CSCI3150 Introduction to Operating Systems

Lecture 11: Memory Management Part II: Paging

Hong Xu

https://github.com/henryhxu/CSCI3150

Overview

- Paging
- Translation lookaside buffer, TLB
- Advanced page tables

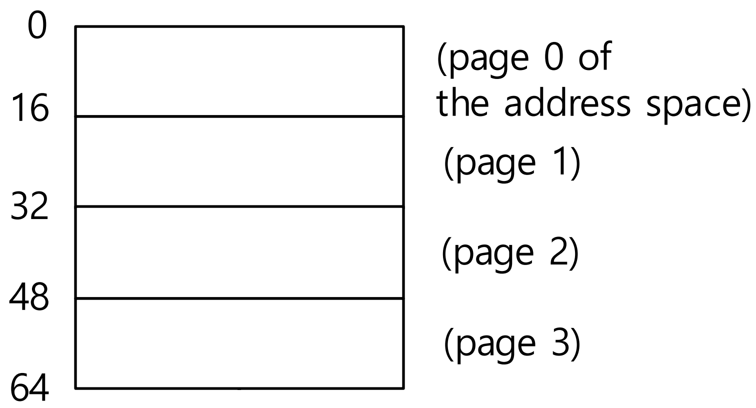
Paging: Introduction

Concept of Paging

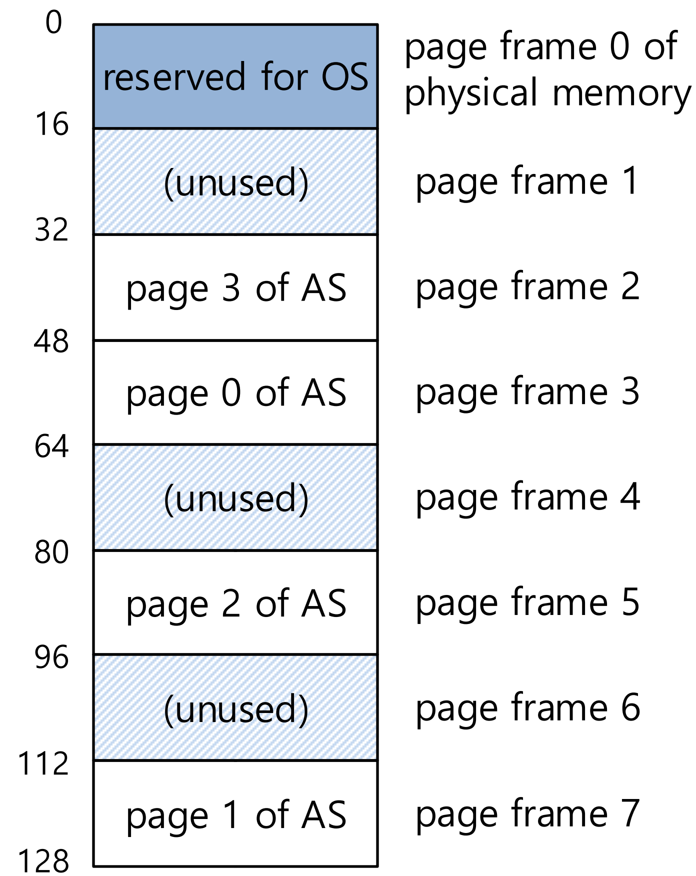
- Segmentation divides an address space into variable sizes of segments
- Paging splits up an address space into fixed-sized units, or **pages**
- With paging, **physical memory** is also split into some number of pages called a **page frame**

Example: A Simple Paging Scheme

- 128-byte physical memory with 16-byte page frames
- 64-byte address space with 16-byte pages



A Simple 64-byte Address Space



64-Byte Address Space Placed In Physical Memory

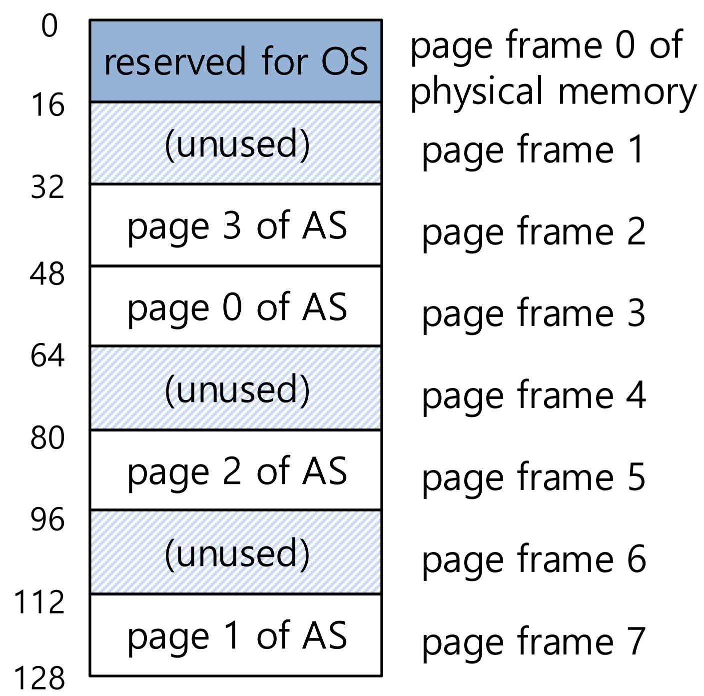
Advantages of Paging

- **Flexibility:** Supporting the abstraction of address space effectively
  - Don't need assumption how heap and stack grow and are used.

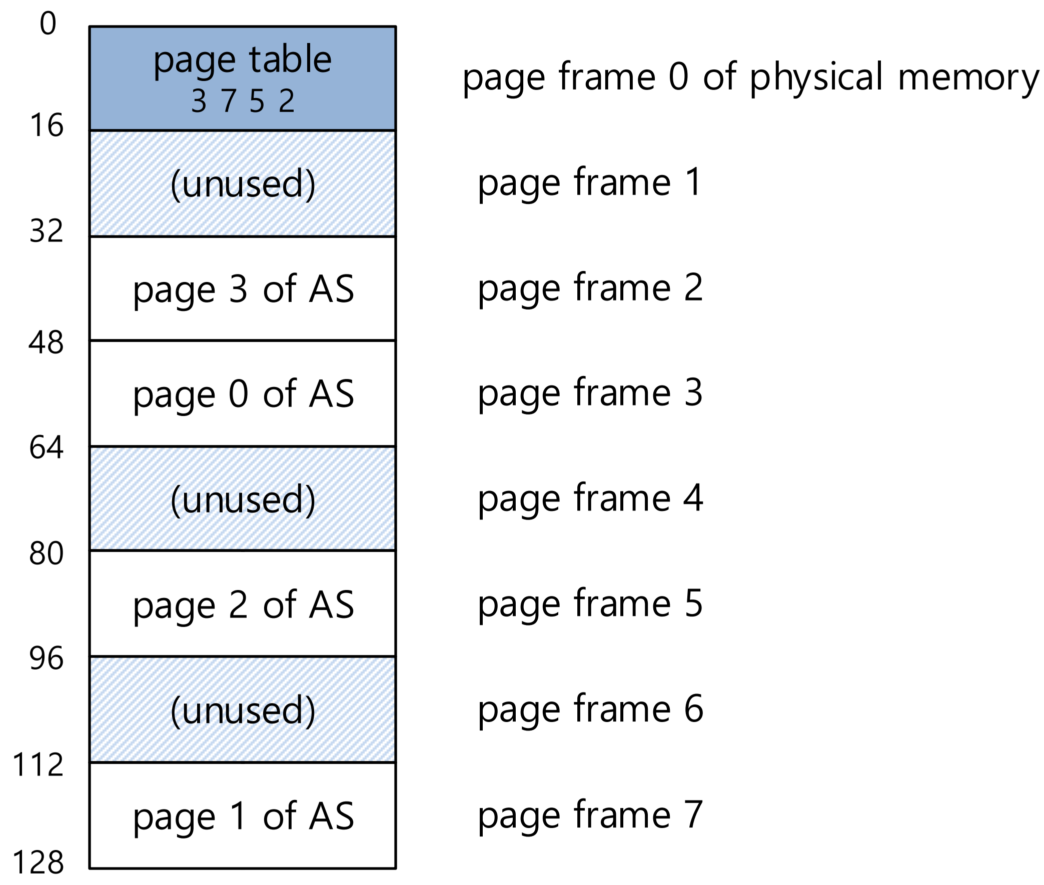
- **Simplicity**: ease of free-space management
  - The page in address space and the page frame are the same size.
  - Easy to allocate and keep a free list

