# ABSTRACT

# Context switching is a pivotal operation within operating systems, enabling them to manage multiple processes or threads concurrently on a single CPU. At its core, context switching involves the seamless transition between executing different tasks by saving and restoring the state of each process or thread as needed. This allows the CPU to efficiently allocate resources and time to various tasks, thereby maximizing system utilization and responsiveness.

# One of the key components of context switching is the concept of process state. Each process possesses its own state, encompassing crucial information such as CPU registers, program counter, stack pointer, and other relevant data necessary for execution. The operating system must meticulously manage and update these states during context switches to ensure the continuity and integrity of each process's execution.

# Central to the context switching process is the scheduler, which dictates the order in which processes are executed on the CPU. The scheduler employs various algorithms, such as round-robin or priority-based scheduling, to determine which process to execute next. By intelligently selecting processes based on predefined criteria, the scheduler optimizes resource allocation and system performance.

# Interrupts play a crucial role in triggering context switches. Hardware interrupts, such as I/O requests

# or timer interrupts, and software interrupts, such as system calls, prompt the CPU to suspend the currently executing process and transfer control to the interrupt handler. This mechanism allows the operating system to swiftly respond to external events and manage system resources effectively.

# The context switching mechanism itself involves a series of steps. Firstly, the state of the current process, often referred to as the "old process," is saved to memory. Subsequently, the scheduler selects the next process to execute, known as the "new process." The state of the new process is then loaded from memory, and control is transferred to it. Additionally, memory management data structures may need to be updated to reflect changes in process states.

# The Process Control Block (PCB) serves as a critical data structure maintained by the operating system for each process. It contains vital information such as process ID, state, CPU registers, memory allocation, and scheduling details. During context switches, the PCB is updated to reflect the current state of each process, facilitating efficient process management and scheduling.

# INTRODUCTION

In modern operating systems, multitasking is a fundamental feature that allows multiple processes to execute concurrently on a single processor. Context switching plays a vital role in enabling efficient multitasking and resource allocation. It involves saving the state of a currently executing process and restoring the state of a new process to ensure a seamless transition between them.

The purpose of context switching is twofold: to facilitate process scheduling and to handle various interrupt-driven events such as I/O operations. In this particular implementation, context switching, and Round Robin (RR) scheduling have been implemented to manage process execution.

RR scheduling is a widely used algorithm in which each process is given a fixed time slice or quantum of CPU time. Once a process's time slice is completed, the processor is allocated to the next process in the scheduling queue. This cyclic allocation of CPU time among processes ensures fairness and prevents any single process from monopolizing system resources.

When a process switch occurs, the current state of the running process needs to be saved. This state includes critical information such as the program counter, register values, and other execution context data. By saving this information, the operating system can resume the execution of the process from where it was left off when it retains control of the processor.

In addition to process scheduling, context switching also plays a crucial role in handling I/O interrupts. When an I/O operation completes or an external device requires attention, an interruption is generated. The processor temporarily suspends the execution of the current process and transfers control to the interrupt handler. To ensure that the interrupted process can later resume its execution seamlessly, its context is saved during the interrupt processing and restored once the interrupt handling is complete.

The implementation of context switching for both scheduling and I/O interrupts enhances resource utilization and responsiveness. It allows the operating system to efficiently manage the execution of multiple processes, ensuring that each process receives a fair share of processor time. Additionally, context switching enables the system to handle interrupt-driven events effectively, preventing delays and ensuring the overall system performance remains optimal.

# ABOUT LINUX

Linux is a popular open-source operating system kernel that serves as the foundation for numerous Linux-based operating systems (distributions or distros) such as Ubuntu, Fedora, Debian, and CentOS. Developed by Linus Torvalds in 1991, Linux was inspired by the UNIX operating system and designed to be highly customizable, stable, secure, and efficient.

Key features and characteristics of Linux include:

