CSE331 TERM PROJECT

By

Ali Emir Altın

Burak Eymen Çevik

Yusuf Üngör

CSE 331 Operating Systems Design

Term Project Report

Yeditepe University

Faculty of Engineering

Department of Computer Engineering

Spring 2023

ABSTRACT

We have developed a scheduling algorithm for the Linux kernel, known as Stride Scheduling. In contrast to the default Linux scheduling algorithm, the Stride algorithm introduces its own unique ordering mechanism.

TABLE OF CONTENTS

1. INTRODUCTION

In a multiprogramming environment,concurrent processes compete with each other for the CPU's attention, fairness, and proportionality. This project aims to design an alternative scheduling policy called Stride Scheduling. By using the "tickets" concept, stride scheduling assigns each process a numerical value representing its claim on CPU cycles. More tickets equate to bigger slices of the CPU pie, guaranteeing that no process gets left behind. This mechanism promotes fairness and equity in resource allocation.

In the first phase of this project, we have learned how to implement a system call to create an effective connection between the user and kernel space. In the second part of this project, we have modified the default CPU scheduling.

Our modifications to CPU scheduling consist of the following parts:

- Dynamic Ticket Allocation: Upon process creation, a random ticket value is assigned, forming the basis for stride calculation (a ticket divided by a multiplier).
- User-Controlled Activation: A dedicated system call empowers users to seamlessly switch between Stride and the default Linux scheduler, offering ultimate flexibility.
- Algorithm Integration: The schedule function's Repeat_Schedule part in the kernel is strategically enhanced to implement the Stride scheduling algorithm. Processes with the lowest "pass" value gain execution priority, and their pass value increments inversely proportional to their ticket count after each quantum, ensuring a balanced progression and continuous fairness.

We aim to comprehensively test our Stride scheduling algorithm through the creation of four distinct test scenarios involving 2, 3, 4, and 5 processes, respectively. Within each test case, we'll conduct 10 individual tests to track CPU utilization over a 100-second period, deriving accurate average values. These 10 average CPU percentages per test case will serve as our basis for comparison against the expected values, facilitating an evaluation of our algorithm's performance across varying workloads and process counts.

2. DESIGN AND IMPLEMENTATION

2.1. Linux Default Scheduler:

The time-sharing algorithm in Linux prioritizes ensuring that every process gets a fair turn to use the CPU.

Priority: Each process is given a priority number. Lower numbers mean higher priority, like being in the front of a line. Higher numbers mean lower priority, like being at the back of the line. This priority number is called "nice."

Priority value:

- higher nice → low priority (nice : 0 – 20)
- lower nice → high priority (nice : -19 – 0)
- Medium priority → nice=0

Time Slices (Quantums): Everyone gets a certain amount of time to use the CPU. The more important (lower "nice") you are, the longer your turn. This is called a "time slice."

Counter: Each process has a timer. When it's your turn to use the CPU, its timer starts counting down from the time slice it was given. When it reaches zero, it’s turn is over, and it has to wait for the next turn.

Being Fair: The goal is to make sure every process gets a fair chance. If multiple processes have the same priority and are waiting to use the CPU, they take turns. None of the processes can use the CPU for a really long time and leave others waiting too long.

In simple terms, the time-sharing algorithm makes sure every process gets a fair amount of time to do their work. It uses priority (the "nice" number) and time slices to figure out which process gets to use the CPU next.

2.2. Stride Scheduler:

The Stride scheduling algorithm is all about making sure that different processes running on a computer get their fair share of processing time from the CPU. It works like giving out tickets to these processes, and these tickets determine how often a process gets to use the CPU.

Ticket Assignment: Each process gets a random number of tickets when created. This operation is completed in the function called “do_fork()”.

Choosing a Process: Each process also gets a pass value, which tells us which process to run next. In the function “schedule()”, the process with the smallest pass value is chosen to run next.

Incrementing “pass” variable: After the process is run, the pass variable is incremented by the stride value of the process. The stride variable is found by dividing the fixed value int_multiplier by the ticket value.

In simple terms, the Stride algorithm makes sure that processes with fewer tickets (lower priority) still get their turn to use the CPU. It prevents any process from being ignored for too long and keeps things fair, even when some processes are more important than others.

