**SOFTENG 370 Assignment 1 Report**

**Name: Jinkai Zhang**

**UPI: Jzha541**

**Answer to Questions**

1. Install Ubuntu 20.04.2 LTS on PC.

Number of CPUs: 4

Total amount of memory: 8036648 kilobytes

Version of the operating system: Ubuntu 20.04.2 LTS

OS: GNU/Linux

Kernel Version: #51~20.04.1 – Ubuntu

Architecture: x86\_64

1. The MAX parameter number which I found is 30. This is because of the memory pressure if over 30 it will have error of “Unable to allocate space for data”.

In the report, I will use **27** because of the time pressure. Overt 27 for some steps will take a very long waiting time.

1. The **times()** function get the process times and the result will return the number of CPU clock ticks for an arbitrary period. In this code, we print out **tms\_utime** which is the clock ticks only for the user process.

The **gettimeofday()** function get the current time the result is stored in a **timeval** struct and return in seconds and microseconds for an arbitrary period. This is the real time which includes all process running in the system(both user&kenel).

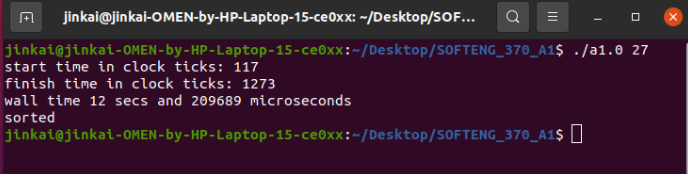
1. The input 24 and 25 means the size of the array is (2^24) \* sizeof(int) = 67108864 bytes and 2(2^25) \* sizeof(int) = 134217728 bytes. The size of input 25 is twice of the input 24.

Since there is only 3 GiB of memory for the VM, the free memory for running this process is quite small. For a1.0 we process everything in one thread, there is only one CPU core to handle the task. For input 24, a lot of data can store in the RAM directly, so we do not need to perform too many times of system call to swap out/in data between RAM and hard disk. The time we use is only 2 more seconds. For input 25, the array size is far over the capacity of RAM, a lot of virtual memory will be used. There will be a lot of system call to swap out/in the data and it is quite time consuming. Since there is only one thread, so the processing time is much longer than 24.

**Analysis on each step**

**Step 0: The base case which use as a reference to compare with others.**

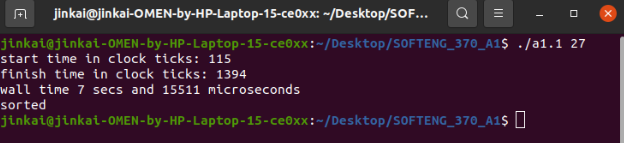
**Implementations:** The given step which runs the hybrid sort in one thread. The code runs all the works in just one thread. There is only one CPU core performing the process. Therefore, the usage of the CPU core is almost 100% during the processing period.

**Figures:**

**Step 1: Use two threads to perform the sort.**

**Implementations:** Divide the array into two parts and create a new thread to handle one part. The main thread runs the other part. We use pthread which is a system call to create thread. Therefore, both thread can be scheduled on different processors (Appendix S1) and run at the same time. The result is almost half time of step 0 as we use two threads. Since the two threads share the same memory, we do not need to worry about data transfer. One CPU core start to work early than the other, because the one(CPU3) is the caller thread that need to create the new thread.

**Figures:**

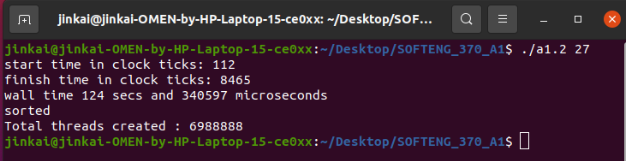


**Step 2: Use a new thread every time you call merge\_sort.**

**Implementations:** Inside the merge\_sort function, create a new thread after it divides the block into two parts. Assign the new thread to run merge\_sort again with the left part and let the caller thread run the merge\_sort with the right part. Then join the new thread to the caller thread. If the join is failed, it will perform the merge\_sort with the left part in the caller thread.

Because we call the function merge\_sort recursively, if the array size is large, we will call the function so many times. Each time we call the function, we try to create a new thread and join it. The cost of the system call is very expensive. Therefore, the time using is much longer then two threads. All for CPU cores are working on a high percentage all the time.

