Paging

- In the paging scheme, the process address space is not limited by the address of the RAM

- Each process virtual address space is (0 – 0xffffffff)

- OS creates a lookup table that is walked by the paging hardware during runtime to convert the virtual address into the physical address

Paging

VA	Page Table	PA
0	5	0
1	3	1
2	4	2
3	1	3
4	2	4
5	6	5
6	7	5
7	0	7

Unsigned Page table[8];

PA = pagetable[0];

PA = pagetable[4];

Paging

- The virtual and physical address spaces are divided into pages
- Pages are the contiguous area of memory of size 2^k
- Page addresses are 2^k byte aligned (i.e., the page address is always divisible by 2^k)

Paging

- Total number of virtual and physical pages: $2^{32} / 2^k$

- The page table keeps a mapping from virtual page number (VPN) to the physical page number (PPN)

- $VPN = VA / 2^k$ VA : virtual address

- $PPN = PA / 2^k$ PA : physical address

Paging

- Because the hardware walks the page tables at every memory access a simple hash table is used to convert the VPN to PPN
- The key to the hash table is the VPN itself

How large is the page table?

- k == 0 (PAGE SIZE = 1 byte)
  - Number of entries in the page table = 2^{32}
  - Requires a lot of space
- k == 26 (PAGE SIZE = 64 MB)
  - Number of entries in the page table = 2^{6}
  - Fragmentation

Page tables

- The typical page size is 4096 bytes

- Total number of virtual and physical pages = $(2^{32} / 2^{12}) = 2^{20}$

VA to PA

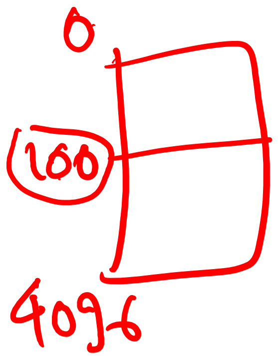
VA : 0x10100

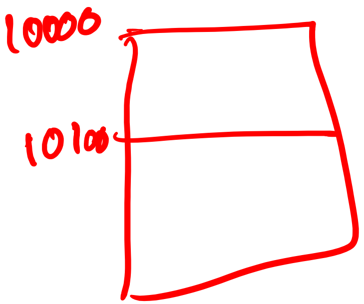
VPN = $VA / 2^{12}$ = $VA >> 12$ = 0x10

offset = $VA \% 2^{12}$ = $VA \& (2^{12} - 1)$ = 0x100

PPN = hash (VPN) 0x40

PA = $(0x40 \times 2^{12}) + 0x100$





Page table size = $2^{20} \times 2^{2}$

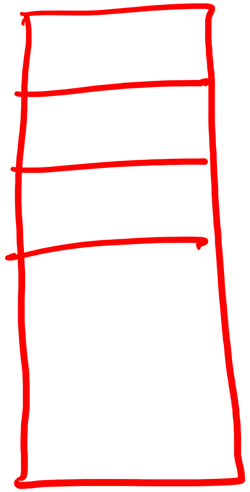
Paging

- 4 MB page table size is not good
  - Most of the applications use few KB’s of memory
  - a lot of VA – PA mappings does not exist for most processes

