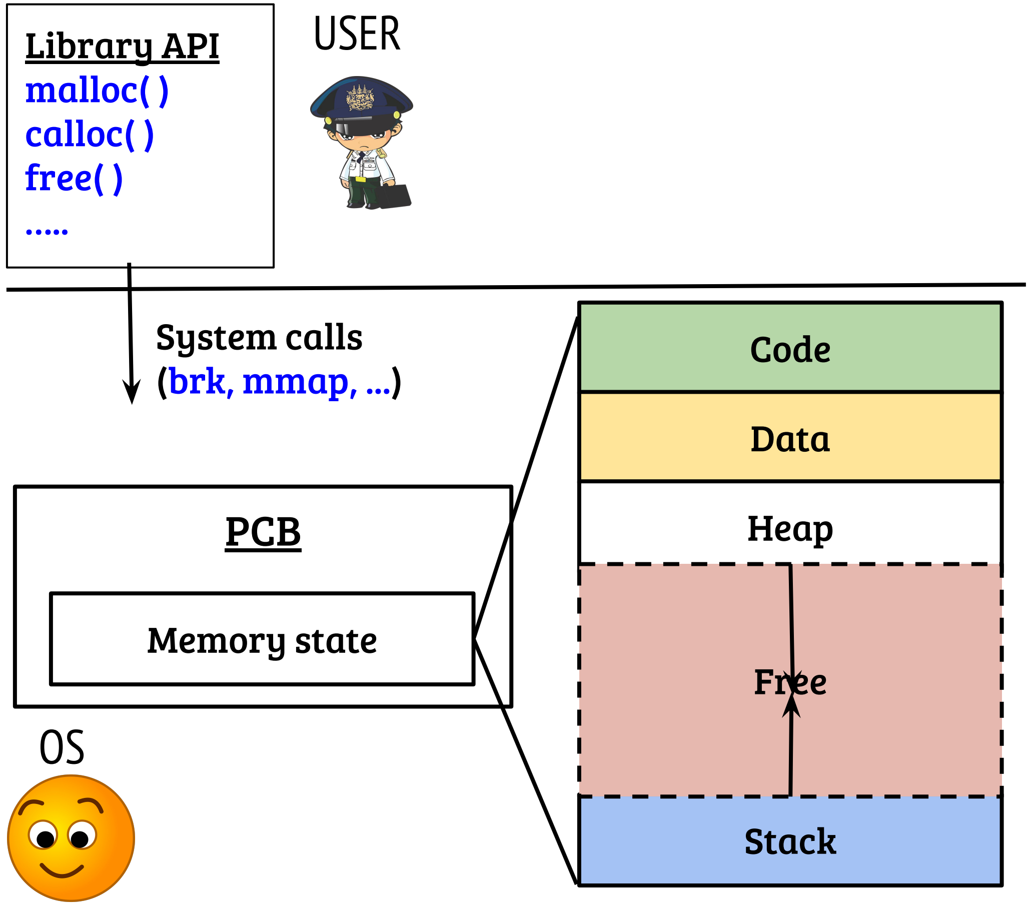
CS614: Linux Kernel Programming

Virtual Memory

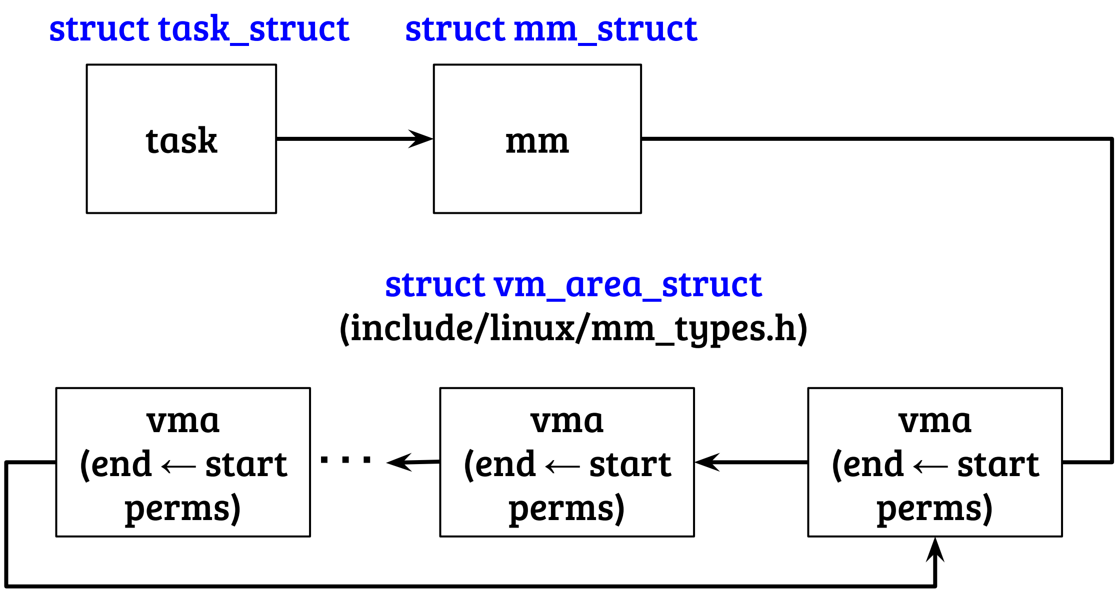
Debadatta Mishra, CSE, IIT Kanpur

User API for memory management



- Generally, user programs use library routines to allocate/deallocate memory
- OS provides some address space manipulation system calls (today's agenda)

Virtual memory management



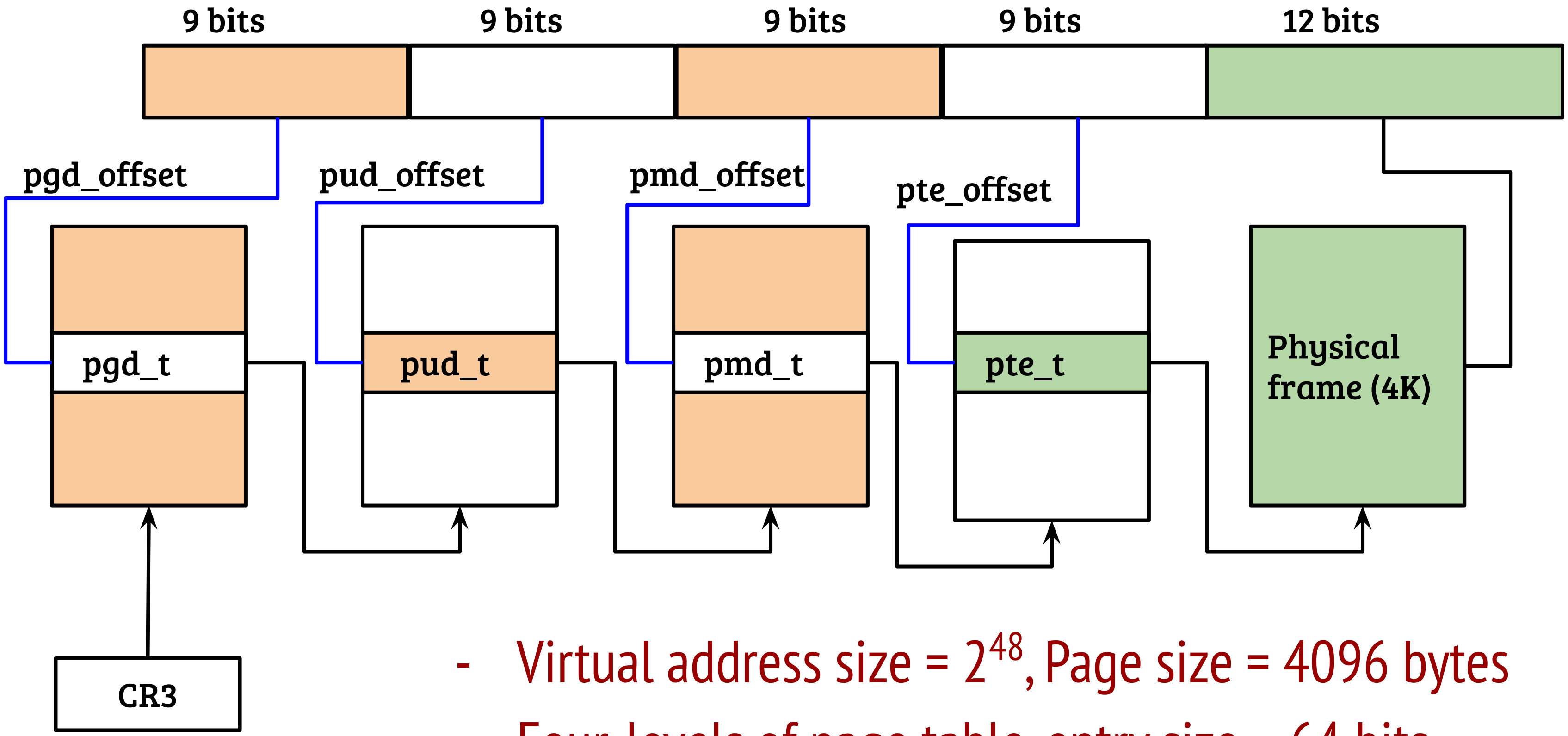
- start and end never overlaps between two vm areas
- can merge/extend vmas if permissions match
- linux maintains both rb_tree and a sorted list (see mm/filemap.c)

The OS implements VM system calls like mmap( ), mprotect( ) by manipulating the VMAs

Address translation: Paging

- The idea of paging
  - Partition the address space into fixed sized blocks (call it pages)
  - Physical memory partitioned in a similar way (call it page frames)
  - OS creates a mapping between *page* to *page frame* , H/W uses the mapping to translate VA to PA
- With increased address space size, single level page table entry is not feasible, because
  - Increasing page size increases internal fragmentation
  - Small pages may not be suitable to hold all mapping entries

4-level page tables: 48-bit VA (Intel x86_64)



- Virtual address size = 2^{48}, Page size = 4096 bytes
- Four-levels of page table, entry size = 64 bits

Paging example (structure of an example PTE)

- PFN occupies a significant portion of PTE entry (8 bits in this example)

