# Lecture Note 9. Paging and Beyond Physical Memory

June 3, 2020

Jongmoo Choi

Dept. of Software
Dankook University
http://embedded.dankook.ac.kr/~choijm

(This slide is made by Jongmoo Choi. Please let him know when you want to distribute this slide)

# Contents

From Chap 18~22 of the OSTEP

Chap 18. Paging: Introduction
- ✓ Page Table
- ✓ Address Translation and Memory Trace

Chap 19. TLB (Translation Lookaside Buffer)
- ✓ Faster Translation
- ✓ TLB hit: Fast translation vs TLB miss: TLB management

Chap 20. Advanced Page Tables
- ✓ Multi-level Page Table
- ✓ Inverted Page Table

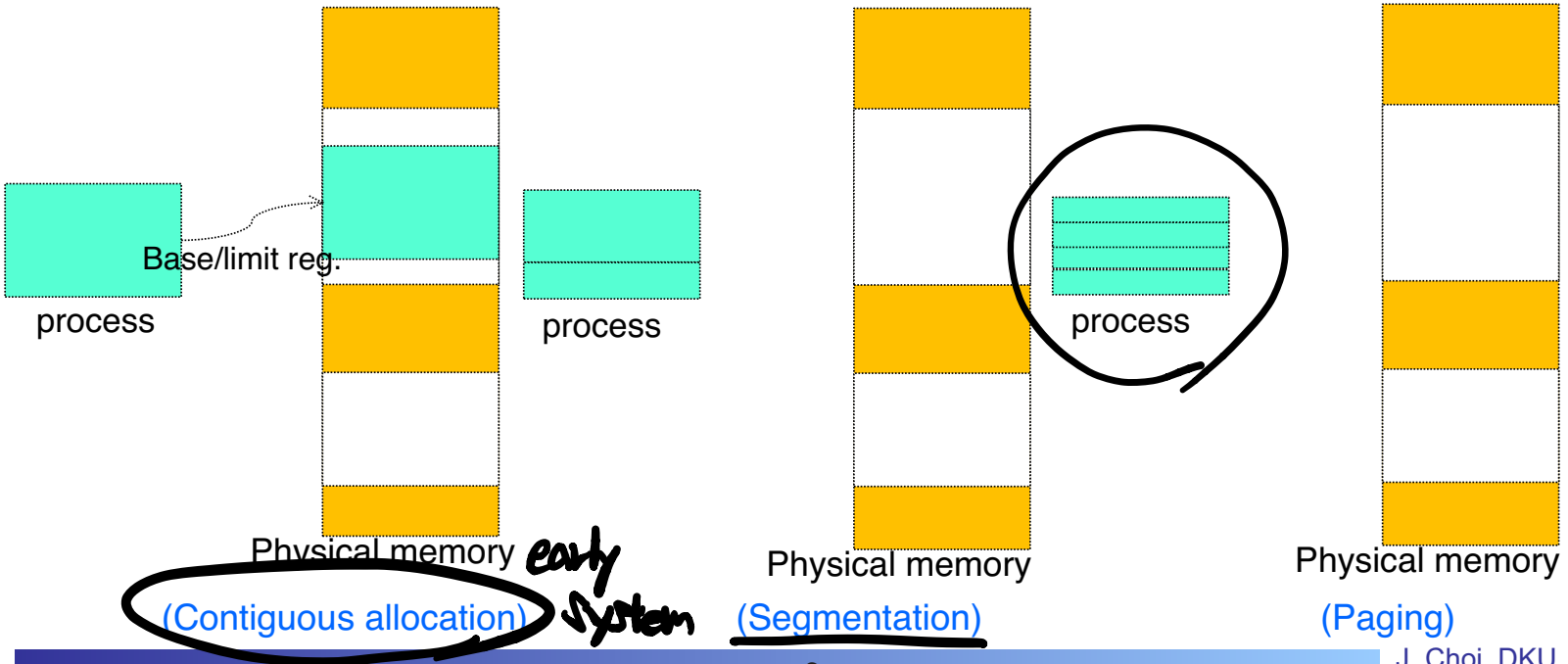
Chap 21. Beyond Physical Memory: Mechanism
- ✓ Memory Hierarchy and on-demand loading
- ✓ Swap and Page Fault

Chap 22. Beyond Physical Memory: Policies
- ✓ Cache management model: Locality, Trashing
- ✓ Page replacement policies: FIFO, LRU, OPT, Approximate LRU, …

# Executive Summary

Comparison among contiguous, segmentation and paging

- ✓ Contiguous allocation: based on base, limit register
- ✓ Non-contiguous allocation
  - Segmentation: variable size
  - Paging: fixed-size

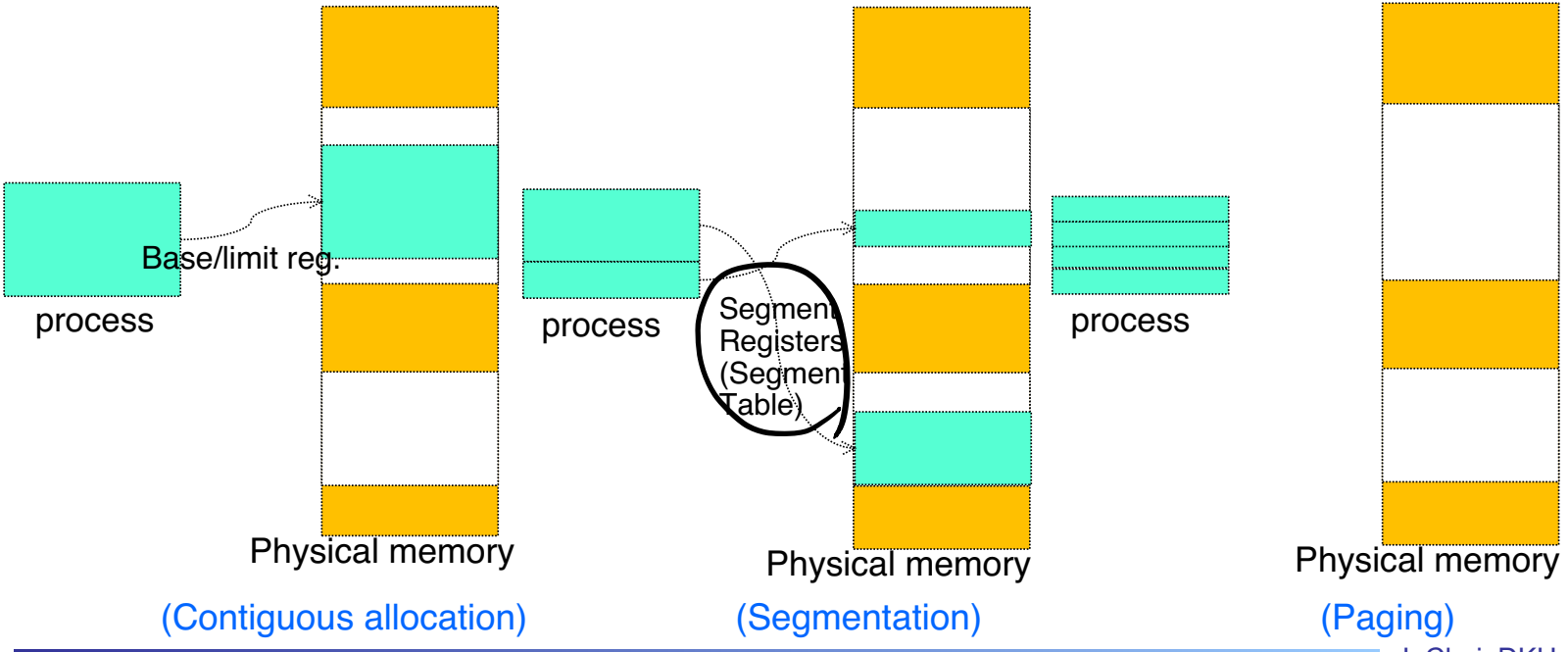


# Executive Summary

Comparison among contiguous, segmentation and paging

- ✓ Contiguous allocation: based on base, limit register
- ✓ Non-contiguous allocation

  Segmentation: variable size
  Paging: fixed-size

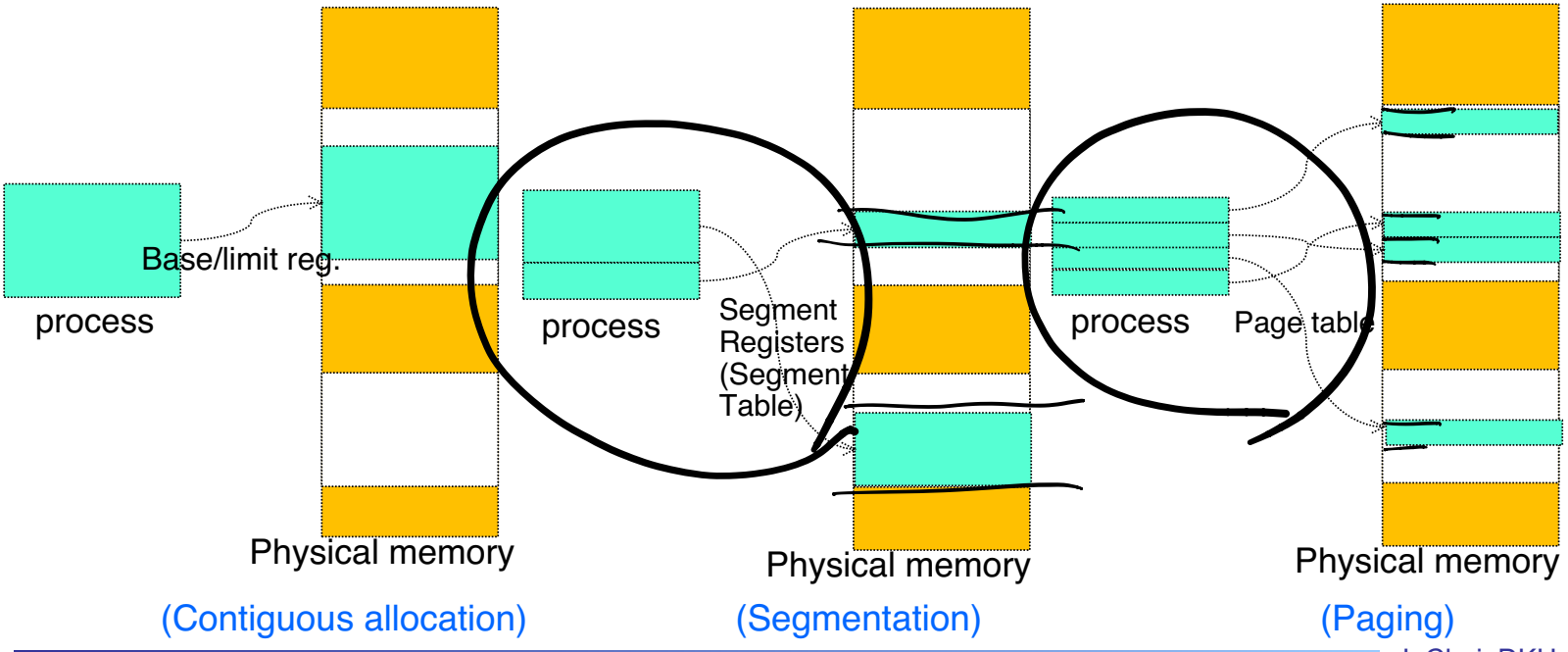


# Executive Summary

Comparison among contiguous, segmentation and paging

- ✓ Contiguous allocation: based on base, limit register
- ✓ Non-contiguous allocation

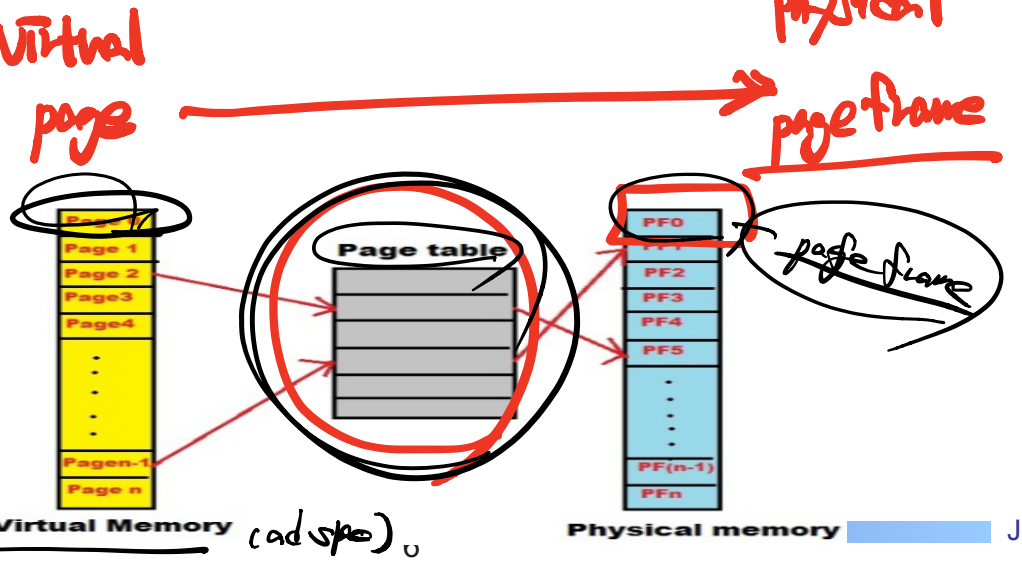
  Segmentation: variable size
  Paging: fixed-size



