NITTE EDUCATION TRUST | NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY

(An Autonomous Institution, Affiliated to Visvesvaraya Technological University, Belgaum
Approved by UGC, AICTE & Govt. of Karnataka)
Yelahanka, Bangalore-560064

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

LAB MANUAL

6th Semester

UNIX SYSTEM PROGRAMMING LAB

Sub Code: 21ISG64

Prepared by:

Dr. Govind Sreekar Sheoney
Associate Professor

Mrs. Kusumitha H R
Assistant Professor

Scheme: 2021

TABLE OF CONTENTS

VISION AND MISSION OF THE DEPARTMENT

Vision

To achieve excellence in information science and engineering by empowering students with state-of-the-art technology and skills, inspiring them to drive innovation and entrepreneurship in addressing global challenges.

Mission

M1: Offer state-of-the-art curriculum through comprehensive pedagogical methods.

M2: Collaborate with industries and research institutions to address real-world challenges.

M3: Provide an environment that fosters creativity, critical thinking, innovation, and entrepreneurship.

Program Educational Objectives (PEOs)

- Graduates will progress in their careers in IT industries of repute.
- Graduates will succeed in higher studies and research.
- Graduates of Information Science and engineering will demonstrate highest integrity with ethical values, good communication skills, leadership qualities and self-learning abilities.

Program Outcomes

PO-1	Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems. .
PO-2	Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences
PO-3	Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations
PO-4	Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO-5	Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO-6	The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO-7	Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO-8	Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice
PO-9	Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO-10	Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO-11	Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO-12	Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Objectives (PSOs)

- Student will be able to understand the architecture and working of computer system with relevant system software and apply appropriate system calls.
- Student will be able to apply mathematical methodologies in modelling real world problems for the development of software applications using algorithms, data structures and programming tools.

INTRODUCTION AND SCOPE OF THE COURSE

Unix system programming refers to the development of software applications and systems that interact closely with the Unix operating system. This includes writing programs that utilize Unix system calls, libraries, and utilities to perform various tasks such as file I/O, process management, memory management, networking, and interprocess communication.

The scope of Unix system programming is broad and encompasses various areas, including but not limited to:

- File I/O: Reading from and writing to files, directories, and file systems using system calls like open, read, write, close, lseek, mkdir, rmdir, etc.
- Process Management: Creating, managing, and controlling processes using system calls like fork, exec, wait, exit, kill, etc.
- Memory Management: Allocating, deallocating, and managing memory resources using system calls like malloc, free, mmap, munmap, etc.
- Interprocess Communication (IPC): Facilitating communication and synchronization between processes using mechanisms such as pipes, message queues, shared memory, semaphores, sockets, and signals.
- Networking: Developing networked applications for communication over TCP/IP or other network protocols using socket programming.
- Thread Programming: Creating and managing threads for concurrent execution within a process using threading libraries such as POSIX threads (pthreads).
- System Administration: Developing tools and utilities for system administration tasks such as user management, file system management, process monitoring, etc.
- Shell Scripting: Writing scripts to automate tasks and execute commands in the Unix shell environment, often involving file manipulation, text processing, and command-line interface (CLI) interaction.

Unix system programming is essential for developing robust, efficient, and scalable software applications on Unix-like operating systems such as Linux, macOS, and various flavors of Unix itself. It provides developers with low-level access to system resources and facilities, allowing them to create custom solutions tailored to specific requirements and performance constraints. Moreover, understanding Unix system programming concepts is valuable for system administrators, software engineers, and developers working in diverse fields such as system software, embedded systems, server-side development, and high-performance computing.

Semester: VI **Year: 2023-24**

UNIX SYSTEM PROGRAMMING LAB				
Course Code	21ISG64		**Credits**	04
Hours/Week (L-T-P-S)	0-0-2-0		**CIE Marks**	50
Total Teaching Hours	13 hours		**SEE Marks**	-
Exam Hours	-		**Course Type**	Theory/ Integrated
Course Component	Engineering			

Course Outcomes:

Students will be able to:

Cos	Course Outcome Description	Blooms Level
1	Describe need for Standardizing the UNIX Environment.	L3
2	Apply appropriate UNIX File APIs to solve the given problem.	L3
3	Apply appropriate Unix APIs for process and job control.	L3
4	Apply signal related APIs to solve the given problem.	L3
5	Demonstrate inter-process communication using different IPC structures.	L3

Teaching Methodology:

i. Black Board Teaching
ii. Power Point Presentation
iii. Laboratory experiments

Assessment Methods:

1. Rubrics for continuous evaluation of laboratory experiments for 30 marks.
2. Two Continuous Internal Evaluations (CIEs) for 20 Marks each will be conducted and average will be considered.

