The Abdus Salam
International Centre
for Theoretical Physics

2494-22

Workshop on High Performance Computing (HPC) Architecture and Applications in the ICTP

14 - 25 October 2013

Compiling and Linking with Static and Shared Libraries Using Multiple Programming Languages

Axel Kohlmeyer
ICTP, Trieste/Temple University
Philadelphia
USA

Compiling and Linking with Static and Shared Libraries Using Multiple Programming Languages

Dr. Axel Kohlmeyer

Sr. Scientific Computing Expert
ICTP, Trieste

Associate Dean for Scientific Computing
Temple University, Philadelphia

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

The Abdus Salam International Centre for Theoretical Physics

Pre-process / Compile / Link

- Creating an executable includes multiple steps
- The “compiler” (gcc) 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
- The “wrapper” selects the compiler language from source file name, but links “its” runtime
- We will look into a C example first, since this is the language the OS is (mostly) written in

The Abdus Salam International Centre for Theoretical Physics

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**)

The Abdus Salam International Centre for Theoretical Physics

Step 1: Pre-processing

- Pre-processing is <u>mandatory</u> in C (and C++)
- Pre-processing will handle '#' directives
  - File inclusion with support for 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`**

The Abdus Salam International Centre for Theoretical Physics

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`**)

The Abdus Salam International Centre for Theoretical Physics

Compilation cont'd

```
        .file   "hello.c"
        .section    .rodata
.LC0:
        .string "hello, world!"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    puts
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.5.1 20100924 (Red Hat 4.5.1-4)"
        .section        .note.GNU-stack,"",@progbits
```

gcc replaced `printf` with `puts`

try: gcc -fno-builtin -S hello.c

```
#include <stdio.h>
int main(int argc,
         char **argv)
{
 printf("hello world\n");
 return 0;
}
```

The Abdus Salam International Centre for Theoretical Physics

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 (ld) puts binary together with startup code and required libraries
- Final step, result is executable.

  Try: > `gcc -o hello hello.o`

The Abdus Salam International Centre for Theoretical Physics

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;
}
```

- `> gcc -o exp exp.c`

  Fails with “undefined reference to 'exp'”. Add: -lm

- `> gcc -O3 -o exp exp.c`

  Works due to inlining at high optimization level.

The Abdus Salam International Centre for Theoretical Physics

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);
}

int main(int argc, char **argv) {
     int val5 = 20;
     printf("%d / %d / %d\n",
            add_abs(val1,val2),
            add_abs(val3,val4),
            add_abs(val1,val5));
     return 0;
}
```

```
nm visibility.o:
00000000 t add_abs
         U errno
00000024 T main
         U printf
00000000 r val1
00000004 R val2
00000000 d val3
00000004 D val4
```

The Abdus Salam International Centre for Theoretical Physics

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 current compilers 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

Static 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 copied 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 name of the symbol is checked, not whether its argument list matches

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 (soname) 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

Differences When Linking

- Static libraries are fully resolved “left to right”; circular dependencies are only resolved between explicit objects or inside a library
  -> need to specify libraries multiple times
  or use: **-Wl,--start-group (...) -Wl,--end-group**
- Shared libraries symbols are **not** fully resolved at link time, only checked for symbols required by the object files. **Full check** only at runtime.
- Shared libraries may depend on other shared libraries whose symbols will be globally visible

The Abdus Salam International Centre for Theoretical Physics

Semi-static Linking

- Fully static linking is a bad idea with GNU libc; it requires matching shared objects for NSS
- Dynamic linkage of add-on libraries requires a compatible version to be installed (e.g. MKL)
- Static linkage of individual libs via linker flags **-Wl,-Bstatic,-lfftw3,-Bdynamic**
- can be combined with grouping, example:

```
-Wl,--start-group,-Bstatic                \
      -lmkl_gf_lp64 -lmkl_sequential      \
      -lmkl_core -Wl,--end-group,-Bdynamic
```

The Abdus Salam International Centre for Theoretical Physics

Meta-Libraries

- The GNU linker supports linker scripts a library
- Can be used to build a library-of-libraries:

```
[~]$ cat libscalapack.a

GROUP (-lscalapack_gnu -lblacsF77 -lblacs -llapack -lf77blas)
```

