CS 5600 Computer Systems

Spring 2026

Virtualization: Memory
Lecture 4.1
February 3, 2026

Prof. Scott Valcourt

s.valcourt@northeastern.edu

603-380-2860 (cell)

Portland, ME

The Roux Institute
Northeastern University

Memory Virtualization

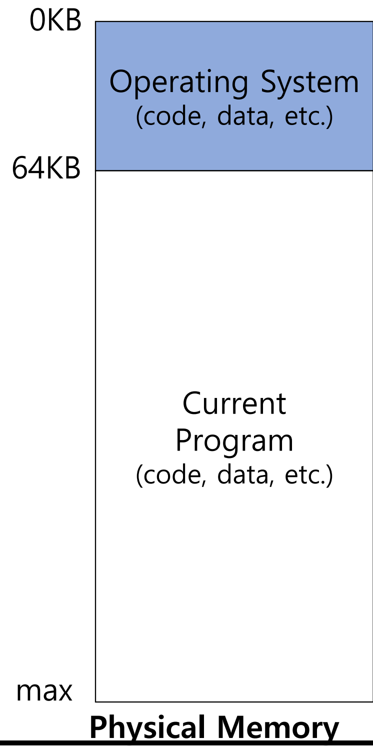
- What is **memory virtualization**?
  - OS virtualizes its physical memory.
  - OS provides an illusion memory space per each process.
  - It seems like each process uses the whole memory.

Benefits of Memory Virtualization

- Ease of use in programming
- Memory efficiency in terms of time and space
- The guarantee of isolation for processes as well as OS
  - Protection from errant accesses of other processes

OS in the Early System Years

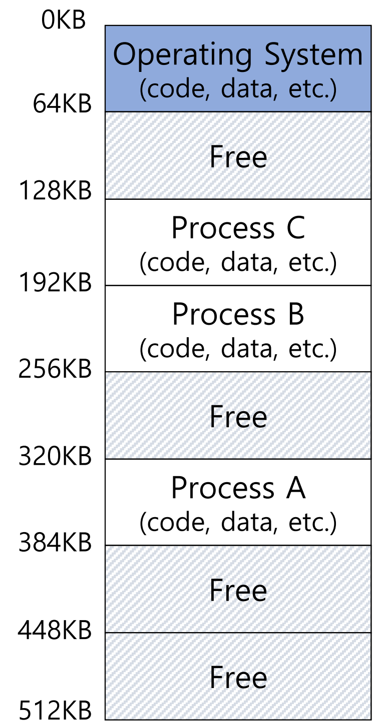
- Load only one process in memory.
  - Poor utilization and efficiency



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.

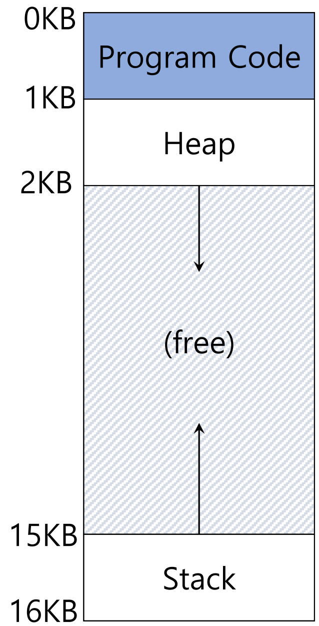
- Causes an important **protection issue**.
  - Errant memory accesses from other processes



Physical Memory

Address Space

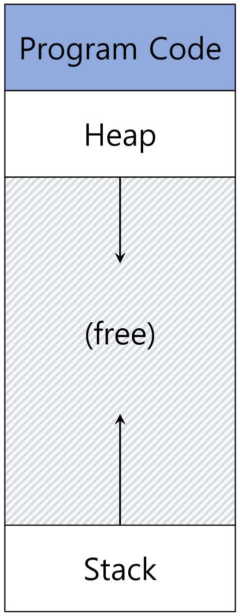
- OS creates an **abstraction** of physical memory.
  - Address space contains everything about a running process.
  - That is program code, heap, stack, etc.



Address Space

Address Space (Cont.)

- 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 and arguments to routines.



Address Space

Virtual Address

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

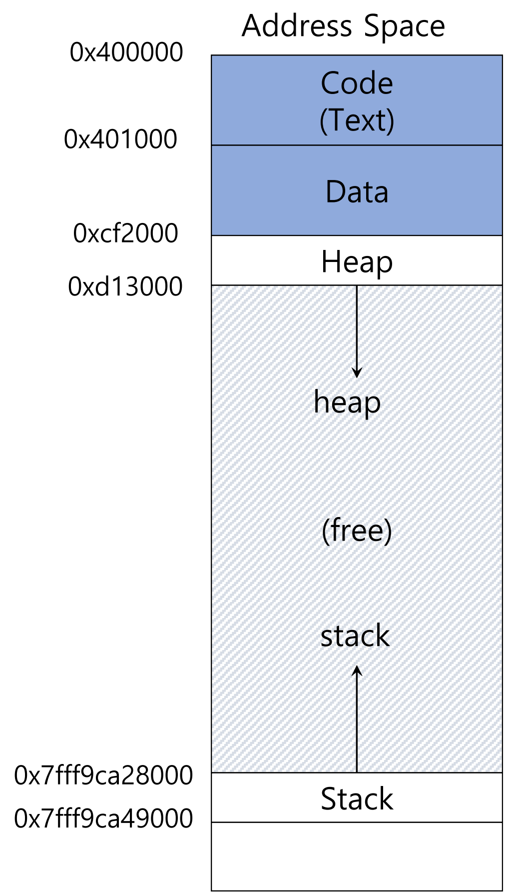
```
#include <stdio.h>
#include <stdlib.h>

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;
    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 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 API: `sizeof()`

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

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

```
4
```

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

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

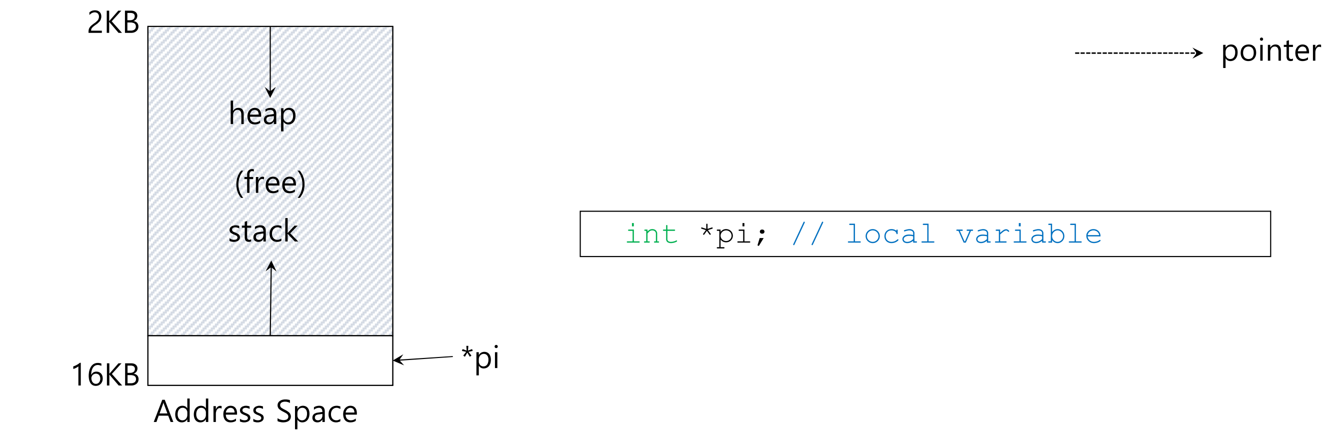
```
40
```

Memory API: `free()`

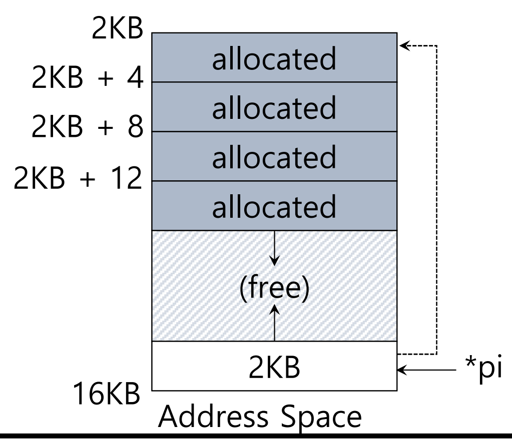
```
#include <stdlib.h>

void free(void* ptr)
```

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

Memory Allocating

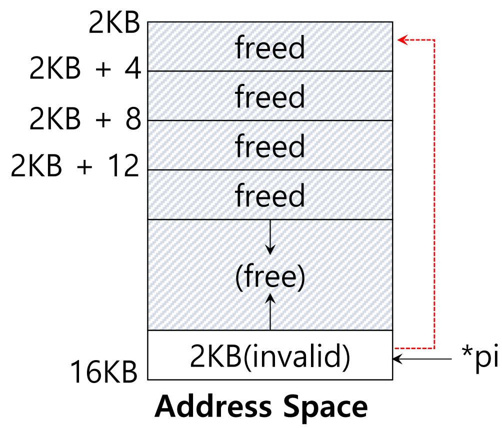



```
int *pi; // local variable
```

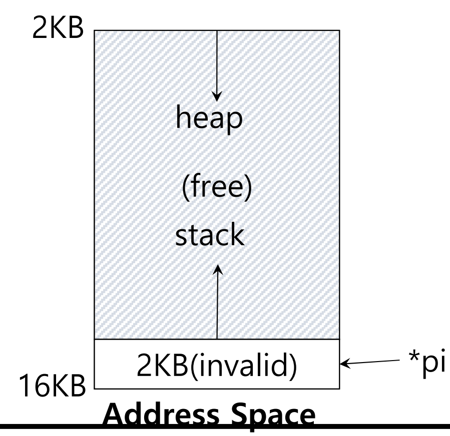



```
pi = (int *)malloc(sizeof(int)* 4);
```

Memory Freeing



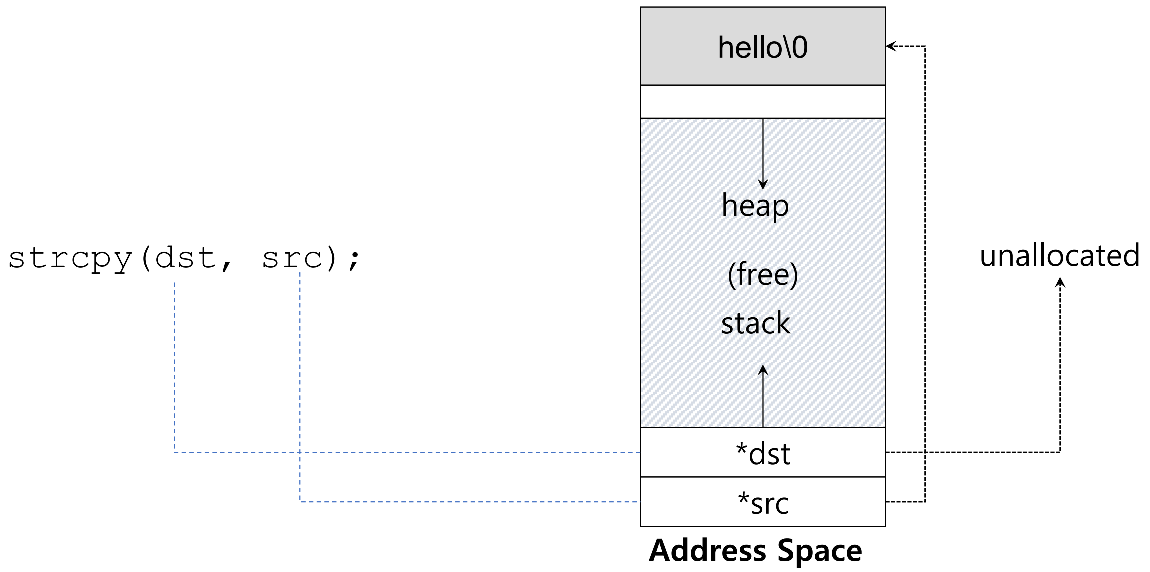

