Project 5

Thread Pool & Producer-Consumer Problem

Introduction

The thread pool is a design pattern that is used to manage a group of worker threads. The thread pool is used to reduce the overhead of creating and destroying threads. The user can submit tasks to the thread pool, and the thread pool will automatically assign the tasks to the worker threads.

The producer-consumer problem is a classic synchronization problem that involves two types of threads: producers and consumers. The producers produce items and put them into a shared buffer, while the consumers consume items from the shared buffer.

Implementation

Thread Pool

The thread pool is implemented with the following functions:

```
void  execute(void (*somefunction)(void *p), void *p);
int   pool_submit(void (*somefunction)(void *p), void *p);
void *worker(void *param);
void  pool_init(void);
void  pool_shutdown(void);
```

- The `pool_init` function initializes the thread pool. It creates a pool of worker threads and starts them, initializes the task queue, and initializes the mutex `queueLock` and the semaphores `full` and `empty`.
- The `pool_submit` function submits a task to the thread pool. It first locks the queue and checks if the queue is full. If the queue is full, it releases the mutex and returns. Otherwise, it adds the task to the queue, releases the mutex, and signals the semaphore `full`. The function returns 1 if the task is successfully submitted and 0 otherwise.
- The `worker` function is the worker thread function. It runs in an infinite loop, waiting for semaphore `full`. When the semaphore is signaled, it locks the queue, retrieves a task from the queue, calls `execute` function, releases the mutex, and executes the task.
- The `execute` function executes the task.
- The `pool_shutdown` function shuts down the thread pool. It destroys the mutex and semaphores and cancels all worker threads.

There is also a client function to test the thread pool. It tries to submit 30 tasks to the thread pool and waits for all tasks to complete. If a submission fails, it tries to submit the task again.

Producer-Consumer Problem

- The main function receives the sleep time, the number of producer threads, and the number of consumer threads as command-line arguments. It initializes the buffer, the buffer mutex, and the semaphores `full` and `empty`. It then creates the producer and consumer threads. After the sleep time, it cancels all threads and exits.
- The producer thread runs in an infinite loop. It sleeps for a random time and then produces an item. It waits for semaphore `empty`, locks the buffer, adds the item to the buffer, releases the buffer, signals the semaphore `full`, and repeats.
- The consumer thread runs in an infinite loop. It waits for semaphore `full`, locks the buffer, consumes an item from the buffer, releases the buffer, signals the semaphore `empty`, and repeats.

In the producer and consumer threads, the buffer is implemented as a circular queue with a fixed size. The buffer is an array of integers, and the buffer index `front` and `rear` are used to keep track of the front and rear of the queue. Whenever a production or consumption occurs, a message is printed to the console.

Correctness

Thread Pool

The correctness of the thread pool implementation is verified by submitting `adder` tasks to the thread pool. The `adder` task adds two integers and prints the result to the console. The client function submits 30 `adder` tasks to the thread pool and waits for all tasks to complete. The result is shown in the console:

```
ceryl@Ceryl:~/os_projects$ ./example
I add two values 3 and 6 result = 9
I add two values 13 and 15 result = 28
I add two values 6 and 12 result = 18
I add two values 9 and 1 result = 10
I add two values 2 and 7 result = 9
I add two values 10 and 19 result = 29
I add two values 3 and 6 result = 9
I add two values 0 and 6 result = 6
I add two values 12 and 16 result = 28
I add two values 11 and 8 result = 19
I add two values 7 and 9 result = 16
I add two values 2 and 10 result = 12
I add two values 2 and 3 result = 5
I add two values 7 and 15 result = 22
I add two values 9 and 2 result = 11
I add two values 2 and 18 result = 20
I add two values 9 and 7 result = 16
I add two values 13 and 16 result = 29
I add two values 11 and 2 result = 13
I add two values 17 and 15 result = 32
I add two values 9 and 13 result = 22
I add two values 1 and 19 result = 20
I add two values 4 and 17 result = 21
I add two values 18 and 4 result = 22
I add two values 15 and 10 result = 25
I add two values 13 and 6 result = 19
I add two values 11 and 0 result = 11
I add two values 16 and 13 result = 29
I add two values 16 and 1 result = 17
I add two values 2 and 10 result = 12
ceryl@Ceryl:~/os_projects$
```

