Latinamerican School for Computational Materials Science.

from source code to running application..

Stefano Cozzini

CNR-INFM DEMOCRITOS, Trieste

Santiago, Chile - Enero, 2009

AIM

- examine how to go from a piece of source code to a running application
- Discuss:
  - which programming languages are available (briefly)
  - what is a compiler and how it works
  - different stages involved in the process of compiling
  - how a program actually runs
- Motivation: the interaction between application and the system is often poorly understood but a greater knowledge can be helpful to efficient scientific software development

Programming languages

- choice of language for a given task is often a thorny issue.
- For us: a programming language is only a tool for writing scientific code.
- The computer language that you use will hopefully be the one that best facilitates this task.
- Many differences between high level language which may be viewed as advantages or disadvantages depending on the task you are trying to solve.

Interpreted languages

- An interpreter is a program which itself executes other programs. Examples:
  - Basic, JavaScript, Perl and awk
- Advantages:
  - it can be quicker to run the code under the interpreter than compile and run it with a compiler.
  - code is easier to debug (interpreter will analyze each statement in the code each time it is executed.)
- Where to use interpreted languages:
  - for small applications prototyping and testing of code when an edit-interpret-debug cycle can often be much quicker than an edit- compile-run-debug cycle.
- NOT a good idea to perform numerically intensive calculations using interpreted languages.

compiled languages

- Needed to perform numerically intensive calculations
- Run Time >> Compiling/debugging time
- Examples:
  - Fortran
  - C
  - C++
  - Java
  - Others ?

Fortran

- The main language within the scientific community
- It is likely to hold this position for a long time.
- Why ?
  - Some issues within C that make the language inherently more difficult to compile and produce good optimization (mainly dynamical d-referencing of pointers).
  - Tons of libraries written in Fortran
  - Tons of computational codes written in F77
  - laziness of users (especially Sissa users)

C for scientific computing

- C works well in many domains: graphics, I/O, O.S. world
- C issues such as pointer aliasing makes difficult for compilers to produce highly optimised code.
- Other limits in numerical computation:
  - 1. complex arithmetic is missing
  - 2. F90 array notation is missing
  - 3. Tons of numerical software is written in F77/90
- Interested in C ? check out http://www.accu.org.

C++

- C++ developed by Bjarne Stroustrup in 1983.
- C++ was initially an extension to C to incorporate full object-oriented(OO)programming techniques.
- C can be regarded as a subset of C++, so a C++ compiler will (hopefully) be able to compile a C code to achieve the same performance.
- The main design aim of C++ is to design and build large applications using OO techniques.
- However performance can drop if you use OO without care..
- Interested in OO ? Use this language, calling a Fortran library for the more intensive work.

Why Compilers?

- Compiler
  - A program that translates from 1 language to another
  - It must preserve semantics of the source
  - It should create an efficient version of the target language
- In the beginning, there was machine language
  - Ugly – writing code, debugging
  - Then came textual assembly – still used on some devices..
  - High-level languages – Fortran, Pascal, C, C++
  - Machine structures became too complex and software management too difficult to continue with low-level languages

