Operating System Laboratory Record

[Text Wrapping Break]

Department of Computer Science and Engineering[Text Wrapping Break]Shiv Nadar University Chennai

[Text Wrapping Break][Text Wrapping Break]

Name : _____

Reg. No.: _____

Semester: _____

Section _____

Subject: Operating System + Lab (CS2701)

[Text Wrapping Break][Text Wrapping Break]

Submitted to:[Text Wrapping Break]Department of Computer Science and Engineering

Exp. No.	COs	Date	Name of Experiments	Signature
1	CO1		Introduction to Basic Unix Commands	
2	CO1		Working with Shell and Meta Characters in Unix, Writing and Executing Shell Scripts	
3	CO2		Creation of Processes using `fork()` System Call	
4	CO2		Inspect Zombie and Orphan process, wait, waitpid and exec	
5	CO2		Demonstration of Parent–Child Process Synchronization using `wait()`	
6	CO2		Unix IOs	
7	CO2		Creation and Execution of POSIX Threads (pthreads)	
8	CO3		Synchronization between Threads using Mutex Locks	
9	CO3		Implementation of a Program Demonstrating Mutex and Deadlock	
10	CO3		Implementation of Critical Section using Semaphores	
11	CO3		Investigating Deadlock and Livelock Scenarios in Multithreaded Programs	
12	CO3		Implementation of Interprocess Communication using Pipes, message queue and shared memory	
13	CO3		Simulation of Scheduling algorithms	

EXPERIMENT 01 - Introduction to Basic Unix Commands

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.

Procedure

A new directory named 04012018 was created using the mkdir command.
The current working directory was changed to 04012018 using the cd command.
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.
Two additional files, file1 and file2, were created (e.g., using cat > file1) with at least five lines of text each.
The ls command was used to list all files in the directory.
The ls -t command was used to list files sorted by modification time.
The ls -l command was used to display a long listing format with detailed file information.
The ls -l Junk command was run to see the detailed information for a single file, and the output fields were analyzed.
The ls -lt command was used to demonstrate grouping options, listing files in long format, sorted by time.
The ls -i command was used to display the i-node (index) number for each file.
The cat Junk and cat Temp commands were used to display the contents of each file.
The cat Junk Temp file1 file2 command was used to concatenate and display the contents of all files at once.
The cat -n Junk command was used to display the contents of the Junk file with line numbers.

Code

```
# 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 if 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 of 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
```

Output

$ 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
# Jan 4 12:01: Date and time of last modification

```
 # Junk : File
$ 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 if 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 if 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.

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.

EXPERIMENT 02- Working with Shell and Meta Characters in Unix, Writing and Executing Shell Scripts

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.

Procedure

1. A new directory was created using my registration number and branch: mkdir 24011103037_cyber.
2. The current directory was changed into this new folder: cd 24011103037_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.

Program (Commands Executed)

```
# 1. Create and enter the main directory
mkdir 24011103037_cyber
cd 24011103037_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_$(date +%y)"
mkdir "$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_string)
  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" "707")
  MODE=${MODES[$((RANDOM % ${#MODES[@]}))]}
  chmod "$MODE" "$FULL_PATH"
  echo "Applied chmod $MODE to $FULL_PATH"
done
cd ..;  # 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 ---
# (Executed from 24011103037_cyber directory)
```

```
# 1. List all files in lab_random_* (including hidden)
ls -a lab_random_1730528400/ # (Using an example timestamp)

# 2. List all files in all subdirectories (using -R for recursive)
ls -R lab_random_1730528400/

# 3. Use wc on a file
# (Creating a dummy file first, as none exist in this dir)
echo "Hello world, this is a text." > testfile.txt
wc testfile.txt

# 4. Use grep (using the command from the prompt)
grep main testfile.txt # Will likely return no output

# 5. Sort a list of numbers
# (Creating a file with numbers first)
echo -e "50\n10\n1\n7\n3" > numbers.txt
sort -n numbers.txt

# 6. Use a pipe to count logged-in users
who | wc -l

# --- Part B: Directory & File Exploration ---

# 1. Navigate into the generated directory
cd lab_random_1730528400/

# 2. Count number of directories and files
find . -type d | wc -l
find . -type f | wc -l

# 3. List all files with full paths and sizes
find . -type f -exec ls -lh {} \;

# 4. Display all files ending with .txt
find . -name "*.txt"

# --- Part C: Content Inspection ---

# 1. Find all files containing the word "RANDOM"
grep -r "RANDOM" .

# 2. Extract and print the random string from all files
grep "RANDOM STRING:" ./*/*
```

```
# 3. Print the output of 'who' from any file
# (Using a placeholder for the random file name)
cat dir_123445/file_67890.txt

# 4. Use head and tail on any file
head -n 3 dir_123445/file_67890.txt
tail -n 2 dir_123445/file_67890.txt
tail -n 3 dir_12345/file_67890.txt

# --- Part D: File Permissions ---

# 1. List all file permissions
find . -type f -exec ls -l {} \;

# 2. Find files with 777 permissions
find . -type f -perm 0777

# 3. Change permissions of all .txt files to 644
find . -name "*.txt" -exec chmod 644 {} \;

# 4. Verify permissions again
find . -name "*.txt" -exec ls -l {} \;
```

Output

```
$ mkdir 24011103037_cyber; cd 24011103037_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_144216/file_19957.txt
Applied chmod 644 to dir_144216/file_19957.txt
Creating file dir_55432/file_11223.txt
Applied chmod 777 to dir_55432/file_11223.txt
All done. Explore: ./lab_random_1730528400

