Experiment 1

1

Ls -l

2.

mkdir lab_files
cd lab_files
touch example.txt

3.

cp example.txt example_backup.txt

4.

cd lab_files
rm example.txt

5.

cd lab_files
mkdir folder1 folder2 folder3

6.

The /etc directory in the Linux file system serves a crucial role as it stores configuration files and system-wide settings for the operating system and installed applications

7

The /usr directory in Linux typically contains user-related programs, libraries, documentation, and other files not required for system booting or repair. Here's a list of common directories and files found in the /usr directory:

bin: Contains user executable binaries (programs).
lib: Libraries required for binaries in /usr/bin and /usr/sbin.
local: Typically contains locally installed software and libraries managed by the system administrator.
sbin: Contains system administrator binaries (programs).
share: Architecture-independent data files shared among different architectures.
include: Header files required for compiling applications.
src: Source code of various software packages.

8.

The /bin and /sbin directories contain essential executable binaries (programs) in the Linux file system. Here's an explanation of their significance:

/bin directory:

This directory contains essential user-level binaries (programs) that are required for normal system operation and are accessible to all users.

These binaries are fundamental to the system's functioning and are often used during the boot process or by regular users for everyday tasks.

Common binaries found in /bin include basic system utilities like ls (list directory contents), cp (copy files), mv (move files), rm (remove files), mkdir (make directories), cat (concatenate and display files), and many others.

Since /bin binaries are essential for the system's operation, they are typically located in a directory that is included in the system's default executable search path, allowing users to execute them without specifying their full path.

/sbin directory:

This directory contains system administration binaries (programs) that are essential for system administration tasks and are typically restricted to use by the root user or other privileged users.

The binaries in /sbin are used for system maintenance, configuration, and troubleshooting purposes.

Common binaries found in /sbin include administrative utilities like mount (mount filesystems), umount (unmount filesystems), ifconfig (configure network interfaces), fdisk (disk partitioning), shutdown (shutdown or reboot the system), and others.

By placing these binaries in /sbin, Linux ensures that only users with administrative privileges can access and execute them, helping to prevent accidental misuse or unauthorized changes to critical system settings.

Experiment 2

1.

```
#!/bin/bash

# Prompt the user to enter a value
echo "Enter a number:"
read num

# Initialize sum
sum=0

# Loop from 1 to the user-provided value and calculate the sum
for (( i=1; i<=$num; i++ ))
do
  sum=$((sum + i))
done

# Print the sum
echo "The sum of numbers from 1 to $num is: $sum"
```

2.

```
#!/bin/bash

# Prompt the user to enter the number of terms
echo "Enter the number of Fibonacci terms:"
read num_terms

# Initialize the first two terms of the Fibonacci series
a=0
b=1

# Print the first two terms
echo "Fibonacci series up to $num_terms terms:"
echo -n "$a $b "

# Loop to generate the remaining terms
for (( i=3; i<=num_terms; i++ ))
do
  # Calculate the next term
```

```
    next=$((a + b))

    # Print the next term
    echo -n "$next "

    # Update variables for the next iteration
    a=$b
    b=$next
done

echo "" # Print a newline at the end
```

3.

```
#!/bin/bash

# Function to check if a number is prime
is_prime() {
    n=$1
    if [ $n -le 1 ]; then
        echo "$n is not prime."
        exit 1
    fi

    # Loop to check divisibility
    for (( i=2; i*i<=n; i++ ))
    do
        if [ $((n % i)) -eq 0 ]; then
            echo "$n is not prime."
            exit 0
        fi
    done

    echo "$n is prime."
}

# Prompt the user to enter a number
echo "Enter a number:"
read num

# Call the is_prime function with the user-input number
is_prime $num
```

4.

```
#!/bin/bash

# Function to reverse a number
reverse_number() {
    num=$1
    reversed=0

    while [ $num -gt 0 ]
```

```
    do
        # Extract the last digit of the number
        digit=$((num % 10))

        # Append the digit to the reversed number
        reversed=$((reversed * 10 + digit))

        # Remove the last digit from the number
        num=$((num / 10))
    done

    echo "Reversed number: $reversed"
}

# Prompt the user to enter a number
echo "Enter a number:"
read input_number

# Call the reverse_number function with the user-input number
reverse_number $input_number
```