Knowing your compiler

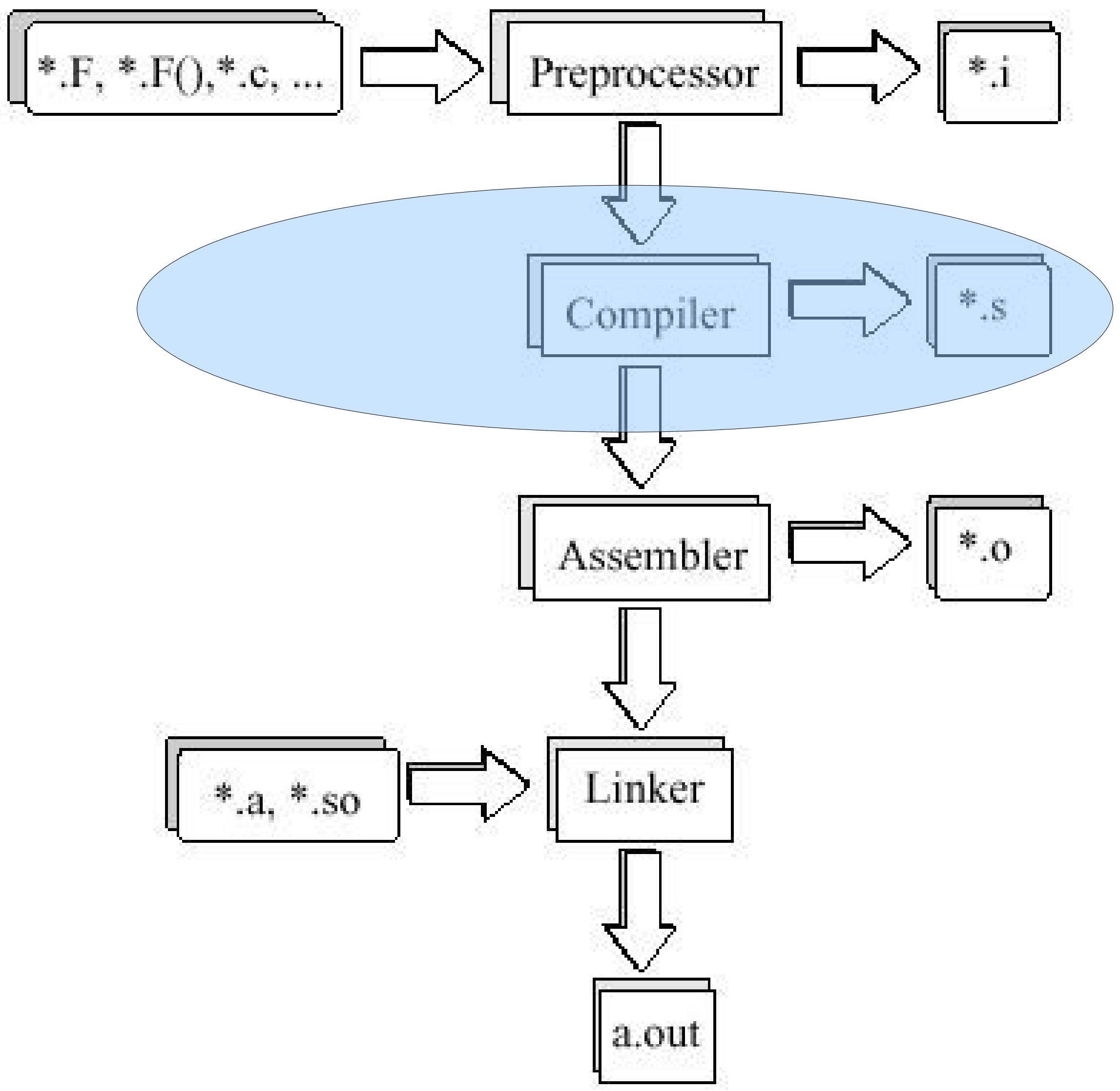
- Calling a compiler you invoke a driver program that hides the different compilation stages.
- You can use compiler flags to show all the in-between stages and/or output intermediate results from any of these stages.
- These flags tends to be compiler dependent

```
Example: to show all the gory details
of the intermediate stages for gcc
cc -v -o myprogram myprogram.c
```

Proposed exercise:

- know your preferred compiler
- Have a look at the man page to identify flags to:
  - check syntax
  - change the levels of optimisation
  - identify the compilation stages (trough intermediate output)

compilation steps (1)



A word of caution...

- The terminology here might be slightly confusing as we are using the term compiler in two ways
  - the compilation process is what will take source code and produce executable machine code.
  - In the diagram above we use "compiler" to encompass just a step of the procedure:
  - This is composed by following four stages:
    - 1. lexical analysis;
    - 2. syntax and
    - 3.semantic analysis;
    - 4. Intermediate Level Code (ILC) generation.

Using a preprocessor

- C files are automatically pre-processed before they are passed to the front end of the compiler.
- You can output preprocessed files (*.i suffix) using -P -E
- To preprocess Fortran files use *.F (fixed) or *.F90 (free) extensions for the source files.
- Preprocessing can be done explicitly by cpp, or fpp or using embedded preprocessor that comes with the F90 compiler itself.

