16 Memory Management

RAM, Design Space, Examples

January 12 2009
Winter Term 2008/09
Gerd Liefländer
Recommended Reading

- Bacon, J.: Operating Systems (5)
- Bovet, D.: Understanding the Linux Kernel
- Nehmer, J.: Grundlagen modener BS, (4)
- Silberschatz, A.: Operating System Concepts (7)
- Stallings, W.: Operating Systems (7)
- Tanenbaum, A.: Modern Operating Systems (4)
Agenda

- Motivation
- Architecture of RAM Management
- Design Parameters
- Example Memory Managers
  - Ring Buffer
  - Stack
  - Boundary Tag Systems
  - Buddy System
  - Slab Allocating
  - Heap Management
Why Memory Management?

1. Each application needs RAM to run on a CPU, ⇒ we have to establish
   - **static** address regions, e.g. code, global data
   - **dynamic** address regions, e.g. heap or stack

2. The kernel needs memory for its
   - resident part
   - loadable kernel modules

3. Devices often only can use **physically** addressed memory for their **buffers** etc.

⇒ Every executable needs some RAM
Why Memory Management?

Entities of an “application” AS (address regions):

- Code (“text segment” in Unix jargon)
- Data ((un)initialized data segment)
- Heap
- Stack

Data types (entities) of the kernel AS*:

- Buffer
- TCB, page table, free list, bit map, ...

*At the end of the course you should be able to enumerate at least 10 different kernel data types
System Goals concerning Memory

- **Increase (maximize)**
  - memory utilization
  - Sometimes we must reserve some RAM capacity for
    - high priority applications or
    - system emergency functions

- **Reduce (minimize)**
  - application’s response time
  - application’s turnaround time
  - memory manager’s overhead
What does a Memory Manager?

- Keeping track of memory that is currently allocated, i.e. in use or reserved for future use
  - Pinning parts of memory for specific tasks
- free

- Looking for fitting free memory in case of a request
  - If found, allocate free memory according to some policy
- Free memory in case of a release
  - potentially look for free neighbors in order to reunify free neighbored memory pieces
Memory Management (1)

- Programmers want memory being
  - large & fast & non volatile

- *Current technology* does not support all of these at once, but *future* technologies like MRAM* might do

- System architects offer a *memory hierarchy*
  - small high-speed caches (expensive)
  - medium sized fast main memory (RAM)
  - flash memory
  - Giga bytes of slow, cheap disk storage
  - Terra bytes of very slow archive memory

*MRAM = Magnetic RAM*
Motivation

Memory Management (2)

Two main goals:

1. Manage memory efficiently
   - appropriate algorithms & data structures
   - program MMU (e.g. TLB)

2. Establish effective usability of memory
   - maximize usage of (main) memory
   - support low cache footprint
Why to bother about RAM?

Memory is large and cheap, and if not, we’ll use virtual memory.

However, reality tells us:

1. Many computers don’t use virtual memory at all
2. Modern programs tend to be memory greedy & virtual memory ≠ unlimited memory
3. Memory management has to be done anyway at
   - system level
   - application level
Architecture of RAM Management
Memory Management Module

Allocate

Release

Internal MMM Data Base

Note:
exists several related and/or unrelated memory managers
MMMDB reflects what parts are allocated and what parts are free

We must solve the following problems when handling memory requests:

- **Efficiently** select a fitting free memory block
  - As fast as possible

- Try to utilize the memory efficiently
  - Avoid *unusable* memory leaks

- Meet additional constraints
  - Avoid unbounded waiting in front of MMM
Smart System Architecture of a MMM

Main advantage of the above architecture?
Design Parameters
Orthogonal Parameters

- Sequence of allocate/release-operations
  - Size of memory blocks
  - Data structures
  - Fragmentation (*not design parameter but result of a design!!!*)
  - Allocation policy
  - Reunification of released blocks
  - ...?
MM Design Parameters (1a)

- Sequence of allocate/release-operations
  - ~FIFO = queue

![Diagram showing MM Design Parameters]

- Head of allocated buffers
- Head of free buffers
MM Design Parameters (1b)

- Sequence of allocate/release-operations
  - FIFO, LIFO = stack
MM Design Parameters (1c)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary = in general

Additionally we have to solve the problem of memory holes
MM Design Parameters (2)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
MM Design Parameters (2a)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary

- Size of memory blocks
  - Constant size = most buffering
MM Design Parameters (2b)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary

- Size of memory blocks
  - Constant size, multiple of fixed size
MM Design Parameters (2c)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary

- Size of memory blocks
  - Constant size, multiple size, fixed size = reservoir of frequently used blocks
MM Design Parameters (2d)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary

- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential size = buddy system
MM Design Parameters (2e)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of Memory blocks
  - Identical size, multiple size, fixed size, exponential size, arbitrary
Real Memory Management