# Chap 18. Paging: Introduction

## Why paging?

- ✓ Two common approaches for non-contiguous management
  - Variable size: segmentation
    - Sharing, Protection support
    - Address translation: using segment table
    - But, memory becomes fragmented (external fragmentation), thus allocation becomes more challenging over time
  - Fixed size: paging
    - No external fragmentation: Easy for HW supports (e.g. TLB)
- ✓ Terms for paging
  - Virtual memory: divided into a fixed size unit called page
  - Physical memory: also divided into a fixed size unit called page frame
  - Address translation: using page table



# 18.1 A simple example and overview

## Example of Paging

- ✓ Virtual memory
  Tiny address space of a process: 64B total size, page size: 16B ? 4 pages in an address space
- ✓ Physical memory
  Tiny physical memory: 128B, page frame size: 16B ? total 8 frames
  - Frame 0 for OS itself
  - Frame 2, 3, 5 and 7 for the process (Note that they are not contiguous and not in order)
  - Other frames are managed by a free list (a bitmap or list is enough)

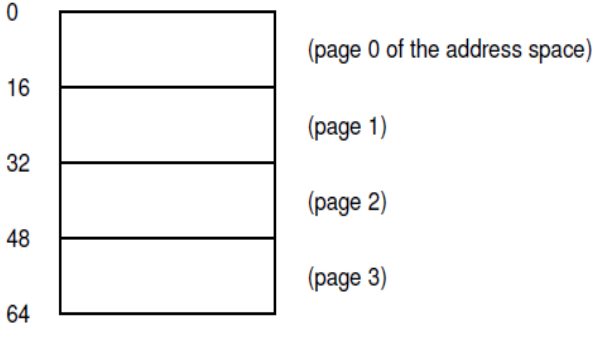



Figure 18.1: A Simple 64-byte Address Space

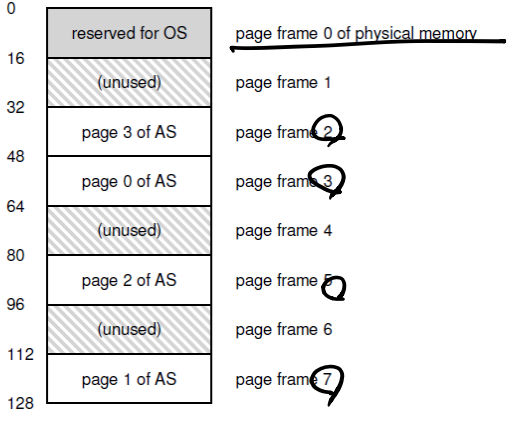



Figure 18.2: A 64-Byte Address Space In A 128-Byte Physical Memory

# 18.1 A simple example and overview

## Page table

- A data structure that records where each page is placed in physical memory (which frame): same role as segment table
- Per-process data structure
- Used for address translation

Virtual address: 4 → physical address: 3 x 16B + 4 = 52
Virtual address: 44 → physical address: 5 x 16B + 12 = 92
Virtual address: 21 → physical address: 7 x 16B + 5 = 117

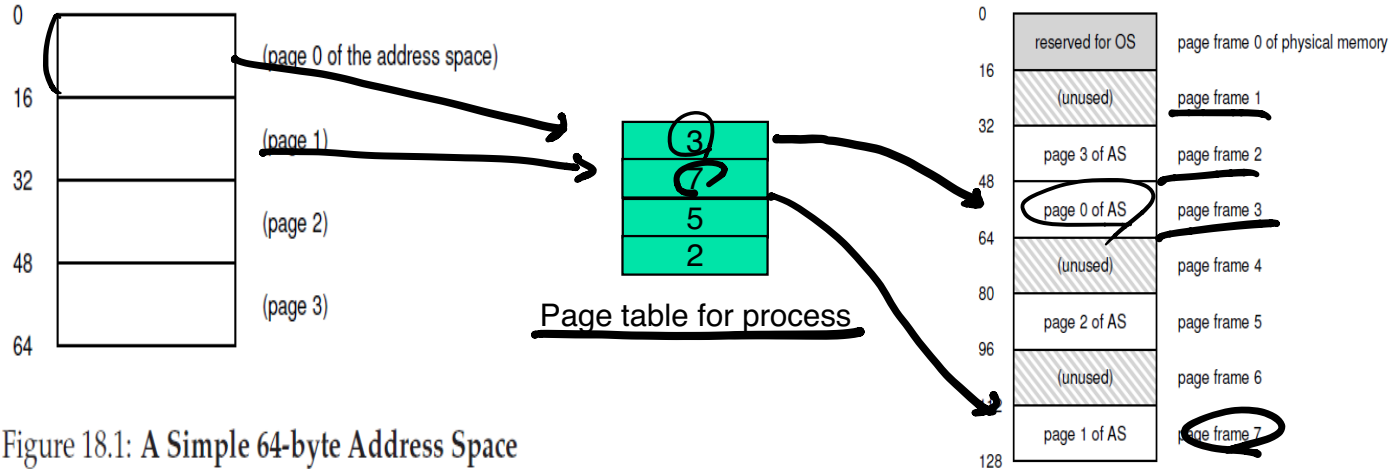



Figure 18.1: A Simple 64-byte Address Space

Figure 18.2: A 64-Byte Address Space In A 128-Byte Physical Memory

Note: all pages are not required to locate in physical memory → demand paging (chap. 21)

# 18.1 A simple example and overview

## Address translation in formal

- ✓ **Address size**
  - Address space size: 64B → virtual address size: 6-bit (26 = 64B)
    - c.f.) Address space size of 32-bit CPU: 4GB → address size: 32-bit (232 = 4GB)
  - Physical memory size: 128B → physical address size: 7-bit (27 = 128B)
- ✓ **Virtual address: consists of VPN (virtual page number) and offset**
  - Page (and frame) size: 16B → offset size: 4-bit. As the result, the remaining 2-bit becomes VPN (note that there are 4 pages (22))
  - VPN is used for searching page table: VPN → PFN (Physical Frame Number)
- ✓ **Physical address**
  - PFN x page size + offset (VPN is translated while offset is not)
- ✓ **Example**
  - Virtual address: 21 → bit: 01 0101 → VPN: 01, offset: 0101 → PFN: 111 → 111*16B + 0101 → physical address: 117

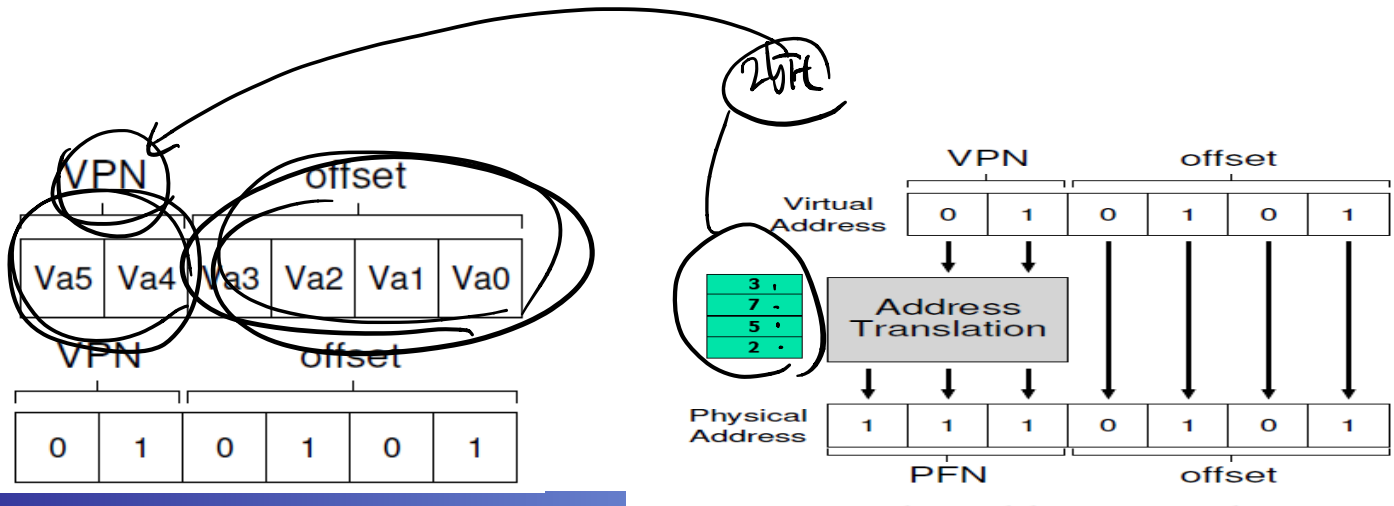
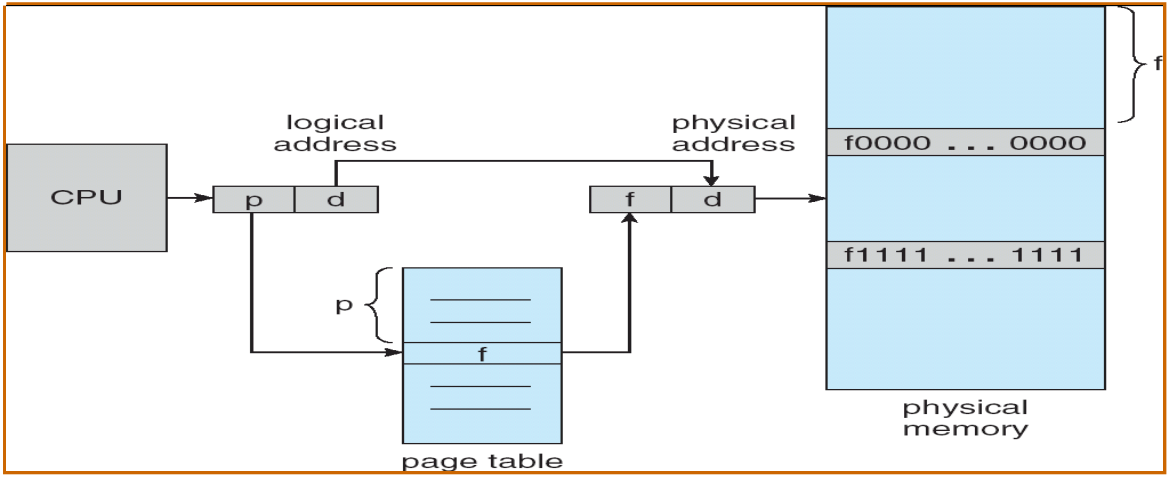



Figure 18.3: The Address Translation Process

# 18.1 A simple example and overview

## Address translation summary

- Virtual address is divided in two parts: page number(p) and offset(d)
  - Page number: used as an index into a page table
  - Offset: used to locate the physical address within a frame
- Each entry of the page table contains the starting address of the corresponding frame.
- Combining the starting address with the page offset generates physical address




(Source: A. Silberschatz, "Operating system Concept")

# 18.2 Where Are Page Table Stored?

## Page table management

- ✓ Per process data structure
- ✓ Stored in PCB (or separated data structure linked with PCB) in kernel space in memory

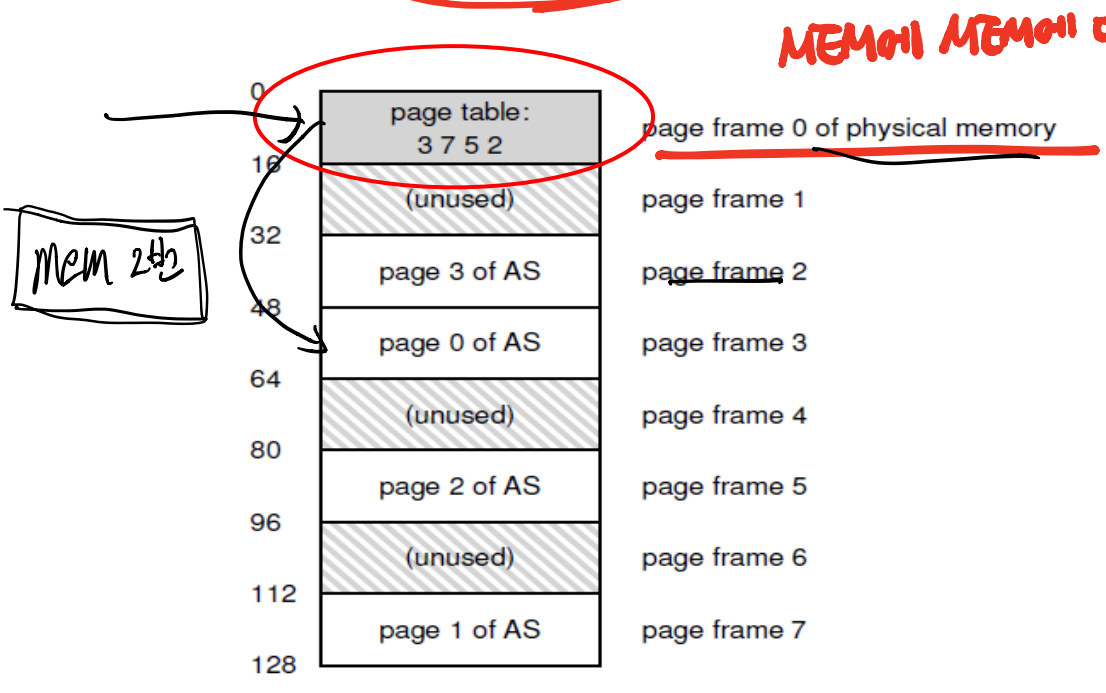
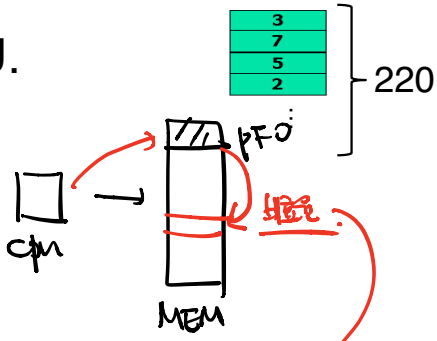



