

Lecture 5 - Exercises

In this exercise there is to be worked with the basics of thread communication, but with using the message queue concept. There is to be created a system that consists of two threads, where one sends information and the other receives it.

Exercise 1 - Creating a message queue

Implementating the class Message

[Class Message](#)

```
#include <iostream>

using namespace std;

class Message
{
public:
    virtual Message~() {}

};
```

The implementation of the class Message is simply a virtual destructor. The destructor is virtual to ensure that not only the base class (Message) objects will be deleted. For the used communication in this exercise, it's important to take all the derived classes into account as well.

Implementating the class MsgQueue

In the class MsgQueue, we have used the template from Listing 1.2 from the exercise description and added struct Item and objects to the class' private.

The chosen container for the implementation is a queue, since it works as a FIFO, and is fast at inserting a new item and deleting the first most. For the operations needed in this exercise, we won't need to insert or delete in the middle of the container, which makes a queue the obvious choice.

[Class MsgQueue](#)

```
#include "Message"
#include <queue>
#include <pthread.h>

using namespace std;

class MsgQueue
{
public:
    MsgQueue(unsigned long maxSize);
    void send(unsigned long id, Message* msg = NULL);
    Message* receive(unsigned long& id);
    ~MsgQueue();
private:
    struct Item
    {
        unsigned long int id_;
        Message* msg_;
    }

    queue<Item*> container_;
    unsigned long maxSize_;
```

```

pthread_cond_t sendCond_          = PTHREAD_COND_INITIALIZER;
pthread_cond_t receiveCond_       = PTHREAD_COND_INITIALIZER;
pthread_mutex_t sendMutex_        = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t receiveMutex_     = PTHREAD_MUTEX_INITIALIZER;

};

```

The implementation of the send function, is made as seen in the collapse under. The while loop ensures, that the function will block, if the maximum capacity of the container is reached.

[Implementation of send function](#)

```

void MsgQueue::send(unsigned long id, Message* msg)
{
    //lock sendMutex_
    pthread_mutex_lock(&sendMutex_);

    while (container_.size() >= maxSize_)
    {
        pthread_cond_wait(&sendCond_, &sendMutex_);
    }

    //Create item and values
    Item* itemPtr = new Item;
    itemPtr->id_ = id;
    itemPtr->msg_ = msg;

    //Put item in queue
    container_.push(itemPtr);

    //Signal receiver
    pthread_cond_signal(&receiveCond_);

    //unlock sendMutex_
    pthread_mutex_unlock(&sendMutex_);

    cout << "Sending ID " << id << " to receiver" << endl;
}

```

For the implementation of the receive function, the while loop blocks if the container queue is empty.

[Implementation of receive function](#)

```

Message* MsgQueue::receive(unsigned long& id)
{
    //lock receiveMutex_
    pthread_mutex_lock(&receiveMutex_)

    while (container_.size() == 0)
    {
        pthread_cond_wait(&receiveCond_, &receiveMutex_);
    }

    // Receive item and save in new variables
    Item* resultPtr = container_.front();
    id = resultPtr->id_;
    Message* msgPtr = resultPtr->msg_;

    //Pop that shit
}

```

```

    container_.pop();

    //Delete item to clean up
    delete resultPtr;

    //Signal sender
    pthread_cond_signal(&sendCond_);

    //Unlock receiveMutex_
    pthread_mutex_unlock(&receiveMutex_);

    cout << "Received ID " << id << " from sender" <<endl;

}

```

The full implementation can be found in our [repository for lecture 5, exercise 1](#)

Exercise 2 - Sending data from one thread to another

The goal for this exercise is to test Message and MsgQueue implemented in Exercise 1. This is done by creating two threads, a *sender* and a *receiver* and creating a `main()` which spawns the two threads.

The sender thread, will create a `Point3D`-object with 3 coordinate-points. The receiver thread will wait for the message and print the x, y and z coordinates to the terminal as soon as it receives it.

The `point3D` is implemented as a simple struct, as given in Listing 2.1.

The rest of the program for printing out the coordinates are written in a [main.cpp](#)

The most interesting functions for the program are the Sender and the Receiver, and while the `handleMsg` also is very important for the program, it isn't doing anything new or groundbreaking, which is why, we won't comment further on it for now.

