Lecture Note 8: Memory Management

May 11, 2021
Jongmoo Choi

Dept. of Software
Dankook University
http://embedded.dankook.ac.kr/~choijm

Contents

- From Chap 12~17 of the OSTEP
- Chap 12. A Dialogue on Memory Virtualization
- Chap 13. The Abstraction: Address Space
- Chap 14. Interlude: Memory API
  - ✓ malloc(), free(), brk(), mmap(), …
- Chap 15. Mechanism: Address Translation
  - ✓ Base & Limit (Bound), Dynamic Relocation
- Chap 16. Segmentation
  - ✓ Generalization, Sharing, Protection
- Chap 17. Free-Space Management
  - ✓ Fragmentation, Splitting and Coalescing
  - ✓ Strategies: Best fit, First fit, Worst fit, …
  - ✓ Segregated list, Buddy algorithm, …

Chap 12. Dialogue

- Memory virtualization

Student: *So, are we done with virtualization?*

Professor: *No!*

Student: *Hey, no reason to get so excited; I was just asking a question. Students are supposed to do that, right?*

Professor: *Well, professors do always say that, but really they mean this: ask questions,* ***if*** *they are good questions,* ***and*** *you have actually put a little thought into them.*

Student: *Well, that sure takes the wind out of my sails.*

Professor: *Mission accomplished. In any case, we are not nearly done with virtualization! Rather, you have just seen how to virtualize the CPU, but really there is a big monster waiting in the closet: memory. Virtualizing memory is complicated and requires us to understand many more intricate details about how the hardware and OS interact.*

Student: *That sounds cool. Why is it so hard?*

Professor: *Well, there are a lot of details, and you have to keep them straight in your head to really develop a mental model of what is going on. We'll start simple, with very basic techniques like base/bounds, and slowly add complexity to tackle new challenges, including fun topics like TLBs and multi-level page tables. Eventually, we'll be able to describe the workings of a fully-functional modern virtual memory manager.*

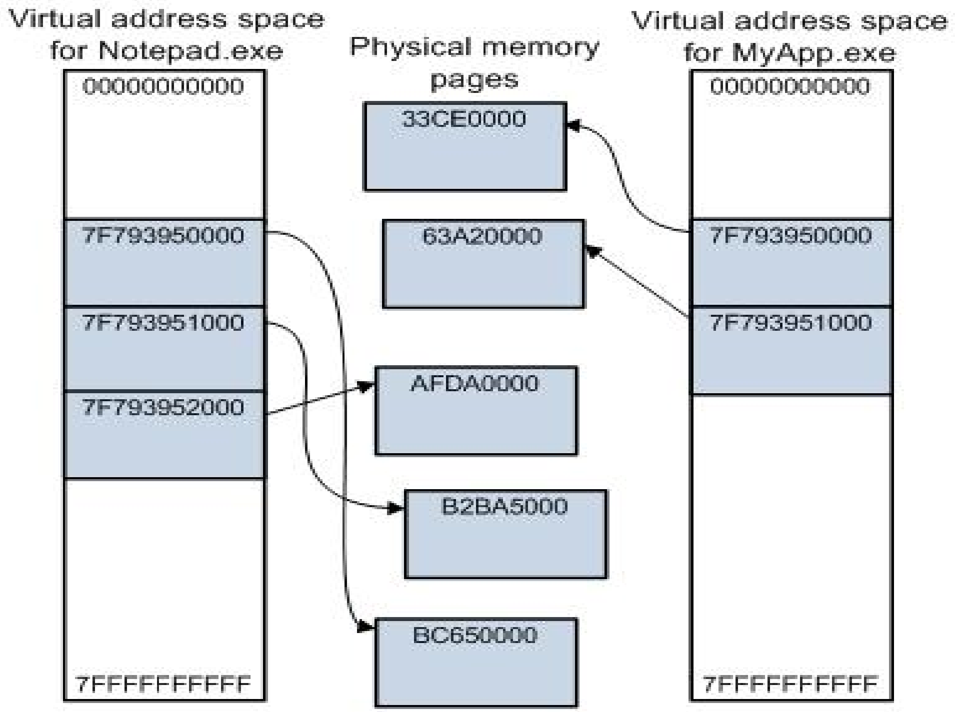
Student: *Neat! Any tips for the poor student, inundated with all of this information and generally sleep-deprived?*

Professor: *For the sleep deprivation, that's easy: sleep more (and party less). For understanding virtual memory, start with this:* ***every address generated by a user program is a virtual address****. The OS is just providing an illusion to each process, specifically that it has its own large and private memory; with some hardware help, the OS will turn these pretend virtual addresses into real physical addresses, and thus be able to locate the desired information.*

- Address space (Large and Private), Virtual/Physical Address, Address Translation, Isolation,…

Chap 13. The abstraction: address space

- Early system
- Multiprogramming and Time sharing
- Address space
- Goals



(Source: https://msdn.microsoft.com/en-us/windows/hardware/drivers/gettingstarted/virtual-address-spaces)

13.1 Early Systems

- Use physical memory directly
  - ✓ OS and current program ➔ single programming system
  - ✓ No (limited) protection
  - ✓ Larger program than physical memory ➔ Overlay

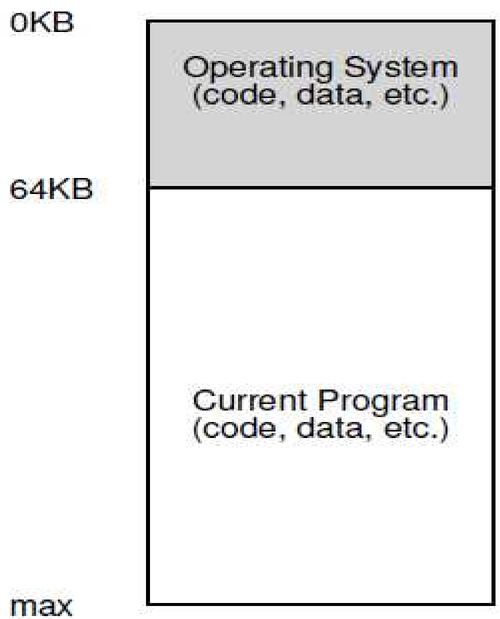


Figure 13.1: **Operating Systems: The Early Days**

13.2 Multiprogramming and Time sharing

- Computer becomes bigger
  - ✓ Multiprogramming: multiple processes are ready to run
  - ✓ Time sharing: switch CPUs among ready processes
  - ✓ Issues
    - Protection becomes a critical issue
    - How to find suitable free space

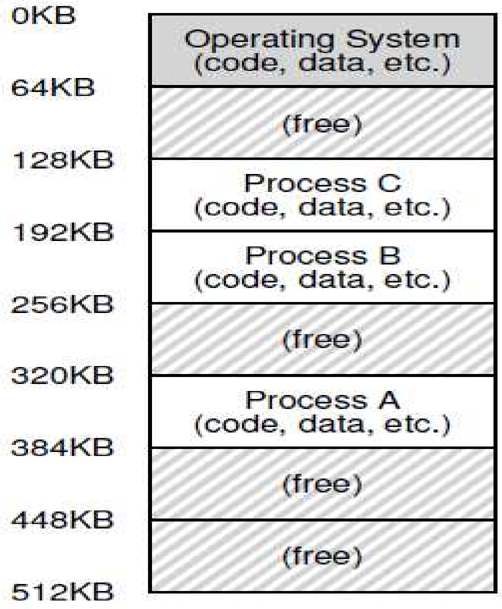
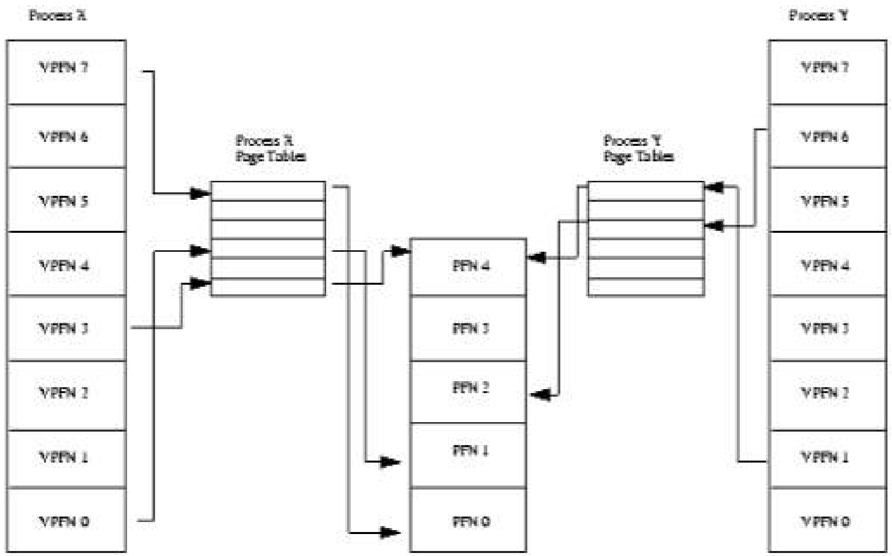


Figure 13.2: **Three Processes: Sharing Memory**

13.3 Address space

- Abstraction
  - ✓ A process has an illusion that it uses exclusively all memory even though it is shared by multiple processes ➔ virtual memory
  - ✓ Well defined layout ➔ address space
    - Code (instruction), Data (statically-initialized variables), Stack (function call chain and local variables), Heap (dynamically allocated)
    - Code is located at virtual address 0x0, but not physically



Virtual memory **Physical memory** **Virtual memory**

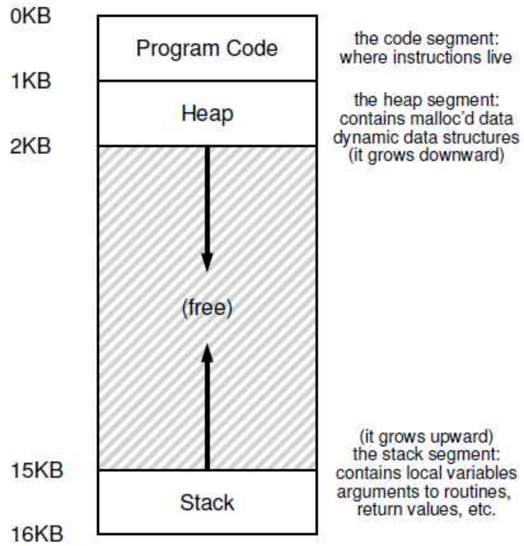


Figure 13.3: **An Example Address Space**

13.4 Goals

- Transparency (easy to use)
  - ✓ Programmer: no need to aware the memory size or available space
- Efficiency
  - ✓ Both in terms of time and space (not slow and not requires much additional overhead) ➔ Various HW support (e.g. TLB)
- Protection (isolation)
  - ✓ Protect processes from one another
- Note: every address you see is virtual

```
#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;
}
```

