Name: Ramon Arambula

Lab Topic: Performance Measurement (Lab #: 4)

Part 2: CPU Usage with Time

Question #1:

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

Answer:

Arr_size = 100,000	"Real" Time	"User" Time	"Sys" Time
Bubble Sort	0m 22.011s	0m 21.995s	0m 0.000s
Tree Sort	0m 0.044s	0m 0.043s	0m 0.000s

Question #2:

In the sorting program, what actions take user time?

Answer:

User is the amount of CPU time spent in user-mode code (outside the kernel) *within* the process. This is only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.

Question #3:

In the sorting program, what actions take kernel time?

Answer:

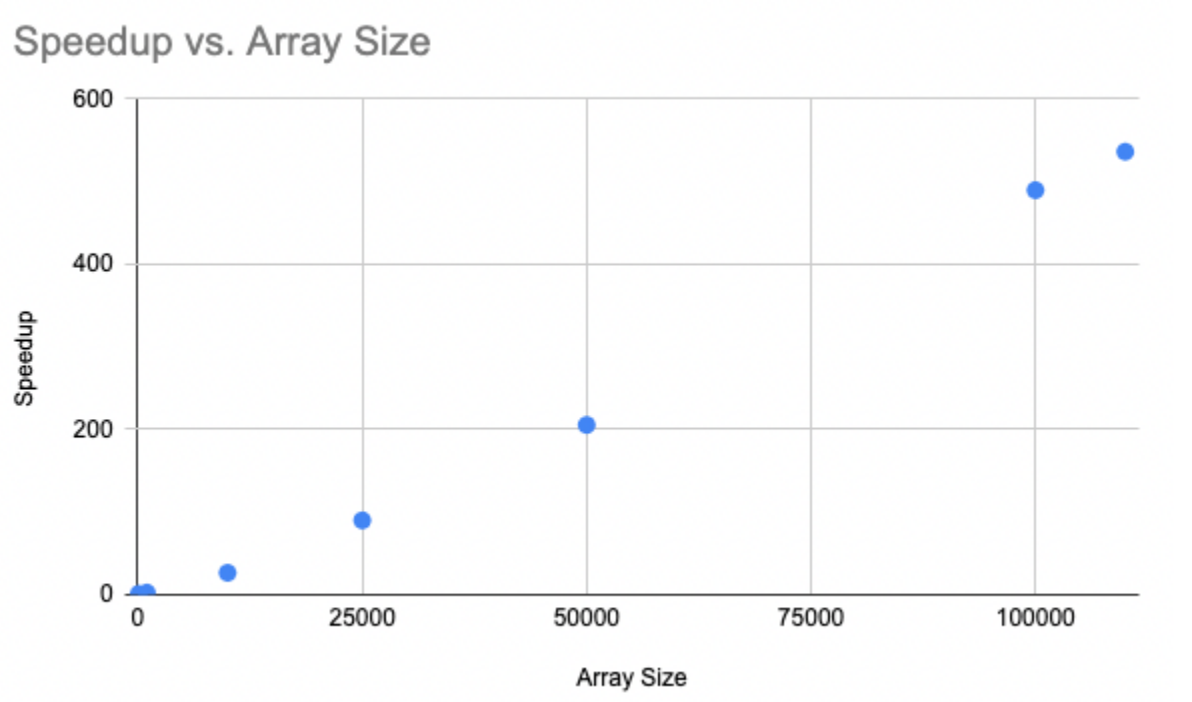
Kernel time, also known as Sys time is the amount of CPU time spent in the kernel within the process. This means executing CPU time spent in system calls *within the kernel,*as opposed to library code, which is still running in user-space. Like 'user', this is only CPU time used by the process. See below for a brief description of kernel mode (also known as 'supervisor' mode) and the system call mechanism.

