Advanced Operating Systems: Three Easy Pieces

1. Virtualization:
1.2 The Memory

Outline

- Memory Abstraction: Address Space
- Memory APIs
- Address Translation

- Segmentation
- Free Space Management
- Paging: Introduction
- Translation Lookaside Buffers (TLB)

- Paging: Smaller Tables
- Swapping: Mechanisms and Policies

Memory Abstraction: Process Address Space

Memory Virtualization

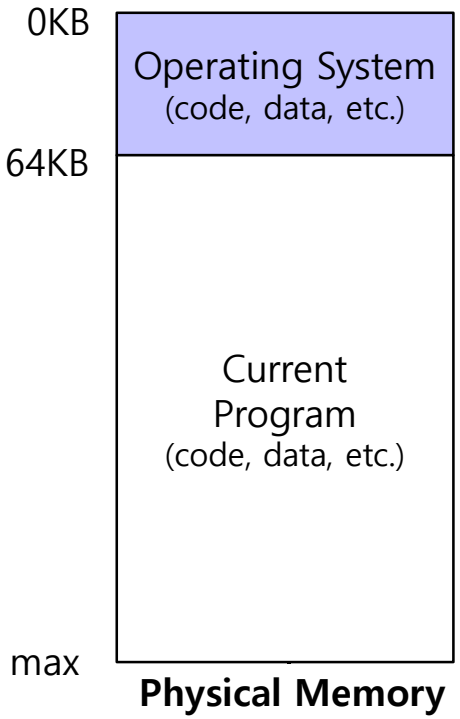
- **What is memory virtualization?**
  - OS virtualizes its physical memory.
  - OS provides an illusion of private memory space for each process.
  - It seems to be seen like each process uses the whole memory.

Benefit of Memory Virtualization

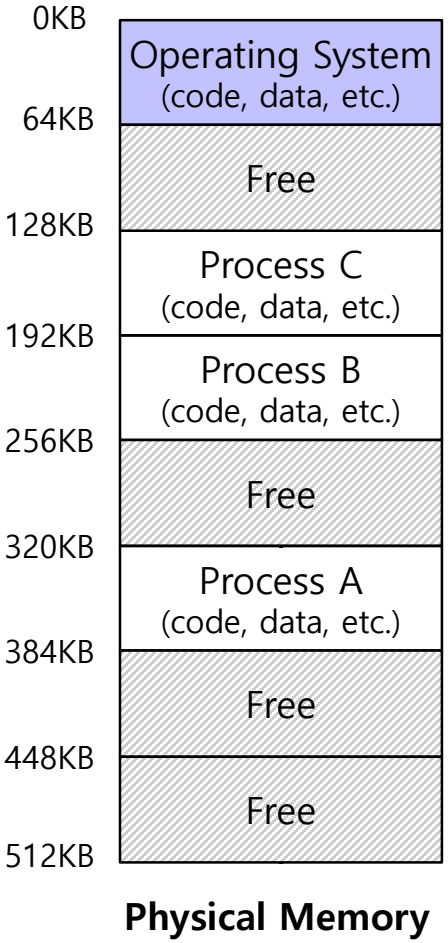
- Ease of use in programming
- Memory efficiency in terms of times and space
- The guarantee of isolation for processes as well as the OS
  - Protection from errant (straying from the acceptable course) accesses of other processes

OS in The Early System

- **Load only one process in memory:**
  - Poor utilization and efficiency



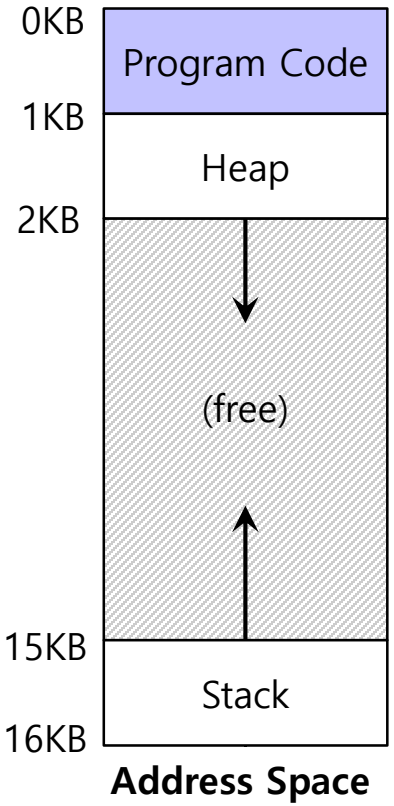
- **Load multiple processes in memory:**
  - Execute one for a short while.
  - Switch between processes in memory.
  - Increase utilization and efficiency.
- **Cause an important protection issue:**
  - Errant memory accesses from other processes



Multiprogramming and Time Sharing

Address Space

- **OS creates an abstraction of physical memory:**
  - The address space contains all info about a running process.
  - That is it consists of program code, heap, stack, etc.
- **Code:**
  - Where instructions live
- **Heap:**
  - Dynamically allocate memory:
    - **malloc**() in C language
    - **new**() in object-oriented language
- **Stack:**
  - Function calls; store return addresses or values.
  - Contain local variables & arguments to routines.



Virtual Address

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

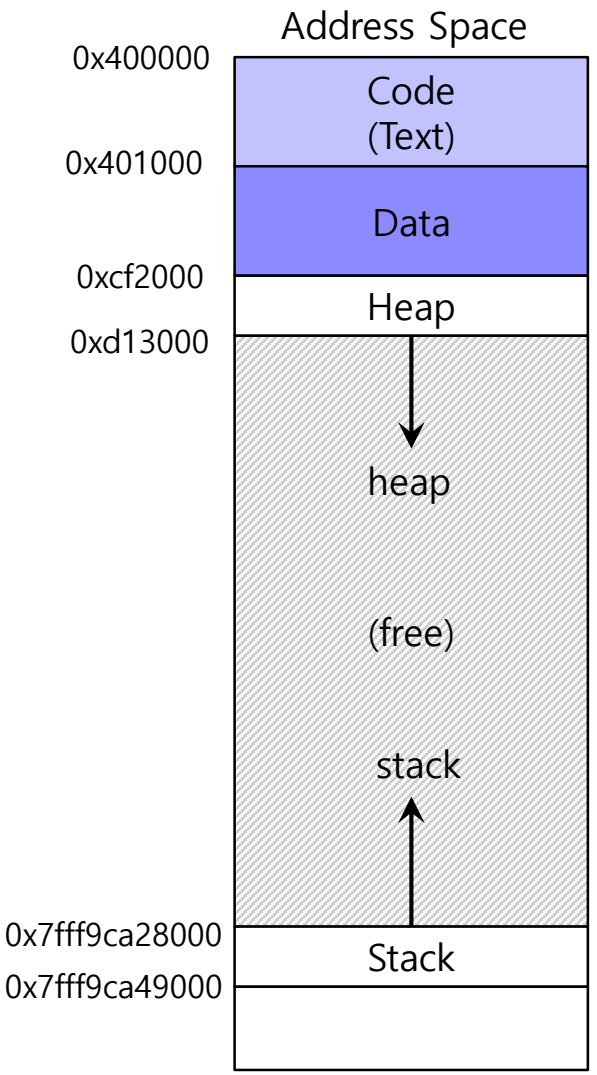
```
#include <stdio.h>
#include <stdlib.h>

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

    printf("location of code  : %p\n", (void *) main);
    printf("location of heap  : %p\n", (void *) malloc(1));
    int x = 3;              // 1st local variable allocated in the stack
    printf("location of stack : %p\n", (void *) &x);

    return x;
}
```

A simple program that prints out addresses

Virtual Address(Cont.)

- The output in 64-bit Linux machine

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



Memory APIs

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));
```

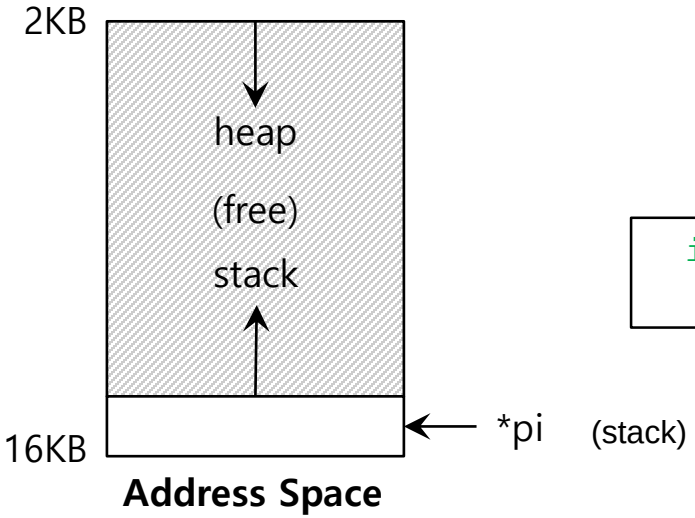
```
40
```

Memory API: free()

```
#include <stdlib.h>

void free(void* ptr)
```

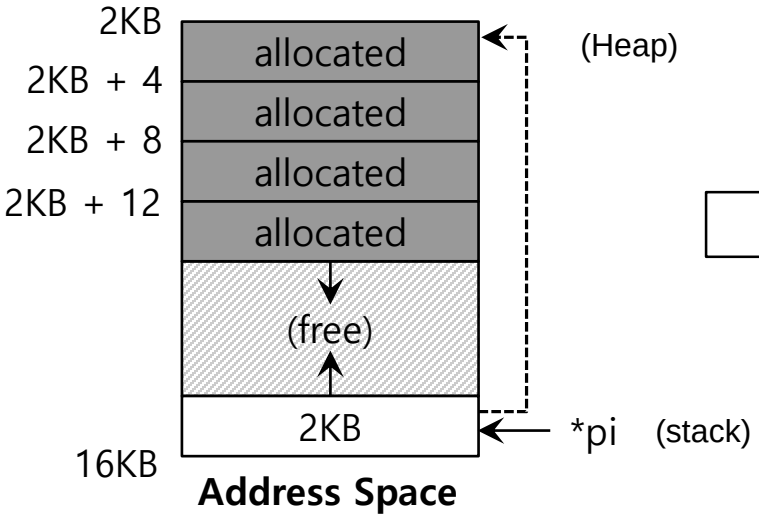
- **Free a memory region allocated earlier by a call to malloc():**
  - **Argument**
    - void *ptr: a pointer to a memory block allocated with malloc()
  - **Return**
    - None
  - How the free() call knows how many bytes to free?

Memory Allocating



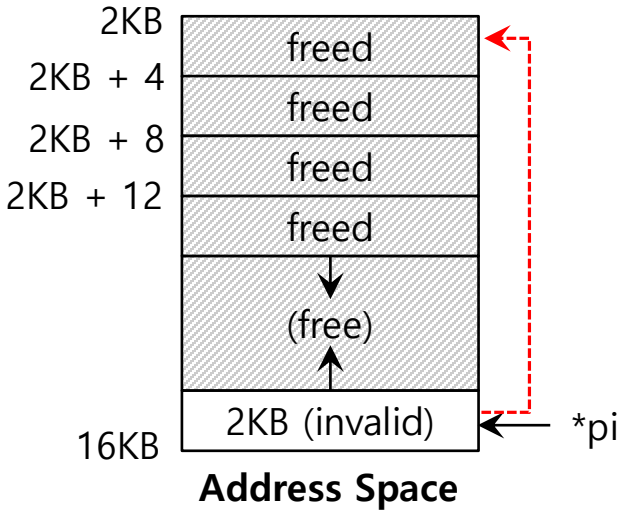
pointer

```
int *pi;    // local variable
            // allocated in the stack
```

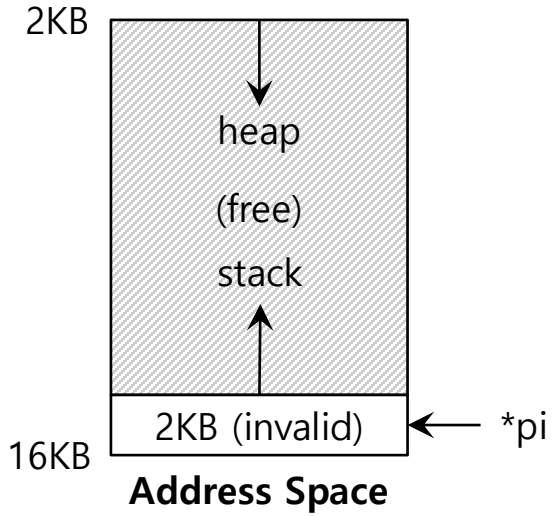



```
pi = (int *)malloc(sizeof(int)* 4);
```

Memory Freeing



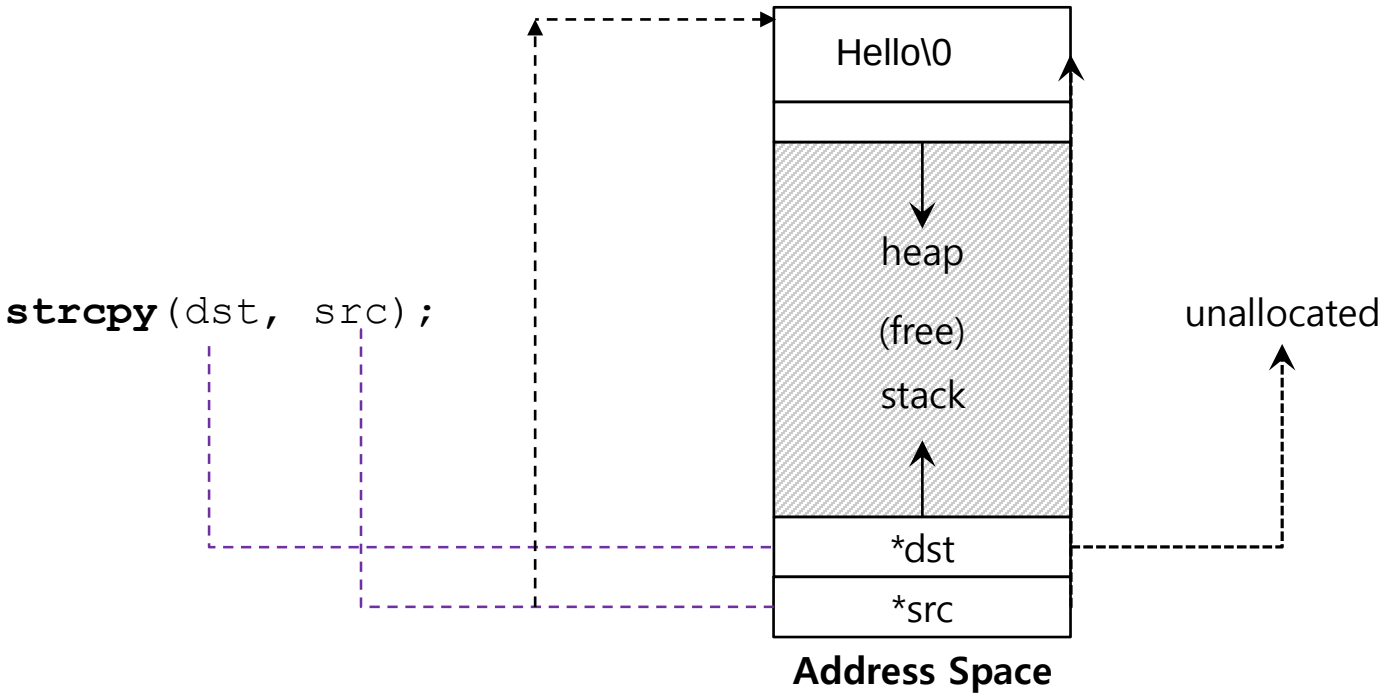

