More Virtual Memory and Physical memory management

ECE 469, Jan 25

Aravind Machiry

Recap – Page Table & Addr Translation

0x08048000	
Page number (20-bits) 0x08048	Offset (12-bits) 0x000

Directory Index (10-bits)	Table index (10-bits)
0x20	0x48

31 22 12

Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

Mem access #1

CR3[0x20]

	Page Directory Entry
0	Addr PT1
…	Addr PT2
0x20	Addr PT10
0x3ff	Addr PT500

PT 10	Page Table Entry
0	NO MAPPING
0x48	0x10000
0x49	0x11000
0x4a	0x50000

Recap – Page Table & Addr Translation

0x08048000

Page number (20-bits)	Offset (12-bits)
0x08048	0x000

Directory Index (10-bits)	Table index (10-bits)
0x20	**0x48**

31 22 12

Mem access #1

CR3[0x20]

	Page Directory Entry
0	Addr PT1
...	Addr PT2
0x20	Addr PT10
0x3ff	Addr PT500

PT 10	Page Table Entry
0	NO MAPPING
0x48	0x10000
0x49	0x11000
0x4a	0x50000

Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

Recap – Page Table & Addr Translation

0x08048000

Page number (20-bits)	Offset (12-bits)
0x08048	0x000

Directory Index (10-bits)	Table index (10-bits)
0x20	**0x48**

31 22 12

Mem access #1
CR3[0x20]

	Page Directory Entry
0	Addr PT1
...	Addr PT2
0x20	Addr PT10
0x3ff	Addr PT500

Mem access #2 **PDE[0x48]**

PT 10	**Page Table Entry**
0	NO MAPPING
0x48	0x10000
0x49	0x11000
0x4a	0x50000

Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

Recap – Page Table & Addr Translation

0x08048000

Page number (20-bits) 0x08048	Offset (12-bits) 0x000

Directory Index (10-bits) 0x20	Table index (10-bits) 0x48

31 22 12

Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

Mem access #1

CR3[0x20]

	Page Directory Entry
0	Addr PT1
...	Addr PT2
0x20	Addr PT10
0x3ff	Addr PT500

Mem access #2 PDE[0x48]

PT 10	Page Table Entry
0	NO MAPPING
0x48	0x10000
0x49	0x11000
0x4a	0x50000

Mem access #3

Physical access

Recap – Page Table & Addr Translation

0x08048000

Page number (20-bits)	Offset (12-bits)
0x08048	0x000

Directory Index (10-bits)	Table index (10-bits)
0x20	0x48

31 22 12

Mem access #1 CR3[0x20]

	Page Directory Entry
0	Addr PT1
…	Addr PT2
0x20	Addr PT10
0x3ff	Addr PT500

Mem access #2 PDE[0x48]

PT 10	Page Table Entry
0	NO MAPPING
0x48	0x10000
0x49	0x11000
0x4a	0x50000

Mem access #3

Physical access

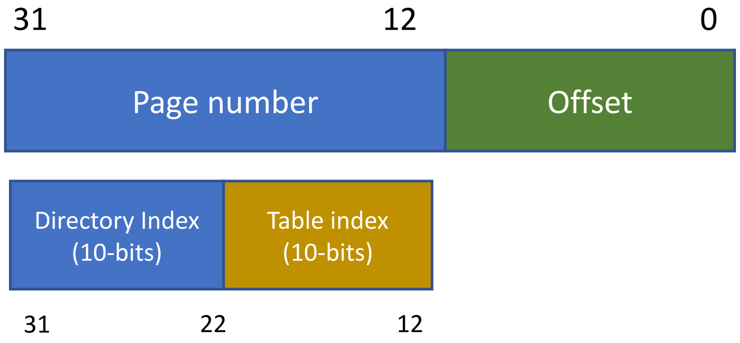
Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

More Virtual Memory

- Multi-level page tables
- Page Table Permissions
- Physical Memory Management

Page Directory / Table

- In x86 (32-bit), CPU uses 2-level page table
- 10-bit directory index
- 10-bit page table index
- 12-bit offset
- **2-level paging**



Size of Page Directory!

- Page Size = 4 KB

One page,
4KB

PDE 0
PDE 1
PDE 2
PDE 3
PDE …
PDE …
PDE 1022
PDE 1023

Each entry is 4-byte (32 bits)

4096 / **4** = 1024 entries

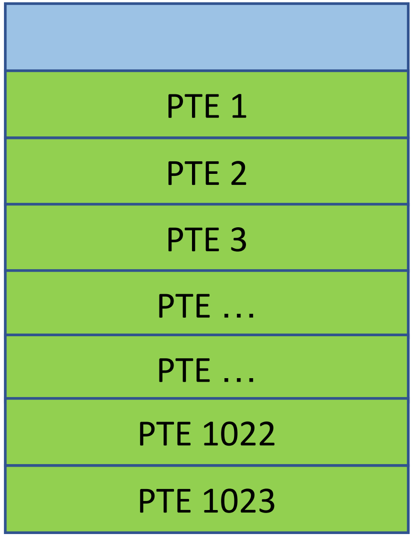
1024 == 2^{10}

10-bit index for PD

Size of Page Table!

- Page Size = 4 KB

One page, 4KB



Each entry is 4-byte (32 bits)

4096 / **4** = 1024 entries

$1024 == 2^{10}$

10-bit index for PT

Increasing Virtual Address Space

- 32-bit address:
  - 2^32 == total 4GB
  - We ignore lower 12 bits = 2^52 total pages
  - Page directory: 4KB.
  - Page table chunk or Second level page table: 4KB.
  - PTE/PDE entry size = 4 bytes

- **All metadata is of size 4KB**!!

