

Operating Systems

3. Memory Virtualizing



Pablo Prieto Torralbo

DEPARTMENT OF COMPUTER ENGINEERING
AND ELECTRONICS

This material is published under:

[Creative Commons BY-NC-SA 4.0](https://creativecommons.org/licenses/by-nc-sa/4.0/)

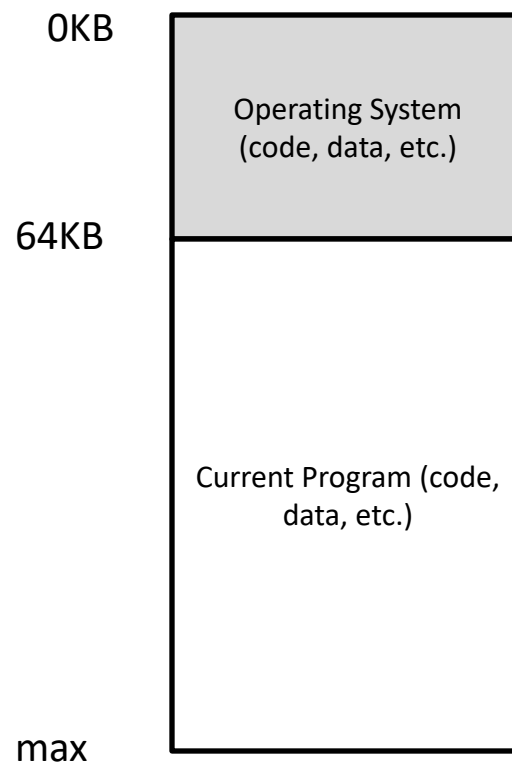


3.1 Memory Virtualization -Address Space

Early Days

- ▶ No memory abstraction.
- ▶ OS as a set of routines (standard library).
 - At address 0 in the example.
- ▶ One running program uses the rest of the memory.

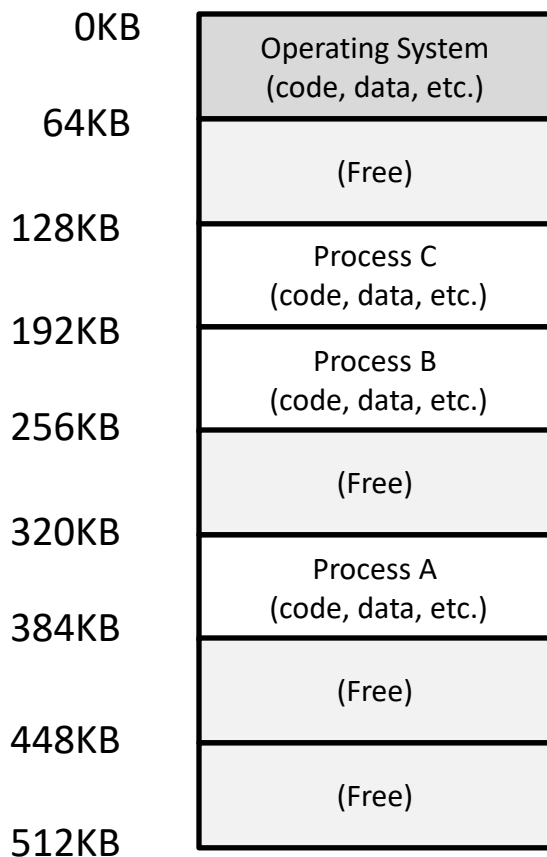
- ▶ Physical memory →



Multiprogramming

- ▶ CPU Virtualization was the illusion of having a private CPU per process.
- ▶ Memory Virtualization is the illusion of having a private memory per process.
- ▶ CPU: Multiple processes at a time.
 - OS switches between them saving the state of the CPU (Time sharing).
- ▶ Memory: Allow a program to use all memory and save it ~~on~~ disk when switching.
 - Too slow.
- ▶ Leave processes in memory and switch between them.

Multiprogramming



- ▶ Each process has a portion of physical memory.
- ▶ Multiple programs reside concurrently in memory.
- ▶ Processes need Protection

How does a process know where it is in the memory?

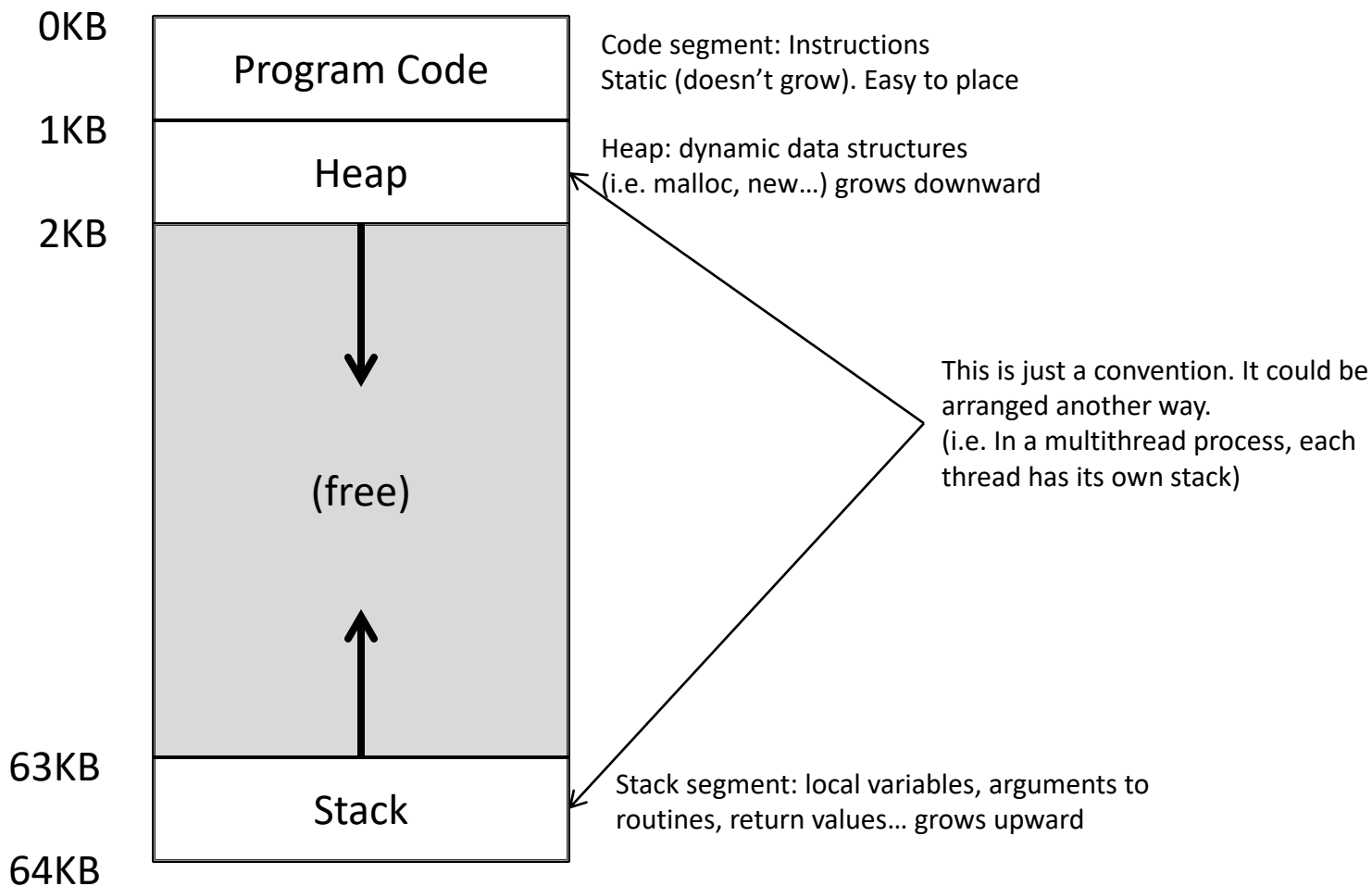
Address Space

- ▶ Easy to use abstraction of physical memory.
 - Illusion of private memory.
 - Virtual Address.
 - Extend LDE (Limited Direct Execution).

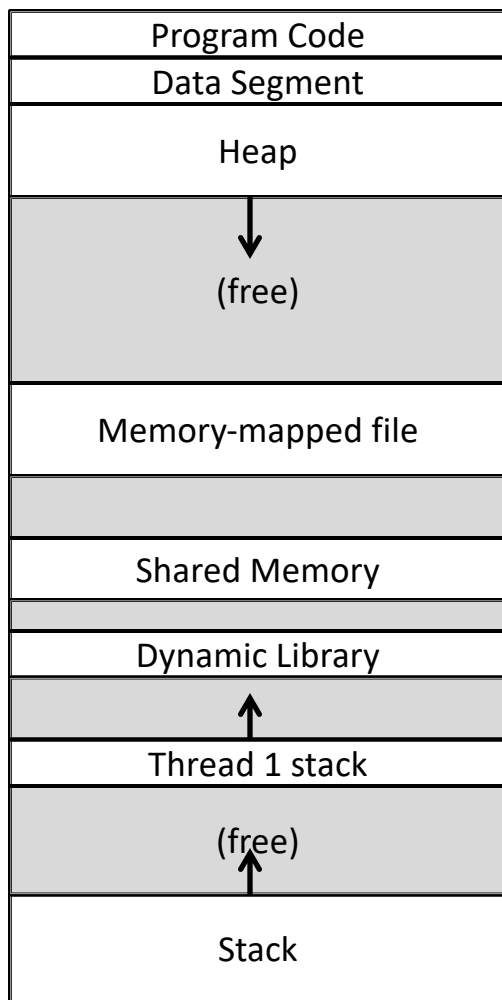
- ▶ Contains memory state of a program:
 - Code (instructions, static data).
 - Heap (data, dynamically-allocated memory).
 - Stack (local variables and routines parameters).

In OSTEP

Address Space



Address Space



```
int x=25;
```

```
int main(int argc, char *argv[]) {
    int y;
    int *z = malloc(sizeof(int));
    ...
}
```

x

main

y

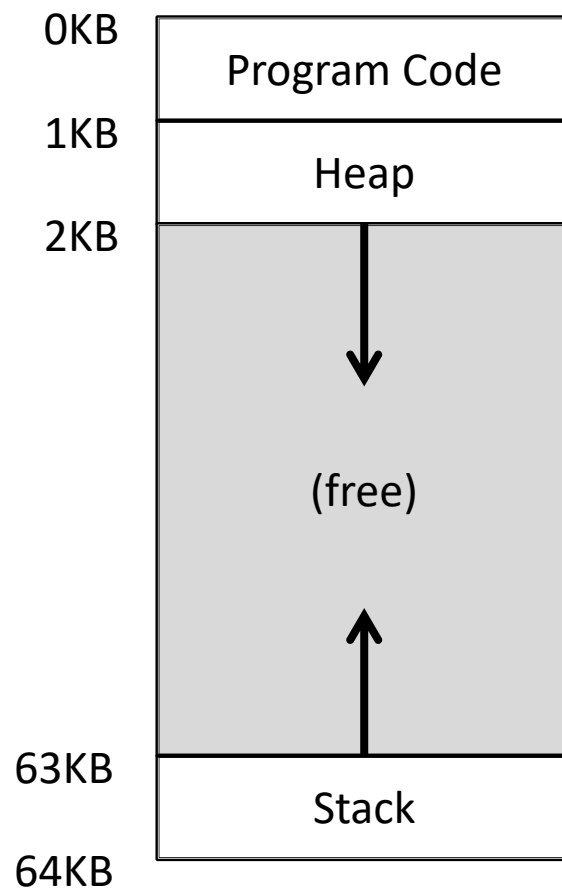
z

pointer (address) on the stack
int on the heap

Segments Arrangement may vary.

Address Space

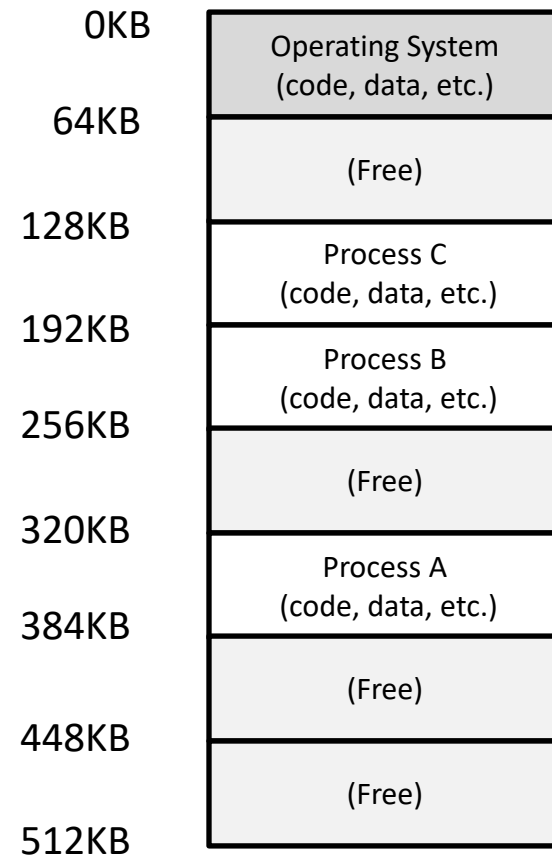
- ▶ Abstraction provided by the OS.
 - Program is not in physical addresses 0 through 64KB.
 - Loaded in an arbitrary physical address by the OS (In previous example, Process A is in address 320KB).



Address Space

- ▶ Abstraction provided by the OS.
 - Program is not in physical addresses 0 through 64KB.
 - Loaded in an arbitrary physical address by the OS (In previous example, Process A is in address 320KB).
 - When a process performs an access to an address from its address space (i.e. 1KB, virtual address) → OS with hardware support should load correct physical address (i.e. 321KB for process A).

- ▶ This is the key to virtualization of memory.



How does a process know where it is?

Processes don't know where they actually are

Address Space: Goals

- ▶ Transparency: invisible to the running programs.
 - Programs do not realize memory is virtualized.

- ▶ Efficiency:
 - Time efficient → programs do not run much more slowly.
 - Space efficient → not too much memory is used for supporting virtualization.

- ▶ Protection: to protect processes from one another and the OS itself from processes.
 - A process should not be able to access memory outside its address space.
 - Isolation among processes → one can fail without affecting others. Prevents harm.
 - Some modern Operating Systems isolate pieces of the OS from other pieces of the OS providing great reliability.

3.2 Memory Virtualization -Address Translation

Address Translation

- ▶ Virtualization of the CPU uses Limited Direct Execution:
 - Allow the program run directly on hardware.
 - Make the OS control critical points.

- ▶ Two main goals: Efficiency + Control.

- ▶ How can we efficiently and flexibly virtualize memory while ensuring protection (control)?
 - Hardware support → address translation.
 - Transform each virtual address used by program instructions to physical address where information is actually located.
 - OS manage memory, keeping track of which locations are used/free.

Address Translation

- ▶ Starting Assumptions:
 - User's address space is placed contiguously in physical memory.
 - Size of address space is less than the size of physical memory.
 - Each address space is exactly the same size.

- ▶ As with virtualized CPU, unrealistic, but we will relax these assumptions gradually.

Address Translation

▶ Example #1:

```
void func() {
    int x;
    x = x + 3; // line of code we are interested in
```

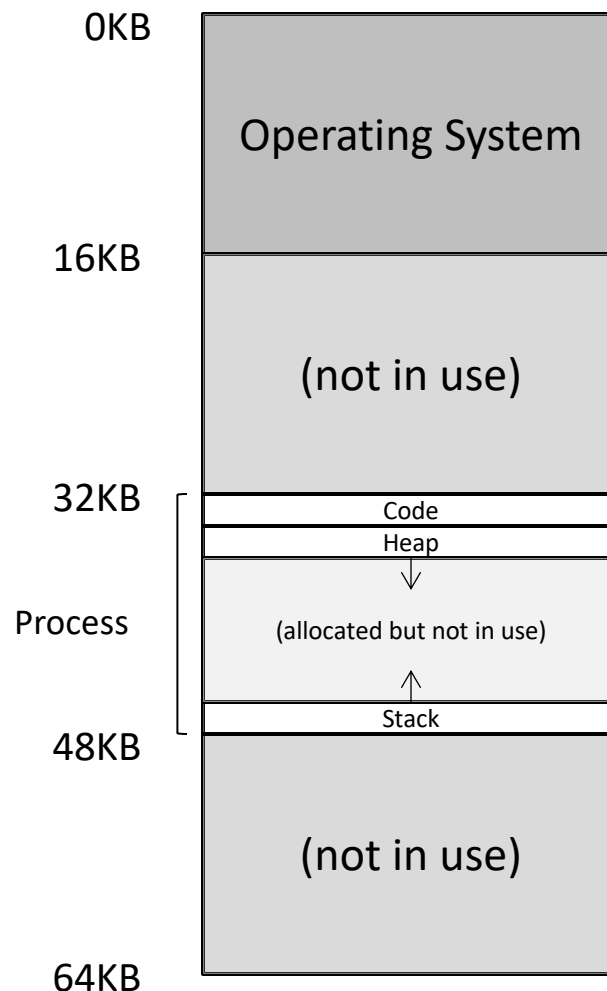


```
-----
128: movl 0x0(%ebx), %eax      ; load 0+ebx into eax
132: addl $0x03, %eax          ; add 3 to eax register value
136: movl %eax, 0x0(%ebx)      ; store eax back to mem
```

- ▶ Presuming address of x is in the register ebx.
 - Let's assume 0x3C00 (15KB, in the stack near the bottom on a 16KB address space).

