Contents

Articles

References

Article Licenses

Linux Applications Debugging Techniques

Preamble

A hands-on guide to debug applications under Linux and to build your own debugging tools. Probably useful for debugging dogs. Partly applicable to other Unices. Work in progress.

Authors

Aurelian Melinte

Table of Contents

Using GDB

Preparations

Someday some hard to reproduce issue will be found on a production machine. Typically, such a machine is difficult to access, has no development environment and nothing can be installed on it. At most it will have `gdb`, but very likely not. Typically also, the heaviest user will be the one to find the issue and typically still, the heaviest user is the one being the bigger -- money-wise. And that issue has to be root cause diagnosed and fixed.

Unless the application has been prepared for this forensic gathering moment, not much can be done to diagnose where the issue is. Thus, preparations should start with compilation:

- Have a "symbol server" and make sure it is reacheable. Compile on the symbol server.
- Compile the release with debugging symbols. Strip them if you do not want to ship them but keep them.
- Ship gdbserver with the application for remote debugging.

These preparations will allow one to:

- Debug the application running on any machine, including machines where there is no `gdb` installed.
- Debug from any machine that has network visibility to the symbol server.

Also, think beforehand how would you debug on the machine:

- Embed a breakpoint in the code, at hard of reach places of interest, then
  - Start the application
  - Attach to it with the debugger
  - Wait until the breakpoint is hit

The "symbol server"

One way to easily reach the right code from within the debugger is to build the binaries within an auto-mounted folder, each build in its own sub-folder. The same auto-mount share should be accessible from the machine you are debugging on.

Debian

- Install autofs
- In /etc/auto.master uncomment the line:

```
/net      -hosts
```

- In `/etc/exports` export the folder:

```
# Build directory: rw for localhost, read-only for any other host
/home/amelinte/projects/lpt/lpt  localhost(rw,sync,no_subtree_check)
*(ro,root_squash,no_subtree_check)
```

- As root: restart autofs & nfs and export the build share:

```
/etc/init.d/autofs restart
/etc/init.d/nfs-kernel-server restart
/usr/sbin/exportfs -a
```

Redhat

- Export the folder: edit `/etc/exports`
- As root (RedHat): `service autofs start`

Finally, build the binaries within the automounted directory on the build machine (here the build machine is `bear`):

```
cd /net/bear/home/amelinte/projects/lpt/lpt.destructor
make clean
make
```

Notice the filename path that is resolved with the symbol information:

```
[0x413464] lpt::stack::call_stack<40ul>::call_stack(bool)+0x52
        At
/net/bear/home/amelinte/projects/lpt/lpt.destructor/lpt/include/lpt/call_stack.hpp:74
        In binaries/bin/stacktest
```

If the symbols have been stripped from the binaries, point `gdb` to the folders where the symbols are with the `debug-file-directory` directive.

References

- Debugging Information in Separate Files [1]

Remote debugging

- On the machine where the application runs (`appmachine`):
  - If gdbserver is not present , copy it over.
  - Start the application.
  - Start gdbserver: `gdbserver gdbmachine:2345 --attach program`
- On `gdbmachine`:
  - At the gdb prompt, enter: `target remote appmachine:2345`

Sometimes you may have to tunnel over ssh:

- On gdbmachine:
  - `ssh -L 5432:appmachine:2345 user@appmachine`
  - At the gdb prompt: `target remote localhost:5432`

References

- GDB Tunneling [2]

Attaching to a process

Find out the PID of the process, then:

```
(gdb) attach 1045
Attaching to process 1045
Reading symbols from /usr/lib64/firefox-3.0.18/firefox...(no debugging
symbols found)...done.
Reading symbols from /lib64/libpthread.so.0...(no debugging symbols
found)...done.
[Thread debugging using libthread_db enabled]
[New Thread 0x448b4940 (LWP 1063)]
```

```
[New Thread 0x428b0940 (LWP 1054)]
....
(gdb) detach
```

Embedding breakpoints in the source

On x86 platforms:

```
#define EMBEDDED_BREAKPOINT  asm volatile ("int3;")
```

Or a more elaborate one:

```
#define EMBEDDED_BREAKPOINT \
    asm("0:"                                \
        ".pushsection embed-breakpoints;" \
        ".quad 0b;"                         \
        ".popsection;")
```

This will break into the debugger on hard to reach conditions:

```
    if (cpuload > 3.1416 && started > 111*MINS*AGO &&
whatever-dynamic-condition)
    {
        EMBEDDED_BREAKPOINT;
    }
```

References

- http://mainisusuallyafunction.blogspot.com/2012/01/embedding-gdb-breakpoints-in-c-source.html [3]

Data breakpoints (watchpoints)

Watchpoints can be imlemented either in software either in hardware if the CPU supports it. Typically on an Intel processor there are eight debug registers out of which only four can be used for hardware breakpoints and this limits the number of watchpoints system wide.

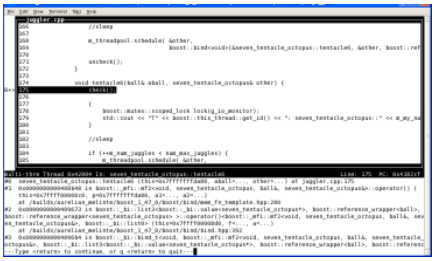
The text user interface

GDB features a text user interface for code, disassembler and registers. For instance:

- **Ctrl-x 1** will show the code pane
- **Ctrl-x a** will hide the TUI panes

None of the GUI interfaces to `gdb` (Qt Creator stands out for being intuitive and easy to use) can offer access to all of the `gdb` functionality.

curses gdb [4] offers an improved TUI. A comprehensive list of debugger GUIs is available here [5].



GDB TUI

Reverse debugging

As an example, reverse debugging is a functionality no GUI offers access to:

```
(gdb) l
1       /* 1 */ int var;
2       /* 2 */ int main (void) {
3       /* 3 */   int i; for (i=0;i<100;i++)
4       /* 4 */     var=i/10;
5       /* 5 */  return 0; }

(gdb) start
(gdb) record
(gdb) adv 5
main () at history.c:5
5       /* 5 */  return 0; }

(gdb) watch var
Hardware watchpoint 2: var

(gdb) reverse-continue
Continuing.
Hardware watchpoint 2: var

Old value = 9
New value = 8
0x00000000004004c3 in main () at history.c:4
4       /* 4 */     var=i/10;
(gdb) p i
$1 = 90

(gdb) reverse-continue
Continuing.
Hardware watchpoint 2: var

Old value = 8
New value = 7
0x00000000004004c3 in main () at history.c:4
4       /* 4 */     var=i/10;

(gdb) p i
$2 = 80
```

Register watch

You can watch registers. Note this will force the debugger to single step the debugged program and it will run very slowly:

```
(gdb) watch $ebp
```

References

- http://sourceware.org/gdb/onlinedocs/gdb/TUI.html [6]
- http://sourceware.org/gdb/news/reversible.html [7]

.gdbinit

As a note, in upcoming gdb releases, `.gdbinit` will be replaced by `gdb-gdb.gdb`:

```
gdb-gdb.gdb
 ^   ^   ^
 ^   ^   ^---- It's a gdb script.
 ^   ^         If it were Python this would be .py.
 ^   ^
 ^   ^-------- "-gdb" is a gdb convention, it's the suffix added to a
file
 ^             for auxiliary support.
 ^             E.g., gdb will auto-load libstdc++.so-gdb.py (version
elided)
 ^             which contains the std c++ pretty-printers.
 ^
 ^------------ This init script is for the program named "gdb".
               If this were for readelf the script would be named
               readelf-gdb.gdb.
```

C++ support

Canned gdb macros

- gdb STL support [8]
- STL macros (and more) [9]

Mangling

`gdb` might need a bit of guidance with C++11 binaries:

```
(gdb) set cp-abi gnu-v3
(gdb) set language c++
(gdb) maintenance demangle _ZNSt16nested_exceptionD2Ev
std::nested_exception::~nested_exception()
```

- [10]

References

[1] http://sourceware.org/gdb/current/onlinedocs/gdb/Separate-Debug-Files.html
[2] http://www.cucy.net/lacp/archives/000024.html
[3] http://mainisusuallyafunction.blogspot.com/2012/01/embedding-gdb-breakpoints-in-c-source.html
[4] http://cgdb.github.com/
[5] http://www.drdobbs.com/testing/13-linux-debuggers-for-c-reviewed/240156817?pgno=1
[6] http://sourceware.org/gdb/onlinedocs/gdb/TUI.html
[7] http://sourceware.org/gdb/news/reversible.html
[8] http://sourceware.org/gdb/wiki/STLSupport
[9] http://www.yolinux.com/TUTORIALS/GDB-Commands.html#STLDEREF
[10] http://mentorembedded.github.io/cxx-abi/abi.html#mangling

Core files

A core dump is a snaphot of the memory of the program, processor registers including program counter and stack pointer and other OS and memory management information, taken at a certain point in time. As such, they are invaluable for capturing the state of rare occurring races and abnormal conditions.

What is more, such rarities will be found usually on heavily used production or QA machines where `gdb` is not available, nor is access easy to the machine. Worse, the heaviest users are usually the biggest clients (moneywise...). As such, it is important to get as much forensic data as available, and plan for it.

One can force a core dump from within the program or from outside at chosen moments. What a core cannot tell is how the application ended up in that state: the core is no replacement for a good log. Verbose logs and core files go hand in glove.

Prerequisites

For a process to be able to dump core, a few prerequisites have to be met:

- the set core size limit should permit it (see the man page for ulimit). E.g.: `ulimit -c unlimited`. It can also be set from within the program.
- the process to dump core should have write permissions to the folder where the core is to be dumped to (usually the current working directory of the process)

Where is my core?

Usually the core is dumped in the current working directory of the process. But the OS can be configured otherwise:

```
# cat /proc/sys/kernel/core_pattern
%h-%e-%p.core

# sysctl -w "kernel.core_pattern=/var/cores/%h-%e-%p.core"
```

Dumping core from outside the program

One possibility is with gdb, if available. This will let the program running:

```
(gdb) attach <pid>
(gdb) generate-core-file <optional-filename>
(gdb) detach
```

Another possibility is to signal the process. This will terminate it, assuming the signal is not caught by a custom signal handler:

```
kill -s SIGABRT <pid>
```

Dumping core from within the program

Again, there are two possibilities: dump core and terminate the program or dump and continue:

```
void dump_core_and_terminate(void)
{
    /*
     * Alternative:
     *   char *p = NULL; *p = 0;
     */
    abort();
}

void dump_core_and_continue(void)
{
    pid_t child = fork();
    if (child < 0) {
        /*Parent: error*/
    }
    else if (child == 0) {
        dump_core_and_terminate(); /*Child*/
    }
    else {
        /*Parent: continue*/
    }
}
```

Note: use `dump_core_and_continue()` with care: in a multi-threaded program, the forked child will have only a clone of the parent thread that called `fork()` [Butenhof Ch5; re: threads & fork]. This has number of implications, in particular with respect to mutexes, but the particular point here is that the core that the child will dump will contain information only for one thread. If you need to dump a core with all threads without aborting the process, try to use the google core dumper library [1], even if it has not been maintained for years.

Shared libraries

To obtain a good call stack, it is important that the gdb loads the same libraries that were loaded by the program that generated the core dump. If the machine we are analyzing the core has different libraries (or has them in different places) from the machine the core was dumped, then copy over the libraries to the analyzing machine, in a way that mirrors the dump machine. For instance:

```
$ tree .
.
|-- juggler-29964.core
|-- lib64
|   |-- ld-linux-x86-64.so.2
|   |-- libc.so.6
|   |-- libm.so.6
|   |-- libpthread.so.0
|   `-- librt.so.1
...
```

At the gdb prompt:

```
(gdb) set solib-absolute-prefix ./

(gdb) set solib-search-path .

(gdb) file

../../../../../threadpool/bin.v2/libs/threadpool/example/juggler/gcc-4.1.2/debug/link-static/threading-multi/juggler

Reading symbols from

/home/aurelian_melinte/threadpool/threadpool-0_2_5-src/threadpool/bin.v2/libs/threadpool/example/juggler/gcc-4.1.2/debug/link-static/threading-multi/juggler...done.

(gdb) core-file juggler-29964.core

Reading symbols from ./lib64/librt.so.1...(no debugging symbols

found)...done.

Loaded symbols for ./lib64/librt.so.1

Reading symbols from ./lib64/libm.so.6...(no debugging symbols

found)...done.

Loaded symbols for ./lib64/libm.so.6

Reading symbols from ./lib64/libpthread.so.0...(no debugging symbols

found)...done.

Loaded symbols for ./lib64/libpthread.so.0

Reading symbols from ./lib64/libc.so.6...(no debugging symbols

found)...done.

Loaded symbols for ./lib64/libc.so.6

Reading symbols from ./lib64/ld-linux-x86-64.so.2...(no debugging

symbols found)...done.

Loaded symbols for ./lib64/ld-linux-x86-64.so.2

Core was generated by

`../../../../bin.v2/libs/threadpool/example/juggler/gcc-4.1.2/debug/link-static/'.

Program terminated with signal 6, Aborted.

#0  0x0000003684030265 in raise () from ./lib64/libc.so.6

(gdb) frame 2

#2  0x0000000000404ae1 in dump_core_and_terminate () at juggler.cpp:30
```

analyze-cores

Here is a script that will generate a basic report per core file. Useful the days when cores are raining on you:

```
#!/bin/bash

#
# A script to extract core-file informations
#


if [ $# -ne 1 ]
then
  echo "Usage: `basename $0` <for-binary-image>"
  exit -1
else
  binimg=$1
fi


# Today and yesterdays cores
cores=`find . -name '*.core' -mtime -1`

#cores=`find . -name '*.core'`


for core in $cores
do
  gdblogfile="$core-gdb.log"
  rm $gdblogfile

  bininfo=`ls -l $binimg`
  coreinfo=`ls -l $core`

  gdb -batch \
      -ex "set logging file $gdblogfile" \
      -ex "set logging on" \
      -ex "set pagination off" \
      -ex "printf \"**\n** Process info for $binimg - $core \n**
Generated `date`\n\"" \
      -ex "printf \"**\n** $bininfo \n** $coreinfo\n**\n\"" \
      -ex "file $binimg" \
      -ex "core-file $core" \
      -ex "bt" \
      -ex "info proc" \
      -ex "printf \"*\n* Libraries \n*\n\"" \
      -ex "info sharedlib" \
      -ex "printf \"*\n* Memory map \n*\n\"" \
      -ex "info target" \
```

```
      -ex "printf \"*\n* Registers \n*\n\"" \
      -ex "info registers" \
      -ex "printf \"*\n* Current instructions \n*\n\"" -ex "x/16i \$pc"
 \
      -ex "printf \"*\n* Threads (full) \n*\n\"" \
      -ex "info threads" \
      -ex "bt" \
      -ex "thread apply all bt full" \
      -ex "printf \"*\n* Threads (basic) \n*\n\"" \
      -ex "info threads" \
      -ex "thread apply all bt" \
      -ex "printf \"*\n* Done \n*\n\"" \
      -ex "quit"
done
```

An alternative worth exploring is btparser [2].

