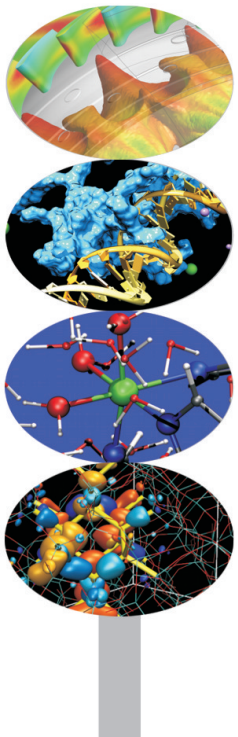
CINECA SCAI
SuperComputing Applications and Innovation

SAPIENZA
UNIVERSITÀ DI ROMA

# Programmazione Avanzata


## Vittorio Ruggiero

(v.ruggiero@cineca.it)

Roma, Maggio 2015

CINECA

# Outline



Debugging
gdb

# My program fails!



- Erroneous program results.
- Execution deadlock.
- Run-time crashes.

# Assumption 1



Question: The same code still worked last week.

- Possible problem: An application runs in an environment with many components (OS, dynamic libraries, network, disks, ..). If any one has changed, this can impact the application.
- Fix: ask the helpdesk if anything changed since last week.
- Alternative problem: Often, the code is not exactly the same, only the user assumed that the change was not important.
- Fix: a version control system (cvs, svn, git, ..) will help you check that the code really was the same as last week or to see what changes were made.

# Assumption 2



Question: The same program works for my colleague.

- Possible problem: do you really use the same executable?
- Check: use

```
which [command]
```

to see the location.

- Possible problem: dynamic executables load libraries at runtime. These are loaded from directories in your $*LD_LIBRARY_PATH*. Maybe you use different versions of libraries without knowing it?
- Check: use

```
ldd [program]
```

to check which libraries are used.

# Assumption 3



Question: the program works fine on my own system, so something is wrong with yours

```
program helloworld
implicit none
print*,'Hello World!'
return
end program
```

- works fine with GNU and IBM compilers, but won't compile with the Intel compiler. Why?
- return statement is not allowed in Fortran mainprogram

```
> ifort return.f90
return.f90(5): error #6353: A RETURN statement is invalid in
the main program.
```

- Solution: check your program to see if it follows the language standard.

# Assumption 4



Question: My program crashed. What did I do wrong?

- Answer: Your program depends on other libraries, the compiler, the OS, the network, etc. System libraries and compilers have bugs and hardware can fail, so it might not be your program's fault.

```
CALL MPI_Barrier(MPI_COMM_SELF, IERR)
```

- This call would segfault in some version of IBM MPI, although it is a correct MPI call.
- Solutions: read Changelogs or KnownBugs. Isolate the problem and send it to the helpdesk. Ask if it could be a known bug.
- However, 99% of the problems are related to the application.

# Assumption 5



Question: It works only sometimes.

- Problem: Probably a race condition (bugs that cause undefined behaviour, depending on timing differences)

# The ideal debugging process



- Find origins.
  - Identify test case(s) that reliably show existence of fault (when possible). Duplicate the bug.
- Isolate the origins of infection.
  - Correlate incorrect behaviour with program logic/code error.
- Correct.
  - Fixing the error, not just a symptom of it.
- Verify.
  - Where there is one bug, there is likely to be another.
  - The probability of the fix being correct is not 100 percent.
  - The probability of the fix being correct drops as the size of the program increases.
  - Beware of the possibility that an error correction creates a new error.

# Bugs that can't be duplicated



- Dangling pointers.
- Initializations errors.
- Poorly syinchronized threads.
- Broken hardware.

# Isolate the origins of infection



- Divide and conqueror.
- Change one thing at time.
- Determine what you changed since the last time it worked.
- Write down what you did, in what order, and what happened as a result.
- Correlate the events.

