Avoiding OOM (Out of Memory) on Embedded Linux

April 15, 2008
YoungJun Jang yj03.jang@samsung.com

Software Laboratories
Samsung Electronics
Contents

- Problem Statement
- Background
  - Demand Paging
  - Overcommit
  - Address Space Usage: Virtual vs. Physical
  - Overcommit Problem
- Approach & Implementation
  - RSS Quota
- Limitation & Future Works
- Another Approaches
Problem statement

What’s the problem of this code?

On Linux system
64M RAM available

```c
main(void)
{
    char *p;
    for (i=0; i<100; i++)
    {
        p = malloc( 1M );
        if ( p == NULL )
        {
            // error handling
        }
    }
}
```

[ CODE 1 ]
Then how about that code?

On Linux system
- 64M RAM available

```c
main(void)
{
    char *p;
    for (i=0; i<100; i++)
    {
        p = malloc(1M);
        if (p == NULL)
        {
            // error handling
        }
        memset(p, 0, 1M);
    }
}
```

[ CODE 2 ]
Demand Paging

- Separated virtual and physical address
- Reserve virtual address first, and allocate physical memory when accessing virtual memory later.

```c
textmain(void)
{
    ...
    char *p;
    p = malloc(10M);
    ...
    memset(p+2M, 0, 5M);
    ...
    memtest.c
```
Overcommit

The total aggregation of virtual address space can be larger than physical memory size.

Application can allocate ‘large’ memory without considering physical memory size.
Memory Usage Example.

![Memory Usage Graph](image-url)
Memory Usage Example. (per each area)

Address Space Usage: Virtual vs. Physical (2/2)

Code RSS

Heap RSS

Data RSS

Stack RSS
Overcommit Problem (1/2)

Allocation request (ex. malloc()) larger than available memory size always succeeds.
Linux invokes OOM killer and kills application.

main(void)
{
    ...
    char *p;
    p = malloc(10M);
    ...
    memtest(p,0,10M);
    ...
}

Allocated pages to other processes
Free pages (FreeMem, PageCache,...)
* Page size: 4KB

- Page faults occur 6*256 times, App will be killed by OOM-killer
Overcommit Problem (2/2)

Generally, SW developer thinks:
- `malloc()` will return NULL, if the system has no available memory.

Then, developers add error handling code when `malloc()` returns NULL.

But, indeed linux kernel doesn’t return NULL.
- Instead, invokes OOM killer when allocated spaces are really used in low memory condition.

So, the system gains no opportunity to handling ‘allocation failure’ error.

It’s a problem for all embedded linux systems.
Limit maximum physical memory size per each application.

If memory usage exceeds limitation, malloc() returns NULL.

**Data Structure**
- `cur_rss`: current rss ( = current physical memory usage )
- `max_rss`: maximum rss ever been used since process created
- `limit_max`: rss limitation of each process can use

\[
\text{cur_rss} \leq \text{max_rss}
\]
**Why ‘max_rss’ is needed?**

- $\text{max\_rss}$: maximum rss ever been used since process created

[example: trace physical memory usage]

We can say that this application’s real memory usage is 70. With the ‘max\_rss’ value, we can limit application’s memory usage.
Approach & Implementation (3/5)

2 phases: memory usage profiling, run-time allocation control

- Profiling: collect max RSS for each address section (text, data, ..) of target process
- Run-time memory allocation control: admission control of memory area allocation

Based on profiling result, we can do allocation control.

- App's maximum memory usage
  \[ M_{\text{app required}} = i.T.\text{max} + i.D.\text{max} + i.H.\text{max} + i.S.\text{max} \]

- App's total sum of virtual address space
  \[ M_{\text{app virt total}} = i.T.\text{size} + i.D.\text{size} + i.H.\text{size} + i.S.\text{size} \]

\[ M_{\text{app required}} << M_{\text{app virt total}} \]

Memory Usage Profiling
With the limit value and current rss, we can decide allocation will succeed or not.

To succeed in allocation, 2 conditions must be fulfilled.

- allocation size < limit - rss
- allocation size < system free memory
  - Because another process can use free memory.

**Heap.Limit = 20MB**
**Heap.RSS = 15MB**

```c
main(void)
{
  ...

  char *p;
p = malloc(10M);      //Malloc() FAILS!!
  ...
```

Because 10M > (Heap.Limit - Heap.RSS)
Other regions (text, data, stack region) uses memory as well as heap.

- Keep memory usage per each region.
- Need for allocation control of fork(), exec(), mremap(), ...