Figure 18.4: **Example: Page Table in Kernel Physical Memory**

# 18.2 Where Are Page Table Stored?

## Why in memory? (instead of CPU)

- ✓ Note that the base/limit register is in CPU.
- ✓ Since the page table is too large.
  - 32-bit CPU, page: 4KB ? offset: 12-bit, VPN: 20-bit
  - 220 entries in a page table ? (PTE: Page Table Entry)
  - Usually 4B per PTE ? 220 x 4B = 4MB size
- ✓ Assume that there are 100 processes
  - 100 x 4MB = 400MB for page tables
  - Too big to fit in a CPU ? place them in memory




## Two Issues

- ✓ Each memory access requires address translation ? translation needs to access a page table ? page table is in memory ? Does this mean that each memory access actually requires two memory accesses? ? TLB (Translation Lookaside Buffer) ? chapter 19
- ✓ Even though they are in memory, they are still big ? fixed size chopping requires a large amount of mapping information ? multi-level page table or inverted page table ? chapter 20

# 18.3 What's actually in the Page Table?

Page table

- ✓ Consists of PTEs (Page Table Entries), where each maps a page into a page frame (map a virtual address into a physical address)
  - like an array where each entry is indexed by VPN, having PFN as the value of each entry
- ✓ In addition, each PTE has several information bits
  - P (Present bit): whether this page is in physical memory or on disk (swap out)
  - R/W (Read/Write bit): Whether writes are allowed to this page
  - U/S (User/Supervisor bit): if user-mode processes can access the page
  - A (Access bit, a.k.a. reference bit): for replacement (see chapter 22)
  - D (Dirty bit): whether the page has been modified
  - Others
    - G, PAT, PCD, PWT: determine how HW caching works for the page
    - Valid bit: used or unused (e.g. space between stack and heap which is not used)
    - Various Protection bits

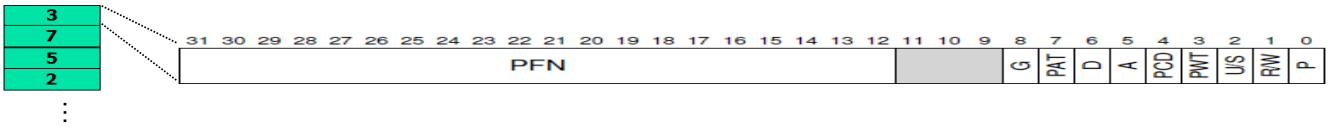


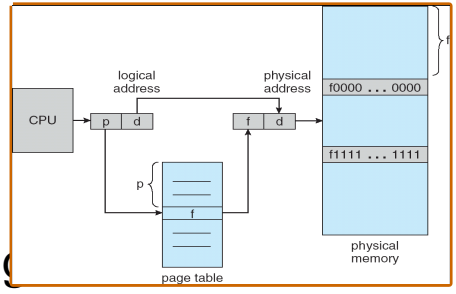

Figure 18.5: An x86 Page Table Entry (PTE)

☞ What is the difference between page fault (P bit) and segmentation fault (Valid bit)?

# 18.4 Paging: Also Too Slow

To access memory (e.g. mov 21, %eax)

- ✓ Find PTE address
  PTBR: Page table base register
- ✓ Fetch PTE /* access memory */
- ✓ Check bits
- ✓ Fetch physical address /* access memory a




```
1   // Extract the VPN from the virtual address
2   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4   // Form the address of the page-table entry (PTE)
5   PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7   // Fetch the PTE
8   PTE = AccessMemory(PTEAddr)
9
10  // Check if process can access the page
11  if (PTE.Valid == False)
12      RaiseException(SEGMENTATION_FAULT)
13  else if (CanAccess(PTE.ProtectBits) == False)
14      RaiseException(PROTECTION_FAULT)
15  else
16      // Access is OK: form physical address and fetch it
17      offset   = VirtualAddress & OFFSET_MASK
18      PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19      Register = AccessMemory(PhysAddr)
```

two Memory Access

Figure 18.6: **Accessing Memory With Paging**

# 18.5 A Memory Trace

High-level viewpoint

Assembly viewpoint

Memory trace

- ✓ Assumption
  - Page/frame size: 1KB (1024B)
  - Code: VPN:1, PFN:4 (PA = 4*1024)
  - Array: VPN:39, PFN:7 (PA = 7*1024)
  - PT: located in PA 1024
- ✓ Figure 18.7: first five loop
  - PT[1] for instruction address
  - Instruction fetch
  - PT[39] for data address
  - Data fetch
  - 10 memory accesses per each loop (4 for instruction, 1 for

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;

1024 movl $0x0,(%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne  0x1024
```

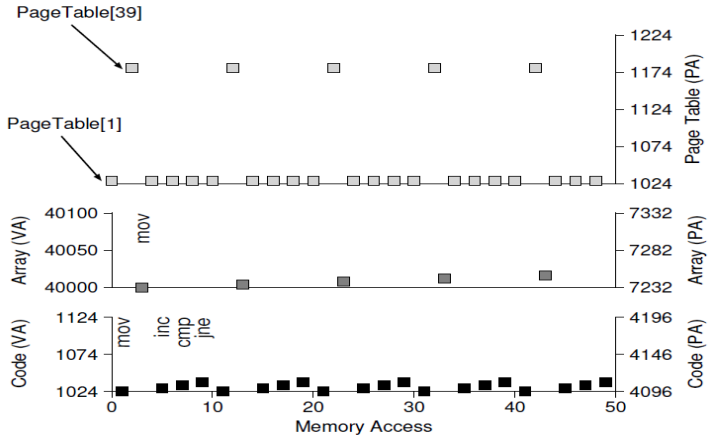


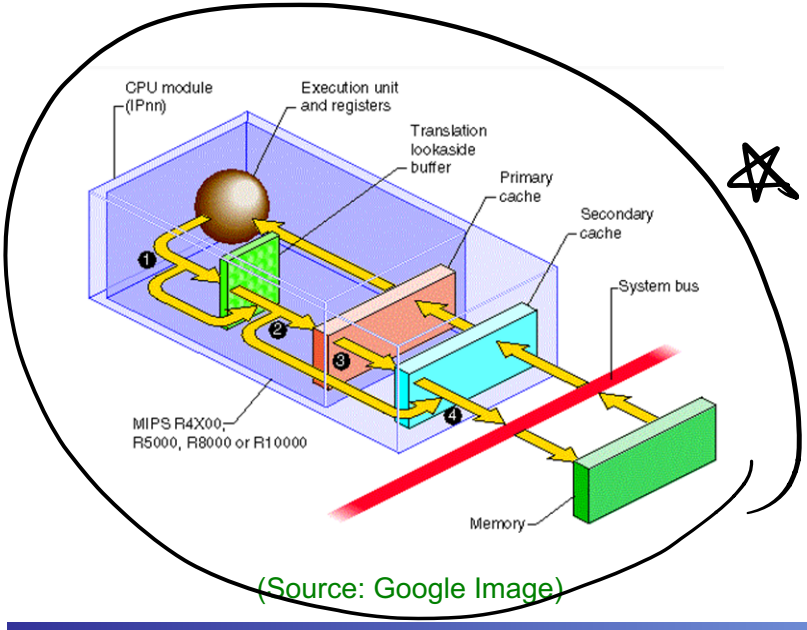

Figure 18.7: **A Virtual (And Physical) Memory Trace**

- Simple question to take attendance: 1) Explain why the VA 1074 is translated into the PA 4146 in Figure 18.7. Why the VA 40050 is translated into the PA 7282? (explain using the term of VPN and offset). 2) Discuss the structure of page table in the above process and how many entries are in the table (assume that the page/frame size is 1KB and the total size of virtual memory is 64KB (hint: see page 10 of chapter 18 in OSTEP)) (until 6 PM, June 11th).

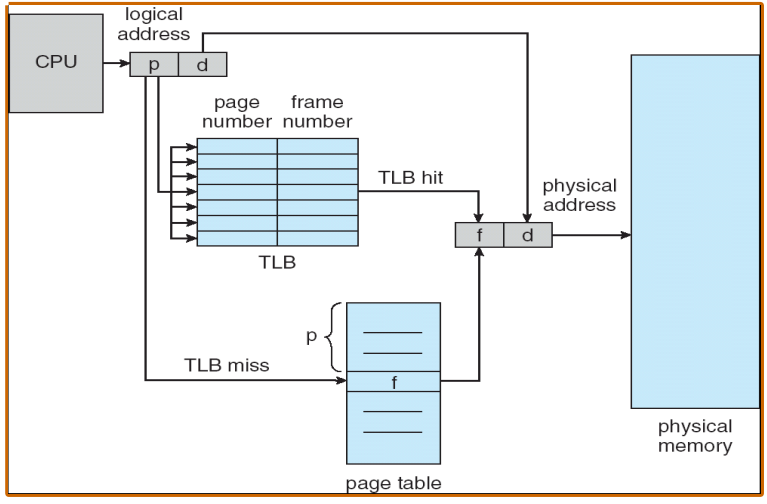
# Chap. 19 Paging: Faster Translations (TLBs)

## TLB (Translation Lookaside Buffer)

- A part of MMU (Memory Management Unit) for faster translation
- Cache of recent used PTEs (popular virtual-to-physical pairs)
  a better name would be an address-translation cache
- Translation step: 1) HW first check TLB, 2) if (hit), translation performs quickly without having to consult PT, 3) otherwise, access PT, 4) update TLB to cache the recently used PTE




(Source: Google Image)




(Source: A. Silberschatz, "Operating system Concept")

# 19.1 TLB Basic Algorithm

How HW might handle an address translation?

- ✓ Hit ? only one memory access (line 7)
- ✓ Miss ? two accesses, one for PTE (line 12) and the other for read data access (line 7 via line 19) + TLB update (line 18)
- ✓ Locality: most accesses hit in TLB

```
1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3   if (Success == True)    // TLB Hit
4       if (CanAccess(TlbEntry.ProtectBits) == True)
5           Offset   = VirtualAddress & OFFSET_MASK
6           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7           Register = AccessMemory(PhysAddr)
8       else
9           RaiseException(PROTECTION_FAULT)
10  else                    // TLB Miss
11      PTEAddr = PTBR + (VPN * sizeof(PTE))
12      PTE = AccessMemory(PTEAddr)
13      if (PTE.Valid == False)
14          RaiseException(SEGMENTATION_FAULT)
15      else if (CanAccess(PTE.ProtectBits) == False)
16          RaiseException(PROTECTION_FAULT)
17      else
18          TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19          RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**

# 19.2 Example: Accessing an Array

Example code

- ✓ int a[10] ? 4B x 10
- ✓ Page size: 16B ? 4 array entries at most ? Assume Figure 19.2 layout
- ✓ Memory access behavior
  - Access a[0] ? TLB miss ? two memory accesses
  - Access a[1],a[2] ? TLB hit ? one memory access
  - Access a[3],a[7] ? TLB miss, Access a[4/5/6], a[8/9] ? TLB hit
  - TLB hit ratio: 70% (usually > 99% in general)

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

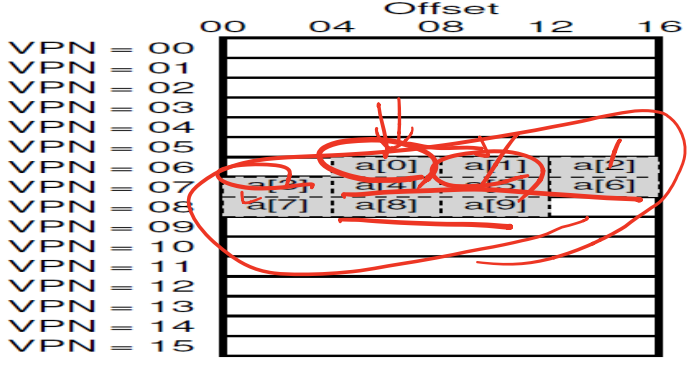



Figure 19.2: Example: An Array In A Tiny Address Space

- ☞ If the page size is 32B, how is TLB miss ratio?
- ☞ What about if there exists an outer loop? (e.g. “for (j=0; j<2; j++)”)

# 19.2 Example: Accessing An Array

## Use caching when possible

### TIP: USE CACHING WHEN POSSIBLE

Caching is one of the most fundamental performance techniques in computer systems, one that is used again and again to make the "common-case fast" [HP06]. The idea behind hardware caches is to take advantage of **locality** in instruction and data references. There are usually two types of locality: **temporal locality** and **spatial locality**. With temporal locality, the idea is that an instruction or data item that has been recently accessed will likely be re-accessed soon in the future. Think of loop variables or instructions in a loop; they are accessed repeatedly over time. With spatial locality, the idea is that if a program accesses memory at address $x$, it will likely soon access memory near $x$. Imagine here streaming through an array of some kind, accessing one element and then the next. Of course, these properties depend on the exact nature of the program, and thus are not hard-and-fast laws but more like rules of thumb.

