PRIORITY BASED SCHEDULER IN RUST BASED REDOX OPERATING SYSTEM

A Project Report

submitted by

ROHIT SAINI

in partial fulfilment of requirements
for the award of the dual degree of

BACHELOR OF TECHNOLOGY AND MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS

MAY 2018

THESIS CERTIFICATE

This is to certify that the thesis titled **PRIORITY BASED SCHEDULER IN RUST BASED REDOX OPERATING SYSTEM**, submitted by **Rohit Saini**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology and Master of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Chester Rebeiro
Research Guide
Assistant Professor
Dept. of Computer Science and Engg
IIT-Madras, 600 036

Dr. Arun D. Mahindrakar
Research Co-Guide
Associate Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 1st May 2018

ACKNOWLEDGEMENTS

I would like to express my sincerest gratitude to Prof. Chester Rebeiro for guiding me throughout my dual degree project, without whose undying support and trust, none of this would have been possible. His guidance kept me motivated and helped me pursue the area of development of Scheduler in operating system in rust programming language.

ABSTRACT

Since a long time operating system codes were written in C/C++. Writing crictical codes in C/C++ is never a piece of cake and the developers have to endure with the never ending problem like segmentation faults, dangling pointers, null pointers and memory leaks. Thus Rust came along promising all the features of a higher level language without compromising on the performance, safety and concurrency-features of the low level languages. Most of the checks performed by other languages during run-time are performed by Rust during the compile time itself, thus making it blazingly fast. It is often pitched with the three goals : safety, speed and concurrency. Its ability to seem like a higher level language for writing even low-level code like Operating systems, device drivers.

The process scheduling is the activity of the process manager that handles the process of adding, running and removal of processes from the CPU and the selection of another process to execute on CPU. Priority scheduling is a method of scheduling processes based on their priority. In this method, the scheduler chooses the tasks with highest priority to schedule. Assigning priority to every process, and processes with higher priorities are carried out first, whereas tasks with equal priorities are carried out on round robin basis.

The main objective of this paper is to impelement a better approach for CPU scheduling algorithm which improves the performance of CPU in real time operating system. The proposed Priority based Round-Robin CPU Scheduling algorithm is based on the integration of round-robin and priority scheduling algorithm. It retains the advantage of round robin in reducing starvation and also integrates the advantage of priority scheduling. The proposed algorithm also implements the concept of aging by assigning new priorities to the processes. The proposed algorithm improves all the drawbacks of round robin CPU scheduling algorithm.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

IITM	Indian Institute of Technology, Madras
CPU	Central Processing Unit
FCFS	First Come First Serve
RR	Round Robin
PBS	Priority Based Scheduler
OS	Operating System
ML	Meta Language
RTFM	Read the Fine Manual

CHAPTER 1

Introduction

This thesis is concerned with operating system process scheduling algorithm that schedules different processes according to the assigned criteria and defined algorithm, improving overall throughput by removing processes which are unrunnable, like waiting for resource aquisition, and pushing runnable process in processer to continue their execution.

Scheduling is the process by which all processes are given access to system resources (e.g. processor cycles, memory access, internet usage, etc.). The need for a better and better scheduling algorithm always arises from the requirement of fast computer systems to perform multitasking (execute multiple processes at a time) and multiplexing(transmit multiple flows at a time). Scheduling is a fundamental function of operating system that determines which process to schedule in processor, when there are multiple runnable processes ready to execute. Process scheduling is important because it impacts resource utilization and other performance parameters. There exists a number of CPU scheduling algorithms like First Come First Serve, Shortest Job First Scheduling, Round Robin scheduling, Priority Scheduling etc,.

