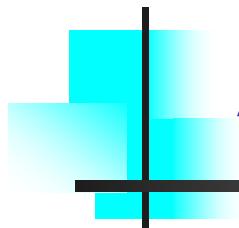


Deferred Dynamic Loading

A Memory Reduction Technique



2007.04.17
Tetsuji Yamamoto,
Matsushita Electric Industrial Co., Ltd.
Masashige Mizuyama,
Panasonic Mobile Communications Co., Ltd.

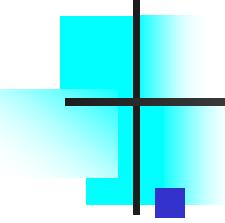


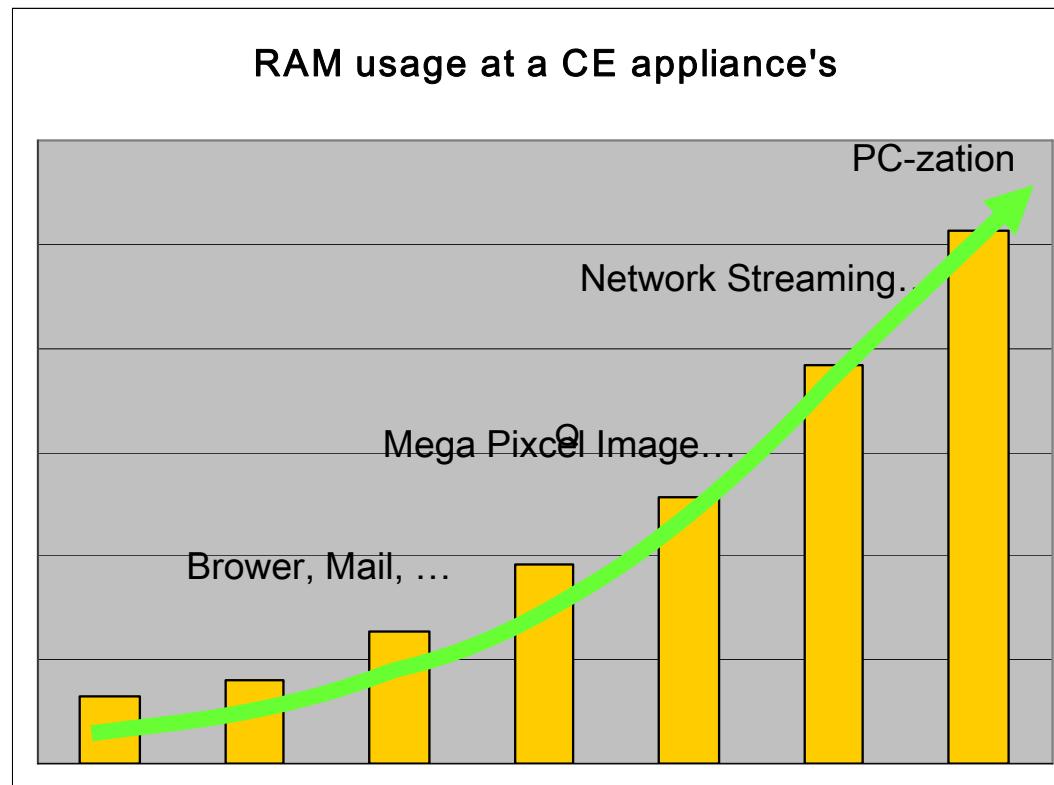
Table of Contents

■ Background

- Approach
- Deferred Loading
 - (overview, implementation, issues)
- Effectiveness

Background: CE products get fat

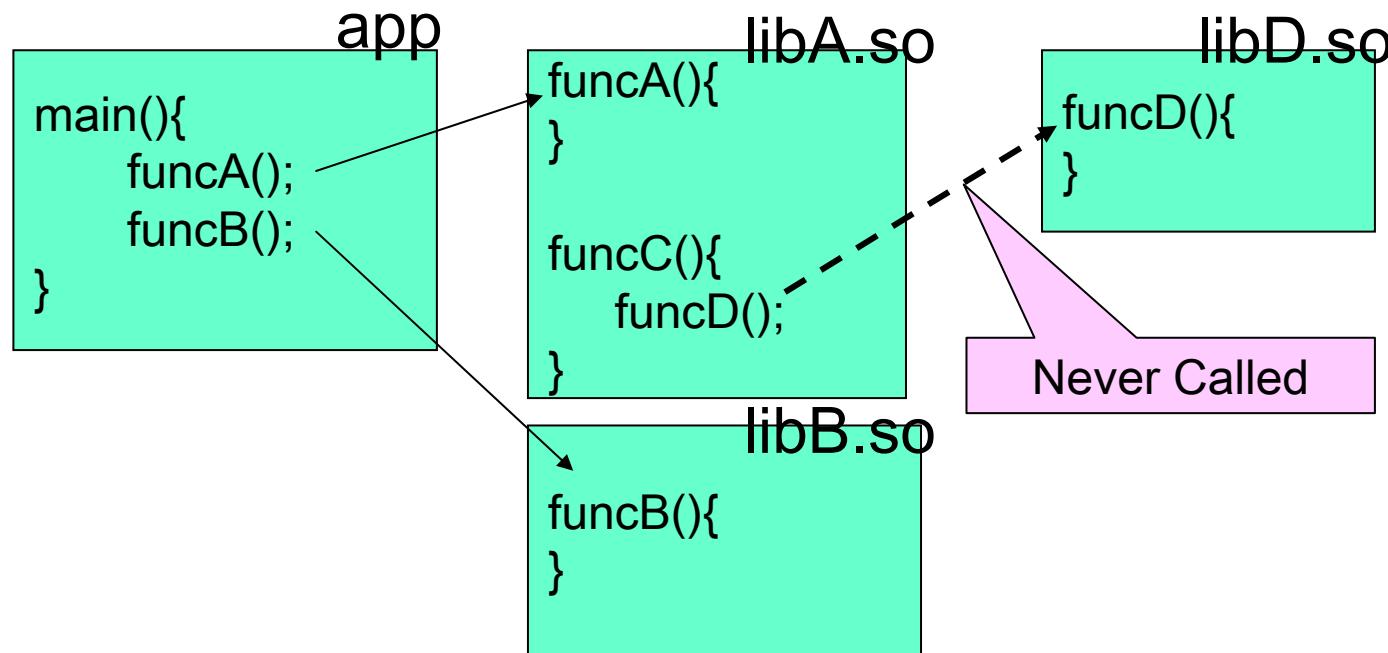
Reducing memory consumption gets more and more important for cost and power consumption.



