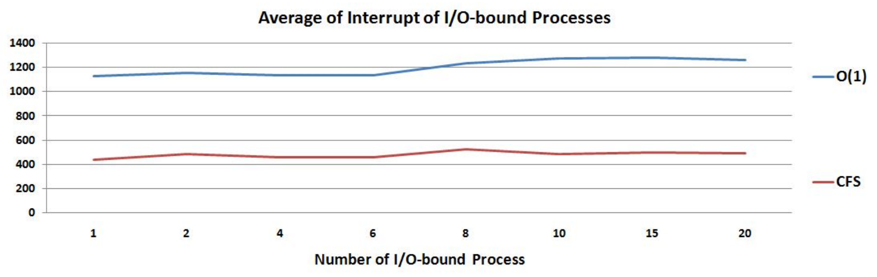


Figure 4.38: Average of Interrupts of Bonnie++ Process

than CFS; second, both schedulers scale well with I/O-bound process. The number of interrupts of both schedulers keeps almost unchanged.

4.3 Mixed-I/O-CPU Processes

This part, both CPU-bound and I/O-bound processes are running at the same time. Compared with only CPU-bound or I/O-bound process competing system resources, the result will show us how schedulers schedule different types of processes.

4.3.1 CPU-bound Processes Results

The next figure 4.39 shows the elapsed time of only 50 CPU-bound processes in the system, and elapsed time of 50 CPU-bound processes together with 10 I/O-bound processes in the system.
As we see from the figure, in O(1), the elapsed time of CPU-bound processes

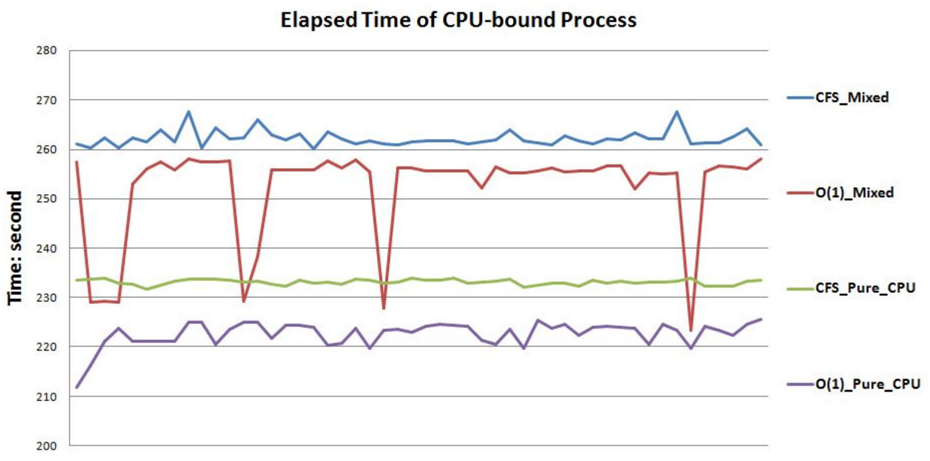


Figure 4.39: Elapsed Time of CPU-bound Process and Mixed Process

vary a lot when they are competing with I/O-bound processes, which means the CPU-bound processes are much affected by I/O-bound processes. However, for CFS, it still can keep those processes be fair to each other.
The next figure 4.40 shows the involuntary context switch of only 50 CPU-bound processes in the system, and elapsed time of 50 CPU-bound processes together with 10 I/O-bound processes in the system.
As can be seen from the figure, the involuntary context switch of CFS is much

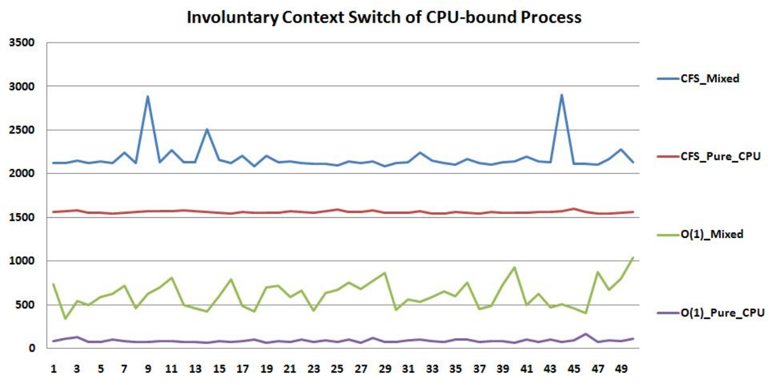


Figure 4.40: Involuntary of CPU-bound Process and Mixed Process

greater than that of O(1) in both scenarios. And CFS_Mixed and O(1)_Mixed show stronger volatility compared with CFS_Pure_CPU and O(1)_Pure_CPU.

4.3.2 I/O-bound Processes Results

The next figure 4.41shows the elapsed time of only 10 I/O-bound processes in the system, an elapsed time of of 50 CPU-bound processes together with 10 I/O-bound processes in the system. It is difficult to tell the difference from

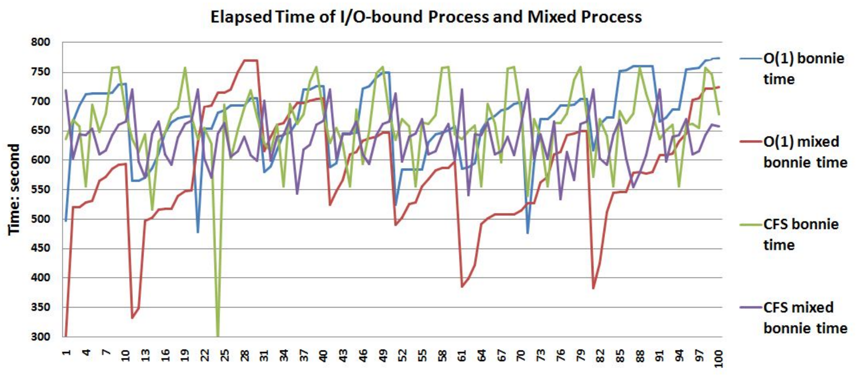


Figure 4.41: Elapsed Time

the figure, since all the curves are so overlapped. But the average of them, as the next table shows, that for both schedulers, the elapsed time goes down from from the situation where there are only I/O-bound processes to the situation where there are both I/O-bound process and CPU-bound process. This is weird, since there are 50 more CPU-bound process in the system, the elapsed time should goes up.

