Universidade Estadual de Campinas
Instituto de Computação

MC504 Sistemas Operacionais

T10 Memory Virtualization Segmentation

Referência principal

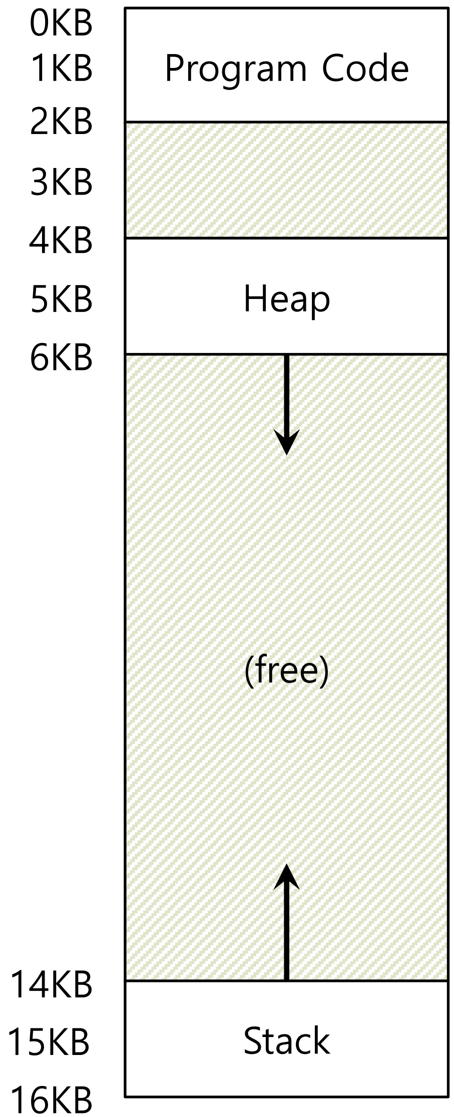
Ch.16 of Operating Systems: Three Easy Pieces by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 29 de agosto de 2018

Arthur João Catto, PhD

2º semestre de 2018

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

- Memory-management scheme that supports user view of memory.
- A program is a collection of segments.

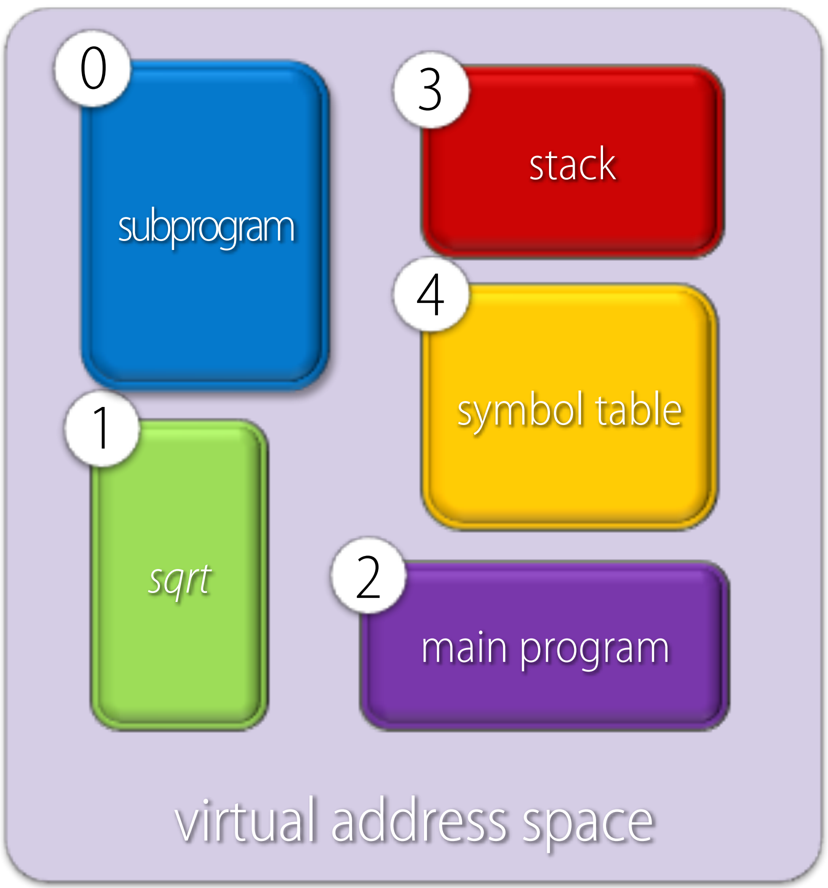
- A segment is a virtual unit such as:
  - main program
  - procedure
  - function
  - method
  - object
  - local variables
  - global variables
  - common block
  - stack
  - symbol table

