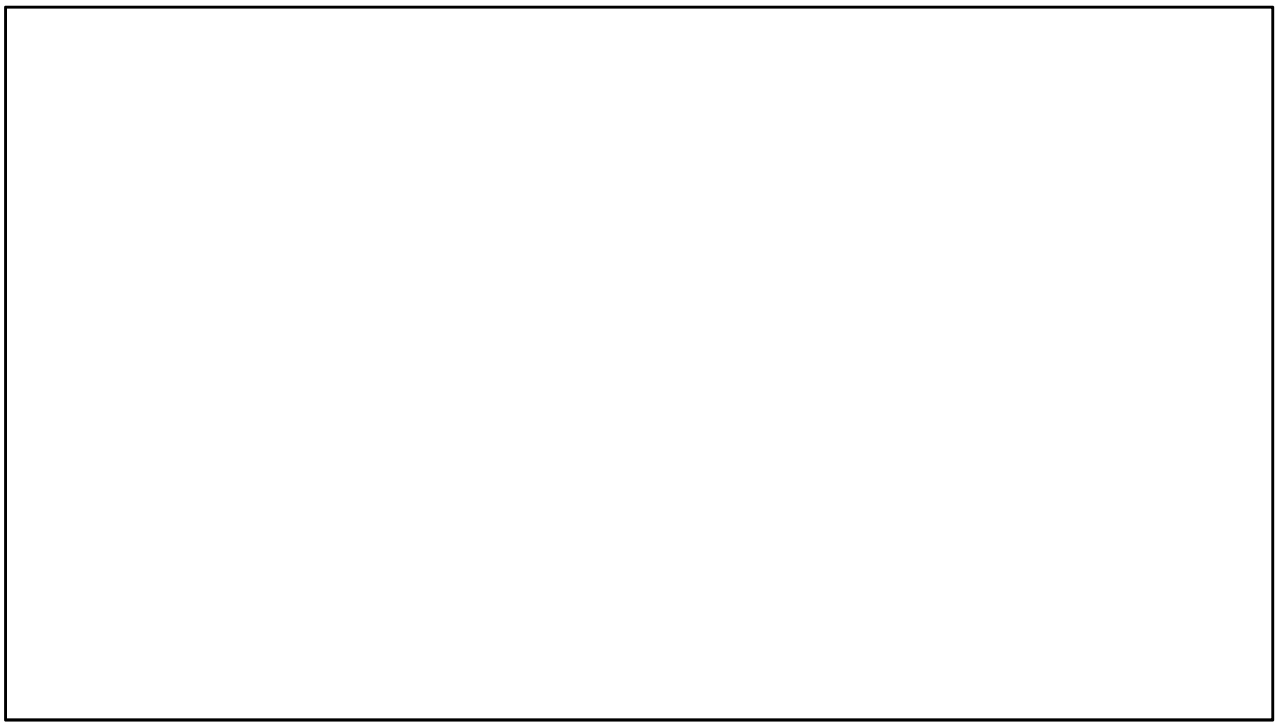


Assignment-4

- In this assignment, you need to implement a custom system call handler in Linux
- Right now, the system call handler in Linux is present at index 128 in the IDT
- One way of extending Linux kernel is adding a kernel module

Kernel module

- A kernel module executes in kernel address space
- You don’t need to compile the whole kernel for using the kernel module
- Kernel module can be compiled separately and loaded in the kernel address space manually

Kernel module

- After the kernel module is loaded, a user application can talk to the kernel module using ioctl system call
- The ioctl system call takes an identifier and an unsigned long value (that can also be a pointer) for user
- Depending on the user's input, ioctl can perform various actions

Kernel module

- In this assignment, you are already provided a working skeleton of a kernel module

- You need to implement the following routines
  - syscall_handler is the custom system call handler (partially implemented)
  - resgister_syscall puts the custom system call handler at index 15 (currently unused) in the IDT
  - UNREGISTER_SYSCALL restores the original IDT
  - syscall_u invokes the custom system call handler from the user-space (partially implemented)

klib

- The kernel module contains a library that you can use in your implementation
- The library provides interfaces for manipulating IDT, custom system call handler in C, etc.

Debugging

- In kernel module printf won't work
  - You can use printk that is similar to printf
  - The output of printk is written to a log file that can be displayed on the terminal using dmesg command

Assigment-4

- You can get the 32-bit address of the custom system call handler by simply typecasting the system call handler to an unsigned long value

unsigned long syscall_addr = (unsigned long)syscall_handler;

The lower16 and higher16 values in struct idt_entry need to be set to the lower 16 and higher 16 bits of syscall_addr.

Assignment-4

- imp_copy_idt can be used to make a copy of current IDT
  - You can change the index 15 in the copied IDT
- imp_load_idt loads an IDT in the IDTR
  - You can load the copied and then modified IDT in the IDTR using imp_load_idt

Optional assignment-4

- Car racing game
  - four components: car1, car2, cheatmode, report
  - all these components are different processes

car racing game

- Each car moves a random distance between 1-10 every second and sends its status to report
- cheatmode can relocate car1 and car2
  - gets a new location of car1/car2 from the user
  - instructs car1/car2 to move to the new position
- report prints the status of both the cars after every second
- Whoever completes a distance of 100 wins the race
  - report prints the winner

IPC

- All processes (car1, car2, report, cheatmode) communicate using pipes
- After the winner is decided all processes are terminated

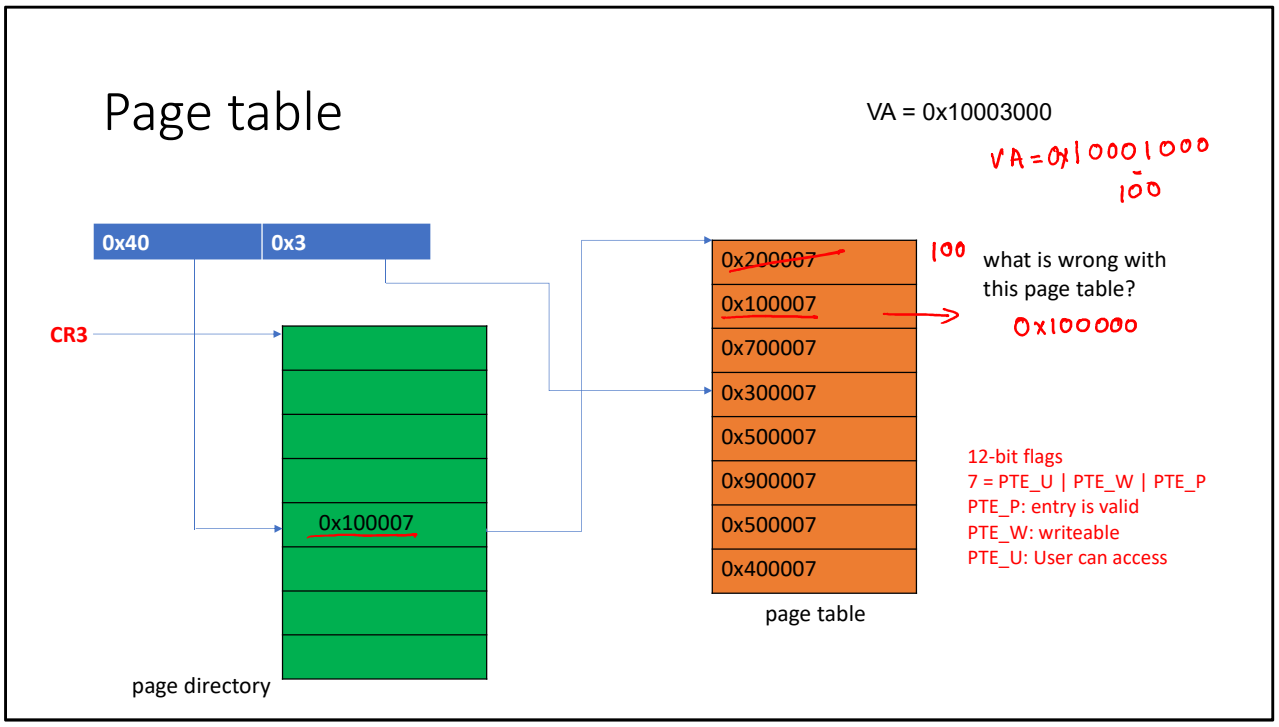
Page table

- two-dimensional page tables
  - The top 10-bits of the VA are used to index in a page directory
  - page-directory contains the physical addresses of a page table
  - The next 10-bits (after top 10 bits) are used to index in the page table
  - The corresponding entry in the page table contains the physical address

