Wiki »

Exercise 6 - Thread Communication

I denne øvelse vil der blive præsenteret tråd-kommunikation ved at bruge message-queue konceptet som en klasse. Til at vise dette vil der blive oprette et system af to threads som kommunikerer, hvor den ene er sender og den anden er modtager.

Exercise 1 Creating a message queue

I denne øvelse vil klassen MsgQueue blive beskrevet, designet og implementeret. Klassen er afgørende for de næste opgaver hvor den vil blive brugt til først at sende et 3D koordinat mellem to tråde og derefter vil et tidligere system, nemlig PLCS, blive omdannet til også at gøre brug at MsgQueue klassen til at kommunikere mellem tråde.

Design

For at gøre implementations-fasen nemmere og mere overskuelig, vil klasserne Messages, MsgQueue samt funktioner først blive beskrevet i pesdokode, og kan ses nedestående:

Pesudokode:

Message

```
variabel: messages

//empty:
send -> messages++ addMessage(); go to !empty&!full

//full:
```

```
receive -> messages-- exportMessage(); go to !empty&!full

//!empty & !full:
receive -> messages-- exportMessage(); if messages==0 go to empty
send -> messages++ addMessage(); if messages==size go to full
```

MsgQueue

```
 MsgQueue(unsigned long maxSize)
{
set max size of queue
}

~MsgQueue()
{
delete queue;
}
```

Send og Receive

```
Message* receive(unsigned long& ID);
{
while(queueEmpty){}
retrieve first messagequeue struct in queue (queue.front)
assign id of messagequeue object to ID
remove first Messagequeue struct (queue.pop)
decrement number of items in queue
return MessageToReceive;
}

send(unsigned long id,Message* msg=NULL)
{
while(queueFull){}
add message to message queue (queue.push)
increment number of items in queue
}
```

Til valg af container har vi følgende krav.

- fifo - first in first out.
- push() - man skulle kunne indsætte et objekt bagerst i køen.
- pop() - man skulle kunne fjerne det forreste det element i køen.

Valget er derfor faldet på en *queue*, den opfylder alle ovenstående krav, og vil derfor være en oplagt container, til at håndtere de forskellige messages

Implementation

Implementations-processen var pænt straight forward, da det ovenstående pesdokode er relativ detaljeret og fyldestgørende. Og implemntation af de to klasser ser derfor således ud:

Message.hpp

```
#ifndef MESSAGE_HPP
#define MESSAGE_HPP

class Message
{
    public:
    virtual ~Message(){}
};
#endif
```

MsgQueue.hpp

```
#include "Message.hpp"
#include <queue>

struct msgQueueItem
{
    Message *msg;
    unsigned long id;
};

class MsgQueue
{
    public:
    MsgQueue(unsigned long maxSize)
    {
        maxSize_=maxSize;
        currentSize_=0;
    }

    void send(unsigned long id, Message* msg=nullptr)
    {
        while(currentSize_>=maxSize_)
        {}
        msgQueueItem *itemToAdd=new msgQueueItem;
        itemToAdd->id=id;
        itemToAdd->msg=msg;
        msgQueue_.push(itemToAdd);
        ++currentSize_;
    }

    Message* receive(unsigned long *id)
    {
        while(currentSize_==0){}
        msgQueueItem *item=msgQueue_.front();
        Message *retMsg=item->msg;
        *id=item->id;

        msgQueue_.pop();
        --currentSize_;
        delete item;
        return retMsg;
    }
```

```
    ~MsgQueue()
    {
        //tøm og slet alle elementer i queue.
        while(!msgQueue_.empty())
        {
            delete msgQueue_.front();
            msgQueue_.pop();
        }
    }

    private:
    unsigned long maxSize_;
    unsigned long currentSize_;
    std::queue<msgQueueItem*> msgQueue_;

};
```

Diskussion og Konklusion

Som det kan ses i koden er klassen Message blevet implementeret med en virtuel destructor. Dette er væsentlig fordi destructoren skal tilpasses til den afledte klasse, da den afledte klasse skal sørge for at rydde ordenligt op efter sig selv. Der kan muligvis være dynamisk allokerede objekter i klassen og disse skal slettes korrekt, derfor laves destructoren virtuel.

