# Introduction to Computer Systems
# 15-213/18-243 Spring 2009
# March 30, 2009

## Dynamic Memory Allocation

# Overview

- **News**
- **Virtual Memory**
  - 2-level example
- **Malloc**
  - Basics
  - Theory
  - Implicit list

# News

- Shell Lab due Tuesday, 3/31
- Malloc lab out soon
- Exam 2 on Apr 7

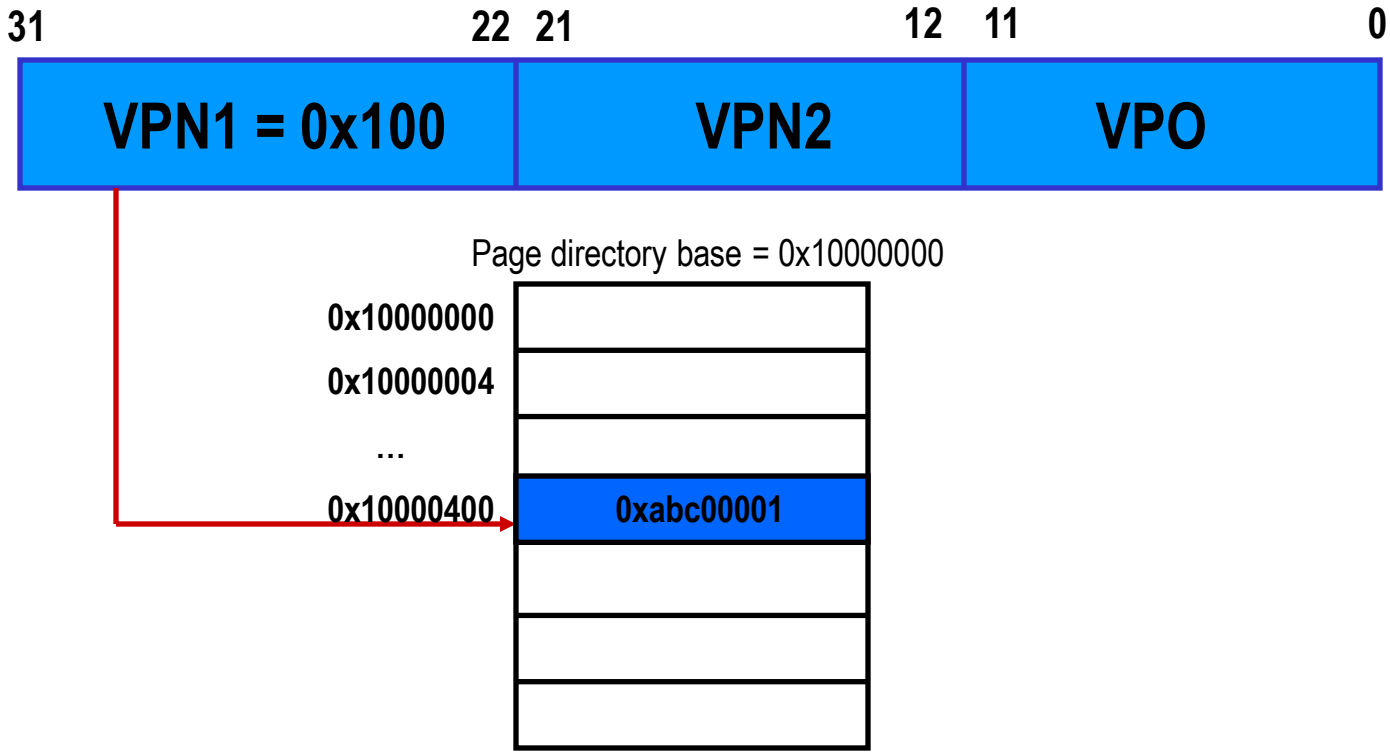
# VM (review)

- Each virtual address encodes several things
  - Virtual Page Number(s) – Index into the page table/directory
  - Virtual Page Offset – Offset in terms of bytes from the start of the page frame. This is equal to the physical page offset


