Reducing Memory Usage at Shard Library Use on Embedded Devices

2007.02.22

Tetsuji Yamamoto,
Matsushita Electric Industrial Co., Ltd.
Masashige Mizuyama,
Panasonic Mobile Communications Co., Ltd.

[translated by Takao Ikoma]

Table of Contents

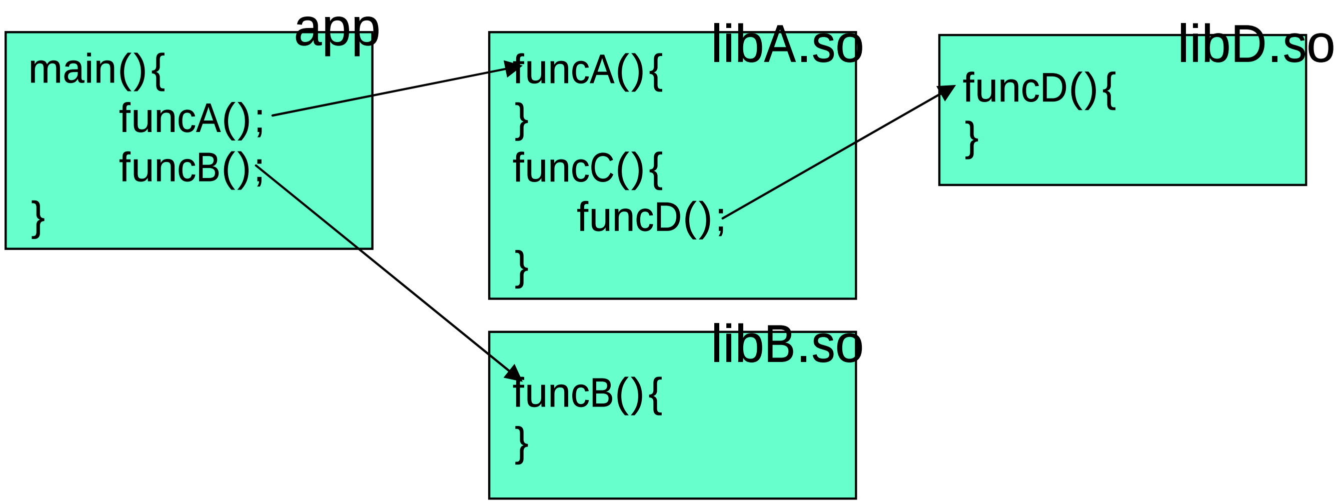
- Background
- Analysis
- Method discussed
- Lazy Loading

  (overview, idea, implementation)
- Results
- Issues

Background(1)

- In embedded systems such as cellular phones, features of applications have grown up, so more dynamic libraries are getting linked (in some cases, dozens of libraries are linked)
- dependency among dynamic libraries gets complicated, unnecessary library for the application, or library required only for a specific feature, must be linked

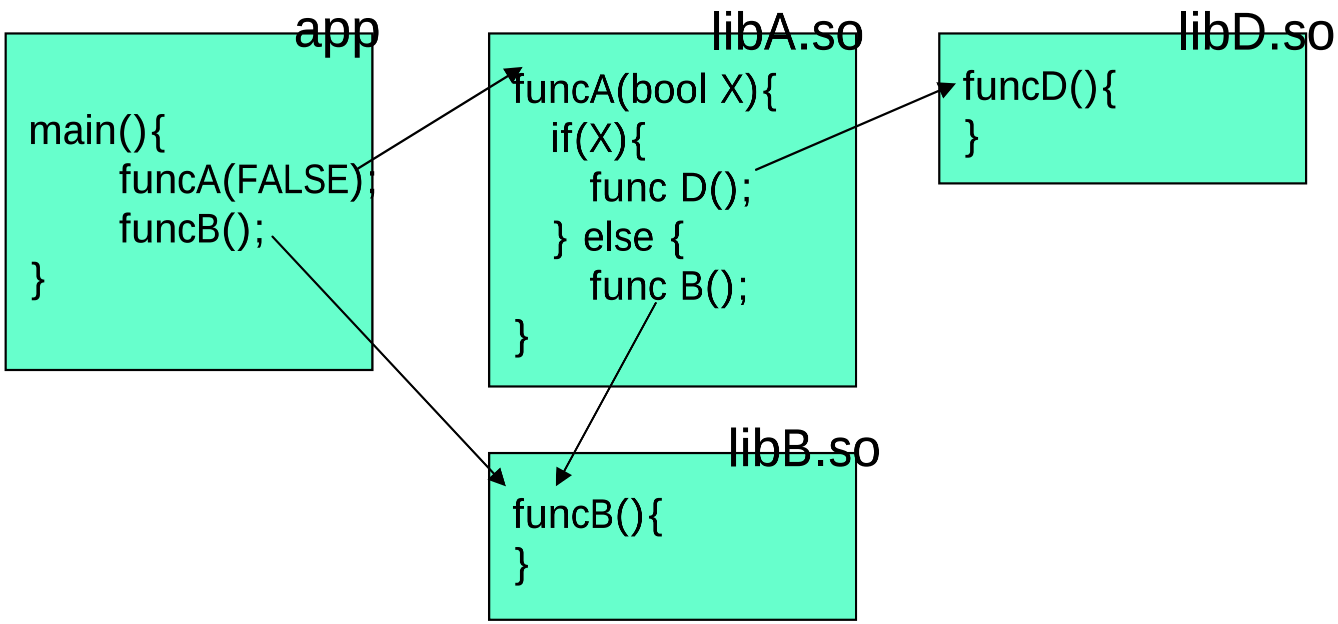
CASE1



For the dependency shown above, lib A, B and D should be linked; but libD.so is not actually used, thus the memory for the library is wasted.

