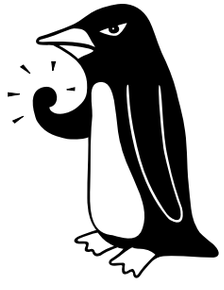
Debugging and Testing Tools



Warwick RSE

12/12/2017

Prototools

Mk 1 Human Brain

- Several of the sorts of bug in this workshop are logical errors
- No automated tool yet in existence will find these
- Testing goes some way towards catching these, but can't catch everything
- The most vital debugging and testing tool is your own brain.
- Always start at the first error! Often the rest are the same error popping up later on.

Compilers

- Compilers offer a range of tools to help
- Compile time
  - Warnings for breach of language standards
    - Warnings for uninitialised variables
    - Compiler warnings are not usually benign!
- Run time
  - Out of bounds warnings on arrays

Optimisation

- Modern compilers optimise code
  - Reorder code, subject to code dependencies
  - Why keeping to standards and away from undefined behaviour is important
    - Compiler might think it can reorder code when it can’t
- Compiled code might no longer map exactly to source code

Optimisation

- Probably want to debug code with optimisation off
  - "-O0" for most compilers
- This might appear to fix your problem
  - Might still be able to find problem with debugger
  - Can mean uninitialised variable
  - Can mean misusing language standards

Program output

- Despite your best efforts of testing etc. sometimes bugs crop up only during real runs
- Only have normal diagnostic output
  - Especially if running on large HPC machines
- Make sure minimal useful information is printed by the code itself

Symbolic Debuggers

Symbolic Debuggers

- External program that you run your code in
- Lets you add breakpoints to your code where it will pause execution
- When a breakpoint is reached, lets you print the state of internal variables
  - Can set variables to new values too if that will help
- When certain types of errors such as segmentation (seg) faults occur, stops automatically
- Can then use “backtrace” functions to find where the code has crashed and how it got there

GDB

- Most popular free symbolic debugger for compiled codes is **GDB**

- Part of the Gnu Compiler Collection (**GCC**)

- Have to compile code with "**debugging symbols**"
  - Usually "**-g**" flag to most compilers

- Then run your code through gdb

- **gdb {my_code_name}**

- End up at GDB's own internal command line

Print - like debugging

- Can use debugger in the same way as print statement debugging from earlier
  - Better because you don't have to recompile and can change your mind about what output you want as the code runs
- You can set a break point with "**b**" or "**break**" and then either a line number or a function name
  - **b 123**
  - **b my_func**
  - **b myfile:my_func** or **b myfile:123** if you have multiple files
- GDB will stop just **before** the line specified

Execution control

- To start running your code type
  - **run**
- Once a breakpoint has been reached type
  - **continue** to continue execution from the breakpoint
  - **run** to restart the code from scratch

Printing data

- You can print all local variables at the point where GDB has stopped using
  - **info locals**
- You can print the value of variables by typing
  - **print {varname}**
- You can print the value of an element of an array by typing
  - **print {varname}({array_position, …})**

Printing data

- Gets more difficult with dynamically allocated arrays and pointer

- See http://www.delorie.com/gnu/docs/gdb/gdb_54.html for more details

Conditional breaks

- You can add a breakpoint that only triggers conditionally, for example
  - **break {linenum} if {argname} >0**
- Same if breaking on a function name rather than a line number

Removing breakpoints

- You can delete a breakpoint using
  - **clear {function_name}**
  - **clear {line_number}**
  - **delete {breakpoint_number}**

Frames or traces

- Each time a function is called a new **stack_frame** is created

- Contains information about
  - Calling parameters
  - Return point for function

- This creates a hierarchy of stack frames that let you know how your code got to where it is

Frames or traces

- GDB can show you this stack hierarchy using the command

  - **backtrace** or **bt**

```
(gdb) backtrace
#0 push_particles() at src/particles.F90 : 135
#1 0x00000001000a19be in pic() at src/epoch2d.F90:190
#2 0x00000001000a1caa in main (argc=1,
argv=0x7fff5fbffb18) at src/epoch2d.F90:35
```