Hardware caches, whether for instructions, data, or address translations (as in our TLB) take advantage of locality by keeping copies of memory in small, fast on-chip memory. Instead of having to go to a (slow) memory to satisfy a request, the processor can first check if a nearby copy exists in a cache; if it does, the processor can access it quickly (i.e., in a few CPU cycles) and avoid spending the costly time it takes to access memory (many nanoseconds).

You might be wondering: if caches (like the TLB) are so great, why don't we just make bigger caches and keep all of our data in them? Unfortunately, this is where we run into more fundamental laws like those of physics. If you want a fast cache, it has to be small, as issues like the speed-of-light and other physical constraints become relevant. Any large cache by definition is slow, and thus defeats the purpose. Thus, we are stuck with small, fast caches; the question that remains is how to best use them to improve performance.

# 19.3 Who Handles the TLB Miss?

## Two approaches

- ✓ **HW-managed TLB**
  - HW has a logic to manipulate TLB including TLB update
  - HW must exactly know the PT format, address format, …
  - E.g.) Intel CPU ? CISC
- ✓ **SW-managed TLB**
  - HW simply raises an exception
  - OS (TLB trap handler) explicitly manages TLB ? more flexible
  - E.g.) MIPS, Sun SPARC v9 ? RISC

```
1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3   if (Success == True)    // TLB Hit
4       if (CanAccess(TlbEntry.ProtectBits) == True)
5           Offset   = VirtualAddress & OFFSET_MASK
6           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7           Register = AccessMemory(PhysAddr)
8       else
9           RaiseException(PROTECTION_FAULT)
10  else                    // TLB Miss
11      RaiseException(TLB_MISS)
```

Figure 19.3: TLB Control Flow Algorithm (OS Handled)

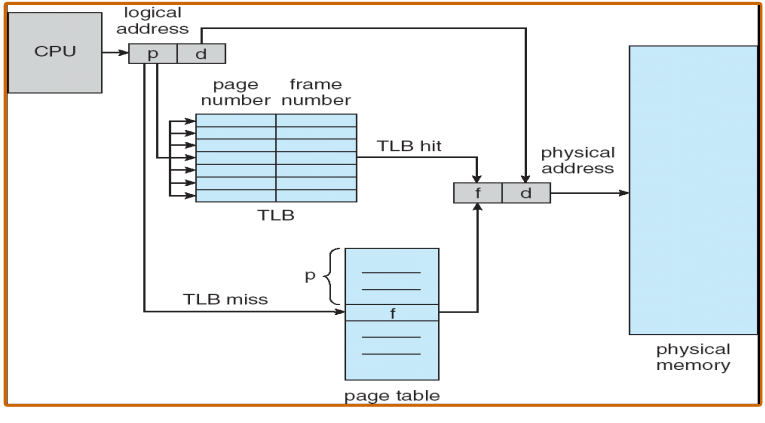
# 19.4 TLB Contents: What's in There?

## A TLB entry

- VPN + PFN + bits (32 or 64 or 128 bits)
- Bits

  VPN | PFN | other bits

  Valid bit: whether the entry has a valid translation or not
  Protection bits: R/W/E
  Others: ASID (Address-Space IDentifier), dirty bit, …
- Fully-associative
  Can place any entry
- Search in parallel

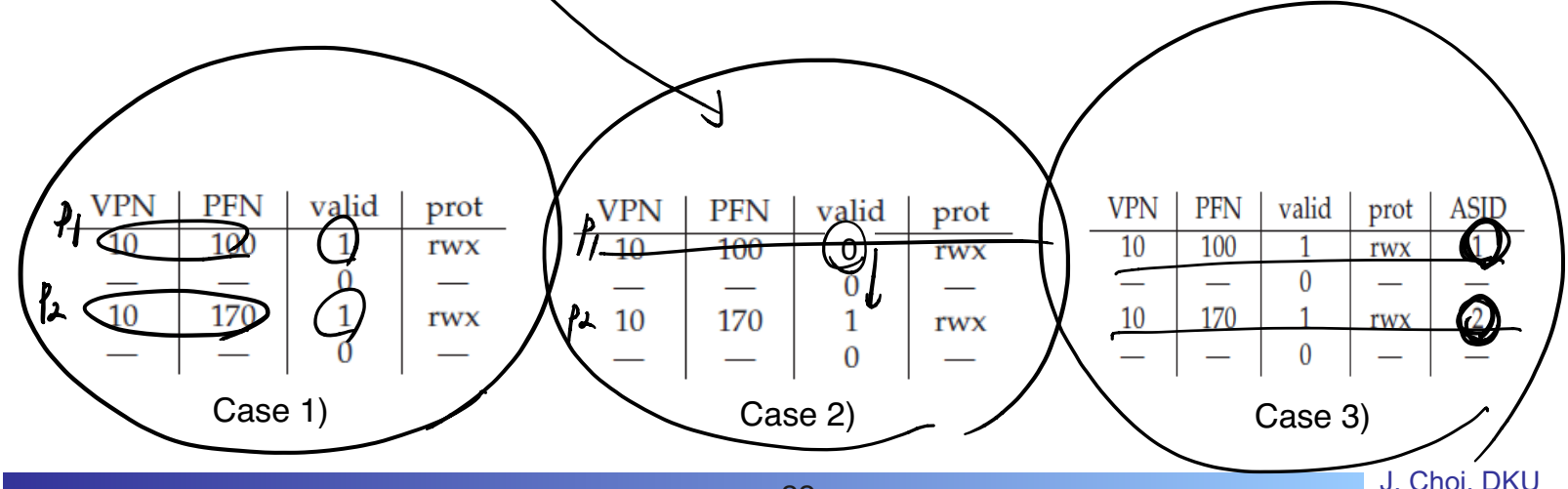



(Source: A. Silberschatz, "Operating system Concept")

# 19.5 TLB Issue: Context Switches

## TLB

- ✓ Contains virtual-to-physical mapping
- ✓ Only valid for current-running process
  - Context Switch ? need to invalid or distinguish TLB entries btw processes
- ✓ Example
  - P1: VPN 10 ? PFN 100, P2: VPN 10 ? PFN 170
  - P1 run ? P1 accesses VPN 10 ? CS from P1 to P2 ? P2 accesses VPN 10 ? Case 1: cause problem
  - Solution: 1) flush before CS (set all valid bit as 0) ? case 2, 2) ASID (Address Space IDentifier) ? case 3
    - TLB flush is a heavy operation ? causing high TLB misses



P1

| VPN | PFN | valid | prot |
|---|---|---|---|
| 10 | 100 | 1 | rwx |
| — | — | 0 | — |
| 10 | 170 | 1 | rwx |
| — | — | 0 | — |

Case 1)

P1

| VPN | PFN | valid | prot |
|---|---|---|---|
| 10 | 100 | 0 | rwx |
| — | — | 0 | — |
| 10 | 170 | 1 | rwx |
| — | — | 0 | — |

Case 2)

| VPN | PFN | valid | prot | ASID |
|---|---|---|---|---|
| 10 | 100 | 1 | rwx | 1 |
| — | — | 0 | — | — |
| 10 | 170 | 1 | rwx | 2 |
| — | — | 0 | — | — |

Case 3)

# Chap. 20 Paging: Smaller Tables

## Page table

- ✓ Locate in main memory
- ✓ How large it is?
  - 32-bit address space (232), 4KB page size (212) ? PTEs in PT = 220
  - PTE size = 4B ? 4MB for a PT
  - Note that PT is managed per a process (400MB if there are 100 processes) ? May cause a memory shortage
- ✓ How to make smaller PT?
  - Bigger pages (page size: 4KB ? 2MB, called huge page)
  - Hybrid approach: segmentation + paging
  - Multi-level page table
  - Inverted page table

# 20.1 Simple Solution: Bigger Pages

## Bigger pages

- 32-bit address space (232), 4B for PTE
  - 4KB Page size (212) ? PTEs in PT = 220 ? PT size = 4MB
  - 8KB Page size (213) ? PTEs in PT = 219 ? PT size = 2MB
  - 4MB Page size (222) ? PTEs in PT = 210 ? PT size = 4KB
- Pros): Simple, positive effect on TLB hit
- Cons): Internal fragmentation (waste of memory), loading time
- How about multiple sizes for page?
  - Support 4KB, 16KB and 4MB at the same time (like huge page + base page in Intel)
  - Pros): Flexible, less internal fragmentation
  - Cons): Complexity in OS (still in progress)

# 20.2 Hybrid Approach: Paging and Segments

## Hybrid approach

- ✓ Limit in a segment can reduce PT size
- ✓ Simple example: 16KB address space, 1KB page size

Use 4 pages (1 for code, 1 for heap and 2 for stack) ? Fig. 20.1
Place in page frame 10, 23, 28 and 4 respectively
Paging only: Fig. 20.2

- 16 PTES, most PTEs are invalid

Hybrid approach

- Code: base = 0, limit = 1K ? 1 PTE for code, Heap: base=4K, limit = 5K ? 1 PTE for heap, Stack: base=14K, limit = 16K ? 2 PTE for stack
- The limit register holds the maximum valid pages ? access above the limit generates the segmentation fault ? PT can hold valid page only
- E.g. Intel) Virtual addr. ? segmentation ? Linear addr. ? Paging ? Physical addr.

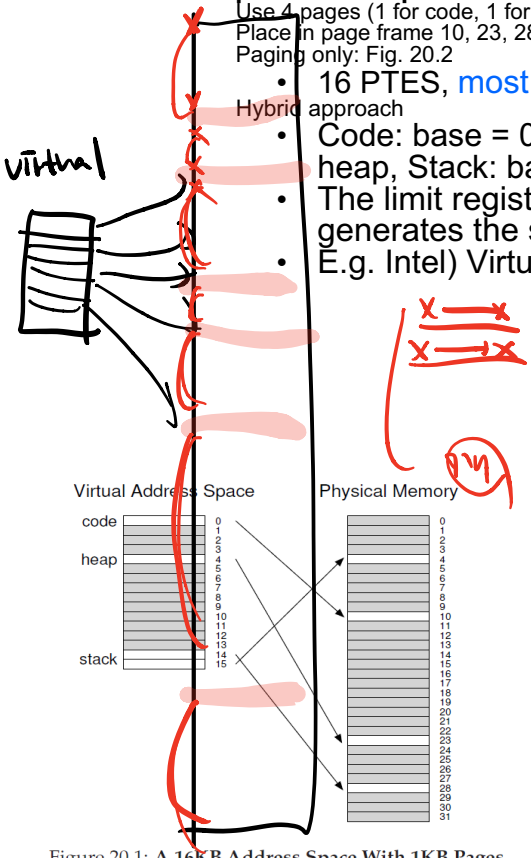



Figure 20.1: **A 16KB Address Space With 1KB Pages**

| PFN | valid | prot | present | dirty |
|---|---|---|---|---|
| 10 | 1 | r-x | 1 | 0 |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| 23 | 1 | rw- | 1 | 1 |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| 28 | 1 | rw- | 1 | 1 |
| 4 | 1 | rw- | 1 | 1 |

Figure 20.2: **A Page Table For 16KB Address Space**

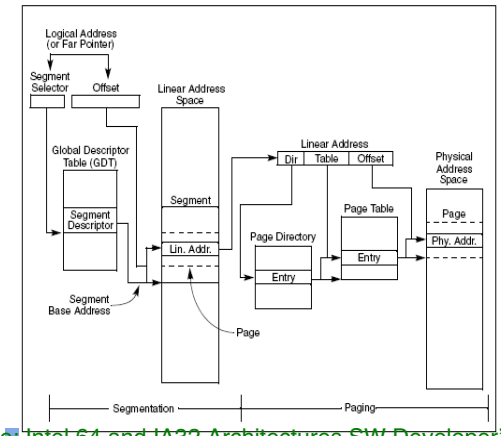



Figure 3-1. Segmentation and Paging

(Source: Intel 64 and IA32 Architectures SW Developer's Manual, Volume 3: System Programming Guide)

# 20.3 Multi-level Page Table

## Idea

- ✓ Using structural PTs (instead of linear table) ? multi-level table
  - Page directory: each entry represents related PT
  - Can reduce a large number of unallocated page tables using valid bits in the page directory

