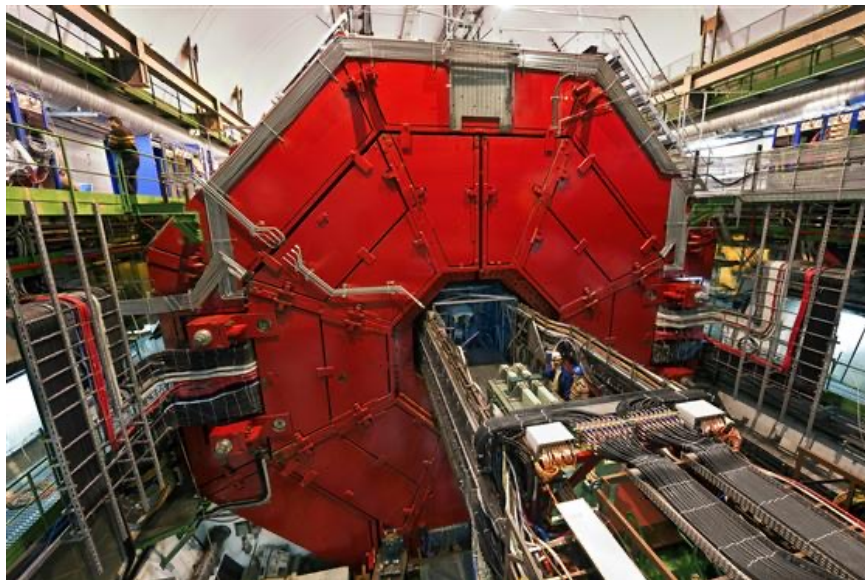


# Software Design in the Many-Cores era

**A. Gheata, S. Hageböck**

**CERN, EP-SFT/IT**

**CERN School of Computing 2023**



## Lecture III

# Understanding, Debugging and Profiling a Complex Multithreaded Application

# Outline of This Lecture

## The Goals:

- Understand the relation between performance and correctness
- Master the strategies to be able to **analyse**, **debug** and **profile** a complex parallel application

*Before running the application:*

- 1) Elements of static code analysis: Clang

**Three logical steps**

*If something goes wrong:*

- 2) Understanding and debugging a multithreaded application with GDB

*Now that it works, how fast is it?*

- 3) Elements of performance measurement: igprof

# Performance and Correctness

- **Correctness comes first:** if your program is buggy, unreliable, unpredictable, no performance consideration makes sense (at all)
- **Performance is then crucial\*:** algorithms translate to real machine code, running on real hardware with its own features (CPUs, memory hierarchy, accelerators)

**A high-quality test  
suite must be part of  
every software tool**

*“Make it work, make it ~~beautiful~~, make it  
fast”* *right*

\*For many areas of scientific computing at least



# Performance and Correctness

- **Correctness and performance: tightly correlated**
- **Correctness checked quickly and extensively** → runtime/memory improvements validated more easily
  - **Be in condition to label “changes”** in the final results as “acceptable”, “expected” or “in the wrong direction”
  - Pandora’s box: **what is the “right” result?** The one we had before? The new one? The “reference” one? **Not trivial at all!**
  - Use a grain of salt, **be in control of what happens!**

# Features of A Good Testsuite

- It's **easy to run**
  - One single command runs all tests
  - Tests can be selected, e.g. with regular expressions
- It's **automatically ran**
  - N times per day, or
  - Continuously check new code committed by developers
- **Results are easy to interpret**
  - E.g. Published on the web
  - Easy to track down problem, e.g. “test # 1206 failed with this output”

# Testing and parallel execution

- Test: minimal program aiming to stress a particular feature of the code
- Parallel code: no predictable order of operations possible
- The “same” test, execution pattern can be “different”
- **Solution:** properly designed tests
  - E.g. Maximising contention to “challenge” stability of the software

# Reproducibility

- E.g. two subsequent runs of the program produce the same histograms, identical bin by bin
- Simple for small setups
- Can be tricky with 5M lines, ~100 shared libraries
- Performance optimisations can lead to variations in final result (e.g. migration of entries to neighbouring bins)
  - Fundamental to remove all sorts of “noise”
- Non reproducibility in the sequential case: absence of control on the system
  - E.g. uninitialised variables, sloppy seeding of random generations, bogus memory access

