LAB 2 REPORT

Due 10/27/2021

Pablo Ruiz
pablo_ruiz@student.uml.edu

Objectives:

The purpose of the first part of the lab was to demonstrate the creation of a large number of threads and to synchronize said threads so that only one is accessing the critical shared data at any time. Part one of the assignment aims to create two provider threads and 260 buyer threads. The providers are responsible for adding unique integers to a buffer which represent objects which the buyers can buy by removing them from the buffer. After creating the providers, they should wait until they receive a signal from the main function to begin adding items to the buffer. On the other hand, after being created the buyers should wait until they receive a signal from the providers indicating that products are available in the buffer and one of them can buy a product and exit. Additionally, all threads should notify the user by printing to terminal any time they add or remove an item from the product buffer. Once all of the providers have bought a product and exited, the providers should exit as well. The main program which should have been waiting for all its child threads to join (exit) should also exit once they have all done so. In part two of the lab the number of provider threads will be reduced to one while the number of buyer threads will be reduce to a much smaller number like six. The purpose of part two is to optimize the efficiency of the program when dealing with a reduced number of threads.

Background:

Threads are treated equally to processes by the cpu however they have a few key differences. First, newly created threads unlike processes do not execute all the main programs code after their creation call, instead they are given a pointer to a function, and they will only execute code within that function. Additionally, threads do not obtain a copy of their parent's heap and stack instead they share their parent's heap but have their own independent stack. This means that any data structures that have to be shared between multiple threads must be declared in the parent's heap. Additionally, this poses a new problem, since all threads are potentially trying to access the same data unspecified behavior is likely to occur as the scheduler switches from one thread to the next while they are still running. In order to prevent this it is important that threads be synchronized when accessing critical shared data and to do this a binary flag managed by the OS called a mutex is used. When a mutex is locked only the thread that locked it can access the critical data while the other threads wait in a queue until the mutex is unlocked. A semaphore is a type of mutex which contains a non-negative integer counter that if set to 0 acts as a locked mutex, otherwise it acts as an unlocked mutex and any threads that want to access the critical data decrement it while they do. In effect a semaphore acts as a mutex which can let N threads access the critical data at once. A semaphore can also be used as a signaling tool to make sure multiple threads begin executing their code virtually at

the same time. Finally, a conditional variable is a tool that uses a mutex, conditional statement based on a variable, and a signal from a different thread to “unlock” a thread and allow it to execute its code. All threads waiting on a conditional variable will be placed in a queue and they will stay there until they are notified by another thread either one by one or all at once. Once they have been notified the thread will go ahead and check the conditional variable and if its state/value is correct the thread will be placed in the mutex queue.

Algorithms:

The algorithm used in part one is composed of 3 main functions: the main, the provider, and the buyer functions. The main function first uses the first argument passed during the execution call to determine how many buyer threads should be created. Then it initializes an array of thread objects which will be used to store the PIDs of every thread generated in this program, and two arrays one for the providers and one for the buyers which will store their rank (order in which they were created) so that it can be passed to each thread as a pointer. Then the function proceeds to initialize the semaphore that will be used to signal the providers to begin, to zero because it the provider threads should start executing at the same time, not as they are created. Subsequently, a loop is used to generate the required number of provider threads using the provider function and another loop is used to generate the required number of buyers and additionally lock the mutex every iteration to increase the counter that keeps track of active buyers at all times. In the next loop the value of the semaphore is incremented (sem_post()) once for every provider thread so that the providers can begin executing now that all the threads have been created and are ready to begin. Finally, the last for loop iterates over the array of PIDs and waits for them to exit and join back with the main program. Once they have all joined the main program exits as well.

The provider function receives a pointer to its rank (identification number, not PID) as an argument. It first waits (sem_wait()) for the value on the semaphore to be larger than 0 to continue and start executing its code. Once it continues, it first seeds the random function with the current time in seconds multiplied times its rank number, this will ensure that the random numbers it generates are different from the other providers since it is possible that they will all start executing within the same second. Subsequently, it enters an infinite loop in which it first attempts to lock the mutex for the buffer, then checks the number of active buyers and if its zero it breaks the loop, increments the semaphore so that other providers may continue (this is useless in this implementation since the initial semaphore value set by main would accommodate all the providers at once but could be useful in other implementations), and proceeds to exit. Otherwise, the loop continues, a randomly generated integer is pushed onto the buffer, the provider informs the user of the number added to the buffer by printing to terminal, and it notifies one of the buyers waiting for the conditional variable. Since the lock_guard wrapper was used to lock the mutex within the loop, once the loop moves on to the

next loop it loses scope and the mutex is unlocked allowing the buyer that was notified or another provider to access the buffer and active buyer counter.

