Memory Virtualization

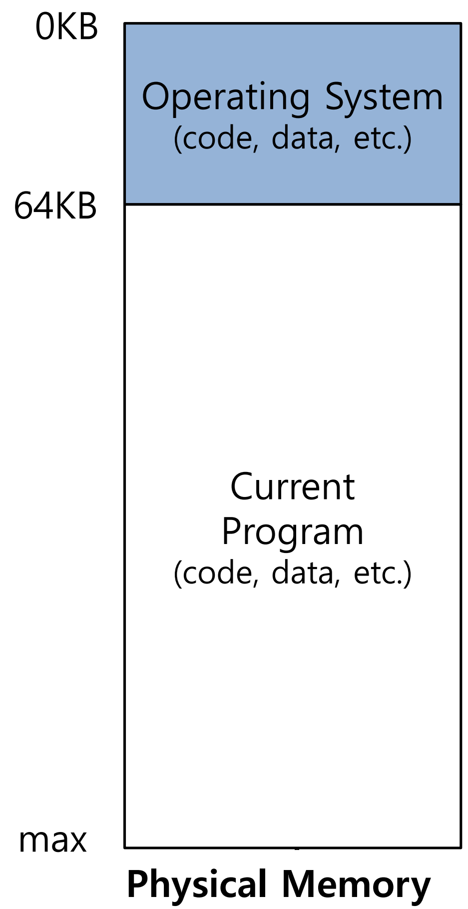
The Abstraction: Address Space

Memory Virtualization

- What is **memory virtualization**?
  - OS virtualizes its physical memory.
  - OS provides an illusion memory space per process.
  - It seems to be seen like each process uses the whole memory .
- Benefits
  - Transparency: ease of use in programming
  - Memory efficiency in terms of times and space
  - The guarantee of isolation for processes as well as OS
    - Protection from errant accesses of other processes

OS in The Early System

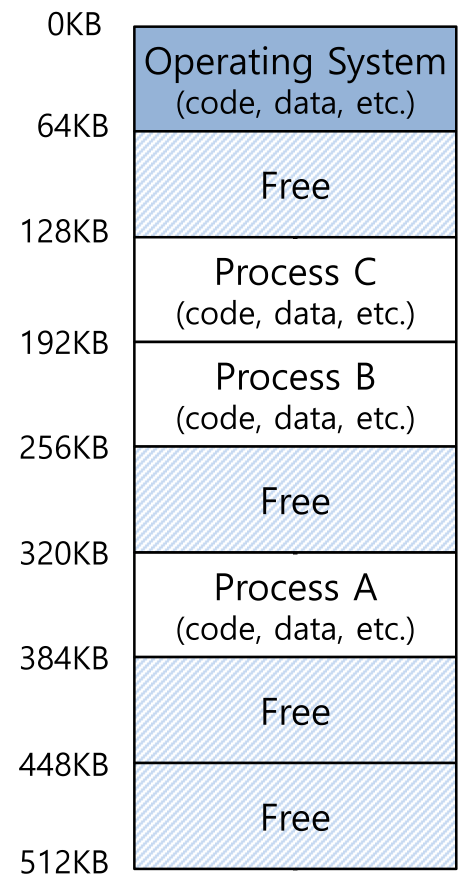
- Load only one process in memory.
  - Poor utilization and efficiency



Multiprogramming and Time Sharing

- **Load multiple processes** in memory.
  - Execute one for a short while.
  - Switch processes between them in memory.
  - Increase utilization and efficiency.

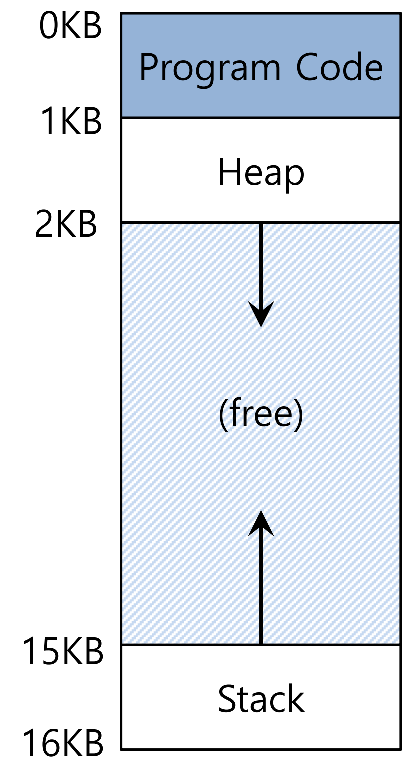
- Cause an important **protection issue**.
  - Errant memory accesses from other processes



Physical Memory

Address Space

- OS creates an **abstraction** of physical memory.
  - The address space contains all about a running process.
  - That is consist of program code, heap and stack
    - Code
      - Where instructions live
    - Heap
      - Dynamically allocate memory.
        - `malloc` in C language
        - `new` in object-oriented language
    - Stack
      - Store return addresses or values.
      - Contain local variables arguments to routines.



Address Space

Virtual Address

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

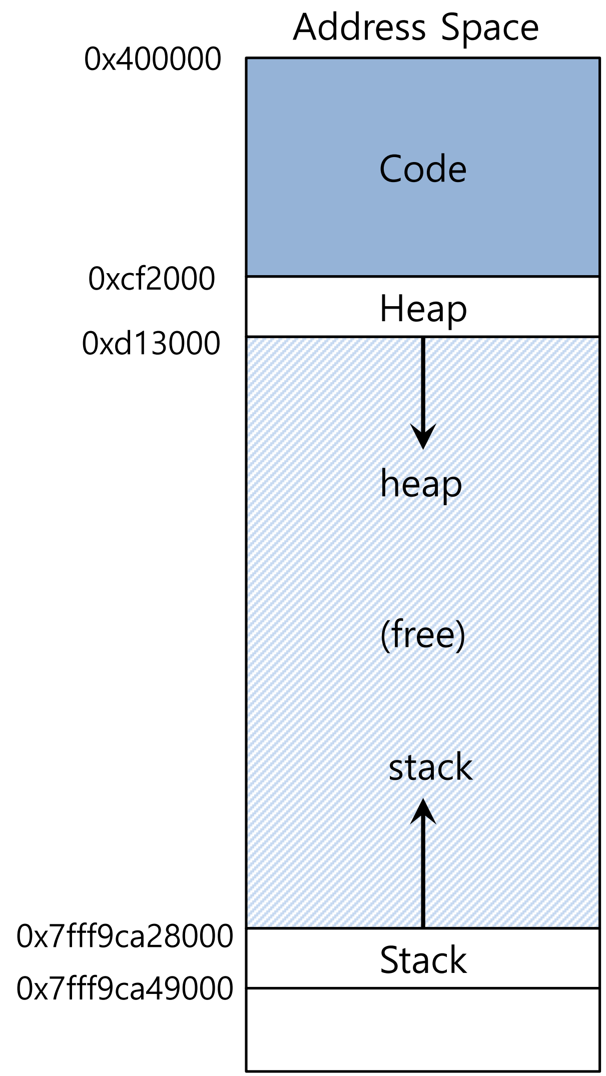
```
#include <stdio.h>
#include <stdlib.h>

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;
    printf("location of stack : %p\n", (void *) &x);

    return x;
}
```

A simple program that prints out virtual addresses

Virtual Address(Cont.)

- The output in 64-bit Linux machine

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



Memory API

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

Memory types

- Stack: compiler managed
  - Allocates and deallocates memory on stack
  - Function parameters, local variables
  - Short-lived
- Heap: user managed
  - User explicitly allocates or deallocates
  - Live beyond a function call

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

sizeof()

- Routines and macros are utilized for `size` in `malloc` instead of 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
```

sizeof() thinks we are simply asking how big a *pointer* to an integer is, not how much memory we have dynamically allocated.

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

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

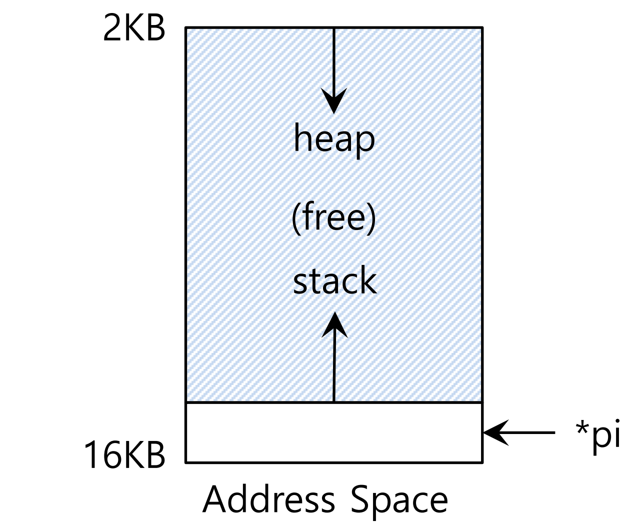
```
40
```

Memory API: free()

```
#include <stdlib.h>

void free(void* ptr)
```

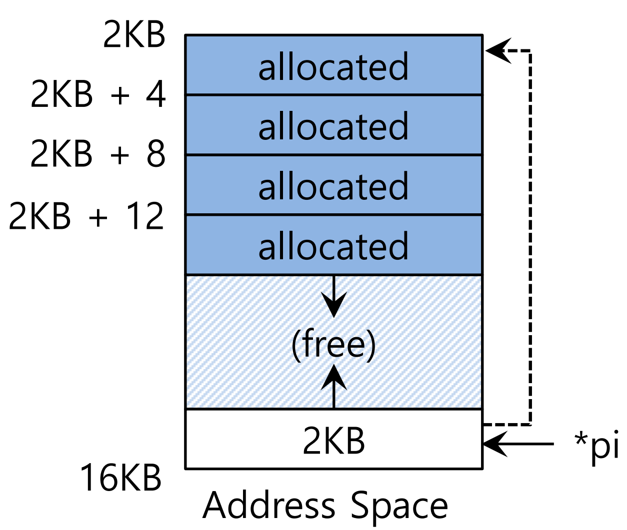
- Free a memory region allocated by a call to `malloc`.
  - Argument
    - `void *ptr` : a pointer to a memory block allocated with `malloc`
  - Return
    - none

Memory Allocating



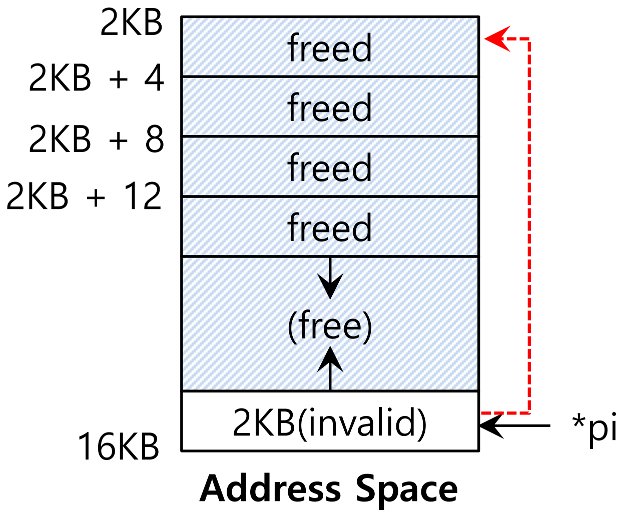
pointer

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

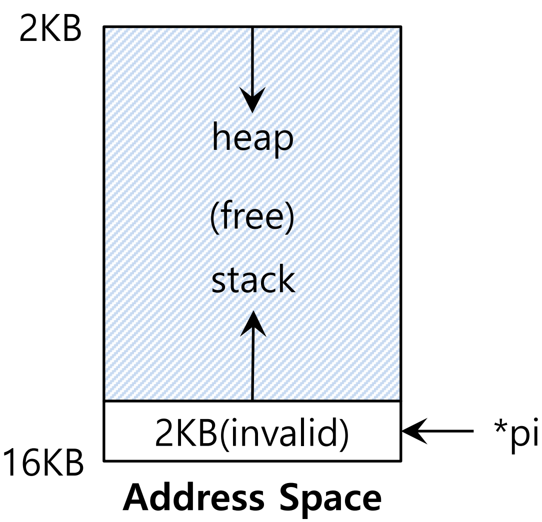



```
pi = (int *)malloc(sizeof(int)*4);
```

Memory Freeing




```
free(pi);
```

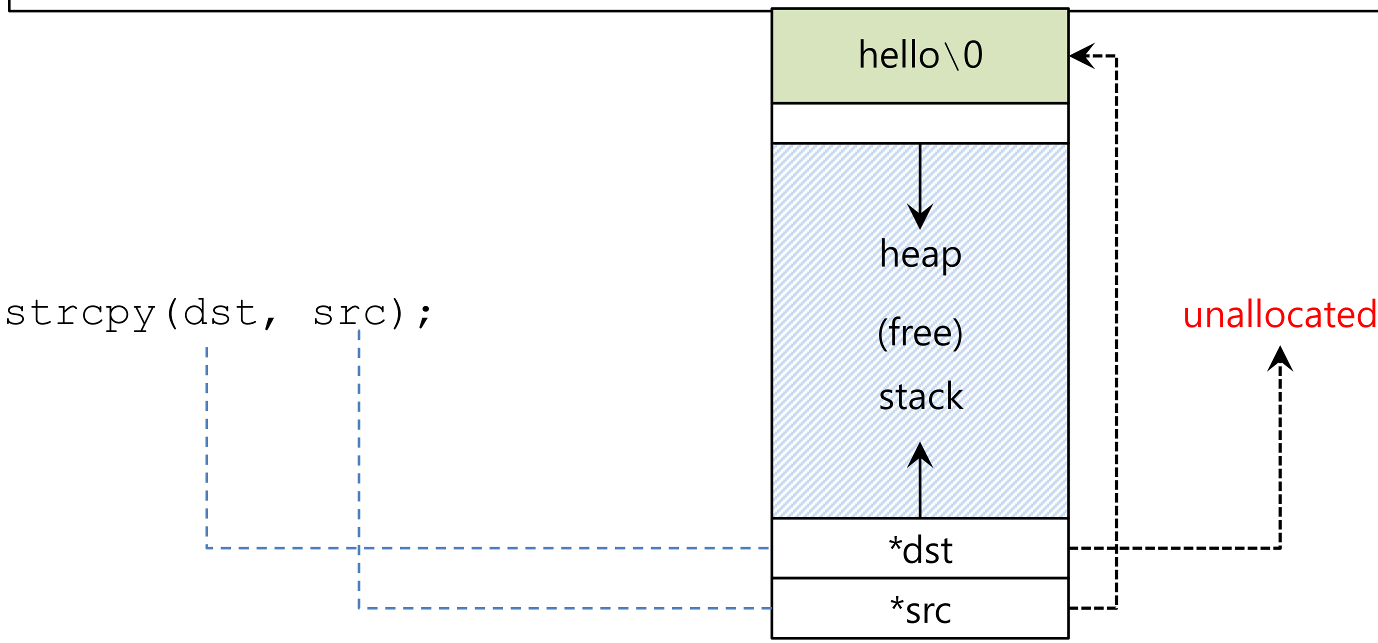


Forgetting To Allocate Memory

- Many routines expect memory to be allocated before you call them

Incorrect code

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

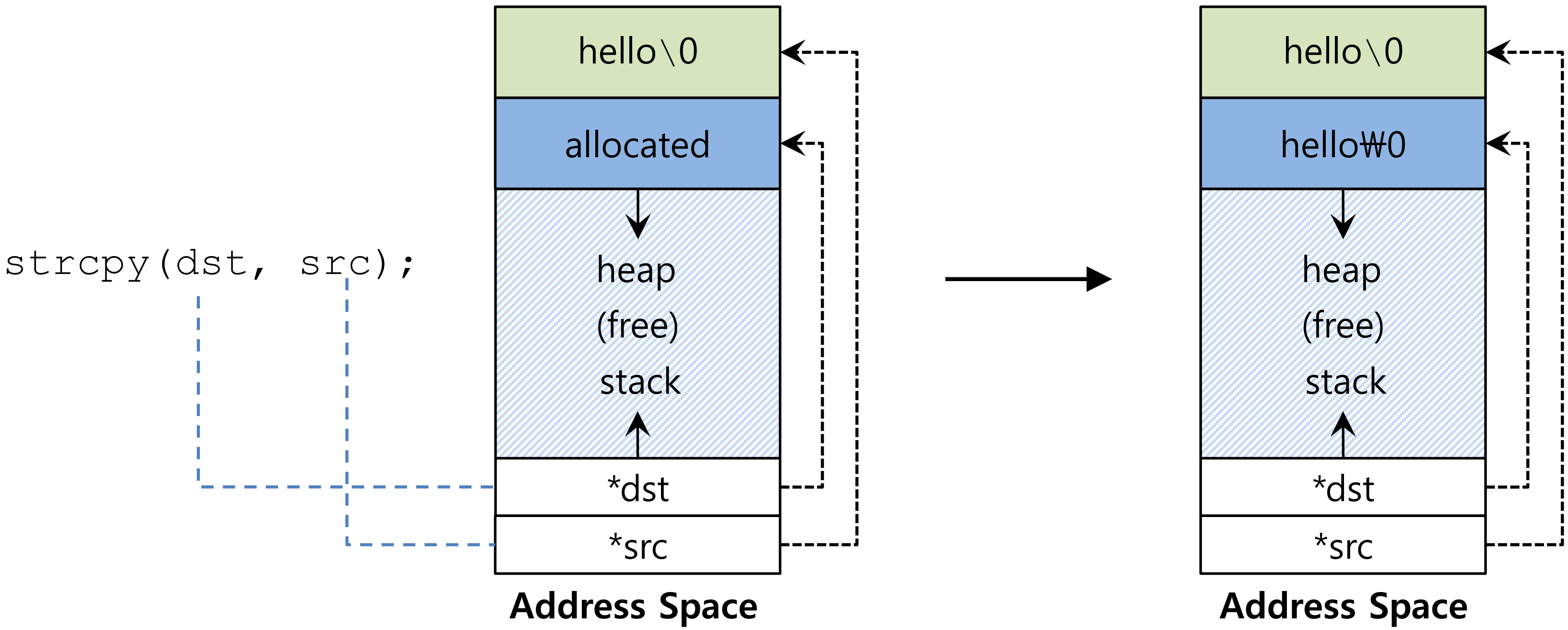


Address Space

Forgetting To Allocate Memory(Cont.)

- Correct code

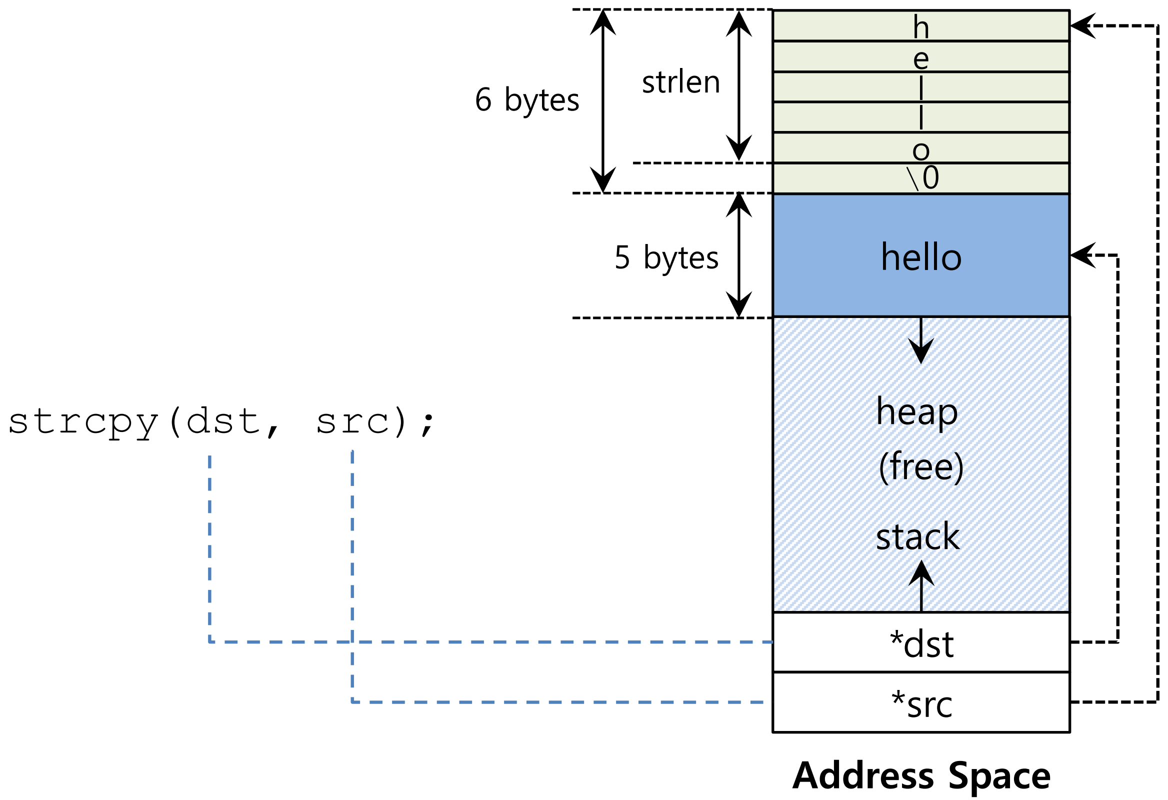
```
char *src = "hello";   //character string constant
char *dst =(char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);      //work properly
```



Not Allocating Enough Memory (Buffer overflow)

- Incorrect code, but work properly

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



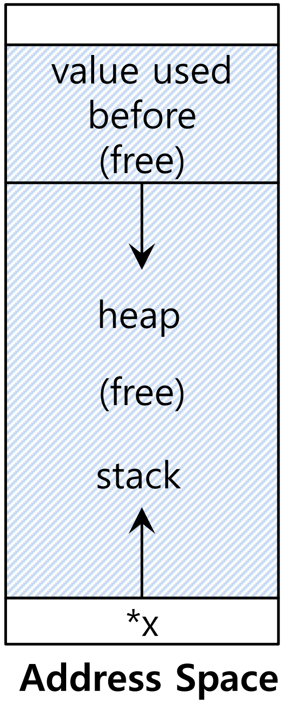
'\0' is omitted
Or
Overwrites other variable

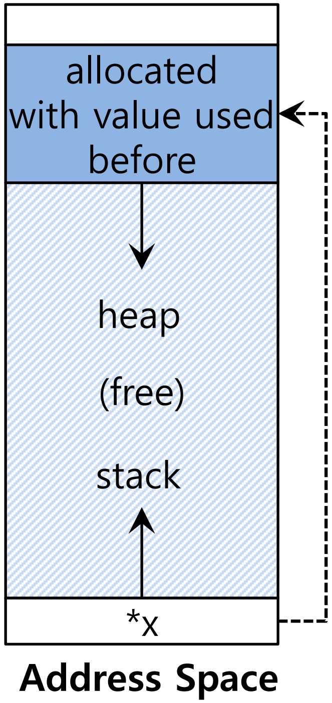
It may run correctly once but does not mean that program is correct.

Forgetting to Initialize

- Encounter an uninitialized read

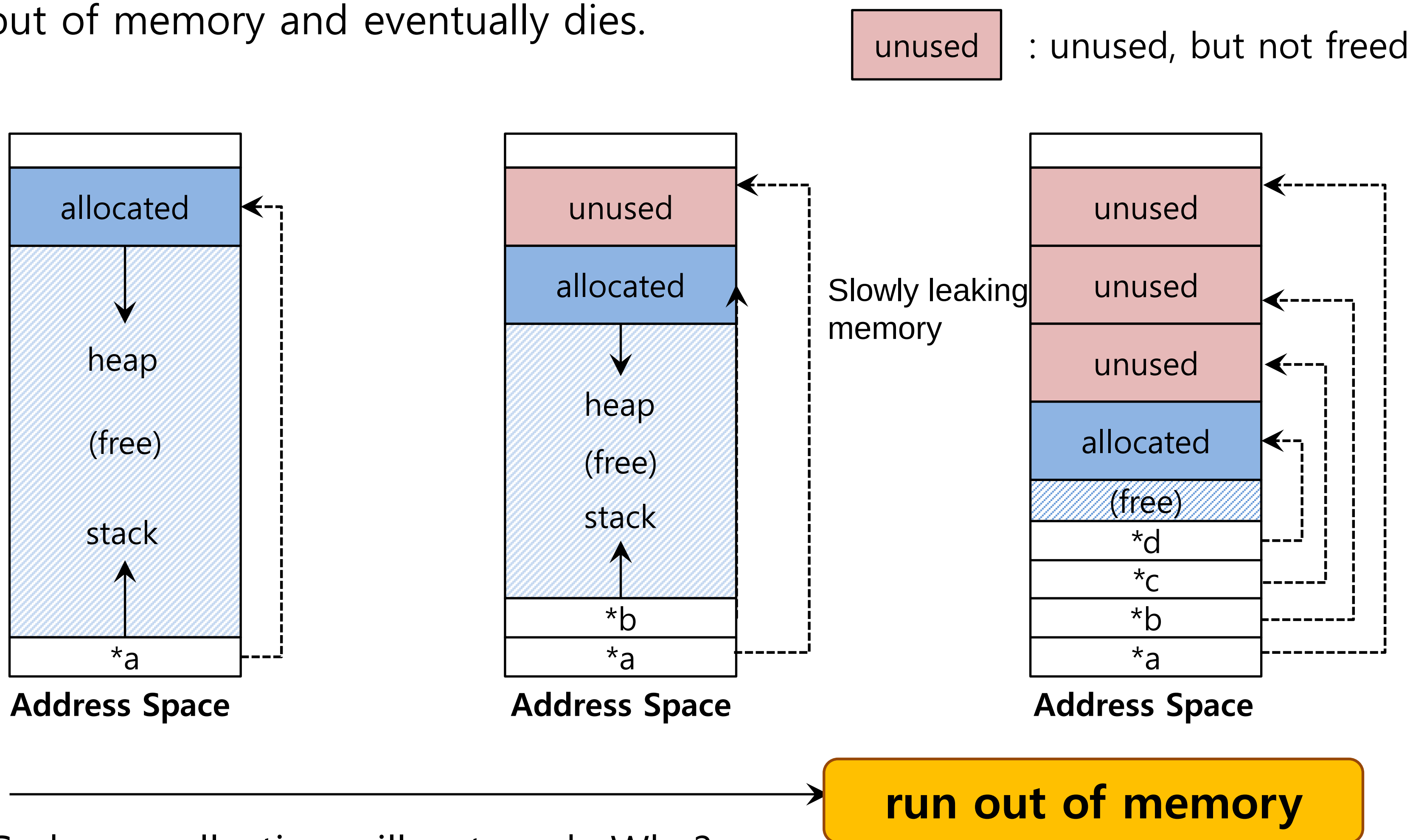
```
int *x = (int *)malloc(sizeof(int)); // allocated
printf("*x = %d\n", *x); // uninitialized memory access
```





Memory Leak

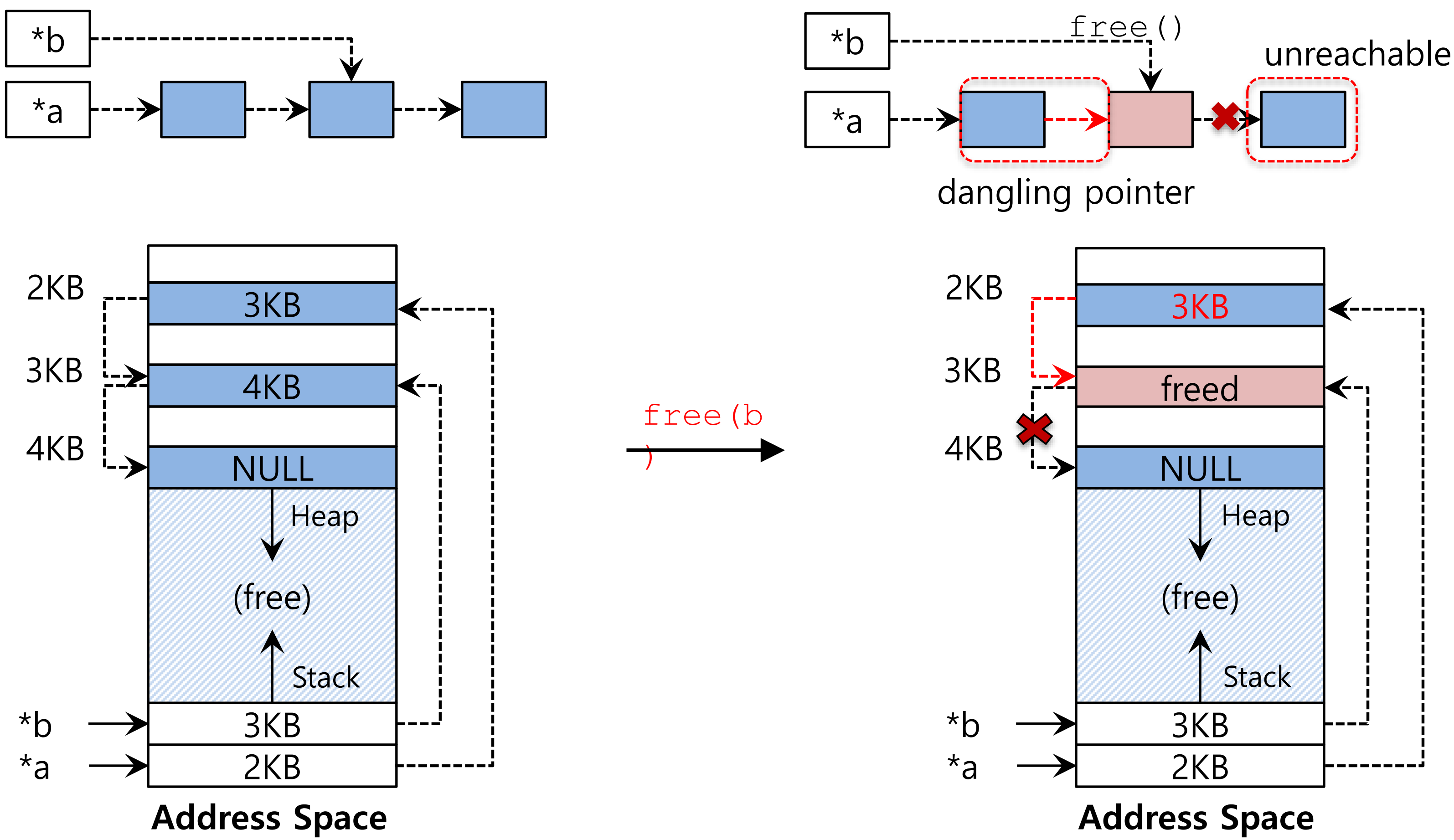
- In long running applications if unused memory is not freed, a program runs out of memory and eventually dies.



- Garbage collection will not work. Why?

Dangling Pointer

- Freeing memory before its use is finished
  - Dangling pointer: pointer to freed memory: assign NULL after deallocation



Other Memory APIs: calloc()

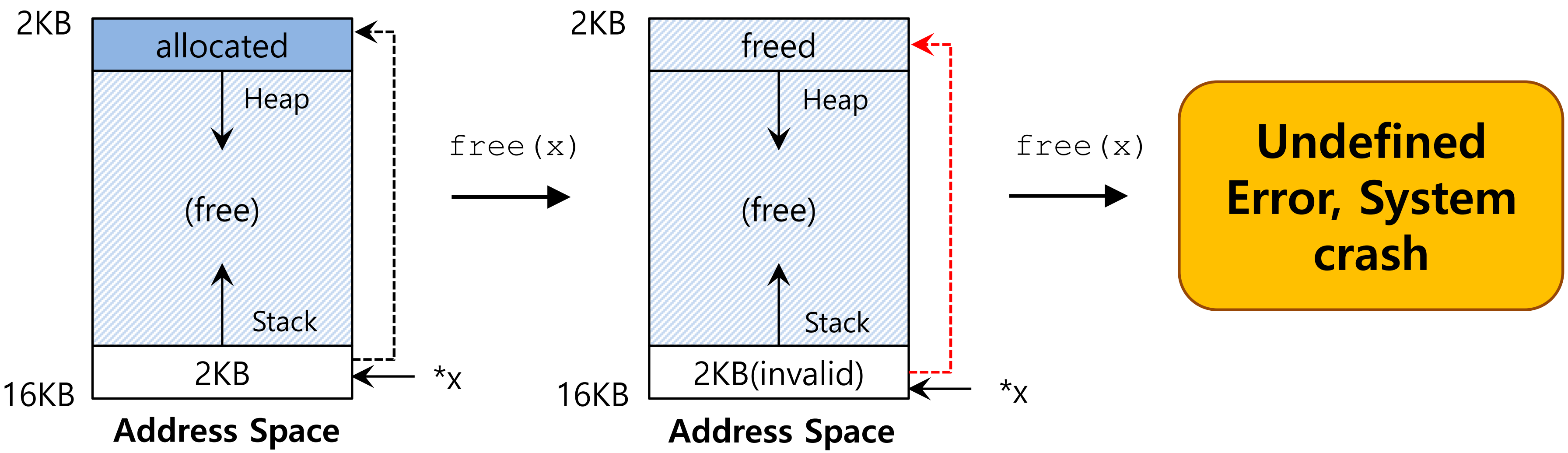
```
#include <stdlib.h>

