組込み機器における動的ライブラリ使用時のメモリ使用量削減

2007.02.22
松下電器産業株式会社
山本　哲士
パナソニックモバイルコミュニケーションズ
水山　正重

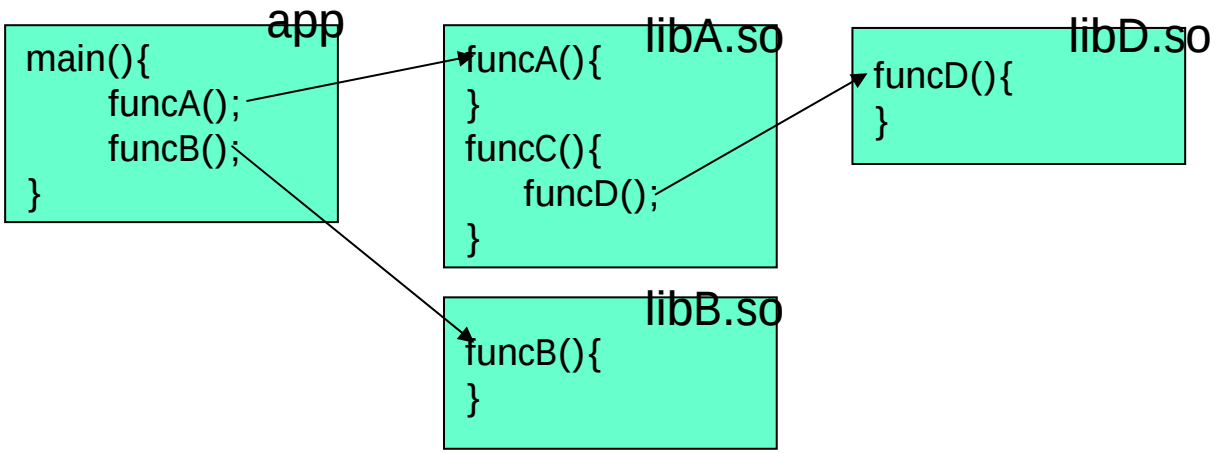
目次

- 背景
- 分析
- 方式検討
- 遅延ローディング(概要、考え方、実装)
- 効果
- 今後の課題

背景-1

- 携帯電話をはじめとする組込み機器では、各アプリでの機能の増大のため、動的ライブラリを多くリンクするケースが多くなってきている。(数10ライブラリをリンクするようなケースも存在)
- 動的ライブラリの依存関係が複雑になり、本来そのアプリで必要のないライブラリ、あるいはある機能を利用時のみしか使わないライブラリもリンクする必要がある。

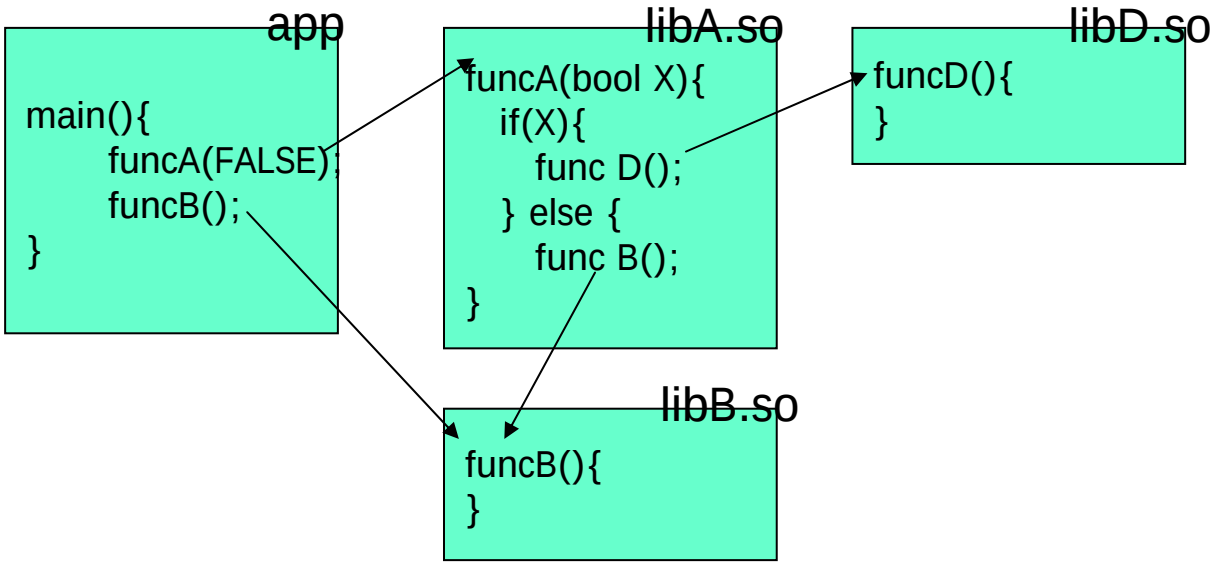
CASE1



上記依存関係では、libA,B,Dをリンクする必要があるが、実際にはlibD.soは使われず、そのライブラリが使用するメモリは無駄になる。

背景-2

CASE2:



上記構造では、動作時にはfuncD()は呼ばれることはないが、リンク関係は必要。

→動的ロード(dlopen(),dlsym())で対応するためには、ライブラリ、アプリの構造をすべて変更する必要があり、移行への敷居が高い。

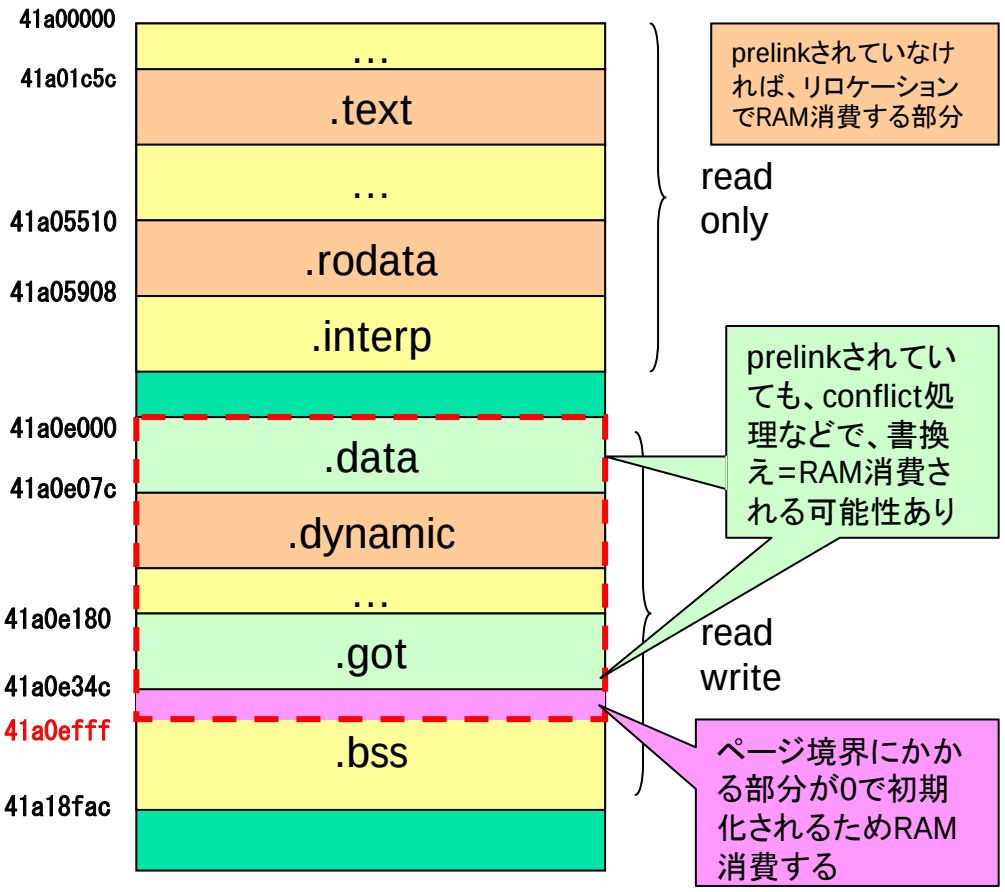
- このような、使用しないライブラリがロードされるだけで使われる無駄な税金分のメモリを削減したい。

分析-1

- ライブラリをロードするだけでどれだけのメモリを消費するか？
  →動的ライブラリをリンクすると、(使用しなくても) 1ページ/ライブラリ以上のRAMを消費する。(data領域とbss領域の境界が、page境界でない場合、bssの初期化のための0埋めする。(1ページを使う))
  →conflictで、.data/.gotを書き換える場合も、RAMを消費する。

ection Headers: (librt.so.1)

[Nr] Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]	NULL	00000000	000000	000000	00		0	0	0
[1] .note.ABI-tag	NOTE	000000f4	0000f4	000020	00	A	0	0	4
[2] .hash	HASH	00000114	000114	0004d0	04	A	3	0	4
[3] .dynsym	DYNSYM	000005e4	0005e4	000840	10	A	4	2a	4
[4] .dynstr	STRTAB	00000e24	000e24	0004ae	00	A	0	0	1
[5] .gnu.version	VERSYM	000012d2	0012d2	000108	02	A	3	0	2
[6] .gnu.version_d	VERDEF	000013dc	0013dc	00005c	00	A	4	3	4
[7] .gnu.version_r	VERNEED	00001438	001438	0000b0	00	A	4	2	4
[8] .rel.dyn	REL	000014e8	0014e8	0001c8	08	A	3	0	4
[9] .rel.plt	REL	000016b0	0016b0	0001d8	08	A	3	b	4
[10] .init	PROGBITS	00001888	001888	000014	00	AX	0	0	4
[11] .plt	PROGBITS	0000189c	00189c	0003c0	04	AX	0	0	4
[12] .text	PROGBITS	00001c5c	001c5c	003808	00	AX	0	0	4
[13] __libc_freeres_fn	PROGBITS	00005464	005464	0000a0	00	AX	0	0	4
[14] .fini	PROGBITS	00005504	005504	00000c	00	AX	0	0	4
[15] .rodata	PROGBITS	00005510	005510	0003f8	00	A	0	0	4
[16] .interp	PROGBITS	00005908	005908	000014	00	A	0	0	4
[17] .data	PROGBITS	0000e000	006000	000070	00	WA	0	0	4
[18] __libc_subfreeres	PROGBITS	0000e070	006070	000008	00	WA	0	0	4
[19] .eh_frame	PROGBITS	0000e078	006078	000004	00	A	0	0	4
[20] .dynamic	DYNAMIC	0000e07c	00607c	0000f0	08	WA	4	0	4
[21] .ctors	PROGBITS	0000e16c	00616c	000008	00	WA	0	0	4
[22] .dtors	PROGBITS	0000e174	006174	000008	00	WA	0	0	4
[23] .jcr	PROGBITS	0000e17c	00617c	000004	00	WA	0	0	4
[24] .got	PROGBITS	0000e180	006180	0001cc	04	WA	0	0	4
[25] .bss	NOBITS	0000e34c	00634c	00ac60	00	WA	0	0	4
[26] .comment	PROGBITS	00000000	00634c	00085e	00		0	0	1