Table 4.3: Elapsed Time

	I/O-bound Process	Mixed Process
CFS	658.6824	636.0812
O(1)	668.3195	584.2723

The next figure 4.42 shows the involuntary context switch of only 10 I/O-bound processes in the system, and involuntary context switch of 50 CPU-bound processes together with 10 I/O-bound processes in the system. As the figure demonstrates, for CFS, the involuntary context switch of I/O-bound process increases from the situation where there are only I/O-bound processes to the situation where there are both I/O-bound process and CPU-bound process.
However, for O(1), the involuntary context switch of I/O-bound process decreases from the situation where there are only I/O-bound processes to the situation where there are both I/O-bound process and CPU-bound process. This is weird, since there are 50 CPU-bound processes more in mixed situation, the number of involuntary context switch is expected to go up.

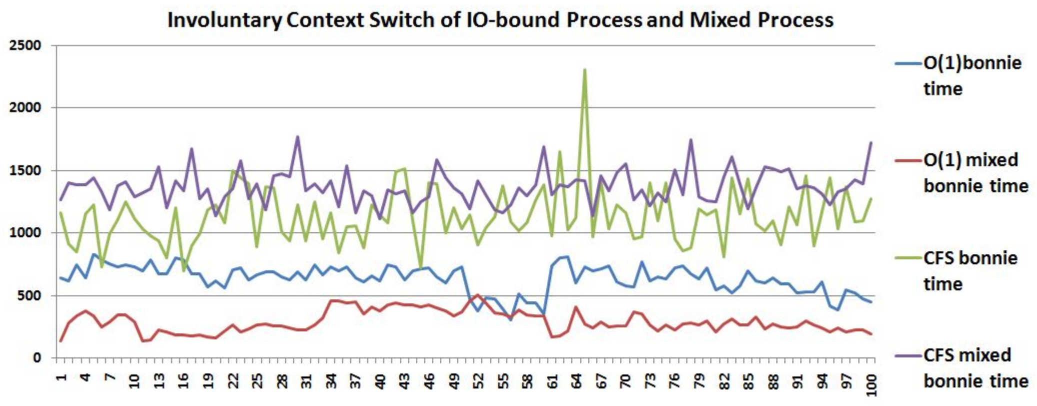


Figure 4.42: Involuntary Context Switch

Table 4.4: Involuntary Context Switch

	I/O-bound Process	Mixed Process
CFS	1138,13	1365,82
O(1)	637,51	291,52

4.4 Interactive Process

4.4.1 Editing Process

The next figure 4.43 shows the elapsed time of editing processes.
It is shown that CFS has greater elapsed time than 0(1), to be precise, about

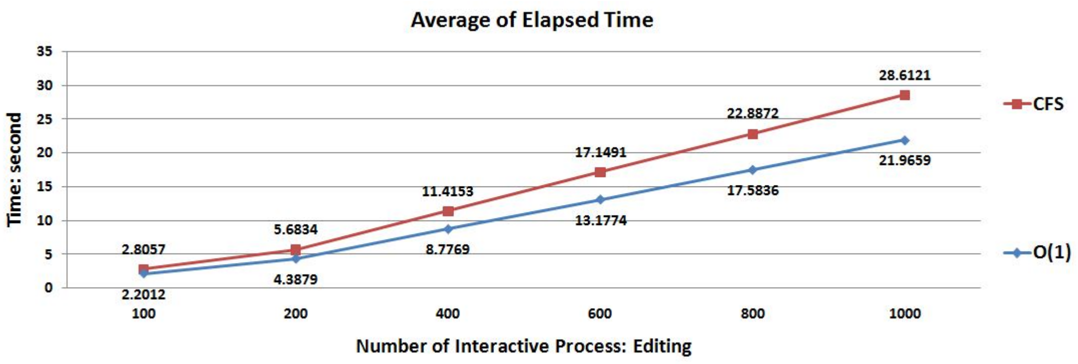


Figure 4.43: Average of Elapsed Time of Editing Processes

30% greater than O(1), as, shown in the below table. Compared with the situations of CPU-bound process and I/O-bound process, where difference of elapsed time between CFS and O(1) is relatively small, it can be conclude that O(1) does have "improvement" with respect to interactive editing process. The reason, as described in the background part, O(1) has extra bonus for interactive process. The next figure 4.4 shows the standard deviation of elapsed time.

Table 4.5: Elapsed Time of 100 Editing Process

	100	200	400	600	800	1000
CFS	2.8057	5.6834	11.4153	17.1491	22.8872	28.6121
O(1)	2.2012	4.3879	8.7769	13.1774	17.5836	21.9659
(CFS-O(1))/O(1)	27.46%	29.52%	30.06%	30.14%	30.16%	30.26%

It is shown from the figure that, CFS has lower standard deviation, which

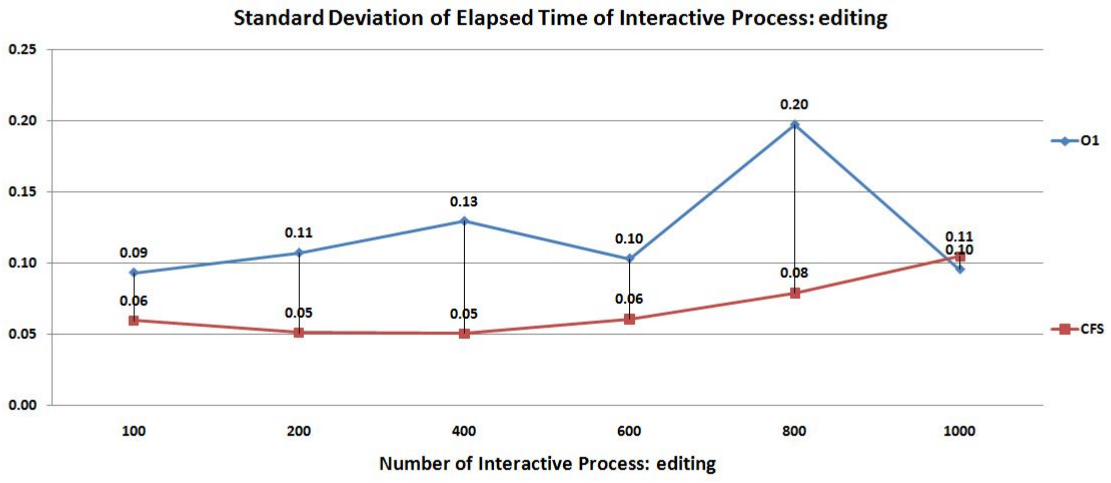


Figure 4.44: Standard Deviation of Elapsed Time of Editing Processes

means those editing processes in CFS are more fairly scheduled. But unlike in the CPU-bound Processes scenario, this difference is very slight.

