

CS2701: Operating Systems and Lab

Pranaov S (24011103051)

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction to Basic Unix Commands | 3 |
| 1.1 | Aim | 3 |
| 1.2 | Procedure | 3 |
| 1.3 | Program (Commands Executed) | 4 |
| 1.4 | Output | 5 |
| 1.5 | Result | 7 |
| 2 | Working with Shell and Meta Characters in Unix, Writing and Executing Shell Scripts | 7 |
| 2.1 | Aim | 7 |
| 2.2 | Procedure | 7 |
| 2.3 | Program (Commands Executed) | 8 |
| 2.4 | Output | 10 |
| 2.5 | Result | 12 |
| 3 | Creation of Processes using fork() System Call | 13 |
| 3.1 | Aim | 13 |
| 3.2 | Procedure | 13 |
| 3.3 | Program | 13 |
| 3.4 | Output | 15 |
| 3.5 | Result | 16 |
| 4 | Zombie, Orphan, wait, waitpid and exec | 16 |
| 4.1 | Aim | 16 |
| 4.2 | Procedure | 16 |
| 4.3 | Program | 17 |
| 4.4 | Output | 20 |
| 4.5 | Result | 21 |
| 5 | Demonstration of Parent-Child Process Synchronization using wait() | 21 |
| 5.1 | Aim | 21 |
| 5.2 | Procedure | 21 |
| 5.3 | Program | 22 |

| | | |
|-----------|---|-----------|
| 5.4 | Output | 23 |
| 5.5 | Result | 23 |
| 6 | Unix I/O System Calls | 23 |
| 6.1 | Aim | 23 |
| 6.2 | Procedure | 23 |
| 6.3 | Program | 24 |
| 6.4 | Output | 26 |
| 6.5 | Result | 27 |
| 7 | Creation and Execution of POSIX Threads (pthreads) | 27 |
| 7.1 | Aim | 27 |
| 7.2 | Procedure | 27 |
| 7.3 | Program | 28 |
| 7.4 | Output | 35 |
| 7.5 | Result | 35 |
| 8 | Synchronization between Threads using Mutex Locks | 36 |
| 8.1 | Aim | 36 |
| 8.2 | Procedure | 36 |
| 8.3 | Program | 37 |
| 8.4 | Output | 38 |
| 8.5 | Result | 39 |
| 9 | Implementation of a Program Demonstrating Mutex and Deadlock | 39 |
| 9.1 | Aim | 39 |
| 9.2 | Procedure | 39 |
| 9.3 | Program | 40 |
| 9.4 | Output | 42 |
| 9.5 | Result | 42 |
| 10 | Experiment 10: Implementation of a Critical Section using Semaphores | 42 |
| 10.1 | Aim | 42 |
| 10.2 | Procedure | 42 |
| 10.3 | Program | 43 |
| 10.4 | Output | 44 |
| 10.5 | Result | 45 |
| 11 | Experiment 11: Investigating Deadlock and Livelock Scenarios | 45 |
| 11.1 | Aim | 45 |
| 11.2 | Procedure | 46 |
| 11.3 | Program | 47 |
| 11.4 | Output | 50 |
| 11.5 | Result | 50 |

| | |
|--|-----------|
| 12 Implementation of Interprocess Communication | 50 |
| 12.1 Aim | 50 |
| 12.2 Procedure | 51 |
| 12.3 Program | 52 |
| 12.4 Output | 55 |
| 12.5 Result | 56 |
| 13 Simulation of Scheduling Algorithms | 56 |
| 13.1 Aim | 56 |
| 13.2 Procedure | 56 |
| 13.3 Program | 57 |
| 13.4 Output | 61 |
| 13.5 Result | 62 |

1 Introduction to Basic Unix Commands

1.1 Aim

To learn and practice basic Unix/Linux commands for directory creation, file creation, and file manipulation using commands like `mkdir`, `cd`, `ls`, and `cat`.

1.2 Procedure

1. A new directory named `04012018` was created using the `mkdir` command.
2. The current working directory was changed to `04012018` using the `cd` command.
3. Two files, `Junk` and `Temp`, were created using the `ed` line editor.
 - `ed Junk` was run, followed by `a` to append text.
 - The specified content for `Junk` was typed, followed by `.` to exit append mode, and `w` to write the file and `q` to quit.
 - This process was repeated for the `Temp` file.
4. Two additional files, `file1` and `file2`, were created (e.g., using `cat > file1`) with at least five lines of text each.
5. The `ls` command was used to list all files in the directory.
6. The `ls -t` command was used to list files sorted by modification time.
7. The `ls -l` command was used to display a long listing format with detailed file information.

8. The `ls -l Junk` command was run to see the detailed information for a single file, and the output fields were analyzed.
9. The `ls -lt` command was used to demonstrate grouping options, listing files in long format, sorted by time.
10. The `ls -i` command was used to display the i-node (index) number for each file.
11. The `cat Junk` and `cat Temp` commands were used to display the contents of each file.
12. The `cat Junk Temp file1 file2` command was used to concatenate and display the contents of all files at once.
13. The `cat -n Junk` command was used to display the contents of the Junk file with line numbers.

1.3 Program (Commands Executed)

1. Create directory

```
mkdir 04012018
```

2. Change directory

```
cd 04012018
```

3. Create Junk and Temp (using a modern, non-interactive equivalent for demonstration)

```
# The 'ed' command is interactive. A common way to create  
→ files in scripts is 'echo' or 'cat'.
```

```
# Creating Junk
```

```
echo "The Unix system is full duplex: The character" > Junk  
echo "you type on the keyboard are sent to the system," >>  
→ Junk  
echo "which sends them back to the terminal to be" >> Junk  
echo "printed on the screen." >> Junk
```

```
# Creating Temp
```

```
echo "Normally, this echo" > Temp  
echo "process copies the characters directly to the" >> Temp  
echo "screen, so you can see what you are typing," >> Temp  
echo "but some times, such as when you are typing a" >> Temp  
echo "secret password, the echo is turned off so the" >> Temp  
echo "characters do not appear on the screen" >> Temp
```

4. Create two more files

```
echo "This is line 1 of file1." > file1
echo "This is line 2 of file1." >> file1
echo "This is line 3 of file1." >> file1
echo "This is line 4 of file1." >> file1
echo "This is line 5 of file1." >> file1

echo "This is line 1 of file2." > file2
echo "This is line 2 of file2." >> file2
echo "This is line 3 of file2." >> file2
echo "This is line 4 of file2." >> file2
echo "This is line 5 of file2." >> file2
```

5. List files

```
ls
```

6. List by time

```
ls -t
```

7. List in long format

```
ls -l
```

8. Long list a single file

```
ls -l Junk
```

9. List in long format, sorted by time

```
ls -lt
```

10. List with i-node numbers

```
ls -i
```

11. Display file content

```
cat Junk
cat Temp
```

12. Concatenate and display multiple files

```
cat Junk Temp file1 file2
```

13. Display file with line numbers

```
cat -n Junk
```

1.4 Output

(Note: Exact output like user/group names, file sizes, and times will vary)

```

$ mkdir 04012018
$ cd 04012018
$ ... (File creation commands run) ...
$ ls
Junk Temp file1 file2

$ ls -t
file2 file1 Temp Junk

$ ls -l
total 16
-rw-r--r-- 1 user group 148 Jan 4 12:01 Junk
-rw-r--r-- 1 user group 230 Jan 4 12:01 Temp
-rw-r--r-- 1 user group 105 Jan 4 12:02 file1
-rw-r--r-- 1 user group 105 Jan 4 12:02 file2

$ ls -l Junk
-rw-r--r-- 1 user group 148 Jan 4 12:01 Junk

```

Description of output:

```

-rw-r--r-- : File permissions (Read/Write for user, Read-only for group/others)
1 : Number of hard links
user : File owner
group : File group
148 : File size in bytes
Jan 4 12:01 : Date and time of last modification
Junk : File name

```

```

$ ls -lt
total 16
-rw-r--r-- 1 user group 105 Jan 4 12:02 file2
-rw-r--r-- 1 user group 105 Jan 4 12:02 file1
-rw-r--r-- 1 user group 230 Jan 4 12:01 Temp
-rw-r--r-- 1 user group 148 Jan 4 12:01 Junk

```

```

$ ls -i
12345 Junk 12346 Temp 12347 file1 12348 file2

```

```

$ cat Junk
The Unix system is full duplex: The character
you type on the keyboard are sent to the system,
which sends them back to the terminal to be
printed on the screen.

```

```

$ cat Temp
Normally, this echo
process copies the characters directly to the

```

screen, so you can see what you are typing, but some times, such as when you are typing a secret password, the echo is turned off so the characters do not appear on the screen

```
$ cat Junk Temp file1 file2
```

(Output shows content of all four files printed one after
→ another)

```
$ cat -n Junk
```

```
1 The Unix system is full duplex: The character  
2 you type on the keyboard are sent to the system,  
3 which sends them back to the terminal to be  
4 printed on the screen.
```

1.5 Result

Basic Unix commands for directory and file management (`mkdir`, `cd`), listing (`ls` with options `-t`, `-l`, `-i`), and file viewing (`cat` with option `-n`) were successfully executed and their outputs were observed.

2 Working with Shell and Meta Characters in Unix, Writing and Executing Shell Scripts

2.1 Aim

To learn how to write, make executable, and run a basic shell script. To practice using meta-characters and powerful Unix utilities like `find`, `grep`, `wc`, `sort`, and `chmod` for file system exploration, content inspection, and permission management.

2.2 Procedure

1. A new directory was created using my registration number and branch:

```
mkdir 24011103051_cyber
```
2. The current directory was changed into this new folder:

```
cd 24011103051_cyber
```
3. A new shell script file named `random_file_gen.sh` was created using a text editor (as specified, `gedit`, or using `cat` redirection for documentation).
4. The provided bash script content was pasted into this file.
5. The script was made executable using the `chmod +x` command.

