

Lecture 9

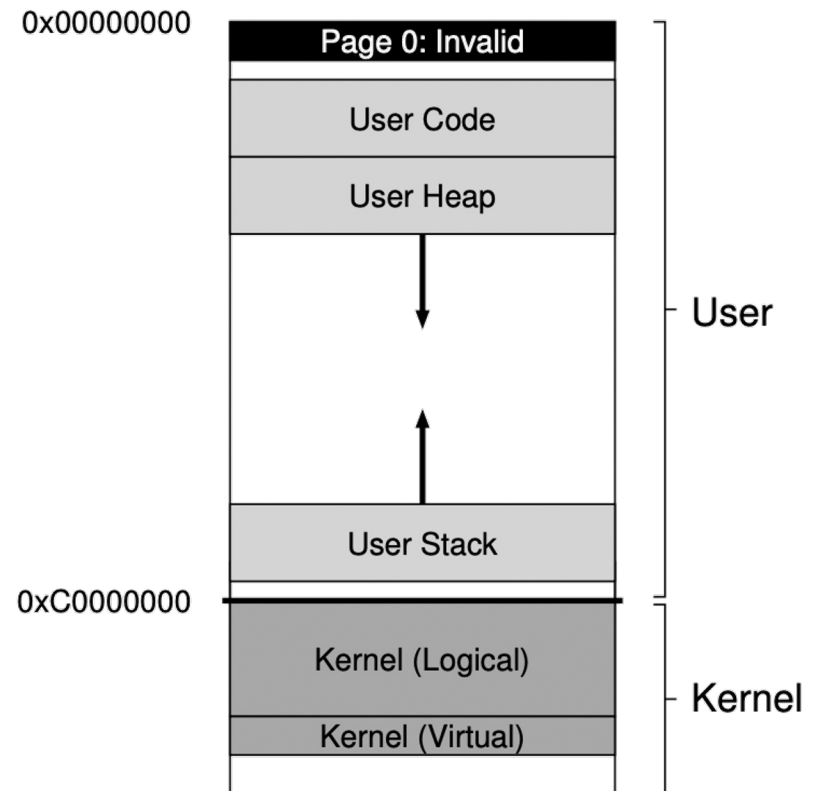
Linux Memory Management

Prof. Yinqian Zhang

Spring 2023

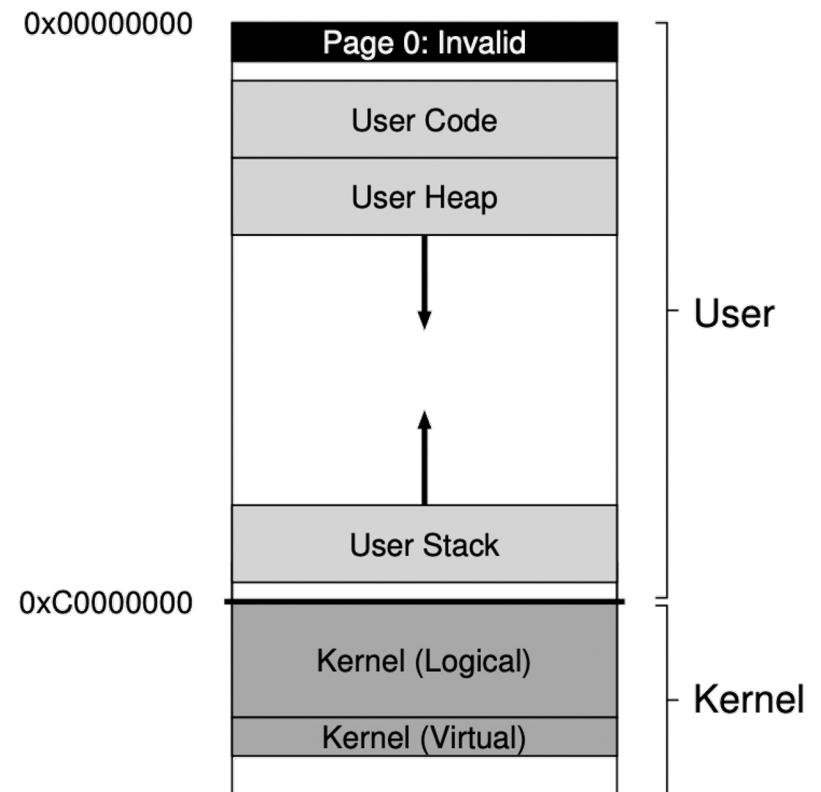
Address Space in Linux

- The virtual address space of each process is split between user and kernel portions
 - Virtual addresses 0 through 0xBFFFFFFF are user virtual addresses
 - Page 0 is invalid to detect NULL pointers
 - 0xC0000000 through 0xFFFFFFFF are in the kernel's virtual address space.
- 64-bit Linux has a similar split but at slightly different points.

