COMP3230 Principles of Operating Systems

Programming Assignment Two

Due date: Nov. 19, 2022 at 23:59

Total 10 points

(Version: 1.0)

Programming Exercise – multithreading and synchronization

Objectives

1. An assessment task related to ILO 4 [Practicability] – "demonstrate knowledge in applying system software and tools available in the modern operating system for software development".
2. A learning activity related to ILO 2.
3. The goals of this programming exercise are:
   - to have direct practice in designing and developing multithreading programs;
   - to learn how to use POSIX pthreads (and semaphore) libraries to create, manage, and coordinate multiple threads in a shared memory environment;
   - to design and implement synchronization schemes for multithreaded processes using semaphores, or mutex locks and condition variables.

Task

Write a multi-threaded program that parallelizes the samplesort algorithm. The samplesort algorithm is a generalization of the bucketsort algorithm. Unlike quicksort that only selects a single pivot value, samplesort takes a few sample values and uses them to divide the data into buckets and performs the sorting. This makes the algorithm more suitable for parallelization, and a common strategy is to set the number of buckets to the number of threads.

We will provide the seqsort.c file, which makes use of the quicksort library function as the sequential version. The program gives you a framework to generate the data, measure the sorting execution time, and check the correctness of the algorithm. Therefore, this sequential version becomes the baseline for your task. **You will use either the semaphores or (mutex locks & condition variables) to implement a multi-threading version of the samplesort algorithm.**

Samplesort Algorithm

The basic samplesort algorithm can be summarized as follows:

1. Choose ***b*** samples from the data sequence.
2. Sort the samples such that $s_1 < s_2 < s_3 < \cdots < s_b$.

3. Partition the data sequence into ***b*** + 1 subsequences (i.e., buckets) based on the samples, such that every element in the j^{th} subsequence is greater than s_{j-1} and smaller than or equal to s_j . We assume $s_0 = -\infty$ and $s_{b+1} = +\infty$.
4. Sort each subsequence.
5. Concatenate the sorted subsequences to form the sorted data sequence.

To devise a samplesort implementation, one needs to decide on the number of buckets (***b***+1) and a way to choose the ***b*** samples.

Parallelizing Samplesort

Samplesort can be implemented by splitting the sorting of the data sequence (with ***n*** numbers) by multiple threads, where the number of buckets is equal to the number of threads (***p***). Therefore, each thread gets roughly the same size ($^{n}/_{p}$). The parallel samplesort algorithm has five phases.

Phase1 – Each thread gets a subsequence with approximately $^{n}/_{p}$ numbers and uses the quicksort algorithm to sort its subsequence. After the sorting, each thread chooses *p* samples from its sorted subsequence at local indices 0, $^{n}/_{p^2}$, $^{2n}/_{p^2}$, $\cdots$, $^{(p-1)n}/_{p^2}$.

For example, we have a sequence with 40 numbers (i.e., *n* = 40) and the number of threads is four (i.e., *p* = 4).

42	98	2	31	86	87	5	13	99	44	67	37	17	7	87	3	96	71	40	19	58	13	61	77	11	13	6	81	76	18	24	14	63	59	99	17	36	84	1	48

Each thread gets 10 numbers in its subsequence.

42	98	2	31	86	87	5	13	99	44
Thread0									

67	37	17	7	87	3	96	71	40	19
Thread1									

58	13	61	77	11	13	6	81	76	18
Thread2									

24	14	63	59	99	17	36	84	1	48
Thread3									

Each thread sorts its subsequence concurrently.

2	5	13	31	42	44	86	87	98	99
Thread0									

3	7	17	19	37	40	67	71	87	96
Thread1									

6	11	13	13	18	58	61	76	77	81
Thread2									

1	14	17	24	36	48	59	63	84	99
Thread3									

Each thread chooses 4 samples from its subsequence at indices 0, 2, 5, & 7.

