

EECS 3540 – Operating Systems & Systems Programming

Spring 2021 – Programming Lab #2 – Multithreading Applications

Due Sunday 3/7/21 at 11:59 PM

For this lab, you will be implementing POSIX-style Pthreads to multi-thread Quicksort for sorting huge lists. For this assignment, we will stick with integers as the values to sort.

Your code will be written in C for the Virtual Machine supplied in class.

The syntax for the command line for your program will be:

```
// The command-line syntax is
//
// project2 SIZE THRESHOLD [SEED [MULTITHREAD [PIECES [THREADS]]]]
//
// where:
// SIZE          is the number of items to sort (must be a positive integer)
//
// THRESHOLD     is the point at which we switch from recursive Quicksort to Shell sort.
//               If the size of the segment is less than or equal THRESHOLD, we use Shell
//               This implies that if we start the program with the same value for SIZE
//               and THRESHOLD, it will make one call into Quicksort, which will then sort
//               the entire array using Shell sort. At the other end, if we set THRESHOLD
//               to zero, we will ALWAYS recursively Quicksort the entire array
//
// SEED          is an optional (integer) parameter. If omitted, the random number generator
//               is
//               not seeded, and we run with whatever sequence it generates (note: it will
//               generate the same sequence ever time, so there will be the appearance of
//               randomness WITHIN a run of the program, but no randomness BETWEEN runs.
//               If SEED is specified, we will see the random number generator with the
//               supplied value. If that value is -1, we will seed it with clock().
//
// MULTITHREAD   is also optional, and may only be specified if SEED is given. MULTITHREAD is
//               a single char, used to determine whether to multithread the solution or not.
//               If it is 'n' or 'N', simply call Quicksort on a single thread; otherwise,
//               run it multithreaded.
//
// PIECES        Specifies how many partitions to divide the original list into. The
//               default value is ten. This parameter is also optional, and can only be
//               specified if SEED and MULTITHREAD are given. If MULTITHREAD is 'n' or 'N',
//               then setting PIECES has no effect
//
// MAXTHREADS    Is another optional parameter, which can only be given when PIECES is also
//               specified. It gives the number of threads to attempt to run at once, but
//               MAXTHREADS must be no more than PIECES (we can't run 3 pieces on 4 threads).
```

Your program will begin by creating a (global) array of SIZE integers, and initialize them to 0 – SIZE (i.e., if SIZE is 10, your array would contain {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}). Then your program will scramble these elements (perhaps resulting in {4, 7, 1, 0, 5, 9, 3, 8, 2, 6}). Time how long it takes your program to build the array, fill it, and to scramble it (report all three times, to the nearest 0.001 seconds – see below for details).

From there on, it's a question of putting the values BACK into sorted order, using Quicksort.

Your version of Quicksort will be a hybrid implementation, switching to Shell sort when the size of the segment to be sorted falls below the specified threshold. Shell sort will run with Hibbard's sequence (each interval is one less than a power of 2, starting with the smallest such value that is less than the number of values to sort).

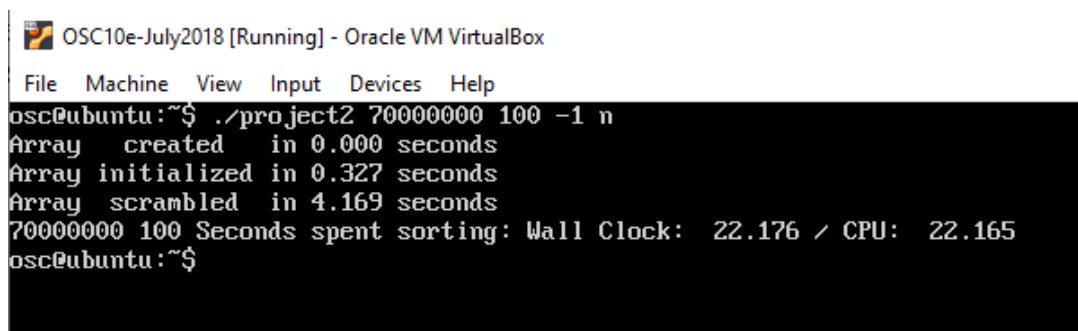
When the size of the interval is less than 2, Quicksort should do nothing – just return.

When the size of the interval is exactly 2, compare the two values, swap them if necessary, and return.

When the size of the interval is > 2 , but $\leq \text{THRESHOLD}$, use Shell sort on that interval

When the size of the interval is $> \text{THRESHOLD}$, partition and recursively Quicksort the two sides using these rules.