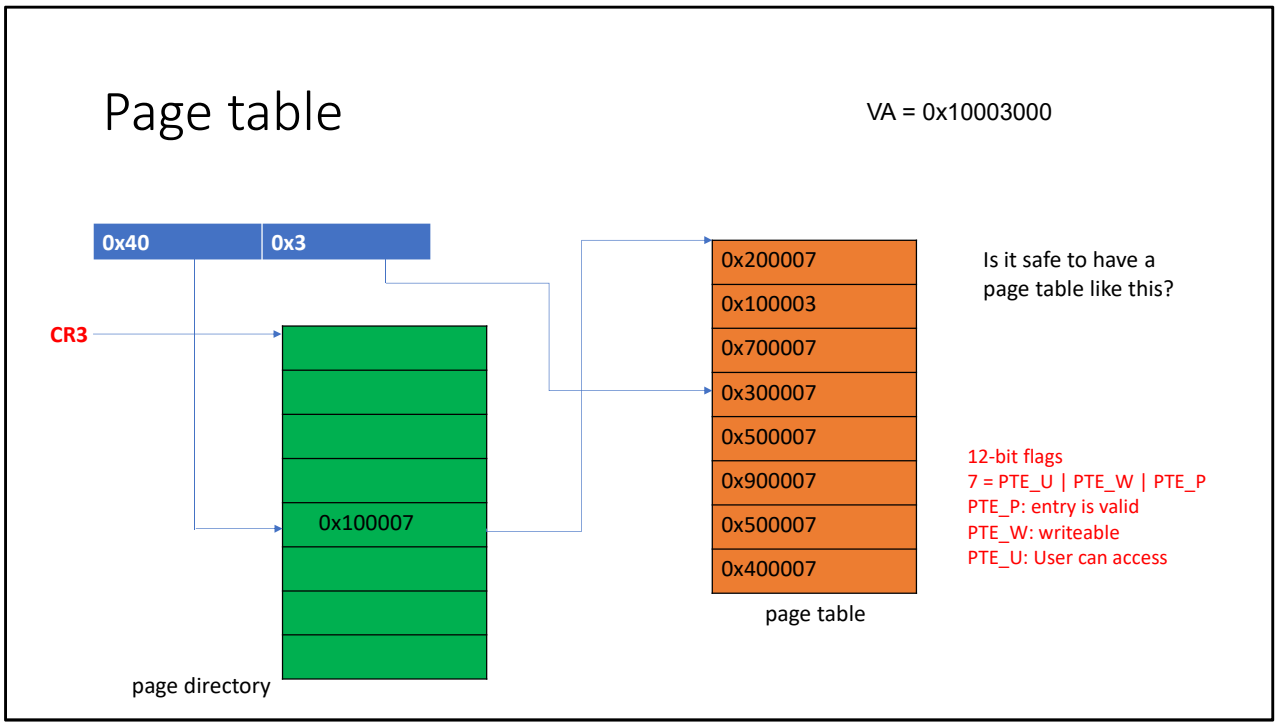
- Read Section-4.3 from Intel manual-3

Page table

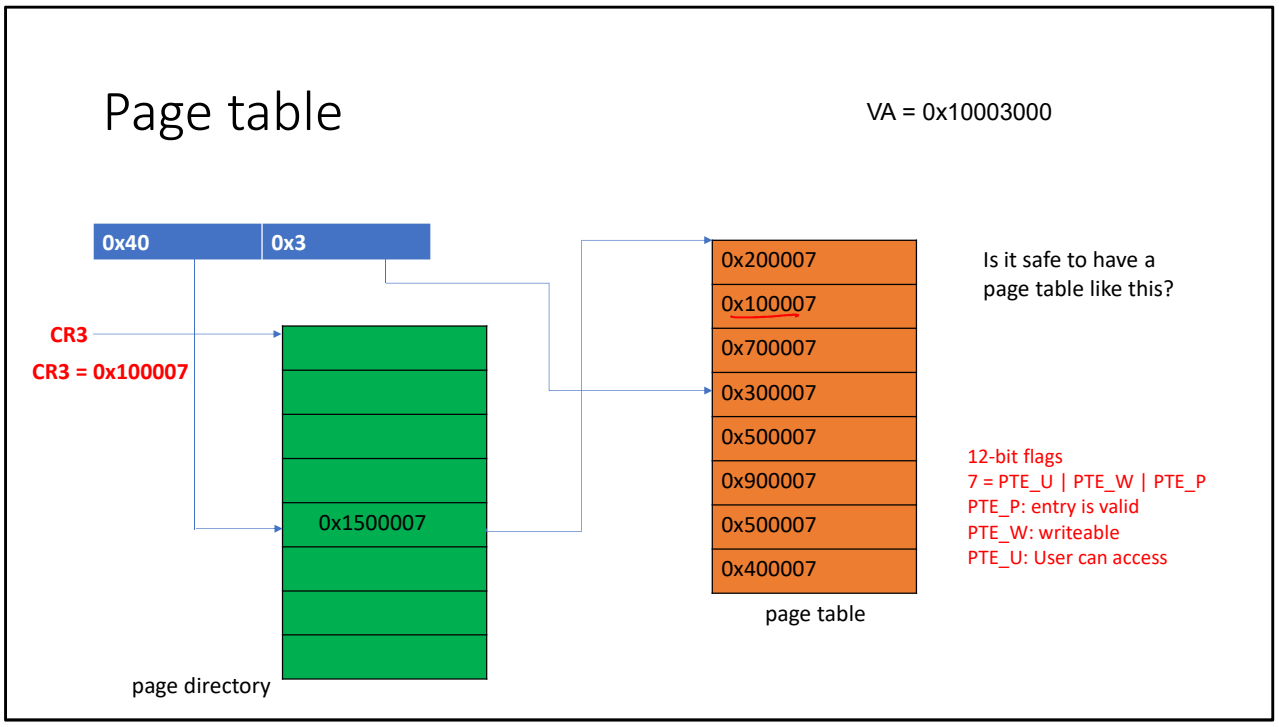
- Why can a user not modify the page table pages?
  - page table pages are not mapped corresponding to the user virtual pages



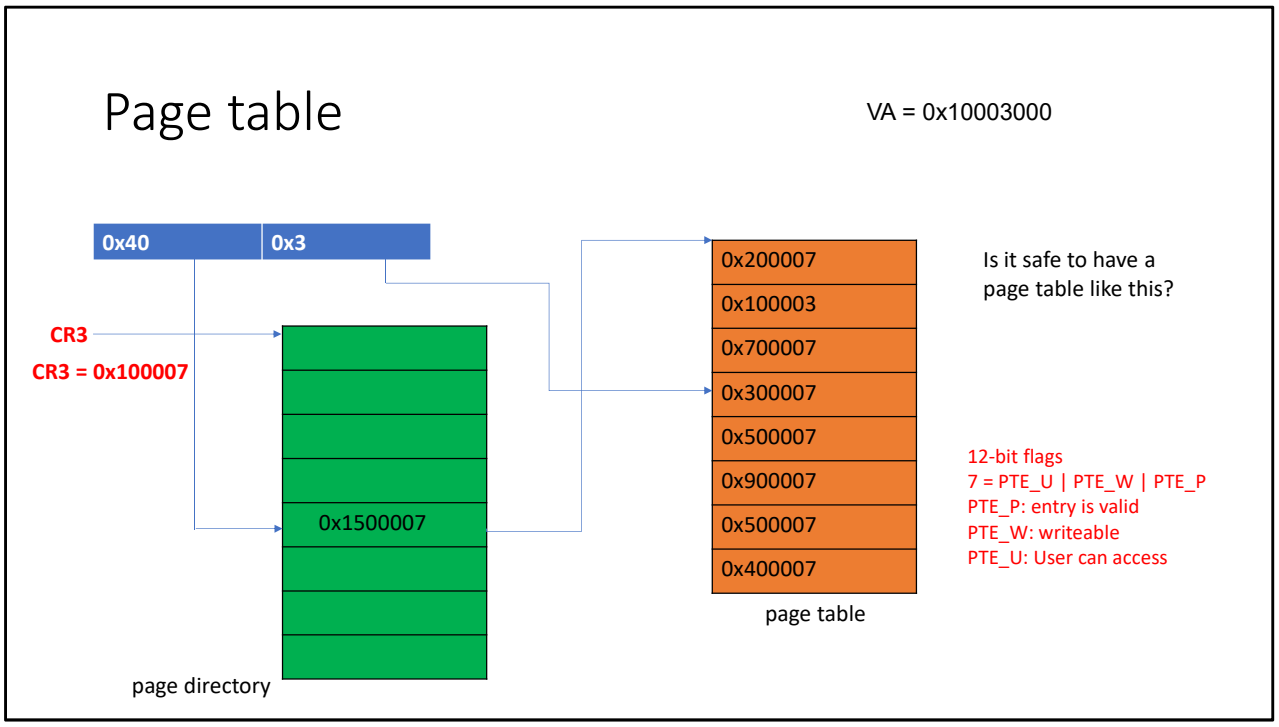
In this example, the physical address (0x1000000) is the address of a page table. Because this physical address is also mapped at index 1, in the page table (which is a user-accessible entry) at index 0x40 in the page directory, the user can overwrite the page table by writing to virtual page 0x10001000. If the users can modify the page table, then they can access any physical address by modifying the page table. The kernel doesn't let this happen by not mapping page table pages in user virtual address space.