The next figure 4.45 shows the involuntary of editing processes.

As the figure demonstrates, the difference between CFS and O(1) is quite

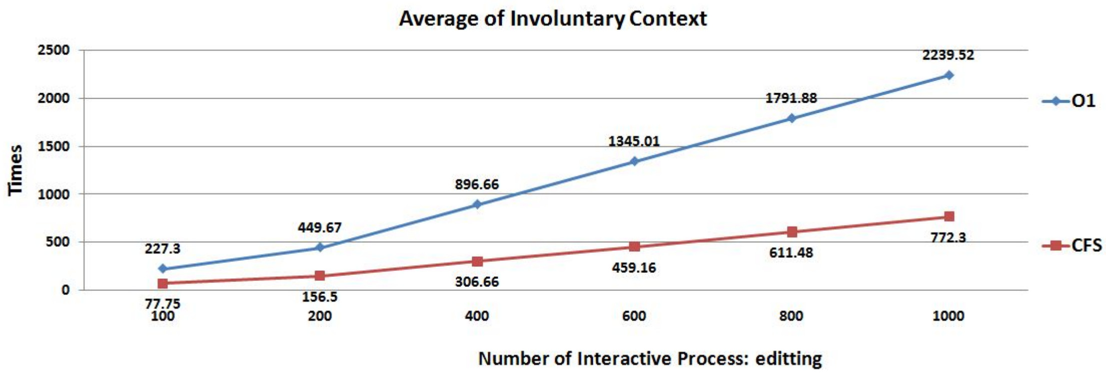


Figure 4.45: Involuntary Context Switch of Editing Processes

impressive: O(1) has much greater of involuntary context switch than CFS. As discussed before, an involuntary context switch occurs when a process finishes its time slice or when the system identifies a higher-priority thread to

run. Since all the editing processes have the same priority, and there is no other user process running in the system, the only reason for context switch here is that this process has ran out its timeslice. Based on this, it can be concluded that O(1) has much smaller timeslice than CFS when it is coming to interactive processes.
As shown in other processes, CFS always has greater involuntary context switch than O(1). However, when it comes to editing process, it is opposite. The background part describes that O(1) has extra bonus for interactive process. The interactive process has higher priority

The next figure 4.46 shows the voluntary of editing processes.
The graphs shows that the number of voluntary context switch increases with

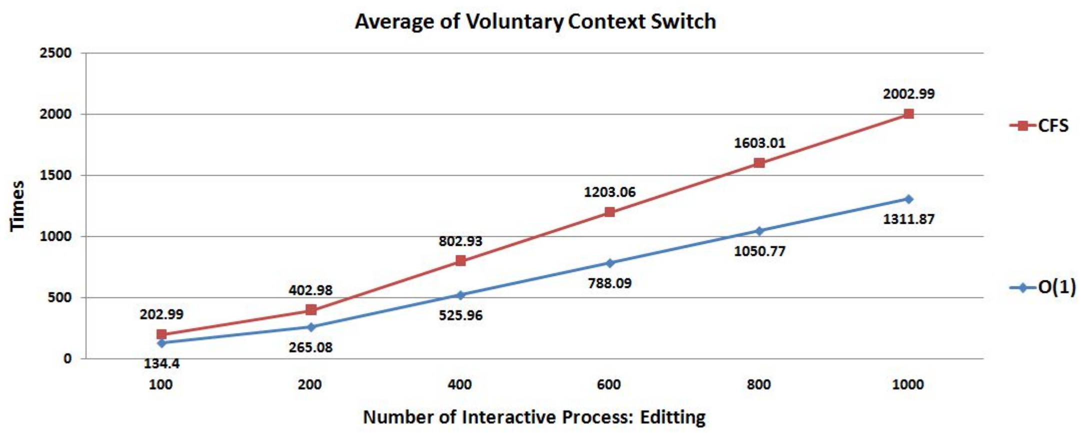


Figure 4.46: Voluntary Context Switch of Editing Processes

the number of editing processes for both schedulers, and CFS has greater value of that than O(1). As it is known that, a voluntary context switch occurs when a process must wait for the availability of certain resources or an event arrives. The editing processes waits for data from input from a file, then it doesn't need for CPU. A *sleep()* function is called in the system, which then preempted the process.

4.4.2 Firefox

In this part, the results from vmstat will be presented as there are 20 firefox processes running in the system.
The next figure 4.47 shows the context switch.
As the graph demonstrates, the curve of both schedulers just overlaps each other, which means the number of context switch of both schedulers with respect to Firefox process is approximately the same to each other. In order to ensure that this result is not produced by chance but has statistical significance, a t.test is needed here. The null hypothesis in this case is that the number of context switch of both schedulers is the same. Results of this test is included

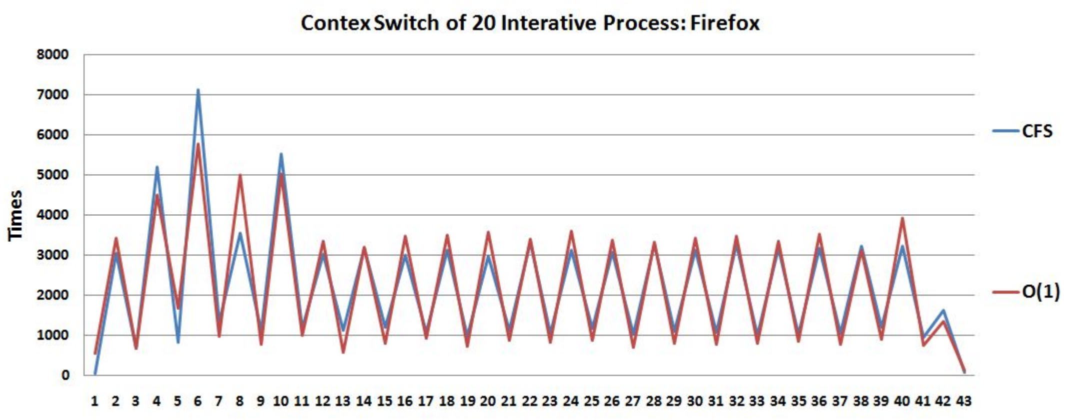


Figure 4.47: Context Switch of 20 Firefox Processes

in the following table:

```
        Welch Two Sample t-test

data:  firefox.cfs.cs and firefox.o1.cs
\textcolor {cyan} t = -0.024, df = 83.899, p-value = 0.9809
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -669.7014  653.7479
sample estimates:
mean of x mean of y
 2203.512  2211.488
```

Since the P value is greater than pre specified significance level of 0.05, the null hypothesis is accepted, which means that CFS and O(1) has no difference of context switch with respect to Firefox process. The next figure 4.48 shows the interrupts.