6. The script was executed, which created a new subdirectory named `lab_random_[timestamp]`.
7. All tasks from **Part A (Directory & File Exploration)** were performed. This included listing files, counting lines/words/chars in a file, searching for content with `grep`, sorting a file's contents, and using a pipe to count logged-in users.
8. The directory was changed to the newly generated `lab_random_*` directory.
9. All tasks from **Part B (Directory & File Exploration)** were performed. This involved using `find` to count directories and files, list files with full paths and sizes, and find all `.txt` files.
10. All tasks from **Part C (Content Inspection)** were performed. This included using `grep` to find files containing a specific word, extracting specific lines from files, and using `head` and `tail` to inspect file content.
11. All tasks from **Part D (File Permissions)** were performed. This involved listing file permissions, finding files with specific (777) permissions, changing permissions for all `.txt` files to 644, and verifying the change.

2.3 Program (Commands Executed)

```
# 1. Create and enter the main directory
mkdir 24011103051_cyber
cd 24011103051_cyber

# 2. Create the shell script
# (Using cat redirection to show the content being added)
cat << 'EOF' > random_file_gen.sh
#!/bin/bash

# Number of folders and files to create (adjust as needed)
NUM_DIRS=$((RANDOM % 5 + 3)) # 3 to 7 directories
NUM_FILES=$((RANDOM % 10 + 5)) # 5 to 14 files

# Base directory
BASE_DIR="./lab_random_$(date +%s)"
mkdir "$BASE_DIR"
cd "$BASE_DIR" || exit

echo "Creating $NUM_DIRS directories inside $BASE_DIR..."

# Generate random directories
for ((i = 1; i <= NUM_DIRS; i++)); do
    DIR="dir_$RANDOM"
    mkdir "$DIR"
done
```

```

echo "Creating $NUM_FILES random files..."

# Random strings
random_string() {
    tr -dc A-Za-z0-9 </dev/urandom | head -c 12
}

# Get list of created directories
DIRS=$(ls -d */)

# Create random files and insert content
for ((j = 1; j <= NUM_FILES; j++)); do
    RAND_DIR=${DIRS[$((RANDOM % NUM_DIRS))]}
    FILENAME="file_$RANDOM.txt"
    FULL_PATH="${RAND_DIR}${FILENAME}"
    echo "Creating file: $FULL_PATH"
    {
        echo "PWD: $(pwd)/$RAND_DIR"
        echo "WHO:"
        who
        echo "DATE: $(date)"
        echo "RANDOM STRING: $(random_string)"
    } > "$FULL_PATH"

    # Apply random chmod (e.g., 644, 600, 755, 777)
    MODES=("600" "644" "755" "777" "700")
    MODE=${MODES[$((RANDOM % ${#MODES[@]}))]}
    chmod "$MODE" "$FULL_PATH"
    echo "Applied chmod $MODE to $FULL_PATH"
done

echo "All done. Explore: $BASE_DIR"
EOF

# 3. Make the script executable
chmod +x random_file_gen.sh

# 4. Run the script
./random_file_gen.sh

# --- Part A: Directory & File Exploration ---
ls -la lab_random_1730528400/ # Example timestamp
ls -R lab_random_1730528400/
echo "Hello world, this is a test." > testfile.txt
wc testfile.txt

```

```

grep main testfile.txt
echo -e "5\n10\n1\n3\n8" > numbers.txt
sort -n numbers.txt
who | wc -l

# --- Part B: Directory & File Exploration ---
cd lab_random_1730528400/
find . -type d | wc -l
find . -type f | wc -l
find . -type f -exec ls -lh {} \;
find . -name "*.txt"

# --- Part C: Content Inspection ---
grep -ril "RANDOM" .
grep "RANDOM STRING:" ./**
cat dir_12345/file_67890.txt
head -n 3 dir_12345/file_67890.txt
tail -n 3 dir_12345/file_67890.txt

# --- Part D: File Permissions ---
find . -type f -exec ls -l {} \;
find . -type f -perm 0777
find . -name "*.txt" -exec chmod 644 {} \;
find . -name "*.txt" -exec ls -l {} \;

```

2.4 Output

```

$ mkdir 24011103051_cyber
$ cd 24011103051_cyber
$ chmod +x random_file_gen.sh
$ ./random_file_gen.sh

Creating 5 directories inside ./lab_random_1730528400...
Creating 12 random files...
Creating file: dir_14234/file_19876.txt
Applied chmod 644 to dir_14234/file_19876.txt
... (more files created) ...
Creating file: dir_5432/file_11223.txt
Applied chmod 777 to dir_5432/file_11223.txt
All done. Explore: ./lab_random_1730528400

```

2.4.1 Part A Output

```

$ ls -la lab_random_1730528400/
. .. dir_14234 dir_22345 dir_33456 dir_5432 dir_9876

$ ls -R lab_random_1730528400/

```

```
lab_random_1730528400/:
dir_14234 dir_22345 dir_33456 dir_5432 dir_9876
... (lists contents of each subdirectory) ...
```

```
$ wc testfile.txt
1 6 29 testfile.txt
```

```
$ grep main testfile.txt
(No output)
```

```
$ sort -n numbers.txt
1
3
5
8
10
```

```
$ who | wc -l
1
```

2.4.2 Part B Output

```
$ cd lab_random_1730528400/
$ find . -type d | wc -l
6 # (5 created + current dir '.')
```

```
$ find . -type f | wc -l
12 # (12 files created)
```

```
$ find . -type f -exec ls -lh {} \;
-rw-r--r-- 1 user user 120 Nov 2 17:40 ./dir_14234/file_19876.txt
-rwx--x--x 1 user user 118 Nov 2 17:40 ./dir_22345/file_55443.txt
-rwxrwxrwx 1 user user 122 Nov 2 17:40 ./dir_5432/file_11223.txt
... (and so on)
```

```
$ find . -name "*.txt"
./dir_14234/file_19876.txt
./dir_22345/file_55443.txt
... (and so on)
```

2.4.3 Part C Output

```
$ grep -ril "RANDOM" .
./dir_14234/file_19876.txt
./dir_22345/file_55443.txt
... (lists all 12 files)
```

```
$ grep "RANDOM STRING:" ./**
./dir_14234/file_19876.txt:RANDOM STRING: aBc123XyZ456
./dir_22345/file_55443.txt:RANDOM STRING: PqR789MnB098
... (and so on)

$ cat dir_14234/file_19876.txt
PWD: /home/user/24011103051_cyber/lab_random_1730528400/dir_14234
WHO:
user tty1 2025-11-02 17:30
DATE: Sun Nov 2 17:40:00 IST 2025
RANDOM STRING: aBc123XyZ456

$ head -n 3 dir_14234/file_19876.txt
PWD: /home/user/24011103051_cyber/lab_random_1730528400/dir_14234
WHO:
user tty1 2025-11-02 17:30

$ tail -n 3 dir_14234/file_19876.txt
user tty1 2025-11-02 17:30
DATE: Sun Nov 2 17:40:00 IST 2025
RANDOM STRING: aBc123XyZ456
```

2.4.4 Part D Output

```
$ find . -type f -exec ls -l {} \;
-rw-r--r-- 1 user user 120 Nov 2 17:40 ./dir_14234/file_19876.txt
-rwx--x--x 1 user user 118 Nov 2 17:40 ./dir_22345/file_55443.txt
-rwxrwxrwx 1 user user 122 Nov 2 17:40 ./dir_5432/file_11223.txt
... (rest of the files)

$ find . -type f -perm 0777
./dir_5432/file_11223.txt
... (any other files that randomly got 777)

$ find . -name "*.txt" -exec chmod 644 {} \;
(No output)

$ find . -name "*.txt" -exec ls -l {} \;
-rw-r--r-- 1 user user 120 Nov 2 17:40 ./dir_14234/file_19876.txt
-rw-r--r-- 1 user user 118 Nov 2 17:40 ./dir_22345/file_55443.txt
-rw-r--r-- 1 user user 122 Nov 2 17:40 ./dir_5432/file_11223.txt
... (all files now show -rw-r--r--)
```

2.5 Result

A shell script was successfully created, made executable, and run. This script generated a random directory structure and files. All specified commands for

file searching (`find`), content inspection (`grep`, `cat`, `head`, `tail`), file analysis (`wc`, `sort`), and permission management (`chmod`) were executed successfully.

3 Creation of Processes using `fork()` System Call

3.1 Aim

To understand and implement process creation in Unix/Linux using the `fork()` system call. To demonstrate how to get process IDs, differentiate between parent and child processes, and create process chains and process fans.

3.2 Procedure

1. Program A: A C program was written to display the current Process ID (PID) and Parent Process ID (PPID) using the `getpid()` and `getppid()` system calls.
2. Program B: A second C program was written to demonstrate the `fork()` system call. The program uses the integer return value of `fork()` to identify whether the current process is the parent (returns child's PID) or the child (returns 0) and prints a different message for each.
3. Program C: A third C program was written to create a "process chain." This program accepts a number `n` as a command-line argument and creates a chain of `n` processes, where each child process becomes the parent of the next one.
4. Program D: A fourth C program was written to create a "process fan." This program also takes `n` as a command-line argument and has the original parent process create `n-1` direct children.
5. Each program was compiled using `gcc` (e.g., `gcc basic_fork.c -o basic_fork`).
6. Each compiled program was executed from the terminal, and the outputs were observed to verify the parent-child relationships and process structures.