64 KB process

- Only 16 out of 2^{20} entries are used

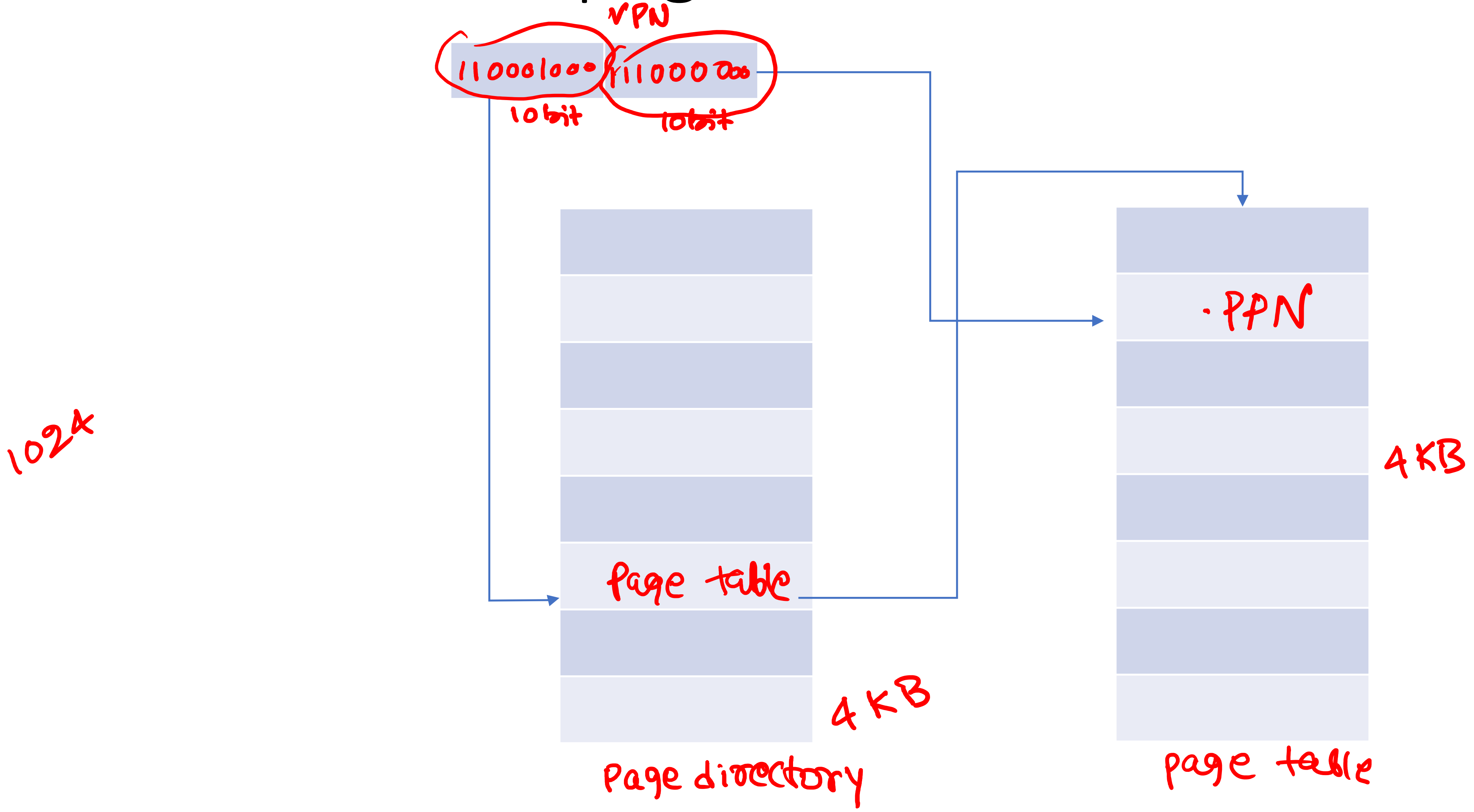
- One dimensional page table cause fragmentation



