Systems Software

VISUALIZATIONANDMULTIMEDIALAB

BINF31aa (BINF3101)

Prof. Dr. Renato Pajarola

Exercise Completion Requirements

- Exercises are mandatory, at least 3 of them must be completed successfully to finish the class and take part in the final exam. The first exercise serves as an introduction to C/C++ and does not count towards admission to the final exam.
- Exercises are graded coarsely into only two categories: ***fail*** or ***pass***
  - if all the exercises (without considering the first introductory exercise) are completed successfully, then a bonus is carried over to the final exam[1]
- Turned in solutions will be scored based on functionality and readability
  - a solution which does not compile, run or produce a result will be a ***fail***
  - the amount of time spent on a solution cannot be taken into account for the score
- Exercises can be made and submitted in pairs
  - both partners must fully understand and be able to explain their solution
- Students may randomly be selected to explain their solution
  - details to be determined during exercises by the assistant

Exercise Submission Rules

- Submitted code must compile without errors.
- Code must compile and link either on Mac OS X or on a Unix/Linux/Cygwin environment using a Makefile

The whole project source code (including Makefile) must be submitted via OLAT by the given deadline. Include exactly the files as indicated in the exercise.

We will use MOSS to detect code copying or other cheating attempts, so write your own code!

The whole project (including Makefile) must be zipped, and the .zip archive has to be named: ss_ex_EXERCISENUMBER_MATRIKELNUMBER.zip (ss_ex_3_01234567.zip - one student, Systems Software exercise number 3; ss_ex_1_01234567_02345678.zip - two students, Systems Software exercise number 1).

Submit your code through OLAT course page.

Teaching Assistant: Claudio Mura (claudio@ifi.uzh.ch)

Help with C++, Makefiles and UNIX programming

C++ Tutorial:	http://www.cplusplus.com/doc/tutorial/
C++ Library Reference:	http://www.cplusplus.com/reference/
C++ STL:	http://www.sgi.com/tech/stl/table_of_contents.html
GNU Make Tutorial:	http://www.gnu.org/software/make/manual/make.html
Reference book:	"Advanced Programming in the UNIX® Environment, 2nd edition" *W. R. Stevens, S.A. Rago,* Addison Wesley

[1] the bonus will be in the order of 10% of the total number of points achievable in the final exam

Exercise 6: Simulating the Behaviour of a Memory Allocator in a Multi-process Environment

Your goal for this exercise is to develop a multi-threaded application that simulates the behaviour of a simple memory allocator in a multi-process environment. To accomplish this task you will make use of the thread-management routines and of the mutual exclusion tools provided by the PThreads library.

The task of a memory allocator is to manage the available memory and to allocate free blocks to satisfy memory allocation requests from processes. For this exercise, we will consider the memory as divided into *B* blocks of fixed size. At any point in time, a block can be unallocated or it can be allocated to a certain process. Our memory allocator holds a data structure containing *B* entries, one for each block, and stores in each entry the id of the process that is currently holding the corresponding memory block. In our system, there is a fixed number *P* of processes. Every now and then, each process needs to perform some work, and needs *some* memory blocks (the number depends on the amount of work it has to perform) to accomplish this task. Whenever it needs to perform some computation, a process determines the number of memory blocks *W* it needs and requests *W* memory blocks to the allocator. If the allocation is successful, the process completes its task and then signals the allocator that the memory blocks used can be released; otherwise, it aborts the computation.

In your application, the behaviour of the processes will be simulated by *P* threads. Every *T* milliseconds, with *T* being a random number in the range [T_{min}, T_{max}], a thread tries to get *W* blocks of memory from the allocator; *W* is chosen randomly in the range [W_{min}, W_{max}]. If the allocation is successful, the thread will simulate performing some computation by sleeping for a random amount of milliseconds in the range [T_{min}, *W* x T_{max}]; upon wake up, the thread will communicate the allocator that the *W* memory blocks it was holding can be released. If the allocation is not successful, the thread does nothing. To sum up, the thread runs a loop similar to this:

```
do
      generate random number W in the range [ Wmin , Wmax ]
      request allocation of W blocks of memory
      if allocation successful
            sleep for some random time in the range [ Tmin , W x Tmax ]
            release allocated blocks
      end
      sleep for some random time in the range [ Tmin , Tmax ]
while simulation is running
```

Your implementation of a memory allocator should include two main elements:

1. a data structure to represent the state of the memory blocks: it must be an `std::vector<int>` and should have a size of *W*, so that there is one entry for every memory block; each entry will contain either -1 (if the corresponding memory block is free) or the id representing the thread to which it has been assigned;
2. two functions/methods to allocate and release blocks of memory: the function that performs the allocation should take an integer representing the number of blocks needed and, if there are enough blocks available, should allocate them to the calling thread and mark the corresponding entries of the data structure accordingly; the function that releases the memory should simply mark as non allocated the entries corresponding to the blocks currently assigned to the calling thread.

Since the data structure will be modified by multiple threads running simultaneously, you **must** ensure that the accesses are properly synchronised, avoiding race conditions. This must be a key point in your solution. To achieve this goal, make use of the pthread mutexes (no other synchronization tools are allowed!).

Some additional remarks about this task:

- the number of threads P, the number of blocks B and the values W_{min} , W_{max} , T_{min} , T_{max} must be passed to your application as command-line argument, in this order;
- the simulation should end when the user presses the ‘e’ key followed by the return key;
- you must provide a simple yet meaningful console output that shows how the simulation is running; in particular, each thread should print information on the memory allocation requests, including how many blocks were requested and the outcome of the request; since the console is a shared resource, synchronise the writes using a mutex.

Further clarification on this exercise will be provided during the tutorial session of November, 23rd.

Notes:

- you are provided with some sample source files and with a simple Makefile; you may use them as a basis for your solution.

Deliverables:

Electronically submit your project files (i.e. source files and Makefile) and a README file that briefly explains your program, all in a single ZIP file.

Deadline: 6th December, 2015 at 23.59h.

NOTE: for this exercise it is assumed that you have acquired sufficient familiarity with the C++ language and with Makefiles. Before you start writing the solution make sure you have fully understood the relevant theory chapters from the Operating Systems reference books.