Background: Using dynamic lib Case1

- libD.so is loaded but never used.
 - Despite of demand paging, some pages are wasted by libD.so.

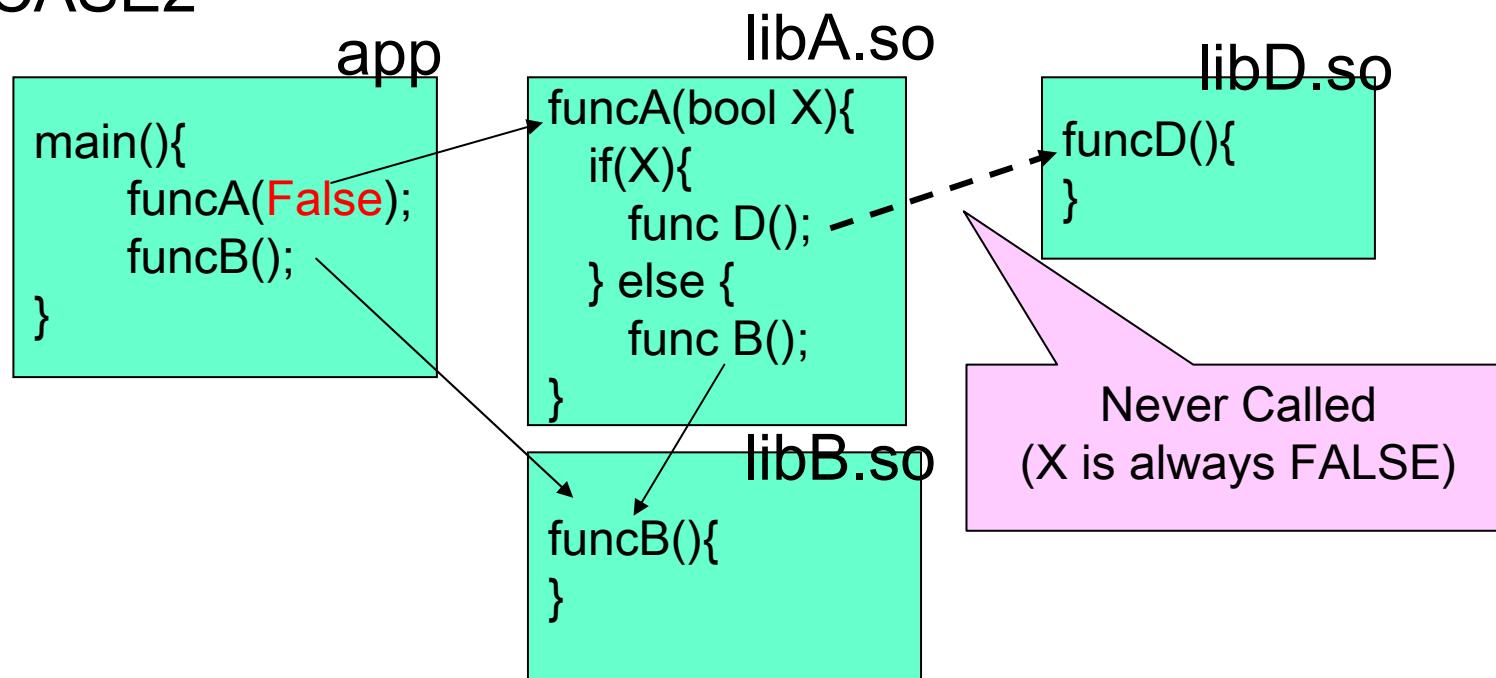
CASE1



Background: Using dynamic lib Case2

- libD.so also wastes memory.

CASE2



Why is RAM used for libD.so?

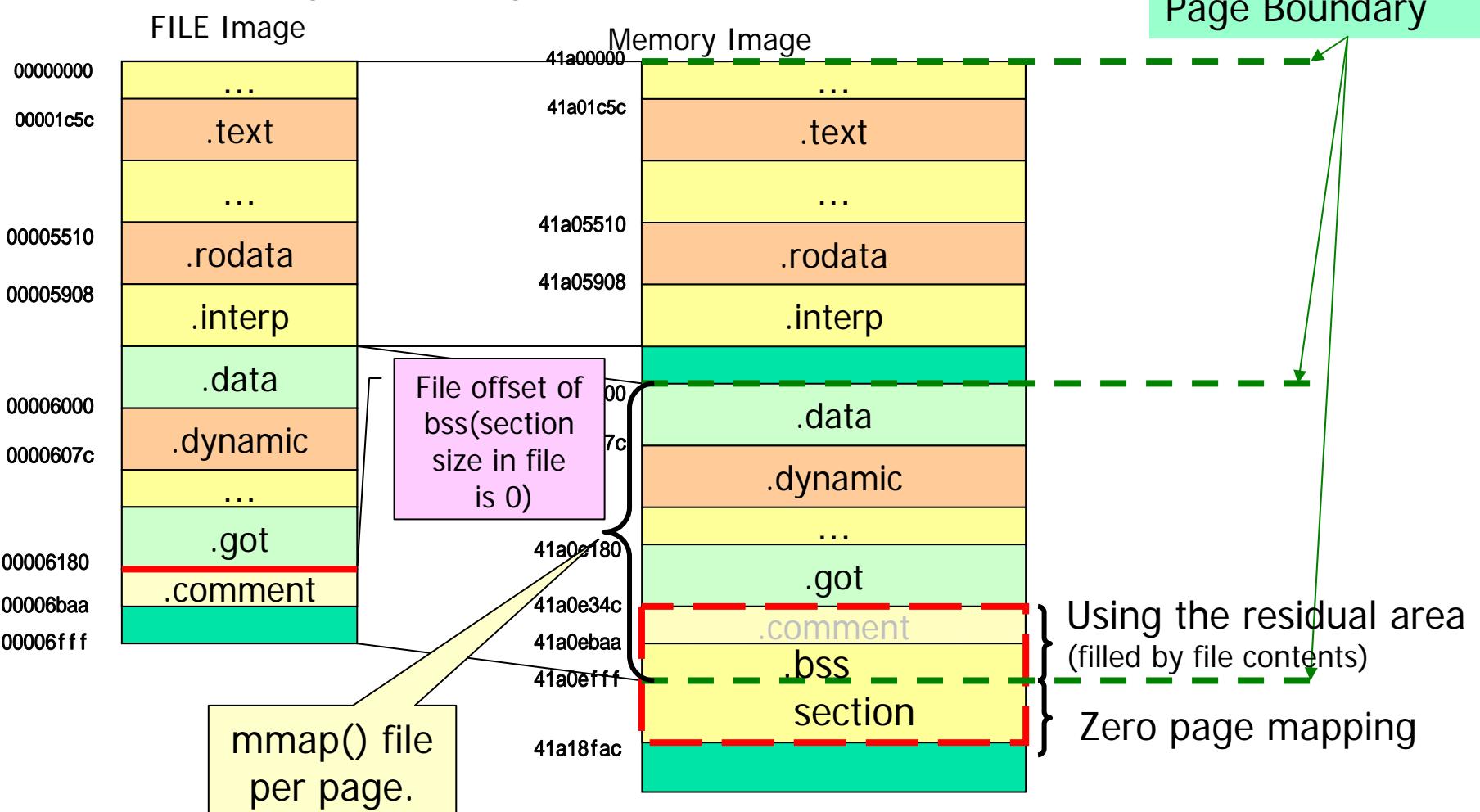
Some pages are used at loading time
(before main()) for:

1. Padding the .bss section with zero
2. Resolving conflict of symbols (after *prelinking*)

Part of .bss resides with other sections.

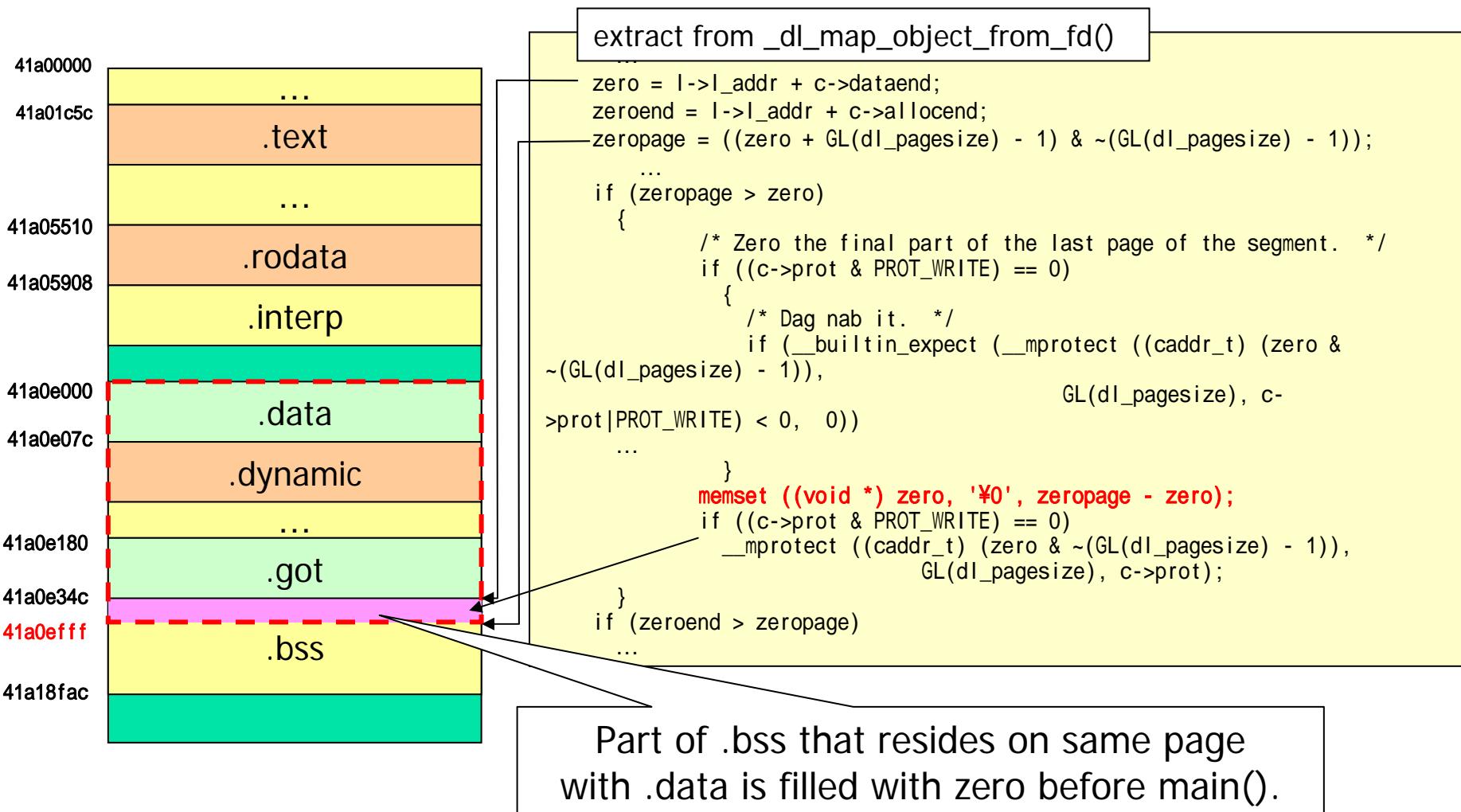
.bss section is divided into 2 parts by the page boundary.

1. Using the residual of other sections of page for saving memory.
2. Zero page mapping.



Padding 0 in .bss

Code for .bss initialization (ld.so (elf/dl-load.c))