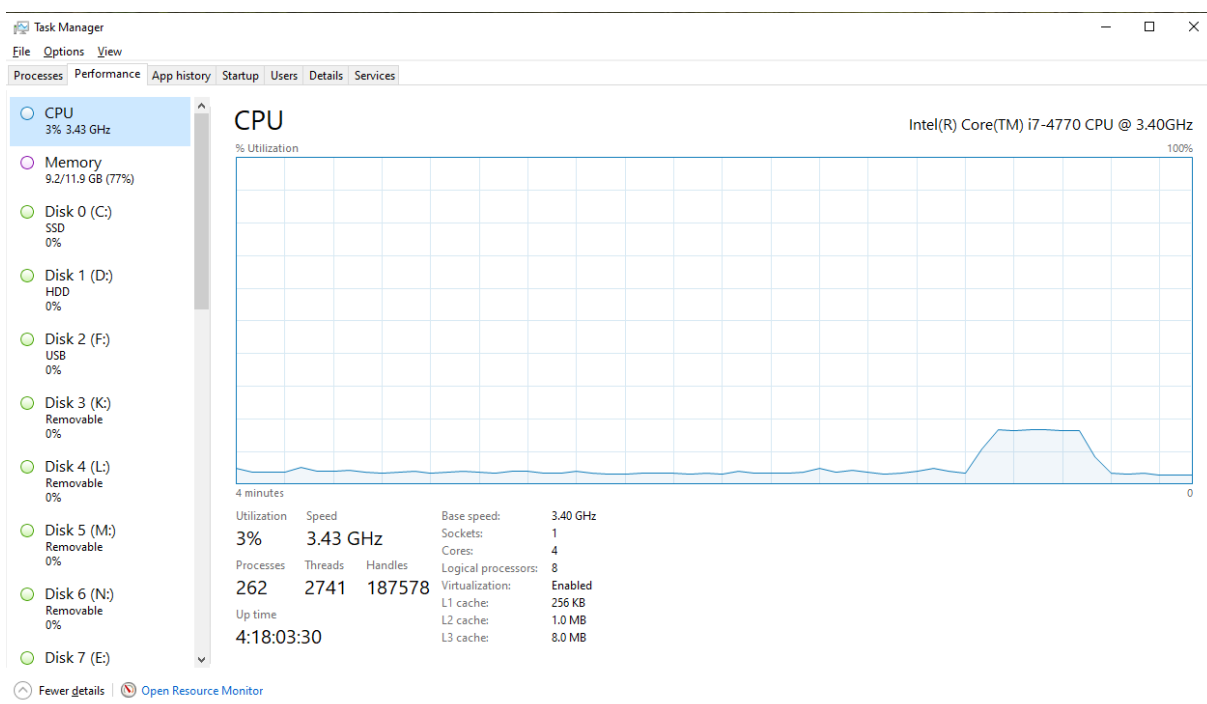
If you invoke the program with multithreading turned off, it will Quicksort the array on one thread. This gives you an opportunity to compare single- and multi-threaded performance:



```
OSC10e-July2018 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
osc@ubuntu:~$ ./project2 70000000 100 -1 n
Array created in 0.000 seconds
Array initialized in 0.327 seconds
Array scrambled in 4.169 seconds
70000000 100 Seconds spent sorting: Wall Clock: 22.176 / CPU: 22.165
osc@ubuntu:~$
```

This screen shot shows sorting 70,000,000 integers, switching to Shell sort when the size of the interval is 100 or less, but using one thread. The resulting “wall clock” and CPU times were essentially identical.

When this instance ran, you can see how the PC's CPU time went up and then came back down:



When your program runs *without* the ‘n’ option for multithreading, it will create multiple threads to do the sorting on. Let’s start with the default values:

By default, it will divide the original array into 10 intervals. First, it will call partition on the whole array. Let’s keep the numbers simple, and assume you’re using 1000 values, and I’ll round off a little here and there, but it should not detract from the idea. If this first partitioning results in a 60/40 split, you will have 600 values on one size and 400 on the other.

Take the larger side, and partition it. Let’s say this one splits 70/30, so now you have 420 and 180 (from the 600-element “big piece” from the first partitioning), and 400 on the other. Choose the 420-element partition, and partition it. Let’s assume 30/70 on that one, so those 420 elements become 126 / 294. Now you have pieces that are 126, 180, 294, and 400 elements – the latter from the original partitioning, and the first three from the subsequent ones.