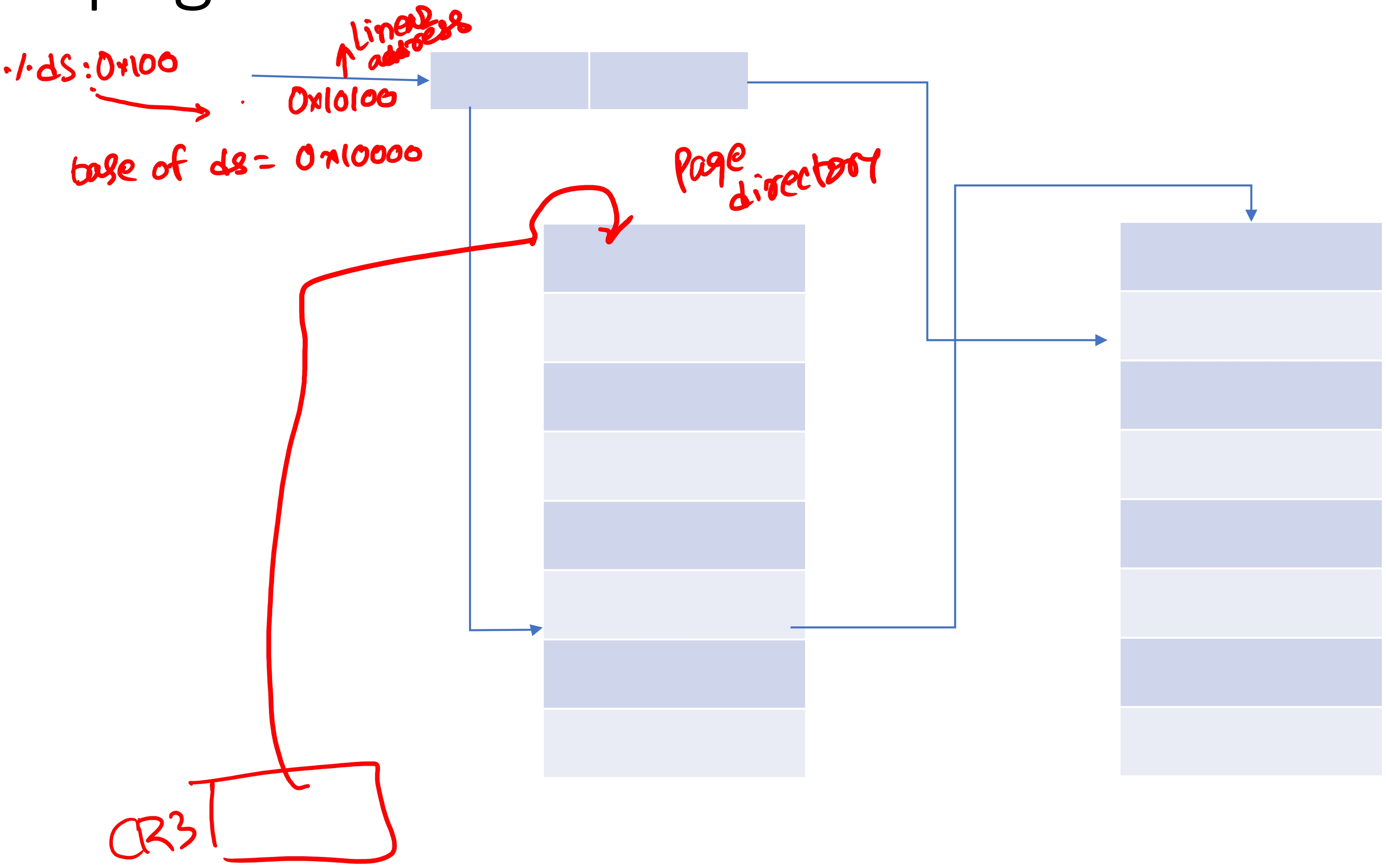
Two dimensional page table

- VPN is 20 bits
- Lookup is done in two steps
- The higher 10 bits are used to index into a table called the page directory
- A page directory entry contains the address of the page table
- The lower 10 bits are indexed into the page table to get the PPN

Two dimensional page table



x86 page table



Segmentation and paging

- In x86, segmentation is always enabled
- Paging is optional
  - Can be enabled by setting paging bit in %cr0 register
- Why disabling paging can also be a good idea?

Linear address

- When paging hardware is disabled the segmentation hardware converts VA to PA

- But, when paging is enabled, the segmentation hardware converts VA to linear address

- Paging hardware converts linear address to the physical address

Segmentation and paging

base of %ds = 0x10000

%ds:0x100

Linear address= 0x10100

VPN= 0x10

OFFSET= 0x100

PPN = Pagetable (0x10) = 0x40

PA = (0x40 << 12) | 0x100 = 0x40100

Segmentation and paging

- Most of the OS sets the base and limit of all segments to 0 and 0xffffffff

Where is page table stored?

- The page tables are stored in RAM

How does processor find the page directory?

- %cr3 register contains the base address of the page directory
- The OS creates the page table in memory and sets the %cr3 register to the base of the page directory

mov %eax, %cr3

Why can’t user programs load their own page table

- mov to %cr3 is a privilege instruction

Page table entry

- Page table entries are 32 bit long, whereas the PPN is only 20 bits long
- The extra 12 bits are used for other purposes

