Linkerek és Programbetöltők

azaz hogyan jutunk a gépi kódtól a `main()`-ig

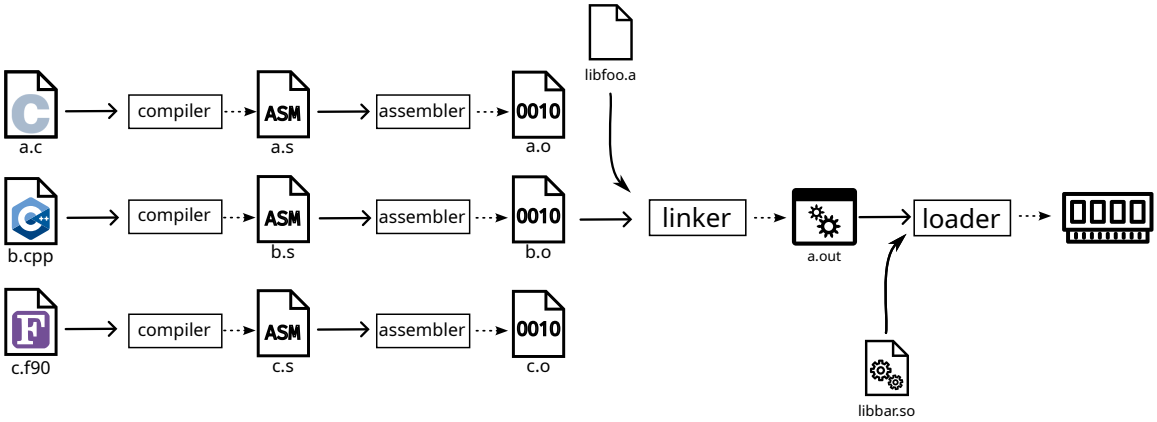
Bertalan Dániel

ELTE Bolyai Kollégium

2023. október 24.

Outline

A fordítás folyamata



Lépésenként – manuálisan

Próbáljuk meg kézzel replikálni mindazt, ami a GCC futtatása során történik.

Fordítás

```
gcc -S example.cpp -o example.s
```

Assembling

```
as example.s -o example.o
```

Linkelés

```
ld example.o -o example
```

Lépésenként – manuálisan

Hibaüzenet

```
$ ld example.o -o example
ld: warning: cannot find entry symbol _start; defaulting to 00401000
ld: example.o: in function `main':
example.cpp:(.text+0×f): undefined reference to `printf'
```

A linker nem képes csupán ennyi információval egy programot sikeresen elkészíteni, valami még hiányzik. Ez pedig a C standard könyvtárra való hivatkozás.

Egyébként a GCC/Clang -### argumentumával kiírathatjuk, hogy pontosan milyen paraméterekkel vannak a különféle lépések meghívva.

Programkönyvtárak

- Cél: Egymáshoz szorosan kapcsolódó kód csoportosítása és újrafelhasználhatóvá tétele
- **Statikus linkelés:**
  - Tulajdonképpen object file-ok egy fájlba "gyúrása" $\Rightarrow$ UNIX *archive* elnevezés
  - A **meghivatkozott** függvények, változók a kimeneti fájlba bemásolódnak
  - Semmi overhead a saját magunk által megírt kódhoz képest
  - Frissítés csak újra-linkeléssel
  - `.a` (*NIX), `.lib` (Windows)
- **Dinamikus linkelés:**
  - Önálló állomány, melyre a program név szerint hivatkozik
  - Indításkor **egészét** a memóriába másolja a programbetöltő (*loader*)
  - Függvényhívás költségesebb, csak pointer-indirekción keresztül
  - A könyvtár a rá hivatkozó kódtól függetlenül frissíthető
  - `.so` (ELF), `.dylib` (macOS), `.dll` (Windows)

