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

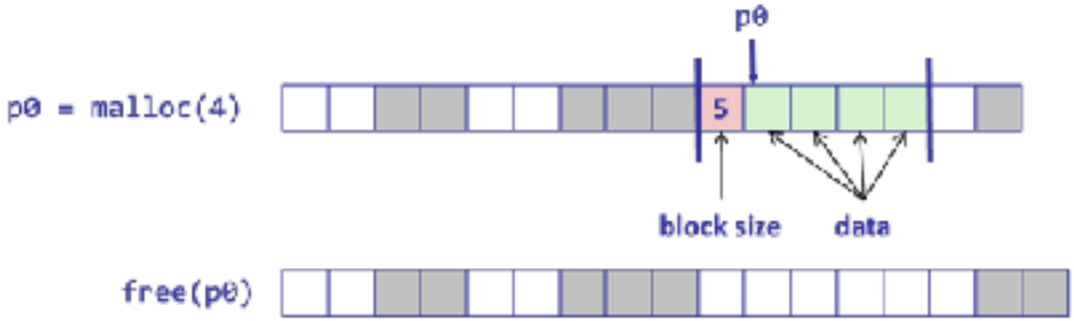
- Throughput: number of completed allocation/free requests per unit time
    - We don't want the heap allocator to become the performance bottleneck
- Memory utilization: total memory size required to fulfill the requests
    - Fragmentations and poor allocation policies can cause poor memory utilization

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

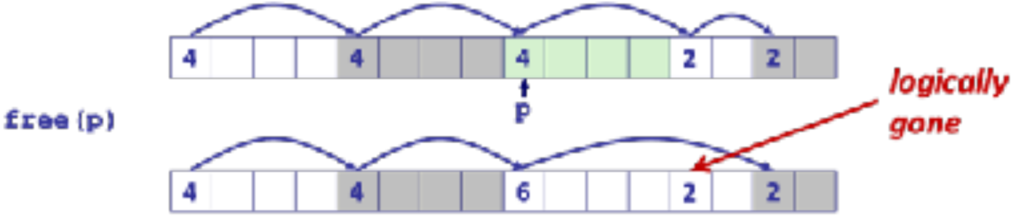
Which Block to Pick

- 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

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

A Review on Memory

Memory

What is memory

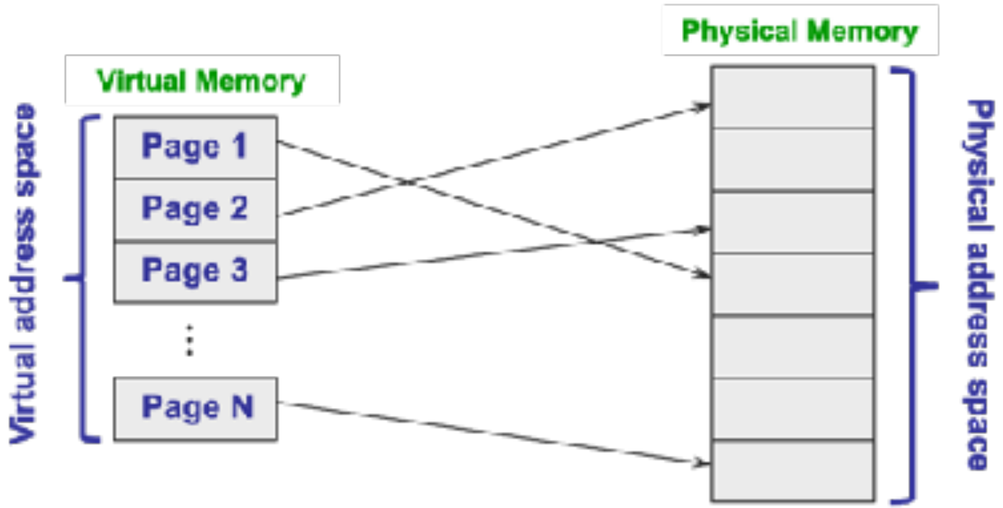
- From programmers’ perspective
    - A “place” to store data
- How to access data in memory?
    - Variables?
    - Names?
    - Addresses?
- Memory can be viewed as a big array
    - content = memory[address]

