CSC 2700: Scientific Computing

Compiling, Debugging, Profiling

Dr Frank Löffler

Center for Computation and Technology
Louisiana State University, Baton Rouge, LA

February 18th 2014

Compiling

Compiler

- computer program that transforms source code written in a programming language (the *source language*) into another computer language

Reason:

- CPU can only directly execute *machine code*.
- Developer wants to program in high-level language.

Hofstadter: "*Looking at a program written in machine language is vaguely comparable to looking at a DNA molecule atom by atom.*"

Don't confuse *machine code* with *Assembly language*

- Low-level programming language
- Symbolic representation of the binary machine code

Terms

Cross-compiler

- Generates *target code* which can only be run on different CPU/OS than compiling host

Decompiler

- Translates from a low level language to a higher level one

Language translator

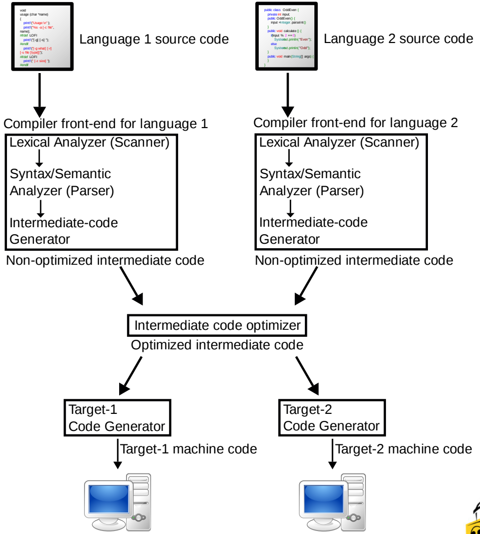
- translates between high-level languages

Bootstrapping

- Writing a compiler in the target language which it is intended to compile (chicken and egg problem)

Compiler operations

- Preprocessing
- Lexical analysis
- Parsing
- Semantic analysis
- Code generation
- Code optimization



Compiler structure

1. Frontend
   - checks whether the program is correctly written in terms of the programming language syntax and semantics
   - errors are reported, if any, in a useful way
   - type checking is also performed by collecting type information
   - generates intermediate representation for the middle-end
2. Middle-end
   - optimizations for performance
   - generates intermediate representation for the back-end
3. Back-end
   - translation of intermediate representation into the target assembly code
   - utilizes the hardware by figuring out how to e.g. keep parallel FUs busy

Executable

File that causes a computer "to perform indicated tasks according to encoded instructions"

- Sometimes designated filename extensions (such as `.exe`)
- Sometimes noted alongside file metadata (such as "execute bit")
- Usually file is checked before actual execution regardless
- Types:
  - Binary format, containing machine code (e.g. ELF)
  - Source code for use in scripting language (in a loose sense)
- Example formats
  - a.out - "assembler output"
  - COFF - "Common Object File Format"
  - ELF - "Executable and Linkable Format"
  - DOS COM - "Command file"
  - PE - "Portable Executable"

Object files / Libraries

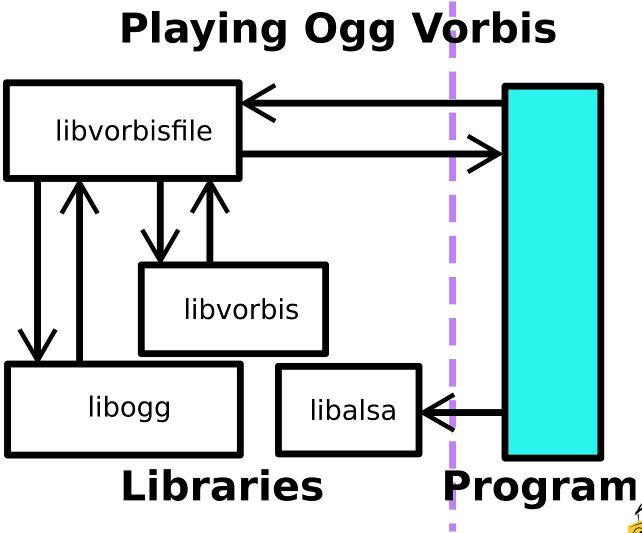
Object file: Organized collection of separate, named sequences of machine code

- contains instructions for the host machine to accomplish some task
- possibly accompanied by related data and metadata
- linker is typically used to generate an executable or library by combining parts of object files

Library: Collection of subroutines or classes

- Contains code and data that provide services to independent programs
- Allows the sharing and changing of code and data in a modular fashion