1. **Open Source**: Linux is released under the GNU General Public License (GPL), which grants users the freedom to view, modify, and distribute the source code. This fosters collaboration, innovation, and a vibrant community of developers.
2. **Kernel**: Linux is primarily a kernel, which is the core component responsible for managing system resources, facilitating hardware communication, and providing essential services to other software components.
3. **Multiuser and Multitasking**: Linux supports multiple users simultaneously, allowing multiple individuals to use the system and run processes concurrently. It also provides multitasking capabilities, enabling efficient execution of multiple processes at the same time.
4. **Stability and Reliability**: Linux is known for its stability and reliability, often powering critical infrastructure and servers. It is designed to handle high workloads and maintain system integrity even under heavy usage.
5. **Security**: Linux emphasizes security and provides robust mechanisms to protect the system and user data. Its permission-based file system and user-level security policies contribute to a secure computing environment.
6. **Flexibility and Customizability**: Linux offers extensive customization options, enabling users to tailor their operating system to their specific needs. Different desktop environments, package managers, and software configurations can be chosen, making Linux highly adaptable.
7. **Package Management**: Linux distributions typically provide package management systems (e.g., apt, yum, pacman) that simplify software installation, updates, and dependency resolution. These systems offer vast repositories of precompiled software packages.
8. **Command-Line Interface (CLI)**: Linux offers a powerful command-line interface, allowing users to interact with the system through text commands. This provides greater control, automation capabilities, and the ability to script complex tasks.
9. **Wide Hardware Support**: Linux supports a wide range of hardware architectures, making it versatile for use on various devices, including desktops, servers, embedded systems,mobile devices, and supercomputers.
10. **Supportive** Community: Linux benefits from a large and passionate community of users, developers, and contributors who actively provide support, documentation, and continually enhance the system.
    1. Linux has gained significant popularity and widespread adoption due to its stability, security, versatility, and cost-effectiveness. It powers a diverse range of applications, from personal computers and servers to mobile devices, IoT devices, and cloud computing infrastructure.

**2.SYSTEMANALYSIS**

**2.1. EXISTING SYSTEM:**

Traditional context switching techniques involve saving the entire state of a process to memory before loading the state of another process. This approach incurs significant overhead due to memory accesses and context synchronization. Additionally, as the number of processes increases, the overhead of context switching becomes more pronounced, leading to degraded system performance.

This process incurs overhead, impacting system performance, particularly in scenarios with frequent switches or resource-intensive tasks. Efficient management of context switching is crucial for optimizing system responsiveness and resource utilization.

**2.2.DISADVANTAGES:**

The existing system's context switching mechanism in the operating system may suffer from several disadvantages.

One prominent issue is inefficiency, where context switches incur a significant overhead due to the need to save and restore extensive task states.

This inefficiency can lead to decreased system responsiveness and degraded performance.

Additionally, the existing system may lack sophisticated scheduling algorithms, resulting in suboptimal task prioritization and resource allocation.

This can lead to scenarios where critical tasks are delayed or starved of resources, impacting overall system stability and user experience.

Furthermore, the complexity of managing context switches in a multitasking environment can introduce challenges such as concurrency issues, race conditions, and deadlocks, making the system more prone to bugs and vulnerabilities.

Overall, these limitations highlight the need for improvements in the existing context switching mechanism to enhance system efficiency, reliability, and scalability.

# 2.2 PROPOSED SYSTEM:

# In the proposed system, we aim to address the limitations of the existing context switching mechanism in the operating system by introducing several enhancements. Firstly, we plan to optimize context switching efficiency through techniques such as lightweight context switching or lazy context switching, minimizing the overhead associated with saving and restoring task states. Additionally, we intend to implement advanced scheduling algorithms that prioritize tasks based on factors like priority, fairness, and real-time requirements, ensuring optimal resource utilization and system responsiveness. Furthermore, the proposed system will incorporate robust concurrency control mechanisms to prevent issues such as race conditions and deadlocks, enhancing system stability and reliability.

# 2.2.1 ADVANTAGES:

# The advantages of our proposed system include:

# Reduced overhead: By selectively saving and restoring process state, our approach reduces memory accesses and context synchronization overhead.

# Improved efficiency: Faster context switches lead to improved system responsiveness and better utilization of CPU resources.

# Scalability: Our lightweight context switching mechanism scales more efficiently with increasing numbers of processes, enabling better support for multi-core and multi-threaded systems.

# Real-time performance: By minimizing context switching overhead, our approach is well-suited for real-time systems, ensuring timely execution of critical tasks.

# 3.ABOUT PROGRAMMING LANGUAGE

Programming languages are formal languages used to write computer programs, scripts, and algorithms that instruct computers to perform specific tasks. Here's a brief overview of programming languages:

1. High-level vs. Low-level Languages:
   * High-level languages like Python, Java, and C++ provide abstractions and are easier to read, write, and understand. They often have extensive libraries and frameworks, making development efficient.
   * Low-level languages like Assembly and Machine Code are closer to the hardware and provide more direct control over system resources. They are less readable and require a deep understanding of computer architecture.
2. Imperative vs. Declarative Languages:
   * Imperative languages, such as C, Python, and Ruby, focus on specifying detailed steps to solve a problem. They use statements and control structures to manipulate program state and control flow.
   * Declarative languages, like SQL and Prolog, focus on describing the desired outcome rather than specifying the steps. They use expressions and logical rules to define relationships and constraints.
3. Object-Oriented Programming (OOP):
   * OOP languages, like Java, C++, and Python, organize code around objects that encapsulate data and behavior. They emphasize concepts like inheritance, polymorphism, and encapsulation for code reusability and modularity.
4. Functional Programming (FP):
   * FP languages, such as Haskell, Scala, and Lisp, treat computation as the evaluation of mathematical functions. They emphasize immutability, higher-order functions, and avoiding side effects for improved clarity and maintainability.
5. Scripting Languages:
   * Scripting languages, like JavaScript, Python, and Ruby, are often used for automating tasks and rapid development. They are interpreted at runtime and provide dynamic typing, making them flexible and easy to use.
6. Compiled vs. Interpreted Languages:

Compiled languages, like C, C++, and Go, are translated into machine code before execution. They often provide performance benefits but require a separate compilation step.

* + Interpreted languages, like Python, JavaScript, and Ruby, are executed line-by-line without a separate compilation step. They offer dynamic typing and faster development cycles.

1. Domain-Specific Languages (DSLs):
   * DSLs are programming languages designed for specific domains or industries. Examples include SQL for database queries, HTML/CSS for web development, and MATLAB for scientific computing.
2. Popular Languages:
   * Python: Versatile, easy to read, with extensive libraries and frameworks.
   * Java: Platform-independent, object-oriented, widely used for enterprise applications.
   * JavaScript: Mainly used for web development, runs in web browsers.
   * C/C++: Efficient, low-level languages used for system programming and performance- critical applications.
   * Ruby: Known for its simplicity and productivity, commonly used in web development with the Ruby on Rails framework.
   * Go: Modern language known for its concurrency support and fast execution.
   * Swift: Developed by Apple, used for iOS/macOS app development.
   * Rust: Focused on safety, concurrency, and performance, popular for systems programming.

Each programming language has its strengths and areas of application. The choice of language depends on factors such as project requirements, performance needs, development team expertise, and ecosystem support.

# SYSTEM ANALYSIS

## ABOUT THE PROJECT

Context switching is a crucial mechanism in operating systems that enables efficient multitasking and resource allocation. In the context of this particular implementation, context switching, and Round Robin (RR) scheduling have been implemented. Context switching occurs whenever a process switch takes place, allowing the processor to allocate its resources to the next process in the scheduling queue. This is particularly relevant when the time slice allocated to a process is completed, prompting the need to switch to the next process.

To ensure a smooth transition between processes, it is necessary to save the state of the current process before switching to the next one. This saved state includes vital information such as the program counter, register values, and other relevant execution context data. By preserving this information, the operating system can resume the execution of the process from where it was left off, ensuring continuity, and preventing any loss of progress or data.

In addition to process scheduling, the implemented context switch also caters to I/O interrupts. When an I/O interrupt occurs, the processor needs to temporarily suspend the execution of the current process and handle the interrupt request. During this time, the processor saves the current process's context, allowing it to later resume its execution once the interrupted processing is complete.

By incorporating context switching for both scheduling and I/O interrupts, the operating system ensures efficient resource utilization, improved responsiveness, and fair allocation of processor time among various processes. This implementation enables seamless multitasking and effectively manages the execution of multiple processes in a time-shared manner.

## ANALYSIS Problem definition:

We have implemented Context Switching and RR scheduling. We need to do Context Switching whenever Process switch takes place. Here Scheduling is done, so whenever time slice of one process is completed, processor is allocated to next process, for this we need to save the state of process so that next time it should run from where it was left. We have implemented context switch for Scheduling and I/O interrupts.

## O.S. concepts used:

Five-state model: For handling processes we have considered the five-state model. States are Blocked, Running and Ready. Blocked and Ready States are Implemented through Queue. Ready Queue is implemented using circular Queue because process should remain ready until it terminates, or its execution gets completed. If the process suffers from I/O interrupt then that Process is dequeued from Ready queue and enqueued to Blocked Queue. Running is not queue because we have considered that one process can run at a time. Whenever Resource is available It is removed from Blocked queue and enqueued to ready queue.

## Scheduling:

For managing Ready Queue, scheduling is done. We have considered short term scheduling. For this, Round Robin is chosen as scheduling algorithm and quantum=2.Scheduling is done for the process present in ready queue. Quantum is chosen to be 2 to minimize the risk of starvation. Also, if the process is short then RR provides good response time. It gives fair treatment to all processes.

