CS179F: Projects in Operating System

Lab 2: Memory Allocation

Emiliano De Cristofaro and Lian Gao

Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free*
- Types of allocators
    - Explicit allocator: application allocates and frees space (e.g., malloc and free in C)
    - Implicit allocator: application allocates, but does not free space (e.g., garbage collection in Java, ML, and Lisp)
- We will discuss explicit memory allocation

malloc-like functions

void *malloc(size_t size)

- Successful: returns a memory block of at least size bytes (if size == 0, returns NULL)
- Failure: returns NULL

void free(void *p)

- Returns the memory block to the pool
- p must come from a previous call to malloc or realloc

Design Goals

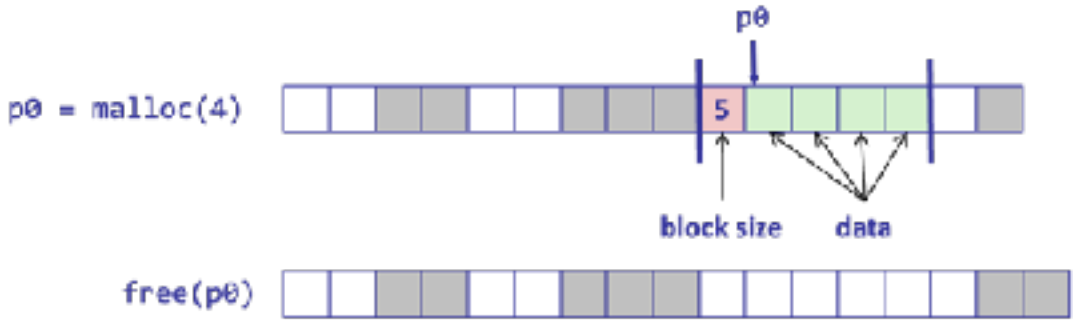
- Throughput: number of completed allocation/free requests per unit time
    - We don't want the heap allocator to become the performance bottleneck
- Memory utilization: total memory size required to fulfill the requests
    - Fragmentations and poor allocation policies can cause poor memory utilization

Implementation Issues

- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- How do we pick a block to use for allocation -- many might fit?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we reinsert freed block?

Track Block Size

- Implicit: the size can be inferred with other information (e.g., the buddy allocator)
- Explicit: record the size information as part of the allocated memory block
    - Requires extra space to store the header



Keep Tracking of Free Blocks

- Implicit list: using a length field to “link” all blocks, and an allocation flag/bit to indicate availability/free

- Explicit list: using a (doubled) link list to track all free blocks
- Separate free list: an explicit free list for each different size of blocks
- A heap (sorted tree): a balanced tree (e.g., red-black tree) to store memory blocks, using size as the key

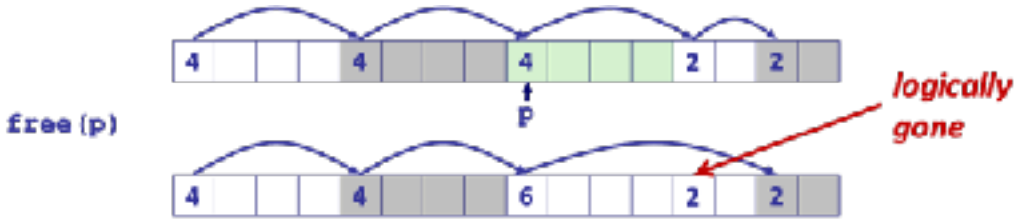
Which Block to Pick

- Frist fit
    - Search the list from the beginning, choose the first free block that fits
    - Can take linear time to scan, can cause “splinters” at the beginning
- Next fit
    - Like first fit, but starts from where the previous search finished
- Best fit
    - Choose the smallest block that fits (keeps fragmentations small)