Canned user-defined commands

Same reporting functionality can be canned for gdb:

```
define procinfo
    printf "**\n** Process Info: \n**\n"
    info proc

    printf "*\n* Libraries \n*\n"
    info sharedlib

    printf "*\n* Memory Map \n*\n"
    info target

    printf "*\n* Registers \n*\n"
    info registers

    printf "*\n* Current Instructions \n*\n"
    x/16i $pc

    printf "*\n* Threads (basic) \n*\n"
    info threads
    thread apply all bt
end
document procinfo
Infos about the debugee.
end

define analyze
    procinfo

    printf "*\n* Threads (full) \n*\n"
```

```
    info threads
    bt
    thread apply all bt full
end
```

analyze-pid

A script that will generate a basic report and a core file for a running process:

```
#!/bin/bash

#
# A script to generate a core and a status report for a running
process.
#


if [ $# -ne 1 ]
then
  echo "Usage: `basename $0` <PID>"
  exit -1
else
  pid=$1
fi


gdblogfile="analyze-$pid.log"
rm $gdblogfile

corefile="core-$pid.core"

gdb -batch \
      -ex "set logging file $gdblogfile" \
      -ex "set logging on" \
      -ex "set pagination off" \
      -ex "printf \"**\n** Process info for PID=$pid \n** Generated
`date`\n\"" \
      -ex "printf \"**\n** Core: $corefile \n**\n\"" \
      -ex "attach $pid" \
      -ex "bt" \
      -ex "info proc" \
      -ex "printf \"*\n* Libraries \n*\n\"" \
      -ex "info sharedlib" \
      -ex "printf \"*\n* Memory map \n*\n\"" \
      -ex "info target" \
      -ex "printf \"*\n* Registers \n*\n\"" \
      -ex "info registers" \
      -ex "printf \"*\n* Current instructions \n*\n\"" -ex "x/16i \$pc"
 \
```

```
      -ex "printf \"*\n* Threads (full) \n*\n\"" \
      -ex "info threads" \
      -ex "bt" \
      -ex "thread apply all bt full" \
      -ex "printf \"*\n* Threads (basic) \n*\n\"" \
      -ex "info threads" \
      -ex "thread apply all bt" \
      -ex "printf \"*\n* Done \n*\n\"" \
      -ex "generate-core-file $corefile" \
      -ex "detach" \
      -ex "quit"
```

Thread Local Storage

TLS data is rather difficult to access with gdb in the core files, and `__tls_get_addr()` cannot be called.

References

- __thread variables [3]

References

[1] http://code.google.com/p/google-coredumper/
[2] https://fedorahosted.org/btparser/
[3] http://www.technovelty.org/linux/thread-variable-debug.html

Core analysis

Basics

To obtain a good call stack, it is important that the gdb loads the same libraries that were loaded by the program that generated the core dump. If the machine we are analyzing the core has different libraries (or has them in different places) from the machine the core was dumped, then copy over the libraries to the analyzing machine, in a way that mirrors the dump machine. For instance:

```
$ tree .
.
|-- juggler-29964.core
|-- lib64
|   |-- ld-linux-x86-64.so.2
|   |-- libc.so.6
|   |-- libm.so.6
|   |-- libpthread.so.0
|   `-- librt.so.1
...
```

At the gdb prompt:

```
(gdb) set solib-absolute-prefix ./

(gdb) set solib-search-path .

(gdb) file
```

```
../../../../../threadpool/bin.v2/libs/threadpool/example/juggler/gcc-4.1.2/debug/link-static/threading-multi/juggler

Reading symbols from

/home/aurelian_melinte/threadpool/threadpool-0_2_5-src/threadpool/bin.v2/libs/threadpool/example/juggler/gcc-4.1.2/debug/link-static/threading-multi/juggler...done.

(gdb) core-file juggler-29964.core

Reading symbols from ./lib64/librt.so.1...(no debugging symbols

found)...done.

Loaded symbols for ./lib64/librt.so.1

Reading symbols from ./lib64/libm.so.6...(no debugging symbols

found)...done.

Loaded symbols for ./lib64/libm.so.6

Reading symbols from ./lib64/libpthread.so.0...(no debugging symbols

found)...done.

Loaded symbols for ./lib64/libpthread.so.0

Reading symbols from ./lib64/libc.so.6...(no debugging symbols

found)...done.

Loaded symbols for ./lib64/libc.so.6

Reading symbols from ./lib64/ld-linux-x86-64.so.2...(no debugging

symbols found)...done.

Loaded symbols for ./lib64/ld-linux-x86-64.so.2

Core was generated by

`../../../../bin.v2/libs/threadpool/example/juggler/gcc-4.1.2/debug/link-static/'.

Program terminated with signal 6, Aborted.

#0  0x0000003684030265 in raise () from ./lib64/libc.so.6

(gdb) frame 2

#2  0x00000000000404ae1 in dump_core_and_terminate () at juggler.cpp:30
```

analyze-cores

Here is a script that will generate a basic report per core file. Useful the days when cores are raining on you:

```
#!/bin/bash

#
# A script to extract core-file informations
#


if [ $# -ne 1 ]
then
  echo "Usage: `basename $0` <for-binary-image>"
  exit -1
else
  binimg=$1
fi


# Today and yesterdays cores
cores=`find . -name '*.core' -mtime -1`
```

```
#cores=`find . -name '*.core'`


for core in $cores
do
  gdblogfile="$core-gdb.log"
  rm $gdblogfile

  bininfo=`ls -l $binimg`
  coreinfo=`ls -l $core`

  gdb -batch \
      -ex "set logging file $gdblogfile" \
      -ex "set logging on" \
      -ex "set pagination off" \
      -ex "printf \"**\n** Process info for $binimg - $core \n**
Generated `date`\n\"" \
      -ex "printf \"**\n** $bininfo \n** $coreinfo\n**\n\"" \
      -ex "file $binimg" \
      -ex "core-file $core" \
      -ex "bt" \
      -ex "info proc" \
      -ex "printf \"*\n* Libraries \n*\n\"" \
      -ex "info sharedlib" \
      -ex "printf \"*\n* Memory map \n*\n\"" \
      -ex "info target" \
      -ex "printf \"*\n* Registers \n*\n\"" \
      -ex "info registers" \
      -ex "printf \"*\n* Current instructions \n*\n\"" -ex "x/16i \$pc"
 \
      -ex "printf \"*\n* Threads (full) \n*\n\"" \
      -ex "info threads" \
      -ex "bt" \
      -ex "thread apply all bt full" \
      -ex "printf \"*\n* Threads (basic) \n*\n\"" \
      -ex "info threads" \
      -ex "thread apply all bt" \
      -ex "printf \"*\n* Done \n*\n\"" \
      -ex "quit"
done
```

Canned user-defined commands

Same reporting functionality can be canned for gdb:

```
define procinfo
    printf "**\n** Process Info: \n**\n"
    info proc

    printf "*\n* Libraries \n*\n"
    info sharedlib

    printf "*\n* Memory Map \n*\n"
    info target

    printf "*\n* Registers \n*\n"
    info registers

    printf "*\n* Current Instructions \n*\n"
    x/16i $pc

    printf "*\n* Threads (basic) \n*\n"
    info threads
    thread apply all bt
end
document procinfo
Infos about the debugee.
end

define analyze
    procinfo

    printf "*\n* Threads (full) \n*\n"
    info threads
    bt
    thread apply all bt full
end
```

analyze-pid

A script that will generate a basic report and a core file for a running process:

```
#!/bin/bash

#
# A script to generate a core and a status report for a running
process.
#


if [ $# -ne 1 ]
```

```
then
  echo "Usage: `basename $0` <PID>"
  exit -1
else
  pid=$1
fi


gdblogfile="analyze-$pid.log"
rm $gdblogfile

corefile="core-$pid.core"

gdb -batch \
      -ex "set logging file $gdblogfile" \
      -ex "set logging on" \
      -ex "set pagination off" \
      -ex "printf \"**\n** Process info for PID=$pid \n** Generated
`date`\n\"" \
      -ex "printf \"**\n** Core: $corefile \n**\n\"" \
      -ex "attach $pid" \
      -ex "bt" \
      -ex "info proc" \
      -ex "printf \"*\n* Libraries \n*\n\"" \
      -ex "info sharedlib" \
      -ex "printf \"*\n* Memory map \n*\n\"" \
      -ex "info target" \
      -ex "printf \"*\n* Registers \n*\n\"" \
      -ex "info registers" \
      -ex "printf \"*\n* Current instructions \n*\n\"" -ex "x/16i \$pc"
 \
      -ex "printf \"*\n* Threads (full) \n*\n\"" \
      -ex "info threads" \
      -ex "bt" \
      -ex "thread apply all bt full" \
      -ex "printf \"*\n* Threads (basic) \n*\n\"" \
      -ex "info threads" \
      -ex "thread apply all bt" \
      -ex "printf \"*\n* Done \n*\n\"" \
      -ex "generate-core-file $corefile" \
      -ex "detach" \
      -ex "quit"
```

Thread Local Storage

TLS data is rather difficult to access with gdb in the core files.

References

- __thread variables [3]

The call stack

The API

Sometimes we need the call stack at a certain point in the program. These are the API functions to get basic stack information:

```
#include <execinfo.h>

int backtrace(void **buffer, int size);
char **backtrace_symbols(void *const *buffer, int size);
void backtrace_symbols_fd(void *const *buffer, int size, int fd);

#include <cxxabi.h>
char* __cxa_demangle(const char* __mangled_name, char* __output_buffer,
 size_t* __length, int* __status);

#include <dlfcn.h>
int dladdr(void *addr, Dl_info *info);
```

Notes:

- C++ symbols are still mangled. Use abi::__cxa_demangle() [1] or something similar.
- Some of the these functions do allocate memory - either temporarily either explicitly - and this might be a problem if the program is unstable already.
- Some of the these functions do acquire locks (e.g. `dladdr()`).
- `backtrace*` are expensive calls on some platforms (x86_64 for instance), the deeper the call stack is the more expensive the call is.
- `dladdr` will fail to rsolve any symbol that is not exported. Hence, to get the most out of it:
  - Compile with `-rdynamic`
  - Link with `-ldl`

To extract more information (file, line number), use libbfd:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

#include <execinfo.h>
#include <signal.h>
#include <bfd.h>
#include <unistd.h>
```

```
/* globals retained across calls to resolve. */
static bfd* abfd = 0;
static asymbol **syms = 0;
static asection *text = 0;

static void resolve(char *address) {
    if (!abfd) {
         char ename[1024];
         int l = readlink("/proc/self/exe",ename,sizeof(ename));
         if (l == -1) {
           perror("failed to find executable\n");
           return;
         }
         ename[l] = 0;

         bfd_init();

         abfd = bfd_openr(ename, 0);
         if (!abfd) {
             perror("bfd_openr failed: ");
             return;
         }

         /* oddly, this is required for it to work... */
         bfd_check_format(abfd,bfd_object);

         unsigned storage_needed = bfd_get_symtab_upper_bound(abfd);
         syms = (asymbol **) malloc(storage_needed);
         unsigned cSymbols = bfd_canonicalize_symtab(abfd, syms);

         text = bfd_get_section_by_name(abfd, ".text");
    }

    long offset = ((long)address) - text->vma;
    if (offset > 0) {
        const char *file;
        const char *func;
        unsigned line;
        if (bfd_find_nearest_line(abfd, text, syms, offset, &file,
&func, &line) && file)
            printf("file: %s, line: %u, func %s\n",file,line,func);
    }
}
```

A C++ wrapper

With this simple class, getting the stack is one line away:

```
class call_stack
{
public:

    static const int depth = 40;
    typedef std::array<void *, depth> stack_t;

    class const_iterator;
    class frame
    {
    public:

        frame(void *addr = 0)
                : _addr(0)
                , _dladdr_ret(false)
                , _binary_name(0)
                , _func_name(0)
                , _demangled_func_name(0)
                , _delta_sign('+')
                , _delta(0L)
                , _source_file_name(0)
                , _line_number(0)
        {
            resolve(addr);
        }

        // frame(stack_t::iterator& it) : frame(*it) {} //C++0x
        frame(stack_t::const_iterator const& it)
                : _addr(0)
                , _dladdr_ret(false)
                , _binary_name(0)
                , _func_name(0)
                , _demangled_func_name(0)
                , _delta_sign('+')
                , _delta(0L)
                , _source_file_name(0)
                , _line_number(0)
        {
            resolve(*it);
        }

        frame(frame const& other)
        {
            resolve(other._addr);
        }
```

```
        frame& operator=(frame const& other)
        {
            if (this != &other) {
                resolve(other._addr);
            }
            return *this;
        }

        ~frame()
        {
            resolve(0);
        }

        std::string as_string() const
        {
            std::ostringstream s;
            s << "[" << std::hex << _addr << "] "
              << demangled_function()
              << " (" << binary_file() << _delta_sign << "0x" << std::hex << _delta << ")"
              << " in " << source_file() << ":" << line_number()
              ;
            return s.str();
        }

        const void* addr() const               { return _addr; }
        const char* binary_file() const        { return
safe(_binary_name); }
        const char* function() const           { return
safe(_func_name); }
        const char* demangled_function() const { return
safe(_demangled_func_name); }
        char        delta_sign() const         { return _delta_sign; }
        long        delta() const              { return _delta; }
        const char* source_file() const        { return
safe(_source_file_name); }
        int         line_number() const        { return  _line_number;
}

    private:

        const char* safe(const char* p) const { return p ? p : "??"; }

        friend class const_iterator; // To call resolve()
        void resolve(const void * addr)
        {
            if (_addr == addr)
```

