Memory

Johan Montelius

KTH

2019
Memory layout for a 32-bit Linux process
64-bit Linux on a x86_64 architecture

- code
- data
- heap
- stack
- kernel

0x00... 0x00007ff.. 0xffff800.. 0xff...
64-bit Linux on a x86_64 architecture

- Code
- Data
- Heap
- Stack (→)
- Not used
- Kernel

Memory addresses:
- Code: 0x00...
- Data: 0x00007ff..
- Heap: 0xffff800..
- Stack: 0xff..
- Kernel: 0x00000000
Every process has an address space from zero to some maximal address. A program contains instructions that of course rely on that code and data can be found at expected addresses. We only have one physical memory. Let's start from the beginning.
Memory virtualization

Every process has an address space from zero to some maximal address.
Memory virtualization

Every process has an address space from zero to some maximal address.

A program contains instructions that of course rely on that code and data can be found at expected addresses.
Every process has an address space from zero to some maximal address.

A program contains instructions that of course rely on that code and data can be found at expected addresses.

We only have one physical memory.
Memory virtualization

Every process has an address space from zero to some maximal address.

A program contains instructions that of course rely on that code and data can be found at expected addresses.

We only have one physical memory.
Every process has an address space from zero to some maximal address.

A program contains instructions that of course rely on that code and data can be found at expected addresses.

We only have one physical memory.
IBM System 360

- 1964, 8-64 Kbyte memory
- 12+12 bit address space
- Batch operating system

*Chief architect: Gene Amdahl*
when things were simple

Batch processing:

0x0000
0xffff
when things were simple

Batch processing:

\[ 0x0000 \]

\[ 0xffffffff \]
when things were simple

Batch processing:

0x0000

0xffffff
when things were simple

Batch processing:

0x0000

0xffff
when things were simple

**Batch processing:**

0x0000

0xffff
when things were simple

Batch processing:

0x0000

0xffffffff
when things were simple

Batch processing:

0x0000

0xffff
when things were simple

Batch processing:

0x0000

0xffff
when things were simple

Batch processing:

0x0000

0xffff
Arnold Spielberg was in the team that designed the GE-235
Time-sharing:

0x0000

0xffff
Time-sharing:

0x0000

0xffffffff
Time-sharing:
Time-sharing:
Time-sharing:

0x0000

0xffffff
Time-sharing:

0x0000

0xffff
Time-sharing:

0x0000

0x0000

0xffffffff
Time-sharing:

0x0000

0xffff
Time-sharing:

0x0000

0xffff
Time-sharing:

0x0000

0xffff
Time-sharing:

0x0000

0xffff
why not switch between two programs

If both programs will fit in memory:

0x0000

0xffff
why not switch between two programs

If both programs will fit in memory:

0x0000

0xffffffff
why not switch between two programs

If both programs will fit in memory:

0x0000

0xffff
why not switch between two programs

If both programs will fit in memory:

0x0000

0xffffffff
why not switch between two programs

If both programs will fit in memory:

0x0000

0xffff
why not switch between two programs

If both programs will fit in memory:

```
0x0000
```

```
0xffff
```
why not switch between two programs

If both programs will fit in memory:

What is the problem?
Virtual memory

Physical memory

0x0000

0xffff
Virtual memory

Physical memory

0x0000

0xffff
Virtual memory

Physical memory

0x0000

0xffffffff
Virtual memory

Physical memory

Process view

0x0000

0xffffffff
Virtual memory

Physical memory

Process view

0x0000

0x7fff

0xffff

Transparent: processes should be unaware of virtualization.

Protection: processes should not be able to interfere with each other.