分析-2

.bssの初期化箇所(ld.so (elf/dl-load.c))

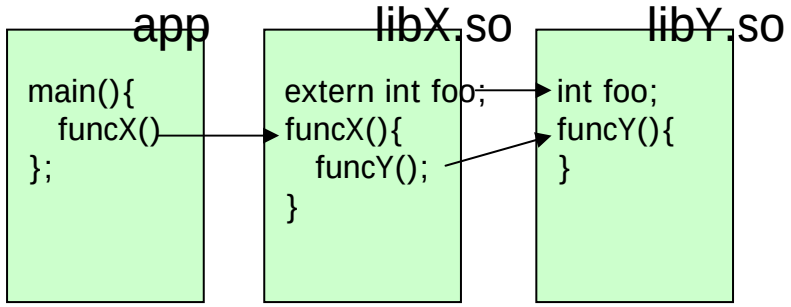
_dl_map_object_from_fd() 内

```
  ...
zero = l->l_addr + c->dataend;
zeroend = l->l_addr + c->allocend;
zeropage = ((zero + GL(dl_pagesize) - 1) & ~(GL(dl_pagesize) - 1));
    ...
if (zeropage > zero)
  {
        /* Zero the final part of the last page of the segment.  */
        if ((c->prot & PROT_WRITE) == 0)
          {
            /* Dag nab it.  */
            if (__builtin_expect (__mprotect ((caddr_t) (zero & ~(GL(dl_pagesize) - 1)),
                                  GL(dl_pagesize), c->prot|PROT_WRITE) < 0,  0))
  ...
          }
        memset ((void *) zero, '¥0', zeropage - zero);
        if ((c->prot & PROT_WRITE) == 0)
          __mprotect ((caddr_t) (zero & ~(GL(dl_pagesize) - 1)),
                      GL(dl_pagesize), c->prot);
  }
if (zeroend > zeropage)
  ...
```

参考(prelinkのconflictについて)

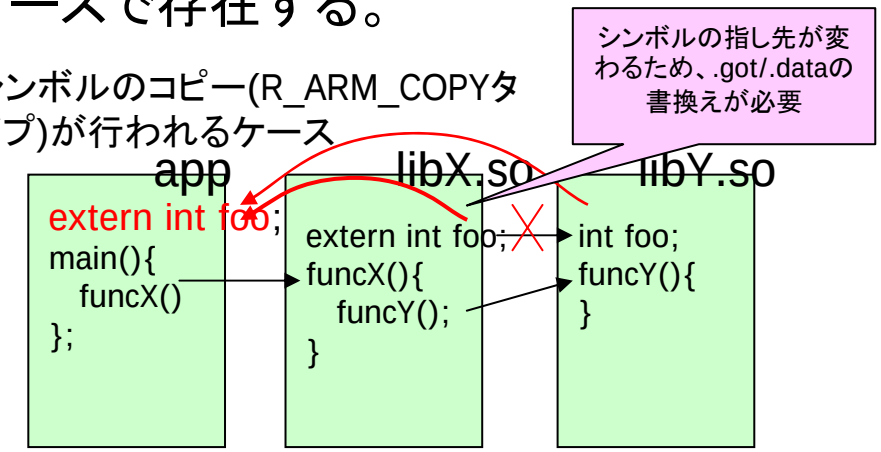
- conflictの発生する原因
  - prelinkにより解決されたシンボル→アドレスとの対応が、ライブラリが依存しているライブラリ群で解決する場合と、実行ファイルを含めて解決する場合とで異なるケースで存在する。

通常ケース



appがlibX.soに依存し、libX.soがlibY.soに依存しているケース
→この場合にはconflictはない。

シンボルのコピー(R_ARM_COPYタイプ)が行われるケース



実行ファイルで参照されるシンボルは、実行ファイル側に実体がコピーされるため、ライブラリは参照先を変更する必要がある。
(現在、ARMコンパイラではオプションの –znocopyrelocが有効でないため上記動作を抑制できない)
→libX.soのfooの指し先を app側に変更(conflict)する。