3.3 Program

3.3.1 Program A: Get Process ID and Parent Process ID

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void) {
    printf("I am process %ld\n", (long)getpid());
    printf("My parent is %ld\n", (long)getppid());
    return 0;
}
```

3.3.2 Program B: Basic fork() with Parent/Child Distinction

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void) {
    pid_t childpid;
    childpid = fork();
    if (childpid == -1) {
        perror("Failed to fork");
        return 1;
    }
    if (childpid == 0) {
        /* child code */
        printf("I am child %ld, my parent is %ld\n",
→ (long)getpid(), (long)getppid());
    } else {
        /* parent code */
        printf("I am parent %ld, I created child %ld\n",
→ (long)getpid(), (long)childpid);
    }
    return 0;
}
```

3.3.3 Program C: Creating a Chain of n Processes

```
#include <stdio.h>
#include <stdlib.h> // For atoi()
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t childpid = 0;
    int i, n;

    if (argc != 2) { /* check for valid number of command-line
→ arguments */
        fprintf(stderr, "Usage: %s processes\n", argv[0]);
        return 1;
    }

    n = atoi(argv[1]); // number of processes

    for (i = 1; i < n; i++)
        if (childpid = fork())
```

```

        break; // Parent breaks, only child continues to loop
        ↪ and fork again

    fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child
    ↪ ID:%ld\n", i, (long)getpid(), (long)getppid(),
    ↪ (long)childpid);
    return 0;
}

```

3.3.4 Program D: Creating a Fan of n Processes

```

#include <stdio.h>
#include <stdlib.h> // For atoi()
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t childpid = 0;
    int i, n;

    if (argc != 2) { /* check for valid number of command-line
    ↪ arguments */
        fprintf(stderr, "Usage: %s processes\n", argv[0]);
        return 1;
    }

    n = atoi(argv[1]); // number of processes

    for (i = 1; i < n; i++)
        if ((childpid = fork()) <= 0)
            break; // Child breaks, parent continues to loop and
            ↪ fork again

    fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child
    ↪ ID:%ld\n", i, (long)getpid(), (long)getppid(),
    ↪ (long)childpid);
    return 0;
}

```

3.4 Output

Output for Program A:

```

$ gcc getpid_example.c -o getpid_example
$ ./getpid_example
I am process 6102
My parent is 4050

```

Output for Program B:

```
$ gcc basic_fork.c -o basic_fork
$ ./basic_fork
I am parent 6104, I created child 6105
I am child 6105, my parent is 6104
```

Output for Program C (Process Chain, with n=4):

```
$ gcc process_chain.c -o process_chain
$ ./process_chain 4
i:1 process ID:6107 parent ID:4050 child ID:6108
i:2 process ID:6108 parent ID:6107 child ID:6109
i:3 process ID:6109 parent ID:6108 child ID:6110
i:4 process ID:6110 parent ID:6109 child ID:0
```

Output for Program D (Process Fan, with n=4):

```
$ gcc process_fan.c -o process_fan
$ ./process_fan 4
i:4 process ID:6112 parent ID:4050 child ID:6115
i:1 process ID:6113 parent ID:6112 child ID:0
i:2 process ID:6114 parent ID:6112 child ID:0
i:3 process ID:6115 parent ID:6112 child ID:0
```

3.5 Result

The `fork()` system call was successfully used to create new processes. The behavior of parent and child processes and the use of the `fork()` return value were observed. Process chains (where each child creates a new child) and process fans (where one parent creates multiple children) were successfully implemented and verified by checking the process ID and parent process ID of each process.

4 Zombie, Orphan, wait, waitpid and exec

4.1 Aim

To understand and demonstrate the process life cycle concepts of “Zombie” and “Orphan” processes. To implement the `wait()` system call as a means to reap terminated child processes and prevent zombies. To use the `exec1()` system call to replace a process’s image with a new program.

4.2 Procedure

1. Program A (Zombie): A C program was written to create a zombie process. The child process exits immediately, while the parent process `sleep()`s for a long duration without calling `wait()`. This allows the child’s `<defunct>` state to be observed.

2. Program B (Orphan): A C program was written to create an orphan process. The parent process exits immediately after forking, while the child process continues to run. The child's Parent Process ID (PPID) is printed before and after the parent's termination to show it has been re-parented by the `init` process (PID 1).
3. Program C (`wait()`): A C program was written to demonstrate the proper use of `wait()`. The parent process forks a child, then calls `wait(&status)`. This blocks the parent until the child terminates. The status macros (e.g., `WIFEXITED`) are used to check the child's exit status, which also prevents the child from becoming a zombie.
4. Program D (`execl()`): A C program was written to demonstrate the `exec` family of calls. The parent process forks a child. The child process then uses `execl()` to replace its own process image with the `/bin/ls` command, passing `ls`, `-l`, and `NULL` as arguments. The parent process `wait()`s for the child (now running `ls`) to complete.
5. All programs were compiled using `gcc` and executed from the terminal. The `ps` command was used in a separate terminal to observe the zombie process state.

4.3 Program

4.3.1 Program A: Demonstrating a Zombie Process

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t childpid;
    childpid = fork();

    if (childpid == -1) {
        perror("fork error");
        return 1;
    } else if (childpid == 0) {
        /* Child process */
        printf("I am child, my process ID=%ld\n",
→ (long)getpid());
        printf("I am child, I am exiting now.\n");
        exit(0);
    } else {
        /* Parent process */
        printf("I am parent, My PID=%ld\n", (long)getpid());
```

```

        printf("I am parent, I am sleeping for 20 seconds.\n");
        sleep(20); // Child will be a zombie during this time
        printf("I am parent, I am now waiting...\n");
        wait(NULL);
        printf("I am parent, my child has been reaped.\n");
        exit(0);
    }
}

```

4.3.2 Program B: Demonstrating an Orphan Process

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t childpid;
    childpid = fork();

    if (childpid == -1) {
        perror("fork error");
        return 1;
    } else if (childpid == 0) {
        /* Child process */
        printf("Child: PID=%ld -- PPID=%ld\n", (long)getpid(),
→ (long)getppid());
        sleep(5); // Give parent time to exit
        printf("Child (after parent exit): PID=%ld --
→ PPID=%ld\n", (long)getpid(), (long)getppid());
        exit(0);
    } else {
        /* Parent process */
        printf("Parent: PID=%ld -- PPID=%ld\n", (long)getpid(),
→ (long)getppid());
        printf("Parent: Exiting now.\n");
        exit(0); // Parent exits, orphaning the child
    }
}

```

4.3.3 Program C: Using wait() and Checking Exit Status

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

```

```

int main(void) {
    pid_t childpid, pid;
    int status;
    pid = fork();

    if (pid == 0) {
        /* Child process */
        printf("Child: Executing and exiting normally.\n");
        exit(0);
    } else {
        /* Parent process */
        childpid = wait(&status);
        if (childpid == -1)
            perror("Failed to wait for child\n");
        else if (WIFEXITED(status) && !WEXITSTATUS(status))
            printf("Parent: Child %ld terminated normally (exit
→ status 0)\n", (long)childpid);
        else if (WIFEXITED(status))
            printf("Parent: Child %ld terminated with return
→ status %d\n", (long)childpid, WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
            printf("Parent: Child %ld terminated due to uncaught
→ signal %d\n", (long)childpid, WTERMSIG(status));
        }
    return 0;
}

```

4.3.4 Program D: Using execl()

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void) {
    pid_t childpid;
    childpid = fork();

    if (childpid == -1) {
        perror("Failed to fork");
        return 1;
    }

    if (childpid == 0) {
        /* child code */

```

```

        printf("Child: I am now replacing my image with 'ls
↪ -l'\n");
        execl("/bin/ls", "ls", "-l", NULL);
        // execl only returns if an error occurs
        perror("Child failed to exec ls");
        return 1;
    }

    /* parent code */
    if (childpid != wait(NULL)) {
        perror("Parent failed to wait due to signal or error");
        return 1;
    }

    printf("Parent: Child process terminated.\n");
    return 0;
}

```

4.4 Output

4.4.1 Output for Program A (Zombie)

Terminal 1:

```

$ gcc zombie.c -o zombie && ./zombie
I am parent, My PID=8102
I am child, my process ID=8103
I am child, I am exiting now.
I am parent, I am sleeping for 20 seconds.

```

Terminal 2 (During 20s sleep):

```

$ ps -elf | grep 8103
F S UID  PID  PPID  C PRI NI ADDR SZ WCHAN  STIME TTY TIME CMD
0 Z user 8103 8102  0 80  0 -   0 -      18:10 pts/0 00:00:00
↪ [zombie] <defunct>

```

After 20 seconds (Terminal 1):

```

I am parent, I am now waiting...
I am parent, my child has been reaped.

```

4.4.2 Output for Program B (Orphan)

```

$ gcc orphan.c -o orphan && ./orphan
Parent: PID=8105 -- PPID=4050
Parent: Exiting now.
Child: PID=8106 -- PPID=8105

```

(After 5 seconds)

Child (after parent exit): PID=8106 -- PPID=1

4.4.3 Output for Program C (wait)

```
$ gcc wait_status.c -o wait_status && ./wait_status
Child: Executing and exiting normally.
Parent: Child 8108 terminated normally (exit status 0)
```

4.4.4 Output for Program D (execl)

```
$ gcc execl_example.c -o execl_example && ./execl_example
Child: I am now replacing my image with 'ls -l'
total 32
-rwxr-xr-x 1 user user 8456 Nov 2 18:12 execl_example
-rw-r--r-- 1 user user 680 Nov 2 18:11 execl_example.c
-rwxr-xr-x 1 user user 8320 Nov 2 18:10 orphan
-rw-r--r-- 1 user user 590 Nov 2 18:09 orphan.c
... (rest of 'ls -l' output) ...
Parent: Child process terminated.
```

4.5 Result

The creation and behavior of both zombie and orphan processes were successfully demonstrated and observed. The `wait()` system call was shown to be the correct mechanism for a parent to reap a terminated child and prevent zombie processes. Finally, the `execl()` function was successfully used to load and execute a new program within a child process, demonstrating how the process image is replaced.

5 Demonstration of Parent–Child Process Synchronization using `wait()`

5.1 Aim

To demonstrate how the `wait()` system call is used to synchronize the execution of a parent process with its child process, ensuring the parent waits for the child to terminate before continuing.

5.2 Procedure

1. A C program was written that utilizes the `fork()` system call to create a new child process.
2. The program's logic is structured to handle all three possible return values from `fork()`:
 - **-1 (Error):** An error message is printed to stderr.

- **0 (Child):** The child process identifies itself by printing its PID.
 - **>0 (Parent):** The parent process receives the child's PID.
3. In the parent's code block, it immediately calls `wait(NULL)`. This call blocks the parent, forcing it to pause execution and wait for the child to finish.
 4. The child process prints its message and terminates (implicitly).
 5. Once the child terminates, the `wait(NULL)` call in the parent unblocks and returns the PID of the terminated child.
 6. The parent compares the return value from `wait()` with the `childpid` it originally received from `fork()`.
 7. If they match, the parent prints a final message confirming it is the parent and identifying its child, thus demonstrating that it has successfully waited for the child to complete.
 8. The program was compiled with `gcc` and executed in a Linux terminal.