Library real-world example



Library types

- Static
  - collection of ordinary object files
  - conventional suffix: `.a`
  - created using an archiver program, e.g. `ar`:
    `ar rcs my_library.a file1.o file2.o`
  - linked into executable at compile time
- Shared / Dynamic
- not included in executable
- conventional suffix: `.so` or `.dll`
- Two types:
  - Loaded at program start
  - Loaded when needed while program execution

Shared Library names

Shared Library Names

- "linker name"
  - prefix `lib` + name of library + `.so`
  - used e.g. when requesting to link against this library
  - usually link to "soname"
  - example: `libreadline.so`
- "soname"
  - linker name + `.` + version number
  - changes with every library API change
  - used for dependencies
  - example: `libreadline.so.3`
- "real name"
  - filename of file containing actual library code
  - soname + `.` + minor number (+ `.` + release number)
  - example: `libreadline.so.3.0.0`

Filesystem placement

Filesystem Hierarchy Standard (FHS):

- Most libraries: `/usr/lib`
- Libraries required for startup: `/lib`
- Libraries that are not part of the system: `/usr/local/lib`

Starting ELF executable:

- Program loader is loaded and run (`/lib/ld-linux.so.X`)
- Loader finds and loads all other required shared libraries
- Uses library location information cache (see `ldconfig`)

Creating shared library

- Use "position independent code": compiler flag `-fPIC` / `-fpic`
- Specify soname to linker: `-soname your_soname`,
  or through compiler: `-Wl,-soname,your_soname`

In one command:

```
gcc -shared -Wl,-soname,your_soname \
    -o library_name file_list library_list
```

Complete Example:

```
gcc -fPIC -g -c -Wall a.c
gcc -fPIC -g -c -Wall b.c
gcc -shared -Wl,-soname,libmystuff.so.1 \
    -o libmystuff.so.1.0.1 a.o b.o -lc
```

Environment Variables

Colon-separated directory lists:

- `LD_RUN_PATH`
  - Additional search-path for libraries, encoded into executable while linking
  - Ignored if `-R`/`-rpath` linker option given
  - Check binary: `readelf -d FILE | grep RPATH`
- `LD_LIBRARY_PATH`
  - Additional search-path for libraries, used at run-time by dynamical linker
  - Good for testing, but try to avoid longer use

Library path examples

- Set rpath on executable/library through compiler
  `-Wl,-rpath,/path/to/used/library`
- Add directory to search-path for linking files
  `LD_RUN_PATH=/path/to/used/library:$LD_RUN_PATH`
- Add directory to temporary search run-time path
  `LD_LIBRARY_PATH=/path/to/used/library:$LD_LIBRARY_PATH`
- List used libraries of object

```
$ ldd /bin/cat
 linux-vdso.so.1 =>  (0x00007fff4fde3000)
 libc.so.6 => /lib64/libc.so.6 (0x0000003afa600000)
 /lib64/ld-linux-x86-64.so.2 (0x0000003afa200000)
```

List symbols

`nm`

- Reports symbols in given library
- Details: name, value, type, definition location (filename and line number)
- Local types: lower case
- Global types: upper case
  - `T`: typical definition in code section
  - `D`: initialized data section
  - `B`: uninitialized data section
  - `U`: undefined symbol (used but not defined here)
  - `W`: weak symbol, can be overwritten

Example:

```
$ nm /lib/libm.so.6 | grep " sqrt"
0000a870 W sqrt
00011840 W sqrtf
00019180 W sqrtl
```

GCC Command Reference

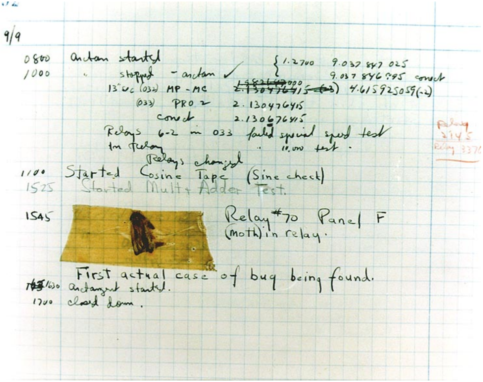
-c	Compile, but don't link
-o filename	Specify output filename
-v	Be verbose
-Wall	Print warnings about (potential) problems
-g	Produce debugging information
-pg	Produce profiling information for gprof
-O0	Optimize nothing
-O, *-O1*	Optimize some
-O2	Optimize a lot
-O3	Optimize the most, potentially unreliable
-std=standard	Select language standard (e.g. c99)
-Idirectory	Searched directory for header files
-Ldirectory	Searches directory for libraries while linking
-llibrary	Links against library, put after needing object

