What makes LLD so fast?

Peter Smith, Linaro

Contents and Introduction

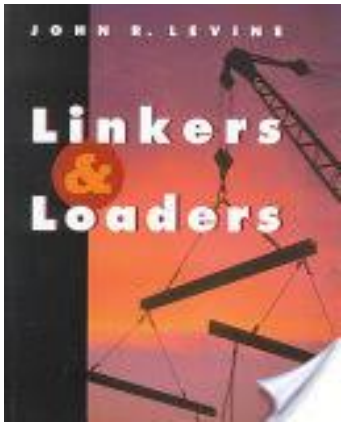
- What we are covering Today
  - What job does a linker do?
  - How fast is LLD compared to GNU ld and gold?
  - LLD design and implementation choices.
  - Non technical factors.
- My history with Linkers
  - Working in the Linaro TCWG team on LLVM.
  - Have made contributions to LLD, focusing on Arm and AArch64.
  - Have previously worked on Arm's embedded linker armlink.

What job does a linker do?

- Components of an ELF file
- Generic linker design
- Finding content
- Layout
- Relocation

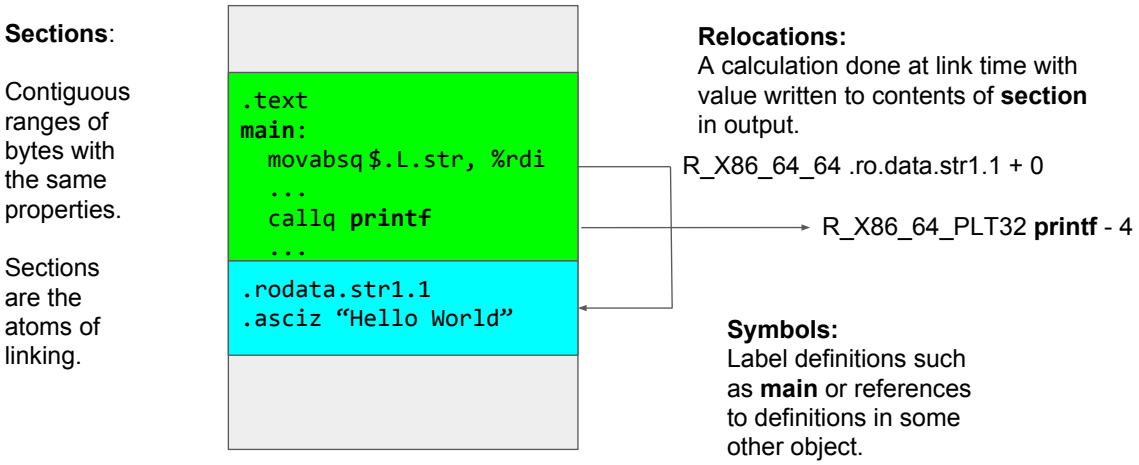
What is a Linker?

- Levine defines in Linkers and Loaders
  - “Binds more abstract names to more concrete names, which permits programmers to write code using more abstract names”
  - In general concrete representations are higher performance but less flexible
- In practice a tool that glues together the outputs of separate compilation into a single output



A linker is in many ways a slightly more sophisticated version of cat. The linker must copy the necessary data from all the input objects, rearrange it and resolve fixups. It is worth bearing this in mind when complaining that the linker is slow, just doing a cat of all the object files can also take a significant amount of time, particularly when debug information is involved.

Important components of a relocatable object file



The ELF file format is defined in
http://www.sco.com/developers/gabi/latest/contents.html

There are 3 types of ELF file, denoted by a field in the ELF header: ET_REL (Relocatable Object File), ET_EXEC (Executables) and ET_DYN (Shared Libraries and position independent executables). Relocatable object files have a Section Level View, Executables and Shared Libraries have a Segment Level View used by a program loader and usually retain a Section Level view.

A relocatable object file is an incomplete portion of a program, such as the output of the compiler from a single source file, that needs to be linked with other relocatable object files to form the program.
The three most important concepts in ELF are Sections, Symbols and Relocations.

