Lab 8: Optimizing and profiling

Lab sessions Tue Nov 28 to Thu Nov 30

Lab written by Julie Zelenski

Learning goals

After this lab, you will have

1. experimented with different mechanisms for execution timing
2. examined code compiled with and without optimization
3. run a profiler to analyze a program's runtime behavior

Share a computer with somebody new. Brag to each other about your plan to crush heap allocator.

Get started

Clone the lab8 repo by using the command below.

```
git clone /afs/ir/class/cs107/repos/lab8/shared lab8
```

Open the lab checkoff form (https://web.stanford.edu/class/cs107/cgi-bin/lab8).

Lab exercises

1) Timing tools

A simple means to measure runtime is using the system process timer via the Unix `time` command. This measurement is fairly coarse and is limited to timing the entire program but it's convenient in that you can time any program, whether you have source or not, and without making code changes. Let's try it now!

- The `samples` subdir for this lab contains `mysort_soln` from assign4 and `bbsort` from assign5, along with input files of random numbers. Compare the time for these two programs with the standard unix sort command on the same input. (Note: the `>/dev/null` at the end of the command will suppress output so you can more easily see the timing information).

  ```
  time samples/mysort_soln -n samples/million >/dev/null
  time samples/bbsort -n samples/million >/dev/null
  time sort -n samples/million >/dev/null
  ```

  How do the runtimes of the three sort programs rank?

- A `profiler` is a developer tool that observes an executing program to gather detailed statistics about resource consumption such as time spent, instructions executed, memory used, and so on. Although our primary use of valgrind has been for help finding memory errors and leaks, it can also operate as a profiler by using the `callgrind` tool to count instructions and memory accesses. (More info in the CS107 guide to callgrind (/class/cs107/guide/callgrind.html)) Run each of the three sort programs under callgrind and find the number labeled `I refs` in the output, this is the total count of instructions executed.

  ```
  valgrind --tool=callgrind samples/mysort_soln -n samples/thousand >/dev/n
  ull
  valgrind --tool=callgrind samples/bbsort -n samples/thousand >/dev/null
  valgrind --tool=callgrind sort -n samples/thousand >/dev/null
  ```

  The number of instructions counted should match the relative scale of the earlier results you got from time. Do they?

- The total instruction count is a whole program measurement, but callgrind also dumps detailed information to a file named *callgrind.out.pid* (where pid is the process number shown in left column of callgrind report, e.g. `==29240==`). If you run `callgrind_annotate callgrind.out.29240` (replace 29240 with your pid), it will further break down the instruction counts to show in which function the cycles were concentrated. Run the annotator on the output files created for the callgrind runs on mysort and bbsort. Look at the annotated output to find the function that accounts for the majority of the instructions executed. What function is it?

- If you have source for the program, you can move beyond external tools and directly instrument the code to get more fine-grained timing data. The real-time cycle counter available on the x86 chip is ideal for this. You accessed this timer for assign5. If you read the RTC counter before starting an operation and again when finished, you can subtract to count the elapsed cycles. The `fcyc.c` module contains cycle-counting routines from Chapter 5 of B&O. We will use this code in some of the later exercises.

2) Math: integer, float, and bitwise

The `trials.c` program has code to implement some numeric operations in a variety of ways (some clever, some less so, and some rather lame). The program has been instrumented with the RTC counter to measure cycle counts and will report on the relative speed of the different approaches.

- The `two_to_power` functions are various ways of computing 2^n. Versions A and B use the math library routines `pow` and `exp2`. Library routines are fast, right? Well, math.h operations are written for float/double -- scroll through the 300+ lines of the musl implementation of pow (http://git.musl-libc.org/cgit/musl/tree/src/math/pow.c) to get a glimpse of what is involved. Integers don't get a free ride here, they must be converted to float and operated on using fp instructions. How much do you expect we have to pay for that complexity relative to the versions C and D that use only integer operations?
- The `is_power` functions are various ways of testing whether an integer is an exact power of 2. Take a look at the different functions and how they approach the problem. How do you think these versions will stack up in performance?
- Before running the program, talk it over with your partner and make your own predictions about the relative performance. Run `trials` to see the timings and find out if your intuition holds up. You must now promise to never again use a floating point math function to do what could have been a simple integer or bitwise operation!

Re-implementing functionality already present in the C library is generally a big lose. The library routines are already written, tested, debugged, and aggressively optimized -- what's not to like? However, you do need to take care to choose the right tool for the job. Knowing the cost/benefit of the library functions and being knowledgeable of the compiler's optimizing capabilities will allow you better focus your attention on passages that require human intervention and not bother where the library functions and gcc can work their magic on your behalf.

