CMSC 216 Project #6 Spring 2013

Date due: Tuesday, May 7, 10:00:00 p.m.

1 Administrative

The TAs' "extra" office hours for this project will run from Friday, May 3, through Thursday, May 9.

You **must** know how to use the `gdb` debugger, and `valgrind` (`memcheck`), and have used them, to throughly debug your code before asking for debugging help in the TAs' office hours. The TAs will not debug any code for you that you have not already made every effort to debug yourself first.

You only need to run one copy of Emacs at a time. Running many copies of Emacs just slows down the system and the wireless network for everyone. If you just want to look at a file, use `less` instead of the text editor. You can use one Emacs to edit multiple files and switch back and forth between them, as described in the UNIX tutorial and demonstrated in discussion section.

You have to write a `Makefile` again in this project. Some students had problems in earlier projects with their `Makefile`. Although their code may have worked perfectly, it failed to compile due to not having written a `Makefile` at all, or due to having an incorrect `Makefile`. This could have resulted in significant loss of credit. You are emphatically urged **not** wait to write your `Makefile`– write it the first time you are ready to compile any code, and just add on to it as needed. This will have two significant advantages: firstly it will greatly facilitate the process of compiling and testing your code, by making it unnecessary to type numerous compilation commands manually, and secondly, you will be testing and debugging your `Makefile` every time you compile your code, so you will be much less likely to have problems with it when you are ready to submit.

After finals are over, be sure to come back to this assignment and read the appendix that discusses accessing files and cleaning up your TerpConnect account after the semester ends, otherwise you could lose information that you may want to save.

2 Introduction

Mergesort is an algorithm that can be easily adapted to use multithreading to decrease execution time. In this project you will modify an implementation of mergesort that we are providing, which does not use multithreading, to add the capability to operate with up to four threads. You will also write a program to aid in testing and timing your modified mergesort implementation.

3 Project specifications

3.1 What to do

The sequential (nonthreaded) mergesort implementation we are supplying consists of two functions, `mergesort()` and `merge()`. What you are to do:

1. Write a multithreaded mergesort function that allows an array to be sorted using one, two, or four threads. This multithreaded mergesort should call the nonthreaded mergesort to do much of the actual sorting work. Note that to do this you do not need to modify our `mergesort()` function, or its helper function `merge()`, at all; you just need to call them in your code.

2. Write a program to store random numbers in an array, and time how long it takes your mergesort implementation to sort this array.

   You must add to your testing program calls to your multithreaded mergesort to time the sort using different numbers of threads.

3. In a plain text file named `report`, answer several questions about the performance of your mergesort implementation, as measured by the testing program; the questions are described in Section 3.6.

   The timing calls necessary to do this may not have been covered yet in class by the date this project is assigned, but if not they will be shortly.

Note that the modifications to our mergesort can be accomplished in around 60 lines of code, which means that that is all you have to write to pass all of the public and secret tests. The testing program you need to create to write the report will be a little longer, but can probably be written in around 125 lines of code.

3.2 The mergesort algorithm

Mergesort, in its simplest form, is an algorithm that divides its input list into two lists, sorts them, and then merges the two sublists into a sorted version of the input list. The algorithm is shown here:

```
MERGESORT($A$):
  if LENGTH($A$) $\leq 1$
    then return $A$
  else begin
    $B \leftarrow$ MERGESORT(first half of $A$)
    $C \leftarrow$ MERGESORT(second half of $A$)
    return MERGE($B$, $C$)
  end
```

