Computer System Organization
Today’s agenda

Overview of how things work

- Compilation and linking system
- Operating system
- Computer organization
A software view

- User Interface
  - Users
    - Standards utility programs (shell, editors, compilers, etc)
      - Standard library (open, close, read, write, fork, etc)
        - UNIX operating system (process management, memory management, the file system, I/O, etc)
          - Hardware (CPU, memory, disks, terminals, etc)
Hello, world!

```
#include <stdio.h>
#define FOO 4
int main() {
    printf("hello, world %d\n", FOO);
}
```
The Compilation system

gcc is the *compiler driver*

gcc invokes several other *compilation phases*

- Preprocessor
- Compiler
- Assembler
- Linker

**What does each one do? What are their outputs?**

<table>
<thead>
<tr>
<th>Preprocessor</th>
<th>Compiler</th>
<th>Assembler</th>
<th>Linker</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello.c</td>
<td>hello.i</td>
<td>hello.s</td>
<td>hello.o</td>
</tr>
<tr>
<td>Program Source</td>
<td>Modified Source</td>
<td>Assembly Code</td>
<td>Object Code</td>
</tr>
<tr>
<td>Executable Code</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Preprocessor

First, gcc compiler driver invokes cpp to generate expanded C source

- cpp just does text substitution
- Converts the C source file to another C source file
- Expands #defines, #includes, etc.
- Output is another C source file

```c
#include <stdio.h>
#define FOO 4
int main(){
    printf("hello, world %d\n", FOO);
}
```

```c
... extern int printf (const char *__restrict __format, ...);
...
int main() {
    printf("hello, world %d\n", 4);
}
```
Preprocessor

Included files:

```cpp
#include <foo.h>
#include "bar.h"
```

Defined constants:

```cpp
#define MAXVAL 40000000
```

By convention, all capitals tells us it’s a constant, not a variable.

Macros:

```cpp
#define MIN(x,y)   ((x)<(y) ? (x):(y))
#define RIDX(i, j, n)  (((i) * (n) + (j))
```
Preprocessor

Conditional compilation:

```c
ifdef ...
or ifdef( ...
endif
```

- Code you think you may need again (e.g. debug print statements)
  - Include or exclude code based on `#define`, `#ifdef`
  - `gcc -D DEBUG equivalent to #define DEBUG`
  - More readable than commenting code out

http://thefengs.com/wuchang/courses/cs201/class/03/def
Preprocessor

Portability

- Compilers with “built in” constants defined
- Use to conditionally include code
  - Operating system specific code
    ```
    #if defined(__i386__) || defined(WIN32) || ...
    ```
  - Compiler-specific code
    ```
    #if defined(__INTEL_COMPILER)
    ```
  - Processor-specific code
    ```
    #if defined(__SSE__)
    ```
Next, gcc compiler driver invokes cc1 to generate assembly code

- Translates high-level C code into assembly
  - Variable abstraction mapped to memory locations and registers
  - Logical and arithmetic functions mapped to underlying machine opcodes
extern int printf (const char *__restrict __format, ...);

int main() {
    printf("hello, world %d\n", 4);
}

.section .rodata
.LC0:
    .string "hello, world %d\n"
.text
main:
    pushq  %rbp
    movq   %rsp, %rbp
    movl   $4, %esi
    movl   $.LC0, %edi
    movl   $0, %eax
    call   printf
    popq   %rbp
    ret
Assembler

Next, gcc compiler driver invokes as to generate object code

- Translates assembly code into binary object code that can be directly executed by CPU
.section .rodata
.LC0:
.string "hello, world %d\n"
.text
main:
    pushq %rbp
    movq %rsp, %rbp
    movl $4, %esi
    movl $.LC0, %edi
    movl $0, %eax
    call printf
    popq %rbp
ret

Hex dump of section `.rodata':
  0x004005d0 01000200 68656c6c 6f2c2077 6f726c64 ....hello, world
  0x004005e0 00                          %d..