Try your hand at this quiz on what gcc can/will optimize (http://ridiculousfish.com/blog/posts/will-it-optimize.html) -- fun and informative!

3) Copying data

The `copy` program explores the relative cost of different means of copying a large array. The total amount of data copied is constant, but changing how much work is done per iteration and number of iterations can have a profound impact on performance. The program experiments with a variety of ways: chunk-by-chunk in a loop, a memcpy in a loop, or a single call to memcpy or memmove the entire block. The program has been instrumented with the RTC counter to measure cycle counts and will report on the relative speed of the different approaches.

- Run the `copy` program several times and look at the cycle counts for the char, int, and long loops. About how much faster is copying by ints instead of chars? By longs instead of ints? The larger chunk demonstrates the value of "loop unrolling" to amortize the loop overhead by doing more work in each iteration and iterating fewer times.
- Compare the long loop to the memcpy loop (memcpy 8 bytes each iteration, same number of iterations as long loop) and learn the cost of using memcpy when a simple assignment would do-- ouch!
- Check out the cycle counts for the block memcpy/memmove. Wowzers -- somebody sure worked hard at optimizing the library routines! A speedup near an order of magnitude is achieved from copying the entire block in one go rather than copying each element individually.
- memcpy can have a performance advantage over memmove, since memcpy can ignore overlapping regions while memmove has to deal with them. Read the man page for `memmove` to see what it says about handling overlap. The wording seems to say that the data is copied from src to tmp and then from tmp to dst, but the closeness in the cycle counts for memcpy versus memmove suggests otherwise. Copying the data twice is one means to cope with overlap, but would incur a noticeable cost. Do you remember what the musl_memmove (review lab4 (/class/cs107/lab4)) did instead to handle overlap?

4) Stack vs heap allocation

Stack allocation is preferable to heap allocation for a multitude of reasons (type-safety, automatic allocation/deallocation, cache-friendly), while also offering significant performance advantages. Let's use callgrind to profile a sorting program to explore the relative runtime cost of using the stack versus the heap.

The `isort` program implements a simple insertion sort. When sorting the array, it will need temporary space when swapping array elements and the program explores the performance impact of allocating that temporary space using a fixed-size stack buffer, a variable-length stack buffer, or dynamically allocating from the heap.

- Run callgrind on the `isort` program using the command below (the use of toggle-collect tells callgrind to only count instructions within functions named swap_). After the program executes, run the annotator on the generated output file (replace 24256 with the pid of your file).

```
myth> valgrind --tool=callgrind --toggle-collect='swap_*' ./isort

==24256== Callgrind, a call-graph generating cache profiler
==24256== Command: ./isort
...
myth> callgrind_annotate --auto=yes callgrind.out.24256
```

The annotator displays the isort.c source file, and annotates each line in the C source with the number of assembly instructions executed on its behalf. You can identify performance bottlenecks by scanning the annotated result to find those lines with large counts. A larger count means the line was executed more times and/or it corresponds to many assembly instructions.

- Examine the counts for the two swap functions that use the stack. The function opening/closing brace is annotated with instruction counts for the function prolog/epilog (which includes the instructions to make space on the stack). The isort program calls each swap function roughly a million times so if you divide the counts by a million, you can estimate the per-call counts. How many instructions are executed in the function prolog/epilog per each call to swap_fixedstack? How many instructions per each call to swap_varstack?
- Compare the disassembly for swap_fixedstack versus swap_varstack. Allocating a constant-length stack array is one instruction (subtracting a compile-time constant from %rsp), how many additional instructions are needed to create a variable-length stack array? Does the count of additional instructions in the disassembly jive with the count of executed instructions reported by callgrind?
- Now look at the instruction counts for swap_heap. The annotation will indicate a small-ish number for the instructions in swap_heap that set up for the call to malloc/free and a much larger number of instructions executed within the malloc and free library functions. About how many instructions does each call to malloc incur? Each call to free? How does that compare to the cost of allocating and deallocating space on the stack?

Check off with TA

Before you leave, be sure to submit your checkoff sheet (in the browser) and have lab TA come by and confirm so you will be properly credited for labIf you don't finish everything before lab is over, we strongly encourage you to finish the remainder on your own. Double-check your progress with self check (/class/cs107/selfcheck.html#lab8).