Page table

- A **page table** is needed to translate the virtual address to physical address
  - So we know where each virtual page is in the physical memory!
  - In our example, page 0 is in physical frame 3, page 1 in frame 7, page 2 in frame 5, and page 3 in frame 2
- Page table is per process



Example: Page Table in Kernel Physical Memory

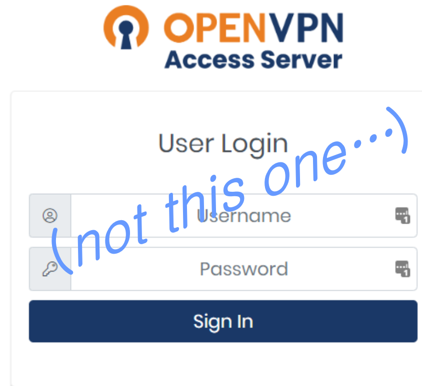


Physical Memory

Address Translation

- Two components in the virtual address
  - VPN: virtual page number
  - Offset: offset within the page

VPN		offset			
Va5	Va4	Va3	Va2	Va1	Va0

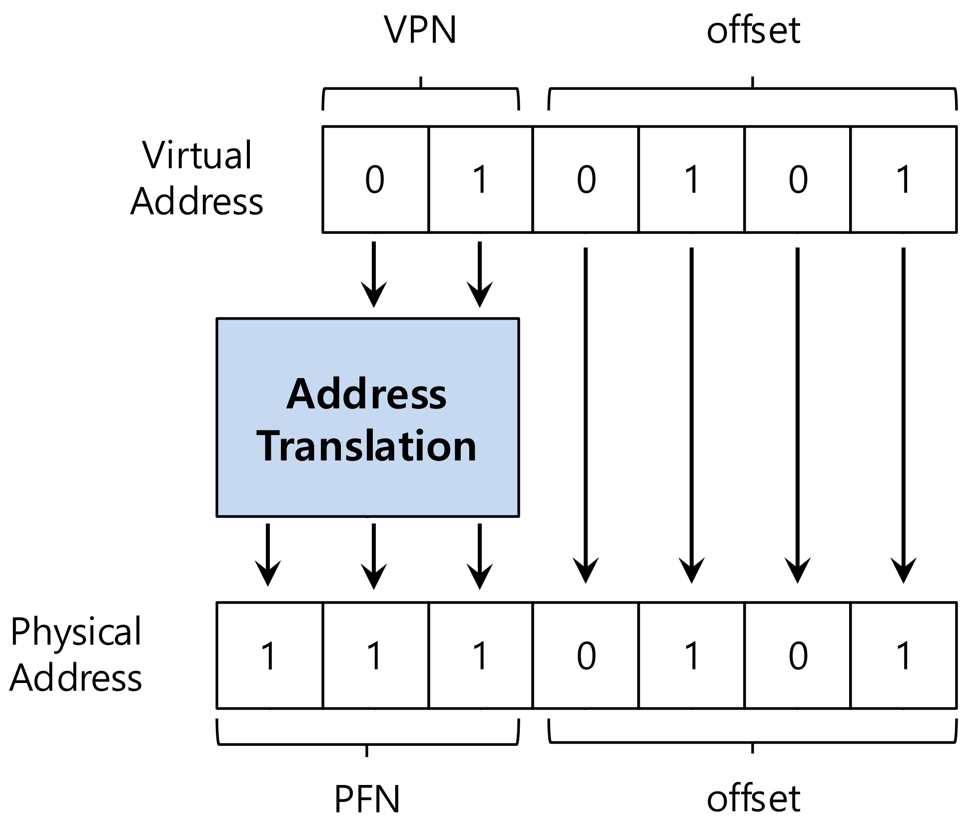


- Example: virtual address 21 in a 64-byte address space

VPN		offset			
0	1	0	1	0	1

Example: Address Translation

- The virtual address 21 in 64-byte address space



Where are Page Tables Stored?

- Page tables can get awfully large!
  - 32-bit address space with 4-KB pages, 20 bits for VPN
    - Page offset for 4 Kbyte pages: 12 bits
    - $2^{20}\ entries, \sim 1M,\ \ 4\ Bytes\ per\ page\ table\ entry$ => 4MB per process!

- Page tables for each process are stored in memory

What is in the Page Table?

- The page table is a **data structure** that is used to map the virtual address to physical address.
  - Simplest form: a linear page table, an array

- The OS **indexes** the array by VPN, and looks up the page-table entry (PTE)

Common Flags of Page Table Entry

- **Valid Bit**: Indicating whether the particular translation is valid (unused space)
- **Protection Bit**: Indicating whether the page could be read from, written to, or executed from
- **Present Bit**: Indicating whether this page is in physical memory or on disk (swapped out)
- **Dirty Bit**: Indicating whether the page has been modified since it was brought into memory
- **Reference Bit (Accessed Bit):** Indicating that a page has been accessed

Example: x86 Page Table Entry

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="20">PFN</td><td colspan="3"></td><td>G</td><td>PAT</td><td>D</td><td>A</td><td>PCD</td><td>PWT</td><td>U/S</td><td>R/W</td><td>P</td></tr>
</table>

An x86 Page Table Entry(PTE)

- P: present
- R/W: read/write bit
- U/S: supervisor
- A: accessed bit
- D: dirty bit
- PFN: the page frame number

Paging: Also Too Slow

- To find a location of the desired PTE, the **starting location** of the page table is **needed**.

- For every memory reference, paging requires the OS to perform one **extra** memory reference

Accessing Memory With Paging

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

```

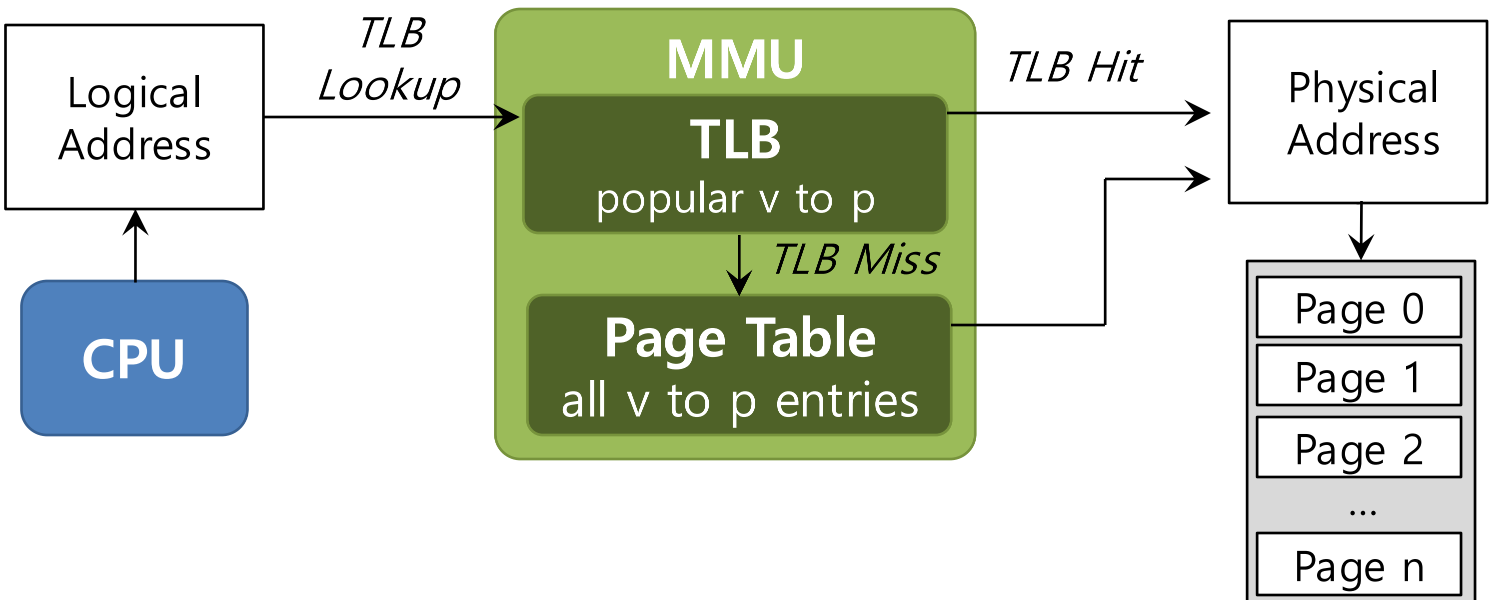
Accessing Memory With Paging