# What is a Debuggers?



*Debuggers are a software tools that help determine why program does not behave correctly. They aid a programmer in understanding a program and then finding the cause of the discrepancy. The programmer can then repair the defect and so allow the program to work according to its original intent. A debugger is a tool that controls the application being debugged so as to allow the programmer to follow the flow of program execution and , at any desired point, stop the program and inspect the state of the program to verify its correctness.*

**"How debuggers works Algorithms, data,structure, and Architecture"**

Jonathan B. Rosemberg

# Why use a Debugger?



- No need for precognition of what the error might be.
- Flexible.
  - Allows for ”live” error checking (no need to re−write and re−compile when you realize a certain type of error may be occuring).
- Dynamic.
  - Execution Control Stop execution on specified conditions: breakpoints
  - Interpretation Step-wise execution code
  - State Inspection Observe value of variables and stack
  - State Change Change the state of the stopped program.

# Why people don't use debuggers?



- With simple errors, may not want to bother with starting up the debugger environment.
  - Obvious error.
  - Simple to check using prints/asserts.
- Hard-to-use debugger environment.
- Error occurs in optimized code.
- Changes execution of program
  (error doesn't occur while running debugger).

# Why don't use print?



- Cluttered code.
- Cluttered output.
- Slowdown.
- Loss of data.
- Time consuming.
- And can be misleading.
  - Moves things around in memory, changes execution timing, etc.
  - Common for bugs to hide when print statements are added, and reappear when they're removed.

# Outline



Debugging
gdb

# What is gdb?



- The GNU Project debugger, is an open-source debugger.
- Released by GNU General Public License (GPL).
- Runs on many Unix-like systems.
- Was first written by Richard Stallmann in 1986 as part of his GNU System.
- Its text-based user interface can be used to debug programs written in C, C++, Pascal, Fortran, and several other languages, including the assembly language for every micro-processor that GNU supports.
- `www.gnu.org/software/gdb`

# prime-numbers program



- Print a list of all primes which are less than or equal to the user-supplied upper bound *UpperBound*.
- See if *J* divides $K \leq$ *UpperBound*, for all values *J* which are
  - themselves prime (no need to try *J* if it is nonprime)
  - less than or equal to sqrt(K) (if *K* has a divisor larger than this square root, it must also have a smaller one, so no need to check for larger ones).
- *Prime[I]* will be 1 if *I* is prime, 0 otherwise.

# Main.c



```
#include <stdio.h>
#define MaxPrimes 50
int Prime[MaxPrimes],UpperBound;
int main()
{
    int N;
    printf("enter upper bound\n");
    scanf("%d",&UpperBound);
    Prime[2] = 1;
    for (N = 3; N <= UpperBound; N += 2)
        CheckPrime(N);
    if (Prime[N]) printf("%d is a prime\n",N);
    return 0;
}
```

# CheckPrime.c



```
#define MaxPrimes 50
extern int Prime[MaxPrimes];
void CheckPrime(int K)
{
    int J; J=2;
    while (1)  {
        if (Prime[J] == 1)
            if (K % J == 0)  {
                Prime[K] = 0;
                return;
            }
            J++;
    }
    Prime[K] = 1;
}
```

# Compilation and run



```
<~>gcc Main.c CheckPrime.c  -o trova_primi
```

```
<~> ./trova_primi
```

```
enter upper bound
20
Segmentation fault
```

# Compilation options for gdb



- You will need to compile your program with the appropriate flag to enable generation of symbolic debug information, the -g option is used for this.
- Don’t compile your program with optimization flags while you are debugging it.
  Compiler optimizations can "rewrite" your program and produce machine code that doesn’t necessarily match your source code.
  Compiler optimizations may lead to:
  - Misleading debugger behaviour.
    - Some variables you declared may not exist at all
    - some statements may execute in different places because they were moved out of loops
  - Obscure the bug.

# Starting gdb



```
<~>gcc Main.c CheckPrime.c -g  -o trova_primi
```

```
<~>gdb trova_primi
```

```
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-23.el5_5.1)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
```

```
(gdb)
```

# gdb: Basic commands



- run (r): start debugged program.
- kill (k): kill the child process in which program is running under gdb.
- where, backtrace (bt): print a backtrace of entire stack.
- quit(q): exit gdb.

