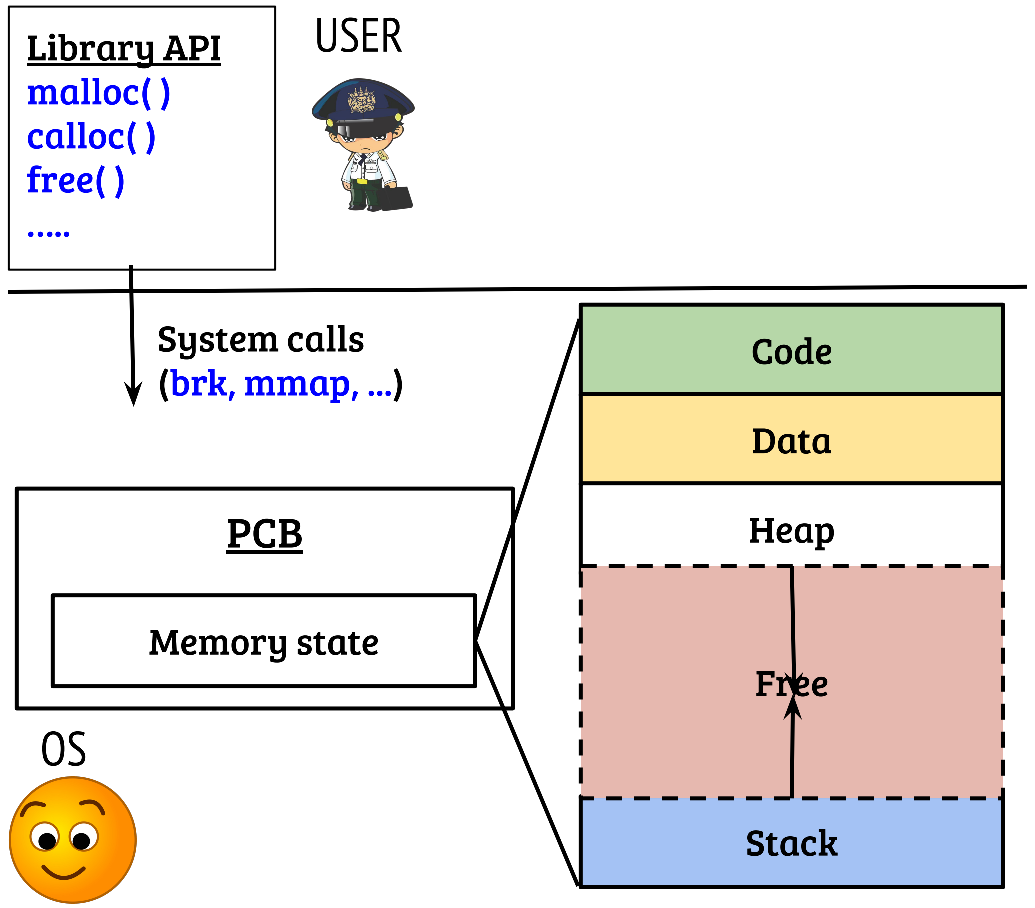
CS614: Linux Kernel Programming

Virtual Memory

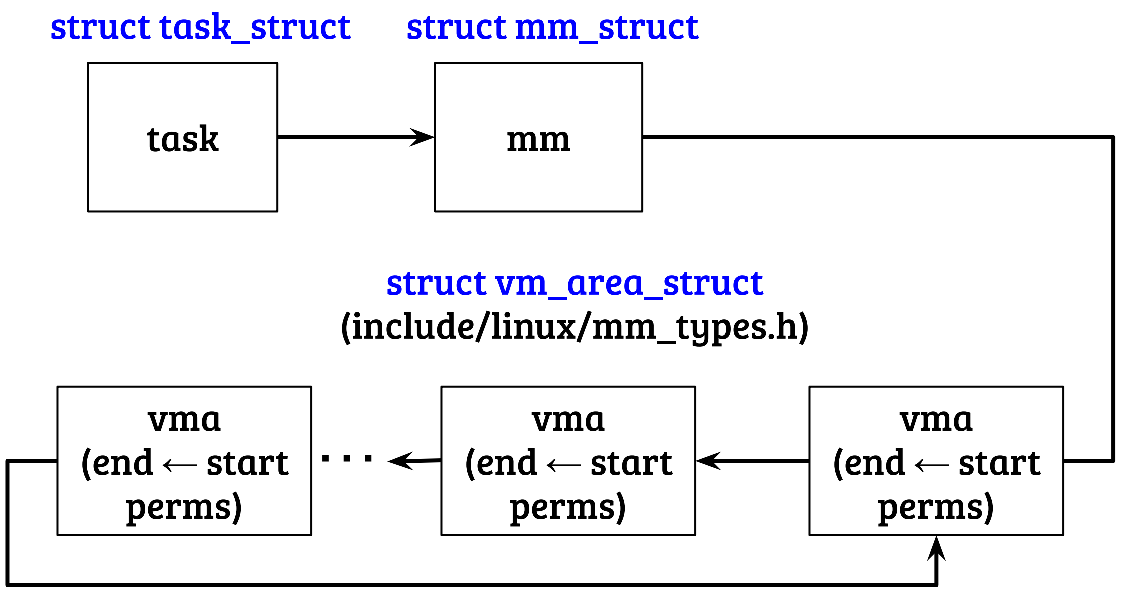
Debadatta Mishra, CSE, IIT Kanpur

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



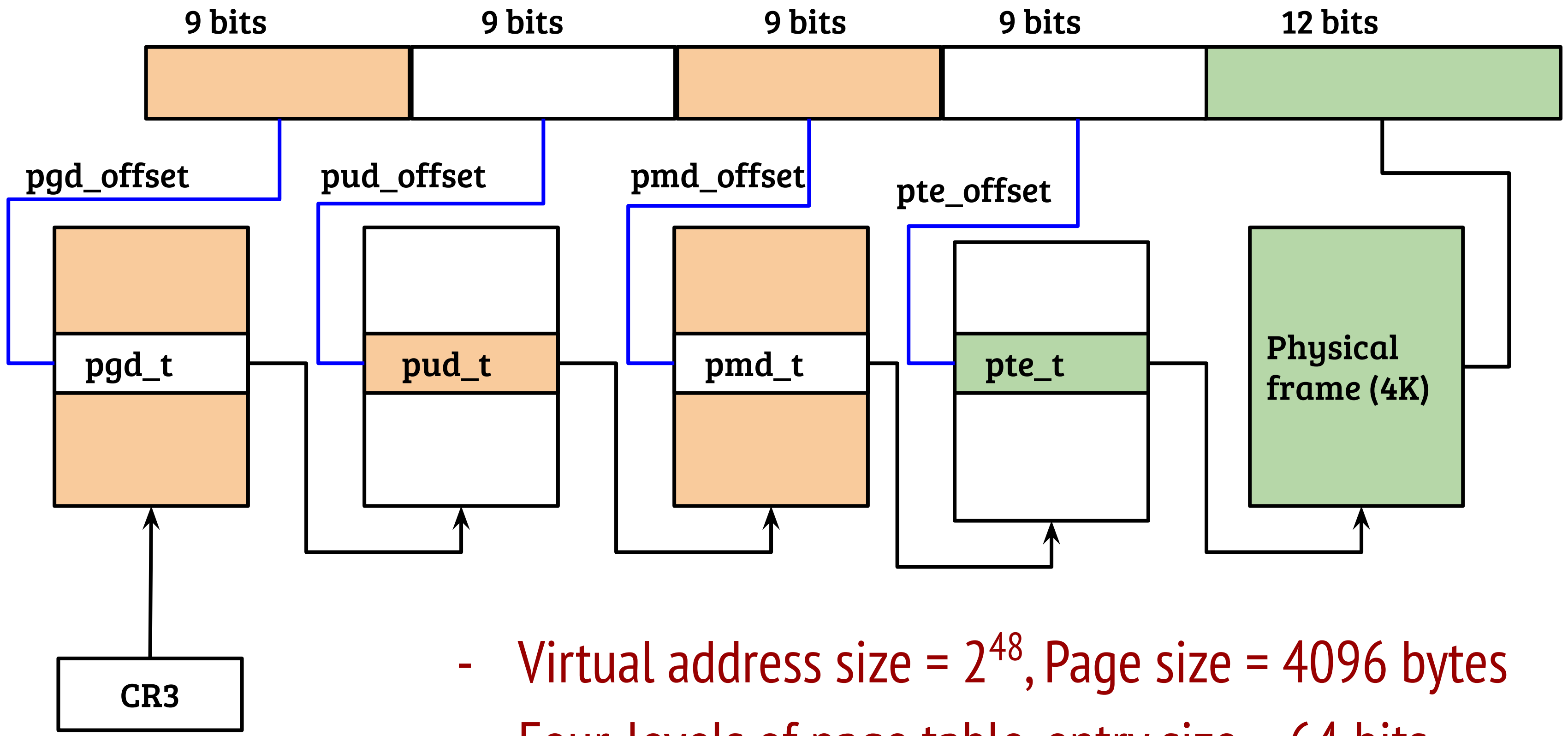
- start and end never overlaps between two vm areas
- can merge/extend vmas if permissions match
- linux maintains both rb_tree and a sorted list (see mm/filemap.c)

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

Address translation: Paging

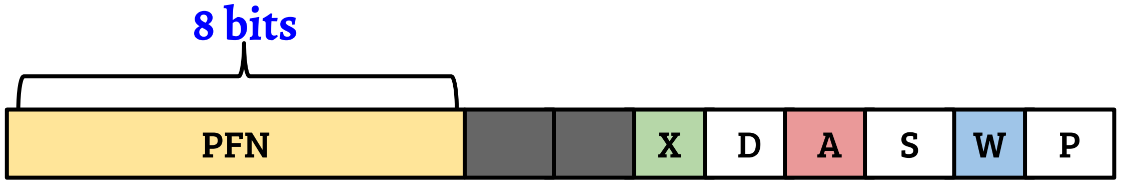
- The idea of paging
  - Partition the address space into fixed sized blocks (call it pages)
  - Physical memory partitioned in a similar way (call it page frames)
  - OS creates a mapping between *page* to *page frame* , H/W uses the mapping to translate VA to PA
- With increased address space size, 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