Course Outcome to Program Outcome Mapping:

Cos	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
CO1	3	1												
CO2	3	2	3						1	1		1	3	
CO3	3	2	3						1	1		1	3	
CO4	3	1	3						1	1		1	3	
CO5	3	1	3											
21ISG64	3	2	3						1	1		1	3	

COURSE CONTENT

Program No.	Program Title	CO Mapping
Note: **1. The following programs can be executed on C/C++/any equivalent tool/language.**		
1	Write a C/C++ POSIX compliant program to check the following limits: (i) No. of clock ticks (ii) Max. no. of child processes (iii) Max. path length (iv) Max. no. of characters in a file name (v) Max. no. of open files/ process	1
2	Write a C program to check and display the _POSIX_VERSION constant of the system on which it is run, and also print the POSIX defined configuration options supported on any given system	1
3	Write C program to emulate the following Unix Commands? a. mv b. grep	2
4	Write a program to create hard and symbolic links for a file and display information about these links.	1
5	Write a program to demonstrate file locking and unlocking using UNIX file APIs.	2
6	Write a program to demonstrate inter-process communication using POSIX message queues	4
7	Demonstrate the following programs regarding Interprocess communication (IPC) in C: a. Write a program that illustrates how to execute two commands concurrently with a command pipe.? b. Write a program that illustrate communication between two unrelated processes using named pipe.?	4
8	Develop a C program that demonstrates various process scenarios. • Create a Zombie process. • Prevent zombie processes by implementing a double fork technique. • Illustrate the creation of an orphan process. • Implement a scenario where a parent process creates a child process and displays 'parent', while the child process displays 'child' on the screen.	3
9	Write a C/C++ program that demonstrates the use of the sigaction function for signal handling, and additionally checks whether the SIGINT signal is present in a process's signal mask and adds it if it's not already there	4
10	Write a C program that demonstrates the use of semaphores to control access to a shared resource among multiple threads.	5
	OR / Write client and server programs (using c) for interaction between server and client processes using Unix Domain sockets	

Lab evaluation rubrics matrix

Performance Indicator	Excellent	Good	Fair	Poor
Fundamental Knowledge (3M)	Student is able to demonstrate thorough knowledge about the fundamental concepts of unix	Student is ableto demonstrate partial knowledge about the fundamental concepts of unix	Student is able to demonstrate fair knowledge about the fundamental concepts of unix	Student is not able to demonstrate knowledge aboutthe fundamental concepts of unix
Understanding of problem definition (2M)	Student is ableto completely understand the problem definition	Student is ableto partially understand the problem definition	Student is able to fairly understand the definition of theproblem	Student is not able to understand the problem definition
Design of experiment (5M)	Student exhibits excellent abilityto design and analyze the given problem	Students exhibits good ability to designand analyze thegiven problem	Students exhibits fair ability to design and analyze the given problem	Students exhibitsno ability to design and analyze the given problem
Execution (5M)	All conditions handled and extensive testingof the code.	Partial conditions handled and extensive testing of the code.	Not all conditions handled and extensive testingof the code.	No Execution
Documentation (15 M)	Student has followed all the guidelines givenby the tutor in writing the lab record and observations	Student has partially followed the guidelines given by the tutor in writingthe lab record and observations	The lab record and observationsare written in a fair manner without incorporating changes specified by the tutor	Student has not followed the guidelines givenby the tutor in writing the lab record and observations

1.Write a C/C++ POSIX compliant program to check the following limits:

(i) No. of clock ticks
(ii) Max. no. of child processes
(iii) Max. path length
(iv) Max. no. of characters in a file name
(v) Max. no. of open files/ process

Additionally, include code to fetch the system's page size, hostname, and current working directory.

Solution:

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

int main() {
   // (i) No. of clock ticks
   printf("Maximum number of clock ticks per second: %ld\n", sysconf(_SC_CLK_TCK));

   // (ii) Max. no. of child processes
   printf("Maximum number of child processes: %ld\n", sysconf(_SC_CHILD_MAX));

   // (iii) Max. path length
   printf("Maximum path length: %ld\n", pathconf("/", _PC_PATH_MAX));

   // (iv) Max. no. of characters in a file name
   printf("Maximum number of characters in a file name: %ld\n", pathconf("/", _PC_NAME_MAX));

   // (v) Max. no. of open files per process
   printf("Maximum number of open files per process: %ld\n", sysconf(_SC_OPEN_MAX));

   // Retrieve system's page size
   printf("System's page size: %ld bytes\n", sysconf(_SC_PAGESIZE));

   // Retrieve system's hostname
   char hostname[HOST_NAME_MAX];
   if (gethostname(hostname, HOST_NAME_MAX) == 0) {
      printf("System hostname: %s\n", hostname);