# --- Part A Output ---
$ ls -a 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
```

/lab_random_1730528400/dir_14234:
testfile.txt

/lab_random_1730528400/dir_22345:

/lab_random_1730528400/dir_33456:

/lab_random_1730528400/dir_5432:

/lab_random_1730528400/dir_9876:

$ wc testfile.txt
1 6 29 testfile.txt

$ grep main testfile.txt
(no output)

$ sort -n numbers.txt
1
3
5
7
10
50

$ who | wc -l
1

# --- Part B Output ---
$ cd lab_random_1730528400/
$ find . -type d | wc -l
6 (The 5 created + the current dir .)

$ find . -type f | wc -l
1 (only testfile.txt)

$ find . -type f -exec ls -lh {} \;
-rw-r--r-- 1 user user 120 Nov 12 17:40 ./dir_14234/file_19957.txt
-rwxrwxrwx 1 user user 120 Nov 12 17:40 ./dir_55432/file_11223.txt
(won't work for user at time) (also ls will sort all files)

# --- Part C Output ---
$ grep -r "RANDOM STRING:" ./*
./dir_14234/file_19957.txt:RANDOM STRING: aBc123Xz456

```
./dir_55432/file_11223.txt:RANDOM STRING: FgPzY98MN089
(and so on)
$ cat dir_14234/file_19876.txt
PWD: /home/user/24011103037_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/24011103037_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

# --- 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
-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--) ...
```

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.

EXPERIMENT 03 - Creation of Processes using fork() System Call

Aim: Creation of Processes using fork() System Call

Algorithm:

- ➢ Input: No external input (processes created internally).
- ➢ Output: Messages showing parent and child process' pid

- ➢ Create a new process using fork() pid ← fork()
- ➢ If pid < 0 then
  - ○ Print "fork failed"
  - ○ Exit program
- ➢ Else if pid == 0 then
  - ○ Print "Child process: PID = getpid(),
  - ○ Parent PID = getppid()"
- ➢ Else
  - ○ Print "Parent process: PID = getpid(),
  - ○ Child PID = pid"

Code:

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

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

    if (pid < 0) {
        perror("fork failed");
        return 1;
    } else if (pid == 0) {
        printf("Child process: PID = %d, Parent PID = %d\n", getpid(), getppid());
    } else {
        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
    }
    return 0;
}
```

Output:

```
prerana@apples-iMac Downloads % gcc fork.c
prerana@apples-iMac Downloads % ./a.out
Parent process: PID = 1644, Child PID = 1645
Child process: PID = 1645, Parent PID = 1
prerana@apples-iMac Downloads %
```

EXPERIMENT 04 - Inspect Zombie and Orphan process, wait, waitpid and exec

Aim: Inspect Zombie and Orphan process, wait, waitpid and exec

Algorithm:

(Zombie process)

- ➢ Input: No external input (threads increment a shared counter).
- ➢ Output: pid of Parent and Child

- ➢ Create a new process using fork() → pid ← fork()
- ➢ If pid == 0 (Child process) then
  - ○ Print "Child: Running exec (PID)"
  - ○ Replace process using execlp()
- ➢ Else (Parent process)
  - ○ Sleep for 2 seconds (allows child to become zombie briefly)
  - ○ Print "Parent: Waiting for child using waitpid"
  - ○ Invoke waitpid(pid, NULL, 0) to reap zombie
  - ○ Print "Zombie child reaped"

(Orphan process)

- ➢ Input: No external input (threads increment a shared counter).
- ➢ Output: pid of Parent and Child
- ➢ Create a new process using fork() → pid ← fork()
- ➢ If pid == 0 (Child process) then
  - ○ Sleep for 3 seconds (allow parent to exit first)
  - ○ Print "Orphan child: adopted by new parent (getppid())"
- ➢ Else (Parent process)
  - ○ Print "Parent exiting without waiting"
- ➢ If pid > 0 then
  - ○ Invoke wait(NULL) (demonstrating wait usage)

Code:

(Zombie Process)

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

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

   if (pid == 0) {
      printf("Child (PID %d): Running exec...\n", getpid());
      execlp("/bin/echo", "echo", "Hello from exec in Child!", NULL);
      printf("This wont print if exec works\n");
   }
   else {
      sleep(2);  // Ensures child becomes zombie for a moment
      printf("Parent (PID %d): Waiting for child using waitpid...\n", getpid());
      waitpid(pid, NULL, 0);  // Reaps zombie child
      printf("Parent: Reaped zombie child!\n");
```

```
    }
    return 0;
}
```

(Orphan Process)

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

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

    if (pid == 0) {
        sleep(3); // Give time for parent to exit
        printf("Orphan Child (PID %d): Adopted by %d\n", getpid(), getppid());
    }
    else {
        printf("Parent (PID %d): Exiting without wait (Child becomes orphan)\n", getpid());
    }

    if (pid > 0) {
        wait(NULL);  // wait() used just to demonstrate it
        printf("Parent used wait() before terminating\n");
    }

    return 0;
}
```

Output:

```
prerana@apples-iMac Downloads % gcc zombie.c
prerana@apples-iMac Downloads % ./a.out
Child (PID 1656): Running exec...
Hello from exec in Child!
Parent (PID 1655): Waiting for child using waitpid...
Parent: Reaped zombie child!
prerana@apples-iMac Downloads %
```

```
prerana@apples-iMac Downloads % gcc orphan.c
prerana@apples-iMac Downloads % ./a.out
Parent (PID 1666): Exiting without wait (Child becomes orphan)
Orphan Child (PID 1667): Adopted by 1666
Parent used wait() before terminating
prerana@apples-iMac Downloads %
```

EXPERIMENT 05 - Demonstration of Parent–Child Process Synchronization using wait()

Aim: Demonstration of Parent–Child Process Synchronization using wait()

Procedure:

Input: No external input (processes created internally).

Output: Messages showing parent and child process execution and synchronization.

- Create a new process using fork()
- → pid ← fork()
- If pid < 0 then
  - Print "fork failed"
  - Exit program
- Else if pid == 0 then
  - Print "Child: starting work (PID)"
  - Sleep for 2 seconds
  - Print "Child: finished work"
  - Exit child process
- Else
  - Print "Parent: waiting for child (PID)"
  - Wait for child process to finish (wait(NULL))
  - Print "Parent: child finished, continuing."