```
free(pi);
```



Forgetting To Allocate Memory

- **Incorrect code**

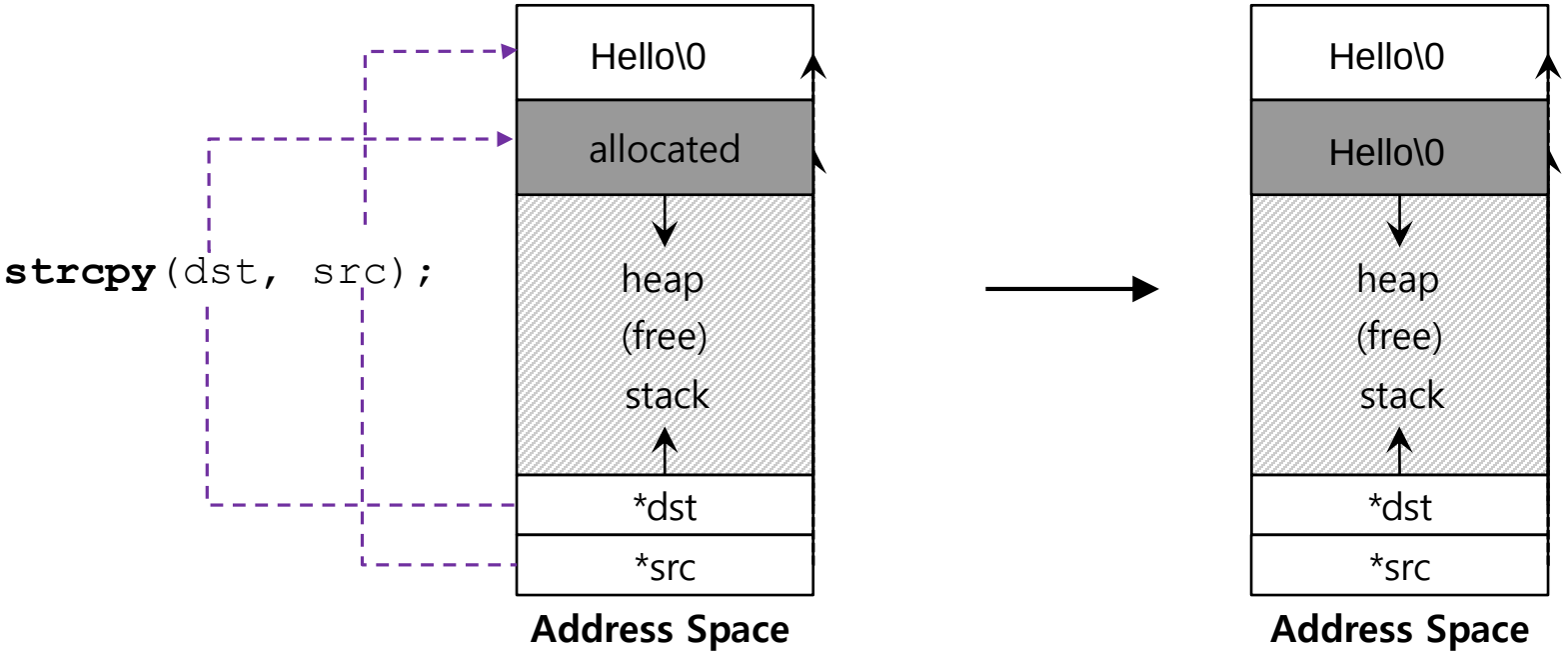
```
char *src = "hello"; //character string constant
char *dst;           //pointer - unallocated memory
strcpy(dst, src);    //segfault and die - should used strdup()
                     // or set dst to point to an allocated buffer
```



Forgetting To Allocate Memory(Cont.)

- **Correct code**

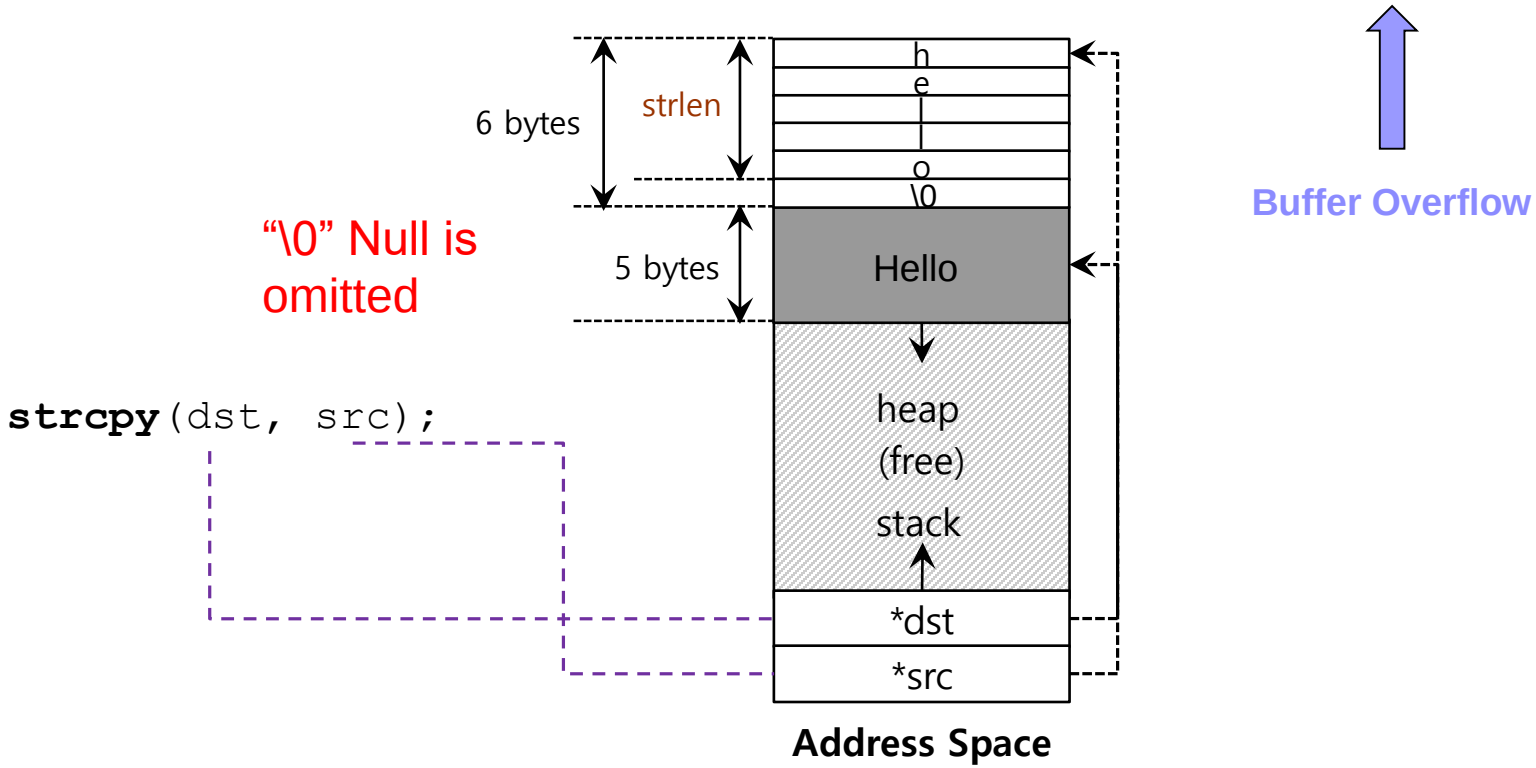
```
char *src = "hello";          //character string constant
char *dst (char *)malloc(strlen(src) + 1 );    // dst allocated
strcpy(dst, src);             //work properly
```



Not Allocating Enough Memory

- Incorrect code, but work properly

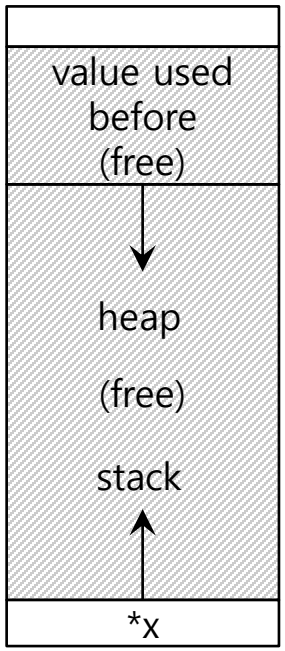
```
char *src = "hello";   //character string constant
char *dst (char *)malloc(strlen(src));        // too small
strcpy(dst, src);      //work properly
```



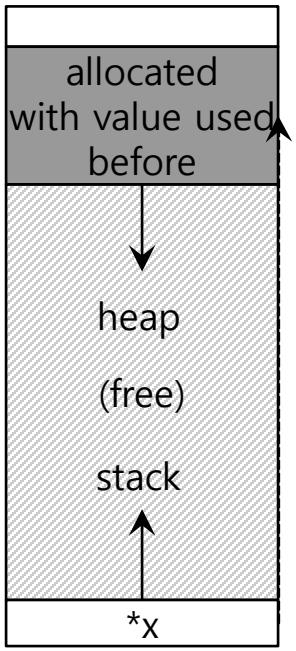
Forgetting to Initialize

- Encounter an uninitialized read

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



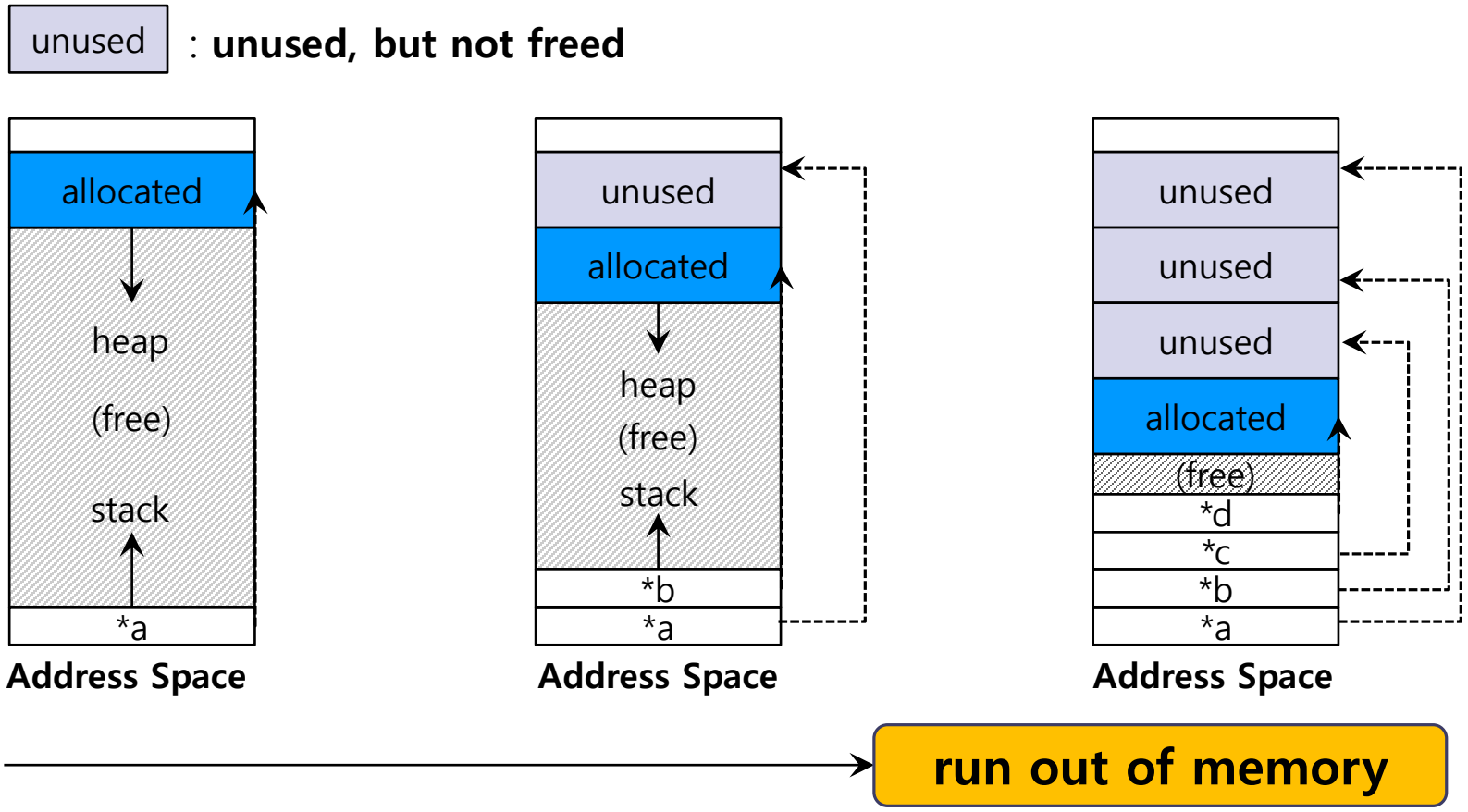
Address Space



Address Space

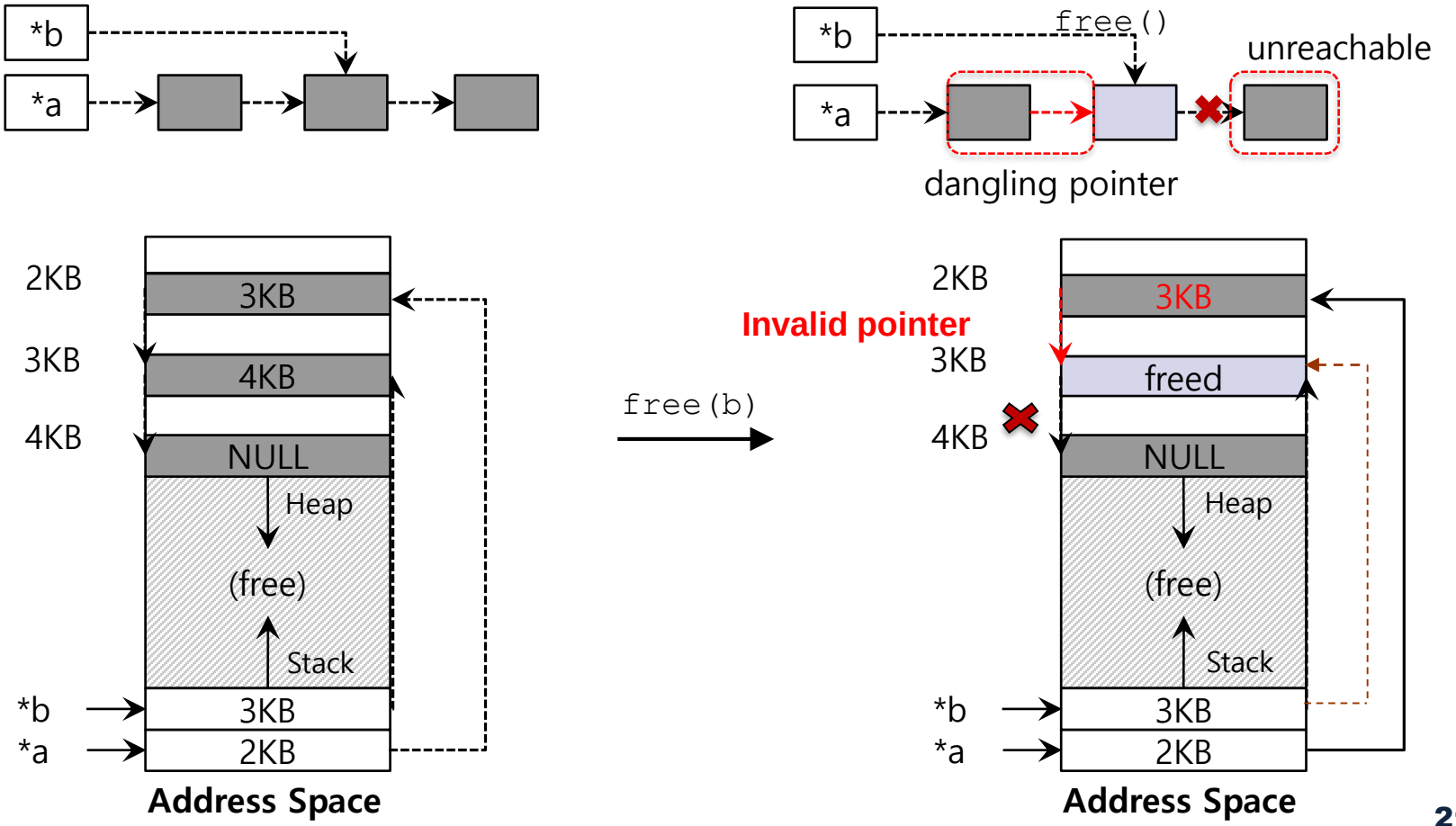
Memory Leak

- A program runs out of memory and eventually dies.



Dangling Pointer

- Freeing memory before adjusting relevant pointers
  - A program accesses to memory with invalid pointer



Other Memory APIs: calloc()

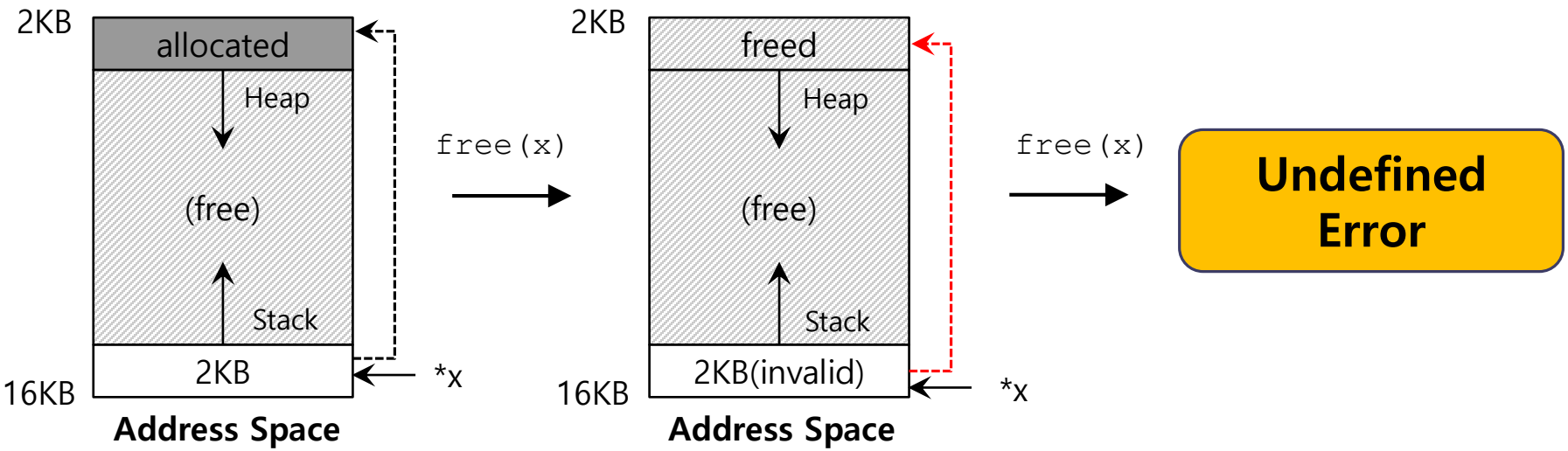
```
#include <stdlib.h>

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

- Allocate memory (array) on the heap and zeroes it before returning:
  - **Argument:**
    - **size_t num**: number of objects/blocks to allocate
    - **size_t size**: size of each object/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 (changed size of existing block) or a new (allocate new block).
  - **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 **set the program *break*** ***(end of data segment)***.
    - ***break***: The location of **the end of the data segment** in address space. Increase/decrease data segment size. Program break is the 1st address after the uninitialized data segment (end of data segment).
  - **sbrk**() is similar to brk(); it **increments**, i.e., rather than **setting**, the program data segement .
  - **brk**() set the pointer to the new end-of-data segment while sbrk() increments current end-of-data segment . Sbrk() on success returns the previous program break. brk() returns 0 on success.
  - Programmers **should never directly call** either brk() or sbrk().

System Calls(Cont.)

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

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

- **mmap**() system call can create **an anonymous (flags)** memory region, i.e., the mapping is not backed up by file but instead its content is set to zero.

Address Translation

Memory Virtualizing with Efficiency and Control

- **In memory virtualization**: efficiency and control are attained by **hardware support**.
  - e.g., registers, TLB (Translation Look-aside Buffer), and Page-Table

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

Address Translation

- **The OS must get involved** at key points to set up the hardware:
  - The OS must manage memory to judiciously intervene.
- **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 into some register
  - **Increment** the register by three
  - **Store** the register value back into memory

  Each of the above statements is atomic but the 3 together is not.

Example: Address Translation(Cont.)

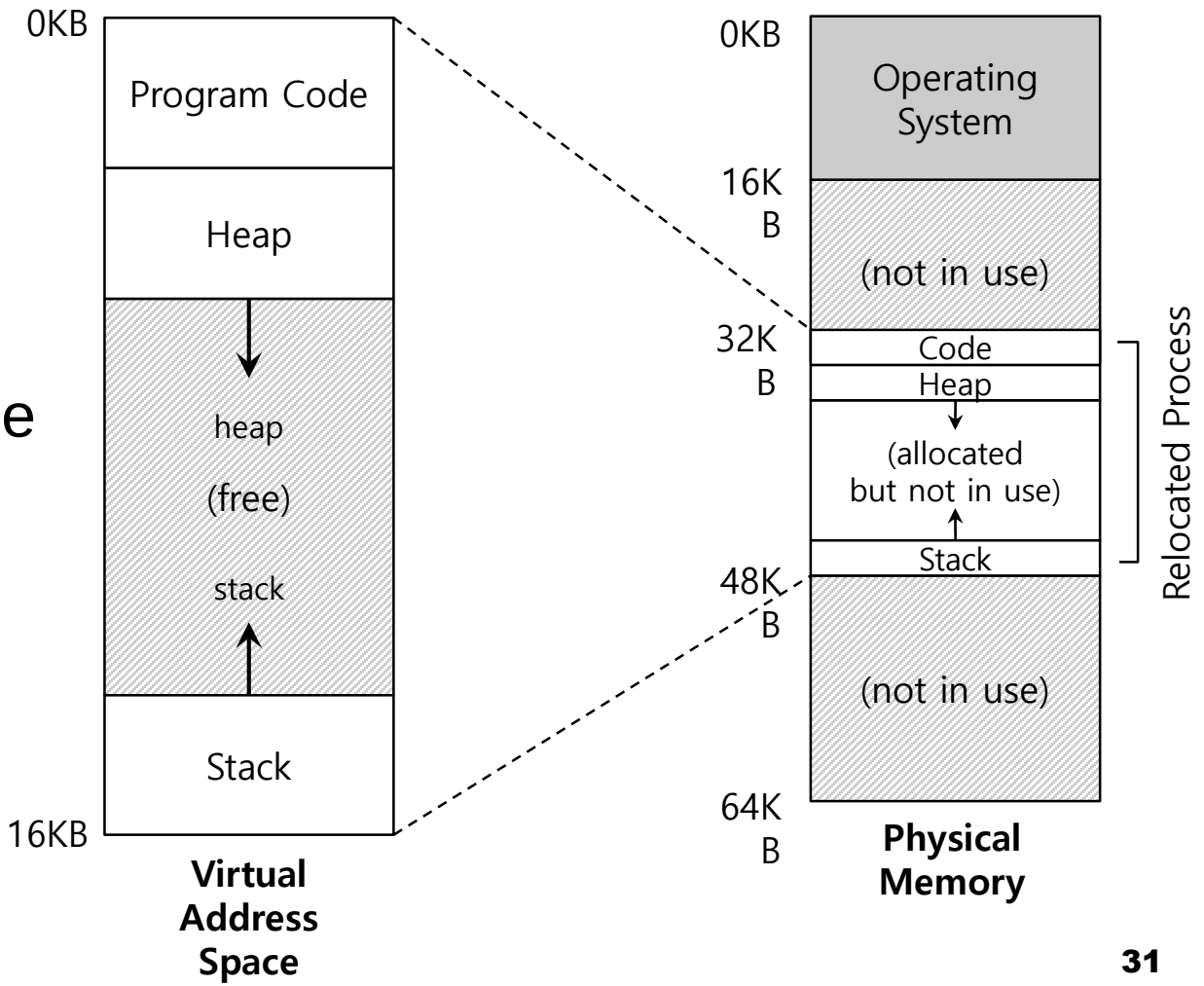
- **Assembly (for the above C-statement):**

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

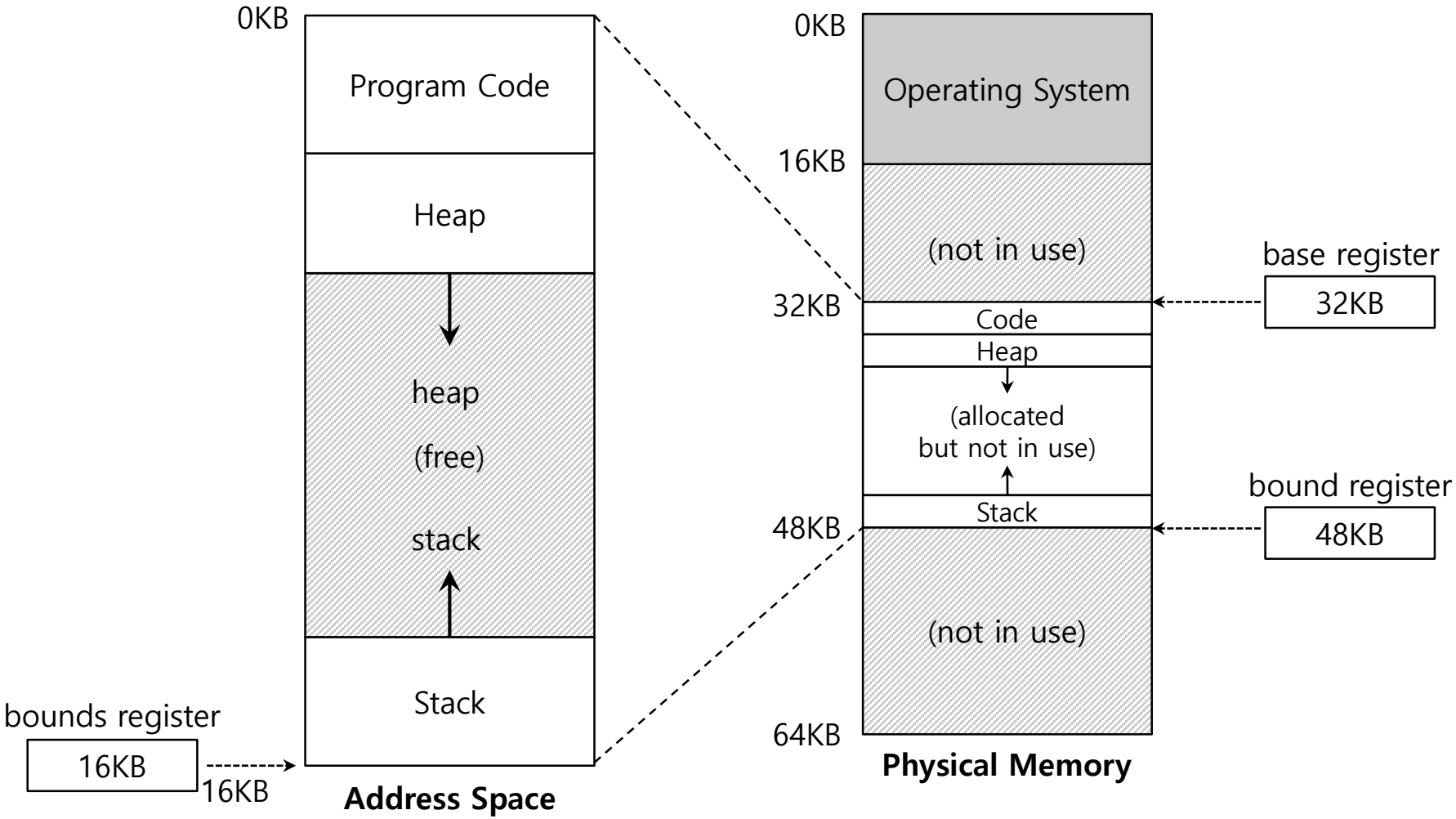
  - **Presume** that the address of ‘x’ has been placed in ebx register.
  - **Load** the value at that address into eax register.
  - **Add** 3 to eax register.
  - **Store** the value in eax back into memory.

