Writing better C code: debugging and profiling

Alex Böhnert

Rob's technical seminar

8.11.2013

Overview

Debugging

- gdb

Profiling

- gprof
- valgrind

Finding memory leaks

- memwatch
- valgrind memcheck

Coverage analysis

- gcov
- `-coverage`

Memory leaks

How to use memwatch

WARNING: Do not use memwatch with multithreaded code

- get it at `http://www.linkdata.se/sourcecode/memwatch/`
- Gets compiled directly into your code (just put `memwatch.c` and `memwatch.h` in your source directory).
- Put `-DMEMWATCH` in your compiler flags
- Run your code normally – at the end, a file `memwatch.log` gets written.

How to use valgrind

```
valgrind --leak-check=full mycode
```

What a debugger does (not) do

What it does

- Find bugs!
- Tell you the state of variables etc. after crash
- Lets you walk through the program hierarchy (did I pass that pointer correctly?)
- Check the state of the program during execution
- Walk through your code line by line

What it does not do

- Logic bugs
- “Heisenbugs”
- find wrong code
- Deliberate code-breaking attempts

How to do it

Code prerequisites

- Turn on ALL compiler warnings (`-Wall` doesn't do that!)
- Compiler flags: `-g`
- NO `-Ox`, with $x \neq 0$
- `-DMEMWATCH` to turn on memwatch; your source files need to `#include "memwatch.h"`

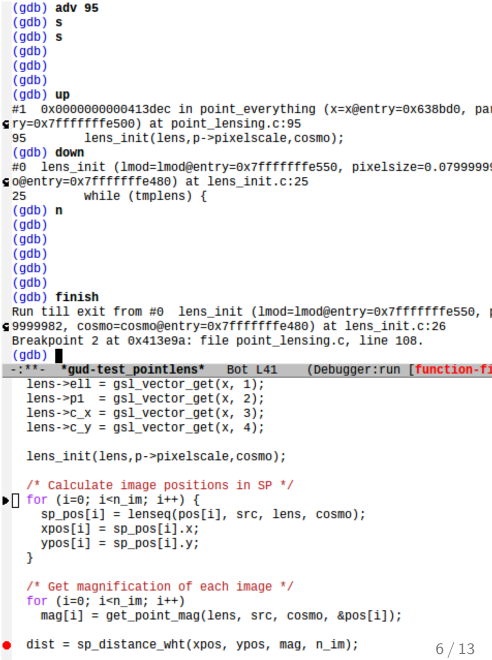
How to run it

- Emacs! The gdb-mode works really well...
- `M-x gdb` *`filename`*
- Set command-line arguments with "`set args arg1 arg2 arg3`"
- Type `run` to run (`r` works, too)

Using gdb from Emacs

gdb commands: moving around

- `break myfunc.c:65`: Breakpoint at line 65 of...
- `r`, `run`: Run the code
- `n`, `next`: Next line
- `s`, `step`: Step into function (if possible, otherwise like `n`)
- `up`, `down`: In the function hierarchy
- `advance X`, `adv X`: Run code to line X
- `finish`: Finish current function, go up one step
- `c`, `continue`: Run again until the next breakpoint



```
(gdb) adv 95
(gdb) s
(gdb) s
(gdb)
(gdb)
(gdb)
(gdb) up
#1  0x0000000000413dec in point_everything (x=x@entry=0x638bd0, par
ry=0x7fffffffe500) at point_lensing.c:95
95          lens_init(lens,p->pixelscale,cosmo);
(gdb) down
#0  lens_init (lmod=lmod@entry=0x7fffffffe550, pixelsize=0.0799999
o@entry=0x7fffffffe480) at lens_init.c:25
25          while (tmplens) {
(gdb) n
(gdb)
(gdb)
(gdb)
(gdb)
(gdb)
(gdb) finish
Run till exit from #0  lens_init (lmod=lmod@entry=0x7fffffffe550, p
9999982, cosmo=cosmo@entry=0x7fffffffe480) at lens_init.c:26
Breakpoint 2 at 0x413e9a: file point_lensing.c, line 108.
(gdb) █
-:**-  *gud-test_pointlens*   Bot L41    (Debugger:run [function-fi
  lens->ell = gsl_vector_get(x, 1);
  lens->p1  = gsl_vector_get(x, 2);
  lens->c_x = gsl_vector_get(x, 3);
  lens->c_y = gsl_vector_get(x, 4);

  lens_init(lens,p->pixelscale,cosmo);

  /* Calculate image positions in SP */
▶ for (i=0; i<n_im; i++) {
    sp_pos[i] = lenseq(pos[i], src, lens, cosmo);
    xpos[i] = sp_pos[i].x;
    ypos[i] = sp_pos[i].y;
  }

  /* Get magnification of each image */
  for (i=0; i<n_im; i++)
    mag[i] = get_point_mag(lens, src, cosmo, &pos[i]);

● dist = sp_distance_wht(xpos, ypos, mag, n_im);
```

