

# Debugging and Profiling

Carlos García Campos  
*carlosgc@gsync.es*



- The GNU Debugger
- Supports C, C++, and Modula-2 programs
- Three ways to debug with gdb
  - Start a program, debug it and stop it
  - Debug a running process
  - Debug a core file
- See the refcard<sup>1</sup> for a complete summary of the GDB commands

---

<sup>1</sup><http://www.cs.dal.ca/student-services/refcards/gdbref.pdf>



# Basic GDB commands

## ● Starting GDB

```
$ gdb ./nocrash
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show
copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb)
```

## ● Run

```
(gdb) run
Starting program:
./nocrash

Program exited normally.
(gdb)
```

*See the complete example (attachment nocrash.c)*



# Basic GDB commands (cont.)

- **Setting a break point and display expressions values**

```
(gdb) break print_AB
Breakpoint 1 at 0x804851b: file nocrash.c, line 53.
(gdb) run
Starting program:
./nocrash

Breakpoint 1, print_AB (ab=0x91f1008) at nocrash.c:53
53printf ("%s %s\n", ab->a, ab->b);
(gdb) print ab
$1 = (AB *) 0x91f1008
(gdb) print ab->a
$2 = 0x91f1018 "Hello"
(gdb) print ab->b
$3 = 0x91f1028 "World"
(gdb)
```

- **Getting a backtrace**

```
(gdb) bt
#0  0xb7e1ffa9 in ?? () from /lib/tls/i686/cmov/libc.so.6
#1  0xb7e205b6 in free () from /lib/tls/i686/cmov/libc.so.6
#2  0x080484b0 in destroy_AB (ab=0x91f1008) at nocrash.c:29
#3  0x0804859a in main () at nocrash.c:70
```

- **Killing the program**

```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb)
```



# Valgrind

- Valgrind is a framework for building program analysis tools
- It includes several tools
  - Memory checker (memcheck)
  - Cache profiler (cachegrind)
  - Call graph profiler (callgrind)
  - Heap profiler (massif)
  - Race conditions spotter (helgrind)
- Memcheck is the most widely used tool



# Valgrind: Memecheck

Memcheck can detect the following problems

- Use of uninitialised memory
- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks
  - Definitely lost
  - Possibly lost
  - Still reachable
- Mismatched use of malloc/new/new [] vs free/delete/delete []
- Overlapping src and dst pointers in memcpy() and related functions



# Valgrind: Running Memecheck

## Common options

- `-undef-value-errors=<yes|no>` [default: yes]
- `-track-origins=<yes|no>` [default: no]
- `-leak-check=<no|summary|yes|full>` [default: summary]
- `-show-reachable=<yes|no>` [default: no]

Example: (*attachment error.c*)

```
$ valgrind --track-origins=yes --leak-check=full --show-reachable=yes ./error
```



# Valgrind: Memcheck output

- Use of uninitialised memory

```
==11834== Conditional jump or move depends on uninitialised value(s)
==11834==    at 0x8048539: main (error.c:24)
==11834==    Uninitialised value was created by a stack allocation
==11834==    at 0x80484D2: main (error.c:9)
==11834==
==11834== Syscall param write(buf) points to uninitialised byte(s)
==11834==    at 0x40007F2: (within /lib/ld-2.9.so)
==11834==    by 0x405E774: (below main) (in /lib/tls/i686/cmov/libc-2.9.so)
==11834== Address 0x41ac148 is 0 bytes inside a block of size 30 alloc'd
==11834==    at 0x402703E: malloc (vg_replace_malloc.c:177)
==11834==    by 0x8048597: main (error.c:36)
==11834==    Uninitialised value was created by a heap allocation
==11834==    at 0x402703E: malloc (vg_replace_malloc.c:177)
==11834==    by 0x8048597: main (error.c:36)
```



# Valgrind: Memcheck output (cont.)

- Reading/writing off the end of malloc'd blocks

```

==11834== Invalid read of size 4
==11834==    at 0x804855C: main (error.c:28)
==11834==   Address 0x41ac118 is 4 bytes after a block of size 20 alloc'd
==11834==    at 0x402703E: malloc (vg_replace_malloc.c:177)
==11834==   by 0x8048552: main (error.c:26)
==11834==
==11834== Invalid write of size 4
==11834==    at 0x804857F: main (error.c:32)
==11834==   Address 0x41ac120 is 12 bytes after a block of size 20 alloc'd
==11834==    at 0x402703E: malloc (vg_replace_malloc.c:177)
==11834==   by 0x8048552: main (error.c:26)