Segmentation

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

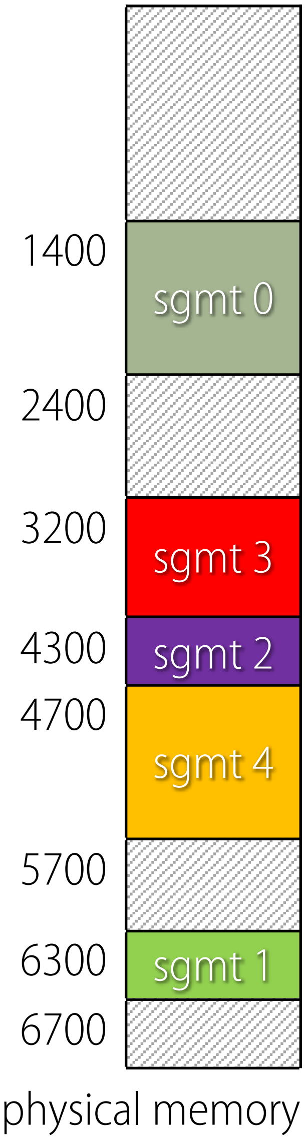
- Each segment can be placed in a different part of physical memory.
  - Base and bound are defined for each segment.

Logical View of Segmentation



	base	bound
0	1400	1000
1	6300	400
2	43000	400
3	3200	1100
4	4700	1000

segment table



Segmentation

- Segmentation provides a virtual view of main memory.
- The segments of a program may have different lengths.
- There is a maximum segment length.
- Since segments are not equal, segmentation resembles dynamic partitioning.

Segmentation

- Segment is a contiguous region of virtual memory
- Segment can be located anywhere in physical memory
  - Each segment has: start, length, access permission
- Processes can share segments
  - Same start, length, same/different access permissions

Segmentation Architecture

- Virtual address consists of a pair:

  $<$segment-number, offset$>$

- Each process has a segment table (in hardware)
  - Segment number → entry in table

- A segment table maps a two-dimensional virtual address onto a one-dimensional physical address.
  - Each entry in a segment table has at least two fields
    - Base: the segment's starting physical address in memory.
    - Bound: the length of the segment.

Segmentation Architecture

- Protection
  - With each entry in segment table associate
    - validation bit (if zero $\Rightarrow$ illegal segment)
    - privilege bits (read/write/execute)
  - Protection bits associated with segments
    - Code sharing occurs at segment level.
- Since segments vary in length, memory allocation is a dynamic storage allocation problem.

