Linker Relocation

Peter Smith

Introduction and purpose

- Why do object files have relocations?
- How is a relocation represented in an Object file?
- How does a linker resolve a relocation?
- How to read the ABI documentation?
- GOT and PLT generating relocations and dynamic relocations?
- Relative relocations

Why do we have relocations?

- Linking is the process of turning the abstract into the concrete
- Compilers and assemblers use symbolic references
- Assembler can resolve symbolic “fixups” within the same section
- Static linker can resolve offsets between sections and knows absolute addresses
- Dynamic linker can resolve references between shared libraries and relative displacements
- Relocations express the calculation that the linker or loader must perform

How is a relocation represented in an object file

```
int val;

int func(void) {
  return val;
}

clang --target=aarch64-linux-gnu -O2

.global val

0000000000000000 adrp x8, val
// 0000000000000000: R_AARCH64_ADR_PREL_PG_HI21    val
0000000000000004 ldr w0, [x8, :lo12:val]
// 0000000000000004: R_AARCH64_LDST32_ABS_LO12_NC  val
0000000000000008 ret
```

Differences when compiling with PIC

```
int val; // Preemptible

int func(void) {
  return val;
}

clang --target=aarch64-linux-gnu -O2  -fpic

.global val

0000000000000000 adrp x8,  :got: val
// 0000000000000000:  R_AARCH64_ADR_GOT_PAGE    val
0000000000000004 ldr  x8, [x8,  :got_lo12:val]
// 0000000000000004:  R_AARCH64_LD64_GOT_LO12_NC    val
0000000000000008  ldr w0, [x8]
000000000000000c ret
```

How does a linker resolve a relocation

- Relocation resolution is the last action before writing the file
  - Image layout must be complete
- Relocation code identifies all the actions a linker needs to take
  - Smart format, dumb linker
- Simplified steps, with S as VA of symbol, P as VA of place of relocation
  - Extract the addend A from the instruction (REL only)
  - Perform the calculation, most common ones are S + A (absolute) or S + A - P (relative)
  - Perform any alignment or overflow checks
  - Encode the result of the calculation in the instruction

How to read the ABI documentation

- Will also need the Architecture Reference manual for the instruction encodings
- Descriptions of relocation operators used in the tables
  - “**`GDAT(S+A)`** represents a pointer-sized entry in the **`GOT`** for address **`S+A`**. The entry will be relocated at run time with relocation **`R_<CLS>_GLOB_DAT(S+A)`**.”
- Tables of relocation codes, sometimes partitioned into related relocations
  - Almost always partitioned into static and dynamic
  - “Table 36 Table 4-14, GOT-relative instruction relocations”
- At a minimum each relocation has code, calculation, remarks
  - Fill in the gaps with the Architecture reference manual and some imagination
- Relocations with special behavior such as PLT, GOT generation

Example R_AARCH64_ADR_PREL21

ELF for the 64-bit Arm Architecture table entry

275	11	R <CLS> ADR_PREL_PG_HI21	**Page(S+A)-Page(P)**	Set an ADRP immediate value to bits [32:12] of the X; check that **-2^{32} <= X < 2^{32}**

ADRP instruction Arm ARM entry

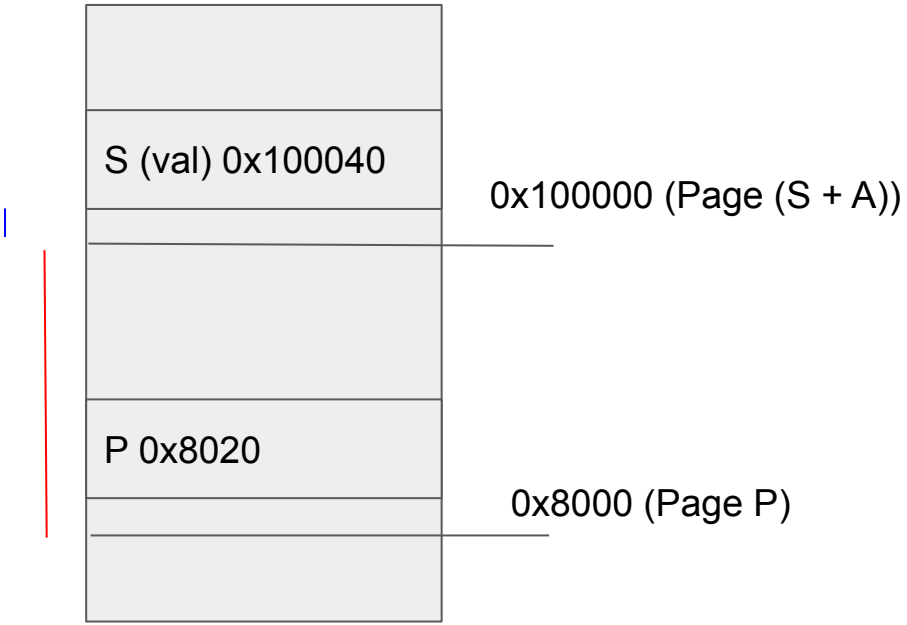
1	Immlo (2)	1 0 0 0 0	Immhi (18)	Rd (5)

```
imm = SignExtend(immhi:immlo:Zeros(12), 64);
```

Linker implementation

- **imm** = checkIntOverflow(((**S** + **A**) & ~0xfff) - (**P** & ~0xfff)), 33) >> 12
- **immLo** = (**imm** & 0x3) << 29 ; **immHi** = (**imm** & 0x1ffffc << 3);
- **immMask** = (0x3 << 29) | (0x1ffffc << 3)
- write32le (buf, (read32le(buf) & ~ **immMask**) | **immLo** | **immHi**);

