An Evolutionary Study of Linux Memory Management for Fun and Profit

Jian Huang

Moinuddin K. Qureshi  Karsten Schwan

Georgia Tech  CARET
Virtual Memory: A Long History
Virtual Memory: A Long History

Virtual Memory (per process)

Physical Hardware

DRAM

Disk
Virtual Memory: A Long History

Virtual Memory (per process)

Physical Hardware

DRAM

Disk
Virtual Memory: A Long History

Virtual Memory (per process)

Physical Hardware

DRAM

Disk

OS Core Component + Development
Virtual Memory: A Long History

Virtual Memory (per process)

Physical Hardware

DRAM

Disk

OS Core Component + 60 Years Development ➞ Pervasively Used
Why Memory Manager Study Matters?

Features & Functions
Why Memory Manager Study Matters?

Features & Functions

Hardware Support
Why Memory Manager Study Matters?

Features & Functions

Hardware Support

System Reliability
Why Memory Manager Study Matters?

Features & Functions

Hardware Support

System Reliability

Study on Memory Manager
Why Memory Manager Study Matters?

- Features & Functions
- Hardware Support
- System Reliability

Study on Memory Manager → Building Better Memory Manager
On the Study of Memory Management

Understanding the Linux Virtual Memory Manager
[Mel Gorman, July 9, 2007]
On the Study of Memory Management

Understanding the Linux Virtual Memory Manager
[Mel Gorman, July 9, 2007]

Approach: Source code analysis, Linux 2.4, 2.6
On the Study of Memory Management

Understanding the Linux Virtual Memory Manager
[Mel Gorman, July 9, 2007]

Approach: Source code analysis, Linux 2.4, 2.6
On the Study of Memory Management

Understanding the Linux Virtual Memory Manager
[Mel Gorman, July 9, 2007]

Approach: Source code analysis, Linux 2.4, 2.6

Our Focus: Patch study, Linux 2.6 – 4.0
On the Study of Memory Management

Understanding the Linux Virtual Memory Manager
[Mel Gorman, July 9, 2007]

Approach: Source code analysis, Linux 2.4, 2.6

Our Focus: Patch study, Linux 2.6 – 4.0

Pattern + Memory Bug + Optimization + Semantic
Preview of Our Findings

- Code changes are highly concentrated around the key functions
- 80% of patches $\rightarrow$ 25% of its source code
- ......
Preview of Our Findings

- Code changes are highly concentrated around the key functions
- 80% of patches → 25% of its source code
- ......

- Memory error – Checking – Concurrency – Logic – Programming
- Memory errors: Null pointer & page alignment
- Complex page states → Checking & logic bugs
- ......
Preview of Our Findings

- Code changes are highly concentrated around the key functions
- 80% of patches → 25% of its source code
- ......

- Memory error – Checking – Concurrency – Logic – Programming
- Memory errors: Null pointer & page alignment
- Complex page states → Checking & logic bugs
- ......

- Data structures -- Policy trade-off -- Fast path
- 4 data structures, 5 design trade-offs, 8 types of fast paths
- ......
Preview of Our Findings

- Code changes are **highly concentrated** around the key functions
- 80% of patches $\rightarrow$ 25% of its source code
- ......

- Memory error – Checking – Concurrency – Logic – Programming
- Memory errors: **Null pointer & page alignment**
- Complex page states $\rightarrow$ **Checking & logic bugs**
- ......

- Data structures -- Policy trade-off -- Fast path
- 4 data structures, 5 design trade-offs, 8 types of fast paths
- ......

- 35 key functionalities in 13 hot files
- The well-developed **memory allocators** still have many **checking & lock bugs**
- ......
Methodology Used in Our Study

- Memory Allocation
- Resource Controller
- Garbage Collection
- Page Cache & Write-back
- Virtual Memory Management
- Swapping
- Exception Handling
- Misc (e.g., data structure)

8 components
Methodology Used in Our Study

- Memory Allocation
- Resource Controller
- Garbage Collection
- Page Cache & Write-back
- Virtual Memory Management
- Swapping
- Exception Handling
- Misc (e.g., data structure)

8 components


4587 patches in 5 years
Methodology Used in Our Study

Patches

- Description
- Follow-up Discussions
- Source Code Analysis
Methodology Used in Our Study

Patches

- Description
- Follow-up Discussions
- Source Code Analysis

Labeling & MChecker

BugID

- Commit Time
- Component
- Type
- Causes
- Involved Functions
- ...

MPatch
How Is the Memory Manager Changed?
How Is the Memory Manager Changed?

The LoC has increased by 60% since Linux 2.6.32.
Where Is the Memory Manager Changing?
Where Is the Memory Manager Changing?

Memory Manager Components
Where Is the Memory Manager Changing?
Where Is the Memory Manager Changing?