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main() {
   pid_t pid = fork();
   if (pid < 0) {
      perror("fork failed");
      return 1;
   } else if (pid == 0) {
      printf("Child: starting work (PID=%d)\n", getpid());
      sleep(2);
      printf("Child: finished work\n");
      exit(0);
   } else {
      printf("Parent: waiting for child (PID=%d)\n", pid);
      wait(NULL);
      printf("Parent: child finished, continuing.\n");
   }
   return 0;
}
```

Output:

```
prerana@apples-iMac Downloads % gcc ex5.c
prerana@apples-iMac Downloads % ./a.out
Parent: waiting for child (PID=1456)
Child: starting work (PID=1456)
Child: finished work
Parent: child finished, continuing.
prerana@apples-iMac Downloads %
```

EXPERIMENT 06 - Unix IOs

Aim: Demonstration of Unix Input and Output

Algorithm:

- Input: No external input (file operations handled internally).
- Output: Contents of the file displayed on the screen after writing and reading.

- fd ← open("sample.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR)
- If fd < 0 then
    - Print "open failed"
    - Exit program
- write(fd, "This is a sample file for UNIX I/O.\n", 36)
- lseek(fd, 0, SEEK_SET)
- n ← read(fd, buffer, sizeof(buffer) - 1)
- If n < 0 then
    - Print "read failed"
    - Close the file and exit program
- buffer[n] ← '\0'
- print("Read n bytes:\n", buffer)
- close(fd)

Code:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main() {
    int fd;
    char buffer[128];
    ssize_t n;
    /* Ensure sample.txt exists or create it */
    fd = open("sample.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    if (fd < 0) {
        perror("open");
        return 1;
    }
    /* Write a sample line */
    write(fd, "This is a sample file for UNIX I/O.\n", 36);
    /* Move to start and read */
    lseek(fd, 0, SEEK_SET);
    n = read(fd, buffer, sizeof(buffer)-1);
    if (n < 0) {
        perror("read");
        close(fd);
        return 1;
    }
    buffer[n] = '\0';
    printf("Read %zd bytes:\n%s", n, buffer);
    close(fd);
    return 0;
}
```

Output:

```
prerana@apples-iMac Downloads % gcc unixio.c
prerana@apples-iMac Downloads % ./a.out
Read 36 bytes:
This is a sample file for UNIX I/O.
prerana@apples-iMac Downloads %
```

EXPERIMENT 07 - Creation and Execution of POSIX Threads (pthreads)

Aim: Creation and Execution of POSIX Threads (pthreads)

Algorithm:

- ➢ Input: No external input (thread and message created internally).
- ➢ Output: Messages showing thread execution and main thread completion.

- ➢ pthread_t tid
- ➢ msg ← "Hello from thread"
- ➢ pthread_create(&tid, NULL, task, (void*)msg)
- ➢ If thread creation fails then
  - ○ Print "pthread_create failed"
  - ○ Exit program
- ➢ In the thread function (task):
  - ○ Print "Thread running: message = <msg>"
  - ○ Return from the thread
- ➢ In the main function,
- ➢ pthread_join(tid, NULL)
- ➢ print "Thread joined, exiting main."

Code:

```
#include <stdio.h>
#include <pthread.h>
void* task(void* arg) {
    printf("Thread running: message = %s\n", (char*)arg);
    return NULL;
}
int main() {
    pthread_t tid;
    const char *msg = "Hello from thread";
    if (pthread_create(&tid, NULL, task, (void*)msg) != 0) {
        perror("pthread_create");
        return 1;
    }
    pthread_join(tid, NULL);
    printf("Thread joined, exiting main.\n");
    return 0;
}
```

Output:

```
prerana@apples-iMac Downloads % gcc posix.c
prerana@apples-iMac Downloads % ./a.out
Thread running: message = Hello from thread
Thread joined, exiting main.
prerana@apples-iMac Downloads %
```

EXPERIMENT 08 -Synchronization between Threads using Mutex Locks

Aim: Synchronization between Threads using Mutex Locks

Algorithm:

Input: No external input (threads increment a shared counter).
Output: Final counter value (Expected: 20000).

```
counter ← 0
Initialize mutex lock

Define function increment():
    For i ← 1 to 10000 do
        Acquire lock (pthread_mutex_lock)
        counter ← counter + 1
        Release lock (pthread_mutex_unlock)
    End For
    Return
Create thread t1 to run increment()
Create thread t2 to run increment()
Wait for t1 and t2 to finish (pthread_join)
Print "Final counter value: ", counter
Destroy mutex lock.
```

Code:

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

pthread_mutex_t lock;
int counter = 0;

void* increment(void* arg) {
    for (int i = 0; i < 10000; ++i) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Final counter value: %d (expected 20000)\n", counter);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

Output:

```
prerana@apples-iMac Downloads % gcc mutex.c
prerana@apples-iMac Downloads % ./a.out
Final counter value: 20000 (expected 20000)
prerana@apples-iMac Downloads %
```

EXPERIMENT 09 - Implementation of a Program Demonstrating Mutex and Deadlock

Aim

To implement and demonstrate a **deadlock situation** using multiple threads and mutex locks in C, showing how improper lock acquisition order causes circular waiting.

Procedure

- Create two mutex locks lock1 and lock2.
- Create two threads t1 and t2.
- Let t1 lock lock1 first, then try to acquire lock2.
- Let t2 lock lock2 first, then try to acquire lock1.
- Observe that both threads are blocked indefinitely, causing deadlock.

Code

```
 #include <stdio.h>
#include <pthread.h>
#include <unistd.h>
pthread_mutex_t lock1;
pthread_mutex_t lock2;
void* thread_func1(void* arg) {
    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);
    printf("Thread 1 acquired lock2\n");
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return NULL;
}
void* thread_func2(void* arg) {
    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);
    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);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_mutex_destroy(&lock1);
    pthread_mutex_destroy(&lock2);
    return 0;
}
```

Output

```
[prerana@apples-iMac Downloads % gcc deadlock_mutex.c
[prerana@apples-iMac Downloads % ./a.out
Thread 2 acquired lock2
Thread 1 acquired lock1
Thread 1 trying to acquire lock2...
Thread 2 trying to acquire lock1...
^C
prerana@apples-iMac Downloads %
```

Result

The multithreaded C program was successfully executed with mutex protection.