Page table entry

20 bits

PPN				D	A			U/S	R/W	P

P – Present
R/W – Read/Write
U/S – User/Supervisor
A – Access bit
D – Dirty bit

Memory isolation

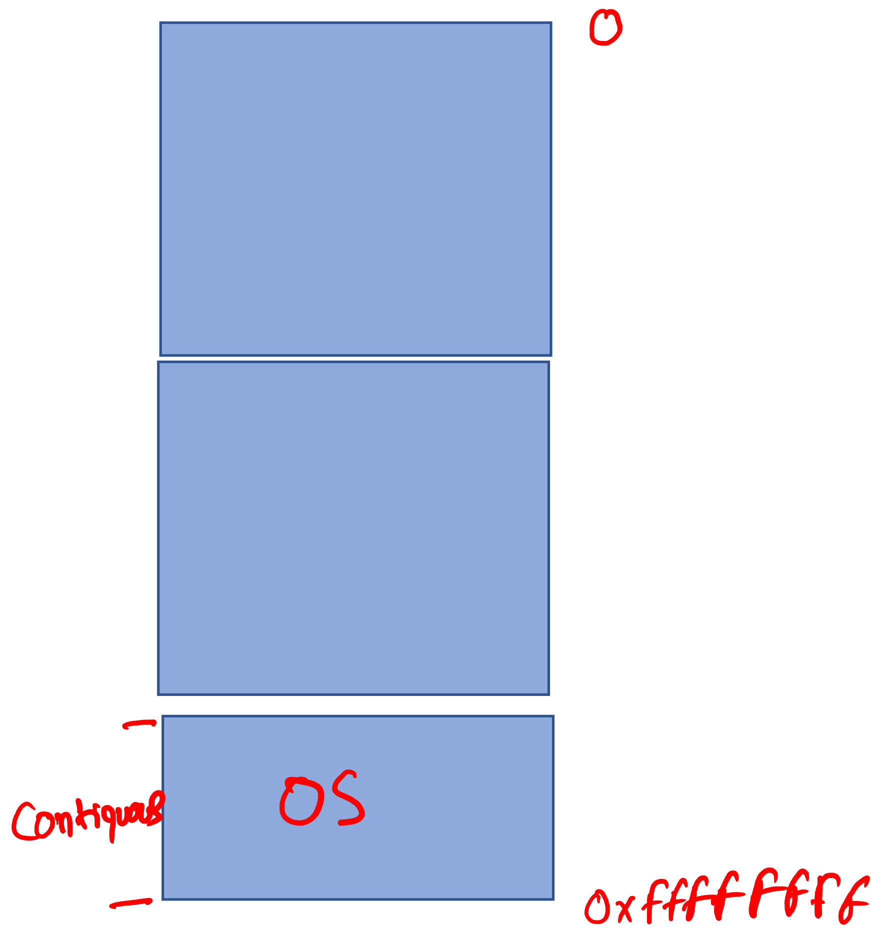
- OS creates a page table for every process
- OS uses different VA to PA mappings for different processes

Page sharing

- Interestingly, page sharing can be simply done using the same VA to PA mapping in different page tables

Where does OS live?

- OS maps itself into each process page tables
- OS pages are shared among all the processes



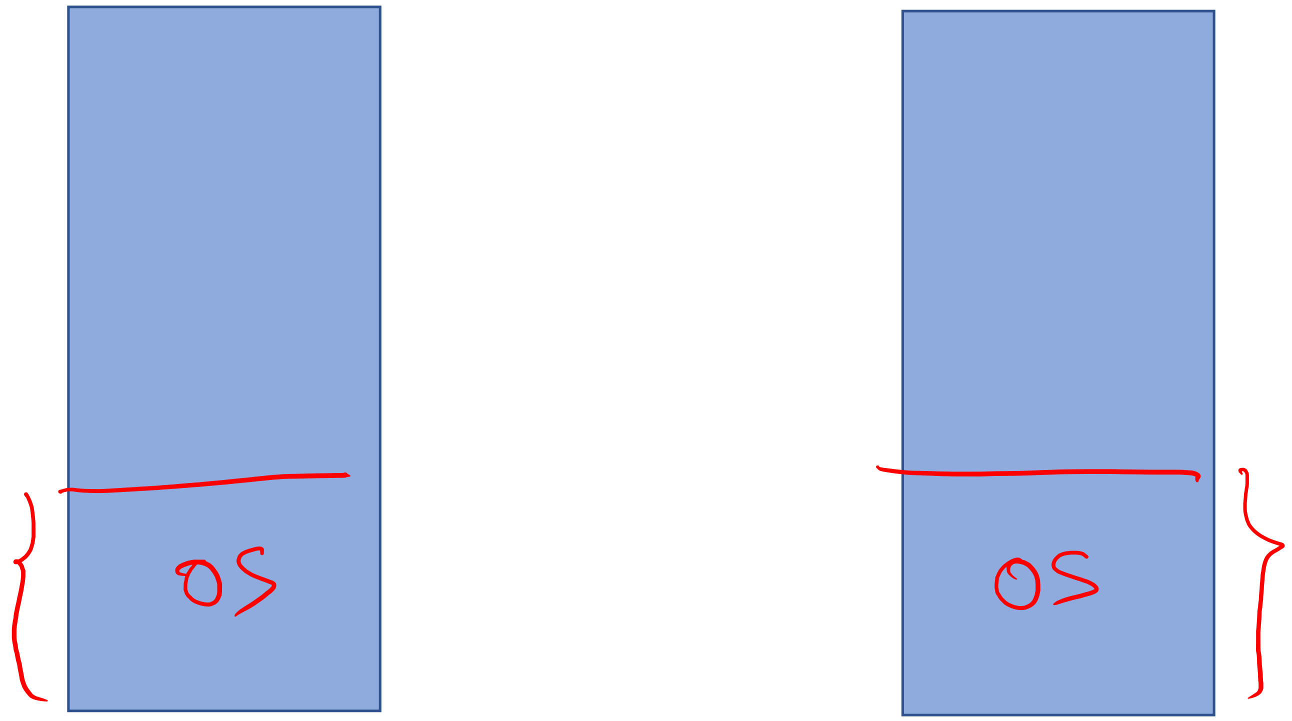
Why can user applications not access OS pages?