The buyer function also receives a pointer to its rank (identification number, not PID) as an argument. Its first action is to wait until it is notified through the conditional variable. Once its notified, it uses the bufferNotEmpty() to check whether the buffer contains any products at the moment. If it does the buyer proceeds to put itself in the buffer mutex queue awaiting its turn to access the buffer. Otherwise, it returns to waiting for a notification. Once it is able to lock the mutex for the buffer the buyer removes an item from the buffer and proceeds to notify the user of which product it is buying by printing to terminal. Finally, it decrements the active buyer counter and exits at which time the mutex lock leaves scope and thus the mutex is unlocked.

The program for part 2 is very similar to the program from part 1 with a few key changes implemented in order to make it more efficient when dealing with only one provider and a smaller number of buyers. The biggest change was entirely removing the semaphore used to coordinate the providers. Since in this case there is only one provider it is not necessary to signal it to start by using a semaphore which is design to signal or lock a mutex for multiple threads at a time. Additionally, the for loop that creates the provider threads and the array that stores the provider ranks were also removed since with a single provider they were unnecessary and wasteful. Ultimately the order of the thread generation was altered so the buyers were generated before the provider, that way once the provider thread is created it can start right away without waiting. Finally, the seed for the random function within the provider function was altered to just depend on time instead of time and provider rank since there is only one provider and there is no risk of the random numbers overlapping.

Results:

Part 1 Input:

```
pruiz@DESKTOP-NHPDNJL:~/FALL 2021/Operating Systems/Assignment 2$ ./part1 260
```

Part 1 Output:

Beginning:

```
Provider 1 is adding product 888881199
Provider 1 is adding product 289249272
Provider 1 is adding product 834017909
Buyer 3 is buying product 888881199 and exiting
Buyer 2 is buying product 289249272 and exiting
Buyer 1 is buying product 834017909 and exiting
Provider 1 is adding product 2048282135
Provider 1 is adding product 1435953684
Buyer 5 is buying product 2048282135 and exiting
Provider 2 is adding product 1396223940
Provider 2 is adding product 453642559
Buyer 7 is buying product 1435953684 and exiting
Buyer 6 is buying product 1396223940 and exiting
Provider 2 is adding product 442122196
Provider 2 is adding product 80987779
```

End:

```
Buyer 260 is buying product 802093420 and exiting
Provider 2 is adding product 1383543517
Provider 2 is adding product 1414414461
Provider 2 is adding product 765254876
Provider 2 is adding product 2012175147
Provider 2 is adding product 568528995
Provider 2 is adding product 867585309
Buyer 257 is buying product 640357455 and exiting
Buyer 255 is buying product 2007860086 and exiting
Buyer 254 is buying product 1192831585 and exiting
All threads have finished running and main is exiting cleanly. Good bye!
```

Part 2 Input:

```
pruiz@DESKTOP-NHPDNJL:~/FALL 2021/Operating Systems/Assignment 2$ ./part2 6
```

Part 2 Output:

Beginning:

```
Provider is adding product 756075404
Provider is adding product 1754052712
Provider is adding product 1708975108
Buyer 2 is buying product 756075404 and exiting
Provider is adding product 1788276714
Provider is adding product 1211906431
Provider is adding product 417482865
Provider is adding product 1530355748
Buyer 3 is buying product 1754052712 and exiting
Buyer 1 is buying product 1708975108 and exiting
Buyer 5 is buying product 1788276714 and exiting
Buyer 6 is buying product 1211906431 and exiting
Provider is adding product 278196517
Provider is adding product 735359870
Provider is adding product 409581119
Provider is adding product 1863368091
Provider is adding product 1036640016
Provider is adding product 1809425358
```

End:

```
Provider is adding product 708446006
Provider is adding product 1556574797
Provider is adding product 284834746
Provider is adding product 1878060938
Provider is adding product 704234777
Provider is adding product 1724423678
Provider is adding product 1234441327
Provider is adding product 33867182
Provider is adding product 496711693
Provider is adding product 1334651940
Provider is adding product 1408840102
Buyer 4 is buying product 417482865 and exiting
All threads have finished running and main is exiting cleanly. Good bye!
```