2.3. Implementation:

In order to handle stride scheduling, first we added the necessary variables and macros to “sched.h” as seen in the image. “pass”, “stride” and “ticket” variables are added into the “task_struct” structure.

```
#ifndef SCHED_FLAG_H
#define SCHED_FLAG_H

extern int sched_flag;

#endif /* SCHED_FLAG_H */

#define int_multiplier 300
struct task_struct {
    /*
```

```
long counter;
long nice;
unsigned long policy;
struct mm_struct *mm;
int processor;
int pass;
int stride;
int ticket;
```

In the “schedule()” function in “sched.c”, we have added an if statement which checks the flag “sched_flag” declared in “sched.h” and chooses the scheduling algorithm accordingly. In our design, “sched_flag == 1” corresponds to the default algorithm, while “sched_flag == 0” corresponds to the stride algorithm.

```
if(sched_flag){
    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;
    }
}
```

```
else{
    c = int_multiplier * 2;
    list_for_each(tmp, &runqueue_head) {
        p = list_entry(tmp, struct task_struct, run_list);
        if (can_schedule(p, this_cpu)) {
            if (p->pass < c){
                c = p->pass;
                next = p;
            }
        }
    }

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

        spin_unlock_irq(&runqueue_lock);
        read_lock(&tasklist_lock);
        for_each_task(p)
            p->pass = 0;
        read_unlock(&tasklist_lock);
        spin_lock_irq(&runqueue_lock);
        goto repeat_schedule;
    }
    next->pass = next->pass + next->stride;
}
```

In the “do_fork()” function in “fork.c”, we set the initial values of the variables of newly created processes. In the get_random_bytes function, a 2 byte value is taken to ensure that it is positive. This value is assigned to the “ticket” variable and the “stride” variable is calculated. Zero is assigned to the “pass” variable.

```
*p = *current;
int rand_num = 0;
get_random_bytes(&rand_num, 2);
p->ticket = rand_num % 10 + 1;
p->stride = int_multiplier / p->ticket;
p->pass = 0;
```

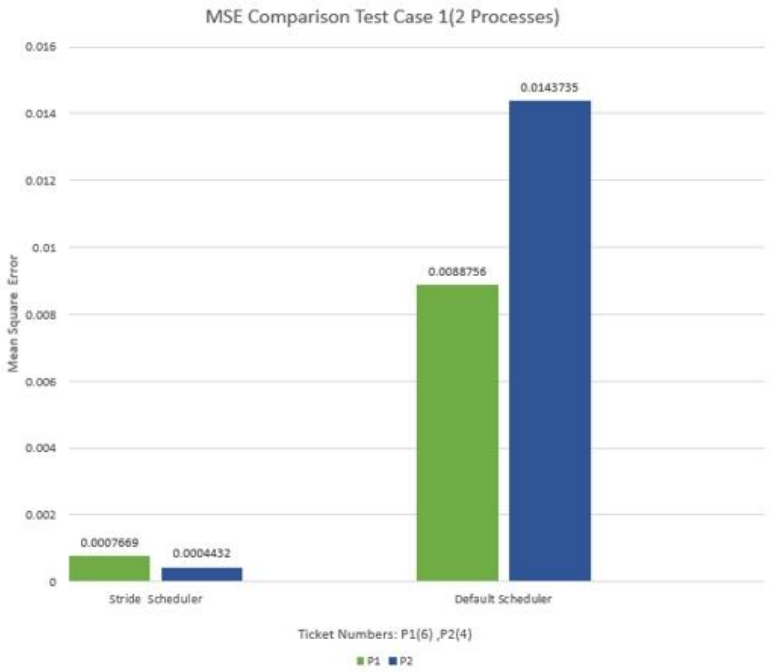
We have added a system call called “change_sched_flag” to be able to switch to a different scheduling algorithm in user space. This system call changes the value of “sched_flag” directly.

