

Lab Report

ECPE 170 – Computer Systems and Networks – Spring 2022

Name: Thomas Lau

Lab Topic: Performance Measurement (Lab #: 4)

Question #1:

Create a table that shows the real, user, and system times measured for the bubble and tree sort algorithms.

Answer:

	Real	User	Sys
Bubble Sort	0m27.167s	0m27.146s	0m0.004s
Tree Sort	0m0.046s	0m0.039s	0m0.004s

Question #2:

In the sorting program, what actions take user time?

Answer:

Comparisons, assignments, and so on.

Question #3:

In the sorting program, what actions take kernel time?

Answer:

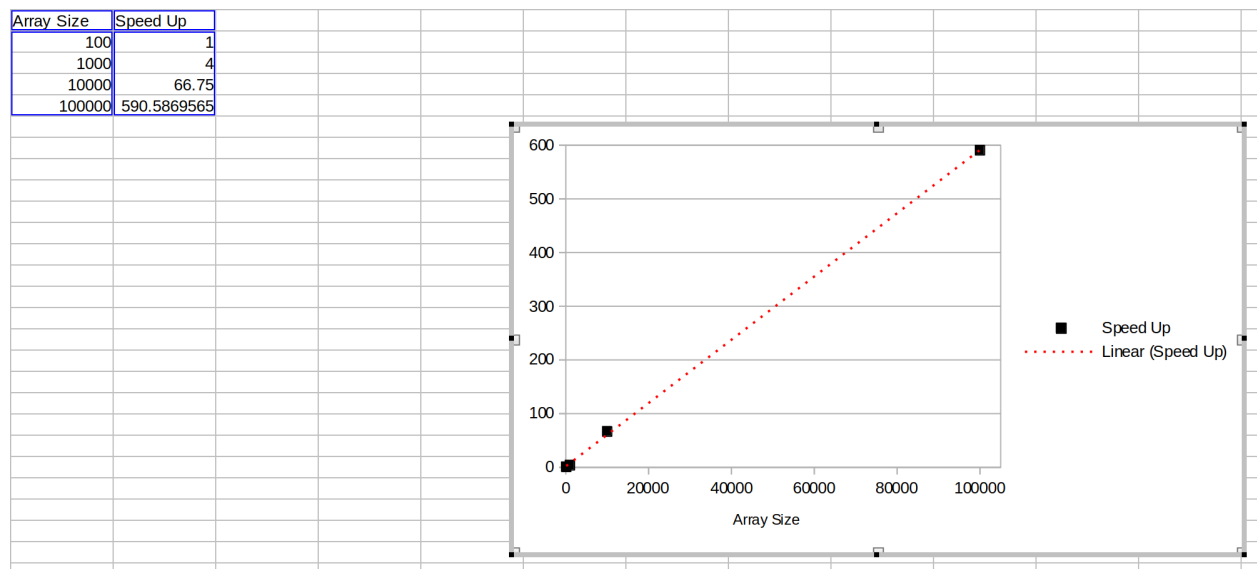
Memory Allocation takes kernel time.

Question #4:

Which sorting algorithm is fastest? (It's so obvious, you don't even need a timer)

Answer:

Tree Sort is fastest



From the table we can see that there seems to be a linear relationship between array size and speed up achieved. This is interesting because we generally know of the time complexity for bubble sort to be n^2 and tree sort to be $n \log n$. But from the looks of the table we have on the top right of the screen shot, we can see more of a linear relation, rather than exponential due to the drastic increase in array size. The difference in speed is insignificant when the array size is relatively smaller. Once we reach 10,000 and 100,000 elements, we begin to see a large difference in speed.

Question #5:

Create a table that shows the total Instruction Read (IR) counts for the bubble and tree sort routines. (In the text output format, IR is the default unit used. In the GUI program, un-check the "% Relative" button to have it display absolute counts of instruction reads).

Answer:

Size = 100	Inclusive Cost
Bubble Sort	159,429inclusive/158,951 self /379,870 overall
Tree Sort	55,321inclusive/1,523 self/275,793 overall

Question #6:

Create a table that shows the top 3 most active functions for the bubble and tree sort programs by IR count (excluding main()) - Sort by the "self" column if you're using kcachegrind, so that you see the IR counts for *just* that function, and not include the IR count for functions that it calls.

Answer:

	Rank 1	Rank 2	Rank 3
Bubble Sort	bubble_Sort	verifySort	initArray
Tree Sort	insert_element with 14,541	inorder'2 with 29,403	free_btree'2 with 28,368

Question #7:

Create a table that shows, for the bubble and tree sort programs, the most CPU intensive line that is part of the most CPU intensive function. (i.e. First, take the most active function for a program. Second, find the annotated source code for that function. Third, look inside the whole function code - what is the most active line? If by some chance it happens to be another function, drill down one more level and repeat.)

