哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

操作系统

Operating Systems

刘川意　教授

liuchuanyi@hit.edu.cn

哈尔滨工业大学（深圳）

2021年9月

Module 4: Virtualizing Memory 虚拟内存

1. 虚拟地址空间
2. 内存管理**API**（**memory API** ）
3. **Mechanism**: 地址翻译（Address Translation）
4. 分段（ Segmentation ）
5. 空闲空间管理（ Free-Space Management ）
6. 分页（Paging）
7. 地址翻译缓存 TLB（Translation Lookaside Buffers）
8. 交换（**Swapping**）

Virtual Address

- **Every address** in a running program is virtual.
  - OS translates the virtual address to physical address

```
1.  #include <stdio.h>
2.  #include <stdlib.h>

3.  int main(int argc, char *argv[]){

4.      printf("location of code  : %p\n", (void *) main);
5.      printf("location of heap  : %p\n", (void *) malloc(1));
6.      int x = 3;
7.      printf("location of stack : %p\n", (void *) &x);

8.      return x;
9.  }
```

A simple program that prints out addresses

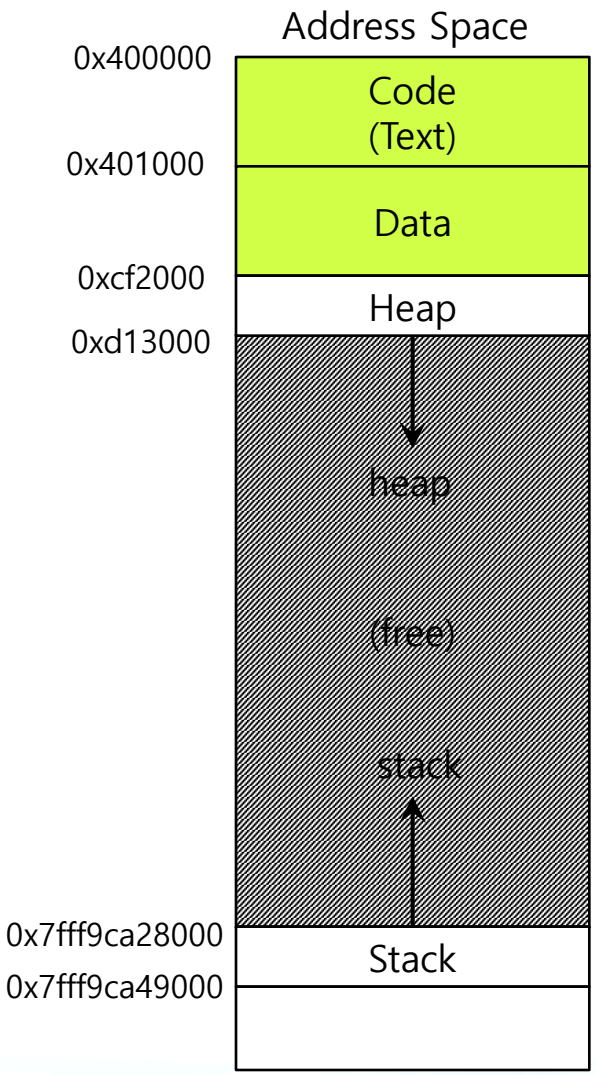
```
location of code  :0x556f705fb155
location of heap  :0x556f70ead670
location of stack :0x7ffcdc4cde64
```

- code： 存放程序执行代码
- stack：由编译器自动分配释放，存放函数的参数值，局部变量等值
- heap：存放进程运行中被动态分配的内存

Virtual Address(Cont.)

- The output in 64-bit Linux machine

```
location of code  : 0x40057d
location of heap  : 0xcf2010
location of stack : 0x7fff9ca45fcc
```



Module 4: Virtualizing Memory 虚拟内存

1. 虚拟地址空间
2. 内存管理API（Memory API ）
3. **Mechanism**: 地址翻译（Address Translation）
4. 分段（ Segmentation ）
5. 空闲空间管理（ Free-Space Management ）
6. 分页（Paging）
7. 地址翻译缓存 TLB（Translation Lookaside Buffers）
8. 交换（**Swapping**）

Memory API: malloc()

```
#include <stdlib.h>

void* malloc(size_t size)
```

- Allocate a memory region on the heap.
  - Argument
    - `size_t size` : size of the memory block(in bytes)
    - `size_t` is an unsigned integer type.
  - Return
    - Success : a void type pointer to the memory block allocated by `malloc`
    - Fail : a null pointer

sizeof()

- Routines and macros are utilized for `size` in `malloc` instead typing in a number directly.
- Two types of results of `sizeof` with variables
  - The actual size of `'x'` is known at run-time.

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

```
4
```

  - The actual size of `'x'` is known at compile-time.

```
int x[10];
printf("%d\n", sizeof(x));
```

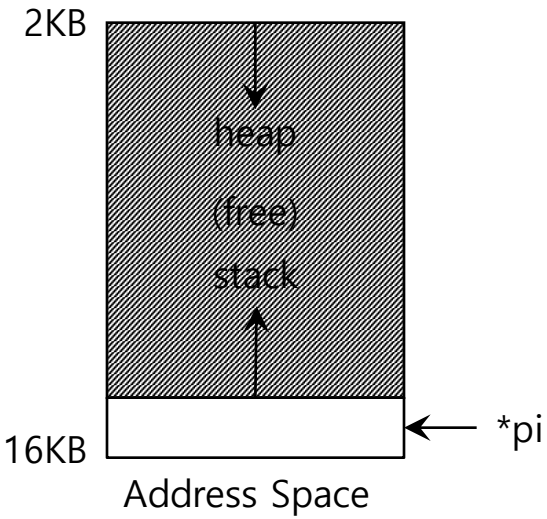
```
40
```

Memory API: free()

```
#include <stdlib.h>

void free(void* ptr)
```

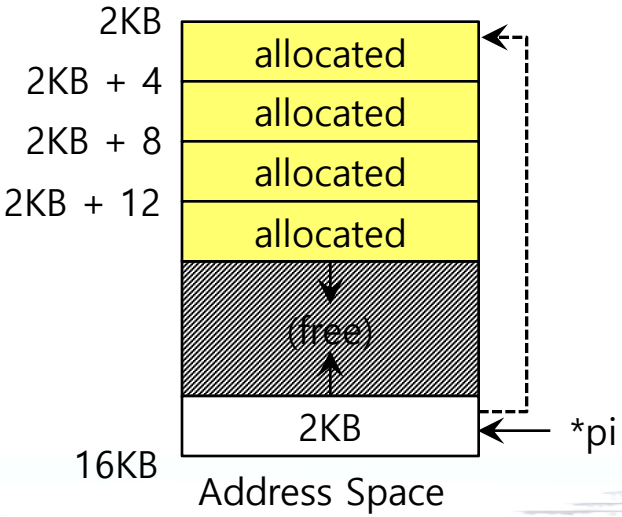
- Free a memory region allocated by a call to `malloc`.
  - Argument
    - `void *ptr` : a pointer to a memory block allocated with `malloc`
  - Return
    - none

Memory Allocating



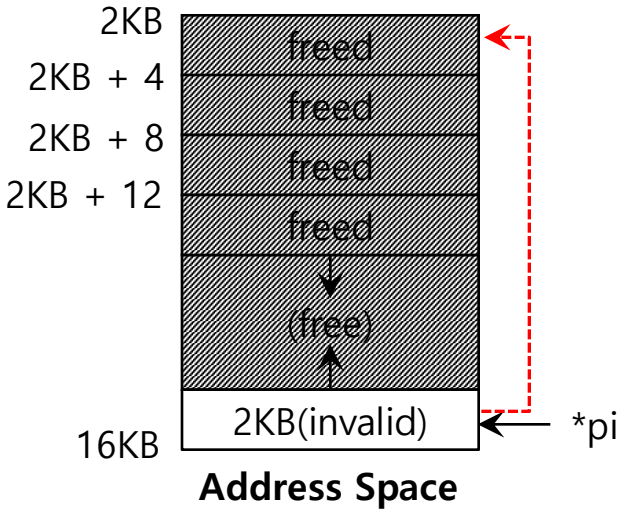
pointer

```
int *pi; // local variable
```

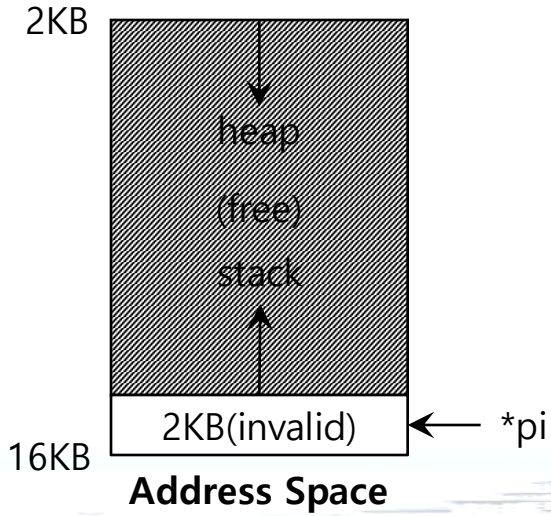



```
pi = (int *)malloc(sizeof(int)* 4);
```

Memory Freeing



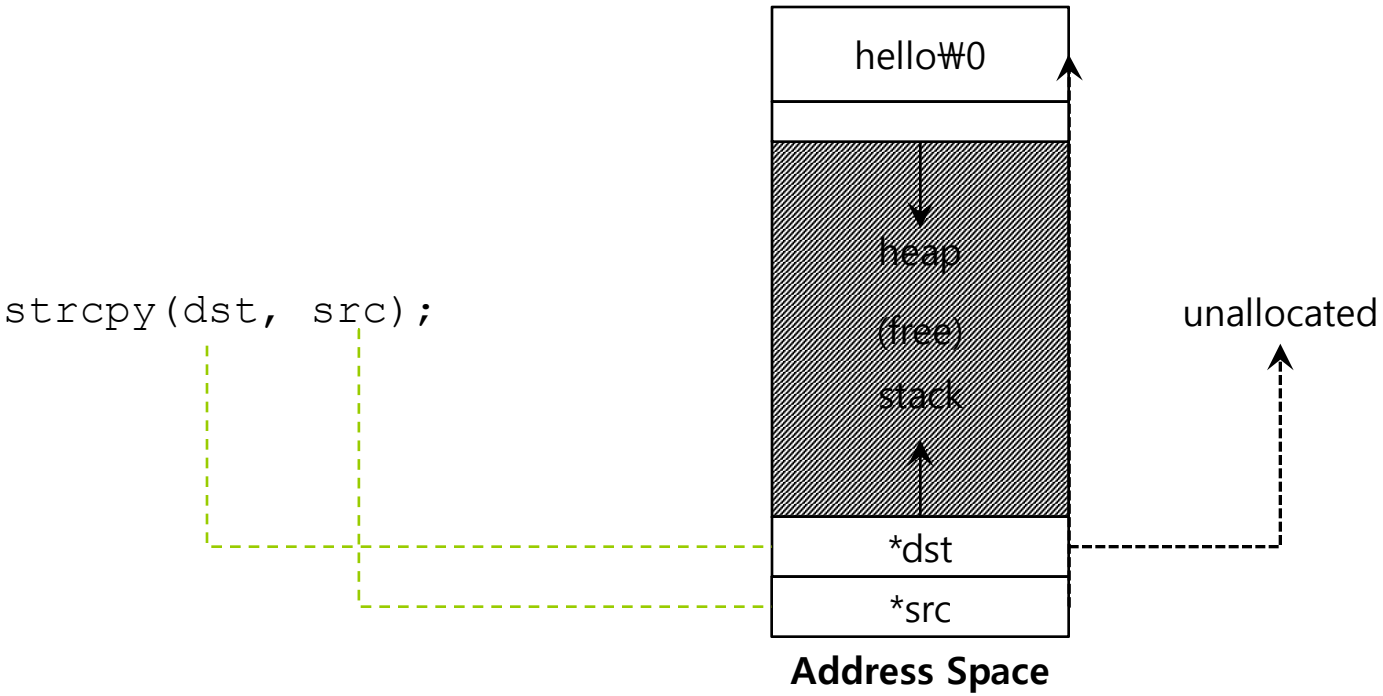

```
free(pi);
```



Forgetting To Allocate Memory

- Incorrect code

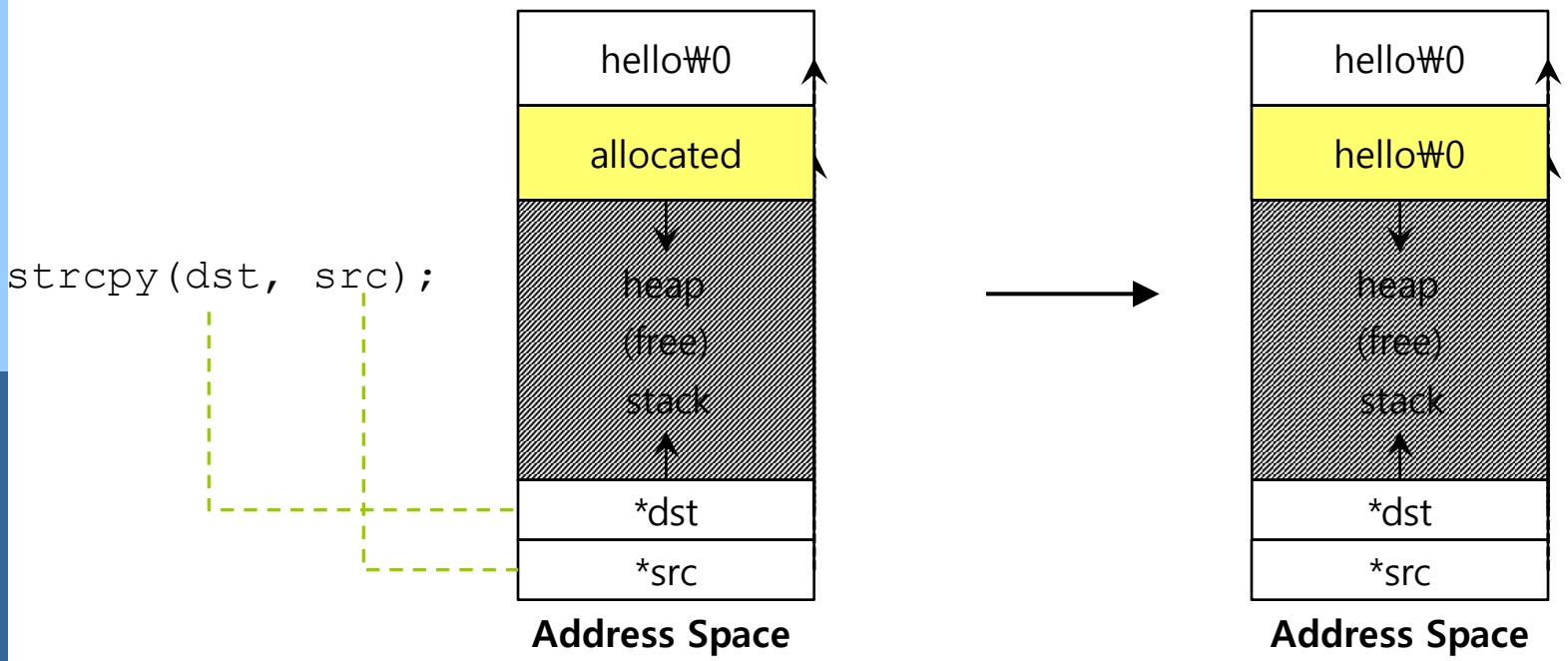
```
char *src = "hello"; //character string constant
char *dst;           //unallocated
strcpy(dst, src);    //segfault and die
```



Forgetting To Allocate Memory(Cont.)

- Correct code

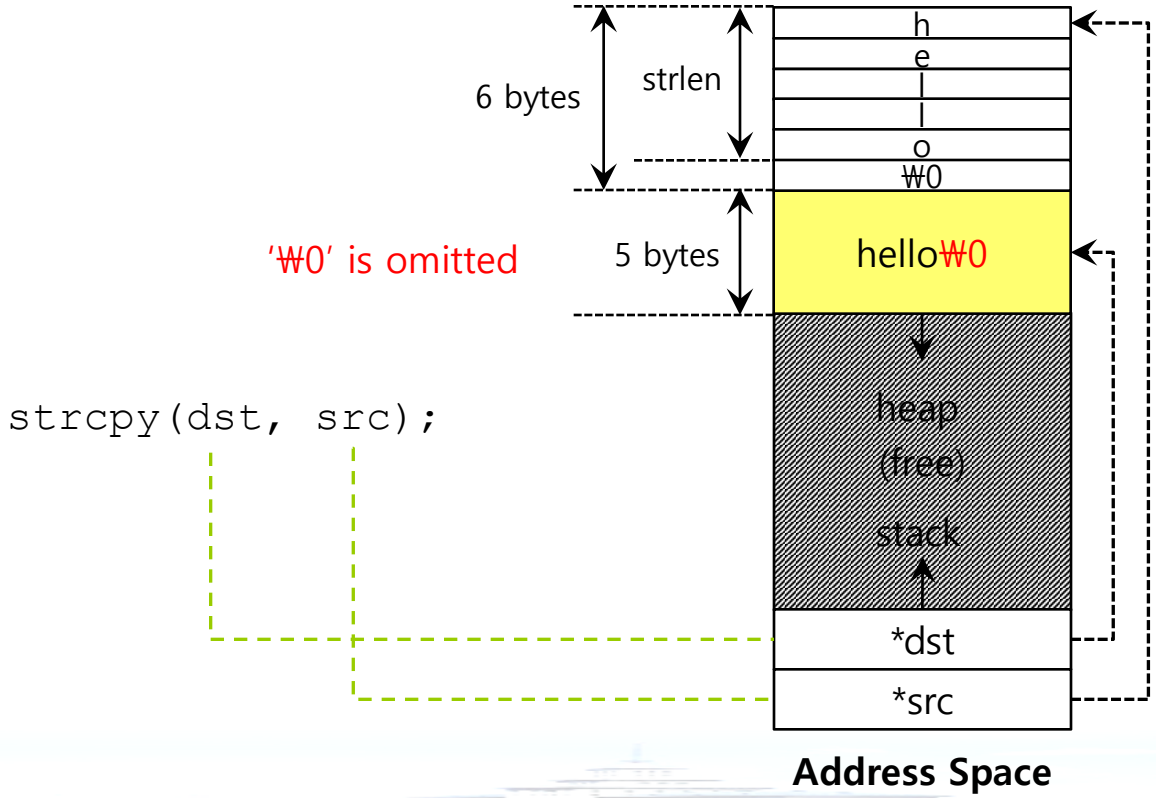
```
char *src = "hello";   //character string constant
char *dst (char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);      //work properly
```



Not Allocating Enough Memory

- Incorrect code, but work properly

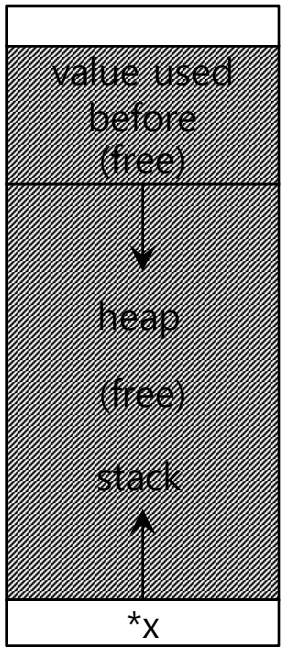
```
char *src = "hello"; //character string constant
char *dst (char *)malloc(strlen(src)); // too small
strcpy(dst, src);    //work properly
```



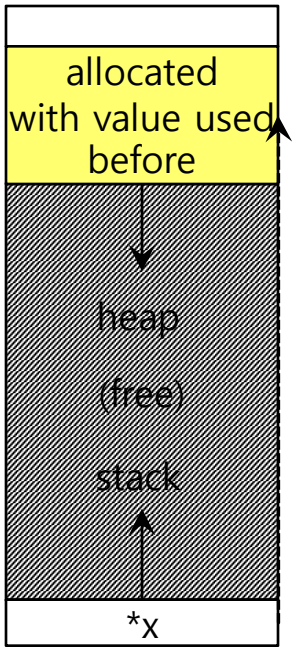
Forgetting to Initialize

- Encounter an uninitialized read

```
int *x = (int *)malloc(sizeof(int)); // allocated
printf("*x = %d\n", *x); // uninitialized memory access
```



Address Space

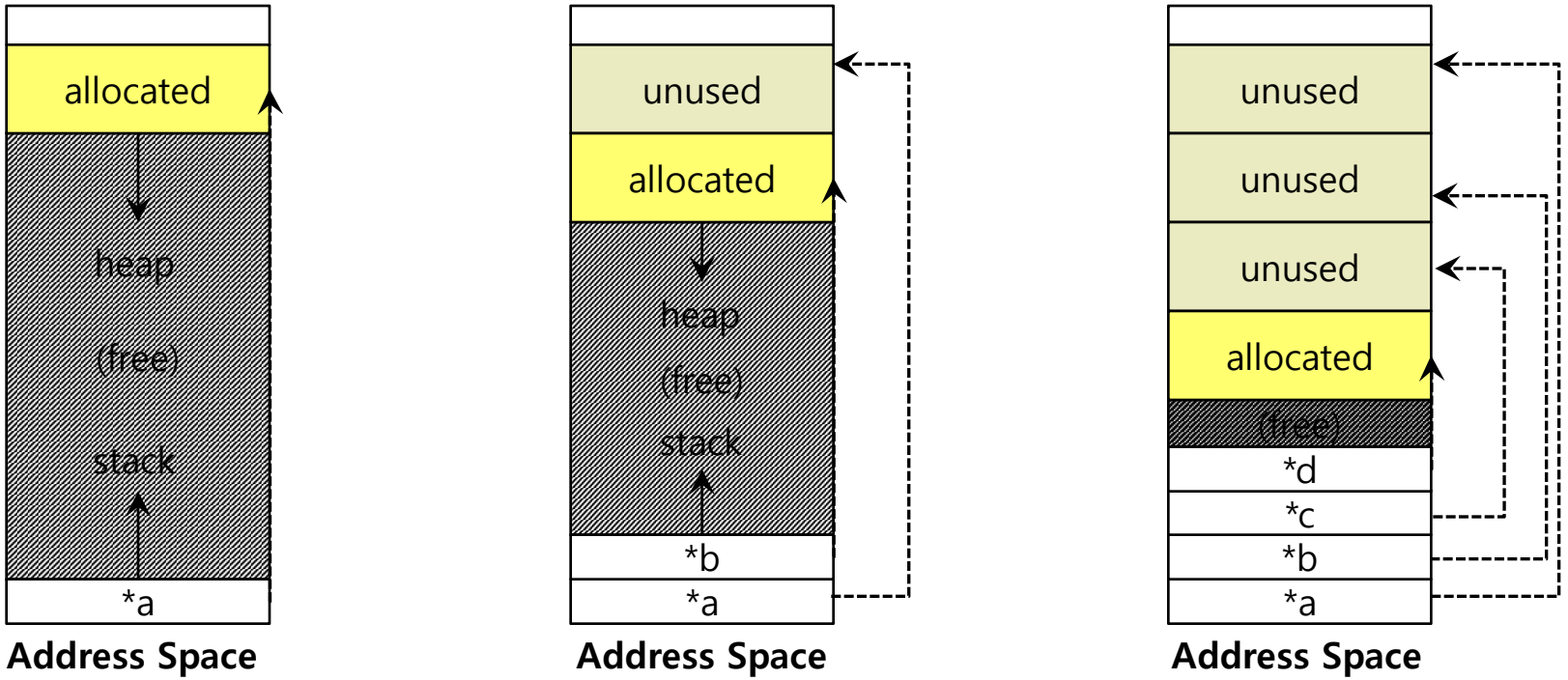


Address Space

Memory Leak

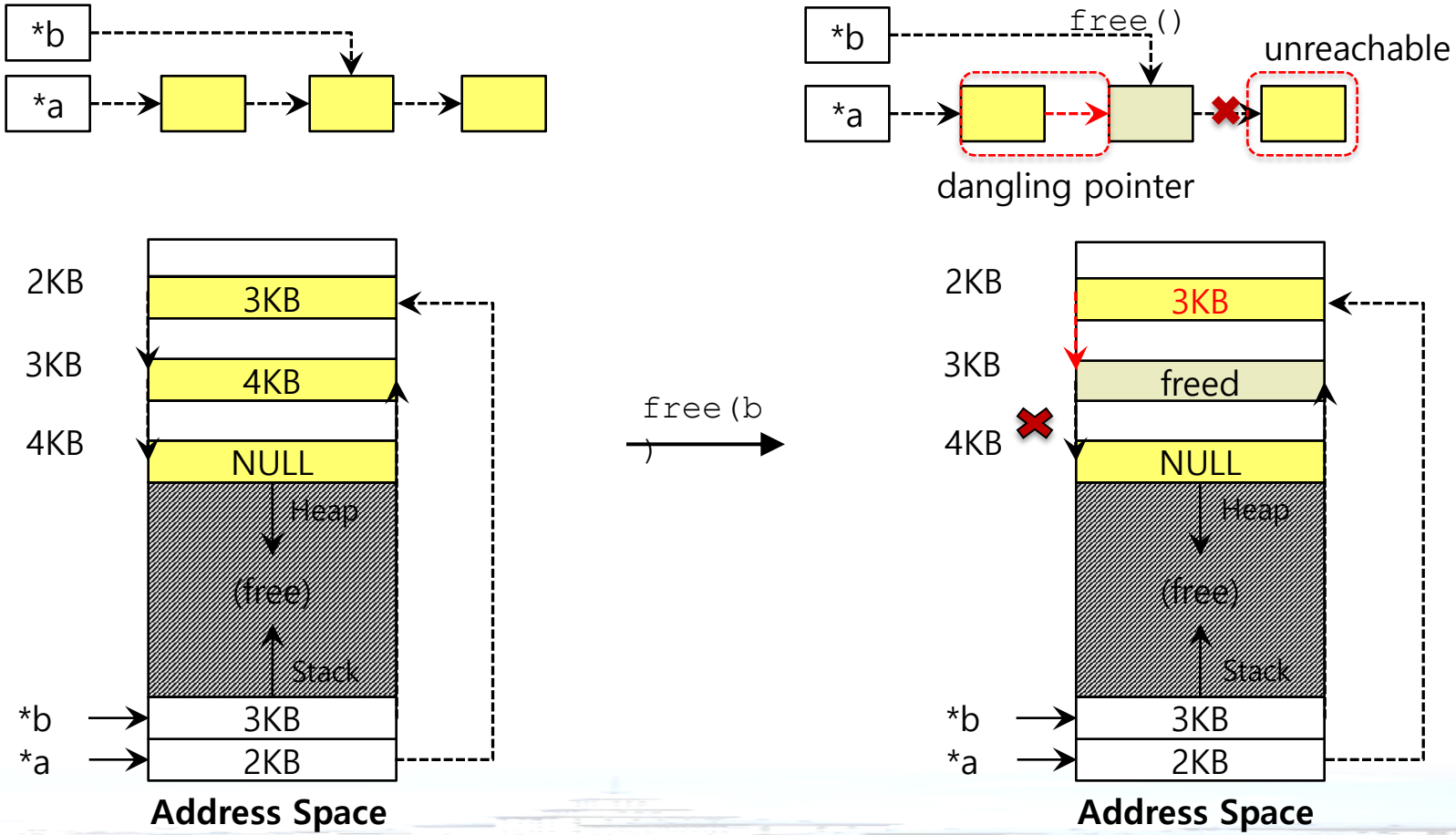
- A program runs out of memory and eventually dies.

unused : unused, but not freed



Dangling Pointer

- Freeing memory before it is finished using
  - A program accesses to memory with an invalid pointer



Other Memory APIs: calloc()

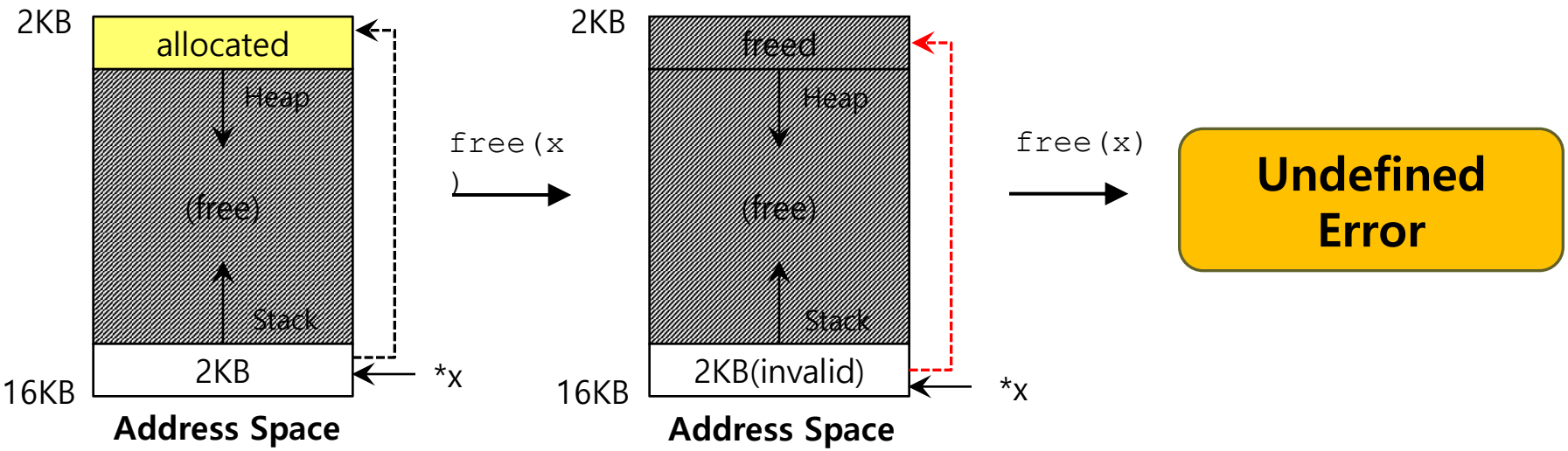
```
#include <stdlib.h>

void *calloc(size_t num, size_t size)
```

- Allocate memory on the heap and zeroes it before returning.
  - Argument
    - `size_t num` : number of blocks to allocate
    - `size_t size` : size of each block(in bytes)
  - Return
    - Success : a void type pointer to the memory block allocated by `calloc`
    - Fail : a null pointer

Double Free

- Free memory that was freed already.

```
int *x = (int *)malloc(sizeof(int)); // allocated
free(x); // free memory
free(x); // free repeatedly
```



Other Memory APIs: realloc()

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size)
```

- Change the size of memory block.
  - A pointer returned by `realloc` may be either the same as `ptr` or a new.
  - Argument
    - `void *ptr`: Pointer to memory block allocated with `malloc`, `calloc` or `realloc`
    - `size_t size`: New size for the memory block(in bytes)
  - Return
    - Success: Void type pointer to the memory block
    - Fail : Null pointer

System Calls

```
#include <unistd.h>

int brk(void *addr)
void *sbrk(intptr_t increment);
```

- `malloc` library call use `brk` system call.
  - `brk` is called to expand the program's *break*.
    - *break*: The location of **the end of the heap** in address space
  - `sbrk` is an additional call similar with `brk`.
  - Programmers **should never directly call** either `brk` or `sbrk`.

System Calls(Cont.)