```
                return;

            _addr = addr;
            _dladdr_ret = false;
            _binary_name = 0;
            _func_name = 0;
            if (_demangled_func_name) {
                free(_demangled_func_name);
                _demangled_func_name = 0;
            }
            _delta_sign = '+';
            _delta = 0L;
            _source_file_name = 0;
            _line_number = 0;

            if (!_addr)
                return;

            _dladdr_ret = (::dladdr(_addr, &_info) != 0);
            if (_dladdr_ret)
            {
                _binary_name = safe(_info.dli_fname);
                _func_name   = safe(_info.dli_sname);
                _delta_sign  = (_addr >=  _info.dli_saddr) ? '+' : '-';
                _delta = ::labs(static_cast<const char *>(_addr) -
static_cast<const char *>(_info.dli_saddr));

                 int status = 0;
                 _demangled_func_name = abi::__cxa_demangle(_func_name,
 0, 0, &status);
            }
        }

    private:

        const void* _addr;
        const char* _binary_name;
        const char* _func_name;
        const char* _demangled_func_name;
        char        _delta_sign;
        long        _delta;
        const char* _source_file_name; //TODO: libbfd
        int         _line_number;

        Dl_info     _info;
        bool        _dladdr_ret;
    }; //frame
```

```
class const_iterator
        : public std::iterator< std::bidirectional_iterator_tag
                              , ptrdiff_t
                              >
{
public:

    const_iterator(stack_t::const_iterator const& it)
            : _it(it)
            , _frame(it)
    {}

    bool operator==(const const_iterator& other) const
    {
        return _frame.addr() == other._frame.addr();
    }

    bool operator!=(const const_iterator& x) const
    {
        return !(*this == x);
    }

    const frame& operator*() const
    {
        return _frame;
    }
    const frame* operator->() const
    {
        return &_frame;
    }

    const_iterator& operator++()
    {
        ++_it;
        _frame.resolve(*_it);
        return *this;
    }
    const_iterator operator++(int)
    {
        const_iterator tmp = *this;
        ++_it;
        _frame.resolve(*_it);
        return tmp;
    }
```

```
        const_iterator& operator--()
        {
            --_it;
            _frame.resolve(*_it);
            return *this;
        }
        const_iterator operator--(int)
        {
            const_iterator tmp = *this;
            --_it;
            _frame.resolve(*_it);
            return tmp;
        }

    private:

        const_iterator();

    private:

        frame                    _frame;
        stack_t::const_iterator  _it;
    }; //const_iterator


    call_stack() : _num_frames(0)
    {
        _num_frames = ::backtrace(_stack.data(), depth);
        assert(_num_frames >= 0 && _num_frames <= depth);
    }

    std::string as_string()
    {
        std::string s;
        const_iterator itEnd = end();
        for (const_iterator it = begin(); it != itEnd; ++it) {
            s += it->as_string();
            s += "\n";
        }
        return std::move(s);
    }

    virtual ~call_stack()
    {
    }

    const_iterator begin() const { return _stack.cbegin(); }
```

```
    const_iterator end() const   { return
stack_t::const_iterator(&_stack[_num_frames]); }

private:

    stack_t  _stack;
    int      _num_frames;
};
```

This class can be used with assertions, logging or exceptions.

The full code can be found at the LPT site and it offers three symbol resolver varieties. This code will require a C++11 compiler.

- A "basic" address-only resolver, without memory side-effects
- A libc resolver
- A libbfd resolver

A boost-based [2] version of the call stack utilities collection can be found at https:// github. com/ melintea/ Boost-Call_stack.

A stack resolved with the libc resolver, embedded in an exception, with debug-compiled code would look like:

```
Exception PAPI_add_event: Event exists, but cannot be counted due to
hardware resource limits
At:
0x41e11f ??+0x41e11f
        At ??:0
        In binaries/bin/papitest
0x421ae0 lpt::stack::call_stack&lt;40ul&gt;::call_stack(bool)+0x52
        At ??:0
        In binaries/bin/papitest
0x420653 lpt::papi::papi_error::papi_error(std::string const&,
int)+0x6f
        At ??:0
        In binaries/bin/papitest
0x426ffe lpt::papi::library&lt;lpt::papi::counting_per_thread,
lpt::papi::multiplex_none&gt;::init()+0x722
        At ??:0
        In binaries/bin/papitest
0x426815 lpt::papi::library&lt;lpt::papi::counting_per_thread,
lpt::papi::multiplex_none&gt;::library()+0x5b
        At ??:0
        In binaries/bin/papitest
0x4265d4
lpt::singleton&lt;lpt::papi::library&lt;lpt::papi::counting_per_thread,
 lpt::papi::multiplex_none&gt; &gt;::singleton()+0x18
        At ??:0
        In binaries/bin/papitest
0x42624e
lpt::singleton&lt;lpt::papi::library&lt;lpt::papi::counting_per_thread,
```

```
 lpt::papi::multiplex_none&gt; &gt;::instance()+0x42
        At ??:0
        In binaries/bin/papitest
0x425b47 lpt::papi::thread&lt;lpt::papi::counting_per_thread,
lpt::papi::multiplex_none&gt;::init()+0x25
        At ??:0
        In binaries/bin/papitest
0x425502 lpt::papi::thread&lt;lpt::papi::counting_per_thread,
lpt::papi::multiplex_none&gt;::thread()+0x2e
        At ??:0
        In binaries/bin/papitest
0x424554
lpt::singleton&lt;lpt::papi::thread&lt;lpt::papi::counting_per_thread,
lpt::papi::multiplex_none&gt; &gt;::singleton()+0x18
        At ??:0
        In binaries/bin/papitest
0x4230a5
lpt::singleton&lt;lpt::papi::thread&lt;lpt::papi::counting_per_thread,
lpt::papi::multiplex_none&gt; &gt;::instance()+0x42
        At ??:0
        In binaries/bin/papitest
0x421d33 lpt::papi::counters&lt;lpt::papi::stdout_print,
lpt::papi::counting_per_thread,
lpt::papi::multiplex_none&gt;::counters(std::string const&)+0xc9
        At ??:0
        In binaries/bin/papitest
0x41e206 tests()+0x67
        At ??:0
        In binaries/bin/papitest
0x41edb8 ??+0x11
        At ??:0
        In binaries/bin/papitest
0x2ad66a7b9ead ??+0xfd
        At ??:0
        In /lib/x86_64-linux-gnu/libc.so.6
0x41dea9 ??+0x41dea9
        At ??:0
        In binaries/bin/papitest
```

Note how `dladdr()` fails to resolve some of the frames. Whereas a stack resolved with the bfd resolver would show file and line numbers and would resolve all frames:

```
Exception PAPI_add_event: Event exists, but cannot be counted due to
hardware resource limits
At:
[0x41e20f] backtrace+0x41e20f
        At
/home/amelinte/projects/lpt/lpt/lpt/include/lpt/detail/glibc_call_stack.hpp:61
```

```
        In binaries/bin/papitest
[0x421c4e] lpt::stack::call_stack&lt;40ul&gt;::call_stack(bool)+0x52
        At
/home/amelinte/projects/lpt/lpt/lpt/include/lpt/call_stack.hpp:74
        In binaries/bin/papitest
[0x4207a1] lpt::papi::papi_error::papi_error(std::string const&,
int)+0x6f
        At /home/amelinte/projects/lpt/lpt/lpt/include/lpt/papi.hpp:112
        In binaries/bin/papitest
[0x42716c] lpt::papi::library&lt;lpt::papi::counting_per_thread,
lpt::papi::multiplex_none&gt;::init()+0x722
        At /home/amelinte/projects/lpt/lpt/lpt/include/lpt/papi.hpp:371
        In binaries/bin/papitest
[0x426983] lpt::papi::library&lt;lpt::papi::counting_per_thread,
lpt::papi::multiplex_none&gt;::library()+0x5b
        At /home/amelinte/projects/lpt/lpt/lpt/include/lpt/papi.hpp:289
        In binaries/bin/papitest
[0x426742]
lpt::singleton&lt;lpt::papi::library&lt;lpt::papi::counting_per_thread,
 lpt::papi::multiplex_none&gt; &gt;::singleton()+0x18
        At
/home/amelinte/projects/lpt/lpt/lpt/include/lpt/singleton.hpp:19
        In binaries/bin/papitest
[0x4263bc]
lpt::singleton&lt;lpt::papi::library&lt;lpt::papi::counting_per_thread,
 lpt::papi::multiplex_none&gt; &gt;::instance()+0x42
        At
/home/amelinte/projects/lpt/lpt/lpt/include/lpt/singleton.hpp:28
        In binaries/bin/papitest
[0x425cb5] lpt::papi::thread&lt;lpt::papi::counting_per_thread,
lpt::papi::multiplex_none&gt;::init()+0x25
        At /home/amelinte/projects/lpt/lpt/lpt/include/lpt/papi.hpp:433
        In binaries/bin/papitest
[0x425670] lpt::papi::thread&lt;lpt::papi::counting_per_thread,
lpt::papi::multiplex_none&gt;::thread()+0x2e
        At /home/amelinte/projects/lpt/lpt/lpt/include/lpt/papi.hpp:409
        In binaries/bin/papitest
[0x4246c2]
lpt::singleton&lt;lpt::papi::thread&lt;lpt::papi::counting_per_thread,
lpt::papi::multiplex_none&gt; &gt;::singleton()+0x18
        At
/home/amelinte/projects/lpt/lpt/lpt/include/lpt/singleton.hpp:19
        In binaries/bin/papitest
[0x423213]
lpt::singleton&lt;lpt::papi::thread&lt;lpt::papi::counting_per_thread,
lpt::papi::multiplex_none&gt; &gt;::instance()+0x42
        At
```

```
/home/amelinte/projects/lpt/lpt/lpt/include/lpt/singleton.hpp:28
        In binaries/bin/papitest
[0x421ea1] lpt::papi::counters&lt;lpt::papi::stdout_print,
lpt::papi::counting_per_thread,
lpt::papi::multiplex_none&gt;::counters(std::string const&)+0xc9
        At /home/amelinte/projects/lpt/lpt/lpt/include/lpt/papi.hpp:646
        In binaries/bin/papitest
[0x41e2f6] tests()+0x67
        At /home/amelinte/projects/lpt/lpt/tests/papitest.cpp:51
        In binaries/bin/papitest
[0x41eea8] main+0x11
        At /home/amelinte/projects/lpt/lpt/tests/papitest.cpp:176
        In binaries/bin/papitest
[0x2b5f7aa1bead] __libc_start_main+0xfd
        At ??:0
        In /lib/x86_64-linux-gnu/libc.so.6
[0x41df99] _start+0x41df99
        At ??:0
        In binaries/bin/papitest
```

Caveats

- Do NOT use in asynchronous interrupts such as signal handlers.
- It might or it might not work when the program is unstable - prefer a core dump if possible. Same for memory issues handlers .
- Performance hit varies greatly with platform and compiler version.

From gdb

A canned command to resolve a stack address from within gdb:

```
define addrtosym
    if $argc == 1
        printf "[%u]: ", $arg0
        #whatis/ptype EXPR
        #info frame ADDR
        info symbol $arg0
    end
end
document addrtosym
Resolve the address (e.g. of one stack frame). Usage: addrtosym addr0
end
```

References

[1] http://gcc.gnu.org/onlinedocs/libstdc++/manual/ext_demangling.html
[2] http://www.boost.org

The interposition library

The dynamic liker allows for interception of any function call an application makes to any shared library it uses. As such, interposition is a powerful technique allowing to tune performance, collect runtime statistics, or debug the application without having to instrument the code of that application.

As an example, we can use an interposition library to trace calls, with arguments' values and return codes.

Call tracing

Note that part of code below is 32-bit x86 and gcc 4.1/4.2 specific.

Code intrumentation

In the library, we want to address the following points:

- when a function/method is entered and exited.
- what were the call arguments when the function is entered.
- what was the return code when the function is exited.
- optionally, where was the function called from.

The first one is easy: if requested, the compiler will instrument functions and methods so that when a function/method is entered, a call to an instrumentation function is made and when the function is exited, a similar intrumentation call is made:

```
    void __cyg_profile_func_enter(void *func, void *callsite);
    void __cyg_profile_func_exit(void *func, void *callsite);
```

This is achieved by compiling the code with the `-finstrument-functions` flag. The above two functions can be used for instance to collect data for coverage; or for profiling. We will use them to print a trace of function calls. Furthermore, we can isolate these two functions and the supporting code in an interposition library of our own. This library can be loaded when and if needed, thus leaving the application code basically unchanged.

Now when the function is entered we can get the arguments of the call:

```
    void __cyg_profile_func_enter( void *func, void *callsite )
    {
        char buf_func[CTRACE_BUF_LEN+1] = {0};
        char buf_file[CTRACE_BUF_LEN+1] = {0};
        char buf_args[ARG_BUF_LEN + 1] = {0};
        pthread_t self = (pthread_t)0;
        int *frame = NULL;
        int nargs = 0;

        self = pthread_self();
        frame = (int *)__builtin_frame_address(1); /*of the 'func'*/

        /*Which function*/
        libtrace_resolve (func, buf_func, CTRACE_BUF_LEN, NULL, 0);
```

```
        /*From where.  KO with optimizations. */
        libtrace_resolve (callsite, NULL, 0, buf_file, CTRACE_BUF_LEN);

        nargs = nchr(buf_func, ',') + 1; /*Last arg has no comma
after*/
        nargs += is_cpp(buf_func);       /*'this'*/
        if (nargs > MAX_ARG_SHOW)
            nargs = MAX_ARG_SHOW;

        printf("T%p: %p %s %s [from %s]\n",
               self, (int*)func, buf_func,
               args(buf_args, ARG_BUF_LEN, nargs, frame),
               buf_file);
    }
```

And when the function is is exited, we get the return value:

```
    void __cyg_profile_func_exit( void *func, void *callsite )
    {
        long ret = 0L;
        char buf_func[CTRACE_BUF_LEN+1] = {0};
        char buf_file[CTRACE_BUF_LEN+1] = {0};
        pthread_t self = (pthread_t)0;

        GET_EBX(ret);
        self = pthread_self();

        /*Which function*/
        libtrace_resolve (func, buf_func, CTRACE_BUF_LEN, NULL, 0);

        printf("T%p: %p %s => %d\n",
               self, (int*)func, buf_func,
               ret);