EXPERIMENT 10 - Implementation of Critical Section using Semaphores

Aim : To implement and demonstrate a critical section situation using semaphores

Procedure

- **Algorithm 1: Unnamed Semaphore with Threads**

- Aim: To synchronize two threads using an unnamed semaphore that controls access to a shared variable.
- 1. Start
- 2. Initialize shared variable var = 0.
- 3. Create an unnamed semaphore sem with initial value 1.
- 4. Create Thread 1 to execute f1:
  - Repeat 20 times:
    - – Wait (decrement) semaphore.
    - – Increment var by 1.
    - – Print var.
    - – Signal (increment) semaphore.
    - – Sleep 1 second.
- 5. Create Thread 2 to execute f2:
  - Repeat 20 times:
    - – Wait (decrement) semaphore.
    - – Decrement var by 1.
    - – Print var.
    - – Signal (increment) semaphore.
    - – Sleep 1 second.
- 6. Wait for both threads to complete.
- 7. Print final value of var.
- 8. Destroy semaphore.
- 9. Stop.
- Both threads modify var safely using the unnamed semaphore to prevent race conditions.
- **Algorithm 2: Named Semaphore with Threads**
- Aim: To synchronize two threads using a named semaphore shared via the system.
- 1. Start
- 2. Initialize shared variable var = 5.
- 3. Create or open named semaphore /mysem1 with initial value 1.
- 4. Create Thread 1 to execute f1:
  - Repeat 20 times:
    - – Sleep 1 second.
    - – Wait (decrement) semaphore.
    - – Increment var by 1.
    - – Signal (increment) semaphore.
    - – Print var.
- 5. Create Thread 2 to execute f2:
  - Repeat 20 times:
    - – Sleep 1 second.
    - – Wait (decrement) semaphore.
    - – Decrement var by 1.
    - – Signal (increment) semaphore.
    - – Print var.

- 6. Wait for both threads to complete.
- 7. Print final value of var.
- 8. Close and unlink semaphore (sem_unlink("/mysem1")).
- 9. Stop.
- Both threads safely modify the shared variable using a named semaphore that can also be shared across processes.

Code

semunm_thread.c

```
#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);
        var = var + 1;
        fprintf(stderr, "thread1 var:%ld\n", var);
        sem_post(&sem);
        sleep(1);
    }
    return NULL;
}


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


int main() {
    pthread_t t1, t2;
    sem_init(&sem, 0, 1); // unnamed semaphore for threads


    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;
}
```

semnm_thread.c

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


long var = 5;
sem_t *sem;   // named semaphore for threads


void* f1(void* arg);
void* f2(void* arg);


int main() {
    // Create/open named semaphore
    sem = sem_open("/mysem1", O_CREAT, 0644, 1);
    if (sem == SEM_FAILED) {
        perror("sem_open");
        return 1;
    }


    printf("Parent starts\n");


    pthread_t t1, t2;
    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_close(sem);
    sem_unlink("/mysem1"); // remove from system


    return 0;
}


void* f1(void* arg) {
    for (long i = 0; i < 20; i++) {

        sleep(1);
        sem_wait(sem);
        var = var + 1;
        sem_post(sem);
        fprintf(stderr, "f1 var:%ld\n", var);

    }
    return NULL;
}


void* f2(void* arg) {
    for (long i = 0; i < 20; i++) {

        sleep(1);
        sem_wait(sem);
        var = var - 1;
        sem_post(sem);
        fprintf(stderr, "f2 var:%ld\n", var);

    }
    return NULL;
}
```

Output

```
prerana@apples-iMac Downloads % gcc semunm_thread.c
semunm_thread.c:38:5: warning: 'sem_init' is deprecated [-Wdeprecated-declarations]
    sem_init(&sem, 0, 1); // unnamed semaphore for threads
    ^
/Library/Developer/CommandLineTools/SDKs/MacOSX10.15.sdk/usr/include/sys/semaphore.h:55:42: note: 'sem_init' has been explicitly marked deprecated here
int sem_init(sem_t *, int, unsigned int) __deprecated;
                                         ^
/Library/Developer/CommandLineTools/SDKs/MacOSX10.15.sdk/usr/include/sys/cdefs.h:187:40: note: expanded from macro '__deprecated'
#define __deprecated    __attribute__((__deprecated__))
                                       ^
semunm_thread.c:51:5: warning: 'sem_destroy' is deprecated [-Wdeprecated-declarations]
    sem_destroy(&sem);
    ^
/Library/Developer/CommandLineTools/SDKs/MacOSX10.15.sdk/usr/include/sys/semaphore.h:53:26: note: 'sem_destroy' has been explicitly marked deprecated here
int sem_destroy(sem_t *) __deprecated;
                         ^
/Library/Developer/CommandLineTools/SDKs/MacOSX10.15.sdk/usr/include/sys/cdefs.h:187:40: note: expanded from macro '__deprecated'
#define __deprecated    __attribute__((__deprecated__))
                                       ^
2 warnings generated.
prerana@apples-iMac Downloads % ./a.out
Parent starts
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
thread2 var:-1
thread1 var:0
final var:0
prerana@apples-iMac Downloads %
```