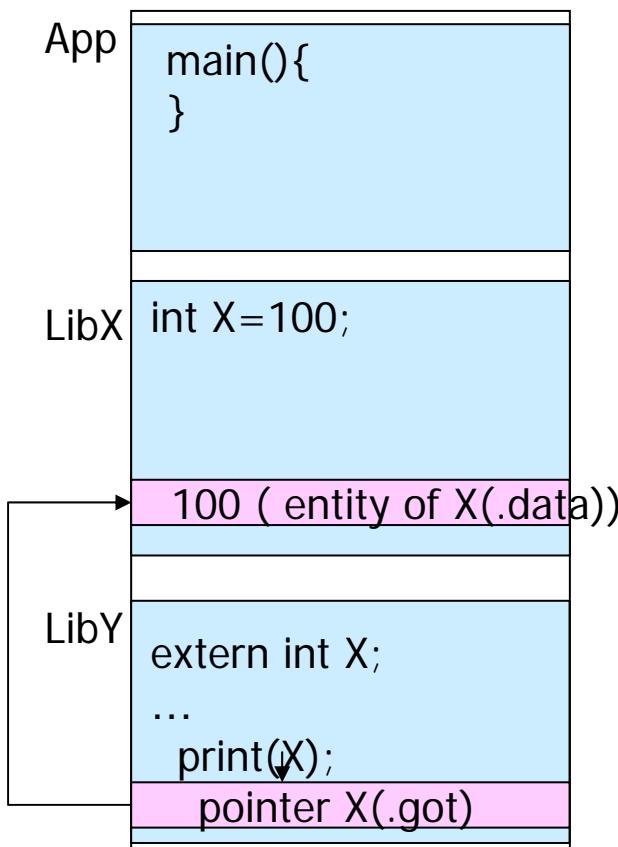


Why conflict occurs

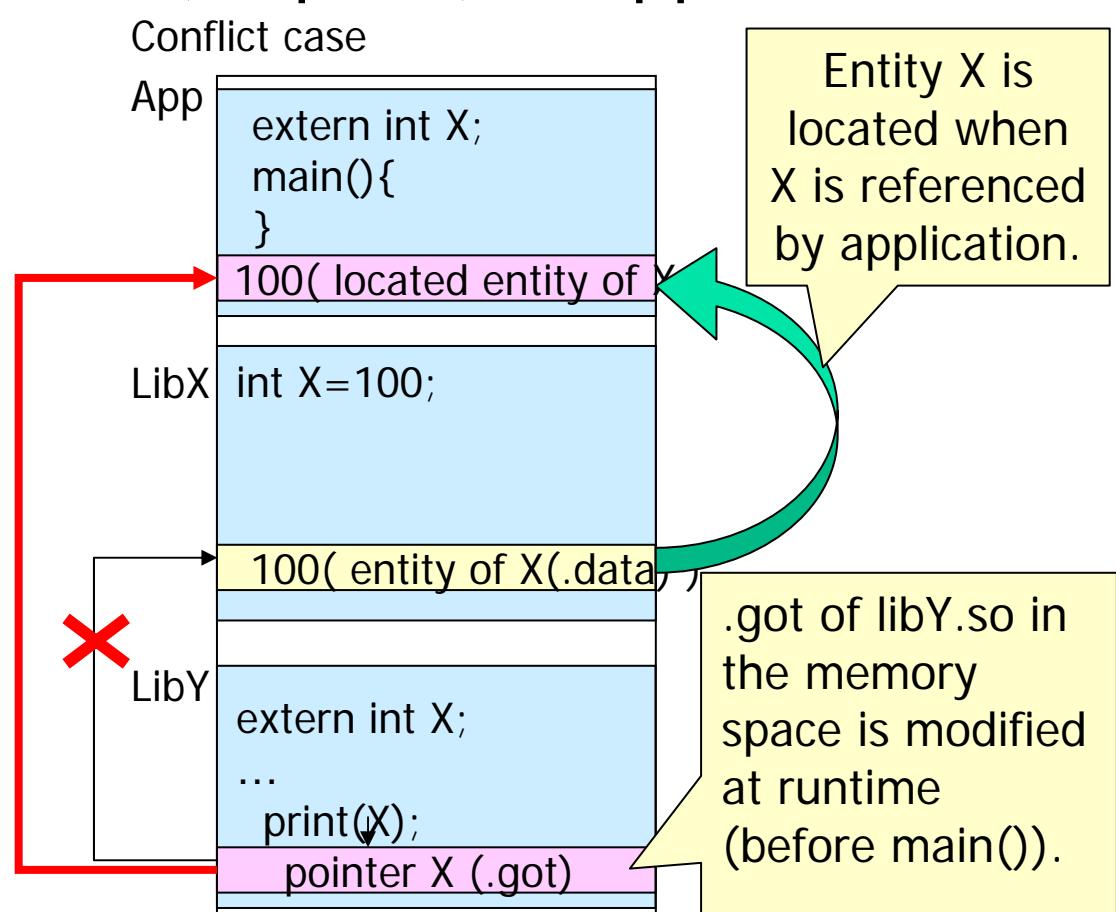
An example case:

- When application references library's variable, the variable entity is located ("copied") in application side.