```

- Reading/writing memory after it has been free'd

```

==11834== Invalid write of size 4
==11834==    at 0x8048666: main (error.c:51)
==11834==   Address 0x41ac108 is 8 bytes inside a block of size 20 free'd
==11834==    at 0x4025E5A: free (vg_replace_malloc.c:293)
==11834==   by 0x8048654: main (error.c:48)
==11834==
==11834== Invalid read of size 4
==11834==    at 0x8048689: main (error.c:59)
==11834==   Address 0x41ac198 is 0 bytes inside a block of size 20 free'd
==11834==    at 0x4025E5A: free (vg_replace_malloc.c:293)
==11834==   by 0x8048685: main (error.c:57)

```



# Valgrind: Memcheck output (cont.)

- Mismatched use of malloc/new/new [] vs free/delete/delete []

```

==11834== Invalid free() / delete / delete[]
==11834==   at 0x4025E5A: free (vg_replace_malloc.c:293)
==11834==   by 0x804858B: main (error.c:34)
==11834== Address 0x41ac0b2 is 2 bytes inside a block of size 30 alloc'd
==11834==   at 0x402703E: malloc (vg_replace_malloc.c:177)
==11834==   by 0x8048520: main (error.c:19)
==11834==
==11834== Invalid free() / delete / delete[]
==11834==   at 0x4025E5A: free (vg_replace_malloc.c:293)
==11834==   by 0x804865F: main (error.c:49)
==11834== Address 0x41ac100 is 0 bytes inside a block of size 20 free'd
==11834==   at 0x4025E5A: free (vg_replace_malloc.c:293)
==11834==   by 0x8048654: main (error.c:48)

```

- Overlapping src and dst pointers in memcpy() and related functions

```

==11834== Source and destination overlap in memcpy(0xBEF5BED0, 0xBEF5BEBC, 21)
==11834==   at 0x4027C29: memcpy (mc_replace_strmem.c:380)
==11834==   by 0x804860F: main (error.c:44)
==11834==
==11834== Source and destination overlap in memcpy(0xBEF5BEBC, 0xBEF5BEBC, 21)
==11834==   at 0x4027C29: memcpy (mc_replace_strmem.c:380)
==11834==   by 0x8048649: main (error.c:46)

```



# Valgrind: Memcheck output (cont.)

## ● Memory leaks

```

==11834== ERROR SUMMARY: 28 errors from 23 contexts (suppressed: 0 from 0)
==11834== malloc/free: in use at exit: 70 bytes in 3 blocks.
==11834== malloc/free: 6 allocs, 5 frees, 130 bytes allocated.
==11834== For counts of detected errors, rerun with: -v
==11834== searching for pointers to 3 not-freed blocks.
==11834== checked 53,580 bytes.

```

### ● Definitely lost

```

==11834== 10 bytes in 1 blocks are definitely lost in loss record 1 of 3
==11834==    at 0x402703E: malloc (vg_replace_malloc.c:177)
==11834==    by 0x80484F7: main (error.c:11)

```

### ● Still reachable

```

==11834== 30 bytes in 1 blocks are still reachable in loss record 2 of 3
==11834==    at 0x402703E: malloc (vg_replace_malloc.c:177)
==11834==    by 0x8048597: main (error.c:36)

```

### ● Possibly lost

```

==11834== 30 bytes in 1 blocks are possibly lost in loss record 3 of 3
==11834==    at 0x402703E: malloc (vg_replace_malloc.c:177)
==11834==    by 0x8048520: main (error.c:19)

```



# Valgrind: Massif

- It measures how much heap memory a program uses
- It can also measure the size of a program's stack(s).  
Disabled by default
- It might also help to find memory leaks that memcheck doesn't detect, because memory is never lost, but it's not in use either
- It provides detailed information about which parts of the program are responsible for allocating the head memory



# Valgrind: Running Massif

Example: (*attachment massif.c*)

```
$ valgrind --tool=massif --time-unit=B ./massif
```

