W4118: dynamic memory allocation

Instructor: Junfeng Yang

References: Modern Operating Systems (3rd edition), Operating Systems Concepts (8th edition), previous W4118, and OS at MIT, Stanford, and UWisc

Outline

- Dynamic memory allocation overview
- Heap allocation strategies
- Memory management review
    - Copy-on-write

Dynamic memory allocation

- Static (compile time) allocation is not possible for all data

- Two ways of dynamic allocation
  - Stack allocation
    - Restricted, but simple and efficient
  - Heap allocation
    - More general, but less efficient
    - More difficult to implement

Dynamic allocation issue: fragmentation

- Fragment: small trunks of free memory, too small for future allocation requests "holes"
  - External fragment: visible to system
  - Internal fragment: visible to process (e.g. if allocate at some granularity)

- Goal
  - Reduce number of holes
  - Keep holes large

- Stack fragmentation v.s. heap fragmentation

Typical heap implementation

- Data structure: free list
  - Chains free blocks together
- Allocation
  - Choose block large enough for request
  - Update free list
- Free
  - Add block back to list
  - Merge adjacent free blocks

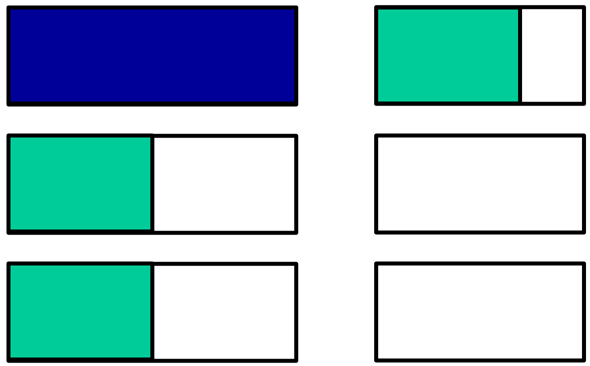
Heap allocation strategies

- Best fit
  - Search the whole list on each allocation
  - Choose the smallest block that can satisfy request
  - Can stop search if exact match found
- First fit
  - Choose first block that can satisfy request
- Worst fit
  - Choose largest block (most leftover space)

Which is better?

Example

- Free space: 2 blocks, size 20 and 15
- Workload 1: allocation requests: 10 then 20

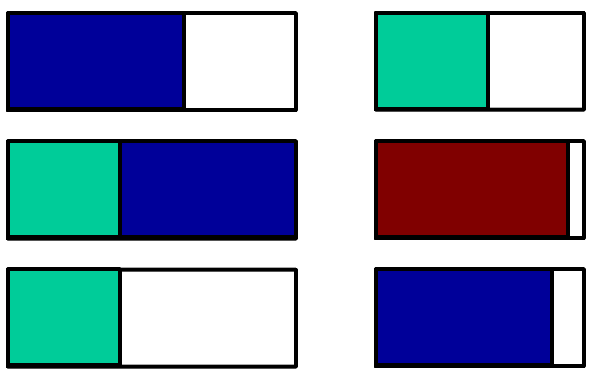


Best fit

First fit — Request of 20: fail!

Worse fit — Request of 20: fail!

- Workload 2: allocation requests: 8, 12, then 13



Best fit — Request of 13: fail!

First fit

Worse fit — Request of 13: fail!

Comparison of allocation strategies

- Best fit
  - Tends to leave very large holes and very small holes
  - Disadvantage: very small holes may be useless
- First fit:
  - Tends to leave "average" size holes
  - Advantage: faster than best fit
- Worst fit:
  - Simulation shows that worst fit is worst in terms of storage utilization

Buddy allocator motivation

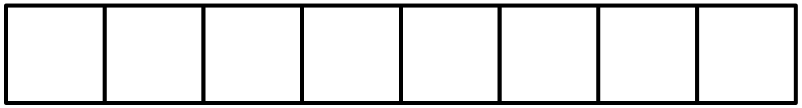
- Allocation requests: frequently 2^n
  - E.g., allocation physical pages in Linux
  - Generic allocation strategies: overly generic
- Fast search (allocate) and merge (free)
  - Avoid iterating through free list
- Avoid external fragmentation for req of 2^n
- Keep physical pages contiguous

