Lecture 27 — Memory Profiling

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 28, 2018

Thus far we have focused on CPU profiling.

Memory profiling is also a thing; specifically heap profiling.

$\Rightarrow$ "Still Reachable": remained allocated   & we still had pointers to them, but were not properly deallocated?

If we don't look after those things, we're just using more and more memory over time.

That likely means more paging and the potential for running out of heap space altogether.

Again, the memory isn't really lost, because we could free it.

So, our tool for this comes from the Valgrind tool suite.

So what does Massif do?

It will tell you about how much heap memory your program is using, and also how the situation got to be that way.

So let's start with the example program from the documentation.

```c
#include <stdlib.h>

void g ( void ) {
    malloc( 4000 );
}

void f ( void ) {
    malloc( 2000 );
    g();
}

int main ( void ) {
    int i;
    int* a[10];

    for ( i = 0; i < 10; i++ ) {
        a[i] = malloc( 1000 );
    }
    f();
    g();

    for ( i = 0; i < 10; i++ ) {
        free( a[i] );
    }
    return 0;
}
```

After we compile (remember the -g option for debug symbols), run the
command:

```
jz@Loki:~/ece459$ valgrind --tool=massif ./massif
==25187== Massif, a heap profiler
==25187== Copyright (C) 2003-2013, and GNU GPL'd, by Nicholas Nethercote
==25187== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==25187== Command: ./massif
==25187==
==25187==
```

What happened?

Your program executed slowly, as is the case with any of the Valgrind toolset.

You don't get summary data on the console like we did with Valgrind or helgrind or cachegrind.

Weird. What we got instead was the file `massif.out.25187` (matches the PID of whatever we ran).
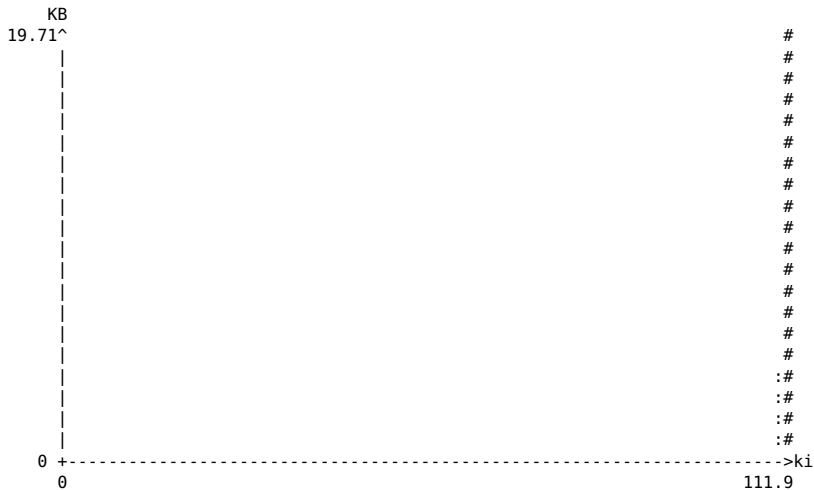
This file, which you can open up in your favourite text editor is not especially human readable, but it's not incomprehensible like the output from cachegrind.

There is an associated tool for summarizing and interpreting this data in a much nicer way: `ms_print`.