Question #4:
Which sorting algorithm is fastest? (It's so obvious, you don't even need a timer)

Execute both the bubble sort and tree sort algorithms for array sizes: 100, 1000, 10,000, and 100,000. The speedup achieved by tree sort over bubble sort is S=(Tbubble/Ttree) where Tbubble is the execution time of the bubble sort and Ttree is the execution time of the tree sort. Create a scatter plot for speedup vs. array-size using LibreOffice. Make thorough comments (in 5-6 lines) on any trend(s) that you observe (speed grows linearly? exponentially? something else?).

Answer:
Of the two, tree sort is faster

The graph of speedup vs array size seems to be growing at a linear rate. To confirm this, I added 3 extra data points to recommended ones. I additionally tested with array sizes 25000, 50000, and 110000 to confirm that the growth rate of bubble sort vs tree sort was linear.



Part 3: CPU Usage with Valgrind

****NOTE** Array Size = 100,000**

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:

	Instruction Read (IR) Count
Bubble Sort	159,818,326,400
Tree Sort	99,462,699

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:

(3 most active functions in order)	1	2	3
Bubble Sort	BubbleSort'2 IR: 159,807,764,817	Random IR: 3,700,054	Random_r IR: 2,393,550
Tree Sort	Insert_element'2 IR: 46,597,696	_int_malloc'2 IR: 12,199,81	_int_free IR: 9,399,727

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:

	Most CPU-intensive Line
Bubble Sort	if (array_start[j-1] > array_start[j]) IR: 64,999,350,000
Tree Sort	else if (element < (*node)->element) IR: 11,972,106

Part 3: Memory leaks with Valgrind

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:

Intentional memory leak output below, highlighted in blue (line 14)

part1 > memcheck.txt

```
1  ==5085== Memcheck, a memory error detector
2  ==5085== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
3  ==5085== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
4  ==5085== Command: ./sorting_program merge
5  ==5085== Parent PID: 3373
6  ==5085==
7  ==5085==
8  ==5085== HEAP SUMMARY:
9  ==5085==     in use at exit: 400,000 bytes in 1 blocks
10 ==5085==   total heap usage: 2 allocs, 1 frees, 401,024 bytes allocated
11 ==5085==
12 ==5085== 400,000 bytes in 1 blocks are definitely lost in loss record 1 of 1
13 ==5085==    at 0x484C0A4: calloc (in /usr/lib/aarch64-linux-gnu/valgrind/vgpreload
14 ==5085==    by 0x108B17: main (sorting.c:44)
15 ==5085==
16 ==5085== LEAK SUMMARY:
17 ==5085==    definitely lost: 400,000 bytes in 1 blocks
18 ==5085==    indirectly lost: 0 bytes in 0 blocks
19 ==5085==      possibly lost: 0 bytes in 0 blocks
20 ==5085==    still reachable: 0 bytes in 0 blocks
21 ==5085==         suppressed: 0 bytes in 0 blocks
22 ==5085==
23 ==5085== For lists of detected and suppressed errors, rerun with: -s
24 ==5085== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
25
```

Question #9:

How many bytes were leaked in the buggy program?

Answer:

400,000 bytes were leaked

Question #10:

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

Answer:

memcheck.txt output after fixing intentional memory leak below

part1 > memcheck.txt

```
==4827== Memcheck, a memory error detector
==4827== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4827== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4827== Command: ./sorting_program merge
==4827== Parent PID: 3373
==4827==
==4827==
==4827== HEAP SUMMARY:
==4827==     in use at exit: 0 bytes in 0 blocks
==4827==   total heap usage: 2 allocs, 2 frees, 401,024 bytes allocated
==4827==
==4827== All heap blocks were freed -- no leaks are possible
==4827==
==4827== For lists of detected and suppressed errors, rerun with: -s
==4827== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Question #11:

Show the Valgrind output file for your tree sort.

Answer:

part1 > memcheck_tree.txt

```
==5821== Memcheck, a memory error detector
==5821== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5821== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==5821== Command: ./sorting_program tree
==5821== Parent PID: 3373
==5821==
==5821==
==5821== HEAP SUMMARY:
==5821==     in use at exit: 0 bytes in 0 blocks
==5821==   total heap usage: 100,001 allocs, 100,001 frees, 2,401,024 bytes allocated
==5821==
==5821== All heap blocks were freed -- no leaks are possible
==5821==
==5821== For lists of detected and suppressed errors, rerun with: -s
==5821== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Yay no leaks