Observations:

Even though conditional variables can be used on multiple threads simultaneously and all of the threads can be notified at once it is surprising that only one thread can access the critical data at once since the conditional variable wait() function still relies on a mutex. If the intent of a program is to signal multiple threads at once to proceed such as in the provider function outlined above it is a wiser option to use a semaphore since it offers simultaneous or near simultaneous activation. Even if said threads rely on a condition it would still be preferable to use a semaphore and then implement a conditional statement after the fact using a global variable.

Conclusions:

In conclusion, threads are a more resource efficient way to parallelize a program than processes. However, this efficiency also carries with it benefits and disadvantages. On one hand threads can more easily collaborate and interact with each other and their parent process since they all share the same heap memory space. On the other hand, this also opens up the possibility for undefined behavior to occur when multiple threads are accessing or referencing the same memory space at the same time while the scheduler is switching between them. Thus it is imperative that the programmer actively protect the shared "critical data" by utilizing OS tools such as mutexes, semaphores and conditional variables. By using these tools it is possible for threads to access the data in and orderly manner, sequentially and without overlapping.

Source Code:

Part 1:

```
/*****************************************************************************/
/*                          Programmer: Pablo Ruiz                           */
/*                                                                           */
/*                          Course: Operating Systems                        */
/*                                                                           */
/*                          Project: Assignment 2                            */
/*                                                                           */
/*****************************************************************************/

#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <condition_variable>
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
```

```
#include <time.h>

using namespace std ;

// Declare global variables
// this mutex protects both the buffer and the number of active Buyers
mutex bufferMutex ;
condition_variable cvBuffer ;
sem_t providerSemaphore ;
queue<int> productBuffer ;
int activeBuyers ;


// Check conditional variable
bool bufferNotEmpty() {return !productBuffer.empty() ;}

// Buyer Function
void *buyer(void *rank) {

    // Generate unique lock corresponding to the bufferMutex for each buyer
    unique_lock<mutex> lock(bufferMutex) ;

    // Wait to be notified and for the buffer not to be empty and lock the mutex
    cvBuffer.wait(lock, bufferNotEmpty) ;

    // Print buyer status and pop a product from the buffer
    cout << "Buyer " << *(int*)rank << " is buying product " <<
productBuffer.front() << " and exiting\n" ;
    productBuffer.pop() ;

    // Decrement the number of active buyers
    activeBuyers-- ;

    // Mutex is unlocked when it leaves the scope

}

// Provider function
void *provider(void *rank) {

    // Wait until the provider semaphore is larger than 0
    sem_wait(&providerSemaphore) ;
```

```
    // Seed rand function with the current time multiplied times the provider
rank to make sure they both start at around the same time but have different
seeds
    srand(time(NULL)* *(int*)rank) ;

    while (true) {

        // Lock the mutex until the end of each loop
        lock_guard<mutex> lock(bufferMutex) ;

        // If no buyers are left exit loop and return
        if (activeBuyers == 0){break ;}

        // Push number into buffer
        productBuffer.push(rand()) ;

        // Print the number pushed onto the buffer by the provider
        cout << "Provider " << *(int*)rank << " is adding product " <<
productBuffer.back() << "\n" ;

        // Notify one of the buyers that is waiting
        cvBuffer.notify_one();

    }

    // Increment semaphore before returning to allow any threads that are waiting
to accesss it
    sem_post(&providerSemaphore) ;

}

int main(int argc, char *argv[]) {

    int numProviders = 2,
        numBuyers = atoi(argv[1]),
        numThreads = numProviders + numBuyers,
        providerRank[numProviders],
        buyerRank[numBuyers],
        i ;

    // Initialize thread ID array to store PIDs of every thread
    pthread_t threadID[numThreads] ;

    // Initialize the semaphore to 0 to prevent the providers from starting until
signaled
```

```
    sem_init(&providerSemaphore, 0, 0) ;

    // Loop once for every provider rquired
    for (i = 0; i < numProviders; i++) {

        // Store the providers rank so a pointer to it can be passed to the
thread
        providerRank[i] = i + 1 ;

        // Create a provider thread and pass it its rank
        pthread_create(&threadID[i], NULL, provider, &providerRank[i]) ;

    }

    // Loop once for every buyer rquired
    for (i = 0; i < numBuyers; i++) {

        // Store the buyers rank to a pointer so it can be passed to the thread
        buyerRank[i] = i + 1 ;

        // Create buyer thread and pass it its rank
        pthread_create(&threadID[i + numProviders], NULL, buyer, &buyerRank[i]) ;

        // Lock mutex until the end of each loop to increment the number of
active buyers
        lock_guard<mutex> lock(bufferMutex) ;

        // Increment the activeBuyer count
        activeBuyers++ ;

    }

    // Increment (signal) the semaphore once for every provider thread
    for (i = 0; i < numProviders; i++)
        sem_post(&providerSemaphore) ;

    // Wait for all threads to exit
    for (i = 0; i < numThreads; i++)
        pthread_join(threadID[i], NULL) ;

    cout << "All threads have finished running and main is exiting cleanly. Good
bye!\n" ;

    return 0 ;
```

```
}
```