3.3 Comparison functions

Our mergesort implementation is able to sort any type of data, including `int`s, `double`s, strings, and `struct`s, because it is written so its array parameter is a `void` pointer. Since arrays are passed to functions as pointers (to their first element), and any pointer can be assigned or passed into a `void` pointer, this allows the function to be called with any type of array. However, because sorting different data types requires different comparison code for each type (e.g., strings must use `strcmp()`, while `int`s can be compared using relational operators), it uses function pointer to perform comparisons. The caller of your sorting function supplies a pointer to a function that the caller will, in general, also write (except possibly in the case of `strcmp()`, which is already available in the C string library). (In other words, your mergesort will not contain the comparison function and you will not write it, except perhaps in creating your own tests; any user of your code, such as our tests, must write it.) The prototype for the comparison function is in `mergesort.h`, but it must return a value in the same way that `strcmp()` does. That means the comparison function must return an integer greater than, equal to, or less than 0, if the value pointed to by its first argument is greater than, equal to, or less than the value pointed to by its second argument, respectively. In order to be able to called on different data types, the comparison function prototype takes two `void` pointers as parameters. A particular comparison function must cast them to the actual type of data being sorted, then compare the elements being pointed to.

Use of the comparison function is roughly analogous to the way that `Collections.sort()` can sort a `Collection` of any type of object in Java, but since a `Collection` can store any type of objects, a `Comparator` must be supplied, that knows how to compare objects of the specific type being stored, that it will call anytime objects are being compared in the process of sorting.

3.4 Functions to be written

Your `mergesort.c` may contain any helper functions you find useful, but it must contain the following functions, whose prototypes are in `mergesort.h`. First read over our supplied sequential mergesort implementation, so you understand what it does, since you will have to call our functions in the process of writing your own function.

```
void mergesort(void *arr, int num_elts, int elt_size, Compare_fn cmp)
```

This function is provided for you, and sorts the first `num_elts` elements of the array `arr` using the comparison function `cmp` to define the ordering of elements. It uses the mergesort algorithm to perform its sorting. If either `arr` or `cmp` are `NULL` the function just returns, doing nothing to the input array.

```
void threaded_mergesort(void *arr, int num_elts, int elt_size, Compare_fn cmp,
                        int num_threads)
```

This function, which you will write, also performs a mergesort in the same manner as the `mergesort()` function, but splits the work to be done among `num_threads` new worker threads (i.e., there should be `num_threads + 1` threads, including the main thread). The main thread should not do any sorting work while the worker threads are doing their work. Here is some more explanation:

- If `num_threads` is not one of $\{0, 1, 2, 4\}$, or the array or comparison function are `NULL`, the function should do nothing and make no changes to the input array.
- If `num_threads` is 0, the function should just call our nonthreaded `mergesort()` function to sort the array without creating any threads. (This is just so the performance of the nonthreaded case can be easily compared to using different numbers of threads.)
- If `num_threads` is 1, the function should create one thread, and that thread should call our `mergesort()` on the entire array. (This is just to be able to compare the performance of different numbers of threads, including only one thread.) Notice that this will involve creating a thread and passing several arguments to it; how to do this was illustrated in discussion section.
- If `num_threads` is 2, the function should create two threads, and each thread should call our `mergesort()` on half of the array. (Each thread will do exactly the same thing as the single thread created in the case above, just on a subpart of the array.) When the threads finish, the two halves of the array will be sorted, but then they need to be merged to create the final sorted array, which your function will have to do. (Hint– our function `merge()` will do this.)
- If `num_threads` is 4, the function should create four threads, and each thread should call our `mergesort()` on a quarter of the array. (Each thread will do exactly the same thing as the single thread created in the case above, just on a subpart of the array.)

  When the threads finish, the array will be in four sorted subparts, which your function needs to merge, but notice that two quarters will need to be merged to create a sorted half, the other two quarters will need to be merged to create the other sorted half, then the two sorted halves will need to be merged to create the final sorted array.

3.5 Testing program

Your program `mergetest.c` should be run with three integer arguments on its command line. The first argument, `n`, is the number of integers that must be sorted. The second argument, `s`, is the number that is used to seed the random number generator you will use to generate the array of numbers to sort. The third argument, `m`, is the upper bound on the value of the numbers to be sorted. You may assume that the numeric values of all three arguments can be stored in a signed `int` without any loss of data. The `atoi()` function may be used to convert command line arguments to integer values.

Your program will need to dynamically allocate an array to hold the `n` integers. The array should be filled with `n` random integers obtained using this algorithm:

```
srand(s);
for (i= 0; i < n; i++)
  arr[i]= rand() % m;
```

After setting up the array, your program should proceed to sort it using each of the mergesorts implemented (nonthreaded, multithreaded with 1, 2, and 4 threads) – making sure to sort a **copy** of the original array **each** time, as sorting already sorted data will invalidate your results. After each sort, you should report to standard output the time spent sorting, in terms of wall clock time, user time, and system time, in the format below.

```
0 threads:  0.751018s wall;  0.735888s user;  0.002999s sys
1 threads:  0.774871s wall;  0.779882s user;  0.000000s sys
2 threads:  0.417598s wall;  0.781881s user;  0.003000s sys
4 threads:  0.251508s wall;  0.779882s user;  0.012998s sys
```

You should also check after each sort to ensure that the array truly is in sorted order, and that it is a permutation of the original array (see the project public tests for an idea about doing this). Note that none of our tests will directly run your mergetest program; you are writing it in order to gather data necessary to write the short report described in the following section.

3.6 Report questions

In your `report` file, you should answer the following questions. Lengthy explanation is not required, but be sure to address all of the questions.

1. For each of the four versions of mergesort (nonthreaded, and threaded with one, two, and four threads), how long did that version take to sort the numbers, measured by wall clock, user, and system time?
2. Measuring wall clock time, what speedup did each of your three multithreaded runs obtain compared to the nonthreaded mergesort? *Speedup* is computed as the time for the nonthreaded run divided by the time for a multithreaded run.
3. Measuring user time, what speedup did each of your three multithreaded runs obtain compared to the nonthreaded mergesort?
4. Explain the relationship between an increase in the number of threads working on a mergesort and the time (both wall clock and user time) required to perform the sort. In other words, how much speedup is gained by doubling the number of threads? Does this speedup increase at a constant rate as the number of threads becomes larger? Why or why not?

It's suggested you us 2^{22} integers for your measurements (about four million), but you may make your own choices. However, if your choice is too small, the effects you are trying to find may not show up, and you won't get full credit for the report.

Be sure to run your mergesort multiple times. You may choose as the representative time for each number of threads the **best** time you see for that number. This is because deviations from the best time are most likely due to interference from other users of the Grace machine.

3.7 Create a `Makefile`

As for some of the prior projects you must create a `Makefile` that will be used to build the public tests on the submit server. Your `Makefile` must use separate compilation (i.e., all source files must be turned into object files, which are then linked together in a separate step). All object files must be built using the same flags to `gcc` as in previous projects. Your `Makefile` must have the following targets:

all: should create all executables

clean: should delete all object files and executables

`mergesort.o` and `mergetest.o`: the object files for your mergesort code and your testing program

A target for each public test executable: each target name must end in `.x`, and its rule must builds an executable whose name ends in `.x` (e.g., a test `public1.c` should have a target `public1.x` in your `Makefile`, that builds an executable `public1.x`).

A target for each public test object file: as above, separate compilation should be used for all files.

`mergetest.x`: the executable testing program created from `mergetest.c`.

We will use the `Makefile` you provide to build the public tests on the submit server; if there is no `Makefile`, or its name is spelled wrong, or it has any errors, the tests will not be built, and you will not receive credit for any tests.

4 Important points and hints

1. For full credit, your threaded mergesort should work for any number of elements; however, you can still receive credit on some tests if your mergesort works for specific numbers of elements. (Note that "number

of elements" refers to the number of array elements to be sorted; the number of threads can only be 0, 1, 2, or 4.) All implementations should work for arrays of size 2^n, where $n \in [3, 22]$; better implementations might work for arrays of size $4n$, where `n` is any positive integer. Other things that might cause your implementation to be less than perfect are a failure to handle duplicate items correctly, or an inability to handle items other than integers.

