Chapter 16 Segmentation

Base and bounds is not as flexible as we would like it to be:

- Requires big chunk of “free” space in the middle
- Taking up physical memory
- Doesn’t look so bad with our small address spaces
  - Imagine a 32-bit address space (4 GB in size)
  - Typical program only use a few megabytes of memory

16.1 Segmentation: Generalized Base & Bounds

Idea: use multiple base/bounds pairs

- One base/bounds-pair for each **logical segment** of the address space

Def. Segment: *Contiguous position of address space of a particular length*.

Address space have three logically different segments

- Code
- Stack
- Heap

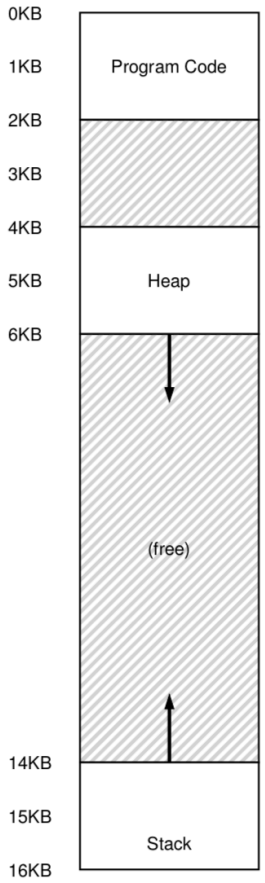


Figure 16.1: **An Address Space (Again)**

Example Fig 16.1: Each segment can be placed independently in physical memory.

Example Fig 16.2: 64 KB physical memory

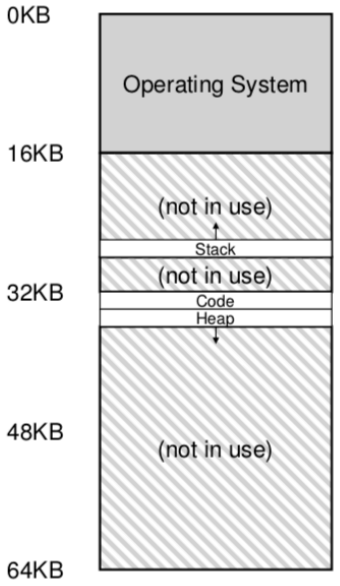
Large amounts of unused address space can be accommodated.

HW: MMU support for segmentation

- set of three base and bounds register pairs

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

Figure 16.3: **Segment Register Values**



16.2: **Placing Segments In Physical Memory**

Example: Translation (refer to Fig 16.1)

- Instruction fetch from virtual address 100 (code segment)
  - Translate: 100+32 KB (32 KB = 32 *1024) = 32868
- Heap: consider virtual address 4200 (heap segment)
  - Translate: 4200 - 4096 = 104 + 34 KB = 34920

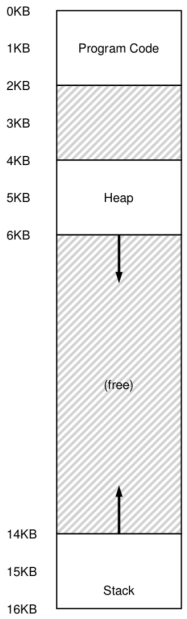


Figure 16.1: **An Address Space (Again)**

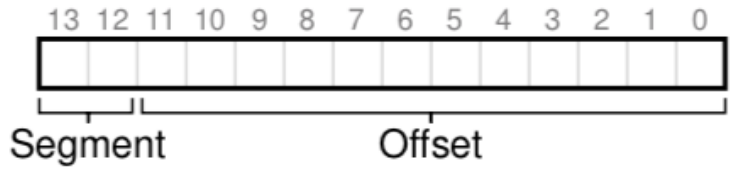
16.2 Which Segment are we Referring to?

Q: How does the HW know which segment the address is referring to?