Address Translation

- ▶ Instructions work with addresses from the program address space (virtual).
 - Addresses are “hardcoded” into program binaries.
- ▶ Physical memory looks like:



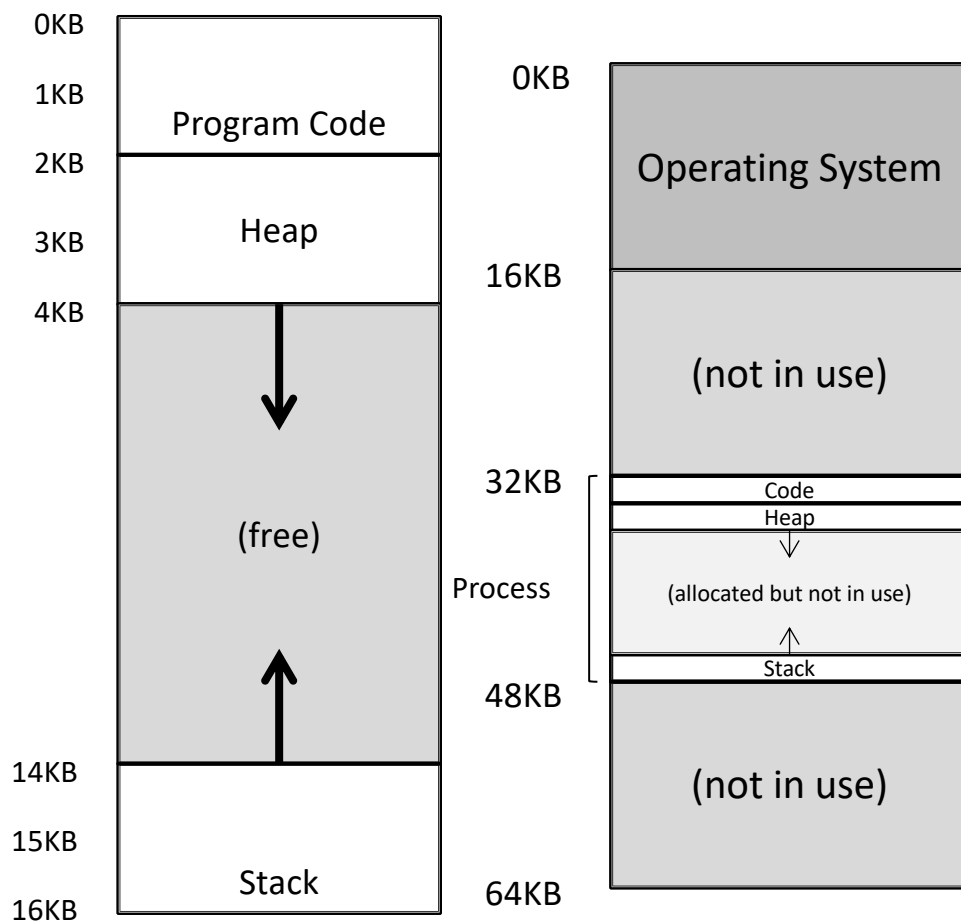
Dynamic Relocation

	Base	Size
Process	32KB (0x8000)	16KB

► Example #1

- Fetch instruction at addr 0x8128
- Exec load from addr 0xBC00
- Fetch instruction at addr 0x8132
- Exec, no load
- Fetch instruction at addr 0x8136
- Exec store to addr 0xBC00

0x128: movl 0x3C00, %eax
0x132: addl 0x3, %eax
0x136: movl %eax, 0x3C00



Dynamic Relocation

- ▶ Also referred to as base and bounds (two hardware registers):
 - Base register
 - Bounds/limit register

- ▶ As a process starts running, the OS decides where to load the address space in the physical memory → stores the value in base register.

- ▶ Bounds (limit) register has the size of the address space. → Protection.

- ▶ Address translation:

Physical address = virtual address + base

 - Memory reference should be within bounds.

Dynamic Relocation

- ▶ Example #2: A process with an address space of 16KB loaded at physical address 32KB.
 - Base register: 32KB (32768)
 - Bounds register: 16KB (16384)

Virtual Address	Physical Address
0	32KB
1KB	33KB
6000	38768
17100	Fault (out of bounds)

Dynamic Relocation

- ▶ The part of the processor that helps with address translation is called the Memory Management Unit (MMU)
 - Base/bounds registers.
 - Ability to translate virtual address and check if within bounds.
 - Privileged instructions to update base/bounds.
 - Privileged instructions to register exception handlers (“out of bounds” and “illegal instruction”).
 - Ability to raise exceptions.

Dynamic Relocation

- ▶ OS management:
 - Find space at process creation. Easy with our assumptions. Free list.
 - Reclaim all the process memory when it terminates. Back to the free list.
 - Save and restore base/bounds registers when a context switch occurs (PCB).
 - Provide exception handlers.

- ▶ It is possible for the OS to move an address space from one location in memory to another (when the process is not running).
 - memcpy

OS boot
(kernel mode)

Hardware

Program
(user mode)

Initialize trap table

Remember address of:
System call handler, Timer handler, Illegal
mem access handler, Illegal instruction
handler...

Start interrupt timer

Start timer. Interrupt after X ms.

Initialize process table
Initialize free list

OS run
(kernel mode)

Hardware

Program
(user mode)

To start process A:
allocate entry in process table
allocate memory for process
set base/bounds registers
 return from trap (into A)

Restore registers of A
 Move to user mode
 Jump to A (initial) PC

Process A runs
 Fetch instruction

Translate virtual address and perform fetch

Execute instruction

If explicit load/store:
Ensure address is in-bounds,
Translate virtual address and
perform load/store

...

...

...

Timer interrupt:
 move to kernel mode
 Jump to interrupt handler

Handle the trap:
 Call switch routine:
 save regs(A) to proc-struct(A)
(including base/bounds)
 restore regs(B) from proc-struct(B)
(including base/bounds)
 return from trap (into B)

Restore registers of B
 Move to user mode
 Jump to B's PC

Process B runs

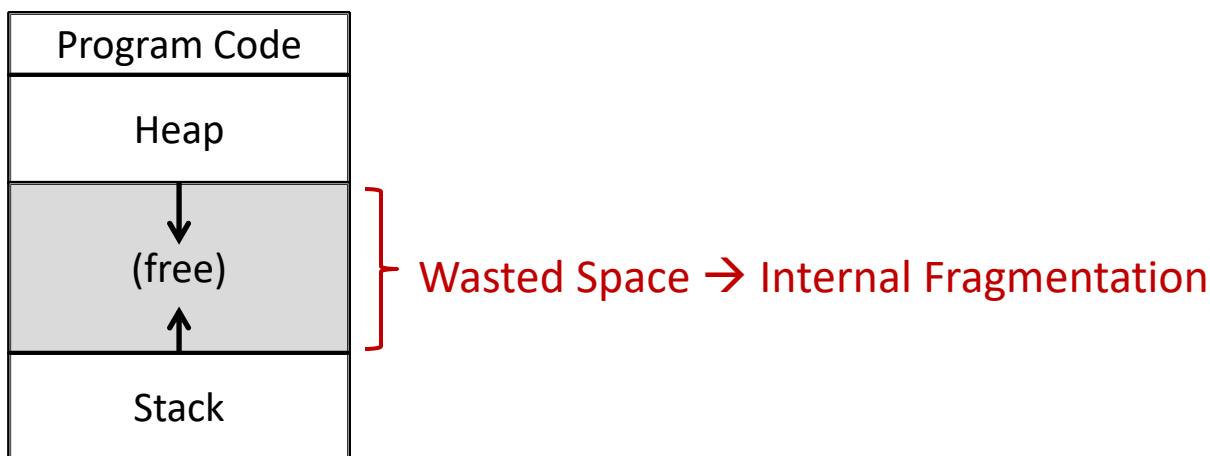
Dynamic Relocation

► Pros: ✓

- Fast and Simple.
- Offers protection.
- Little overhead (2 registers per process)

► Cons: ✗

- Not flexible.
- Wastes memory for large address spaces
 - Internal Fragmentation.



3.3 Memory Virtualization -Segmentation

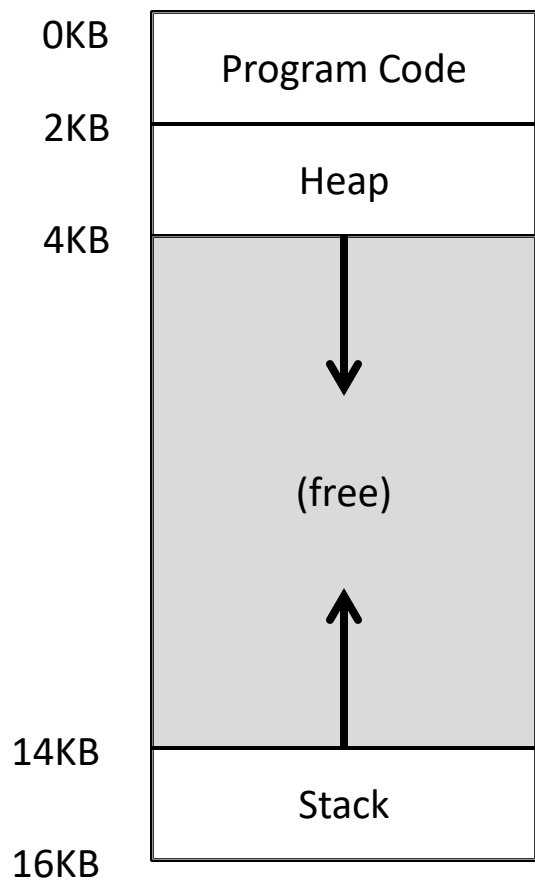
Segmentation

- ▶ Segmentation: instead of one base/bound register in MMU, one pair per logical segment of address space.
 - Avoid internal fragmentation.
 - Allow running programs with address spaces that don't fit entirely into memory.

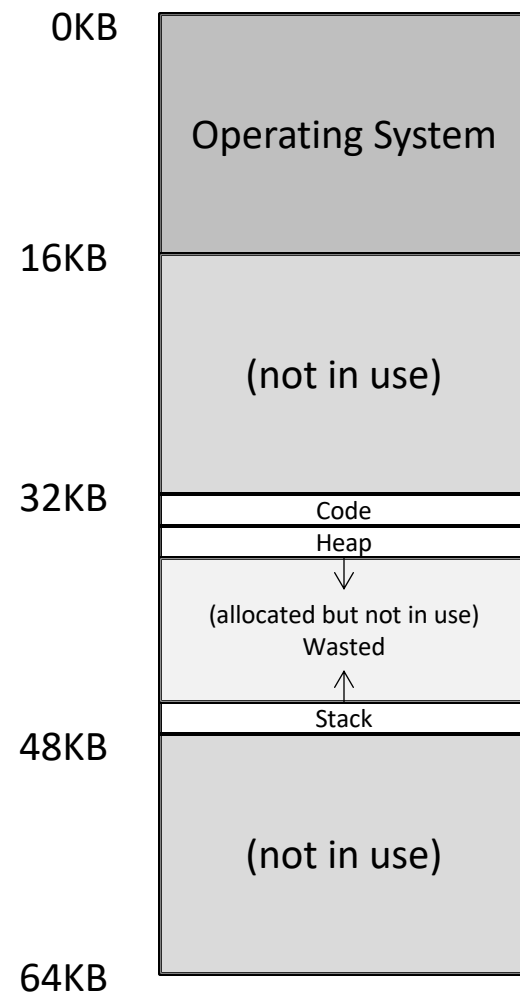
- ▶ A segment is a contiguous portion of the address space. In our simple address space:
 - Code segment
 - Heap segment
 - Stack segment

Segmentation

► Example #3: dynamic relocation

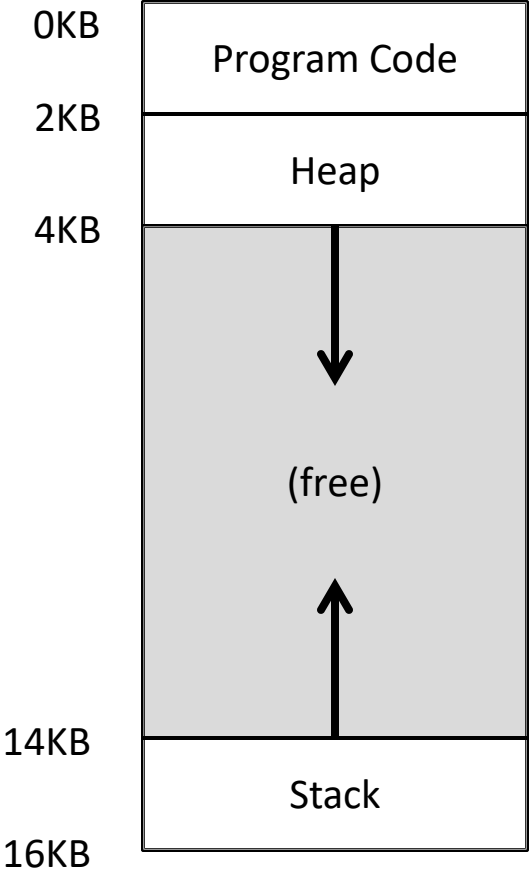


Base	Bound
32KB	16KB

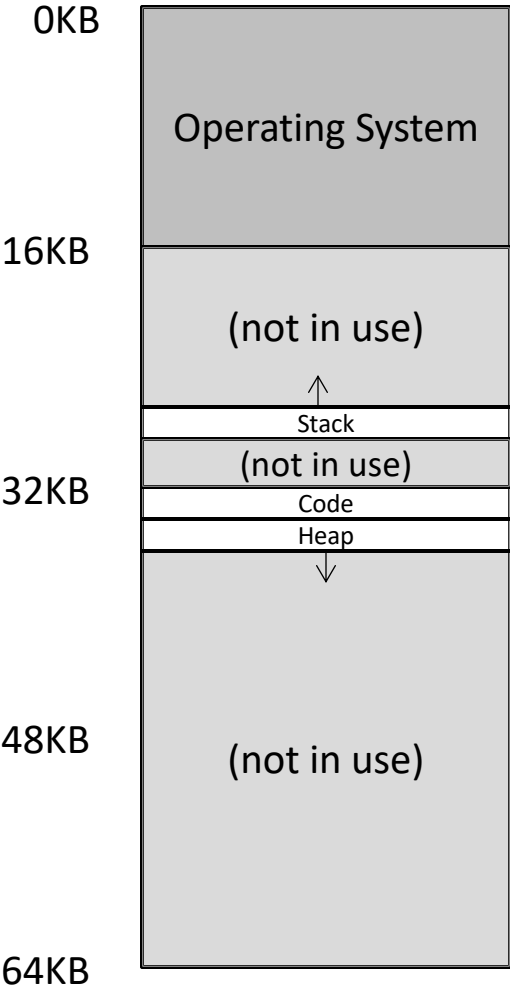


Segmentation

▶ Example #3: segmentation

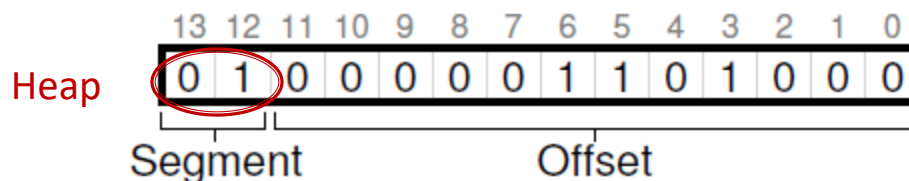


Segment	Base	Size (bounds)
Code	32KB	2KB
Heap	34KB	2KB
Stack	28KB	2KB



Segmentation

- ▶ Which segment does an address refer to?
 - Implicit approach:
 - The hardware determines the segment by noticing how the address was formed.
 - i.e. from the program counter (fetch) → code segment
 - from the stack pointer → stack segment.
 - Explicit approach:
 - Top few bits of virtual address (used in VAX/VMS)
 - i.e. In our example, three segments, we need 2 bits of the 14 bits address. So address 0x1068



Segment	Number
Code+data	0
Heap	1
Stack	2

Segmentation

- ▶ What about the stack? (grows backwards)

Segment	Base	Size	Grows +?
Code	32KB	2KB	1
Heap	34KB	2KB	1
Stack	28KB	2KB	0

- ▶ Example : Access to address 15KB (virtual):

- Stack access (negative growth)

Stack 11 1100 0000 0000 (hex 0x3C00)

Offset: 3KB → in Ca2 -1KB

- ▶ The offset is negative (Ca2). VA offset (3KB) minus max. segment size (4KB) equals real offset (-1KB).
 - In the example, physical address 27KB.