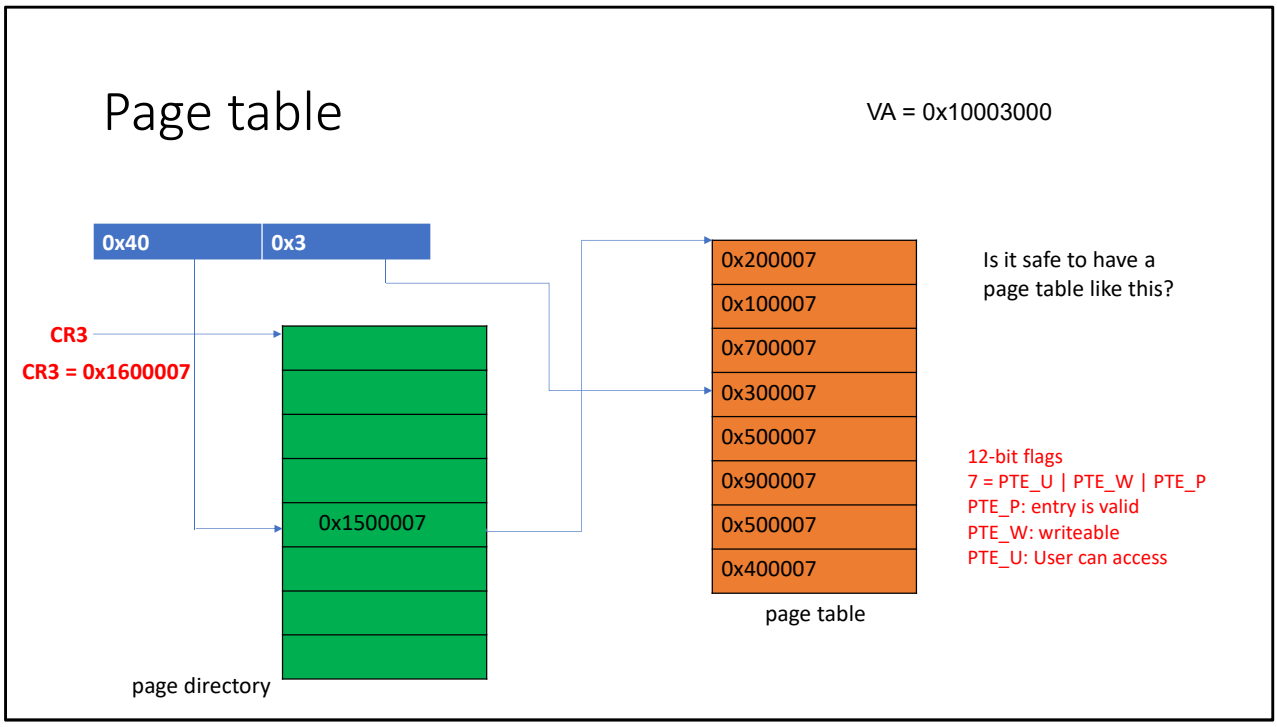
Yes, because even though a page table page is mapped in the page table itself, the user flag is not set. So, the user can't access it.



Here, the page directory is mapped in the page table. This is unsafe because the user can overwrite the page directory.

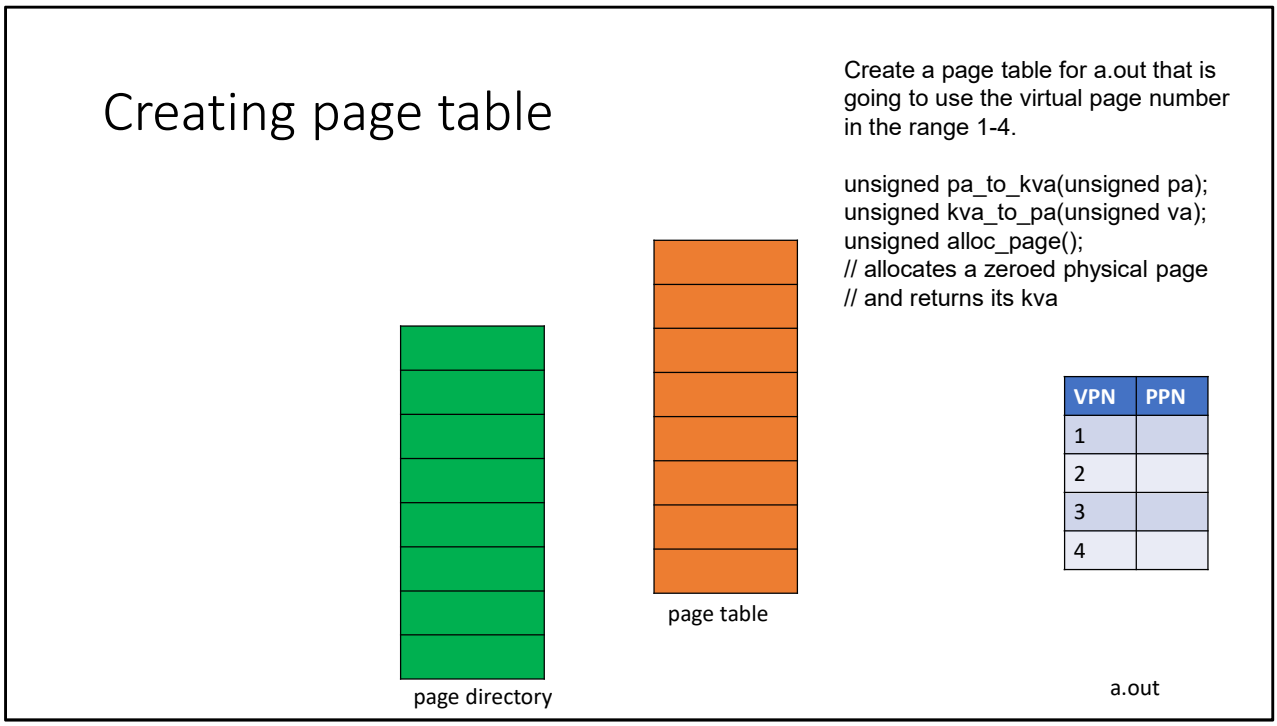


Although the page directory is mapped in the page table, the user flag is not set. Therefore it is safe.



Neither page table nor page directory is mapped in the page table. So, this is safe.

Creating page table



Let us look at how the OS allocates the page table for an application. The application is using virtual pages in the range 1-4.

```
unsigned *page_directory = alloc_page();              // alloc_page returns 0x80001000

// allocating page table entry for VPN 1
// page directory index 0
page_directory[0] is invalid
unsigned *page_table = alloc_page();                  // alloc_page returns 0x80002000
page_directory[0] = kva_to_pa((unsigned)page_table) | (PTE_U | PTE_W | PTE_P);
// page table index 1
page_table[1] is invalid
unsigned *pa = alloc_page();                          // alloc_page returns 0x80003000
page_table[1] = kva_to_pa((unsigned)pa) | (PTE_U | PTE_W | PTE_P);

// allocating page table entry for VPN 2
// page directory index 0
page_directory[0] is valid
unsigned *page_table = pa_to_kva(page_address(page_directory[0]));
// page table index 2
page_table[2] is invalid
unsigned *pa = alloc_page();                          // alloc_page returns 0x80004000
page_table[2] = kva_to_pa((unsigned)pa) | (PTE_U | PTE_W | PTE_P);
```

VPN	PPN
1	3
2	4
3	
4	

First, OS allocates a page directory for the application. For VPN 1, the page directory index is 0, and the page table index is 1. Because nothing is mapped at index 0 in the page directory, the OS allocates a page table whose physical page number is going to be stored in the page directory at index 0. alloc_page returns kernel virtual address. The lower 12-bits of the page table's physical address is zero. The page directory entry is computed by doing a “logical or” of page table‘s physical address and the flags (PTE_U|PTE_W|PTE_P). After the page directory entry is set, the next step to allocate the actual physical page for the virtual page 1. The page table entry (say pteval) is the “logical or” of physical page address and the flags (PTE_U|PTE_W|PTE_P). The page table entry at index 1 (lower 10 bits of virtual page number 1) is set to pteval. For VPN 2, the page directory index is 0 that already contains a valid page table. The page table index is 2, which contains an invalid entry. A physical page is allocated for the virtual page 2. A page table entry (say pteval) is computed by doing the “logical or” of physical page address and page table flags. The page table entry at index 2 (lower 10 bits of virtual page 2) is set to pteval computed in the previous step.

```
// allocating page table entry for VPN 3
// page directory index 0
page_directory[0] is valid
unsigned *page_table = pa_to_kva(page_address(page_directory[0]));
// page table index 3
page_table[3] is invalid
unsigned *pa = alloc_page();                    // alloc_page returns 0x80005000
page_table[3] = kva_to_pa((unsigned)pa) | (PTE_U | PTE_W | PTE_P);

// allocating page table entry for VPN 4
// page directory index 0
page_directory[0] is valid
unsigned *page_table = pa_to_kva(page_address(page_directory[0]));
// page table index 4
page_table[4] is invalid
unsigned *pa = alloc_page();                    // alloc_page returns 0x80006000
page_table[4] = kva_to_pa((unsigned)pa) | (PTE_U | PTE_W | PTE_P);
```

VPN	PPN
1	3
2	4
3	5
4	6

The same process is repeated for mapping virtual pages 3 and 4 in the page table.