Coalescing

Defragmentation

- Join (coalesce) with next/previous blocks, if they are free



Freeing With Explicit Free Lists

Insertion Policy

- LIFO (last-in-first-out)
    - Insert freed block at the beginning of the free list
    - Simple and constant time, but can cause more fragmentations
- Address-ordered
    - Blocks in the free list is ordered by addresses: addr(prev) < addr(curr) < addr(next)
    - Requires search, but keeps fragmentations smaller

Buddy Allocator Overview

- Memory allocation algorithm that divides memory into partitions to try to satisfy a memory request as suitably as possible
- Uses of splitting memory into halves to try to give a best fit
- There are various forms of the buddy system, each block subdivided into two smaller blocks (the simplest and most common variety)
    - Every memory block has an order, an integer ranging from 0 to a specified upper limit.
    - Size of a block of order n is proportional to 2^n, so that the blocks are exactly twice the size of blocks that are one order lower
    - Power-of-two block sizes make address computation simple
    - When a larger block is split, it is divided into two smaller blocks, and each smaller block becomes a unique buddy to the other. A split block can only be merged with its unique buddy block, which then reforms the larger block they were split from.

The Buddy Allocator

Split and Merge

Step	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K
1	2^4															
2.1	2^3								2^3							
2.2	2^2				2^2				2^3							
2.3	2^1		2^1		2^2				2^3							
2.4	2^0	2^0	2^1		2^2				2^3							
2.5	A: 2^0	2^0	2^1		2^2				2^3							
3	A: 2^0	2^0	B: 2^1		2^2				2^3							
4	A: 2^0	C: 2^0	B: 2^1		2^2				2^3							
5.1	A: 2^0	C: 2^0	B: 2^1		2^1		2^1		2^3							
5.2	A: 2^0	C: 2^0	B: 2^1		D: 2^1		2^1		2^3							
6	A: 2^0	C: 2^0	2^1		D: 2^1		2^1		2^3							
7.1	A: 2^0	C: 2^0	2^1		2^1		2^1		2^3							
7.2	A: 2^0	C: 2^0	2^1		2^2				2^3							
8	2^0	C: 2^0	2^1		2^2				2^3							
9.1	2^0	2^0	2^1		2^2				2^3							
9.2	2^1		2^1		2^2				2^3							
9.3	2^2				2^2				2^3							
9.4	2^3								2^3							
9.5	2^4															

1. The initial situation.
2. Program A requests memory 34 K.
3. Program B requests memory 66 K.
4. Program C requests memory 35 K.
5. Program D requests memory 67 K.
6. Program B releases its memory.
7. Program D releases its memory.
8. Program A releases its memory.
9. Program C releases its memory.

https://en.wikipedia.org/wiki/Buddy_memory_allocation

Lazy Allocation

- Swap space is reserved dynamically as the system needs to reclaim physical memory
- Rather than having to allocate it in advance for every page of anonymous memory
  - That is, memory devoted to the stack and heap of a process, and to data that is not file-backed
- When the list of active pages falls below limit and OSF memory manager attempts to reclaim pages for the free list by paging out virtual pages, if the available swap space has already been exhausted, the OSF memory manager does not back off of the page-out and instead simply discards the page
  - Thus, when the process whose page has been discarded takes a page fault and attempts to reactivate the missing page, unpredictable behavior results