- To link the entire sequence of libraries only the flag `-lscalapack` is needed
- Useful to hide implementation details or handle library dependencies for static libraries (not an issue with shared libraries, since the shared library is linked to its dependencies)

The Abdus Salam International Centre for Theoretical Physics

Dynamic Linker Properties

- 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

The Abdus Salam International Centre for Theoretical Physics

Using LD_PRELOAD

- Using the LD_PRELOAD environment variable, symbols from a shared object can be preloaded into the global object table and will <u>override</u> those in later resolved shared libraries
=> replace specific functions in a shared library
- Example: override log() with a faster version:
```
#include "amdlibm.h"
double log(double x) { return amd_log(x); }
```
gcc -shared -o fasterlog.so faster.c -lamdlibm
- LD_PRELOAD=./fasterlog.so ./myprog-with

The Abdus Salam International Centre for Theoretical Physics

Before LD_PRELOAD

```
PerfTop:    8016 irqs/sec  kernel: 9.9%  exact:  0.0% [1000Hz cycles],  (all, 8 CPUs)
-------------------------------------------------------------------------------------

            samples  pcnt function                    DSO
            _______ _____ ___________________________ ______________________________

           53462.00 52.2% __ieee754_log               /lib64/libm-2.12.so
           10490.00 10.3% R_binary                    /opt/binf/R-2.13.0/lib64/R/bin/exec/R
            8704.00  8.5% clear_page_c                [kernel.kallsyms]
            5737.00  5.6% __ieee754_exp               /lib64/libm-2.12.so
            4645.00  4.5% math1                       /opt/binf/R-2.13.0/lib64/R/bin/exec/R
            3070.00  3.0% __log                       /lib64/libm-2.12.so
            3020.00  3.0% __isnan                     /lib64/libc-2.12.so
            2094.00  2.0% R_gc_internal               /opt/binf/R-2.13.0/lib64/R/bin/exec/R
            1643.00  1.6% do_summary                  /opt/binf/R-2.13.0/lib64/R/bin/exec/R
            1251.00  1.2% __isnan@plt                 /opt/binf/R-2.13.0/lib64/R/bin/exec/R
            1210.00  1.2% real_relop                  /opt/binf/R-2.13.0/lib64/R/bin/exec/R
            1161.00  1.1% __GI___exp                  /lib64/libm-2.12.so
             754.00  0.7% __isnan                     /lib64/libm-2.12.so
             739.00  0.7% R_log                       /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             553.00  0.5% __kernel_standard           /lib64/libm-2.12.so
             550.00  0.5% do_abs                      /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             462.00  0.5% __mul                       /lib64/libm-2.12.so
             439.00  0.4% coerceToReal                /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             413.00  0.4% finite                      /lib64/libm-2.12.so
             358.00  0.3% log@plt                     /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             182.00  0.2% get_page_from_freelist      [kernel.kallsyms]
             120.00  0.1% __alloc_pages_nodemask      [kernel.kallsyms]
```

ICTP International Centre for Theoretical Physics

After LD_PRELOAD

```
  PerfTop:    8020 irqs/sec  kernel:17.2%  exact:  0.0% [1000Hz cycles],  (all, 8 CPUs)
------------------------------------------------------------------------------------------------

             samples  pcnt function                    DSO
             _______ _____ ___________________________ ______________________________________

            24702.00 19.5% __amd_bas64_log             /opt/libs/fastermath-0.1/libamdlibm.so
            22270.00 17.6% R_binary                    /opt/binf/R-2.13.0/lib64/R/bin/exec/R
            18463.00 14.6% clear_page_c                [kernel.kallsyms]
            10480.00  8.3% __ieee754_exp               /lib64/libm-2.12.so
             9834.00  7.8% math1                       /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             9155.00  7.2% log                         /opt/libs/fastermath-0.1/fasterlog.so
             6269.00  5.0% __isnan                     /lib64/libc-2.12.so
             4214.00  3.3% R_gc_internal               /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             3074.00  2.4% do_summary                  /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             2285.00  1.8% real_relop                  /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             2257.00  1.8% __isnan@plt                 /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             2076.00  1.6% __GI___exp                  /lib64/libm-2.12.so
             1346.00  1.1% R_log                       /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             1213.00  1.0% do_abs                      /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             1075.00  0.8% __kernel_standard           /lib64/libm-2.12.so
              894.00  0.7% coerceToReal                /opt/binf/R-2.13.0/lib64/R/bin/exec/R
              780.00  0.6% __mul                       /lib64/libm-2.12.so
              756.00  0.6% finite                      /lib64/libm-2.12.so
              729.00  0.6% amd_log@plt                 /opt/libs/fastermath-0.1/fasterlog.so
              706.00  0.6% amd_log                     /opt/libs/fastermath-0.1/libamdlibm.so
              674.00  0.5% log@plt                     /opt/binf/R-2.13.0/lib64/R/bin/exec/R
```