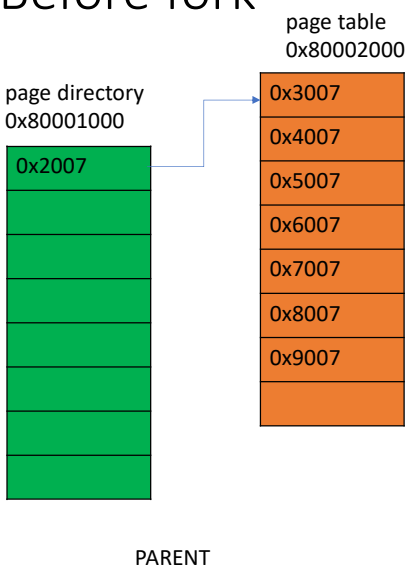
Page table overhead

- For a 16 kb process, how many physical pages needs to be allocated when 2D page table is used
  - six (one for page directory, one for page table, four user pages)
  - if all virtual pages don't belong to the same page table then this number may change
- For a 16 kb process, how many physical pages needs to be allocated when 1D page table is used
  - 1028 (1024 for page table, four user pages)

Fork

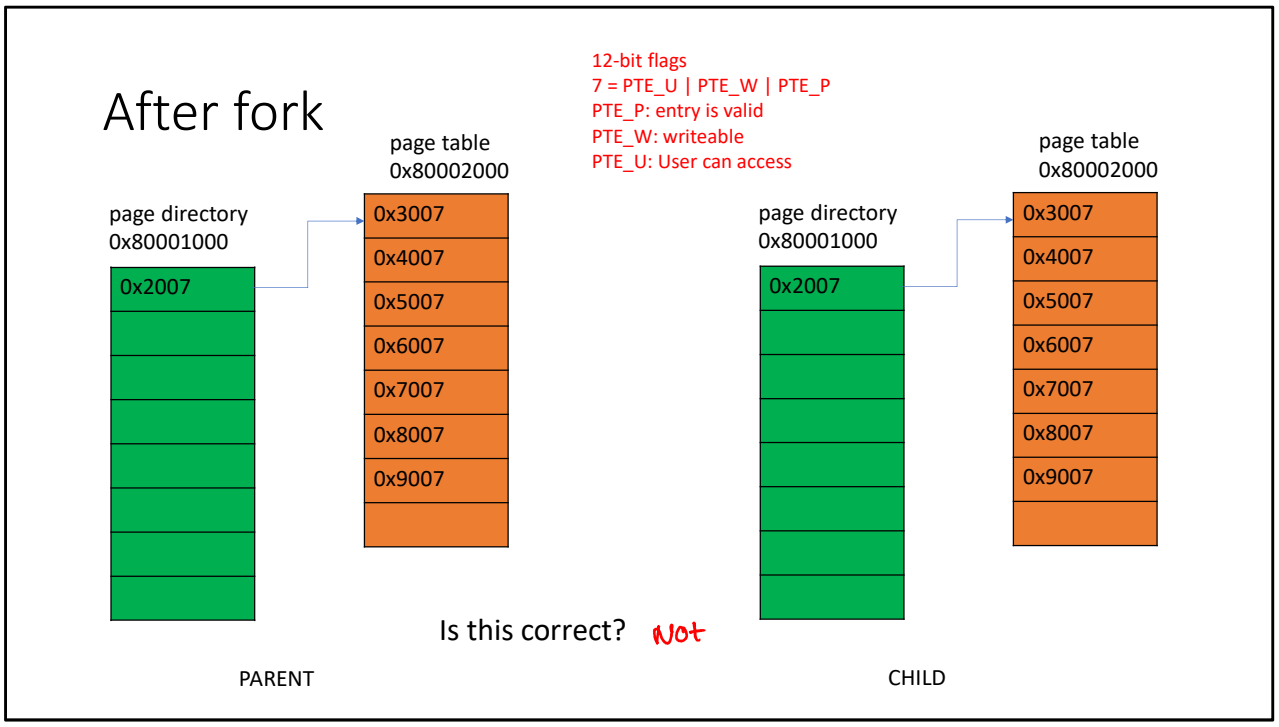
- On fork system call, the OS creates a new page table for the child process
- The entire memory of a parent process is copied to the child process
  - need to allocate the same number of physical pages for the child as parent

Before fork

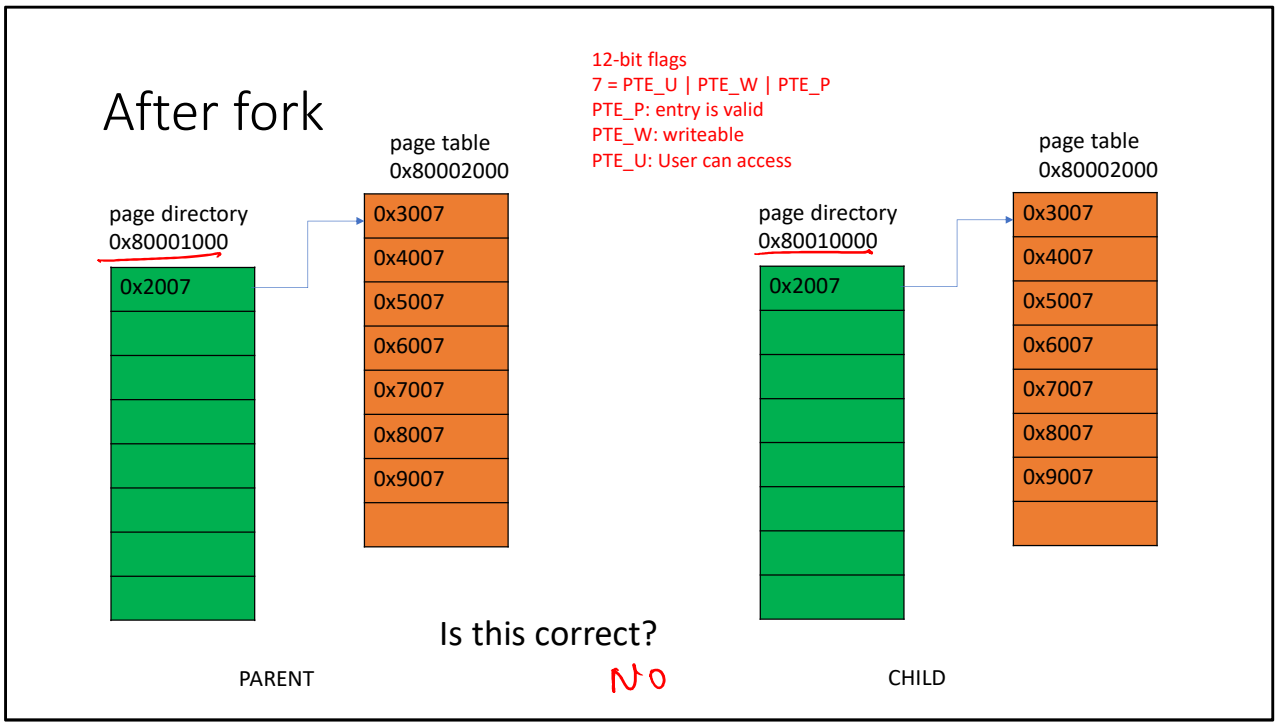


PARENT

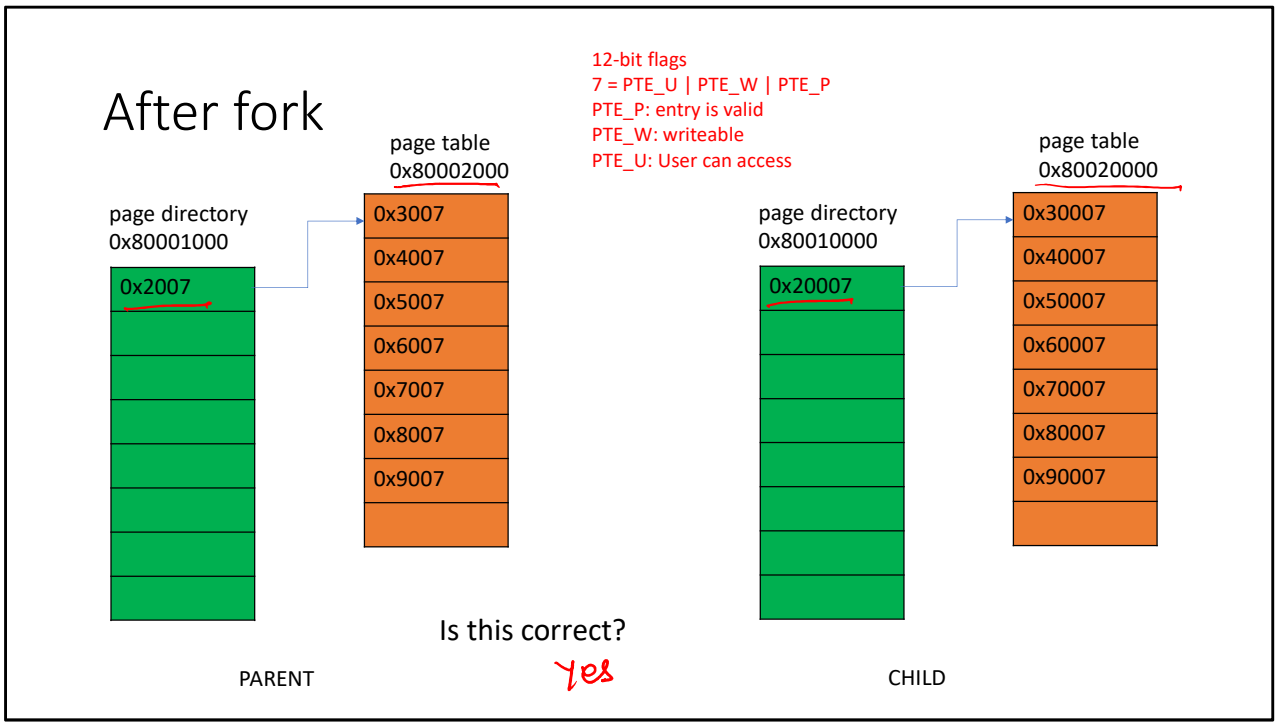
12-bit flags
7 = PTE_U | PTE_W | PTE_P
PTE_P: entry is valid
PTE_W: writeable
PTE_U: User can access



