LABWORK
COURSE: DISTRIBUTED SYSTEMS
CHAPTER 6: SYNCHRONIZATION

1. Deploy the synchronization for threads of a multithreaded process

1.1. Contents

We consider now the problem in the Distributed Systems where multiple processes try to access the shared resource, however this resource can support only one (or only some processes) at a moment. Another problem is that multiple processes may sometimes need to agree on the ordering of events, such as whether message m1 from process P was sent before or after message m2 from process Q. These problems can be solved in using synchronization mechanism in Distributed Systems.
In this labwork, you will learn a synchronization technique in Java. In order to facilitate the things, you will work with threads instead of processes.

1.2. Requirements

1.2.1. Theory

- synchronization

1.2.2. Hardwares

- Laptop/PC on any OS

1.2.3. Softwares

- any Java IDE (Eclipse is recommended)

1.3. PRACTICAL STEPS

You will try to simulate an environment with a shared resource and several threads that want to access this resource.

Create a class, named *ResourcesExploiter*, which has a private variable *rsc* (type *int*) that is considered as the shared resource. Since this is a private variable so you do need two methods to set and get the value for it:

```
public void setRsc(int n){
	rsc = n;
}

public int getRsc(){
	return rsc;
}
```

The constructor method of this class takes only one input parameter to initialize the *rsc* variable.

```
public ResourcesExploiter(int n){
        rsc = n;
}
```

You also create the *exploit()* method that increases the *rsc* variable by 1 unit.

```
public void exploit(){
        setRsc(getRsc()+1);
}
```

Create a class named *ThreadedWorkerWithoutSync* that extends the *Thread* class (available in the Java library).
Create a private variable named *rExp* of type *ResourcesExploiter*.
In the *run()* method that you must override, you put a loop *for* of 1000 times calling the *exploit()* method of the variable *rExp*.

Create a class for the executable program (with the *main* method).
In the *main* method, follow these steps:
- Create an instance named *resource* of type *ResourcesExploiter* with the initial value of the input parameter is 0.

```
ResourcesExploiter resource = new ResourcesExploiter(0);
```

- Create 3 instances named *worker1*, *worker2*, and *worker3* of type *ThreadedWorkerWithoutSync* (don't forget to put the instance *resource* as the input of the constructor method):

```
ThreadedWorkerWithoutSync worker1 = new ThreadedWorkerWithoutSync(resource);
ThreadedWorkerWithoutSync worker2 = new ThreadedWorkerWithoutSync(resource);
ThreadedWorkerWithoutSync worker3 = new ThreadedWorkerWithoutSync(resource);
```

- Start these three *worker1-3* threads (by calling the *start()* method).
- Don't forget to call the method *join()* for each thread to wait until this thread finishes its work.
- After, print the value of the *rsc* variable of the instance *resource*.

Question 1: Launch this program several times. What do you notice? Explain it!

Now, it's time to apply the synchronization mechanism to your program.
Create a class named *ThreadedWorkerWithSync* that is similar to the *ThreadedWorkerWithoutSync* class except that it now applies the ***synchronized*** on the *rExp* variable, and that's for the entire loop *for*.

```
synchronized(rExp){
  for(int i=0;i<1000;i++){
        rExp.exploit();
  }
```

Question 2: Change the code of the general executable program by replacing *ThreadedWorkerWithoutSync* with *ThreadedWorkerWithSync* to initiate three instances *worker1-3*. What is the difference between the output of this program and that of question 1? Explain it!

