21.Worst Fit Algorithm

Aim: to develop a c program to implement worst fit algorithm of memory management

Algorithm:

1. Initialize memory blocks.
2. Define worst Fit(size):
   a. Find the largest block that fits the process of size 'size'.
   b. If found, allocate the memory; otherwise, return -1.
3. Define deallocate Memory (block Index, size):
   a. Increase the size of the specified block.
4. Define display Memory Status ():
   a. Print the current status of memory blocks.
5. In the main program:
   a. Initialize memory.
   b. Accept memory size.
   c. Enter a loop for the user menu:
      i. Accept user choice.
      ii. Perform actions based on user choice:
         - Allocate memory using worst Fit(size).
         - Deallocate memory using deallocate Memory (block Index, size).
         - Display memory status using display Memory Status ().
         - Exit the program.

End of Algorithm.

Code:

```
#include <stdio.h>
#define MAX_BLOCKS 100
int memory [MAX_BLOCKS];
int process Size [MAX_BLOCKS];

// Function prototypes
void initialize Memory (int size);
int worst Fit (int size);
```

```
void deallocateMemory(int blockIndex, int size);
void displayMemoryStatus(int size);

int main() {
    int memorySize, choice, processSize, blockIndex;

    printf("Enter the size of memory: ");
    scanf("%d", &memorySize);

    initializeMemory(memorySize);

    while (1) {
        printf("\n1. Allocate Memory\n2. Deallocate Memory\n3. Display Memory Status\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the size of the process: ");
                scanf("%d", &processSize);
                blockIndex = worstFit(processSize);
                if (blockIndex != -1) {
                    printf("Memory allocated successfully at block %d\n", blockIndex);
                    displayMemoryStatus(memorySize);
                } else {
                    printf("Memory allocation failed. No suitable block found.\n");
                }
                break;

            case 2:
                printf("Enter the block index to deallocate: ");
```

```
            scanf("%d", &blockIndex);
            deallocateMemory(blockIndex, memory[blockIndex]);
            printf("Memory deallocated successfully from block %d\n", blockIndex);
            displayMemoryStatus(memorySize);
            break;

        case 3:
            displayMemoryStatus(memorySize);
            break;

        case 4:
            return 0;

        default:
            printf("Invalid choice. Please enter a valid option.\n");
      }
   }

   return 0;
}

void initializeMemory(int size) {
   for (int i = 0; i < MAX_BLOCKS; i++) {
      memory[i] = size;
   }
}

int worstFit(int size) {
   int maxBlockSize = -1;
   int blockIndex = -1;

   for (int i = 0; i < MAX_BLOCKS; i++) {
```

```
        if (memory[i] >= size && memory[i] > maxBlockSize) {
            maxBlockSize = memory[i];
            blockIndex = i;
        }
    }

    if (blockIndex != -1) {
        memory[blockIndex] -= size;
    }

    return blockIndex;
}

void deallocateMemory(int blockIndex, int size) {
    memory[blockIndex] += size;
}

void displayMemoryStatus(int size) {
    printf("Memory Status:\n");

    for (int i = 0; i < MAX_BLOCKS; i++) {
        printf("Block %d: %d\n", i, memory[i]);
    }
}
```

Input:

Memory blocks: [10, 20, 5, 15] Processes: [12, 7, 10]

Output:

Process 1 (12 units) allocated to Memory Block 2 (20 units).

Process 2 (7 units) allocated to Memory Block 3 (15 units).

Process 3 (10 units) allocated to Memory Block 2 (20 units).

22. Best Fit Algorithm

Aim: Construct a C program to implement best fit algorithm of memory management.

Algorithm:

Algorithm: Best-Fit Memory Management

1. Initialize memory blocks.
2. Function bestFit(size):
   a. Find the smallest block that fits the process of size 'size'.
   b. If found, allocate the memory; otherwise, return -1.
3. Function deallocateMemory(blockIndex, size):
   a. Increase the size of the specified block.
4. Function displayMemoryStatus():
   a. Print the current status of memory blocks.
5. In the main program:
   a. Initialize memory.
   b. Accept memory size.
   c. Enter a loop for the user menu:
      i. Accept user choice.
      ii. Perform actions based on user choice:
         - Allocate memory using bestFit(size).
         - Deallocate memory using deallocateMemory(blockIndex, size).
         - Display memory status using displayMemoryStatus().
         - Exit the program.

