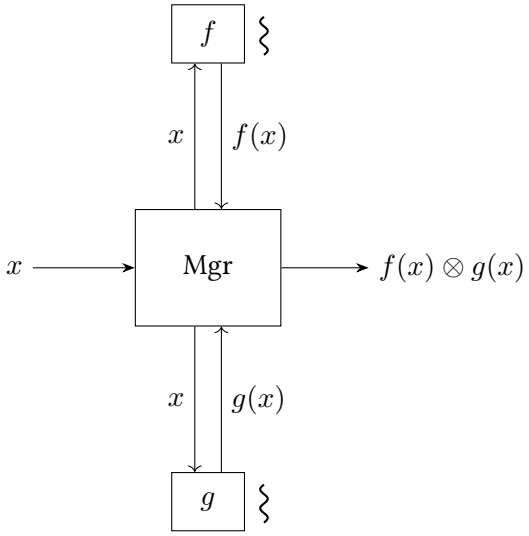

Figure 1: Schematic of the system

Lab 1: Tasks synchronization and parallelization

Problem description

The aim of this lab is to correctly organize computation using multiple processes/ threads. Solution should compute expression $f(x) \otimes g(x)$, where $\otimes$ is the binary operation, integer value x is scanned from the input, computations f and g are parameters and specified independently. The schematic of the system is given in figure 1. The major requirement of the exercise is that computations f and g should run in parallel to main component, manager. Manager initializes computation process, computes the final result and organizes cancellation. Depending on variant manager can be single- or multi-threaded. Multithreaded manager can compute the result in main thread or additional threads which is also stated in the variant. Manager should make no assumptions about computational specifics of f and g. But it is assumed that function result could be undefined for some inputs. System should make allowance for this.

Logic

The application should compute the value of the expression unless computation is canceled.

Short-circuit evaluation Operation $\otimes$ has a *zero* value. If either of functions returns zero value then the system must not wait for the other function to compute and zero result of the computation should be reported immediately.

Cancellation There are three mutually exclusive types of cancellation: by escape key, by periodic prompt, combined. Following is their detailed description.

1. Allow computation cancellation by special key, e.g. ESC.

2. Instead of asynchronous interrupt by key add the periodic user prompt for cancellation. Pay attention to implementing the following logic. User should be given three options:

   (a) continue,

   (b) continue without prompt,

   (c) cancel.

   While user input is pending no additional information should be reported and system status should not change. If cancel was chosen but computation can be completed the result is printed, otherwise reason of incompleteness is reported. Computation short-circuit should be reported.

3. For third cancellation system listens for the escape key. When the escape key is hit the cancellation prompt screen (or window) pops up (obscuring the main screen). One-minute termination countdown is started. If no response is given then the computation is terminated when this period expires. Prompt provides two options:

   (a) terminate,

   (b) continue.

   User could opt to confirm the cancellation and terminate the computation immediately. If user reconsiders then continue option is chosen. In this case prompt screen should disappear without further action. If the result is computed during the countdown then the prompt screen should disappear and the result should be reported accordingly.

Implement either first two types of the cancellation (as different versions of the system) or the third one.

Output All the output should be done by the manager component. System must

- print the result if it is computed;
- report that computation was canceled and inform why computation could not be completed.

Result should be reported as soon as possible for one exception. If the periodic cancellation is implemented then computation output *must not* intervene in user prompt, i.e. all updates should be delivered to user once he or she chooses an option for continuation/cancellation. Result *must* be printed even if user chooses

cancel but result can be computed immediately. Unlike periodic cancellation for combined cancellation system *must* close the prompt screen and show the result as soon as it is available.

Testing

Before submitting the solution make sure to check following test-cases. For interrupt cancellation:

1. f finishes before g with non-zero value. Verify the result.
2. g finishes before f with non-zero value. Verify the result.
3. f finishes with zero value, g hangs. Verify result/time.
4. g finishes with zero value, f hangs. Verify result/time.
5. f finishes with non-zero value, g hangs. Check cancellation, verify status.
6. g finishes with non-zero value, f hangs. Check cancellation, verify status.

For periodic prompt cancellation do additionally.

1. Computation finishes before first prompt. Verify the result/time.
2. Computation finishes during the prompt. Verify noninterruption of dialog, correct report of result despite cancellation, the result/time.
3. f finishes with non-zero value, g hangs. Verify time between answer "continue" and next prompt.
4. Verify all possible answers to the prompt.

It is convenient to organize test-cases as a table that determines desirable f and g behavior for reserved values of x. Elements of this table could be pairs: value of function and time of its computation (including infinity). Have this table readily available for demonstration and explanation.

For system demonstration you should import trial functions from the supplementary code. Details depend on the implementation language and operation used. Here are some language-specific instructions:

- **C++**. Functions are implemented as header-only lib **`demofuncs`**. It defines global function templates `f_func` and `g_func` in `spos::lab1::demo` namespace. Templates are parameterized by enumeration `op_group` defined as

  ```
  enum op_group { AND, OR, INT, DOUBLE };
  ```

  Enum values correspond to binary operations. Sample code:

  ```
  #include "demofuncs"
  //...
  spos::lab1::demo::f_func<spos::lab1::demo::INT>(x)
  ```

Library is tested with `gcc-c++`, `mingw-w64`, `VC 2019`. Since library uses designated initialization feature (C++20 standard/GCC extension) make sure you pass `/std:c++latest` switch to `VC` compiler. For GNU compiler on Linux you have to provide `-pthread` option for compilation and linking.