- Number on left is level number. 0 is current level

- Shows filenames and line numbers at the point where the code's execution is currently halted

Frames or traces

- Can only directly examine global variables and variables in the active stack frame
- Can step between stack frames to see variables
- Uses
  - **up** - Go to the next highest numbered stack frame (towards start of calling program)
  - **down** - Go to the next lowest numbered stack frame (towards the point where the code is currently stopped)
  - **frame {frame_number}** - Go to a specific numbered frame

Frames or traces

- Can be very useful
- You know you are getting a negative number as a parameter to a function that needs positive numbers, so you put a breakpoint there
- You then move up one stack frame and examine the variables in the function that calls the failing function to see why it's giving you a negative number

Typical example

```
 >>gdb . / eg1
(gdb) run
Program received signal SIGSEGV, Segmentation fault .
 15 printf(“%d\n",ptr);
(gdb) break 15
(gdb) run
 The program being debugged has been started already .
 Start it from the beginning ? ( y or n) y

 Breakpoint 1 , main (argc=1, argv=0 x 7fff5fbff8c0) at
 eg1.c:15
 15 printf("%dnn", ptr);
(gdb) print ptr
$1 = (int) 0xfffff
```

PDB for Python

- PDB was self consciously designed to work very much like GDB
- You **import pdb** and then **pdb.run("expression")**
- You are then dropped into a very similar environment to GDB
- Important differences
  - No "**run**" statement, always "**continue**"
  - If a function isn't in global scope, you have to remember it's scoping unit for setting breakpoints
    - **break my_import.my_function**

PDB for Python

- Important differences
  - You always have to specify a filename when specifying a line number for a breakpoint
    - **break my_function.py:4**
  - Accessing array elements uses square brackets
  - There is a "**pp**" function for pretty printing python objects

GDB cheatsheet

- **break {line_number}** or **break {function_name} {if condition} -** Set breakpoint {conditional breakpoint}
- **bt** - show stackframes
- **clear {line_number}** or **clear {function_name}** - Clear breakpoint
- **continue** - continue execution of paused code
- **delete {breakpoint_number}** - delete a breakpoint by number
- **down** - go down a stack level
- **info {command}** - many bits of information. Most common **info locals**
- **print {variable}** - print a variable. Can subscript arrays
- **run** - execute code from start
- **stack {frame number}** - go to numbered stack frame
- **up** - go up a stack level

Valgrind

Valgrind

- Languages like Python (and famously Java) have **garbage collection** to handle memory
  - Memory cannot be lost permanently, the core language keeps track of all memory and deletes it when your code isn't using it any more
  - Can still lose memory by keeping a reference to memory that you're no longer using
- C, C++ and Fortran do not have garbage collection
  - Memory can get lost permanently

Memory Leak

```c
#include <stdlib.h>

int main(int argc, char** argv)
{
  int i;
  int *ptr;
  //After each iteration of this loop the memory is lost permanently
  //because no reference to that memory exists anywhere
  for (i = 0; i< 10; i++){
    ptr = (int*) malloc(100 * sizeof(int));
  }
}
```

```fortran
PROGRAM main

  INTEGER :: i
  INTEGER, DIMENSION(:), POINTER :: ptr

  !After each iteration memory is lost permanently
  !Because there is no longer any reference to the previous allocated version
  DO i = 1, 10
    ALLOCATE(ptr(1:100))
  END DO

END PROGRAM main
```

Valgrind

- Valgrind is a collection of tools for various purposes, from debugging to profiling
- Original and most used tool is “memcheck”
- It detects memory leaks using a custom malloc library
- Simple to use in theory
  - **valgrind {program_name}**
- Can add additional options, such as
  - **valgrind --leakcheck = yes {program_name}**

