CSE 331 TERM PROJECT

by

Yahya KOYUNCU

İrem Tuğba SAĞSÖZ

Begüm YILDIRIM

CSE 331 Operating Systems Design

Term Project Report

Yeditepe University

Faculty of Engineering

Department of Computer Engineering

Spring 2023

ABSTRACT

In this project, we replaced the default scheduler algorithm of Linux with our custom Stride Scheduler algorithm by introducing a new system call. Through the use of a flag sent to this system call, we were able to switch between the two scheduler algorithms. The primary goal was to compare the default scheduler and the Stride Scheduler algorithm.

We conducted CPU usage measurements for both scheduler usages across specific test cases. Subsequently, we compared the obtained data with theoretical values. At this stage, Mean Squared Error (MSE) was calculated to assess the disparity between the observed and expected results. This process allowed us to gain insights into the effectiveness of each algorithm and highlighted the superior aspects of both algorithms.

It became apparent that the effectiveness of an algorithm is not solely dependent on CPU utilization. We concluded that both algorithms have their own strengths, emphasizing that the efficacy of an algorithm extends beyond just CPU utilization.

TABLE OF CONTENTS

1. INTRODUCTION

In today's computing landscape, optimizing the performance of operating systems and efficiently managing workloads is of paramount importance. In this context, scheduler algorithms constitute fundamental building blocks of operating systems. This project thoroughly examines our Stride algorithm, specifically designed for comparison with Linux's default scheduler algorithm.

In this project, we compare Linux's default scheduler algorithm with our own stride algorithm according to certain criteria. First of all, we examine in detail the basic features of Completely Fair Scheduler (CFS), Linux's default scheduler algorithm, and the Stride Scheduler algorithms we developed, and determine the advantages and limitations of each algorithm. Then we use specifically designed test cases to test and compare the performance of these algorithms in specific use cases.

This project consists of two key phases that play an important role in complex operating systems. The first stage includes an infrastructure created using system calls in order to establish effective communication between the kernel and user space. At this stage, a system call that performs a simple addition operation is designed to evaluate a basic functionality. This formed the basis of the project and provided a basis for moving into the second phase. After the completion of the first phase, the second phase of the project started. At this stage, a special system call is created that will serve as a flag to be used when communicating between the kernel and user space. This system call provides a more specific and customized infrastructure for the general purpose of the project. In the second stage, the main focus of the project is the "stride algorithm", a unique scheduling algorithm. This algorithm aims to provide effective resource management by organizing and prioritizing workloads. This created algorithm was integrated into the kernel in accordance with the purpose of the project and made available through system calls.

In the design and implementation section of this document, both Linux's default scheduler algorithm and the stride algorithm we designed are mentioned in detail. In the Test and Results section, there are the tests we performed using the two algorithms and the comparison of the results we obtained from these tests, and the error rates we encountered when compared to the default scheduler. The Conclusion section includes the evaluation of the results we obtained as a result of the tests.

2. DESIGN and IMPLEMENTATION

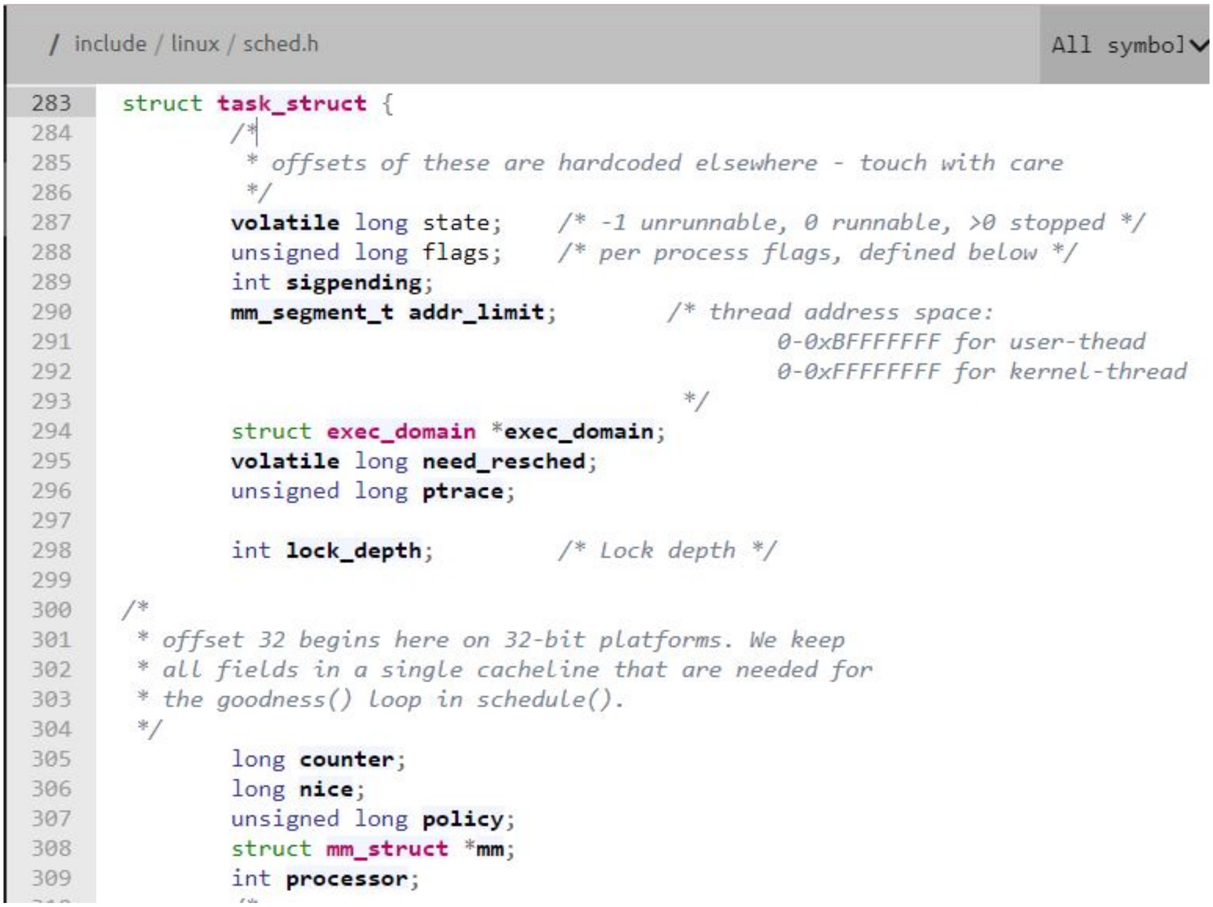
Process management is a crucial aspect of operating systems, and it involves various components, one of which is the scheduler. The scheduler is responsible for determining which process gets to use the CPU and for how long. Operating systems use various scheduling algorithms to manage complex and diverse tasks effectively. Linux supports 3 scheduling policies: SCHED_FIFO, SCHED_RR, and SCHED_OTHER. SCHED_OTHER is the default universal time-sharing scheduler policy used by most processes; 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.

Within the Linux default scheduler, each process is assigned a value known as "nice." Nice values determine the dynamic priority of the respective process. Newly created processes have a default nice value of 0. Negative nice values indicate higher priority, while positive values result in lower priority. The process priority is calculated as 20 minus the nice value. If a process experiences I/O bursts, the system decreases the nice value to boost priority. However, if a process solely undergoes CPU bursts, the nice value remains unchanged.

Time slices allocated based on process priority are directly proportional. The system manages these allocations using a counter variable assigned to each process. This counter variable indicates the number of time slices allocated to a process. When these calculated counters are exhausted, the system recalculates the counters for each process, termed an “epoch."

In summary, the Linux scheduling system dynamically adjusts process priorities using nice values, allocates time slices based on priorities, and maintains fairness through periodic recalculations during epochs. This dual approach of time-sharing and real-time algorithms ensures effective multitasking and responsiveness in diverse computing environments.

In the scheduler (/include/linux/sched.h) there is a structure named **task_struct** and it holds the information of a process. It keeps some values like nice, counter etc. It uses this information to select the process to run.

