CS444/544
Operating Systems II

Lecture 6
JOS Memory Management and Quiz 1 Prep.
4/17/2024

Acknowledgement: Slides drawn heavily from Yeongjin Jiang
Due Reminder

• Lab 1 past due...
  • 75% due: 4/22 11:59 PM

• Lab 2 posted
  • 100% due: 4/29 11:59 PM
  • 75% due: 5/6 11:59 PM
QUIZ 1 (4/22)

• We will have Quiz 1 on Monday
  • More info later

• No class on Monday (4/22)
Recap: PDE/PTE Permission Examples

- Virtual address 0x01020304

- PDE: PTE_P | PTE_W

- PTE: PTE_P | PTE_U

  - valid, inaccessible by ring3, not writable

  - PTE_P (PRESENT)
    - 0: invalid entry
    - 1: valid entry

  - PTE_W (WRITABLE)
    - 0: read only
    - 1: writable

  - PTE_U (USER)
    - 0: kernel (only ring 0 can access)
    - 1: user (accessible by ring 3)
Recap: PDE/PTE Permissions CAVEAT

• A virtual address access is allowed if both PDE and PTE entries allows the access...

• General practice: put a more permissive permission bits in PDE, and be strict on setting permission bits in PTE

• For a conflicting permission setup for Kernel/User, add an additional virtual address mapping can enable such a setup
Recap: You can setup the following page permissions...

- **Kernel:** RW, **User:** R
  - VA 0x00001000 -> PA 0x50000, PTE_P | PTE_U (User R)
  - VA 0xf0050000 -> PA 0x50000, PTE_P | PTE_W (Kernel RW)

- **Kernel:** R, **User:** RW
  - VA 0x00002000 -> PA 0x60000, PTE_P | PTE_U | PTE_W (User RW)
  - VA 0xf0060000 -> PA 0x60000, PTE_P (Kernel R)

- **Kernel:** --, **User:** RW
  - VA 0x00003000 -> PA 0x70000, PTE_P | PTE_U | PTE_W
  - VA 0xf0070000 -> PA 0x70000, 0 for flag...
Today’s Topic

• Managing Physical/Virtual Memory in JOS

• Prep for Quiz 1
Creating a Virtual Memory Space

• A page directory manages the entire virtual memory space
  • For a process

• CR3 points to the Page directory, and each PDE entry points to a PT...
Assigning VA -> PA mapping

• Suppose a process would like to use a virtual address
  • 0x400000 (RW from user)

• Allocation procedure
  • Check page directory entry (PDE)
    • If not set with PTE_P, allocate a physical page for a new page table
Assigning VA -> PA mapping

• Suppose a process would like to use a virtual address
  • 0x400000 (RW from user)

• Allocation procedure
  • Check page directory entry (PDE)
    • If not set with PTE_P, **allocate a physical page** for a new page table

```
Page at 0x11223000
PDE 0: EMPTY
PDE 1: 0x11223
PDE 2: EMPTY
PDE ...: EMPTY
PDE ...: EMPTY
PDE 1022: EMPTY
PDE 1023: EMPTY

Page at 0x153000
```

```
PTE 0: EMPTY
PTE 1: EMPTY
PTE 2: EMPTY
PTE ...: EMPTY
PTE 1022: EMPTY
PTE 1023: EMPTY
```
Assigning VA -> PA mapping

• Suppose a process would like to use a virtual address
  • 0x400000 (RW from user)

• Allocation procedure
  • Check page directory entry (PDE)
    • If not set with PTE_P, allocate a physical page for a new page table
  • Check page table entry (PTE)
    • If not set with PTE_P, allocate a physical page to enable access
Assigning VA -> PA mapping

• Suppose a process would like to use a virtual address
  • 0x400000 (RW from user)

• Allocation procedure
  • Check page directory entry (PDE)
    • If not set with PTE_P, allocate a physical page for a new page table
  • Check page table entry (PTE)
    • If not set with PTE_P, allocate a physical page to enable access

• We need to keep track of ‘free’ physical pages...
Struct PageInfo *pages in JOS

• A **one-to-one** mapping from a **struct PageInfo** to a physical page
  • An 8 byte struct per each physical memory page
  • If we **support 128MB memory**, then we will create
    • Total number of physical pages: \( 128 \times 1048576 / 4096 = 32768 \)
    • Total size of pages = \( 32768 \times 8 = 262,144 = 256\text{KB} \) for **pages**

• A **linked-list** for managing free physical pages
  • Starting from **page_free_list->pp_link**...

  **pp_ref**
  • Count references
  • Non-zero – in-use
  • Zero – free

```c
struct PageInfo {
    // Next page on the free list.
    struct PageInfo *pp_link;

    // pp_ref is the count of pointers (usually in page table entries)
    // to this page, for pages allocated using page_alloc.
    // Pages allocated at boot time using pmap.c's
    // boot_alloc do not have valid reference count fields.

    uint16_t pp_ref;
};
```
How JOS manages Physical Memory?