Background(2)

CASE2:



app

```
main(){
        funcA(FALSE);
        funcB();
}
```

libA.so

```
funcA(bool X){
   if(X){
        func D();
   } else {
        func B();
}
```

libD.so

```
funcD(){
}
```

libB.so

```
funcB(){
}
```

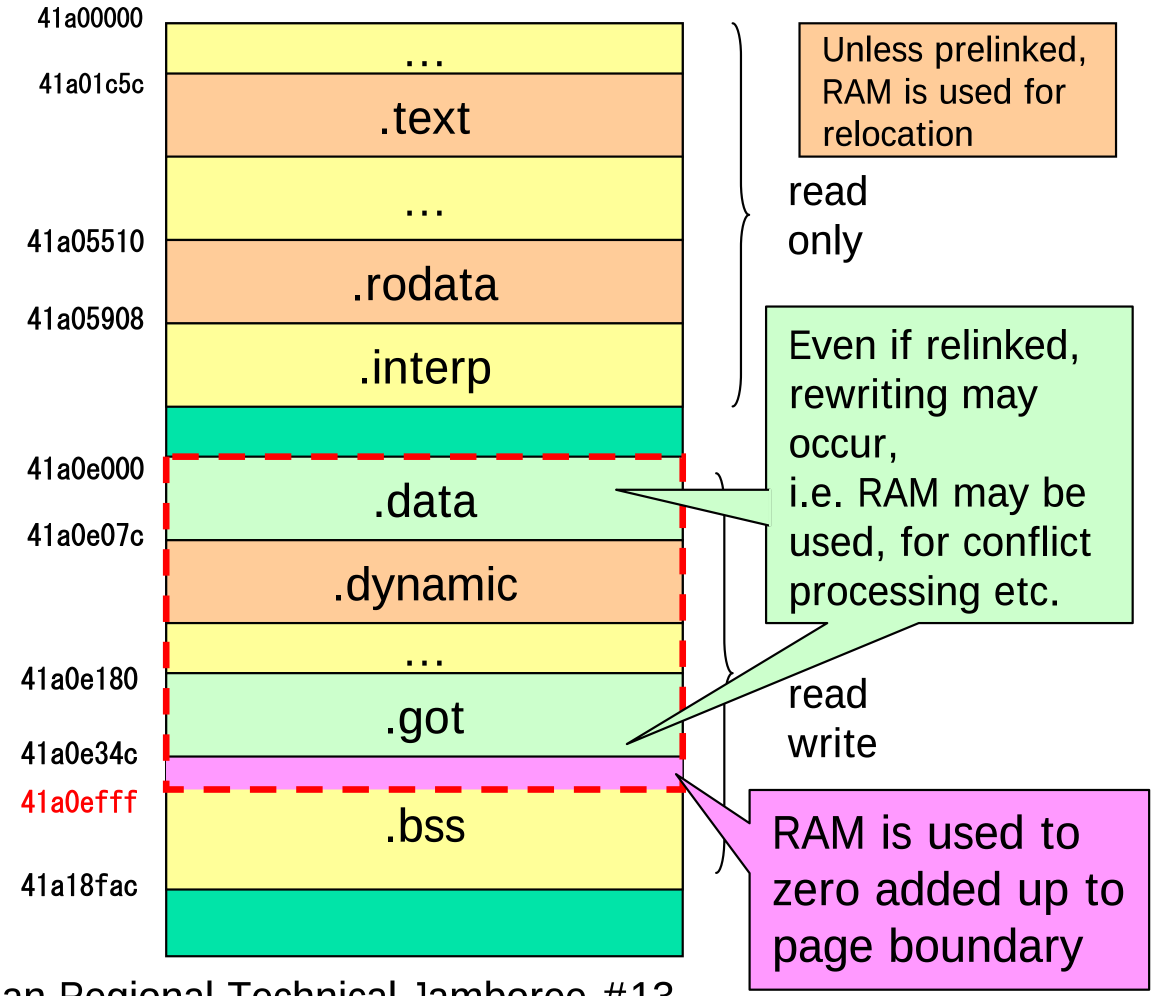
For the structure shown above, funcD() is never called at runtime, but should be linked.
→ To handle this with dynamic loading (dlopen(), dlsym()), both the application and the libraries should be restructured, which would cost a lot

- Would like to save this memory overhead for which unused library is loaded.

