12. Segmentation

there is a big chunk of "free" space right in the middle,
between stack & heap

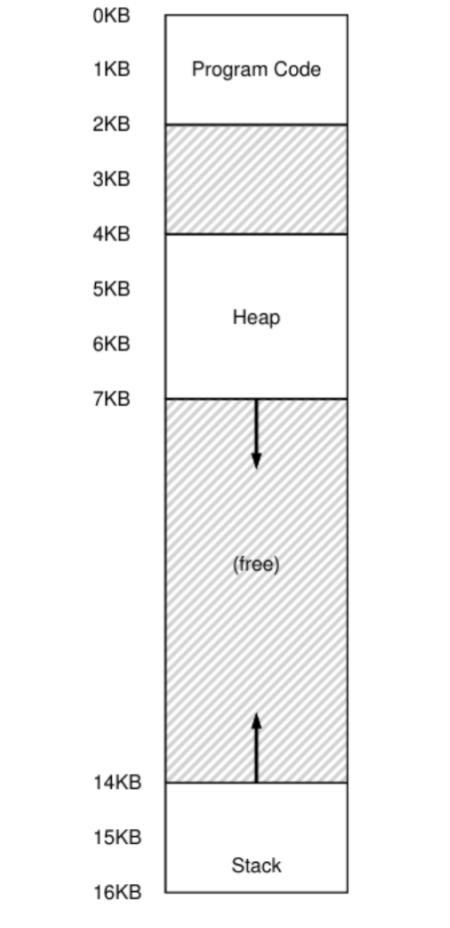


Figure 16.1: **An Address Space (Again)**

although the space between the stack & heap : is not being used by the process, it is still taking up physical memory

=> simple approach of using a base & bounds register pair to virtualize memory : wasteful

- also hard to run a program when the entire address space doesn't fit into memory.

Segmentation

instead of having just one base & bounds pair in MMU,
have a base & bounds pair per logical segment of the address space

- segment : a contiguous portion of the adress space of a particular length.

In picture, we have 3 logically-different segments : code, stack, and heap.

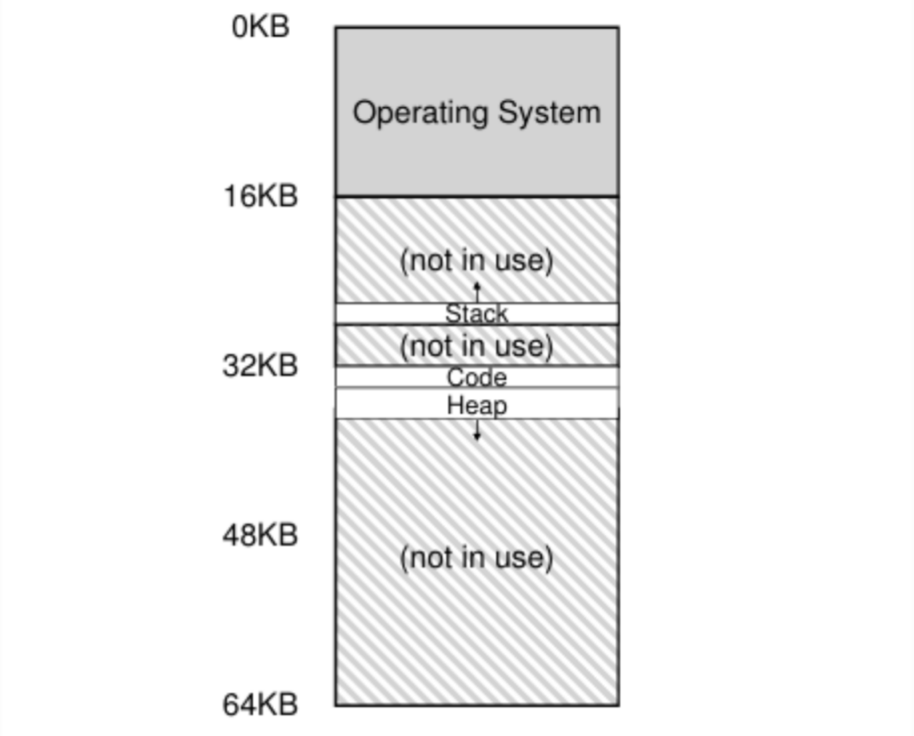


Figure 16.2: **Placing Segments In Physical Memory**

only used memory is allocated space in physical memory

Hardware structure in MMU required to support segmentation

: a set of 3 base & bounds register pairs

Segment	Base	Size
Code	32K	2K
Heap	34K	3K
Stack	28K	2K

Figure 16.3: **Segment Register Values**

size == bound

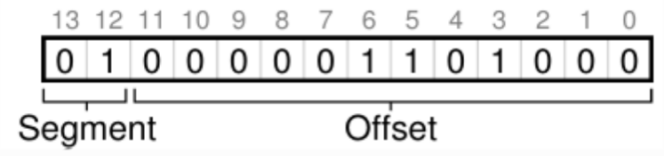
segmentation fault : from a memory access on a segmented machine to an illegal address

Which Segment are we referring to?