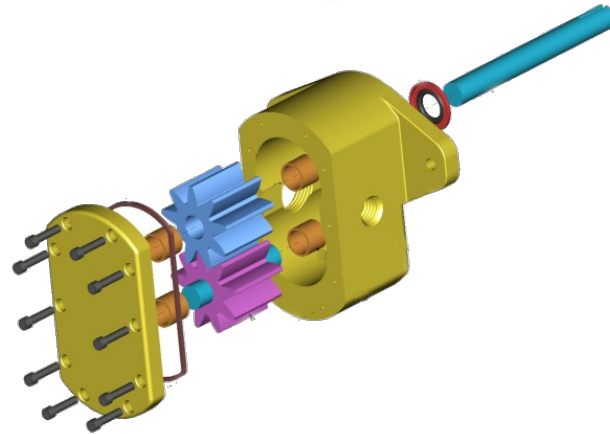
# Attitude Towards Testing

- Aim to **test-driven development**: write tests before code
  - Test features individually one by one
- For each bug reported/found: **create a reproducing test, add it to the suite, fix it.**
  - If it's not reproduced the bug does not exist!
- **Don't live with broken windows**: follow up each failure
  - Assume it always points to a serious problem
- Time invested in writing tests is strategic
  - It always rewards

**If a software tool or one of its functionalities is not tested always assume it does not work**



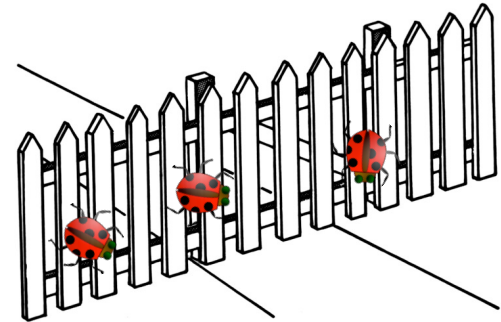
# Elements of Static Code Analysis





# Static Code Analysis

- Idea: embed static analysis in testing suites
- The procedure of **analysing the source code *before* compiling and running to automatically find bugs**
  - Rise (yet other) fences to protect against mistakes and bugs
  - Easily pluggable in big projects' build infrastructures
  - E.g. code blocks never executed because of faulty logic in if statements, *thread unsafe constructs*, etc.
- Several tools available, commercial and open source
  - Reference on the market: Coverity
  - Open source: Clang Static Analyzer



# LLVM and Clang

## LLVM

- Free and open source
- A compiler infrastructure
- Frontend [C++,C,...] → Optimizer → Backend [x86, CUDA, ...]
- <http://llvm.org>

## Clang

- LLVM frontend for C,C++ and Objective-C
- A possible alternative to GCC in some respects
- A lot of users - e.g. Apple, Intel (OpenCL)
- <http://clang.llvm.org>

Very powerful technology: e.g. C++ interpreter built on LLVM & Clang, **Cling**

<http://root.cern.ch/drupal/content/cling>



# Clang Static Analyser

## The static analyser is part of the clang frontend

It offers the possibility to examine the program code on two levels:

- Analysis of the **Abstract Syntax Tree** ( AST )
- **Symbolic Execution:**
  - Every possible path through the program is explored and validated
- **A battery of checks already included:** uninitialized access, dead stores, dereferencing null, invalid malloc calls, ...
  - User-defined can be added
- HTML report created automatically, detailed annotations of the source
- `scanbuild` tool: automatically replace the calls to the compiler in a makefile

<http://clang-analyzer.lvm.org/>

To fire static analysis: `scan-build make`

# An AST



logic decomposition  
of the program by the  
compiler



# Analysis for Thread Unsafety

Clang Static Analyzer: Custom checks can be added in form of a plugin written in C++

Checks for thread unsafety were developed by the LHC experiments

- Used in production for Q/A of experiments software
- **Useful in general!**

Some examples:

- Non const global/local statics
- Use of mutable keyword
- Use of `const_cast` to remove constness
- Other removals of constness (e.g. explicit cast)
- ...