Using a preprocessor (2)

- Typically preprocessors perform mainly text based manipulations.
- Preprocessor commands (known as "pragmas" in C) always begin with #, (#include, #define,#ifdef)
- It is possible to use the same preprocessor for Fortran programs and the # must be located in the first character position.
- These commands instruct the preprocessor to:
  - include external (header) files conditionally
  - enable source code compilation
  - perform textual substitution (expansion on Macro/embedding constant)

Using cpp preprocessor with Fortran code:

- Cpp replace C comments (/* ...*/) by single spaces, backslash-newline combinations are deleted
- You must bear this in mind when using cpp to process Fortran programs to avoid some strange behaviour ....
- Try to use fpp .. (man fpp)

conditional compilation (an example)

- to comment out sections of code, possibly machine specific:

```
#ifdef X86_64
 /* Opteron specific code */
 #endif
```

- The code will be compiled: at compile time by adding #define X86_64 at the top of the relevant source file or in a generic header file from the command line:

```
>cc -DX86_64  ...
```

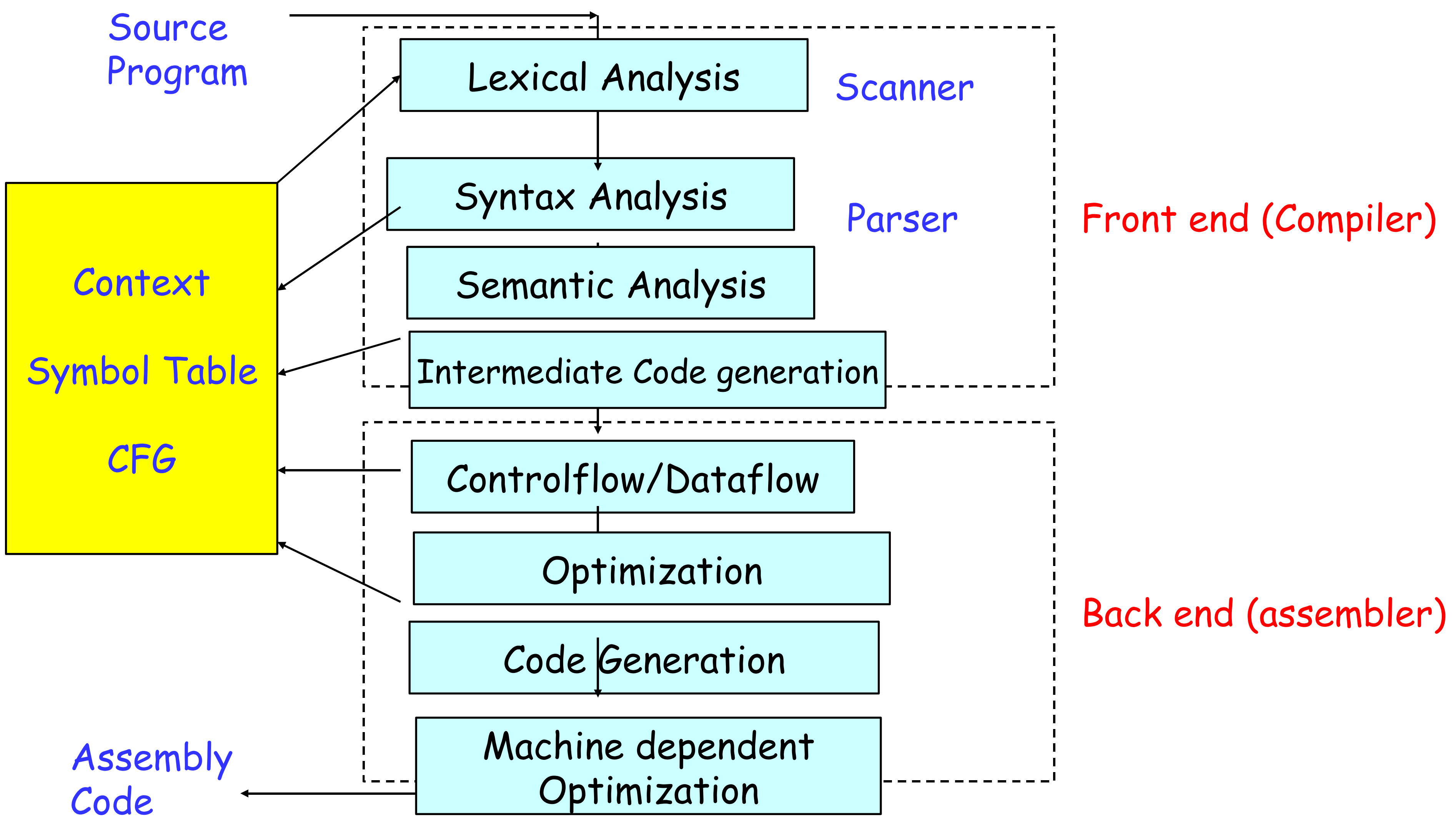
Preprocessing: final considerations

- Great tools: very useful (portability)
- However: for the sake of clarity, do not overuse #ifdef.
- TIP: If you find that you have a lot of different options it might be better to separate the source code into separate files and then use **make** to perform the required compilation.

Compiler stages

- The front end stages:
  - several front end (each for any language)
  - each of one produce the ILC (portable)
- the back-end stage
  - produce machine-specific assembler code then to re-locatable object code.
- the linker stage:
  - link together all the pieces to produce the executable

a picture to help



the frontend stages: analysis

- done on a file-by-file basis as independent compilation
- can perform its tasks in one pass or multiple passes
- process is complex and specialised.
