Data Acquisition and Convolution System

Objective

Design a Scalable solution to Generate blocks which is loosely coupled with the client system. For enabling Data access across each Blocks in the system, They should communicate with each other. We should implement a fast and reliable communication medium available across the blocks.

Goals

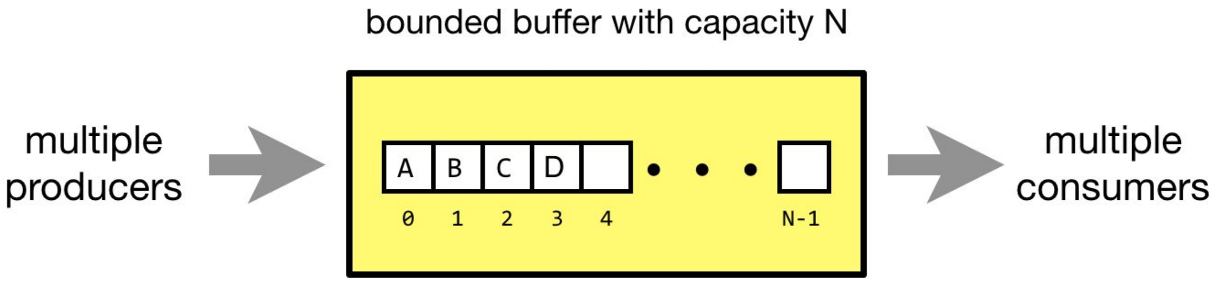
Each of the blocks can have similar structure, and provides a core operation to the client. Thus the creation of blocks follow a general pattern. The blocks should be loosely coupled with the client application which uses them, since the addition of a block should not force any change in client application.

Blocks should communicate with each other using a shared queue.
This Queue should be created only as a single instance and maintain it during a complete program lifecycle. The shared queue can be written by any producer block and read by the consumer block which has access.

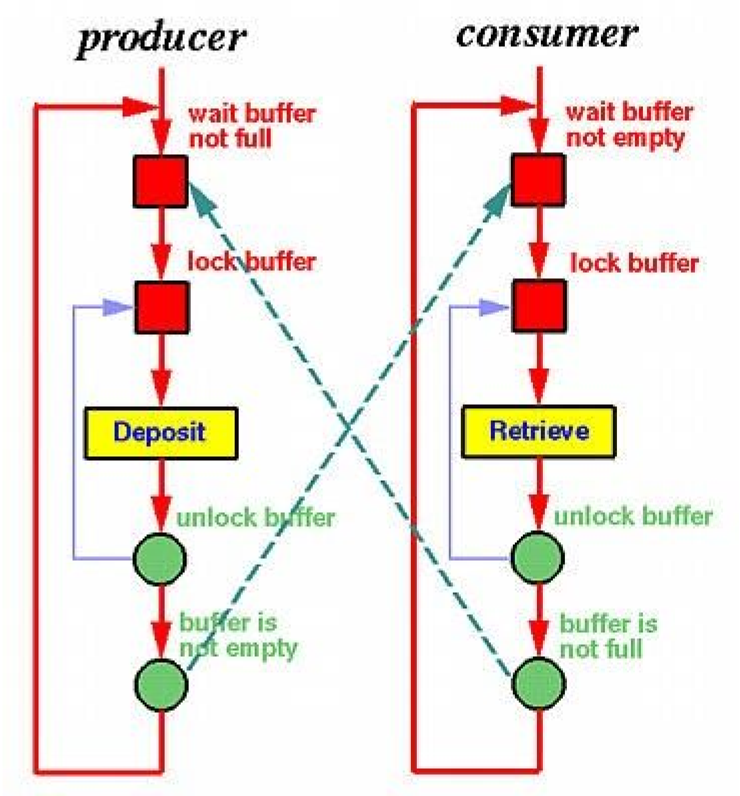
Proposed Solution

Implement Block creation using factory method pattern, Thus providing a loosely coupled architecture between blocks and the client.
We propose a BlockFactory class which has a CreateBlock Method which can create new instances of DataGenerationBlock and DataInterpretationBlock initially, and can be scaled later on based on the requirement of new blocks.