Som der ser ud nu ser klassen fin ud, men den er ikke blevet testet til det vi gerne vil have den til, nemlig at bruges som værktøj til at kommunikere mellem tråde. Det vil viforsøge i de næste opgaver.

Exercise 2 Sending data from one thread to another

Denne exercise handler om at lave et simpelt program som sender data mellem to threads, ved at bruge de to klasser som blev implementeret i forrige opgaver.

Design

Igen vil der i design-fasen blive lavet pesudokode for overskuelighedens skyld, selvom pesudokode for Sender- og Modtager-threads vil blive relativ simple. Pesudokode for Sender- og Receiver-threads ser derfor således ud:

Pesudokode:

3dPoint

```
//Sender:
while
{
opret besked
send besked
vent 1 sekund.
}

//Modtager:
while
{
```

```
Modtag besked
Handle besked
slet besked
}

//send besked:
tilfoej til msg queue: Point3D
```

Implementation

Ud fra pesudo-koden for sender, modtager og implementeringen af Message og MsgQueue, er det muligt at implementere et program som opretter en message-kø og igangsætter Sender og Modtager, som har kommunikere via message-køen. Sender er sat til at oprette et nyt 3dpunkt hvert sekund, som så vil blive sendt til Modtager, og printer det ud i terminalen.

3dPoint.cpp

```cpp
#include "MsgQueue.hpp"
#include "Message.hpp"
#include <pthread.h>
#include <iostream>
#include <unistd.h>

struct Point3D : public Message
{
    int x;
    int y;
    int z;
};

//Reciever
MsgQueue receiverMsgQueue(10);
pthread_t receiverID;

enum
{
    ID_3D_POINT_IND
};

void receiverHandle3DPointInd(Point3D* ind)
{
    std::cout<<"Koordinater for punkt:\nx: "<<ind->x<<"\ny: "<<ind->y<<"\nz: "<<ind-
}

void receiverHandler(unsigned long id, Message* msg)
{
    switch(id)
    {
        case ID_3D_POINT_IND:
            receiverHandle3DPointInd(
                static_cast<Point3D*>(msg));
            break;
        default:
            break;
    }
```

```
}

void* receiver(void*)
{
    for(;;)//ever
    {
        unsigned long id;
        Message *msg=receiverMsgQueue.receive(&id);
        receiverHandler(id,msg);
        delete msg;
    }
}

//sender
pthread_t senderID;
void* sender(void*)
{

    int x=1;
    int y=1;
    int z=1;
    for(;;)//ever
    {
        x*=2;
        y*=3;
        z*=4;
        Point3D *msg=new Point3D;
        msg->x=x;
        msg->y=y;
        msg->z=z;
        receiverMsgQueue.send(ID_3D_POINT_IND,msg);
        sleep(1);

    }
}
    int main(void)
    {
        pthread_create(&senderID,nullptr,&sender,nullptr);
        pthread_create(&receiverID,nullptr,&receiver,nullptr);
        void *res;
        pthread_join(senderID,&res);
        pthread_join(receiverID,&res);
    }
```

Resultater

Eksekvering af *3dPoint.cpp*

```
stud@stud-virtual-machine:~/i3isu_f2020_beany_business/exe6$ ./bin/host/prog
Koordinater for punkt:
x: 2
y: 3
z: 4
Koordinater for punkt:
x: 4
y: 9
z: 16
Koordinater for punkt:
x: 8
y: 27
z: 64
Koordinater for punkt:
x: 16
y: 81
z: 256
Koordinater for punkt:
x: 32
y: 243
z: 1024
Koordinater for punkt:
x: 64
y: 729
z: 4096
```

Det ses her hvordan forskellige 3d-punkter bliver oprette med forskellige længder i alle tre dimensioner.

Diskussion og Konklusion

Receiveren er ansvarlig for at fjerne beskeder der er blevet handlet(consumed). Det giver mest mening, da det er receiveren der ved hvornår en given besked er blevet handlet. På den måde undgår man også at sende beskeder ud af receiveren om at en besked er blevet handlet, hvilket kunne skabe andre problemer.

Vi har valgt at gøre instansen af klasse MsgQueue global, da dette gør det muligt for trådfunktionerne receiver og sender at tilgå og dermed skrive til MsgQueue.
MsgQueue ejes teknisk set ikke af noget, men i en forstand er det receiveren der 'ejer' den, da det er receiver der primært har brug for funktionaliteten af queuen.