Disassembly of section .text:
000000000040052d <main>:
  40052d: 55                pushq %rbp
  40052e: 48 89 e5          mov %rsp, %rbp
  400531: be 04 00 00 00    movl $4, %esi
  400536: bf d4 05 40 00    movl $0x4005d4, %edi
  40053b: b8 00 00 00 00    movl $0x0, %eax
  400540: e8 cb fe ff ff    callq 400410 <printf@plt>
  400544: 5d                popq %rbp
  400546: c3                retq
Linker

Finally, gcc compiler driver calls linker (ld) to generate executable

- Merges multiple relocatable (.o) object files into a single executable program
- Copies library object code and data into executable
- Relocates relative positions in library and object files to absolute ones in final executable
Linker

Resolves external references

- **External reference**: reference to a symbol defined in another object file (e.g. `printf`)
- Updates all references to these symbols to reflect their new positions.
  - References in both code and data
    ```c
    printf();  /* reference to symbol printf */
    int *xp=&x;  /* reference to symbol x */
    ```

This is the executable program
Benefits of linking

Modularity and space

- Program can be written as a collection of smaller source files, rather than one monolithic mass.
- Can build libraries of common functions (more on this later)
  - e.g., Math library, standard C library
- Compilation efficiency
  - Change one source file, compile, and then relink.
  - No need to recompile other source files.
- Space efficiency
  - Libraries of common functions can be aggregated into a single file used by all programs
Summary of compilation process

*Compiler driver* (cc or gcc) coordinates all steps

- Invokes preprocessor (cpp), compiler (cc1), assembler (as), and linker (ld).
- Passes command line arguments to appropriate phases
Creating and using libc

Archiver (ar)

`ar rs libc.a atoi.o printf.o ... random.o`

C standard library archive of relocatable object files concatenated into one file

Linker (ld)

Executable object file (with code and data for libc functions needed by `p1.c` and `p2.c` copied in)
LibC libraries

**libc.a (the C standard library)**
- 5 MB archive of more than 1000 object files.
- I/O, memory allocation, signals, strings, time, random numbers

**libm.a (the C math library)**
- 2 MB archive of more than 400 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t /usr/lib/x86_64-linux-gnu/libc.a | sort
... 
fork.o 
...
fprintf.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
...
```

```
% ar -t /usr/lib/x86_64-linux-gnu/libm.a | sort
...
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_asin.o
e_acosl.o
e_asinl.o
e_asinf.o
...
Creating your own static libraries

Archiver (ar)

mathtest.c

Translator

mathtest.o

cubeit.c

Translator

cubeit.o

Archiver (ar)

ar rs libmyutil.a squareit.o cubeit.o

Archive of your object files concatenated into one file

Linker (ld)

P

executable object file (with code and data for libmyutil functions needed by mathtest.c copied in)
Creating your own static libraries

Suppose you have utility code in squareit.c and cubeit.c that all of your programs use

- Create a library libmyutil.a using ar and ranlib and link library in statically

  libmyutil.a : squareit.o cubeit.o
  
ar rvu libmyutil.a squareit.o cubeit.o
  ranlib libmyutil.a

- Compile your program that uses library calls and link in library statically

  gcc -o mathtest mathtest.c -L. -lmyutil

  ● Note: Only the library code “mathtest” needs from libmyutil is copied directly into binary

  ● List functions in binary or library

    nm libmyutil.a

http://thefengs.com/wuchang/courses/cs201/class/03/libexample
Problems with static libraries

Multiple copies of common code on disk

- “gcc program.c -lc” creates an a.out with libc object code copied into it (libc.a)
- Almost all programs use libc!
- Large number of binaries on disk with the same code in it
Libraries and linking

Two types of libraries

- **Static libraries**
  - Library of code that linker copies into the executable at compile time

- **Dynamic shared object libraries**
  - Code loaded at run-time by system loader upon program execution
Dynamic libraries

Have binaries compiled with a reference to a library of shared objects on disk

- Libraries loaded at run-time from file system rather than copied in at compile-time
- "ldd <binary>" to see dependencies
  - gcc flag "-shared" to create dynamic shared object files (.so)
- Caveat
  - How does one ensure dynamic libraries are present across all run-time environments?
  - Static linking (via gcc’s -static flag) to create self-contained binaries and avoid problems with DLL versions
Dynamically Linked Shared Libraries

Translators (cc1, as)  Translators (cc1, as)

m.c  a.c

m.o  a.o

Linker (ld)

libc.so

Loader/Dynamic Linker (ld-linux.so)

libc.so functions called by m.c and a.c are loaded, linked, and (potentially) shared among processes.