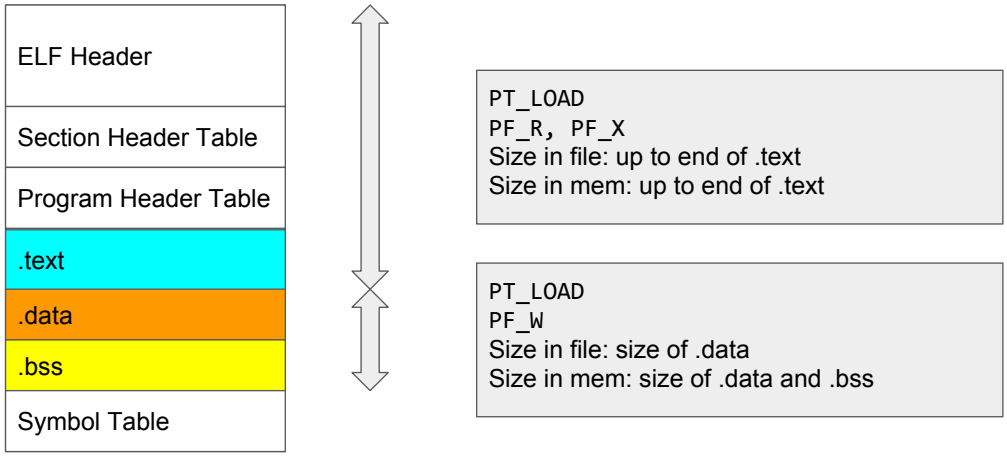
Sections:
Represent a contiguous range of bytes that a linker must handle as a single indivisible chunk. In general the linker does not understand the contents of a section so splitting it up may not be safe. The name of a section is not significant within the specification, however there are conventions such as .text for executable code, .data for read-write data and .bss for zero initialized data.

Symbols:
Can be a definition (labels something in this object file), or a reference (something defined in another object file)

Most commonly defined as an offset from the base of a Section (Section Relative), but can be absolute, in which case they represent a number. A typical use for a symbol will be a label for the start of a function.

Relocations:
Instructions to the linker to fixup the contents of a section when all addresses are known. In the example in the slide the compiler does not know the address of printf so it leaves the field in the instruction blank, at the end of the linking process the linker knows what the address is so it can substitute the correct value. As the linker does not understand the section contents the relocation directives give the linker precise instructions in how to modify the contents. The vast majority of relocations are defined relative to a symbol.

Important components of an executable ELF file

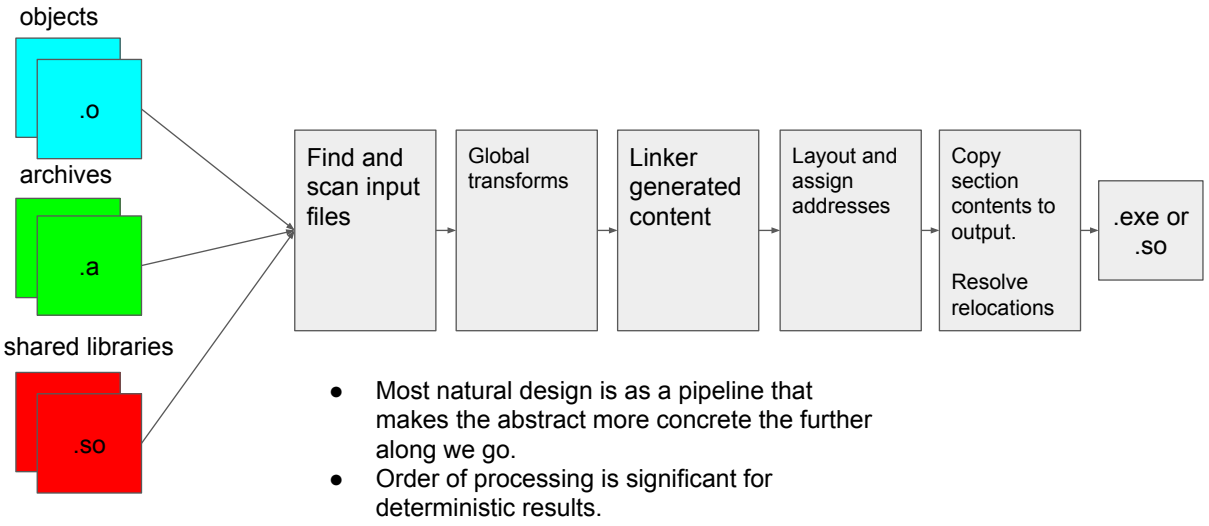


ELF executables and shared objects use segments to describe ranges of the ELF file. Segments are described by program headers and contain information that a dynamic loader or an operating system might use. For example the flags PF_R, PF_W and PF_X describe the properties of the memory page that an OS must use for the address range.

In theory executables and shared objects do not need the section level view, but in practice unstripped ELF files will retain the section level view as it aids disassembly and other post-linker ELF processing tools.