End of Algorithm.

Code:

```
#include <stdio.h>

#define MAX_BLOCKS 100

int memory[MAX_BLOCKS];
int processSize[MAX_BLOCKS];

// Function prototypes
void initializeMemory(int size);
int bestFit(int size);
void deallocateMemory(int blockIndex, int size);
void displayMemoryStatus(int size);

int main() {
    int memorySize, choice, processSize, blockIndex;

    printf("Enter the size of memory: ");
    scanf("%d", &memorySize);

    initializeMemory(memorySize);

    while (1) {
        printf("\n1. Allocate Memory\n2. Deallocate Memory\n3. Display Memory Status\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the size of the process: ");
```

```
            scanf("%d", &processSize);
            blockIndex = bestFit(processSize);
            if (blockIndex != -1) {
                printf("Memory allocated successfully at block %d\n", blockIndex);
                displayMemoryStatus(memorySize);
            } else {
                printf("Memory allocation failed. No suitable block found.\n");
            }
            break;

        case 2:
            printf("Enter the block index to deallocate: ");
            scanf("%d", &blockIndex);
            deallocateMemory(blockIndex, memory[blockIndex]);
            printf("Memory deallocated successfully from block %d\n", blockIndex);
            displayMemoryStatus(memorySize);
            break;

        case 3:
            displayMemoryStatus(memorySize);
            break;
case 4:
            return 0;

        default:
            printf("Invalid choice. Please enter a valid option.\n");
      }
   }

   return 0;
}
```

```
void initializeMemory(int size) {
    for (int i = 0; i < MAX_BLOCKS; i++) {
        memory[i] = size;
    }
}

int bestFit(int size) {
    int minBlockSize = memory[MAX_BLOCKS - 1] + 1;
    int blockIndex = -1;

    for (int i = 0; i < MAX_BLOCKS; i++) {
        if (memory[i] >= size && memory[i] < minBlockSize) {
            minBlockSize = memory[i];
            blockIndex = i;
        }
    }

    if (blockIndex != -1) {
        memory[blockIndex] -= size;
    }

    return blockIndex;
}

void deallocateMemory(int blockIndex, int size) {
    memory[blockIndex] += size;
}

void displayMemoryStatus(int size) {
    printf("Memory Status:\n");

    for (int i = 0; i < MAX_BLOCKS; i++) {
```

```
        printf("Block %d: %d\n", i, memory[i]);
    }
}
```

Input:

Memory blocks: [10, 20, 5, 15]

Processes: [12, 7, 10]

Output:

Process 1 (12 units) allocated to Memory Block 2 (20 units).

Process 2 (7 units) allocated to Memory Block 1 (10 units).

Process 3 (10 units) allocated to Memory Block 4 (15 units).

23.First Fit Algorithm

Aim: to construct a c program to implement first fit algorithm of memory management

Algorithm:

1. **Initialize Memory:**
   - Create a memory array with blocks, each having size and an allocated flag.
2. **Memory Allocation Function:**
   - Iterate through blocks.
   - If an unallocated block is found with sufficient size, allocate it and return the index.
   - If no suitable block is found, return -1.
3. **Main Program:**
   - Initialize memory.
   - Accept memory requests until the user exits.
4. **Display Function:**
   - Display the current state of memory.
5. **User Input Loop:**
   - Display memory state.
   - Accept user input for memory requests.

- If the user enters 0, exit.
- Allocate memory and display the result.

6. **Exit:**
- End the program when the user exits.