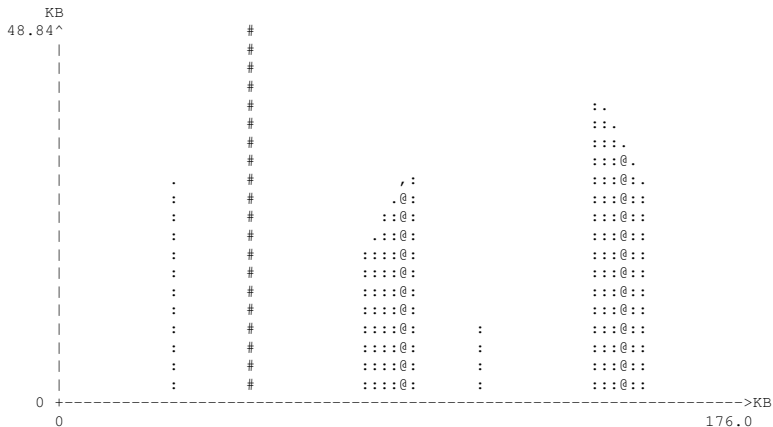
- It doesn't print any statistics, instead it writes everything into a file called `massif.out.<pid>`
- In order to make this file more “human-readable”, we need to use the `ms_print` script

```
$ ms_print massif.out.14579 > massif-example.txt
```

- The output includes a graph showing the memory consumption and detailed information about the allocations



# Valgrind: The Massif Graph



Number of snapshots: 28

Detailed snapshots: [3 (peak), 13, 23]



# Valgrind: The Massif Graph (cont.)

- Massif takes snapshots (that is, a measurement of the memory usage at a certain point in time) that are represented in the graph by every vertical bar
  - Normal snapshots: only basic information is recorded for them. Represented by bars consisting of ':' and '' characters
  - Detailed snapshots: Detailed information about where allocations happened is recorded. Represented by bars consisting of '@' and ',' character
  - Peak snapshots: there's at most one. It's a detailed snapshot representing the greatest memory consumption point



# Valgrind: Massif snapshot details

- After the graph, there's detailed information about the snapshots.
- There's a table containing information about every snapshot
  - Its number id
  - The time
  - Total memory consumed at that point
  - Useful bytes allocated in the heap at that point
  - Extra bytes allocated in the heap at that point
  - The size of the stack
- Additionally, for detailed snapshots, the allocation tree is provided. This tree contains the stack trace of every single allocation which gives a complete picture of how and why all heap memory was allocated



## Valgrind: Massif snapshot details (cont.)

```

-----
n           time (B)           total (B)  useful-heap (B)  extra-heap (B)  stacks (B)
-----
0             0                0          0                0                0
1          30,008            30,008      30,000            8                0
2          50,016            50,016      50,000           16                0
3          50,016            50,016      50,000           16                0
99.97% (50,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->59.98% (30,000B) 0x8048404: bar (massif.c:8)
| ->59.98% (30,000B) 0x8048417: foo (massif.c:18)
|   ->59.98% (30,000B) 0x8048450: main (massif.c:33)
|
->39.99% (20,000B) 0x8048426: foo (massif.c:20)
  ->39.99% (20,000B) 0x8048450: main (massif.c:33)

```

- There are two allocations at this point:
  - Line 8:  $p = \text{malloc}(30000)$ ;
  - Line 20:  $p2 = \text{malloc}(20000)$ ;
- The total size is 50,016 Bytes which means this snapshot was taken between lines 20 and 22.



# Sysprof

- Sysprof is a sampling CPU profiler for Linux
- Profiles all running processes showing the time spent in each branch of the call tree
- It uses a kernel module that needs to be added before running Sysprof.
- It only supports the i386 and x86-64 architectures



# Running Sysprof

- First we have to load the kernel module

```
$ sudo modprobe sysprof-module
```

- And finally launch the GUI program

```
$ sudo sysprof
```



## Sysprof GUI


System Profiler

Profiler View Help

Samples: 82

Functions	Self	Total	Descendants	Self	Cumulative
Everything	0,00 %	100,00 %	▼ [epiphany]	0,00 %	54,88 %
__libc_start_main	0,00 %	80,49 %	▼ _start	0,00 %	45,12 %
[epiphany]	0,00 %	54,88 %	▼ __libc_start_main	0,00 %	45,12 %
gtk_main	0,00 %	50,00 %	▼ main	0,00 %	45,12 %
g_main_loop_run	0,00 %	50,00 %	▶ gtk_main	0,00 %	45,12 %
g_main_context_iterate	0,00 %	48,78 %	▼ __clone	0,00 %	4,88 %
_start	0,00 %	45,12 %	▶ start_thread	0,00 %	4,88 %
main	0,00 %	45,12 %	▶ No map ([epiphany])	0,00 %	2,44 %
g_main_context_dispatch	2,44 %	39,02 %	▶ In file [heap]	0,00 %	1,22 %
kernel	0,00 %	31,71 %	__kernel_vsyscall	1,22 %	1,22 %
In file /usr/lib/xulrunner-1.9.0.7/libxul.so	8,54 %	30,49 %			
__kernel_vsyscall	2,44 %	26,83 %			
g_io_unix_dispatch	0,00 %	25,61 %			
...enter_de_cell	0,00 %	24,38 %			

