MULTICORE ARCHITECTURE AND PROGRAMMING LABORATORY (CYL71)

LIST OF LAB PROGRAMS

PROGRAM 1: SETTING UP AND RUNNING A PARALLEL PROGRAM USING MPI (MESSAGE PASSING INTERFACE) IN A LINUX ENVIRONMENT.

PROGRAM 2: SOFTWARE FRAMEWORKS AND LIBRARIES TO DEMONSTRATE MULTI-CORE ARCHITECTURE AND PARALLEL PROGRAMMING.

PROGRAM 3: IMPLEMENT PARALLEL PROCESSING TO CALCULATE THE SUM OF AN ARRAY.

PROGRAM 4: IMPLEMENT CONCURRENT SORTING OF AN ARRAY USING THE MERGE SORT ALGORITHM.

PROGRAM 5: IMPLEMENT MULTI-THREADING TO SIMULATE CONCURRENT BANK TRANSACTIONS.

PROGRAM 6: IMPLEMENT A PRODUCER-CONSUMER MODEL USING THREADS FOR TASK MANAGEMENT.

PROGRAM 7: IMPLEMENT MUTEXES FOR SYNCHRONIZING BANKING TRANSACTIONS AMONG THREADS.

PROGRAM 8: IMPLEMENT CONDITION VARIABLES TO MANAGE THREAD COMMUNICATION IN PRODUCTION SCENARIOS.

PROGRAM 9: IMPLEMENT PARALLEL ARRAY SUMMATION USING OPENMP FOR PERFORMANCE OPTIMIZATION.

PROGRAM 10: IMPLEMENT TASK PARALLELISM IN IMAGE PROCESSING APPLICATIONS USING OPENMP.

PROGRAM 11: IMPLEMENT TIMEOUT MECHANISMS TO HANDLE DEADLOCKS EFFECTIVELY.

PROGRAM 12: IMPLEMENT A WORK-STEALING SCHEDULER TO OPTIMIZE TASK PROCESSING AMONG THREADS.

PROGRAM 1: SETTING UP AND RUNNING A PARALLEL PROGRAM USING MPI (MESSAGE PASSING INTERFACE) IN A LINUX ENVIRONMENT.

```
sudo su

apt install mpich

gedit pp.c

mpicc -o pp pp.c

mpirun -np 5 ./pp
```

CODE: MPI HELLO WORLD

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

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

OUTPUT:

```
root@admin:/home/administrator# gedit pp1.c

(gedit:5979): dconf-WARNING **: 10:05:32.131: failed to commit changes to dconf: Failed to execute child process “dbus-launch” (No such file or directory)

(gedit:5979): dconf-WARNING **: 10:05:32.132: failed to commit changes to dconf: Failed to execute child process “dbus-launch” (No such file or directory)

(gedit:5979): dconf-WARNING **: 10:05:32.250: failed to commit changes to dconf: Failed to execute child process “dbus-launch” (No such file or directory)

(gedit:5979): dconf-WARNING **: 10:05:32.250: failed to commit changes to dconf: Failed to execute child process “dbus-launch” (No such file or directory)

(gedit:5979): dconf-WARNING **: 10:05:32.250: failed to commit changes to dconf: Failed to execute child process “dbus-launch” (No such file or directory)

** (gedit:5979): WARNING **: 10:05:34.326: Set document metadata failed: Setting attribute metadata::gedit-spell-language not supported

** (gedit:5979): WARNING **: 10:05:34.327: Set document metadata failed: Setting attribute metadata::gedit-encoding not supported

** (gedit:5979): WARNING **: 10:05:36.237: Set document metadata failed: Setting attribute metadata::gedit-position not supported

(gedit:5979): dconf-WARNING **: 10:05:36.253: failed to commit changes to dconf: Failed to execute child process “dbus-launch” (No such file or directory)
root@admin:/home/administrator# mpicc -o pp1 pp1.c
root@admin:/home/administrator# mpirun -np 5 ./pp1
Hello world from processor admin, rank 0 out of 5 processors
Hello world from processor admin, rank 1 out of 5 processors
Hello world from processor admin, rank 3 out of 5 processors
Hello world from processor admin, rank 2 out of 5 processors
Hello world from processor admin, rank 4 out of 5 processors
root@admin:/home/administrator# mpirun -np 11 ./pp1
Hello world from processor admin, rank 1 out of 11 processors
Hello world from processor admin, rank 7 out of 11 processors
Hello world from processor admin, rank 0 out of 11 processors
Hello world from processor admin, rank 4 out of 11 processors
Hello world from processor admin, rank 5 out of 11 processors
Hello world from processor admin, rank 9 out of 11 processors
Hello world from processor admin, rank 3 out of 11 processors
Hello world from processor admin, rank 6 out of 11 processors
Hello world from processor admin, rank 8 out of 11 processors
Hello world from processor admin, rank 2 out of 11 processors
Hello world from processor admin, rank 10 out of 11 processors
root@admin:/home/administrator#
```

PROGRAM 2: SOFTWARE FRAMEWORKS AND LIBRARIES TO DEMONSTRATE MULTI-CORE ARCHITECTURE AND PARALLEL PROGRAMMING.

There are several software frameworks and libraries that you can use to demonstrate multi-core architecture and parallel programming. Here are some popular ones:

1. OPENMP

- Description: A widely used API for multi-platform shared-memory multiprocessing programming in C, C++, and Fortran.

- Key Features:

- Easy to use with compiler directives.

- Good for parallelizing loops and sections of code.

- Use Case: Suitable for applications where tasks can be performed concurrently on shared data.

2. PTHREADS (POSIX THREADS)

- Description: A C library for creating and managing threads in shared-memory environments.

- Key Features:

- Fine-grained control over thread creation and management.

- Good for low-level thread handling.

- Use Case: Useful for applications requiring precise control over thread execution and synchronization.

3. CUDA

- Description: A parallel computing platform and application programming interface model created by NVIDIA for general computing on GPUs.

- Key Features:

- Designed to leverage the massive parallel processing power of NVIDIA GPUs.

- Supports C/C++ and Fortran.

- Use Case: Ideal for compute-intensive tasks such as scientific simulations, image processing, and machine learning.

4. MPI (Message Passing Interface)

- **Description:** As you've already seen, MPI is designed for parallel programming in distributed-memory systems.

- **Key Features:**

  - Efficient communication between processes.

  - Scalable for large clusters.

- **Use Case:** Ideal for high-performance computing tasks that require communication between nodes in a cluster.

5. OPENCL

- **Description:** An open standard for parallel programming across heterogeneous platforms, including CPUs, GPUs, and other processors.

- **Key Features:**

  - Allows code to run on various hardware types.

  - Good for cross-platform applications.

- **Use Case:** Suitable for applications that require flexibility in execution across different hardware.

6. Threading Building Blocks (TBB)

- **Description:** An Intel library that helps in creating parallel applications in C++.

- **Key Features:**

  - Provides high-level parallel constructs, including parallel loops and task scheduling.

  - Works well with existing C++ codebases.

- **Use Case:** Useful for applications requiring task-based parallelism without getting into low-level thread management.

7. DASK

- **Description:** A flexible parallel computing library for analytics in Python.

- **Key Features:**

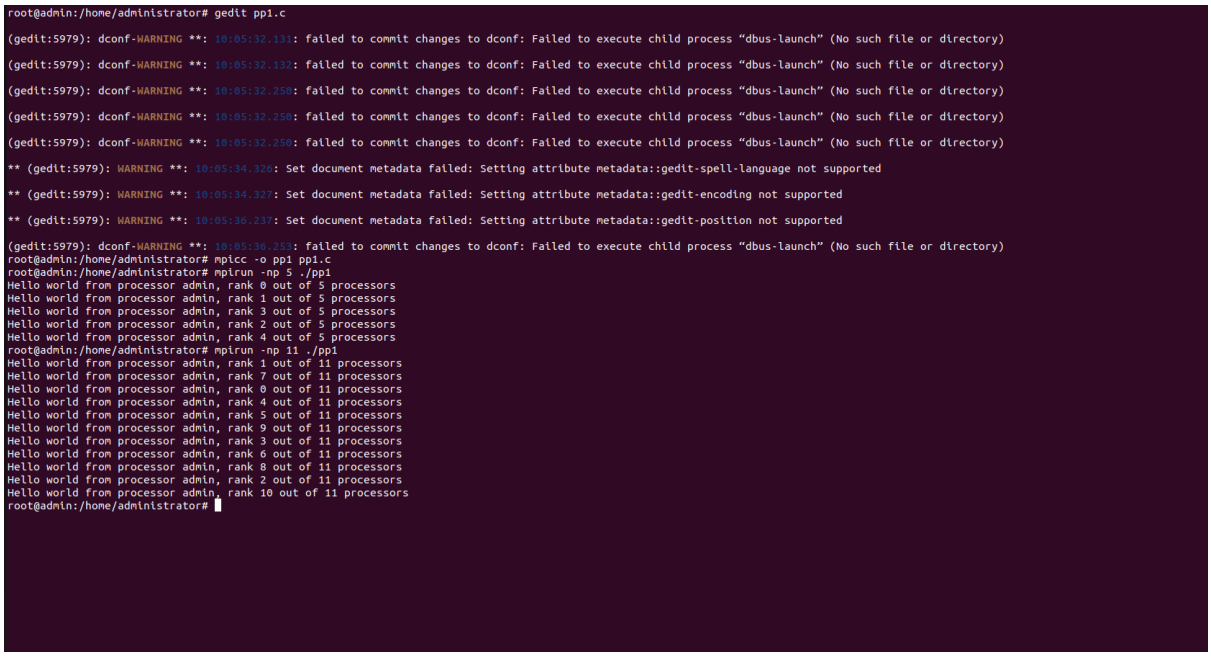
  - Allows parallel execution of Python code.