- you have no access to this section but.. the tasks it attempts to perform may be controllable by flags

The front-end: actions

- first: The parser, syntax and semantic analysis removes unnecessary white spaces and any remaining comments.
- second: The source code is split into "tokens":
- third: syntax checker makes sure that each is a valid construct (The majority of the errors that are detectable by the compiler are caught here)
- fourth: The code generator produces the ILC output.
- fifth: the The code optimiser attempts to optimise the ILC
- check man pages to see which flags are available and what they do:
  - enforcing strict syntax checking
  - looking for un-used variables etc..

Optimization

- How to make the code go faster
- Classical optimizations
  - Dead code elimination – remove useless code
  - Common subexpression elimination – recomputing the same thing multiple times
- Machine independent (classical)
  - Focus of this class
  - Useful for almost all architectures
- Machine dependent
  - Depends on processor architecture
  - Memory system, branches, dependences

Optimization (2)

- Compilers do many different transformations to produce fast code
- Some of them could be controlled by flags on the command line
- This is not always straightforward (complex inter-dependencies)
- To increase performance play with flags..
  (learn a lot about that next lecture...)
- Check out results are still correct.

Symbol Table

- produced by compiler and used by the linker to find the information required to build the whole code.
- Like a dictionary that records each identifier or keyword found:
  - the type (variable, array, procedure, . . .)
  - the data type (integer,real, . . .)
  - the run-time address pointer to access more information (like the bounds of an array, . . .
- The symbol table is very important for debugging.
- Debuggers use a more complete symbol table with every variable listed and references to the source lines where they are modified.
- This is generally produced with the -g flag.