Number of Committed Patches
Where Is the Memory Manager Changing?
Where Is the Memory Manager Changing?

80% of the code changes → 25% of the source code
Where Is the Memory Manager Changing?
Where Is the Memory Manager Changing?
Where Is the Memory Manager Changing?
Where Is the Memory Manager Changing?

13 hot files from 90 files → recent development focus
Why Is the Memory Manager Changed?

- Bug
- Code Maintenance
- Optimization
- New Feature

<table>
<thead>
<tr>
<th>Percentage (%)</th>
<th>Linux version (released year)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.33 (2010)</td>
<td></td>
</tr>
<tr>
<td>2.6.38 (2011)</td>
<td></td>
</tr>
<tr>
<td>3.2 (2012)</td>
<td></td>
</tr>
<tr>
<td>3.10 (2013)</td>
<td></td>
</tr>
<tr>
<td>3.14 (2014)</td>
<td></td>
</tr>
<tr>
<td>4.0-rc4 (2015)</td>
<td></td>
</tr>
</tbody>
</table>
Why Is the Memory Manager Changed?

![Graph showing the percentage of code maintenance and new feature changes in Linux versions from 2010 to 2015. The graph displays a line graph with two lines: one for code maintenance and one for new feature. The y-axis represents the percentage (%) and the x-axis represents the Linux version (released year) from 2.6.33 (2010) to 4.0-rc4 (2015).]
Why Is the Memory Manager Changed?

70% more bugs in well-developed memory manager!
On the Bugs in Memory Manager
On the Bugs in Memory Manager

Types of Memory Bugs
On the Bugs in Memory Manager

Memory Manager Component
Memory Allocation: 26%, Virtual Memory Management: 22%, GC: 14%
On the Bugs in Memory Manager
On the Bugs in Memory Manager

• Page alignment
• Null pointer
On the Bugs in Memory Manager

- Page alignment
- Null pointer
- Inappropriate check
- Missing check
- Wrong check
On the Bugs in Memory Manager

- Page alignment
- Null pointer

- Lock contention
- Missing lock

- Inappropriate check
- Missing check
- Wrong check
On the Bugs in Memory Manager

- Page alignment
- Null pointer
- Inappropriate check
- Missing check
- Wrong check
- Lock contention
- Missing lock
- Inappropriate check
- Missing check
- Wrong check
- Fault handler
- State update
- Case-by-case
On the Bugs in Memory Manager

- Page alignment
- Null pointer
- Inappropriate check
- Missing check
- Wrong check
- Lock contention
- Missing lock
- Fault handler
- State update
- Case-by-case
- API
- Misc
Memory Bugs: Case Studies

Page Alignment

```c
#include <linux/vm.h>

unsigned long do_mremap(unsigned long addr, struct vm_area_struct *vma);

/* insanity checks first */

if (old_len == 0 || new_len == 0)
    return (unsigned long) -EINVAL;
```
Memory Bugs: Case Studies

Page Alignment

```
mm/nommu.c

@@ -1762,6 +1765,8 @@ unsigned long do_mremap(unsigned long addr,
 struct vm_area_struct *vma;

 /* insanity checks first */

     if (old_len == 0 || new_len == 0)
         return (unsigned long) -EINVAL;
```

**Bug:** device drivers’ mmap() failed.

**Cause:** NOMMU does not do page_align(), which is inconsistent with MMU arch.
Memory Bugs: Case Studies

Page Alignment

```c
unsigned long do_mremap(unsigned long addr, struct vm_area_struct *vma);

/* insanity checks first */
+ old_len = PAGE_ALIGN(old_len);
+ new_len = PAGE_ALIGN(new_len);
if (old_len == 0 || new_len == 0)
    return (unsigned long) -EINVAL;
```

**Bug:** device drivers’ `mmap()` failed.

**Cause:** NOMMU does not do `page_align()`, which is inconsistent with MMU arch.
Memory Bugs: Case Studies

Checking

```c
#include "bootmem.h"

static void __init
free_bootmem_core(bootmem_data_t *bdata, unsigned long addr,
```
Checking

mm/bootmem.c

@@ -156,21 +157,31 @@ static void __init
free_bootmem_core(bootmem_data_t *bdata, unsigned long addr,

Bug: free pages wrongly.
Cause: miss boundary checking.
Memory Bugs: Case Studies

Checking

```
mm/boottmem.c

@@ -156,21 +157,31 @@
static void __init
free_bootmem_core(bootmem_data_t *bdata, unsigned long addr,

+ BUG_ON(!size);
+ /* out range */
+ if (addr + size < bdata->node_boot_start ||
+     PFN_DOWN(addr) > bdata->node_low_pfn)
+ return;
```

