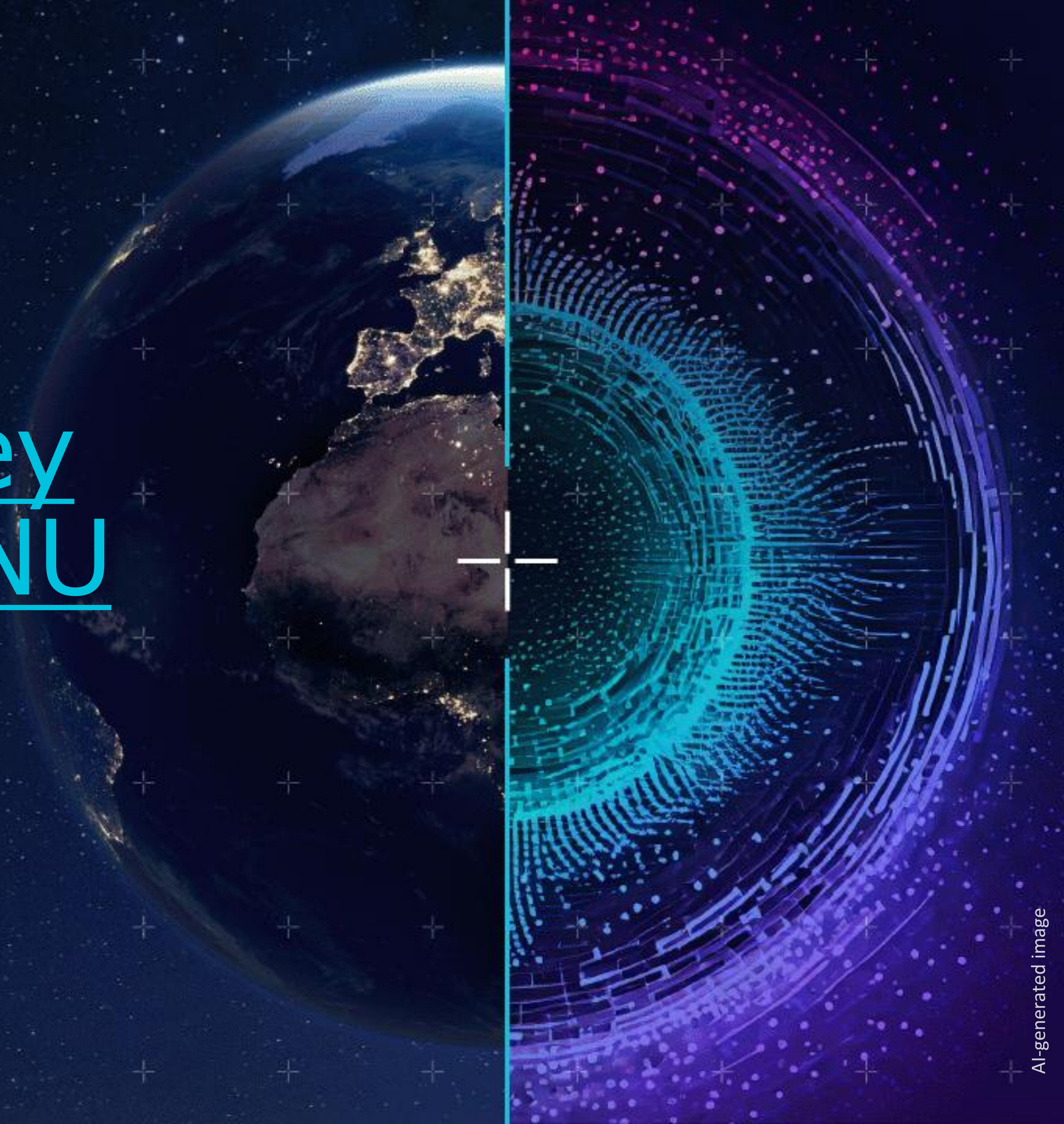
Linker Scripts in LLD and how they compare with GNU ld