```
#include <sys/mman.h>

void *mmap(void *ptr, size_t length, int port, int flags,
int fd, off_t offset)
```

- `mmap` system call can create **an anonymous** memory region.

Module 4: Virtualizing Memory 虚拟内存

1. 虚拟地址空间
2. 内存管理**API**（**Memory API** ）
3. **Mechanism**: 地址翻译（Address Translation）
4. 分段（ Segmentation ）
5. 空闲空间管理（ Free-Space Management ）
6. 分页（Paging）
7. 地址翻译缓存 TLB（Translation Lookaside Buffers）
8. 交换（**Swapping**）

Memory Virtualizing with Efficiency and Control

- Memory virtualizing takes a similar strategy known as **limited direct execution(LDE)** for efficiency and control.
- In memory virtualizing, efficiency and control are attained by hardware support.
  - e.g., registers, TLB(Translation Look-aside Buffer)s, page-table

Address Translation

- Hardware transforms a **virtual address** to a **physical address**.
  - The desired information is actually stored in a physical address.

- The OS must get involved at key points to set up the hardware.
  - The OS must manage memory to judiciously intervene.

Example: Address Translation

- C - Language code

```
void func()
        int x;
        ...
        x = x + 3; // this is the line of code we are interested in
```

  - **Load** a value from memory
  - **Increment** it by three
  - **Store** the value back into memory

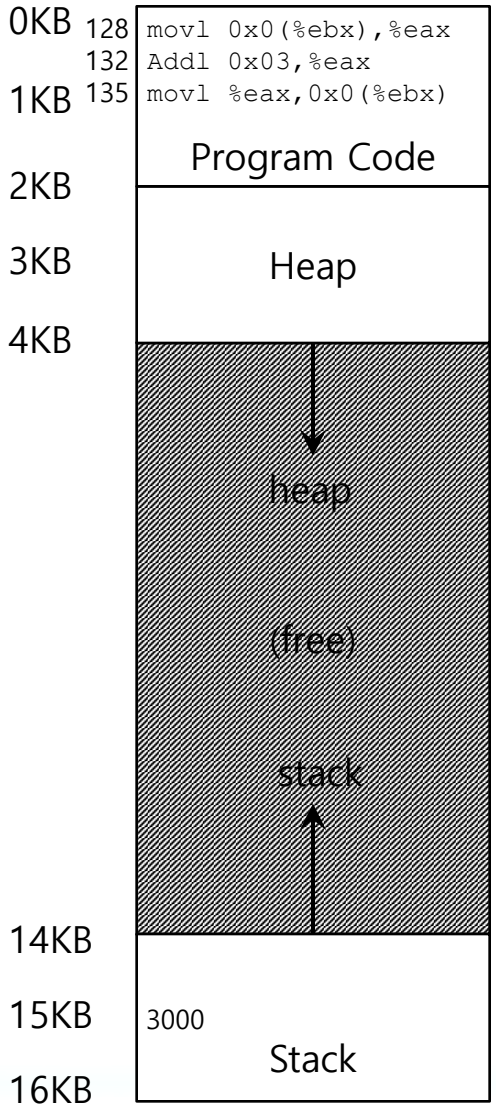
Example: Address Translation(Cont.)

- Assembly

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

  - **Load** the value at that address into `eax` register.
  - **Add** 3 to `eax` register.
  - **Store** the value in `eax` back into memory.

Example: Address Translation(Cont.)

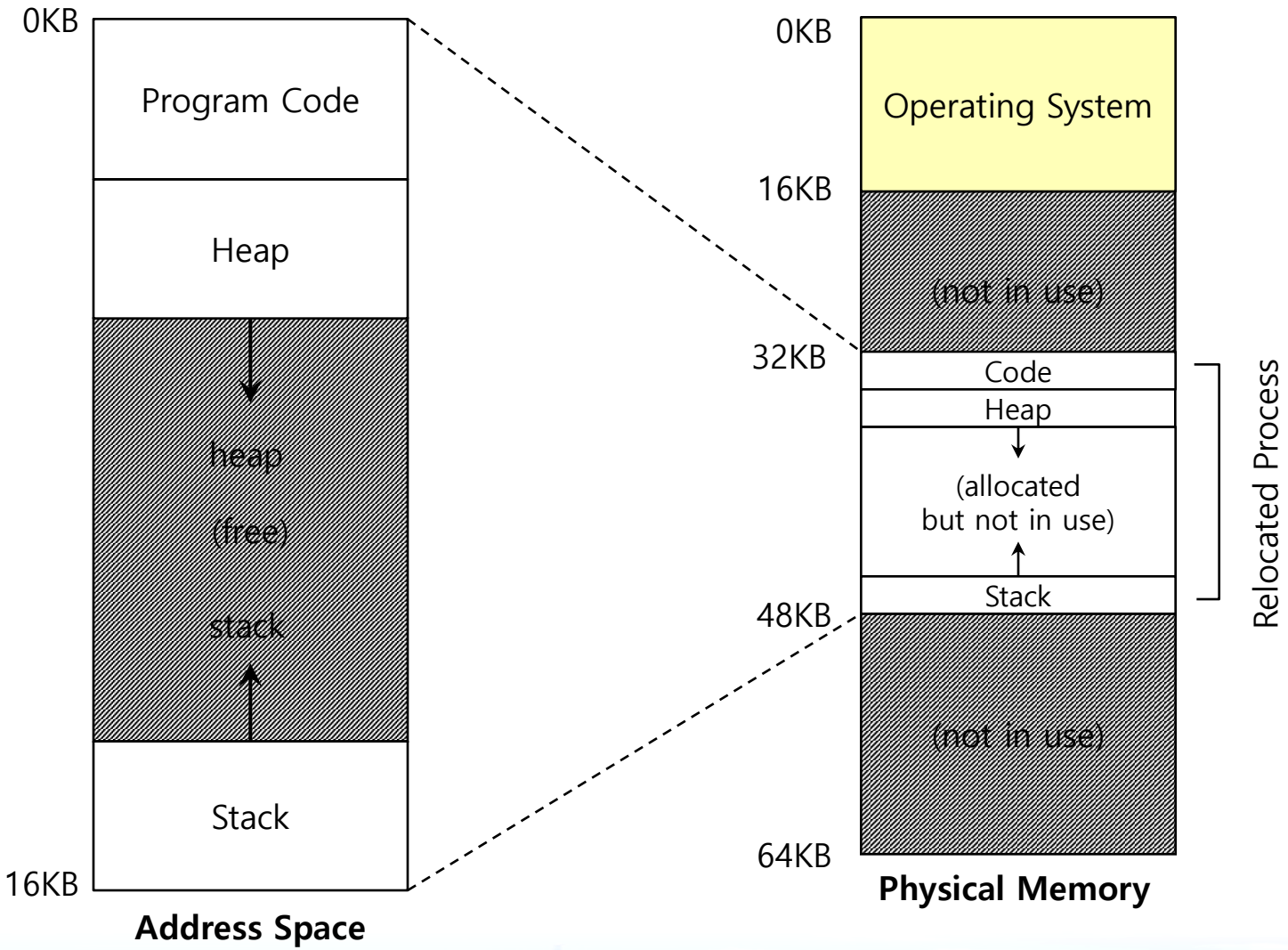


- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

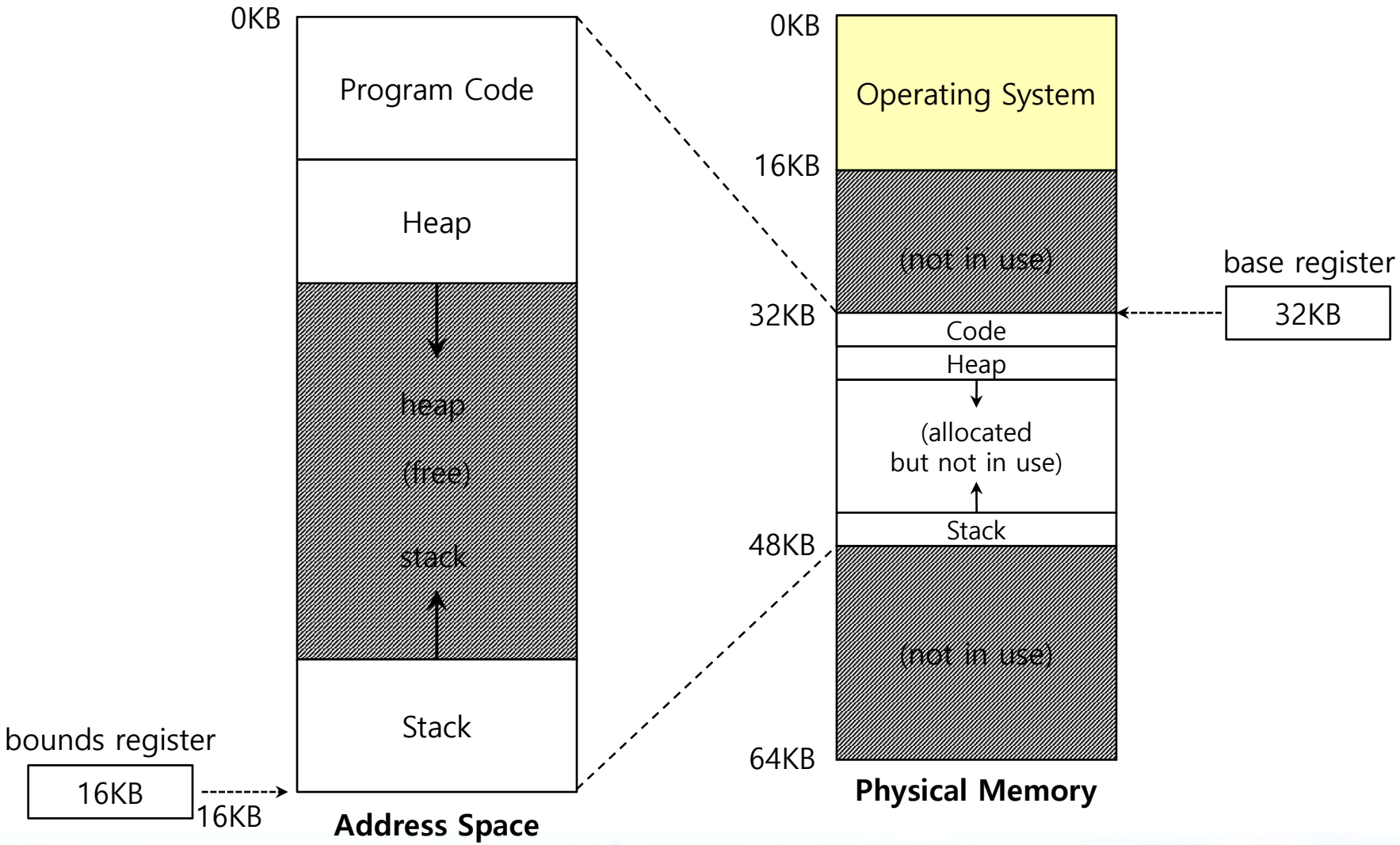
Relocation Address Space

- The OS wants to place the process **somewhere else** in physical memory, not at address 0.
  - The address space start at address 0.

A Single Relocated Process



Base and Bounds Register



Dynamic(Hardware base) Relocation

- When a program starts running, the OS decides **where** in physical memory a process should be **loaded**.
  - Set the **base** register a value.

$$phycal\ address = virtual\ address + base$$

  - Every virtual address must **not be greater than bound** and **negative.**

$$0 \leq\ virtual\ addressvirtual\ address < bounds$$

Relocation and Address Translation

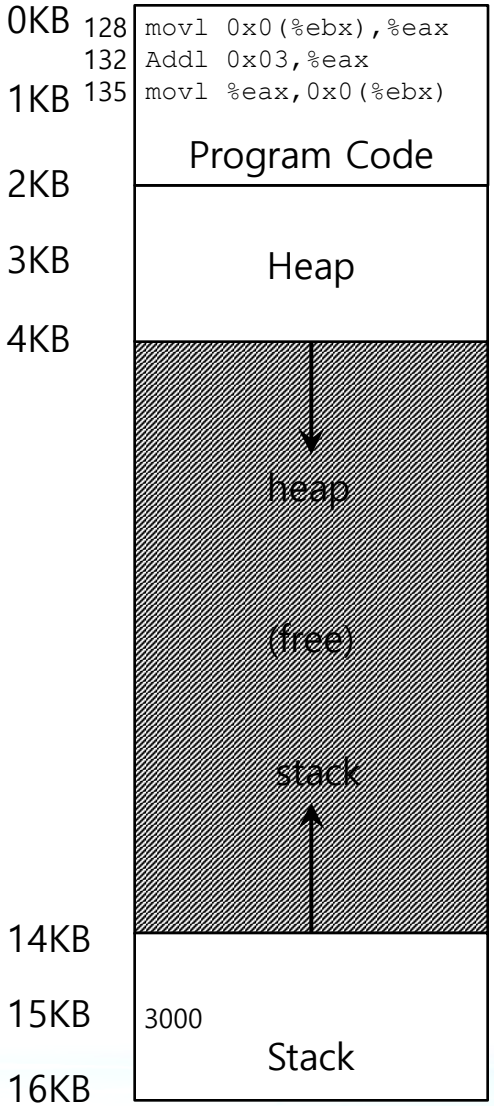
```
128 : movl 0x0(%ebx), %eax
```

- **Fetch** instruction at address 128

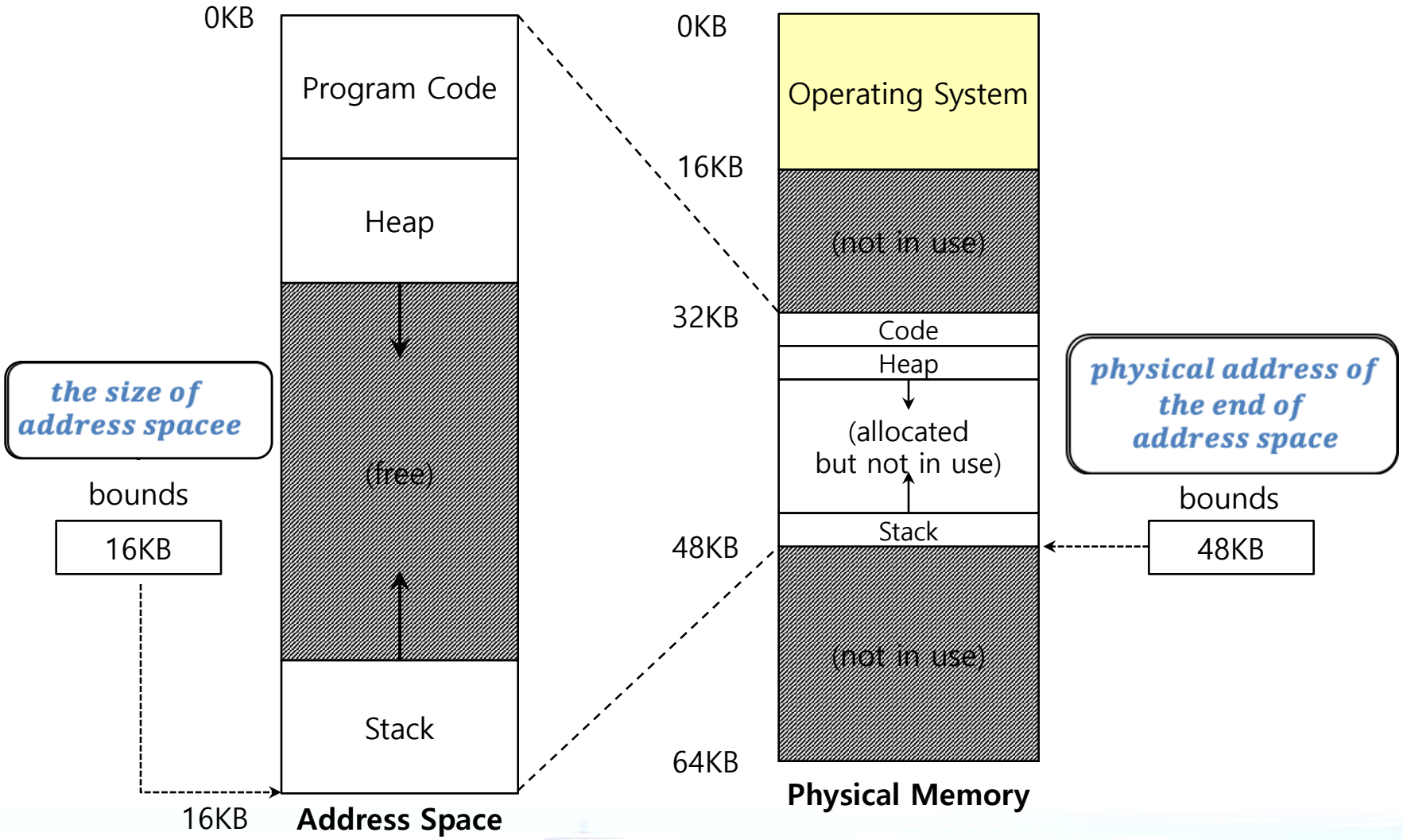
$$32896 = 128 + 32KB(base)$$

- **Execute** this instruction
  - Load from address 15KB

$$47KB = 15KB + 32KB(base)$$



Two ways of Bounds Register

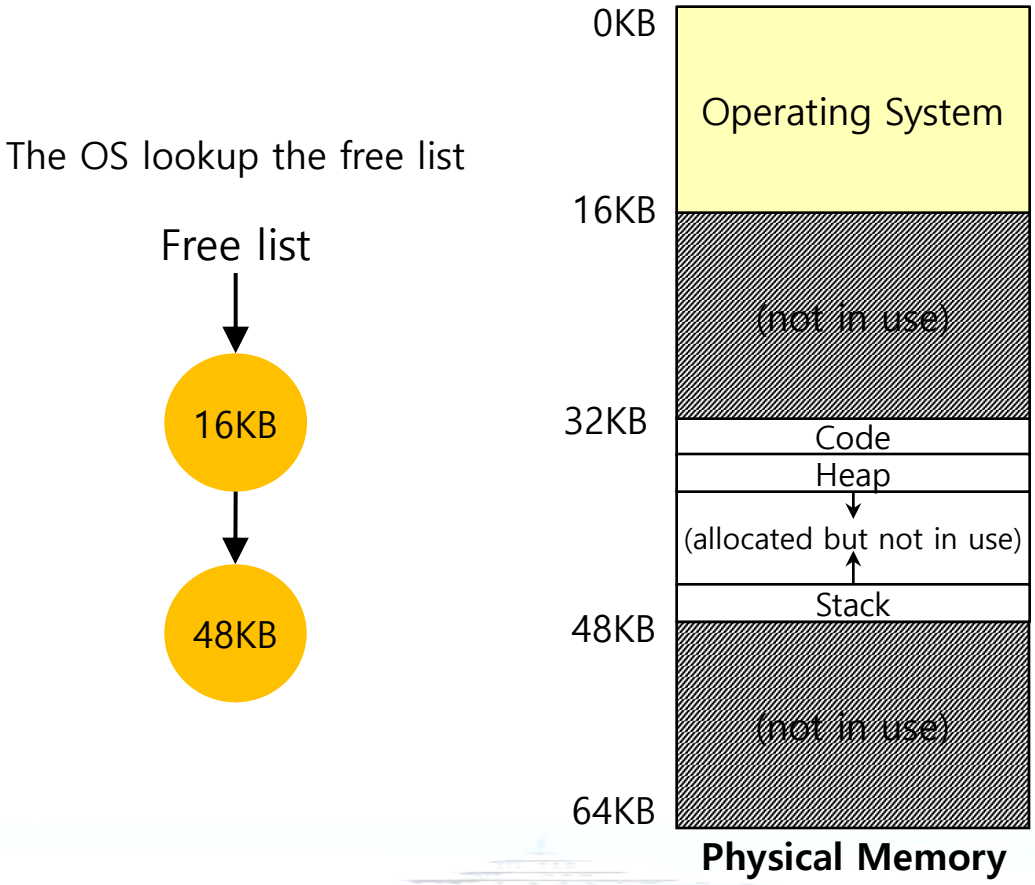


OS Issues for Memory Virtualizing

- The OS must **take action** to implement **base-and-bounds** approach.
- Three critical junctures:
  - When a process **starts running:**
    - Finding space for address space in physical memory
  - When a process is **terminated:**
    - Reclaiming the memory for use
  - When context **switch occurs:**
    - Saving and storing the base-and-bounds pair

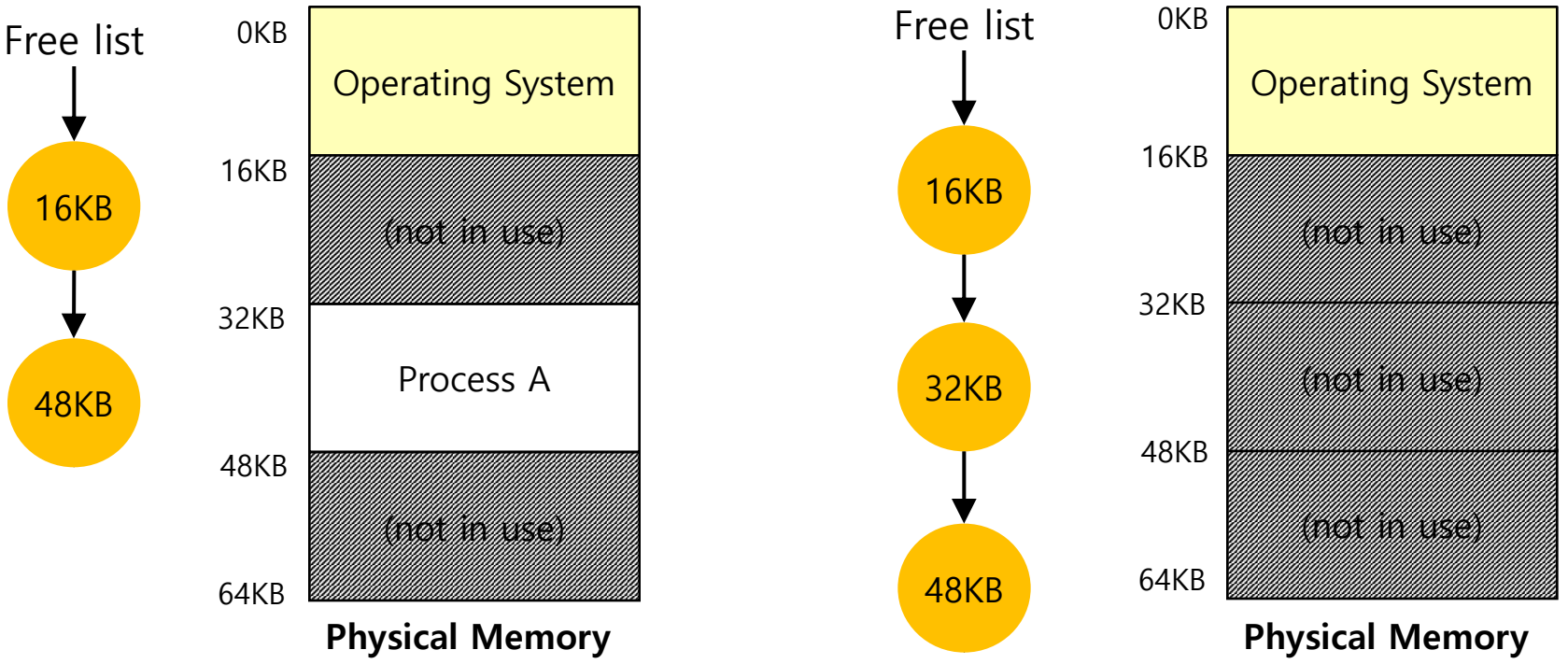
OS Issues: When a Process Starts Running

- The OS must **find a room** for a new address space.
  - free list : A list of the range of the physical memory which are not in use.



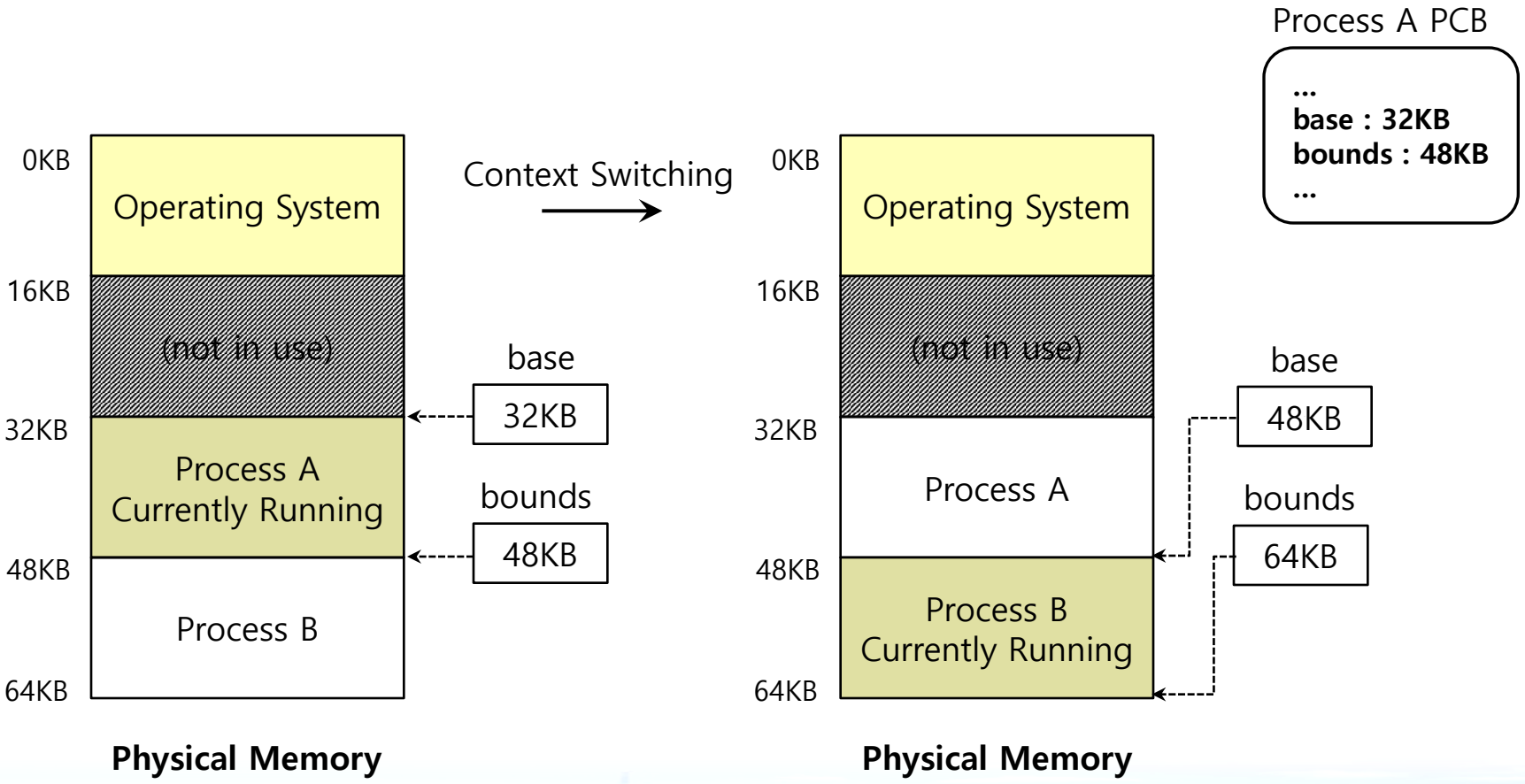
OS Issues: When a Process Is Terminated

- The OS must **put the memory back** on the free list.



OS Issues: When Context Switch Occurs

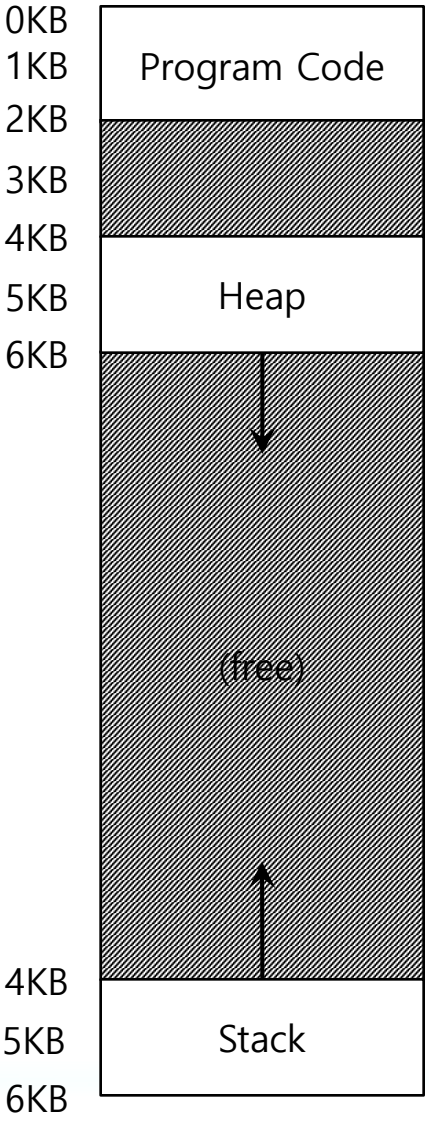
- The OS must **save and restore** the base-and-bounds pair.
  - In **process structure** or **process control block(**PCB)



Module 4: Virtualizing Memory 虚拟内存

1. 虚拟地址空间
2. 内存管理**API**（**Memory API** ）
3. **Mechanism**: 地址翻译（Address Translation）
4. 分段（ **Segmentation** ）
5. 空闲空间管理（ Free-Space Management ）
6. 分页（Paging）
7. 地址翻译缓存 TLB（Translation Lookaside Buffers）
8. 交换（**Swapping**）

Inefficiency of the Base and Bound Approach



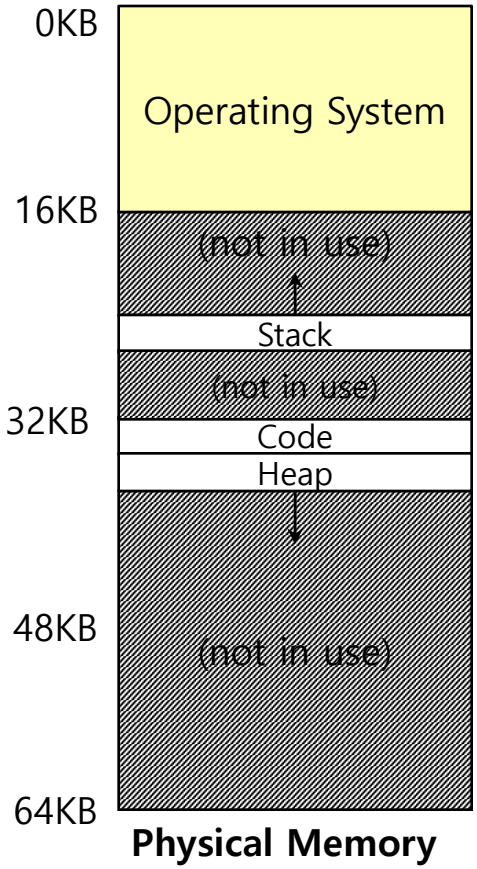
- **Big chunk of "free"** space
- "free" space **takes up** physical memory.
- Hard to run when an address space **does not fit** into physical memory

Segmentation

- Segment is just **a contiguous portion** of the address space of a particular length.
  - Logically-different segment: code, stack, heap

- Each segment can be **placed** in **different part of physical memory**.
  - **Base** and **bounds** exist **per each segment**.

Placing Segment In Physical Memory

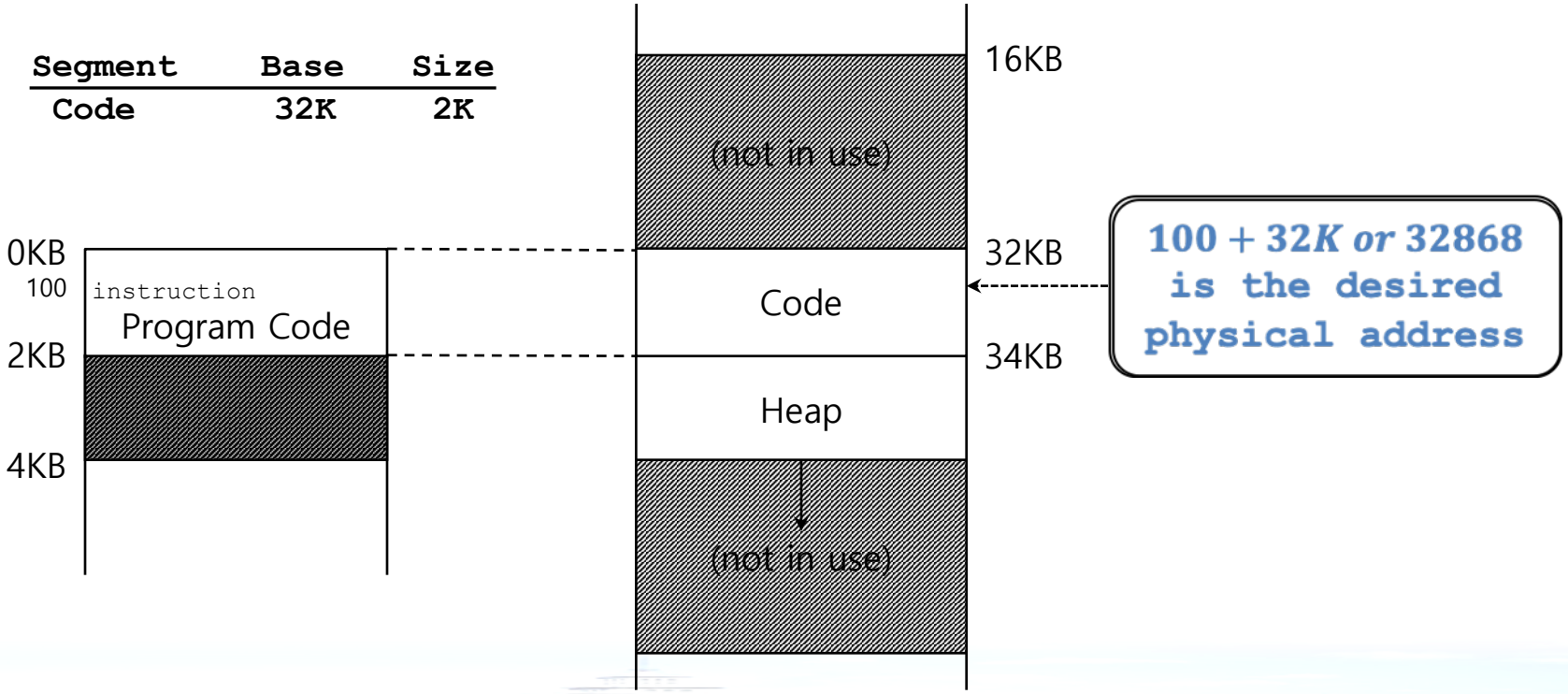


Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Address Translation on Segmentation

$$physical\ address = offset + base$$

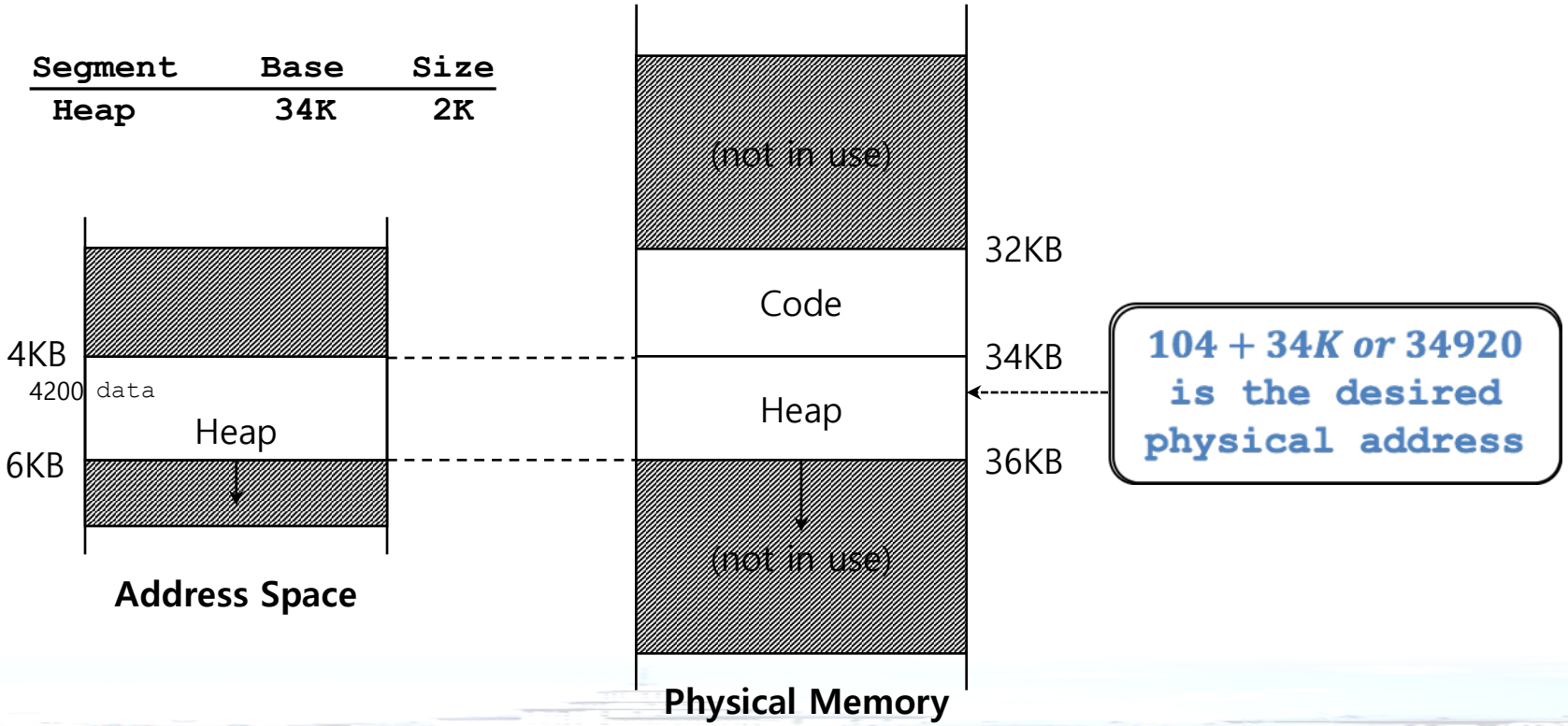
- The `offset` of virtual address 100 is `100`.
  - The code segment **starts at virtual address 0** in address space.



Address Translation on Segmentation(Cont.)

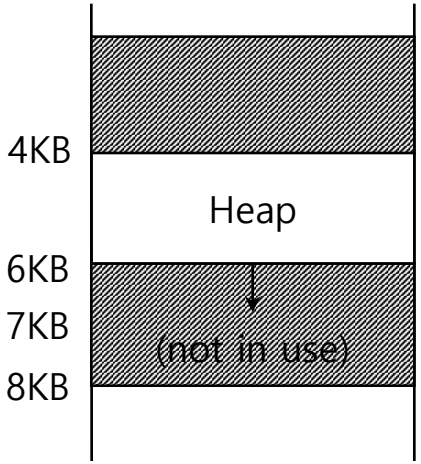
***Virtual address + base* is not the correct physical address.**

- The `offset` of virtual address 4200 is `104`.
  - The heap segment **starts at virtual address 4096** in address space.



Segmentation Fault or Violation

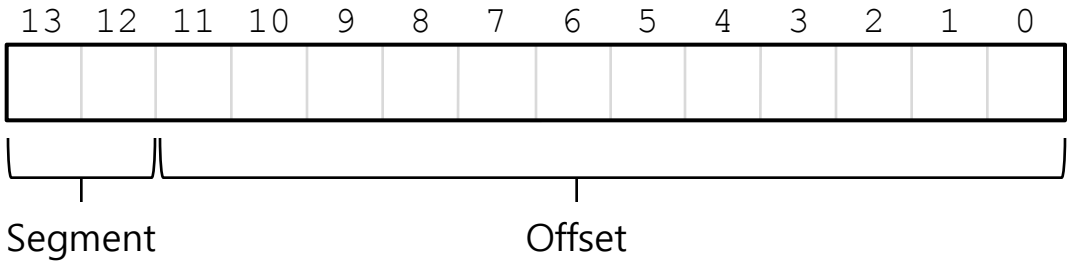
- If an **illegal address** such as 7KB which is beyond the end of heap is referenced, the OS occurs **segmentation fault**.
  - The hardware detects that address is **out of bounds**.



Address Space

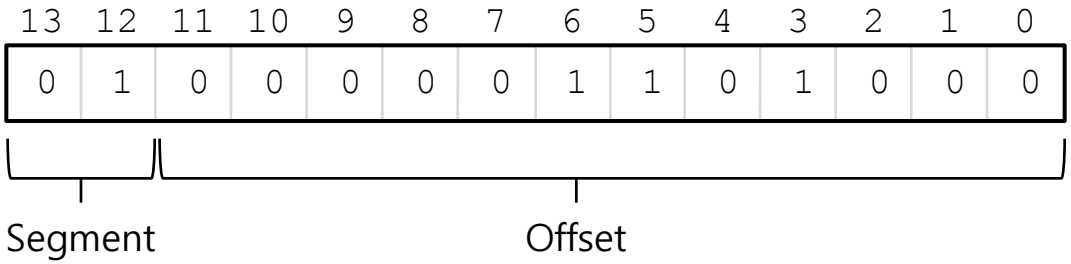
Referring to Segment

- **Explicit approach**
  - Chop up the address space into segments based on the **top few bits** of virtual address.



- Example: virtual address 4200 (01000001101000)

Segment	bits
Code	00
Heap	01
Stack	10
-	11



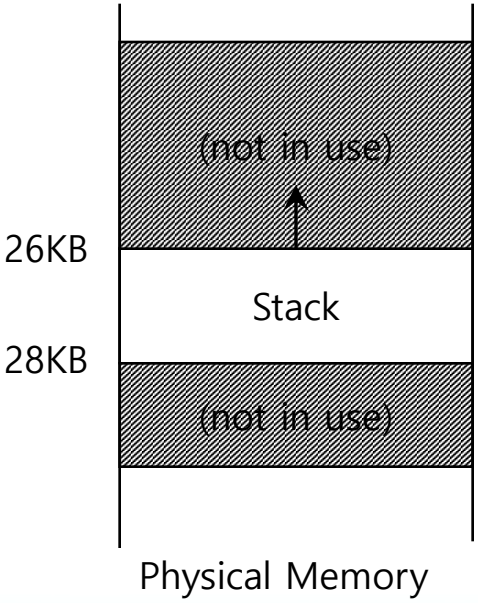
Referring to Segment(Cont.)