```
free(pi);
```



Forgetting To Allocate Memory

- Incorrect code

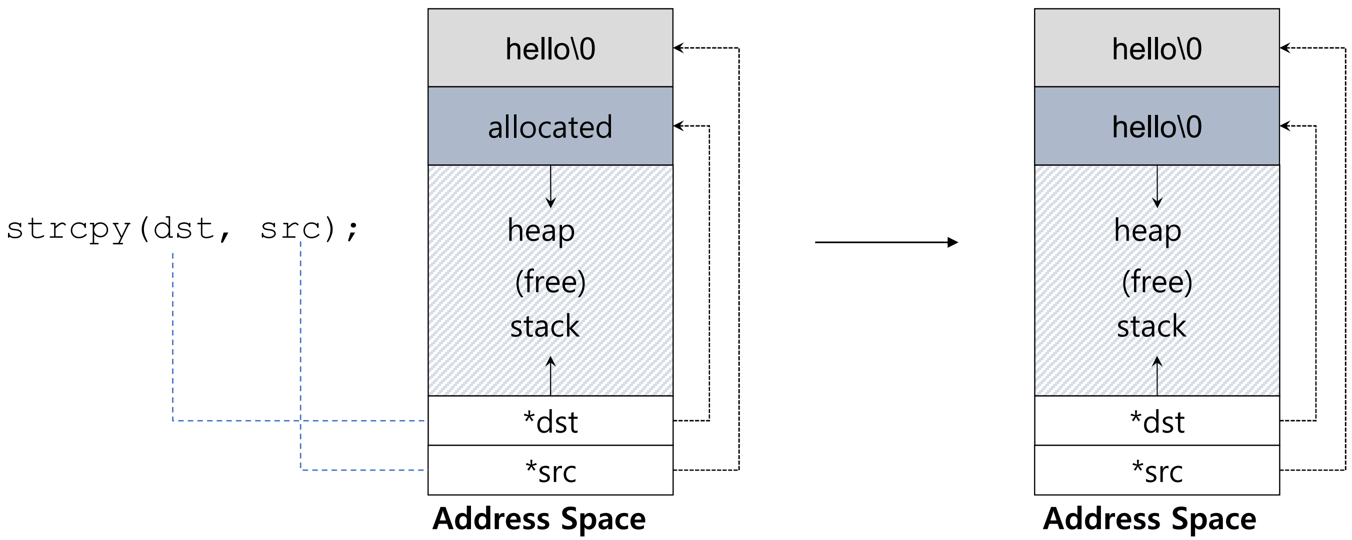
```
char *src = "hello"; //character string constant
char *dst;            //unallocated
strcpy(dst, src);    //segfault and die
```



Forgetting To Allocate Memory(Cont.)

- Correct code

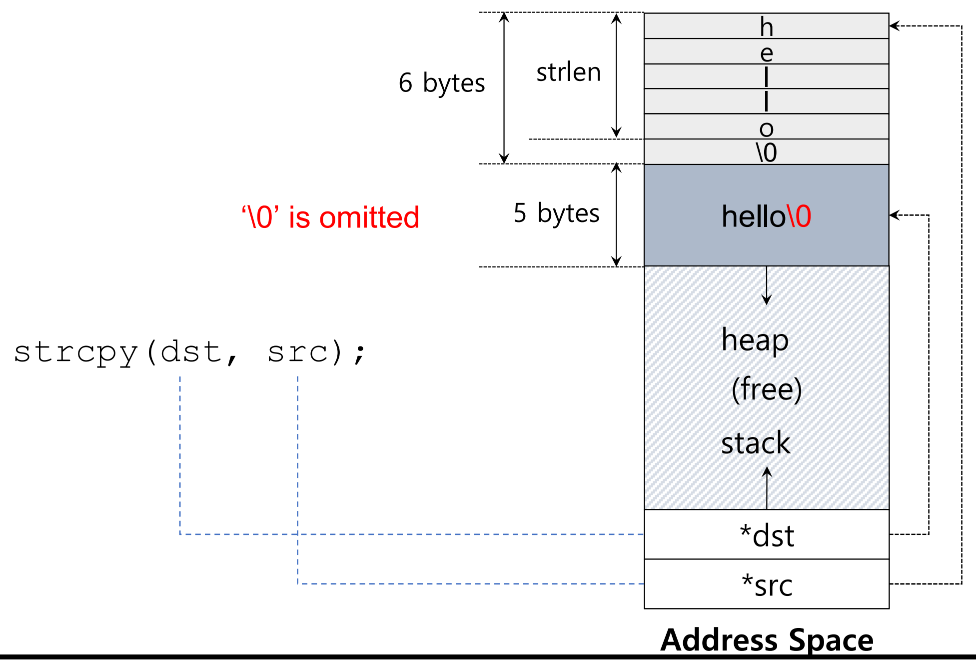
```
char *src = "hello";   //character string constant
char *dst (char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);      //work properly
```



Not Allocating Enough Memory

- Incorrect code, but works properly

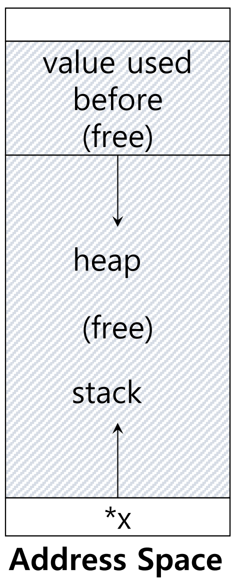
```
char *src = "hello"; //character string constant
char *dst (char *)malloc(strlen(src)); // too small
strcpy(dst, src);     //work properly
```

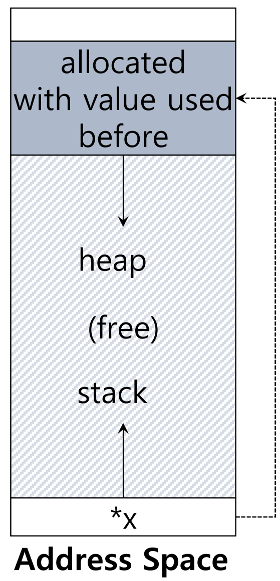


Forgetting to Initialize

- Encounter an uninitialized read

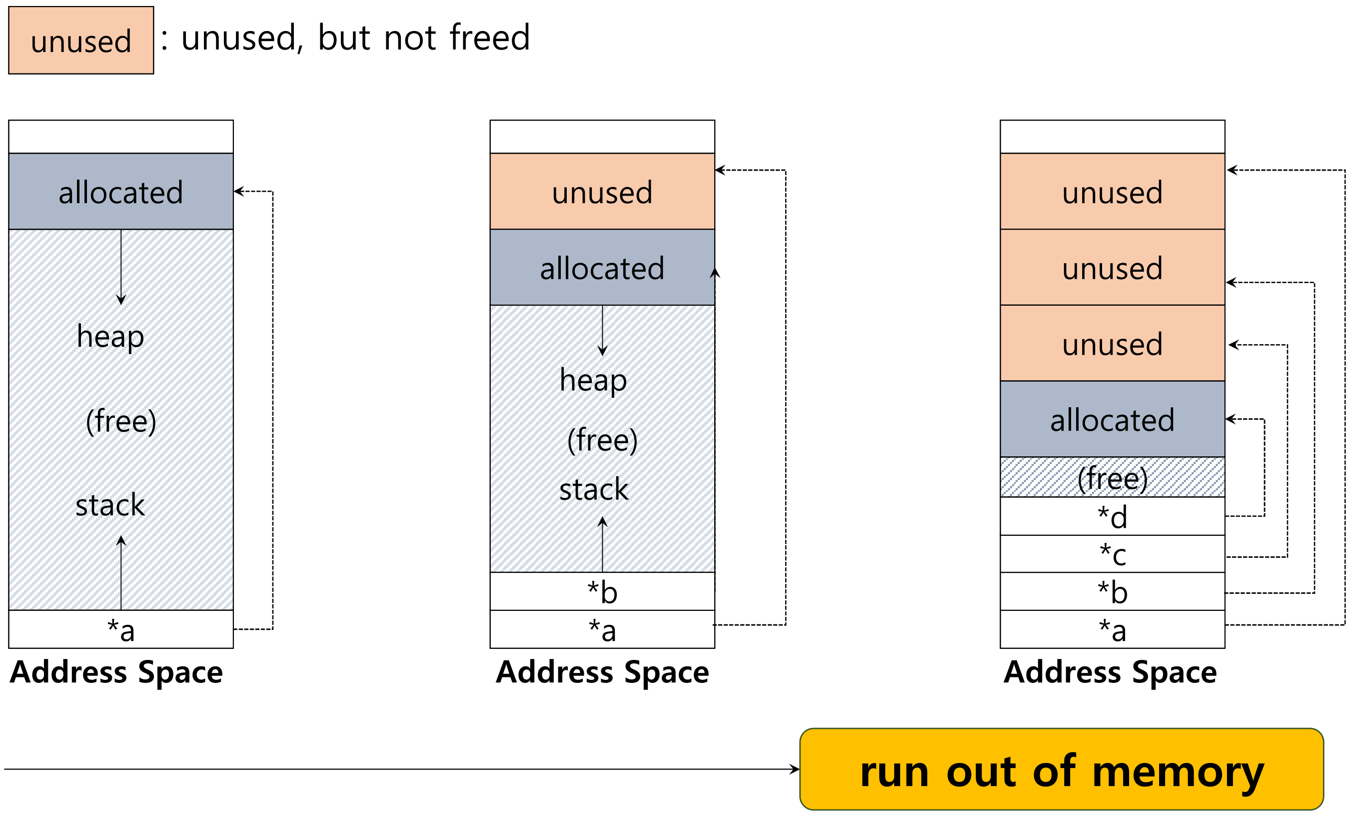
```
int *x = (int *)malloc(sizeof(int)); // allocated
printf("*x = %d\n", *x);             // uninitialized memory access
```





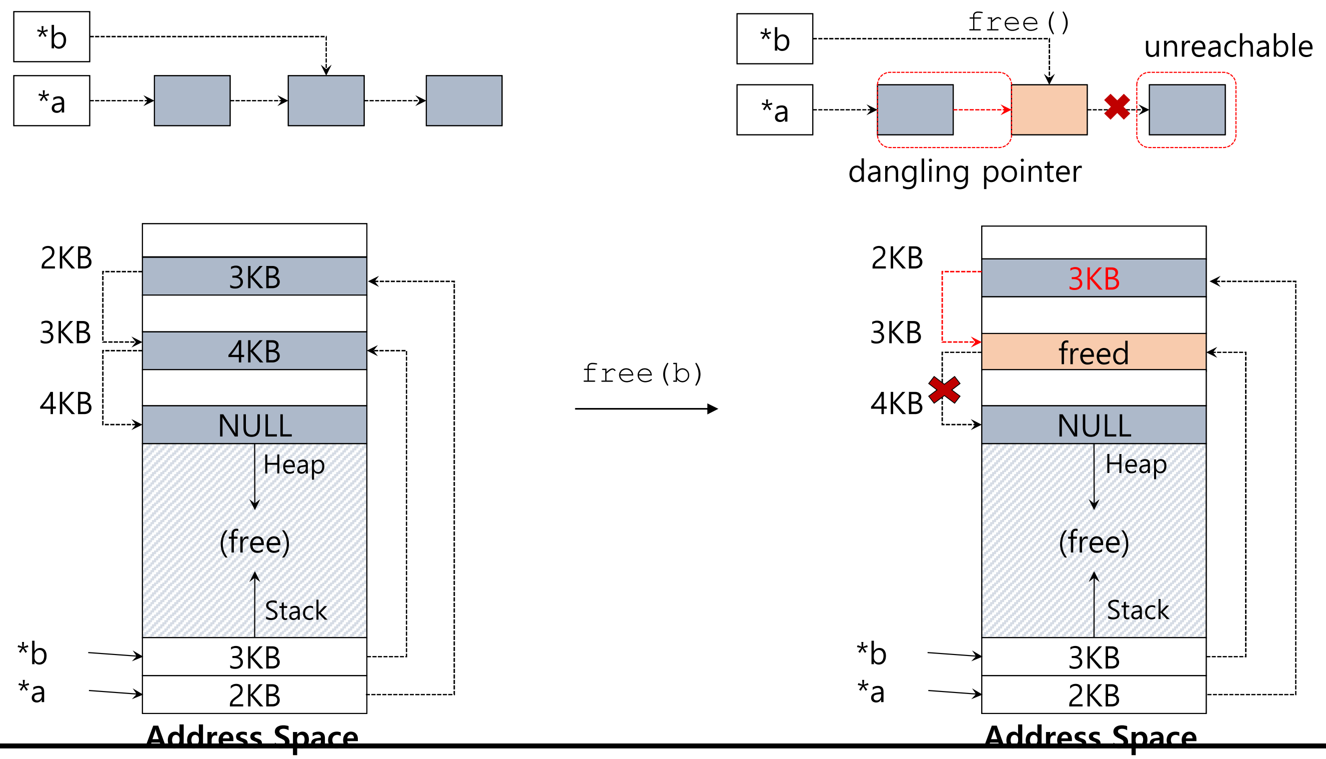
Memory Leak

- A program runs out of memory and eventually dies.



Dangling Pointer

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



Other Memory APIs: `calloc()`

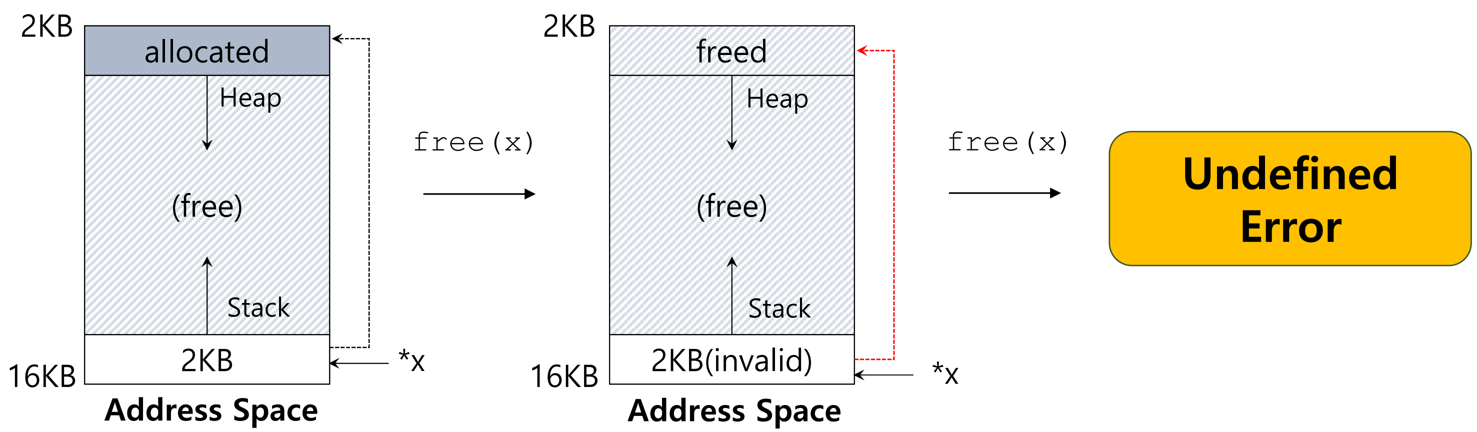
```
#include <stdlib.h>

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

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

Double Free

- Free memory that was freed already.

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



Other Memory APIs: `realloc()`

```
#include <stdlib.h>

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

- Change the size of memory block.
  - A pointer returned by `realloc` may be either the same as `ptr` or a new one.
  - 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: a `NULL` pointer

System Calls

```
#include <unistd.h>

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

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

System Calls (Cont.)

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

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

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

Quiz Time! Match that Address Location

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

Possible segments: static data, code, stack, heap

Address	Location
x	Static data → Code
main	Code
y	Stack
z	Stack
*z	Heap

What if no static data segment exists?

Break

Lecture Agenda

- Chapter 15 – Address Translation
- Chapter 16 – Segmentation
- Chapter 17 – Free Space Management
- Chapter 18 – Paging
- Chapter 19 – Translation Lookaside Buffers
- Chapter 20 – Paging: Smaller Tables
- Chapter 21 – Swapping: Mechanisms
- Chapter 22 – Swaping: Policies

Memory Virtualizing with Efficiency and Control

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

Address Translation

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

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

Example: Address Translation

- C - Language code

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

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

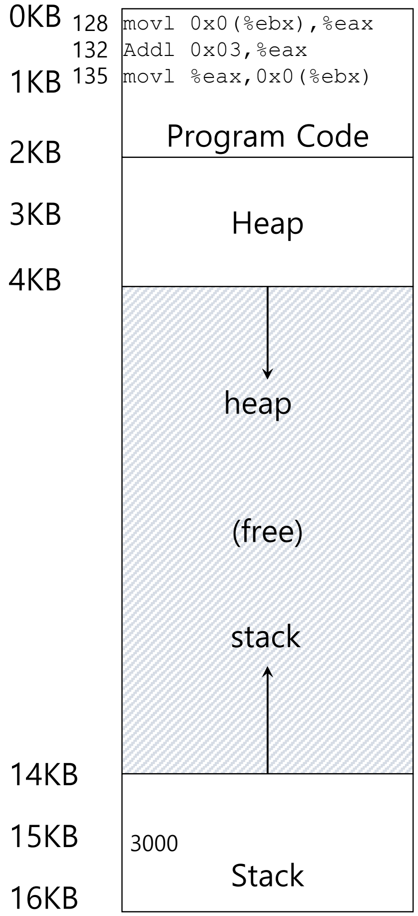
Example: Address Translation (Cont.)

- Assembly

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

- 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.

Example: Address Translation (Cont.)

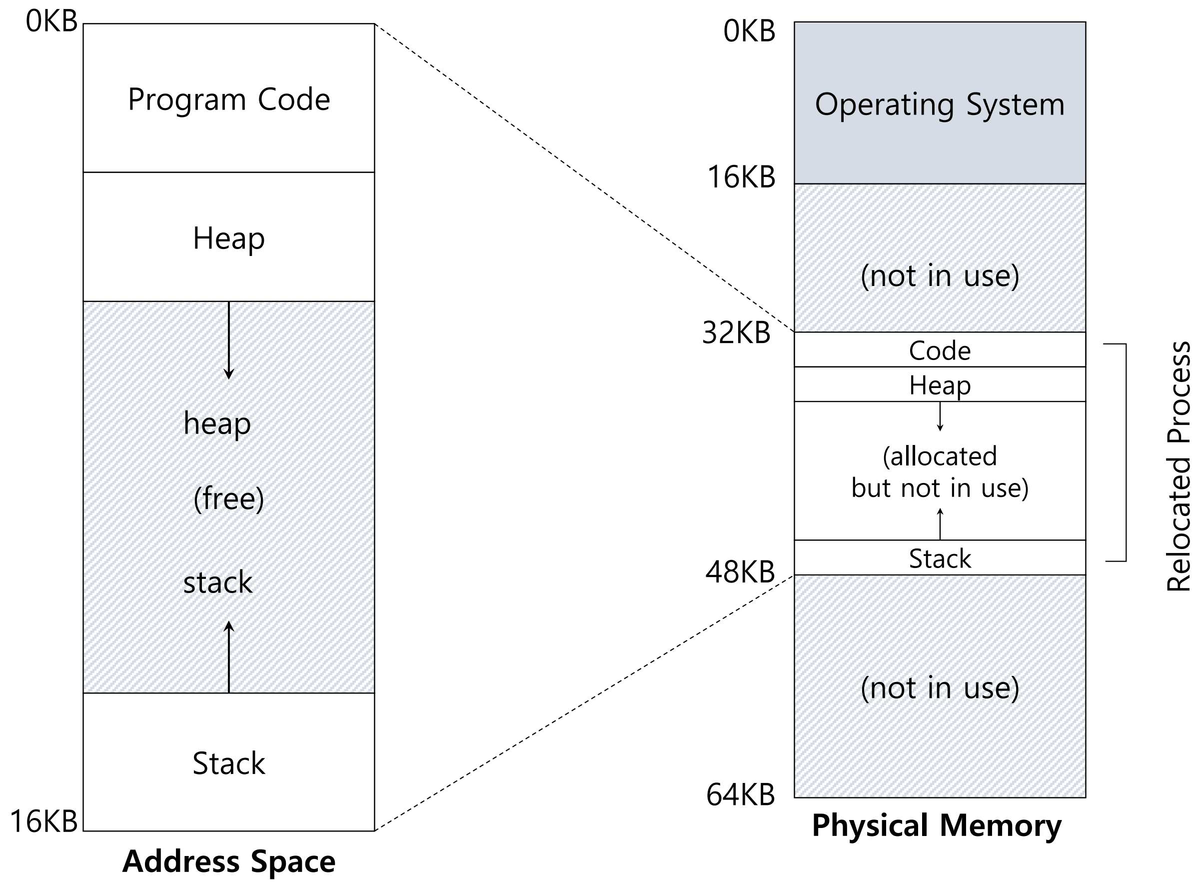


- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

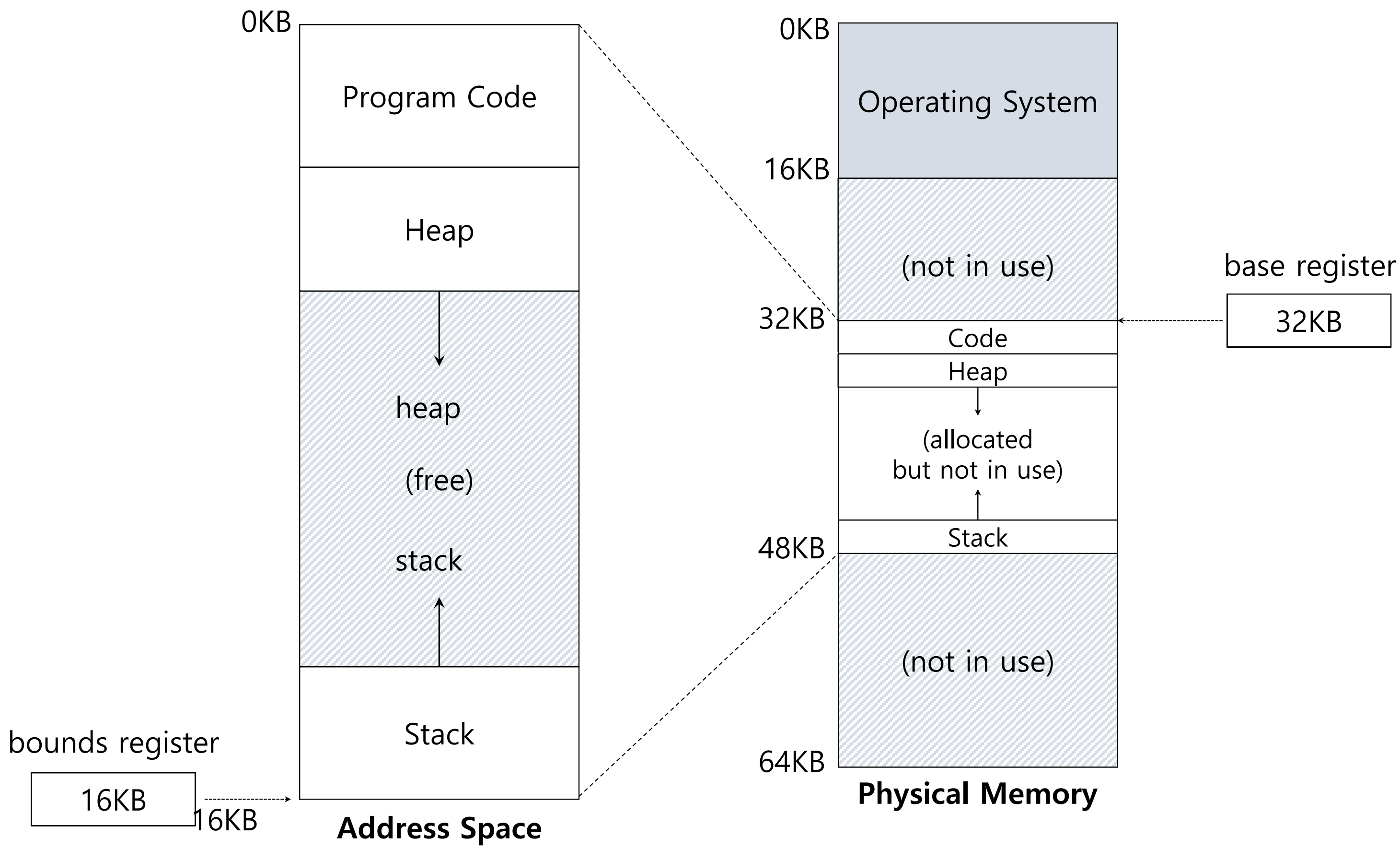
Relocation Address Space

- 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 to a value.

