Wiki »

Exercise6 Inter Thread Communication ¶

Vi vil igennem denne øvelse kreere et system hvor to tråde kan agere sender/modtager ved brug af en besked kø, og til sidst lave en opgraderet udgave af den tidligere implementerede PLCS.

Exercise 1 Creating a message queue

Part 1 - Design

Som kickstarter til vores implementering af Message Queue har vi startet med at udforme pseudo-kode for hvordan implementeringen kunne se ud i grove træk. Pseudo-koden kan ses nedenfor:

Klassen Message Queue:

```
class MsgQueue
{
    public:
        Constructor()
        Send()
        Receive()
        Destructor()
    private:
        struct{}
        queue<struct> msgq
        mutex
        senderCondition
        receiverCondition

}
```

Constructor:

```
MsgQueue{

    MaxSize_ = MaxSize
    mutex init
    cond init send()
    cond init receive()

    };
```

Destructor:

```
~MsgQueue{
```

```
    };
```

Sender:

```
Send()
{
    Lock()

    while(Queue == full)
    {
        wait()
    }

    Create Struct Object
    give ID and MSG

    if(Queue == 1)
    {
        signal(condition)
    }
    Unlock()
}
```

Receiver:

```
Receive()
{
    Lock()

    Create Struct Object

    while(Queue == empty)
    {
        wait()
    }

    POP to msgQueue

    Create Struct Object
    give ID and MSG

    if(Queue == notFull)
    {
        signal(condition);
    }

    Unlock();

    return frontMessage;
}
```

Why is it important that the destructor in the Message class is virtual?

- Det er vigtigt at destructoren i Message klassen er virtuel, så at de nedarvede klasser bruger den rigtige destructor.

Part 2 - Implementation

Valg af STL container:

- Vi har valgt at gøre brug af en Queue, da den prioriterer alt efter hvilket objekt der først bliver lagt ind i MessageQueue, hvilket giver mening ift. det vi skal implementere.

```
#include <iostream>
#include <pthread.h>
#include <mutex>
#include <unistd.h>
#include <thread>
#include <queue>

using namespace std;

pthread_cond_t qu_empty_cond = PTHREAD_COND_INITIALIZER;
pthread_cond_t qu_full_cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qu_mutex = PTHREAD_MUTEX_INITIALIZER;

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

};

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

    queue <Data> msgq;
    unsigned long MaxSize_;
    pthread_mutex_t mut;
    pthread_cond_t senderCond;
    pthread_cond_t receiverCond;
};

MsgQueue::MsgQueue(unsigned long MaxSize)
{
    MaxSize_ = MaxSize;
```

```
    pthread_mutex_init(&mut, NULL);
    pthread_cond_init(&senderCond, NULL);
    pthread_cond_init(&receiverCond, NULL);
}

void MsgQueue::send(unsigned long id, Message* msg)
{
  pthread_mutex_lock(&mut);
        while(msgq.size() == MaxSize_)
        {
            pthread_cond_wait(&senderCond, &mut);
        }
        Data temp;
        temp.id_=id;
        temp.msg_=msg;
        msgq.push(temp);

        if(msgq.size()==1){
          pthread_cond_signal(&receiverCond);
        }
    pthread_mutex_unlock(&mut);
}

Message* MsgQueue::receive(unsigned long &id){

  pthread_mutex_lock(&mut);
  Data temp;

    while(msgq.size()==0){
      pthread_cond_wait(&receiverCond,&mut);
    }

    temp=msgq.front();
    msgq.pop();
    if(msgq.size()==MaxSize_-1){
      pthread_cond_signal(&senderCond);
    }

  pthread_mutex_unlock(&mut);
  return temp*;
}

MsgQueue::~MsgQueue(){

}
```

Exercise 2 Sending data from one thread to another

I denne opgave skal vi sende data fra en tråd til en anden. Til at udføre dette bruger vi i dette tilfælde nogle koordinater fra en *struct* vi har oprettet, som var givet af opgaveformuleringen.
Implementering skulle bestå af to tråde, som hhv. skulle sende og modtage, som til sidst tester i et lille testprogram. Implementeringen kan ses nedenfor:

```
#include "msgqueue.hpp"
```

```
using namespace std;

const int MAXSIZE=10;
MsgQueue mq(MAXSIZE);

enum{
  ID_PING
};

struct point3d:public Message{
  int x;
  int y;
  int z;
};

void handlerid(point3d* p){
  cout << "  -------" << endl;
  cout << " | " << "x = " << p->x << " |"<< endl;
  cout << " | " << "y = " << p->y << " |" << endl;
  cout << " | " << "z = " << p->z << " |" << endl;
  cout << "  -------" << endl;
}

void handler(unsigned long id, Message* msg){
  switch(id){
    case(ID_PING):
      handlerid(static_cast<point3d*>(msg));
      break;

      default:
      break;
  }
}

void* receiver(void*){
  for(;;){
    unsigned long id;
    Message* msg = mq.receive(id);
    handler(id,msg);
    delete msg;
  }
}

void* sender(void*){
  int x = 0;
  int y = 0;
  int z = 0;
  for(;;){
    sleep(1);
    x = rand()%10;
    y = rand()%10;
    z = rand()%10;

    point3d* p = new point3d;
    p->x=x;
    p->y=y;
    p->z=z;
    mq.send(ID_PING,p);
```

```
  }
}

int main(){
  pthread_t t1,t2;

  pthread_create(&t1,nullptr,receiver,nullptr);
  pthread_create(&t2,nullptr,sender,nullptr);

  pthread_join(t1,nullptr);
  pthread_join(t2,nullptr);

  return 0;
}
```