   } else {
      perror("gethostname");
   }

   // Retrieve current working directory
```

```
    char cwd[PATH_MAX];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        printf("Current working directory: %s\n", cwd);
    } else {
        perror("getcwd");
    }

    return 0;
}
```

Output:

```
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ mkdir nethra
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ cd nethra
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~/nethra$ gedit p1.c
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~/nethra$ gcc p1.c
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~/nethra$ ./a.out
Maximum number of clock ticks per second: 100
Maximum number of child processes: 30187
Maximum path length: 4096
Maximum number of characters in a file name: 255
Maximum number of open files per process: 1024
System's page size: 4096 bytes
System hostname: admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC
Current working directory: /home/admin1/nethra
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~/nethra$
```

2. Write a program to check and display the _POSIX_VERSION constant of the system on which it is run, and also print the POSIX defined configuration options supported on any given system.

Solution:

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

int main() {
    // Check and display _POSIX_VERSION constant
    printf("_POSIX_VERSION constant: %ld\n", _POSIX_VERSION);

    // Print POSIX defined configuration options supported on the system
#ifdef _POSIX_ASYNCHRONOUS_IO
    printf("POSIX Asynchronous I/O supported\n");
#endif

#ifdef _POSIX_BARRIERS
    printf("POSIX Barriers supported\n");
#endif

#ifdef _POSIX_CHOWN_RESTRICTED
    printf("POSIX Chown Restricted supported\n");
#endif

#ifdef _POSIX_CLOCK_SELECTION
    printf("POSIX Clock Selection supported\n");
#endif

#ifdef _POSIX_CPUTIME
    printf("POSIX CPU Time supported\n");
#endif

#ifdef _POSIX_FSYNC
    printf("POSIX Fsync supported\n");
#endif

#ifdef _POSIX_IPV6
    printf("POSIX IPv6 supported\n");
#endif

#ifdef _POSIX_JOB_CONTROL
    printf("POSIX Job Control supported\n");
#endif
```

```
#ifdef _POSIX_MAPPED_FILES
   printf("POSIX Mapped Files supported\n");
#endif

#ifdef _POSIX_MEMLOCK
   printf("POSIX Memlock supported\n");
#endif

#ifdef _POSIX_MEMLOCK_RANGE
   printf("POSIX Memlock Range supported\n");
#endif

#ifdef _POSIX_MEMORY_PROTECTION
   printf("POSIX Memory Protection supported\n");
#endif

#ifdef _POSIX_MESSAGE_PASSING
   printf("POSIX Message Passing supported\n");
#endif

#ifdef _POSIX_MONOTONIC_CLOCK
   printf("POSIX Monotonic Clock supported\n");
#endif

#ifdef _POSIX_NO_TRUNC
   printf("POSIX No Truncation supported\n");
#endif

#ifdef _POSIX_PRIORITIZED_IO
   printf("POSIX Prioritized I/O supported\n");
#endif



#ifdef _POSIX_PRIORITY_SCHEDULING
   printf("POSIX Priority Scheduling supported\n");
#endif

#ifdef _POSIX_RAW_SOCKETS
   printf("POSIX Raw Sockets supported\n");
#endif

#ifdef _POSIX_REALTIME_SIGNALS
   printf("POSIX Real-time Signals supported\n");
#endif

#ifdef _POSIX_SAVED_IDS
```

```
   printf("POSIX Saved IDs supported\n");
#endif

#ifdef _POSIX_SEMAPHORES
   printf("POSIX Semaphores supported\n");
#endif

#ifdef _POSIX_SHARED_MEMORY_OBJECTS
   printf("POSIX Shared Memory Objects supported\n");
#endif

#ifdef _POSIX_SHELL
   printf("POSIX Shell supported\n");
#endif

#ifdef _POSIX_SPAWN
   printf("POSIX Spawn supported\n");
#endif

#ifdef _POSIX_SPIN_LOCKS
   printf("POSIX Spin Locks supported\n");
#endif

#ifdef _POSIX_SPORADIC_SERVER
   printf("POSIX Sporadic Server supported\n");
#endif

#ifdef _POSIX_SYNCHRONIZED_IO
   printf("POSIX Synchronized I/O supported\n");
#endif

#ifdef _POSIX_THREAD_ATTR_STACKADDR
   printf("POSIX Thread Attribute Stack Address supported\n");
#endif

#ifdef _POSIX_THREAD_ATTR_STACKSIZE
   printf("POSIX Thread Attribute Stack Size supported\n");
#endif

#ifdef _POSIX_THREAD_CPUTIME
   printf("POSIX Thread CPU Time supported\n");
#endif

#ifdef _POSIX_THREAD_PRIO_INHERIT
   printf("POSIX Thread Priority Inheritance supported\n");
#endif
```

```
#ifdef _POSIX_THREAD_PRIO_PROTECT
   printf("POSIX Thread Priority Protect supported\n");
#endif

#ifdef _POSIX_THREAD_PRIORITY_SCHEDULING
   printf("POSIX Thread Priority Scheduling supported\n");
#endif

#ifdef _POSIX_THREAD_PROCESS_SHARED
   printf("POSIX Thread Process Shared supported\n");
#endif

#ifdef _POSIX_THREAD_ROBUST_PRIO_INHERIT
   printf("POSIX Thread Robust Priority Inheritance supported\n");
#endif

#ifdef _POSIX_THREAD_ROBUST_PRIO_PROTECT
   printf("POSIX Thread Robust Priority Protect supported\n");
#endif

#ifdef _POSIX_THREAD_SAFE_FUNCTIONS
   printf("POSIX Thread Safe Functions supported\n");
#endif

#ifdef _POSIX_THREAD_SPORADIC_SERVER
   printf("POSIX Thread Sporadic Server supported\n");
#endif

#ifdef _POSIX_THREADS
   printf("POSIX Threads supported\n");
#endif

#ifdef _POSIX_TIMEOUTS
   printf("POSIX Timeouts supported\n");
#endif

#ifdef _POSIX_TIMERS
   printf("POSIX Timers supported\n");
#endif

#ifdef _POSIX_TRACE
   printf("POSIX Trace supported\n");
#endif

#ifdef _POSIX_TYPED_MEMORY_OBJECTS
   printf("POSIX Typed Memory Objects supported\n");
#endif
```

```
#ifdef _POSIX_VDISABLE
   printf("POSIX VDISABLE supported\n");
#endif

#ifdef _POSIX_V6_ILP32_OFF32
   printf("POSIX V6 ILP32 OFF32 supported\n");
#endif

#ifdef _POSIX_V6_ILP32_OFFBIG
   printf("POSIX V6 ILP32 OFFBIG supported\n");
#endif

#ifdef _POSIX_V6_LP64_OFF64
   printf("POSIX V6 LP64 OFF64 supported\n");
#endif

#ifdef _POSIX_V6_LPBIG_OFFBIG
   printf("POSIX V6 LPBIG OFFBIG supported\n");
#endif

#ifdef _POSIX_V7_ILP32_OFF32
   printf("POSIX V7 ILP32 OFF32 supported\n");
#endif

#ifdef _POSIX_V7_ILP32_OFFBIG
   printf("POSIX V7 ILP32 OFFBIG supported\n");
#endif

#ifdef _POSIX_V7_LP64_OFF64
   printf("POSIX V7 LP64 OFF64 supported\n");
#endif

#ifdef _POSIX_V7_LPBIG_OFFBIG
   printf("POSIX V7 LPBIG OFFBIG supported\n");
#endif

   return 0;
}
```