Egy program részei

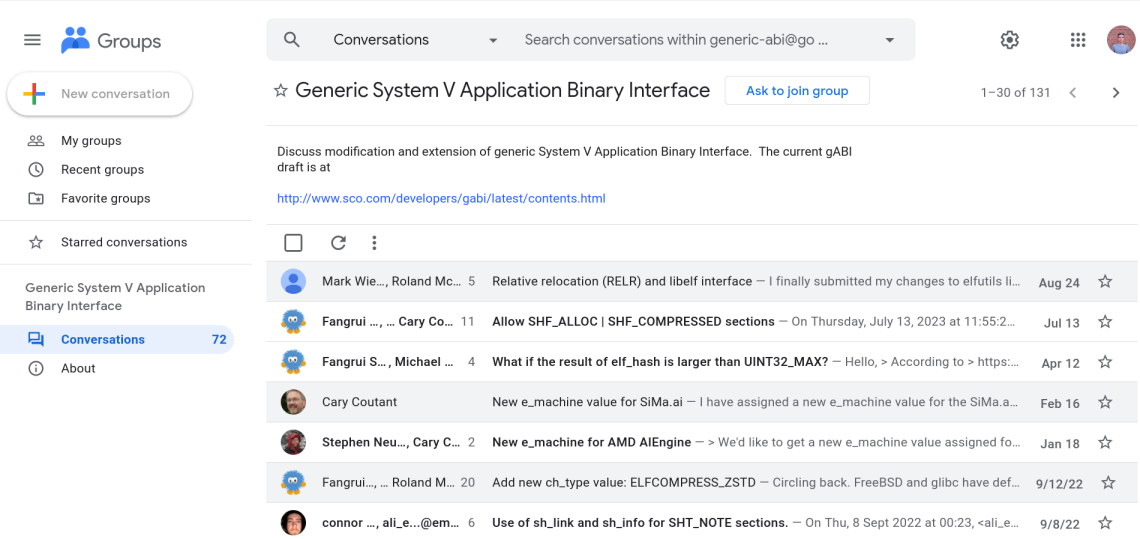
- `.rodata`: csak olvasható adat (pl. globális konstansok)
- `.data`: írható-olvasható adat (pl. globális/function-local `static` változók)
- `.text`: gépi kód
- `.bss`: 0-ra inicializált adat
- rendszertől függő metaadat
  - definiált szimbólumok listája
  - hivatkozások listája
  - programbetöltő számára utasítások
  - ...

Outline

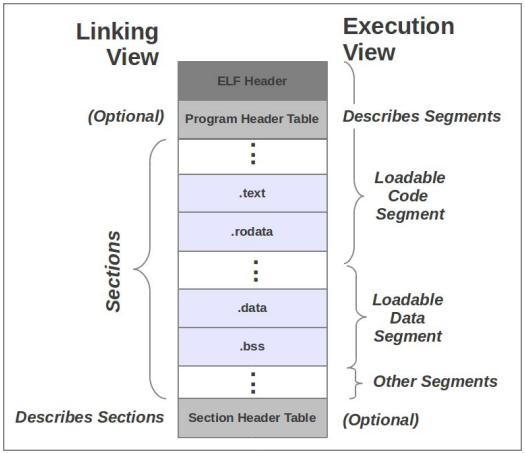
Az ELF fájlformátum

- Executable and Linkable Format
- System V Release 4 (1988.)
- általános konténer formátum
  - futtatható állomány, shared library, tárgykód, coredump
  - 180+ architektúra: x86, AArch64, RISC-V, ...
  - Linux, *BSD, commercial UNIX, beágyazott rendszerek
    - kivétel: Mach-O: Darwin/macOS, PE: Windows
  - big-endian, little-endian, 32 és 64 bit
- élő szabvány
  - `www.sco.com/developers/gabi` – 2013. június
  - `generic-abi` levelezési lista (Cary Coutant) – aktív
    - tervben van a GitHub-on publikálás
    - új architektúrák, új section típusok

Az élő szabvány



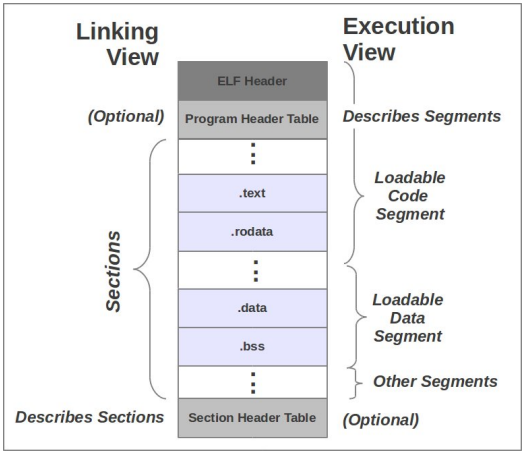
Egy ELF felépítése



Forrás: *nairobi-embedded.org*

1. ELF Header
   - fájlszignatúra: `"\x7fELF"`
   - ABI, CPU, fájltípus
   - belépési pont (`_start`) memóriacíme
   - további táblák címei: Section és Program Headerek
2. Program Header-ek
   - programbetöltéshez információ, fájlban összefüggő blokkok (több section)
   - `PT_LOAD`: futás közben betöltendő
   - `PT_DYNAMIC`: programbetöltő számára információ
   - `PT_INTERP`: programértelmező (betöltő) neve