Analysis(1)

- How much of memory are used just by loading libraries?
  - When linking dynamic libraries, (even if not used) RAM of more than one page/library is consumed (If the boundary between data region and bss region is not on a page boundary, zeros are padded for bss initialization (one page used)
  - RAM is also used by rewriting .data/.got due to conflict

```
Section 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
```



Unless prelinked, RAM is used for relocation

Even if relinked, rewriting may occur, i.e. RAM may be used, for conflict processing etc.

RAM is used to zero added up to page boundary

Analysis(2)

- Initialization part of .bss (ld.so (elf/dl-load.c))

_dl_map_object_from_fd() 内

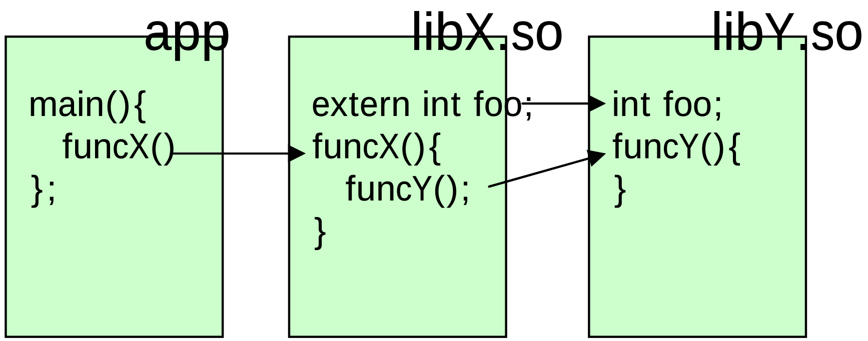
```
  …
zero = l->l_addr + c->dataend;
zeroend = l->l_addr + c->allocend;
zeropage = ((zero + GL(dl_pagesize) - 1) & ~(GL(dl_pagesize) - 1));
     …
if (zeropage > zero)
  {
        /* Zero the final part of the last page of the segment.  */
        if ((c->prot & PROT_WRITE) == 0)
          {
            /* 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)
  …
```

c.f. On conflict in prelink

Cause of conflict occurence

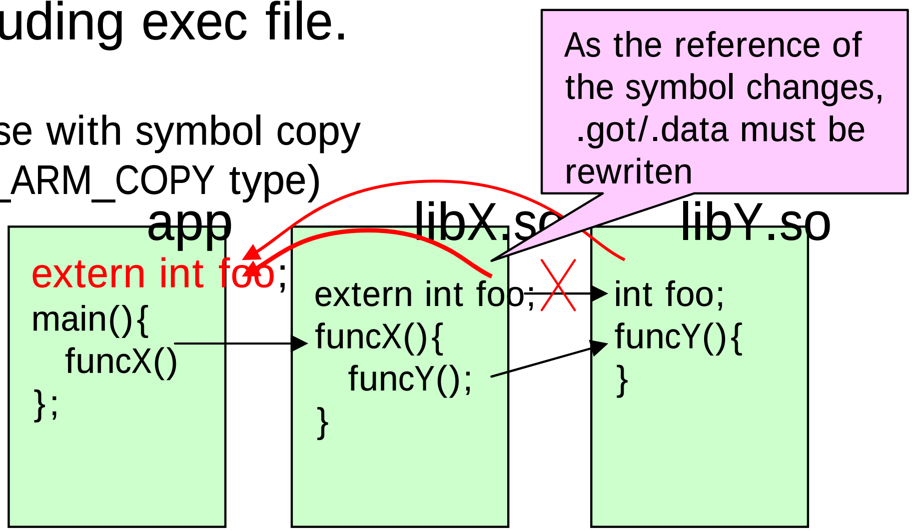
- Symbol->address mapping resolved with prelink may not be same for the case it is resolved within libraries for which the library depends on, and for the case resolved including exec file.

Usual Case



app depends on libX.so, and libX.so depends on libY.so
→no conflict in this case

Case with symbol copy
(R_ARM_COPY type)



For symbol referred in execution file, its entity is copied to execution file side, so reference in libraries should be modified (Current ARM compiler does not support –znocopyreloc option, so behavior above can not be suppressed)
→ Modify reference of foo in libX.so to app side (conflict)

Further, conflict information is generated when library dependency is insufficient at link time, or when symbol is doubly defined.
Addresses whose symbols were resolved with prelink exist in .got/.data sections. These sections may be modified.

Methods Studied

What can we do to load required libraries only when they work?

