Lecture 27 — Memory Profiling

Jeff Zarnett

~~Memory Profiling~~ Return to Asgard

Thus far we have focused on CPU profiling. Other kinds of profiling got some mention, but they're not the only kind of profiling we can do. Memory profiling is also a thing, and specifically we're going to focus on heap profiling. We kind of touched on the subject a little bit earlier when we looked at finding memory leaks. The ideas are the same: we don't want to leak memory, but remember that last category (other than suppressed), "Still Reachable", things that remained allocated and we still had pointers to them, but were not properly deallocated? Right, we care about them too, and for that we want to do heap profiling.

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.

Well, let's start with where we left off. Returning to the realm of Asgard, we're going to call again on our old friend Valgrind. Except this time we're going to use a fourth tool in it: Massif. This is, obviously, a joke on "massive", combined with the name Sif, a Norse goddess associated with the earth (and in the Marvel movies, Shieldmaiden to Thor). While we're on the subject, Sif has an axe (shield?) to grind with Loki, because at some point he cut off her golden hair (and in the Marvel films, it grew back in dark). That Loki – what a trickster! Right, we're digressing... what do you mean the course isn't ECE 459: Norse Mythology?!

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 [Dev16]:

```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;
}
```

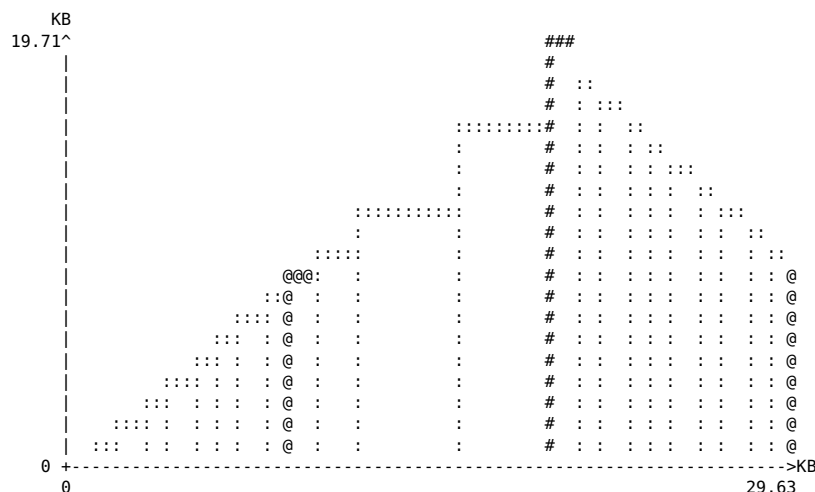After we compile (remember the `-g` option for debug symbols), run the command:

```
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==
```

Doesn't that look useful?! What happened? Your program executed slowly, as is always the case with any of the Valgrind toolset, but 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 ("Aha, a 1 in column 4 of line 2857. That's what's killing our performance!"). 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.

