LINUX MEMORY MANAGEMENT

MANISH SHRIVASTAVA

PAGE FRAME MANAGEMENT

- Page frames are 4KB in Linux.
- The kernel must keep track of the current status of each frame.
  - Are page frames allocated or free?
  - If allocated, do they contain process or kernel pages?
  - Linux maintains an array of page frame descriptors (one for each frame) of type **`struct page`**.
- NOTE: see the **`mm`** directory for memory management, especially **`page_alloc.c`**.

PAGE FRAME DESCRIPTORS

- Each descriptor has several fields, including:
  - **`count`** - equals 0 if frame is free, >0 otherwise.
  - **`flags`** - an array of 32 bits for frame status.
    - Example flag values:
      - **`PG_locked`** - page cannot be swapped out.
      - **`PG_reserved`** - page frame reserved for kernel code or unusable.
      - **`PG_Slab`** - included in a slab (more later).

THE `MEM_MAP` ARRAY

- All page frame descriptors are stored in the **`mem_map`** array.
- Descriptors are less than 64 bytes. Therefore, **`mem_map`** requires about 4 page frames for each MB of RAM.
- The **`MAP_NR`** macro computes the number of the page frame whose address is passed as a parameter:
  - **`#define MAP_NR(addr) (__pa(addr) >> PAGE_SHIFT)`**
  - **`__pa`** macro converts logical address to physical.

REQUESTING PAGE FRAMES

- Main routine for requesting page frames is: **`__get_free_pages(gfp_mask,order)`**
- Request 2^{order} contiguous page frames.
- **`gfp_mask`** specifies how to look for free frames. It is a bitwise OR of several flags, including:
  - **`__GFP_WAIT`** – Allows kernel to discard page frame contents to satisfy request.
  - **`__GFP_IO`** – Allows kernel to write pages to disk to free page frames for new request.
  - **`__GFP_HIGH/MED/LOW`** – Request priority. Usually user requests are low priority while kernel requests are higher.
    - eg., **`GFP_ATOMIC=__GFP_HIGH;`** **`GFP_USER=__GFP_WAIT=1|__GFP_IO=1|__GFP_LOW;`**

RELEASING PAGE FRAMES

- Main routine for freeing pages is: **`Free_pages(addr,order)`**
  - Check frame at physical address **`addr`**.
    - If not reserved, decrement descriptor's **`count`** field.
      - If **`count==0`**, free 2^{order} contiguous frames.
        - **`free_pages_ok()`** inserts page frame descriptor of 1st free page in list of free page frames.

EXTERNAL FRAGMENTATION

- *External fragmentation* is a problem when small blocks of free page frames are scattered between allocated page frames.
  - Becomes impossible to allocate large blocks of *contiguous* page frames.
- Solution:
  - Use paging h/w to group non-contiguous page frames into contiguous linear (virtual) addresses.
  - Track free blocks of contiguous frames & attempt to avoid splitting *large* free blocks to satisfy requests.
    - DMA controllers, which bypass the paging hardware, sometimes need contiguous page frames for buffers.
    - Contiguous frame allocation can leave page tables unchanged – TLB contents don't need to be flushed, so memory access times are reduced.

THE BUDDY SYSTEM

- Free page frames are grouped into lists of blocks containing 2^n contiguous page frames.
  - Linux has 10 lists of 1,2,4,…,512 contiguous page frames.
  - Physical address of 1st frame in a block is a multiple of the group size e.g., multiple of $16*2^{12}$ for a 16-page-frame block.

BUDDY ALLOCATION

- Example: Need to allocate 65 contiguous page frames.
  - Look in list of free 128-page-frame blocks.
  - If free block exists, allocate it, else look in next highest order list (here, 256-page-frame blocks).
  - If first free block is in 256-page-frame list, allocate a 128-page-frame block and put remaining 128-page-frame block in lower order list.
  - If first free block is in 512-page-frame list, allocate a 128-page-frame block and split remaining 384 page frames into 2 blocks of 256 and 128 page frames. These blocks are allocated to the corresponding free lists.