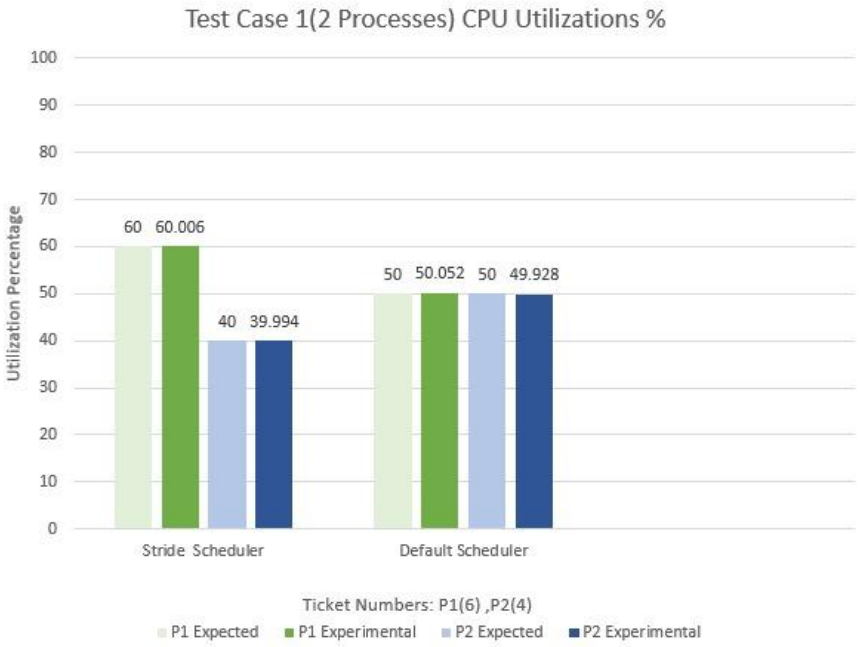
```
#include <linux/change_sched_flag.h>
#include <linux/sched.h>
asmlinkage void sys_change_sched_flag(int flag){
    sched_flag = flag;
}
```

3. TESTS AND RESULTS

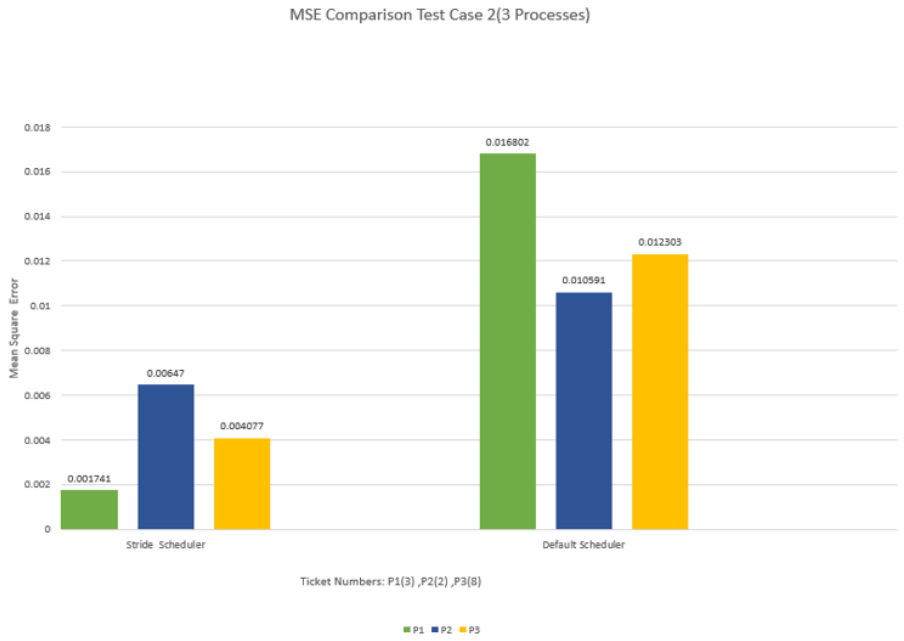
4 test cases were considered in this project. 2 processes, 3 processes, 4 processes and 5 processes. Each test case comprises 10 sets of data collected at 1-second intervals over a span of 100 seconds.