In contrast to sections, segments may overlap so we may have a large segment describing the read-only part of the ELF file, and a smaller segment that describes a sub-range of addresses with a specific property such as the dynamic segment.

Generic Linker design



The natural design for a linker is as a pipeline, with each stage making the representation more specific and low-level. At the end of the link every symbol needs to be defined at a specific address and all relocations are resolved.
At a high level the main stages are:

- Read in all content, doing local transformations as we go
- Do global transformations that require global knowledge, for example garbage collection
- Generate any linker generated sections such as the PLT and GOT
- Arrange the sections (layout) and assign an address to each section (and hence symbol)
- Write the output file

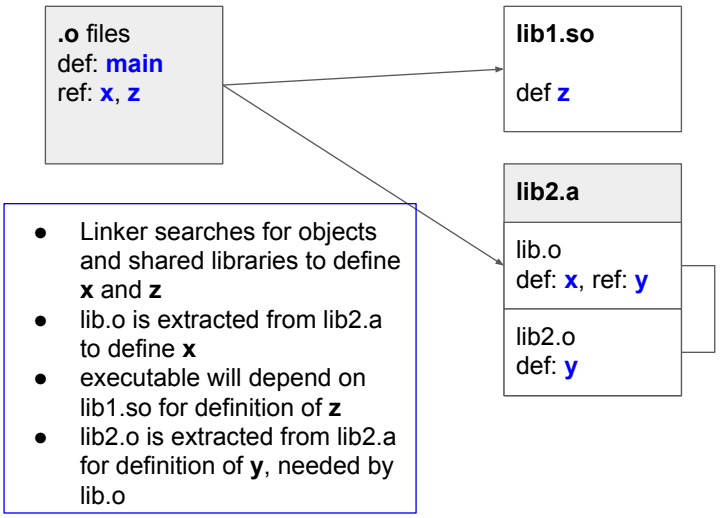
There are considerable variations in design available to a linker writer. For example:

- Does the linker try to abstract away the file format differences by using a library such as BFD, or concentrate on one file format?
- Does a linker maintain an abstract internal representation, or does it just maintain references to contents of the original ELF files?
- Does a linker memory map the input files, this can improve performance but burn a lot of valuable address space on 32-bit systems?
- What type of control script language does a linker support?
- Does the linker fabricate an internal control script if none is provided?

There will be some commonality between linker implementations

- Objects, libraries, symbols, sections and relocations will be common data structures in all implementations

Finding all the content for a link



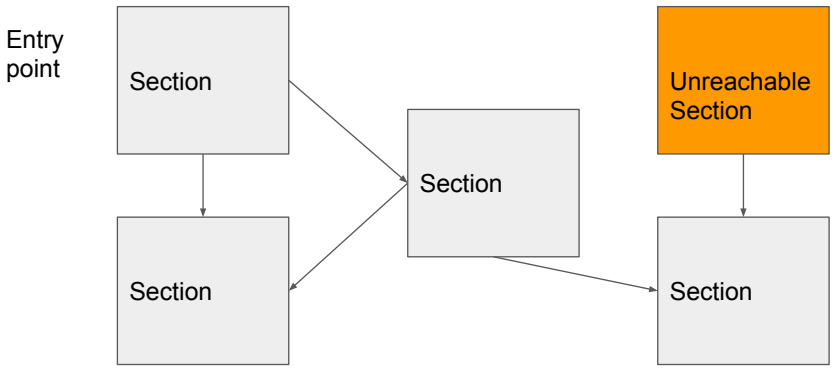
Loading an object from a static library may introduce new undefined references as well as definitions, this may require more objects to be loaded to satisfy these references.

Shared libraries may also be “loaded” to provide symbol definitions and references, although loading just means that the shared library is added as a dependency of the output. The contents of the shared library beyond the symbol table aren’t needed.

Finding all the content for a link is an iterative process that finishes when no more objects are loaded to resolve an undefined reference.

On some linkers a library will only be scanned once in command line order. On such a linker, if an object is loaded on the command library in a position after the library that that refers to a symbol defined in the earlier library it will not be scanned. For example for the command line: “file1.o lib1.a file2.o”, if file2.o needs an object defined in lib1.a then lib1.a won’t be scanned. Typical work arounds for this problem are to add the library to the command line more than once “file1.o lib1.a file2.o lib1.a” or use the -start-group and -end-group command line options.