- Question: What is the worst-case *internal* fragmentation?

BUDDY DE-ALLOCATION

- When blocks of page frames are released the kernel tries to merge pairs of "buddy" blocks of size ***b*** into blocks of size ***2b***.
- Two blocks are buddies if:
  - They have equal size ***b***.
  - They are located at contiguous physical addresses.
  - The address of the first page frame in the first block is aligned on a multiple of $2b*2^{12}$.
- The process repeats by attempting to merge buddies of size ***2b***, ***4b***, ***8b*** etc…

BUDDY DATA STRUCTURES

- An array of 10* elements (one for each group size) of type **`free_area_struct`**.
  - **`free_area[0]`** points to array for non-ISA DMA buddy system.
  - **`free_area[1]`** points to array for ISA DMA buddy system.
  - Linux 2.4.x has a 3rd buddy system for high physical memory! Makes dynamic memory mgt fast!
- A group of binary arrays (**`bitmaps`**) for each group size in each buddy system.

EXAMPLE BUDDY MEMORY MGT

- 128MB of RAM for non-ISA DMA.
  - `free_area[0][k]` consists of ***n*** bits, one for each pair of blocks of size $\boldsymbol{2^K}$ page frames.
    - Each bit in a bitmap is 0 if a pair of buddy blocks are ***both*** free or allocated, else 1.
  - `free_area[0][0]` consists of ***16384*** bits, one for each pair of the ***32768*** page frames.
  - `free_area[0][9]` consists of ***32*** bits, one for each pair of blocks of ***512*** contiguous page frames.

MEMORY AREA MANAGEMENT

- *Memory areas* are contiguous physical addresses of arbitrary length e.g., from a few bytes to several KBs.
- Could use a buddy system for allocating memory in blocks of size 2^K within pages, and then another for allocating blocks in power-of-2 multiples of pages.
- Linux uses a ***slab allocator*** for arbitrary memory areas.
  - Memory is viewed as a collection of related objects.
    - Objects are cached when released so that they can be allocated quickly for new requests.
    - Memory objects of the same type are repeatedly used e.g., process descriptors for new/terminating processes.
  - Can have memory allocator for commonly used objects of known size and buddy system for other cases.

LINUX SLAB ALLOCATOR

- Objects of same type are grouped into ***caches***.
  - Can view caches by reading /proc/slabinfo.
  - Caches are divided into ***slabs***, with >=1 page frames containing both allocated & free objects.
  - See mm/slab.c for more details.
- A newly created cache does not contain any slab or free objects.
- Slabs are assigned to a cache when:
  - A request for allocating a new object occurs.
  - Cache does not already have a free object.
    - Buddy system is invoked to get new pages for a slab.

SLAB DE-ALLOCATION

- A slab is released only if the following conditions hold:
  - Buddy system is unable to satisfy a new request for a group of page frames.
  - Slab is empty – all objects in it are free.
- ***Destructor methods**** on objects in an empty slab are invoked.
- Contiguous page frames in slab are returned to buddy system.

- NOTE: when objects are created, they also have corresponding ***constructor methods*** (possibly NULL valued) that can initialize objects.

OBJECT ALIGNMENT

- Memory accesses usually faster if objects are word aligned e.g., on 4-byte boundaries with 32-bit Intel architecture.
- Linux does not align objects if space is continually wasted as a result.
  - Linux rounds up object size to a factor of cache size if internal fragmentation does not exceed a threshold.
    - Idea is to trade fragmentation for aligning object of larger size.

GENERAL PURPOSE OBJECTS

- Linux maintains a list of general purpose objects, having geometrically distributed sizes from 32 to 131072 bytes.
- These objects are allocated using: **`void *kmalloc(size_t size, int flags);`**
  - **`flags`** is same as for **`get_free_pages()`**.
    - e.g. **`GFP_ATOMIC`**, **`GFP_KERNEL`**.
- Gen purpose objects are freed using **`kfree()`**.