- break (b) : set breakpoint at specified line or function

- print (p) expr: print value of expression expr
- display expr: print value of expression expr each time the program stops.

# gdb: Basic commands



- continue: continue program being debugged, after signal or breakpoint.
- next: step program, proceeding through subroutine calls.
- step: step program until it reaches a different source line

- help (h): print list of commands.
- she: execute the rest of the line as a shell command.
- list (l) linenum: print lines centered around line number lineum in the current source file.

# prime-number



```
(gdb) r
```

```
Starting program: trova_primi
enter upper bound

20

Program received signal SIGSEGV, Segmentation fault.
0xd03733d4 in number () from /usr/lib/libc.a(shr.o)
```

```
(gdb) where
```

```
#0 0x0000003faa258e8d in _IO_vfscanf_internal () from /lib64/libc.so.6
#1 0x0000003faa25ee7c in scanf () from /lib64/libc.so.6
#2 0x000000000040054f in main () at Main.c:8
```

# prime-number finding program



```
(gdb) list Main.c:8
```

# prime-number finding program



```
(gdb) list Main.c:8
```

```
3    int Prime[MaxPrimes],UpperBound;
5      main()
6      {   int N;
7          printf("enter upper bound\n");
8          scanf("%d",UpperBound);
9          Prime[2] = 1;
10         for (N = 3; N <= UpperBound; N += 2)
11            CheckPrime(N);
12            if (Prime[N]) printf("%d is a prime\n",N);
```

# Main.c :new version



```
1  #include <stdio.h>
2  #define MaxPrimes 50
3  int Prime[MaxPrimes],UpperBound;
4  int main()
5  {
6        int N;
7        printf("enter upper bound\n");
8        scanf("%d",  UpperBound);
9        Prime[2] = 1;
10       for (N = 3; N <= UpperBound; N += 2)
11          CheckPrime(N);
12          if (Prime[N]) printf("%d is a prime\n",N);
13       return 0;
14 }
```

# Main.c :new version



```
1  #include <stdio.h>
2  #define MaxPrimes 50
3  int Prime[MaxPrimes],UpperBound;
4  int main()
5  {
6        int N;
7        printf("enter upper bound\n");
8        scanf("%d", &UpperBound);
9        Prime[2] = 1;
10       for (N = 3; N <= UpperBound; N += 2)
11          CheckPrime(N);
12          if (Prime[N]) printf("%d is a prime\n",N);
13       return 0;
14 }
```

# Main.c :new version



```
1  #include <stdio.h>
2  #define MaxPrimes 50
3  int Prime[MaxPrimes],UpperBound;
4  int main()
5  {
6        int N;
7        printf("enter upper bound\n");
8        scanf("%d", &UpperBound);
9        Prime[2] = 1;
10       for (N = 3; N <= UpperBound; N += 2)
11          CheckPrime(N);
12          if (Prime[N]) printf("%d is a prime\n",N);
13       return 0;
14 }
```

In other shell COMPILATION

# prime-number



```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound

20

Program received signal SIGSEGV, Segmentation fault.
0x0000000004005bb in CheckPrime (K=0x3) at CheckPrime.c:7
7                 if (Prime[J] == 1)
```

# prime-number finding program



```
(gdb) p J
```

```
$1 = 1008
```

```
(gdb) l CheckPrime.c:7
```

```
2       extern int Prime[MaxPrimes];
3         CheckPrime(int K)
4         {
5             int J; J=2;
6             while (1)  {
7                if (Prime[J] == 1)
8                   if (K % J == 0)  {
9                      Prime[K] = 0;
10                     return;
11                  }
```

# CheckPrime.c:new version



```
1  #define MaxPrimes 50
2  extern int Prime[MaxPrimes];
3  void CheckPrime(int K)
4     {
5       int J; J = 2;
6        while (1){
7           if (Prime[J] == 1)
8              if (K % J == 0)  {
9                 Prime[K] = 0;
10                return;
11             }
12       J++;
13       }
14       Prime[K] = 1;
15    }
```