The graph indicates that O(1) has more interrupts than CFS. Also can be seen from the figure, the interrupt of O(1) starts to increase sharply at the beginning of the test, when 1 Firefox is started, and begins to fall at the time stamp of 8, when 2 Firefox process has started. So far, this is the same with CFS. After that, the interrupt of O(1) keeps the same, while this number of CFS still goes up and down.

The next figure 4.49 shows the user time.

Quite obviously that O(1) takes more user time than CFS. The O(1) scheduler has special bonus for interactive process. It spends more time wait for user's input from keyboard, mouse or other input devices. However, CFS doesn't have that mechanism for interactive, so in this Firefox process, it has less user time than O(1).

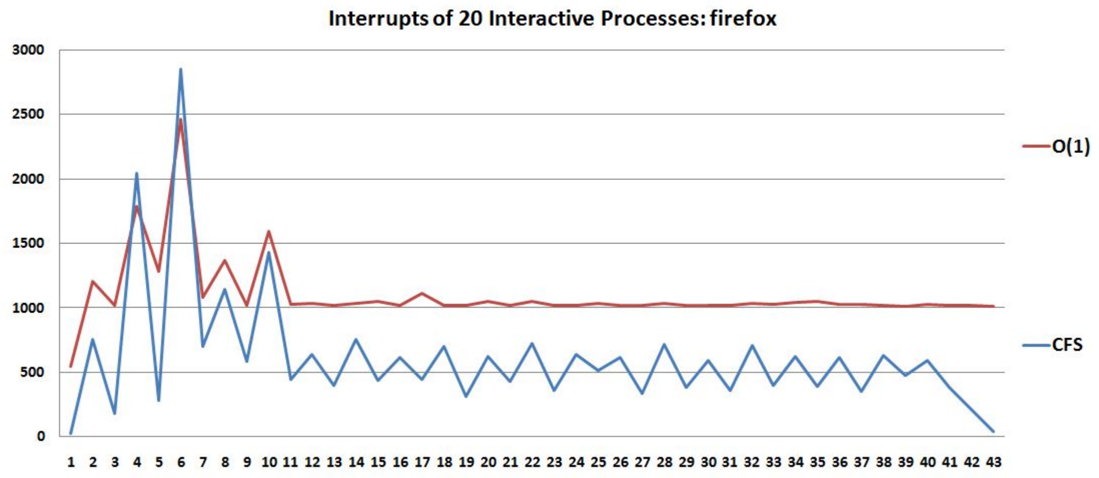


Figure 4.48: Interrupts of 20 Firefox Processes

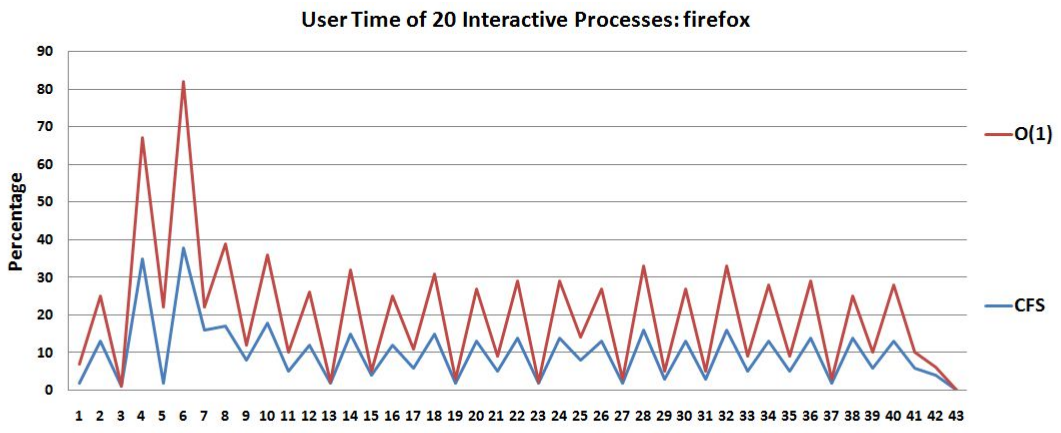


Figure 4.49: User Time of 20 Firefox Processes

Chapter 5

Analysis

This chapter covers the analysis of the whole project, from the methodology to testing results. The analysis part makes a summary of all the results and presents what is behind the results.

5.1 Analysis

CPU-bound Process

As can be seen from figure 4.1 and figure 4.2, for single CPU-bound process CFS consumes more CPU time, both elapsed time and user time. At the same time, as shown in the figure 4.16 and 4.24, O(1) has much greater standard deviation of elapsed time. This applies in every scenario of CPU-bound process testing, which definitely shows that CFS behaviors as it claims to be, completely fair to processes. The reason behind it is that, as presented in the background part, the red-black tree structure switches processes differently from the priority queue structure in O(1). The results from this part verify the theoretical differences between these two schedulers.
But on the other hand, the improvement of the fairness accuracy causes the increase of context switch in CFS. Compared with O(1), this increase is huge, as shown in the figure 4.8, 4.11 and 4.17. As discussed in the background chapter, switching processes costs time, so the context switch should be reduced as much as possible. But, from the result of elapsed time, as shown in the figure 4.12, there is a slight difference of elapsed time between CFS and O(1), which surprisingly suggests that the number of context switch doesn't affect the elapsed time a lot, actual very little. The next figure 5.1 shows that.
In the figure, the blue line is the size of how much CFS context switch is bigger than that of O(1); the red line is the size of how much CFS elapsed time is bigger than that of O(1), and the green line is the size of how much CFS fairness is bigger than that of O(1). The fairness here can be get by the following equation. Actually, there is no such a definition of fairness, but based on its

