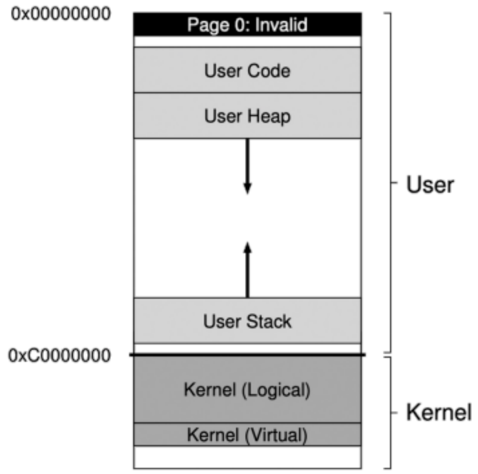
Lecture9 Linux Memory Management

1. Address Space in Linux



- The virtual address space of each process is split between **user and kernel portions**
    - user virtual addresses: 0 - 0xBFFFFFFF
        - Page 0 is invalid to detect **NULL pointers**
    - kernel's virtual addresses: 0xC0000000 - 0xFFFFFFFF
- 64 -bit Linux has a similar split but at slightly different points
- Why is kernel memory mapped into the address space of **each process**?
    - No need to change page table (i.e., switch CR3) when trapped into the kernel – no TLB flush
    - Kernel code may access user memory when needed
- The kernel memory in each address space is **the same**

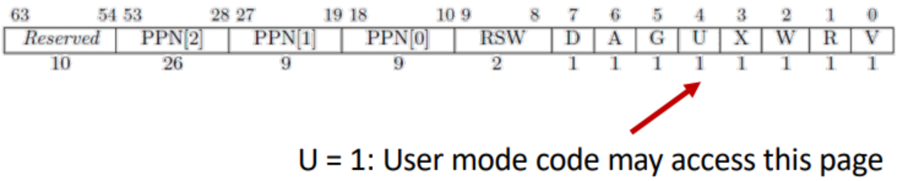
Kernel Space

- Kernel logic addresses
    - Most kernel structures
        - page tables
        - per-process kernel stacks
        - `kmallpc()`, never swapped out
    - Starts with 0xC0000000, always map to **continuous physical address** starting from 0x00000000

- Easy for DMA or other devices that requires continuous physical memory
- Kernel virtual addresses
    - virtually continuous memory
    - `vmalloc()`

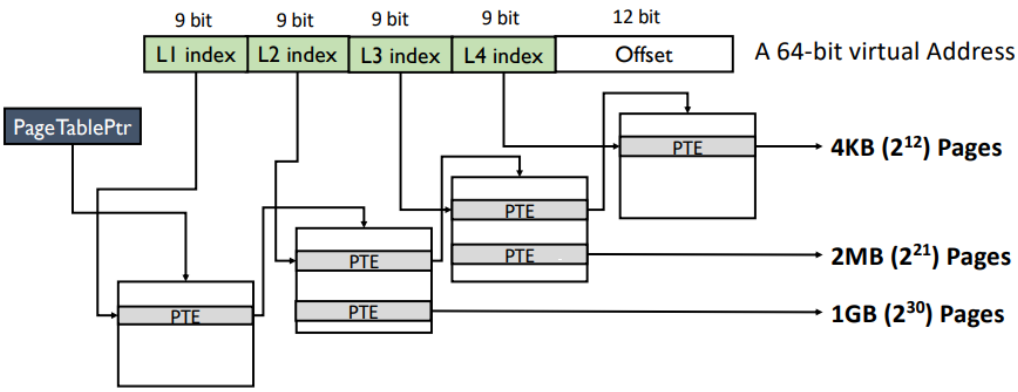
Isolation

- Isolation between processes
    - Not the same address space
- Isolation between user process and kernel
    - How to protect kernel space from user process: Page table permission bits