1 level page table

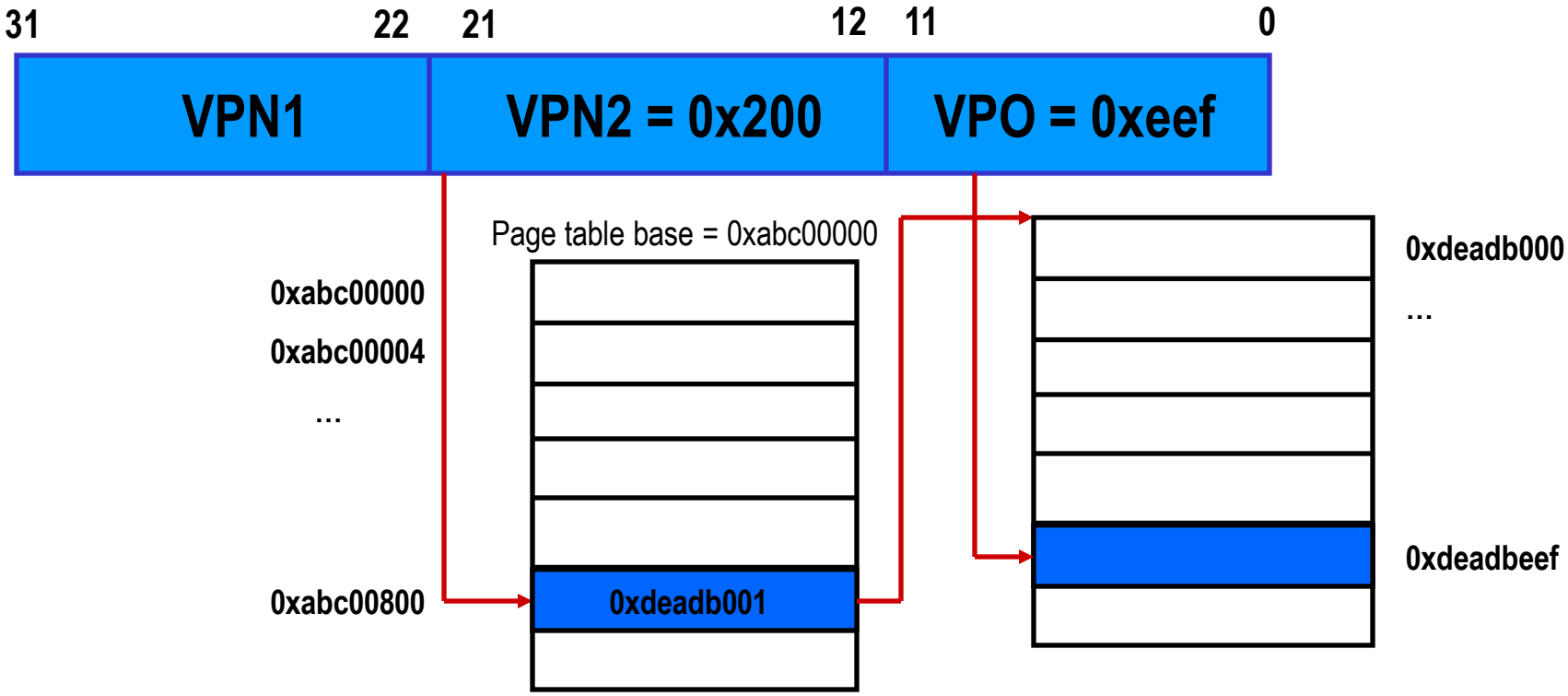

2 level page table

# Example: 2 level page table




Use the first VPN to index into the page directory. This gives the address of the start of the page table.

# 2-level page table – cont'd




Use the second VPN to index into the page table. This gives the address of the start of the page frame. Add the offset to obtain the location in physical memory.

# VM – Example (Spring ’08 final)

The following problem concerns virtual memory and the way virtual addresses are translated into physical addresses. Below are the specifications of the system on which the translation occurs.