Internal state after loading content

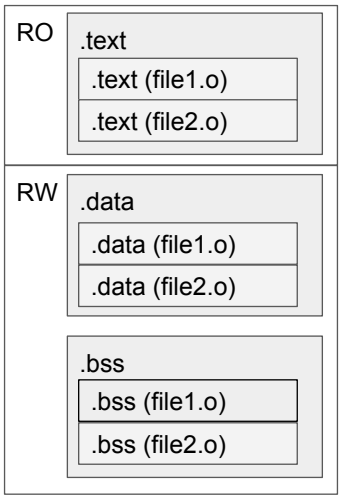


- A directed graph with sections as vertices and relocations as addresses.
- No layout or address information available.
- Can do transformations such as garbage collection.

When all content has been loaded from the input objects and libraries the linker has a directed graph of sections connected by relocations. This is essentially equivalent to one giant relocatable object file. As all content has been loaded we can now proceed with transformations that require global information about the program, but do not require addresses. A classic example is garbage collection where the linker removes unconnected components that are not reachable from an entry point.

Layout and address allocation

```
                    0x1000
SECTIONS
{
    . = 0x1000;
    .text : { *(.text*) }
    .data : { *(.data*) }
    .bss  : { *(.bss*) }
}
```



- Sections from objects InputSections are assigned to OutputSections
- Can be controlled by script or by defaults
- OutputSections assigned an address
- InputSections assigned offsets within OutputSections
- Similar OutputSections are described by a Program Segment
- Adding, removing or changing section size affects addresses.

Up to now we have been dealing with properties of sections that are independent of the address of the section. For example we don't need to know the functions address to know that it is used or not. To complete the linking process we need to assign addresses to every section (and hence symbol defined within it) so that we can resolve the fix ups and write the output to the file.

In most linkers the layout of sections is controlled by a linker script, this can either be provided by the user, or a default script. An alternative is to hard code the layout process.

Definitions:

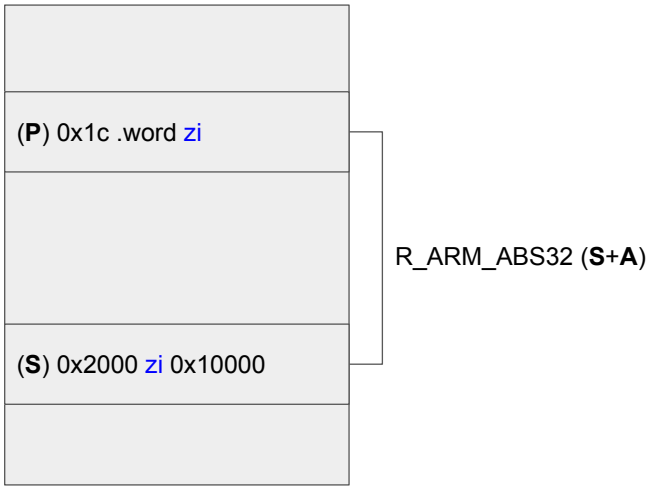
- The "." is a special location counter that refers to the current address. It can be assigned to update the address.
- InputSection: a section read from an input object.
- OutputSection: a section that will appear in the output file.
- InputSectionDescription: the filter that matches InputSections to OutputSections

InputSections are assigned to OutputSections either by explicitly matching an InputSectionDescription like *(.text*) or via some default rule

The linker assigns sections ascending addresses depending on the value given by the location counter and the alignment requirements of the section.

Segments and Program Headers are often auto-generated based on the OutputSections

Relocation



Once final addresses of all sections are known then relocations are resolved. In general for a relocation at address **P**

- Extract addend **A** from relocation record (RELA) or from location (REL)
- Find destination symbol address **S**
- Perform calculation
  - **S** + **A** for absolute
  - **S** + **A - P** for relative
- Write result to **P**

Once every section has its final address, every relocation and symbol defined at some offset to the start of the section will have an address. We can now resolve the relocations.

Relocations are defined by the ABI for the target architecture. In theory they can be any expression that the linker can calculate based on the relocation code, place P, symbol value S and the addend A. In practice the majority of relocations are of the form S + A for absolute relocations and S + A - P for pc-relative relocations, with the relocation code identifies the type of instruction that is being relocated. This means that a linker does not need to disassemble the place to work out what it has to do.

In general RISC targets tend to have more complicated relocations as the instruction set size is fixed, this means that there needs to be a relocation code for each different addressing mode. On a variable length encoding there is not usually a restriction on immediate size.