MM Design Parameters (3a)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential size, arbitrary
- Data structures
  - Integrated within memory block(s) = mostly with larger blocks
MM Design Parameters (3b)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary

- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential size, arbitrary

- Data structures
  - Integrated, special memory block(s) = stack
Result of Design (4)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary

- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential size, arbitrary

- Data Structures
  - Integrated, special memory block(s)

- Fragmentation
  - With(out) in(ex)ternal fragmentation
MM Design Parameters (5a)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary

- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential size, arbitrary

- Data structures
  - Integrated, special memory block(s)

- Fragmentation
  - With(out) in(ex)ternal fragmentation

- Allocation policy
  - First-Fit = take the \textbf{first fitting block} within an \textbf{ordered set of free blocks}

\textit{How would you order the set of free blocks?}
MM Design Parameters (5b)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary

- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential sized, arbitrary

- Data structures
  - Integrated, special memory block(s)

- Fragmentation
  - With(out) in(ex)ternal fragmentation

- Allocation policy
  - First-, Next-Fit = take the next fitting block within an ordered set of free blocks
MM Design Parameters (5c)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary

- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential sized, arbitrary

- Data structures
  - Integrated, special memory block(s)

- Fragmentation
  - With(out) in(ex)ternal fragmentation

- Allocation policy
  - First-, Next-, BestFit = take the best fitting block within an ordered set of free blocks
MM Design Parameters (5d)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary

- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential sized, arbitrary

- Data structures
  - Integrated, special memory block(s)

- Fragmentation
  - With(out) in(ex)ternal fragmentation

- Allocation policy
  - First-, Next-, Best-, Worst-Fit = take the largest fitting block within an ordered set of free blocks
MM Design Parameters (5e)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary

- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential sized, arbitrary

- Data structures
  - Integrated, special memory block(s)

- Fragmentation
  - With(out) in(ex)ternal fragmentation

- Allocation policy
  - First-, Next-, Best-, Worst-, Nearest-Fit, = take the fitting block closest to the previous fitting block
MM Design Parameters (6a)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary

- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential size, arbitrary

- Data structures
  - Integrated, special memory block(s)

- Fragmentation
  - With(out) in(ex)ternal fragmentation

- Allocation policy
  - First-, Next-, Best-, Worst-, Nearest-Fit

- **Reunification** of released blocks
  - **Eager reunification** with neighbored free blocks (if any)
MM Design Parameters (6b)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential size, arbitrary
- Data structures
  - Integrated, special memory block(s)
- Fragmentation
  - With(out) in(ex)ternal fragmentation
- Allocation policy
  - First-, Next-, Best-, Worst-, ..., Nearest-Fit
- Reunification of Released blocks
  - Eager, lazy reunification = wait a while until you reunify
Summary of Designing MM

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary

- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential size, arbitrary

- Data structures
  - Integrated, special memory block(s)

- Fragmentation
  - With(out) internal or external fragmentation

- Allocation policy
  - First-, Next-, Best-, Worst-, ..., Nearest-Fit

- Reunification of Released blocks
  - Immediate, lazy reunification
Data Structures

- Bit Map (for fixed sized units, e.g. pages)
  - Extra data structure
  - Integrated

- Table/List (for arbitrary sized units, e.g. segments)
  - Extra data structure
  - Integrated
Bit Map Data Base

Extra Bit Map:

111010001110000_BITMAP

Overhead per bit map:
The smaller the memory units, the larger the bit map

Integrated Bit Map:
Table Data Base

Extra Tables (sorted according to addresses):

<table>
<thead>
<tr>
<th>Free Table:</th>
<th>Allocate Table:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Address</td>
</tr>
<tr>
<td>Length</td>
<td>Length</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

∃ other useful sorting criteria?
List oriented Data Base

Integrated List (sorted according to addresses):

Head of Allocated List

Head of Free List
Fragmentation

**Internal Fragmentation:**
- allocated memory can be larger than requested memory
- memory management **rounds up** requested memory to the next manageable memory block unit

**External Fragmentation:**
- Sum of total free memory space could satisfy a request, but free memory is **scattered** and is **not contiguous**

Real Memory Management
Allocation Policies

- First-Fit
- Next-Fit (Rotating First-Fit)
- Best-Fit
- Nearest-Fit

Analyze pros and cons of each of them
Reunification Policies

- **Eager reunification** (when releasing memory)

  ![Diagram of eager reunification]

- **Lazy reunification**

  ![Diagram of lazy reunification]
Additional Requirements

- With lazy reunification MM neighboring blocks can be free. However, none of these can satisfy the current memory request:

  ⇒ Garbage Collection\(^1\)

- With an arbitrary allocation scheme (e.g. pure segmentation) we might get external fragmentation with a lot of scattered free blocks (Swiss cheese)

  ⇒ Compaction

\(^1\)Garbage collection = releasing memory of no longer referenced objects
None of the 5 free blocks is large enough, but there are some neighboring free blocks.

