**Mutex**

**Example 1** - The following code uses two threads that simultaneously increase/decrease a global variable. In the end, the output should be zero.

```cpp
#include <iostream>
#include <thread>
using namespace std;

int ballance = 0;

void change(int n){
    for (int i=0; i<1000000; i++){
        ballance += n;
    }
}

int main(int argc, char const *argv[]){
    thread t1(change, 1);
    thread t2(change, -1);
    t1.join();
    t2.join();
    cout<<ballance<<endl;
    return 0;
}
```

**Thread synchronization** is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as a critical section.

**Example 2** - Use mutex to protect the global variable in Example 1.

```cpp
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;
mutex myMutex;

int ballance = 0;

void change(int n){
```
for (int i=0; i<1000000; i++){
    myMutex.lock();
    ballance += n;
    myMutex.unlock();
}

int main(int argc, char const *argv[]) {
   thread t1(change, 1);
   thread t2(change, -1);
   t1.join();
   t2.join();
   cout<<ballance<<endl;
   return 0;
}

Write the previous example without using threads. Then compare the running time of both programs.

**Example 3 - Example 2 without using threads.**

```cpp
#include <iostream>
using namespace std;

int ballance = 0;

int main(int argc, char const *argv[]) {
   for (int i=0; i<1000000; i++)
   {
      ballance += +1;
      ballance += -1;
   }
   cout<<ballance<<endl;
   return 0;
}
```

- Compare the running time of Example 2 and Example 3?
- Which one is faster? Why?

**Example 4 -** The following code uses two threads to print 50 * in one line and 50 $ in another line.
```cpp
#include <iostream>
#include <thread>

using namespace std;

void print_block(int n, char ch) {
    for (int i=0; i<n; i++) {
        cout << ch;
    }
    cout << endl;
}

int main(int argc, char const *argv[]) {
    thread t1 (print_block, 500, '*');
    thread t2 (print_block, 500, $');

    t1.join();
    t2.join();

    return 0;
}

- Write, compile, and run it.
- Discuss the output with your peer.
- Discuss with your peer and use a mutex to fix the program.
```
The following examples are for students that are interested to use the thread and mutex with older versions of C / C++.

**Thread with C**

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>   // for sleep

void *print_hello_world(void *unused) {
    while(1) {
        printf("Hello Word\n");
        sleep(1);
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t mythread;
    int status;
    printf("Main here ...\n");
    status = pthread_create(&mythread, NULL, &print_hello_world, NULL);
    if (status != 0) {
        printf("Oops, pthread_create returned error code %d\n", status);
        exit(-1);
    }
    while(1) {
        printf("Hello CPSC\n");
        sleep(1);
    }
    return 0;
}
```

---

**Thread and Mutex with old versions of C++**

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

#define num_loops 100000000

long long sum = 0;
```
void* counting(void *arg) {
    int offset = *(int *) arg;
    for (long long i=0; i<num_loops; i++) {
        pthread_mutex_lock(&mutex);
        sum+=offset;
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t id1;
    int offset1 = 1;
    pthread_create(&id1, NULL, counting, &offset1);
    pthread_t id2;
    int offset2 = -1;
    pthread_create(&id2, NULL, counting, &offset2);

    //Wait for threads to finish
    pthread_join(id1, NULL);
    pthread_join(id2, NULL);

    printf("Sum = %lld\n", sum);

    return(0);
}