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.

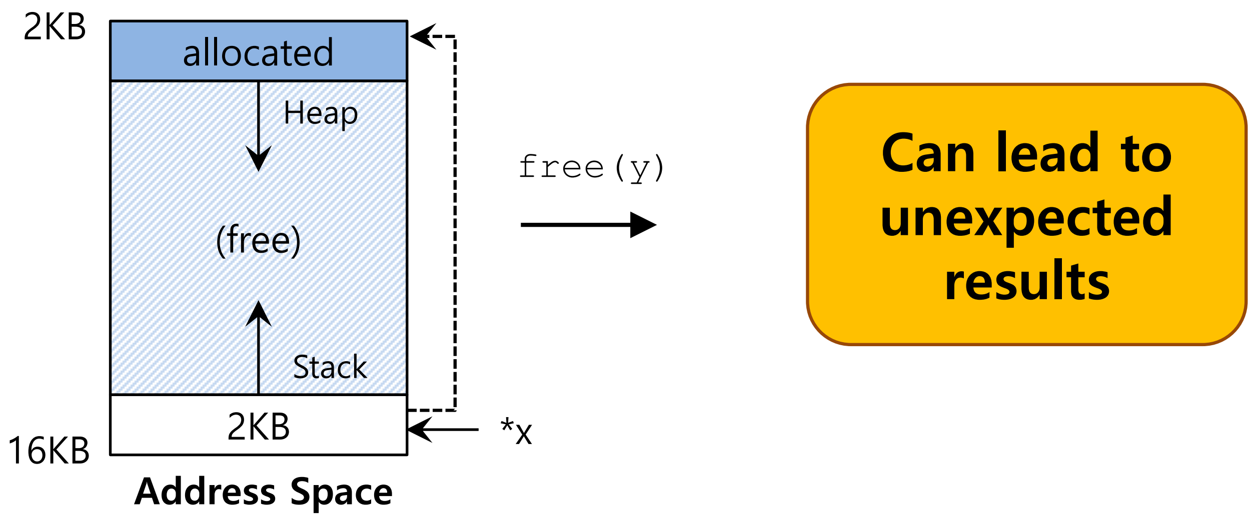
```
int *x = (int *)malloc(sizeof(int)); // allocated
free(x); // free memory
free(x); // free repeatedly
```



Invalid free()

- Calling free() incorrectly

```
int *x = (int *)malloc(sizeof(int)); // allocated
free(y); // only pass pointers received from malloc
```



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

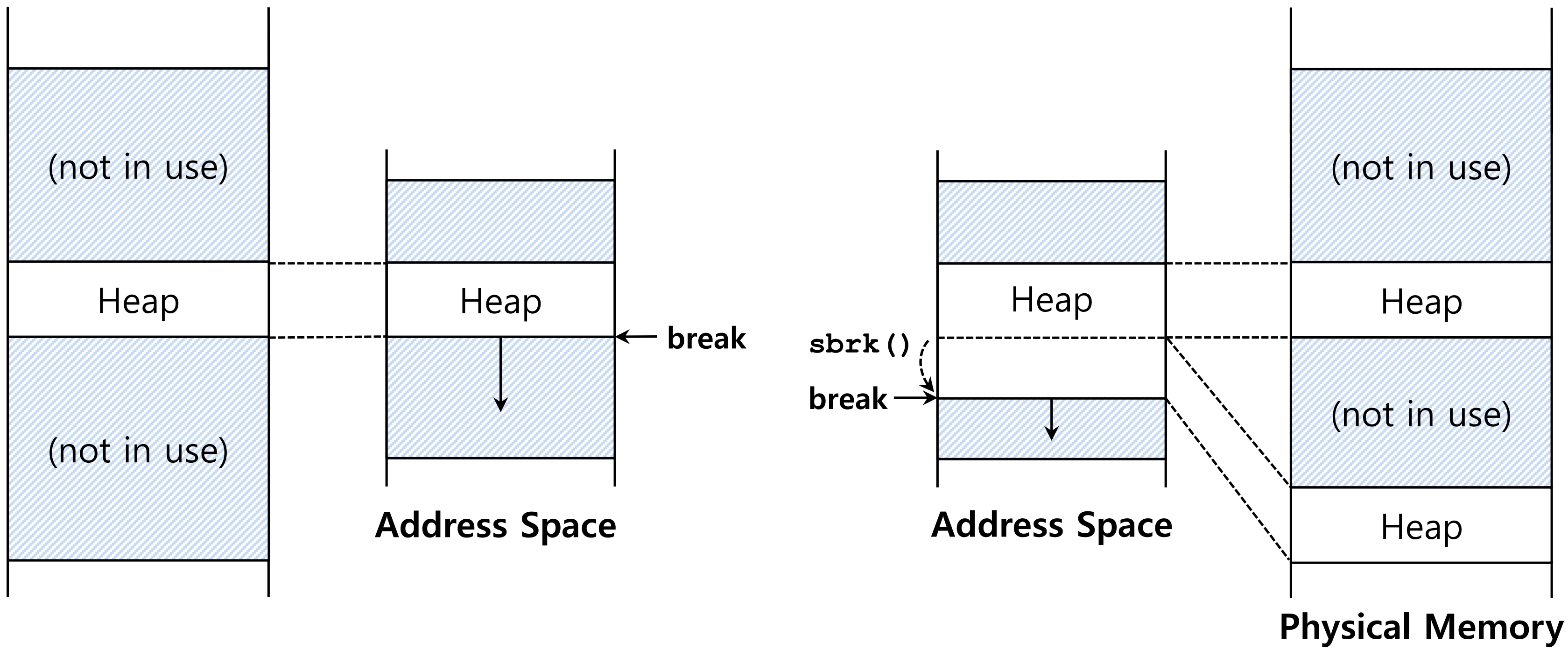
```
#include <unistd.h>
int brk(void *addr)
void *sbrk(intptr_t increment);
```

- `malloc` library call uses `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`.
  - `Argument`
    - `brk: new value of break`
    - `sbrk: increment to current heap size`
  - `Return`
    - `brk returns 0 on success, -1 otherwise`
    - `sbrk returns pointer to the prior break, otherwise (void*)-1`

Programmers **should never directly call** either `brk` or `sbrk`.

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.



System Calls(Cont.)

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

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

- **mmap**() creates a new mapping in the virtual address space of the calling process.
- Example

```
// 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);
```

- ptr: starting address or NULL
- prot: PROT_READ, PROT_WRITE, PROT_EXEC
- flags: MAP_ANON, MAP_PRIVATE, MAP_SHARED
- fd and offset set to -1 and 0, respectively when allocating memory

Address Translation

Memory Virtualizing with 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
- Assumptions:
  - Address space placed contiguously in physical memory
  - Address space size is less than the physical memory
  - All address spaces are equal in size