P — Present bit, 1 ⇒ entry is valid

W — Write bit, 1 ⇒ Write allowed

S — Privilege bit, 0 ⇒ only kernel mode access is allowed

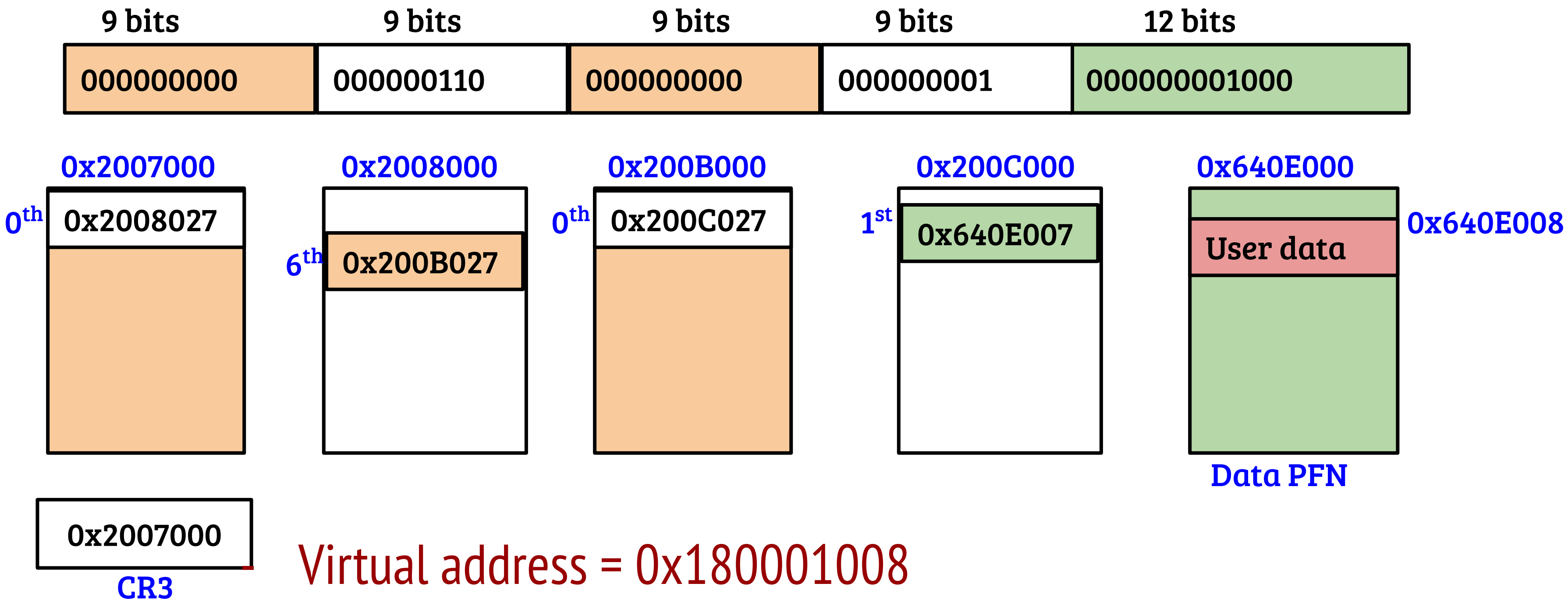
A — Accessed bit, 1 ⇒ Address accessed (set by H/W during walk)

D — Dirty bit, 1 ⇒ Address written (set by H/W during walk)

X — Execute bit, 1 ⇒ Instruction fetch allowed for this page

Reserved/unused bits

4-level page tables: example translation



- Virtual address = 0x180001008
- Hardware translation by repeated access of page table stored in physical memory
- Page table entry: 12 bits LSB is used for access flags

Paging: translation efficiency

```
sum = 0;
for(ctr=0; ctr<10; ++ctr)
      sum += ctr;
```

```
0x20100:  mov $0, %rax;
0x20102:  mov %rax, (%rbp);     // sum=0
0x20104:  mov $0, %rcx;         // ctr=0
0x20106:  cmp $10, %rcx;        // ctr < 10
0x20109:  jge  0x2011f;         // jump if >=
0x2010f:  add %rcx, %rax;
0x20111:  mov %rax, (%rbp);     // sum += ctr
0x20113:  inc %rcx              // ++ctr
0x20115:  jmp 0x20106           // loop
0x2011f:  .............
```

- Considering four-level page table, how many memory accesses are required (for translation) during the execution of the above code?

Paging: translation efficiency

```
0x20100:   mov $0, %rax;
0x20102:   mov %rax, (%rbp);        // sum=0
```

- Instruction execution: Loop = 10 * 6, Others = 2 + 3
  - Memory accesses during translation = 65 * 4 = 260
- Data/stack access: Initialization = 1, Loop = 10
  - Memory accesses during translation = 11 * 4 = 44
- A lot of memory accesses (> 300) for address translation
- How many distinct pages are translated?

- Considering four-level page table, how many memory accesses are required (for translation) during the execution of the above code?

Paging with TLB: translation efficiency

TLB

Page	PTE
0x20	0x750
0x7FFF	0x890

```
Translate(V){
    PageAddress P = V >> 12;
    TLBEntry entry = lookup(P);
    if (entry.valid) return entry.pte;
    entry = PageTableWalk(V);
    MakeEntry(entry);
    return entry.pte;
}
```

- TLB is a hardware cache which stores *Page* to *PFN* mapping
- After first miss for instruction fetch address, all others result in a TLB hit
- Similarly, considering the stack virtual address range as 0x7FFF000 - 0x8000000, one entry in TLB avoids page table walk after first miss

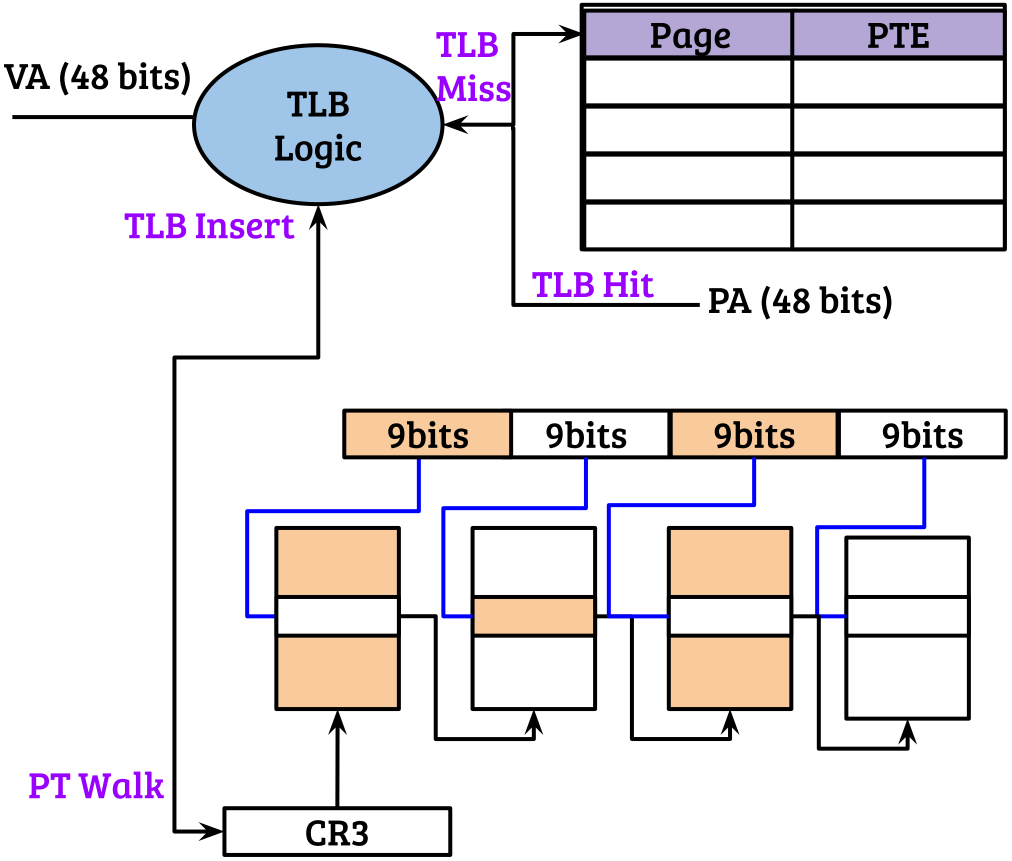
Paging: translation efficiency

0x20100:   mov $0, %rax;

- Instruction execution: Loop = 10 * 6, Others = 2 + 3
    - Memory accesses during translation = 65 * 4 = 260
- Data/stack access: Initialization = 1, Loop = 10
    - Memory accesses during translation = 11 * 4 = 44
- A lot of memory accesses (> 300) for address translation
- How many distinct pages are translated?
- One code page (0x20) and one stack page (0x7FFF). Caching these translations, will save a lot of memory accesses.

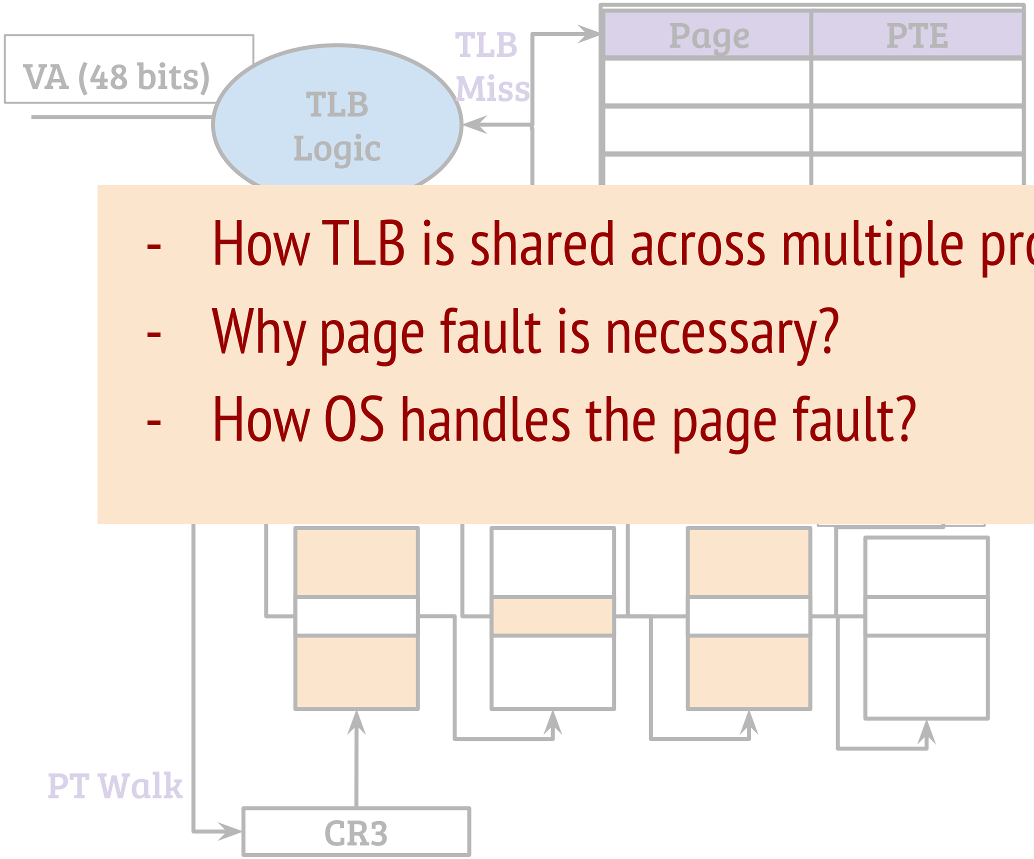
required (for translation) during the execution of the above code?

