Base and Bounds (contd)

Involvement of OS

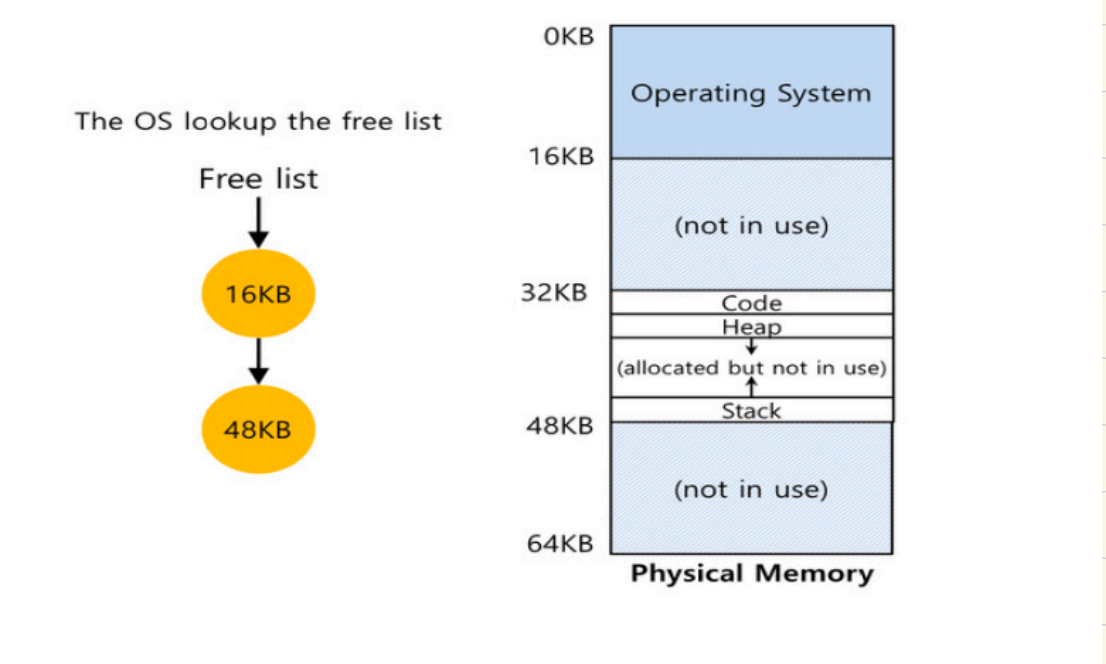
The OS must take action to implement base and bound approach

Three critical junctures → * when a process is created
* when a process is terminated
* when context switch occurs

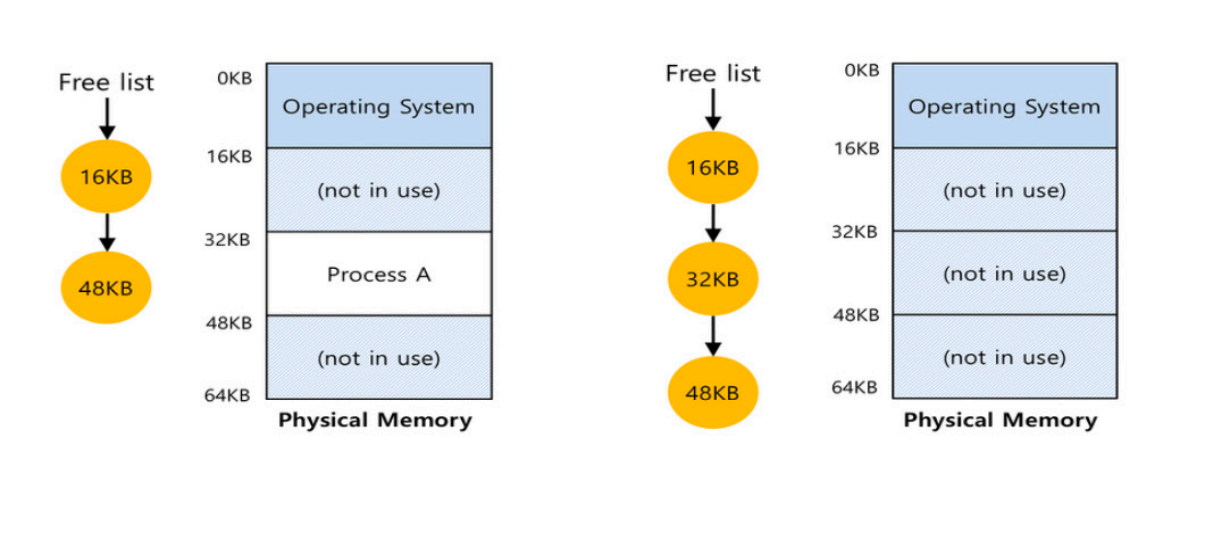
When a process is created →
OS must be able to allocate memory to process,