Segmentation Architecture

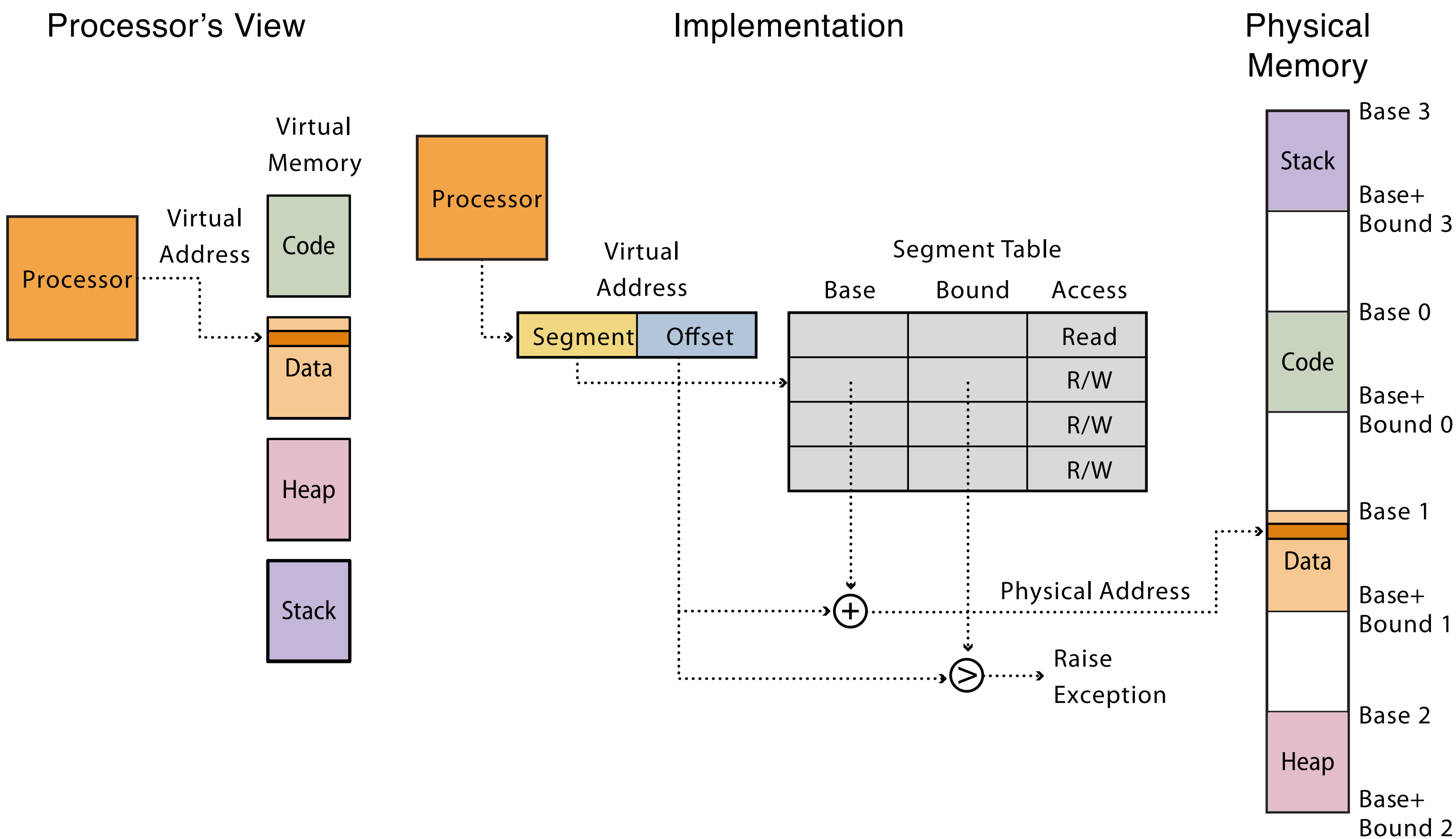
Two hardware registers are used to keep track of the segment table

- The Segment-Table Base Register (STBR) points to the segment table's location in memory.
- The Segment-Table Length Register (STLR) indicates the number of segments used by a program
  - Segment number s is legal if $s < STLR$.

Fine-Grained and Coarse-Grained Segmentation

- **Coarse-Grained** means segmentation in small numbers.
  - e.g., code, heap, stack.
- **Fine-Grained** segmentation allows a more flexible address space.
  - To support many segments, hardware support with a **segment table** is required.