```
// Check if process can access the page
if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
else
        // Access is OK: form physical address and fetch it
        offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
        Register = AccessMemory(PhysAddr)
```

Translation Lookaside Buffer (TLB)

TLB

- Part of the chip's memory-management unit (MMU).
- A hardware cache of **popular** virtual-to-physical address translation.



Address Translation with MMU

Physical Memory

TLB Basic Algorithms

```
VPN = (VirtualAddress & VPN_MASK ) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
    if(Success == TRUE){ // TLB Hit
        if(CanAccess(TlbEntry.ProtectBit) == True ){
            offset = VirtualAddress & OFFSET_MASK
            PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
            AccessMemory(PhysAddr)
        }else RaiseException(PROTECTION_ERROR)
```

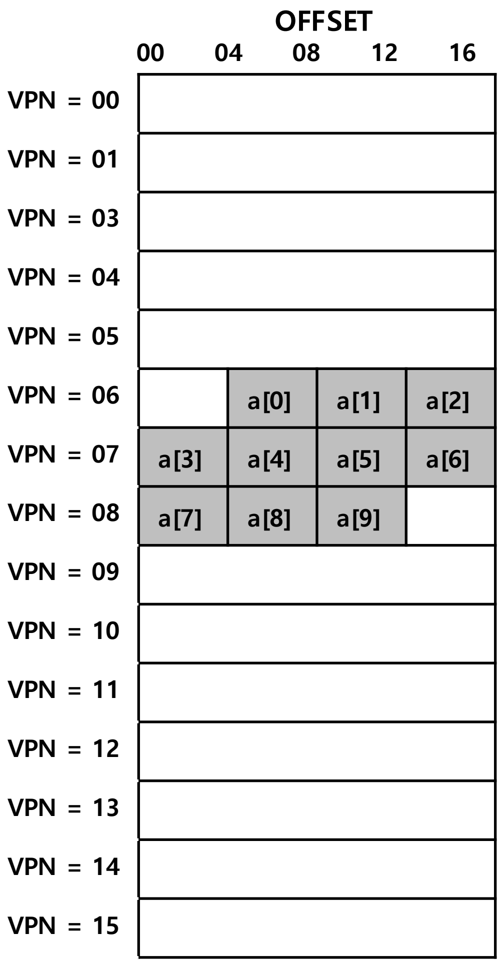
TLB Basic Algorithms (Cont.)

```
11:     }else{ //TLB Miss
12:         PTEAddr = PTBR + (VPN * sizeof(PTE))
13:         PTE = AccessMemory(PTEAddr)
14:         if(PTE.Valid == False)
              RaiseException SEGFAULT) ;
15:         else{
16:             TLB_Insert( VPN , PTE.PFN , PTE.ProtectBits)
17:             RetryInstruction()
18:         }
19:     }
```

- (11-12 lines) The hardware accesses the page table to find the translation.
- (16 lines) updates the TLB with the translation.

Example: Accessing An Array

- How a TLB can improve performance




```
int sum = 0 ;
for(i=0; i<10; i++){
        sum += a[i];
}
```

First access to a page is a miss, but the subsequent ones are hits

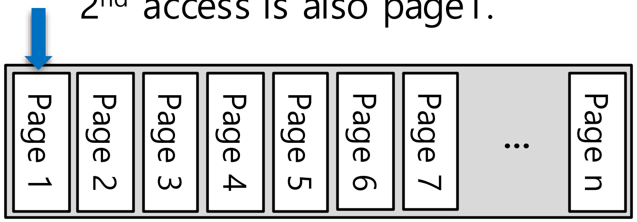
3 misses and 7 hits.
Thus TLB hit rate is 70%.

The TLB improves performance due to spatial locality

Locality

- Temporal Locality
  - An instruction or data item that has been recently accessed will likely be re-accessed soon again

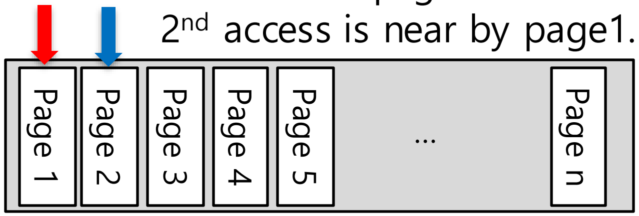
1st access is page1.
2nd access is also page1.



Virtual Memory

- Spatial Locality
  - If a program accesses memory at address `x`, it will likely soon access memory near `x`.

1st access is page1.
2nd access is near by page1.



Virtual Memory

Who Handles a TLB Miss?

- Hardware handles the TLB misses entirely on CISC
  - The hardware has to know exactly where the page tables are located in memory.
  - The hardware would "walk" the page table, find the correct page-table entry and extract the desired translation, update and retry instruction.
  - **hardware-managed TLB.**
  - **Intel x86**
- RISC has what is known as a **software-managed TLB**
  - On a TLB miss, the hardware raises an exception (trap handler)
    - **Trap handler is code** within the OS that is written with the express purpose of handling TLB miss.

TLB Control Flow algorithm (OS Handled)

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB Hit
if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
else
        RaiseException(PROTECTION_FAULT)
else // TLB Miss
        RaiseException(TLB_MISS)
```

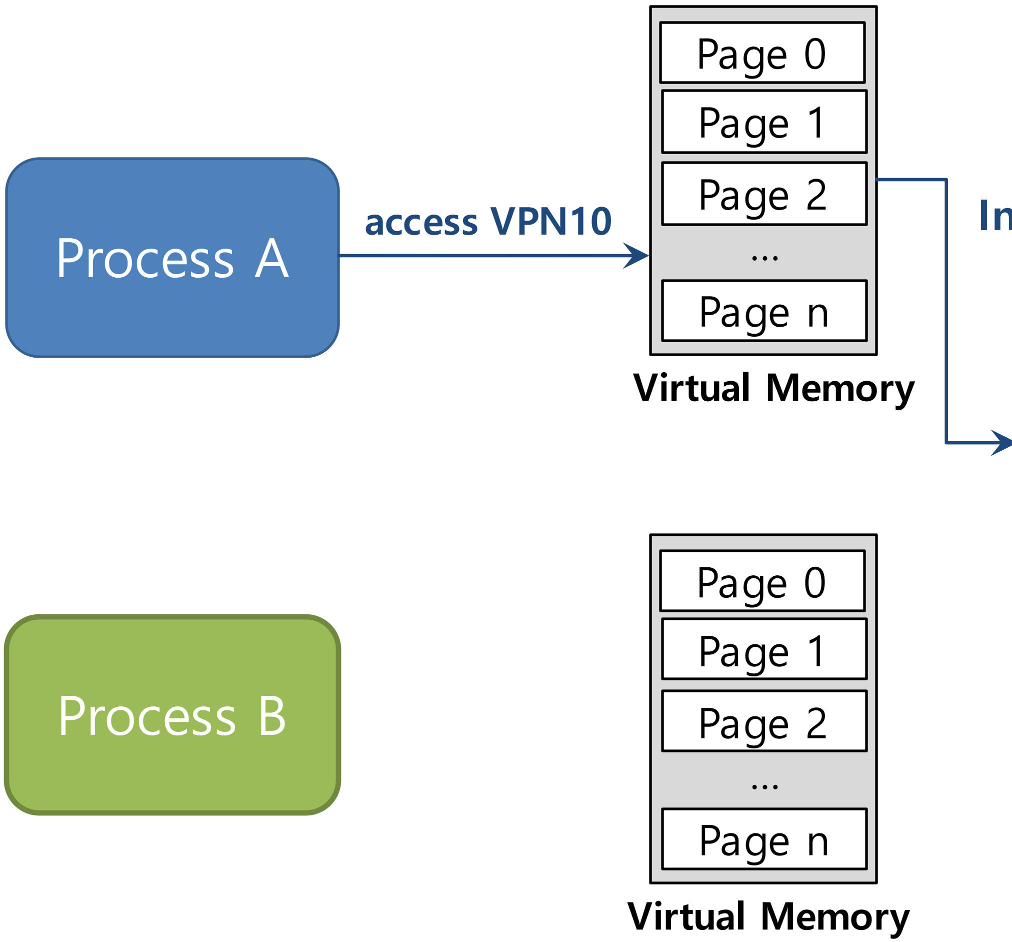
TLB entry

- TLB is managed by **fully-associative*** method.
  - A typical TLB has 32, 64, or 128 entries.
  - Hardware searches the entire TLB in parallel to find the desired translation.
  - Other bits: valid bits, protection bits, address-space identifier, dirty bit

