BTI1341 – Operating Systems

Part 1: Virtualization – 3) Memory 1

Autumn 2025-26 `master@352a46f (20250901-155000)`

P. Mainini / E. Benoist / C. Fuhrer / L. Ith

Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences

Outline

- Address Spaces
- Address Translation
- Dynamic Relocation
(Base and Bounds)
- Interlude: Memory Management
- Segmentation
- Appendix

Introduction

Up until now, we have investigated how to *virtualize the CPU:*

- Using the *process* as a basic abstraction
- *Limited direct execution* as enabling mechanism
- *Scheduling policies/disciplines* for control

In this part, we proceed in a similar fashion to *extend* limited direct execution to virtualize memory:

- Using the address space as abstraction
- Address translation as mechanism
- Segmentation and later paging as memory management schemes (policies)

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

Address Spaces

Initial Memory Layout

Early operating systems where basically *libraries* and there was a *single* running program (and no abstraction):

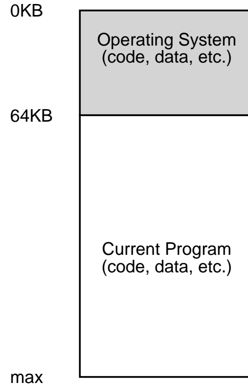

Figure: Single Process Memory Layout

Courtesy of [ADAD18]

Multiple Processes

Multiprogramming and time sharing introduced multiple processes which where potentially ready to run (or waiting for I/O). While leading to better *utilization* and *efficiency*, it also introduced new challenges for memory management:

- If the whole memory (besides the OS) is assigned to a process, it needs to be *exchanged* at every process switch
- Exchanging memory is highly inefficient (not comparable to a "normal" context switch)
- Only solution: having multiple processes simultaneously in memory

This leads to a new problem: protection is required between processes (and the OS).

Multiple Processes in Memory

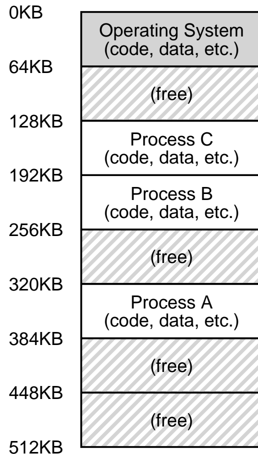

Figure: Memory Layout with Three Processes

Courtesy of [ADAD18]

The (Virtual) Address Space

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

Definition

An address space is the abstraction of physical memory as seen from a process:

- Contains its full memory state: code, initialized data, dynamic stack and heap etc.
- Provides virtual addresses which normally do not correspond to physical addresses
  - Program may be loaded physically at a different address
  - Address space can be larger than physical memory
- Provides isolation for protection and *ease of use*:
  - Every process sees its *own* address space
  - No need to care about memory layout and other processes

An Example Address Space

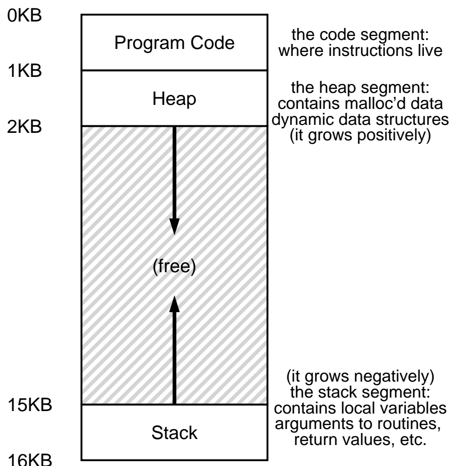

Figure: A 16 KiB Address Space

Courtesy of [ADAD18]

Goals for Address Spaces

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

Some of the design goals given in the *Introduction (00)* also apply in particular for address spaces:

Transparency Processes should not notice that their memory is virtualized. The OS provides the illusion that the process has access to a large, contiguous memory space on its own.

Flexibility Processes may organize their address spaces however they like.

Efficiency Memory virtualization must be highly efficient, as memory is *accessed permanently*.

Protection Individual processes must be protected from each other and the OS itself also needs protection. A process must thus not be able to access or affect foreign memory (isolation).

