Lab Report

ECPE 170 – Computer Systems and Networks – Fall 2012

Name: Erich Viebrock

Lab Topic: Performance Measurement (Lab #: 4)

Lab

Question #1:
Create a table that shows the real, user, and system times measured for each of the three algorithms.
Answer:

	Bubble	**Selection**	**Merge**
Real	1m10.320s	0m28.348s	
User	1m5.608s	0m25.922s	
Sys	0m0.152s	0m0.060s	

Question #2:
In the sorting program, what actions take user time?
Answer:
The actions that take user time are any actions after the program has initially created the array. Those actions include choosing which sorting algorithm to use, and then actually using that algorithm. The most time in user time goes towards actually sorting the array, because there are a large amount of array elements.

Question #3:
In the sorting program, what actions take kernel time?
Answer:
The actions that take kernel time are any actions that do not require user time. Hence, the only action that does not require user interaction is when the program initially creates the array. We expect this time to be fairly small, and at 0m0.152s, it behaves as predicted.

Question #4:
Which sorting algorithm is fastest? *(It's so obvious, you don't even need a timer).*
Answer:
Between bubble and selection, selection was obviously faster. Because merge sort uses heap memory, I would imagine it would take longer than selection.

Question #5:
Create a table that shows the total Instruction Read (IR) counts for all three programs. (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:

	Bubble	**Selection**	**Merge**
Instruction Read	124,849,911,046	70,014,822,047	

Question #6:
Create a table that shows the top 3 most active functions for all three programs by IR count (excluding main()).
Answer:

	Bubble	**Selection**	**Merge**
Function #1	bubbleSort	selectionSort	
Function #2	initArray	initArray	
Function #3	verifySort	verifySort	

Question #7:
Create a table that shows, for all three 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 Active Line
Bubble	`if (array_start[j-1] > array_start[j])`
Selection	`if (array_start[j] < array_start[min])`
Merge	

Question #8:
Using the "Screenshot" program in Ubuntu, include a screen capture that shows the kcachegrind utility displaying the callgraph for the merge sort algorithm, starting at the main() function, and going down (into the sort algorithm) to its deepest point.
Answer:

Question #9:
The Valgrind utility is very CPU intensive. This low-level monitoring of your program does not come for free! Use the time utility to compare the execution time of the bubble sort program both with and without Valgrind massif monitoring. Create a table that shows the difference in time. Also show the exact commands you used to perform the measurement.
Answer:

Bubble Sort	**With Valgrind**	**Without Valgrind**
Execution Time (Real)	2m40.135s	1m5.709s

With Valgrind:
```
time valgrind --tool=massif --stacks=yes --detailed-freq=1000000
--massif-out-file=massif.out ./sorting_program bubble
```
Without Valgrind:
```
time ./sorting_program bubble
```

Question #10:
Document the **total** memory, **heap** memory, and **stack** memory used for the *bubble sort* program at three points: The first snapshot, a middle snapshot, and the last snapshot.
Answer:

Bubble Sort	**First Snapshot**	**Second Snapshot**	**Third Snapshot**
Total Memory	400,364	400,260	400,200
Heap Memory	0	0	0
Stack Memory	400,364	400,260	400,200

Question #11:
Document the total memory, heap memory, and stack memory used for the *selection sort* program at three points: The first snapshot, a middle snapshot, and the last snapshot.
Answer:

Selection Sort	**First Snapshot**	**Second Snapshot**	**Third Snapshot**
Total Memory	400,364	400,308	400,200
Heap Memory	0	0	0
Stack Memory	400,364	400,308	400,200

Question #12:
Document the total memory, heap memory, and stack memory used for the *merge sort* program at three points: The first snapshot, a middle snapshot, and the last snapshot.
Answer:

Merge Sort	**First Snapshot**	**Second Snapshot**	**Third Snapshot**
Total Memory			
Heap Memory			
Stack Memory			

Question #13:
Why does the merge sort use heap memory, while the bubble and selection sort programs do not?
Support your answer with a specific line from the program source code.
Answer:
The other functions do not use dynamic memory allocation, which is created by `malloc` and `calloc`. The line of code responsible for this is:
```
temp_array = calloc(ARRAY_SIZE, sizeof(int));
```

Question #14:
Show the Valgrind output file corresponding to the intentional memory leak. Clearly highlight the line where Valgrind identified where the block was originally allocated.
Answer:

Question #15:
How many bytes were leaked in the buggy program?
Answer:

Question #16:
Show the Valgrind output file after you fixed the intentional leak.
Answer:

Question #17:
Is the Valgrind output report now clean (0 leaks, 0 other errors), or do you have other problems in the code you produced in the pre-lab?
Answer:

Post-Lab

Question #1:
How many seconds of real, user, and system time were required to complete the download? Document the command used to measure this.
Answer:
Real - 0m1.544s
User - 0m0.008s
Sys - 0m0.064s

Question #2:
Why does real time != (user time + system time)?
Answer:
User time and system time do not include download time.