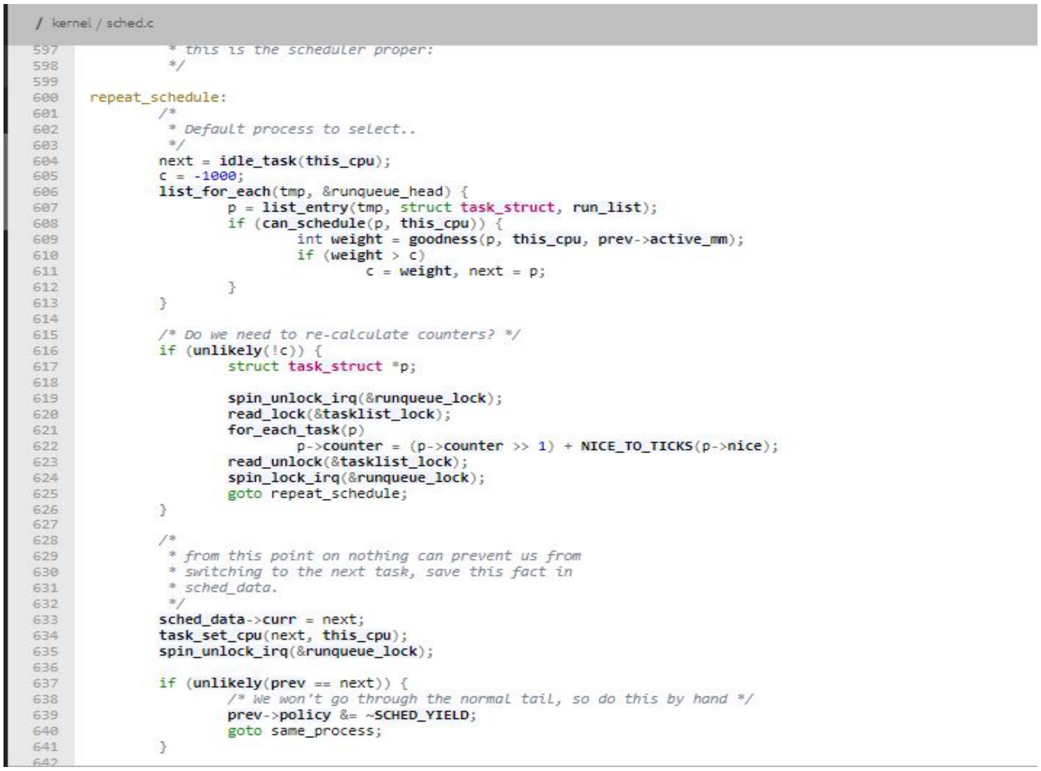


```
struct task_struct {
        /*
         * offsets of these are hardcoded elsewhere - touch with care
         */
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
        unsigned long flags;    /* per process flags, defined below */
        int sigpending;
        mm_segment_t addr_limit;        /* thread address space:
                                                0-0xBFFFFFFF for user-thead
                                                0-0xFFFFFFFF for kernel-thread
                                         */
        struct exec_domain *exec_domain;
        volatile long need_resched;
        unsigned long ptrace;

        int lock_depth;         /* Lock depth */

/*
 * offset 32 begins here on 32-bit platforms. We keep
 * all fields in a single cacheline that are needed for
 * the goodness() loop in schedule().
 */
        long counter;
        long nice;
        unsigned long policy;
        struct mm_struct *mm;
        int processor;
```

Linux makes the process selection with the schedule() function, which is invoked at regular 10ms intervals to identify the process with the highest priority. Within this function, a for loop navigates through processes in the ready state, assessing whether they are eligible for scheduling. Simultaneously, a "goodness value" is computed based on each process's priority, contributing to the decision-making process.

2.1. LINUX DEFAULT SCHEDULER(SCHED_OTHER)

In the Linux process scheduling mechanism, the determination of the best-suited candidate process is driven by the computation of a "goodness value" within the schedule() function. This value serves as a critical metric for selecting processes for execution. If the calculated goodness value reaches -1000, the process is deemed ineligible for selection. A goodness value of 0 indicates that the process has exhausted its allocated time slice. In cases where the goodness value falls within the range of 0 to 1000, the process is eligible to run, and its value is calculated using the formula (20-nice) + counter. Notably, if the goodness value exceeds or equals 1000, signifying a real-time process, it is granted higher priority for execution. This process of evaluating and selecting the optimal candidate is reiterated at regular 10ms intervals, ensuring dynamic and efficient management of processes within the Linux system.



```
	 * this is the scheduler proper:
	 */

repeat_schedule:
	/*
	 * Default process to select..
	 */
	next = idle_task(this_cpu);
	c = -1000;
	list_for_each(tmp, &runqueue_head) {
		p = list_entry(tmp, struct task_struct, run_list);
		if (can_schedule(p, this_cpu)) {
			int weight = goodness(p, this_cpu, prev->active_mm);
			if (weight > c)
				c = weight, next = p;
		}
	}

	/* Do we need to re-calculate counters? */
	if (unlikely(!c)) {
		struct task_struct *p;

		spin_unlock_irq(&runqueue_lock);
		read_lock(&tasklist_lock);
		for_each_task(p)
			p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
		read_unlock(&tasklist_lock);
		spin_lock_irq(&runqueue_lock);
		goto repeat_schedule;
	}

	/*
	 * from this point on nothing can prevent us from
	 * switching to the next task, save this fact in
	 * sched_data.
	 */
	sched_data->curr = next;
	task_set_cpu(next, this_cpu);
	spin_unlock_irq(&runqueue_lock);

	if (unlikely(prev == next)) {
		/* We won't go through the normal tail, so do this by hand */
		prev->policy &= ~SCHED_YIELD;
		goto same_process;
	}
```

2.2. STRIDE SCHEDULER

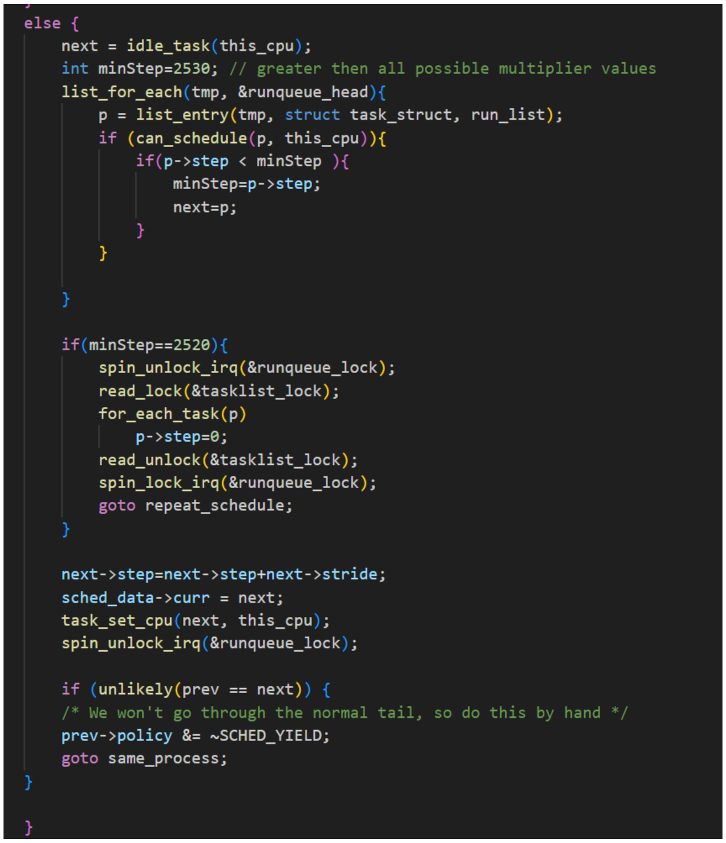
In Stride scheduling algorithm, a cpu utilization of a process is randomly created. . In other words, unlike the sched_other algorithm, the CPU utilization of a process does not change according to certain nice values. Therefore, there is no fair time sharing like in sched_other. In the Stride scheduling algorithm, OS creates a random ticket value between 1 and 10 for each process and basically this number decides how many times that process is going to be executed in an epoch. After each epoch, another ticket value for each process is assigned. The time intervals for the execute time is still 10ms as it is in SCHED_OTHER, which means higher ticket value has more number of 10 milliseconds.

