From Source Code to Executable

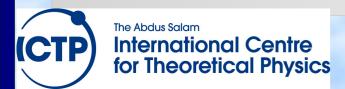
Dr. Axel Kohlmeyer

Scientific Computing Expert

Information and Telecommunication Section
The Abdus Salam International Centre
for Theoretical Physics

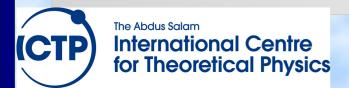
http://sites.google.com/site/akohlmey/

akohlmey@ictp.it



Pre-process / Compile / Link

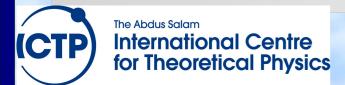
- Creating an executable includes multiple steps
- The "compiler" is a wrapper for <u>several</u> commands that are executed in succession
- The "compiler flags" similarly fall into categories and are handed down to the respective tools
- When compiling for different languages, only the first steps are language specific.
- We will look into a C example first, since this is the language the OS is (mostly) written in



A simple C Example

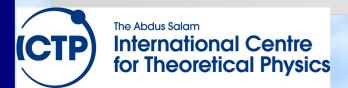
```
    Consider the minimal C program 'hello.c':

 #include <stdio.h>
 int main(int argc, char **argv)
        printf("hello world\n");
        return 0;
• i.e.: what happens, if we do:
 > gcc -o hello hello.c
 (try: gcc -v -o hello hello.c)
```



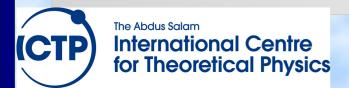
Step 1: Pre-processing

- Pre-processing is <u>mandatory</u> in C (and C++)
- Pre-processing will handle '#' directives
 - File inclusion with nested inclusion
 - Conditional compilation and Macro expansion
- In this case: /usr/include/stdio.h and all files are included by it are inserted and the contained macros expanded
- Use -E flag to stop after pre-processing:
 - > cc -E -o hello.pp.c hello.c



Step 2: Compilation

- Compiler converts a high-level language into the specific instruction set of the target CPU
- Individual steps:
 - Parse text (lexical + syntactical analysis)
 - Do language specific transformations
 - Translate to internal representation units (IRs)
 - Optimization (reorder, merge, eliminate)
 - Replace IRs with pieces of assembler language
- Try:> gcc -S hello.c (produces hello.s)



Compilation cont'd

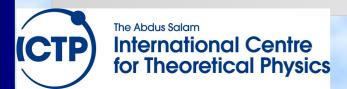
```
.file "hello.c"
                               gcc replaced printf with puts
       section
                 rodata
.LC0:
       .string "hello, world!
                                try: gcc -fno-builtin -S hello.c
       .text
.globl main
                              #include <stdio.h>
              main, @function
       .type
main:
                              int main(int argc,
       pushl
              %ebp
                                         char **arqv)
       movl %esp, %ebp
       andl $-16, %esp
       subl $16, %esp
                               printf("hello world\n");
       movl $.LCO, (%esp
                               return 0;
       call puts
             $0, %eax
       movl
       leave
       ret
       .size
              main, .-main
       .ident
              "GCC: (GNU) 4.5.1 20100924 (Red Hat 4.5.1-4)"
       .section
                      .note.GNU-stack, "", @progbits
```

Step 3: Assembler / Step 4: Linker

- Assembler (as) translates assembly to binary
 - Creates so-called object files (in ELF format)

```
Try: > gcc -c hello.c
Try: > nm hello.o
00000000 T main
U puts
```

- Linker (Id) puts binary together with startup code and required libraries
- Final step, result is executable.
 Try: > gcc -o hello hello.o