This is not correct because the page directory of parent and child can't be the same.



This is incorrect because even though the page directory is different, both parent and child share the same page table.



This is correct because the page tables and the contents of page tables are different in parent and child.

Fork

- Why a parent and the child can't use the same page table?
  - Because parent and child are different processes
  - They have different address spaces

Threads

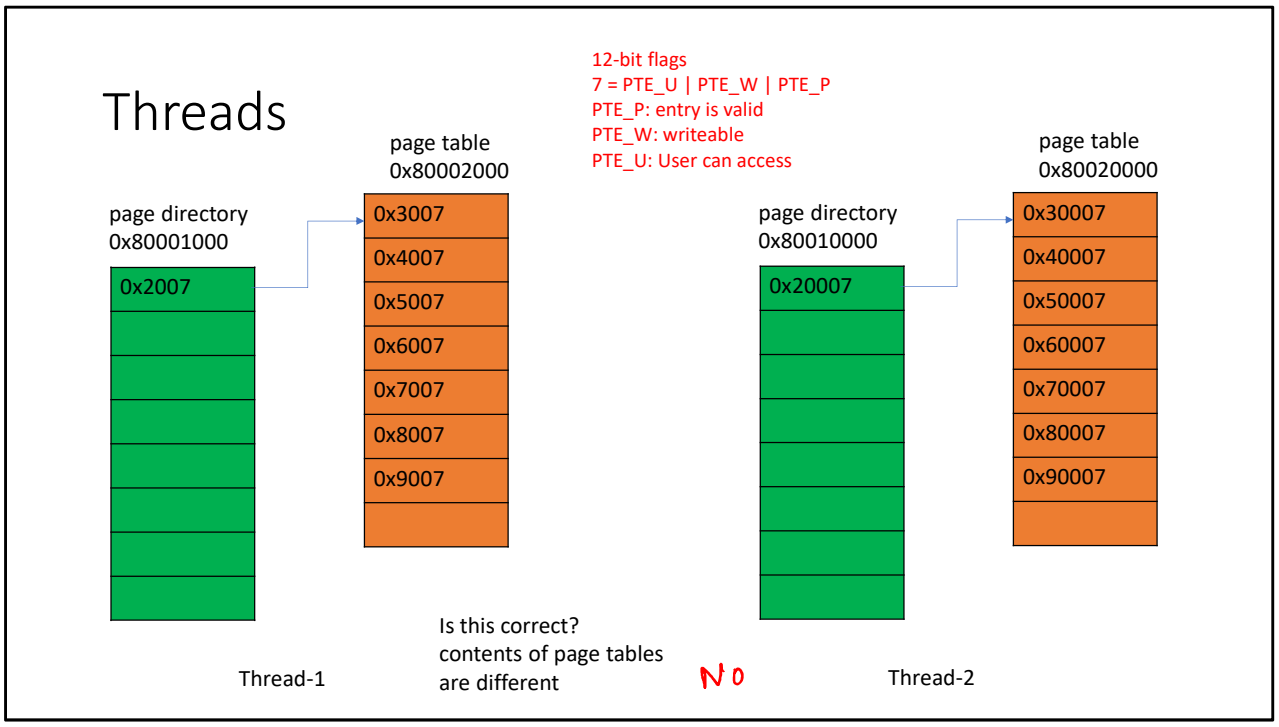
- Do we need to create separate page tables for threads?

Threads

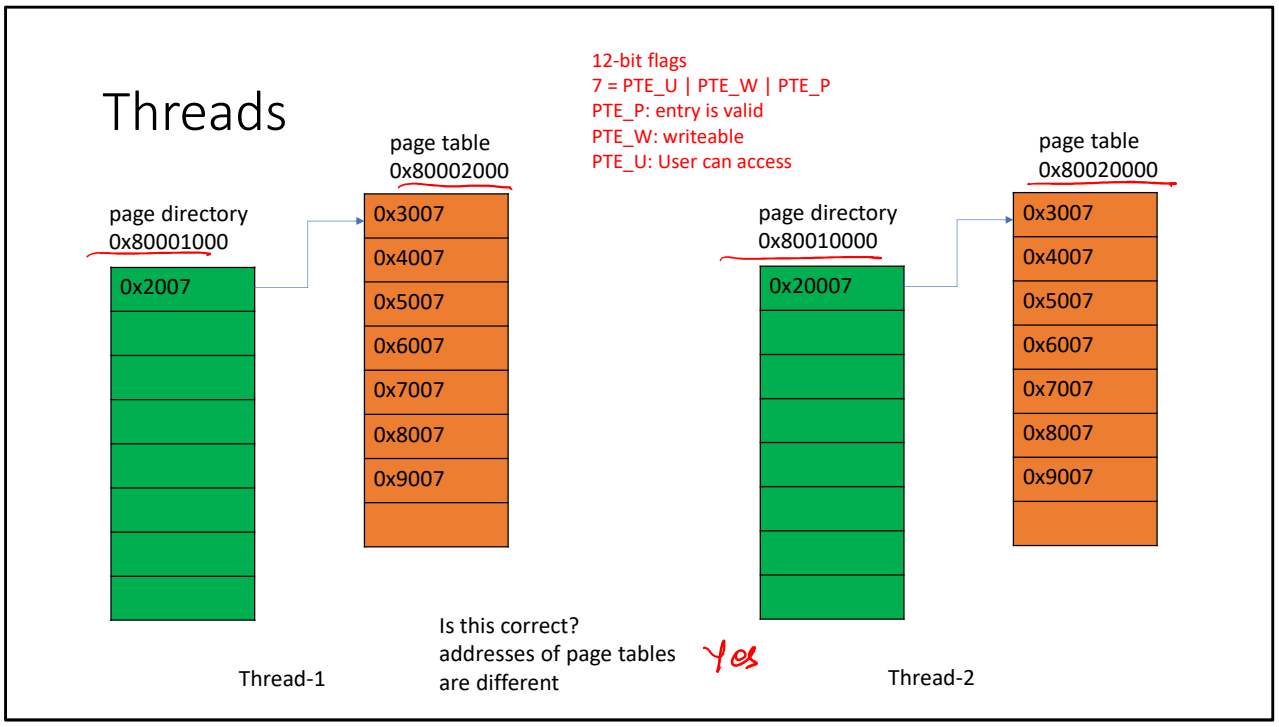
- Do we need to create separate page tables for threads?
  - No
- What can go wrong if we have separate page tables for threads?