**Note the importance  
of const correctness**

# Web Reports: an Example

FastJet tool taken as guinea pig: used to cluster jets by several experiments' software - <http://fastjet.fr/>

Example coming from an old version of fastjet!

```

944
945 void VoronoiDiagramGenerator::plotinit()
946 {
947     double dx,dy,d;
948
949     dy = ymax - ymin;
950     dx = xmax - xmin;
951     d = (double)(( dx > dy ? dx : dy ) * 1.1);
952     pxmin = (double)(xmin - (d-dx)/2.0);
953     pxmax = (double)(xmax + (d-dx)/2.0);
954     pymin = (double)(ymin - (d-dy)/2.0);
955     pymax = (double)(ymax + (d-dy)/2.0);
956     cradius = (double)((pxmax - pxmin)/350.0);
957     //GS unused: openpl();
958     //GS unused: range(pxmin, pymin, pxmax, pymax);
959 }
960
961
962 void VoronoiDiagramGenerator::clip_line(Edge *e)
963 {
964     Site *s1, *s2;
965     double x1=0,x2=0,y1=0,y2=0; //, temp = 0;
966
967     x1 = e->reg[0]->coord.x;
968     x2 = e->reg[1]->coord.x;
969
970     Value stored to 'x2' is never read
971
972     y1 = e->reg[0]->coord.y;
973     y2 = e->reg[1]->coord.y;
974
975     //if the distance between the two points is less than
976     //the square root of 2, then ignore it
977     //TODO improve/remove
978     //if(sqrt(((x2 - x1) * (x2 - x1)) + ((y2 - y1) * (y2 - y1))) < 1.414)
979     // {
980     //     return;
981     // }
982     pxmin = borderMinX;
983     pxmax = borderMaxX;
  
```

```

void VoronoiDiagramGenerator::clip_line(Edge *e)
{
    Site *s1, *s2;
    double x1=0,x2=0,y1=0,y2=0; //, temp = 0;

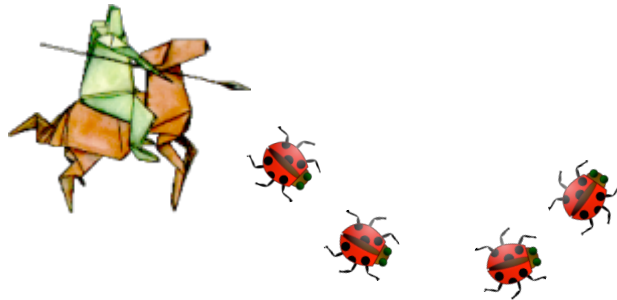
    x1 = e->reg[0]->coord.x;
    x2 = e->reg[1]->coord.x;

    Value stored to 'x2' is never read

    y1 = e->reg[0]->coord.y;
    y2 = e->reg[1]->coord.y;

    //if the distance between the two points is less than
  
```

# Understanding and Debugging - GDB



# Debugging

Suppose something is wrong with your application:

- It nicely terminates but yields wrong results (worst case scenario!)
- It crashes
- It runs forever occupying several CPUs
- It hangs forever with no CPU usage (e.g. a deadlock)

## **Effective debugging strategies and tools are the solution**

The same techniques are also handy not only in case of problems

- Suppose that the overall behavior of a very complex application (~MLOC) is to be understood
  - E.g. CMS/Atlas/LHCb/Alice reconstruction

# Debugging Strategies

## Write and use programs without bugs?

- There is *no such thing*, except in totally trivial cases
- All programs have and will have bugs

## If possible, try not to introduce bugs in the first place!

Debug printouts as *'poor man's solution'*:

- ⊕ Immediate to everybody: sometimes it's enough!
- ⊖ Hard (impossible) to add printouts in 3<sup>rd</sup> party libraries
- ⊖ Distract the user from focussing on the debugging itself
- ⊖ Hard to use in a parallel program, encourage Heisenbugs influencing timing behaviour



