PTHREAD MUTEX

Mutex (mutual exclusion)
When a thread need exclusive access sections of code, it can use mutex.
Mutex can prevent other thread from executing a sections of code.

Lets start with an example.

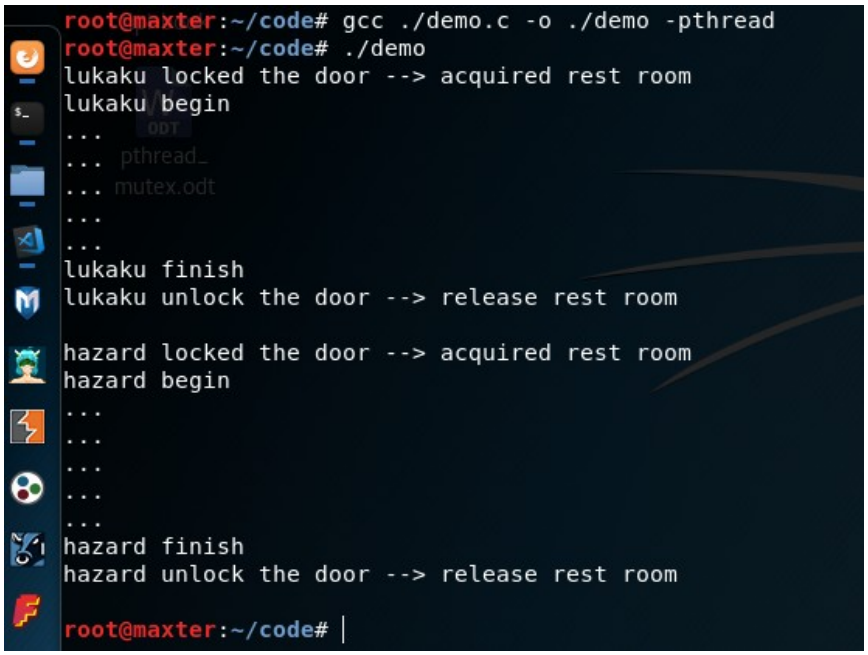
Lukaku and Hazard need to enter a rest room. However, the rest room capability is only one man at a time.	Thread_1 and Thread_2 need to execute a section of code. However, this code is designed to be executed by only one thread at a time.
Lukaku is acquiring rest room. The door is locked. One minutes later. Hazard need to enter the rest room. However, the door is locked. Hazard must wait. Lukaku finish his job. Lukaku open the door and get out. Now, Hazard can enter the rest room.	Thread_1 is executing the code. The mutex is locked. One minutes later. Thread_2 need to execute the code, too. However, mutex is locked, Thread_2 must wait Thread_1 finish execution. Thread_1 unlock the mutex Now, Thread_2 can execute the code.

Code

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

static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

void* enter_rest_room(void* arg){
    pthread_mutex_lock(&mtx); //lock the door

    printf("%s locked the door --> acquired rest room\n", (char*)arg);

    printf("%s begin\n", (char*)arg);

    //use the rest room in five minutes
    for(int i = 0; i < 5; i++) {
        printf("...\n");
        sleep(1);
    }

    printf("%s finish\n", (char*)arg);

    pthread_mutex_unlock(&mtx); //unlock the door

    printf("%s unlock the door --> release rest room\n\n", (char*)arg);
}

int main() {
    pthread_t lukaku;
    pthread_t hazard;

    //lukaku enter the rest room
    pthread_create(&lukaku, NULL, enter_rest_room, "lukaku");

    sleep(1); //one minute later

    //harard need to enter the rest room
    pthread_create(&hazard, NULL, enter_rest_room, "hazard");
```

```
    sleep(11);
    return 0;
}
```

Result



In above example, the mutex works like a door of the rest room.
If the door is locked, later person must wait.
Until the door is unlocked, he can acquire the rest room.

If the mutex is locked, later thead which need to execute the exclusive code must wait.
Util the mutex is unlocked, it can acquire the mutex and execute the code.

Mutex Features

Atomicity

Two thread can not lock the same mutex at the same time.

Singularity

If a thread acquire the mutex, no other thread will able to lock the mutex.

Non-Busy Wait

If a thread is waiting for a mutex. It will be suspended and not consume any CPU resouces.

Initialize Mutex
A mutex variable is represented by the *pthread_mutex_t* data type.
We must first initiallize it, before using it.

