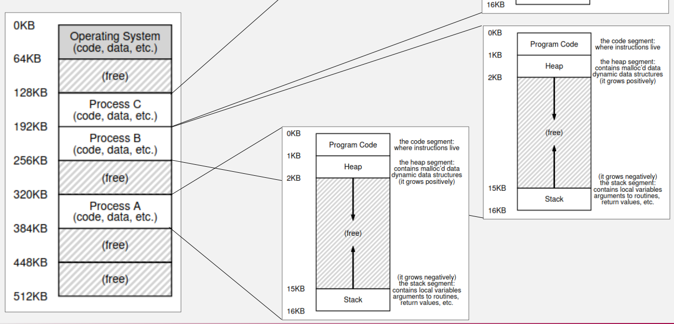
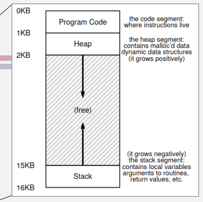
Virtualizing Memory

- each process has its own address space





https://pages.cs.wisc.edu/~remzi/OSTEP/vm-intro.pdf

Address Translation

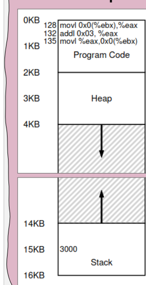
0x0(%ebx) refers to an offset of 0 from the address in register ebx – assuming x's address is in ebx, this refers to x's location in memory

```
int x = 3000;
x = x + 3;
```

```
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
```

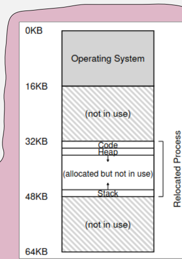
loads the value of x into register eax

what the process sees –



- 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)

– fetch instruction at address 32KB+128
– execute this instruction (load from address 32KB+15KB)
– fetch instruction at address 32KB+132
– execute this instruction
– fetch the instruction at address 32KB+135
– execute this instruction (store to address 32KB+15KB)



what actually needs to happen –

https://pages.cs.wisc.edu/~remzi/OSTEP/vm-mechanism.pdf

Simplifying Assumptions

Assume (for now) –

- the address space is contiguous in physical memory
  - (virtual address space is always contiguous)
- the entire address space fits into physical memory
- every address space is the same size

Static Relocation

- software-based
  - program compiles with addresses starting at 0
  - when the executable is loaded, the loader rewrites memory access instructions to offset the addresses by the location of the address space
- problems with static relocation
  - lacks protection
    - no checks on the address a process is accessing – it could be anywhere!
  - difficult to relocate address space once the process has started running
    - may be needed if there are too many processes to all fit in memory at once

Dynamic Relocation (Address Translation)

- hardware support is needed for efficiency and protection
- add two hardware registers – part of the *memory management unit* (MMU)
  - *base* register
    - contains the physical memory address of the beginning of the virtual address space
  - *bounds* (or *limit*) register
    - contains the size of the address space, or (equivalently) the physical memory address of the upper end of the virtual address space
- usage – when carrying out an instruction involving a memory address
  - the hardware checks that it is within the address space
    - if it is out of bounds, the CPU raises an exception (and the process terminated)
  - if so, the hardware adds the base register to the virtual address

Recap: Hardware Support for Dynamic Relocation

- privileged/kernel mode
  - current mode is stored in a bit in the *processor status word*, a dedicated register storing status information for the CPU
- memory management unit (MMU)
  - base and bounds registers
    - to store where the address space for the current process resides in physical memory
  - circuitry to do address translation and check bounds
- ability to raise exceptions
  - when user processes try to use privileged instructions or access out-of-bounds memory
- privileged instructions
  - for updating base and bounds registers
    - needed by OS to switch between processes
  - to register exception handlers

The OS Side of Things

- the hardware provides the mechanisms
  - address translation
  - bounds checking
- the OS provides the management
  - allocates memory for new processes, deallocates memory from terminated processes
  - keeps track of free vs used memory via *free list*
    - if all address spaces are the same size, free list can just be an array of slots with “in use” or “free” status
    - want data structure for efficient location of a free slot
  - saves and restores base, bounds registers on context switch
    - saved to per-process *process control block* (PCB) in kernel’s address space
  - provides exception handlers to deal with processes that try to do something illegal
    - installed at boot
    - typically terminates the offending process

Limited Direct Execution with Dynamic Relocation

at boot –

OS @ boot (kernel mode)	**Hardware**	**(No Program Yet)**
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**		

when a process runs →

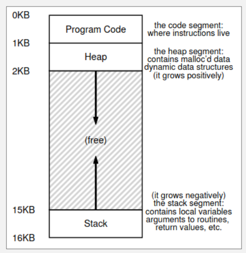
OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
To start process A: allocate entry in process table alloc memory for process set base/bound registers **return-from-trap** (into A)		
	restore registers of A move to **user mode** jump to A’s (initial) PC	
		Process A runs Fetch instruction
	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		