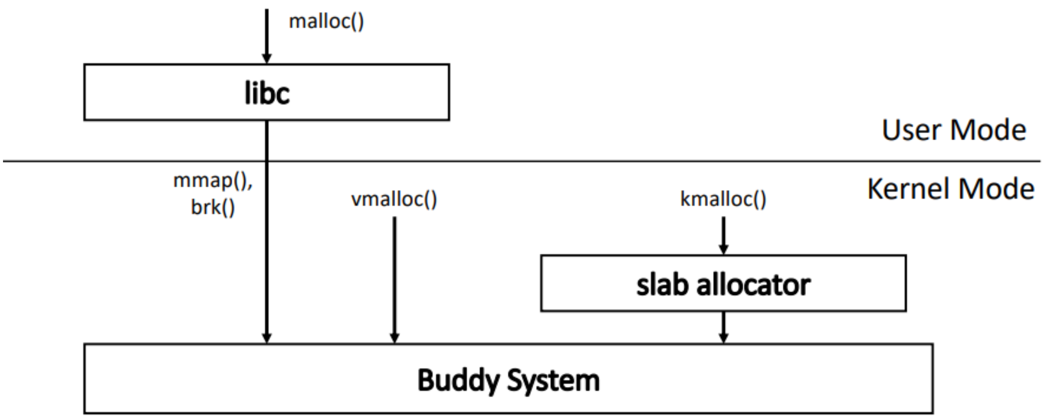
Large Page Support

- x86 support 4KB, 2MB, 1GB pages
    - Hardware enforces page alignments
    - 4KB pages are 4KB aligned (lower 12 bits are 0)
    - 2MB pages are 2MB aligned (lower 21 bits are 0)
    - 1GB pages are 1GB aligned (lower 30 bits are 0)
- Linux also adds supports to **huge page** (Linux term)
    - Fewer TLB misses
    - Applications may need **physically continuous** physical memory
    - Leads to **internal fragmentation**

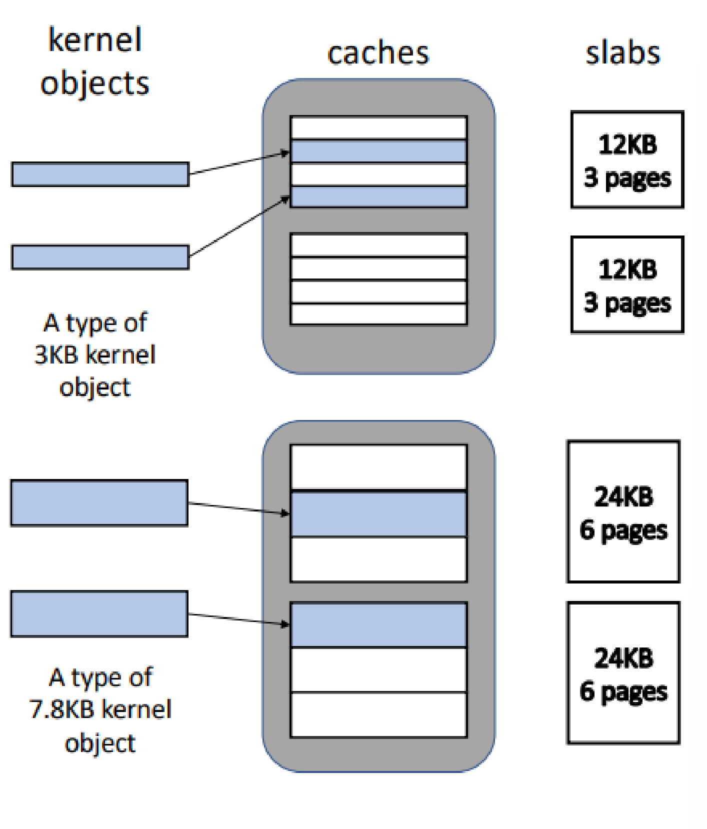


- Different page size uses different level of page tables

2. Linux Physical Memory Management



Slab Allocator



- A **slab** consists of one or more **physically contiguous pages**
- A **cache** consists of one or more slabs
  - One cache for each type of **kernel objects**
- When a slab is allocated to a cache, **objects** are initialized and marked as **free**
- Slab states:
  - **empty**: all objects are free
  - **partial**: some objects are free
  - **full**: all objects are used