- Page table entries corresponding to OS pages are not marked as user pages

Why can user applications not modify the page table entries?

- Page table pages are only mapped in the kernel address space

Can OS be mapped at different virtual addresses in different processes?



What is the other alternative?

- Separate page table for kernel
- Only map kernel stacks and interrupt handlers wrappers in the user program
- The interrupt handler wrappers switch to the kernel page table before jumping to the original handler

Pros and cons of separate page tables for kernel

- Kernel cannot directly access the user pointers

- The user buffers are always copied to the kernel address space during system calls

char *buf = malloc (size);

read (fd, buf, size);

0x4000000

```
read (int fd, char *buf, size_t size);
```

Overhead of page table

- Hardware walks the page table on every memory access
  - Two additional memory access on every memory access
- To reduce the overhead of extra memory accesses, on-chip translation lookaside buffer (TLB) is used
- TLB caches VPN – PPN mappings
- Before walking the page tables, the hardware first looks into the TLB
- TLB accesses are very fast

TLB

- No additional memory accesses on TLB hit
- Good TLB hit rate (>99.9%) for most applications justifies the existence of page tables
- TLB hit rate depends on the current working set

TLB

- What happens if the OS modifies a page table entry which is cached in TLB?
  - TLB entries are not updated to the new value

- The hardware uses the cached entries in TLB even though the actual entries are modified