- Works with NumPy, pandas, and scikit-learn.

- **Use Case:** Ideal for data analysis and machine learning tasks that can benefit from parallel processing.

SUMMARY

Each of these frameworks and libraries has its strengths and is suited for different types of applications. Depending on your specific use case, programming language, and architecture, you can choose the one that fits best.



INSTALLATION AND EXECUTION STEPS

Let's go through the installation and execution steps for OPENMP, PTHREADS, CUDA, and MPI, detailing the commands required for each. I'll provide a clear breakdown for each framework, starting with the installation process.

1. OPENMP

Installation Steps

OPENMP is typically included in modern compilers, so you might not need a separate installation.

- For GCC on Linux:

You can check if GCC is installed by running:

```bash
gcc --version
```

If GCC is not installed, install it with:

```bash
sudo apt update
sudo apt install build-essential
```

Example Program

Create a simple OPENMP program.

1. Create a file named `omp_example.c`:

```
#include <omp.h>
#include <stdio.h>
int main() {
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        printf("Hello from thread %d\n", thread_id);
    }
    return 0;
```

```
}
```

2. Compile the program:

```
gcc -fopenmp -o omp_example omp_example.c
```

3. Run the program:

```
./omp_example
```

2. PTHREADS

Installation Steps

Pthreads is usually included with the GCC package. Just ensure you have GCC installed (as shown above).

Example Program

Create a simple Pthreads program.

1. Create a file named `pthread_example.c`:

```
#include <pthread.h>
#include <stdio.h>
void* print_message(void* thread_id) {
    int tid = *((int*)thread_id);
    printf("Hello from thread %d\n", tid);
    return NULL;
}
int main() {
    pthread_t threads[5];
    int thread_ids[5];
    for (int i = 0; i < 5; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, print_message, (void*)&thread_ids[i]);
    }
    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }
    return 0;
}
```

2. Compile the program:

```
gcc -pthread -o pthread_example pthread_example.c
```

3. Run the program:

```
./pthread_example
```

3. CUDA

Installation Steps

To install CUDA, follow these steps:

1. Download CUDA from the NVIDIA website: [NVIDIA CUDA Toolkit] (https://developer.nvidia.com/cuda-downloads).

2. Install CUDA:

- Follow the installation instructions for your specific Linux distribution.

3. Set up environment variables:

You may need to add CUDA to your PATH. Edit your `~/.bashrc` or `~/.bash_profile` file:

```bash
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
```

Then source the file:

```bash
source ~/.bashrc
```

Example Program

Create a simple CUDA program.

1. Create a file named `cuda_example.cu`:

```
#include <stdio.h>
__global__ void helloFromGPU() {
    printf("Hello from GPU thread %d\n", threadIdx.x);
}
int main() {
    helloFromGPU<<<1, 5>>>();
    cudaDeviceSynchronize();
```

```
    return 0;
}
```

2. Compile the program:

```
nvcc -o cuda_example cuda_example.cu
```

3. Run the program:

```
./cuda_example
```

4. MPI

Installation Steps

You can install MPICH (or OpenMPI) as follows:

1. Install MPICH:

```
sudo apt update
sudo apt install mpich
```

Example Program

Create a simple MPI program.

1. Create a file named `mpi_example.c`:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    printf("Hello from process %d out of %d processes\n", world_rank, world_size);
    MPI_Finalize();
    return 0;
}
```

2. Compile the program:

```
mpicc -o mpi_example mpi_example.c
```

3. Run the program:

```
mpirun -np 5 ./mpi_example
```

SUMMARY

These steps guide you through the installation and execution of programs using OPENMP, PTHREADS, CUDA, and MPI. Each framework has its unique setup and usage, so you can choose the one that fits your parallel programming needs.

PROGRAM 3: IMPLEMENT PARALLEL PROCESSING TO CALCULATE THE SUM OF AN ARRAY.

Basic Multi-Core Matrix Addition

Let’s break down the concept of multithreading and your matrix addition program using a real-life scenario.

Real-Life Scenario: Team Project

Imagine you have a team of three people (let's call them Alice, Bob, and Charlie) working on a project to combine two sets of data (like lists of numbers) into a new set. Each person is responsible for processing one part of the project simultaneously.

1. Data Sets:
- Set A: {1, 2, 3}, {4, 5, 6}, {7, 8, 9}
- Set B: {9, 8, 7}, {6, 5, 4}, {3, 2, 1}

They want to create a new set (let’s call it Set C) where each element is the sum of the corresponding elements in Set A and Set B.

2. Division of Work:
- Alice will handle the first row: {1 + 9, 2 + 8, 3 + 7}
- Bob will handle the second row: {4 + 6, 5 + 5, 6 + 4}
- Charlie will handle the third row: {7 + 3, 8 + 2, 9 + 1}

3. Simultaneous Processing:
- While Alice is working on the first row, Bob can start working on the second row, and Charlie can work on the third row at the same time. This is like how threads work in a program—each thread handles a part of the task independently and simultaneously.

How the Code Reflects This Scenario?
In your matrix addition program:
1. Matrix Representation:
- Each row of the matrices \(A \) and \(B \) corresponds to a task that a team member (thread) will perform.

2. Threads:
- Each thread represents a team member (Alice, Bob, or Charlie). When you create a thread for each row, you are effectively assigning that row to a specific team member.

3. Thread Creation:
- When you call `pthread_create`, you start a new thread (or team member) to handle the addition of that particular row. Each thread gets its own row to work on, similar to how each team member has their own section of the project.

4. Thread Identification:
- Each thread prints its unique ID when it starts processing its row. This is akin to Alice saying, "I’m Alice, and I'm working on the first row!" This helps you know which thread is working on which part of the task.

5. Joining Threads:
- After creating the threads, you wait for all of them to finish with `pthread_join`. This is like having a meeting where everyone presents their results before moving on. Only after all threads have completed their work do you print the final matrix.

Visualization of the Process

Imagine a workplace:
- **Workstations:** Each workstation is like a thread in your program.
- **Tasks:** Each team member (thread) works on their assigned task simultaneously without interfering with each other.
- **Final Review:** Once everyone is done, their results are combined into a final report (the resultant matrix).

Final Result
When you run the program:
- Each thread processes its row and stores the result in the resultant matrix \(C \).
- Finally, the program prints the combined results, just like your team would present the final project after everyone has done their part.

Conclusion
By using threads, you make your program efficient, just like a team of people working together can complete a project faster than one person working alone. This way, the total processing time is reduced, and the task can be handled in a structured, organized manner.

CODE:

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

#define SIZE 3  // Matrix size

int A[SIZE][SIZE] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
int B[SIZE][SIZE] = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};
int C[SIZE][SIZE];  // Resultant matrix

// Structure to hold both row index and thread ID
typedef struct {
    int row;
    pthread_t thread_id;
} thread_data;

// Step 1: Function to add elements of matrices
void* add_matrices(void* arg) {
    thread_data* data = (thread_data*)arg;
    int i = data->row;

    printf("Thread ID: %lu is processing row %d\n", (unsigned long)pthread_self(), i);

    for (int j = 0; j < SIZE; j++) {
        C[i][j] = A[i][j] + B[i][j];  // Element-wise addition
    }

    free(data);  // Free the allocated memory for thread data
    return NULL;
}
```

```
int main() {
    pthread_t threads[SIZE];  // Thread array

    // Step 2: Create threads for each row
    for (int i = 0; i < SIZE; i++) {
        thread_data* data = malloc(sizeof(thread_data));  // Allocate memory for thread data
        data->row = i;  // Set the row index

        pthread_create(&threads[i], NULL, add_matrices, data);
    }

    // Step 3: Join threads
    for (int i = 0; i < SIZE; i++) {
        pthread_join(threads[i], NULL);
    }

    // Step 4: Print the resultant matrix
    printf("Resultant Matrix:\n");
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

OUTPUT:

Thread ID: 135944938194496 is processing row 0

Thread ID: 135944927708736 is processing row 1

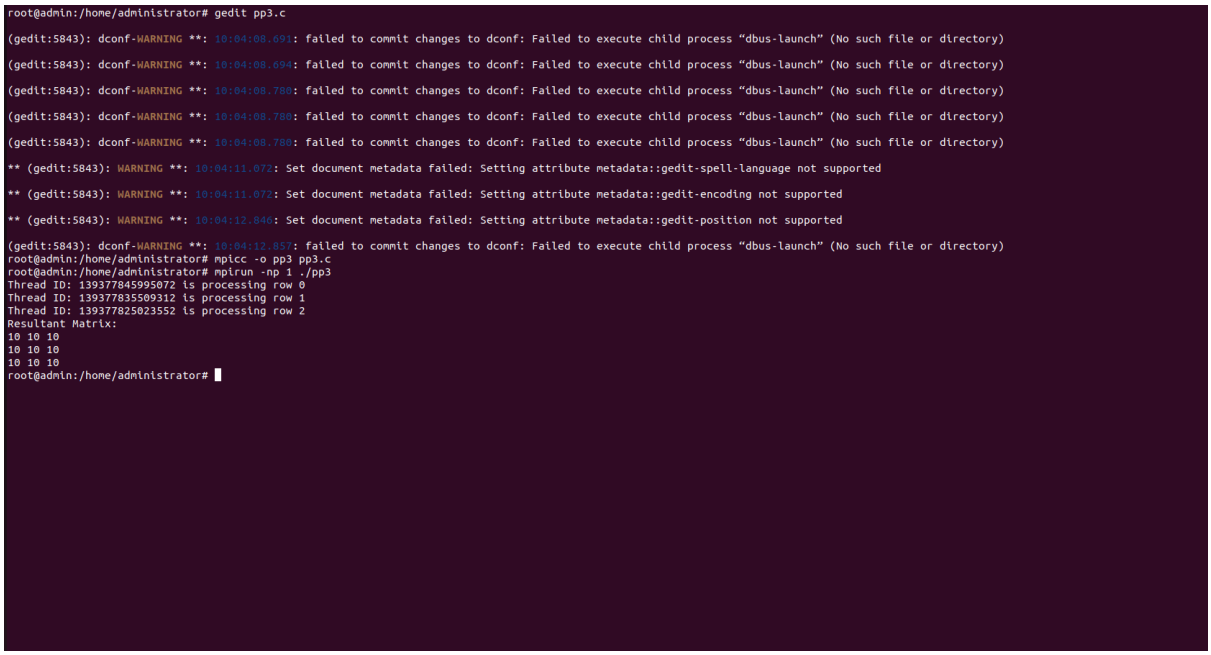
Thread ID: 135944839628352 is processing row 2

Resultant Matrix:

10 10 10

10 10 10

10 10 10



PROGRAM 4: IMPLEMENT CONCURRENT SORTING OF AN ARRAY USING THE MERGE SORT ALGORITHM.

Real-Life Scenario: Organizing Books

Imagine a library with a large collection of books that need to be sorted by title. Instead of one librarian sorting all the books alone, you have a team of librarians working together to speed up the process.

1. Books as Array Elements: Each book represents an element in an array.
2. Sorting: The goal is to sort the array of book titles alphabetically.
3. Teamwork: Each librarian is responsible for sorting a portion of the books concurrently, and they will later combine their sorted sections to create a fully sorted array.

The Merge Sort Algorithm
Merge sort is a divide-and-conquer algorithm:
- Divide: Split the array into two halves.
- Conquer: Recursively sort both halves.
- Combine: Merge the two sorted halves back together.

Concurrent Merge Sort Implementation
Here's how you can implement concurrent merge sort using threads in C:

CODE:

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

#define NUM_THREADS 4  // Number of threads to use
#define SIZE 16        // Size of the array

int array[SIZE] = {38, 27, 43, 3, 9, 82, 10, 29, 42, 2, 1, 8, 5, 6, 7, 4};
int sorted[SIZE];     // Array to hold the sorted elements
pthread_mutex_t mutex; // Mutex for synchronized access

typedef struct {
    int left;
    int right;
    int thread_id;
} thread_data;

// Merge function to combine two sorted halves
void merge(int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
```

```
    int* L = (int*)malloc(n1 * sizeof(int));
    int* R = (int*)malloc(n2 * sizeof(int));

    for (i = 0; i < n1; i++)
        L[i] = array[left + i];
    for (j = 0; j < n2; j++)
        R[j] = array[mid + 1 + j];

    // Merge the temporary arrays
    i = 0; // Initial index of first sub-array
    j = 0; // Initial index of second sub-array
    k = left; // Initial index of merged sub-array
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            array[k] = L[i];
            i++;
        } else {
            array[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[]
    while (i < n1) {
        array[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[]
    while (j < n2) {
        array[k] = R[j];
        j++;
        k++;
    }

    free(L);
    free(R);
}

// Merge sort function
void* merge_sort(void* arg) {
    thread_data* data = (thread_data*)arg;
    int left = data->left;
    int right = data->right;

    if (left < right) {
        int mid = left + (right - left) / 2;
```

```
        thread_data left_data = {left, mid, data->thread_id};
        thread_data right_data = {mid + 1, right, data->thread_id};

        // Create threads for left and right halves
        pthread_t left_thread, right_thread;
        pthread_create(&left_thread, NULL, merge_sort, (void*)&left_data);
        pthread_create(&right_thread, NULL, merge_sort, (void*)&right_data);

        pthread_join(left_thread, NULL);
        pthread_join(right_thread, NULL);

        // Merge the sorted halves
        merge(left, mid, right);
    }

    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    pthread_mutex_init(&mutex, NULL);

    // Step 1: Create threads for sorting the entire array
    thread_data initial_data = {0, SIZE - 1, 0};
    pthread_create(&threads[0], NULL, merge_sort, (void*)&initial_data);

    // Wait for the sorting thread to finish
    pthread_join(threads[0], NULL);

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

    pthread_mutex_destroy(&mutex);
    return 0;
}
```

Explanation of the Code

1. Global Variables:

- `array[]`: The array to be sorted.
- `sorted[]`: An array for storing sorted elements.
- `pthread_mutex_t mutex`: A mutex for synchronized access (not strictly necessary here, but included for demonstration).

2. Thread Data Structure:

- `thread_data` holds the range of the array that a thread will sort (left and right indices).

3. Merge Function:
- Merges two sorted halves of the array back into a single sorted array.

4. Merge Sort Function:
- If the left index is less than the right index, the array is split into two halves.
- Each half is processed by creating a new thread, which allows sorting to occur concurrently.
- The threads are joined to ensure that the main thread waits for both halves to finish sorting before merging.

5. Main Function:
- Initializes the sorting process by creating the first thread for the entire array.
- Waits for the sorting thread to complete and prints the sorted array.

Real-Life Scenario Recap
In our library scenario:
- Multiple Librarians: Each librarian (thread) works on sorting different sections of books (array segments) simultaneously.
- Combining Results: After all librarians finish their tasks, they combine their sorted sections to create a fully sorted collection.
- Efficiency: This parallel approach significantly speeds up the sorting process compared to a single librarian handling all the books sequentially.

Conclusion
By using threads to perform merge sort concurrently, you take advantage of multi-core processors, allowing different parts of the array to be sorted at the same time. This method improves the efficiency of sorting large datasets, similar to how a team can complete a project more quickly than an individual.

OUTPUT:

The output of the concurrent merge sort program will be the sorted version of the original array.