- The system is a 16-bit machine - words are 2 bytes.
- Memory is byte addressable.
- The maximum size of a virtual address space is 64KB.
- The system is configured with 16KB of physical memory.
- The page size is 64 bytes.
- The system uses a two-level page tables. Tables at both levels are 64 bytes (1 page) and entries in both tables are 2 bytes

# What does this tell us?

**Memory is byte addressable**

- Must be able to offset to any byte of a page

**The maximum size of a virtual address space is 64KB.**

- Makes sense – 2^16 = 64 KB

**The system is configured with 16KB of physical memory.**

- Not really important – VM can also map to disk

# What does this tell us?

**The page size is 64 bytes.**

**The system uses a two-level page tables. Tables at both levels are 64 bytes (1 page) and entries in both tables are 2 bytes**

- 64 bytes/2 bytes/entry = 32 = $2^5$ entries/page
- Need 5 bits to index a page
- Address will have 2 VPNs of 5 bits each

**Sanity check: 16 bits/address – 5*2 (for VPNs) = 6 bits for VPO**

**$2^6$ = 64, so offset can byte-address a page**

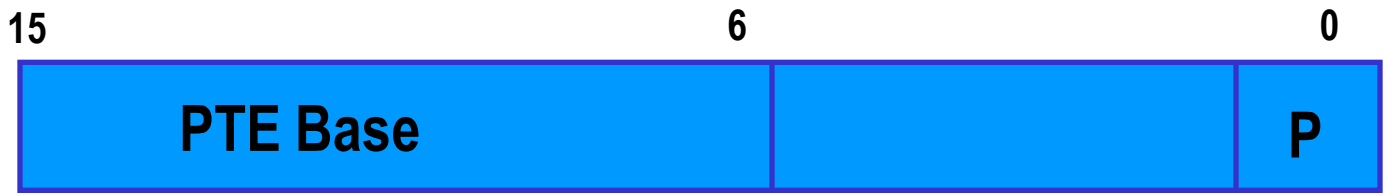
**Result:**

# PDE Format (given)

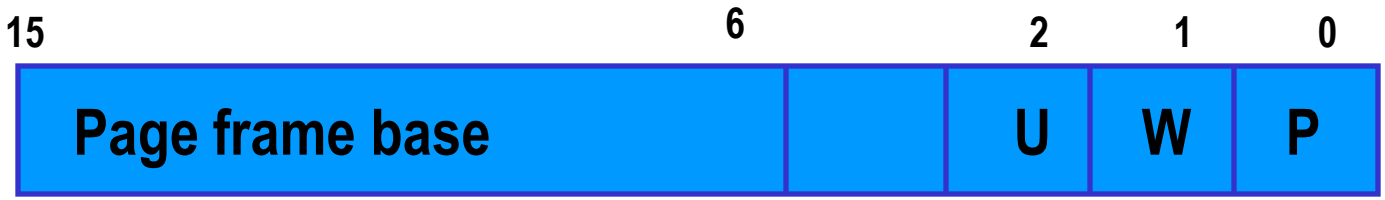

# PTE Format (given)


- P = 1 => Present
- W = 1 => Writable (applies both in kernel and user mode)
- U = 1 => User-mode

# PDE Format




# PTE Format




Why 10 bytes?

- Page table & page both 64 = $2^6$ bytes
- 16 bit address – 6 bits = 10 bit base

# Memory contents:

| Address | Contents |
|---|---|
| 0118 | 2381 |
| 0130 | 2101 |
| 0160 | 2281 |
| 018E | 1581 |
| 019C | 1201 |
| 01B8 | 1A01 |
| 120A | 2701 |
| 1214 | 27C1 |
| 1228 | 2741 |
| 158A | 25C1 |
| 1594 | 2541 |
| 15A8 | 2501 |
| 1A0A | 2041 |

| Address | Contents |
|---|---|
| 1A14 | 20C1 |
| 1A28 | 2081 |
| 2106 | 3FC7 |
| 210C | 3A47 |
| 2118 | 3587 |
| 2286 | 3107 |
| 228C | 3447 |
| 2298 | 3007 |
| 2386 | 33C7 |
| 238C | 3887 |
| 2398 | 3247 |
| | |
| | |

Assume omitted entries' contents are 0

Give PDE & PTE addresses, and result for:

0xC1B2 (write in user mode, PD base 0x0100)

0x728F (write in kernel mode, PD base 0x0180)

Hint: Write out bits, divide into VPNs and VPO

# Answers:

0xC1B2 (write in user mode, PD base 0x0100)

PDE addr: 0x0130

PTE addr: 0x210C

Final addr: 0x3A72

Success

0x728F (write in kernel mode, PD base 0x0180)

PDE addr: 0x019C

PTE addr: 0x1214

Final addr: 0x27CF

Failure (page not writable)

# Dynamic memory allocation

Why?