**Or better: Use a debugger like GDB**





# GDB: The GNU Project Debugger

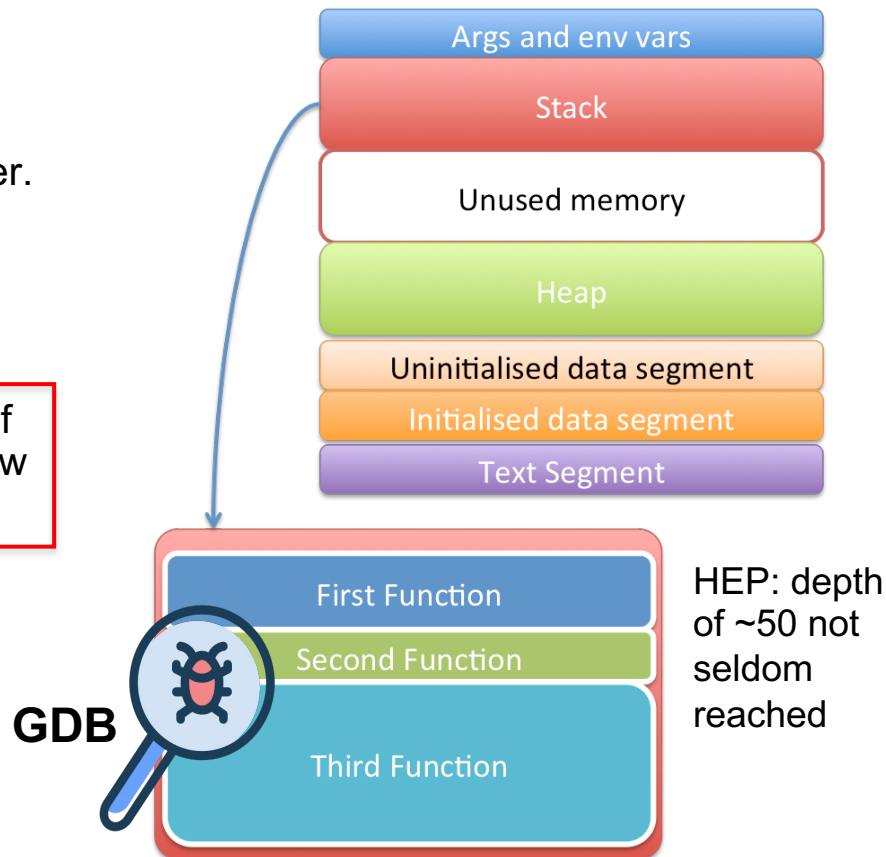
- **Free and open source**, available on every Linux box
- GDB is an interactive command line tool which can “see”:
  - Within a program during its execution
  - A posteriori, what a program was doing when it crashed
- Works with applications written in C and C++ (among other languages)
- **No recompilation needed** (although debugging symbols can be handy)
- **Stop the execution at some specified point**
  - Execute line by line, stepping into functions if needed
- Examine what is happening: e.g. print variable content
- **Thread aware: e.g. Stop threads, switch among them ...**



<http://www.gnu.org/software/gdb/>

# Reminder: A Program in Memory

- **Text Segment:** code to be executed.
- **Initialized Data Segment:** global variables initialized by the programmer.
- **Uninitialized Data Segment:** This segment contains uninitialized global variables.
- **The stack:** The stack is a collection of stack frames. It grows whenever a new function is called. “Thread private”.
- **The heap:** Dynamic memory (e.g. requested with “new”).



# An Example

```
#include <iostream>

void display(int x, int *xp) {
    std::cout << "In display():\n"
                << " o value of x is " << x
                << ", address of x is " << &x <<std::endl
                << " o xp points to " << xp
                << " which holds " << *xp <<std::endl; }

int main() {
    int a = 42;
    int *ap = &a;
    std::cout << "In main():\n"
                << " o value of a is " << a
                << ", address of a is " << &a << std::endl
                << " o ap points to " << ap
                << " which holds " << *ap << std::endl;

    display(a, ap);
    return 0; }
```

```
g++ -o myExample
myExample.cpp -g
```

```
To fire gdb: gdb myexecutable
```

# An Example

```

#i $ gdb myExample
vo [ ... Some output ... ]
(gdb) run
Starting program: /Users/<whoever>/gdb/myExample
Reading symbols for shared libraries
+++..... done
In main():
  o value of a is 5, address of a is 0x7fff5fbff744
in  o ap points to 0x7fff5fbff744 which holds 5
In display():
  o value of x is 5, address of x is 0x7fff5fbff71c
  o xp points to 0x7fff5fbff744 which holds 5

Program exited normally.
    << " o ap points to << ap
    << " which holds " << *ap << std::endl;
display(a, ap);
return 0; }
  
```

To fire gdb: `gdb myexecutable`

# Break Points



- So far so good – you could have done this already without GDB!
- **But**, GDB allows you to stop the execution of the application at a certain line or function with **break points**:

```
(gdb) break 12
Breakpoint 1 at 0x100000c60: file myExample.cpp, line 12.
(gdb) run
Starting program: /Users/<whoever>/gdb/myExample

Breakpoint 1, main () at myExample: 12
12 int *ap = &a;
(gdb)
```