VPN	PFN	Other bits

*More on cache associativity: http://csillustrated.berkeley.edu/PDFs/handouts/cache-3-associativity-handout.pdf

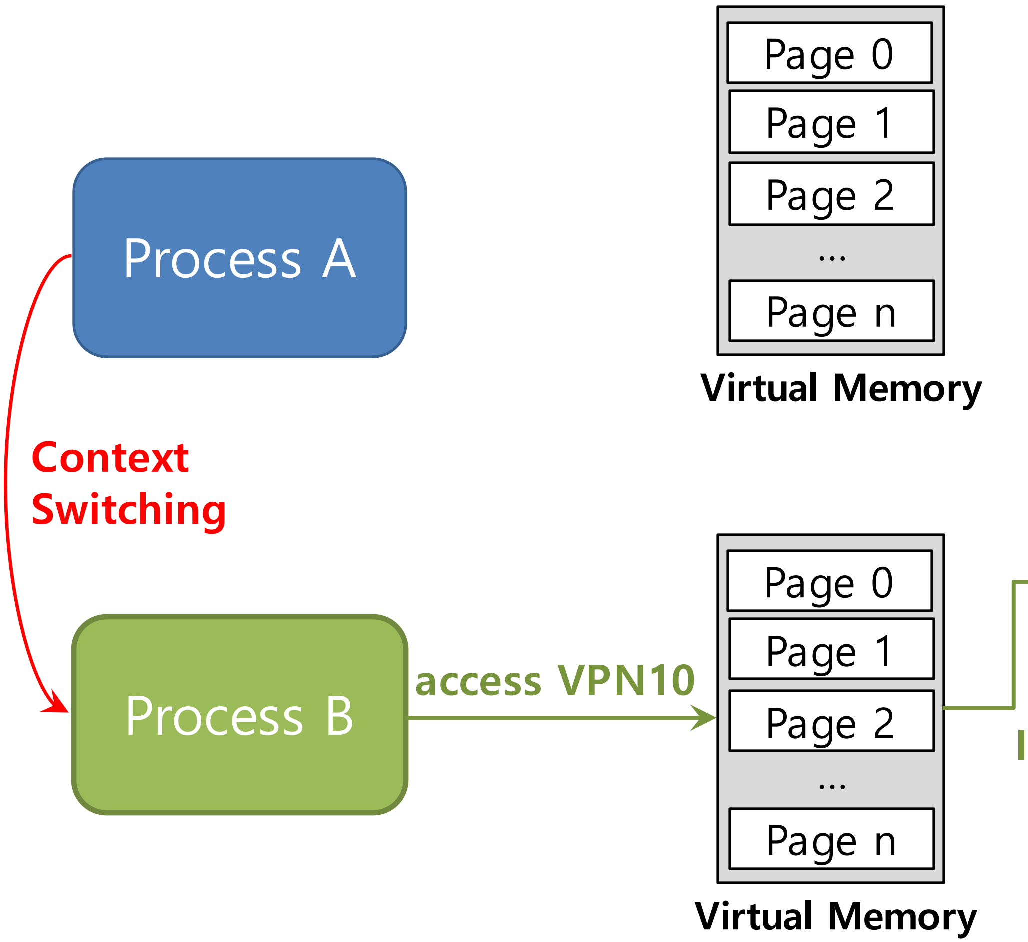
TLB Issue: Context Switching



TLB Table

VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
-	-	-	-
-	-	-	-

TLB Issue: Context Switching



TLB Table

VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

Insert TLB Entry

TLB Issue: Context Switching

Process A

Page 0
Page 1
Page 2
…
Page n

Virtual Memory

Process B

Page 0
Page 1
Page 2
…
Page n

Virtual Memory

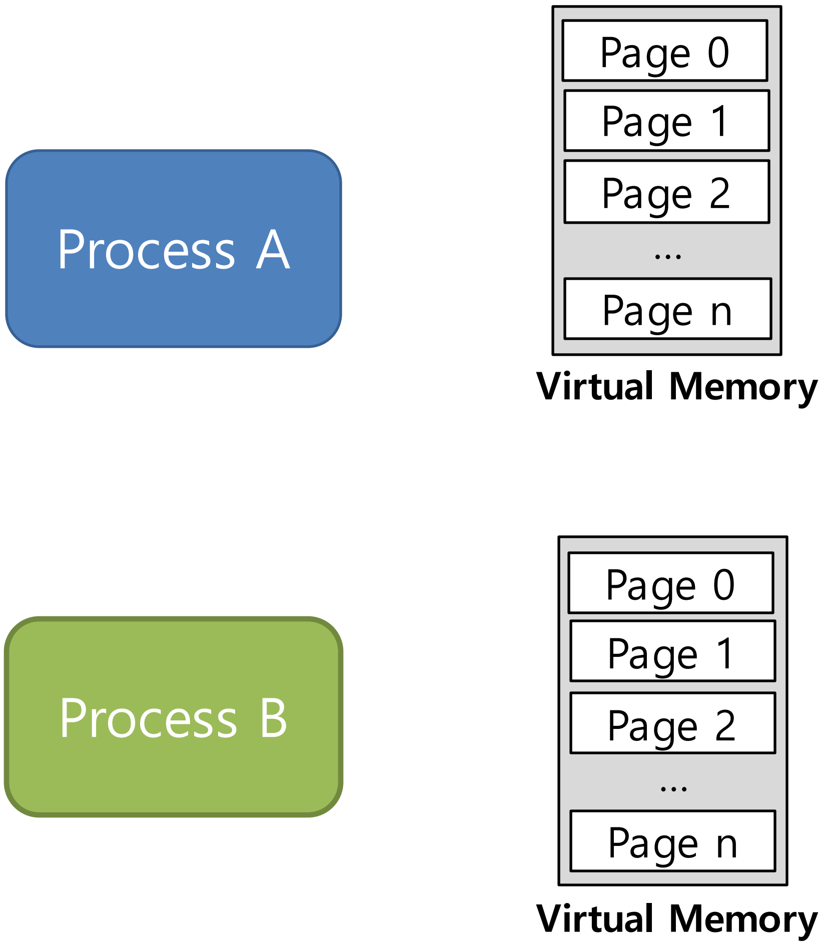
TLB Table

VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

Can't Distinguish which entry is meant for which process

Solution

- Add an address space identifier (ASID) field in the TLB



TLB Table

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
-	-	-	-	-
10	170	1	rwx	2
-	-	-	-	-

Another Case

- Two processes share a page
  - Process 1 is sharing physical frame 101 with Process 2.
  - P1 maps this frame into the 10th page of its address space.
  - P2 maps this frame to the 50th page of its address space.