Keep track which parts of free memory are not in use.

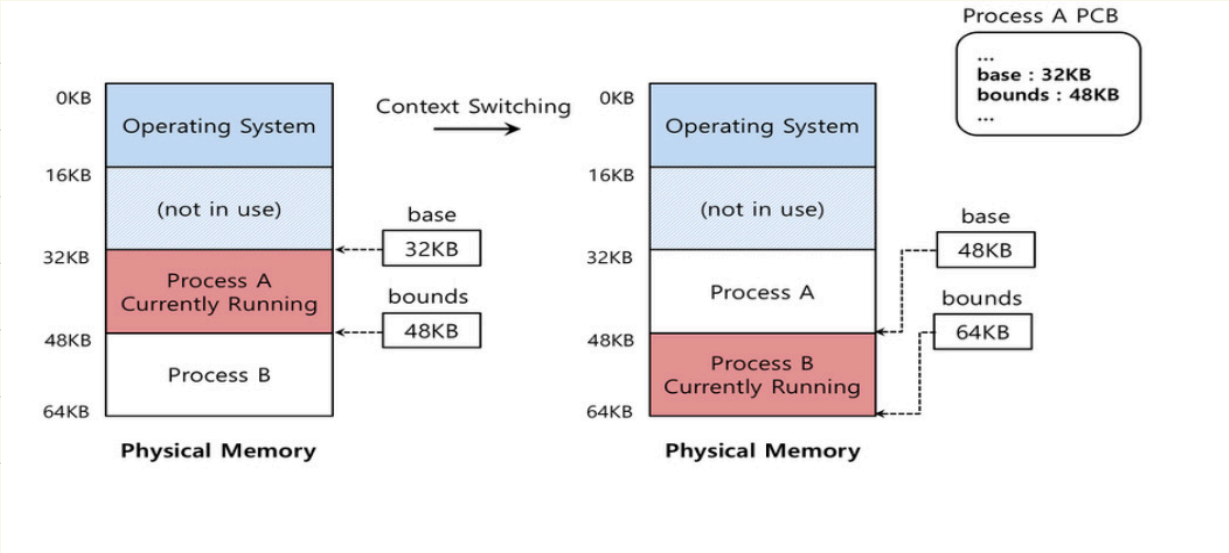
A free list is a list of the ranges of the physical memory which are not currently in use.



when a process is terminated
→ the OS must put the memory back on the free list.



when context switch occurs → the OS must save and restore the base and bounds pair in the process control block (PCB)



Apart from above, OS must provide exception handlers.

* if a process tries to access memory outside its bounds.
  the CPU will raise an exception
* the OS must be prepared to take action when such exception arises.