        SET_EBX(ret);
    }
```

Since these two instrumentation functions are aware of addresses and we actually want the trace to be readable by humans, we need also a way to resolve symbol addresses to symbol names: this is what libtrace_resolve() does.

Binutils and libbfd

First, we have to have the symbols information handy. To achieve this, we compile our application with the -g flag. Then, we can map addresses to symbol names and this would normally require writing some code knowledgeable of the ELF format.

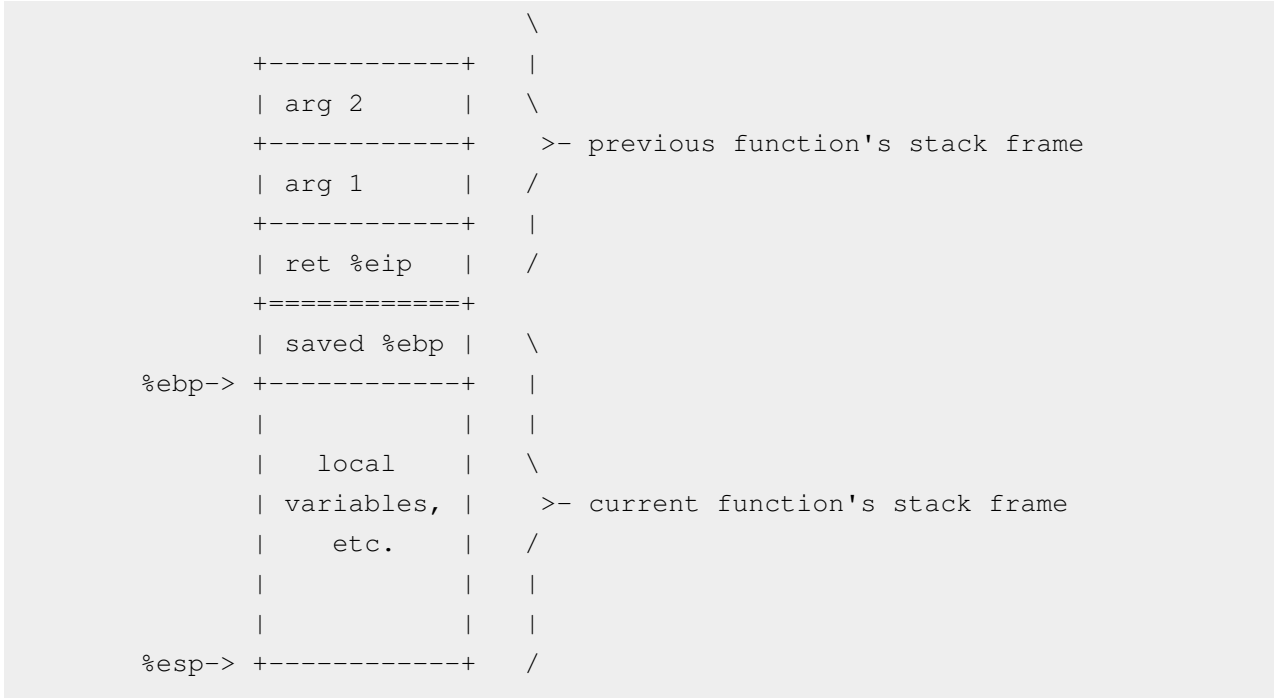
Luckily, the there is the binutils package which comes with a library that does just that: libbfd; and with a tool: addr2line. addr2line is a good example on how to use libbfd and I have simply used it to wrap around libbfd. The result is the libtrace_resolve() function.

Since the instrumentation functions are isolated in a stand-alone module, we tell this module the name of the instrumented executable through an environment variable (CTRACE_PROGRAM) that we set before running the program. This is needed to properly init libbfd to search for symbols.

Stack layout

To address the first point the work has been architecture-agnostic (actually libbfd is aware of the architecture, but things are hidden behind its API). However, to retrieve function arguments and return values we have to look at the stack, write a bit of architecture-specific code and exploit some gcc quirks. Again, the compilers I have used were gcc 4.1 and 4.2; later or previous versions might work differently. In short:

- x86 dictates that stack grows down.
- GCC dictates how the stack is used - a "typical" stack is depicted below.
- each function has a stack frame marked by the ebp (base pointer) and esp (stack pointer) registers.
- normally, we expect the eax register to contain the reurn code



In an ideal world, the code the compiler generates would make sure that upon instrumenting the exit of a function: the return value is set, then CPU registers pushed on the stack (to ensure the instrumentation function does not affects them), then call the instrumentation function and then pop the registers. This sequence of code would ensure we always get access to the

```
return value in the instrumentation function. The code generated by the compiler is a bit different...
```

Also, in practice, many of gcc's flags affect the stack layout and registers usage. The most obvious ones are:

- -fomit-frame-pointer. This flag affects the stack offset where the arguments are to be found.

- The optimization flags: -Ox; each of these flags aggregates a number of optimizations. These flags did not affected the stack, and, quite amazingly, arguments were always passsed to functions through the stack, regardless of the optimization level. One would have expected that some arguments would pe passed through registers - in which case getting these arguments would have proven to be difficult to impossible. However, these flags did complicated recovering the return code. However, on some architectures, these flags will suck in the -fomit-frame-pointer optimization.
- In any case, be wary: other flags you use to compile your application may reserve surprises.

Function arguments

In my tests with the compilers, all arguments were invariably passed through the stack. Hence this is trivial business, affected to a small extent by the -fomit-frame-pointer flag - this flag will change the offset at which arguments start.

How many arguments a function has, how many arguments are on the stack? One way to infer somehow the number of arguments is based on its signature (for C++, beware of the 'this' hidden argument) and this is the technique used in __cyg_profile_func_enter().

Once we know the offset where arguments start on the stack and how many of them there are, we just walk the stack to retrieve their values:

```
    char *args(char *buf, int len, int nargs, int *frame)
    {
        int i;
        int offset;

        memset(buf, 0, len);

        snprintf(buf, len, "(");
        offset = 1;
        for (i=0; i&lt;nargs && offset&lt;len; i++) {
            offset += snprintf(buf+offset, len-offset, "%d%s",
                               *(frame+ARG_OFFET+i),
                               i==nargs-1 ? " ...)" : ", ");
        }

        return buf;
    }
```

Function return values

Obtaining the return value proved to be possible only when using the -O0 flag.

Let's look what happens when this method

```
    class B {
        ...
        virtual int m1(int i, int j) {printf("B::m1()\n"); f1(i);
return 20;}
        ...
    };
```

is instrumented with -O0:

```
     080496a2 <_ZN1B2m1Eii>:
     80496a2:    55                      push   %ebp
     80496a3:    89 e5                   mov    %esp,%ebp
     80496a5:    53                      push   %ebx
     80496a6:    83 ec 24                sub    $0x24,%esp
     80496a9:    8b 45 04                mov    0x4(%ebp),%eax
     80496ac:    89 44 24 04             mov    %eax,0x4(%esp)
     80496b0:    c7 04 24 a2 96 04 08    movl   $0x80496a2,(%esp)
     80496b7:    e8 b0 f4 ff ff          call   8048b6c <__cyg_profile_func_enter@plt>
     80496bc:    c7 04 24 35 9c 04 08    movl   $0x8049c35,(%esp)
     80496c3:    e8 b4 f4 ff ff          call   8048b7c <puts@plt>
     80496c8:    8b 45 0c                mov    0xc(%ebp),%eax
     80496cb:    89 04 24                mov    %eax,(%esp)
     80496ce:    e8 9d f8 ff ff          call   8048f70 <_Z2f1i>

==>  80496d3:    bb 14 00 00 00          mov    $0x14,%ebx
     80496d8:    8b 45 04                mov    0x4(%ebp),%eax
     80496db:    89 44 24 04             mov    %eax,0x4(%esp)
     80496df:    c7 04 24 a2 96 04 08    movl   $0x80496a2,(%esp)
==>  80496e6:    e8 81 f5 ff ff          call   8048c6c <__cyg_profile_func_exit@plt>
     80496eb:    89 5d f8                mov    %ebx,0xfffffff8(%ebp)
==>  80496ee:    eb 27                   jmp    8049717 <_ZN1B2m1Eii+0x75>
     80496f0:    89 45 f4                mov    %eax,0xfffffff4(%ebp)
     80496f3:    8b 5d f4                mov    0xfffffff4(%ebp),%ebx
     80496f6:    8b 45 04                mov    0x4(%ebp),%eax
     80496f9:    89 44 24 04             mov    %eax,0x4(%esp)
     80496fd:    c7 04 24 a2 96 04 08    movl   $0x80496a2,(%esp)
     8049704:    e8 63 f5 ff ff          call   8048c6c <__cyg_profile_func_exit@plt>
     8049709:    89 5d f4                mov    %ebx,0xfffffff4(%ebp)
     804970c:    8b 45 f4                mov    0xfffffff4(%ebp),%eax
     804970f:    89 04 24                mov    %eax,(%esp)
     8049712:    e8 15 f5 ff ff          call   8048c2c <_Unwind_Resume@plt>

==>  8049717:    8b 45 f8                mov    0xfffffff8(%ebp),%eax
     804971a:    83 c4 24                add    $0x24,%esp
     804971d:    5b                      pop    %ebx
     804971e:    5d                      pop    %ebp
     804971f:    c3                      ret
```

Note how the return code is moved into the ebx register - a bit unexpected, since, traditionally, the eax register is used for return codes - and then the instrumentation function is called. Good to retrieve the return value but to avoid that the ebx register gets clobbered in the instrumentation function, we save it upon entering the function and we restore it upon exit.

When the compilation is done with some degree of optimization (-O1...3; shown here is -O2), the code changes:

```
    080498c0 <_ZN1B2m1Eii>:
     80498c0:    55                      push   %ebp
     80498c1:    89 e5                   mov    %esp,%ebp
```

```
     80498c3:    53                      push   %ebx
     80498c4:    83 ec 14                sub    $0x14,%esp
     80498c7:    8b 45 04                mov    0x4(%ebp),%eax
     80498ca:    c7 04 24 c0 98 04 08    movl   $0x80498c0,(%esp)
     80498d1:    89 44 24 04             mov    %eax,0x4(%esp)
     80498d5:    e8 12 f4 ff ff          call   8048cec <__cyg_profile_func_enter@plt>
     80498da:    c7 04 24 2d 9c 04 08    movl   $0x8049c2d,(%esp)
     80498e1:    e8 16 f4 ff ff          call   8048cfc <puts@plt>

     80498e6:    8b 45 0c                mov    0xc(%ebp),%eax
     80498e9:    89 04 24                mov    %eax,(%esp)
     80498ec:    e8 af f7 ff ff          call   80490a0 <_Z2f1i>
     80498f1:    8b 45 04                mov    0x4(%ebp),%eax
     80498f4:    c7 04 24 c0 98 04 08    movl   $0x80498c0,(%esp)
     80498fb:    89 44 24 04             mov    %eax,0x4(%esp)
==>  80498ff:    e8 88 f3 ff ff          call   8048c8c <__cyg_profile_func_exit@plt>
     8049904:    83 c4 14                add    $0x14,%esp
==>  8049907:    b8 14 00 00 00          mov    $0x14,%eax
     804990c:    5b                      pop    %ebx
     804990d:    5d                      pop    %ebp
==>  804990e:    c3                      ret

     804990f:    89 c3                   mov    %eax,%ebx
     8049911:    8b 45 04                mov    0x4(%ebp),%eax
     8049914:    c7 04 24 c0 98 04 08    movl   $0x80498c0,(%esp)
     804991b:    89 44 24 04             mov    %eax,0x4(%esp)
     804991f:    e8 68 f3 ff ff          call   8048c8c <__cyg_profile_func_exit@plt>
     8049924:    89 1c 24                mov    %ebx,(%esp)
     8049927:    e8 f0 f3 ff ff          call   8048d1c <_Unwind_Resume@plt>
     804992c:    90                      nop
     804992d:    90                      nop
     804992e:    90                      nop
     804992f:    90                      nop
```

Note how the instrumentation function gets called first and only then the eax register is set with the return value. Thus, if we absolutely want the return code, we are forced to compile with -O0.

Sample output

Finally, below are the results. At at shell prompt type:

```
$ export CTRACE_PROGRAM=./cpptraced
$ LD_PRELOAD=./libctrace.so ./cpptraced
```

```
T0xb7c0f6c0: 0x8048d34 main (0 ...) [from ]
./cpptraced: main(argc=1)
T0xb7c0ebb0: 0x80492d8 thread1(void*) (1 ...) [from ]
T0xb7c0ebb0: 0x80498b2 D (134605416 ...) [from cpptraced.cpp:91]
T0xb7c0ebb0: 0x8049630 B (134605416 ...) [from cpptraced.cpp:66]
B::B()
```