Debugging

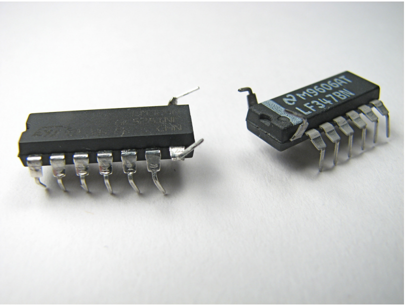
Bugs

Thomas Edison in 1878: "It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise—this thing gives out and [it is] then that 'Bugs' — as such little faults and difficulties are called—show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached."

1947:



Bug prevention



oskay@flickr

- Programming style
- Programming techniques
  e.g. self-checking programs
- Development methodologies
  e.g. managing programmer activity
- Programming language support
  e.g. types, name spaces, modules
- Code analysis
- Instrumentation

Debugging

Methodical process of finding and reducing the number of bugs

Debugger: software tools which enable the programmer to

- monitor the execution of a program
- stop it
- re-start it
- set breakpoints
- change values in memory

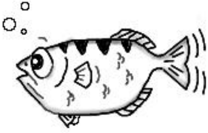
Debugging Steps

1. Reproduce problem
   - Problem might be reported by someone else
   - Problem might only occur in "some situations"
   - Knowing what "some situations" means can be important
2. Simplify problem
   - might be parallel
   - might take very long to reproduce
3. Use debugger to examine program states
   - variable values
   - call stack
4. Find cause of problem
   - not necessarily where it shows
5. Fix it

Debugging Techniques

- Print debugging (tracing)
  - watching print/trace statements, indicating flow of process
- Direct debugging by
  - Starting process inside debugger
  - Attaching debugger to running, local process
- Post-mortem debugging
  - debugging after program has crashed
  - e.g. analysis of memory dump (core dump)
- Remote debugging
  - Debug process on remote system via network interaction

GDB - The GNU Project Debugger

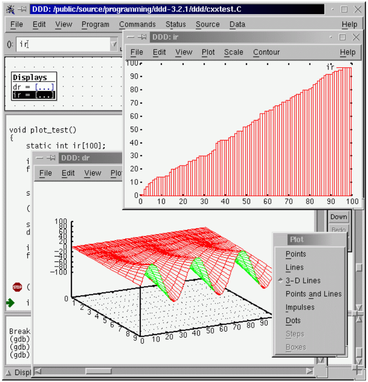


Can

- Start programs, specifying anything that might affect its behavior
- Make programs stop on specified conditions
- Examine what has happened, when programs stopp/crash
- Change things during program execution
- Debug many languages, e.g. Ada, C, C++, Objective-C, Pascal
- Debug locally or remotely
- Can run on most popular Unix and Microsoft Windows variants

DDD - Display Data Debugger

- Graphical front-end for command-line debuggers (e.g. gdb)
- Can display data structure and contents, including dependencies
- Can plot numerical data



GDB Example

```c
#include <stdio.h>

int main()
{
  /* Initialize variables */
  const int x     = 10;
  int factorial =  1;
  /* Loop through all x */
  for (int i=0; i<=x; i++)
  {
    factorial *= i;
  }
  printf("%d! is %d\n", x, factorial);
  return 0;
}
```

GDB Example - Run program

- Compile: `gcc -g -std=c99 -o A7 A7.c`
- Run: `10! is 0` → bug
- Debug:

```
$ gdb A7
(gdb) b 11
Breakpoint 1 at 0x400501: file A7.c, line 11.
(gdb) r
Starting program: /home/login/path/A7
Breakpoint 1, main () at A7.c:11
11          factorial *= i;
(gdb) print factorial
$2 = 1
(gdb) print i
$1 = 0
condition 1 i==5
(gdb) c
Breakpoint 1, main () at A7.c:11
11          factorial *= i;
(gdb) print i
$1 = 5
(gdb) print factorial
$3 = 0
```

