UD1414 — Assignment 1
Pipeline tools communication

Deadline	30th of October 23:55
Submission	Individual Zip file uploaded to Its Learning
Grading	U, G

1 Description

Sharing information between applications is very important in the process of game development. This could be done offline, sharing models, texture files, etc., or online in real-time using any form of interprocess communication available in the Operating System or even between computers through the network.

In this assignment, you will write a solution in C++ to share pieces of information (messages) between one producer and many consumers. The code developed will be subsequently used in the final project to build a Maya plugin that communicates with an external realtime scene viewer.

2 Detailed description

You have to develop a single application/program, that will behave differently depending on the arguments provided when executing it. This will be achieved through command line options. There are two roles in this problem, **producer** and **consumer**. One producer and many consumers will communicate through a shared memory area, using Windows filemaps [1], and they will coordinate the access to this shared resource using a Windows Mutex [2]. In order to allow for two or more processes to communicate, we must establish some rules, we call these rules the **communication protocol**. The rules will define when a process is allowed to read/write in specific areas of the shared space and what actions should be taken after reading or writing. If the protocol is well designed and thoroughly thought then the applications (producers and consumers) will communicate without problems.

2.1 Implementation details

1. Before starting to code, devise carefully the protocol that each party in the communication has to follow.

2. There will be one producer, and many consumers. We assume a fixed amount of space in the shared memory, so memory has to be reused once data has been produced and consumed by all consumers. Only when all consumers have consumed the same message, the producer is allowed to forget about this message and reuse this space.

3. A message can have any size smaller than a quarter of the total shared memory area. This implies for example that if the shared memory is of 8 Megabytes, we will only be able to send messages up to 2 Megabytes. To put things in perspective, we will probably use over 200 Megabytes for the shared memory size to be able to send very big messages to the consumers.

4. All messages, whatever size they have, will be "padded" with extra data to always be multiple of 256. So if a message is 560 bytes, then it will be padded with 208 bytes so the total size is 768 (3*256). We will not explicitly need to create this padding data, we will simply say "the message starts in memory at certain position, has certain length, and certain amount of padding", so we know exactly where the next message starts and ends.

5. The producer can generate messages of different size, but all messages have a fixed size header which contains the following fields:

   (a) A unique message Id.

   (b) Message length bytes (including the header size, and multiple of 256)

   (c) Bytes of padding. The amount of data added after the real content to fill the message size.

   For example, a message could be of length 256 bytes, with a 32 bytes header, 20 bytes of actual content, and the rest just padding (filling the gap with garbage until 256).

   The learned reader must have observed, that if messages are always a multiple of 256 bytes we only need to know the real message length, as we can derive how much padding would be needed to reach the next multiple of 256.

   See Figure 1 for details.

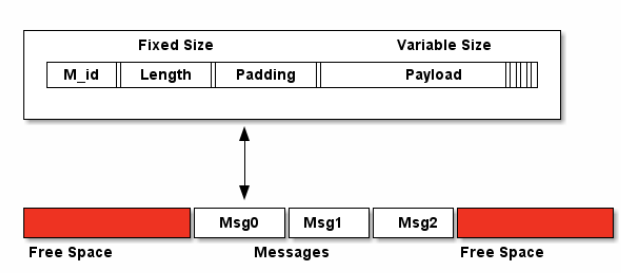

Figure 1: Message structure

6. The producer and consumer applications have to allow for a delay (sleeping/waiting time doing nothing) specified in the command line. Also, the

consumers, as a rule, have to make a local copy of the message before working with its contents, to allow for other applications to access the shared memory.

2.2 Command line arguments and behavior

The executable you have to produce will be named "shared.exe", and it will always expect the following command line specification (for simplicity arguments are *not* optional):

> **shared.exe producer|consumer delay memorySize numMessages random|msgSize**

- **producer|consumer**: Defines the role of the program
- **delay**: Delay in milliseconds after processing each message
- **memorySize**: Size of the shared memory in megabytes, not in bytes. (File map)
- **numMessages**: Number of messages to produce and consume.
- **random|msgSize**: The word **random** to indicate that the size for the messages can vary during one single execution, or msgSize to make all messages of a given size in Bytes.

Note that the arguments have to follow this exact order, and all have to be present in the command line. The following command line examples will start one producer and two consumers, where the producer is quicker than the consumers at processing messages. This example will stress the test scenario where the writer has to wait for the slower consumers all the time.

>shared.exe producer 1 50 10000 random

>shared.exe consumer 2 50 10000 random

>shared.exe consumer 3 50 10000 random

Note that the last 3 arguments are always the same for the consumers and producers for a particular test. When the producer sees "'random"', it will generate random sizes of messages within the allowed sizes whilst the consumer will ignore the option because the producer is really who defines the size of the messages.

3 Program output and testing

In practice, the message content can be anything, but for testing purposes we will use strings as content. A small function to generate random null-terminated strings will be provided. The producer will create messages, that will contain

random strings, and the consumer will read those messages and print them out to the screen.

A Python script will be provided to test different scenarios running your executable. With this script you will be able to test your solution before the submission. For this script to work, the output of your program has follow this specification in a strictly manner.

Let us put some emphasis here, *the output of your program has to follow this specification in a strictly manner*.

When running without any DEBUG output, the program should only output, for each message (sent or received):

- The message id (this could be a number)
- A null terminated random string contained in the message
- a newline character

For example, if we execute the following two programs in separate terminals (one producer, one consumer):

>shared.exe producer 0 1 2 1024
>shared.exe consumer 0 1 2 1024

The producer will produce two messages, of 1024 bytes, where each message will contain a random null-terminated string that fits in the message.

The producer will print (for example):
1 asdfqwe
2 lkjasoiueroisuer

The consumer HAS to print
1 asdfqwe
2 lkjasoiueroisuer

4 Submission and grading

The assignment will be done and assessed individually, and it has to be submitted to the ITsLearning webpage for the course. Before creating a submission (zip) file, clean the solution in Visual Studio so only the source code and project details are included. To obtain a grading pass, your solution has to pass all the tests defined in the Python script.

References

[1] Windows File Mapping. `http://goo.gl/w1xJgW`

[2] Using Mutex Objects. http://goo.gl/fZlxA0

[3] Circular Buffers. http://goo.gl/uRBUCo

[4] Useful notes on C pointers. http://goo.gl/17VTZ0