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;
```

Assembly

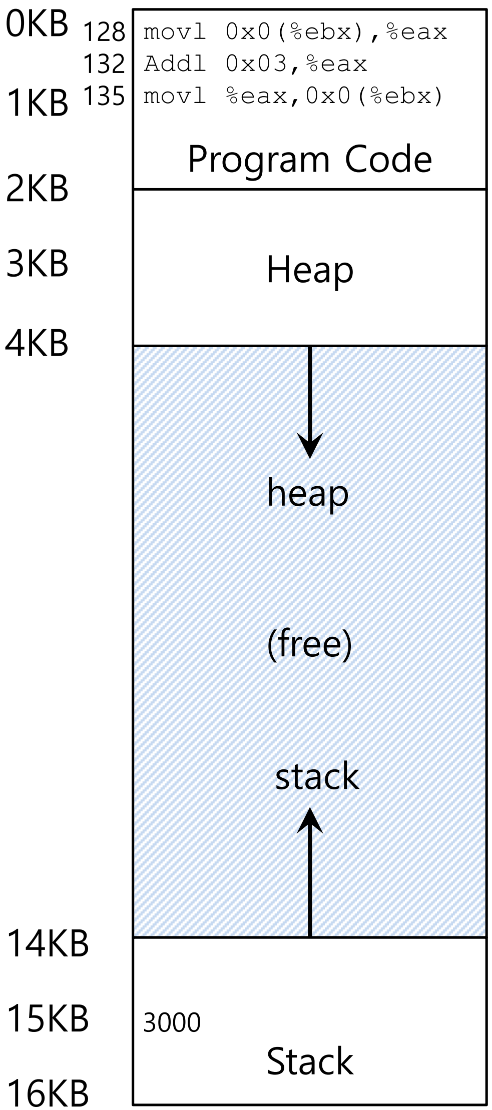
```
128 : movl 0x0(%ebx), %eax    ; load 0+ebx into eax
132 : addl $0x03, %eax        ; add 3 to eax register
135 : movl %eax, 0x0(%ebx)    ; store eax back to mem
```

- Presume that the address of 'x' has been placed in `ebx` register.

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

- **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.)

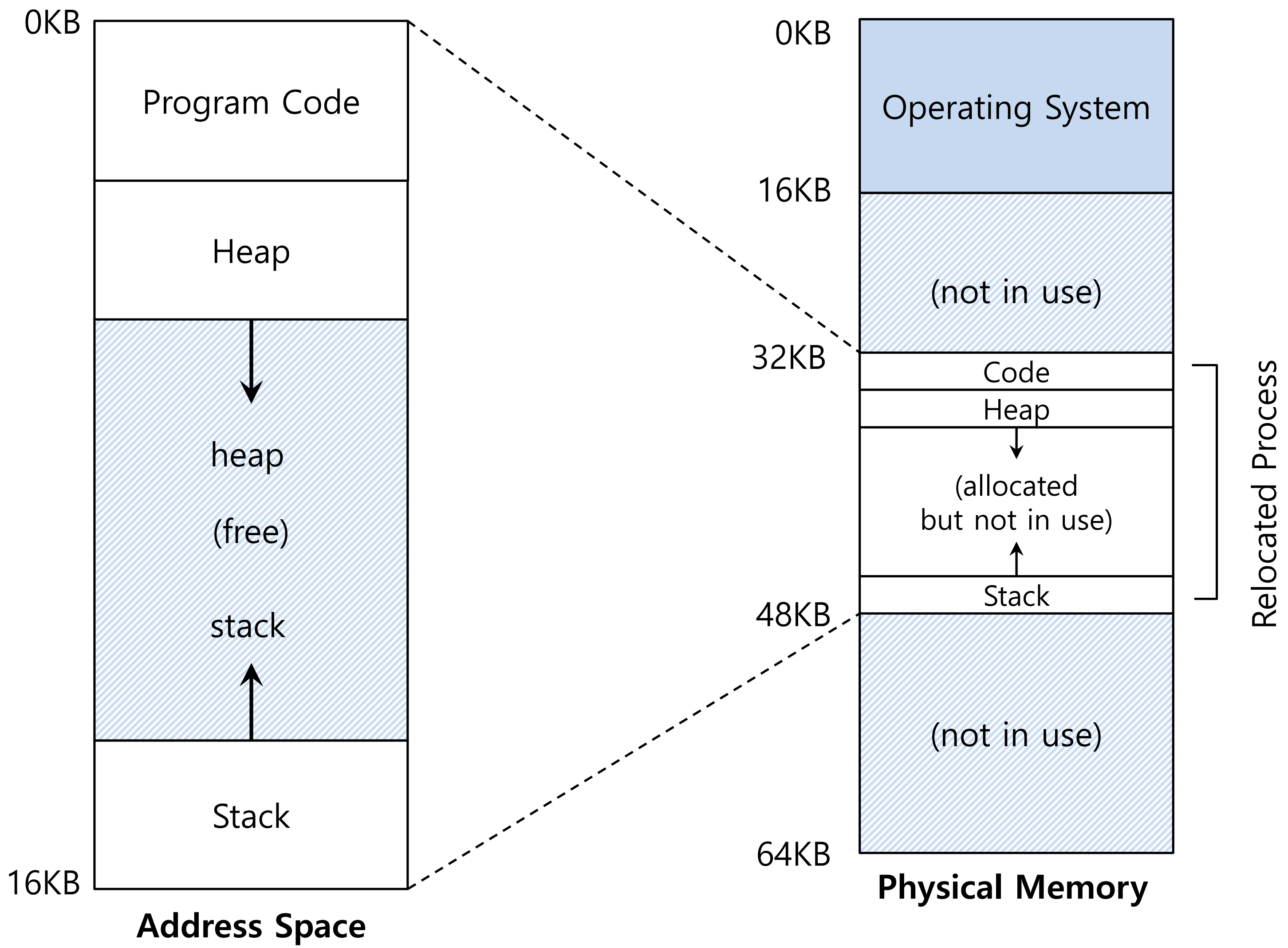


- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

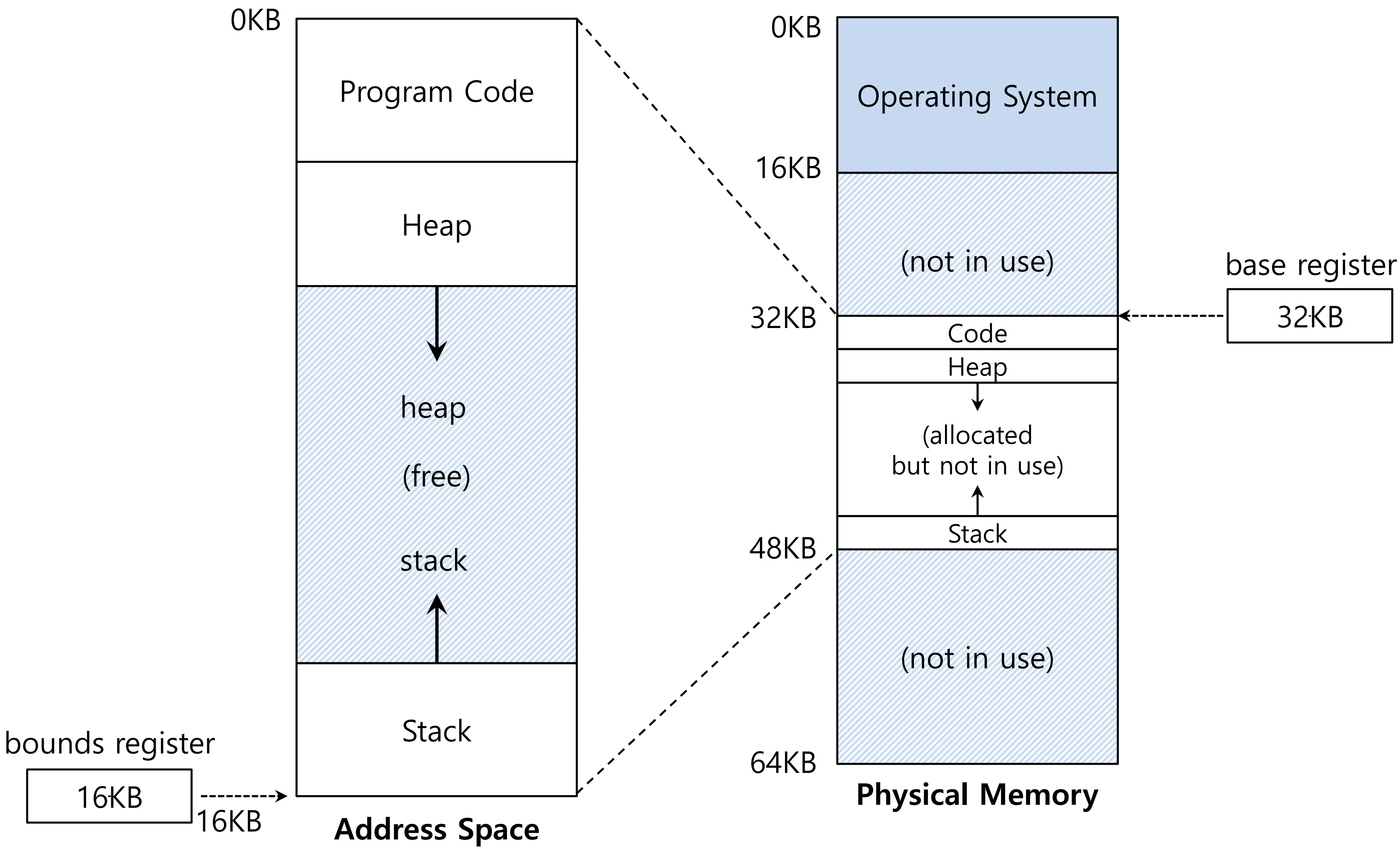
Relocation Address Space

- 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.**

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

Relocation and Address Translation

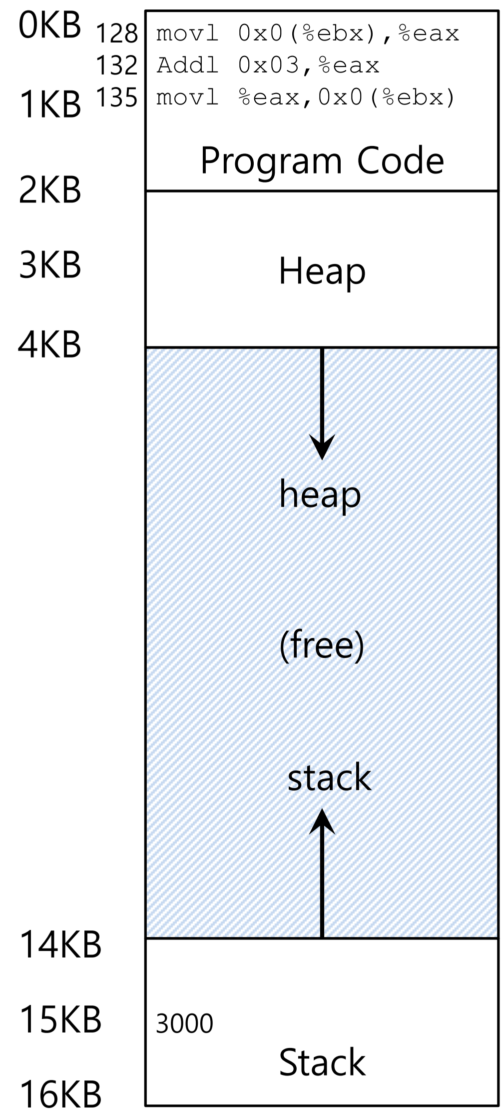
```
128 : movl 0x0(%ebx), %eax
```

- **Fetch** instruction at address 128

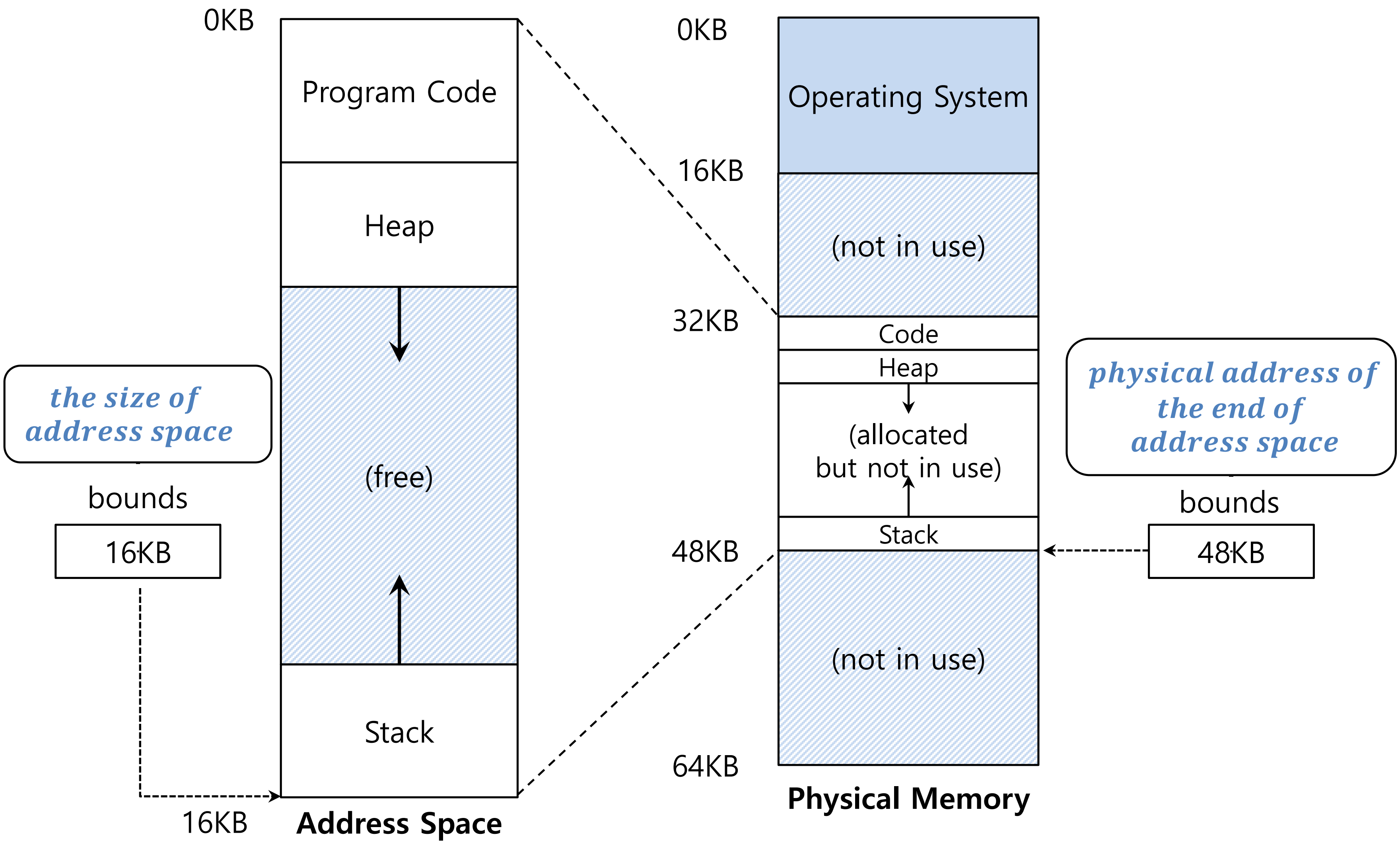
$$32896 = 128 + 32KB(base)$$

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

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



Two ways of Bounds Register



Example translations

- Assume process with an address space of size 4 KB loaded at physical address 16 KB

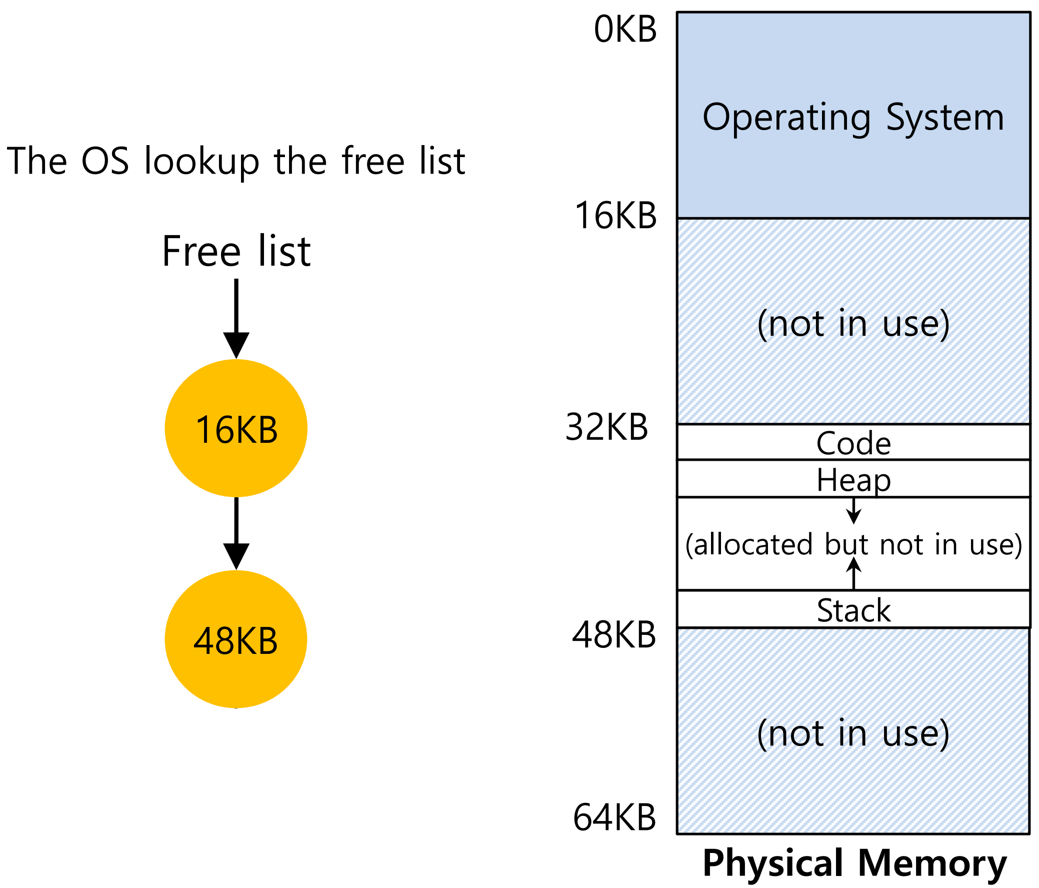
- Virtual Address 0 → Physical Address 16 KB
- VA 1 KB → PA 17 KB
- VA 3000 → PA 19384
- VA 4400 → Fault (out of bounds)

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

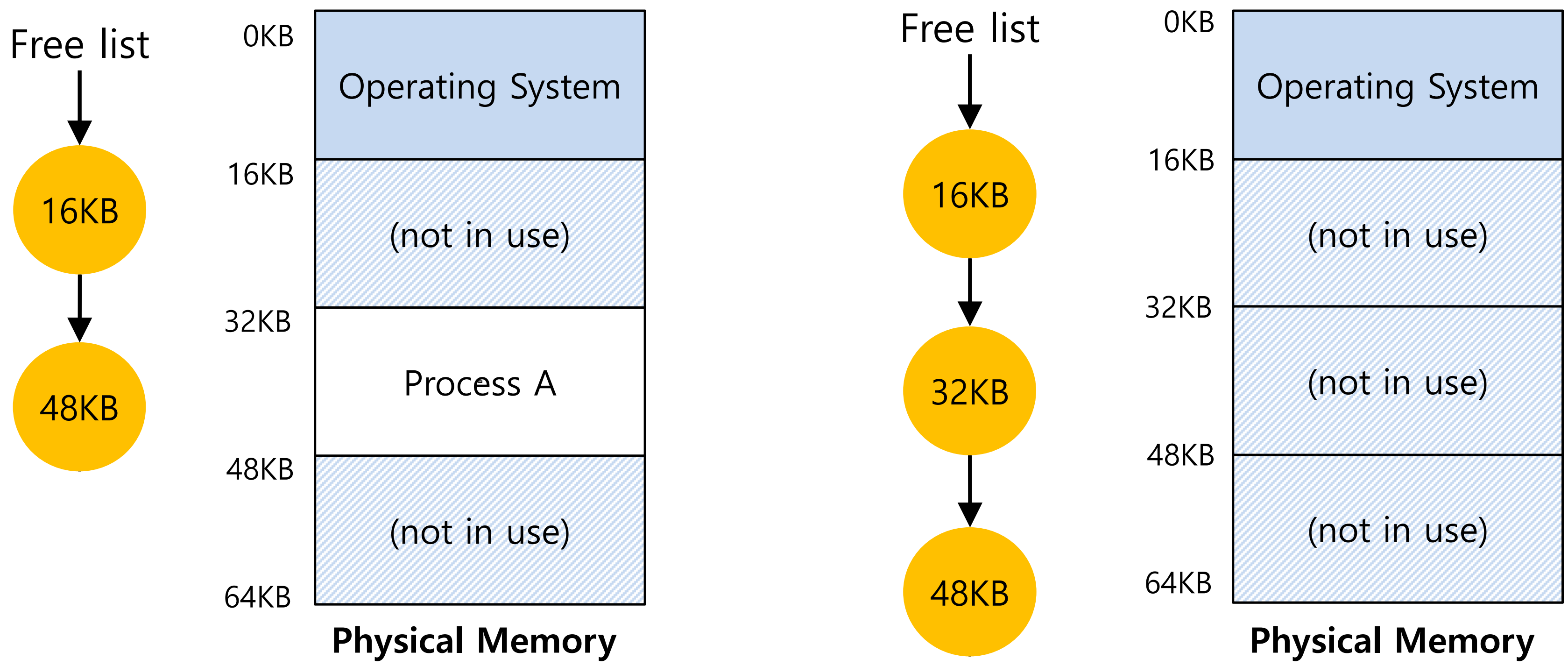
OS Issues: When a Process Starts Running

- 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.



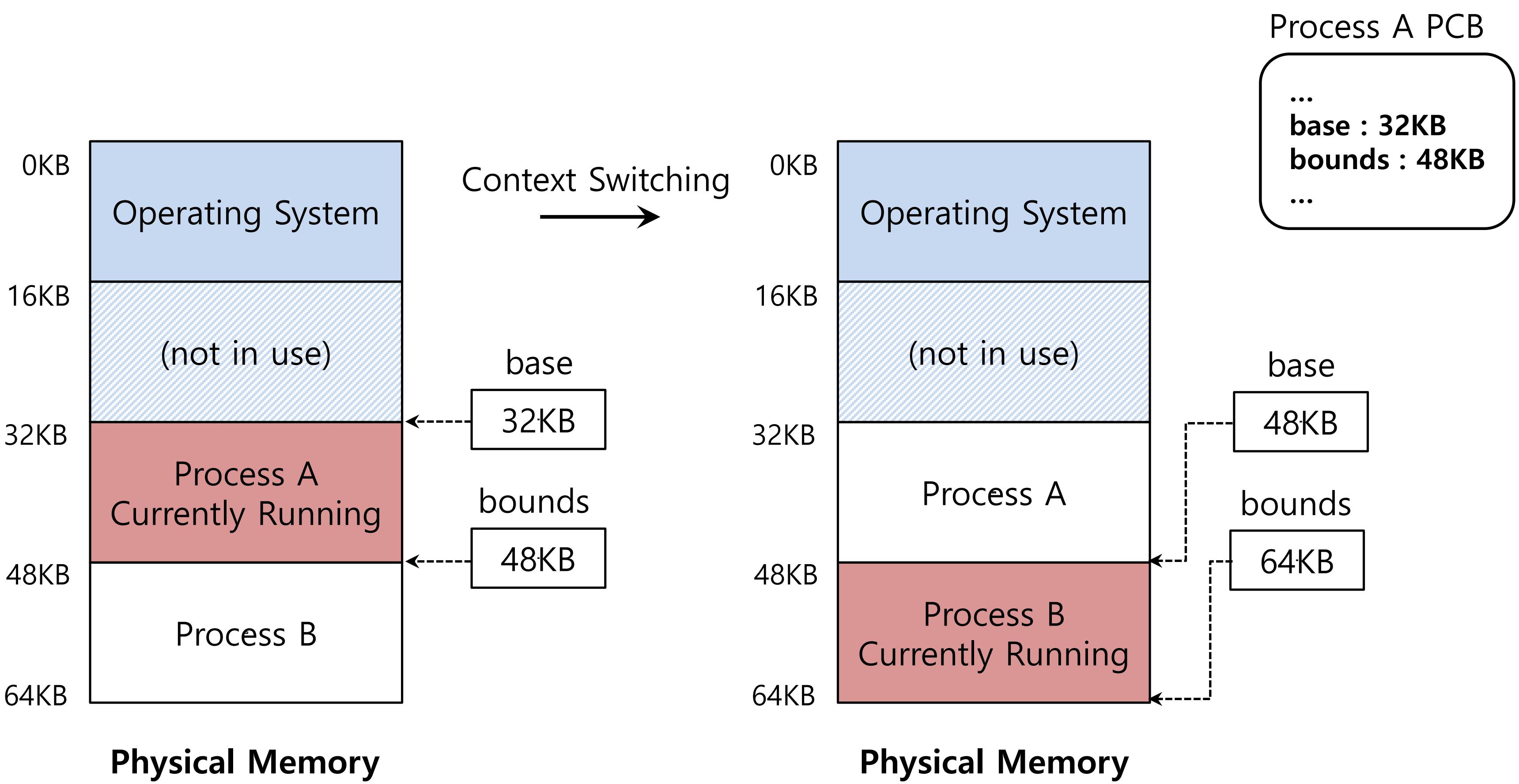
OS Issues: When a Process Is Terminated

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



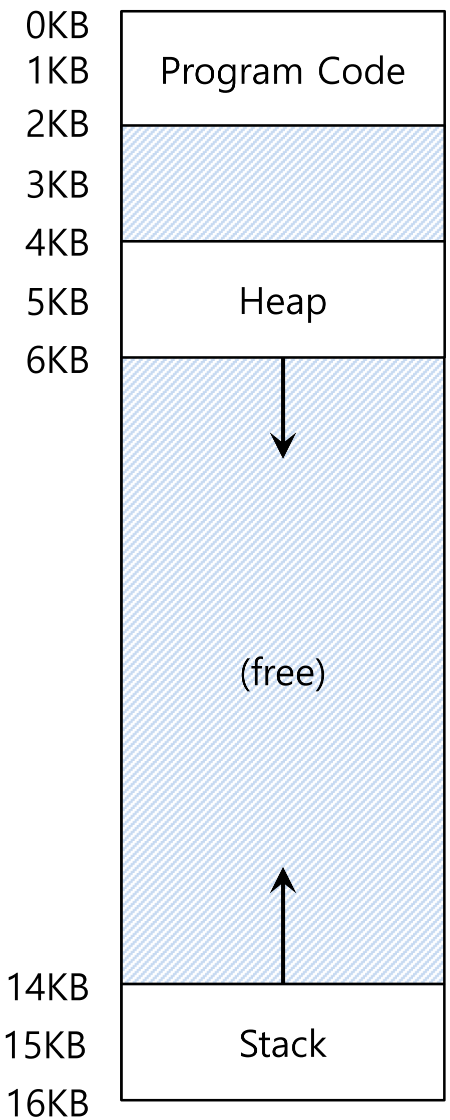
OS Issues: When Context Switch Occurs

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



Segmentation

Inefficiency of the Base and Bound Approach



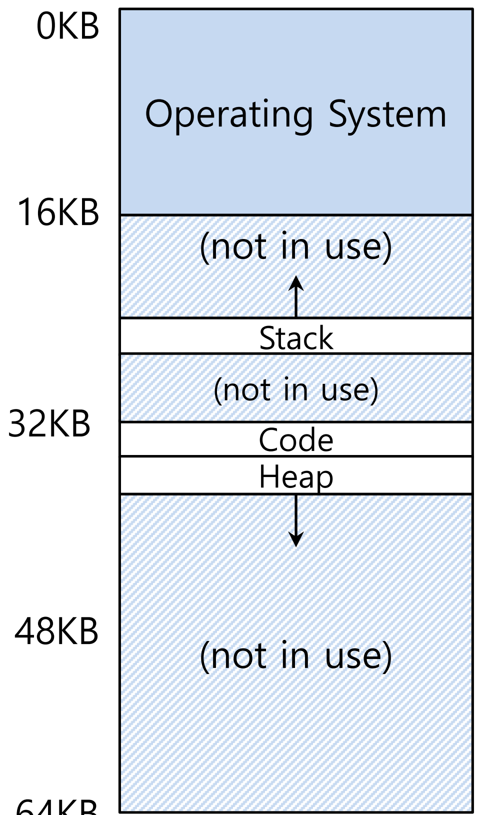
- **Big chunk of "free"** space
- "free" space **takes up** physical memory.
- Hard to run when an address space **does not fit** into physical memory

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



Physical Memory

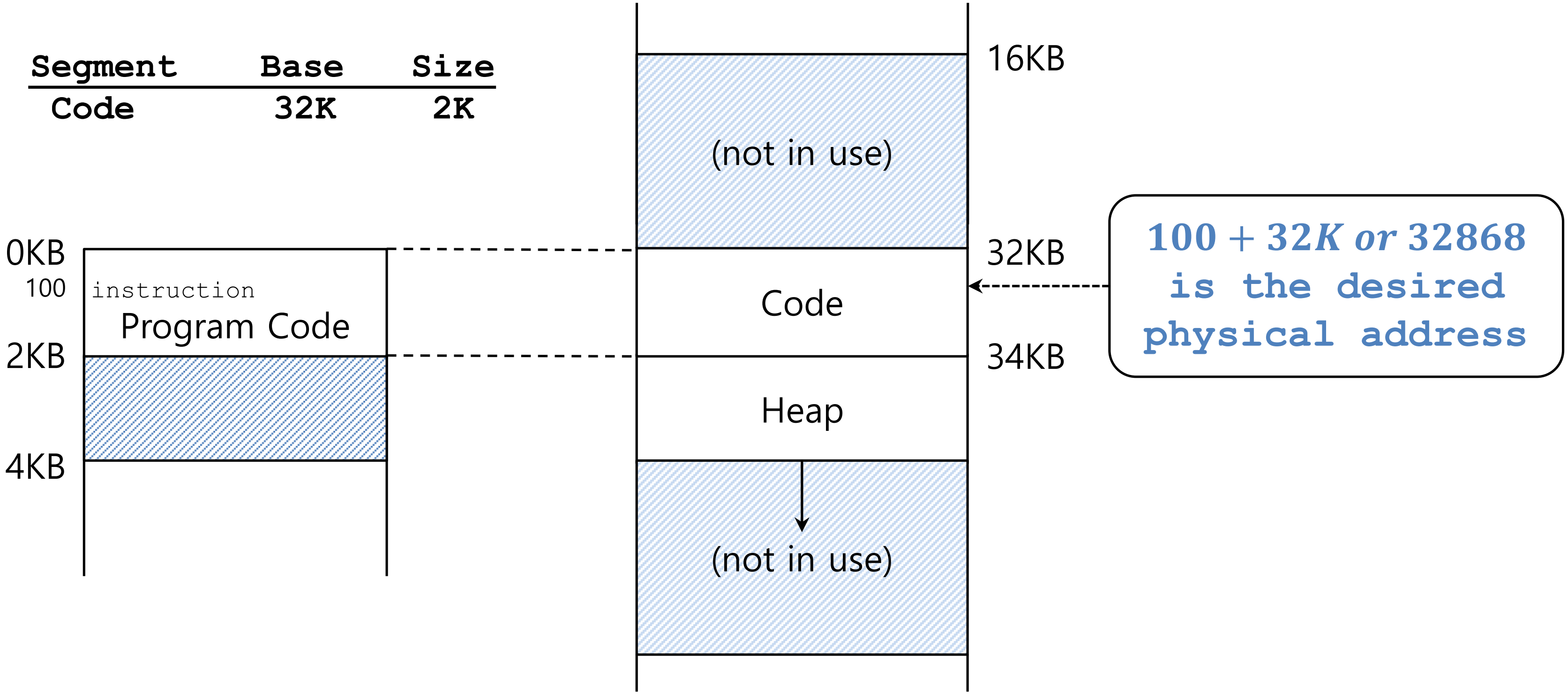
Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

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.

Segment	Base	Size
Code	32K	2K

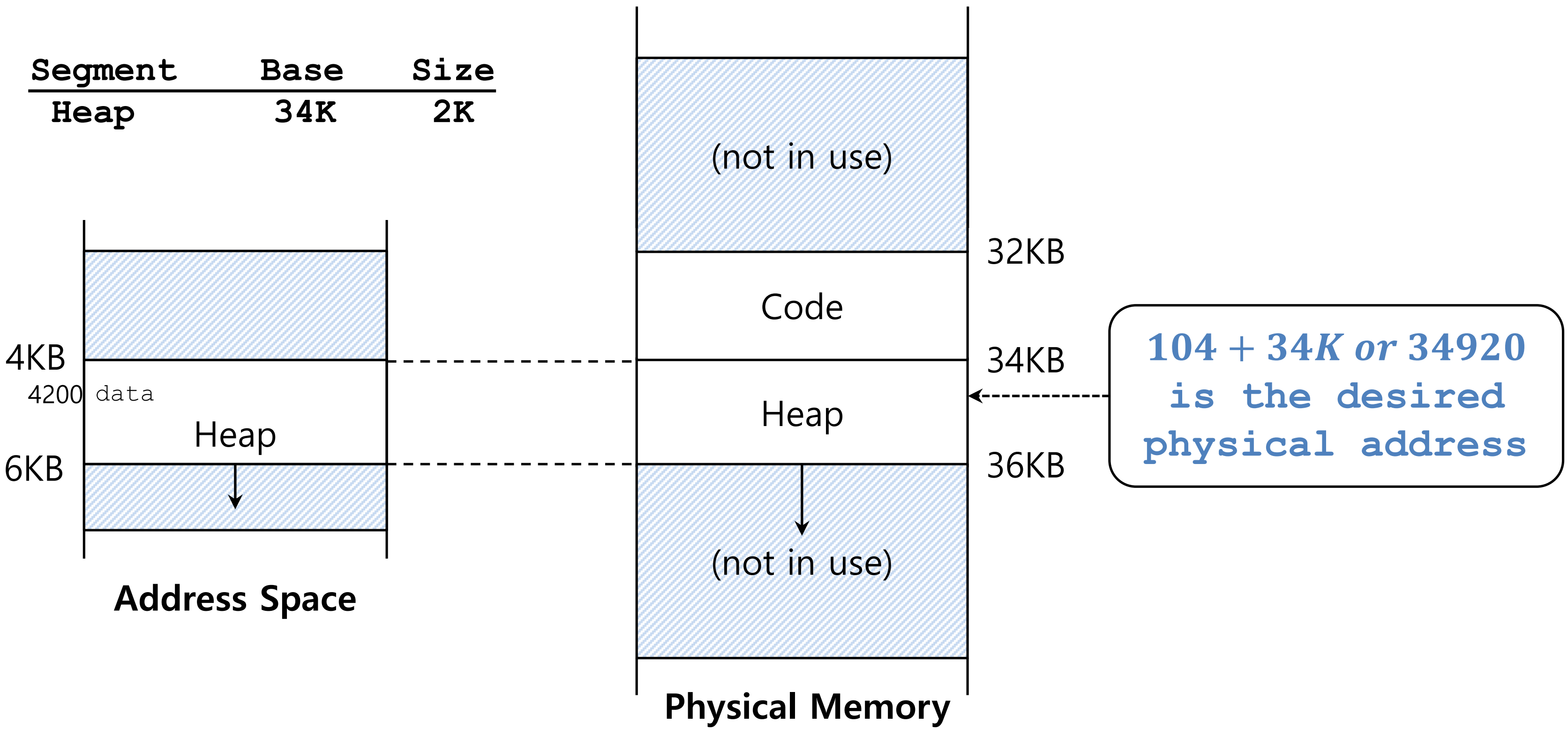


Address Translation on Segmentation(Cont.)

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

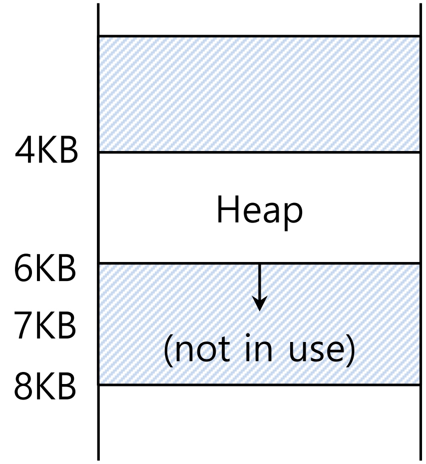
- The `offset` of virtual address 4200 is `104`.
  - The heap segment **starts at virtual address 4096** in address space.

Segment	Base	Size
Heap	34K	2K



Segmentation Fault or Violation

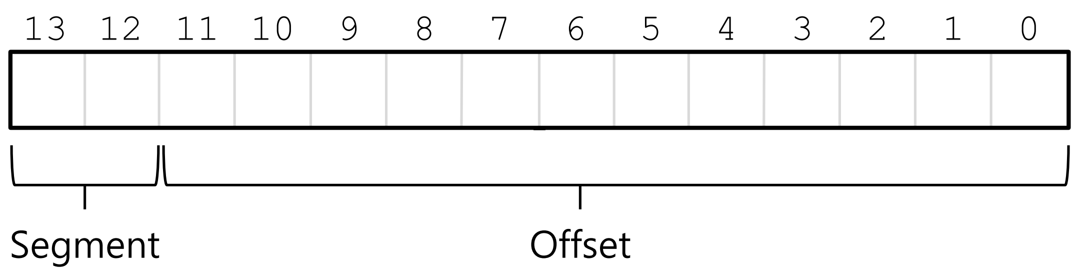
- If an **illegal address** such as 7KB which is beyond the end of heap is referenced, the OS generates **segmentation fault**.
  - The hardware detects that address is **out of bounds**.



Address Space

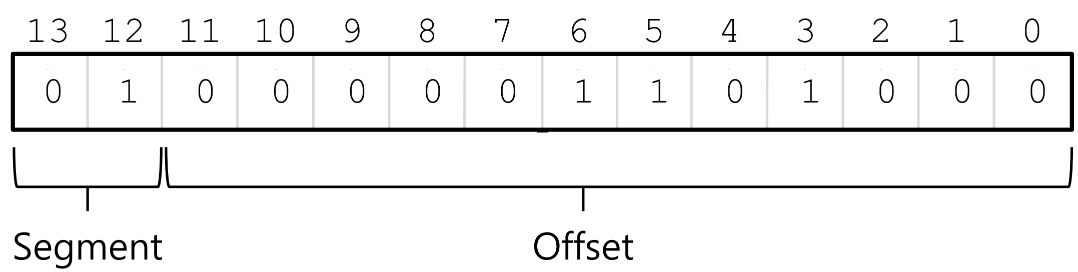
Referring to Segment

- **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



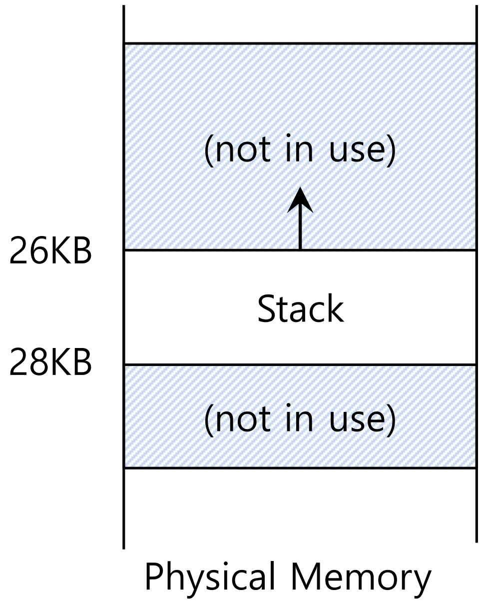
Referring to Segment(Cont.)