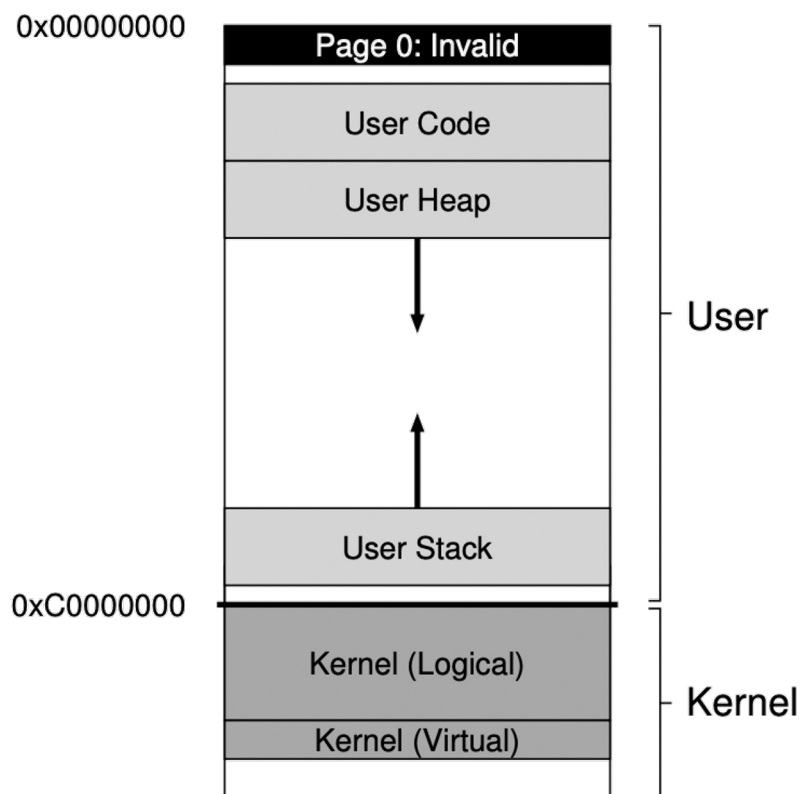


Address Space in Linux (Cont'd)

- 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
 - system call, interrupts, exception
 - Kernel code may access user memory when needed
- The kernel memory in each address space is the same