Peter Smith
31/01/2024

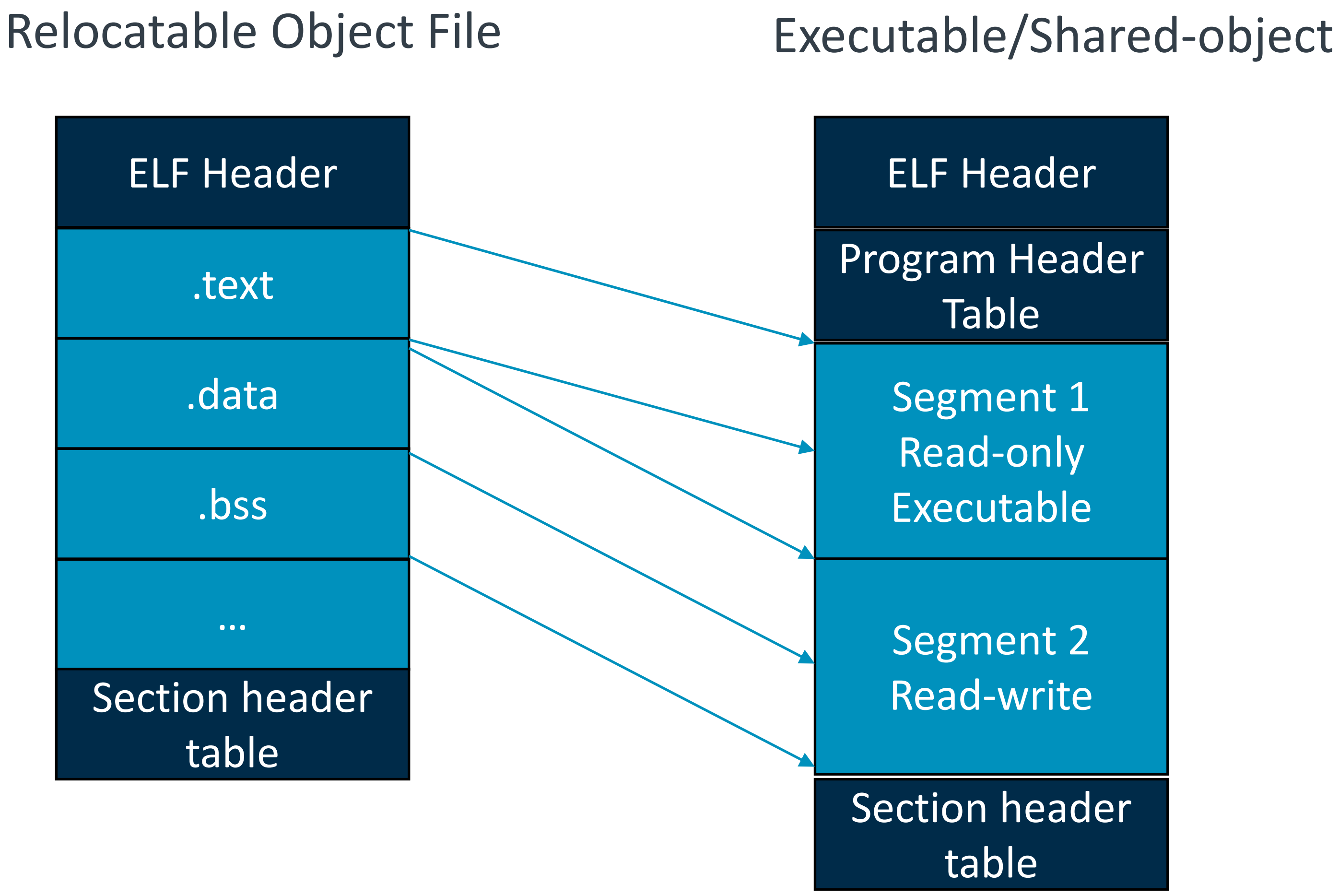


AI-generated image

Linker script essentials

What do you need to know to get anything out of this talk?

ELF components



- Relocatable objects and executables/shared-objects use same file format.
- Sections in relocatable objects such as .text are consolidated into larger sections in the output file.
- Segments contain one or more sections.
- A segment is described by a program header.
- Program loaders operate on segments.
- Section level view present for debugging.

Linker control scripts

- A text file written in the linker command language
- GNU linker ld.bfd always uses a linker script even if none provided.
- LLD and ld.gold have a separate code-path for when there is no linker script.
- Command line option `-T/--script` or as an input file
  - When `-T/--script` used this replaces the default linker script.
  - When a linker script is an input file it is combined with all other linker scripts.
- Controls how sections from input files (input sections) map to the sections in the output file (output sections).
  - `.text : { *(.text .text.*) }`
- Control the layout of output sections in memory and the section to segment mapping.

Linker Script Illustrative example

```
MEMORY
{
  FLASH (rx) : ORIGIN = 0x0, LENGTH = 0x20000 /* 128K */
  RAM (rwx) : ORIGIN = 0x10000000, LENGTH = 0x2000 /* 8K */
}

SECTIONS
{
  .text : {*(.text*) } >FLASH
  __exidx_start = .;
  .ARM.exidx : { *(.ARM.exidx*) } >FLASH
  __exidx_end = .;
  __etext = ALIGN (4);
  .data : { *(.data) } >RAM AT>FLASH
  .bss : { *(.bss) } >RAM
}
```

- Define memory sizes and properties.
- Define output sections
- `.` is DOT, the location counter
- `ALIGN` is a built-in function
- > assigns output section to memory region that it will execute in (VMA)
- >AT assigns output section to memory region that it will load in (LMA)

GNU ld and LLD linker script handling

- The GNU linker manual is the closest there is to a specification for linker scripts
  - https://sourceware.org/binutils/docs/ld/Scripts.html
- Some parts are underspecified, some are implementation defined
  - Placement of orphan sections.
  - Section to segment mapping.
  - Alignment in memory regions.
- GNU ld and LLD are moving targets
  - Not all features are implemented in LLD.
- Sometimes LLD has made a design decision to differ from GNU ld
  - https://lld.llvm.org/ELF/linker_script.html#linker-script-implementation-notes-and-policy

Orphan Placement

Input sections that are not specified by the script

Orphan sections

- A linker script does not have to give a complete mapping from input section to output section.
- Input sections that do not match any input section description are called “orphan sections”.
- Linker is expected to automatically find a place for orphan sections
- `--orphan-handling=[place (default), discard, warn, error]` can be used to alter policy.
  - `--orphan-handling=warn` will tell you where orphans have been placed.
- `--unique` prevents orphan sections with same name from being consolidated.