Now, you'll try the *lock* mechanism.
Create a class named *ResourcesExploiterWithLock* that extends the *ResourcesExploiter* class. First, you have to import two classes as follows:

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;
```

This class has a *lock* variable of type *ReentrantLock*:

```
private ReentrantLock lock;
```

For the constructor, you have to use *super* method to use the constructor of the class *ResourcesExploiter*:

```
    public ResourcesExploiterWithLock(int n){
        super(n);
        lock = new ReentrantLock();
    }
```

In its *exploit()* method, use the *lock.tryLock* method to block the variable's increase operation:

```
        try{
            if(lock.tryLock(10, TimeUnit.SECONDS)){
                setRsc(getRsc()+1);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }finally{
            //release lock
            lock.unlock();
        }
```

Create a class named *ThreadedWorkerWithLock* that extends the *Thread* class.
This class looks like the class *ThreadedWorkerWithoutSync* except that the type of its variable *rExp* is *ResourcesExploiterWithLock*.

Question 3: Change the code of the general executable program by replacing *ThreadedWorkerWithSync* with *ThreadedWorkerWithLock* to initiate three instances *worker1-3*. What is the difference between the output of this program and the output in question 1? Explain it!

2. Parallel Programming with Critical Sections

2.1. Contents

In the *Chapter 3: Processes and Threads*, we explored threads and their usefulness for concurrent programming. Threads are a way to divide the resources of a process so that individual parts can be scheduled independently. We also described this as user level parallelism, as oppose to O.S. level parallelism that is provide by processing.

There are many benefits to user level parallelism, such as simpler programming structure and design. User level parallelism also means that each thread shares the same resources, including memory. However, using shared resources comes at a cost. Imagine we have multiple threads trying to manipulate a single memory address. You might think that each thread will be able to act without interference, but computers are finicky and the thread scheduling routine is not transparent. A thread may be just in the middle of an operation and then be interrupted by another thread that is also operating on the same data. The result is that the data can become inconsistent and neither thread has the true representation of the data.

To solve these problems, we need a new mechanism to ensure that all critical operations are atomic or mutually exclusive, that is, an operation completes full within one thread without the possibility of another thread preempting that process. This mechanism is called Critical Section. In the 2nd part of this labwork, we are going to learn how to add mutual exclusion to our programs to avoid inconsistencies and how to avoid using these tools in ways to hamper program progress.

2.2. Requirements

2.2.1. Theory

- Parallel programming with critical sections

2.2.2. Hardwares

- Laptop/PC on any OS (recommended on Linux)

2.2.3. Softwares

- gcc

2.3. PRACTICAL STEPS

You begin with a very simple multi-threaded program.
Create a file *simple.c* with the content as below:

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

#include <unistd.h>
#include <sys/types.h>
```

```
#include <pthread.h>

int shared = 10;

void * fun(void * args){

  time_t start = time(NULL);
  time_t end = start+5; //run for 5 seconds

  YOUR-CODE-HERE

  return NULL;
}

int main(){
  pthread_t thread_id;

  pthread_create(&thread_id, NULL, fun, NULL);

  pthread_join(thread_id, NULL);

  printf("shared: %d\n", shared);

  return 0;
}
```

Question 4: Complete this file above (in the part **YOUR-CODE-HERE**) with a loop to increase the variable shared by 1 for 5 seconds.
(hint: `time(NULL)` will return the present system time in second).

Compile and run this program with the command:

```
$gcc -pthread simple.c -o simple
$./simple
```

The main thread will block on the function `pthread_join` and print the result of variable *shared*. This is only possible because both threads, the worker thread and the main thread, share memory.

Now, you will develop a program with more than 2 threads (multi-threaded program). So, the program has to manage the shared resource in using the *Locking* method. The concept of resource locking is a huge area of study in computer and operating system research. The big idea is that when a program enters a *critical section* or a set of code that must fully complete without interruption, the program holds a *lock* that only one program (or thread) can hold a time. In this way only one thread can ever be in the critical section at any time.

Now, first you develop a multi-threaded program without using *locking* method.

Create a file *without-lock.c* to simulate a simple banking service.

