CS395T: Introduction to Scientific and Technical Computing

Compilers

Instructors

Dr. Karl W. Schulz, TACC

Dr. Victor Eijkhout, TACC

UNIX and C

- UNIX originally written in B (C's predecessor)
- Computer architecture changes necessitated language changes
  - developers' machine went from word-oriented memory to byte-oriented memory
  - B -> C
- UNIX quickly rewritten in C as it developed
- Simplicity of C enabled easier portability
- Easier portability let UNIX jump architectures
- C Standard Library became an integral part of UNIX

Compilers

- Compilers (and assemblers and linkers) turn human-readable programming languages in to machine-executable programs
- Scientific computing tends towards
  - FORTRAN (and its dialects)
    - predates UNIX
    - designed so that compiler can know easily what optimizations it can do
  - C (and its dialects)
    - UNIX is written in C
    - supercomputers run UNIX

Compiler Availability

- OS Defaults
  - Linux: GNU
  - AIX: XL
  - Solaris: Sun Studio
- Vendor
  - Intel: Intel
  - IBM: XL
  - Sun: Sun Studio
- 3rd Party (x86)
  - Portland Group
  - Pathscale

Popular Compiler Familes

- GCC: GNU Compiler Collection
  - Supported languages: C, C++, Objective C, Fortran77/95*, Java, Ada
  - Highly portable (available for most architectures and some toasters)
  - Free Software
- Intel
  - Supported languages: C, C++, Fortran77/95
  - Available for Linux and Windows for x86 architectures, optimized for Intel chips (rumors of crippling for AMD chips)
  - Free for non-commercial use on Linux
- Portland Group
  - Supported languages: C, C++, Fortran77/95, HPF
  - Available for x86 (Linux and Windows)
- Pathscale (acquired by QLogic, subsequently acquired by SiCortex)
  - Supported languages: C, C++, Fortran77/95
  - Available for x86 (Linux only)
- IBM XL
  - Supported languages: C, C++, Fortran77/95
  - Available for Power and PowerPC (AIX, OS X, and Linux)
- Sun Studio
  - Supported languages: C, C++, Fortran77/95
  - Available for Solaris (Sparc and x86), probably Linux some day through OpenSolaris

Compilers and Linkers

- Traditionally, *cc* was the compiler and *ld* was the linker on UNIX systems
- Compiler produced object files
  - one source file—one object file
  - contain machine code
  - but not executable independently
- Linker put them together into executables
- These days the compiler knows how to call the linker for you

Compilers and Linkers

- Every object file contains
  - executable machine code
  - symbol table
    - functions
    - variables
    - types
- The linker coordinates symbols amongst object files (and system libraries) to create the executable
  - usually no symbol table in the result
    - debuggers need the symbols
    - compiler/linker flags to add them to the executable

Invocation

GNU	gcc, g++, g77, gfortran/g95
Intel	icc, ifort
IBM XL	xlc, xlC, xlf, xlf95
Sun Studio	cc, CC, f77, f95
Portland Group	pgcc, pgCC, pgf77, pgf95
Pathscale	pathcc, pathCC, pathf77, pathf95

Invocation

- Most basic
  $CC foo.c
  – Creates the executable *a.out*
  – Links in default libraries only (*libc* for certain, others vary by compiler/architecture/OS)
- Name the output
  $CC –o foo foo.c
  – creates executable *foo*
- Compile only (no external linking)
  $CC –c foo.c
  – creates object file *foo.o*

Invocation

- Compile only, name the output

  $CC –o bar.o –c foo.c

- Add debugging symbols

  $CC –g foo.c –o foo

Invocation

- Multiple source files

  $CC foo.c bar.c baz.c –o foo

- Link multiple object files into one executable

  $CC foo.o fun1.o fun2.o –o foo

- Link in a library by hand

  $CC foo.c –o foo /home/user/mylib/libmylib.a

  – Works just like object file linking

Invocation

- Link in a library

  $CC foo.c –o foo –lm

  – looks for *libm.a* or *libm.so* in the compiler/linker search path

  – most compilers choose **.so* over **.a*

  – */lib* and */usr/lib* + compiler/linker internal directories included in the default search path