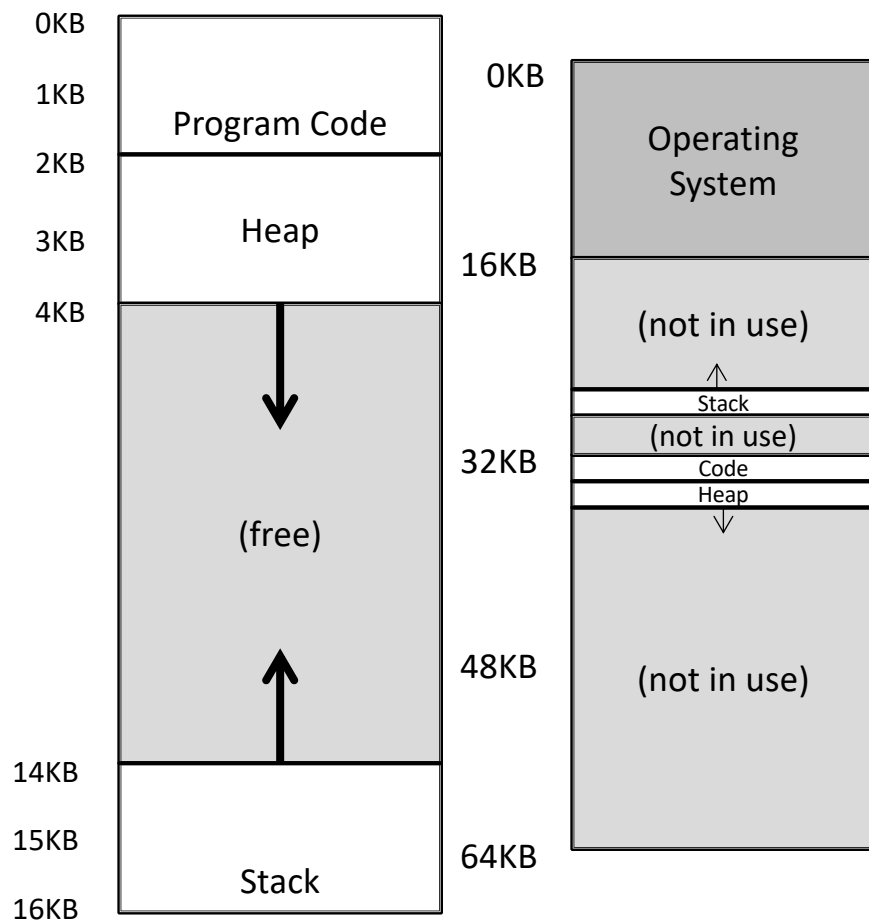
Address Translation

Segment	Base	Size	Grows +?
Code	32KB (0x8000)	2KB	1
Heap	34KB (0x8800)	2KB	1
Stack	28KB (0x7000)	2KB	0

► Example #4 with Segmentation

- Fetch instruction at addr 0x8128
- Exec load from addr 0x6C00
- Fetch instruction at addr 0x8132
- Exec, no load
- Fetch instruction at addr 0x8136
- Exec store to addr 0x6C00

0x128: movl 0x3C00, %eax
0x132: addl 0x3, %eax
0x136: movl %eax, 0x3C00



Segmentation

- ▶ Support for sharing → protection bits

Segment	Base	Size	Grows +?	Protection
Code	32KB	2KB	1	Read-Execute
Heap	34KB	2KB	1	Read-Write
Stack	28KB	2KB	0	Read-Write

Code sharing

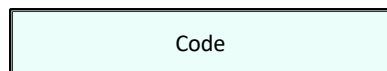
- ▶ Fine-grained vs. Coarse-grained Segmentation.
 - Large vs. small number of segments.
 - Flexibility vs. Cost.

Segmentation - Fragmentation

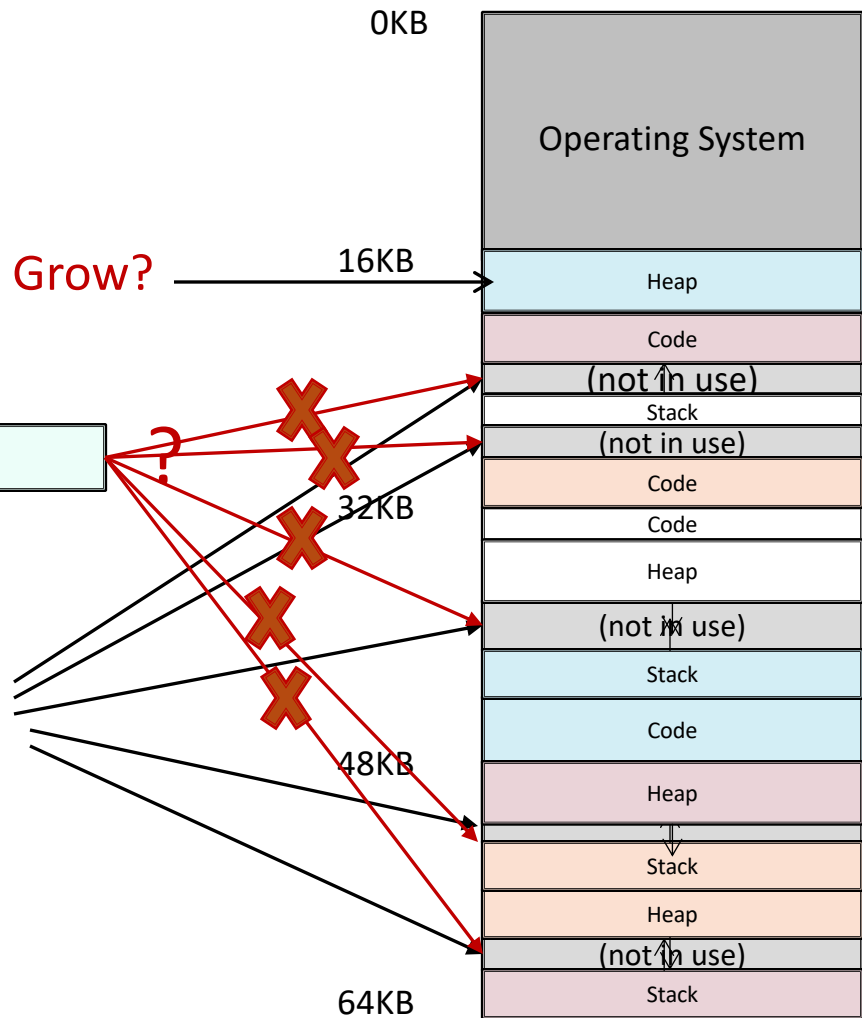
- System with 4 processes running



- A 5th process arrives



External Fragmentation!



Segmentation - Summary

► Pros:

- Easy and Fast.
- Supports sparse address space (no internal fragmentation).
- Allows sharing and fine-grained protection.
- Little overhead (few registers per process)

► Cons:

- External Fragmentation.
 - Free-list Management reduces it, but it still exists.
- Complex Free Space Management.
- Segment growing could mean memcpy.

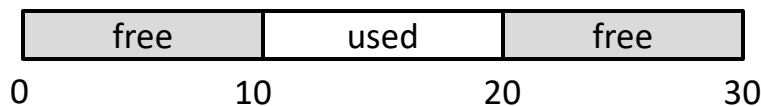
3.4 Memory Virtualization -Free Space Management

Free Space Management

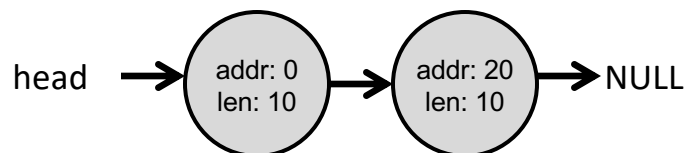
- ▶ Minimize external fragmentation (without compacting).
- ▶ Management of the free-list to keep track of free space.
- ▶ Basic mechanism: Splitting and Coalescing
 - OS with segmentation
 - User-level memory-allocation library → heap

Splitting and Coalescing

- ▶ Assume a 30-byte heap:

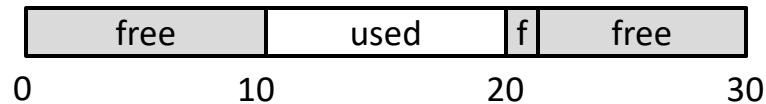


- ▶ Free list:

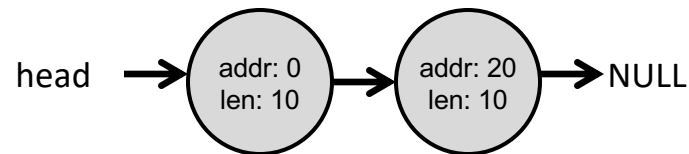


Splitting and Coalescing

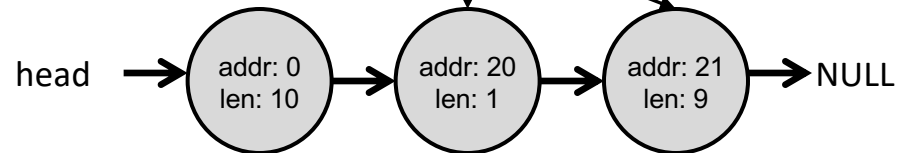
- Assume a 30-byte heap:



- Free list:

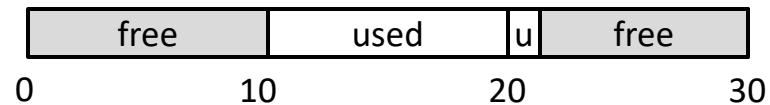


- Ask for 1 byte (splitting):

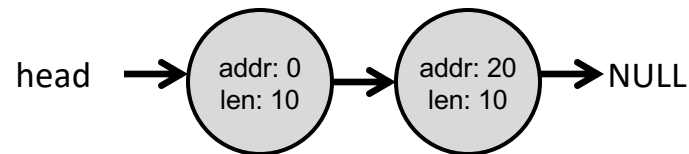


Splitting and Coalescing

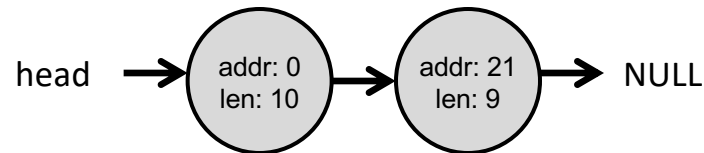
- Assume a 30-byte heap:



- Free list:

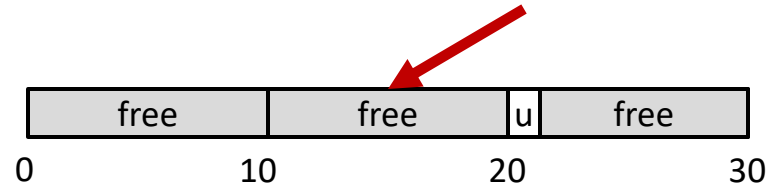


- Ask for 1 byte (splitting):

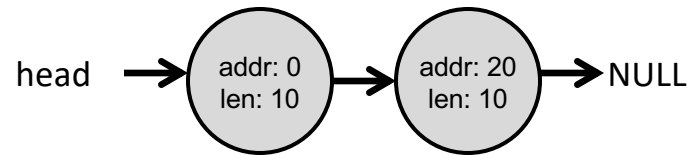


Splitting and Coalescing

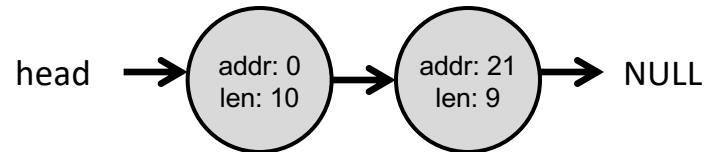
- Assume a 30-byte heap:



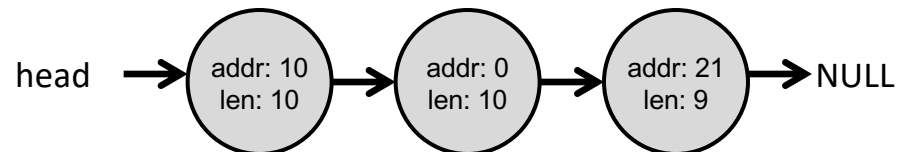
- Free list:



- Ask for 1 byte (splitting):

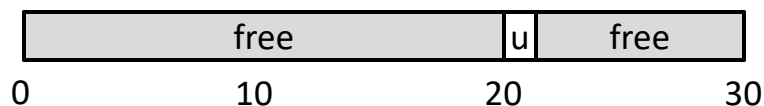


- Free:

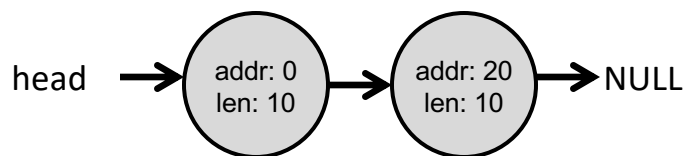


Splitting and Coalescing

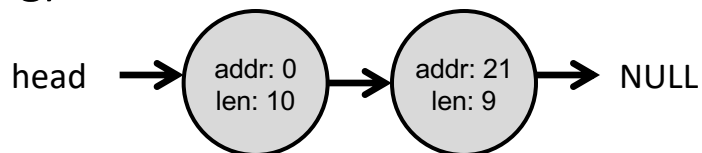
- Assume a 30-byte heap:



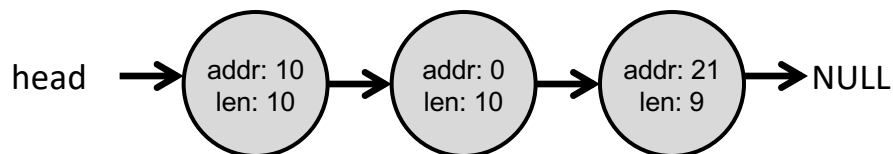
- Free list:



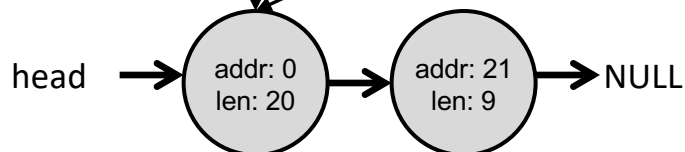
- Ask for 1 byte (splitting):



- Free:



- Coalescing:



Free Space Management

Basic Strategies

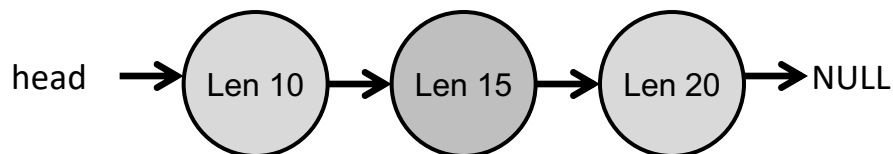
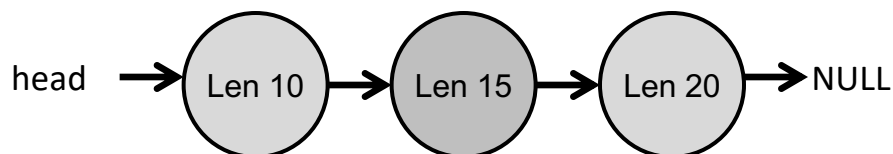
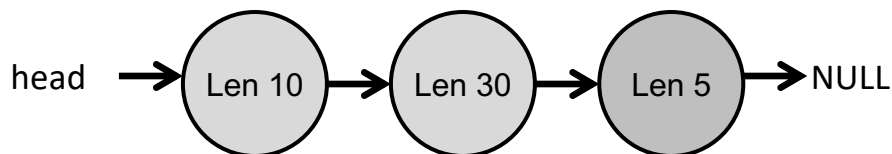
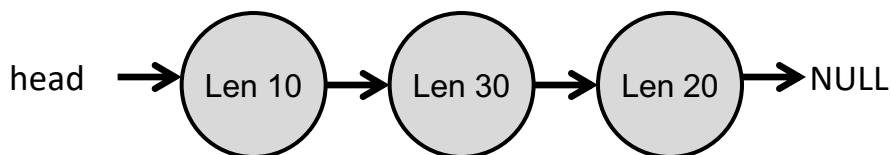
- Ask for 15

- Best Fit

- Worst Fit

- First Fit

- Next Fit

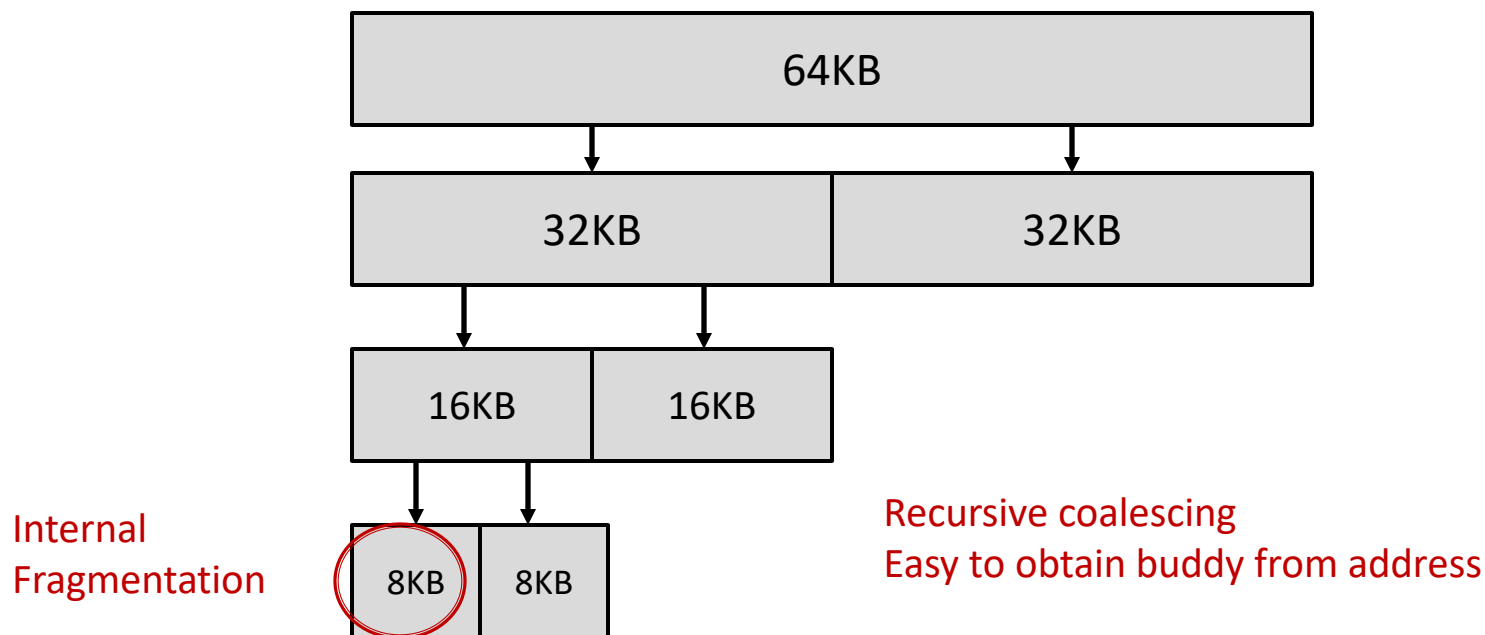


Similar to First Fit
No search. Spread allocates.