Output

```
csb@computenode:~$ valgrind --leak-check=yes ./leaky
==16761== Memcheck, a memory error detector
==16761== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==16761== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==16761== Command: ./leaky
==16761==
==16761==
==16761== HEAP SUMMARY:
==16761==     in use at exit: 4,000 bytes in 10 blocks
==16761==   total heap usage: 10 allocs, 0 frees, 4,000 bytes allocated
==16761==
==16761== 4,000 bytes in 10 blocks are definitely lost in loss record 1 of 1
==16761==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==16761==    by 0x4005F7: main (test.c:10)
==16761==
==16761== LEAK SUMMARY:
==16761==    definitely lost: 4,000 bytes in 10 blocks
==16761==    indirectly lost: 0 bytes in 0 blocks
==16761==      possibly lost: 0 bytes in 0 blocks
==16761==    still reachable: 0 bytes in 0 blocks
==16761==         suppressed: 0 bytes in 0 blocks
==16761==
==16761== For counts of detected and suppressed errors, rerun with: -v
==16761== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Valgrind

- Definitely lost
  - Memory has been allocated and all references to it have been definitely lost
- Indirectly lost
  - Only possible if some memory is definitely lost
  - The memory that has been definitely lost contained a pointer to other memory that has no other pointers to it
  - If the definitely lost memory is prevented from being lost, the indirectly lost memory will be found automatically

Valgrind

- Possibly lost
  - Valgrind can no longer find a remaining pointer to memory, but “unusual” pointer use could mean that this isn’t a memory leak. See the Valgrind user manual for “unusual” here
- Still reachable
  - Memory not deallocated when program finished, but there were still active pointers to it
  - Probably not a problem unless you intend your program to be a daemon/TSR/System Tray program

Valgrind

- Suppressed
  - You can suppress warnings and they will appear here
  - Prevent you forgetting that suppressed warnings are happening

Output

```
csb@computenode:~$ valgrind --leak-check=yes ./leaky
==16761== Memcheck, a memory error detector
==16761== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==16761== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==16761== Command: ./leaky
==16761==
==16761==
==16761== HEAP SUMMARY:
==16761==     in use at exit: 4,000 bytes in 10 blocks
==16761==   total heap usage: 10 allocs, 0 frees, 4,000 bytes allocated
==16761==
==16761== 4,000 bytes in 10 blocks are definitely lost in loss record 1 of 1
==16761==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==16761==    by 0x4005F7: main (test.c:10)
==16761==
==16761== LEAK SUMMARY:
==16761==    definitely lost: 4,000 bytes in 10 blocks
==16761==    indirectly lost: 0 bytes in 0 blocks
==16761==      possibly lost: 0 bytes in 0 blocks
==16761==    still reachable: 0 bytes in 0 blocks
==16761==         suppressed: 0 bytes in 0 blocks
==16761==
==16761== For counts of detected and suppressed errors, rerun with: -v
==16761== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Line 10

```c
#include <stdlib.h>

int main(int argc, char** argv)
{
  int i;
  int *ptr;
  //After each iteration of this loop the memory is lost permanently
  //because no reference to that memory exists anywhere
  for (i = 0; i< 10; i++){
    ptr = (int*) malloc(100 * sizeof(int));
  }
}
```

- Reports the lines where memory was first allocated
- Not the line where the memory is lost
- Valgrind can’t tell you that

Other Valgrind errors

- Valgrind memcheck is a general memory tool. It can also find
  - Uninitialised values
  - Writing outside the bounds of an array (even if it doesn’t cause a segfault)
  - Reading from memory that has already been deallocated/freed

Uninitialised Value

```c
#include <stdio.h>

int main(int argc, char** argv)
{
  int a;
  if(a) {return 1;}
  return 0;
}
```

```
==19406== Conditional jump or move depends on uninitialised value(s)
==19406==    at 0x4004E5: main (in /home/csb/a.out)
```

Writing outside of array

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
  int *a;
  a=(int*)malloc(10*sizeof(int));
  a[10] = 1;
}
```