## Context Switching:

When context switch occurs, for example if process runs for one time slice, but its execution is not completed, then whenever next time processor is allocated to it process must start from where it has left. For this purpose, PCB is used. This stored data is the context of the process.

## Process Control Block:

Process has many elements. Out of which Program and code are essential. PCB contains crucial information needed for a process to execute. We have considered that PCB contains PID (process identifier), State (Describes in which state the process is), PC (Program Counter: it contains address of the next instruction which will be executed), SP (Stack Pointer: it is small register that stores the address of the last program request in a stack).

# SYSTEM REQUIREMENTS SPECIFICATION

## HARDWARE REQUIREMENTS

The hardware requirements for the "**Context switching.** "

* + 1. **Computer:** A computer with sufficient processing power and memory to support the development environment and operating system is required.
    2. **Storage:** Sufficient storage space is required to store the development environment, operating system,and any related files. At least 20 GB of free disk space is recommended.
    3. **Monitor:** A monitor with a minimum resolution of 1024x768 is recommended for an optimal user experience while developing and testing the Linux shell.
    4. **Input devices:** Input devices such as a keyboard and mouse are necessary to interact with the development environment and test the Linux shell.

## SOFTWARE REQUIREMENTS

The software requirements for the "Making Your Own Linux Shell":

1. **Operating system:** The project will involve developing a custom Linux shell, so the project team will need a Linux distribution installed on their development machine. Popular distributions for development include Ubuntu, Debian, Fedora, and CentOS.
2. **Text editor or Integrated Development Environment (IDE):** The project team will need a text editoror IDE to write and edit the code for the Linux shell. Popular options include Visual Studio Code, Atom,Sublime Text, and Vim.
3. **Compiler and build tools:** The project team will need a compiler and build tools to compile the sourcecode into an executable binary file. For example, the team may choose to use the GNU Compiler Collection (GCC) and GNU Make.
4. **Virtualization software:** If the project team plans to test the Linux shell on different distributions andenvironments, then virtualization software such as VirtualBox or VMWare may be needed.
5. **GTK (GIMP Toolkit):-** The GTK module in Linux refers to the GTK (GIMP Toolkit) library, which is a popular toolkit for creating graphical user interfaces (GUIs) in Linux and other Unix-like operating systems.

# IMPLEMENTATION

## SOURCE CODE

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/wait.h>

char \*built\_in\_str[] = {

"cd",

"help",

"exit",

"pwd",

"history",

"!!"

};

int num\_built\_in = sizeof(built\_in\_str) / sizeof(char \*);

int my\_cd(char \*\*args) {

chdir(args[1]);

return 1;

}

int my\_exit() {

return 0;

}

int my\_help() {

int i = 0;

printf("\n");

printf("Linux Shell, version 1.0.\n");

printf("These Shell commands are defined internally.\n");

printf("\n");

for (i = 0; i < num\_built\_in; i++) {

printf("%s\n", built\_in\_str[i]);

}

printf("\n");

return 1;

}

int my\_pwd() {

char cwd[1024];

if (getcwd(cwd, sizeof(cwd)) != NULL) {

fprintf(stdout, "%s\n", cwd);

} else {

printf("getcwd() error!\n");

}

return 1;

}

int my\_history(char \*\*history, int count) {

int i;

if (count == 0) {

printf("history is empty!\n");

return 1;

}

for (i = 0; i < count - 1; i++) {

printf("%s\n", history[i]);

}

return 1;

}

int my\_exclamation(char \*\*history, int count, char \*\*args) {

if (count == 0) {

printf("no previous command!\n");

return 1;

} else {

if (strcmp(history[count - 1], built\_in\_str[0]) == 0) {

my\_cd(args);

}

if (strcmp(history[count - 1], built\_in\_str[1]) == 0) {

my\_help();

}

if (strcmp(history[count - 1], built\_in\_str[2]) == 0) {

my\_exit();

}

if (strcmp(history[count - 1], built\_in\_str[3]) == 0) {

my\_pwd();

}

if (strcmp(history[count - 1], built\_in\_str[4]) == 0) {

my\_history(history, count);

}

}

return 1;

}

int check\_built\_in\_command(char \*\*args) {

int i;

if (args[0] == NULL) {

return 0;

}

for (i = 0; i < num\_built\_in; i++) {

if (strcmp(args[0], built\_in\_str[i]) == 0) {

return 1;

}

}

return 0;

}

int execute\_built\_in\_command(char \*\*args, char \*\*history, int count) {

int flag;

if (strcmp(args[0], built\_in\_str[0]) == 0) {

flag = my\_cd(args);

}

if (strcmp(args[0], built\_in\_str[1]) == 0) {

flag = my\_help();