Virtual Address Space

Physical Memory

Max. 6MB
Max. 20MB

Current 2MB

2MB

Heap will use additional 2MB memory

Text will use additional 4MB memory

A malloc(4MB) should be failed, even though free memory (6MB) is still enough for requested memory(4MB).
RSS Quota (1/3)

System Diagram

1. Setting RSS Quota
2. Save settings into memory descriptor
3. Trace & update physical memory page counters
4. Memory allocation requests
5. Compare the counter with the limit value
6. Return results (success or fail)

H/W – Physical Memory

Application

System Call Interface

RSS Quota Parameter Manager

RSS Limit Values

VAS Admission Control

Virtual Memory Mgmt.

Physical Memory Mgmt.

Physical Memory Tracer

Process Memory Descriptor

Physical Memory

H/W – Physical Memory

File System

Ext3

Proc FS

File System

Process Memory

Physical Memory Mgmt.

H/W – Physical Memory
RSS Quota (2/3)

Data Structure
- `cur_rss[]`
- `max_rss[]`: used for profiling.
- `limit_max[]`: used for allocation control.

```c
int main(void) {
    ... 
    pthread_create( &tid, ... );
    ...
}
```

Kernel Space

```
limit_max[4];
cur_rss[4];
max_rss[4];
```

User Space

- App
- Glibc

Filesystem

- `testapp`
- `glibc`

Text

Data

BSS+Heap

Heap

Heap stack

2MB

Stack

`vm_start`, `vm_end`

`*vm_next`

`*pgd`

`pte_t[]`

`pgd_t[]`

`vm_area_struct`

`task_struct`

`stack`

`comm[]`
RSS Quota (3/3)

Results

<table>
<thead>
<tr>
<th>Commands</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code># ./rssq_test 10</code></td>
<td>malloc start... : 1M x 10</td>
</tr>
<tr>
<td>malloc (000)</td>
<td>malloc (001)</td>
</tr>
<tr>
<td>memory freeing...</td>
<td>memory free complete.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Commands</th>
<th>Results</th>
</tr>
</thead>
</table>
| `# ps` | `PID Uid VmSize Stat Command`
| 1 0 528 S init | 2 0 SW< [ksoftirqd/0] |
| 316 0 620 S /bin/sh | 317 0 404 S ./rssq_test 10 |
| 318 0 664 R ps | `malloc & memset completed` |

<table>
<thead>
<tr>
<th>Commands</th>
<th>Results</th>
</tr>
</thead>
</table>
| `# cat /proc/317/rss` | `2684 90 13 4`
| Process Max : 10736K | Code Max : 360K |
| Data Max : 52K | Stack Max : 16K |
| Other Max : 10308K | `malloc error` |
| memory freeing... | memory free complete. |

<table>
<thead>
<tr>
<th>Commands</th>
<th>Results</th>
</tr>
</thead>
</table>
| `# echo "2684 90 13 4 /test/rssq_test"` | `>` `/proc/sys/vm/rss_quota`
| `# cat /proc/sys/vm/rss_quota` | `2684 90 13 4 /test/rssq_test`
| `#` | `#` |

test program (1M x 10 allocation) Restrict memory based on profiling After restrict, allocate 1M x 11
Limitation & Future Works

Limitation
- We must profile each application.
- Inaccuracy of calculating system free memory.

Future Works
- Calculates free memory more accurately.
- Improves PFRA (Page Frame Reclaiming Algorithm)
- Shared Memory Accounting.
Another Approaches (1/3)

- No overcommit
  - It doesn’t use overcommit policy.
  
  - “echo 2 > /proc/sys/vm/overcommit_memory”

- Pros
  - We can sure that OOM will never occur.

- Cons
  - There’s no merit of demand paging.
  - Some or more applications may not run.
Another Approaches (2/3)

**OOM notify to application**
- When occurred lack of kernel memory, kernel notify it to user applications. And each application manages OOM situation.

**WinCE**

**mem notify patch**
- [http://lwn.net/Articles/267013/](http://lwn.net/Articles/267013/)

**Pros**
- Effective manipulation of OOM.
  - Because application knows well which memory allocation is useless than the other allocations.

**Cons**
- Application developer should consider about OOM.
- Also, the existing application code must be changed.
Another Approaches (3/3)

- Improves OOM policy
  - Improves victim selecting method.

- Android platform
  - Android has its own victim selecting method.
  - There’s “importance hierarchy” based on the state of components.

- Pros
  - Effective than kernel OOM killer’s victim policy.

- Cons
  - OOM still exists. Because it changes only victim policy.
Thank You.
Q&A