For establishing communication between these independent threads/processes created by blocks we should use an operating system shared resource. As Operating system's shared memory is considered fast among all 2 way IPC mechanisms and available across processes, we can use this for our queue storage.



One thing which we should keep in mind is the synchronization inside this queue. Multiple processes can read/write this queue at the same time. Causing a risk of raise condition at any instance.

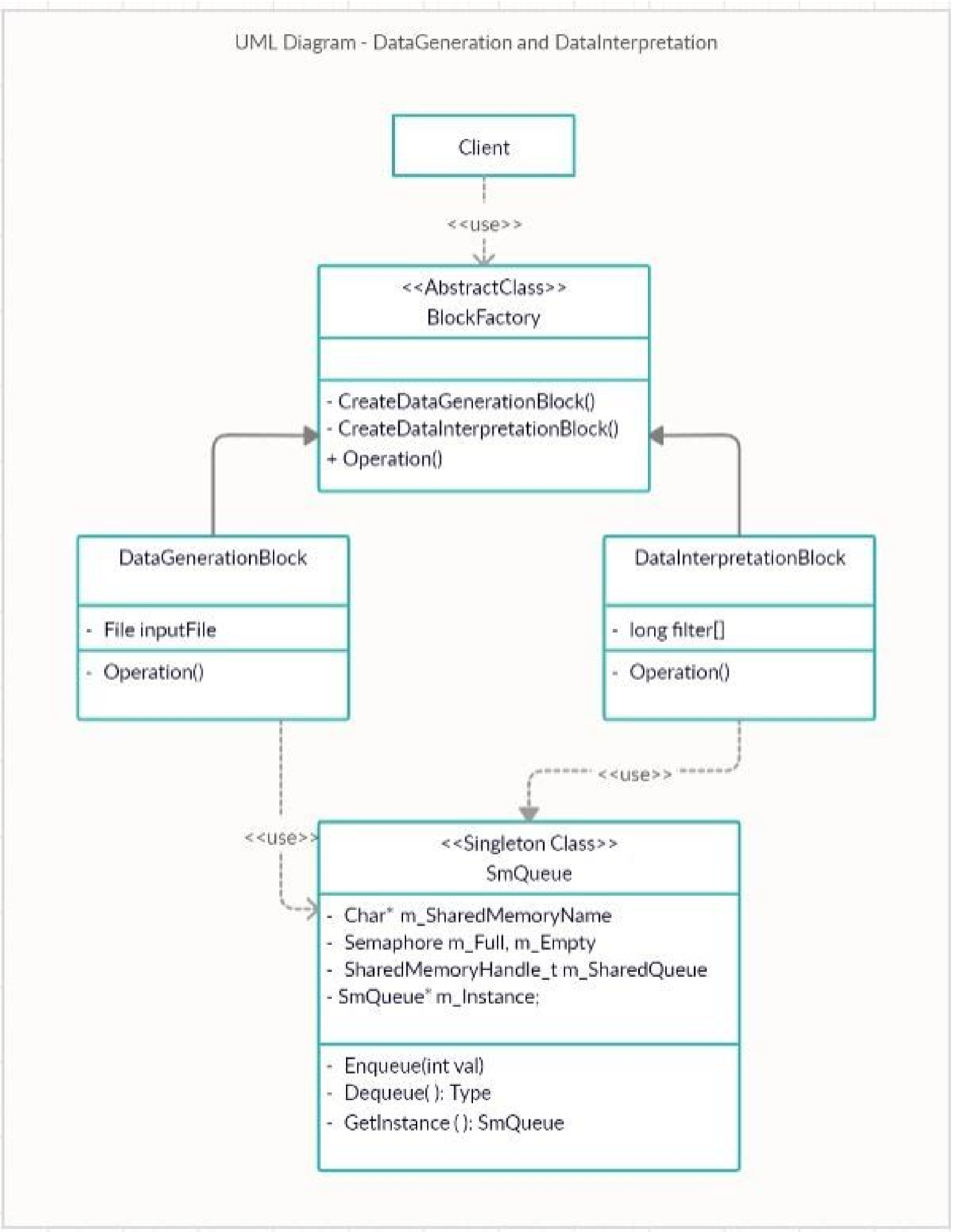


This should be considered as an operating systems bounded buffer problem and can be solved using 2 semaphores.
An 'Empty semaphore' will be signalled when there is at least one empty slot in the queue. And a 'Full semaphore' will be signalled when there is at least one element present in the queue. A producer can wait upon an empty semaphore. And a Consumer can wait on a full semaphore. When these semaphore signals, the waiting processes can perform their corresponding operations.

Links: Solution of Bounded Buffer Problem using Semaphores

As the Queue should be present as a single instance in memory throughout the program execution, This should be implemented as a singleton class.

Overall UML Diagram



Convolution Algorithm Implementation

Convolution is a versatile technique used in signals and image processing. One major use is to increase the speed of operations on signals.
As our inputs are 1D signals and the operation we want to perform is identical to convoluting it with 9 sized kernel array, we propose to use a convolution based algorithm for this purpose.

Normal iterative or brute-force implementation of Convolution requires 9*N multiplications and corresponding additions, which seems very time consuming.
We can reduce the number of multiplications and additions, using the concepts of Fourier transforms in signal processing.

Several operations in the time domain can be simplified by converting it to frequency domain using fourier transforms and performing equivalent operation in fourier transform and finding the inverse fourier transform of result.

Convolution in time domain is nothing but multiplication in frequency domain.
Read it here
https://dspillustrations.com/pages/posts/misc/the-convolution-theorem-and-application-examples.html

Hence, for finding convolution for a window of 9 size, we don't need to perform convolution for each of the 9 elements. Instead we can directly calculate the FFT of the signal and kernel, and multiply the fourier transforms and then get the inverse fourier transform.
This has reduced 9 multiplications into a single fft calculation of signal window and kernel.
For finding Fourier transforms, we need some mathematical library that has FFT Implementation.
mathematical libraries for c++ like intel's MKL (math kernel library) includes FFT operation, but what i am using is FFTW library which is light weight and pinpoints our requirement.

One major issue with fft convolution is it adds up some distortions in starting and ending parts of the signal. This can be avoided using the overlap and add method described in the following link
https://www.dspguide.com/ch18/2.htm