```
[prerana@apples-iMac Downloads % gcc semnm_thread.c
[prerana@apples-iMac Downloads % ./a.out
Parent starts
f1 var:6
f2 var:5
f1 var:6
f2 var:5
f1 var:6
f2 var:5
f1 var:6
f2 var:5
f1 var:6
f2 var:5
f1 var:6
f2 var:5
f1 var:6
f2 var:5
f2 var:4
f1 var:5
f2 var:4
f1 var:5
f2 var:4
f1 var:5
f2 var:4
f1 var:5
f2 var:4
f1 var:5
f2 var:4
f1 var:5
f2 var:4
f1 var:5
f2 var:4
f1 var:5
f2 var:4
f1 var:5
f2 var:4
f1 var:5
f2 var:4
f1 var:5
f2 var:4
f1 var:5
f2 var:4
f1 var:5
final var:5
prerana@apples-iMac Downloads %
```

Result

Both names and unnamed semaphore have been implemented for the critical section successfully

EXPERIMENT 11 - Investigating Deadlock and Livelock Scenarios in Multithreaded Programs

Aim : To implement and demonstrate a deadlock and livelock situation using multiple threads.

Procedure

Procedure

- **Algorithm 1: Deadlock**
- Aim: To show how deadlock occurs when two threads wait for each other's lock.
- 1. Start
- 2. Initialize two mutexes: first_mutex, second_mutex.
- 3. Create Thread 1:
  - Lock first_mutex.
  - Wait 1 second.
  - Lock second_mutex.
  - Do some work.
  - Unlock both mutexes.
- 4. Create Thread 2:
  - Lock second_mutex.
  - Wait 1 second.
  - Lock first_mutex.
  - Do some work.
  - Unlock both mutexes.
- 5. Wait for both threads to finish.
- 6. Destroy both mutexes.
- 7. Stop.
- Result: Both threads can get stuck — Thread 1 waits for second_mutex and Thread 2
- waits for first_mutex → Deadlock.
- **Algorithm 2: Livelock 1**
- Aim: To show a livelock where both threads keep retrying without progress.
- 1. Start
- 2. Initialize two mutexes: first_mutex, second_mutex.
- 3. Create Thread 1:
  - Lock first_mutex.
  - Try to lock second_mutex:
- If successful → do work, unlock both, exit.
- If not → unlock first_mutex, wait 1 second, and retry.
- 4. Create Thread 2:
  - Lock second_mutex.
  - Try to lock first_mutex:

- If successful → do work, unlock both, exit.
- If not → unlock second_mutex, wait 1 second, and retry.
- 5. Wait for both threads to complete.
- 6. Destroy both mutexes.
- 7. Stop.
- Result: Both threads stay active but keep retrying → Livelock (no actual work done).
- **Algorithm 3: Livelock 2**
- Aim: To demonstrate livelock caused by identical retry timing.
- 1. Start
- 2. Initialize two mutexes: first_mutex, second_mutex.
- 3. Create Thread 1:
  - Lock first_mutex.
  - Try to lock second_mutex:
- If successful → do work, unlock both, exit.
- If not → unlock first_mutex, wait 1 second, and retry.
- 4. Create Thread 2:
  - Lock second_mutex.
  - Try to lock first_mutex:
- If successful → do work, unlock both, exit.
- If not → unlock second_mutex, wait 1 second, and retry.
- 5. Wait for both threads to finish.
- 6. Destroy both mutexes.
- 7. Stop.
- Result: Both threads release and retry locks at the same time → Livelock (they never
- make progress).

Code

deadlock.c

```
#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); // simulate work, gives thread 2 a chance to run
    pthread_mutex_lock(&second_mutex);
    printf("Thread 1 acquired second_mutex\n");
    printf("Thread 1 doing work...\n");
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
```

```
    pthread_exit(0);}
void *do_work_two(void *param) {
    pthread_mutex_lock(&second_mutex);
    printf("Thread 2 acquired second_mutex\n");
    sleep(1); // simulate work
    pthread_mutex_lock(&first_mutex);
    printf("Thread 2 acquired first_mutex\n");
    printf("Thread 2 doing work...\n");
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);}
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");
    return 0;}
```

Livelock1.c

```
#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) {
        // Lock first mutex
        pthread_mutex_lock(&first_mutex);
        printf("Thread 1: Locked first_mutex\n");
        // Try to lock second mutex (non-blocking)
        if (pthread_mutex_trylock(&second_mutex) == 0) {
            printf("Thread 1: Locked second_mutex\n");
            // --- Critical Section ---
            printf("Thread 1: Doing some work...\n");
            sleep(1);
            // ------------------------
            pthread_mutex_unlock(&second_mutex);
            pthread_mutex_unlock(&first_mutex);
            done = 1;  // success
            printf("Thread 1: Released both locks\n");
        } else {
            // Could not get second_mutex → release first and retry
            printf("Thread 1: Could not lock second_mutex, retrying...\n");
            pthread_mutex_unlock(&first_mutex);
            sleep(1);  // give time to other thread} }
```

```
    pthread_exit(0);}
void *do_work_two(void *param) {
    int done = 0;
    while (!done) {
        // Lock second mutex
        pthread_mutex_lock(&second_mutex);
        printf("Thread 2: Locked second_mutex\n");
        // Try to lock first mutex (non-blocking)
        if (pthread_mutex_trylock(&first_mutex) == 0) {
            printf("Thread 2: Locked first_mutex\n");
            // --- Critical Section ---
            printf("Thread 2: Doing some work...\n");
            sleep(1);
            // ----------------------
            pthread_mutex_unlock(&first_mutex);
            pthread_mutex_unlock(&second_mutex);
            done = 1;  // success
            printf("Thread 2: Released both locks\n");
        } else {
            // Could not get first_mutex → release second and retry
            printf("Thread 2: Could not lock first_mutex, retrying...\n");
            pthread_mutex_unlock(&second_mutex);
            sleep(1);
    }
    }

    pthread_exit(0);
}
int main() {
    pthread_t tid1, tid2;
    // Initialize both mutexes
    pthread_mutex_init(&first_mutex, NULL);
    pthread_mutex_init(&second_mutex, NULL);
    // Create both threads
    pthread_create(&tid1, NULL, do_work_one, NULL);
    pthread_create(&tid2, NULL, do_work_two, NULL);
    // Wait for both threads to finish
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    // Clean up
    pthread_mutex_destroy(&first_mutex);
    pthread_mutex_destroy(&second_mutex);
    printf("Main: All threads completed successfully.\n");
    return 0;
}
```

livelock2.c