5.

```
#!/bin/bash

# Prompt the user to enter the directory path
echo "Enter the directory path:"
read directory

# Check if the directory exists
if [ ! -d "$directory" ]; then
    echo "Error: Directory not found."
    exit 1
fi

# Change to the specified directory
cd "$directory"

# Display all files in the directory using a for loop
echo "Files in the directory:"
for file in *
do
    # Check if the item is a file
    if [ -f "$file" ]; then
        echo "$file"
    fi
done
```

6.

```
#!/bin/bash

# Define correct username and password
```

```
correct_username="user"
correct_password="password"

# Prompt the user to enter a username
echo "Enter username:"
read username

# Prompt the user to enter a password
echo "Enter password:"
read -s password # '-s' option hides the password as it's typed

# Check if both username and password are correct
if [ "$username" = "$correct_username" ] && [ "$password" = "$correct_password" ]; then
   echo "Access granted."
else
   echo "Access denied."
fi
7
#!/bin/bash

# Function to perform addition
addition() {
   result=$(echo "$1 + $2" | bc)
   echo "Result: $result"
}

# Function to perform subtraction
subtraction() {
   result=$(echo "$1 - $2" | bc)
   echo "Result: $result"
}

# Function to perform multiplication
multiplication() {
   result=$(echo "$1 * $2" | bc)
   echo "Result: $result"
}

# Function to perform division
division() {
   if [ $2 -eq 0 ]; then
      echo "Error: Division by zero is not allowed."
   else
      result=$(echo "scale=2; $1 / $2" | bc)
      echo "Result: $result"
   fi
}

# Menu
```

```
echo "Menu:"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"
echo "5. Exit"

# Prompt user for choice
read -p "Enter your choice (1-5): " choice

# Perform action based on user choice
case $choice in
  1) read -p "Enter first number: " num1
     read -p "Enter second number: " num2
     addition $num1 $num2
     ;;
  2) read -p "Enter first number: " num1
     read -p "Enter second number: " num2
     subtraction $num1 $num2
     ;;
  3) read -p "Enter first number: " num1
     read -p "Enter second number: " num2
     multiplication $num1 $num2
     ;;
  4) read -p "Enter first number: " num1
     read -p "Enter second number: " num2
     division $num1 $num2
     ;;
  5) echo "Exiting..."
     exit 0
     ;;
  *) echo "Invalid choice. Please enter a number from 1 to 5."
esac
```

Experiment 3

1.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>
#include<sys/types.h>
#include<sys/stat.h>
int main(void){
   int rfd=open("File.txt",O_RDONLY);
   int wfd=open("test.txt",O_CREAT| O_RDWR,0777);
   if(rfd==-1 || wfd==-1){
      printf("Error in opening file\n");
      exit(1);
```

```
        }
        off_t fileSize =lseek(rfd,0,SEEK_END);
        lseek(rfd,0,SEEK_SET);
        char buffer[fileSize/2];
        ssize_t bytesRead=0;
        int n;
        printf("Select the option:\n");
        printf("1. Copy the first half of data from file.\n");
        printf("2. Copy the second half of data from file.\n");
        scanf("%d",&n);
        switch(n){
           case 1:
                 lseek(wfd,0,SEEK_SET);
                 bytesRead=read(rfd,buffer,fileSize/2);
                 write(wfd,buffer,fileSize/2);
                 break;
           case 2:
                 lseek(wfd,0,SEEK_SET);
                 lseek(rfd,fileSize/2,SEEK_SET);
                 bytesRead=read(rfd,buffer,fileSize/2);
                 write(wfd,buffer,bytesRead);
                 break;
           default:
              printf("Invalid option\n");
              exit(1);
        }
        close(wfd);
        return 0;
    }
```

2.