$$physical\ address = virtual\ address + base$$

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

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

Relocation and Address Translation

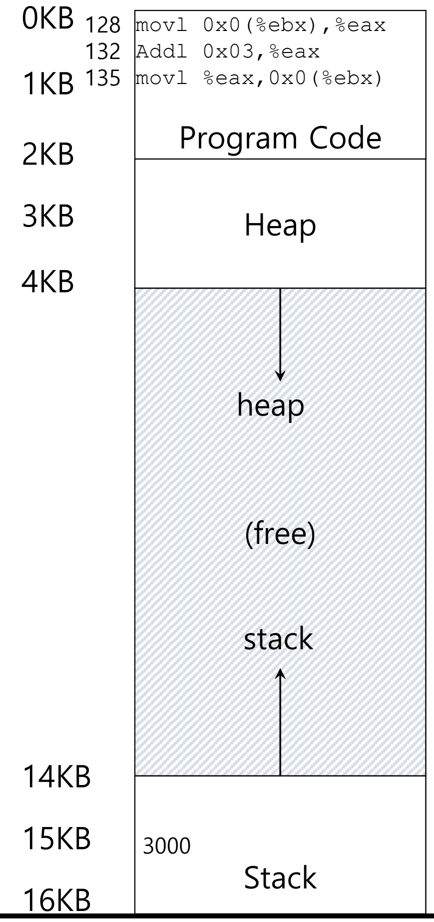
```
128 : movl 0x0(%ebx), %eax
```

- **Fetch** instruction at address 128

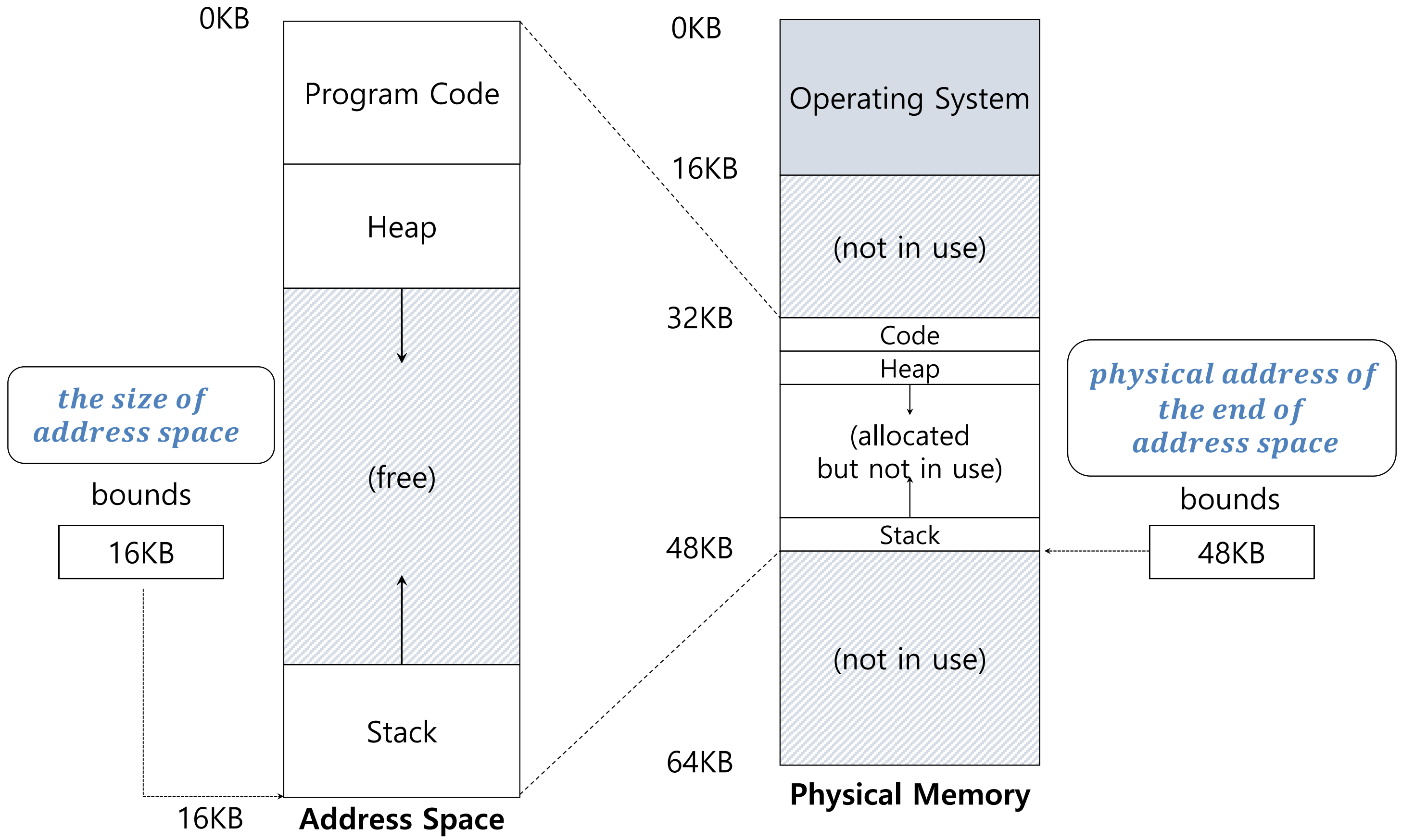
$$32896 = 128 + 32KB(base)$$

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

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



Two Ways of Bounds Register

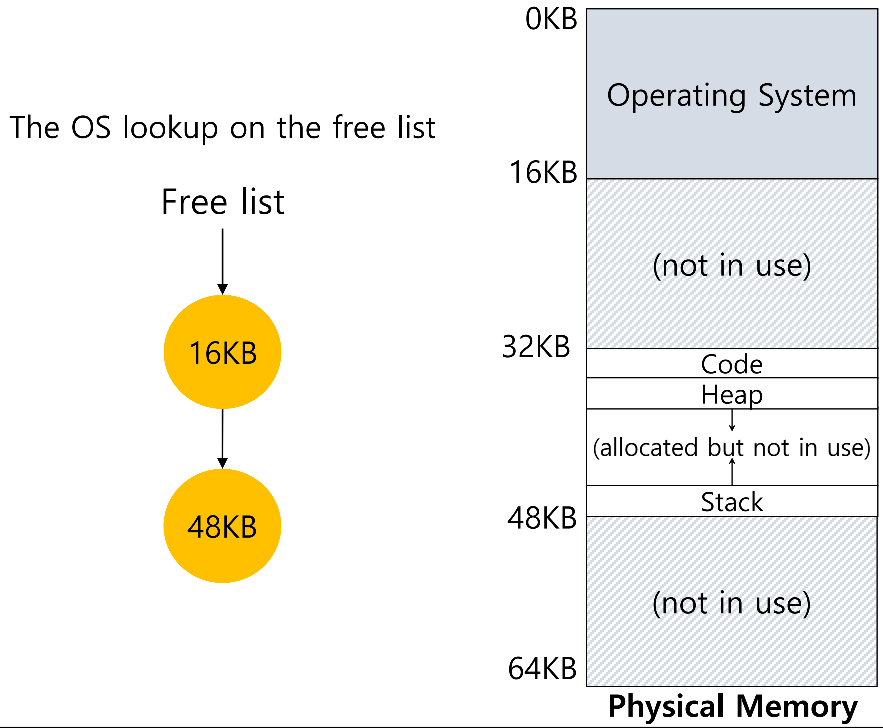


OS Issues for Memory Virtualizing

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

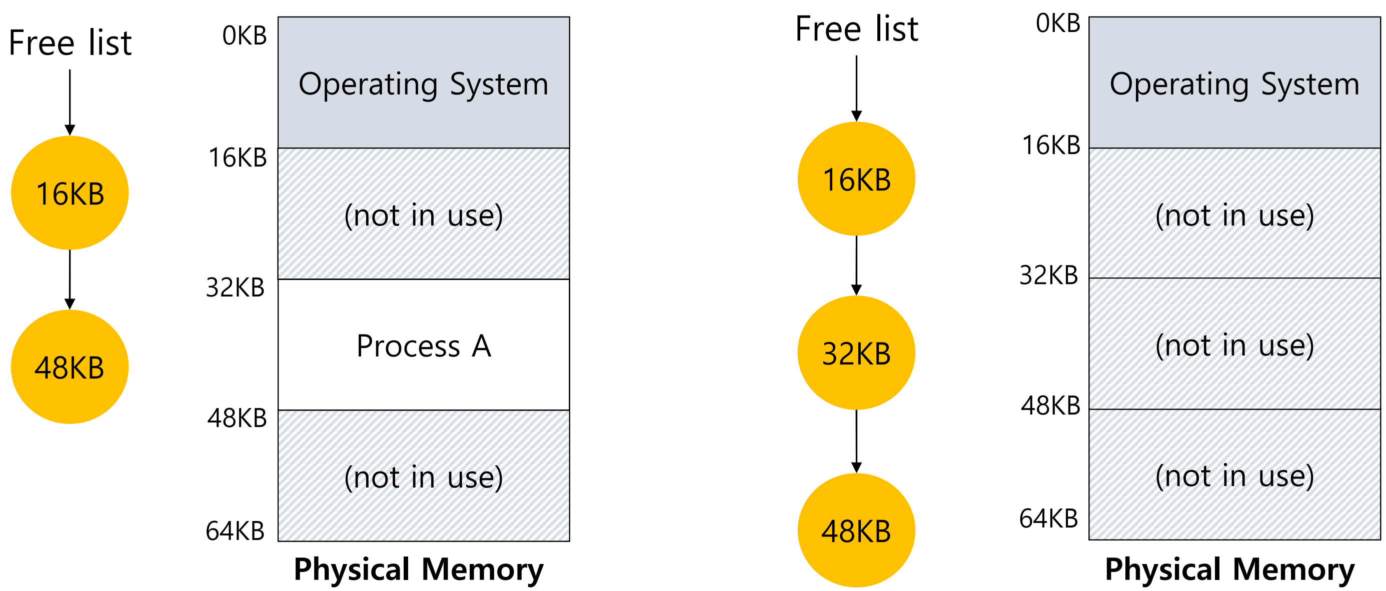
OS Issues: When a Process Starts Running

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



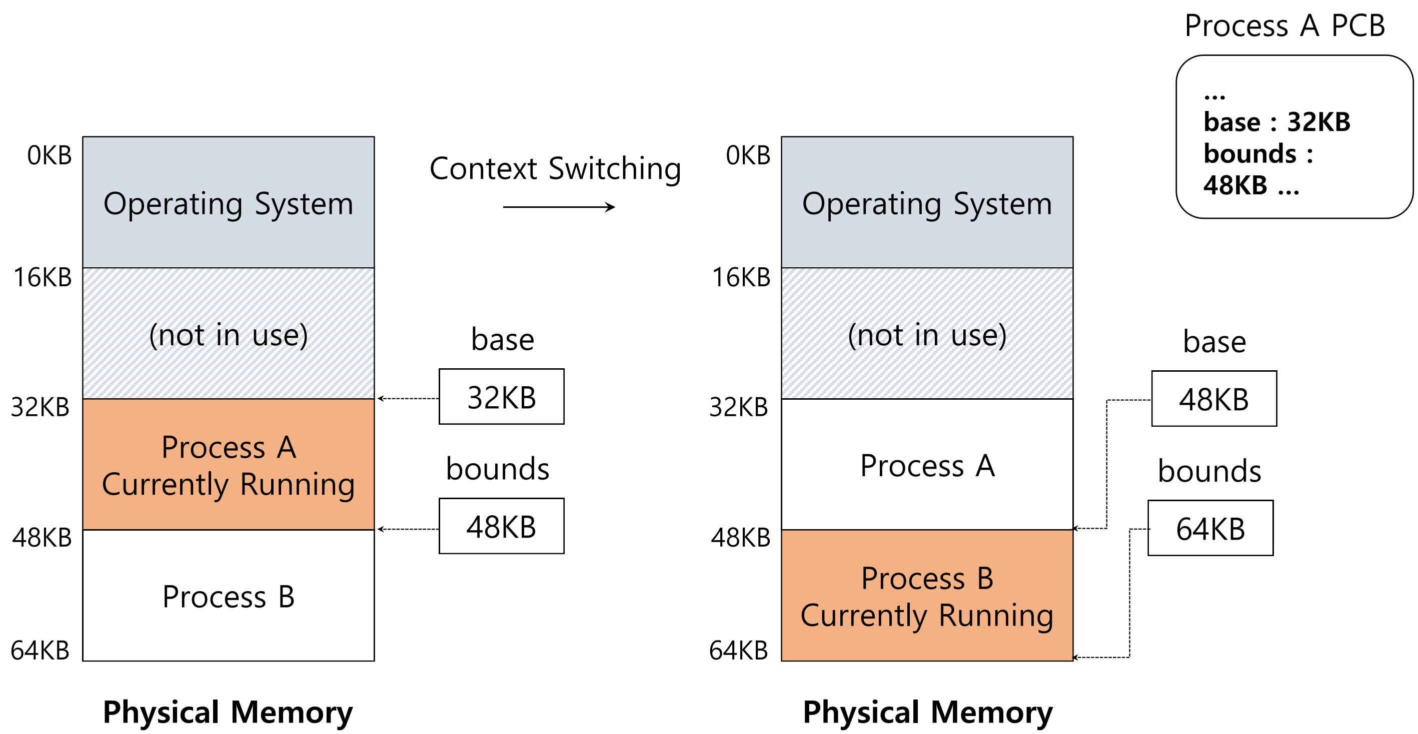
OS Issues: When a Process Is Terminated

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

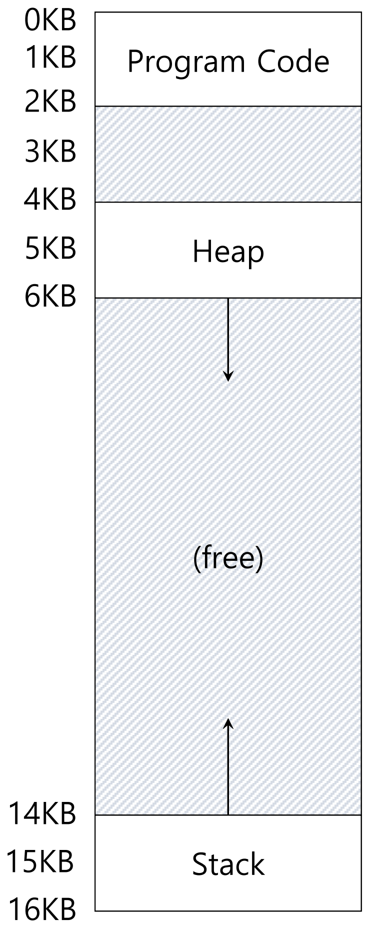


OS Issues: When Context Switch Occurs

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



Inefficiency of the Base and Bound Approach



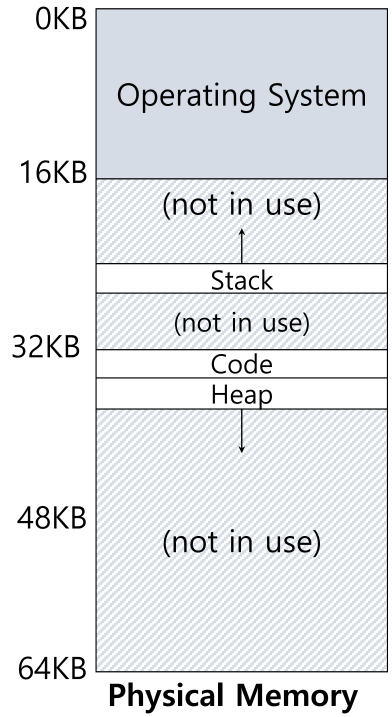
- **Big chunk of "free"** space
- "free" space **takes up** physical memory.
- Hard to run when an address space **does not fit** into physical memory

Segmentation

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

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

Placing Segment In Physical Memory



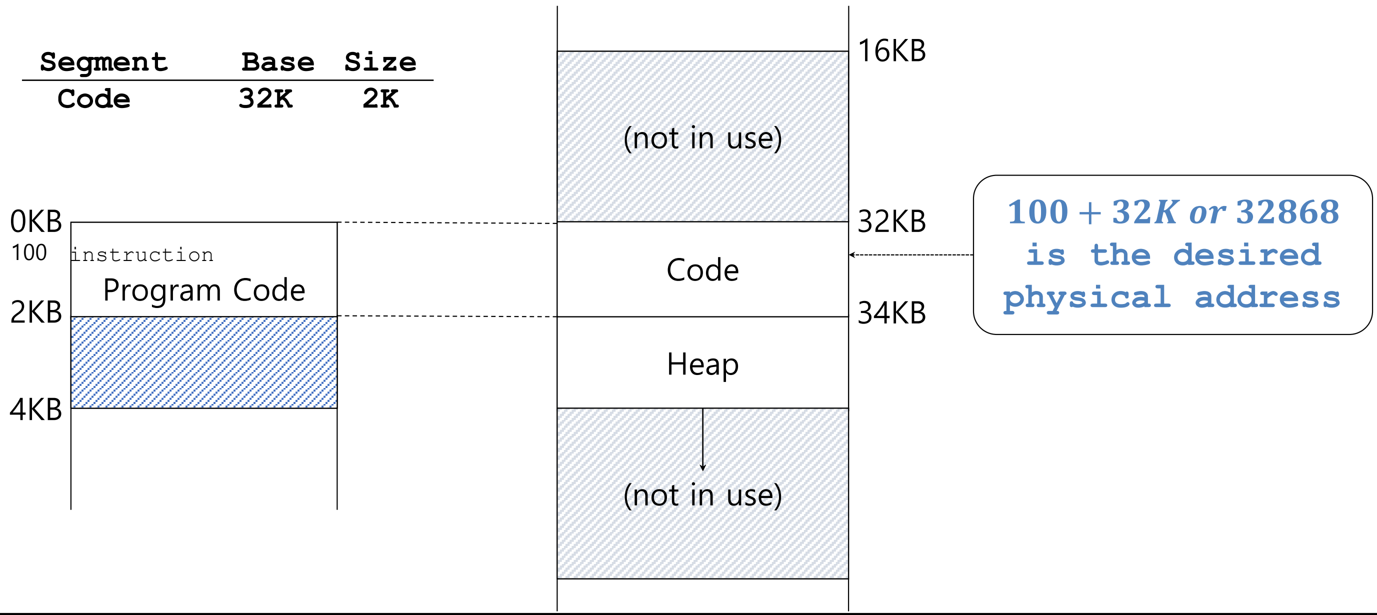
Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Address Translation on Segmentation

$$physical\ address = offset + base$$

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

Segment	Base	Size
Code	32K	2K

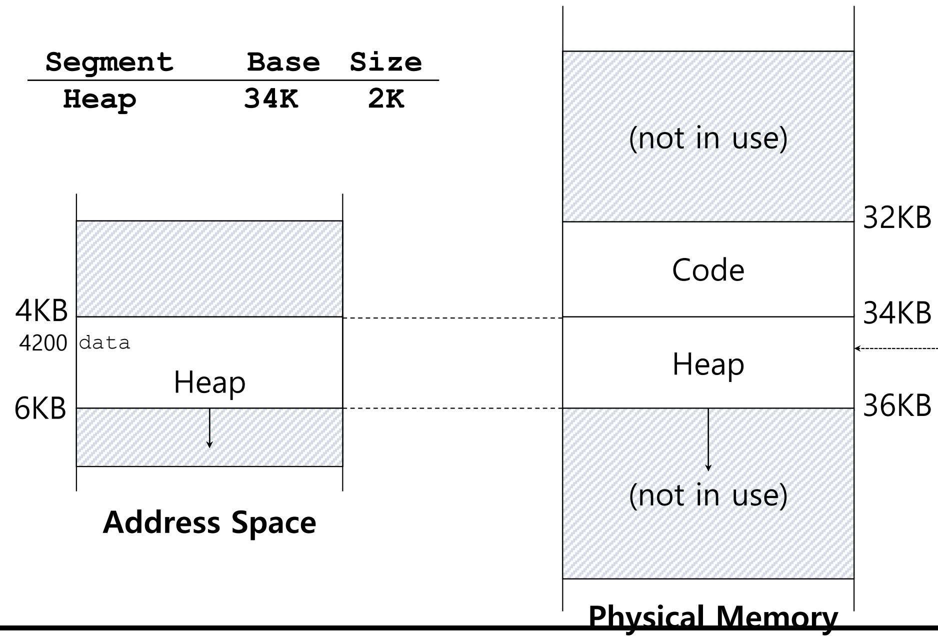


Address Translation on Segmentation (Cont.)