Part 2:

```
/*****************************************************************************/
/*                          Programmer: Pablo Ruiz                           */
/*                                                                           */
/*                          Course: Operating Systems                        */
/*                                                                           */
/*                          Project: Assignment 2                            */
/*                                                                           */
/*****************************************************************************/

// Removed semaphore, provider generation for loop, provider rank and its seed
rand effect.

#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <condition_variable>
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <time.h>

using namespace std ;

// Declare global variables
// this mutex protects both the buffer and the number of active Buyers
mutex bufferMutex ;
condition_variable cvBuffer ;
queue<int> productBuffer ;
int activeBuyers ;


// Check conditional variable
bool bufferNotEmpty() {return !productBuffer.empty() ;}

// Buyer Function
```

```
void *buyer(void *rank) {

    // Generate unique lock corresponding to the bufferMutex for each buyer
    unique_lock<mutex> lock(bufferMutex) ;

    // Wait to be notified and for the buffer not to be empty and lock the mutex
    cvBuffer.wait(lock, bufferNotEmpty) ;

    // Print buyer status and pop a product from the buffer
    cout << "Buyer " << *(int*)rank << " is buying product " <<
productBuffer.front() << " and exiting\n" ;
    productBuffer.pop() ;

    // Decrement the number of active buyers
    activeBuyers-- ;

    // Mutex is unlocked when it leaves the scope

}

// Provider function
void *provider(void *rank) {

    // Seed rand function with the current time multiplied times the provider
rank to make sure they both start at around the same time but have different
seeds
    srand(time(NULL)) ;

    while (true) {

        // Lock the mutex until the end of each loop
        lock_guard<mutex> lock(bufferMutex) ;

        // If no buyers are left exit loop and return
        if (activeBuyers == 0){break ;}

        // Push number into buffer
        productBuffer.push(rand()) ;

        // Print the number pushed onto the buffer by the provider
        cout << "Provider is adding product " << productBuffer.back() << "\n" ;

        // Notify one of the buyers that is waiting
        cvBuffer.notify_one();
```

```
    }

}

int main(int argc, char *argv[]) {

    int numProviders = 1,
        numBuyers = atoi(argv[1]),
        numThreads = numProviders + numBuyers,
        providerRank[numProviders],
        buyerRank[numBuyers],
        i ;

    // Initialize thread ID array to store PIDs of every thread
    pthread_t threadID[numThreads] ;

    // Loop once for every buyer required
    for (i = 0; i < numBuyers; i++) {

        // Store the buyers rank to a pointer so it can be passed to the thread
        buyerRank[i] = i + 1 ;

        // Create buyer thread and pass it its rank
        pthread_create(&threadID[i], NULL, buyer, &buyerRank[i]) ;

        // Lock mutex until the end of each loop to increment the number of
active buyers
        lock_guard<mutex> lock(bufferMutex) ;

        // Increment the activeBuyer count
        activeBuyers++ ;

    }

    // Create a provider thread
    pthread_create(&threadID[numThreads - 1], NULL, provider, NULL) ;

    // Wait for all threads to exit
    for (i = 0; i < numThreads; i++)
        pthread_join(threadID[i], NULL) ;

    cout << "All threads have finished running and main is exiting cleanly. Good
bye!\n" ;

    return 0 ;
```

```
}
```