Lab 2 Memory Management

YANG Weikun 1100012442

25th September 2013, 11:45 UTC+0800

- update on challenges 26th Sept. 2013 00:05 +0800

Preface

Jos git repo was first switched to branch lab2 (using `git checkout -b lab2 origin/lab2`), then the changes performed in lab1 was merged by `git merge lab1`.

Ex.1

physical page allocator

This exercise requires a *physical* page allocator, which needs to keep track of all pages, and more importantly, free pages. Physical pages are referenced using `struct PageInfo`, and free pages are linked together. Before any page allocation to occur, we must reserve space in memory to hold all the `PageInfo` structures, using `boot_alloc()`:

```
boot_alloc(uint32_t n)
{
    static char *nextfree;
    char *result;
    if (!nextfree) {
        // end is marked by the linker to indicate the end of memory used by
        // the kernel, after which we may use for bootstrap allocation.
        extern char end[];
        nextfree = ROUNDUP((char *) end, PGSIZE);
    }
    result = nextfree;
    // memory allocated mudt be page-aligned
    nextfree = ROUNDUP(nextfree+n, PGSIZE);
    return result;
}
```

Then, inside `mem_init()`, we allocate a page (4K bytes) for our new page directory `kern_pgdir` (later used in paging), and map the *virtual* address at `UVPT` to *physical* address of `kern_pgdir`. What happened here?

```
kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
// note by wky: each entry in page directory is 4-bytes, so we
// have 1024 entries in a 4K page.
memset(kern_pgdir, 0, PGSIZE);
kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
```

Explained: page directory contains page directory entries (PDEs), which contain the ***physical*** address of the corresponding page table and marker bits. The last statement, sets the **page table** that maps *virtual* address `UVPT` , to `kern_pgdir` itself, with user read permission. So whenever we address somewhere in `[UVTP, UVTP+4MB)` , the first step in paging lead us to a page table at `kernpg_dir` , then our page table index becomes page directory index, so reading 4 bytes from `UVPT+4*n` will give us the *n*-th **PDE**, which stores *physical* address and permissions of the corresponding page table (that contains mapping in the range `[n*4MB,(n+1)*4MB)`).

Then we allocate space for `PageInfo` , and initialise them:

```
pages = (struct PageInfo*)boot_alloc(npages * sizeof(struct PageInfo));
page_init();
```

In `page_init()` , we must

1. Mark page 0 (memory [0, 4K)) as in use, to leave the old interrupt vector table and bios structures alone.
2. Mark memory `[PGSIZE, IOPHYSMEM)` free, link them on the list.
3. The IO hole occupies `[IOPHYSMEM, EXTPHYSMEM)` which is [640K, 1M). The kernel itself plus page directory and `PageInfo` s occupy the range from `EXPHYSMEM` to physical address of `boot_alloc(0)` .
4. Mark the rest free, and link them to the list starting from `page_free_list` . The list is actually reverse ordered, if you count from lower memory.

Next, we implement `page_alloc()` and `page_free()` , just easy pointers handling.

Now we should be able to pass `check_page_alloc()` .

Ex.2

How a x86 processor translates virtual to physical?

After we have entered protected mode, enabled paging, our dear CPU would provide a two stage translation for us.

1. **Virtual -> Linear** Each of our 4-byte pointer are actually just an offset, to a 'selector', like in old times (8086 segment<<4 + offset). But the GDT we installed at start-up

effectively sets all segment selectors to start from `0x0`, and extends to `0xffffffff`. So currently in `jos`, segmentation has no effect. Our virtual and linear addresses are equivalent.

2. **Linear -> Physical** Linear addresses are then mapped to physical addresses, here using two stage 'Paging' mechanism. Explained in Chapter 5, section 2 of Intel 80386 Reference Manual, a linear address is plot in 3 parts, page-directory-index (10 bits, MSB), page-table-index (10 bits in middle), and offset within page (12 bits, LSB). The processors MMU's mechanism in detail:
    - Page directory's physical address is stored in CR3, a special register. Take 4 bytes from CR3 + 4 * page-directory-index, which is a page-directory-entry holding the physical address of the page table and permission bits.
    - Find the page table, use the page-table-index to get the page-table-entry that holds the physical address of the actual page, and use the 12 bit offset to address the wanted memory.

What a mess! And we must access main memory 3 times, to load a couple of bytes to our register, so the CPUs really rely on Translation-Look-aside-Buffers for better performance.