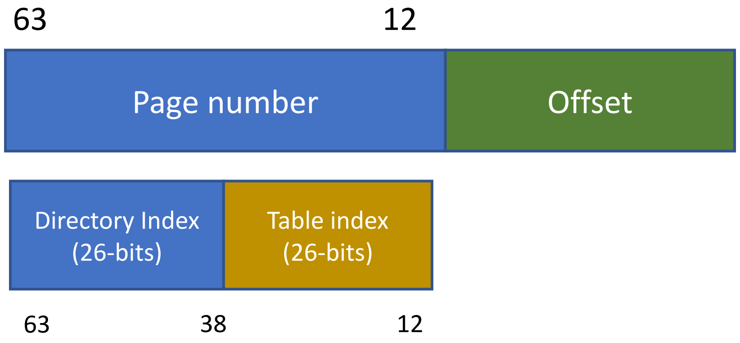
Increasing Virtual Address Space

- 64-bit addresses:
  - 2^64 == 16 EB == 16,384 PB == 16,777,216 TB
  - We ignore lower 12 bits = 2^52 total pages

- **How should we handle paging?**

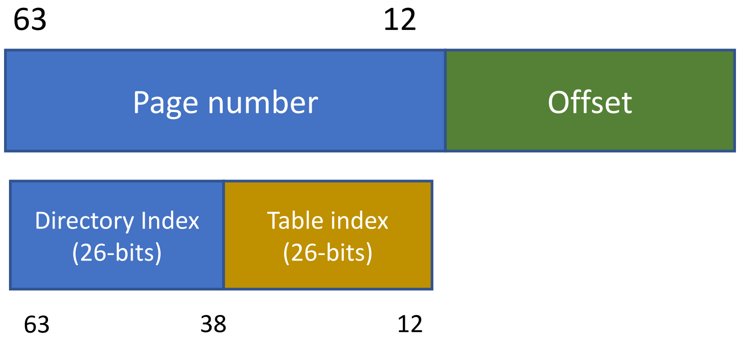
Choices of paging in x64

- 2-level paging
  - 26-bit directory index
  - 26-bit page table index
  - 12-bit offset
  - Each PTE/PDE size = 8 bytes



Choices of paging in x64

- 2-level paging
  - 26-bit directory index
  - 26-bit page table index
  - 12-bit offset
  - Each PTE/PDE size = 8 bytes

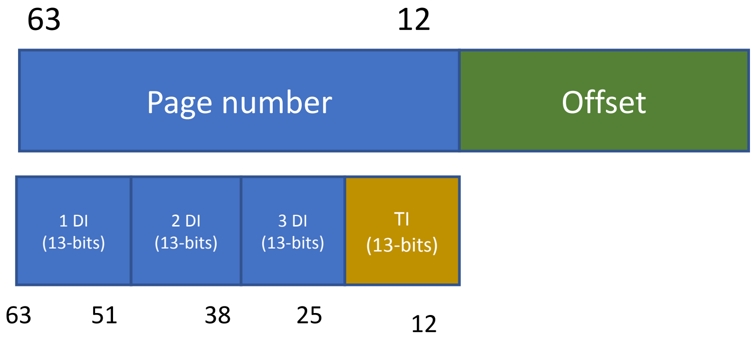


- **Page directory size or Page table size:**
  - **(2^26)*8 = 512 MB**

Choices of paging in x64

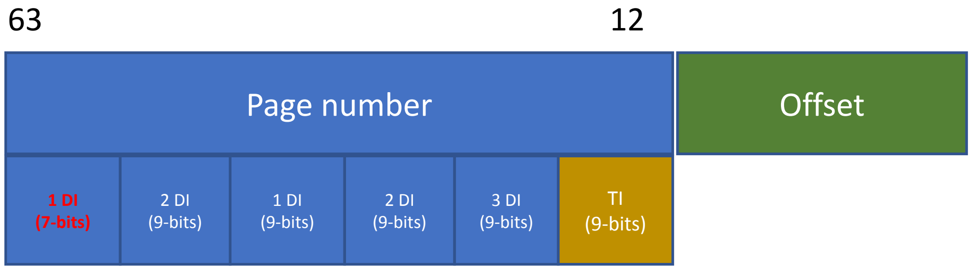
- 4-level paging
  - 13-bit directory indexes - 3 level
  - 13-bit page table index
  - 12-bit offset
  - Each PTE/PDE size = 8 bytes