No search needed → Faster
Order becomes an issue.

Free Space Management

- ▶ Binary buddy allocator:
 - Free memory as a space of size 2^N .
 - i.e. request for a 7KB block.



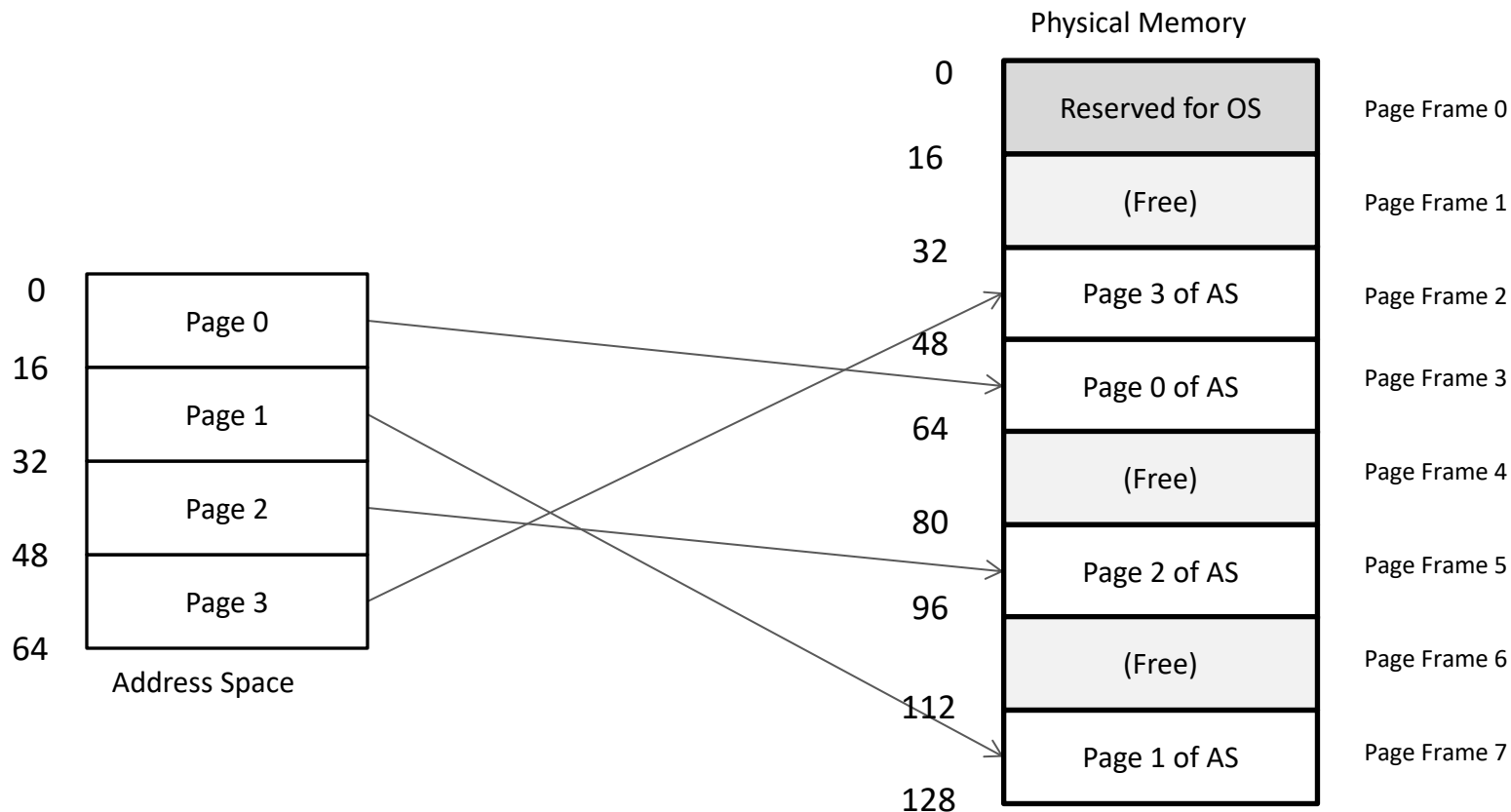
3.5 Memory Virtualization -Paging

Introduction

- ▶ Segmentation involves chopping up memory space into variable-sized pieces.
 - Is too coarse grained. Complex free space management.

- ▶ Paging chops up memory space into fixed-sized pieces.
 - We divide the address space into fixed-sized units called pages.
 - Correspondingly, the physical memory is viewed as an array of fixed-sized slots called page frames.
 - Each virtual page is independently mapped to a physical page.
 - More flexible and easier free-space management.

Introduction

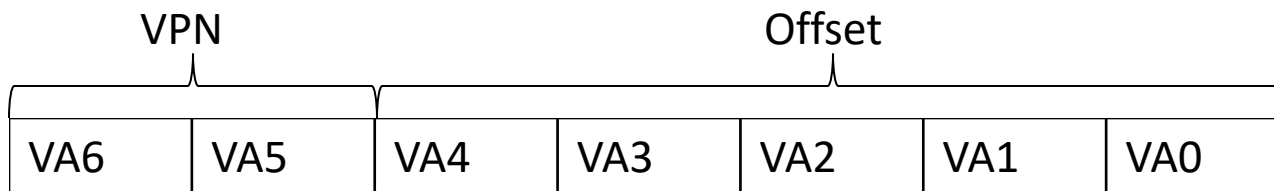


- ▶ What techniques do we need?
- ▶ How much space and overhead does it need?
- ▶ What is the correct page size?

Address Translation

- ▶ For segmentation:
 - high bits → segment.
 - low bits → offset.

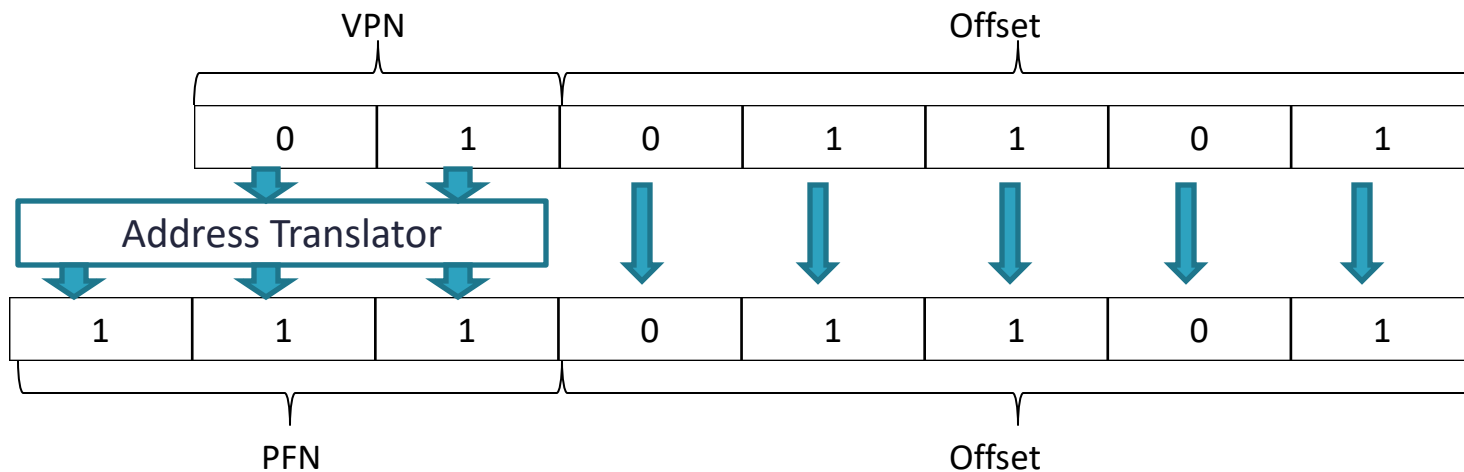
- ▶ For paging:
 - high bits → page.
 - low bits → offset.



How many bits?

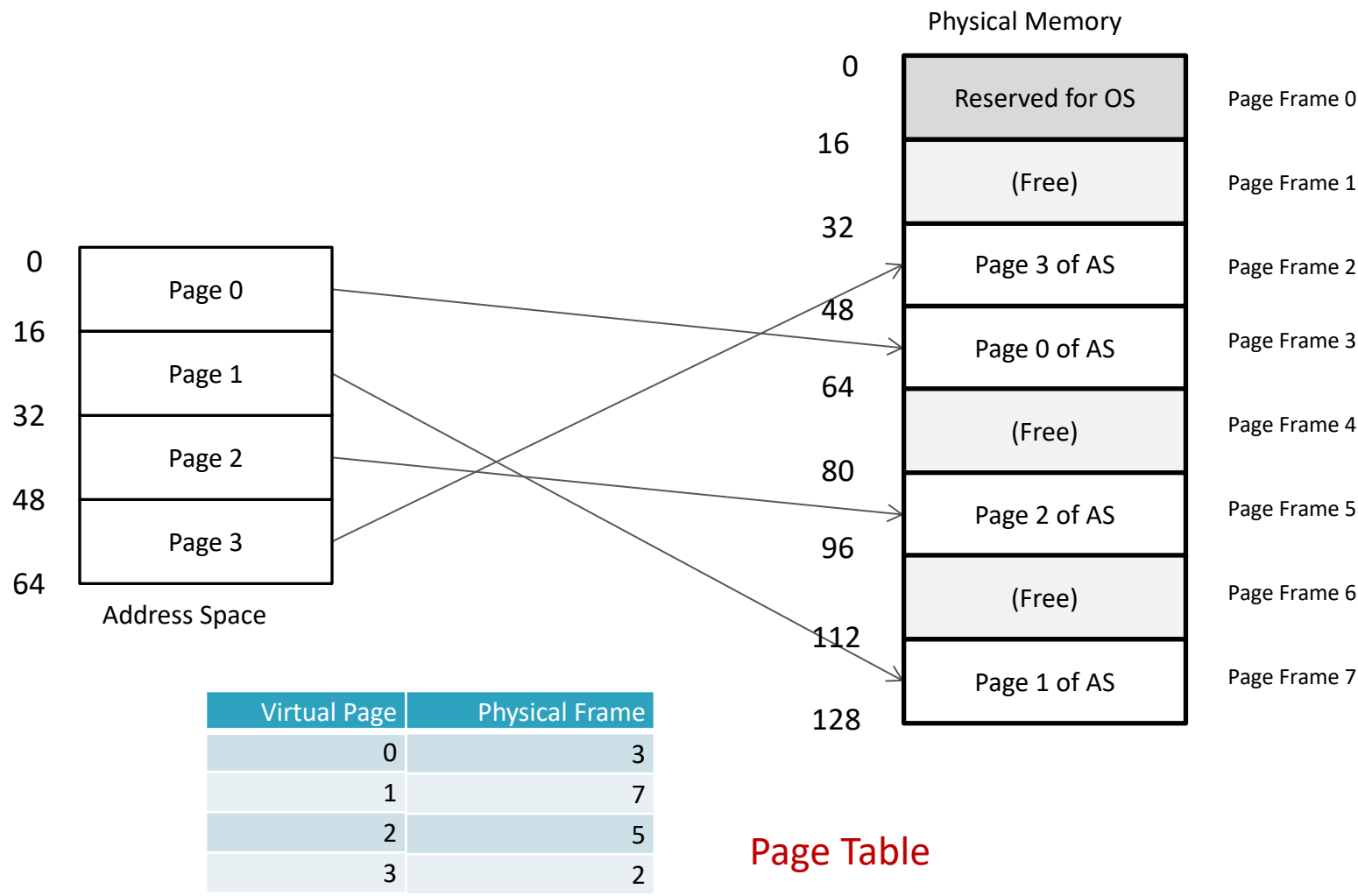
Address Translation

Page Size	Low Bits (offset)	Virtual Address Bits	High Bits (VPN)	Virtual Pages
16 bytes	4	10	6	64
1KB	10	20	10	1K
1MB	20	32	12	4K
512bytes	9	16	5	32
4KB	12	32	20	1MB



Where do we store the translations?

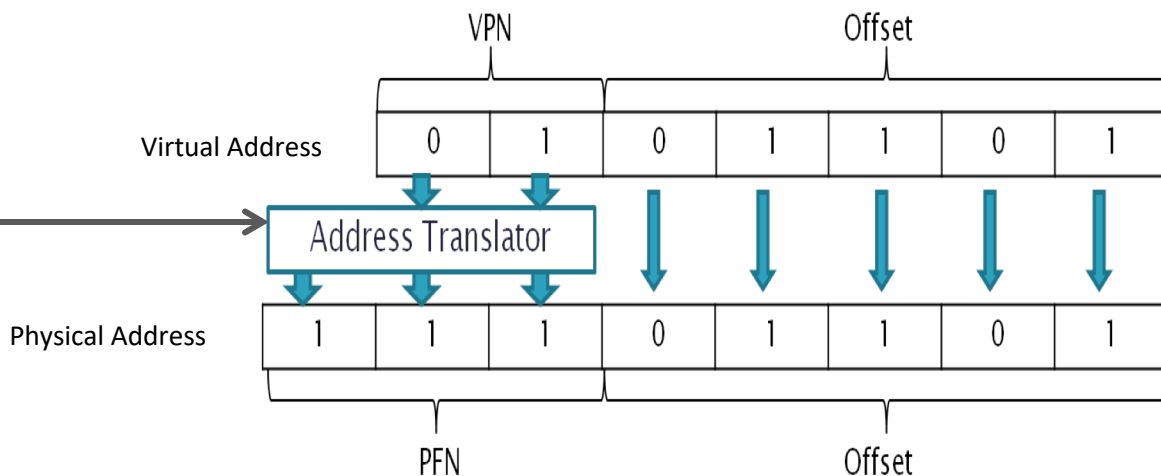
Address Translation



Address Translation

- ▶ Page table per process to record where each virtual page is placed in physical memory.
- ▶ Page table stores address translation for each virtual page of the address space.

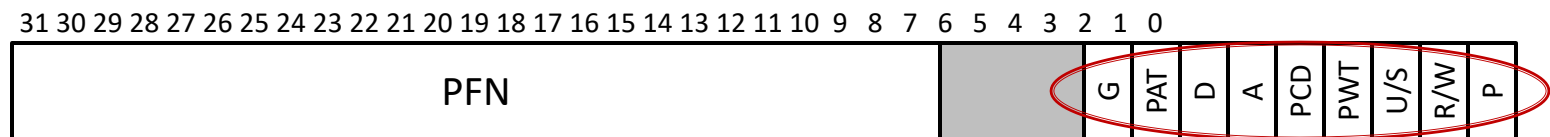
Virtual Page	Physical Frame
0	3
1	7
2	5
3	2



- ▶ We need to know the virtual page number (VPN) to get the Physical Frame Number (PFN).

Page Table

- ▶ Real address space: 32 bits (4GB) or 64 bits...
- ▶ Page tables can be terribly large.
 - i.e. 32-bits address space with 4KB pages:
 - 20-bit VPN $\rightarrow 2^{20}$ translations (~1 million per process).
 - 12-bit offset (4KB page size).
 - Assuming 4 bytes per page table entry (PTE)
 - \rightarrow 4MB of memory for each page table \rightarrow **per Process!**



x86 Page Table Entry

Info bits: valid, protection,
present, reference, dirty...

- ▶ Not in MMU, but in memory (kernel space).

