

**Lab report**

|  |  |
| --- | --- |
| **Course**: | Operating System Principle |
| **Semester**: | 2nd semester of the academic year **2020-2021** |
| **Major**: | Software Engineering |
| **Class**: | 2019 |
| **Student Name**: | Ruoxuan Fu |
| **Student ID:** | 222019321062060 |
| **Teacher:** | ZHAO, Hengjun (赵恒军) |

**School of Computer and Information Science**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | | Pthread Library and Concurrent Programming | | | |
| Date | | April，2021 | Type | | √ Confirmatory  √ Design  √ Comprehensive |
| 1. **Objective & Requirements**    1. Grasp the Pthreads API for thread creation, termination operations    2. Grasp concurrent programming skills | | | | | |
| 1. **Experimental environment (**platform and software**)**   Virtualbox + Ubuntu (or other platform+linux system combinations) | | | | | |
| 1. **Experimental content and design** (Main Content, Procedure, Codes and Results) 2. Task1    * 1. Create a new thread in the main thread      2. Pass to integers to the new thread and calculate the sum of the two integers by the new thread 3. Task2    * 1. Define an integer array of length 200000      2. Randomly initialize the integer array      3. Sort the initialized integer array and measure the time cost (hint: use the time command) 4. Task3    * 1. Write a C program to merge two sorted integer arrays in to a single sorted integer array 5. Task4   Write a multithreaded sorting program that works as follows:   * + 1. Set the number of CPUs of your virtual machine to at lease 2 in VirtualBox     2. Define two GLOBAL integer arrays **a** and **b**, both of length 200000     3. Randomly initialize the array **a**     4. In the main thread, create two new threads to sort the first half and the second half of array **a** respectively     5. The main thread waits for the two new threads to terminate, and then merge the sorted first and second half of array **a** into array **b**     6. Compare the time cost of your multithreaded program with the time you obtained in Task 2, and compute the speedup. (hint: **time** command)      1. Please provide your procedure and source codes to perform the tasks.   Task1:   1. To pass integers to runner function, the solution is to pass the pointer pointing to an array      1. Compile and run     Task2:   1. First I should write a sort function to complete the sort task      1. And then use pthread.h and sort.h to generate a random array      1. Compile and run     Task3:   1. First is to make two sorted array and pass these two arrays and their length to the sort function      1. In sort function, I use two flags to point current position of two arrays. After comparing these two arrays’ elements, I store the smaller elements in a temporary array recursively. And finally assign the elements of the temporary array to arr1.      1. Compile and run     Task4:   1. First is to adjust the cpu cores to at least two      1. Write a header file to contain the sort algorithm      1. Write a c source file to contain two global arrays. In this file, I initialize the first array and split it into two temporary arrays to sort in two new thread. After sorting them, use the merge function used in task3 to merge two sorted arrays.   main.c   1. #include <pthread.h> 2. #include <stdio.h> 3. #include <stdlib.h> 4. #include <time.h> 5. #include "QuickSort.h" 6. #include "MergeSort.h" 7. #define SIZE 200000 8. #define MAX 10 9. **int** a[SIZE]={0},b[SIZE]={0}; 11. **void** initElem() 12. { 13. srand((unsigned)time(NULL)); 14. **for**(**int** i = 0;i < SIZE;i++) 15. a[i] = rand() % MAX; 16. } 18. **void** \*runner(**void** \*param) 19. { 20. **int** \*arr = (**int**\*) param; 21. quick\_sort(arr,0,SIZE/2); 22. } 24. **int** main() 25. { 26. initElem(a); 27. **int** temp1[SIZE/2],temp2[SIZE/2]; 28. **for** (**int** i = 0; i < SIZE/2; i++) 29. { 30. temp1[i] = a[i]; 31. } 32. **for** (**int** i = 