2	5	13	31	42	44	86	87	98	99
Thread0									

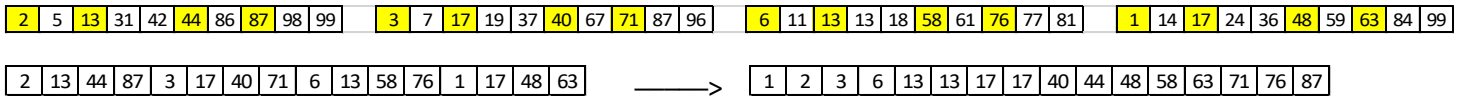
3	7	17	19	37	40	67	71	87	96
Thread1									

6	11	13	13	18	58	61	76	77	81
Thread2									

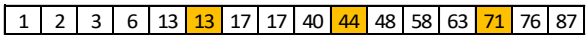
1	14	17	24	36	48	59	63	84	99
Thread3									

Phase2 – One thread gathers all samples and sorts them. Then, it selects *p*-1 pivot values from this sorted list of samples. The pivot values are selected at indices $p + \lfloor {}^{p}/_{2} \rfloor - 1,\ 2p + \lfloor {}^{p}/_{2} \rfloor - 1,\ \cdots\ ,\ (p-1)p + \lfloor {}^{p}/_{2} \rfloor - 1$.

Continue with the above example, one thread gets all samples and sorts them.

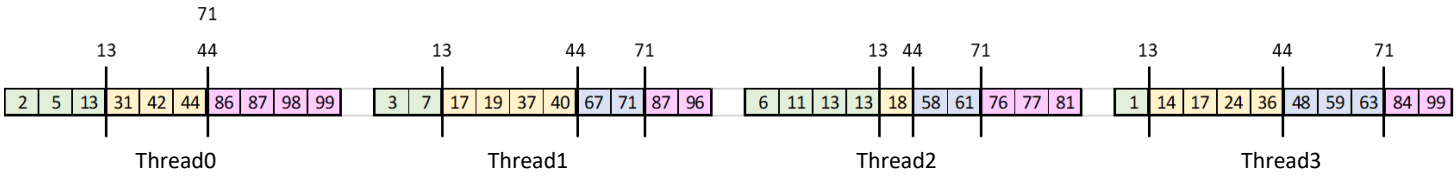


Then the thread selects the pivot values at indices 5, 9, & 13.

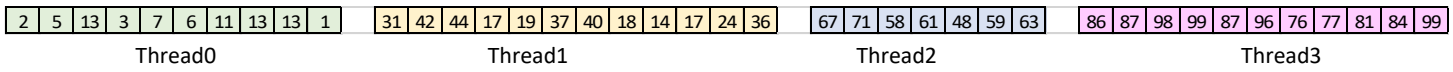


Phase3 - All threads get the *p*-1 pivot values, then each thread partitions its sorted subsequence into *p* disjoint pieces, using the pivot values as separators between pieces.

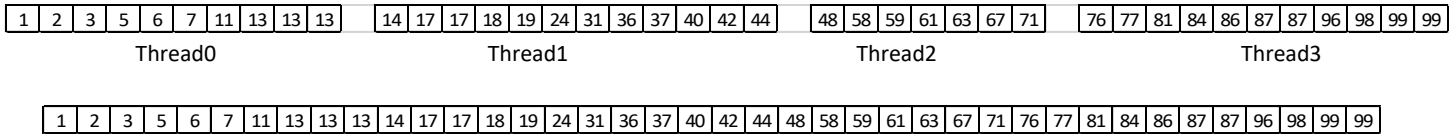
For our example, each thread partitions its subsequence into at most 4 pieces.



Phase4 – Thread ***i*** keeps its i^{th} partition and collects from other threads their i^{th} partitions and merges them into a single sequence.



Phase5 – All threads sort their sequences and concatenate them to form the final sorted sequence.



Programming Tasks

1. Download the sequential program – seqsort.c from the Course's website at moodle.hku.hk. The program accepts one input argument:

   ./seqsort <number>

   where <*number*> represents the number of unsigned integers to be created and sorted by the program.