- **Page directories size or Page table size:**
  - **(2^13)*8 = 64 KB**



Choices of paging in x64

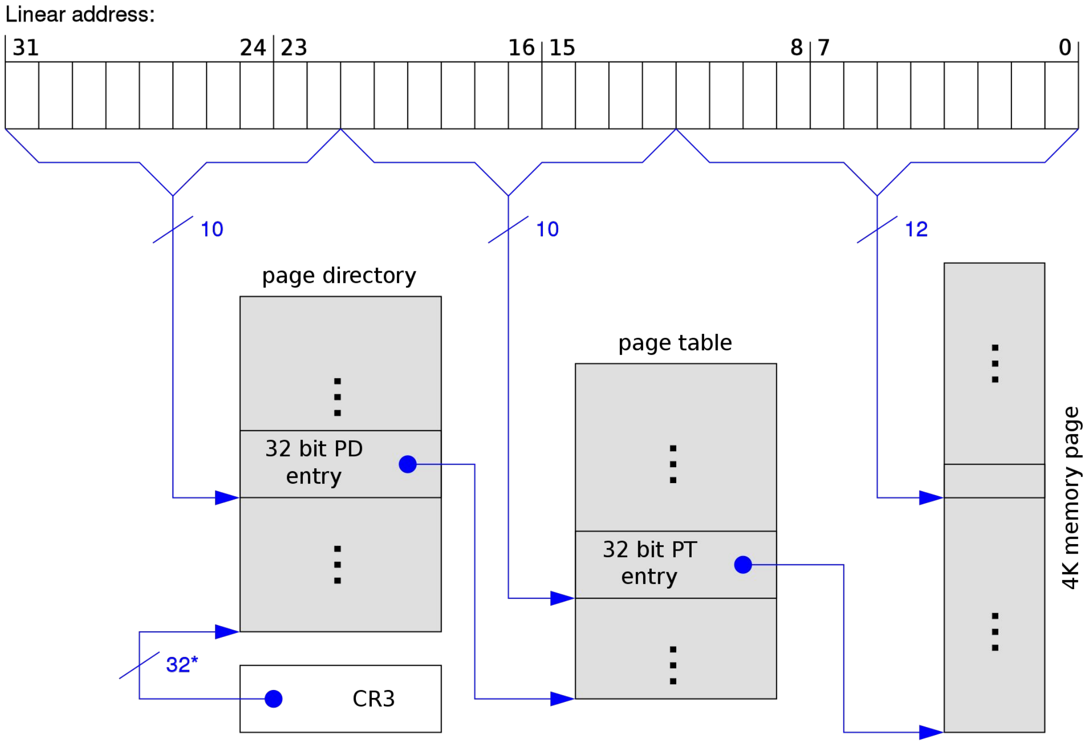
- All metadata should be of 4KB (contiguous memory) chunks for efficient memory usage
- What should be the ideal level of paging for 64-bit address?
  - Each entry (PTE/PDE) size = 8 bytes
  - Number of entries in 4KB = (4KB/8) = 512 = (2^9) = 9-bit index
  - **64-bit address:**
    - **12-bit offset**
    - **52-bit page number:**
      - **52/9 = 5.777 = 6**
      - **6-level paging**



x86_64: 48-bit address space

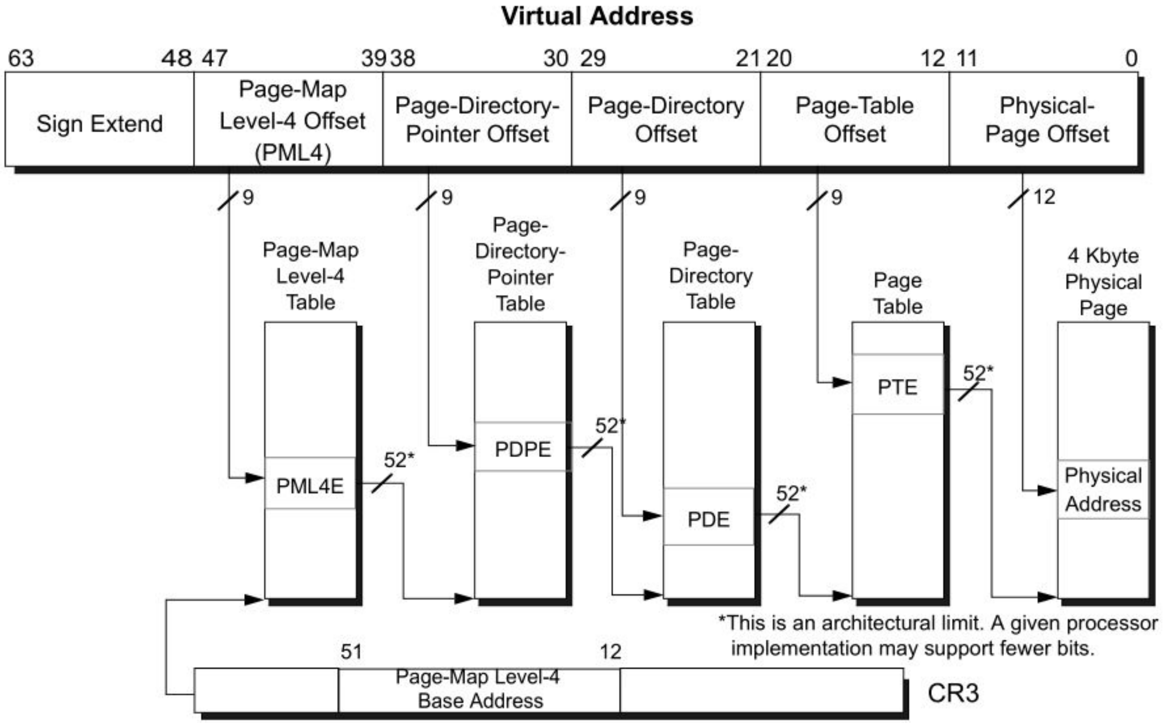
- Initial amd64 processors use only 48-bit virtual address space
  - 2^48 = 256 TB
- Each level of table tree can process 9 bits (512 entries in table)
- We ignore lower 12 bits
  - 48 – 12 = 36 (total 2^36 pages)
  - 36 / 9 = 4   (each table can process 9 bits of address space)
- **We use 4-level page table**