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.

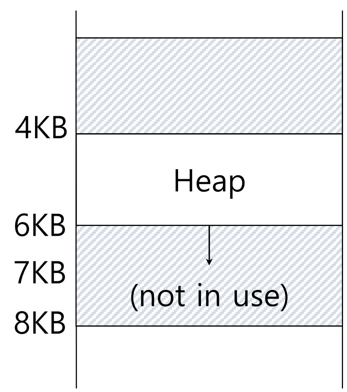
Segment	Base	Size
Heap	34K	2K



$104 + 34K$ *or* 34920 is the desired physical address

Segmentation Fault or Violation

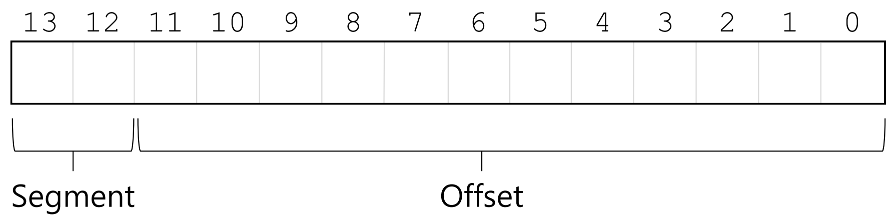
- If an **illegal address** such as 7KB which is beyond the end of heap is referenced, the OS occurs a **segmentation fault**.
  - The hardware detects that address is **out of bounds**.



Address Space

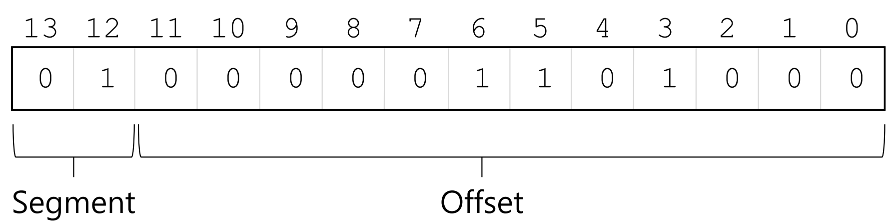
Referring to Segment

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



- Example: virtual address 4200 (01000001101000)

Segment	bits
Code	00
Heap	01
Stack	10
-	11



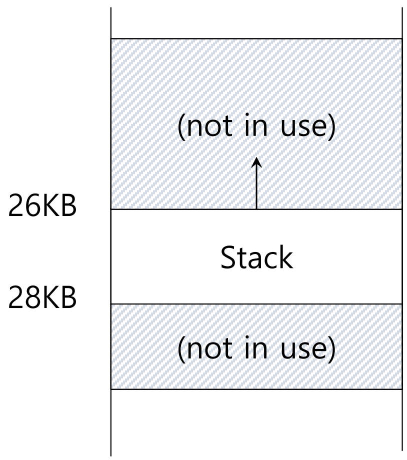
Referring to Segment (Cont.)

```
// 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



Physical Memory

Segment Register(with Negative-Growth Support)

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

Support for Sharing

- Segment can be **shared between address** space.
  - **Code sharing** is still in use in systems today.
  - by extra hardware support.
- Extra hardware support is need for 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

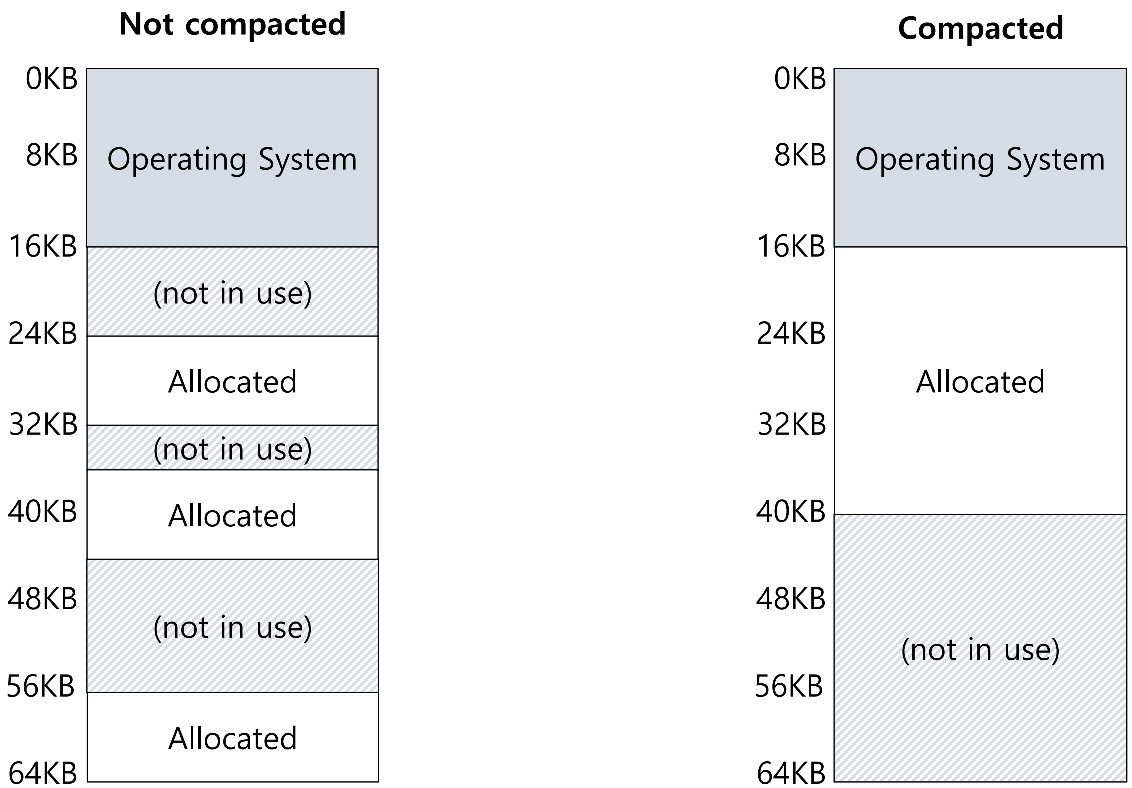
- **Coarse-Grained** means segmentation in a small number.
  - e.g., code, heap, stack.
- **Fine-Grained** segmentation allows **more flexibility** for the address space in some early systems.
  - To support many segments, hardware support with a **segment table** is required.

OS Support: Fragmentation

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

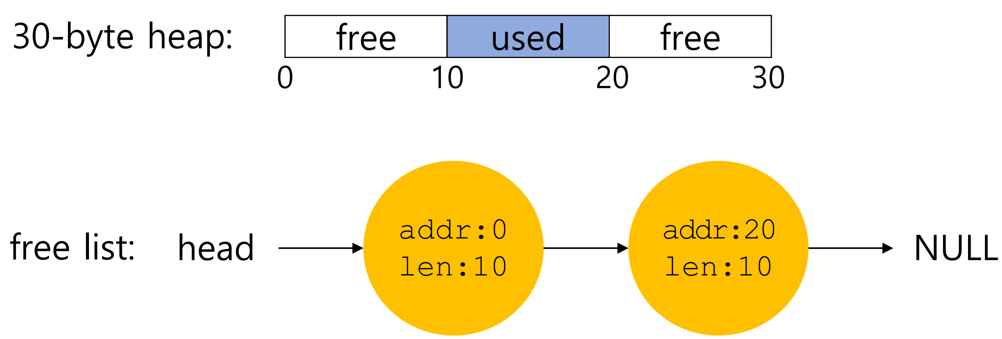
- **Compaction**: **rearranging** the exiting segments in physical memory.
  - Compaction is **costly**.
    - **Stop** running process.
    - **Copy** data to somewhere.
    - **Change** segment register value.

Memory Compaction



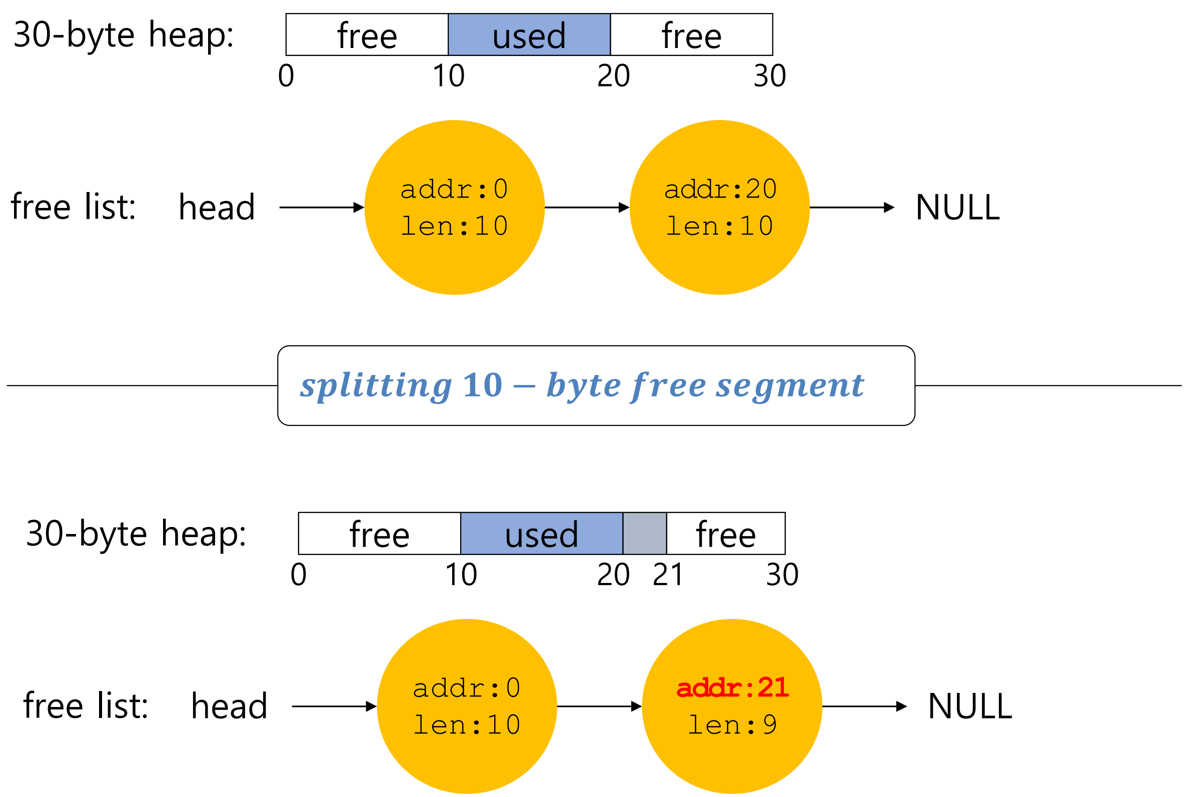
Splitting

- 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.



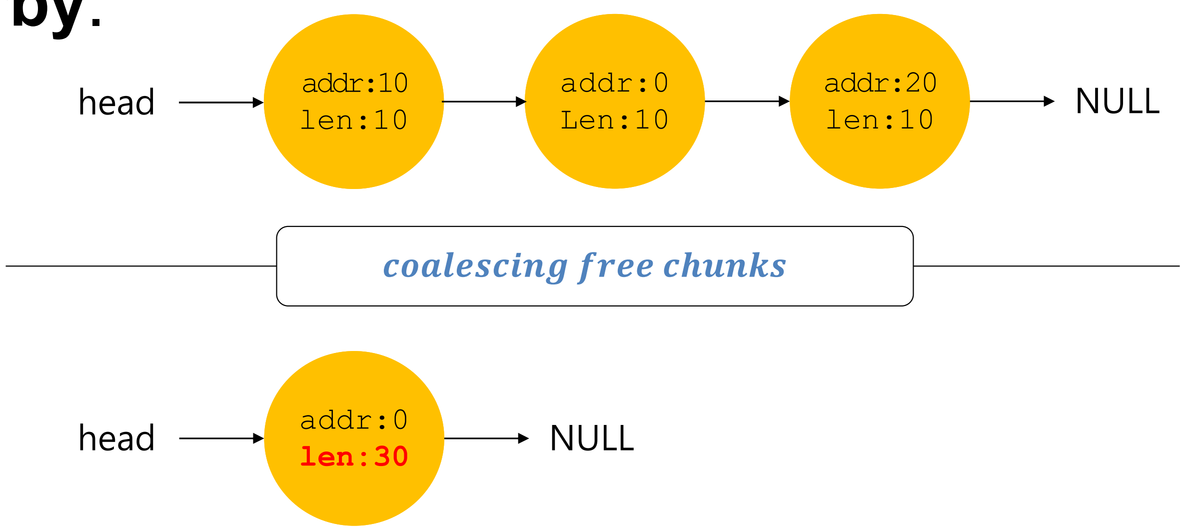
Splitting (Cont.)

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



Coalescing

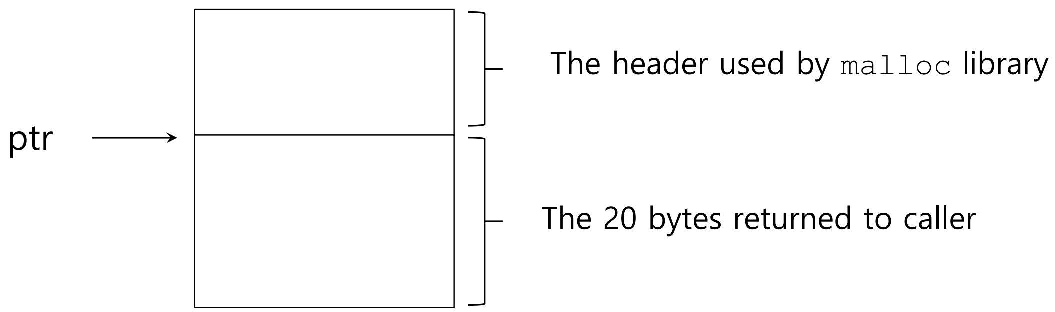
- If a user requests memory that is **bigger than free chunk size**, the list will **not find** such a free chunk.
- Coalescing: **Merge** returning a free chunk with existing chunks into a large single free chunk if **addresses** of them are **nearby**.



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 the memory region that will be back **into the free list**?
- Most allocators store **extra information** in a **header** block.

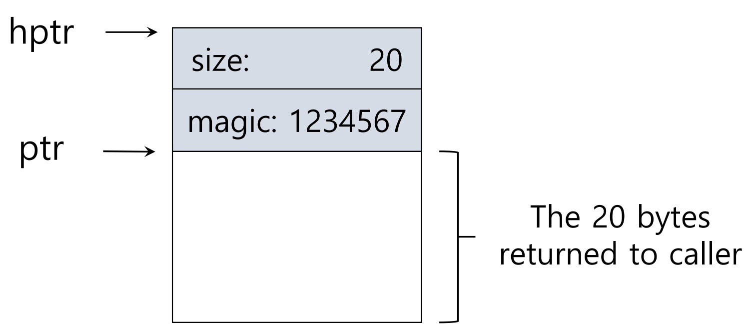
```
ptr = malloc(20);
```



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 de-allocation
  - A magic number for integrity checking



Specific Contents Of The Header

```c
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 **requests `N` bytes**, the library searches for a free chunk of **size `N` plus the size of the header**

- Simple pointer arithmetic to find the header pointer.

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

Embedding A Free List

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

Embedding A Free List (Cont.)