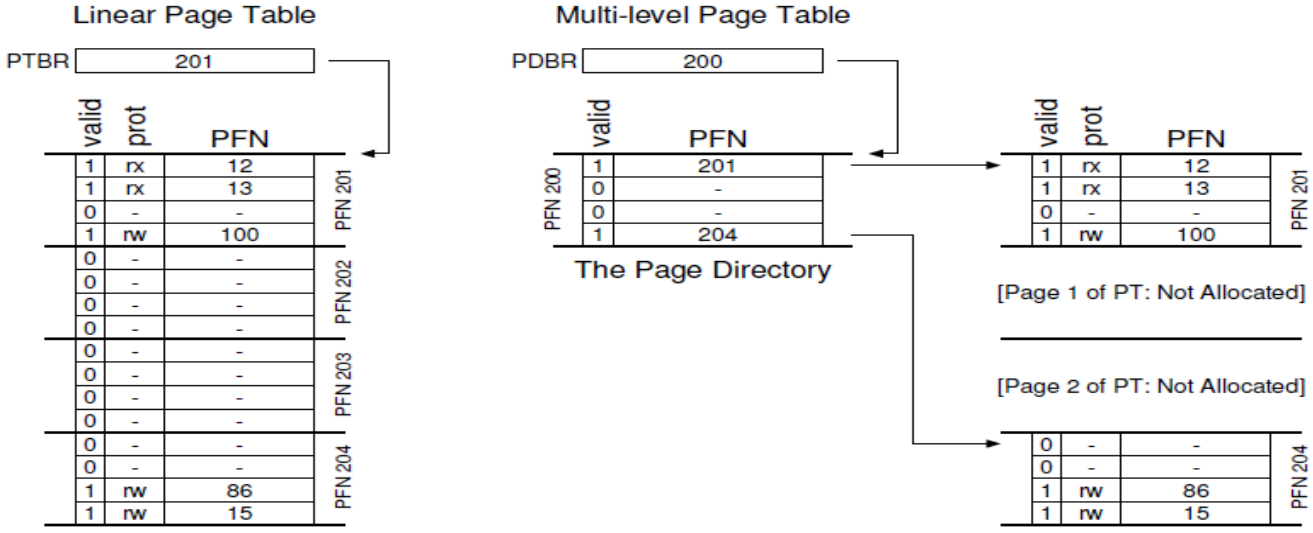



Figure 20.3: **Linear (Left) And Multi-Level (Right) Page Tables**

# 20.3 Multi-level Page Table

## Example: how much memory space for PT can be reduced?

- ✓ Address space: 16KB (14bit), page size: 64B (6bit) ? 256 PTEs ( 8bit)
  - Assume 6 pages (0, 1, 4, 5, 254, 255) are used for code, heap and stack? Figure 20.4
  - Pages are allocated in frames 10, 23, 80, 59, 55 and 45, respectively
- ✓ How many frames are needed for PT in the linear approach?
  - Total 256 PTEs ? 16 PTE (64B/4B) in a frame ? 16 frames
- ✓ How about in the multi-level approach?
  - 1 directory + 2 last-level PTs ? 3 frames (around 20%)

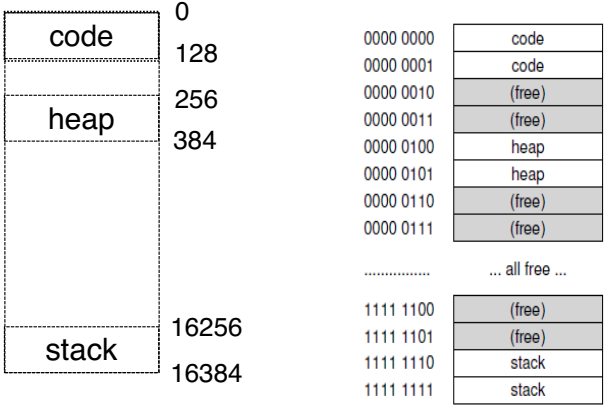



Figure 20.4: **A 16KB Address Space With 64-byte Pages**

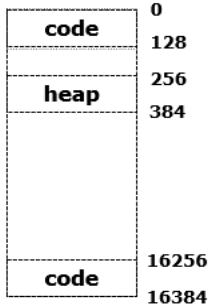
| Page Directory | | Page of PT (@PFN:100) | | | Page of PT (@PFN:101) | | |
|---|---|---|---|---|---|---|---|
| PFN | valid? | PFN | valid | prot | PFN | valid | prot |
| 100 | 1 | 10 | 1 | r-x | — | 0 | — |
| — | 0 | 23 | 1 | r-x | — | 0 | — |
| — | 0 | — | 0 | — | — | 0 | — |
| — | 0 | — | 0 | — |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

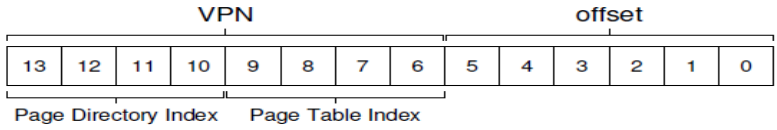
Figure 20.5: **A Page Directory, And Pieces Of Page Table**

# 20.3 Multi-level Page Table

## Address translation

- ✓ Virtual address is divided into three parts: Directory index, PT index and offset (instead of two parts: VPN, offset)
  - Virtual memory size: 16KB ? address : 14bit
  - Page size: 64B ? offset bit: 6bit
  - PTEs in a frame: 16 ? PT index: 4bit
  - PTEs in a directory: 16 ? Directory index: 4bit




| 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- |

Figure 20.5: **A Page Directory, And Pieces Of Page Table**

- E.g. 1) VA = 100 ? 00 0000 0110 0100 ? Directory: 0000, PT index: 0001, Offset: 100100 ? PA = 23 * 64B + 32 + 4
- E.g. 2) VA= 300 ? 00 0001 0010 1100 ? Directory: 0000, PT index: 0100, offset: 101100 ? PA = 80* 64B + 32+8+4
- E.g. 3) 16257 = 11 1111 1000 0001 ? Directory: 1111, PT index: 1110, offset: 000001 ? PA = 55 x 64B + 1
- E.g. 4) VA= 200 ? 00 0000 1100 1000 ? Directory: 0000, PT index: 0011, offset: 001000 ? invalid in PT
- E.g. 5) VA= 1030 ? 00 0100 0000 0110 ? Directory: 0001, PT index: 0000, offset: 000110 ? invalid in directory

# 20.3 Multi-level Page Table

## Address translation in Pseudo-code

- ✓ Concerns of the Multi-level PT
  - Address translation requires two accesses to PTs (vs. one access in the linear approach)
  - Increased HW complexity for multi-level translation
- ✓ Remember TLB ? It can hide them

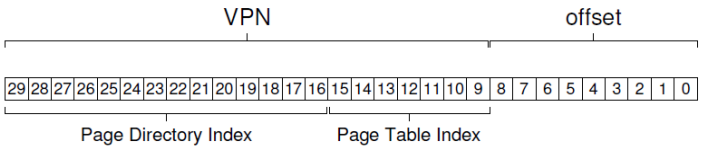
```
1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3   if (Success == True)    // TLB Hit
4       if (CanAccess(TlbEntry.ProtectBits) == True)
5           Offset   = VirtualAddress & OFFSET_MASK
6           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7           Register = AccessMemory(PhysAddr)
8       else
9           RaiseException(PROTECTION_FAULT)
10  else                    // TLB Miss
11      // first, get page directory entry
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
18          // PDE is valid: now fetch PTE from page table
19          PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20          PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
21          PTE     = AccessMemory(PTEAddr)
22          if (PTE.Valid == False)
23              RaiseException(SEGMENTATION_FAULT)
24          else if (CanAccess(PTE.ProtectBits) == False)
25              RaiseException(PROTECTION_FAULT)
26          else
27              TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28              RetryInstruction()
```

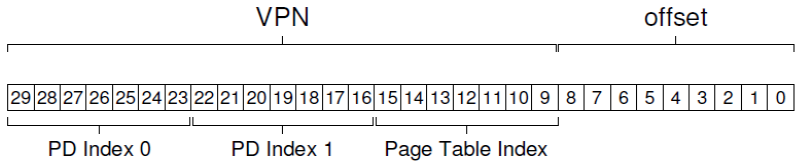
Figure 20.6: **Multi-level Page Table Control Flow**

# 20.3 Multi-level Page Table

## More than two levels

- ✓ Virtual address: 30-bit, page size: 512B
  - Address: 30bit, offset: 9bit ? VPN: 21bit, PTEs in a page: 128 (512/4)
  - 2-level: left figure
    - PT index = 7 bit (27 = 128), Page directory: need to cover remaining 14-bits ? 214 PTEs ? 128 pages for Page directory
  - 3-level: right figure
    - PT index = 7 bit, PD index0 = 7 bit (upper-level), PD index1 = 7 bit ? one page for PD index0, PD index1 and PT needed only for valid PTEs ? save memory



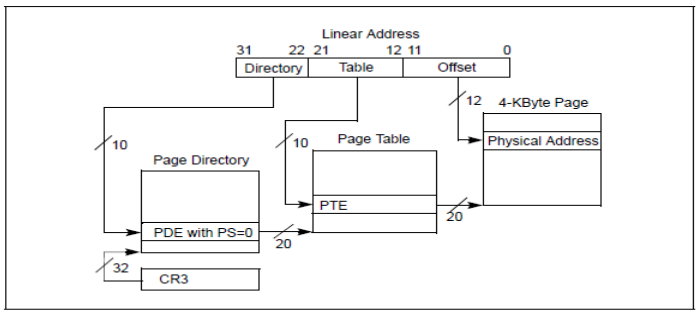




Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

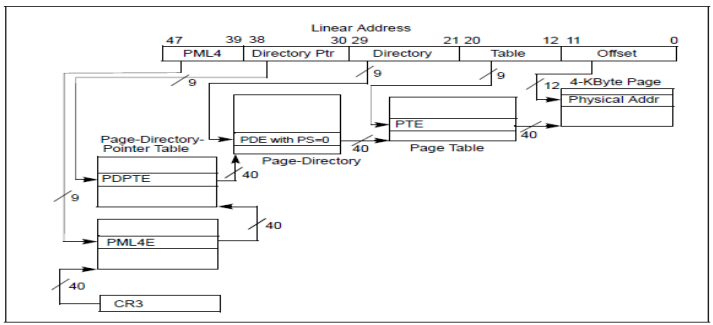



Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

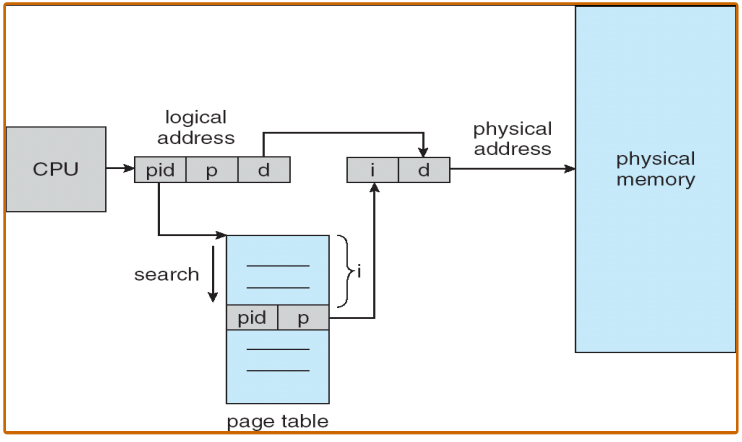
# 20.4 Inverted Page Tables

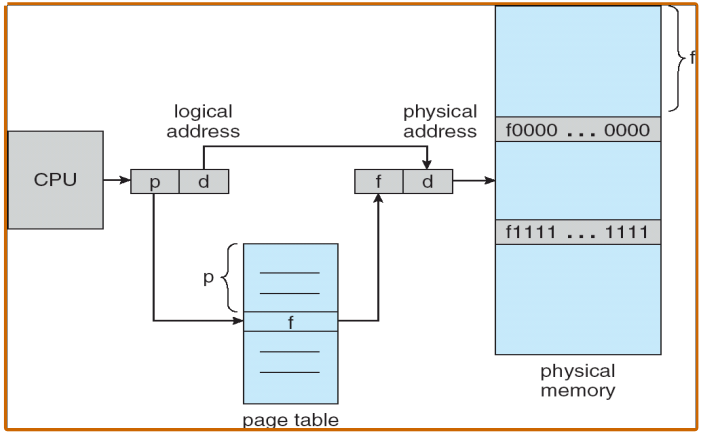
## Page table

- ✓ VPN ? PFN, One per process in a system

## Inverted Page Table

- ✓ PFN ? VPN, Only one in a system (hence reduce memory for PT)
  - Page table index: physical frame number (one entry per physical page)
  - PTE: virtual page number, process ID that maps the physical page
- ✓ Address Translation
  - Need search: 1) linear scan, 2) hash



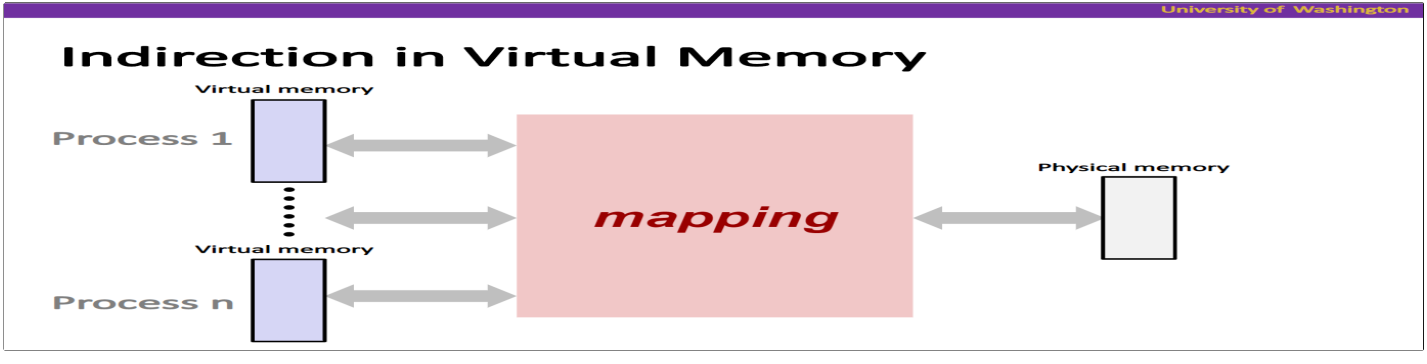




(Source: A. Silberschatz, "Operating system Concept")