Address Translation

Overview

To virtualize the CPU, we used the mechanism of *limited direct execution:*

- A process runs directly on the hardware itself
- *At some points*, it is interrupted and the control is returned to the OS (with *hardware support*: timer)

This provides both *efficiency* and *protection*, which we also want to achieve for memory virtualization:

- *At some point*, we want the OS to be in control of any memory accesses made by a process
- Memory access has to be efficient
- Again, this is not feasible without support from the hardware

Address Translation

The main idea behind (hardware-based) address translation is the distinction between different kinds of memory addresses:

Virtual addresses as seen from the running process

Physical addresses used in hardware to address physical memory

For *every* memory access (fetching an instruction, load or store a value), a translation has to be performed. There are two roles in this:

OS Manages memory, knows which parts belong to which process and configures the hardware.

Hardware The Memory Management Unit (MMU) provides efficient translation between virtual and physical addresses.

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

A First (Simplistic) Model

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

As before, we make some (unrealistic) assumptions first, which we will relax later when introducing more complex concepts:

1. A user address space must be placed *contiguously* in physical memory
2. Its size *must not be too big*, i.e. it is smaller than physical memory
3. Every address space is exactly *the same size*

Example I

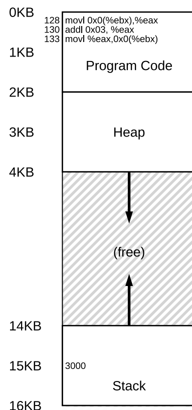

Figure: Example Address Space

Courtesy of [ADAD18]

Example II

Assume the following function:

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

Generated x86 assembly could look like this (address of `x` is in `EBX`):

```
128: movl 0x0 (%ebx), %eax   ; load value of x into EAX
130: addl $0x03, %eax        ; add 3
133: movl %eax, 0x0 (%ebx)   ; write EAX back to x
```

Example III

Assuming the given example address space, the following memory accesses take place:

1. Fetch instruction at address 128
2. Execute this instruction (load from address at 15K)
3. Fetch instruction at address 130
4. Execute this instruction (no memory reference)
5. Fetch instruction at address 133
6. Execute this instruction (store to address at 15K)

Example IV

For the process, the address space starts at address 0 and goes up to 16 KiB. In reality however, the address space of the process will be relocated and probably looks more like this:

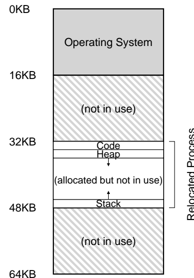

Figure: Physical Memory with Single Relocated Process

Courtesy of [ADAD18]

Dynamic Relocation
(Base and Bounds)

Dynamic Relocation I

A first approach (late 1950s) to address translation is dynamic relocation or base and bounds . The idea is simple:

- Introduce two new *hardware registers*, called **`base`** and **`bounds/limit`**
- Programs are compiled to start at a fixed address, usually 0
- When loading, the OS decides upon the size and start address of the corresponding address space in physical memory
- **`base`** is set to the start address
- **`bounds`** is set to the highest *legal* virtual address
- References to addresses higher than **`bounds`** lead to a fault (to be handled by the OS)

Memory references are then simply *translated* (*at run time!*) as follows:

```
physical address = base + virtual address
```

Dynamic Relocation II

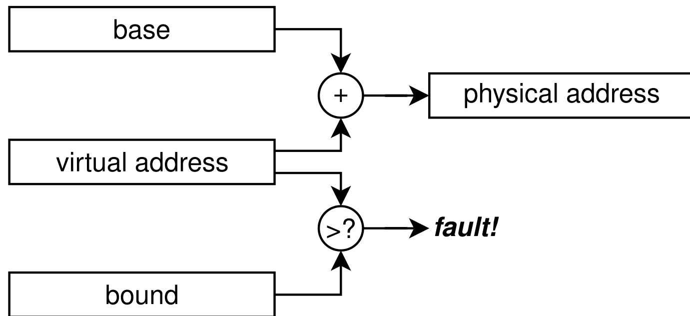

Figure: Base and Bounds

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

Example, Revisited

Looking at the following instruction from the previous example:

```
128: movl 0x0 (%ebx), %eax    ; load value of x into EAX
```

Dynamic relocation works as follows:

`base = 32768` / `bounds = 16383`

1. Program counter (PC) is set to 128 (virt. addr. ≤ 16383 ✔)
2. To fetch the instruction, `base` is added: $32768 + 128 = 32896$
3. The instruction is fetched from physical address 32896
4. A memory load from address 15K is requested (≤ 16383 ✔)
5. To load the data, `base` is added: $32768 + 15000 = 47768$
6. Data is loaded from 47768

Hardware Requirements

Requirement	Notes
1. Privileged mode	To enable privileged instructions.
2. **base** and **bounds** register	Per CPU, for address translation and bounds check.
3. Ability to translate virtual addresses, check bounds	In HW for efficiency.
4. Instructions to update **base** and **bounds**	Required by OS prior running a process, *privileged*.
5. Ability to raise exceptions	If a process executes privileged instructions or accesses out-of-bounds memory.
6. Instructions to register exception handlers	*Privileged*.

OS Requirements

Requirement	Notes
1. Memory management	Allocate and reclaim memory for/from processes, in general using some free list
2. `base` and `bounds` management	Must be performed at each context switch
3. Exception handling	Do something when a process misbehaves, in general: terminate it

Note: Given current Assumptions 2 (address spaces smaller than physical mem) and 3 (all are equal in size), memory management so far is simple. What would have to be done?

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

Limited Direct Execution I

With Base and Bounds

OS @boot (kernel mode)	Hardware	
Initialize trap table		
	Store addr. of syscall-, timer- and **mem-fault handlers**	
Start interrupt timer		
	Start timer Interrupt CPU in X ms	

OS @run (kernel mode)	Hardware	Process (user mode)
Start A		
Create process list entry		
Allocate memory		
Set base/bounds (A)		
`return-from-trap`		
	Restore regs(A) Move to user mode Jump to PC (A)	

Limited Direct Execution II

With Base and Bounds

OS @run (kernel mode)	Hardware	Process (user mode)
		A is running Instruction fetch or load/store data
	Translate address, check bounds, fetch	
		Execute instruction ...
	Timer interrupt Save regs(A) Move to kernel mode Jump to trap handler	
Handle trap Call **`switch()`**... ...save regs(A) ...restore regs(B) ...set base/bounds (B) **`return-from-trap`**		

Interlude: Memory Management

Overview

We have seen, that managing memory is easy when free space is available in *fixed sized units*. Otherwise, it becomes challenging:

- When using *segmentation* (next section)
- For user space memory allocators (e.g. `malloc()` implementations)

The reason for this is external fragmentation :

- Over time, memory regions of various sizes are allocated and subsequently de-allocated
- This leads to *holes* in memory
- It then becomes more and more difficult to find free regions of appropriate size for new allocations

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

External Fragmentation

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

Figure: Example for External Fragmentation

Courtesy of [ADAD18]

In this example, there are 24 KiB free, however the OS cannot serve a process requesting 20 KiB due to external fragmentation.

Compaction

A possible solution for external fragmentation could be compaction .
However, in general it is either too expensive (copying memory) or
impossible (user space allocators: OS has no control over assigned regions).

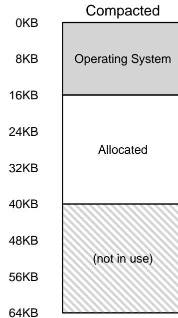

Figure: After Compaction

Courtesy of [ADAD18]

Mechanisms for Memory Management

Memory management typically tracks *free space* in some form of free space list. [1] Example:

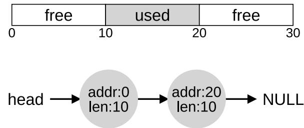

Figure: Free Space and Corresponding List

Courtesy of [ADAD18]

Two basic, low-level mechanisms are then used for managing free space: splitting and coalescing .

[1] Other forms are possible, e.g. using a bit map. The list may be implemented as separate data structure or within the managed memory space. ★ See [ADAD18, Section 17.2] for details.

Splitting

Upon request for memory, splitting is applied:

1. Search the list for a large enough chunk of free space
2. Split it and return one part to the caller

Example:

Initial free space list:

Address	Length
0	10
20	10

After requesting 1 byte of memory:

Address	Length
0	10
21	9

Coalescing

When coalescing, adjacent free regions are combined into a single region.