Callers	Self	Total
Everything	0,00 %	54,88 %



# strace

- Intercepts and records the system calls which are called by a process as well as the signals received
- Each line in the trace contains the system call name, followed by its arguments in parentheses and its return value

```
open("/dev/null", O_RDONLY) = 3
open("/foo/bar", O_RDONLY) = -1 ENOENT (No such file or directory)
```



# Running strace

- Without arguments, its output is printed to stderr

```
$ strace command
```

- Useful options when profiling

- -ttt: Prefix each line of the trace with the time as the number of seconds since the epoch including microseconds too
- -f: Trace child processes as they are created by currently traced processes
- -o: Write the trace output to the file filename rather than to stderr

```
$ strace -ttt -f -o program.strace program
```



# Using strace for time profiling

- Why does my application take so long to do ...?
- First of all we need to identify which parts of the code are the slowest
- We can add traces to our program that are visible in a strace output with the following function:

```
#include <unistd.h>
static void
program_log (const char *format, ...)
{
    va_list args;
    char *formatted, *str;

    va_start (args, format);
    formatted = g_strdup_vprintf (format, args);
    va_end (args);

    str = g_strdup_printf ("MARK: %s: %s", g_get_prpname(),
                          %formatted);

    g_free (formatted);

    access (str, F_OK);
    g_free (str);
}
```



# Using strace for time profiling

- The function is specific for GLib applications, but it's easy to write something similar without using GLib at all
- The key point of the function is the access system call
- access checks whether the calling process has permissions to access the given filename
- By providing a custom string as filename, access will always fail, but it will appear in the strace output

```
21830 1237557677.223326 access("MARK: evince: parsing arguments",  
    F_OK) = -1 ENOENT (No such file or directory)
```

- We can now parse the strace output to see how long it takes between the marks added to the code



# strace: using plot-timeline.py script

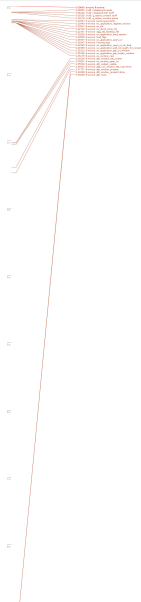
- plot-timeline.py is a script that parses the strace output and plots a graph with the results

```
$ python plot-timeline.py -o graph.png program.strace
```

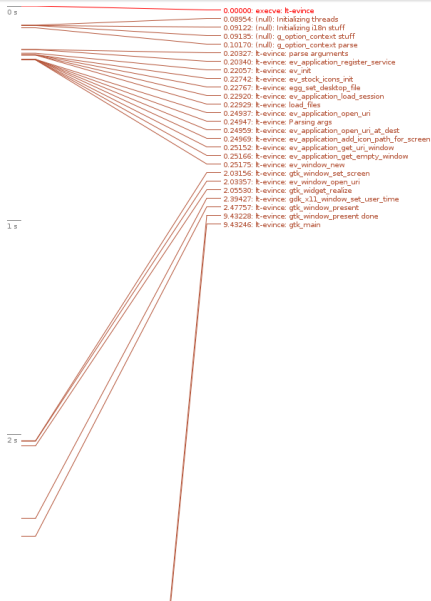
*(plot-timeline.py script is available as an attachment to this document)*



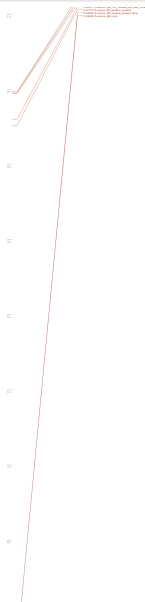
# strace: plot-timeline.py output graph



## strace: plot-timeline.py output graph



# strace: plot-timeline.py output graph



# strace: plot-timeline.py output graph

- Looking at the graph we can clearly identify two point to worry about
  - `ev_window_new`: it takes a long time compared to the other functions. Is it normal? Can we improve it somehow?
  - `gtk_window_present`: there's definitely something weird there. What's going on?
- We should always start looking at the function that take the longest.

*(The original graph file is attached to this document:  
ev-timeline.png)*