When run on a 64-bit Mac OS X machine, we get the following output:

```
location of code  : 0x1095afe50
location of heap  : 0x1096008c0
location of stack : 0x7fff691aea64
```

Chap 14. Interlude: Memory API

- Types of Memory
- The malloc() call
- The free() call
- Common errors
- Underlying OS Support
- Other Calls

14.1 Types of Memory

- Two types of memory
  - ✓ Static: Code (also called as text), Data
  - ✓ Dynamic: Heap, Stack
    - Stack
      - Implicitly by the compiler (hence sometimes called automatic memory)
      - Short-lived memory

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

    - Heap
      - Explicitly by the programmer
      - (relatively) Long-lived memory

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

14.2/3 The malloc()/free() call

- The malloc() call
  - ✓ Input: memory size (how many bytes you need)
  - ✓ Output: pointer to the newly-allocated space (or NULL if it fails)

  - ✓ Use well-defined macros or routines, instead of number as input
    - ✓ malloc(sizeof(int));
    - ✓ malloc(strlen(s) + 1);

- The free() call
  - ✓ Input: pointer (size is not specified, meaning that it is managed by the library)

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

14.4 Common errors

- Common errors
  - ✓ Forgetting to allocate memory

```
char *src = "hello";
char *dst;           // oops! unallocated
strcpy(dst, src); // segfault and die
```

  - Correct version (or strdup())

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

  - We frequently meet the segmentation fault. Hence ➔

> When you run this code, it will likely lead to a segmentation fault[3], which is a fancy term for YOU DID SOMETHING WRONG WITH MEMORY YOU FOOLISH PROGRAMMER AND I AM ANGRY.

☛ Make use of a debugger (e.g. gdb)

14.4 Common errors

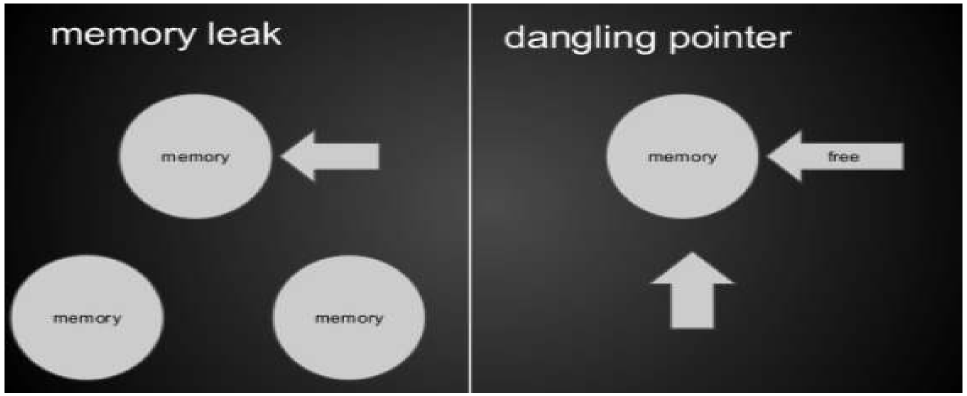
- Common errors
  - ✓ Not allocating enough Memory

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

    - It seems work, but not correctly ('\0'), which causes buffer overflow, leading to several security vulnerabilities.
    - Some library allocates a little extra space.
  - ✓ Forgetting to initialize allocated memory
    - Heap has data of unknown value.
  - ✓ Forgetting to free memory
    - Memory leak
    - Some languages support the garbage collection mechanism that manages memory automatically without requiring explicit free() by programmers → but if you still have a reference, the collector will never free it (still problem)

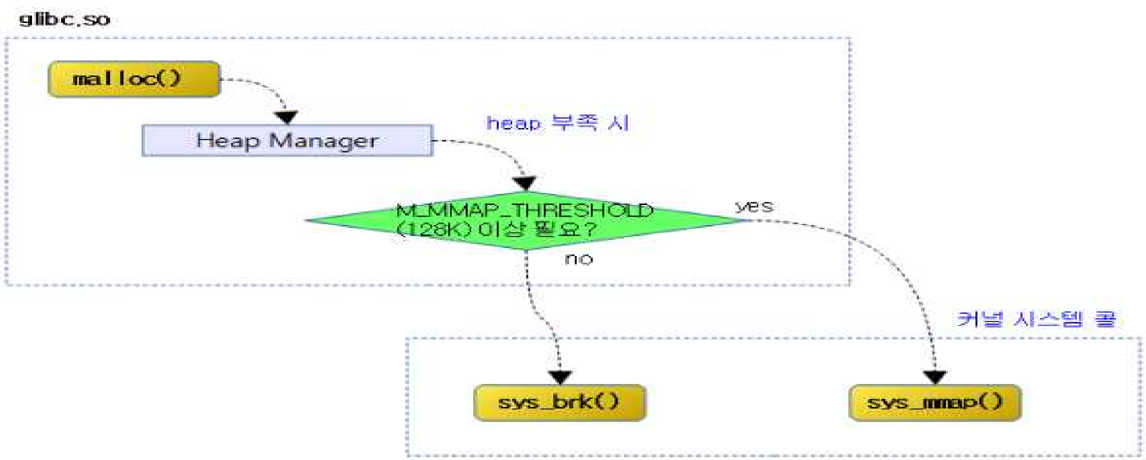
14.4 Common errors

- Common errors
  - ✓ Freeing memory before you are done with it
    - Dangling pointer
    - Subsequent use can crash the program and even system
  - ✓ Freeing memory repeatedly
    - Double free
  - ✓ Calling free() incorrectly
    - Invalid free
- Tools for solving memory-related problems
  - ✓ Purify
  - ✓ Valgrind
  - ✓ …



14.5/6 Underlying OS Support/Other Calls (Optional)

- Underlying OS Support
  - ✓ malloc()/free() ➔ library
  - ✓ It internally allocates several pages using the sys_brk() or sys_mmap() system call and manages them to serve the malloc() and free() request
  - ✓ If its space becomes too small, it requests more pages to OS again using the sys_brk() or sys_mmap() ➔ system call
- Other Calls
  - ✓ calloc(): allocate and zero space
  - ✓ realloc(): allocate a new larger region, copy the old region into it and returns the pointer of the new region



Chap. 15 Mechanism: Address Translation

- CPU virtualization
  - ✓ Limited Direct Execution
    - Direct execution: process run independently for the most time (efficiency)
    - Limited: OS get involved (control)
  - ✓ Two mechanisms
    - Restricted operations (e.g. system call): user mode ➔ kernel mode (OS control)
    - Timer interrupt: user mode ➔ kernel mode (OS control), do periodic jobs such as scheduling and context switch
- Memory virtualization
  - ✓ Address Translation
    - Address space: virtual memory (using virtual address)
    - During execution: physical memory (using physical address which is translated from virtual address)
  - ✓ Again, we will pursue both efficiency and control
    - Efficiency: small overhead ➔ hardware-based address translation
    - Control: OS ensures that no processes is allowed to access any memory but its own ➔ OS memory management

15.1 Assumption/15.2 An Example

- A program
  - ✓ High-level viewpoint

```
void func() {
    int x = 3000; // thanks, Perry.
    x = x + 3;    // this is the line of code we are interested in
```

  - ✓ Assembly viewpoint

```
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
```

  - ✓ Process viewpoint (address space)
    - Instructions: address 128 ~ 135 at code
    - Variable x: address 15KB (15,360B) at stack
  - ✓ Execution viewpoint (fetch + execution)
    - Fetch instruction at address 128
    - Execute this instruction (load from address 15 KB)
    - 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)

☛ Need to access memory (128, 15KB, ...) during execution

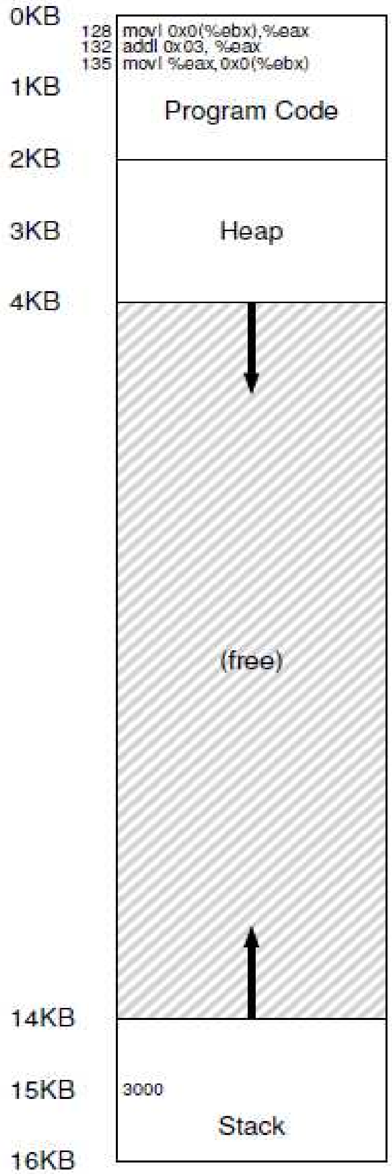


Figure 15.1: A Process And Its Address Spa

15.1 Assumption/15.2 An Example

- Focusing on memory
  - ✓ Address space (virtual memory)
    - Starts at address 0
    - Grows to maximum of 16 KB
  - ✓ Physical memory
    - Can place any free space, not necessarily at address 0 ➔ relocation
  - ✓ Address translation
    - Assume that the process is located at 32KB~48KB in physical memory
    - Then the virtual address 0 needs to be translated into the physical address 32KB ➔ address translation
  - ✓ Other slots (e.g. 16~32KB) are not used ➔ free space management
  - ✓ OS also locates in physical memory

