# From Source Code to Executable

**Dr. Axel Kohlmeyer**

Assistant Dean for High-Performance Computing
Associate Director, ICMS
Associate Director, TMI
College of Science and Technology
Temple University, Philadelphia
**axel.kohlmeyer@temple.edu**

External Scientific Associate
International Centre for Theoretical Physics, Trieste, Italy
**akohlmey@ictp.it**

# Pre-process / Compile / Link

- Creating an executable includes multiple steps
- The “compiler” (gcc) is a wrapper for several 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

# 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 mandatory 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`**

# 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"
        .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;
}
```

# 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
```

# 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.

# 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

```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
```

# 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 library 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

# Dynamic Linker Properties

- Linux defaults to dynamic libraries:

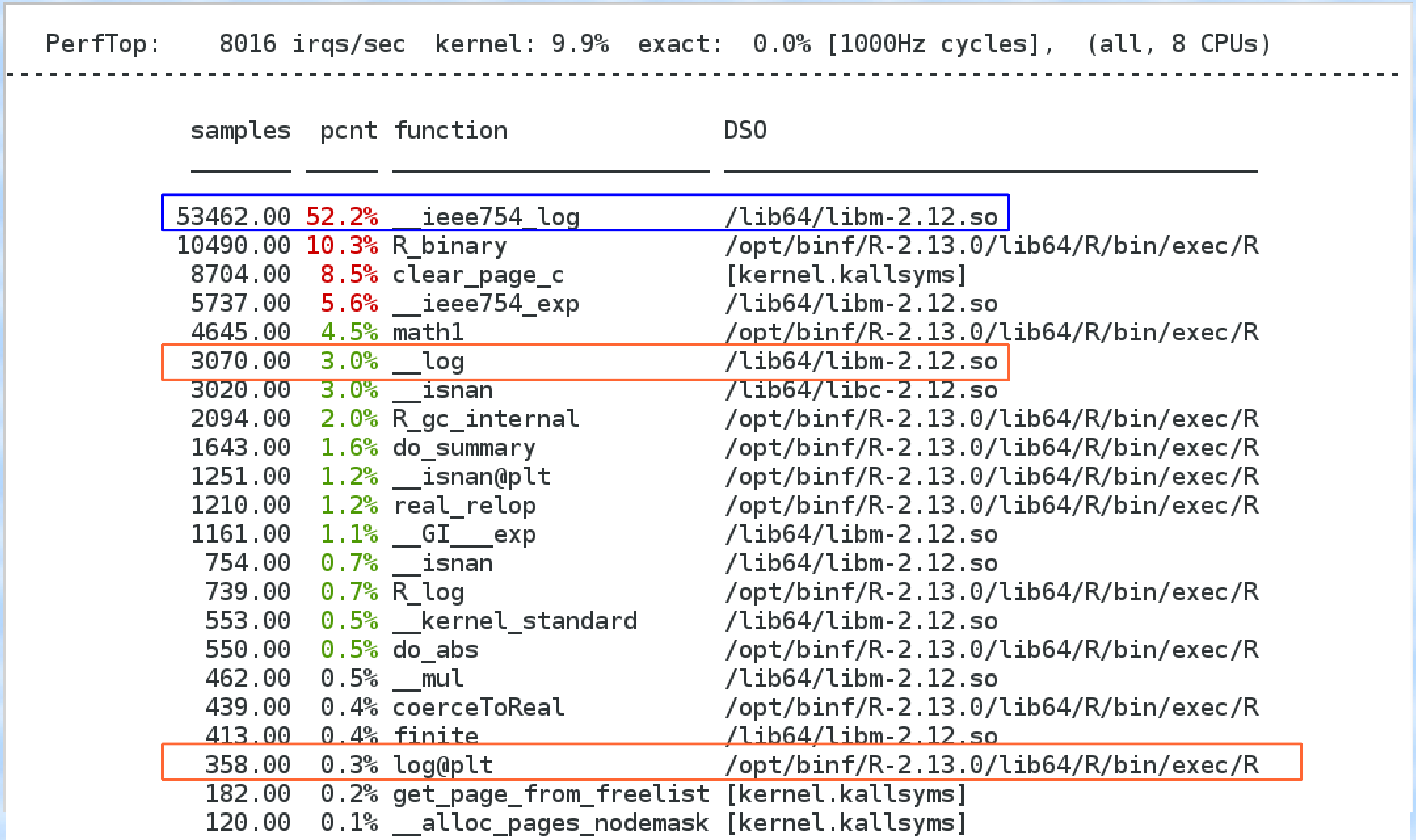
  ```
  > ldd hello
  linux-gate.so.1 =>  (0x0049d000)
  libc.so.6 => /lib/libc.so.6
  (0x005a0000)
  /lib/ld-linux.so.2 (0x0057b000)
  ```