```
#include<unistd.h>

#include<fcntl.h>

#include<sys/types.h>

#include<sys/stat.h>

int main(){

   int fd=open("input.txt",O_CREAT|O_RDWR,0777);

   off_t fileSize=lseek(fd,0,SEEK_END);

   char  buffer[1024];

   ssize_t totalbytesRead=0;

   char ch;
```

```
    while(totalbytesRead<sizeof(buffer)-1){
        ssize_t bytesRead=read(0,&ch,1);
        if(ch=='$'){
            break;
        }
        buffer[totalbytesRead++]=ch;
    }
    buffer[totalbytesRead]='\0';
    write(fd,buffer,totalbytesRead);
    close(fd);
    return 0;
}
```

3.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>
#include<sys/types.h>
#include<sys/stat.h>
void encrypt(char buffer[],int shift,int bytesRead){
    for(int i=0;i<=bytesRead;i++){
        if(buffer[i]>='a' && buffer[i]<='z'){
            buffer[i]=((buffer[i]-'a')+shift)%26+'a';
        }
        else if(buffer[i]>='A' && buffer[i]<='Z'){
            buffer[i]=((buffer[i]-'A')+shift)%26+'A';
        }
```

```
    }
}
int main(){
    int wfd=open("encrypt.txt",O_CREAT| O_RDWR,0666);
    int rfd=open("file2.txt",O_CREAT| O_RDONLY);
    if(rfd==-1 || wfd==-1){
        printf("Error while opening file\n");
        exit(1);
    }
    off_t fileSize=lseek(rfd,0,SEEK_END);
    lseek(rfd,0,SEEK_SET);
    char buffer[fileSize];
    ssize_t bytesRead=read(rfd,buffer,fileSize);
    int shift;
    printf("Enter the shift value: ");
    scanf("%d",&shift);
    encrypt(buffer,shift,fileSize);
    write(wfd,buffer,bytesRead);
    close(wfd);
    close(rfd);
    return 0;
}
```

Experiment 4

1.
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

int main() {
   char dirname[100];

   // Prompt the user to enter the directory name
   printf("Enter the directory name: ");
   scanf("%s", dirname);

   // Create the directory using mkdir system call
   if (mkdir(dirname, 0777) == -1) {
      perror("Error creating directory");
      exit(EXIT_FAILURE);
   }

   printf("Directory '%s' created successfully.\n", dirname);

   return 0;
}
```

2.
```
#include<stdio.h>
#include<unistd.h>
#include<dirent.h>
int main ()
{
printf("Enter the path");
char path[100];
scanf("%s",path);
DIR *dir;
struct dirent *de
dir=opendir(path);
if(dir)
{
printf("the contents are \n");
while(de=readdir(dir))
{
printf("%s\n",de->d_name);
}
closedir(dir);
}
return 0;
}
```

3.
```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int main()
{
   char dirName[16];
   int ret = 0;
   printf("Enter directory name: ");
   scanf("%s", dirName);

   ret = rmdir(dirName);

   if (ret == 0)
      printf("Given empty directory removed successfully\n");
   else
      printf("Unable to remove directory %s\n", dirName);

   return 0;
}
```

4.

```
#include<unistd.h>
#include<stdlib.h>
#include<stdio.h>
int main()
{
char *buf;
buf=(char *)malloc(100*sizeof(char));
//Malloc is used for dynamic memory allocation and is useful when you don't know the amount of
//memory needed during compile time
getcwd(buf,100);
printf("\n %s \n",buf);
}
```

Experiment 5

1.

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

int main()
{
        int x,i;
        printf("Enter the number of fork commands:\n");
        scanf("%d",&i);
        for(x=0;x<i;x++) // loop will run n times
        {
                if(fork() == 0)
                {
```

```
                printf("[son] pid %d from [parent] pid %d\n",getpid(),getppid());
                exit(0);
            }
        }

        wait(NULL);

}
```

2.