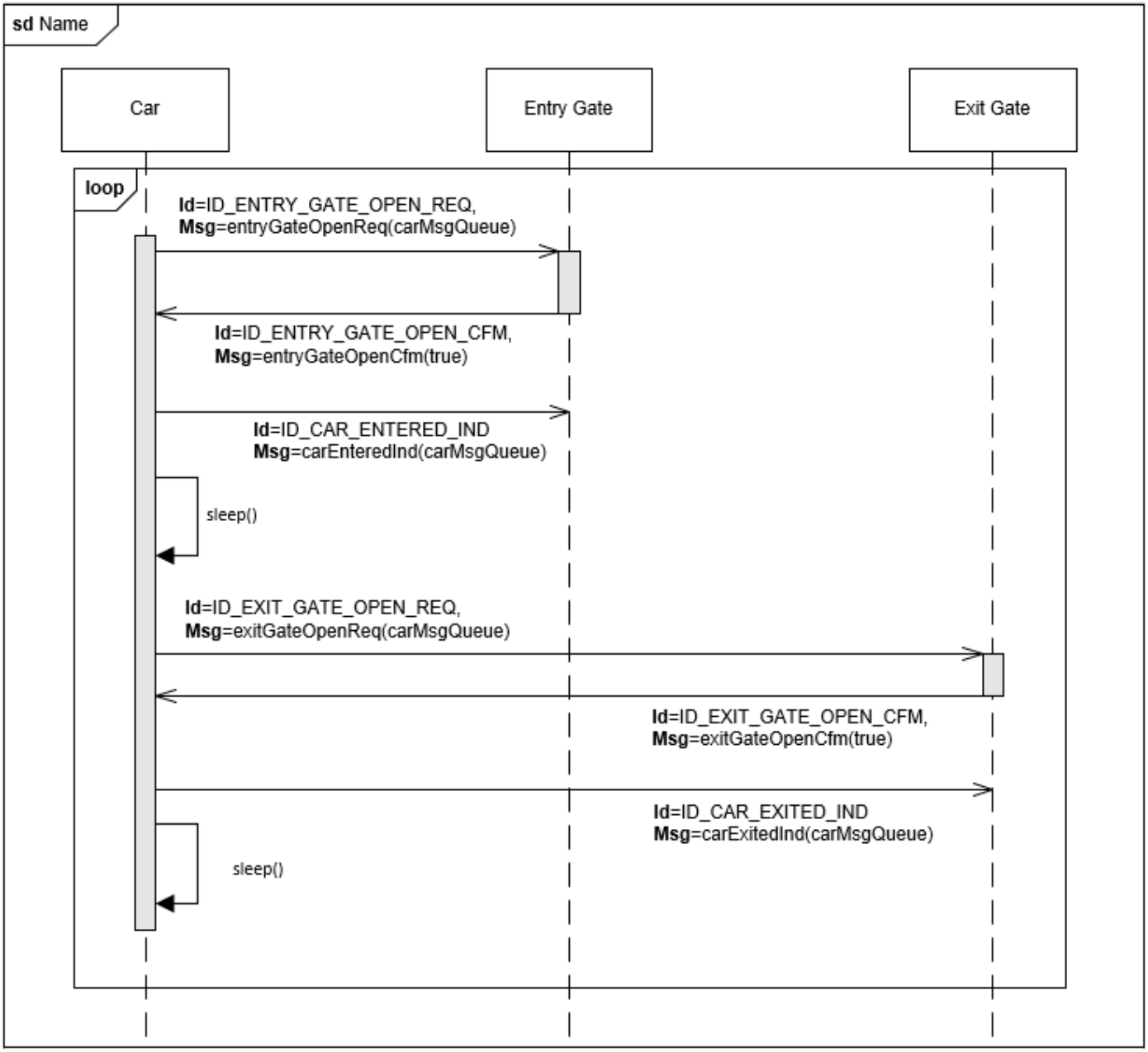
Trådene kender ikke direkte MsgQueue, kendskabet er indirekte gennem funktionerne receiver og sender som bruger MsgQueue klassens funktioner.

Alt i alt fik vi implementeret opgaven og tråde således de kunne kommunikere og opfylde opgaven. Som vi kan se på resultatet lykkedes det at få korrekte udskrifter af forskellige koordinater som var planen.