Code:

```
#include <stdio.h>
#define MAX_MEMORY 1000
// Structure to represent a memory block
struct MemoryBlock {
    int size;
    int allocated; // 0 for unallocated, 1 for allocated
};
// Function to initialize the memory blocks
void initializeMemory(struct MemoryBlock memory[], int size) {
    for (int i = 0; i < size; i++) {
        memory[i].size = 0;
        memory[i].allocated = 0;
    }
}

// Function to display the current state of memory
void displayMemory(struct MemoryBlock memory[], int size) {
    printf("Memory State:\n");
    for (int i = 0; i < size; i++) {
        printf("Block %d: Size=%d, Allocated=%s\n", i, memory[i].size,
               memory[i].allocated ? "Yes" : "No");
    }
    printf("\n");
}

// Function to allocate memory using First Fit algorithm
int allocateMemory(struct MemoryBlock memory[], int size, int requestSize) {
```

```
    for (int i = 0; i < size; i++) {
        if (!memory[i].allocated && memory[i].size >= requestSize) {
            memory[i].allocated = 1;
            return i; // Return the index of the allocated block
        }
    }
    return -1; // No suitable block found
}
int main() {
    struct MemoryBlock memory[MAX_MEMORY];
    int memorySize, requestSize, blockIndex;
    printf("Enter the size of memory: ");
    scanf("%d", &memorySize);
    initializeMemory(memory, memorySize);
    while (1) {
        displayMemory(memory, memorySize);
        printf("Enter the size of memory request (or enter 0 to exit): ");
        scanf("%d", &requestSize);
        if (requestSize == 0) {
            printf("Exiting the program.\n");
            break;
        }
        blockIndex = allocateMemory(memory, memorySize, requestSize);
        if (blockIndex != -1) {
            printf("Memory allocated successfully in block %d.\n", blockIndex);
        } else {
            printf("No suitable block found for allocation.\n");
        }
    }

    return 0;
}
```

Input:

Memory blocks: [10, 20, 5, 15]

Processes: [12, 7, 10]

Output:

Process 1 (12 units) allocated to Memory Block 2 (20 units).

Process 2 (7 units) allocated to Memory Block 1 (10 units).

Process 3 (10 units) allocated to Memory Block 2 (20 units).

24. Demonstration of UNIX System Calls for File Management

Aim: to design a c program to demonstrate unix system calls for file management.

Algorithm:

1. **Open/Create File for Writing:**
   - Use `open` to create or open a file for writing.
2. **Write Data to File:**
   - Use `write` to write data to the file.
3. **Close File:**
   - Use `close` to close the file.
4. **Open File for Reading:**
   - Use `open` to open the file for reading.
5. **Read Data from File:**
   - Use `read` to read data from the file.
6. **Close File Again:**
   - Use `close` to close the file after reading.
7. **Display Results:**
   - Print the data written to and read from the file.
8. **Exit:**
   - End the program.

Code:

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

int main() {
    // File descriptor
    int fd;
    // Create a new file or open an existing file for writing
    fd = open("example.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    // Write data to the file
    const char *data = "Hello, UNIX System Calls!";
    ssize_t bytes_written = write(fd, data, strlen(data));
    if (bytes_written == -1) {
        perror("write");
        close(fd);
        exit(EXIT_FAILURE);
    }
    printf("Data written to the file: %s\n", data);

    // Close the file
    if (close(fd) == -1) {
        perror("close");
        exit(EXIT_FAILURE);
    }
```

```
    // Open the file for reading
    fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    // Read data from the file
    char buffer[100];
    ssize_t bytes_read = read(fd, buffer, sizeof(buffer) - 1);
    if (bytes_read == -1) {
        perror("read");
        close(fd);
        exit(EXIT_FAILURE);
    }
    buffer[bytes_read] = '\0'; // Null-terminate the string
    printf("Data read from the file: %s\n", buffer);
    // Close the file
    if (close(fd) == -1) {
        perror("close");
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

Input:

Create a file named "example.txt".

Write "Hello, world!" to the file.

Read the contents of "example.txt" and print them.

Output:

File "example.txt" created successfully.

"Hello, world!" written to "example.txt".

Contents of "example.txt": "Hello, world!"

25. Construct a C program to implement the I/O system calls of UNIX

Aim: to construct a c program to implement the I/O system calls of UNIX (fcntl,seek,stat,opendir,readir).

Algorithm:

1.Open a File: Use open() system call to open a file.

2.Read/Write Operations: Use read() and write() system calls to perform read and write operations on the file.