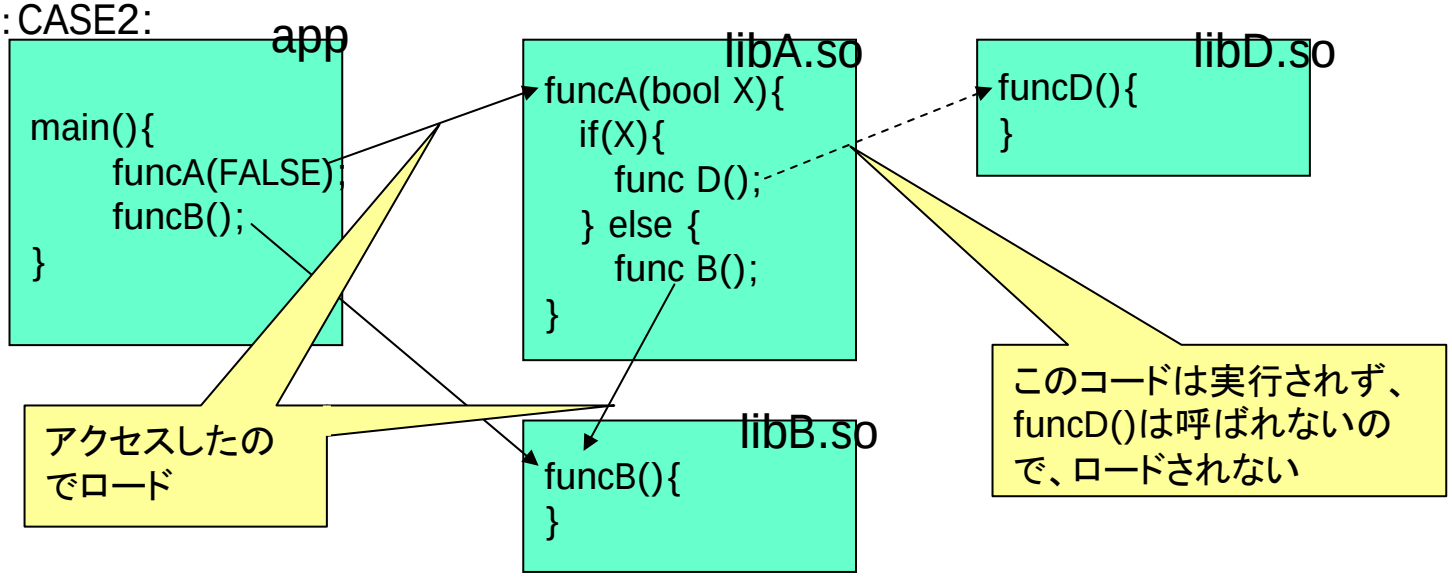
そのほかにも、ライブラリのリンク時の依存関係が不十分だったり、シンボルが2重定義されていたりすると、conflict情報が生成する。
prelinkでシンボル解決されたアドレスは.gotや.dataセクションに存在するため、それら領域が書き換えられる可能性がある。

方式検討

必要なライブラリだけをロードして動作するにはどうしたらよいか？

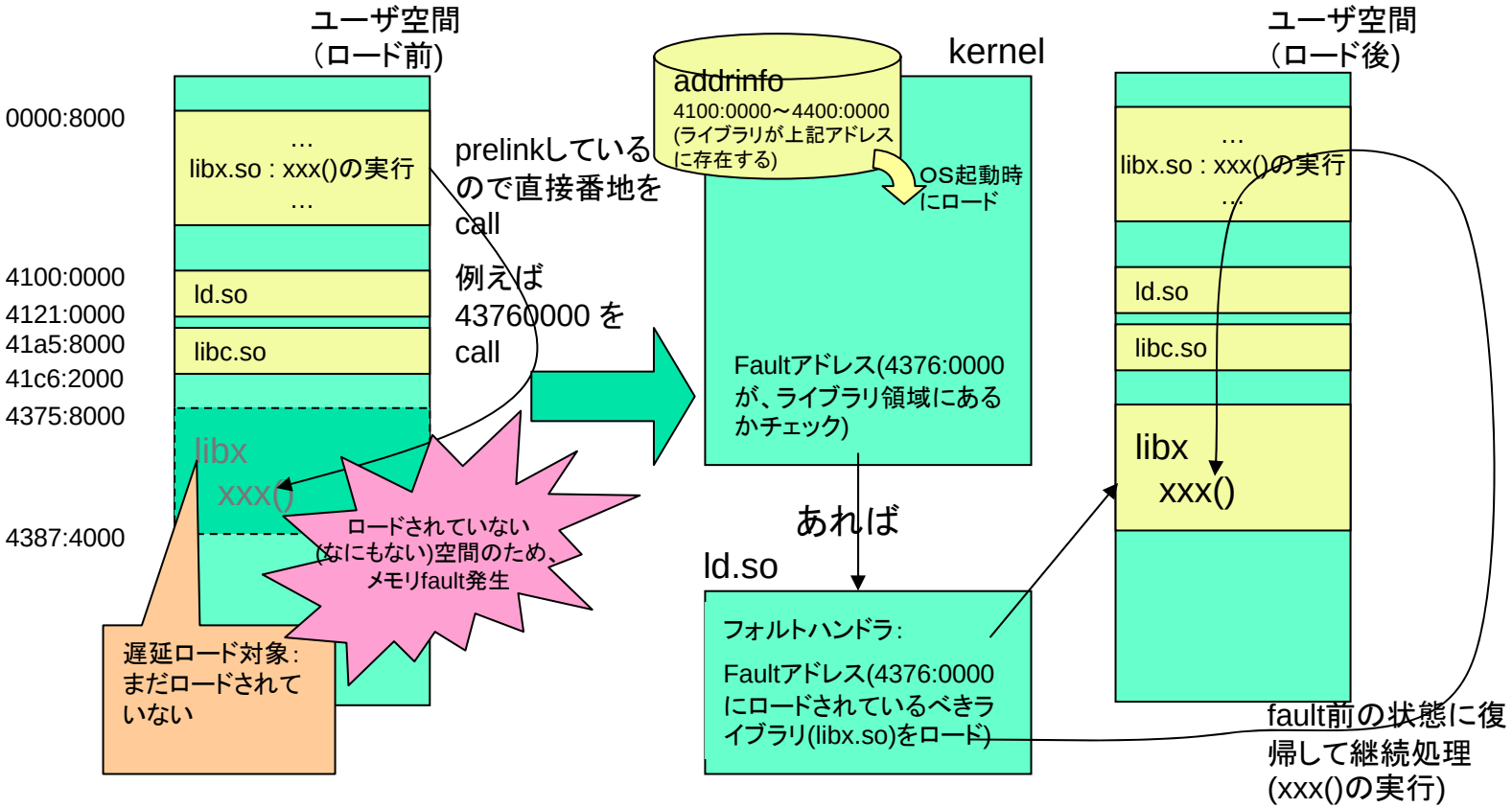
- 案1
  どのライブラリを使うか、全体で整合性を合わせて設計する。
  →正論ではあるが、数100にもおよぶライブラリの管理コストが大
- 案2
  動的なライブラリロード(dlopen())で実現する。
  →アプリ、ライブラリの実装コードをすべて変更していく必要がある。
  prelink時ができない問題があり、dlopen()処理時のオーバヘッド(シンボル解決処理)がかかる。
- 案3
  必要となった地点(ライブラリを実行/アクセスした地点)でローダ/OSが自動的にライブラリをロードする(lazyload)

