# Chapter 15 Mechanism: Address Translation

Generic technique: comes in addition to limited direct execution

(Hardware-based)

**Address translation:**

- Transform each memory access (instruction fetch, load / store data)
  - Changing virtual address (provided by the instruction or the PC)
  - To a physical address where the information is actually located (in memory)

(OS support)

**Manage memory:**

- Keep track of free/used space
- Maintain control over how memory is used

Goal: **provide illusion that each process has its own private memory**

**15.1 (Initial) Assumptions**

- Address space is placed contiguously in physical memory
- | address space | < | physical memory |
- For all *i, j*: | as( *i* ) | = | as( j ) | : All address spaces have the same size.

## 15.2 An Example

```
void func() {
        int x = 3000;
        x = x + 3;
}
```

Compiler turns this program into assembly:

```
128: movl 0x0(%ebx), %eax   // load 0+ebx into eax
132: add $0x03, %eax        // add 3 to eax register
135: movl %eax, 0x0(%ebx).   // store eax back to memory
```

When executing a program:

We need pull instructions (and data) from RAM

Into the CPU to be executed…

When these instructions run,

The following memory accesses take place:

- PC: 128
- Fetch the instruction at 128
- Execute - load from address 15 KB into eax
- Fetch instruction at 132
- Execute - add (no memory references)
- Fetch instruction at 135
- Execute - store to address 15 KB

Address space range: [0, 16KB]
All memory references must be within these bounds.

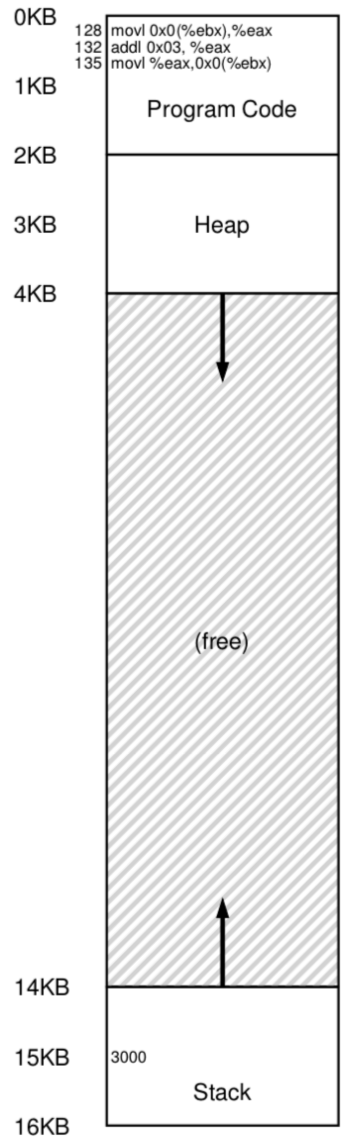


Figure 15.1: **A Process And Its Address Space**

Digression into number systems!! Should be covered in a math course

0x84 0x01 0x00 0x02

1010 = $1*2^3 + 0*2^2 + 1*2^1 + 0*2^0$

= 8 + 0 + 2 + 0

= 10 (base 10)

= A (base 16)

Base 16 (Hexadecimal system)

Replace 10 (base 10) = A (base 16), 11 = B, 12 = C, 13 = D, 14 = E, 15 = F

1110 0011 = E3

1110 = $1*2^3 + 1*2^2 + 1*2^1 + 0*2^0$ = E = 14

0011 = $0*2^3 + 0*2^2 + 1*2^1 + 1*2^0$ = 3

1110 0011= $1*2^7 + 1*2^6 + 1*2^5 + 0*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0$

= 128 + 64 + 32 + 0 + 0 + 0 + 2 + 1 = 227 (base 10) = E3 (base 16)

IP address version 4: 152.94.0.1 = 32 bit number = 1001_1000.0101_1110.0000_0000.0000_0001

E3 (base 16) => 227 (base 10)

BEEF = B*16^3 + E*16^2 + E*16^1 + F*16^0

= 11*4096 + 14*256 + 14*16 + 15*1

= 48879

DEADBEEF (base 16) = 3735928559 (base 10)

Q: How can we relocate this process in memory (transparent to the process itself)??

Q: How can we make the virtual address start at 0, when address space is really at some other location?