2.3. IMPLEMENTATION

In the Stride scheduling algorithm implementation, we create a random ticket value in the fork.c file and this value determines the CPU utilization of that process. In other words, unlike the sched_other algorithm, the CPU utilization of a process does not change according to certain nice values. Therefore, there is no fair time sharing like in sched_other.

In the Stride scheduling algorithm, first a random ticket value between 1 and 10 is assigned to the process, then we calculate the stride value by dividing the multiplier value, which does not change and is 2520, by this ticket value. Another variable is the step variable, which keeps the step numbers of each process. This variable determines how many times processes will run in a epoch. The process with the smallest step variable is run at that step and the stride value of the process is added to this step value. In this way, after the processes run a certain number of times in parallel with the ticket numbers, the epoch ends when the step variable of each process reaches the multiplier value, that is, 2520, the step values of the processes are changed to 0 again and the next epoch is started.

We use a system call and call it in main function to switch from the default scheduler to stride scheduler. This system call decides which scheduler the system will switch to by looking at the value of the variable in it. If this variable is 1, the OS works with default scheduler algorithm, if it is 0, it works with Stride algorithm.



```
else {
    next = idle_task(this_cpu);
    int minStep=2530; // greater then all possible multiplier values
    list_for_each(tmp, &runqueue_head){
        p = list_entry(tmp, struct task_struct, run_list);
        if (can_schedule(p, this_cpu)){
            if(p->step < minStep ){
                minStep=p->step;
                next=p;
            }
        }

    }

    if(minStep==2520){
        spin_unlock_irq(&runqueue_lock);
        read_lock(&tasklist_lock);
        for_each_task(p)
            p->step=0;
        read_unlock(&tasklist_lock);
        spin_lock_irq(&runqueue_lock);
        goto repeat_schedule;
    }

    next->step=next->step+next->stride;
    sched_data->curr = next;
    task_set_cpu(next, this_cpu);
    spin_unlock_irq(&runqueue_lock);

    if (unlikely(prev == next)) {
    /* We won't go through the normal tail, so do this by hand */
    prev->policy &= ~SCHED_YIELD;
    goto same_process;
}

}
```

3. TESTS AND RESULTS

Performance testing is conducted to assess the speed and efficiency of our code. By evaluating the performance metrics, we aim to identify potential bottlenecks and address any performance issues that may arise. This process enables us to make essential adjustments, optimizing the code for improved efficiency and enhancing the overall user experience.

Performance tests not only measure the code's speed and efficiency but also assess its functionality across diverse hardware and software environments.

Consequently, by conducting performance tests, we consistently advance and optimize the code we create.

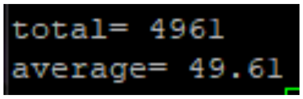
Below are the mean square errors (MSE) calculated from samples obtained by the default and stride algorithms. By examining the difference in MSEs between these two algorithms, we will gain insights into the performance of our code and the effectiveness of the stride algorithm. Here are the results we obtained.

3.1. DEFAULT SCHEDULING ALGORITHM

3.1.1. AVERAGE CPU UTILIZATION

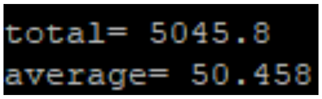
An average consists of 100 samples and a test case consists of 10 averages. Below, you can see the each processes one screenshot out of ten executions.

TEST CASE -1:



```
total= 4961
average= 49.61
```

process1 of test case 1



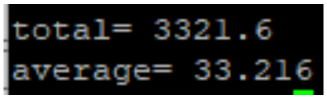
```
total= 5045.8
average= 50.458
```

process2 of test case 1

Average MSE for process1- test1 with default value 50.00 is: 0.133336

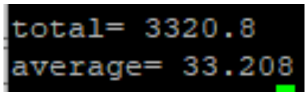
Average MSE for process2- test1 with default value 50.00 is: 0.116962

TEST CASE -2:



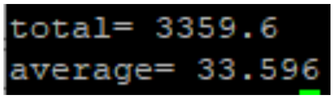
```
total= 3321.6
average= 33.216
```

process1 of test case 2



```
total= 3320.8
average= 33.208
```

process2 of test case 2



```
total= 3359.6
average= 33.596
```

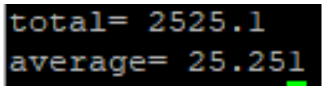
process3 of test case 2

Average MSE for process1- test2 with default value 33.33 is: 0.042229

Average MSE for process2- test2 with default value 33.33 is: 0.052931

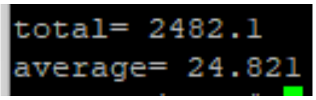
Average MSE for process3- test2 with default value 33.33 is: 0.055578

TEST CASE -3:



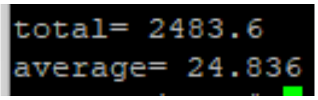
```
total= 2525.1
average= 25.251
```

process1 of test case 3



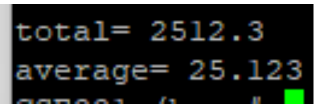
```
total= 2482.1
average= 24.821
```

process2 of test case 3



```
total= 2483.6
average= 24.836
```

process3 of test case 3



```
total= 2512.3
average= 25.123
```

Process4 of test case 3

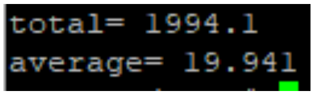
Average MSE for process1- test3 with default value 25.00 is: 0.043055

Average MSE for process2- test3 with default value 25.00 is: 0.031211

Average MSE for process3- test3 with default value 25.00 is: 0.074557

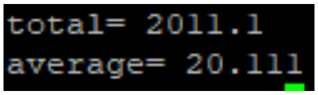
Average MSE for process4- test3 with default value 25.00 is: 0.060517

TEST CASE -4:



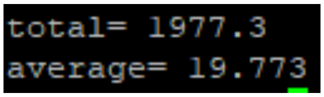
```
total= 1994.1
average= 19.941
```

process1 of test case 4



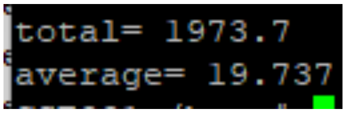
```
total= 2011.1
average= 20.111
```

process2 of test case 4



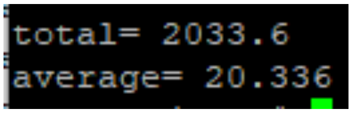
```
total= 1977.3
average= 19.773
```

process3 of test case 4



```
total= 1973.7
average= 19.737
```

Process4 of test case 4



```
total= 2033.6
average= 20.336
```

process5 of test case 4

Average MSE for process1- test4 with default value 20.00 is: 0.029853

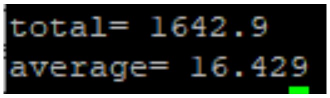
Average MSE for process2- test4 with default value 20.00 is: 0.049095

Average MSE for process3- test4 with default value 20.00 is: 0.036556

Average MSE for process4- test4 with default value 20.00 is: 0.058433

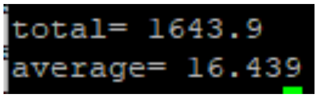
Average MSE for process5- test4 with default value 20.00 is: 0.034427

TEST CASE -5:



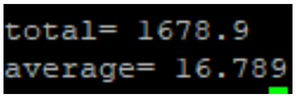
```
total= 1642.9
average= 16.429
```

Process1 of test case 5



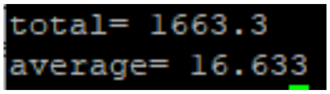
```
total= 1643.9
average= 16.439
```

Process2 of test case 5



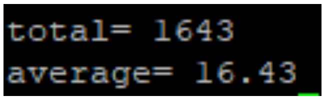
```
total= 1678.9
average= 16.789
```

Process3 of test case 5



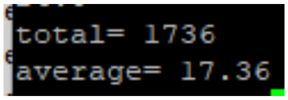
```
total= 1663.3
average= 16.633
```

Process4 of test case 5



```
total= 1643
average= 16.43
```

Process5 of test case 5



```
total= 1736
average= 17.36
```

Process6 of test case 5

Average MSE for process1- test5 with default value 16.66 is: 0.137783

Average MSE for process2- test5 with default value 16.66 is: 0.087124

Average MSE for process3- test5 with default value 16.66 is: 0.113362

Average MSE for process4- test5 with default value 16.66 is: 0.090072

Average MSE for process5- test5 with default value 16.66 is: 0.030723

Average MSE for process6- test5 with default value 16.66 is: 0.095103

3.2. STRIDE SCHEDULING ALGORITHM

3.2.1. AVERAGE CPU UTILIZATION

These are our randomly assigned ticket values:

process 1 ticket -> 9

process 2 ticket -> 5

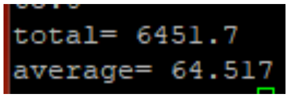
process 3 ticket -> 3

process 4 ticket -> 10

process 5 ticket -> 9

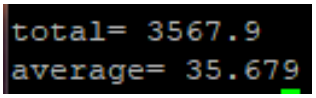
process 6 ticket -> 2

TEST CASE -1:



```
total= 6451.7
average= 64.517
```

process1 of test case 1



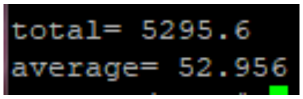
```
total= 3567.9
average= 35.679
```

process2 of test case 1

Average MSE for process1- test1 with default value 64.29 is: 0.022330

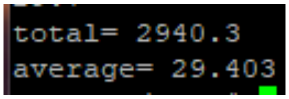
Average MSE for process2- test1 with default value 35.71 is: 0.005301

TEST CASE -2:



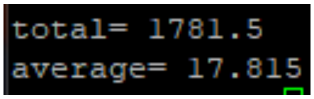
```
total= 5295.6
average= 52.956
```

process1 of test case 2



```
total= 2940.3
average= 29.403
```

process2 of test case 2



```
total= 1781.5
average= 17.815
```

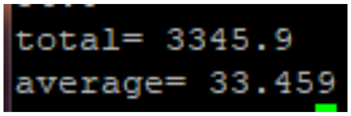
process3 of test case 2

Average MSE for process1- test2 with default value 52.94 is: 0.014854

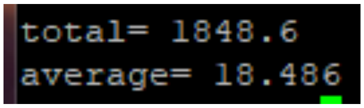
Average MSE for process2- test2 with default value 29.41 is: 0.006357

Average MSE for process3- test2 with default value 17.65 is: 0.004488

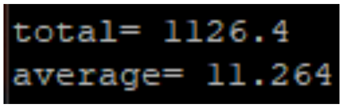
TEST CASE -3:



```
total= 3345.9
average= 33.459
```

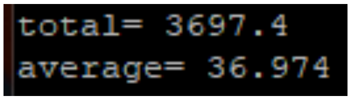


```
total= 1848.6
average= 18.486
```



```
total= 1126.4
average= 11.264
```

process1 of test case 3 **process2 of test case 3** **process3 of test case 3**



```
total= 3697.4
average= 36.974
```

Process4 of test case 3

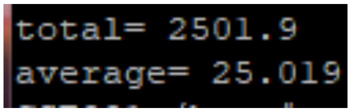
Average MSE for process1- test3 with default value 33.33 is: 0.007645

Average MSE for process2- test3 with default value 18.52 is: 0.008212

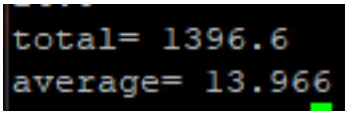
Average MSE for process3- test3 with default value 11.11 is: 0.007813

Average MSE for process4- test3 with default value 37.04 is: 0.007091

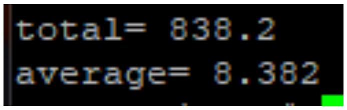
TEST CASE -4:



```
total= 2501.9
average= 25.019
```

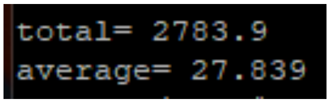


```
total= 1396.6
average= 13.966
```

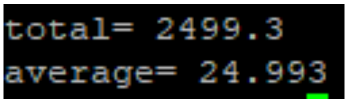


```
total= 838.2
average= 8.382
```

process1 of test case 4 **process2 of test case 4** **process3 of test case 4**



```
total= 2783.9
average= 27.839
```



```
total= 2499.3
average= 24.993
```

process4 of test case 4 **process5 of test case 4**

Average MSE for process1- test4 with default value 25.00 is: 0.005522

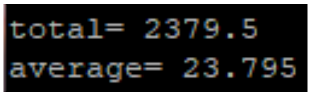
Average MSE for process2- test4 with default value 13.89 is: 0.009115

Average MSE for process3- test4 with default value 8.33 is: 0.006560

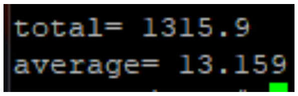
Average MSE for process4- test4 with default value 27.78 is: 0.004823

Average MSE for process5- test4 with default value 25.00 is: 0.006716

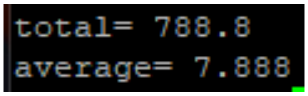
TEST CASE -5:



```
total= 2379.5
average= 23.795
```

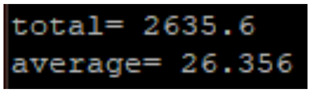


```
total= 1315.9
average= 13.159
```

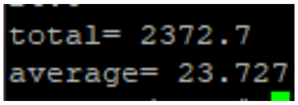


```
total= 788.8
average= 7.888
```

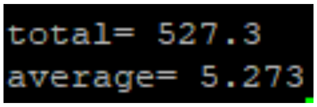
process1 of test case 5 **process2 of test case 5** **process3 of test case 5**



```
total= 2635.6
average= 26.356
```



```
total= 2372.7
average= 23.727
```



```
total= 527.3
average= 5.273
```