Test af systemet:

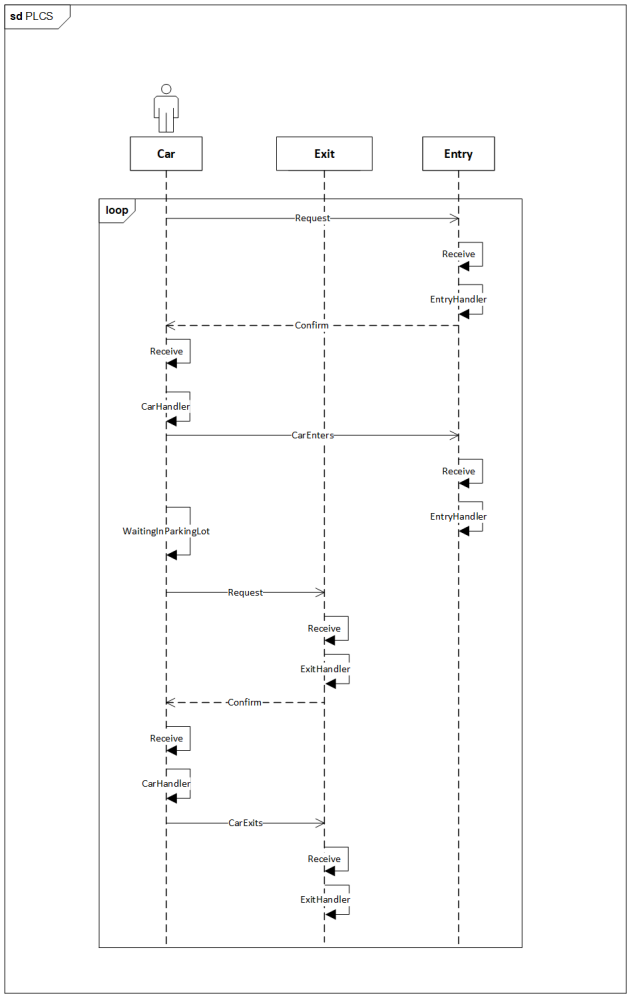
Nedenfor ses et billede ved kørsel af programmet. Det ses at modtagertråden modtager og håndterer de koordinater der sendes, som skrives ud i terminalen.

Questions to answer:

- **Who is responsible for disposing any given message?**
  Det er modtagertråden der sletter den modtagne besked. Ud fra princippet om at vi skal "Receive -> Handle -> Delete". Selvom det ville give mest mening at den som opretter beskeden burde slette den igen.
- **Who should be the owner of the object instance of class MsgQueue; Is it relevant in this particular scenario?**
  Senderen som opretter objektet burde være ejeren af det oprettede objekt og nedlægge det igen, men i dette tilfælde er det modtageren som ender med at nedlægge dette. I vores tilfælde er dette meget relevant, da vi ellers skulle sende en besked fra modtager til sender om at beskeden er håndteret og derefter skulle sender nedlægge beskeden og så vil vi køre i ring i programmet.
- **How are the threads brought to know about the object instance of MsgQueue?**
  Trådene i programmet informeres om de oprettede objekter ved brug af en besked handler, som videregiver informationer.

Exercise 3 Enhancing the PLCS with Message Queues

Vi har udformet følgende sekvensdiagram, som viser den overordnede funktionalitet for PLCS.



Vi har forsøgt i mange timer at få vores opgraderede udgave af PLCS til at kunne fungere, men desværre uden held. Enkelte funktionaliteter fungerer, og programmet har kunne køre til dels, men vi endte tit ud i at støde på en segmenterings fejl. Den udarbejdede kode til implementeringen ud fra sekvensdiagrammet af exercise 3 kan findes i vores repository, og på dette link: Link til exercise 3

Vi har således svaret på spørgsmålene til denne øvelse til bedste evne, ud fra at vi manglede den sidste del af implementationen for at komme i mål med denne.

Questions to answer:

- **What is an event driven system?**
  Dette er et system der aktiveres af en hændelse, modsat fx polling. Det kunne være en sensor i et system, der ved ændring sender en besked videre og derved aktiverer en/flere handlinger i et system
- **How and where do you start this event driven system?**
  Det event drevne system startes når beskedkøen, messageQueue, ikke er tom
- **Explain your design choice in the specific situation where a given car is parked inside the carpark and waiting before leaving - How is the waiting situation handled?**
  Det kunne have været implementeret med en sleep() men grundet vi ikke fik det til at virke er det ikke blevet implementeret.
- **Why is it important that each *car* has its own MsgQueue?**
  Hver bil skal have sin egen messagequeue, så den kan holde styr på hvor den er henne i parkeringskælderen. Hvis de deler en messagequeue kan det være at en bil som venter på at køre ind lige pludselig kører ud af parkeringskælderen, altså de har den samme ID, derfor skal de være tilknyttet hvert sit objekt.
- **Compare the original *Mutex/Conditional* solution to the *Message Queue* solution.**
  - In which ways do they resemble each other? Consider the stepwise differences.
  istedet for at der signaleres en conditional, som står og venter, så reageres der på handlinger som lægges ind i en kø.
  - What do you consider to be the most important benefit achieved by using the EDP approach?
  Den vigtigste fordel ved event driven programming er at man ikke skal holde styr på at mutexes bliver

unlocked efter den er blevet locked og derfor opstår der færre fejl fra programmørens side. Derved er der mindre risiko for deadlocks

SDexe3.png (40 KB) Benjamin Vølund, 2019-04-02 18:19

exe2test.png (16,1 KB) Benjamin Vølund, 2019-04-02 18:38