Two-level paging (32-bit)



*) 32 bits aligned to a 4-KByte boundary

Four-level paging (64-bit)



More memory with increasing address space…

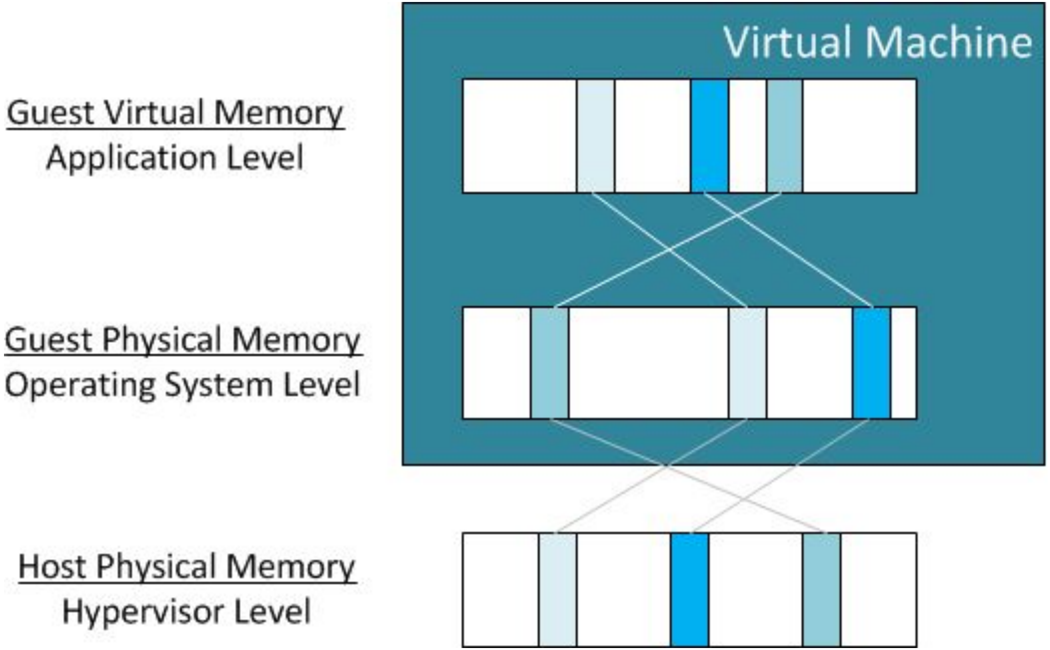
- 256 TB might not enough for big data processing
  - E.g., analyzing online social network of users in Facebook
    - More than 1 billion users, more than 1 trillion edges among users
    - 1 byte per edge = 1TB

- 4 levels, 48 bit

- 5 levels? Yes we can, 48 + 9 = 57 bits = 128PB

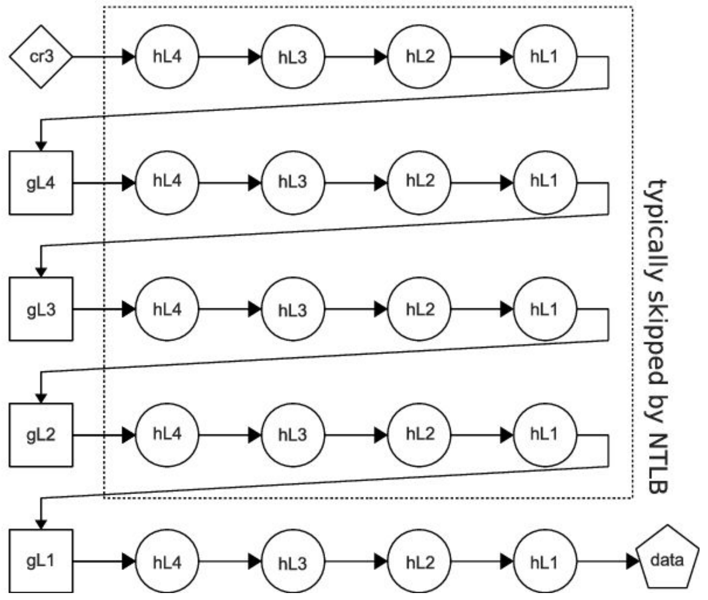
- 6 levels? Maybe, but 57 + 9 = 68 bits, out of 64 bits…