Orphans and linker scripts

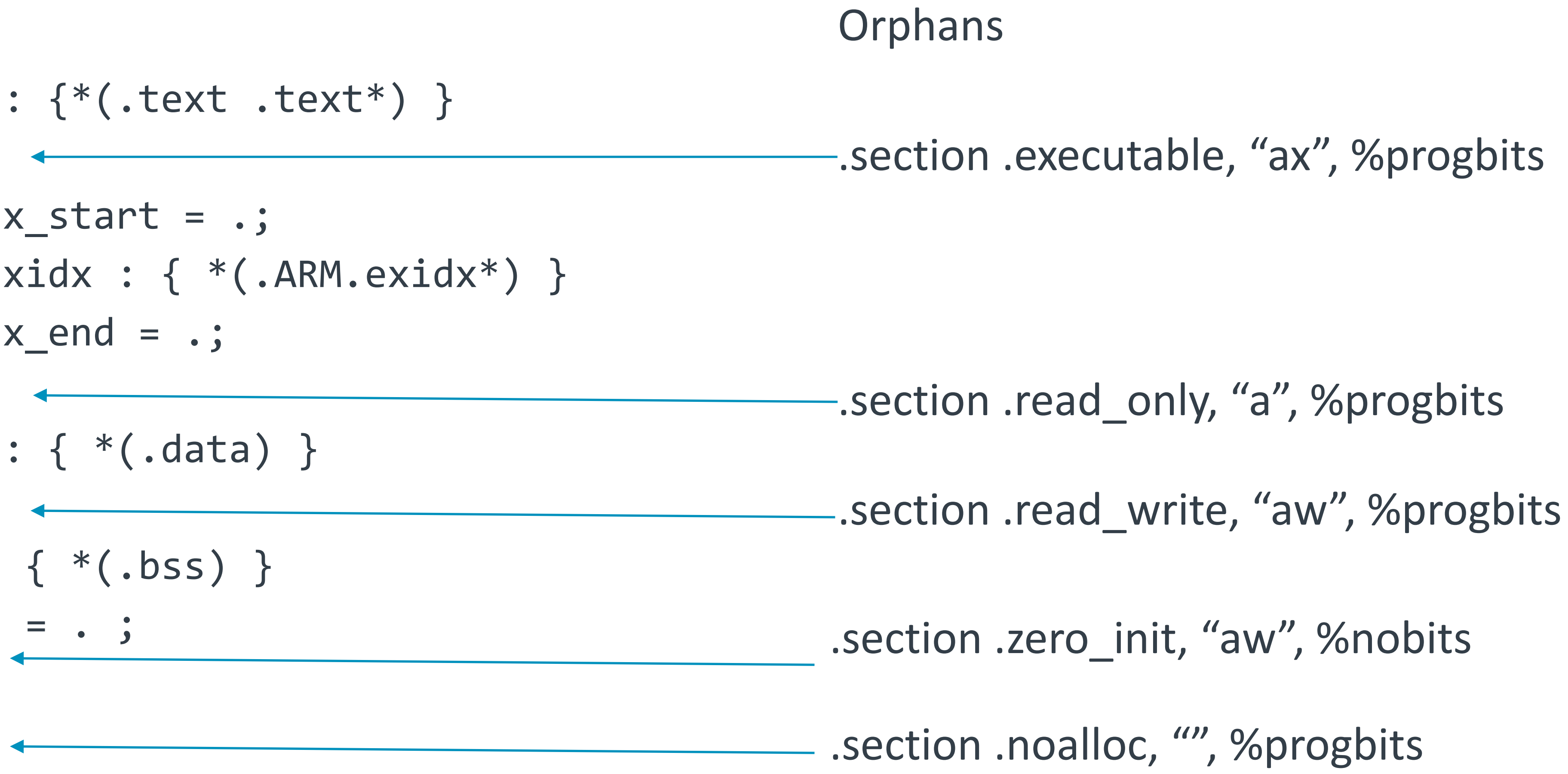
```
SECTIONS
{
  .text : {*(.text .text*) }

  __exidx_start = .;
  .ARM.exidx : { *(.ARM.exidx*) }
  __exidx_end = .;

  .data : { *(.data) }

  .bss : { *(.bss) }
   __end = . ;


}
```



Orphans

.section .executable, “ax”, %progbits

.section .read_only, “a”, %progbits

.section .read_write, “aw”, %progbits

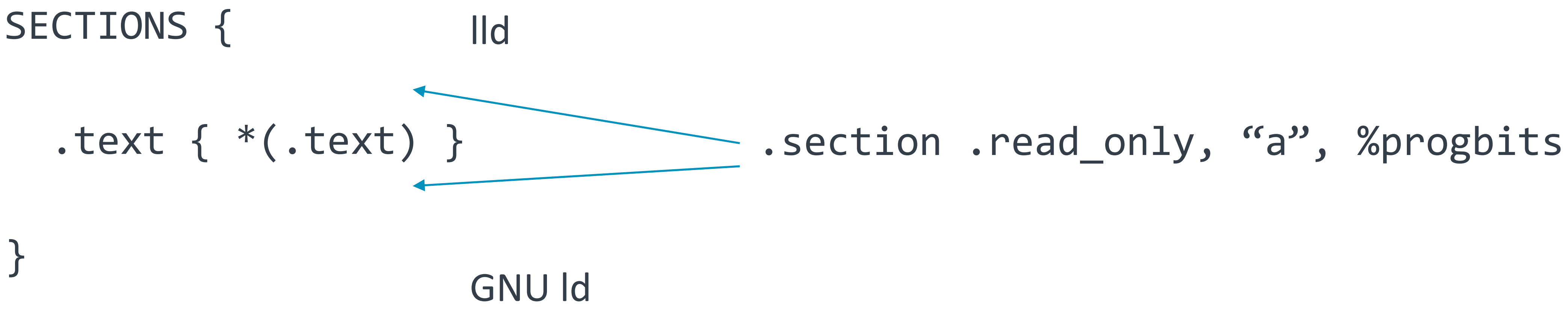
.section .zero_init, “aw”, %nobits

.section .noalloc, “”, %progbits

LLD and GNU ld orphan placement

- Both use similar examples but there are differences in detail
- Similarities
  - Orphans matching an output section name are assigned to that output section.
    - `.foo : { *(.bar) } /* Matches orphans with name .foo */`
  - New output section created for orphans that don’t match by name.
- Output sections and orphans ranked by property flags
  - Read-only, executable …
- Orphan placed at the after the last output section with the closest rank.
- Have to avoid breaking symbol assignments
  - `start = .; foo : { *(foo) } end = .;`
  - `.foo : { *(.bar); . += 0x1000 ; } /* .foo placed after . expression */`
- Orphans placed after the last output section placed after all trailing commands.

Example difference of orphan placement



```
SECTIONS {

  .text { *(.text) }

}
```

lld

```
.section .read_only, “a”, %progbits
```

GNU ld

- Without a read-only output section in the Linker Script LLD ranks before .text and GNU ld after.
- Can be solved by adding at least one output section that contains only read-only data.