Ex.3

Memory Inspection using QEMU monitor commands

When the kernel enters monitor loop after initialisation, typing `info pg` in QEMU command console shows mapping details (ordered by physical page number, not complete):

```
VPN range        Entry        Flags        PPN range
...
[f0000-f03ff]  PDE[3c0]     ----A--UWP
    [f0000-f0000]  PTE[000]     --------WP 00000
    [f0001-f009f]  PTE[001-09f] ---DA---WP 00001-0009f
    ...
    [f0100-f0101]  PTE[100-101] ----A---WP 00100-00101
    [f0102-f0102]  PTE[102]     --------WP 00102
    ...
    [f03be-f03ff]  PTE[3be-3ff] --------WP 003be-003ff
...
```

`info registers` shows

```
...
GDT=     00007c4c 00000017
IDT=     00000000 000003ff
CR0=80050033 CR2=00000000 CR3=00123000 CR4=00000000
...
```

Just another word: MIT's description on entering QEMU's monitor '*To enter the monitor, press Ctrl-a c in the terminal running QEMU*'. I thought *Ctrl-a c* meant *press `ctrl` and hold, press `a` and hold, then press `c`*. But actually it is ***press `ctrl` and hold, press `a`, release, then press `c`***.

Q&A

code:

```
mystery_t x;
char* value = return_a_pointer();
*value = 10;
x = (mystery_t) value;
```

Here `mystery_t` should be `uintptr_t`, since `char* value` is a virtual address pointer.

Ex.4

Page Table Management

`pgdir_walk()` is reference in many other functions so it is implemented first. It must perform exactly as hardware MMU, and create new mappings when necessary. I used `goto` to avoid complex control structures here and many other functions.

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    pte_t* pgtbl;   // points to a page table
    pde_t* pde;     // points to a page directory entry
    struct PageInfo* newpg;

    pde = &pgdir[PDX(va)];
    pgtbl = (pte_t*)PTE_ADDR(*pde);

    if (*pde & PTE_P)
        goto get_page_entry;    // if the pde is in use already
    if (!create)
        return NULL;
    newpg = page_alloc(ALLOC_ZERO); // allocate new page for page table
    if (!newpg)
        return NULL;
    newpg->pp_ref ++;
    pgtbl = (pte_t*)page2pa(newpg);
    *pde = (uintptr_t)pgtbl | PTE_P | PTE_W | PTE_U;

get_page_entry:
    pgtbl = KADDR((uintptr_t)pgtbl);
    return (pte_t*)(&pgtbl[PTX(va)]);   // return an entry in page table
}
```

The next one is `boot_map_region()`. For every page in range `[va, va+size)`, I used `pgdir_walk()` to acquire the page-table-entry, and fill it with appropriate physical address and permission. **Note** that `va+size` can cause 32-bit integer overflow, when `va+size` is the end of 4GB.

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int
perm)
{
    uintptr_t end_va = va + size;
    pte_t *entry;
    for (; va != end_va; va += PGSIZE, pa += PGSIZE){
        entry = pgdir_walk(pgdir, (void*)va, true);
        if (entry == NULL)
            panic("Page table allocation failed.");
        *entry = pa | perm | PTE_P;
    }
}
```

Next, `page_lookup()`. Again I used `pgdir_walk()` to find the corresponding PTE, and convert the physical address to `PageInfo*`.

```
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    pte_t * entry = pgdir_walk(pgdir, va, false);
    if (pte_store)
        *pte_store = entry;
    if (!entry)
        return NULL;
    return pa2page(PTE_ADDR(*entry));
}
```

Moving on to `page_remove()`. Use `page_lookup()` to acquire both PTE and the actual page. Easy one.

```
void
page_remove(pde_t *pgdir, void *va)
{
    struct PageInfo *page;
    pte_t * entry;
    page = page_lookup(pgdir, va, &entry);
    if (!(page && (*entry & PTE_P)))
        return;
    page_decref(page);
    *entry = 0;
    tlb_invalidate(pgdir, va);
}
```

At last, we must deal with `page_insert`. The 'corner-case hint' said '*there's an elegant way to handle everything in one code path*', on mapping the same page to the same address just to change permission. But to the extent of my now knowledge and experience, I found no '*one code path*' to solve all cases. So i tried to detect the corner-case, and handle reference count manually to avoid the page being freed (when ref-cnt reaches 0). Here's my code for clarity:

```
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
    pte_t *entry = pgdir_walk(pgdir, va, true);
    if (!entry)
        return -E_NO_MEM;
    if (!(*entry & PTE_P))
        goto map_new_page;
    if (PTE_ADDR(*entry) == page2pa(pp)){
        tlb_invalidate(pgdir, va);
        pp->pp_ref --;
    }else page_remove(pgdir, va);
map_new_page:
    *entry = page2pa(pp) | perm | PTE_P;
    pp->pp_ref ++;
    return 0;
}
```

That completes page table management.

Ex.5

The Kernel Address Space

Before moving on, we must setup correct mappings, for those pages marked used in `page_init()` (they will not be managed by function implemented in Ex.4, so we are doing the mapping manually). In `mem_init()` after `check_page()`, 3 mappings were called:

1. Map `PageInfo` structures to virtual address `UPAGES`, with user read permission.
2. Map `[KSTACKTOP-KSTKSIZE, KSTACKTOP)` to the actual stack, with kernel write permission, and leave `[KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE)` unmapped so a page fault will be thrown if the kernel stack overflows.
3. Map `KERNBASE` to end of 4GB, to physical memory starting from `0x0`. Doing this will limit our OS to 256MB of RAM, but never mind.