- Link a library in a non-standard path

  $CC foo.c –o foo –L/home/user/mylib/ –lmylib

  – adds */home/user/mylib/path* to the library search path (at link time, more on run time later)

  – looks for *libmylib.a* or *libmylib.so* in the search path

Static vs. Dynamic Linking

- Static
  - puts all the external routines into the created executable
  - no dependencies at run time (for static libraries)
  - leads to larger binaries
  - takes longer to build the executable
- Dynamic
  - leaves the routines in the library file
  - loads the routines at run time
  - decreases the the build time and binary size
  - dynamic libraries can be shared by different executables in memory!

Static vs. Dynamic Libraries

- Static
  - usually called *'libfoo.a'*
  - created as an archive of object files
  - no special options needed when building
- Dynamic
  - usually called *'libfoo.so'*
  - more complicated to build than static libraries

Forcing Static Linking

- Dynamic linking preferred on most systems when both *‘libfoo.a’* and *‘libfoo.so’* are available
  - the in memory sharing can be a big win for certain libraries that everyone uses
- Most compilers can be forced to link statically
  - GNU and Intel: *-static*
- Sometimes a static version of the library isn’t available and using *–static* will cause a error
  - use the “by hand” linking method in these cases

TACC

Finding Dynamic Libraries

- At run time, the Linux Loader *(ld.so(8))* tries to resolve the shared library dependencies of an executable before it runs it
- It looks in
  - paths listed in its configuration file: */etc/ld.so.conf*
  - *LD_LIBRARY_PATH* in your environment
    - a colon-separated list of places to look just like *PATH and MANPATH*
  - the search path built in to the executable

Adding to the Executable's Search Path

- You can add to the search path embedded in the executable

  $CC –o foo –lmylib –L/home/user/mylib/mylib
  –Wl,-rpath,/home/user/mylib/mylib

- *-Wl* used to pass command line arguments directly to the linker
- *-rpath* linker option to add to the executable's search path

ldd

- The *ldd* command can be used to investigate the shared library dependencies of an executable

```
lslogin1$ ldd foo
        libm.so.6 => /lib64/tls/libm.so.6 (0x0000003ee3d00000)
        libc.so.6 => /lib64/tls/libc.so.6 (0x0000003ee3a00000)
        libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x0000003eeb400000)
        libdl.so.2 => /lib64/libdl.so.2 (0x0000003ee3f00000)
        /lib64/ld-linux-x86-64.so.2 (0x0000003ee3800000)
```

Examining Object Files

- *nm* can be used to list the symbols in object files, libraries, and executables
  - default shows static function name symbols only
  - can be used to display debugging and dynamic symbols as well
  - most useful for determining actual names of undefined symbols (for inter-language calls especially)

nm Example

foo.c:

```
#include "bar.h“
int c=3;
int d=4;
int main()
{
  int a=2;

  return(bar(a*c*d));
}
```

bar.c:

```
#include "bar.h"
int bar(int a)
{
  int b=10;
  return(b*a);
}
```

bar.h:

```
int bar(int);
```

TACC

nm Example

- Compile

```
gcc -g -c -o foo.o foo.c
gcc -g -c -o bar.o bar.c
```

- Link

```
gcc foo.o bar.o -o foo
```

nm Example

```
lslogin1$ nm foo.o bar.o
foo.o:
         U bar
00000000 D c
00000004 D d
00000000 T main
bar.o:
00000000 T bar
```

- “U” that the symbol “bar” is unknown in foo.o
- “T” means the symbol is listed in the text section of the object file
- “D” means that the symbol defines the location of global, initialized data

nm

- Useful options
  - *-a* show all symbols
  - *-u* show only undefined symbols
- Uppercase letters for global symbols, lowercase for local symbols
- Other codes
  - C, uninitialized data
  - N, debugging symbol
  - R, read-only data

