自作ローダのためのlibc初期化ハック
Kernel/VM探検隊@東京 No16

Name	Akira Kawata
Biography	https://akawashiro.github.io/
twitter	@a_kawashiro
mastodon	mstdn.jp/@a_kawashiro

自己紹介

- 河田 旺 (Akira Kawata) / a_kawashiro
- 仕事
  - 深層学習ASIC向けのコンパイラ・ランタイム
  - sold - ちょっと変わったリンカ
    - https://github.com/pfnet/sold
- 非仕事
  - jendeley - JSONベースの文書管理ソフトウェア
  - ros3fs - S3 like用の読み取り専用FUSE
  - sloader - Linuxのローダ
    - 今日はこの話

ローダとは何か?

- 実行バイナリファイル(ELF)を起動するもの
  - **execv するたびにほとんど毎回呼ばれている**
  - x86-64 上のLinuxだと /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
- ローダの仕事
  - 与えられたELFファイルをメモリ上にロード
  - 依存する共有ライブラリを検索・ロード
  - シンボル解決等の実行時のバイナリ書き換え(再配置)

ローダの仕事

```
$ cat ./hoge.c
#include <stdio.h>
void hoge(){ puts("hoge\n"); }
$ cat ./hello.c
void hoge();
int main(){ hoge(); }
$ gcc -o libhoge.so -fPIC -shared hoge.c
$ gcc -o hello -fPIC hello.c libhoge.so
$ ./hello
hoge
```

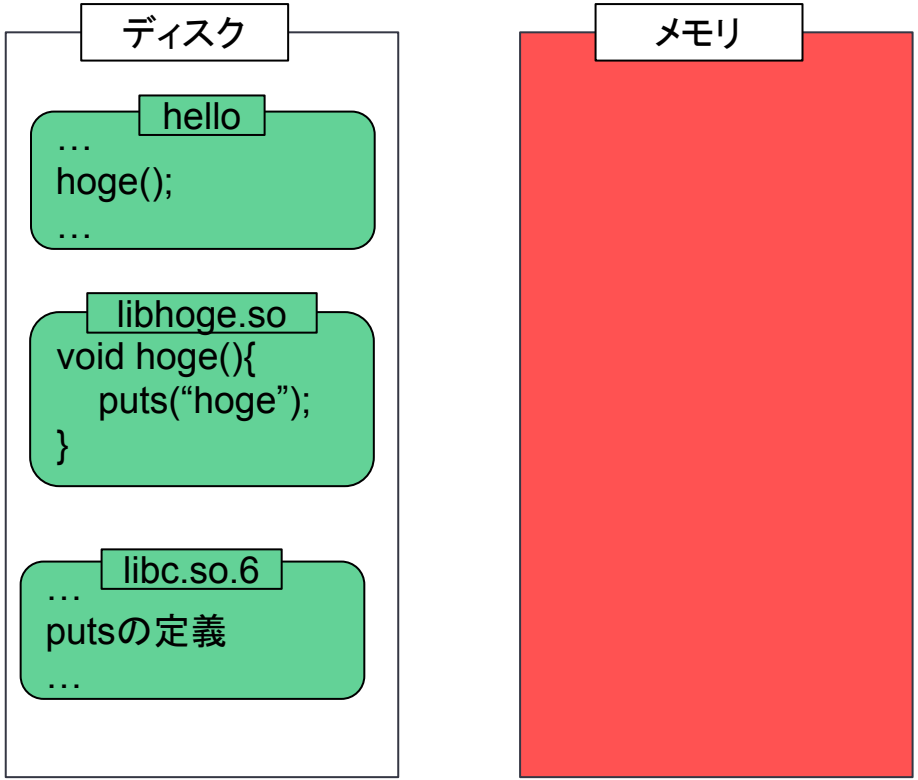
- hello は実行すると“hoge” と出力するプログラム
- libhoge.so の中の hoge() を呼び出す

ローダの仕事

```
$ ldd hello
    linux-vdso.so.1
    libhoge.so
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
    /lib64/ld-linux-x86-64.so.2
$ ldd libhoge.so
    linux-vdso.so.1
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
    /lib64/ld-linux-x86-64.so.2
```