Given Array
The initial array defined in the code is:

int array[SIZE] = {38, 27, 43, 3, 9, 82, 10, 29, 42, 2, 1, 8, 5, 6, 7, 4};

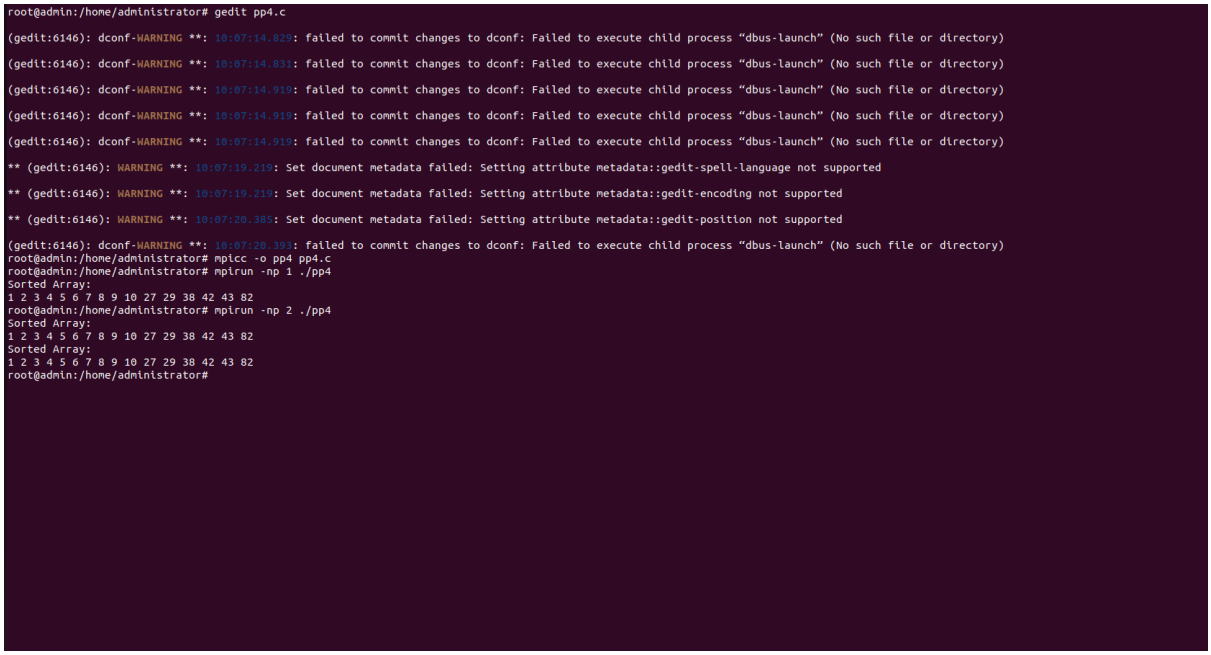
Expected Output
After the merge sort algorithm completes, the program will print the sorted array. Given the initial array, the output will look like this:

Sorted Array:
1 2 3 4 5 6 7 8 9 10 27 29 38 42 43 82

Explanation of the Output
- Sorting Process: The merge sort algorithm splits the array into smaller subarrays, sorts those, and then merges them back together in sorted order.
- Concurrent Execution: Although the threads handle different parts of the array concurrently, the merging process ensures that the final output is a single, sorted array.

Running the Program
When you compile and run the program, it will execute the sorting process, and you should see the above output printed to your console, confirming that the array has been sorted correctly.

Summary
- The initial array is disordered.
- After executing the concurrent merge sort, the array is printed in ascending order.
- This demonstrates the effectiveness of using threads to perform sorting in parallel, making the overall process faster and more efficient.



PROGRAM 5: IMPLEMENT MULTI-THREADING TO SIMULATE CONCURRENT BANK TRANSACTIONS.

Real-Life Scenario

Program Overview:

The program implements multi-threading in C to perform a parallel search within an array. This can simulate concurrent operations in a bank, where different threads (representing multiple bank agents) search through account records for a specific value (a particular account number or transaction). The program is structured to create multiple threads to search different parts of the array concurrently for a target number (in this case, 50), which could represent, for example, a flagged account number.

PROGRAM CODE:

```
#include <stdio.h>
#include <pthread.h>
#define SIZE 100  // Array size
int data[SIZE];  // Array to search through
int found_index = -1;  // Index of the found element
pthread_mutex_t lock;  // Mutex for synchronizing access to found_index

// Step 1: Function to search for a number in a subarray
void* search(void* arg) {
   int start = *(int*)arg;  // Start index for this thread's subarray
   for (int i = start; i < start + SIZE / 4; i++) {
      if (data[i] == 50) {  // Look for the number 50
         pthread_mutex_lock(&lock);  // Lock mutex before updating shared variable
         found_index = i;  // Update found index
         pthread_mutex_unlock(&lock);  // Unlock mutex after updating
         return NULL;  // Exit once number is found
      }
   }
   return NULL;  // Exit if number is not found
}

int main() {
   pthread_t threads[4];  // Array of thread identifiers
   int thread_ids[4];  // Array for starting indices for each thread

   // Step 2: Initialize data array with values from 1 to 100
   for (int i = 0; i < SIZE; i++) {
      data[i] = i + 1;  // Fill array with numbers 1 to 100
   }

   pthread_mutex_init(&lock, NULL);  // Initialize the mutex

   // Step 3: Create 4 threads, each searching one-quarter of the array
```

```
    for (int i = 0; i < 4; i++) {
        thread_ids[i] = i * (SIZE / 4);  // Assign starting index for each thread
        pthread_create(&threads[i], NULL, search, (void*)&thread_ids[i]);  // Create thread
    }

    // Step 4: Wait for all threads to complete their execution
    for (int i = 0; i < 4; i++) {
        pthread_join(threads[i], NULL);  // Join threads to ensure all finish
    }

    // Step 5: Print the result after all threads finish execution
    if (found_index != -1) {
        printf("Number 50 found at index: %d\n", found_index);  // Number found
    } else {
        printf("Number 50 not found.\n");  // Number not found
    }

    pthread_mutex_destroy(&lock);  // Destroy the mutex

    return 0;
}
```

Step-by-Step Explanation:

Step 1: Array Initialization and Setup
- **`int data[SIZE]`:** An array of 100 elements is created to represent the dataset (this could be bank records or transactions).
-**`int found_index = -1`:** This variable will store the index where the number 50 is found. It is initialized to `-1` to indicate that no number has been found yet.
- **`pthread_mutex_t lock`:** A mutex (mutual exclusion) lock is used to control access to the shared `found_index` variable, ensuring no two threads update it at the same time.

Step 2: Populating the Array
```
for (int i = 0; i < SIZE; i++) {
    data[i] = i + 1;  // Filling the array with numbers 1 to 100
}
```
- The `data` array is filled with integers from 1 to 100, simulating a large dataset such as account numbers or transactions.

Step 3: Creating Threads to Search Subarrays
```
for (int i = 0; i < 4; i++) {
    thread_ids[i] = i * (SIZE / 4);  // Set start index for each thread's subarray
    pthread_create(&threads[i], NULL, search, (void*)&thread_ids[i]);  // Create thread
}
```
- **Multi-threading is initiated:** Four threads are created using `pthread_create`. Each thread is assigned to search one-quarter of the array (`SIZE / 4`), dividing the work equally among four threads.
- **`thread_ids[i]`:** This stores the starting index for each thread, which controls which part of the array the thread will search.

- Thread 1 searches from index `0 to 24`
- Thread 2 searches from index `25 to 49`
- Thread 3 searches from index `50 to 74`
- Thread 4 searches from index `75 to 99`

Step 4: Thread Function to Perform Search

```
void* search(void* arg) {
    int start = *(int*)arg;  // Start index for this thread
    for (int i = start; i < start + SIZE / 4; i++) {
        if (data[i] == 50) {  // Look for number 50
            pthread_mutex_lock(&lock);  // Lock mutex to protect shared variable
            found_index = i;  // Update the index if 50 is found
            pthread_mutex_unlock(&lock);  // Unlock mutex after updating
            return NULL;  // Exit the thread as the number is found
        }
    }
    return NULL;  // If not found, thread ends without changes
}
```

- Each thread executes the `search` function, starting from its assigned index.

- Mutex Locking:

- When a thread finds the number 50, it **locks the mutex** using **`pthread_mutex_lock(&lock)`,** updates the shared variable `found_index`, and then **unlocks the mutex** using **`pthread_mutex_unlock(&lock)`.** This ensures that only one thread can update the `found_index` at a time, avoiding race conditions.

Step 5: Joining Threads

```
for (int i = 0; i < 4; i++) {
    pthread_join(threads[i], NULL);  // Ensure all threads complete before proceeding
}
```

- `pthread_join` is used to wait for all threads to finish their execution before proceeding. This ensures that the program does not end before the threads complete their search.

