Dynamic memory

Herbert Bos
Vrije Universiteit Amsterdam
http://www.cs.vu.nl/~herbertb
Important stuff

• Memory is the new(?) bottleneck
• Management must be fast+space-efficient
Roadmap

- Memory management in Linux
- Memory for efficient I/O:
  - A slow way of doing it
  - A Linux way of doing it
  - A faster way of doing it: Fbufs & IO-Lite
  - A different way of doing it: Beltway Buffers & pipesfs
Linux kernel memory management

- MM in kernel is harder than in uspace
- x86 as running example
- Physical pages are basic units of MM
- Every page is represented by: struct page
  - ulong flags ← dirty, locked
  - atomic_t count ← how many refs to page
  - struct list_head list
  - struct AS *mapping ← address space associated to the page
  - ulong index
  - struct list_head lru
  - (pte)
  - (private)
  - void * virtual ← virtual address (could be null)

- page frame descriptor array is called mem_map
Zones

- zone DMA
  - pages capable of undergoing DMA
- zone normal
  - regularly mapped pages
- zone highmem
  - pages not permanently mapped in kernel's AS
  - (in)famous 896MB boundary
zone pools

- different pools for different zones
- if we need DMA-able memory, use zone DMA pool
- no hard requirement (e.g., normal page could also come from zone DMA or highmem)
say we have 2GB of memory
getting pages

• **low-level** mechanism to get pages:
  – `struct page *alloc_pages (uint gfp_mask, uint order)`
  or (if you are only interested in the logical address):
  – `ulong __get_free_pages (uint gfp_mask, uint order)`

• allocates $2^{\text{order}}$ pages
• GFP mask encapsulates a bunch of things
  – zone
  – behaviour of allocator (can it block? can it start disk or FS I/O? etc)
• Examples: GFP_KERNEL, GFP_ATOMIC, GFP_DMA
higher level

- **alloc_pages** (and friends) are low-level
  - work on pages
  - are good when you need a set of physically contiguous pages
- for byte sized allocations
  - `kmalloc (size, gfp_mask)`: physically contiguous
  - `vmalloc (size, gfp_mask)`: virtually contiguous
hold on…

* this is actually quite complicated
* we must track free memory, handle external fragmentation, etc.
  - first fit? best fit? worst fit?
  - defragmentation?
* and it has to be fast
* so how does allocation really work?
buddy allocation

(Harry Markowitz in 1963)
buddy allocation

Fast, simple allocation for blocks of $2^n$ bytes

```c
void *allocate (k bytes)
- raise allocation to nearest $s = 2^n$
- search free list for appropriate size
  - Represent free list with bitmap
  - Recursively divide larger free
  - blocks until find block of size $s$
  - “Buddy” block remains free
- mark corresponding bits as allocated
```

```c
free(ptr)
- mark bits as free
- recursively coalesce block with buddy, if buddy is free
  - May coalesce lazily (later, in background) to avoid overhead
```
Buddy allocation

- Program A requests memory 34K..64K in size
- Program B requests memory 66K..128K in size
- Program C requests memory 35K..64K in size
- Program D requests memory 67K..128K in size

- Program C releases its memory
- Program A releases its memory
- Program B releases its memory
- Program D releases its memory

<table>
<thead>
<tr>
<th>t</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 0</td>
<td>1024K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t = 1</td>
<td>A-64K</td>
<td>64K</td>
<td>128K</td>
<td>256K</td>
<td>512K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t = 2</td>
<td>A-64K</td>
<td>64K</td>
<td>B-128K</td>
<td>256K</td>
<td>512K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t = 3</td>
<td>A-64K</td>
<td>C-64K</td>
<td>B-128K</td>
<td>256K</td>
<td>512K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t = 4</td>
<td>A-64K</td>
<td>C-64K</td>
<td>B-128K</td>
<td>D-128K</td>
<td>128K</td>
<td>512K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t = 5</td>
<td>A-64K</td>
<td>64K</td>
<td>B-128K</td>
<td>D-128K</td>
<td>128K</td>
<td>512K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t = 6</td>
<td>128K</td>
<td>B-128K</td>
<td>D-128K</td>
<td>128K</td>
<td>512K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t = 7</td>
<td>256K</td>
<td>D-128K</td>
<td>128K</td>
<td>512K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t = 8</td>
<td>1024K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Buddy allocation

● works really well

● but has a serious disadvantage...
Buddy allocation

- suffers from internal fragmentation that can be quite significant
- also: consider frequent allocation and de-allocation of same-size chunk
slab allocator

- many allocations and de-allocations are for same-size objects
- we don't want the overhead
- so: cache the allocation
Slab Allocation

- Slab – One or more phys. contiguous pages
- Cache – One or more slabs
  - Single cache for each unique kernel data structure (process descriptor, inode, semaphore, ...)
- Object – instance of a kernel data structure
Slab allocation