```
1   // get top 2 bits of 14-bit VA
2   Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3   // now get offset
4   Offset = VirtualAddress & OFFSET_MASK
5   if (Offset >= Bounds[Segment])
6        RaiseException(PROTECTION_FAULT)
7   else
8        PhysAddr = Base[Segment] + Offset
9        Register = AccessMemory(PhysAddr)
```

- SEG_MASK = 0x3000(11000000000000)
- SEG_SHIFT = 12
- OFFSET_MASK = 0xFFF (00111111111111)

Referring to Stack Segment

- Stack grows **backward**.
- **Extra hardware support** is needed.
  - The hardware checks which way the segment grows.
  - 1: positive direction, 0: negative direction



Segment Register(with Negative-Growth Support)

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

Support for Sharing

- Segment can be **shared between address** space.
  - **Code sharing** is still in use in systems today.
  - by extra hardware support.
- Extra hardware support is needed in form of **Protection bits.**
  - **A few more bits** per segment to indicate **permissions** of **read,** write and **execute**.

Segment Register Values(with Protection)

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

Fine-Grained and Coarse-Grained

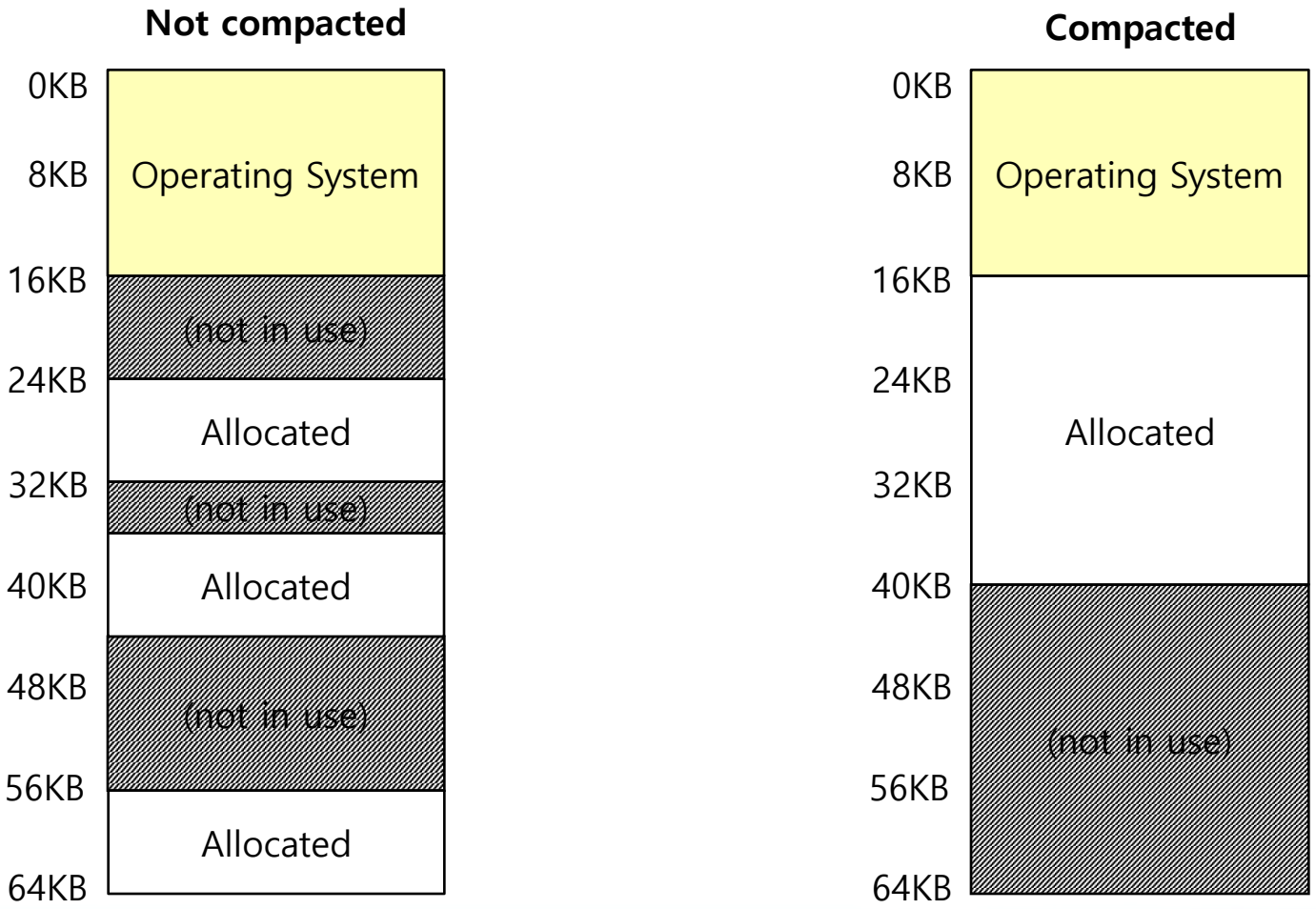
- **Coarse-Grained** means segmentation in a small number.
  - e.g., code, heap, stack.
- **Fine-Grained** segmentation allows **more flexibility** for address space in some early system.
  - To support many segments, Hardware support with a **segment table** is required.

OS support: Fragmentation

- **External Fragmentation**: little holes of **free space** in physical memory that make difficulty to allocate new segments.
  - There is **24KB free**, but **not in one contiguous** segment.
  - The OS **cannot** satisfy the **20KB request**.

- **Compaction**: **rearranging** the exiting segments in physical memory.
  - Compaction is **costly**.
    - **Stop** running process.
    - **Copy** data to somewhere.
    - **Change** segment register value.

Memory Compaction

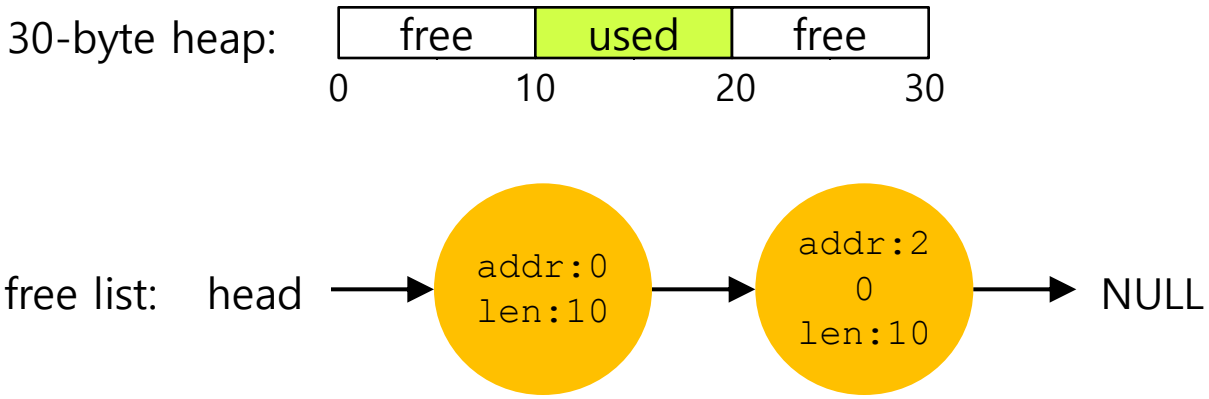


Module 4: Virtualizing Memory 虚拟内存

1. 虚拟地址空间
2. 内存管理API（Memory API ）
3. **Mechanism**: 地址翻译（Address Translation）
4. 分段（ **Segmentation** ）
5. 空闲空间管理（ **Free-Space Management** ）
6. 分页（Paging）
7. 地址翻译缓存 TLB（Translation Lookaside Buffers）
8. 交换（**Swapping**）

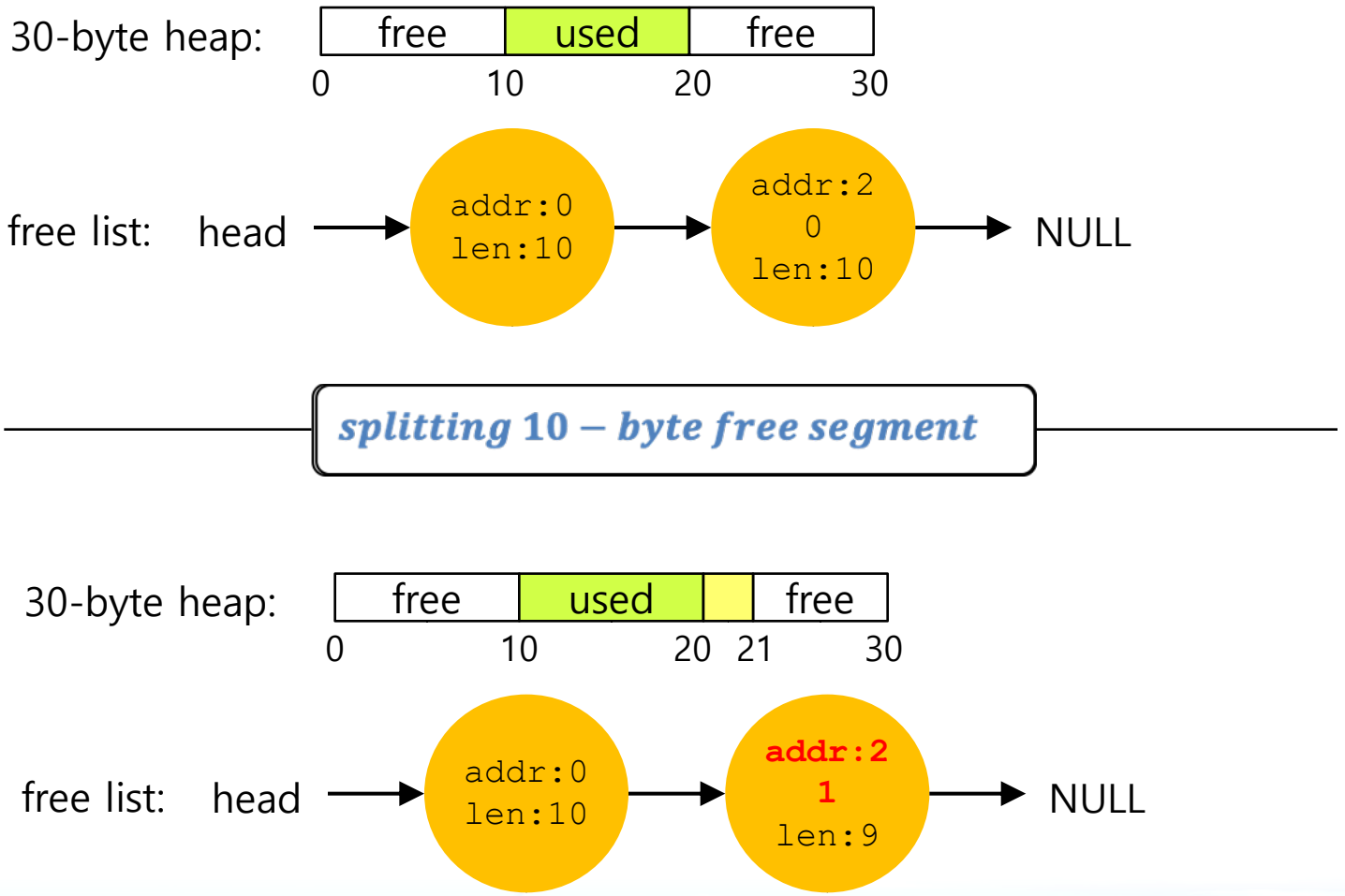
Splitting

- Finding a free chunk of memory that can satisfy the request and splitting it into two.
  - When request for memory allocation is **smaller** than the size of free chunks.



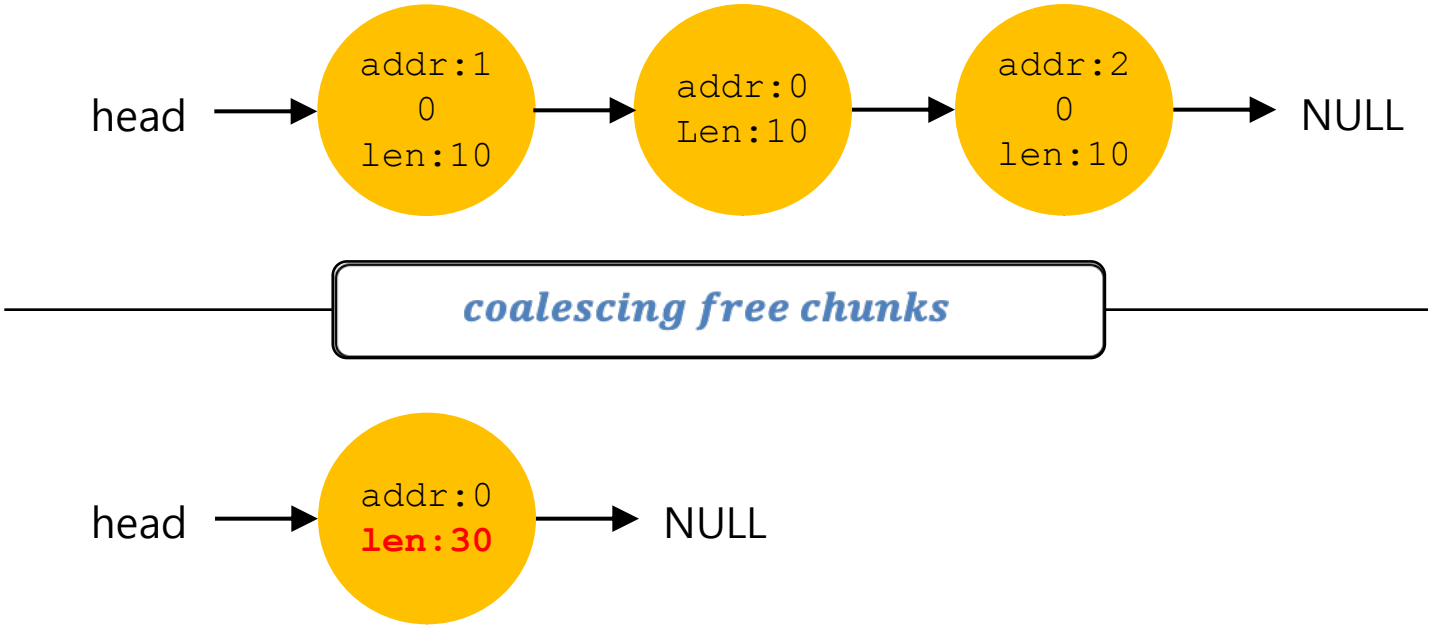
Splitting(Cont.)

- Two 10-bytes free segment with **1-byte request**



Coalescing

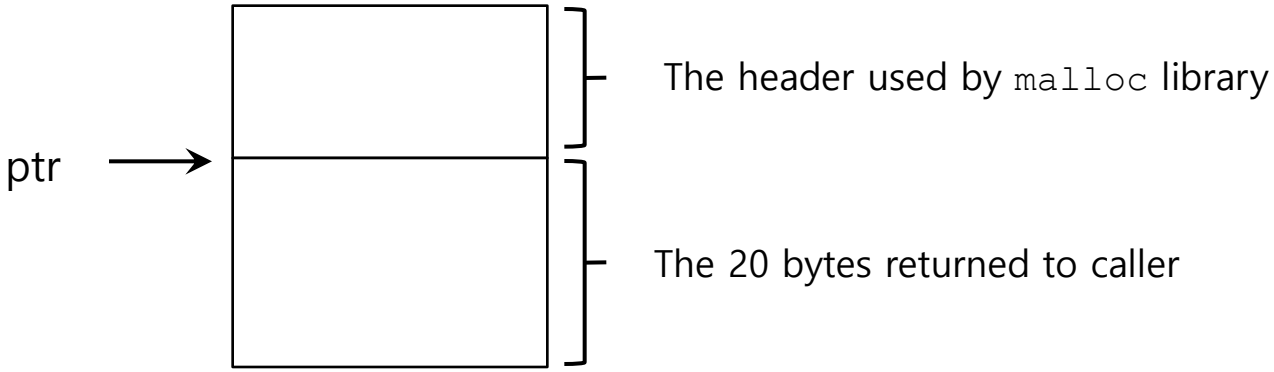
- If a user requests memory that is **bigger than free chunk size**, the list will **not find** such a free chunk.
- Coalescing: **Merge** returning a free chunk with existing chunks into a large single free chunk if **addresses** of them are **nearby**.



Tracking The Size of Allocated Regions

- The interface to `free(void *ptr)` does **not take** a **size parameter**.
  - How does the library **know the size** of memory region that will be back **into free list**?
- Most allocators store **extra information** in a **header** block.

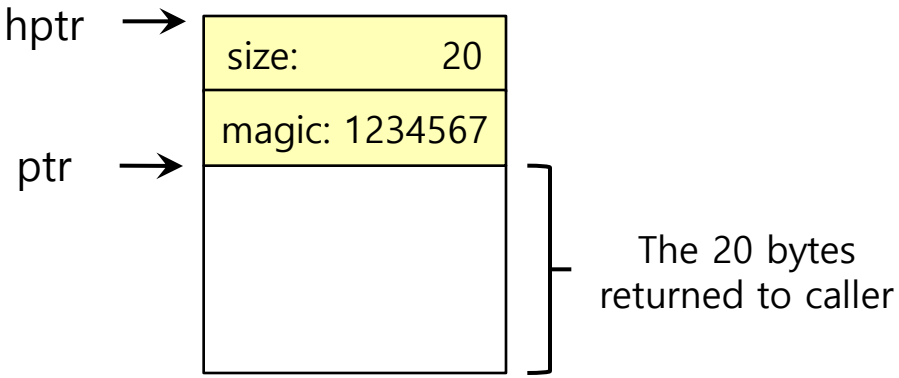
```
ptr = malloc(20);
```



An Allocated Region Plus Header

The Header of Allocated Memory Chunk

- The header minimally **contains the size** of the allocated memory region.
- The header may also contain
  - Additional pointers to speed up deallocation
  - A magic number for integrity checking



Specific Contents Of The Header