How fast is LLD

- Comparing link time against ld.gold and ld.bfd
- Factors influencing link time in large programs
- Time spent in each phase

Warning on numbers

- Aim is to show relative proportions only, numbers are aggressively rounded.
- Tested on a single machine only. Your mileage may vary.
  - Intel(R) Xeon(R) CPU E5-1660 v4 @ 3.20GHz with 32GB of memory.
- Have run some smaller tests on a Thunder-X1
  - AArch64, high core-count but low single-threaded performance relative to Xeon
  - When program works well, multithreading can make a larger difference.

How fast is LLD compared to ld.bfd and gold?

Program\Linker Time in seconds	ld.bfd	ld.gold 1 thread	ld.gold 4 threads	ld.lld 1 thread	ld.lld threads
Clang	4	1.56	1.55	0.62	0.5
Clang debug -O2 icf=all	32 (no icf)	27	21	14.5	7.5
Clang debug -O1 icf=all	32 (no icf)	22	15	11	3
Libchrome.so -O2	53	25	22	10	8
Libchrome.so -O1	61	21	19	7	5

In the above chart the -O level can affect the amount of merging the linker will do. On LLD -O0 does no merging, -O1 merges identical items, -O2 does tail sharing of strings. On LLD more of the -O1 operation can be done in parallel.

ld.bfd does not support the identical code folding (icf) optimisation. It didn't take a significant amount of time in LLD or Gold.

In general lld is faster than gold by 2 to 3 times and can be from 5 to 10 times faster than ld.bfd.
As we will see the structure of the object files will see some parts of the linker working harder in different programs.

Where is the time spent during linking?

- Can depend on the input program and optimization choice.
  - -ffunction-sections significantly increases number of symbols and sections.
  - C++ programs and exceptions add extra complexity.
  - Debug information adds a lot of data and strings.
- Example programs
  - Clang X86_64, release with debug, icf=all
  - libchrome.so aarch64, -ffunction-sections

The compiler option -ffunction-sections compiles each function in its own section. This gives the linker more opportunity to optimize, but increases the number of ELF components.

C++ programs add complexity to the linking process:

- Templates and inline functions need to be deduplicated. ELF uses COMDAT groups for this purpose, a linker must keep only one instance of a COMDAT group with the same signature.
- To optimize Exception tables for size, linkers have to understand the section contents to process them.

Debug information is much larger than program information and can stress string deduplication algorithms.

ELF components in a clang build

- Clang is a relatively large C++ program that is statically linked by default.

Elf Component	Approximate number
Object Files	2000
Sections	100,000
Global Symbols	225,000
Local Symbols	225,000
Relocations	1,500,000 non debug 0 Otherwise
Size of sections in **Megabytes**	80 SHF_ALLOC 10 Otherwise

ELF components in a clang build with debug

- Debug information dwarfs non debug information, changes in red.

Elf Component	Approximate number
Object Files	2000
Sections	110,000 (+100,000)
Global Symbols	225,000
Local Symbols	235,000 (+100,000)
Relocations	1,500,000 non-debug 100,000,000 debug (+100,000,000)
Size of sections in **Megabytes**	80 non-debug 1,600 debug (+1.6Gb)

Adding in debug information to the release build shows up how much extra data and metadata debug information adds. This can slow link time considerably.

ELF components in a libchrome.so link

- Libchrome.so has been compiled with -ffunction-sections

Elf Component	Approximate number
Object Files	21,000
Sections	1,000,000
Global Symbols	2,000,000
Local Symbols	3,000,000
Relocations	2,900,000 SHF_ALLOC 20,000,000 Otherwise
Size of sections in **Megabytes**	100 SHF_ALLOC 800 Otherwise

Despite being similar in size, libchrome.so has more smaller files containing larger number of components.

Where does link time go? Clang debug

Phase	-O3 First Link	-O3 subsequent	-O1 (no tail merge)
Input file reading	15%	3%	8%
Split sections	33%	4%	12%
ICF and GC	1%	1.5%	3%
Merge sections	4%	**75**%	25%
Code Layout	1%	2%	5%
Write Output File	**46**%	15%	**44**%
Overall Time	**163** seconds	**7** seconds	**2** seconds

On the first link the file cache is cold and disk access dominates. The link time is considerably longer as well.
When high optimization is used the tail merge of all the debug strings starts to dominate link time.
At a lower level of optimization lld can parallelise the string merging process more resulting in less merge overhead.

Where does link time go? Chrome for AArch64

Phase	-O3 First Link	-O3 subsequent	-O1 (no tail merge)
Input file reading	**30%**	13%	20%
Split sections	9%	2%	3%
ICF and GC	7%	17%	**25%**
Merge sections	4%	**35%**	3%
Code Layout	1%	16%	21%
Write Output File	6%	12%	17%
Overall Time	**98**s	**9**s	**6**s

The timer information in the -O3 First Link highlight that I've missed timing a subsection that took a substantial amount of link-time.
Code Layout for AArch64 takes longer than on X86_64 due to needing to calculate thunks and perform the Cortex-A53 Errata fix.

