Pablo Alcain
pabloalcain@gmail.com

Herramientas para debugging y profiling

Debuggear

Objetivo final: quitar los errores del programa

En un mundo ideal, nadie necesita debuggear

Debuggear es mucho más que quitar los errores

- Saber que están
- Saber dónde están
- Quitarlos

Muchos programadores no saben debuggear

- Es un *soft skill*: no agrega funcionalidad ni mejora la ciencia

Probablemente la habilidad que más intución requiere

- Cuanto mejor sea un desarrollador, mejor va a *saber* debuggear

Bugs (errores)

Errores fáciles

- El programa se rompe (segfault)
- El programa se cuelga (*side note: The Halting Problem*)

¿Por qué son fáciles?

- **Saber que están**
- **Saber dónde están**
- **Quitarlos**

- Cuanto mejor sea un desarrollador, mejor va a *saber* debuggear

Bugs (errores)

Errores medios

- El programa da resultados erróneos
- El programa da NaN o inf (generalmente **FPE**)

¿Por qué son medios?

- **Saber que están**
- **Saber dónde están**
- **Quitarlos**

Bugs (errores)

Errores difíciles

- El programa da resultados erróneos en unos pocos casos
- Sólo aparece en ciertos niveles de optimización
- Sólo aparece en condiciones extremas

¿Por qué son difíciles?

- **Saber que están**
- **Saber dónde están**
- **Quitarlos**

y... son difíciles de reproducir

Errores: el punto de vista del desarrollador

Oportuncrisis!

Aprendan del programa/lenguaje que escriben

- El error es porque no entendimos bien algo del programa

Aprendan del error que cometen

- ¿Por qué lo cometieron?
- ¿Cómo podrían haberlo evitado?
- ¿Hay otros errores similares en el código?

Aprendan a debuggear

- ¿Cómo pueden encontrar este error más fácilmente?

WTPC
Workshop en técnicas de programación científica

¿Cómo debuggear?

Imprimir mensajes

```
...
printf(“I'm here!\n”);
buggy_code();
printf(“Now I'm there\n”);
...
```

Utilizar un debugger

- Core analysis
- Insertar (*attach*) el debugger a un programa corriendo
- Correr el programa desde cero con el debugger

Pedir ayuda

¿Para qué es un debugger?

Mas (**mucha más**) información que un par de prints

Pueden elegir ejecutar paso a paso cada función

Incluso cada sentencia de assembler

(en tu cara, printf)

Análisis post-mortem

No sustituye pensar

Pero pensar no sustituye a un buen debugger

¿Qué hace un debugger?

Obtiene información del objeto

Compilando con `-g` se aseguran que dé mucha información

`gcc` permite información de debug y optimización

- partes del código o variables pueden ser “opt'd out”
- optimizar modifica el código: no se corresponde a la fuente
- la única solución es compilar con `-O0` (por defecto tiene ciertas opt)

Los bugs difíciles son **difíciles**

¿Cómo uso un debugger? gdb

```
$ gdb ./my_program.e
...
(gdb)                                          prompt
```

```
(gdb) break <ident>    se detiene la ejecución si llega a ident
(gdb) watch <ident>    se detiene la ejecución ident cambia
(gdb) run              ejecuta el programa
(gdb) continue         continúa la ejecución
(gdb) next             siguiente paso (sin entrar en subrutinas)
(gdb) step             siguiente paso (entrando en subrutinas)
(gdb) print <ident>    imprime el valor de ident
(gdb) delete <bpn>     borra el breakpont
```

WTPC
Workshop en técnicas de programación científica

Otro “debugger”: valgrind

`valgrind` es mucho más que un debugger

Es una virtual machine: permite ver todo

Es una virtual machine: es súper lento

```
$ valgrind --leak-check=full --track-origins=yes ./my_program.e
```

También puede hacer un profiling muy exhaustivo

Otro “debugger”: ¡otra persona!