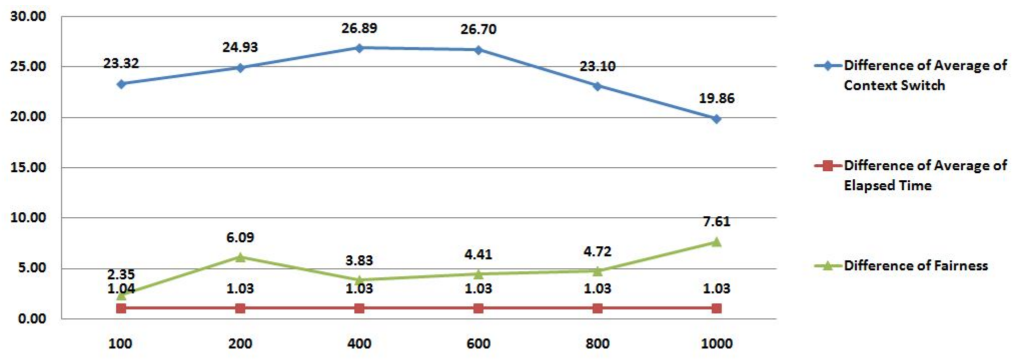


Figure 5.1: Differences of Context Switch, Elapsed Time and Fairness of CPU-bound Process

meaning, I defined it like this way:

$$Fairness = 1/StandardDeviationofElapsedTime \tag{5.1}$$

So, as the equation shows, the less of standard deviation, the greater the fairness. This definition is simple but reasonable, and does shows the fairness of scheduler.
As the figure shows, even though the average context switch of CFS is about 25 times bigger than that of O(1), fairness of CFS is about 5 times more than that of O(1), the average elapsed time of CFS is only about 1.03 bigger than that of O(1). This suggests that CFS has greatly improved its fairness without even reducing too much performance of total elapsed time.

As described in the background chapter, the priority of a process plays different roles in CFS and in O(1). In O(1), the priority directly determines how appropriate a process should be scheduled, as shown in the figure 2.7 and 2.8. In general, in O(1), higher priority means more frequent of scheduling, and in CFS, higher priority means longer processing time. This is the theoretical difference, and the results, as figure 4.18 shows, indicate that lower priority processes with nice values are much nicer to normal processes in CFS than in O(1). To make it clear, the following table is filled with elapsed time of nice, normal and same (which means elapsed time from processes with same priority) to show the differences between CFS and O(1).

The following column chart 5.2 visualizes the average elapsed time in the above table.
As can be seen from the chart, there are two things same for both schedulers. First, for CFS and O(1), nice value and same value are quite close to each other. Second, normal value is quite far away from same and nice value. There are also two differences between CFS and O(1). First, both same and nice values of O(1) are lower than that of CFS respectively. Second, the normal value of O(1) is higher than that of CFS.

Table 5.1: Differences between CFS and O(1) withe Respect to CPU-bound Processes

		CFS			O(1)		
	value	same	nice	normal	same	nice	normal
100	average	466.66	467.47	257.51	448.65	453.19	324.13
	stdv	1.89	0.90	2.23	4.45	0.97	17.53
200	average	928.43	946.06	520.72	898.03	907.92	672.51
	stdv	2.89	1.38	4.83	17.58	2.66	26.47
400	average	1855.39	1863.69	1027.99	1801.24	1816.66	1343.55
	sdv	7.91	3.34	7.013	30.30	4.42	44.37
600	average	2781.09	2789.89	1538.83	2701.76	2716.78	2000.14
	stdv	10.33	6.49	8.25	45.60	6.46	75.56
800	average	3708.04	3721.89	2053.52	3606.03	3622.34	2680.47
	stdv	13.93	7.29	11.80	65.73	8.80	84.90
1000	average	4634.90	4650.92	2565.72	4478.38	4567.29	3368.56
	stdv	19.15	9.36	14.33	145.80	10.93	123.64

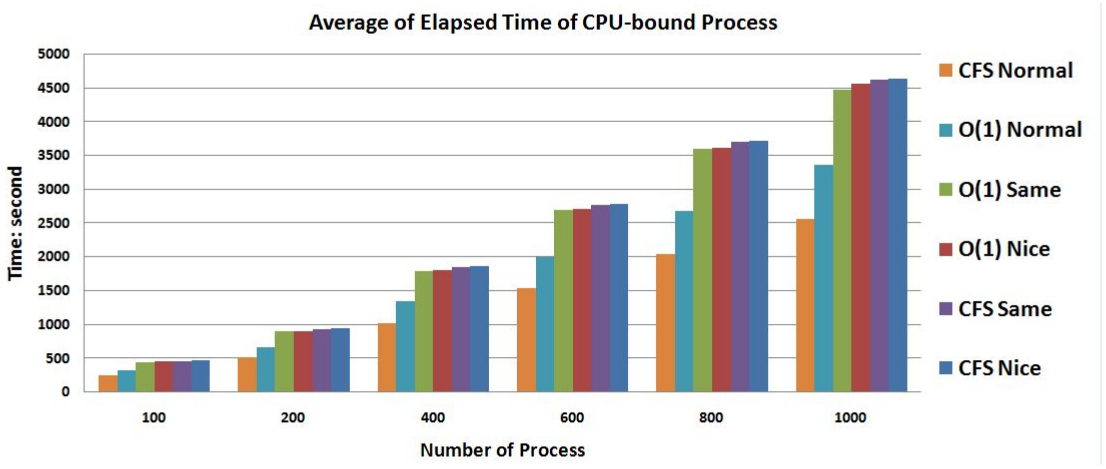


Figure 5.2: average Elapsed Time of CPU-bound Process

So, the results indicate that, when there are nice processes in the system, those nice processes behavior as all the processes are the same, nothing different from the scenario of same priority process, while those normal processes get much benefits from schedulers. Besides, since the normal value of CFS is lower than that of O(1), which indicates that those normal processes benefit more from CFS than from O(1). So, it can be concluded from above data that, longer processing time which is the mechanism of CFS, is better than more frequent scheduling, which is the mechanism of O(1).
The following column chart visualizes the fairness (reciprocal of standard deviation) of elapsed time in the above table.
The chart demonstrates that, CFS nice has highest fairness, followed by O(1)