1. explicit approach

: chop up the address space into segments based on the top few bits of the virtual address

in our example, we have 3 segments => need 2 bits (00, 01, 10)




```
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)
```

SEG_MASK : 11 0000 0000 0000
OFFSET_MASK : 00 1111 1111 1111

USING the top **so many bits** to select a segment **limits** use of the virtual address space : segment + Offset = virtual address space
if segment up => offset (range that we can express) down

2. implicit

: hardware determines the segment by noticing how the address was formed.

- the address was generated from the program counter (= it was an instruction fetch) => address is within the code segment
- the address is based off of the stack or base pointer => stack segment
- any other address => heap

Stack grows backward : translation must proceed differently

the hardware also needs to know which way the segment grows

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**

How to translate

base + (Virtual address - segment index part - maximum segment size)

ex.

base + 11 1100 0000 0000 (15KB) - 11 0000 0000 0000 (12KB) - maximum segment size

= base + 1100 0000 0000 (3KB) - 4KB (size of 2^12) = base -1KB

ensure the absolute value of the negative offset is less than or equal to the segment's current size(in this case, 2KB > 1)

in stack : the first valid byte is actually baseKB-1

Support for Sharing

to save memory, sometimes it is useful to share certain memory segments between address spaces.

Ex. Code sharing is common & still in use in systems

protection bits
indicating whether or not a program can read or write a segment, or execute code that lies within the segment.

By setting a code segment to read-only, the same code can be shared across multiple processes, without worry of harming isolation.

while each process still thinks that it is accessing its own private memory, the OS is secretly sharing memory which cannot be modified by the process, and thus the illusion is preserved.
with protection bits, the hardware algorithm
: in addition to checking whether a virtual address is within bounds, the hardware also has to check **whether a particular access is permissible**.

Coarse-grained Segmentation
: chops up the address space into relatively large, coarse chunks.

fine-grained segmentation
: consist of a large number of smaller segments

Supporting many segments requires even further hardware support, with a segment table of some kind stored in memory.

Segment table usually support the creation of a very large number of segments, and thus enable a system to use segments in more flexible ways than we have thus far discussed.

by having fine-grained segments, the OS could better learn about which segments are in use and which are not and thus utilize main memory more effectively.

OS SUPPORT

What should the OS do on a context switch?

the segment registers must be saved & restored.
Clearly, each process has its own virtual address space, and the OS must make sure to set up these registers correctly before letting the process run again.

OS interaction when segments grow (or shrink)

1. program may call malloc() / to allocate an object
- O -> existing heap will be able eto service the request
- X -> heap segment it self may need to grow
    => the memory allocation library will perform a system call to grow the heap (ex. sbrk())
    * OS could reject the request, if no more physical memory is available, or if it decides that the calling process already has too much

Managing Free space in physical memory

When a new address space is created, the OS has to be able to find space in physical memory for its segments.

Now, we have a number of segments per process, and each segment might be a different size.

physical memory quickly becomes full of little holes of free space, making it difficult to allocate new segments, or to grow existing ones.
= external fragmentation.