Address translation (TLB + PTW)



- TLB in the path of address translation
- Separate TLBs for instruction and data, multi-level TLBs
- In X86, OS can not make entries into the TLB directly, it can flush entries

Address translation (TLB + PTW)



- TLB in the path of address

- How TLB is shared across multiple processes?
- Why page fault is necessary?
- How OS handles the page fault?

d

into the TLB directly, it can flush entries

TLB: Sharing across applications

Process (A)	Process (B)

Page	PTE
0x100	0x200007
0x101	0x205007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution

TLB: Sharing across applications

Process (A) | Process (B)

Page	PTE
0x100	0x200007
0x101	0x205007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution
- Process B may be using the same addresses used by A. Result: Wrong translation

TLB: Sharing across applications

Process (A) | Process (B)

Page	PTE
~~0x100~~	~~0x200007~~
~~0x101~~	~~0x205007~~

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution
- Correctness ensured. Performance is an issue (with frequent context switching)

TLB: Sharing across applications

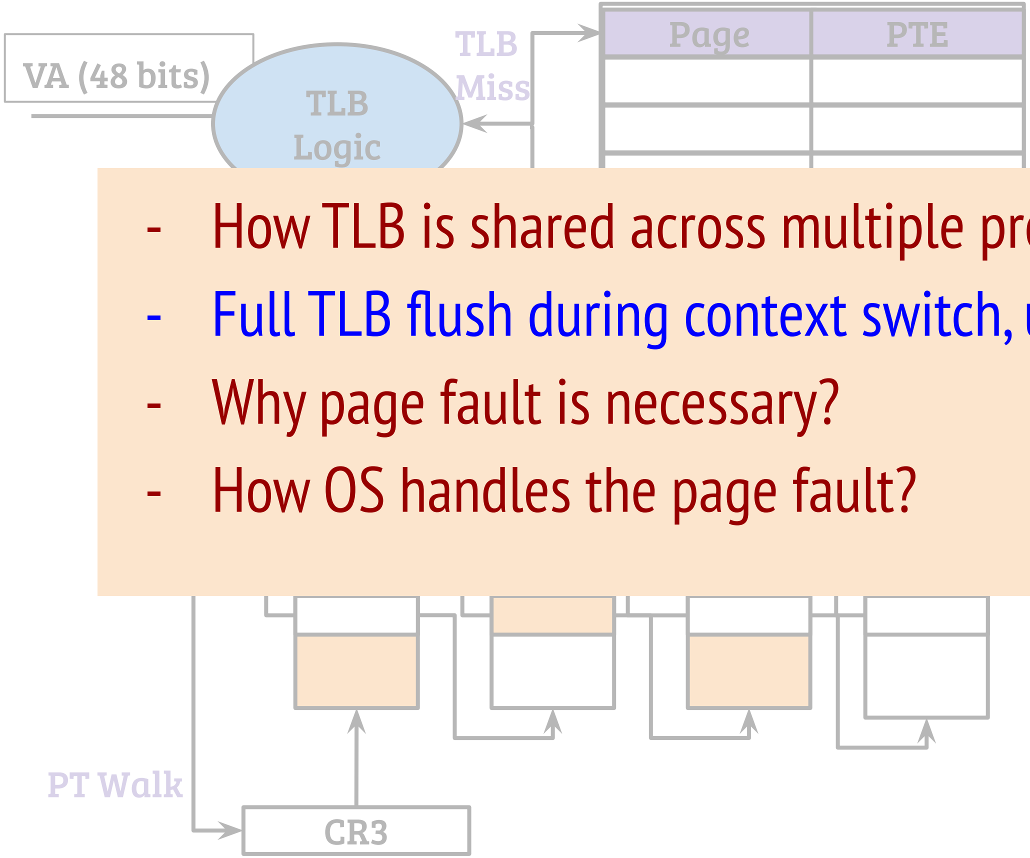
Process (A) | Process (B)

ASID	Page	PTE
A	0x100	ox200007
A	0x101	ox205007
B	0x100	0x301007
B	0x101	0x302007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution
- Address space identified (ASID) along with each TLB entry to identify the process

Address translation (TLB + PTW)



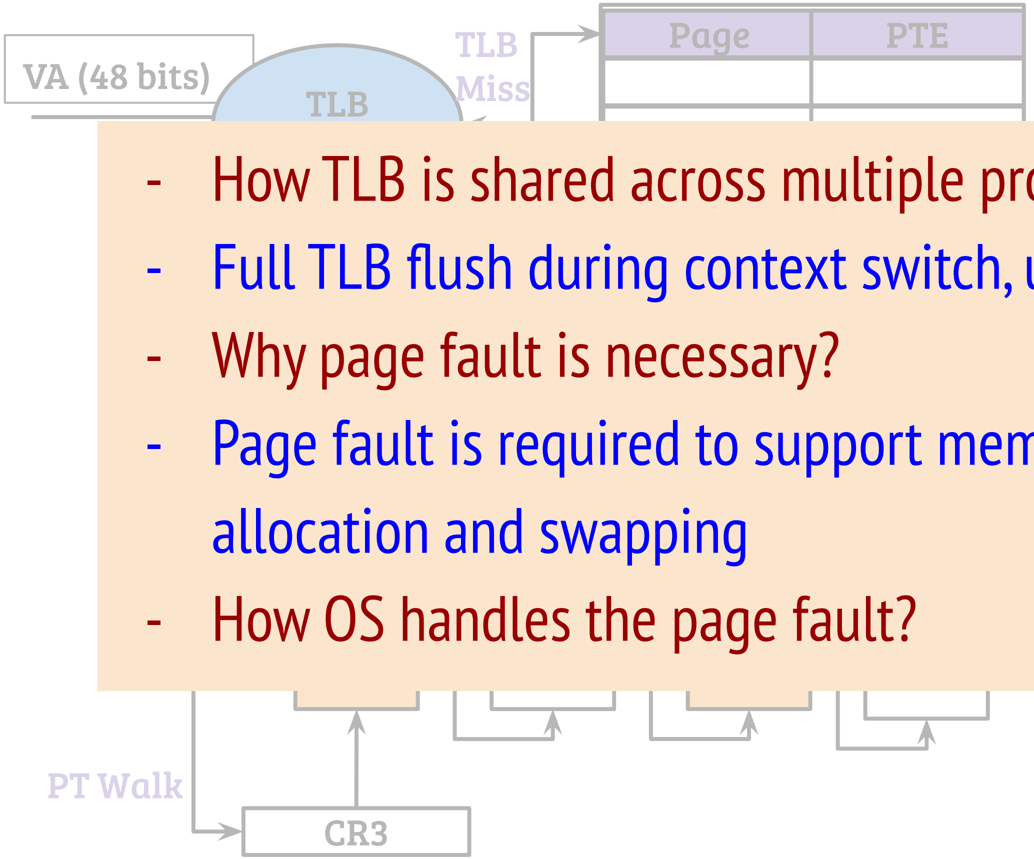
- TLB in the path of address

d

entries

- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- How OS handles the page fault?

Address translation (TLB + PTW)



- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- Page fault is required to support memory over-commitment through lazy allocation and swapping
- How OS handles the page fault?

Page fault handling in X86: Hardware

```
If( !pte.valid ||
    (access == write && !pte.write) ||
    (cpl != 0 && pte.priv == 0)){
        CR2 = Address;
        errorCode = pte.valid
                    | access << 1
                    | cpl << 2;
        Raise pageFault;
} // Simplified
```

Page fault handling in X86: Hardware

```
If( !pte.valid ||
    (access == write && !pte.write) ||
    (cpl != 0 && pte.priv == 0)){
        CR2 = Address;
        errorCode = pte.valid
                    | access << 1
                    | cpl << 2;
        Raise pageFault;
} // Simplified
```

Error code

Other and unused	I	R	U	W	P

- **P** — **Present bit, 1 ⇒ fault is due to protection**
- **W** — **Write bit, 1 ⇒ Access is write**
- **U** — **Privilege bit, 1 ⇒ Access is from user mode**
- **R** — **Reserved bit, 1 ⇒ Reserved bit violation**
- **I** — **Fetch bit, 1 ⇒ Access is Instruction Fetch**

- Error code is pushed into the kernel stack by the hardware

Page fault handling in X86: OS fault handler

```
HandlePageFault( u64 address, u64 error_code)
{
    If ( AddressExists(current → mm_state, address) &&
        AccessPermitted(current → mm_state, error_code) {
            PFN = allocate_pfn( );
            install_pte(address, PFN);
            return;
        }
    RaiseSignal(SIGSEGV);
}
```

Address translation (TLB + PTW)