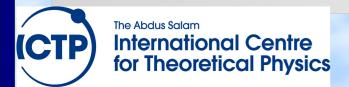
Adding Libraries

```
• Example 2: exp.c
#include <math.h>
#include <stdio.h>
int main(int argc, char **argv)
    double a=2.0;
     printf("exp(2.0)=%f\n", exp(a));
     return 0;
• > qcc -o exp exp.c
    Fails with "undefined reference to 'exp'". exp()
    is in "libm", but compiler does not link to it
• => gcc -o exp exp.c -lm
```



Symbols in Object Files & Visibility

- Compiled object files have multiple sections and a symbol table describing their entries:
 - "Text": this is executable code
 - "Data": pre-allocated variables storage
 - "Constants": read-only data
 - "Undefined": symbols that are used but not defined
 - "Debug": debugger information (e.g. line numbers)
- Entries in the object files can be inspected with either the "nm" tool or the "readelf" command



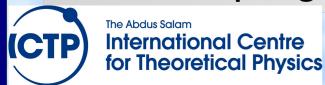
Example File: visbility.c

```
static const int val1 = -5;
const int val2 = 10;
static int val3 = -20;
int val4 = -15;
extern int errno;
static int add abs(const int v1, const int v2) {
   return abs(v1)+abs(v2);
                                       nm visibility.o:
                                       00000000 t add abs
int main(int argc, char **argv) {
                                                  U errno
    int val5 = 20;
                                       00000024 T main
    printf("%d / %d / %d\n",
                                                  U printf
           add abs(val1, val2),
           add abs(val3, val4),
                                       00000000 r val1
           add abs(val1,val5));
                                       00000004 R val2
    return 0:
                                       00000000 d val3
                                       00000004 D val4
```



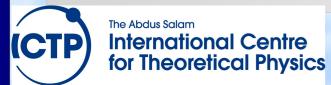
What Happens During Linking?

- Historically, the linker combines a "startup object" (crt1.o) with all compiled or listed object files, the C library (libc) and a "finish object" (crtn.o) into an executable (a.out)
- With shared libraries it is more complicated.
- The linker then "builds" the executable by matching undefined references with available entries in the symbol tables of the objects
- crt1.o has an undefined reference to "main" thus C programs start at the main() function



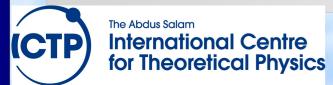
Libraries

- Static libraries built with the "ar" command are collections of objects with a global symbol table
- When linking to a static library, object code is <u>copied</u> into the resulting executable and all direct addresses recomputed (e.g. for "jumps")
- Symbols are resolved "from left to right", so circular dependencies require to list libraries multiple times or use a special linker flag
- When linking only the <u>name</u> of the symbol is checked, not whether its argument list matches



More on Shared Libraries

- Shared libraries are more like executables that are missing the main() function
- When linking to a shared library, a marker is added to load the library by its "generic" name and the list of undefined symbols
- When resolving a symbol (function) from shared library all addresses have to be recomputed (relocated) on the fly.
- The shared linker program is executed first and then loads the executable and its dependencies

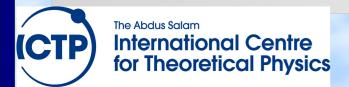


Dynamic Linker Issues

Linux defaults to dynamic libraries:

```
> ldd hello
linux-gate.so.1 => (0x0049d000)
libc.so.6 => /lib/libc.so.6
(0x005a0000)
/lib/ld-linux.so.2 (0x0057b000)
```

- /etc/ld.so.conf, LD_LIBRARY_PATH define where to search for shared libraries
- gcc -Wl,-rpath,/some/dir will encode /some/dir into the binary for searching



What is Different in Fortran?

- Basic compilation principles are the same
- In Fortran, symbols are case insensitive
 => most compilers translate them to lower case
- To make Fortran symbols different from C symbols, their names are modified (e.g. function have an underscore appended)
- Fortran programs don't have a "main" in the same way as C programs have (no arguments)
 PROGRAM => MAIN___ (in gfortran)
- C-like main provided as startup (to store args)