0,j = SIZE / 2; i < SIZE/2,j<SIZE; i++,j++) 33. { 34. temp2[i] = a[j]; 35. } 36. pthread\_t tid1,tid2; 37. pthread\_attr\_t attr; 38. pthread\_create(&tid1,&attr,runner,temp1); 39. pthread\_create(&tid2,&attr,runner,temp2); 40. pthread\_join(tid1,NULL); 41. pthread\_join(tid2,NULL); 42. merge\_sort(temp1,temp2,SIZE/2,SIZE/2); 43. **return** 0; 44. }   MergeSort.h   1. **void** merge\_sort(**int** \*arr1,**int** \*arr2,**int** len1,**int** len2) 3. **int** NewArr[len1+len2]; 4. **for**(**int** i = 0;i < len1+len2;i++) 5. NewArr[i] = 0; 6. **int** p1 = 0,p2 = 0; 7. **int** curr; 8. **while**(p1 < len1 || p2 < len2) 9. { 10. **if**(p2 == len2) 11. curr = arr1[p1++]; 12. **else** **if**(p1 == len1) 13. curr = arr2[p2++]; 14. **else** **if**(arr1[p1] < arr2[p2]) 15. curr = arr1[p1++]; 16. **else** 17. curr = arr2[p2++]; 18. NewArr[p1+p2-1] = curr; 19. } 20. **for**(**int** i = 0;i != len1+len2;i++) 21. \*(arr1+i) = \*(NewArr+i);   QuickSort.h   1. **void** swap(**int** a,**int** b) 2. { 3. **int** temp = b; 4. b = a; 5. a = temp; 6. }  9. **void** quick\_sort(**int** s[], **int** l, **int** r) 10. { 11. **if** (l < r) 12. { 13. swap(s[l], s[(l + r) / 2]); 14. **int** i = l, j = r, x = s[l]; 15. **while** (i < j) 16. { 17. **while**(i < j && s[j] >= x) 18. j--; 19. **if**(i < j) 20. s[i++] = s[j]; 22. **while**(i < j && s[i] < x) 23. i++; 24. **if**(i < j) 25. s[j--] = s[i]; 26. } 27. s[i] = x; 28. quick\_sort(s, l, i - 1); 29. quick\_sort(s, i + 1, r); 30. } 31. } 32. Compile and run it. Compare two results and calculate the speedup.   Task4    Task2    The speed up is 4.911/1.712 = 2.87(quoted in two decimal places.) | | | | | |
| 1. **Result analysis and discussion**（Analysis of experimental results and summing up the harvest and the existing problems）   Result analysis:  The results of the experiment were more satisfactory. The first experiment was able to calculate the sum of two numbers correctly. The second experiment was able to correctly sort randomly generated arrays with a low time complexity. The third experiment was able to correctly output the result of reordering two ordered arrays after merging them. The fourth experiment clearly shows a significant reduction in execution time for the multi-threaded program compared to the single-threaded program  Harvest:  A thread is the smallest unit in an operating system that performs operations. It can be considered as a subtask of the process and the unit in the process that actually performs the operations. A process can have multiple concurrent threads. Threads of the same process share all the system resources of that process, such as the process's heap. An example is that different threads of the same process share global variables, but not local variables. The reason is that each thread has its own program counter, register environment and stack. While local variables are stored in the stack, global variables are stored in static storage.  The subsumption sort is a very typical application of using the partitioning method. The already ordered subsequences are combined to obtain a completely ordered sequence; that is, each subsequence is first ordered, and then the subsequence segments are ordered between them. The specific operation can be done either by merging the subsequences and reordering them or by using two pointers to the two subsequences to continuously compare them and deposit them into the total sequence.  The purpose of the Linux time command is to measure information about the time and system resources required to execute a specific command. In the output, "real" is the actual time, "user" is the user CPU time, and "sys" is the system CPU time.  Existing problems:  None | | | | | |
| Comments & Evaluation | Content & Design (A-E) | | |  | |
| Procedure & Codes (A-E) | | |  | |
| Results (A-E) | | |  | |
| Analysis & Discussion (A-E) | | |  | |
| Score (A-E):  Feedback comments: | | | | |