Normal case

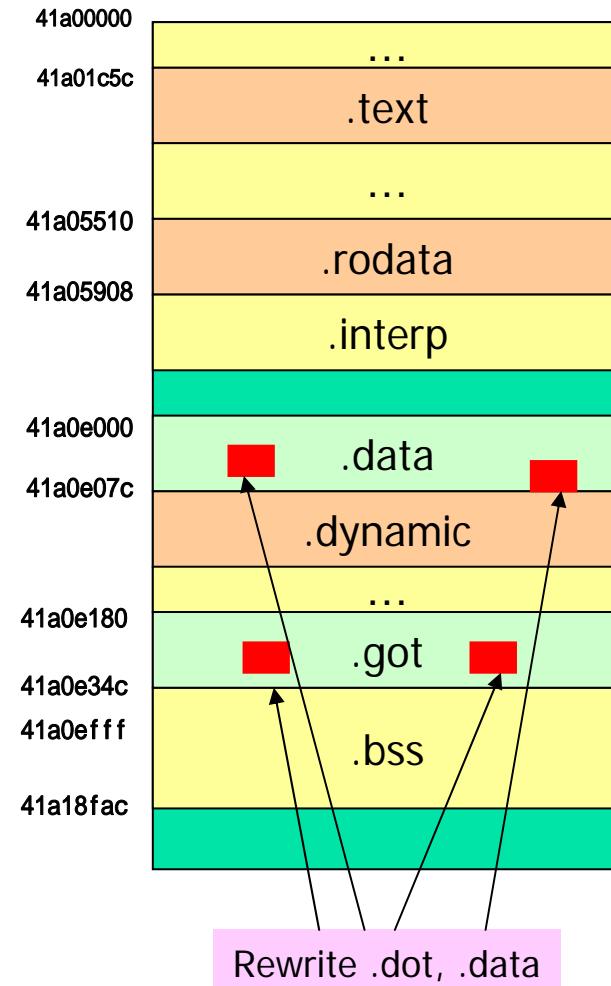


Conflict case



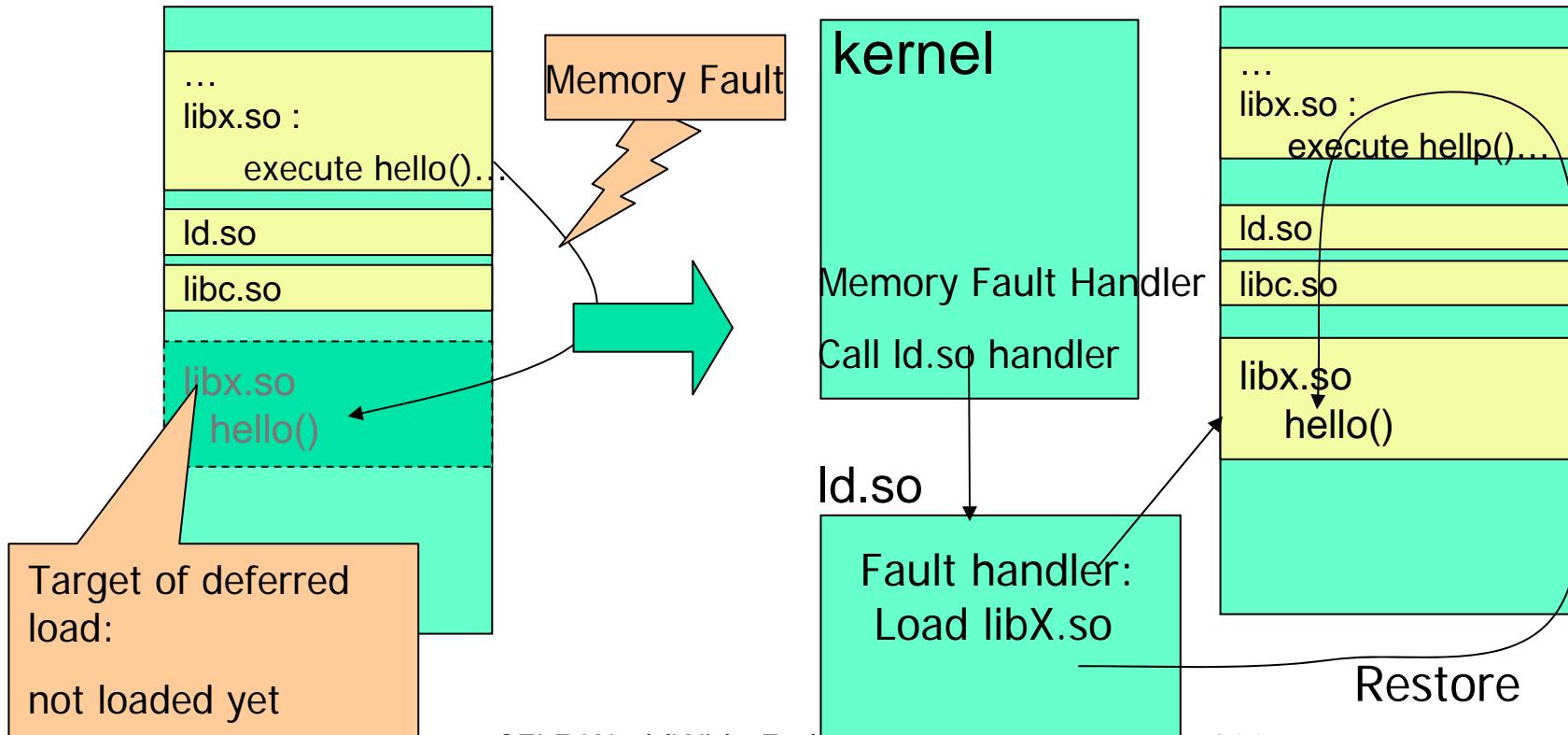
Conflict of Symbols

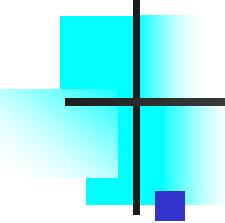
- To resolve conflict, .got and/or .data of libraries is modified
- This makes dirty .got/.data pages.



Our approach: Deferred Dynamic Loading

- Id.so does not load libraries before main().
 - Memory fault occurs when a library invoked
 - Route memory fault to a handler in Id.so
 - The handler loads library





Prerequisite

- Prelink enabled
 - Current implementation on MontaVista CEE3.1
 - kernel 2.4.20, glibc 2.3.2

We are working on kernel 2.6 now.

Code modified for our implementation

Major Changes for the kernel

- arch/arm/kernel/call.S :
 - Add system call
 - 1. Registering the fault handler
 - 2. Obtaining register info at fault to resume
- arch/arm/kernel/sys_arm.c :
 - Replace return PC address to redirect fault handling
- arch/arm/kernel/dlfault.c : (new)
 - Handler code for fault and misc.
- arch/arm/mm/fault-common.c :
 - Add branches at the regular memory fault handler
- init/main.c :
 - Reading library address information to identify a target virtual address space for deferred loading

Code modified for our implementation

■ Major Changes of glibc (ld.so)

- elf/rtld.c :
 - Enabling deferred loading when configured by env
 - Fault handler and misc
- elf/dl-load.c :
 - Storing dynamic load information for deferred loading
 - Loader body for deferred loading
- elf/dl-init.c :
 - Library wise initialization for deferred loading
- elf/conflict.c :
 - Conflict processing for deferred loading
- include/link.h :
 - Added variables (load management, addr info)
- sysdeps/genelic/ldsodefs.h :
 - Added variables (enabled/disabled)

- Patches will be published on CELF web-site

Source Code (memory fault handler)

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)){ // Within the library area ?
            dl_fault_savereg(tsk,regs,addr); // Save restore info (register info)
            dl_fault_setpc(tsk,regs); // Set return address to ld.so hanndler
        }
        else{
            __do_user_fault(tsk,addr,error_code,SEGV_MAPERR,regs);
        }
    }
    else
        __do_kernel_fault(mm,addr,error_code,regs);
}
```

Source Code: (load handler in ld.so)

Kernel -> Load Handler -> Return with process below

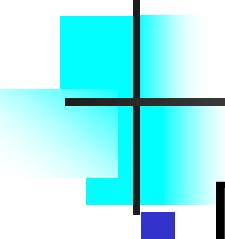
kernel

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){
        // Find out which .so to load with the load address info (link_map)
        // that the fault address matches
        ...
    }
    if(!found){ // If not found, delete handler registration and invoke fault again
        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); // Resume the registers at fault
    }
}
```

Resume to
just before
fault



Enable/Disable

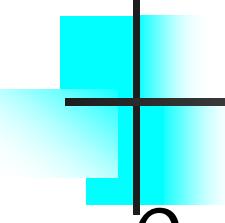
- Disable per library (for avoiding issues)

Write library path to disable in
“/etc/ld.so.forbid_lazyload”

- Disable per process (for debugging)