**Figures:**

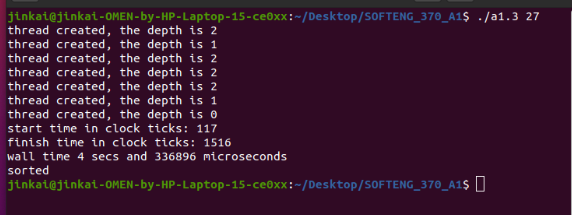


**Step 3: Use 8 threads to perform the sort.**

**Implementations:** Create 7 new threads by put the pthread\_create inside an if statement which will limit the calling time of pthread\_create. Create a new struct and put an integer depth with struct block together. In the merge\_sort function, increase the depth value on the one half (left block) by 1 after we divide the array into two parts. Therefore, the first time we invoke merge\_sort we will have a new thread, the second time both the caller thread and callee thread will create a new thread. After 3 time we will have 8 threads in total. By using this method, each time the new thread will assign half amount of work than last time. The workload is not evenly distributed to 8 threads. Four CPU cores will be used as we have 8 system threads. All 8 threads will share the same memory and work at the same time after they all be created.

The time using is much less than two threads. As the computer only has four CPUs, only use four threads will have a slight better performance.

**Figures:**

****

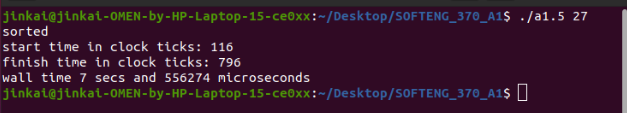
CPU History in Appendix – S3.

Step 4:

**Step 5: Use two processes rather than two threads.**

**Implementations:** Instead of using pthread, we use fork to create a new process. This duplicates the currently running process. The two process are not sharing memory. We need to use pipe to transfer the data between them. As I choose 27 as input, the array size is larger than the maximum capacity of the pipe (which is 65536 on my Linux). I divide the array into maximum sizes of chunks and use a for loop to do the write/read. In the main function, I divide the array into two parts. Let the child process to sort one and transfer the data back to parent process through pipe. Two processes method is slightly slower than two threads because they are not share memory. The pipe operation will take times.

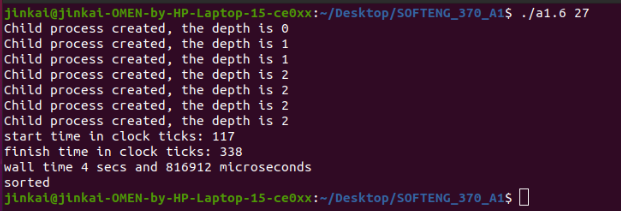
**Figures:**

****

**Step 6: Use 8 processes rather than 8 threads.**

**Implementations:** The method that uses to create 8 processes is basically the same as creating 8 thread. Also because of the data transferring, the 8 processes are slightly slower than 8 threads.

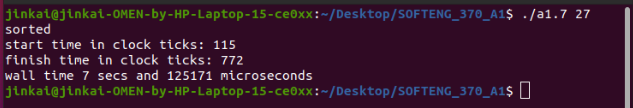
**Figures:**



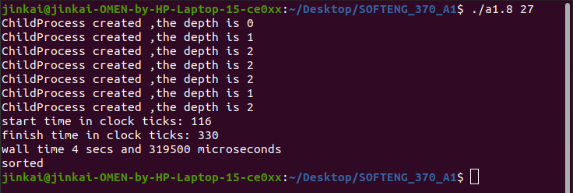
**Step 7&8: Base on step 5&6 and share the memory.**

**Implementations:** Instead of using **calloc** to allocate and duplicate the memory for each process. We use **mmap** to share the memory for all processes. Let the parent process to wait for the child process to finish and then merge the array. Because the memory sharing does not require inter-process communication(pipe), the running time is slightly faster than the step 5&6. The behaviour of sharing memory multi-processes is close to multi-threads. Therefore, the running times are also very similar.