5.3 Program

```

/* Program: Parent-Child Synchronization using wait()
   Demonstrates handling all fork() return cases.
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t childpid;

    childpid = fork();

    if (childpid == -1) {
        fprintf(stderr, "Failed to fork\n");
        return 1;
    }

    if (childpid == 0) {
        /* Child process */

```

```

        printf("I am child %ld\n", (long)getpid());
    } else if (wait(NULL) != childpid) {
        /* Parent process: wait() returned an unexpected value */
        printf("A signal must have interrupted the wait!\n");
    } else {
        /* Parent process: wait() returned the expected child PID
        ↪ */
        printf("I am parent %ld with child %ld\n",
        ↪ (long)getpid(), (long)childpid);
    }

    return 0;
}

```

5.4 Output

```

$ gcc sync_wait_full.c -o sync_wait_full && ./sync_wait_full
I am child 9310
I am parent 9309 with child 9310

```

5.5 Result

The experiment successfully demonstrated process synchronization. The `wait()` system call, placed in the parent’s code block, forced the parent process to suspend its execution until the child process completed. The output “I am parent ...” is only printed after the `wait(NULL)` call returns, which happens only *after* the child process (which prints “I am child ...”) has terminated. This confirms that `wait()` acts as a synchronization point between parent and child processes.

6 Unix I/O System Calls

6.1 Aim

To understand and demonstrate the use of basic Unix I/O system calls (`open`, `read`, `write`, `close`) and file descriptor manipulation for I/O redirection using `dup2()`.

6.2 Procedure

1. Program A (Basic I/O):

- A C program was written to create and open a file named `basic_io.txt` using the `open()` system call with `O_WRONLY` | `O_CREAT` flags.

- The `write()` system call was used to write a string "Hello from Program A!" into the file.
 - The file descriptor was closed using `close()`.
 - The same file was then reopened in read-only mode (`O_RDONLY`) using `open()`.
 - The `read()` system call was used to read the content into a buffer.
 - The buffer's contents were then written to standard output (`STDOUT_FILENO`) using `write()`.
 - Finally, the file was closed again.
2. **Program B (I/O Redirection):**
 - A second C program was written to open a file named `dup2test.txt` using `open()` with write and create flags.
 - The `dup2()` system call was used to duplicate the file descriptor onto the standard output descriptor (`STDOUT_FILENO`), redirecting all future output to the file.
 - The original file descriptor was closed since `STDOUT_FILENO` now points to the file.
 - Both `write()` and `printf()` calls were used to demonstrate that all output goes to the file instead of the terminal.
 3. Both programs were compiled with `gcc` and executed. The outputs on the terminal and the contents of the generated files (`basic_io.txt`, `dup2test.txt`) were verified.

6.3 Program

6.3.1 Program A: Basic I/O (open, write, read, close)

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

int main() {
    int fd;
    char buffer[20];
    char *text = "Hello from Program A!";

    // 1. Open for writing, create if it doesn't exist
```

```

        // S_IRUSR | S_IWUSR are user read/write permissions
        fd = open("basic_io.txt", O_WRONLY | O_CREAT, S_IRUSR |
→ S_IWUSR);
        if (fd == -1) {
            perror("Failed to open for writing");
            return 1;
        }

        // 2. Write to the file
        write(fd, text, 21);
        printf("Wrote to basic_io.txt\n");

        // 3. Close the file
        close(fd);

        // 4. Open for reading
        fd = open("basic_io.txt", O_RDONLY);
        if (fd == -1) {
            perror("Failed to open for reading");
            return 1;
        }

        // 5. Read from the file
        read(fd, buffer, 21);

        // 6. Write the buffer content to standard output
        printf("Read from basic_io.txt: ");
        write(STDOUT_FILENO, buffer, 21);
        printf("\n");

        // 7. Close the file
        close(fd);

        return 0;
    }

```

6.3.2 Program B: Redirecting Standard Output using dup2()

```

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

// Define flags for file creation
#define CREATE_FLAGS (O_WRONLY | O_CREAT | O_APPEND)
#define CREATE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

```

```

int main(void) {
    int fd;

    fd = open("dup2test.txt", CREATE_FLAGS, CREATE_MODE);
    if (fd == -1) {
        perror("Failed to open dup2test.txt");
        return 1;
    }

    // Redirect standard output to the file
    if (dup2(fd, STDOUT_FILENO) == -1) {
        perror("Failed to redirect standard output");
        return 1;
    }

    // Close the original file descriptor, it is no longer needed
    close(fd);

    // This write call now goes to "dup2test.txt"
    write(STDOUT_FILENO, "OK\n", 3);

    // This printf will ALSO go to the file
    printf("This message is also in the file.\n");

    return 0;
}

```

6.4 Output

6.4.1 Output for Program A (Basic I/O)

(Terminal Output)

```

$ gcc basic_io.c -o basic_io && ./basic_io
Wrote to basic_io.txt
Read from basic_io.txt: Hello from Program A!

```

(Content of basic_io.txt)

Hello from Program A!

6.4.2 Output for Program B (I/O Redirection)

(Terminal Output — no visible output since stdout is redirected)

```

gcc redirect_dup2.c -o redirect_dup2 && ./redirect_dup2

```

(Content of dup2test.txt)

OK

This message is also in the file.

6.5 Result

The experiment successfully demonstrated the use of fundamental Unix I/O system calls (`open`, `read`, `write`, and `close`) and file descriptor manipulation through `dup2()`.

- **Program A** showed how to perform basic file I/O operations.
- **Program B** illustrated how standard output redirection works by using `dup2()` to redirect output streams to a file. Both programs behaved as expected, validating the understanding of low-level file I/O in Unix.

7 Creation and Execution of POSIX Threads (pthreads)

7.1 Aim

To demonstrate multithreaded programming using the POSIX threads (pthreads) library by creating two separate applications:

1. A multithreaded Sudoku solution validator that checks rows, columns, and 3x3 subgrids concurrently.
2. A program that uses `fork()` to create a child process and `pthread_create()` to create a thread in the parent process. The child executes `wc -l` using `execlp()`, and the parent's thread counts vowels in the same file.

7.2 Procedure

7.2.1 Program A: Sudoku Validator

1. A 9x9 Sudoku grid was defined as a global 2D array, pre-filled with a valid solution.
2. Three global flag variables (`rows_ok`, `cols_ok`, `subgrids_ok`) were defined to store validation results.
3. Three thread-handler functions were created:
 - `check_rows()` → Validates all 9 rows.
 - `check_columns()` → Validates all 9 columns.
 - `check_subgrid()` → Nine threads run this function, each checking one 3x3 subgrid.

4. Each validation function checks for duplicates and invalid numbers (outside 1–9).
5. The main function creates 11 threads (1 for rows, 1 for columns, 9 for subgrids) and joins all using `pthread_join()`.
6. If all validation flags remain 1, the Sudoku is valid.

7.2.2 Program B: `fork()` and `pthread()` File Analyzer

1. Input File Generation:

A Base64-encoded signature file was created from the student’s registration number (24011103051) using OpenSSL:

```
“bash echo “24011103051” > file.txt openssl genpkey -algorithm RSA
-out private_key.pem -pkeyopt rsa_keygen_bits:2048 openssl rsa -in pri-
vate_key.pem -pubout -out public_key.pem openssl dgst -sha256 -sign
private_key.pem -out file.sig file.txt base64 file.sig > file.sig.b64
```

2. **C Program Logic:**

- * The program accepts a filename as a command-line argument.
- * It forks a child process.
- * The **child** executes `wc -l` on the provided file using `execlp()`.
- * The **parent** creates a POSIX thread using `pthread_create()`, which counts vowels in
- * Synchronization is achieved using both `pthread_join()` and `wait()`.

3. Both programs were compiled using `-lpthread`:

```
```bash
gcc sudoku_validator.c -o sudoku_validator -lpthread
gcc file_analyzer.c -o file_analyzer -lpthread
```

## 7.3 Program

### 7.3.1 Program A: Sudoku Validator

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define SIZE 9

// clang-format off
int sudoku[9][9] = {
 {6, 2, 4, 5, 3, 9, 1, 8, 7},
 {5, 1, 9, 7, 2, 8, 6, 3, 4},
```

```

 {8, 3, 7, 6, 1, 4, 2, 9, 5},
 {1, 4, 3, 8, 6, 5, 7, 2, 9},
 {9, 5, 8, 2, 4, 7, 3, 6, 1},
 {7, 6, 2, 3, 9, 1, 4, 5, 8},
 {3, 7, 1, 9, 5, 6, 8, 4, 2},
 {4, 9, 6, 1, 8, 2, 5, 7, 3},
 {2, 8, 5, 4, 7, 3, 9, 1, 6}
};

// clang-format on

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int is_valid = 1; // shared

void exit_if_invalid(int condition) {
 if (!condition) {
 pthread_mutex_lock(&lock);
 if (is_valid) {
 is_valid = 0;
 pthread_mutex_unlock(&lock);
 printf("Sudoku solution is invalid\n");
 exit(EXIT_FAILURE);
 }
 pthread_mutex_unlock(&lock);
 }
}

void *check_rows(void *arg) {
 for (int i = 0; i < SIZE; i++) {
 int seen[SIZE] = {0};
 for (int j = 0; j < SIZE; j++) {
 int num = sudoku[i][j];
 if (num < 1 || num > 9 || seen[num - 1]++) {
 exit_if_invalid(0);
 }
 }
 }
 return NULL;
}

void *check_cols(void *arg) {
 for (int j = 0; j < SIZE; j++) {
 int seen[SIZE] = {0};
 for (int i = 0; i < SIZE; i++) {
 int num = sudoku[i][j];
 if (num < 1 || num > 9 || seen[num - 1]++) {

```

```

 exit_if_invalid(0);
 }
}
}
return NULL;
}

typedef struct {
 int startRow;
 int startCol;
} SubgridArgs;

void *check_subgrid(void *arg) {
 SubgridArgs *args = (SubgridArgs *)arg;
 int seen[SIZE] = {0};
 for (int i = args->startRow; i < args->startRow + 3; i++) {
 for (int j = args->startCol; j < args->startCol + 3; j++) {
 int num = sudoku[i][j];
 if (num < 1 || num > 9 || seen[num - 1]++) {
 exit_if_invalid(0);
 }
 }
 }
 free(arg);
 return NULL;
}

int main() {
 pthread_t threads[11];

 // rows
 pthread_create(&threads[0], NULL, check_rows, NULL);

 // columns
 pthread_create(&threads[1], NULL, check_cols, NULL);

 // 3x3 subgrid
 int thread_index = 2;
 for (int row = 0; row < SIZE; row += 3) {
 for (int col = 0; col < SIZE; col += 3) {
 SubgridArgs *args = malloc(sizeof(SubgridArgs));
 args->startRow = row;
 args->startCol = col;
 pthread_create(&threads[thread_index++], NULL,
→ check_subgrid, args);
 }
 }
}

```