2. Recall that a `void` pointer cannot be dereferenced without casting to another type, one consequence of which is that pointer arithmetic cannot be performed with a `void` pointer. However, you will need to perform pointer arithmetic with the array parameter to `threaded_mergesort()`, which is a void pointer. Recall that a `char` is defined to always be one byte of memory. This means that `char` pointers can be used to perform pointer arithmetic when needed in the project, as pointers to `char` can be used to access arbitrary byte addresses. You will note that we used `char` pointers in this way in our `mergesort()` and `merge()` functions; you can do the same.

3. First get your `threaded_mergesort()` to work for only one thread. That will involve being able to create a thread, pass it the proper arguments, etc., to ensure that you can do these things correctly. It should also give exactly the same results as the sequential version not using threads at all, so if your results are the same, you must be creating and using your thread correctly. Then you can go on to modify it to use multiple threads (two and four).

A Development procedure review

A.1 Obtaining the project files, compiling, and checking your results

You can obtain the necessary project files by logging into one of the Grace machines using commands similar to those used before:

```
cd ~/216
cp ~/216public/project6/project6.tgz .
tar -zxvf project6.tgz
```

The tarfile will create a directory `project5` that contains the necessary files for the project, including the header file `mergesort.h` that contains prototypes for the functions you must write, and also `mergesort.c`, which contains the nonthreaded implementation of mergesort.

A.2 Submitting your program

As before, the command `submit` will submit your project. **After** you submit, you **<u>must</u>** then log onto the submit server and check whether your program worked right on the public tests there. Unless you have versions of all required functions that will at least compile– whether they are used in the public tests or not– your program will fail to compile at all on the submit server.

Do **<u>not</u>** submit programming assignments using the submit server's mechanism for uploading a jarfile or zipfile or individual files. Do not wait until the last minute to submit your program, because the submission server enforces deadlines to the exact second.

Project extensions will not be given to individual students as a result of hardware problems, network problems, power outages, etc., so you are urged to finish **early** so any such situations will not affect you even if they do arise.

A.3 Grading criteria and style grading

Your project grade will be determined according to the following weights:

Results of public tests	35%
Results of secret tests	35%
Code style grading	15%
Report	15%

The public tests will not be comprehensive, and you will need to do testing on your own to ensure the correctness of your functions.

The only files that will be graded for style are `mergesort.c` and `mergetest.c`; we will check your `Makefile` for deviations from the requirements in Section 3.7, but it does not have to follow any of the stylistic requirements for C code described in the project style guide. Code that you add to `mergesort.c` should be commented, but we feel that the code we're supplying you is sufficiently commented, so do not feel any obligation to add any comments to the code that we wrote. Be sure to **carefully read** the course project grading policy posted on ELMS, which contains the style guide.

B Project–specific requirements, and notes

- You may **not use** the C library function `qsort()` anywhere in writing `threaded_mergesort` (although some or all of the tests may use it, to check for correct results). You may not exec a program that does sorting. You must modify and use the `mergesort()` function provided, with threads, otherwise you will receive **no credit** for the project. (You may use `qsort()` in writing your mergetest program if desired.)
- You **cannot** modify the declaration of anything in the file `mergesort.h`, or add anything to it, because your submission will be compiled on the submit server using our version of this file.

  Your code **may not** comprise any source (.c) files other than `mergesort.c` and `mergetest.c`, so all your code **must** be in these files. Their names must be spelled and capitalized **exactly** as shown, so just use the versions provided in the project tarfile (which just contains a skeleton version of `mergetest.c`).

