CSCI3240 Lab 11: Thread-based Parallelism

Thread-based parallel programming is a technique in which multiple threads of execution are used to perform different tasks simultaneously. This can lead to significant performance improvements in applications that can be parallelized, as it allows for more efficient use of available resources.
C provides a standard library called **pthreads** (short for "POSIX threads") that can be used for thread-based parallel programming. This library provides functions for creating and managing threads, as well as synchronization primitives for coordinating access to shared resources.

Compilation Code:

```
$gcc -o program source.c -pthread
```

-pthread *flag in C compilation informs the compiler to add support for multi-threading with the POSIX threads library.*

1. **Creating Threads:**

The first step in thread-based parallel programming is to create threads. In C, this is done using the **pthread_create()** function, which takes four arguments:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void*), void *arg);
```

The ***thread*** argument is a pointer to a **pthread_t** variable that will hold the thread ID once the thread is created. The ***attr*** argument is a pointer to a **pthread_attr_t** structure that can be used to specify various attributes for the thread (such as its stack size). If you don't need to specify any attributes, you can pass NULL for this argument.

The ***start_routine*** argument is a pointer to the function that the new thread will execute. This function must take a single void* argument and return a void* value. Any arguments that you want to pass to the thread function must be passed via the ***arg*** argument.

The **pthread_join()** function for threads is the equivalent of wait() for processes. A call to **pthread_join** blocks the calling thread until the thread with identifier equal to the first argument terminates. The function prototype is:

```
int pthread_join(pthread_t thread, void **retval);
```

The ***thread*** argument implies the thread to be joined. And, ***retval*** argument is a pointer to a location where the exit status of the thread can be stored. This argument can be set to NULL if the exit status is not needed.

Here is an example of creating a new thread:

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

void *my_thread_function(void *arg)
{
    int my_arg = *(int*)arg;
    printf("Hello from thread %d\n", my_arg);
    return NULL;
}

int main()
{
    pthread_t my_thread;
    int arg = 42;
    pthread_create(&my_thread, NULL, my_thread_function, &arg);
    pthread_join(my_thread, NULL);
    return 0;
}
```

In this example, we define a thread function called my_thread_function() that takes a single integer argument and prints a message to the console. We then create a new thread and pass the address of the arg variable as the argument to the thread function. Finally, we call pthread_join() to wait for the thread to complete before exiting the program.

2. Synchronization primitives:

When multiple threads are accessing shared resources (such as memory or I/O devices), It is essential to synchronize thread access to shared resources to prevent data races, deadlocks, starvation, and other synchronization-related problems in concurrent programming. pthreads provides several synchronization primitives that can be used for this purpose.

3. Mutexes:

A mutex (short for "mutual exclusion") is a synchronization primitive that can be used to protect shared resources from concurrent access. A mutex has two states: locked and unlocked. When a thread wants to access a shared resource, it must first acquire the mutex by calling pthread_mutex_lock(). This function will block the thread if the mutex is currently locked by another thread. Once the mutex is acquired, the thread can access the shared resource. When the thread is done with the shared resource, it must release the mutex by calling pthread_mutex_unlock().

Here is an example of using a mutex to protect access to a shared counter variable:

```c
#include <pthread.h>
#include <stdio.h>
pthread_mutex_t mutex;
int counter = 0;

void *my_thread_function(void *arg)
{
    for (int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&mutex);
        counter++;
        pthread_mutex_unlock(&mutex);
    }
    return;
}

int main()
{
    pthread_t my_thread1, my_thread2;
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&my_thread1, NULL, my_thread_function, NULL);
    pthread_create(&my_thread2, NULL, my_thread_function, NULL);
    pthread_join(my_thread1, NULL);
    pthread_join(my_thread2, NULL);
    printf("Final value of counter: %d\n", counter);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

In this example, we define a shared counter variable and two threads that increment it by calling **pthread_mutex_lock()** to acquire the mutex before incrementing, and calling **pthread_mutex_unlock()** to release the mutex after incrementing. We also use **pthread_mutex_init()** to initialize the mutex before creating the threads, and **pthread_mutex_destroy()** to destroy the mutex after the threads have completed.

4. **Condition Variables:**

A condition variable is a synchronization primitive that can be used to signal one or more threads when a certain condition has been met. A condition variable is always used in conjunction with a mutex. When a thread wants to wait for a condition to become true, it first acquires the **mutex**, then calls **pthread_cond_wait()** on the condition variable. This function will atomically release the mutex and block the thread until another thread signals the condition variable by calling **pthread_cond_signal()** or **pthread_cond_broadcast().** When the waiting thread is woken up, it re-acquires the mutex and checks the condition again to make sure that it is true.
Here is an example of using a condition variable to implement a simple producer-consumer pattern:

```
#include <pthread.h>
#include <stdio.h>
#define BUFFER_SIZE 10
pthread_mutex_t mutex;
pthread_cond_t buffer_not_empty;
pthread_cond_t buffer_not_full;
int buffer[BUFFER_SIZE];
int head = 0, tail = 0;
void *producer_thread_function(void *arg)
{
    for (int i = 0; i < 50; i++) {
        pthread_mutex_lock(&mutex);
        while ((tail + 1) % BUFFER_SIZE == head) {
            pthread_cond_wait(&buffer_not_full, &mutex);
        }
        buffer[tail] = i;
        tail = (tail + 1) % BUFFER_SIZE;
        pthread_cond_signal(&buffer_not_empty);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

```
void *consumer_thread_function(void *arg)
{
    for (int i = 0; i < 50; i++) {
        pthread_mutex_lock(&mutex);
        while (head == tail) {
            pthread_cond_wait(&buffer_not_empty, &mutex);
        }
        int value = buffer[head];
        head = (head + 1) % BUFFER_SIZE;
        pthread_cond_signal(&buffer_not_full);
        pthread_mutex_unlock(&mutex);
        printf("Consumed %d\n", value);
    }
    return NULL;
}

int main()
{
    pthread_t producer_thread, consumer_thread;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&buffer_not_empty, NULL);
    pthread_cond_init(&buffer_not_full, NULL);
    pthread_create(&producer_thread, NULL, producer_thread_function, NULL);
    pthread_create(&consumer_thread, NULL, consumer_thread_function, NULL);
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&buffer_not_empty);
    pthread_cond_destroy(&buffer_not_full);
    return 0;
}
```

In this example, we define a circular buffer with a maximum size of **BUFFER_SIZE**. We also define two threads: a **producer** thread that generates values and adds them to the buffer, and a **consumer** thread that removes values from the buffer and prints them to the console.

The **producer** thread acquires the mutex, checks if the buffer is full, and if it is, waits on the **buffer_not_full** condition variable. If the buffer is not full, it adds the value to the buffer, updates the tail pointer, signals the **buffer_not_empty** condition variable to wake up any waiting consumer threads, and releases the mutex.

The consumer thread acquires the mutex, checks if the buffer is empty, and if it is, waits on the **buffer_not_empty** condition variable. If the buffer is not empty, it removes the value from the buffer, updates the head pointer, signals the **buffer_not_full** condition variable to wake up any waiting producer threads, and releases the mutex. It then prints the value to the console.

Note that we use a while loop around the **pthread_cond_wait()** call in both threads. This is because **pthread_cond_wait()** can wake up even if the condition variable has not been signaled due to a spurious wakeup. A while loop around the **pthread_cond_wait()** call ensures that the thread checks the condition again after being woken up to make sure that it is true.

Same example using semaphore (as discussed in class):

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

#define BUFFER_SIZE 10

int buffer[BUFFER_SIZE];
int buffer_index = 0;

sem_t empty;
sem_t full;
pthread_mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < 50; i++) {
        sem_wait(&empty); // decrement empty slots (P function)
        pthread_mutex_lock(&mutex);
        buffer[buffer_index++] = i;
        printf("Produced %d\n", i);
        pthread_mutex_unlock(&mutex);
        sem_post(&full); // increment full slots (V function)
    }
    pthread_exit(NULL);
}
```

```c
void *consumer(void *arg) {
    int i, item;
    for (i = 0; i < 50; i++) {
        sem_wait(&full); // decrement full slots (P function)
        pthread_mutex_lock(&mutex);
        item = buffer[--buffer_index];
        printf("Consumed %d\n", item);
        pthread_mutex_unlock(&mutex);
        sem_post(&empty); // increment empty slots (V function)
    }
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t producer_thread, consumer_thread;
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

Next, please attempt the Lab11_Thread-basedParallelism quiz in D2L.