**Exercise 4b – Mutual Exclusion and Co-operation**

**Notes from Internet – Series-4**

**Thread Synchronization is achieved via Mutual exclusion and Co-operation**

**Mutual Exclusion:** When a process/thread is executing its critical section no other processes are allowed to execute its critical section. (Each process has a code segment called “Critical section” in which shared data is accessed.)

*Techniques used: Mutex Locks, Semaphores and Monitors*

**Co-operation:** When threads are trying to achieve a common goal via working together, these threads need cooperation between them. They need to synchronize when they focus on a common goal.

*Techniques used: Monitors*

**Thread Synchronization via Mutual Exclusion using Mutex – Sample Problems**

#include<stdio.h>

#include<string.h>

#include<pthread.h>

#include<stdlib.h>

#include<unistd.h>

pthread\_t tid[2];

int counter;

void\* doSomeThing(void \*arg)

{

unsigned long i = 0;

counter += 1;

printf("\n Job %d started\n", counter);

for(i=0; i<(0xFFFFFFFF);i++);

printf("\n Job %d finished\n", counter);

return NULL;

}

int main(void)

{

int i = 0;

int err;

while(i < 2)

{

err = pthread\_create(&(tid[i]), NULL, &doSomeThing, NULL);

if (err != 0)

printf("\ncan't create thread :[%s]", strerror(err));

i++;

}

pthread\_join(tid[0], NULL);

pthread\_join(tid[1], NULL);

return 0;

}

The above code is a simple one in which two threads(jobs) are created and in the start function of these threads, a counter is maintained through which user gets the logs about job number which is started and when it is completed. The code and the flow looks fine but when we see the output :

$ ./tgsthreads

Job 1 started

Job 2 started

Job 2 finished

Job 2 finished

If you focus on the last two logs, you will see that the log ‘Job 2 finished’ is repeated twice while no log for ‘Job 1 finished’ is seen.

**Logical flaw in the example program:**

* The log ‘Job 2 started’ is printed just after ‘Job 1 Started’ so it can easily be concluded that while thread 1 was processing the scheduler scheduled the thread 2.
* If the above assumption was true then the value of the ‘counter’ variable got incremented again before job 1 got finished.
* So, when Job 1 actually got finished, then the wrong value of counter produced the log ‘Job 2 finished’ followed by the ‘Job 2 finished’ for the actual job 2 or vice versa as it is dependent on scheduler.
* So we see that it’s not the repetitive log but the wrong value of the ‘counter’ variable that is the problem.
* The actual problem was the usage of the variable ‘counter’ by second thread when the first thread was using or about to use it. In other words, we can say that lack of synchronization between the threads while using the shared resource ‘counter’ caused the problems or in one word we can say that this problem happened due to ‘Synchronization problem’ between two threads.

**Mutexes**

Now since we have understood the base problem, lets discuss the solution to it. The most popular way of achieving thread synchronization is by using Mutexes.

A Mutex is a lock that we set before using a shared resource and release after using it. When the lock is set, no other thread can access the locked region of code. So, we see that even if thread 2 is scheduled while thread 1 was not done accessing the shared resource and the code is locked by thread 1 using mutexes then thread 2 cannot even access that region of code. So, this ensures a synchronized access of shared resources in the code.

Internally it works as follows:

* Suppose one thread has locked a region of code using mutex and is executing that piece of code.
* Now if scheduler decides to do a context switch, then all the other threads which are ready to execute the same region are unblocked.
* Only one of all the threads would make it to the execution but if this thread tries to execute the same region of code that is already locked then it will again go to sleep.
* Context switch will take place again and again but no thread would be able to execute the locked region of code until the mutex lock over it is released.
* Mutex lock will only be released by the thread who locked it.
* So, this ensures that once a thread has locked a piece of code then no other thread can execute the same region until it is unlocked by the thread who locked it.
* Hence, this system ensures synchronization among the threads while working on shared resources.

**A mutex is initialized and then a lock is achieved by calling the following two functions :**

* **int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, const pthread\_mutexattr\_t \*restrict attr);**
* **int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);**

The first function initializes a mutex and through second function any critical region in the code can be locked.

**The mutex can be unlocked and destroyed by calling following functions :**

* int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);
* int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

The first function above releases the lock and the second function destroys the lock so that it cannot be used anywhere in future.