- The software can use “invlpg” instruction to invalidate a TLB entry

- The entire TLB is flushed when the OS reloads the %cr3 register

TLB flush

```
mov %cr3, %eax
mov %eax, %cr3
```

Large pages

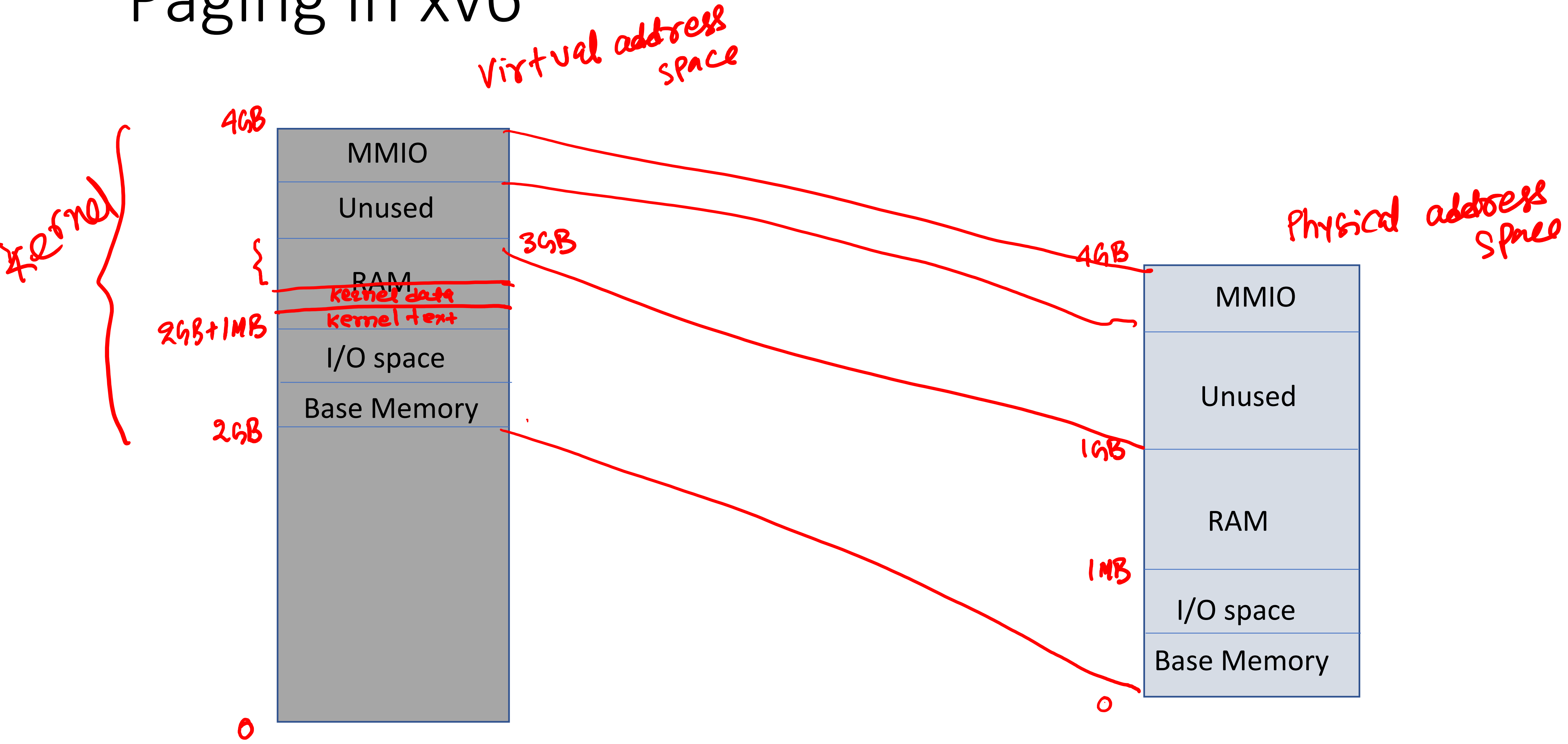
- 4 MB pages in x86

$2^2 \times 2^{20}$

22 bits

- Large pages reduce the TLB pressure

Paging in xv6



P2V and V2P

```
#define KERNELBASE 0x80000000 // 2GB
P2V (x)   (KERNELBASE + x)
V2P (x)   (x - KERNELBASE)
```

xv6 memory allocation

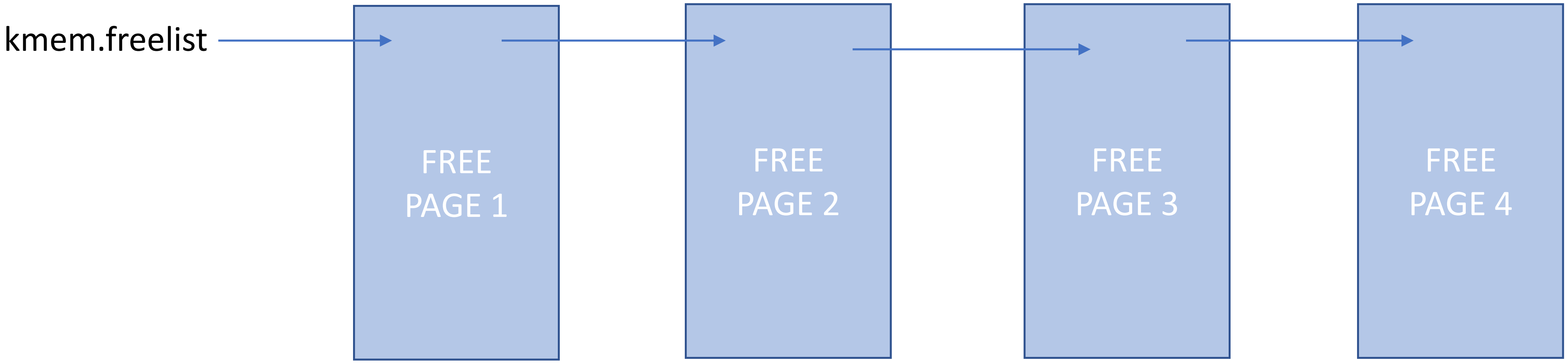
- kalloc() : allocates one virtual page
- kfree() : frees the virtual page allocated by kalloc
- kinit2(vstart, vend) : frees all the virtual pages between vstart to vend

kinit

- kinit2 (P2V(4*1024*1024), P2V(PHYSTOP))