Egy ELF felépítése



Forrás: *nairobi-embedded.org*

3. Section Header-ek
   - linkelés során felhasznált információ
   - kód és adat (`.text`, `.data`, ...)
   - szimbólumok nevei (`.symtab`) és adatai (`.strtab`)
   - exportált szimbólumok: `.dynsym`
   - relocation-ök (linkelés során módosítandó adat): `.rela`
   - DWARF debug info (`.debug_line`, `.debug_info`, ...)
   - ...

Hivatkozások

example.cpp

```
extern int global;

static int foo(int x) {
  return global + x;
}

int main(int argc, char**) {
  printf("%d\n", foo(argc));
}
```

example.s

```
  .section .rodata.1,"aMS",@progbits,1
.LC0: .string  "%d\n"
  .text
foo(int):
    movq  global@GOTPCREL(%rip), %rax
    addl  (%rax), %edi
    movl  %edi, %eax
    ret
main: call  foo(int)
      leaq  .LC0(%rip), %rdi
      movl  %eax, %esi
      xorl  %eax, %eax
      call  printf@PLT
      xorl  %eax, %eax
      ret
```

Hivatkozások

Hivatkozás globális változókra

`global@GOTPCREL`

- Address Space Layout Randomization: dinamikus könyvtárak címe nem ismert
- `W^X`, memória megosztás: a programbetöltő nem írhat bele futtatható szegmensbe
- megoldás: indirekció bevezetése – a változók abszolút címének tárolása lokálisan (relatív címzés) $\Rightarrow$ **Global Offset Table**
- ha mégis lokálisan van definiálva, a linker közvetlen eléréssé relaxálhatja

example.s

```
  .section .rodata.1,"aMS",@progbits,1
.LC0: .string  "%d\n"
  .text
foo(int):
    movq  global@GOTPCREL(%rip), %rax
    addl  (%rax), %edi
    movl  %edi, %eax
    ret
main: call  foo(int)
      leaq  .LC0(%rip), %rdi
      movl  %eax, %esi
      xorl  %eax, %eax
      call  printf@PLT
      xorl  %eax, %eax
      ret
```

Hivatkozások

Külső függvények meghívása

`printf@PLT`

- probléma: általában sok importált függvény, ezek megkeresése lassú
- megoldás: lustaság, első híváskor ugrás a loaderbe ⇒ **Procedure Linkage Table**
- manapság részben szerepét vesztette (PLT felülírás sérülékenység): `LD_BIND_NOW`, `-z now` és `-fno-plt`
- ELF Symbol Interposition szabály: alapértelmezetten minden felülírható (pl. `LD_PRELOAD`), még ha van lokális definíció is

example.s

```
  .section .rodata.1,"aMS",@progbits,1
.LC0: .string  "%d\n"
  .text
foo(int):
    movq  global@GOTPCREL(%rip), %rax
    addl  (%rax), %edi
    movl  %edi, %eax
    ret
main: call  foo(int)
      leaq  .LC0(%rip), %rdi
      movl  %eax, %esi
      xorl  %eax, %eax
      call  printf@PLT
      xorl  %eax, %eax
      ret
```

Hivatkozások

Lokális függvények, változók

- modern architektúrákon van PC-relatív címzés – elég offsetet tárolni
  - kivétel: i386 `__x86.get_pc_thunk.bx`
- globális konstans pointerek (pl. C++ vtable): címzés az első szegmenshez viszonyítva
  - a loader hozzáadja a könyvtár memóriacímét: `R_$ARCH_RELATIVE` relocation

example.s

```
  .section .rodata.1,"aMS",@progbits,1
.LC0: .string  "%d\n"
  .text
foo(int):
    movq  global@GOTPCREL(%rip), %rax
    addl  (%rax), %edi
    movl  %edi, %eax
    ret
main: call  foo(int)
      leaq  .LC0(%rip), %rdi
      movl  %eax, %esi
      xorl  %eax, %eax
      call  printf@PLT
      xorl  %eax, %eax
      ret
```