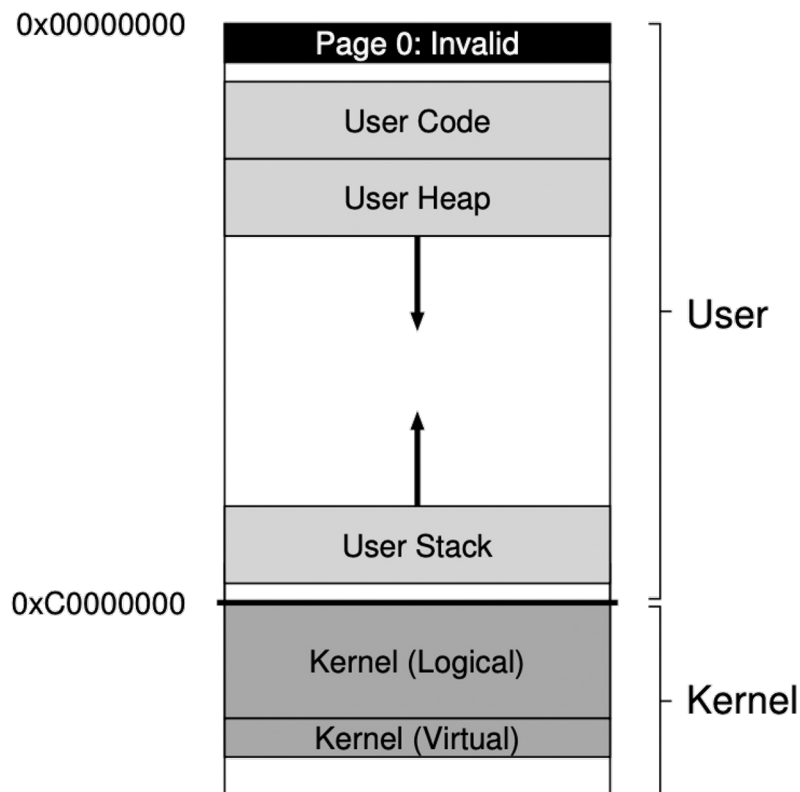


User Space and Kernel Space



- **Kernel logic addresses** 虚拟与物理
 - Most kernel data structures 都连续.
 - page tables
 - per-process kernel stacks
 - **kmalloc()**, 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()**

User Space and Kernel Space



- 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

63	54 53	28 27	19 18	10 9	8	7	6	5	4	3	2	1	0
<i>Reserved</i>	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V	
10	26	9	9	2	1	1	1	1	1	1	1	1	

