Examples on the use of shared memory to share acquired images between C# and Python-based processes

https://github.com/hcostelha/chsarp_python_cv_sharedmem

Author: Hugo Costelha <hugo.costelha@ipleiria.pt>
Version: 1.0 – 16/12/2021

1 Introduction

The provided examples implement "Producer" and "Consumers". He producers are responsible for acquiring images from a camera (using OpenCV) and sharing them using shared memory. The consumers access this shared memory to have access to the camera images.

The producers are available in C#, while the consumers are available in C# and Python. Both the C# consumers and producers are available with an EMGU and OpenCVSharp-based implementations.

The producer creates a shared memory space to shared image information and the images themselves. Depending on the number of processes (currently fixed through a variable definition in the producer code), it creates several slots (2 more than the expected number of processes) to share the images. This approach allows the consumers to work at their maximum frames per second, without depending on the other consumers framerate. If a given slot/image is being used by one or more consumers, at least one slot will be free to store a new acquired image. In order to prevent the consumers to have to continuously pool a shared memory variable to determine that a new image is available, a named pipe is used between the producer and each consumer to trigger/signal that event. Using a camera at 30 FPS allowed the producer and all the consumers to work at 30 FPS, even if we limit one particular consumer to a smaller number of FPS.

The choice of shared memory was due to performance reasons. You can see a brief comparison in this link. This approach is portable, so it can be used at least in both Windows and Linux. The used of the named pipes for the triggering was also due to performance reasons and the fact that we can have a blocking read operation to determine that we have received a trigger for a new available image.

The implementation is sharing an BGR image for debugging purposes only and multiple grayscale images to be used by the consumers. The code needs to be adapted if the consumers need access to the BGR image.

Both the producer and the consumers use a named mutex to guarantee that reading and writing operations do not collide. The application assumes that no changes are done by the consumers to the shared image buffers.

2 Repository description

The C# projects were developed using Visual Studio, while the Pyhon projects were developed using Visual Studio Code. A solution was used for both EMGU and OpenCVSharp producers, and a single solucion was also used for the EMGU and OpenCVShartp consumers. The github repository is organized as described in

Folder	Description
CSharpAppConsumer	Contains the Visual Studio C# consumer solution and the EMGU consumer implementation project.
CSharpAppConsumerOpenCVSharp	Contains the Visual Studio C# consumer project based on OpenCVSharp.
CSharpAppProducer	Contains the Visual Studio C# producer solution and the EMGU producer implementation project.
OpenCVSharpProducer	Contains the Visual Studio C# producer project based on OpenCVSharp.

PythonAppConsumer	Contains the Python-based consumer implementation.
doc	This documentation.

Table 1 – Github folders description

3 Functional description

When the producer is started, it immediately creates a named pipe server and starts acquiring images and putting them in the shared image buffers, even of no consumer is accessing to the shared memory. As soon as a consumer connects to the named pipe, the producer starts sending the trigger to that consumer. Recall that the number of consumers is fixed at compile time through the NUM_PROCESSES variable in the producer's code. Note also that each consumer should connect to its one named pipe end connection with the specific name '//./pipe{n}' (where {n} is to be replaced by the consumer index). Figure 1 and Figure 2 show, respectively, the flowchart for the producer process and the flowchart for the consumer processes.