The break could have been introduced when a certain function is invoked:

→ ***break <function name>***

(“***break display***” in our case)

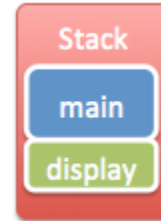
Impossible to do with  
printouts 😊



# And Now?

You can dump the stack with *where*:

```
(gdb) where
#0  display (x=42, xp=0x7fff5fbff7c4) at myExample:6
#1  0x0000000100000d2c in main () at myExample.cpp:14
```



See some of the surrounding code with *list*:

```
(gdb) list
#include <iostream>

1 void display(int x, int *xp) {
2     std::cout << "In display():\n"
               << " o value of x is " << x
               << ", address of x is " << &x <<std::endl
               << " o xp points to " << xp
               << " which holds " << *xp <<std::endl; }

3 int main() {
4     int a = 42;
5     int *ap = &a;
```



# Interlude: Debugging Symbols

**The compiler does not automatically bring the names of the symbols in the executables and libraries, the machine does not need them!**

Humans do: include *debugging symbols* in the compiled binaries.

- Names of variables, functions, classes, namespaces, ...

**Debugging symbols, 3 facts to remember:**

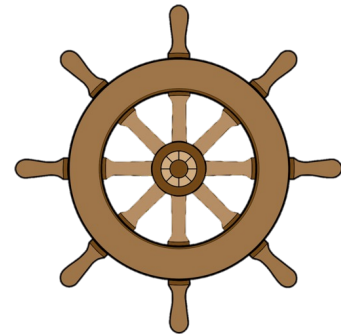
- **Do not** slow down the program!
- **Do not** increase its memory footprint!
- **Do** make binaries bigger (more disk space needed)!

With GCC:  
`g++ [...] -g`

# Navigating Program Execution

To “navigate” the program execution you can use:

- **step**: continue running until control reaches new line. “Step into” functions
- **next**: like step but functions are executed without stopping
- **finish**: continue until end of current stack frame
- **return <expression>**: prematurely exit the stack returning expression.
- **break**: show break points list
  - **disable <n>**: disable break point n
  - **enable <n>**: enable break point n
  - **delete <n>**: delete break point n
- **info threads**: show threads
- **thread <n>**: step into thread n



search “gdb cheat sheet”

# The Print Statement

```
1 #include "time.h"
2 #include <iostream>
3 int main() {
4     int t = clock();
5     std::cout << t << std::endl;
6     return 0;
7 }
```

**print** allows you to inspect the value of a variable.

```
(gdb) break 5
Breakpoint 1 at 0x100000d50: file ex12_2.cpp, line 5.
(gdb) run
Starting program: /Users/danilopiparo/gdb/ex12_2
Reading symbols for shared libraries
++..... done

Breakpoint 1, main () at ex12_2.cpp:5
5         std::cout << t << std::endl;
(gdb) print t
$1 = 6637
(gdb) next
6637
6         return 0;
```

# Interlude 3: Machine Code with GDB

Food  
For  
Thought

```
double myFloor(double x) {  
    const int xi = int(x);  
    return x < 0 ? xi - 1 : xi;  
}  
int main() {  
    myFloor(-3.14);  
}
```

- Looking at the assembly is the only way to understand what the compiler actually did
- GDB allows to do that easily with `disass`

# Interlude 3: Machine Code with GDB

Food  
For  
Thought

```
(gdb) disass /m myFloor
Dump of assembler code for function myFloor(double):
1      double myFloor(double x){
2          const int xi = int(x);
      0x00000000004004e0 <+0>:    cvttss2sd %xmm0,%eax
3          return x<0?xi-1:xi;
      0x00000000004004e4 <+4>:    cmpltd 0x113(%rip),%xmm0          # 0x400600
      0x00000000004004ed <+13>:   lea    -0x1(%rax),%edx
      0x00000000004004f0 <+16>:   cvtsi2sd %eax,%xmm2
      0x00000000004004f4 <+20>:   cvtsi2sd %edx,%xmm1
      0x00000000004004f8 <+24>:   andpd  %xmm0,%xmm1
      0x00000000004004fc <+28>:   andnpd %xmm2,%xmm0
      0x0000000000400500 <+32>:   orpd  %xmm1,%xmm0
4          }
      0x0000000000400504 <+36>:   retq
End of assembler dump.
```