- hello は libhoge.so に依存している
- libhoge.so は libc.so.6 に依存している

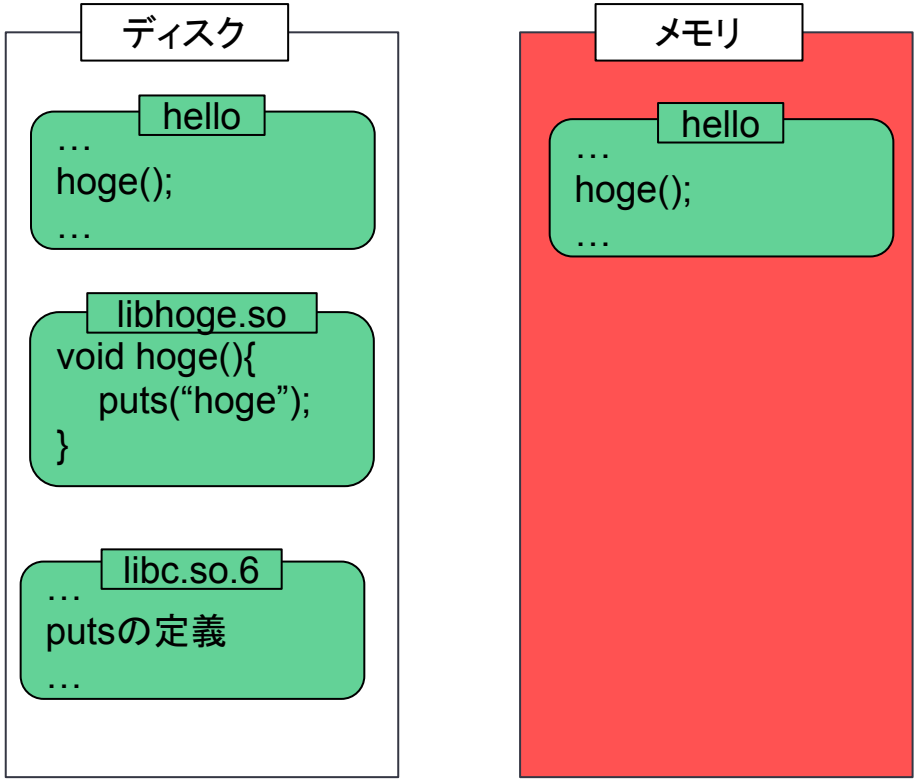
ローダの仕事



- 与えられたELFファイルをメモリ上にロード
- 依存する共有ライブラリを検索・ロード
- シンボル解決等の実行時のバイナリ書き換え(再配置)

説明の簡略化のため実際の起動プロセスとは違う部分があります。より正確な情報は How To Write Shared Libraries を参照してください。

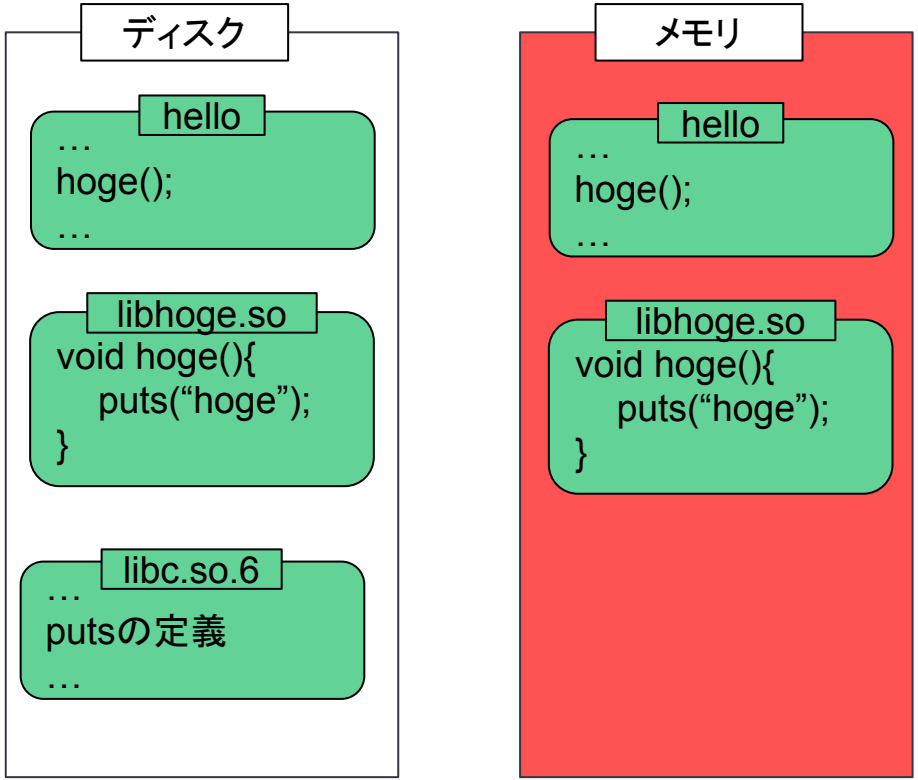
ローダの仕事



- 与えられたELFファイルをメモリ上にロード
- 依存する共有ライブラリを検索・ロード
- シンボル解決等の実行時のバイナリ書き換え(再配置)