Segmentation

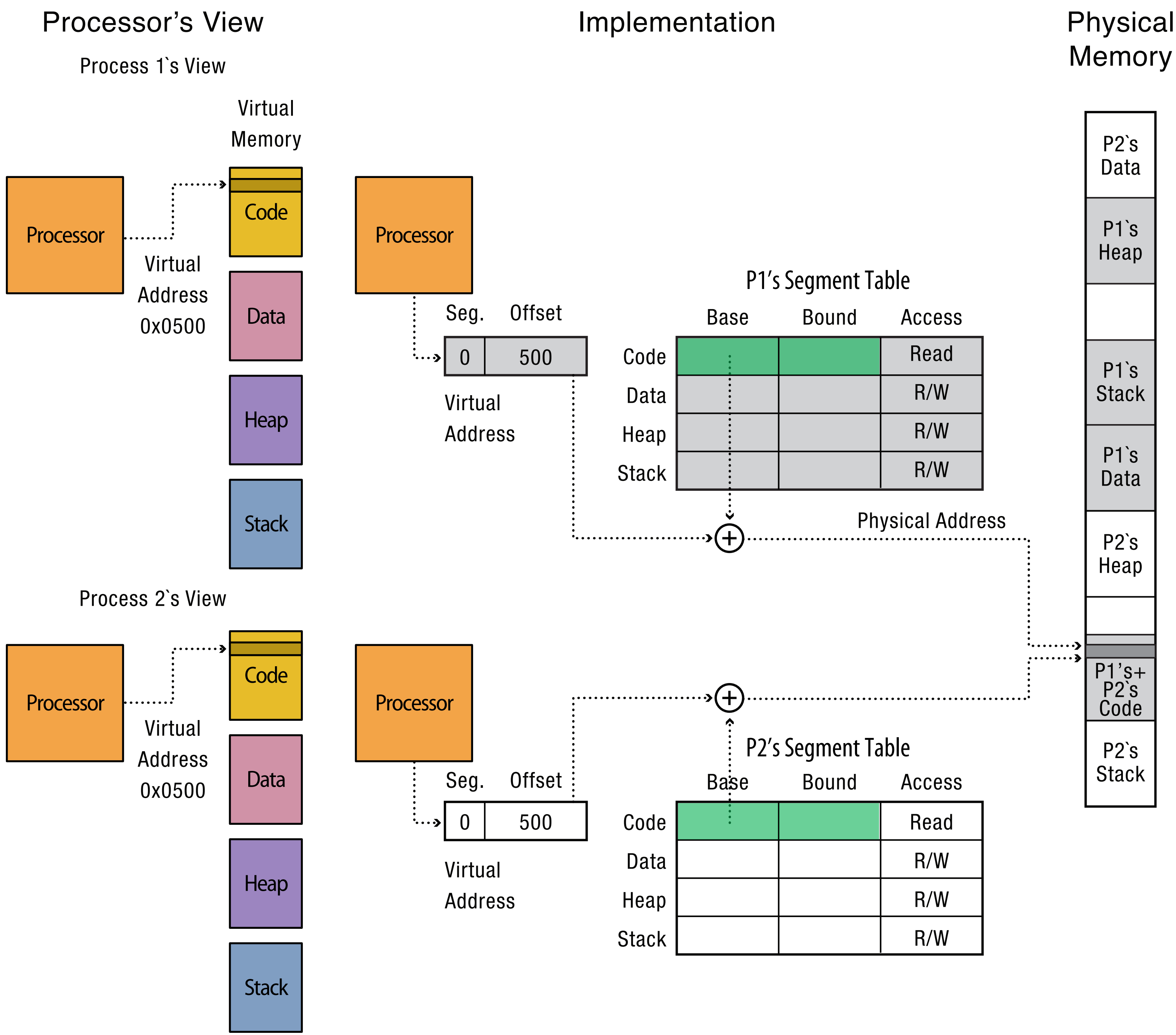


Segmentation

- Pros?
  - Can share code/data segments between processes
  - Can protect code segment from being overwritten
  - Can transparently grow stack/heap as needed
  - Can detect if need to copy-on-write or zero-on-reference
- Cons?
  - Complex memory management
    - Need to find chunk of a particular size
  - May need to rearrange memory from time to time to make room for new segment or growing segment
    - External fragmentation: wasted space between chunks

UNIX fork and Copy on Write

- UNIX fork
  - Makes a complete copy of a process
- Segments allow a more efficient implementation
  - Copy segment table into child
  - Mark parent and child segments as read-only
  - Start child process; return to parent
  - If child or parent writes to a segment (ex: stack, heap)
    - trap into kernel
    - make a copy of the segment and resume


Processor's View
Implementation
Physical Memory
Process 1`s View
Virtual Memory
Processor
Virtual Address 0x0500
Code
Data
Heap
Stack
Processor
Seg. Offset
0 500
Virtual Address
P1's Segment Table
Base Bound Access
Code Read
Data R/W
Heap R/W
Stack R/W
Physical Address
Process 2`s View
Processor
Virtual Address 0x0500
Code
Data
Heap
Stack
Processor
Seg. Offset
0 500
Virtual Address
P2's Segment Table
Base Bound Access
Code Read
Data R/W
Heap R/W
Stack R/W
P2`s Data
P1`s Heap
P1`s Stack
P1`s Data
P2`s Heap
P1's+ P2`s Code
P2`s Stack

