

CS 236: Operating Systems Lab Spring 2025-26 Labquiz4-Lite (20 marks)	A
--	----------

Instructions

- No Internet. No phone. No devices. No handwritten notes. No books.
- No friends (strictly for the duration of the exam only!).
- Unethical practices will be an automatic recommendation for the FR grade.
- **Hardcoding of outputs will result in -5 marks per instance.**
- Please read the questions and submission instructions carefully.
- The lab quiz consists of 2 problems.
- **All problems are compulsory.**

Submission Guidelines

- **Untar and cd into the labquiz4-lite directory on your Desktop. Use:**
 - `./run.sh` to run xv6 in containerized environment
 - `submit-xv6` to submit (ask a TA to enter password)
- **Ensure that your final xv6 submission compiles without any errors. Submissions that do not compile will not be considered for evaluation.**

1. return of quiz1

Find the maximum value of an integer array (of size N) stored in a file, with the first line of the file being the size N. Then subsequent N lines are the N elements of the array. A function to read such a file into an array is given as part of the template code. Assume values to be used are greater than equal to 0 and less than 256.

The parent process P should **fork** twice to create two child processes, C1 and C2, and each child process should fork to create a new child process each, C3 (child of C1) and C4 (child of C2), for a total of 5 processes. The array should be almost equally (maximum difference of 1) divided into 4 parts, and one part each should be assigned to the C1, C2, C3 and C4 processes.

Each child process should calculate the maximum value of its portion of the array. C3 and C4 should return this value via their **exit** status to C1 and C2, respectively. C1 and C2 should then compare this with their own calculated maximum and return the greater of the two to the parent process P via their **exit** status. Parent process P should then compute the overall maximum and print this information.

`p1.c` contains a template for your code, and some sample test arrays are in `tc1.txt`, `tc2.txt` and `tc3.txt` for you to test your implementation.

Sample outputs

None

```
$ q1 tc1.txt
```

```
Array: 20 77 94 248 107 103 159 145 133 133 107 146 195 54 95
```

```
[pid 31221] Created child process: 31222
```

```
[pid 31221] Created child process: 31223
```

```
[pid 31222] Created child process: 31224
```

```
[pid 31222] arr[3:7]: max_est: 248
```

```
[pid 31224] arr[0:3]: max_est: 94
```

```
[pid 31223] Created child process: 31225
```

```
[pid 31223] arr[11:15]: max_est: 195
```

```
[pid 31225] arr[7:11]: max_est: 145
```

```
[pid 31222] arr[0:7]: self_max_est: 248, child_max_est: 94, max_est: 248
```

```
[pid 31223] arr[7:15]: self_max_est: 195, child_max_est: 145, max_est: 195
```

```
[pid 31221] max(array): 248
```

```
$ q1 t2.txt
```

```
Array: 156
```

```
[pid 1734643] Created child process: 1734644
```

```
[pid 1734643] Created child process: 1734645
```

```
[pid 1734644] Created child process: 1734646
```

```
[pid 1734644] arr[0:0]: max_est: 0
```

```
[pid 1734645] Created child process: 1734647
```

```
[pid 1734646] arr[0:0]: max_est: 0
```

```
[pid 1734645] arr[0:1]: max_est: 156
```

```
[pid 1734647] arr[0:0]: max_est: 0
```

```
[pid 1734644] arr[0:0]: self_max_est: 0, child_max_est: 0, max_est: 0
```

```
[pid 1734645] arr[0:1]: self_max_est: 156, child_max_est: 0, max_est: 156
```

```
[pid 1734643] max(array): 156
```

2. funny sort

Given an array of integers, use multiple threads to sort the array.

An array of size **N** uses **N-1** threads to sort it. Assuming threads are numbered from **0**, each thread **i** compares elements at index **i** and index **i+j** and swaps them if element at **i** is greater than element at index **i+j**, for all $j=1$ to k , such that $i+j < \text{size of array}$.

The logic for this sorting loop is given as part of the template code.

TASKS

1. Check the template code for comments that mark sections of the source code that need to be changed. Look for the string FUNNYSORT for sections marked of changes. No other sections of the source code should be modified.
2. Create **N-1** threads (using the **pthread**s API), each of which calls the sorting function
3. Check if the code works with N-1 threads
4. Fix the code to add synchronization primitives wherever needed.

Sample outputs

Test case 1: Multi-thread implementation with no synchronization

Shell

```
> ./q2 100 10
```

```
Sorting an array of size 100 using 99 threads...
```

```
Sorting failed.
```

```
Current array: 0 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3 4 5 5 7 8 8 8 8 10 10 10  
11 12 12 15 15 15 17 17 17 18 18 22 23 25 25 28 30 30 30 33 33 34 38 39 45 47  
49 49 49 52 53 56 56 57 57 60 63 65 66 66 67 68 69 70 71 72 73 74 76 76 77 78  
79 81 82 83 85 86 87 88 89 90 91 92 93 94 95 96 97 99
```

```
Sorted array: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24  
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50  
51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76  
77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
```

Test case 2: Multi-thread implementation with *not-efficient-but-correct* synchronization

Shell

```
make time
```

```
cc      q2.c      -o q2
```

```
time -p ./q2 1000 10
```

```
Sorting an array of size 1000 using 999 threads...
```

```
Sorting successful!  
real 2.79  
user 19.80  
sys 31.57
```

Test case 2: Multi-thread implementation with *best* synchronization

```
Shell  
› make time  
cc      q2.c      -o q2  
time -p ./q2 1000 10  
Sorting an array of size 1000 using 999 threads...  
Sorting successful!  
real 0.62  
user 1.27  
sys 9.70
```