MULTICORE ARCHITECTURE AND PROGRAMMING LABORATORY (CYL71)

List of Programs:

Program 1: Implement parallel processing to calculate the sum of an array.

Program 2: Implement concurrent sorting of an array using the merge sort algorithm.

Program 3: Implement multi-threading to simulate concurrent bank transactions.

Program 4: Implement a producer-consumer model using threads for task management.

Program 5: Implement mutexes for synchronizing banking transactions among threads.

Program 6: Implement condition variables to manage thread communication in production scenarios.

Program 7: Implement parallel array summation using OpenMP for performance optimization.

Program 8: Implement task parallelism in image processing applications using OpenMP.

Program 9: Implement timeout mechanisms to handle deadlocks effectively.

Program 10: Implement a work-stealing scheduler to optimize task processing among threads.

Program 1: Implement parallel processing to calculate the sum of an array.

Basic Multi-Core Matrix Addition

Case Study: Multi-core processors allow simultaneous operations on different data, making them suitable for tasks like image processing where multiple pixels can be processed independently.

Program:

```
#include <stdio.h>
#include <pthread.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
// Step 1: Function to add elements of matrices
void* add_matrices(void* arg) {
    int i = *(int*)arg;
    for (int j = 0; j < SIZE; j++) {
        C[i][j] = A[i][j] + B[i][j];  // Element-wise addition
    }
    return NULL;
}
int main() {
    pthread_t threads[SIZE];  // Thread array
    // Step 2: Create threads for each row
    for (int i = 0; i < SIZE; i++) {
        pthread_create(&threads[i], NULL, add_matrices, (void*)&i);
    }
    // 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;
}
```

Step-by-Step Explanation:

1. Matrix Initialization: Two 3x3 matrices, `A` and `B`, are initialized.

2. Thread Creation: A thread is created for each row of the matrix to perform the addition concurrently.

3. Element-wise Addition: Each thread computes the sum of corresponding elements in the two matrices and stores the result in matrix `C`.

4. Real-life Use Case: This method can be used in image processing where multiple pixels can be processed simultaneously, speeding up image transformations.

Program 2: Implement concurrent sorting of an array using the merge sort algorithm.

Simple Multi-Core Factorial Calculation

Case Study: Calculating factorials for large numbers can be parallelized to utilize multi-core processors effectively, significantly speeding up computations.

Program :

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4
long long results[NUM_THREADS]; // Array to store results from threads
// Step 1: Function to calculate factorial
void* factorial(void* arg) {
    int thread_id = *(int*)arg;
    long long fact = 1;
    for (int i = 1; i <= 5 + thread_id; i++) {
        fact *= i;  // Calculate factorial
    }
    results[thread_id] = fact;  // Store result
    return NULL;
}
int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];
    // Step 2: Create threads
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, factorial, (void*)&thread_ids[i]);
    }
    // Step 3: Join threads
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
        printf("Factorial computed by thread %d: %lld\n", i, results[i]);
    }
    return 0;
}
```

Step-by-Step Explanation:

1. Shared Array for Results: An array `results` stores the computed factorials from each thread.

2. Thread Execution: Each thread calculates the factorial of a small range of numbers and stores it in the results array.

3. Real-life Use Case: This demonstrates how multi-core processing can efficiently compute factorials, which is useful in algorithms involving combinations and permutations.

Program 3: Implement multi-threading to simulate concurrent bank transactions.

Parallel Search in an Array

Case Study: Searching large datasets, such as user records in a database, can be optimized using parallel search algorithms.

Program:

```c
#include <stdio.h>
#include <pthread.h>
#define SIZE 100
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;
    for (int i = start; i < start + SIZE / 4; i++) {
        if (data[i] == 50) {  // Looking for the number 50
            pthread_mutex_lock(&lock);
            found_index = i;  // Update found index
            pthread_mutex_unlock(&lock);
            return NULL;
        }
    }
    return NULL;
}
int main() {
    pthread_t threads[4];
    int thread_ids[4];
    // Step 2: Initialize data array with values
    for (int i = 0; i < SIZE; i++) {
        data[i] = i + 1;  // Fill with numbers 1-100
    }
    pthread_mutex_init(&lock, NULL);  // Initialize mutex
    // Step 3: Create threads for parallel search
    for (int i = 0; i < 4; i++) {
        thread_ids[i] = i * (SIZE / 4);
        pthread_create(&threads[i], NULL, search, (void*)&thread_ids[i]);
    }
    // Step 4: Join threads
    for (int i = 0; i < 4; i++) {
        pthread_join(threads[i], NULL);
    }
    // Step 5: Print the result
    if (found_index != -1) {
        printf("Number 50 found at index: %d\n", found_index);
    } else {
        printf("Number 50 not found.\n");
    }
    pthread_mutex_destroy(&lock);  // Destroy mutex
    return 0;
}
```

Step-by-Step Explanation:

1. Array Initialization: The `data` array is populated with integers from 1 to 100.

2. Thread Functionality: Each thread searches a quarter of the array for the number 50.

3. Mutex Locking: A mutex is used to protect access to the shared variable `found_index`.

4. Real-life Use Case: This program simulates how databases might search through large datasets, utilizing multiple threads to improve speed.

Program 4: 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
int array[SIZE];  // Array to sort
int temp[SIZE];  // Temporary array for merging
// Step 1: 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];  // Copy merged elements back
    }
}
// Step 2: Merge sort function
void* merge_sort(void* arg) {
    int left = *(int*)arg;
    int right = left + (SIZE / 4) - 1;  // Each thread sorts a quarter
    int mid;

    if (left < right) {
        mid = (left + right) / 2;  // Find the midpoint
        merge_sort((void*)&left);  // Sort left half
        merge_sort((void*)&mid);  // Sort right half
        merge(left, mid, right);  // Merge sorted halves
    }
    return NULL;
}
int main() {
    pthread_t threads[4];
    int thread_ids[4];
    // Step 3: Initialize array with random numbers
    for (int i = 0; i < SIZE; i++) {
        array[i] = rand() % 1000;  // Random numbers between 0-999
    }
    // Step 4: Create threads for parallel sorting
    for (int i = 0; i < 4; i++) {
        thread_ids[i] = i * (SIZE / 4);
        pthread_create(&
threads[i], NULL, merge_sort, (void*)&thread_ids[i]);
    }

    // Step 5: Join threads
    for (int i = 0; i < 4; i++) {
        pthread_join(threads[i], NULL);
    }
    // Step 6: Print sorted array
    printf("Sorted Array:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
    return 0;
}
```

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.

Program 5: 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;
}
```

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 6: 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
// Step 1: 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 for the consumer
        }
        bread_count++;  // Produce a loaf of bread
        printf("Produced bread. Total: %d\n", bread_count);
        pthread_cond_signal(&cond);  // Signal the consumer
        pthread_mutex_unlock(&lock);
        sleep(1);  // Simulate time taken to produce bread
    }
    return NULL;
}
// Step 2: 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 for the producer
        }
        bread_count--;  // Consume a loaf of bread
        printf("Consumed bread. Remaining: %d\n", bread_count);
        pthread_cond_signal(&cond);  // Signal the producer
        pthread_mutex_unlock(&lock);
        sleep(1);  // Simulate time taken to consume bread
    }
    return NULL;
}
int main() {
    pthread_t prod_thread, cons_thread;
    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&cond, NULL);
    // Step 3: Create producer and consumer threads
    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);
    // Step 4: Join threads (they will run indefinitely in this example)
    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);
    pthread_mutex_destroy(&lock);
    pthread_cond_destroy(&cond);
    return 0;
}
```

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.

Program 7: 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;
}
```

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 8: 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;
}
```

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 9: Implement timeout mechanisms to handle deadlocks effectively.

<u>**Handling Deadlocks with Timeouts**</u>

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;
}
```

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 10: 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:

```c
#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;
}
```

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.

Summary of Real-life Applications

- **Matrix Operations: Used in image processing.**
- **Factorial Calculation: Useful in algorithms for combinations and permutations.**
- **Parallel Search: Applicable in database queries.**
- **Sorting Algorithms: Essential in data processing tasks.**
- **Synchronization: Important in banking and transaction systems.**
- **Image Processing: Beneficial in media applications.**
- **Deadlock Management: Critical in database transaction handling.**
- **Work-Stealing: Improves performance in multi-threaded servers.**