Wiki »

# Exercise 6 - Thread Communication

## Introduction:

In this exercise we will use basic thread communication, using the message queue concept. We will create a system consisting of two threads that communicate, one sending information that the other receives. This will help us get knowledge of how to create a message, send it via. message queue and receive and handle it in another thread.

# Exercise 1: Creating a message queue

Firstly we were to use pseudo code to create a design

Pseudo

```
 void handler (id, message)
{
    if id == ..
     call appropriate handler subroutine with a message as parameter.
    break;

}

void reciever
{
    // Loop forever
      {
        // Recieve message
        // handle message
        // Delete message
      }

}

void sender
{
   //
}
```

We start out by creating the class Message. This class will now serve as the base of all messages must inherit from this class. It is important that the destructor is virtual.

**This is important, because if we must delete a derived class object using a pointer to a base class that has a non-virtual destructor, it may result in undefined behavior.**

Message.hpp

```


using namespace std;
class Message
```

```
{
public:
    virtual ~Message() {}
};
```

MsgQueue.hpp

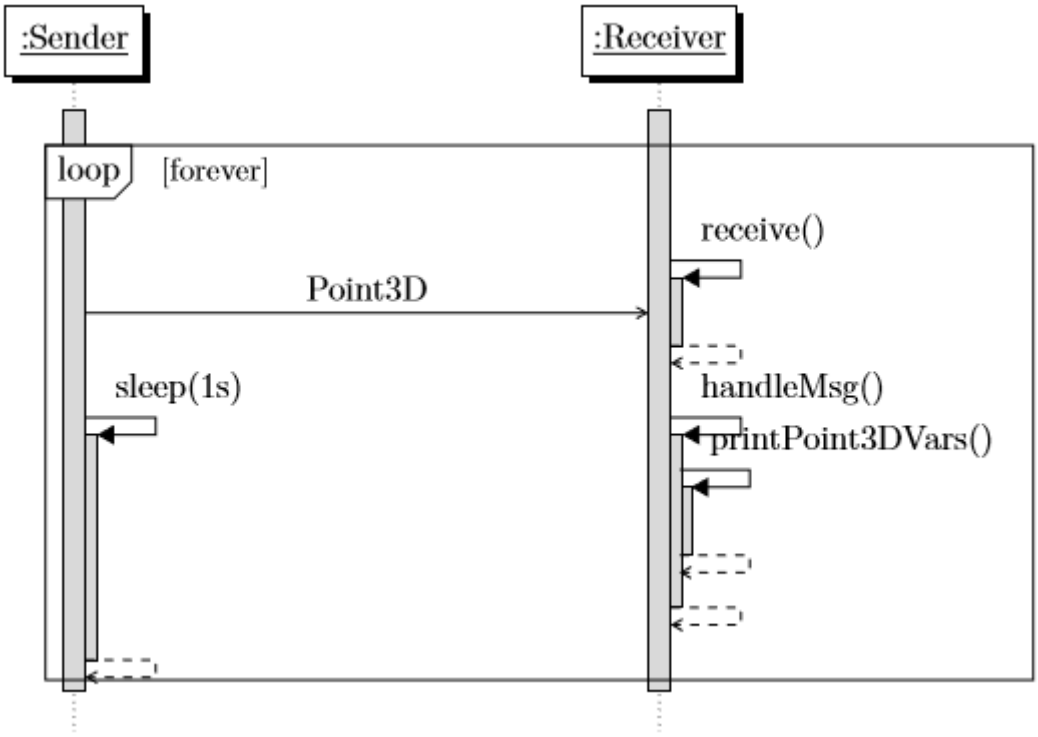
```
#ifndef MSGQUEUE_HPP_
#define MSGQUEUE_HPP_
#include <stdio.h>
#include <queue>
#include "Message.hpp"
#include <iostream>
using namespace std;

class MsgQueue : public Message
{
public:
    MsgQueue(unsigned long maxSize);
    {
        space = maxSize;
        numOfMessages = 0;
    }
    void send(unsigned long id, Message* msg = NULL);
    {
        pthread_mutex_lock(&messageMutex);
        while(numOfMessages == space) {
            pthread_cond_wait(&messageContainerFull, &messageMutex);
        }
        cout << "send\n" << endl;
        Message message;
        message._id = id;
        message._msg = msg;
        message.push(Message);
        numOfMessages++;
        pthread_cond_signal(&messageContainerEmpty);

        pthread_mutex_unlock(&messageMutex);
    }

    Message* receive(unsigned long& id); {
        pthread_mutex_lock(&messageMutex);
        if(current==0) {
            pthread_cond_wait(&messageContainerEmpty, &messageMutex);
        }
        Message message = message.front();
        message.pop();
        numOfMessages--;
        id = message._id;
        pthread_mutex_unlock(&messageMutex);
        return message._msg;
    }

    ~MsgQueue() {
    }

private:
```

```
    struct Message                  //struct som holder message med id
    {
        unsigned long _id;
        Message* _msg = NULL;
    };

    std::Queue<Message> container;          //message container
    int space = 10;                        //plads i køen
    int numOfMessages = 0;       //antal messages i køen
    pthread_cond_t messageContainerFull = PTHREAD_COND_INITIALIZER;   //Condition si
    pthread_cond_t messageContainerEmpty = PTHREAD_COND_INITIALIZER; //Condition sig
    pthread_mutex_t messageMutex = PTHREAD_MUTEX_INITIALIZER;            //Mutex
};

#endif
```