- **`/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

# Using LD_PRELOAD

- Using the LD_PRELOAD environment variable, symbols from a shared object can be preloaded into the global object table and will override 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

# Before LD_PRELOAD

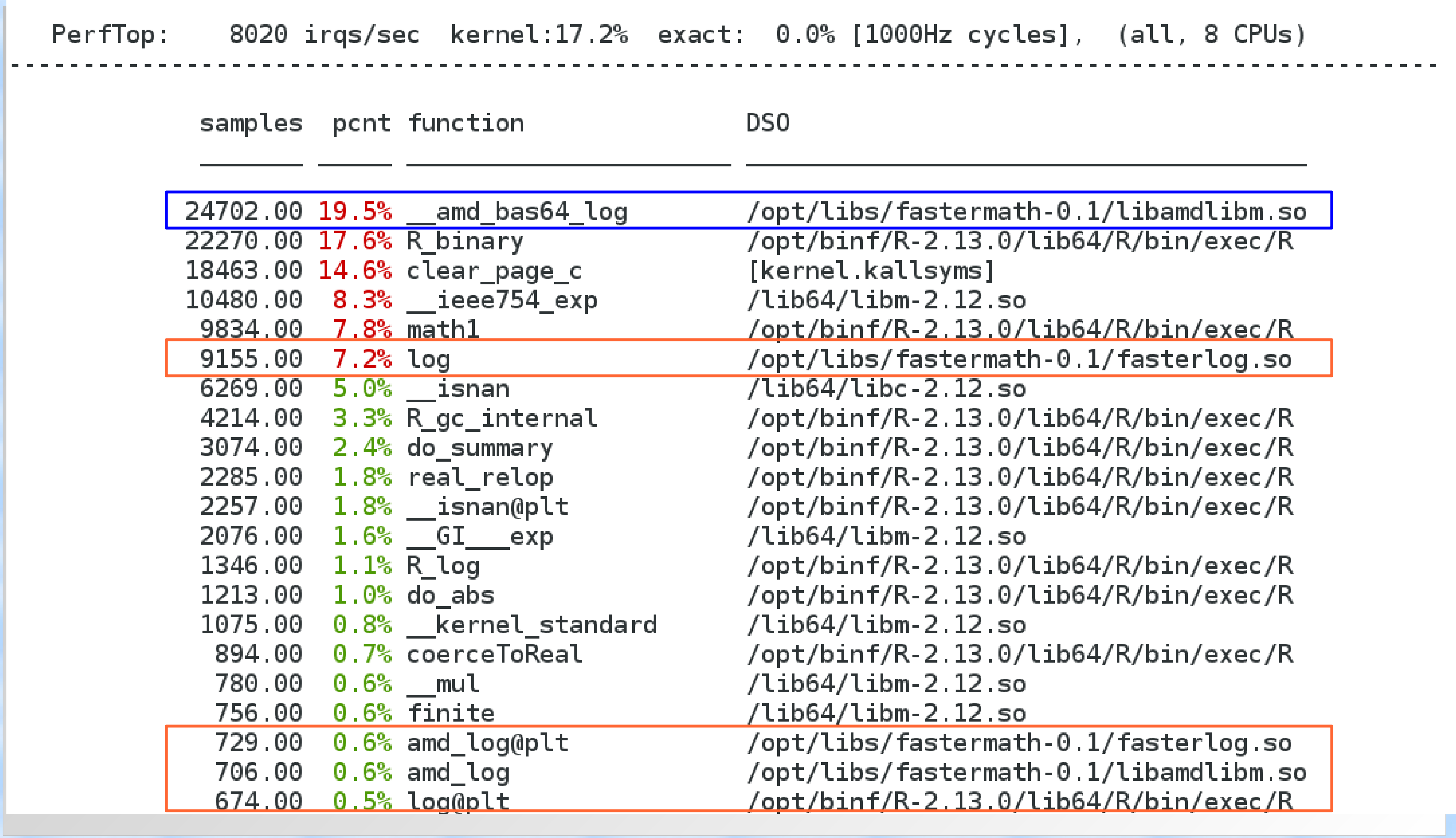


```
 PerfTop:    8016 irqs/sec  kernel: 9.9%  exact:  0.0% [1000Hz cycles],  (all, 8 CPUs)
------------------------------------------------------------------------------------------

           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]
```