Part 5: Profiling image amplifier code

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	0m 0.474s
User	0m 0.427s
Sys	0m 0.032s

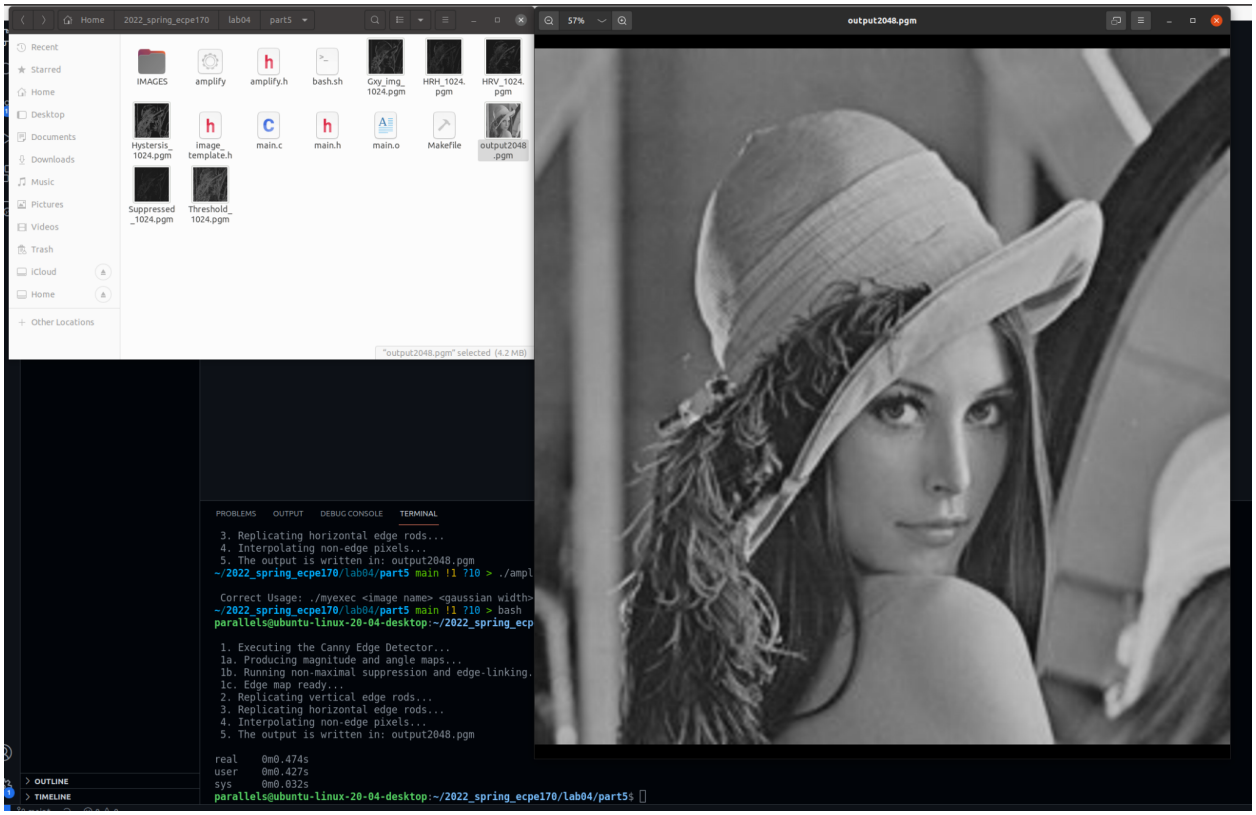
Question #13:

Research and answer: why does real time != (user time + system time)?

Answer:

Real time does not equal (user + system time) because real time not only measures (user+ system) time, but it also measures the amount of time that passes with slowdowns caused by system I/O.