We are using a Queue as our container, since this is a type which fits the exercise well, since you easily can pop the first element.

# Exercise 2: Sending data from one thread to another

Now we will create a main program with two threads: Sender and Receiver. The Sender creates an object Point3D every second or so, and sends it to the Receiver. The Receiver should continuously wait for new messages, and on reception print the x,y and z coordinates of the received Point3D object to the console. The following figure illustrates this:




Here we show the main program code:

Here we show the output in the console:

```
stud@stud-virtual-machine:~/i3isu/lecture6/Ex1.2$ ./main
send
 ( x =0, y=1, z=2 )
send
 ( x =1, y=2, z=3 )
send
 ( x =2, y=3, z=4 )
send
 ( x =3, y=4, z=5 )
send
 ( x =4, y=5, z=6 )
send
 ( x =5, y=6, z=7 )
send
 ( x =6, y=7, z=8 )
send
 ( x =7, y=8, z=9 )
send
 ( x =8, y=9, z=10 )
send
 ( x =9, y=10, z=11 )
send
 ( x =10, y=11, z=12 )
send
 ( x =11, y=12, z=13 )
send
 ( x =12, y=13, z=14 )
```

Exercise questions:

**Who is responsible for disposing any given message?**

It is the receiver who runs the handler, and deallocates the message.

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

In this Exercise(2), the object of MsgQueue is declared as a global instance. Since we only have one user, we like to use a global MsgQueue, which is shared between the Sender and Receiver thread functions.