```

 }

 // Join all threads
 for (int i = 0; i < 11; i++) {
 pthread_join(threads[i], NULL);
 }

 printf("Sudoku solution is valid\n");
 return EXIT_SUCCESS;
}

```

### 7.3.2 Program B: fork(), exec(), and pthread() File Analyzer

```

#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>

#define BUFFER_SIZE 4096
#define VOWEL_MAP "AEIOUaeiou"

// Structure to pass filename to thread
typedef struct {
 char *filename;
 int total_vowels;
 int vowel_counts[5]; // Vowel count
} VowelData;

int is_vowel(char c, int vowel_counts[]) {
 char lc = tolower(c);

 switch (lc) {
 case 'a':
 vowel_counts[0]++;
 return 1;
 case 'e':
 vowel_counts[1]++;
 return 1;
 case 'i':
 vowel_counts[2]++;

```

```

 return 1;
 case 'o':
 vowel_counts[3]++;
 return 1;
 case 'u':
 vowel_counts[4]++;
 return 1;
 default:
 return 0;
 }
}

// Thread function to count vowels in file
void *vowel_counter_thread(void *arg) {
 VowelData *data = (VowelData *)arg;
 char buffer[BUFFER_SIZE];
 int fd;
 ssize_t bytes_read;

 // Initialize counts
 data->total_vowels = 0;
 for (int i = 0; i < 5; i++) {
 data->vowel_counts[i] = 0;
 }

 // Open file for reading
 fd = open(data->filename, O_RDONLY);
 if (fd == -1) {
 perror("Thread - Failed to open file");
 pthread_exit(NULL);
 }

 // Read file in chunks and count vowels
 while ((bytes_read = read(fd, buffer, BUFFER_SIZE)) > 0) {
 for (int i = 0; i < bytes_read; i++) {
 data->total_vowels += is_vowel(buffer[i],
↪ data->vowel_counts);
 }
 }

 if (bytes_read == -1) {
 perror("Thread - Error reading file");
 }

 // Close on error
 if (close(fd) == -1) {

```

```

 perror("Thread - Error closing file");
}

printf("[PARENT] Found %d vowels in total:\n",
↪ data->total_vowels);
printf(" - 'a'/'A': %d occurrences\n", data->vowel_counts[0]);
printf(" - 'e'/'E': %d occurrences\n", data->vowel_counts[1]);
printf(" - 'i'/'I': %d occurrences\n", data->vowel_counts[2]);
printf(" - 'o'/'O': %d occurrences\n", data->vowel_counts[3]);
printf(" - 'u'/'U': %d occurrences\n", data->vowel_counts[4]);

pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
 pid_t child_pid;
 int status;
 pthread_t tid;
 VowelData vowel_data;
 time_t current_time;
 char time_string[100];

 // Check arguments
 if (argc != 2) {
 fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
 return EXIT_FAILURE;
 }

 // Get and display current time
 current_time = time(NULL);
 strftime(time_string, sizeof(time_string), "%Y-%m-%d %H:%M:%S",
 localtime(¤t_time));

 printf("Start time: %s\n", time_string);

 vowel_data.filename = argv[1];

 // Create child process
 printf("[PARENT] Creating child process\n");
 child_pid = fork();

 if (child_pid < 0) {
 // Fork failed
 perror("Fork failed");
 return EXIT_FAILURE;
 } else if (child_pid == 0) {

```

```

// Child process
printf("[CHILD] Process started (PID: %d)\n", getpid());
printf("[CHILD] Executing 'wc -l %s'\n\n", argv[1]);

// Execute wc command
execlp("wc", "wc", "-l", argv[1], NULL);

// if execlp failed
perror("Exec failed");
_exit(EXIT_FAILURE); // exit only child
} else {
 // Parent process
 printf("[PARENT] Child process created with PID: %d\n",
→ child_pid);
 printf("[PARENT] Creating thread to count vowels\n");

 // Create thread for vowel counting
 if (pthread_create(&tid, NULL, vowel_counter_thread,
→ &vowel_data) != 0) {
 perror("Thread creation failed");
 return EXIT_FAILURE;
 }

 // Wait for child process to complete
 printf("[PARENT] Waiting for child process to complete\n");
 if (waitpid(child_pid, &status, 0) == -1) {
 perror("Wait failed");
 return EXIT_FAILURE;
 }

 if (WIFEXITED(status)) {
 printf("\n[PARENT] Child process completed with status:
→ %d\n",
 WEXITSTATUS(status));
 } else {
 printf("\n[PARENT] Child process terminated abnormally\n");
 }

 // Wait for thread to complete
 printf("[PARENT] Waiting for vowel counter thread to
→ complete\n");
 if (pthread_join(tid, NULL) != 0) {
 perror("Thread join failed");
 return EXIT_FAILURE;
 }
}

```

```

 printf("File: %s\n", argv[1]);
}

return EXIT_SUCCESS;
}

```

## 7.4 Output

### 7.4.1 Program A

```

$ gcc -pthread -o main main.c && ./main
Sudoku solution is valid

```

### 7.4.2 Program B

(After generating file.sig.b64)

```

$ gcc -Wall -Wextra -pthread ./main.c && ./a.out

```

```

Start time: 2025-08-14 15:15:39
[PARENT] Creating child process
[PARENT] Child process created with PID: 429421
[PARENT] Creating thread to count vowels
[PARENT] Waiting for child process to complete
[CHILD] Process started (PID: 429421)
[CHILD] Executing 'wc -l /home/personal/tmp/input/file.sig.b64'

[PARENT] Found 55 vowels in total:
- 'a'/'A': 9 occurrences
- 'e'/'E': 12 occurrences
- 'i'/'I': 9 occurrences
- 'o'/'O': 10 occurrences
- 'u'/'U': 15 occurrences
5 /home/personal/tmp/input/file.sig.b64

[PARENT] Child process completed with status: 0
[PARENT] Waiting for vowel counter thread to complete
File: /home/personal/tmp/input/file.sig.b64

```

## 7.5 Result

The experiment was successful.

- **Program A** demonstrated concurrent validation of a Sudoku grid using 11 threads and synchronized completion using `pthread_join()`.
- **Program B** implemented hybrid concurrency using both process (`fork()`, `execvp()`) and thread-based parallelism (`pthread_create()`). Both

programs effectively used synchronization primitives (`pthread_join()`, `wait()`) to ensure proper execution order and output correctness.

## 8 Synchronization between Threads using Mutex Locks

### 8.1 Aim

To implement a multithreaded C program where multiple worker threads update a shared counter and write to a shared log file. The program will use a `pthread_mutex_t` to protect these shared resources and prevent race conditions. The experiment will also demonstrate the effect of a race condition by running the program with and without the mutex enabled via a command-line flag.

### 8.2 Procedure

1. A global integer `counter` was initialized to 0. A global mutex `lock` was declared. A global flag `use_mutex` was set to 1 (true) by default.
2. A thread function `increment()` was created. This function receives a `FILE*` pointer as its argument.
3. Inside the thread function, a loop runs 10,000 times.
  - **Critical Section 1 (Counter):** Before incrementing the global counter, the thread checks if `use_mutex` is true. If so, it locks the mutex, increments the counter, and unlocks the mutex afterward.
4. After the loop finishes, the thread writes to the shared log file.
  - **Critical Section 2 (File I/O):** To prevent garbled output from simultaneous writes, the thread locks the mutex before writing to the file, then unlocks it afterward.
5. The main function handles command-line arguments:
  - Requires one argument for the number of threads.
  - Accepts an optional `--no-mutex` argument to disable mutex protection.
6. The main function opens `log.txt` in write mode.
7. If `use_mutex` is true, it initializes the mutex.
8. It creates and launches the specified number of threads, passing the `FILE*` pointer to each.

9. `pthread_join()` is called for all threads to wait for their completion.
10. If `use_mutex` is true, the mutex is destroyed.
11. The log file is closed, and the final value of the counter is printed to the console.
12. The program was compiled with `gcc -o race_demo race_demo.c -lpthread` and executed twice — once with and once without mutex protection.

### 8.3 Program

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int counter = 0;
pthread_mutex_t mutex;
int use_mutex = 1;
void *worker(void *arg) {
 int thread_id = *(int *)arg;
 for (int i = 0; i < 10000; i++) {
 if (use_mutex) {
 pthread_mutex_lock(&mutex);
 }
 counter++;
 if (use_mutex) {
 pthread_mutex_unlock(&mutex);
 }
 }
 if (use_mutex) {
 pthread_mutex_lock(&mutex);
 }
 FILE *log_file = fopen("log.txt", "a");
 if (log_file != NULL) {
 fprintf(log_file, "Thread %d finished, counter = %d\n",
 ↪ thread_id, counter);
 fclose(log_file);
 }
 if (use_mutex) {
 pthread_mutex_unlock(&mutex);
 }
 return NULL;
}
int main(int argc, char *argv[]) {
```

```

 if (argc < 2) {
 fprintf(stderr, "Usage: %s <num_threads> [--no-mutex]\n",
→ argv[0]);
 return 1;
 }
 int num_threads = atoi(argv[1]);
 if (argc > 2 && strcmp(argv[2], "--no-mutex") == 0) {
 use_mutex = 0;
 }
 pthread_t threads[num_threads];
 int thread_ids[num_threads];
 if (use_mutex) {
 pthread_mutex_init(&mutex, NULL);
 }
 FILE *log_file = fopen("log.txt", "w");
 if (log_file != NULL) {
 fclose(log_file);
 }
 for (int i = 0; i < num_threads; i++) {
 thread_ids[i] = i + 1;
 pthread_create(&threads[i], NULL, worker, &thread_ids[i]);
 }
 for (int i = 0; i < num_threads; i++) {
 pthread_join(threads[i], NULL);
 }
 printf("Final counter value: %d\n", counter);
 if (use_mutex) {
 pthread_mutex_destroy(&mutex);
 }
 return 0;
}

```

## 8.4 Output

```

$ gcc -pthread -o thrd program.c
$./thrd
Usage: ./thrd <num_threads> [--no-mutex]
$./thrd 9
Final counter value: 90000
$./thrd 9 --no-mutex
Final counter value: 43613

```

### Contents of log.txt (With Mutex):

```

Thread 7 finished, counter = 40118
Thread 1 finished, counter = 45915
Thread 8 finished, counter = 64461

```

```
Thread 5 finished, counter = 72831
Thread 2 finished, counter = 73993
Thread 9 finished, counter = 75608
Thread 4 finished, counter = 77398
Thread 3 finished, counter = 86270
Thread 6 finished, counter = 90000
```

#### Contents of log.txt (Without Mutex):

```
Thread 1 finished, counter = 9793
Thread 3 finished, counter = 14282
Thread 4 finished, counter = 16414
Thread 2 finished, counter = 16414
Thread 5 finished, counter = 26414
Thread 6 finished, counter = 34267
Thread 7 finished, counter = 38612
Thread 8 finished, counter = 41552
Thread 9 finished, counter = 43613
```

## 8.5 Result

The experiment successfully demonstrated synchronization using mutex locks.