If we look at the output there (hint: pipe the output to `less` or something, otherwise you get a huge amount of data thrown at the console), it looks much more user friendly.

```
    KB
19.71^                                                              #
     |                                                              #
     |                                                              #
     |                                                              #
     |                                                              #
     |                                                              #
     |                                                              #
     |                                                              #
     |                                                              #
     |                                                              #
     |                                                              #
     |                                                              #
     |                                                              #
     |                                                              #
     |                                                              #
     |                                                              #
     |                                                             :#
     |                                                             :#
     |                                                             :#
     |                                                             :#
   0 +----------------------------------------------------------------->ki
     0                                                            111.9
```
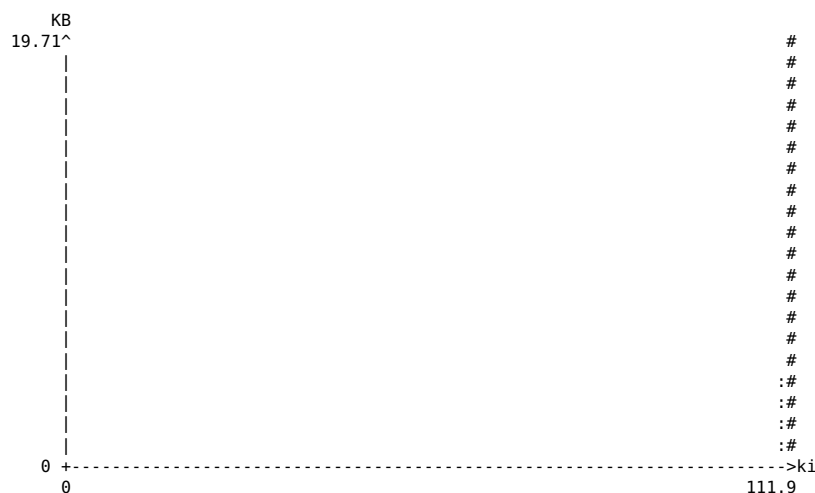
Now wait a minute. This bar graph might be user friendly but it's not exactly what I'd call... useful, is it? For a long time, nothing happens, then... kaboom! According to the docs, what actually happened here is, we gave in a trivial program where most of the CPU time was spent doing the setup and loading and everything, and the trivial program ran for only a short period of time, right at the end. 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
```

Neat. 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, and for a long running program, toss some old data if necessary. Let's look in the documentation to see what the symbols mean (they're not just to look pretty). So, from the docs [Dev16]:

2

- Most snapshots are normal (they have just basic information) They use the ':' characters.

- Detailed snapshots are shown with '@' characters. By default, every 10th snapshot is detailed.

- There is at most one peak snapshot. The peak snapshot is a detailed snapshot, and records the point where memory consumption was greatest. The peak snapshot is represented in the graph by a bar consisting of '#' characters.

As a caveat, the peak can be a bit inaccurate. Peaks are only recorded when a deallocation happens. This just avoids wasting time recording a peak and then overwriting it; if you are allocating a bunch of blocks in succession (e.g., in assignment 1, a bunch of structs that have a buffer) then you would constantly be overwriting the peak over and over again. Also, there's some loss of accuracy to speed things up. Well, okay.

So let's look at the snapshots. We'll start with the normal ones. There are 9 of those, numbers 0 through 8:

```
--------------------------------------------------------------------
 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 columns are pretty much self explanatory, with a couple exceptions. 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[1] in the blocks we received. The stacks column shows as zero because by default, Massif doesn't look at the stack. It's a heap profiler, remember?

Number 9 is a "detailed" snapshot, so I've separated it out, and reproduced the headers there to make this a little easier to remember what they are.

```
--------------------------------------------------------------------
 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.

Then let's look at the peak snapshot (again, trimmed a bit to call out exactly what we need to see here):

```
--------------------------------------------------------------------
 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)
|
```

---

[1]Remember from operating systems: if the user asked for some n bytes where n is not a nice multiple the returned block may be "rounded up". So a request for 1000 bytes is bumped up to 1016 bytes in this example. The extra space is "wasted" but it's nicer than having a whole bunch of little tiny useless fragments of the heap to be managed.

```
->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 that traces the path through which all of these various memory allocations occurred. So not just where the `malloc` call happened, but also how we got there.

When program termination occurs we get 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 with us.

```
 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.

Things to watch out for are: 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?

Other cool things we can do with Massif [Dev16]:

- Look into stack allocation (`-stacks=yes`) option. This slows stuff down a lot, and not really necessary since we want to look at heap.

- 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. As a live demo, let's see what happens when we take the program complexity up a little bit by (1) looking at the search program we saw in the earlier talk about valgrind; and (2) looking at the original (unmodified) paster.c file from assignment 1 (and then perhaps fixing it and going on). Depending on time available, we may look at some more complex programs.

References

[Dev16] Valgrind Developers. Massif: a heap profiler, 2016. Online; accessed 23-January-2016. URL: `http://valgrind.org/docs/manual/ms-manual.html`.