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

// Driver code
int main()
{
        int pid, pid1, pid2;

        // variable pid will store the
        // value returned from fork() system call
        pid = fork();

        // If fork() returns zero then it
        // means it is child process.
        if (pid == 0) {

                // First child needs to be printed
                // later hence this process is made
                // to sleep for 3 seconds.
                sleep(3);

                // This is first child process
                // getpid() gives the process
                // id and getppid() gives the
                // parent id of that process.
                printf("child[1] --> pid = %d and ppid = %d\n",
                        getpid(), getppid());
        }

        else {
                pid1 = fork();
                if (pid1 == 0) {
                        sleep(2);
                        printf("child[2] --> pid = %d and ppid = %d\n",
                                getpid(), getppid());
                }
                else {
                        pid2 = fork();
                        if (pid2 == 0) {
```

```
                                        // This is third child which is
                                        // needed to be printed first.
                                        printf("child[3] --> pid = %d and ppid = %d\n",
                                                getpid(), getppid());
                                }

                                // If value returned from fork()
                                // in not zero and >0 that means
                                // this is parent process.
                                else {
                                        // This is asked to be printed at last
                                        // hence made to sleep for 3 seconds.
                                        sleep(3);
                                        printf("parent --> pid = %d\n", getpid());
                                }
                        }
                }

                return 0;
}
```

3.

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

#define oops(m) {perror(m); exit(EXIT_FAILURE);}

int main() {
   pid_t pid1_1, pid1_2, pid1_1_1, pid1_1_2, pid1_2_1, pid1_2_2;

   pid1_1 = fork();

   if (pid1_1 < 0) {
      oops("Fork Failed!");
   }

   // child 1.1
   if (pid1_1 == 0) {
      printf("I am the child %d\n", getpid());
      if (execlp("./iam", "iam", "1.1", NULL) < 0)
      oops("Execlp Failed!");

   } else {
      // grandchild 1.1.1
      pid1_1_1 = fork();
```

```
    if (pid1_1_1 < 0) {
        oops("Fork Failed!");
    }
    if (pid1_1_1 == 0) {
        printf("I am the grandchild %d\n", getpid());
        if (execlp("./iam", "iam", "1.1.1", NULL) < 0)
        oops("Execlp Failed!");
    }
    //grandchild 1.1.2
    pid1_1_2 = fork();
    if (pid1_1_2 < 0) {
        oops("Fork Failed!");
    }
    if (pid1_1_2 == 0) {
        printf("I am the grandchild %d\n", getpid());
        if (execlp("./iam", "iam", "1.1.2", NULL) < 0)
        oops("Execlp Failed!");
    }
}

pid1_2 = fork();

if (pid1_2 < 0) {
    oops("Fork Failed!");
}
// child 1.2
if (pid1_2 == 0) {
    printf("I am the child %d\n", getpid());
    if (execlp("./iam", "iam", "1.2", NULL) < 0)
    oops("Execlp Failed!");
} else {
    // grandchild 1.2.1
    pid1_2_1 = fork();
    if (pid1_2_1 < 0) {
        oops("Fork Failed!");
    }
    if (pid1_2_1 == 0) {
        printf("I am the grandchild %d\n", getpid());
        if (execlp("./iam", "iam", "1.2.1", NULL) < 0)
        oops("Execlp Failed!");
    }
    // grandchild 1.2.2
    pid1_2_2 = fork();
    if (pid1_2_2 < 0) {
        oops("Fork Failed!");
    }
    if (pid1_2_2 == 0) {
        printf("I am the grandchild %d\n", getpid());
        if (execlp("./iam", "iam", "1.2.2", NULL) < 0)
```

```
        oops("Execlp Failed!");
      }
   }

   // pid > 0 ==> must be parent
   printf("I am the parent %d\n", getpid());
   /* parent will wait for the child to complete */
   if (waitpid(-1, NULL, 0) < 0)
      printf("-1 from wait() with errno = %d\n", errno);

   printf("Child terminated; parent exiting\n");
   exit(EXIT_SUCCESS);
}
```

Experiment 6

1.

```
#include <stdio.h>
#include<stdlib.h>
#include<string.h>
#include<pthread.h>
      //char str1[100] = "Hi ", str2[100] = "World";
      char str1[100],str2[100],str3[100];
void *concat()
{
      strcat(str1,str2);
      strcpy(str3,str1);
      pthread_exit(NULL);
}
int main()
{
pthread_t t1;
printf("Enter the string1:");
scanf("%s",str1);
printf("Enter the string2:");
scanf("%s",str2);
pthread_create(&t1,NULL,concat,NULL);
pthread_join(t1,NULL);
printf("%s",str3);
return 0;
}
```