1. **With Mutex:** The program produced the correct result ( $90,000 = 9 \times 10,000$ ). The mutex protected the critical sections, ensuring safe concurrent access to shared data and consistent log output.
2. **Without Mutex:** The result was incorrect and nondeterministic. The final counter was less than expected due to lost updates caused by simultaneous unprotected access. This illustrates the necessity of mutex locks in multithreaded programs.

## 9 Implementation of a Program Demonstrating Mutex and Deadlock

### 9.1 Aim

To demonstrate a **deadlock** condition in a multithreaded program. This is achieved by creating a circular wait situation where two threads attempt to acquire two mutex locks in reverse order, causing both to block indefinitely.

### 9.2 Procedure

1. A C program was written using two global `pthread_mutex_t` variables, `lock1` and `lock2`.
2. Two thread functions were created: `thread_func1` and `thread_func2`.

3. **thread\_func1:**
  - Acquires `lock1`.
  - Prints a message indicating it has acquired `lock1`.
  - Sleeps for 1 second (to allow `thread_func2` to acquire `lock2`).
  - Attempts to acquire `lock2`. Since `thread_func2` holds it, this thread blocks.
4. **thread\_func2:**
  - Acquires `lock2`.
  - Prints a message indicating it has acquired `lock2`.
  - Sleeps for 1 second (to allow `thread_func1` to acquire `lock1`).
  - Attempts to acquire `lock1`. Since `thread_func1` holds it, this thread blocks.
5. This situation creates a **circular wait**, where Thread 1 waits for `lock2` while holding `lock1`, and Thread 2 waits for `lock1` while holding `lock2`.
6. The main function initializes both mutexes using `pthread_mutex_init()`.
7. It creates and starts both threads, `t1` and `t2`.
8. The main thread calls `pthread_join()` for both threads. Since both are deadlocked, the program hangs indefinitely.
9. The program was compiled using:

```
gcc -o deadlock deadlock.c -lpthread
```

### 9.3 Program

```
// Program: Demonstrating a deadlock condition using two threads
→ and two mutex locks.
```

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
```

```

void* thread_func1(void* arg) {
 // Thread 1 locks lock1 first
 pthread_mutex_lock(&lock1);
 printf("Thread 1 acquired lock1\n");

 sleep(1); // Give Thread 2 time to acquire lock2
 printf("Thread 1 trying to acquire lock2...\n");

 pthread_mutex_lock(&lock2); // waits forever
 printf("Thread 1 acquired lock2\n");

 pthread_mutex_unlock(&lock2);
 pthread_mutex_unlock(&lock1);

 return NULL;
}

void* thread_func2(void* arg) {
 // Thread 2 locks lock2 first
 pthread_mutex_lock(&lock2);
 printf("Thread 2 acquired lock2\n");

 sleep(1); // Give Thread 1 time to acquire lock1
 printf("Thread 2 trying to acquire lock1...\n");

 pthread_mutex_lock(&lock1); // waits forever
 printf("Thread 2 acquired lock1\n");

 pthread_mutex_unlock(&lock1);
 pthread_mutex_unlock(&lock2);

 return NULL;
}

int main() {
 pthread_t t1, t2;

 pthread_mutex_init(&lock1, NULL);
 pthread_mutex_init(&lock2, NULL);

 pthread_create(&t1, NULL, thread_func1, NULL);
 pthread_create(&t2, NULL, thread_func2, NULL);

 // Main thread will block here since both threads are
 ↪ deadlocked
 pthread_join(t1, NULL);

```

```

pthread_join(t2, NULL);

pthread_mutex_destroy(&lock1);
pthread_mutex_destroy(&lock2);

return 0;
}

```

## 9.4 Output

```

$ gcc deadlock.c -o deadlock -lpthread && ./deadlock
Thread 1 acquired lock1
Thread 2 acquired lock2
Thread 1 trying to acquire lock2...
Thread 2 trying to acquire lock1...

```

*(Program hangs indefinitely and must be manually terminated, e.g., with Ctrl+C)*

## 9.5 Result

The experiment was successful. Thread 1 acquired `lock1` and Thread 2 acquired `lock2`. Each then attempted to acquire the other lock, resulting in both threads waiting indefinitely. This created a **circular wait**, a key condition for a deadlock. Both threads and the main thread (waiting on `pthread_join()`) were stuck indefinitely, clearly demonstrating a deadlock scenario.

# 10 Experiment 10: Implementation of a Critical Section using Semaphores

## 10.1 Aim

To use a basic unnamed POSIX semaphore to protect a critical section (a shared variable) from race conditions in a multithreaded program, demonstrating its use as a **binary semaphore** (mutex).

## 10.2 Procedure

1. A C program was written to demonstrate synchronization using a semaphore.
2. A global variable `var` was initialized to 0. This shared variable is accessed by two threads.
3. A global semaphore `sem` of type `sem_t` was declared.
4. The semaphore was initialized in the main function using:

```
sem_init(&sem, 0, 1);
```

- The 0 indicates it is shared among threads within the same process.
- The 1 initializes the semaphore as available, making it act as a **binary semaphore**.

5. Two thread functions were defined:

- **f1 (Increment Thread)**: Increments `var` by 1.
- **f2 (Decrement Thread)**: Decrements `var` by 1.

6. Both threads use `sem_wait(&sem)` to enter the critical section and `sem_post(&sem)` to exit.

7. Each thread loops 20 times, updating and printing the variable while sleeping for 1 second between iterations.

8. The main function creates two threads, joins them, destroys the semaphore, and prints the final value of `var`.

9. The program was compiled using:

```
gcc -o semaphore_demo semaphore_demo.c -lpthread
```

10. The program was executed to verify correct synchronization behavior.

### 10.3 Program

```
// Program: Demonstrating a critical section protected by a
↳ binary semaphore (unnamed)
// using incrementing and decrementing threads.

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

long var = 0;
sem_t sem;

void* f1(void* arg) {
 for (int i = 0; i < 20; i++) {
 sem_wait(&sem); // Enter critical section
 var = var + 1;
 fprintf(stderr, "thread1 var:%ld\n", var);
 sem_post(&sem); // Exit critical section
 sleep(1);
 }
 return NULL;
}
```

```

void* f2(void* arg) {
 for (int i = 0; i < 20; i++) {
 sem_wait(&sem); // Enter critical section
 var = var - 1;
 fprintf(stderr, "thread2 var:%ld\n", var);
 sem_post(&sem); // Exit critical section
 sleep(1);
 }
 return NULL;
}

int main() {
 pthread_t t1, t2;

 sem_init(&sem, 0, 1); // Initialize unnamed semaphore
 fprintf(stderr, "Parent starts\n");

 pthread_create(&t1, NULL, f1, NULL);
 pthread_create(&t2, NULL, f2, NULL);

 pthread_join(t1, NULL);
 pthread_join(t2, NULL);

 fprintf(stderr, "final var:%ld\n", var);
 sem_destroy(&sem);

 return 0;
}

```

## 10.4 Output

```

$ gcc semaphore_demo.c -o semaphore_demo -lpthread &&
↪ ./semaphore_demo
Parent starts
thread1 var:1
thread2 var:0
thread1 var:1
thread2 var:0
thread1 var:1
thread2 var:0
thread1 var:1
thread2 var:0
thread1 var:1
thread2 var:0
thread1 var:1
thread2 var:0
thread1 var:1
thread2 var:0

```

[illegible]

## 10.5 Result

The experiment successfully demonstrated the use of a semaphore to protect a critical section.

- By initializing the semaphore with a value of 1, it acted as a **binary semaphore** (mutex).
- The `sem_wait()` and `sem_post()` calls correctly locked and unlocked access to the shared variable.
- This prevented race conditions between the two threads.
- Since one thread performed 20 increments and the other performed 20 decrements, the final value of `var` was **0**, proving proper synchronization.

## 11 Experiment 11: Investigating Deadlock and Livelock Scenarios

### 11.1 Aim

To write and analyze two multithreaded C programs that demonstrate concurrency failures:

1. A **Deadlock** scenario — threads block indefinitely while waiting for each other's locks.
2. A **Livelock** scenario — threads remain active but make no progress due to continuous mutual yielding.

## 11.2 Procedure

### 11.2.1 Program A: Deadlock

1. Two global mutexes (`first_mutex`, `second_mutex`) were declared.
2. **Thread 1:**
  - Locks `first_mutex`.
  - Sleeps for one second.
  - Attempts to lock `second_mutex` (blocks indefinitely).
3. **Thread 2:**
  - Locks `second_mutex`.
  - Sleeps for one second.
  - Attempts to lock `first_mutex` (blocks indefinitely).
4. This creates a **circular wait** condition:
  - Thread 1 holds `first_mutex` and waits for `second_mutex`.
  - Thread 2 holds `second_mutex` and waits for `first_mutex`.
5. The main function initializes both mutexes, creates both threads, and joins them.  
Since both threads never complete, the program hangs indefinitely.

### 11.2.2 Program B: Livelock

1. Two global mutexes were declared: `first_mutex` and `second_mutex`.
2. **Thread 1:**
  - Locks `first_mutex`.
  - Attempts `pthread_mutex_trylock()` on `second_mutex`.
  - If it fails, unlocks `first_mutex`, sleeps, and retries.
3. **Thread 2:**

- Locks `second_mutex`.
  - Attempts `pthread_mutex_trylock()` on `first_mutex`.
  - If it fails, unlocks `second_mutex`, sleeps, and retries.
4. This leads to a **livelock**, where both threads repeatedly acquire and release locks without progressing, each deferring to the other.

## 11.3 Program

### 11.3.1 Program A: Deadlock

*// Demonstration of Deadlock using two mutexes and two threads*

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

void* do_work_one(void* param) {
 pthread_mutex_lock(&first_mutex);
 printf("Thread 1 acquired first_mutex\n");
 sleep(1);
 printf("Thread 1 trying to acquire second_mutex...\n");
 pthread_mutex_lock(&second_mutex); // Blocks forever
 printf("Thread 1 acquired second_mutex\n");
 pthread_mutex_unlock(&second_mutex);
 pthread_mutex_unlock(&first_mutex);
 return NULL;
}

void* do_work_two(void* param) {
 pthread_mutex_lock(&second_mutex);
 printf("Thread 2 acquired second_mutex\n");
 sleep(1);
 printf("Thread 2 trying to acquire first_mutex...\n");
 pthread_mutex_lock(&first_mutex); // Blocks forever
 printf("Thread 2 acquired first_mutex\n");
 pthread_mutex_unlock(&first_mutex);
 pthread_mutex_unlock(&second_mutex);
 return NULL;
}

int main() {
```

```

pthread_t tid1, tid2;
pthread_mutex_init(&first_mutex, NULL);
pthread_mutex_init(&second_mutex, NULL);

pthread_create(&tid1, NULL, do_work_one, NULL);
pthread_create(&tid2, NULL, do_work_two, NULL);

pthread_join(tid1, NULL);
pthread_join(tid2, NULL);

pthread_mutex_destroy(&first_mutex);
pthread_mutex_destroy(&second_mutex);
printf("Main: completed successfully.\n"); // Never reached
return 0;
}