Relocation Address Space

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



Base and Bounds Register



Dynamic (Hardware base) Relocation

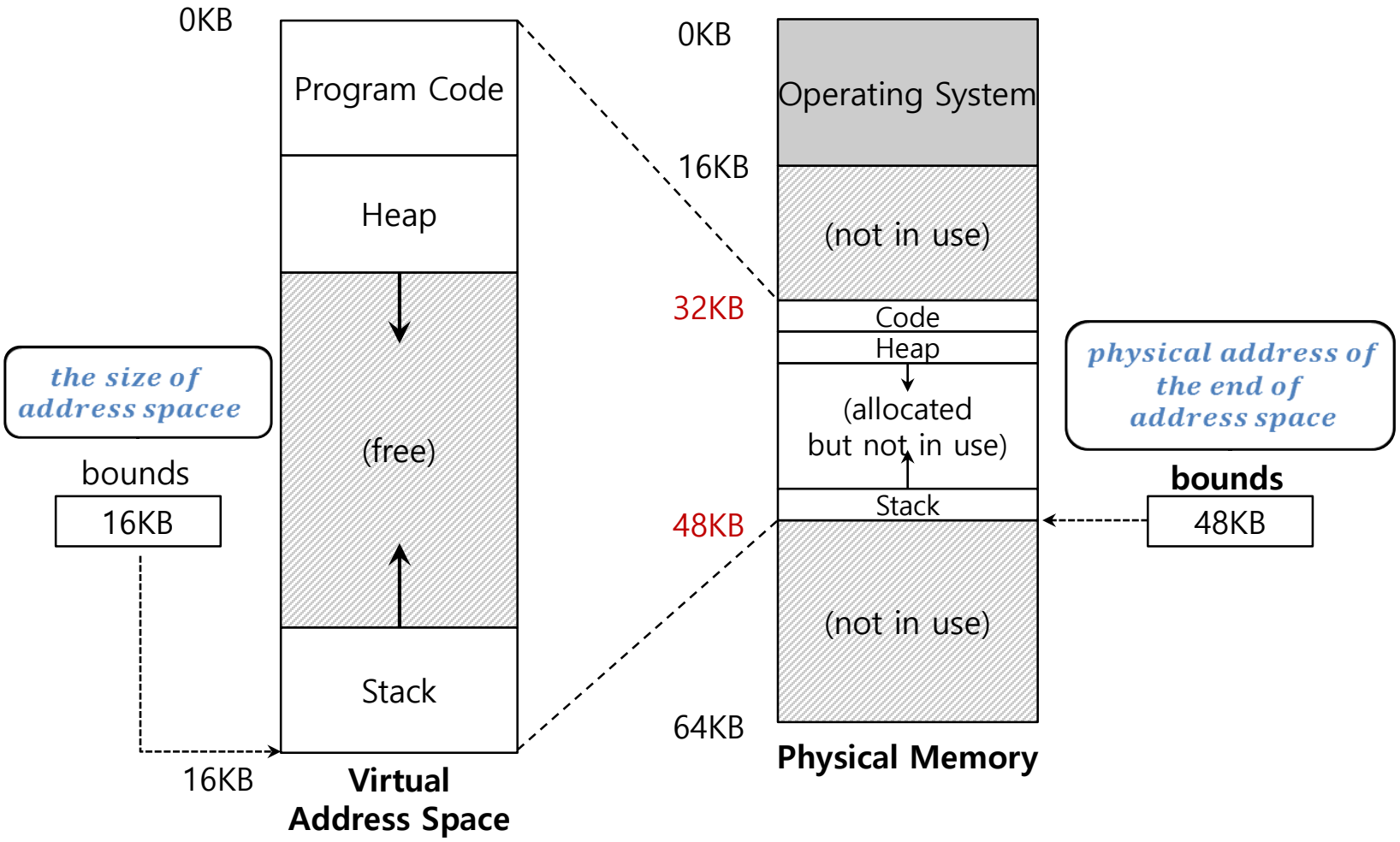
- When a program starts running, the OS decides **where** in physical memory a process should be **loaded**:
  - Set the **base** register a value.

physical address = virtual address + base

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

0 ≤ virtual address && virtual address < bounds

Two ways of Bounds Register

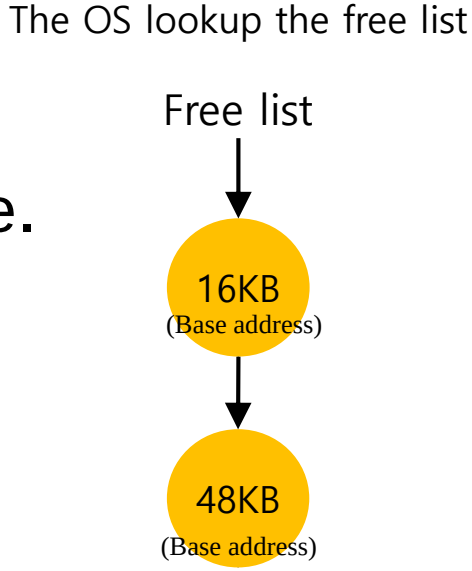


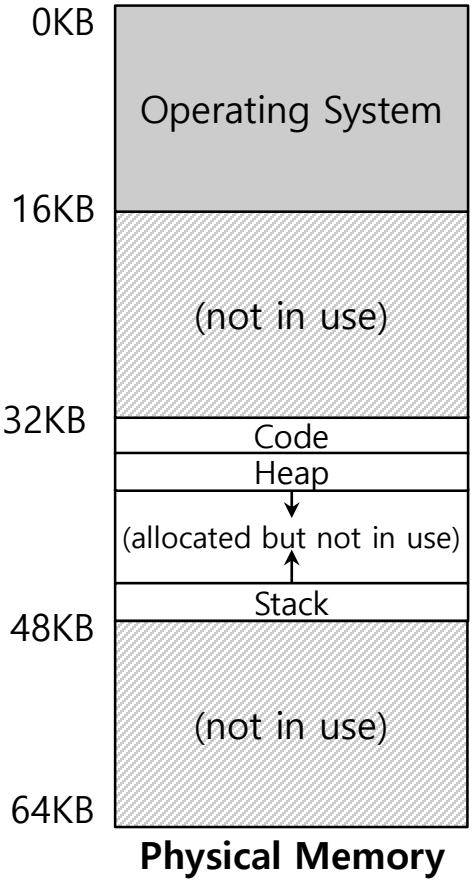
OS Issues for Memory Virtualizing

- 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 by others
  - When context **switch occurs**:
    - Saving and storing the base-and-bounds register pairs

OS Issues: When a Process Starts Running

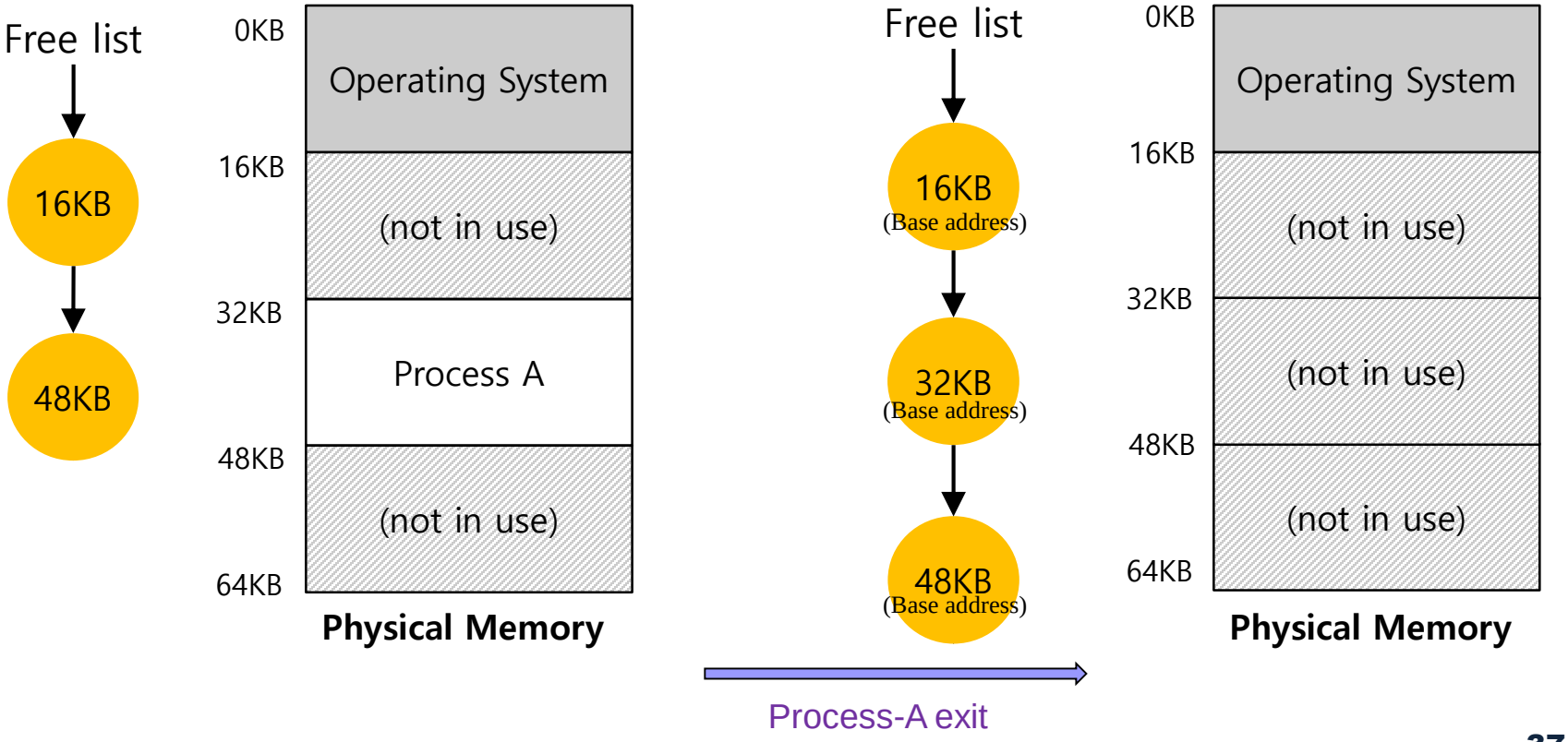
- The OS must **find a room** for a new address space.
  - **free list:** A list of the range of the physical memory which are not in use.





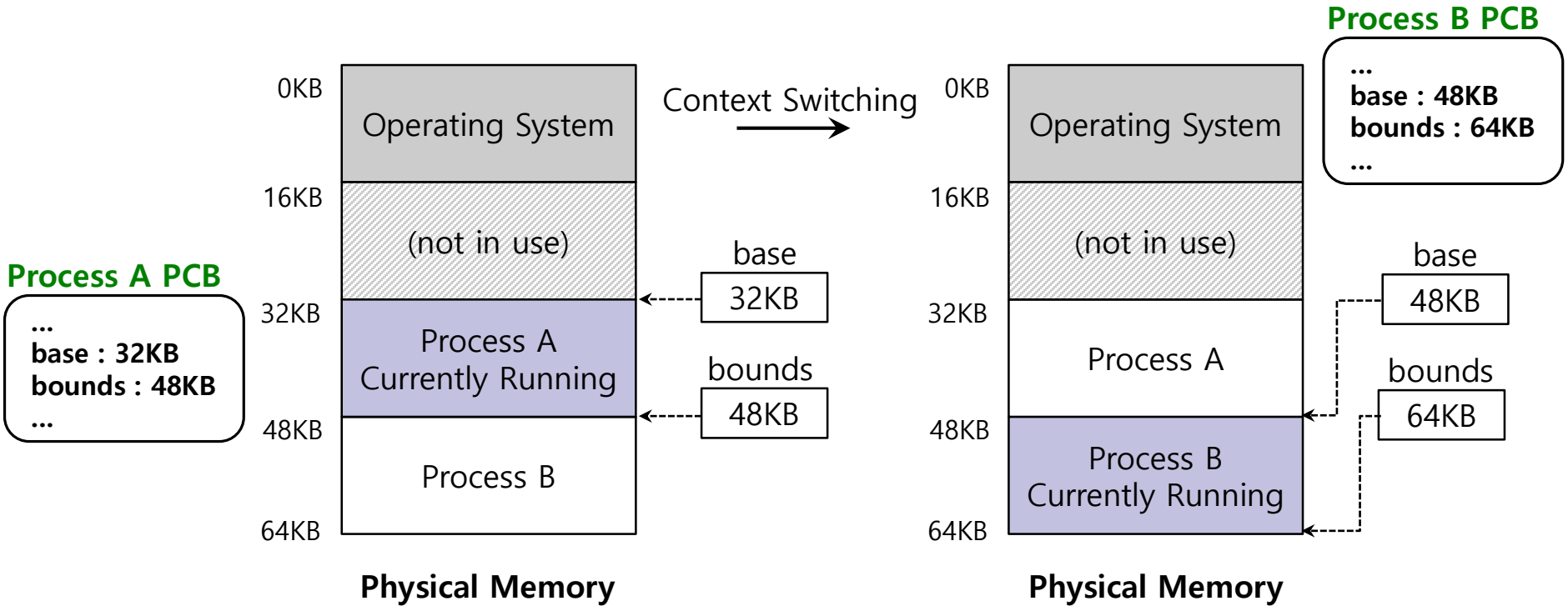
OS Issues: When a Process Is Terminated

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

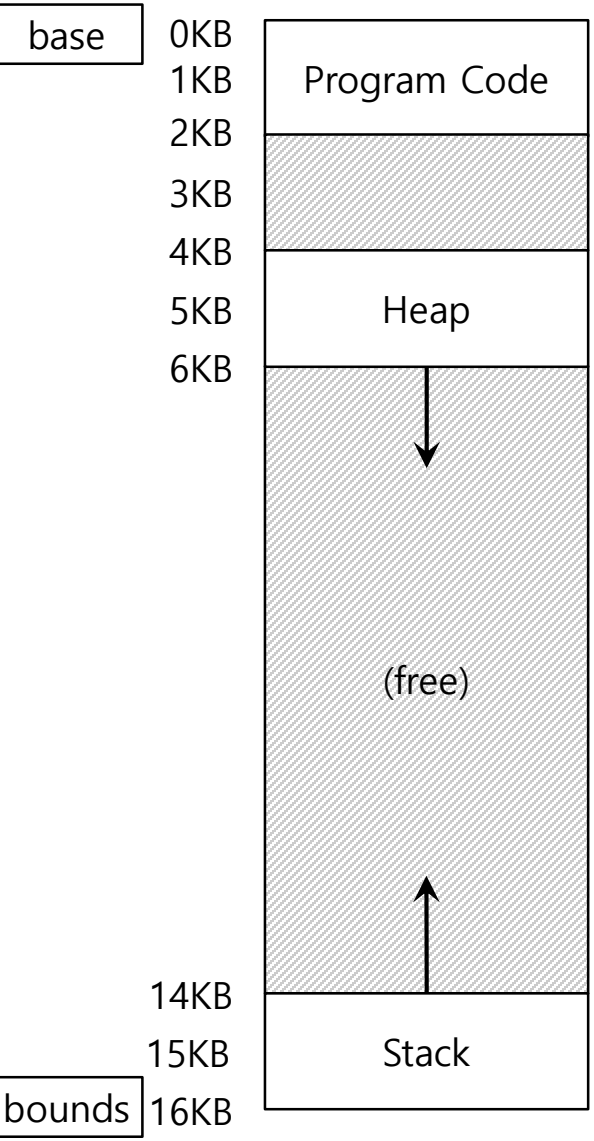


OS Issues: When Context Switch Occurs

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



Segmentation



Inefficiency of the Base and Bound Approach

- **Big chunks of "free"** space between the Base/Bound registers within one process address space
- **Scattered "free"** space **takes up** physical memory.
- **Hard to run** when an address space **does not fit** into physical memory

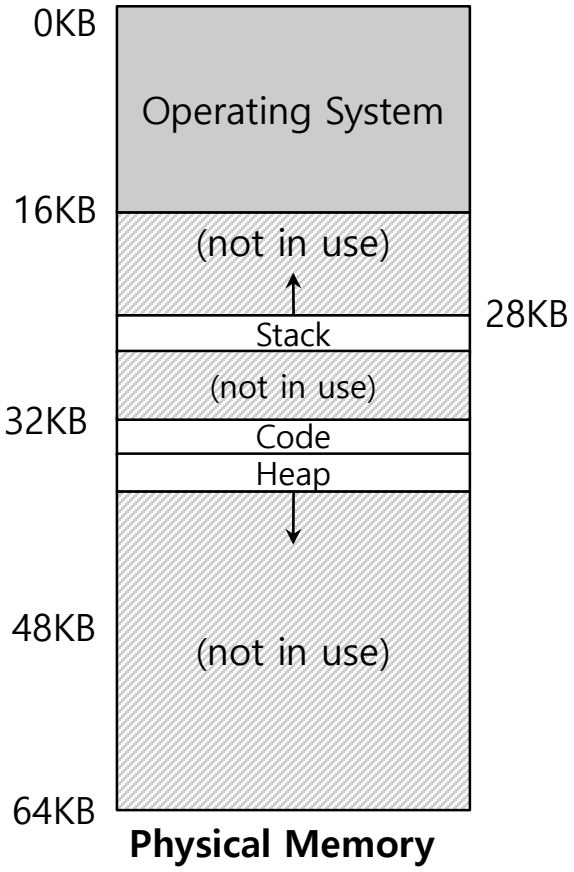
Segmentation

- **Inefficiency of the Base and Bound approach:** Hard to run when the address space does not fit into physical memory!

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

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

Placing Segment In Physical Memory

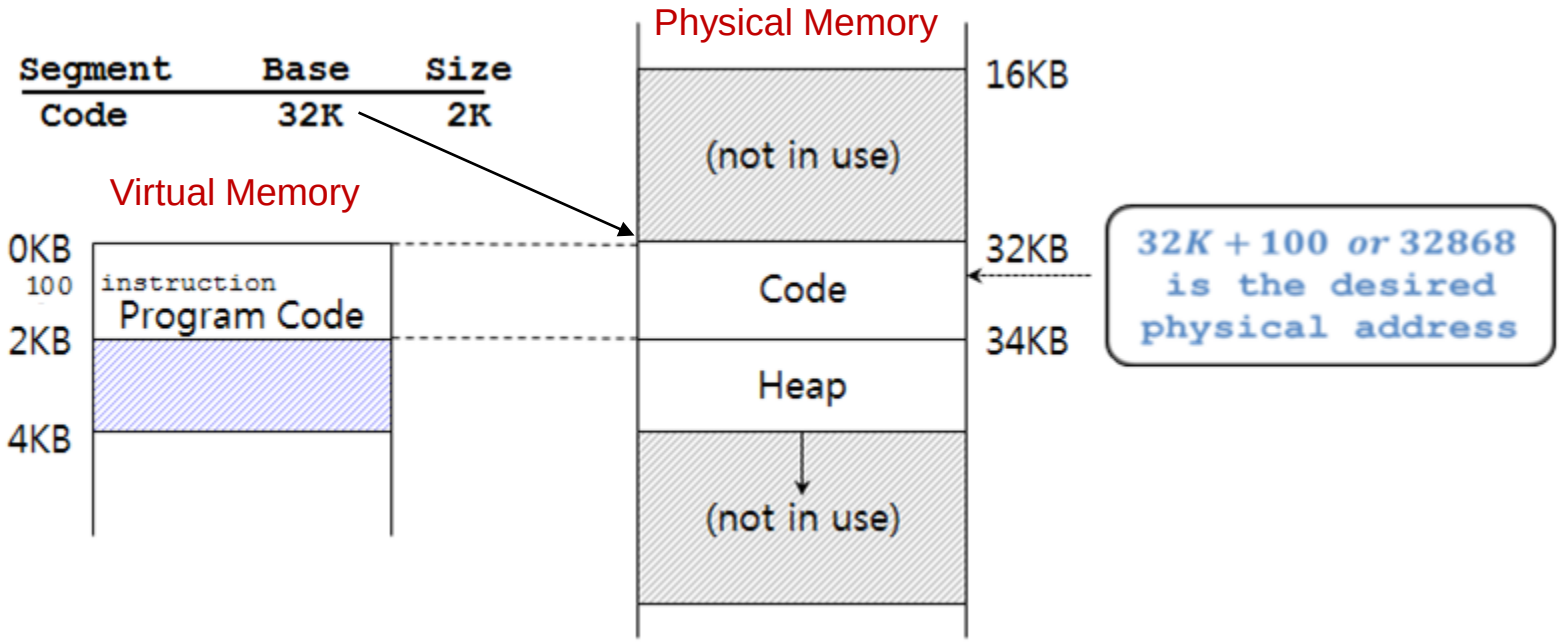


Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Address Translation on Segmentation

$$physical\ address = base + offset$$

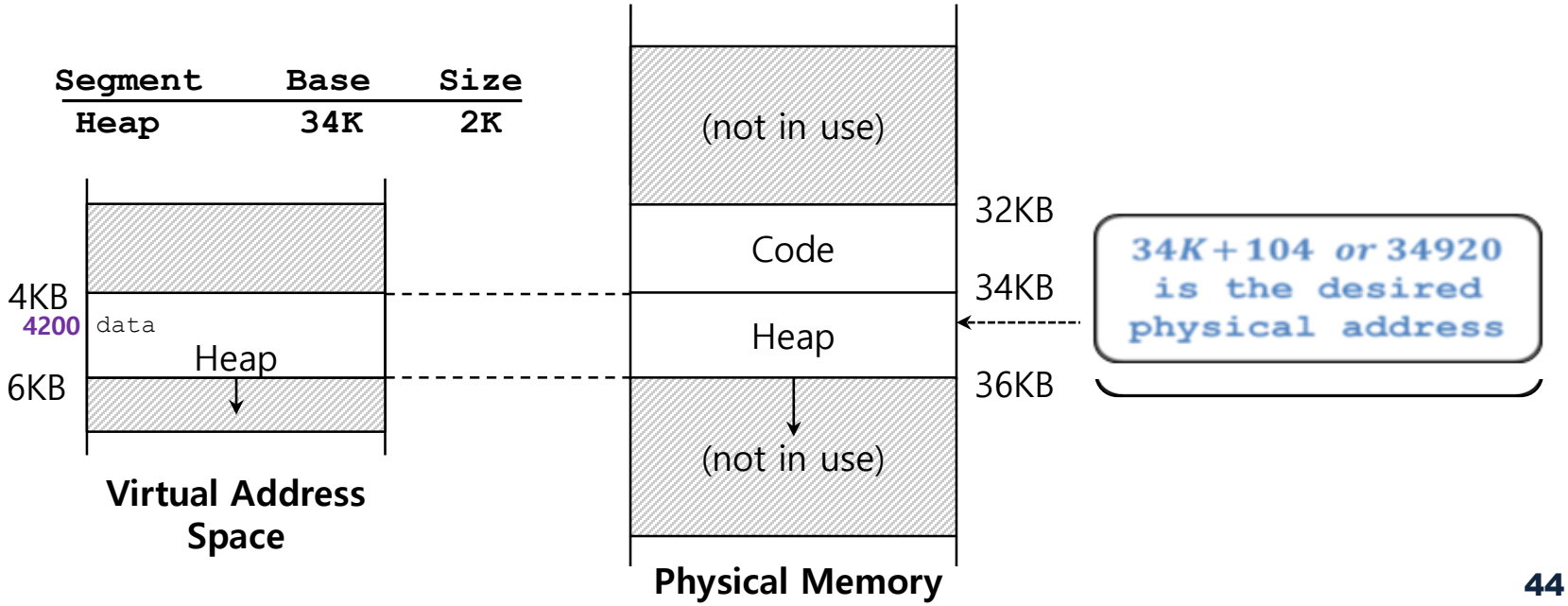
- The offset of virtual address 100 is 100 (for segment with start VA = 0):
  - The code segment **starts at virtual address 0** in the VA space. Assume the segment physical address starts at PA = 32K. Then corresponding PA to (VA = 100) is = 32K + 100



Address Translation on Segmentation (Cont.)