Real: used in FreeBSD and Linux

Buddy allocator implementation

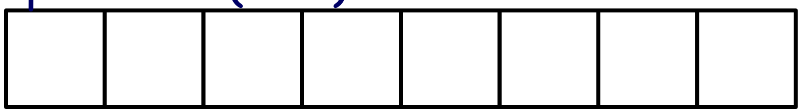
- Data structure
  - N free lists of blocks of size 2^0, 2^1, …, 2^N
- Allocation restrictions: 2^k, 0<= k <= N
- Allocation of 2^k:
  - Search free lists (k, k+1, k+2, …) for appropriate size
    - Recursively divide larger blocks until reach block of correct size
    - Insert “buddy” blocks into free lists
- Free
  - Recursively coalesce block with buddy if buddy free

Buddy allocation example



freelist[3] = {0}

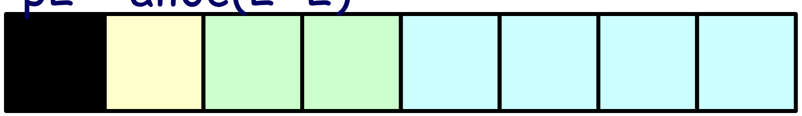
p1 = alloc(2^0)



freelist[0] = {1}, freelist[1] = {2}

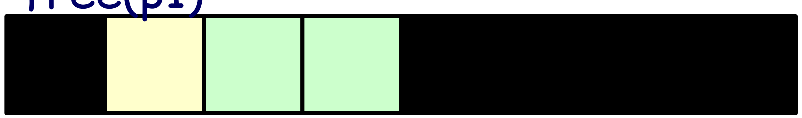
freelist[2] = {4}

p2 = alloc(2^2)



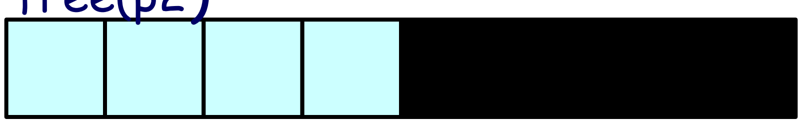
freelist[0] = {1}, freelist[1] = {2}

free(p1)



freelist[2] = {0}

free(p2)



freelist[3] = {0}

Pros and cons of buddy allocator

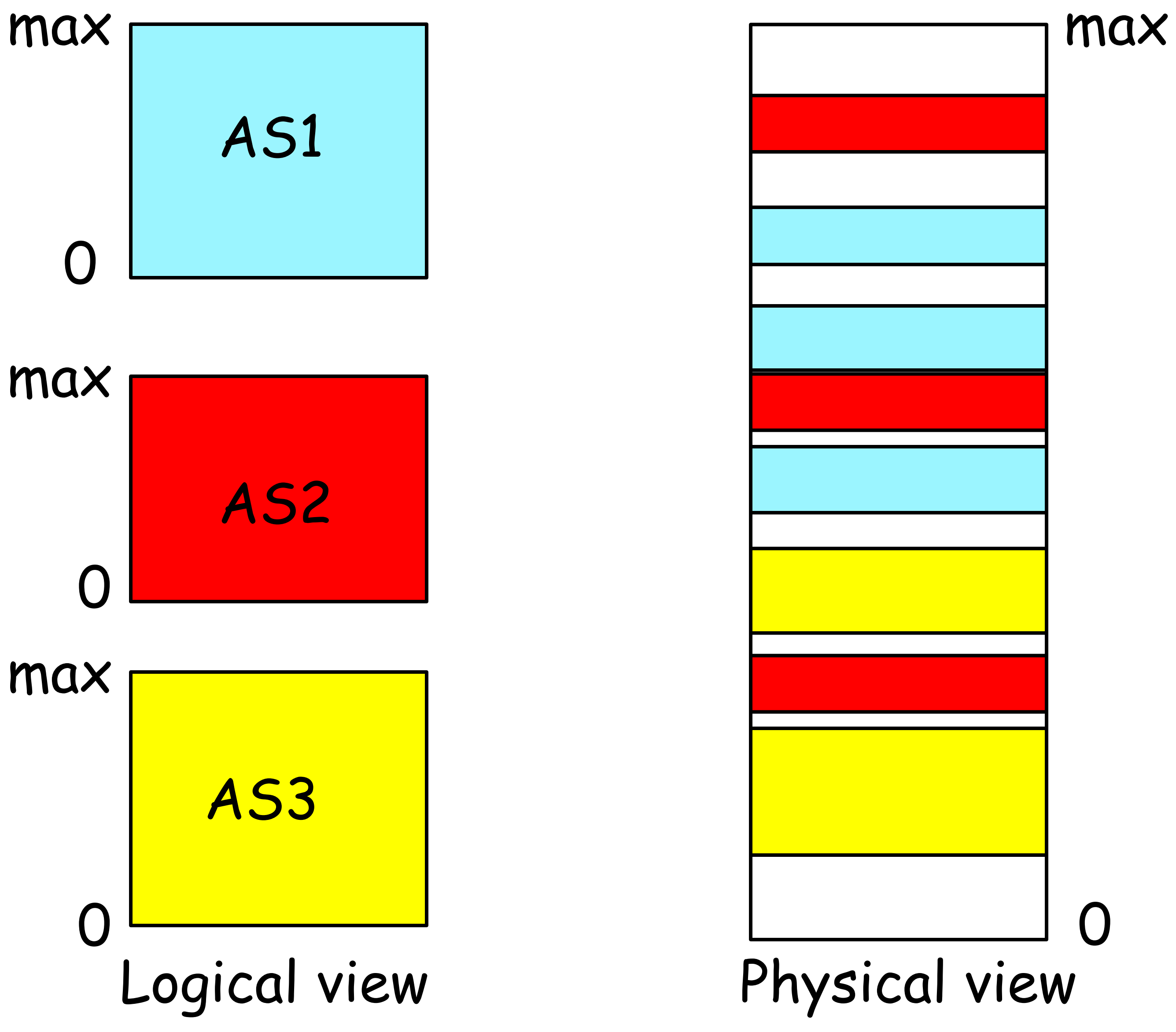
- Advantages
  - Fast and simple compared to general dynamic memory allocation
  - Avoid external fragmentation by keeping free physical pages contiguous
- Disadvantages
  - Internal fragmentation
    - Allocation of block of k pages when k != 2^n