- Size of needed data structures may only be known at runtime
- Might want memory to persist after function returns

# Memory allocator

- Manages a large block of memory
- Hands out blocks of memory of requested sizes
- Handles freeing of memory when no longer needed
- Important: On Unix systems, allocated memory must be 8-byte aligned

# Malloc package

- **#include <stdlib.h>**
- **void *malloc(size_tsize)**
  - Successful:
    - Returns a pointer to a memory block of at least sizebytes(typically) aligned to 8-byte boundary
    - If size == 0, returns NULL
  - Unsuccessful: returns NULL (0) and sets errno
- **void free(void *p)**
  - Returns the block pointed at by pto pool of available memory
  - p must come from a previous call to malloc() or realloc()
- **void *realloc(void *p, size_tsize)**
  - Changes size of block pand returns pointer to new block
  - Contents of new block unchanged up to min of old and new size
  - Old block has been free()'d (logically, if new != old)

# Performance goals

- **Throughput**
  - Requests/time
  - Similar to MFLOPs in perflab
- **Memory utilization**
  - Ratio of memory used by program to heap size
  - Reduced by fragmentation
    - Internal (overhead)
    - External (unused space)

# Internal Fragmentation

- **Green = memory requested by program**
- **Red = internal fragmentation**
- **Why?**
  - Allocator overhead
    - Retaining sizes, pointers, etc.
  - Alignment issues
    - Memory + overhead must be a multiple of 8
    - Sometimes, allocate more than requested

# External Fragmentation

- Occurs when there is enough aggregate heap memory, but no single free block is large enough

Example:

p1 = malloc(4); p2 = malloc(5); p3 = malloc(6);


free(p2)


p4 = malloc(6)

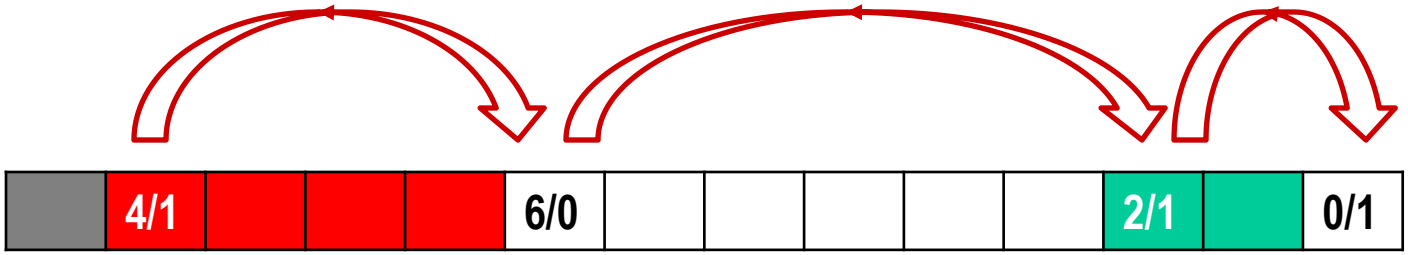
- Need to extend heap
- Space where p2 was is wasted (for now)

# Implementation

- **Implicit list**
  - Remember only block sizes
  - Traverse all blocks (free & allocated) to find a block for malloc
- **Explicit list**
  - Free blocks keep “next” pointers (linked list)
  - Traverse only free blocks
- **Segregated list**
  - Separate lists for different “size classes”
  - Reduces external fragmentation
- **Blocks sorted by size**
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key
  - Not covered in 213

# Implicit list

- Need to store size and if block is allocated or free
- Revelation: thanks to alignment, sizes will be multiples of 8
- Can use low bit of size to store allocated/free




- Here, 1 square = 1 word
- First word unused thanks to alignment constraint
- Last word is a sentinel – marks end of list

# Finding a free block

- **First-fit**
  - Traverse list from start, use first free block that fits requested size
  - Possibly linear time
  - Can cause “splinters” at start of list
- **Next-fit**
  - Start where you left off on the last search
  - Usually faster than first-fit
  - Some studies claim it causes more fragmentation
- **Best-fit**
  - Choose best fit for requested size
  - Always have to traverse whole list
  - Reduces external fragmentation

# Questions?