Address Translation

Page Table at addr 0x2000

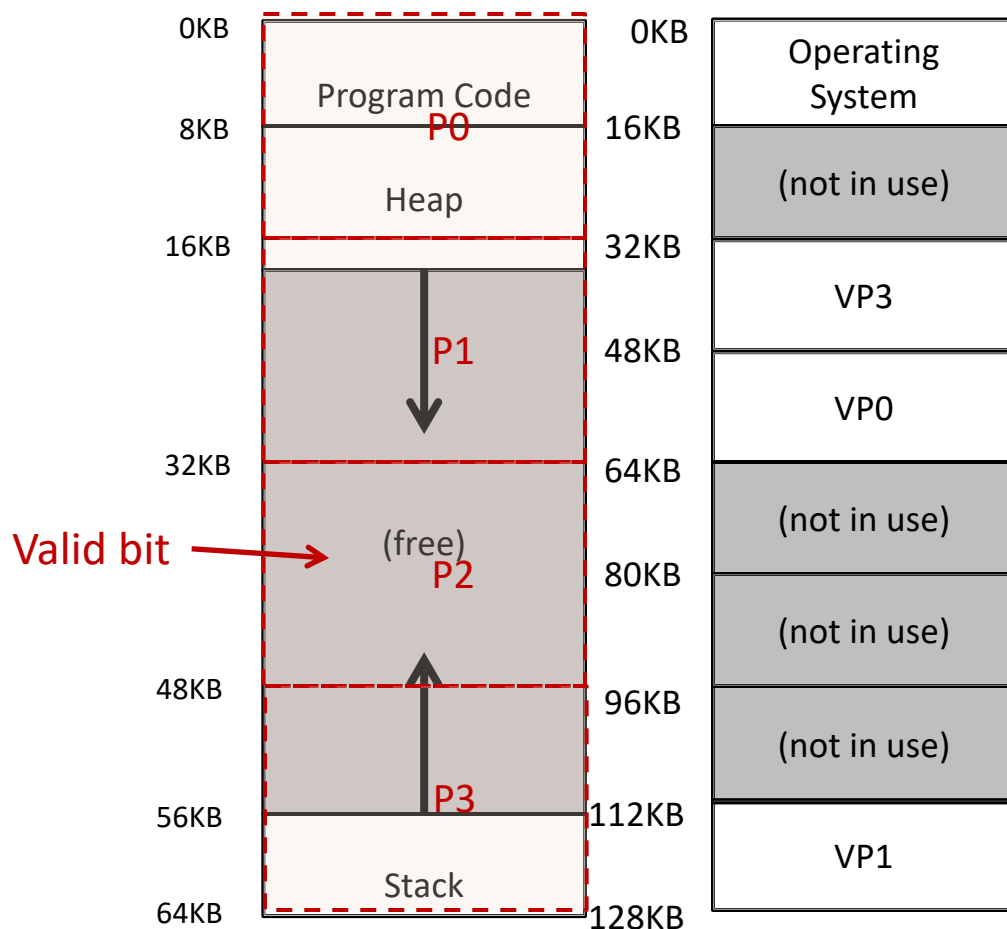
Virtual Page	Physical Frame
0	3
1	7
2	5
3	2

▶ Example #5 with Paging

- Load (PT) from addr 0x02000
- Fetch instruction at addr 0x0C128
- Load (PT) from addr 0x0200C
- Exec load from addr 0x0BC00
- Load (PT) from addr 0x02000
- Fetch instruction at addr 0x0C132
- Exec, no load
- Load (PT) from addr 0x02000
- Fetch instruction at addr 0x0C136
- Load (PT) from addr 0x0200C
- Exec store to addr 0x0BC00

Too Slow!

0x128: movl 0xFC00, %eax
0x132: addl 0x3, %eax
0x136: movl %eax, 0xFC00



Page Table

- ▶ Being in memory, accessing a page table is slow.
 - Page-table base register (PTBR) points to the page table.
 - Page-table length register (PTLR) indicates the size of the page table.
- ▶ Every data/instruction access requires two memory accesses. One for the page table, one for the data/instruction.

```
movl 21, %eax
```

Implies:

```
VPN = (VirtualAddress & VPN_MASK) >> offset_bits
PTEAddr = PTBR + (VPN*sizeof(PTE))
```

First memory access to PTEAddr to get PFN

```
Offset = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << offset_bits) | offset
```

Second memory access to get data in PhysAddr (and store it in %eax)

Paging

► Pros: ✓

- Very Flexible.
- No external Fragmentation.
- No need to move memory blocks.
- Easy Free Space Management.
 - Simple free list (valid bit).
 - Don't need to find contiguous memory.
 - No need to coalesce (fixed size pages).

► Cons: ✗

- Expensive translation (too slow).
- Huge Overhead.

3.6 Memory Virtualization

-Paging Improvements

Remember...

Mechanism	Fragmentation	Flexibility	Overhead	Speed	Free Space	AS bigger than PMem
Base & bounds	Internal (big)	Small	Small	Fast	Simple	No
Segmentation	External (variable)	Medium	Small	Fast	Complex	Yes
Paging	Internal (small)	High	Big	Slow	Simple	Yes

Reduce the impact of these

Paging too slow

▶ Too Slow → Translation Steps

▶ For each mem reference:

- cheap ○ extract VPN (virt page num) from VA (virt addr)
- cheap ○ calculate addr of PTE (page table entry)
- expensive ○ fetch PTE
- cheap ○ extract PFN (page frame num)
- cheap ○ build PA (phys addr)
- expensive ○ fetch PA to register

▶ Which steps are expensive?

▶ Which expensive step can we avoid?

Paging too slow

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

Virtual	Physical
0x3000	Load 0x100C (PT) Load 0x7000
0x3004	Load 0x100C (PT) Load 0x7004
0x3008	Load 0x100C (PT) Load 0x7008
0x300C	Load 0x100C (PT) Load 0x700C
0x3010	Load 0x100C (PT) Load 0x7010
...	...

Asume:

- 4KB pages (12 bits offset)
- array a in addr 0x3000
- Page Table in addr 0x1000
- Then, translation of VA 3
at entry in addr 0x100C
- Translation: VA 3 → PA 7
- Just data array accesses

- ▶ Take advantage of repetition/locality

Common translation:

0x3000 → 0x7000

- ▶ Use some kind of CPU cache for translations.

TLB

- ▶ The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative registers or translation look-aside buffers (TLBs).
- ▶ A TLB is part of the memory-management unit (MMU).
- ▶ It is an address-translation cache that stores popular virtual-to-physical address translations.

TLB

- ▶ Upon a virtual memory reference:
 - MMU first checks the TLB to see if the translation is stored therein.
 - TLB Hit (quick)
 - Extract PFN and get physical address (PA).
 - TLB Miss (slow)
 - access page table to find the translation.
 - update TLB with the translation.
 - extract the PFN and get the physical address (PA).

TLB

► Effective Address Time (EAT)

- Associative lookup = ε nanoseconds.
- Memory cycle time is β nanoseconds.
- Hit ratio = α (percentage of times TLB hits).

$$\begin{aligned} \text{EAT} &= (\beta + \varepsilon)\alpha + (2\beta + \varepsilon)(1 - \alpha) = \\ &= 2\beta + \varepsilon - \alpha\beta \end{aligned}$$

$$\left\{ \begin{array}{l} \text{EAT} \xrightarrow{\alpha \rightarrow 1} \beta + \varepsilon \\ \text{EAT} \xrightarrow{\alpha \rightarrow 0} 2\beta + \varepsilon \end{array} \right. \quad \beta \gg \varepsilon$$

TLB

▶ Example #6: Accessing an array

- First entry (a[0]) at VPN=03.
- 4KB pages.
- PT at addr 0x1000

```
int sum = 0;
for (i=0; i<4096; i++)
{
    sum += a[i];
}
```

▶ Just consider data array accesses (ignore instructions and sum, and i variables).

- How many TLB lookups per page?
 $4096/\text{sizeof}(\text{int})=1024$
- How many TLB misses?
if $a\%4096$ (4K) is 0 then 4, else 5.
- Miss rate?
 $4/4096 \approx 0.1\%$ or $5/4096 \approx 0.12\%$

Page Table

5	3	-	7	6	4	...
---	---	---	---	---	---	-----

Virtual	Physical
0x3000	Load 0x100C (PT) Load 0x7000
0x3004	TLB hit Load 0x7004
0x3008	TLB hit Load 0x7008
0x300C	TLB hit Load 0x700C
0x3010	TLB hit Load 0x7010
...	...
0x4000	Load 0x1010 (PT) Load 0x6000
...	...

TLB	VPN	PFN
	3	7
	4	6

TLB

- ▶ TLB improves performance due to:
 - spatial locality → Elements of the array are packed into pages.
 - temporal locality → Quick re-referencing of memory items in time.

(like any cache)

TLB Issue. Replacement Policy

- ▶ TLB is finite → need to replace an entry when installing a new one.
- ▶ Goal: Minimize miss rate (increase hit rate)
- ▶ Typical policies:
 - Least-recently-used (LRU)
 - Random (sometimes better than LRU!)
 - FIFO

Like most caches!

More about replacement policies later

TLB behavior

- ▶ When does TLB perform ok?
 - Sequential accesses can almost always hit in the TLB
 - Fast translation!

- ▶ What kind of pattern would be slow?
 - Highly random (no repeat accesses).
 - Sequential accesses that load one page at a time and need more pages than TLB size and LRU.
 - i.e. 4KB pages. 4 entries TLB
 Virtual address accesses to:
 0x1000 0x2000 0x3000 0x4000 0x5000 0x6000...

Who handles the TLB miss?

▶ Hardware

- Needs to know the page-table location (PTBR).
- Hardware-managed TLB, like CISC Intel x86 multi-level page table.

▶ Operating System

- Software-managed TLB. RISC systems (MIPS, SPARC...)
- On a miss, hardware raises an exception → trap handler.
- Special return-from-trap (same instruction, not next).
- Avoid chain TLB misses from handler (TLB handler in unmapped physical memory → always hit TLB).
- Flexibility and Simplicity

TLB Issue. Context Switches

- ▶ TLB contains virtual to physical translation valid for the current process.
- ▶ What happens if a process uses the cached TLB entries from another process?
 - Flush the TLB (set all entries as invalid) → valid bit.
 - Address space identifier → ASID field. (kind of PID).
 - Remember which entries are for each process.
 - Even with ASID, other processes “pollute” the TLB

Context Switches are expensive!

TLB examples

- ▶ Typical TLB: 32, 64 or 128 entries fully associative cache (search in parallel in all entries).

- ▶ TLB entry:

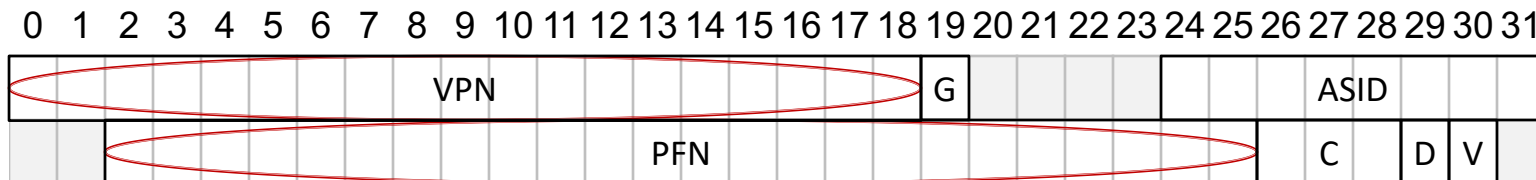
VPN | PFN | Other bits

- Other bits:

- valid bit (valid translation or not) \neq page table valid entry.
- Protection bits (read/write/execute)
- Address-space identifier
- Dirty bit...

- Real MIPS TLB entry (32 bits address space, 4KB pages):

19 bit VPN!



Support systems up to 64GB physical memory

Paging: Smaller Tables

- ▶ Page tables are too big and consume too much memory.
- ▶ Why do we want big virtual address spaces?
 - programming is easier
 - applications don't need to worry (as much) about fragmentation

Simple Solution. Bigger Pages

- ▶ 32-bits address space with 4KB pages and 4-byte PTE means 4MB page table size.
- ▶ 32-bits address space with 16KB pages (18-bits VPN and 14-bits offset) and PTE 4-bytes means 1MB page table size.

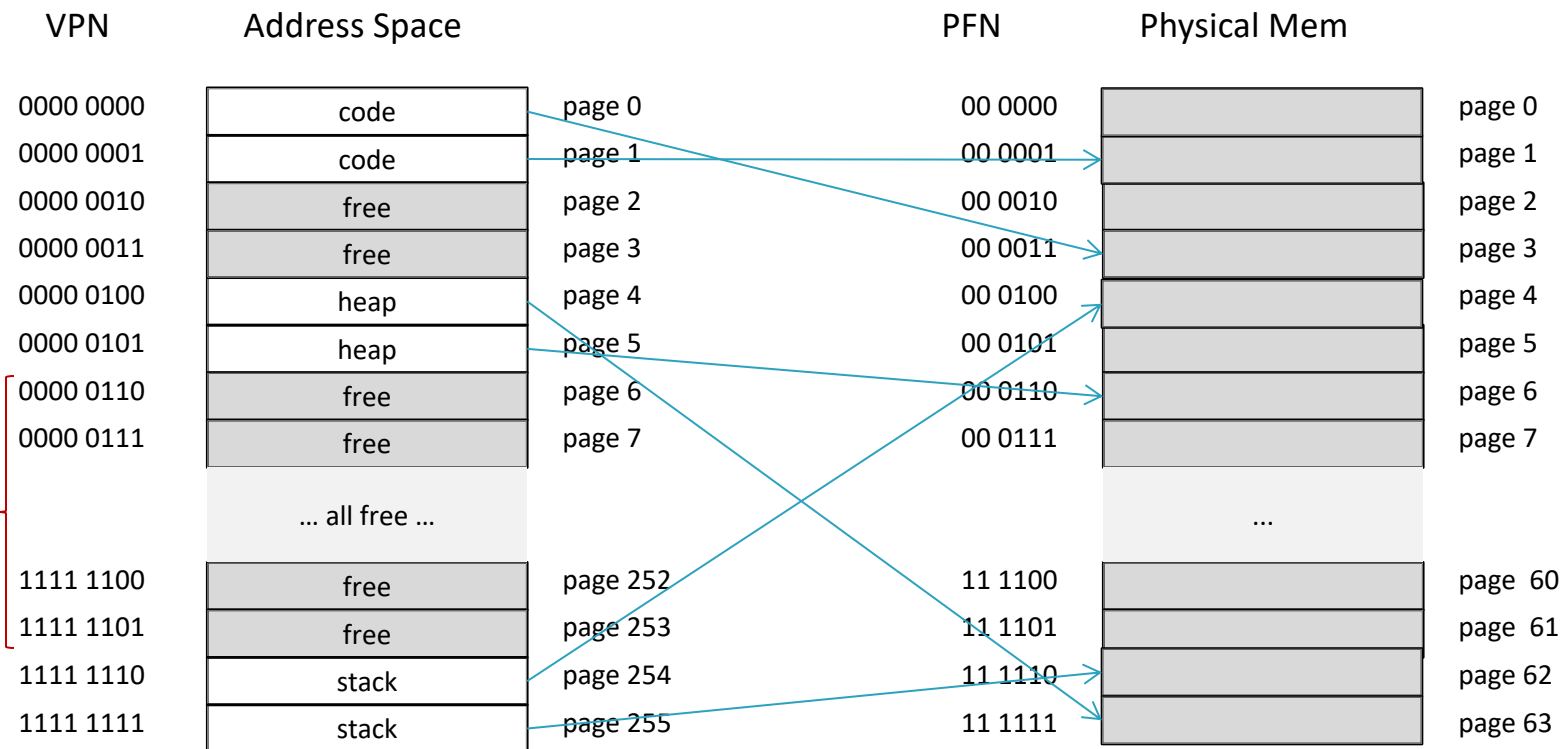
Why don't we use bigger pages?

- ▶ Bigger Pages lead to more internal fragmentation (waste space within each page).

Many architectures support multiple page sizes (4KB, 2MB, 1GB)

Page Table – Wasted Space

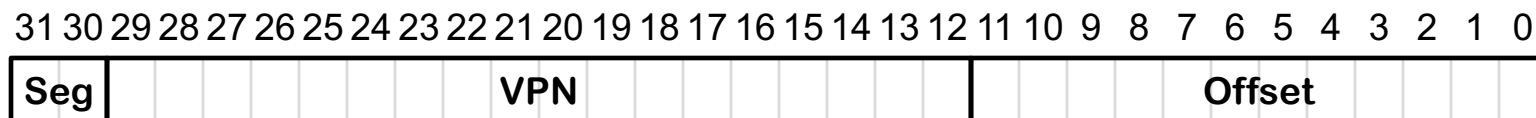
Not used!



But present in the page table

Hybrid: Paging and Segmentation

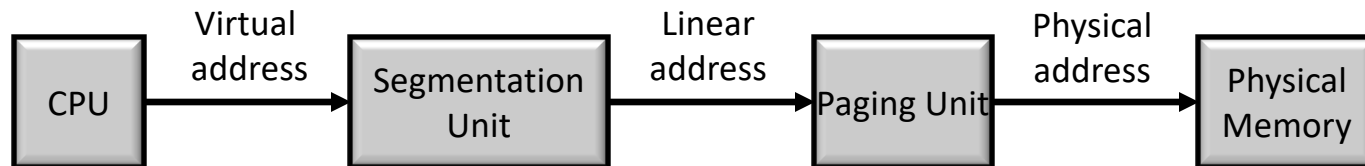
- ▶ Reduce the amount of space allocated for page tables.
 - Wasted non-valid entries.
- ▶ One page table per segment → page table of arbitrary size.