Slab allocator

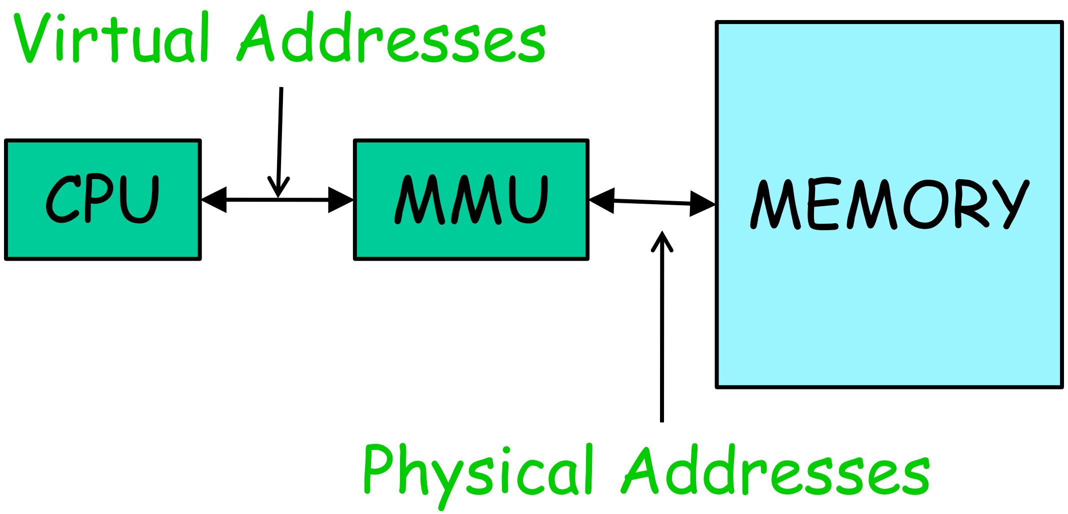
- Motivation:
  - Frequent (de)allocationof certain kernel objects
    - E.g., file struct and inode
  - Other allocators: overly general; assume variable size
- Slab: cache of "slots"
  - Slot size = object size
  - Free memory management = bitmap
  - Allocate: set bit and return slot
  - Free: clear bit
- Real: used in FreeBSD and Linux, implemented on top of buddy page allocator, for objects smaller than a page

Memory management review

Multiple address spaces co-exist



Memory Management Unit (MMU)

