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
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.

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:

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

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

```
funcB()
{
}
```

```
funcD()
{
}
```

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)**

<table>
<thead>
<tr>
<th>Nr</th>
<th>Name</th>
<th>Type</th>
<th>Addr</th>
<th>Off</th>
<th>Size</th>
<th>ES</th>
<th>Flg</th>
<th>Lk</th>
<th>Inf</th>
<th>Al</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>NULL</td>
<td>00000000</td>
<td>000000</td>
<td>000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>.note.ABI-tag</td>
<td>NOTE</td>
<td>000000f4</td>
<td>0000f4</td>
<td>000020</td>
<td>0</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>.hash</td>
<td>HASH</td>
<td>00000114</td>
<td>000114</td>
<td>0004d0</td>
<td>0</td>
<td>A</td>
<td>3</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>.dynsym</td>
<td>DYNSYM</td>
<td>000005e4</td>
<td>0005e4</td>
<td>000840</td>
<td>10</td>
<td>A</td>
<td>4</td>
<td>2a</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>.dynstr</td>
<td>STRTAB</td>
<td>0000ea24</td>
<td>000ea24</td>
<td>0004ea</td>
<td>0</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>.gnu.version</td>
<td>VERSYM</td>
<td>00012d2</td>
<td>0012d2</td>
<td>000108</td>
<td>02</td>
<td>A</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>.gnu.version_d</td>
<td>VERDEF</td>
<td>00013dc</td>
<td>0013dc</td>
<td>00005c</td>
<td>0</td>
<td>A</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>.gnu.version_r</td>
<td>VERNEED</td>
<td>0001438</td>
<td>001438</td>
<td>0000b0</td>
<td>0</td>
<td>A</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>.rel.dyn</td>
<td>REL</td>
<td>00014e8</td>
<td>0014e8</td>
<td>000108</td>
<td>08</td>
<td>A</td>
<td>3</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>.rel.plt</td>
<td>REL</td>
<td>00016b0</td>
<td>0016b0</td>
<td>0001d8</td>
<td>08</td>
<td>A</td>
<td>3</td>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>.init</td>
<td>PROGBITS</td>
<td>0001888</td>
<td>001888</td>
<td>000014</td>
<td>00</td>
<td>AX</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>.plt</td>
<td>PROGBITS</td>
<td>000189c</td>
<td>00189c</td>
<td>000030</td>
<td>04</td>
<td>AX</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>.text</td>
<td>PROGBITS</td>
<td>0001c5c</td>
<td>001c5c</td>
<td>003808</td>
<td>00</td>
<td>AX</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>__libc_freeres_fn</td>
<td>PROGBITS</td>
<td>0005464</td>
<td>005464</td>
<td>000ae0</td>
<td>0</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>.fini</td>
<td>PROGBITS</td>
<td>0005504</td>
<td>005504</td>
<td>00000c</td>
<td>00</td>
<td>AX</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>.rodata</td>
<td>PROGBITS</td>
<td>0005510</td>
<td>005510</td>
<td>0003f8</td>
<td>0</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>.interp</td>
<td>PROGBITS</td>
<td>0005908</td>
<td>005908</td>
<td>000014</td>
<td>00</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>.data</td>
<td>PROGBITS</td>
<td>000e000</td>
<td>00e000</td>
<td>000070</td>
<td>0</td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>__libc_subfreeres</td>
<td>PROGBITS</td>
<td>000e070</td>
<td>00e070</td>
<td>000008</td>
<td>0</td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>.eh_frame</td>
<td>PROGBITS</td>
<td>000e078</td>
<td>00e078</td>
<td>000004</td>
<td>0</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>.dynamic</td>
<td>DYNAMIC</td>
<td>000e07c</td>
<td>00e07c</td>
<td>000010</td>
<td>08</td>
<td>MA</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>21</td>
<td>.ctors</td>
<td>PROGBITS</td>
<td>000e16c</td>
<td>0016c</td>
<td>000008</td>
<td>0</td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>22</td>
<td>.dtors</td>
<td>PROGBITS</td>
<td>000e174</td>
<td>00174</td>
<td>000008</td>
<td>0</td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>23</td>
<td>.jpr</td>
<td>PROGBITS</td>
<td>000e17c</td>
<td>0017c</td>
<td>000004</td>
<td>0</td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>24</td>
<td>.got</td>
<td>PROGBITS</td>
<td>000e180</td>
<td>00e180</td>
<td>0001c0</td>
<td>04</td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>.bsa</td>
<td>NOBITS</td>
<td>000e234</td>
<td>00e234</td>
<td>000a60</td>
<td>0</td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>26</td>
<td>.comment</td>
<td>PROGBITS</td>
<td>00000000</td>
<td>000000</td>
<td>00005c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