Sender

The Sender first cast the void pointer to a `MsgQueue` pointer. Then the randomized values to the coordinate is made and sent to the queue.

To make sure we get the same values in the receiver as we sent, we chose to print out our programs randomized coordinates fast to compare.

Lastly the function sleeps for one second.

[The Sender function can be seen here](#)

```

void* Sender(void* arg)
{
    MsgQueue* messageQueue = (MsgQueue*)(arg);

    while(true) {
        point3D* point3d = new point3D;
        point3d->x = rand() %10+1;
        point3d->y = rand() %10+1;
        point3d->z = rand() %10+1;
        messageQueue->send(p3d, point3d);

        cout << "SEND THIS: (" << point3d->x << ", " << point3d->y << ", " << point3d->z << ")"
        << endl;

        sleep(1); //Every second or so
    }
}

```

Receiver

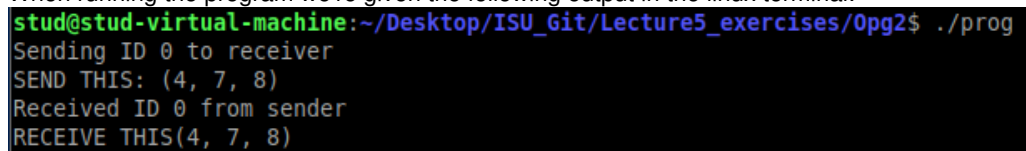
The Receiver keeps checking for a message from the Sender. As soon as the Receiver gets the message, it calls the handler, which makes prints the received coordinates to the terminal. As soon as the Receiver has sent the information to the handler, it deletes the message.

ReceiverReceiver

```
void* Receiver(void* arg)
{
    MsgQueue *messageQueue = (MsgQueue*)(arg);
    unsigned long id;
    while(true)
    {
        Message* msg = messageQueue -> receive(id);
        handleMsg (id, msg);
        delete msg;
    }
}
```

The full implementation can be found in our [repository for lecture 5, exercise 2](#).

When running the program we're given the following output in the linux terminal:



```
stud@stud-virtual-machine:~/Desktop/ISU_Git/Lecture5_exercises/0pg2$ ./prog
Sending ID 0 to receiver
SEND THIS: (4, 7, 8)
Received ID 0 from sender
RECEIVE THIS(4, 7, 8)
```

Terminal output for exercise 2 - Point3D

By showing both the expected and the received output, we have shown the program works as planned.

Questions

Who is responsible for disposing any given message?

The receiver is responsible for disposing the message after it's done using it.

Who should be the owner of the object instance of class MsgQueue; Is it relevant in this particular scenario?

Both the Sender and Receiver need to access the MsgQueue, since they both need to use the function implemented in the class.

Is it relevant for us? :(

How are the threads brought to know about the object instance of MsgQueue?

As soon as the threads are created, the threads are told about it. This is done by adding MsgQueue as the fourth parameter of the pthread_create:

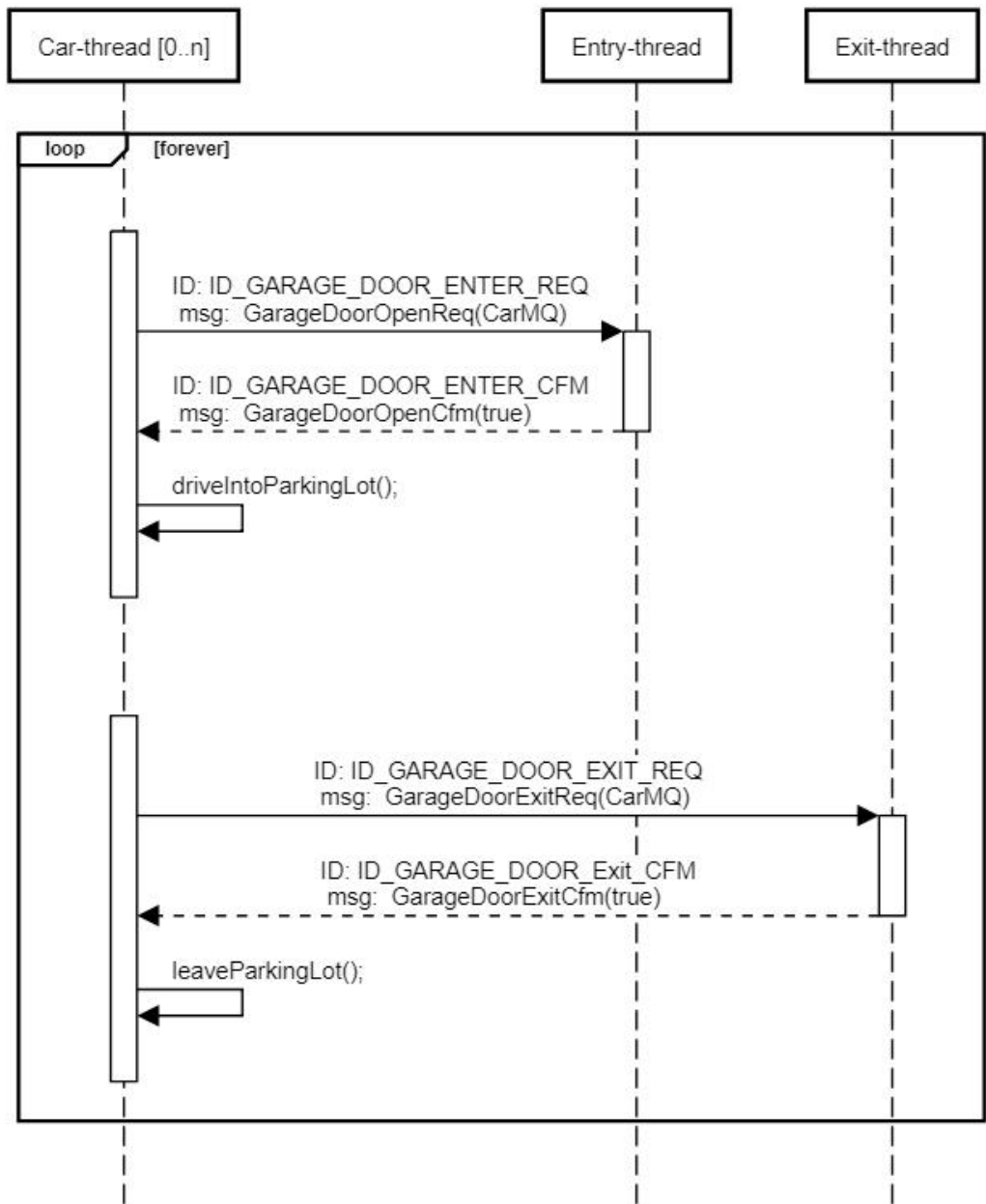
```
pthread_create(&senderThread, nullptr, Sender, messageQueue);
pthread_create(&receiverThread, nullptr, Receiver, messageQueue);
```

Exercise 3 - Enhancing the PLCS with Message Queues

The goal for this exercise is to reimplement the PLCS solution from [Lecture 4 - Exercises](#) using Message Queues as a mean of communication.

Before making the new implementation of the Park Lot Control system the following sequence diagram is made to illustrate the communication sequence between the car(s), the EntryDoor and the ExitDoor.

SQ PLCS



Sequence diagram for PLCS with Message Queues

However, even though we tried, we didn't succeed at implementing the PLCS with message queues. Our attempt can be found in our repo.

Questions

What is an event driven system? - And were do you start *this* event driven system?

An Event Driven System (EDS) is system that relies on event to run the program. For instance our program would never do

anything, if the one of the cars didn't request entry to the parking lot; the event ID_GARAGE_DOOR_OPEN_REQ is the one starting out EDS.

Explain your design choice in the specific situation where a given car is parked inside the carpark and waiting before leaving. Specifically how is the waiting situation handled?

Pass

Why is it important that each *car* has it own *MsgQueue*?

If all the cars shared the same *MsgQueue*, the entry and exit threads wouldn't be able to navigate which car should enter or exit.

Compare the original *Mutex/Conditional* solution to the *Message Queue* solution.

The Message Queue solution didn't work out for us.

Filer

Terminal_opg2.png	12,8 KB	2018-10-30	Mie Nielsen
sq_plcs.png	42,6 KB	2018-10-30	Mie Nielsen