# GDB And Threads



- GDB allows to inspect the behaviour of the threads of a process
  - `info threads`: display running threads
  - `thread <n>`: step into a thread

```
#include <thread>
#include <vector>
#include <chrono>
void sleep() {
    std::this_thread::sleep_for(std::chrono::seconds(100)); }
int main() {
    std::vector<std::thread> myThreads;
    for (int i=0; i<2; i++) myThreads.emplace_back(std::thread(sleep));
    // Line 11
    for (auto& t : myThreads) t.join();
}
```

```
$ gdb ./threadsSleep  
[ ... some output ... ]  
Reading symbols from /home/dpiparo/CSC/Examples/threadsSleep...done.  
(gdb) break 11
```

Set a break point at line 11



```
$ gdb ./threadsSleep
[ ... some output ... ]
Reading symbols from /home/dpiparo/CSC/Examples/threadsSleep...done.
(gdb) break 11
Breakpoint 1 at 0x400a8f: file threadsSleep.cpp, line 11.
(gdb) run
Starting program: /home/dpiparo/CSC/Examples/threadsSleep
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff6fe7700 (LWP 4440)]
[New Thread 0x7ffff67e6700 (LWP 4441)]

Breakpoint 1, main () at threadsSleep.cpp:12
12      for (auto& t : myThreads)
```

- GDB informs us it found the line at which it will break
- Run the application
- GDB informs us that 2 threads were spawned
- The breakpoint is reached

```
$ gdb ./threadsSleep
[ ... some output ... ]
Reading symbols from /home/dpibaro/CSC/Examples/threadsSleep...done.
```

- Get info about threads
- GDB prints the threads ids and which function is being executed
- The \* identifies the thread where the break point was successful
- By default GDB freezes all threads simultaneously at a breakpoint
  - “Take a snapshot of the execution status”

```
14         for (auto& t : myThreads)
(gdb) info threads
Id  Target Id          Frame
 3  Thread 0x7ffff67e6700 (LWP 4441) "threadsSleep" 0x00007ffff76b252d in nanosleep () at
../sysdeps/unix/syscall-template.S:82
 2  Thread 0x7ffff6fe7700 (LWP 4440) "threadsSleep" 0x00007ffff76b252d in nanosleep () at
../sysdeps/unix/syscall-template.S:82
* 1  Thread 0x7ffff7fd4740 (LWP 4437) "threadsSleep" main () at threadsSleep.cpp:14
```

```
$ gdb ./threadsSleep
```

- Suppose we are interested in thread 2, let's switch to it
- GDB informs us we are now in thread 2
- The cryptic messages are due to the fact that we compiled our exe with debugging symbols, not all the components it depends on!

```
[New Thread 0x7ffff6e6700 (LWP 4441)]
```

```
Breakpoint 1, main () at threadsSleep.cpp:12
```

```
14     for (auto& t : myThreads)
```

```
(gdb) info threads
```

```
Id      Target Id      Frame
 3      Thread 0x7ffff67e6700 (LWP 4441) "threadsSleep" 0x00007ffff76b252d in nanosleep () at
../sysdeps/unix/syscall-template.S:82
 2      Thread 0x7ffff6fe7700 (LWP 4440) "threadsSleep" 0x00007ffff76b252d in nanosleep () at
../sysdeps/unix/syscall-template.S:82
•1      Thread 0x7ffff7fd4740 (LWP 4437) "threadsSleep" main () at threadsSleep.cpp:12
```

```
(gdb) thread 2
```

```
[Switching to thread 2 (Thread 0x7ffff6fe7700 (LWP 4440))]
```

```
#0  0x00007ffff76b252d in nanosleep () at ../sysdeps/unix/syscall-template.S:82
```

```
82      ../sysdeps/unix/syscall-template.S: No such file or directory.
```

```

$ gdb ./threadsSleep
[ ... some output ... ]
Reading symbols from /home/dpibaro/CSC/Examples/threadsSleep...done.
(gdb) Break
(gdb) Start
[Thread
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff6fe7700 (LWP 4440)]
[New Thread 0x7ffff67e6700 (LWP 4441)]

Breakpoint 1, main () at threadsSleep.cpp:12
14     for (auto& t : myThreads)
(gdb) info threads
  Id  Target Id          Frame
   3   Thread 0x7ffff67e6700 (LWP 4441) "threadsSleep" 0x00007ffff76b252d in nanosleep () at
  ../sysdeps/unix/syscall-template.S:82
   2   Thread 0x7ffff6fe7700 (LWP 4440) "threadsSleep" 0x00007ffff76b252d in nanosleep () at
  ../sysdeps/unix/syscall-template.S:82
  * 1   Thread 0x7ffff7fd4740 (LWP 4437) "threadsSleep" main () at threadsSleep.cpp:12
(gdb) thread 2
[Switching to thread 2 (Thread 0x7ffff6fe7700 (LWP 4440))]
#0  0x00007ffff76b252d in nanosleep () at ../sysdeps/unix/syscall-template.S:82
82  ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb) where
#0  0x00007ffff76b252d in nanosleep () at ../sysdeps/unix/syscall-template.S:82
#1  0x000000000400caf in sleep() () at /usr/include/c++/4.8/thread:279
#2  0x00007ffff7b87a10 in ?? () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#3  0x00007ffff76aae9a in start_thread (arg=0x7ffff6fe7700) at pthread_create.c:308
#4  0x00007ffff73d7ccd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:112
#5  0x0000000000000000 in ?? ()
(gdb)