Threads

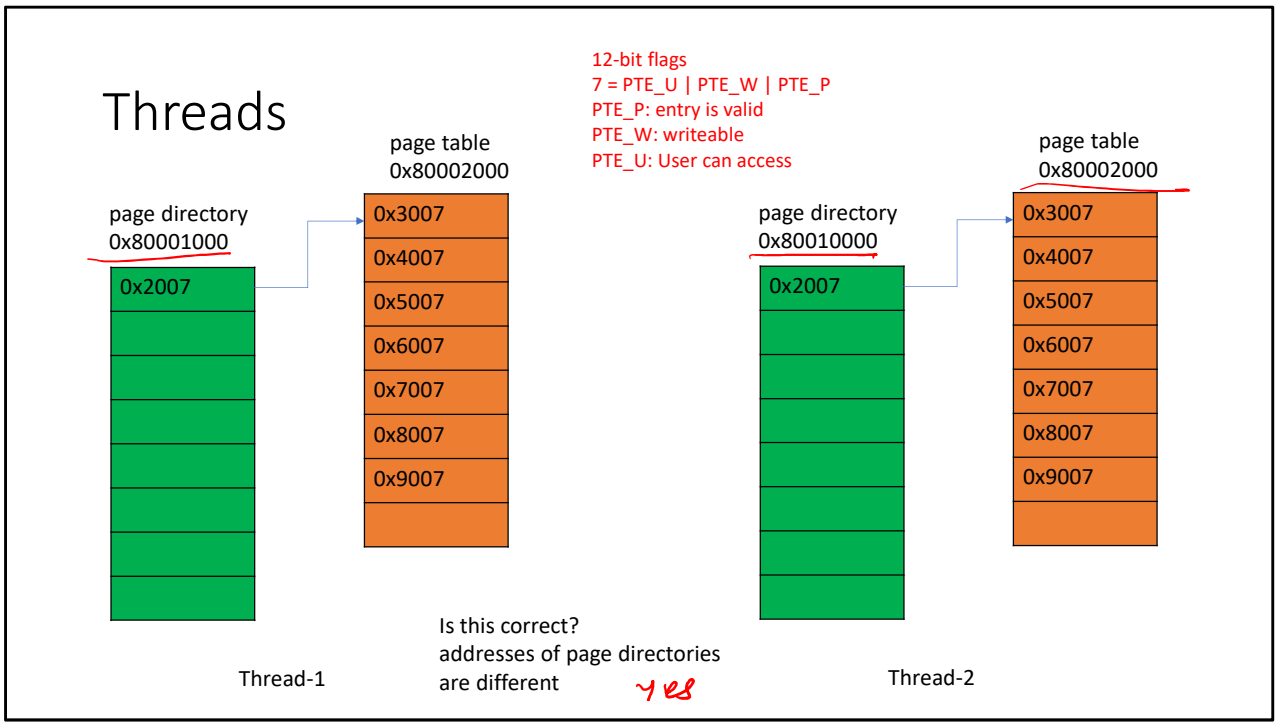
- Do we need to create separate page tables for threads?
- What can go wrong if we have separate page tables for threads?
  - If they map different physical address, then the values of global variables, code, etc., will be different in both the threads



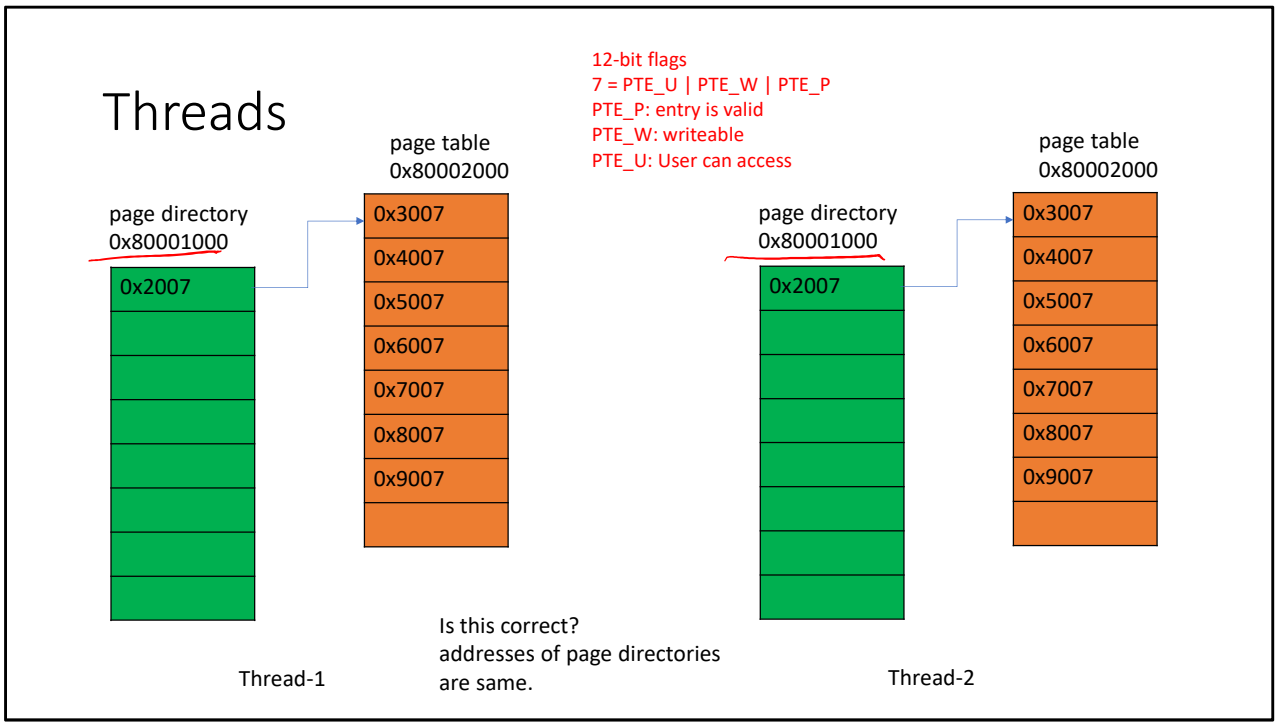
This is not correct because both the threads should see the same VA to PA mapping.



This is correct because even though the page tables addresses in thread-1 and thread-2 are different, their contents are the same. So, both the threads will see the same VA to PA mapping.



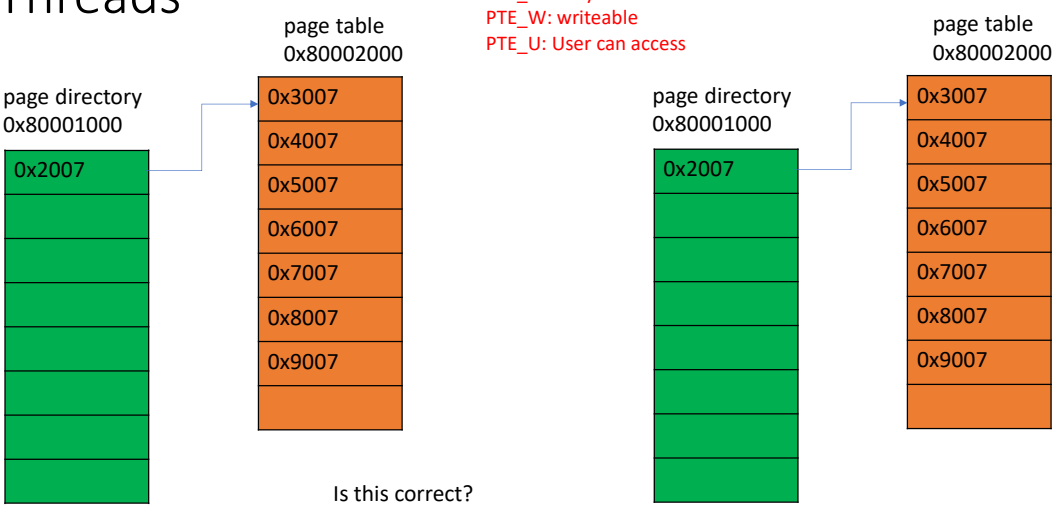
This is correct because even though the page directories addresses in thread-1 and thread-2 are different, they point to the same page table. So, both the threads will see the same VA to PA mapping.



This is the best approach. Both threads should share the same page directory.

Threads

12-bit flags
7 = PTE_U | PTE_W | PTE_P
PTE_P: entry is valid
PTE_W: writeable
PTE_U: User can access



Thread-1

Is this correct?
addresses of page directories are same.
Yes, this is the best choice.

Thread-2

Threads

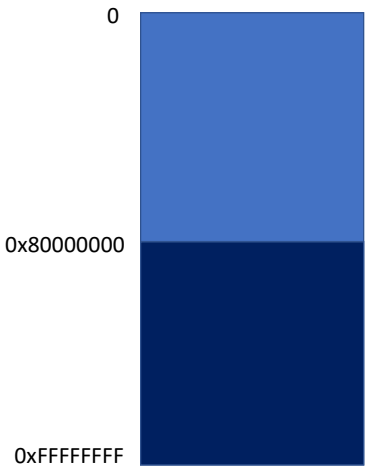
- Why can multiple threads share the same page table?

Threads

- Why can multiple threads share the same page table?
  - because threads belong to the same process and share the same address space

Process address space

- Most OSes reserve some space for the kernel in the process address space
  - xv6 reserves 0x80000000 - 0xFFFFFFFF for the kernel
  - Linux reserves 0xC0000000 – 0xFFFFFFFF for the kernel
  - Windows also does something similar
- Can we map the kernel at the different addresses in different processes?



Process address space

- Can we map the kernel at the different addresses in different processes?
  - Let us say we have two processes P1 and P2
    - P1 maps the kernel at 0x80000000 – 0xC0000000
    - P2 maps the kernel at 0xC0000000 – 0xFFFFFFFF

Process address space

P1 : KERNBASE = 0x80000000

P2: KERNBASE = 0xC0000000

```
create_thread_syscall() {
   struct thread *t = kmalloc(sizeof(struct thread));
   push_list(t);
   …
}
```

P1 : KERNBASE = 0x80000000
P2: KERNBASE = 0xC0000000

P1 creates T1
create_thread syscall is invoked
struct thread is allocated using kmalloc
kmalloc returns 0x80001000
0x80001000 was added to ready list
P2 is scheduled
P2 calls scheduler
schedule picks T1
schedule accesses T1's struct thread (0x80001000)
invalid address according to P2's page table
kernel panic

because the kernel is shared among all the processes all the kernel addresses need to be consistent

If we load the kernel in different processes at different addresses, the addresses of global variables and heap will be different in different processes. Because the kernel is shared among all the processes, it may cause inconsistencies, as shown on this slide.