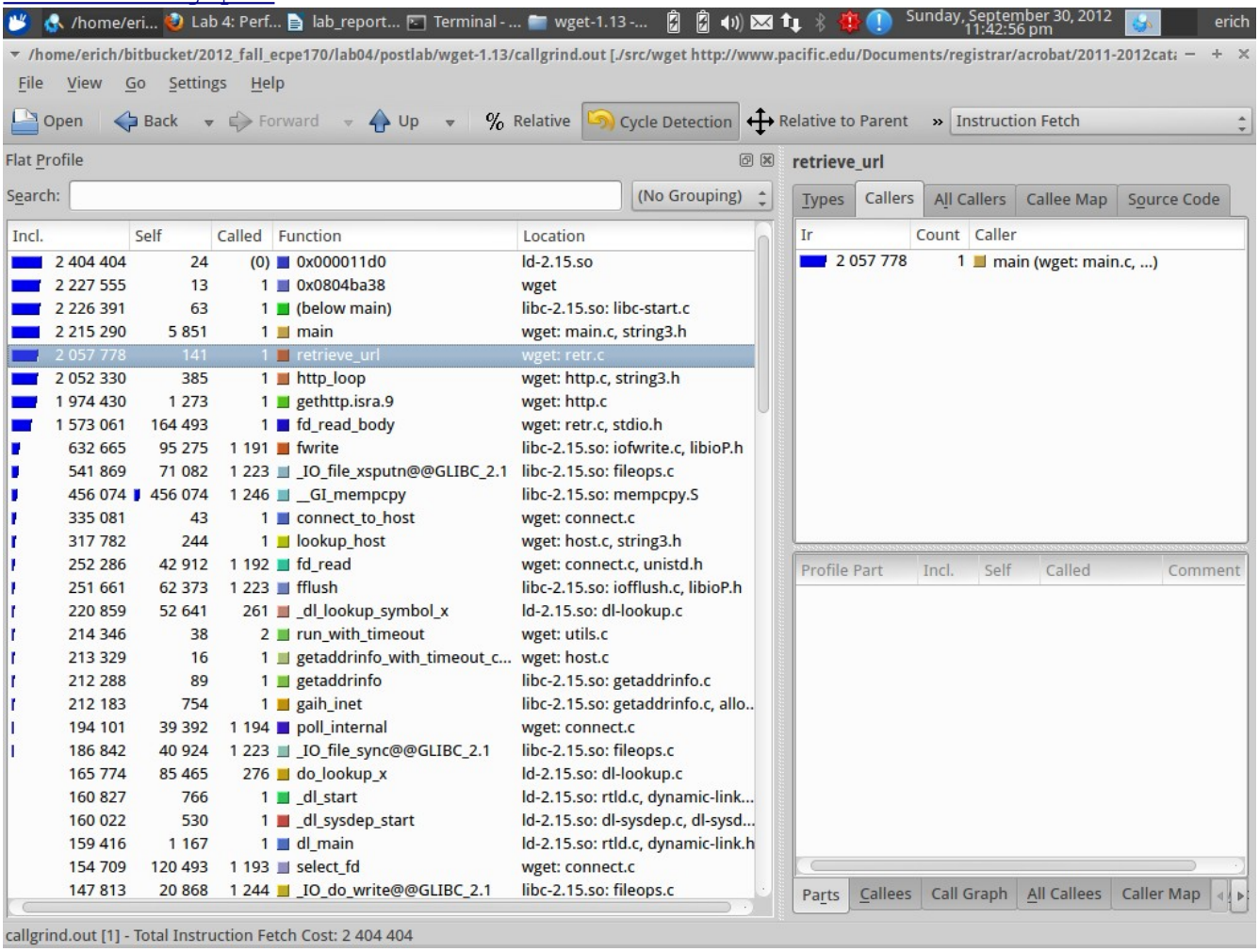
Question #3:
How big is the downloaded file on disk in bytes?
Answer:
The file is 1,819,713 bytes.