---

**Note:**
- 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

---

**Figure:**

- Memory usage breakdown
- Table showing memory usage for different sections
- Diagram illustrating memory allocation and usage

---

**References:**
- Matsushita Electric Industrial Co., Ltd.
- Linux documentation
- Dynamic link resolution

---

**Questions:**
- What are the implications of using more than one page/library of RAM?
- How can RAM usage be optimized when linking dynamic libraries?
Analysis(2)

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

```c
_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 ((__builtin_expect ((__mprotect ((caddr_t) (zero & ~(GL(dl_pagesize) - 1)),
                        GL(dl_pagesize), c->prot|PROT_WRITE) < 0, 0))

    memcpy ((void *) zero, '¥0', zeropage - zero);
    if ((__builtin_expect ((__mprotect ((caddr_t) (zero & ~(GL(dl_pagesize) - 1)),
                          GL(dl_pagesize), c->prot|PROT_WRITE) < 0, 0))

    if (zeroend > zeropage)
```
c.f. On conflict in prelink

Cause of conflict occurrence

- 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

libX.so

libY.so

main()
{
    funcX();
}

extern int foo;
funcX()
{
    funcY();
}

int foo;
funcY()
{
}

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

Case with symbol copy (R_ARMD_COPY type)

app

libX.so

libY.so

main()
{
    funcX();
    funcY();
}

extern int foo;

int foo;
funcY()
{
}

For symbol referred in execution file side, 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 hundreds 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:**

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

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

```
funcB();
```

```
funcD(){
}
```

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.
LazyLoading (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
     - unmmap 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

```c
#define _BAD AREA

void 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);
}
```

Fault handler

```
do_translation_fault()
```

arch/arm/mm/fault-common.c
Lazy Loading (Source Code: excerpts)

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){
        // 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
                _dl_init_lazy(l); // Do conflict processing within function
                l->l_lazy = 0; // Call initialization of the library
                /* 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
}

elf/rtld.c
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

- Assuming that each of 35 processes links to 40 libraries, and that 60% of them need not be loaded any more,
  
  \[ 35 \times (40 \times 0.6) \times 4\text{KB} = 3.36\text{M} \]
  
  → 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)

Consideration 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 guaranteed
- No effect for libraries which dlopen()
- Race condition at fault under multithread

Performance

- While suppressing to load unused libraries, 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 warranted (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).

```
init(): executed in the order of loading, regardless of dependency; unless loaded, not executed
fini(): only loaded libraries are called in the reverse order of dependency
```

**Diagram:**

- Dependency:
  - (Libpthread.so depends on libc.so, etc...)
- Processing of ordinary loader:
  - app
  - libY.so
  - libX.so
  - libpthread.so
  - libc.so
- Processing of Lazy loading:
  - 1. Init: executed in the order of loading, regardless of dependency; unless loaded, not executed
  - 2. Fini: only loaded libraries are called in the reverse order of dependency

Example:
- 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:) dynamic link of the library to dlopen()
  - x
  - memory is not consumed (although some memory for management (link_info; about 500B/library) is consumed).

```scheme
app
  libA.so
    libB.so
    libC.so
  libc.so
    fopen()

libxxx.so
  
  { 
    fopen();
  }
```

By linking, the library is now target of prelink, and the symbol has been resolved before loading (can omit lookup at dlopen()).
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 mmap()'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)

- text
- data
- text
- data
- bss

mmap() text
mmap() data
fix bss/conflict

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

Current implementation

- text
- data
- text
- data
- bss

mmap() text
fix bss/conflict
mmap() data
rewriting by conflict

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

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

Reverse the order to load

Here, as loading has been completed, no problem occurs
Thanks you

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