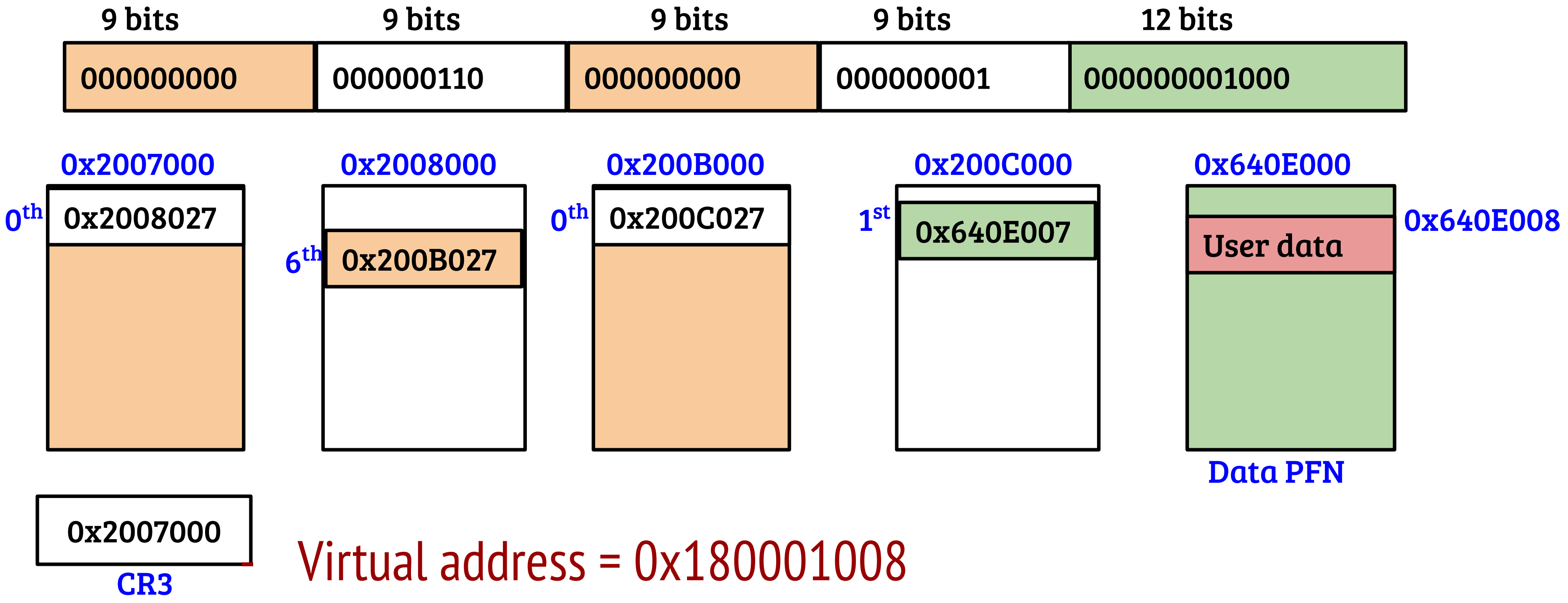
Paging example (structure of an example PTE)



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

P	Present bit, 1 $\Rightarrow$ entry is valid
W	Write bit, 1 $\Rightarrow$ Write allowed
S	Privilege bit, 0 $\Rightarrow$ only kernel mode access is allowed
A	Accessed bit, 1 $\Rightarrow$ Address accessed (set by H/W during walk)
D	Dirty bit, 1 $\Rightarrow$ Address written (set by H/W during walk)
X	Execute bit, 1 $\Rightarrow$ 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

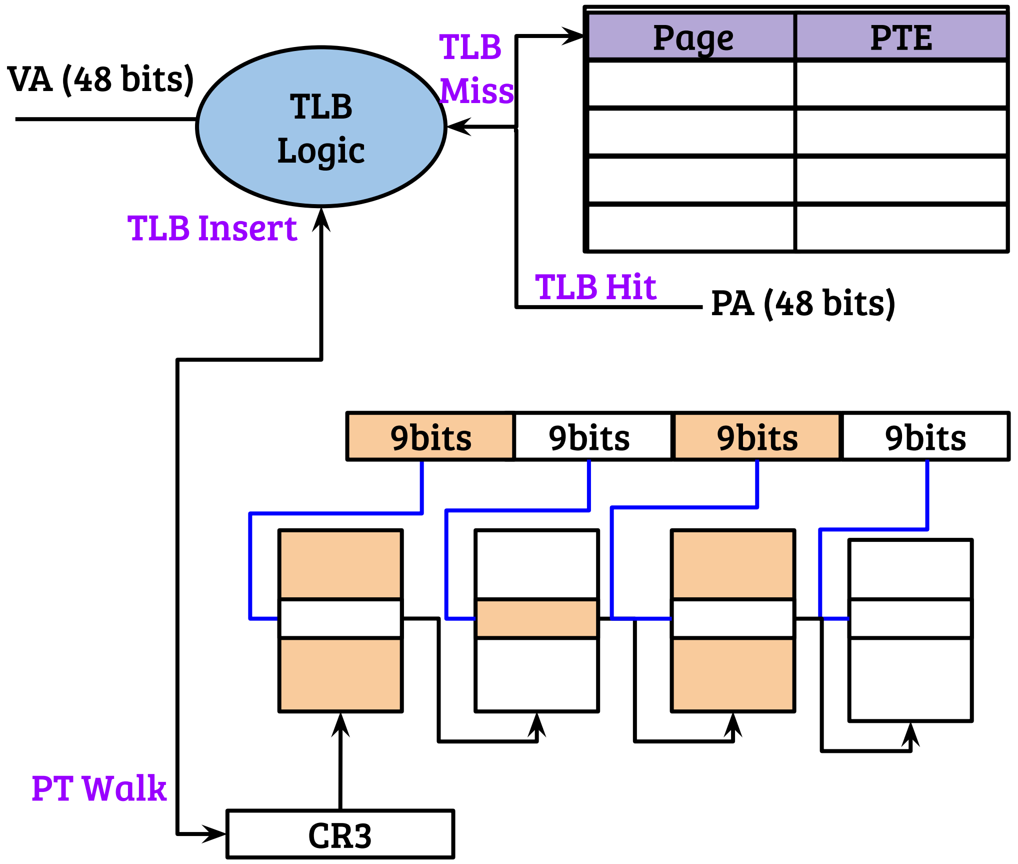
Paging: translation efficiency

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.