```
#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) {
    while (1) {
        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 some work...\n");
            sleep(1);
            pthread_mutex_unlock(&second_mutex);
            pthread_mutex_unlock(&first_mutex);
            printf("Thread 1: released both locks\n");
            break;
        } else {
            printf("Thread 1: couldn't lock second_mutex, retrying...\n");
            pthread_mutex_unlock(&first_mutex);
            sleep(1); // both threads sleep same time → livelock likely
        }
    }
    pthread_exit(0);
}


void *do_work_two(void *param) {
    while (1) {
        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 some work...\n");
            sleep(1);
            pthread_mutex_unlock(&first_mutex);
            pthread_mutex_unlock(&second_mutex);
            printf("Thread 2: released both locks\n");
            break;
        } else {
            printf("Thread 2: couldn't lock first_mutex, retrying...\n");
            pthread_mutex_unlock(&second_mutex);
            sleep(1); // same sleep time → livelock risk
        }
    }
    pthread_exit(0);
}
int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&first_mutex, NULL);
    pthread_mutex_init(&second_mutex, NULL);
    pthread_create(&t1, NULL, do_work_one, NULL);
    pthread_create(&t2, NULL, do_work_two, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
```

```
    pthread_mutex_destroy(&first_mutex);
    pthread_mutex_destroy(&second_mutex);
    printf("Main: done.\n");
    return 0;
}
```

Output

```
prerana@apples-iMac Downloads % gcc deadlock.c
prerana@apples-iMac Downloads % ./a.out
Thread 2 acquired second_mutex
Thread 1 acquired first_mutex
```

```
prerana@apples-iMac Downloads % gcc livelock1.c
prerana@apples-iMac Downloads % ./a.out
Thread 2: Locked second_mutex
Thread 2: Could not lock first_mutex, retrying...
Thread 1: Locked first_mutex
Thread 1: Locked second_mutex
Thread 1: Doing some work...
Thread 1: Released both locks
Thread 2: Locked second_mutex
Thread 2: Locked first_mutex
Thread 2: Doing some work...
Thread 2: Released both locks
Main: All threads completed successfully.
prerana@apples-iMac Downloads %
```

```
prerana@apples-iMac Downloads % gcc livelock2.c
prerana@apples-iMac Downloads % ./a.out
Thread 1: locked first_mutex
Thread 1: couldn't lock second_mutex, retrying...
Thread 2: locked second_mutex
Thread 2: locked first_mutex
Thread 2: doing some work...
Thread 2: released both locks
Thread 1: locked first_mutex
Thread 1: locked second_mutex
Thread 1: doing some work...
Thread 1: released both locks
Main: done.
prerana@apples-iMac Downloads %
```

Result

The deadlock and livelock have been demonstrated using multiple threads.

EXPERIMENT 12 - Implementation of Inter process Communication using Pipes, message queue and shared memory

12.1 Aim

To implement **Inter-Process Communication (IPC)** using a **Message Queue** in C where one process (sender) writes a message to the queue and another process (receiver) reads it.

Procedure

- Include the required header files:
  stdio.h, sys/ipc.h, sys/msg.h, and string.h.
- Define a structure mesg_buffer containing:
  - long mesg_type
  - char mesg_text[20]
- In the sender program:
  - Generate a unique key using ftok().
  - Create or access a message queue using msgget().
  - Take input from the user.
  - Send the message to the queue using msgsnd().
- In the receiver program:
  - Use the same ftok() call to get the same key.
  - Access the same queue using msgget().
  - Receive the message using msgrcv().
  - Display the received message.
  - Remove the message queue using msgctl() with IPC_RMID.
- Compile both sender and receiver programs.
- Run the sender first, then the receiver to verify message transfer.

Code

msgsnd.c

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
// structure for message queue
struct mesg_buffer {
        long mesg_type;
        char mesg_text[20];
} message;
int main()
{
```

```
        key_t key;
        int msgid;
        // ftok to generate unique key
        key = ftok("msgrcv.c", 65);
        // msgget creates a message queue
        // and returns identifier
        msgid = msgget(key, 0600 | IPC_CREAT);
        printf("Message Queue ID: %d\n", msgid);
        message.mesg_type = 1;
        printf("Write Data : ");
        fgets(message.mesg_text, sizeof(message.mesg_text), stdin);
        // msgsnd to send message
msgsnd(msgid, &message, sizeof(message), 0);
        // display the message
printf("Data send is : %s \n", message.mesg_text);
return 0;
 }
```

msgrcv.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[20];
} message;
int main()
{
    key_t key;
    int msgid;
    // ftok to generate unique key
    key = ftok("msgrcv.c", 65);
    // msgget creates a message queue and returns identifier
    msgid = msgget(key, 0600 | IPC_CREAT);
    printf("Message Queue ID: %d\n", msgid);
    // msgrcv to receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);
    // display the message
    printf("Data Received is : %s\n", message.mesg_text);
    // to destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);
    return 0;
}
```

Output

```
prerana@apples-iMac Downloads % gcc msgsnd.c -o sender
prerana@apples-iMac Downloads % gcc msgrcv.c -o receiver
prerana@apples-iMac Downloads % ./sender
Message Queue ID: 65536
Write Data : Hello Message Queue
Data send is : Hello Message Queue
prerana@apples-iMac Downloads % ./receiver
Message Queue ID: 65536
Data Received is : Hello Message Queue
prerana@apples-iMac Downloads %
```

Result

The program was successfully executed.
The sender process sent a message to the message queue, and the receiver process retrieved it correctly, demonstrating inter-process communication using message queues.

12.2 Aim

To write a C program that demonstrates **Inter-Process Communication (IPC)** using **Shared Memory**, where one process (Sender) writes data into shared memory and another process (Receiver) reads it.

Procedure

- Create two separate C files: one for **Sender** and one for **Receiver**.
- Use ftok() to generate a common key for both processes.
- The **Sender** process creates a shared memory segment using shmget() and writes an integer value to it.
- The **Receiver** process attaches to the same memory segment using the same key and reads the stored value.
- Both processes detach from the shared memory using shmdt().
- Optionally, the memory can be deleted using shmctl() with IPC_RMID.

Code

shm_snd.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int main() {
   int id;
   int *svar;
   key_t key;
   // Generate unique key
   key = ftok("shm_recv.c", 8);
   // Create shared memory segment
   id = shmget(key, 128, 0642 | IPC_CREAT);
   printf("Shared Memory ID: %d\n", id);
   // Attach shared memory to process address space
   svar = (int *)shmat(id, NULL, 0);
   // Write data into shared memory
   *svar = 100;
   // Detach from shared memory
   shmdt(svar);
```

```
    return 0;
}
```