GDB Example - Attach to process

- Find xpdf process number: `ps afxuwww | grep xpdf`
- Attach gdb:

```
$ gdb -p 5256
...
Attaching to process 5256
Reading symbols from /usr/bin/xpdf.bin...(no debugging symbols found)...done.
Reading symbols from /usr/lib/libt1.so.5...(no debugging symbols found)...done.
...
0x00007f64dcf69b9f in poll () from /lib/libc.so.6
(gdb) bt
#0  0x00007f64dcf69b9f in poll () from /lib/libc.so.6
#1  0x00007f64de9269fa in _XtWaitForSomething () from /usr/lib/libXt.so.6
#2  0x00007f64de927b03 in XtAppNextEvent () from /usr/lib/libXt.so.6
#3  0x00007f64de91ac9b in XtAppMainLoop () from /usr/lib/libXt.so.6
#4  0x00000000004aa6b6 in ?? ()
#5  0x00007f64dcec11a6 in __libc_start_main () from /lib/libc.so.6
#6  0x0000000000406329 in ?? ()
```

GDB Example - Dead process

- Run normally: `Segmentation fault (core dumped)`
- Run gdb on "core dump":

```
$ gdb testit core
Core was generated by ‘testit’.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /usr/lib/libstdc++-libc6.1-1.so.2...done.
Reading symbols from /lib/libm.so.6...done.
Reading symbols from /lib/libc.so.6...done.
Reading symbols from /lib/ld-linux.so.2...done.
#0  0x804851a in main () at testit.c:10
10              temp[3]='F';
```

Look at declaration of temp: `char *temp = "Paras";`

GDB Command Reference

gdb program [core]	debug program [using coredump]
r [arglist]	start program [with arglist]
p expr	print value of expr
set var=expr	set variable to value of expr
c	continue
n	next line, step over function calls
s	next line, step into function calls
b [file:]function	set breakpoint at function [in file]
b [file:]line	set breakpoint at line [in file]
b ... **if** expr	break conditionally on nonzero expr
watch expr	set watchpoint for expr
info break	show defined breakpoints
info watch	show defined watchpoints
clear ...	clear breakpoint
bt	backtrace: show program stack
up n	select frame n frames up
down n	select frame n frames down
info args	arguments of selected frame
info locals	local variables of selected frame
q	quit

Profiling

Profiling

Investigation of a program's behavior using information gathered as the program executes, usually with the goals

- to increase its overall speed
- to decrease its memory requirement

Profiler measures typically the frequency and duration of function calls and can output:

- A statistical summary of the events observed (a profile)
- A stream of recorded events (a trace)
- An ongoing interaction with the hypervisor

Can in simple cases done by hand by inserting

- print statements and directly observing output at runtime
- user-built timers around interesting program parts

Profiler types

Statistical profilers

- Operate by sampling: probes the target program's program counter at regular intervals
- Typically less numerically accurate and specific
- Allow for near full speed
- Can often provide a more accurate picture
  - not as intrusive to the target program
  - don't have as many side effects

Instrumenting profilers

- Instrument the target program with additional instructions
- Can cause changes in the performance of the program
- Can be on just one machine instruction on some targets
- Impact of instrumentation can often be eliminated from the results

An example: gprof

gprof: statistical and instrumenting profiler

- Compile program to generate profile data: `-g -pg`
- Execute program to generate profile data: `gmon.out`
- Run gprof:
  `gprof options [executable [profile-data-files ...]]`

Some gprof options:

-a	suppresses the printing of statically declared (private) functions
-e function name	don't print information about the function name
-f function name	limit the call graph to the function name and its children

An example: gprof

Flat profile:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
```

An example: gprof

Call graph:

```
index   called          name
    72384/72384         sym_id_parse [54]
[3]   72384             match [3]
----------------------
          4/9052        cg_tally [32]
       3016/9052        hist_print [49]
       6032/9052        propagate_flags [52]
[4]    9052             sym_lookup [4]
----------------------
       5766/5766        core_create_function_syms [41]
[5]    5766             core_sym_class [5]
----------------------
         24/1537        parse_spec [19]
       1513/1537        core_create_function_syms [41]
[6]    1537             sym_init [6]
----------------------
       1511/1511        core_create_function_syms [41]
[7]    1511             get_src_info [7]
----------------------
          2/1510        arc_add [31]
       1508/1510        cg_assemble [38]
[8]    1510             arc_lookup [8]
----------------------
       1509/1509        cg_dfn [15]
[9]    1509             is_numbered [9]
----------------------
       1508/1508        propagate_flags [52]
[10]   1508             inherit_flags [10]
----------------------
       1508/1508        cg_dfn [15]
[11]   1508             is_busy [11]
```