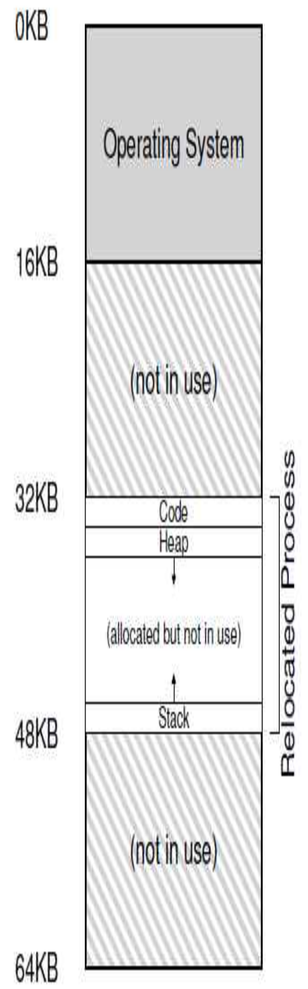


Figure 15.2: Physical Memory with a Single Relocated Process

15.3 Dynamic (Hardware-based) Relocation

- Integration of Virtual and Physical memory
  - ✓ Virtual memory: 0~16KB vs Physical Memory: 0~64KB
    - Being loaded into 32KB~48KB
  - ✓ Address translation: virtual address ➔ physical address
    - First instruction: 128 ➔ 32KB + 128 (32768 + 128 = 32896)
    - Variable x: 15KB ➔ 32KB + 15KB = 47KB
    - In general: base address + offset (instruction or variable's address)

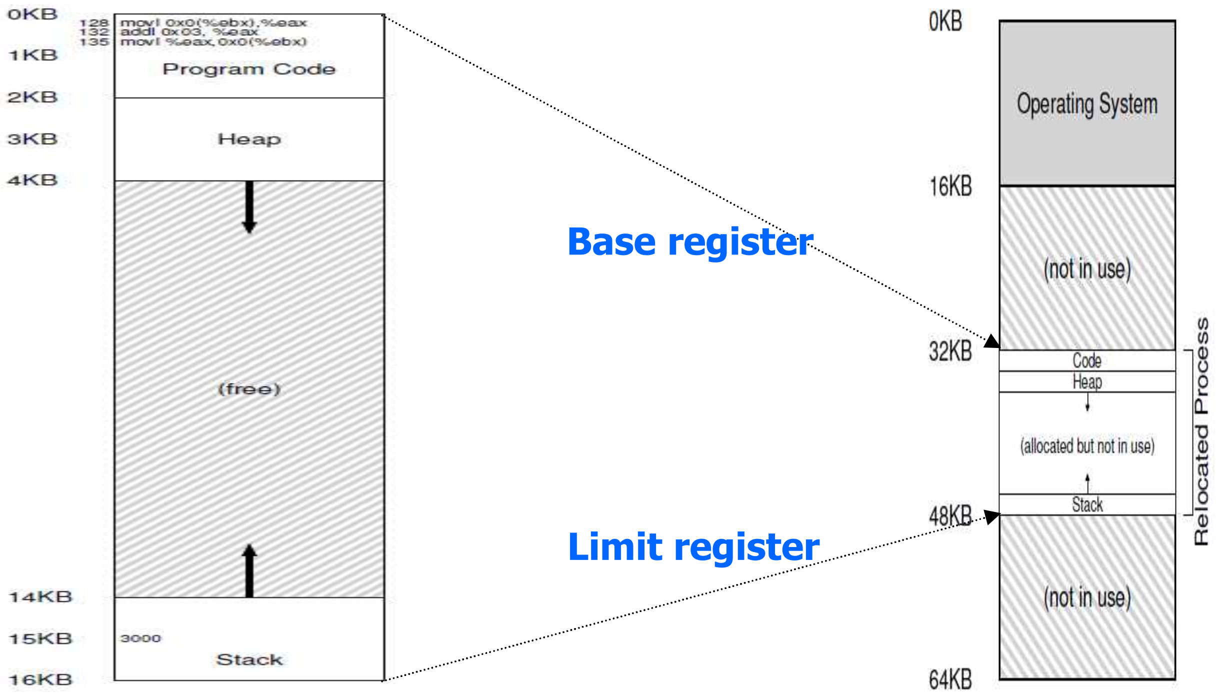


Figure 15.1: A Process And Its Address Space

Figure 15.2: Physical Memory with a Single Relocated Process

☛ What if a virtual address is larger than the limit register?

Quiz for 12th-Week 1st-Lesson

- Quiz
  - ✓ 1. Discuss the definition of the following terms using the below figure : 1) virtual memory, 2) physical memory, 3) address space, 4) address translation, and 5) relocation.
  - ✓ 2. What are the physical addresses when we access the virtual addresses of 1KB, 3KB, 15KB and 17KB in the below figure.
  - ✓ Due: until 6 PM Friday of this week (21th, May)

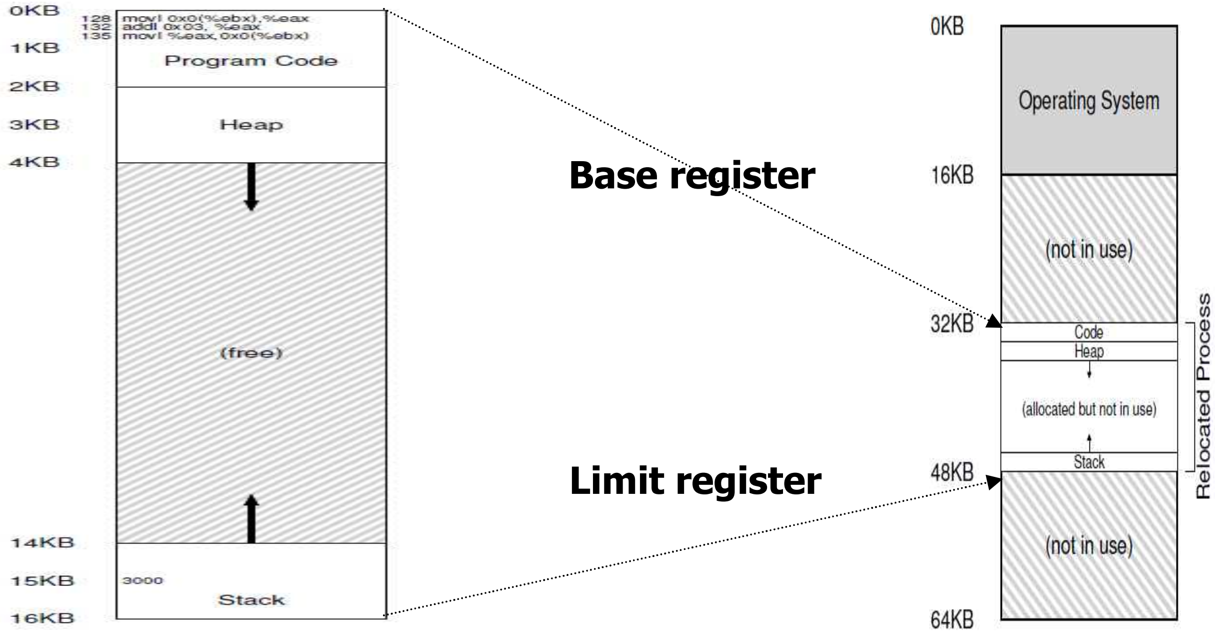


Figure 15.1: A Process And Its Address Space

Figure 15.2: Physical Memory with a Single Relocated Process

15.3 Dynamic (Hardware-based) Relocation

- Summary of address translation (and relocation)
  - ✓ Virtual memory: per process (exclusive), start at 0x0 (size: 16KB)
  - ✓ Physical memory: shared by processes, start at any address (different among processes)
  - ✓ Three main components: Compiler, OS and Hardware (Architecture)
    - A program is compiled as if it is loaded at address 0 (virtual memory).
    - The program is loaded any space in physical memory, while setting base and limit registers appropriately ➔ relocatable
    - An address requested by CPU is translated into a physical address while running (and protected)

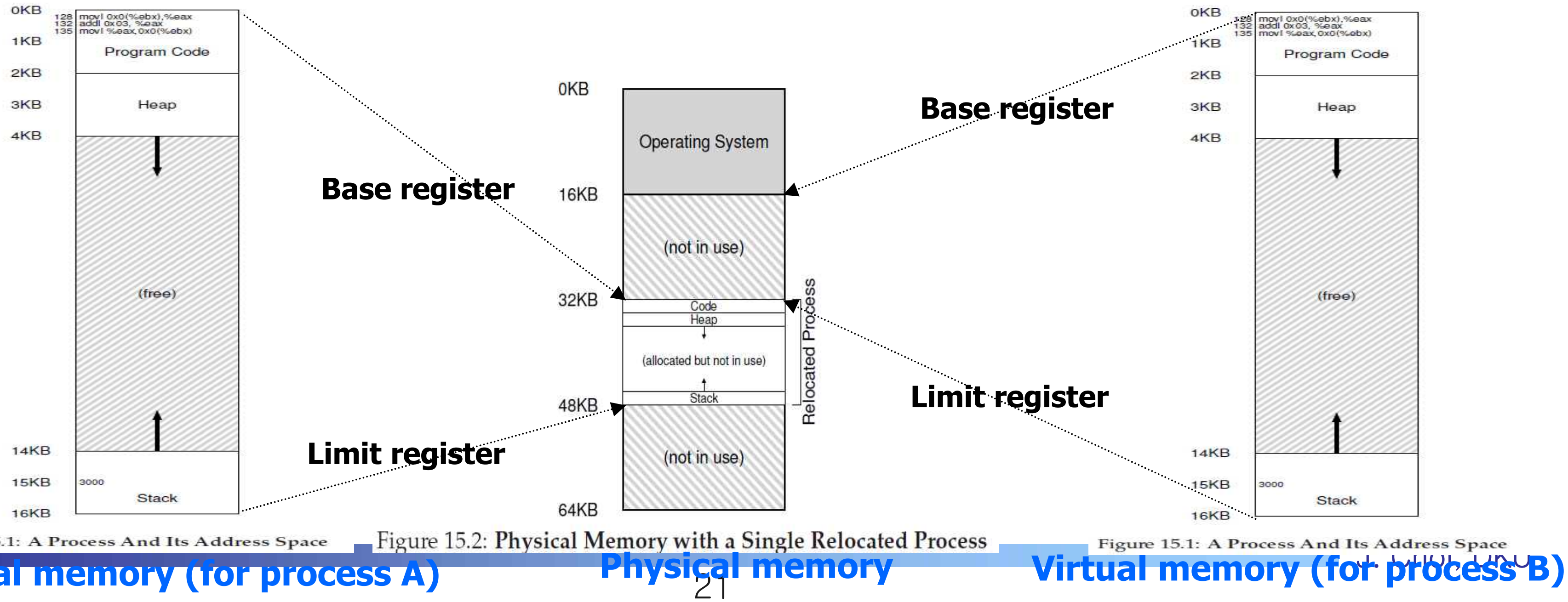
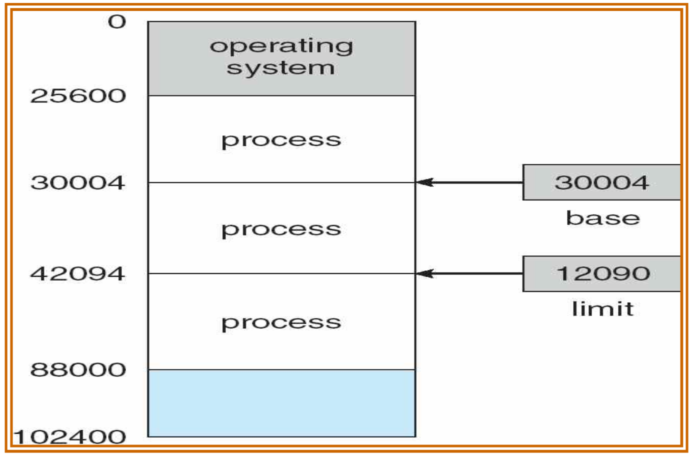
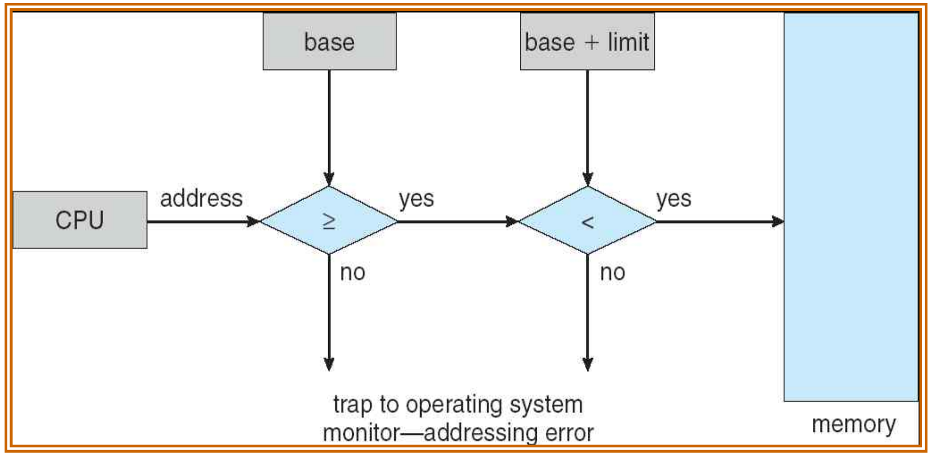