Efficiency: execution should be as close to real execution as possible.
Virtual memory

Virtual memory is divided into two main parts:

- **Physical memory**
  - Ranges from 0x0000 to 0xffffffff

- **Process view**
  - Ranges from 0x0000 to 0x7fff

The diagram illustrates the relationship between physical memory and process view, showing how processes access memory within their specified ranges. The transparency ensures processes are unaware of virtualization, protection ensures they cannot interfere with each other, and efficiency aims for execution as close to real execution as possible.
Virtual memory

- **Physical memory**
  - 0x0000
  - 0xffff

- **Process view**
  - 0x0000
  - 0x7fff
  - 0x0000
  - 0x4fff

- **Transparent**: processes should be unaware of virtualization.

- **Protection**: processes should not be able to interfere with each other.

- **Efficiency**: execution should be as close to real execution as possible.
- Transparent: processes should be unaware of virtualization.
- Protection: processes should not be able to interfere with each other.
Virtual memory

- **Transparent**: processes should be unaware of virtualization.
- **Protection**: processes should not be able to interfere with each other.
- **Efficiency**: execution should be as close to real execution as possible.
Let the operating system run an emulator that interprets the operations of the process and changes the memory addresses as needed.
Emulator - simple but slow

Let the operating system run an emulator that interprets the operations of the process and changes the memory addresses as needed.

This is similar to how the JVM works.
When a program is loaded, all references to memory locations are changed so that they correspond to the actual location in RAM where the program is loaded.
Static relocation - ehh, static

Physical memory

0x0000
0x1000
0x2000
0x3000
0x4000
0x5000
0x5fff

Process view

0x0000
0x200
0x1fff

When a program is loaded, all references to memory locations are changed so that they correspond to the actual location in RAM where the program is loaded.
When a program is loaded, all references to memory locations are changed so that they correspond to the actual location in RAM where the program is loaded.
When a program is loaded, all references to memory locations are changed so that they correspond to the actual location in RAM where the program is loaded.
When a program is loaded, all references to memory locations are changed so that they correspond to the actual location in RAM where the program is loaded.
When a program is loaded, all references to memory locations are changed so that they correspond to the actual location in RAM where the program is loaded.
When a program is loaded, all references to memory locations are changed so that they correspond to the actual location in RAM where the program is loaded.

How do we know we have changed all addresses?
Dynamic relocation

Change every memory reference, on the fly, to a region in memory allocated for the process.
Dynamic relocation

Change every memory reference, on the fly, to a region in memory allocated for the process.
Dynamic relocation

Change every memory reference, on the fly, to a region in memory allocated for the process.
Dynamic relocation

Change every memory reference, on the fly, to a region in memory allocated for the process.
Base register

MMU

virtual addr.

base
Base register

MMU

virtual addr. → (+) → base → physical address
Base problem

Who is allowed to change the base register?

How do we prevent one process from overwriting another process?
Who is allowed to change the base register?
Base problem

- Who is allowed to change the base register?
- How do we prevent one process from overwriting another process?
Base problem

- Who is allowed to change the base register?
- How do we prevent one process from overwriting another process?
Who is allowed to change the base register?

How do we prevent one process from overwriting another process?
Base problem

- Who is allowed to change the base register?
- How do we prevent one process from overwriting another process?

Can we prevent this at compile or load time?
Base and bound

MMU

virtual addr.____________
Base and bound

MMU

virtual addr.  

base
Base and bound

MMU

virtual addr. → base + → physical address
Base and bound

MMU

virtual addr.  \rightarrow  bound

\rightarrow  physical address

base
Base and bound

MMU

virtual addr. \(\rightarrow\) \(\leq\) \(\rightarrow\) bound \(\rightarrow\) physical address

base
Base and bound

MMU

virtual addr. → < → yes → within bounds

bound

+ → base

physical address
Base and bound

MMU

virtual addr. → < bound → yes within bounds

+ base

exception

no
Pros:

- Transparent to a process.
- Simple to implement.
- Easy to change process.
Pros:
- Transparent to a process.
- Simple to implement.
- Easy to change process.