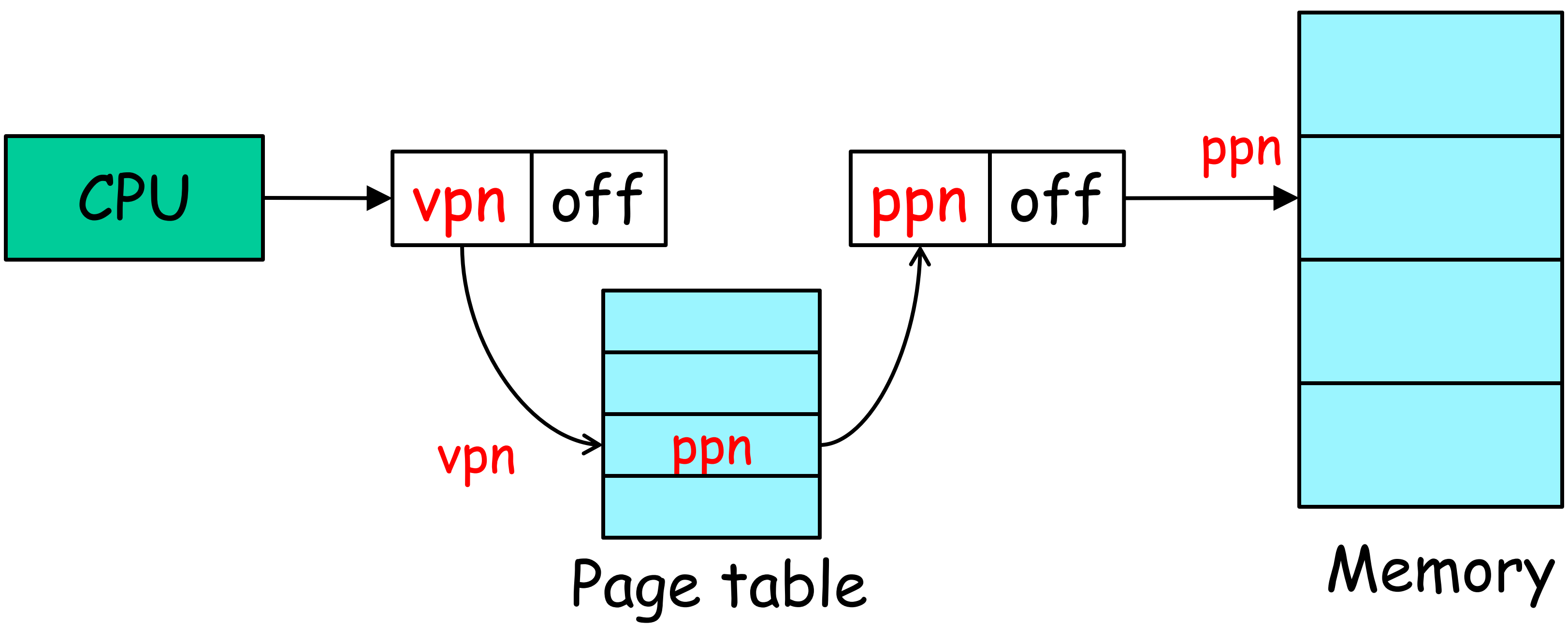


- Map program-generated address (virtual address) to hardware address (physical address) dynamically at every reference
- Check range and permissions
- Programmed by OS

Page translation

- Address bits = page number + page offset
- Translate virtual page number (vpn) to physical page number (ppn) using page table

    pa = page_table[va/pg_sz] + va%pg_sz



Page protection

- Implemented by associating protection bits with each virtual page in page table
- Protection bits
  - present bit: map to a valid physical page?
  - read/write/execute bits: can read/write/execute?
  - user bit: can access in user mode?
  - x86: PTE_P, PTE_W, PTE_U
- Checked by MMU on each memory access

A cool trick: copy-on-write

- In fork(), parent and child often share significant amount of memory
  - Expensive to copy all pages
- COW Idea: exploit VA to PA indirection
  - Instead of copying all pages, share them
  - If either process writes to shared pages, only then is the page copied
- Real: used in virtually all modern OSes

How to implement COW?

- (Ab)use page protection
- Mark pages as read-only in both parent and child address space
- On write, page fault occurs
- In page fault handler, distinguish COW fault from real fault
  - How?
- Copy page and update page table if COW fault