shm_rcv.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int main() {
    int id;
    int *rvar;
    key_t key;
    // Generate same key as sender
    key = ftok("shm_recv.c", 8);
    // Access the shared memory segment
    id = shmget(key, 128, 0642 | IPC_CREAT);
    printf("Shared Memory ID: %d\n", id);
    // Attach shared memory to process address space
    rvar = (int *)shmat(id, NULL, 0);
    // Read data from shared memory
    printf("Shared value is: %d\n", *rvar);
    // Detach from shared memory
    shmdt(rvar);
    return 0;
}
```

Output

```
[prerana@apples-iMac Downloads % gcc shm_send.c -o send
[prerana@apples-iMac Downloads % gcc shm_recv.c -o recv
[prerana@apples-iMac Downloads % ./send
[shm id:65536%
 prerana@apples-iMac Downloads % ./recv
 shm id:65536shared value is:100
 prerana@apples-iMac Downloads %
```

Result

The program successfully demonstrates **Inter-Process Communication using Shared Memory**. The **Sender process** writes a value into the shared memory, and the **Receiver process** reads the same value, confirming successful data sharing between processes.

12.3 Aim

To write a C program that demonstrates **Inter-Process Communication (IPC)** using **Shared Memory** between a **parent and child process** created with fork().

Procedure

- Create a shared memory segment using shmget().
- Attach the segment to the process address space using shmat().
- Initialize a shared variable with a value (e.g., 10).
- Create a **child process** using fork().
- In the child process, modify the shared variable after a delay.
- In the parent process, observe the shared variable before and after the child update.
- Detach the memory using shmdt() and remove it using shmctl().

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int main() {
    pid_t pid;
    int shmid, *shvar;
    key_t key = ftok(".", 45);
    // Create shared memory
    shmid = shmget(key, sizeof(int), 0664 | IPC_CREAT);
    printf("Key=%d .....Shmid=%d\n", key, shmid);
    // Attach shared memory
    shvar = shmat(shmid, NULL, 0);
    *shvar = 10;
    printf("Initial value of *shvar = %d\n", *shvar);
    pid = fork();
    if (pid == 0) {
        // Child process
        sleep(10); // simulate delay
        *shvar = *shvar + 90;
        printf("Child updated *shvar = %d\n", *shvar);
        exit(0);
    } else {
        // Parent process
        wait(NULL);  // Wait for child to complete
        printf("Parent sees *shvar (before child updates) = %d\n", *shvar);
        sleep(2); // Give child time to update
        printf("Parent sees *shvar (after delay) = %d\n", *shvar);
    }
    // Detach and remove shared memory
    shmdt(shvar);
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
```

Output

```
prerana@apples-iMac Downloads % gcc shmfork.c -o shmf
prerana@apples-iMac Downloads % ./shmf
Key=755292543 ........Shmid=65537
Initial value of *shvar = 10
Child updated *shvar = 100
Parent sees *shvar (before child updates) = 100
Parent sees *shvar (after delay) = 100
prerana@apples-iMac Downloads %
```

Result

The program successfully demonstrates **shared memory communication** between a **parent and child process**. The **child process** updates the shared variable, and the **parent process** observes the updated value through the shared memory segment.

12.4 Aim

To write a C program that demonstrates **Inter-Process Communication (IPC)** between a parent and child process using **pipes**.

Procedure

- Create a pipe using the pipe() system call.
- Use fork() to create a child process.
- The **child process** writes data into the pipe using write().
- The **parent process** reads the data from the pipe using read().
- Display the received message on the screen.
- Both processes communicate through the same pipe without using any external file.

Code

```
#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];
   // Create a pipe
   pipe(fd);

   if ((childpid = fork()) == -1)
   {
      perror("fork");
      return 1;
   }
   if (childpid == 0)
   {
      /* Child process - closes input side of pipe */
      // close(fd[0]);
      /* Send "string" through the output side of pipe */
      write(fd[1], string, sizeof(string));
      return 1;
   }
   else
   {
      /* Parent process - closes output side of pipe */
      // close(fd[1]);
      /* Read in a string from the pipe */
      nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
```

```
        printf("Received string: %s", readbuffer);
    }

    return 0;
}
```

Output

```
prerana@apples-iMac Downloads % gcc pipe3.c -o pip
prerana@apples-iMac Downloads % ./pip
Received string: Hello, world!
prerana@apples-iMac Downloads %
```

Result

The program successfully demonstrates **inter-process communication using a pipe**. The **child process** sends a string through the pipe, and the **parent process** successfully reads and displays the received message.

EXPERIMENT 13 - Simulation of Scheduling algorithms

Aim

To simulate scheduling algorithms.