```
T0xb7c0ebb0: 0x8049630 B => -1209622540 [from ]
D::D(int=-1210829552)
T0xb7c0ebb0: 0x80498b2 D => -1209622540 [from ]
Hello World! It's me, thread #1!
./cpptraced: done.
T0xb7c0f6c0: 0x8048d34 main => -1212090144 [from ]
T0xb740dbb0: 0x8049000 thread2(void*) (2 ...) [from ]
T0xb740dbb0: 0x80498b2 D (134605432 ...) [from cpptraced.cpp:137]
T0xb740dbb0: 0x8049630 B (134605432 ...) [from cpptraced.cpp:66]
B::B()
T0xb740dbb0: 0x8049630 B => -1209622540 [from ]
D::D(int=-1210829568)
T0xb740dbb0: 0x80498b2 D => -1209622540 [from ]
Hello World! It's me, thread #2!
T#2!
T0xb6c0cbb0: 0x8049166 thread3(void*) (3 ...) [from ]
T0xb6c0cbb0: 0x80498b2 D (134613288 ...) [from cpptraced.cpp:157]
T0xb6c0cbb0: 0x8049630 B (134613288 ...) [from cpptraced.cpp:66]
B::B()
T0xb6c0cbb0: 0x8049630 B => -1209622540 [from ]
D::D(int=0)
T0xb6c0cbb0: 0x80498b2 D => -1209622540 [from ]
Hello World! It's me, thread #3!
T#1!
T0xb7c0ebb0: 0x80490dc wrap_strerror_r (134525680 ...) [from cpptraced.cpp:105]
T0xb7c0ebb0: 0x80490dc wrap_strerror_r => -1210887643 [from ]
T#1+M2 (Success)
T0xb740dbb0: 0x80495a0 D::m1(int, int) (134605432, 3, 4 ...) [from cpptraced.cpp:141]
D::m1()
T0xb740dbb0: 0x8049522 B::m2(int) (134605432, 14 ...) [from cpptraced.cpp:69]
B::m2()
T0xb740dbb0: 0x8048f70 f1 (14 ...) [from cpptraced.cpp:55]
f1 14
T0xb740dbb0: 0x8048ee0 f2(int) (74 ...) [from cpptraced.cpp:44]
f2 74
T0xb740dbb0: 0x8048e5e f3 (144 ...) [from cpptraced.cpp:36]
f3 144
T0xb740dbb0: 0x8048e5e f3 => 80 [from ]
T0xb740dbb0: 0x8048ee0 f2(int) => 70 [from ]
T0xb740dbb0: 0x8048f70 f1 => 60 [from ]
T0xb740dbb0: 0x8049522 B::m2(int) => 21 [from ]
T0xb740dbb0: 0x80495a0 D::m1(int, int) => 30 [from ]
T#2!
T#3!
```

Note how libbfd fails to resolve some addresses when the function gets inlined.

Resources

- Code [1]
- Overview of GCC on x86 platforms [2]
- The Intel stack [3]
- etrace: a tool to generate the run-time function call tree with gcc [4]
- ELF details [5]

References

[1] http://freeshell.de/~amelinte/software.html
[2] http://pdos.csail.mit.edu/6.828/2004/lec/l2.html
[3] http://dsrg.mff.cuni.cz/~ceres/sch/osy/text/ch03s02s02.php
[4] http://ndevilla.free.fr/etrace/
[5] http://www.acsu.buffalo.edu/~charngda/elf.html

Leaks

What to look for

Memory can be allocated through many API calls:

1. `malloc()`
2. `memalign()`
3. `realloc()`
4. `mmap()`
5. `brk()` / `sbrk()`

To return memory to the OS:

1. `free()`
2. `munmap()`

Valgrind

Valgrind should be the first stop for any memory related issue. However:

1. it slows down the program by at least one order of magnitude. In particular server C++ programs can be slowed down 15-20 times.
2. from experience, some versions might have difficulties tracking `mmap()` allocated memory.
3. on amd64, the vex dissasembler is likely to fail (v3.7) rather sooner than later so valgrind is of no use for any medium or intensive usage.
4. you need to write suppressions to filter down the issues reported.

If these are real drawbacks, lighter solutions are available.

```
LD_LIBRARY_PATH=/path/to/valgrind/libs:$LD_LIBRARY_PATH
/path/to/valgrind
    -v \
    --error-limit=no \
    --num-callers=40 \
    --fullpath-after= \
    --track-origins=yes \
    --log-file=/path/to/valgrind.log \
```

```
    --leak-check=full \
    --show-reachable=yes \
    --vex-iropt-precise-memory-exns=yes \
    /path/to/program program-args
```

mudflap

- http://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging [1]

DIY:libmtrace

The GNU C library comes with a built-in functionality to help detecting memory issues: `mtrace()`. One of the shortcomings: it does not log the call stacks of the memory allocations it tracks. We can build an interposition library to augment `mtrace()`.

The basics

The malloc implementation in the GNU C library provides a simple but powerful way to detect memory leaks and obtain some information to find the location where the leaks occurs, and this, with rather minimal speed penalties for the program.

Getting started is as simple as it can be:

- `#include mcheck.h` in your code.
- Call `mtrace()` to install hooks for `malloc()`, `realloc()`, `free()` and `memalign()`. From this point on, all memory manipulations by these functions will be tracked. Note there are other untracked ways to allocate memory.
- Call `muntrace()` to uninstall the tracking handlers.
- Recompile.

```
      #include <mcheck.h>
...
21    mtrace();
...
25    std::string* pstr = new std::string("leak");
...
27    char *leak = (char*)malloc(1024);
...
32    muntrace();
...
```

Under the hood, `mtrace()` installs the four hooks mentioned above. The information collected through the hooks is written to a log file.

Note: there are other ways to allocate memory, notably `mmap()`. These allocations will not be reported, unfortunately.

Next:

- Set the `MALLOC_TRACE` environment variable with the memory log name.
- Run the program.
- Run the memory log through mtrace.

```
$ MALLOC_TRACE=logs/mtrace.plain.log  ./dleaker
$ mtrace  dleaker  logs/mtrace.plain.log  >  logs/mtrace.plain.leaks.log
```

```
$ cat logs/mtrace.plain.leaks.log

Memory not freed:
-----------------
   Address     Size     Caller
0x081e2390      0x4  at 0x400fa727
0x081e23a0     0x11  at 0x400fa727
0x081e23b8    0x400  at
/home/amelinte/projects/articole/memtrace/memtrace.v3/main.cpp:27
```

One of the leaks (the `malloc()` call) was precisely traced to the exact file and line number. However, the other leaks at line 25, while detected, we do not know where they occur. The two memory allocations for the `std::string` are buried deep inside the C++ library. We would need the stack trace for these two leaks to pinpoint the place in **our** code.

We can use GDB (or the trace_call macro) to get the allocations' stacks:

```
$ gdb ./dleaker
...
(gdb) set env MALLOC_TRACE=./logs/gdb.mtrace.log

(gdb) b __libc_malloc
Make breakpoint pending on future shared library load? (y or [n])
Breakpoint 1 (__libc_malloc) pending.

(gdb) run
Starting program:
/home/amelinte/projects/articole/memtrace/memtrace.v3/dleaker
Breakpoint 2 at 0xb7cf28d6
Pending breakpoint "__libc_malloc" resolved

Breakpoint 2, 0xb7cf28d6 in malloc () from /lib/i686/cmov/libc.so.6
(gdb) command
Type commands for when breakpoint 2 is hit, one per line.
End with a line saying just "end".
>bt
>cont
>end
(gdb) c
Continuing.

...

Breakpoint 2, 0xb7cf28d6 in malloc () from /lib/i686/cmov/libc.so.6
#0  0xb7cf28d6 in malloc () from /lib/i686/cmov/libc.so.6
#1  0xb7ebb727 in operator new () from /usr/lib/libstdc++.so.6
#2  0x08048a14 in main () at main.cpp:25             <== new std::string("leak");
...
Breakpoint 2, 0xb7cf28d6 in malloc () from /lib/i686/cmov/libc.so.6
```

```
#0  0xb7cf28d6 in malloc () from /lib/i686/cmov/libc.so.6
#1  0xb7ebb727 in operator new () from /usr/lib/libstdc++.so.6   <== mangled: _Znwj
#2  0xb7e95c01 in std::string::_Rep::_S_create () from
/usr/lib/libstdc++.so.6
#3  0xb7e96f05 in ?? () from /usr/lib/libstdc++.so.6
#4  0xb7e970b7 in std::basic_string<char, std::char_traits<char>, std::allocator<char> >::basic_string ()
from /usr/lib/libstdc++.so.6
#5  0x08048a58 in main () at main.cpp:25          <== new std::string("leak");
...

Breakpoint 2, 0xb7cf28d6 in malloc () from /lib/i686/cmov/libc.so.6
#0  0xb7cf28d6 in malloc () from /lib/i686/cmov/libc.so.6

#1  0x08048a75 in main () at main.cpp:27           <== malloc(leak);
```

A couple of improvements

It would be good to have `mtrace()` itself dump the allocation stack and dispense with gdb. The modified `mtrace()` would have to supplement the information with:

- The stack trace for each allocation.
- Demangled function names.
- File name and line number.

Additionally, we can put the code in a library, to free the program from being instrumented with `mtrace()`. In this case, all we have to do is interpose the library when we want to trace memory allocations (and pay the performance price).

Note: getting all this information at runtime, particularly in a human-readable form will have a performance impact on the program, unlike the plain vanilla `mtrace()` supplied with glibc.

The stack trace

A good start would be to use another API function: `backtrace_symbols_fd()`. This would print the stack directly to the log file. Perfect for a C program but C++ symbols are mangled:

```
@ /usr/lib/libstdc++.so.6:(_Znwj+27)[0xb7f1f727] + 0x9d3f3b0 0x4
**[ Stack: 8
./a.out(__gxx_personality_v0+0x304)[0x80492c8]
./a.out[0x80496c1]
./a.out[0x8049a0f]
/lib/i686/cmov/libc.so.6(__libc_malloc+0x35)[0xb7d56905]
/usr/lib/libstdc++.so.6(_Znwj+0x27)[0xb7f1f727]           <=== here
./a.out(main+0x64)[0x8049b50]
/lib/i686/cmov/libc.so.6(__libc_start_main+0xe0)[0xb7cff450]
./a.out(__gxx_personality_v0+0x6d)[0x8049031]
**] Stack
```

For C++ we would have to get the stack (`backtrace_symbols()`), resolve each address (`dladdr()`) and demangle each symbol name (`abi::__cxa_demangle()`).

Caveats

- Memory allocation is one of these basic operation everything else builds on. One needs to allocate memory to load libraries and executables; needs to allocate memory to track memory allocations; and we hook onto it very early in the life of a process: the first pre-loaded library is the memory tracking library. Thus, any API call we make inside this interposition library can reserve surpises, especially in multi-threaded environments.
- The API functions we use to trace the stack can allocate memory. These allocations are also going through the hooks we installed. As we trace the new allocation, the hooks are activated again and another allocation is made as we trace this new allocation. We will run out of stack in this infinite loop. We break out of this pitfall by using a per-thread flag.
- The API functions we use to trace the stack can deadlock. Suppose we would use a lock while in our trace. We lock the trace lock and we call `dladdr()` which in turn tries to lock a dynamic linker internal lock. If on another thread `dlopen()` is called while we trace, dlopen() locks the same linker lock, then allocates memory: this will trigger the memory hooks and we now have the `dlopen()` thread wait on the trace lock with the linker lock taken. Deadlock.
- On some platforms (gcc 4.7.2 amd64) TLS calls would trip the memalign hook. This could result in an infinite recursion if the memalign hook in its turn, accesses a TLS variable.

What we got

Let's try again with our new library:

```
$ MALLOC_TRACE=logs/mtrace.stack.log LD_PRELOAD=./libmtrace.so
./dleaker
$ mtrace dleaker logs/mtrace.stack.log > logs/mtrace.stack.leaks.log
$ cat logs/mtrace.stack.leaks.log

Memory not freed:
-----------------
   Address     Size     Caller
0x08bf89b0      0x4  at 0x400ff727    <=== here
0x08bf89e8     0x11  at 0x400ff727
0x08bf8a00    0x400  at
/home/amelinte/projects/articole/memtrace/memtrace.v3/main.cpp:27
```

Apparently, not much of an improvement: the summary still does not get us back to line 25 in main.cpp. However, if we search for address 8bf89b0 in the trace log, we find this:

```
@ /usr/lib/libstdc++.so.6:(_Znwj+27)[0x400ff727] + 0x8bf89b0 0x4     <=== here
**[ Stack: 8
[0x40022251]  (./libmtrace.so+40022251)
[0x40022b43]  (./libmtrace.so+40022b43)
[0x400231e8]  (./libmtrace.so+400231e8)
[0x401cf905] __libc_malloc (/lib/i686/cmov/libc.so.6+35)
[0x400ff727] operator new(unsigned int) (/usr/lib/libstdc++.so.6+27) <== was: _Znwj
[0x80489cf] __gxx_personality_v0 (./dleaker+27f)
[0x40178450] __libc_start_main (/lib/i686/cmov/libc.so.6+e0)         <=== here
[0x8048791] __gxx_personality_v0 (./dleaker+41)
**] Stack
```

This is good, but having file and line information would be better.

File and line

Here we have a few possibilities:

- Run the address (e.g. 0x40178450 above) through the `addr2line` tool. If the address is in a shared object that the program loaded, it might not resolve properly.
- If we have a core dump of the program, we can ask gdb to resolve the address. Or we can attach to the running program and resolve the address.
- Use the API described here. The downside is that it takes a quite heavy toll on the performance of the program.

Resources

- The GNU C library manual [2]
- Using libbfd [3]
- Linux Programming Toolkit [1]

DIY:libmemleak

Building on libmtrace, we can go one step further and have an interposition library track the memory allocations made by the program. The library generates a report on demand, much like Valgrind does.

Libmemleak is significantly faster than valgrind but also has limited functionality (only leak detection).

Operation

`libmemtrace` has two drawbacks:

- The log file will quickly reach gigs
- You are left grepping the log to figure out what leaks when

A better solution would be to have an interposition library to collect memory operations information and to generate a report on-demand.

For `mmap()/munmap()` we have no choice but hook these directly. Thus, an call from within the application would first hit the hooks in `libmemleak`, then go to `libc`. For `malloc()realloc()/memalign()/free()` we have two options:

- Use `mtrace()/muntrace()` as before, to install hooks that will be called from within `libc`. Thus, a `malloc()` call would first go through `libc` which will then call the hooks in `libmemtrace`. This leave us at the mercy of `libc`.
- The second solution is to hook these like `m(un)map`.

The second solution also frees `mtrace()/muntrace()` for on-demand report generation:

- A first call to `mtrace()` will kick in data collection.
- Subsequent calls to `mtrace()` will generate reports.
- `muntrace()` will stop data collection and will generate a final report.
- `MALLOC_TRACE` is not needed.

The application can then sprinkle its code with `mtrace()` calls at strategic places to avoid reporting much noise. These calls will do nothing in a normal operation as long as `MALLOC_TRACE` is not set. Or, the application can be completely ignorant of the ongoing data collection (no `mtrace()` calls within the application code) and `libmemleak` can start collecting as early as being loaded and generate one report upon being unloaded.

To control the `libmemleak` functionality, an environment variable - `MEMLEAK_CONFIG` - has to be set before loading the library:

```
export MEMLEAK_CONFIG=mtraceinit
```

- `mtraceinit` will instruct the library to start collecting data upon being loaded. Default is off and the application has to be instrumented with `m(un)trace` calls.

Thus, all the hooks have to do is to call into the reporting:

```
extern "C" void *__libc_malloc(size_t size);
extern "C" void *malloc(size_t size)
{
    void *ptr = __libc_malloc(size);
    if ( _capture_on) {
            libmemleak::alloc(ptr, size);
    }

    return ptr;
}

extern "C" void __libc_free(void *ptr);
extern "C" void free(void *ptr)
{
    __libc_free(ptr);
    if (_capture_on) {
        libmemleak::free(ptr, 0); // Call to reporting
    }
    else {
        serror(ptr, "Untraced free", __FILE__, __LINE__);
    }
}

extern "C" void mtrace ()
{
    // Make sure not to track memory when globals get destructed
    static std::atomic<bool> _atexit(false);
    if (!_atexit.load(std::memory_order_acquire)) {
        int ret = atexit(muntrace);
        assert(0 == ret);
        _atexit.store(true, std::memory_order_release);
    }

    if (!_capture_on) {
        _capture_on = true;
    }
    else {
        libmemleak::report();
    }
}
```

Sample report

```
// Leaks since previous report
=====================================

// Ordered by Num Total Bytes
// Stack Key,  Num Total Bytes,  Num Allocs,  Num Delta Bytes
   5163ae4c,   1514697,          5000,        42420
...