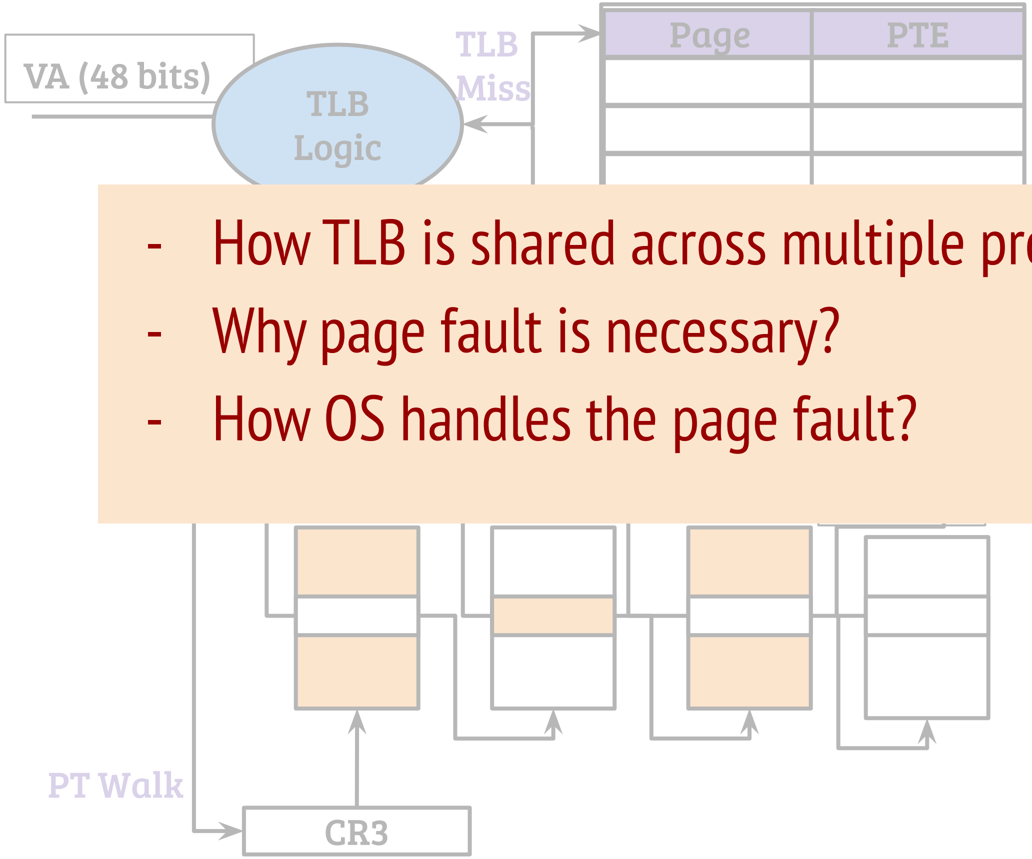
required (for translation) during the execution of the above code?

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

d

into the TLB directly, it can flush entries

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

TLB: Sharing across applications

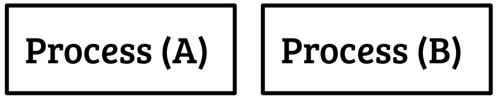
Process (A) **Process (B)**

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



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)

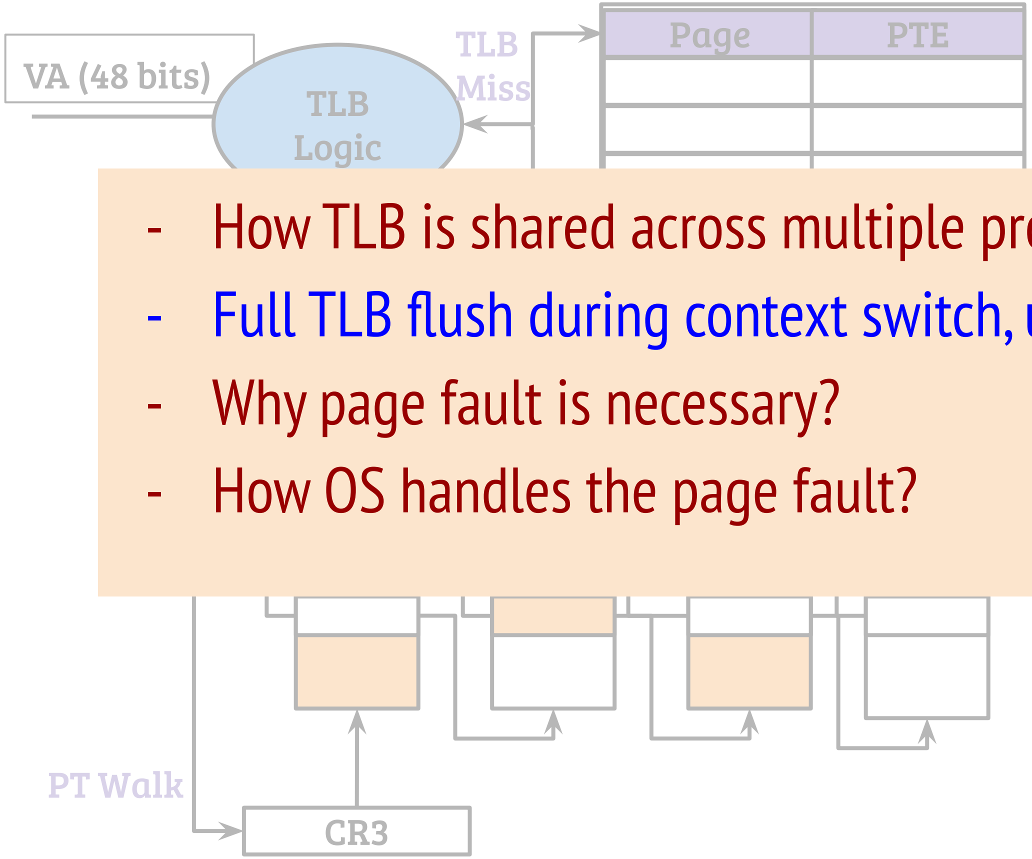
Process (B)