Hardware/OS interaction in timeline

OS @ run (kernel mode)	Hardware	Program (user mode)
To start process A:		
allocate entry in process table		
alloc memory for process		
set base/bound 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	
	perform fetch	
		Execute instruction
	if explicit load/store:	
	ensure address is legal	
	translate virtual address	
	perform load/store	
		(A runs...)
	Timer interrupt	
	move to **kernel mode**	
	jump to handler	
Handle timer		
decide: stop A, run B		
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 kill process B		
deallocate B's memory		
free B's entry		
in process table		

Pros of Base and Bound approach

* simple
* fast (2 registers, adder, comparator)

Cons of Base and Bound approach

* can't keep programs from accidentally overwriting its own code

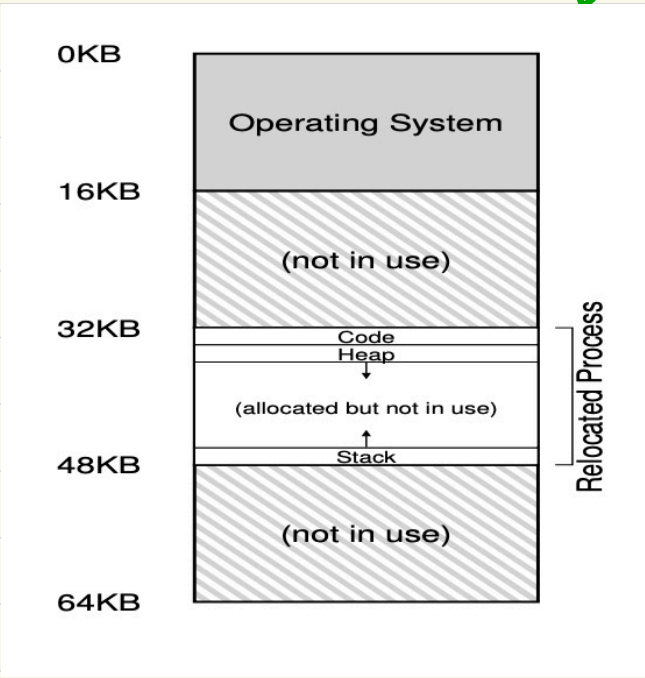
(because, here CPU only ensures you stay within your own allocated region)

* can't share code/data with other processes.

(because, the mapping is contiguous and exclusive)

* can't grow stack/heap as needed.
* Internal fragmentation

if the process's stack and heap are not too big, all the space b/w the two is simply wasted.



Question: How do we support a large address space with (potentially) a lot of free space b/w the stack and the heap?

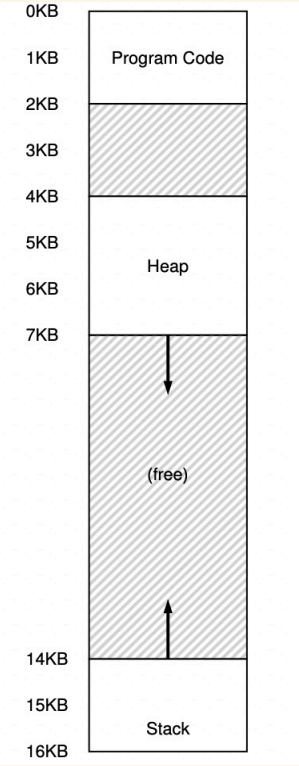
Segmentation

Segment is just a contiguous portion of the address space of a particular length.

- logically different segments: code, heap, stack

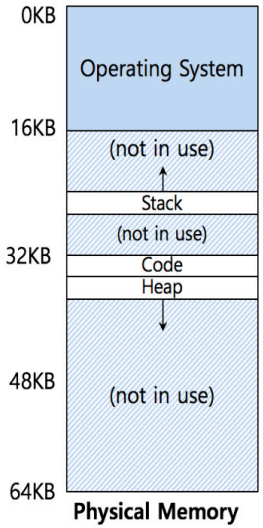
each segment can be placed in different part of physical memory.

How? Instead of one base and bounds pair, maintain a pair for each logical segments, code, heap and stack.