# 20.6 Summary

Indirection(Translation): popularly used in a variety area of CS

- ✓ Virtual address ? Physical address
- ✓ Domain name ? IP address in Internet
- ✓ Key ? Value in No-SQL DB
- ✓ VM in Cloud ? PM in Infrastructure
- ✓ …

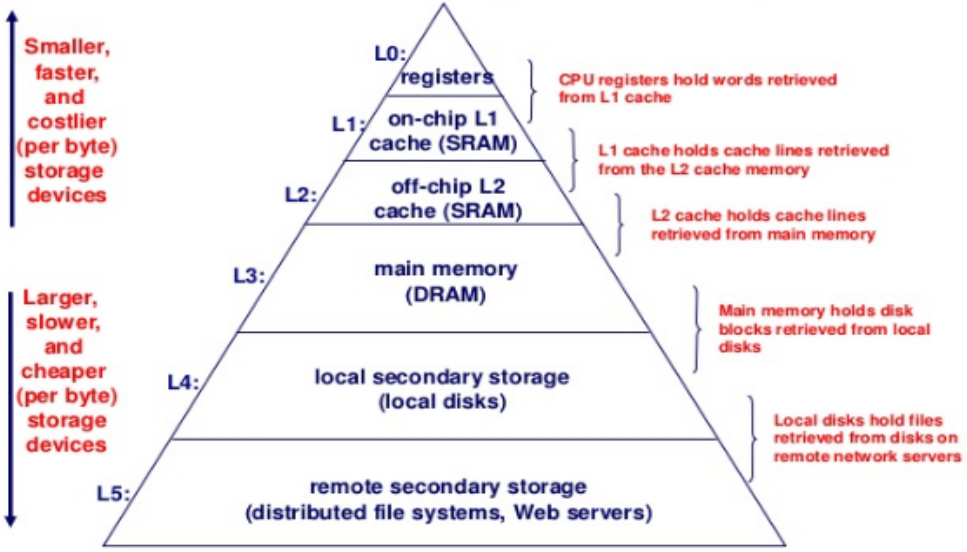


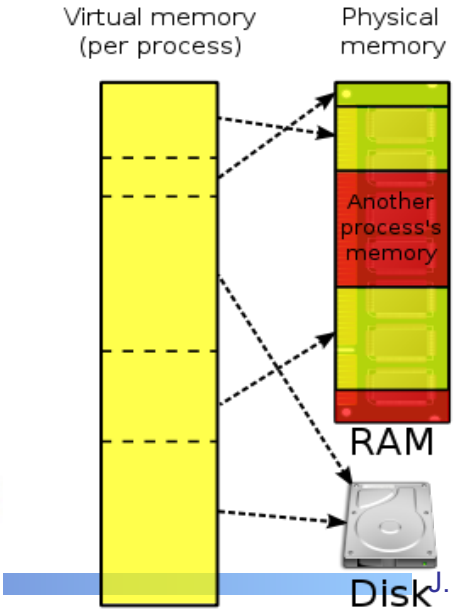

- Simple question to take attendance: 1) In page 18, we know the TLB hit ratio for accessing a [] is 70%. What is the TLB hit ratio when the page size is 32B (instead of 16B)? What if there exists an outer loop like “for (j=0; j<2; j++)” above the “for (i=0; i<10; i++)” . 2) Explain the similarity and difference between Paging discussed in here and FTL (Flash Translation Layer) discussed in page 31 of LN 7 (Advanced FS). (until 6 PM, June 17th). J

# Chap. 21 Beyond Physical Memory: Mechanisms

## Memory hierarchy

- ✓ Register, Cache, Memory, Disk (or SSD), Sever, …
- ✓ VM focus on Memory and Disk
  - Memory: relatively fast but small
  - Disk: relatively slow but large
- ✓ OS wants to execute multiple processes at the same time
  - Frequently accessed data ? place in memory
  - Seldom accessed data ? place in disk, bring into memory if necessary ( demand loading or demand paging)



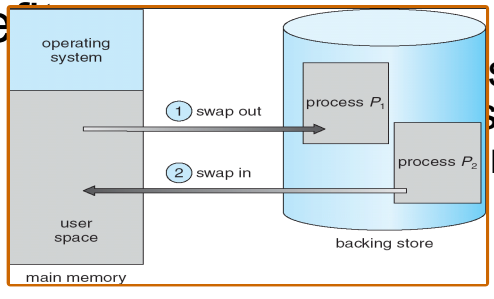
# 21.1 Swap space

## Swap definition

- ✓ Space in disk for moving pages back and forth
  - To migrate data from memory to disk when available memory space is insufficient
  - Moving granularity: page vs. process (light vs heavy memory hungry condition)
  - E.g.) 4 frames and 8-page swap space
    - Proc 0/1/2 ? ready or running, Proc 3 ? suspended (swap out)

## Bene

- ✓ sion o
- ✓ sical m
- ✓ mers (




(Source: A. Silberschatz, "Operating system Concept")

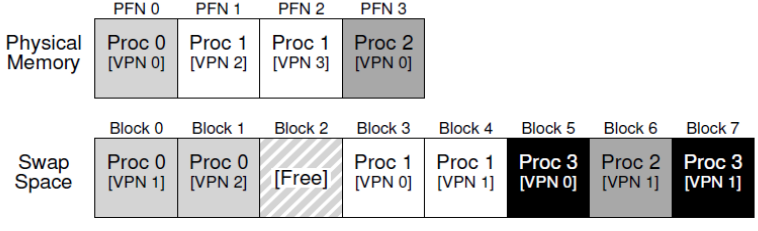



Figure 21.1: **Physical Memory and Swap Space**

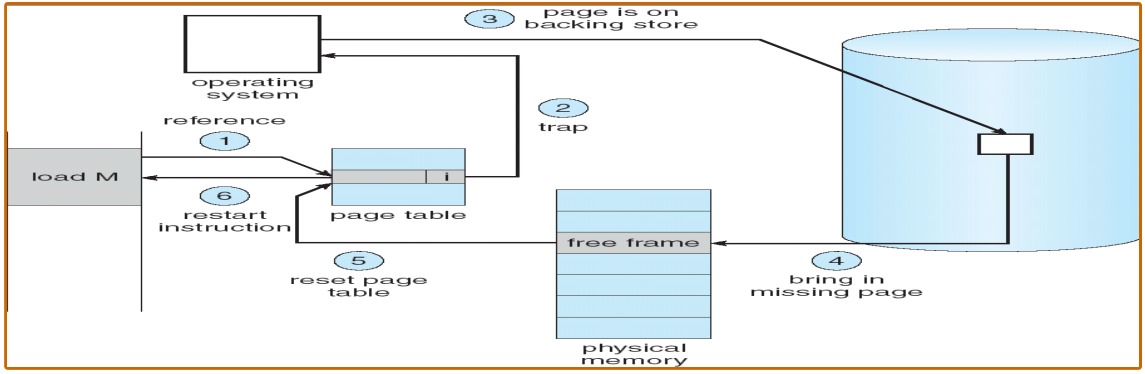
# 21.2 Present Bit / 21.3 Page Fault

## Present bit in PTE

- ✓ To identify whether a page is in memory or swap out
  - Present bit == 1, access the page
  - Present bit == 0, ? page fault

## Page fault

- ✓ Trigger page fault handler that bring the page from disk to memory
- ✓ From swap space or from a file (e.g. demand loading)




(Source: A. Silberschatz, "Operating system Concept")

☞ Load a page in virtual memory ?Read a disk block in file system (using inode) ? VM and FS works together in a integrated manner.

# 21.4 Page Fault Control Flow

## HW control flow

```
1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3   if (Success == True)    // TLB Hit
4       if (CanAccess(TlbEntry.ProtectBits) == True)
5           Offset   = VirtualAddress & OFFSET_MASK
6           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7           Register = AccessMemory(PhysAddr)
8       else
9           RaiseException(PROTECTION_FAULT)
10  else                    // TLB Miss
11      PTEAddr = PTBR + (VPN * sizeof(PTE))
12      PTE = AccessMemory(PTEAddr)
13      if (PTE.Valid == False)
14          RaiseException(SEGMENTATION_FAULT)
15      else
16          if (CanAccess(PTE.ProtectBits) == False)
17              RaiseException(PROTECTION_FAULT)
18          else if (PTE.Present == True)
19              // assuming hardware-managed TLB
20              TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21              RetryInstruction()
22          else if (PTE.Present == False)
23              RaiseException(PAGE_FAULT)
```

Figure 21.2: **Page-Fault Control Flow Algorithm (Hardware)**

```
1   PFN = FindFreePhysicalPage()
2   if (PFN == -1)              // no free page found
3       PFN = EvictPage()       // run replacement algorithm
4   DiskRead(PTE.DiskAddr, pfn) // sleep (waiting for I/O)
5   PTE.present = True          // update page table with present
6   PTE.PFN     = PFN           // bit and translation (PFN)
7   RetryInstruction()          // retry instruction
```

Figure 21.3: **Page-Fault Control Flow Algorithm (Software)**

☞ What are the differences between the page fault and segmentation fault? What if there is no free frame? ⇨ Evict (see Chapter 22)

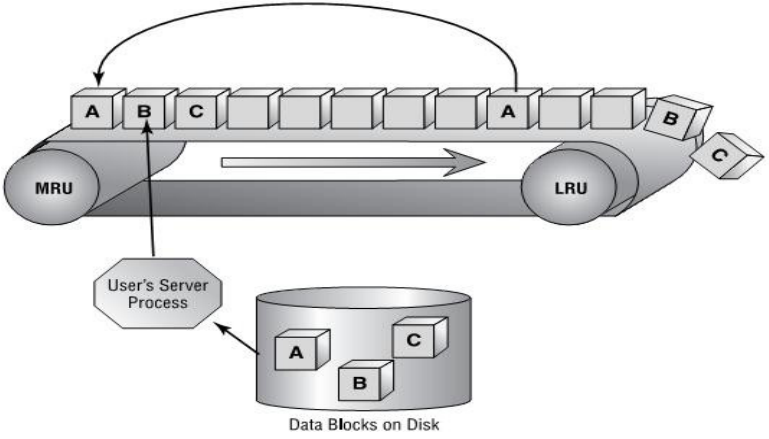
# Chap. 22 Beyond Physical Memory: Policies

## Demand paging (a.k.a. demand loading)

- ✓ Make mapping information without actual loading (fast execution)
- ✓ Start running ? occur page faults ? loading in a lazy manner ( we can load pages that are actually used)
- ✓ Life is easy where there are a lot of free frames

## When little memory is free

- ✓ Memory pressure forces OS for paging out to make room
- ✓ Replacement policy: decide which page (or pages) to evict



# 22.1 Cache Management

## Goal

- ✓ Maximize cache hit (minimize cache miss)

## Model

- ✓ Average memory access time (AMAT)

$$AMAT = (P_{Hit} \cdot T_M) + (P_{Miss} \cdot T_D)$$

Where

- TD: disk access latency
- Phit: probability of finding data in the cache (Pmiss = 1 – Phit)

Example

- Assume that TM= 100ns, TD= 10ms (10,000,000ns)
- Phit= 50% ? AMAT = 0.5 x 100 + 0.5 x 10,000,000 = 5,000,050 = 5ms
- Phit= 90% ? AMAT = 0.9 x 100 + 0.1 x 10,000,000 = 1,000,090 = 1ms
- Phit= 99% ? AMAT = 0.99 x 100 + 0.01 x 10,000,000 = 100,099 = 0.1ms

Hit ratio is quite important

- Expected hit ratio = SM /SD if an access pattern is the uniform distribution
- Remember locality which makes it feasible to obtain high hit ratio

# 22.2 Optimal Replacement Policy

## Optimal replacement policy (known as MIN)

- ✓ Evict a page that will be accessed furthest in the future
  - Best replacement policy
  - Not implementable (comparison purpose, quite useful)
- ✓ Example
  - Reference string: 0 1 2 0 1 3 0 3 1 2 1
  - Cache size: 3 frames
  - Hit ratio = 6/11 = 54.5%
  - Compulsory miss (Cold-start miss), Capacity miss, Conflict miss (Direct mapping or set-associative case)

| Access | Hit/Miss? | Evict | Resulting Cache State |
|---|---|---|---|
| 0 | Miss | | 0 |
| 1 | Miss | | 0, 1 |
| 2 | Miss | | 0, 1, 2 |
| 0 | Hit | | 0, 1, 2 |
| 1 | Hit | | 0, 1, 2 |
| 3 | Miss | 2 | 0, 1, 3 |
| 0 | Hit | | 0, 1, 3 |
| 3 | Hit | | 0, 1, 3 |
| 1 | Hit | | 0, 1, 3 |
| 2 | Miss | 3 | 0, 1, 2 |
| 1 | Hit | | 0, 1, 2 |