11539977 total bytes, since previous report: 42420 bytes
Max tracked: stacks=6, allocations=25011


// All known allocations
=====================================

// Key   Total Bytes    Allocations
4945512: 84983 bytes in 5000 allocations
bbc54f2: 1029798 bytes in 10000 allocations
...

bbc54f2: 1029798 bytes in 10000 allocations
[0x4005286a] lpt::stack::detail::call_stack<lpt::stack::bfd::source_resolver>::call_stack()
(binaries/lib/libmemleak_mtrace_hooks.so+0x66) in crtstuff.c:0
[0x4005238d] _pstack::_pstack()
(binaries/lib/libmemleak_mtrace_hooks.so+0x4b) in crtstuff.c:0
[0x4004f8dd] libmemleak::alloc(void*, unsigned long long)
(binaries/lib/libmemleak_mtrace_hooks.so+0x75) in crtstuff.c:0
[0x4004ee7c] ?? (binaries/lib/libmemleak_mtrace_hooks.so+0x4004ee7c) in
 crtstuff.c:0
[0x402f5905] ?? (/lib/i686/cmov/libc.so.6+0x35) in ??:0
[0x401a02b7] operator new(unsigned int)
(/opt/lpt/gcc-4.7.0-bin/lib/libstdc++.so.6+0x27) in crtstuff.c:0
[0x8048e3b] ?? (binaries/bin/1001leakseach+0x323) in
/home/amelinte/projects/articole/lpt/lpt/tests/1001leakseach.cpp:68
[0x8048e48] ?? (binaries/bin/1001leakseach+0x330) in
/home/amelinte/projects/articole/lpt/lpt/tests/1001leakseach.cpp:74
[0x8048e61] ?? (binaries/bin/1001leakseach+0x349) in
/home/amelinte/projects/articole/lpt/lpt/tests/1001leakseach.cpp:82
[0x8048eab] ?? (binaries/bin/1001leakseach+0x393) in
/home/amelinte/projects/articole/lpt/lpt/tests/1001leakseach.cpp:90
[0x401404fb] ?? (/lib/i686/cmov/libpthread.so.0+0x401404fb) in ??:0
[0x4035e60e] ?? (/lib/i686/cmov/libc.so.6+0x5e) in ??:0

// Crosstalk: leaked bytes per stack frame
---------------------------------------
1029798 bytes: [0x8048e3b] ?? (binaries/bin/1001leakseach+0x323) in
```

```
/home/amelinte/projects/articole/lpt/lpt/tests/1001leakseach.cpp:68
...

// Mem Address, Stack Key, Bytes
-------------------------------------
   0x8ce7988,   bbc54f2,   4
...

This report took 44 ms to generate.
```

mallinfo

The `mallinfo()` API is rumored to be deprecated. But, if available, it is very useful:

```
#include <malloc.h>

namespace process {

class memory
{
public:

    memory() : _meminfo(::mallinfo()) {}

    int total() const
    {
        return _meminfo.hblkhd + _meminfo.uordblks;
    }

    bool operator==(memory const& other) const
    {
        return total() == other.total();
    }

    bool operator!=(memory const& other) const
    {
        return total() != other.total();
    }

    bool operator<(memory const& other) const
    {
        return total() < other.total();
    }

    bool operator<=(memory const& other) const
    {
        return total() <= other.total();
    }
```

```
    bool operator>(memory const& other) const
    {
        return total() > other.total();
    }

    bool operator>=(memory const& other) const
    {
        return total() >= other.total();
    }

private:

    struct mallinfo _meminfo;
};

} //process
```

```
#include <iostream>
#include <string>
#include <cassert>

int main()
{

    process::memory first;

    {
        void* p = ::malloc(1025);
        process::memory second;
        std::cout << "Mem diff: " << second.total() - first.total() << std::endl;
        assert(second > first);

        ::free(p);
        process::memory third;
        std::cout << "Mem diff: " << third.total() - first.total() << std::endl;
        assert(third == first);
    }
    {
        std::string s("abc");
        process::memory second;
        std::cout << "Mem diff: " << second.total() - first.total() << std::endl;
        assert(second > first);
    }

    process::memory fourth;
    assert(first == fourth);

    return 0;
```

```
}
```

References

- mallinfo [4]
- mallinfo deprecated [5]

/proc

Coarse grained information can be obtained from /proc:

```
#!/bin/ksh
#
# Based on:
#
http://stackoverflow.com/questions/131303/linux-how-to-measure-actual-memory-usage-of-an-application-or-process
#
# Returns total memory used by process $1 in kb.
#
# See /proc/PID/smaps; This file is only present if the CONFIG_MMU
# kernel configuration option is enabled
#

IFS=$'\n'

for line in $(</proc/$1/smaps)
do
   [[ $line =~ ^Private_Clean:\s+(\S+) ]] && ((pkb += ${.sh.match[1]}))
   [[ $line =~ ^Private_Dirty:\s+(\S+) ]] && ((pkb += ${.sh.match[1]}))
   [[ $line =~ ^Shared_Clean:\s+(\S+) ]]  && ((skb += ${.sh.match[1]}))
   [[ $line =~ ^Shared_Dirty:\s+(\S+) ]]  && ((skb += ${.sh.match[1]}))
   [[ $line =~ ^Size:\s+(\S+) ]]          && ((szkb +=
${.sh.match[1]}))
   [[ $line =~ ^Pss:\s+(\S+) ]]           && ((psskb +=
${.sh.match[1]}))
done

((tkb += pkb))
((tkb += skb))
#((tkb += psskb))

echo "Total private:        $pkb kb"
echo "Total shared:         $skb kb"
echo "Total proc prop:      $psskb kb Pss"
echo "Priv + shared:        $tkb kb"
echo "Size:                 $szkb kb"

pmap -d $1 | tail -n 1
```

References

- Memory usage script [6]

Various tools

- gdb-heap extension [7]
- Dr. Memory [8]
- Pin tool [9]
- Type-preserving heap profiler [10]

References

[1] http://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging
[2] http://www.gnu.org/s/hello/manual/libc/Allocation-Debugging.html
[3] http://www.beowulf.org/archive/2007-June/018558.html
[4] http://www.gnu.org/software/libc/manual/html_node/Statistics-of-Malloc.html
[5] http://udrepper.livejournal.com/20948.html
[6] http://permalink.gmane.org/gmane.comp.video.gstreamer.devel/10609
[7] https://fedorahosted.org/gdb-heap
[8] http://www.drmemory.org/
[9] http://www.pintool.org/
[10] http://www.slideshare.net/icsm2011/industry-program-analysis-and-verification-typepreserving-heap-profiler-for-c

Heap corruption

Electric Fence

Electric Fence is still the reference for dealing with heap corruption, even if not maintined for a while. RedHat ships a version that can be used as an interposition library.

Drawback: might not work with code that uses `mmap()` to allocate memory.

Duma

Duma is a fork of Electric Fence.

glibc builtin

`man (3) malloc`: Recent versions of Linux libc (later than 5.4.23) and GNU libc (2.x) include a malloc implementation which is tunable via environment variables. When `MALLOC_CHECK_` is set, a special (less efficient) implementation is used which is designed to be tolerant against simple errors, such as double calls of free() with the same argument, or overruns of a single byte (off-by-one bugs). Not all such errors can be protected against, however, and memory leaks can result. If `MALLOC_CHECK_` is set to 0, any detected heap corruption is silently ignored and an error message is not generated; if set to 1, the error message is printed on stderr, but the program is not aborted; if set to 2, abort() is called immediately, but the error message is not generated; if set to 3, the error message is printed on stderr and program is aborted. This can be useful because otherwise a crash may happen much later, and the true cause for the problem is then very hard to track down.

Varia

- http://clang.llvm.org/docs/AddressSanitizer.html

Stack corruption

Stack corruption is rather hard to diagnose. Luckily, gcc 4.x can instrument the code to check for stack corruption:

- **-fstack-protector**
- **-fstack-protector-all**

gcc will add guard variables and code to check for buffer overflows upon exiting a function. A quick example:

```
/* Compile with: gcc -ggdb -fstack-protector-all stacktest.c */

#include <stdio.h>
#include <string.h>

void bar(char* str)
{
    char buf[4];
    strcpy(buf, str);
}

void foo()
{
    printf("It survived!");
}

int main(void)
{
    bar("Longer than 4.");
    foo();
    return 0;
}
```

When run, the program will dump core:

```
$ ./a.out
*** stack smashing detected ***: ./a.out terminated
Aborted (core dumped)
```

```
Core was generated by `./a.out'.
Program terminated with signal 6, Aborted.
#0  0x0000003684030265 in raise () from /lib64/libc.so.6
(gdb) bt full
#0  0x0000003684030265 in raise () from /lib64/libc.so.6
No symbol table info available.
#1  0x0000003684031d10 in abort () from /lib64/libc.so.6
No symbol table info available.
#2  0x000000368406a84b in __libc_message () from /lib64/libc.so.6
```

```
No symbol table info available.
#3  0x00000036840e8ebf in __stack_chk_fail () from /lib64/libc.so.6
No symbol table info available.
#4  0x0000000000400584 in bar (str=0x400715 "Longer than 4.") at
stacktest.c:10
        buf =           "Long"
#5  0x00000000004005e3 in main () at stacktest.c:19
No locals.
```

Further reading

- http://www.drdobbs.com/cpp/240001832 [1]

References

[1] http://www.drdobbs.com/cpp/240001832

Deadlocks

Analysis

Searching for a deadlock means reconstructing the graph of dependencies between threads and resources (mutexes, semaphores, condition variables, etc.) - who owns what and who wants to acquire what. A typical deadlock would look like a loop in that graph. The task is tedious, as some of the parameters we are looking for have been optimized by the compiler into registers.

Below is an analysis of an x86_64 deadlock. On this platform, register r8 is the one containing the first argument: the address of the mutex:

```
(gdb) thread apply all bt
...
Thread 4 (Thread 0x419bc940 (LWP 12275)):
#0  0x0000003684c0d4c4 in __lll_lock_wait () from
/lib64/libpthread.so.0
#1  0x0000003684c08e1a in _L_lock_1034 () from /lib64/libpthread.so.0
#2  0x0000003684c08cdc in pthread_mutex_lock () from
/lib64/libpthread.so.0
#3  0x0000000000400a50 in thread1 (threadid=0x1) at deadlock.c:66
#4  0x0000003684c0673d in start_thread () from /lib64/libpthread.so.0
#5  0x00000036840d3d1d in clone () from /lib64/libc.so.6

Thread 3 (Thread 0x421bd940 (LWP 12276)):
#0  0x0000003684c0d4c4 in __lll_lock_wait () from
/lib64/libpthread.so.0
#1  0x0000003684c08e1a in _L_lock_1034 () from /lib64/libpthread.so.0
#2  0x0000003684c08cdc in pthread_mutex_lock () from
/lib64/libpthread.so.0
#3  0x0000000000400c07 in thread2 (threadid=0x2) at deadlock.c:111
#4  0x0000003684c0673d in start_thread () from /lib64/libpthread.so.0
#5  0x00000036840d3d1d in clone () from /lib64/libc.so.6
```

```
...


(gdb) thread 4
[Switching to thread 4 (Thread 0x419bc940 (LWP 12275))]#2
0x0000003684c08cdc in pthread_mutex_lock ()
   from /lib64/libpthread.so.0

(gdb) frame 2
#2  0x0000003684c08cdc in pthread_mutex_lock () from
/lib64/libpthread.so.0

(gdb) info reg
...
r8             0x6015a0      6296992
...

(gdb) p *(pthread_mutex_t*)0x6015a0
$3 = {
  __data = {
    __lock = 2,
    __count = 0,
    __owner = 12276,    <== LWP 12276 is Thread 3
    __nusers = 1,
    __kind = 0,         <== non-recursive
    __spins = 0,
    __list = {
      __prev = 0x0,
      __next = 0x0
    }
  },
  __size =      "\002\000\000\000\000\000\000\000\364/\000\000\001",
'\000' <repeats 26 times>,
  __align = 2
}

(gdb) thread 3
[Switching to thread 3 (Thread 0x421bd940 (LWP 12276))]#0
0x0000003684c0d4c4 in __lll_lock_wait ()
   from /lib64/libpthread.so.0

(gdb) bt
#0  0x0000003684c0d4c4 in __lll_lock_wait () from
/lib64/libpthread.so.0
#1  0x0000003684c08e1a in _L_lock_1034 () from /lib64/libpthread.so.0
#2  0x0000003684c08cdc in pthread_mutex_lock () from
/lib64/libpthread.so.0
```

```
#3  0x0000000000400c07 in thread2 (threadid=0x2) at deadlock.c:111
#4  0x0000003684c0673d in start_thread () from /lib64/libpthread.so.0
#5  0x00000036840d3d1d in clone () from /lib64/libc.so.6
#2  0x0000003684c08cdc in pthread_mutex_lock () from
/lib64/libpthread.so.0

(gdb) info reg
...
r8             0x6015e0      6297056
...