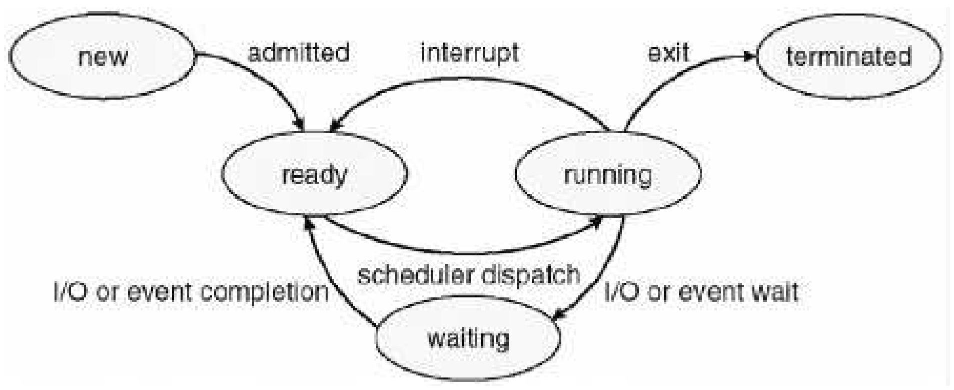


Figure 1.1: A Basic state machine diagram depicting working of scheduler

CHAPTER 2

Process Scheduling

2.1 Objective

A good scheduling algorithm should be fair, efficient, maximize throughput and resource use, minimize response time and overhead, minimize turnaround time, degrade gracefully, enforce priorities, and free from starvation.

There are multiple factors that a process scheduler should attain.

- *Maximize CPU utilization*: Ideally the CPU would be busy 100% of the time, so as to waste 0 CPU cycles. In real time, processor should execute a process for whole of its burst time.
- *Maximize throughput*: A scheduling algorithm should be capable of servicing the maximum number of processes per unit of time.
- *Avoid Starvation*: A process should not wait for unbounded time before or while process service.
- *Minimize Turnaround time*: Time required for a particular process to complete, from submission time to completion.
- *Minimize overhead*: Overhead causes wastage of resources. But when we use system resources effectively, then overall system performance improves greatly.
- *Minimize Waiting time*: Amount of time a process spend in the ready queue waiting for their turn to get on the CPU for execution
- *Minimize Response time*: The time taken in an interactive program from the issuance of a command to the commence of a response to that command.
- *Enforcement of priorities*: if system assigns priorities to processes, the scheduling mechanism should favor the higher-priority processes.
- *Achieve Fairness*: Each process should receive a fair share of CPU resources or execution time.

So in conclusion a good scheduling algorithm for a better system resource allocation and sharing should posses the above characteristics.

2.2 Round Robin scheduling

Round Robin Scheduling is the preemptive FCFS scheduling algorithm. RR scheme solves the problem faced in FCFS scheduler by providing each process a small unit of CPU time known as time quantum/time slice which varies depending on its implementation (from 10 to 100 milliseconds). RR is also one of the simplest scheduling algorithms for processes in an operating system. In fact, after time slice expires, the process is preempted and added to the end of the ready queue. The major advantage of RR is fairness whereby each process gets an equal amount of the CPU time. And its drawback is the average waiting time. The average waiting time can be bad especially when the number of processes is large. For example, let N be the number for processes in ready queue and time slice is T milliseconds, therefore each process gets 1/N of the CPU time. RR offers better performance for jobs with small burst time but context-switching time adds up for large number of jobs.

Disadvantages of Round Robin Scheduler

- *Larger waiting time and Response time*: In RR architecture, waiting time is relatively large as most of the time a process spends in ready queue waiting for its chance to get executed, which inturn increases process completing time. Larger waiting and response time are clearly a drawback in round robin structure as it leads to degradation of system performance.
- *Context Switches*: When the time slice of the task ends and the task is still executing on the processor the scheduler forcibly preempts the tasks on the processor and stores the task context in stack or registers and allocates the processor to the next task available in the ready queue. The action known as context switch. Context switch leads to the significant wastage of time, memory and leads to scheduler overhead.
- *Low throughput*: Throughput is defined as number of process completed per time unit. If round robin is implemented in soft real time systems throughput will be low which leads to severe degradation of system performance. If the number of context switches is low then the throughput will be high. Context switch and throughput are inversely proportional to each other.

Major disadvantage of round robin scheduling is that it doesn't give special preference a task above any other usual ones. This means an urgent request doesn't get handled any faster than other requests in ready queue, which could lead to a catastrophe in real time OS tasks. Priority Bases scheduling architecture helps overcome this problem by assigning higher priorty to more important tasks.