Address Spaces

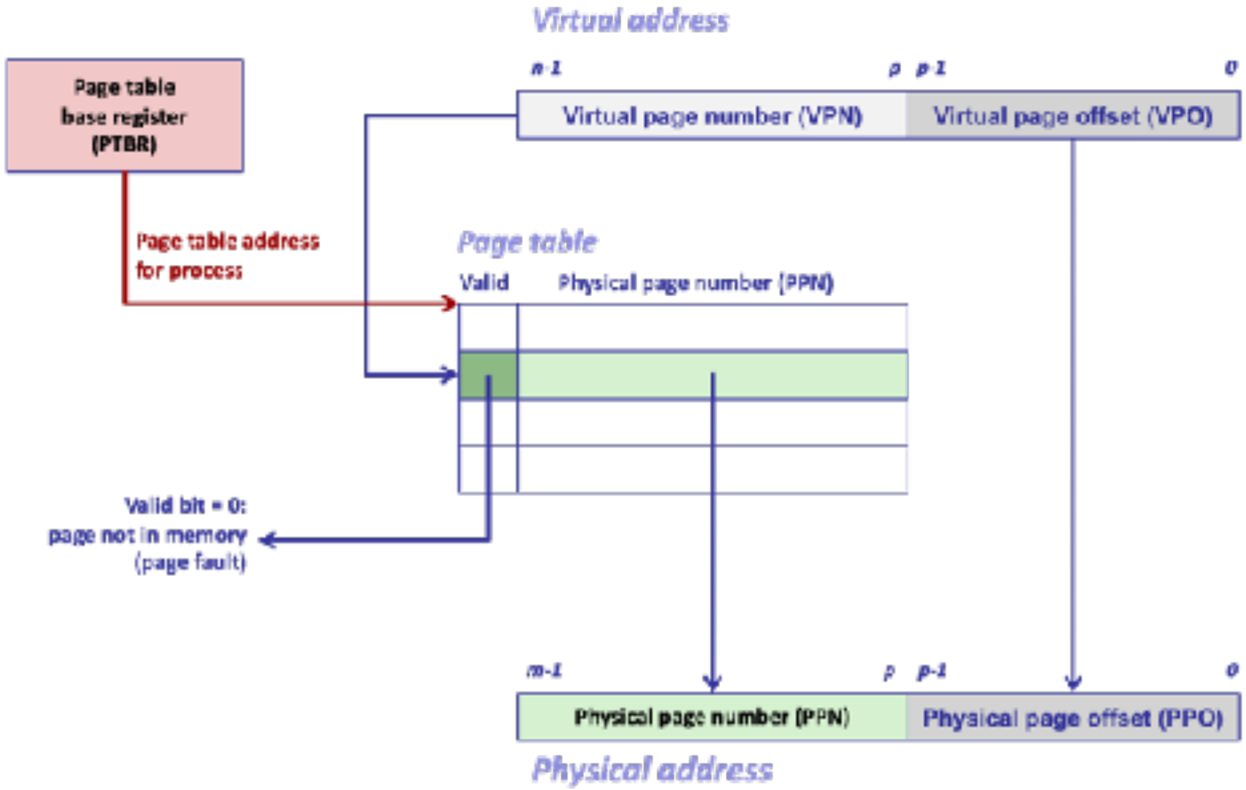
- Address space: ordered set of non-negative integer addresses that can be used to access memory: {0, 1, 2, 3 … }
    - Addresses could be contiguous or segmented
- Virtual address space: set of virtual addresses
- Physical address space: set of physical addresses

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



What’s wrong in pgbug?

```c
int
copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
  uint64 n, va0, pa0;

  while(len > 0){
    va0 = (uint)PGROUNDDOWN(srcva);
    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (srcva - va0);
    if(n > len)
      n = len;
    memmove(dst, (void *)(pa0 + (srcva - va0)), n);

    len -= n;
    dst += n;
    srcva = va0 + PGSIZE;
  }
  return 0;
}
```

```c
void
pgbug(char *s)
{
  char *argv[1];
  argv[0] = 0;
  exec((char*)0xeaeb0b5b00002f5e, argv);

  pipe((int*)0xeaeb0b5b00002f5e);

  exit(0);
}
```

Page Table Entry

When page fault can happen?

63	62 — 52	51 — 12	11 — 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page table physical base address	Unused	G	PS		A	CD	WT	U/S	R/W	P=1

Available for OS (page table location on disk)	P=0

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

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