VPN	PFN	valid	prot	ASID
10	101	1	rwx	1
-	-	-	-	-
50	101	1	rwx	2
-	-	-	-	-

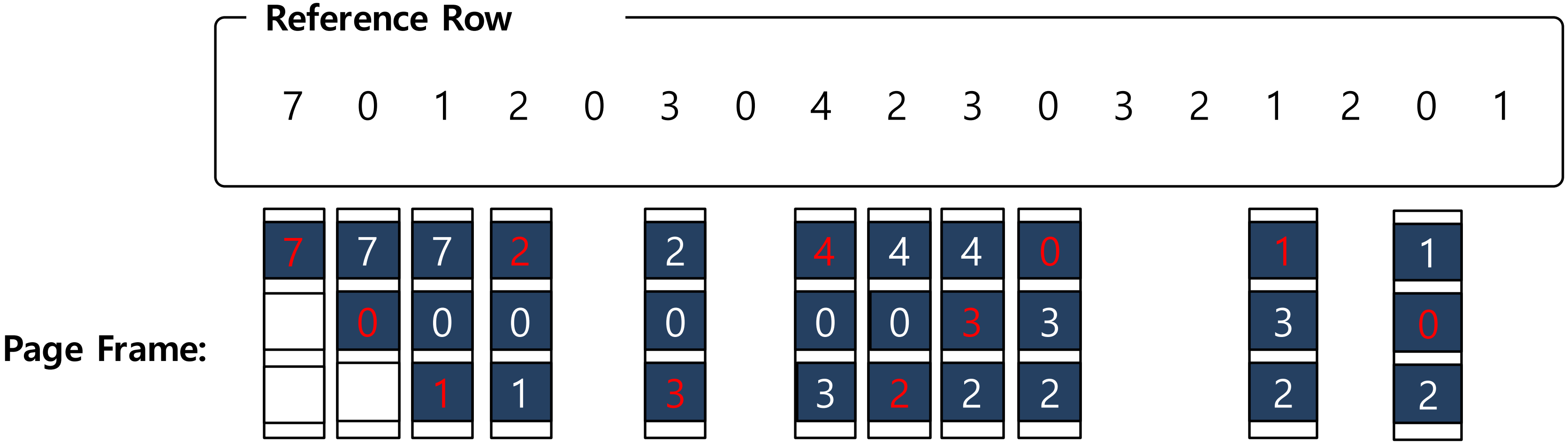
Sharing of pages is useful as it reduces the number of physical pages/frames in use.

On the side: ASID vs. PID

- ASID seems very similar to PID right?
  - To distinguish processes
  - They are related, yes
- Can you just use PID then?
  - Obviously no (otherwise it won't be called ASID…)
  - ASID (hardware) uses much fewer bits than PID (OS)
- How to use ASID when the PID space is much bigger?
  - https://stackoverflow.com/questions/52813239/how-many-bits-there-are-in-a-tlb-asid-tag-for-intel-processors-and-how-to-handl
  - Caveat: ASID is only needed when the process is active (from TLB's perspective)

TLB Replacement Policy

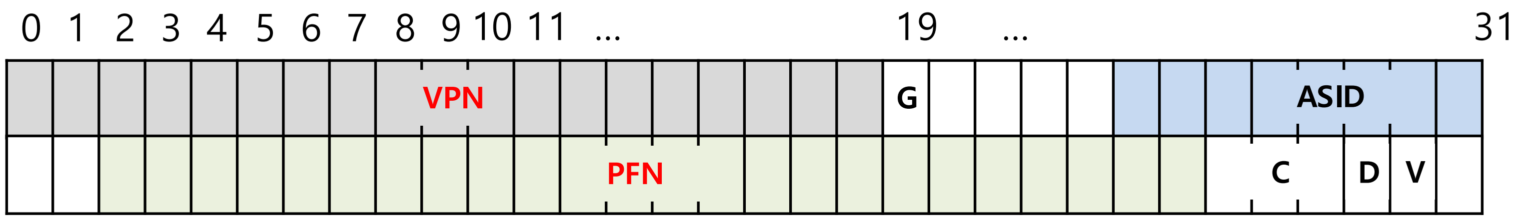
- Least Recently Used, LRU
  - Evict an entry that has not recently been used
  - Take advantage of *locality* in the memory-reference stream.



Total 11 TLB miss

A Real TLB Entry

All 64 bits of this TLB entry (example of MIPS R4000)

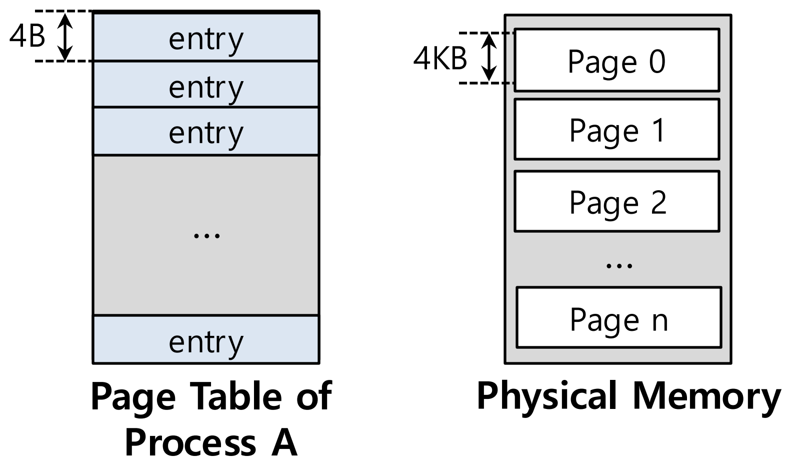


Flag	Content
19-bit VPN	The rest reserved for the kernel.
24-bit PFN	Systems can support with up to 64GB of main memory(pages).
Global bit(G)	Used for pages that are globally-shared among processes.
ASID	OS can use to distinguish between address spaces.
Coherence bit(C)	determine how a page is cached by the hardware.
Dirty bit(D)	marking when the page has been written.
Valid bit(V)	tells the hardware if there is a valid translation present in the entry.

Advanced Page Tables

Paging: Linear Tables

- We usually have one page table for every process in the system.
  - Assume 32-bit address space with 4KB pages and 4-byte page-table entry.

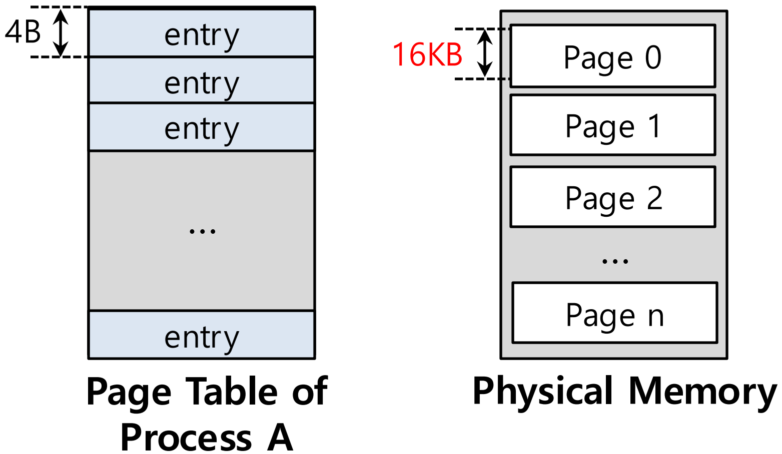


Page table size $= \frac{2^{32}}{2^{12}} * 4Byte = 4MByte$

Page table are too big and thus consume too much memory.

Paging: Smaller Tables

- Page tables are too big and thus consume too much memory.
  - Assume that 32-bit address space with 16KB pages and 4-byte page-table entry.