Virtual Machines - Detour



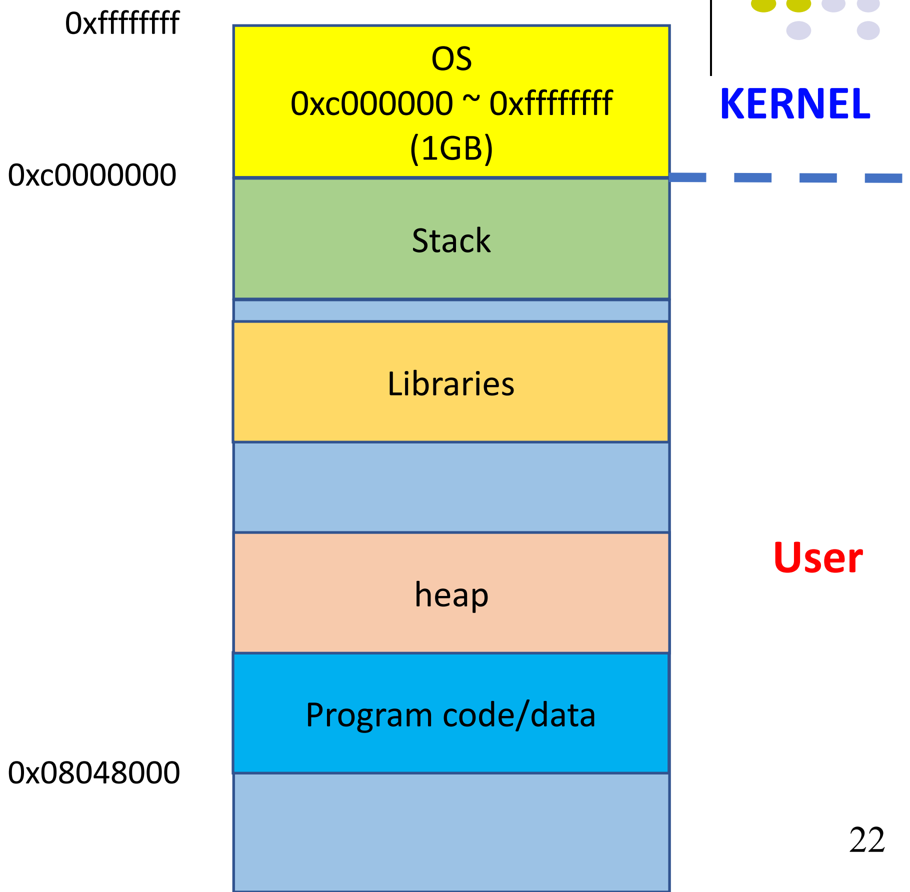
Virtual Machines - Nested Page Tables (NPE/EPE)

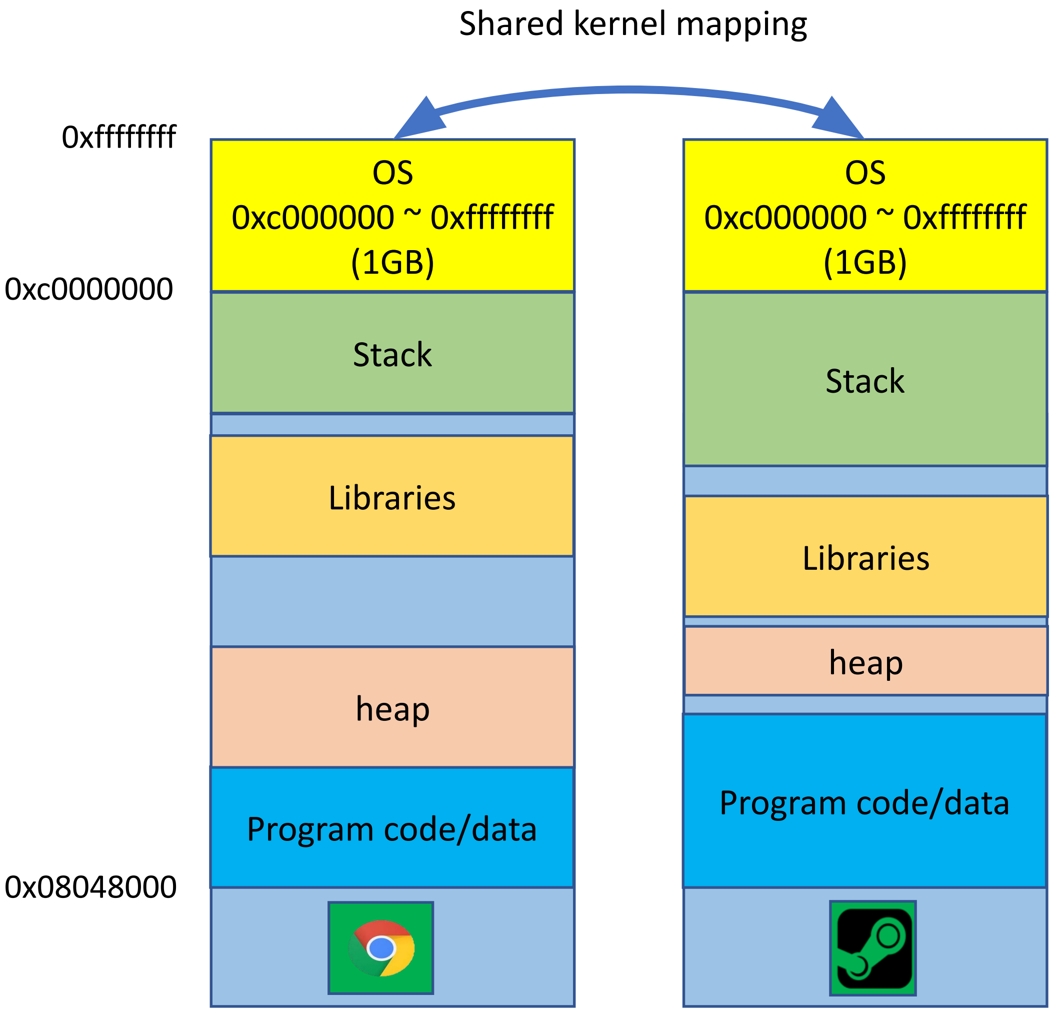
- A case of 48-bit 4-level page table
- Suppose you run Windows
  - Install VMWARE
  - Install a Linux VM
- Hosta: Windows
- Guest: Linux
- Physical address of Linux
  - This is just a virtual address of Windows



Process Virtual Memory Layout

- OS allocates a separate virtual memory space to each process
- Transparency
  - Do not have to worry about a system's memory usage status
- Isolation
  - Others can't access my virtual memory space





Why kernel is mapped in all user processes?