- **Java**. Jar file **`lab1.jar`** provides package `spos.lab1.demo`. It contains 4 classes corresponding to binary operations: `DoubleOps`, `IntOps`, `Conjunction` and `Disjunction`. Each class provides two static functions `funcF`, `funcG` to be used during demonstration. Sample code:

  ```
  import spos.lab1.demo.DoubleOps;
  //...
  DoubleOps.funcF(x)
  ```

  Add `lab1.jar` to your `classpath` when you compile and run your system.

- **C**. Supplementary code consists of two components: header file `demofuncs.h` and library that depends on the OS. Library provides functions f and g for each binary operation. Functions are named `f_func_<op>`, `g_func_<op>`, where `op` is one of the following `imul`, `fmul`, `imin`, `fmin`, `and`, `or`. Include header file in your module and compile with `lab1` lib.

  Sample code:

  ```
  #include <demofuncs.h>
  //...
  g_func_imin(x)
  ```

  - **Linux**. The C supplementary code is shipped in **`lab1.tar.xz`** archive. It contains shared libraries for both 32-bit and 64-bit builds. Typical compilation command is:

    `cc -L./lib/x86_64-linux -I./include/ -llab1 -o lab1 <`*your modules*`>`

    Make sure that you provide search path for the library when you start the program, e.g.

    `LD_LIBRARY_PATH=./lib/x86_64-linux ./lab1`

  - **Windows**. Use **`lab1.zip`** archive. In addition to dynamically linked libraries (32/64-bit) and header file it contains files necessary for linking. Compilation with MinGW is very similar to the one above:

    `cc -DLAB1_LIB_DYNAMIC -I./include/ -L./lib/x64 -llab1 -o lab1.exe <`*your modules*`>`

    where `LAB1_LIB_DYNAMIC` macro is predefined for better compatibility. Make sure that `demofuncs.dll` is in the search path. Simplest is to put it in the same folder as your executable file.

    For Visual Studio setup your project accordingly. See how to use Dynamic Link Library (C/C++) for more information.

Variants

Exercise has following parameters: implementation language – C, Java, C++, entity – process / thread, mean of communication – global variables / status / pipe / socket / message, operation – multiplication, minimum, conjunction, disjunction. Each student should pick individual variant. Operation is assigned separately.

1. Use Java monitors, threads and attributes to store the result of the function.
2. Use Java threads, `java.nio.channels.Pipe` for function result communication. Use pipe in nonblocking mode. Is it possible to avoid constant polling of channels?
3. Use Java `ProcessBuilder` to organize functions computation by separate processes. Redirect standard output for function communication. Avoid blocking when scanning for function result. Is it possible to avoid constant polling?
4. Use Java, processes, sockets (`java.net.Socket`) and blocking IO. The main process is a socket server with passive main thread.
5. Use Java, processes, sockets and nonblocking IO. The main process is a single-threaded socket server.
6. Use Java, processes, sockets (`java.nio.channels.SocketChannel`), and selector (`java.nio.channels.Selector`). The main process is a single-threaded socket server.
7. Use Java, processes, named pipes (preferably on Linux). Create the pipe externally using system tools.
8. Use Java, processes and Unix sockets (`junixsocket`). The main process is a socket server with passive main thread.
9. Use Java, processes and process exit status for result of function computation.
10. Use Java, processes and `java.nio.channels.AsynchronousSocketChannel` and `Future<T>`.
11. Use Java threads, `PipedInputStream`/`PipedOutputStream` for function result communication. Use pipe in nonblocking mode. Is it possible to avoid constant polling of channels?
12. Use Java, processes, sockets (`java.net.Socket`) and blocking IO. The main process is a socket server using `java.util.concurrent.Executor`.
13. Use Java, processes and Unix sockets (`junixsocket`). The main process is a socket server using `java.util.concurrent.Executor`.

14. Use C/C++, processes and Windows messages to communicate the function result. Implement system as GUI application. Use hidden windows for messages. Prompt should appear as a separate dialog/message box. Use additional thread for prompt/main window synchronization.

15. Use C/C++, processes and Windows messages to communicate the function result. Implement system as console application. Use `PostThreadMessage` call for communication. Use additional thread for timing.

16. Use C++, threads, condvars and global vars or attributes to pass the result of function computation.

17. Use C++, `Boost.Process` and unnamed pipes for function communication.

18. Use C++, processes, and named pipes to pass the result of function computation. Use `std::async` to organize monitoring of function computation status.

19. Use C++, processes, and sockets to pass the result of function computation. Use `std::async` to organize monitoring of function computation status in server.

20. Use C, threads (`pthreads`), condvars and global variables to store the function results.

21. Use C, processes (`fork`), `waitpid` for synchronization and exit status to communicate the result. Implement the whole system in one program.

22. Use C, processes (`fork`) and unnamed pipes. Implement the whole system in one program. Use polling.

23. Use C, processes (`fork`) and unnamed pipes. Implement the whole system in one program. Use `select` system call.

24. Use C, processes and named pipes to communicate the function result. Use nonblocking IO.

25. Use C, processes and named pipes to communicate the function result. Use `select` system call.

26. Use C, processes and unix (local) sockets to communicate the function result. Use nonblocking IO.

27. Use C, processes and unix (local) sockets to communicate the function result. Use `select` system call.

Extensions

1. Generalize to n functions.
2. Generalize to arbitrary binary operation.