The assembler

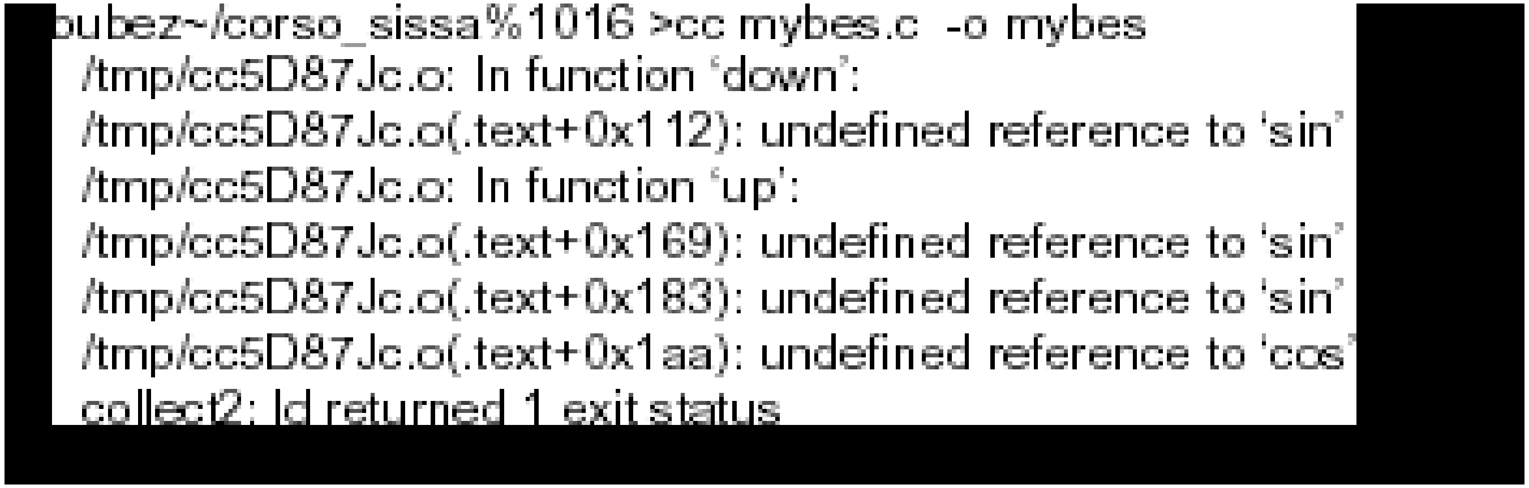
- the assembler creates object files from assembly language source files
- Some specific operations:
  - Anything that cannot be handled by hardware must be done in software:mathematical operations such as inverses, cosines and square roots.
  - Resource conflicts e.g. use of registers, pipelines, etc. must be resolved.
- The assembler code is then converted to relocatable object code by the assembler. (A relocatable object file can be loaded starting at any location in memory)
- It is then added to all the addresses in the object file, so the object file could be loaded into any location in memory by the Unix operating system.

the linker

- All the different object files are finally glued together by the linker.
- to know about it : man ld (very system specific)
- Actions:
  - identifies the main routine as the initial entry point when execution begins.
  - resolves subroutine and identifies function calls by putting in the correct addresses
  - identifies and branch statements and instructions to copy arguments onto the stack.
  - If there are any unresolved symbols it will then try to link in any external libraries which have been specified and default ones from the system.
- The result is an executable file:
  a(ssembler).out(put)

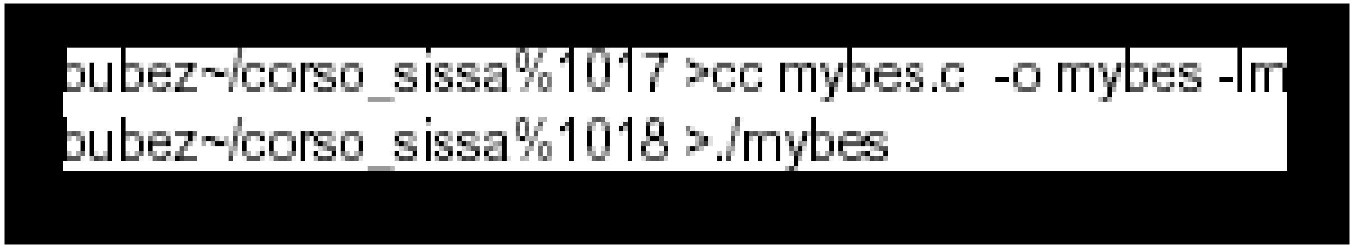
Linker (2)

- Error messages are printed if there are remaining unresolved symbols.



```
bubez~/corso_sissa%1016 >cc mybes.c  -o mybes
 /tmp/cc5D87Jc.o: In function 'down':
 /tmp/cc5D87Jc.o(.text+0x112): undefined reference to 'sin'
 /tmp/cc5D87Jc.o: In function 'up':
 /tmp/cc5D87Jc.o(.text+0x169): undefined reference to 'sin'
 /tmp/cc5D87Jc.o(.text+0x183): undefined reference to 'sin'
 /tmp/cc5D87Jc.o(.text+0x1aa): undefined reference to 'cos'
 collect2: ld returned 1 exit status
```

- Solution: add the maths library at the end of the compilation process:



```
bubez~/corso_sissa%1017 >cc mybes.c  -o mybes -lm
bubez~/corso_sissa%1018 >./mybes
```

Where are the libs ?