**A Practical Example**

#include<stdio.h>

#include<string.h>

#include<pthread.h>

#include<stdlib.h>

#include<unistd.h>

pthread\_t tid[2];

int counter;

pthread\_mutex\_t lock;

void\* doSomeThing(void \*arg)

{

pthread\_mutex\_lock(&lock);

unsigned long i = 0;

counter += 1;

printf("\n Job %d started\n", counter);

for(i=0; i<(0xFFFFFFFF);i++);

printf("\n Job %d finished\n", counter);

pthread\_mutex\_unlock(&lock);

return NULL;

}

int main(void)

{

int i = 0;

int err;

if (pthread\_mutex\_init(&lock, NULL) != 0)

{

printf("\n mutex init failed\n");

return 1;

}

while(i < 2)

{

err = pthread\_create(&(tid[i]), NULL, &doSomeThing, NULL);

if (err != 0)

printf("\ncan't create thread :[%s]", strerror(err));

i++;

}

pthread\_join(tid[0], NULL);

pthread\_join(tid[1], NULL);

pthread\_mutex\_destroy(&lock);

return 0;

}

In the code above :

* A mutex is initialized in the beginning of the main function.
* The same mutex is locked in the ‘doSomeThing()’ function while using the shared resource ‘counter’
* At the end of the function ‘doSomeThing()’ the same mutex is unlocked.
* At the end of the main function when both the threads are done, the mutex is destroyed.
* Now if we look at the output, we find :

$ ./threads

Job 1 started

Job 1 finished

Job 2 started

Job 2 finished

So we see that this time the start and finish logs of both the jobs were present. So thread synchronization took place by the use of Mutex.

**Use the pthread\_mutex\_t Type and pthread\_mutex\_lock Function to Guard the Critical Section of the Code**

Threads share address spaces, which implies that modifications to the shared data like global variables must be synchronized; otherwise, there will be incorrect program behavior.

Note that the following code creates 4 additional threads with the pthread\_create call and passes func3 as a starting point of their execution. func3 modifies the global variable shared with one by one in a 10000 iteration for loop.

Thus, if the four threads increment the value of shared by 10000 each, the program should output 40000.

If you execute the following code, the result will be some random integer, but not 40000. This behavior is classified generally as a race condition, implying that given threads access the shared variable without consulting each other i.e. synchronization. Namely, often when their execution of the loop interleaves, the inconsistency is reached in accesses and stores of the shared variable, and finally, an incorrect sum is yielded.

The code section where multiple threads modify the same object in the memory is called a critical section. Generally, the critical section should be protected with some type of lock that would force other threads to wait until the current thread finishes the execution and ensures that they all get the correct incremented value. Mutex is one of the lock types that can be utilized to guard the critical section like this for loop in func3.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#ifndef NUM\_THREADS

#define NUM\_THREADS 4

#endif

int shared = 0;

void \*func3(void\* param)

{

printf("Incrementing the shared variable...\n");

for (int i = 0; i < 10000; ++i) {

shared += 1;

}

return 0;

}

int main()

{

pthread\_t threads[NUM\_THREADS];

for (int i = 0; i < NUM\_THREADS; ++i) {

pthread\_create(&threads[i], NULL, func3, NULL);

}

for (int i = 0; i < NUM\_THREADS; ++i) {

pthread\_join(threads[i], NULL);

}

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

exit(EXIT\_SUCCESS);

}

Output:

Incrementing the shared variable...

Incrementing the shared variable...

Incrementing the shared variable...

Incrementing the shared variable...

30384

In this case, we will utilize the POSIX threads library and its built-in pthread\_mutex\_t type. pthread\_mutex\_t type variable is usually declared as static storage duration.

**Mutex should be initialized only once before it’s used.**

When the mutex is declared as static, one should use the PTHREAD\_MUTEX\_INITIALIZER macro to initialize it.

Once the mutex is initialized, threads can use pthread\_mutex\_lock and pthread\_mutex\_unlock functions correspondingly. pthread\_mutex\_lock locks the mutex object passed as the only argument.

If the mutex was already locked, the calling thread gets blocked until the mutex becomes available. pthread\_mutex\_unlock should be called to unlock the mutex.