U = 1: User mode code may access this page

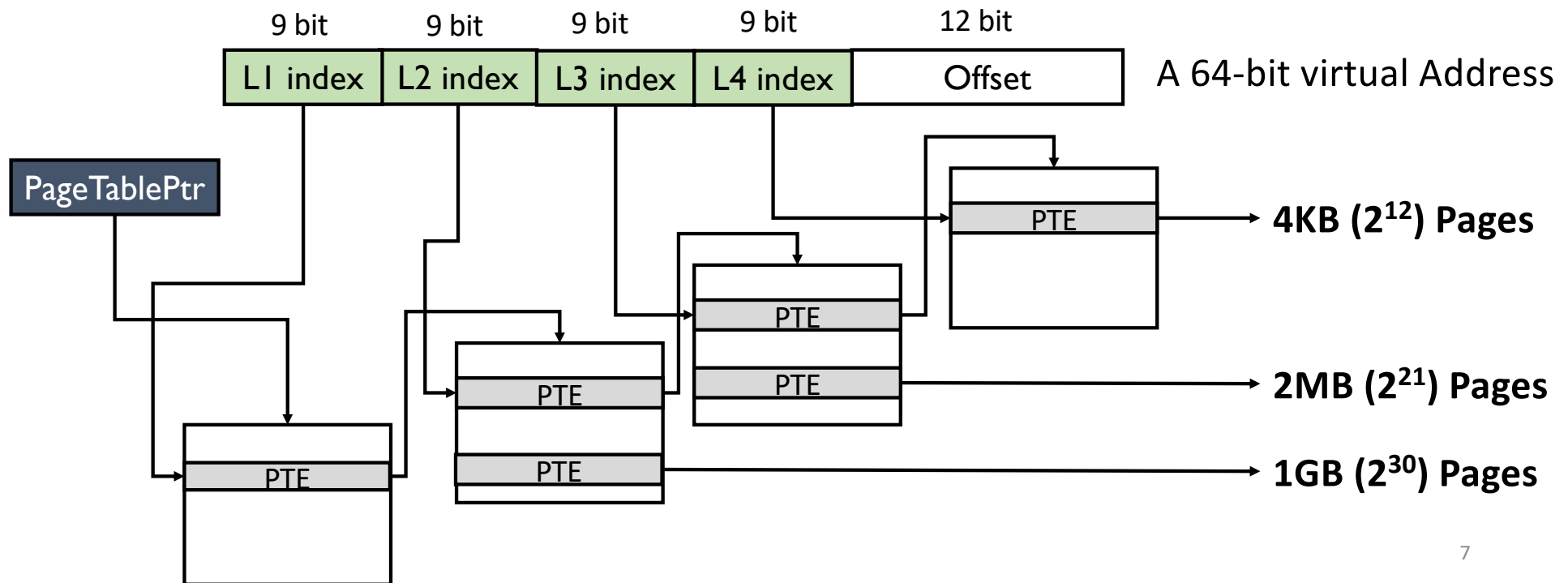
Large Page Support

- 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

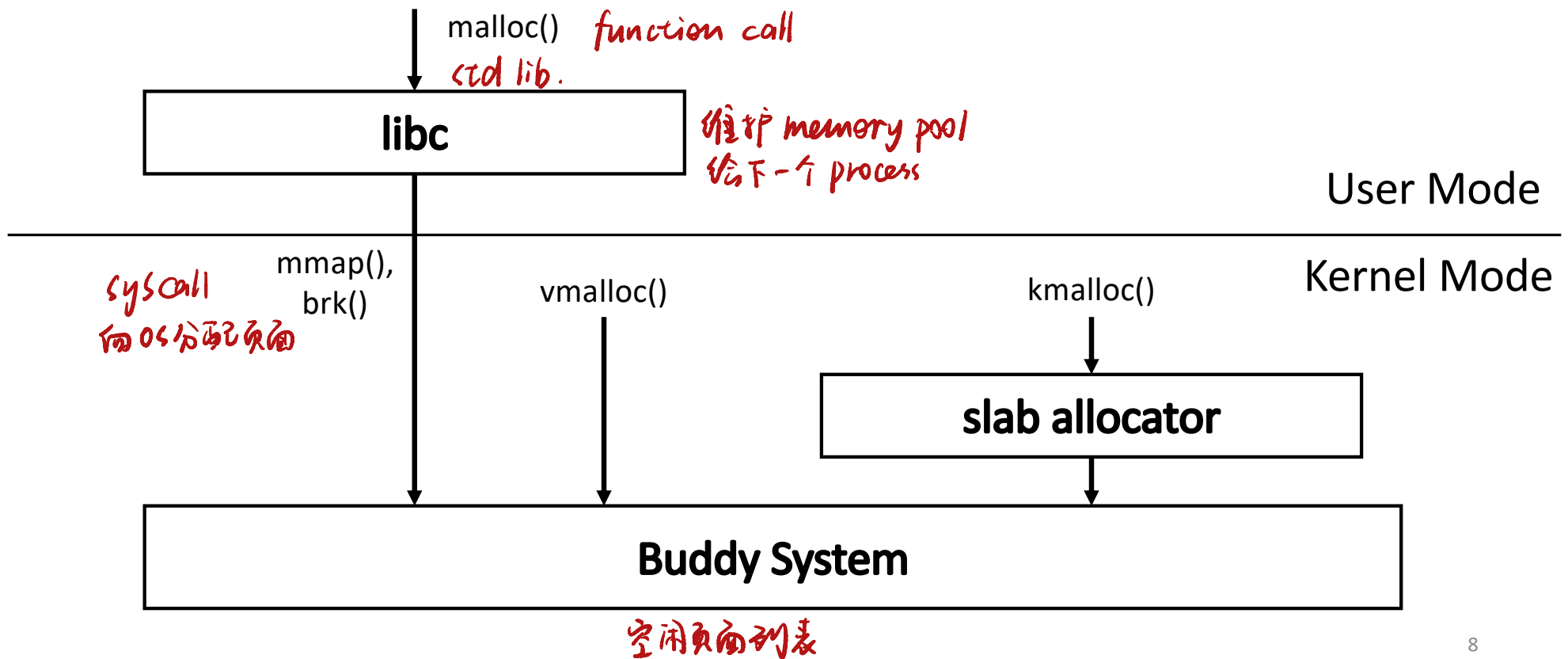
64位
↓ +9 bit index - 1 level

Large Page Support

- Different page size uses different level of page tables



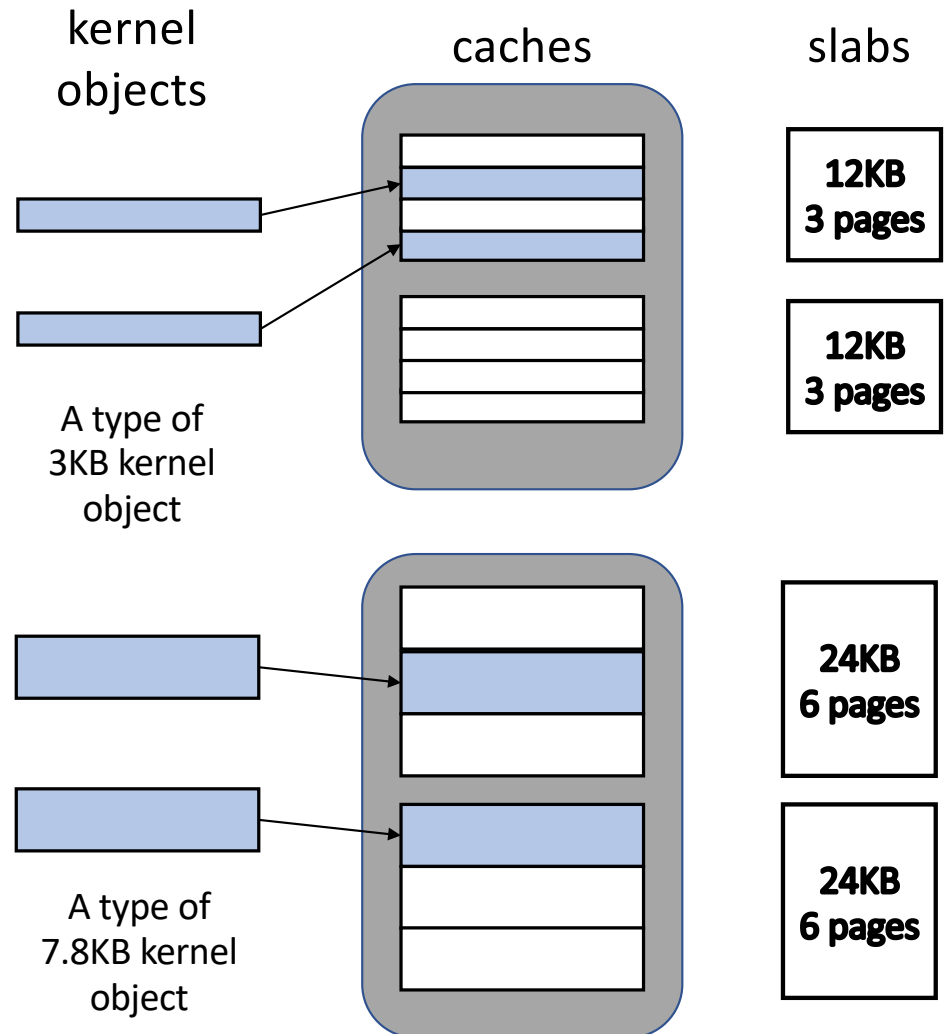
Linux Physical Memory Management



Slab Allocator

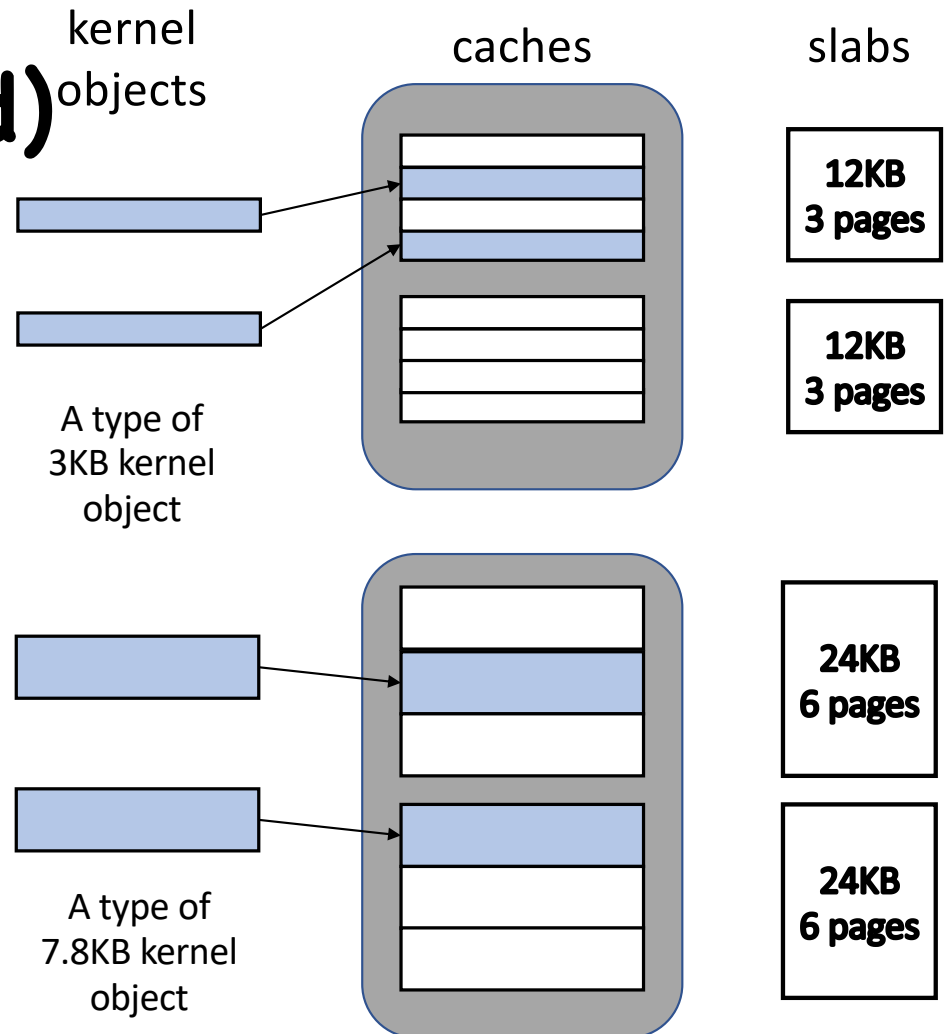