**Bug:** free pages wrongly.

**Cause:** miss boundary checking.
Memory Optimizations

Data Structures

- Radix Tree
- Red-black Tree
- Bitmap
- List
Memory Optimizations

4 Data Structures

Radix Tree  Red-black Tree  Bitmap  List

Decentralize data structures: per-core/per-node/per-device approaches.
Memory Optimizations

4 Data Structures
- Radix Tree
- Red-black Tree
- Bitmap
- List

5 Policy Trade-offs
- Latency Vs. Throughput
- Synchronous Vs. Asynchronous
- Lazy Vs. Non-lazy
- Local Vs. Global
- Fairness Vs. Performance
# Memory Optimizations

## Data Structures
- Radix Tree
- Red-black Tree
- Bitmap
- List

## Policy Trade-offs
- Latency Vs. Throughput
- Synchronous Vs. Asynchronous
- Lazy Vs. Non-lazy
- Local Vs. Global
- Fairness Vs. Performance

## Fast Paths
- Code Reduction
- Lockless Optimization
- New Function
- Inline
- State Caching
- Code Shifting
- Group Execution
- Optimistic Barrier
Memory Optimizations: Policy Trade-offs

- Latency Vs. Throughput: 11%
- Lazy Vs. Non-lazy: 16%
- Fairness Vs. Performance: 18%
- Synchronous Vs. Asynchronous: 22%
- Local Vs. Global: 33%
Memory Optimizations: Policy Trade-offs

- Latency Vs. Throughput: 11%
- Lazy Vs. Non-lazy: 16%
- Fairness Vs. Performance: 18%
- Synchronous Vs. Asynchronous: 22%
- Local Vs. Global: 33%

137 patches committed especially for reducing the latencies of memory operations.
Memory Optimizations: Policy Trade-offs

- **Latency Vs. Throughput**: 11%
- **Lazy Vs. Non-lazy**: 16%
- **Fairness Vs. Performance**: 18%
- **Synchronous Vs. Asynchronous**: 22%
- **Local Vs. Global**: 33%

Lazy policy: delay expensive operations. May change the execution order of functions.
Memory Optimizations: Policy Trade-offs

Latency Vs. Throughput
- 11%

Lazy Vs. Non-lazy
- 16%

Fairness Vs. Performance
- 18%

Synchronous Vs. Asynchronous
- 22%

Local Vs. Global
- 33%

Lazy policy: delay expensive operations. May change the execution order of functions.

- `vmalloc` Lazy TLB flush, lazy unmapping
- `mempolicy` Lazy page migration between nodes
- `huge_memory` Lazy huge zero page allocation
Memory Optimizations: **Policy Trade-offs**

- Latency Vs. Throughput: 11%
- Lazy Vs. Non-lazy: 16%
- Fairness Vs. Performance: 18%
- Synchronous Vs. Asynchronous: 22%
- Local Vs. Global: 33%

Mostly considered in memory allocation & GC.
Memory Optimizations: Policy Trade-offs

Latency Vs. Throughput 11%
Lazy Vs. Non-lazy 16%
Fairness Vs. Performance 18%
Synchronous Vs. Asynchronous 22%
Local Vs. Global 33%

Async is popular, but be careful to its fault handlers!
Memory Optimizations: Policy Trade-offs

Latency Vs. Throughput: 11%
Lazy Vs. Non-lazy: 16%
Fairness Vs. Performance: 18%
Synchronous Vs. Asynchronous: 22%
Local Vs. Global: 33%

Async is popular, but be careful of its fault handlers!
E.g., early termination
Memory Optimizations: **Policy Trade-offs**

- **Latency Vs. Throughput**: 11%
- **Lazy Vs. Non-lazy**: 16%
- **Fairness Vs. Performance**: 18%
- **Synchronous Vs. Asynchronous**: 22%
- **Local Vs. Global**: 33%

Decentralizing global structures for better scalability
Memory Optimizations: **Policy Trade-offs**

- **Latency Vs. Throughput**: 11%
- **Lazy Vs. Non-lazy**: 16%
- **Fairness Vs. Performance**: 18%
- **Synchronous Vs. Asynchronous**: 22%
- **Local Vs. Global**: 33%

**Decentralizing global structures for better scalability**

E.g., dynamic per-cpu allocator.
Memory Optimizations: Fast Path

- Code Reduction: 34%
- Lockless Optimization: 27%
- New Function: 12%
- State Caching: 8%
- Inline: 6%
- Code Shifting: 5%
- Group Execution: 4%
- Optimistic Barrier: 4%
Memory Optimizations: Fast Path

- Code Reduction: 34%
- Lockless Optimization: 27%
- New Function: 12%
- State Caching: 8%
- Inline: 6%
- Code Shifting: 5%
- Group Execution: 4%
- Optimistic Barrier: 4%