Virtual address space

would like to relocate here in physical memory



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

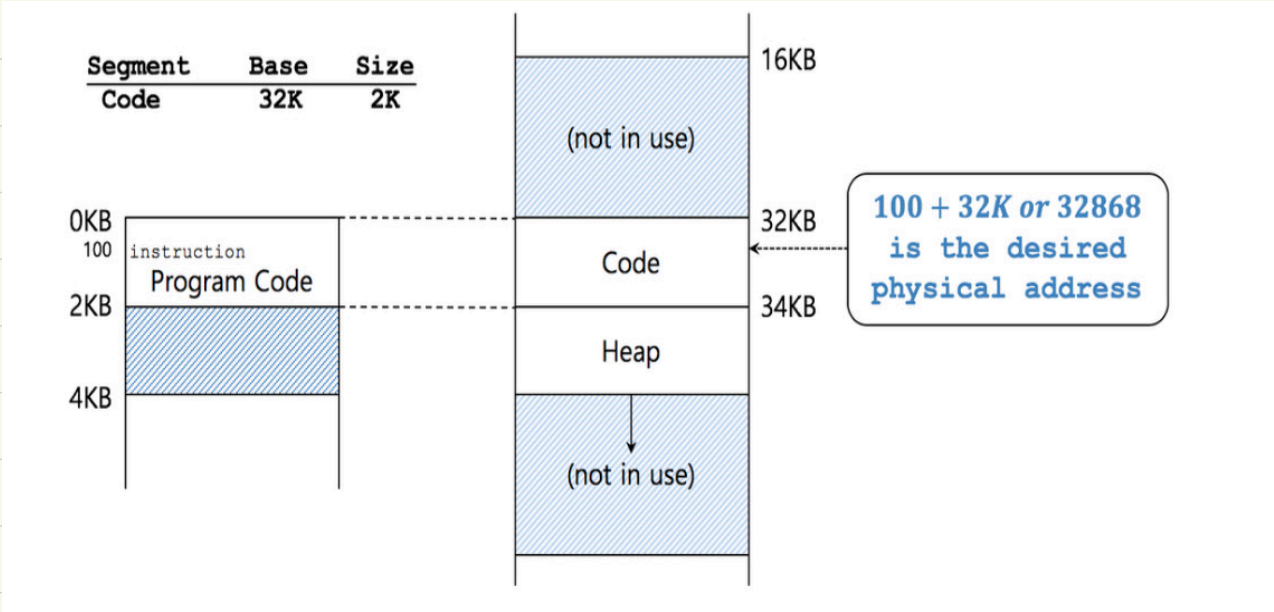
only used memory in each segment is allocated a space, as specified by this size.

eg:- Suppose that a reference is made to virtual address 100 (code segment)

→ virtual address – start address of the corresponding segment

* the offset of virtual address 100 is 100. (the code segment starts at virtual address zero)