C++

- ABI (Application Binary Interface)
  - Different compilers on the same architecture/OS may produce incompatible code
  - Mostly a name mangling problem
    - need a consistent naming scheme for class member functions
  - Just about every compiler uses the same standard now
- Different compilers use different template instantiation methods
  - Must make sure that each template implemented only once (if needed)
  - Must not implement the template if it isn’t need

C++ Template Instantiation

- Borland Model
  – Instantiate templates where used
  – Let the linker sort out the details
- Cfront Model
  – Build a repository of who needs what templates
  – Generate actual instantiations at link time
- Most compilers support more than one method but default to one of the above

FORTRAN

- FORTRAN is pass-by-reference
  - C is pass-by-value
- FORTRAN is case-insensitive
  - C is case sensitive
- FORTRAN and C functions would appear to be incompatible
  - which might mean FORTRAN was incompatible with UNIX!
- FORTRAN compilers protect you by
  - knowing how to call the needed parts of the system libraries
  - mangling your function names

FORTRAN Function Naming

- Every compiler has its own method
- Intel and GNU default to
  - lowercase
  - append 1 underscore
- IBM XL defaults to
  - lowercase
  - doesn’t append underscores (dangerous)

Example

foo.f:

```
      subroutine FoO(a)
        integer a
        a=a+1
      end
```

```
lslogin1$ g77 -c foo.f
lslogin1$ nm foo.o
00000000 T foo_
```

Example (continued)

call_foo.c:

```
void foo_(int *a);
int main()
{
  int b=10;
  foo_(&b);
  return(b);
}
```

```
lslogin1$ gcc -o call_foo
   call_foo.c foo.o
lslogin1$ ./call_foo
lslogin1$ echo $?
11
lslogin1$
```

FORTRAN Strings

- FORTRAN strings are blank padded
- The length is fixed
- Each compiler passes strings its own way
- Intel and GNU use a dummy length argument
  - which is passed by value!

Example

foo.f:

```
      program foo
      character a*10
      a='1234567'
      call bar(a)
      stop
      end
```

bar.c:

```
#include <stdio.h>
void bar_(char *str,
          int len)
{
  int i;
  for(i=0; i < len; i++)
    printf("%c:",str[i]);
  printf("\n");
}
```

Example (continued)

```
lslogin1$ gcc -c bar.c
lslogin1$ g77 -o foo foo.f bar.o
lslogin1$ ./foo
1:2:3:4:5:6:7: : : :
lslogin1$
```

Calling UNIX from FORTRAN

- Many compilers provide a UNIX-FORTRAN portability library
  - gives FORTRAN access to UNIX standard library functions not needed by the compiler itself
- Intel
  - ftp://download.intel.com/support/performancetools/fortran/windows/v9/lib_for.pdf
  - info
    - getenv(3)/GETENV
    - stat(2)/STAT
  - process control
    - kill(2)/KILL
    - sleep(3)/SLEEP
  - directory/file manipulation
    - mkdir(2)/MAKEDIRQQ
- Otherwise you're on your own to write wrappers

Compiler Warnings

- The compiler can warn you about certain constructs that aren't syntactically wrong but may cause strange behavior or run-time errors
- With many compilers *–W<foo>* turns on warnings about *foo*
- *-Wall* turns on many (but not always all) warnings

  $CC –c –Wall foo.c
- *-w* to disable warning messages all together

TACC

Warning Example

```
int a=1;
int main()
{
  int a;
  int b=1;
  return (a+b);
}
```

```
lslogin1$ gcc -O1 -c foo.c
-Wuninitialized -Wshadow

foo.c: In function ‘main’:

foo.c:4: warning: declaration of
‘a’ shadows a global declaration

foo.c:1: warning: shadowed
declaration is here

foo.c:7: warning: ‘a’ is used
uninitialized in this function
```

Compiler Optimization

- By default compilers try to
  - reduced compilation time
  - execute code faithfully
  - make debugging make sense
- Compilier optimization can
  - increase compilation time (dramatically)
  - reduce run time
  - increase or decrease executable size
  - change the order of operations
  - eliminate some code completely
  - introduce new code

Invoking the Optimizer

- Basic

  $CC –O –o foo foo.c

- More

  $CC –O# -o foo foo.c

  – # usually in the range [1-3]