Outline

Linkelés I

- a programfordítás végső része; a linker (*szerkesztő*) végzi
- tárgykód fájlokból egyetlen kimeneti futtatható állomány vagy dinamikus könyvtár készítése

1. tárgykód-fájlok és könvytárak neveinek beolvasása balról jobbra
   - tárgykód: a fájl teljes tartalma bekerül a kimenetbe
   - dinamikus könyvtár: hivatkozás a nevére/elérési útvonalára
   - statikus könyvtár: azon elemek, melyek hivatkozást oldanak fel
     section nem bontható fel – ha egy függvény meg van hivatkozva, a TU összes többije is bekerül

Kitérő: Egy linker által feldolgozott adat mennyisége

Chromium 105 nightly (105.0.5126.0) release build arm64 macOS platformra:

- 11304 tárgykód-fájl
- 1,3 GB input
- 303 MB output
- 893 628 symbol
- 829 816 dinamikus relocation (”fixup”)
- `ld64.lld`: 3,7 s futási idő (Mac Mini M1)

Köszönet Nico Webernek (`@thakis`) az adatokért!

Linkelés I

- a programfordítás végső része; a linker (*szerkesztő*) végzi
- tárgykód fájlokból egyetlen kimeneti futtatható állomány vagy dinamikus könyvtár készítése

1. tárgykód-fájlok és könvytárak neveinek beolvasása balról jobbra
   - tárgykód: a fájl teljes tartalma bekerül a kimenetbe
   - dinamikus könyvtár: hivatkozás a nevére/elérési útvonalára
   - statikus könyvtár: azon elemek, melyek hivatkozást oldanak fel
     section nem bontható fel – ha egy függvény meg van hivatkozva, a TU összes többije is bekerül
2. section-ök csoportosítása, címek hozzárendelése
3. relocation-ök feldolgozása
   - lokális függvényhívás: relatív címzéssé alakul
   - globális pointer változók: dinamikus relocation a kimenetbe
   - könyvtárban definiált változó, függvény: GOT és/vagy PLT bejegyzés létrehozása

Linkelés II

5. metaadatok generálása (LLD: *synthetic sections*)
   - dinamikus könyvtárak – exportált nevek tárolása hasítótáblában
   - Build ID: globálisan egyedi azonosító (pl. tartalom SHA256-ja), külön tárolt debug info hozzárendeléséhez
   - string deduplikálás optimizáció
6. kimeneti fájl kiírása

- bottleneck: nem párhuzamosítható, csak az összes compiler processz végezte után indulhat
- Google: GOMA distributed compiler
  - fejlesztő saját gépéről több száz távoli compiler job a Google szerverein
  - Chrome böngésző 1-2 perc alatt
  - linkert csak lokálisan lehet futtatni – fontos a sebesség
- `mold`: amit csak lehet, párhuzamosan!
  - csak 2x (!) annyi idő alatt fut le mint a `cp`

`ld64.lld` Performance Trace

```
Load input files: 2068 ms                [46.30%]
Replace common symbols: 2 ms             [ 0.04%]
Gathering input sections: 31 ms          [ 0.70%]
markLive: 141 ms                         [ 3.16%]
Fold identical literals: 47 ms           [ 1.06%]
Fold Identical Code Sections: 557 ms     [12.47%]
Scan symbols: 98 ms                      [ 2.20%]
Scan relocations: 122 ms                 [ 2.72%]
Create output sections: 9 ms             [ 0.20%]
Sort segments and sections: 111 ms       [ 2.47%]
Finalize addresses: 233 ms               [ 5.21%]
Finalize __LINKEDIT segment: 162 ms      [ 3.62%]
Apply linker optimization hints: 26 ms   [ 0.58%]
Computing UUID: 6 ms                     [ 0.13%]
Write code signature: 256 ms             [ 5.73%]
Write output file: 596 ms                [13.35%]
```

Programbetöltés

1. ELF fájlszignatúra felismerése (vö. shebang)
2. `PT_INTERP` Program Header keresése
   - ha nincs: teljesen statikusan linkelt program (pl. maga a programbetöltő)
3. programbetöltő meghívása, Auxiliary Vectorban FD a futtatandó programra
4. `PT_LOAD` szegmensek betöltése a memóriába (`mmap()`)
5. relocation-ök elvégzése, initializer függvények megfelelő sorrendben

```
$ llvm-readelf --program-headers /bin/ls
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  PHDR           0x000040 0x0000000000000040 0x0000000000000040 0x0002d8 0x0002d8 R   0x8
  INTERP         0x000318 0x0000000000000318 0x0000000000000318 0x00001c 0x00001c R   0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x000000 0x0000000000000000 0x0000000000000000 0x003638 0x003638 R   0x1000
  [ ... ]
```