```

• Now let's print the stack of thread number 2!



# GDB And Threads



- **By default GDB stops all threads simultaneously if a breakpoint is reached** (so called “stop mode”)
- It allows also to stop the thread where the breakpoint was reached and let the others proceed (“**non-stop mode**”)
  - De facto the user can bend the runtime behaviour of the application to her needs!

```
# Enable the async interface.  
set target-async 1  
  
# Pagination breaks non-stop.  
set pagination off  
  
# Finally, turn it on [off]!  
set non-stop on [off]
```

Commands to switch between stop and non-stop modes within the gdb prompt

# More GDB (Black) Magic

Suppose your program behaves in a weird way now.

- You can “attach” gdb to a running process (e.g. 300% CPU since minutes...)
- ***gdb <PID>***

Impossible to do with  
printouts 😊

To get your pid:  
`ps aux | grep <Program name>`

Suppose your program crashed after hours of running, leaving you with no plots, but a core dump.

- You can resume it as it was at the moment of the crash
- ***gdb program core-file***





# Helgrind and DRD

- Another pair of tools useful for debugging parallel programs
- Part of the Valgrind suite
- Allow to catch thread errors at runtime
  - `valgrind --tool=helgrind ./myProgram`
- **Detection of potential thread unsafe operations, lock ordering problems, ...**
  - Difference between DRD and Helgrind: detection algorithms
- Downside: false positives ☹️
- Complementary tools: address and thread sanitiser offered by CLANG and GCC compiler suites.



# High Level Profiling





# A Simple Question


**Q:** Why should we strive for software performance, correctness, efficiency, ultimately throughput?

# For Money!



From "The Wolf of Wall Street"

# Code Optimisation

- When dealing with large software projects, **performance measurement is daily business**
  - Especially for multithreaded applications: *parallel Vs serial case, performance of different configurations of the parallel applications ...*
- **The identification of the hotspots** (and their removal) is worth an enormous amount of resources
  - But don't optimise before you know "what"! 
- A plethora of tools available, covering all quantities related to performance
- Focus on a simple and easy to use one: igprof
  - Identify features common to all profilers
- Igprof answers questions like:
  - What are the symbols that have the longest runtime?
  - What are the symbols that allocate the most memory?



<http://igprof.org/>

# The Golden Rule of Optimisation

Don't develop theories,  
measure your program!



***It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts. Sherlock Holmes***



# Igprof

- Very general tool, free and open source
- **Measure runtime (speed) and memory of an application**
  - Works in the multithreaded case
  - Different metrics available
- **Little or negligible overhead**
  - Runtime: 50 MB RSS and ~no runtime overhead
  - Memory: it depends. For a large HEP application (600 libraries, ~2GB of memory, allocations at ~1MHz): 1 GB RSS and 250% runtime
- **Non intrusive**
  - No instrumentation needed
  - No Kernel modules needed
  - Hooking mechanism implemented