Output:

```
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~/nethra$ gcc p2.c
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~/nethra$ ./a.out
_POSIX_VERSION constant: 200809
POSIX Asynchronous I/O supported
POSIX Barriers supported
POSIX Chown Restricted supported
POSIX Clock Selection supported
POSIX CPU Time supported
POSIX Fsync supported
POSIX IPv6 supported
POSIX Job Control supported
POSIX Mapped Files supported
POSIX Memlock supported
POSIX Memlock Range supported
POSIX Memory Protection supported
POSIX Message Passing supported
POSIX Monotonic Clock supported
POSIX No Truncation supported
POSIX Prioritized I/O supported
POSIX Priority Scheduling supported
POSIX Raw Sockets supported
POSIX Real-time Signals supported
POSIX Saved IDs supported
POSIX Semaphores supported
POSIX Shared Memory Objects supported
POSIX Shell supported
POSIX Spawn supported
POSIX Spin Locks supported
POSIX Sporadic Server supported
POSIX Synchronized I/O supported
POSIX Thread Attribute Stack Address supported
POSIX Thread Attribute Stack Size supported
POSIX Thread CPU Time supported
POSIX Thread Priority Inheritance supported
POSIX Thread Priority Protect supported
POSIX Thread Priority Scheduling supported
POSIX Thread Process Shared supported
POSIX Thread Robust Priority Inheritance supported
POSIX Thread Robust Priority Protect supported
POSIX Thread Safe Functions supported
POSIX Thread Sporadic Server supported
POSIX Threads supported
POSIX Timeouts supported
POSIX Timers supported
POSIX Trace supported
POSIX Typed Memory Objects supported
POSIX VDISABLE supported
POSIX V6 LP64 OFF64 supported
POSIX V6 LPBIG OFFBIG supported
POSIX V7 LP64 OFF64 supported
POSIX V7 LPBIG OFFBIG supported
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~/nethra$
```