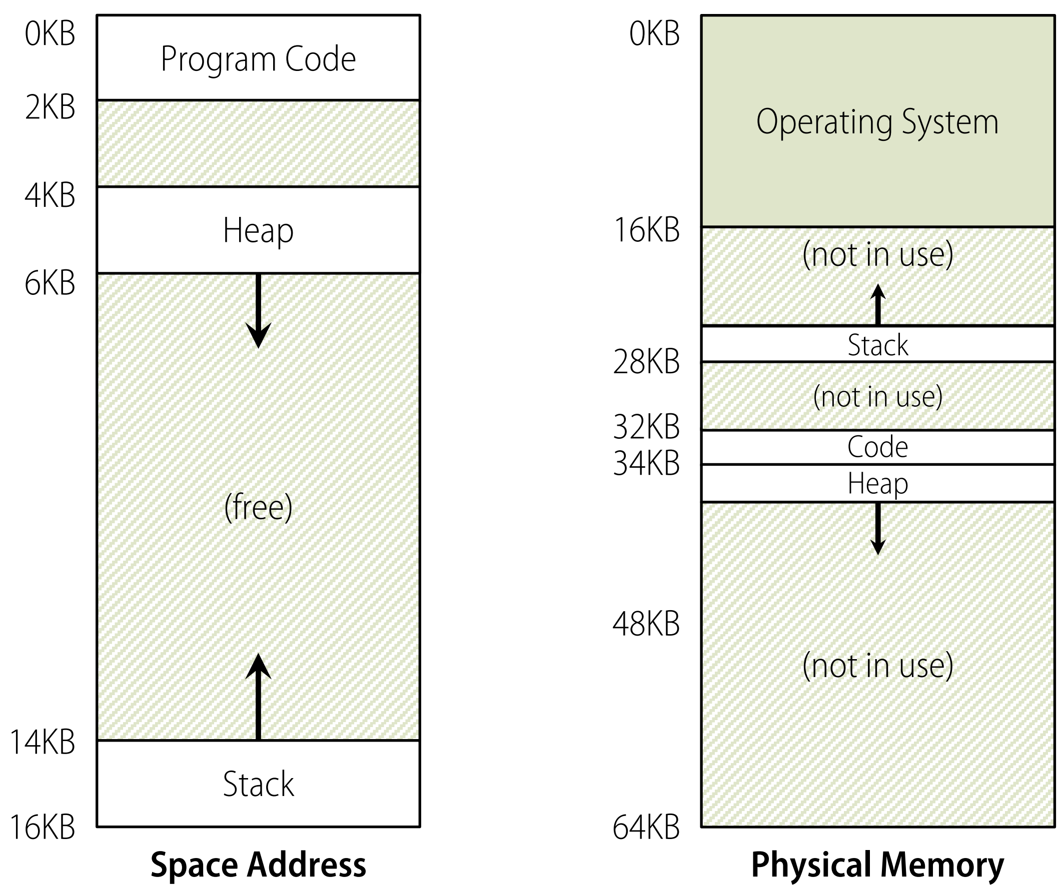
Zero-on-Reference

- How much physical memory is needed for the stack or heap?
  - Only what is currently in use
- When program uses memory beyond end of stack
  - Segmentation fault into OS kernel
  - Kernel allocates some memory
    - How much?
  - Clears the allocated memory
    - Avoid accidentally leaking information!
  - Modify segment table
  - Resume process

A point to ponder

With segmentation, what is saved/restored on a process context switch?