避免跨页的 struct

- A **slab** consists of one or more physically contiguous pages 物理连续页面
- A **cache** consists of one or more slabs
 - One cache for each type of kernel objects



Slab Allocator (Cont'd)

- When a slab is allocated to a cache, objects are initialized and marked as free
- A **slab** can be in one of the following 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**



Slab Allocator (Cont'd)

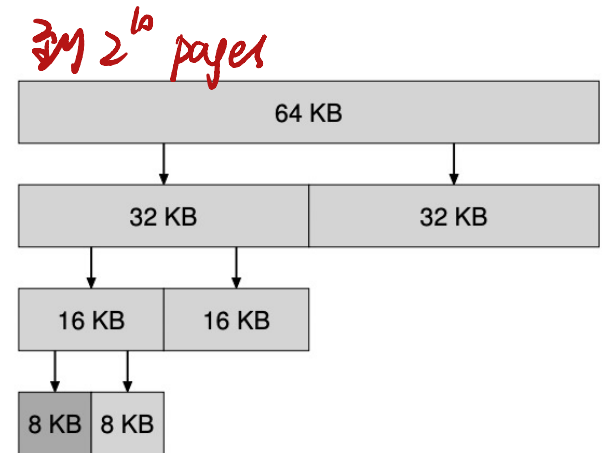