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)



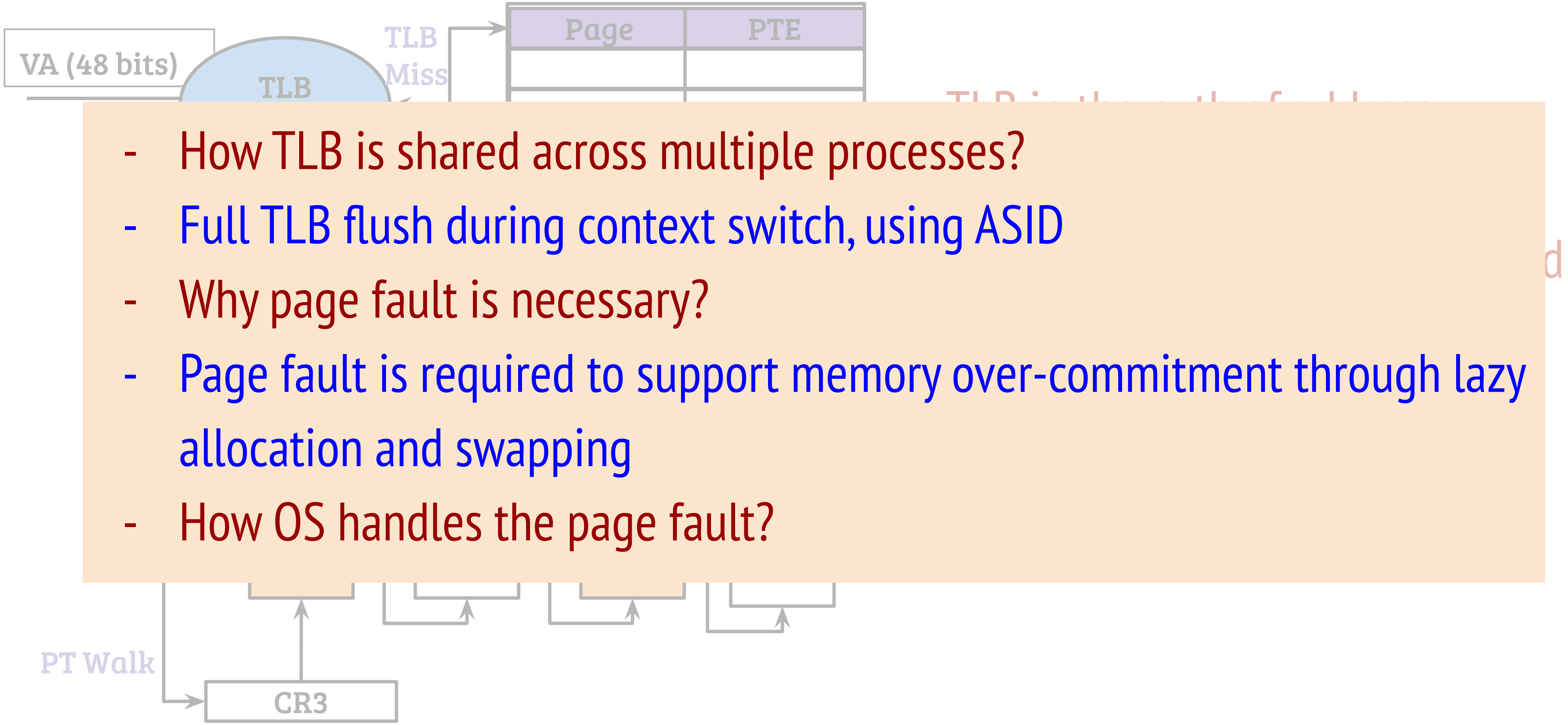
- TLB in the path of address

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

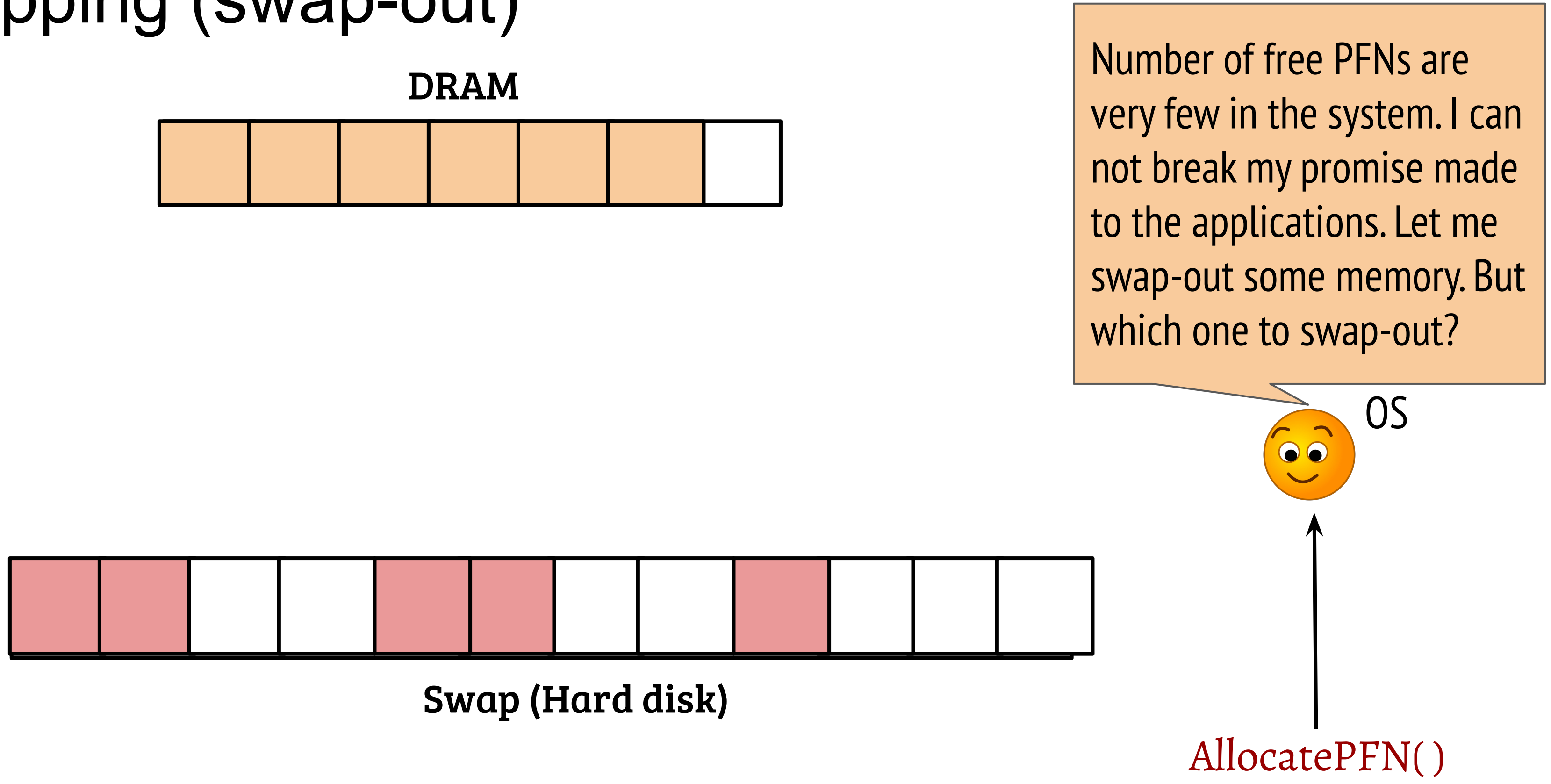