The Abdus Salam International Centre for Theoretical Physics

Difference Between C and Fortran

- Basic compilation principles are the same
  => preprocess, compile, assemble, link
- In Fortran, symbols are <u>case insensitive</u>
  => most compilers <u>translate</u> them to lower case
- In Fortran symbol names may be modified to make them different from C symbols
  (e.g. append one or more underscores)
- Fortran entry point is not “main” (no arguments)
  PROGRAM => MAIN__ (in gfortran)
- C-like main() provided as startup (to store args)

Pre-processing in C and Fortran

- Pre-processing is mandatory in C/C++
- Pre-processing is optional in Fortran
- Fortran pre-processing enabled implicitly via file name: name.F, name.F90, name.FOR
- Legacy Fortran packages often use /lib/cpp:
  /lib/cpp -C -P -traditional -o name.f name.F
  - -C : keep comments (may be legal Fortran code)
  - -P : no '#line' markers (not legal Fortran syntax)
  - -traditional : don't collapse whitespace (incompatible with fixed format sources)

Fortran Symbols Example

```
SUBROUTINE GREET
  PRINT*, 'HELLO, WORLD!'
END SUBROUTINE GREET

program hello
  call greet
end program
```

```
0000006d t MAIN__
         U _gfortran_set_args
         U _gfortran_set_options
         U _gfortran_st_write
         U _gfortran_st_write_done
         U _gfortran_transfer_character
00000000 T greet_
0000007a T main
```

- “program” becomes symbol “MAIN__” (compiler dependent)
- “subroutine” name becomes lower case with '_' appended
- several “undefineds” with '_gfortran' prefix
  => calls into the Fortran runtime library, libgfortran
- cannot link object with “gcc” alone, need to add -lgfortran
  => cannot mix and match Fortran objects from different compilers

Fortran 90+ Modules

- When subroutines or variables are defined inside a module, they have to be hidden

```
module func
  integer :: val5, val6
contains
  integer function add_abs(v1,v2)
    integer, intent(in) :: v1, v2
    add_abs = iabs(v1)+iabs(v2)
  end function add_abs
end module func
```

- gfortran creates the following symbols:

```
00000000 T __func_MOD_add_abs
00000000 B __func_MOD_val5
00000004 B __func_MOD_val6
```

The Abdus Salam International Centre for Theoretical Physics

The Next Level: C++

- In C++ functions with different number or type of arguments can be defined (overloading) => encode prototype into symbol name:

  Example : symbol for `int add_abs(int,int)`
  becomes: `_ZL7add_absii`

- Note: the return type is <u>not</u> encoded
- C++ symbols are no longer compatible with C => add 'extern “C”' qualifier for C style symbols
- C++ symbol encoding is <u>compiler specific</u>

C++ Namespaces and Classes vs. Fortran 90 Modules

- Fortran 90 modules share functionality with classes and namespaces in C++
- C++ namespaces are encoded in symbols
  Example: `int func::add_abs(int,int)`
  becomes: `_ZN4funcL7add_absEii`
- C++ classes are encoded the same way
- Figuring out which symbol to encode into the object as undefined is the job of the compiler
- When using the gdb debugger use '::' syntax

The Abdus Salam International Centre for Theoretical Physics

Why We Need Header or Module Files

- The linker is "blind" for any language specific properties of a symbol => checking of the validity of the interface of a function is only possible during compilation
- A header or module file contains the prototype of the function (not the implementation) and the compiler can compare it to its use
- Important: header/module has to match library => Problem with FFTW-2.x: cannot tell if library was compiled for single or double precision