Easier context-switch

Memory Maps on x64 machine

```
machiry@machiry-home:~$ cat /proc/self/maps | tail
7fe7bcb0a000-7fe7bcb0b000 r--p 00000000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb0b000-7fe7bcb2e000 r-xp 00001000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb2e000-7fe7bcb36000 r--p 00024000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb37000-7fe7bcb38000 r--p 0002c000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb38000-7fe7bcb39000 rw-p 0002d000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb39000-7fe7bcb3a000 rw-p 00000000 00:00 0
7ffda7458000-7ffda7479000 rw-p 00000000 00:00 0                          [stack]
7ffda7584000-7ffda7588000 r--p 00000000 00:00 0                          [vvar]
7ffda7588000-7ffda758a000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

Memory Maps on x64 machine

```
machiry@machiry-home:~$ cat /proc/self/maps | tail
7fe7bcb0a000-7fe7bcb0b000 r--p 00000000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb0b000-7fe7bcb2e000 r-xp 00001000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb2e000-7fe7bcb36000 r--p 00024000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb37000-7fe7bcb38000 r--p 0002c000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb38000-7fe7bcb39000 rw-p 0002d000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb39000-7fe7bcb3a000 rw-p 00000000 00:00 0
7ffda7458000-7ffda7479000 rw-p 00000000 00:00 0                          [stack]
7ffda7584000-7ffda7588000 r--p 00000000 00:00 0                          [vvar]
7ffda7588000-7ffda758a000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

```
machiry@machiry-home:~$ cat /proc/5668/maps | tail
7f300561e000-7f3005641000 r-xp 00001000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f3005641000-7f3005649000 r--p 00024000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f3005649000-7f300564a000 r--s 00000000 00:36 204                        /run/user/1000/dconf/user
7f300564a000-7f300564b000 r--p 0002c000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f300564b000-7f300564c000 rw-p 0002d000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f300564c000-7f300564d000 rw-p 00000000 00:00 0
7ffc023b3000-7ffc023d4000 rw-p 00000000 00:00 0                          [stack]
7ffc023ea000-7ffc023ee000 r--p 00000000 00:00 0                          [vvar]
7ffc023ee000-7ffc023f0000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

How does OS ensure a user process does not access kernel memory?

- OS needs to ensure that a user process cannot access (read/write) kernel (or OS memory)?
  - Why?
    - Hint: Security!
      - Remember: sudo!?

How does OS ensure a user process does not access kernel memory?

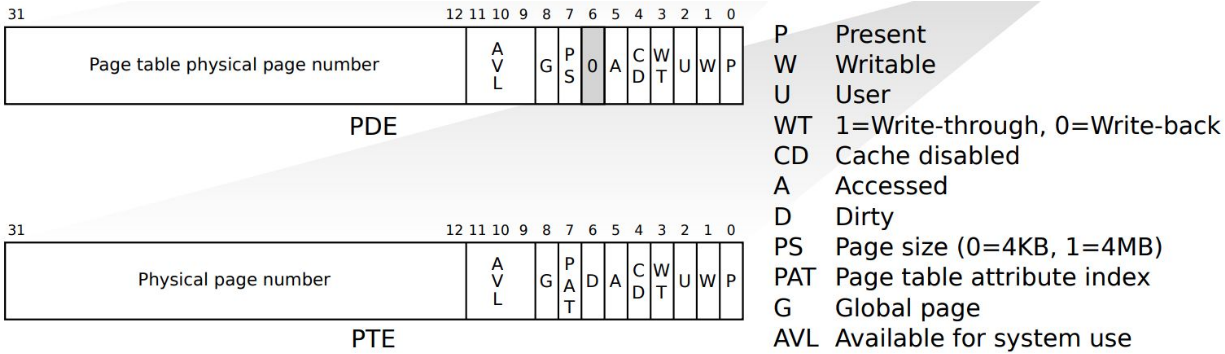
- OS needs to ensure that a user process cannot access (read/write) kernel (or OS memory)?
  - Why?
    - Hint: Security!
      - Remember: sudo!?

- Permissions bits in Page directories and Page Tables!!

Page Directory / Table Entry (PDE/PTE)

- Top 20 bits: physical page number
  - Physical page number of a page table (PDE)
  - Physical page number of the requested virtual address (PTE)

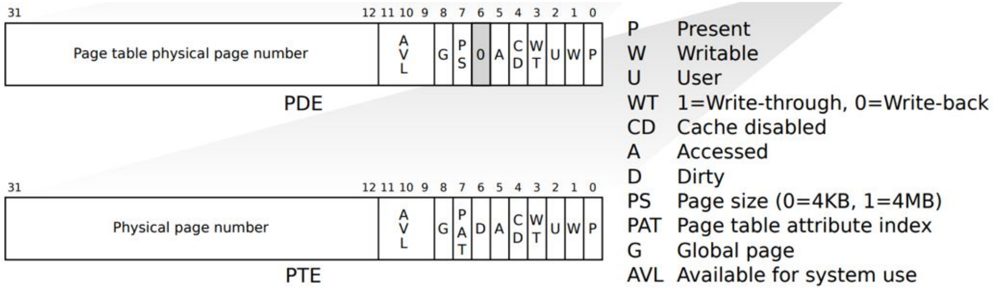
- Lower 12 bits: some flags
  - Permission
  - Etc.



Permission Flags

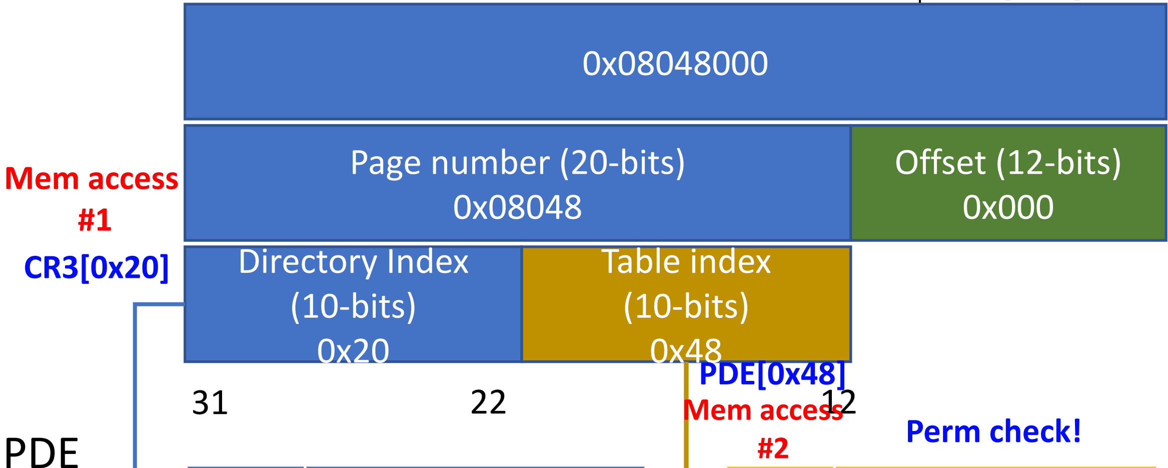
- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