Unallocated sections influence on orphan placement

```
SECTIONS {
  .text : { *(.text) }
  foo : { *(.foo) }
  bar : { *(.bar) }
  baz : { *(.baz) }
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

```
.section .foo, "aw", %progbits
…
.section .bar, “w", %progbits
…
.section .baz, "aw", %progbits
…
```

- None of the sections are orphans
- The SHF_ALLOC flag “a” is missing from .bar. This is a common oversight.
- LLD will insert linker generated sections like .comment after output section bar.
- GNU ld will place linker generated sections like .comment at the end.

Program Header generation

Section to segment mapping

Elf Segments and Alignment

- Segments are described by ELF program headers of type PT_LOAD.

Program Header field	Description
p_type	Type of program header, PT_LOAD in our case.
p_offset	Offset in file of program segment.
p_paddr	Physical address of segment (ignored for System V)
p_vaddr	Virtual address of segment
p_memsz	Size in memory of program segment
p_filesz	Size in file of program segment
p_align	**p_vaddr congruent to p_offset (modulo p_align)**

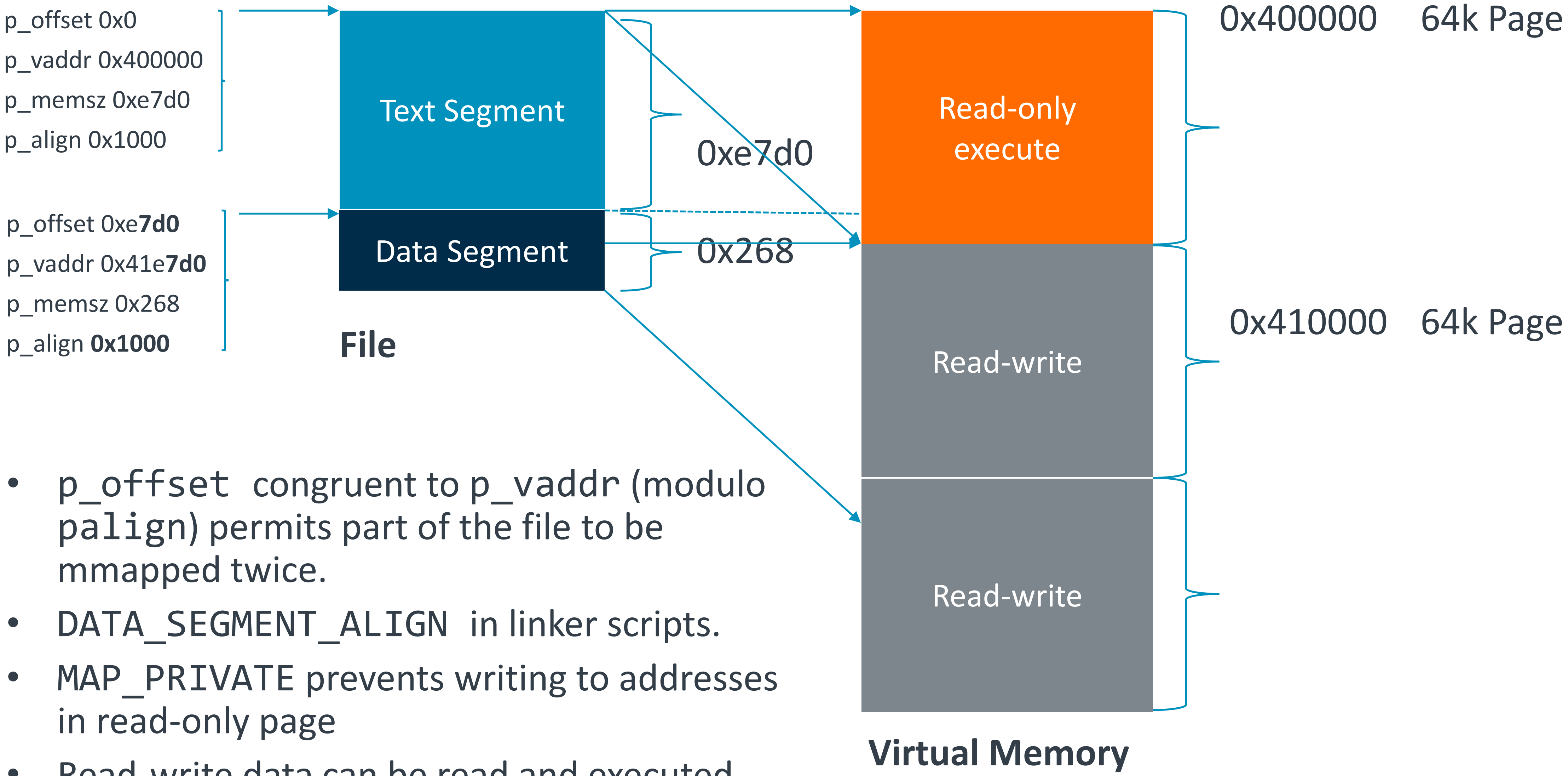
Program Header assignment

- A PT_LOAD program segment is described by an ELF program header
  - Contiguous range of bytes in the file with the same properties