Continue in this manner, partitioning the largest remaining partition, until you have PIECES segments (by default, this is 10 pieces). The screen shot below shows how the program ran on 100,000,000 integers:

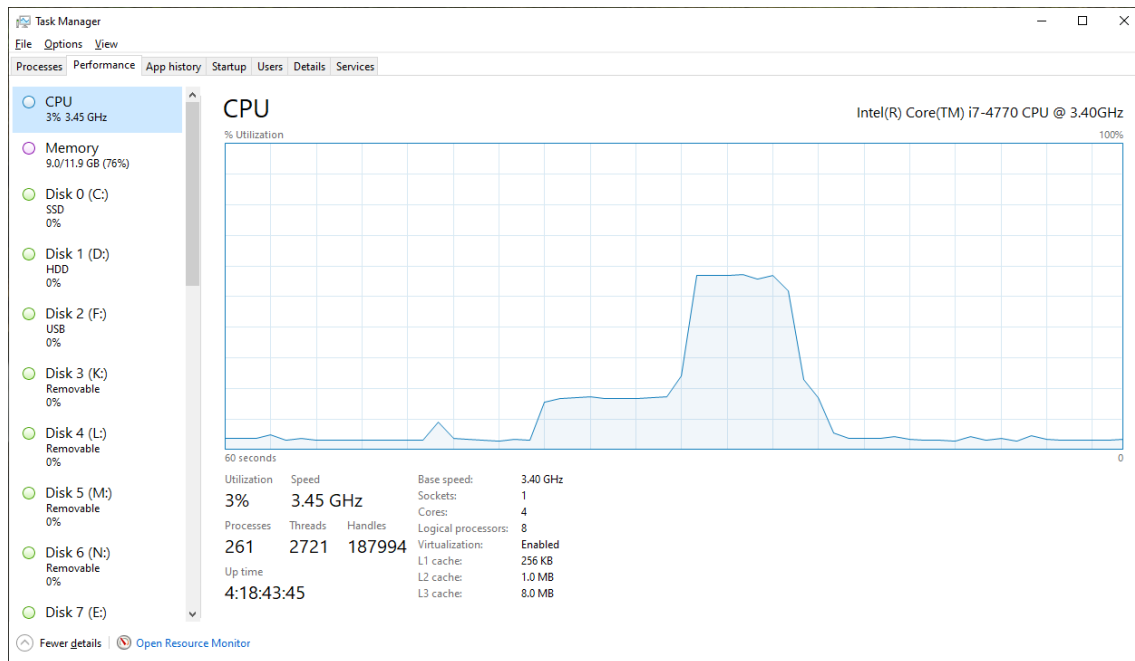
```
Creating multithread partitions
Partitioning      0 - 99999999 (100000000)...result: 72527407 - 27472592 (72.5 / 27.5)
Partitioning      0 - 72527406 ( 72527407)...result: 66660164 - 5867242 (91.9 / 8.1)
Partitioning      0 - 66660163 ( 66660164)...result: 21684455 - 44975708 (32.5 / 67.5)
Partitioning 72527408 - 99999999 ( 27472592)...result: 7023675 - 20448916 (25.6 / 74.4)
Partitioning 21684456 - 66660163 ( 44975708)...result: 12817220 - 32158487 (28.5 / 71.5)
Partitioning      0 - 21684454 ( 21684455)...result: 8219706 - 13464748 (37.9 / 62.1)
Partitioning 34501677 - 66660163 ( 32158487)...result: 15105144 - 17053342 (47.0 / 53.0)
Partitioning 79551084 - 99999999 ( 20448916)...result: 11081906 - 9367009 (54.2 / 45.8)
Partitioning 49606822 - 66660163 ( 17053342)...result: 16408663 - 644678 (96.2 / 3.8)
```

As you can see, sometimes the partition step comes out close to even (47.0/53.0 and 54.2/45.8 – these are percentages), and other times, it’s very uneven (91.9/8.1 and 96.2/3.8). Ideally, Quicksort splits evenly, but as you can see, that doesn’t always happen, even on random data. So, your program will keep “chipping away” at the largest remaining interval. For this example, I wound up with these ten intervals:

#	Lo	Hi	Size	% of Total
1	0	8,219,705	8,219,706	8.2%
2	8,219,707	21,684,454	13,464,747	13.5%
3	21,684,456	34,501,675	12,817,219	12.8%
4	34,501,677	49,606,820	15,105,143	15.1%
5	49,606,822	66,015,484	16,408,662	16.4%
6	66,015,486	66,660,163	644,677	0.6%
7	66,660,165	72,527,406	5,867,241	5.9%
8	72,727,408	79,551,082	7,023,674	7.4%
9	79,551,084	90,632,989	11,081,905	11.1%
10	90,632,991	99,999,999	9,367,008	9.4%