Figure 22.1: Tracing The Optimal Policy

# 22.3 A Simple Policy: FIFO

## FIFO (First In First Out)

- ✓ Evict a page that was brought into memory for the first time
  Like the FCFS scheduling policy (first-in page in a queue)
- ✓ Same example with same reference string (0 1 2 0 1 3 0 3 1 2 1)
  hit ratio = 4/11 = 36.4%

| Access | Hit/Miss? | Evict | Resulting Cache State |
|---|---|---|---|
| 0 | Miss | | First-in→ 0 |
| 1 | Miss | | First-in→ 0, 1 |
| 2 | Miss | | First-in→ 0, 1, 2 |
| 0 | Hit | | First-in→ 0, 1, 2 |
| 1 | Hit | | First-in→ 0, 1, 2 |
| 3 | Miss | 0 | First-in→ 1, 2, 3 |
| 0 | Miss | 1 | First-in→ 2, 3, 0 |
| 3 | Hit | | First-in→ 2, 3, 0 |
| 1 | Miss | 2 | First-in→ 3, 0, 1 |
| 2 | Miss | 3 | First-in→ 0, 1, 2 |
| 1 | Hit | | First-in→ 0, 1, 2 |

Figure 22.2: Tracing The FIFO Policy

- ✓ Pros) :
- ✓ Cons) ess hit ratio with la
  Anom

# 22.4 Another Simple Policy: Random

## Random

- ✓ Evict a page chosen randomly
- ✓ Example: same reference string (0 1 2 0 1 3 0 3 1 2 1)
  - hit ratio = 5/11 = 45.4% ? Figure 22.3
  - Different at each trial ? Figure 22.4
- ✓ Pros) Simple
- ✓ Cons) Not considering locality, unpredictable

| Access | Hit/Miss? | Evict | Resulting Cache State |
|---|---|---|---|
| 0 | Miss | | 0 |
| 1 | Miss | | 0, 1 |
| 2 | Miss | | 0, 1, 2 |
| 0 | Hit | | 0, 1, 2 |
| 1 | Hit | | 0, 1, 2 |
| 3 | Miss | 0 | 1, 2, 3 |
| 0 | Miss | 1 | 2, 3, 0 |
| 3 | Hit | | 2, 3, 0 |
| 1 | Miss | 3 | 2, 0, 1 |
| 2 | Hit | | 2, 0, 1 |
| 1 | Hit | | 2, 0, 1 |

Figure 22.3: **Tracing The Random Policy**

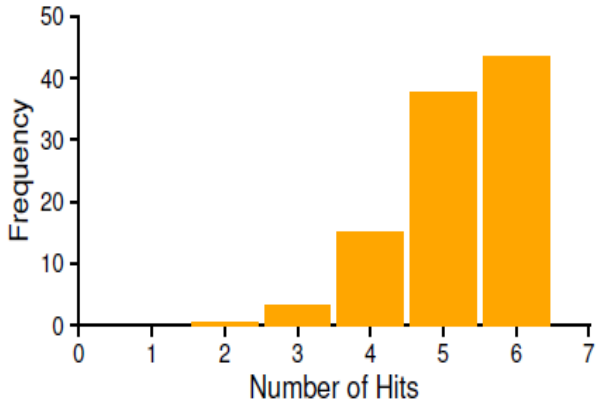



Figure 22.4: **Random Performance Over 10,000 Trials**

# 22.5 Using History: LRU

## LRU (Least Recently Used)

- Evict a page that was accessed oldest in the past
- Example: same reference string (0 1 2 0 1 3 0 3 1 2 1)
  hit ratio = 6/11 = 54.5%
- Pros) Considering locality (temporal locality)
- Cons) Not good for the looping reference

- History
  Use his
  LRU, LF

| Access | Hit/Miss? | Evict | Resulting Cache State |
|---|---|---|---|
| 0 | Miss | | LRU→ 0 |
| 1 | Miss | | LRU→ 0, 1 |
| 2 | Miss | | LRU→ 0, 1, 2 |
| 0 | Hit | | LRU→ 1, 2, 0 |
| 1 | Hit | | LRU→ 2, 0, 1 |
| 3 | Miss | 2 | LRU→ 0, 1, 3 |
| 0 | Hit | | LRU→ 1, 3, 0 |
| 3 | Hit | | LRU→ 1, 0, 3 |
| 1 | Hit | | LRU→ 0, 3, 1 |
| 2 | Miss | 0 | LRU→ 3, 1, 2 |
| 1 | Hit | | LRU→ 3, 2, 1 |

Figure 22.5: Tracing The LRU Policy

# 22.6 Workload Examples

## Workload analysis

- ✓ Workload: amount of work, characteristics of references in this case
- ✓ 3 types in this slide
  - No-locality: LRU == FIFO == RAND
  - 80-20 workload (hot/cold): LRU > FIFO == RAND
  - Loop workload: LRU == FIFO < RAND
- ✓ Most applications show strong locality ? LRU employed popularly
- ✓ Large cache size: close to optimal

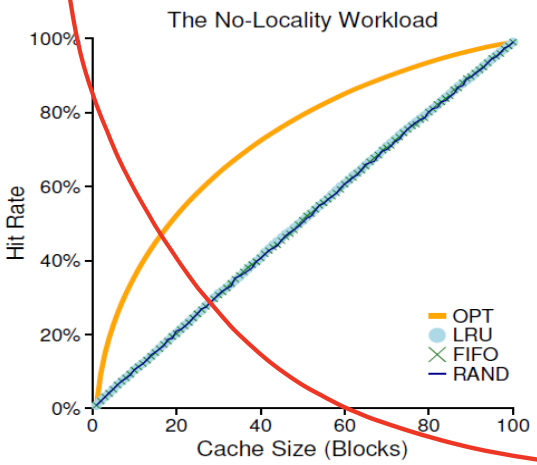



Figure 22.6: The No-Locality Workload

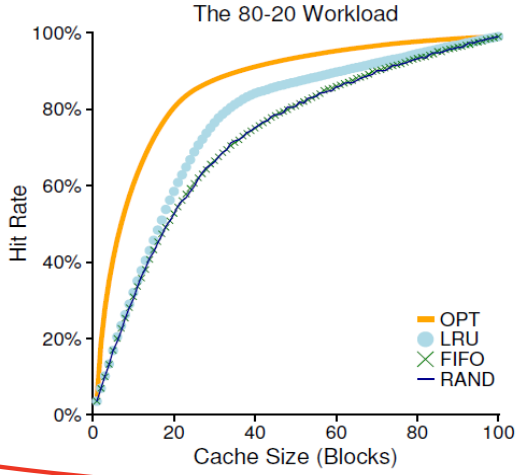



Figure 22.7: The 80-20 Workload

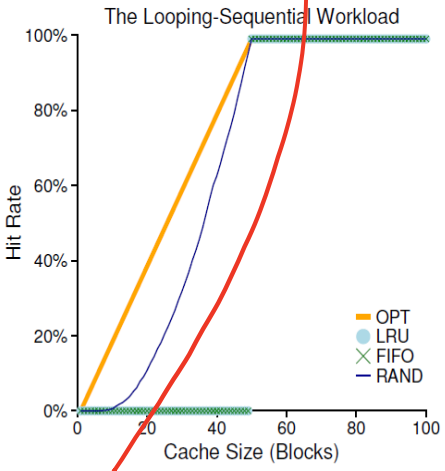
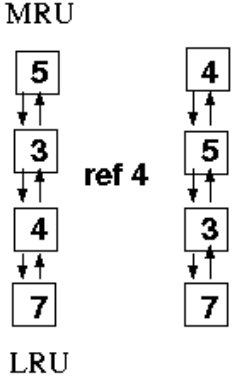



Figure 22.8: The Looping Workload

# 22.7 Implementing Historical Algorithms

## How to implement LRU?

- ✓ Usually linked list
- ✓ Pages access
  - Insert it to the head of the list (MRU position)
  - Move down all pages to the next position
  - Remove the page in the LRU position if necessary (miss case)
- ✓ Need to monitor all memory accesses
  - Feasible in the file cache or server cache
  - May degrade performance in the memory cache ? utilize HW supports such as reference bit and dirty bit



# 22.8 Approximating LRU

## Clock algorithm

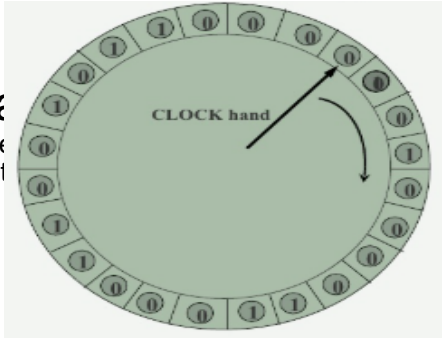
- ✓ FIFO with reference bit (refer to PTE in 13 page)

  HW: set reference bit as 1 when an associated page is accessed
  OS: manage a pointer for next victim
  - if (ref_bit == 1), clear it to 0 and give second chance (check the next page)
  - if (ref_bit == 0), evict it and move the victim pointer to the next page

  Approximate LRU well

- ✓ Adva

  Pe
  Ut ty bit



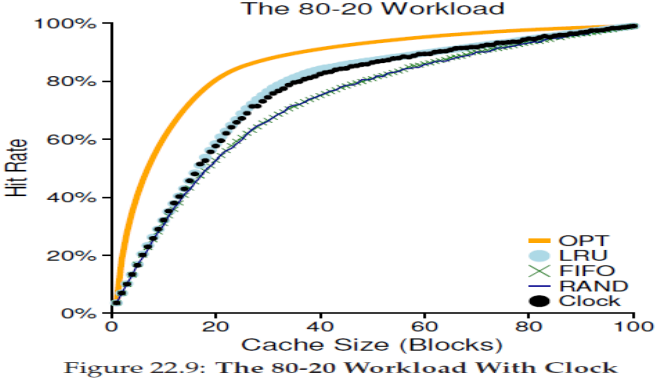
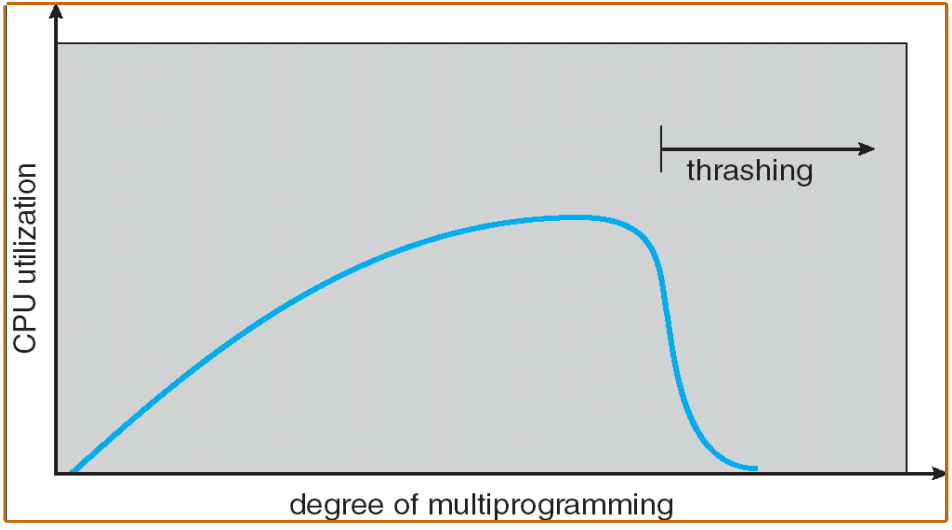




Figure 22.9: The 80-20 Workload With Clock

# 22.11 Thrashing

## Thrashing

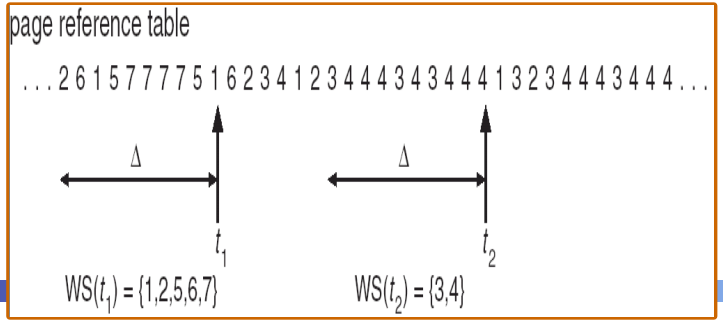
- ✓ A situation where the page fault rate is extremely high as each process does not have enough frames
  - A page fault triggers to replace a page that will be referenced soon, which eventually making another page fault immediately
- ✓ A process is spending more time paging than executing



# 22.11 Thrashing

## Working set

- WS(t): a set of pages referenced between t-? and t
  - To estimate how much memory a process needs
- Application of working set
  - Detect thrashing or Find a chance for new process initiation
  - Mechanism: *D* > *m* ? Thrashing
    - *WSSi* : Working Set Size of Process *Pi*
    - *D* = ? *WSSi* (D: the total demand of frames for all process)
    - m: total # of available frames in a system
  - Working-set Strategy
    - If (D > m), suspend some of the processes
    - If (D < m), another process can be initiated
    - ? Prevent thrashing while keeping multiprogramming degree as high as possible.