}

if (strcmp(args[0], built\_in\_str[2]) == 0) {

flag = my\_exit();

}

if (strcmp(args[0], built\_in\_str[3]) == 0) {

flag = my\_pwd();

}

if (strcmp(args[0], built\_in\_str[4]) == 0) {

flag = my\_history(history, count);

}

if (strcmp(args[0], built\_in\_str[5]) == 0) {

flag = my\_exclamation(history, count, args);

}

return flag;

}

char \*\*parse\_command(char \*my\_line) {

int buffer\_size = 64;

int i = 0;

char \*arg;

char \*\*args = malloc(buffer\_size \* sizeof(char \*));

arg = strtok(my\_line, " \t\r\n\a");

while (arg != NULL) {

args[i] = arg;

i++;

arg = strtok(NULL, " \t\r\n\a");

}

args[i] = NULL;

return args;

}

char \*read\_command\_line(void) {

int bufsize = 1024;

char \*buffer = malloc(sizeof(char) \* bufsize);

int c;

int i = 0;

while ((c = getchar()) != '\n' && c != EOF) {

buffer[i] = c;

i++;

}

buffer[i] = '\0';

return buffer;

}

void print\_prompt() {

printf(">> ");

}

int main(int argc, char \*\*arg) {

char \*my\_line;

char \*\*my\_command;

int bufsize = 1024;

char \*\*history = malloc(bufsize \* sizeof(char \*));

int child\_pid;

int count = 0;

int flag = 1;

int status;

while (flag) {

print\_prompt();

my\_line = read\_command\_line();

my\_command = parse\_command(my\_line);

if (my\_command[0] != NULL) {

if (check\_built\_in\_command(my\_command)) {

if (my\_command[0][0] != '!') {

history[count] = strdup(my\_command[0]);

count++;

}

flag = execute\_built\_in\_command(my\_command, history, count);

} else {

history[count] = strdup(my\_command[0]);

count++;

child\_pid = fork();

if (child\_pid == 0) {

execvp(my\_command[0], my\_command);

exit(EXIT\_FAILURE);

} else {

do {

waitpid(child\_pid, &status, WUNTRACED);

} while (!WIFEXITED(status) && !WIFSIGNALED(status));

}

}

}

free(my\_line);

free(my\_command);

}

return 0;

}

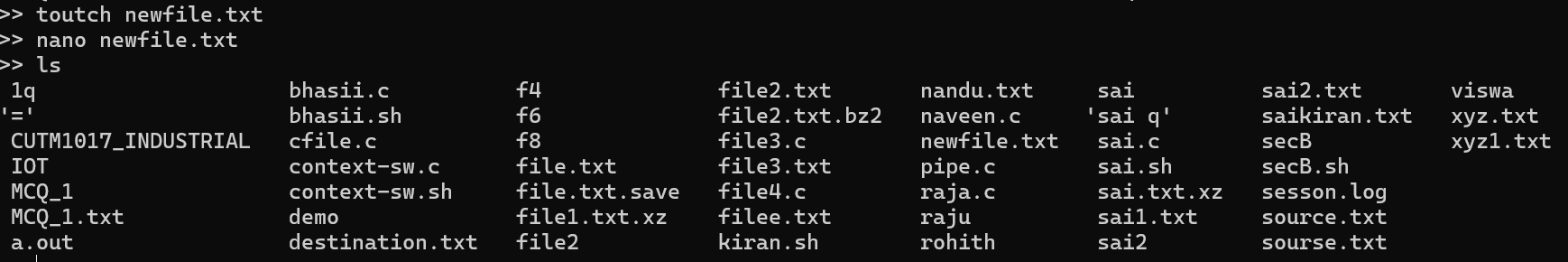
# OUTPUT SCREENS

The “ls” command displays a list of files present in the system.