Kernel

- Different page table for the kernel
- What are the advantages of using a separate page table for the kernel?
  - Better security
    - the kernel can't access user's pointers directly
    - the kernel can't accidentally write its private data to the user's address space

Because as of now, there is no guarantee that the kernel will always do the right thing. It is possible that due to some bug, the kernel may accidentally write its data to the user's memory because the kernel can access all the virtual addresses that are mapped in the page table. One way of preventing this would be to have different page tables for user and kernel. In this case, if the kernel accidentally tries to write to the user's memory, it will cause an exception.

Kernel

- How to implement separate page table for the kernel
  - The hardware doesn't automatically switch the page tables during interrupts/exceptions
  - interrupt/exception wrappers need to be mapped in the user address space
  - the kernel stack needs to be mapped in the user address space
  - the user buffer is copied to the kernel stack in the wrappers

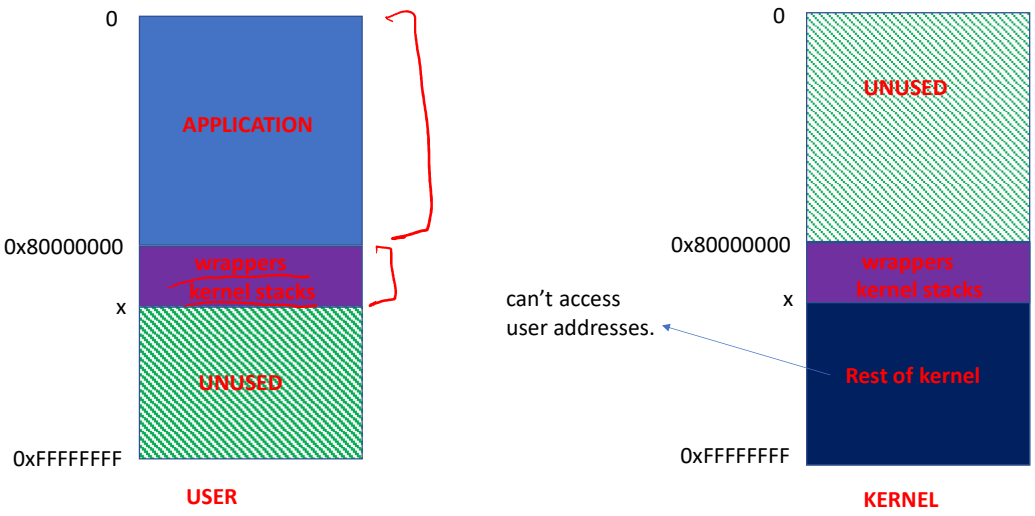
wrapper

```
system_call:
save user registers
char kbuf[128];
copy_from_user(kbuf, ubuf, 128);
load kernel page table
system_call_k(syscall_id, kbuf);
load user page table
copy_to_user(ubuf, kbuf, 128);
restore user registers
iret
```

The only place copy_from_user and copy_to_user are called is the wrappers. The user pointers are never directly accessed in the rest of the kernel.

If the kernel is mapped in a different page table, then we need to switch page tables on entry/exit to/from the kernel. Because, the hardware doesn't automatically do this, the kernel stack and some minimal interrupt handlers (we call it kernel wrappers) need to be mapped in the user's page table. The system call wrapper: copies argument from the user's address to the kernel stack, loads the page table of the kernel, and passes the arguments and system call identifier to a kernel routine. Before returning the user-mode: the wrapper restores the user page table, copies the return values (if any) to the user buffer, and goes back to the user mode.

user and kernel page table



The UNUSED area generates an exception on access.

Kernel

- What is the downside of using a different page table for the kernel
  - page table reloads are expensive

Two dimensional page tables

- What is the downside of two-dimensional page tables
  - two extra memory references for each memory access

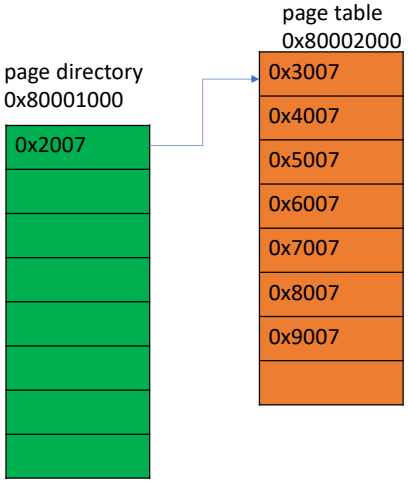
Translation lookaside buffer(TLB)

- Per core, TLB is designed to reduce the number of page table accesses
- Accesses to TLB are very fast
- TLB caches the VPN-PPN entries
- TLB is very small (typically 512 entries)

TLB

- Before walking the page table, the hardware first checks if the VPN is present in the TLB
  - if yes, the hardware uses the corresponding entry for the address translation
    - also called TLB hit
  - otherwise, the hardware walks the page table for address translation
    - TLB miss
- For most applications, TLB hit rate is greater than 99.9%
  - TLB makes page tables practical
  - no severe impact on performance

App page table

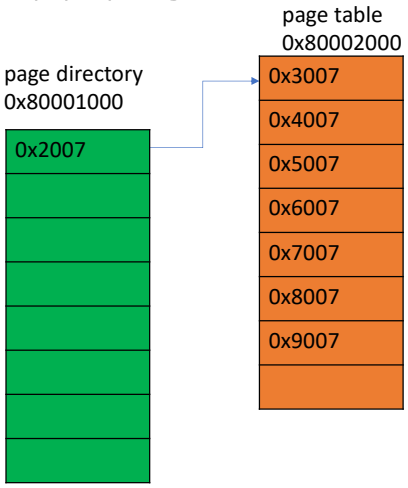


12-bit flags
7 = PTE_U | PTE_W | PTE_P
PTE_P: entry is valid
PTE_W: writeable
PTE_U: User can access

TLB

VPN	PTE

App page table



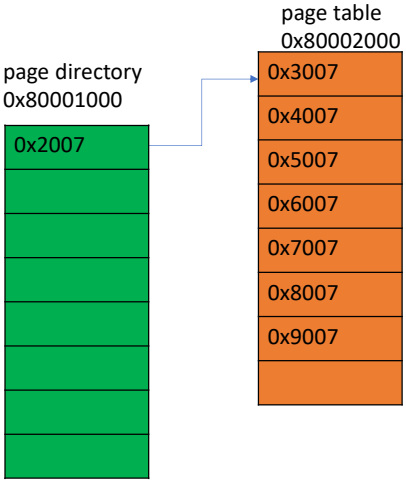
12-bit flags
7 = PTE_U | PTE_W | PTE_P
PTE_P: entry is valid
PTE_W: writeable
PTE_U: User can access

App accesses VPN 0
TLB miss
hardware walks page table

TLB

VPN	PTE

App page table



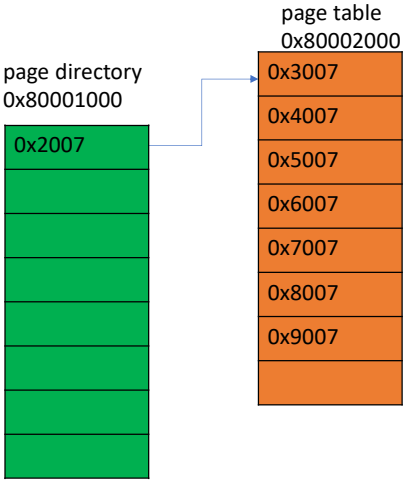
12-bit flags
7 = PTE_U | PTE_W | PTE_P
PTE_P: entry is valid
PTE_W: writeable
PTE_U: User can access

App accesses VPN 0
TLB miss
hardware walks page table
TLB is updated

TLB

VPN	PTE
0	0x3007

App page table



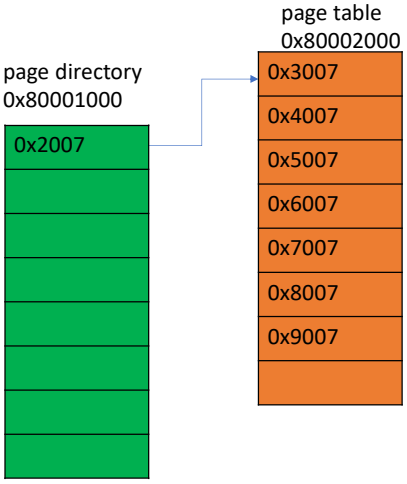
App accesses VPN 0
TLB miss
hardware walks page table
TLB is updated
App accesses VPN 2
TLB miss
hardware walks page table

12-bit flags
7 = PTE_U | PTE_W | PTE_P
PTE_P: entry is valid
PTE_W: writeable
PTE_U: User can access