Cons:
- How do we share data?
- Wasted memory.
How do we write code that can be shared?
shared read-only segments

Physical memory

Process A
shared read-only segments

Physical memory

Process A
How do we write code that can be shared?
How do we write code that can be shared?
How do we write code that can be shared?
How do we write code that can be shared?
shared read-only segments

Physical memory

Process A

Process B

How do we write code that can be shared?
How do we write code that can be shared?
How do we write code that can be shared?
Internal fragmentation

Physical memory
Internal fragmentation

Physical memory

Process view
Internal fragmentation
Internal fragmentation

Physical memory

Process view
Internal fragmentation

Physical memory

Process view

unused
Internal fragmentation

Physical memory

Process view

unused
Internal fragmentation

Physical memory

Process view

wasted

unused
Burroughs B5000

1961

Designed for high-level languages: ALGOL-60
Memory access through a set of segment descriptors, i.e. the view of a process is not a consecutive memory rather a set of individual memory segments.

Donald Knuth was part of the design team.
Burroughs B5000

- 1961
- Designed for high-level languages: ALGOL-60
Burroughs B5000

- 1961
- Designed for high-level languages: ALGOL-60
- Memory access through a set of segment *descriptors* i.e. the view of a process is not a consecutive memory rather a set of individual memory segments.

Donald Knuth was part of the design team.
Burroughs B5000

- 1961
- Designed for high-level languages: ALGOL-60
- Memory access through a set of segment *descriptors* i.e. the view of a process is not a consecutive memory rather a set of individual memory segments.

*Donald Knuth was part of the design team.*
procedure Absmax(a) Size:(n, m) Result:(y) Subscripts:(i, k);
    value n, m; array a; integer n, m, i, k; real y;

comment The absolute greatest element of the matrix a ...

begin
    integer p, q;
    y := 0; i := k := 1;
    for p := 1 step 1 until n do
        for q := 1 step 1 until m do
            if abs(a[p, q]) > y then
                begin y := abs(a[p, q]);
                    i := p; k := q
                end
    end Absmax
The view of the assembler programmer.

The view of the ALGOL programmer.

procedures

data
Segmented architecture

Physical memory
Segmented architecture

Physical memory

Process A
Segmented architecture

Physical memory

Process A
Segmented architecture

Physical memory

Process A
Segmented architecture

Physical memory

Process A
Segmented architecture

Physical memory

Process A
Segmented architecture

Physical memory

Process A

Process B
Segmented architecture

Physical memory

Process A

Process B
Segmented architecture

Physical memory

Process A

Process B
Segmented architecture

Physical memory

Process A

Process B
Segmented MMU

MMU

virtual addr. -> < base

bound

physical address

exception

no

yes within bounds
Segmented MMU

- Virtual address
- Offset
- Index
- Segment table
- Base
- Physical address
- Exception
- Bound
- Within bounds

Virtual address + Offset = Base

If within bounds, yes.

If not within bounds, exception.
Segmented MMU

MMU

virtual addr. // offset -> exception
index

++

within bounds

physical address

segment table

no

yes
PDP-10

- 1966, 1 MHz
- 36 bit words
- 16 bit process address space (64Kword)
- 18 bit physical address (256 Kword)
- base and bound

*The PDP10 had two segments per process, one read only code segment and one read/write for data.*
ARPANET LOGICAL MAP, MARCH 1977

[Diagram showing a network map of ARPANET with various nodes and connections.]

[Please note that while this map shows the host population of the network according to the best information obtainable, no claim can be made for its accuracy. Names shown are IMP names, not necessarily host names.]

ARPANET 1977
• Segments have variable size.
- Segments have variable size.
Segmentation: the solution

- Segments have variable size.

Reclaiming segments will cause holes (external fragmentation). Compaction needed. Is it possible to do compaction?
Segments have variable size.
Segments have variable size.
Segmentation: the solution

- Segments have variable size.
Segments have variable size.
Segments have variable size.
Segments have variable size.
Segments have variable size.