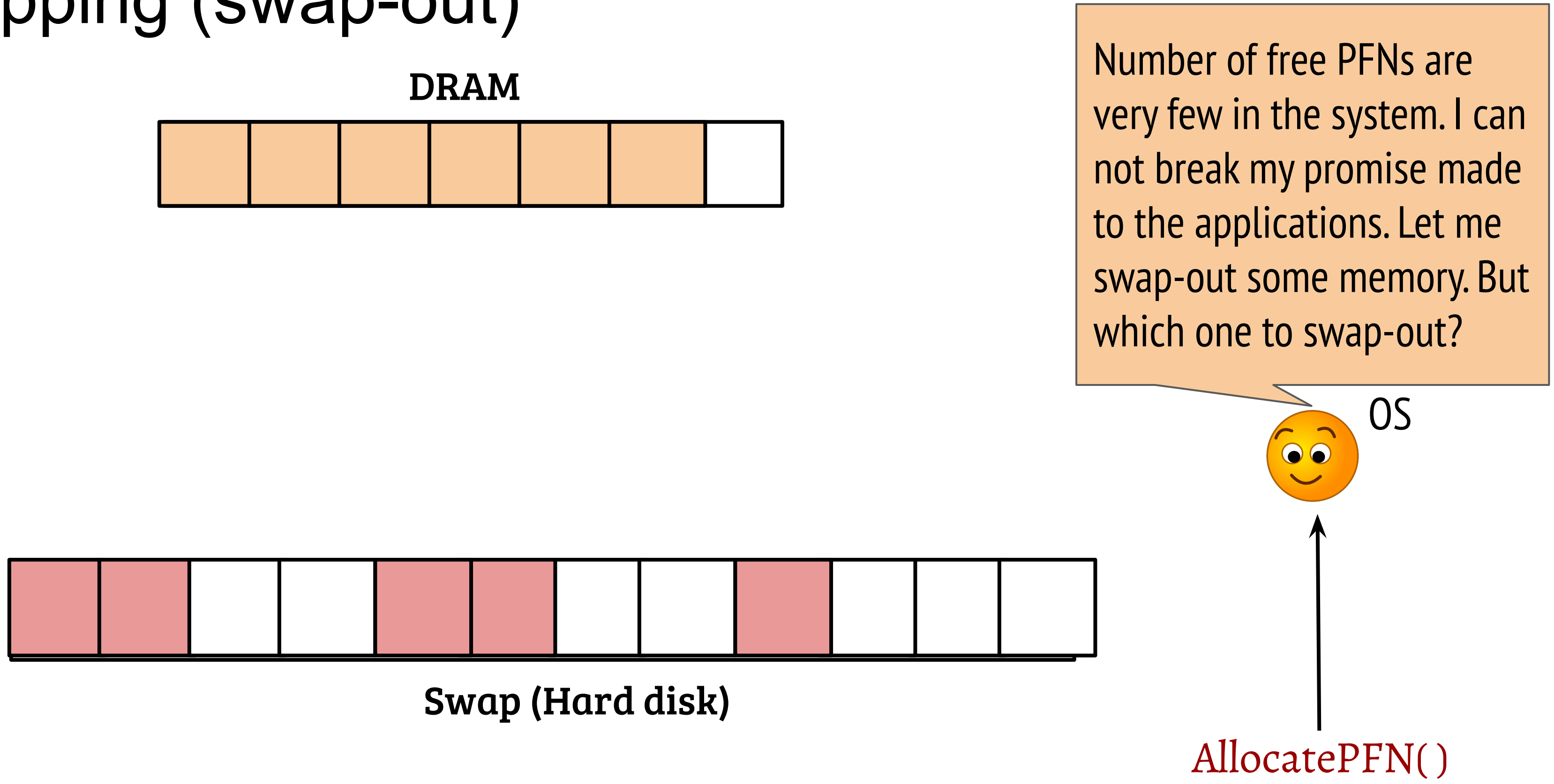
VA (4

- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- Page fault is required to support memory over-commitment through lazy allocation and swapping
- How OS handles the page fault?
- The hardware invokes the page fault handler by placing the error code and virtual address. The OS handles the page fault either fixing it or raising a SEGFAULT.

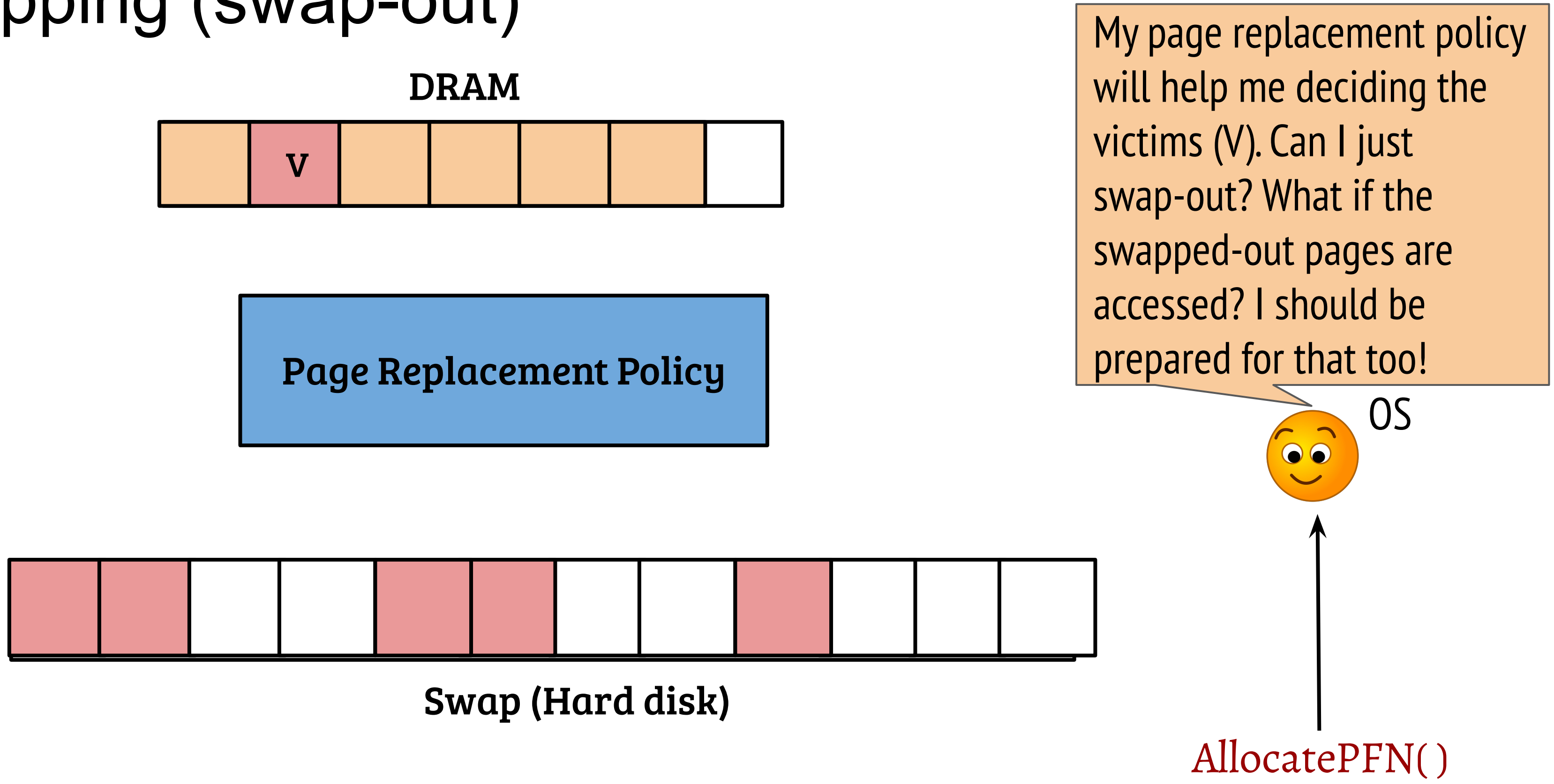
d

PT

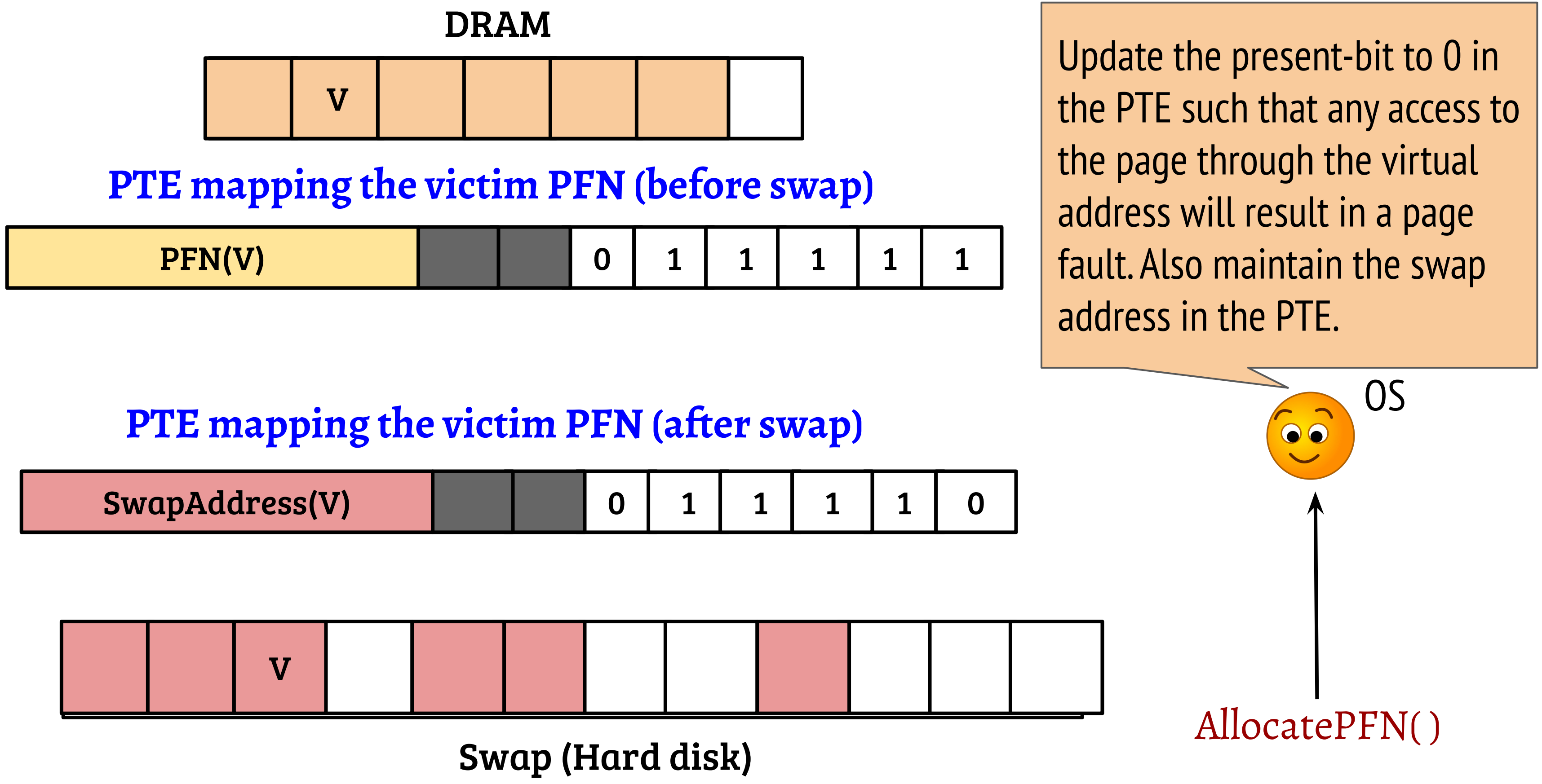
Swapping (swap-out)