3.File Control: Use fcntl() system call to control file-related properties.

4.Seek Operation: Use lseek() system call to change the file offset.

5.File Status: Use stat() system call to retrieve information about a file.

6.Directory Operations: Use opendir() and readdir() system calls to open and read directories.

Code:

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

int main() {
   // Demonstrate fcntl
   int fd = open("example.txt", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
   if (fd == -1) {
      perror("open");
      exit(EXIT_FAILURE);
   }

   // Use fcntl to set the file status flags (here, set it to non-blocking)
```

```
int flags = fcntl(fd, F_GETFL);
flags |= O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) == -1) {
    perror("fcntl");
    close(fd);
    exit(EXIT_FAILURE);
}

printf("File 'example.txt' opened with non-blocking flag.\n");

// Demonstrate lseek
off_t offset = lseek(fd, 5, SEEK_SET);
if (offset == -1) {
    perror("lseek");
    close(fd);
    exit(EXIT_FAILURE);
}

printf("File pointer moved to offset 5.\n");

// Demonstrate stat
struct stat fileStat;
if (fstat(fd, &fileStat) == -1) {
    perror("fstat");
    close(fd);
    exit(EXIT_FAILURE);
}

printf("File Size: %lld bytes\n", (long long)fileStat.st_size);

// Close the file
if (close(fd) == -1) {
```

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

    // Demonstrate opendir and readdir
    DIR *dir = opendir(".");
    if (dir == NULL) {
        perror("opendir");
        exit(EXIT_FAILURE);
    }

    printf("\nFiles in the current directory:\n");

    struct dirent *dp;
    while ((dp = readdir(dir)) != NULL) {
        printf("%s\n", dp->d_name);
    }

    // Close the directory
    closedir(dir);

    return 0;
}
```

Output:

```
File 'example.txt' opened with non-blocking flag.
File pointer moved to offset 5.
File Size: 5 bytes
Files in the current directory:
..
example.txt
other_file.txt
subdirectory
```

26. Construct a C program to implement the file management operations.

Aim: to construct a c program to implement the file management operations.

Algorithm:

1. **File Creation and Writing:**
   - Open a file named "example.txt" for writing using `fopen`.
   - Check if the file is successfully opened; if not, display an error message and exit.
   - Write the string "Hello, File Management!" to the file using `fprintf`.
   - Close the file using `fclose`.
2. **File Reading:**
   - Open the same file for reading using `fopen`.
   - Check if the file is successfully opened; if not, display an error message and exit.
   - Read content from the file using `fgets` into a buffer.
   - Display the content read from the file.
   - Close the file using `fclose`.
3. **Exit:**
   - End the program.

Code:

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

int main() {
    FILE *file; // File pointer

    // File creation and writing
    file = fopen("example.txt", "w");
    if (file == NULL) {
        perror("Error creating file");
        exit(EXIT_FAILURE);
    }

    fprintf(file, "Hello, File Management!\n");
```

```
    fclose(file);

    printf("File 'example.txt' created and written to.\n");

    // File reading
    file = fopen("example.txt", "r");
    if (file == NULL) {
        perror("Error opening file for reading");
        exit(EXIT_FAILURE);
    }

    char buffer[100];
    fgets(buffer, sizeof(buffer), file);

    printf("Content read from file: %s", buffer);

    fclose(file);

    return 0;
}
```

Output:

File 'example.txt' created and written to.

Content read from file: Hello, File Management!

27. Develop a C program for simulating the function of ls UNIX Command.

Aim:

The aim is to create a C program that replicates the basic functionality of the ls command in UNIX-like operating systems. The program should list the contents of a directory specified by the user, showing file names, permissions, sizes, and other relevant information.

Algorithm:

1.Accept User Input: Accept user input to specify the directory whose contents need to be listed. If no directory is specified, default to the current directory.

2.Open Directory: Use the opendir() function to open the specified directory.

3.Read Directory Contents: Use the readdir() function to read the contents of the directory one by one.

4.Process Directory Entries: For each entry read:

Extract information such as file name, permissions, size, etc.

Print this information in a formatted manner.

5.Close Directory: Use the closedir() function to close the directory once all entries have been read.

6.Handle Errors: Handle any errors that may occur during the process, such as failure to open the directory or read its contents.

Code:

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

int main(int argc, char *argv[]) {
    DIR *dir;
    struct dirent *entry;

    // Open the specified directory or use the current directory if none is provided
    const char *path = (argc > 1) ? argv[1] : ".";
```

```
    dir = opendir(path);

    if (dir == NULL) {
        perror("Error opening directory");
        exit(EXIT_FAILURE);
    }

    // List files in the directory
    printf("Files in directory '%s':\n", path);
    while ((entry = readdir(dir)) != NULL) {
        printf("%s\n", entry->d_name);
    }

    closedir(dir);

    return 0;
}
```