Partially linked executable p (on disk)

Fully linked executable p' (in memory)
The Complete Picture

Translator

m.o

Translator

a.o

Static Linker (ld)

libwhatever.a

Static Linker (ld)

libc.so libm.so

Loader/Dynamic Linker
(1d-linux.so)

p

p'
The (Actual) Complete Picture

Dozens of processes use libc.so

- Each process reads libc.so from disk and loads private copy into address space
- Multiple copies of the *exact* code resident in memory for each!
- Modern operating systems keep one copy of library in read-only memory
  - Single shared copy
  - Shared virtual memory (page-sharing) to reduce memory use
Program execution

gcc/cc output an executable in the ELF format (Linux)

- Executable and Linkable Format

Standard unified binary format for

- Relocatable object files (.o),
- Shared object files (.so)
- Executable object files

Equivalent to Windows Portable Executable (PE) format
ELF Object File Format

**ELF header**
- Magic number, type (.o, exec, .so), machine, byte ordering, etc.

**Program header table**
- Page size, addresses of memory segments (sections), segment sizes.

.**text section**
- Code

.**data section**
- Initialized (static) data

.**bss section**
- Uninitialized (static) data
- “Block Started by Symbol”

**Section header table**
- (required for relocatables)
ELF Object File Format (cont)

**.symtab section**
- Symbol table
- Procedure and static variable names
- Section names and locations

**.rela.text section**
- Relocation info for `.text` section

**.rela.data section**
- Relocation info for `.data` section

**.debug section**
- Info for symbolic debugging (`gcc -g`)

---

<table>
<thead>
<tr>
<th>ELF header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program header table (required for executables)</td>
</tr>
<tr>
<td><code>.text</code> section</td>
</tr>
<tr>
<td><code>.data</code> section</td>
</tr>
<tr>
<td><code>.bss</code> section</td>
</tr>
<tr>
<td><code>.symtab</code></td>
</tr>
<tr>
<td><code>.rela.text</code></td>
</tr>
<tr>
<td><code>.rela.data</code></td>
</tr>
<tr>
<td><code>.debug</code></td>
</tr>
</tbody>
</table>

Section header table (required for relocatables)
Relocation code example

**Symbols for code and data**

- Definitions and references
- References can be either *local* or *external*.
- Addresses of references must be resolved when loaded.

```c
int e = 7;
extern int a();
int main() {
    int r = a();
    exit(0);
}

extern int e;
int *ep = &e;
int x = 15;
int y;

int a() {
    return *ep + x + y;
}
```

- **Def of local symbol e**
- **Ref to external symbol exit (defined in libc.so)**
- **Def of local symbol ep**
- **Ref to external symbol a**
- **Defs of local symbols x and y**
- **Ref to external symbol e**
- **Defs of local symbols ep, x, y**
- **Refs of local symbols ep, x, y**
Merging Object Files into an Executable Object File

```c
int e = 7;
extern int a();

int main() {
    int r = a();
    exit(0);
}

extern int e;
int *ep = &e;
int x = 15;
int y;

int a() {
    return *ep + x + y;
}
```

Object Files

- system code
- system data

- main()
  - int e = 7

- a()
  - int *ep = &e
  - int x = 15
  - int y

Executable Object File

- headers
- system code
- main()
- a()
- more system code

- .text
- .data
- .bss
- .symtab
- .debug

- .text
- .data
- .bss
- .symtab
- .debug
Relocation

Compiler does not know where code will be loaded into memory upon execution

- Instructions and data that depend on location must be “fixed” to actual addresses
- i.e. variables, pointers, jump instructions

.rel.text section

- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying
- (e.g. a() in m.c)

.rel.data section

- Addresses of pointer data that will need to be modified in the merged executable
- (e.g. ep in a.c)

readelf -a
Relocation example

What is in .text, .data, .rela.text, and .rela.data?

readelf -a a.o ; .rela.text contains ep, x, and y from a()
                ; .rela.data contains e to initialize ep
objdump -d a.o ; Shows relocations in .text
objdump -d m  ; After linking, references placed at fixed
               ; relative offset to RIP

http://thefengs.com/wuchang/courses/cs201/class/03/elf_example
Relocation example

```c
int e = 7;
extern int a();

int main() {
    int r = a();
    exit(0);
}
```

```c
extern int e;
int *ep = &e;
int x = 15;
int y;

int a() {
    return *ep + x + y;
}
```

readelf -a m.o ; .rela.text contains a and exit from main()
objdump -d m.o ; Show relocations in .text
objdump -d m ; After linking, symbols resolved in <main>
              ; for <a> and <exit>
Operating system