Example:

Initial free space list:

Address	Length
0	10
20	10

After return of the 10 bytes in the middle:

Address	Length
10	10
0	10
20	10

After coalescing:

Address	Length
0	30

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

Strategies I

Ideally, memory management should be *fast* and *minimize fragmentation*. Different strategies exist:

Best fit Search the full list for equal sized or bigger chunks and **choose the smallest.**
Goal: space efficiency. Problem: Performance.

Worst fit Opposite to best fit: **search largest** chunk and split it.
Goal: keep big chunks free. Problems: Performance, excess fragmentation.

First fit Simply **take the first** chunk that fits.
Goal: speed. Problem: Pollution with small allocations.

Next fit Keep a **pointer to the last allocation**, then perform first fit from there.
Goal: equal distribution of allocations. Problem: As for first fit.

Strategies II

In practice, it is difficult to tell which strategy is the best. It heavily depends on utilization, type of workloads etc.

Neither *first fit* nor *best fit* is clearly better than the other in terms of memory utilization, but *first fit is generally faster*.

Next fit in is comparable to *first fit*.

Overall, all three of them perfom better than *worst fit*, which is bad at performance and storage utilization.

It can thus be assumed, that *first fit* performs best in terms of performance and memory utilization, while at the same time providing a straight forward implementation.

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

Buddy Allocation

Besides the simple strategies given, various more complex ones have been developed. As a final example, we present the buddy allocator . Idea: *Make coalescing efficient* by allocating accordingly.

Example:
A request for 7 KiB is served out of 64 KiB of free space.

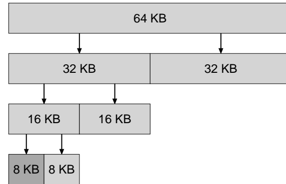

Figure: The Buddy Allocator

Courtesy of [ADAD18]

Note: buddy allocation may suffer from internal fragmentation, introduced next.

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

Segmentation

Problems of Dynamic Relocation I

Dynamic relocation looks like a good approach:

- It is *efficient* as it can be implemented in HW using only two registers
- It offers the required *protection* as no process can access memory outside of its address space

However, it also has severe drawbacks:

- Inefficient, wastes memory
- As it allocates memory for the *whole* address space of a process, it suffers from internal fragmentation
- It makes it hard to run programs with address spaces larger than physical memory

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

Problems of Dynamic Relocation II

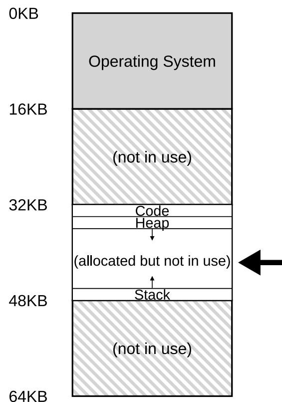

Figure: Internal Fragmentation

Courtesy of [ADAD18], Own Modification

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

Introducing Segmentation I

The next approach, segmentation is simple: *individual* `base` and `bounds` registers *per segment*. A segment is a single, contiguous region within an address space.

We already know potential segments: `text` (or `code`), `heap` and `stack`. Introducing registers for those in the *MMU* allows for efficient and flexible memory management. Example:

Segment	Base	Size
`code`	32K	2K
`heap`	34K	2K
`stack`	28K	2K

Note: This is only one possible layout, segmentation allows for many more segments.

Introducing Segmentation II

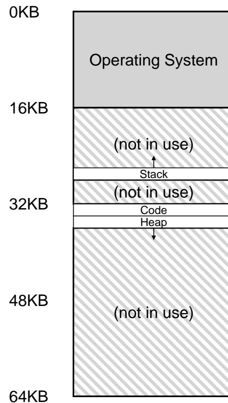

Figure: Physical Memory with Segmentation

Courtesy of [ADAD18]

Example

Assuming an address space of size 16K and the following layout:

Segment	Virt. Addr.	Base	Size
`code`	$0 - 2\text{K}$	32K	2K
`heap`	$4 - 6\text{K}$	34K	2K
`stack`	$14 - 16\text{K}$	28K	2K

Example 1:
Address 100 (`code`): $100 + 32\text{K} = 32868$ ($100 < 2\text{K}$ ✔)

Example 2:
Address 4200 (`heap`): $4200 + 34\text{K} = 39016$ ($4200 \not< 2\text{K}$ ✖)

Calculation must be relative to *offset* within the segment:
Offset: $4200 - 4096 = 104$ (`heap` starts at 4K)

$\Longrightarrow\ 104 + 34\text{K} = 34104$ ($104 < 2\text{K}$ ✔)

Addressing

How does the hardware know, to which segment a *virtual* address belongs?