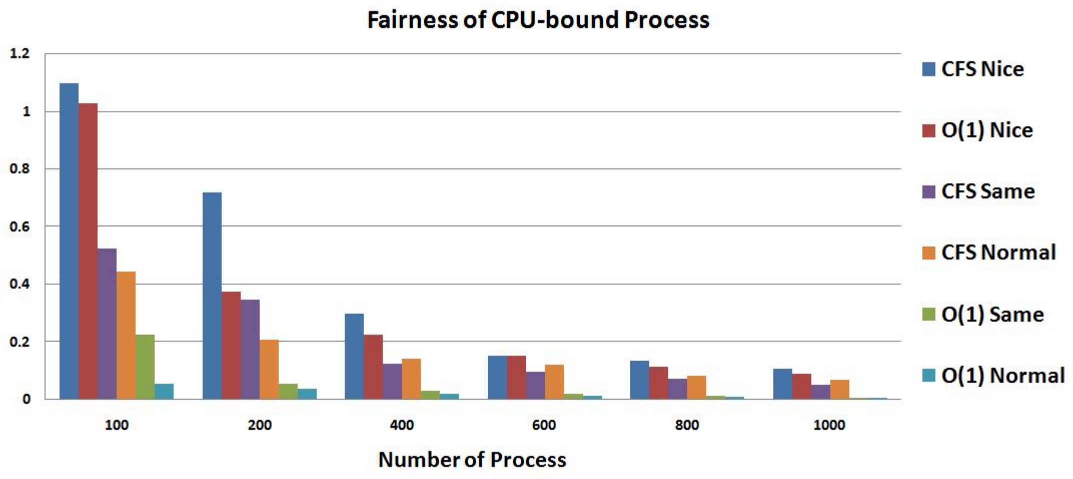


Figure 5.3: Fairness of CPU-bound Process

nice. The next two are CFS normal and CFS same. O(1) same and normal are at the bottom of fairness. The results indicate that, nice processes are more fairly scheduled. The reason is that, since nice processes have lower priority and shorter processing time, which means they are scheduled more times, then they have greater context switch. So the logic is:

$greater\_nice\_value \rightarrow lower\_priority \rightarrow short\_processing\_time \rightarrow scheduled\_more\_times \rightarrow greater\_context\_switch \rightarrow more\_fair$

And the results just verify this! As the figure 5.4 shows, nice processes have greater involuntary context switch with respect to CFS and O(1).

Table 5.2 displays involuntary context switch, and figure 5.4 visualizes the data in the table.

As can be seen from the figure, there are two things same for both schedulers. First, the involuntary context switch doesn't change with the number for process. Second, the order of size of involuntary context switch is nice, same and normal. This applies in both schedulers. The reason is that, as described above, nice processes have shorter processing time and higher involuntary context switch.

Table 5.2: Involuntary Context Switch of CFS and O(1)

		CFS			O(1)		
	Value	same	nice	normal	same	nice	normal
100	average	1573.53	1935.18	1170.04	67.47	105.78	58.52
	stdv	31.87	39.53	20.98	18.36	10.58	8.34
200	average	1559.23	1948.86	1184.18	62.55	107.66	61.33
	stdv	19.47	25.29	22.64	12.26	12.49	9.54
400	average	1565.28	1933.96	1168.13	58.21	109.40	64.42
	stdv	50.38	30.40	15.05	8.25	10.02	10.58
600	average	1563.028	1938.123	1165.813	58.545	103.93	56.89667
	stdv	45.61	81.14	11.57	17.36	6.43	4.31
800	average	1564.02	1928.42	1166.32	67.71	104.74	57.13
	stdv	55.36	40.82	12.60	24.98	15.20	5.35
1000	average	1564.34	1930.40	1165.89	78.76	106.49	60.08
	stdv	53.57	54.40	11.98	25.03	11.03	6.01

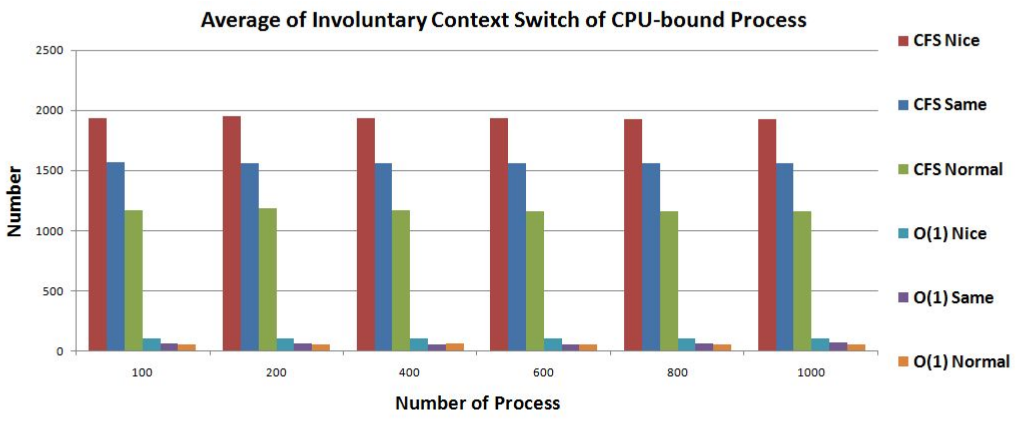


Figure 5.4: Involuntary Context Switch of CPU-bound Process

5.1.1 I/O-bound Process

As figure 4.33 shows, CFS has greater elapsed time than that of O(1) when the number of I/O-bound process increases to 15. And before this point, both schedulers have quite similar values of average elapsed time. To be precise, the following table is filled with elapsed time of I/O-bound process. As described

Table 5.3: Elapsed Time of I/O-bound Process

O(1)	average	23.37	52.43	112.84	182.05	354.42	668.31	1114.09	1497.05
	sdv	4.03	5.54	18.01	13.30	51.67	65.33	126.3	250.46
CFS	average	9.69	43.12	115.27	168.63	351.92	658.68	1289.88	1696.04
	sdv	0.08	7.92	10.84	11.07	57.33	67.80	224.32	231.93

in the background chapter, for I/O-bound processes, they don't consume lots of CPU, but spend much time on waiting for I/O requests. So they run very frequently since they don't need long timeslice from schedulers.
As shown in the table, when there is single I/O-bound process, the elapsed time of O(1) is much greater than that of CFS, actually about twice as much as that of CFS. This difference decreases when the number of I/O-bound process increases until there are 10 I/O-bound processes. After that, elapsed time of CFS becomes greater than that of O(1). As for the standard deviation of elapsed time, as shown in the figure 4.34, CFS has higher standard deviation than O(1). This is quite different from the scenario of CPU-bound process, where CFS has much lower standard deviation of elapsed time, as shown in the figure 4.16.
Also as the figure 4.37 shows, when there are more than 8 I/O-bound processes in the system, O(1) has no CPU idle time, while in CFS, there is still about 25% CPU idle time. One aspect of the performance of scheduler as discussed in the section 3.2, is the efficiency, which means how efficient scheduler has made CPU be. This indicates that CFS doesn't push CPU to work at most when it comes with I/O-bound process.
The results indicate that O(1) shows better performance handling many I/O-bound processes, even though CFS succeeds in the situation of single I/O-bound process.