(gdb) p *(pthread_mutex_t*)0x6015e0
$4 = {
  __data = {
    __lock = 2,
    __count = 0,
    __owner = 12275,  <=== Thread 4
    __nusers = 1,
    __kind = 0,
    __spins = 0,
    __list = {
      __prev = 0x0,
      __next = 0x0
    }
  },
  __size =     "\002\000\000\000\000\000\000\363/\000\000\001",
'\000' <repeats 26 times>,
  __align = 2
}
```

Threads 3 and 4 are deadlocking over two mutexes.

Note: If gdb is unable to find the symbol pthread_mutex_t because it has not loaded the symbol table for pthreadtypes.h, you can still print the individual members of the struct as follows:

```
(gdb) print *((int*)(0x6015e0))
$4 = 2
(gdb) print *((int*)(0x6015e0)+1)
$5 = 0
(gdb) print *((int*)(0x6015e0)+2)
$6 = 12275
```

Automation

An interposition library can be built to automate deadlock analysis [1]. A significant number of APIs have to be interposed and even then there are cases that would go unnoticed by the library. For instance, one creative way to deadlock two threads without involving any userland locking mechanism is to have each thread join the other. Thus, interposition tools have limited diagnostic functionality.

There are a number of other tools available:

- Userspace lockdep [2]. Incipient work.
- Locksmith [3]. Basic.
- Valgrind Helgrind [4]. Does not suffer from the terrible slowdown other Valgrind tools have (in particular the memory analysis ones) but does not survives long on an amd64 platform.

pthreads

`pthreads` has some built-in support for certain synchronization mechanisms (e.g. `PTHREAD_MUTEX_ERRORCHECK` mutexes).

Also, there's a POSIX mutex construction attribute for robust mutexes (a recoverable mutex that was held by a thread whose process has died): an error return code indicates the earlier owner died; and lock acquisition implies acquired responsibility for dealing with any cleanup.

lockdep

On lockdep-enabled kernels, pressing Alt+SysRq+d will dump information about all the locks known to the kernel.

References

[1] http://linuxgazette.net/150/melinte.html
[2] http://lwn.net/Articles/536363/
[3] https://github.com/cmccabe/lksmith
[4] http://valgrind.org/docs/manual/hg-manual.html

Race conditions

Valgrind Helgrind [4]

- [v 3.7] On amd64 platforms it does not survive for long because of the vex disassembler.

Valgrind Drd [1]

- Same.

Relacy [2]

- C++0x/11 synchronization modeler/unit tests tool.

References

[1] http://valgrind.org/docs/manual/drd-manual.html
[2] http://www.1024cores.net/home/relacy-race-detector

Aiming for and measuring performance

gprof & -pg

To profile the application with gprof:

- Compile the code with `-pg`
- Link with `-pg`
- Run the application. This creates a file `gmon.out` in the current folder of the application.
- At the prompt, in the folder where gmon.out lives: `gprof path-to-application`

PAPI

The Performance Application Programming Interface (PAPI) [1] offers the programmer access to the performance counter hardware found in most major microprocessors. With a decent C++ wrapper [1], measuring branch mispredictions and cache misses (and much more) is literally one line of code away.

By default, these are the events that `papi::counters<Print_policy>` is looking for:

```
static  const events_type events = {
      {PAPI_TOT_INS, "Total instructions"}
    , {PAPI_TOT_CYC, "Total cpu cycles"}
    , {PAPI_L1_DCM,  "L1 load  misses"}
//  , {PAPI_L1_STM,  "L1 store  missess"}
    , {PAPI_L2_DCM,  "L2 load  misses"}
//  , {PAPI_L2_STM,  "L2 store  missess"}
    , {PAPI_BR_MSP,  "Branch mispredictions"}
};
```

The counters class is parameterized with a `Print_policy` instructing what to do with the counters once they get out of scope.

As an example, lets look a bit at these lines:

```
  const int nlines = 196608;
  const int ncols  = 64;
  char ctrash[nlines][ncols];
  {
      int x;
      papi::counters<papi::stdout_print> pc("by column"); //<== the famous one-line
      for (int c = 0; c < ncols; ++c) {
          for (int l = 0; l < nlines; ++l) {
              x = ctrash[l][c];
          }
      }
  }
```

The code just loops over an array but in the wrong order: the innermost loop iterates on

```
the outer index. While the result is the same whether we loop over the first index first
or over the last one, theorically, to preserve cache locality, the innermost loop should
iterate over the innermost index. This should make a big difference for the time it
```

takes to iterate over the array:

```
  {
      int x;
      papi::counters<papi::stdout_print> pc("By line");
      for (int l = 0; l < nlines; ++l) {
          for (int c = 0; c < ncols; ++c) {
              x = ctrash[l][c];
          }
      }
  }
```

papi::counters is a class wrapping around PAPI functionality. It will take a

```
snaphost of some performance counters (in our case, we are interested in cache misses
and in branch mispredictions) when a counters object is instantiated and another snapshot when
```

the object is destroyed. Then it will print out the differences.

A first measure, with non-optimized code (-O0), shows the following:

`Delta by column:`

```
PAPI_TOT_INS (Total instructions): 188744788 (380506167-191761379)
PAPI_TOT_CYC (Total cpu cycles): 92390347 (187804288-95413941)
PAPI_L1_DCM (L1 load  misses): 28427 (30620-2193)                  <==
PAPI_L2_DCM (L2 load  misses): 102 (1269-1167)
PAPI_BR_MSP (Branch mispredictions): 176 (207651-207475)           <==
```

Delta By line:

```
PAPI_TOT_INS (Total instructions): 190909841 (191734047-824206)
PAPI_TOT_CYC (Total cpu cycles): 94460862 (95387664-926802)
PAPI_L1_DCM (L1 load  misses): 403 (2046-1643)                     <==
PAPI_L2_DCM (L2 load  misses): 21 (1081-1060)
```

```
PAPI_BR_MSP (Branch mispredictions): 205934 (207350-1416)          <==
```

While the cache misses have indeed improved, branch mispredictions exploded.

```
Not exactly a good tradeoff. Down in the pipeline of the processor,
a comparison operation translates into a branch operation. Something
```

is funny with the unoptimized code the compiler generated.

Typically, branch machine code is generated directly by `if/else`

```
and ternary operators; and indirectly by virtual calls and by calls
```

`though pointers`

Maybe the optimized code (-O2) is behaving better? Or maybe not:

`Delta by column:`

```
PAPI_TOT_INS (Total instructions): 329 (229368-229039)
PAPI_TOT_CYC (Total cpu cycles): 513 (186217-185704)
PAPI_L1_DCM (L1 load  misses): 2 (1523-1521)
PAPI_L2_DCM (L2 load  misses): 0 (993-993)
PAPI_BR_MSP (Branch mispredictions): 7 (1287-1280)
```

Delta By line:

```
PAPI_TOT_INS (Total instructions): 330 (209614-209284)
PAPI_TOT_CYC (Total cpu cycles): 499 (173487-172988)
PAPI_L1_DCM (L1 load  misses): 2 (1498-1496)
PAPI_L2_DCM (L2 load  misses): 0 (992-992)
PAPI_BR_MSP (Branch mispredictions): 7 (1225-1218)
```

This time the compiler optimized the loops out! It figured we do not really use the data in the array, so it got rid of. Completely!

Let's see how this code behaves:

```
  {
      int x;
      papi::counters<papi::stdout_print> pc("by column");
      for (int c = 0; c < ncols; ++c) {
          for (int l = 0; l < nlines; ++l) {
              x = ctrash[l][c];
              ctrash[l][c] = x + 1;
          }
      }
  }
```

`Delta by column:`

```
PAPI_TOT_INS (Total instructions): 62918492 (63167552-249060)
PAPI_TOT_CYC (Total cpu cycles): 224705473 (224904307-198834)
PAPI_L1_DCM (L1 load  misses): 12415661 (12417203-1542)
PAPI_L2_DCM (L2 load  misses): 9654638 (9655632-994)
PAPI_BR_MSP (Branch mispredictions): 14217 (15558-1341)
```

Delta By line:

```
 PAPI_TOT_INS (Total instructions): 51904854 (115092642-63187788)
 PAPI_TOT_CYC (Total cpu cycles): 25914254 (250864272-224950018)
 PAPI_L1_DCM (L1 load  misses): 197104 (12614449-12417345)
 PAPI_L2_DCM (L2 load  misses): 6330 (9662090-9655760)
 PAPI_BR_MSP (Branch mispredictions): 296 (16066-15770)
```

Both cache misses and branch mispredictions improved by at least an order

```
of magnitude. A run with unoptimized code will show the same order of
```

improvement.

References

- Locality of reference

OProfile

OProfile offers access to the same hardware counters as PAPI but without having to instrument the code:

- It is coarser grained than PAPI - at function level.
- Some out of the box kernels (RedHat) are not OProfile-friendly.
- You need root access.

```
#!/bin/bash

#
# A script to OProfile a program.
# Must be run as root.
#


if [ $# -ne 1 ]
then
  echo "Usage: `basename $0` <for-binary-image>"
  exit -1
else
  binimg=$1
fi

opcontrol --stop
opcontrol --shutdown

# Out of the box RedHat kernels are OProfile repellent.
opcontrol --no-vmlinux
opcontrol --reset

# List of events for platform to be found in
/usr/share/oprofile/<>/events
opcontrol --event=L2_CACHE_MISSES:1000
```

```
opcontrol --start

$binimg

opcontrol --stop
opcontrol --dump


rm $binimg.opreport.log
opreport > $binimg.opreport.log

rm $binimg.opreport.sym
opreport -l > $binimg.opreport.sym


opcontrol --shutdown
opcontrol --deinit
echo "Done"
```

References

- OP Manual [2]
- IBM OP Intro [3]
- System Profiling [4]

perf [5]

`perf` is a kernel-based subsystem that provide a framework for performance analysis of the impact of the progams being run on the kernel. It covers: hardware (CPU/PMU, Performance Monitoring Unit) features; and software features (software counters, tracepoints).

perf list [6]

Lists the events available on a particular machine. These events vary based on the performance monitoring hardware and the software configuration of the system.

```
$ perf list

List of pre-defined events (to be used in -e):
  cpu-cycles OR cycles                               [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend    [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend      [Hardware event]
  instructions                                       [Hardware event]
  cache-references                                   [Hardware event]
  cache-misses                                       [Hardware event]
  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  bus-cycles                                         [Hardware event]
```

```
  cpu-clock                                          [Software event]
  task-clock                                         [Software event]
  page-faults OR faults                              [Software event]
  minor-faults                                       [Software event]
  major-faults                                       [Software event]
  context-switches OR cs                             [Software event]
  cpu-migrations OR migrations                       [Software event]
  alignment-faults                                   [Software event]
  emulation-faults                                   [Software event]

  L1-dcache-loads                                    [Hardware cache
event]
  L1-dcache-load-misses                              [Hardware cache
event]
  L1-dcache-stores                                   [Hardware cache
event]
  L1-dcache-store-misses                             [Hardware cache
event]
  L1-dcache-prefetches                               [Hardware cache
event]
  L1-dcache-prefetch-misses                          [Hardware cache
event]
  L1-icache-loads                                    [Hardware cache
event]
  L1-icache-load-misses                              [Hardware cache
event]
  L1-icache-prefetches                               [Hardware cache
event]
  L1-icache-prefetch-misses                          [Hardware cache
event]
  LLC-loads                                          [Hardware cache
event]
  LLC-load-misses                                    [Hardware cache
event]
  LLC-stores                                         [Hardware cache
event]
  LLC-store-misses                                   [Hardware cache
event]
  LLC-prefetches                                     [Hardware cache
event]
  LLC-prefetch-misses                                [Hardware cache
event]
  dTLB-loads                                         [Hardware cache
event]
  dTLB-load-misses                                   [Hardware cache
event]
  dTLB-stores                                        [Hardware cache
```

```
event]
  dTLB-store-misses                                  [Hardware cache
event]
  dTLB-prefetches                                    [Hardware cache
event]
  dTLB-prefetch-misses                               [Hardware cache
event]
  iTLB-loads                                         [Hardware cache
event]
  iTLB-load-misses                                   [Hardware cache
event]
  branch-loads                                       [Hardware cache
event]
  branch-load-misses                                 [Hardware cache
event]
  node-loads                                         [Hardware cache
event]
  node-load-misses                                   [Hardware cache
event]
  node-stores                                        [Hardware cache
event]
  node-store-misses                                  [Hardware cache
event]
  node-prefetches                                    [Hardware cache
event]
  node-prefetch-misses                               [Hardware cache
event]

  rNNN (see 'perf list --help' on how to encode it)  [Raw hardware
event descriptor]

  mem:<addr>[:access]                                [Hardware breakpoint]
```

Note: Running this as root will give out an extended list of events; some of the events (tracepoints?) require root privileges.

perf stat <options> <cmd> [7]

Gathers overall statistics for common performance events, including instructions executed and clock cycles consumed. There are option flags to gather statistics on events other than the default measurement events.

```
$ g++ -std=c++11 -ggdb -fno-omit-frame-pointer perftest.cpp -o perftest

$ perf stat ./perftest
 Performance counter stats for './perftest':

        379.991103 task-clock                #    0.996 CPUs utilized

                62 context-switches          #    0.000 M/sec
```

```
                0 CPU-migrations            #    0.000 M/sec

            6,436 page-faults               #    0.017 M/sec

      984,969,006 cycles                    #    2.592 GHz
                    [83.27%]
      663,592,329 stalled-cycles-frontend   #   67.37% frontend cycles
 idle     [83.17%]
      473,904,165 stalled-cycles-backend    #   48.11% backend  cycles
 idle     [66.42%]
    1,010,613,552 instructions              #    1.03  insns per cycle

                                            #    0.66  stalled cycles
per insn [83.23%]
       53,831,403 branches                  #  141.665 M/sec
                    [84.14%]
          401,518 branch-misses             #    0.75% of all branches
         [83.48%]

      0.381602838 seconds time elapsed

$ perf stat --event=L1-dcache-load-misses ./perftest
 Performance counter stats for './perftest':

        12,942,048 L1-dcache-load-misses


      0.373719009 seconds time elapsed
```

perf record [8]

Records performance data into a file which can be later analyzed using perf report.

```
$ perf record --event=L1-dcache-load-misses ./perftest

[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.025 MB perf.data (~1078 samples) ]

$ ls -al
...
-rw-------  1 amelinte amelinte  27764 Feb 17 17:23 perf.data
```

perf report [8]

Reads the performance data from a file and analyzes the recorded data.