Environment variable
("DL_LAZY_LOAD")

e.g. `DL_LAZY_LOAD=1 # ON`



Remaining Issues

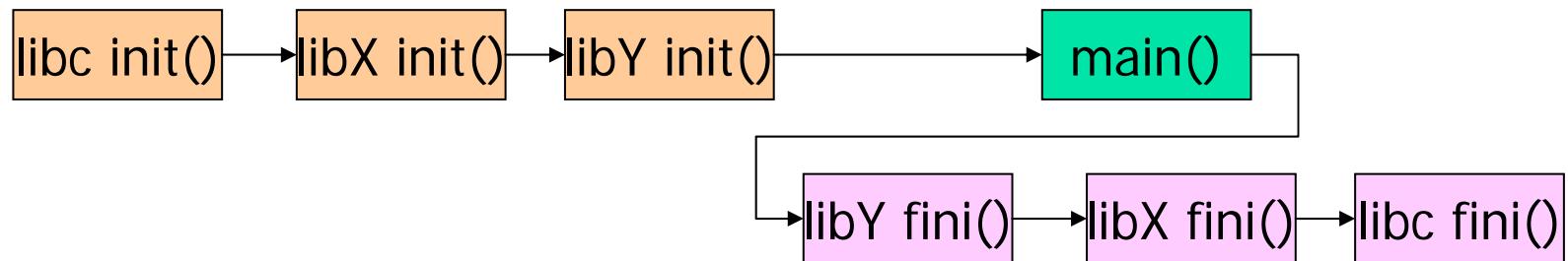
- Call sequence of init/fini
- Using dlopen()/dlsym()
- Race condition at fault under multithread

Welcome for Improvement Proposal

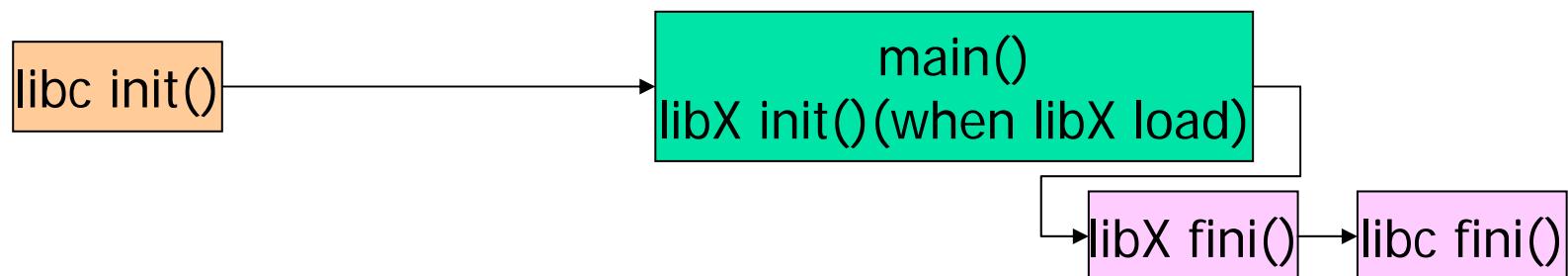
Issue(1):Sequence of init/fini

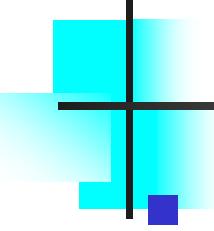
- init() calling with wrong order.
 - init()/fini() is not always called when not loaded.

Original



Deferred dynamic loading



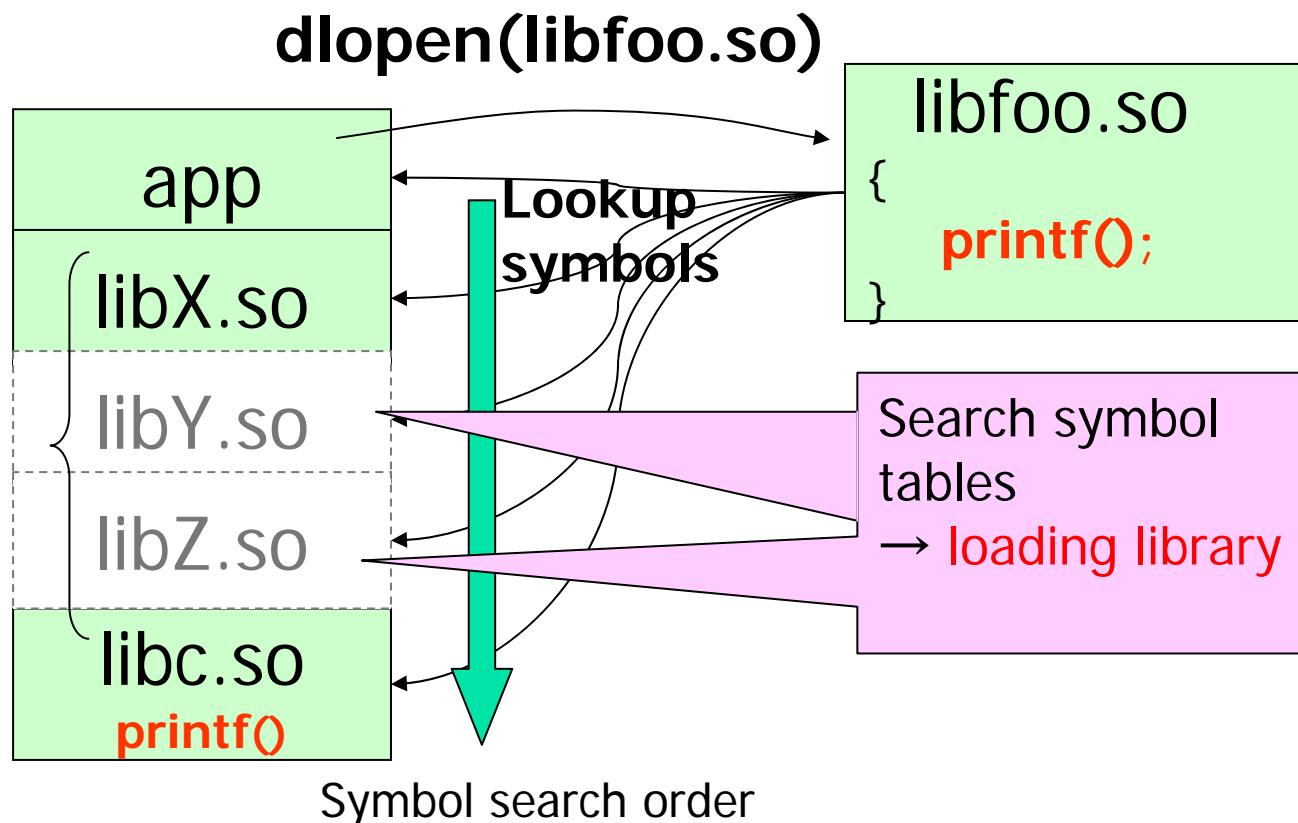


Workaround for Issue(1)

- Almost libraries, init() initialize only library local variable.
 - So, no problem usually happen.
- If it is not the case, disable deferred dynamic loading the library using `ld.so.forbid_lazyload`.