Process4 of test case 5 **process5 of test case 5** **process6 of test case 5**

Average MSE for process1- test5 with default value 23.68 is: 0.003234

Average MSE for process2- test5 with default value 13.16 is: 0.000753

Average MSE for process3- test5 with default value 7.89 is: 0.000462

Average MSE for process4- test5 with default value 26.32 is: 0.007679

Average MSE for process5- test5 with default value 23.68 is: 0.005375

Average MSE for process6- test5 with default value 5.26 is: 0.000753

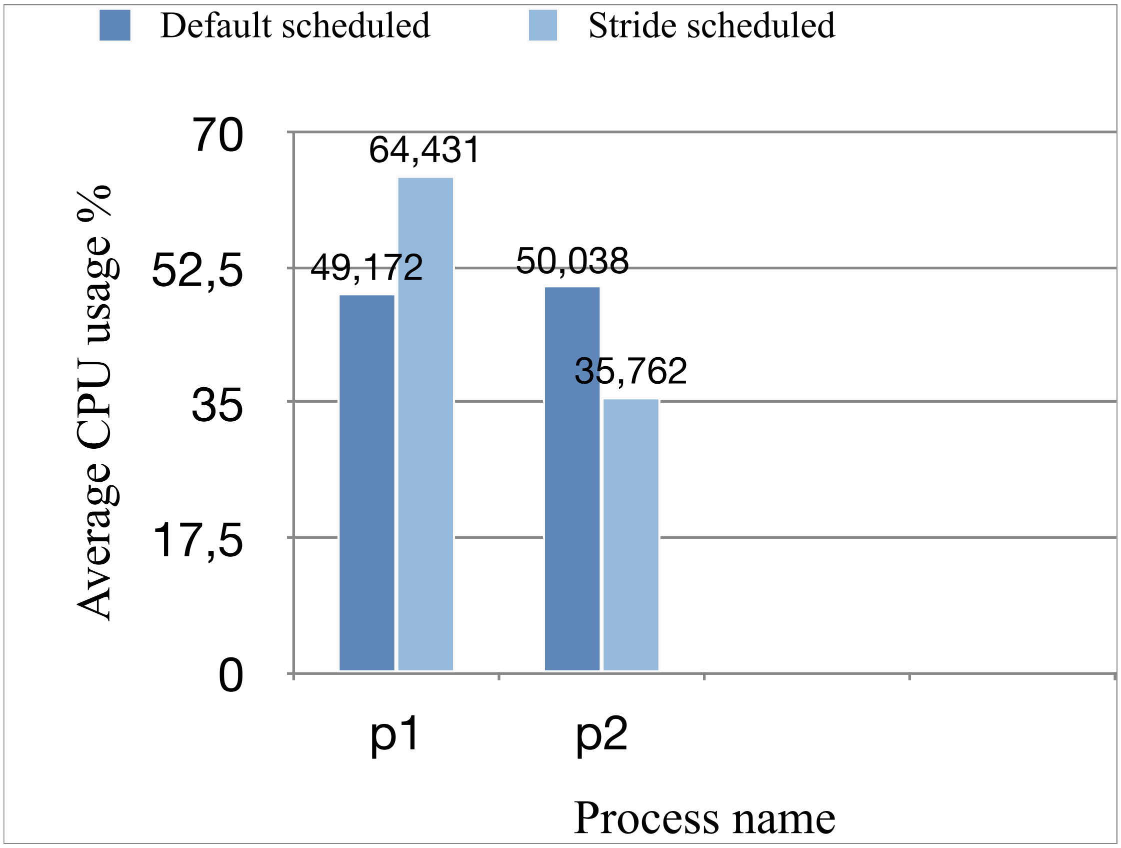
3.3. GRAPHS

3.3.1. Average CPU Usage

TEST CASE -1:

process 1 ticket -> 9

process 2 ticket -> 5

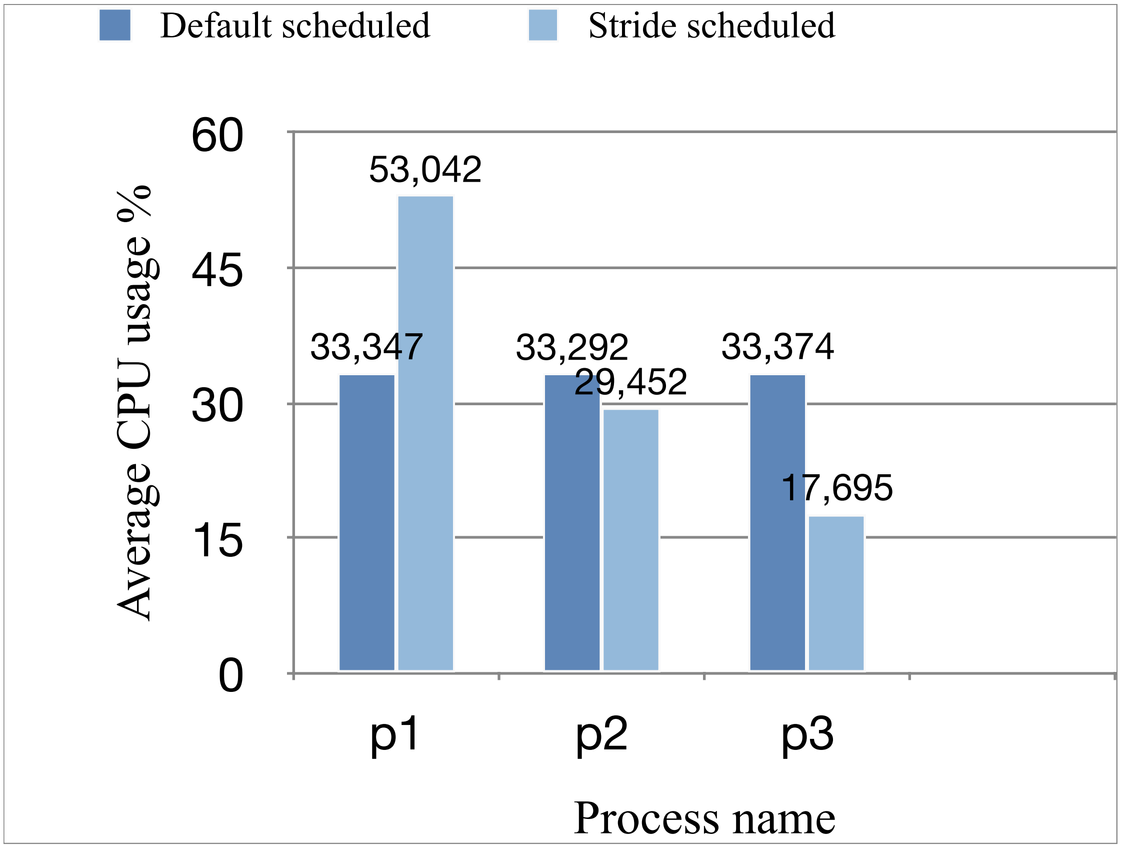


TEST CASE -2:

process 1 ticket -> 9

process 2 ticket -> 5

process 3 ticket -> 3



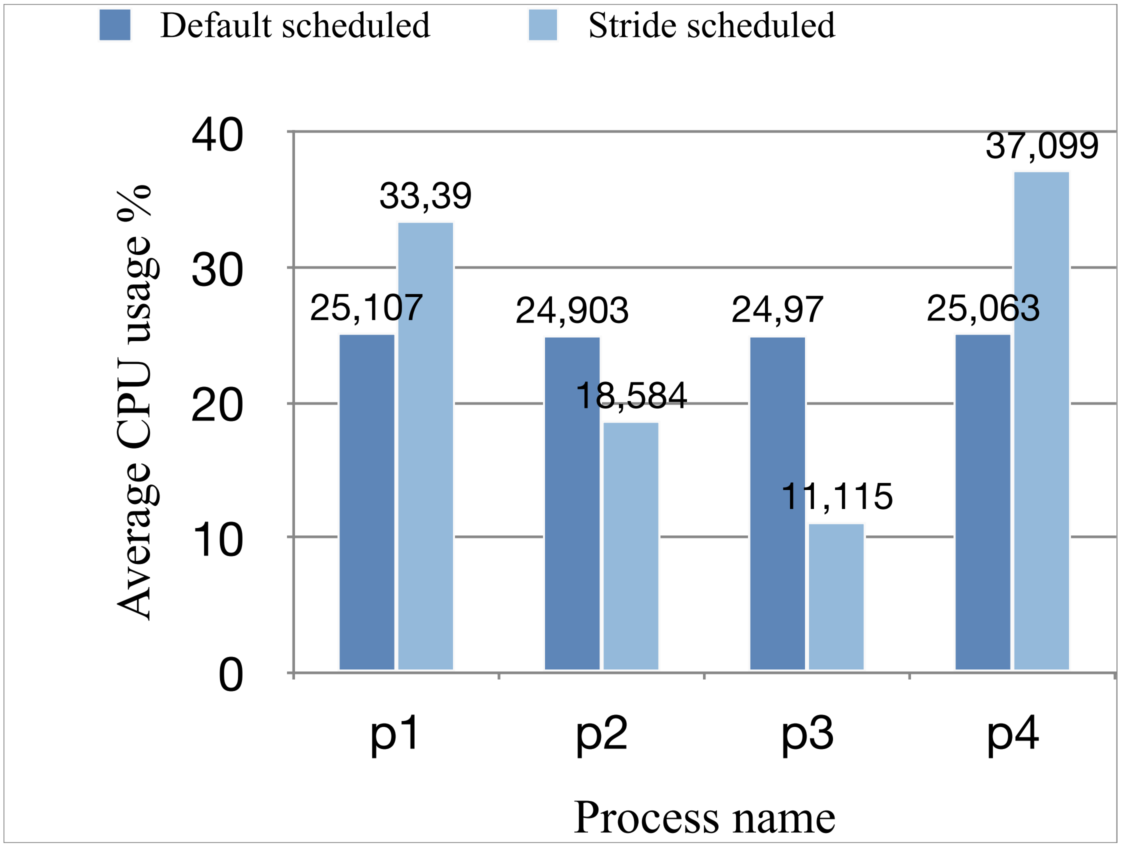
TEST CASE -3:

process 1 ticket -> 9

process 2 ticket -> 5

process 3 ticket -> 3

process 4 ticket -> 10



TEST CASE -4:

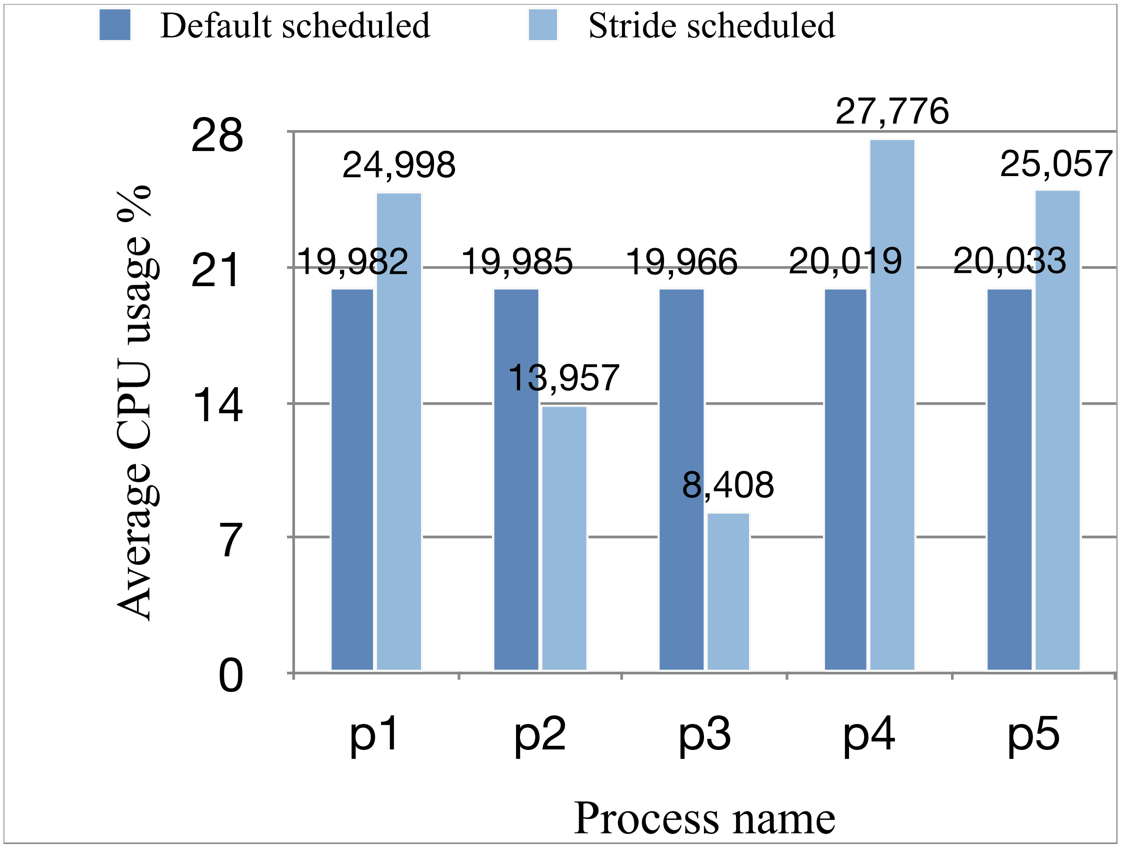
process 1 ticket -> 9

process 2 ticket -> 5

process 3 ticket -> 3

process 4 ticket -> 10

process 5 ticket -> 9



TEST CASE -5:

process 1 ticket -> 9

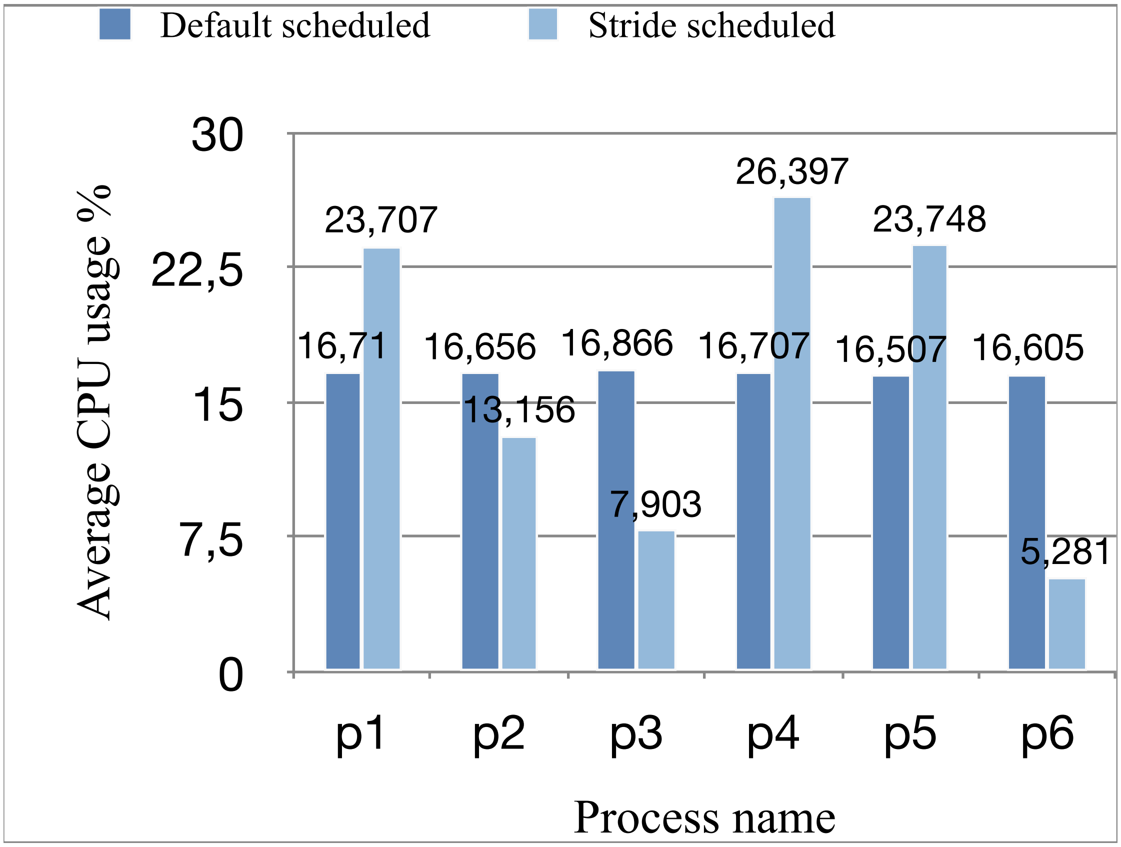
process 2 ticket -> 5

process 3 ticket -> 3

process 4 ticket -> 10

process 5 ticket -> 9

process 6 ticket -> 2

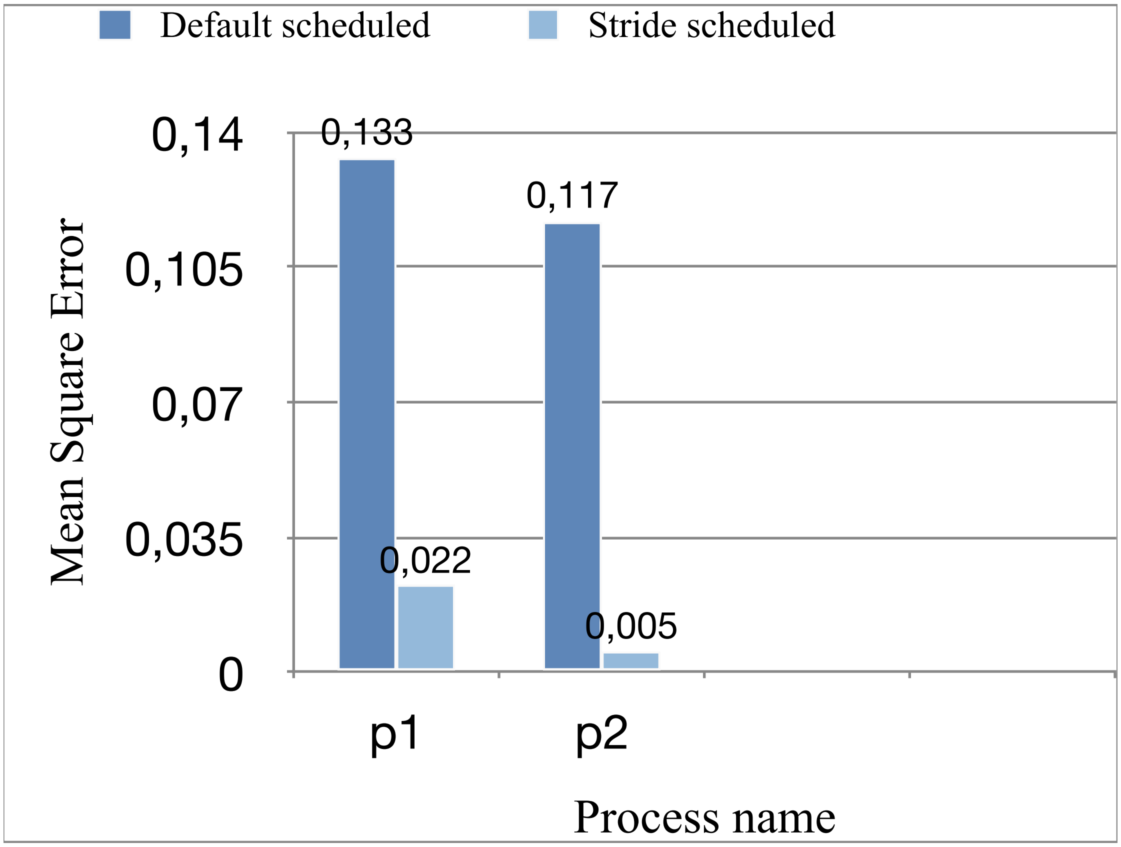


3.3.2. Mean Square Error

TEST CASE -1:

process 1 ticket -> 9

process 2 ticket -> 5

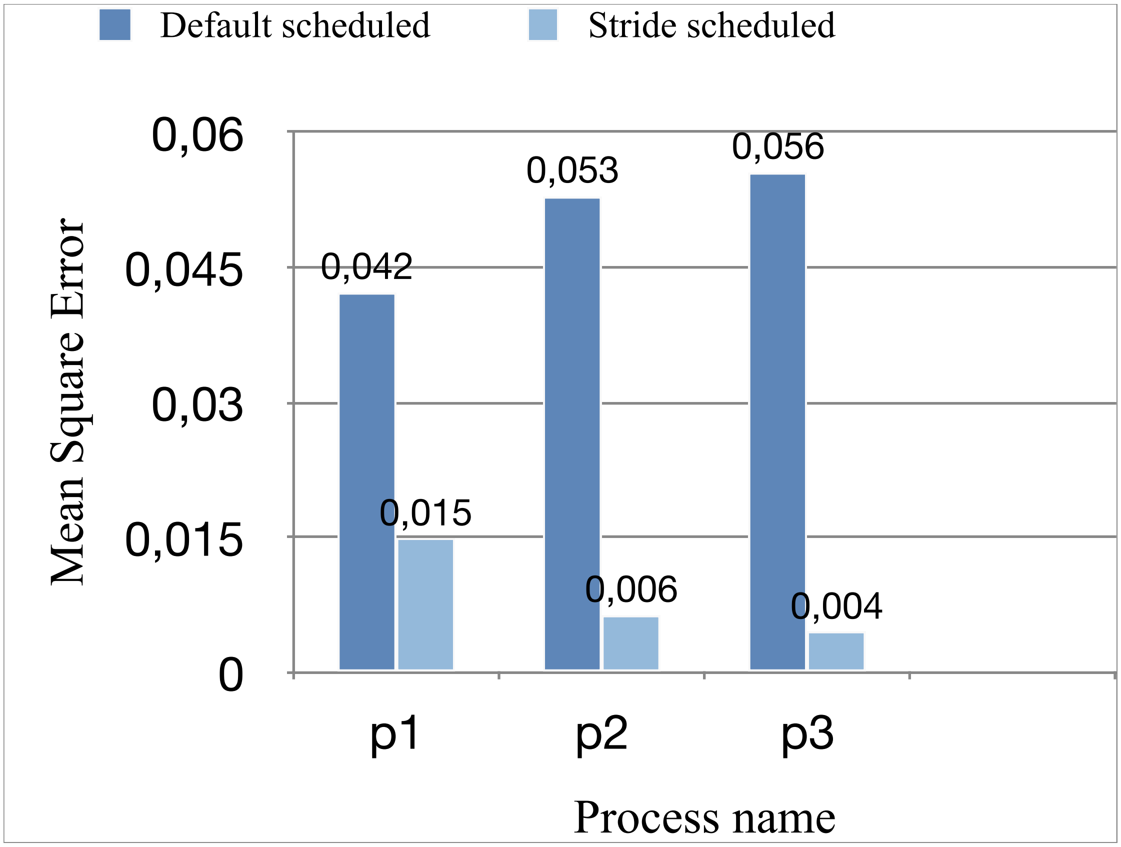


TEST CASE -2:

process 1 ticket -> 9

process 2 ticket -> 5

process 3 ticket -> 3



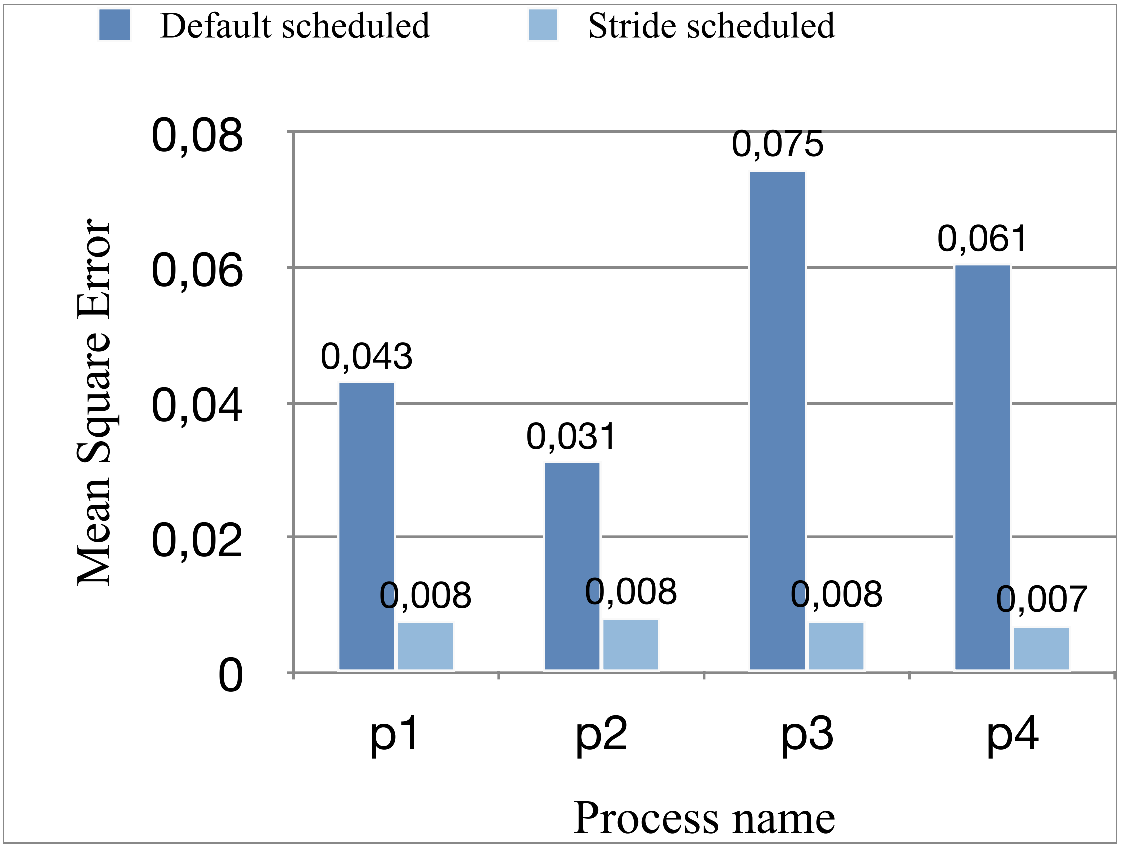
TEST CASE -3:

process 1 ticket -> 9

process 2 ticket -> 5

process 3 ticket -> 3

process 4 ticket -> 10



TEST CASE -4:

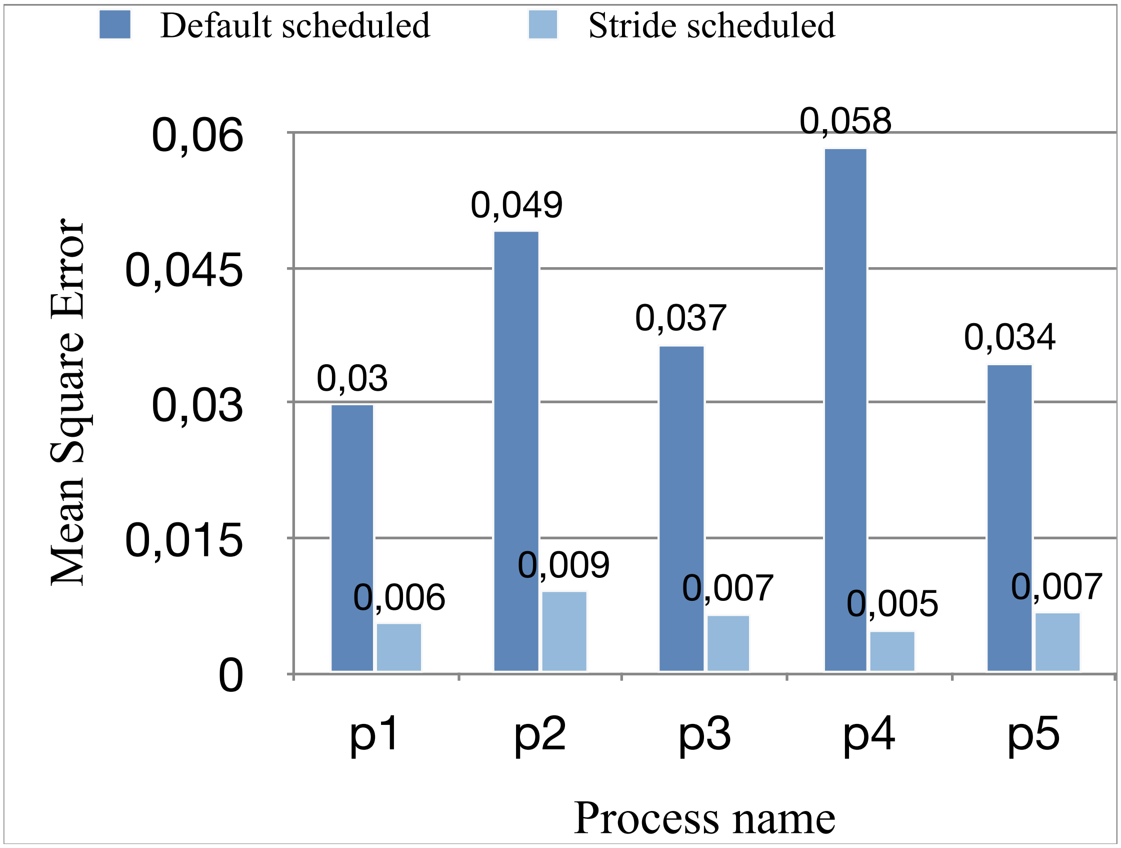
process 1 ticket -> 9

process 2 ticket -> 5

process 3 ticket -> 3

process 4 ticket -> 10

process 5 ticket -> 9



TEST CASE -5:

process 1 ticket -> 9

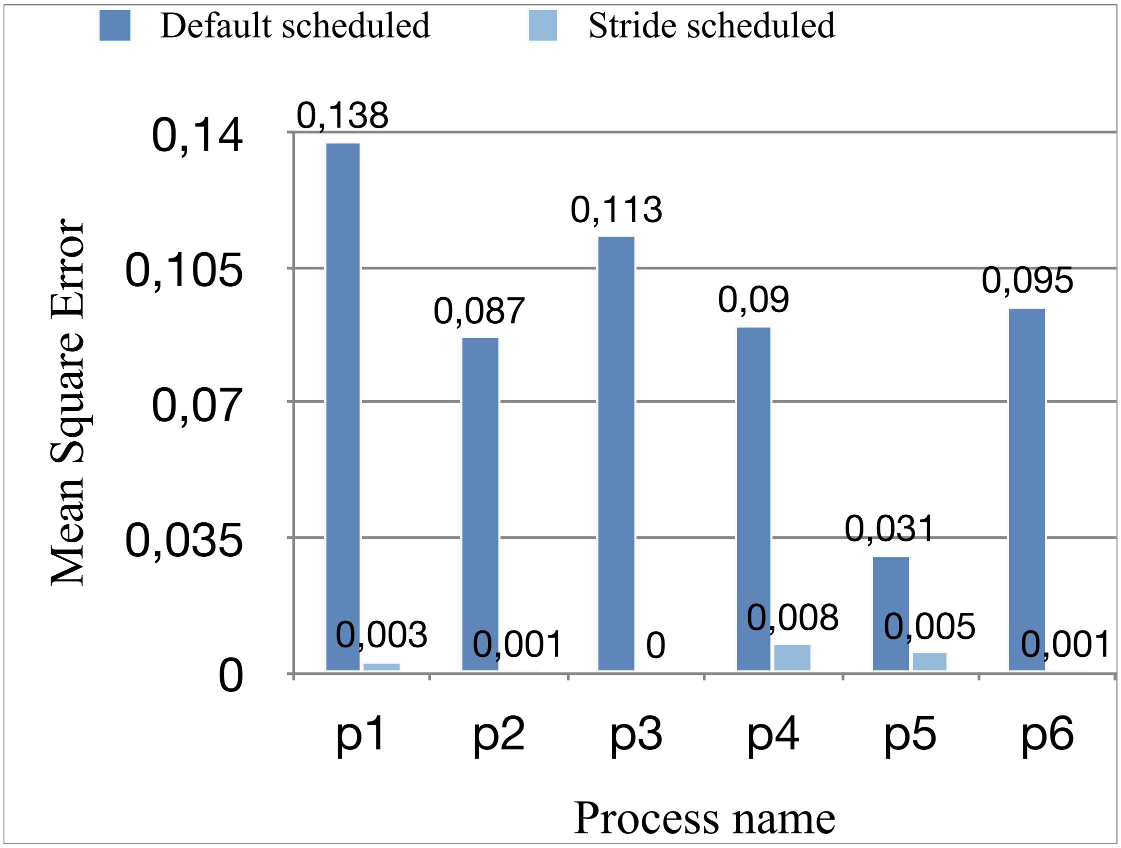
process 2 ticket -> 5

process 3 ticket -> 3

process 4 ticket -> 10

process 5 ticket -> 9

process 6 ticket -> 2



4. CONCLUSION

In the tests above, we individually tested and compared the two scheduler algorithms. According to the obtained results, the Stride Scheduler algorithm appears to be more effective in terms of CPU utilization compared to the default scheduler. The nearly negligible error values suggest that we implemented the algorithm effectively. However, beyond the numerical values on paper, we encountered a performance issue when testing the Stride Scheduler algorithm, manifesting as a slowdown in the computer's operation.We attribute this issue to the multiplier variable we set at 2520. As the number of processes increased, the required integer count also grew, causing a progressively slower performance. Therefore, despite the low error values, we do not consider it appropriate to prefer the Stride Scheduler algorithm over the default algorithm until we address and resolve the multiplier issue.

5. REFERENCES

https://elixir.bootlin.com/linux/2.4.27/source

CSE 331 Operating System Design Course Materials(Adding an example system call.pdf)

https://www.usenix.org/legacy/publications/library/proceedings/usenix01/freenix01/full_papers/alicherry/alicherry_html/node5.html#:~:text=SCHED_OTHER%20is%20the%20default%20universal,processes%20are%20selected%20for%20execution