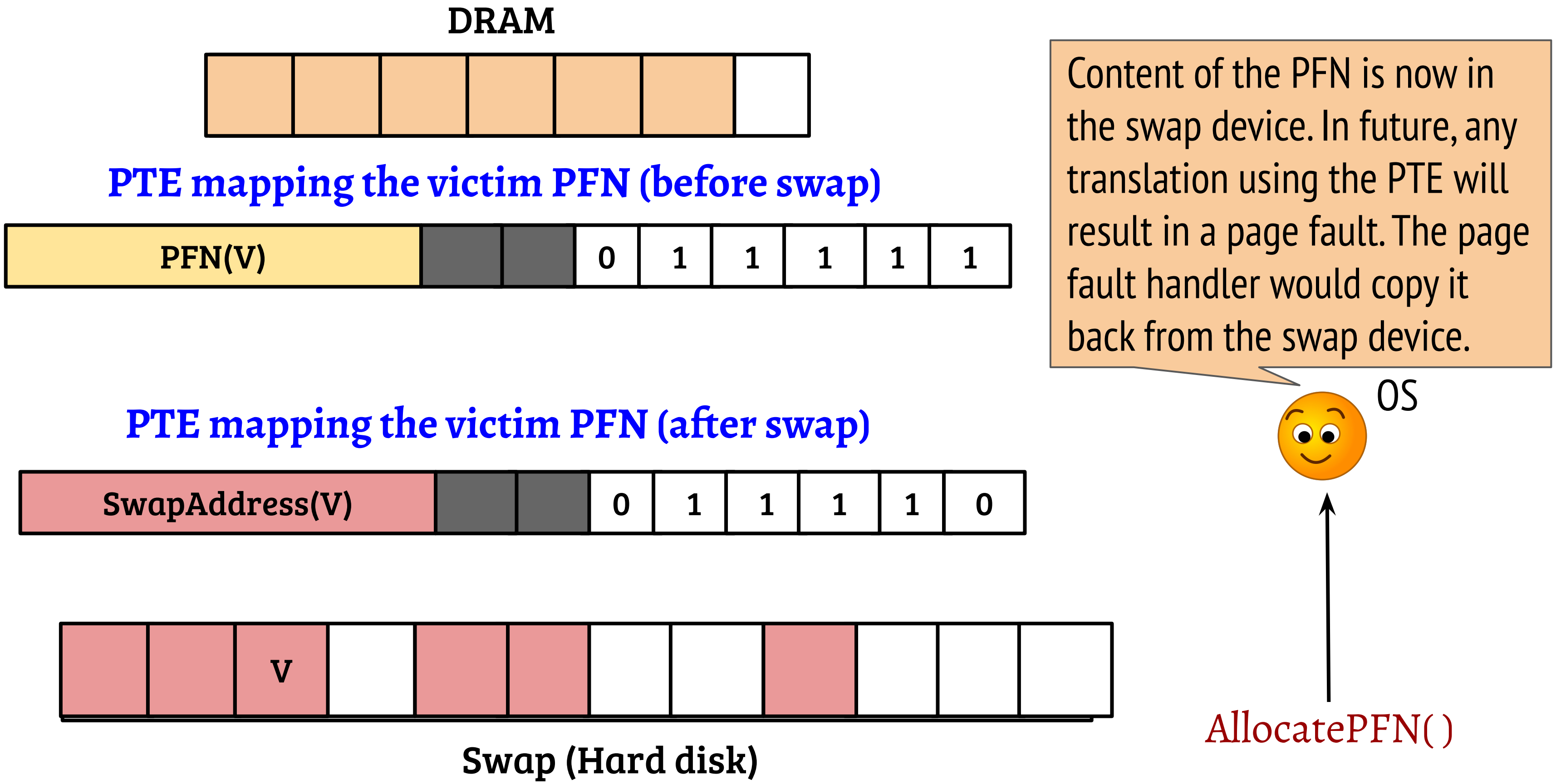
Swapping (swap-out)



Swapping (swap-out)



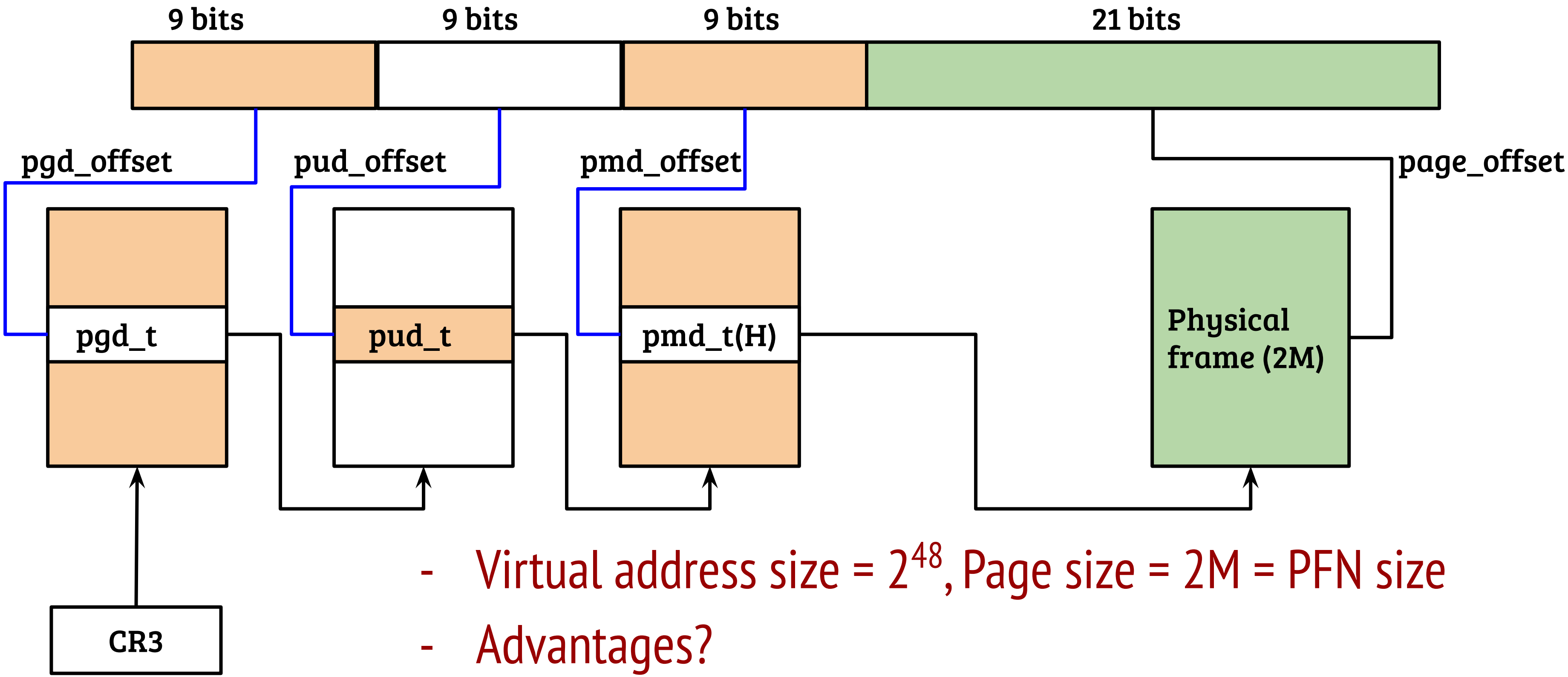
Swapping (swap-out)



Page fault with swap-in

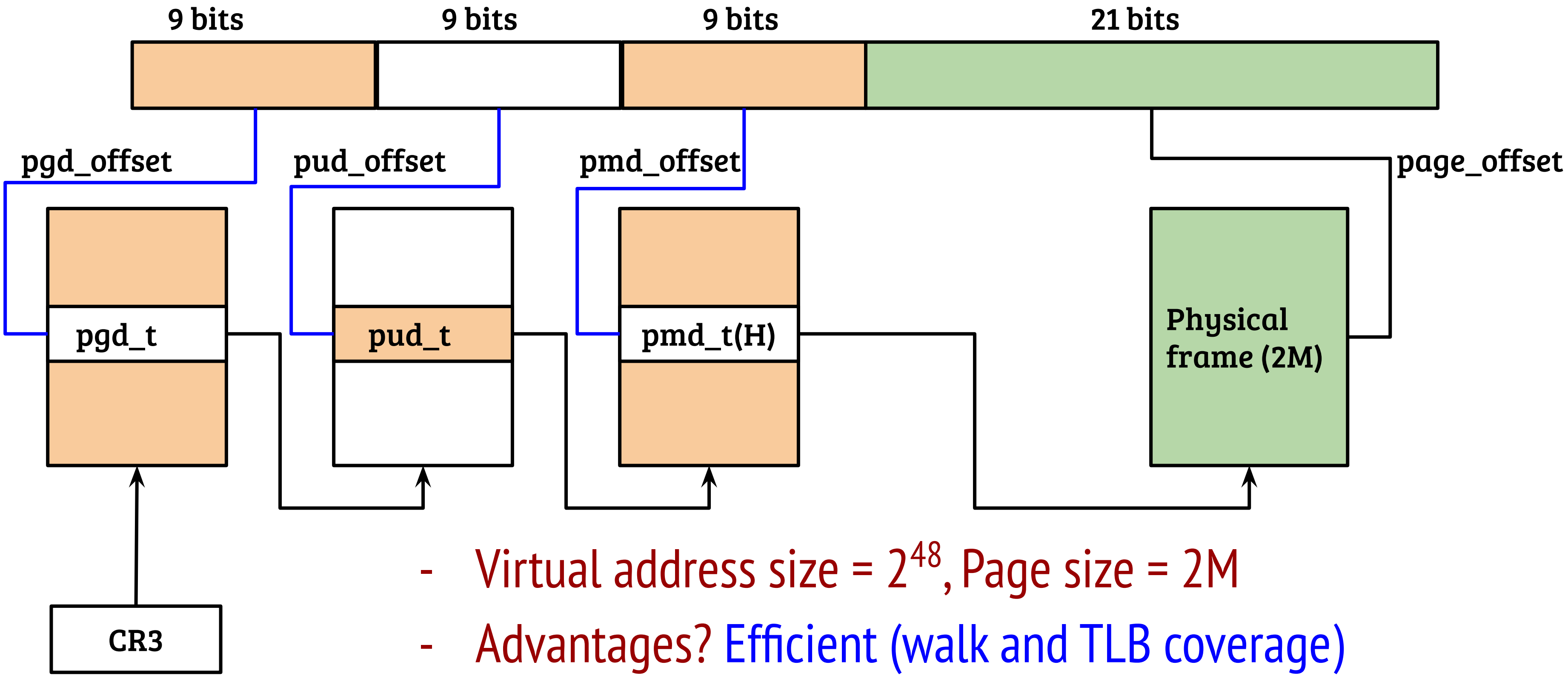
```
HandlePageFault( u64 address, u64 error_code)
{
    If ( AddressExists(current → mm_state, address) &&
        AccessPermitted(current → mm_state, error_code) {
            PFN = allocate_pfn( );
            If ( is_swapped_pte(address) )       // Check if the PTE is swapped out
              swapin(getPTE(address), PFN);    // Copy the swap block  to PFN
            install_pte(address, PFN);           // and update the PTE
           return;
        }
    RaiseSignal(SIGSEGV);
}
```

