Sistemas Operativos: VM Introduction

Pedro F. Souto (pfs@fe.up.pt)

April 22, 2020

CPU Virtualization (Before Concurrency)

Goal Illusion that each process has its own CPU

Mechanism Limited Direct Execution

- Processes run directly on the HW (CPU)
- The OS intervenes only at critical points
  - To prevent processes from interfering with other processes or the OS
  - To maintain control over the HW (interrupts, in particular timer interrupts).

Memory Virtualization

Goal Illusion that each process has its own memory (address space)

Mechanism Address translation

- On every memory access, the VM subsystem maps the virtual address to a physical address

Requirements

Efficiency both in terms of time and space

Control processes must not access the address space of other processes, unless allowed

Transparency processes are not aware that the physical memory is shared among the OS and the running processes

To satisfy these requirements the OS needs help from the HW

Assumptions

1. The user's address space is mapped contiguously in physical memory
2. The size of the address space is smaller than the size of physical memory
3. The size of the address space is the same for all processes

These unrealistic assumptions will be dropped as we go.

Process Virtual Address Space

- Each process has its own virtual address space
  - Beginning at virtual address 0
  - With a size of 16 KiB
- All program addresses are virtual and range from 0 to 0x3FFF
  - Let the code:

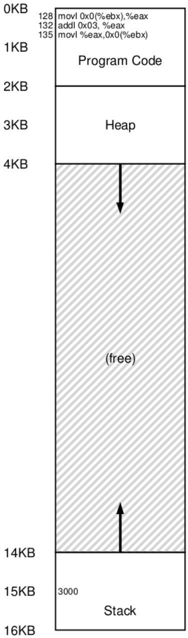
```
void func() {
    int x;
    x = x + 3; // this is line of code of interes
```

  - It may be compiled to the following:

```
0x80: movl 0x0(%ebx), %eax    ;load 0+ebx into eax
0x84: addl 0x03, %eax         ; add 3 to eax regis

0x87: movl %eax, 0x0 (%eax)   ; store eax back to
```

  - Assume that `x` is at address 0x3A00 (15KiB), as shown

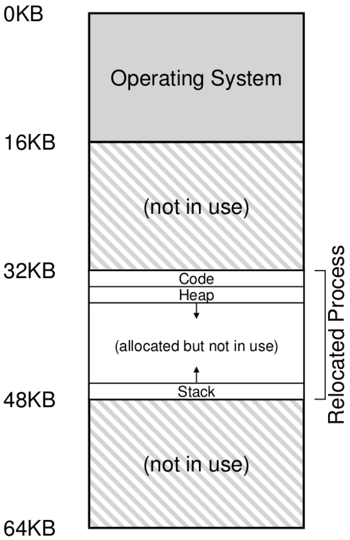


Physical Memory

- Assume that the computer has 64 KiB of main memory
  - And that the OS is loaded to the first 16 KiB
- When the OS loads that program to run, it allocates a region of physical memory that is not used by any other program, e.g. starting at 32 KiB

Question How to relocate the process in memory in a way that is transparent to the process?

- I.e. give the illusion that the process's address space starts at 0, when it is located at another physical address.



Virtual Address Translation

Let the code:

```
0x80: movl 0x0(%ebx), %eax    ;load 0+ebx into eax
0x84: addl 0x03, %eax         ; add 3 to eax register
0x87: movl %eax, 0x0 (%eax)   ; store eax back to memory
```

If mapped at 0x8000

Virtual addr.		Physical addr.	Comment
0x0080	$\longrightarrow$	0x8080	Fetch first instruction
0x3A00	$\longrightarrow$	0xBA00	Load value of x
0x0084	$\longrightarrow$	0x8084	Fetch second instruction
0x0087	$\longrightarrow$	0x8087	Fetch third instruction
0x3A00	$\longrightarrow$	0xBA00	Store new value of x

If mapped at 0x4000

Virtual addr.		Physical addr.	Comment
0x0080	$\longrightarrow$	0x4080	Fetch first instruction
0x3A00	$\longrightarrow$	0x7A00	Load value of x
0x0084	$\longrightarrow$	0x4084	Fetch second instruction
0x0087	$\longrightarrow$	0x4087	Fetch third instruction
0x3A00	$\longrightarrow$	0x7A00	Store new value of x

Dynamic Relocation (Base and bounds)

Idea Use two HW registers

Base which keeps the physical address to which virtual address 0x0 is mapped. E.g. 0x8000 (or 0x4000)

Bounds/limit which keeps the size of the virtual address space

- Allows relaxing the assumption that all address spaces have the same size

Dynamic Relocation (Base and bounds)

Idea Use two HW registers

Base which keeps the physical address to which virtual address 0x0 is mapped. E.g. 0x8000 (or 0x4000)

Bounds/limit which keeps the size of the virtual address space

- Allows relaxing the assumption that all address spaces have the same size

Address Translation by HW on every memory access

```
Phys_Addr = [Base] + Virt_Addr
```

E.g.

0x8000 + 0x0080 $\longrightarrow$ 0x8080

What is the bounds/limit register for?