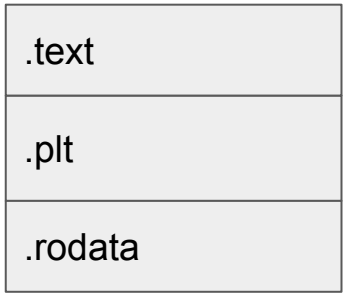
Visualisation




```
R_AARCH64_ADR_PREL21 :
Page (S + A) - Page (P) =
0x100000 - 0x8000 = 0xf8000
adrp x8, 0xf800
x8 = 0x8000 + f8000 = 0x10000

R_AARCH64_LDST32_ABS_LO12_NC
(S + A) & 0xffc
0x100040 & 0xffc = 0x40
ldr w0, [x8, #40]
```

GOT and PLT generating relocations



Read-only
no relocations
shareable

Read-only after
relocation
not-shareable

Read-write
not-shareable

Procedure **L**inkage **T**able **PLT**

- Linker generated stubs to call functions in shared libraries
- Ifuncs
- Load addresses from the .got.plt
- Lazy binding makes .got.plt RW
- Entry generated by call or jump relocation to function that could be defined externally
- .rela.plt

Global **O**ffset **T**able **GOT**

- Pointers to data
- Resolved at load time
- Can be made read-only after relocation
- Relocations can be simplified to R_<arch>_RELATIVE
- .rela.dyn

GOT generating relocation example

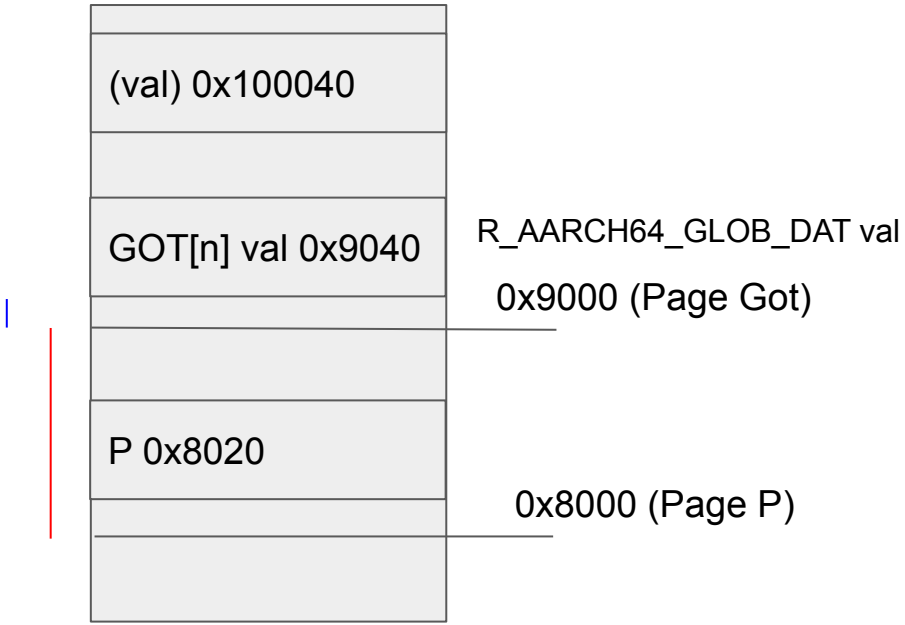
ELF for the 64-bit Arm Architecture entry

311	16	R <CLS> ADR_PREL_PG_HI21	**Page(G(GDAT(S+A))) - Page(P)**	Set an ADRP immediate value to bits [32:12] of the X; check that **-2^{32} <= X < 2^{32}**

- **GDAT(S+A)** represents a pointer-sized entry in the **GOT** for address **S+A**. The entry will be relocated at run time with relocation **R_<CLS>_GLOB_DAT(S+A)**.
- **G(expr)** is the address of the GOT entry for the expression expr.

- Linker creates GOT slot for val, if it doesn't already exist
- Linker creates `R_AARCH64_GLOB_DAT` relocation for got slot with symbol val
- Linker evaluates Page(Address of got slot) - Page (position)
- Linker writes back the offset to the page that the GOT slot is in.
- The `R_AARCH64_LD64_GOT_LO12_NC` loads the low 12 bits as in previous example

Visualisation




```
R_AARCH64_ADR_GOT_PAGE:
Page G(GDAT(S + A)) - Page (P)
=
0x9000 - 0x8000 = 0x1000
adrp x8, 0x1000
x8 = 0x8000 + 0x1000 = 0x9000

R_AARCH64_LD64_GOT_LO12_NC
G(GDAT(S + A)) & 0xffc
0x9040 & 0xffc = 0x40
ldr x8, [x8, #40]
```

Relative relocations

- The R_<Arch>_GLOB_DAT dynamic relocation is expensive to resolve
  - Dynamic loader has to lookup symbol by name
- When the definition of a symbol is DSO Local then the R_<Arch>_RELATIVE dynamic relocation can be used
  - Linker writes the value of the address at static link time into the GOT slot
  - Relative relocation is the displacement from the static link address to the runtime address
  - A simple addition, no symbol lookups, all relative relocations get the same value
  - Sorted before symbol lookups for ease of processing by the dynamic loader
- PIE binaries often have large numbers of relative relocations
- Android compresses relative relocations

Further topics not covered today

- Lazy binding via PLT sequences
- Thread local storage models and relaxation
- Linker generated thunks
- Garbage collection and identical code folding
- String and constant merging
- Comdat groups
- Linker Scripts
- Symbol ordering files