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

```
#include <unistd.h>
#include <sys/types.h>

#include <pthread.h>

#define INIT_BALANCE 50
#define NUM_TRANS 100

int balance = INIT_BALANCE;


int credits = 0;
int debits = 0;

void * transactions(void * args){
  int i,v;

  for(i=0;i<NUM_TRANS;i++){

    //choose a random value
    srand(time(NULL));
    v = rand() % NUM_TRANS;

    //randomnly choose to credit or debit
    if( rand()% 2){
      //credit

      balance = balance + v;
      credits = credits + v;

    }else{
      //debit

      balance = balance - v;
      debits = debits + v;

    }

  }

  return 0;
}

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

  int n_threads,i;
  pthread_t * threads;

  //error check
  if(argc < 2){
    fprintf(stderr, "ERROR: Require number of threads\n");
    exit(1);
  }

  //convert string to int
  n_threads = atol(argv[1]);

  //error check
  if(n_threads <= 0){
    fprintf(stderr, "ERROR: Invalivd value for number of threads\n");
    exit(1);
  }

  //allocate array of thread identifiers
  threads = calloc(n_threads, sizeof(pthread_t));

  //start all threads
  for(i=0;i<n_threads;i++){
    pthread_create(&threads[i], NULL, transactions, NULL);
  }

  //wait for all threads finish its jobs
  for(i=0;i<n_threads;i++){
    pthread_join(threads[i], NULL);
  }

  printf("\tCredits:\t%d\n", credits);
  printf("\t Debits:\t%d\n\n", debits);
  printf("%d+%d-%d=   \t%d\n", INIT_BALANCE,credits,debits,
         INIT_BALANCE+credits-debits);
  printf("\t Balance:\t%d\n", balance);

  //free array
  free(threads);

  return 0;
}
```

Now, build and run the above program. Try it with 5 additional threads

```
$gcc -pthread without-lock.c -o without-lock
$./without-lock 5
```

Question 5: Try to increase the value of threads and the value of the constant NUM_TRANS after each execution time until you obtain the different results between *Balance* and *INIT_BALANCE+credits-debits.* Explain why do you get this difference.

To solve the problem in Question 2, you have to identify the critical sections of the program, that is, sections of code that only one thread can execute at a time. Once a critical section is identified, we use a shared variables to *lock* that section. Only one thread can hold the *lock*, so only one thread executes the critical section at the time.

You first try a Naive Lock method in using a variable lock like the program below. You create a program called *naive-lock.c* with the content as follows:

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

#include <unistd.h>
#include <sys/types.h>

#include <pthread.h>

int lock = 0; //0 for unlocked, 1 for locked

int shared = 0; //shared variable
```

```
void * incrementer(void * args){
  int i;

  for(i=0;i<100;i++){

    //check lock
    while(lock > 0);  //spin until unlocked

    lock = 1; //set lock

    shared++; //increment

    lock = 0; //unlock
  }

  return NULL;
}

int main(int argc, char * argv[]){
  pthread_t * threads;
  int n,i;

  if(argc < 2){
    fprintf(stderr, "ERROR: Invalid number of threads\n");
    exit(1);
  }

  //convert argv[1] to a long
  if((n = atol(argv[1])) == 0){
    fprintf(stderr, "ERROR: Invalid number of threads\n");
    exit(1);
  }

  //allocate array of pthread_t identifiers
  threads = calloc(n,sizeof(pthread_t));

  //create n threads
  for(i=0;i<n;i++){
    pthread_create(&threads[i], NULL, incrementer, NULL);
  }