# CheckPrime.c:new version



```
1   #define MaxPrimes 50
2   extern int Prime[MaxPrimes];
3   void CheckPrime(int K)
4      {
5        int J;
6            for (J = 2; J*J <= K; J++)
7            if (Prime[J] == 1)
8               if (K % J == 0)  {
9                  Prime[K] = 0;
10                 return;
11              }
12
13
14        Prime[K] = 1;
15     }
```

# prime-number finding program



```
(gdb) kill
```

Kill the program being debugged? (y or n) y

```
(gdb) she gcc -g Main.c CheckPrime.c -o triva_primi
```

```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound

20

Program exited normally.
```

# gdb commands



**(gdb) help break**

```
Set breakpoint at specified line or function.
break [LOCATION] [thread THREADNUM] [if CONDITION]
LOCATION may be a line number, function name, or "*" and an address.
If a line number is specified, break at start of code for that line.
If a function is specified, break at start of code for that function.
If an address is specified, break at that exact address.
.........
Multiple breakpoints at one place are permitted,
and useful if conditional.
.........
```

**(gdb) help display**

```
Print value of expression EXP each time the program stops.
.........
Use "undisplay" to cancel display requests previously made.
```

# gdb commands



**(gdb) help step**

```
Step program until it reaches a different source line.
Argument N means do this N times
(or till program stops for another reason).
```

**(gdb) help next**

```
Step program, proceeding through subroutine calls.
Like the "step" command as long as subroutine calls do not happen;
when they do, the call is treated as one instruction.
Argument N means do this N times
(or till program stops for another reason).
```

**(gdb) break Main.c:1**

```
Breakpoint 1 at 0x8048414: file Main.c, line 1.
```

# prime-number finding program



```
(gdb) r

Starting program: trova_primi
Breakpoint 1, main () at Main.c:7
9               printf("enter upper bound\n");

(gdb) next

10               scanf("%d",&UpperBound);

(gdb) next
20
11               Prime[2] = 1;

(gdb) next

12              for (N = 3; N <= UpperBound; N += 2)

(gdb) next

14                 CheckPrime(N);
```

# prime-number finding program



```
(gdb) display N
```

```
1: N = 3
```

```
(gdb) step
```

```
CheckPrime (K=3) at CheckPrime.c:6
6             for (J = 2; J*J <= K; J++)
```

```
(gdb) next
```

```
12            Prime[K] = 1;
```

```
(gdb) next
```

```
13        }
```

```
(gdb)  n

10              for (N = 3; N <= UpperBound; N += 2)
1: N = 3
}

(gdb) n

11                  CheckPrime(N);
1: N = 5

(gdb) n

10              for (N = 3; N <= UpperBound; N += 2)
1: N = 5

(gdb) n

11                  CheckPrime(N);
1: N = 7
```

# prime-number finding program



```
(gdb) l  Main.c:10
```

```
5           main()
6           {   int N;
7               printf("enter upper bound\n");
8               scanf("%d",&UpperBound);
9               Prime[2] = 1;
10              for (N = 3; N <= UpperBound; N += 2)
11                  CheckPrime(N);
12                  if (Prime[N]) printf("%d is a prime\n",N);
13              return 0;
14          }
```

# Main.c :new version



```
1  #include <stdio.h>
2  #define MaxPrimes 50
3   int Prime[MaxPrimes],
4   UpperBound;
5     main()
6     {   int N;
7        printf("enter upper bound\n");
8        scanf("%d",&UpperBound);
9        Prime[2] = 1;
10       for (N = 3; N <= UpperBound; N += 2)
11          CheckPrime(N);
12          if (Prime[N]) printf("%d is a prime\n",N);
13
14       return 0;
15    }
```