Placing Segments In Physical Memory



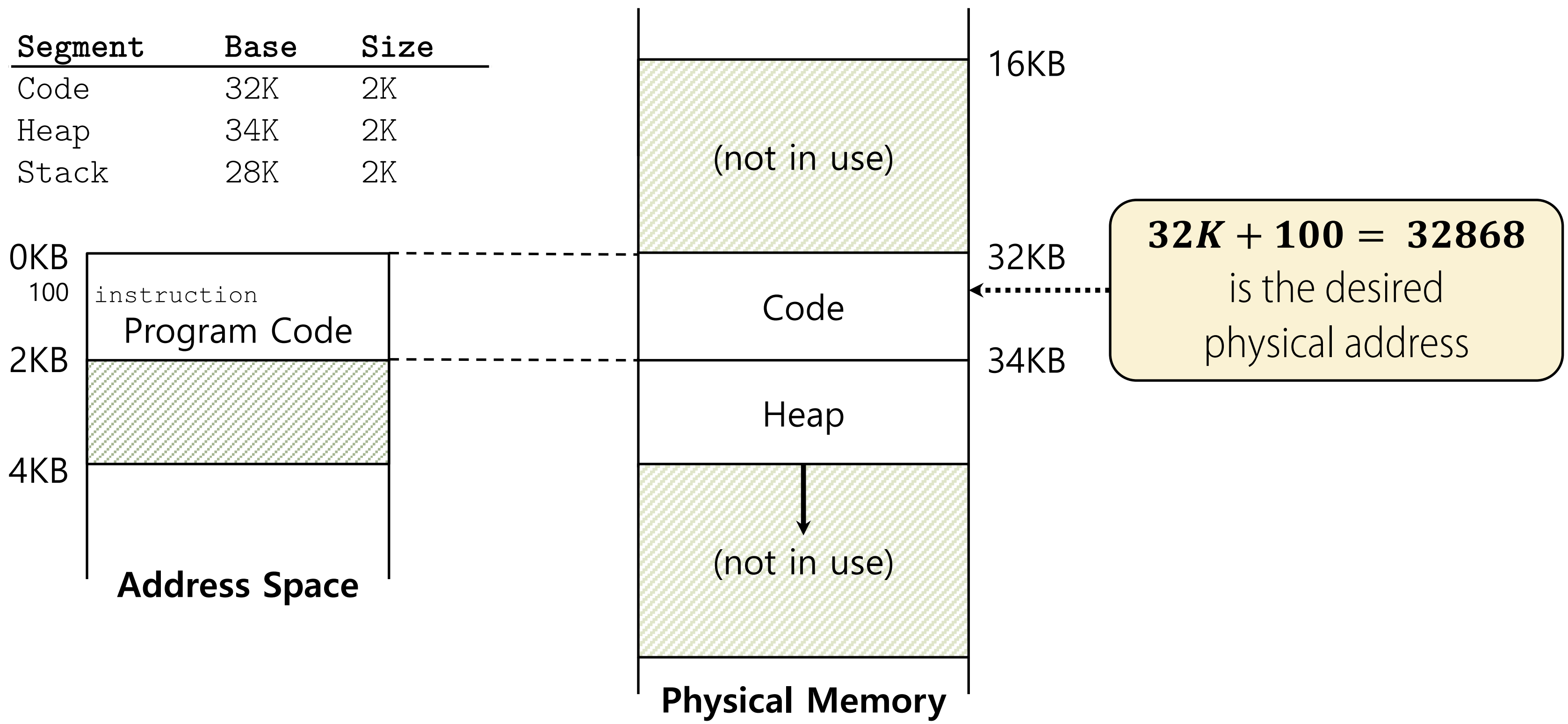
Segment	**Base**	**Size**
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Example

Address Translation on Segmentation

- The code segment starts at virtual address $\mathbf{0}$ in address space.
- The offset of virtual address $\mathbf{100}$ is $\mathbf{100}$.
- *physical address = base + offset*