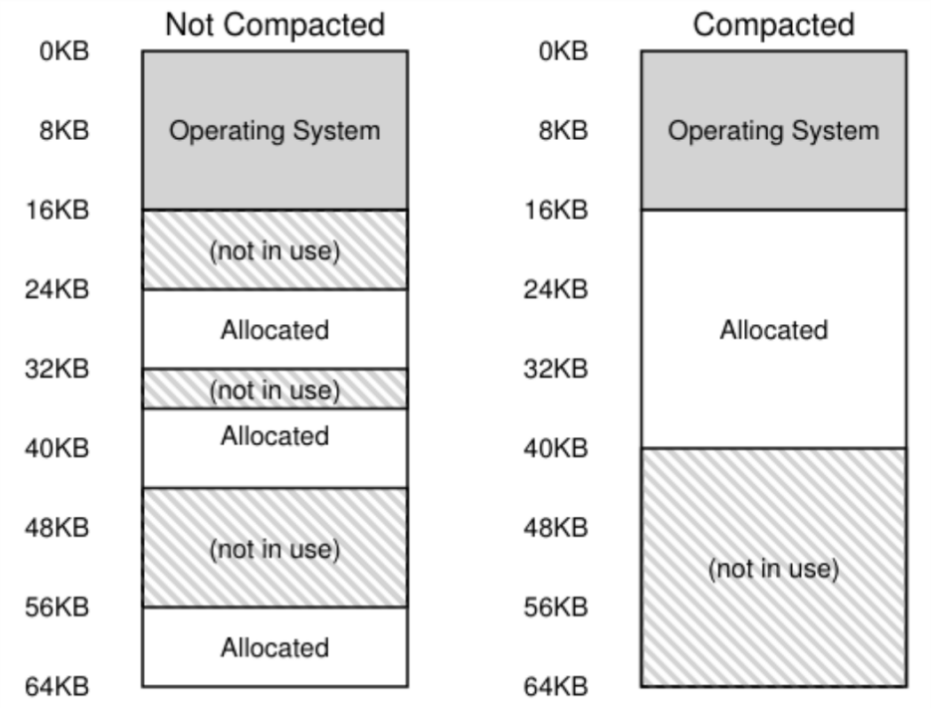


Figure 16.6: **Non-compacted and Compacted Memory**

: **Solution 1. COMPACT physical memory by rearranging the existing segments.**

Ex.
1. the OS could stop whichever processes are running
2. copy their data to one contiguous region of memory
3. change their segment register values to point to the new physical locations
=> thus have a large free extent of memory.

Drawbacks of Compaction

- **expensive**, as copying segments is memory-intensive and generally uses a fair amount of processor time
- makes **requests to grow existing segments hard to serve**, and may thus cause further rearragnement to accomodate such requests.

: Solution 2. a free-list management algorithm that tries to keep large extents of memory available for allocation

- best-fit / worst-fit / first-fit / buddy algorithm

no matter how smart the algorithm, external fragmentation will still exist.
a good algorithm attempts to minimize it.

Summary

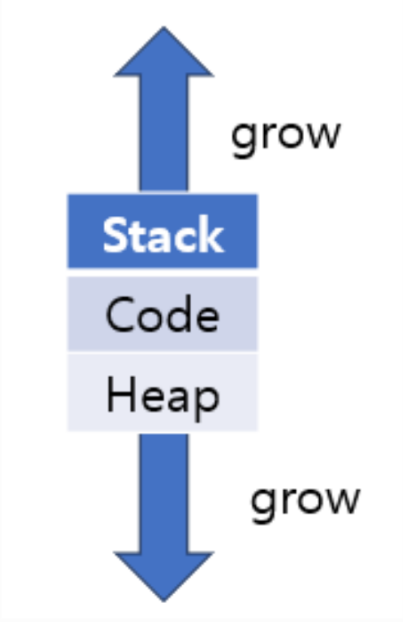
Segmentation is fast, as doing the arithmetic segmentation requires is easy & well suited to hardware.
the overheads of translation are minimal.
A fringe benefit areises too : code sharing.

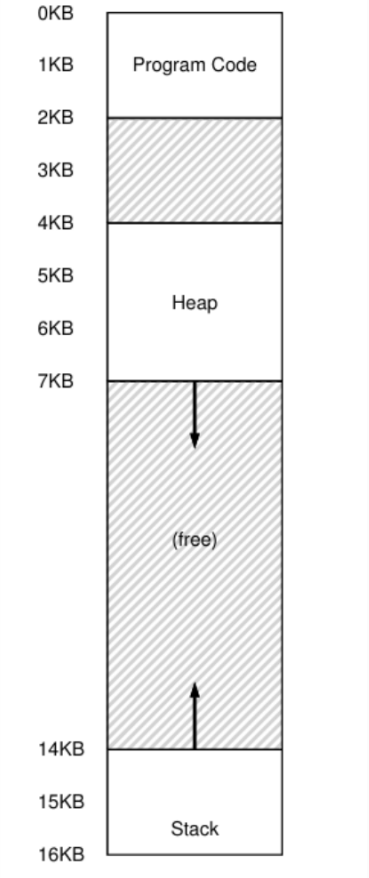
allocating variable-sized segments in memory leads to external fragmentation.

segmetation still isn't flexible.
: sparsely-used heap all in one logical segment => the entire heap must still reside in memory in order to be accessed.
If our model doesn't match how the underlying segmentation has been designed to support it, segmentation doesn't work very well.

How about this one?

- can express more address because it use less segment bits
- at lease less internal fragmentation compared to original all seperated segment





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