• Struct PageInfo
  • A metadata type that counts number of ‘references’ of the page
  • NOT IN USE : pp_ref == 0

• Struct PageInfo * page_free_list
  • A linked list that contains free physical pages

• We will create Struct PageInfo per each Physical Page and then
  • Create a linked list of free pages...
Example

Struct PageInfo * pages (array)

<table>
<thead>
<tr>
<th>idx</th>
<th>pp_ref</th>
<th>pp_link</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Physical memory

Page 0
Page 1
Page 2
Page 3
Page N
Example $n_{\text{pages}}$

$128 \times 1048576 / 4096 = 32768$ Pages

$8$ byte per each entry $= 32K \times 8 = 256KB$

Struct PageInfo * pages (array)

<table>
<thead>
<tr>
<th>idx</th>
<th>pp_ref</th>
<th>pp_link</th>
</tr>
</thead>
<tbody>
<tr>
<td>32K</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

We can put this array into our physical memory
Free Physical Memory (init)

In kern/pmap.c, boot_alloc

```c
static void *
boot_alloc(uint32_t n)
{
    static char *nextfree; // virtual address of next byte of free memory
    char *result;

    // Initialize nextfree if this is the first time.
    // 'end' is a magic symbol automatically generated by the linker,
    // which points to the end of the kernel's bss segment:
    // the first virtual address that the linker did *not* assign
    // to any kernel code or global variables.
    if (!nextfree) {
        extern char end[];
        nextfree = ROUNDUP((char *) end, PGSIZE);
    }
}
```

nextfree will point to the end of the kernel code/data
Free Physical Memory (init)

In kern/pmap.c, boot_alloc

```c
static void *
boot_alloc(uint32_t n)
{
    static char *nextfree;  // virtual address of next byte of free memory
    char *result;

    // Initialize nextfree if this is the first time.
    // 'end' is a magic symbol automatically generated by the linker,
    // which points to the end of the kernel's bss segment:
    // the first virtual address that the linker did *not* assign
    // to any kernel code or global variables.
    if (!nextfree) {
        extern char end[];
        nextfree = ROUNDUP((char *) end, PGSIZE);
    }
}
```

nextfree will point to the end of the kernel code/data
Allocating struct PageInfo

```
// These variables are set in mem_init()
pde_t *kern_pgdir; // Kernel's initial page directory
struct PageInfo *pages; // Physical page state array
static struct PageInfo *page_free_list; // Free list of physical pages

struct PageInfo {
    // Next page on the free list.
    struct PageInfo *pp_link;
    // pp_ref is the count of pointers (usually in page table entries)
    // to this page, for pages allocated using page_alloc.
    // Pages allocated at boot time using pmap.c's
    // boot_alloc do not have valid reference count fields.
    uint16_t pp_ref;
};
```

```
pages = boot_alloc(npages * sizeof(struct PageInfo));
```
Where are the free pages?

• in page_init()

// The example code here marks all physical pages as free.
// However this is not truly the case. What memory is free?
// 1) Mark physical page 0 as in use.
// This way we preserve the real-mode IDT and BIOS structures
// in case we ever need them. (Currently we don't, but...)
// 2) The rest of base memory, [PGSIZE, npages_base_mem * PGSIZE)
// is free.
// 3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
// never be allocated.
// 4) Then extended memory [EXTPHYSMEM, ...).
// Some of it is in use, some is free. Where is the kernel
// in physical memory? Which pages are already in use for
// page tables and other data structures?
// Change the code to reflect this.
// NB: DO NOT actually touch the physical memory corresponding to
// free pages!
Where are the free pages?

- in `page_init()`

```c
// The example code here marks all physical pages as free.
// However this is not truly the case. What memory is free?
// 1) Mark physical page 0 as in use.
//    This way we preserve the real-mode IDT and BIOS structures
//    in case we ever need them. (Currently we don't, but...)
// 2) The rest of base memory, [PGSIZE, npages_baseemem * PGSIZE)
//    is free.
// 3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
//    never be allocated.
// 4) Then extended memory [EXTPHYSMEM, ...).
//    Some of it is in use, some is free. Where is the kernel
// in physical memory? Which pages are already in use for
// page tables and other data structures?
// Change the code to reflect this.
// NB: DO NOT actually touch the physical memory corresponding to
// free pages!
```
Where are the free pages?