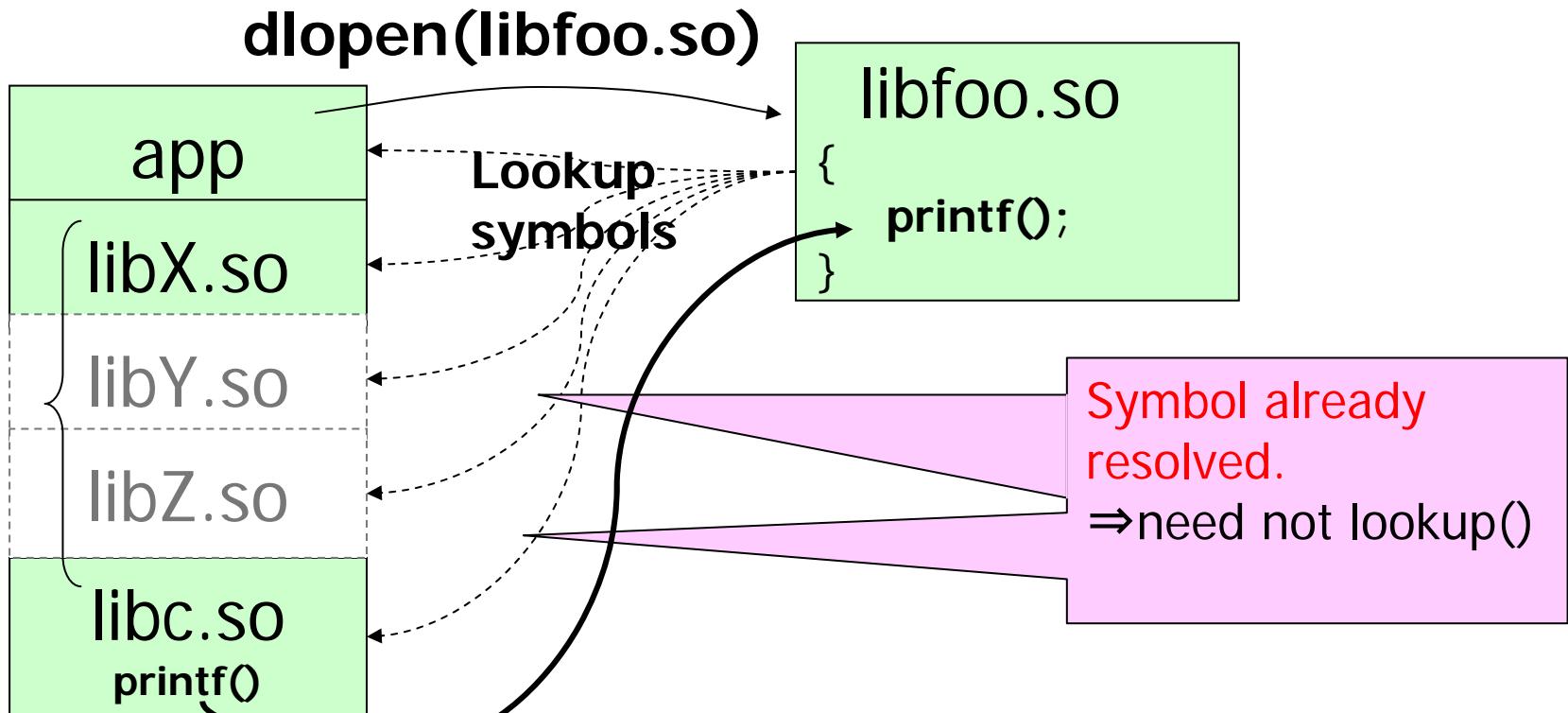
Issue(2): Using dlopen()/dlsym()

- When dlopen()/dlsym() called, almost libraries loaded unnecessarily.



Workaround for Issue(2)

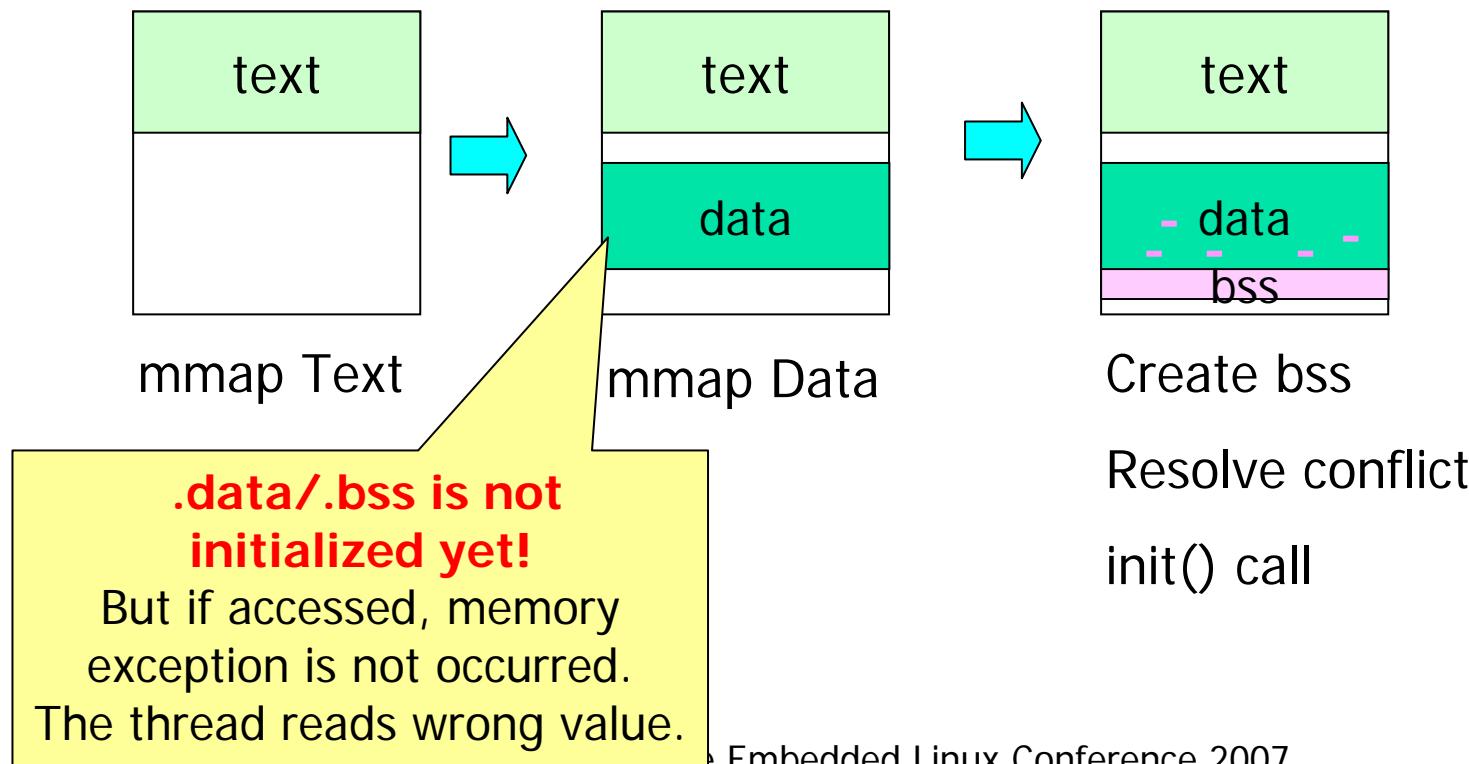
Link library (dlopened library)
⇒ all symbol resolved at prelinking.