- Description of a node of the list

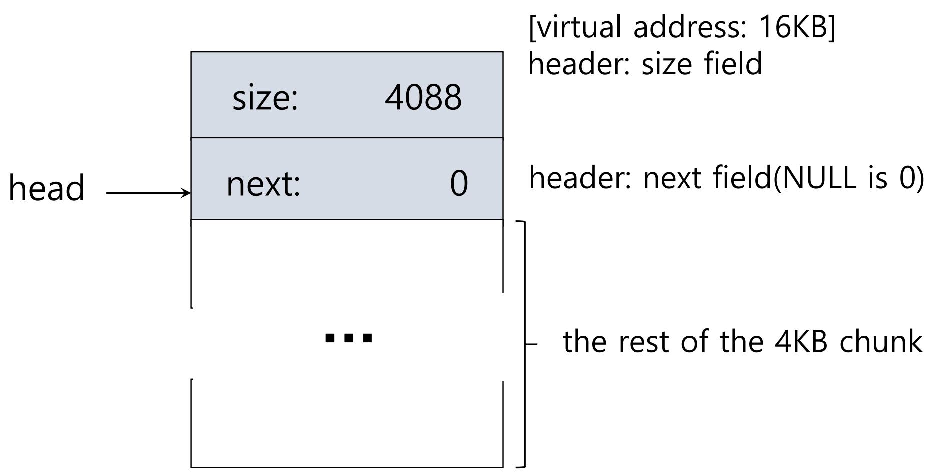
```
typedef struct __node_t {
     int size;
     struct __node_t *next;
} nodet_t;
```

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

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

A Heap With One Free Chunk

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



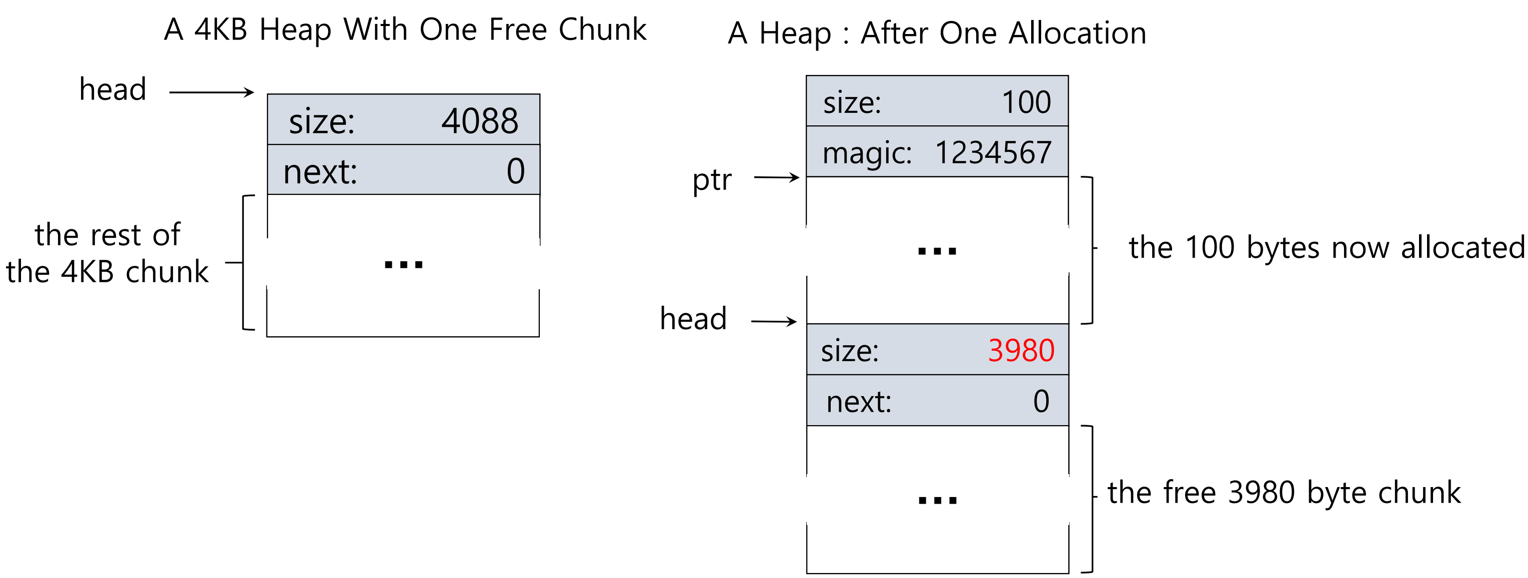
Embedding A Free List: Allocation

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

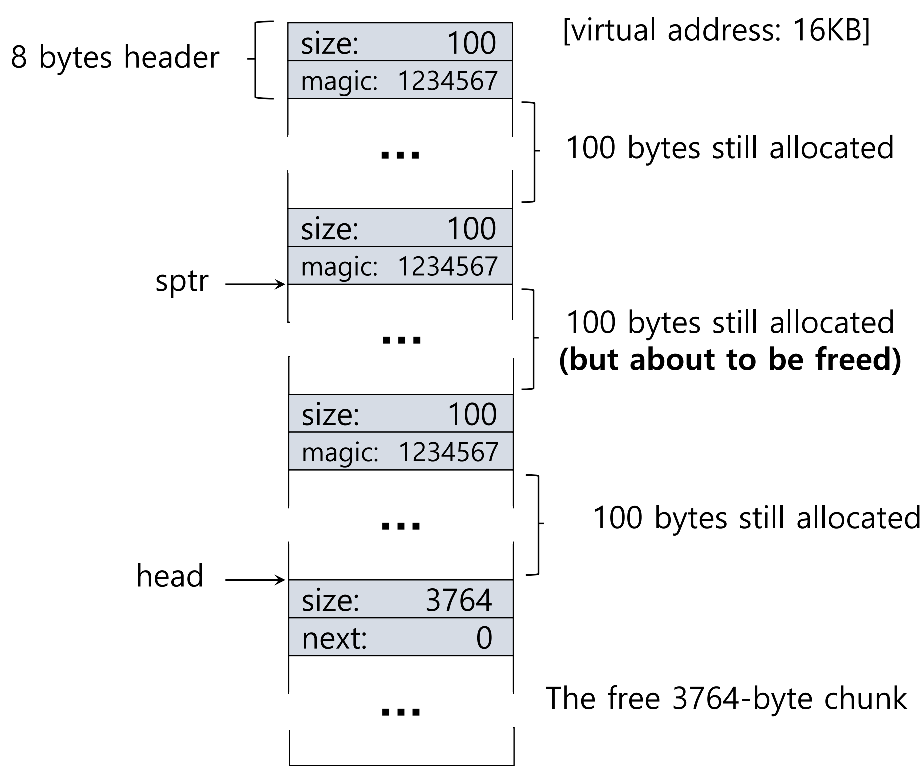
- The library will
  - **Split** the large free chunk into two.
    - **One** for the **request** and the **remaining** free chunk
  - **Shrink** the size of the free chunk in the list.

Embedding A Free List: Allocation (Cont.)

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



Free Space With Chunks Allocated

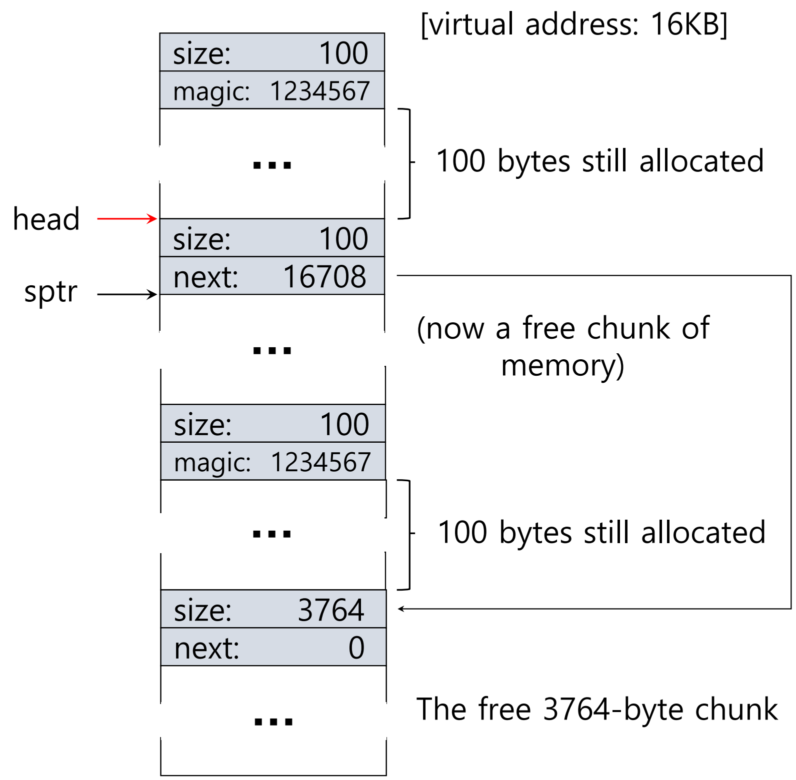


Free Space With Three Chunks Allocated

Free Space With `free()`

Example: `free(sptr)`

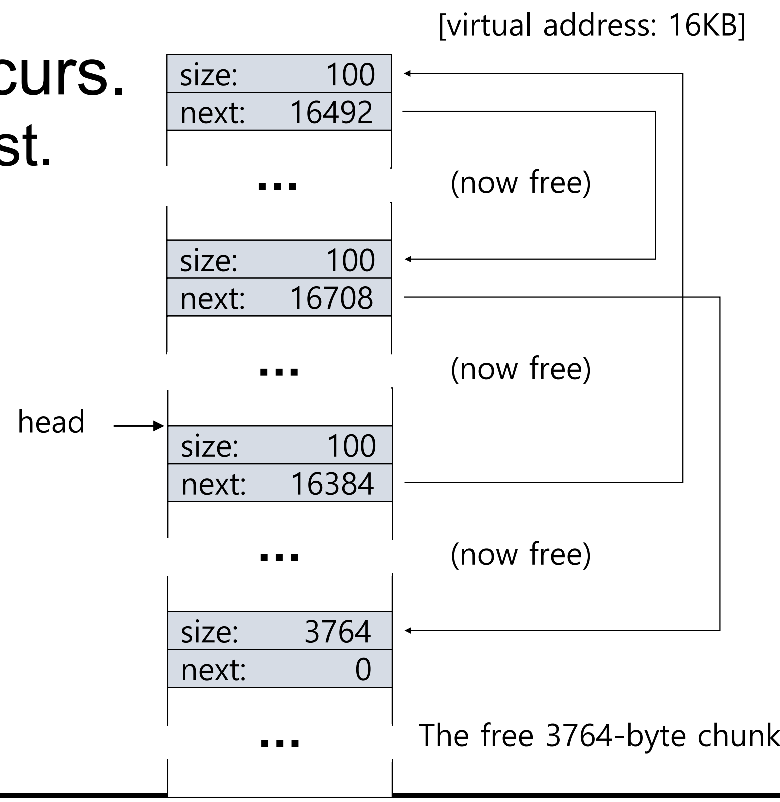
- The 100 bytes chunk is **back into** the free list.
- The free list will **start** with **a small chunk**.
  - The list header will point to the small chunk



Free Space With Freed Chunks

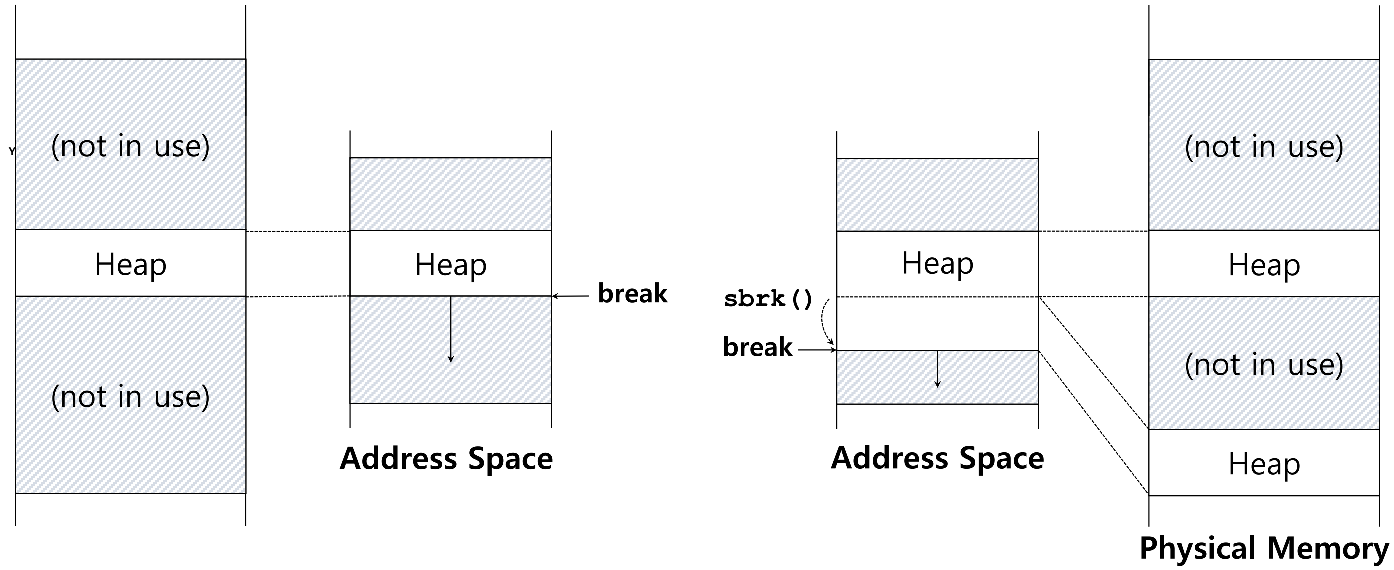
- Let's assume that the last two in-use chunks are freed.

- **External Fragmentation** occurs.
  - **Coalescing** is needed in the list.



Growing The Heap

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



Managing Free Space: Basic Strategies

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

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

Managing Free Space: Basic Strategies (Cont.)

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

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

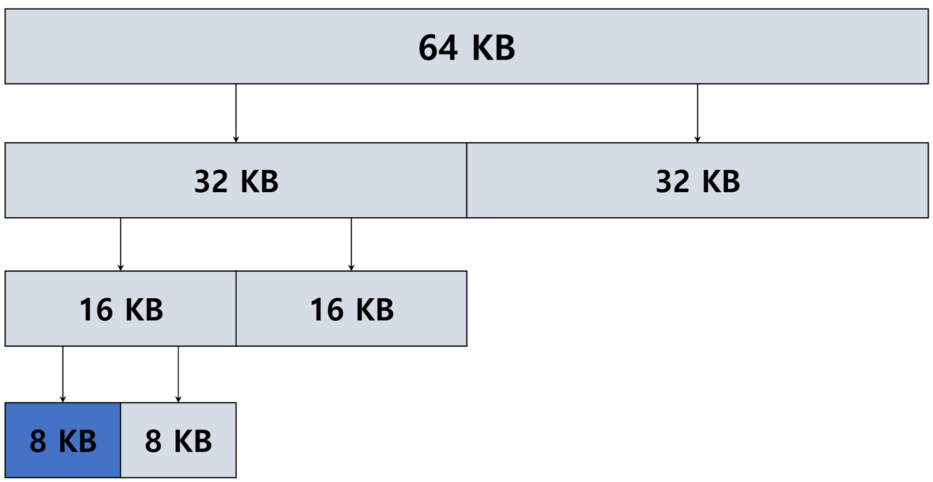
- Segregated List:
  - Keeping free chunks in different size in a separate list for the size of popular request.
  - New Complication:
    - **How much** memory should dedicate to **the pool of memory** that serves **specialized requests** of a given size?
  - **Slab allocator** handles this issue.

Other Approaches: Segregated List (Cont.)

- Slab Allocator
  - Allocate several 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 into the next level of the block.

Break

Concept of Paging

- Paging **splits up** address space into **fixed-zed** 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**.

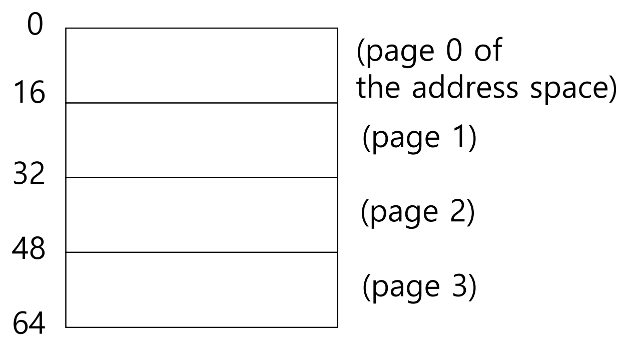
- **Page table** per process is needed **to translate** the virtual address to physical address.

Advantages of Paging

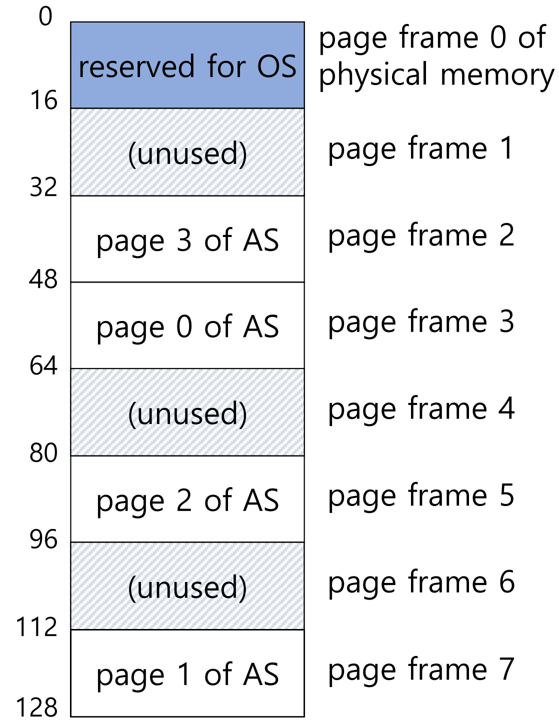
- **Flexibility**: Supporting the abstraction of address space effectively
  - Don’t need assumption how heap and stack grow and are used.
- **Simplicity**: ease of free-space management
  - The page in address space and the page frame are the same size.
  - Easy to allocate and keep a free list

Example: A Simple Paging

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



A Simple 64-byte Address Space



64-Byte Address Space Placed In Physical Memory

Address Translation

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

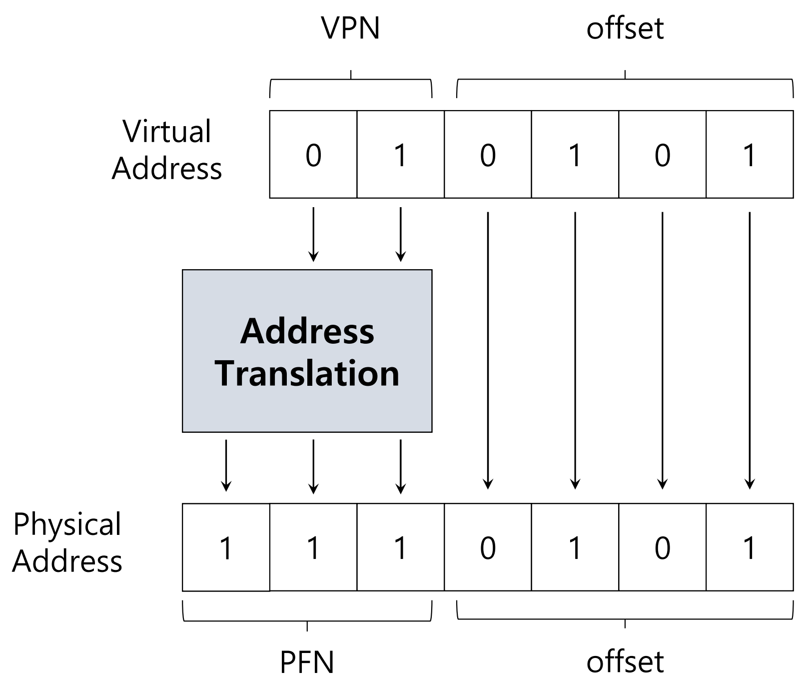
VPN		offset			
Va5	Va4	Va3	Va2	Va1	Va0

- 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

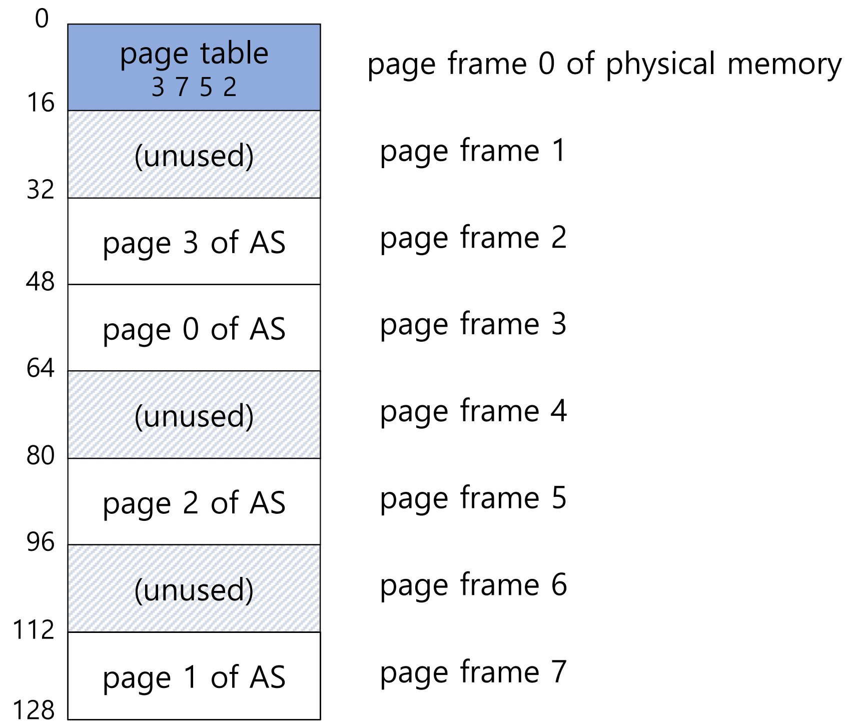


Where are Page Tables Stored?