```
==19513== Invalid write of size 4
==19513==    at 0x40054B: main (test.c:8)
==19513==  Address 0x5204068 is 0 bytes after a block of size 40 alloc'd
==19513==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==19513==    by 0x40053E: main (test.c:7)
==19513==
==19513==
==19513== HEAP SUMMARY:
==19513==     in use at exit: 40 bytes in 1 blocks
==19513==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
```

Writing to already freed memory

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
  int *a;
  a=(int*)malloc(10*sizeof(int));
  free(a);
  a[0] = 1;
}
```

```
==19544== Invalid write of size 4
==19544==    at 0x400593: main (test.c:9)
==19544==  Address 0x5204040 is 0 bytes inside a block of size 40 free'd
==19544==    at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==19544==    by 0x40058E: main (test.c:8)
==19544==  Block was alloc'd at
==19544==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==19544==    by 0x40057E: main (test.c:7)
```

Profilers and Optimisation

Optimisation

- “We should forget about small efficiencies,say about 97% of the time: premature optimization is the root of all evil”

  - Donald Knuth

- One of the most misused quotes in computer programming

- “A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified.”

Profiler

- Profilers tell you how much of the code is spent in a given routine either

    1. By time spent in the routine

    2. By number of calls to a routine

- Look in notes for details on

    - Valgrind Callgrind

    - gprof

    - OS X activity monitor

Overheads

- Branches
  - Modern computers operate using “pipelines”
  - They have to keep instructions and data flowing into the pipeline
  - “If” statements present a problem
  - Computer doesn’t know which set of instructions to put into the pipeline
  - Branch predictors

Overheads

- Branches
  - CPU tries to predict which branch to follow
  - If it predicts wrong then it has to flush the pipeline and refill it
  - Called a “branch miss”
  - Can be found through tools such as “valgrind –tool=cachegrind –branchsim=yes” or perf

Overheads

- Function calls
  - Building the stack frames takes time, and unwinding them at the end also takes time
  - If a compiler can then it changes function calls into inline code instead (called inlining)
  - One of the reasons for debugging with optimisation off. The routine that you're trying to debug might not actually exist in the final code
  - If you have a lot of functions you might get some speedup from manually inlining some of them

Overheads

- Cache, RAM and Disk
  - The fastest level of memory is cache memory, built onto the CPU (nowadays)
    - Only a few MB
  - More data has to be got from RAM (main memory)
    - Much slower than cache
    - Once data has been used once, it goes into cache memory
    - So does some data near the data that you want

Overheads

- Cache, RAM and Disk
  - When the cache runs out of space, the least recently used data is “evicted” to main memory
  - Want to use data in such a way that it is operated on completely before it leaves cache
    - Don’t work on data and then go back to it later
- Because nearby data goes into cache along with the bit that you want, there’s a preferred way of accessing arrays to maximise use of cache

Overheads

- Cache, RAM and Disk
  - In Fortran, want to access data so that first index of an array varies fastest
  - In C want to access so that the last index of an array varies fastest
  - In 1D those are the same thing, but with multidimensional arrays, it really matters

Overheads

- Cache, RAM and Disk
  - Disk is orders of magnitude slower than RAM
  - If your code has to write a lot of data to disk that will slow it down
- Few tricks
  - Don’t keep reopening a file if you can avoid it
  - Don’t seek within a file if you can avoid it
- Ultimately, can’t make it fast. Minimize output to disk

Testing frameworks

Testing

- We've already encountered the idea of testing
- Just like debugging, there are tools to make it easier
- These take the form of "**testing frameworks**"
- These provide tools to help with a lot of the grunt work

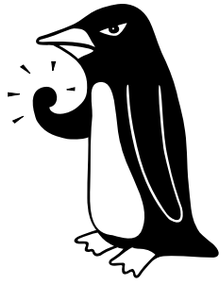
Testing

- Comparing numbers to known precision
- Outputting which files and which routines are passing or failing tests
- Allow some reuse of testing frameworks between codes
- Suitable frameworks for most languages, see notes

Continuous Integration Testing

- Continuous integration is a way of combining the efforts of multiple developers
- A part of it is automated testing of developers contributions whenever they contribute
  - Continuous integration testing
- Quite easy to do now thanks to Github with Travis-CI

Debugging and Testing tools



“The Angry Penguin“, used under creative commons licence from Swantje Hess and Jannis Pohlmann.

Warwick RSE

12/12/2017