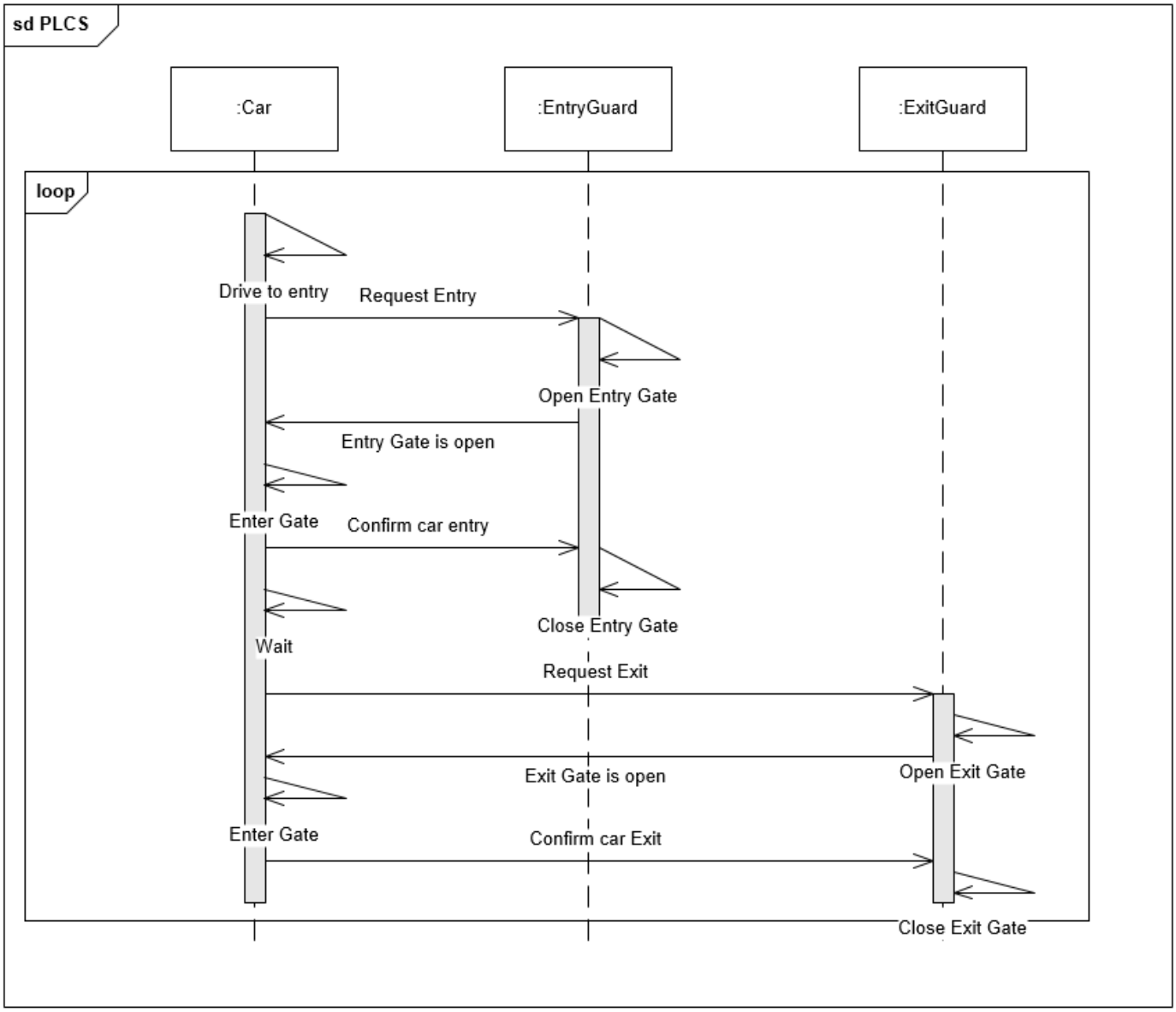
**How are the threads brought to know about the object instance of MsgQueue?**

Since MsgQueue were declared as a global instance, and all threads are created in the same program, Sender and Receiver share the object.

# Exercise 3: Enhancing the PLCS with Message Queues

Now we will reimplement our PLCS solution from Exercise 5, using Message Queues and thus messages as a means to communicate between car threads and door controller thread.