- No memory is wasted due to fragmentation
 - when an object is requested, the slab allocator returns the exact amount of memory required to represent the object
 - Objects are packed tightly in the slab
- Memory requests can be satisfied quickly
 - Objects are created and initiated in advance
 - Freed object is marked as free and immediately available for subsequent requests
- Later Linux kernel also introduces Slub allocator and Slob allocators.

Buddy System

获取连续物理内存

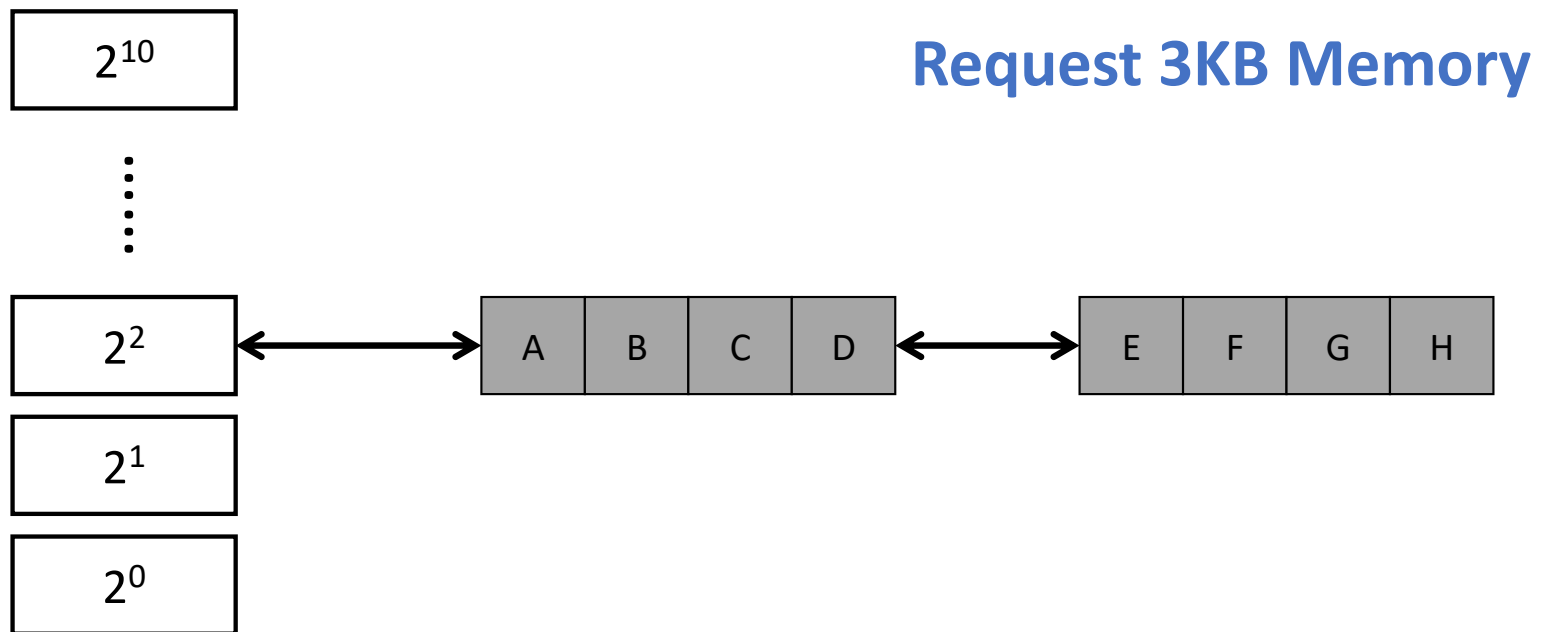
- Free physical memory is considered big space of size 2^N pages
- Allocation: the free space is divided by two until a block that is big enough to accommodate the request is found
 - a further split would result in a space that is too small
- Free: the freed block is recursively merged with its buddy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