Efficient translation: Huge page support



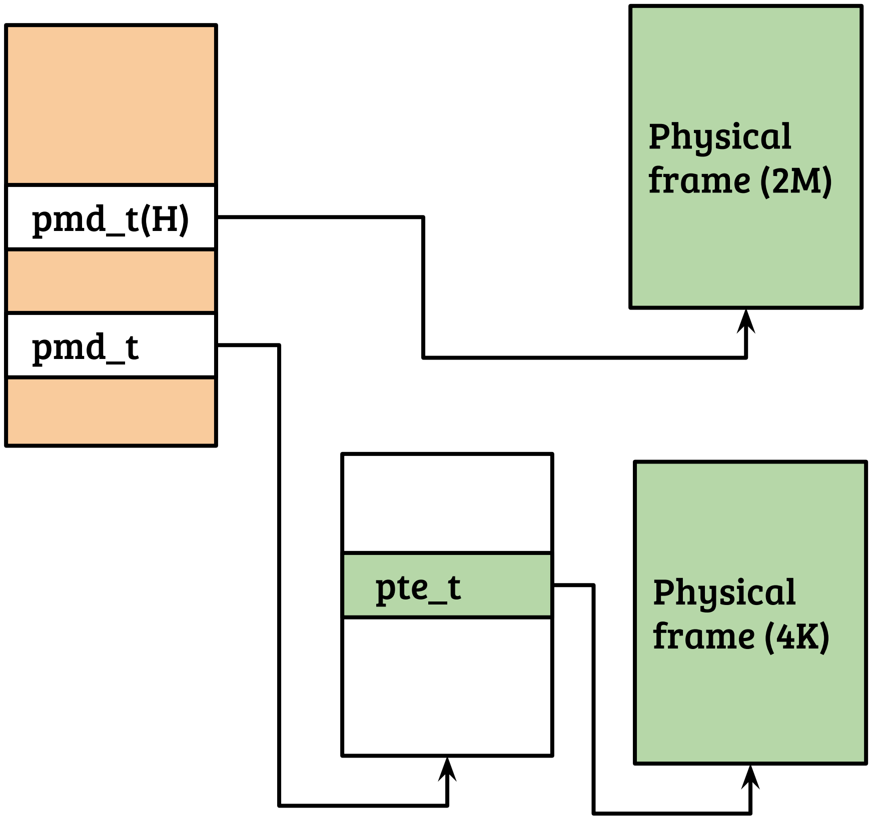
- Virtual address size = 2^{48}, Page size = 2M = PFN size
- Advantages?
- Disadvantages?

Efficient translation: Huge page support



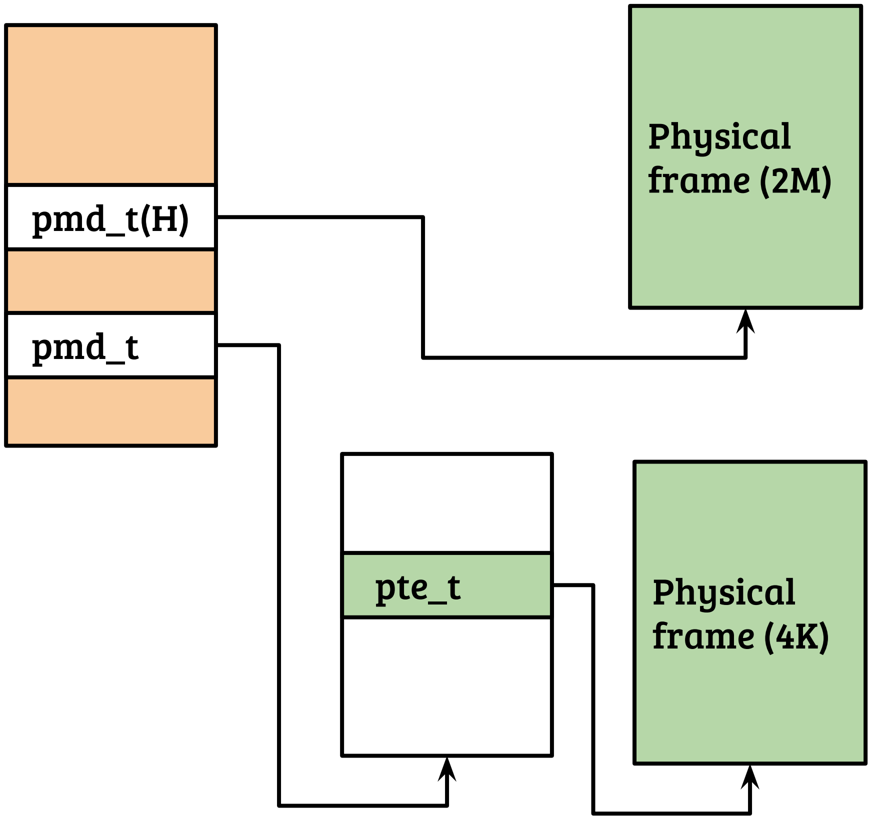
- Virtual address size = 2^{48}, Page size = 2M
- Advantages? Efficient (walk and TLB coverage)
- Disadvantages? Inefficient management

Mixed page size support




```
walk_pmd(pmd, vaddr) {
  if(pmd.H)
     paddr = pmd.nextL( ) + (vaddr & pmask);
 else
      pte = pmd.nextL( ) + pte_offset(vaddr)
} // Simplified H/W logic
```

Mixed page size support




```
walk_pmd(pmd, vaddr) {
   if(pmd.H)
      paddr = pmd.nextL( ) + (vaddr & pmask);
  else
       pte = pmd.nextL( ) + pte_offset(vaddr)
} // Simplified H/W logic
```

- The OS may use the hardware support to implement any policy
- Transparent hugepage (THP) in Linux trie to create huge page mapping in w/o explicit user space assistance
- Policy knobs through sysfs

Kernel Virtual Memory

- Why not treat kernel as an isolated MM context?

Kernel Virtual Memory

- Why not treat kernel as an isolated MM context?
  - Require MM context loading/unloading on user-kernel context switch
  - In kernel context, user data is accessed (a lot!) why?
  - Even worse, user data of many processes accessed
  - In X86, a small part of the kernel can not be isolated as HW does not perform MM context switch
- Requirement: efficient memory isolation between user and kernel

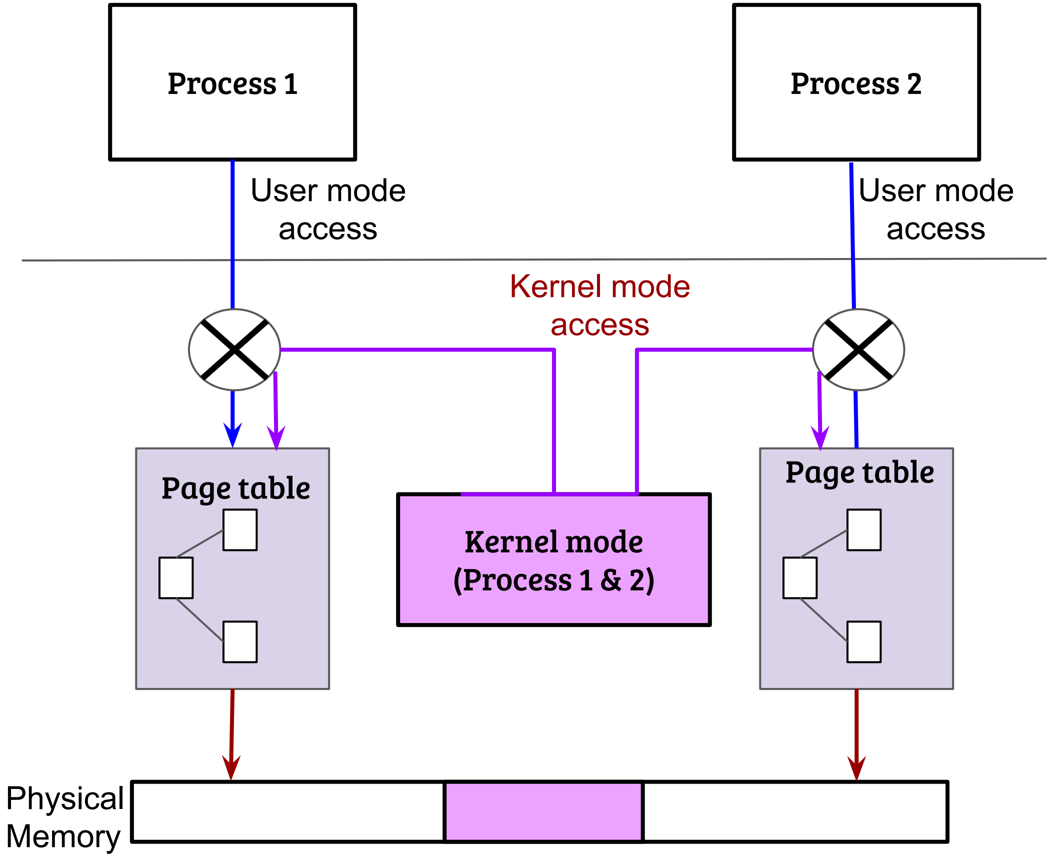
Kernel Virtual Memory

- Why not treat kernel as an isolated MM context?
  - Require MM context loading/unloading on user-kernel context switch
  - In kernel context, user data is accessed (a lot!) why?
  - Even worse, user data of many processes can be accessed
  - In X86, a small part of the kernel can not be isolated as HW does not perform MM context switch
- Requirement: efficient memory isolation between user and kernel
  - Let kernel use the same MM context of the user process
  - No context switch, no problems of accessing user data

Kernel Virtual Memory

- Why not treat kernel as an isolated MM context?
  - Require MM context loading/unloading on user-kernel context switch
  - In kernel context, user data is accessed (a lot!) why?
  - Even worse, user data of many processes can be accessed
  - In X86, a small part of the kernel can not be isolated as HW does not perform MM context switch
- Requirement: efficient memory isolation between user and kernel
  - Let kernel use the same MM context of the user process
  - No context switch, no problems of accessing user data
- How kernel VM change propagated across processes? Isolation issues?

Issue of Kernel VM propagation



- Kernel virtual address mapping should be present in both process page tables.
- Ex: If kernel allocates memory while serving syscall from process-1, process-2 in kernel mode should see it!
- Solution should consider that, "processes and memory are dynamically created and destroyed"

Linux strives on family values!

