Inter Thread Communication

Søren Hansen <sha@ase.au.dk>
V1.5

Introduction

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.

These are key stone elements in Event Driven Programming which is very typical for embedded systems.

There are numerous design considerations that you will be exposed to, thus widening your toolbox.

Prerequisites

You must know:

- What inheritance is and how it is used.
- What polymorphism is and how it is used.
- During class construction and destruction; What happens when.

It is very important that you have understanding of the above subjects, otherwise completing this exercise will be very difficult

Absolute requirements (Must haves for approval)

- The special `while` loop that must be in every thread function.
- State or sequence diagrams denoted as *must*
- Conditionals *must* be used for synchronization (E.g. no semaphores).
- Message handlers may not be part of the `MsqQueue` interface/implementation.
- Each guard and each car must have their own thread and their own message queue.

Before you start each and every exercise, do yourself a favour and read the WHOLE text.
Secondly ***"Plan your design before you code!!!"***

Exercise 1 Creating a message queue

To have something tangible for our upcoming message queue, we start out creating the class `Message`. This class will from now on serve as the basis (parent) of all messages that are passed around in our system. In other words all other messages must inherit from this class. Remember that the destructor must be virtual.
Why is this last bit very important? Explain!

Listing 1.1: The class Message

```
1  class Message
2  {
3  public:
4    virtual ~Message() {}
5  };
```

Next we create the `MsgQueue` class itself. One of its aggregated variables is a container; this container is the one that will hold all incoming messages, before the receiving thread processes them one at a time.

We must remember that the usage scenario is one where we have *multiple writers* and a *single reader*, which is why appropriate protection is vital. This protection *must* be implemented via the use of *conditionals*. Furthermore, as specified in the constructor call, a limit is set on the number of messages in our queue. Handling this limitation is likewise to be achieved using the use of the aforementioned conditionals.

Listing 1.2: The class MsgQueue

```
1  class MsgQueue
2  {
3  public:
4    MsgQueue(unsigned long maxSize);
5    void send(unsigned long id, Message* msg = NULL);
6    Message* receive(unsigned long& id);
7    ~MsgQueue();
8  private:
9    // Container with messages
10   // Plus other relevant variables
11 };
```

Further demands:

- `send()` is blocking if the internal queue is filled to the maximum denoted capacity.
- `receive()` is likewise blocking if the queue is empty.

As stated earlier the incoming message must be placed in a container, which one to choose? Check your nearest STL vendor and see what he believes would be the right choice...

Before you google for the *container to use* think about what you specifically need. A hint; `www.cplusplus.com` can provide you with guidance.

Please do note that there are numerous code examples of STL container uses on the net, so JFGI.

As a last note: *The class interface for `MsgQueue` in regard to methods is complete!* This means you are not to add any yourselves! Your message handler is *NOT* part of this class nor are you supposed to sub-class it. Your message handler is a simple function that is called from your thread function... For more information see presentation.

Exercise 2 Sending data from one thread to another

Create a C++ `struct` named `Point3D` as shown in 2.1.

Listing 2.1: struct Point3D

```
1 struct Point3D : public Message
2 {
3   int x;
4   int y;
5   int z;
6 };
```

Create two threads *Sender* and *Receiver*, where 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.

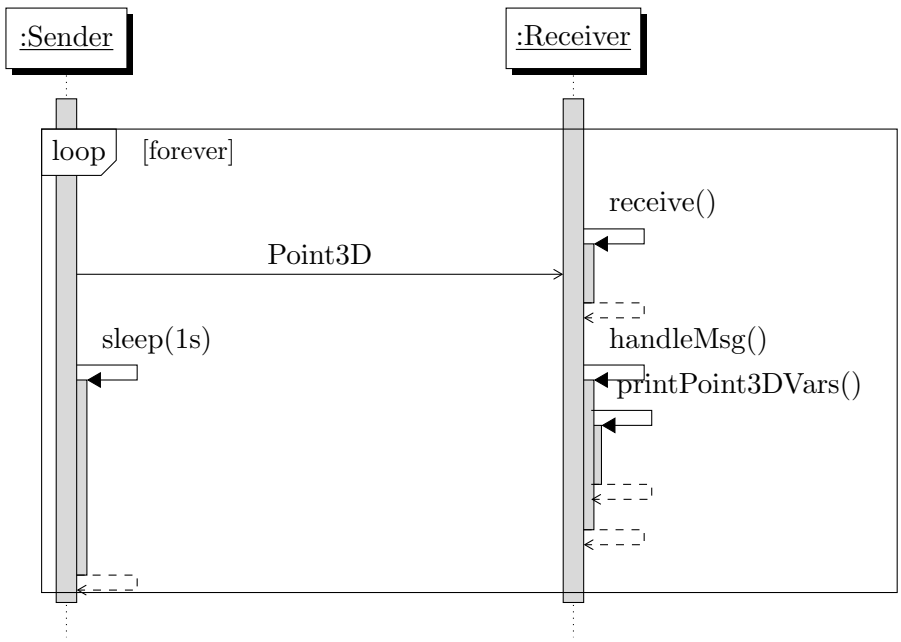

Figure 2.1: The *Sender* sends a `Point3D` object to the *Receiver*

Create a function `main()` that creates a message queue and then spawns the two said threads running *Sender* and *Receiver* respectively that communicate via this message queue.

Test your system - does the receiver thread receive what the sender sends?

Questions to answer:

- Who is responsible for disposing any given message?
- Who should be the owner of the object instance of class `MsgQueue`; Is it relevant in this particular scenario?
- How are the threads brought to know about the object instance of `MsgQueue`?

Exercise 3 Enhancing the PLCS with Message Queues

Reimplement your PLCS solution using Message Queues and thus messages as a means to communicate between car thread(s) and door controller thread.

The following are *must have* requirement for your is solution:

- A thread for the entry guard, one for the exit guard and for each car instance.
- Each thread's thread function, named `threadFunction()` here, ***must*** resemble the code example shown in Listing 3.1. This means keeping to the same structure, but having the freedom to add extra input parameters to say the function `handleMsg()`.
- Your design and implementation must be able to handle multiple cars, but it is optional whether there is a maximum amount allowed in at any given time.

Before you start any coding what so ever, you ***must*** either create a sequence or a state chart diagram! This diagram *must* be depicted in your solution, and you must state why you chose this particular diagram.

Listing 3.1: Loop in thread function

```
1  void threadFunction()
2  {
3    /* Initialization code, if needed */
4    while(running)
5    {
6      unsigned long id;
7      Message* msg = mq.receive(id);
8      handleMsg(msg, id);
9      delete msg;
10   }
11
12   /* Clean up, if needed */
13 }
```

Questions to answer:

- What is an event driven system?
- How and where do you start this event driven system? (remember it is purely reactive!)
- 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?
- Why is it important that each *car* has its own `MsgQueue`?
- 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*.
  - What do you consider to be the most important benefit achieved by using the EDP approach, elaborate.