Page fault handling in X86: OS fault handler

```
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);
}
```

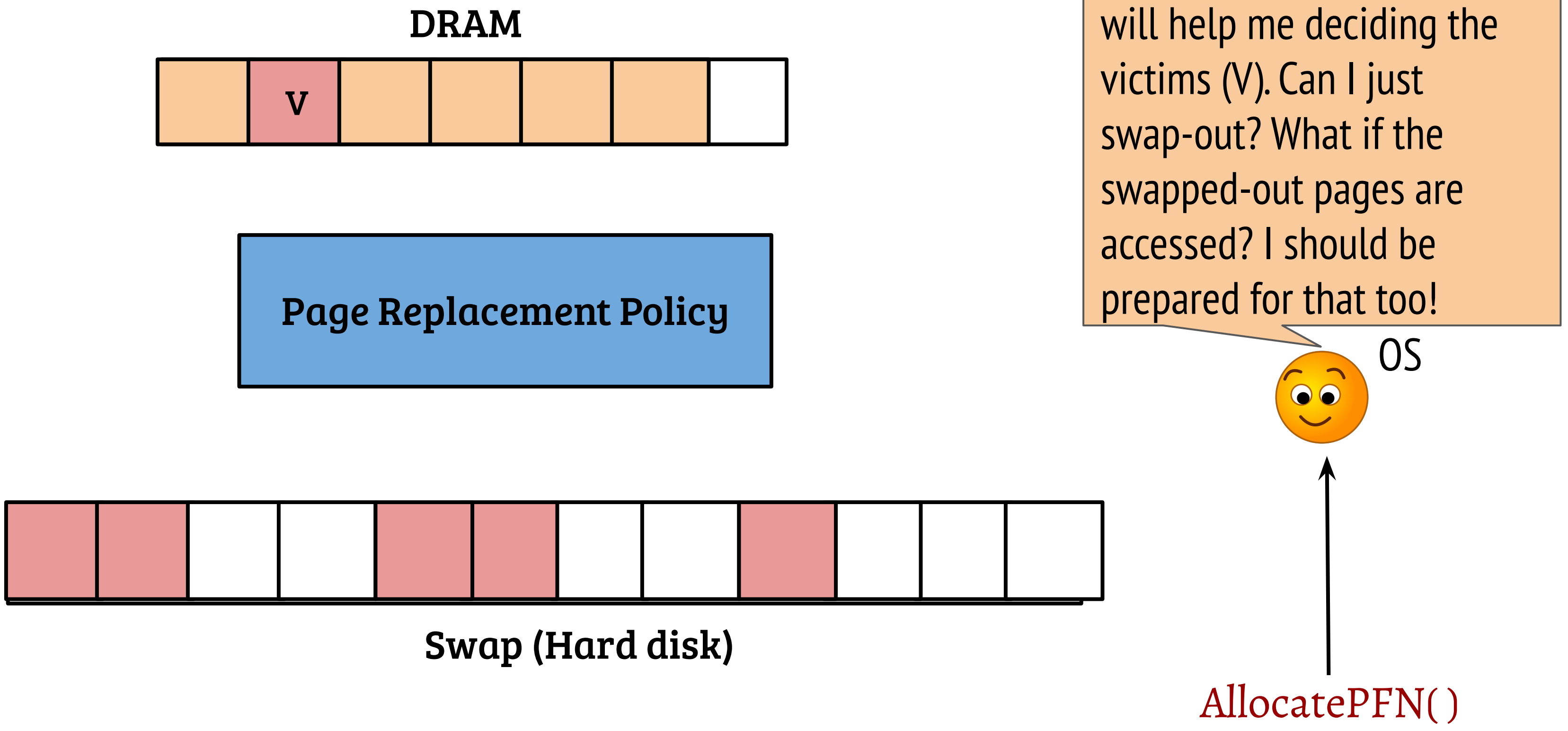
Address translation (TLB + PTW)