- In a System V Operating-System the ELF file will be memory mapped
  - Program segments need to be appropriately aligned.
  - Content is contiguous in the file and in memory.
  - No difference in virtual and physical address.
  - Zero-initialized data must follow non-zero initialized data within segment.
- In an embedded system the ELF file may not be executed directly
  - Program segment contents extracted by a tool like objcopy.
  - System may not have virtual memory.
  - Virtual and physical address may differ (RW data copied to RAM at startup).
  - File contents are contiguous, but memory contents may not be.

Influences on program header assignment

- VMA to LMA offset of an Output Section
  - A single program header can represent many contiguous output sections with the same offset.
  - For memory mapped ELF files this is always 0
  - Can be altered for an output section using AT(offset) or `AT> memory_region`.
- Changes in properties such as RO to RW
  - Configurable by flags as properties can be merged.
- Special cases like `-zrelro` and `-zseparate-code`
- Gaps between compatible output sections
  - Extend a single program segment to cover both output sections with padding in between.

Simplified layout of an ELF file for a System V AArch64 OS



p_offset 0x0
p_vaddr 0x400000
p_memsz 0xe7d0
p_align 0x1000

p_offset 0xe**7d0**
p_vaddr 0x41e**7d0**
p_memsz 0x268
p_align **0x1000**

Text Segment

Data Segment

0xe7d0

0x268

File

Read-only execute

Read-write

Read-write

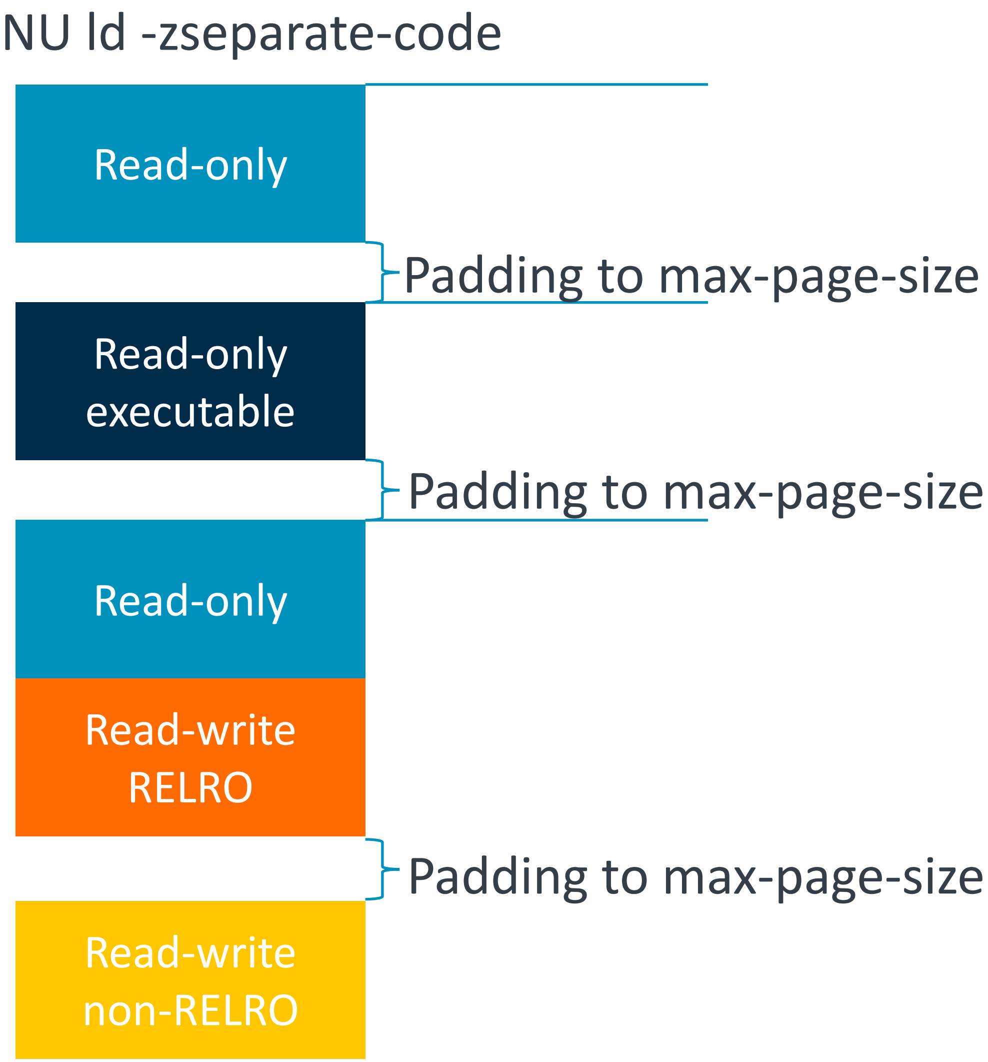
0x400000 64k Page

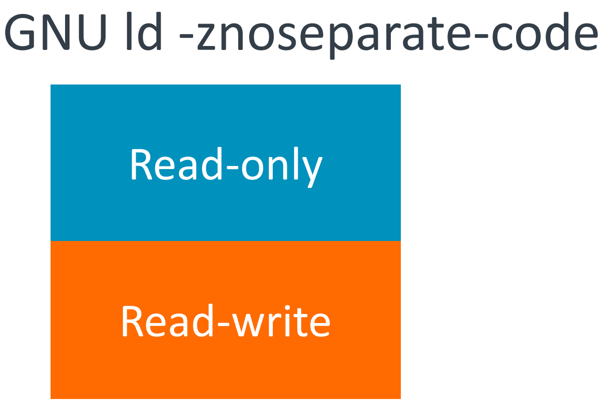
0x410000 64k Page

Virtual Memory

- `p_offset` congruent to `p_vaddr` (modulo `palign`) permits part of the file to be mmapped twice.
- `DATA_SEGMENT_ALIGN` in linker scripts.
- `MAP_PRIVATE` prevents writing to addresses in read-only page
- Read-write data can be read and executed.

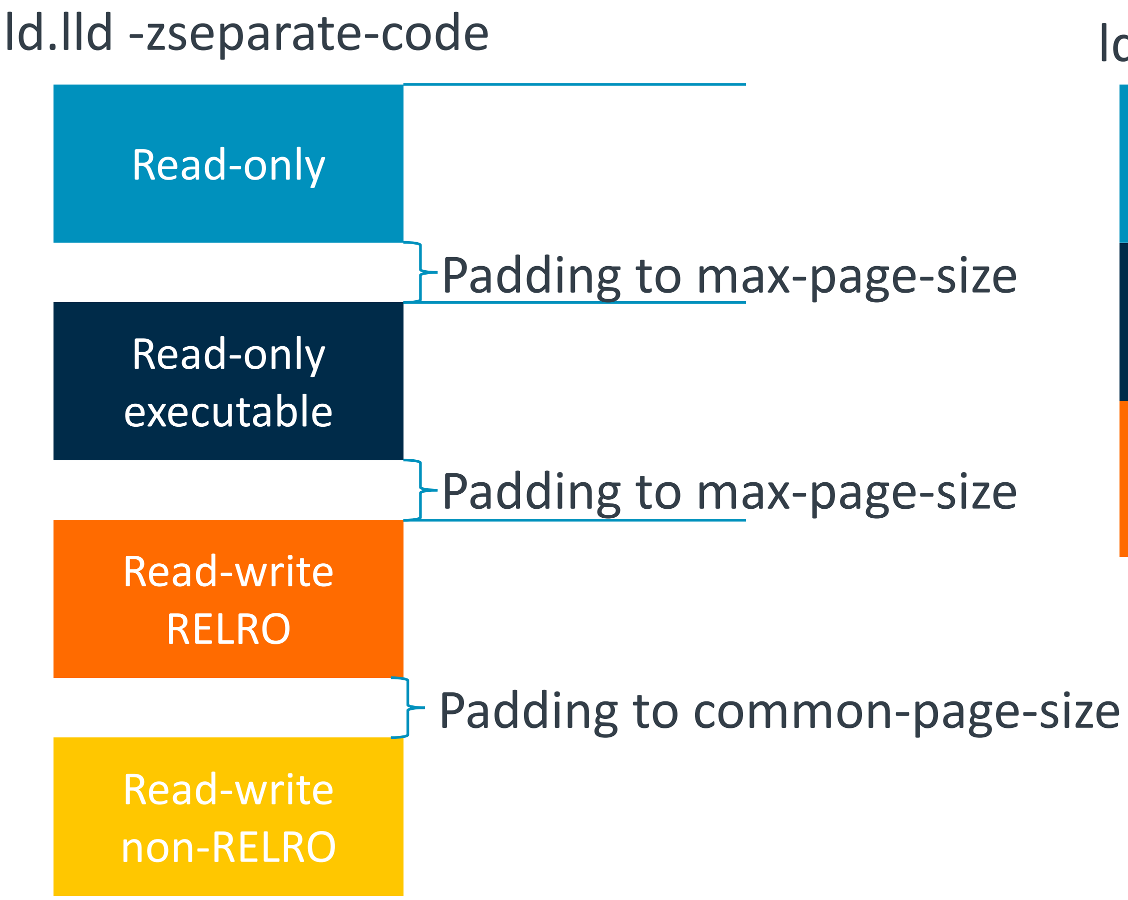
-zseparate-code in GNU ld

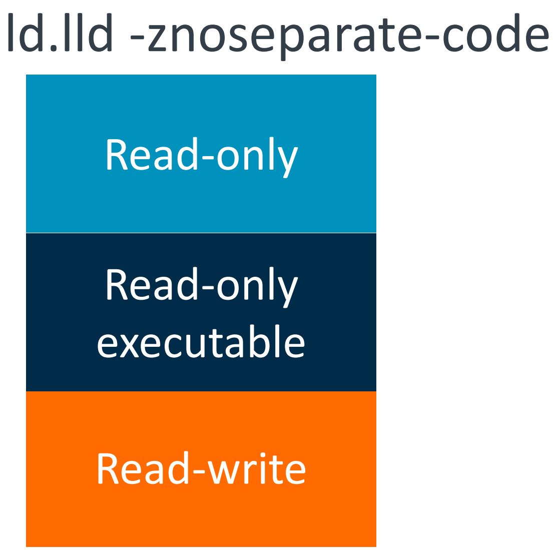




- `-zseparate-code` isolates read-only executable segment by padding to a max-page-size boundary.
- Executable code cannot execute data as code at expense of larger files and increased memory usage. Particularly on systems with large page sizes.
- GNU ld defaults to `-zseparate-code`, can be disabled with `-zno-separate-code`
- `DATA_SEGMENT_RELRO_END` pads to max-page-size boundary.

-zseparate-code in LLD



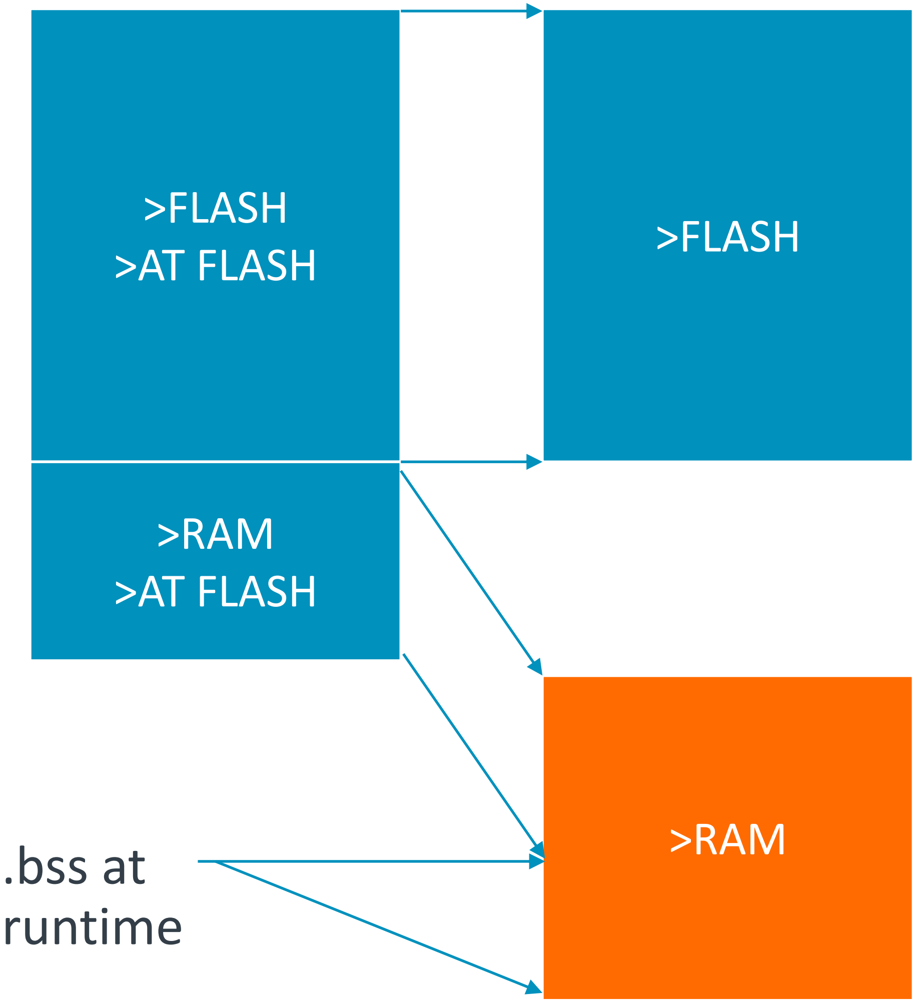


- LLD defaults to `-znoseparate-code`
- LLD doesn’t sandwich the executable segment between read-only segments
- `DATA_SEGMENT_RELRO_END` pads to a common-page-size boundary only.

Program Segments in embedded systems