- calls kfree for each virtual page in this range

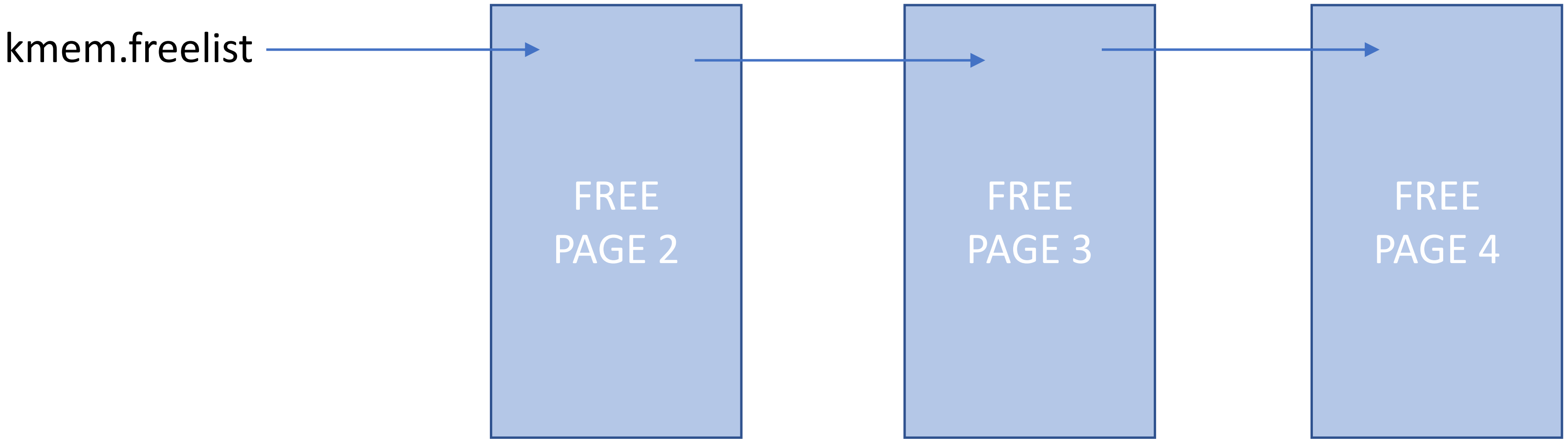
kfree




```
Kfree ( VA)
{
    VA.next = kmem.freelist;
    kmem.freelist = VA;
}
```