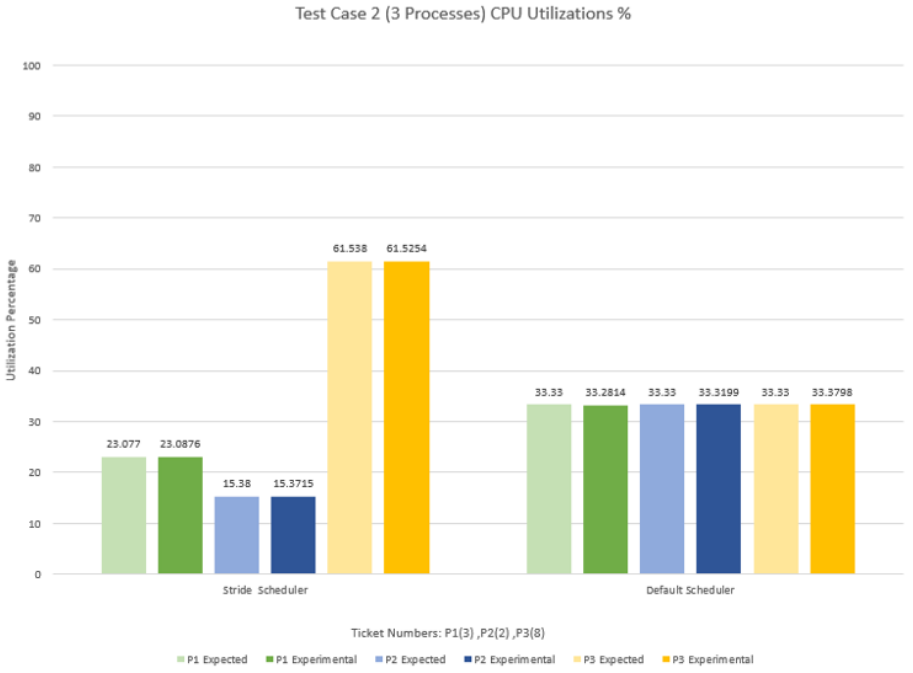
3.1. Test case 1 (2 processes)