説明の簡略化のため実際の起動プロセスとは違う部分があります。より正確な情報は How To Write Shared Libraries を参照してください。

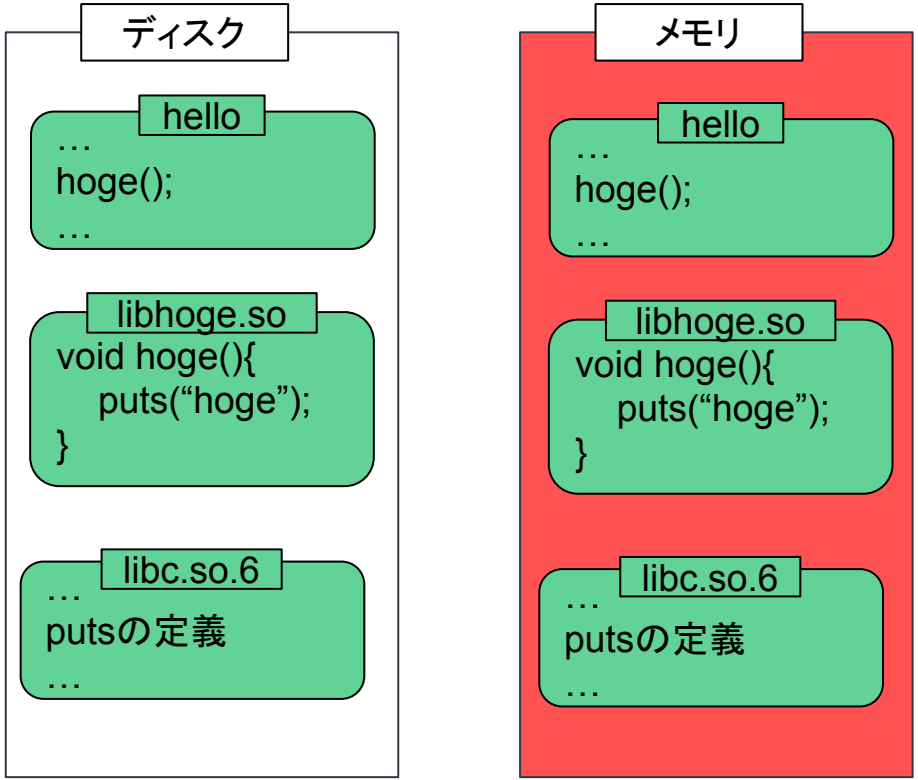
ローダの仕事



- 与えられたELFファイルをメモリ上にロード
- 依存する共有ライブラリを検索・ロード
- シンボル解決等の実行時のバイナリ書き換え(再配置)