Reclaiming segments will cause holes (external fragmentation).
Segmentation: the solution

- Segments have variable size.
- Reclaiming segments will cause holes (external fragmentation).
Segments have variable size.

Reclaiming segments will cause holes (external fragmentation).

Compaction needed.
Segmentation: the solution - not

- Segments have variable size.
- Reclaiming segments will cause holes (external fragmentation).
- Compaction needed.

*Is it possible to do compaction?*
large grain vs fine grain segments

Using few large segments is easier to implement. Using many small segments would allow the compiler and operating system to do a better job.
Using few large segments is easier to implement.
large grain vs fine grain segments

Using few large segments is easier to implement.

Using many small segments would allow the compiler and operating system to do a better job.
The Altair 8800

Intel 8080

- 1972
- 2 MHz
- 16 bit address space (64 Kbyte)

*Altair 8800 would have 4 or 8 Kbytes of memory.*
The workhorse: 8086

Intel 8086

- 1978, 5 MHz
- 16 bit address space (64 Kbyte)
- 20 bit memory bus (1 Mbyte)
- no protection of segments
- segments for: code, data, stack, extra
Segment addressing in 8086 - real mode

Segment register chosen based on instruction:
- code segment
- stack segment
- data segment (and the extra segment).

The segment architecture available still today in real mode, i.e., the 16-bit mode that the CPU is initially in.
Segment addressing in 8086 - real mode

- Segment register chosen based on instruction: *code segment*, *stack segment*, *data segment* (and the extra segment).

- Segment 16 bits
- Offset 16 bits
- Bus 20 bits
Segment addressing in 8086 - real mode

- Segment register chosen based on instruction: *code segment*, *stack segment*, *data segment* (and the extra segment).
- The segment architecture available still today in *real mode* i.e. the 16-bit mode that the CPU is initially in.
Segment addressing in 80386 - protected mode

MMU

virtual addr.  offset

<

+

33 / 35
Segment addressing in 80386 - protected mode

MMU

virtual addr. offset

bound

base
Segment addressing in 80386 - protected mode

- MMU
- virtual addr.
- offset
- base
- linear address
- bound
- exception
- no
- ok

The diagram illustrates the process of segment addressing in 80386 protected mode, showing how virtual addresses are translated to linear addresses through segment registers.
Segment addressing in 80386 - protected mode

MMU

virtual addr.  offset

<

bound

linear address

base

no

exception

ok

gdtr
Segment addressing in 80386 - protected mode

MMU

virtual addr. → offset → bound

base + bound → linear address

gdtr

Global Descriptor Table (GDT)

ok

exception → no → ok
Segment addressing in 80386 - protected mode

Global Descriptor Table (GDT)

MMU

virtual addr.          offset

exception

no

<

bound

descriptor

base

linear address

ok

gdtr
Segment addressing in 80386 - protected mode

MMU

virtual addr.  offset

segment selectors

descriptor

base

bound

linear address

gdtr

Global Descriptor Table (GDT)

exception

no

ok
Segment addressing in 80386 - protected mode

MMU

virtual addr. offset -> < -> ok

segment selectors

code

descriptor

bound

linear address

gdtr

Global Descriptor Table (GDT)
Segment addressing in 80386 - protected mode

MMU

virtual addr.  offset

segment selectors
- code
- stack

gdtr

Global Descriptor Table (GDT)

bound

linear address

base

descriptor

<

ok

no

exception
Segment addressing in 80386 - protected mode

MMU

virtual addr.  offset  exception

no  ok

<

bound

deSCRIPTOR

base

linear address

gdtr

global descriptor table (GDT)

segment selectors

code

stack

data
Segment addressing in 80386 - protected mode

MMU

virtual addr.  offset

exception

no

ok

segment selectors

code
stack
data

descriptor

base

bound

linear address

gdtr

Global Descriptor Table (GDT)
The segments descriptors of code, data and stack all have base address set to 0x0 and limit to 0xffffffff i.e. they all referre to the same 4 Gbyte linear address space.
The segments descriptors of code, data and stack all have base address set to 0x0 and limit to 0xffffffff i.e. they all referre to the same 4 Gibyte linear address space.