Idea: *explicitly* use a part of the address to denote the segment.[2]

E.g. using the two *topmost* bits, we have segments `00`: $0-4$K, `01`: 4K-8K, `10`: 8K-12K and `11`: 12K-16K.

Example: Address 4200 (`heap`) is segment `01` ($2^{12} = 4096$) and offset $104 = 2^6 + 2^5 + 2^3$:

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	0	1	0	0	0

Segment (bits 13–12), Offset (bits 11–0)

Figure: Virtual Address 4200

Courtesy of [ADAD18]

[2] Another approach is to determine the segment *implicitly*, depending on how the address was formed (e.g. using program counter / instruction pointer).

HW Addressing Algorithm

The following pseudocode shows a possible addressing algorithm for placing memory content in a register (to be run in the MMU):

```
OFFSET_MASK = 0xFFF    // 00111111111111
SEG_MASK = 0x3000      // 11000000000000
SEG_SHIFT = 12

Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
Offset = VirtualAddress & OFFSET_MASK

if (Offset >= Bounds[Segment])
  RaiseException(PROTECTION_FAULT)
else
  PhysAddr = Base[Segment] + Offset
  Register = AccessMemory(PhysAddr)
```

Note: `Base` *and* `Bounds` are *arrays with per-segment addresses.*

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

Negative Stack Growth

We have left out so far, that the stack *grows negatively*. To support this, the HW must know the *segment direction*, e.g. using an additional bit in the segment table. To obtain the correct offset, it must then *subtract* the maximum segment size (MSS) from the offset.[3]

[3] MSS is not necessarily equal to segment size set in the MMU!

Example

Segment	Virt. Addr.	Base	Size	Growth
`code`	$0 - 2K$	$32K$	$2K$	$+$
`heap`	$4 - 6K$	$34K$	$2K$	$+$
`stack`	$14 - 16K$	$28K$	$2K$	$-$

Access to location 15360 (`11 110000000000`) in the stack (note: physical address range $28 - 26K$!).

Offset would be 3072, but adding it to `base` ($28K$) is out of bounds!

Knowing that growth is negative, subtract MSS from offset:
$3072 - 4K = -1024 = -1K$. Then, $28K + (-1K) = 27K$ □

The bounds check can simply be adapted to use the absolute value:
$|-1024| < 2K$✔

Segment Sharing

Segments, which are *not written to*, may be shared between multiple address spaces in order to reduce memory usage (e.g. for code sharing). For this, we require some protection bits , indicating the protection of a segment.

Thus, our final segmentation table looks as follows:

Segment	Base	Size	Growth	Protection
`code`	$32K$	$2K$	$+$	read, exec
`heap`	$34K$	$2K$	$+$	read, write
`stack`	$28K$	$2K$	$-$	read, write

Changes for the OS

Segmentation requires the HW to provide `base` and `bounds` registers *per segment*. But also on the OS side, some work is required:

- On a *context switch*, the OS must save and restore the corresponding segment registers
- Memory management becomes more complicated: Segments can have different sizes (also per process) and are not contiguously arranged. This now leads to *external fragmentation*.

Note: Also internal fragmentation in large, but only sparsely used segments remains unsolved.

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

Outlook on Memory 2

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix

We have introduced **address spaces** and had a look at **address translation**, the fundamental mechanism used for memory virtualization. As a first scheme, **segmentation** was presented:

- Segments are of *variable* size, this allows for flexible (and sparse) memory organization
- Mostly fixes *internal fragmentation*
- Supports code sharing and protection

However, segmentation is not in use today anymore:[4]

- Suffers from *external fragmentation*, requires *compaction*
- None of the mainstream OSes has adopted it
- Support for it got dropped with the `x86-64` architecture

In the next part, we will thus look at **paging**, which is widely in use today.

[4] At least not on general purpose platforms and OSes.

Appendix

Bibliography I

[ADAD18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.00 ed., Arpaci-Dusseau Books, August 2018, Available online: `http://ostep.org`.

Address Spaces
Address Translation
Dynamic Relocation (Base and Bounds)
Interlude: Memory Management
Segmentation
Appendix