2.

```
#include <stdio.h>
#include<stdlib.h>
#include<string.h>
#include<pthread.h>
       char length1[100];
      int length=0;
void *length_str()
{
```

```
    length=strlen(length1);
     pthread_exit(NULL);
}
int main()
{
pthread_t t1;
printf("Enter the string:");
scanf("%s",length1);
pthread_create(&t1,NULL,length_str,NULL);
pthread_join(t1,NULL);
printf("The total length of string is %d\n",length);
return 0;
}
```

3.

```
#include <stdio.h>
#include<stdlib.h>
#include<string.h>
#include<pthread.h>
int arr[10]={99,22,00,88,11,102,33,36,66,55};
void *sort(void *arg)
{
for (int i=0;i<10;i++)
    for (int j=0;j<10;j++)
    if(arr[i]<arr[j])
    {
    int temp=arr[i];
    arr[i]=arr[j];
    arr[j]=temp;
}
pthread_exit(NULL);
}
void *min(void *arg)
{
int min=arr[0];
printf("Minimum element is %d\n",min);
pthread_exit(NULL);
}
void *max(void *arg)
{
int max=arr[9];
printf("Maximum element is %d\n",max);
pthread_exit(NULL);
}
void *avg(void *arg)
{
int sum=0;
for (int i=0;i<10;i++)
{
```

```
    sum=sum+arr[i];
    }
    sum=sum/10;
    printf("Average element is %d\n",sum);
    pthread_exit(NULL);
    }
    int main()
    {
    printf("Old array:\n");
    for (int j=0;j<10;j++)
    {
    printf("%d\n",arr[j]);
    }
    pthread_t sort_thread,max_thread,min_thread,avg_thread;
    pthread_create(&sort_thread,NULL,sort,NULL);
    pthread_join(sort_thread,NULL);
    pthread_create(&max_thread,NULL,max,NULL);
    pthread_join(max_thread,NULL);
    pthread_create(&min_thread,NULL,min,NULL);
    pthread_join(min_thread,NULL);
    pthread_create(&avg_thread,NULL,avg,NULL);
    pthread_join(avg_thread,NULL);
    return 0;
    }
```

4.

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

#define ARRAY_SIZE 10

// Structure to pass arguments to the threads
typedef struct {
   int* array;
   int start;
   int end;
} ThreadArgs;

// Function to merge and sort two sorted arrays
void merge(int* arr, int start, int mid, int end) {
   int i = start;
   int j = mid + 1;
   int k = 0;
   int temp[end - start + 1];

   while (i <= mid && j <= end) {
      if (arr[i] <= arr[j]) {
         temp[k++] = arr[i++];
```

```
        } else {
            temp[k++] = arr[j++];
        }
    }

    while (i <= mid) {
        temp[k++] = arr[i++];
    }

    while (j <= end) {
        temp[k++] = arr[j++];
    }

    for (i = start; i <= end; i++) {
        arr[i] = temp[i - start];
    }
}

// Function to sort a portion of the array
void* sort(void* arg) {
    ThreadArgs* args = (ThreadArgs*)arg;
    int* array = args->array;
    int start = args->start;
    int end = args->end;

    // Sorting the portion of the array
    for (int i = start; i <= end; i++) {
        for (int j = i + 1; j <= end; j++) {
            if (array[i] > array[j]) {
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
    }

    pthread_exit(NULL);
}

// Function to create and join threads for sorting
void createAndJoinThreads(pthread_t* threads, ThreadArgs* args, int num_threads) {
    for (int i = 0; i < num_threads; i++) {
        pthread_create(&threads[i], NULL, sort, (void*)&args[i]);
    }

    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }
}
```

```
int main() {
   int array[ARRAY_SIZE] = {9, 5, 2, 7, 1, 8, 4, 3, 6, 0};
   pthread_t threads[3];
   ThreadArgs args[2];

   // Divide the array into two halves
   int mid = ARRAY_SIZE / 2;

   // Thread arguments for the first half of the array
   args[0].array = array;
   args[0].start = 0;
   args[0].end = mid - 1;