`proc-struct` is the PCB or similar

https://pages.cs.wisc.edu/~remzi/OSTEP/vm-mechanism.pdf

Drawbacks and Limitations

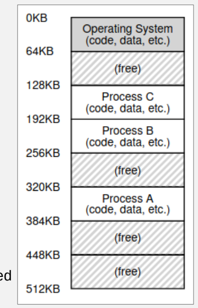
- simplifying assumptions
  - address space occupies contiguous chunk of physical memory
  - address space fits entirely within physical memory
  - address spaces are all the same size
- *internal fragmentation*
  - there is potential for a great deal of wasted space between the heap and the stack – but because this is part of the allocated address space for the process, nothing else can use that memory

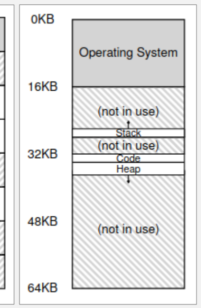


Segmentation

- split the address space into separate code, heap, and stack *segments*
  - each segment can be placed independently into physical memory
- have a separate set of base, bound registers for each segment

non-segmented

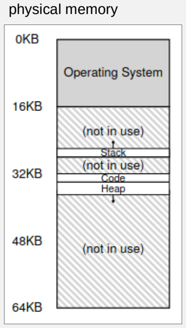




Address Translation Example

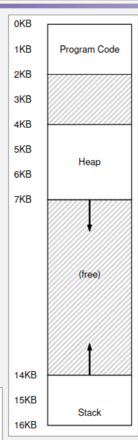
address space

physical memory



base, bound registers

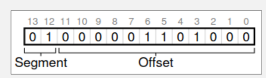
Segment	Base	Size
Code	32K	2K
Heap	34K	3K
Stack	28K	2K



- access virtual address 100
  - in code segment
  - compute offset in segment – 100 – 0KB = 100
  - check bounds – 100 ≤ 2KB
  - access physical address 32K+100
- access virtual address 4200
  - in heap
  - compute offset in heap – 4200 – 4KB = 104
  - check bounds – 104 ≤ 3KB
  - access physical address 34K+104
- out-of-bounds access results in a *segmentation fault*

Which Segment?

- *explicit* approach
  - use the top bits of the virtual address to identify the segment
    - e.g. with two bits 00 – code, 01 – heap, 11 – stack



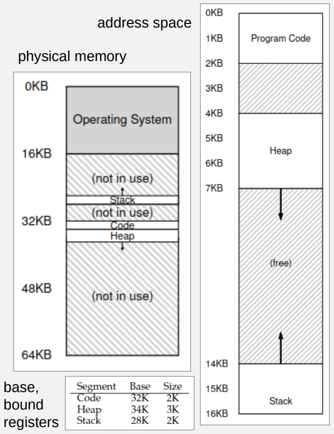
virtual address is $1x2^{12} + 1x2^6 + 1x2^5 + 1x2^3 = 4200$
segment is 01 (heap)
offset is $1x2^6 + 1x2^5 + 1x2^3 = 104$

  - drawbacks and limitations
    - segment 11 is unused, wasting 25% of the address space
      - can combine code and heap segments and use just one segment bit (code/heap or stack)
    - max segment size is limited
      - in the example, the max offset is 2^{12}-1 = 2047 so each segment is limited to 2KB

Which Segment?

- *implicit* approach
  - use the context of the reference to determine the segment
    - address based on the program counter (instruction fetch) → code segment
    - address based on the stack or base pointer → stack
    - otherwise → heap

Address Translation Example



base, bound registers

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

- access virtual address 15KB
  - determine segment
    - 15KB in binary: 11 1100 0000 0000 → stack (11)
  - compute offset in segment
    - stack grows negatively – need offset from the bottom of the stack (end of the segment)
    - $1x2^{11}+1x2^{10}$ = 3KB from the beginning of the segment
    - max segment size is 4KB with 14-bit virtual addresses → 3KB-4KB = -1KB
  - check bounds – |-1KB| ≤ 2KB
  - access physical address 28K-1KB = 27KB

Additional Hardware Support

- the stack and heap grow in opposite directions
  - store direction of growth bit

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

- *code sharing* allows sharing of memory segments between address spaces
  - add *protection bits* to indicate read, write, execute access

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

    - hardware must also check type of access (in addition to bounds checking) during address translation
  - read-only code segments save memory by allowing multiple processes to safely share them
    - e.g. shared libraries
- *fine-grained segmentation* allows many smaller segments
  - supported by a *segment table* in memory