Table 2.1: Table depecting processes and their arrival time in RR

$Process$	$ArrivalTime$	$BurstTime$
P1	0	7
P2	2	4
P3	3	2
P4	9	1

Table 2.2: Table depecting processes arrival time and their execution in RR

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Arrival	P1		P2	P3						P4				
Schedule	P1	P1	P2	P1	P3	P2	P1	P3	P2	P1	P4	P2	P1	P1

Time Slice = 1 units

Average Waiting Time = (7 + 4 + 3 + 3) / 4

= 4.25

Average Response Time = (0 + 0 + 3 + 3) / 4

= 1.5

Average Turnaround Time = (14 + 8 + 2 + 1) / 4

= 7.25

2.3 Priority based preemptive scheduling

In priority based scheduling, scheduler assigns a fixed base priority to every new process, and the scheduler arranges the processes in the ready queue in order of their priority. Scheduler picks the first process in the ready queue i.e., the one with the highest priority and executes it. If there are multiple process with same priority then the tasks are carried out on a round robin basis. Lower priority processes get interrupted by an incoming higher priority processes. Overhead is not minimal, nor is it significant in this case. Waiting time and response time depend on the priority of the process. Higher priority processes have smaller waiting and response times. Deadlines can be easily met by giving higher priority to the earlier deadline processes. Real time OS can have huge advantage with this scheduler, by giving higher priority to urgent tasks.

A major problem with priority scheduling is indefinite blocking or starvation of low priority processes. In priority scheduling some low priority process may keep waiting indefinitely for CPU time. In a heavily loaded system continuous arrival of higher priority processes can prevent low priority process from getting the CPU. One solution to the problem of starvation is aging. Aging is a technique of gradually increasing the priority of process that waits in the ready queue for a longer period of time. Eventually as the process priority increases slowely, it will have a highest priority in the queue and it would be executed. Once it gets its chance to execute, its priority can be brought back to the previous low level.

Let us compare the two scheduling algorithms

Table 2.3: Table depecting processes and their arrival time in PBS

$Process$	$ArrivalTime$	$BurstTime$	$Priority$
P1	0	7	2
P2	2	4	3
P3	3	2	1
P4	9	1	4

Table 2.4: Table depecting processes arrival time and their execution in PBS

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Arrival	P1		P2	P3						P4				
Schedule	P1	P1	P1	P3	P3	P1	P1	P1	P1	P2	P2	P2	P1	P4

Time Slice = 1 units

Average Waiting Time = (2 + 7 + 0 + 5) / 4

= 3.5

Average Response Time = (0 + 7 + 0 + 5) / 4

= 3

Average Turnaround Time = (9 + 11 + 2 + 6) / 4

= 7

As we can see the Average Waiting time and Turnaround time is less in this case improving overall scheduler performance, Avg. response time higher as processes with lower priority have to wait for the higher priority processes to finish.

CHAPTER 3

Rust Programming Language

3.1 About Rust

Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety. Rust is an open-source programming language sponsered by the Mozilla software foundation and a community of volunteers to help the developers create fast and secure applications. Apart from preventing segmentation faults and memory leaks Rust offers zero-cost abstractions, guaranteed memory safety, threads with no data races, pattern matching, type inference and efficient C-bindings with a minimum runtime size. Rust has no in-built garbage collection like those used by Java etc.., but uses a method called Resource Acquisition and initialization and reference counting as in C++. Rust object system consists of structures, implementations and traits. Implementations are similar to functions in classes and are defined with impl keyword whereas Inheritance and Polymorphism are provided by the traits that allow methods to be implemented on structures. Thus the perfect ensemble of high level features and the lower level programming constructs makes it an ideal language to write either user-level applications or kernel level programs with same ease.

3.2 Syntax

The syntax of Rust is similar to C and C++, with blocks of code delimited by curly brackets, and control flow keywords such as if, else, while, and for. Not all C or C++ keywords are implemented, however, and some Rust functionality (such as the use of the keyword match for pattern matching) will be less familiar to programmers coming from these languages. Despite the superficial resemblance to C and C++, the syntax of Rust in a deeper sense is closer to that of the Meta Language family of languages. A function need not end with a return expression: in that case the last expression in the function creates the return value.

3.3 Memory Management

The system is designed to be memory safe, and it does not permit null pointers, dangling pointers, or data races in safe code. Data values can only be initialized through a fixed set of forms, all of which require their inputs to be already initialized. Rust also introduces additional syntax to manage lifetimes, and the compiler reasons about these through its borrow checker.

Rust does not use an automated garbage collection system like those used by Go, Java or .NET Framework. Instead, memory and other resources are managed through resource acquisition is initialization (RAII), with optional reference counting. Rust provides deterministic management of resources, with very low overhead. Rust also favors stack allocation of values and does not perform implicit boxing.