  //join all threads
  for(i=0;i<n;i++){
    pthread_join(threads[i], NULL);
  }

  //print shared value and result
  printf("Shared: %d\n",shared);
  printf("Expect: %d\n",n*100);

  return 0;
}
```

Question 6: Try to build and run this program. Launch it repeatedly until you see the difference between *Shared* and *Expect* values. Analyze the source code to understand the problem that leads to this difference.

Now, you'll try another way, called *mutex lock*, to realize the lock method. The term *mutex* stands for a mutually exclusion, which is a fancy name for lock. A *mutex* is not a standard variable; instead, it is guaranteed to be atomic in operation. The act of acquiring a lock cannot be interrupted.

The steps to deploy *mutex lock* are described as follows:
- First, you declare the variable mutex:
```
pthread_mutext_t mutex;
```

- After, you must first initialize a mutex before you can use it (the second parameter is always *NULL*):
```
pthread_mutex_init(&mutex, NULL);
```

- You then can acquire and unlock a mutex:
```
pthread_mutex_lock(&mutex);

/* critical section here */

pthread_mutex_unlock(&mutex);
```

- Finally, creating a *mutex* allocates memory. So we have to deallocate the *mutex*, or destroy it:
```
pthread_mutex_destroy(&mutex);
```

Question 7: Now, you have to modify the code of the file *without-lock.c* in implementing the *mutex lock* above (you can name it differently like *mutex-lock-banking.c*). Try to launch it repeatedly and evaluate the obtained output. What is the improvement after using *mutex lock*?

There are two strategies for locking critical sections: **Coarse Locking** and **Fine Locking**.
Coarse Locking locks a program in using a single lock for the critical section to protect the entire critical section. It's what you've done in the Question 4.
While Coarse Locking is a reasonable choice, it is inefficient. You look some parallelism because not all parts of the critical section are critical to each other. For example, consider the banking service program you are working on: the variable *credits* and *debits* are used exclusively of each other; each thread only performs a credit or debit but not both. Maybe it would be worthwhile to do more fine grain locking: it's called Fine Locking.

Now you'll modify the code of your banking program (it's better if you create a copy of that and name it differently like *fine-locking-bank.c*).
Instead of using only one variable mutex, you now declare three variables:

```
pthread_mutex_t b_lock,c_lock,d_lock;
```

where `b_lock` is for variable `balance`, `c_lock` for variable `credits`, and `d_lock` for variable `debits`.

In the loop *for*: `for(i=0;i<NUM_TRANS;i++),` you put each statement in a correspondent lock. For example:

```
        pthread_mutex_lock(&b_lock);
        balance = balance + v;
        pthread_mutex_unlock(&b_lock);
```

You do the same thing with other statements for *credits* and *debits*.

Question 8: compare the run times of the two strategies to prove that Fine Locking is faster and much faster on larger load sets.

Make attention while using Fine Locking method because there is a risk of having **deadlocks**. Try to run the program below (name it *deadlocks-test.c*):

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

int a=0,b=0;
pthread_mutex_t lock_a, lock_b;

void * fun_1(void * arg){
  int i;
  for (i = 0 ; i< 10000 ; i++){

    pthread_mutex_lock(&lock_a); //lock a then b
    pthread_mutex_lock(&lock_b);

    //CRITICAL SECTION
    a++;
    b++;


    pthread_mutex_unlock(&lock_a);
    pthread_mutex_unlock(&lock_b);

  }

  return NULL;
}

void * fun_2(void * arg){
  int i;
  for (i = 0 ; i< 10000 ; i++){

    pthread_mutex_lock(&lock_b); //lock b then a
    pthread_mutex_lock(&lock_a);

    //CRITICAL SECTION
    a++;
    b++;


    pthread_mutex_unlock(&lock_b);
    pthread_mutex_unlock(&lock_a);
  }

  return NULL;
}

int main(){

  pthread_t thread_1,thread_2;

  pthread_mutex_init(&lock_a, NULL);
  pthread_mutex_init(&lock_b, NULL);

  pthread_create(&thread_1, NULL, fun_1, NULL);
  pthread_create(&thread_2, NULL, fun_2, NULL);

  pthread_join(thread_1, NULL);
  pthread_join(thread_2, NULL);

  printf("\t a=%d b=%d \n", a,b);

  return 0;
}
```

Question 9: Run this program and what do you get as output? Explain what the *deadlock* is.