Output:

```
./ls_simulation /path/to/directory
```

or if no directory is specified:

```bash
./ls_simulation
```

Example output:

```
Files in directory '/path/to/directory':
file1.txt
file2.c
subdirectory
...
```

28. Write a C program for the simulation of GREP UNIX command

Aim:

The aim is to develop a C program that simulates the basic functionality of the grep UNIX command. The program should search for a specified pattern in a given file or standard input and print the lines that match the pattern.

Algorithm:

1.Accept User Input: Accept user input including the pattern to search for and the file(s) to search within. If no file is specified, read from standard input.

2.Open File(s) (if specified): Use file I/O functions to open the file(s) for reading.

3.Read Lines: Read lines from the file(s) or standard input one by one.

4.Search for Pattern: For each line read, search for the specified pattern within the line.

5.Print Matching Lines: If the pattern is found within the line, print the line to the standard output.

6.Close File(s): Close the file(s) after reading if they were opened.

7.Handle Errors: Handle any errors that may occur during file operations or pattern matching.

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX_LINE_LENGTH 1024


void grep(const char *pattern, FILE *file);
```

```
int main(int argc, char *argv[]) {
    const char *pattern;
    FILE *file;

    // Ensure proper usage
    if (argc < 2) {
        fprintf(stderr, "Usage: %s pattern [file...]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // Get pattern from command line argument
    pattern = argv[1];

    // If files are specified, open each file and grep
    if (argc >= 3) {
        for (int i = 2; i < argc; i++) {
            file = fopen(argv[i], "r");
            if (file == NULL) {
                perror(argv[i]);
                continue;
            }
            grep(pattern, file);
            fclose(file);
        }
```

```
    } else { // Grep from standard input
        grep(pattern, stdin);
    }

    return 0;
}

void grep(const char *pattern, FILE *file) {
    char line[MAX_LINE_LENGTH];

    // Read lines from file or stdin
    while (fgets(line, sizeof(line), file) != NULL) {
        // Search for pattern in line
        if (strstr(line, pattern) != NULL) {
            // Print the line containing the pattern
            printf("%s", line);
        }
    }
}
```

Sample input:

This is a sample text file.
It contains multiple lines.
We will use this file for demonstration.

Output:

This is a sample text file.
We will use this file for demonstration.

29. Write a C program to simulate the solution of Classical Process Synchronization Problem

Algorithm:

1.Initialize the Buffer: Create a fixed-size buffer that can hold a maximum of N items.

2.Initialize Semaphores/Mutexes: Initialize synchronization primitives such as semaphores or mutexes to control access to the shared buffer.

For example, initialize a semaphore to track the number of empty slots in the buffer (emptyCount) and a semaphore to track the number of filled slots in the buffer (fillCount).

3.Create Producer and Consumer Threads: Create multiple producer and consumer threads, each executing their respective producer and consumer functions.

4.Producer Function:

Wait on the emptyCount semaphore to ensure there is space in the buffer.

Acquire a mutex to protect the critical section (access to the buffer).

Produce an item and add it to the buffer.

Release the mutex.

Signal the fillCount semaphore to indicate that the buffer has a new item.

5.Consumer Function:

Wait on the fillCount semaphore to ensure there are items in the buffer.

Acquire a mutex to protect the critical section (access to the buffer).

Consume an item from the buffer.

Release the mutex.

Signal the emptyCount semaphore to indicate that the buffer has an empty slot.

6.Handle Thread Creation and Joining: Create and join producer and consumer threads, ensuring proper synchronization and termination.

7.Handle Cleanup: Properly release resources like semaphores or mutexes and perform any necessary cleanup before exiting the program.

Code:

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

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0, out = 0, count = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t full = PTHREAD_COND_INITIALIZER;
pthread_cond_t empty = PTHREAD_COND_INITIALIZER;

void *producer(void *arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        item = rand() % 100;  // Simulate producing an item
        pthread_mutex_lock(&mutex);

        while (count == BUFFER_SIZE) {
            pthread_cond_wait(&empty, &mutex);
        }

        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        count++;

        printf("Produced: %d\n", item);

        pthread_cond_signal(&full);
        pthread_mutex_unlock(&mutex);
```

```
    }

    pthread_exit(NULL);
}

void *consumer(void *arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        pthread_mutex_lock(&mutex);

        while (count == 0) {
            pthread_cond_wait(&full, &mutex);
        }

        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;

        printf("Consumed: %d\n", item);

        pthread_cond_signal(&empty);
        pthread_mutex_unlock(&mutex);
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t producer_thread, consumer_thread;

    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);
```

```
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    return 0;
}
```