3.4 Ownership in rust

Ownership is Rust's most unique feature, and it enables Rust to make memory safety guarantees without needing a garbage collector. Rust has an ownership system where all values have a unique owner and the scope of the value is the same as the scope of the owner. When the owner goes out of scope, the value will be dropped. Values can be passed by immutable reference using &T, by mutable reference using &mut T or by value using T. At all times, there can either be multiple immutable references or one mutable reference. The Rust compiler enforces these rules at compile time and also checks that all references are valid.

3.5 Cargo: Rust Package Manager

Cargo is a tool that allows Rust projects to declare their various dependencies and ensure that you'll always get a repeatable build. Cargo helps in building the code, downloading all the dependencies that code needs, and building those dependencies.

CHAPTER 4

Redox Operating System

4.1 About Redox

Redox is a Unix-like Operating System written in Rust, aiming to bring the innovations of Rust to a modern microkernel and full set of applications, a language with focus on safety and high performance. Redox, following the microkernel design, aims to be secure, usable, and free. Redox is inspired by previous kernels and operating systems, such as SeL4, MINIX, Plan 9, and BSD. Redox is not just a kernel, it's a full-featured Operating System, providing packages (memory allocator, file system, display manager, core utilities, etc.) that together make up a functional and convenient operating system. You can loosely think of it as the GNU or BSD ecosystem, but in a memory safe language and with modern technology. See this list for overview of the ecosystem. Click to visit Redox: redox-os.org

4.2 How redox OS looks

Attached some images of redox deployed on Lenovo Thinkpad

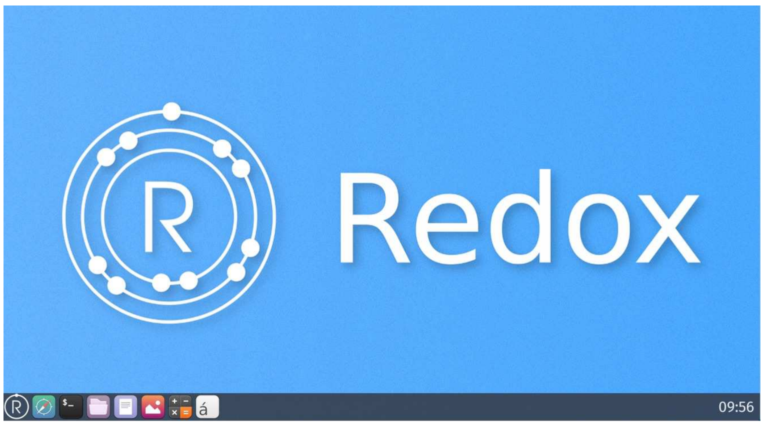
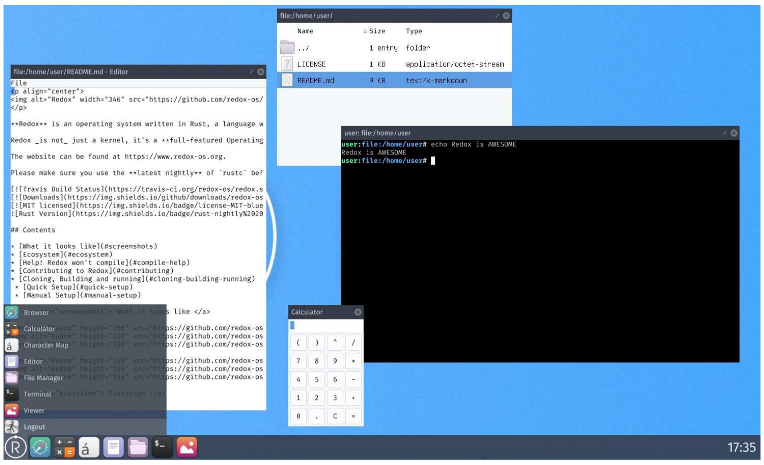