Results can be looked at as web or ASCII reports

*Today we focus on  
the runtime  
measurements*

# Igprof Runtime Profiler

- **Igprof measures the time spent in all the functions in seconds**
  - Easy overall view, understand rapidly where the time is spent
- **Two kinds of costs per symbol (typical of ~every profiler):**
  - **Cumulative cost:** time spent in the symbol and callees (i.e. `int main()` has the biggest cumulative cost)
  - **Self cost:** time spent in the symbol itself

Both are necessary to find hot spots

Produce profiling information:

```
igprof -d -pp -z -o myProfInfo.pp.gz myApp [arg1 arg2 ...]
```

Produce report out of it:

```
igprof-analyse --sqlite -d -v -g myProfInfo.pp.gz | sqlite3 igreport_perf.sql3
```

# Igprof: An Example of CMS Simulation

## igprof\_cmssw610pre2 - igprof-navigator, cgi-bin

[Back to profiles index](#)

**Counter: PERF\_TICKS, first 1000 entries**

**Sorted by self cost**

[Sort by cumulative cost](#)

Rank	Total %	Self	Symbol name
32	5.70	277.42	<u>G4CrossSectionDataStore::GetCrossSection(G4DynamicParticle const*, G4Element const*, G4Material const*)</u>
52	3.77	183.18	<u>ieee754 log</u>
54	3.40	165.40	<u>G4Mag UsualEqRhs::EvaluateRhsGivenB(double const*, double const*, double*) const</u>
55	2.98	145.07	<u>G4HadronCrossSections::CalcScatteringCrossSections(G4DynamicParticle const*, int, int)</u>
45	2.16	105.28	<u>G4Navigator::LocateGlobalPointAndSetup(CLHEP::Hep3Vector const&amp;, CLHEP::Hep3Vector const*, bool, bool)</u>
72	2.04	99.18	<u>init</u>
63	1.94	94.54	<u>G4PolyconeSide::DistanceAway(CLHEP::Hep3Vector const&amp;, bool, double&amp;, double*)</u>
92	1.80	87.42	<u>ieee754 exp</u>
95	1.74	84.57	<u>ieee754 atan2</u>
43	1.62	78.85	<u>G4ClassicalRK4::DumbStepper(double const*, double const*, double, double*)</u>

Real measurement, not the latest greatest: it does not reflect the current state of the CMS software.

# Igprof: An Example of CMS Simulation

## igprof\_cmssw610pre2 - igprof-navigator, cgi-bin

[Back to profiles index](#)

Counter: PERF\_TICKS, first 1000 entries

Sorted by self cost

Counter: PERF\_TICKS

Rank	% total	Counts		Paths		Symbol name
		to / from this	Total	Including child / parent	Total	
	1.30	63.33	73.22	6	6	<a href="#">G4HadronCrossSections::GetInelasticCrossSection(G4DynamicParticle const*, int, int)</a>
	2.09	101.64	115.76	8	8	<a href="#">G4HadronCrossSections::GetElasticCrossSection(G4DynamicParticle const*, int, int)</a>
[55]	3.39	145.07	19.90	14	14	<a href="#">G4HadronCrossSections::CalcScatteringCrossSections(G4DynamicParticle const*, int, int)</a>
	0.16	7.82	46.58	5	158	<a href="#">pow</a>
	0.13	6.45	26.86	11	24	<a href="#">G4HadronCrossSections::GetParticleCode(G4DynamicParticle const*)</a>
	0.07	3.58	101.50	9	192	<a href="#">exp</a>
	0.04	2.05	104.31	10	1,349	<a href="#">_init</a>

[Back to summary](#)

92	1.80	87.42	<a href="#">ieee754 exp</a>
95	1.74	84.57	<a href="#">ieee754 atan2</a>
43	1.62	78.85	<a href="#">G4ClassicalRK4::DumbStepper(double const*, double const*, double, double*)</a>

Real measurement, not the latest greatest: it does not reflect the current state of the CMS software.



# Take Away Messages

Dealing with a parallel application is complex:

Use procedures to **rise fences to protect against mistakes**, like static analysis to find bugs in an automatic way

- **Embed such tools in the build infrastructure of your SW**

Use tools to **inspect, manipulate their behaviour at runtime, like GDB**

- Become familiar with them, multithreaded programs are tough to debug

**Use tools to measure performance**, do not speculate

- Start from simple yet powerful tools like igprof
- Choose more complex ones to dive into the details