$SN = (VirtualAddress \& SEG_MASK) \gg SN_SHIFT$

$VPN = (VirtualAddress \& VPN_MASK) \gg VPN_SHIFT$

$AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))$

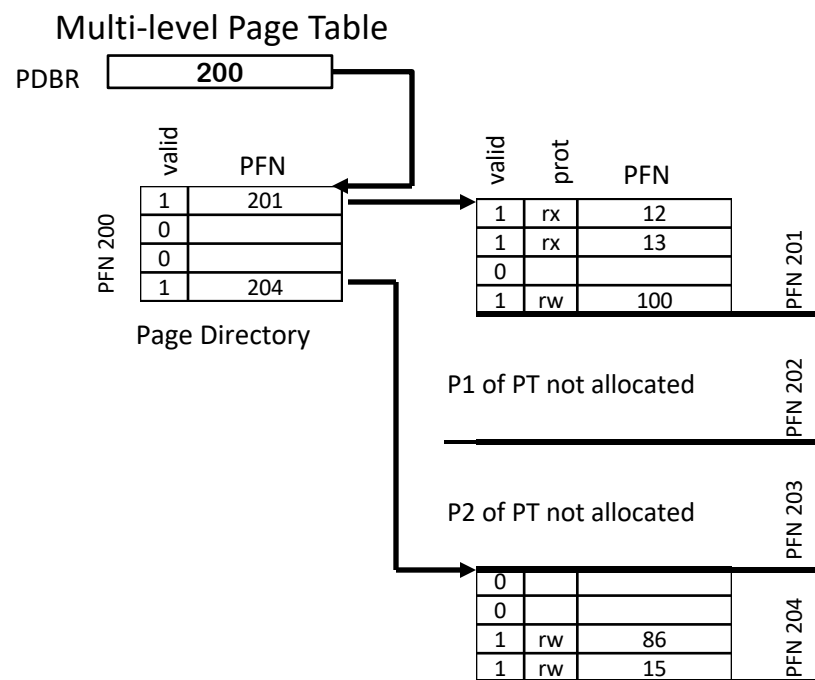
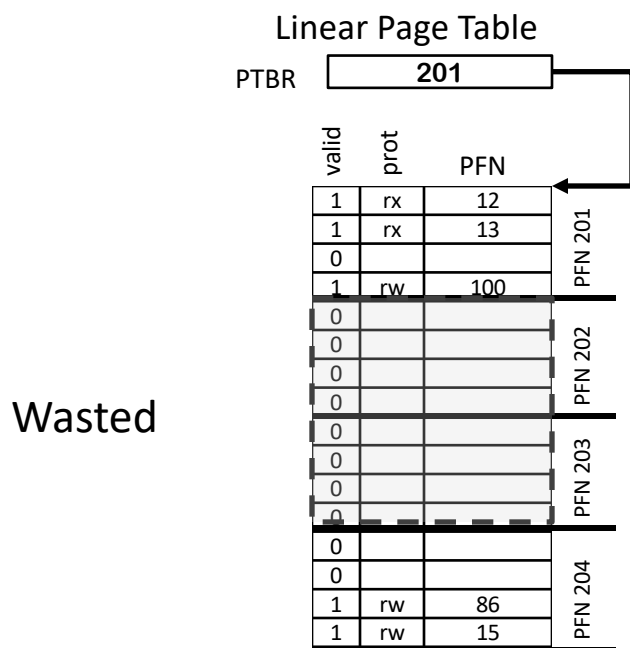


Known Segmentation Problems:

- No benefits on sparse Address Spaces (big segments)
- “External Fragmentation” (page table of variable size)

Multi-level Page Tables

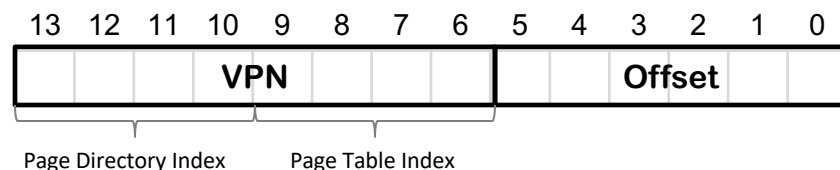
- ▶ Reduce the number of invalid regions in the page table converting linear page table into a tree-like page table (multi-level page table).
- ▶ Chop up page table into page-sized units.
- ▶ New structure called page directory.
 - Where the page of the page table is.
 - Or an entire page of the page table is invalid.



Multi-level Page Tables

▶ Example #7: 16KB address space, 64-byte pages, 4-bytes PTE.

- 14-bit virtual address space.
- 8-bits VPN + 6-bits offset.
- Linear page table: 256 entries (2^8).



VPN	Address Space	
0000 0000	code	page 0
0000 0001	code	page 1
0000 0010	free	page 2
0000 0011	free	page 3
0000 0100	heap	page 4
0000 0101	heap	page 5
0000 0110	free	page 6
0000 0111	free	page 7
	... all free ...	
1111 1100	free	page 252
1111 1101	free	page 253
1111 1110	stack	page 254
1111 1111	stack	page 255

Linear page table
(1KB)

Valid	Prot	PFN
1	rx	10
1	rx	23
0		
0		
1	rw	80
1	rw	59
0		
0		
...
...
0		
0		
1	rw	55
1	rw	45

page directory
(10's bytes)

Valid	PFN
1	100
0	
0	
0	
0	
0	
0	
0	
0	
0	
0	
0	
0	
0	
0	
0	
1	101

page of PT
(64bytes)

Valid	Prot	PFN
1	rx	10
1	rx	23
0		
0		
1	rw	80
1	rw	59
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		

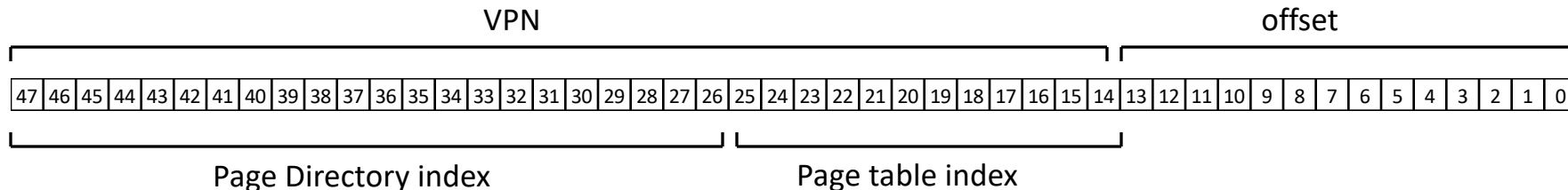
page of PT
(64bytes)

Valid	Prot	PFN
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
1	rw	55
1	rw	45

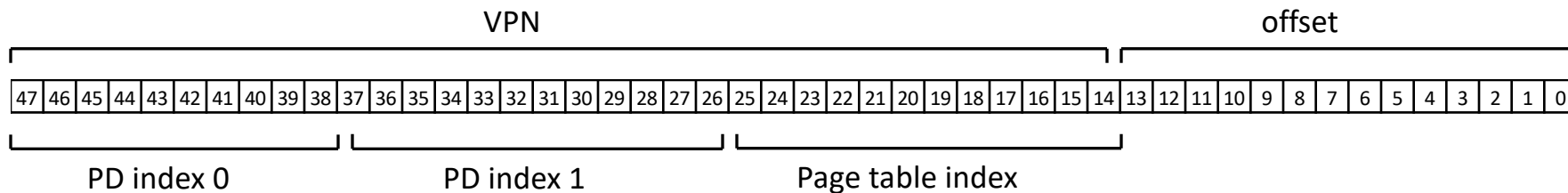
Total ~140 bytes

Multi-level Page Tables

- ▶ What if the page directory gets too big?
 - 48-bits address space, 16KB pages, PTE 4-bytes.
 - 1 page, 4096 PTEs.



- Directory Size: $2^{22} * 4 = 16\text{MB}$ (1K pages)



- More than two levels
 - 4 accesses, first look at the TLB!

Page Table. Reducing space

- ▶ Page tables are just data structures, we can try anything...
- ▶ Inverted Page tables (hash-table):
 - No multiple tables (one per process).
 - One single table with an entry for each physical page frame of the system.
 - Each entry has information of the process using the frame, and the virtual page mapped.
 - Expensive linear search → complex search mechanisms.
 - Used in PowerPC (IBM).
- ▶ Swapping Page Tables to Disk:
 - Page table resides in kernel-reserved physical memory.
 - Even reducing its size, it could be too big.
 - Some systems place the page table in kernel virtual memory, so it can swap some page tables to disk (i.e. VAX/VMS).
 - More about swapping next...

3.7 Memory Virtualization -Beyond Physical Memory

Mechanisms: Swapping

- ▶ We have assumed that every address space fits in physical memory.
 - Should we be aware of the physical memory available when programming?

- ▶ Indeed, we wish to support many concurrently-running large address spaces.
 - Not all pages will reside in physical memory.
 - We need a place to stash pages without great demand.
 - Should have more capacity (slower) → usually a hard disk drive.

Swap Space

- ▶ OS needs to reserve some space on the disk to allow moving pages → swap space.
 - OS reads from and writes to swap space in page-sized units.
 - OS needs disk address.
 - Swap space size determines the maximum number of memory pages in the system at a given time.

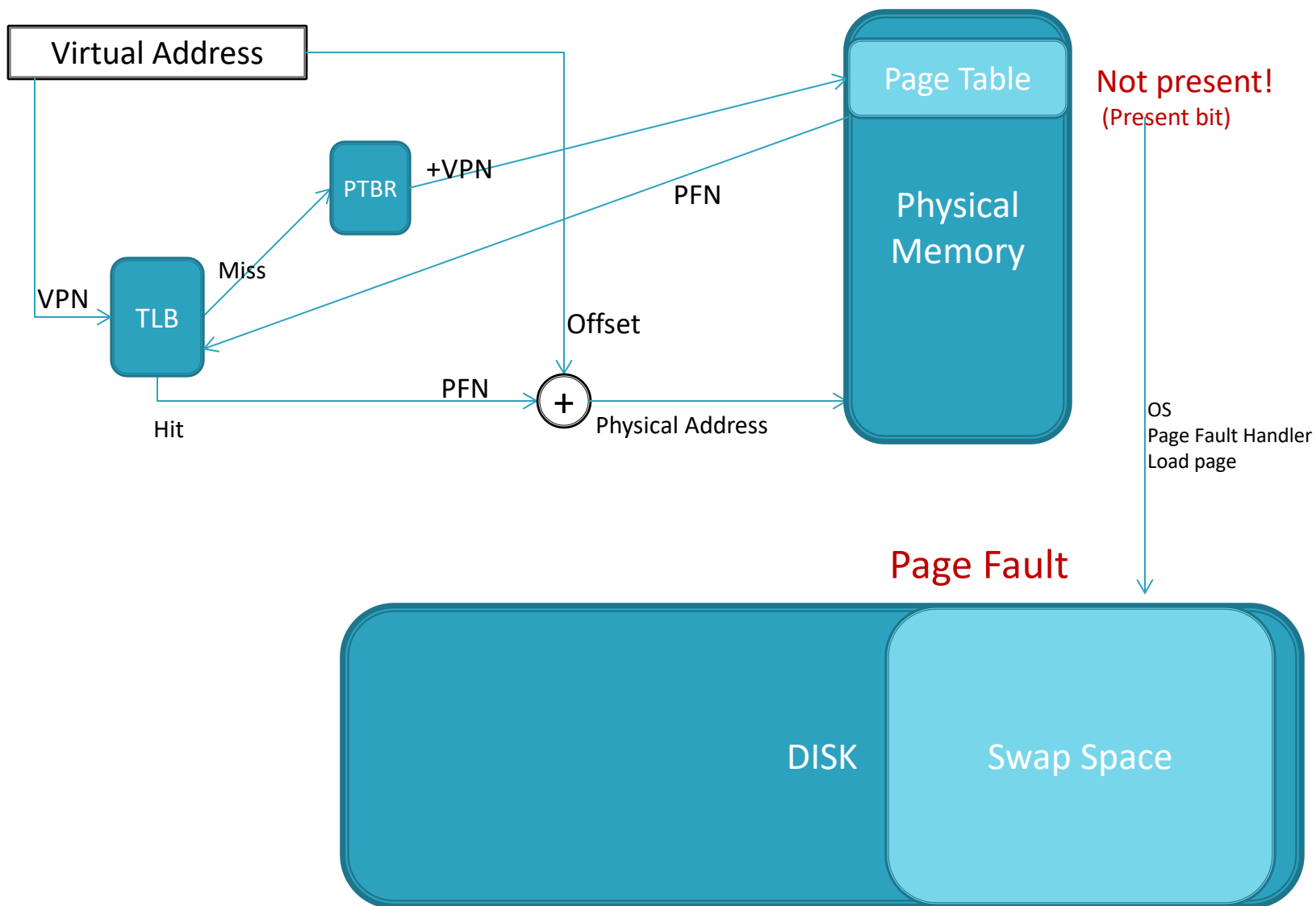
Physical Memory

Proc0 [VPN0]
Proc1 [VPN2]
Proc0 [VPN2]
Proc2 [VPN0]
Proc1 [VPN0]
Proc0 [VPN1]
Proc2 [VPN3]
Proc4 [VPN0]

Swap Space

Proc1 [VPN1]	Proc3 [VPN3]		Proc4 [VPN2]
Proc1 [VPN3]	Proc3 [VPN0]		
Proc0 [VPN3]			
		Proc0 [VPN4]	Proc4 [VPN03]
	Proc2 [VPN2]		
Proc3 [VPN1]	Proc2 [VPN1]		
	Proc2 [VPN4]	Proc3 [VPN4]	
Proc3 [VPN2]		Proc4 [VPN1]	Proc4 [VPN4]

Swap Space



Page Fault

- ▶ On a TLB miss, hardware locates the page table in memory. If the page is not in physical memory (present bit) → OS is invoked to handle it (page-fault handler).
 - OS swaps the page into memory from disk and updates the page table (present bit and PFN).
 - Next try will fail in TLB, hit in table and update TLB. Last try will hit TLB and request to memory.

- ▶ During a page-fault, process will be blocked (I/O).

- ▶ What if memory is full? (no place for the swapped-in page).
 - One or more pages need to be swapped-out to disk.
 - This is known as replacement and requires a page-replacement policy.

3.8 Memory Virtualization

-Swapping: Policies

Which to Replace

- ▶ Physical memory is smaller than the accumulated address spaces of all the processes:
 - Physical memory as a cache of the virtual memory pages.
- ▶ How long does it take to access a 4-byte int?
 - RAM: **tens ns** per int (depending on TLB hit)
 - Disk: **tens ms** per int
- ▶ We want to reduce the number of cache misses (fetch a page from disk).
 - $AMAT = (P_{HIT} \cdot T_M) + (P_{MISS} \cdot T_D)$

The one evicted
impacts hit rate

Policy!
- ▶ The OS decides which page to evict according to the replacement policy.

Optimal Replacement Policy

- ▶ Optimal Replacement Policy: Replace the page that will be accessed furthest in the future.
- ▶ Access pattern: 0,1,2,0,1,3,0,3,1,2,1

Access	Hit/Miss?	Evict	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

→ Compulsory misses

- ▶ Hit rate: 54.5% (85.7% ignoring compulsory misses)
- ▶ WARNING: Future is not generally known...
 - But serves as a close-to-perfect comparison point.

Simple Policy: FIFO

- ▶ First-in, First-out replacement. Pages are in a queue, when a replacement occurs, page on the tail is evicted.
- ▶ Same access pattern: 0,1,2,0,1,3,0,3,1,2,1

Access	Hit/Miss?	Evict	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	2	3,0,1
2	Miss	3	0,1,2
1	Hit	-	0,1,2

- ▶ Hit rate: 36.4% (57.1% ignoring compulsory misses)
- ▶ FIFO cannot determine the relevance of blocks.

Using History: LRU

- ▶ Least Recently Used (LRU): If a page has been accessed in the near past, it is likely to be accessed again in the near future.
- ▶ Same access pattern: 0,1,2,0,1,3,0,3,1,2,1

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

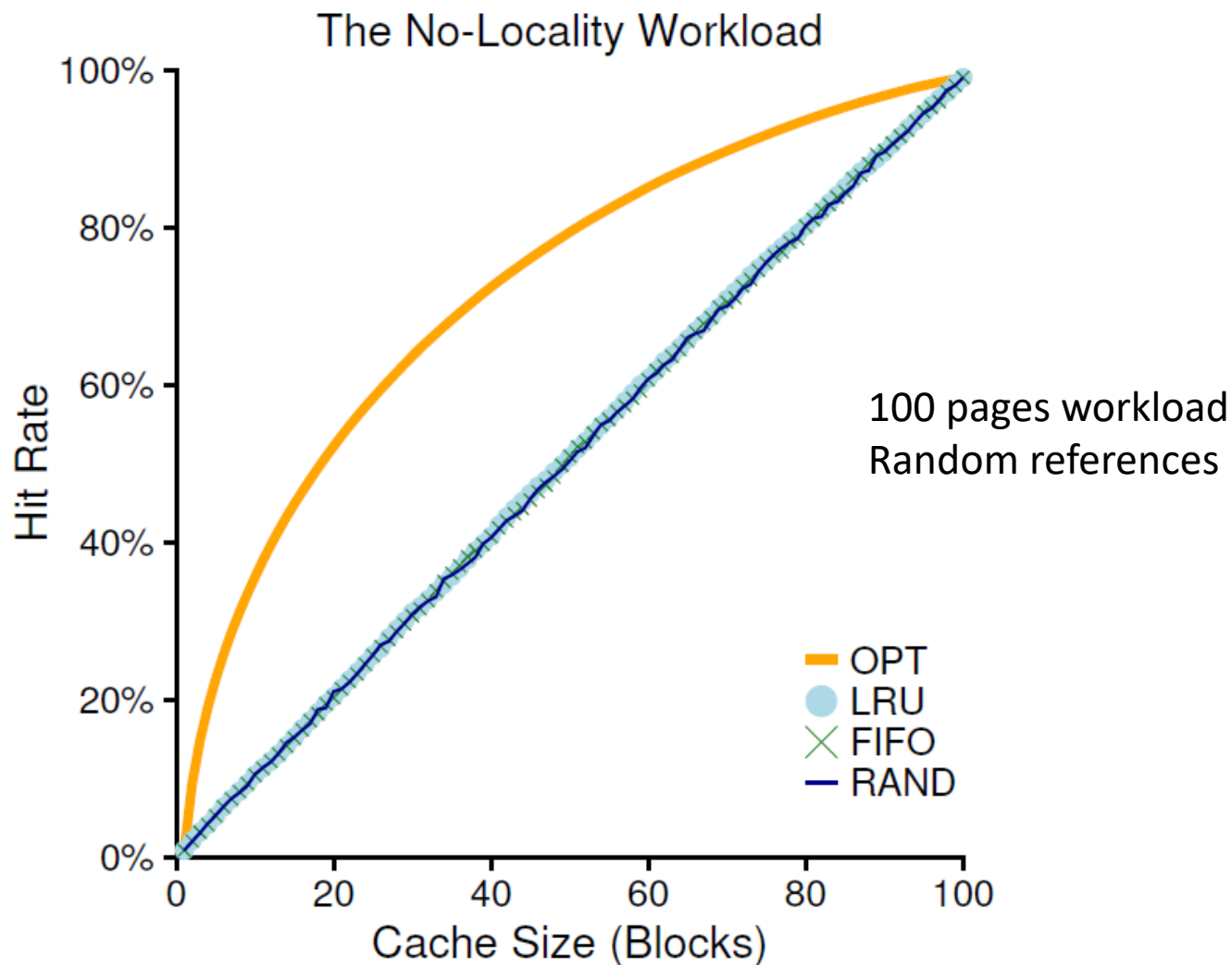
- ▶ Same Hit rate as Optimal: 54.5% (85.7% ignoring compulsory misses).