Figure 15.1: A Process And Its Address Space
Figure 15.2: Physical Memory with a Single Relocated Process
Figure 15.1: A Process And Its Address Space

15.3 Dynamic (Hardware-based) Relocation

- Summary of address translation (and relocation)
  - ✓ How to translate? Using two hardware registers
    - Base register: start address (30004 in this example)
      - physical address = base register + virtual address
      - E.g. virtual address = 10 ➔ physical address = 30014
    - Limit register (Bound register): upper bound (or size, 12090 in this example)
      - E.g. virtual address = 13000 ➔ segmentation fault
    - Base/Limit registers are switched at each context switch time
      - E.g. base register: 30004 ➔ 25600





(Source: A. Silberschatz, “Operating system Concept”)

15.4 Hardware Support: A Summary

- MMU (Memory management unit)
  - ✓ Part of CPU that helps with address translation
  - ✓ E.g.) Base/limit registers, Segmentation related registers, Paging related registers, TLB (Translation Lookaside Buffer) + circuitry
- Summary of HW support for Dynamic relocation

Hardware Requirements	Notes
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

Figure 15.3: **Dynamic Relocation: Hardware Requirements**

15.5 Operating Systems Issues

- OS responsibilities
  - ✓ Memory management
    - Allocation for new processes, free list manipulation, …
    - Reclaim the space of terminated processes
  - ✓ Base/limit registers management during Context switch
    - Save/restore base/limit registers into/from PCB (MMU)
    - Process relocation if necessary
  - ✓ Exception handling
    - Handlers + Table (e.g. segmentation fault handler + IVT)

OS Requirements	Notes
Memory management	*Need to allocate memory for new processes; Reclaim memory from terminated processes; Generally manage memory via* ***free list***
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise; likely action is to terminate offending process*

Figure 15.4: **Dynamic Relocation: Operating System Responsibilities**

15.5 Operating Systems Issues

- Global view

Initialization

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember addresses of... system call handler timer handler illegal mem-access handler illegal instruction handler
start interrupt timer	
	start timer; interrupt after X ms
initialize process table initialize free list	

Direct execution

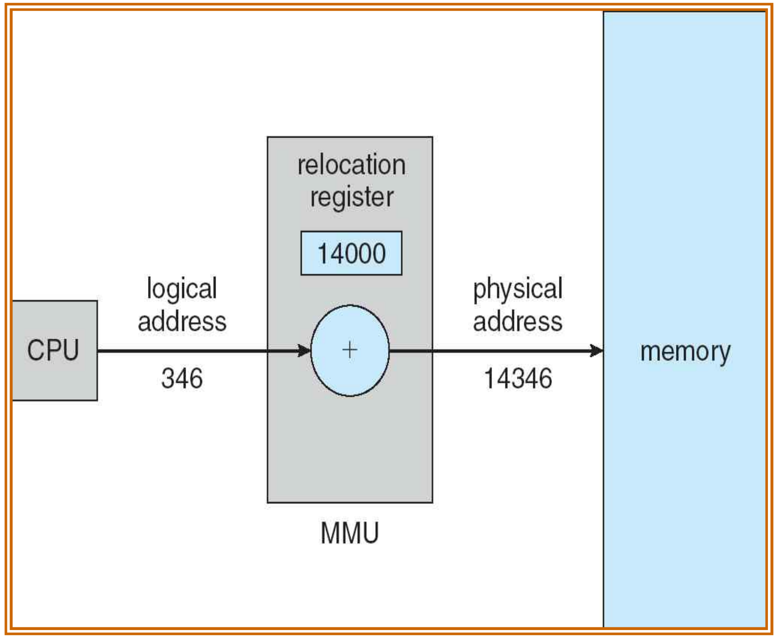
OS involved

OS @ run (kernel mode)	Hardware	Program (user mode)
To start process A: allocate entry in process table allocate memory for process set base/bounds registers **return-from-trap** (into A)		
	restore registers of A move to **user mode** jump to A's (initial) PC	
		Process A runs Fetch instruction
	Translate virtual address and perform fetch	
		Execute instruction
	If explicit load/store: Ensure address is in-bounds; Translate virtual address and perform load/store	
		...
	Timer interrupt move to **kernel mode** Jump to interrupt handler	
Handle the trap Call switch() routine save regs(A) to proc-struct(A) (including base/bounds) restore regs(B) from proc-struct(B) (including base/bounds) **return-from-trap** (into B)		
	restore registers of B move to **user mode** jump to B's PC	
		Process B runs Execute bad load
	Load is out-of-bounds; move to **kernel mode** jump to trap handler	
Handle the trap Decide to terminate process B de-allocate B's memory free B's entry in process table		

Figure 15.6: Limited Direct Execution (Dynamic Relocation) @ Runtime

15.6 Summary

- Memory virtualization
  - ✓ Address translation
    - OS: memory allocation/free, base/limit initialize, exception control (infrequent event)
    - HW: virtual to physical at every execution (frequent event, MMU)
    - Support transparency: users have no idea where their processes are
  - ✓ Mechanisms
    - Contiguous allocation
      - 1) Base and limit registers
      - Pros: Simple and Offer protection
      - Cons: Internal fragmentation
    - Non-contiguous allocation
      - 2) Segmentation: Variable size
      - 3) Paging: Fixed size



(Source: A. Silberschatz, "Operating system Concept")

Chap. 16 Segmentation

- Issues of the base/limit register based dynamic relocation
  - ✓ A big chunk of "free" space in the middle of address space
    - Even though they are free, they are taking up physical memory
  - ✓ Hard to run a program when the entire address space does not fit into an available space in physical memory

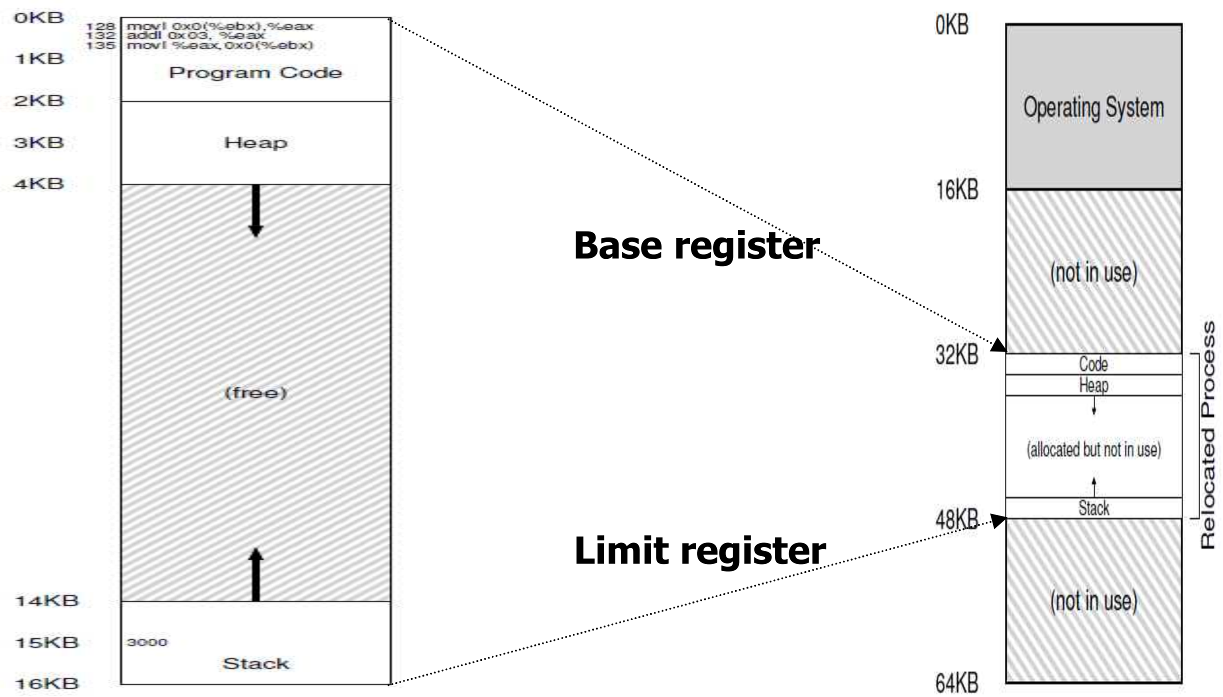


Figure 15.1: A Process And Its Address Space

Figure 15.2: Physical Memory with a Single Relocated Process

☛ How large the free space between heap and stack in 32-bit CPU?

