Concurrent Merge Sort in Shared Memory

Given a number 'n' and a n numbers, sort the numbers using **Concurrent** Merge Sort. (Hint: Try to use shmget, shmat system calls).

Part1: The algorithm (HOW?)
Recursively make two child processes, one for the left half, one of the right half. If the number of elements in the array for a process is less than 5, perform a Insertion Sort. The parent of the two children then merges the result and returns back to the parent and so on. But how do you make it concurrent?

Part2: The logical (WHY?)
The important part of the solution to this problem is not algorithmic, but to explain concepts of Operating System and kernel.
To achieve concurrent sorting, we need a way to make two processes to work on the same array at the same time. To make things easier Linux provides a lot of system calls via simple API endpoints. Two of them are, **shmget()** (for shared memory allocation) and **shmat()** (for shared memory operations). We create a shared memory space between the child process that we fork. Each segment is split into left and right child which is sorted, the interesting part being they are working concurrently! The shmget() requests the kernel to allocate a shared page for both the processes.

Why traditional fork() does not work?
The answer lies in what fork() actually does. From the documentation, "fork() creates a new process by duplicating the calling process". The child process and the parent process run in separate memory spaces. At the time of fork() both memory spaces have the same content. Memory writes, file-descriptor(fd) changes, etc, performed by one of the processes do not affect the other. Hence we need a shared memory segment.

```
// C program to implement concurrent merge sort
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void insertionSort(int arr[], int n);
void merge(int a[], int l1, int h1, int h2);

void mergeSort(int a[], int l, int h)
{
    int i, len=(h-l+1);

    // Using insertion sort for small sized array
    if (len<=5)
    {
        insertionSort(a+l, len);
        return;
    }

    pid_t lpid,rpid;
    lpid = fork();
    if (lpid<0)
    {
        // Lchild proc not created
        perror("Left Child Proc. not created\n");
        _exit(-1);
    }
    else if (lpid==0)
    {
        mergeSort(a,l,l+len/2-1);
        _exit(0);
    }
    else
    {
        rpid = fork();
        if (rpid<0)
        {
            // Rchild proc not created
            perror("Right Child Proc. not created\n");
            _exit(-1);
        }
        else if(rpid==0)
        {
            mergeSort(a,l+len/2,h);
            _exit(0);
        }
    }

    int status;

    // Wait for child processes to finish
    waitpid(lpid, &status, 0);
    waitpid(rpid, &status, 0);

    // Merge the sorted subarrays
    merge(a, l, l+len/2-1, h);
}

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++)
   {
       key = arr[i];
       j = i-1;

       /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
       while (j >= 0 && arr[j] > key)
       {
           arr[j+1] = arr[j];
           j = j-1;
       }
       arr[j+1] = key;
   }
}

// Method to merge sorted subarrays
void merge(int a[], int l1, int h1, int h2)
{
    // We can directly copy  the sorted elements
    // in the final array, no need for a temporary
    // sorted array.
    int count=h2-l1+1;
    int sorted[count];
    int i=l1, k=h1+1, m=0;
    while (i<=h1 && k<=h2)
    {
        if (a[i]<a[k])
            sorted[m++]=a[i++];
        else if (a[k]<a[i])
            sorted[m++]=a[k++];
        else if (a[i]==a[k])
        {
            sorted[m++]=a[i++];
            sorted[m++]=a[k++];
        }
    }

    while (i<=h1)
        sorted[m++]=a[i++];

    while (k<=h2)
        sorted[m++]=a[k++];

    int arr_count = l1;
    for (i=0; i<count; i++,l1++)
        a[l1] = sorted[i];
}

// To check if array is actually sorted or not
void isSorted(int arr[], int len)
{
    if (len==1)
    {
        printf("Sorting Done Successfully\n");
        return;
    }

    int i;
    for (i=1; i<len; i++)
    {
        if (arr[i]<arr[i-1])
        {
            printf("Sorting Not Done\n");
            return;
        }
    }
    printf("Sorting Done Successfully\n");
    return;
}

// To fill randome values in array for testing
// purpise
void fillData(int a[], int len)
{
    // Create random arrays
    int i;
    for (i=0; i<len; i++)
        a[i] = rand();
    return;
}

// Driver code
int main()
{
    int shmid;
    key_t key = IPC_PRIVATE;
    int *shm_array;
    // Using fixed size array.  We can uncomment
    // below lines to take size from user
    int length = 128;

    /* printf("Enter No of elements of Array:");
    scanf("%d",&length); */

    // Calculate segment length
    size_t SHM_SIZE = sizeof(int)*length;

    // Create the segment.
    if ((shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666)) < 0)
    {
        perror("shmget");
        _exit(1);
    }

    // Now we attach the segment to our data space.
    if ((shm_array = shmat(shmid, NULL, 0)) == (int *) -1)
    {
        perror("shmat");
        _exit(1);
    }

    // Create a random array of given length
    srand(time(NULL));
    fillData(shm_array, length);

    // Sort the created array
    mergeSort(shm_array, 0, length-1);

    // Check if array is sorted or not
    isSorted(shm_array, length);

    /* Detach from the shared memory now that we are
       done using it. */
    if (shmdt(shm_array) == -1)
    {
        perror("shmdt");
        _exit(1);
    }

    /* Delete the shared memory segment. */
    if (shmctl(shmid, IPC_RMID, NULL) == -1)
    {
        perror("shmctl");
        _exit(1);
    }

    return 0;
}
```

Run on IDE

Output:

```
Sorting Done Successfully
```

Performance improvements?

Try to time the code and compare its performance with the traditional sequential code. You would be surprised to know that sequential sort performance better!

When, say left child, access the left array, the array is loaded into the cache of a processor. Now when the right array is accessed (because of concurrent accesses), there is a cache miss since the cache is filled with left segment and then right segment is copied to the cache memory. This to-and-fro process continues and it degrades the performance to such a level that it performs poorer than the sequential code.

There are ways to reduce the cache misses by controlling the workflow of the code. But they cannot be avoided completely!

This article is contributed by **Pinkesh Badjatiya** .If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Company Wise Coding Practice   Topic Wise Coding Practice

1 Comment Category: Project Strings

Related Posts:

- Creating a Calculator for Android devices
- Creating a C/C++ Code Formatting tool with help of Clang tools
- Implementation of Minesweeper Game
- OpenCV C++ Program for coin detection
- Basic Graphic Programming in C++
- Cartooning an Image using OpenCV – Python
- Creating a PortScanner in C
- Creating a Proxy Webserver in Python | Set 2

(Login to Rate and Mark)

3.6 Average Difficulty : **3.6/5.0**
Based on **3** vote(s)

☐ Add to TODO List
☐ Mark as DONE

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.