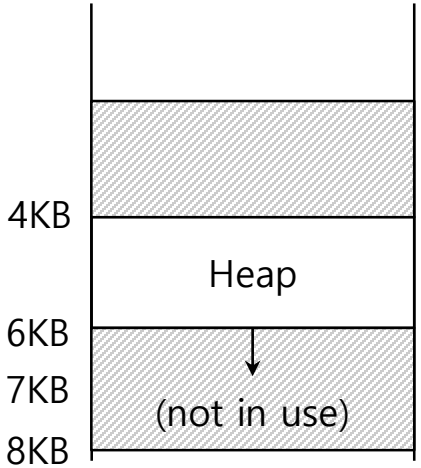
Base physical address + virtual address is not the correct physical address
Base physical address + segment offset is the correct physical address

- The offset of virtual address 4200 is (4200 – 4096 = 104).
  - Because the heap segment **starts at virtual address 4096** in the VA address space.



Segmentation Fault or Violation

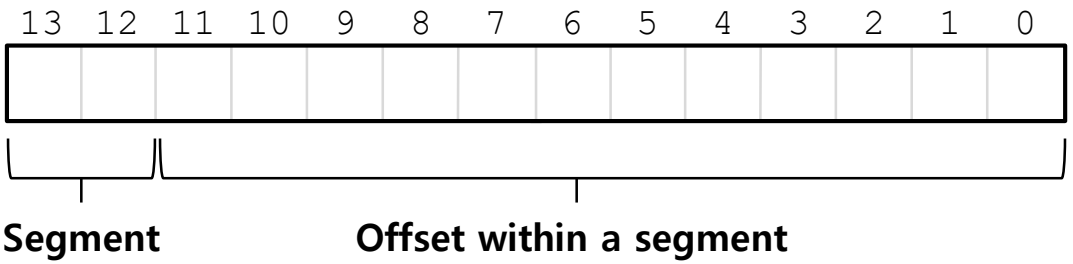
- If an **illegal address** such as 7KB which is beyond the end of heap is referenced, the OS generates **segmentation fault**.
- Valid VA that is not mapped into physical address → **segmentation fault**
- The hardware detects that address is **out of bounds**.