A Review on Memory

Memory

What is memory

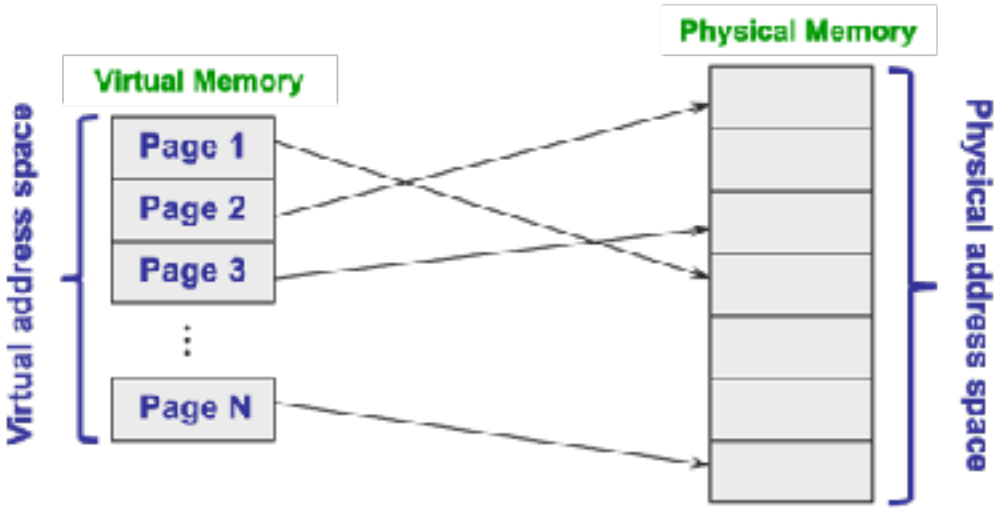
- From programmers' perspective
    - A "place" to store data
- How to access data in memory?
    - Variables?
    - Names?
    - Addresses?
- Memory can be viewed as a big array
    - content = memory[address]

Address Spaces

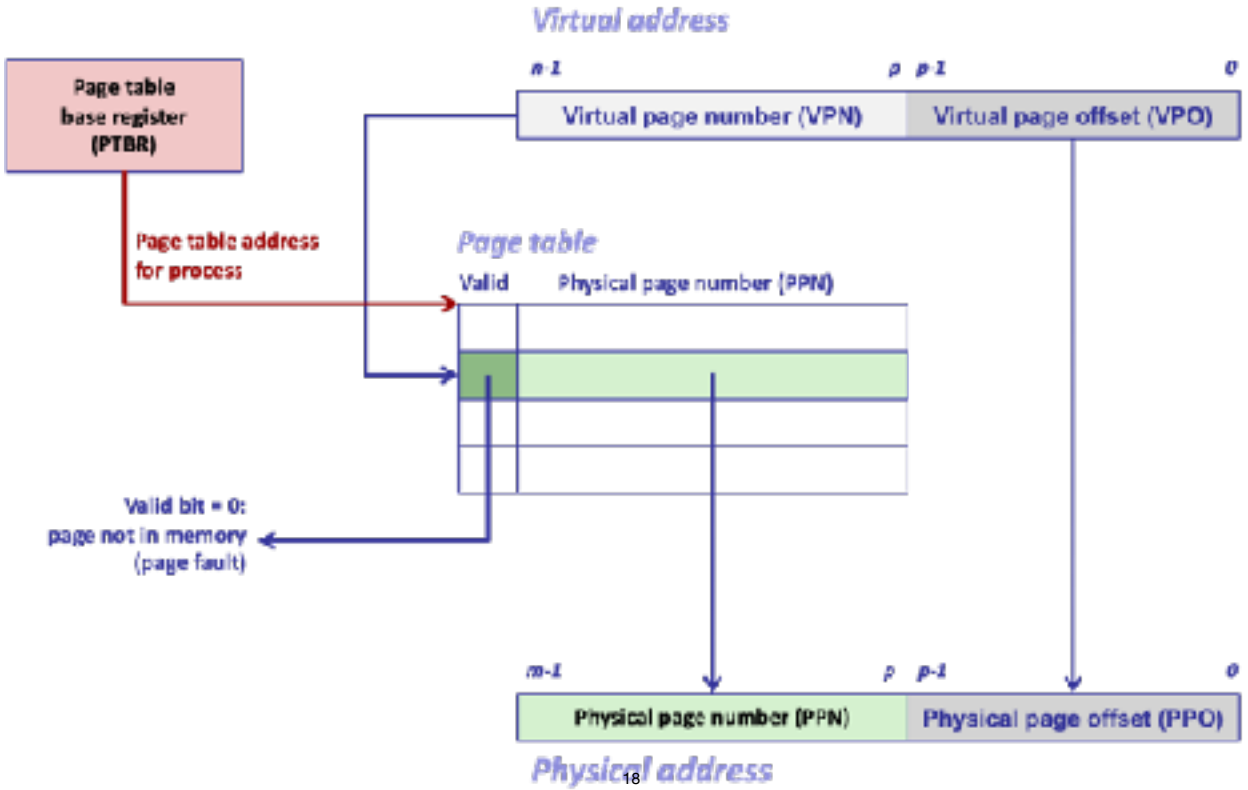
- Address space: ordered set of non-negative integer addresses that can be used to access memory: {0, 1, 2, 3 … }
    - Addresses could be contiguous or segmented
- Virtual address space: set of virtual addresses
- Physical address space: set of physical addresses