Explicit Approach:

- two top bits of the virtual address space — to indicate segment type

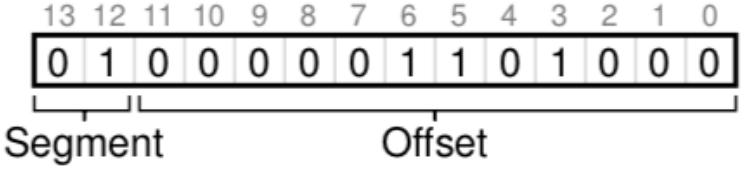
“00”	Code
“01”	Heap
“10”	Stack
“11”	Unused



From the Example above: Heap segment with virtual address 4200:

Segment bits: 01

Offset bits: 0b0000 0110 1000 = 0x068 = 104 decimal



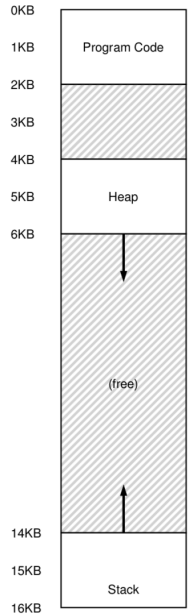


Figure 16.1: **An Address Space (Again)**

Physical address = offset + base register

SEG_MASK = 0x3000 (bits 12 and 13: 11)
Example:

```
        01 0000 0110 1000
AND     11 0000 0000 0000
        01 0000 0000 0000
SHIFT   00 0000 0000 00 01
```

SEG_SHIFT = 12
In order to move the segment bits to the lowest two bits.

```
        01 0000 0110 1000
AND     00 1111 1111 1111
        00 0000 0110 1000
```

OFFSET_MASK = 0x0FFF (12 bits for the offset)

Base and Bounds variables are arrays with one entry per segment.

```
1  // get top 2 bits of 14-bit VA
2  Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3  // now get offset
4  Offset  = VirtualAddress & OFFSET_MASK
5  if (Offset >= Bounds[Segment])
6      RaiseException(PROTECTION_FAULT)
7  else
8      PhysAddr = Base[Segment] + Offset
9      Register = AccessMemory(PhysAddr)
```

Pseudo code on the right can be used to compute the

Physical address from virtual address…

Implicit approach

- The hardware determine the segment based on how the virtual address was formed…
  - Generated from PC because of an instruction fetch — address is code segment
  - Generated from SP because the CPU is using it — address is stack segment
  - Any other address — heap segment

Another digression into binary and (simple) boolean algebra!

AND TABLE:
a b = c
0 0 = 0
0 1 = 0
1 0 = 0
1 1 = 1

Example:
x = 0001
y = 1101
z = 0001

Bitwise AND: x & y = 0001

Logical AND: len(array) > 3 && timeSliceExpired = true/false

OR TABLE:
a b = c
0 0 = 0
0 1 = 1
1 0 = 1
1 1 = 1

Example:
x = 0001
y = 1101
z = 1101

Bitwise OR: a | b = 1101

Logical OR: len(array) > 3 || timeSliceExpired = true/false

XOR TABLE:

a b = c
0 0 = 0
0 1 = 1
1 0 = 1
1 1 = 0

16.3 What About the Stack?

Since the stack grows backwards — need extra bit in hardware to indicate the growing direction of the segment

Hardware can now translate such virtual address differently…

Example. Access virtual address 15KB (stack Fig 16.1).

15KB = 11 1100 0000 0000 (hex: 0x3C00)

Two top bits: 11 - stack segment

OFFSET_MASK: 0x0FFF

Offset: 0x0C00 = 3KB = 3072

To find the correct negative offset (into the stack segment)

- Subtract max segment size (4 KB in this example) from 3 KB
- Negative offset: 3KB - 4KB = -1KB
- Simply add negative offset to the base: 28KB + (-1KB) = 27 KB (physical address)
- (Maps to physical address 27KB in Fig 16.2)

