Lab Report

ECPE 170 – Computer Systems and Networks – Fall 2015

Name: Zhiyun Yan

Lab Topic: Performance Measurement

Question(1) Create a table that shows the real, user, and system times measured for the bubble and quick sort algorithms

Sort Name	Real	User	System
Bubble	41.386s	41.220s	0.000s
Quick	0.015s	0.012s	0.000s

Question(2) In the sorting program, what actions take user time?
Answer: Create an array, fill it with random number, then put it into function.

Question(3) In the sorting program, what actions take kernel time?
Answer: Read array form system's memory and re-write data to memory.

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

Question(5) Create a table that shows the total Instruction Read (IR) counts for the bubble and quick 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).

Sort Name	IR
Bubble	164 783 028 663
Quick	55 578 494

Question(6) Create a table that shows the top 3 most active functions for the bubble and quick 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.

Sorting name	Top 3 active IR count	
Bubble	bubbleSort	164 775 226 856
	initArray	5 899 999
	rand	46
Quick	quickSort	47 775 905
	initArray	5 899 999
	random	42

Question(7) Create a table that shows, for the bubble and quick 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.)

Sort Name	Most Intensive Line
Bubble	if (array_start[j-1] > array_start[j])
Quick	1 call(s) to 'quickSort' (sorting_program: your_functions.c)

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.

```
==8299== Memcheck, a memory error detector
==8299== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==8299== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==8299== Command: ./sorting_program merge
==8299== Parent PID: 7728
==8299==
==8299==
==8299== HEAP SUMMARY:
==8299==     in use at exit: 400,000 bytes in 1 blocks
==8299==   total heap usage: 1 allocs, 0 frees, 400,000 bytes allocated
==8299==
==8299== 400,000 bytes in 1 blocks are definitely lost in loss record 1 of 1
==8299==    at 0x4C2CC70: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8299==    by 0x400759: main (sorting.c:42)
==8299==
==8299== LEAK SUMMARY:
==8299==    definitely lost: 400,000 bytes in 1 blocks
==8299==    indirectly lost: 0 bytes in 0 blocks
==8299==      possibly lost: 0 bytes in 0 blocks
==8299==    still reachable: 0 bytes in 0 blocks
==8299==         suppressed: 0 bytes in 0 blocks
==8299==
==8299== For counts of detected and suppressed errors, rerun with: -v
==8299== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Question(9) How many bytes were leaked in the buggy program?
Answer: 400,000 bytes

Question(10) Show the Valgrind output file for the merge sort after you fixed the intentional leak.

```
==8613== Memcheck, a memory error detector
==8613== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==8613== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==8613== Command: ./sorting_program merge
==8613== Parent PID: 7728
==8613==
==8613==
==8613== HEAP SUMMARY:
==8613==     in use at exit: 0 bytes in 0 blocks
==8613==   total heap usage: 1 allocs, 1 frees, 400,000 bytes allocated
==8613==
==8613== All heap blocks were freed -- no leaks are possible
==8613==
==8613== For counts of detected and suppressed errors, rerun with: -v
==8613== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Question(11) Show the Valgrind output file for your quick sort.

```
==8621== Memcheck, a memory error detector
==8621== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==8621== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==8621== Command: ./sorting_program quick
==8621== Parent PID: 7728
==8621==
==8621==
==8621== HEAP SUMMARY:
==8621==     in use at exit: 0 bytes in 0 blocks
==8621==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==8621==
==8621== All heap blocks were freed -- no leaks are possible
==8621==
==8621== For counts of detected and suppressed errors, rerun with: -v
==8621== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Question(12) How many seconds of real, user, and system time were required to complete the nmap scan? Document the command used to measure this.
Answer: time ./nmap www.pacific.edu