No teman en pedir ayuda

Pero no pidan ayuda que está en otros lados

Pero no pidan ayuda si no se esforzaron

Pero no pidan ayuda si no muestran cómo reproducir el problema

Pero no pidan ayuda sin decir qué versión, compilador, etc... usan

Pero no pidan ayuda si a los demás no les es fácil ayudar

Otro “debugger”: ¡otra persona!

Algunos ejemplos famosos de interacción

```
Mauro, SHUT THE FUCK UP!

It's a bug alright - in the kernel. How long have you been a maintainer? And
you *still* haven't learnt the first rule of kernel Maintenance?

If a change results in user programs breaking, it's a bug in the kernel. We
never EVER blame the user programs. How hard can this be to Understand?
...
Shut up, Mauro. And I don't _ever_ want to hear that kind of obvious garbage
and idiocy from a kernel maintainer again. Seriously.
...
WE DO NOT BREAK USERSPACE!

Seriously. How hard is this rule to understand? We particularly don't break
user space with TOTAL CRAP. I'm angry, because your whole email was so
_horribly_ wrong, and the patch that broke things was so obviously crap
```

Otro “debugger”: ¡otra persona!

Algunos ejemplos famosos de interacción

```
> When I first looked at Git source code two things struck me as odd:
> 1. Pure C as opposed to C++. No idea why. Please don't talk about
portability,
> it's BS.

*YOU* are full of bullshit.

C++ is a horrible language. It's made more horrible by the fact that a lot of
substandard programmers use it, to the point where it's much much easier to
generate total and utter crap with it. Quite frankly, even if the choice of C
were to do *nothing* but keep the C++ programmers out, that in itself would be
a huge reason to use C.
...
So I'm sorry, but for something like git, where efficiency was a primary
objective, the "advantages" of C++ is just a huge mistake. The fact that we
also piss off people who cannot see that is just a big additional advantage.
```

Otro “debugger”: ¡otra persona!

Algunos ejemplos famosos de interacción

```
i didn't say it was wrong. it is just a horrible style or programming.
almost as bad as computed GOTOs in fortran 4.

...

second, there is useless use of the pow() function (what is x**0 ?)

...

let me have you participate in a little "secret": pow(x,0.) for any
valid choice of x is simply 1.0
```

Profiling

¿Cómo sabemos si el código es rápido? **midiendo**

Con un cronómetro externo

Con un cronómetro interno

Con un timer (tic...toc)

Con un profiler

Profiling: gprof

```
$ gcc code.c -pg -o program.e
$ ls -tr
code.c program.e
$ ./program.e
$ ls -tr
code.c program.e gmon.out
$  gprof program.e gmon.out
....
<all the profiling info>
```

Profiling: gprof

```
$ gcc code.c -pg -o program.e
$ ls -tr
code.c program.e
$ ./program.e
$ ls -tr
code.c program.e gmon.out
$  gprof program.e gmon.out > profile.info
$ ls -tr
code.c program.e gmon.out profile.info
```

Profiling: gprof

```
$ gcc code.c -pg -o program.e
$ ls -tr
code.c program.e
$ ./program.e
$ ls -tr
code.c program.e gmon.out
$  gprof program.e gmon.out > profile.info
$ ls -tr
code.c program.e gmon.out profile.info
```

Cuando prendemos optimizaciones, el profiling puede ser mentiroso

Profiling: perf

Profiling: perf

Aprovecha que los CPUs modernos tienen contadores internos

Linux tiene acceso: muy poco overhead, muy confiable

```
$ gcc code.c -o program.e
$ perf stat ./program.e
Performance counter stats for './calc.e'
.....
```

Profiling: perf

Aprovecha que los CPUs modernos tienen contadores internos

Linux tiene acceso: muy poco overhead, muy confiable

```
$ gcc code.c -o program.e
$ perf record ./progam.e
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.013 MB perf.data (~586
samples) ]
$ perf report -i perf.data
```

Pablo Alcain
pabloalcain@gmail.com

Herramientas para debugging y profiling