Program runs on top of operating system that implements abstract view of resources

- Files as an abstraction of storage and network devices
- System calls an abstraction for OS services
- Virtual memory a uniform memory space abstraction for each process
  - Gives the illusion that each process has entire memory space
- A process (in conjunction with the OS) provides an abstraction for a virtual computer
  - Slices of CPU time to run in
  - CPU state
  - Open files
  - Thread of execution
  - Code and data in memory

Protection

- Protects the hardware/itself from user programs
- Protects user programs from each other
- Protects files from unauthorized access
Program execution

The operating system creates a process.
- Including among other things, a virtual memory space

System loader reads program from file system and loads its code into memory
- Program includes any statically linked libraries
- Done via DMA (direct memory access)

System loader loads dynamic shared objects/libraries into memory

Links everything together and then starts a thread of execution running
- Note: the program binary in file system remains and can be executed again
- Program is a cookie recipe, processes are the cookies
Loading Executable Binaries

Executable object file for example program p

<table>
<thead>
<tr>
<th>Address</th>
<th>ELF header</th>
<th>Program header table (required for executables)</th>
<th>.text section</th>
<th>.data section</th>
<th>.bss section</th>
<th>.symtab</th>
<th>.rel.text</th>
<th>.rel.data</th>
<th>.debug</th>
<th>Section header table (required for relocatables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Process image

- init and shared lib segments
  - .text segment (r/o) at 0x0408494
  - .data segment (initialized r/w) at 0x040a010
  - .bss segment ( uninitialized r/w) at 0x040a3b0

Virtual addr:
- 0x04083e0
- 0x0408494
- 0x040a010
- 0x040a3b0

Loading Executable Binaries
Where are programs loaded in memory?

An evolution....

Primitive operating systems
- Single tasking.
- Physical memory addresses go from zero to N.

The problem of loading is simple
- Load the program starting at address zero
- Use as much memory as it takes.
- Linker binds the program to absolute addresses at compile-time
- Code starts at zero
- Data concatenated after that
- etc.
Where are programs loaded, cont’d

Next imagine a multi-tasking operating system on a primitive computer.

- Physical memory space, from zero to N.
- Applications share space
- Memory allocated at load time in unused space
- Linker does not know where the program will be loaded
- Binds together all the modules, but keeps them relocatable

How does the operating system load this program?

- Not a pretty solution, must find contiguous unused blocks

How does the operating system provide protection?

- Not pretty either
Where are programs loaded, cont’d

Next, imagine a multi-tasking operating system on a modern computer, with hardware-assisted virtual memory (Intel 80286/80386)

OS creates a virtual memory space for each program.
- As if program has all of memory to itself.

Back to the simple model
- The linker statically binds the program to virtual addresses
- At load time, OS allocates memory, creates a virtual address space, and loads the code and data.
- Binaries are simply virtual memory snapshots of programs (Windows .com format)
Modern linking and loading

Reduce storage via dynamic linking and loading

- Single, uniform VM address space still
- But, library code must vie for addresses at load-time
  - Many dynamic libraries, no fixed/reserved addresses to map them into
  - Code must be relocatable again
  - Useful also as a security feature to prevent predictability in exploits (Address-Space Layout Randomization)
Extra
More on the linking process (ld)

Resolves multiply defined symbols with some restrictions

- Strong symbols = initialized global variables, functions
- Weak symbols = uninitialized global variables, functions used to allow overrides of function implementations
- Simulates inheritance and function overriding (as in C++)

Rules
- Multiple strong symbols not allowed
- Choose strong symbols over weak symbols
- Choose any weak symbol if multiple ones exist
Modern 64-bit memory map

48-bit canonical address space implementations

- Reduce width of addresses to make page-tables smaller
- Kernel addresses have high-bit set

```
0x7ffe96110000
  user stack
    (created at runtime)
  %esp (stack pointer)

0x7f81bb0b5000
  memory mapped region for shared libraries
  brk

0x04000000
  read/write segment
  (\.data, .bss)

0x00400000
  read-only segment
  (\.init, .text, .rodata)

0xffffffffffffffff
  reserved for kernel
    (code, data, heap, stack)

0xfffff80000000000
  memory invisible to user code

cat /proc/self/maps
```