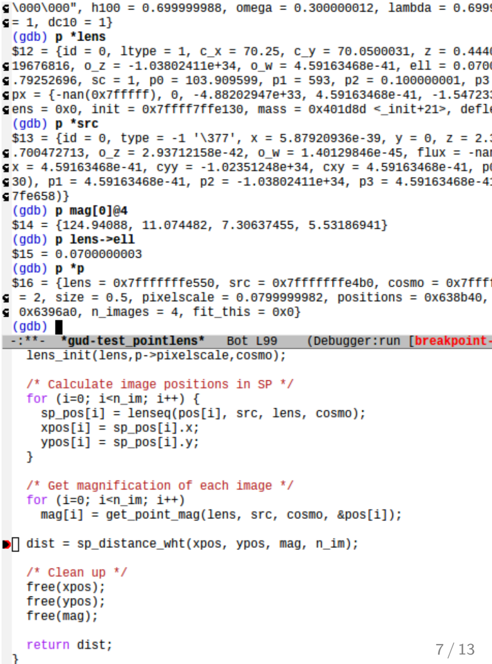
Using gdb from Emacs

gdb commands: looking at your code

- `p, print X`: Print variable `X`; if `X` is a pointer, print adress
- `p *X`: Print the data that pointer `X` points to
- `p &X`: Print the adress (pointer) of datum `X`
- `p X[5]@10`: Print values 5-15 of array `X`

More stuff

- `del 4, delete 4`: Delete breakpoint number 4
- `q, quit`: Exit the debugger



```
\000\000", h100 = 0.699999988, omega = 0.300000012, lambda = 0.6999
= 1, dc10 = 1}
(gdb) p *lens
$12 = {id = 0, ltype = 1, c_x = 70.25, c_y = 70.0500031, z = 0.4440
19676816, o_z = -1.03802411e+34, o_w = 4.59163468e-41, ell = 0.0700
.79252696, sc = 1, p0 = 103.909599, p1 = 593, p2 = 0.100000001, p3
px = {-nan(0x7fffff), 0, -4.88202947e+33, 4.59163468e-41, -1.54723
ens = 0x0, init = 0x7ffff7ffe130, mass = 0x401d8d <_init+21>, defle
(gdb) p *src
$13 = {id = 0, type = -1 '\377', x = 5.87920936e-39, y = 0, z = 2.
.700472713, o_z = 2.93712158e-42, o_w = 1.40129846e-45, flux = -nan
x = 4.59163468e-41, cyy = -1.02351248e+34, cxy = 4.59163468e-41, p0
30), p1 = 4.59163468e-41, p2 = -1.03802411e+34, p3 = 4.59163468e-4
7fe658)}
(gdb) p mag[0]@4
$14 = {124.94088, 11.074482, 7.30637455, 5.53186941}
(gdb) p lens->ell
$15 = 0.0700000003
(gdb) p *p
$16 = {lens = 0x7fffffffe550, src = 0x7fffffffe4b0, cosmo = 0x7ffff
 = 2, size = 0.5, pixelscale = 0.0799999982, positions = 0x638b40,
 0x6396a0, n_images = 4, fit_this = 0x0}
(gdb) 
-:**-  *gud-test_pointlens*   Bot L99    (Debugger:run [breakpoint-
  lens_init(lens,p->pixelscale,cosmo);

  /* Calculate image positions in SP */
  for (i=0; i<n_im; i++) {
    sp_pos[i] = lenseq(pos[i], src, lens, cosmo);
    xpos[i] = sp_pos[i].x;
    ypos[i] = sp_pos[i].y;
  }

  /* Get magnification of each image */
  for (i=0; i<n_im; i++)
    mag[i] = get_point_mag(lens, src, cosmo, &pos[i]);

  dist = sp_distance_wht(xpos, ypos, mag, n_im);

  /* Clean up */
  free(xpos);
  free(ypos);
  free(mag);

  return dist;
}
```

Walkthrough

How not to debug

- Lots of `printfs`

How to debug in a few simple steps

1. Check all compiler warnings – fix them!
2. Check memwatch (or other leak-checking tool) output – fix those issues
3. If the code does not get that far, run it in the debugger and look for anomalies at / before the point where it crashes
4. Use debugger to check what's happening at the places where your code produces strange / unexpected / wrong output
5. Repeat until the code works
6. If it's too slow, move on to profiling

Profiling

What it does

- Check which part of the code takes how long
- Can also check for cache-misses etc.

What it doesn't do

- Tell you how to improve your code!

What profilers are there?

- `gprof` – very basic, but useful (−pg compiler flag)
- `valgrind` – more options (also does memory checking)
- Other, specialized profilers (from google, for intel compiler, CUDAprof, ...)

gprof

How to use gprof

- Turn optimization back on (if wanted)
- Compile with `-pg` flag
- Run your code normally – a file `gmon.out` gets written at the end
- Run `gprof` *`yourcodename`* – you get a list of functions and how often they were called, how much time they took, etc.
- (I usually pipe the output to `less` or a file)

Valgrind

A lot of tools

- Memcheck
- Callgrind
- Cachegrind
- Massif
- Helgrind
- DRD
- And
- Plenty
- More

How to use it

- Compile with `-g` (debug) flag
- Call `valgrind --tool=X mycode`
- For memcheck: Can specify output file, otherwise look at summary at the end
- Callgrind writes `callgrind.out.PID` file after running your code (same for Cachegrind)
- `callgrind_annotate cachegrind.out.PID` will print formatted output (we'll look at it later)
- `callgrind_annotate cachegrind.out.PID myfile.c` prints the number of calls, number of cache misses, etc. next to each line in your source file

General differences

memwatch, gprof

- No extra program needed to run it
- Need special compiler flags / extra code
- Fast! Very little runtime difference with / without
- Rather basic, but still important / usable output

valgrind

- An extra program runs your code
- Works without special preparation in the code
- Slow! Ca. factor 10-100 slower than normal execution
- Very detailed output
- Lots more functionality

Summary and list of tools

Debugging

- memwatch
- valgrind (memcheck is the default tool)
- gdb

Profiling

- gprof
- valgrind --tool=callgrind
- valgrind --tool=cachegrind
- callgrind_annotate
- kcachegrind – graphical output for cachegrind.out.PID files