3. Write C program to emulate the following Unix Commands?

a. mv

b. grep

a. Emulate mv Command:

Solution:

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

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source> <destination>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (rename(argv[1], argv[2]) == -1) {
        perror("mv");
        exit(EXIT_FAILURE);
    }

    printf("Moved '%s' to '%s'\n", argv[1], argv[2]);
    return 0;
}
```

Output:

```
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~/nethra$ gcc p3.c
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~/nethra$ ./a.out m1.txt k1.txt
Moved 'm1.txt' to 'k1.txt'
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~/nethra$
```

b. Emulate grep Command:

Solution:

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

#define MAX_LINE_LENGTH 1024

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <pattern>\n", argv[0]);
        return 1;
    }

    char pattern[MAX_LINE_LENGTH];
    strcpy(pattern, argv[1]);

    char line[MAX_LINE_LENGTH];
    while (fgets(line, sizeof(line), stdin) != NULL) {
        if (strstr(line, pattern) != NULL) {
            printf("%s", line);
        }
    }

    return 0;
}
```

Output:

```
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ gcc p3b.c
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ ./a.out NETHRAVATHI < t1.txt
NETHRAVATHI
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$
```

4. Write a program to create hard and symbolic links for a file and display information about these links.

Solution:

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

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    char *filename = argv[1];
    char hard_linkname[256], sym_linkname[256];
    struct stat file_stat;

    // Create hard link
    if (link(filename, "hard_link") == -1) {
        perror("Error creating hard link");
        exit(EXIT_FAILURE);
    }

    // Create symbolic link
    if (symlink(filename, "sym_link") == -1) {
        perror("Error creating symbolic link");
        exit(EXIT_FAILURE);
    }

    // Get file information
    if (stat(filename, &file_stat) == -1) {
        perror("Error getting file information");
        exit(EXIT_FAILURE);
    }

    // Display information about hard link
    printf("Information about hard link:\n");
    printf("Number of links: %ld\n", file_stat.st_nlink);

    // Display information about symbolic link
    if (lstat("sym_link", &file_stat) == -1) {
        perror("Error getting symbolic link information");
        exit(EXIT_FAILURE);
```

```
    }

    printf("Information about symbolic link:\n");
    printf("Number of links: %ld\n", file_stat.st_nlink);

    return 0;
}
```

Output:

```
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ gcc p4.c
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ ./a.out h1.txt d1.txt
Information about hard link:
Number of links: 2
Information about symbolic link:
Number of links: 1
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$
```

5. Write a program to demonstrate file locking and unlocking using UNIX file APIs.

Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

void file_lock(int fd) {
    struct flock fl;

    fl.l_type = F_WRLCK;  // Write lock
    fl.l_whence = SEEK_SET;
    fl.l_start = 0;
    fl.l_len = 0;  // Lock entire file

    // Attempt to acquire the lock
    if (fcntl(fd, F_SETLKW, &fl) == -1) {
        perror("fcntl");
        exit(EXIT_FAILURE);
    }

    printf("File locked successfully!\n");
}

void file_unlock(int fd) {
    struct flock fl;

    fl.l_type = F_UNLCK;  // Unlock
    fl.l_whence = SEEK_SET;
    fl.l_start = 0;
    fl.l_len = 0;  // Unlock entire file

    // Attempt to release the lock
    if (fcntl(fd, F_SETLK, &fl) == -1) {
        perror("fcntl");
        exit(EXIT_FAILURE);
    }

    printf("File unlocked successfully!\n");
}

int main() {
    int fd;
```

```
    char *filename = "testfile.txt";

    // Open the file
    if ((fd = open(filename, O_RDWR | O_CREAT, 0666)) == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    // Write some content to the file
    if (write(fd, "Hello, world!\n", 14) != 14) {
        perror("write");
        exit(EXIT_FAILURE);
    }

    // Lock the file
    file_lock(fd);

    // Try to write to the locked file
    if (write(fd, "Locked file content\n", 20) == -1) {
        perror("write");
        printf("Error: Cannot write to the locked file.\n");
    }

    // Unlock the file
    file_unlock(fd);

    // Write to the unlocked file
    if (write(fd, "Unlocked file content\n", 22) != 22) {
        perror("write");
        exit(EXIT_FAILURE);
    }

    // Close the file
    close(fd);

    return 0;
}
```

Output:

```
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ gcc p5.c
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ ./a.out
File locked successfully!
File unlocked successfully!
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$
```

6. Write a program to demonstrate inter-process communication using POSIX message queues

Solution:

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

#define QUEUE_NAME "/my_queue"
#define MAX_MSG_SIZE 256

void sender_process() {
    mqd_t mqd;
    char msg[MAX_MSG_SIZE];

    // Open the message queue for writing
    mqd = mq_open(QUEUE_NAME, O_WRONLY);
    if (mqd == (mqd_t)-1) {
        perror("mq_open");
        exit(EXIT_FAILURE);
    }

    // Sender process
    printf("Sender process: Enter a message to send to receiver (type 'exit' to quit):\n");
    while (1) {
        fgets(msg, MAX_MSG_SIZE, stdin);
        // Send the message to the message queue
        if (mq_send(mqd, msg, strlen(msg), 0) == -1) {
            perror("mq_send");
            exit(EXIT_FAILURE);
        }
        if (strncmp(msg, "exit", 4) == 0) {
            break;
        }
    }

    // Close the message queue
    mq_close(mqd);
}

void receiver_process() {
```

```
    mqd_t mqd;
    char msg[MAX_MSG_SIZE];
    ssize_t bytes_read;

    // Open the message queue for reading
    mqd = mq_open(QUEUE_NAME, O_RDONLY);
    if (mqd == (mqd_t)-1) {
        perror("mq_open");
        exit(EXIT_FAILURE);
    }

    // Receiver process
    printf("Receiver process: Waiting for messages...\n");
    while (1) {
        // Receive a message from the message queue
        bytes_read = mq_receive(mqd, msg, MAX_MSG_SIZE, NULL);
        if (bytes_read == -1) {
            perror("mq_receive");
            exit(EXIT_FAILURE);
        }
        // Null-terminate the received message
        msg[bytes_read] = '\0';

        // Print the received message
        printf("Received message: %s", msg);

        // Check if received message is "exit" to quit the receiver process
        if (strncmp(msg, "exit", 4) == 0) {
            break;
        }
    }

    // Close the message queue
    mq_close(mqd);
}

int main() {
    // Create message queue attributes
    struct mq_attr attr;
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;       // Maximum number of messages
    attr.mq_msgsize = MAX_MSG_SIZE; // Maximum message size
    attr.mq_curmsgs = 0;       // Number of messages currently in queue

    // Create the message queue
    mqd_t mqd = mq_open(QUEUE_NAME, O_CREAT | O_RDWR, 0666, &attr);
    if (mqd == (mqd_t)-1) {
```

```
        perror("mq_open");
        exit(EXIT_FAILURE);
    }

    // Fork a child process
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {  // Child process (receiver)
        receiver_process();
    } else { // Parent process (sender)
        sender_process();
    }

    // Unlink the message queue
    mq_unlink(QUEUE_NAME);

    return 0;
}
```

Output:

```
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ gcc p6.c
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ ./a.out
Sender process: Enter a message to send to receiver (type 'exit' to quit):
Receiver process: Waiting for messages...
hello
Received message: hello
exit
Received message: exit
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$
```

7. Demonstrate the following programs regarding Interprocess communication (IPC) in C:

a. Write a program that illustrates how to execute two commands concurrently with a command pipe.?(co4)

b. Write a program that illustrate communication between two unrelated processes using named pipe.? (co4)

Solution: a.

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

int main() {
    int pipefd[2];

    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {  // Child process
        close(pipefd[0]);  // Close reading end of the pipe in the child process
        dup2(pipefd[1], STDOUT_FILENO);  // Redirect stdout to the writing end of the pipe
        close(pipefd[1]);  // Close the original writing end of the pipe
        execlp("ls", "ls", "-l", NULL);  // Execute the first command
        perror("execlp ls");
        exit(EXIT_FAILURE);
    } else {  // Parent process
        close(pipefd[1]);  // Close writing end of the pipe in the parent process
        dup2(pipefd[0], STDIN_FILENO);  // Redirect stdin to the reading end of the pipe
        close(pipefd[0]);  // Close the original reading end of the pipe
        execlp("wc", "wc", "-l", NULL);  // Execute the second command
        perror("execlp wc");
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

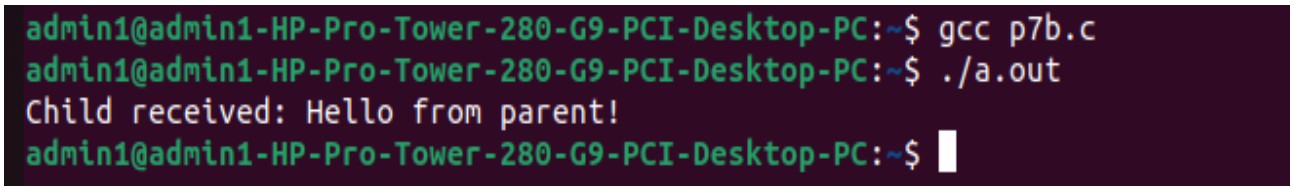
Output:

```
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ gcc p7a.c
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ ./a.out
33
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$
```

Solution: b.

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

#define FIFO_FILE "/tmp/myfifo"

int main() {
    char buf[256];
    int fd;

    // Create the FIFO (named pipe)
    mkfifo(FIFO_FILE, 0666);

    // Parent process writes to the FIFO
    if (fork() != 0) {
        fd = open(FIFO_FILE, O_WRONLY);
        write(fd, "Hello from parent!", sizeof("Hello from parent!"));
        close(fd);
    }
    // Child process reads from the FIFO
    else {
        fd = open(FIFO_FILE, O_RDONLY);
        read(fd, buf, sizeof(buf));
        printf("Child received: %s\n", buf);
        close(fd);
    }

    return 0;
}
```

Output:



8. Develop a C program that demonstrates various process scenarios.

- **Create a Zombie process.**
- **Prevent zombie processes by implementing a double fork technique.**
- **Illustrate the creation of an orphan process.**
- **Implement a scenario where a parent process creates a child process and displays 'parent', while the child process displays 'child' on the screen.**

Solution:

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

void create_zombie_process() {
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid > 0) {
        // Parent process
        printf("Parent process created a zombie process (PID: %d)\n", pid);
    } else {
        // Child process
        printf("Zombie process created (PID: %d)\n", getpid());
        exit(EXIT_SUCCESS);
    }
}

void prevent_zombie_processes() {
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid > 0) {
        // Parent process
        printf("Parent process created a child process (PID: %d)\n", pid);
        sleep(2);  // Wait for child to exit
    } else {
        // First child process
        pid = fork();

        if (pid == -1) {
            perror("fork");
```

```
            exit(EXIT_FAILURE);
        } else if (pid > 0) {
            // First child process exits immediately
            printf("First child process (PID: %d) exited\n", getpid());
            exit(EXIT_SUCCESS);
        } else {
            // Grandchild process
            printf("Grandchild process (PID: %d)\n", getpid());
            sleep(5);  // Simulate some work
            printf("Grandchild process completed\n");
            exit(EXIT_SUCCESS);
        }
    }
}

void create_orphan_process() {
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid > 0) {
        // Parent process
        printf("Parent process (PID: %d)\n", getpid());
        sleep(2);  // Let parent exit first
    } else {
        // Child process becomes orphan
        printf("Orphan process created (PID: %d)\n", getpid());
        sleep(5);  // Simulate some work
        printf("Orphan process completed\n");
        exit(EXIT_SUCCESS);
    }
}

void parent_child_scenario() {
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid > 0) {
        // Parent process
        printf("Parent process (PID: %d)\n", getpid());
        wait(NULL); // Wait for child to complete
    } else {
        // Child process
        printf("Child process (PID: %d)\n", getpid());
```

```
        exit(EXIT_SUCCESS);
    }
}

int main() {
    printf("Demonstrating Zombie process:\n");
    create_zombie_process();
    sleep(5);  // Wait for the zombie process to become defunct

    printf("\nPreventing Zombie processes:\n");
    prevent_zombie_processes();
    sleep(7);  // Wait for grandchild process to complete

    printf("\nCreating an Orphan process:\n");
    create_orphan_process();

    printf("\nParent-Child scenario:\n");
    parent_child_scenario();

    return 0;
}
```

Output:

```
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ gcc p8.c
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ ./a.out
Demonstrating Zombie process:
Parent process created a zombie process (PID: 5815)
Zombie process created (PID: 5815)

Preventing Zombie processes:
Parent process created a child process (PID: 5816)
First child process (PID: 5816) exited
Grandchild process (PID: 5817)
Grandchild process completed

Creating an Orphan process:
Parent process (PID: 5814)
Orphan process created (PID: 5819)