- 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.

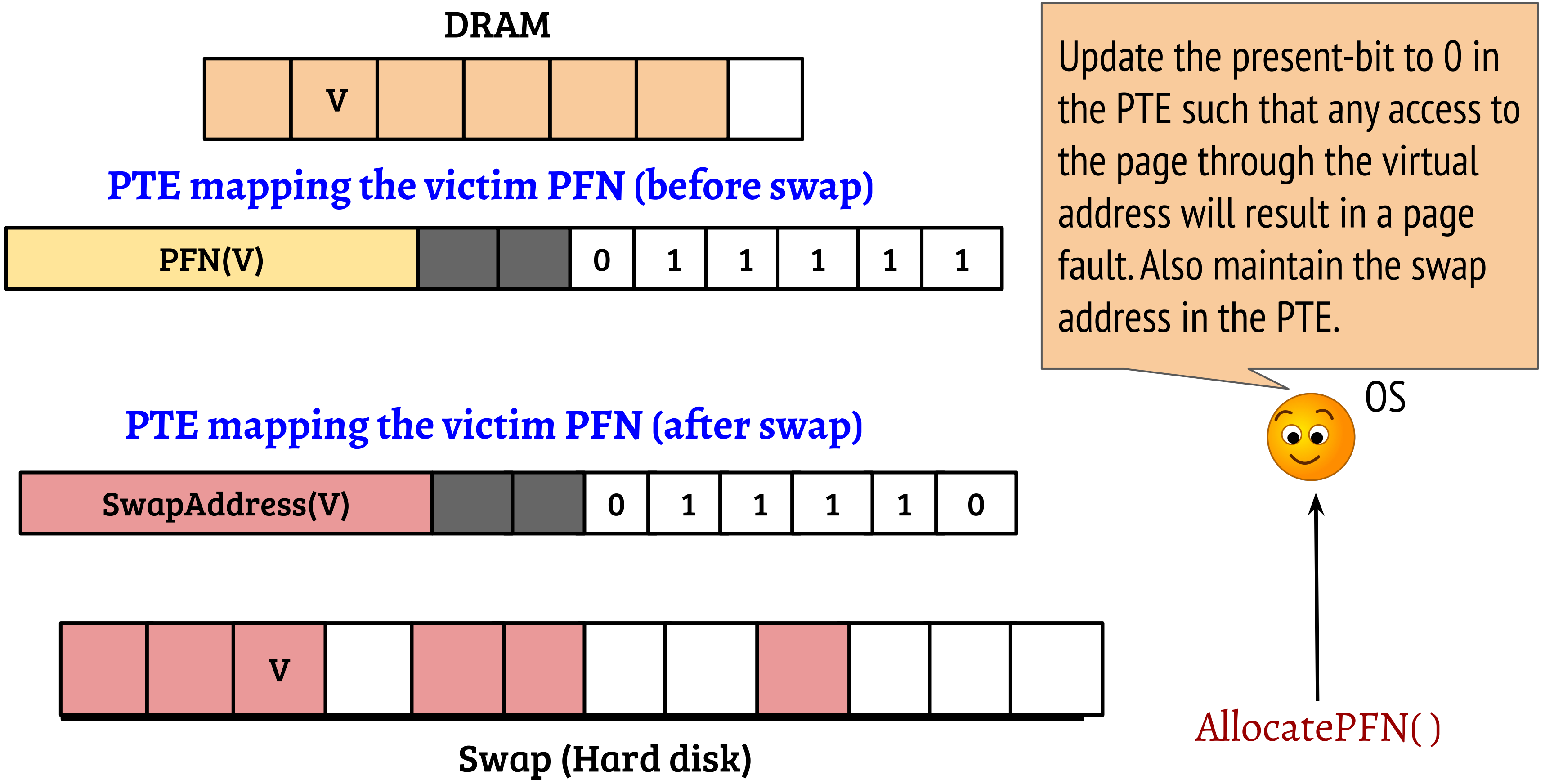
Swapping (swap-out)