Exercise 3 Enhancing the PLCS with Message Queues

Denne opgave handler om at erstatte sin *Mutex og Conditions* løsning til Park-a-Lot 2000, men istedet bruge *Message Queues* og herved opnå at programmet kører event-driven.

Design



Systemet består af tre forskellige typer tråde, der kommunikerer med hinanden vha. Hinandens message queues. Disse tre typer er entry gate(1 instans) exit gate(1 instans) og car(n instanser).
Disse tre tråde implementerer hver i sær et loop der modtager beskeder, handler beskeder, ved at kalde dispatcheren med den modtagne besked og ID. Og til sidst slette beskeden.
Dispatcheren står for at reagere på de enkelte beskeder.
For car indebærer dette at køre op til entry gate når bilen startes, samt at køre ud af henholdsvis entry og exit-gate når den modtager en confirmation, om at de er åbne, fra disse.
For entry og exit – gate indeholder handleren metoder til at reagere på en bil der venter på at komme ind/ud, hvis der modtages et car waiting request. Derudover metoder til at lukke gaten, når der modtages en car entered/exited indikator.

Implementation

carPark.cpp

```
#include "MsgQueue.hpp"
#include "Message.hpp"
#include <pthread.h>
#include <iostream>
#include <unistd.h>
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_t entryGateControllerId;
pthread_t exitGateControllerId;
//structs and enums-----------------------------------------------------------
enum{
    ID_ENTRY_GATE_OPEN_CFM,
    ID_EXIT_GATE_OPEN_CFM,
    ID_CAR_START_IND,
    ID_ENTRY_GATE_OPEN_REQ,
    ID_CAR_ENTERED_IND,
    ID_EXIT_GATE_OPEN_REQ,
    ID_CAR_EXITED_IND
};
struct EntryGateOpenReq : public Message
{
    MsgQueue* senderMsgQueue;
};
struct ExitGateOpenReq : public Message
{
    MsgQueue* senderMsgQueue;
};
struct ExitGateOpenCfm:public Message
{
    bool open;
};
struct EntryGateOpenCfm : public Message
{
    bool open;
};
struct CarEnteredInd : public Message
{
};
struct CarExitedInd : public Message
{

};
struct CarStartInd: public Message{};
//entry gate -----------------------------------------------------------------
struct entryGateArg
{
    MsgQueue *entryGateMsgQueue;

};
bool entryGateOpen=false;
bool openEntryGate()
{
   return true;
}
bool closeEntryGate()
{
    return false;
}
void entryGateControllerHandleIDEntryGateOpenReq(EntryGateOpenReq *req)
{
    std::cout << "---EntryGate Opens---"<<std::endl;
    EntryGateOpenCfm *cfm=new EntryGateOpenCfm;
    cfm->open=openEntryGate();
    req->senderMsgQueue->send(ID_ENTRY_GATE_OPEN_CFM,cfm);
}
```

```
void entryGateControllerHandleIDCarEnteredInd(CarEnteredInd *ind)
{
    std::cout << "---EntryGate Closes---"<<std::endl;
    closeEntryGate();
}
void entryGateHandler(unsigned id, Message* msg)
{
    switch(id)
    {
        case ID_ENTRY_GATE_OPEN_REQ:
            entryGateControllerHandleIDEntryGateOpenReq(static_cast<EntryGateOpenReq
        break;
        case ID_CAR_ENTERED_IND:
            entryGateControllerHandleIDCarEnteredInd(static_cast<CarEnteredInd*>(msg
        break;
        default:
        break;
    }
}
void* entryGateFunc(void *arg)
{
    entryGateArg *argu=static_cast<entryGateArg*>(arg);
    MsgQueue *entryGateMsgQueue=argu->entryGateMsgQueue;
    for(;;)//ever
    {
        unsigned long id;
        Message* msg=entryGateMsgQueue->receive(&id);
        entryGateHandler(id,msg);
        delete (msg);
    }
}
void entryGateThread(entryGateArg *arg)
{
    pthread_create(&entryGateControllerId, nullptr, entryGateFunc, arg);
}
//exit gate ------------------------------------------------------------------------
struct exitGateArg
{
    MsgQueue *exitGateMsgQueue;
};
bool exitGateOpen=false;
bool openExitGate()
{
    return true;
}
bool closeExitGate()
{
    return false;
}
void exitGateControllerHandleIDExitGateOpenReq(ExitGateOpenReq *req)
{
    std::cout << "---ExitGate Opens---"<<std::endl;
    ExitGateOpenCfm *cfm_exit = new ExitGateOpenCfm;
    cfm_exit->open = openExitGate();
    req->senderMsgQueue->send(ID_EXIT_GATE_OPEN_CFM,cfm_exit);
}
void exitGateControllerHandleIDCarEnteredInd(CarExitedInd *ind)
{
```

```
    std::cout << "---ExitGate Closes---"<<std::endl;
    closeEntryGate();
}
void exitGateHandler(unsigned id, Message* msg)
{
    switch(id)
    {
        case ID_EXIT_GATE_OPEN_REQ:
            exitGateControllerHandleIDExitGateOpenReq(static_cast<ExitGateOpenReq*>(
        break;
        case ID_CAR_EXITED_IND:
            exitGateControllerHandleIDCarEnteredInd(static_cast<CarExitedInd*>(msg))
        break;