Parent-Child scenario:
Parent process (PID: 5814)
Child process (PID: 5820)
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ Orphan process completed
```

9. Write a C/C++ program that demonstrates the use of the sigaction function for signal handling, and additionally checks whether the SIGINT signal is present in a process's signal mask and adds it if it's not already there
Or "Write a C/C++ program that demonstrates the following:

- **The use of the sigaction function for signal handling.**
- **Checking whether the SIGINT signal is present in a process's signal mask and adding it if it's not already there."**

Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

// Signal handler function
void sig_handler(int signum) {
    printf("Received signal: %d\n", signum);
}

int main() {
    struct sigaction sa;
    sigset_t old_mask, new_mask;

    // Set up the signal handler
    sa.sa_handler = sig_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;

    // Register signal handler for SIGINT
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    printf("Signal handler registered for SIGINT (Ctrl+C)\n");

    // Initialize new signal mask
    sigemptyset(&new_mask);
    sigaddset(&new_mask, SIGINT);

    // Get current signal mask
    if (sigprocmask(SIG_BLOCK, NULL, &old_mask) == -1) {
        perror("sigprocmask");
        exit(EXIT_FAILURE);
    }
```

```
    // Check if SIGINT is already in the signal mask
    if (sigismember(&old_mask, SIGINT)) {
        printf("SIGINT is already present in the signal mask.\n");
    } else {
        // Add SIGINT to the signal mask
        if (sigprocmask(SIG_BLOCK, &new_mask, NULL) == -1) {
            perror("sigprocmask");
            exit(EXIT_FAILURE);
        }
        printf("Added SIGINT to the signal mask.\n");
    }

    // Wait for signals indefinitely
    while (1) {
        sleep(1);
    }

    return 0;
}
```

Output:

```
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ gcc p9.c
admin1@admin1-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ ./a.out
Signal handler registered for SIGINT (Ctrl+C)
Added SIGINT to the signal mask.
```

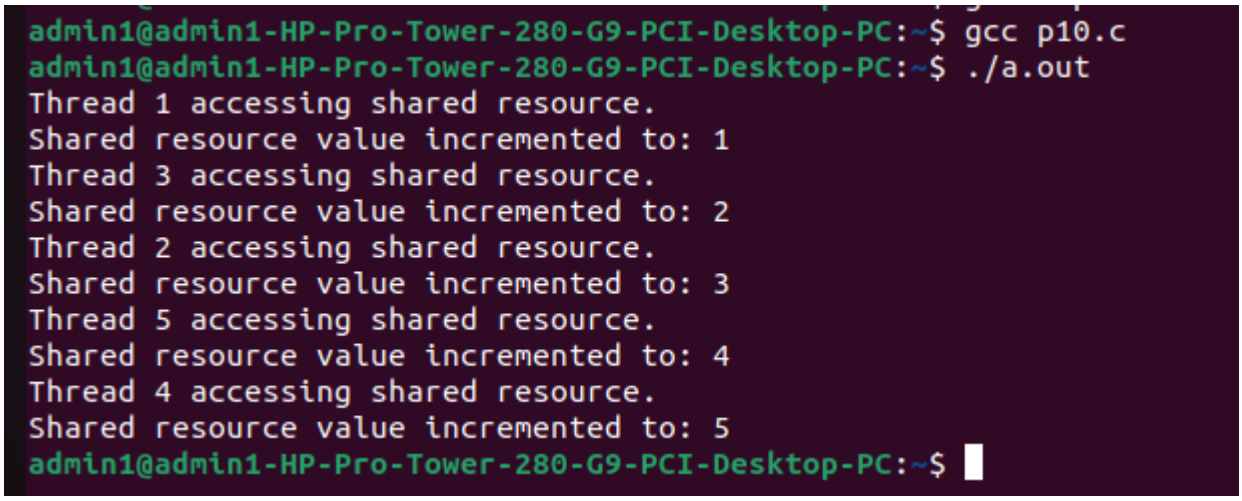
10. Write a C program that demonstrates the use of semaphores to control access to a shared resource among multiple threads.

Solution:

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

#define NUM_THREADS 5

// Shared resource
int shared_resource = 0;

// Semaphore declaration
sem_t semaphore;

// Thread function
void *thread_function(void *arg) {
    int thread_id = *((int *)arg);

    // Wait for semaphore
    sem_wait(&semaphore);

    // Critical section
    printf("Thread %d accessing shared resource.\n", thread_id);
    shared_resource++;
    printf("Shared resource value incremented to: %d\n", shared_resource);

    // Release semaphore
    sem_post(&semaphore);

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    // Initialize semaphore
    sem_init(&semaphore, 0, 1); // Initial value is 1, indicating semaphore is available

    // Create threads
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i + 1;
        if (pthread_create(&threads[i], NULL, thread_function, (void *)&thread_ids[i]) != 0) {
```

```
            fprintf(stderr, "Error creating thread %d\n", i);
            exit(EXIT_FAILURE);
        }
    }

    // Join threads
    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_join(threads[i], NULL) != 0) {
            fprintf(stderr, "Error joining thread %d\n", i);
            exit(EXIT_FAILURE);
        }
    }

    // Destroy semaphore
    sem_destroy(&semaphore);

    return 0;
}
```

Output:



[References]

[1] UNIX System Programming Using C++, Terrence Chan, Prentice Hall India, 1999

[2] Advanced Programming in the UNIX Environment, W. Richard Stevens, Addison–Wesley, 2nd Edition, 1992