And the corresponding code:

```
boot_map_region(kern_pgdir, UPAGES,
    ROUNDUP(npages * sizeof(struct PageInfo), PGSIZE),
    PADDR(pages), PTE_U | PTE_P);

boot_map_region(kern_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE,
    PADDR(bootstack), PTE_W);

boot_map_region(kern_pgdir, KERNBASE, -KERNBASE, 0, PTE_W);
```

Now my version of jos lab2 passes all checks.

Q&A

1. page mappings.
2. protection?
3. JOS supports maximum 256MB of RAM.
4. If we were to manage 4GB RAM, using `jos` 's style, we must:
    - allocate `PageInfo` s, 4GB-total-RAM / 4KB-per-page * 8bytes-per-PageInfo = 8MB
    - allocate space for kernel page directory, 4KB
    - allocate space for kernel page tables. When all of the physical pages are used, we need 4GB-RAM / 4K-per-page / 1K-PTE-per-page * 4K-per-page = 4MB. We need 4MB for page tables.

    Therefore, we need a bit more than 12MB to manage all physical pages in 4GB-RAM.
5. `EIP` went above `KERNBASE` after the `imp *%eax` instruction. We can still run at low `EIP` before the `jmp` , because that the page directory and page tables in `entrypgdir.S` maps virtual address `[0, 4MB)` and `[KERNBASE, KERNBASE+4MB)` to the same physical address `[0, 4MB)` . This transition is necessary, for the reason that our kernel ELF's VMA is in high address.

Challenges

Large page size, 4MB pages

First we must check that the CPU supports `PSE` , page size extension. I've integrated the `cpuinfo` command into kernel monitor. And yes, our dear friend QEMU supports `PSE` . In order to enable 4M pages, we must set both the `CR4.PSE` and the `PDE.PS` bit.

reference:Understanding 4M Page Size Extensions on the Pentium Processor, Robert R. Collins

But in reality, we must completely refactor our kernel, especially the memory management part, to make use of 4M pages, since most of our functions relied on the fact that pages were 4K in size, and our hardware MMU did two-stage paging.

In fact it is not a bad idea to combine 4M super-pages with 4K normal pages. The advantages are promising but it seams that both OS developers and CPU designers prefers 4K pages. I've checked TLB details on recent Intel processors namely i5-2400 Sandy Bridge, and discovered that the processor has significantly less TLB entries for 4M pages. I know we would need less TLB entries, and the miss penalty is significantly reduced, but the processors are really tuned to use 4K pages.

For systems using 64-bit processors and much more than 4G RAM, paging is some sort of performance bottle neck for real world applications. So large pages might become a new trend. (Well, not before the DRAM price ever drop back to a reasonable level, which will never happened if it follows the path of hard-disks).

Extend the kernel monitor to show and manage memory mappings

I've written a `showmappings` command, which take 2 arguments, interpret them as virtual addresses, and show mappings in between (inclusive, to page boundary). The permission bits: see QEMU source code `<QEMU_DIR>/monitor.c:2190-2197`, and their meanings: Paging - OSDev. And the screenshot `showmappings.png`.

~~I seriously doubt the necessity of implementing a function to '*Explicitly set, clear, or change the permissions of any mapping in the current address space*'. It might be cool to fiddle with page permissions through a command-line interface, but wouldn't it be better to do that by code? Another reason is that it is very annoying to parse command line arguments, without proper libraries.~~

And I did `memdump` command, to inspect both virtual memory and physical memory. It does careful checking on page borders and ranges. See the screenshot `memdump.png`.

Redesign the kernel to allow user programs to use the full 4GB virtual address space

Currently I do not have a very clear view of Process Management, or System Call Interface. Therefore it is rather hard to present a thorough design, especially when Google fails to come up with a decent page explaining what exactly '*Follow up the bouncing kernel*' is.

I don't see many advantages at the moment, to give user programs the whole address space, on architectures using 32-bit address lines. We are all using 64-bit PC, and we need more than 4G RAM (not just 4G address space, which might be backed by pages on hard-disk), and without PAE. For 32-bit mobile devices, currently no manufacturer stacks 4G RAM inside a *mobile* device. And Apple's just introduce iPhone 5S, with a 64-bit processor! That solves all problems, regarding limited address space.

As for disadvantages, the first one is: Where do I keep the kernel in this virtual address space? OK, we could redirect page directories, each time a context switch occurs, which obviously slows down our OS. Next, how does the kernel grab data from user space? OK, we could do manual address translation, and remap the physical address to some where in kernel space, but again, EVERY time a system call occurs.

So, most rational system designers allocated some room in the virtual address space for the all mighty kernel to breathe.

Implement a general allocator, to support 2^n-pages allocations

The most straight forward solution could be '*buddy system*'. A better one would be using the `malloc/free` 's idea, on page granularity. I'll implement this a bit later, after I figure out typical cases where a general allocator is necessary.