Concurrent Quick Sort

Breakdown of the code for Quick Sort with different processes

Shared memory function

```
int * shared_mem(ll n)
{
    key_t key = IPC_PRIVATE;
    size_t shm_size = sizeof(int)*(n+5);
    int *arr;

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

    if ((arr = shmat(shmid, NULL, 0)) == (int *) -1)
    {
        perror("shmat"); _exit(1);
    }

    for(int i=0; i <n; i++)
        scanf("%d", &arr[i]);

    return arr;
}
```

This creates the segment and attaches it to our data space. It also takes inputs for the array.

Creating separate processes

```
void quicksort_proc(int *arr, int l, int r)
{
    if(l > r)
        _exit(1);

    if(r - l +1 <= 5)
    {
        insertion_sort(arr, l, r);
        return;
    }

    int pi = partition(arr, l, r);
    pid_t lpid,rpid;
    lpid = fork();

    if(lpid == 0)
    {
        quicksort_proc(arr, l, pi-1);
        _exit(1);
    }

    else
    {
        rpid = fork();
        if(rpid == 0)
        {
            quicksort_proc(arr, pi+1, r);
            _exit(1);
        }

        else
        {
            int status;
            waitpid(lpid, &status, 0);
            waitpid(rpid, &status, 0);
        }

    }

    return;
}
```

This part of the code checks for array length and implmenets insertion sort if length is less than or equal to 5. If it is greater, it partitions the array into left and right part. It forks and creates a process to sort left part. In the parent, it forks again to create a process to sort right part. In the parent, it waits for the child processes sorting left and right part, to join.

Insertion sort

The code for insertion sort is as follows

``` void insertion_sort(int *arr, int l, int r) { for(int i=l+1; i<=r; i++) { int key = arr[i]; int j = i-1; while(j >= l && arr[j] > key) { arr[j+1] = arr[j]; j--; }

```
    arr[j+1] = key;
}

return;
```

}

```
### Quick sort & partition

The code for partitioning randomly is as follows
```

int partition (int *arr, int l, int r)
{
srand(time(NULL)); int random = l + rand() % (r - l); swap(arr, random, r); int pivot = arr[r], i = l - 1, temp;

```
for (int j = l; j <= r-1; j++)
{
    if (arr[j] < pivot)
    {
        i++;
        swap(arr, i , j);
    }
}

swap(arr, i+1, r);
return (i + 1);
```

}

```
The code for Normal Quick Sort is as follows
```

void quicksort(int *arr, int l, int r) { if(l >= r) return;
```

```
 if(r - l +1 <= 5)
 {
     insertion_sort(arr, l, r);
     return;
 }

 int pi = partition(arr, l, r);
 quicksort(arr, l, pi - 1);
 quicksort(arr, pi + 1, r);

 return;
```

}

```
 ### Detaching memory segment

 This detaches the segment from the shared memory now that we are done using it.
 It deletes the shared memory segment thereafter.
```

void detach(int *arr) { if (shmdt(arr) == -1) { perror("shmdt"); _exit(1); }

```
 if (shmctl(shmid, IPC_RMID, NULL) == -1)
 {
     perror("shmctl"); _exit(1);
 }
```

}

```
 ## BONUS: Breakdown of the code for Quick Sort with threads

 Insertion sort code remains the same, and the logic for quicksort remains the same. Inst

 ### Creating threads
```

void *quicksort_threads(void *a) { struct arg *info = (struct arg*) a; int *arr = info -> arr; int l = info->l, r = info ->r;

```
 if(l > r)
    return NULL;

if(r-l+1 <= 5)
{
    insertion_sort(arr, l, r);
    return NULL;
}

int pi = partition(arr, l, r);
struct arg a1;
a1.l = l, a1.r = pi-1, a1.arr = arr;
pthread_t l_tid;
pthread_create(&l_tid, NULL,quicksort_threads, &a1);

struct arg a2;
a2.l = pi+1, a2.r = r, a2.arr = arr;
pthread_t r_tid;
pthread_create(&r_tid, NULL, quicksort_threads, &a2);

pthread_join(l_tid, NULL);
pthread_join(r_tid, NULL);
```

}

```

Threads are created to sort left and right half of the array. Structs are passed to it with the details of the left and right half of the arrays respectively. These threads execute the sorting and we wait for both of them to join towards the end.

## Conclusions

The normal quick sort runs the fastest due to no overheads of creating processes and threads. Threads quick sort runs faster than process quick sort as creating new processes with different stacks etc requires more time and has more overhead.

### For n = 10

Time taken by Normal QuickSort = 0.000116 Time taken by Concurrent QuickSort = 0.036541 Time taken by Threaded Concurrent QuickSort = 0.005083

Normal QuickSort is :

314.276294 times faster than Concurrent QuickSort 43.715032 times faster than Threaded Concurrent QuickSort

### For n = 100
```

Time taken by Normal QuickSort = 0.000130 Time taken by Concurrent QuickSort = 0.008321 Time taken by Threaded Concurrent QuickSort = 0.008319

Normal QuickSort is :

63.869440 times faster than Concurrent QuickSort 63.854496 times faster than Threaded Concurrent QuickSort

For n = 1000

Time taken by Normal QuickSort = 0.004970 Time taken by Concurrent QuickSort = 15.282341 Time taken by Threaded Concurrent QuickSort = 0.098592

Normal QuickSort is :

3074.883728 times faster than Concurrent QuickSort 19.837208 times faster than Threaded Concurrent QuickSort