• Page 0 is in-use

• Pages in [IOPHYSMEM ~ EXTPHYSMEM] are in-use

• Pages for the kernel code are in-use

• Pages for struct PageInfo *pages are in-use

• How can you point this?
  • pages + npages ?
  • boot_alloc(0)?

boot_alloc(0) is better...
Reference Counting

• A typical mechanism for tracking free memory blocks

• Mechanism
  • Count up the value (pp_ref++) if the page is referenced by others (in use!)
  • Count down the value (pp_ref--) if not used for one of usages anymore
  • Free if pp_ref == 0

• In C++, `shared_ptr<T>`
  • When a pointer is assigned to a variable, count up!
  • When the variable no longer uses the variable, count down!
  • Free the memory when the count become 0
Ref. Counting with struct PageInfo

• For in-use memory
  • Set \texttt{pp\_ref = 1}

• For not-in-use memory
  • Invariant: \texttt{pp\_ref == 0}
  • Must be linked with \texttt{pages\_free\_list}

• When assigning the page to a virtual address
  • \texttt{pp\_ref++}

• When releasing the page from a virtual address
  • \texttt{pp\_ref--}
Caveat

• Some pages are mapped but does not have to be marked as in-use
• Make sure you do not count up pages for dirmap
  • 0xf0000000 ~ 0xffffffff

• Read the comment at the top of boot_map_region thoroughly

// This function is only intended to set up the `static' mappings
// above UTOP. As such, it should *not* change the pp_ref field on the
// mapped pages.
//
// Hint: the TA solution uses pgdir_walk
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
Linked-list for Free Pages

• Start with NULL at the head
  • page_free_list = NULL;

• After set pp_ref of all pages, do something like the following:

```c
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```
page2pa(struct PageInfo *pp)

• Changes a pointer to struct PageInfo to a physical address
• \( \text{idx} = (\text{pp} - \text{pages}) \)
  • Gets the index of pp in pages
  • E.g., &pages[idx] == pp
• idx here is a physical page number

<table>
<thead>
<tr>
<th>idx</th>
<th>pp_ref</th>
<th>pp_link</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

pp – pages = 4
0x4000 \( \leftarrow \) physical page address!
pa2page(physaddr_t pa)

- PNUM(pa)
  - Returns page number
- &pages[PNUM(pa)]
  - Returns struct PageInfo * of that pa.

```c
static inline struct PageInfo* pa2page(physaddr_t pa) {
    if (PNUM(pa) >= npages)
        panic("pa2page called with invalid pa");
    return &pages[PNUM(pa)];
}
```
Quiz 1 (4/22)

• Released via CANVAS
  • Quiz 1 available at 8:00 am
  • Deadline: 4/22 11:59pm
  • Duration: 90 min, but you can finish it around 30 min

• You will have 2 attempts to take quiz (but you CANNOT see the quiz results for those attempts)

• Open material; you may refer to
  • Contents at our Canvas course website
  • Slides
  • Lab document and tutorials
  • Your code for Lab 1 / Lab 2
  • Textbook (not required)

Communicating with others during Quiz is not allowed
Quiz 1 (4/22)

• Question type: T/F, multiple choices, less than 15 questions
  • 1 pts per each question
• All three weeks content will be covered in the Quiz 1
  • BIOS/Booting/CPU, Real mode segmentation (Lecture 2)
  • Protected mode segmentation and Paging (Lecture 3)
  • Virtual address translation (Lecture 4)
  • Virtual memory layout (Lecture 5)
  • JOS Memory management (Lecture 6)
  • JOS Lab 1 (Lab Tutorial 1 & 2)
  • First part of JOS Lab 2 (Lab Tutorial 3)
Prep for Quiz 1

• Which one of the following is not a job that JOS Bootloader does?
  • A. Enable protected mode
  • B. Enable paging
  • C. Load kernel image from disk
  • D. Enable A20
Prep for Quiz 1

Which one of the following is not a job that JOS Bootloader does?

• A. Enable protected mode
• B. Enable paging (is done in kernel, in kern/entry.S)
• C. Load kernel image from disk
• D. Enable A20
Prep for Quiz 1

• In the x86 real mode, which address the following segment:offset pair points to?

• 0x8000:0x3131
  • A. 0xb131
  • B. 0x3131
  • C. 0x83131
  • D. 0x103131
  • E. 0x11131
Prep for Quiz 1

• In the x86 real mode, which address the following segment:offset pair points to?

• 0x8000:0x3131
  • A. 0xb131
  • B. 0x3131
  • C. 0x83131 (0x8000 * 16 + 0x3131 = 0x80000 + 0x3131 = 0x83131)
  • D. 0x103131
  • E. 0x11131
Prep for Quiz 1

• Which of the following x86 register stores the current privilege level?
  • A. ds
  • B. eip
  • C. ebp
  • D. esp
  • E. cs
Prep for Quiz 1

• Which of the following x86 register stores the end of the current stack frame (and moves if the CPU runs push/pop) ?
  • A. ds
  • B. eip
  • C. ebp
  • D. esp
  • E. cs

Answer: D. esp
Which of the following x86 register stores the start of the current stack frame (also points to the address that stores previous frame’s stack base pointer)?

A. ds
B. eip
C. ebp
D. esp
E. cs
Prep for Quiz 1

• What kind of benefit can we enjoy by enabling virtual memory?
• Choose all (no partial credits)
  • A. Performs faster execution than when using physical memory
  • B. Suffers less memory fragmentation than when using physical memory
  • C. Provides a better isolation / protection than when using physical memory
  • D. Provides memory transparency
  • E. Enables virtual reality