Issues(3) : multithread

- A thread can execute the code during other thread is loading the library.

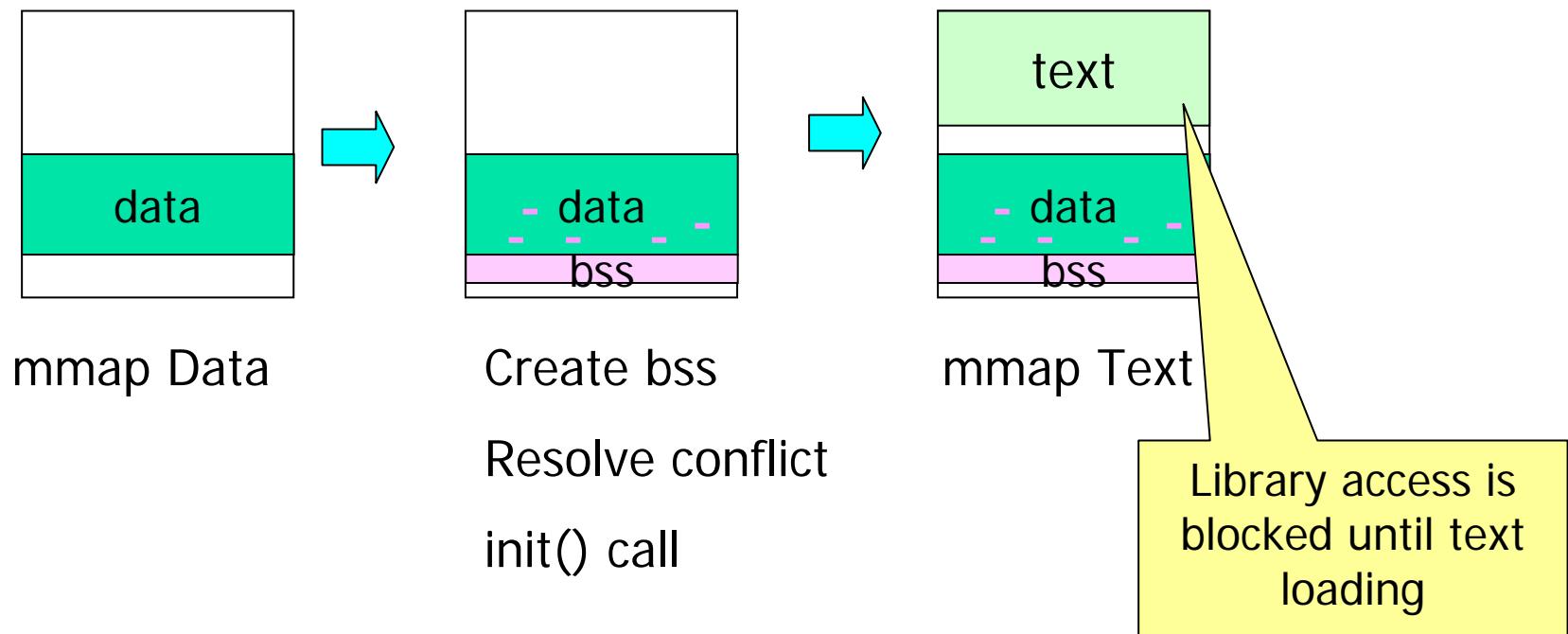
Library Loading Process(Original)



Workaround of Issues(3)

- Change Loading process.
 - First access must be from text section.

Library Loading Process (deferred mode)

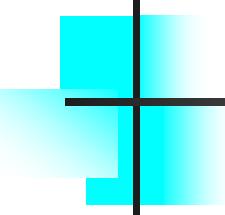


Number of Library link in Fedora Core6

Ranking Top 20

Program Name	Number of Linking Libraries
/usr/bin/evolution	101
/usr/bin/evolution-2.8	101
/usr/bin/bug-buddy	92
/usr/bin/ekiga	92
/usr/bin/gnome-about-me	91
/usr/bin/rhythmbox	88
/usr/bin/gnome-help	85
/usr/bin/yelp	85
/usr/bin/nautilus	84
/usr/bin/nautilus-connect-server	84
/usr/bin/nautilus-file-management-properties	84
/usr/bin/totem	83
/usr/lib/openoffice2.0/program/sdraw.bin	83
/usr/lib/openoffice2.0/program/simpress.bin	83
/usr/bin/create-branching-keyboard	81
/usr/bin/gok	81
/usr/bin/gpilotd-control-applet	81
/usr/bin/totem-video-thumbnailer	81
/usr/lib/openoffice2.0/program/scalc.bin	81

CELF WorldWide Embedded Linux Conference 2007

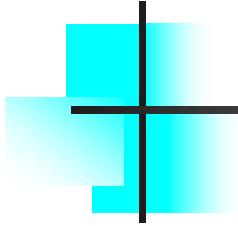


Effectiveness

- Assumed condition
 - 35 process running
 - 40 libraries linking per process
 - 60% of library is not necessary
- Reduction of RAM pages

$$35 \times (40 \times 0.6) \times 4\text{KB} = \sim 3.36\text{M}$$

(Further, due to less virtual space required, PTE cache is saved (up to several hundred kilobytes))



Thank you!