	Page Table Entry	
0	Addr PT	
0x48	0x10000 << 12 | PTE_U | PTE_W	Invalid
0x49	0x11000 << 12 | PTE_P | PTE_W	Kernel, writable
0x4a	0x50000 << 12 | PTE_P | PTE_U	User, read-only



When CPU Checks Permission Bits?

- In address translation
- 1. Virtual address
- 2. PDE = CR3[PDX]
  - Checks permission bits in PDE
- 3. PTE = PDE[PTX]
  - Checks permission bits in PTE



	Page Directory Entry
0	Addr PT | P | U | W
0x20	Addr PT | P | U | W
0x30	Addr PT | P | U | W
0x3ff	Addr PT | | U | W

	Page Table Entry
0	0x0 | P | U | W
0x48	0x10000 | P | U | W
0x49	0x11000 | P | W
0x4a	0x50000 | U | W

When CPU Checks Permission Bits?

- A virtual memory address is inaccessible if PDE disallows the access

- A virtual memory address is inaccessible if PTE disallows the access

- Both **PDE and PTE should allow the access**…

PDE/PTE Permission Examples 0

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W | PTE_U
- PTE: PTE_P | PTE_W | PTE_U
- **Can user (ring 3) access it? Is it writable?**

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 0

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W | PTE_U
- PTE: PTE_P | PTE_W | PTE_U
- **Can user (ring 3) access it? Is it writable?**
  - Valid, accessible by ring 3, and writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 1

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W | PTE_U
- PTE: PTE_P | PTE_U
- **Can user (ring 3) access it? Is it writable?**

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 1

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W | PTE_U
- PTE: PTE_P | PTE_U
- **Can user (ring 3) access it? Is it writable?**
  - Valid, accessible by ring 3, but not writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 2

- Virtual address 0x01020304

- PDE: PTE_P | PTE_U

- PTE: PTE_P | PTE_W | PTE_U

- **Can user (ring 3) access it? Is it writable?**

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry

- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable

- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 2

- Virtual address 0x01020304
- PDE: PTE_P | PTE_U
- PTE: PTE_P | PTE_W | PTE_U
- **Can user (ring 3) access it? Is it writable?**
  - Valid, accessible by ring 3, but not writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 3

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W | PTE_U
- PTE: PTE_P
- **Can user (ring 3) access it? Is it writable?**

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 3

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W | PTE_U
- PTE: PTE_P
- **Can user (ring 3) access it? Is it writable?**
  - valid, inaccessible by ring3, not writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 4

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W
- PTE: PTE_P | PTE_U
- **Can user (ring 3) access it? Is it writable?**

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 4

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W
- PTE: PTE_P | PTE_U
- **Can user (ring 3) access it? Is it writable?**
  - valid, inaccessible by ring3, not writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 5

- Virtual address 0x01020304
- PDE: PTE_P | PTE_U
- PTE: PTE_U
- **Can user (ring 3) access it? Is it writable?**

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 5

- Virtual address 0x01020304
- PDE: PTE_P | PTE_U
- PTE: PTE_U