5.1.2 Mixed Processes

CPU-bound Process

In this part, elapsed time of mixed processes is compared with pure I/O-bound processes and pure CPU-bound processes. As shown in the figure 4.39, the elapsed time of 50 CPU-bound processes varies a lot between pure CPU-bound processes and mixed processes. First, the elapsed time increases for both schedulers. This is obvious since there are more 10 I/O-bound processes. Second, in O(1) scheduler, CPU-bound processes in the mixed situation is much affected by I/O-bound processes. As can be seen that, O(1)_Mixed curve has much stronger volatility than that O(1)_Pure curve, which means O(1) decreases its performance of fairness in the mixed processes scenario. However, this doesn't apply for CFS, which still shows high performance of fairness to processes, no matter whatever kind of they are. Meanwhile, for involuntary context switch, there is also big difference between O(1)_Mixed and O(1)_Pure. As the figure 4.40 shows. While for CFS, it is not as fluctuant as O(1), which again shows that CFS higher performance of fairness.

I/O-bound Process

As shown in the figure 4.41, elapsed time of 10 I/O-bound processes both in mixed scenario and in pure scenario are quite similar to each other. This applies for both schedulers. As shown in the following table:
It is quite surprising that for both schedulers, the elapsed time in mixed

Table 5.4: Elapsed Time of 10 I/O-bound Processes

	I/O-bound Process	Mixed Process
CFS	658.6824	636.0812
O(1)	668.3195	584.2723

scenario is less than that in the pure I/O-bound scenario. For CFS, it decreases from 658.68 to 636.08, and for O(1), it decreases from 68.32 to 584.27. In the mixed scenario, there are 60 processes in total, 10 I/O-bound processes, and 50 CPU-bound processes, while in the pure I/O-bound scenario; there are only 10 I/O-bound processes. As the table shows, the elapsed time just decreases for both schedulers, though there are 50 CPU-bound processes more.
This is really unexpected result before this project. Generally speaking, more processes means more total elapsed time. Before trying to dig deep to scheduler and system to find reasons for the results, a t.test is needed to make sure that this result is not caused by chance.
The following table shows the t.test result of O(1).

```
      Welch Two Sample t-test

data:  o1_pure_io_elapsed_time and o1_mixed_io_elapsed_time
\textcolor[cyan]t = 7.1836, df = 173.418, p-value = 1.924e-11
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  60.9546 107.1398
sample estimates:
```

```
mean of x mean of y
 668.3195  584.2723
```

As the p-value is much less than 0.05, so the null hypothesis, which is that these two samples are the same, is rejected, which means from statistical point of view, the results have significant meaning. The same test for CFS, as shown below:

```
        Welch Two Sample t-test

data:  cfs_pure_io_elapsed_time and cfs_mixed_io_elapsed_time
t = 2.8436, df = 163.954, p-value = 0.005029
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  6.907453 38.294947
sample estimates:
mean of x mean of y
 658.6824  636.0812
```

Again, the t.test verifies that the observed differences from tests were not produced by chance.

The reason, could be that, when the system is busier (mixed case), there is actually less resource contention because bonnie processes' I/O requests get spread out in time more, and don't happen at the same time. When there are only 10 I/O-bound processes, they are all effectively waiting for the same thing and "stepping on each other's toes". However, reasons behind this probably are not clear enough. But at least the results indicate that the situation of 10 simultaneous bonnie processes must be very inefficient.

Interactive Processes

Editing Porcesses

As shown in the figure 4.43, the elapsed time of editing process of CFS is again greater than that of O(1), just like in all other scenarios. However, what's different this time is that the involuntary context switch of O(1) is much greater than that of CFS just as the figure 4.44 shows. The following table presents all the numbers in those two figures: As discussed in the

Table 5.5: Editing Processes Results of O(1) and CFS

CFS	elapsed time	2.81	5.68	11.41	17.14	22.88	28.61
	cs	77.75	156.5	306.66	459.16	611.48	772.3
O(1)	elapsed time	2.20	4.38	8.77	13.17	17.58	21.96
	cs	227.3	449.67	896.66	1345.01	1791.88	2239.52

background chapter, there is one theoretical difference between CFS and

O(1) toward interactive process, which is that O(1) has extra bonus for interactive process.
In O(1), when interactive processes run out of time slice, they don't go to a lower priority list but still in the active queue. In other words, they have higher static priority. After all the interactive processes are finished, the active queue then is swapped with expired queue. So, for those interactive processes, they are kept switched from one to another in the active queue until all of them are finished. This means, they should have less elapsed time and greater involuntary context switch. On the other hand, in CFS, all the processes are treated as fairly as possible, there is no such mechanism to give convenience to interactive processes. And the results just verify the theoretical differences between CFS and O(1).

Firefox Processes

The difference between CFS and O(1) is not as significant as in other tests. As shown in the figure 4.47, there is no difference between CFS and O(1) with respect to involuntary context switch. The value keeps going up and down in the figure, because there is time interval of 4 seconds between every two Firefox process. However, O(1) has greater value of user time than CFS as shown in the figure 4.49. User time is the amount of CPU time spent in user-mode code (outside the kernel) within the process, the real time spent on executing the process. In this case, CFS shows slightly better performance than O(1).

Chapter 6

Discussion and Conclusion

This chapter covers the discussion and conclusion of this thesis. The discussion part presents the evaluations of methodology and result, and also suggests the future work. The conclusion part gives a brief conclusion of this thesis.

6.1 Discussion

6.1.1 Evaluation of Project

Methodology

The purpose of this thesis is to compare two different schedulers. The comparison consists two parts, the theoretical comparison and the performance comparison. The first part describes the mechanism, and the second part is to verify what covers in the first part, and also offers a first-hand reference to people who are going to choose different schedulers.
To compare the performance, different testing plans are designed. The testing plan in this thesis covers most typical kinds of processes, and all the other situations can be considered as a combination of different processes in this thesis. For each plan, benchmark tool, system utility are used to saturate the system and record data.
In order to eliminate the experiment errors and make the data reliable, for all the tests, they are repeated for 10 times, then average and standard deviation are presented.

Implementation Problems

O(1) and CFS are in different versions of Red Hat. In order to make controlled variables same to both schedulers, they are installed in two separate disks which share same processors and other physical settings.
The testing machine is using a dynamic IP address from a *dhcp* server

which causes the problem of ssh connection to the machine. There are two solutions for this problem. The first one is to set up a crontab job, which runs a network restart command. So the machine will get same IP every time. The second solution is to use a tool called *ddclient* which updates dynamic DNS entries. One can get a free domain name from either `www.dyn.com` or `www.no-ip.com` websites. Once *ddclient* is configured with a static domain name, people could set up ssh connection to the machine through that domain name.
In the Interactive Processes scenario, one problem occurs when running the editing script. Red Hat 5.7 doesn't support Perl 5.10 and Expect. One solution is to upgrade Perl 5.8 to Perl 5.10 by installing source package from CPAN, and to make a symbolic link to the new Perl version. Besides, it is also necessary to exclude Perl from yum to to prevent yum from modifying the new version of Perl.