- Standard places are searched if you use standard libraries [/usr/lib /usr/local/lib]
- check out the LD_LIBRARY_PATH env variable
- Otherwise: explicitly specify the path to the library and the name of that library:
  -L/Path_to_library -lmpi
- this refers to a library file called:
  - /Path_to_library/libmpi.s (shared) o (bject)
  - --> (dynamic library)
  - /Path_to_library/libmpi.a(rchive) [see ar command]
    --> (static library)

Static vs Dynamic libraries...

- Dynamic libraries the external reference will not be resolved until the code starts running and even then the linking will not take effect until an actual call is made to the routine requiring that library.
- Static libraries the actual code to execute the external routine will be physically copied into your executable. This is the way that linking used to always be done.
- There are advantages/disadvantages to both methods

Static libraries: *.a (from archive command [man ar])

- pro:
  - Libraries are physically copied into the process space which can produce marginally faster code with no dynamic linking overhead and better linking optimisation.
- cons:
  - Wasteful: every user can have their own private copy of library routines linked into executable on disk

Dynamic libraries: have a *.so extension.

- PRO:
  - Libraries are linked in dynamically at run time which means you will have a smaller executable
  - One copy of a library can be used by more than one process if it is running on the same machine => better use of memory resources.
  - The latest version of a library will be linked in at run time or if different systems are available it may be possible to have libraries optimised for that system you are running on dynamically linked.
- CONS:
  - Changes in library locations, version number conflicts or missing libraries can lead to cryptic run time failures.
  - There is a small associated overhead with the dynamic linking of libraries

typical issue with dynamic libraries

- the dynamic linker cannot find the dynamic libraries requested or licenses for them, e.g.:

```
> ./hello
ld.so.1: ./hello: fatal: libmpi.so.1: open failed: No such file or directory
Killed
```

- How to overcome this problem ? use ldd
- ldd lists the dynamic dependencies expected by an executable files or shared objects and the location of where it is expecting to find these libraries.

```
[cozzini@bubez cozzini]$ ldd -v /usr/bin/who
libc.so.6 => /lib/i686/libc.so.6 (0x40022000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Run the program..

- what is an executable ?

  just a regular file that knows how to initialise a new execution context.

- Check with the file command

-
```
[cozzini@bubez cozzini]$ file h_din.x
h_din.x: ELF 32-bit LSB executable, Intel 80386, version 1, dynamically linked
(uses shared libs), not stripped
```

a word about memory: segments

- The shell will use execve to fork a new Unix process. Parts of the binary file will then be mapped directly to memory by the loader. To make this easier the executable is divided into segments some of which are shared by the Unix process:
  - executable magic number
  - executable internals
  - Text segment (process)
  - Data Segment (process)
  - size for bss segment (process)
  - stack (only in the process)
  - heap (only in the process)
-

Stack and heap segment

- stack segment:
  - is allocated at run time and used to store automatic variables and arrays (created when one goes into a subroutine) and keep track of stack frames for procedure calls. It is a temporary piece of scratch space but has a fixed size.
- heap segment:
  - is allocated at run time and used for data declared dynamically such as that created by statements such as Fortran ALLOCATE or C malloc and calloc. It also has a fixed size and the return values of these calls must be tested to ensure that it is not exceeded or errors will occur (probably a segmentation fault, see later).

stack size sometime too small..

- symptome: a code runs well but when you increase data size it gives : segmentation fault
- to check if this is related to stack size :
  - check the size of your stack on the computer (ulimit command)

```
cozzini$ ulimit -s
8192
```

  - check where data are allocated by your program

```
[cozzini@bubez lezioni_sissa]$ nm -f s  a.out | grep array
array1              |08049540|   D  |                  |      |     |
array2              |080497e0|   B  |                  |      |     |
```

trace/strace command

- See the interaction between your program and the Unix kernel
- very useful to understand where and why your program fails due to external problems
- Output very long and complex...