Outline

Name mangling I

- probléma:
  - C++ function overloading és generikus programozás – ugyanazzal a névvel eltérő dolgokra hivatkozhatunk
  - linkerek: a *symbol*-okat csak a nevük alapján különböztetjük meg
- az assemblyben kódolni szeretnénk az összes olyan információt, mely két objektumot megkülönböztet
  - névterek
  - template típus argumentumok
  - NTTP-k
  - függvényargumentumok típusai
  - lambda definíciójának helye
  - ...
- Microsoft ABI: másik séma, extra adat is: pl. `public`/`private` method, `class`/`struct` – olyan dolgokat is megkülönböztet, melyeket a nyelv nem

Name mangling II

```
$ c++filt _ZL3fooi
foo(int)
```

```
$ c++filt _Z3funIiEiv
int fun<int>(void)
```

```
$ c++filt _ZN7Example4testEiRNS_6StructE
Example::test(int, Example::Struct&)
```

```
$ c++filt _ZTVN3lld5macho12MarkLiveImplILb0EEE
vtable for lld::macho::MarkLiveImpl<false>
```

https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling

Name mangling III

Elfajuló eset: exponenciálisan növekvő backreference-ek

```
$ c++filt '_Z1f1TIS_IS_IS_IS_IiiiiiiiiiiiiiiiiiiiiiES0_S0_S0_S0_S0_S0_S0_S0_S0_
           S0_S0_S0_S0_S0_S0_S0_S0_S0_S0_ES1_S0_S0_S0_S0_S0_S0_S0_S0_S_IS0_S0_
           S0_S0_S0_S0_S0_S0_S0_S0_S1_S1_S1_S1_S1_S1_S1_S1_S1_S1_ES2_S2_S2_S2_
           S2_S2_S2_S2_S2_ES3_S2_S1_S0_S0_S0_S0_S0_S0_S0_S_IS2_S2_S1_S0_S0_S0_S0_
           S0_S0_S0_S3_S3_S3_S3_S3_S3_S3_S3_S3_S3_ES4_S4_S4_S4_S4_S4_S4_S4_S4_S4_
           ES4_S3_S2_S1_S0_S0_S0_S0_S0_S_IS4_S4_S3_S2_S1_S0_S0_S0_S0_S0_S5_S5_
           S5_S5_S5_S5_S5_S5_S5_S5_ES6_S6_S6_S6_S6_S6_S6_S6_S6_S6_E' | wc -c
 2383201444
```

Ez is Nico Webertől származik.

Statikus inicializáció

```
struct Foo {
  static int numFoos;
  Foo() {
    numFoos++;
  }
  ~Foo() {
    numFoos--;
  }
};
int Foo::numFoos;

Foo global;
```

Matt Godbolt: The Bits Between the Bits

- globális objektumoknak a `main()` kezdete előtt létre kell jönniük
- megoldás:
  - minden tárgykód-fájlban függvény, mely meghívja a konstruktorokat
  - destruktorok regisztrálása `__cxa_atexit` függvénnyel
  - a linker ezeket egy speciális `.init_array` section-be teszi
  - az *entry point*ra ugrás előtt ezek meghívása
  - sorrend: dependency-k előbb (tehát pl. `std::cout` felhasználói kódban már használható)

static és inline függvények

- static
  - alapvetően minden név globális látókörű: exportálás, *interposition* miatt PLT/GOT
  - lokális nevek: gyorsabb elérés, programbetöltésnél nincs overhead
  - azonos nevű objektumokat nem egyesíti a linker (azonos nevű segédfüggvények, DE: túl sok – méret megnő)
  - anonymous namespace: egész osztály (minden metódusa) elrejtése
  - `-fvisibility=hidden`: csak explicit `__attribute__((visibility("default")))` exportálódik
- inline
  - a definíciónak több TU-ban látszódnia kell (optimizálás, template)
  - többszörös definíció engedése (weak symbol), de egyezzenek meg!
  - a linker ezeket egyesíti
  - ODR – mindenhol egyezni kell a címeknek $\Rightarrow$ lokális fv. is GOT-indirekcióval
  - `-fvisibility-inlines-hidden` (non-conforming)
    - `https://github.com/SerenityOS/serenity/pull/20507`: 75%-kal kevesebb relocation