```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
// now get offset
Offset = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
    RaiseException(PROTECTION_FAULT)
else
    PhysAddr = Base[Segment] + Offset
    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 need.
  - The hardware checks which way the segment grows.
  - 1: positive direction, 0: negative direction



Segment Register(with Negative-Growth Support)

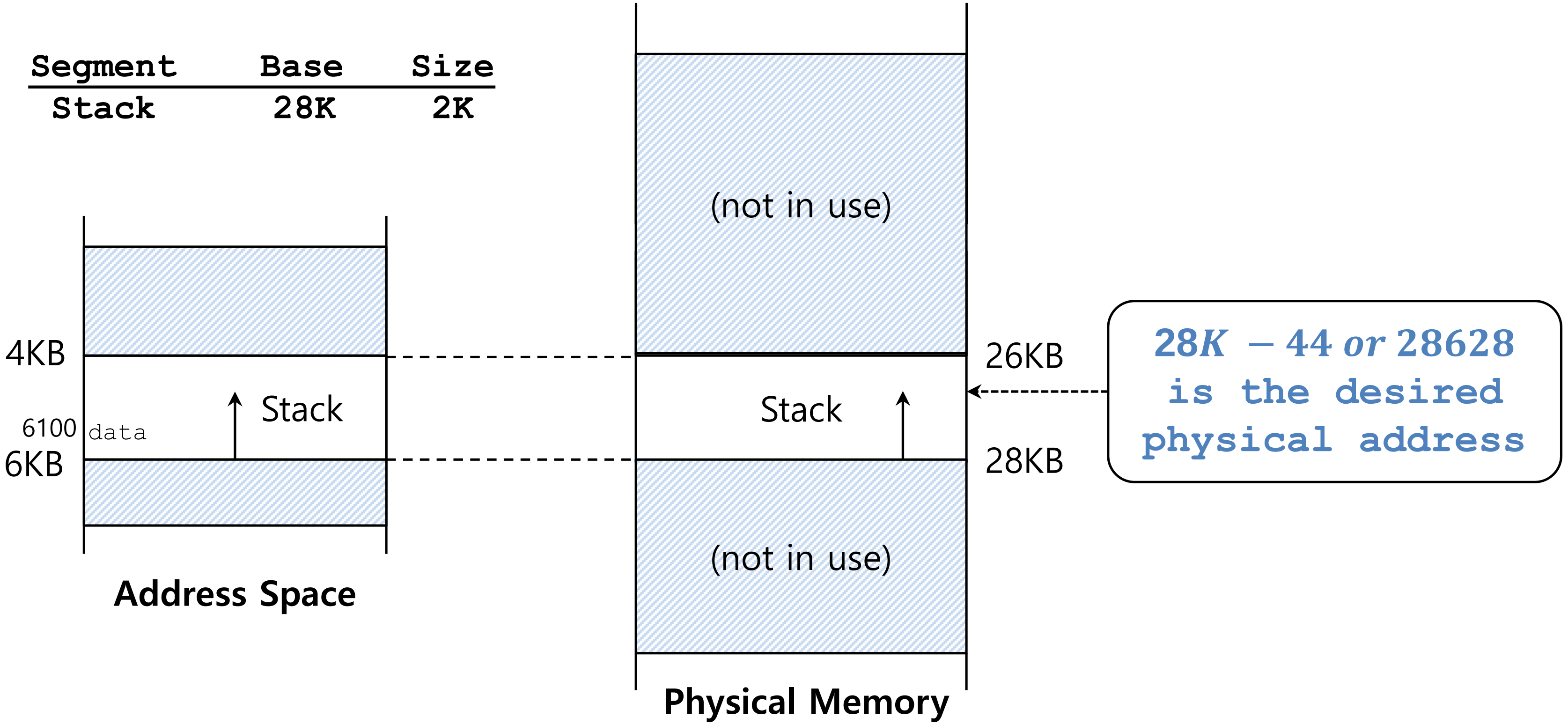
Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

Address Translation on Segmentation(Cont.)

Virtual address + base is not the correct physical address.

- The `offset` of virtual address 6100 is `?`.
  - The stack segment **starts at virtual address 6144** in address space.

Segment	Base	Size
Stack	28K	2K



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 in the 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

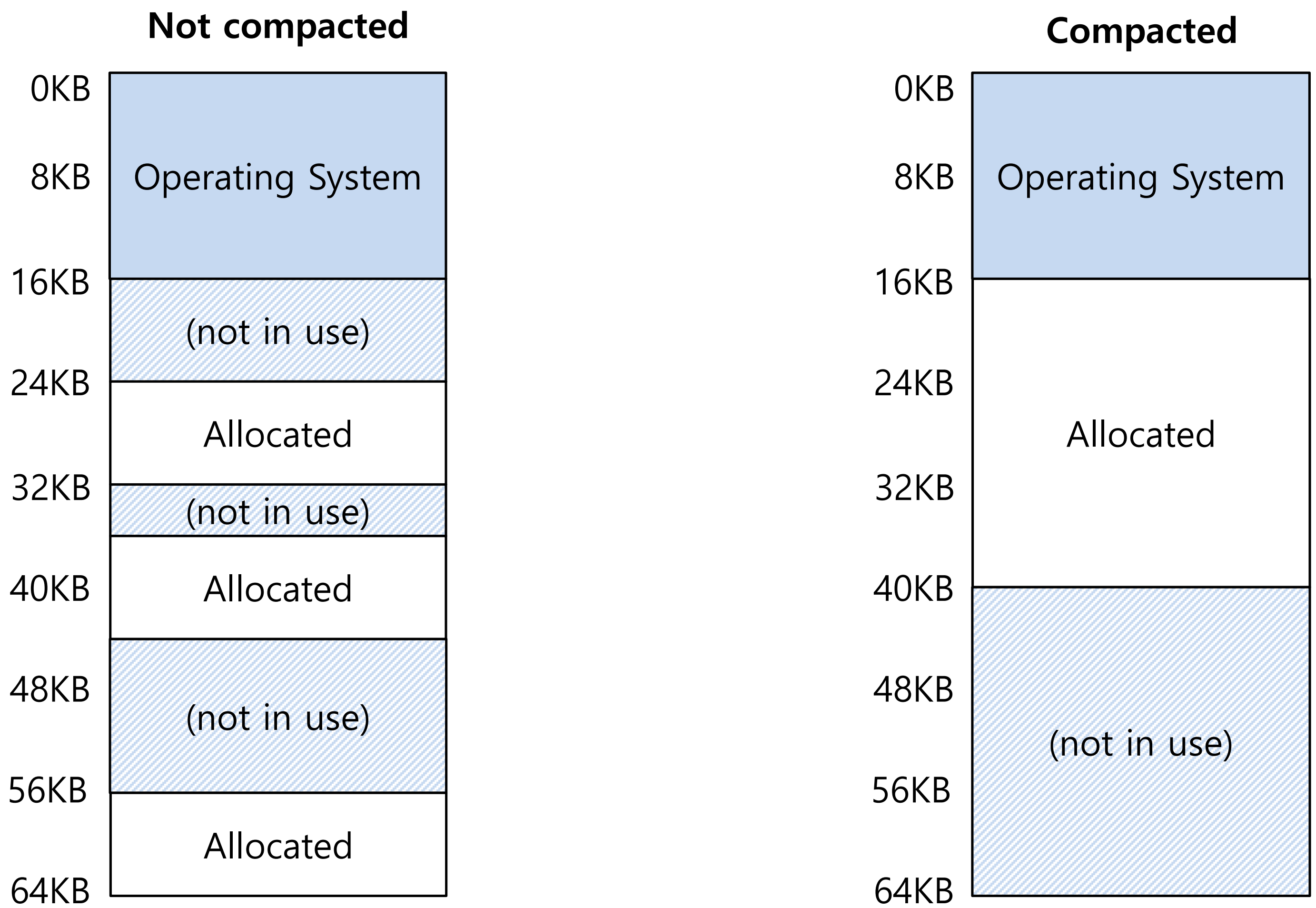
- **Coarse-Grained** means segmentation in a small number.
  - e.g., code, heap, stack.
- **Fine-Grained** segmentation allows **more flexibility** for address space in some early system.
  - To support many segments, Hardware support with a **segment table** is required.

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

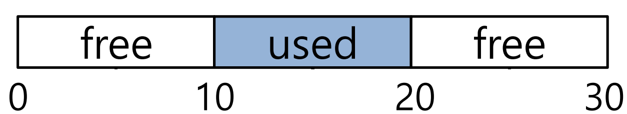


Free-Space Management

Free space management

- Managing fixed-sized memory units is easy
  - Allocate the first free block
- Not so easy with **variable-sized units**:
  - user-level memory-allocation library (malloc() and free())
  - OS managing physical memory when using **segmentation** to implement virtual memory.
  - **external fragmentation**: free space gets chopped into little pieces of different sizes and is thus fragmented.
    - Request of 20 bytes cannot be fulfilled

30-byte heap:

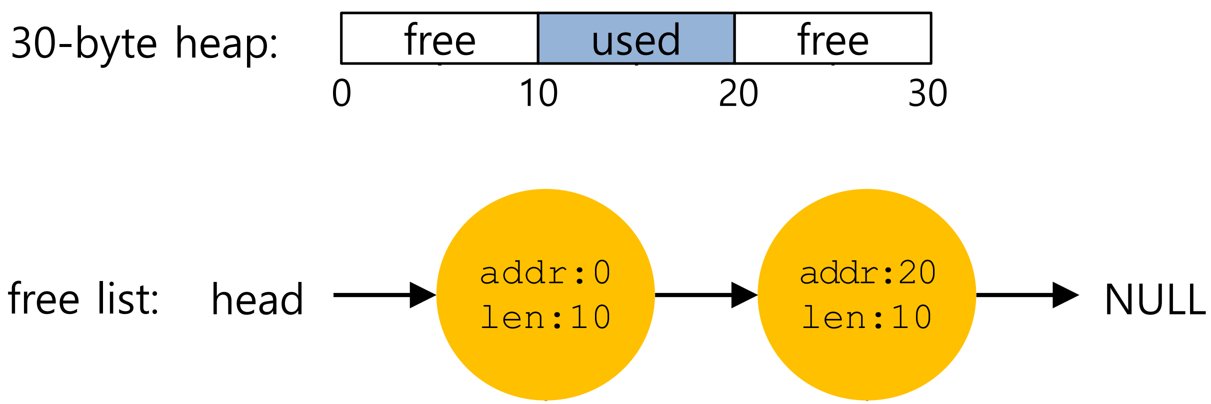


Low level mechanisms of memory allocators

- Splitting
- Coalescing

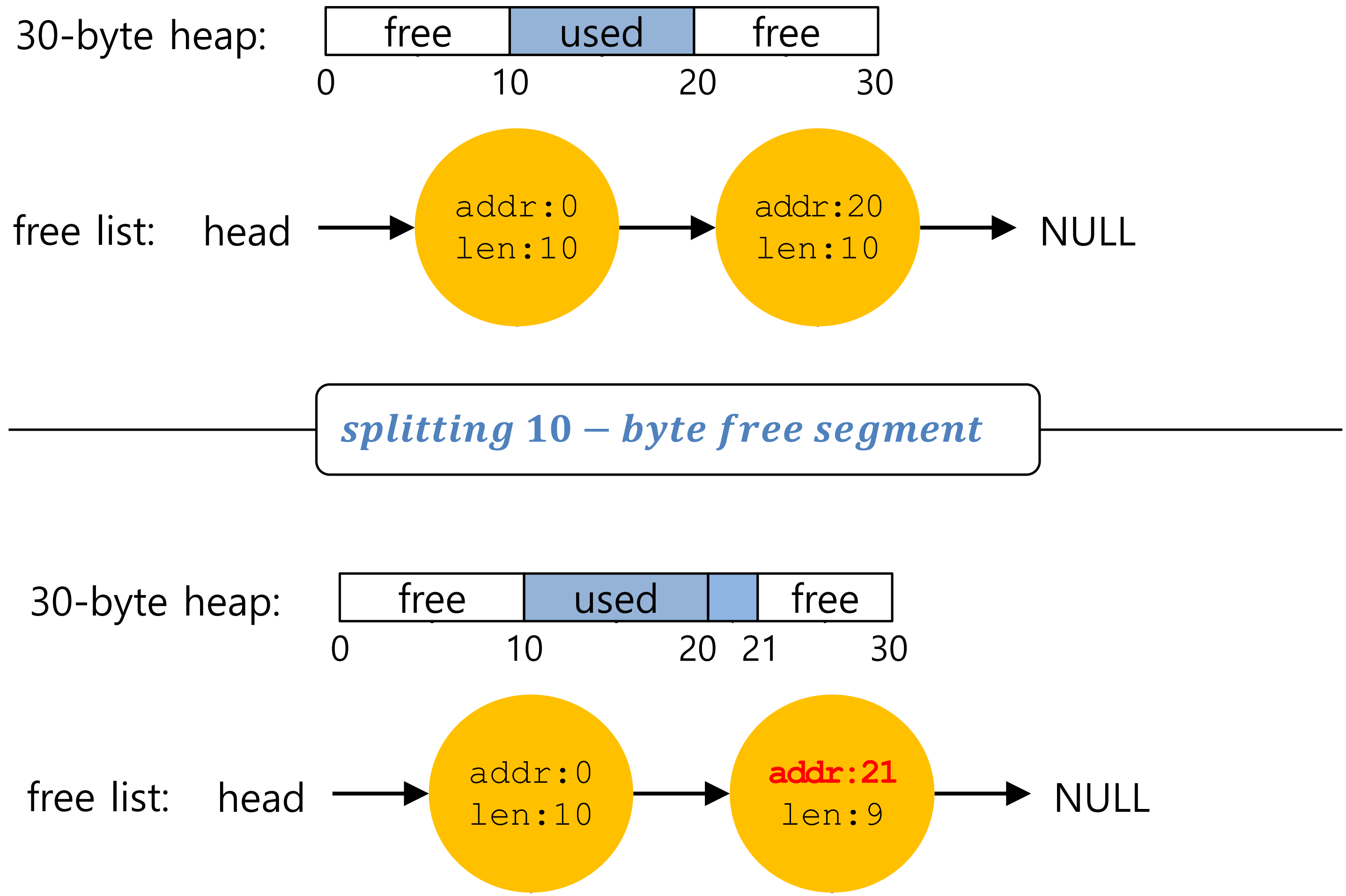
Splitting

- 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.