**Example Fig 15.2.** Divide physical memory 64KB into four address spaces each of 16KB.

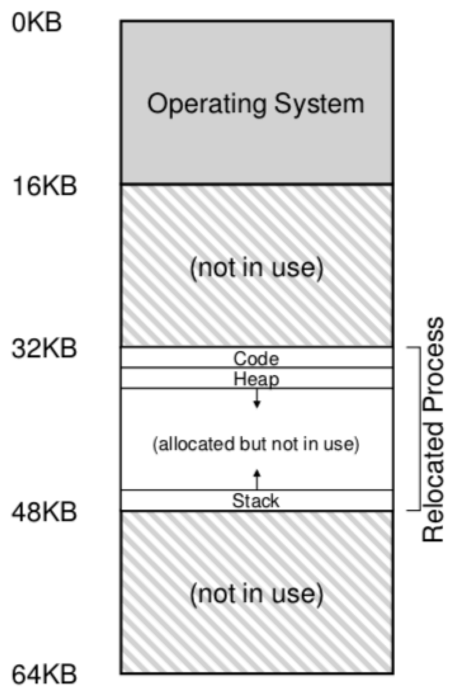


Figure 15.2: **Physical Memory with a Single Relocated Process**

**15.3 Dynamic Relocation**

(also called: Base and Bounds)

Old CPUs had two registers:

- Base register
- Bounds register

With these, we can

- Place address space anywhere in physical memory
- Ensure process only access its own address space

Program is written and compiled to be loaded at address 0x0000.
When run OS decides:

- Where in physical memory to load it: X
- Sets the base register to X

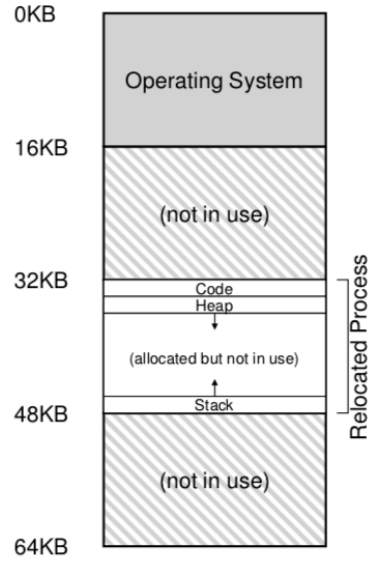


Figure 15.2: **Physical Memory with a Single Relocated Process**

**Example.** Fig 15.2 base = 32KB

When process runs

- memory references generated by the process (CPU) are translated:
  - Physical Address = Virtual Address + base

Process (CPU) generating virtual addresses…

**Example.** Tracing a single instruction

128: movl 0x0(%ebx), %eax

PC is set to 128.

To fetch this instruction HW computes

- Physical address = PC + value of base register (32 KB = 32768) = 32768 + 128 = 32896

Execute move instruction:

- Processor generates a load from virtual address 15 KB
- Physical address = 15 KB + 32 KB = 47 KB
- HW fetches the desired content @ 47 KB into the CPU

Because this happens at runtime: **dynamic relocation**

**The Bounds Register**

Why?: To help with protection

CPU will check that a memory reference is within bounds to ensure it is legal.

```
If virtual address > bounds OR virtual address < 0 then
    Raise an exception (terminate the process)
```

**Memory Management Unit (MMU):**

- CPU registers: base and bounds
-

**Example Translations**

Consider an address space with size 4 KB (4096)

| Virtual Address | Physical Address |
|---:|---|
| 0 | 16 KB (16384) |
| 1 KB | 17 KB |
| 3000 | 16384 + 3000 = 19384 |
| 4400 | Fault (out of bounds) |

## 15.4 Hardware Support: A Summary

| **Hardware Requirements** | **Notes** |
|---|---|
| Privileged mode | Privileged instructions (kernel mode) to modify base and bounds registers. Can’t let user mode processes change these registers… |
| Base/bounds registers | Need pair of registers per CPU to support address translation and bounds checking… |
| Ability to translate virtual addresses and check if within bounds | Circuitry to do translation and check limits; quite simple… |
| Privileged instructions to update base/bounds | OS must be able to set these values before letting the user program run… |
| Privileged instructions to register exception handlers | OS must be able to tell hardware what code to run if exception occurs… (bounds violation) |
| Ability to raise exceptions | When processes try to access privileged instructions or out-of-bounds memory |

**15.5 Operating System Issues**

OS must:

- When process is created
  - Find space for the process's address space in physical memory
    - Search free list (a data structure) for available room for proc's address space
    - Mark space as used
- When process is terminated
  - Reclaim all of the process's memory
    - Put it back on the free list
- Save and restore (base, bounds)-register pair on context switches (switching between processes)
  - Saved in process control block (PCB) struct
- Provide exception handlers
  - To terminate offending processes