Figure 4.1: Screenshot of Redox-os desktop

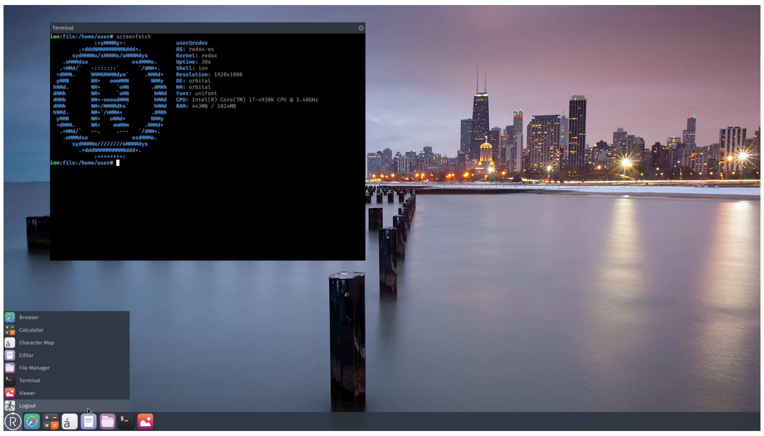


```
File
<p align="center">
<img alt="Redox" width="346" src="https://github.com/redox-os/
</p>

**Redox** is an operating system written in Rust, a language w

Redox _is not_ just a kernel, it's a **full-featured Operating

The website can be found at https://www.redox-os.org.

Please make sure you use the **latest nightly** of `rustc` bef

[![Travis Build Status](https://travis-ci.org/redox-os/redox.s
[![Downloads](https://img.shields.io/github/downloads/redox-os
[![MIT licensed](https://img.shields.io/badge/license-MIT-blue
![Rust Version](https://img.shields.io/badge/rust-nightly%2020

## Contents

* [What it looks like](#screenshots)
* [Ecosystem](#ecosystem)
* [Help! Redox won't compile](#compile-help)
* [Contributing to Redox](#contributing)
* [Cloning, Building and running](#cloning-building-running)
 * [Quick Setup](#quick-setup)
 * [Manual Setup](#manual-setup)
```

```
Name              ↓Size     Type
../               1 entry   folder
LICENSE           1 KB      application/octet-stream
README.md         9 KB      text/x-markdown
```

```
user:file:/home/user# echo Redox is AWESOME
Redox is AWESOME
user:file:/home/user# 
```

Figure 4.2: Screenshot of Redox-os start menu and some programs



```
ion:file:/home/user# screenfetch
              :+yMMMMy+:                user@redox
         .+dddNMMMMMMMMMNddd+.          OS: redox-os
       sydMMMMo/sMMMMs/oMMMMdys         Kernel: redox
    .oMMMdso               osdMMMo.     Uptime: 30s
   .+MMd/`   -:::::::`      `/dMM+.     Shell: ion
  +dMMN.     NMMNNNMMdyo`     .NMMd+    Resolution: 1920x1080
  yMMN       NM+   oomMMN       NMMy    DE: orbital
 hNMd.       NM+     `oMN       .dMNh   WM: orbital
 dMMh        NM+     `oMN        hMMd   Font: unifont
 dMMh        NM+-ooodMMN         hMMd   CPU: Intel(R) Core(TM) i7-4930K CPU @ 3.40GHz
 dMMh        NM+/MMMMdhs`        hMMd   RAM: 443MB / 1024MB
 hNMd.       NM+`/mMMm+         .dMNh
  yMMN       NM+   oNMd+        NMMy
  +dMMN.     NM+    omMMm     .NMMd+
   .+MMd/`   --.     .---    `/dMM+.
    .oMMMdso               osdMMMo.
       sydMMMMo////////oMMMMdys
         .+dddNMMMMMMMMMNddd+.
              :+++++++++:
ion:file:/home/user# 
```