説明の簡略化のため実際の起動プロセスとは違う部分があります。より正確な情報は How To Write Shared Libraries を参照してください。

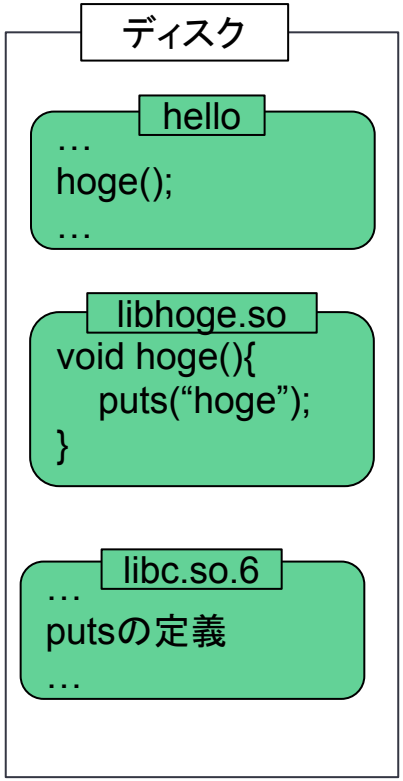
ローダの仕事

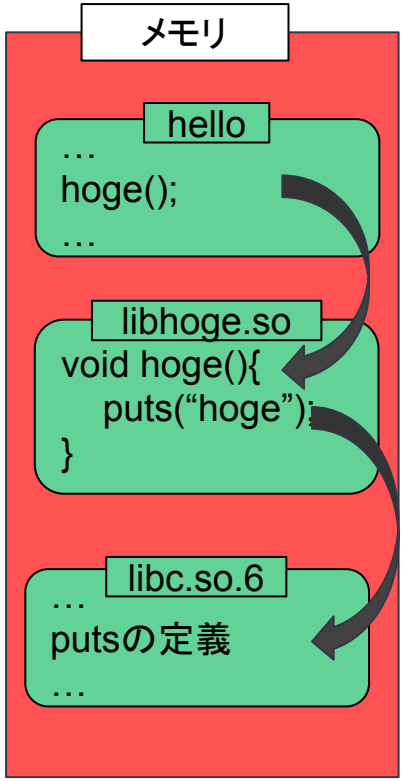


- 与えられたELFファイルをメモリ上にロード
- 依存する共有ライブラリを検索・ロード
- シンボル解決等の実行時のバイナリ書き換え(再配置)

説明の簡略化のため実際の起動プロセスとは違う部分があります。より正確な情報は How To Write Shared Libraries を参照してください。

ローダの仕事





jmp命令の引数にhogeのアドレスを埋める

call命令の引数にputsのアドレスを埋める

- 与えられたELFファイルをメモリ上にロード
  依存する共有ライブラリを検索・ロード
- シンボル解決等の実行時のバイナリ書き換え(再配置)

説明の簡略化のため実際の起動プロセスとは違う部分があります。より正確な情報は How To Write Shared Libraries を参照してください。

ローダを自作する動機

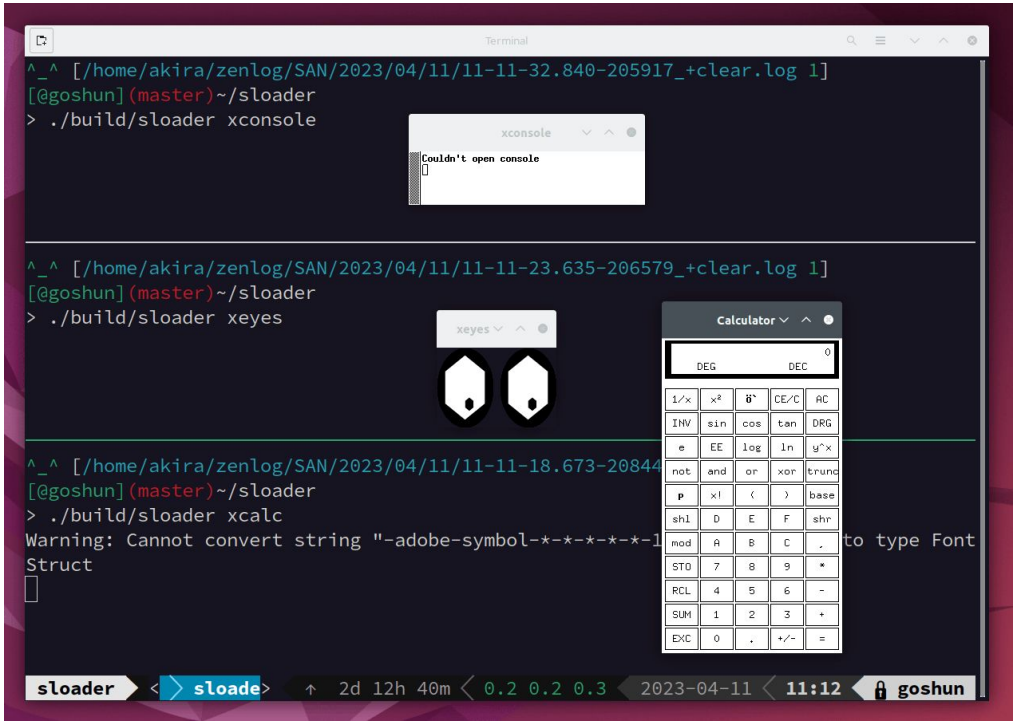
- 仕事でリンカ(sold)を作るのにローダの知識が必要
- ちょっと作ってみるか
  - もう2年ぐらいやっておりちょっとではない

sloader – Simple Loader

- https://github.com/akawashiro/sloader
- 対応しているアーキテクチャは x86-64だけ
  - Simple!
- C++で書いてある
  - Rustにすると glibc からソースコードをコピーできなくて不便
- 読みやすい
  - ld-linux-x86-64.so.2 は マクロだらけのC
- 自分が日常的に使うソフトウェアを全てロードするのが目標
  - cmake、g++、htop、firefox ...