Address Space

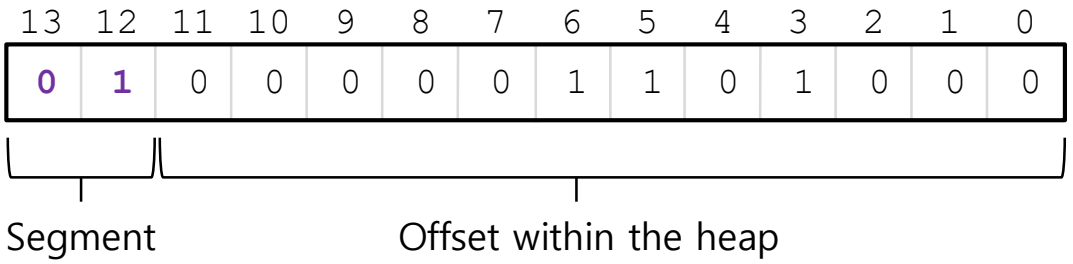
Referring to Segment

- **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



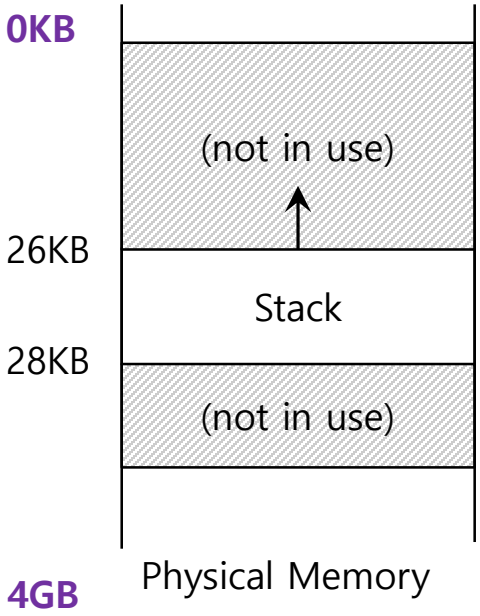
Referring to Segment(Cont.)

```
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(110000000000000)
- SEG_SHIFT = 12
- OFFSET_MASK = 0xFFF (001111111111111)

Referring to Stack Segment

- Stack grows **backward**.
- **Extra hardware support** is need.
  - 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 spaces**.
  - **Code sharing** is in use in systems today.
  - By extra hardware support.
- **Extra hardware support** is needed for a 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

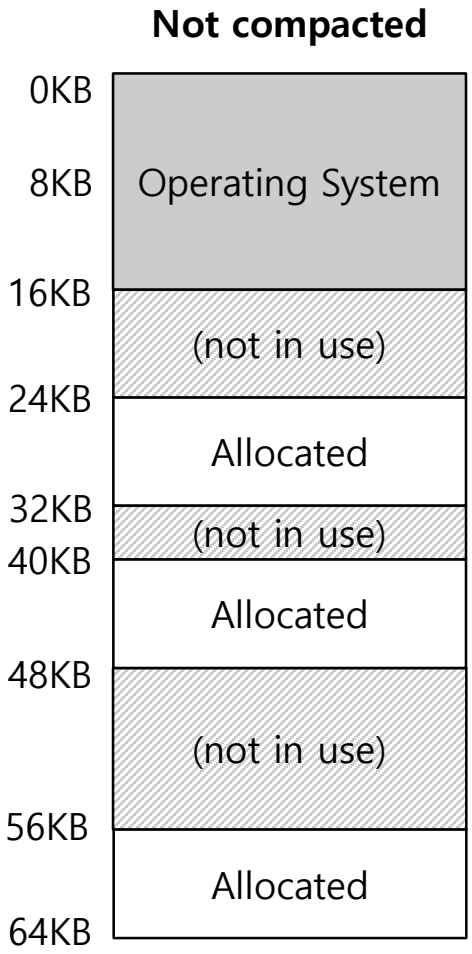
Fine-Grained and Coarse-Grained

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

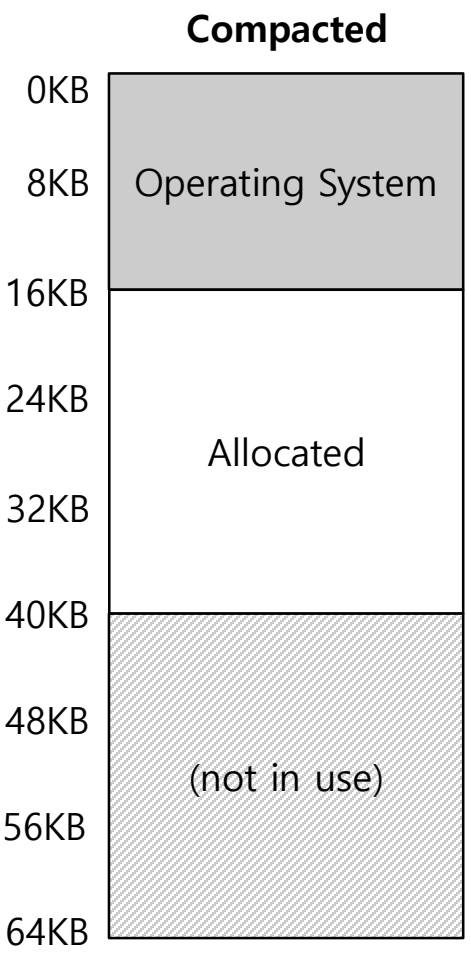
OS support: Fragmentation

- **External Fragmentation**: little holes of **free space** in physical memory that make it difficult to allocate new segments.
  - There is total **24KB free**, but **not in one contiguous** segment, i.e., memory fragmentation.
  - The OS **cannot** satisfy a **20KB request**!
- **Compaction**: **rearranging** the exiting segments in physical memory.
  - Compaction is **costly**.
    - **Stop** running process.
    - **Copy** data to somewhere, i.e., new location.
    - **Change** segment register value.

Memory Compaction



Free memory = 8 + 8 + 8 = 24KB

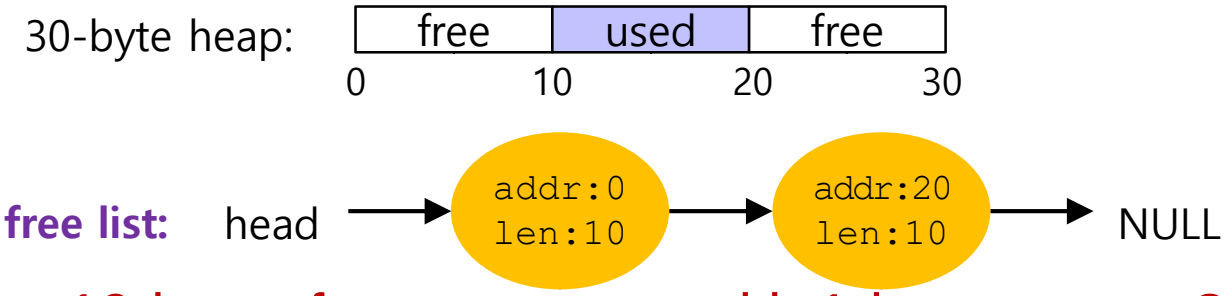


Free memory = 64 - 40 = 24KB

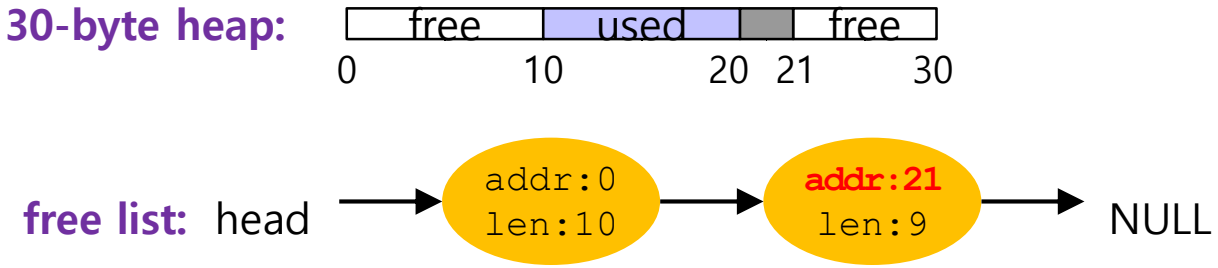
Free Space Management

Splitting

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



  - Two 10-bytes free segments with 1-byte request?



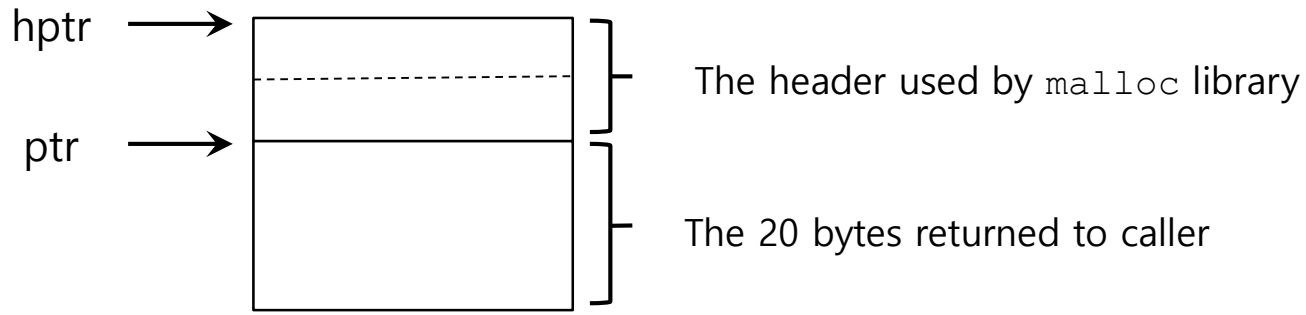
Tracking The Size of Allocated Regions

- **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** (fixed size) block.

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

  - The **size** for free region is the **size of the header plus the size of the space** allocated to the user.

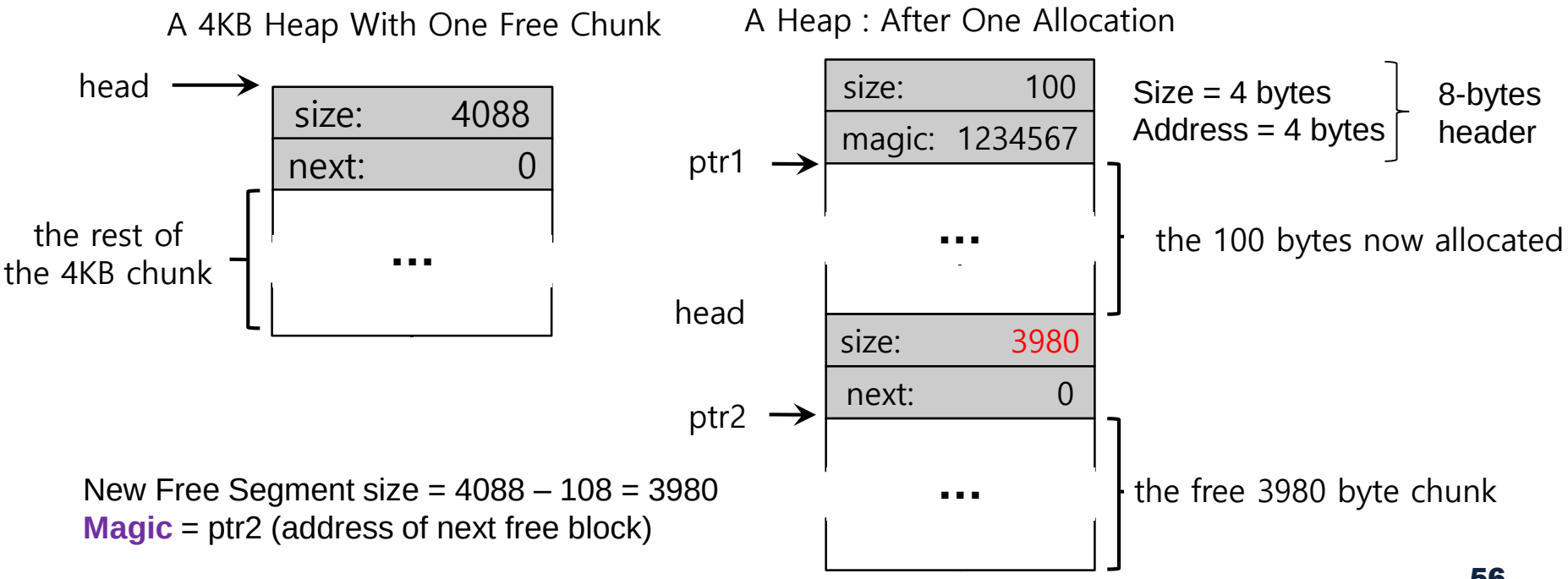
```
ptr = malloc(20);
```



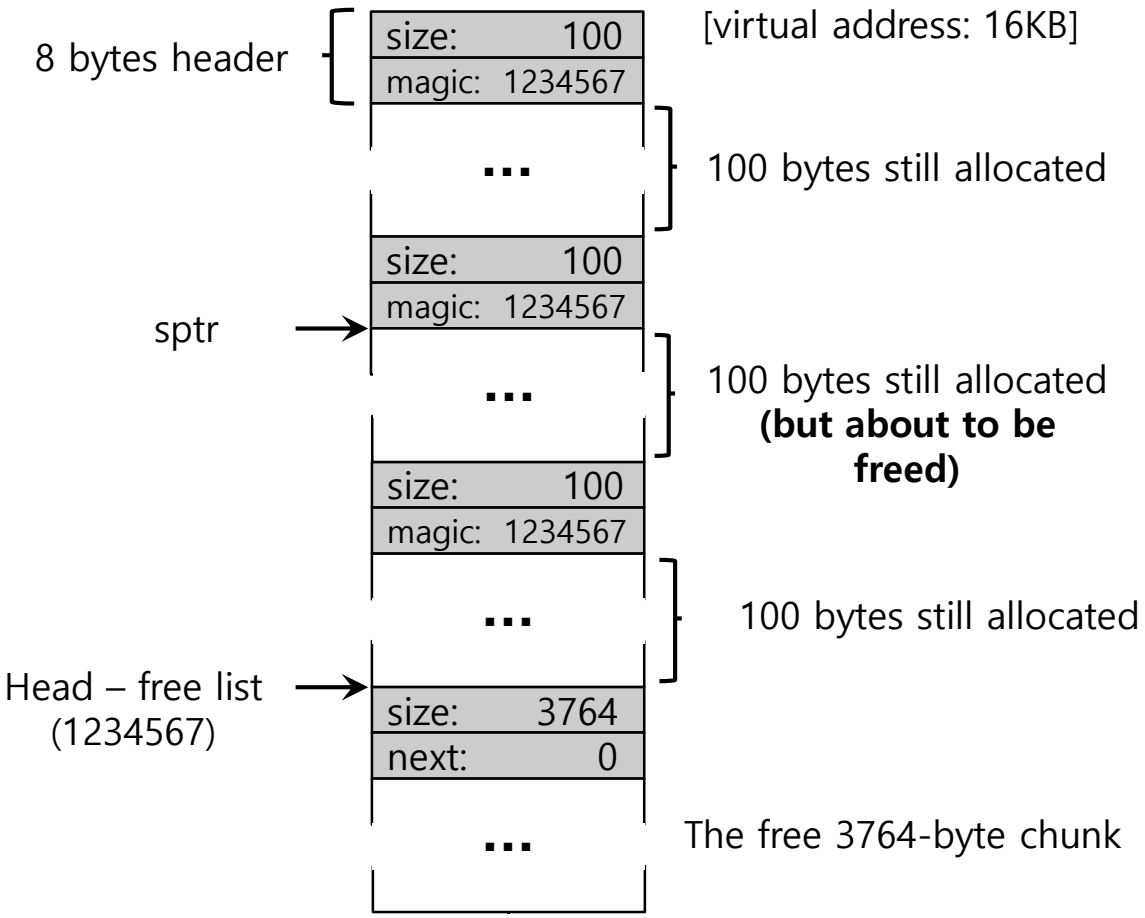
An Allocated Region Plus Header

Embedding A Free List: Allocation

- **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).



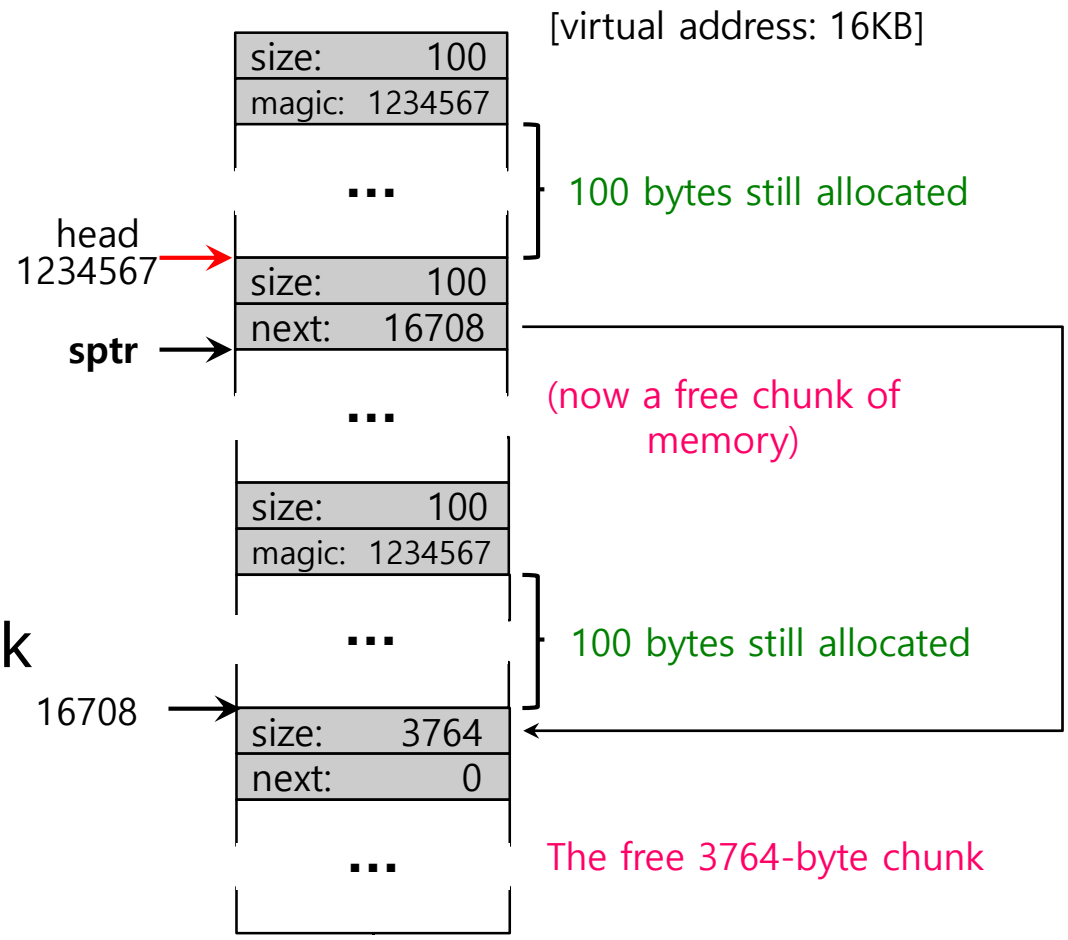
Free Space With Chunks Allocated



Free Space With Three Chunks Allocated

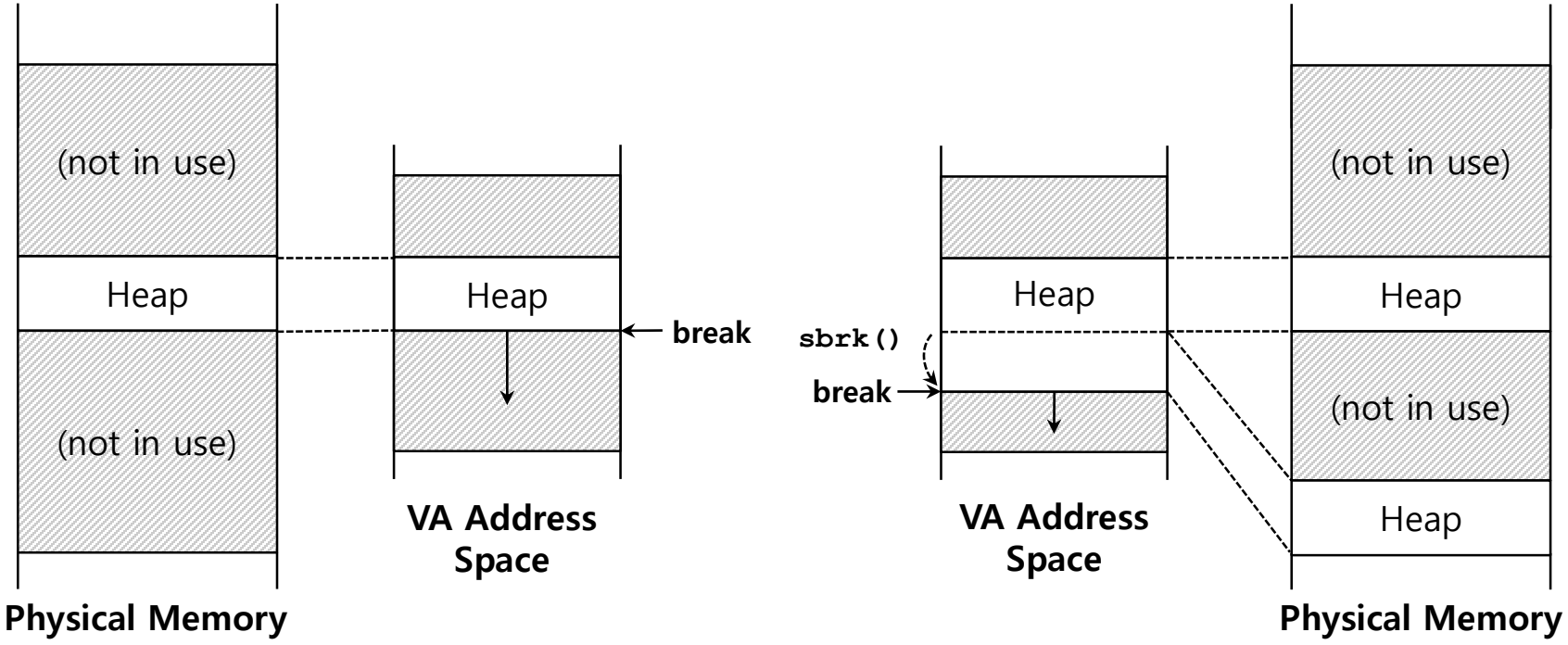
Free Space With free()

- **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 to the small chunk



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(increment), brk(end of heap segment) in most UNIX systems. ← called by malloc()



Managing (Allocating) Free Space: Basic Strategies

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

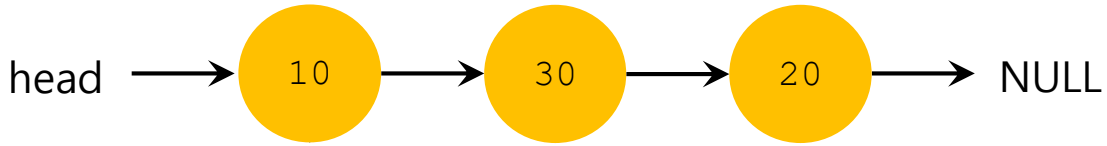
- **Worst Fit:**
  - Finding the **largest free chunks** and allocation the amount of the request
  - **Keeping the remaining chunk** on the free list.

Managing Free Space: Basic Strategies(Cont.)

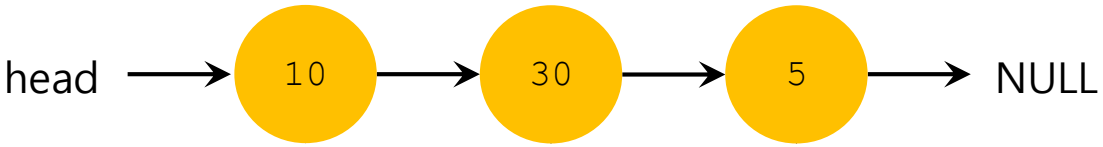
- **First Fit:**
  - Finding the **first chunk** that is **big enough** for the request – start from the beginning of the list.
  - Returning the requested amount and keep/remaining the rest of the chunk.
- **Next Fit:**
  - Finding the first chunk that is big enough for the request – Start from where you were last search
  - Searching at **where one was looking** at instead of the begging of the list.

Examples of Basic Strategies

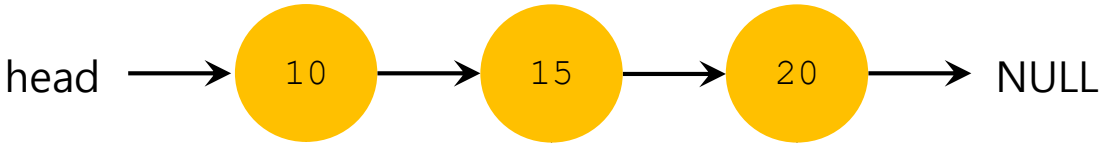
- Allocation Request Size 15



- Result of Best-fit

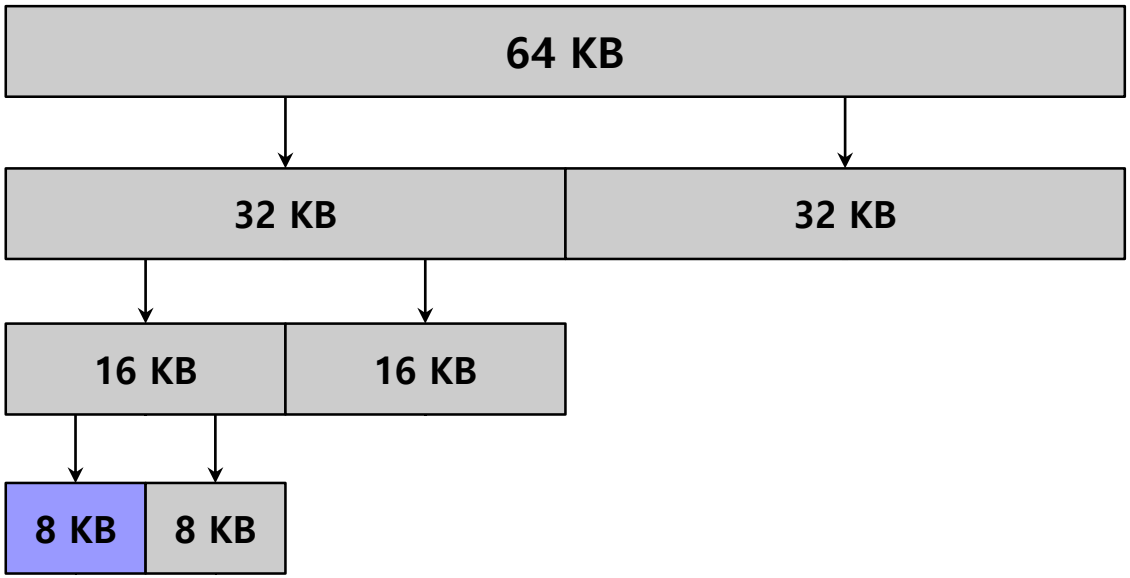


- Result of Worst-fit



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 and 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.

Paging - Introduction

Concept of Paging

- **Paging splits up address space** into **fixed-sized** unit called a **page (virtual page)**.
  - **Segmentation:** variable size of logical segments (code, stack, heap, etc.)

- **With paging, physical memory** is also **split** into some number of physical pages called a **page frame**.

- **Page table** per process is needed **to translate** the virtual address to physical address.

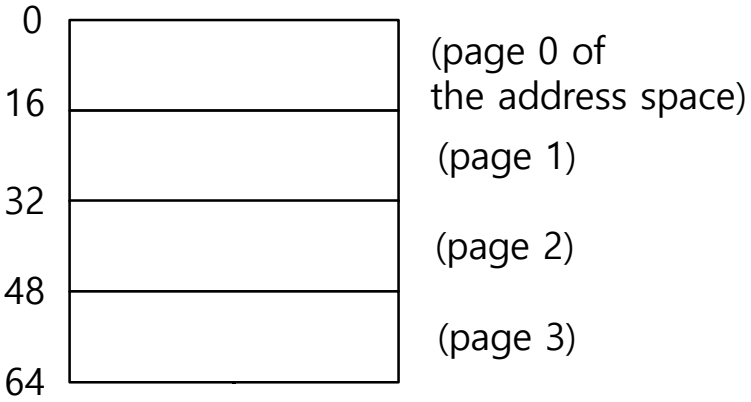
Advantages Of Paging

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

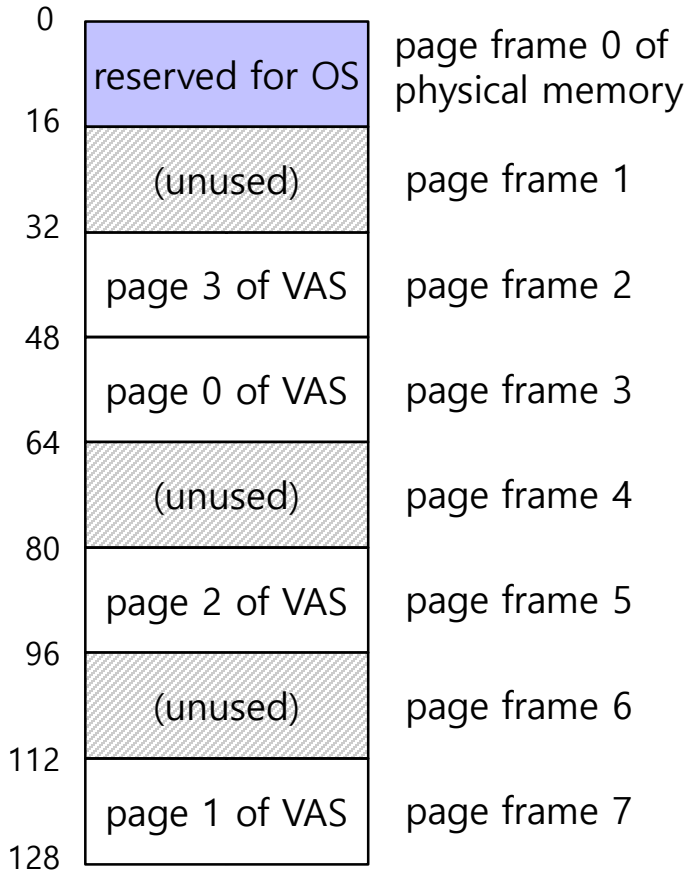
- **Simplicity**: ease of free-space management
  - The page in address space and the page frame are the same size.
  - Easy to allocate and keep a free list

Example: A Simple Paging

- 128-byte physical memory with 16 bytes page frames (8-Frames)
- 64-byte address space with 16 bytes virtual pages



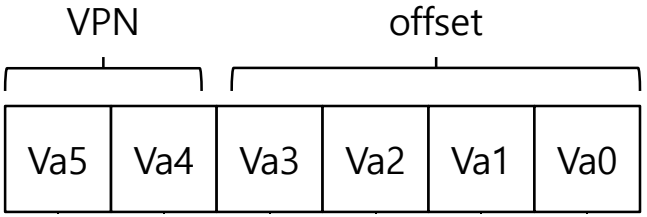
A Simple 64-byte Address Space



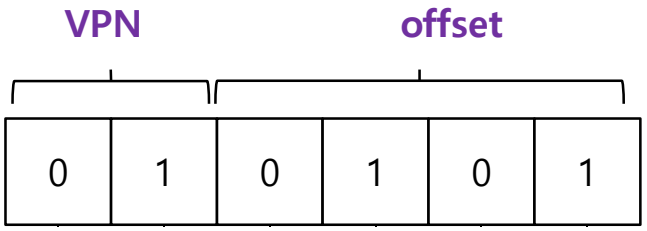
64-Byte virtual Address Space Placed In Physical Memory

Address Translation

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



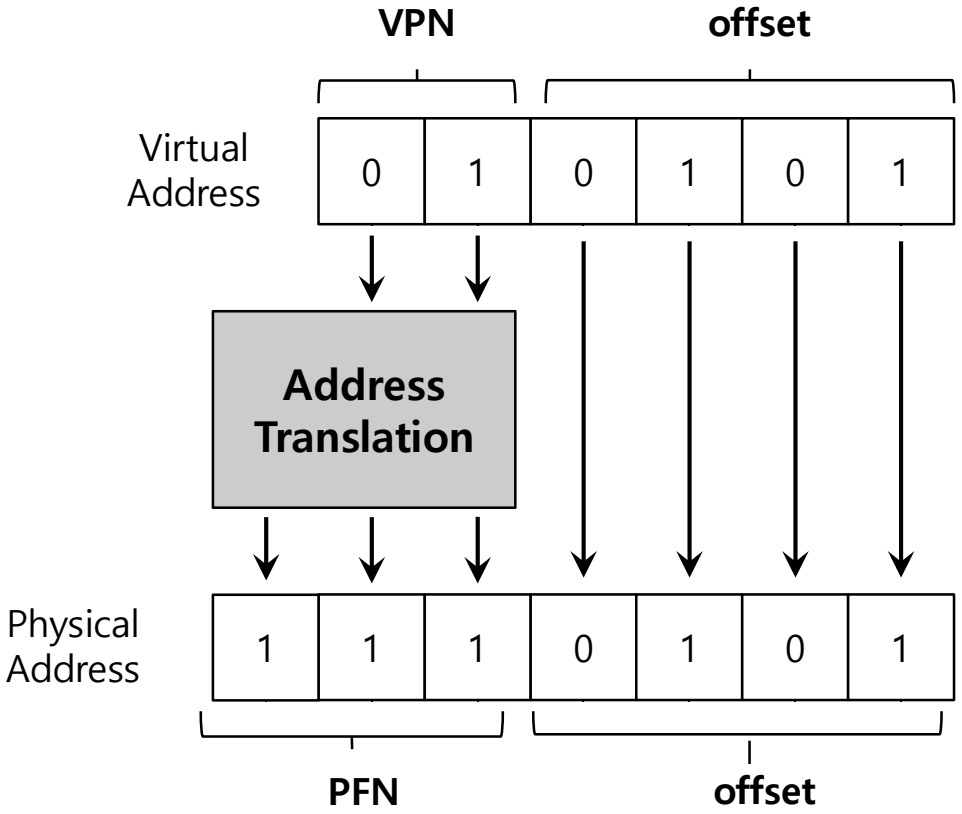
- **Example:** virtual address 21 in 64-byte address space (6-bits) ➔



VPN = 1 (2-bits)
Offset = 5
VP size = 16 bytes (4-bits)

Example: Address Translation

- The virtual address 21 in 64-byte address space (6-bits) mapping into 7-bits physical memory.

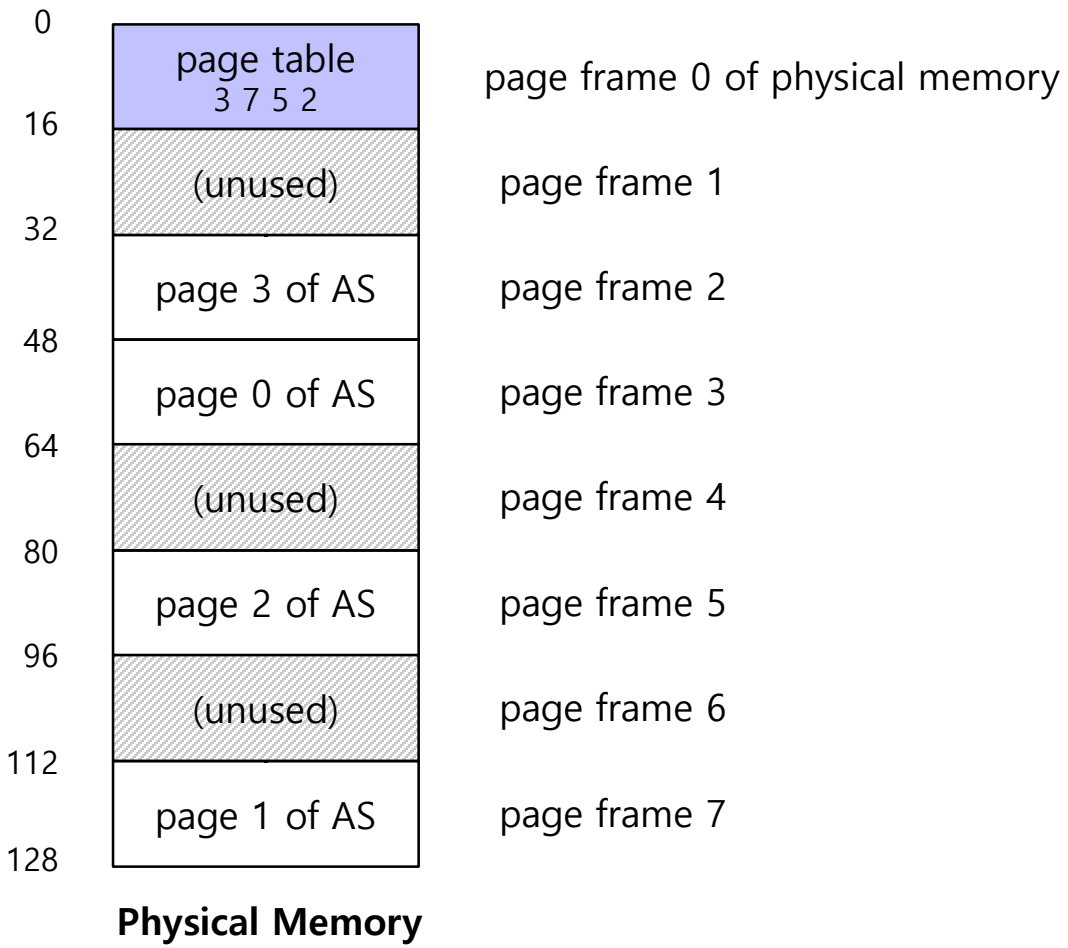


Where Are Page Tables Stored?

- **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 each process are stored in memory

Example: Page Table in Kernel Physical Memory

AS: Address Space



What Is In The Page Table?

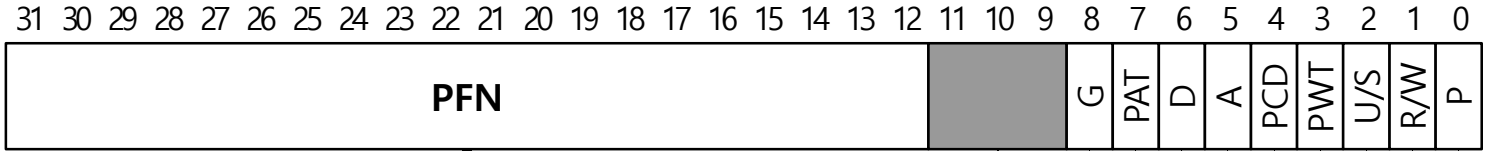
- **The page table is just a data structure** that is used to map the virtual address to physical address. Page table resides in the kernel physical memory:
  - **Simplest form:** a linear page table, i.e., an array

- The OS **indexes** the PT array by VPN, and looks up the page-table entry (PTE) to get the corresponding Frame number.

Common Flags Of Page Table Entry

- **Valid Bit**: Indicating whether the particular entry (translation) is valid.
- **Protection Bit**: Indicating whether the page could be **R**ead by user x, **W**ritten by x, or e**X**ecuted by x.
- **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**: user/supervisor bit
- **A**: accessed bit
- **D**: dirty bit
- **PFN**: the page frame number

Paging: Too Slow

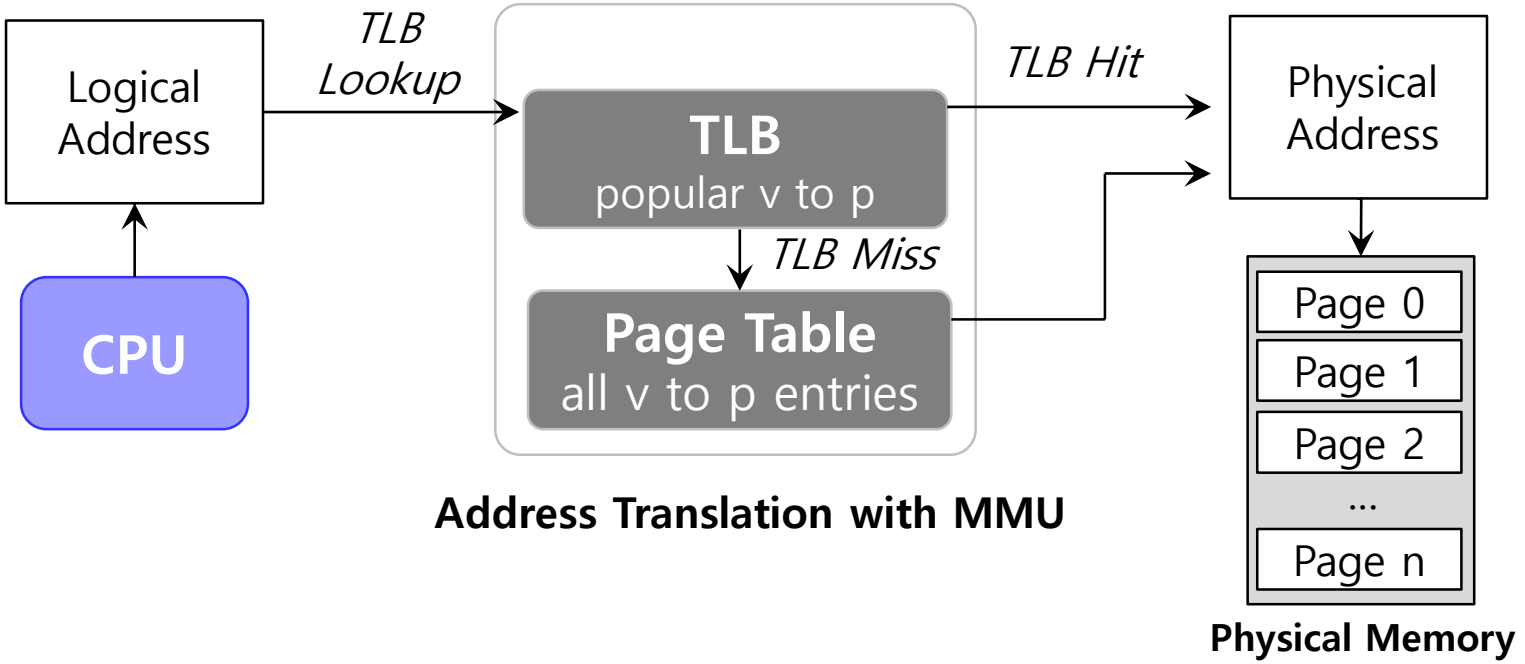
- To find a location of the desired PTE, the **starting location** of the page table is **needed**.
- For every memory reference, paging requires the OS to perform one **extra memory reference**.