# 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
```

# What is Different in Fortran?

- Basic compilation principles are the same
  => preprocess, compile, assemble, link
- In Fortran, symbols are case insensitive
  => most compilers translate 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)

# Compilers on x86

- GNU default on Linux: gcc, g++, gfortran, ...
  - Free, C/C++ quite good, gfortran focus on standards
  - 'native' Linux compilers
  - Support for many platforms, cross-compilation
- Other free compilers: clang/LLVM, open64
- Several commercial compilers for Linux:
  - Intel, PGI, Cray, NAG, Absoft, ...
- MacOS: clang/LLVM (used to be GNU)
- Windows: Microsoft, Intel, GNU (Cygwin,MinGW)

# Common Compiler Flags

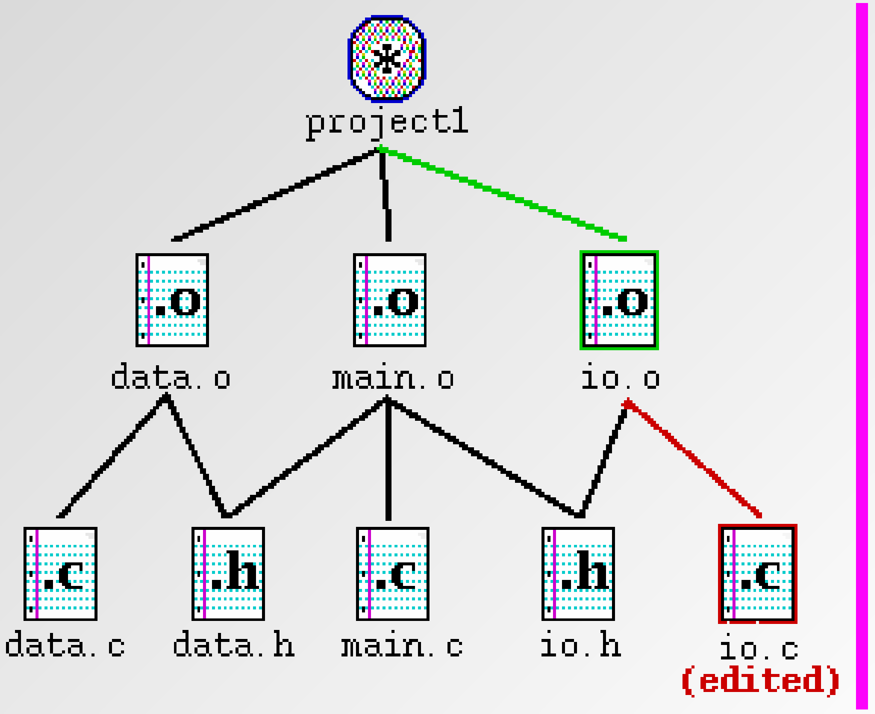
- Optimization: -O0, -O1, -O2, -O3, -O4, ...
  - Compiler will try to rearrange generated code so it executes faster
  - Aggressive compiler optimization may not always execute faster or may miscompile code
  - High optimization level (> 2) may alter semantics
- Preprocessor flags: -I/some/dir -DSOM_SYS
- Linker flags: -L/some/other/dir -lm
  -> search for libm.so/libm.a also in /some/dir

# Noteworthy Compiler Flags: GNU

- `-mtune=i686 -march=i386`
  optimize for i686 cpu, use i386 instruction set
  => compatibility with all 32-bit x86 CPUs
- `-msse, -msse2, -msse4.1 -mavx`
  enable using SSE, SSE2, SSE4.1, AVX for FP
- `-ffast-math`
  replace (some) mathematical constructs with faster alternatives, even if it reduces precision
- `-fopenmp`
  enable handling of OpenMP directives

# Makefiles: Concepts

- Simplify building large code projects
- Speed up re-compile on small changes
- Consistent build command: make
- Platform specific configuration via Variable definitions



# Makefiles: Syntax

- Rules:

```
target: prerequisites
          command
```

^this must be a 'Tab' (|<- ->|)

- Variables:

```
NAME= VALUE1 VALUE2 value3
```

- Comments:

```
# this is a comment
```

- Special keywords:

```
include linux.mk
```

# Makefiles: Rules Examples

```
# first target is default:
all: hello sqrt

hello: hello.c
       cc -o hello hello.c

sqrt: sqrt.o
       f77 -o sqrt sqrt.o
sqrt.o: sqrt.f
       f77 -o sqrt.o -c sqrt.f
```

# Makefiles: Variables Examples

```
# uncomment as needed
CC= gcc
#CC= icc -i-static
LD=$(CC)
CFLAGS= -O2

hello: hello.o
       $(LD) -o hello hello.o

hello.o: hello.c
       $(CC)-c $(CFLAGS) hello.c
```

# Makefiles: Automatic Variables

```
CC= gcc
CFLAGS= -O2

howdy: hello.o yall.o
       $(CC) -o $@ $^

hello.o: hello.c
       $(CC)-c $(CFLAGS) $<

yall.o: yall.c
       $(CC)-c $(CFLAGS) $<
```

# Makefiles: Pattern Rules

```
OBJECTS=hello.o yall.o

howdy: $(OBJECTS)
       $(CC) -o $@ $^

hello.o: hello.c
yall.o: yall.c

.c.o:
       $(CC)-o $@ -c $(CFLAGS) $<
```

Rule to translate all XXX.c files to XXX.o files

# Makefiles: Special Targets

```
.SUFFIXES:
.SUFFIXES: .o .F

.PHONY: clean install

.F.o:
     $(CPP) $(CPPFLAGS) $< -o $*.f
     $(FC)-o $@ -c $(FFLAGS) $*.f

clean:
         rm -f *.f *.o
```

Clear list of all known suffixes

Register new suffixes

Tell make to not look for theses files

# Makefiles: Calling make

- Override Variables:

```
make CC=icc CFLAGS='-O2 -unroll'
```

- Dry run (don't execute):

```
make -n
```

- Don't stop at errors (dangerous):

```
make -i
```

- Parallel make (requires careful design)

```
make -j2
```

- Use alternative Makefile

```
make -f make.pgi
```

# From Source Code to Executable

**Dr. Axel Kohlmeyer**

Assistant Dean for High-Performance Computing
Associate Director, ICMS
Associate Director, TMI
College of Science and Technology
Temple University, Philadelphia
**axel.kohlmeyer@temple.edu**

External Scientific Associate
International Centre for Theoretical Physics, Trieste, Italy
**akohlmey@ictp.it**