例：CASE2:



遅延ローディング(仕組み)

- メモリfaultを利用して、ローダのハンドラを呼び出し、ライブラリの遅延ロード機能を実現



遅延ローディング(概要)

- 概要
  - 環境
    - MontaVista CEE3.1(kernel 2.4.20, glibc 2.3.2) をベースに、Linuxカーネル、glibc(ld.so)を修正
  - 前提条件
    - prelinkを行い、ロード番地が確定・BIND処理がされていること。

    →ロード番地を見て、フォルトの判断をしているため

    →シンボルのアドレス解決(BIND)が実施されていない場合、シンボル検索のため他ライブラリをロードしてしまい、RAM削減効果がなくなる
  - 動作概要

    (1) カーネルがライブラリアドレス情報をロード

    (2) プロセス起動時の処理

    遅延ローディングON判定
    ↓
    フォルトハンドラのカーネルへの登録
    ↓
    ライブラリのREAD ONLYでのmmap
    ↓
    ローダ内の構造体(link_info)の作成
    ↓
    ライブラリのunmap
    ↓
    main()へ

    (3) main()処理以降で、ライブラリをアクセスするとフォルト(セグメンテーションフォルト)発生し、ld.soのハンドラにJUMP →アドレスから、該当ライブラリをロードして復帰

遅延ローディング(変更箇所)

- 主な変更箇所
  - Linuxカーネル
    - arch/arm/kernel/call.S：システムコール追加
      (フォルトハンドラ登録用, フォルト時のレジスタ情報取得)
    - arch/arm/kernel/sys_arm.c：フォルト時、復帰PCアドレスの差し替え
    - arch/arm/kernel/dlfault.c(新規)：フォルト用のハンドラコード
    - arch/arm/mm/fault-common.c：メモリフォルト時の分岐
    - init/main.c：ライブラリアドレス情報の読み込み
  - glibc (ld.so)
    - elf/rtld.c：遅延ローディング判定、ON/OFFなど、フォルトハンドラなど
    - elf/dl-load.c：遅延ローディング用ロード情報保存、ロード処理
    - elf/dl-init.c：遅延ローディング用ライブラリ初期化処理
    - elf/conflict.c：遅延ローディング用conflict処理
    - include/link.h：遅延ローディング用変数追加(ロード管理、番地情報)
    - sysdeps/genelic/ldsodefs.h：遅延ローディング用変数追加(ON/OFF処理)
- patchについては、CELFで公開する予定

遅延ローディング(ソースコード:抜粋)

- 下記処理で、メモリフォルト→ハンドラへjump

フォルトハンドラ

→ do_translation_fault()

arch/arm/mm/fault-common.c

```
do_bad_area(struct task_struct *tsk, struct mm_struct *mm, unsigned long addr,
                int error_code, struct pt_regs *regs)
{
            /*
             * If we are in kernel mode at this point, we
             * have no context to handle this fault with.
             */
        if (user_mode(regs)){
                if(!search_dl_hash(addr)){  //  ライブラリエリアにあるか検索
                        dl_fault_savereg(tsk,regs,addr);  // レジスタ情報を退避
                        dl_fault_setpc(tsk,regs);  // 復帰時のアドレスをローダハンドラ
                                                   // に書換え
                }
                else{
                    __do_user_fault(tsk,addr,error_code,SEGV_MAPERR,regs);
                }
        }
        else
                __do_kernel_fault(mm,addr,error_code,regs);
}
```