Answer:

	CPU Intensive line
Bubble Sort	if(array_start[j-1] > array_start[j]) is the

	most CPU intensive line with 74250 IR
Tree Sort	Struct BTreeNode *tempnode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode)) is the most CPU Intensive line with 297 IR, which calls to address that is 18909 IR

Question #8:

Show the Valgrind output file for the merge sort with the intentional memory leak. Clearly highlight the line where Valgrind identified where the block was originally allocated.

Answer:

```

part1 > E memcheck.txt
1  ==3921== Memcheck, a memory error detector
2  ==3921== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
3  ==3921== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
4  ==3921== Command: ./sorting_program merge
5  ==3921== Parent PID: 2226
6  ==3921==
7  ==3921==
8  ==3921== HEAP SUMMARY:
9  ==3921==    in use at exit: 400 bytes in 1 blocks
10 ==3921== total heap usage: 2 allocs, 1 frees, 1,424 bytes allocated
11 ==3921==
12 ==3921== 400 bytes in 1 blocks are definitely lost in loss record 1 of 1
13 ==3921==    at 0x483DD99: calloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
14 ==3921==    by 0x10934B: main (sorting.c:42)
15 ==3921==
16 ==3921== LEAK SUMMARY:
17 ==3921==    definitely lost: 400 bytes in 1 blocks
18 ==3921==    indirectly lost: 0 bytes in 0 blocks
19 ==3921==    possibly lost: 0 bytes in 0 blocks
20 ==3921==    still reachable: 0 bytes in 0 blocks
21 ==3921==    suppressed: 0 bytes in 0 blocks
22 ==3921==
23 ==3921== For lists of detected and suppressed errors, rerun with: -s
24 ==3921== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
25

```

From line 12 to 14, it highlights where the memory leakage occurred.

Question #9:

How many bytes were leaked in the buggy program?

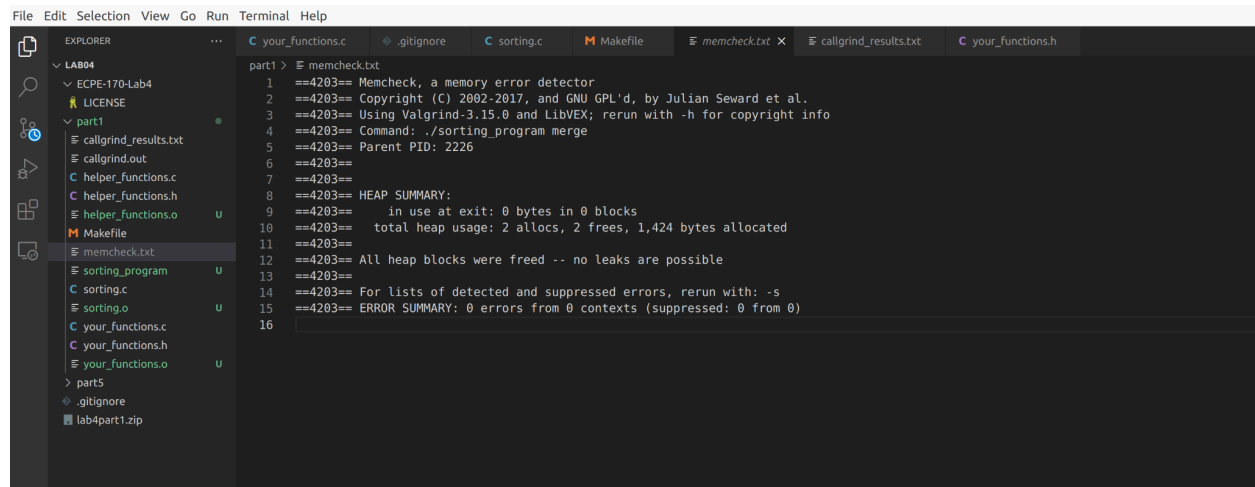
Answer:

400 bytes were leaked

Question #10:

Show the Valgrind output file for the merge sort after you fixed the intentional leak.