最近のsloader

- 多くのソフトウェアが起動できるようになった
  - cmake、g++、ld、htop、ninja、xeyes、 ...
  - xeyesがなぜか六角形になっている
- firefoxはまだ起動できていない



自作ローダはまりポイント

- libc.so.6 の初期化
  - 多くのソフトウェアが依存しており避けられない
- Thread Local Storage (TLS)の初期化
  - 今日は時間の都合で話しません

libc.so.6 とは?

- 標準Cライブラリ
- https://www.gnu.org/software/libc/sources.html
- puts とか open とか非常によく使う関数が入っている
- /lib/x86_64-linux-gnu/libc.so.6 にある

ほとんどのソフトウェアはlibc.so.6 に依存している

```
$ find /usr/bin -mindepth 1 -maxdepth 1 | wc -l
2602
$ find /usr/bin -mindepth 1 -maxdepth 1 \
  | xargs ldd \
  | grep libc.so.6 \
  | wc -l
1949
```

- このため sloaderもlibc をロードする必要がある

libc.so.6 をロードするのは難しい

- ld-linux-x86-64.so.2 と libc.so.6 は同じリポジトリに入っている
  - libc.so.6 だけを分離して扱うことを考えていなさそう
- libc.so.6 をロードしようとすると...
  - mallocが失敗する
    - Thread Local Storage が壊れる
  - 半年ほど格闘するはめに

sloader での 対策

- **libc.so.6 をロードするのやめる!**
- sloader に静的リンクされているlibc.aを使う
  - 再配置中にlibc.so.6の中のシンボルが出てきたら
    sloaderの中の関数ポインタの値に解決する
  - ロードされたプログラム中のputsがsloaderの中のputsを指す
  - シンボルと関数ポインタの値との対応が libc_mapping.cc にある

libc_mapping.cc

- シンボルとsloader中の関数ポインタの値の対応が書いてある
- libcが外部に公開していない関数は自分でラッパを書いている
  - __memcpy_chk とか