```
typedef struct __header_t {
        int size;
        int magic;
} header_t;
```

A Simple Header

The Header of Allocated Memory Chunk(Cont.)

- The **size** for free region is the **size of the header plus the size of the space** allocated to the user.
- If a user **request `N` bytes**, the library searches for a free chunk of **size `N` plus the size of the header**
- Simple pointer arithmetic to find the header pointer.

```
void free(void *ptr) {
        header_t *hptr = (void *)ptr - sizeof(header_t);
}
```

Embedding A Free List

- The memory-allocation library **initializes** the heap and **puts** the first element of **the free list** in the **free space**.
  - The library **can't use** `malloc()` to build a list **within itself**.

Embedding A Free List(Cont.)

- Description of a node of the list

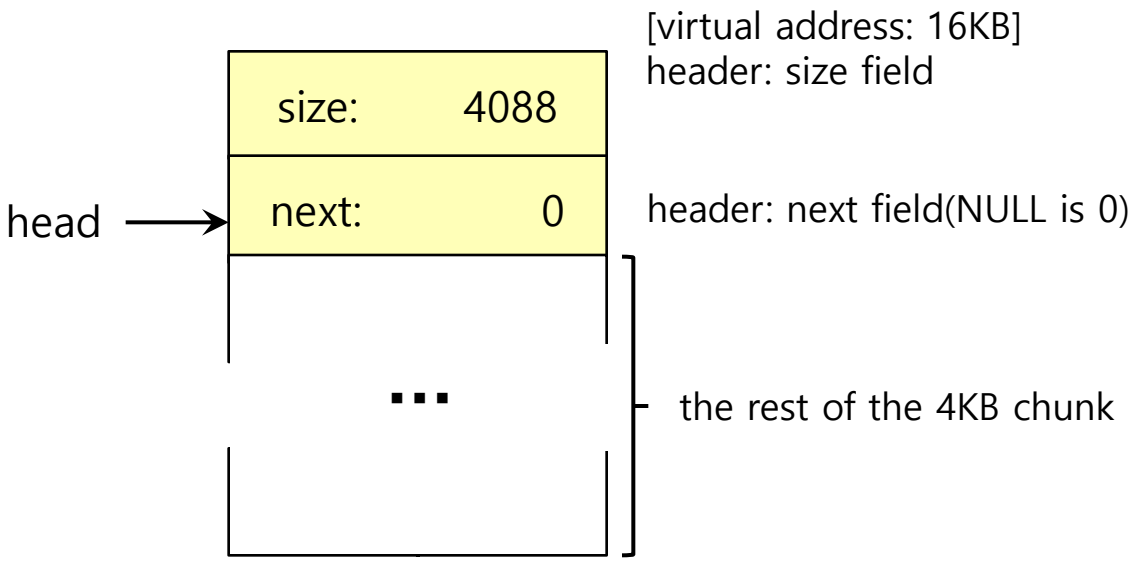
```
typedef struct __node_t {
        int size;
        struct __node_t *next;
} nodet_t;
```

- Building heap and putting a free list
  - Assume that the heap is built vi `mmap()` system call.

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                                MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```

A Heap With One Free Chunk

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                                MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```



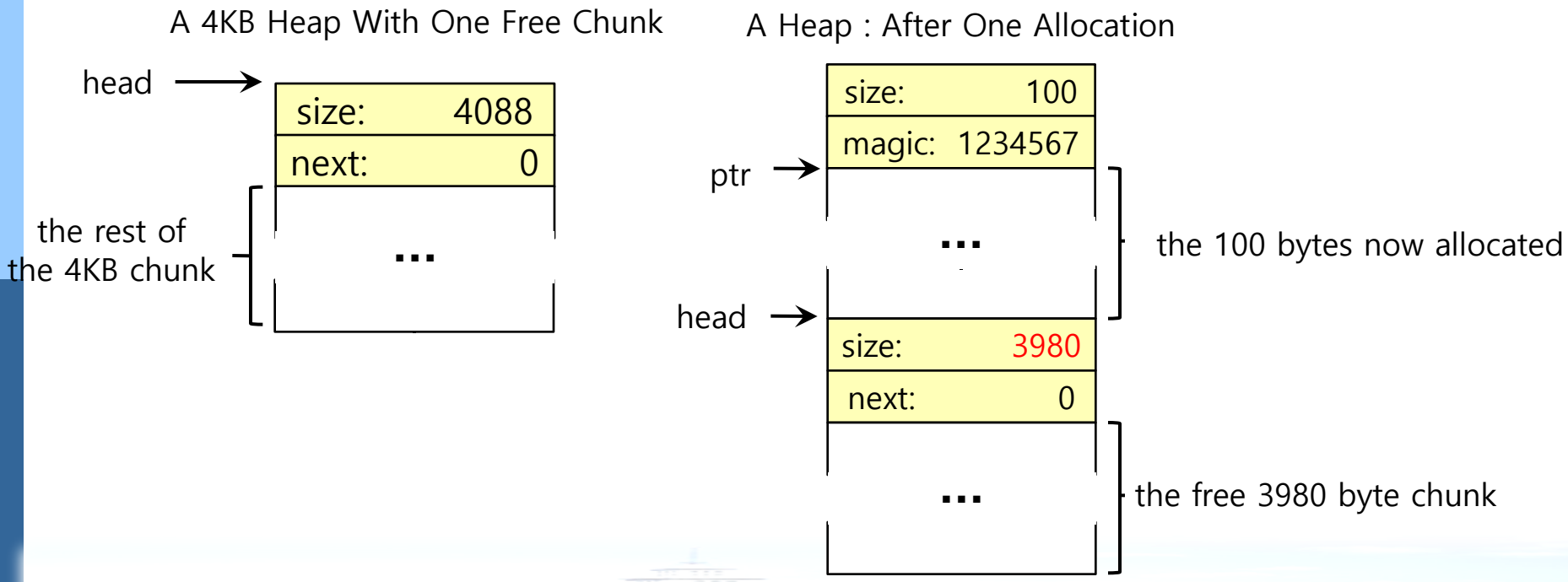
Embedding A Free List: Allocation

- If a chunk of memory is requested, the library **will first find** a chunk that is **large enough** to accommodate the request.

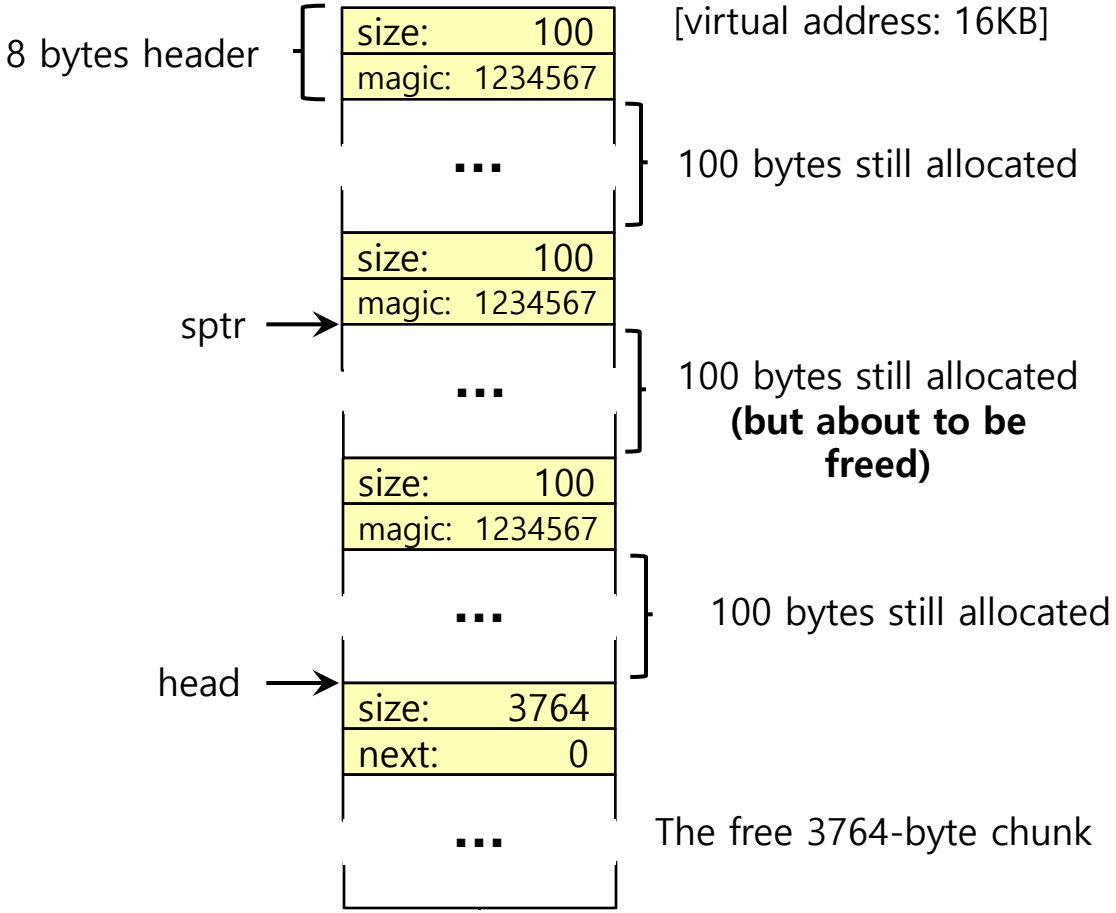
- The library will
  - **Split** the large free chunk into two.
    - **One** for the **request** and the **remaining** free chunk
  - **Shrink** the size of free chunk in the list.

Embedding A Free List: Allocation(Cont.)

- Example: a request for 100 bytes by `ptr = malloc(100)`
  - Allocating 108 bytes out of the existing one free chunk.
  - shrinking the one free chunk to 3980(4088 minus 108).



Free Space With Chunks Allocated

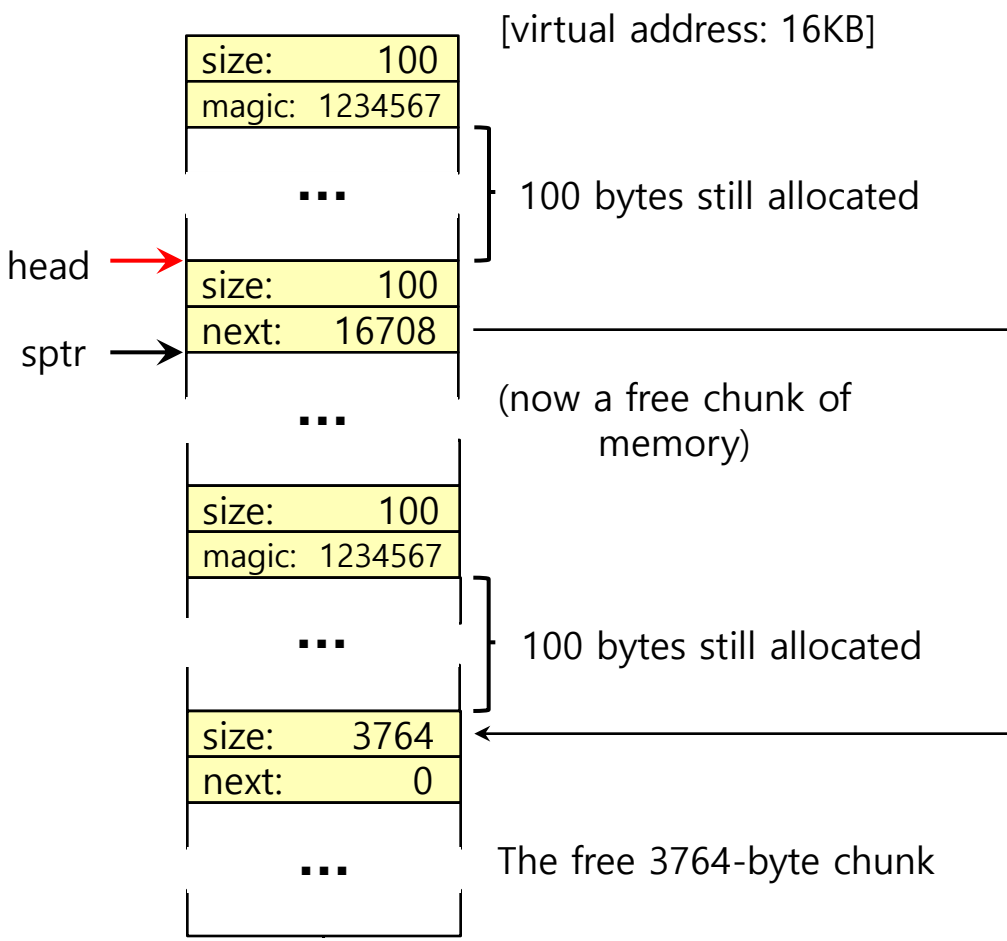


Free Space With Three Chunks Allocated

Free Space With `free()`

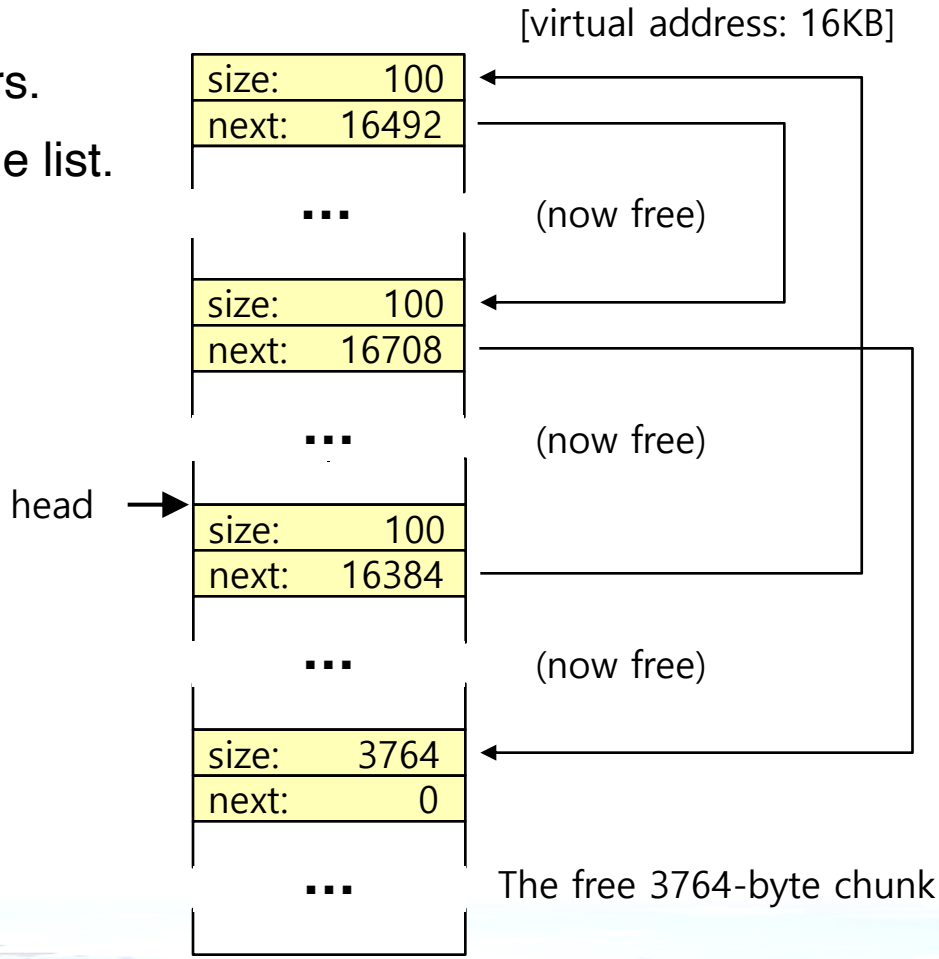
- Example: `free(sptr)`

- The 100 bytes chunks is **back into** the free list.
- The free list will **start** with **a small chunk**.
  - The list header will point the small chunk



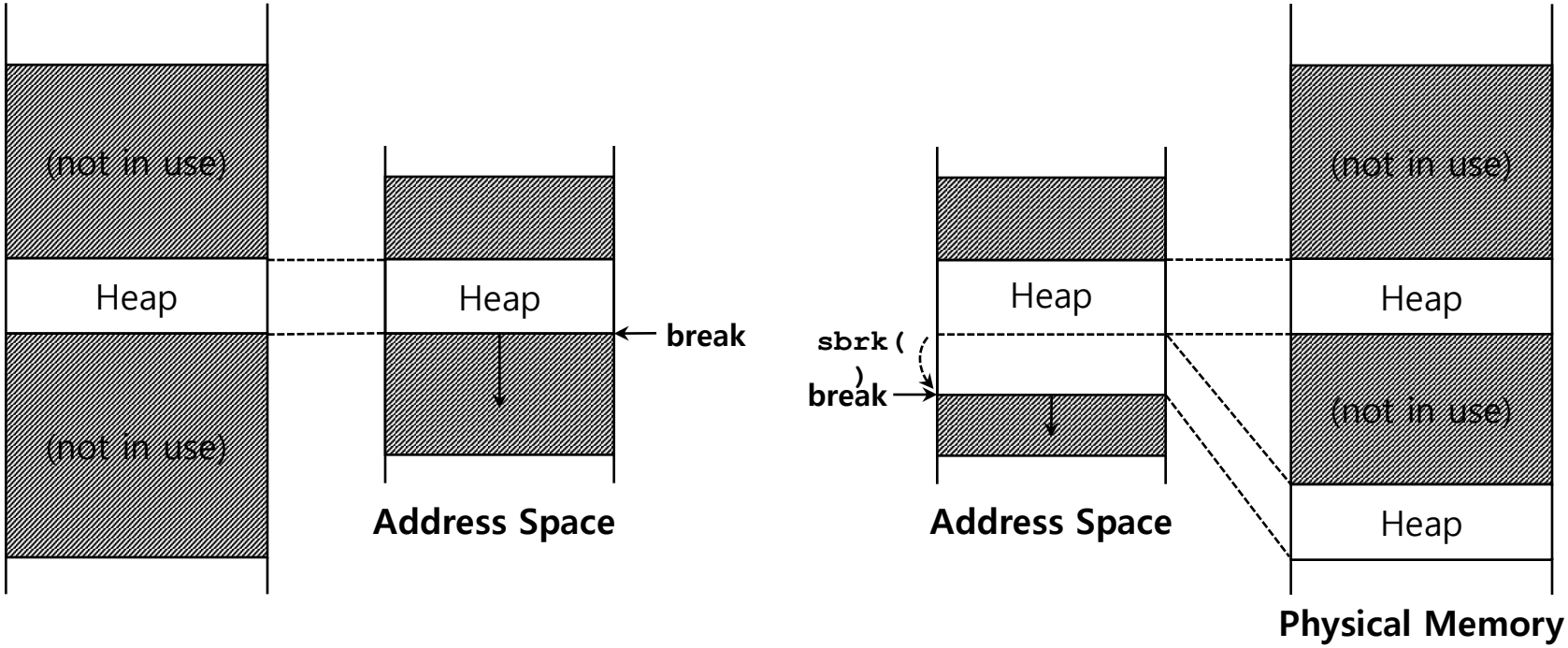
Free Space With Freed Chunks

- Let's assume that the last two in-use chunks are freed.
- **External Fragmentation** occurs.
  - **Coalescing** is needed in the list.



Growing The Heap

- Most allocators **start** with **a small-sized heap** and then **request more** memory from the OS when they run out.
  - e.g., `sbrk()`, `brk()` in most UNIX systems.



Managing Free Space: Basic Strategies

- Best Fit:
  - Finding free chunks that are **big or bigger than the request**
  - Returning the **one of smallest** in the chunks **in the group** of candidates

- Worst Fit:
  - Finding the **largest free chunks** and allocation the amount of the request
  - **Keeping the remaining chunk** on the free list.

Managing Free Space: Basic Strategies(Cont.)

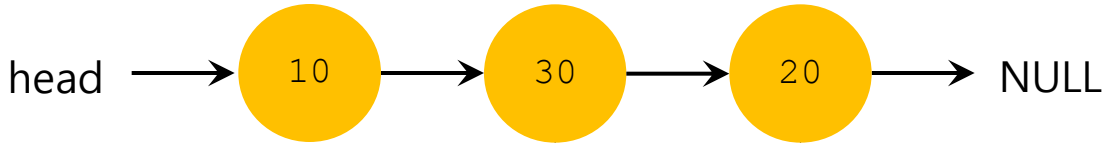
- First Fit:
  - Finding the **first chunk** that is **big enough** for the request
  - Returning the requested amount and remaining the rest of the chunk.

- Next Fit:
  - Instead of always beginning the first-fit search at the beginning of the list
  - Keep an **extra pointer** to the location within the list where one was **looking last**. 即搜索的起始点是上次返回的节点位置

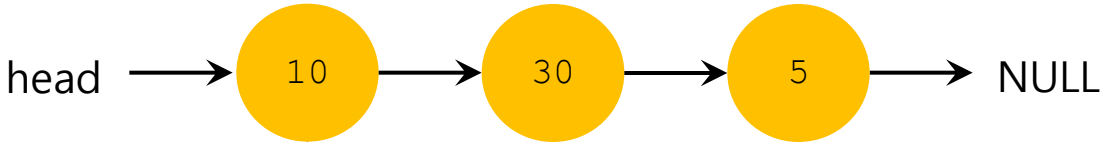
  - The idea is to **spread the searches for free space throughout the list more uniformly**, thus avoiding splintering of the beginning of the list

Examples of Basic Strategies

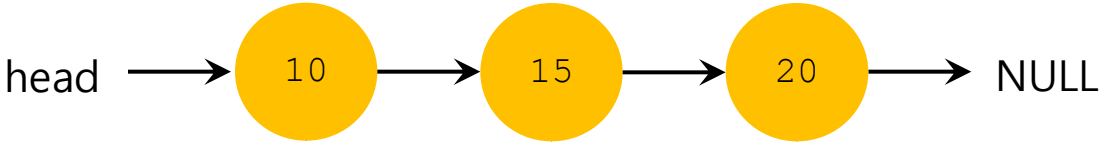
- Allocation Request Size 15



- Result of Best-fit



- Result of Worst-fit



Other Approaches: Segregated List

- Segregated List:
  - Keeping free chunks in different size in a separate list for the size of popular request.
  - New Complication:
    - **How much** memory should dedicate to **the pool of memory** that serves **specialized requests** of a given size?
  - **Slab allocator** handles this issue.

Other Approaches: Segregated List(Cont.)

- Slab Allocator
  - Allocate a number of object caches.
    - The objects are likely to e requested frequently.
    - e.g., locks, file-system inodes, etc.
  - **Request some memory** from a more general memory allocator when **a given cache is running low** on free space.

Other Approaches: Segregated List(Cont.)

- Slab
  - The slab is the primary unit of currency in the slab allocator.
  - A separate memory pool for frequently created & deleted objects of same size.
    - e.g., task_struct, inode in Linux, etc

Other Approaches: Segregated List(Cont.)

- Slab
  - Consists of one or more pages of virtually contiguous memory carved up into equal-size chunks
  - With a reference count indicating how many of those chunks have been allocated

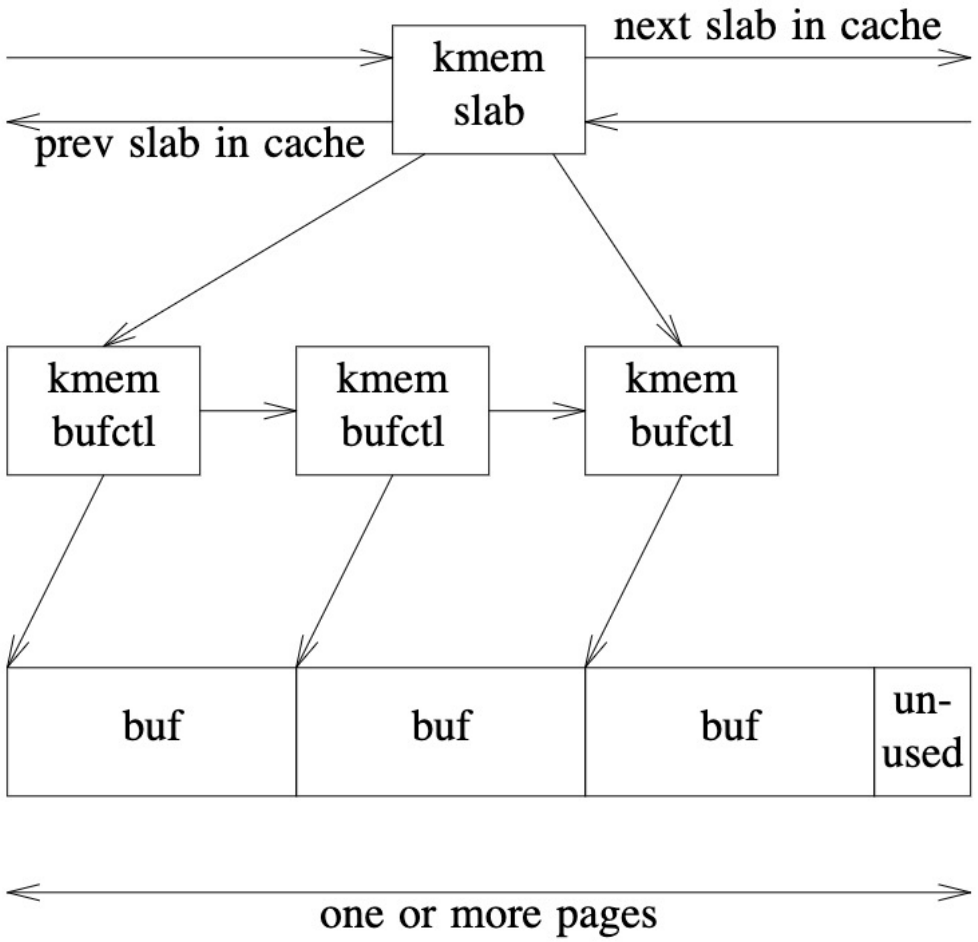
Other Approaches: Segregated List(Cont.)

- Slab
  - Benefits:
    - Reclaiming unused memory is trivial.
    - Allocating and freeing memory are fast, constant-time operations.
    - Severe external fragmentation (unused buffers on the freelist) is unlikely.
    - Internal fragmentation (per-buffer wasted space) is minimal.

Other Approaches: Segregated List(Cont.)

- `kmem_slab`: Each slab is managed, which maintains the slab's linkage in the cache, its reference count, and its list of free buffers.

- `kmem_bufctl`: Each buffer in the slab is managed, which holds the freelist linkage, buffer address, and a back pointer to the controlling slab.