16.1 Segmentation: Generalized Base/Limits

- Key idea
  - ✓ Contiguous ➔ Non-contiguous
  - ✓ Segment: divide a program into multiple segments (each segment is a contiguous portion of the address space)
    - E.g.) code segment, data segment, stack segment, heap segment, …
  - ✓ Support base/limit per segment
    - OS places segments independently in physical memory

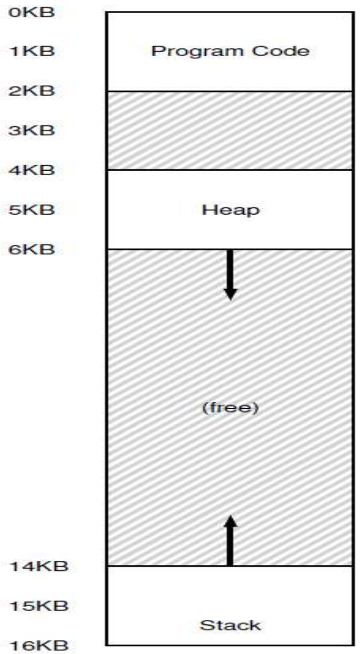


Figure 16.1: An Address Space (Again)

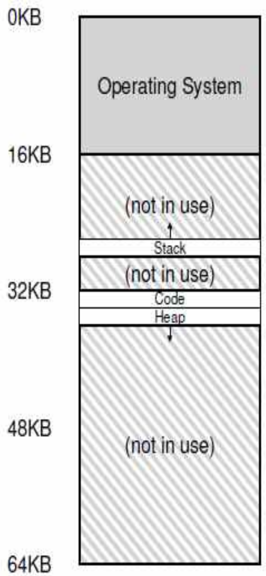


Figure 16.2: Placing Segments In Physical Memory

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

Figure 16.3: Segment Register Values

16.1 Segmentation: Generalized Base/Limits

- Address translation
  - ✓ virtual address 100 (e.g. PC) ➔ physical address: 32KB + 100
  - ✓ virtual address 4200 (e.g. pointer x) ➔ physical address 34K + 104
  - ✓ virtual address 7000 (or 3000) ➔ segmentation fault
  - ✓ virtual address: segment number + offset
    - Segment number: choose appropriate segment register (or table entry)
    - Offset: location within the segment (assume that it begins with 0)

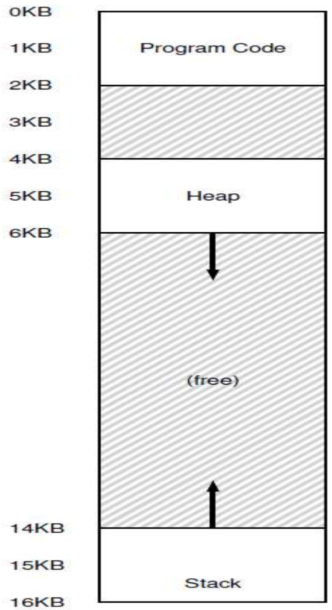


Figure 16.1: An Address Space (Again)

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

Figure 16.3: Segment Register Values

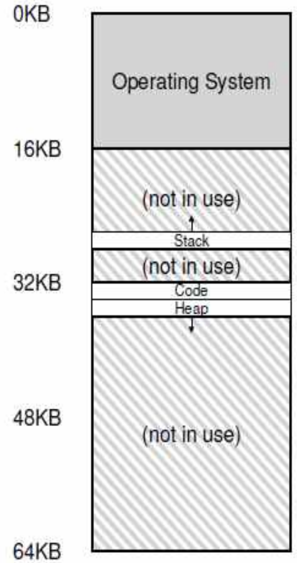
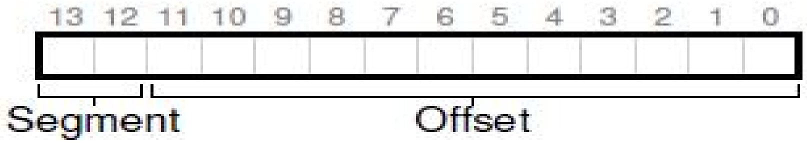


Figure 16.2: Placing Segments In Physical Memory

16.2 Which Segmentation Are We Referring To?

- Segment encoding in virtual address
  - ✓ Segment number part + offset part
  - ✓ In the previous example
    - Address space size: 16KB = 2^14 ➔ 14 bit
    - Number of segment: 3 ➔ 2 bit
    - Number of offset: remaining 12 bit ➔ maximum size of a segment: 4KB



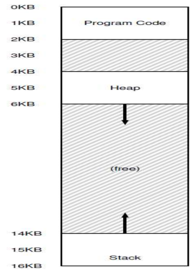
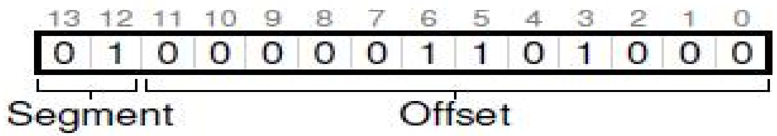


Figure 16.1: An Address Space (Again)

    - Segment: 00 ➔ code, 01 ➔ heap, 11 ➔ stack
    - virtual address 4200 = 4096 + 64 + 32 + 8



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

Figure 16.3: Segment Register Values

      - Segment number: Used for searching its related base register
      - Offset: If this offset is larger than the limit, trigger the segmentation fault. Otherwise, add offset with the value of the base register, generating the physical address (4200 ➔ "01 (heap) + 104" ➔ 34K + 104)

☛ How about the virtual addresses 100 and 7000 discussed in the previous slide?

Quiz for 12th-Week 2nd-Lesson

- Quiz
  - ✓ 1. Discuss the roles of 1) compiler, 2) operating system, and 3) CPU (or HW) for memory virtualization (hint: 21 page).
  - ✓ 2. Using the below left figure, explain the physical addresses of the virtual addresses of 100, 3000 and 5000 (using the terms of segment number and offset)
  - ✓ Bonus. Discuss the values of SEG_MASK, SEG_SHIFT and OFFSET_MASK in the below right figure (hint: see 5 page in the OSTEP)
  - ✓ Due: until 6 PM Friday of this week (21th, May)

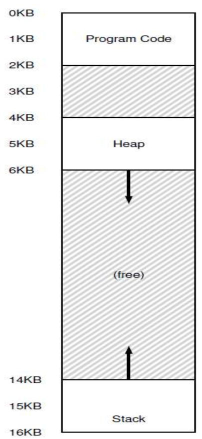


Figure 16.1: An Address Space (Again)

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

Figure 16.3: Segment Register Values

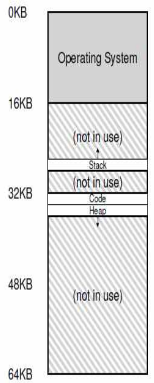
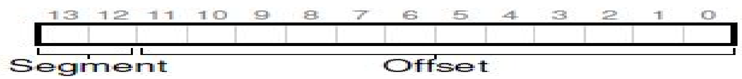


Figure 16.2: Placing Segments In Physical Memory



Pseudo code for address translation in segmentation

```
// 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)
```

16.3 What About the Stack?

- Stack issue
  - ✓ It grows backward ➔ translation must proceed differently
    - Need extra HW support

Segment	Base	Size (max 4K)	Grows Positive?
$Code_{00}$	32K	2K	1
$Heap_{01}$	34K	3K	1
$Stack_{11}$	28K	2K	0

Figure 16.4: **Segment Registers (With Negative-Growth Support)**

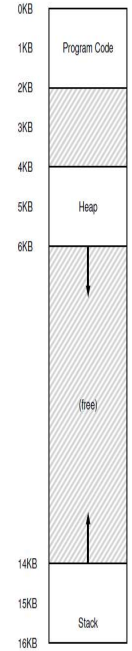


Figure 16.1: An Address Space (Again)

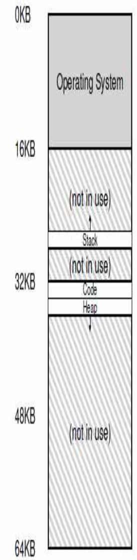


Figure 16.2: Placing Segments In Physical Memory

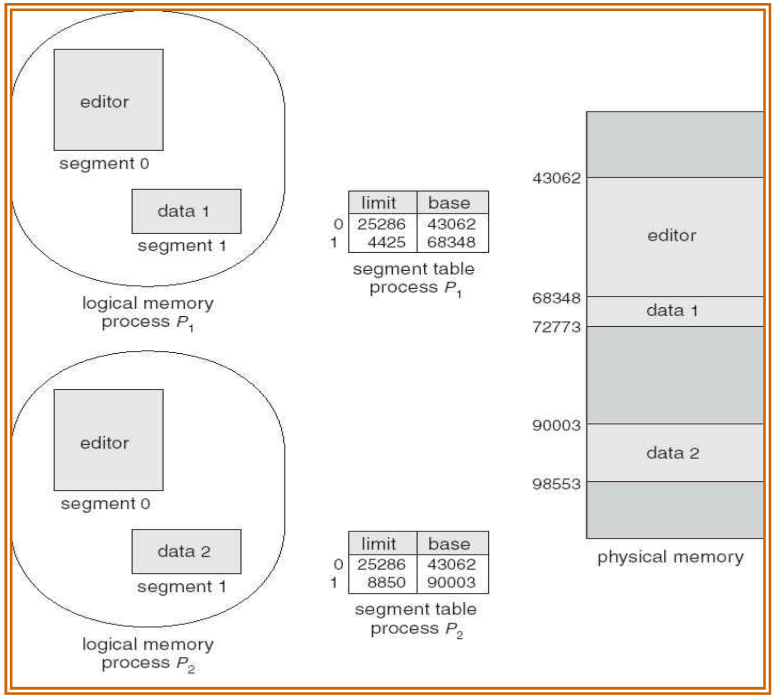
  - ✓ Instead of offset, adding “virtual address - total address space size” (or “offset in stack - maximum segment size”) with the value in base register
    - Virtual address: 15KB = 11 1100 0000 0000
      - Segment number 11 ➔ stack
      - Offset 1100 0000 0000 ➔ 3KB
    - Physical address: 28KB + (15KB - 16KB) or 28KB + (3KB – 4KB) ➔ 27KB
    - Another example: 16KB – 4B = 16380 = b 11 1111 1111 1100 ➔ seg. Number = 11 + offset = 1111 1111 1100 = 4902 ➔ physical address = 28KB + (4902B – 4096B) = 28KB – 4B

16.4/5 Support for Sharing/ Granularity

- Benefit of segmentation
  - ✓ Sharing among multiple processes
  - ✓ Protection support

Segment	Base	Size (max 4K)	Grows Positive?	Protection
$Code_{00}$	32K	2K	1	Read-Execute
$Heap_{01}$	34K	3K	1	Read-Write
$Stack_{11}$	28K	2K	0	Read-Write

Figure 16.5: **Segment Register Values (with Protection)**



(Source: A. Silberschatz, "Operating system Concept")

- Segment size
  - ✓ Coarse-grained
    - Relatively large size, small # of segments in a process (around 4)
  - ✓ Fine-grained
    - Relatively small size, large # of segments in a process
    - Make use of a table (segment table) for manipulating large # of segments.