Simplify the slow path logic
Memory Optimizations: **Fast Path**

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Reduction</td>
<td>34%</td>
</tr>
<tr>
<td>Lockless Optimization</td>
<td>27%</td>
</tr>
<tr>
<td>New Function</td>
<td>12%</td>
</tr>
<tr>
<td>State Caching</td>
<td>8%</td>
</tr>
<tr>
<td>Inline</td>
<td>6%</td>
</tr>
<tr>
<td>Code Shifting</td>
<td>5%</td>
</tr>
<tr>
<td>Group Execution</td>
<td>4%</td>
</tr>
<tr>
<td>Optimistic Barrier</td>
<td>4%</td>
</tr>
</tbody>
</table>

### Simplify the slow path logic

E.g., Avoid redundant get/put_page in `munlock_vma_range` as pages will not be referred anymore.

```c
@@ -303,8 +303,10 @@ static void __munlock_pagevec(
    if (PageLRU(page)) {
        lruvec = mem_cgroup_page_lruvec(page, zone);
        lru = page_lru(page);
-            /* We already have pin from follow_page_mask() so we can spare the get_page() here. */
+            /* We already have pin from follow_page_mask() */
+            /* so we can spare the get_page() here. */
    }
```
Memory Optimizations: **Fast Path**

- **Code Reduction**: 34%
- **Lockless Optimization**: 27%
- **New Function**: 12%
- **State Caching**: 8%
- **Inline**: 6%
- **Code Shifting**: 5%
- **Group Execution**: 4%
- **Optimistic Barrier**: 4%

Reduce the usage of lock and atomic operations
Memory Optimizations: Fast Path

- Code Reduction: 34%
- Lockless Optimization: 27%
- New Function: 12%
- State Caching: 8%
- Inline: 6%
- Code Shifting: 5%
- Group Execution: 4%
- Optimistic Barrier: 4%

Reduce the usage of lock and atomic operations
E.g., lockless memory allocator in SLUB
Memory Optimizations: Fast Path

- Code Reduction: 34%
- Lockless Optimization: 27%
- New Function: 12%
- State Caching: 8%
- Inline: 6%
- Code Shifting: 5%
- Group Execution: 4%
- Optimistic Barrier: 4%

Cache states to avoid expensive operations
Memory Optimizations: Fast Path

- Code Reduction: 34%
- Lockless Optimization: 27%
- New Function: 12%
- State Caching: 8%
- Inline: 6%
- Code Shifting: 5%
- Group Execution: 4%
- Optimistic Barrier: 4%

Cache states to avoid expensive operations
E.g., pre-calculate the number of online nodes vs. always calling expensive num_online_nodes
Memory Optimizations: **Fast Path**

- **Code Reduction**: 34%
- **Lockless Optimization**: 27%
- **New Function**: 12%
- **State Caching**: 8%
- **Inline**: 6%
- **Code Shifting**: 5%
- **Group Execution**: 4%
- **Optimistic Barrier**: 4%

Move infrequently executed code from fast path to slow path.
Memory Optimizations: **Fast Path**

- **Code Reduction**: 34%
- **Lockless Optimization**: 27%
- **New Function**: 12%
- **State Caching**: 8%
- **Inline**: 6%
- **Code Shifting**: 5%
- **Group Execution**: 4%
- **Optimistic Barrier**: 4%

**Move infrequently executed code from fast path to slow path**

E.g., in SLUB allocator, slow path executes the interrupt enable/disable handlers, fast path executes them only at fallback.
Memory Semantics

Memory Resource Controller

memory cgroup

charge/uncharge

cgroup management

memcontrol.c
Memory Semantics

Memory Resource Controller

memory cgroup

charge/uncharge

cgroup management

Bug: Concurrency issues

memcontrol.c
Memory Semantics

Memory Resource Controller

memory cgroup

charge/uncharge

cgroup management

memcontrol.c

Bug: Concurrency issues

Cause: missing locks in charging & uncharging pages (truncation, reclaim, swapout and migration)
Memory Semantics

Virtual Memory Management

memory policy
Memory Semantics

Virtual Memory Management

memory policy

- policy definition
- policy enforcement

mempolicy.c
Memory Semantics

Virtual Memory Management

memory policy

policy definition

policy enforcement

mempolicy.c

Bug: policy enforcement failure
Bug: policy enforcement failure

Cause: missing check on page states & statistics, e.g., whether a page is dirty, cache hit/miss rate
Conclusion

- Pattern
- Memory Bug
- Optimization
- Semantic
Conclusion

- Complex page states \(\rightarrow\) Concurrency bugs \(\rightarrow\) Simplified page management
- Fast path \(\rightarrow\) Introduce new errors \(\rightarrow\) Fast path verification
- Bugs in checking \(\rightarrow\) Model checking for memory manager
- ......
Thanks!

Jian Huang
jian.huang@gatech.edu

Moinuddin K. Qureshi    Karsten Schwan

Q&A