   To compile the program, use ***gcc***:

gcc seqsort.c -o seqsort

Upon invocation, the program generates a random sequence of numbers using a fixed random seed. This allows the program to generate the same random sequence across different executions of the program. The program uses the qsort() function to sort the sequence. A pair of gettimeofday() function calls are invoked to measure the execution time of the sorting algorithm.

```
// measure the start time
gettimeofday(&start, NULL);

// just call the qsort library
// replace qsort by your parallel sorting algorithm using pthread
qsort(intarr, size, sizeof(unsigned int), compare);

// measure the end time
gettimeofday(&end, NULL);
```

Then the program calls the checking() function to check the correctness of the sorting. In the end, the program prints the total elapsed time.

For example, when invoking the program to sort a sequence of 100000000 numbers, the program prints out the following information:

```
First : 2
At 25%: 536690108
At 50%: 1073542222
At 75%: 1610448550
Last  : 2147483639
Total elapsed time: 19.6378 s
```

The last output line tells us the total execution time of the sorting algorithm. In this case, the qsort() library function. The top five lines tell us about the values at 0%, 25%, 50%, 75%, & 100% of the sorted sequence. The purpose of printing out these values is for checking; in particular, when checking your parallel implementation of the samplesort algorithm.

You are going to implement the samplesort algorithm to replace the qsort() function. Therefore, you should examine the sequential program and understand its execution logic before moving to the pthread implementation.

2. Duplicate a copy of the seqsort.c program and change its name to psort.c. Change the psort.c program to accept one more optional input argument.

   ./psort <number> [<no_of_workers>]

   If the user provides this optional argument, the program should create the number of **worker threads** that equals this input argument; otherwise, the program should create 4 worker threads only (i.e., the default value is 4).

   It is meaningless to just use one worker thread to execute the parallel code; therefore, the program should check that the argument provided by the user must be greater than 1.

3. **Implement the parallel Samplesort algorithm by using multiple threads.**
   Add appropriate pthread functions and synchronization mechanisms (**either semaphores or (mutex locks and condition variables)** to the psort.c program between the two **gettimeofday()** functions.

```
// measure the start time
gettimeofday(&start, NULL);

// add the parallel implementation of the samplesort algorithm here

// measure the end time
gettimeofday(&end, NULL);
```

   Here are some suggestions for the implementation:

   - Add all code statements that are related to the parallelization of the samplesort algorithm between the two gettimeofday() functions.
   - The program creates ***p*** worker threads to perform the sorting and uses the main thread to perform the coordination, selection of pivots, and concatenate the subsequences to form the final sorted sequence; thus, in total, we use ***p*+1** threads.
     - All worker threads concurrently perform all activities related to phase1 and the main thread waits for them to complete.
     - The main thread performs all activities related to phase2 and the worker threads wait for it to complete.
     - All worker threads concurrently perform all activities related to phase3 and phase4. They need a way to synchronize between themselves before exchanging their partitions. The main thread may be involved in the coordination (depends on your algorithm design).
     - All worker threads concurrently sort their subsequences and then terminate. Before termination, they pass the sorted subsequences to the main thread. The main thread waits for all threads to terminate and then merges their subsequences into the sorted sequence.
   - All threads use qsort() to perform local sorting.

   The random sequence is accessible via the pointer ***intarr*** before the sorting and we expect you to use the same pointer *intarr* to point to the final sorted sequence.