16.6 OS Support

- For segmentation support
  - ✓ Context switch: save/restore segment related registers
  - ✓ Free space management
    - Try to reduce external fragmentation ➔ coalescing and compaction

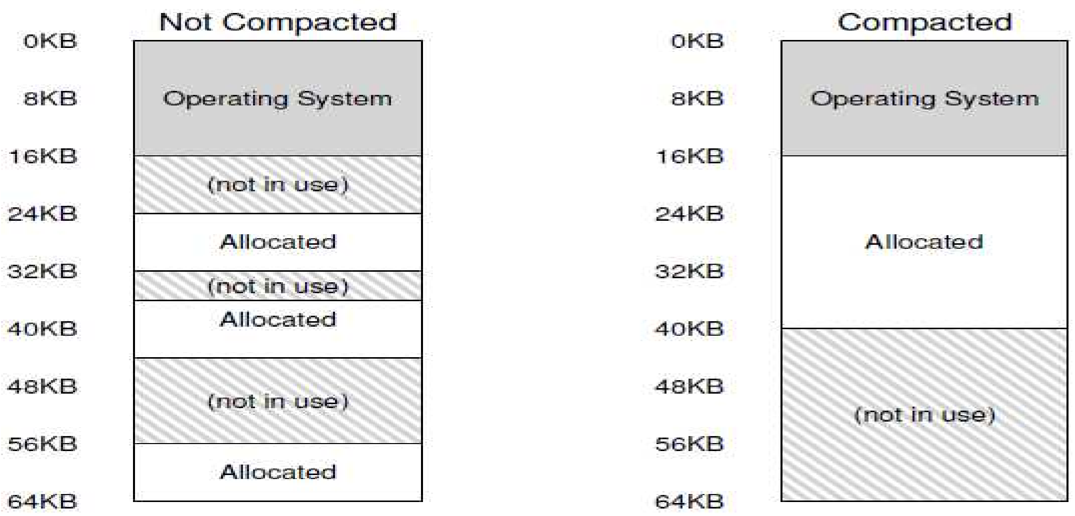


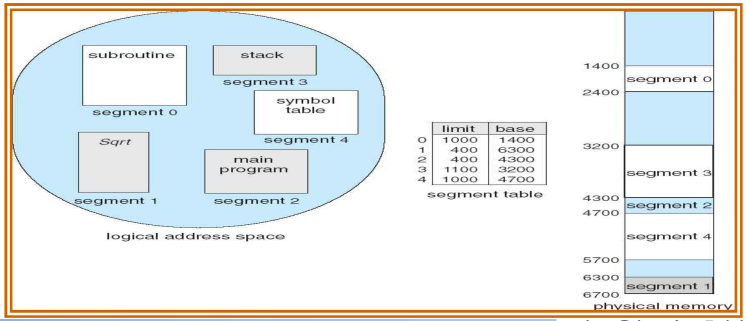
Figure 16.6: Non-compacted and Compacted Memory

  - ✓ Allocation
    - Best-fit, worst-fit, first-fit, buddy algorithm (➔ see chapter 17)

☛ Compaction in memory: prepare for large free space vs Compaction in disks: reduce seek time

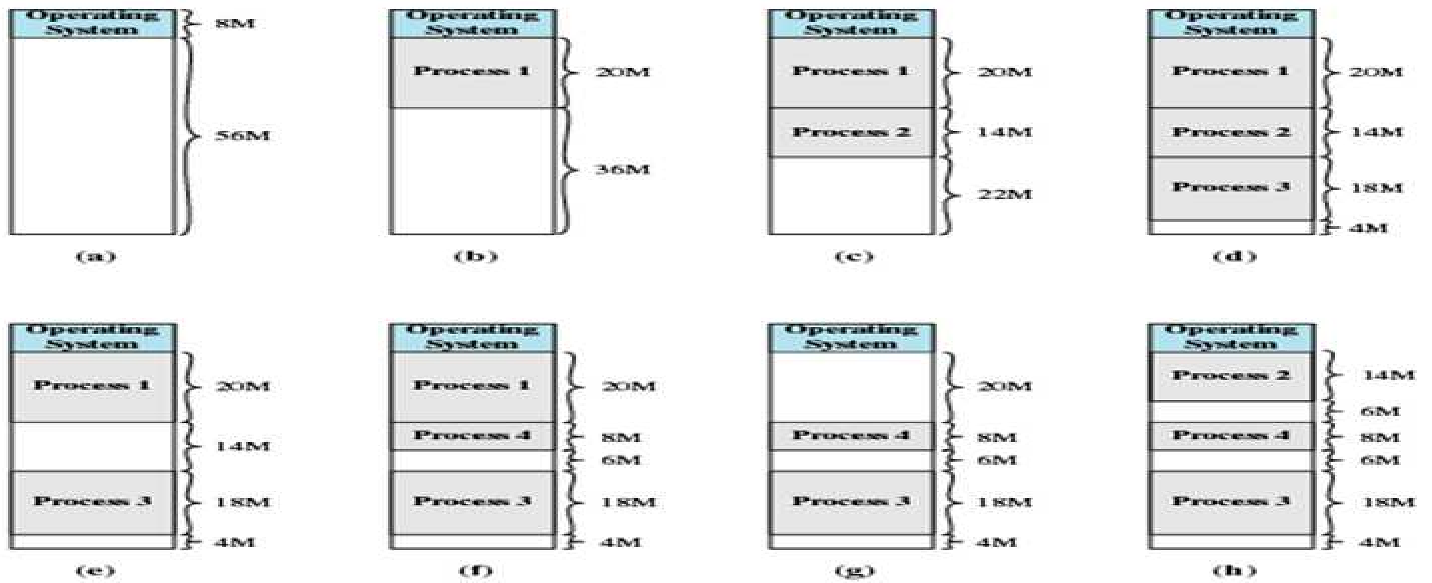
16.7 Summary

- Segmentation
  - ✓ Divide address space into logical regions called segment
  - ✓ Overcome the memory wasted between segments (e.g. heap and stack in the base/limit mechanism)
  - ✓ Flexible: support sharing and protection
- But, still have some problems
  - ✓ Variable size ➔ relatively hard to implement in hardware, may cause external fragmentation which complicate free space management
  - ✓ Memory waste within a segment, especially sparse segment ➔ need to allocate address space that are actually used by a process
  - ✓ Alternative: fixed size ➔ Paging (chap 18.)



Chap. 17 Free Space Management

- Free-space management
  - ✓ Variable size (e.g. malloc() or segmentation)
    - Complicate, need to handle external fragmentation ➔ in this chapter
  - ✓ Fixed size (e.g. paging)
    - Relatively easy, usually a list of free fixed-size units ➔ later chapters



(Source: A. Silberschatz, "Operating system Concept")

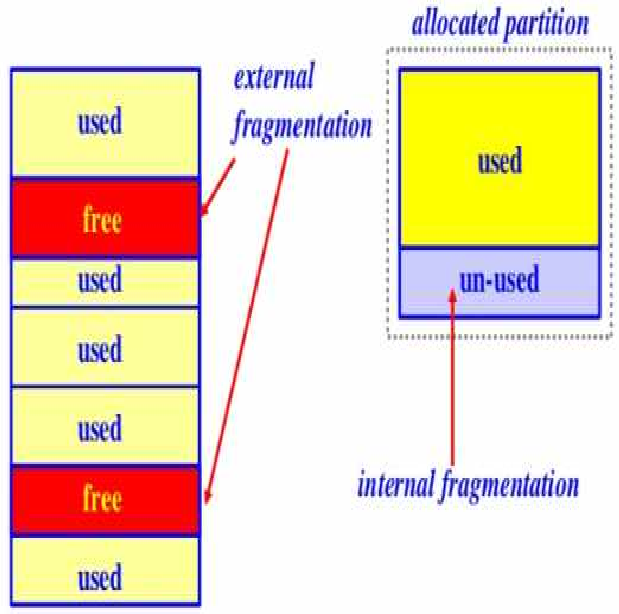
☞ **Process 2 is "relocated" dynamically**

☞ **Need the swap space (in a disk) when a process is suspended.**

☞ **How to handle when a new process is forked at (h) step whose size is 3 or 10MB?**

17.1 Assumptions

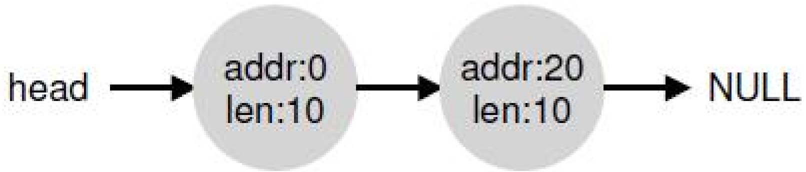
- Interfaces
  - ✓ malloc()/free()
- Free space
  - ✓ Managed by a list (free list)
  - ✓ In actual OSes, free space is managed by various data structures including a hashed list or tree (e.g. buddy system)
- Fragmentation
  - ✓ External: variable-size allocation
  - ✓ Internal: fixed-size allocation
  - ✓ Focus on external fragmentation