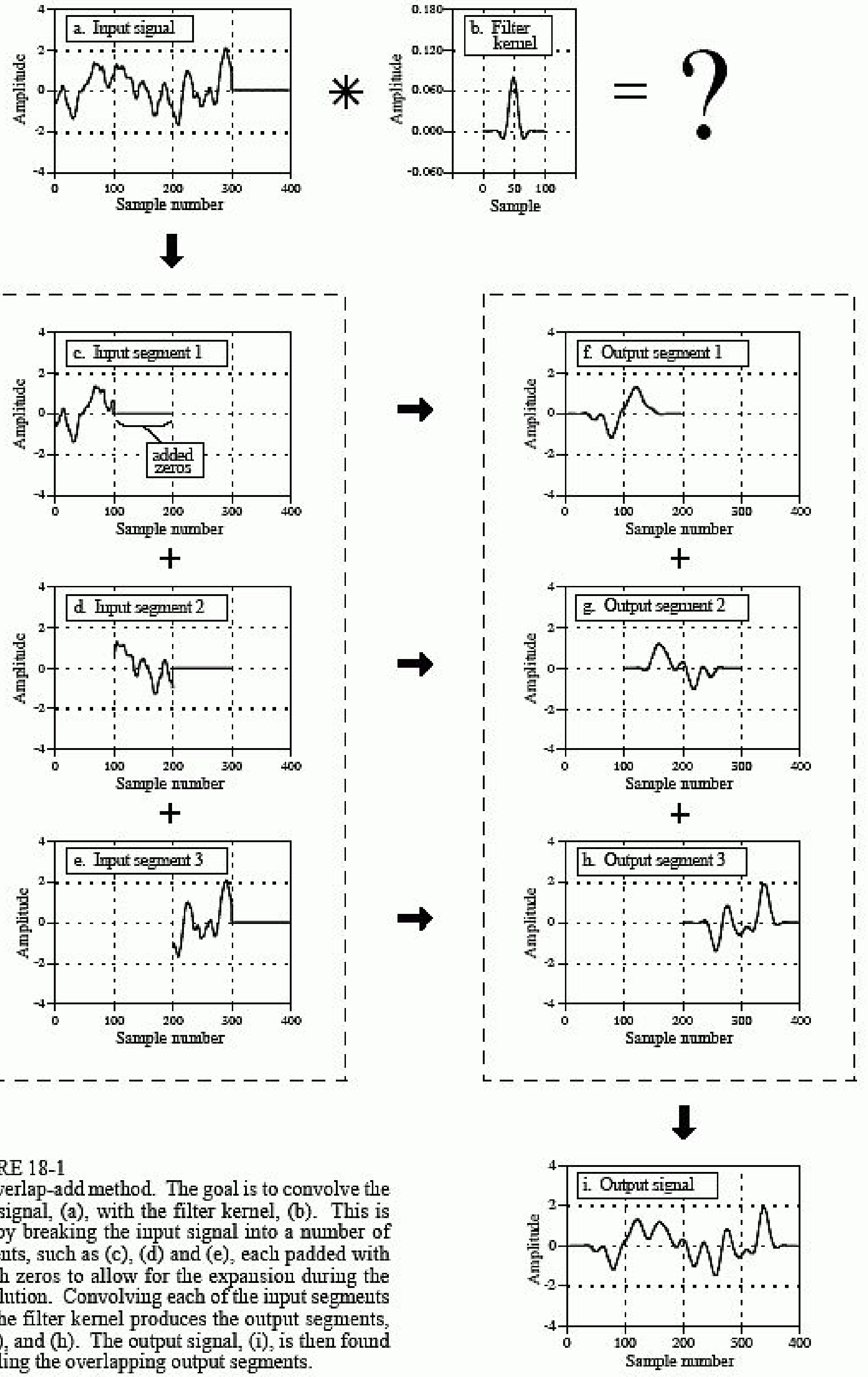


FIGURE 18-1
The overlap-add method. The goal is to convolve the input signal, (a), with the filter kernel, (b). This is done by breaking the input signal into a number of segments, such as (c), (d) and (e), each padded with enough zeros to allow for the expansion during the convolution. Convolving each of the input segments with the filter kernel produces the output segments, (f), (g), and (h). The output signal, (i), is then found by adding the overlapping output segments.

Image courtesy: http://www.dspguide.com/ch18/2.htm

Using the executable

System requirements
Gcc - g++ version 6 or above
Fftw3 library (use sudo apt-get install fftw3 command for installing)

Input file: incsv.txt in source directory.
Suppose we have m lines.

Input format:
<Number of Lines>
<Number of columns in line1>,<elt 1>,<elt 2>,...,<elt u>
<Number of columns in line2>,<elt 1>,<elt 2>,...,<elt v>
.
.
<Number of columns in line m>,<elt 1>,<elt 2>,...,<elt w>

Output file

File: **Out.txt** in source directory
Output format:
<elt 1><elt 2>.........<elt u>
<elt 1><elt 2>.........<elt v>
<elt 1><elt 2>.........<elt w>

Building the code and running

Use the following commands to execute the binary
make clean => cleans the source directory and deletes the binaries and system resources allocated
Make => builds the executable ‘out’ in source directory
./out => Execute the binary ‘out’

Timeline

Day1: Implementation of Blocks
Day2: Implementation of Shared Memory queue
Day3: Implementation of Convolution Algorithm