Output:

```
Produced: 24
Produced: 56
Produced: 92
Produced: 11
Produced: 78
Consumed: 24
Produced: 35
Produced: 99
Produced: 43
Produced: 72
Consumed: 56
Consumed: 92
Consumed: 11
Produced: 64
Produced: 89
Produced: 33
Consumed: 78
Consumed: 35
Produced: 54
Produced: 21
Produced: 66
Produced: 48
Consumed: 99
Consumed: 43
Consumed: 72
```

Produced: 87

Consumed: 64

Consumed: 89

Consumed: 33

Produced: 12

Produced: 67

Produced: 23

Produced: 44

Produced: 79

Consumed: 54

Consumed: 21

Consumed: 66

Consumed: 48

Consumed: 87

Consumed: 12

Consumed: 67

Consumed: 23

Consumed: 44

Consumed: 79

30. Write C programs to demonstrate the following thread relatedconcepts

Aim:

The aim is to create simple C programs to demonstrate various thread-related concepts. These concepts include thread creation, thread synchronization, mutual exclusion, thread joining, and thread termination.

Algorithm:

1.Thread Creation:

Use the appropriate function (pthread_create() in POSIX) to create a new thread.

Provide the entry point function for the thread, which will be executed when the thread starts.

Pass any necessary parameters to the thread function.

2.Thread Synchronization:

Use synchronization primitives such as mutexes, semaphores, or condition variables to synchronize access to shared resources among multiple threads.

Acquire the synchronization primitive before accessing the shared resource and release it afterward to ensure mutual exclusion.

3.Mutual Exclusion:

Use mutexes to achieve mutual exclusion, ensuring that only one thread can access a shared resource at a time.

Acquire the mutex before accessing the shared resource and release it afterward to allow other threads to access it.

4.Thread Joining:

Use the pthread_join() function to wait for a thread to terminate before proceeding with further execution.

Joining ensures that the main thread or another thread waits for the specified thread to complete its execution.

4.Thread Termination:

Use the appropriate mechanism to terminate a thread's execution when its task is completed.

This could involve returning from the thread function or explicitly calling pthread_exit().

5.Error Handling:

Implement error handling to handle potential errors that may occur during thread creation, synchronization, or other operations.

Check return values of thread-related functions for errors and handle them appropriately.

PROGRAM:

```
#include <stdio.h>

#include <pthread.h>
```

```
#define NUM_THREADS 3

void *thread_function(void *thread_id) {
    long tid = (long)thread_id;
    printf("Thread %ld created.\n", tid);
    pthread_exit(NULL);
}

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

    for (i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, thread_function, (void *)(intptr_t)(i + 1));
    }

    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
        printf("Thread %d terminated.\n", i + 1);
    }

    return 0;
}
```

Output:

Thread 1 created.

Thread 2 created.

Thread 3 created.

Thread 1 terminated.

Thread 2 terminated.

Thread 3 terminated.