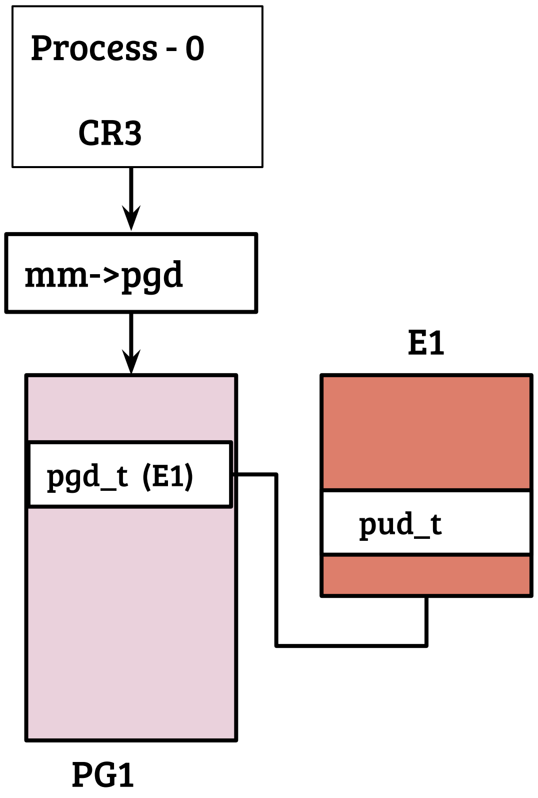
- A child process page table inherits the kernel mappings of the parent
- By implication, the inheritance tree is rooted at the first process
- Mapping changes $\longrightarrow$ update mapping in every process?
    - Does not look good!

Linux strives on family values!

- A child process page table inherits the kernel mappings of the parent
- By implication, the inheritance tree is rooted at the first process
- Mapping changes → update mapping in every process?
  - Does not look good!

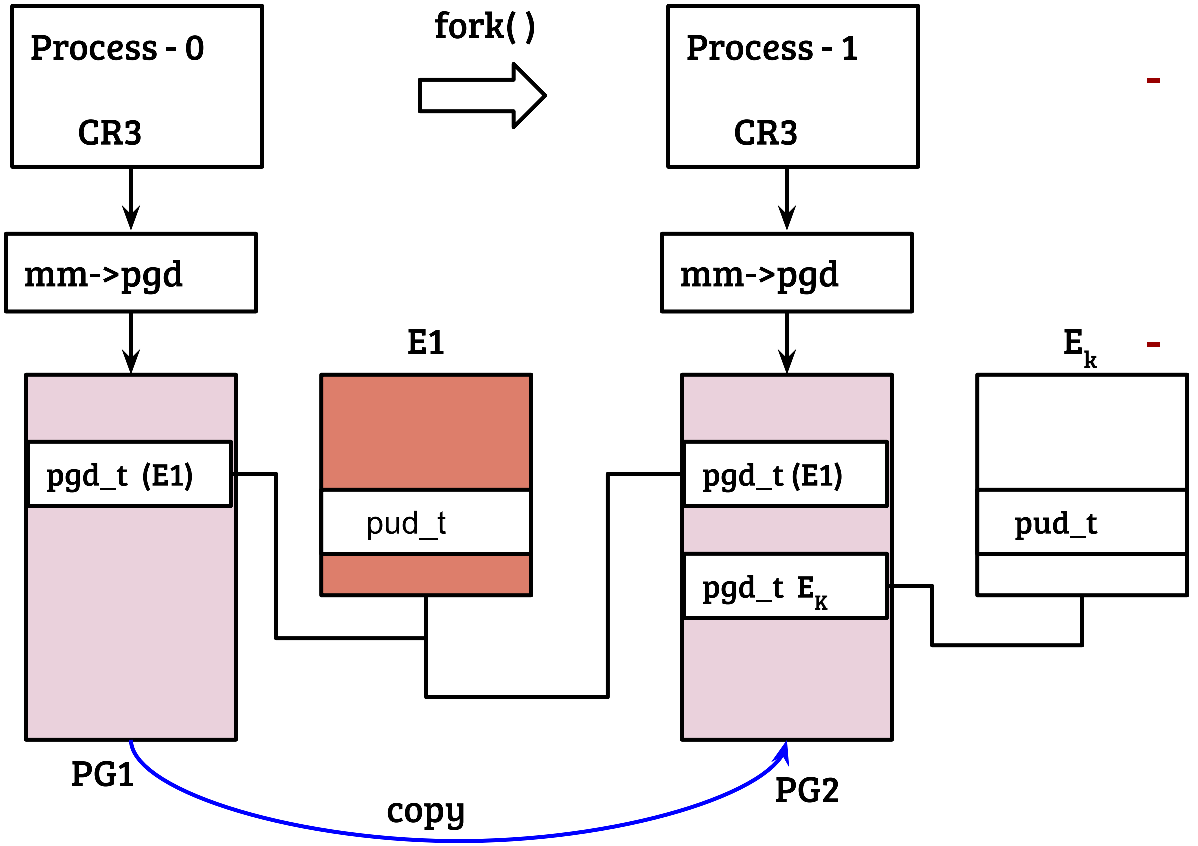
Solution: Every process owns its own **pgd** entries but inherits the kernel **pgd** entries from the parent :-)

Solution overview



- One (or more) entries in PGD-level (level-4) reserved for kernel mapping
- How many?
- Depends on VA-range covered by one entry and the kernel VA size

Solution overview



- All updates to E1 are visible across all the processes
- So we are at peace! Not really.

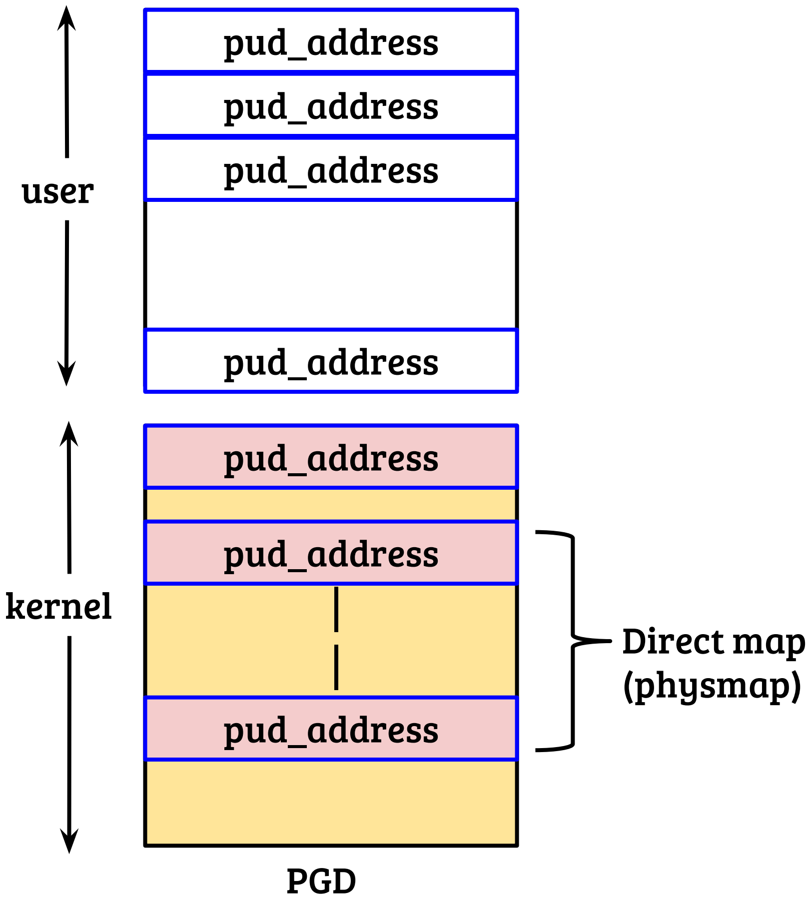
Virtual memory layout (x86_64)

User VA [0x0 - 0x00007fffffffffff] — **2^{47} bytes**

Kernel VA [0xffff800000000000- 0xffffffffffffffff] — **2^{47} bytes**

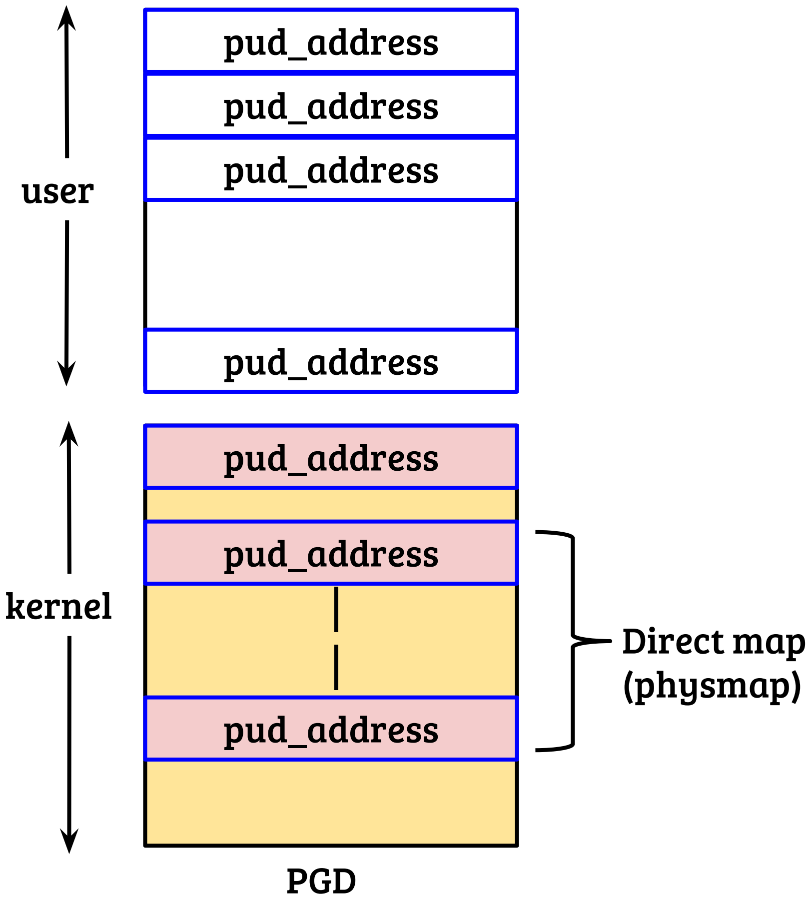
- User virtual addresses use the LSB 47 bits
- Kernel virtual address does not start from 0x8000000000, but from 0xffff800000000000
- Why? Because X86 hardware enforces if 47th bit is one, 48-63 must be set to one

Process address space (user + kernel)



- Virtual address space is split into two parts, user VA and kernel VA
- Kernel mappings are isolated from user through **S/U** bit of page table entry
- Advantages: isolation + efficiency
- What is the need for direct map?