- Plan1

  Design with total consistency considering which libraries to use

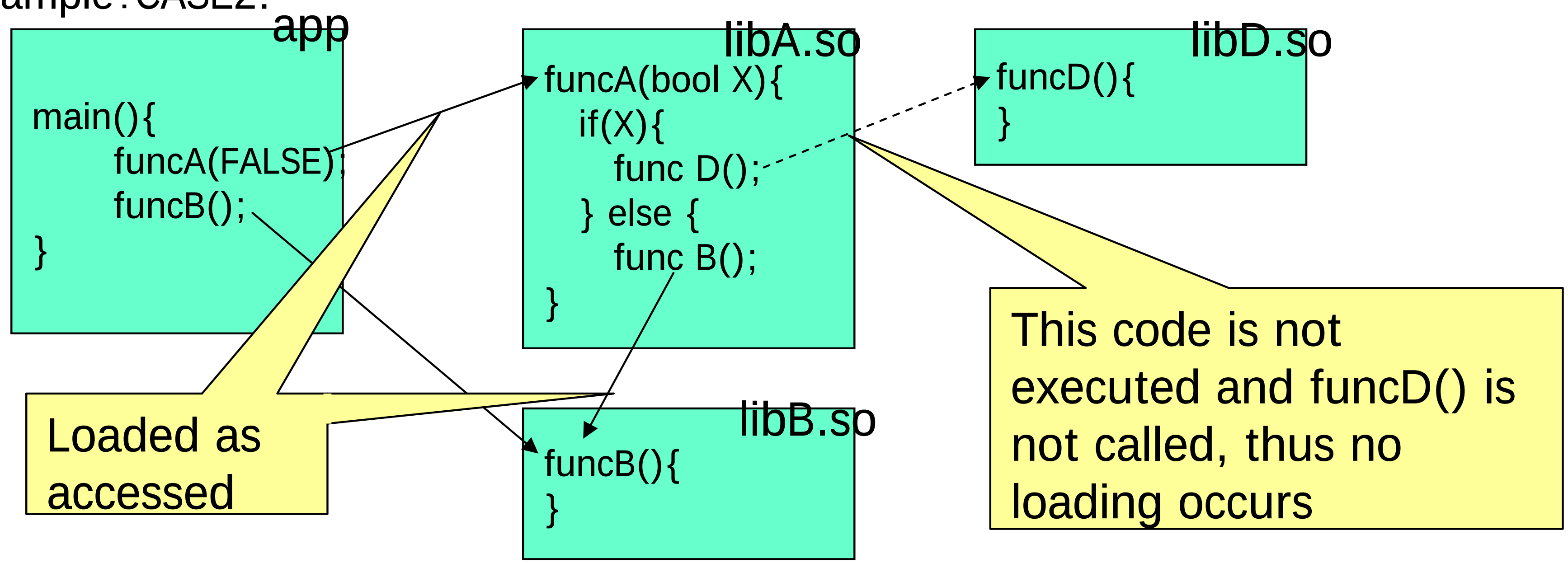
  → Straightforward approach, but cost to manage several handreds libraries is prohibitive
- Plan2

  Implement with dynamic library loading (dlopen())

  → All of application and libraries implemented have to be fixed.

  →Prelink is not applicable, and overhead of dlopen() (symbol resolution processing) is large
- Plan3
  - Make loader/OS to automatically load libraries when required (when libraries are executed/accessed) (lazyload)

Example：CASE2:



app

```
main(){
        funcA(FALSE);
        funcB();
}
```

libA.so

```
funcA(bool X){
   if(X){
      func D();
   } else {
      func B();
}
```

