MULTI-THREADED SORTING APPLICATION

(Project Group: 16)

Team Members:

Bharathi Ranganathan (010687690)

Keerthanaa Alagudurai (010713898)

Meenu Ganesh (010726625)

Course: CMPE 180-94 Fall 2015

Date: 11/19/2015

1. Executive Summary:

In this project we use threads to sort a list of integers. Initially the list is divided into two sub-lists. Each of the sub-lists is sorted using separate threads using selection sort. Once these threads are sorted, a third thread merges the two lists into a single sorted list.

Since global data is shared across all the threads, the array to be sorted and the array in which the sorted elements are placed are established as global arrays. The threads that sort both the halves work simultaneously. Whereas, the thread that merges both the sorted arrays, merges both the sorted arrays once the sorting is done.

2. Introduction:

Modern operating systems support the concept of thread control. Threads are lightweight processes, which has its own, thread ID, a program counter, stack and register set. Threads of the same process share text section and data section. Multi-threading enhances the parallelism of a process and thus makes the execution faster. And, because of this reason, we have chosen to code the sorting application using multi-threading.

For this project, we will be writing the code in C language and will be using pthreads in UNIX environment. The starting point and the ending point are specified using structures.

3. Background and objectives:

We learnt about processes and threads; and the role that they play in the execution of a program. Processes are active whereas programs are passive. Furthermore, a process can be split into multiple threads; so that each thread is a mini process. A single threaded application can execute only one process at a time; whereas a multi-threaded application can implement parallelism. Because of this, we are being able to achieve such a high execution speed.

Thus, we decided to implement our program using multi-threading. Our project aims at synchronizing the threads in order to avoid race conditions and unpredictable results.

4. Approach and methodology:

Our discussion commenced with the framing of pseudo code. Eventually, the project phase was switched to developing in which we created two child threads using pthread_create function and sorted the sub lists using selection sort. The two threads are synchronized using pthread_join. Here, the parent thread was waiting for the child threads; once pthread_join was invoked, the child threads joined the parent thread. A third child thread was created to merge the sub-lists to a new array of sorted list. And, at the end the third child thread also joined the parent thread. Since we had to pass both the start and end positions of the sub-array that we had to sort; we made use of a structure whose data members are the start and end position of the sub-array. Also, since, the sorting of one half was independent of the other; we could achieve data parallelism by sorting both the sub-arrays at the same time.

```
/*
 Program to do multithreaded sorting
 */
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int unsorted_array[10] = {100,350, 10, 3, 150,1000, 4 , 0, 96,
28};
int sorted_array[10] = {0};

/*
 Structure to know the positons of the array that we need to
access.
 */
struct array_pos{
    int start_pos;
    int end_pos;
}positions_first, positions_second;

void *selection_sort(void* positions);
void *merge_sort(void*);

int main(){

    pthread_t p_thread[3];
    int thr_id;
    int status;
    printf("The unsorted array is:\n");
    for(int i=0;i<10;i++){
        printf("%d\n",unsorted_array[i]);
    }

    positions_first.start_pos = 0;
    positions_first.end_pos = 5;
    thr_id = pthread_create(&p_thread[0], NULL, selection_sort,
(void*) &positions_first);
    if(thr_id < 0)
    {
        perror("Thread create error.");
        exit(0);
    }
    positions_second.start_pos = 5;
    positions_second.end_pos = 10;
    thr_id = pthread_create(&p_thread[1], NULL, selection_sort,
(void*) &positions_second);
    if(thr_id < 0)
    {
        perror("Thread create error.");
        exit(0);
    }
    pthread_join(p_thread[0], NULL);
    printf("return thread 0: %d\n", status);
    pthread_join(p_thread[1], NULL);
    printf("return thread 1: %d\n", status);
    printf("The array after sorting of two halves: \n");
    for(int i = 0; i<10;i++){
        printf("%d \n" , unsorted_array[i]);
    }

    thr_id = pthread_create(&p_thread[2], NULL, merge_sort,
(void*) NULL);
    printf("return thread 2: %d\n", status);
    pthread_join(p_thread[2], NULL);
    printf("The sorted array is:\n");
    for(int i=0;i<10;i++){
        printf("%d\n",sorted_array[i]);
    }

    return 0;
}
/*
 Function to perform selection sort.
 */
void *selection_sort(void *positions){
    long tid;
    tid = (long)pthread_self();
    printf("Thread ID: (%lu) \n", tid);
    int temp = 0;
    struct array_pos* pos;
    pos = (struct array_pos*) positions;
    for(int i = pos->start_pos; i < pos->end_pos; i++){
        for(int j= i+1; j< pos->end_pos; j++){
            if(unsorted_array[i]> unsorted_array[j]){
                temp = unsorted_array[i];
                unsorted_array[i] = unsorted_array[j];
                unsorted_array[j] = temp;
            }
        }
    }
    return NULL;
}
/*
 Function to merge the two sorted arrays
 */
void *merge_sort(void *arg){

    long tid;
    tid = (long)pthread_self();
    printf("Thread ID: (%lu) \n", tid);
    int a,b,c;
    for(a=0,b=5,c=0;(a<5 && b<10);){
     if(unsorted_array[a] < unsorted_array[b]){
          sorted_array[c++]=unsorted_array[a++];
     }
     else{
          sorted_array[c++]=unsorted_array[b++];
     }

    }
    if(a<5){
     while(a<5){
     sorted_array[c++]=unsorted_array[a++];
     }
    }
    else{
     while(b<10){
          sorted_array[c++]=unsorted_array[b++];
     }
    }
    return NULL;
}
```

5. Finding and analysis:

We observed that their respective threads are sorting the two sub-arrays simultaneously. Those threads have their own unique thread ids. Since the third child thread is being created only after the termination of the first two child threads, the thread id of the first child thread is being reused. Also, we observed that there were no synchronization issues such as race conditions and unpredictable results since pthread_join was used.

Here is the output of the program

```
The unsorted array is:
100
350
10
3
150
1000
4
0
```

```
96
28
Thread ID: (123145302839296)
Thread ID: (123145303375872)
return thread 0: 0
return thread 1: 0
The array after sorting of two halves:
3
10
100
150
350
0
4
28
96
1000
return thread 2: 0
Thread ID: (123145302839296)
The sorted array is:
0
3
4
10
28
96
100
150
350
1000
```

6. Conclusions and Recommendation:

It is evident from the above sorting application which we implemented that multi-threading executes faster since it runs several tasks simultaneously. Hence, it is recommended to use multi-threading for applications that involves parallelism.

7. References

a) Operating system concepts by Abraham Silberschatz
b) Data Structures using C++ by D.S.Malik
c) http://man7.org/linux/man-pages/man3/pthread_create.3.html
d) http://man7.org/linux/man-pages/man3/pthread_join.3.html