```

### 11.3.2 Program B: Livelock

*// Demonstration of Livelock using non-blocking trylock and  
→ repeated retries*

```

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>

pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

void* do_work_one(void* param) {
 int done = 0;
 while (!done) {
 pthread_mutex_lock(&first_mutex);
 printf("Thread 1: Locked first_mutex\n");

 if (pthread_mutex_trylock(&second_mutex) == 0) {
 printf("Thread 1: Locked second_mutex\n");
 printf("Thread 1: Doing work...\n");
 sleep(1);
 pthread_mutex_unlock(&second_mutex);
 pthread_mutex_unlock(&first_mutex);
 printf("Thread 1: Released both locks\n");
 done = 1;
 } else {
 printf("Thread 1: Could not lock second_mutex,
→ retrying...\n");
 pthread_mutex_unlock(&first_mutex);

```

```

 sleep(1);
 }
}
return NULL;
}

void* do_work_two(void* param) {
 int done = 0;
 while (!done) {
 pthread_mutex_lock(&second_mutex);
 printf("Thread 2: Locked second_mutex\n");

 if (pthread_mutex_trylock(&first_mutex) == 0) {
 printf("Thread 2: Locked first_mutex\n");
 printf("Thread 2: Doing work...\n");
 sleep(1);
 pthread_mutex_unlock(&first_mutex);
 pthread_mutex_unlock(&second_mutex);
 printf("Thread 2: Released both locks\n");
 done = 1;
 } else {
 printf("Thread 2: Could not lock first_mutex,
→ retrying...\n");
 pthread_mutex_unlock(&second_mutex);
 sleep(1);
 }
 }
 return NULL;
}

int main() {
 pthread_t tid1, tid2;
 pthread_mutex_init(&first_mutex, NULL);
 pthread_mutex_init(&second_mutex, NULL);

 pthread_create(&tid1, NULL, do_work_one, NULL);
 pthread_create(&tid2, NULL, do_work_two, NULL);

 pthread_join(tid1, NULL);
 pthread_join(tid2, NULL);

 pthread_mutex_destroy(&first_mutex);
 pthread_mutex_destroy(&second_mutex);

 printf("Main: All threads completed successfully.\n");
 return 0;
}

```

```
}
```

## 11.4 Output

### 11.4.1 Output A: Deadlock

```
$ gcc deadlock.c -o deadlock -lpthread && ./deadlock
Thread 1 acquired first_mutex
Thread 2 acquired second_mutex
Thread 1 trying to acquire second_mutex...
Thread 2 trying to acquire first_mutex...
```

(Program hangs indefinitely — must be terminated manually.)

### 11.4.2 Output B: Livelock

```
$ gcc livelock.c -o livelock -lpthread && ./livelock
Thread 1: Locked first_mutex
Thread 2: Locked second_mutex
Thread 1: Could not lock second_mutex, retrying...
Thread 2: Could not lock first_mutex, retrying...
Thread 1: Locked first_mutex
Thread 2: Locked second_mutex
Thread 1: Could not lock second_mutex, retrying...
Thread 2: Could not lock first_mutex, retrying...
...
```

(This pattern continues indefinitely; both threads remain active but make no progress.)

## 11.5 Result

1. **Deadlock:** The program successfully demonstrated a circular wait deadlock. Thread 1 and Thread 2 each held one mutex and waited for the other, causing the entire program to halt indefinitely.
2. **Livelock:** The livelock example showed threads that actively released and reacquired locks but never progressed. Both threads continuously yielded, demonstrating how livelocks differ from deadlocks — the program remains responsive but unproductive.

## 12 Implementation of Interprocess Communication

### 12.1 Aim

To implement and demonstrate three different methods of Interprocess Communication (IPC) in C:

1. **Pipes (Anonymous):** One-way communication between a parent and child process using `pipe()`.
2. **Message Queues (System V):** Message-based communication using `msgget()`, `msgsnd()`, and `msgrcv()`.
3. **Shared Memory (System V):** High-speed communication between processes using `shmget()` and `shmat()`.

## 12.2 Procedure

### 12.2.1 Part A: IPC using Pipes

1. A simple anonymous pipe was demonstrated using `pipe(fd)`.
2. The process was forked using `fork()`.
3. **Child Process:** Writes a string "Hello, world!" to the pipe's write end using `write(fd[1], ...)`.
4. **Parent Process:** Reads the string from the read end using `read(fd[0], ...)` and prints it.
5. The program was compiled and executed to verify successful communication.

### 12.2.2 Part B: IPC using Message Queues

1. Two programs were written — a **Writer** and a **Reader**.
2. A common structure was used:

```

struct mesg_buffer {
 long mesg_type;
 char mesg_text[20];
};

```
3. **Writer:**
  - Generates a key using `ftok("msgrcv.c", 65)`.
  - Creates or connects to a message queue using `msgget()`.
  - Sends a message using `msgsnd()`.
4. **Reader:**
  - Uses the same `ftok()` key.
  - Reads messages using `msgrcv()`.
  - Destroys the queue using `msgctl(..., IPC_RMID, NULL)`.
5. Writer and Reader were compiled and run in separate terminals.

### 12.2.3 Part C: IPC using Shared Memory

1. Two programs were written — **Writer** and **Reader**.
2. Both used `ftok("shm_recv.c", 8)` to generate the same key.
3. **Writer:**
  - Created a shared memory segment using `shmget()`.
  - Attached it with `shmat()`.
  - Wrote an integer value (`*svar = 100`).
  - Detached the segment using `shmdt()`.
4. **Reader:**
  - Accessed the same memory segment using `shmget()` and `shmat()`.
  - Read the stored integer.
  - Detached the segment after reading.
5. The Writer was run first, followed by the Reader.

## 12.3 Program

### 12.3.1 Program A: Pipes

*// Program: Simple IPC using pipe() and fork()*

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void) {
 int fd[2], nbytes;
 pid_t childpid;
 char string[] = "Hello, world!\n";
 char readbuffer[80];

 pipe(fd);

 if ((childpid = fork()) == -1) {
 perror("fork");
 return 1;
 }

 if (childpid == 0) {
 // Child process: writes to pipe
 write(fd[1], string, sizeof(string));
 return 0;
 } else {
 // Parent process: reads from pipe
```

```

 nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
 printf("Received string: %s", readbuffer);
 }

 return 0;
}

```

### 12.3.2 Program B: Message Queues

*// Program: msg\_writer.c (Writer)*

```

#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

struct msg_buffer {
 long msg_type;
 char msg_text[20];
} message;

int main() {
 key_t key;
 int msgid;

 key = ftok("msgrcv.c", 65);
 msgid = msgget(key, 0600 | IPC_CREAT);

 printf("Message Queue ID: %d\n", msgid);
 message.msg_type = 1;
 printf("Write Data : ");
 fgets(message.msg_text, sizeof(message.msg_text), stdin);

 msgsnd(msgid, &message, sizeof(message), 0);
 printf("Data sent: %s\n", message.msg_text);

 return 0;
}

```

*// Program: msg\_reader.c (Reader)*

```

#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msg_buffer {

```

```

 long mesg_type;
 char mesg_text[20];
 } message;

 int main() {
 key_t key;
 int msgid;

 key = ftok("msgrcv.c", 65);
 msgid = msgget(key, 0600 | IPC_CREAT);

 printf("Message Queue ID: %d\n", msgid);
 msgrcv(msgid, &message, sizeof(message), 1, 0);

 printf("Data Received: %s\n", message.mesg_text);
 msgctl(msgid, IPC_RMID, NULL); // Destroy message queue

 return 0;
 }

```

### 12.3.3 Program C: Shared Memory

*// Program: shm\_writer.c (Writer)*

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main() {
 int id;
 int *svar;
 key_t key;

 key = ftok("shm_recv.c", 8);
 id = shmget(key, 128, 0642 | IPC_CREAT);
 printf("shm id: %d\n", id);

 svar = (int *)shmat(id, NULL, 0);
 *svar = 100;
 printf("Wrote 100 to shared memory.\n");

 shmdt(svar);
 return 0;
}

```

*// Program: shm\_reader.c (Reader)*

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main() {
 int id;
 int *rvar;
 key_t key;

 key = ftok("shm_recv.c", 8);
 id = shmget(key, 128, 0642 | IPC_CREAT);
 printf("shm id: %d\n", id);

 rvar = (int *)shmat(id, NULL, 0);
 printf("Shared value is: %d\n", *rvar);

 shmdt(rvar);
 return 0;
}

```

## 12.4 Output

### 12.4.1 Output A: Pipes

```

$ gcc pipe_demo.c -o pipe_demo && ./pipe_demo
Received string: Hello, world!

```

### 12.4.2 Output B: Message Queues

Terminal 1 (Reader):