Paging

- Split the virtual and physical address space into multiple fixed size partitions (i.e., pages), each virtual page can be translated to any physical page



Address Translation with Page Table



Page Table Entry

When page fault can happen?

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

CD: Caching disabled or enabled for the child page table.

A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size either 4 KB or 2 MB (defined for Level 1 PTEs only).

G: Global page (don't evict from TLB on task switch)

Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

XD: Non-executable pages

Page Fault Handling

OS-induced page faults

- Lazy allocation
    - Aggressive lazy allocation
- Copy-on-Write
- Virtual memory

Lab 2

Overview

- We replaced the page allocator in the xv6 kernel with a buddy allocator
- 1) Modify xv6 to use this allocator to allocate and free file structs so that xv6 can have more open file descriptors than the existing system-wide limit NFILE
- 2) Make memory allocation to be done in a lazy manner

Task 1: Heap Allocator

- xv6 has only a page allocator and cannot dynamically allocate objects smaller than a page
  - To work around this limitation, xv6 declares objects smaller than a page statically
  - E.g, xv6 declares an array of file structs, an array of proc structures, and so on
  - As a result, the number of files the system can have open is limited by the size of the statically declared file array, which has NFILE entries (see kernel/file.c and kernel/param.h).
- The solution
  - Adopt the buddy allocator, which we have added to xv6 in kernel/buddy.c and kernel/list.c.
- Your job
  - Further improve xv6's memory allocation: modify kernel/file.c to use the buddy allocator so that the number of file structures is limited by available memory rather than NFILE.

Task 2: Lazy Page Allocation

- One tricks with page table hardware is lazy allocation of user-space heap memory
- Xv6 applications ask the kernel for heap memory using the sbrk() system call
  - In the kernel we've given you, sbrk() allocates physical memory and maps it into the process's virtual address space.
  - There are programs that use sbrk() to ask for large amounts of memory but never use most of it, e.g. large sparse arrays.
- To optimize for this case, sophisticated kernels allocate user memory lazily. That is, sbrk() doesn't allocate physical memory, but just remembers which addresses are allocated.
  - When the process first tries to use any given page of memory, the CPU generates a page fault, which the kernel handles by allocating physical memory, zeroing it, and mapping it.
  - sYou'll add this lazy allocation feature to xv6 in this lab.

Additional Information

- Background:
  - xv6 book: Chapter 2, Chapter 3, and Chapter 4
  - MIT lecture notes: https://pdos.csail.mit.edu/6.828/2022/schedule.html
    - programming xv6 in C
    - OS design
    - Virtual Memory/Page tables
    - Page faults
- Grading specification:
  - Task 1, pass filetest: 5 points.
  - Task 2, pass lazytests: 10 points.
  - Task 2, pass usertests: 5 points.
- TA will run **ALL** student's git diff, patch and verify the result.

Grading

- You will get points:
  - Have a **meaningful** report
    - **screenshots** of the grading script; What tests passed and did not pass; Your username.
    - **explanation** of your code showing your understanding (comments)
  - **Valid** git diff file. Function level code comments
- You will lose points:
  - Do not follow the above instructions
  - Fail on some tests
  - Implement something else not asked in the requirement
- You will **not get any** points:
  - Direct plagiarism
  - The results reproduced by TA do not match what you present from your report
  - Invalid git diff file
  - Invalid submission. Please **upload two files**. report.pdf + mycode.diff