**Figure 7:**

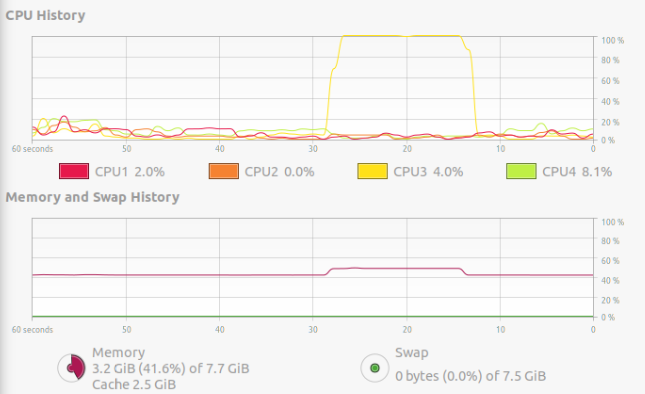


**Figure 8:**

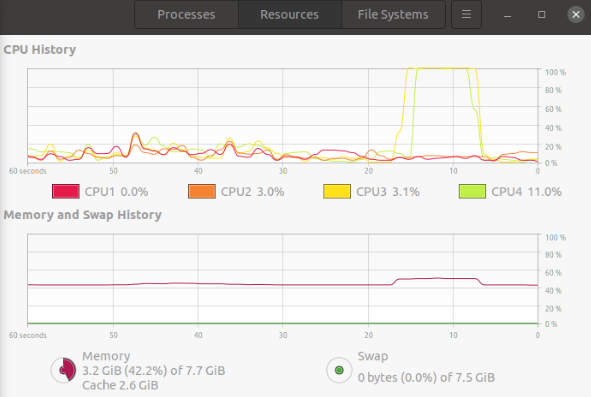


**APPENDIX – CPU History**

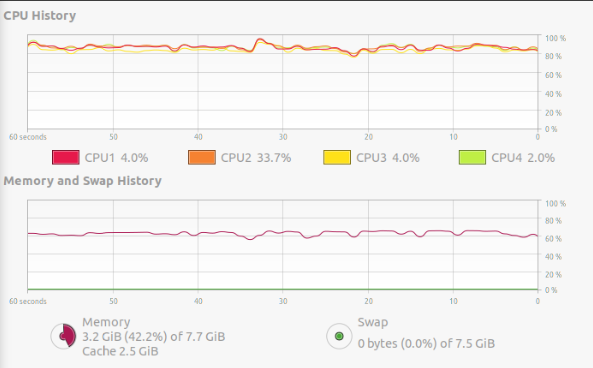
STEP 0:

****

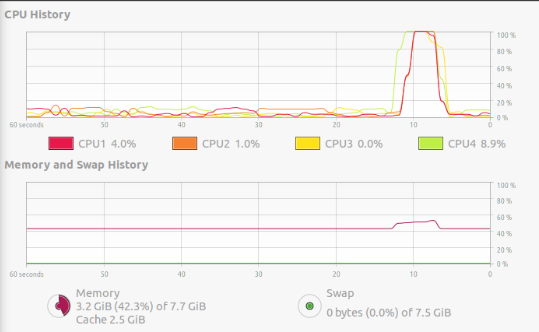
STEP 1:



STEP 2:

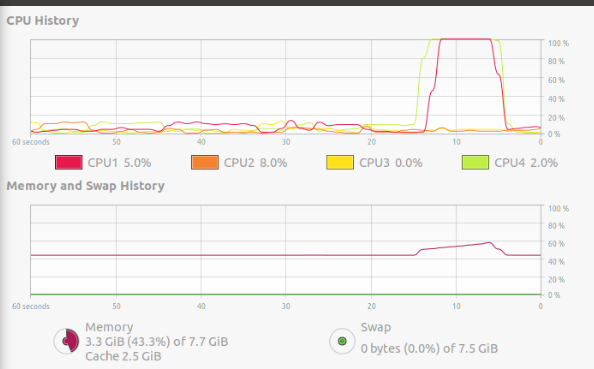


STEP 3:

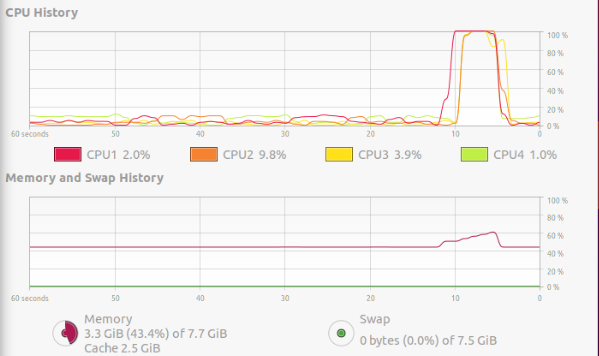
****

STEP 4:

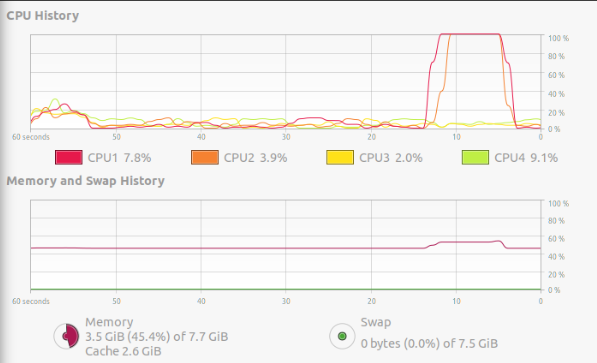
STEP 5:



STEP 6:



STEP 7:



STEP 8:

