Gruppe 33 - Exercise6 - # 34

Inter Thread Communication

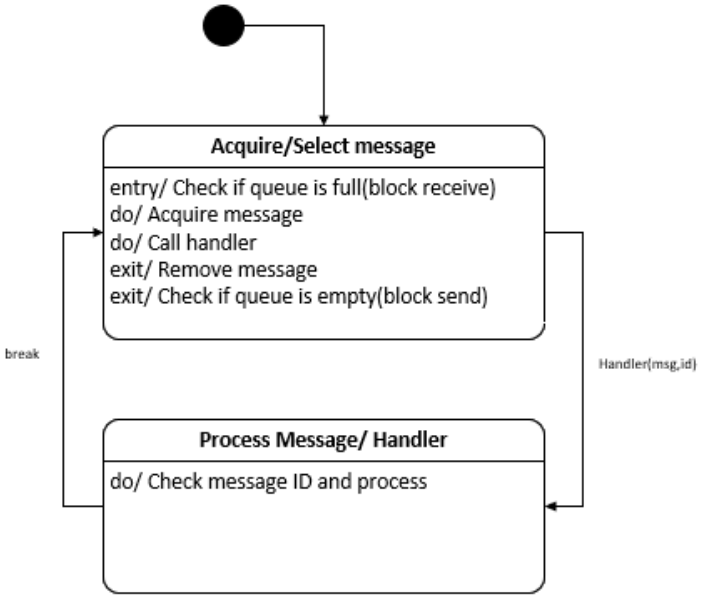
"In this exercise you will learn the basics of thread communication using the message queue concept presented in class. You will create a system consisting of two threads that communicate, one sending information that the other receives. You will hereby acquire knowledge of how to create a message, send it via a message queue and, receive and handle it in another thread."

Læringsmål for øvelsen:

1. 1
2. 2
3. 3
4. 4

Exercise 1 - Creating a message queue

Exercise 1.1 - Creating a message queue - Design



Exercise 1.2 - Creating a message queue - Implementation

Implementering af klassen Message

Først laves klassen Message, som vist i øvelsesbeskrivelsen listing 1.1

```
#ifndef MESSAGE_H
#define MESSAGE_H

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

#endif
```

Implementering af klassen MsgQueue

Herefter klassen MsgQueue, med udgangspunkt i listing 1.2 i øvelsesbeskrivelsen.

```
#ifndef MSGQUEUE_H
#define MSGQUEUE_H

#include "Message.hpp"

#include <queue>
#include <iostream>

using namespace std;

struct MsqItem
{
    MsqItem (unsigned long id, Message * msg  ):id_(id), msg_(msg)
    {};
    unsigned long id_;
    Message* msg_;
};

class MsgQueue
{
public:
    MsgQueue( unsigned long maxSize):maxSize_ (maxSize)
    {};
    void send( unsigned long id, Message* msg = NULL);
    Message* receive( unsigned long &id);
    ~MsgQueue();
private:
    // Container with messages
    queue < MsqItem * >messageQueue_;
    // Plus other relevant variables
    const unsigned long maxSize_;

};

void MsgQueue::send( unsigned long id, Message* msg)
{
    while(messageQueue_.size() >= maxSize_) //Blocking if queue is filled
    {};

    MsqItem *item = new MsqItem(id, msg);
    messageQueue_.push(item); //Add to queue
}

Message* MsgQueue::receive( unsigned long &id)
{
    while(messageQueue_.empty()) //Blocking if queue is empty
    {};

    MsqItem *item = messageQueue_.front(); //First in queue
    messageQueue_.pop(); //Remove first

    id = item->id_; //Identify

    return item->msg_; //Return Message
}

#endif
```

Diskussion / Konklusion

Remember that the destructor must be virtual.Why is this last bit very important? Explain!

Vi ønsker at undgå at når en Message * msg slettes, slettes kun denne dervied class.

Message er vores base clase hvor vi implementere den virtuel destructor.
Når vi laver flere udgaver af Message (polymorphism) som med tiden skal nedlægges, og plads frigives, hvis destrucotren i base klassen kaldes i stedet for dervied klassens egen destructor, opstår der problemer.