```

$ gcc msg_reader.c -o reader && ./reader
Message Queue ID: 12345
(Waiting for message...)
Data Received: Hello IPC

```

Terminal 2 (Writer):

```

$ gcc msg_writer.c -o writer && ./writer
Message Queue ID: 12345
Write Data : Hello IPC
Data sent: Hello IPC

```

### 12.4.3 Output C: Shared Memory

Terminal 1 (Writer):

```
$ gcc shm_writer.c -o writer && ./writer
shm id: 54321
Wrote 100 to shared memory.
```

**Terminal 2 (Reader):**

```
$ gcc shm_reader.c -o reader && ./reader
shm id: 54321
Shared value is: 100
```

## 12.5 Result

All three IPC mechanisms were successfully implemented:

1. **Pipes:** Enabled unidirectional communication between related processes.
2. **Message Queues:** Allowed message-based communication between unrelated processes with queue persistence.
3. **Shared Memory:** Provided the fastest interprocess data exchange by directly sharing memory between processes.

# 13 Simulation of Scheduling Algorithms

## 13.1 Aim

To implement, simulate, and compare two preemptive CPU scheduling algorithms in C:

1. **UNIX-style Dynamic Priority Scheduling:** Demonstrates how process priorities dynamically change based on CPU usage.
2. **Round Robin with Shortest Job First (SJF):** Implements a hybrid preemptive scheduling approach that combines time quantum and shortest remaining burst time.

## 13.2 Procedure

### 13.2.1 Part 1: UNIX Dynamic Priority Scheduling

1. A **Process** structure was created to store process attributes: PID, burst time, remaining time, base priority, CPU usage, and completion time.
2. Time quantum was set to **2**.
3. Dynamic priority was calculated as:  
**Dynamic Priority = Base Priority + (CPU Usage / 2)**
4. At every scheduling cycle:
  - Priorities were updated.

- The process with the **lowest dynamic priority value** (highest priority) was selected.
  - It executed for one quantum or until completion.
5. After each completion, **Turnaround Time (TAT)** and **Waiting Time (WT)** were calculated as:
    - **TAT = Completion Time** (since all processes arrive at time 0)
    - **WT = TAT - Burst Time**
  6. The simulation printed a Gantt chart and completion stats for each process.

### 13.2.2 Part 2: Round Robin with SJF

1. Modified the scheduler to implement **Round Robin + SJF** logic.
2. Time quantum was set to **3**.
3. Instead of priority-based selection, the scheduler:
  - Maintains a static variable to track the last scheduled process.
  - Selects the process with the **shortest remaining burst time** starting from that index.
4. The scheduler preemptively runs each selected process for up to one time quantum.
5. On completion, **TAT** and **WT** are calculated and printed.

## 13.3 Program

### 13.3.1 Program A: UNIX Dynamic Priority Scheduling

*// Program: Simulation of UNIX-style Dynamic Priority Scheduling*

```
#include <stdio.h>
#define MAX_PROCESSES 10
#define TIME_QUANTUM 2

typedef struct {
 int pid;
 int burst_time;
 int remaining;
 int base_priority;
 int dyn_priority;
 int cpu_usage;
 int completed;
 int completion_time;
```

```

} Process;

void update_priority(Process p[], int n) {
 for (int i = 0; i < n; i++) {
 if (!p[i].completed)
 p[i].dyn_priority = p[i].base_priority +
→ p[i].cpu_usage / 2;
 }
}

int select_process(Process p[], int n) {
 int best = -1;
 for (int i = 0; i < n; i++) {
 if (!p[i].completed) {
 if (best == -1 || p[i].dyn_priority <
→ p[best].dyn_priority)
 best = i;
 }
 }
 return best;
}

int main() {
 Process p[MAX_PROCESSES];
 int n;

 printf("Enter number of processes: ");
 scanf("%d", &n);

 for (int i = 0; i < n; i++) {
 p[i].pid = i + 1;
 printf("Enter burst time and base priority (nice value)
→ for P%d: ", p[i].pid);
 scanf("%d %d", &p[i].burst_time, &p[i].base_priority);
 p[i].remaining = p[i].burst_time;
 p[i].cpu_usage = 0;
 p[i].completed = 0;
 }

 int time = 0, completed = 0;

 printf("\n--- CPU Scheduling Simulation (UNIX-style) ---\n");

 while (completed < n) {
 update_priority(p, n);
 int i = select_process(p, n);
 }
}

```

```

 if (i == -1) break;

 printf("Time %2d - %2d: Process P%d (priority=%d)\n",
 time,
 time + (p[i].remaining < TIME_QUANTUM ?
→ p[i].remaining : TIME_QUANTUM),
 p[i].pid, p[i].dyn_priority);

 if (p[i].remaining <= TIME_QUANTUM) {
 time += p[i].remaining;
 p[i].remaining = 0;
 p[i].completed = 1;
 p[i].completion_time = time;
 completed++;

 printf(" -> P%d finished execution.\n", p[i].pid);
 int tat = p[i].completion_time;
 int wt = tat - p[i].burst_time;
 printf(" Turnaround Time: %d, Waiting Time: %d\n",
→ tat, wt);
 } else {
 p[i].remaining -= TIME_QUANTUM;
 p[i].cpu_usage += TIME_QUANTUM;
 time += TIME_QUANTUM;
 }
 }

 printf("\nAll processes completed by time %d.\n", time);
 return 0;
}

```

### 13.3.2 Program B: Round Robin with SJF

*// Program: Simulation of Round Robin with Shortest Job First  
→ (SJF)*

```

#include <stdio.h>
#define MAX_PROCESSES 10

typedef struct {
 int pid;
 int burst_time;
 int remaining;
 int completed;
} Process;

int select_process(Process p[], int n) {

```

```

static int li = -1;
int next = -1;

for (int i = 1; i <= n; i++) {
 int id = (li + i) % n;
 if (!p[id].completed) {
 if (next == -1 || p[id].remaining <
 ↪ p[next].remaining)
 next = id;
 }
}
li = next;
return next;
}

int main() {
 Process p[MAX_PROCESSES];
 int n, tq = 3;

 printf("Enter number of processes: ");
 scanf("%d", &n);

 for (int i = 0; i < n; i++) {
 p[i].pid = i + 1;
 printf("Enter burst time and base priority for P%d: ",
 ↪ p[i].pid);
 scanf("%d %d");
 p[i].remaining = p[i].burst_time;
 p[i].completed = 0;
 }

 int time = 0, completed = 0;

 printf("\n--- CPU Scheduling Simulation (RR + SJF) ---\n");

 while (completed < n) {
 int i = select_process(p, n);
 if (i == -1) break;

 if (p[i].remaining <= tq) {
 int start = time;
 time += p[i].remaining;
 printf(" %d<---P%d--->%d -> P%d finished
 ↪ execution.\n", start, p[i].pid, time, p[i].pid);
 p[i].remaining = 0;
 p[i].completed = 1;
 }
 }
}

```

```

 completed++;

 int tat = time;
 int wt = tat - p[i].burst_time;
 printf(" Turnaround Time: %d, Waiting Time: %d\n",
→ tat, wt);
 } else {
→ tq);
 printf(" %d<----P%d---->%d\n", time, p[i].pid, time +
 p[i].remaining -= tq;
 time += tq;
 }
 }

 printf("\nAll processes completed by time %d.\n", time);
 return 0;
}

```

## 13.4 Output

### 13.4.1 Output A: UNIX Dynamic Priority Scheduling

```

Enter number of processes: 5
Enter burst time and base priority (nice value) for P1: 5 3
Enter burst time and base priority (nice value) for P2: 4 2
Enter burst time and base priority (nice value) for P3: 3 4
Enter burst time and base priority (nice value) for P4: 2 5
Enter burst time and base priority (nice value) for P5: 1 3

```

--- CPU Scheduling Simulation (UNIX-style) ---

```

Time 0 - 2: Process P2 (priority=2)
Time 2 - 4: Process P1 (priority=3)
Time 4 - 6: Process P2 (priority=3)
-> P2 finished execution.
Turnaround Time: 6, Waiting Time: 2
Time 6 - 7: Process P5 (priority=3)
-> P5 finished execution.
Turnaround Time: 7, Waiting Time: 6
Time 7 - 9: Process P1 (priority=4)
Time 9 - 11: Process P3 (priority=4)
Time 11 - 12: Process P1 (priority=5)
-> P1 finished execution.
Turnaround Time: 12, Waiting Time: 7
Time 12 - 14: Process P3 (priority=5)
-> P3 finished execution.
Turnaround Time: 14, Waiting Time: 11

```

Time 14 - 15: Process P4 (priority=5)  
-> P4 finished execution.  
Turnaround Time: 15, Waiting Time: 13

All processes completed by time 15.

### 13.4.2 Output B: Round Robin with SJF

Enter number of processes: 5  
Enter burst time and base priority (nice value) for P1: 5 3  
Enter burst time and base priority (nice value) for P2: 4 2  
Enter burst time and base priority (nice value) for P3: 3 4  
Enter burst time and base priority (nice value) for P4: 2 5  
Enter burst time and base priority (nice value) for P5: 1 3

--- CPU Scheduling Simulation (RR + SJF) ---  
0<----P5---->1 -> P5 finished execution.  
Turnaround Time: 1, Waiting Time: 0  
1<----P4---->3 -> P4 finished execution.  
Turnaround Time: 3, Waiting Time: 1  
3<----P3---->6 -> P3 finished execution.  
Turnaround Time: 6, Waiting Time: 3  
6<----P2---->9  
9<----P2---->10 -> P2 finished execution.  
Turnaround Time: 10, Waiting Time: 6  
10<----P1---->13  
13<----P1---->15 -> P1 finished execution.  
Turnaround Time: 15, Waiting Time: 10

All processes completed by time 15.

## 13.5 Result

Both CPU scheduling algorithms were successfully implemented and tested.

1. **UNIX-style Dynamic Priority:** Showed fair CPU sharing by dynamically adjusting process priorities based on CPU usage.
2. **Round Robin + SJF:** Favored shorter jobs, leading to reduced waiting times for smaller processes.

The results confirmed that dynamic priority scheduling ensures balanced fairness, while RR+SJF provides improved response time for short processes.