If there are threads waiting on the same mutex, the scheduling policy determines which one gets the object lock. Finally, we call pthread\_join on each of the four threads and print the integer - shared, which in this case should have the correct value stored.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#ifndef NUM\_THREADS

#define NUM\_THREADS 4

#endif

int shared = 0;

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

void \*func3(void\* param)

{

pthread\_mutex\_lock(&mutex);

printf("Incrementing the shared variable...\n");

for (int i = 0; i < 10000; ++i) {

shared += 1;

}

pthread\_mutex\_unlock(&mutex);

return 0;

}

int main()

{

pthread\_t threads[NUM\_THREADS];

for (int i = 0; i < NUM\_THREADS; ++i) {

pthread\_create(&threads[i], NULL, func3, NULL);

}

for (int i = 0; i < NUM\_THREADS; ++i) {

pthread\_join(threads[i], NULL);

}

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

exit(EXIT\_SUCCESS);

}

Output:

Incrementing the shared variable...

Incrementing the shared variable...

Incrementing the shared variable...

Incrementing the shared variable...

40000

**Thread Synchronization via Mutual Exclusion through Semaphores**

Semaphore is a data handling technique which is very useful in process synchronization and multithreading. In this article, we will explore how we can use semaphore in C language.

We have the POSIX semaphore library in Linux systems. We will use this in our case.

The basic code of a semaphore cannot be typed directly into client code as it should be atomic and writing code directly would lead to a context switch which would lead to unexpected results.The POSIX system in Linux presents its own built-in semaphore library. To use it, we have to :

* Include semaphore.h header file
* Compile the code by linking with -lpthread -lrt
* To lock a semaphore, we can use the sem\_wait function:

**int sem\_wait(sem\_t \*sem);**

* To release or signal a semaphore, we use the sem\_post function:

**int sem\_post(sem\_t \*sem);**

* A semaphore is initialised by using sem\_init (for processes or threads) or sem\_open (for Interprocess communication).

**sem\_init(sem\_t \*sem, int pshared, unsigned int value);**

1. sem : Specifies the semaphore to be initialized.
2. pshared : This argument specifies whether or not the newly initialized semaphore is shared between processes or between threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.
3. value : Specifies the value to assign to the newly initialized semaphore.

* To destroy a semaphore, we can use sem\_destroy.

**sem\_destoy(sem\_t \*mutex);**

* To declare a semaphore, the data type is **sem\_t.**

#include < stdio.h>

#include < pthread.h>

#include < semaphore.h>

#include < unistd.h>

sem\_t mutex;

void\* thread(void\* arg)

{

//wait

sem\_wait(&mutex);

printf("\nEntered thread\n");

//critical section

sleep(4);

//signal

printf("\n Exit thread\n");

sem\_post(&mutex);

}

int main()

{

sem\_init(&mutex, 0, 1);

pthread\_t t1,t2;

pthread\_create(&t1,NULL,thread,NULL);

sleep(2);

pthread\_create(&t2,NULL,thread,NULL);

pthread\_join(t1,NULL);

pthread\_join(t2,NULL);

sem\_destroy(&mutex);

return 0;

}

Compilation should be done with gcc a.c -lpthread -lrt

Explanation of above code

* 2 threads are being created, one 2 seconds after the first one.
* The first thread will sleep for 4 seconds after acquiring the lock.
* Thus the second thread will not enter immediately after it is called, it will enter 4 – 2 = 2 secs after it is called.

So the output is:

Entered thread

Exit thread

Entered thread

Exit thread

If we would not have used semaphore, the output would have been as follows due to context switching:

Entered thread

Entered thread

Exit thread

Exit thread

## **Thread Synchronization using Monitors - Mutual exclusion and Cooperation**

## **What Is a Monitor?**

A monitor is a synchronization mechanism that allows threads to have:

* mutual exclusion – only one thread can execute the method at a certain point in time, using locks
* cooperation – the ability to make threads wait for certain conditions to be met, using wait-set

Why is this feature called “monitor”? Because**it monitors how threads access some resources.**

Monitors formally became subject of interest in the early ’70s in the paper written by P.B. Hansen named [Shared Classes](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.112.6400&rep=rep1&type=pdf). After that, C.A.R. Hoare wrote the paper [Monitors – an Operating System Structuring Concept](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.3720&rep=rep1&type=pdf) where he developed the concept of Monitors even further. The paper introduced a form of synchronization and proposed a model of implementation using semaphores.