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

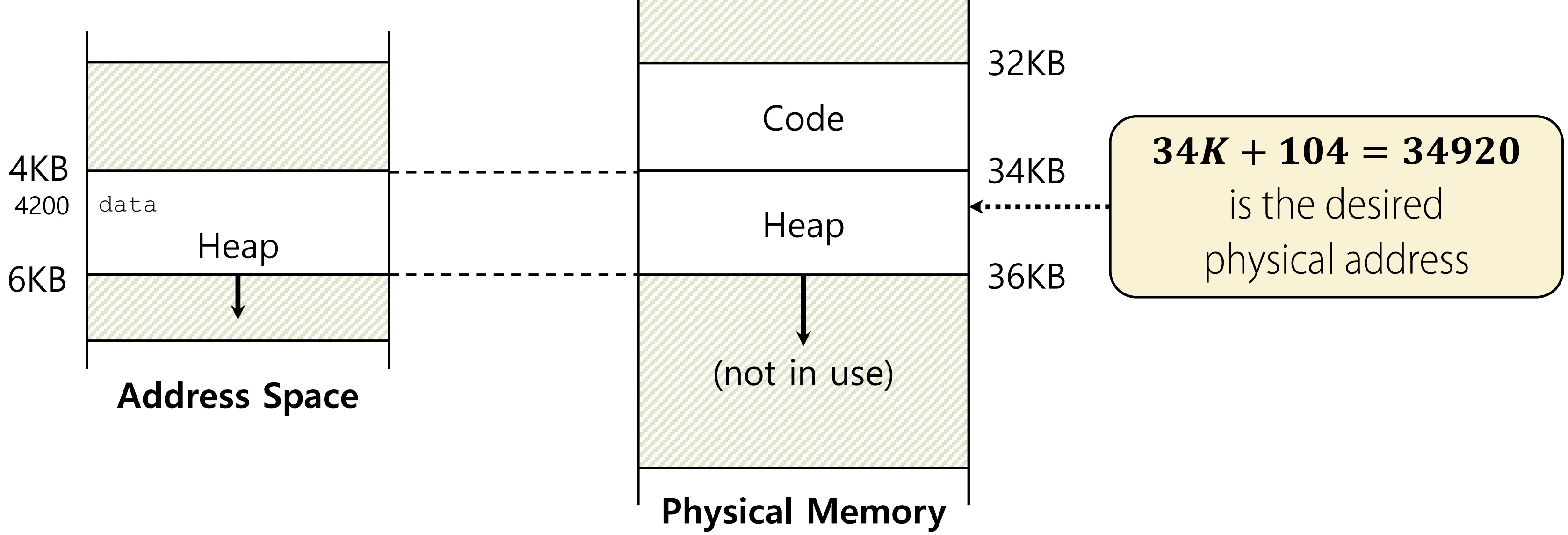


Example

Address Translation on Segmentation

- The heap segment starts at virtual address 4096 in address space.
- Thus, the offset of virtual address 4200 is $4200 - 4096 = 104$.
- *physical address = base + offset*.

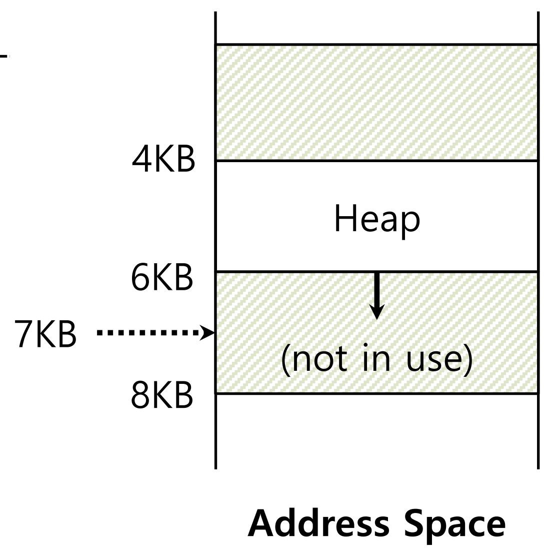
Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K



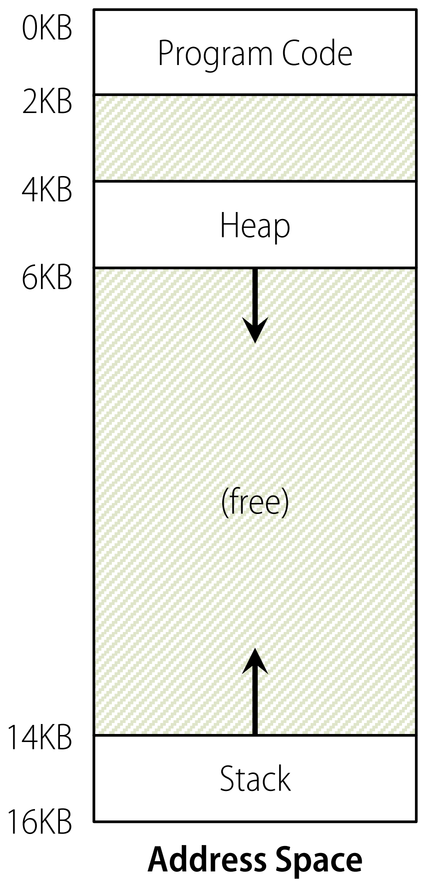
Segmentation Fault or Violation

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

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

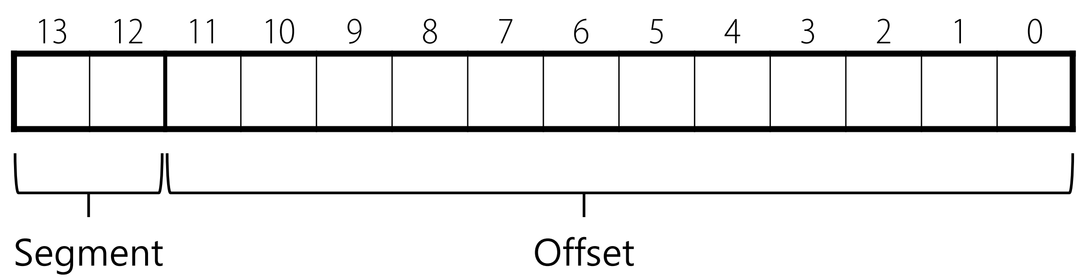


Which Segment Are We Referring To?



- The hardware uses segment registers during address translation. **How does it know the offset into a segment, and to which segment an address refers?**

- One possible approach is to chop up the address space into segments based on the **top few bits** of the virtual address.
  - For example, if the OS requires three segments, the top two bits of the virtual address could be used to represent the segment number.
  - Assuming an address space of 16KB, the remaining 12 bits will be used to represent the offset into the segment, which leaves us with a maximum segment size of 4KB.