(bufctl-to-slab back-pointers not shown)

The Slab Allocator original paper

哈爾濱工業大學 HARBIN INSTITUTE OF TECHNOLOGY

The Slab Allocator: An Object-Caching Kernel Memory Allocator

Jeff Bonwick
Sun Microsystems

Abstract

This paper presents a comprehensive design overview of the SunOS 5.4 kernel memory allocator. This allocator is based on a set of object-caching primitives that reduce the cost of allocating complex objects by retaining their state between uses. These same primitives prove equally effective for managing stateless memory (e.g. data pages and temporary buffers) because they are space-efficient and fast. The allocator's object caches respond dynamically to global memory pressure, and employ an object-coloring scheme that improves the system's overall cache utilization and bus balance. The allocator also has several statistical and debugging features that can detect a wide range of problems throughout the system.

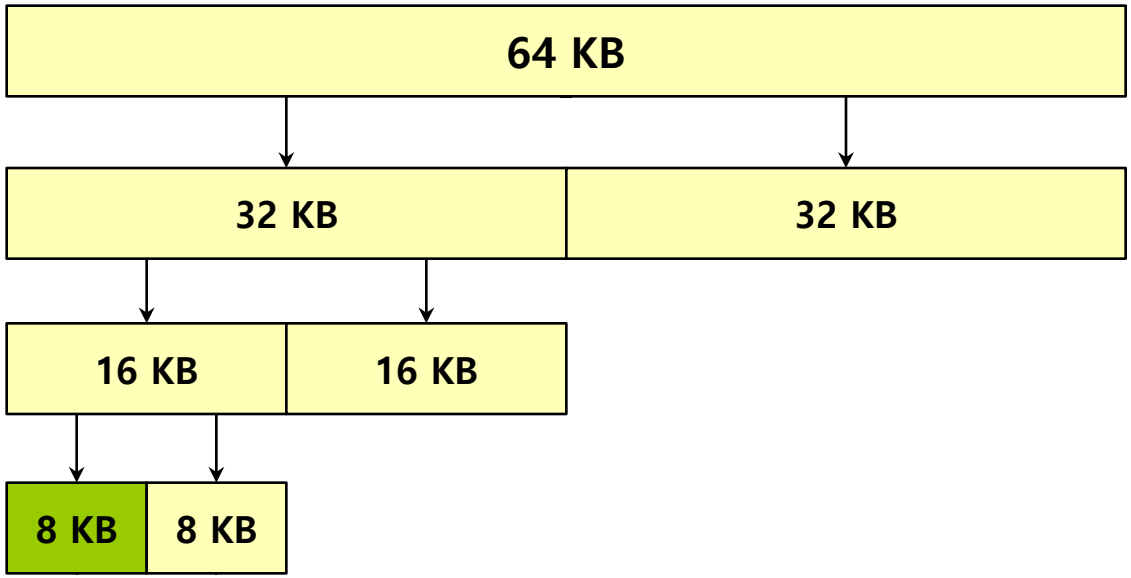
generally superior in both space *and* time. Finally, Section 6 describes the allocator's debugging features, which can detect a wide variety of problems throughout the system.

2. Object Caching

Object caching is a technique for dealing with objects that are frequently allocated and freed. The idea is to preserve the invariant portion of an object's initial state — its *constructed* state — between uses, so it does not have to be destroyed and recreated every time the object is used. For example, an object containing a mutex only needs to have `mutex_init()` applied once — the first time the object is allocated. The object can then be freed and reallocated many times without incurring

Other Approaches: Buddy Allocation

- Binary Buddy Allocation
  - The allocator **divides free space** by two **until a block** that is big enough to accommodate the request is **found**.



64KB free space for 7KB request

Other Approaches: Buddy Allocation(Cont.)

- Buddy allocation can suffer from **internal fragmentation**.
- Buddy system makes **coalescing** simple.
  - **Coalescing** two blocks in to the next level of block.
  - 判断任意block的buddy block是否in use很简单：**地址只有1 bit区别**

Module 4: Virtualizing Memory 虚拟内存

1. 虚拟地址空间
2. 内存管理API（**Memory API** ）
3. **Mechanism**: 地址翻译（Address Translation）
4. 分段（ **Segmentation** ）
5. 空闲空间管理（ Free-Space Management ）
6. 分页（**Paging**）
7. 地址翻译缓存 TLB（Translation Lookaside Buffers）
8. 交换（**Swapping**）

Concept of Paging

- Paging **splits up** address space into **fixed-zed** unit called a **page**.
  - 回顾对比 **Segmentation**: variable size of logical segments(code, stack, heap, etc.)
- With paging, **physical memory** is also **split** into some number of pages called a **page frame**.
- **Page table** per process is needed **to translate** the virtual address to physical address.

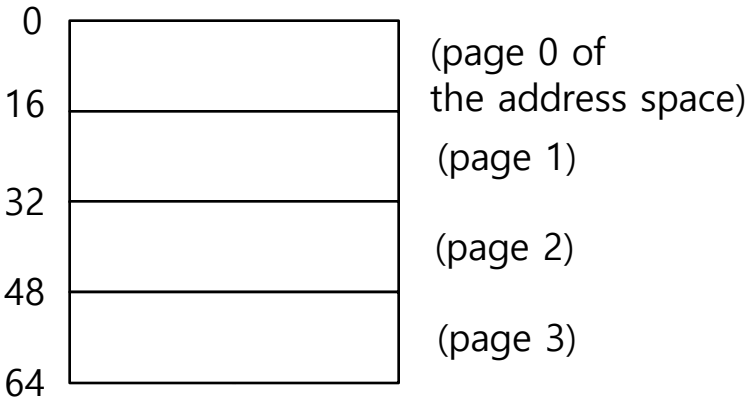
Advantages Of Paging

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

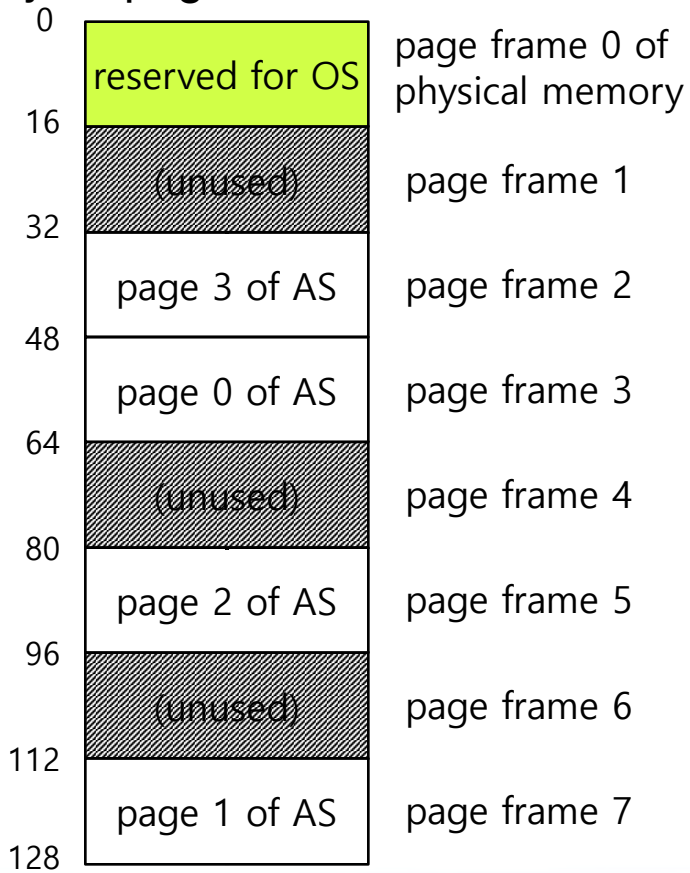
- **Simplicity**: ease of free-space management
  - The page in address space and the page frame are the same size.
  - Easy to allocate and keep a free list

Example: A Simple Paging

- **Physical**: 128-byte physical memory with 16 bytes page frames
- **Virtual**: 64-byte address space with 16 bytes pages



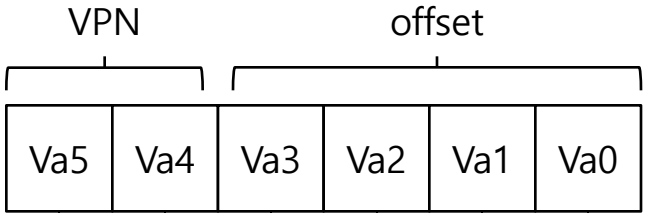
A Simple 64-byte Address Space



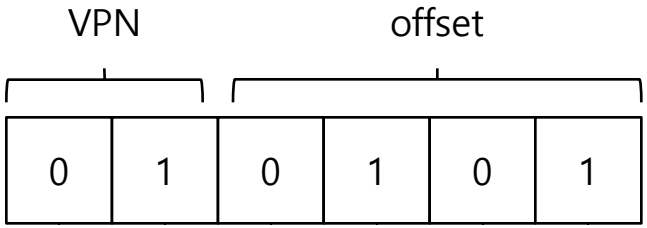
64-Byte Address Space Placed In Physical Memory

Address Translation

- Two components in the virtual address
  - VPN: Virtual Page Number
  - Offset: offset within the page

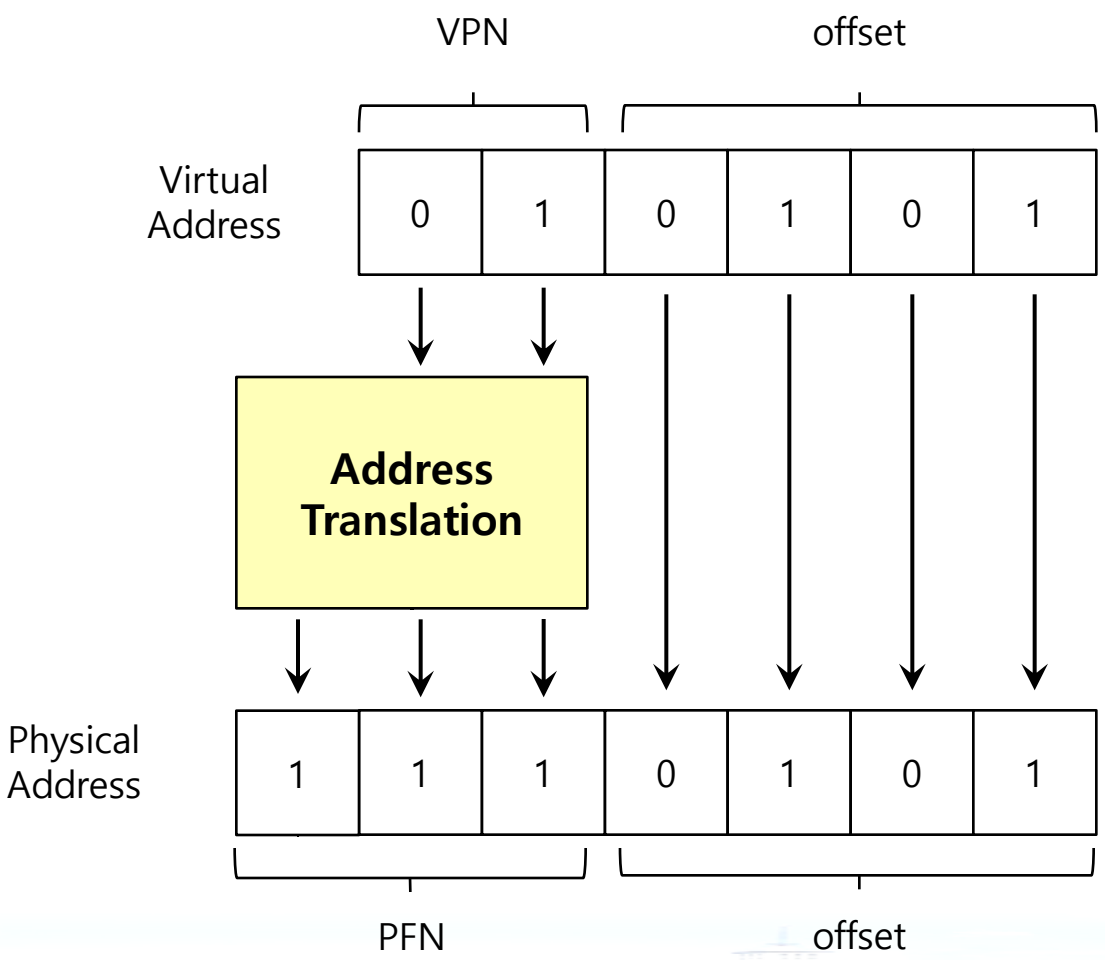


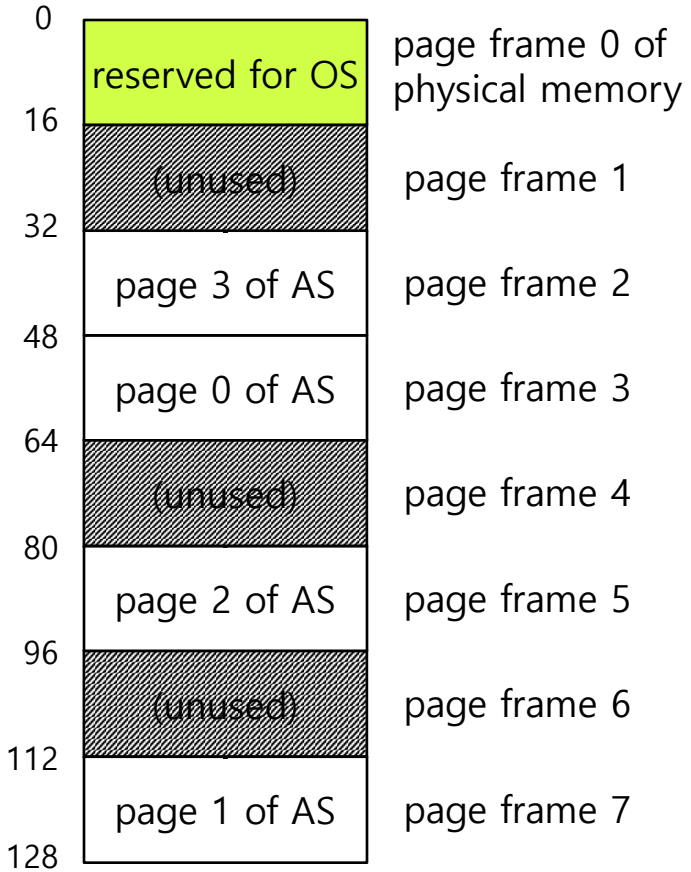
- Example: virtual address 21 in 64-byte address space



Example: Address Translation

- The virtual address 21 in 64-byte address space





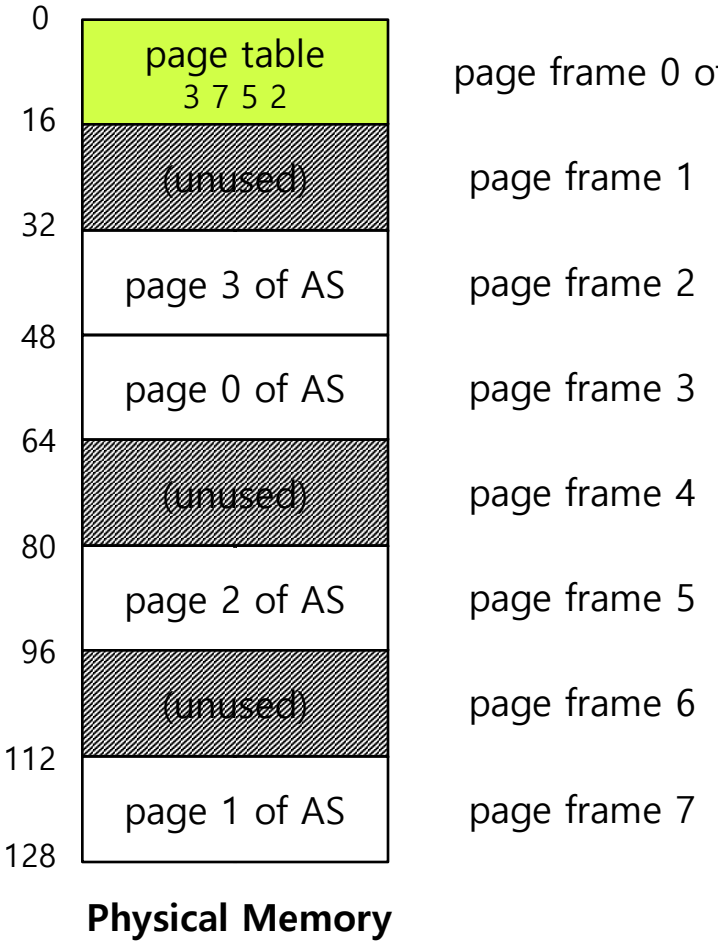
64-Byte Address Space Placed In Physical Memory

Where Are Page Tables Stored?

- Page tables can get awfully large
  - 32-bit address space with 4-KB pages, 20 bits for VPN
    - $4MB = 2^{20}\ entries * 4\ Bytes\ per\ page\ table\ entry$
- Page tables for peach process are stored in memory.

- Because page tables are so big, we don't keep any special on-chip hardware in the MMU to store the page table of the currently-running process.
- Instead, we store the page table for each process ***in memory*** somewhere

Example: Page Table in Kernel Physical Memory



- page frame 0 of physical memory 给出了一个简单的page table示例
- 注意：
  - VPN就不需要存储了
  - Index代表PFN

What Is In The Page Table?

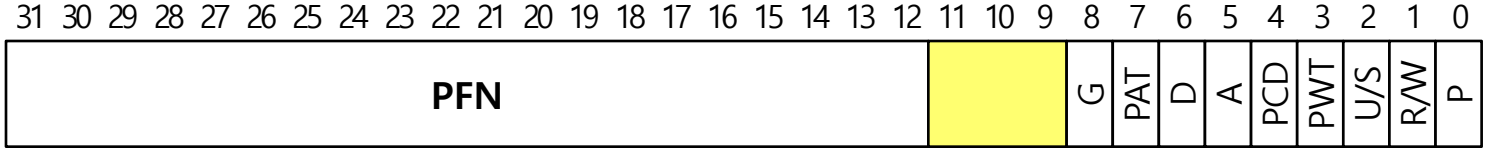
- The page table is just a **data structure** that is used to map the virtual address to physical address.
  - Simplest form: a linear page table, an array

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

Common Flags Of Page Table Entry

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

Example: x86 Page Table Entry



An x86 Page Table Entry(PTE)

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

- Read the Intel Architecture Manuals for more details on x86 paging support.
- Be forewarned, however; reading manuals such as these, while quite informative can be challenging at first. A little **patience**, and a lot of **desire**, is required.

Paging: Too Slow

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

- For every memory reference, paging requires the OS to perform one **extra memory reference, 准确的说是至少一次**

Accessing Memory With Paging


```
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)
```

A Memory Trace

- Example: A Simple Memory Access

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

- Compile and execute

```
prompt> gcc -o array array.c -Wall -o
prompt>./array
```

- Resulting Assembly code