# Main.c :new version



```
1  #include <stdio.h>
2  #define MaxPrimes 50
3   int Prime[MaxPrimes],
4   UpperBound;
5     main()
6     {   int N;
7         printf("enter upper bound\n");
8         scanf("%d",&UpperBound);
9         Prime[2] = 1;
10        for (N = 3; N <= UpperBound; N += 2){
11           CheckPrime(N);
12           if (Prime[N]) printf("%d is a prime\n",N);
13      }
14        return 0;
15    }
```

# prime-number finding program



```
(gdb) kill
```

Kill the program being debugged? (y or n) y

```
(gdb) she gcc -g Main.c CheckPrime.c -o trova_primi
```

```
(gdb) d
```

Delete all breakpoints? (y or n) y

# prime-number finding program



```
(gdb) r
```

```
Starting program: trova_primi
enter upper bound

20

3 is a prime
5 is a prime
7 is a prime
11 is a prime
13 is a prime
17 is a prime
19 is a prime

Program exited normally.
```

# prime-number finding program



```
(gdb) list Main.c:6
```

```
1       #include <stdio.h>
2       #define MaxPrimes 50
3        int Prime[MaxPrimes],
4        UpperBound;
5          main()
6          {   int N;
7              printf("enter upper bound\n");
8              scanf("%d",&UpperBound);
9              Prime[2] = 1;
10             for (N = 3; N <= UpperBound; N += 2){
```

# prime-number finding program



```
(gdb) break Main.c:8
```

```
Breakpoint 1 at 0x10000388: file Main.c, line 8.
```

```
(gdb)run
```

```
Starting program: trova_primi
enter upper bound
Breakpoint 1, main () at /home/guest/Main.c:8
8               scanf("%d",&UpperBound);
```

```
(gdb) next
```

```
20

9               Prime[2] = 1;
```

# prime-number finding program



```
(gdb) set UpperBound=40
(gdb) continue
```

```
Continuing.
3 is a prime
5 is a prime
7 is a prime
11 is a prime
13 is a prime
17 is a prime
19 is a prime
23 is a prime
29 is a prime
31 is a prime
37 is a prime

Program exited normally.
```

# Debugging post mortem



- When a program exits abnormally the operating system can write out core file, which contains the memory state of the program at the time it crashed.
- Combined with information from the symbol table produced by -g the core file can be used fo find the line where program stopped, and the values of its variables at that point.
- Some systems are configured to not write core file by default, since the files can be large and rapidly fill up the available hard disk space on a system.
- In the GNU Bash shell the command ulimit -c control the maximum size of the core files. If the size limit is set to zero, no core files are produced.

```
ulimit –c unlimited
gdb exe_file core.pid
```

# Graphical Debuggers



- gdb -tui or gdbtui (text user interface)
- ddd (data dysplay debugger) is a graphical front-end for command-line debuggers.
- allinea ddt (Distributed Debugging Tool) is a comprehensive graphical debugger for scalar, multi-threaded and large-scale parallel applications that are written in C, C++ and Fortran.
- Rouge Wave Totalview
- Etc.

# Esercitazione:debugging



```
https://hpc-forge.cineca.it/files/CoursesDev/public/2014/
Introduction_to_HPC_Scientific_Programming:_tools_and_techniques/
Rome/Debug_esercizi.tar

tar xvf Debug_esercizi.tar
```

- TEST1: semplice bug per familiarizzare con i comandi
- TEST2: altro semplice bug per familiarizzare con i comandi
- TEST3: calcolo di un elemento della successione di Fibonacci.
  - Input: un numero che rappresenta la posizione dell'elemento della successione di cui si vuole il valore
  - Output: stampa a schermo del valore dell'elemento richiesto
- TEST4: sorting
  - Input: un intero che rappresenta il numero di interi che si vogliono ordinare ed i relativi valori (in C i valori da ordinare)
  - Output: i valori in ordine crescente
- TEST5: Crivello di Eratostene (ricerca dei numeri primi)
  - Input: un numero intero che rappresenta il limite di ricerca
  - Output: L'elenco dei numeri primi inferiori e uguali a quello fornito come limite