遅延ローディング(ソースコード:抜粋)

- 下記処理で、カーネル→ロードハンドラ→復帰

kernel

elf/rtld.c

```
static void _dl_lazy_trap_handler( void ) {
  unsigned regs[17];
  unsigned addr;
  struct link_map *l = GL(dl_loaded);
  int found=0;
  SWI_ARG2(270, &addr, regs);              // レジスタ情報の取得
  /* search maps */
  for(;l;l = l->l_next){
      // フォルトアドレスがライブラリ情報(link_map)に保存しているロードアドレス情報の
      // どの部分にマッチするか検索
      …
  }
  if(!found){ // 見つからなかったら、ハンドラ登録を抹消して、再度フォルトを起こす
    SWI_ARG1(269, NULL);     /* clear handler */
  } else {
    if(l->l_lazy){ // まだライブラリロードされていなければロードする
      while(!compare_and_swap((long *)&(l->l_map_working), 0, 1))
              usleep(30000); // 汚いが、競合状態時に30ms待たせる
      if(l->l_lazy){  // まだライブラリロードされていなければロードする
              _dl_map_object_lazy(l,  GL(dl_locked_load_mode), 1); // ライブラリのロード
                                                                   // 内部でconflict処理も実施
              _dl_init_lazy(l);            // ライブラリの初期化ルーチン呼び出し
              l->l_lazy = 0;               /* load finished */
      }
      while(!compare_and_swap((long *)&(l->l_map_working), 1, 0)){
              usleep(30000);/* wait 30ms */
  } } }
  RETURN_TO_FAULTPROG((long)regs);         // フォルト時のレジスタに復帰する
}
```

フォルトの直前に復帰

オプション機能

- ライブラリ単位で遅延ローディングのOFF指定可能

  目的:

  （1）常にロードされるようなライブラリ(libc.so, libpthread.soなど)で余分なオーバヘッドを避ける

  （2）起動(main()) までに、必ずinitを呼ばなければならないようなライブラリについては、初期化できるようにする

  方式:

  /etc/ld.so.forbid_lazyload ファイルにライブラリパスを記述

  →dl-load.c内で比較し、対象外か判断

- 環境変数(“DL_LAZY_LOAD”)で遅延ローディングのON/OFF可能

  →デバッグ、評価のため

遅延ローディング(効果)

効果

- 35プロセスのプロセスが、40ライブラリずつリンクしていて、その6割がロード不要になったとすると

  35 x (40 x 0.6) x 4KB = 3.36MB

  →3.36MB以上の効果がある

  (通常1ライブラリ4KB以上使うから)

  →さらに、仮想空間が節約されるため、PTEキャッシュが節約された。(～数100KB)

35プロセス ： PC版Linuxでの通常プロセス数
40ライブラリ: Linuxアプリ(gnome関連など)で40前後のライブラリに依存
約60%　　： ライブラリ実使用率(1機器での実測)

遅延ローディング(検討)

- 他の実装の検討(実装時に検討した機能):

  ライブラリの初回アクセスをフックするほかの方法がないか？→現状見つかっていない

  - 既存 libcの類似機構
    - lazy_binding
      - シンボル解決を遅延する機能。起動性能には有効だがメモリ削減効果はない
    - filter :
      - ライブラリのシンボル情報を取り込む部分については利用可能であるが、今回の用途には使えない

今後の課題

- よい改善案を募集中
  - init/finiのcall順が保証されない
  - dlopen()するライブラリで効果なし
  - マルチスレッド下での、フォルト発生時の競合
  - 性能
    - 不用ライブラリをロードしないにもかかわらず、起動性能の向上には役立っていない:

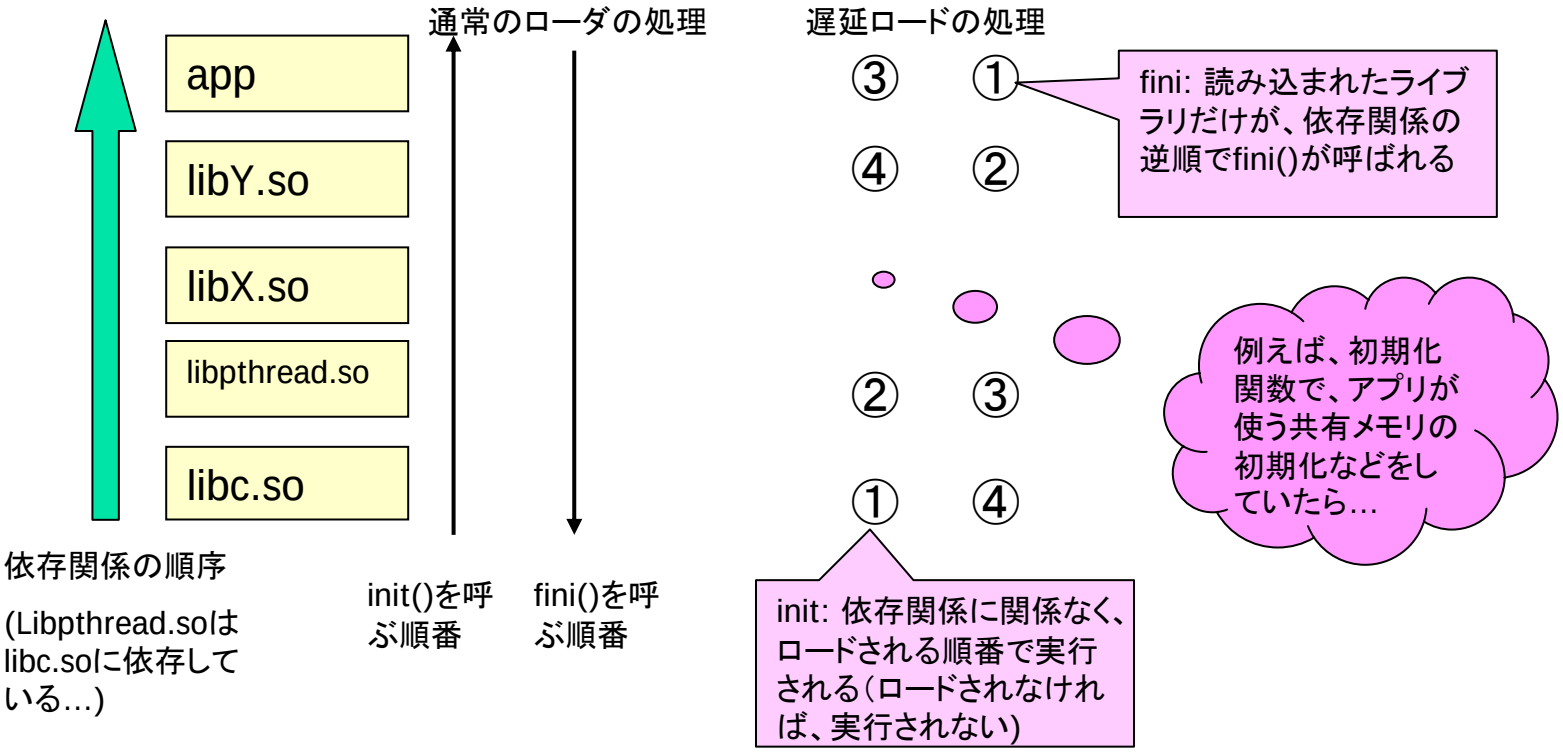
      起動時に

      mmap() → .dynamic等の読み取り →unmap()

      しているため(現状unmap()が遅い)

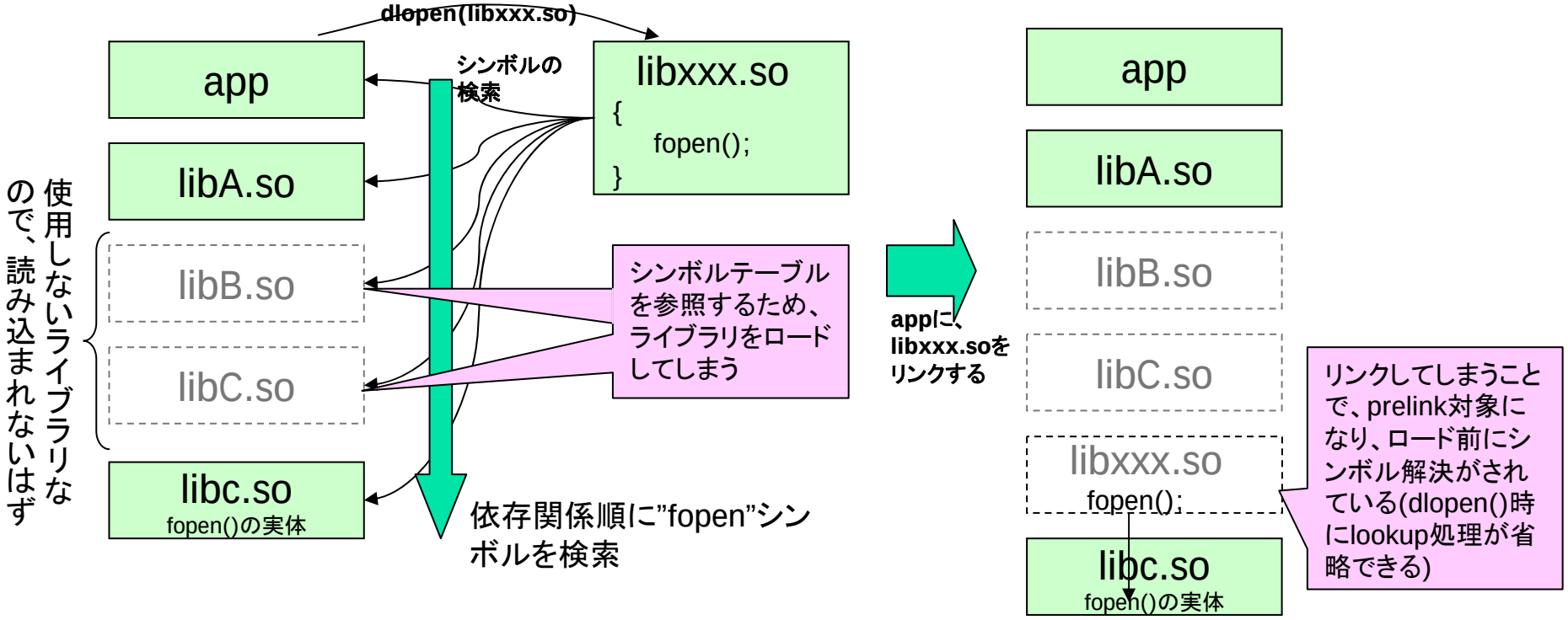
今後の課題(説明[1])

- init/finiの問題
  - 通常ライブラリは、依存関係の下流から初期化ルーチンを呼んでいき、終了ルーチンはその逆順で呼ぶ
  - 遅延ローディングの場合、ロードされる順番で呼ばれるため、順序が保証されない。(ロードしなければ、初期化ルーチンすら呼ばれない)
  - ただし、実際に問題になるケースはほとんどない(初期化順ではなく、初期化が必要なケースは、該当ライブラリを遅延ロード対象外にすれば回避可能)



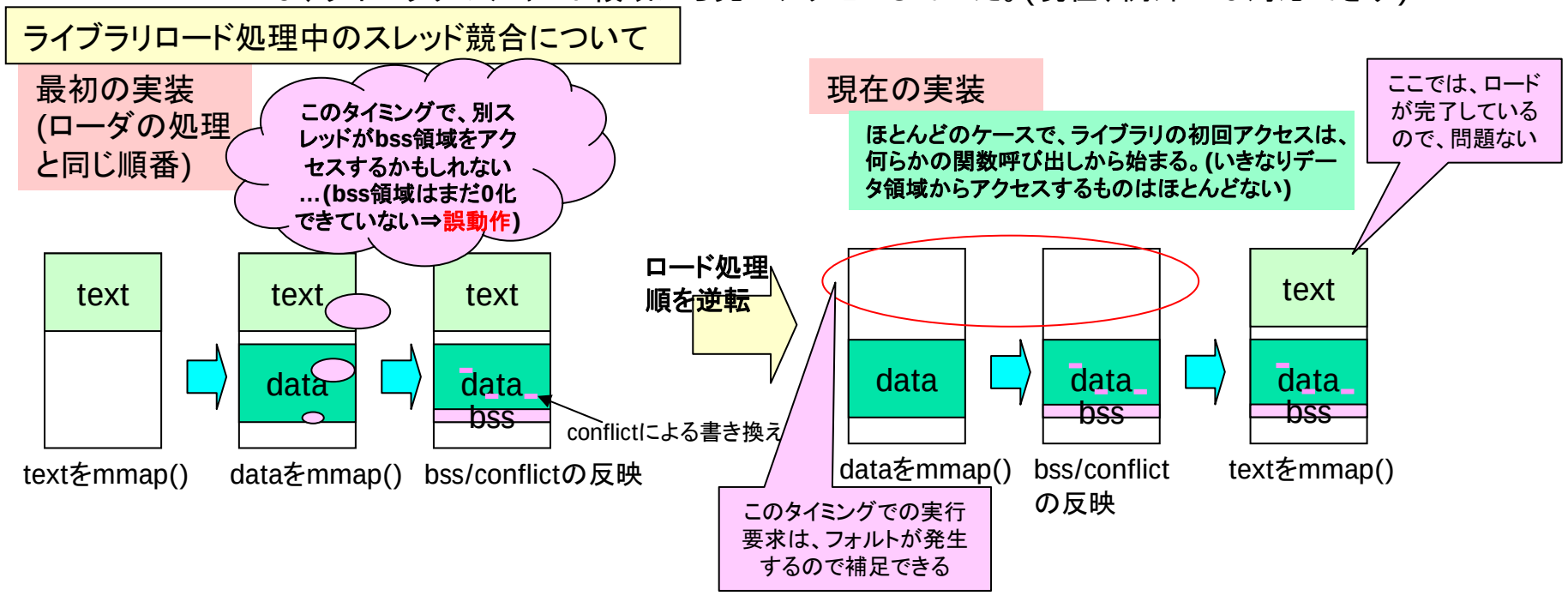
今後の課題(説明[2])

- dlopen()の問題
  - dlopen()するライブラリはロード時に使用するシンボルの検索(lookup処理)を行う。
  - この処理で、各ライブラリのシンボルテーブルを順に見ていくため、必要のないライブラリもアクセスされて、メモリに乗ってしまう。
  - [暫定回避策] dlopen()するライブラリをダイナミックリンクしてしまう。
    →prelink対象となり、dlopen()時にシンボル検索しない。
    →メモリについては、遅延ロードによりdlopen()までアクセスされないのでメモリ消費はしない。(一部管理メモリ(link_info 約500B/ライブラリ)は消費するが…)



今後の課題(説明[3])

- マルチスレッド下でのライブラリロード時の競合
  - マルチスレッドで同時にフォルト発生した場合の競合については対策実施
  - ロード途中で競合状態になった場合に問題発生。
    - ライブラリロード時(一部のメモリをmmap()している途中)でタスクスイッチし、別のスレッドがそのメモリをアクセスしまった場合。(メモリはまだconflict処理などされていないため、正しくないかもしれない)
  - [暫定回避策]
    - ロード処理順を修正： データ領域のmmap() →conflict処理 →テキスト領域のmmap()
    - ライブラリのアクセス動作を観測したところ、通常、新しいライブラリをアクセスする場合には、ライブラリのテキスト領域から先にアクセスしていた。(現在、例外には対応できず)



ご静聴ありがとうございました。