Procedure

- Start the program.
- Define data structures:
- Create a structure Process with fields:
  pid, burst_time, remaining, base_priority, dyn_priority, cpu_usage, completion_time, turnaround_time, waiting_time, and completed.
- Implement update_priority() algorithm:
- Input: Array of processes p[], number of processes n.
- For each incomplete process, update the dynamic priority using the formula:
  dyn_priority = base_priority + (cpu_usage / 2).
- This simulates UNIX-style dynamic priority adjustment.
- Implement select_process() algorithm:
- Input: Array p[], number of processes n, and last_index (index of the last executed process).
- The function selects the next process using a hybrid Round Robin + Shortest Job First (SJF) approach.
- It searches circularly from (last_index + 1) and selects the incomplete process having the minimum remaining burst time.
- In the main() function:
- Read the total number of processes n.
- For each process, input Burst Time and Base Priority.
- Initialize fields: remaining = burst_time, cpu_usage = 0, and completed = 0.
- Initialize variables:
- time = 0, completed = 0, last_index = -1, time_quantum = 3.
- Create empty arrays gantt_pids[] and gantt_times[] for the Gantt chart.
- Scheduling loop:
- While all processes are not completed:
    - Update priorities using update_priority(p, n).
    - Select the next process using select_process(p, n, last_index).
    - If no process is available, exit the loop.
    - Execute the selected process for either:
        - The remaining time (if ≤ time quantum), or
        - One time quantum.
    - Update CPU usage, remaining time, and system time.
    - If a process finishes, mark it as completed and record its completion time.
    - Store execution order and timestamps in Gantt arrays.
    - Update last_index with the current process index.
- After scheduling:
- Calculate Turnaround Time (TAT) and Waiting Time (WT) for each process using:

- TAT = Completion Time
- WT = TAT − Burst Time

- Display the results:
- Print a formatted table showing:
  PID | Burst Time | Base Priority | Completion Time | Turnaround Time | Waiting Time
- Display the Gantt Chart using recorded process IDs and time points.
- Stop the program.

Code

```
#include <stdio.h>
#define MAX_PROCESSES 10
typedef struct{
    int pid;
    int burst_time;
    int remaining;
    int base_priority;
    int dyn_priority;
    int cpu_usage;
    int completed;
    int completion_time;
    int waiting_time;
    int turnaround_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 last_index){
    int min_remaining=9999, index=-1;
    int start=(last_index + 1)%n;
    for(int offset=0; offset<n;offset++){
        int i=(start+offset)%n;
        if(!p[i].completed && p[i].remaining<min_remaining){
            min_remaining=p[i].remaining;
            index=i;
        }
    }
    return index;
}
int main(){
    Process p[MAX_PROCESSES];
    int n;
    int time_quantum=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 (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,last_index=-1;
    int gantt_pids[100],gantt_times[100],gcount = 0;
    printf("\n--- CPU Scheduling Simulation (RR + SJF Hybrid) ---\n");
    while(completed<n){
        update_priority(p,n);
        int i=select_process(p,n,last_index);
        if (i==-1) break;
        int exec=(p[i].remaining<=time_quantum)?p[i].remaining:time_quantum;
        printf("Time %2d - %2d: Process P%d (priority=%d)\n",time,time +
exec,p[i].pid,p[i].dyn_priority);
        if(gcount == 0||gantt_pids[gcount - 1]!=p[i].pid){
            gantt_pids[gcount]=p[i].pid;
            gantt_times[gcount]=time;
            gcount++;
        }
        p[i].remaining-=exec;
        p[i].cpu_usage+=exec;
        time+=exec;
        if(p[i].remaining==0){
            p[i].completed=1;
            p[i].completion_time=time;
            completed++;
            printf(" -> P%d finished execution.\n",p[i].pid);
        }
        last_index=i;
    }
    gantt_times[gcount]=time;
    for(int i=0;i<n;i++){
        p[i].turnaround_time=p[i].completion_time;
        p[i].waiting_time=p[i].turnaround_time-p[i].burst_time;
    }
    printf("\nProcess\tBT\tPri\tCT\tTAT\tWT\n");
    for(int i=0;i<n;i++){
        printf("P%d\t%d\t%d\t%d\t%d\t%d\n",p[i].pid,
p[i].burst_time,p[i].base_priority,p[i].completion_time,p[i].turnaround_time,p[i].waiting_time);
    }
    printf("\nGantt Chart:\n");
    for(int i=0; i<gcount;i++) printf("<--P%d-->",gantt_pids[i]);
```

```
    printf("\n");
    for(int i=0;i<=gcount;i++) printf("%3d ",gantt_times[i]);
    printf("\n");
    return 0;
}
```

Output

```
prerana@apples-iMac Downloads % gcc sched.c
prerana@apples-iMac Downloads % ./a.out
Enter number of processes: 5
Enter burst time and base priority (nice value) for P1: 3 4
Enter burst time and base priority (nice value) for P2: 5 7
Enter burst time and base priority (nice value) for P3: 4 5
Enter burst time and base priority (nice value) for P4: 1 2
Enter burst time and base priority (nice value) for P5: 6 1

--- CPU Scheduling Simulation (RR + SJF Hybrid) ---
Time  0 -  1: Process P4 (priority=2)
 -> P4 finished execution.
Time  1 -  4: Process P1 (priority=4)
 -> P1 finished execution.
Time  4 -  7: Process P3 (priority=5)
Time  7 -  8: Process P3 (priority=6)
 -> P3 finished execution.
Time  8 - 11: Process P2 (priority=7)
Time 11 - 13: Process P2 (priority=8)
 -> P2 finished execution.
Time 13 - 16: Process P5 (priority=1)
Time 16 - 19: Process P5 (priority=2)
 -> P5 finished execution.

Process BT      Pri     CT      TAT     WT
P1      3       4       4       4       1
P2      5       7       13      13      8
P3      4       5       8       8       4
P4      1       2       1       1       0
P5      6       1       19      19      13

Gantt Chart:
<--P4--><--P1--><--P3--><--P2--><--P5-->
  0   1   4   8  13  19
prerana@apples-iMac Downloads %
```

Result

The program for UNIX Dynamic Priority Preemptive Scheduling (Hybrid RR + SJF) was successfully executed.

- The scheduler dynamically adjusted the process priorities based on CPU usage, giving preference to shorter and less CPU-intensive processes.
- The execution order of processes and their corresponding completion times were displayed through a Gantt Chart.
- The Waiting Time (WT) and Turnaround Time (TAT) for each process were correctly calculated and printed in tabular form.

Hence, the UNIX Dynamic Priority Preemptive Scheduling algorithm was successfully implemented, executed, and verified.