DRAM
Swap (Hard disk)
Number of free PFNs are very few in the system. I can not break my promise made to the applications. Let me swap-out some memory. But which one to swap-out?
OS
AllocatePFN( )

Swapping (swap-out)

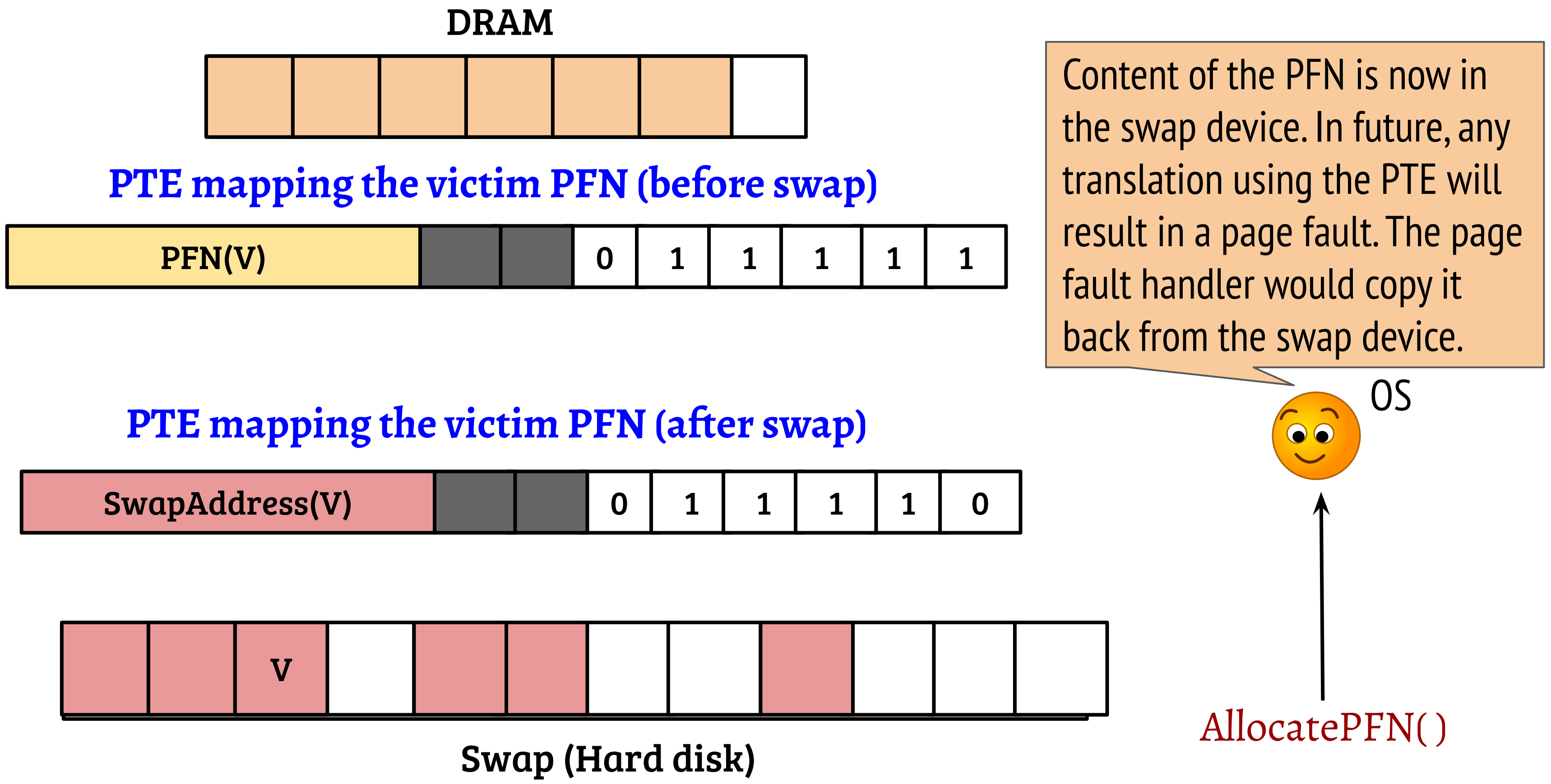

DRAM
V
Page Replacement Policy
Swap (Hard disk)
My page replacement policy will help me deciding the victims (V). Can I just swap-out? What if the swapped-out pages are accessed? I should be prepared for that too!
OS
AllocatePFN( )

Swapping (swap-out)


DRAM
V
PTE mapping the victim PFN (before swap)
PFN(V)
0
1
1
1
1
1
PTE mapping the victim PFN (after swap)
SwapAddress(V)
0
1
1
1
1
0
V
Swap (Hard disk)
OS
AllocatePFN( )

Update the present-bit to 0 in the PTE such that any access to the page through the virtual address will result in a page fault. Also maintain the swap address in the PTE.

Swapping (swap-out)


DRAM
PTE mapping the victim PFN (before swap)
PFN(V)
0 1 1 1 1 1
PTE mapping the victim PFN (after swap)
SwapAddress(V)
0 1 1 1 1 0
V
Swap (Hard disk)
OS
AllocatePFN( )

Content of the PFN is now in the swap device. In future, any translation using the PTE will result in a page fault. The page fault handler would copy it back from the swap device.

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);
}
```