```
real 55.716s
user 0.148s
sys 0.596s
```

Question(13)Why does real time != (user time + system time)?
Answer: The real time means the time between computer start the program from end program. The user time means the time cost of user tell computer what to do and how to do. System time means computer run program when read and write in memory

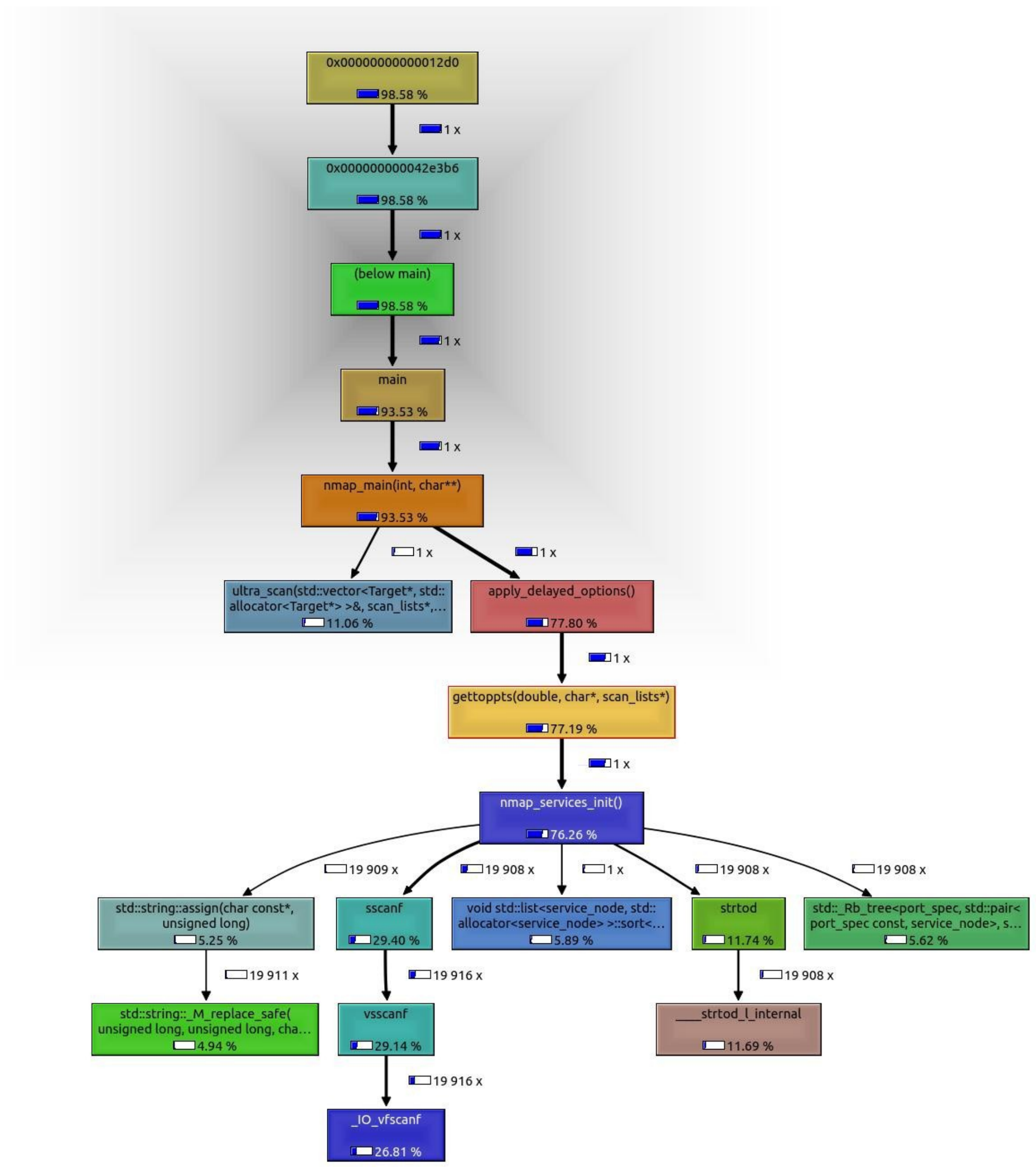
Question(14)As reported by nmap, what network ports and services are open and listening on www.pacific.edu?
Answer: 80/tcp for http service, 443/tcp for https service

Question(15) Excluding main() and everything before it, what are the top three functions run by nmap 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: The command is

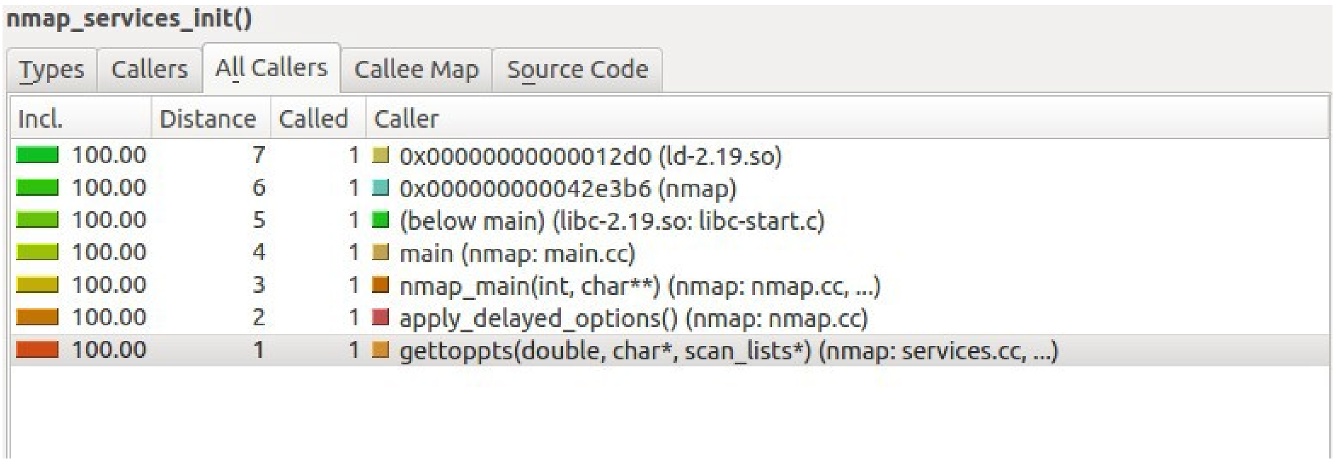
```
unix >> valgrind --tool=callgrind --dump-instr=yes --callgrind-out-file=callgrind.out ./nmap
www.pacific.edu
unix >> callgrind_annotate --auto=yes --threshold=100 callgrind.out > callgrind_results.txt
unix >>kcachegrind callgrind.out &
```

The top three function run is
1.int nmap_main(int argc, char *argv[]) ; %95.53
2.apply_delayed_options() %77.80
3.gettoppts(double, char*, scan_lists*) %77.19

Question(16) Include a screen capture that shows the kcachegrind utility displaying the callgraph for the nmap scan, starting at main(), and going down for several levels.



Question(17)Find nmap_services_init() in the profile. (It should be near the top of the sorted list). It seems to be relatively important, whatever it does. What function calls nmap_services_init()?



nmap_services_init()

Types | Callers | All Callers | Callee Map | Source Code

Incl.	Distance	Called	Caller
100.00	7	1	0x00000000000012d0 (ld-2.19.so)
100.00	6	1	0x000000000042e3b6 (nmap)
100.00	5	1	(below main) (libc-2.19.so: libc-start.c)
100.00	4	1	main (nmap: main.cc)
100.00	3	1	nmap_main(int, char**) (nmap: nmap.cc, ...)
100.00	2	1	apply_delayed_options() (nmap: nmap.cc)
100.00	1	1	gettoppts(double, char*, scan_lists*) (nmap: services.cc, ...)