⇒ physical address = offset + base



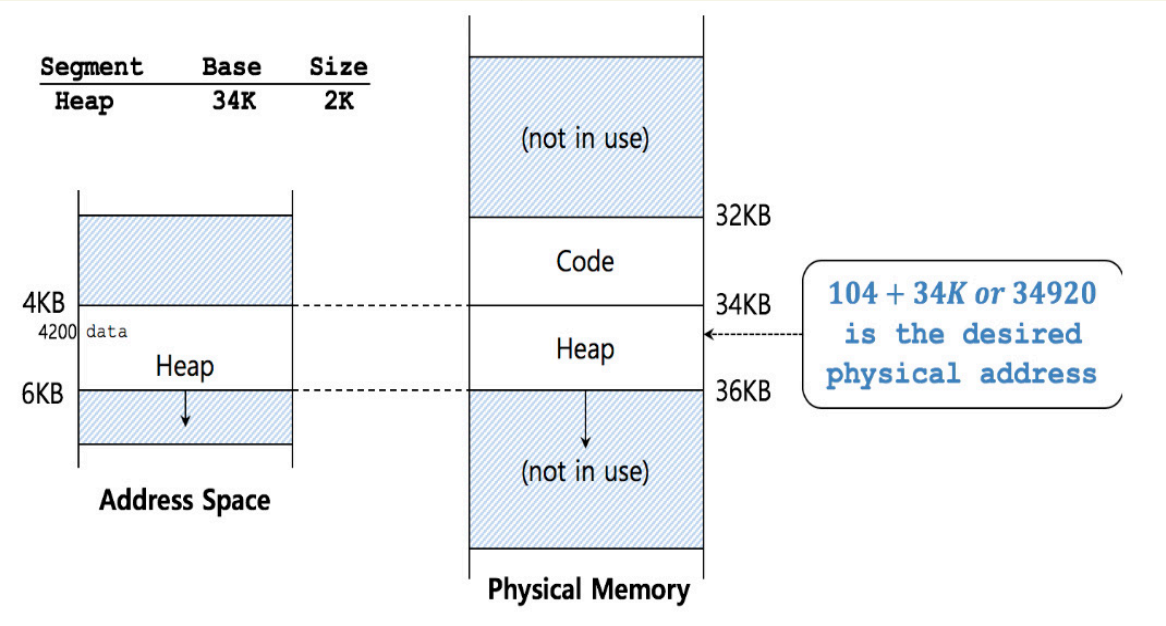
* Now, lets look at an address in the heap, virtual address 4200

(recall, base of heap = 34K)

virtual address + Base = 4200 + 34816 = 39016 ✗ not correct!

What to do? Extract the offset into the heap.

Note that heap starts at 4KB $\Rightarrow$ offset = 4200 − (4×1024) = 104

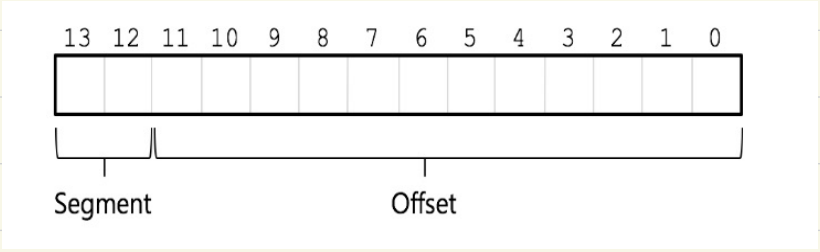


Remark: If an illegal address such as 7KB, which is beyond the end of heap is referenced, the hardware detects that the address is out of bounds → traps into the OS → leads to the famous "segmentation fault"

Question: While translating, how does the hardware know the offset into a segment, and to which segment an address refers?

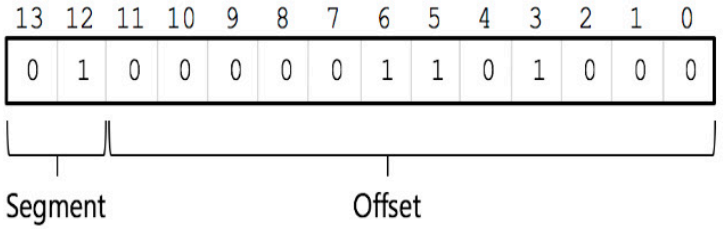
* Explicit approach

Chop up the address space into segments based on the top few bits of virtual address.



eg:- Virtual address 4200 (0100000011101000)