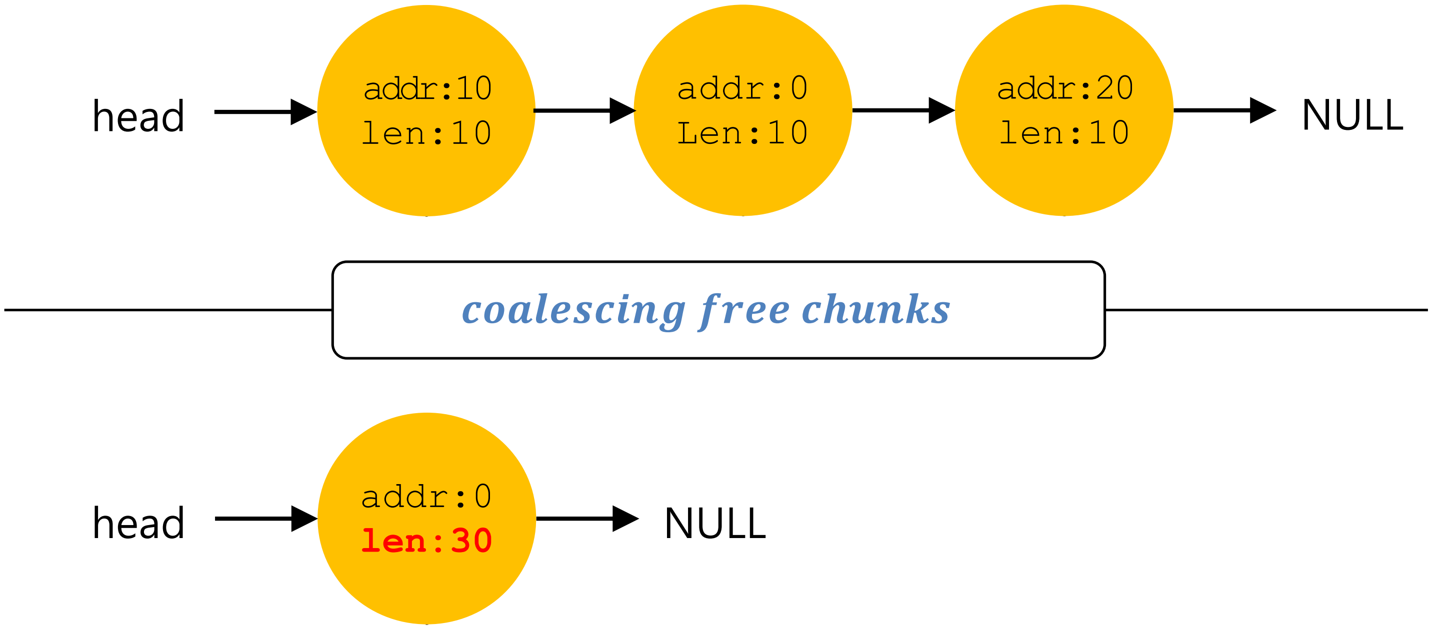
Splitting(Cont.)

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



Coalescing

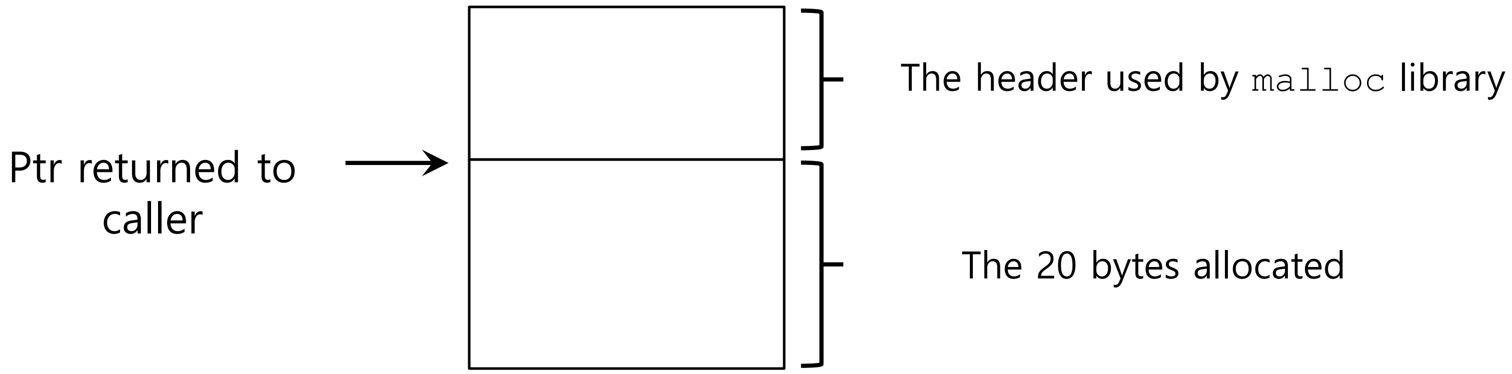
- If a user requests memory that is **bigger than free chunk size**, the list will **not find** such a free chunk.
- Coalescing: **Merge** returning a free chunk with existing chunks into a large single free chunk if **addresses** of them are **nearby**.



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.

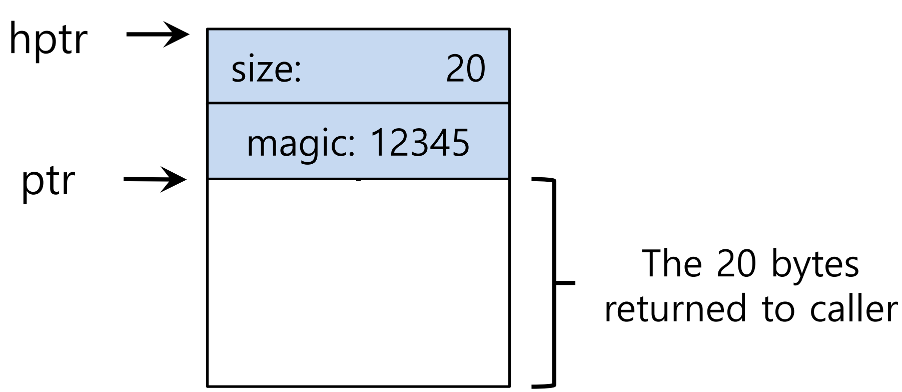
```
ptr = malloc(20);
```



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 in a free list
  - Assume that the heap is built via `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;
```

- **mmap**() creates a new mapping in the virtual or physical address space?

Interlude: mmap system Call

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

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

- **mmap**() creates a new mapping in the virtual address space of the calling process.
- Example

```
// 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);
```

- ptr: starting address or NULL
- prot: PROT_READ, PROT_WRITE, PROT_EXEC
  - Pages may be read, written to , executed
- flags: MAP_ANON, MAP_PRIVATE, MAP_SHARED
  - fd and offset set to -1 and 0, respectively when allocating memory

mmap()

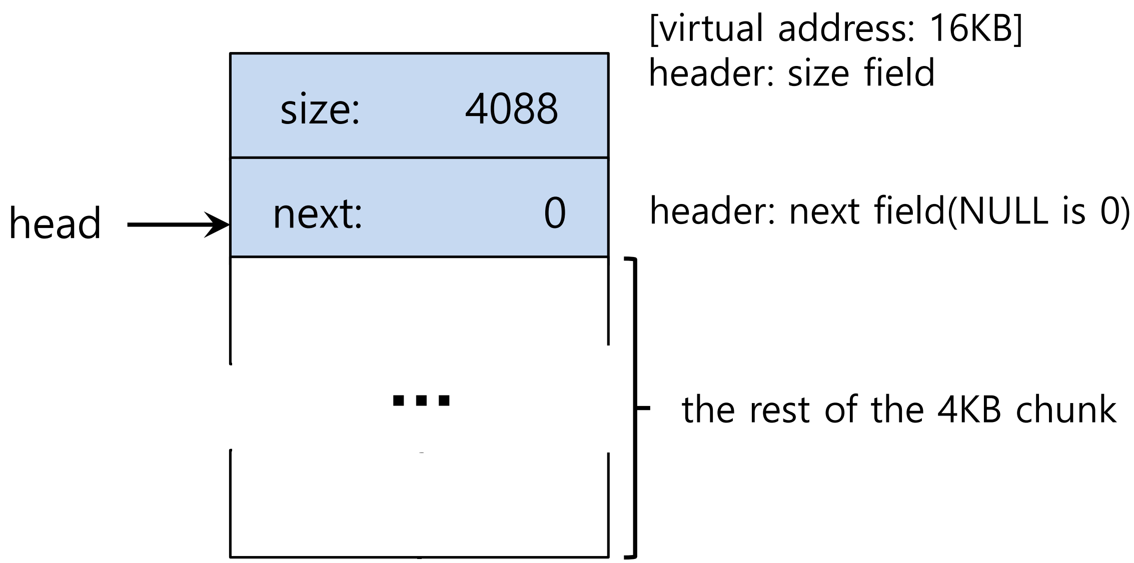
- MAP_ANON: The mapping is not backed by any file; its contents are initialized to zero. The fd argument is ignored some implementations require fd to be -1

- MAP_PRIVATE: Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file.

- MAP_SHARED: Share this mapping. Updates to the mapping are visible to other processes mapping the same region,

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;
```



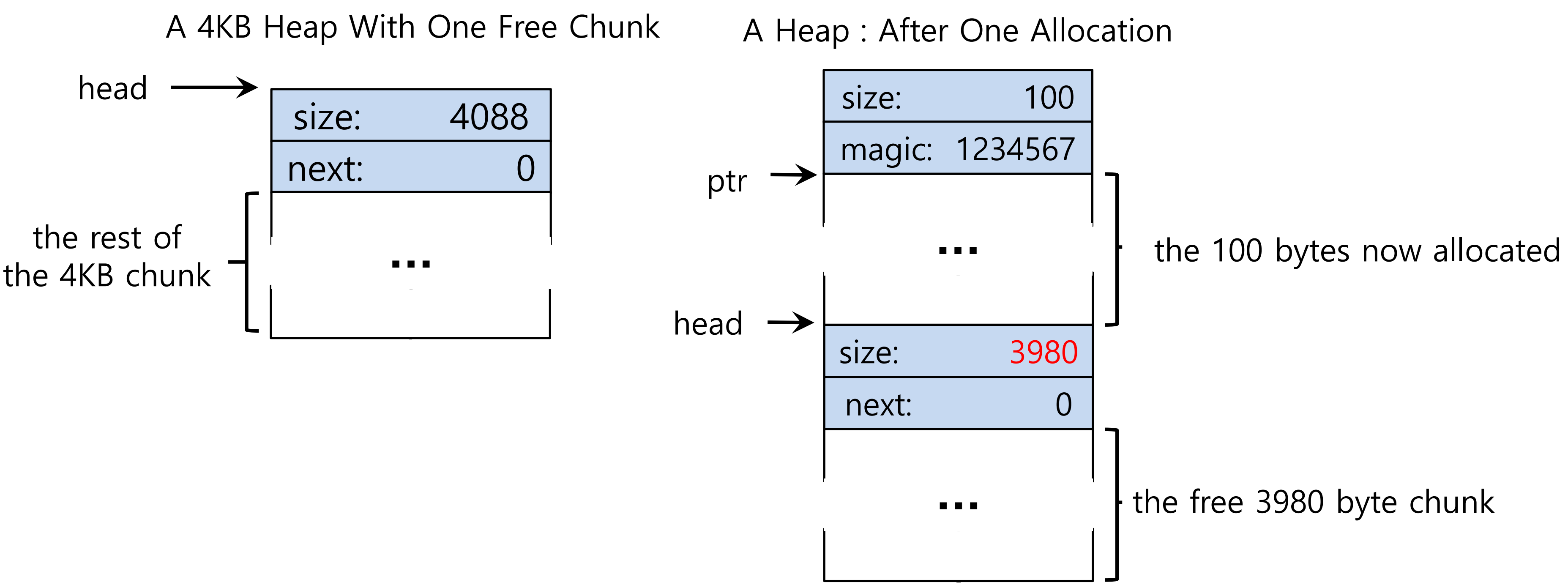
Embedding A Free List: Allocation

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

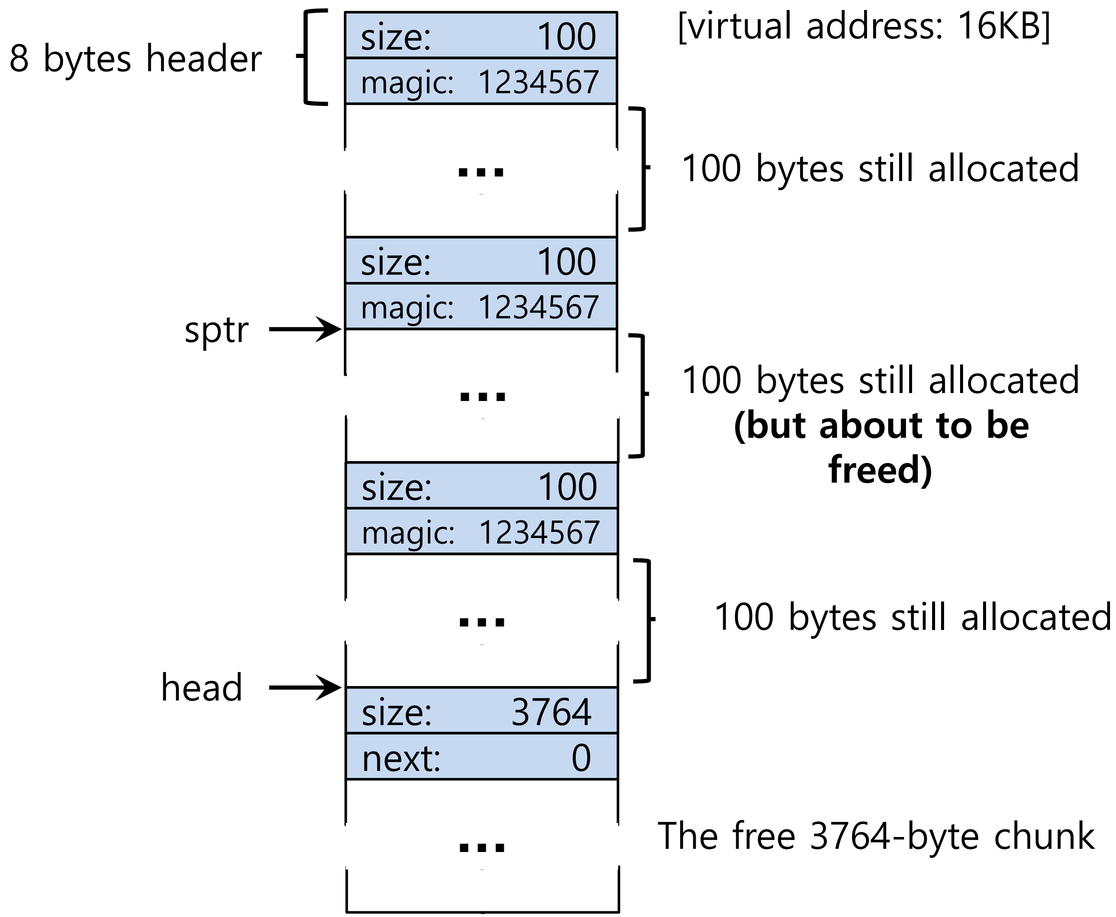
- The library will
  - **Split** the large free chunk into two.
    - **One** for the **request** and the **remaining** free chunk
  - **Shrink** the size of free chunk in the list.

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).



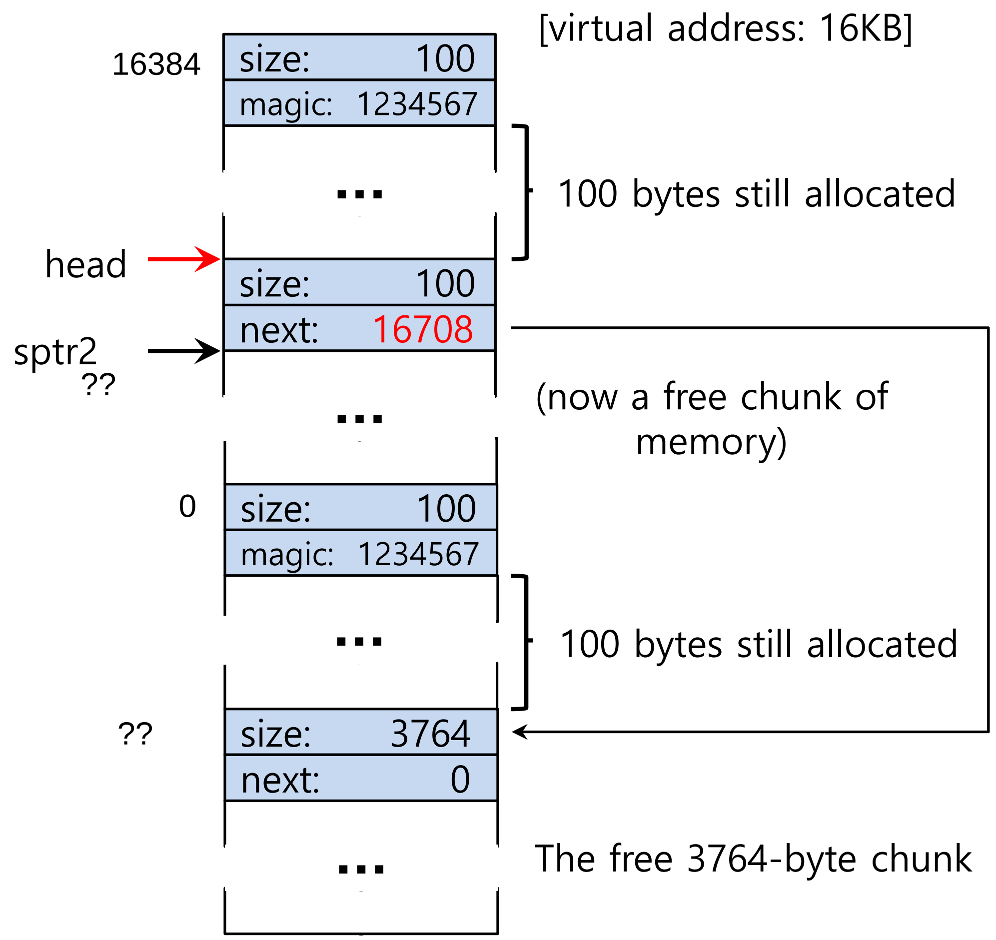
Free Space With Chunks Allocated



Free Space With Three Chunks Allocated

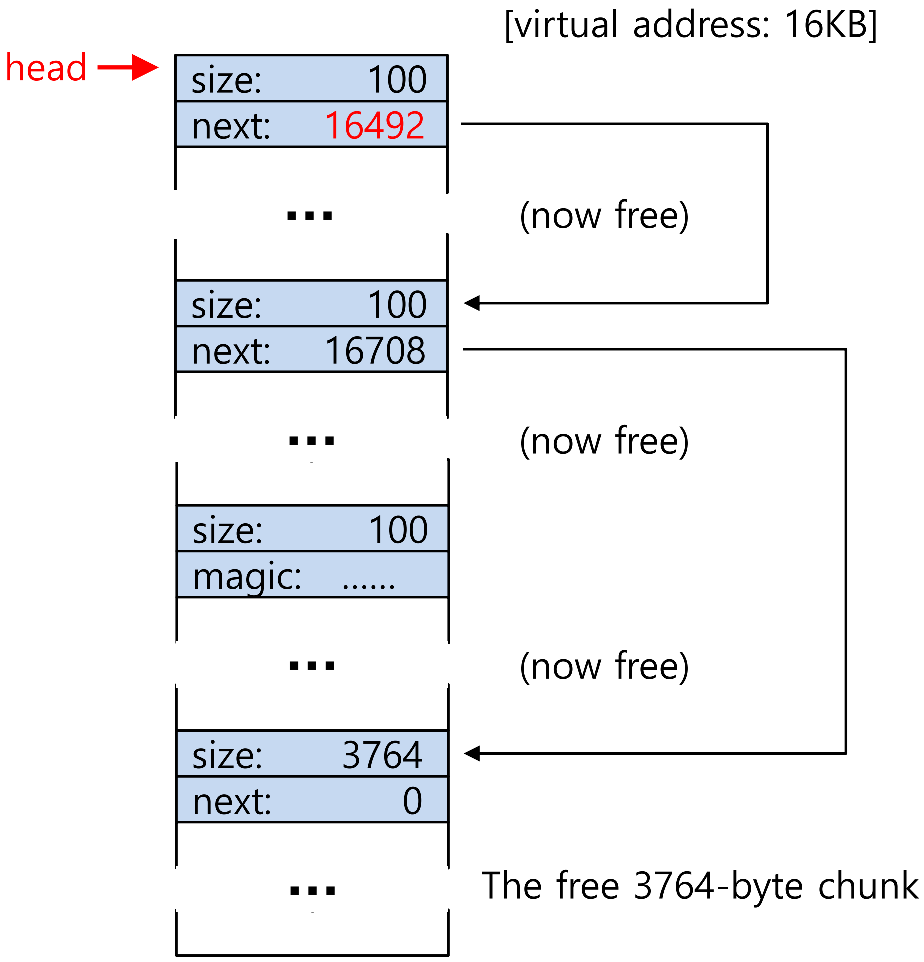
Free Space With `free()`

- Example: `free(sptr2)`
  - 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
  - Assume that the free node is **inserted at the head** of the free list.



Free Space With Freed Chunks

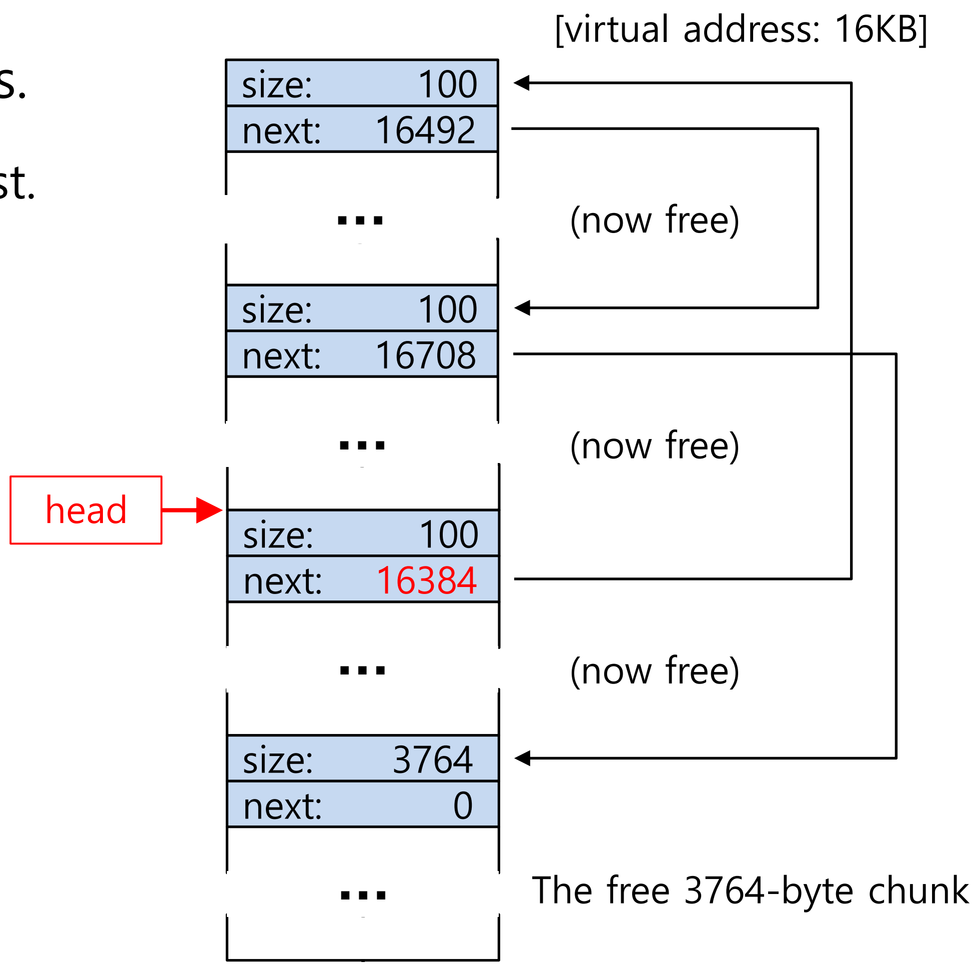
- Let's assume that the last two in-use chunks are freed.
- **External Fragmentation** occurs.
  - **Coalescing** is needed in the list.
  - Free sptr2
  - Free sptr1



Free Space With Freed Chunks

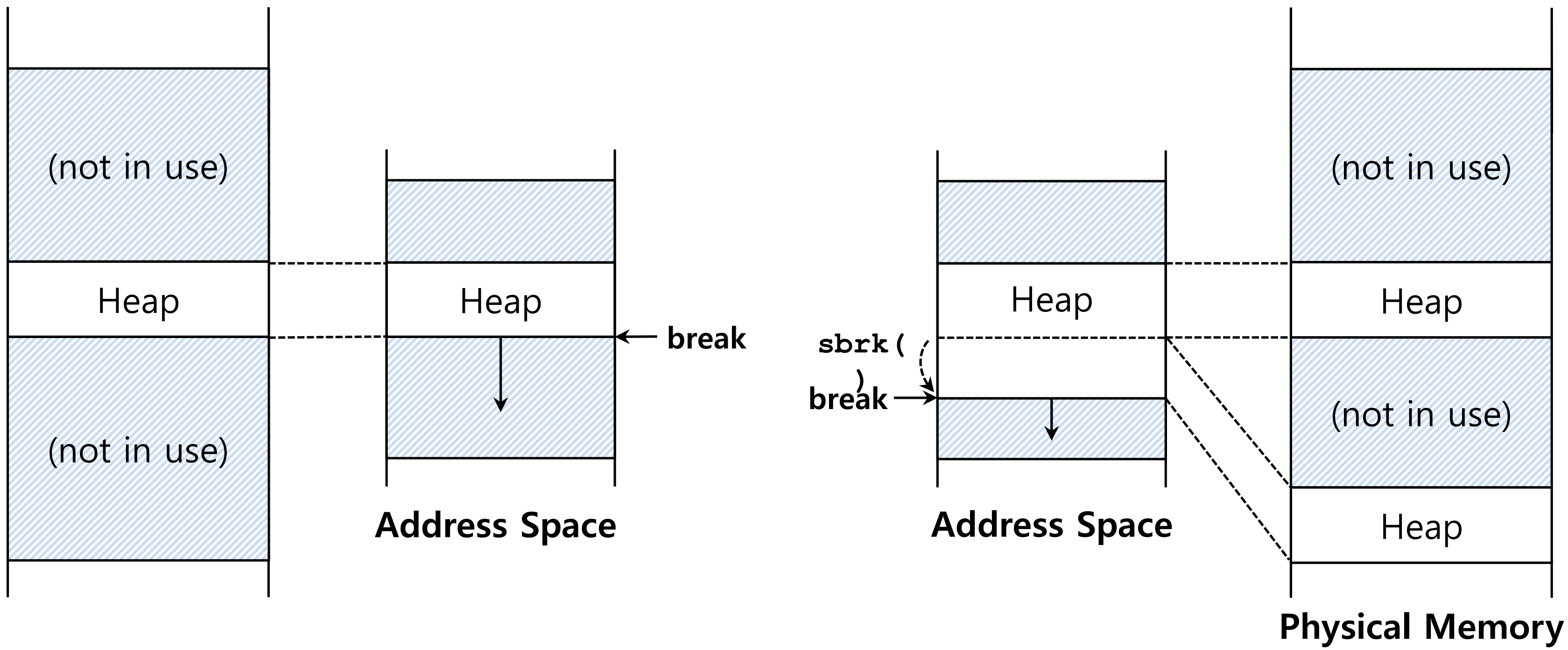
- Let's assume that the last two in-use chunks are freed.