We created the following sequence diagram to show the how the PLCS will work:




Here we show the code:

main.cpp

```
#include "MsgQueue.hpp"
#include <unistd.h>

#define NUMBER_CARS 10

MsgQueue mqCar(NUMBER_CARS);
MsgQueue mqExit(NUMBER_CARS);
MsgQueue mqEntry(NUMBER_CARS);

int numCars = 0;
bool openEntry();
bool openExit();

void driveIntoParkingLot();
void driveOutOfParkingLot();

pthread_t carT[NUMBER_CARS];
pthread_t entryT, exitT;
```

```
enum
{
    ID_ENTRY_RQ,
    ID_ENTRY_CFM,
    ID_EXIT_RQ,
    ID_EXIT_CFM,
    ID_CAR_IN,
    ID_CAR_OUT
};

struct entryGuardOpenReq : public Message
{
    MsgQueue * mq;
};
struct entryGuardOpenCfm : public Message
{
    bool result_;
};
struct exitGuardOpenReq : public Message
{
    MsgQueue * mq;
};
struct exitGuardOpenCfm : public Message
{
    bool result_;
};
struct carIn : public Message
{
    MsgQueue * mq;
};
struct carOut : public Message
{
    MsgQueue * mq;
};

void carHandleIdEntryGuardCfm(entryGuardOpenCfm* cfm)
{
    if(cfm->result_)
    {
        driveIntoParkingLot();
    }
}
void carHandleIdExitGuardCfm(exitGuardOpenCfm* cfm)
{
    if(cfm->result_)
    {
        driveOutOfParkingLot();
    }
}
void carHandleIdEntryGuardReq(entryGuardOpenReq* req)
{
    entryGuardOpenCfm* cfm = new entryGuardOpenCfm;
    cfm->result_ = openEntry();

    req->mq->send(ID_ENTRY_CFM, cfm);
}
void carHandleIdExitGuardReq(exitGuardOpenReq* req)
```

```
{
    exitGuardOpenCfm* cfm = new exitGuardOpenCfm;
    cfm->result_ = openExit();

    req->mq->send(ID_EXIT_CFM, cfm);
}
void carHandleIn()
{
    entryGuardOpenReq* req = new entryGuardOpenReq;
    req->mq = &mqCar;

    mqEntry.send(ID_ENTRY_RQ, req);
}
void carHandleOut()
{
    exitGuardOpenReq* req = new exitGuardOpenReq;
    req->mq = &mqCar;

    mqExit.send(ID_EXIT_RQ, req);
}

void carHandler(unsigned long id, Message* msg)
{
    switch(id)
    {
        case ID_CAR_IN:
            carHandleIn();
            break;
        case ID_CAR_OUT:
            carHandleOut();
            break;
        case ID_ENTRY_CFM:
            carHandleIdEntryGuardCfm(static_cast<entryGuardOpenCfm*>(msg));
            break;
        case ID_EXIT_CFM:
            carHandleIdExitGuardCfm(static_cast<exitGuardOpenCfm*>(msg));
            break;
        default:
            break;
    }
}
void* entryGuardController(unsigned id, Message* msg)
{
    switch(id)
    {
        case ID_ENTRY_RQ:
            carHandleIdEntryGuardReq(static_cast<entryGuardOpenReq*>(msg));
            break;
        default:
            break;
    }
    return nullptr;
}
void* exitGuardController(unsigned id, Message* msg)
{
    switch(id)
    {
        case ID_EXIT_RQ:
```

```
            carHandleIdExitGuardReq(static_cast<exitGuardOpenReq*>(msg));
            break;
        default:
            break;
    }
    return nullptr;
}
void *entryGuard(void* ptr)
{
    while(1)
    {
        unsigned long id;
        Message* msg = mqEntry.receive(id);
        entryGuardController(id, msg);
        delete(msg);
    }
}

void* exitGuard(void* ptr)
{
    while(1)
    {
        unsigned long id;
        Message* msg = mqExit.receive(id);
        sleep(1);
        exitGuardController(id, msg);
        delete(msg);
    }
}
void *cars(void*)
{
    while(1)
    {
        unsigned long id;
        Message* msg = mqCar.receive(id);
        carHandler(id, msg);
        delete(msg);
    }
}

void startCarThread()
{
    for(int i = 0; i<NUMBER_CARS; i++)
    {
        pthread_create(&carT[i], nullptr, cars, nullptr);
        cout << "Car " << i << " being manufactured!" << endl;
    }
    mqCar.send(ID_CAR_IN);
}

int main(int argc, char * argv[])
{
    startCarThread();
    pthread_create(&entryT, nullptr, entryGuard, nullptr);
    pthread_create(&exitT, nullptr, exitGuard, nullptr);

    pthread_join(entryT, nullptr);
    pthread_join(exitT, nullptr);
```

```
    return 0;
}
void driveIntoParkingLot()
{
    numCars++;
    cout << "A car has entered the parking lot.\nNumber of cars in parking lot: " <<
    sleep(1);
    mqCar.send(ID_CAR_OUT);

}
void driveOutOfParkingLot()
{
    numCars--;
    cout << "A car has left the parking lot.\nNumber of cars in parking lot: " << nu
    mqCar.send(ID_CAR_IN);
}
bool openEntry()
{
    cout << "Entry is opened!" << endl;
    return true;
}
bool openExit()
{
    cout << "Exit is opened!" << endl;
    return true;
}
```

