Inter Thread Communication

shan@iha.dk
V1.0

Introduction

In this exercise you will learn the basics of thread communication using the message queue 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 it in a given thread.

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 subject, otherwise completing this exercise will be quite difficult

Goal

When you have completed this exercise, you will:

- have gained some proficiency in the declaration and use of message queues using Posix synchronization elements.
- have improved understanding of the a-synchronicity challenge when dealing with messages being send between threads.
- have gained insight into the challenge of data ownership; Who allocates and who deallocates, not necessarily obvious.

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 the message queue to handle, we start out creating 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 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 handled via the use of conditionals. Furthermore, as specified in the constructor call, a limit is set on the number of messages in our queue. Again this is to be handled as well via the use of the aforementioned conditionals.

Listing 2 The class MsgQueue

```
1  class MsgQueue
2  {
3  public:
4    MsgQueue(size_t maxSize);
5    void send(size_t id, Message* msg = NULL);
6    Message* receive(size_t& 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.cpluscplus.com` can provide you with guidance. Please do note that there are numerous code examples of STL container uses on the net, so JFGI.

Exercise 2 Sending data from one thread to another

Create a C++ `struct` named `Point3D` that contains `x`, `y` and `z` values (all integers). Then create two thread functions, `sender()` and `receiver()` hence known as the `sender thread` and `receiver thread`. The `sender thread` should send a `Point3D` object to the `receiver thread` via a message queue every second or so. The `receiver thread` should continuously wait for new messages and, on reception, print the `x`, `y`, and `z` coordinates of the received `Point3D` object to the console.

Create a function `main()` that creates a message queue and then spawns two 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 consider:

- Who is supposed to be responsible for disposing of the incoming messages?
- 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`?
- How does the `struct Point3D` relate to `class Message` in your particular design?
  - By inheritance
  - By composition or association

  Elaborate on your design choice.

Exercise 3 Enhancing the PCLS with Message Queues

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

HINT: In each thread consider using a simple state machine, that controls the communication between the threads. This is especially beneficial due to the asynchronous nature of the system.

How do you handle state machines?