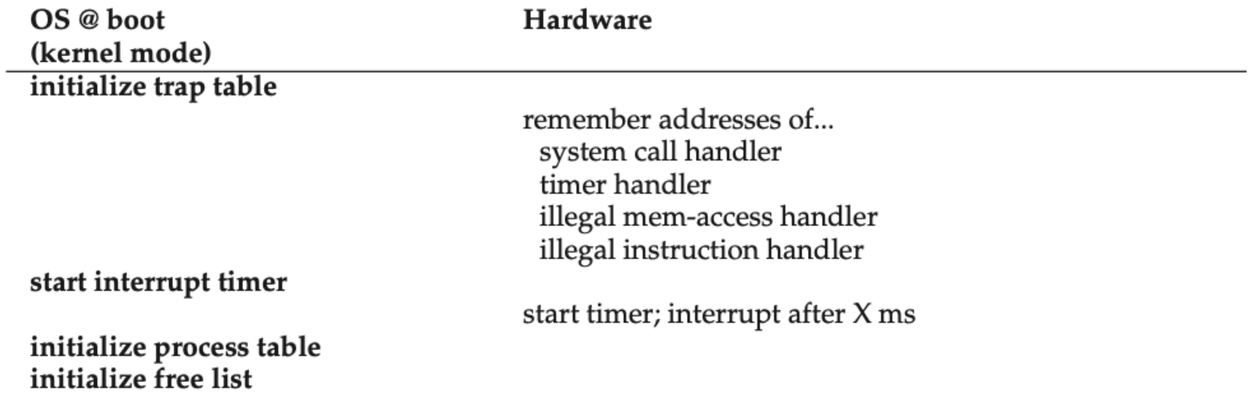
With support for base and bounds — it is easy:

To move a process, when it is stopped. OS simply:

- Copies the address space from currently location to new location
- Update the base register (in the PCB) to point the new location

(the process would be oblivious to this happening …)

Fig. 15.5 Limited Direct Execution Protocol (w/ Dynamic Relocation)



| **OS @ boot (kernel mode)** | **Hardware** |
|---|---|
| **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** | |

| **OS @ run (kernel mode)** | **Hardware** | **Program (user mode)** |
|---|---|---|
| **To start process A:** | | |
| allocate entry in process table | | |
| allocate memory for process | | |
| set base/bounds 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 and perform fetch | |
| | | Execute instruction |
| | If explicit load/store: Ensure address is in-bounds; Translate virtual address and perform load/store | |
| | | ... |
| | **Timer interrupt** | |
| | move to **kernel mode** | |
| | Jump to interrupt handler | |

| | | |
|---|---|---|
| **Handle the trap**<br>Call `switch()` routine<br>save regs(A) to proc-struct(A)<br>(including base/bounds)<br>restore regs(B) from proc-struct(B)<br>(including base/bounds)<br>**return-from-trap** (into B) | | |
| | restore registers of B<br>move to **user mode**<br>jump to B's PC | |
| | | **Process B runs**<br>Execute bad load |
| | Load is out-of-bounds;<br>move to **kernel mode**<br>jump to trap handler | |
| **Handle the trap**<br>Decide to terminate process B<br>de-allocate B's memory<br>free B's entry in process table | | |

Figure 15.5: **Limited Direct Execution Protocol (Dynamic Relocation)**

## 15.6 Summary

Address translation w/base&bounds

- Fast translation
  - A single add and a single compare operation
- Transparent to the process
- Provides protection

Problems w/base&bounds:

- if stack and heap space are not big
  - Space between is wasted
    - Called internal fragmentation
  - Caused by the fixed-size slots (due to our assumption of same size address spaces)

Next chapter: Segmentation

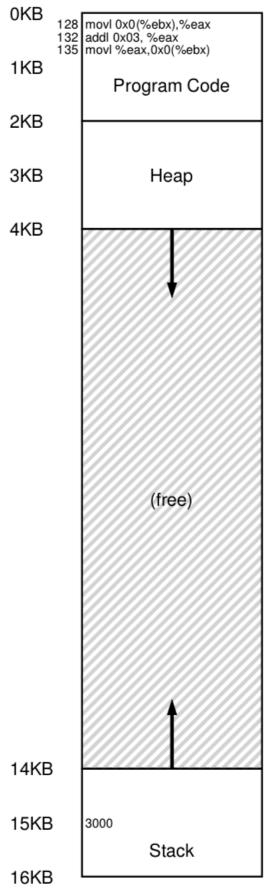


Figure 15.1: **A Process And Its Address Space**