- Set of objects pre-allocated
- Marked as free
- When needed assign a free one and mark as used
- No free ones available?
  - Allocate a new slab
  - Slab states: full, empty, partial
  - Fill partial slab first
- Advantage
  - No fragmentation
  - Memory requests are satisfied quickly
slab descriptors

• cache represented by `kmem_cache_t` struct
  - three lists:
    • `slabs_full`
    • `slabs_empty`
    • `slabs_partial`

• slab descriptor, `struct slab`, represents each slab

```c
struct slab {
    struct list_head list;
    ulong colour;
    void *s_mem;
    uint in_use;
    kmem_bufctl_t free;
};
```

← it is a list!
← first object in slab
← number of allocated objects
← first free object (if any)
slab descriptors

- slab descriptor are allocated
  - inside slab if total slab size small
  - outside slab in a general cache

```c
struct slab {
    struct list_head list;
    ulong colour;
    void *s_mem;
    uint in_use;
    kmem_bufctl_t free;
};
```

← it is a list!
← first object in slab
← number of allocated objects
← first free object (if any)
General and specific caches

- specific caches can be created to store objects of certain sizes
- general caches contain 13 geometrically distributed caches
  - from 32 to 131,072 bytes increased in power of 2
- `kmalloc` is built on top of slab layer using a family of general purpose caches
SLAB is great

- but are there any disadvantages?
SL*B

- **SLAB**
  - traditional UNIX with SLAB emulation
  - for embedded systems
  - slower

- **SLOB**
  - newer way of doing SLAB
  - removes some problems (e.g., alignment issues due to descriptors in slab)
when would you use

- slab?
- buddy?
the process address space
Linux process address space

- flat 32b or 64b address space
- memory areas: intervals of legal addresses
- processes can dynamically add and remove memory areas
- Memory regions have an initial logical address and a length, which is a multiple of 4K
- SEGV: process accesses invalid memory area
process address spaces

- Typical situations in which a process gets new memory regions
  - growing its stack
  - creating shared memory (shmat())
  - expanding its heap (malloc())
  - Creating a new process (fork())
  - loading an entirely new program (execve())
  - memory mapping a file (mmap())
process memory descriptor

- All info related to the process address space is included in the memory descriptor (mm_struct) referenced by the mm field of the process descriptor.
- Memory descriptors are allocated from the slab allocator cache using mm_alloc().
- Some examples of included information:
  - A pointer to the top level of the page table, the Page Global Directory.
  - Number of page frames allocated to the process.
  - Process’ address space size in pages.
  - Number of locked pages.
  - Number of processes sharing the same mm_struct, i.e., threads.
  - List of memory areas (struct vm_area struct *mmap).
memory areas

- Linux represents a memory region with **vm_area struct**
  - Contains a reference to the memory descriptor that owns the region (**vm_mm** field),
  - the start (**vm_start** field) and end (**vm_end** field) of the interval
- Memory areas never overlap
- Kernel tries to merge contiguous regions (if access rights match)
- All regions are maintained on a simple list (**vm_next** field) in ascending order by address
global view
Example:
```
cat /proc/`pidof bc`/maps
```

```
08048000-08058000 r-xp 00000000 08:01 15024544 /usr/bin/bc
08058000-08059000 rw-p 00010000 08:01 15024544 /usr/bin/bc
08059000-0807a000 rw-p 08059000 00:00 0 [heap]
b7cfd000-b7d3c000 r--p 00000000 08:01 15089747 /usr/lib/locale/en_AU.utf8/LC_CTYPE
b7d3c000-b7d3d000 rw-p b7d3c000 00:00 0
b7d3d000-b7d3f000 r-xp 00000000 08:01 15501619 /lib/tls/i686/cmov/libdl-2.7.so
b7d3f000-b7d41000 rw-p 00010000 08:01 15501619 /lib/tls/i686/cmov/libdl-2.7.so
b7d41000-b7e8a000 r-xp 00000000 08:01 15501532 /lib/tls/i686/cmov/libc-2.7.so
b7e8a000-b7e8b000 r--p 00149000 08:01 15501532 /lib/tls/i686/cmov/libc-2.7.so
b7e8b000-b7e8d000 rw-p 0014a000 08:01 15501532 /lib/tls/i686/cmov/libc-2.7.so
b7e8d000-b7e91000 rw-p b7e8d000 00:00 0
b7e91000-b7ebe000 r-xp 00000000 08:01 15466565 /lib/libncurses.so.5.6
b7ebe000-b7ec1000 rw-p 0002c000 08:01 15466565 /lib/libncurses.so.5.6
b7ec1000-b7eed000 r-xp 00000000 08:01 15466577 /lib/libreadline.so.5.2
b7eed000-b7ef1000 rw-p 0002c000 08:01 15466577 /lib/libreadline.so.5.2
b7ef1000-b7ef2000 rw-p b7ef1000 00:00 0
b7eff000-b7f06000 r--s 00000000 08:01 15042231 /usr/lib/gconv/gconv-modules.cache
b7f06000-b7f09000 rw-p b7f06000 00:00 0
b7f09000-b7f0a000 r-xp b7f09000 00:00 0 [vdso]
b7f0a000-b7f24000 r-xp 00000000 08:01 15466534 /lib/ld-2.7.so
b7f24000-b7f26000 rw-p 00019000 08:01 15466534 /lib/ld-2.7.so
bf99f000-bf9b4000 rw-p bffe0000 00:00 0 [stack]
```
creating an address interval