- Free 1
- Free 2
- Free 3
- Free 4
- Free 5
- Free 6

New block to allocate

Can be reunified to become a larger free block
Garbage Collection (2)

Now 2 of the 3 free blocks are large enough

- Free 1'
- Free 2'
- Free 3'

New block to allocate

either

or
Observation: None of the 3 free blocks is large enough, but \exists enough free memory.

Idea: Move the allocated blocks towards a chosen memory boundary.
Compaction (2)

In textbooks garbage collection is often mixed up with compaction.

Free 1

New block to allocate
Examples: Memory Manager

- Ringbuffer
- Stack

Boundary Tag System ("Randkennzeichnungsverfahren")
- Operations in arbitrary order
- Arbitrary sized blocks
- Integrated management data structures
- Allocation according to xyz-Fit (best-fit is possible)
- External fragmentation
- Immediate reunification

Buddy System ("Halbierungsverfahren")
- Operations in arbitrary order
- Allocated blocks of $2^0$, $2^1$, $2^2$, $2^3$, ...
- Explicit management data structures
- Allocation according to "Best-Fit"
- Internal and external fragmentation
- Immediate (or lazy) reunification

- Linux Slab Allocator ("Stückchenzuteiler")

Very specific design parameters
Boundary Tag

allocated

Not Free

free

allocated

Not Allocated

free

allocated

pair of pointers for the free list
What to do?
Because length and status field are of fixed size ⇒ we can easily get the length of the block to be released. Furthermore we can look over both boundaries to get necessary information about the neighboring blocks.
Reunification (2)

Release this block!!

We can detect whether both (one or no) neighbor(s) are(is) free ⇒
Reunification (3)

Finally we have to adjust the new pointers in the free list and the resulting length field (in this case: \( l1' = l1 + l2 + l3 \))
Buddy System (1)

$2^{20}$ Byte = 1 MB free

Suppose a client requests 100 KB
Real Memory Management

Buddy System (2)

1 MB free

<table>
<thead>
<tr>
<th>512 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 KB</td>
</tr>
<tr>
<td>512 KB</td>
</tr>
<tr>
<td>128 KB</td>
</tr>
<tr>
<td>256 KB</td>
</tr>
<tr>
<td>512 KB</td>
</tr>
<tr>
<td>100 KB</td>
</tr>
<tr>
<td>128 KB</td>
</tr>
<tr>
<td>256 KB</td>
</tr>
<tr>
<td>512 KB</td>
</tr>
</tbody>
</table>

Dividing free memory until an appropriate free block

Internal fragmentation
Buddy System (3)

<table>
<thead>
<tr>
<th>Block Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 KB</td>
</tr>
<tr>
<td>128 KB</td>
</tr>
<tr>
<td>256 KB</td>
</tr>
<tr>
<td>512 KB</td>
</tr>
</tbody>
</table>

Suppose another 200 KB block is requested

1 MB free
Buddy System (4)

<table>
<thead>
<tr>
<th>100 KB</th>
<th>128 KB</th>
<th>256 KB</th>
<th>512 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 KB</td>
<td>128 KB</td>
<td>200 KB</td>
<td>512 KB</td>
</tr>
</tbody>
</table>

Suppose another 200 KB block is requested
Buddy System (5)

<table>
<thead>
<tr>
<th>Size</th>
<th>Used</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 KB</td>
<td>128 KB</td>
<td>256 KB</td>
</tr>
<tr>
<td>100 KB</td>
<td>128 KB</td>
<td>200 KB</td>
</tr>
<tr>
<td>100 KB</td>
<td>128 KB</td>
<td>200 KB</td>
</tr>
</tbody>
</table>

1 MB free

Suppose another 130 KB block is requested
Real Memory Management

Buddy System (6)

<table>
<thead>
<tr>
<th>100 KB</th>
<th>128 KB</th>
<th>256 KB</th>
<th>512 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 KB</td>
<td>128 KB</td>
<td>200 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>100 KB</td>
<td>128 KB</td>
<td>200 KB</td>
<td>200 KB</td>
</tr>
<tr>
<td></td>
<td>128 KB</td>
<td>200 KB</td>
<td>130 KB</td>
</tr>
</tbody>
</table>

Suppose we have to release the first 200 KB
Suppose we have to release the other 200 KB
<table>
<thead>
<tr>
<th>100 KB</th>
<th>128 KB</th>
<th>256 KB</th>
<th>512 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 KB</td>
<td>128 KB</td>
<td>200 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>100 KB</td>
<td>128 KB</td>
<td>200 KB</td>
<td>200 KB</td>
</tr>
<tr>
<td>100 KB</td>
<td>128 KB</td>
<td>200 KB</td>
<td>200 KB</td>
</tr>
<tr>
<td>100 KB</td>
<td>128 KB</td>
<td>256 KB</td>
<td>200 KB</td>
</tr>
<tr>
<td>100 KB</td>
<td>128 KB</td>
<td>256 KB</td>
<td>256 KB</td>
</tr>
</tbody>
</table>

Can we reunify these two blocks forming a 512 KB free block?