Example

Referring to a Segment

- Let us assume that we want to know what segment and offset a virtual address 4200 refers to.
  - Since $4200_{10} = 01000001101000_2$ that address would be read as

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	0	1	0	0	0

Segment = 1 Offset = 64 + 32 + 8 = 104

  - Assuming that our segments are numbered as shown on the table, virtual address 4200_{10} would refer to an offset of 104_{10} into the **heap** (segment #1).

Segment	**bits**
Code	00
Heap	01
Stack	10
(unused)	11

Referring to a Segment

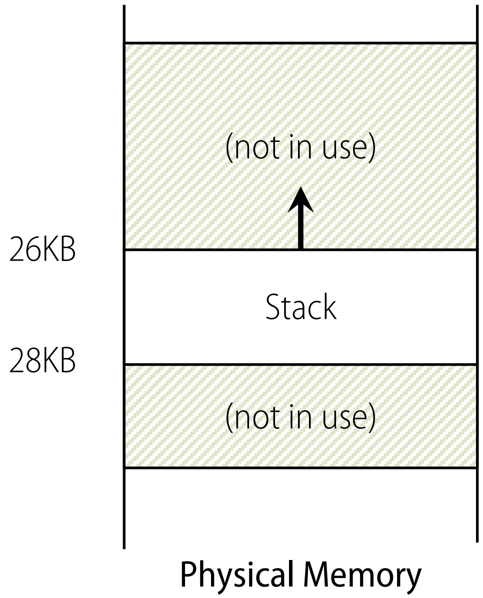
- Assuming that **base** and **bounds** were arrays indexed by segment number, to translate a virtual address into a physical one, the hardware would be doing something like…

```
#define SEG_MASK 0x3000   // (11000000000000)
#define SEG_SHIFT 12
#define OFFSET_MASK 0xFFF // (00111111111111)

// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT;
// get offset
Offset = VirtualAddress & OFFSET_MASK;
if (Offset >= Bounds[Segment])
    RaiseException(PROTECTION_FAULT);
else {
    PhysicalAddress = Base[Segment] + Offset;
    Register = AccessMemory(PhysicalAddress);
}
```

Referring to the Stack Segment

- Remember that **the stack grows backwards**.
- **Extra hardware support** is needed.
  - The hardware must check which way a segment grows.
    - Let us use a bit for that: 1 = positive direction, 0 = negative direction



Segment Registers with Support for Negative-Growth

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

Example

Referring to an Address within the Stack

- Let us assume that we want to know what segment and offset a virtual address $15K$ refers to.
  - Since $15K = 11110000000000_2$ that address would be read as

13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0

Segment = 1

Offset = 64 + 32 + 8 = 104

  - Assuming that our segments are numbered as shown on the table, virtual address 4200_{10} would refer to an offset of 104_{10} into the **heap** (segment #1).

Segment	bits
Code	00
Heap	01
Stack	10
(unused)	11

Example

Referring to an Address within the Stack

- Let us assume that we want to know what segment and offset a virtual address $15K$ refers to.
  - Since $15K = 11110000000000_2$ that address would be read as

13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0

Segment = 3 Offset = 3K = 2048 + 1024 = 3072

  - Since the stack segment grows backwards the $3K$ offset must be subtracted from the maximum segment size ($4K$) to obtain the correct negative offset.

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

  - $3K - 4K = -1K$ which will now be added to the stack base address ($28K$) to generate the correct physical address ($27K$).
  - The offset is also checked against bounds by comparing its absolute value with the segment's size.

Sharing

- Segments can also be **shared among address spaces**.
  - **Code sharing**, for example, is common and still used in systems today.
- Segment sharing requires extra hardware support in the form of **protection bits.**
  - A few more bits are added to the segment registers or to the sgment table to indicate permission to read, write and/or execute.

Segment Registers with Support for Negative-Growth and Sharing

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

Fragmentation

- **External Fragmentation**
  - Holes of free space in physical memory that make it difficult to allocate new segments.
    - There are 24KB free, but not in one contiguous segment.
    - The OS cannot satisfy a 20KB request.
- **Compaction**
  - Rearranging the existing segments in physical memory, but this is expensive.
    - Stop running processes.
    - Copy data to somewhere.
    - Change segment base register value.

Memory Compaction