Lecture 28 — Causal Profiling, Memory Profiling

Patrick Lam & Jeff Zarnett
patrick.lam@uwaterloo.ca jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 3, 2022

~~Memory Profiling~~ Return to Asgard

As with queueing theory:

$$\text{allocs} > \text{frees} \implies \text{usage} \to \infty$$

At least more paging, maybe total out-of-memory.

But! Memory isn't really lost: we could free it.

Our tool for this comes from the Valgrind tool suite.

DHAT

DHAT tracks allocation points and what happens to them over a program's execution.

An allocation point is a point in the program which allocates memory.

Leaf Node

```
AP 1.1.1.1/2 {
  Total:     31,460,928 bytes (2.32%, 1,565.9/Minstr) in 262,171 blocks (4.41%, 13.05/Minstr
     avg size 120 bytes, avg lifetime 986,406,885.05 instrs (4.91% of program duration)
  Max:       16,779,136 bytes in 65,543 blocks, avg size 256 bytes
  At t-gmax: 0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  At t-end:  0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  Reads:     5,964,704 bytes (0.11%, 296.88/Minstr), 0.19/byte
  Writes:    10,487,200 bytes (0.51%, 521.98/Minstr), 0.33/byte
  Allocated at {
    ^1: 0x95CACC9: alloc (alloc.rs:72)
      [omitted]
    ^7: 0x95CACC9: parse_token_trees_until_close_delim (tokentrees.rs:27)
    ^8: 0x95CACC9: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::
                          StringReader<'a>>::parse_token_tree (tokentrees.rs:81)
    ^9: 0x95CAC39: parse_token_trees_until_close_delim (tokentrees.rs:26)
    ^10: 0x95CAC39: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::
                          StringReader<'a>>::parse_token_tree (tokentrees.rs:81)
    #11: 0x95CAC39: parse_token_trees_until_close_delim (tokentrees.rs:26)
    #12: 0x95CAC39: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::
                          StringReader<'a>>::parse_token_tree (tokentrees.rs:81)
  }
}
```

Parents

Going up the tree, DHAT tells you about all of the allocation points that share a call-stack prefix.

The ˆ before the number indicates that the line is copied from the parent.

A # indicates that the line is unique to that node.

A sibling of the example above would diverge at the call on line 10.

Shieldmaiden to Thor

Using Massif

What does Massif do?

- How much heap memory is your program using?
- How did this happen?

Next up: example from Massif docs.

Example Allocation Program

```
fn g() {
    let a = Vec::<u8>::with_capacity(4000);
    std::mem::forget(a)
}

fn f() {
    let a = Vec::<u8>::with_capacity(2000);
    std::mem::forget(a);
    g()
}

fn main() {

    let mut a = Vec::with_capacity(10);
    for _i in 0..10 {
        a.push(Box::new([0;1000]))
    }
    f();
    g();
}
```

Send in Sif

After we compile, run the command:

```
plam@amqui ~/c/p/l/l/L/alloc> valgrind --tool=massif target/debug/alloc
==406569== Massif, a heap profiler
==406569== Copyright (C) 2003-2017, and GNU GPL’d, by Nicholas Nethercote
==406569== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==406569== Command: target/debug/alloc
==406569==
==406569==
```

That Was Useful!!!

What happened?

1. The program ran slowly (because Valgrind!)
2. No summary data on the console
   (like memcheck or helgrind or cachegrind.)

Weird. What we got instead was the file `massif.out.[PID]`.

Post-Processing

`massif.out.[PID]`:

plain text, sort of readable.

Better: `ms_print`.

Which has nothing whatsoever to do with Microsoft. Promise.

Post-Processed Output

```
    KB
19.71^                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                      :#
     |                                                                      :#
     |                                                                      :#
     |                                                                      :#
   0 +----------------------------------------------------------------------->ki
     0                                                                   111.9
```

User Friendly, But Not Useful

For a long time, nothing happens, then… kaboom!

Why? We gave it a trivial program.

We should tell Massif to care more
about bytes than CPU cycles,
with `--time-unit=B`.

Let's try that.

Valgrind (Memcheck) First

Run valgrind (Memcheck) 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?

Easy!

- Start with peak (worst case scenario)
  and see where that takes you (if anywhere).
- You can probably identify some cases where memory is hanging around unnecessarily.

Places to Look with Massif

Memory usage climbing over a long period of time, perhaps slowly, but never decreasing—memory filling with junk?

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

Other Massif-ly Useful Things

- stack allocation (`--stacks=yes`).
- children of a process
  (anything split off with `fork`) if desired.
- low level stuff: if going beyond `malloc`, `calloc`, new, etc. and using `mmap` or `brk` that is usually missed, can do profiling at page level (`--pages-as-heap=yes`).

Live Demos

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

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