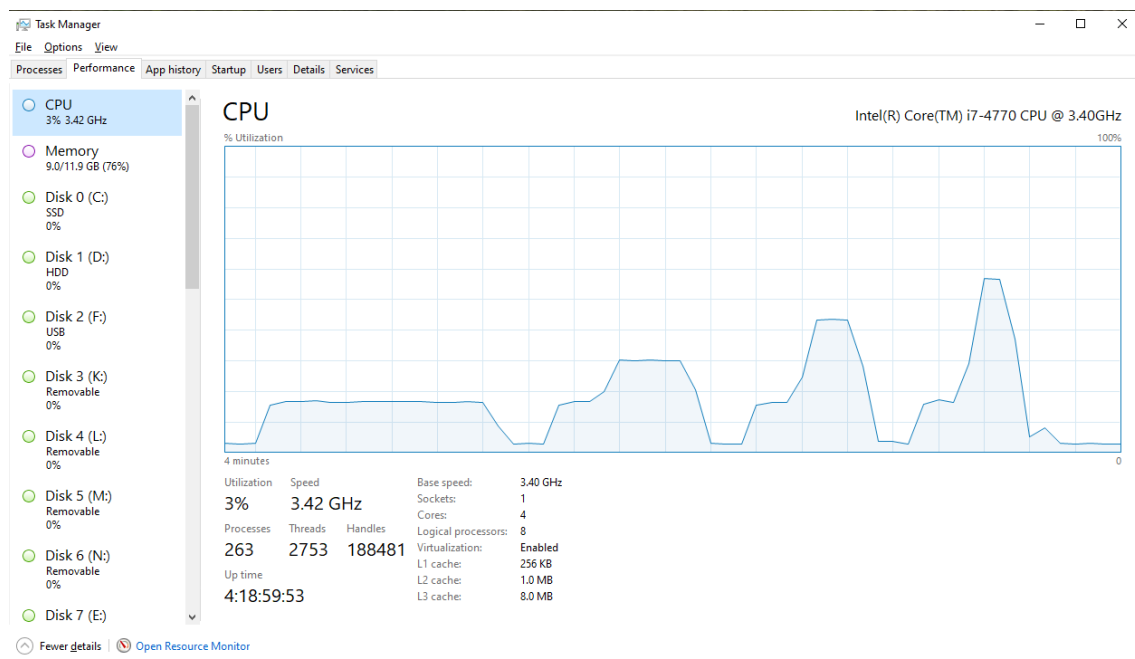
These ten intervals will eventually be run on one thread each, but not all at once. The default value of MAX_THREADS is 4, so we would start THE FOUR LARGEST intervals on four threads. When one of those finishes, we start the next largest, and so on, until they have *all* been started. Then we just wait for them all to finish, report the time taken to partition and sort them, and then (finally) double-check that the sort worked (don’t time the double-check).

When this example runs, the screen shot below of the CPU usage shows what happens – the plateau is the creation, initialization, and scrambling of the data (one thread). The other plateau is when the four threads were running simultaneously. The dip at the right end (the reason it’s not “more squared off”) is because the threads don’t all finish at the same time.



This screen shot shows what happens when the following command lines are executed:

```
./project2 150000000 -1 y 10 1
./project2 150000000 -1 y 10 2
./project2 150000000 -1 y 10 3
./project2 150000000 -1 y 10 4
```



Some pieces you will want to have in your program, or things to consider:

```
#include <pthread.h>
```

You will want to compile your program with: `gcc -g -pthread project2.c -o project2`

To get the “wall clock” time, look up the function `gettimeofday()`.

To get the CPU time taken (by ALL threads in a process), use `clock()`.

We COULD just launch four (or `MAX_THREADS`) threads, and use `join()` to wait for them to all finish, and then launch `MAX_THREADS` more of them, but if one of the threads finishes well ahead of the others (and we’ve already seen that the partition sizes can be SO uneven that it can happen), then we would squander the opportunity to keep one or more of the cores busy, because `join()` waits for ALL threads to complete.

Rather, we will POLL the threads (checking each, one at a time), to see when they complete, and we will launch a new one in its place. So, if you’re starting with four threads, when the first one finishes, start the fifth-largest segment. When another thread finishes, start the sixth-largest, etc.

When we go into a polling loop, we use CPU time just doing the polling. I recommend checking all `MAX_THREADS` threads, and if none have completed, wait 50 ms (look up `usleep()`) before polling again. That way, the next 50 ms can go towards letting the sort threads run, rather than having `main()` keep checking to see if they’ve completed in the microsecond since you just realized that none had.

To see if a thread has completed or not, use `pthread_tryjoin_np()` (this one will cause you to want to `#include <errno.h>`, too).

C doesn’t know about the `bool` data type, so you may need to `#include <stdbool.h>`.

The LAST thing your program should do in `main()` is check to see that the array has been re-sorted. If you started with the value 0 – (`SIZE-1`), it takes only a few lines of code to write `isSorted()`, which you can call on the way out of `main()`. Obviously, if it returns false, you have a problem somewhere!

If you run your program with the same value for `THRESHOLD` as for `SIZE`, then Quicksort doesn’t actually do anything besides let Shell sort process the array, so this is a good way to test that Shell sort is working. If you set `THRESHOLD` to 1, then Shell sort doesn’t get used at all, producing a (non-hybrid) 100%-Quicksort solution.