Question #4:
What was the maximum heap memory usage used by the wget program at any time? Document the commands used to measure this.
Answer:
The maximum heap memory usage by the wget program was 22,640 bytes. The command used to measure this were:

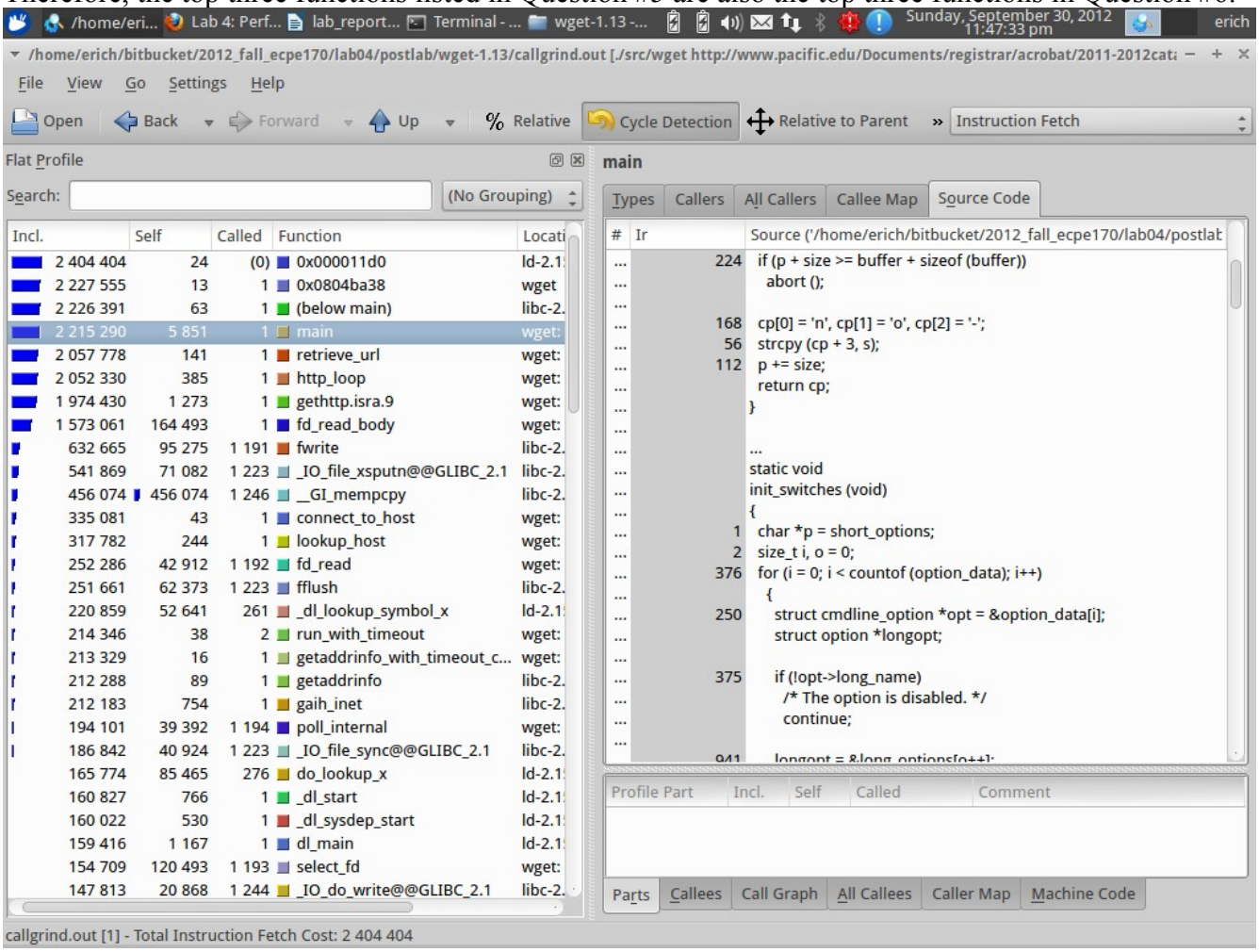
```
valgrind --tool=massif --stacks=yes --detailed-freq=1000000 --massif-out-file=massif.out ./src/wget http://www.pacific.edu/Documents/registrar/acrobat/2011-2012catalog.pdf
```

Question #5:
Excluding main() and everything before it, what are the top three functions run by wget when downloading the file when you **do** count sub-functions in the total? What percent of the time are they running? Document the commands used to measure this and include a screen capture supporting your answer. (Tip: The "Incl" column in kcachegrind should be what you want).
Answer:
1. retrieve_url
2. http_loop
3. gethttp.isra.9

