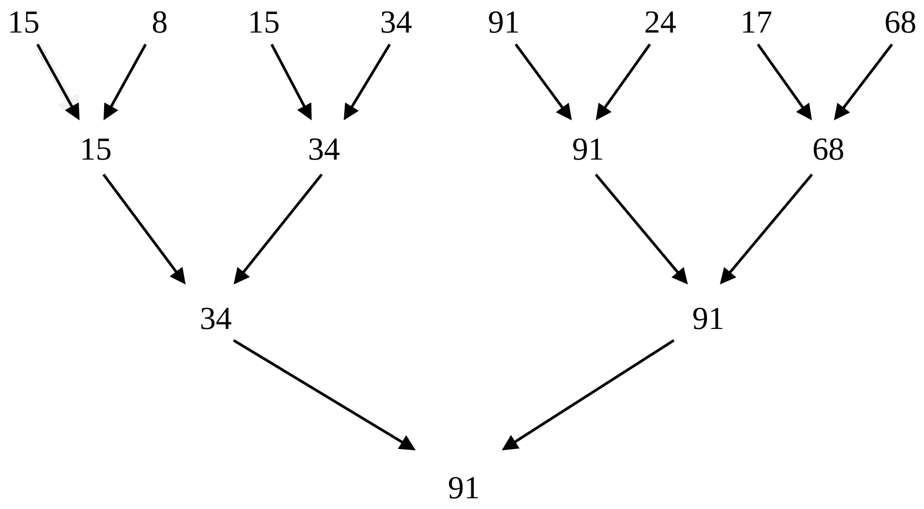
CS4414 (SP2017) Operating Systems

Project 2

Title: Maximum Finder
Due: March 28, 2017 (12:30 p.m.)
Points: 20

Write a program in C/C++ to determine the largest number for a sequence of integers using a divide-and-conquer algorithm. The following is an example showing you how to determine the largest integer for a sequence of 8 integers using the algorithm: 15 8 15 34 91 24 17 68. **Note** that the integers contained in the sequence do NOT have to be unique. **Note** also that the algorithm uses 3 rounds to determine the answer.



Your program should create multiple threads in each round to determine the larger integer for each pair of integers. For example, in round 1 of the example shown above, 4 threads are created to determine the larger one for each pair of integers (a total of 4 pairs). All threads created in round 1 will be synchronized in such a way that when a thread finishes determining the larger one in its pair, it waits until all other threads are done their jobs before round 2 is started. This process is iterated until the largest integer is determined.

The integers from which the largest one is to be determined MUST be stored in a **.txt** file, which is taken as a command line argument (the second argument in the command) by your program. The integers MUST be stored in the file in such a way that each integer is stored on one line. You may assume that the number of integers stored in the file is always a power of two (e.g. 8, 16, 32, *etc*.). You may also assume that the total number of integers stored in the file is no more than 4096. Once the largest integer is determined, print the result to the display (e.g. The largest integer is 123) and terminate your program.

The total number of rounds for a sequence of integers your program needs to run can be determined using the following formula: $\log_2 n$, where n is the total number of integers contained in the sequence. The total number of integers to be processed at each round can be determined by dividing the total number of integers processed at the previous round by 2. Your program continues to run until the last round is finished.

The best way to handle synchronization among threads for each round of comparisons is to implement a **barrier** of which its main job is to make a finished thread wait until all other threads are done. The barrier can be implemented in such a way that it is shared by all rounds of comparisons *via* passing a value to it indicating the number of threads running concurrently for a particular round. When a thread finishes its job, it issues a call to the barrier and the barrier checks to see if the thread needs to wait (What the barrier does here is to decrement the value passed to it. If the value after decrement is nonzero, then the thread calling the barrier is blocked on the **condition variable**. Otherwise, the barrier will release all threads from the condition variable *via* invoking the *pthread_cond_broadcast*() function). Make sure you use a **semaphore** or **mutex** to protect the barrier from being accessed by multiple threads at the same time.

In your written evaluation (at least two pages long in **PDF** format), be sure to discuss your implementation, specifically: (1) how you implemented the barrier; (2) how you tested your barrier; (3) whether or not your main thread participates in comparisons; and (4) how you organized (in memory) the intermediate results.

Miscellaneous

(1) The project does NOT allow team work.
(2) You MUST use a single barrier for all rounds of comparisons.
(3) You MUST NOT use the pthread barrier type (*i.e. pthread_barrier_t*) and the pthread barrier functions (*i.e. pthread_barrier_init*() and *pthread_barrier_wait*()) for the project; you MUST implement the barrier in your program using pthread synchronization primitives (e.g. mutex, semaphores, condition variables, *etc*.). Using system-supported barrier type and functions will lose you all the points.
(4) You MUST NOT use the *pthread_join*() function to wait for all of the threads to finish for each round of comparisons.
(5) You MUST use a makefile to compile your program.
(6) You MUST use the Debian VM for this project.
(7) Your implementation MUST be in C/C++ and use the POSIX APIs for threads and synchronization functions (e.g. mutex, semaphores, condition variables, *etc*.)

(8) Your program should print only the final answer (e.g. the largest number is 125) and suppress all unnecessary output generated for debugging purposes (e.g. the largest numbers for round 1 are 32 71 16 55).

(9) Turn your project in *via* **Collab** in a **tar** file consisting of (i) all headers you have created; (ii) source-code file for project 2; (iii) a make file; and (iv) a write-up showing your design and implementation of the project (at least two pages long in **PDF** format). The **tar** file should be named as follows: **p2**, followed by your computing ID, and followed by the file extension (*i.e.* **tar**). For example, the **tar** file turned in by **John Smith** should be named **p2js7nk.tar**, where **js7nk** is the computing ID of **John Smith**.

(10) Your program should be successfully compiled. Programs failing to compile will NOT receive any points.

(11) You should use the following cases to test your program: (i) an empty file; (ii) all integers are unique; (iii) some integers are identical; and (iv) all integers are identical. You should test your program with the number of integers up to 4096 (you may use a random number generator to generate and store the 4096 integers in the input file). Make sure you change the base memory size in VirtualBox to 2046MB (~ 2GB) when you test your program with 4096 integers. The setting can be found under "System" after you launch the VirtualBox application. You should also type "ulimit –s 512" (without quotes) in the Debian environment to reduce the stack size to 512KB for each thread created in your program so that more than 6000 threads total can be created. More threads can be created if you further reduce the stack size from 512KB to 256KB.