- **External Fragmentation** occurs.
  - **Coalescing** is needed in the list.
  - Free sptr2
  - Free sptr1
  - Free sptr3



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

- Next Fit:
  - Finding the first chunk that is big enough for the request.
  - Searching at **where one was looking** at instead of the beginning of the list.

Examples of Basic Strategies

- Allocation Request Size 15

head → 10 → 30 → 20 → NULL

- Result of Best-fit

head → 10 → 30 → 5 → NULL

- Result of Worst-fit

head → 10 → 15 → 20 → NULL

Other Approaches: Segregated List

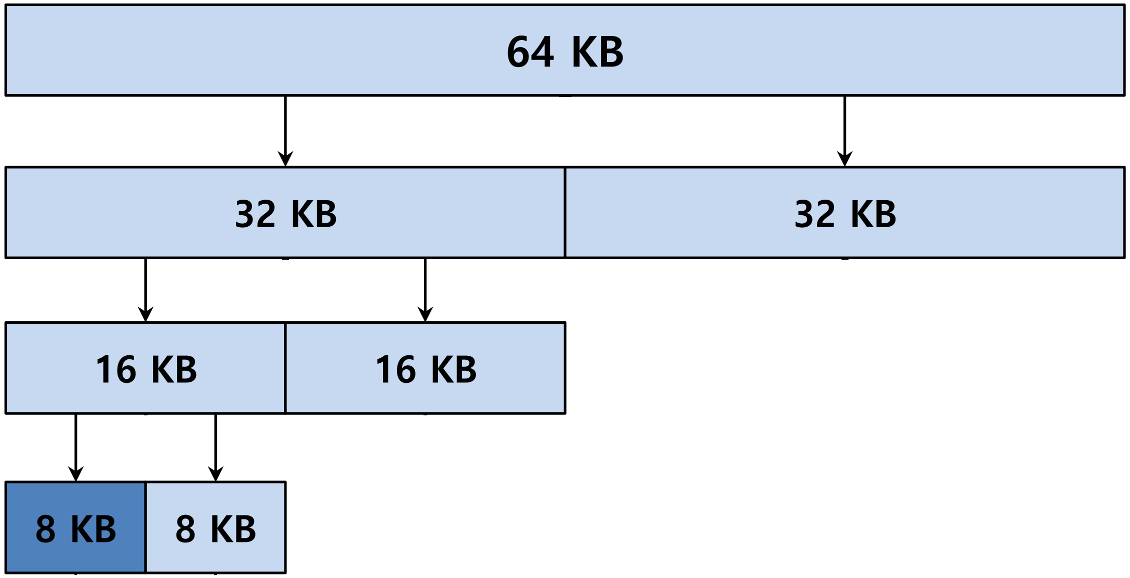
- Segregated List:
  - Keeping free chunks in different size in a separate list for the size of popular request.
  - New Complication:
    - **How much** memory should dedicate to **the pool of memory** that serves **specialized requests** of a given size?
  - **Slab allocator** handles this issue.

Other Approaches: Segregated List(Cont.)

- Slab Allocator
  - Allocate a number of object caches.
    - The objects are likely to be 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: 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.

Paging

Concept of Paging

- Paging **splits up** address space into **fixed-sized** unit called a **page**.
  - 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.

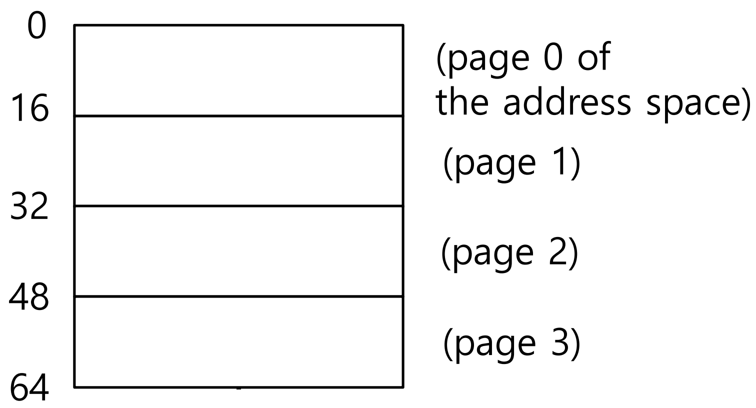
Advantages Of Paging

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

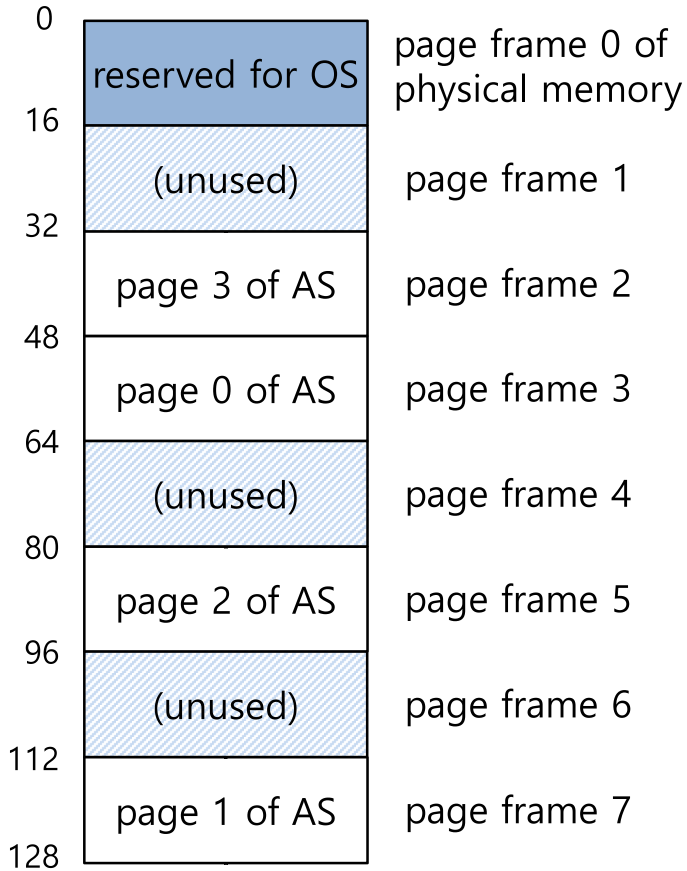
- **Simplicity**: ease of free-space management
  - The page in address space and the page frame are the same size.
  - Easy to allocate and keep a free list

Example: A Simple Paging

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



A Simple 64-byte Address Space



64-Byte Address Space Placed In Physical Memory

Address Translation

- movl <virtual address>, %eax

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

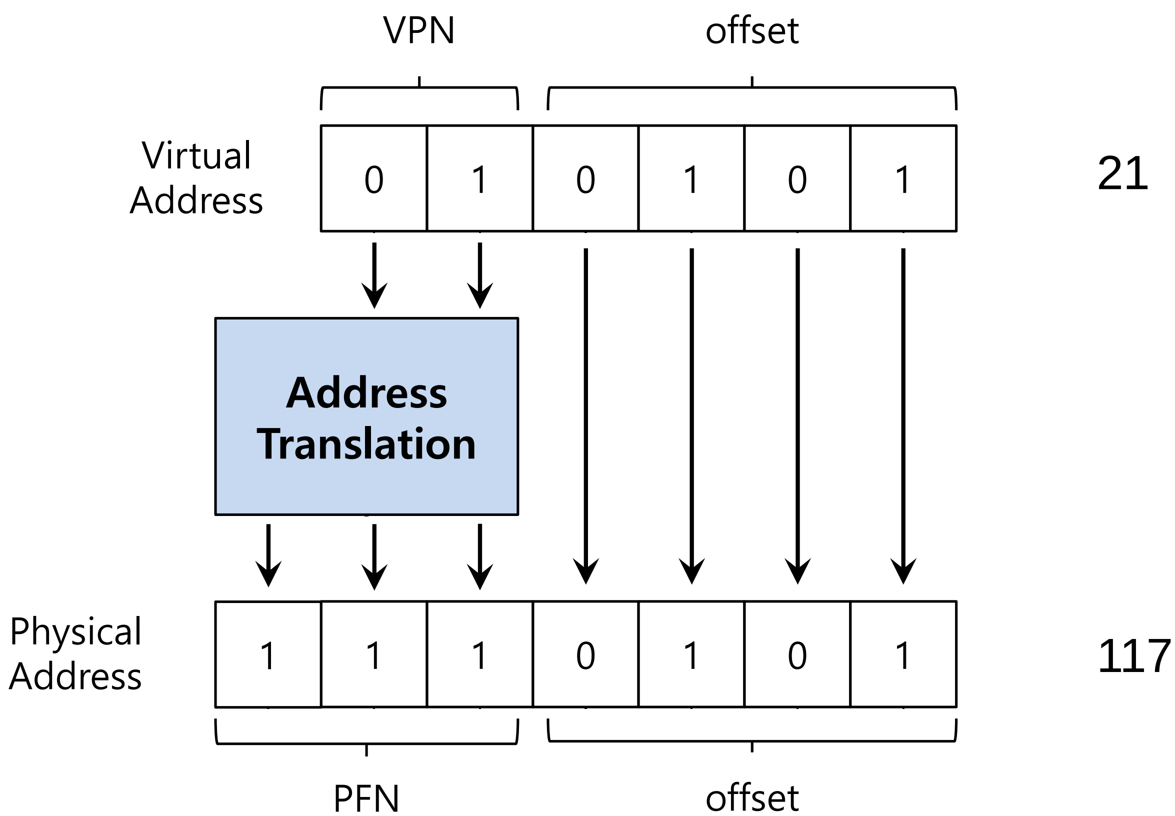
VPN		offset			
Va5	Va4	Va3	Va2	Va1	Va0

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

VPN		offset			
0	1	0	1	0	1

Example: Address Translation

- The virtual address 21 in 64-byte address space

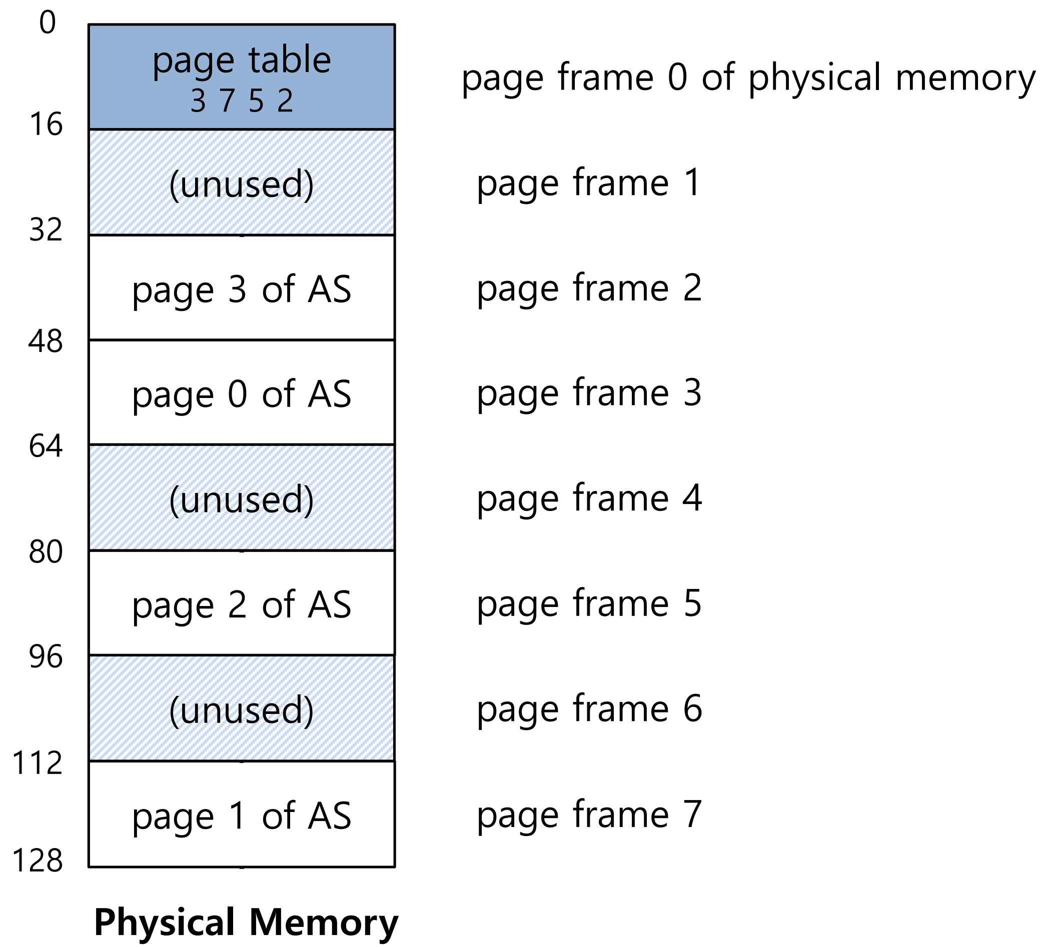


What Is In The Page Table?

- The page table is a **data structure** that is used to map the virtual address to physical address.
  - Simplest form: a linear page table, an array
- The OS **indexes** the array by VPN, and looks up the page-table entry.
- Page tables can get awfully large
  - 32-bit address space with 4-KB pages,
    - How many pages?
      - 2^{20}
    - bits for VPN?
      - 20 bits
    - if 4 bytes per pte, size of page table?
      - $4MB = 2^{20}\ entries\ * 4\ Bytes\ per\ page\ table\ entry$

Example: Page Table in Kernel Physical Memory

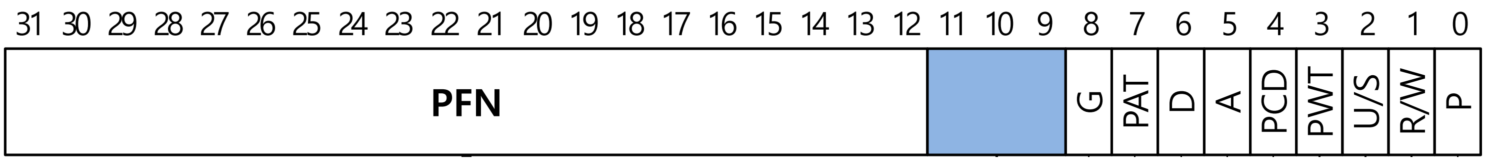
- Page tables for each process are stored in memory.



Common Flags Of Page Table Entry

- **Valid Bit**: Indicating whether the particular translation is valid.
  - unused space between heap and stack will be marked **invalid**
- **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: page frame number

Paging: Too Slow

- To find a location of the desired PTE, the **starting location** of the page table is **needed: page table base register (PTBR)**.

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

Accessing Memory With Paging

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

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

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

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

A Memory Trace

- Example: A Simple Memory Access

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

VA of array: 40000 through 44000-1

VPNs: 39-42

Virtual to physical mappings

(VPN 1 → PFN 4)
(VPN 39 → PFN 7),
(VPN 40 → PFN 8),
(VPN 41 → PFN 9),
(VPN 42 → PFN 10).

- Compile and execute

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

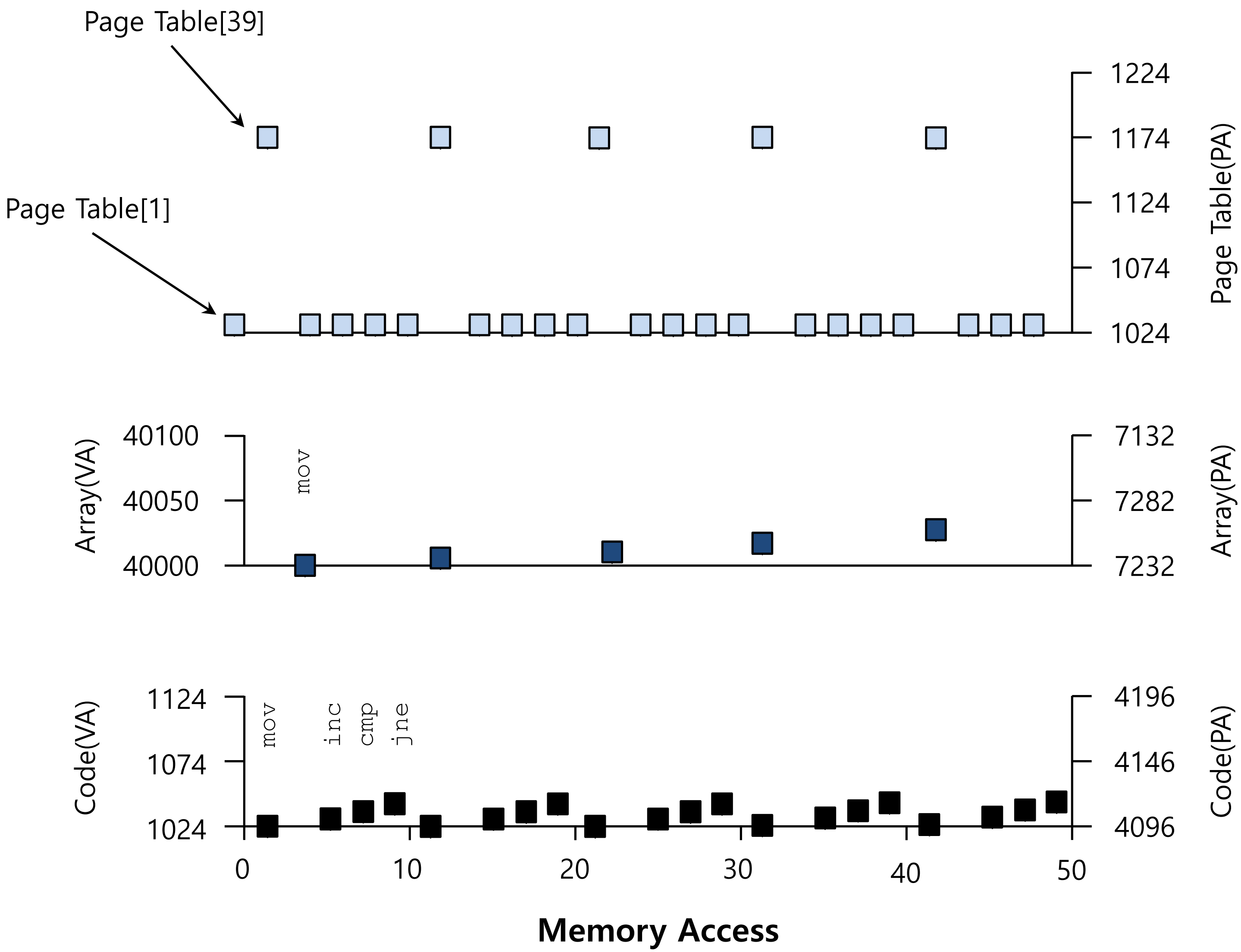
- Resulting Assembly code

Assume linear (array-based) page table and that it is located at physical address 1 KB (1024).

```
0x1024 movl $0x0,(%edi,%eax,4)//Val[edi + eax*4] <-0
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne 0x1024
```

edi contains base address of array and eax contains the array index

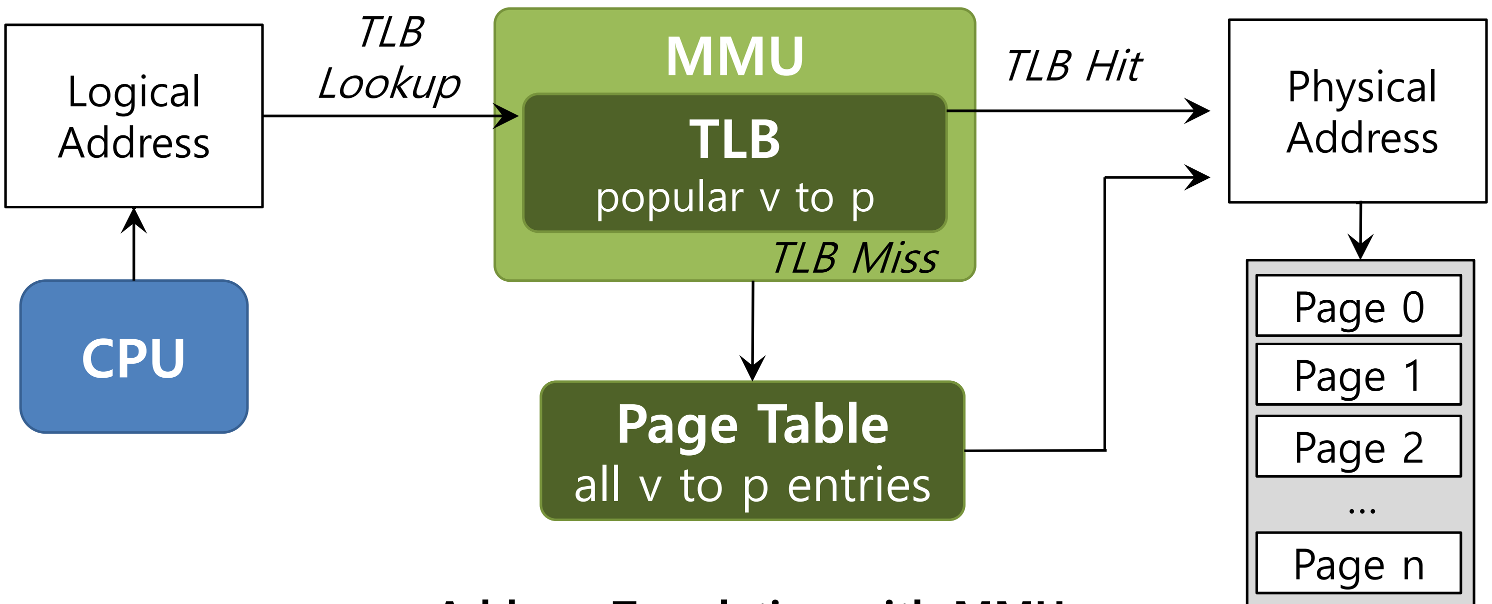
A Virtual(And Physical) Memory Trace



Translation Lookaside Buffers

TLB

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



Address Translation with MMU

Physical Memory

TLB Basic Algorithms

```
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 translation for this VPN.
- (5-8 lines) extract the page frame number from the relevant TLB entry, and form the desired physical address and access memory.

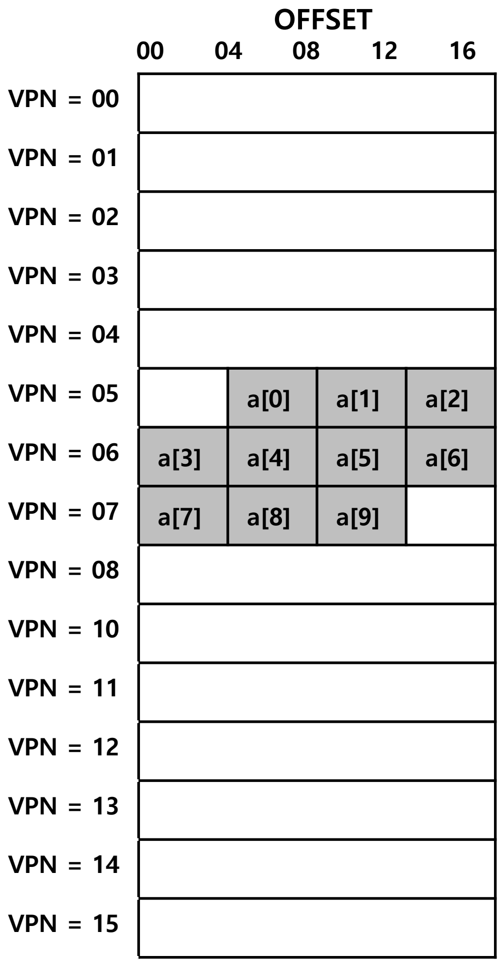
TLB Basic Algorithms (Hardware managed)

```
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

- How a TLB can improve its performance.