- **Can user (ring 3) access it? Is it writable?**
  - invalid

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 6

- Virtual address 0x01020304
- PDE: PTE_U
- PTE: PTE_P | PTE_U
- **Can user (ring 3) access it? Is it writable?**

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 6

- Virtual address 0x01020304
- PDE: PTE_U
- PTE: PTE_P | PTE_U
- **Can user (ring 3) access it? Is it writable?**
  - invalid

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

Valid permission bits..

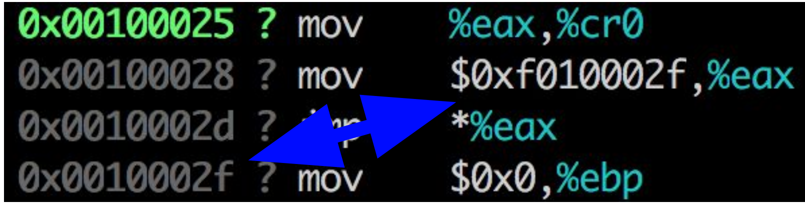
- Kernel: R, User: --
  - PTE_P
- Kernel: R, User: R
  - PTE_P | PTE_U
- Kernel: RW, User: RW
  - PTE_P | PTE_U | PTE_W

Cannot have permissions such as …

- Kernel: RW, User: R
  - PTE_P | PTE_W | PTE_U -> User RW…
  - PTE_P | PTE _W -> User --
- Kernel: R, User: RW
  - PTE_P | PTE_U | PTE_W -> Kernel RW…
  - PTE_P | PTE_U   -> User R…
- Kernel: --, User: RW
  - PTE_P | PTE_U | PTE_W -> Kernel RW…

Flexibility of virtual memory!

- Virtual to physical address mapping is in N-to-1 relation
  - N number of virtual addresses could be mapped to 1 physical address
- E.g., for a physical address 0x100000
  - JOS maps VA 0x100000 to PA 0x100000
  - JOS maps VA 0xf0100000 to PA 0x100000
- Why?
  - EIP before enabling paging: 0x100025
  - EIP after enabling paging: 0x100028
  - Then jumps to 0xf010002f



Sharing a Physical Page!

- Example: Loading of the same program

Process 0, runs /bin/bash, loads at virt addr 0x35555000

	Page Table Entry
0	…
0x155	0x10303 | FLAG

Process 1, runs /bin/bash, loads at virt addr 0x43132000

	Page Table Entry
0	…
0x132	0x10303 | FLAG

Physical memory

/bin/bash at
Phys addr 0x10303000

2 or more mappings to 0x10303000 is possible!

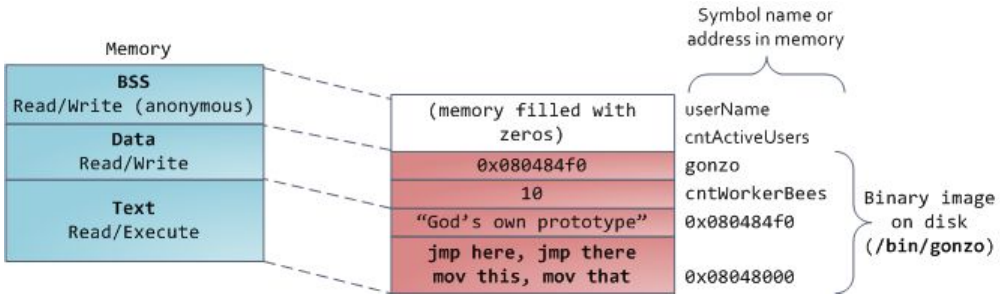
Allocating Virtual Memory

- Static allocation is inefficient:
  - Why don't we just allocate entire virtual address space to a process?
  - Inefficient: The process may not access entire virtual address space

- Solution: Dynamic, Request based

Dynamic space allocation

- OS allocates space (valid PTE entries in page table) as dictated by the program binary and start running the process.



Dynamic space allocation

- When a process tries to access memory that is not allocated i.e., there is no corresponding valid PTE, then, OS kills the process.
    - E.g., Segmentation Fault!

- A process needs to **explicitly request OS** to allocate additional space (and create valid PTEs).
    - brk **system call** (We will cover this later)

Dynamic space allocation

- We use malloc and free, which actually use brk system call internally

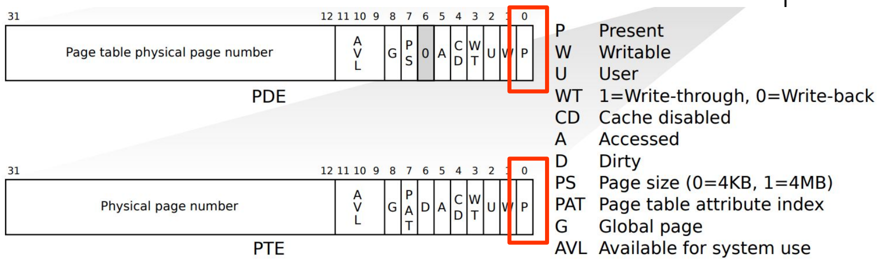
- brk only **allocates virtual memory**! **not physical memory**!

Dynamic space allocation: Example

```
#include<stdio.h>
#include<stdlib.h>
int main() {
        char p;
        int *i = (int *)malloc(16*1024*1024*1024);
        printf("We just requested 16GB of virtual memory!!\n");
        printf("Check memory now, using htop!\n");
        printf("You will be surprised to see your memory usage.\nPress any key to exit.\n");
        scanf("%c", &p);
        return 0;

}
```

What happens when we call malloc?



- Before malloc()?
  - No PTEs
- After malloc()?
  - PDE/PTE updated but present bit not-set
- Upon first access?
  - Assign physical page (and page table) and set the valid bit.

Handling Page faults in Kernel