Exercise 2 - Sending data from one thread to another

Exercise 2.1 - Sending data from one thread to another - Design

Sekvensdiagrammet til denne opgave er kopieret fra opgavebeskrivelsen, og yderligere design i form af statemachine- eller sekvensdiagrammer er undladt.

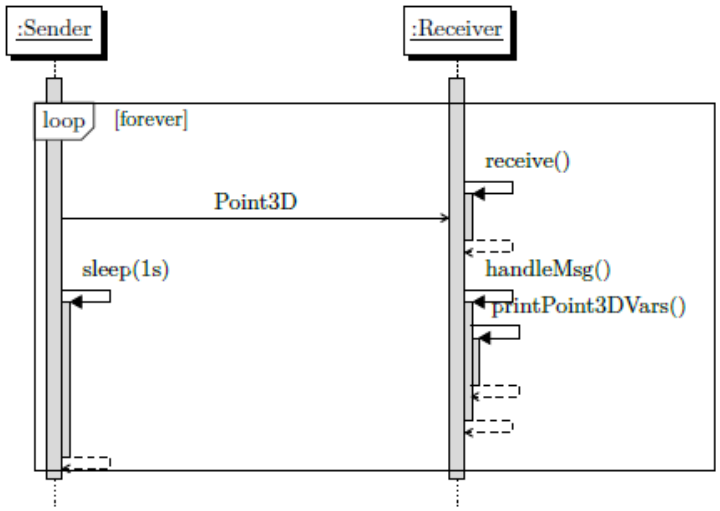


Figure 2.1: The *Sender* sends a `Point3D` object to the *Receiver*

Exercise 2.2 - Sending data from one thread to another - Implementation

Implementering af Point3D

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

#include <Message.hpp>
#include <MsgQueue.hpp>

using namespace std;

struct Point3D : public Message
{
    Point3D(int x_in, int y_in, int z_in):x(x_in), y(y_in), z(z_in)
    {};

    int x;
    int y;
    int z;
};

enum { ID_POINT3D};

void *senderFunction(void *arg)
{
    MsgQueue* point3D_queue = (MsgQueue*)arg;
    for(;;)
    {
        Point3D *p3D = new Point3D(5,2,7);
        point3D_queue->send(ID_POINT3D, p3D); //Push to queue

        sleep(1);
    }
}
```

```
void point3DHandler(Message *msg, int id)
{
    switch(id)
    {
        case ID_POINT3D:
            Point3D * pointMsg = static_cast < Point3D * >(msg);
            cout << "x: " << pointMsg->x <<" y: "<< pointMsg->y <<" z: "<< pointMsg->z <<"\n";
        break;
        /*default:
            cout << "No ID found \n";*/
    }
}

void *receiverFunction(void *arg)
{
    MsgQueue* point3D_queue = (MsgQueue*)arg;

    for(;;)
    {
        unsigned long id;
        Message *msg = point3D_queue->receive(id);
        point3DHandler(msg, id);

        delete msg;
    }
}

int main(void){

    MsgQueue point3D_queue(2); //Max size 2

    pthread_t sender, receiver;
    int tSender, tReceiver;

    tSender = pthread_create(&sender, NULL, senderFunction, &point3D_queue);
    tReceiver = pthread_create(&receiver, NULL, receiverFunction, &point3D_queue);

    pthread_join(sender, NULL);
    pthread_join(receiver, NULL);

    return 0;
}
```

Resultat

```
stud@stud-virtual-machine:~/isu_work/exercise6/exercise2/bin/x86-64$ ./point3D
x: 5 y: 2 z: 7
x: 5 y: 2 z: 7
x: 5 y: 2 z: 7
x: 5 y: 2 z: 7
x: 5 y: 2 z: 7
^C
```

Diskussion / Konklusion

Questions to answer:
• Who is responsible for disposing any given message?

Recieveren læser en besked, kalder handler(som håndtere eventet), og Recieveren sletter herefter beskeded.

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

Recieveren bør af ovenstående oversag også have ejerskabet over MsgQueue. I et scenarie med flere der sender og en enkelt receiver vil det give mening,
men i dette tilfælde med 1 sender og 1 modtager, gør det ikke stor forskel.

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

Når en thread oprettes, gøres dette med en reference til køen - Eksemepel:

```
tSender = pthread_create(&sender, NULL, senderFunction, &point3D_queue);
```

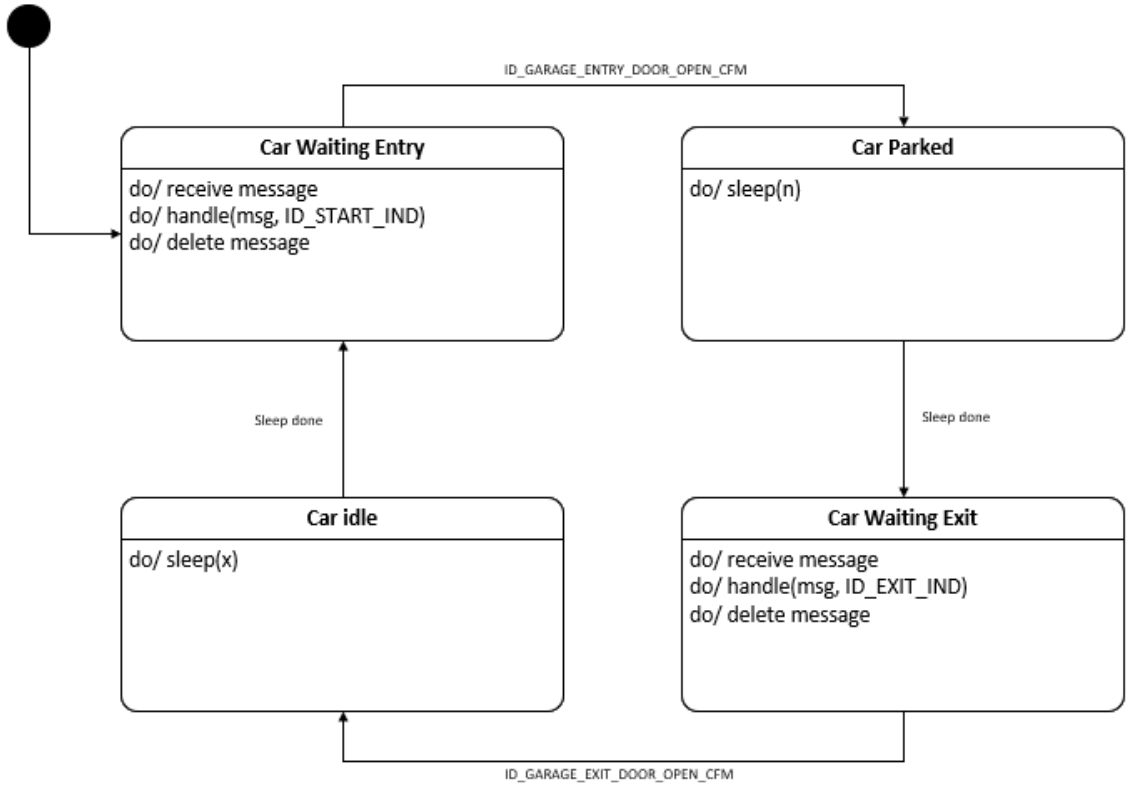
printPoint3DVars som er vist på sekvensdiagrammet er ikke implementeret som funktion, men blot udskrift "cout" i handleren.

Exercise 3 - Enhancing the PLCS with Message Queues

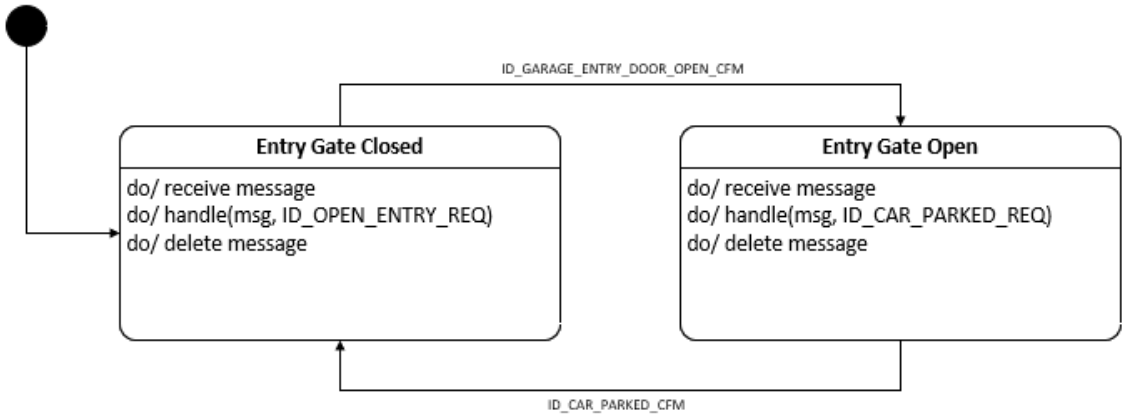