Figure 4.3: Screenshot Redox-os with terminal

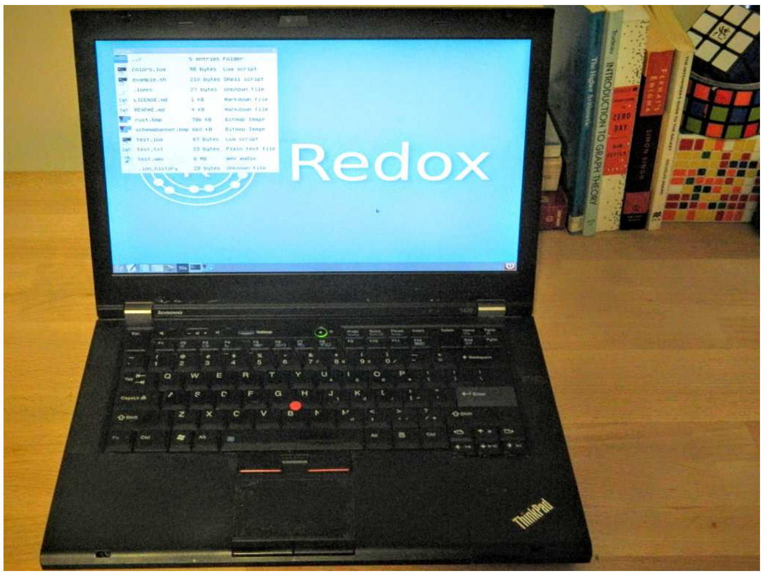


Figure 4.4: Snapshot of Redox-os deployed on Lenovo thinkpad

4.3 Setup and Compiling Redox

1. Make sure you have a Redox toolchain installed (**x86_64-unknown-redox-gcc**). You can install it from .deb packages (**https://static.redox-os.org/toolchain/apt/**) or build redox/libc manually.

2. Run command
   $ rustup update.

3. Run command
   $ make clean pull.

4. Make sure you have the latest version of Rust nightly! (rustup.rs is recommended for managing Rust versions, install it form the link: **https://www.rustup.rs**. If you already have rustup installed, then run commands *$rustup*).

5. Update packages GNU Make, NASM and QEMU/VirtualBox.

6. Pull the upstream master branch, use command below
   $ git remote add upstream git@github.com:redox-os/redox.git;
   $ git pull upstream master

7. Update submodules
   $ git submodule update –recursive –init

8. Finally build and Launch system using,
   build: *$ make all*
   Launch: *$ make qemu*

CHAPTER 5

Redox Scheduler

5.1 Redox Default Scheduler

Redox implements Round Robin scheduler as default scheduler, giving redox os the ability to schedule the processes and be deployed on actual hardware. But as the work load increases the overall waiting time and turnaround time increases, reducing the overall performance of the scheduler. Thus a need for better scheduler is scheduler was arising and priority based scheduler comes to the rescue giving the required improvements needed.

5.2 Implementing Priority based scheduler in Redox

Below are the code snippets of implementation of Priority based scheduling

```
pub struct ContextList {
     map: BTreeMap<ContextId, Arc<RwLock<Context>>>,
     next_id: usize,
     priority_list: Set<(priority,ContextId)>,
     def_priority: u8,
     priority_inc: u8
}
```

priority_list: A auto sorted Set data structure stores a tuple(priority,ContextId) stores *id* and *priority* of a process.

def_priority: Default priority value assigned to new processes.

priority_inc: increment priority of remaining processes to avoid starvation

The below code block uses priority scheduling scheme to find a new process to schedule on processor.

```
loop {
    // Get element from list sorted by priority
    let next_id = contexts.get_context_ptlist(to_index);
    let context_lock = contexts.get(next_id);
    let mut context = context_lock.write();

    // Check if process is runnable
    if runnable(&mut context, cpu_id) {
        to_ptr = context.deref_mut() as *mut Context;
        if (&mut *to_ptr).ksig.is_none() {
            to_sig = context.pending.pop_front();
        }

        // Increments the priority
        contexts_mut.inc_priority(to_index);

        // move the process back to ready
        contexts_mut.reorder(to_index);
        break;
    }
    to_index += 1;
}
```

The use of loop is to find the first runnable process in the queue and schedule it,after scheduling we increment the priority of the other processes by *priority_inc* and move the current process to the queue in appropriate place, such that the whole list is sorted.

5.3 Performance of new scheduler

Priority based scheduler performs better than Round robin in improving waiting time and also overall job completing time.

The Context switch time is increased by a slightly due to increased task of finding highest priority task from the ready queue and increasing the priority of low priority tasks to referain them for starving.

References

1. **Y.A. Adekunle, Z.O. Ogunwobi, A. Sarumi Jerry, B.T. Efuwape, Seun Ebiesuwa, and Jean-Paul Ainam.** A Comparative Study of Scheduling Algorithms for Multiprogramming in Real-Time Systems *International Journal of Innovation and Scientific Research*, Vol. 12 No.1, 181, (2014).

2. **Ishwari Singh Rajput and Deepa Gupta.** A Priority based Round Robin CPU Scheduling Algorithm for Real Time Systems. *International Journal of Innovations in Engineering and Technology (IJIET)*, Vol. 1 No.1, (2012).

3. William Stallings, Operating Systems Internal and Design Principles, 5th Edition , 2006.