```
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)
```

Translation Lookaside Buffers (TLB)

TLB

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



Address Translation with MMU

TLB Basic Algorithm

```
1: VPN = (VirtualAddress & VPN_MASK ) >> SHIFT
2: (Success , TlbEntry) = TLB_Lookup(VPN)
3:    if(Success == True){       // 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)
```

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

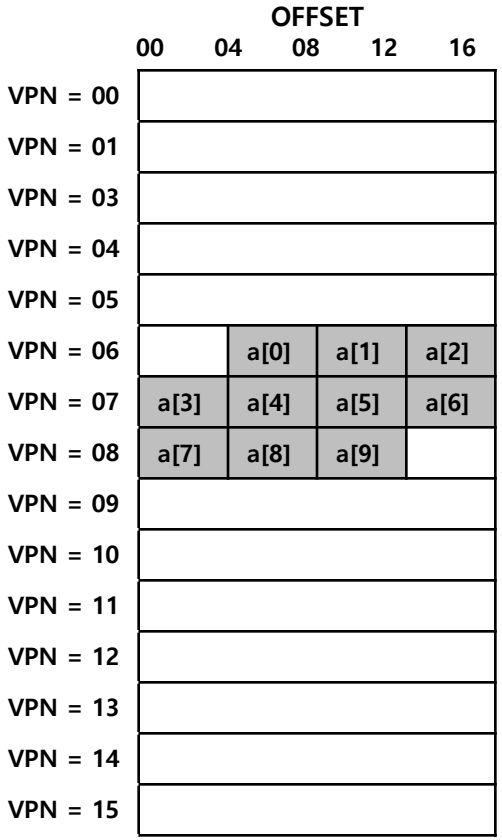
TLB Basic Algorithms (Cont.)

```
11:     }else{                  //TLB Miss
12:         PTEAddr = PTBR + (VPN * sizeof(PTE))     // PTBR: Page Table Base register
13:         PTE = AccessMemory(PTEAddr)
14:         (…)
15:         //Update TLB then retry the instruction
16:         TLB_Insert( VPN , PTE.PFN , PTE.ProtectBits)
17:         RetryInstruction()
18:     }
19:}
```

- (lines 11-12) The hardware accesses the page table to find the translation.
- (line15-16) updates the TLB with the translation.

Example: Accessing An Array

- How a TLB can improve its performance.



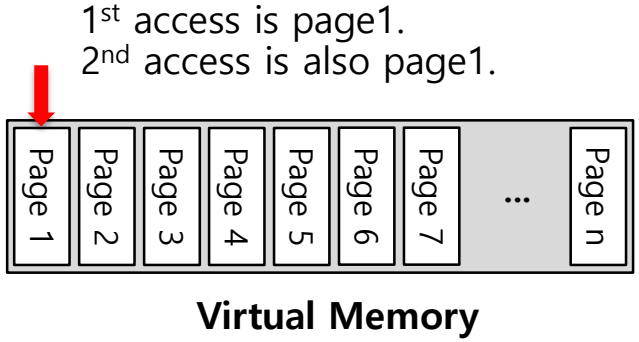

```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum += a[i];
3:      }
```

The TLB improves performance due to spatial locality

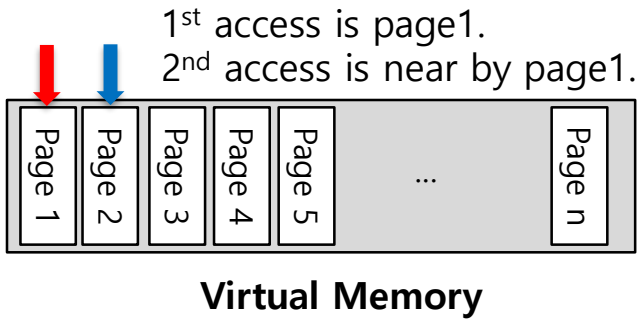
3 VPN misses {VPN 6,7,8} and 10 hits {a[0] .. A[9]}.
Thus TLB hit rate is = (10/13) = 77%.

Locality

- **Temporal (time) Locality:**
  - An instruction or data item that has been recently accessed will likely be re-accessed soon in the near 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 handles the TLB miss** entirely on CISC CPUs:
  - 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 the TLB** and **retry** instruction.
  - **hardware-managed TLB** ←

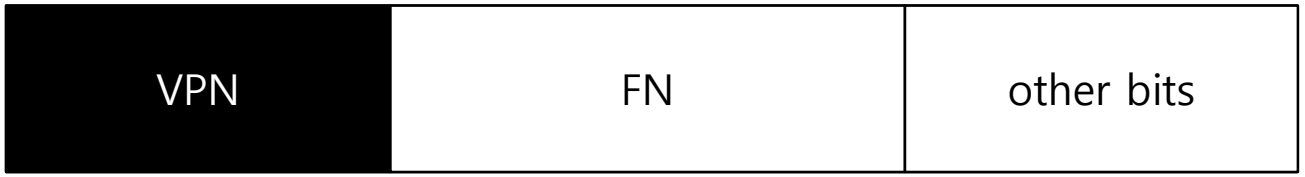
- **RISC CPUs** have what is known as a **software-managed TLB:**
  - On a TLB miss, the hardware raises exception (trap handler).
    - **Trap handler is code** within the OS that is written with the express purpose of **handling TLB miss**.

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)   // need to populate TLB entry from the PT
```

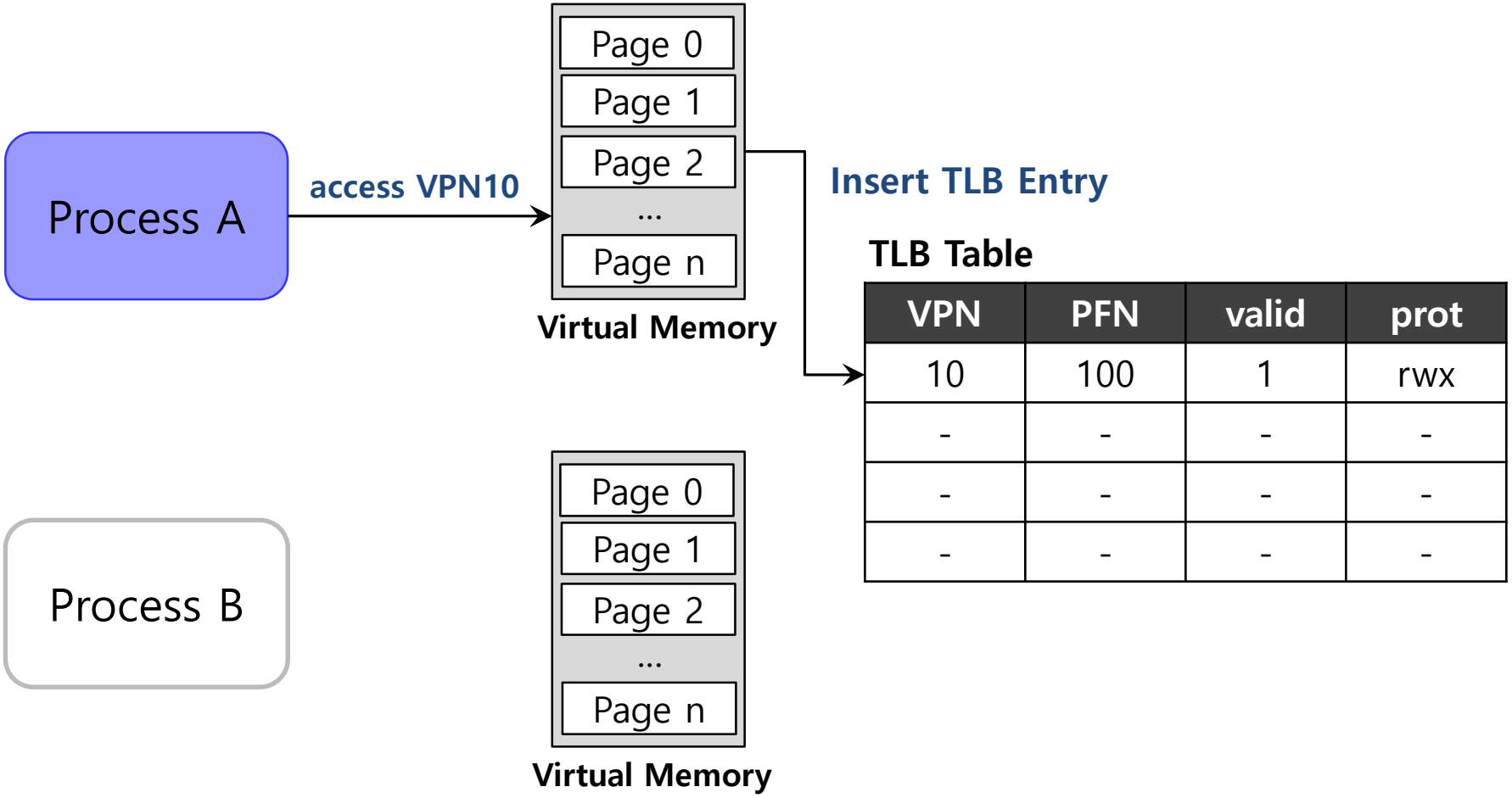
TLB entry

- **TLB is managed by Full Associative** access 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

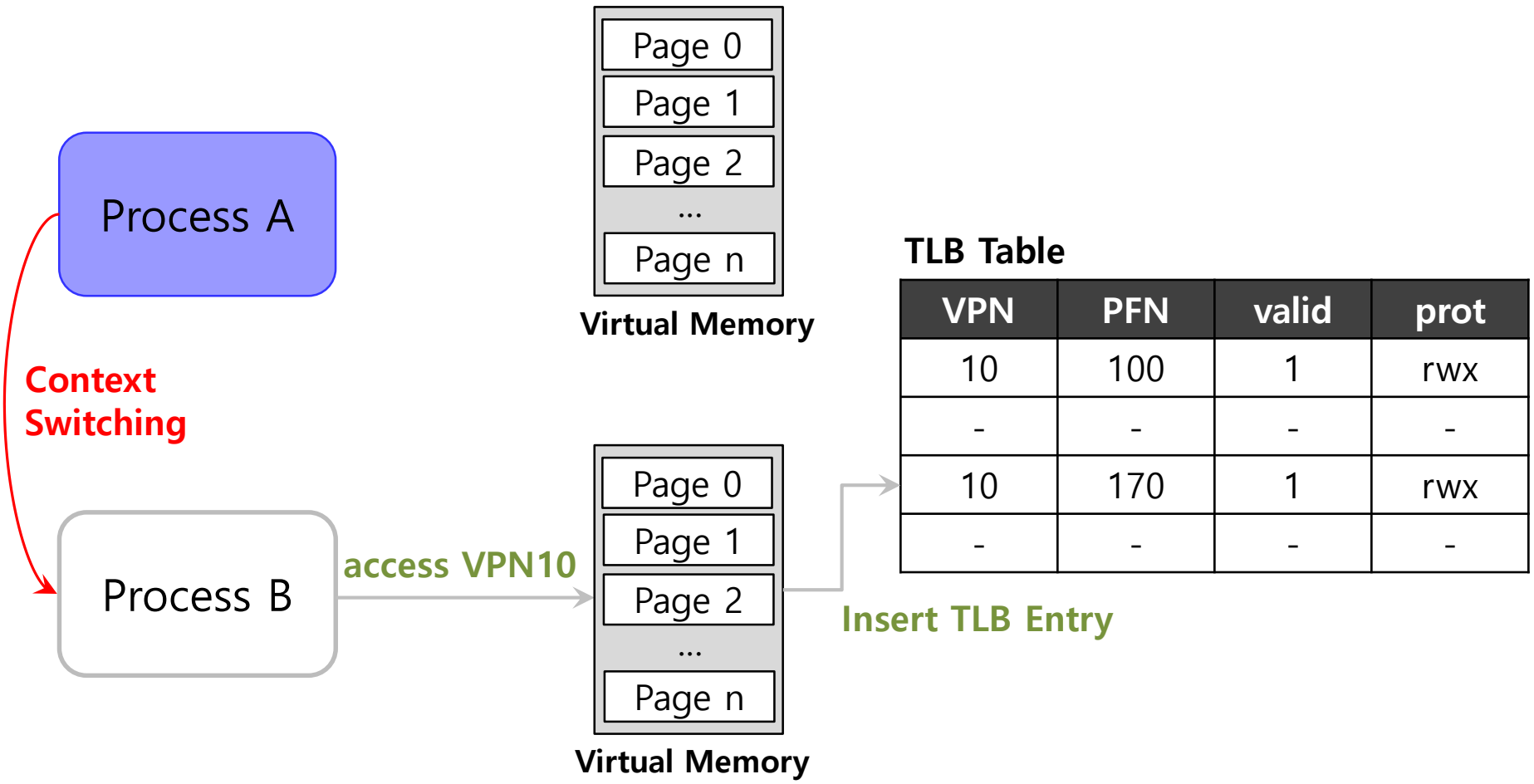


Typical TLB entry look like this

TLB Issue: Context Switching

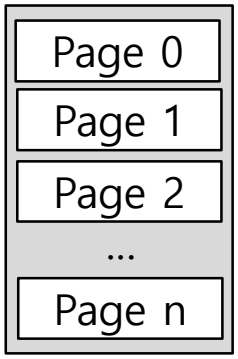


TLB Issue: Context Switching



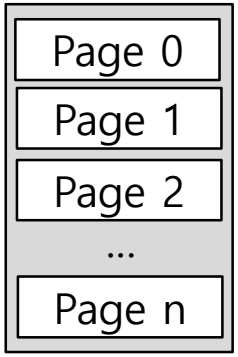
TLB Issue: Context Switching

Process A



Virtual Memory

Process B



Virtual Memory

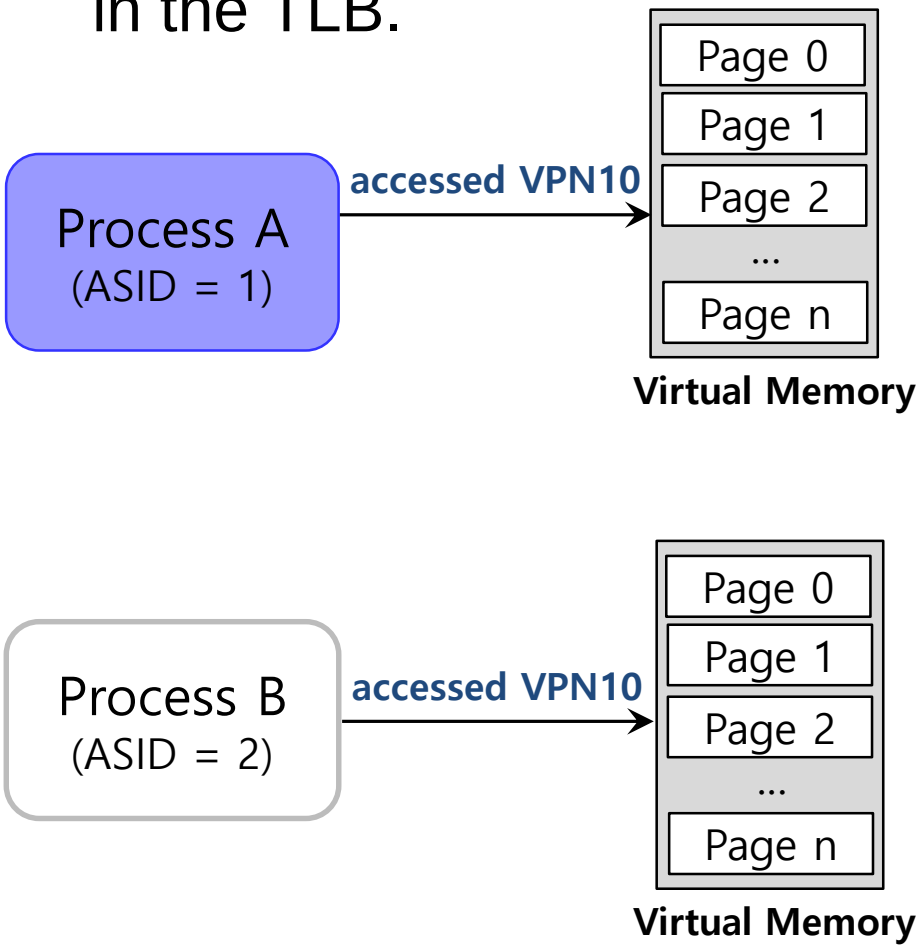
TLB Table

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 / PID**) field in the TLB.



pid

TLB Table

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

Another Case

- **Two processes share a page:**
  - To share memory between 2 processes:

System V → id = shmget(IPC_PRIVATE, ...);
char *shmat(id, …..);

OR

fd = shm_open(name, ...) ← Posix
Ptr = mmap(addres, ..., fd, ...); ← Linux / Unix
int munmap(addr, length);

  - 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.