```
[cozzini@bubez lezioni_sissa]$ strace ./a.out
execve("./a.out", ["./a.out"], [/* 37 vars */]) = 0
uname({sys="Linux", node="bubez", ...}) = 0
brk(0)                                  = 0x8049970
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
x40017000
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=40472, ...}) = 0
old_mmap(NULL, 40472, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40018000
close(3)                                = 0
```

Compilers for Linux

- Free/Open Source:
  - GNU http://www.gnu.org/ (Fortran 77, C, C++, ...)
- Commercial:
  - PGI (Fortran 77, Fortran 90, C, C++) http://www.pgroup.com/
  - Intel (Fortran 77/95, C/C++) (individual Linux license free of charge)
  - PathScale (Fortran 77, Fortran 90, C, C++) http://www.pathscale.com (x86_64)
  - NAG http://www.nag.co.uk
  - Lahey http://www.lahey.com/
  - Absoft http://www.absoft.com/
- Almost all allow you a 15 day evaluation license

How to choose a compiler for scientific computing?

- Efficiency
  - Does it produce efficient code?
  - Does it produce correct code?
  - Is it able to exploit the hardware?
- Availability/Cost
  - How much does it differ from the GNU compilers?
- Interoperability
  - Does it operate with other tools/compiler/languages?
- Utilities / Tools
  - Does it have a Debugger/ Profiler / other utilities?
- Diagnostic Capabilities
  - Is it able to detected errors/bugs in programs?
- Documentation/ support /training..

Gnu compiler collection

- The Cross-Platform compiler package
- Supports many OS/CPU combinations
- Already bundled with Linux distributions
- Support for C/C++ good
- Fortran 77 support limited (performance, completeness
- Fortran 95 available from version 4.x (gfortran)
- Debugger, several GUI frontends
- Profiler, GUI frontends
- Many additional, supporting tools available

the PGI suite

- Widespread (for a long time only alternative to GNU)
- Platforms: x86 and x86_64, for Linux, Win32 and Solaris
- Native OpenMP Support
- Good Fortran 77/90/95 support
- GNU Interoperability: can link g77 libraries&
- Advanced Optimizations: IPO and PGO
- PGDBG graphical debugger
- PGProf: graphical profiler
- Extensive online documentation
- Precompiled libraries come bundled (usage not recommended)

The intel suite

- Personal non-commercial license for Linux at no cost
- Platforms: x86, ia64, x86_64, Win32 and Linux
- Native OpenMP support
- Very good Fortran support
- C/C++ compiler supports many GNU extensions
- GNU Interoperability: can link g77 libraries&
- Advanced Optimizations: IPO and PGO
- Dbx debugger, but GNU tools can be used, too
- GNU gprof compatible profiler
- Extensive online documentation, tuning guides

IMHO

- Intel
  - too many releases.. (=too many bugs)
  - Performances: at the moment quite good...
- PGI:
  - diagnostic: not so good (it does not detect too many errors...)
  - Performances:still good but Intel is now performing better [at least on my codes]) than PGI compiler
- NAG:
  - Diagnostic: excellent !!
  - Performances: poor (at least on my code)
- Gfortran/G77
  - diagnostic: good
  - performances: not so good

First flag: know something about your compiler...

- Which version are you using ? (provide always this information to your sys. Adm.)
  - Gnu: -v
  - Pgi: -V
  - Intel: -v or -V

```
azorka~ 13>ifort -v
Version 8.0
azorka~ 14>ifort -V
Intel(R) Fortran Compiler for 32-bit applications, Version 8.0
\  Build 20040616Z Package ID: l_fc_pc_8.0.046_pe049.1
Copyright (C) 1985-2004 Intel Corporation.  All rights reserved.
FOR NON-COMMERCIAL USE ONLY
```

second flag: know everything about your compiler...

- Which flags can I use ?
  - Gnu: --help
  - Pgi: -help
  - Intel: --help

TIPS:

```
azorka~ 13>ifort -help | less
                     Intel(R) Fortran Compiler Help
                     ==============================

usage: ifort [options] file1 [file2 ...]

  where options represents zero or more compiler options

        fileN is a Fortran source (.f .for .ftn .f90 .fpp), assembly (.s),
            object (.o), static library (.a), or other linkable file
```

third flag: name your executable !

- By default all compiler will produce an executable named a.out.
- You can use a -o flag (standard for all the compiler) to generate something more meaningful ..

```
azorka~ 13>ifort -o my_code.x  my_code.f90
```

Summing up…

- We just scraped surface: lot of gory details not mentioned but leave them to computer scientists…
- I recommend you try out some of the commands on your preferred platform and gradually deepen your understanding.
- In general the better you understand the compiler you are using the better you will be able to exploit it.