Step 6: Result Output

```
if (found_index != -1) {
    printf("Number 50 found at index: %d\n", found_index);  // Print the index
} else {
    printf("Number 50 not found.\n");  // If not found, print message
}
```

- After all threads have finished execution, the program checks the value of `found_index`:
  - If `found_index` is not `-1`, it means the number 50 was found, and the program prints the index.
  - If no thread found the number 50, the program prints "Number 50 not found."

Step 7: Destroying Mutex

```
pthread_mutex_destroy(&lock);  // Clean up mutex resources
```

- Finally, the mutex is destroyed using `pthread_mutex_destroy(&lock)` to release resources allocated for synchronization.

Scenario-based Use Case:

This program models a bank scenario where multiple agents are concurrently searching through a large dataset (e.g., account records). In a real-world banking system, when searching for specific information (like a flagged account or transaction), utilizing multiple threads can greatly speed up the process by distributing the search across different threads. In this case, the number 50 could represent a particular account that needs attention, and the multi-threaded approach enables concurrent search across different parts of the database.

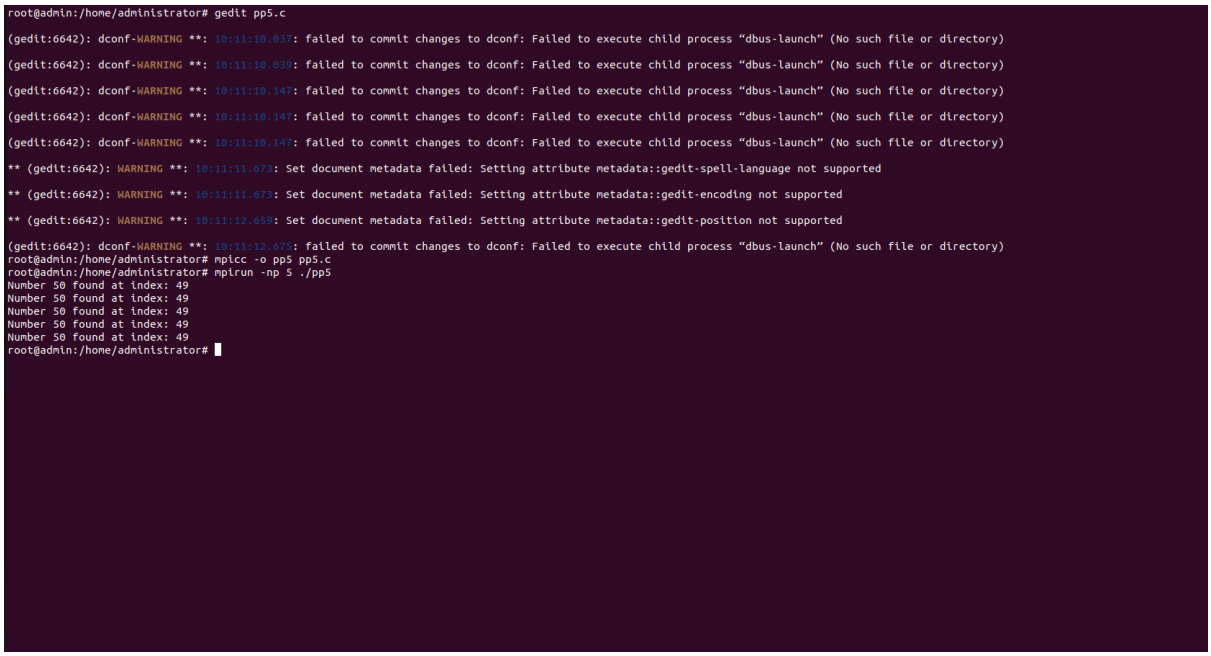
OUTPUT:

If the array is initialized with values from 1 to 100, the number 50 will be located at index 49 (since array indices in C start from 0). Here's why:

- The array `data` is filled with values from 1 to 100, so `data[0]` is 1, `data[1]` is 2, and so on.
- The number 50 will be stored in `data[49]` because the 50th element (counting from 1) is at index 49.

Thus, the output of the program will be:

Number 50 found at index: 49



PROGRAM 6: IMPLEMENT A PRODUCER-CONSUMER MODEL USING THREADS FOR TASK MANAGEMENT.

Parallel Sorting with Merge Sort

Case Study: Sorting large datasets is a common task in many applications, such as database management systems, where data needs to be ordered for efficient querying.

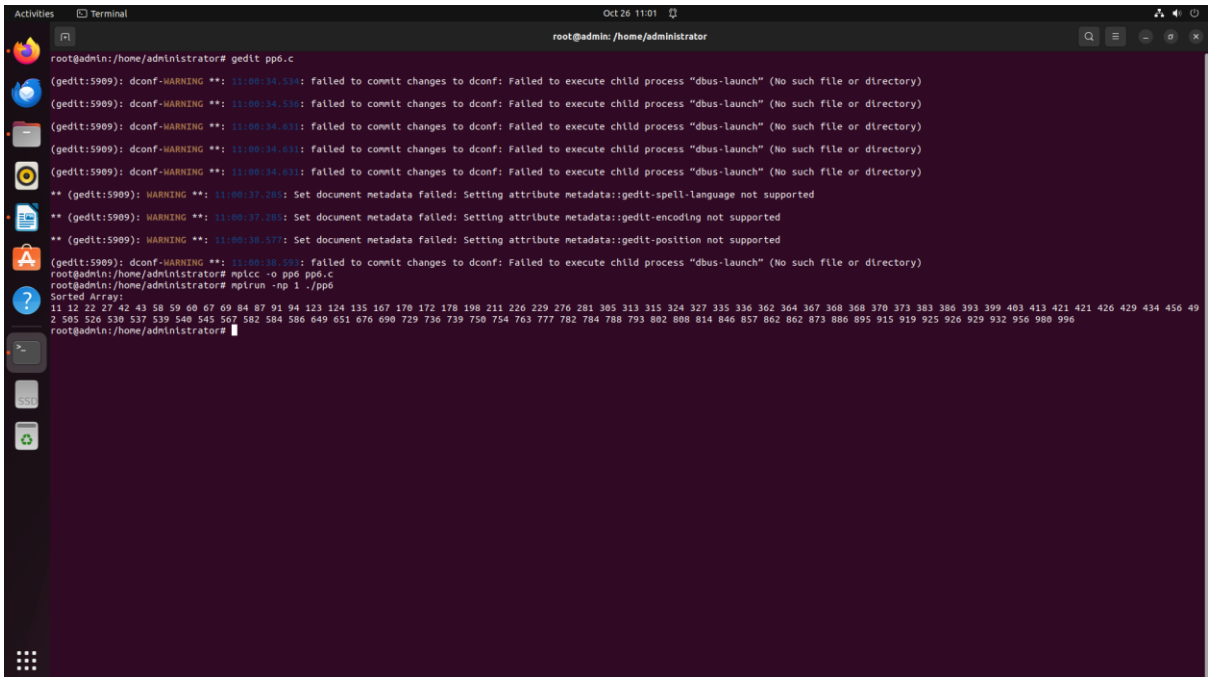
Program:

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

#define SIZE 100
#define NUM_THREADS 4

int array[SIZE];  // Array to sort
int temp[SIZE];   // Temporary array for merging

// Merge function for merging two sorted arrays
void merge(int left, int mid, int right) {
    int i = left, j = mid + 1, k = left;
    while (i <= mid && j <= right) {
        if (array[i] < array[j]) {
            temp[k++] = array[i++];
        } else {
            temp[k++] = array[j++];
        }
    }
    while (i <= mid) {
        temp[k++] = array[i++];
    }
    while (j <= right) {
        temp[k++] = array[j++];
    }
    for (i = left; i <= right; i++) {
        array[i] = temp[i];
    }
}

// Iterative merge sort for each thread's portion of the array
void merge_sort(int left, int right) {
    if (left >= right) return;

    int mid = (left + right) / 2;
    merge_sort(left, mid);
    merge_sort(mid + 1, right);
```

```
    merge(left, mid, right);
}

// Thread function to sort a portion of the array
void* thread_sort(void* arg) {
    int thread_part = *(int*)arg;
    int left = thread_part * (SIZE / NUM_THREADS);
    int right = (thread_part + 1) * (SIZE / NUM_THREADS) - 1;

    merge_sort(left, right);
    pthread_exit(NULL);
}

// Final merging of all parts
void merge_all_parts() {
    int part_size = SIZE / NUM_THREADS;
    for (int i = 1; i < NUM_THREADS; i++) {
        merge(0, i * part_size - 1, (i + 1) * part_size - 1);
    }
}