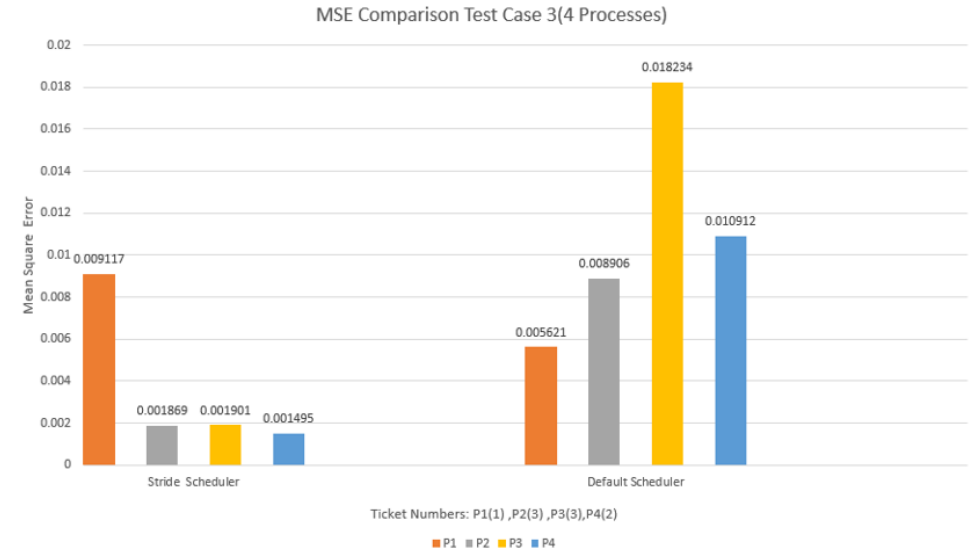


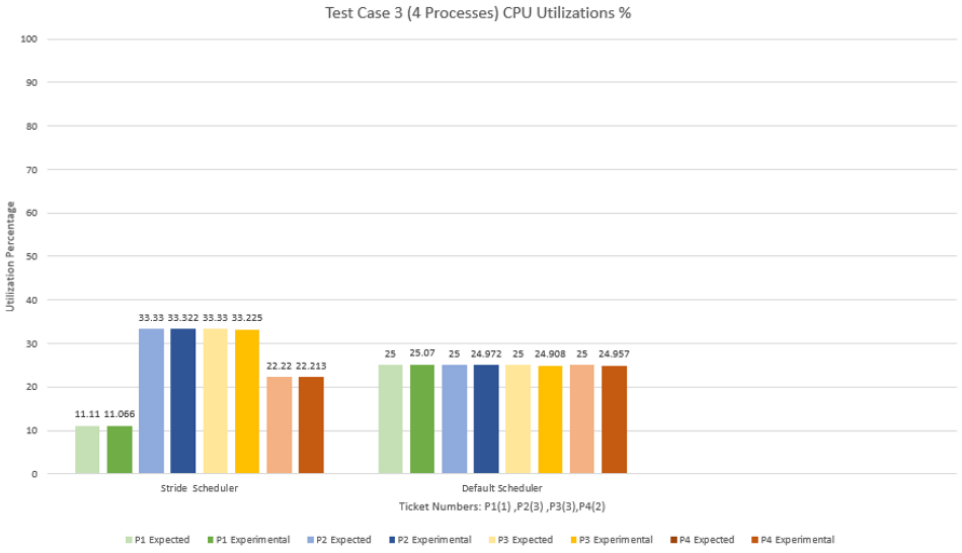
3.2. Test case 2 (3 processes)