- A request is first served by a **partial** slab, then **empty** slab, then a **new** slab can be allocated from **buddy system**
- No memory is wasted due to fragmentation
  - when an object is requested, the slab allocator returns the **exact amount of memory** required to represent the object
- Memory requests can be satisfied quickly
  - Objects are created and initiated in advance
  - Freed object is marked as **free** and immediately available for subsequent requests

Buddy System

- Free physical memory is considered big space of size 2^N pages
- **Allocation**: the free space is divided by two until a block that is big enough to accommodate the request is found
  - a further split would result in a space that is too small
- **Free**: the freed block is recursively merged with its buddy
  - Two buddy blocks have physical addresses that differ only in 1 bit

Example

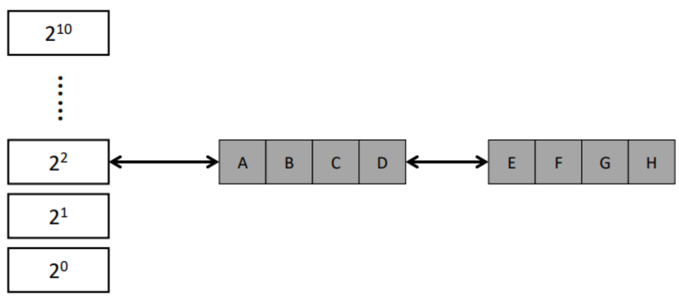
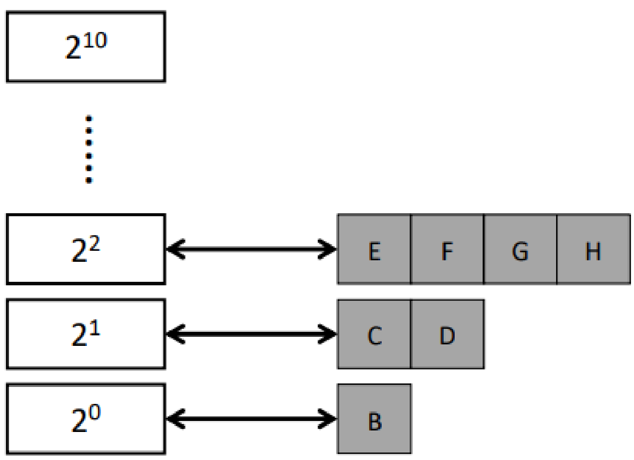
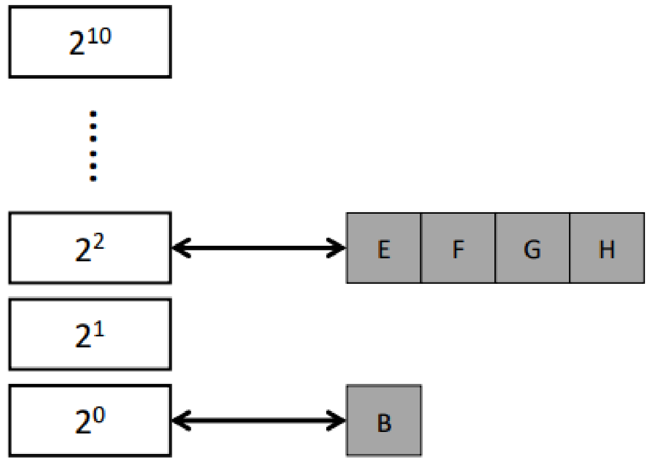
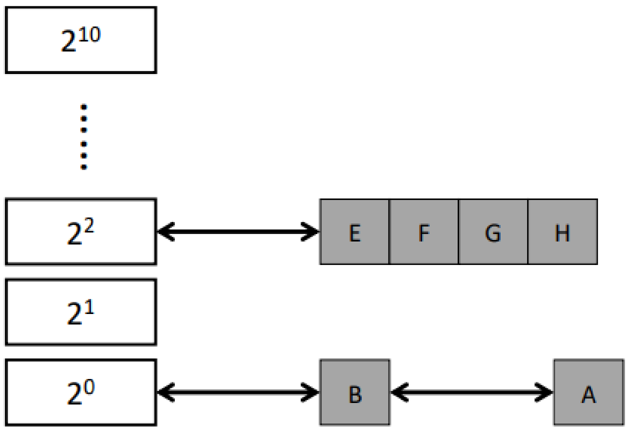
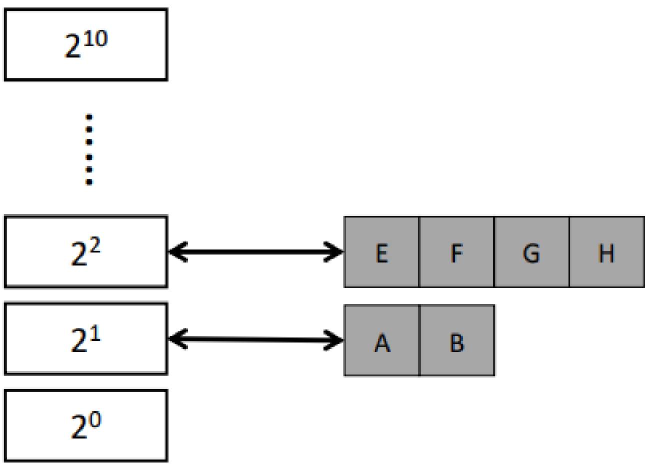
Illustration	Description
	1 block: 4KB
	Page A allocated Request 5KB Memory