```
MEMORY
{
  FLASH (rx) : ORIGIN = 0x0, LENGTH = 0x20000 /* 128K */
  RAM (rwx) : ORIGIN = 0x10000000, LENGTH = 0x2000 /* 8K */
}

SECTIONS
{
  .text : {*(.text*) } >FLASH
  __exidx_start = .;
  .ARM.exidx : { *(.ARM.exidx*) } >FLASH
  __exidx_end = .;
  __etext = ALIGN (4);
  .data : { *(.data) } >RAM AT>FLASH
  .bss : { *(.bss) } >RAM
}
```



LLD Program Header Generation Known problems

- LLD address assignment assumes that output sections VMA within a program header monotonically increase
  - Possible to break this assumption using memory regions.
  - https://discourse.llvm.org/t/overflow-related-to-program-headers/75150

```
second_section (0x10000000 +64) : { KEEP (*(.second_in_section)); } > mem
first_section 0x10000000 : { KEEP (*(.first_in_section)); } > mem
```

- GNU ld reorders output sections so that VMA and LMA monotonically increase

```
[ 1] second_section    PROGBITS       0000000010000040 001040 000001 00  AX  0   0  1
[ 2] first_section     PROGBITS       0000000010000000 001000 000001 00  AX  0   0  1
```

arm

Miscellaneous Differences

Symbol assignment differences

- Dot assignment within an output section
  - `.section : { *(.text); . = 4; *(.text.*) }`
  - In GNU ld symbol assignments in an output section are relative to the start of the output section.
  - In lld it assigns the location counter to the value, normally provoking an error message.
- This is also the case for named symbols
  - .section : { *(.text); foo = 4; *(.text.*) }
  - In GNU ld foo is a section relative symbol with value of .section + 4.
  - In lld foo is an absolute symbol defined to 4.
- For portability
  - Use `. += <value>` to move the location counter
  - Define a symbol at the current location counter `foo = .;`

References

References

- MaskRay's blog posts
  - https://maskray.me/blog/2020-11-15-explain-gnu-linker-options
  - https://maskray.me/blog/2020-12-19-lld-and-gnu-linker-incompatibilities
  - https://maskray.me/blog/2023-12-17-exploring-the-section-layout-in-linker-output
- GNU documentation
  - https://sourceware.org/binutils/docs/ld/Scripts.html
- LLD documentation
  - https://lld.llvm.org/ELF/linker_script.html
- LLVM Bugzilla (archive)
  - https://bugs.llvm.org/show_bug.cgi?id=42327 lld and GNU ld orphan handling difference
- GNU Bugzilla and patch notes
  - https://sourceware.org/bugzilla/show_bug.cgi?id=28824 relro security issues
    - Has a good description of max-page-size and common-page-size

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה
ధన్యవాదములు

Backup

LLD Program Header Generation

- Create a new program header if next Output Section
  - Program header flags are different (read-only, writeable, executable).
  - Different memory region (given by > region).
  - Different LMA memory region (given by `AT> region` or `AT(address)`).
  - Previous output section was `SHT_NOBITS` and this one is `SHT_PROGBITS`.
- LLD address assignment assumes that output sections VMA within a program header monotonically increase
  - Possible to break this assumption using memory regions.
  - https://discourse.llvm.org/t/overflow-related-to-program-headers/75150
    - `second_section (0x10000000 +64) : { KEEP (*(.second_in_section)); } > mem`
    - `first_section 0x10000000 : { KEEP (*(.first_in_section)); } > mem`
- LLD writes SHT_NOBITS contents to file as 0 if followed by SHT_PROGBITS

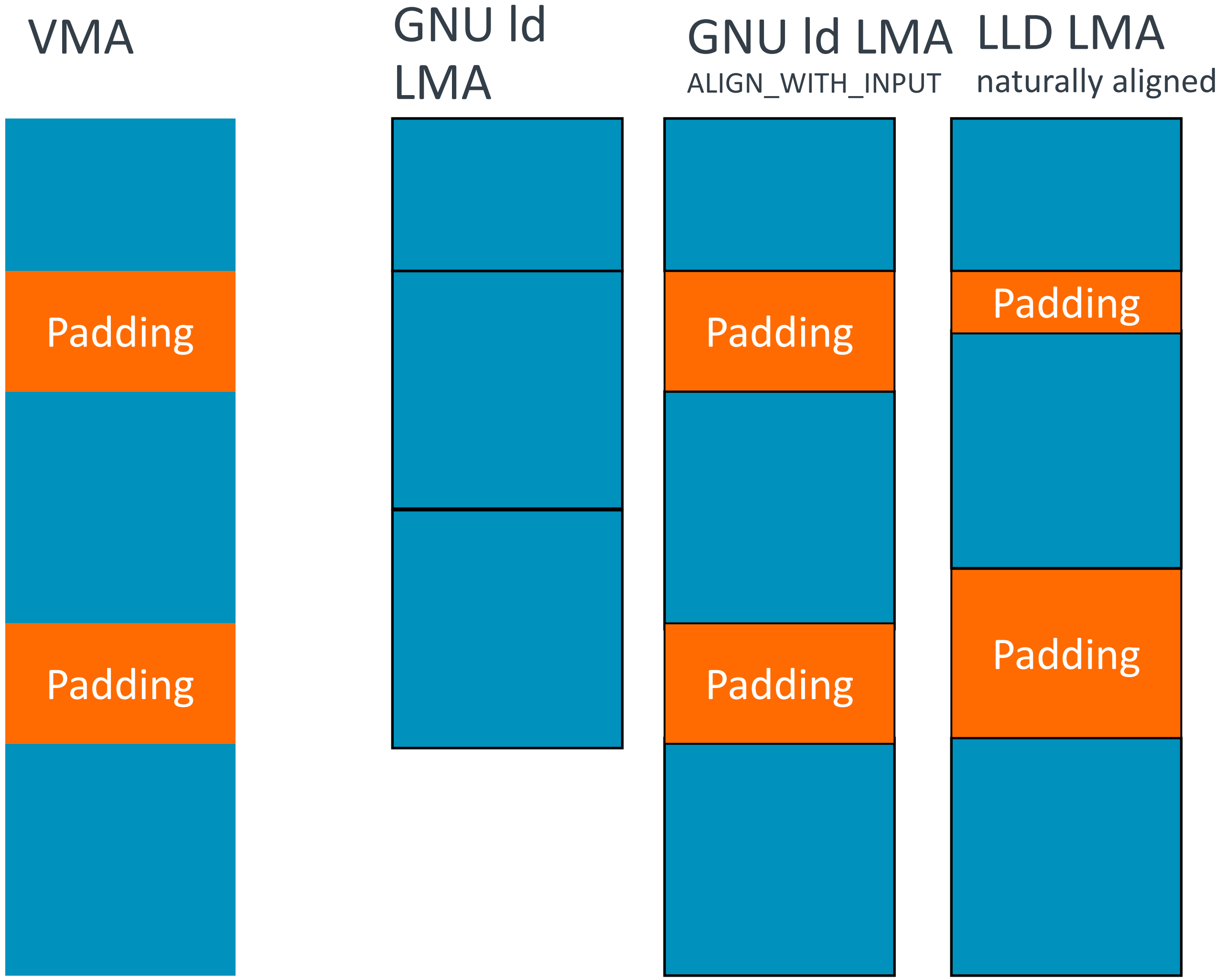
GNU ld and program header creation

- Output sections are sorted by ascending LMA, then VMA
- Create a new program header if next Output Section
  - VMA to LMA offset is different.
  - LMA overlaps with previous section LMA range [LMA, LMA + LMA size).
  - Would cause a page to be skipped within the segment.
  - If paged, section is writeable and previous section was read-only.
- GNU ld reorders output sections so that VMA and LMA monotonically increase
  - `[ 1] second_section    PROGBITS       0000000010000040 001040 000001 00  AX  0   0  1`
  - `[ 2] first_section     PROGBITS       0000000010000000 001000 000001 00  AX  0   0  1`

Alignment when VMA != LMA

