OBJECT FILES IN LLVM

Michael Spencer

What was my goal in doing this object file project...

- I wanted a fast, free, and cross platform linker
- Integrated binary tool-chain
 - Why not replace all of binutils?
- Needed to share object file handling code throughout the tool-chain
 - Library!

What is an Object File?

- An Object File is a structured collection of binary data
 - □ PE/COFF, ELF, MachO, ...
 - Translation Units, Executables, and Dynamic Libraries
 - Archives
 - Raw ".bin" files
 - LLVM bitcode
- Different formats provide a disjoint set of features

What is an Object File? (cont)

- There is a common subset of features
 - Symbols
 - Sections
 - Segments
 - Relocations
- Every other feature is based off of these
 - eg C++ Construction and Destruction of static objects must occur before main is run. This requires special handling by the linker and loader.

Current Technology

- Not much has changed with object files
- Libraries
 - □ libbfd Binary File Descriptor
 - 20 years old?
- □ Linkers
 - □ gnu-ld slow and GPL
 - gold ELF only and GPL
 - □ link.exe slow, COFF only, and proprietary
 - Various system linkers

LLVM's Current Support

- The Machine Code (MC) library handles assembling and writing out object files
- Supports various formats
 - COFF
 - ELF
 - MachO
- Assembler specific, not designed for generic object file handling

Why Object Files?

- □ Freedom
 - Not GPL
 - Not Proprietary
- Cross Platform Consistency
 - Toolset
 - API
- Performance
 - IPO Through Shared Libraries
 - JIT Caching
 - Integrated Linker

More Than Just a Linker

- Make it easy to add link time features that would normally require a "prelinker"
 - Constructors and Destructors
 - Already handled because of C++, but required changes to linkers and loaders.
 - C++ Open Methods
 - LTO

The LLVM Object File Library

- Goals
 - Library based
 - Unified API for various formats
 - Access to details when needed
 - Provide replacements for all of binutils
 - Speed

Architecture

Normalization

Serialization

MC

System

Support

Architecture

- Layered
 - Low Level: Serialization
 - Base library that reads and writes
 - Depends only on the LLVM System and Support libraries
 - No interpretation is performed at this point
 - Provides symbols, sections, segments, and relocations
 - eg. In ELF the relocation section shows up here, even though other object files store relocations differently
 - Useful for tools like objdump and nm

Architecture

- High Level: Normalization
 - Interprets data into a common representation
 - Provides a common API for tools to use to access the data provided by the serialization layer
 - Understands and can perform relocations, layout, etc...
 - Common form -> {ELF, COFF, MachO} -> Common Form will end up with what you started with (if the format supports the feature).

How is libobj faster?

□ Zero-Copy

- libbfd copies non-section data from the memory mapped object file into a struct
- □ *libobj* returns an object that contains a reference into the memory mapped object file and knows how to extract data from it
- This lowers the physical memory usage and therefore increases cache reuse and reduces swap file usage

How is libobj faster? (cont)

- Read data a field at a time
 - Object File formats are generally specified in terms of
 C structs with a specified endianness and alignment
 - libbfd deals with these issues by reading a byte at a time
 - □ libobj speeds this up by reading an entire field at a time and byte swapping if necessary. It reverts to reading a byte at a time only when the format does not guarantee alignment and the host does not support misaligned loads

Implementation

- Endianness and Alignment
 - Transparently and quickly dealt with using packed_endian_specific_integral

```
// Read in some data from disk.
uint8_t *data = read_binary_data();
                                                            calll _read_binary_data
// Read a little endian int32
                                                            movl (%eax), %eax
int32_t value = *reinterpret_cast<little32_t*>(data);
                                                            movl %eax, 4(%esp)
                                                            movl 4(%eax), %eax
// Read a big endian int32
                                                            bswapl %eax
int32_t be_value = *reinterpret_cast<big32_t*>(data + 4);
                                                            movl %eax, 8(%esp)
// Print the result
                                                            movl $L_.str, (%esp)
printf("%d-%d", value, be_value);
                                                            call _printf
```

Implementation

- Endianness and Alignment (cont...)
 - These can be combined into a POD struct.

```
typedef struct {
 aligned_ulittle32_t r_offset;
 aligned ulittle32 tr info;
} Elf32_Rel;
Elf32_Rel *reloc =
         (Elf32_Rel*)read_binary_data(location_of_relocation_entry);
printf("Reloc{ addr: %p, info: %u }",
         (void*)(reloc->r offset),
         unsigned(reloc->r info));
```

Performance

- I added object file support to llvm-nm and tested it vs binutils-nm.
- 2x faster on Linux
- □ ~30x faster on Windows (vs. nm via MinGW)

Current Status

- I have currently implemented symbols and sections in the Serialization layer for COFF and ELF
- I have also worked on some tools
 - □ IIvm-nm
 - Modified to support object files
 - Ilvm-objdump
 - Added with support for disassembly via MC
- Working on getting patches into trunk, please review!

OBJECT FILES IN LLVM

Questions?