Illustration	Description
	Page C and D allocated Page A freed
	Page A linked to 2^0 Check if Buddy is free
	Merge and move to 2^1

Page Cache

- Page cache: an area of **physical memory** to hold data that are stored on a hard disk or other permanent storage
  - memory-mapped files: all binaries and dynamic libraries
  - anonymous memory: stacks and heaps that are stored in swap space
- The page cache tracks if entries are clean or dirty

- Dirty pages are **periodically written back** to disk (pdflush)

- Page replacement policy for page cache
  - Use 2Q replacement
  - LRU may perform poorly in certain corner cases

2Q replacement policy

- Two lists: inactive list and active list (LRU queues)
  - When accessed for the first time, a page is placed in inactive list
  - When it is re-referenced, the page is promoted to the active list
- Replacement takes place in the inactive list
  - Linux periodically moves some page from the bottom of active list to inactive list, keeping active list 2/3 of the size of page cache
  - Clock algorithm used to approximate LRU

Data Execution Prevention (DEP)

- Buffer overflow is a well-known software vulnerability
  - Attacker provides input to an application (possibly from remote
  - After a stack overflow, a function is returned to instructions on the stack (also provided by the attacker
- DEP is a **security feature** that prevents data pages to be executed by software
  - A page is either writable or executable (NX bit in PTE)
  - Stacks are not executable

Address Space Layout Randomization

```
int main(int argc, char *argv[]) {
    int stack = 0;
    printf("%p\n", &stack);
    return 0;
}
```

```
prompt> ./random
0x7ffd3e55d2b4
prompt> ./random
0x7ffe1033b8f4
prompt> ./random
0x7ffe45522e94
```

- Return-to-libc attacks and its successor - Return Oriented Programming (ROP) attacks
  - Stack overflow leads to returns to functions in libc or gadgets in libc
  - Gadgets are short code snippets that can be chained together
- Address space layout randomization (ASLR)
  - **Randomize virtual address of stacks, heaps, and libraries so libc and stack/heap address are not known**