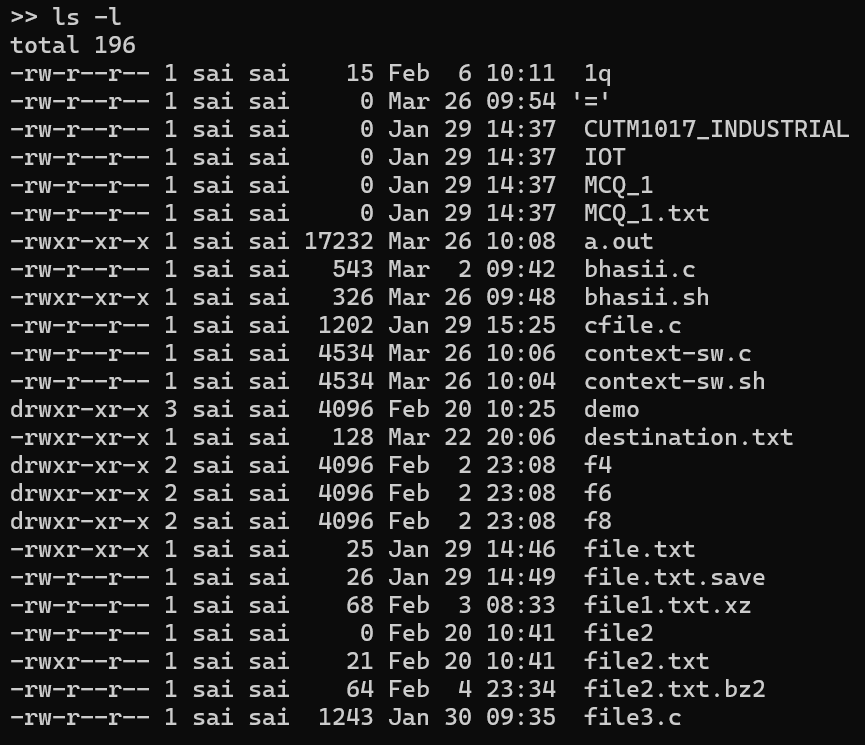
### Figure 7.1 “ls” command implementation

The following output shows the creation of an empty file using touch command.



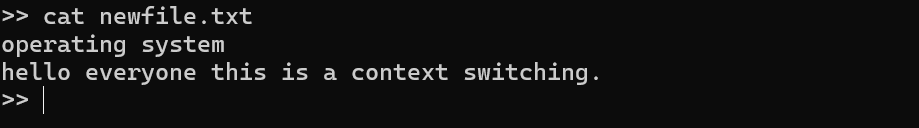
### Figure 7.2 Creating empty file using “touch” command

The ls -l commands show the access controls, memory in bytes etc., along with filename



### Figure 7.3 Displaying files with directories using “ls -l” command.

Cat command is used to view the content in the file.



### Figure 7.4 Displaying content in the file using “cat” command

The “gcc” command is used to compile .c files.



**Figure 7.5 Compiling “.c file” using “gcc” command in our terminal.**

# SYSTEM TESTING

Software testing can be stated as the process of verifying and validating that a software or application is bug free, meets the technical requirements as guided by its design and development and meets the user requirements effectively and efficiently with handling all the exceptional and boundary cases.

Software testing is defined as an activity to check whether the actual results match the expected results and to ensure that the software system is Defect free. It involves execution of a software component or system component to evaluate one or more properties of interest. Software testing also helps to identify errors, gaps, or missing requirements in contrary to the actual requirements. It can be either done manually or using automated tools. Some prefer Software testing as a White Box and Black Box Testing. In simple terms, Software Testing means Verification of Application Under Test (AUT).

The process of software testing aims not only at finding faults in the existing software but also at finding measures to improve the software in terms of efficiency, accuracy, and usability. It mainly aims at measuring specification, functionality and performance of a software program or application.

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks at implementation of the software. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs.

The main aim of software testing is:

* + - Meets the business and technical requirements that guided its design and development.
    - Works as expected.
    - Can be implemented with the same characteristics.

Software testing, depending on the testing method employed, can be implemented at any time in the development process. However, most of the test effort occurs after the requirements have been defined and the coding process has been completed. As such, the methodology of the test is governed by the software development methodology adopted.

### Importance of Testing

Testing is important because software bugs can be expensive or even dangerous. Software bugs can potentially cause monetary and human loss, and history is full of such examples.

* In April 2015, Bloomberg terminal in London crashed due to software glitch affected more than 300,000 traders on financial markets. It forced the government to postpone a 3bn pound debt sale.
* Nissan cars must recall over 1 million cars from the market due to software failure in the airbag sensory detectors. There has been reported two accidents due to this software failure.
* Starbucks was forced to close about 60 percent of stores in the U.S and Canada due to software failure in its POS system. At one point the store served coffee for free as they were unable to process the transaction.
* Some of Amazon’s third-party retailers saw their product price reduced to 1p due to a software glitch. They were left with heavy losses.
* Vulnerability in Window 10. This bug enables users to escape from security sandboxes through a flaw in the win32k system.
* In 2015 fighter plane F-35 fell victim to a software bug, making it unable to detect targets correctly.
* China Airlines Airbus A300 crashed due to a software bug on April 26, 1994, killing 264 innocents live
* In 1985, Canada's Therac-25 radiation therapy machine malfunctioned due to software bug and delivered lethal radiation doses to patients, leaving 3 people dead and critically injuring 3 others.
* In April of 1999, a software bug caused the failure of a $1.2 billion military satellite launch, the costliest accident in history
* In May of 1996, a software bug caused the bank accounts of 823 customers of a major U.S. bank to be credited with 920 million US dollars.