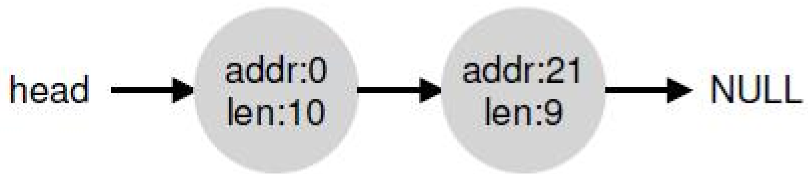
17.2 Low-level Mechanisms

- Splitting and Coalescing
  - ✓ Memory: 30-byte heap

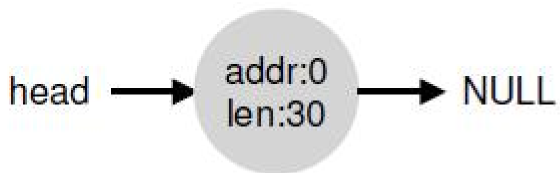
  - ✓ Free list



  - ✓ Request
    - 10B ➔ allocate one of the free entries
    - Larger than 10B ➔ fail or need compaction
    - Smaller than 10B ➔ need splitting
      - Allocate 1 byte



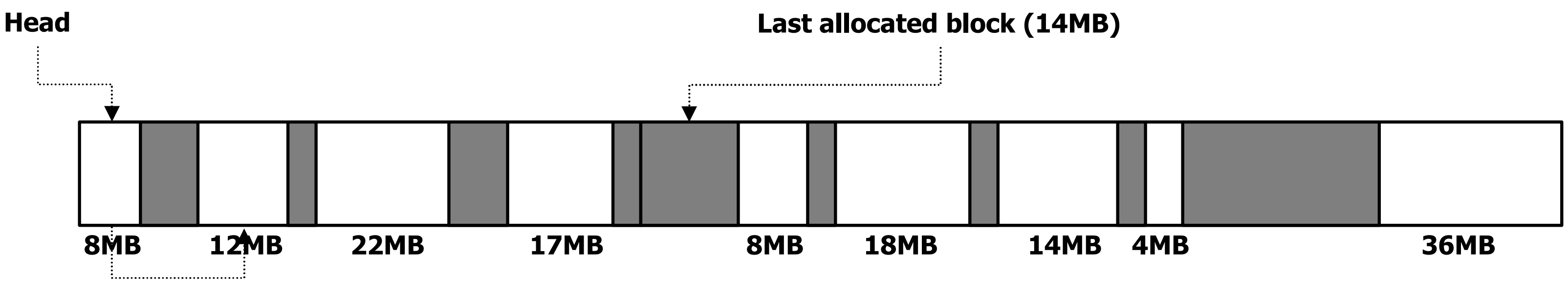
  - ✓ Free
    - Free the used space 10~19 ➔ need coalescing
      - Sort free entries, check neighbors when inserting into the free list



☞ **See appendix (page 43~45) and 17.2 in OSTEP for real free space management.**

17.3 Basic Strategies

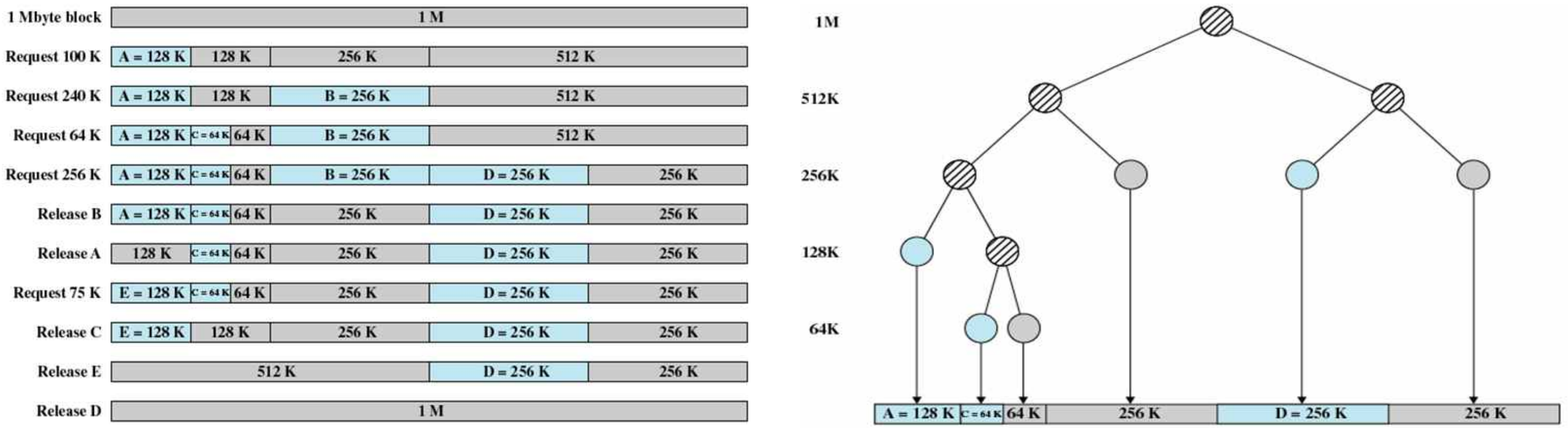
- Free-space allocation policy
  - ✓ Best-fit
    - allocate from the smallest chuck which is bigger than the request size
  - ✓ Worst-fit
    - allocate from the largest chuck which is bigger than the request size
  - ✓ First-fit
    - allocate from the first chuck which is bigger than the request size, search start from head
  - ✓ Next-fit
    - allocate from the first chuck which is bigger than the request size, search start from the last allocated chunk



☞ **Need to allocate 16MB available space. Which one by each policy?**

17.4 Other Approaches

- Buddy allocation
  - ✓ To make splitting/coalescing simple
  - ✓ Allocate a free memory with the size of 2^n (e.g. 4KB, 8KB, …)
- Segregated Lists
  - ✓ Some applications have one (or a few) popular-sized request
  - ✓ Manage them in a segregated list ➔ same size ➔ easier to split and coalescing
  - ✓ Popular example: slab allocator in Solaris (and in Linux)
- Others
  - ✓ More complex data structure for fast searching (e.g. balanced B-tree)



(Source: A. Silberschatz, "Operating system Concept")

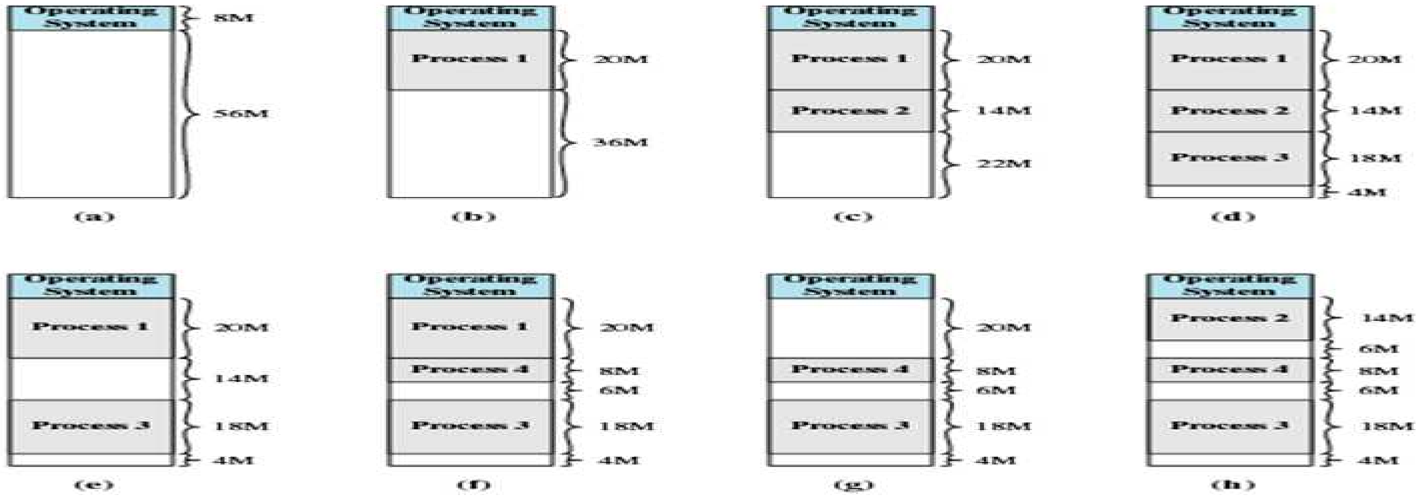
17.5 Summary

- Memory virtualization
  - ✓ Goal: Transparency, isolation, efficiency
  - ✓ Virtual memory (Address space) and Physical memory
  - ✓ Address translation: virtual to physical address
- Dynamic relocation
  - ✓ Base & Limit (Bound) approach
  - ✓ Generalized approach ➔ segmentation
- Free-Space Management
  - ✓ Reduce fragmentation (external/internal)
  - ✓ Mechanism: Splitting, Coalescing and Compaction
  - ✓ Policy: Best fit, First fit, Worst fit, Buddy algorithms, Slab, …
  - ✓ ➔ Variable size makes management complex (1000 solutions)

> **TIP: IF 1000 SOLUTIONS EXIST, NO GREAT ONE DOES**
> The fact that so many different algorithms exist to try to minimize external fragmentation is indicative of a stronger underlying truth: there is no one "best" way to solve the problem. Thus, we settle for something reasonable and hope it is good enough. The only real solution (as we will see in forthcoming chapters) is to avoid the problem altogether, by never allocating memory in variable-sized chunks.