```
0x1024 movl $0x0,(%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne 0x1024
```

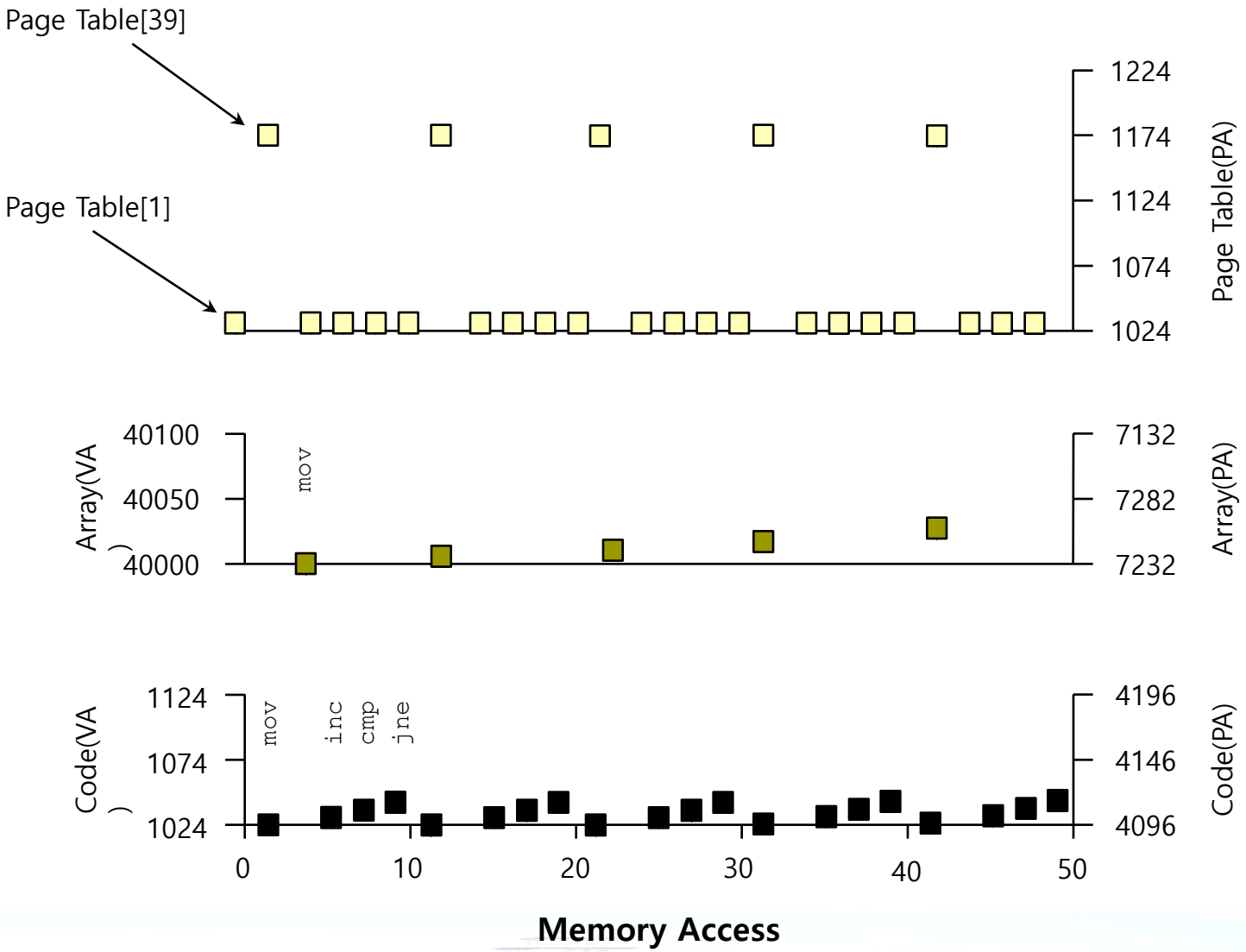
A Virtual(And Physical) Memory Trace

Some assumptions:

- virtual address space size: 64KB
- page size:1KB
- a linear (array-based) page table at physical address 1KB (1024)
- Code: virtual address 1024; VPN=1 → PFN=4
- Array: virtual addresses 40000 through 44000; VPN=39, ... , VPN=42

VPN	PFN
39	7
40	8
41	9
42	10

A Virtual(And Physical) Memory Trace



A Virtual(And Physical) Memory Trace

each instruction fetch will generate two memory references:

- access the page table to find the physical frame that the instruction resides within
- access the instruction itself to fetch it to the CPU for processing

mov instruction will generate one explicit memory reference:

- translate the array virtual address to the correct physical one

each array element will generate one memory references.

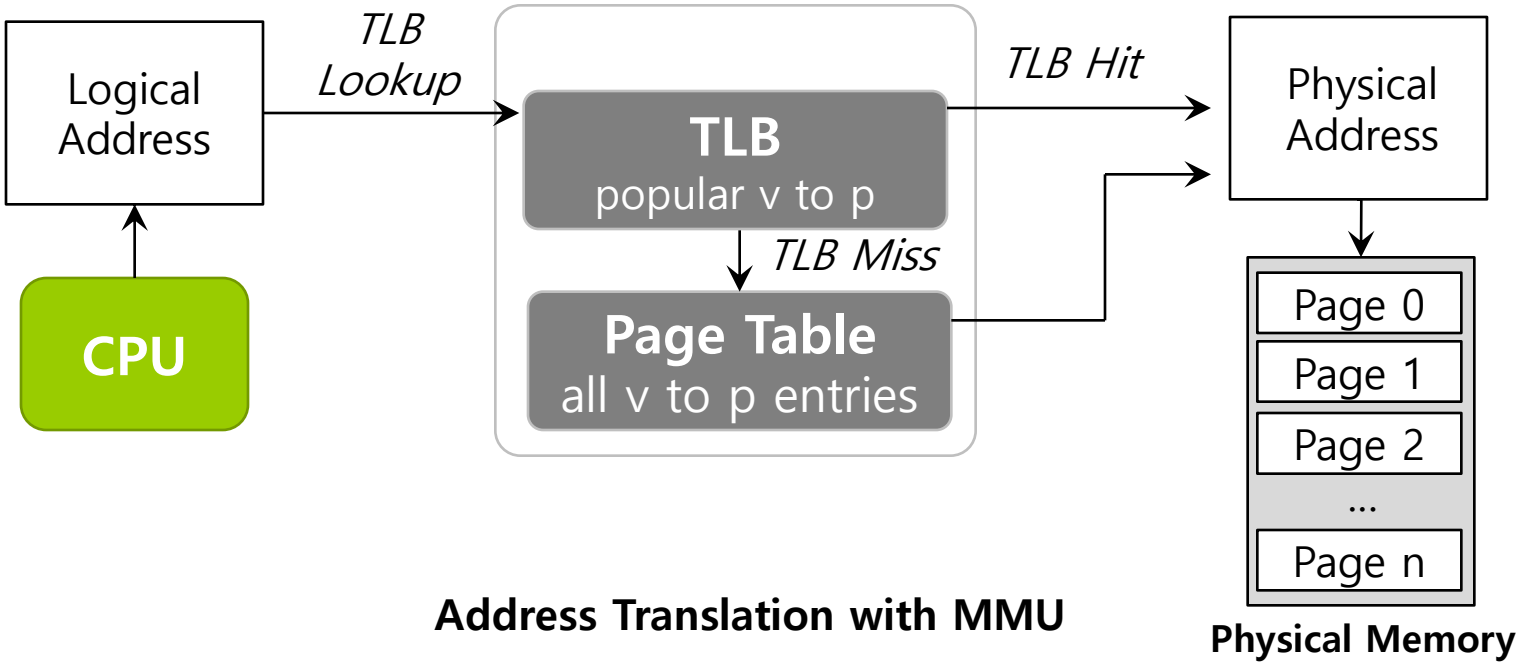
For each element of array:
Mov instruction accesses the page table (topmost graph) and the instruction itself (bottom graph), then accesses array virtual address (topmost graph). After that, the array accesses memory (middle graph). The other three instructions access memory twice for each.

Module 4: Virtualizing Memory 虚拟内存

1. 虚拟地址空间
2. 内存管理API（Memory API ）
3. **Mechanism**: 地址翻译（Address Translation）
4. 分段（ **Segmentation** ）
5. 空闲空间管理（ Free-Space Management ）
6. 分页（Paging）
7. **Paging performance: 地址翻译缓存 TLB（ Translation Lookaside Buffers）**
8. **Small Page Table: 多级页表**
9. 交换（**Swapping**）

TLB

- Part of the chip's Memory-Management Unit (MMU).
- A hardware cache of **popular** virtual-to-physical address translation.



Address Translation with MMU

TLB Basic Algorithms

```
1: VPN = (VirtualAddress & VPN_MASK ) >> SHIFT
2: (Success , TlbEntry) = TLB_Lookup(VPN)
3:    if(Success == Ture){ // TLB Hit
4:    if(CanAccess(TlbEntry.ProtectBit) == True ){
5:        offset = VirtualAddress & OFFSET_MASK
6:        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7:        AccessMemory( PhysAddr )
8:    }else RaiseException(PROTECTION_ERROR)
```

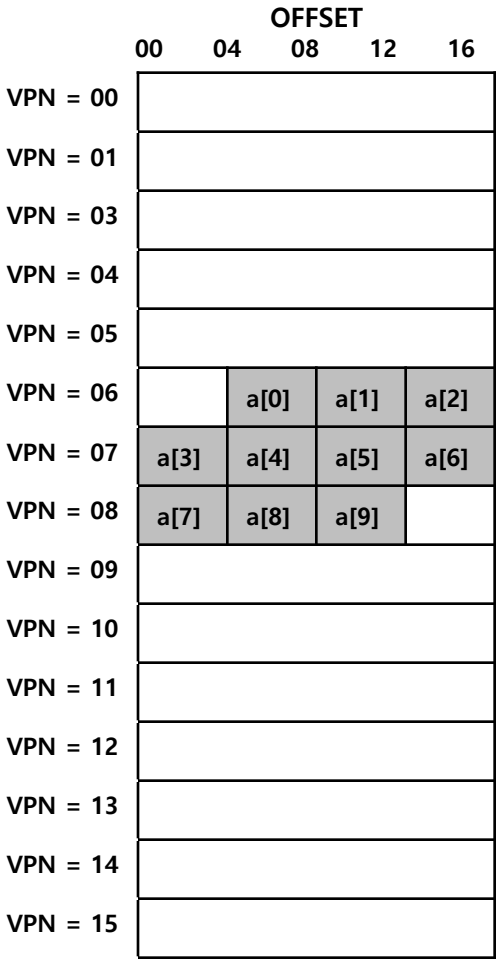
- (1 lines) extract the virtual page number(VPN).
- (2 lines) check if the TLB holds the transalation for this VPN.
- (5-8 lines) extract the page frame number from the relevant TLB entry, and form the desired physical address and access memory.

TLB Basic Algorithms (Cont.)

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

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

Example: Accessing An Array




```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

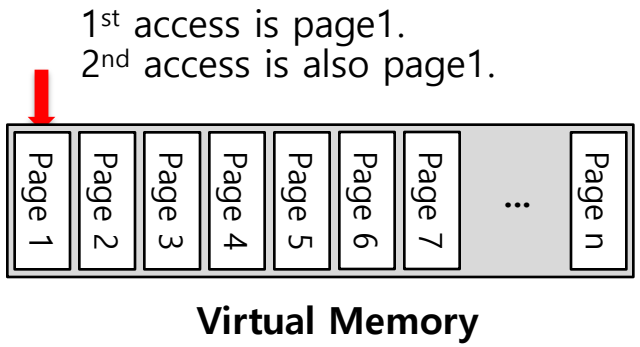
- assume an array of 10 4-byte integers in memory, starting at virtual address 100.
- a small 8-bit virtual address space, with 16-byte pages
- pretend that the only memory accesses the loop generates are to the array
-

The TLB improves performance due to spatial locality

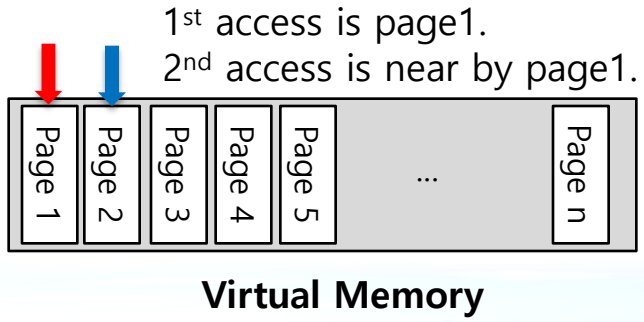
3 misses and 7 hits.
Thus TLB hit rate is 70%.

Locality

- Temporal Locality
  - An instruction or data item that has been recently accessed will likely be re-accessed soon in the future.



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



Who Handles The TLB Miss?

- Hardware handle the TLB miss entirely on CISC（往往由复杂指令集需要提供特定的CPU指令）.
  - The hardware has to know exactly where the page tables are located in memory.
  - The hardware would “walk” the page table, find the correct page-table entry and extract the desired translation, update and retry instruction.
  - **hardware-managed TLB.**

Who Handles The TLB Miss? (Cont.)

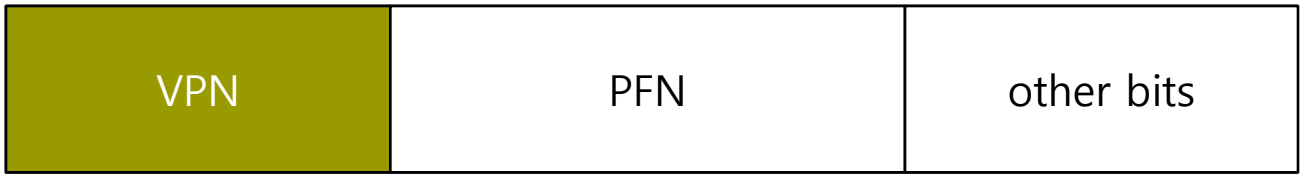
- RISC have what is known as a **software-managed TLB.**
  - On a TLB miss, the hardware raises exception(trap handler).
    - **Trap handler is code** within the OS that is written with the express purpose of handling TLB miss.

TLB Control Flow algorithm(OS Handled)

```
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)
```

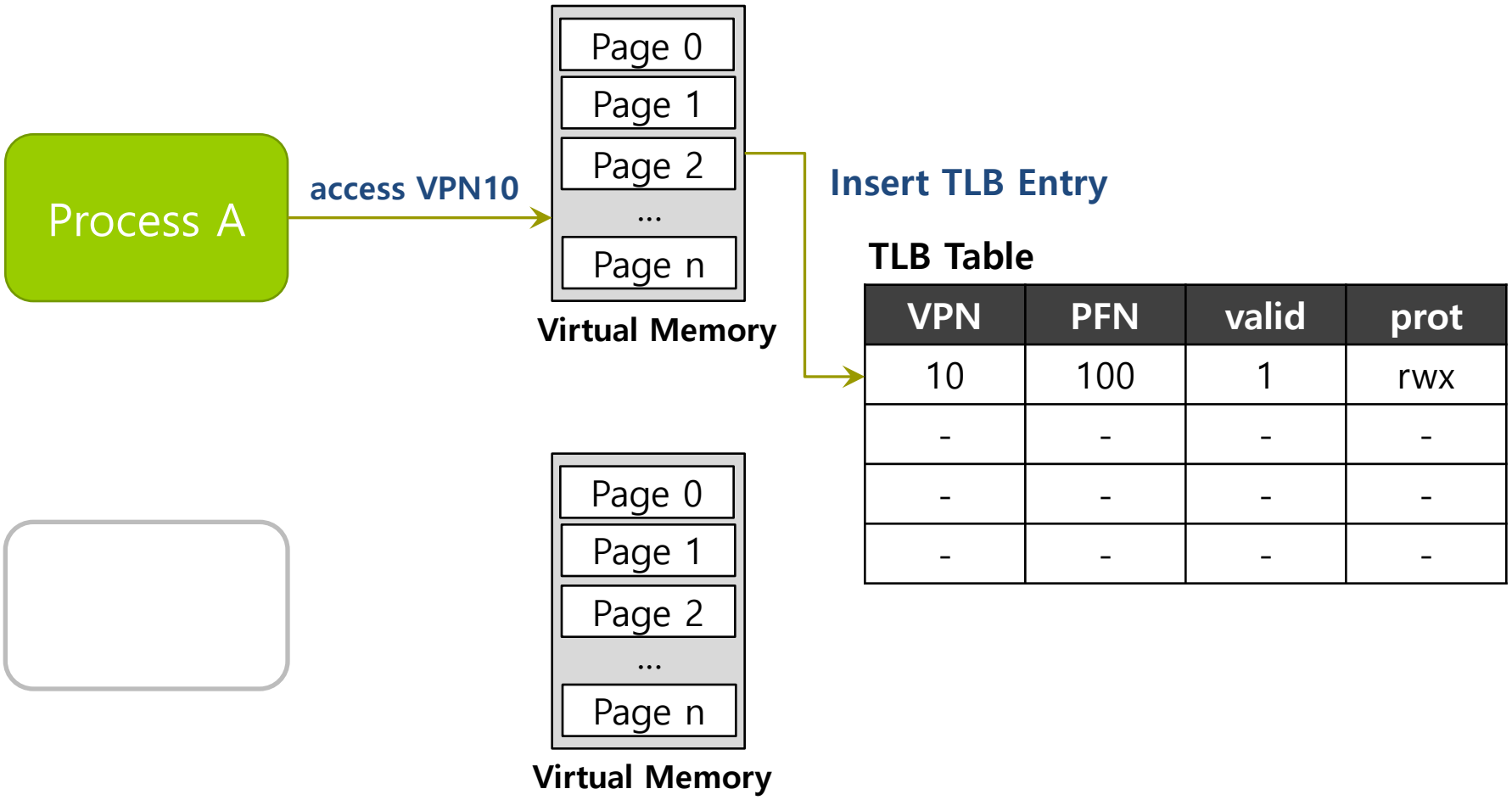
TLB entry

- TLB is managed by **Full Associative** method.
  - A typical TLB might have 32,64, or 128 entries.
  - Hardware search the entire TLB in parallel to find the desired translation.
  - other bits: valid bits , protection bits, address-space identifier, dirty bit

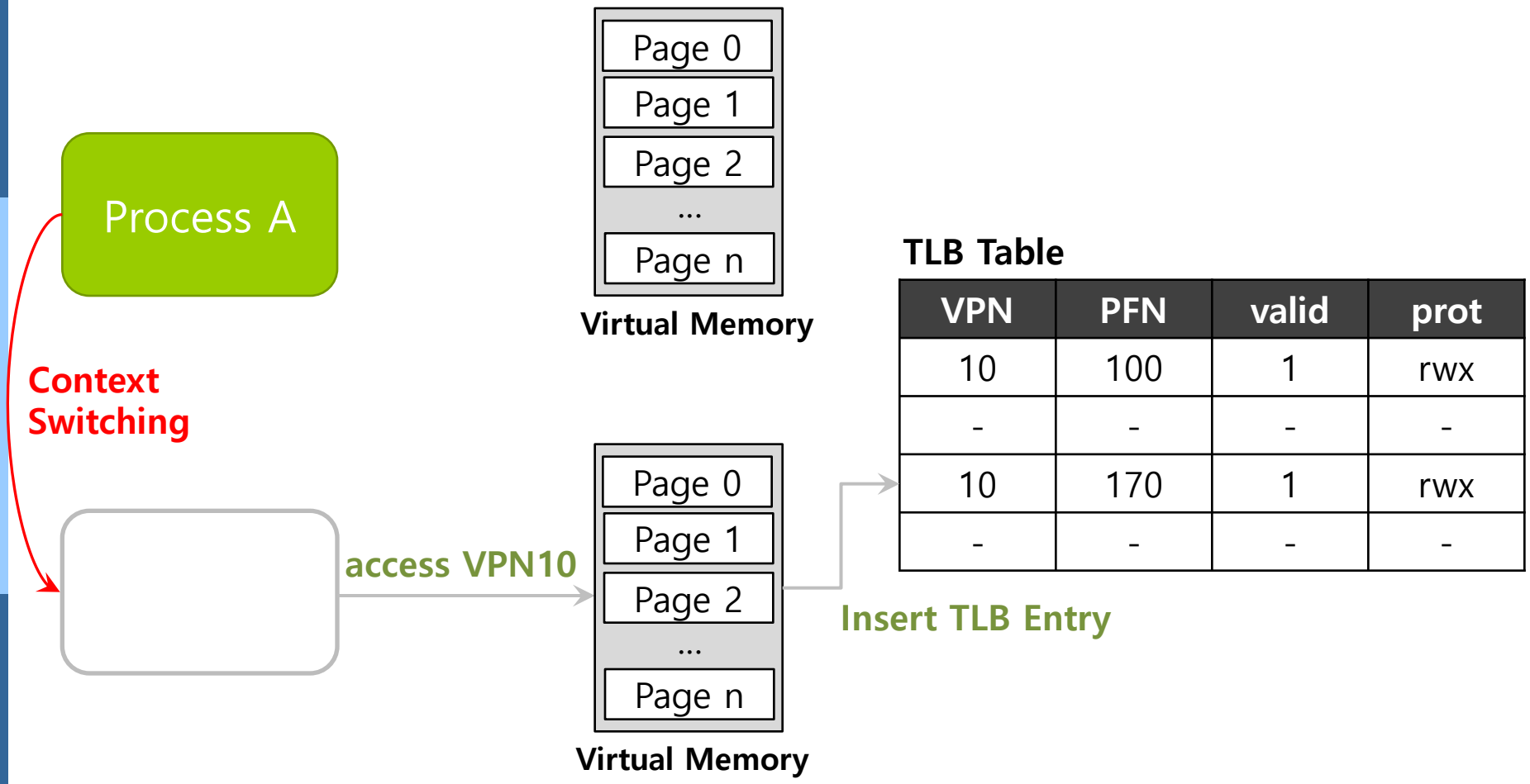


Typical TLB entry look like this

TLB Issue: Context Switching

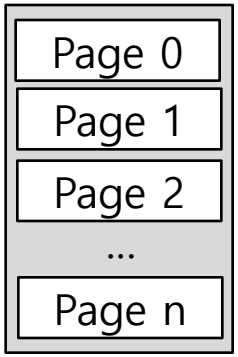


TLB Issue: Context Switching

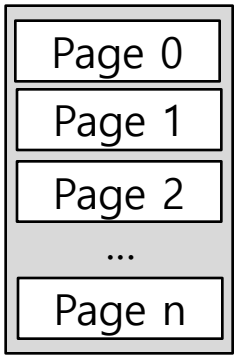


TLB Issue: Context Switching

Process A



Virtual Memory



Virtual Memory

TLB Table

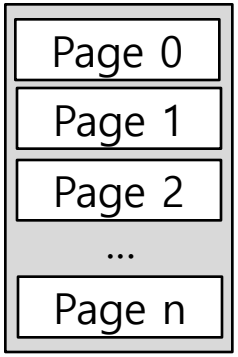
VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

Can't Distinguish which entry is meant for which process

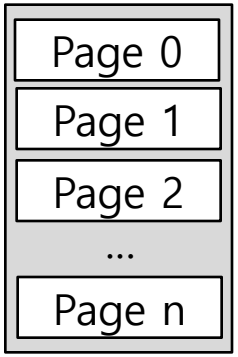
To Solve Problem

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

Process A



Virtual Memory



Virtual Memory

TLB Table

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

Another Case

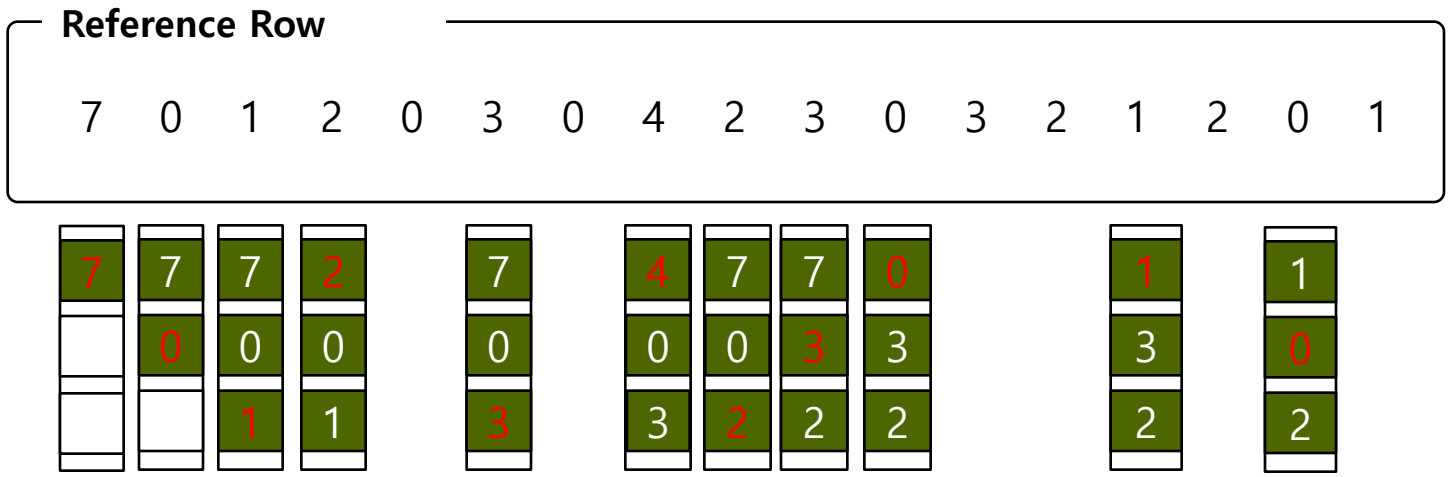
- Two processes share a page.
  - Process 1 is sharing physical page 101 with Process2.
  - P1 maps this page into the 10th page of its address space.
  - P2 maps this page to the 50th page of its address space.

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

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

TLB Replacement Policy

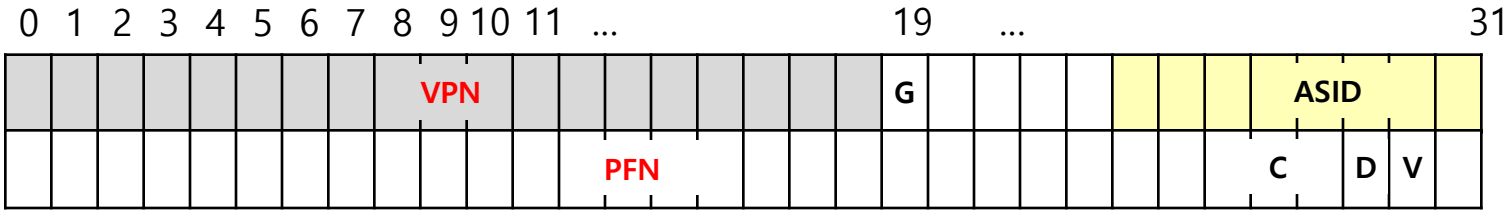
- LRU(Least Recently Used)
  - Evict an entry that has not recently been used.
  - Take advantage of *locality* in the memory-reference stream.



Total 11 TLB miss

A Real TLB Entry

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



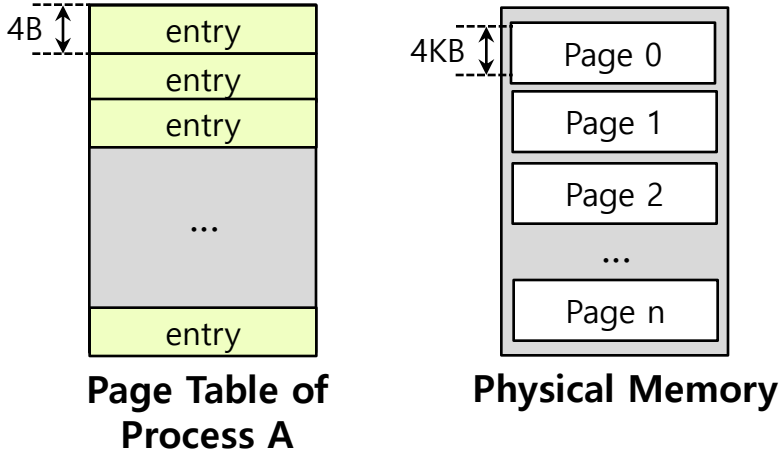
Flag	Content
19-bit VPN	The rest reserved for the kernel.
24-bit PFN	Systems can support with up to 64GB of main memory($2^{24} * 4KB$ pages).
Global bit(G)	Used for pages that are globally-shared among processes.
ASID	OS can use to distinguish between address spaces.
Coherence bit(C)	determine how a page is cached by the hardware.
Dirty bit(D)	marking when the page has been written.
Valid bit(V)	tells the hardware if there is a valid translation present in the entry.