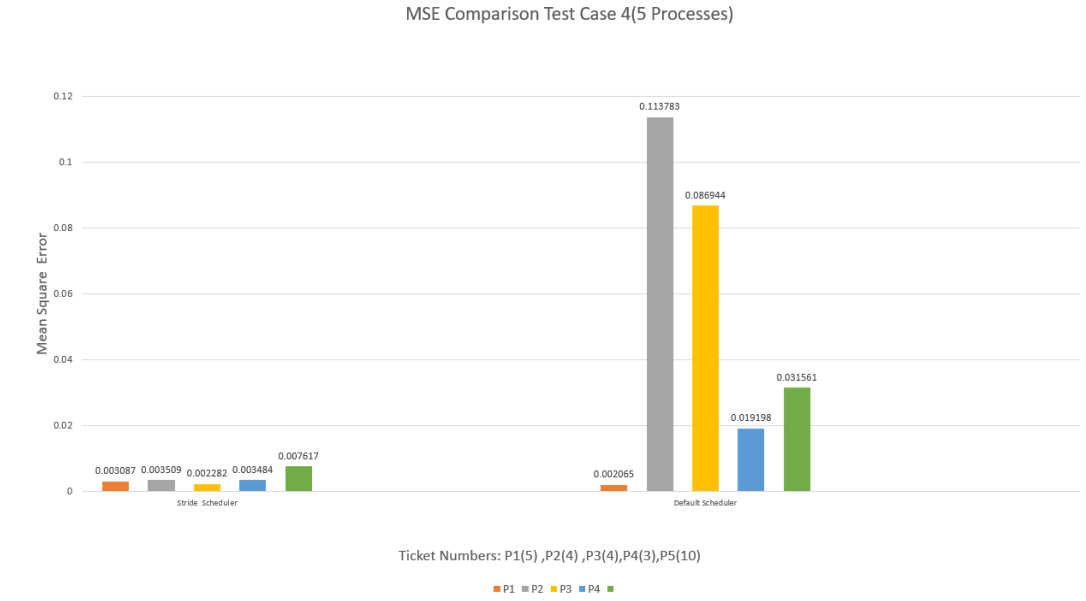


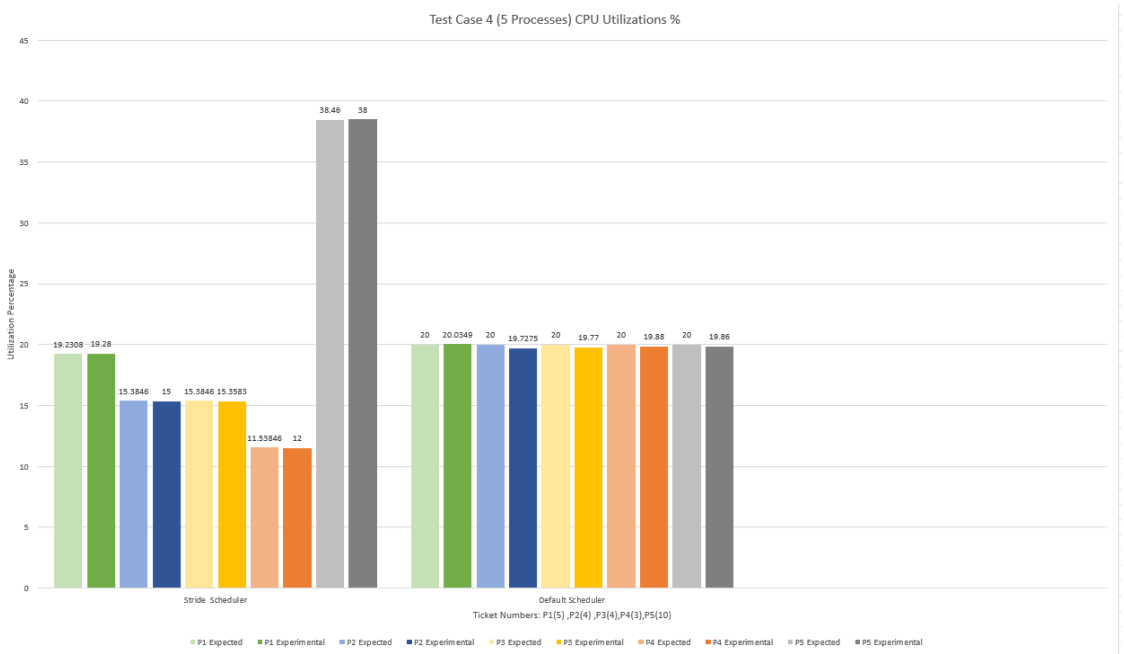
3.3. Test case 3 (4 processes)





3.4. Test case 4 (5 processes)





Default Scheduler:

In the case of the default scheduler, the initial expectation was that it would ensure an even distribution of CPU processing time among processes, resulting in a uniform 20% CPU utilization for each process considering Test Case 4 for example. However, in practice, the observed CPU utilization deviated slightly from these predictions. Among a set of five processes, some ended up using a slightly higher share of the CPU's processing time, while others utilized slightly less.

Stride Scheduler:

The Stride Scheduler determines predicted CPU utilization values for processes based on their allocated ticket numbers. For instance, in test case 1, process p1 has 6 tickets, and p2 has 4 tickets. As a result, p1's predicted CPU utilization is expected to be 60%, while p2's predicted CPU utilization is expected to be 40%. However, in practice, small deviations between these predicted values and the actual CPU usage and utilization may occur. This happens because processes with more tickets are supposed to get more CPU time, but sometimes there are small differences between what's expected and what actually happens due to real-world factors.

Comparison Between Default and Stride Scheduler Experimental Results:

When comparing the Stride Scheduler to the Default Scheduler, it's clear that the Stride Scheduler has some advantages. It's better at predicting and distributing CPU time, thanks to its ticket-based method, which makes CPU usage more precise. This leads to less error and makes it more adaptable in real-world situations where processes have different needs and priorities. On the other hand, the Default Scheduler, although it works in many cases, may struggle to be completely fair, especially when handling different workloads and process priorities. The Stride Scheduler's ability to allocate resources efficiently based on ticket numbers makes it a good choice, but the final decision depends on your specific system needs.

4. CONCLUSION

In conclusion, this project introduced the Stride Scheduling algorithm as an alternative to the default Linux scheduling algorithm, aiming to provide fairness and equity in CPU resource allocation among concurrent processes. The Stride Scheduler, with its ticket-based approach, demonstrated the ability to more precisely predict and distribute CPU time, resulting in less error and improved adaptability in real-world scenarios compared to the Default Scheduler. While the Default Scheduler ensures fairness to some extent, it may struggle with perfect fairness, especially under varying workloads and differing process priorities. Ultimately, the Stride Scheduler's efficient resource allocation based on ticket numbers positions it as a favorable choice, offering a robust solution for diverse system requirements and workloads.

5. REFERENCES

o Operating System Concepts, 9th Edition by Abraham Silberschatz , Peter Baer Galvin and Greg Gagne.
o Class materials for the project.