Question #14:
In your report, put the output image for the Lenna image titled output<somenumber>.pgm.
Answer:



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:

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

kcachegrind callgrind_lenna.out &

Top 3 functions:

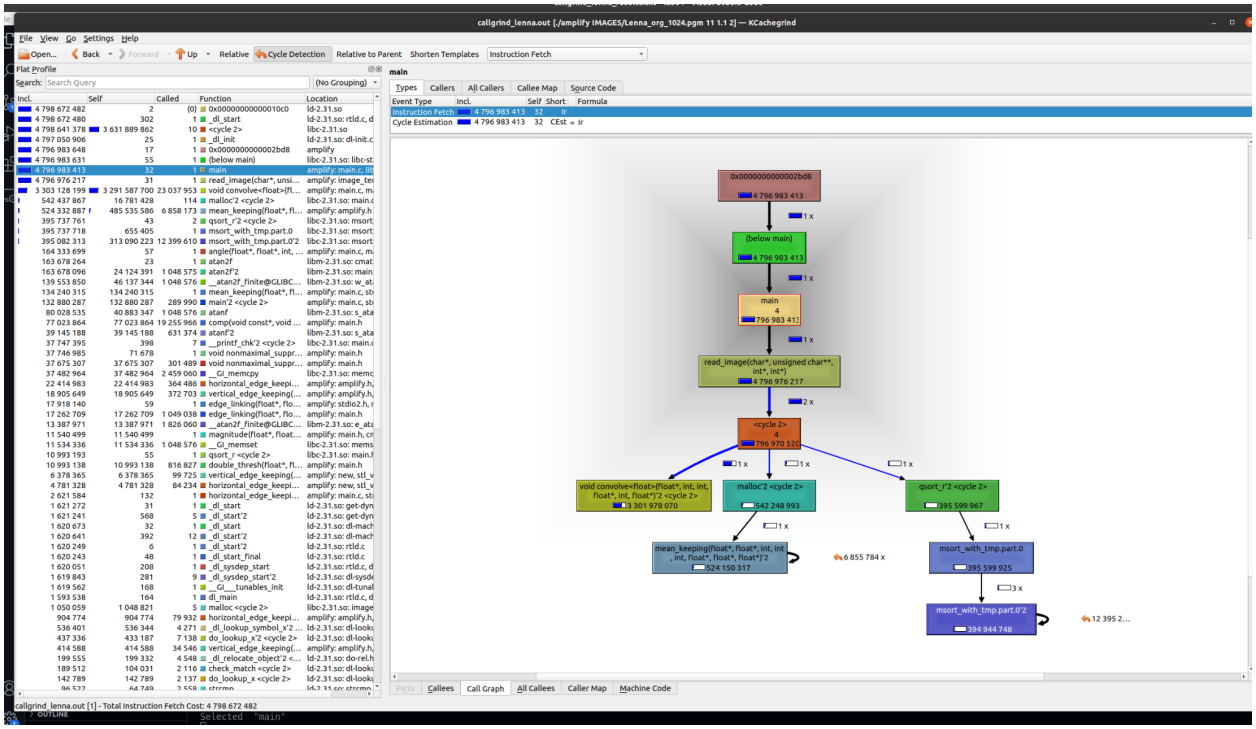
1. read_image()
2. convolve()
3. malloc’2

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:

XXXXX



Question #17:

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

Answer:

convolve() dominates the execution time, with 68.83%

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:

There is a lot of memory leaks,s so I'm just going to put the LEAK summary

```
92   ==14933==
93   ==14933== LEAK SUMMARY:
94   ==14933==    definitely lost: 16,778,236 bytes in 8 blocks
95   ==14933==    indirectly lost: 0 bytes in 0 blocks
96   ==14933==      possibly lost: 33,554,432 bytes in 2 blocks
97   ==14933==    still reachable: 0 bytes in 0 blocks
98   ==14933==         suppressed: 0 bytes in 0 blocks
99   ==14933==
```