Same as Optimal! (Just an example)

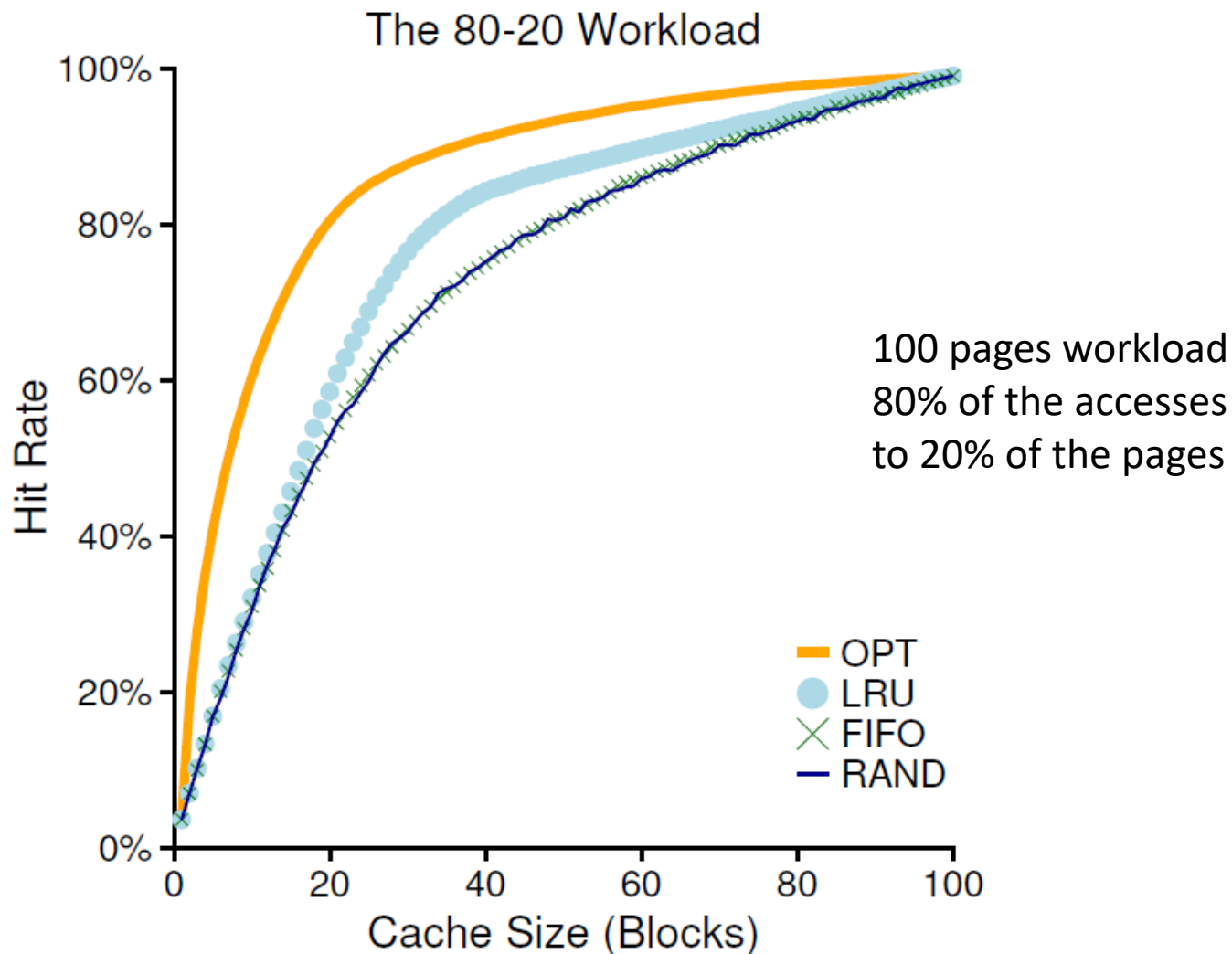
Using History: LRU

- ▶ Use history to guess the future. This family of policies are based on the principle of locality.
 - Usually programs access certain code and data frequently (loops).
 - Temporal locality: pages accessed in the near past are likely be accessed in the near future.
 - Spatial locality: if a page P is accessed, pages around it ($P-1$, $P+1$) are likely to be accessed (data arrays).
- ▶ Main members of the historically-based algorithms:
 - Least-Recently-Used (LRU): based on recency, replaces the least-recently-used page.
 - Least-Frequently-Used (LFU): based on access frequency, replaces the least-frequently-used page.
- ▶ The opposites of these algorithms exist:
 - Most-Recently-Used (MRU).
 - Most-Frequently-Used (MFU).
 - In most cases (not all), programs exhibit locality and these algorithms do not perform well.

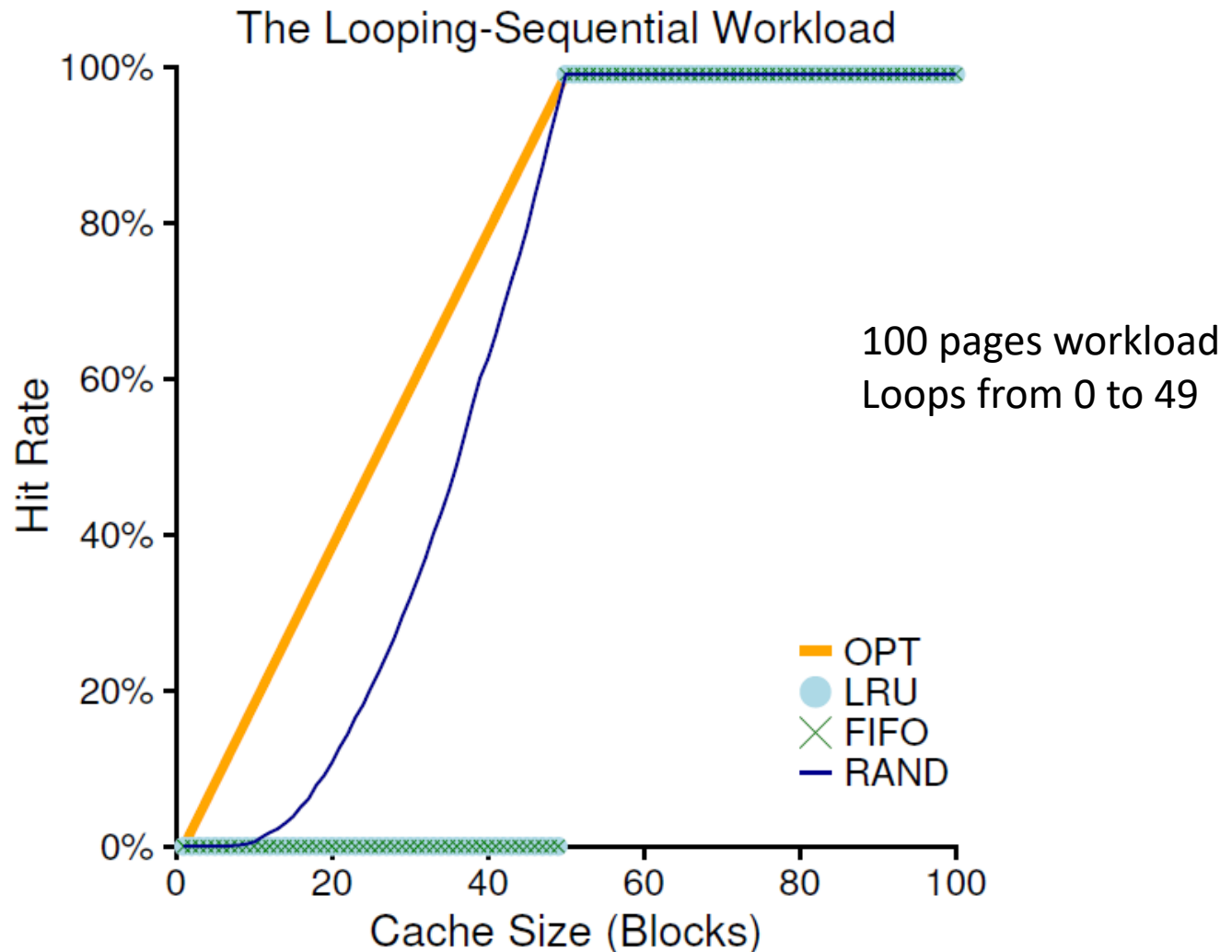
Policy Behavior - Workloads



Policy Behavior - Workloads



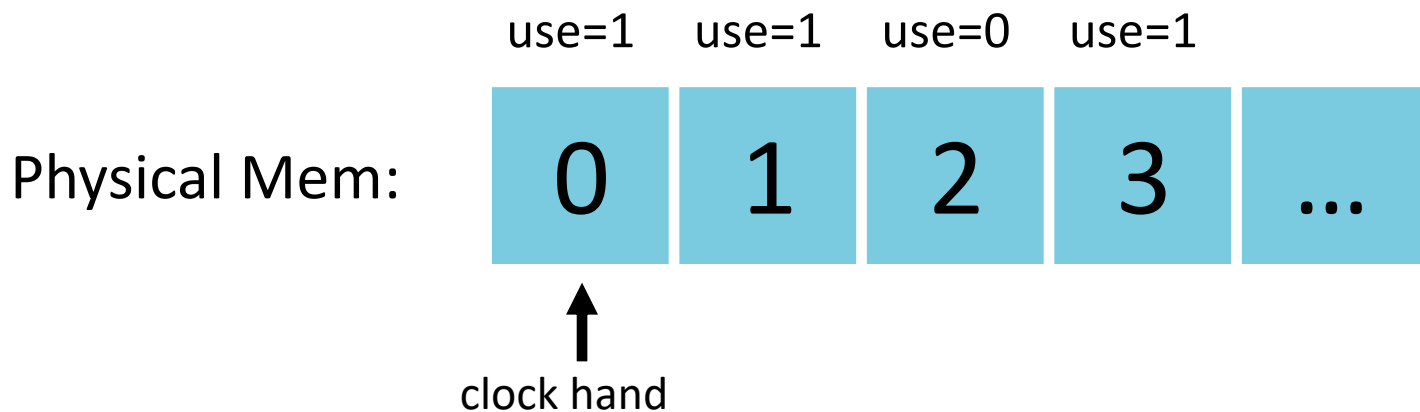
Policy Behavior - Workloads



LRU Implementation

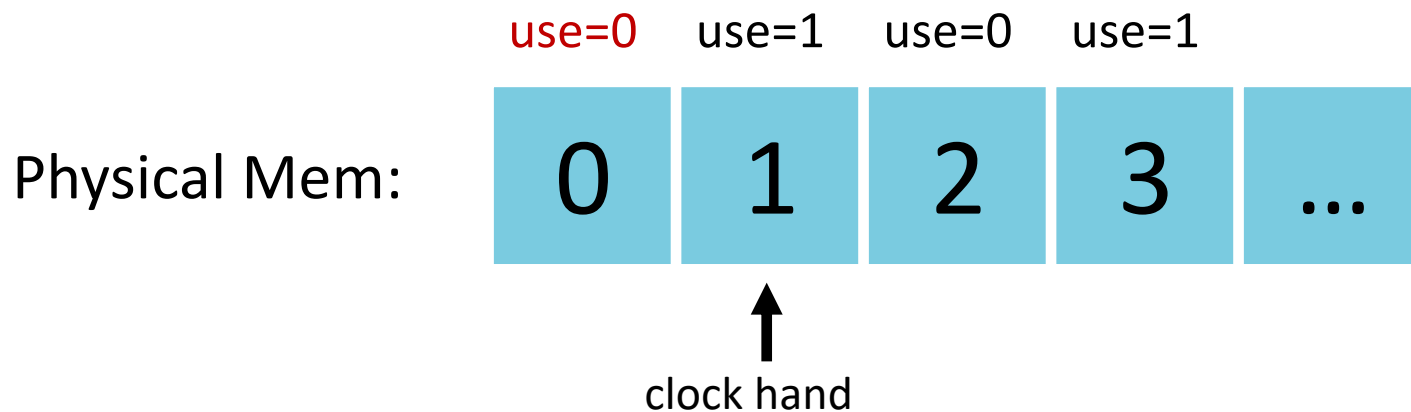
- ▶ To be perfect, must grab a timestamp on every memory reference and store it in the PTE (too expensive).
- ▶ We need an approximation. Hardware support: use bit or reference bit.
 - Whenever a page is referenced → ref bit set to 1.
- ▶ Counter implementation:
 - Keep a counter on PTE.
 - At regular intervals for each page, do:
 - if ref bit == 1, increase counter.
 - if ref bit == 0, zero the counter.
 - regardless, ref bit = 0.
- ▶ Clock Algorithm:

Clock Algorithm

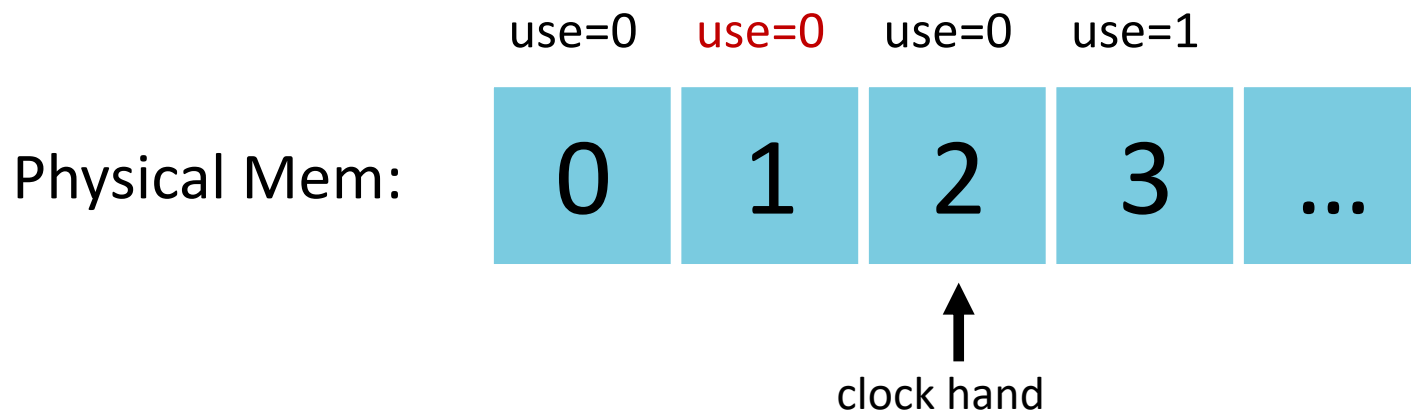


Eviction!