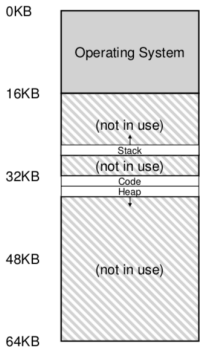


Figure 16.2: **Placing Segments In Physical Memory**

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

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

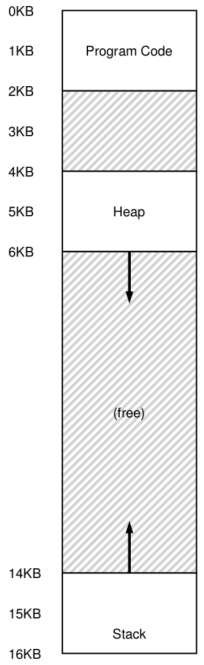


Figure 16.1: **An Address Space (Again)**

16.4 Support for Sharing

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

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

Idea: save memory by sharing certain memory segments between address spaces (=process's)

HW: add protection bits

- A few extra bits per segment
- Read, Write, and Execute

Setting code segment to read-only:

- Multiple processes can share code
- Without harming isolation
  - Each process still thinks that it is accessing its own private memory
  - While OS is secretly sharing their memory!

HW: must check if a particular access is permissible

Example: if user process tries to

- Write to a read-only segment, or
- Execute from a non-executable segment,
- HW should raise an exception

16.6 OS Support

Motivation and Context

- Unused space between stack and heap need not be allocated in physical memory.
- Allowing for more address spaces (processes) in physical memory.

Tasks for the OS with segmentation:

- Q: What should the OS do on a context switch?
  - Segmentation registers must be saved and restored.
- Free space management
  - Find space for new processes
  - A process's address space has segments of different sizes.

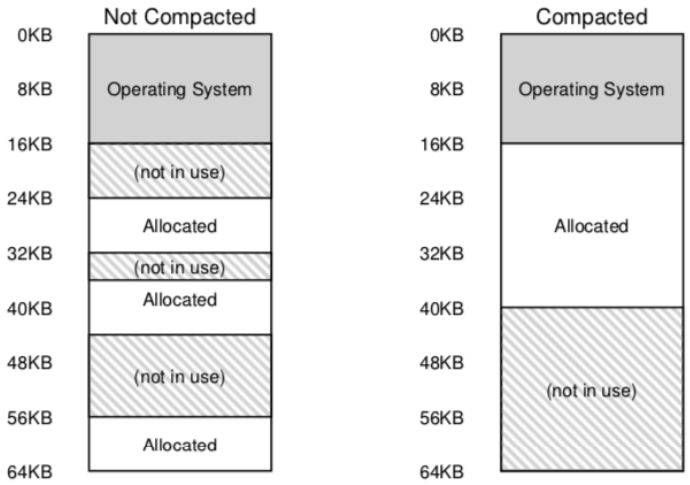


Figure 16.6: **Non-compacted and Compacted Memory**

General problem with free space management

- Physical memory becomes full of little holes of free space
- Makes it difficult to
  - Allocate new segments or
  - Grow existing ones
- This is called external fragmentation (Fig. 16.6)

Example Fig. 16.6: A process wish to allocate 20KB segment.

There is 24KB free, but not in one contiguous segment (rather in three non-contiguous chunks).
OS cannot satisfy the 20KB request.

One solution:

- Stop processes one-by-one, move segments, update segment registers to point to new location
- Expensive: memory-intensive and use of processor time

Simpler/Better approach:

- Use a free-list management algorithm
  - Keep large **extents** of memory available for allocation
- Ex. Best-fit algorithm — keeps list of free spaces — returns the one closest in size that satisfies the desired allocation for a memory allocation request.

16.7 Summary

Segmentation

- Support sparse address spaces
- Avoid wasting memory between logical segments of an address space
- Fast translation with low overhead
- Code sharing

Problems with segmentation (variable-sized segments)

- Free memory gets chopped into odd-sized pieces (external fragmentation)
- Memory allocation difficult (many smart algorithms exists)
  - But fundamentally hard to avoid external fragmentation
- Not flexible enough, to support fully generalized sparse address space
  - e.g., if heap is sparsely-used — must keep entire heap in memory …