## **Monitor Features**

Monitors provide three main features to the concurrent programming:

* only one thread at a time has mutually exclusive access to a critical code section
* threads running in a monitor could be blocked while they’re waiting for certain conditions to be met
* one thread can notify other threads when conditions they’re waiting on are met

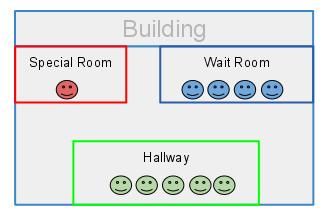
### **Building and Exclusive Room Analogy**

In literature, authors use a building and exclusive room analogy to represent the monitor mechanism. In this analogy, only one person can be present in an exclusive room at a time.

So, in this analogy:

* the monitor is a building that contains two rooms and a hallway
* the synchronized resource is the “special room”
* wait set is a “waiting room”
* entry set is a “hallway”
* threads are people who want to get to the exclusive room

A monitor can be considered as a building which contains a special room. The special room can be occupied by only one customer(thread) at a time. The room usually contains some data and code.



When the person wants to enter the special room, he first goes to the hallway (the entry set) where he waits for a scheduler. Therefore, the scheduler will pick the person and send him to the special room.

Hence, when the scheduler picks the person [FIFO}, he enters the exclusive room. It could be that some specific situation is happening in this room, so that person needs to go out and wait for the special room to become available again. Therefore, that person will end up in the waiting room (the wait set). Consequently, the scheduler will schedule this person to enter an special room later.

Also, it’s important to mention the steps that threads go through during this process, using the same analogy:

* entering the building – entering the monitor
* entering the special room – acquiring the monitor
* being in the special room – owning the monitor
* leaving the special room – releasing the monitor
* leaving the building – exiting the monitor.

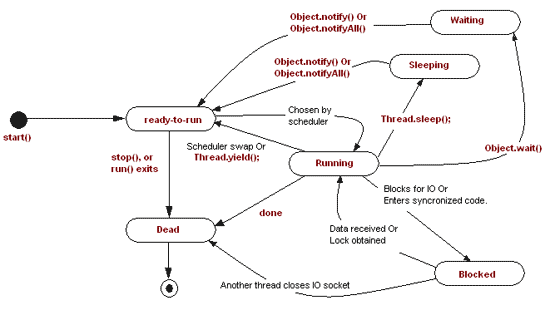
### **wait() and notify()**

[wait() and notify()](https://www.baeldung.com/java-wait-notify) are key methods in Java used in synchronized blocks that enable collaboration between threads.

**wait() orders the calling thread to release the monitor and go to sleep until some other thread enters this monitor and calls notify()**.

Also, notify() wakes up the first thread that called wait() on the specific object.

**How is it implemented in Java?**



* In the Java virtual machine, every object and class is logically associated with a monitor. To implement the mutual exclusion capability of monitors, a lock (sometimes called a mutex) is associated with each object and class. This is called a semaphore in operating systems books, mutex is a binary semaphore.
* If one thread owns a lock on some data, then no others can obtain that lock until the thread that owns the lock releases it. It would be not convenient if we need to write a semaphore all the time when we do multi-threading programming. Luckily, we don't need to since JVM does that for us automatically.
* To claim a monitor region which means data not accessible by more than one thread, Java provide synchronized statements and synchronized methods. Once the code is embedded with synchronized keyword, it is a monitor region. The locks are implemented in the background automatically by JVM.

**In Java synchronization code, which part is monitor?**

* We know that each object/class is associated with a Monitor. I think it is good to say that each object has a monitor, since each object could have its own critical section, and capable of monitoring the thread sequence.
* To enable collaboration of different threads, Java provide wait() and notify() to suspend a thread and to wake up another thread that are waiting on the object respectively. In addition, there are 3 other versions:

|  |
| --- |
| wait(**long** timeout, **int** nanos)  wait(**long** timeout) notified by other threads or notified by timeout.  notify(all) |

* Those methods can only be invoked within a synchronized statement or synchronized method. The reason is that if a method does not require mutual exclusion, there is no need to monitor or collaborate between threads, every thread can access that method freely.

Refer

[Java Threads (villanova.edu)](http://www.csc.villanova.edu/~mdamian/threads/javathreads.html)

<http://www.csc.villanova.edu/~mdamian/threads/javamonitors.html>

<http://web.mit.edu/javadev/doc/tutorial/java/threads/monitors.html>

<https://www.bestprog.net/en/2021/02/06/java-interaction-between-threads/>

[Java notify() and wait() examples (programcreek.com)](https://www.programcreek.com/2009/02/notify-and-wait-example/)