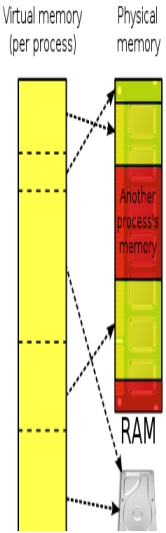
# Summary

## Virtual memory concept

- ✓ Separation of virtual memory from physical memory
  - Virtual memory: user's (or programmer's) viewpoint, exclusive
  - Physical memory: system's viewpoint, shared by multiple processes
- ✓ Allow the execution of a process that are not completely in memory
  - Logical address space can therefore be much larger than physical address space.
  - Allows more programs running

## Virtual memory can be implemented via:

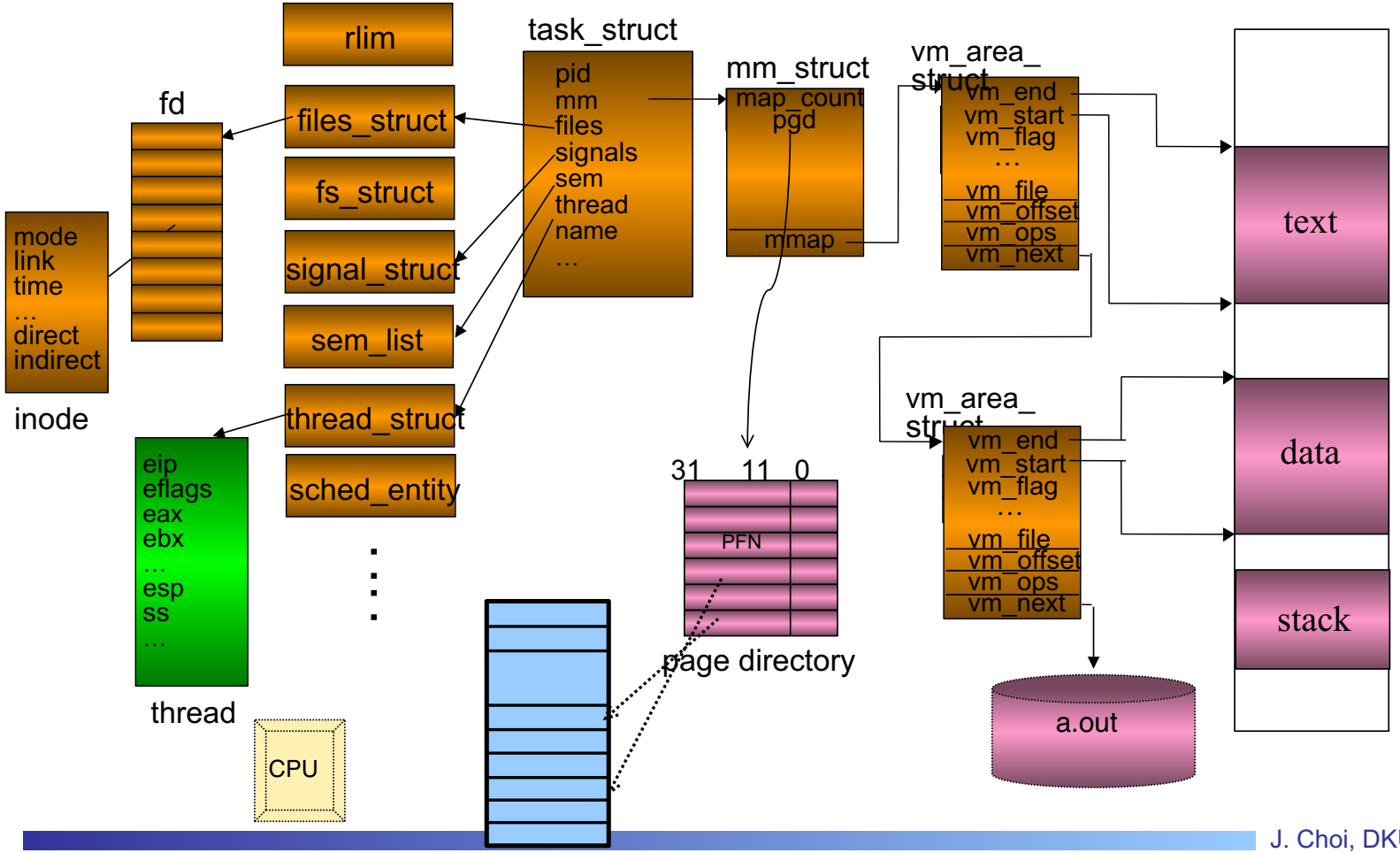
- ✓ Address translation, Demand paging and Replacement




- Lab 4: Make a LRU simulator (see 42 and 44 page)
  - How to submit? 1) report (Introduction, Description (data structure/function), Results (at least two outputs), Discussion (each role at your team)) ? email to professor, 2) Source code and report ? email to TA
  - Environment ? See Lab. 0 in the lecture site
  - Due: until the same day of the next week (6PM. June 24th).
  - Bonus: Analysis with different cache size and workload (No locality, locality, loop: Page 43 )
- Simple question to take attendance: 1) Calculate the hit ratio under the FIFO, LRU and OPT policies when there are four frames and the page reference string is "1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5", 2) What is the Belady's anomaly? Why LRU does not suffer from this problem? ( hint: See 5 page in Chapter 22 of OSTEP) (until 6 PM, June 18th).

# Appendix 1

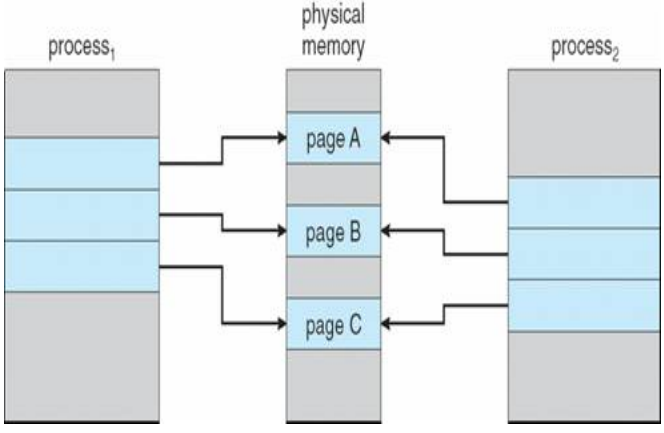
## Linux implementation for segmentation and paging



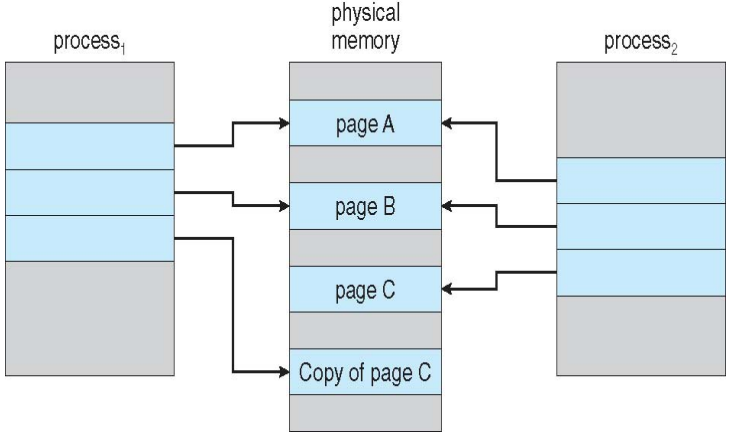
# Appendix 1

## Copy-on-Write (COW)

- ✓ Allow both processes can share pages even though they are not actually shared pages (set the copy-on-write bit in page table)
- ✓ If either process modifies a shared page, the page is copied
- ✓ Good for fork() and exec()
  - Allow both parent and child processes to initially *share* the same pages in memory
  - More efficient for process creation




(Before process 1 modified page C)




(After process 1 modified page C)

# Appendix 2

## 19.6 TLB Issue: Replacement Policy

- ✓ TLB: caching only recently accessed VPN-PFN pairs
  - TLB is small while PT is relatively large
- ✓ When TLB is full and we need to insert a new VPN-PFN pair, which existing pair can be kicked out?
  - Goal: least affect on TLB hit ratio
- ✓ Policies
  - FIFO (First In First Out)
  - LRU (Least Recently Used)
  - LFU (Least Frequently Used)
  - All are further discussed in later chapters

THE CRUX: HOW TO DESIGN TLB REPLACEMENT POLICY
Which TLB entry should be replaced when we add a new TLB entry? The goal, of course, being to minimize the **miss rate** (or increase **hit rate**) and thus improve performance.

# 19.7 A Real TLB Entry

- ✓ Real TLB example: MIPS R4000

SW-managed TLB, 4KB page/frame size
32 or 64 entries in TLB (related to TLB coverage)
Bit description

- VPN: 19-bits ? we expect 20-bits. But, user addresses will only come from half of the address space (2GB for user, 2GB for kernel) ? 19-bits are enough
- PFN: 24-bits ? support up to 64GB physical memory (224 x 4KB)
- ASID: 8-bits, to identify which process own the VNP-PFN pair
- G: global bit ? shared among processes (ASID is ignored)
- C: coherence bit ? for coherence protocol
- D: dirty bit
- V: valid bit
- Page mask: for supporting multiple page sizes

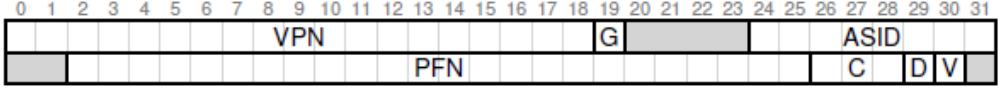


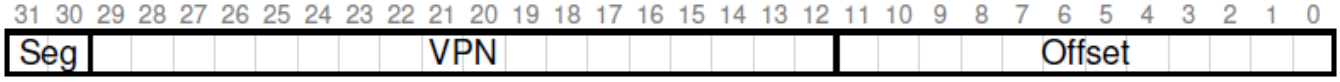

Figure 19.4: A MIPS TLB Entry

# Appendix 2

## 20.2 Hybrid Approach: Paging and Segments

- ✓ Real system example: 32-bit address space, 4KB page size, address space consists of four segments

  Address format: Seg # + VPN + Offset




The limit register holds the maximum valid pages ? access above the limit generates the segmentation fault ? can reduce the PT size

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

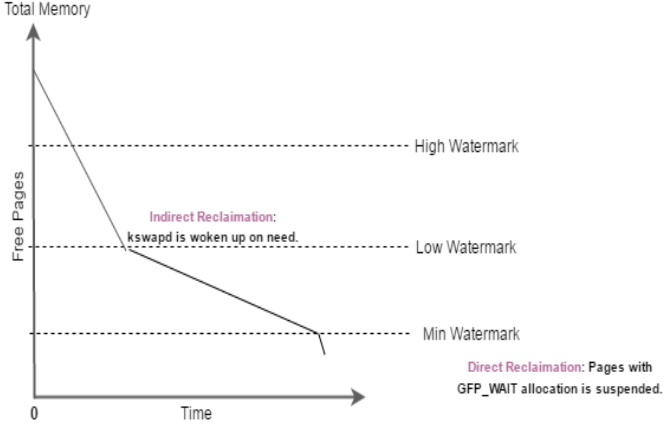
# Appendix 2

## 21.6 When Replacements Really Occur

- ✓ OS actually prepares free memory proactively, not wait until memory is full
  - Background thread: swap daemon or page daemon
  - When free memory is below the low watermark, it begins to evict pages until free memory becomes above the high watermark
- ✓ Group a number of pages and write them out at once (to make sequential writes for better performance)

## Summary of Swapping

- ✓ Larger than physical memory
  - Present bit in PT
  - Page fault handler in OS
- ✓ Transparent to user
  - Sometimes not (stuck)



# Appendix 2

## 22.9 Considering Dirty Pages/22.10 Other VM Policies

- ✓ Dirty bit
  - To indicate whether a page is modified or not
  - Used for delayed write
  - Can be exploited for replacement
    - E.g.) (R, D) pairs
    - Eviction preference: (0,0) > (0,1) > (1,0) > (1,1) ? differentiate un-referenced pages (or referenced pages) using dirty bit
- ✓ Other VM policies
  - When to bring a page into memory?
    - All loading: large loading overhead
    - Demand paging: loading when it is really accessed (less loading time, save memory)
    - Prefetching: exploit spatial locality
    - cf) the latency for reading two blocks from disk is similar to that for one block
  - Write policy
    - Try to group dirty pages and write them in a batch manner (clustering)

# Chap. 24 Summary Dialogue on Memory Virtualization

**Student:** *(Gulps) Wow, that was a lot of material.*

**Professor:** *Yes, and?*

**Student:** *Well, how am I supposed to remember it all? You know, for the exam?*

**Professor:** *Goodness, I hope that's not why you are trying to remember it.*

**Student:** *Why should I then?*

**Professor:** *Come on, I thought you knew better. You're trying to learn something here, so that when you go off into the world, you'll understand how systems actually work.*

**Student:** *Hmm... can you give an example?*

**Professor:** *Sure! One time back in graduate school, my friends and I were measuring how long memory accesses took, and once in a while the numbers were way higher than we expected; we thought all the data was fitting nicely into the second-level hardware cache, you see, and thus should have been really fast to access.*

**Student:** *(nods)*

**Professor:** *We couldn't figure out what was going on. So what do you do in such a case? Easy, ask a professor! So we went and asked one of our professors, who looked at the graph we had produced, and simply said "TLB". Aha! Of course, TLB misses! Why didn't we think of that? Having a good model of how virtual memory works helps diagnose all sorts of interesting performance problems.*