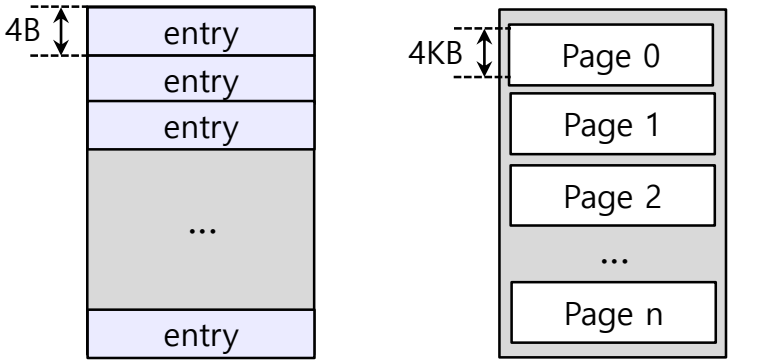
TLB Replacement Policy

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

Advanced Paging: Smaller Tables

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.

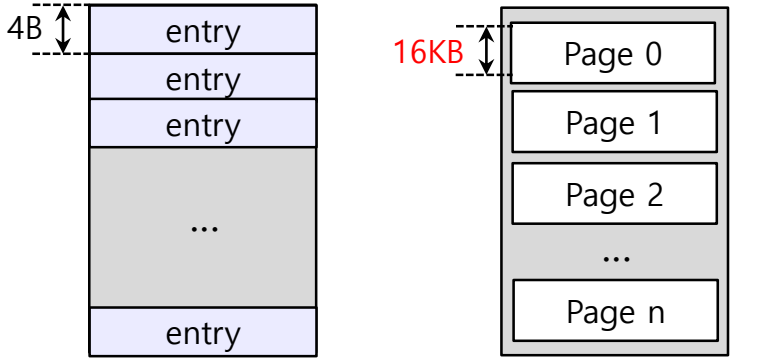


Page table size = $\frac{2^{32}}{2^{12}} * 4Byte = 4MByte$ ⟸ For one process

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

Paging: Smaller Tables

- 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.

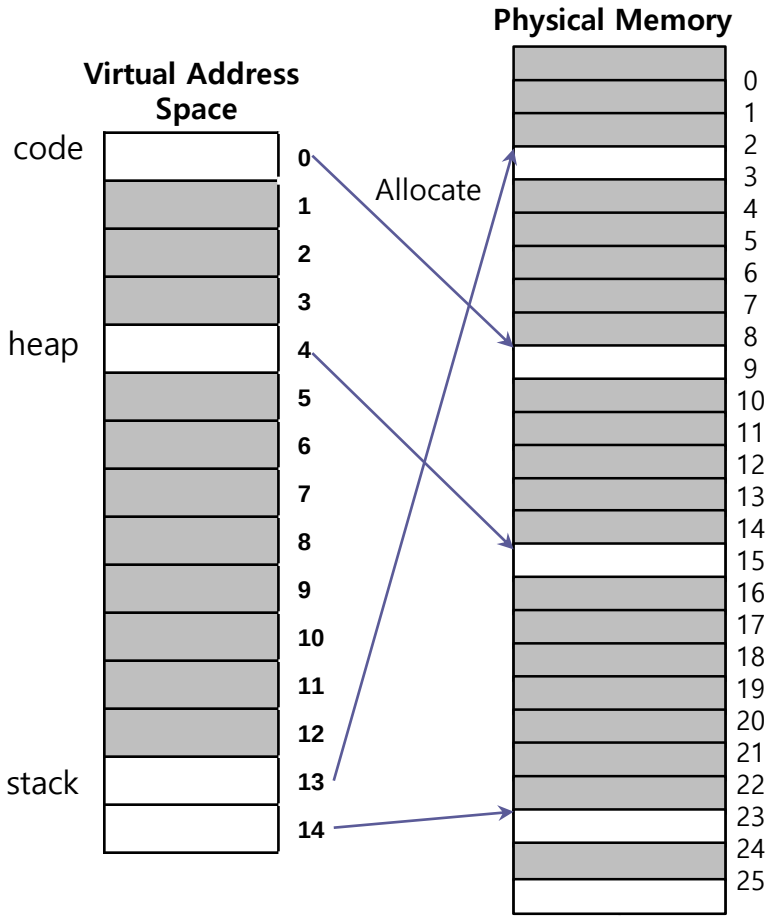


$\frac{2^{32}}{2^{14}} * 4 = 1MB$ per page table

But Big pages lead to internal memory fragmentation.

Problem

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



A 16KB Address Space with 1KB Pages

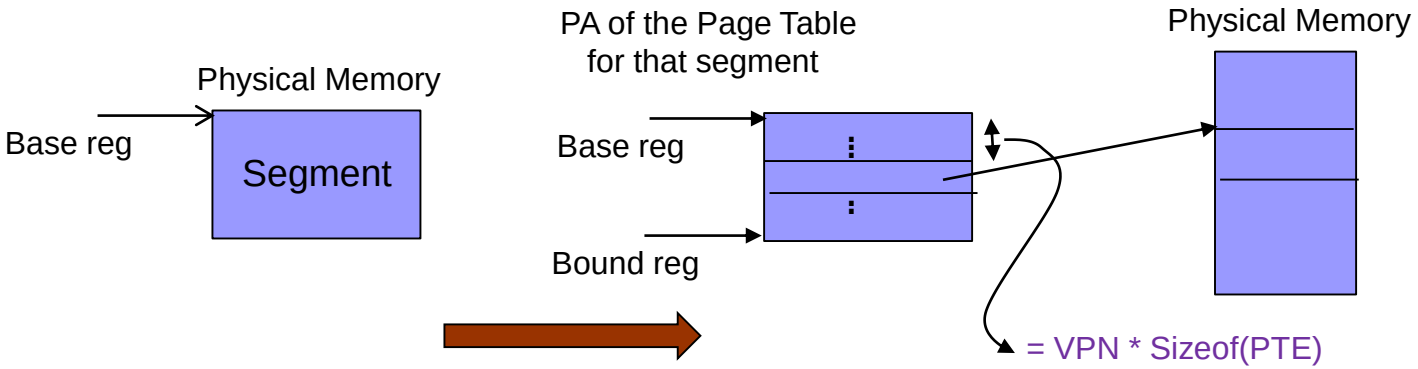
PFN	valid	prot	present	dirty
9	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
...	...	...	...	...
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

A Page Table For 16KB Address Space

Lot of invalid/unused (0) entries ☻

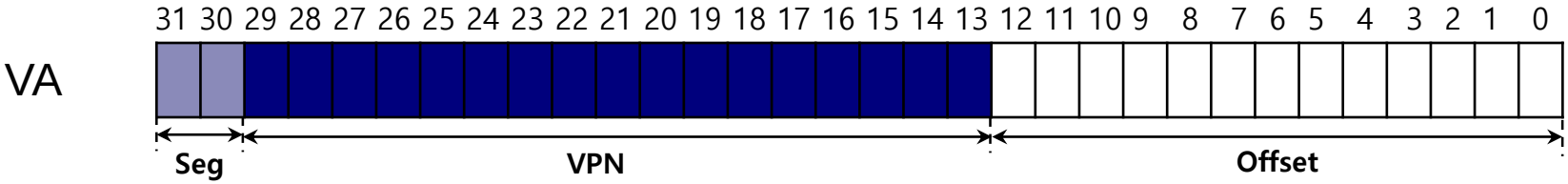
Hybrid Approach: Paging and Segments

- **In order to reduce the memory overhead** of a segment:
  - Using **base** register not to point to the segment itself but rather to hold the physical address of the page table of that segment. Now some pages in the segment might not be allocated and no memory fragmentation
  - The **bounds** register is used to indicate the end of the page table.



Simple Example of Hybrid Approach

- Each process has three page tables (code, heap, stack) 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.

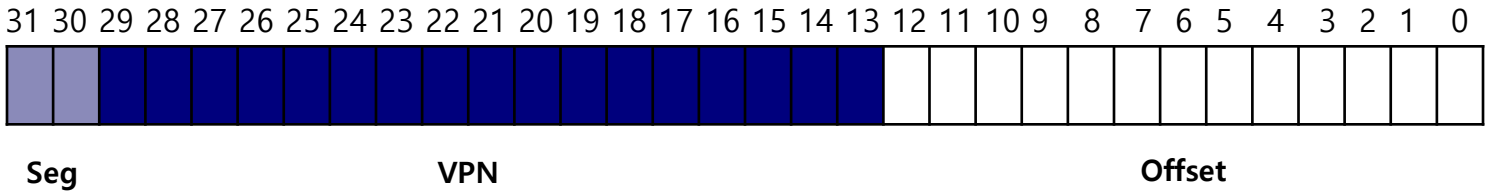


32-bit Virtual address space with 4KB pages

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

TLB miss on Hybrid Approach

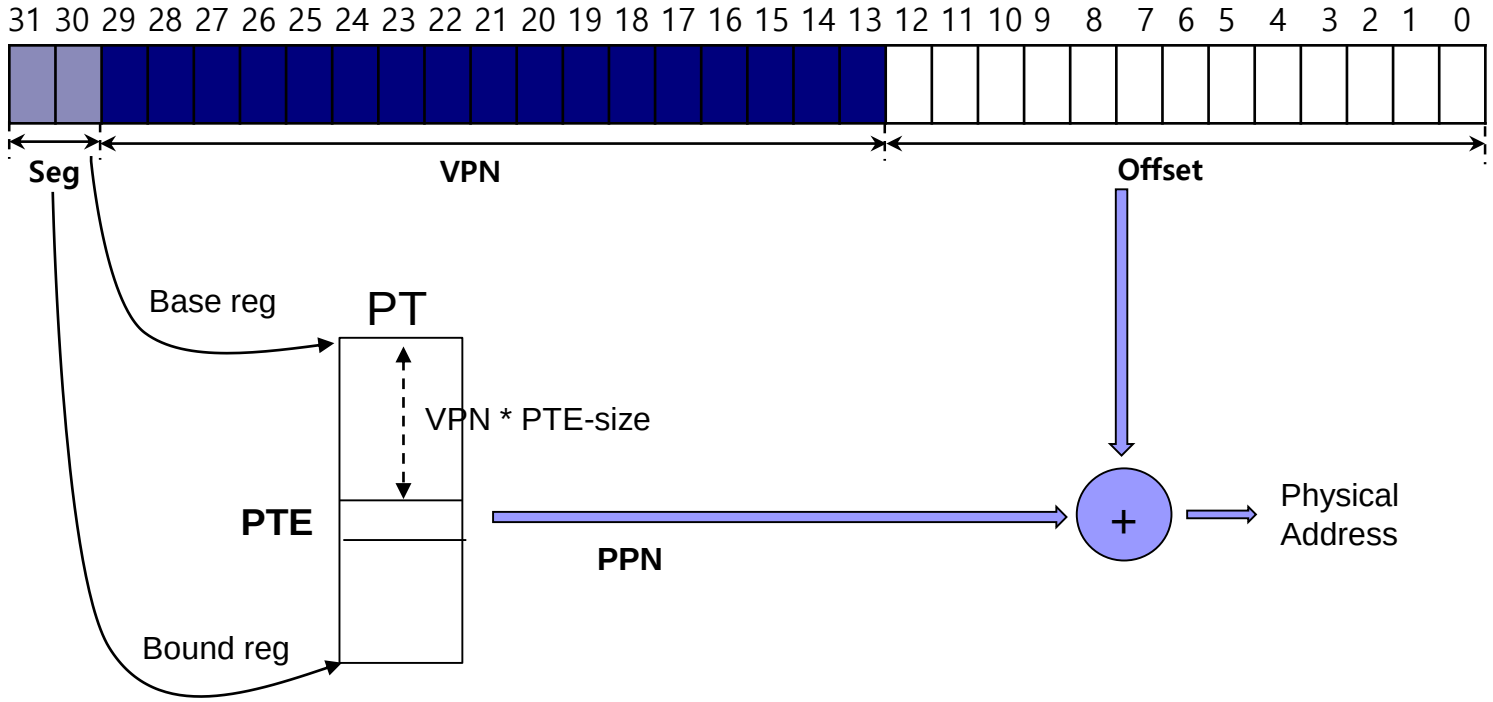
- The hardware gets to **physical address** from **page table**:
  - The hardware uses the segment bits (SN) to determine which base and bounds pair to use for the PT.



  - The hardware then takes the physical base 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

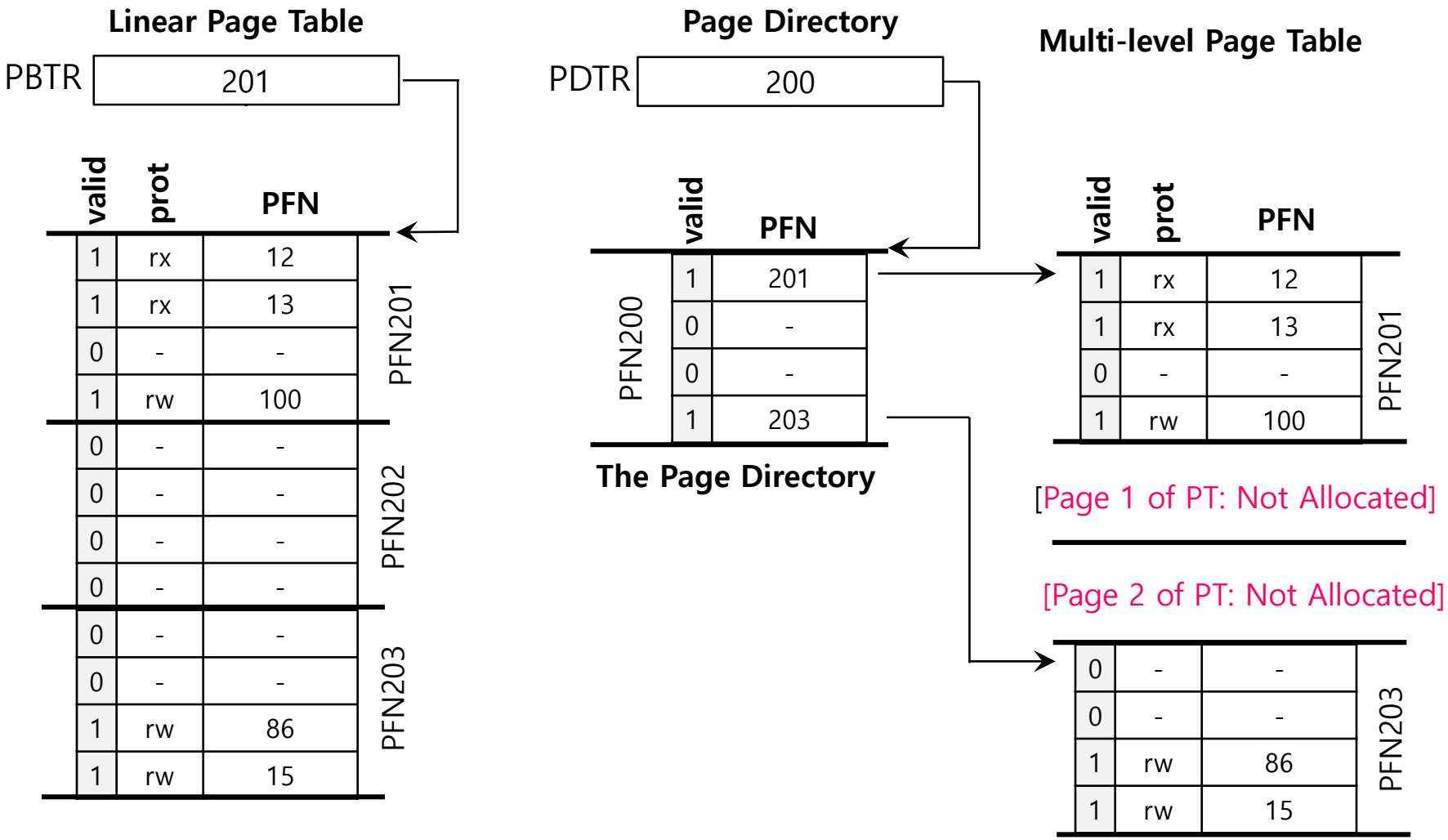


- **Hybrid Approach is not without problems:**
  - If we have a large but sparsely-used heap, we can still end up with a lot of page table entries (PTEs) wasted.

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 (additional level of indirection)**.

Multi-level Page Tables: Page directory



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

Multi-level Page Tables: Page Directory Entries

- The **page directory** contains one entry (PDE) per page of the page table; Page table is a set of non-contiguous pages where each page has set of PTEs
  - **Page Directory** consists of a number of page directory entries (PDE).

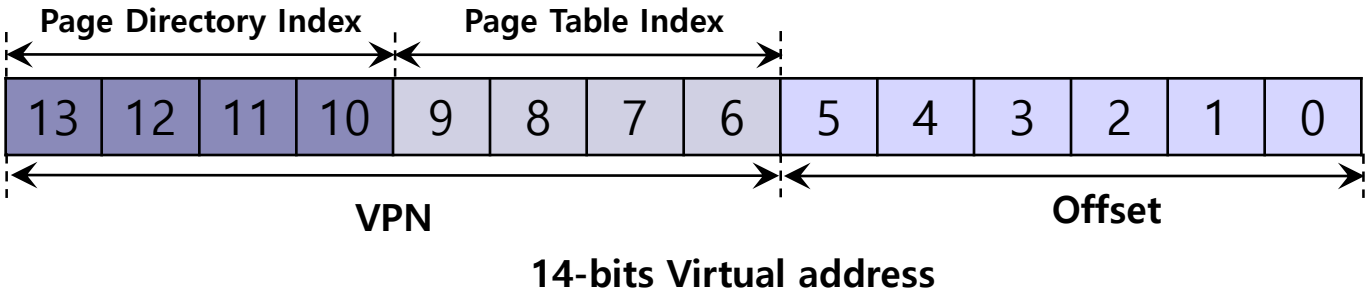
- **PDE has a valid bit** and page frame number (PFN) that points to a page of the Page Table.

Multi-level Page Tables: Advantage & Disadvantage

- **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.**

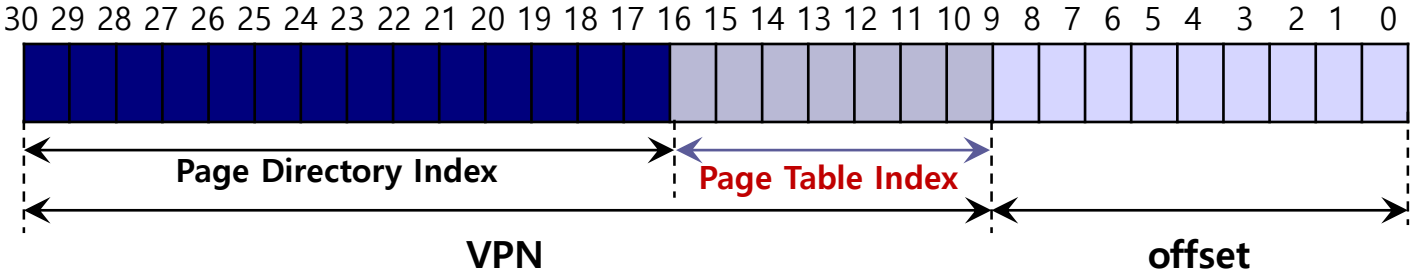
A Detailed Multi-Level Example: Page Table Index

- **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 : Page Table Index

- **In some cases**, a deeper tree is possible.
- **PD Index** points to a page in the page table. The Page Table Index points to the PTE within that page in the page table

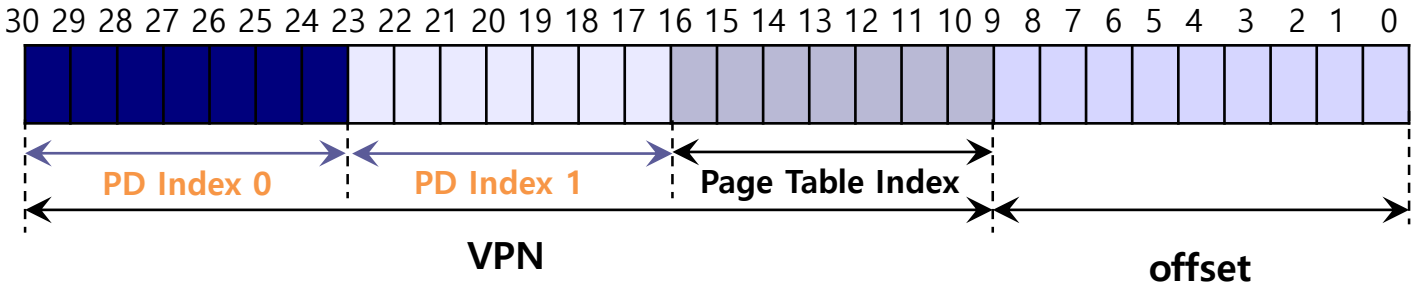


Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit
Page entry per page	128 PTEs

→ $\log_2 128 = 7$-bits

More than Two Levels: Page Directory

- **If our page directory** has 2^{14} entries, each entry spans not one page of PTEs but rather 128 pages ⟸ 7-bits
- **Allocating the PD** (16K entries array even though you might not be using many entries) is a problem. 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.