Segment	bits
Code	00
Heap	01
Stack	10
-	11



→ 0000 1101 1000

= 104 in decimal

* Bound check is simple with offset

(just need to check, whether offset ≤ bounds)

```
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

* SEG_SHIFT = 12 * OFFSET_MASK = 00 1111 1111 1111

Disadvantages of this approach

* top two bits can be actually used for four segments, and we have only three segments ⇒ virtual address space is under utilized.

(a way to avoid this problem; by putting code and heap in a single segment and stack in the other, so that only 1 bit is been used for 2 segments)

* each segment is limited to maximum size (eg:- in the above case 4KB)

* Implicit approach

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

* If the address was generated from the program counter (⇒ it was an instruction fetch)
  ⇒ the address is within the code segment.

* If the address is based off of the stack or base pointer, it must be in the stack segment.

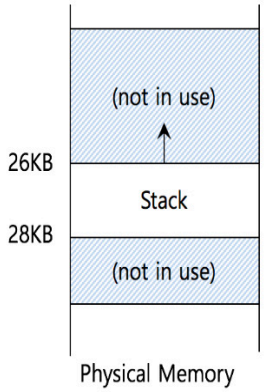
* any other address must be in the heap.

Question: How to refer the stack segment?
What is special about stack segment? → it grows backward.

For instance, in the previous example, in physical memory, stack starts at 28KB and grows back to 26KB (corr. to VA, 16KB to 14KB)
∴ we need a mechanism to know the direction of the growth
↳ extra hardware support!

* the hardware checks which way the segment grows.

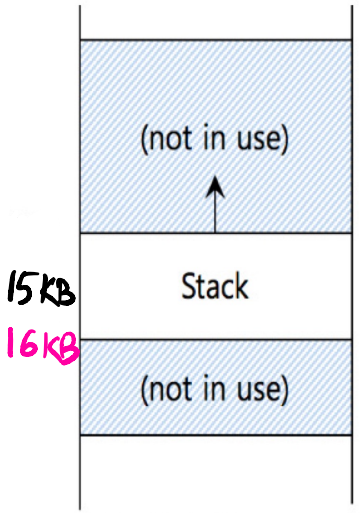
* 1: positive direction 0: negative direction.



Segment Register(with Negative-Growth Support)

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

consider an example, assume we wish to access virtual address 15KB, which should be mapped to 27 kB.



virtual address is 11 1100 0000 0000

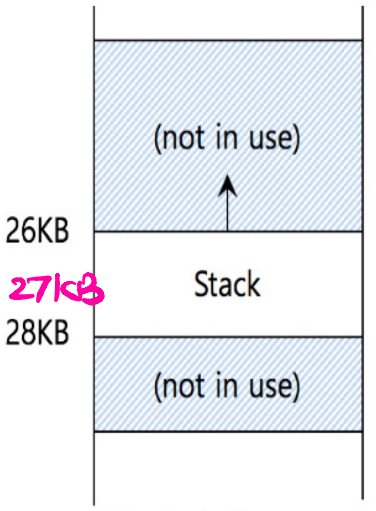
segment: Stack offset: 11

$2^{11}+2^{10}$

$= 2048 + 1024$

$= 3072 = 3\,kB$

This offset makes sense for a +ve growing segment.



But here, the segment grows downwards from a high address toward lower addresses. On hardware's view, the base (28 KB) is the top (highest address) of the segment, and the offset should decrease as you go deeper in it.

⇒ we cannot add the above offset 3KB to the virtual address!

∴ To obtain the correct -ve offset (-1KB), we must subtract the maximum segment size from 3KB.

Here, max segment size = 4KB

⇒ correct -ve offset = +ve offset - max_seg = -1KB (i.e, 1KB below the base address in memory)

⇒ physical address = Base + (-ve) offset

= 28KB + -1KB = 27KB

↳ (Because of the 2KB bound, this should go only ≤2KB below)

Modified algorithm

```
Segment = (virtual address & SEG_MASK) >> SEG_SHIFT
Offset = virtual address & OFFSET_MASK

if (Direction[segment] == 1) {
        if (offset > Bounds[segment])
                Raise Exception (PROTECTION_FAULT)
        else
                PhysAddr = Base[segment] + offset }
```

```
else {
    Disp = Offset - MaxSize[segment]
    if (~Disp) >= Bounds[segment])
        RaiseException(PROTECTION_FAULT)
    else
        PhysAddr = Base[segment] + Disp
    }
```