```
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t attr)
```

The mutex argument identifies the mutex to be initialized.
The attr argumen define attributes for the mutex. If attr is NULL, the mutex will be initialized with default attributes.

Attempting to initalize an already initalized mutex will lead to undefined behavior.

In case mutex is statically allocated, we should use macro *PTHREAD_MUTEX_INITIALIZER* to initialize it.
Among below case, we can use *pthread_mutex_init()* to initialize
- The mutex is dynamically allocated on the heap
- The mutex is an automatically variable allocated on the stack
- We want to custome attributes of static mutex.

Destroy Mutex
When a mutex is no long required, it should be destroyed by using *pthread_mutex_destroy()*

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

It will be safe to destroy an initialized mutex that is locked.
Atempting to destroy a locked mutex will lead to undefined behavior.

It is not neccessary to call *pthread_mutex_destroy* on static mutex. Static variable will be destroyed automatically when the process is terminated.

Lock Mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

If the mutex is unlocked, the calling thread will acquire the mutex.
If the mutex is locked by another thread, the calling thread will block until the mutex becomes avaible.
If a thread attempt to lock a mutex that it already owns, deadlock will occur.

Unlock Mutex

```
int pthead_mutex_unlock(pthread_mutex_t *mutex)
```

The *pthead_mutex_unlock()* function unlock the mutex previously locked by the calling thread.
If other threads are currently waiting the mutex, one of these acquire the muex and resume its execution.

Attempting to unlock a mutex that is not currenly locked, deadlock or undefined behavior error will occur.
Attempting to unlock a mutex that is locked by another thread, deadlock or undefined behavior error will occur.

Sample Code
Initialize static mutex

```
#include <pthread.h>

static pthread_mutex_t = PTHREAD_MUTEX_INITIALIZER; //initialize static mutex
```

Inititalize dynamic mutex

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

```

```
void foo(){
    pthread_mutex_t mutex;
    pthread_mutex_init(&mutex, NULL);//initialize dynamic mutex

    pthread_mutex_destroy(&mutex);//destroy dynamic mutex
}

void bar(){
    pthread_mutex_t* p_mutex =
(pthread_mutex_t*)malloc(sizeof(pthread_mutex_t));
    pthread_mutex_init(p_mutex, NULL);//initialize dynamic mutex

    pthread_mutex_destroy(p_mutex);//destroy dynamic mutex

    free(p_mutex);
}

int main() {
    foo();
    bar();

    return 0;
}
```

Notes:
What happen if attempting to relock a mutex that it has already locked?
It depend on mutex type.