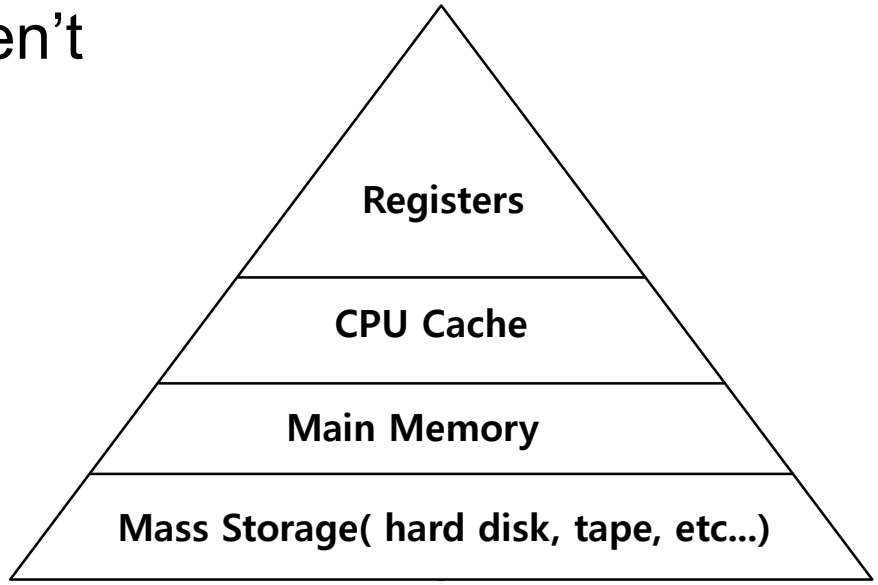
Inverted Page Tables

- **Keeping a single page table** for the whole system that has an entry for each physical page of the system.
- **The entry tells us which process** is using this page, and which virtual page of that process maps to this physical page.
- **It is good model to control memory usage** for translation independent of how many processes are running

Swapping Mechanisms

Beyond Physical Memory: Mechanisms

- **Require an additional level in the memory hierarchy:**
  - **OS needs** 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

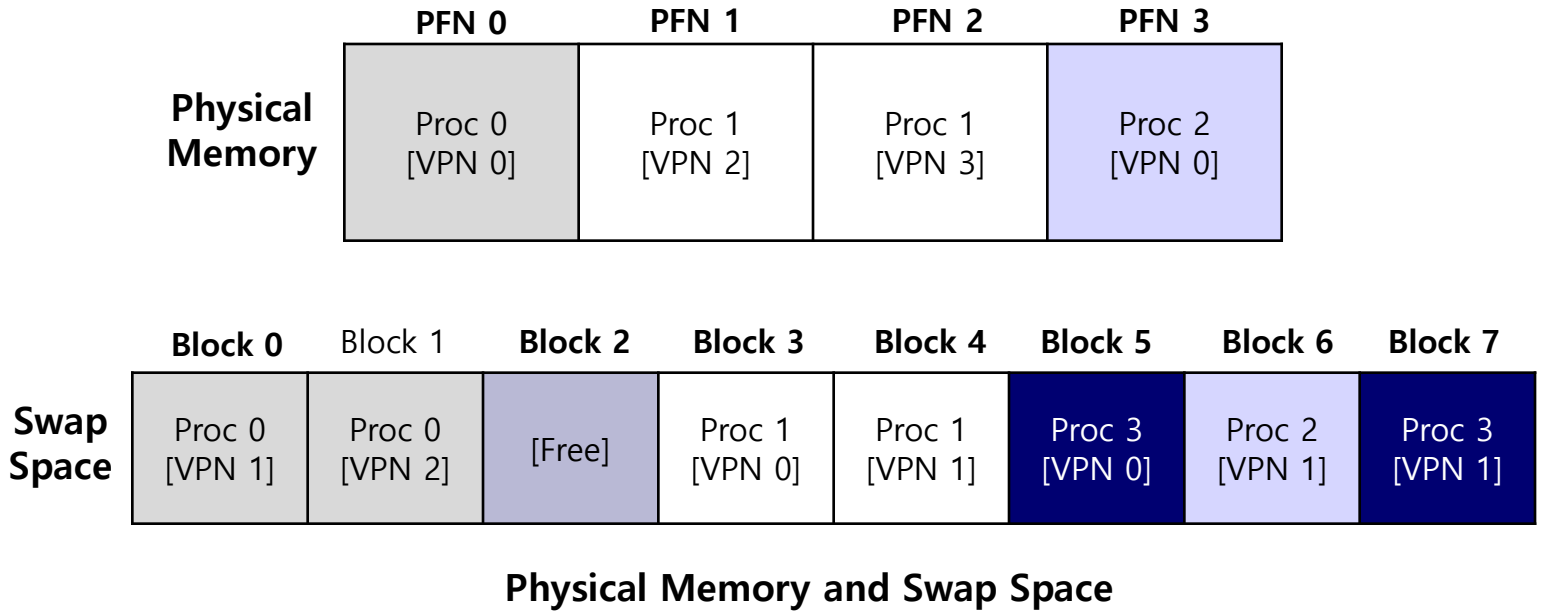
Single large address for a process

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

- **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 processes

Swap Space

- **Reserve some space on the disk** for moving memory pages back and forth.
- **OS needs to refer 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.

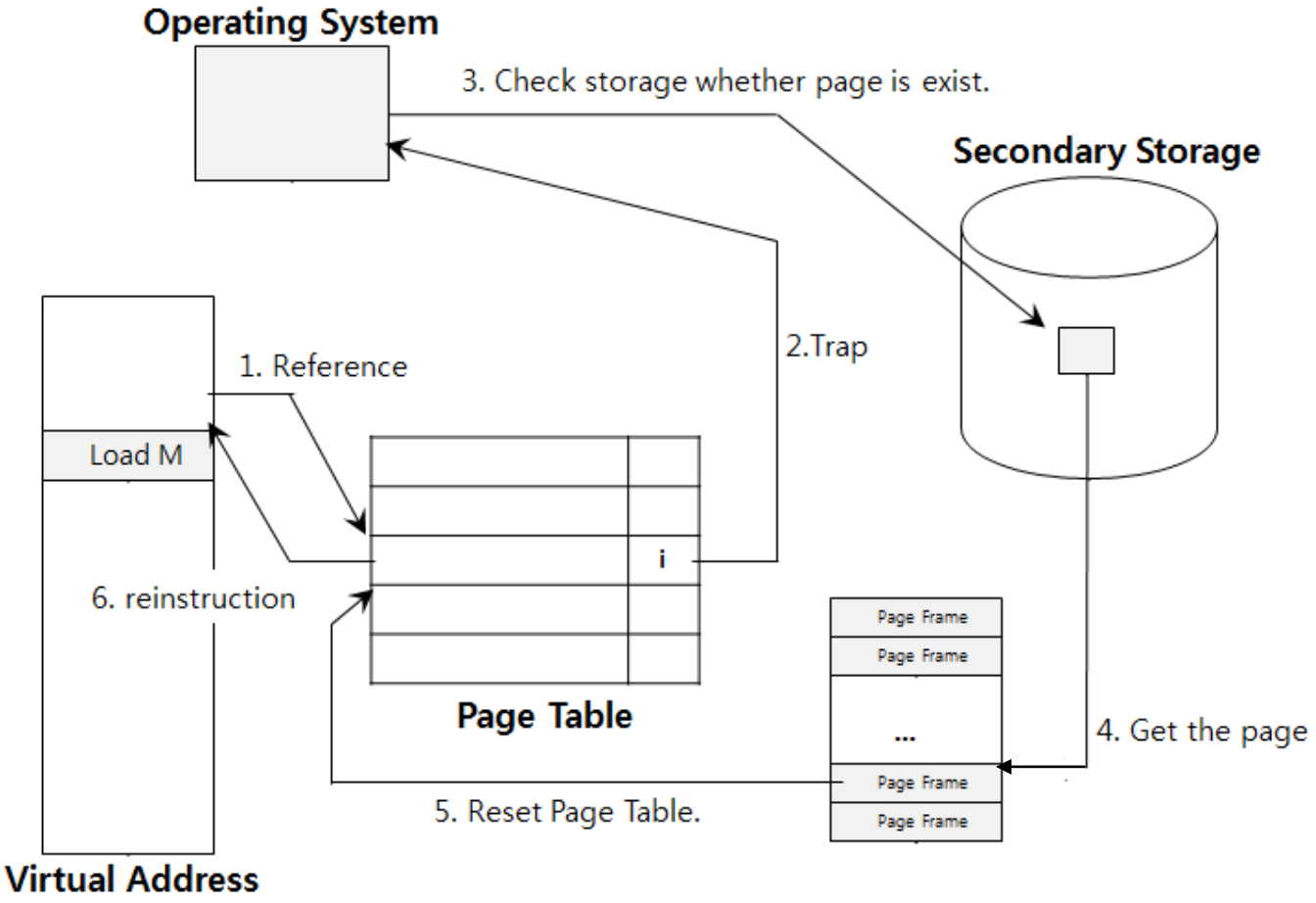
What If Memory Is Full ?

- **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.

- **Page Fault:** Accessing page that is not in physical memory:
  - **If a page is not present** and has been swapped earlier to disk, the OS needs to swap the page back 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.

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 reasons for the OS to keep a small portion of memory free (low watermark) more proactively.
- **Swap Daemon, Page Daemon:**
  - There are fewer than LW (low watermark) pages available, a background thread that is responsible for freeing memory runs.
  - The thread evicts pages until there are HW (high watermark) pages available.

Swapping Policies

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 page replacement policy of the OS.
- **Goal of the replacement policy** is to minimize the number of cache misses
- **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
- **Example:** Assume the cache consists of 3 pages and the number of pages being referenced are 4 pages!

Tracing the Optimal Policy

Reference Row

0 1 2 0 1 3 0 3 1 2 1

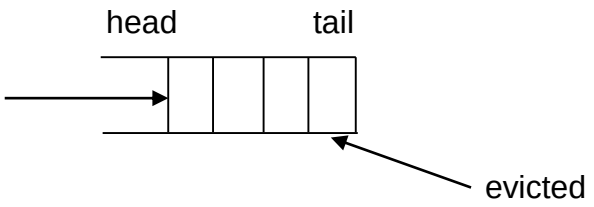
Assume we know future references, and we will pick page to evict that is the furthest in the future

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

Hit rate is $\frac{Hits}{Hits+Misses} = \frac{6}{11} = \mathbf{54.6\%}$

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 at the tail of the queue (the "**First-in**" pages) is evicted.
  - **It is simple to implement**, but can't incorporate the importance of blocks.
- **Same Example….**

Observation: cache hit decreases (refer to example next page); pages are evicted even though they have been accessed multiple times!

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

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

Hit rate is $\frac{Hits}{Hits+Misses} = \frac{4}{11} = 36.4\%$

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.
  - In this scenario, random was close to Optimal (i.e., 6 hits)

Hit rate is $\frac{Hits}{Hits+Misses} = \frac{5}{11}$

$= 45.5\%$

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

Using History

- Lean from the past and use **history to predict the future**
  - Two type of historical information.

Historical Information	Meaning	Algorithms
recently	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 replcecd as it clearly has some value	LFU

- **LRU:**
  6-hits –
  almost perfect

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

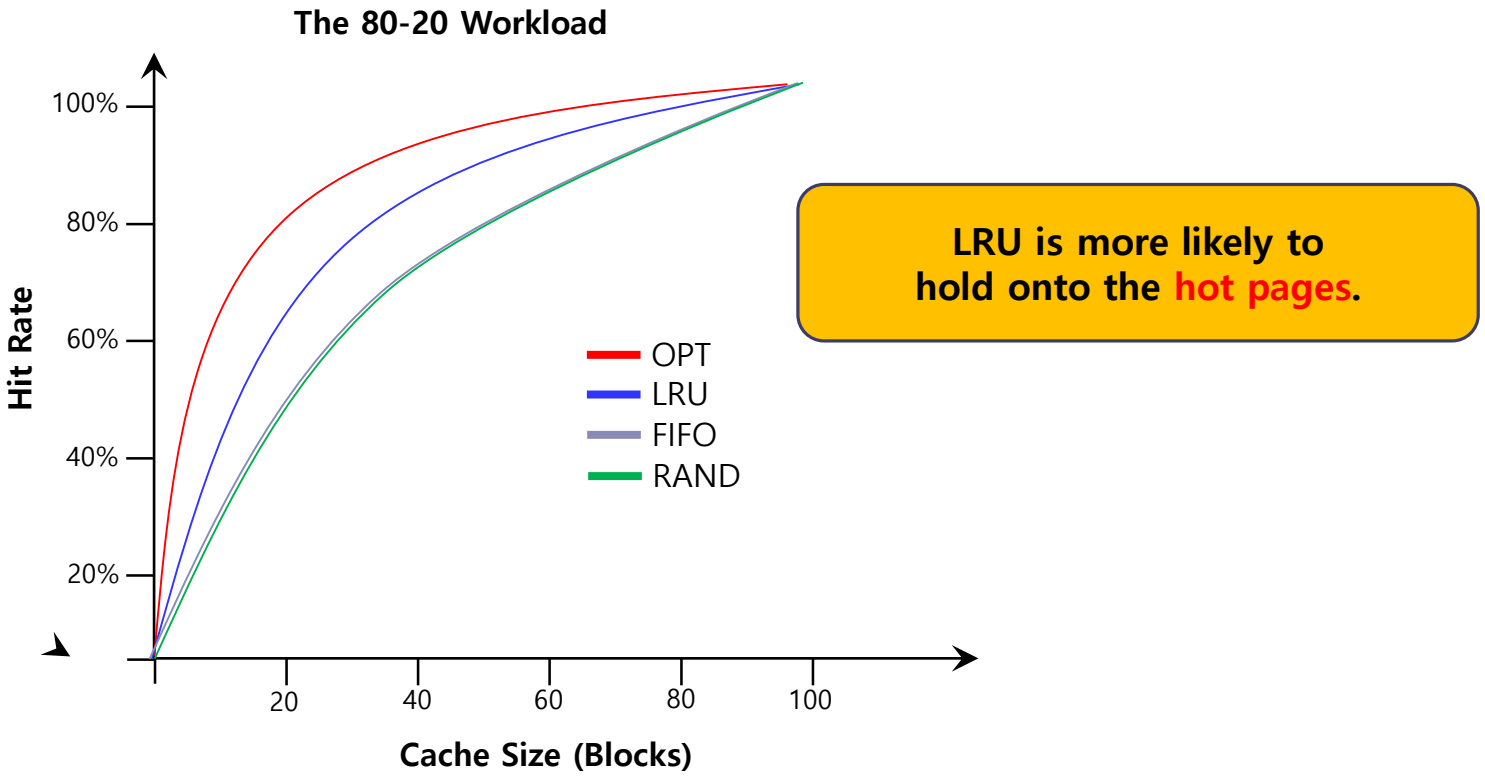
Page-2 is Least referenced in recent past

Hit rate is $\frac{Hits}{Hits+Misses} = \frac{6}{11}$

$= 54.6\%$

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.

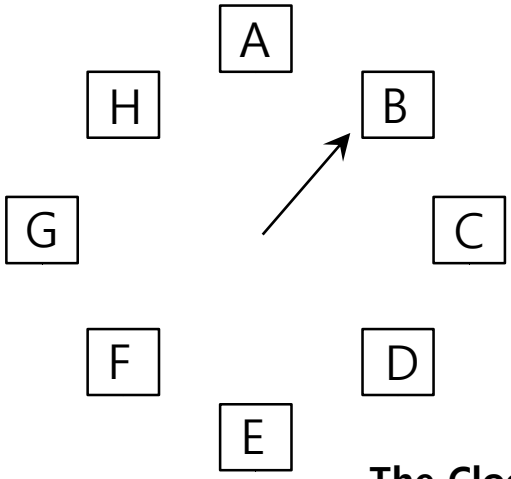
Implementing Historical Algorithms

- **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 or Access 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 are arranged in a circular list.
- **A clock hand** points to some particular page to begin with.
- **The algorithm** continues until it finds a use bit that is set to 0 (evict).
- **If the use-bit** is 1 then OS reset it to 0, and move to test next page.



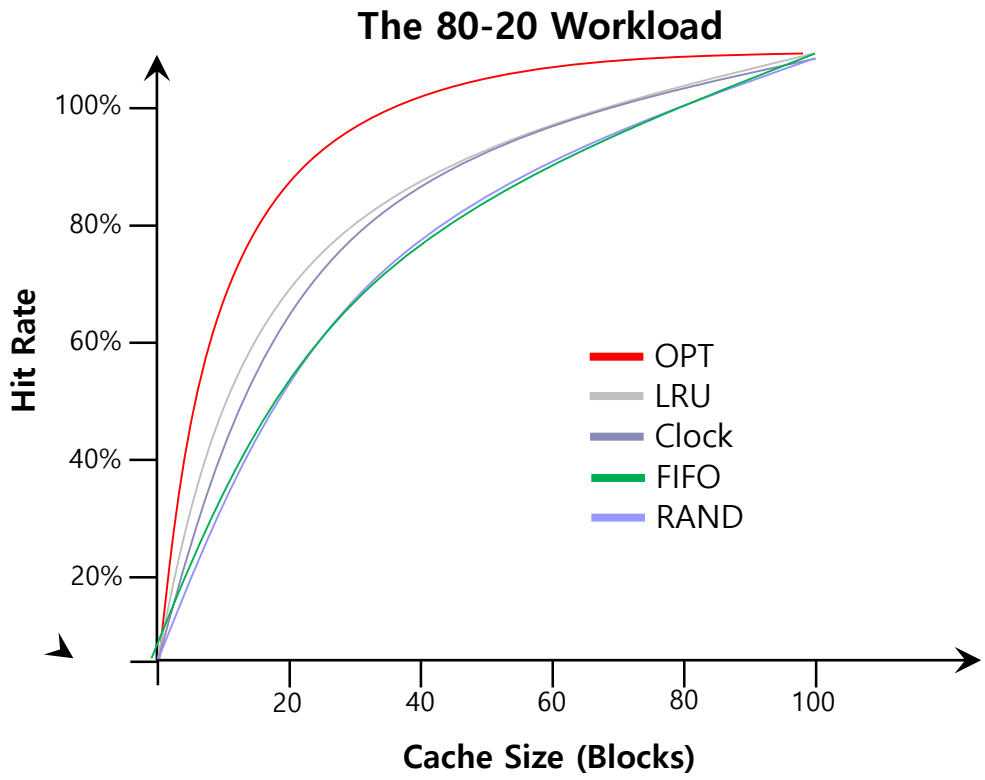
Use bit	Meaning
0	Evict the page
1	Clear **Use bit** and advance hand

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 than an 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 before you can evict it.
  - **Page has not been modified**, the eviction is free (immediately reuse).

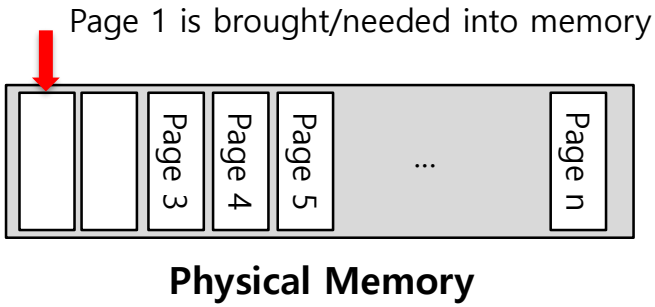
Page Selection Policy

- **The OS** has to decide when to bring a page into memory.

- **Presents the OS** with some different options:
  - **Prefetching** – when read
  - **Clustering** – when write
  - **Thrashing** – memory is over subscribed

Prefetching ← Read

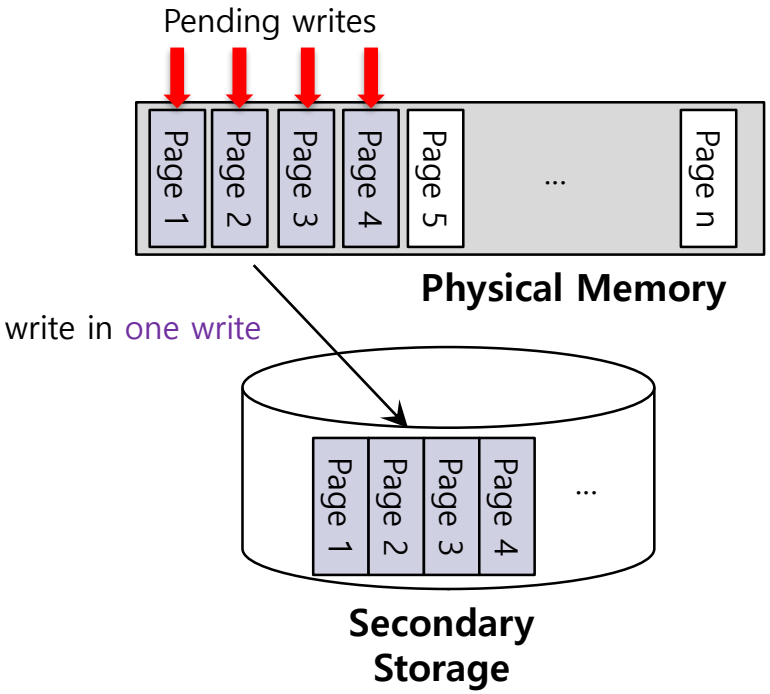
- **The OS guess** that a page is about to be used, and thus bring it in ahead of time.



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

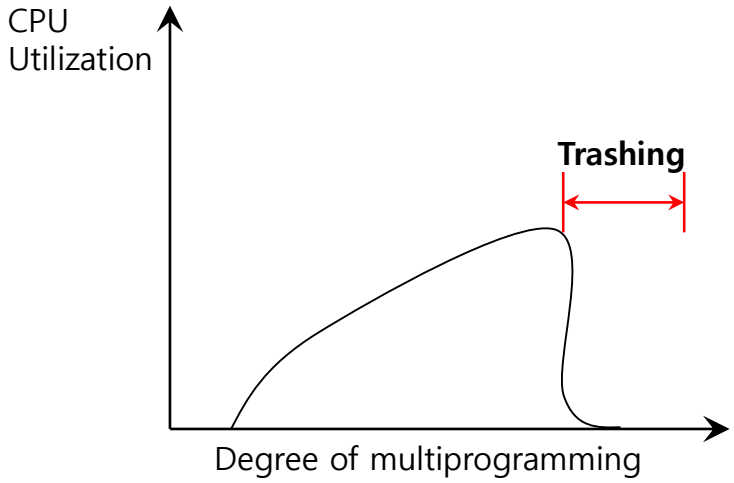
Clustering, Grouping ← Write

- **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 over-subscribed** 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.



END