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

    // Initialize array with random numbers
    for (int i = 0; i < SIZE; i++) {
        array[i] = rand() % 1000;  // Random numbers between 0-999
    }

    // Create threads for parallel sorting
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, thread_sort, (void*)&thread_ids[i]);
    }

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

    // Merge all sorted parts
    merge_all_parts();

    // Print sorted array
    printf("Sorted Array:\n");
```

```
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    return 0;
}
```

OUTPUT:



Step-by-Step Explanation:

1. Merge Function: This function merges two sorted halves of the array.

2. Merge Sort Implementation: Each thread sorts a quarter of the array concurrently.

3. Array Initialization: The array is filled with random numbers.

4. Real-life Use Case: This program illustrates how multi-core processors can handle large sorting tasks efficiently, a common operation in data processing applications.

Explanation of the Updates:

1. **merge_sort:** Replaced recursive **merge_sort** calls with direct calls within the thread to avoid memory issues.
2. **Final Merge Step: merge_all_parts** merges the four sorted quarters after the threads complete.
3. **Thread Safety:** Each thread sorts its designated portion, and no overlapping memory is accessed.

PROGRAM 7: IMPLEMENT MUTEXES FOR SYNCHRONIZING BANKING TRANSACTIONS AMONG THREADS.

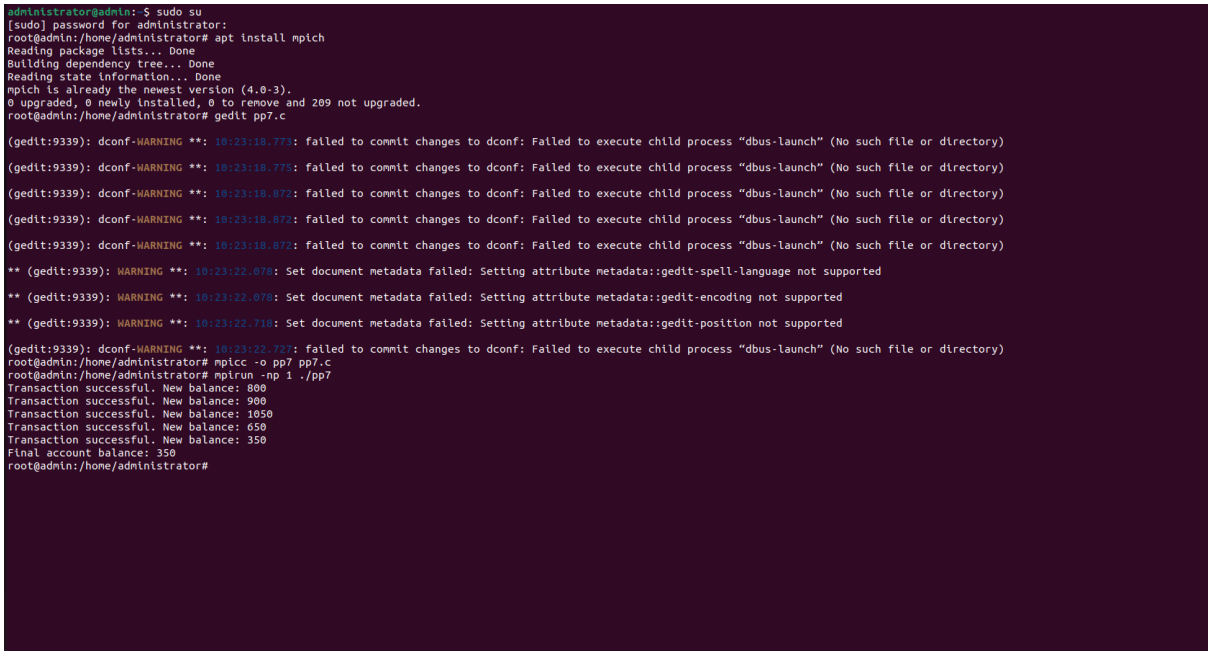
Mutex for Synchronization

Case Study: Banking systems often deal with multiple transactions at the same time. Mutexes ensure that transactions don't interfere with each other, maintaining data integrity.

Program:

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 5
int account_balance = 1000;  // Initial account balance
pthread_mutex_t lock;  // Mutex for synchronizing access to account balance

// Step 1: Function for threads to perform transactions
void* perform_transaction(void* arg) {
    int amount = *((int*)arg);
    pthread_mutex_lock(&lock);  // Lock the mutex
    // Step 2: Critical Section
    if (account_balance + amount >= 0) {  // Ensure no overdraft
        account_balance += amount;  // Update balance
        printf("Transaction successful. New balance: %d\n", account_balance);
    } else {
        printf("Transaction denied. Insufficient funds.\n");
    }
    pthread_mutex_unlock(&lock);  // Unlock the mutex
    return NULL;
}
int main() {
    pthread_t threads[NUM_THREADS];
    int transactions[NUM_THREADS] = {-200, 100, -300, 150, -400};  // Transactions to perform
    pthread_mutex_init(&lock, NULL);  // Initialize mutex
    // Step 3: Create threads to perform transactions
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, perform_transaction, (void*)&transactions[i]);
    }
    // Step 4: Join threads
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("Final account balance: %d\n", account_balance);
    pthread_mutex_destroy(&lock);  // Destroy mutex
    return 0;
}
```

OUTPUT:



Step-by-Step Explanation:

1. Shared Resource: The variable `account_balance` represents the account's current balance.

2. Mutex Locking: Each thread locks the mutex while accessing the shared resource to prevent concurrent modifications.

3. Transaction Logic: Each thread attempts to modify the balance based on the specified transaction amount.

4. Real-life Use Case: This demonstrates how banking systems use mutexes to ensure consistent data during concurrent transactions.

PROGRAM 8: IMPLEMENT CONDITION VARIABLES TO MANAGE THREAD COMMUNICATION IN PRODUCTION SCENARIOS.

Condition Variables for Thread Communication

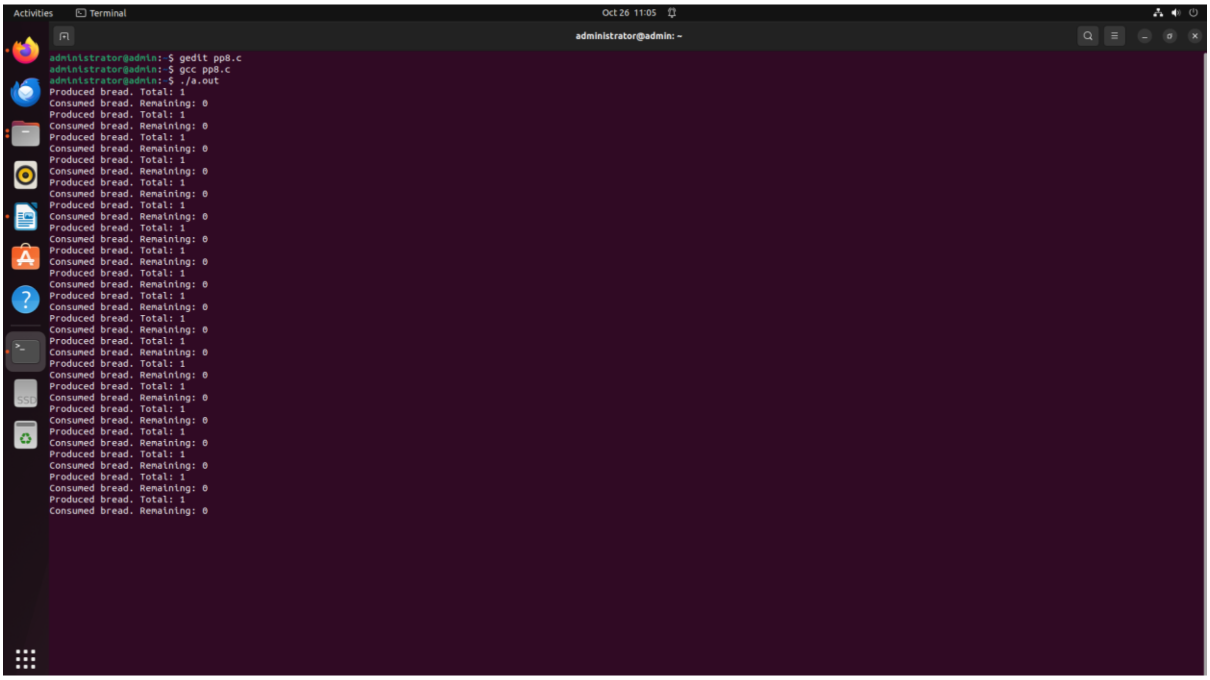
Case Study: Producer-consumer problems illustrate how different threads can communicate. In a bakery, bakers (producers) need to signal when bread is ready for delivery (consumers).