  You also **may not** add any header files to the project.
- Note that the name of the report file must be `report` (as in the skeleton version supplied in the project tarfile), and **not** `REPORT`, `Report`, `report.txt`, `RePoRt`, `reports`, or any other variation or spelling. It **must** be a plain text file, created on the Grace machines using a text editor (e.g., Emacs), not a PDF, Word document, or any other format. (Since different operating systems have different conventions for newline characters, even a text file created on another operating system may be unreadable when transferred to a UNIX system, unless it is properly converted first.) The grading TAs do not have time to search manually through 240 submissions or so for everyone's report using different filenames, or to convert file formats in order to read them. Using a different filename or file format will result in not receiving credit for the report.
- Be sure to carefully reread the requirements for your makefile in Section 3.7 before submitting.
- Recall that the course syllabus says that all your projects must work on **at least half of the public tests** (by the end of the semester) in order for you to be eligible to pass the course. See the project grading policy for full details. (The compilation penalties mentioned below only apply to submissions made before the end of the project late submission period.)
- For this project you will **lose one point** from your final project score for every submission that you make in excess of five submissions. You will also **lose one point** for every submission that does not compile, in excess of two noncompiling submissions.

C Academic integrity

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus– please review it at this time.

D End–of–semester TerpConnect account cleanup

Over the summer you will lose permission to access the files in your extra disk space (including all your projects), as well as the files we've put in the directory `216public`. All these files will be automatically deleted later by OIT. If you want to save any of your projects or other coursework, or any of the files or information we provided (examples from discussion, secret tests, etc.), it will be necessary to do so soon after finals are over. Sometimes students are asked for copies of class projects during job interviews, as part of scholarship applications, for internships, etc.; the instructor and TAs will not be able to provide these in the future, so you'll need to save copies yourself if you want them. Recall that the `-r` option to the `cp` command recursively copies a directory and all its contents, including subdirectories, so a command like

```
cp -r ~/216 ~/216.sp13
```

would copy everything in your extra course disk space (which the symbolic link `216` in your home directory points to) to a directory in your TerpConnect account home directory named "`216.sp13`" (use a different name as you like). Of course you have to have enough free disk space in your TerpConnect account to store the files; recall from the UNIX tutorial that the `quota` command lets you know how much free space you have. Although you will lose your login permission to the Grace systems after the semester, you will still be able to log into your TerpConnect account via any TerpConnect machine (just log in to `terpconnect.umd.edu`), since your TerpConnect account will remain as long as you're associated with the University.

You can also download files to your own computer; if you're using Windows a WinSCP client that will do this can be obtained from `http://winscp.net/eng/index.php`.

When you're copying files you want to save, be sure to undo the changes to your account that were made during an early discussion section, since you may use the Grace systems for other courses in the future and the changes you made for this course may conflict with changes necessary for them.

1. Remove our directory from your path in your file `.path`, by editing it and just removing the line beginning "`setenv PATH ...`" that you added earlier. (Of course,if you've added any directories of your own to your path then leave those changes alone.)

2. Remove the symbolic link named `216` that you created from your home directory to your extra disk space, by just removing the symlink, as in `rm ~/216`. You could still reach the files in your extra disk space after that if you wanted to (until you lose access to them) just by using the full pathname, for example, you could still use a command like:

   ```
   cd /afs/glue/class/spring2013/cmsc/216/0101/student/loginID
   ```

   instead of `cd ~/216`. For convenience though, you probably want to wait to remove the symlink after you've copied any desired files from your extra disk space.

3. Also remove the symbolic link named `216public` that you created in your home directory, which points to the class public directory `/afs/glue/class/spring2013/cmsc/216/0101/public`, after copying any files you want to from there.

4. Remove the line `tap -q java` that you added to your `~/.cshrc.mine` file, as well as the line `source ~/216public/.216settings` from your file `~/.startup.tty`.

   You can remove the `maxproc` limit from your `~/.cshrc.mine`, unless you plan to run Project #5 again later.

5. You may want to remove the alias for `gcc`, if you added one to your `~/.cshrc.mine` file. Any other useful aliases you added to your `~/.aliases` file you probably want to leave, so they'll still be in effect if you use the Grace systems for other courses.

Lastly, at some point the class ELMS space will become inaccessible (although it may last for several semesters, the University probably will not leave it up indefinitely), so if there's any information you want from it that you don't already have, be sure to save it after your finals are over.