维护
11个链表



[K65] "A Fast Storage Allocator" by Kenneth C. Knowlton.
Communications of the ACM, Volume 8:10, October 1965.

Buddy System Illustrated



Buddy System Illustrated

2^{10}

⋮

2^2

2^1

2^0

E F G H

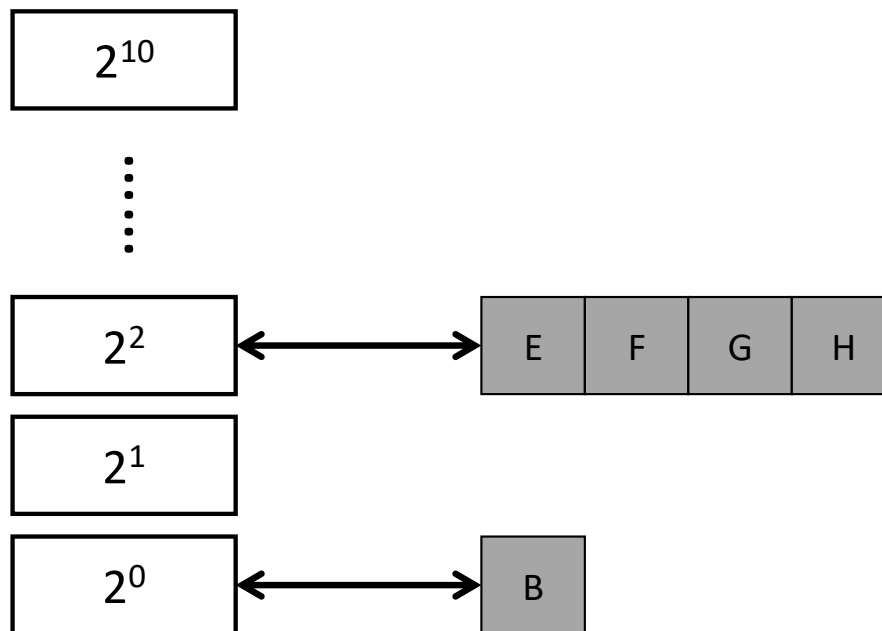
C D

B

Page A allocated

Request 5KB Memory

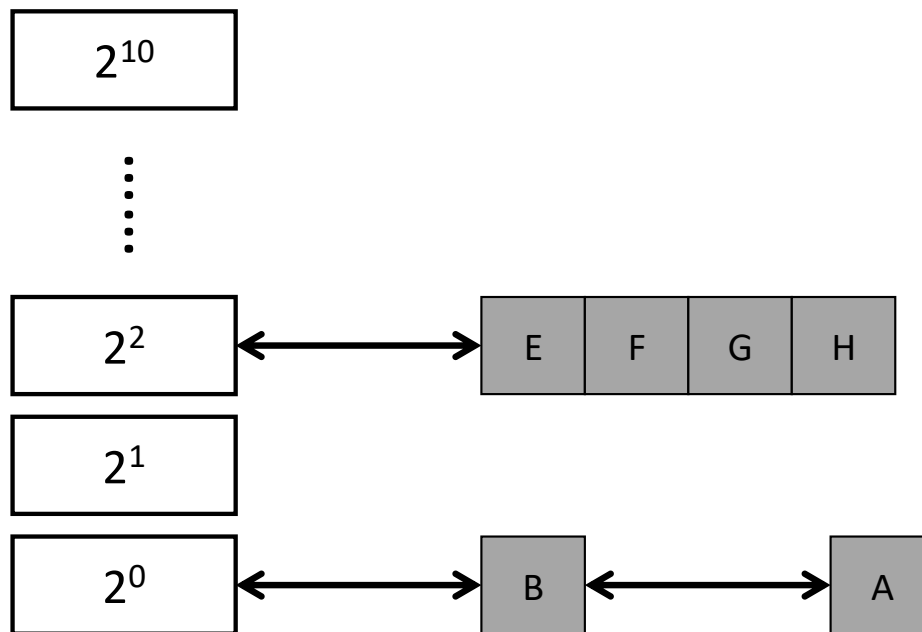
Buddy System Illustrated



Page C and D allocated

Page A freed

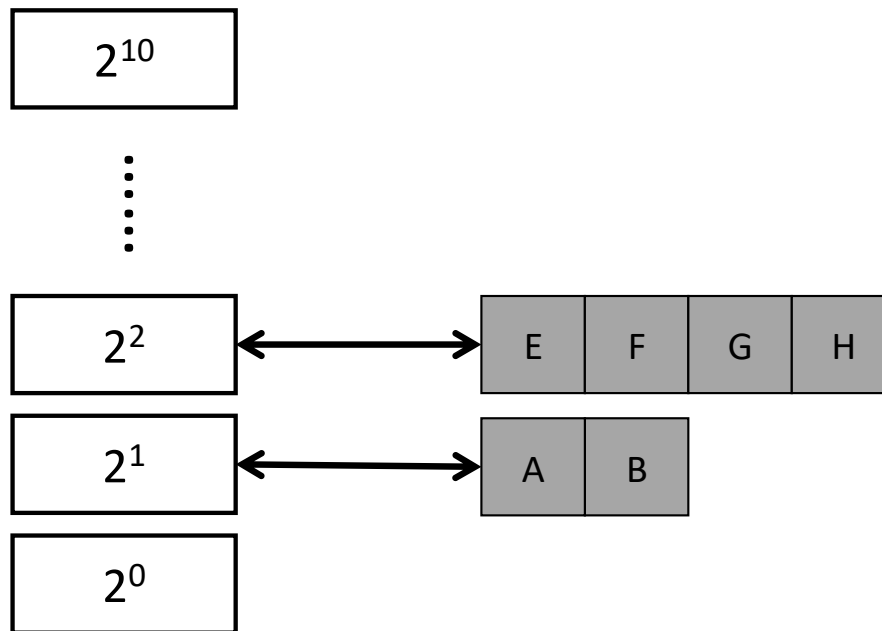
Buddy System Illustrated



Page A linked to 2^0

Check if Buddy is free

Buddy System Illustrated



Merge and move to 2^1

0 0
0 1
1 0
1 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)

```
0000000000400000    372K r-x-- tcsh
000000000019d5000  1780K rw--- [anon ]
00007f4e7cf06000  1792K r-x-- libc-2.23.so
00007f4e7d2d0000    36K r-x-- libcrypt-2.23.so
00007f4e7d508000   148K r-x-- libtinfo.so.5.9
00007f4e7d731000   152K r-x-- ld-2.23.so
00007f4e7d932000    16K rw--- [stack ]
```

Page Cache (Cont'd)

- Page replacement policy for page cache
 - Use 2Q replacement
 - LRU may perform poorly in certain corner cases
 - e.g., when a large file is accessed sequentially, all other files will be evicted
- 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 $\frac{2}{3}$ 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

```
int some_function(char *input) {  
    char dest_buffer[100];  
    strcpy(dest_buffer, input); // oops, unbounded copy!  
}
```

Address Space Layout Randomization

- 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

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

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

Thank you!