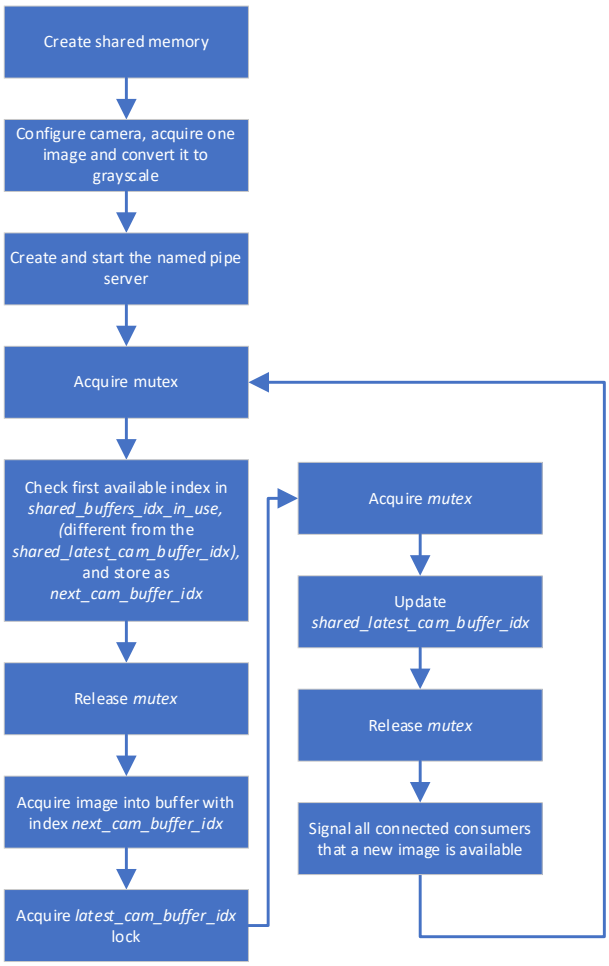

Figure 1 – Producer flowchart.

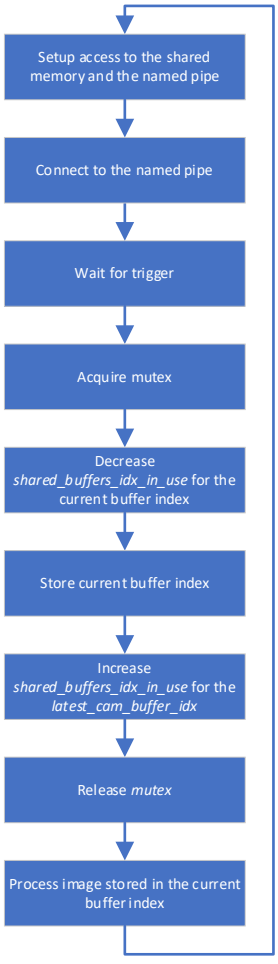

Figure 2 – Consumers' flowchart.

3.1 Relevant variables

In terms of relevant variables, the consumer and each producer have the named mutex, the named pipe, and the data stored in the shared memory, with this later one described in more detail in Table 2.

Variable	Type [C#]	Size [Bytes]	Description
shared_frame_height	int	4	Acquired image height.
shared_frame_width	int	4	Acquired image width.

shared_gray_frame_step	int	4	Image step per row.
num_frame_buffers	byte	1	Number of shared image buffers (equal to the number of processes plus 2).
shared_latest_cam_buffer_idx	byte	1	Index of the shared buffer with the latest camera image.
shared_buffers_idx_in_use	byte[]	num_frame_buffers	Array with the number of consumers using each shared image buffer.
bgr_image	byte[]	*shared_frame_height x shared_frame_width x 3*	Last acquired BGR image. (Should not be used by the consumers – for debugging purposes only).
gray_image[0]	byte[]	*shared_frame_height x shared_frame_width*	Grayscale image (buffer 0).
gray_image[1]	byte[]	*shared_frame_height x shared_frame_width*	Grayscale image (buffer 1).
⋮	⋮	⋮	⋮
gray_image[num_frame_buffers-1]	byte[]	*shared_frame_height x shared_frame_width*	Grayscale image (buffer num_frame_buffers-1).

Table 2 – Data stored in the shared memory.

Regarding the date stored in the shared memory, the variables `shared_frame_height`, `shared_frame_width`, `shared_gray_frame_step` and `num_frame_buffers` are initially written by the producer and do not change throughout the application lifetime. The `shared_latest_cam_buffer_idx`, `bgr_image` and `gray_image[n]` images are continuously updated by the producer and read by the consumers (this must not be changed by the consumers). The `shared_buffers_idx_in_use` is continuously updated by the consumers and read by the producer.

4 Usage

To test both the produces and consumers, make sure you have access to the EXE files by compiling the intended projects first. Open the two solutions in Visual Studio (producer and consumer) and the Python-based consumer in Visual Studio code (for instance). To compile the EMGU implementations, make sure you install Emgu.CV and Emgu-CV.runtime.windows using nuget. To compile the OpenCVSharp implementation, install the Windows and the runtime using nuget.

Run the producer (either the EMGU or the OpenCVSharp-based implementation). The start the consumers. The producer is expecting 3 consumers, so you can start, for instance, 1 C#-based consumer and the Python-based consumer (the Python-based consumer actually starts 2 consumers). If you enable the DEBUG_WINDOWS flags you will see several windows with the acquired images. If you enable the DEBUG_FPS flag, you will see the estimated FPS.