Module 4: Virtualizing Memory 虚拟内存

1. 虚拟地址空间
2. 内存管理**API**（**Memory API** ）
3. **Mechanism**: 地址翻译（Address Translation）
4. 分段（ **Segmentation** ）
5. 空闲空间管理（ Free-Space Management ）
6. 分页（Paging）
7. Paging performance: 地址翻译缓存 TLB（ Translation Lookaside Buffers）
8. **Small Page Table:** 多级页表
9. 交换（**Swapping**）

Paging: Linear Tables

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

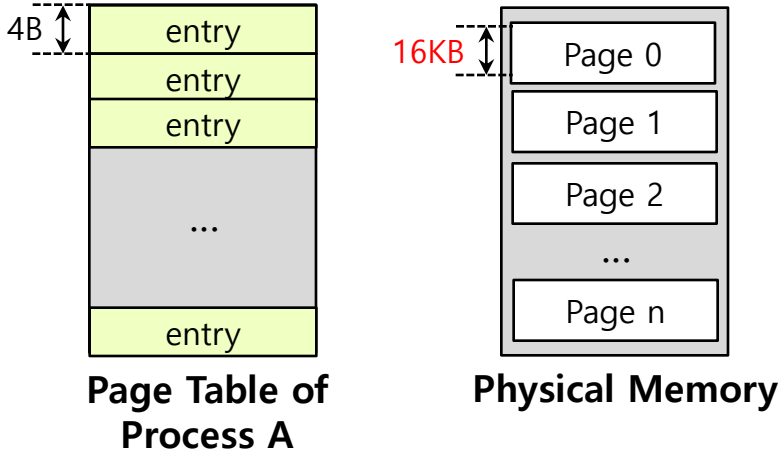


$$\text{Page table size} = \frac{2^{32}}{2^{12}} * 4Byte = 4MByte$$

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

Smaller Tables: Big pages

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

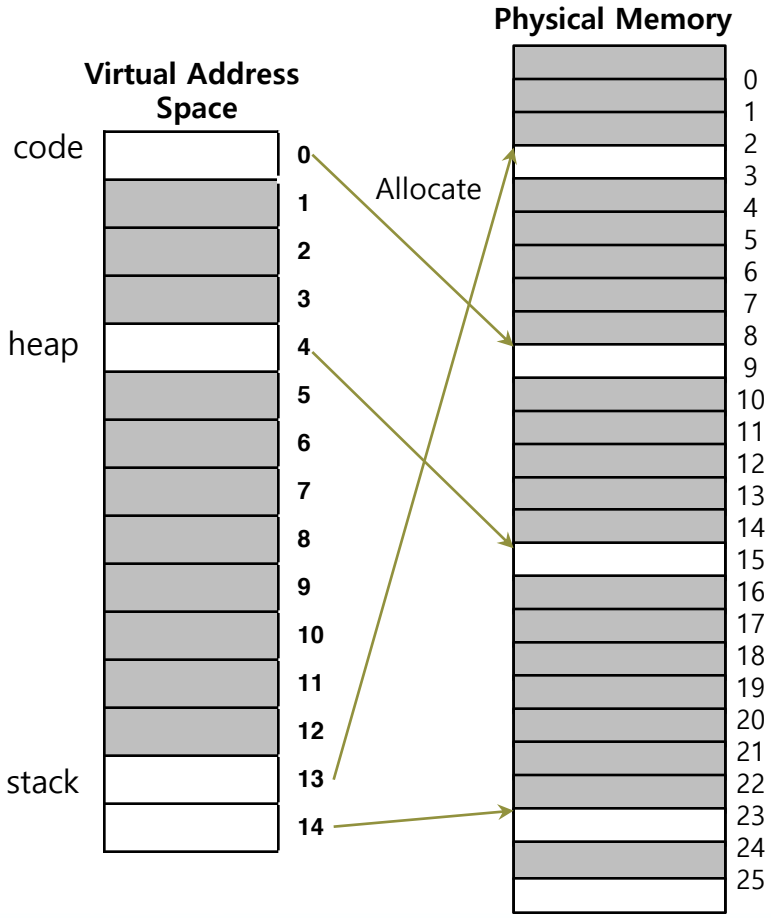


$\frac{2^{32}}{2^{16}} * 4 = 1MB$ per page table

Big pages lead to internal fragmentation.

Problem

- Single page table for the entries address space of process.



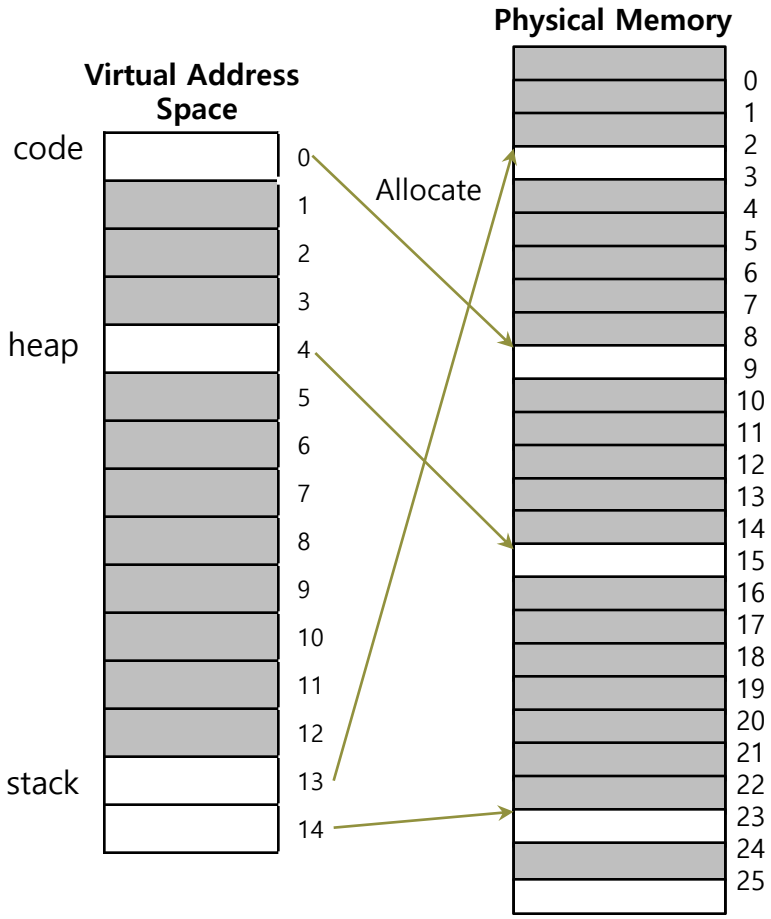
A 16KB Address Space with 1KB Pages

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

A Page Table For 16KB Address Space

Problem

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



A 16KB Address Space with 1KB Pages

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

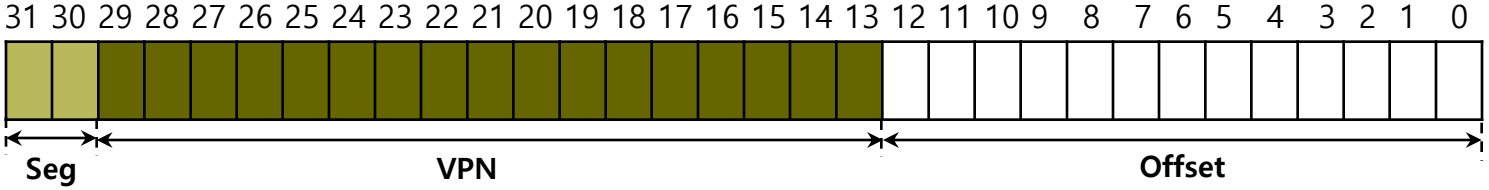
A Page Table For 16KB Address Space

Hybrid Approach: Paging and Segments

- In order to reduce the memory overhead of page tables.
  - Using base not to point to the segment itself but rather to hold the physical address of the page table of that segment.
  - The bounds register is used to indicate the end of the page table.

Simple Example of Hybrid Approach

- Each process has three page tables associated with it.
  - When process is running, the base register for each of these segments contains the physical address of a linear page table for that segment.



32-bit Virtual address space with 4KB pages

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

TLB miss on Hybrid Approach

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

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

Problem of Hybrid Approach

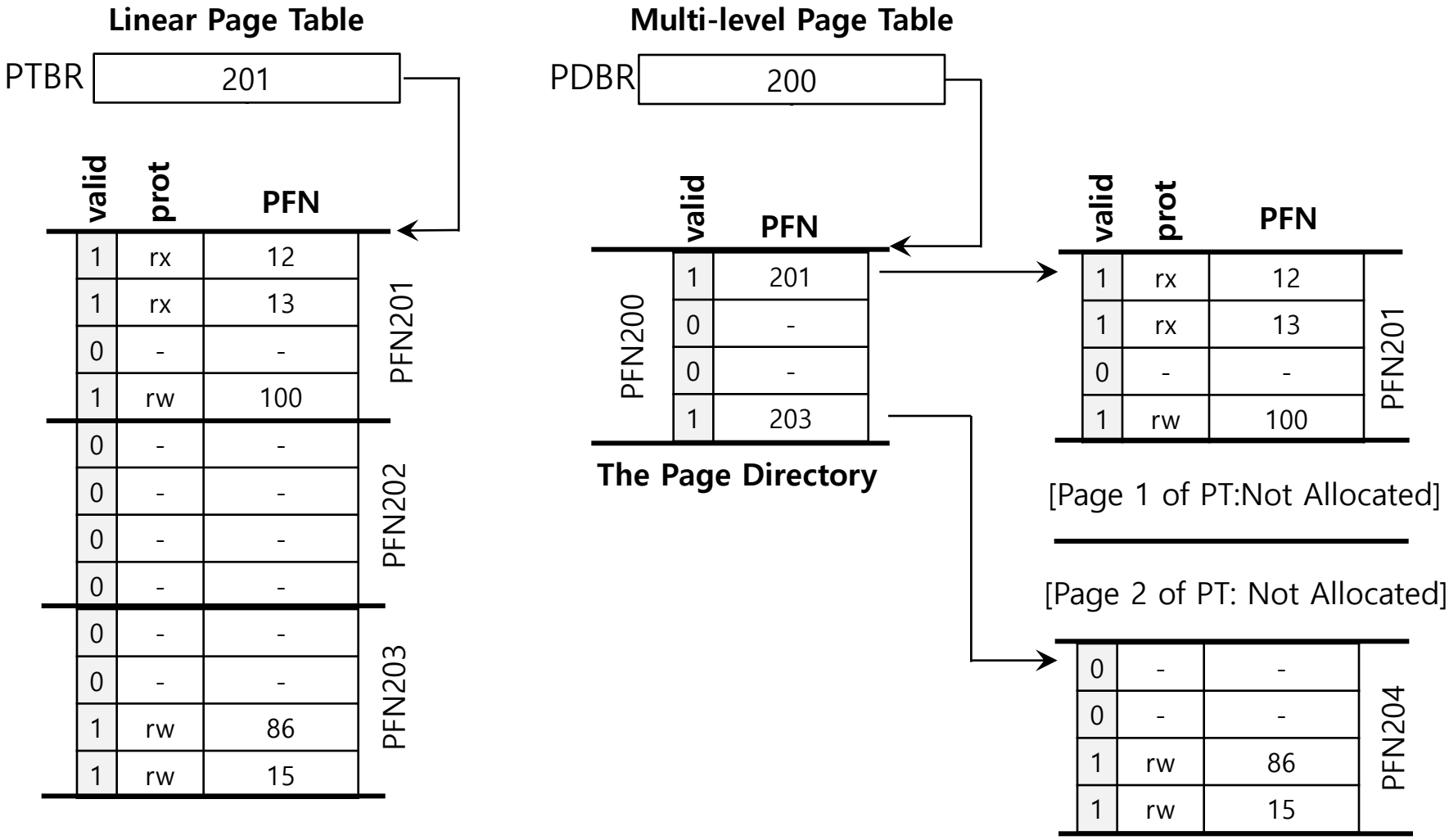
- Hybrid Approach is not without problems.
  - If we have a large but sparsely-used heap, we can still end up with a lot of page table waste.
  - Causing external fragmentation to arise again.

Big pages 和hybrid都不是好设计！怎么办？改进Page Table的数据结构，引入多级页表（Multi-level Page Tables）

Multi-level Page Tables

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

Example: Page directory



Linear (Left) And Multi-Level (Right) Page Tables

Example: analysis

- Linear Page Table
  - even though most of the middle regions of the address space are not valid, we still require page-table space allocated for those regions (i.e., the middle two pages of the page table).
- Multi-level Page Table
  - The page directory marks just two pages of the page table as valid (the first and last); thus, just those two pages of the page table reside in memory.
  - one way to visualize what a multi-level table is doing: it just makes parts of the linear page table disappear (freeing those frames for other uses), and **tracks which pages of the page table are allocated with the page directory**.

Multi-level Page Tables: Page directory entries

- The page directory contains one entry per page of the page table.
  - It consists of a number of page directory entries(PDE).
- PDE has a valid bit and page frame number(PFN).

Multi-level Page Tables: Pros. & Cons.

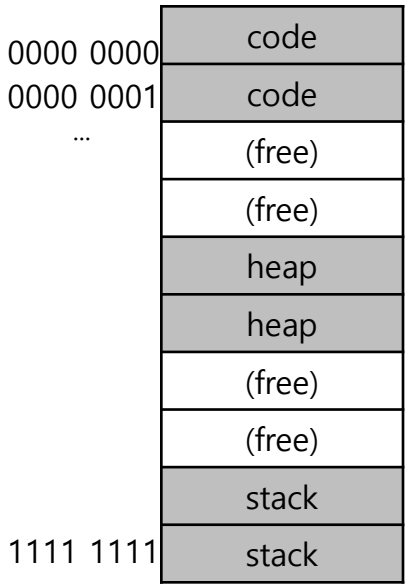
- Advantage
  - Only allocates page-table space in proportion to the amount of address space you are using.
  - The OS can grab the next free page when it needs to allocate or grow a page table.
- Disadvantage
  - Multi-level table is a small example of a time-space trade-off.
  - Complexity.

Multi-level Page Table: Level of indirection

- A multi-level structure can adjust level of indirection through use of the page directory.
  - Indirection place page-table pages wherever we would like in physical memory.

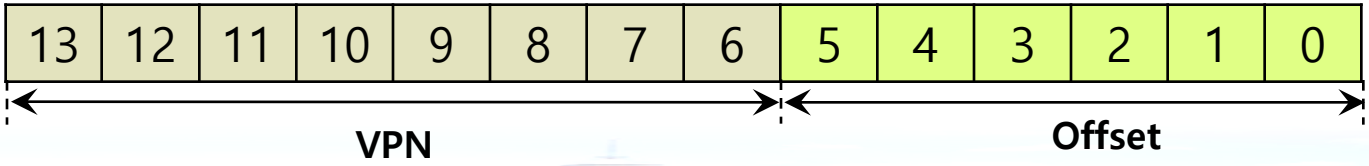
A Detailed Multi-Level Example

- To understand the idea behind multi-level page tables better, let's do an example.



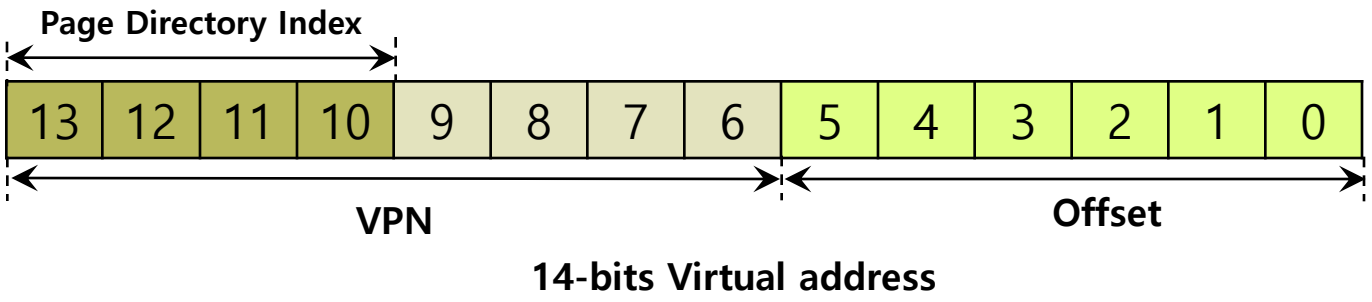
Flag	Detail
Address space	16 KB
Page size	64 byte
Virtual address	14 bit
VPN	8 bit
Offset	6 bit
Page table entry	2^8(256)

A 16-KB Address Space With 64-byte Pages



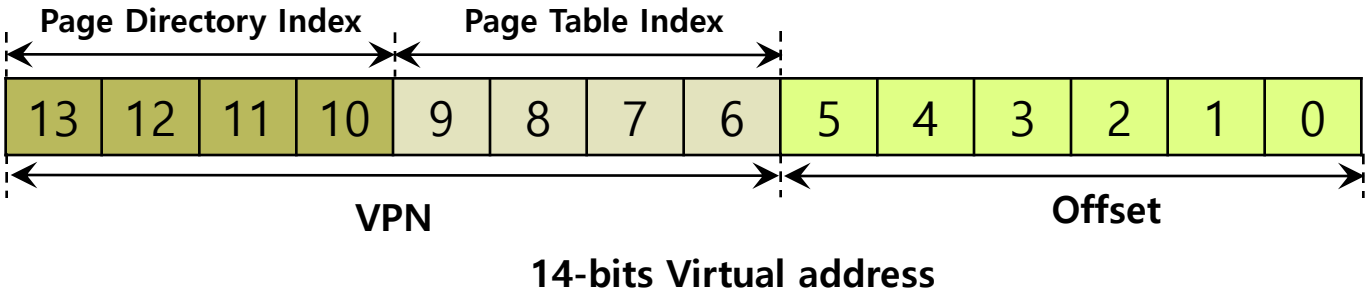
Page Directory Idx

- The page directory needs one entry per page of the page table
  - it has 16 entries.
- The page-directory entry is invalid → Raise an exception (The access is invalid)



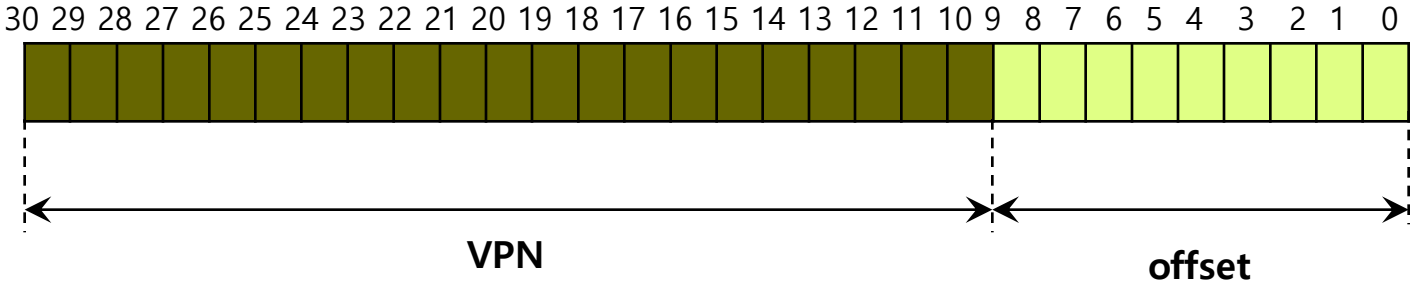
Page Table Idx (cont.)

- The PDE is valid, we have more work to do.
  - To fetch the page table entry(PTE) from the page of the page table pointed to by this page-directory entry.
- This page-table index can then be used to index into the page table itself.



More than Two Level

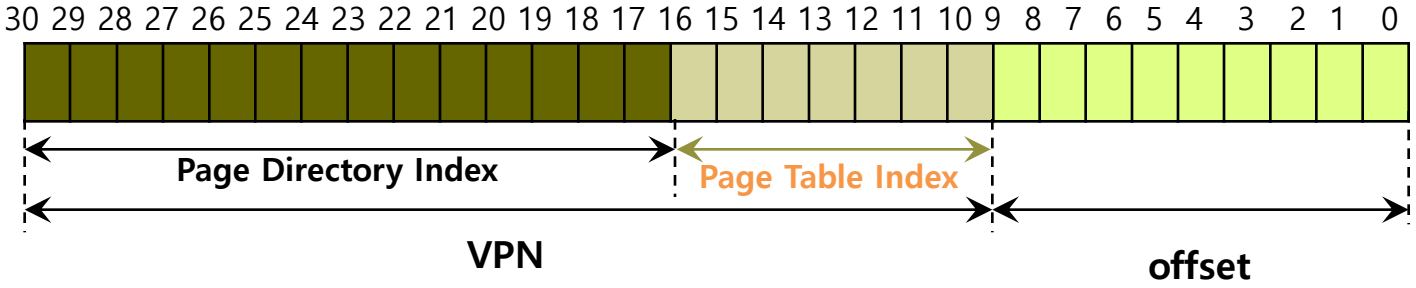
- In some cases, a deeper tree is possible.



Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit

More than Two Level : Page Table Index

- In some cases, a deeper tree is possible.

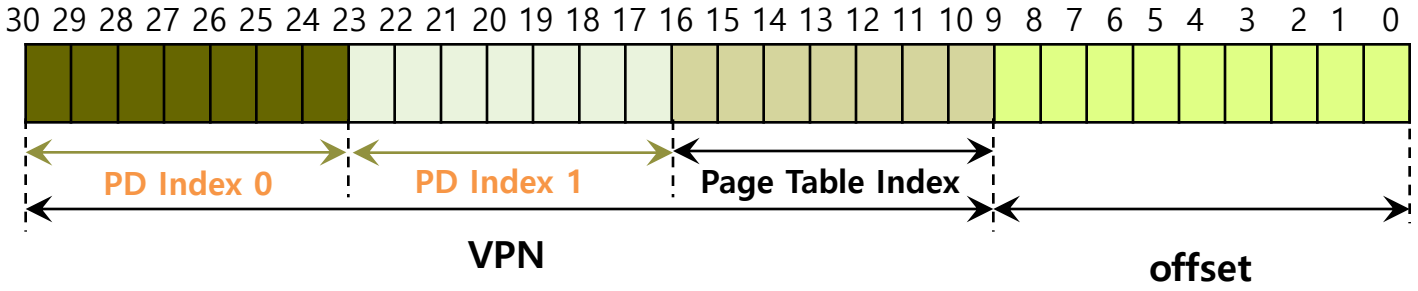


Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit
Page entry per page	128 PTEs

$\log_2 128 = 7$

More than Two Level : Page Directory

- If our page directory has 2^{14} entries, it spans not one page but 128.
- To remedy this problem, we build a further level of the tree, by splitting the page directory itself into multiple pages of the page directory.



Multi-level Page Table Control Flow

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

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

Multi-level Page Table Control Flow

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

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

The Translation Process: Remember the TLB

```
18:     PTIndex = (VPN & PT_MASK) >> PT_SHIFT
19:     PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
20:     PTE = AccessMemory(PTEAddr)
21:     if(PTE.Valid == False)
22:             RaiseException(SEGMENTATION_FAULT)
23:     else if(CanAccess(PTE.ProtectBits) == False)
24:             RaiseException(PROTECTION_FAULT);
25:     else
26:             TLB_Insert(VPN, PTE.PFN , PTE.ProtectBits)
27:             RetryInstruction()
```

Inverted Page Tables

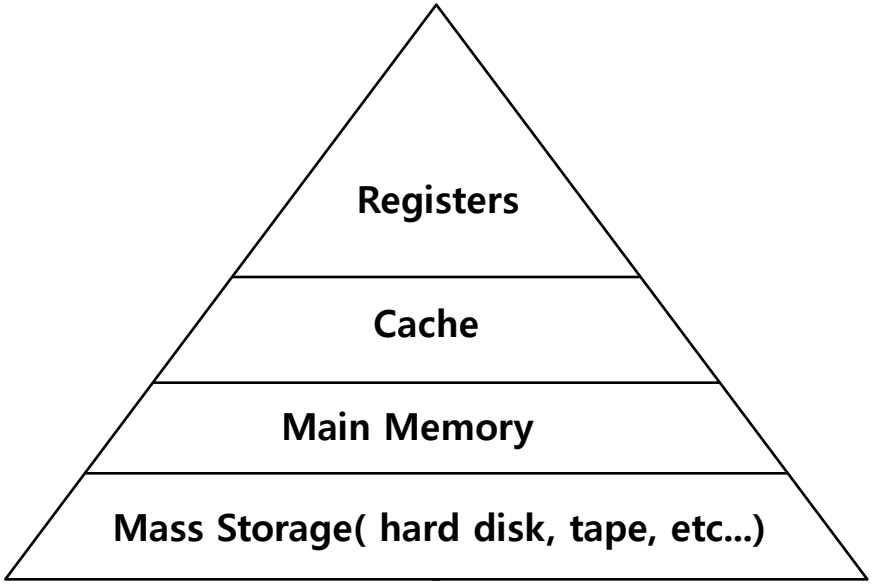
- A single page table that has an entry for each physical page of the system.
- The entry tells:
  - which process is using this page
  - Which virtual page of that process maps to this physical page
- Finding the correct entry is now a matter of searching through this data structure
  - hash table is often built over the base structure to speed up lookups
  - PowerPC 是典型例子

Module 4: Virtualizing Memory 虚拟内存

1. 虚拟地址空间
2. 内存管理**API**（**Memory API** ）
3. **Mechanism**: 地址翻译（Address Translation）
4. 分段（ **Segmentation** ）
5. 空闲空间管理（ Free-Space Management ）
6. 分页（Paging）
7. Paging performance: 地址翻译缓存 TLB（ Translation Lookaside Buffers）
8. Small Page Table: 多级页表
9. 交换（**Swapping**）

Beyond Physical Memory: Mechanisms

- Require an additional level in the memory hierarchy.
  - OS need a place to stash away portions of address space that currently aren't in great demand.
  - In modern systems, this role is usually served by a hard disk drive



Memory Hierarchy in modern system

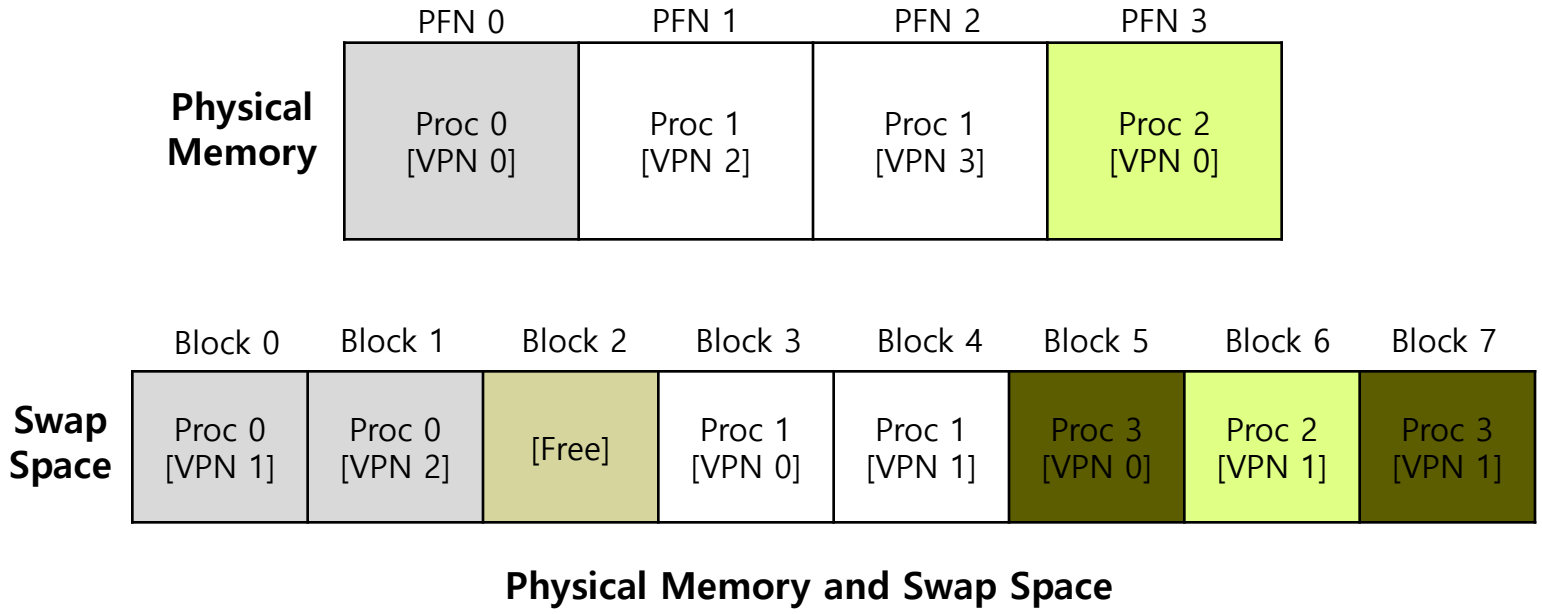
Single large address for a process

- Always need to first arrange for the code or data to be in memory when before calling a function or accessing data.