Dynamic Relocation (Base and bounds)

Idea Use two HW registers

- **Base** which keeps the physical address to which virtual address 0x0 is mapped. E.g. 0x8000 (or 0x4000)
- **Bounds/limit** which keeps the size of the virtual address space
  - Allows relaxing the assumption that all address spaces have the same size

Address Translation by HW on every memory access

```
Phys_Addr = [Base] + Virt_Addr
```

E.g.

0x8000 + 0x0080 $\longrightarrow$ 0x8080

What is the bounds/limit register for?

- **Protection** HW checks if the address is within bounds and raises an exception if not
  - **Exception handler belongs to OS** and most likely kills offending process

Dynamic Relocation: HW requirements

Hardware Requirements	**Notes**
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

Dynamic Relocation: OS involvement

OS Requirements	Notes
Memory management	*Need to allocate memory for new processes;* *Reclaim memory from terminated processes;* *Generally manage memory via* ***free list***
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise;* *likely action is to terminate offending process*

Dynamic Relocation: Evaluation

Pros

Fast

Simple

Little memory overhead

- Need only store the values of 2 registers, per process

Cons

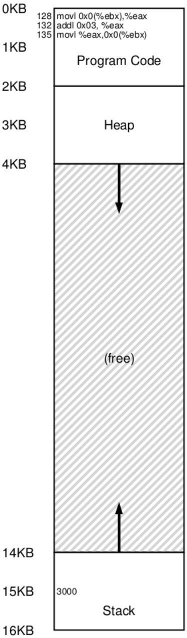
Not flexible

- Hard to grow the size of the AS

Wastes memory Especially for large AS

- The space between the heap and the stack needs to be allocated, even if it is not used

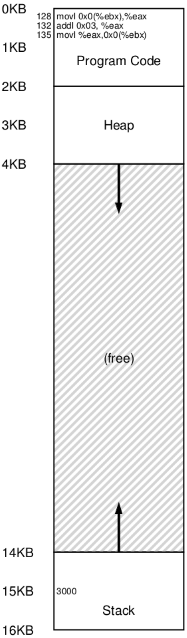
Segmentation: Idea



Observation with B&B, for large AS, there is a lot of memory space that is not used but that is allocated nevertheless

How to avoid this?

Segmentation: Idea



Observation with B&B, for large AS, there is a lot of memory space that is not used but that is allocated nevertheless

How to avoid this?

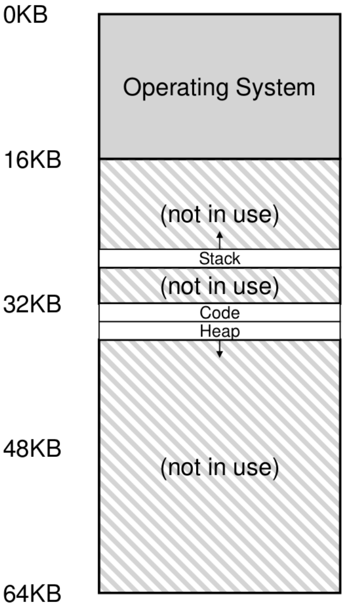
Observation rather than a single contiguous memory region, an AS is usually composed of several contiguous memory regions: segments

- E.g. code, heap, stack
- Each of these can be relocated independently

Idea use a pair of B&B register per segment.

- Again, for efficiency reasons address translation is done by HW

Segmentation: Independent Segment Relocation



- Only the space that is actually used needs to be allocated
- The Memory Management Unit (MMU) consists only of 3 base and bounds registers pairs:

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

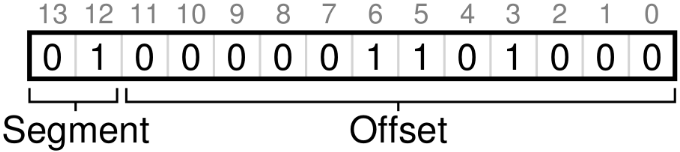
Segmentation: Issue

Issue How to determine which base/bounds pair to use?

Segmentation: Issue

Issue How to determine which base/bounds pair to use?

Explicit Use some bits of the address to specify the segment. The remaining are used for the offset.




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

Implicit E.g. if the address was formed using the PC then use the "code segment", if the address was formed using the SP then use the stack segment

Segmentation: OS Involvement

New process
: Need to allocate physical memory for all segments of the new process

Growth of segment
: Need to allocate more physical memory

- Preferrably, using free memory contiguously to the physical memory already being used by the segment
- Alternatively, need to copy contents of segment to the newly allocated physical memory

Context switch

1. For each segment of the process that was running, save in the PCB the values of the pair of B&B registers
2. For each segment of the process selected to run, load from the PCB the values of the pair of B&B registers

Segmentation: Evaluation

Pros

- Fast
- Simple
- Little memory overhead
  - Need only store the values of 2 registers per segment, per process
- Supports sparse AS efficiently
- Fine-grained protection
  - E.g., allow only read-execute on the code segment, and only read-write on the heap and stack segments

Cons

- Not flexible
  - Hard to grow the size of the segments sometimes
- Wastes memory
  - Only external memory fragmentation

External Memory Fragmentation