- 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



Physical Memory

What is in the Page Table?

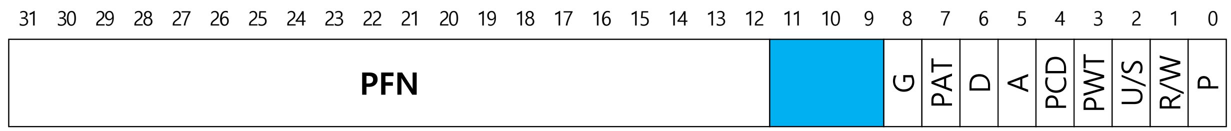
- The page table is just a **data structure** that is used to map the virtual address to physical address.
  - Simplest form: a linear page table, an array

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

Common Flags of Page Table Entries

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

Example: x86 Page Table Entry



An x86 Page Table Entry(PTE)

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

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**.

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;
```

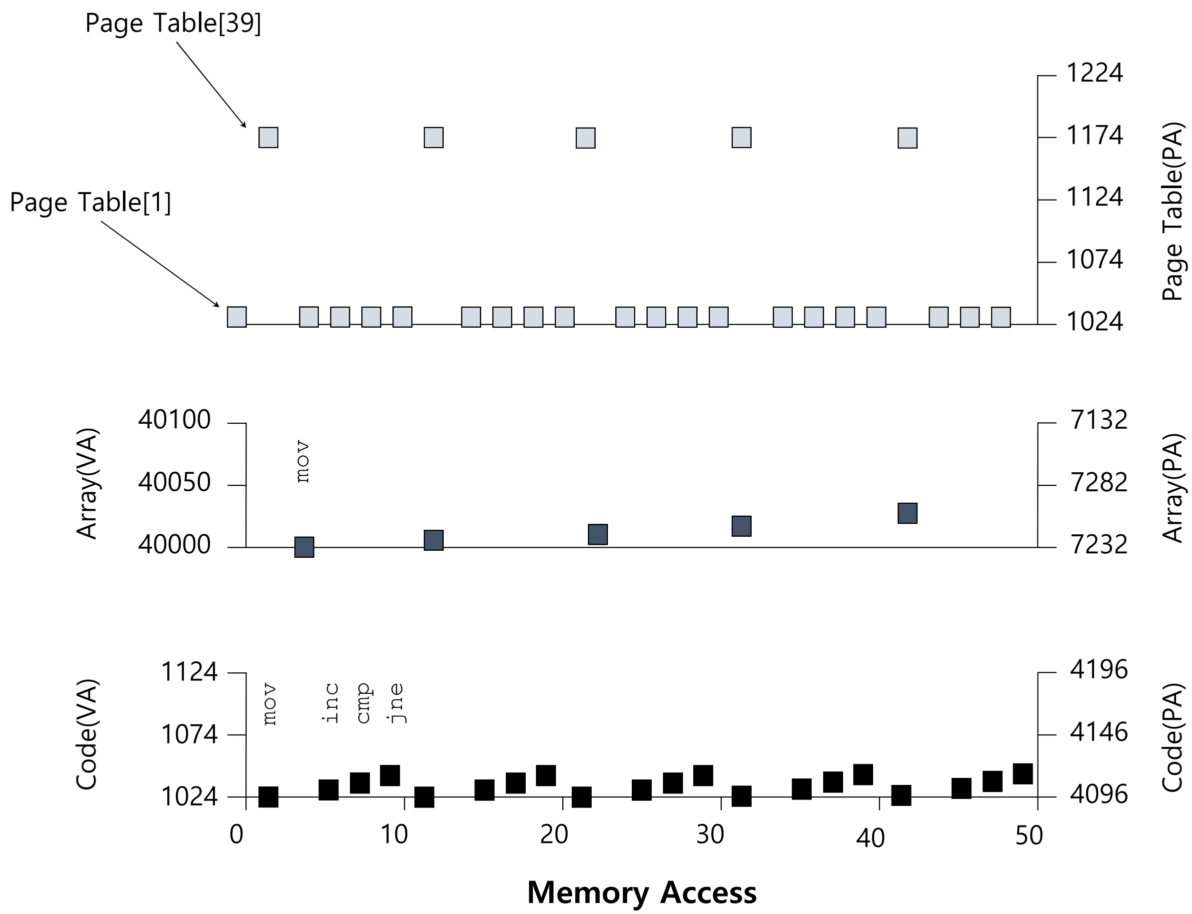
- Compile and execute

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

- Resulting Assembly code

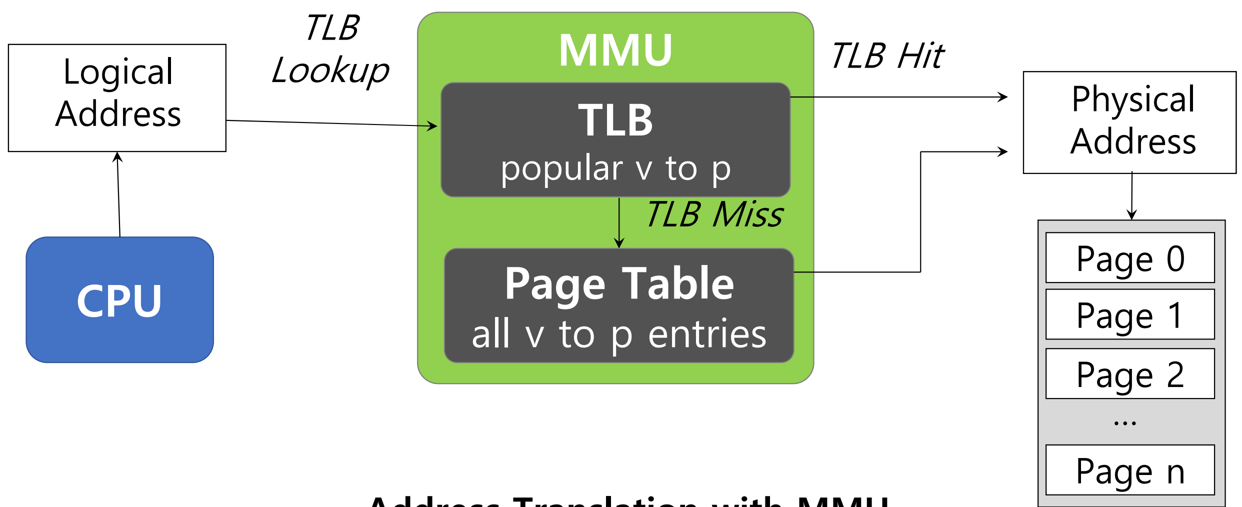
```
0x1024 movl $0x0,(%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne 0x1024
```

A Virtual (and Physical) Memory Trace



Translation Lookaside Buffer (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)
```

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

TLB Basic Algorithms (Cont.)

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

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

Example: Accessing An Array

- How a TLB can improve its performance

OFFSET	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

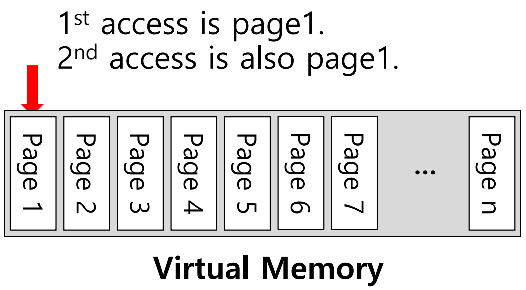
```c
int sum = 0 ;
for( i=0; i<10; i++){
      sum+=a[i];
}
```

The TLB improves performance due to spatial locality

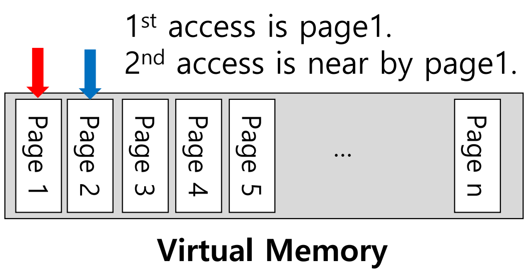
3 misses and 7 hits.
Thus TLB hit rate is 70%.

Locality

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



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



Who Handles the TLB Miss?

- Hardware handle the TLB miss entirely on CISC.
  - The hardware has to know exactly where the page tables are located in memory.
  - The hardware would “walk” the page table, find the correct page-table entry and extract the desired translation, update and retry instruction.
  - **hardware-managed TLB.**

Who Handles the TLB Miss? (Cont.)

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

TLB Control Flow Algorithm (OS Handled)

```
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)
```

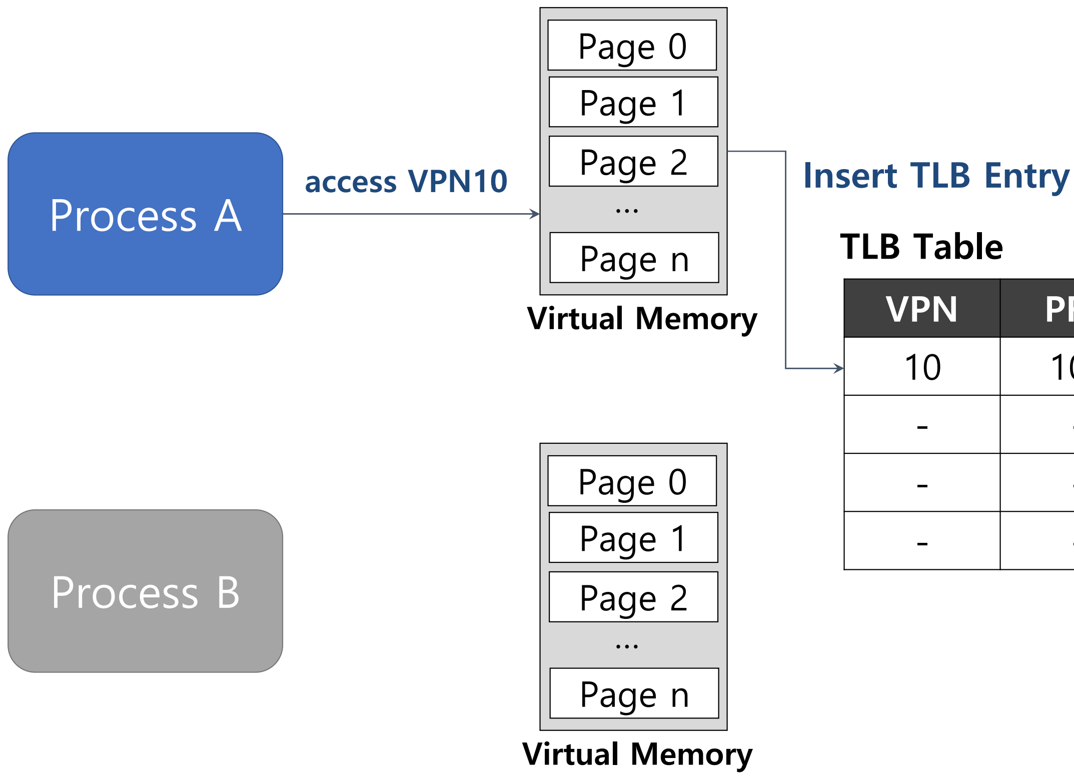
TLB Entry

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

VPN	PFN	other bits

Typical TLB entry look like this

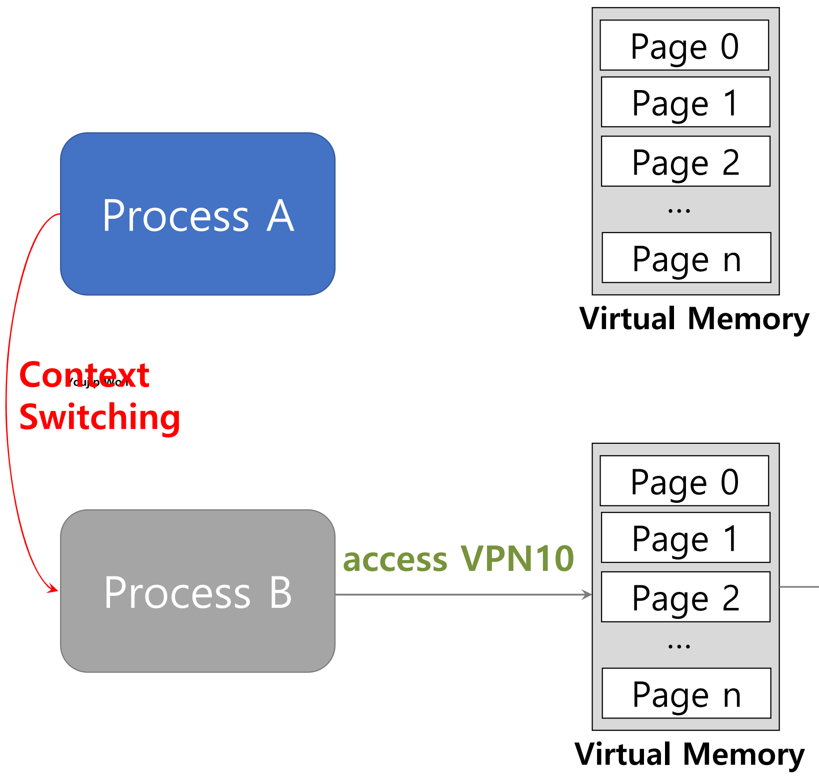
TLB Issue: Context Switching



TLB Table

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

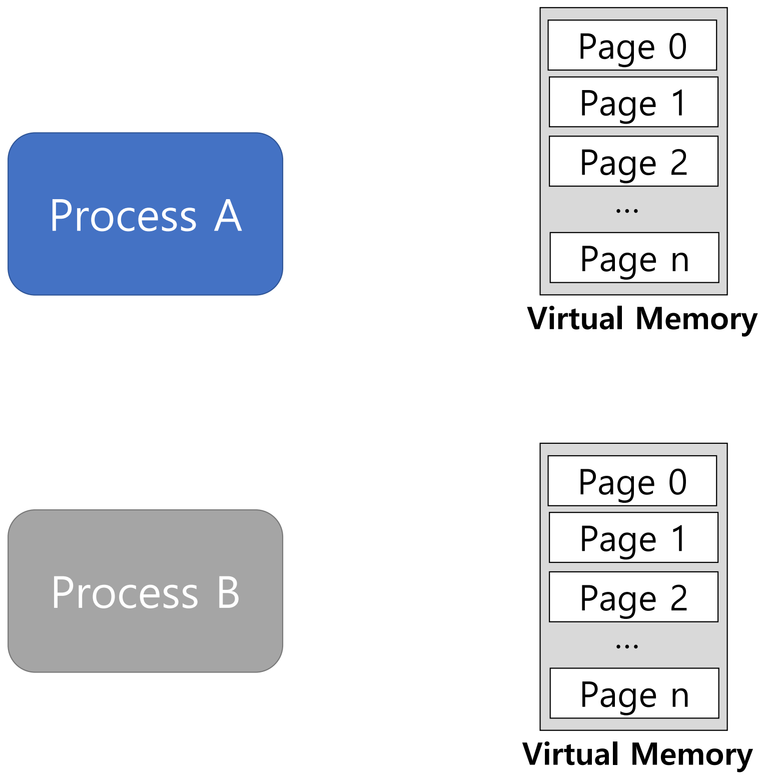
TLB Issue: Context Switching



TLB Table

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

TLB Issue: Context Switching



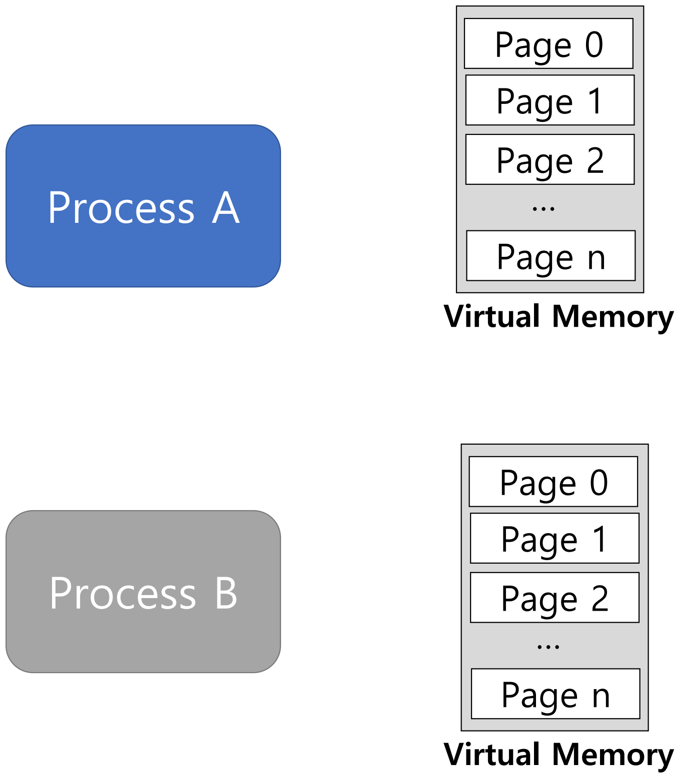
TLB Table

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

Can't distinguish which entry is meant for which process

To Solve the 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.