Result

The results from CPU-bound Processes scenario verifies that the performance difference between CFS and O(1) matches the theoretical difference between this two schedulers. The results from I/O-bound Processes scenario, however, doesn't strongly support that CFS behaviors as it claims to be, because both schedulers have quite similar average value of elapsed time and fairness. Results from Mixed Processes scenario shows surprising results for both schedulers. The results from editing processes scenario, then shows that CFS has greater elapsed time, but has quite similar fairness with O(1) towards editing processes. This results also verify the theoretical design of O(1), which has extra bonus for interactive processes. For Firefox processes, the data do not show great differences between two schedulers.

6.1.2 Recommendations

The result section shows scheduler performance varies a lot according to different workloads. Based on the results from this thesis, some recommendations could be made to system administrators and normal users.
For the workload of CPU-bound processes, CFS is a better choice for system administrators, because CFS makes every process much more fair to each other than O(1). On the other hand, from users' point of view, they only want to finish their jobs as soon as possible, so O(1) is a better choice.
For the workload of I/O-bound processes, both schedulers show similar performances. However, it is recommended to use mixed process, because in the Mixed Processes scenario, those I/O-bound processes are processed more efficiently.
For the workload of interactive processes, especially for editing processes, O(1)

is a better choice for normal users, and CFS is a better choice for system administrators.
In general, the results from this thesis could be a reference for scheduler users, and people should choose different schedulers based on their own needs.

6.1.3 Future Work

This thesis compares two schedulers based on 4 testing scenarios. To make the comparison more complete, it would be beneficial to add more testing plans. For example, the future work could use more mixed scenarios, not only to mix two types of processes, but three and four types of processes. And also in each mixed scenario, the test could be divided into different priorities groups. Furthermore, the testing plan could be operated in the server environment, for example, to compare two schedulers in an e-mail server or a dns server.
As discussed in the background chapter, CFS has lots of features for example group scheduling policy. It would be very interesting to test this feature in real life. Then more users would be involved in the test, which would be a good simulation of real case.
There are some surprising results in this project, for example, as described in the section 5.1.2. The future work could be keeping digging into schedulers and kernel to find proper reasons for that result.
The future work can also include finding tunable parameter for both schedulers, which is also useful for people who are now using them. By modifying some settings and testing them, people could find a way to make the best use of these two schedulers.
Finally, there are some scheduler benchmark tools can be used in the future, for example Latt and Klogger. Latt benchmarks latency and turnaround time under various workloads and Klogger logs kernel events into a special log file, which can be analyzed later.

6.2 Conclusion

The problem statement in the introduction chapter of consists two parts: what is the difference of two schedulers and which has better performance under different workloads. And this thesis project has answered the first question at the beginning by presenting theoretical differences between two schedulers with respect to structures, algorithms and policies of them. Also this project has addressed performance differences between both schedulers by presenting results from different testing scenarios.
For the second question, those two schedulers have their own designs and features. There is no such an absolute answer about which is better or not. Just as described in the recommendation section 6.1.2, they should be used in different ways based on users' purposes.

Bibliography

[1] A closer look at the linux o(1) scheduler. "http://www.hpl.hp.com/research/linux/kernel/o1.php", 12 2011.

[2] what's a kernel. "http://systhread.net/texts/200606kern_def.php", Jun 2006.

[3] The linux kernel: introduction introduction. "http://cs-pub.bu.edu/fac/richwest/cs591_w1/notes/wk1.pdf", 2001.

[4] M. Tim Jones. Inside the linux 2.6 completely fair scheduler. "http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/", Dec 2009.

[5] M. Tim Jones. inside the linux scheduler. "http://www.ibm.com/developerworks/linux/library/l-scheduler/".

[6] Ingo Molnar. Modular scheduler core and completely fair scheduler [cfs]. "http://lwn.net/Articles/230501/".

[7] Tuning the task scheduler. "http://doc.opensuse.org/documentation/html/openSUSE/opensuse-tuning/cha.tuning.taskscheduler.html".

[8] Josh Aas. Understanding the linux 2.6.8.1 cpu scheduler. "urlhttp://www.cs.unm.edu/ eschulte/classes/cs587/data/10.1.1.59.6385.pdf".

[9] M. Tim Jones. Anatomy of linux process management. "http://www.ibm.com/developerworks/linux/library/l-linux-process-management/".

[10] Linux performance and tuning guidelines. "http://www.redbooks.ibm.com/redpapers/pdfs/redp4285.pdf", July 2007.

[11] David A. Rusling. Linux processes. "http://www.science.unitn.it/~fiorella/guidelinux/tlk/node45.html".

[12] Varoon Sahgal. What is the difference between a thread and a process ? "http://www.programmerinterview.com/index.php/operating-systems/thread-vs-process/".

[13] Context switch definition. "http://www.linfo.org/context_switch.html", May 2006.

[14] Joel Wein David Karger, Cli Stein. Scheduling algorithms. "http://people.csail.mit.edu/karger/Papers/scheduling.pdf".

[15] Juergen Haas. sched-setscheduler description. "http://linux.about.com/library/cmd/blcmdl2_sched_setscheduler.htm".

[16] Robert Love. *Linux Kernel Development*. Developer's Library. Sams, 1st edition, Sep 2003.

[17] By Daniel P. Bovet Marco Cesati. *Understanding Linux Kernel*. O'Reilly Media, October 2000.

[18] Red-black and b trees. "http://www.eli.sdsu.edu/courses/fall95/cs660/notes/RedBlackTree/RedBlack.html#RTFToC1", October 1995.

[19] C.S. Wong. Fairness and interactive performance of o(1) and cfs linux kernel schedulers. In *Fairness and Interactive Performance of O(1) and CFS Linux Kernel Schedulers*. Information Technology, 2008. ITSim 2008. International Symposium on, 2008.

[20] Eric Schulte Taylor Groves, Jeff Knockel. Bfs vs. cfs scheduler comparison. "http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf", December 2009.

[21] Avinesh Kumar. Multiprocessing with the completely fair scheduler. "http://www.ibm.com/developerworks/linux/library/l-cfs/", Jan 2008.

[22] Variables in your science fair project. "http://www.sciencebuddies.org/science-fair-projects/project_variables.shtml".