   // Thread arguments for the second half of the array
   args[1].array = array;
   args[1].start = mid;
   args[1].end = ARRAY_SIZE - 1;

   // Create and join threads for sorting each half of the array
   createAndJoinThreads(threads, args, 2);

   // Merge and sort the two sorted halves
   merge(array, 0, mid - 1, ARRAY_SIZE - 1);

   // Print the sorted array
   printf("Sorted array:\n");
   for (int i = 0; i < ARRAY_SIZE; i++) {
      printf("%d ", array[i]);
   }
   printf("\n");

   return 0;
}
```

5.

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

#define NUM_THREADS 5

// Structure to pass arguments to the thread functions
typedef struct {
   int thread_id;
} ThreadArgs;

// Function executed by each thread
void *thread_function(void *args) {
   ThreadArgs *thread_args = (ThreadArgs *)args;
```

```
    int thread_id = thread_args->thread_id;

    printf("Thread %d: Executing\n", thread_id);

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    ThreadArgs thread_args[NUM_THREADS];

    // Creating multiple threads
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_args[i].thread_id = i + 1;  // IDs start from 1

        if (pthread_create(&threads[i], NULL, thread_function, (void *)&thread_args[i]) != 0) {
            perror("pthread_create");
            exit(EXIT_FAILURE);
        }
    }

    // Waiting for all threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_join(threads[i], NULL) != 0) {
            perror("pthread_join");
            exit(EXIT_FAILURE);
        }
    }

    printf("All threads have finished execution.\n");

    return 0;
}
```

6.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h> // for sleep()

#define NUM_THREADS 2

// Global variable to indicate if the thread should continue running
int keep_running = 1;

// Thread function demonstrating graceful termination
void *graceful_thread_function(void *arg) {
    int thread_id = *((int *)arg);

    printf("Graceful Thread %d: Started\n", thread_id);
```

```
    // Loop until instructed to stop
    while (keep_running) {
        printf("Graceful Thread %d: Running\n", thread_id);
        sleep(1);
    }

    printf("Graceful Thread %d: Cleanup\n", thread_id);

    // Cleanup resources
    // Simulating cleanup by printing a message
    printf("Graceful Thread %d: Exiting\n", thread_id);

    pthread_exit(NULL);
}

// Thread function demonstrating abrupt termination
void *abrupt_thread_function(void *arg) {
    int thread_id = *((int *)arg);

    printf("Abrupt Thread %d: Started\n", thread_id);

    // Loop indefinitely without cleanup
    while (1) {
        printf("Abrupt Thread %d: Running\n", thread_id);
        sleep(1);
    }

    // Cleanup code will never be reached

    pthread_exit(NULL);
}

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

    // Create threads for graceful termination
    for (int i = 0; i < NUM_THREADS; ++i) {
        thread_ids[i] = i + 1;
        if (pthread_create(&threads[i], NULL, graceful_thread_function, &thread_ids[i]) != 0) {
            perror("pthread_create");
            exit(EXIT_FAILURE);
        }
    }

    // Wait for some time before terminating the threads gracefully
    sleep(5);
```

```
    // Set the flag to stop the threads
    keep_running = 0;

    // Join the threads to wait for their termination
    for (int i = 0; i < NUM_THREADS; ++i) {
        if (pthread_join(threads[i], NULL) != 0) {
            perror("pthread_join");
            exit(EXIT_FAILURE);
        }
    }

    printf("Graceful thread termination complete.\n");

    // Create threads for abrupt termination
    for (int i = 0; i < NUM_THREADS; ++i) {
        thread_ids[i] = i + 1;
        if (pthread_create(&threads[i], NULL, abrupt_thread_function, &thread_ids[i]) != 0) {
            perror("pthread_create");
            exit(EXIT_FAILURE);
        }
    }

    // Wait for some time before abruptly terminating the threads
    sleep(5);

    // Cancel the threads abruptly
    for (int i = 0; i < NUM_THREADS; ++i) {
        if (pthread_cancel(threads[i]) != 0) {
            perror("pthread_cancel");
            exit(EXIT_FAILURE);
        }
    }

    printf("Abrupt thread termination complete.\n");

    return 0;
}
```