REPORT
QUESTION 1

Submitted by : Tanvi Kamble
2018114004

Aim: Implementing a Concurrent version of Quicksort algorithm

Traditional Quick Sort: We take any random element (the last element of the array in this case) as the pivot and move all the elements greater that the pivot to its right and others to the left. Then divide this array into two subarrays one containing elements lesser than pivot and the other one containing the ones greater than or equal to, and repeat the same process until the subarrays can't be divided

CONCURRENCY

- ➔ We use two API endpoints namely shmget() and shmat()
- ➔ **shmget()** used for shared memory allocation.
- ➔ **shmat()** used 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 individually yet concurrently.
- ➔ We use shared memory spaces as normal fork() splits the process into two different processes independently, but we want concurrency!

Code:

- ➢ Int main():

```
int shmid;
key_t key = IPC_PRIVATE;
int *share_arr;
int input_arr[100];
int len;
```

- Shmid = id of the new shared memory
- Share_arr = the array belonging to the shared memory
- Input_arr = array given by user that needs to be sorted
- Len = length of the input array

```
if ((shmid = shmget(key, sh_sz, IPC_CREAT | 0666)) < 0)
    {
        perror("shmget");
        _exit(1);
    }
```

- Create a private memory (key) of size sh_mz (space occupied by the array)

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

- This attaches the segment that we created to our data space in the form of share_arr

➢ void quicksort(int arr[], int i, int j):

- Function used for quicksort, parameters are the array to be sorted, and the range from i to j that needs to be sorted

```
if (i < j)
    {
        int l = j - i + 1;
        if (l <= 5)
        {
            insertionSort(arr, i, j);
            return;
        }
```

- If the range ('l') is less than 5 we do insertion sort to sort the elements

```
        int part = partition(arr, i, j);
```

- Part is the pivot from which the array needs to be subdivided

```
        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)
{
    quicksort(arr, i, i + part - 1);
    _exit(0);
}
```

- Create a left child process that sorts the left subarray from the pivot

```
else
{
    rpid = fork();
    if (rpid < 0)
    {
        // Rchild proc not created
        perror("Right Child Proc. not created\n");
        _exit(-1);
    }
    else if (rpid == 0)
    {
        quicksort(arr, i + part, j);
        _exit(0);
    }
}
```

- In the parent process create a new child process which sorts the right subarray from the pivot

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

- Wait for both the child processes to end.

➢ int partition(int arr[], int low, int high):
- Returns the pivot for the division into subarrays
- This code is similar for traditional and concurrent quick sort

➢ void insertionSort(int arr[], int k, int n)
- Insertion sort to sort the subarray with 5 or less elements

➢ Performance Comparison
- If processes are used then traditional will be faster since time is wasted in allocating memory and fork for every process. Also since concurrent process both will try to access the cache at the same time which will result in a cache miss.
- Threads have a speed comparable to the traditional method since each thread accesses the same memory space