Advanced School in High Performance and GRID Computing -
Concepts and Applications

30 November - 11 December, 2009

From Source Code to Executable:
Preprocessing, Compiling, and Linking
(Part 2)

S.T. Brown
Carnegie Mellon University
Pittsburgh
USA
From Source Code to Executable: Preprocessing, Compiling, and Linking

Shawn T. Brown
Pittsburgh Supercomputing Center
ICTP, Trieste – Italy, Dec. 1st, 2009
Overview / Compiler

- The preprocess / compile / linking process
  - Individual steps in detail
- Preprocessing in C and Fortran
- The C-preprocessor, typical directives
- Compilers and Vendors
- Compiler Flags
- Linking Flags and Utilities
The compiling process

cc source.c –o executable

Source Code

Executable
The compiling process

Preprocessing

Compilation

Linking

Source Code

Executable
Example of the process

- Consider the minimal C program ‘hello.c’:

```c
#include <stdio.h>

int main (void) {
    printf("hello world\n");
    return 0;
}
```

- What happens if we do?
  > cc hello.c –o hello
Step 1: Preprocessing

- Handles all line in source code with ‘#’ directives
  - File inclusion
  - Conditional compilation
  - Macro expansion
- In our simple example:
  
  ```
  #include <stdio.h>
  ```
  
  - This translates to ‘insert file /usr/include/stdio.h’ into my source code.
- If you would like to see the pre-processed source:
  
  ```
  > cc -E hello.c -o hello.pp.c
  ```
Preprocessing, what is it good for?

• Selective compilation:

```c
#include <stdio.h>

int main (void) {

#ifdef LINUX
    printf("hello world from Linux\n");
#else
    printf("hello world from something else");
    return 0;
#endif
}
```

• `cc -DLINUX hello.c -o hello`
  – prints “hello world from Linux”
Step 2: Compilation

Parses Source Language
(lexical + syntactical analysis)

Translate to internal representation (trans-code)
(Optimizations: reorder, merge, eliminate)

Conversion to Assembly
(What the computer actually parses as instructions)
Assembly Language

- `cc -S hello.c`
  - Produces `hello.s`

```
.LC0:
    .string "hello world\n"
.text
.globl main
    .type main,@function
main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
    movl $0, %eax
    subl %eax, %esp
    subl $12, %esp
    pushl $.LC0
    call printf
    addl $16, %esp
    movl $0, %eax
    leave
    ret
```
Optimized Assembly

.LC0:
  .string "hello world"
  .text
  .p2align 2,,3
.globl main
.type main,@function
main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
andl $-16, %esp
subl $12, %esp
pushl $.LC0
call puts
xorl %eax, %eax
leave
ret
Assembler

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

> cc -c hello.c
> nm hello.o

00000000  T main
        U printf
Linker

- The Linker (ld) puts it all together
  - Adds startup code and library code to binary for creation of final executable.

> ld -o hello hello.o
> ./hello
> hello world
Adding Libraries

- Libraries are a powerful tool to give programmers access to optimized or highly used functions
- libmath example
  - `exp(double)` is a function provided by libmath.

```c
#include <math.h>
#include <stdio.h>
int main(int argc, char **argv)
{
    printf("exp(2.0)=%f\n", exp(2.0));
    return 0;
}
```

To compile: `cc -lm hello.c -o hello`
Practical Compiling Issues

• Preprocessing in C and Fortran
  – C/C++ mandatory
  – Optional in Fortran
    • Often implicit via file name: name.F, name.F90, name.FOR

• Can set define variables on the command line
  – -DDEF_ARR=200
  – Use capital letters to signal a define
C Pre-processor directives

- `#define MYVAL 100`
- `#undef MYVAL`
- `#if defined(MYVAL) && defined(__LINUX)`
- `#elif(MYVAL < 200)`
- `#else`
- `#endif`
- `#include "myfile.h"`
- `#include <mysysfile.h>`
Compilers: GNU, PGI, Intel (and pathScale)