Answer:

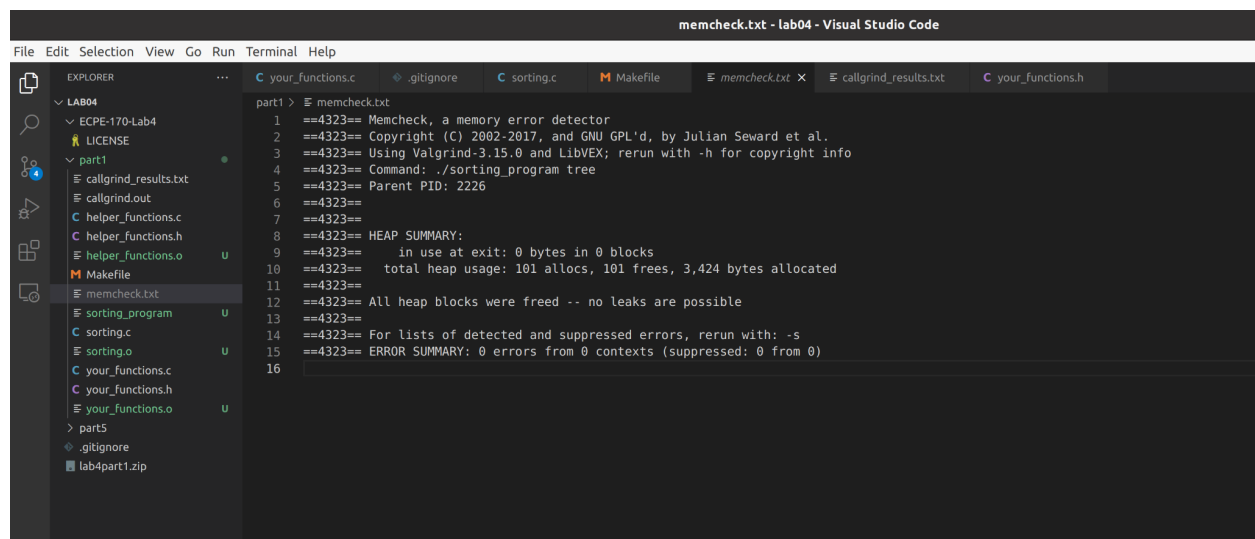


```
part1 > E memcheck.txt
1 ==4203== Memcheck, a memory error detector
2 ==4203== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
3 ==4203== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
4 ==4203== Command: ./sorting_program merge
5 ==4203== Parent PID: 2226
6 ==4203==
7 ==4203==
8 ==4203== HEAP SUMMARY:
9 ==4203==   in use at exit: 0 bytes in 0 blocks
10 ==4203==   total heap usage: 2 allocs, 2 frees, 1,424 bytes allocated
11 ==4203==
12 ==4203== All heap blocks were freed -- no leaks are possible
13 ==4203==
14 ==4203== For lists of detected and suppressed errors, rerun with: -s
15 ==4203== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
16
```

Question #11:

Show the Valgrind output file for your tree sort.

Answer:



```
memcheck.txt - lab04 - Visual Studio Code
part1 > E memcheck.txt
1 ==4323== Memcheck, a memory error detector
2 ==4323== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
3 ==4323== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
4 ==4323== Command: ./sorting_program tree
5 ==4323== Parent PID: 2226
6 ==4323==
7 ==4323==
8 ==4323== HEAP SUMMARY:
9 ==4323==   in use at exit: 0 bytes in 0 blocks
10 ==4323==   total heap usage: 101 allocs, 101 frees, 3,424 bytes allocated
11 ==4323==
12 ==4323== All heap blocks were freed -- no leaks are possible
13 ==4323==
14 ==4323== For lists of detected and suppressed errors, rerun with: -s
15 ==4323== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
16
```

Question #12:

How many seconds of real, user, and system time were required to complete the amplify program execution with Lenna image (Lenna_org_1024.pgm)? Document the command used to measure this.

Answer:

real 0m0.791s

user 0m0.734s

sys 0m0.056s

Command used: `time ./amplify IMAGES/Lenna_org_1024.pgm 11 1.1 2`

Question #13:

Research and answer: why does real time \neq (user time + system time)?

Answer:

Real time \neq (user time + system time) because real time consist of the time from start to finish on a call, but user time + system time is only the CPU time used by your process called.

Question #14:

In your report, put the output image for the Lenna image titled `output<somenumber>.pgm`.

Answer:

`output2048.pgm`

Question #15:

Excluding `main()` and everything before it, what are the top three functions run by `amplify` executable when you do count sub-functions in the total? Document the commands used to measure this. Tip: Sorting by the "Incl" (Inclusive) column in `kcachegrind` should be what you want.