Address space

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

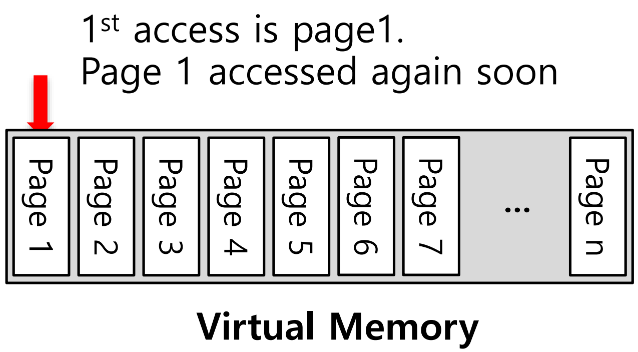
The TLB improves performance due to spatial locality

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

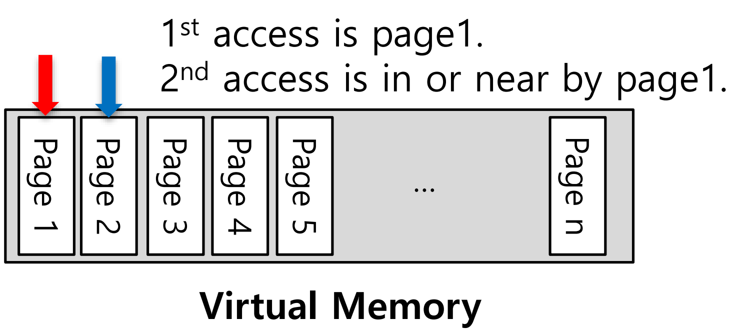
Size of address space?
base address of array?

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.
  - 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.**

- 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)

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

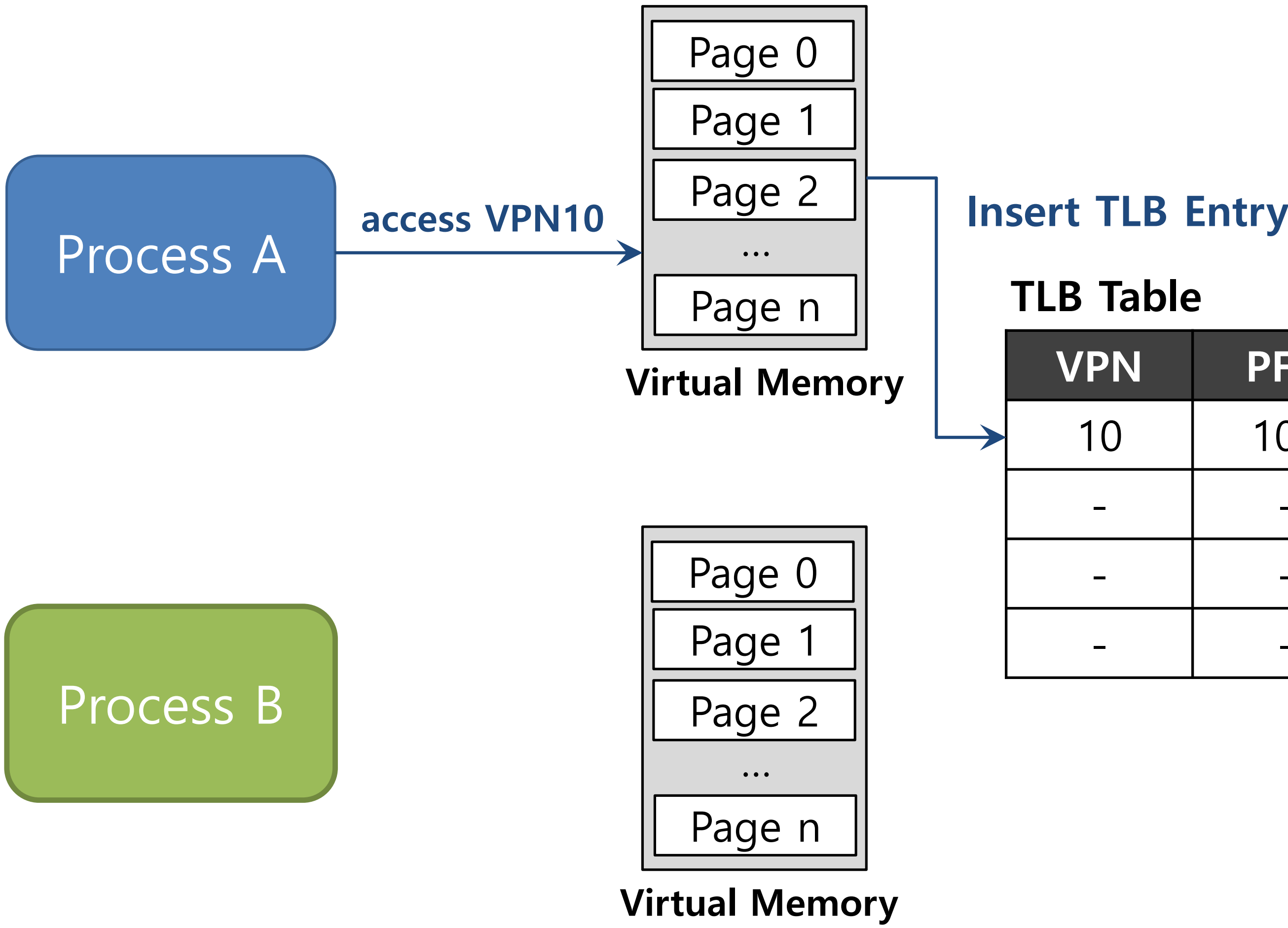
TLB entry

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

VPN	PFN	other bits

Typical TLB entry looks like this

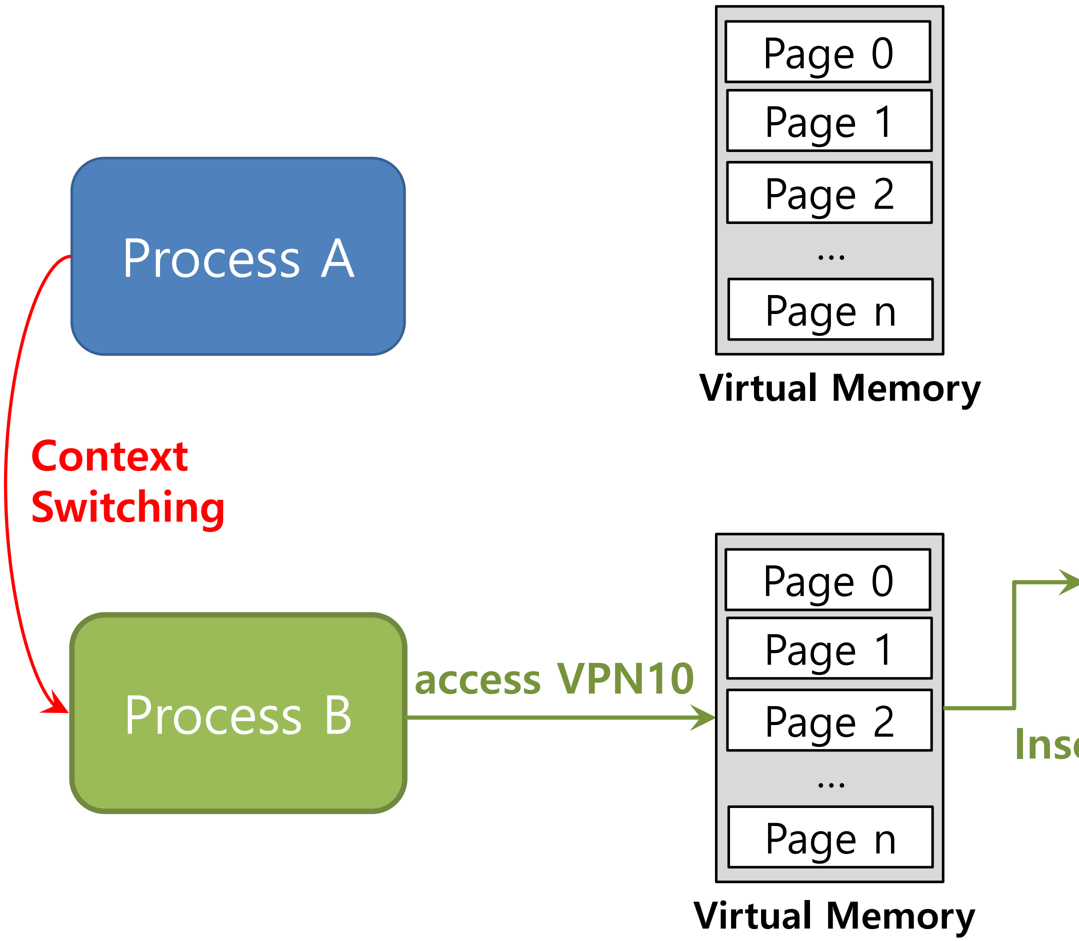
TLB Issue: Context Switching



TLB Table

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

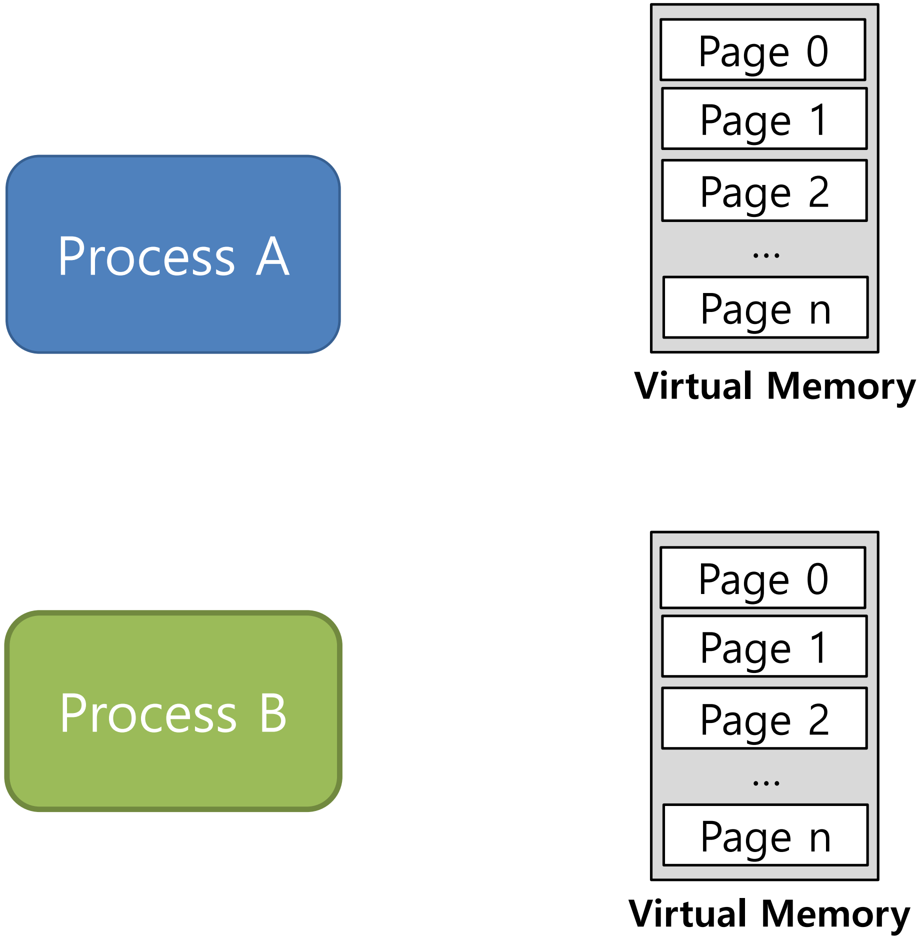
TLB Issue: Context Switching



TLB Table

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

TLB Issue: Context Switching



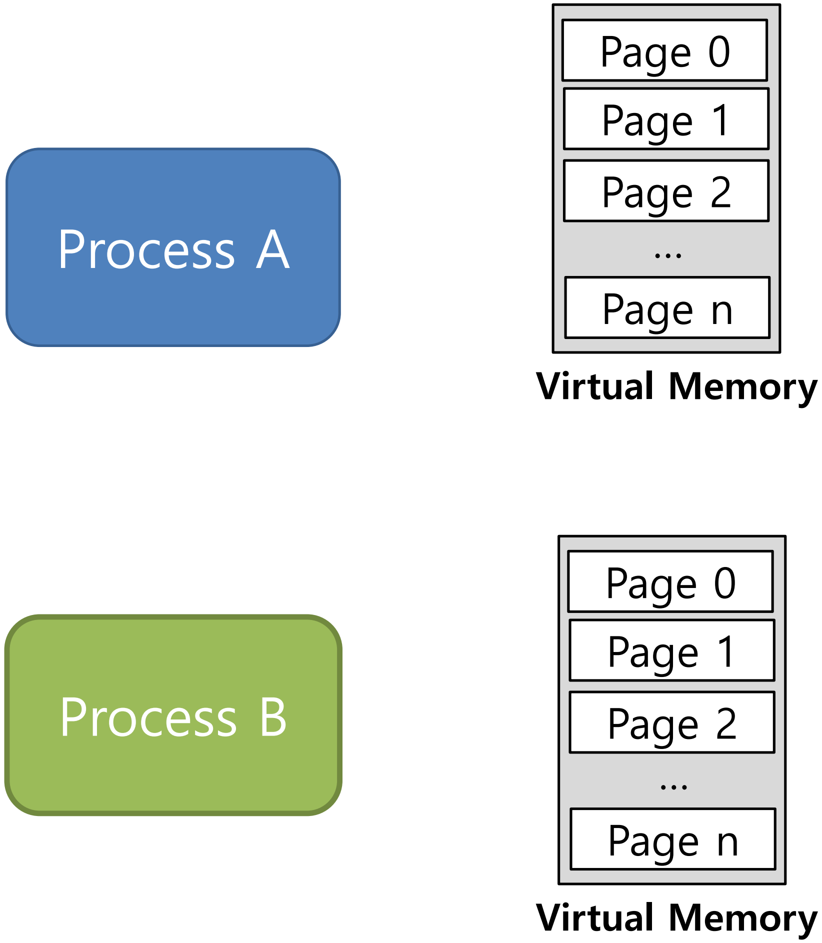
TLB Table

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.



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.

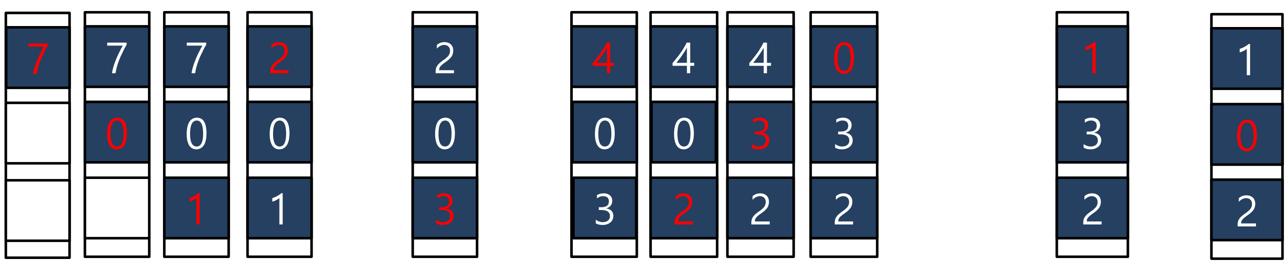
TLB Replacement Policy

- LRU (Least Recently Used)
  - Evict an entry that has not recently been used.
  - Take advantage of *locality* in the memory-reference stream.
  - Random: loop accessing n+1 pages, TLB of size n, LRU replacement policy

Reference Row

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1

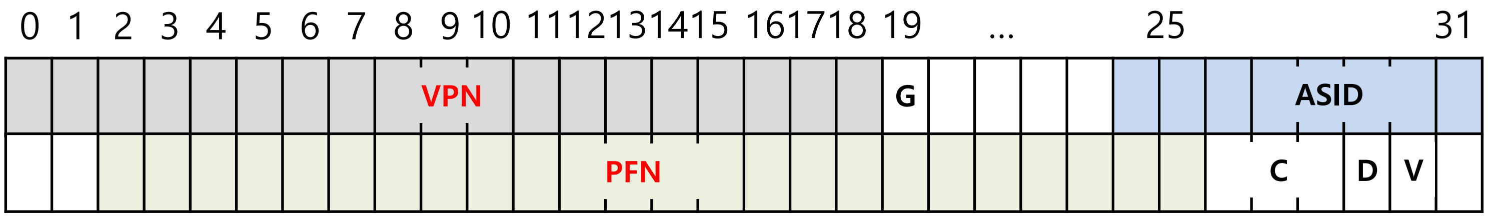
Page Frame:



Total 11 TLB miss

A Real TLB Entry

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



Flag	Content
19-bit VPN	User addresses from half of the address space.
24-bit PFN	Systems can support with up to 64GB of main memory.
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 if the page has been modified.
Valid bit(V)	tells the hardware if there is a valid translation present in the entry.

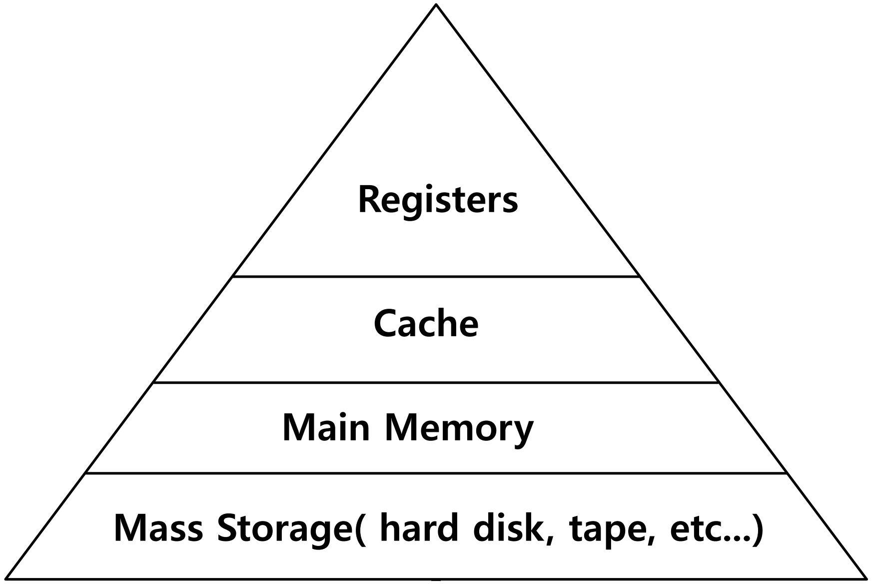
MIPS R4000 supports 32-bit address space with 4KB pages:
Virtual address: ? bit VPN and ? bit offset? 20 bit and 12 bit offset

How much physical memory supported in GB? $2^{24}\ 4KB\ pages$ = 64 GB

Swapping: Mechanisms

Beyond Physical Memory: Mechanisms

- Relaxing assumption: VAS smaller than physical memory
- 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

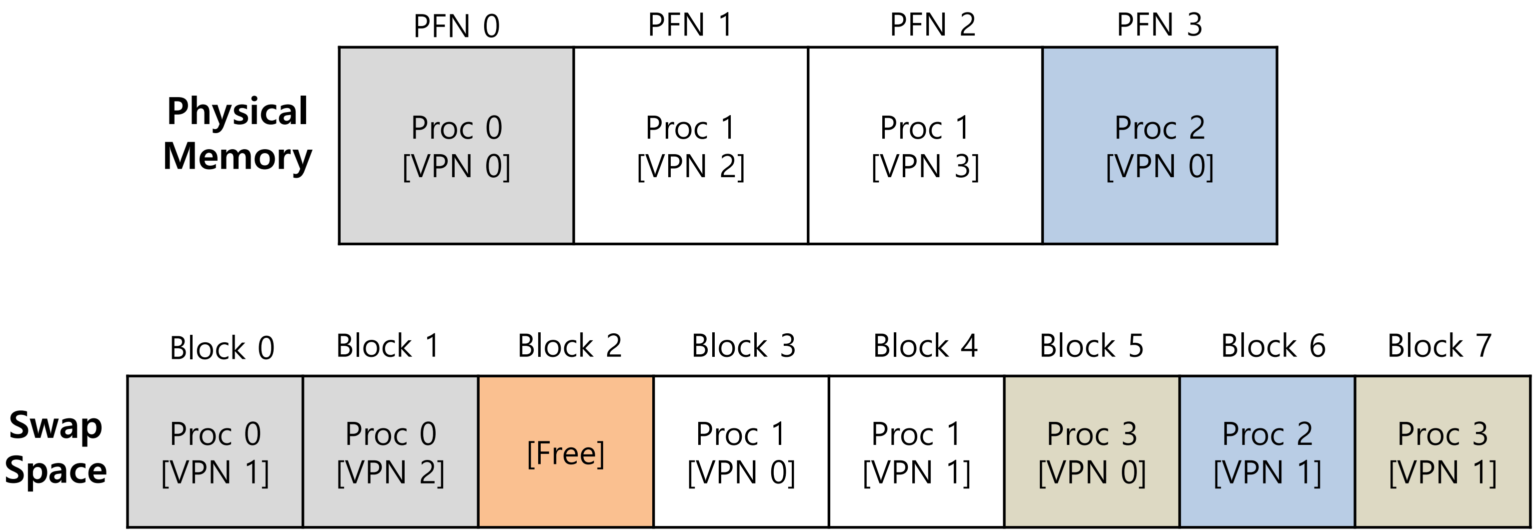
Single large address for a process

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

- 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 disk address to read from and write to the swap space, in page-sized units



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 ?