```
SECTIONS {
.a : {
begin = .;
*(.a)
} > VMA_REGION AT > LMA_REGION
.b : {
*(.b)
} > VMA_REGION AT > LMA_REGION
.c : {
*(.c)
end = .;
} > VMA_REGION AT > LMA_REGION
```

- GNU ld default no LMA alignment
- GNU ld ALIGN_WITH_INPUT uses VMA alignment padding
- LLD naturally aligns in LMA



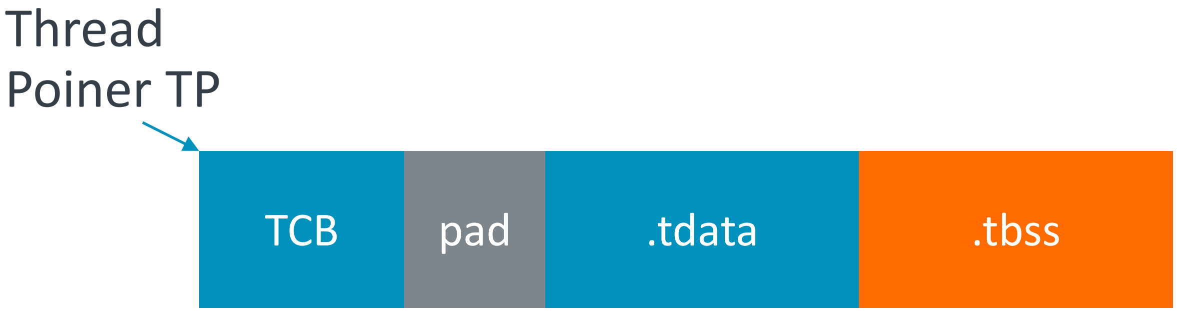
Evaluation

- GNU ld default produces smallest LMA size, but:
  - Requires an individual copy of each OutputSection to VMA.
  - Copy cannot assume alignment of source (for example a 16-byte aligned vector copy).
- GNU ld with align_with_input replicates VMA padding
  - Whole memory region can be copied in one go.
  - OutputSections not guaranteed to be naturally aligned in LMA.
- LLD naturally aligns in LMA
  - If VMA and LMA not congruent (modulo alignment) then cannot copy whole memory region in one.
  - Output sections guaranteed to be naturally aligned.
  - Possible to generate large gap
- All implementation choices reasonable
  - Won’t matter much for small alignments
  - Users sometimes (ab)use large alignments to place sections, could result in large binaries.
  - Could offer an option for ld.bfd alignment, with support for ALIGN_WITH_INPUT

Alignment of 0 size OutputSections in LMA

- https://github.com/llvm/llvm-project/issues/64571
- Source is a zero-sized OutputSection with ALIGN directive
  - .output_section : ALIGN(16) { … }
- GNU ld does not emit the 0 sized section into LMA, no additional padding
- LLD adds the padding to naturally align
- Opportunity to optimize.
- Likely similar case in https://github.com/llvm/llvm-project/issues/65159
  - 0 sized section with lower VMA added to same program header causing negative file offset.

TLS local exec alignment



- ELF file contains `.tdata and .tbss`
  - `PT_TLS` program header for dynamic linking
  - Linker defined symbols for embedded systems
- Linker and library must agree on size of alignment padding for TLS
  - Newlib/picolibc use `MAX(2*wordsize ,MAX(ALIGNOF(.tdata, ALIGNOF(.tbss))))`
  - LLD uses more complex expression that saves padding if overaligned `.tbss`
    - s.getVA(0) + config->wordsize * 2 + ((tls->p_vaddr - config->wordsize * 2) & (tls->p_align - 1));
  - Does not match libraries calculation.
- Linker defined symbol for TLS padding that library can use if defined?