Quiz for 13th-Week 1st-Lesson

- Quiz
  - ✓ 1. Discuss the following terms using the below figure : 1) swap out (also called as "suspend" in LN 2), 2) relocation, 3) external fragmentation, 4) compaction and 5) splitting.
  - ✓ 2. There are a lot of interesting questions in 문의 게시판 in the e-learning campus. Add your response or comment about an existing question that you are interested in (Explain your action as an answer of this quiz).
  - ✓ Due: until 6 PM Friday of this week (28th, May)



Appendix: 17.2 Low-level Mechanisms

- Tracking the size of allocated regions
  - ✓ free(): argument ➔ pointer only, not size
    - Need to track the size of a unit that is freed for coalescing
    - Most allocators utilizes a header block, usually just before the handed-out chunk of memory
      - Size, magic number for integrity checking, additional pointer to speed up deallocation, and other information

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

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

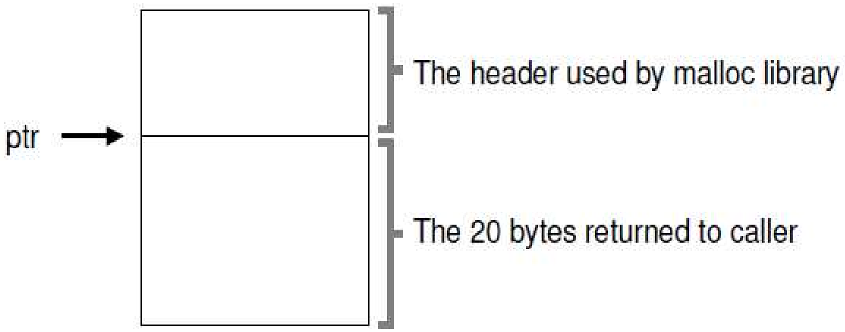


Figure 17.1: An Allocated Region Plus Header

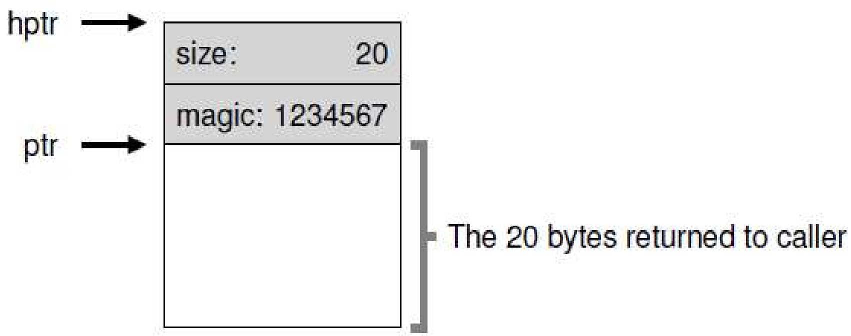


Figure 17.2: Specific Contents Of The Header

Appendix: 17.2 Low-level Mechanisms

- Embedding the free list into a heap
  - ✓ Figure 17.3: initial stage, build a free list inside the free space
    - Free space: 4KB (4096 byte), entry of the free list: 8 byte (size, next) ➔ size becomes 4088.
  - ✓ Figure 17.4: after “malloc(100)”
    - Header for the allocated space: 8 byte (size, magic #) ➔ 3980 (split occurs)
    - Head: pointer for the free list, ptr: pointer returned to malloc()
  - ✓ Figure 17.5: after three “malloc(100)”s ➔ 3764

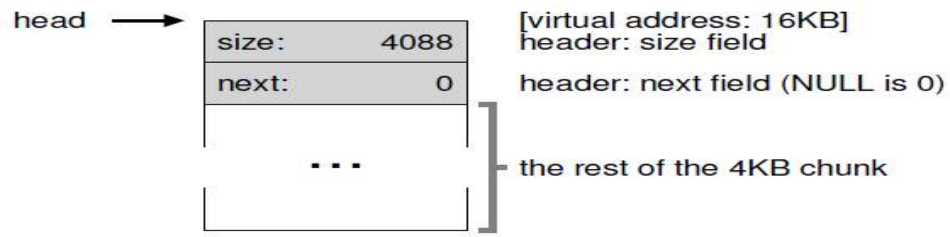


Figure 17.3: A Heap With One Free Chunk

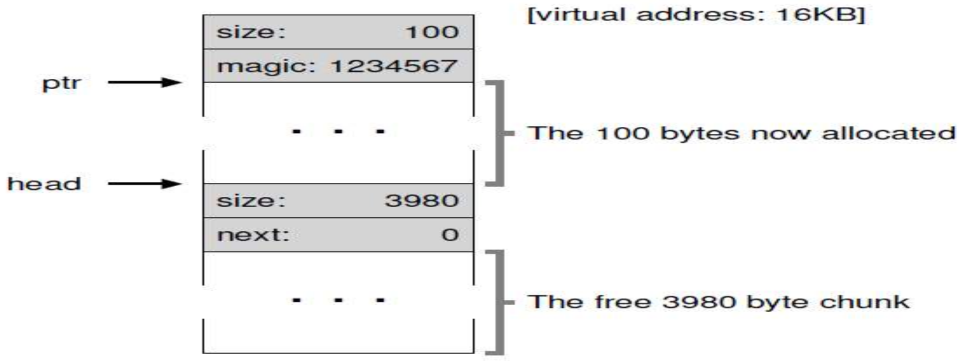


Figure 17.4: A Heap: After One Allocation

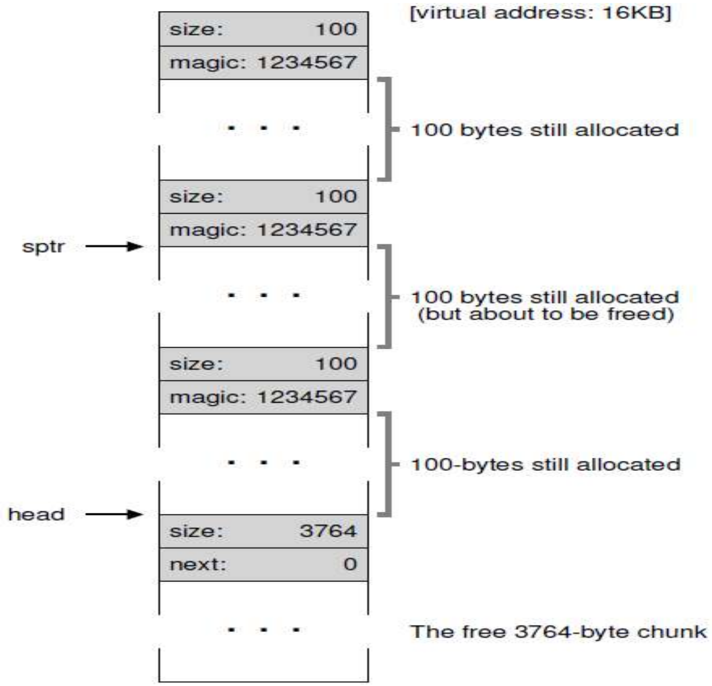


Figure 17.5: Free Space With Three Chunks Allocated

Appendix: 17.2 Low-level Mechanisms

- Embedding the free list into a heap
  - ✓ Figure 17.5: after three “malloc()”s, trigger one “free(sptr)” request
  - ✓ Figure 17.6: after “free(sptr)”
    - Two entries in the free list: head → (100, 16708) → (3764, 0 (NULL))
    - Virtual address 16708 = 16 x 1024 + 3 x 108
  - ✓ Figure 17.7: after three “free()”s
    - Compaction-less version (c.f. Compaction version: Figure 17.3)

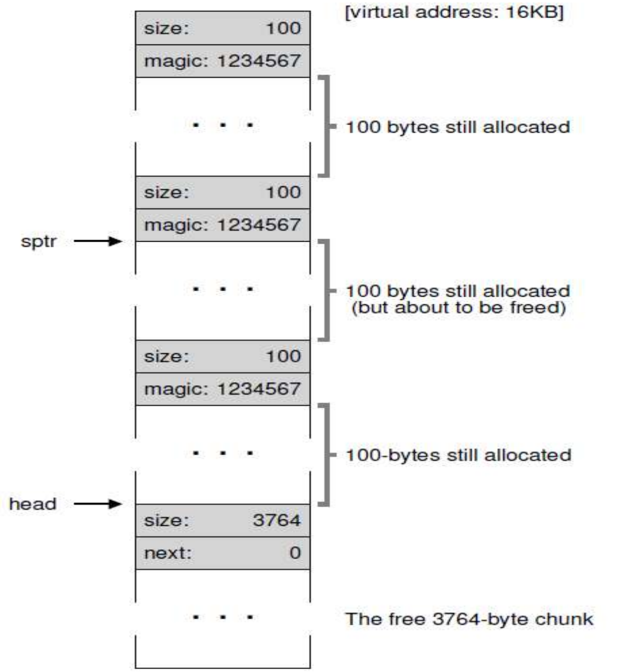


Figure 17.5: Free Space With Three Chunks Allocated

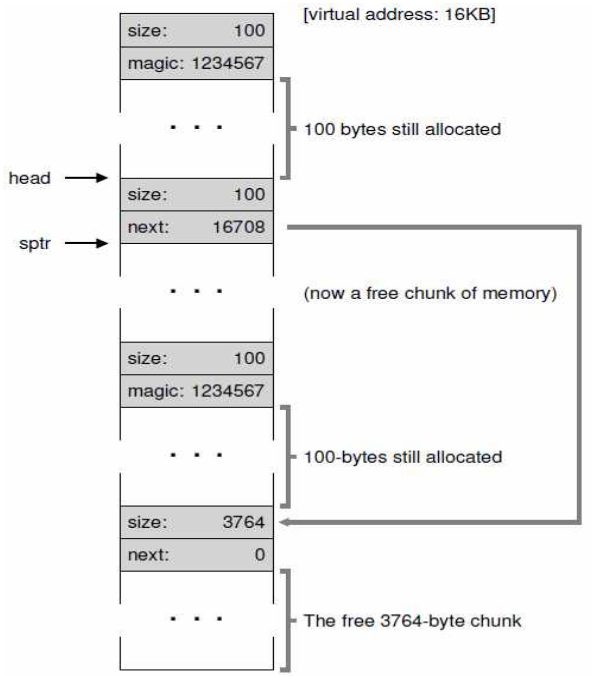


Figure 17.6: Free Space With Two Chunks Allocated

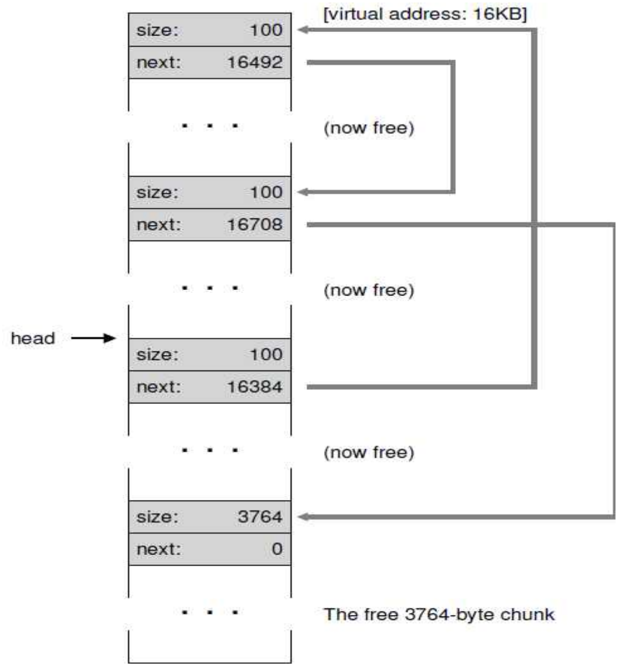


Figure 17.7: A Non-Coalesced Free List