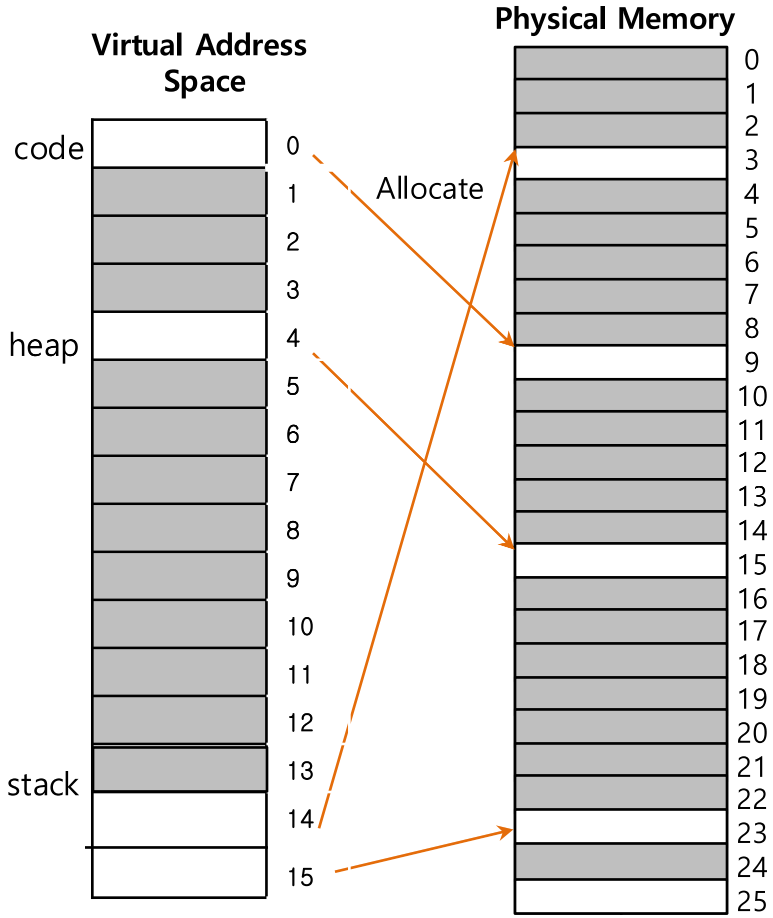


$$\frac{2^{32}}{2^{16}} * 4 = 1MB$$ **per page table**

Big pages lead to internal fragmentation.

Problem

- Single page table for the address space of process.



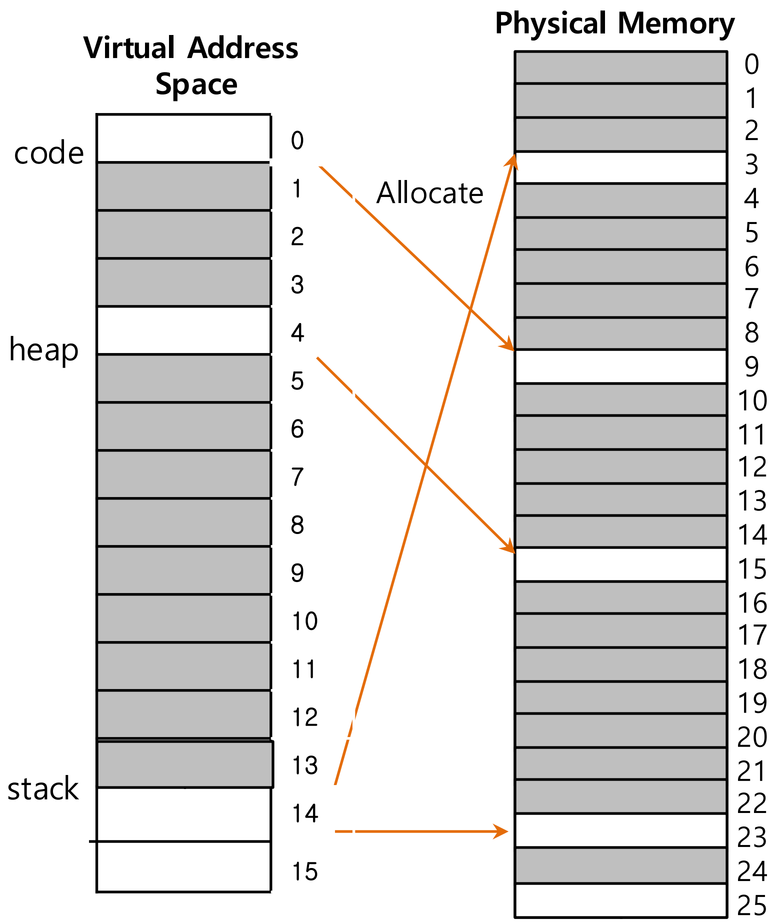
A 16KB Address Space with 1KB Pages

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
…	…	…	…	…
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

A Page Table For 16KB Address Space

Problem

- Most of the page table is **unused**, full of invalid entries.



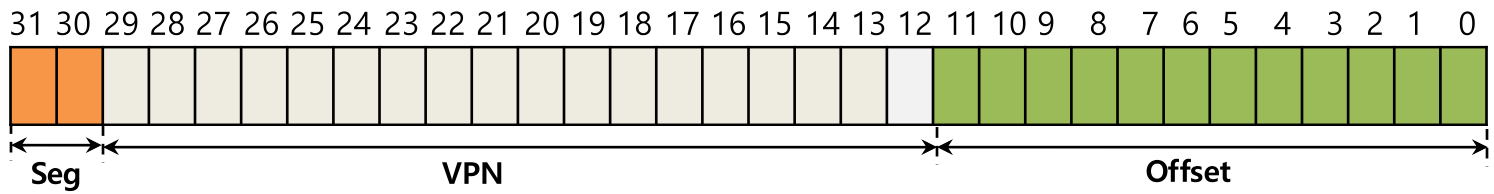
A 16KB Address Space with 1KB Pages

PFN	valid	prot	present	dirty
9	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
…	…	…	…	…
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

A Page Table For 16KB Address Space

Hybrid Approach: Paging and Segments

- Page table for each segment
  - The base register for each of these segments contains the physical address of a linear page table for that segment.
  - The bound register: indicate the end of the page table.
- Example: Each process has three page tables associated with it.