Exercise 3.1 - Sending data from one thread to another - Design

Til denne opgave er der lavet Statemachine(STM) diagrammer for de enkelte threads i systemet(Car, Entry og Exit).

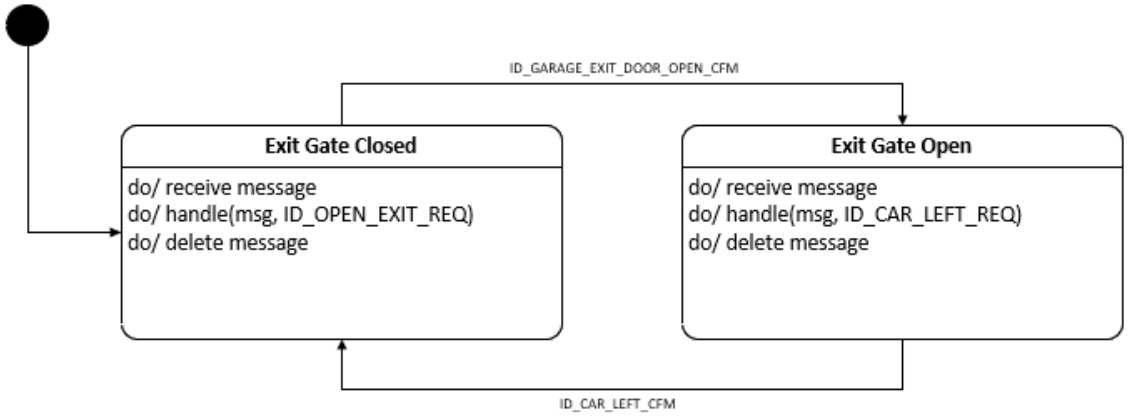
Car thread



Entry Gate Thread



Exit Gate Thread



Exercise 3.2 - Sending data from one thread to another - Implementation

Implementering af Car Thread

Car Thread initiering