- To Beyond just a single process.
  - The addition of swap space allows the OS to support the illusion of a large virtual memory for multiple concurrently-running process

Swap Space

- Reserve some space on the disk for moving pages back and forth.
- OS need to remember to the swap space, in page-sized unit



Physical Memory and Swap Space

Present Bit

- Add some machinery higher up in the system in order to support swapping pages to and from the disk.
  - When the hardware looks in the PTE, it may find that the page is not present in physical memory.

Value	Meaning
1	page is present in physical memory
0	The page is not in memory but rather on disk.

What If Memory Is Full ?

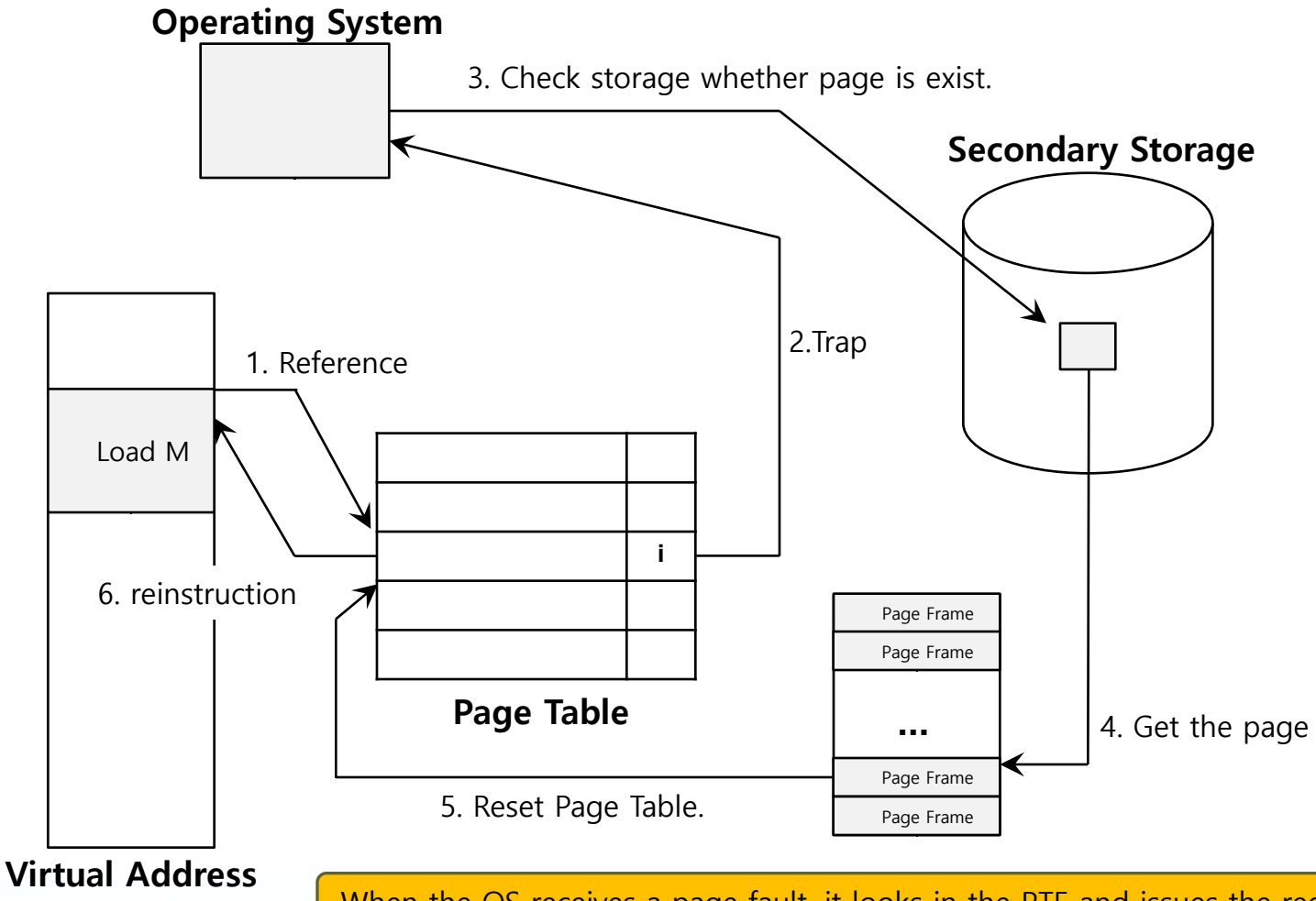
- The OS like to page out pages to make room for the new pages the OS is about to bring in.
  - The process of picking a page to kick out, or replace is known as page-replacement policy

The Page Fault

- Accessing page that is not in physical memory.
  - If a page is not present and has been swapped disk, the OS need to swap the page into memory in order to service the page fault.

Page Fault Control Flow

- PTE used for data such as the PFN of the page for a disk address.



When the OS receives a page fault, it looks in the PTE and issues the request to disk.

Page Fault Control Flow – Hardware

```
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 RaiseException(PROTECTION_FAULT)
```

Page Fault Control Flow – Hardware

```
9:      else // TLB Miss
10:             PTEAddr = PTBR + (VPN * sizeof(PTE))
11:             PTE = AccessMemory(PTEAddr)
12:             if (PTE.Valid == False)
13:                     RaiseException(SEGMENTATION_FAULT)
14:             else
15:     if (CanAccess(PTE.ProtectBits) == False)
16:             RaiseException(PROTECTION_FAULT)
17:     else if (PTE.Present == True)
18:     // assuming hardware-managed TLB
19:             TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
20:             RetryInstruction()
21:     else if (PTE.Present == False)
22:             RaiseException(PAGE_FAULT)
```

Page Fault Control Flow – Software

```
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
```

- The OS must find a physical frame for the soon-be-faulted-in page to reside within.
- If there is no such page, waiting for the replacement algorithm to run and kick some pages out of memory.

When Replacements Really Occur

- OS waits until memory is entirely full, and only then replaces a page to make room for some other page
  - This is a little bit unrealistic, and there are many reason for the OS to keep a small portion of memory free more proactively.

- 后台任务: Swap Daemon, Page Daemon
  - There are fewer than LW pages available, a background thread that is responsible for freeing memory runs.
  - The thread evicts pages until there are HW pages available.

Beyond Physical Memory: Policies

- Memory pressure forces the OS to start paging out pages to make room for actively-used pages.
- Deciding which page to evict is encapsulated within the replacement policy of the OS.

Cache Management

- Goal in picking a replacement policy for this cache is to minimize the number of cache misses.
- The number of cache hits and misses let us calculate the *average memory access time(AMAT)*.

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

Arguement	Meaning
T_M	The cost of accessing memory
T_D	The cost of accessing disk
P_{Hit}	The probability of finding the data item in the cache(a hit)
P_{Miss}	The probability of not finding the data in the cache(a miss)

The Optimal Replacement Policy

- Leads to the fewest number of misses overall
  - Replaces the page that will be accessed furthest in the future
  - Resulting in the fewest-possible cache misses
- Serve only as a comparison point, to know how close we are to perfect

Tracing the Optimal Policy

Reference Row

0 1 2 0 1 3 0 3 1 2 1

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

Hit rate is $\frac{Hits}{Hits+Misses} = \mathbf{54.6\%}$

Future is not known.

A Simple Policy: FIFO

- Pages were placed in a queue when they enter the system.
- When a replacement occurs, the page on the tail of the queue(the "**First-in**" pages) is evicted.
  - It is simple to implement, but can't determine the importance of blocks.

Tracing the FIFIO Policy

Reference Row

0 1 2 0 1 3 0 3 1 2 1

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		3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

Hit rate is $\frac{Hits}{Hits+Misses} = 36.4\%$

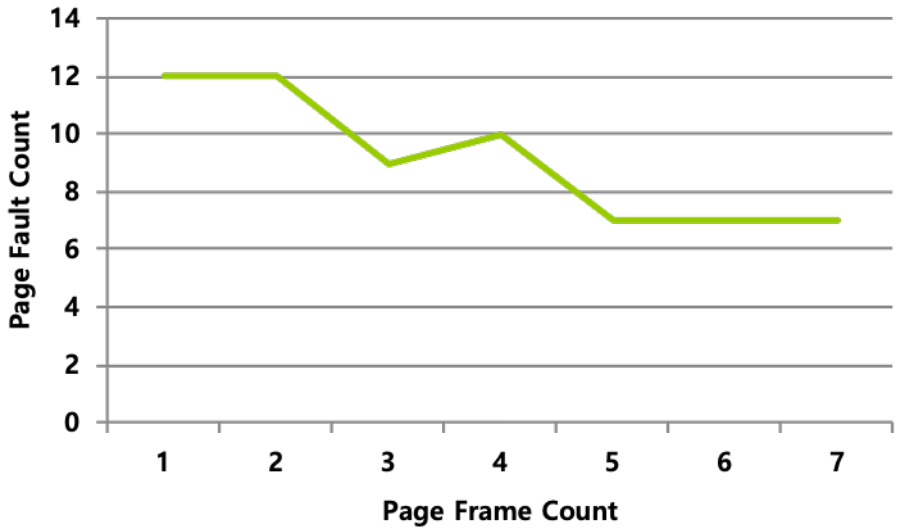
Even though page 0 had been accessed a number of times, FIFO still kicks it out.

BELADY'S ANOMALY

- We would expect the cache hit rate to increase when the cache gets larger. But in this case, with FIFO, it gets worse.

Reference Row

1 2 3 4 1 2 5 1 2 3 4 5



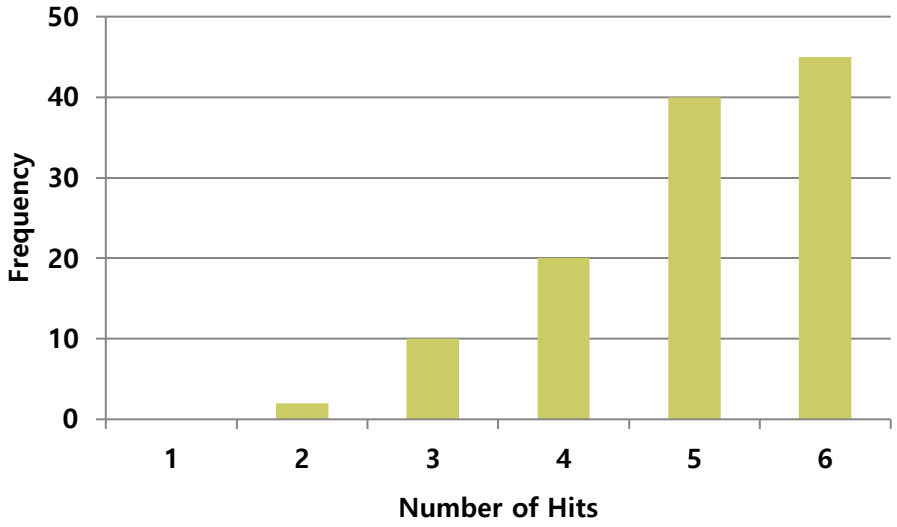
Another Simple Policy: Random

- Picks a random page to replace under memory pressure.
  - It doesn't really try to be too intelligent in picking which blocks to evict.
  - Random does depends entirely upon how lucky Random gets in its choice

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

Random Performance

- Sometimes, Random is as good as optimal, achieving 6 hits on the example trace.



Random Performance over 10,000 Trials

Using History

- Lean on the past and use **history**.
  - Two type of historical information.

Historical Information	Meaning	Algorithms
recency	The more recently a page has been accessed, the more likely it will be accessed again	LRU
frequency	If a page has been accessed many times, It should not be replcaed as it clearly has some value	LFU

Using History : LRU

- Replaces the least-recently-used page.

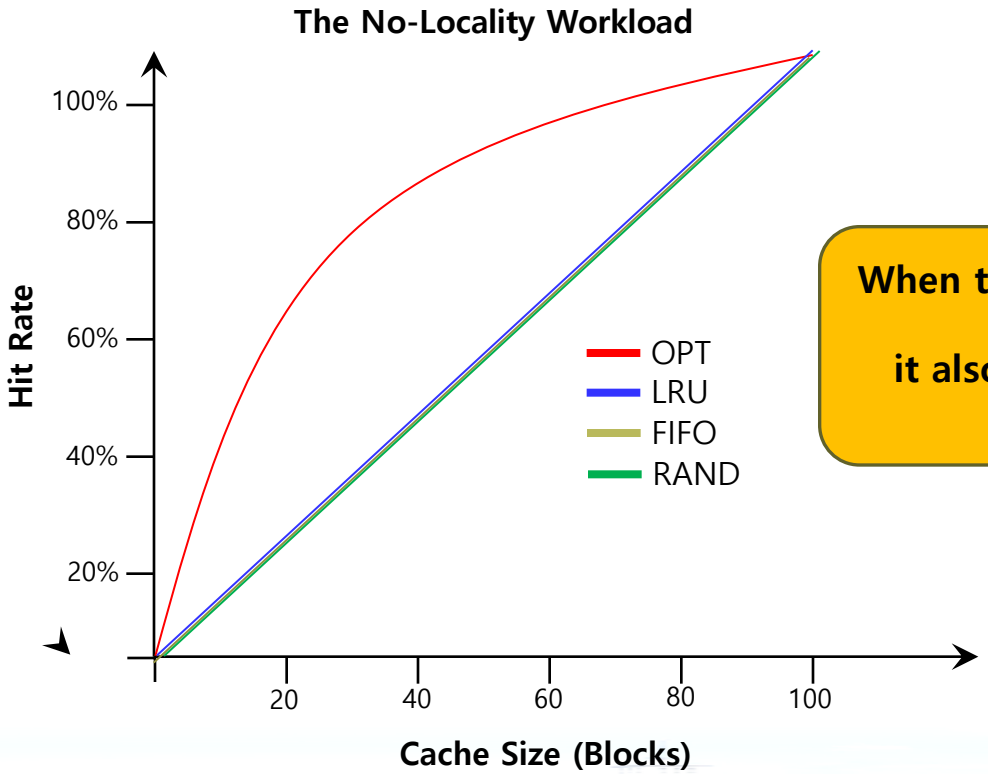
Reference Row

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting 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

Workload Example : The No-Locality Workload

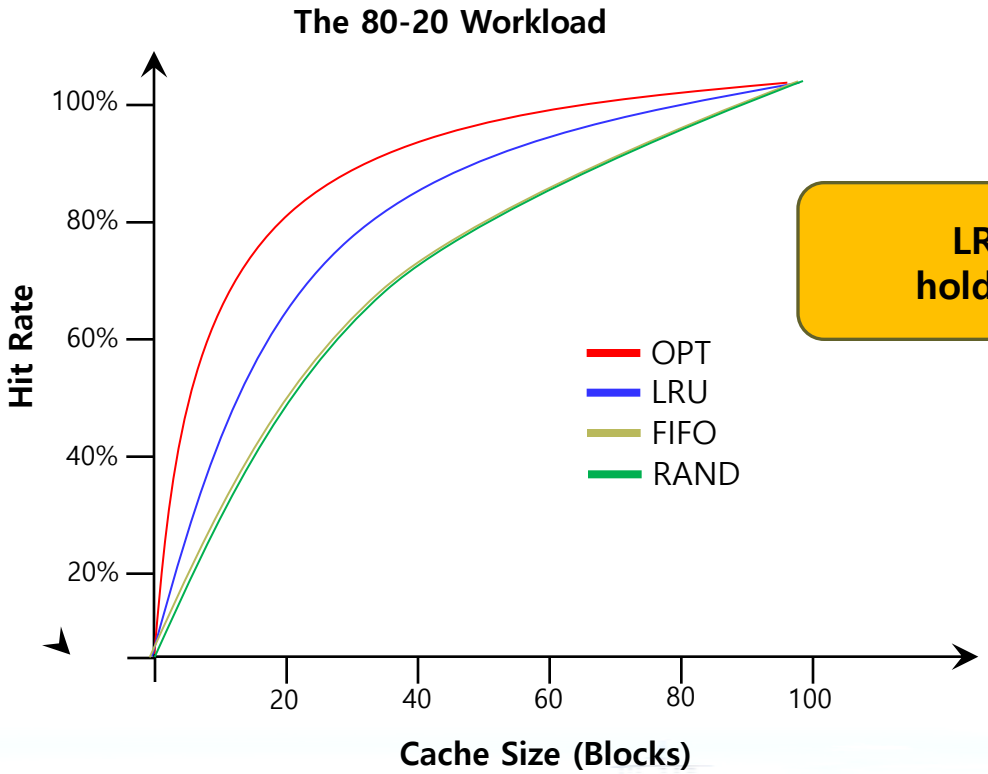
- Each reference is to a random page within the set of accessed pages.
  - Workload accesses 100 unique pages over time.
  - Choosing the next page to refer to at random



When the cache is large enough to fit the entire workload, it also doesn't matter which policy you use.

Workload Example : The 80-20 Workload

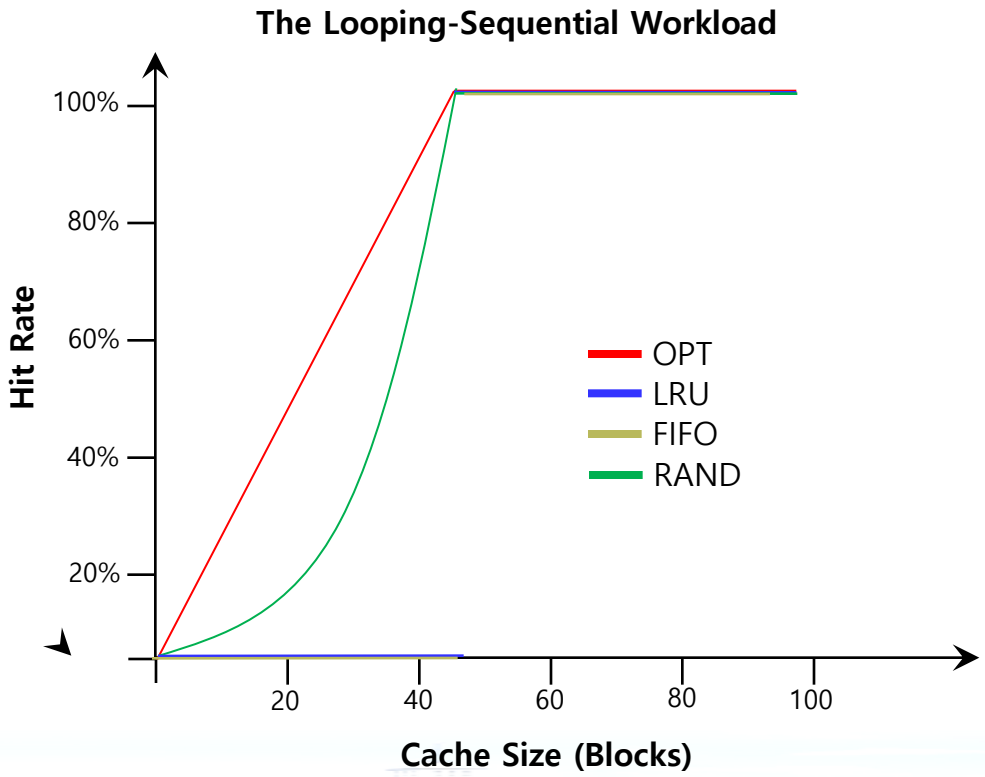
- Exhibits locality: 80% of the reference are made to 20% of the page
- The remaining 20% of the reference are made to the remaining 80% of the pages.



LRU is more likely to hold onto the hot pages.

Workload Example : The Looping Sequential

- Refer to 50 pages in sequence.
  - Starting at 0, then 1, … up to page 49, and then we Loop, repeating those accesses, for total of 10,000 accesses to 50 unique pages.



Implementing Historical Algorithms

- To keep track of which pages have been least-and-recently used, the system has to do some accounting work on **every memory reference.**
  - Add a little bit of hardware support.

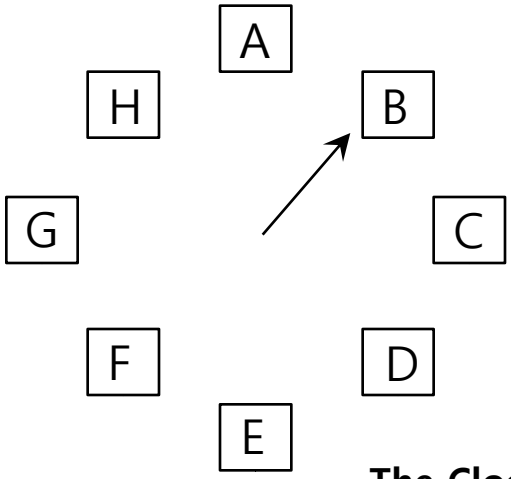
Approximating LRU

- Require some hardware support, in the form of a **use bit**
  - Whenever a page is referenced, the use bit is set by hardware to 1.
  - Hardware never clears the bit, though; that is the responsibility of the OS

- Clock Algorithm
  - All pages of the system arranges in a circular list.
  - A clock hand points to some particular page to begin with.

Clock Algorithm

- The algorithm continues until it finds a use bit that is set to 0.



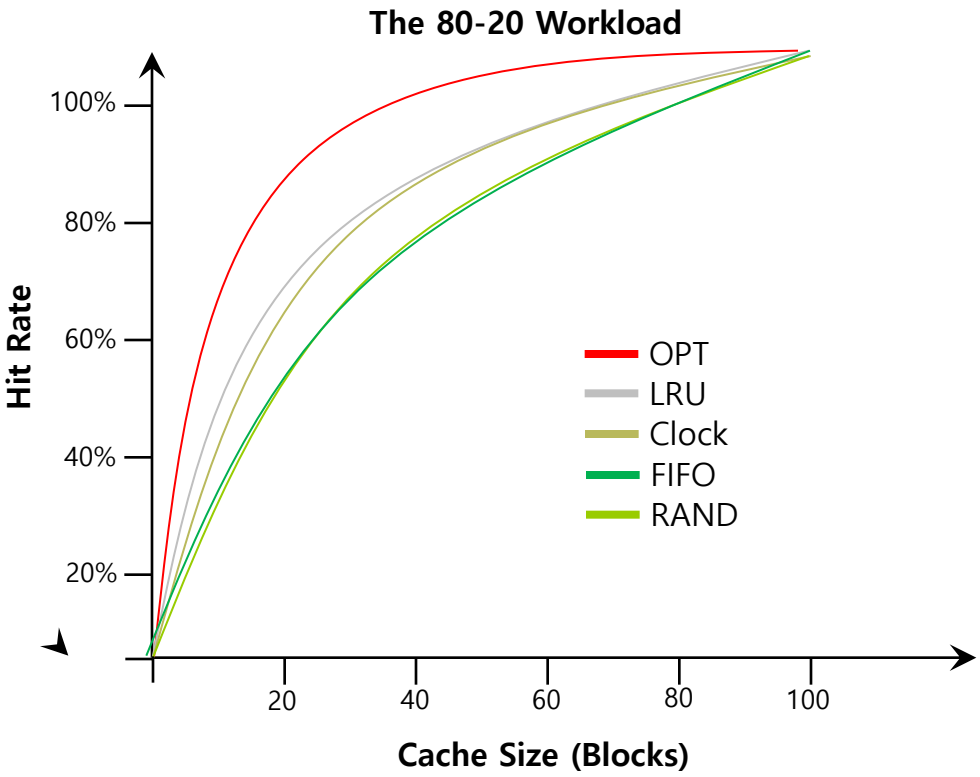
Use bit	Meaning
0	Evict the page
1	Clear **Use bit** and advance hand

The Clock page replacement algorithm

When a page fault occurs, the page the hand is pointing to is inspected.
The action taken depends on the Use bit

Workload with Clock Algorithm

- Clock algorithm doesn't do as well as perfect LRU, it does better then approach that don't consider history at all.



Considering Dirty Pages

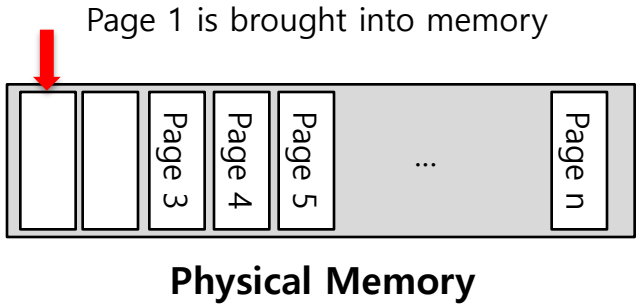
- The hardware include a **modified bit** (a.k.a **dirty bit**)
  - Page has been **modified** and is thus **dirty**, it must be written back to disk to evict it.
  - Page has not been modified, the eviction is free.

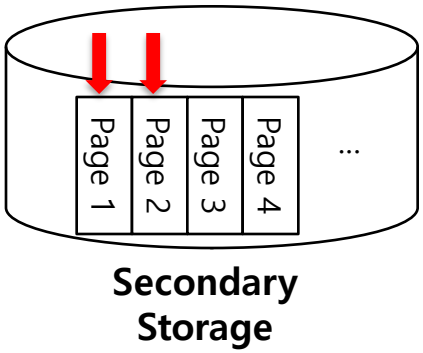
Page Selection Policy

- The OS has to decide when to bring a page into memory.
- Presents the OS with some different options.

Prefetching

- The OS guess that a page is about to be used, and thus bring it in ahead of time.

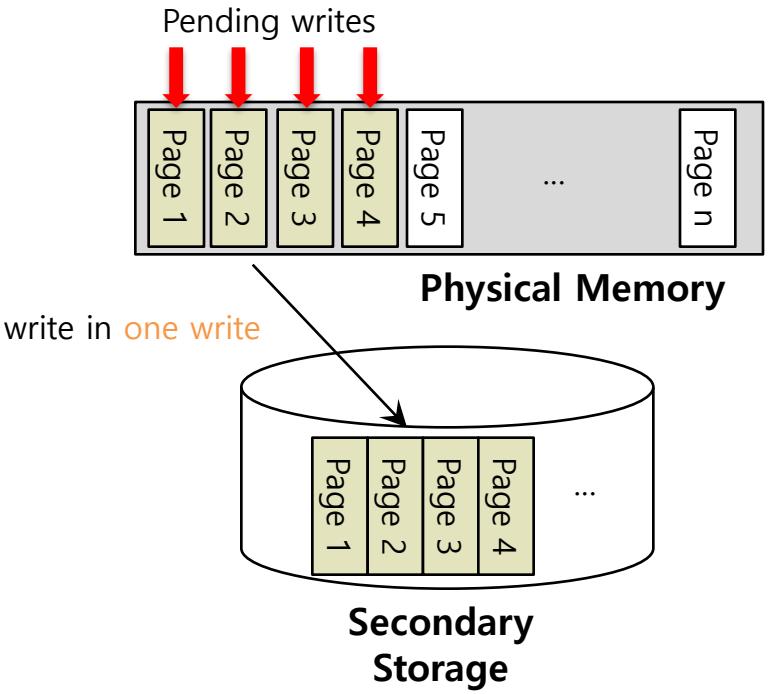




Page 2 likely soon be accessed and
thus should be brought into memory too

Clustering, Grouping

- Collect a number of pending writes together in memory and write them to disk in one write.
  - Perform a **single large write** more efficiently than **many small ones**.



Thrashing

- Memory is oversubscribed and the memory demands of the set of running processes exceeds the available physical memory.
  - Decide not to run a subset of processes.
  - Reduced set of processes working sets fit in memory.