32-bit virtual address space with 4KB pages

Seg value	Content
00	unused segment
01	code
10	heap
11	stack

TLB miss on Hybrid Approach

- The hardware gets **physical address** from **page table**.
  - The hardware uses the segment bits (SN) to determine which base-and-bounds pair to use.
  - The hardware then takes the physical address therein and combines it with the VPN as follows to form the address of the page table entry (PTE)

```
01:    SN = (VirtualAddress & SEG_MASK) >> SN_SHIFT
02:    VPN = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
03:    AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

Problem of the Hybrid Approach

- Hybrid approach is not without problems.
  - If we have a large but sparsely-used heap, we can still end up with a lot of page table waste.
  - Causing external fragmentation, again!

Multi-Level Page Tables

- Turn the linear page table into something like a tree.
  - Chop up the page table into page-sized units.
  - If an entire page of page-table entries is invalid, don't allocate that page of the page table at all.
  - To track whether a page of the page table is valid, use a new structure, called page directory.

Multi-level Page Tables

Linear Page Table

PTBR: 201

valid	prot		
1	rx	12	PFN201
1	rx	13	
0	-	-	
1	rw	100	
0	-	-	PFN202
0	-	-	
0	-	-	
0	-	-	
0	-	-	PFN203
0	-	-	
0	-	-	
0	-	-	
0	-	-	PFN204
0	-	-	
1	rw	86	
1	rw	15	

Multi-level Page Table

PDBR: 200

	valid	PFN
PFN200	1	201
	0	-
	0	-
	1	204

The Page Directory

valid	prot	PFN	
1	rx	12	PFN201
1	rx	13	
0	-	-	
1	rw	100	

[Page 1 of PT: Not Allocated]

[Page 2 of PT: Not Allocated]

0	-	-	PFN204
0	-	-	
1	rw	86	
1	rw	15	

Multi-level Page Tables

- Page Directory
  - The page directory contains one entry per page of the page table.
  - It consists of a number of entries, i.e. page directory entries (PDE).
  - A PDE has a valid bit and page frame number (PFN).
- Advantage
  - Only allocates page-table space in proportion to the amount of address space you are using.
  - The OS can grab the next free page when it needs to allocate or grow a page table.
- Disadvantage
  - Multi-level table is a small example of a time-space trade-off.
  - Complexity.

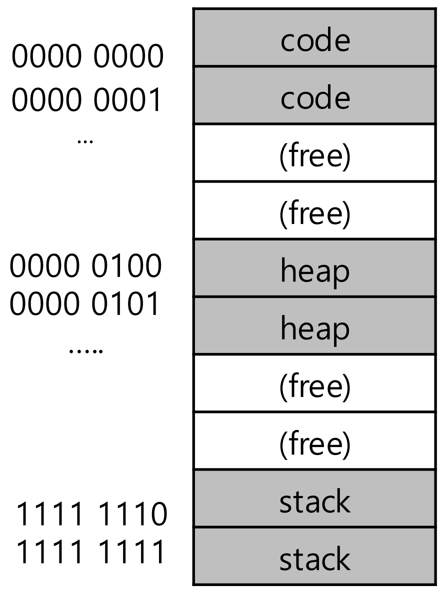
On-the-side: Indirection

- Does this remind you of something familiar?
- Adding another table for a table => adding another level of indirection

All problems in computer science can be solved by another level of indirection.

- David Wheeler

Example

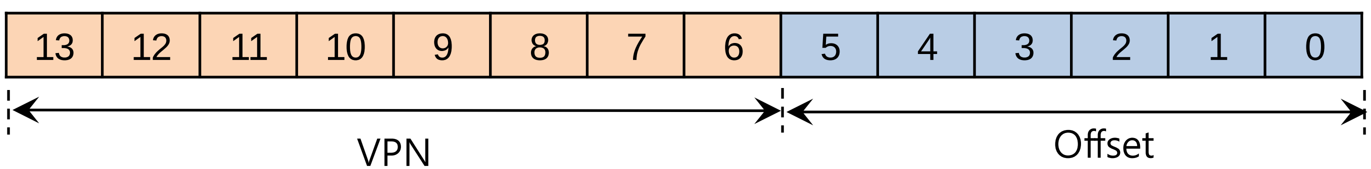


- Page 0,1: code
- Page 4,5: heap
- Page 254, 255: stack

Example

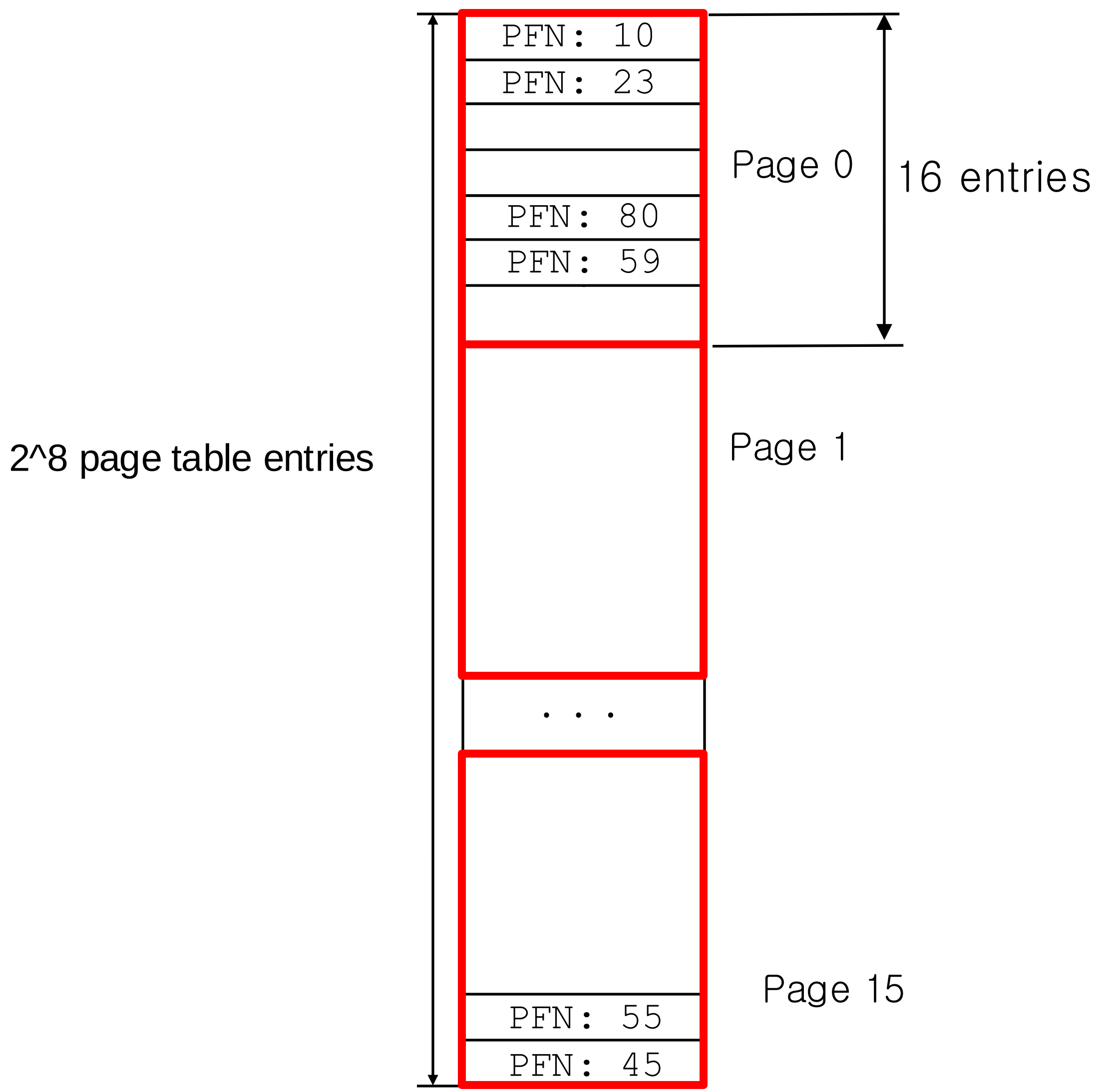
Flag	Detail
Address space	16 KB (2^14 Byte)
Page size	**64 B**
Virtual address	14 bit
VPN	8 bit
Offset	6 bit
Page table entry	4 Byte

A 16-KB Address Space With 64-byte Pages



- Single level paging
  - 256 page table entries: 2^8 entries
  - Page table size: 256 * 4 Byte = 1 Kbyte
  - Page table needs 16 pages of physical memory (64B each): 1024/64 = 16

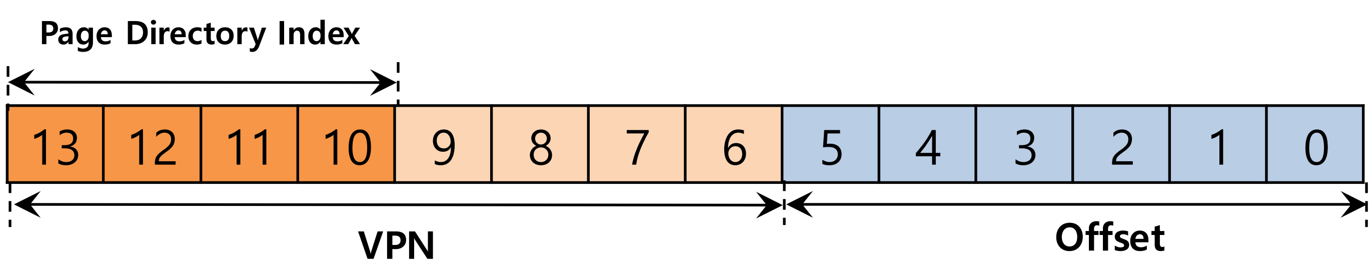
Example: single level page table



Example: two level paging

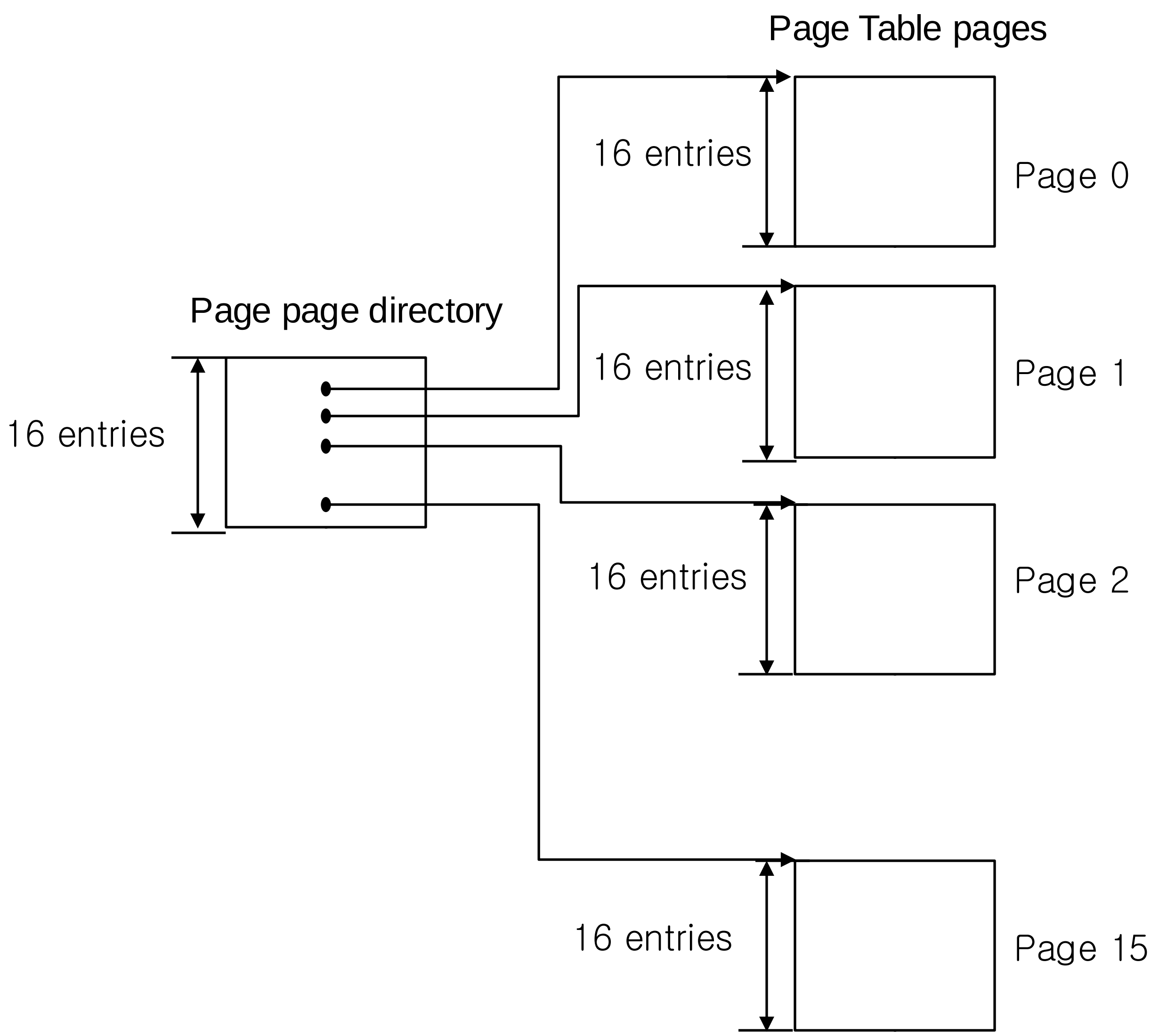
- Page directory index
  - The page table has 16 pages.
  - 16 entries for page directory: one entry per page of the page table.
  - 16*4B = 64B is required for page directory. → One page
  - 4 bits for page directory index.