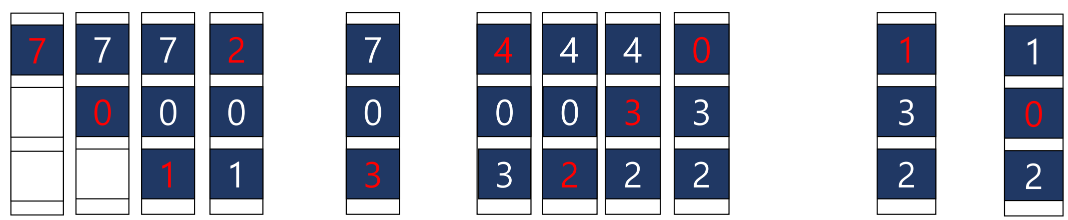
TLB Replacement Policy

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

Reference Row

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

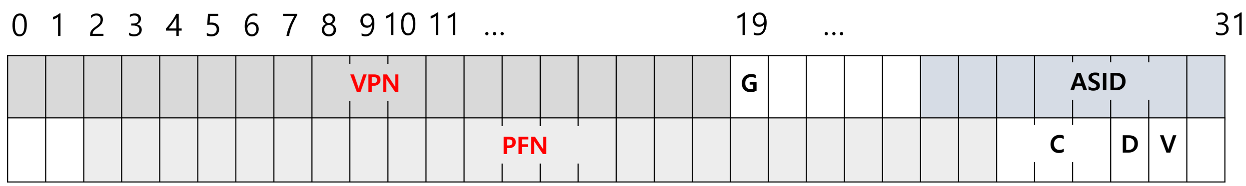
Page Frame:



Total 11 TLB miss

A Real TLB Entry

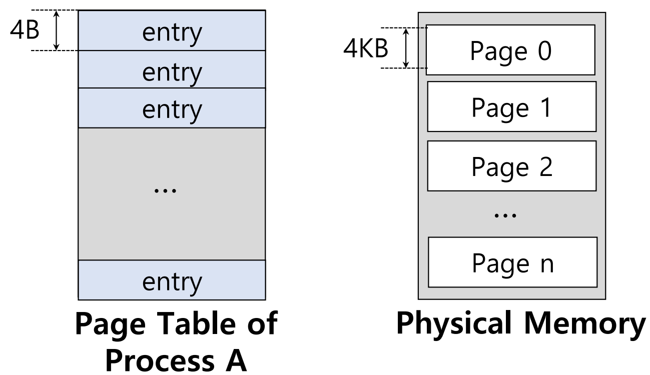
All 64 bits of this TLB entry(example of MIPS R4000)



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

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.

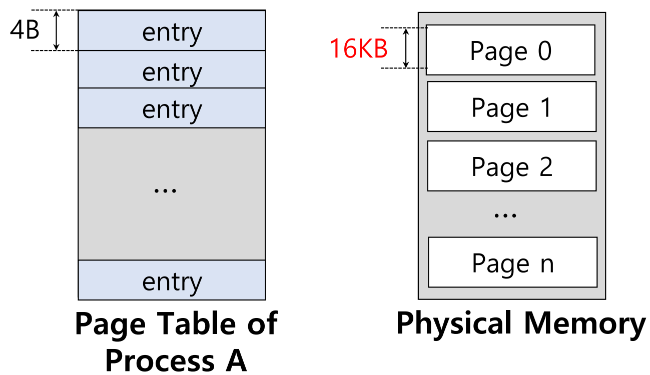


Page table size = $\frac{2^{32}}{2^{12}} * 4Byte = 4MByte$

Page tables 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.

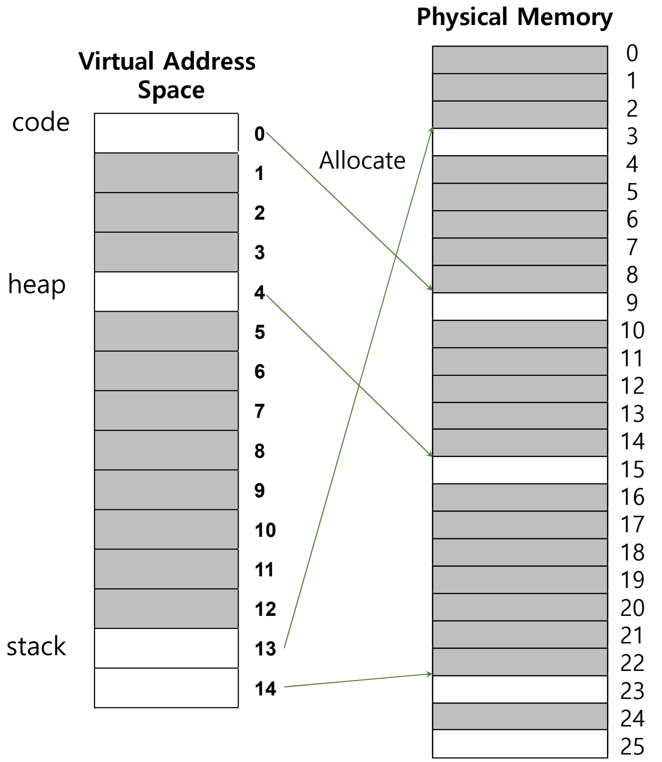


$\frac{2^{32}}{2^{16}} * 4 = 1MB$ per page table

Big pages lead to internal fragmentation.

Problem

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



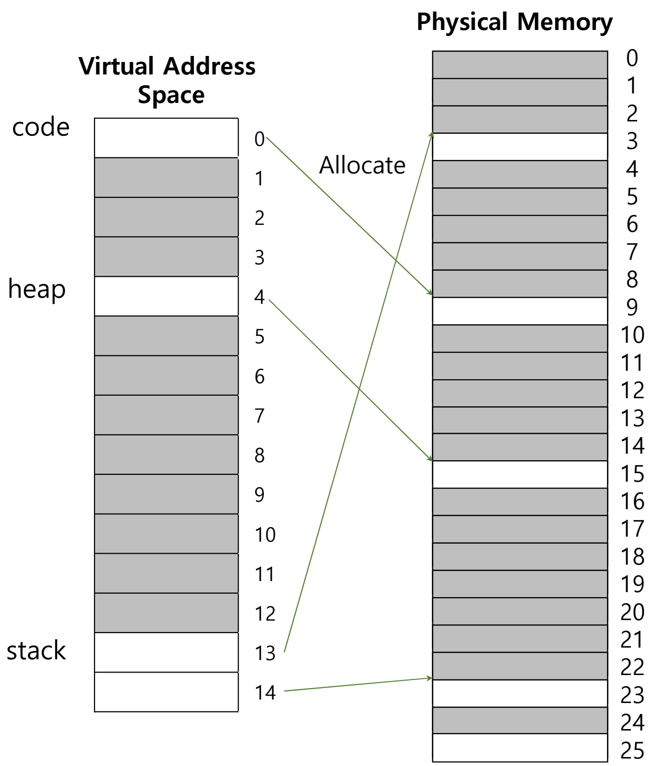
A 16KB Address Space with 1KB Pages

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

A Page Table For 16KB Address Space

Problem

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



A 16KB Address Space with 1KB Pages

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

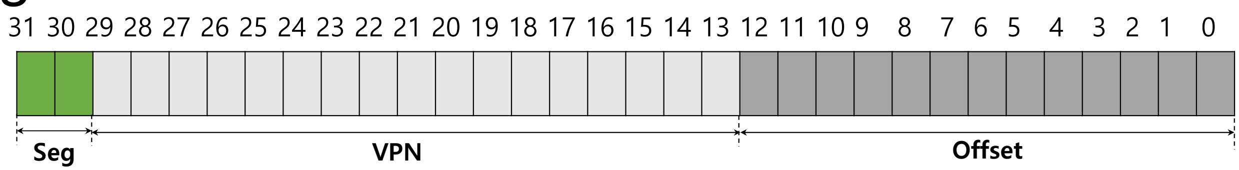
A Page Table For 16KB Address Space

Hybrid Approach: Paging and Segments

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

Simple Example of Hybrid Approach

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



32-bit Virtual address space with 4KB pages

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

TLB miss on Hybrid Approach

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

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

Problem of Hybrid Approach

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

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.

Multi-level Page Tables: Page Directory

Linear Page Table

PBTR: 201

valid	prot	PFN	
1	rx	12	PFN201
1	rx	13	
0	-	-	
1	rw	100	
0	-	-	PFN202
0	-	-	
0	-	-	
0	-	-	
0	-	-	PFN203
0	-	-	
1	rw	86	
1	rw	15	

Multi-level Page Table

PBTR: 200

The Page Directory (PFN200)

valid	PFN
1	201
0	-
0	-
1	203

PFN201

valid	prot	PFN
1	rx	12
1	rx	13
0	-	-
1	rw	100

[Page 1 of PT:Not Allocated]

[Page 2 of PT: Not Allocated]

PFN204

valid	prot	PFN
0	-	-
0	-	-
1	rw	86
1	rw	15

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

Multi-level Page Tables: Page Directory Entries

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

Multi-level Page Tables: Advantages & Disadvantages

- Advantage
  - Only allocates page-table space in proportion to the amount of address space you are using.
  - The OS can grab the next free page when it needs to allocate or grow a page table.

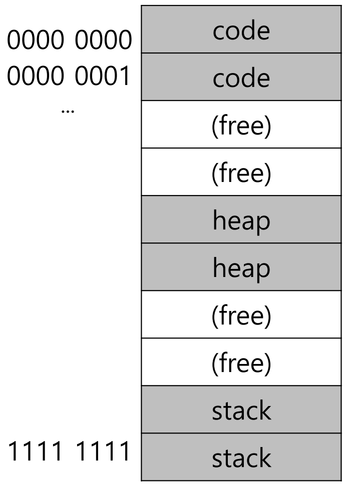
- Disadvantage
  - Multi-level table is a small example of a time-space trade-off.
  - Complexity.

Multi-level Page Table: Level of Indirection

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

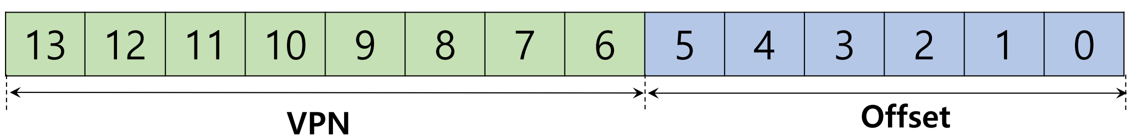
A Detailed Multi-Level Example

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



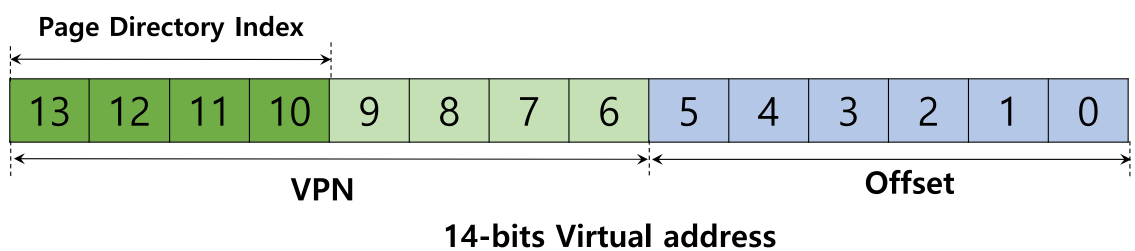
Flag	Detail
Address space	16 KB
Page size	64 byte
Virtual address	14 bit
VPN	8 bit
Offset	6 bit
Page table entry	2^8(256)

A 16-KB Address Space With 64-byte Pages



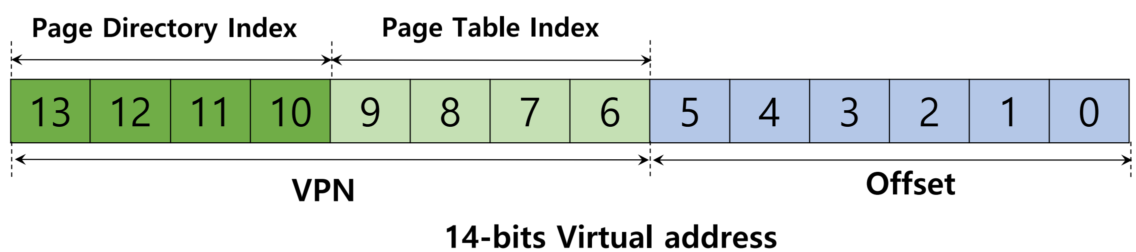
A Detailed Multi-Level Example: Page Directory Index

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



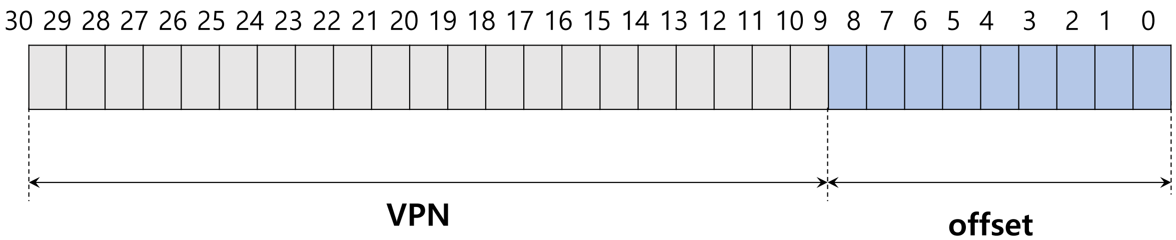
A Detailed Multi-Level Example: Page Table Index

- 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 Levels

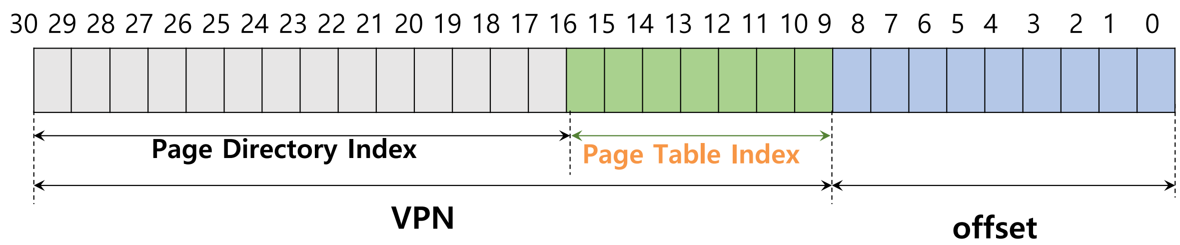
- In some cases, a deeper tree is possible.



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

More than Two Levels : Page Table Index

- In some cases, a deeper tree is possible.

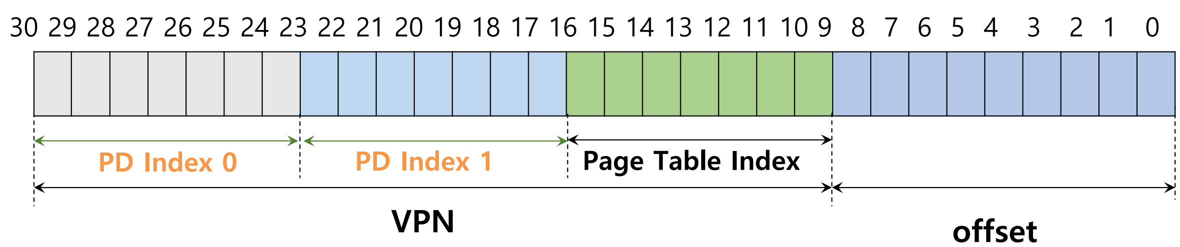


Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit
Page entry per page	128 PTEs

$$\log_2 128 = 7$$

More than Two Levels: Page Directory

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



Multi-level Page Table Control Flow

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

- (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

Multi-level Page Table Control Flow

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

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

The Translation Process: Remember the TLB

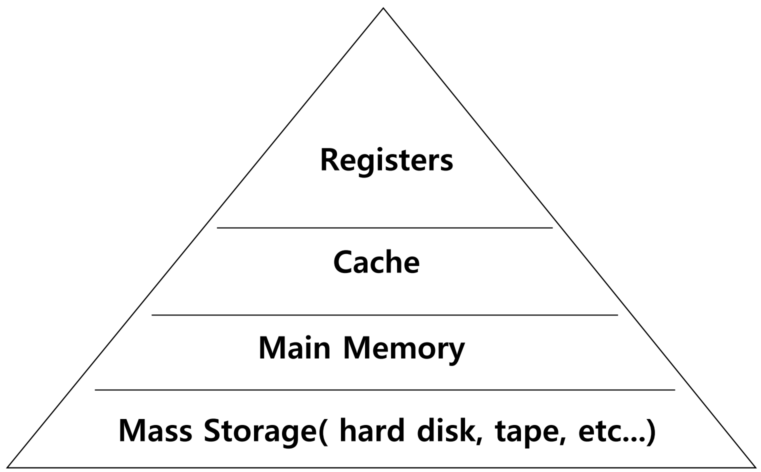
```
PTIndex = (VPN & PT_MASK) >> PT_SHIFT
PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
PTE = AccessMemory(PTEAddr)
if(PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else if(CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT);
else
    TLB_Insert(VPN, PTE.PFN , PTE.ProtectBits)
    RetryInstruction()
```

Inverted Page Tables

- Keeping a single page table that has an entry for each <u>physical page</u> 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.

Beyond Physical Memory: Mechanisms

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



Memory Hierarchy in modern system

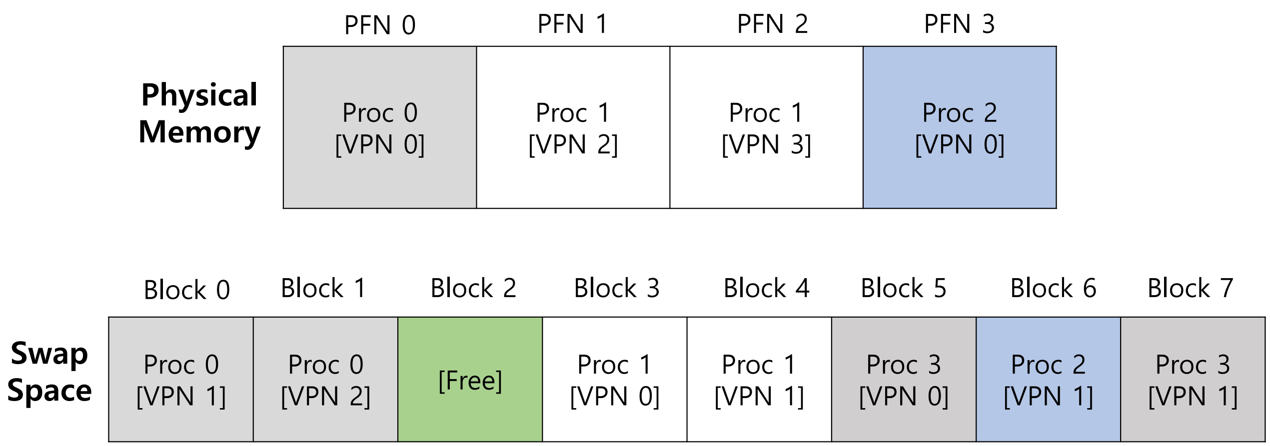
Single large address for a process

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

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

Swap Space

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



Physical Memory and Swap Space

Present Bit

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

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

What If Memory Is Full?