libD.so

```
funcD(){
}
```

libB.so

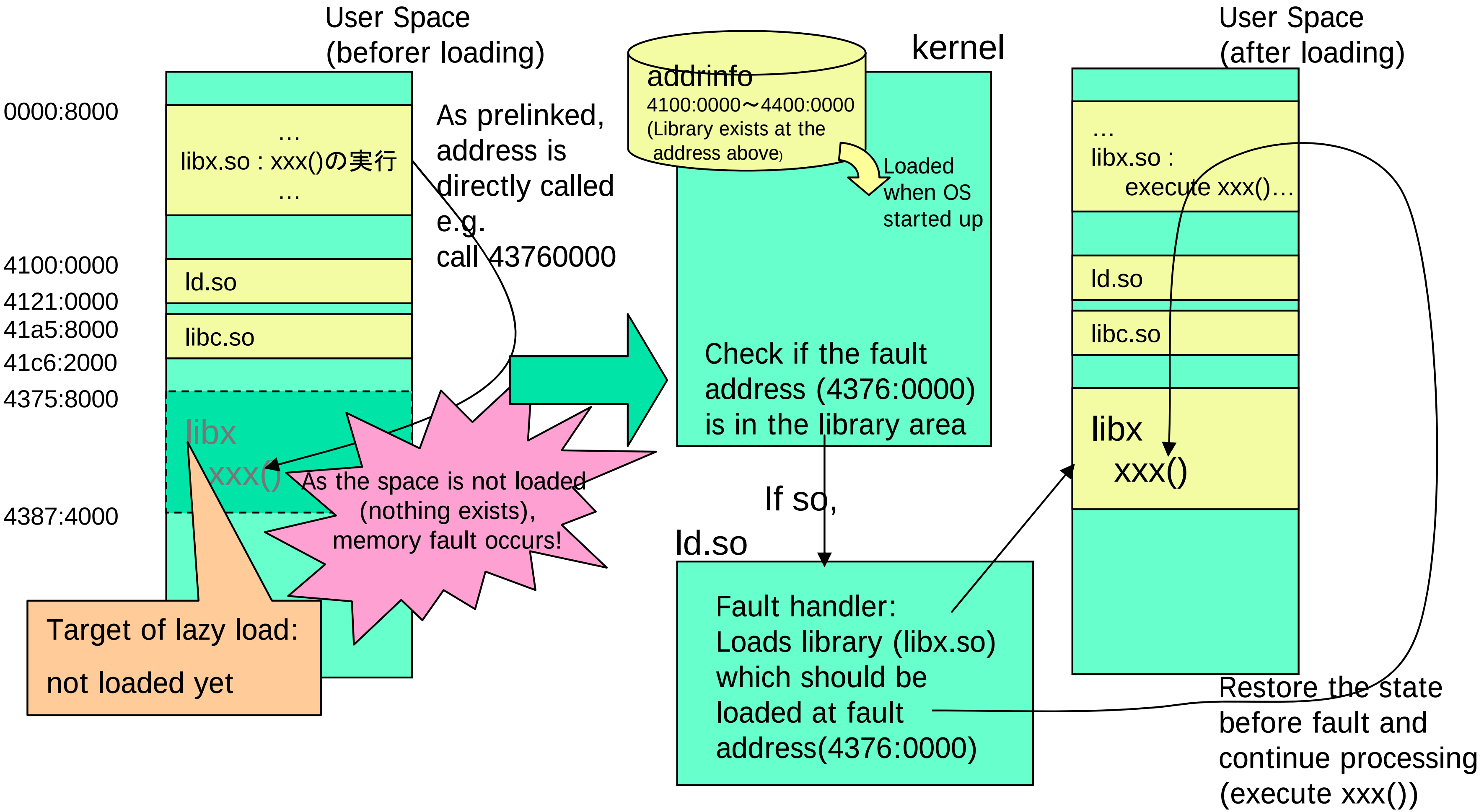
```
funcB(){
}
```

Loaded as accessed

This code is not executed and funcD() is not called, thus no loading occurs

Lazy Loading (Mechanism)

- Using memory fault to invoke handler of loader,lazy load of library is implemented



User Space
(beforer loading)

0000:8000

…
libx.so : xxx()の実行
…

4100:0000
ld.so

4121:0000

41a5:8000
libc.so

41c6:2000

4375:8000
libx
xxx()

4387:4000

As prelinked,
address is
directly called
e.g.
call 43760000

As the space is not loaded
(nothing exists),
memory fault occurs!

Target of lazy load:
not loaded yet

addrinfo
4100:0000～4400:0000
(Library exists at the
address above)

kernel

Loaded
when OS
started up

Check if the fault
address (4376:0000)
is in the library area

If so,

ld.so

Fault handler:
Loads library (libx.so)
which should be
loaded at fault
address(4376:0000)

User Space
(after loading)

…
libx.so :
execute xxx()…

ld.so

libc.so

libx
xxx()

Restore the state
before fault and
continue processing
(execute xxx())

LazyLoading (Overview)

- overview
  - Environment
    - Based on MontaVista CEE3.1(kernel 2.4.20, glibc 2.3.2) , modified Linux kernel、glibc(ld.so).
  - Premise
    - Must be prelinked; load addresses fixed and bind processing done.
      →Because fault is judged with load address
    - →If address resolution (BIND) has not been done, other libraries are loaded for symbol search, and memory saving effect would be disappear
  - Overview of behavior

(1) Kernel loads library address information (at kernel bootup)

(2) Behavior at process invocation

Judge if lazy loading on
↓
Register fault handler to kernel
↓
mmap library as READ ONLY
↓
Build information (struct link_info) in loader
↓
unmap library
↓
To main()

(3) When a library is accessed after main() started, fault (segmentation fault) occurs and control passed to ld.so handler → Judging from the fault address, the designated library is loaded and return.

Lazy Loading (Change Log)

- Major Placed Changed
  - Linux Kernel
    - arch/arm/kernel/call.S : Add system call
      (for fault handler registration, obtaining register info at fault)
    - arch/arm/kernel/sys_arm.c : Replaced return PC address at fault
    - arch/arm/kernel/dlfault.c(new) : Handler code for fault
    - arch/arm/mm/fault-common.c : Branch at memory fault
    - init/main.c : Reading library address information
  - glibc (ld.so)
    - elf/rtld.c : Judging lazy loading, ON/OFF, fault handler, etc
    - elf/dl-load.c : Saving and loading load information for lazy loading
    - elf/dl-init.c : Initialization of library for lazy loading
    - elf/conflict.c : Conflict processing for lazy loading
    - include/link.h : Added variables for lazy loading
      (load management, addr info)
    - sysdeps/genelic/ldsodefs.h : Added variables for lazy loading (ON/OFF)
- Patches will be published on CELF web

Lazy Loading (Source Code: excerpts)