TLB

VPN	PTE
0	0x3007

App page table



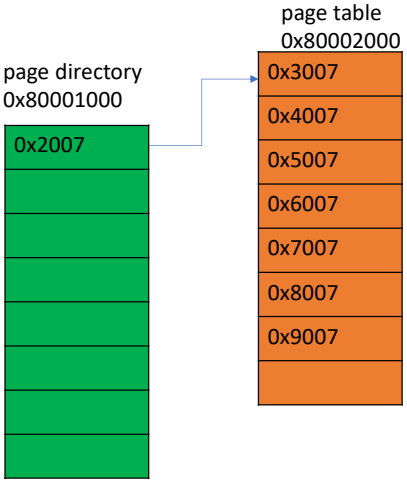
App accesses VPN 0
TLB miss
hardware walks page table
TLB is updated
App accesses VPN 2
TLB miss
hardware walks page table
TLB is updated

12-bit flags
7 = PTE_U | PTE_W | PTE_P
PTE_P: entry is valid
PTE_W: writeable
PTE_U: User can access

TLB

VPN	PTE
0	0x3007
2	0x5007

App page table



App accesses VPN 0
TLB miss
hardware walks page table
TLB is updated
App accesses VPN 2
TLB miss
hardware walks page table
TLB is updated
hardware accesses VPN 0
TLB hit
no page table walk

12-bit flags
7 = PTE_U | PTE_W | PTE_P
PTE_P: entry is valid
PTE_W: writeable
PTE_U: User can access

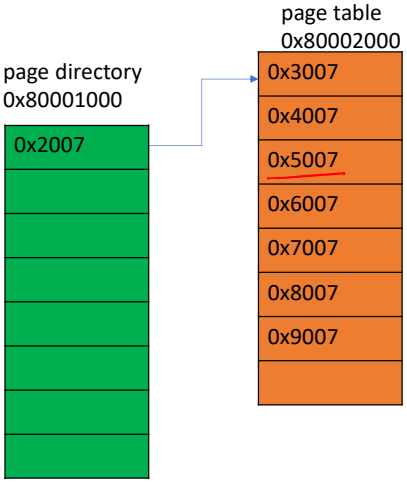
TLB

VPN	PTE
0	0x3007
2	0x5007

TLB

- TLB is not automatically updated when the page table entry is updated

App page table



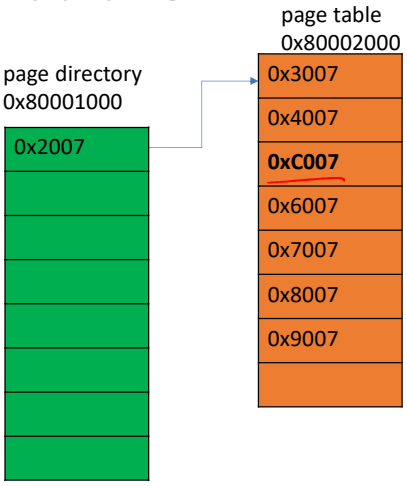
App accesses VPN 0
TLB miss
hardware walks page table
TLB is updated
App accesses VPN 2
TLB miss
hardware walks page table
TLB is updated
hardware accesses VPN 0
TLB hit
no page table walk

12-bit flags
7 = PTE_U | PTE_W | PTE_P
PTE_P: entry is valid
PTE_W: writeable
PTE_U: User can access

TLB

VPN	PTE
0	0x3007
2	0x5007

App page table



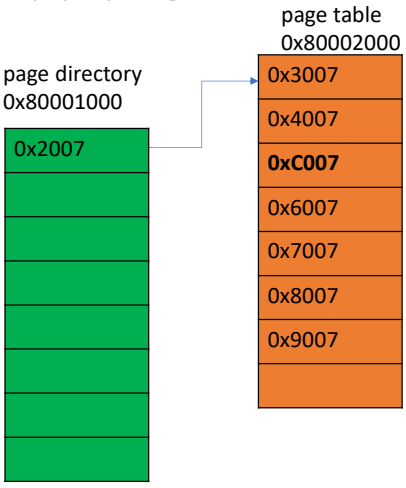
App accesses VPN 0
TLB miss
hardware walks page table
TLB is updated
App accesses VPN 2
TLB miss
hardware walks page table
TLB is updated
hardware accesses VPN 0
TLB hit
no page table walk
PTE of VPN 2 is updated

12-bit flags
7 = PTE_U | PTE_W | PTE_P
PTE_P: entry is valid
PTE_W: writeable
PTE_U: User can access

TLB

VPN	PTE
0	0x3007
2	0x5007

App page table



App accesses VPN 0
TLB miss
hardware walks page table
TLB is updated
App accesses VPN 2
TLB miss
hardware walks page table
TLB is updated
hardware access VPN 0
TLB hit
no page table walk
PTE of VPN 2 is updated
hardware accesses VPN 2
TLB hit
hardware accesses incorrect PA

12-bit flags
7 = PTE_U | PTE_W | PTE_P
PTE_P: entry is valid
PTE_W: writeable
PTE_U: User can access

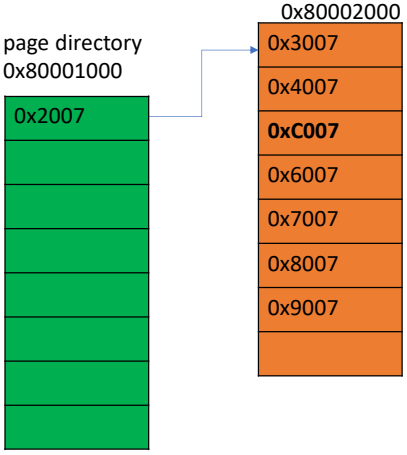
TLB

VPN	PTE
0	0x3007
2	0x5007

App page table

page table
0x80002000

page directory
0x80001000



App accesses VPN 0
TLB miss
hardware walks page table
TLB is updated
App accesses VPN 2
TLB miss
hardware walks page table
TLB is updated
hardware access VPN 0
TLB hit
no page table walk
PTE of VPN 2 is updated
~~hardware accesses VPN 2~~
~~TLB hit~~
~~hardware accesses incorrect PA~~

12-bit flags
7 = PTE_U | PTE_W | PTE_P
PTE_P: entry is valid
PTE_W: writeable
PTE_U: User can access

TLB

VPN	PTE
0	0x3007
2	0x5007

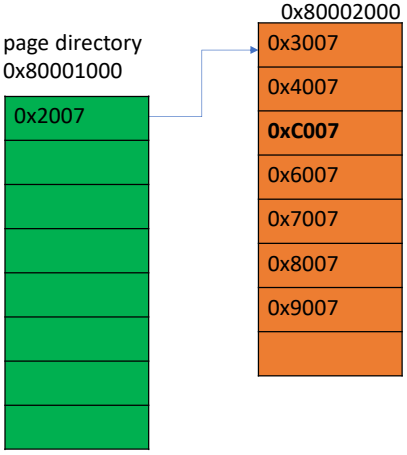
invlpg

- invlpg va invalidates the TLB entry corresponding to the va
- invlpg is a privilege instruction

App page table

page table
0x80002000

page directory
0x80001000



App accesses VPN 0
TLB miss
hardware walks page table
TLB is updated
App accesses VPN 2
TLB miss
hardware walks page table
TLB is updated
hardware access VPN 0
TLB hit
no page table walk
PTE of VPN 2 is updated
~~hardware accesses VPN 2~~
~~TLB hit~~
~~hardware accesses incorrect PA~~
invlpg 0x2000

12-bit flags
7 = PTE_U | PTE_W | PTE_P
PTE_P: entry is valid
PTE_W: writeable
PTE_U: User can access

TLB

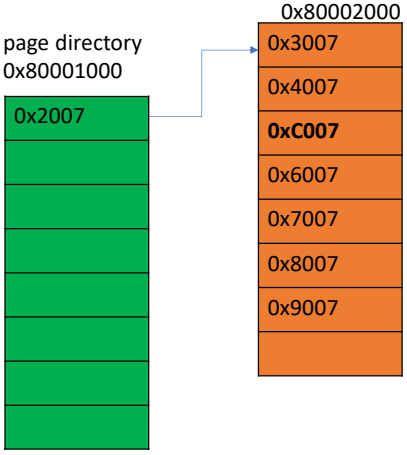
VPN	PTE
0	0x3007
~~2~~	~~0x5007~~

After updating a page table, the software must invalidate the corresponding entry in the TLB.

App page table

page table
0x80002000

page directory
0x80001000



App accesses VPN 0
TLB miss
hardware walks page table
TLB is updated
App accesses VPN 2
TLB miss
hardware walks page table
TLB is updated
hardware access VPN 0
TLB hit
no page table walk
PTE of VPN 2 is updated
~~hardware accesses VPN 2~~
~~TLB hit~~
~~hardware accesses incorrect PA~~
invlpg 0x2000
hardware accesses VPN 2
TLB miss
hardware walks page table
TLB is updated
hardware accesses correct PA

12-bit flags
7 = PTE_U | PTE_W | PTE_P
PTE_P: entry is valid
PTE_W: writeable
PTE_U: User can access

TLB

VPN	PTE
0	0x3007
2	**0xc007**