Figure 1: Thread Pool Test Result

Producer-Consumer Problem

The correctness of the producer-consumer problem implementation is verified by running the program with different sleep time, producer thread count, and consumer thread count. The results are shown in the console:

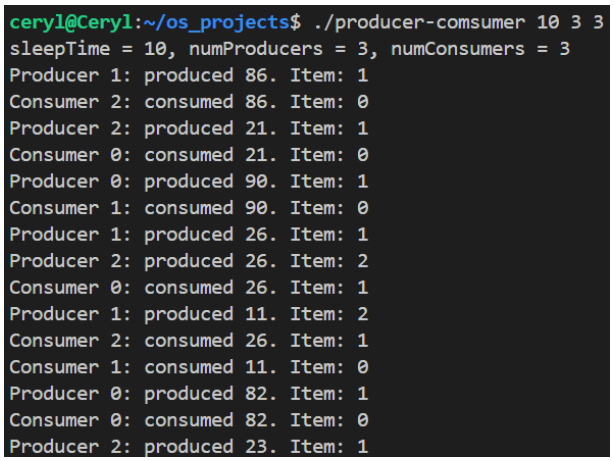

Figure 2: Producer-Consumer Problem Test Result (buffer size: 5)

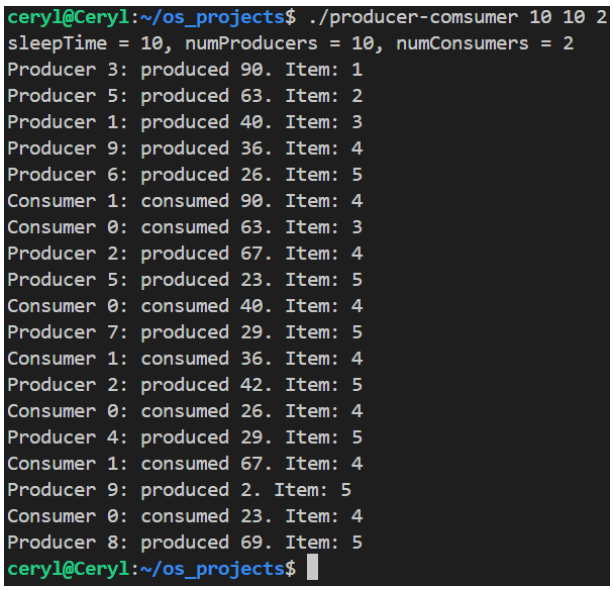

Figure 3: Producer-Consumer Problem Test Result (buffer size: 5)

Bonus

The setting of the core thread count in a thread pool has a significant impact on performance and resource utilization. If there are too many threads, the system may become overloaded, leading to context switching overhead and resource contention. If there are too few threads, the system may not be fully utilized, leading to underutilization of resources.

Therefore, it is important to set the core thread count based on the workload and system characteristics. The core thread count should be set to a value that maximizes performance and resource utilization while minimizing overhead and contention. For example:

- If the workload is I/O-bound, the core thread count can be set to match the speed of the I/O devices to avoid unnecessary waiting and context switching.

- If the workload is CPU-bound, the core thread count can be set to match the number of CPU cores to fully utilize the available processing power.

Conclusion

In this project, we implemented a thread pool and solved the producer-consumer problem using pthreads and semaphores. The two problems are classic synchronization problems that are commonly encountered in concurrent programming. The implementations demonstrate the use of pthreads, mutexes, and semaphores to synchronize threads and manage shared resources. The use of synchronization primitives is essential to ensure the correctness and efficiency of concurrent programs.