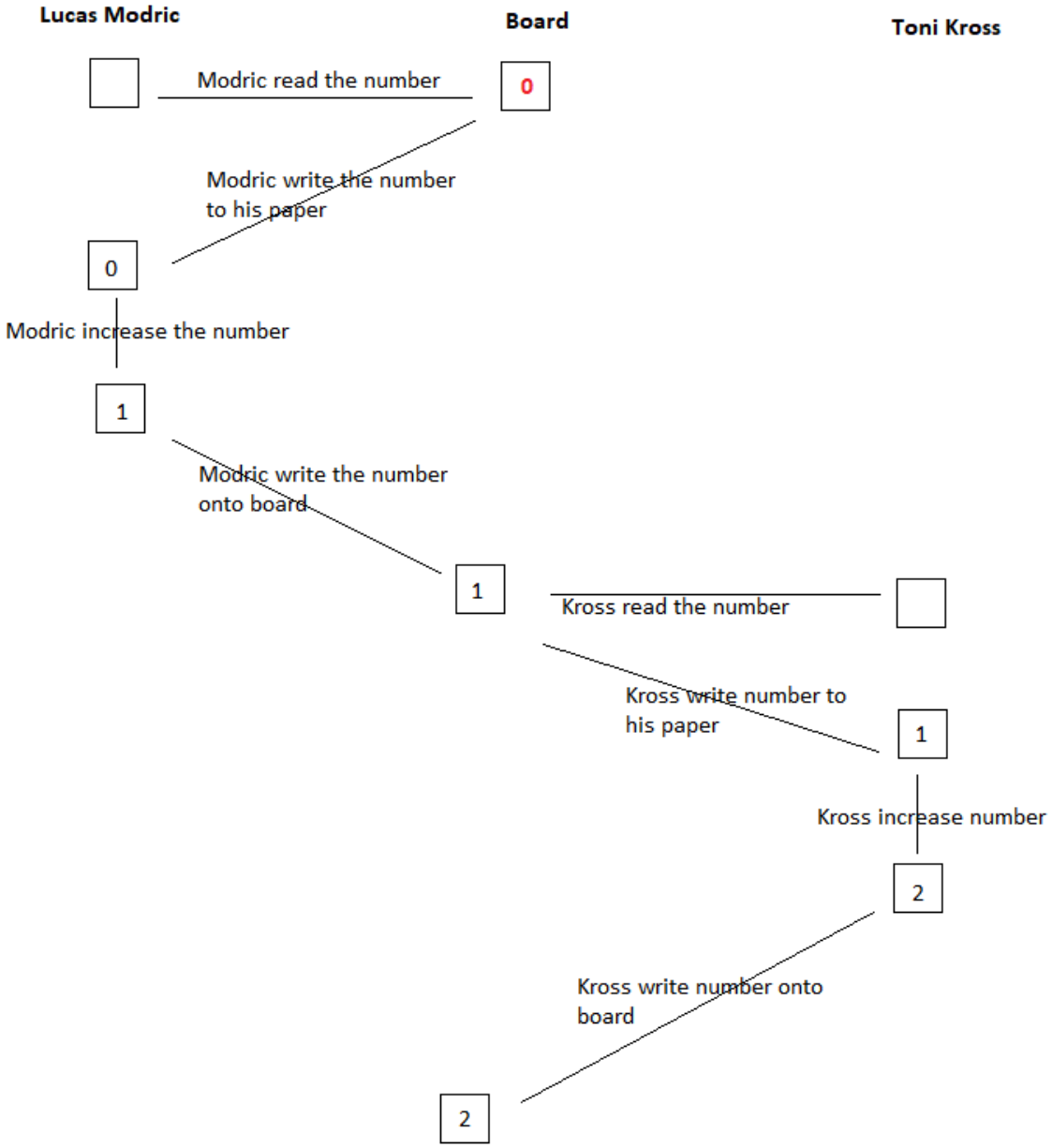
Mutex Type	Result
PTHREAD_MUTEX_NORMAL	Deadlock
PTHREAD_MUTEX_ERRORCHECK	Return an error code
PTHREAD_MUTEX_RECURSIVE	Lock count is incremented by one. If we lock a recursive mutex twice, we need to unlock it twice to release.
PTHREAD_MUTEX_DEFAULT	Undefined behavior

What happen if attempting to unlock a mutex that it was unlocked?
It depend on mutex type, too.

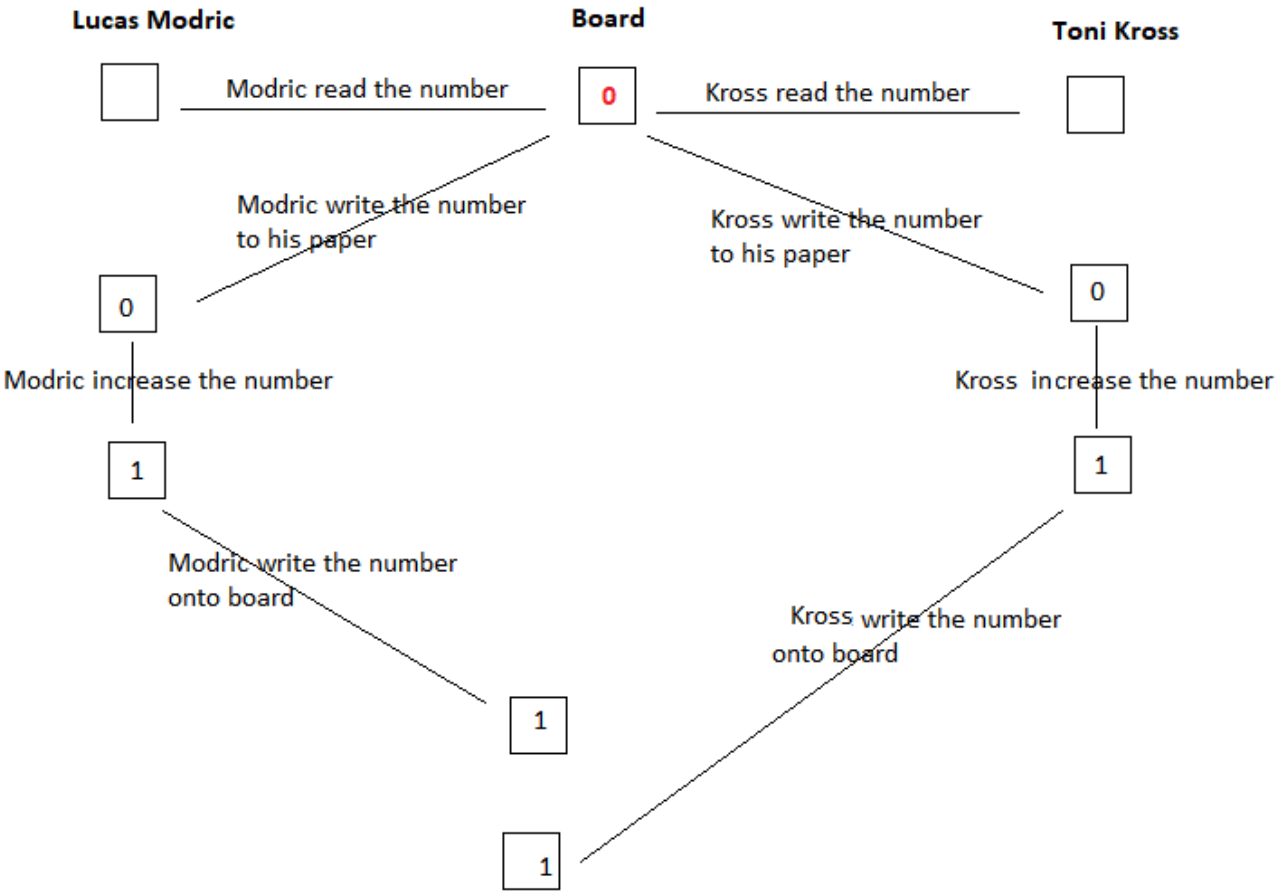
Mutex Type	Result
PTHREAD_MUTEX_NORMAL	Undefined behavior.
PTHREAD_MUTEX_ERRORCHECK	Return error code
PTHREAD_MUTEX_RECURSIVE	Return error code
PTHREAD_MUTEX_DEFAULT	Undefined behavior