- Jump from memory fault to handler with process below

Fault handler

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)){  //  search if in the library area
                        dl_fault_savereg(tsk,regs,addr);  // save register info
                        dl_fault_setpc(tsk,regs);  // rewrite return address to
                                                   // loader handler
                }
                else{
                   __do_user_fault(tsk,addr,error_code,SEGV_MAPERR,regs);
                }
        }
        else
                __do_kernel_fault(mm,addr,error_code,regs);
}
```

Lazy Loading (Source Code: excerpts)

- Kernel -> Load Handler -> Return with process below

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);              // Get register info

  /* search maps */
  for(;l;l = l->l_next){
      // Search which part of the load address info (link_map)
      // the fault address matches
      ...
  }
  if(!found){ //  If not found, delete hadler registration and reinvoke fault
    SWI_ARG1(269, NULL);      /* clear handler */
  } else {
    if(l->l_lazy){ // Load if library has not been loaded yet
      while(!compare_and_swap((long *)&(l->l_map_working), 0, 1))
              usleep(30000); // Ugly; 30ms wait for race condition
      if(l->l_lazy){          // Load if the library has not been loaded
              _dl_map_object_lazy(l, GL(dl_locked_load_mode), 1); // Load the library
                                       // Do conflict processing within function
              _dl_init_lazy(l);        // Call initialization of the library
              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);         // Restore the registers at fault
}
```

Return to just before fault

Optional Features

- Can disable lazy loading per library

  Objective:

  (1) To avoid overhead for libraries which are always loaded
  (libc.so, libpthread.so etc.)

  (2) To make it possible to initialize for libraries which must always call init before invocation (main())

  Methods:

  Specify library path in file /etc/ld.so.forbid_lazyload
  → compare in dl-load.c and judge if exception or not

- Can set ON/OFF of lazy loading with environment variable ("DL_LAZY_LOAD")

  → For debug, evaluation

Lazy Loading (Results)

- Results
  - Asssuming that each of 35 processes links to 40 libraries,and that 60% of them need not be loaded any more,

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

    → more than 3.36MB would be reduced

    (as an ordinary library consumes more than 4KB)

    →Further, due to less virtual space required, PTE cache is saved(up to several hundred kilobytes)
  - 35 processes: common number of processes on PC Linux
  - 40 libraries: Linux application (such that gnome related one) depends on around 40 libraries
  - 60%: Actual library use rate (actually measured on a single device)

Lazy Loading (Discussion)

- Cosideration of other implementation (features studied at implementation) Any other method to hook first access of library?

  → Not found so far

  Similar mechanism on existing libc

  - - lazy_binding
    - -- Function to defer symbol resolution. Effective for start up performance but no effect to save memory
  - - filter
    - -- Usable for the part to incorporate symbol information of library, but no use for our purpose

Issues

- Improvement proposal welcome
  - Call sequence of init/fini not garranteed
  - No effect for libraries which dlopen()
  - Race condition at fault under multithread
  - Performance
    - While suppressing to load unused librares, no improvement of start up time observed:

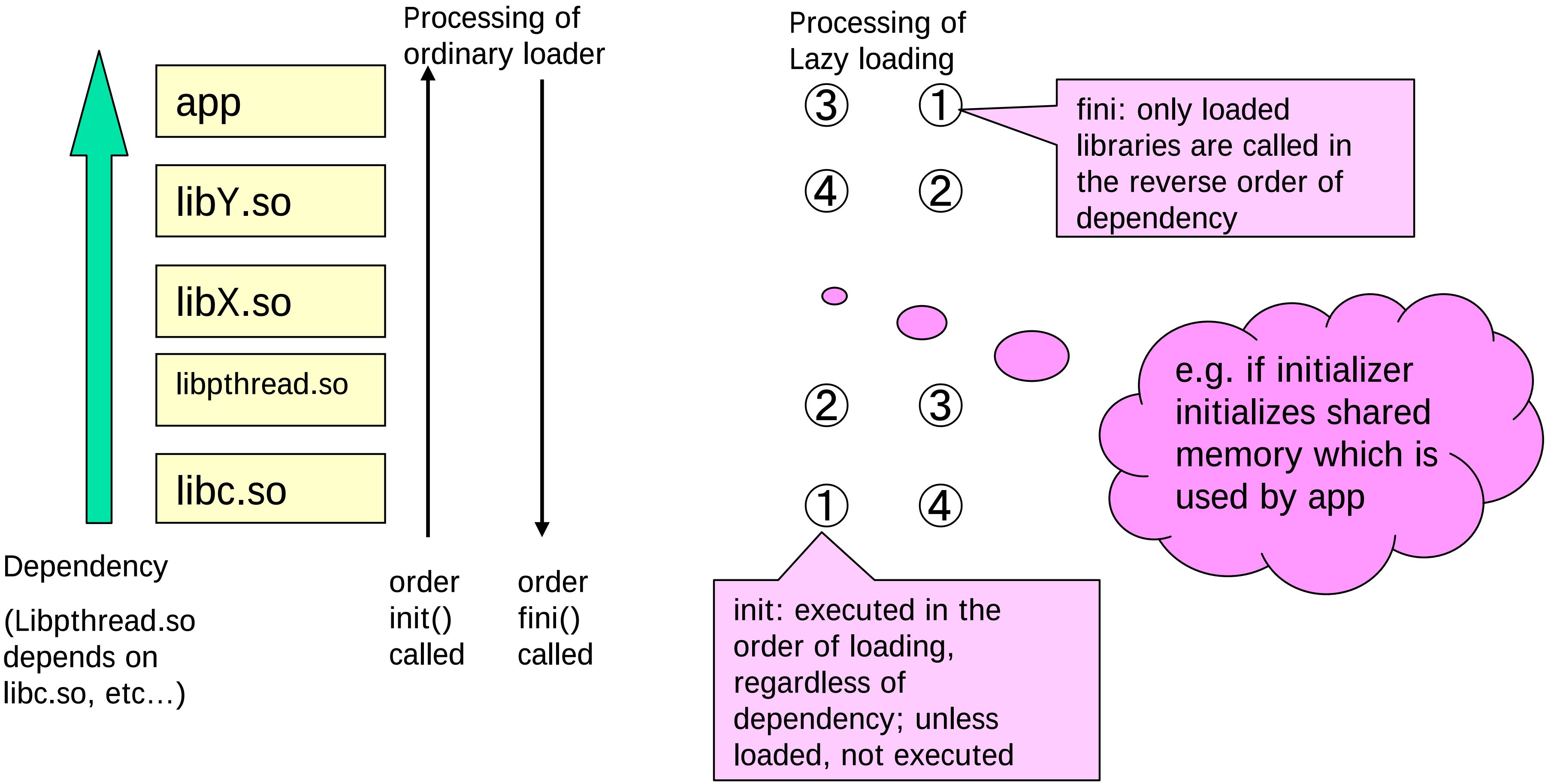
      because of

      mmap() -> read .dynamic etc -> unmap()

      at start up (current unmap() is slow)

Issues (detail (1))

- Issue of init/fini
  - For ordinary libraries, initializers are called from the bottom of dependency, and finalizers are called in the reverse order
  - In the case of lazy loading, initializers are called when loaded, so the order is not warranteed (If not loaded, its initializer is called at all).
  - But in many cases, there will not actually be any problem (In case that initialization, not the order of the initialization, matters, the problem can be avoided by excluding the library from lazy loading)



fini: only loaded libraries are called in the reverse order of dependency

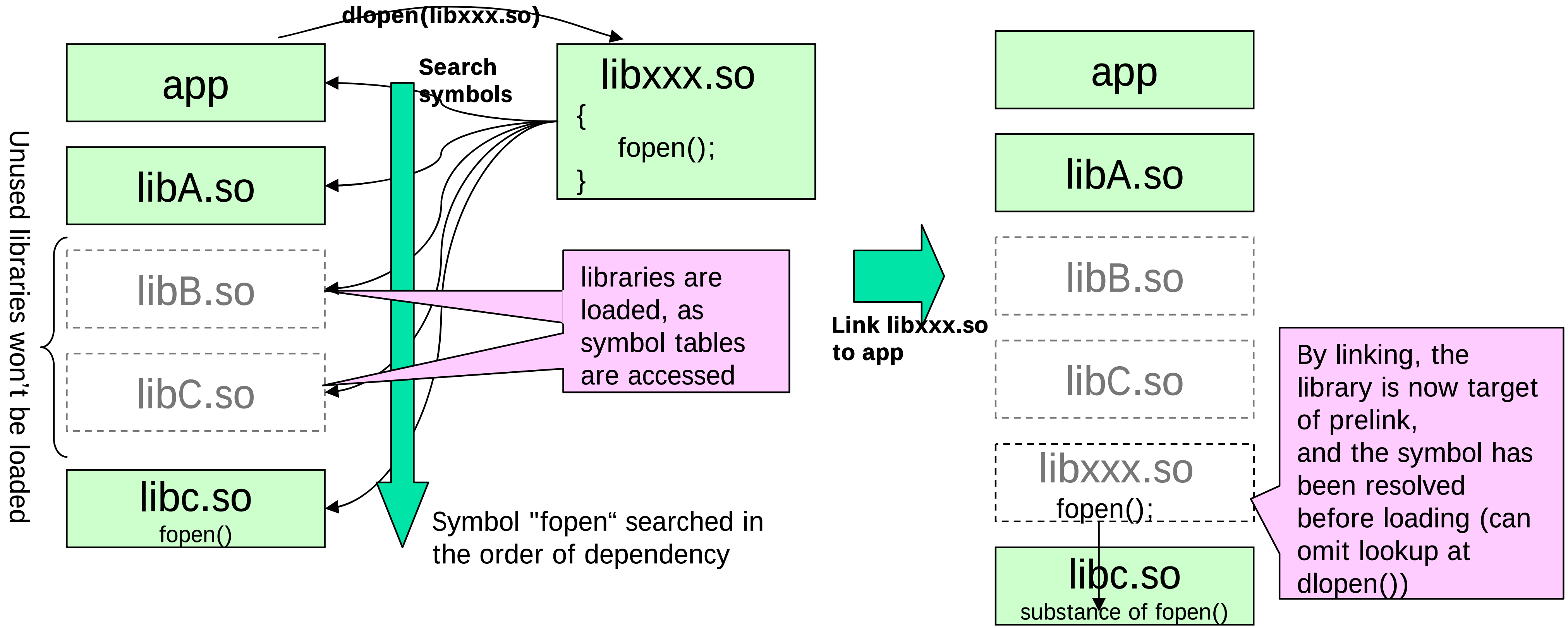
init: executed in the order of loading, regardless of dependency; unless loaded, not executed

e.g. if initializer initializes shared memory which is used by app

Issues (detail (2))