In x86_64 long mode (64 bit mode) Intel removed some support for segments and enforce that these segments are set to 0x0 and 0xff..ff.
The segments descriptors of code, data and stack all have base address set to 0x0 and limit to 0xffffffff i.e. they all refer to the same 4 Gabyte linear address space.

In x86_64 long mode (64 bit mode) Intel removed some support for segments and enforce that these segments are set to 0x0 and 0xff..ff.

Segmentation is still used to refer to memory that belongs to a specific core or to thread specific memory.
Virtual address space: provide a process with a view of a private address space.
Virtual address space: provide a process with a view of a private address space.

- Transparent: processes should be unaware of virtualization.
Virtual address space: provide a process with a view of a private address space.

- Transparent: processes should be unaware of virtualization.
- Protection: processes should not be able to interfere with each other.
Virtual address space: provide a process with a view of a private address space.

- Transparent: processes should be unaware of virtualization.
- Protection: processes should not be able to interfere with each other.
- Efficiency: execution should be as close to real execution as possible.
Virtual address space: provide a process with a view of a private address space.

- **Transparent**: processes should be unaware of virtualization.
- **Protection**: processes should not be able to interfere with each other.
- **Efficiency**: execution should be as close to real execution as possible.

- **Emulator**: two slow.
Summary

Virtual address space: provide a process with a view of a private address space.

- **Transparent**: processes should be unaware of virtualization.
- **Protection**: processes should not be able to interfere with each other.
- **Efficiency**: execution should be as close to real execution as possible.

- **Emulator**: two slow.
- **Static relocation**: not flexible.

Dynamic relocation:
- **Base and bound**: simple to implement
- **Segmentation**: more flexible

Cliffhanger - paging, the solution.
Summary

Virtual address space: provide a process with a view of a private address space.

- **Transparent:** processes should be unaware of virtualization.
- **Protection:** processes should not be able to interfere with each other.
- **Efficiency:** execution should be as close to real execution as possible.

- **Emulator - two slow.**
- **Static relocation - not flexible.**
- **Dynamic relocation:**
Virtual address space: provide a process with a view of a private address space.

- Transparent: processes should be unaware of virtualization.
- Protection: processes should not be able to interfere with each other.
- Efficiency: execution should be as close to real execution as possible.

- Emulator - two slow.
- Static relocation - not flexible.
- Dynamic relocation:
  - base and bound - simple to implement
Summary

Virtual address space: provide a process with a view of a private address space.

- Transparent: processes should be unaware of virtualization.
- Protection: processes should not be able to interfere with each other.
- Efficiency: execution should be as close to real execution as possible.

- Emulator - two slow.
- Static relocation - not flexible.
- Dynamic relocation:
  - base and bound - simple to implement
  - segmentation - more flexible
Summary

Virtual address space: provide a process with a view of a private address space.

- Transparent: processes should be unaware of virtualization.
- Protection: processes should not be able to interfere with each other.
- Efficiency: execution should be as close to real execution as possible.

- Emulator - two slow.
- Static relocation - not flexible.
- Dynamic relocation:
  - base and bound - simple to implement
  - segmentation - more flexible
  - problems: fragmentation, sharing of code
Summary

Virtual address space: provide a process with a view of a private address space.

- Transparent: processes should be unaware of virtualization.
- Protection: processes should not be able to interfere with each other.
- Efficiency: execution should be as close to real execution as possible.

- Emulator - two slow.
- Static relocation - not flexible.
- Dynamic relocation:
  - base and bound - simple to implement
  - segmentation - more flexible
  - problems: fragmentation, sharing of code

*Cliffhanger - paging, the solution.*