LLD Design and implementation choices.

- Characteristics of linkers
- Abstractions
- Memory Management
- Multithreading

Characteristics of Linkers

- All ELF linkers have to splice large amounts of data from many input files into one large file.
- “Smart format, dumb linker.” A linker does not need to understand the contents of Sections
  - Linker largely manipulates the ELF metadata.
  - Relocations give linker instructions on how to manipulate section contents.
- Linker optimizations need to work on a large amount of data.
- Internal representation of program does not change throughout link.
- Most data structures persist for lifetime of the program.
- More opportunity for parallelism within stages of the pipeline than running multiple pipelines in parallel.

Linker’s have a number of characteristics that a design can exploit to obtain high performance.

- We will be allocating a lot of memory, but not freeing it.
- We will be iterating over many data structures of the same type, cache locality will matter.
- Algorithms will need to scale to large inputs.
- There are ordering dependencies and synchronization points that can make what looks like easy parallelism more difficult.

Abstractions

- LLD has a lightweight abstraction representing ELF components.
- Structured as a program first, library second
  - In contrast to clang and llvm.
- InputSection, OutputSection, Symbol, Relocation.
  - Content of InputSections only accessible read-only via mmap.
- Merge sections represented by pieces with an input-output offset map.
- Linker created content represented as SyntheticSections.
- No built-in linker script
  - Simpler, faster code-path for most common case.
- One pass through relocations to create PLT, GOT Synthetic Sections.
- Limited amount of Target Specific code.

LLD initially started out with a shared codebase for COFF, ELF and MachO. Due to the differences between the formats, extra dependencies betweens sections and atoms (roughly equivalent to functions) had to be added. This made the codebase difficult to work with. The COFF and ELF codebases were created to share a design and algorithms but each would be customized for the file format and only include features needed by the linker it was emulating.

Most LLVM programs are built as reusable libraries first, with the command line programs being wrappers around them. This permits reuse of components throughout many programs, for example the parsing code in clang can be embedded in refactoring and IDE tools. LLD is structured as a program that can be wrapped in a library, this permits a simpler more direct approach to get at components at the expense of presenting a very coarse and inflexible interface when used as a library. In short LLD is not built upon a linker construction kit that can be used to build linker like tools. The general feeling is that no-one has yet to come up with a good enough use case to do the work to implement a linker construction kit.

LLD only reads the information it needs from the input files. Copying section data to the output very late. Alternatively it could copy the full input contents into memory at read time. This would permit arbitrary modification of the section data, but would harm performance.

Linker's often need to create extra bits of code and data that isn't present in the object file. LLD chooses to implement these with classes called synthetic sections. These

can be inserted into OutputSections are assigned addresses and written out as if they came from an InputSection. Some linkers create artificial output sections, these work well for sections like the PLT and GOT, but don't work so well for Thunks that need to be inserted in between InputSections from Objects.

The control flow of ld.bfd is entirely driven by linker scripts. When one isn't given a default script is used. This is elegant, and always means that all functionality will work with a linker script, but it can be more expensive to process the script than a custom hard-coded default path.

LLD tries to keep the amount of Target specific code to a minimum, this has the advantage of making it easy to port a new Target if it happens to look similar to other targets. It can make it more difficult if the target does something radically different though.

Data Structures and Algorithms

- LLD makes heavy use of existing of optimized LLVM components
  - ELF Reading and Writing.
  - Memory buffers.
  - smallvector.
  - Threading support.
  - stringref, arrayref and functionref.
  - LTO library.
  - Hashing algorithms.
  - Memory Allocator.

The LLVM project has a rich ADT library that contains optimized containers and libraries that have accelerated LLD's development and give it a performance advantage over default standard library types.