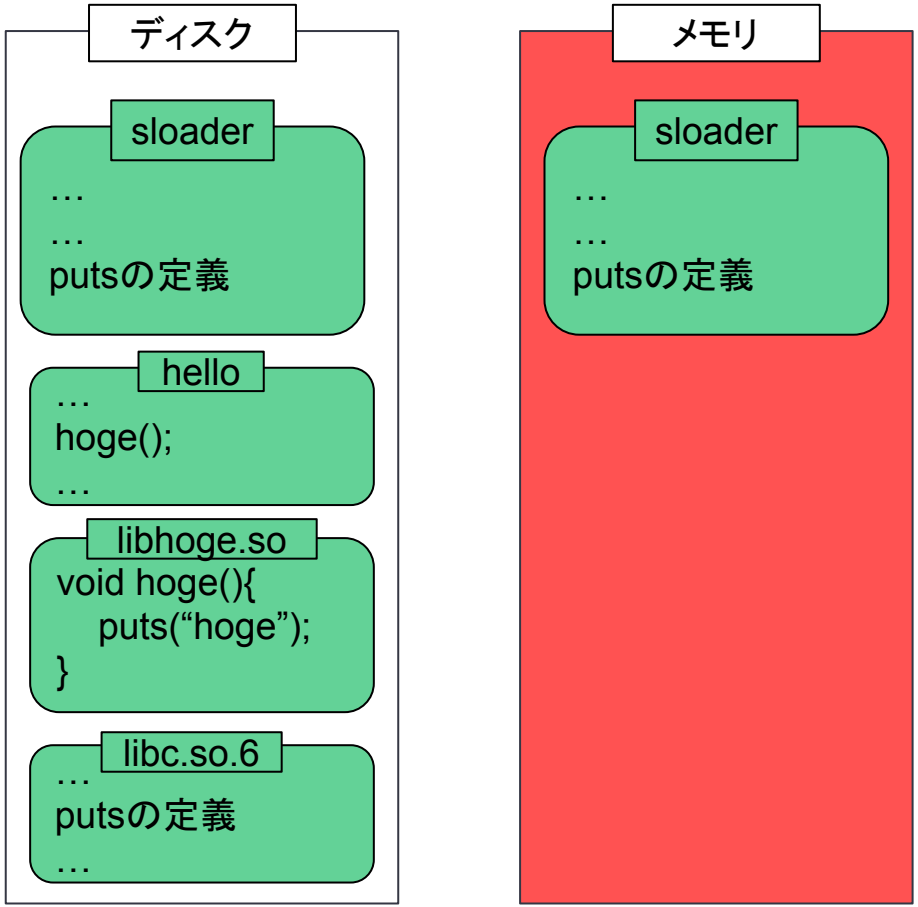
```
{"fchdir", reinterpret_cast<Elf64_Addr>(&fchdir)},
{"fchmod", reinterpret_cast<Elf64_Addr>(&fchmod)},
{"fchmodat", reinterpret_cast<Elf64_Addr>(&fchmodat)},
{"fchown", reinterpret_cast<Elf64_Addr>(&fchown)},
{"fchownat", reinterpret_cast<Elf64_Addr>(&fchownat)},
{"fclose", reinterpret_cast<Elf64_Addr>(&fclose)},
{"fcntl", reinterpret_cast<Elf64_Addr>(&fcntl)},
{"fcntl64", reinterpret_cast<Elf64_Addr>(&fcntl64)},
{"fdatasync", reinterpret_cast<Elf64_Addr>(&fdatasync)},
{"fdopen", reinterpret_cast<Elf64_Addr>(&fdopen)},
{"fdopendir", reinterpret_cast<Elf64_Addr>(&fdopendir)},
{"feof", reinterpret_cast<Elf64_Addr>(&feof)},
{"ferror", reinterpret_cast<Elf64_Addr>(&ferror)},
{"fexecve", reinterpret_cast<Elf64_Addr>(&fexecve)},
{"fflush", reinterpret_cast<Elf64_Addr>(&fflush)},
{"fflush_unlocked", reinterpret_cast<Elf64_Addr>(&fflush_unlocked)},
{"fgetc", reinterpret_cast<Elf64_Addr>(&fgetc)},
{"fgets", reinterpret_cast<Elf64_Addr>(&fgets)},
{"fgets_unlocked", reinterpret_cast<Elf64_Addr>(&fgets_unlocked)},
{"fgetwc", reinterpret_cast<Elf64_Addr>(&fgetwc)},
{"fgetxattr", reinterpret_cast<Elf64_Addr>(&fgetxattr)},
```

sloaderがhelloを起動する様子

```
$ cat ./hoge.c
#include <stdio.h>
void hoge(){puts("hoge\n"); }
$ cat ./hello.c
void hoge();
int main(){ hoge(); }
$ gcc -o libhoge.so -fPIC -shared hoge.c
$ gcc -o hello -fPIC hello.c libhoge.so
$ sloader ./hello
hoge
```

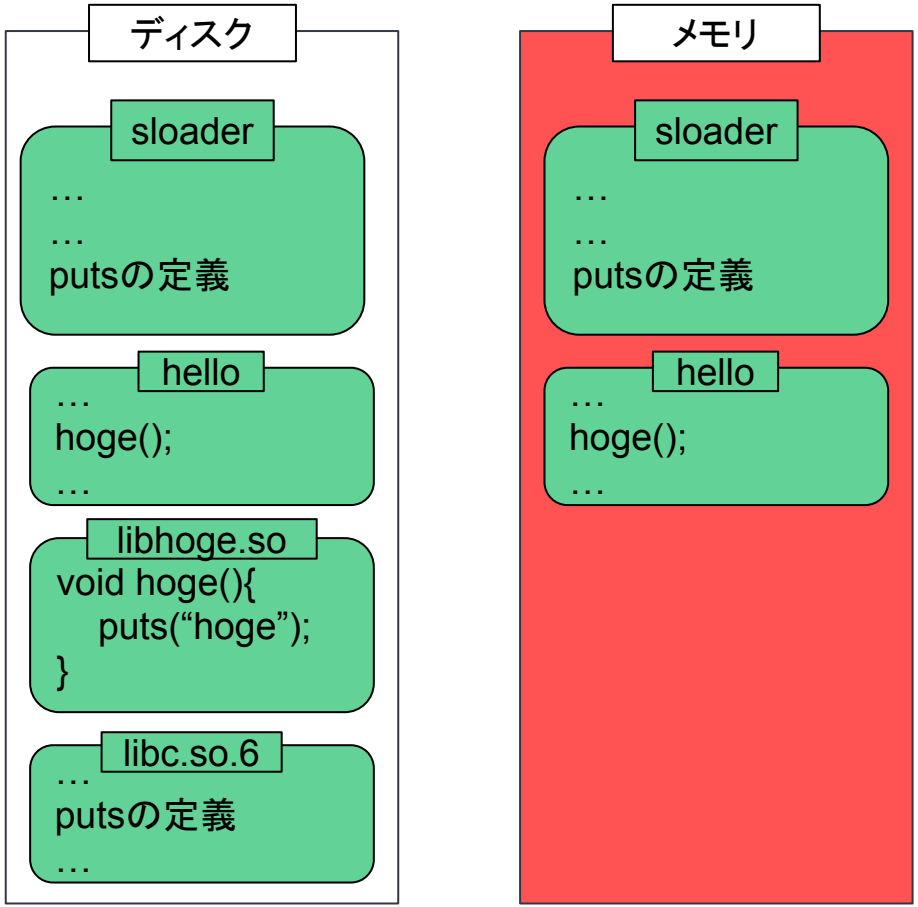
- hello は実行すると“hoge” と出力するプログラム
- libhoge.so の中の hoge() を呼び出す