Which has nothing whatsoever to do with Microsoft. Promise.

```
  KB
19.71^                                                  #
     |                                                  #
     |                                                  #
     |                                                  #
     |                                                  #
     |                                                  #
     |                                                  #
     |                                                  #
     |                                                  #
     |                                                  #
     |                                                  #
     |                                                  #
     |                                                  #
     |                                                  #
     |                                                  #
     |                                                  #
     |                                                 :#
     |                                                 :#
     |                                                 :#
     |                                                 :#
   0 +----------------------------------------------->ki
     0                                              111.9
```
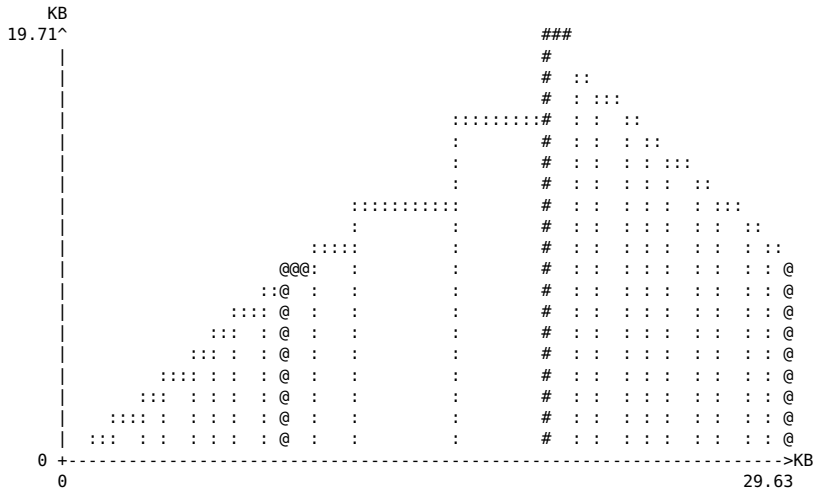
For a long time, nothing happens, then... kaboom!

Problem: we gave in a trivial program.

So for a relatively short program we should tell Massif to care more about the bytes than the CPU cycles, with the `-time-unit=B` option.

Let's try that.

```
  KB
19.71^                                    ###
     |                                    #
     |                                    #  ::
     |                                    #  : :::
     |                          :::::::::::#  :  :  ::
     |                          :          #  : :   : ::
     |                          :          #  : :  : : :::
     |                          :          #  : :  : : :  ::
     |                :::::::::::           #  : :  : : :  : :::
     |                :          :          #  : :  : : :  : : ::
     |             :::::         :          #  : :  : : :  : : : ::
     |          @@@:            :          #  : :  : : :  : :  : @
     |         ::@   :          :          #  : :  : : :  : :  : @
     |       ::::: @  :          :          #  : :  : : :  : :  : @
     |      :::  : @  :          :          #  : :  : : :  : :  : @
     |     ::: :  : @  :          :          #  : :  : : :  : :  : @
     |    :::: : :  : @  :          :          #  : :  : : :  : :  : @
     |    ::: : : :  : @  :          :          #  : :  : : :  : :  : @
     |  :::: : : :  : @  :          :          #  : :  : : :  : :  : @
     |  ::: : : : :  : @  :          :          #  : :  : : :  : :  : @
   0 +------------------------------------------------------------->KB
     0                                                    29.63
```

Now we're getting somewhere. We can see that 25 snapshots were taken.

It will take snapshots whenever there are appropriate allocation and deallocation statements, up to a configurable maximum.

For a long running program, it will toss some old data if necessary.

Let's look in the documentation to see what the symbols mean.

- Normal: :
- Detailed: @
- Peak: #

Peaks can be slightly inaccurate as they are recorded only at deallocation (and to speed up operations in general).

```
--------------------------------------------------------------------------
  n        time(B)        total(B)   useful-heap(B) extra-heap(B)   stacks(B)
--------------------------------------------------------------------------
  0              0               0                0             0            0
  1          1,016           1,016            1,000            16            0
  2          2,032           2,032            2,000            32            0
  3          3,048           3,048            3,000            48            0
  4          4,064           4,064            4,000            64            0
  5          5,080           5,080            5,000            80            0
  6          6,096           6,096            6,000            96            0
  7          7,112           7,112            7,000           112            0
  8          8,128           8,128            8,000           128            0
```

The time(B) column corresponds to time measured in allocations thanks to our choice of the time unit at the command line.

The extra-heap(B) represents internal fragmentation.