        default:
        break;
    }
}
void* exitGateFunc(void *arg)
{
    exitGateArg *argu=static_cast<exitGateArg*>(arg);
    MsgQueue *exitGateMsgQueue=argu->exitGateMsgQueue;
    for(;;)//ever
    {
        unsigned long id;
        Message* msg=exitGateMsgQueue->receive(&id);
        exitGateHandler(id,msg);
        delete (msg);
    }
}
void exitGateThread(exitGateArg *arg)
{
    pthread_create(&exitGateControllerId, nullptr, exitGateFunc, arg);
}
//---------- Car -----------
struct carInfo
{
    int id_;
    MsgQueue *carControllerMsgQueue;
    MsgQueue *entryGateControllerMsgQueue;
    MsgQueue *exitGateControllerMsgQueue;
};
void carDriveToExitGate(carInfo* Car)
{
    pthread_mutex_lock(&mut);
    std::cout << "Car #"<<Car->id_<<" - drives up to Exit-Gate"<<std::endl;
    pthread_mutex_unlock(&mut);
    ExitGateOpenReq* req = new ExitGateOpenReq;
    req->senderMsgQueue = Car->carControllerMsgQueue;
    Car->exitGateControllerMsgQueue->send(ID_EXIT_GATE_OPEN_REQ, req);
}
void carDriveToEntryGate(carInfo* Car)
{
    pthread_mutex_lock(&mut);
    std::cout << "Car #"<<Car->id_<<" - drives up to Entry-Gate"<<std::endl;
    pthread_mutex_unlock(&mut);
    EntryGateOpenReq* req = new EntryGateOpenReq;
    req->senderMsgQueue = Car->carControllerMsgQueue;
    Car->entryGateControllerMsgQueue->send(ID_ENTRY_GATE_OPEN_REQ, req);
```

```
}
void driveIntoParkingLot(carInfo* Car)
{
    pthread_mutex_lock(&mut);
    std::cout << "Car #"<<Car->id_<< " - drives into parkinglot" << std::endl;
    pthread_mutex_unlock(&mut);
    CarEnteredInd* ind = new CarEnteredInd;
    Car->entryGateControllerMsgQueue->send(ID_CAR_ENTERED_IND, ind);
}
void driveOutOfParkinglot(carInfo* Car)
{
    pthread_mutex_lock(&mut);
    std::cout << "Car #"<<Car->id_<< " - drives out of parkinglot" << std::endl;
    pthread_mutex_unlock(&mut);
    CarExitedInd* ind = new CarExitedInd;
    Car->entryGateControllerMsgQueue->send(ID_CAR_EXITED_IND, ind);
}
void carControllerHandleIDCarStartInd(CarStartInd* ind, carInfo* Car)
{
    carDriveToEntryGate(Car);
}
void carControllerHandleIDEntryGateOpenCfm(EntryGateOpenCfm* cfm, carInfo* Car )
{
    if (cfm->open)
    {
        driveIntoParkingLot(Car);
    }
    sleep(Car->id_*1.5);

    carDriveToExitGate(Car);
}
void carControllerHandleIDExitGateOpenCfm(ExitGateOpenCfm* cfm, carInfo* Car)
{
    if (cfm->open)
    {
        driveOutOfParkinglot(Car);
    }

    sleep(Car->id_*1.5);
    carDriveToEntryGate(Car);
}
void carHandler(unsigned id, Message* msg, carInfo* Car)
{
    switch(id)
    {

        case ID_ENTRY_GATE_OPEN_CFM:
            carControllerHandleIDEntryGateOpenCfm(static_cast<EntryGateOpenCfm*>(msg
        break;
        case ID_EXIT_GATE_OPEN_CFM:
            carControllerHandleIDExitGateOpenCfm(static_cast<ExitGateOpenCfm*>(msg),
        case ID_CAR_START_IND:
            carControllerHandleIDCarStartInd(static_cast<CarStartInd*>(msg),Car);
        break;
        default:
        break;
    }
}
```

```
void* carFunc(void *arg)
{
    carInfo *car  = static_cast<carInfo*>(arg);
    for(;;)
    {
        unsigned long id;
        Message* msg = car->carControllerMsgQueue->receive(&id);
        carHandler(id, msg, car);
        delete(msg);
    }
}
void carThread(carInfo* Car, pthread_t* carThread)
{
    pthread_create(carThread, NULL, carFunc, Car);
    CarStartInd *ind=new CarStartInd;
    Car->carControllerMsgQueue->send(ID_CAR_START_IND,ind);
}
int main(void)
{
    int antal_;
    MsgQueue *entryGateControllerMsgQueue= new MsgQueue(10);
    MsgQueue *exitGateControllerMsgQueue= new MsgQueue(10);
    entryGateArg entryArg;
    exitGateArg exitArg;
    entryArg.entryGateMsgQueue=entryGateControllerMsgQueue;
    exitArg.exitGateMsgQueue=exitGateControllerMsgQueue;
    std::cout<<"Indtast antal biler: ";
    std::cin>>antal_;
    std::cout<<std::endl;
    carInfo cars[antal_];
    pthread_t carThreads[antal_];
    entryGateThread(&entryArg);
    exitGateThread(&exitArg);
    for (int i = 1; i <= antal_; ++i)
    {
        cars[i].entryGateControllerMsgQueue=entryGateControllerMsgQueue;
        cars[i].exitGateControllerMsgQueue=exitGateControllerMsgQueue;
        cars[i].carControllerMsgQueue=new MsgQueue(10);
        cars[i].id_ = i;
        carThread(&cars[i],&carThreads[i]);
    }
    for (int i = 0; i < antal_; i++)
    {
        pthread_join(carThreads[i], nullptr);
    }
    pthread_join(entryGateControllerId, nullptr);
    pthread_join(exitGateControllerId, nullptr);
    return 0;
}
```