Answer:

Top 3 functions: `convolve`, `mean_keeping`, and `double_thresh`

Commands:

`valgrind --tool=callgrind --dump-instr=yes --callgrind-out-file=callgrind.out ./amplify IMAGES/Lenna_org_1024.pgm 11 1.1 2`

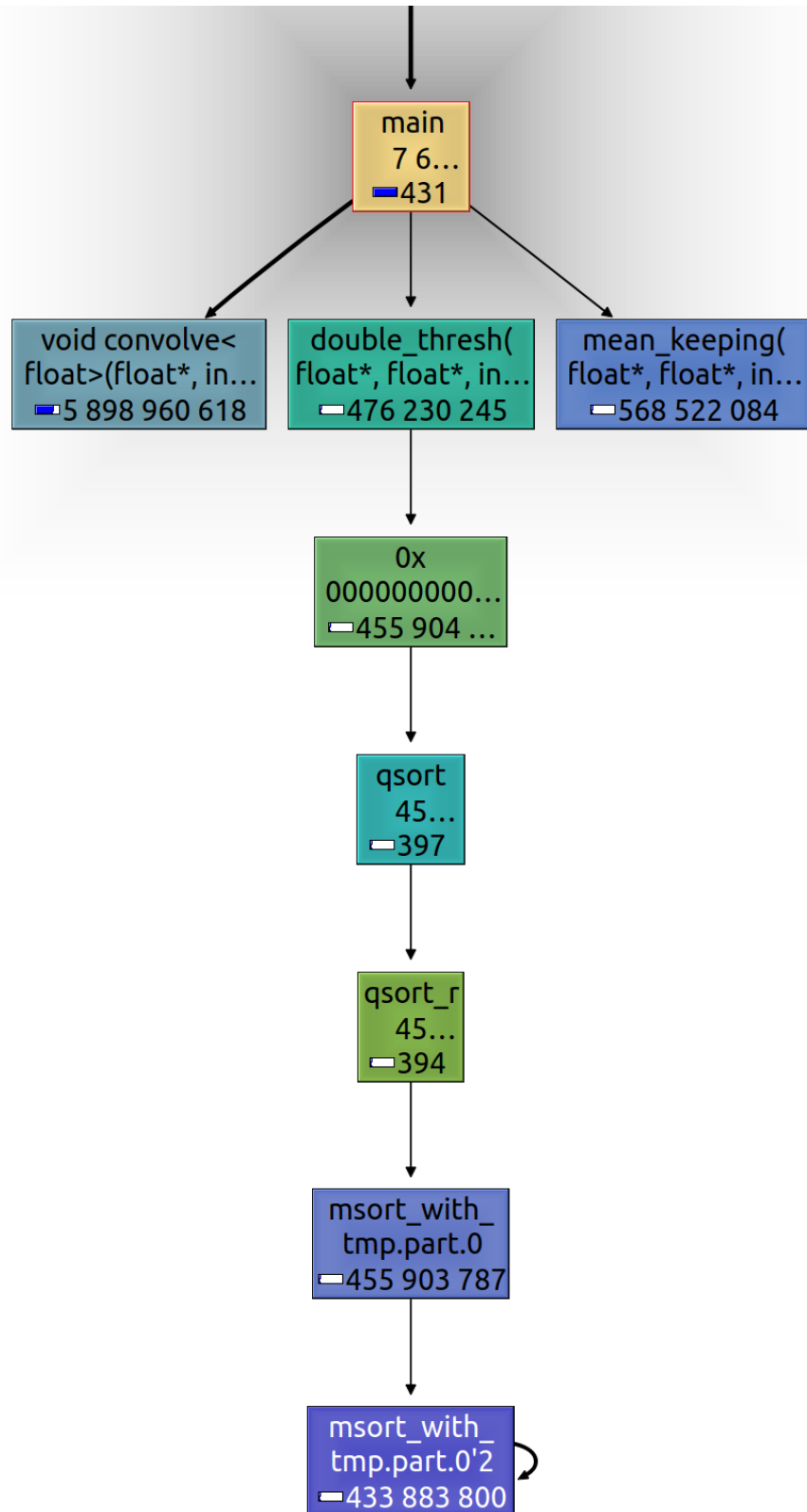
```
callgrind_annotate --auto=yes --threshold=100 callgrind.out > callgrind_results.txt
```

```
kcachegrind callgrind.out &
```

Question #16:

Include a screen capture that shows the kcachegrind utility displaying the callgraph for the amplify executable, starting at main(), and going down for several levels.

Answer:



Question #17:

Which function dominates the execution time? What fraction of the total execution time does this function occupy?

Answer:

Convolve taking $5898960618/7613225256$ or about 77.4%

Question #18:

Does this program show any signs of memory leaks? **Optional:** Remove as many memory leaks as possible for 5 extra credits. Document the specific leak(s) you found and the specific code change(s) you made in your lab report.

Answer:

Yes, this program shows signs of memory leaks.