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.

![RAM usage at a CE appliance's](chart_image)
Background: Using dynamic lib Case1

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

**CASE1**

```
main()
    funcA();
    funcB();

funcA()

funcB()

funcC()
    funcD();

funcD()
```
Background: Using dynamic lib Case2

libD.so also wastes memory.

```c
main()
{
    funcA(False);
    funcB();
}

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

funcB();

funcD();
```

CASE2

app

libA.so

libB.so

libD.so

Never Called

(X is always FALSE)
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 \textit{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.

FILE Image Memory Image

... .text ...
... .rodata ...
.interp .interp
.data .data
.dynamic ... .dynamic ...
.got .got
.comment .comment

mmap() file per page.

Using the residual area (filled by file contents)
Zero page mapping

File offset of bss(section size in file is 0)
Padding 0 in .bss

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

 extract from _dl_map_object_from_fd()

...  
.text  
...  
.rodata  
.interp  
.data  
.dynamic  
...  
.got  
.bss  

Part of .bss that resides on same page with .data is filled with zero before main().

CELF WorldWide Embedded Linux Conference 2007
Copyright (C) 2007 Matsushita Electric Industrial Co., Ltd.
Why conflict occurs

An example case:

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

**Normal case**

```c
main(){
    
}
int X=100;

extern int X;
...
print(X);

pointer X(.got)
```

**Conflict case**

```c
extern int X;
main(){
    
}
int X=100;

extern int X;
...
print(X);

pointer X (.got)
```

Entity X is located when X is referenced by application.

.got of libY.so in the memory space is modified at runtime (before main()).
Conflict of Symbols

To resolve conflict, .got and/or .data of libraries is modified.

This makes dirty .got/.data pages.

Rewrite .dot, .data
Our approach: Deferred Dynamic Loading

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

```
... ld.so:
    execute hello()

Target of deferred load:
not loaded yet
```

```
... libc.so

kernel

Memory Fault Handler
Call ld.so handler

Fault handler:
Load libX.so

libx.so:
execute hello()

Id.so

Id.so

Restore
```
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

```c
Fault handler

do_translation_fault()

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 handler
        }
        else{
            __do_user_fault(tsk,addr,error_code,SEGV_MAPERR,regs);
        }
    }
    else
    __do_kemel_fault(mm,addr,error_code,regs);
}
```
Source Code: (load handler in ld.so)

Kernel -> Load Handler -> Return with process below

```
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->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->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->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
}
```
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**

```
<table>
<thead>
<tr>
<th>libc init()</th>
<th>libX init()</th>
<th>libY init()</th>
<th>main()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>libY fini()</th>
<th>libX fini()</th>
<th>libc fini()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Deferred dynamic loading**

```
<table>
<thead>
<tr>
<th>libc init()</th>
<th>main()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>libX init() (when libX load)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>libX fini()</th>
<th>libc fini()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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.

Symbol search order

```
dlopen(libfoo.so)
```
Workaround for Issue(2)

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

dlopen(libfoo.so)

app
libX.so
libY.so
libZ.so
libc.so

libfoo.so
{
  printf();
}

Symbol already resolved.
⇒ need not lookup()
Issues(3) : multithread

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

Library Loading Process (Original)

- mmap Text
- mmap Data
- Create bss
- Resolve conflict
- init() call

.data/.bss is not initialized yet!
But if accessed, memory exception is not occurred.
The thread reads wrong value.
Workaround of Issues (3)

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

Library Loading Process (deferred mode)

- mmap Data
- Create bss
- Resolve conflict
- init() call

Library access is blocked until text loading
### Number of Library link in Fedora Core6

#### Ranking Top 20

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