Resultater

```
Car #3 - drives into parkinglot
---EntryGate Closes---
Car #1 - drives up to Exit-Gate
---ExitGate Opens---
Car #1 - drives out of parkinglot
Car #1 - drives up to Entry-Gate
Car #1 - drives up to Entry-Gate
---EntryGate Opens---
---EntryGate Opens---
Car #1 - drives into parkinglot
---EntryGate Closes---
Car #2 - drives up to Exit-Gate
---ExitGate Opens---
Car #2 - drives out of parkinglot
Car #1 - drives up to Exit-Gate
```

Ovenstående billede viser et udklip af en kørsel af programmet med 4 biler.
Det skal bemærkes at programmet ikke kører optimalt, da der i nogle tilfælde opstår en segmentation fault. Det er ikke lykkedes at finde frem til hvor denne fejl findes i programmet. Dog er det observeret at programmet kører uden at crashe ved 4 tråde. Altid crasher ved 3 tråde, og nogle gange crasher ved 2 og 8 tråde.
Det er som sagt ikke lykkedes at finde årsagen til dette.
Vi forsøgte at debugge programmet, men ved debugging crashede programmet heller ikke med en kørsel med 3 tråde.

Diskussion og Konklusion

Et event drevet system er et system der, som navnet siger det, er drevet af events. Dvs. at systemet ikke kører sekventielt som meget andet software vi har arbejdet med før. Systemet vil simpelthen 'vente' og konstant tjekke om der er events, og først når et event sker vil der komme en reaktion fra systemet.

Systemet initialisere ved at starte f.eks. et uendeligt for-loop, som tjekker for events. Det er vigtig at objekter og andet der skal initialiseres i programmet er initieret før det uendelig for-loop bliver igangsat, da de ellers aldrig ville blive initieret.