```
PDEAddr = PageDirBase + (PDIndex * sizeof(PDE))
```



- If the page-directory entry is invalid, raise an exception (The access is invalid).

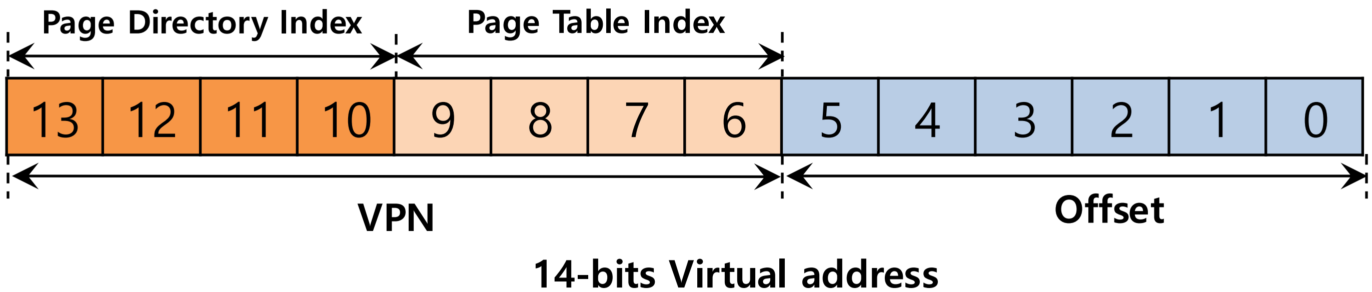
Example: two level paging



Example: Page Table Index

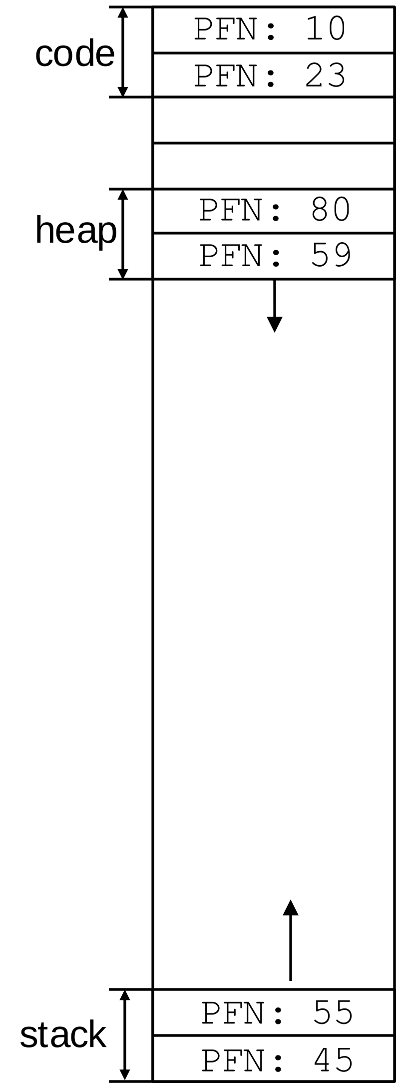
- Page table index
  - It is used to find the address of the page table entry.

```
PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
```



Examples

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-



Single level paging: 16 pages

→ Two level paging: 3 pages

Multi-level Page Table Control Flow

```
01:      VPN = (VirtualAddress & VPN_MASK) >> SHIFT
02:      (Success,TlbEntry) = TLB_Lookup(VPN)
03:      if(Success == True)       //TLB Hit
04:        if(CanAccess(TlbEntry.ProtectBits) == True)
05:              Offset = VirtualAddress & OFFSET_MASK
06:              PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
07:              Register = AccessMemory(PhysAddr)
08:        else RaiseException(PROTECTION_FAULT);
09:      else // perform the full multi-level lookup
```

- (1 lines) extract the virtual page number (VPN)
- (2 lines) check if the TLB holds the translation for this VPN
- (5-8 lines) extract the page frame number from the relevant TLB entry, and form the desired physical address and access memory

Multi-level Page Table Control Flow

```
11:     else
12:         PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13:         PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14:         PDE = AccessMemory(PDEAddr)
15:         if(PDE.Valid == False)
16:             RaiseException(SEGMENTATION_FAULT)
17:         else // PDE is Valid: now fetch PTE from PT
```

- (11 lines) extract the Page Directory Index(PDIndex)
- (13 lines) get Page Directory Entry(PDE)
- (15-17 lines) Check PDE valid flag. If valid flag is true, fetch Page Table entry from Page Table

The Translation Process: Remember the TLB

```
    PTIndex = (VPN & PT_MASK) >> PT_SHIFT
    PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if(PTE.Valid == False)
            RaiseException(SEGMENTATION_FAULT)
    else if(CanAccess(PTE.ProtectBits) == False)
            RaiseException(PROTECTION_FAULT);
    else
            TLB_Insert(VPN, PTE.PFN , PTE.ProtectBits)
            RetryInstruction()
```

Inverted Page Tables

- Keeping a single page table that has an entry for each <u>physical page</u> of the system.
- The entry tells us which process is using this page, and which virtual page of that process maps to this physical page.