sloaderがhelloを起動する様子



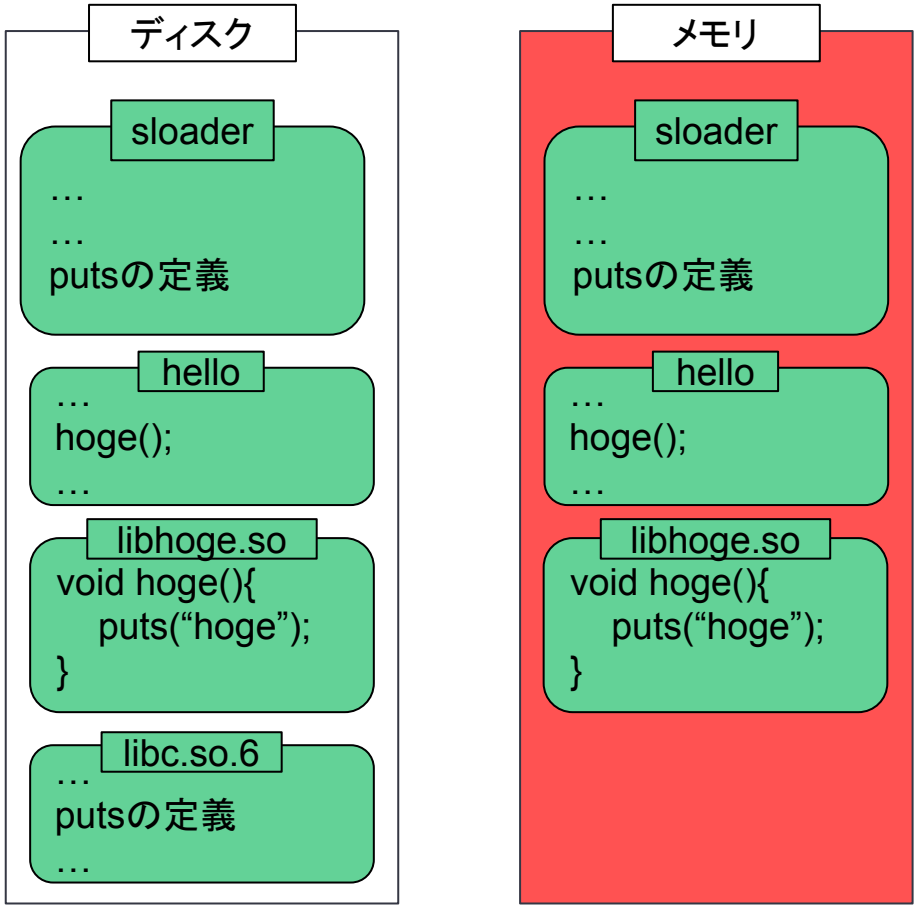
- 与えられたELFファイルをメモリ上にロード
- 依存する共有ライブラリを検索・ロード
- シンボル解決等の実行時のバイナリ書き換え(再配置)