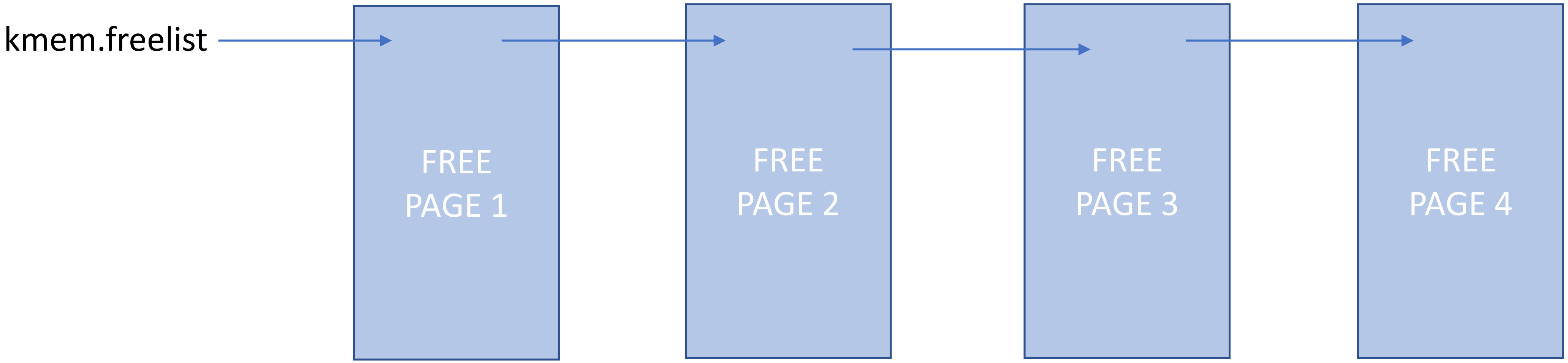
kalloc

- Return PAGE 1



kfree

- Add PAGE 1 back to freelist

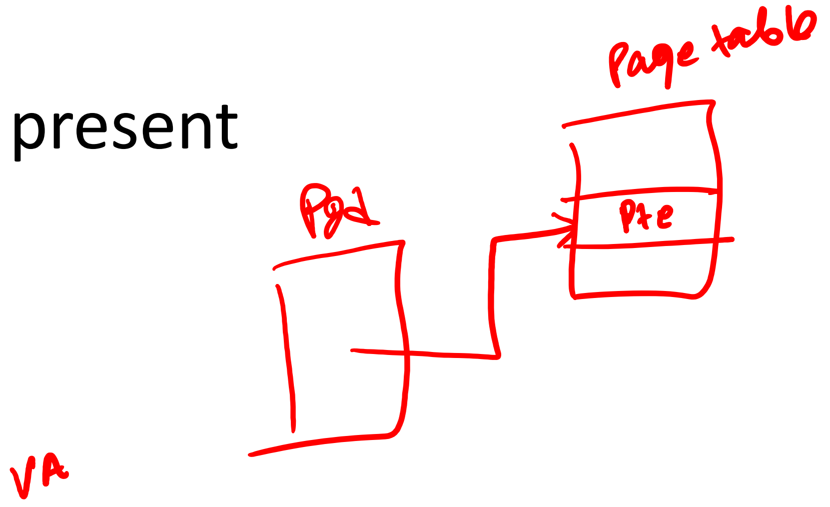


kfree

kalloc

walkpgdir(unsigned *pgdir, unsigned va)

- Expects the valid page directory
- Returns the address of the page table entry for a given va
- Creates a page table page if not already present



```
walkpgdir (unsigned *pgd, unsigned va) {
   int pgd_index = va >> 22;
   unsigned pgd_entry = pgd[pgd_index];
   unsigned *pgtable;
   if (!(pgd_entry & PTE_P)) {  /* create a page table entry */
       pgtable = kalloc();
       memset(pgtable, 0, PAGE_SIZE);
       pgd[pgd_index] = V2P(pgtable) | PTE_P | PTE_W | PTE_U;
   } else {
       pgtable = P2V(PTE_ADDR(pgd_entry));
   }
   unsigned pgtable_idx = (va >> 12) & 0x3ff;
   return &pgtable[pgtable_idx];
}
```

mapuserpage

```
mapuserpage(unsigned *pgdir, unsigned va, unsigned pa) {
    unsigned *pte = walkpgdir(va);
    *pte = pa | PTE_P | PTE_W | PTE_U;
}
```

mapkernelpage

```
mapkernelpage(unsigned *pgdir, unsigned va, unsigned pa) {
    unsigned *pte = walkpgdir(va);
    *pte = pa | PTE_P | PTE_W;
}
```

Create a page table and map all physical pages

```
#define KERNEL_BASE  0x80000000
#define PHYSEND  RAM_SIZE     ( 1 GB)
create_page_table(){
   unsigned *pgdir = kalloc();
   unsigned va;
   memset (pgdir, 0, PAGE_SIZE);
   for (va = KERNEL_BASE; va < P2V(PHYSEND); va+=PAGE_SIZE)
          mapkernelpage(pgdir, va, V2P(va));
}
```

Loading new page table into memory

```
load_page_table(unsigned *pgdir) {
    load_cr3 (V2P(pgdir));
}
```

Page table entries

- Why cr3 and page directory contain physical addresses?

- A page table is used to find the PA corresponding to a linear address. If cr3 and page directory themselves contain linear address then who is going to convert them to PA.

Why can user applications not modify the page table entries?

- Page table pages are only mapped in the kernel address space

Fork

COPY ON WRITE

Page fault