  – each level inclusive of previous levels

  – optimization levels represent a suite of options which may be enabled/disabled individually

Compiler Optimization

- For GCC, level 1 turns on:
  - *-fdefer-pop*
  - *-fdelayed-branch -fguess-branch-probability*
  - *-fcprop-registers*
  - *-floop-optimize*
  - *-fif-conversion -fif-conversion2*
  - *-ftree-ccp -ftree-dce -ftree-dominator-opts -ftree-dse -ftree-ter -ftree-lrs -ftree-sra -ftree-copyrename -ftree-fre -ftree-ch*
  - *-fmerge-constants*
  - *-fomit-frame-pointer* (where it doesn't interfere with debugging)

Compiler Optimization

- *-floop-optimize*
- Perform loop basic optimizations
  - move constant expressions out of loops
  - simplify exit test conditions
  - do strength-reduction

Strength Reduction

- Replacement of an expensive calculation with a cheaper one:
  - replace *x/2.0* by *0.5*x*
  - simplify array addressing in loops

Constant Folding and Propagation

- Constant folding
  - Simplify expressions containing multiple constants

  *i = 3*4*5;* becomes *i=60;*
- Constant Propagation
  - replace constant-valued variable references with values

```
int x = 10;          →          int x = 10;
int y = 5*x;                    int y = 5*10;
```

- Multiple passes may be applied

Example

```
#define M 200
#define N 320
void zero_buf(unsigned int *buf)
{
  unsigned int i, j;
  for (j=0; j<N; ++j)
    for(i=0; i<M; ++i)
      buf[j*N+i]=0;
}
int main()
{
  unsigned int buf[M*N];
  zero_buf(buf);
  return(buf[M-1]);
}
```

How does this function get implemented?

Example

```
        movl    $0, -4(%ebp)
        jmp     .L2
.L3:
        movl    $0, -8(%ebp)
        jmp     .L4
.L5:
        movl    -4(%ebp), %edx
        movl    %edx, %eax
        addl    %eax, %eax
        addl    %eax, %eax
        addl    %edx, %eax
        sall    $6, %eax
        addl    -8(%ebp), %eax
        addl    %eax, %eax
        addl    %eax, %eax
        addl    8(%ebp), %eax
        movl    $0, (%eax)
        leal    -8(%ebp), %eax
        addl    $1, (%eax)
.L4:
        cmpl    $199, -8(%ebp)
        jbe     .L5
        leal    -4(%ebp), %eax
        addl    $1, (%eax)
.L2:
        cmpl    $319, -4(%ebp)
        jbe     .L3
```

Compiled with –O0

```
        movl    8(%ebp), %ecx
        movl    $320, %ebx
.L2:
        movl    %ecx, %eax
        movl    $0, %edx
.L3:
        movl    $0, (%eax)
        addl    $1, %edx
        addl    $4, %eax
        cmpl    $200, %edx
        jne     .L3
        addl    $1280, %ecx
        subl    $1, %ebx
        jne     .L2
```

Compiled with –O1

Example

- Level 0
  - 23 instructions
  - uses memory locations for the counters
  - complex calculation of array index
- Level 1
  - 12 instructions
  - uses registers for the counters
  - maintains base ($j*N$) plus offset (i) explicitly

Making Static Libraries

- Common code that is useful between programs or that changes in frequently can be put in a library with *ar*
- *ar* is more than an object file archiver
  - you can use it for any kind of files
  - nobody does (well, except for Debian)
  - tar is better

Invoking *ar*

- *ar r libfoo.a foo.o bar.o baz.o*
  - creates/adds to *libfoo.a*
  - inserts *foo.o, bar.o,* and *baz.o*
  - overwrites members of the same name
- *ar s libfoo.a*
  - creates or updates the object-file symbol table *libfoo.a*
  - may be combined with ‘r’ to do it all at once
  - *ar rs libfoo.a foo.o bar.o baz.o*
  - *ranlib libfoo.a* is a synonym
- *ar t libfoo.a*
  - prints the list of files contained in *libfoo.a*