sloaderがhelloを起動する様子



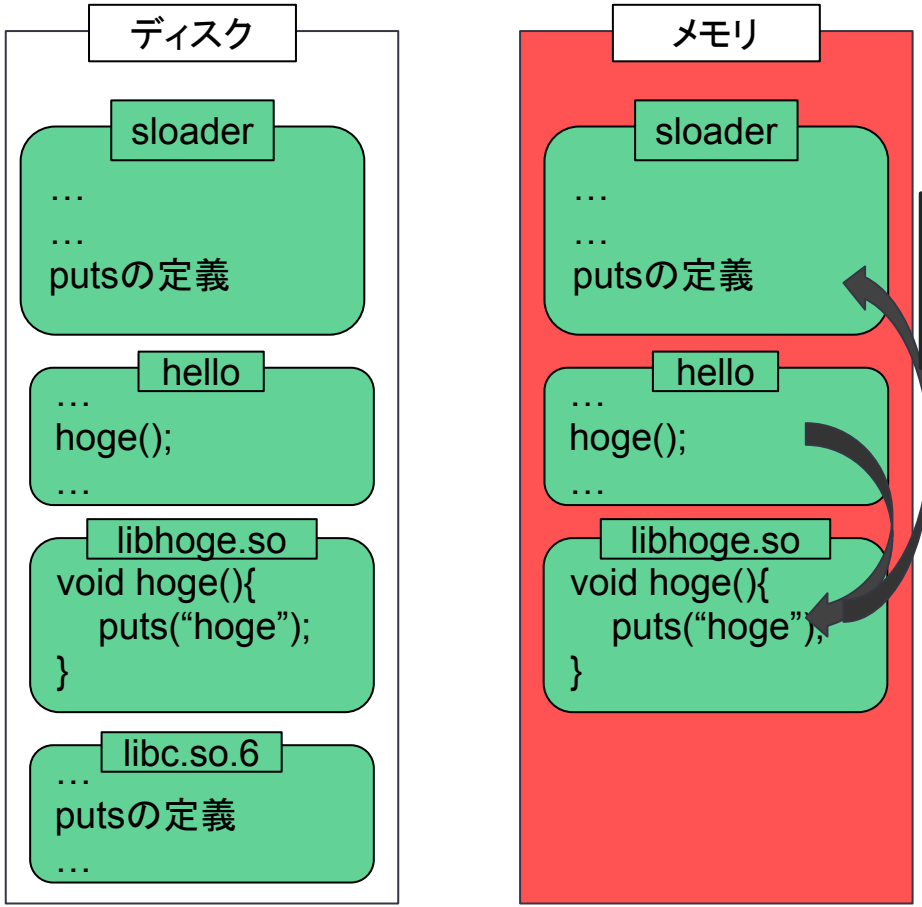
- 与えられたELFファイルをメモリ上にロード
- 依存する共有ライブラリを検索・ロード
- シンボル解決等の実行時のバイナリ書き換え(再配置)

sloaderがhelloを起動する様子



- 与えられたELFファイルをメモリ上にロード
- 依存する共有ライブラリを検索・ロード
- シンボル解決等の実行時のバイナリ書き換え(再配置)

sloaderがhelloを起動する様子



- 与えられたELFファイルをメモリ上にロード
- 依存する共有ライブラリを検索・ロード
- シンボル解決等の実行時のバイナリ書き換え(再配置)

現状のsloaderの問題点

- Thread Local Storage(TLS)周りがバグっている
  - sloaderとロードされたプログラムで
    同じTLS領域を使っているのが原因
    - initial_exec と local_dynamic アクセスモデルがバグっている
  - リンカスクリプトでなんとかしようとしている
- GUIアプリケーションの多くはまだ起動できない

まとめ

- 自作ローダ (sloader) を作成中
- libc.so を自分でロードするのは大変
- sloaderの中のlibcを使うハックを採用
- libcをうまくロードするアイデアを募集中

予備スライド

Thread Local Storage (TLS)の初期化

Thread Local Storage (TLS) とは?

- スレッドごとの固有の記憶領域
- C/C++だと thread_local で使える
- アクセスの方法が通常の変数とは大きく異なる
  - fsレジスタ + オフセットでアドレスを得る
  - __tls_get_addr関数を呼び出してアドレスを得る

TLSにfsレジスタ経由でアクセスする例 (gotbolt)

```
#include <stdint.h>

thread_local uint64_t i0;
int main() {
    i0 = 0xabcdabcdabcdabcd;
}
```

```
i0:
        .zero   8
main:
        push    rbp
        mov     rbp, rsp
        movabs  rax, -6067004223159161907
        mov     QWORD PTR fs:i0@tpoff, rax
        mov     eax, 0
        pop     rbp
        ret
```