Vi har valgt at implementere det således at der som en del af handleren, der confirmer at entrygaten er åben, efter funktionen driveIntoParkinglot er kaldt også kalder en sleep på den specifikke bil der er blevet lukket ind.

Løsningen med *Mutex/Conditional Message Queue* minder lidt om hinanden, da det er de samme tidspunkter man stopper op og venter på et signal i begge programmer.

En af de største forskelle ved at brug af EDP frem for standard sekventiel er, at selve koden bliver mere overskuelig, netop fordi at kode kan blive opdelt i mindre bidder. Navngivningen af funktionerne og id's kan ydereligere give bedre indsigt i, hvad de bruges til og gør. EDP-kode er derfor nemmere at gennemskue i forhold til hvad der skal ske hvornår.
Derudover kommer der heller ikke samme kritiske sektioner, hvor der bliver brug for mutexes og semaphores. Og af samme grund bliver programmet heller ikke blokeret af vente på en bestemt tilstand/ mutex.

Vi har fået løst opgaverne og er kommet omkring EDP og trådkommunikation via Message Queues. EDP har rigtige mange fordele og det er en tilgang til programmering der er meget 'naturlig', da mange programmer og systemer i høj grad er drevet af events, frem for den lidt mere klassiske tilgang, hvor programmet 'driver' sig selv igennem forskellige processer.

- **Tilføjet af Jim Sørensen for 17 dage siden**

Learning objectives

Learning objective 1: Arbejde med Message-Queue klasser og deres beskeder. (/)

Der er skrevet en forståelig pseudokode før I er begyndt på selve programmeringen.
Derudover har i besvaret fyldestgørende hvorfor I har valgt at bruge data strukturen queue til implementeringen af jeres opgave.
Den viste kode i wiki'en opfylder dog ikke læringsmålene fordi jeg ikke kan se hvor henne I har brugt conditionals/conditional variables, hvilket var et krav. Herudover har I heller ikke brugt mutexes.
En yderligere ting det ser ud som om I ikke har forstået er inheritance og polymorphism eftersom I ikke har tilføjet en "public: Message" efter "class MsgQueue".

Learning objective 2: At sende og modtage beskeder fra den ene tråd til den anden(/)

Gruppen viser, at de kan sende og modtage beskeder ved at sende dem fra den ene tråd til den anden.
De har lavet en fin skabelon af deres system, hvor man kan se, hvad,
der skal implementeres, inden de startede med at lave selve koderne. Den kunne godt være lidt mere uddybende, efter min mening.

Learning objective 3: At kunne oprette et event-loop og bruge dispatcheren rigtigt (/)

Gruppen har lavet et fint Sekvens diagram og de beskriver også opgaven fint. Det man kunne savne var at
det var flere kommentar i koden som er lagt ud på wikien men ellers ser det hele i denne opgave fint ud.

Learning objective 4: At kunne lave et sekvensdiagram over et PLCS-system (/)

Gruppen har lavet deres sekvensdiagram, og tilpasset implementationen efter den.

Feedback:

Det ser ud som om I ikke har læst opgaven grundigt igennem før I begyndte på den eller at I har haft lidt for travlt.
Der er væsentlige ting, som mangler i læringsmålene. Til næste lab exercise, ville det være fornuftigt at sætte sig
sammen og være sikker på at alle har forstået og læst hele opgaven grundigt igennem.

Must haves

*Relevant files in repository (Er ikke opfyldt)
*Makefiles in repository (Er ikke opfyldt)

Makefilen ser overskueligt ud. Man kan se, at de også har styr på den del.

Conclusion

Årsagen til at Must haves ikke er 100% opfyldt er fordi det ser ud til at I kun har lagt filer op til opgave 3 i repository.
Dvs. der mangler Makefile og c filer til opg 1 og 2. Dette vil altså gøre at opgaven bliver til external review.
Andre årsager til den lave bedømmelse kan ses i Feedback og læringsmålene ovenover.

opg_2_koordinater.png (27,6 KB) Magnus Nygaard Lund, 2020-03-23 13:17

opg_3_sekvens.png (25,4 KB) Magnus Nygaard Lund, 2020-03-28 15:15

isu.png (16,6 KB) Jens Nørby Kristensen, 2020-03-29 18:31