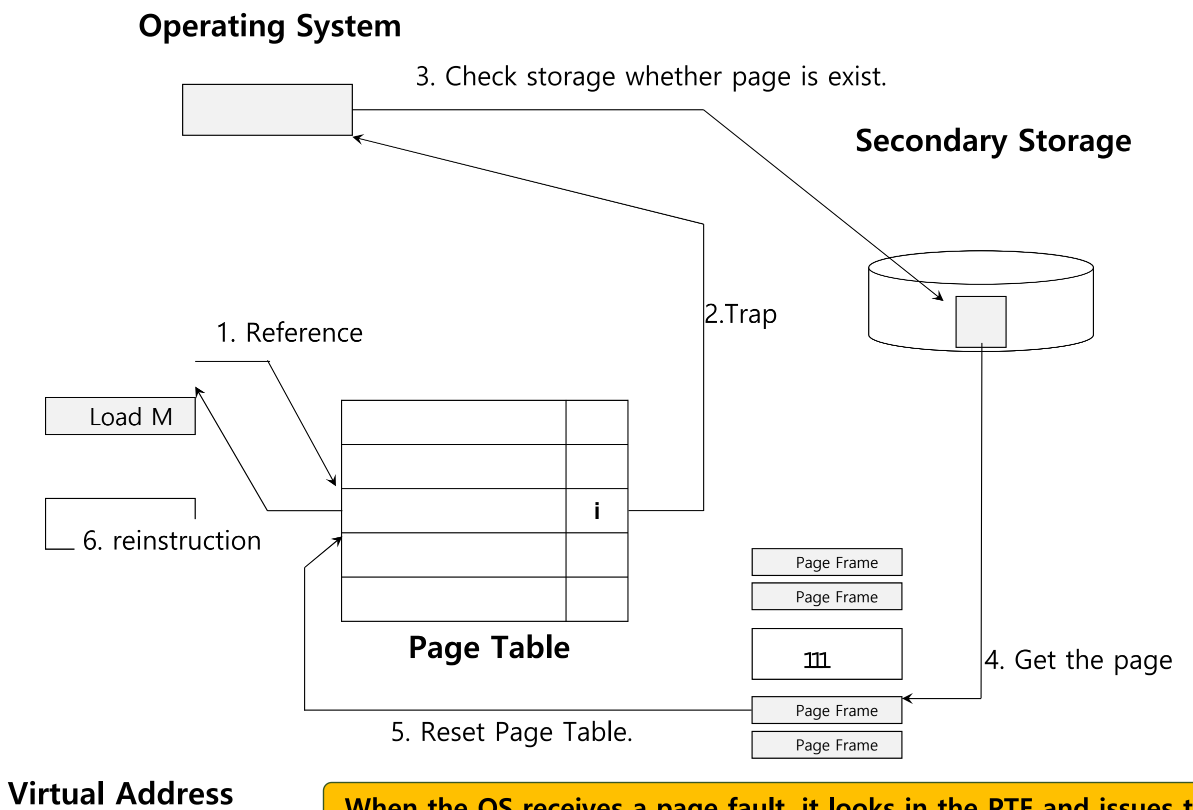
- The OS like to page out pages to make room for the new pages the OS is about to bring in.
  - The process of picking a page to kick out, or replace is known as page-replacement policy

The Page Fault

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

Page Fault Control Flow

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



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

Page Fault Control Flow – Hardware

```
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.

Beyond Physical Memory: Policies

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

Cache Management

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

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

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

The Optimal Replacement Policy

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

Tracing the Optimal Policy

Reference Row

0 1 2 0 1 3 0 3 1 2 1

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

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

Future is not known.

A Simple Policy: FIFO

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

Tracing the FIFO 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 $\frac{Hits}{Hits+Misses} = \mathbf{36.4}\%$

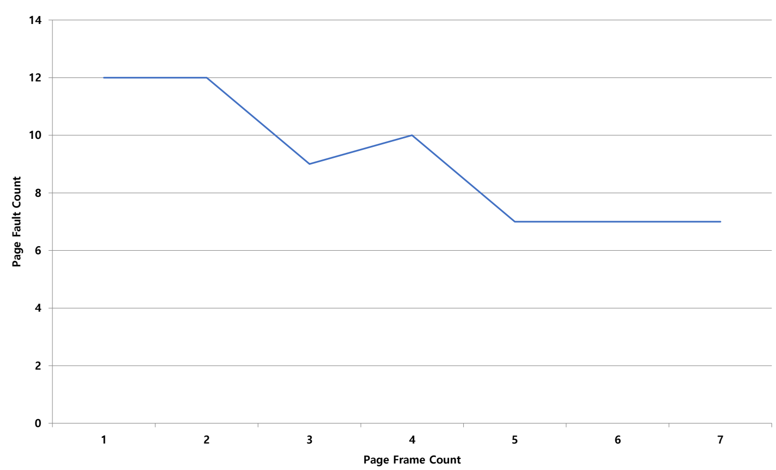
Even though page 0 had been accessed a number of times, FIFO still kicks it out.

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.

Reference Row

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



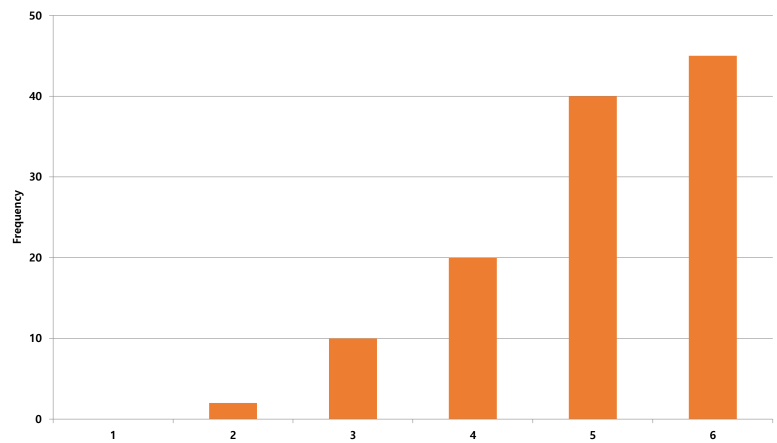
Another Simple Policy: Random

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

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

Random Performance

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



Random Performance over 10,000 Trials

Using History

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

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

Using History : LRU

- Replaces the least-recently-used page.

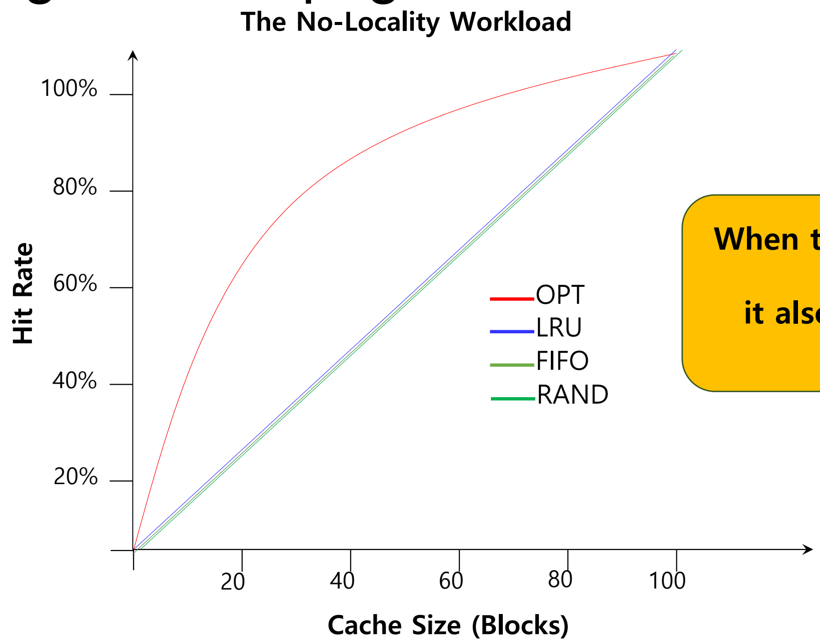
Reference Row

0 1 2 0 1 3 0 3 1 2 1

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

Workload Example : The No-Locality Workload

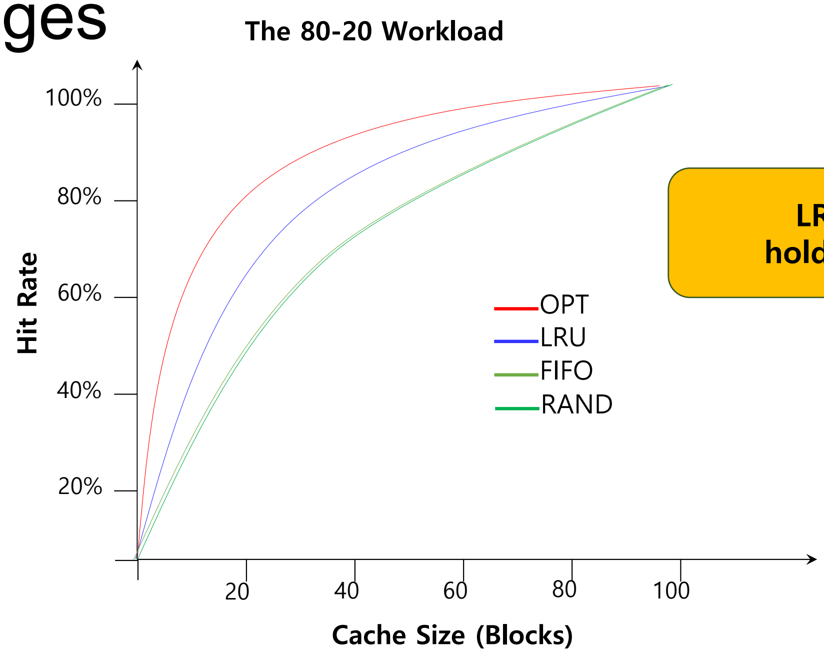
- Each reference is to a random page within the set of accessed pages.
  - Workload accesses 100 unique pages over time.
  - Choosing the next page to refer to at random



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

Workload Example : The 80-20 Workload

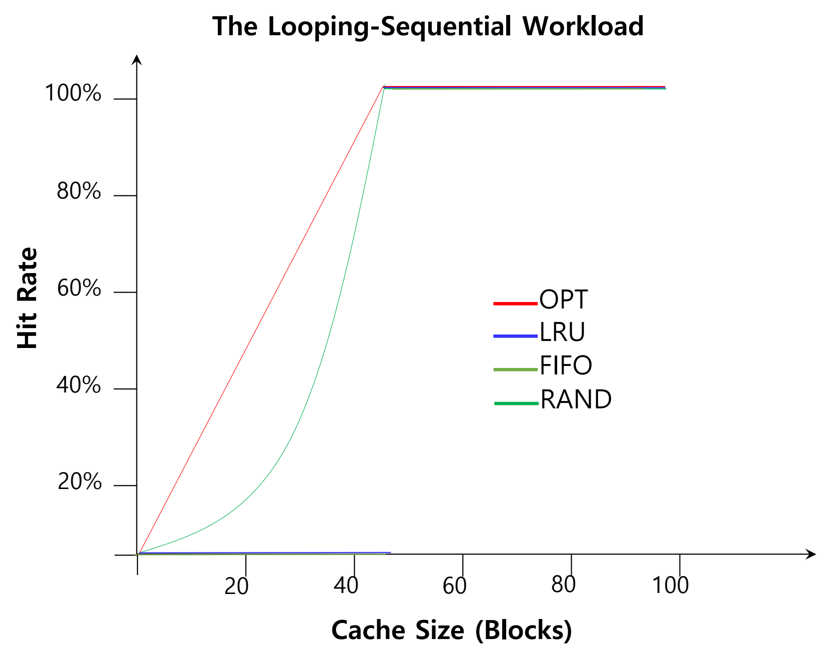
- Exhibits locality: 80% of the reference are made to 20% of the pages
- Remaining 20% of the reference are made to the remaining 80% of the pages



LRU is more likely to hold onto the **hot pages**.

Workload Example: The Looping Sequential

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



Implementing Historical Algorithms

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

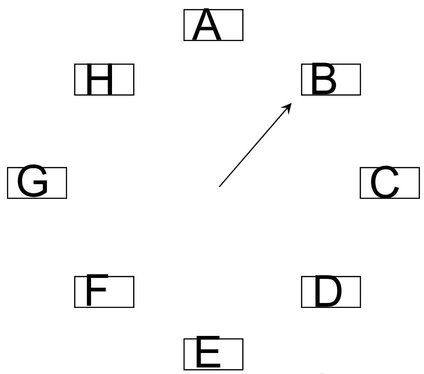
Approximating LRU

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

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

Clock Algorithm

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



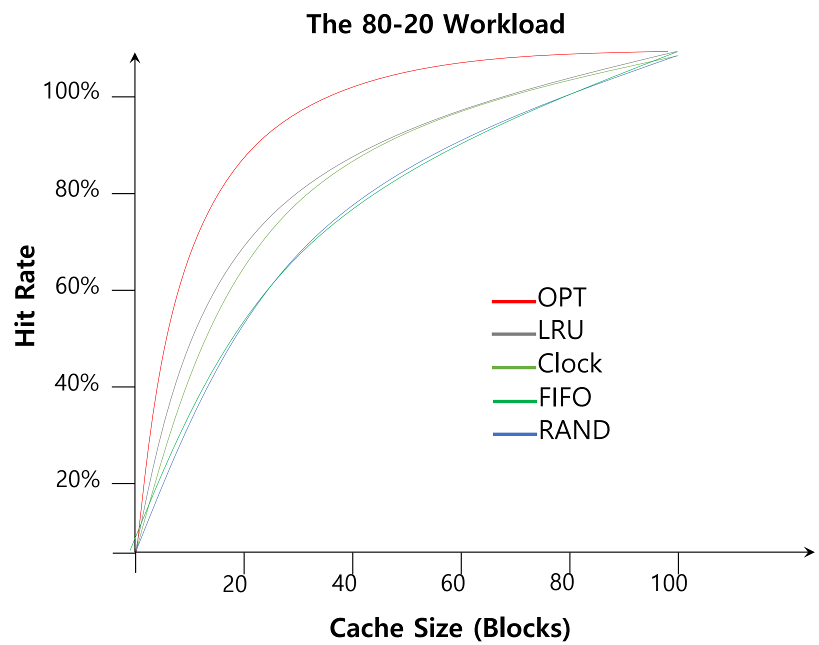
Use bit	Meaning
0	Evict the page
1	Clear **Use bit** and advance hand

The Clock page replacement algorithm

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

Workload with Clock Algorithm

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



Considering Dirty Pages

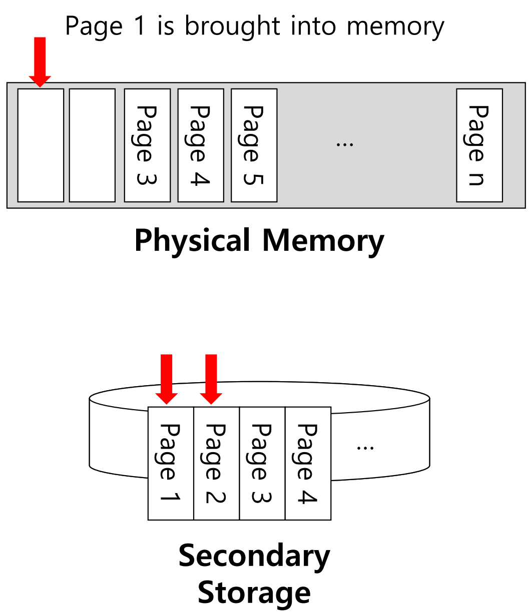
- The hardware include a <u>**modified bit**</u> (a.k.a <u>**dirty bit**</u>)
  - Page has been <u>**modified**</u> and is thus <u>**dirty**</u>, it must be written back to disk to evict it.
  - Page has not been modified, the eviction is free.

Page Selection Policy

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

Prefetching

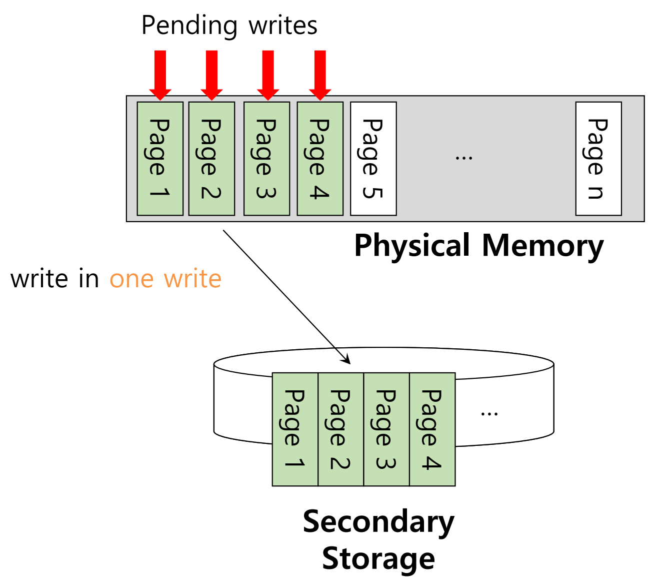
- The OS guess that a page is about to be used, and thus bring it in ahead of time.



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

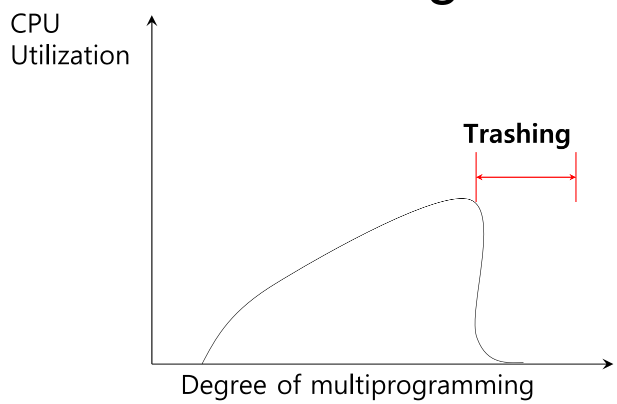
Clustering, Grouping

- 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.



Programming Assignment 2

Quiz 3

Homework 3

Project Launch

What's Next

- Homework Assignment 2
  - Due by Monday, February 9 by 11:59pm ET
- Quiz 2
  - Due by Monday, February 9 by 11:59pm ET
- Homework Assignment 3
  - Due by Sunday, February 15 by 11:59pm ET
- Quiz 3
  - Due by Sunday, February 15 by 11:59pm ET
- Programming Assignment 2 (PA2)
  - Due by Friday, February 27, 2026 by 11:59pm ET
- ~~Course Project~~
  - ~~Due by Friday, February 27, 2026 by 11:59pm ET~~