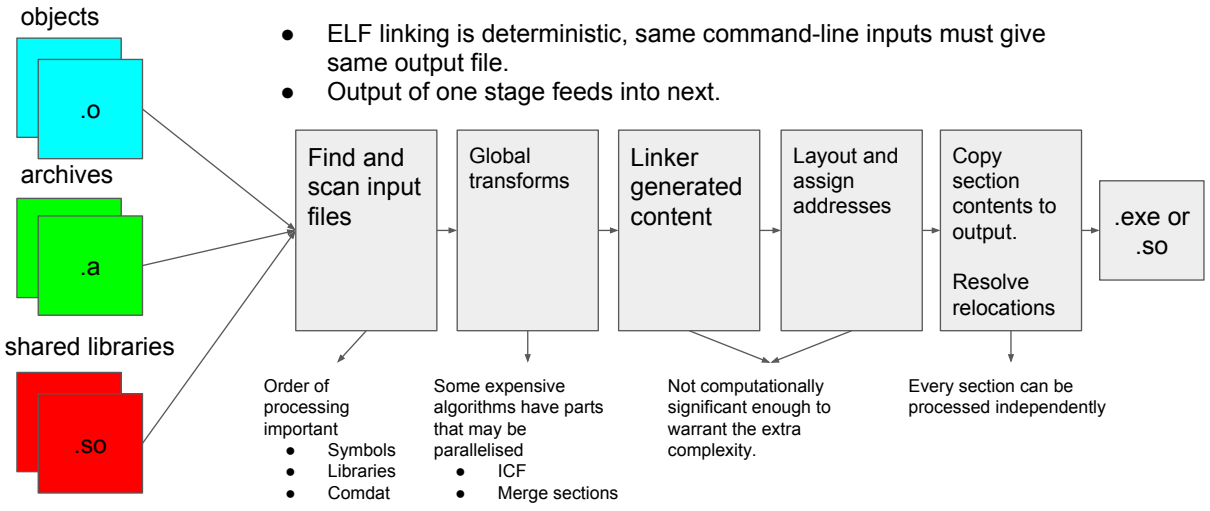
LLD can also take advantage of the LTO library in order to handle LTO without needing a plugin interface.

Memory Management

- Linker has an exploitable pattern of memory access
  - Linker will read Section, Symbol and Relocation metadata at start of link.
  - All optimizations operate on these data structures.
  - Layout is aggregation of these data structures.
  - Section content is memory mapped.
  - Writing the output file uses these data structures.
- Large amounts of memory allocated up front and never released.
- Ideal for a region based memory allocator such as llvm::bumpPtrAllocator.
- LLD uses a region for each major type
  - Has benefits for memory locality when iterating through the same type.

As has been seen previously a linker will need to create hundreds of thousands of sections and symbols and in some cases tens of millions of relocations. This will end up with a lot of allocations of the same type, that can't be released until the output is written (the last thing a linker does). A region based allocator where most allocations are just incrementing a pointer within a larger block can improve performance considerably. LLD has a separate memory region per type so iterating through a type doesn't jump all over the address space.

Opportunities for parallelism when linking



A key user expectation of an ELF linker is that the same inputs will generate the same output. Due to the way that the image layout and even program contents can dependent on the order of processing. This prevents us from just loading each object in parallel. Gold does do some loading in parallel, with symbols loaded sequentially but other metadata loaded in parallel.

Another limiter to parallelism is that some stages require the previous stage to finish. For example garbage collection needs all content loaded before it can determine if a section is not connected to any other.

Ahmdal's law also applies, there isn't much point in adding complexity to stages that don't take up much link time.

The most natural place for parallelism is once all the layout has been done; the process of copying and relocating section contents is independent of any other so there is a big opportunity to use multiple threads.

LLD's threading model

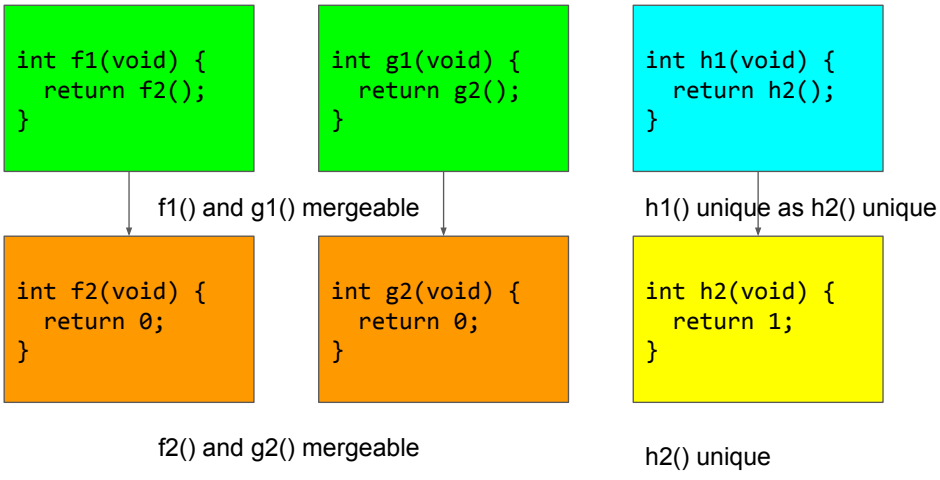
- Keep it as simple as possible!
  - No task model.
  - No shared state.
  - No explicit locking.
  - No memory allocation in parallel sections.
- Use llvm::parallelForEach() to call a function on a slice of data.
  - Identical Code Folding.
  - Merge Sections.
  - GDB Index.
  - GNU build-id.
  - Writing and relocating the output file.