### Types of Testing:

**Unit Testing**

Unit testing involves the design of test cases that validate that the internal program logic is functioning properly, and that program inputs produce valid outputs. All decision branches and internal code flow should be validated. It is the testing of individual software units of the application. It is done after the completion of an individual unit before integration. This is a structural testing, that relies on knowledge of its construction and is invasive. Unit tests perform basic tests at component level and test a specific business process, application, and/or system

configuration. Unit tests ensure that each unique path of a business process performs accurately to the documented specifications and contains clearly defined inputs and expected results.

### Integration Testing

Integration tests are designed to test integrated software components to determine if they run as one program. Testing is event driven and is more concerned with the basic outcome of screens or fields. Integration tests demonstrate that although the components were individually satisfaction, as shown by successfully unit testing, the combination of components is correct and consistent. Integration testing is specifically aimed at exposing the problems that arise from the combination of components.

### Functional Testing

Functional tests provide systematic demonstrations that functions tested are available as specified by the business and technical requirements, system documentation, and user manuals.

Functional testing is centered on the following items:

Valid Input : Identified classes of valid input must be accepted. Invalid Input : Identified classes of invalid input must be rejected. Functions : Identified functions must be exercised.

Output : Identified classes of application outputs must be exercised. Systems/Procedures : Interfacing systems or procedures must be invoked.

### System Testing

System testing ensures that the entire integrated software system meets requirements. It tests a configuration to ensure known and predictable results. An example of system testing is the configuration-oriented system integration test. System testing is based on process descriptions and flows, emphasizing pre-driven process links and integration points.

Software testing can be divided into two steps:

1. **Verification:** it refers to the set of tasks that ensure that software correctly implements a specific function.
2. **Validation:** it refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

Verification: “Are we building the product, right?” Validation: “Are we building the right product?” What are the different types of software testing?

Software Testing can be broadly classified into two types:

1. **Manual Testing**: Manual testing includes testing software manually, i.e., without using any automated tool or any script. In this type, the tester takes over the role of an end-user and tests the software to identify any unexpected behavior or bug. There are different stages for manual testing such as unit testing, integration testing, system testing, and user acceptance testing.

Testers use test plans, test cases, or test scenarios to test software to ensure the completeness of testing. Manual testing also includes exploratory testing, as testers explore the software to identify errors in it.

1. **Automation Testing:** Automation testing, which is also known as Test Automation, is when the tester writes scripts and uses another software to test the product. This process involves automation of a manual process. Automation Testing is used to re-run the test scenarios that were performed manually, quickly, and repeatedly.

Apart from regression testing, automation testing is also used to test the application from load, performance, and stress point of view. It increases the test coverage, improves accuracy, and saves time and money in comparison to manual testing.

* 1. **TEST METHODOLOGIES**
     + **Black box Testing** is the testing process in which testers can perform testing on an application without having any internal structural knowledge of the application. Usually Test Engineers are involved in the black box testing.
     + **White box Testing** is the testing process in which tester can perform testing on an application with having internal structural knowledge. Usually, The Developers are involved in white box testing.
     + **Gray Box Testing**: is the process in which the combination of black box and whitebox testing are used.

## TEST CASES

### CASE-1:

**Input:** To begin, open the terminal and launch the programme file, then input the “ls”(list) command.

**Output:** The “ls” command, when entered, displays a list of files present in the system.

### CASE-2:

**Input:** Provide a “touch” command as input in a command line interface or terminal.

**Output:** When the “touch” command is used as an input, it will generate an empty file as an output.

### CASE-3:

**Input:** Provide a “ls -l” command as input in a command line interface or terminal.

**Output:** The output will typically display a detailed listing of files and directories in the current directory. The "ls" command is commonly used to list directory contents, and the "-l" option is used to provide a long format listing. The output of the "ls -l" command includes information such as file permissions, ownership, file size, creation/modification dates, and file or directory names.

### CASE-4:

**Input:** Provide a “cat <filename>” command as input in a command line interface or terminal. **Output:** If you enter the "cat <filename>" command as input in a command line interface or terminal, where "<filename>" is the name of an actual file, the output will display the contents of that file. The "cat" command is commonly used to concatenate and display the contents of files.