   In the end, the psort program should call the checking() function to check the correctness of the data and print the total elapsed time. For example, here is the output of the psort program using 4 worker threads to sort a sequence of 100000000 numbers.

```
First : 2
At 25%: 536690108
At 50%: 1073542222
At 75%: 1610448550
Last  : 2147483639
Total elapsed time: 7.6479 s
```

4. Test your parallel program against the sequential quicksort with different settings on the workbench2 server:

	No. of worker threads	Data size
seqsort	NA	1,000,000, 5,000,000, 10,000,000, 50,000,000, 100,000,000

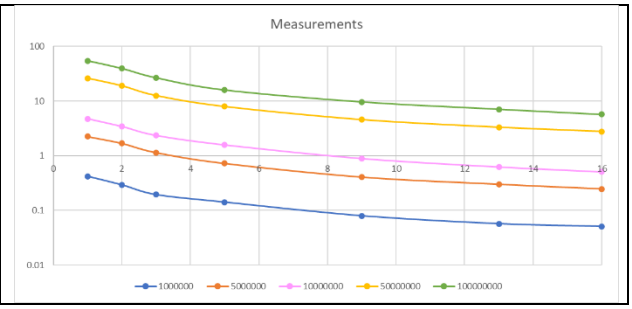
psort	2, 3, 5, 9, 13, 16	1,000,000, 5,000,000, 10,000,000, 50,000,000, 100,000,000

Report the performance of your implementation by completing the following table.

	1000000	5000000	10000000	50000000	100000000
1					
2					
3					
5					
9					
13					
16					

We have prepared a excel file named template.xlsx which contains the measurements of the sequential and parallel program with various no. of threads. It also has a chart helps us to visualize and compare the performance. For example, here is the measured data of the seqsort and psort programs on the Workbench2 server.

	1000000	5000000	10000000	50000000	100000000
1	0.439	2.3223	4.7458	26.8958	55.6706
2	0.2988	1.6035	3.4177	18.9847	39.5712
3	0.1925	1.1939	2.29	12.5512	26.8775
5	0.134	0.7864	1.5368	8.2179	15.7284
9	0.0802	0.4298	0.9064	4.7839	10.1026
13	0.0587	0.2952	0.6171	3.5452	7.356
16	0.0503	0.243	0.5072	3.2652	6.9771



Please download the template.xlsx file from the course's Moodle page and replace the data with your measured data.

Submission

Submit your program to the Programming # 2 submission page at the course's moodle website. Name the program to psort_StudentNumber.c (replace StudentNumber with your HKU student number) and the excel file to measure_StudentNumber.xlsx. As the Moodle site may not accept source code submission, you can compress the C file and xlsx file to the zip format before uploading.

Documentation

1. At the head of the submitted source code, state the
   - File name
   - Student's name and Number
   - Development platform
   - Remark – describe how much you have completed
2. Inline comments (try to be detailed so that your code could be understood by others easily)

Computer platform to use

For this assignment, you can develop and test your program on any Linux/Mac platform, but **you must make sure that the program can correctly execute on the workbench2 Linux server** (as the tutors will use this platform to do the grading). Your program must be written in C and successfully compiled with gcc.

Grading Criteria

1. Your submission will be primarily tested on the workbench2 server. Make sure that your program can be compiled *without any errors*. Otherwise, we have no way to test your submission and you will get a zero mark.
2. As the tutor will check your source code, please write your program with good readability (i.e., with good code convention and sufficient comments) so that you will not lose marks due to confusion.
3. You can use either semaphores or (mutex locks and condition variables) for concurrency and synchronization control. They will be graded without a difference.

Documentation (1 point)	• Include necessary documentation to clearly indicate the logic of the program • Include required student's info at the beginning of the program
Measurement (1 point)	• Measure the performance of the sequential program and your parallel program on the Workbench2 server with various no. of worker threads and data sizes. Complete and submit the measure.xlsx file.
psort.c (8 points)	• The program should be compiled and executed successfully, and the total no. of worker threads created in this program should be equaled to the input parameter.
	• The worker threads must be executed in parallel.
	• Obtain the "correct" results as compared to the sequential program.
	• Can work with different numbers of worker threads and data sizes.
	• The main thread should wait for all worker threads to terminate before displaying the results

Plagiarism

Plagiarism is a very serious offense. Students should understand what constitutes plagiarism, the consequences of committing an offense of plagiarism, and how to avoid it. **Please note that we may request you to explain to us how your program is functioning as well as we may also make use of software tools to detect software plagiarism.**