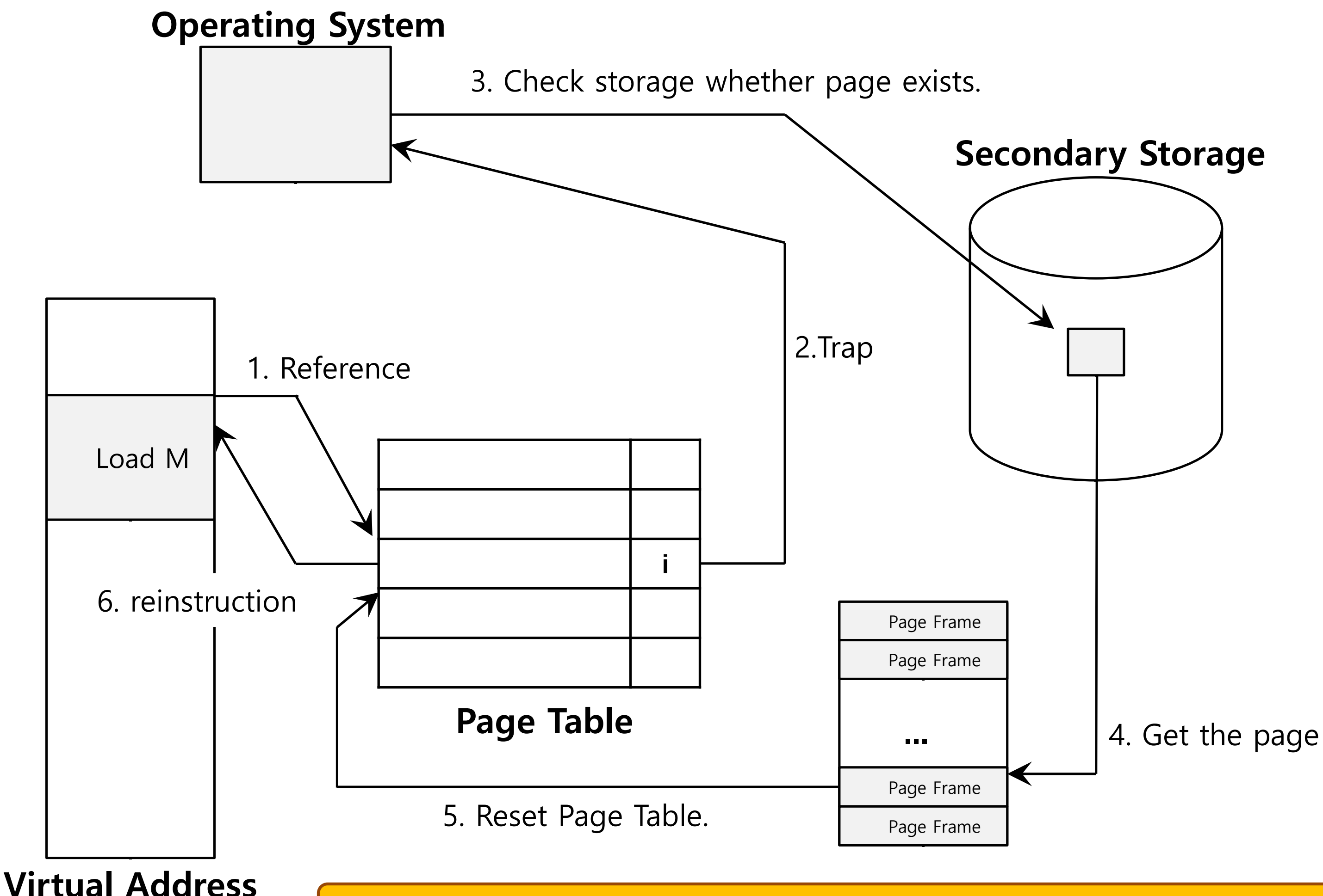
- The OS likes to page out pages to make room for the new pages the OS is about to bring in.
  - The process of picking a page to kick out, or replace is known as page-replacement policy

The Page Fault

- Accessing page that is not in physical memory.
  - If a page is not present and has been swapped to disk, the OS needs 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

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

Page Fault Control Flow – Hardware

```
else // TLB Miss
PTEAddr = PTBR + (VPN * sizeof(PTE))
PTE = AccessMemory(PTEAddr)
if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
else
if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
else if (PTE.Present == True)
// assuming hardware-managed TLB
        TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
        RetryInstruction()
else if (PTE.Present == False)
        RaiseException(PAGE_FAULT)
```

Page Fault Control Flow – Software

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

- 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.

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 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).*
- even a tiny miss rate will quickly dominate the overall AMAT of running programs.

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

Argument	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)

Assume T_M=100 ns and T_D = 10 ms,

hit rate= 90%, AMAT in ms?:
0.9 * 100ns + 0.1 * 10ms = 1.00009 ms, or about 1 millisecond.

hit rate=99.9%, AMAT:
0.999*100ns +0.001* 10ms = 0.0100999 ms(**roughly 100 times faster**).

hit rate= 100%,
AMAT approaches 0.0001 nanoseconds.

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

Assume cache fits 3 pages

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

Cold start miss or compulsory miss

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 head 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 $\frac{Hits}{Hits+Misses} = \mathbf{36.4\%}$

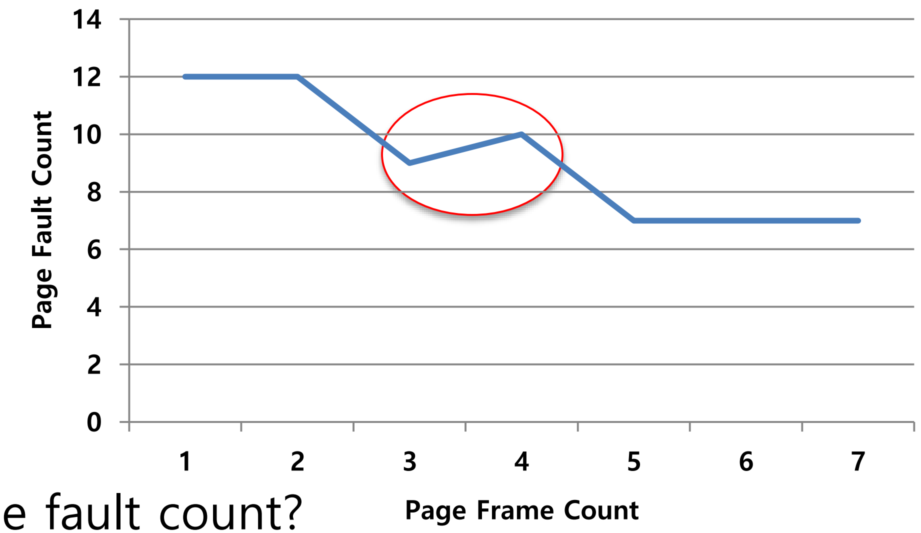
Even though page 0 had been accessed a number of times, FIFO still kicks it out.

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 when cache size increases from 3 to 4 pages.

Reference Row

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



- Cache size 3: page fault count?
- Cache size 4: page fault count?

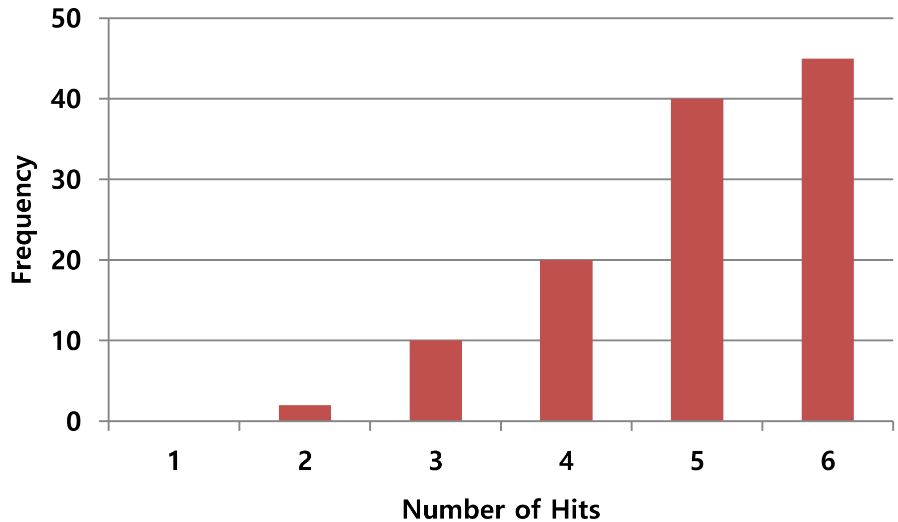
Another Simple Policy: Random

- 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 <u>Random</u> 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, but depends on luck.



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.

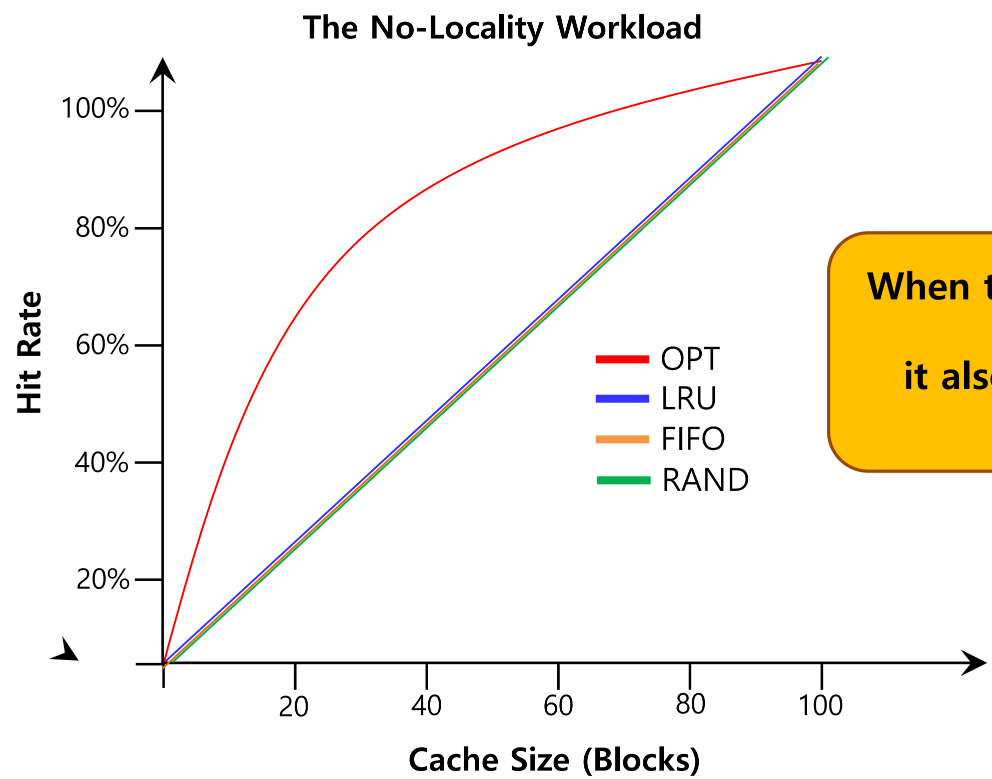
Reference Row

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

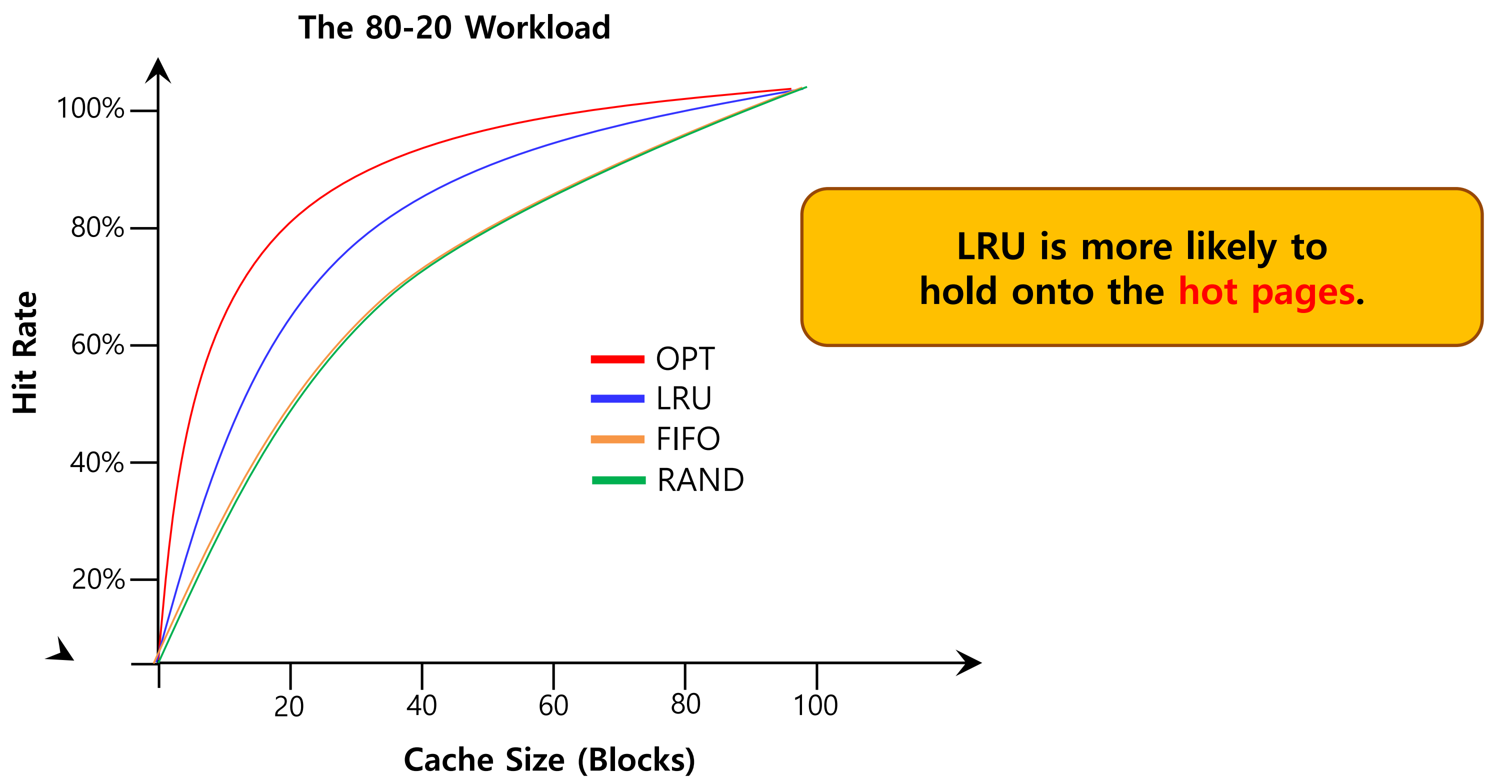
- Each reference is to a random page within the set of accessed pages.
  - Workload accesses 100 unique pages over time.
  - Choosing the next page to refer to at random



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

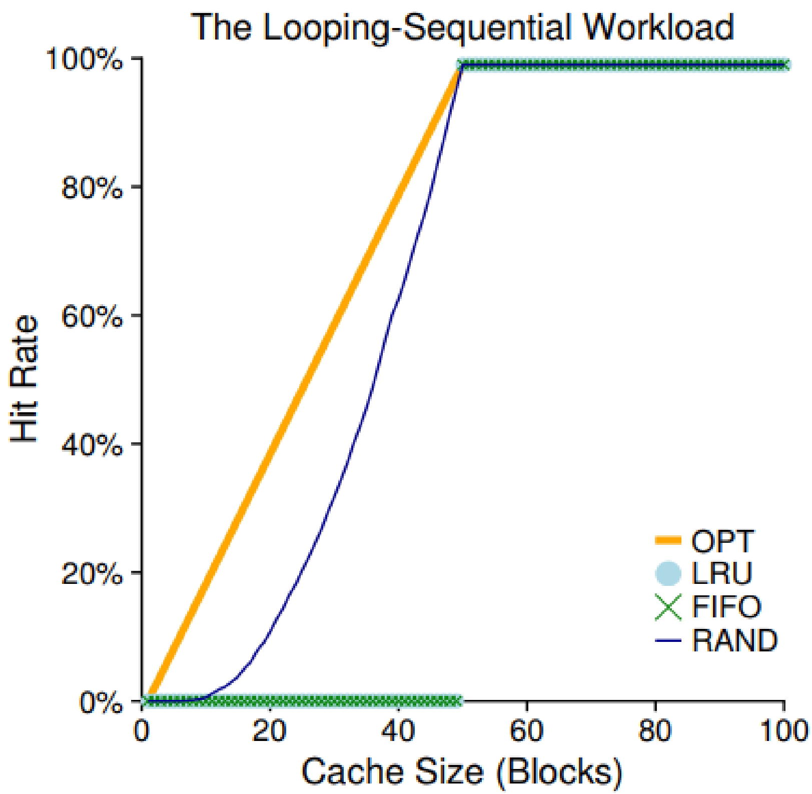
Workload Example : The 80-20 Workload

- 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.



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.

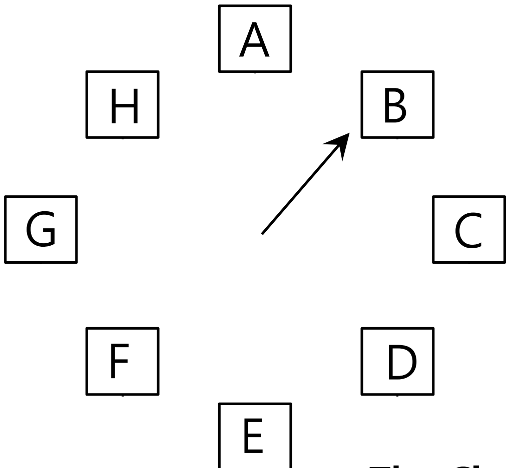
Approximating LRU

- 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.



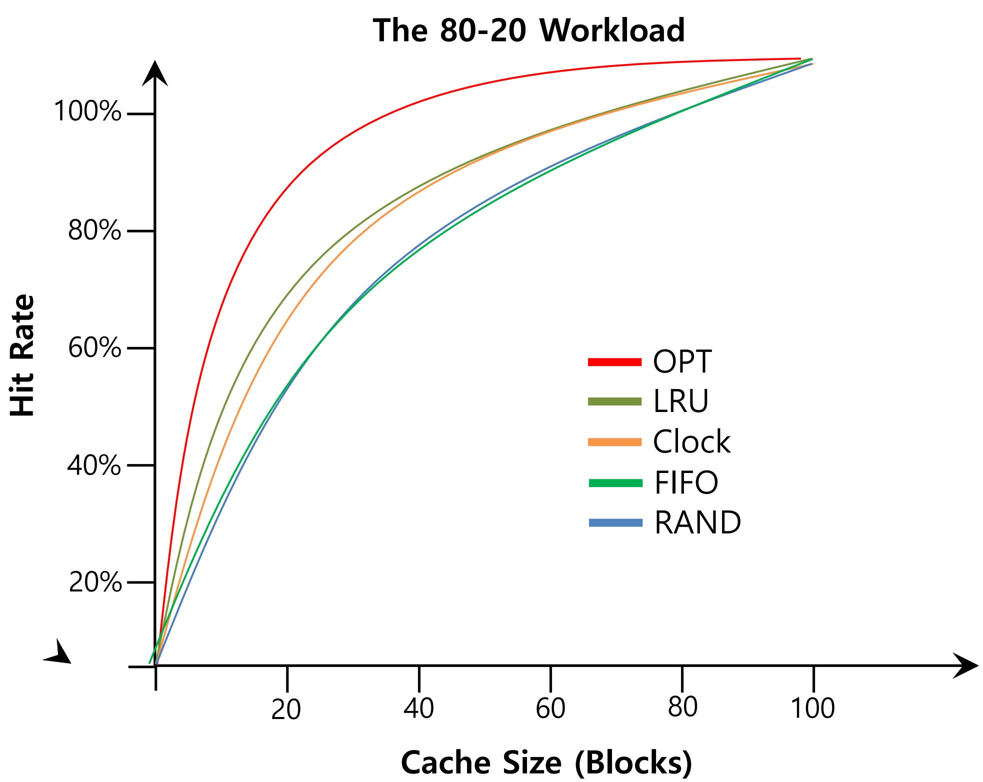
Use bit	Meaning
0	Evict the page
1	Clear **Use bit** and advance hand

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 than approach that don't consider history at all.



Considering Dirty Pages

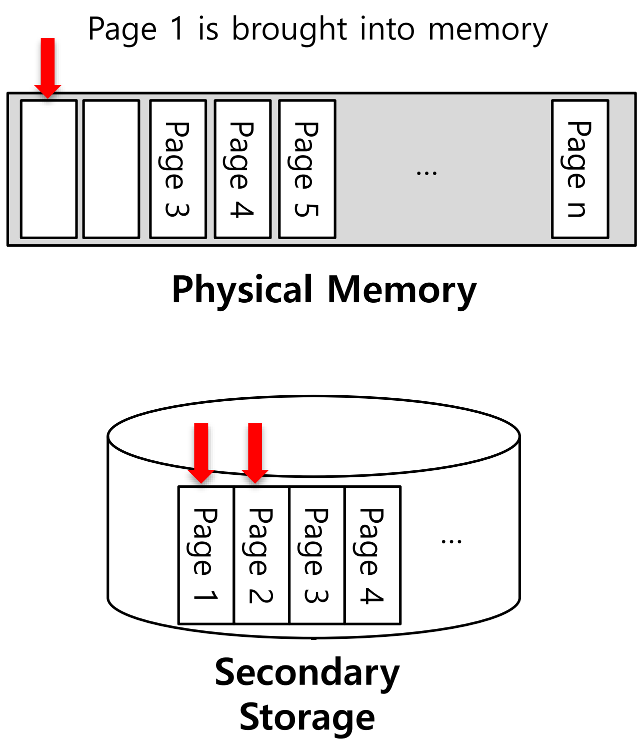
- The hardware include a **modified bit** (a.k.a **dirty bit**)
  - Page has been **modified** and is thus **dirty**, it must be written back to disk to evict it.
  - Page has not been modified, the eviction is free.

Page Selection Policy

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

Prefetching

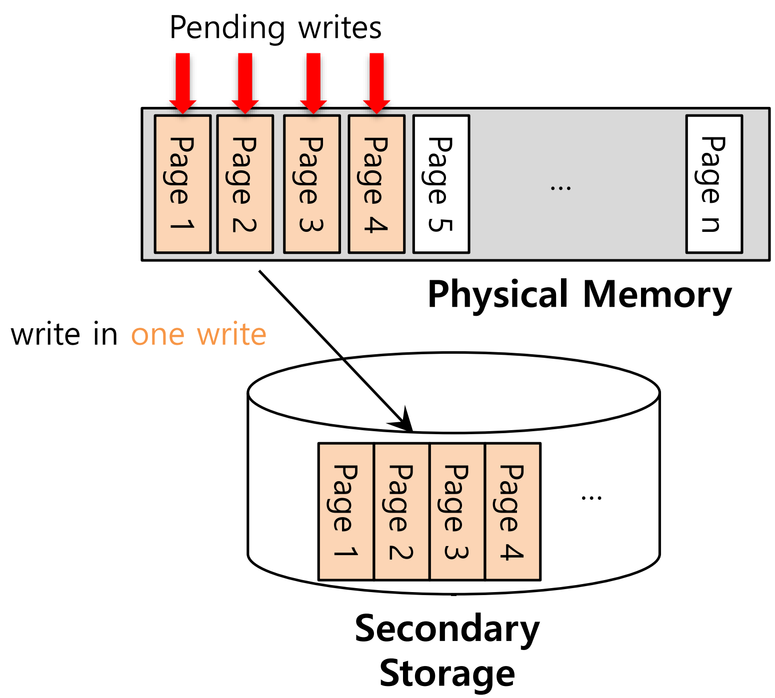
- The OS guess that a page is about to be used, and thus bring it in ahead of time.



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.