**CASE-5:**

**Input:** Provide a “gcc filename.c” command as input in a command line interface or terminal. **Output:** If you enter the "gcc filename.c" command as input in a command line interface or terminal, where "filename.c" is the name of a C source code file, the output will depend on whether the compilation process is successful or encounters any errors.

The "gcc" command is a widely used compiler for the C programming language. It is typically used to compile C source code files into executable programs.

The output might look like:

* If there are no errors in the source code and the compilation is successful, the command will generate an executable file. The output will typically be minimal or non-existent, indicating that the compilation was successful.
* If there are errors in the source code, the command will display error messages pointing to the specific issues encountered during compilation. These error messages will provide information about the nature of the errors, such as syntax errors, missing declarations, or incompatible types.

### CASE-6:

**Input:** Provide a “./a.out” command as input in a command line interface or terminal.

**Output:** If you enter the "./a.out" command as input in a command line interface or terminal, it typically executes the compiled program generated by the "gcc" command.

When you run "./a.out," it executes the compiled program and displays any output generated by that program. The specific output will depend on the functionality of the program its

# CONCLUSION

In conclusion, the implementation of context switching and Round Robin scheduling in this operating system demonstrates the importance of these mechanisms in achieving efficient multitasking and resource allocation. Context switching plays a crucial role in enabling the processor to allocate its resources to different processes, ensuring fair allocation of processor time and improved system responsiveness.

By saving the state of the current process before switching to the next one, the operating system guarantees a smooth transition and continuity of execution. Vital information such as the program counter, register values, and execution context data are preserved, preventing any loss of progress or data. This approach allows the operating system to resume the execution of processes from where they were left off, facilitating seamless multitasking and effective management of multiple processes.

Furthermore, the implemented context switch also caters to I/O interrupts, suspending the execution of the current process to handle interrupt requests. By saving the current process's context during interrupt processing, the operating system ensures that the interrupted process can resume execution once the interrupt is handled.

Overall, the incorporation of context switching for both scheduling and I/O interrupts results in efficient resource utilization, improved responsiveness, and fair allocation of processor time among various processes. These mechanisms enable the operating system to effectively manage the execution of multiple processes in a time-shared manner, promoting efficient multitasking and enhancing the overall performance of the system.

# FUTURE WORK

Here are some potential areas for further research based on the information supplied regarding the operating system's handling of context switching and Round Robin scheduling:

Performance optimization: Assess the context switching mechanism's effectiveness and pinpoint any flaws. Find ways to improve the process of storing and restoring the execution context, decreasing the overhead associated with switching contexts, and improving system performance as a whole.

While Round Robin scheduling is a well-liked option for time-sharing systems, think about investigating and putting alternate scheduling algorithms into place to accommodate various workload conditions. Look at algorithms like priority-based scheduling, shortest job next, or multi-level feedback queues that, in certain situations, could offer better resource utilization and responsiveness.

Implement preemptive context switching so that higher-priority processes can pause lower-priority ones in the middle of execution. This strategy guarantees that crucial tasks receive the requisite processing time when needed, enhancing system responsiveness and task execution according to priority.

Real-Time Task Handling: Look into how the operating system supports and uses real- time task handling. Real-time tasks frequently have stringent deadlines; therefore the operating system should have mechanisms for prioritizing their execution and keeping them on schedule.

Analyze the overhead associated with context switching in detail to determine how much time it takes. By identifying potential bottlenecks or areas for improvement, this study can help you optimize the context switching mechanism and cut down on needless overhead.

Enhance the context switching technique to handle failures and exceptions gracefully. Error Handling and Fault Tolerance. Implement recovery measures to maintain system stability and resilience.

These are just a few ideas for potential future work based on the current situation. You can opt to concentrate on one or more of these areas or investigate other pertinent issues to further improve the functionality and performance of the operating system, depending on your particular needs and objectives.

# BIBILOGRAPHY

1. "Linux Kernel Development" by Robert Love.
2. "Operating System Concepts" by Abraham Silberschatz, Peter B. Galvin, and Greg Gagne.
3. "Modern Operating Systems" by Andrew S. Tanenbaum and HerbertBos.
4. "Supporting Controlled Interaction" by Max Hailperin
5. "Operating Systems: Three Easy Pieces" by Remzi H. Arpaci-Dusseau, Andrea C.
6. <https://www.studytonight.com/operating-system/context-switching>
7. <https://www.geeksforgeeks.org/context-switching-in-operating-system/>
8. <https://www.techopedia.com/definition/27164/context-switching>
9. <https://developer.ibm.com/articles/l-context-switch-linux-kernel/>