Static and Dynamic Libraries

- **LIBRARIES** are commonly used routines stored as a concatenation of "Object files". A global symbol table is maintained for the entire library with entry points for each routine.

- When a routine in a LIBRARY is referenced by an assembly module, the routine’s address is resolved by the **LINKER**, and the appropriate code is added to the executable. This sort of linking is called **STATIC** linking.

- Many programs use common libraries. It is wasteful of both memory and disk space to include the same code in multiple executables. The modern alternative to **STATIC** linking is to allow the **LOADER** and **THE PROGRAM ITSELF** to resolve the addresses of libraries routines. This form of lining is called **DYNAMIC** linking (e.x. .dll).
Dynamically Linked Libraries

- C call to library function:
  
  ```
  printf("sqr[%d] = %d\n", x, y);
  ```

- Assembly code
  
  ```
  mov R0,#1
  mov R1,ctrlstring
  ldr R2,x
  ldr R3,y
  mov IP,__stdio__
  mov LR,PC
  ldr PC,[IP,#16]
  ```

How does dynamic linking work?

Why are we loading the PC from a memory location rather than branching?
Dynamically Linked Libraries

- Lazy address resolution:

```assembly
sysload: stmfd sp!, [r0-r10, lr]

; check if stdio module
; is loaded, if not load it

; backpatch jump table
mov r1, _stdio_
mov r0, dfopen
str r0, [r1]
mov r0, dfclose
str r0, [r1, #4]
mov r0, dfputc
str r0, [r1, #8]
mov r0, dfgetc
str r0, [r1, #12]
mov r0, dfprintf
str r0, [r1, #16]
```

Because, the entry points to dynamic library routines are stored in a TABLE. And the contents of this table are loaded on a "as needed" basis!

Before any call is made to a procedure in "stdio.dll"

```assembly
.globl __stdio__:
__stdio__:
fopen: .word sysload
fclose: .word sysload
fgetc: .word sysload
fputc: .word sysload
fprintf: .word sysload
```

After the first call is made to any procedure in "stdio.dll"

```assembly
.globl __stdio__:
__stdio__:
fopen: dfopen
fclose: dclose
fgetc: dfgetc
fputc: dfputc
fprintf: dprintf
```
Modern Languages

Intermediate "object code language"

- High-level, portable (architecture independent) program description
- PORTABLE mnemonic program description with symbolic memory references
- An application that EMULATES a virtual machine. Can be written for any Instruction Set Architecture. In the end, machine language instructions must be executed for each JVM bytecode

Java program ➔ Compiler ➔ JVM bytecodes ➔ "Library Routines" ➔ Interpreter

In the end, machine language instructions must be executed for each JVM bytecode.
Modern Languages

Intermediate “object code language”

High-level, portable (architecture independent) program description

Java program

Compiler

PORTABLE mnemonic program description with symbolic memory references

JVM bytecodes

While interpreting on the first pass the JIT keeps a copy of the machine language instructions used. Future references access machine language code, avoiding further interpretation

“Library Routines”

JIT Compiler

Machine code

Today’s JITs are nearly as fast as a native compiled code.
Assembly? Really?

- In the early days compilers were dumb
  - literal line-by-line generation of assembly code of "C" source
  - This was efficient in terms of S/W development time
    - C is portable, ISA independent, write once- run anywhere
    - C is easier to read and understand
    - Details of stack allocation and memory management are hidden
  - However, a savvy programmer could nearly always generate code that would execute faster

- Enter the modern era of Compilers
  - Focused on optimized code-generation
  - Captured the common tricks that low-level programmers used
  - Meticulous bookkeeping (i.e. will I ever use this variable again?)
  - It is hard for even the best hacker to improve on code generated by good optimizing compilers
Next Time

- Compiler code optimization
- We look deeper into the Rabbit hole
What would a compiler do?

```c
#include <stdio.h>
int main(void)
{
    int count;
    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.\n");
    return 0;
}
```

Today we’ll look at the assembly code that compiler’s generate...
Code generation

Example C code:

```c
int array[10];
int total;

int main( ) {
    int i;
    total = 0;
    for (i = 0; i < 10; i++) {
        array[i] = i;
        total = total + i;
    }
}
```
Code we might write

```assembly
.word 0x03fffffffc, main

array: .space 10
total: .space 1

main:                             ; int main() {
    sub sp,sp,#4          ;   int i;
    mov r0,#0            ;   total = 0;
    str r0,total          ;   for (i = 0; i < 10; i++) {
    str r0,[sp]           ;
    b _L02

_L01:
    mov r1,#array
    str r0,[r1,r0,lsl #2] ;   array[i] = i;
    ldr r1,total
    add r1,r1,r0
    str r1,total          ;   total = total + i;
    add r0,r0,#1
    str r0,[sp]

_L02:
    cmp r0,#10
    blt _L01             ;   }
    add sp,sp,#4
    bx lr

98 that's not so bad
```
AN ONLINE ARM7 COMPILER

Available at: http://csbio.unc.edu/mcmillan/index.py?run=arm
Unoptimized Compiler Output

.word 0x03fffffc, main
.array: .space 10
.total: .space 1
.globl main
main:
str fp, [sp, #-4]!
add fp, sp, #0
sub sp, sp, #12
ldr r3, _L4
mov r2, #0
str r2, [r3, #0]
mov r3, #0
str r3, [fp, #-8]
b _L2

_L3:
ldr r3, _L4+4
ldr r2, [fp, #-8]
ldr r1, [fp, #-8]
str r1, [r3, r2, asl #2]
ldr r3, _L4
ldr r2, [r3, #0]
ldr r3, [fp, #-8]
add r2, r2, r3
ldr r3, _L4
str r2, [r3, #0]
ldr r3, [fp, #-8]
add r3, r3, #1
str r3, [fp, #-8]

_L2:
ldr r3, [fp, #-8]
cmp r3, #9
ble _L3
mov r0, r3
add sp, fp, #0
ldmfd sp!, {fp}
bx lr

_L5:

.L4:
.word total

.L4:
.word array

Why is this code so bad?
Because it generated for debugging.
Essentially, each line is translated directly.

175, not a good day.
OPTIMIZED CODE

It even relaid out the variables so that all writes are sequential.

.main:
  ldr  r2, _L4
  mov  r3, #0

_L2:
  str  r3, [r2, #4]!
  add  r3, r3, #1
  cmp  r3, #10
  bne  _L2
  mov  r2, #45
  ldr  r3, _L4+4
  str  r2, [r3, #0]

* bx  lr

_L5:

_L4:
  .word  array-4
  .word  total

total:  .space 1
array:  .space 10

45, best ever!
Next Time

We look into the hardware