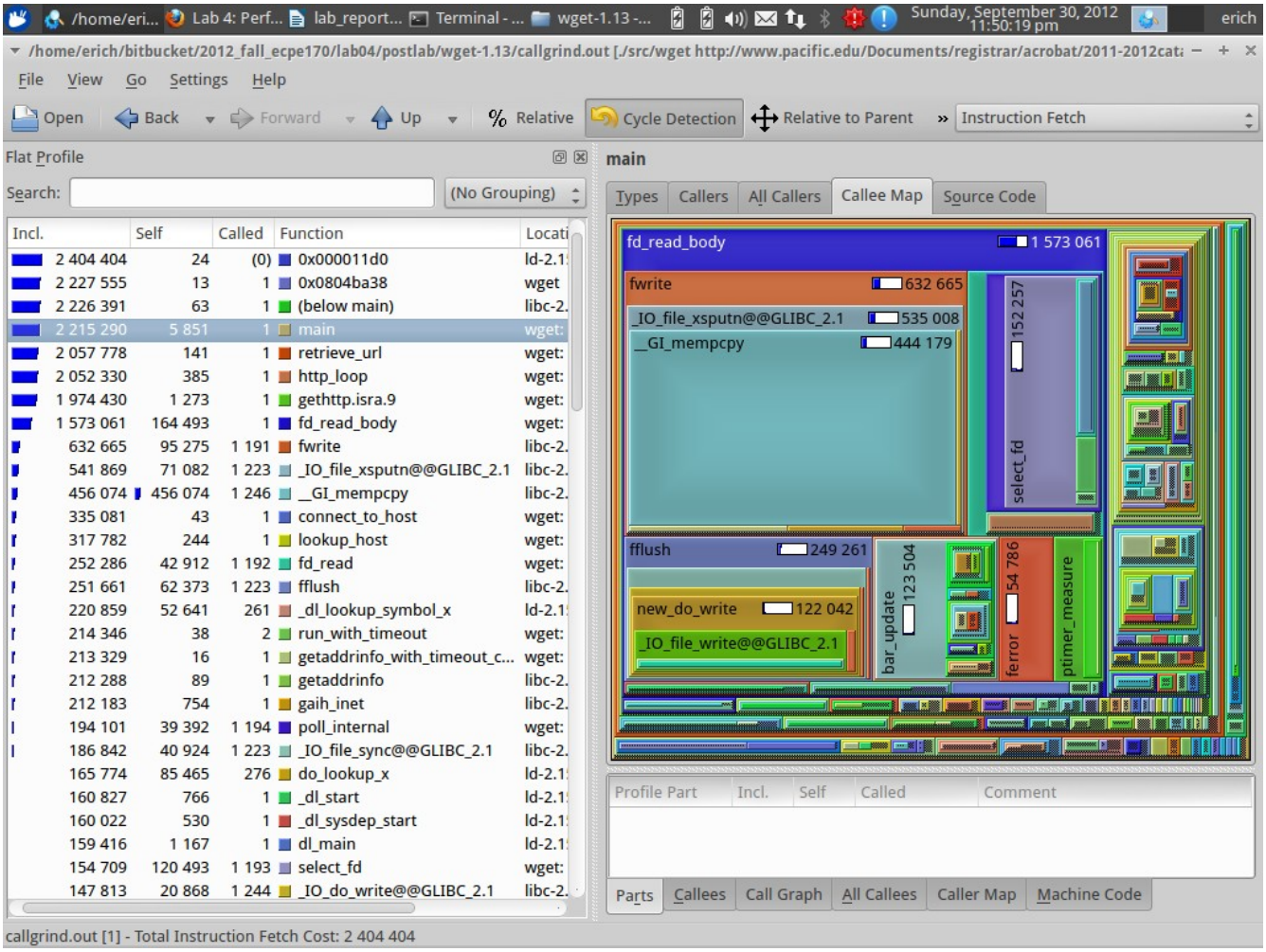
In terms of how often these functions are running, they are used frequently. When comparing the total number of instructions to all other functions, they along with a few other functions make up most of the program run time. The command to document the CPU usage was:
`valgrind --tool=callgrind --dump-instr=yes --callgrind-out-file=callgrind.out ./src/wget http://www.pacific.edu/Documents/registrar/acrobat/2011-2012catalog.pdf`



Question #6:
What are the top three functions run by wget when downloading the file when you **don't** count sub-functions in the total? What percent of the time are they running? (Tip: The "Self" column in kcachegrind should be what you want). Include a screen capture supporting your answer.
Answer:
When I looked through the source code for main, I did not see the top three functions inside main. Therefore, the top three functions listed in Question #5 are also the top three functions in Question #6.



Question #7:
Include a screen capture that shows the kcachegrind utility displaying the callgraph for the download, starting at main(), and going down for several levels.
Answer:



Question #8:
Find fwrite() in the profile. (It should be near the top of the sorted list). It seems to be relatively important, as you might imagine for a program that is writing a file to disk. What function calls fwrite()?
Answer:
The function `fd_read_body` calls `fwrite`.

Post-Lab Wrapup

Question #1:
What was the best aspect of this lab?
Answer:
I found it very interesting to see which sections of code were slowing down the whole process.

Question #2:
What was the worst aspect of this lab?
Answer:
Couldn't say I enjoyed waiting for the simulated tests to complete.

Question #3:
How would you suggest improving this lab in future semesters?
Answer:
You could provide a change to one of the functions you defined (main, bubbleSort, etc.) to show an improvement in execution time.