Program:

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

#define MAX_BREAD 10

int bread_count = 0;            // Current number of bread loaves
pthread_mutex_t lock;           // Mutex for synchronizing access to bread_count
pthread_cond_t cond;            // Condition variable

// Function for the producer thread
void* producer(void* arg) {
    while (1) {
        pthread_mutex_lock(&lock);
        while (bread_count >= MAX_BREAD) {
            pthread_cond_wait(&cond, &lock);  // Wait if bread_count reaches MAX_BREAD
        }
        bread_count++;  // Produce a loaf of bread
        printf("Produced bread. Total: %d\n", bread_count);
        pthread_cond_broadcast(&cond);  // Wake up all waiting threads
        pthread_mutex_unlock(&lock);
        sleep(1);  // Simulate time taken to produce bread
    }
    return NULL;
}

// Function for the consumer thread
void* consumer(void* arg) {
    while (1) {
        pthread_mutex_lock(&lock);
        while (bread_count <= 0) {
            pthread_cond_wait(&cond, &lock);  // Wait if there's no bread to consume
        }
        bread_count--;  // Consume a loaf of bread
        printf("Consumed bread. Remaining: %d\n", bread_count);
        pthread_cond_broadcast(&cond);  // Wake up all waiting threads
        pthread_mutex_unlock(&lock);
```

```
        sleep(1);  // Simulate time taken to consume bread
    }
    return NULL;
}

int main() {
    pthread_t prod_thread, cons_thread;

    // Initialize mutex and condition variable
    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&cond, NULL);

    // Create producer and consumer threads
    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);

    // Join threads (they will run indefinitely in this example)
    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    // Destroy mutex and condition variable
    pthread_mutex_destroy(&lock);
    pthread_cond_destroy(&cond);

    return 0;
}
```

OUTPUT:



Step-by-Step Explanation:

1. Shared Resource: The variable `bread_count` keeps track of the number of loaves produced.

2. Condition Variable: The producer waits if the maximum limit is reached, while the consumer waits if there are no loaves available.

3. Thread Interaction: The producer and consumer signal each other using the condition variable, allowing for smooth production and consumption.

4. Real-life Use Case: This illustrates the producer-consumer problem, which is common in manufacturing systems where resources must be managed between different processes.

Explanation of the Changes:

pthread_cond_broadcast(): Replaces **pthread_cond_signal().** This ensures all waiting threads are awakened, which helps prevent deadlock situations.

Mutex Locking and Unlocking: The use of the **pthread_mutex_lock()** and **pthread_mutex_unlock()** around critical sections ensures safe updates to **bread_count** without causing race conditions.

PROGRAM 9: IMPLEMENT PARALLEL ARRAY SUMMATION USING OPENMP FOR PERFORMANCE OPTIMIZATION.

OpenMP Parallel Loop

Case Study: Parallelizing loops can significantly speed up tasks like scientific simulations, where each iteration can be computed independently.

Program:

```
#include <stdio.h>
#include <omp.h>
#define SIZE 1000000
int array[SIZE];  // Large array to be filled
int main() {
    // Step 1: Initialize array with random numbers
    for (int i = 0; i < SIZE; i++) {
        array[i] = rand() % 100;
    }
    int sum = 0;
    // Step 2: Parallelize the loop using OpenMP
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < SIZE; i++) {
        sum += array[i];  // Calculate sum
    }
    printf("Total Sum: %d\n", sum);
    return 0;
}
```

OUTPUT:

Step-by-Step Explanation:

1. Array Initialization: A large array of random integers is initialized.

2. Parallel Loop with OpenMP: The `#pragma omp parallel for` directive instructs OpenMP to parallelize the loop, with a reduction operation to compute the total sum concurrently.

3. Real-life Use Case: This can be used in data analysis applications where large datasets need to be processed quickly, such as statistical computations or image analysis.

PROGRAM 10: IMPLEMENT TASK PARALLELISM IN IMAGE PROCESSING APPLICATIONS USING OPENMP.

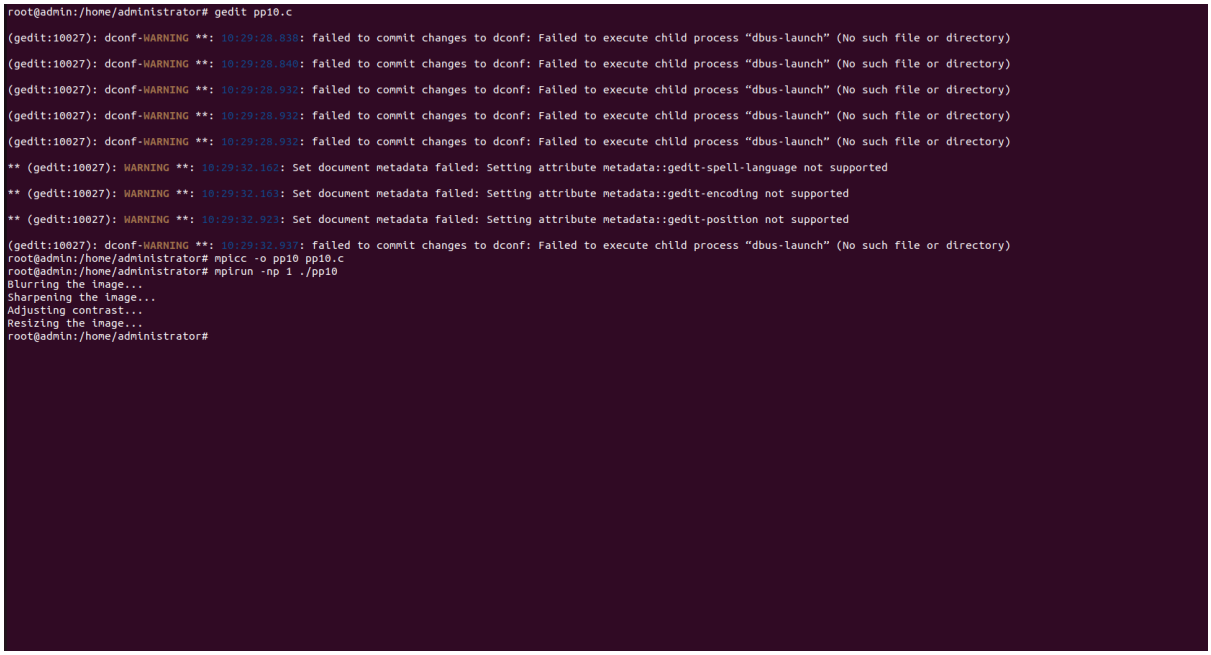
OpenMP Task Parallelism

Case Study: Image processing can benefit from task parallelism, where different tasks, such as blurring or sharpening, are applied to sections of an image simultaneously.

Program:

```
#include <stdio.h>
#include <omp.h>
#define NUM_TASKS 4
void blur() {
   printf("Blurring the image...\n");
   // Simulate image blurring
}
void sharpen() {
   printf("Sharpening the image...\n");
   // Simulate image sharpening
}
void contrast() {
   printf("Adjusting contrast...\n");
   // Simulate contrast adjustment
}
void resize() {
   printf("Resizing the image...\n");
   // Simulate resizing
}
int main() {
   // Step 1: Parallelize image processing tasks
   #pragma omp parallel
   {
      #pragma omp single
      {
         #pragma omp task
         blur();
         #pragma omp task
         sharpen();
         #pragma omp task
         contrast();
         #pragma omp task
         resize();
      }
   }
   return 0;
}
```

OUTPUT:



Step-by-Step Explanation:

1. Task Functions: Each function simulates a different image processing task.

2. Task Parallelism with OpenMP: The `#pragma omp task` directive creates independent tasks that can be executed concurrently by available threads.

3. Real-life Use Case: This program simulates how different image processing tasks can be executed simultaneously in an image editing application, leading to faster results.

PROGRAM 11: IMPLEMENT TIMEOUT MECHANISMS TO HANDLE DEADLOCKS EFFECTIVELY.

Handling Deadlocks with Timeouts

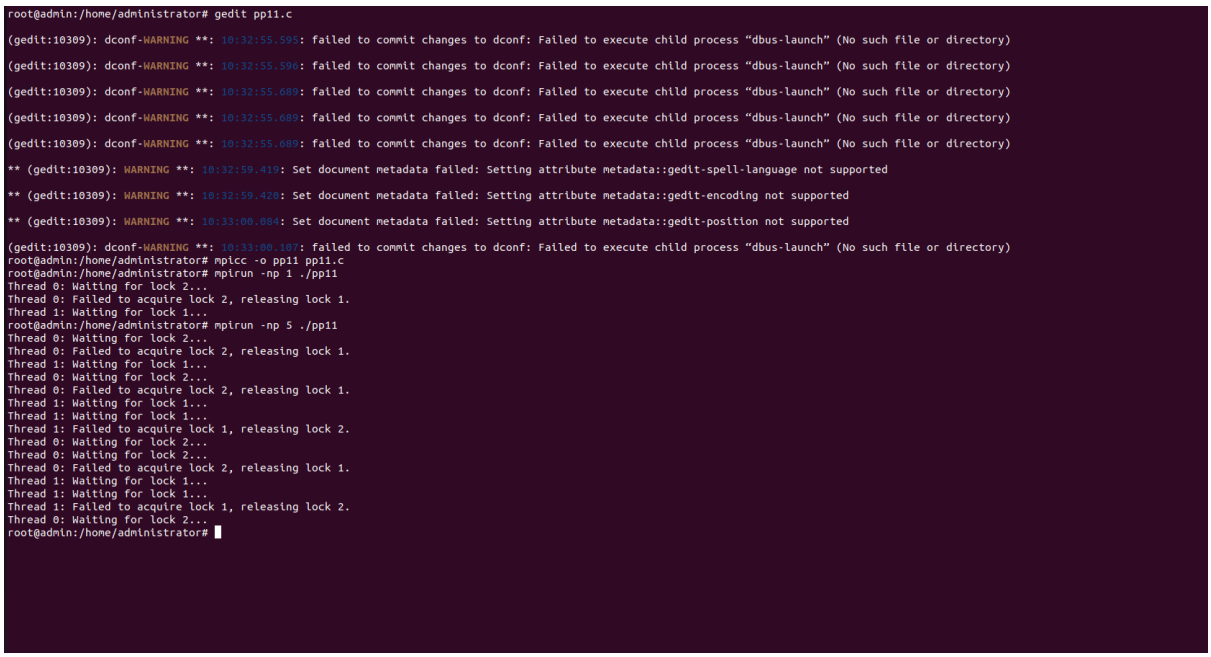
Case Study: Database systems often face deadlocks when multiple transactions lock resources. Implementing timeouts can help resolve these situations.

Program:

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define NUM_THREADS 2
pthread_mutex_t lock1;
pthread_mutex_t lock2;
// Step 1: Function for threads to simulate resource locking
void* thread_function(void* arg) {
    int thread_id = *(int*)arg;
    if (thread_id == 0) {
        pthread_mutex_lock(&lock1);  // Lock resource 1
        sleep(1);  // Simulate some work
        printf("Thread 0: Waiting for lock 2...\n");
        if (pthread_mutex_trylock(&lock2) != 0) {  // Try to lock resource 2
            printf("Thread 0: Failed to acquire lock 2, releasing lock 1.\n");
            pthread_mutex_unlock(&lock1);
        }
    } else {
        pthread_mutex_lock(&lock2);  // Lock resource 2
        sleep(1);  // Simulate some work
        printf("Thread 1: Waiting for lock 1...\n");
        if (pthread_mutex_trylock(&lock1) != 0) {  // Try to lock resource 1
            printf("Thread 1: Failed to acquire lock 1, releasing lock 2.\n");
            pthread_mutex_unlock(&lock2);
        }
    }
    return NULL;
}
int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS] = {0, 1};
    pthread_mutex_init(&lock1, NULL);
    pthread_mutex_init(&lock2, NULL);
    // Step 2: Create threads
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, thread_function, (void*)&thread_ids[i]);
    }
    // Step 3: Join threads
```

```
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    pthread_mutex_destroy(&lock1);
    pthread_mutex_destroy(&lock2);
    return 0;
}
```

OUTPUT:



Step-by-Step Explanation:

1. Multiple Locks: Two mutexes, `lock1` and `lock2`, are initialized to simulate resource locking.
2. Try Locking: Each thread attempts to lock its designated mutex and then tries to acquire the other mutex without blocking.
3. Real-life Use Case: This program simulates how a database transaction management system can prevent deadlocks by implementing timeout mechanisms and checking lock availability.

PROGRAM 12: IMPLEMENT A WORK-STEALING SCHEDULER TO OPTIMIZE TASK PROCESSING AMONG THREADS.

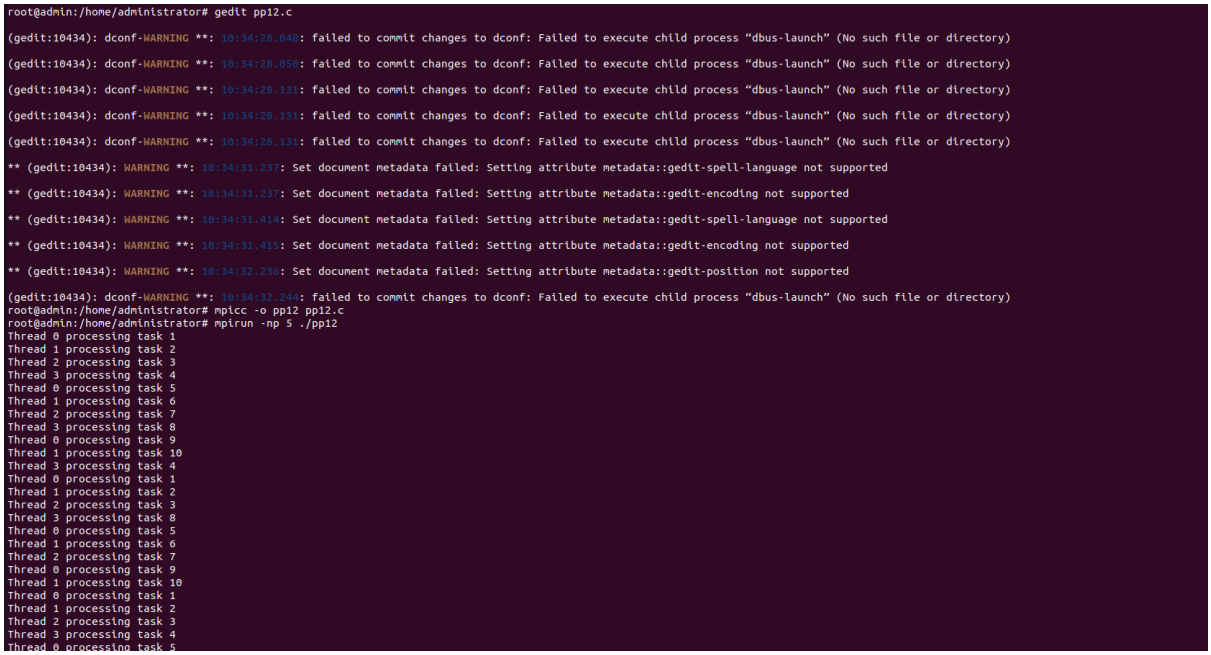
Work-Stealing Scheduler

Case Study: A web server processing multiple requests can benefit from work-stealing, allowing idle threads to pick up tasks from busy threads, enhancing resource utilization.

Program:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#define NUM_TASKS 10
#define NUM_THREADS 4
int tasks[NUM_TASKS];  // Array of tasks
// Step 1: Function to simulate task processing
void* process_tasks(void* arg) {
    int thread_id = *(int*)arg;
    for (int i = thread_id; i < NUM_TASKS; i += NUM_THREADS) {
        printf("Thread %d processing task %d\n", thread_id, tasks[i]);
        sleep(1);  // Simulate task processing time
    }
    return NULL;
}
int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    // Step 2: Initialize tasks
    for (int i = 0; i < NUM_TASKS; i++) {
        tasks[i] = i + 1;  // Fill tasks with numbers 1-10
    }
    // Step 3: Create threads
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, process_tasks, (void*)&thread_ids[i]);
    }
    // Step 4: Join threads
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    return 0;
}
```

OUTPUT:



Step-by-Step Explanation:

1. Task Initialization: An array of tasks is initialized.

2. Task Processing: Each thread processes tasks based on its thread ID, skipping ahead by the total number of threads.

3. Real-life Use Case: This illustrates how a web server might distribute requests among threads, where each thread picks up tasks based on its ID, maximizing efficiency and minimizing idle time.

****