Outline

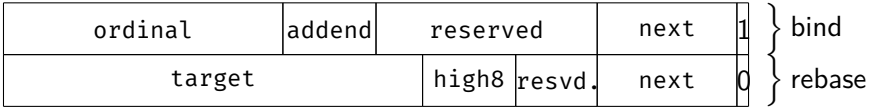
GNU: indirekt függvények

- processzorok fejlődése ⇒ új utasítások, szélesebb vektorregiszterek
- alapvető függvényekből (pl. `memcpy()`, `memset()`, `strlen()`) több verzióra igény
- minden híváskor CPU verzió szerinti elágazás drága
- glibc és binutils fejlesztők megoldása: *indirekt függvények*
- a függvény helyett egy `__attribute__((ifunc))` annotációval ellátott definíciót adunk
- betöltéskor a programbetöltő meghívja a függvényt, GOT-ba ez kerül

```
$ llvm-readelf --dyn-syms /usr/lib/libc.so.6 | grep IFUNC
136: 00000000000a1590    96 IFUNC   WEAK   DEFAULT    16 stpncpy@@GLIBC_2.2.5
194: 00000000000b9190   129 IFUNC   GLOBAL DEFAULT    16 wcsncmp@@GLIBC_2.2.5
220: 00000000000b7870   129 IFUNC   WEAK   DEFAULT    16 wcscmp@@GLIBC_2.2.5
221: 00000000000a3f50    96 IFUNC   GLOBAL DEFAULT    16 strncat@@GLIBC_2.2.5
[ ... ]
```

Apple: page-in linking

- a program indítását lassítja, hogy az összes relocation-t el kell végezni
- megoldás: kooperáció a loader (`dyld`) és a kernel között
- kernel a lap első elérésekor (page fault) végzi el a relocation-t
- láncolt listás ábrázolás
  - minden lap első relocation-jének rögzítése
  - relocation bitfield: offset rögzítése a következőhöz
  - külső függvények: könyvtár, függvény név külön táblában tárolva (`ordinal`)
- a relokáció után konstans adatok (pl. vtable) eldobhatók, a kernel újra tudja generálni őket



GSoC 2022 Final Report: Improvements to the Mach-O LLD linker (tavaly nyári munkám)
WWDC 2022: Link fast – Improve build and launch times

BOLT

- Meta Research (2019.), mára az LLVM része
- post-link optimizer (Facebook workload: 20-30% gyorsítás)
- klasszikus optimalizációk:
  - call-graph profile sorting (HFSort algoritmus)
  - identical code folding (LLD-ben is: `--icf=all`)
- klasszikus linkerekkel szemben kód megváltoztatása:
  - függvényen belül basic block-ok sorrendje (cache locality)
  - közelebb lévő függvényekre, változókra hivatkozás ⇒ rövidebb utasítások

```
adrp x0, var@PAGE          # page címzése ±4GB-ra
add  x1, x0, var@PAGEOFF   # 0-4095 immediate addend
ldr  x2, [x1]              # regiszter-relatív címzés
⇓
ldr x2, var                # 19 bit offset, ±1MB
```

További információk, Források I

- ELF fájlformátum
  - `www.sco.com/developers/gabi` – legújabb Generic ABI draft
  - `generic-abi` levelezési lista
  - ELF x86-64 psABI
- Itanium C++ ABI – mangling
- `https://maskray.me` – az ELF LLD maintainerének blogja
  - All about Global Offset Table
  - All about Procedure Linkage Table
  - ELF interposition and `-Bsymbolic`
- `https://github.com/rui314/mold`
- BOLT: A Practical Binary Optimizer for Data Centers and Beyond
- How programs get run: ELF binaries
- GSoC 2022 Final Report: Improvements to the Mach-O LLD linker (tavaly nyári munkám)

További információk, Források II

- Hasznos parancsok:
  - `objdump`
  - `readelf` (ELF) `otool` és `dyld_info` (Mach-O)
- Matt Godbolt: The Bits Between the Bits – programbetöltés, C runtime
- Michael Spencer: My Little Object File: How Linkers Implement C++
- Jez Ng, Nico Weber: LLD for Mach-O: The Journey
- WWDC 2022: Link fast – Improve build and launch times

Köszönöm a figyelmet!