In contrast to gold, which has a task based model that has worker threads take tasks of a queue, LLD restricts itself to LLVM's parallelForEach construct. This means sharding data into independent sets and calling a function to operate on that set. As the operations are indendent there is no locking needed. A restriction of the model is that no memory allocation can be done within parallelForEach as the allocator is not thread safe.

The focus is to look at key algorithms that may require multiple passes or involve a lot of computation; extract out the steps that can be run within parallelForEach leaving the complex parts to be done sequentially.

Although gold's model is more general, LLD frequently obtains higher CPU utilization in practice. This is particularly observable when writing large amounts of debug data.

Parallel ICF



The ICF algorithm looks at sections that are identical, yet have different symbols. It forms equivalence classes, where each equivalence class contains identical functions. At the end of the ICF algorithm a single candidate is chosen and the rest are eliminated. ICF needs to take account of the connections between sections when deciding whether they are equivalent, this can be quite an expensive process when there are many sections. LLD has been able to parallelise parts of the ICF algorithm.

Parallel ICF

1. Partition sections into equivalence classes ignoring relocations.
2. For each section in equivalence class compare relocation targets
   a. If targets are in different equivalence class then split equivalence class.
3. Repeat step 2 until no more equivalence classes split.
4. Pick 1 section from each equivalence class.

- Use of llvm::parallelForEach
  - Calculation of initial equivalence class.
  - Make current equivalence class immutable, store to next equivalence class permits parallel evaluation of step 2.
  - Each shard processes a number of equivalence classes.

LLD uses optimistic ICF. The first pass optimistically assigns sections that might be equivalent to the same equivalence class. The algorithm checks each member and splits out the sections that turn out not to be valid members. Gold uses a different algorithm that starts with each section in its own equivalence class, and merges equivalence classes that are the same.

The key part of the implementation is for each section to have two ICF equivalence class identities, current and next. In an iteration the current identity is read-only, any change is written to next. This permits an equivalence class to be checked independently.

Non Technical Factors and Conclusion

Non technical factors

- Many contributors to LLD have large programs for which fast link time is important.
- Performance monitoring bot.
- Code reviewers looking out for performance problems.
- Focus on the common case, don’t add support for an edge case until there is a proven need.
- Resist feature creep.

Making a high performance linker is one thing, maintaining that performance over time requires maintainers to monitor performance and pay attention to potential problems at review time. A somewhat controversial choice is to aggressively resist feature creep. One man’s unnecessary feature can be a critical feature for a niche project.

Conclusion

- LLD's fast link time results from a number of technical factors
  - Thin abstraction layer between program and object format.
  - Use of custom memory allocator.
  - Use of optimized data structures from llvm::ADT.
  - Use of multithreading to accelerate critical algorithms.
- Advantage over ld.bfd largely due to BFD design
  - BFD supports more file formats and can convert between them.
  - Design limitations documented in design of gold linker.
- LLD has many similarities to Gold
  - Simpler and more effective threading model.
  - More efficient data structures and implementations of key algorithms.
- User base and upstream that care about and monitor performance.

The GNU linker ld.bfd's design limitations are well known, and are well documented in the Gold design paper (see references). Both gold and LLD share the advantages of being able to make a modern design focusing on ELF only.

LLD's performance advantage over Gold is harder to explain. There is no one thing that can easily explain the difference. It is likely a combination of:

- Better use of multi-threading, including key algorithms such as icf and string merging.
- Region based memory allocation.
- Access to optimized ADT types.

References

- Gold a new Linker
  - Design of Gold and a good reference to the limitations of BFD.
  - https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/34417.pdf
- Concurrent Linking with the GNU Gold Linker
  - Ostensibly a project to implement concurrent linking (linker as a background process) but also has a good description of Gold's tasks and dependencies between tasks.
  - https://smvv.io/gold.pdf
- LLD Performance testing bot
  - http://lab.llvm.org:8011/builders/lld-perf-testsuite/builds/11276
- Linkers and Loaders John R Levine.
  - https://linker.iecc.com