• We will only cover C/C++, and Fortran 77/95
• GNU: gcc, g++, g77, gfortran
  – Free open-source (http://www.gcc.gnu.org)
  – ‘native’ compilers on Linux and Mac OS X
  – Available on virtually all computing architectures
  – C/C++ are very good, Fortran not so good.
• PGI: pgcc, pgCC, pgf77, pgf90
  – Commercial with trial, x86 and x86_64
  – Quite good if you need optimized Fortran code
• Intel: icc, icpc, ifort
  – Commercial with trial and non-commercial for Linux
  – x86, ia64 (Itanium) and EM64t (x86_64)
  – Available for Linux, Windows, and Mac OS X
Common Compiler Flags

• Optimization: -O, -O0, -O1, -O2, etc.
  – Predefined sets of optimization strategies
  – Can alter semantics (be careful)
  – For example: -O1 in gcc yields:

    -fauto-inc-dec -fcprop-registers -fdce -fdefer-pop
    -fdelayed-branch -fdse -fguess-branch-probability
    -fif-conversion2 -fif-conversion -finline-small-functions
    -fipa-pure-const -fipa-reference -fmerge-constants
    -fsplit-wide-types -ftree-built-in-call-dce -ftree-ccp
    -ftree-ch -ftree-copyrename -ftree-dce
    -ftree-dominator-opts -ftree-dse -ftree-fre -ftree-sra
    -ftree-ter -funit-at-a-time

• -g
  – Turns on debugging symbols
Common Compiler Flags

- Some flags are integral to normal compilation
  - `-c` Compile only
  - `-Dx` Preprocessor define
  - `-l/some/dir` search for include files here
  - `-L/some/dir` search for libraries
  - `-lname` link library named `libname`

```
> cc -c hello.cpp -l/usr/local/include
   -L/usr/local/lib -lm -DLINUX -o hello
```
Special Compiler Flags: GNU

- `-mtune=i686 -march=i386 (-mcpu sets both)`
  - optimize for i686 cpu, use i386 instruction set
- `-funroll-loopsheuristic`
  - loop unrolling (for floating point codes)
- `-fopenmp`
  - turns on OpenMP multithreaded parallelism
- `-ffast-mathreplace`
  - some constructs with faster alternatives
- `-fomit-frame-pointeruse`
  - stack pointer as general purpose register
- `-mieee-fptturn`
  - on IEEE754 compliance / comparisons
Special Compiler Flags: PGI

- **-tp=px, -tp=amd64, -tp=x64, -tp=piv**
  - generate architecture specific code

- **-pc=64**
  - set floating-point rounding mode to 64-bit

- **-Munroll**
  - loop unrolling,

- **-Mvect**
  - vectorization (loop scheduling)

- **-fast, -fastsse**
  - short cuts to optimization flags

- **-mp**
  - Turns on OpenMP

- **-Mipa**
  - turn on interprocedural analysis

- **-Kieee**
  - turn on IEEE floating point
Special Compiler Flags: Intel

- `-tpp6`
  - set cpu type, v10 supports GNU style

- `-pc64`
  - set floating point rounding to 64-bit

- `-ip, ipo`
  - interprocedural optimization

- `-axPW`
  - generate SSE, SSE3 instructions

- `-unroll`
  - heuristic loop unrolling

- `-openmp`
  - turn on OpenMP

- `-i-static`
  - link compiler runtime statically

- `-mp`
  - force IEEE floating point handling

- `-mp1`
  - almost force IEEE floating point

- `-fast`
  - shortcut for `-xP -O3 -ipo -no-prec-div`
Where to go?

• Each of these compilers have hundreds of flags
  – You will very likely not need to know but a few to do your work and get decently optimized code.
  – First place to look is documentation
  – The compiler man page should tell you every flag and what it does.