```
$ perf report --stdio
# ========
# captured on: Sun Feb 17 17:23:34 2013
# hostname : bear
# os release : 3.2.0-4-amd64
# perf version : 3.2.17
# arch : x86_64
# nrcpus online : 4
# nrcpus avail : 4
# cpudesc : Intel(R) Core(TM) i3 CPU M 390 @ 2.67GHz
# cpuid : GenuineIntel,6,37,5
# total memory : 3857640 kB
# cmdline : /usr/bin/perf_3.2 record --event=L1-dcache-load-misses
./perftest
# event : name = L1-dcache-load-misses, type = 3, config = 0x10000,
config1 = 0x0, config2 = 0x0, excl_usr = 0, excl_kern = 0, id = {
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# ========
#
# Events: 274  L1-dcache-load-misses
#
# Overhead          Command   Shared Object      Symbol
# ........          .......  .................  ...................
#
#   95.93 percent  perftest  perftest           [.] 0xd35
#    1.06 percent  perftest  [kernel.kallsyms]  [k] clear_page_c
#    0.82 percent  perftest  [kernel.kallsyms]  [k] run_timer_softirq
#    0.42 percent  perftest  [kernel.kallsyms]  [k] trylock_page
#    0.41 percent  perftest  [kernel.kallsyms]  [k] __rcu_pending
#    0.41 percent  perftest  [kernel.kallsyms]  [k] update_curr
#    0.33 percent  perftest  [kernel.kallsyms]  [k] do_raw_spin_lock
#    0.26 percent  perftest  [kernel.kallsyms]  [k] __flush_tlb_one
#    0.18 percent  perftest  [kernel.kallsyms]  [k] flush_old_exec
#    0.06 percent  perftest  [kernel.kallsyms]  [k] __free_one_page
#    0.05 percent  perftest  [kernel.kallsyms]  [k] free_swap_cache
#    0.05 percent  perftest  [kernel.kallsyms]  [k] zone_statistics
#    0.04 percent  perftest  [kernel.kallsyms]  [k] alloc_pages_vma
#    0.01 percent  perftest  [kernel.kallsyms]  [k] mm_init
#    0.01 percent  perftest  [kernel.kallsyms]  [k] vfs_read
#    0.00 percent  perftest  [kernel.kallsyms]  [k] __cond_resched
#    0.00 percent  perftest  [kernel.kallsyms]  [k] finish_task_switch
#
# (For a higher level overview, try: perf report --sort comm,dso)
#
```

```
$ perf record -g ./perftest
$ perf report -g --stdio
...
# Overhead   Command      Shared Object                  Symbol
# ........  ........  .................  ......................
#
    97.23%  perftest  perftest           [.] 0xc75
            |
            --- 0x400d2c
                0x400dfb
                __libc_start_main
```

perf annotate [9]

Reads the input file and displays an annotated version of the code. If the object file has debug symbols then the source code will be displayed alongside assembly code. If there is no debug info in the object, then annotated assembly is displayed. Broken!?

```
$ perf annotate -i ./perf.data -d ./perftest --stdio -f
Warning:
The ./perf.data file has no samples!
```

perf top [10]

Performance top tool look-alike. It generates and displays a performance counter profile in realtime.

Valgrind: cachegrind

Cachegrind simulates a machine with two level ([I1 & D1] and L2) cache and branch (mis)prediction. Useful in that it annotates the code down to line level. Can significantly differ from machine's real CPU. Will not go very far on an AMD64 CPU (vex disassembler issues). Extremely slow, typically slows down the application 12-15 times.

DIY: libhitcount

libmemleak can be easily modified to keep track of what calls into a particular point in the code. Just insert an `mtrace()` call in that place.

How things scale up

```
L1 cache reference ......................... 0.5 ns
Branch mispredict ............................ 5 ns
L2 cache reference ........................... 7 ns
Mutex lock/unlock ........................... 25 ns
Main memory reference ...................... 100 ns
Compress 1K bytes with Zippy ............. 3,000 ns  =   3 µs
Send 2K bytes over 1 Gbps network ....... 20,000 ns  =  20 µs
SSD random read ........................ 150,000 ns  = 150 µs
Read 1 MB sequentially from memory ..... 250,000 ns  = 250 µs
Round trip within same datacenter ...... 500,000 ns  = 0.5 ms
Read 1 MB sequentially from SSD* ..... 1,000,000 ns  =   1 ms
Disk seek ........................... 10,000,000 ns  =  10 ms
```

```
Read 1 MB sequentially from disk .... 20,000,000 ns  =  20 ms
Send packet CA->Netherlands->CA .... 150,000,000 ns  = 150 ms
```

- Source: https://gist.github.com/2843375 [11]

```
Operation           Cost (ns)     Ratio
Clock period        0.6           1.0
Best-case CAS       37.9          63.2
Best-case lock      65.6          109.3
Single cache miss   139.5         232.5
CAS cache miss      306.0         510.0
Comms Fabric        3,000         5,000
Global Comms        130,000,000   216,000,000
Table 2.1: Performance of Synchronization Mechanisms on 4-CPU 1.8GHz
AMD Opteron 844 System
```

- Source: Paul E. McKenney

Notes on Hyper Threading

- Presumed loss of performance for intensive floating point calculations (only one FPU and one ALU (two pipelines) per core).
- http://ayumilove.net/hyperthreading-faq/ [12]
- http://en.wikipedia.org/wiki/Hyper-threading [13]

References

[1] http://icl.cs.utk.edu/papi/
[2] http://oprofile.sourceforge.net/doc/index.html
[3] http://www.ibm.com/developerworks/systems/library/es-oprofile/
[4] http://www.luciddb.org/wiki/SystemProfiling
[5] http://linux.die.net/man/1/perf
[6] http://linux.die.net/man/1/perf-list
[7] http://linux.die.net/man/1/perf-stat
[8] http://linux.die.net/man/1/perf-record
[9] http://linux.die.net/man/1/perf-annotate
[10] http://linux.die.net/man/1/perf-top
[11] https://gist.github.com/2843375
[12] http://ayumilove.net/hyperthreading-faq/
[13] http://en.wikipedia.org/wiki/Hyper-threading

Appendices

Things to watch for

Interrupted calls

A number of API functions return an error code if the call was interrupted by a signal. Usually this is not an error by itself and the call should be restarted. For instance:

```
int raccept( int s, struct sockaddr *addr, socklen_t *addrlen )
{
     int rc;

     do {
         rc = accept( s, addr, addrlen );
     } while ( rc == -1 && errno == EINTR );

     return rc;
}
```

The list of interruptible function differs from Unix-like platform to platform. For Linux see signal(7) [1]. Note that `sem_wait()` & co. are on list.

Spurious wake-ups

Threads waiting on a pthreads condition variable can be waken up even if the condition hs not been met. Upon waking up, the condition should be explicitly checked and return waiting if it is not met.

Code review checklist

1. Functional impact
    1. Which modules are affected; which functionality is impacted.
    2. Which modules & functionality is consumed by the changed code.
    3. Which modules/functionality is using it.
    4. Side effects.
2. Architecture
    1. Scalability
    2. Stability
    3. Functionality
    4. Robustness
3. Technicalities
    1. Generic
        1. First pass: logic. Correctness. What does it (tries to) do.
            1. Side effects.
            2. Why is code there?
        2. Second pass: error handling & odd conditions.
            1. Side effects.
            2. Are returned codes checked. Can the code handle them.
            3. Arguments:

        1. Border/out of bounds/strange values handled.
        2. Arg type: convenient?
        3. Where is the data coming from? Three levels upstream.
        4. Where are the outputs going? Three levels downstream.
            1. What are the possible/permitted outputs? Are all handled downstream?
    4. Variables: initialized?
    5. (over/under)flows:
        1. buffers.
        2. integral types.
    6. Asserted invariants & assumptions.
    7. Loops: off-by-one? infinite?
    8. Recursion: base case?
3. Third pass: thread safety.
4. Localization?
5. Duplicate information?
6. Math:
    1. Small values processed first?
    2. NaN checks/std::isnan & friends.

2. C
    1. return: resource handling.
    2. Use the N functions (ex. s'n'printf vs. sprintf).
    3. printf & friends: do formatters match type and number of arguments?
    4. Non re-entrant/non thread-safe API
    5. Unsafe/race condition afflicted API (mktemp())
    6. enums:
        1. all cases handled?
        2. default returns/logs/throws error
    7. Unspecified/undefined/implementation-defined/non-standard behavior or extensions
3. C++
    1. Fourth pass: exception handling:
        1. In destructors.
        2. Exception catching boundaries?
        3. Are resources RAII guarded.
    2. Fifth pass: performance.
        1. C++11: perfect forwarding & temporaries.
        2. How does it scale?
    3. Initialized data members.
    4. The big 6 (default constructor, destructor, copy/move constructors/operators).
    5. Virtual destructor?
    6. Resource handling: RAII.
    7. Where is created & destructed? Lifetime management?
    8. Thorough const & volatile.
    9. Are predicates stateless.
    10. Use size_t for std::string positions.
    11. Why casts?
4. Platform specific

1. POSIX/SUS
    1. Interrupted calls.
    2. Spurious wake-ups.
    3. Calls known to have race conditions.

Tools

- Source Code Audit [2]
- CERT C Secure Coding Standard [3]

/proc

A pseudo-filesystem exposing information about running processes:

```
# tree /proc/26041
/proc/26041
...
|-- cmdline             # Command line
|-- cwd ->              /current/working/folder/for/PID
|-- environ             # Program environment variables
|-- exe -> /bin/su
|-- fd                  # Open files descriptors
|   |-- 0 -> /dev/pts/21
|   |-- 1 -> /dev/pts/21
|   |-- 2 -> /dev/pts/21
|   `-- 3 -> socket:[113497835]
|-- fdinfo
|   |-- 0
|   |-- 1
|   |-- 2
|   `-- 3
|-- latency
|-- limits
|-- maps
|-- mem
|-- mountinfo
|-- mounts
|-- mountstats
...

# cat /proc/26041/status
...
VmPeak:   103276 kB     # Max virtual memory reached
VmSize:   103196 kB     # Current VM
VmLck:         0 kB
VmHWM:      1492 kB
VmRSS:      1488 kB     # Live memory used
...
Threads:        1
```

```
...
```

References

- Man page [4]
- Kernel doc [5]

sysstat, sar

- Site [6]

Other tools

addr2line

Given an address in an executable or an offset in a section of a relocatable object, `addr2line` translates it into file name and line number.

c++filt

A tool to demangle symbol names.

objdump

- Disassemble binary, with source code: `objdump -C -S -r -R -l <binary>`

Varia

- Source Code Audit [2]

Links

- Bunch of tools [7]
- dstat [8]

Abbreviations

- HLE: Hardware Lock Elision
- HT: Hyper Threading
- RTM: Restricted Transactional Memory
- TSX: Intel Transactional Synchronization Extension ([9], [10])

References

[1] http://www.kernel.org/doc/man-pages/online/pages/man7/signal.7.html
[2] http://www.openfoundry.org/en/resourcecatalog/Security/Source-Code-Audit
[3] https://www.securecoding.cert.org/confluence/x/HQE
[4] http://www.kernel.org/doc/man-pages/online/pages/man5/proc.5.html
[5] http://www.kernel.org/doc/Documentation/filesystems/proc.txt
[6] http://sebastien.godard.pagesperso-orange.fr/
[7] http://voices.canonical.com/tag/tools/?page=1
[8] http://dag.wieers.com/home-made/dstat
[9] http://software.intel.com/en-us/blogs/2012/11/06/exploring-intel-transactional-synchronization-extensions-with-intel-software
[10] http://lwn.net/Articles/533894/

References and further reading

Books & articles

- Agar, Eric; Writing Reliable AIX Daemons [1]
- Alexandrescu, Andrei; volatile: The Multithreaded Programmer's Best Friend [2]
- Butenhof, David; Programming with POSIX Threads [3]
- Drepper, Ulrich; What Every Programmer Should Know About Memory [4]
- McKenney, Paul; Is Parallel Programming Hard [5]
  ([git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git%20 git], blog [6], other papers [7])

Software

- Linux Programming Tools [1]

References

[1] http://www.redbooks.ibm.com/redbooks/pdfs/sg244946.pdf
[2] http://www.drdobbs.com/cpp/volatile-the-multithreaded-programmers-b/184403766
[3] http://www.amazon.ca/Programming-POSIX-Threads-David-Butenhof/dp/0201633922
[4] http://www.akkadia.org/drepper/cpumemory.pdf
[5] http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html
[6] http://paulmck.livejournal.com/
[7] http://www.rdrop.com/users/paulmck/

Article Sources and Contributors

Linux Applications Debugging Techniques *Source*: http://en.wikibooks.org/w/index.php?oldid=2586567 *Contributors*: Auremel, QuiteUnusual

Using GDB *Source*: http://en.wikibooks.org/w/index.php?oldid=2286567 *Contributors*: Auremel, QuiteUnusual, 1 anonymous edits

Core files *Source*: http://en.wikibooks.org/w/index.php?oldid=2547050 *Contributors*: Auremel, QuiteUnusual

Core analysis *Source*: http://en.wikibooks.org/w/index.php?oldid=2287595 *Contributors*: Auremel, QuiteUnusual

The call stack *Source*: http://en.wikibooks.org/w/index.php?oldid=2555000 *Contributors*: Auremel, 1 anonymous edits

The interposition library *Source*: http://en.wikibooks.org/w/index.php?oldid=2367781 *Contributors*: Auremel, QuiteUnusual

Leaks *Source*: http://en.wikibooks.org/w/index.php?oldid=2480566 *Contributors*: Auremel, Recent Runes, 2 anonymous edits

Heap corruption *Source*: http://en.wikibooks.org/w/index.php?oldid=2555868 *Contributors*: Auremel, Recent Runes

Stack corruption *Source*: http://en.wikibooks.org/w/index.php?oldid=2369464 *Contributors*: Auremel, 2 anonymous edits

Deadlocks *Source*: http://en.wikibooks.org/w/index.php?oldid=2512100 *Contributors*: Arjunaw, Auremel

Race conditions *Source*: http://en.wikibooks.org/w/index.php?oldid=2291887 *Contributors*: Auremel

Aiming for and measuring performance *Source*: http://en.wikibooks.org/w/index.php?oldid=2489204 *Contributors*: Auremel, QuiteUnusual

Appendices *Source*: http://en.wikibooks.org/w/index.php?oldid=2556084 *Contributors*: Auremel, 7 anonymous edits

References and further reading *Source*: http://en.wikibooks.org/w/index.php?oldid=2539850 *Contributors*: Auremel

Image Sources, Licenses and Contributors

File:GDB_TUI.png *Source*: http://en.wikibooks.org/w/index.php?title=File:GDB_TUI.png *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:Auremel

License