Why are stacks all shown as 0?

```
--------------------------------------------------------------------------
  n         time(B)        total(B)    useful-heap(B) extra-heap(B)   stacks(B)
--------------------------------------------------------------------------
  9          9,144          9,144           9,000           144            0
98.43% (9,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->98.43% (9,000B) 0x4005BB: main (massif.c:17)
```

So the additional information we got here is a reflection of where our heap allocations took place.

Thus far, all the allocations took place on line 17 of the program, which was `a[i] = malloc( 1000 );` inside that for loop.

```
--------------------------------------------------------------------
 n       time(B)        total(B)   useful-heap(B) extra-heap(B)  stacks(B)
--------------------------------------------------------------------
 14      20,184         20,184         20,000          184          0
99.09% (20,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->49.54% (10,000B) 0x4005BB: main (massif.c:17)
|
->39.64% (8,000B) 0x400589: g (massif.c:4)
| ->19.82% (4,000B) 0x40059E: f (massif.c:9)
| | ->19.82% (4,000B) 0x4005D7: main (massif.c:20)
| |
| ->19.82% (4,000B) 0x4005DC: main (massif.c:22)
|
->09.91% (2,000B) 0x400599: f (massif.c:8)
  ->09.91% (2,000B) 0x4005D7: main (massif.c:20)
```

Massif has found all the allocations in this program and distilled them down to a tree structure.

We see not just where the `malloc` call happened, but also how we got there.

Termination gives a final output of what blocks remains allocated and where they come from.

These point to memory leaks, incidentally, and valgrind would not be amused.

```
 24          30,344          10,024          10,000          24            0
99.76% (10,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->79.81% (8,000B) 0x400589: g (massif.c:4)
| ->39.90% (4,000B) 0x40059E: f (massif.c:9)
| | ->39.90% (4,000B) 0x4005D7: main (massif.c:20)
| |
| ->39.90% (4,000B) 0x4005DC: main (massif.c:22)
|
->19.95% (2,000B) 0x400599: f (massif.c:8)
| ->19.95% (2,000B) 0x4005D7: main (massif.c:20)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)
```

In fact, if I ask valgrind what it thinks of this program, it says:

```
jz@Loki:~/ece459$ valgrind ./massif
==25775== Memcheck, a memory error detector
==25775== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==25775== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==25775== Command: ./massif
==25775==
==25775==
==25775== HEAP SUMMARY:
==25775==     in use at exit: 10,000 bytes in 3 blocks
==25775==   total heap usage: 13 allocs, 10 frees, 20,000 bytes allocated
==25775==
==25775== LEAK SUMMARY:
==25775==    definitely lost: 10,000 bytes in 3 blocks
==25775==    indirectly lost: 0 bytes in 0 blocks
==25775==      possibly lost: 0 bytes in 0 blocks
==25775==    still reachable: 0 bytes in 0 blocks
==25775==         suppressed: 0 bytes in 0 blocks
==25775== Rerun with --leak-check=full to see details of leaked memory
==25775==
==25775== For counts of detected and suppressed errors, rerun with: -v
==25775== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

So probably a good idea to run valgrind first and make it happy before we go into figuring out where heap blocks are going with Massif.

Okay, what to do with the information from Massif, anyway?

It should be pretty easy to act upon this information. Start with the peak snapshot (worst case scenario) and see where that takes you (if anywhere).

You can probably identify some cases where memory is hanging around unnecessarily.

Memory usage climbing over a long period of time, perhaps slowly, but never really decreasing – memory is filling up somehow with some junk?

Large spikes in the graph – why so much allocation and deallocation in a short period?

- Look into stack allocation (`-stacks=yes`) option.
- Look at the children of a process (anything split off with `fork`) if desired.
- Check low level stuff: if we're doing something other than `malloc`, `calloc`, new, etc. and doing low level stuff like `mmap` or `brk` that is usually missed, but we can do profiling at page level (`-pages-as-heap=yes`).

As is often the case, we have examined how the tool works on a trivial program.

Let's see if we can do some live demos of Massif at work.