Data Race
Data race occurs when:
- A memory location are accessed by two or more threads.
- At least one of the accesses is for writing. And the thread are not using any exclusive locks control to the memory.

Lets start with an example
Modric and Kross play a game.
There is a board with a number on it.
Modric and Kross read the number and write down to their paper.
They increase the number and write it onto the board concurrently.
What can happen?
Case 01



Case 02



In above example, the number is increased twice by Modric and Kross, however, its value can be 1 or 2. And we have the same problem in concurency programming. Lets start with a sample code.

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

int number = 0;

void* increase(void* arg){
    number++;
}

int main(){
    pthread_t modric;
    pthread_t kross;

    pthread_create(&modric, NULL, increase, NULL);
    pthread_create(&kross, NULL, increase, NULL);

    pthread_join(modric, NULL);
    pthread_join(kross, NULL);

    printf("%i", number);

    return 0;
}
```

The increment operation usually involve three steps:
- Load the memory location into a register
- Increase the value in register
- Write new value back to the memory location

And below is the instruction orders can happen when increase the same number by two thread concurrently.

Order 1

Step	thread modric	number	thread kross
1		number = 0	
2	load, number = 0		
3	increase, number = 1		
4		number = 1	
5			load, number = 1
6			increase, number = 2
7		**number = 2**	

Order 2

Step	thread modric	number	thread kross
1		number = 0	
2	load, number = 0		
3			load, number = 0
4	increase, number = 1		
			increase, number =1
		number = 1	

Using Mutex to Prevent Data Race
We can protect shared memory by allowing only one thread access it at a time by using mutex.

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

int number = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

void* increase(void* arg){
    pthread_mutex_lock(&mtx);

    number++;

    pthread_mutex_unlock(&mtx);
}

int main(){
    pthread_t modric;
    pthread_t kross;

    pthread_create(&modric, NULL, increase, NULL);
    pthread_create(&kross, NULL, increase, NULL);

    pthread_join(modric, NULL);
    pthread_join(kross, NULL);
```

```
    printf("%i", number);

    return 0;
}
```

Instruction order

Step	thread modric	number	thread kross
1		number = 0	
2	lock mutex		
3	load, number = 0		lock mutex mutex is acquired by thread modric so, thread kross suspend and wait until mutex is released
4	increase, number = 1		Waiting and suspending...
		number = 1	
	unlock mutex		
			mutex just be released thread kross acquires mutex and resume execution
			load, number = 1
			increase, number = 2
		number = 2	
			unlock mutex