Calling C from Fortran 77

- Need to make C function look like Fortran 77
  - Append underscore (except on AIX, HP-UX)
  - Call by reference conventions
  - Best only used for “subroutine” constructs (cf. MPI) as passing return value of functions varies a lot:
    ```
    void add_abs_(int *v1,int *v2,int *res){
    *res = abs(*v1)+abs(*v2);}
    ```
- Arrays are always passed as “flat” 1d arrays by providing a pointer to the first array element
- Strings are tricky (no terminal 0, length added)

Calling C from Fortran 77 Example

```
void sum_abs_(int *in, int *num, int *out) {
 int i,sum;
 sum = 0;
 for (i=0; i < *num; ++i) { sum += abs(in[i]);}
   *out = sum;
   return;
}

/* fortran code:
   integer, parameter :: n=200
   integer :: s, data(n)

   call SUM_ABS(data, n, s)
   print*, s
*/
```

The Abdus Salam International Centre for Theoretical Physics

Calling Fortran 77 from C

- Inverse from previous, i.e. need to add underscore and use lower case (usually)
- Difficult for anything but Fortran 77 style calls since Fortran 90+ features need extra info
  - Shaped arrays, optional parameters, modules
- Arrays need to be “flat”,
  C-style multi-dimensional arrays are lists of pointers to individual pieces of storage, which may not be consecutive
  => use 1d and compute position

The Abdus Salam International Centre for Theoretical Physics

Calling Fortran 77 From C Example

```
subroutine sum_abs(in, num, out)
   integer, intent(in)  :: num, in(num)
   integer, intent(out) :: out
   Integer              :: i, sum
   sum = 0
   do i=1,num
     sum = sum + ABS(in(i))
   end do
   out = sum
end subroutine sum_abs
!! c code:
!   const int n=200;
!   int data[n], s;
!   sum_abs_(data, &n, &s);
!   printf("%d\n", s);
```

The Abdus Salam International Centre for Theoretical Physics

Modern Fortran vs C Interoperability

- Fortran 2003 introduces a standardized way to tell Fortran how C functions look like and how to make Fortran functions have a C-style ABI
- Module “iso_c_binding” provides kind definition: e.g. C_INT, C_FLOAT, C_SIGNED_CHAR
- Subroutines can be declared with “BIND(C)”
- Arguments can be given the property “VALUE” to indicate C-style call-by-value conventions
- String passing tricky, needs explicit 0-terminus

Calling C from Fortran 03 Example

```
int sum_abs(int *in, int num) {
  int i,sum;
  for (i=0,sum=0;i<num;++i) {sum += abs(in[i]);}
  return sum;
}
/* fortran code:
  use iso_c_binding, only: c_int
  interface
    integer(c_int) function sum_abs(in, num) bind(C)
      use iso_c_binding, only: c_int
      integer(c_int), intent(in) :: in(*)
      integer(c_int), value :: num
    end function sum_abs
  end interface
  integer(c_int), parameter :: n=200
  integer(c_int) :: data(n)
  print*, SUM_ABS(data,n)  */
```

The Abdus Salam International Centre for Theoretical Physics

Calling Fortran 03 From C Example

```
subroutine sum_abs(in, num, out) bind(c)
   use iso_c_binding, only : c_int
   integer(c_int), intent(in)  :: num,in(num)
   integer(c_int), intent(out) :: out
   integer(c_int),             :: i, sum
   sum = 0
   do i=1,num
     sum = sum + ABS(in(i))
   end do
   out = sum
end subroutine sum_abs

!! c code:
!    const int n=200;
!    int data[n], s;
!    sum_abs(data, &n, &s);
!    printf("%d\n", s);
```

The Abdus Salam International Centre for Theoretical Physics

Linking Multi-Language Binaries

- Inter-language calls via mutual C interface only due to name “mangling” of C++ / Fortran 90+ => extern “C”, ISO_C_BINDING, C wrappers
- Fortran “main” requires fortran compiler for link
- Global static C++ objects require C++ for link => avoid static objects (good idea in general)
- Either language requires its runtime for link
  => GNU: -lstdc++ and -lgfortran
  => Intel: “its complicated” (use -# to find out)
  more may be needed (-lgomp, -lpthread, -lm)