Not at all, they are not buddies! Only buddies belong with each other!
Data Structures for Buddy System

Array of heads pointing to free blocks of a certain size $2^k$
Allocating block of size $s$:

- Round up $s$ to next power of 2, say $2^i$  
  ($\Rightarrow$ internal fragmentation)
- Access head of the list for the $2^i$ pieces
- If list is not empty get first element of list
- If list is empty (recursively) do:
  - Access head of list for the $2^{i+1}$ pieces
  - If list isn’t empty get first element of list
  - cut element in halves
  - take lower half, insert upper half into list for $2^i$ pieces ...
Operations of Buddy System (2)

Releasing a block of size $2^i$

- Determine its buddy
- If buddy is (partly) allocated, insert the block to be released into the list of pieces of size $2^i$
- If buddy is free reunify both buddies

Question:
How can we efficiently determine the appropriate buddy?
Determining the Buddy

Address calculation for a piece of size $2^{12}$

**Question:** What’s the address of its buddy?

**Remark:** Other pieces of size $2^{12}$ differ only in the leading address bits.
Reunification of Buddies

Each block of size $2^i$ of buddy system contains:

- State of the block (allocated or free)
- Length of the block

Is there an alternative?

- Analyze the implementation of Linux
- Compare both implementations
Scenarios for Reunifications

- **reunification buddy free**
  - xyz000 free, 2
  - xyz100 free, 2

- **no reunification buddy allocated**
  - xyz000 free, 2
  - xyz100 allocated, 2

- **no reunification buddy partly allocated**
  - xyz000 free, 2
  - xyz100 free, 1
  - allocated, 1
Summary of Buddy System

On average, the internal fragmentation is about 25%
  - each memory block is at least 50% occupied

Works efficiently if the size $M$ of RAM is a power of 2
Slab Allocating (in Linux)

- Inside a kernel there are a few data types, e.g.
  - TCB
  - LAS descriptor
  - Page table
  - File handle ...

- Kernel tends to request these data types over and over again, e.g. to establish a new thread/task

- Slab allocation tries to support reusage of previously used data types ⇒ install object type caches

- Following slides:
  - Steffen Wolfer, Proseminar WS 2003 Linux Internals
  - see also: Sven Krohlas, "WS 2004"
New approach (Sun Microsystems, 1994)

- regard a memory area as an „object container“
- collect objects of the same type in specific „logical kernel caches“
- divide caches into slabs
- slab is part of a specific cache
- slab size = s*page frames, s ≥ 1
Slab Allocating (2)

- Reuse already initialized objects
- Simplify complex allocation
- Take page frames from the buddy system
  - Give back only if buddy system needs them
Slab Allocating (3)

- Cache 1
- Cache 2
- Cache 3

- Page frame
- Slabs

- Black: Allocated object
- White: Free object
### Slab Allocating (4)

Table for the caches:

<table>
<thead>
<tr>
<th>Cache Descriptor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>slabs_full</td>
<td>Doubled linked lists with slab descriptors for</td>
</tr>
<tr>
<td></td>
<td>- full</td>
</tr>
<tr>
<td></td>
<td>- partially full</td>
</tr>
<tr>
<td></td>
<td>- empty slabs</td>
</tr>
<tr>
<td>slabs_partial</td>
<td>Pointer to next cache descriptor</td>
</tr>
<tr>
<td>slabs_free</td>
<td>Number of objects per slab, size of object, flags, ...</td>
</tr>
<tr>
<td>next</td>
<td></td>
</tr>
<tr>
<td>num</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
### Slab Allocating (5)

**Table for Slabs:**

<table>
<thead>
<tr>
<th>Slab Descriptor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inuse</td>
<td>number of currently allocated objects</td>
</tr>
<tr>
<td>s_mem</td>
<td>pointer to 1. Object</td>
</tr>
<tr>
<td>free</td>
<td>pointer to 1. free Object</td>
</tr>
<tr>
<td>list</td>
<td>pointer to list of slab descriptors</td>
</tr>
</tbody>
</table>
Allocate memory: `kmem_cache_alloc()`
Slab Allocating (7)

Release memory: kmem_cache_free()
Slab Allocating (8)

Free slabs: `kmem_cache_destroy()`
Summary of Slab Allocation

- Who is using slab allocation?
- What kernel data types are mapped to slabs?
- How much space in total is managed by slabs?
- Do slabs have to be mapped to contiguous memory?
- ...