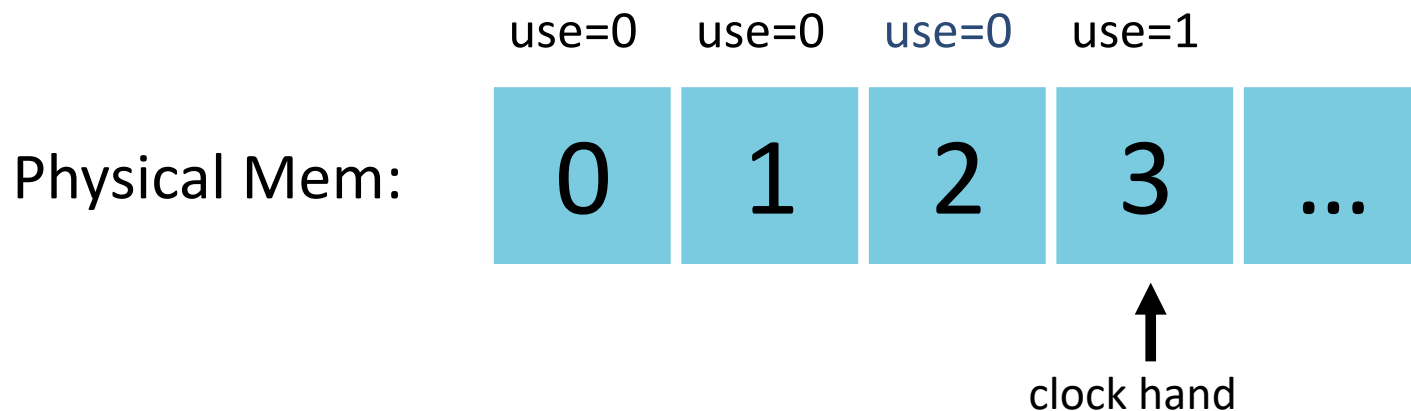
Clock Algorithm



Clock Algorithm

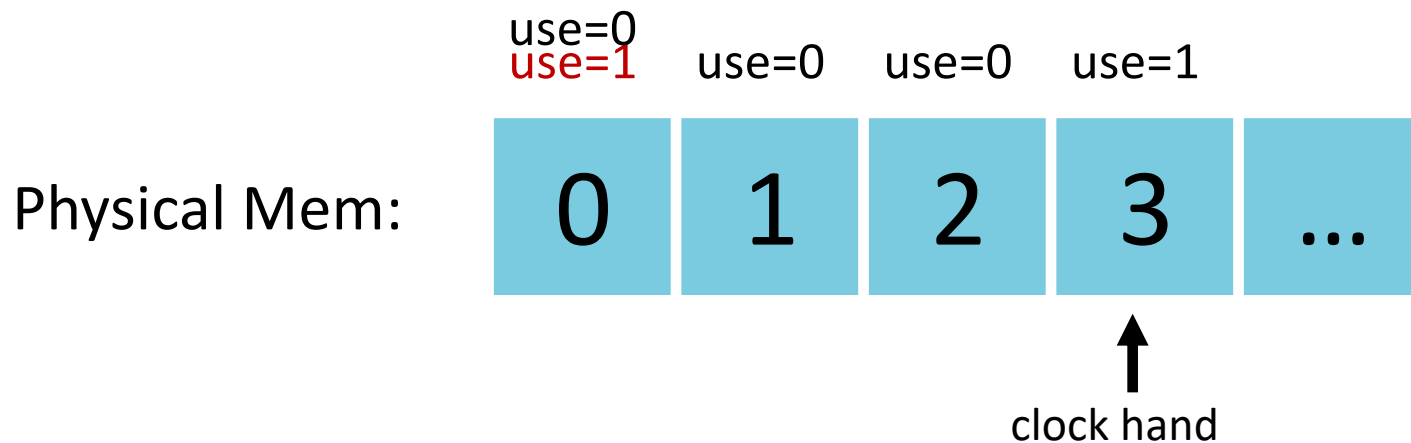


Clock Algorithm



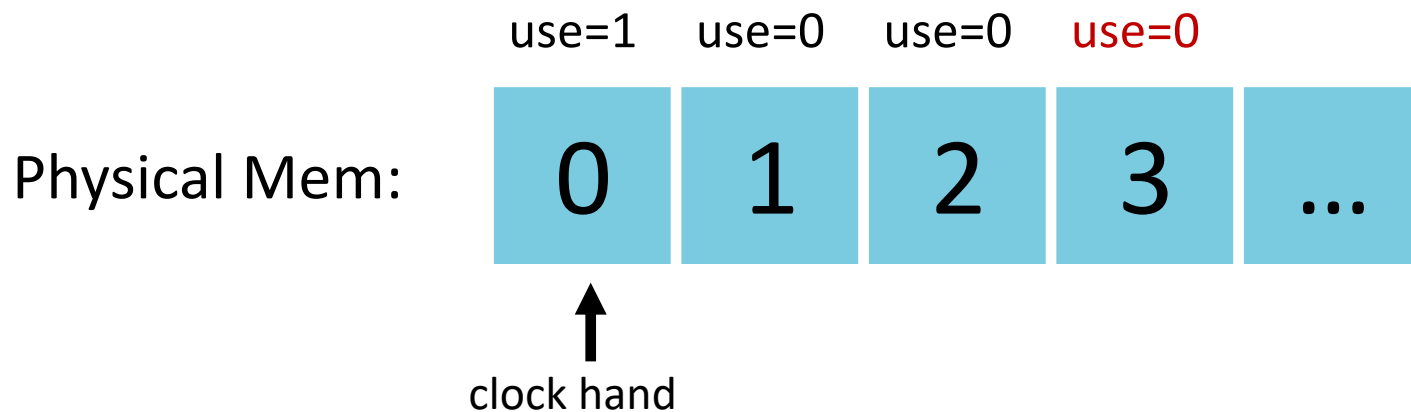
evict **page 2** because it has not been recently used

Clock Algorithm



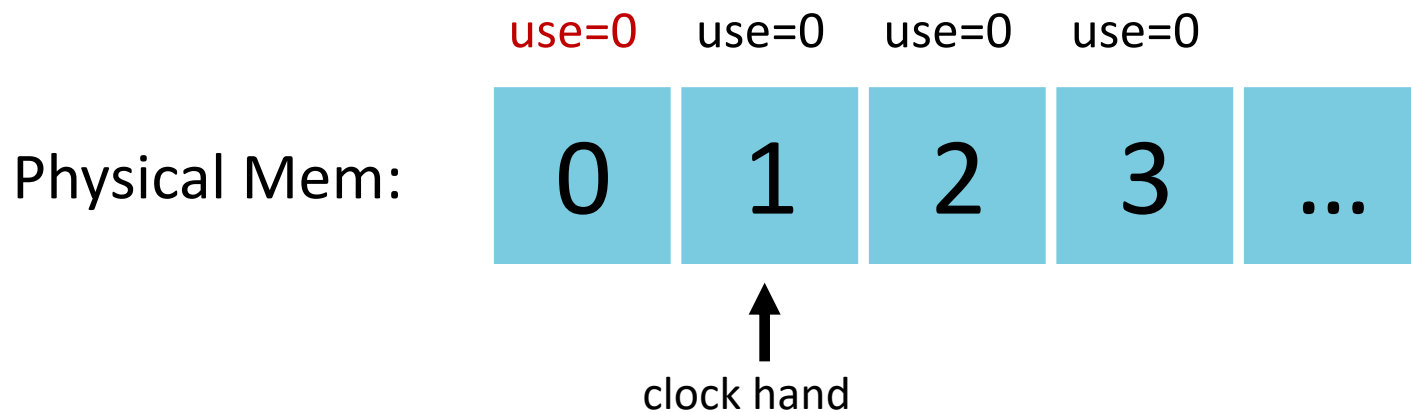
page 0 is accessed...

Clock Algorithm

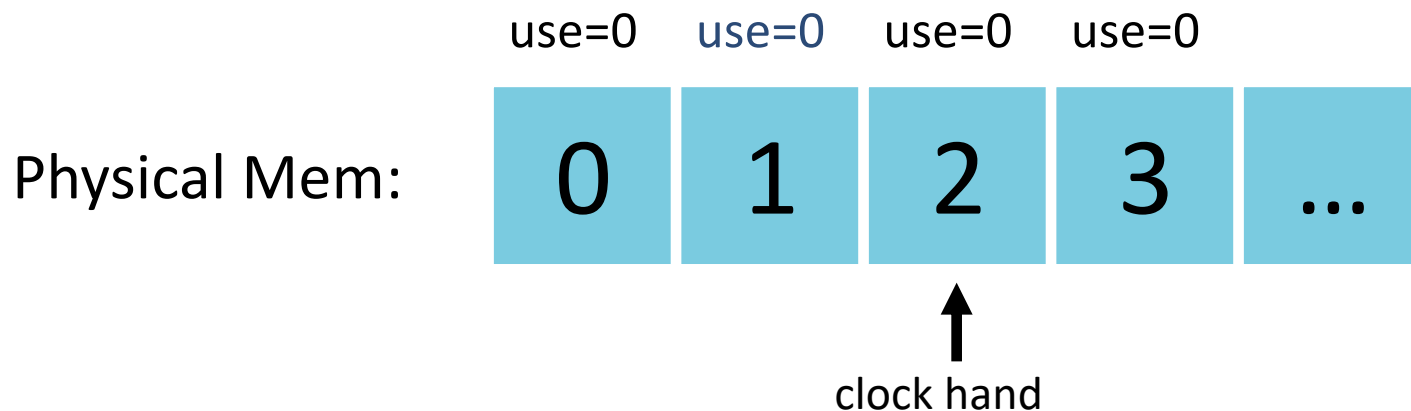


New eviction

Clock Algorithm



Clock Algorithm



evict **page 1** because it has not been recently used

Other Factors

- ▶ Dirty Pages: Do we have to write to disk on eviction?
(Assume page is both in RAM and Disk)
 - Not if the page is clean → “free” eviction.
 - Track with a dirty bit (page has been modified).
 - Can be used in page-replacement algorithm.

- ▶ Prefetching: Instead of bringing pages “on demand”, the OS guesses which page is about to be used.
 - Only when there is a reasonable chance of success (e.g. spatial locality).
 - A prefetch implies an eviction.

When to Replace

- ▶ We can assume the OS waits until memory is full to replace, but there are many reasons not to do that.
- ▶ The OS keeps a small portion of the memory free proactively.
 - High watermark (HW) and Low watermark (LW).
 - When there are fewer than LW pages available, a background thread evicts pages until HW are available again.
 - This background thread is sometimes called swap daemon or page daemon.
- ▶ This way, replacement (swapping) does not slow down most of the page-faults.
- ▶ Writing to the swap partition can be done in clusters (groups) of pages at once (more efficient).

Thrashing

- ▶ If processes do not have “enough” pages, the page-fault rate will be high. This leads to:
 - low CPU utilization.
 - operating system thinks it needs to increase the degree of multiprogramming.
 - another process is added to the system (less memory available per process).
 - more page-faults.
- ▶ Thrashing = processes are busy swapping pages in and out.
- ▶ Solution:
 - admission control: reduced set of processes (less work better than no work).
 - buy more memory...
 - Linux out-of-memory killer! → A daemon that chooses a memory intensive process and kills it.

3.9 Memory Virtualization

-Memory API

Types of memory

► Stack:

- Implicitly allocated/deallocated by the compiler → Automatic.
- In C:

```
void func()
{
    int x; //declares an integer on the stack
    ...
}
```

- Compiler makes sure to make space on the stack when you call into `func()`.
- When you return from the function, the compiler deallocates the memory.
- Information does not live beyond the call invocation.

Types of memory

► Heap:

- Longlife dynamic memory.
- Allocation and deallocation explicitly handled by the programmer (WARNING: bugs!)

```
void func()
{
    int *x = (int *) malloc(sizeof(int));
    ...
    free(x)
}
```

- Stack allocation of a pointer, heap allocation at `malloc()`.
Heap memory deallocation at `free()`.
- `free()` does not need size argument. It must be tracked by the memory-allocation library.

```
#include <stdlib.h>
void free(void* ptr);
```

The malloc() Call

- ▶ The `malloc()` call allocates space in memory. As much as its single parameter in bytes.

```
#include <stdlib.h>
void *malloc(size_t size);
```

- The programmer should not type the number of bytes directly, but reference the type and number of elements to be allocated.


```
int *x = malloc(10*sizeof(int)) //allocate an array of 10 int
```

- The returning pointer is of type `void`, the programmer decides what to do with it (usually casting the correct pointer type).



```
int *x = (int *) malloc(10*sizeof(int)) //allocate an array of 10 int
```

Common Errors

▶ Forgetting to Allocate Memory

```
char *src = "hello"
char *dst;  Unallocated!
strcpy(dst, src); Segmentation Fault
```

▶ Not allocating Enough Memory

```
char *src = "hello"
char *dst= (char *)malloc(strlen(src));  Small, needs  
end-of-string character
strcpy(dst, src);  Seems to work, but will override one  
byte too far
```

```
int *x = (int *) malloc(10*sizeof(int));
...
int *y = (int *) malloc(sizeof(x));
```

 **Not what it seems. Will return size of
the pointer.**

Common Errors

▶ Forgetting to Initialize Allocated Memory

```
int *x = (int *) malloc(10*sizeof(int));
x[0] = 15;
int y = x[1] ← Unknown value
```

▶ Forgetting to Free Memory

- Memory leak → huge problem in long runs.
- OS will take care of leaked memory when the process ends.

▶ Freeing memory incorrectly

- Before you are done with it.
- Freeing repeatedly (double free).
- Calling `free()` with wrong pointer.

Memory API

- ▶ `malloc()` and `free()` are library calls from the C library. The malloc library manages space within the virtual address space and calls the corresponding OS system calls:
 - `brk`: moves the heap pointer to a position.
 - `sbrk`: moves the heap pointer an increment.
 - Depending on the heap pointer movement, it is a `malloc()` or a `free()`.

- ▶ There are other memory management functions:
 - `mmap` creates an anonymous memory region within the program, associated to swap space (treated as heap).
 - `calloc` allocates memory and fills it with zeroes.
 - `realloc` allocates memory and copies a memory region to it.
 - used to increase the size of an already allocated region.

3. Memory Virtualization Exercises

Exercise #1

- ▶ There is a system with base and bounds as memory virtualization mechanism. During the execution of a process, we observe the following address translations:

Virtual Address	Translation: Physical Address
0x0308 (decimal: 776)	Valid: 0x3913 (decimal: 14611)
0x0255 (decimal: 597)	Valid: 0x3860 (decimal: 14432)
0x03A1 (decimal: 929)	Error: Segmentation Fault

- ▶ What can we say about the value of the base register?
How about the bounds register?

Exercise #2

- ▶ There is a system that uses segmentation as its memory virtualization mechanism. The address space uses 32 bits for address, where the two most significant bits are the ones that indicate the segment. The following table shows the segment translation and the values of the corresponding registers:

Segment	Base	Bounds	Protection
0	0x1000	0x100	Read
1	0x2000	0x200	Read/Write
2	0x5000	0x500	Read/Write

- ▶ Obtain the translation of the following memory accesses:
 - ▶ Load 0x00000010
 - ▶ Load 0x40000300
 - ▶ Load 0x80000300
 - ▶ Store 0x00000050
 - ▶ Store 0xC0000010

Exercise #3

Virtual Page Number	Valid	Reference	Dirty	Page Frame Number
0	1	0	0	3
1	1	1	1	7
2	1	0	0	4
3	0	1	1	0
4	0	0	1	4
5	1	0	1	6
6	0	0	0	5
7	1	1	0	0
...				

- ▶ Given the previous page table, and considering a page size of 1024 bytes, obtain the physical address (if possible) of each of the following virtual addresses. (note: There is no need to manage page faults if there are any):
 - 0x0356, b) 0x0DA8, c) 0x8F3, d) 0x14C3, e) 0x1F01

Exercise #4

- There is a system that uses paging as its mechanism for memory virtualization. Specifically, it uses lineal page tables (one single level). The size of the address space of each process is 4GB (32bits) and the page table size is 1KB. Each Page Table Entry (PTE) has: the page frame number of the translation (PFN), a valid bit V, a reference bit R and a dirty bit D. This system allows 2GB of physical memory at most.

PTE:

PFN	V	R	D
-----	---	---	---

- a) How many entries does a page table have in this system? Is this always the case?
- b) How many pages does a page table occupy in memory?
- c) How much memory do the page tables occupy if there are 100 processes running in the system?

Exercise #5

PAGE TABLE

Virtual Page Number	Present bit	Reference bit	Dirty bit	Page Frame Number
0	1	0	0	3
1	1	1	1	2
2	1	0	0	4
3	1	0	1	7
4	0	0	1	—
5	1	0	1	6
6	0	0	0	—
7	1	1	0	0
...				

- ▶ Address space of 16bits, 1KB page size. There are 4 entries in the TLB with the following contents:

TLB

VPN	PFN	LRU
0	3	1
2	4	0
5	6	3
7	0	2

- ▶ The following addresses are accessed sequentially:
0x0356, 0x08F3, 0x14C3, 0x0DA8, 0x0180, 0x0E83
- ▶ The TLB has an LRU replacement policy. TLB hit time is 2ns, memory access time is 200ns and swapping time is 3000ns. Obtain the Effective Access Time (EAT) for this batch of requests and the hit-rate of the TLB.

Exercise #6

PAGE TABLE P1

Virtual page no.	Present bit	Reference bit	Dirty bit	Page frame number
0	1	0	0	6
1	1	1	0	7
....				—
5	1	1	1	0
6	0	0	0	—
7	1	0	0	3
...				—
30	1	1	1	5
31	1	1	1	2

PHYSICAL MEMORY

Page frame no.	Virtual page	Process	LRU
0	5	P1	4
1	0	P2	0
2	31	P1	1
3	7	P1	5
4	31	P2	3
5	30	P1	7
6	0	P1	2
7	1	P1	6

- Two processes run simultaneously in a system with 16KB of physical memory and 2KB page size.

TLB P1

VPN	PFN	LRU	V
0	6	1	1
5	0	2	1
7	3	3	1
31	2	0	1

TLB P2

VPN	PFN	LRU	V
0	1	0	1
31	4	1	1
1	3	2	0
3	7	3	0

- Obtain the contents of both TLBs if the following addresses are accessed sequentially: P1-0x0356, P1-0x08F3, P2-0xFDA8, P1-0x346C
- Each TLB uses LRU as its replacement policy, the same as the physical memory.

Exercise #7

- ▶ There is a system with multi-level page tables (two levels), 15 bits virtual address and 32 bytes page size. Each page directory entry (PDE) and each page table entry (PTE) have the same structure: one valid bit followed by 7 bits that store the page frame number.
- ▶ The register that has the address of the page directory has the decimal value 73.
- ▶ Below is a physical memory dump of three specific page frames:

```
Pg 6:      0A 1C 01 14 0B 1A 19 0A 0A 1A 0C 14 02 0C 1C 0C 15 04 0E 13 17 11 08 05 08 07 04 13 0F 1D 0F 1E
Pg 73:     A2 D2 97 96 D9 7F 87 B4 B7 F2 F4 82 BF 7F BE 93 E8 9D 99 9E F1 7F 7F B0 D8 DA EB B1 81 C3 C2 F6
Pg 114:    7F 7F 7F 7F 82 7F 7F 7F 7F 7F 7F 7F 99 7F 7F 7F 7F 7F 86 7F 7F 7F 7F 7F 7F 8F 7F 7F 7F 7F 7F
```

- ▶ Translate the following addresses and, if the translation is valid, obtain the returning value of the load request (1 byte) :
 - A) load 0x1787
 - B) load 0x2665