- **mmap / do_mmap**
  - creates 'new' address interval
  - except: if adjacent, it tries to merge with existing one (requires permissions to be the same)
  - do_mmap() used in kernel
  - exported to user space using mmap() syscall

- **munmap() / do_munmap()**
the page cache
Linux has one primary disk cache: page cache

It is (drum roll)... *a cache of pages*

idea:
- store in phys. memory data accessed from disk
- next access will be from memory
- use as many available pages as possible

* caches *any* page-based object (e.g., files and file mappings)

* single read-ahead to speed up seq. access*
use as much as possible for PC

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemTotal:</td>
<td>3950112 kB</td>
</tr>
<tr>
<td>MemFree:</td>
<td>622560 kB</td>
</tr>
<tr>
<td>Buffers:</td>
<td>78048 kB</td>
</tr>
<tr>
<td>Cached:</td>
<td>2901484 kB</td>
</tr>
<tr>
<td>SwapCached:</td>
<td>0 kB</td>
</tr>
<tr>
<td>Active:</td>
<td>3108012 kB</td>
</tr>
<tr>
<td>Inactive:</td>
<td>55296 kB</td>
</tr>
<tr>
<td>HighTotal:</td>
<td>0 kB</td>
</tr>
<tr>
<td>HighFree:</td>
<td>0 kB</td>
</tr>
<tr>
<td>LowTotal:</td>
<td>3950112 kB</td>
</tr>
<tr>
<td>LowFree:</td>
<td>622560 kB</td>
</tr>
<tr>
<td>SwapTotal:</td>
<td>4198272 kB</td>
</tr>
<tr>
<td>SwapFree:</td>
<td>4198244 kB</td>
</tr>
<tr>
<td>Dirty:</td>
<td>416 kB</td>
</tr>
<tr>
<td>Writeback:</td>
<td>0 kB</td>
</tr>
<tr>
<td>Mapped:</td>
<td>999852 kB</td>
</tr>
<tr>
<td>Slab:</td>
<td>57104 kB</td>
</tr>
<tr>
<td>Committed_AS:</td>
<td>3340368 kB</td>
</tr>
<tr>
<td>PageTables:</td>
<td>6672 kB</td>
</tr>
<tr>
<td>VmallocTotal:</td>
<td>536870911 kB</td>
</tr>
<tr>
<td>VmallocUsed:</td>
<td>35300 kB</td>
</tr>
<tr>
<td>VmallocChunk:</td>
<td>536835611 kB</td>
</tr>
<tr>
<td>HugePages_Total:</td>
<td>0</td>
</tr>
<tr>
<td>HugePages_Free:</td>
<td>0</td>
</tr>
<tr>
<td>Hugepagesize:</td>
<td>2048 kB</td>
</tr>
</tbody>
</table>

/proc/meminfo for system with 4G of memory
looking up a page is fairly hard

- because multiple non-contiguous disk blocks may be in a page
- you cannot just index page cache with device name and block number
struct address_space {  
    struct inode  
struct radix_tree_root  
    *host;  
    page_tree;  
    rwlock_t  
    tree_lock;  
    unsigned int  
    i_mmap_writable;  
    struct prio_tree_root  
    i_mmap;  
    struct list_head  
    i_mmap_nonlinear;  
    spinlock_t  
    /* protect tree, count, list */  
    unsigned int  
    truncate_count;  
    unsigned long  
    nrpages;  
    writeback_index;  
    spinlock_t  
const struct address_space_operations *a_ops;  
    /* methods */  
    unsigned long  
    flags;  
    /* error bits/gfp mask */  
    struct backing_dev_info  
    *backing_dev_info;  
    struct list_head  
    *assoc_mapping;  
    struct address_space  
    *private_list;  
    /* ditto */  
    spinlock_t  
    *private_list;  
};
reading a page

- lookup: given `address_space` plus `offset` pair

→ use the radix tree for efficient lookup
Look-up of \{00110100101 1100\} in a PATRICIA Trie
radix tree in Linux

Figure 1: 8-bit radix tree
tagged radix tree in Linux

Figure 2: 8-bit radix tree with 2 bitmap indices
old way: the page hash table
dirty pages must be flushed

- pdflush daemon is responsible for this
- when to flush?
  - when free memory shrinks below threshold
  - when dirty data grows older than threshold
- >1 pdflush threads
  - why?
Summary part I

- dynamic memory
  - buddy
  - slab
- process address space
- page cache