Process address space (user + kernel)



- Virtual address space is split into two parts, user VA and kernel VA
- Kernel mappings are isolated from user through **S/U** bit of page table entry
- Advantages: isolation + efficiency
- What is the need for direct map? Helps in mapping physical address to an already mapped kernel vaddr

Issue with shared address space

```
char array[256 * 4096];        //__alligned(4k);
char secret = *(char *) 0xffff880000000000;
array[secret  << 12] = 0;
```

- This program will result in an exception → Segmentation fault
- Everything seems to be under control. What is the problem then?

Information leakage through out-of-order execution

```
1.  mov RCX, $0xFFFF888000000000;
2.  mov RBX, $array;
3.  mov AL, [RCX];
4.  Shl RAX, $0xC;
5.  mov RBX, qword [RBX + RAX];
```

Executed out-of-order

Exception handler

```
1.  cmp CR2, $userend;
2.  Jg raise_segv;
3.  ............
4.  ........
5.  raise_segv:
6.  ..........
```

- By the time the instruction in line#3 is committed (and a fault is raised), instructions in line#4 and #5 are completed out-of-order

Side-effect: access footprint

1. char array[256 * 4096];   //__alligned(4k);
2. char secret = *(char *) 0xffff8800000000000;
3. **array[secret << 12] = 0;**

Array (before the program execution) : block 0 == {0 - 4095} etc.

0	1	2	3	....	k	....	256

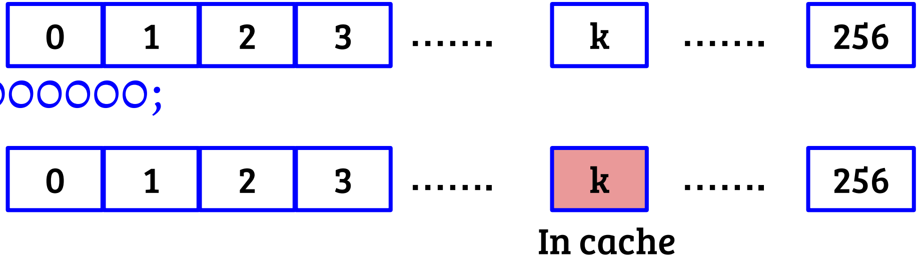
Array (after out-of-order execution of #3) {assume secret = k}

0	1	2	3	....	k	....	256

Accessed

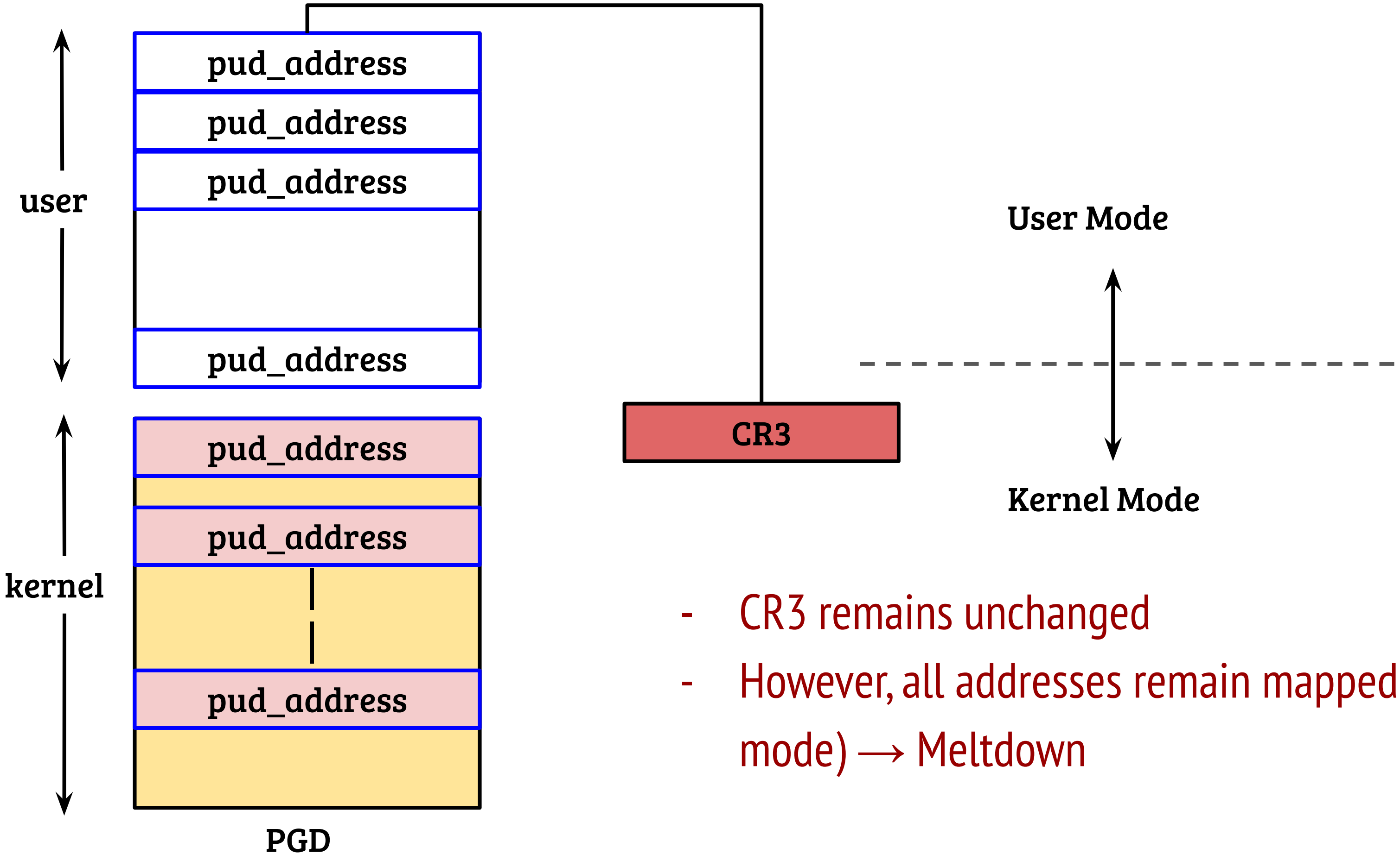
OOO vulnerability + Flush-Reload

```
unsigned time[256];
char array[256 * 4096];
flush_array(array);
char secret = *(char *) 0xffff888000000000;
array[secret  << 12] = 0;
for(i=0; i<256; ++i)
        access_and_time(array, time, i);
secret = find_index_with_min_time( time);
```



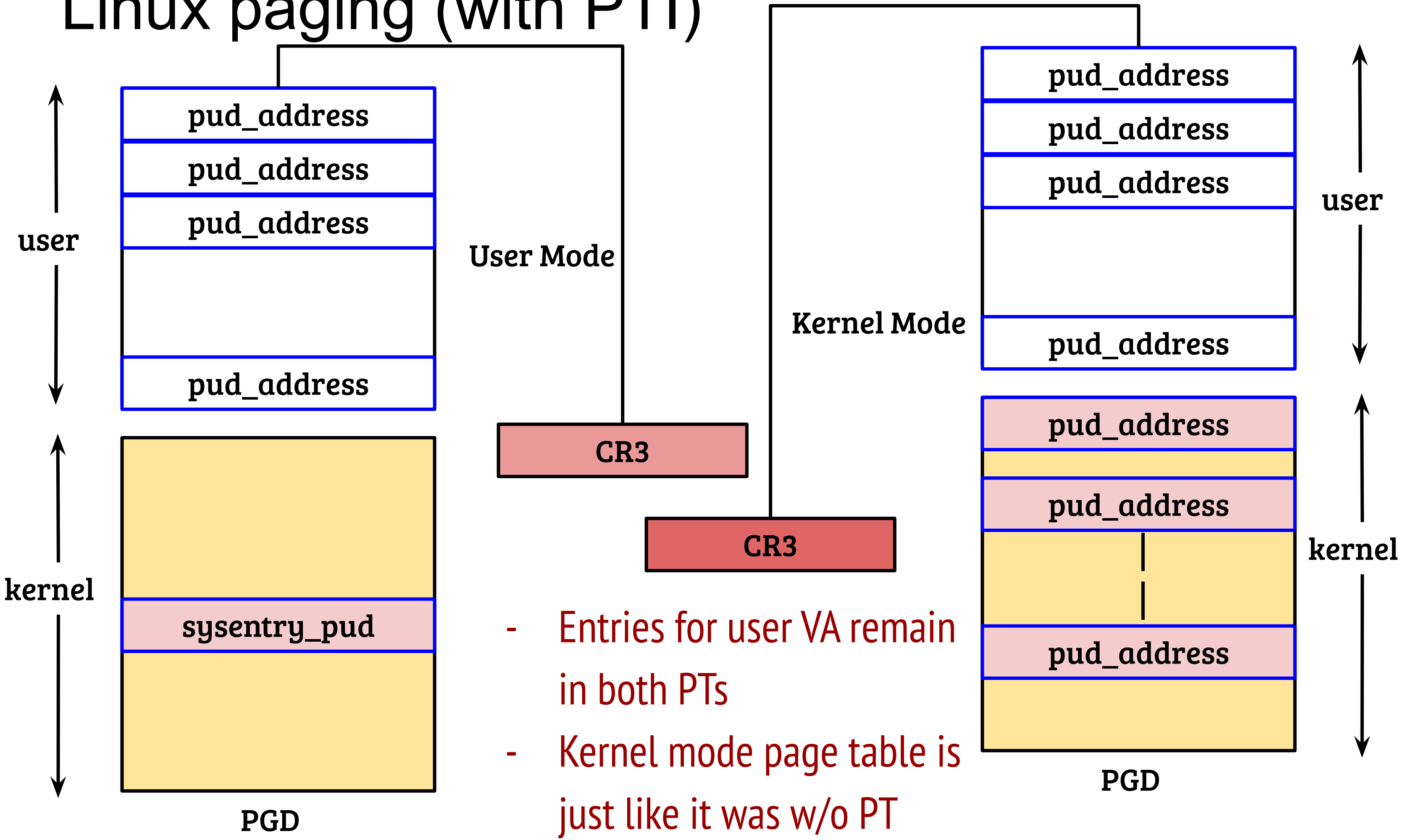
- Result: indirectly read the value of secret
- Meltdown is easy.... Some subtle points still remain
- What is the fix?

Linux paging (before PTI)



- CR3 remains unchanged
- However, all addresses remain mapped (even in user mode) ⟶ Meltdown

Linux paging (with PTI)



- Entries for user VA remain in both PTs
- Kernel mode page table is just like it was w/o PT