Issue of dlopen()

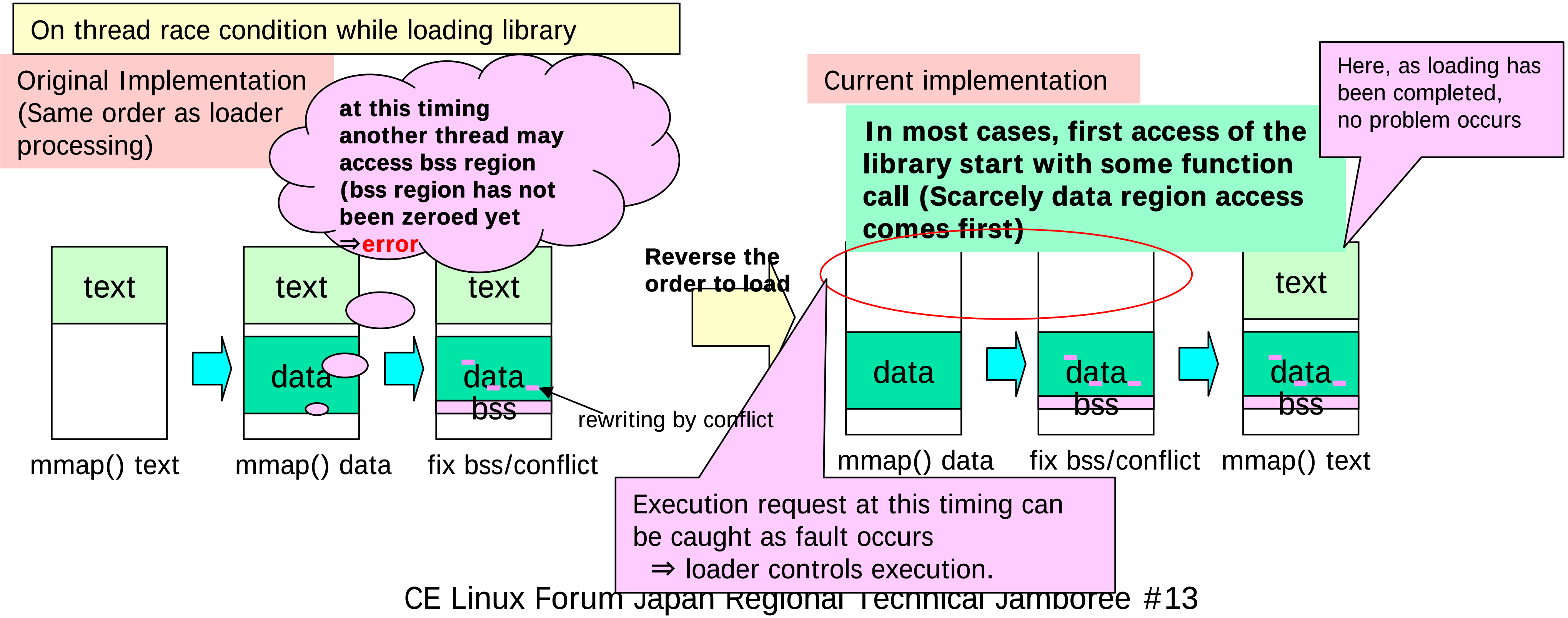
- Library to do dlopen() searches symbols to be used, at loading (lookup)
- At this processing, as symbol table of each library is looked up, even unnecessary libraries are accessed and loaded on memory
- (A tentative measure:) dyanmic link of the library to dlopen()
  - x
  - memory is not consumed (although some memory for management (link_info; about 500B/library) is consumed).



Issues (detail (3))

Race condition of library loading under multithread

- Race condition for simultaneous faults under multiple threads has already been considered
- The problem occurs in case of race condition while loading
  - In such a case that task switches at library loading (while mmpap()'ing some of memory) and that another thread accesses that memory (e.g. conflict processing has not been done yet and it may not be correct)
- [A tentative measure:]
  - Fixed the order of load processing:
    mmap() data region → conflict processing → mmap() text region
    (Observing access behavior of library, text region turned out to be accessed first when accessing a new library())



On thread race condition while loading library

Original Implementation (Same order as loader processing)

at this timing another thread may access bss region (bss region has not been zeroed yet ⇒error

Current implementation

In most cases, first access of the library start with some function call (Scarcely data region access comes first)

Here, as loading has been completed, no problem occurs

Execution request at this timing can be caught as fault occurs
⇒ loader controls execution.

Thanks you

Special thanks to Takao Ikoma.(Translation Jp->En)