Concurrent Quick Sort

Normal Quick sort:
-> In normal quick sort first we select a pivot and then we divide the array in two parts and then put all elements to left of pivot which are smaller than it and other elements to it's right and then apply this operation again on left and right part.

Cocurrent Quick Sort:
-> In concurrent quick sort we create two proceeses using fork() and then apply same sorting technique as normal quick sort on left and right parts.

-> We allocate array in shared memory segment so that all child processess can access it and change it.

-> Finally parent waits for all its children to complete using waitpid and get the result in shared memory region.

Threaded Quick Sort:
-> In threaded Quick sort we create threads for the two sub -arrays and then apply same sorting technique as normal quick sort on left and right parts.

-> We wait for threads to finish using pthread_join to get aour final sorted array.

Example:

```
10
90 87 4335 46354 43654 -9053 -4834 -3897 45 123

Concurrent Quicksort
time = 0.001813

Threaded Concurrent Quicksort
time = 0.002580

Normal Quicksort
time = 0.000006

Sorted Array
-9053 -4834 -3897 45 87 90 123 4335 43654 46354

Normal Quicksort ran:
[ 323.692961 ] times faster than Concurrent Quicksort
[ 460.611744 ] times faster than Threaded Concurrent Quicksort
```

1-> The three Quicksort were run on array of size 10.
2-> Time Taken by each code are shown.

3-> Clearly, Normal Quick Sort takes the least time followed by Concurrent Quick Sort using Threads followed by Threaded concurrent Quick Sort.

4->Sequential sort is better.
Argument:
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.

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,they are working concurrently! The shmget() requests the kernel to allocate a shared page for both the processes.

We need a shared memory segment because on forking two separate memory space gets created,therefore,memory writes performed by one of the processes do not affect the other.