```
void startCarThread()
{
    int carThreads = 5;
    int t;

    pthread_t carThread[carThreads];

    for (int i = 1; i < carThreads; ++i)
    {
        //printf("New Car thread: %i\n", i);
        t = pthread_create(&carThread[i], NULL, threadFunction, NULL);
        if (t != 0)
        { // return 0 on succes
        printf("pthread_create failed\n");
        }
        sleep(2);
    }

};
```

threadFunction

```
void *threadFunction(void *arg)
{
    MsgQueue* carMsg_queue(10);
    carMsg_queue.send(ID_CAR_ENTRY_REQ);

    for(;;)
    {
        unsigned long id;
        Message* msg = carMsg_queue.receive(id);
        carHandler(msg, id);
        delete msg;
    }

};
```

Her kaldes carHandler som er implementeret således

```
void carHandler(Message *msg, int id)
{
    switch(id)
    {
        case ID_CAR_ENTRY_REQ:
```

```
            carHandleEntryReq();
            cout << "Car request entry\n " ;
        break;
        case ID_ENTRY_OPEN_CFM:
            carHandleEntryOpenCfm(static_cast<GarageEntryOpenCfm*>(msg));
            cout << "Entry gate open!";
        break;
    }

};
```

Og de to carHandlers

```
void carHandleEntryReq()
{
    GarageEntryOpenReq *req = new GarageEntryOpenReq;
    req->whoIsAskingMQ_ = &carMsg_queue;

    garageEntryControllerMq.send(ID_CAR_ENTRY_REQ, req);
};

void carHandleEntryOpenCfm(GarageEntryOpenCfm *cfm)
{
    if(cfm->result_)
    {
        cout << "Car is Parking ..\n"
    }
};
```

Implementering af entry Thread

Initiering

```
void startEntryThread()
{
    pthread_t garageEntry;

    MsgQueue garageEntryControllerMq;

    int tEntry;

    tEntry = pthread_create(&garageEntry, NULL, threadFunction, NULL);

};
```

threadFunction

```
void *threadFunction(void *arg)
{
    for(;;)
    {
        unsigned long id;
        Message* msg = carMsg_queue.receive(id);
        entryHandler(id, msg);
        delete msg;
    }

};
```

Entry gate handler

```
void entryOpenReqHandle(GarageEntryOpenReq *req)
{
```

```
    GarageEntryOpenCfm *cfm = new GarageEntryOpenCfm;
    cfm->result_= openEntry();

    req->whoIsAskingMQ_.send(ID_ENTRY_OPEN_CFM, cfm);
};

void entryHandler(unsigned long id, Message *msg)
{
    switch(id)
    {
        case ID_CAR_ENTRY_REQ:
           entryOpenReqHandle(static_cast<GarageEntryOpenReq*>(msg));
        break;
    }
};
```

Resultat af implementering

Det er ikke lykkedes at implementere(compilere og build) apllikationen/systemetet.

Diskussion / Konklusion

Grunden til at der i denne øvelse er benyttet netop statemachine diagrammer skyldes, sammenhængen med pensum for denne øvelse, at netop STM er event driven.

STM er baseret på "Events" og efterfølgende "Actions".

Hvor Even driven programming ligeledes benytter "Events" som signalering, og herefter "Action" i form af en specifik handler function().

Questions to answer:
• What is an event driven system?

Som en statemachine, hvor specifikke events bliver sendt til specifikke handlinger, og derved skiftes tilstand.

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

Det starter ved Acquire/Select new message. Der "lyttes" i et event-loop, og sker der et forventet event, håndteres dette af en specifik handler.

• 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?

Det er designet som implementeringen i sidste øvelse, hvor tiden er "eventet". Det er derfor ikke en nyt event, men en forlængelse af detevent som sikrede at bilen parkerer i garagen -> venter -> signalere til exit_gate(næste event, som registreres i event-loopet).

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

Ellers ville de forskellige carThreads alle sidde i event-loop, læse fra samme MsgQueue og vente på at modtage eksempelvis entryGateOpenCFM. Istedet kan hver thread modtage, handle og slette messages.

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

Message queue gør det muligt at forhindre problemet fra sidste øvelse, hvor sheduleren bestemmer hvilken car thread som får adgang. Kommer bil 2# til en fyldt garage og stiller sig i køre, risikeres der ikke at bil#3 kommer efterfølgende og får først adgang.

– 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.*

Sidste øvelse:
Bil ankommer og "signalere" ankomst.
Entry gaten modtager og åbner op(hvis der er plads) og signalere at porten nu er åben.
Bil modtager bekseden kører ind og signalere at Entry Gaten kan lukke.
Entry gate modtager og lukker porten igen.
Gentages for exit gaten.

Denne øvelse
Car thread venter i event-loop.
Entry gate venter i event-loop.
Car thread læser INIT besked, handleren kaldes og håndtere den og der sendes en request til Entry gaten.
Entry gate modtager beskeden, handleren kaldes og håndtere den og der sendes en confirm til car.
Car modtager beskeden, handleren kaldes og håndtere den - Car kan køre ind i garagen ...

Filer

ex2.PNG	16,9 KB	09.11.2021	Rasmus Baunvig Aagaard
ex3.1.PNG	16,5 KB	09.11.2021	Rasmus Baunvig Aagaard
ex3.2.PNG	10,3 KB	09.11.2021	Rasmus Baunvig Aagaard
ex3.3.PNG	9,52 KB	09.11.2021	Rasmus Baunvig Aagaard
ex1.PNG	12,1 KB	09.11.2021	Rasmus Baunvig Aagaard