Here is the console output:

```
stud@stud-virtual-machine:~/i3isu/lecture6/exercise3$ ./main
Car 0 being manufactured!
Car 1 being manufactured!
Car 2 being manufactured!
Car 3 being manufactured!
Car 4 being manufactured!
Car 5 being manufactured!
Car 6 being manufactured!
Car 7 being manufactured!
Car 8 being manufactured!
Car 9 being manufactured!
Entry is opened!
A car has entered the parking lot.
Number of cars in parking lot: 1
Exit is opened!
A car has left the parking lot.
Number of cars in parking lot: 0
Entry is opened!
A car has entered the parking lot.
Number of cars in parking lot: 1
Exit is opened!
A car has left the parking lot.
Number of cars in parking lot: 0
Entry is opened!
A car has entered the parking lot.
Number of cars in parking lot: 1
Exit is opened!
A car has left the parking lot.
Number of cars in parking lot: 0
Entry is opened!
A car has entered the parking lot.
Number of cars in parking lot: 1
Exit is opened!
A car has left the parking lot.
Number of cars in parking lot: 0
```

We succesfully created cars using the MsgQueue PLCS. However it seems our cars leave the parking lot just as fast as entering, which we couldn't find a solution for.

Questions:

**What is an event driven system?**

An event-driven system or program, works by depending on external events. The system will typically stay in a state of waiting / listening for events, and then handle them and then return to the "wait" state again.

**How and where do you start this event driven system? (remember it is purely reactive!)**

The events in this system happen, when the car requests the Entry / Exit guards. Their receive function will stop blocking and then the handler activates.

**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?**

We use a function called driveIntoParkingLot() in our carHandleIdEntryGuardCfm() and carHandleIdExitGuardCfm(). This function sleeps for 1 second, then a message is sent to MqCar, which signals that we have a car ready to exit.

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

This is needed, so that we may keep track of which car thread is doing the requests.

**Compare the original Mutex/Conditional solution to the Message Queue solution.**
***In which ways do they resemble each other? Consider stepwise what happens in the original code and what happens in your new implementation based on Message Queues.**

They both use mutex and conditionals. However with Message Queues we use switchcases controlled by an id.

***What do you consider to be the most important benefit achieved by using the EDP approach, elaborate.**

The event driven programming approach is beneficial, since we can ensure that a given job is handled by the correct handler. Also EDP makes it possible to implement system with states.

STM_SendReceive.PNG (12,2 KB) Mikkel Mahler, 2019-04-02 15:17
SD_PLCS.PNG (31,8 KB) Mikkel Mahler, 2019-04-02 15:38
Ex2Console.png (32,3 KB) Mikkel Mahler, 2019-04-02 17:56
terminal3.png (49,9 KB) Andreas Ventzel, 2019-04-02 18:00