Segmentation
Objective

• Explain the x86 segmentation model
• Explain how a virtual address is translated by the x86 to a physical address
• Explain the various x86 datastructures and hardware registers
Loading of Programs (1)

• Every program is linked by the linker to be a self-contained executable.
• Programs are linked as if they were loaded to address 0. I.e., all references (jumps, calls, memory loads) are done relative to address 0.
• When a program is loaded into memory, it will be assigned an available region of memory.
• All references need to be adjusted to the base address of this memory region (see next slide).
• Beware: some references (such as 0xB8000) should not be adjusted!
• Note: adjusting references does not guard against one process corrupting another process’ memory!
Loading of Programs (2)

**Program 1**

20  call 10
10  ...
0  jump 20

18  ...
10  movl 18,%eax
0  jump 10

**Program 2**

20  jump 40
18  ...
10  movl 18,%eax
0  jump 10

**Main Memory**

60

40  call 30
30  ...
20  jump 40
18  ...
10  movl 18,%eax
0  jump 10
Address Translation

• “Patching” a program as shown on the previous slide is a complicated matter and also does not protect processes amongst each other.
• A better way would be to have hardware support that does not require this patching.
• This is the goal of address translation: instead of patching a program manually, (virtual) addresses are translated ‘on the fly’.
• Physical memory in a computer is a linear array of bytes called physical address space.
• Physical memory is not directly accessed by programs. They use virtual memory which maps to physical memory.
• Chief benefit of virtual memory is protection as each program (process) gets its own address space. A program can not access address space of another program.
Segmentation and Paging - I

• Segmentation and paging are two address translation techniques supported by the x86.
• Segmentation is a “two dimensional” virtual address space defined by selector:offset.
• Each selector value points to a segment and the offset value points to the offset within that segment
  – Selector: 16 bits
  – Offset: 32 bits
• Segmentation maps virtual addresses to linear addresses.
• All x86 programs have to go through segmentation.
Segmentation and Paging - II

- Linear address space is 32 bit address space for each task
- Allows up to a maximum of 4 GB (\(= 2^{32}\) bytes)
- Paging maps linear address space to physical address space
- Paging can be enabled or disabled (we will not use paging!)
- If paging is disabled, then the linear address corresponds to the physical address
- Both segmentation and paging use translation tables
Segmentation and Paging - III

Selector : Offset

Virtual address

0....15 0.........31

Segmentation Translation Table

Linear address

0.........31

Paging Translation Table

(Paging disabled)

Physical address

0.........31

Segmentation

Paging
Segmentation - Mapping

- Variable size units of memory called segments form the basis of the virtual-to-linear address translation.
- Segments are defined by:
  - base address
  - address limit
  - segment attributes
- Addresses within one segment are relative to the base address of that segment.
- Idea: load each program into its own segment!
Segmentation - Example

• Segment 1 is defined as follows:
  – Base address: 0x50000
  – Limit: 0xffffffff

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Linear address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:0</td>
<td>0x500000</td>
</tr>
<tr>
<td>1:0x67</td>
<td>0x50067</td>
</tr>
<tr>
<td>1:0x10000</td>
<td>Illegal. Beyond limit of segment</td>
</tr>
</tbody>
</table>
Segment Descriptors

- The x86 contains six segment registers: %CS, %DS, %ES, %FS, %GS, and %SS
- A segment register contains a 16 bit segment selector
- Segment selector is an index to the segment descriptor stored in a descriptor table
- Each segment descriptor contains important information about the segment: base, limit, attributes
- The 4 bit type attribute specifies if the segment is read only/read write/execute or other specific combinations thereof
Segment Descriptor Tables

- Segment descriptor tables hold the segment descriptors
- There are two types local (LDT) and global (GDT). The virtual address space is distributed between these two tables
- GDT remains the same for all processes, while the LDT is private to each process
- GDT can be used to store information about OS code and data segments accessible to all programs. LDT can be used to store code and data segments for that program
- GDT also stores other types of descriptors (not relevant to the topic here)
- In TOS we only use the GDT
Anatomy of a Segment Selector and Descriptor

Segment Selector

- Descriptor Index (13 bits)
- T
- I
- RPL

GDT or LDT

Segment Selector

- Segment Base
  - (31...24)
- (Other attributes)
- Type
  - (0...3)
- Segment Base
  - (23.................0)
- Segment Limit
  - (15...........0)

Used by protection mechanism

GDT or LDT

- .
- .
- .
- .
- .
- .
Notes

• Size of one Segment Descriptor is 8 bytes
• Note that the 32 bits of the segment base are not stored consecutively in the Segment Descriptor
• RPL (Requested Privilege Level). Should always be set to 0 in TOS
• TI (Table Index). Should always be set to 0 in TOS
• The only two segment types we will use in TOS are:
  – Data: 0x2
  – Code: 0xa
• GDT and LDT are tables which are stored in ordinary memory
LGDT instruction

• Load GDT instruction (lgdt) loads the 48 bit GDTR register with the base address and limit of the GDT.
• This instruction is used in TOS’ boot loader
Segmentation and X86 Assembly

• How are segments used in day-to-day assembly?
• Each kind of memory access uses a certain default segment register
  – %CS is used for fetching from memory (e.g. CALL, JMP, RET)
  – %SS is used for all memory access to the stack (e.g. PUSH, POP)
  – %DS is used for all other memory access (e.g. MOV)
• Examples:
  – PUSH $1 access %SS:%ESP
  – MOV $1, (%EAX) access %DS:(%EAX)
  – JMP $100 access %CS:$100
• The content of segment registers are typically loaded only once at boot-time and then never changed again
• Segment registers can be loaded from normal x86 register, e.g.
  movw $0x10, %AX
  movw %AX, %DS
loads the %DS segment register with 16
GDT for TOS (1)

- Basic idea: segments in TOS are defined such that a virtual address is identical to the physical address.
- The GDT for TOS is constructed and loaded during the boot process; i.e., before calling `kernel_main()`.
- Size of the GDT is 24 bytes (== 3 * 8 bytes).
- %CS is loaded with 0x8 (GDT entry 1) and %DS and %SS are loaded with 0x10 (GDT entry 2).

<table>
<thead>
<tr>
<th>Entry</th>
<th>Base</th>
<th>Limit</th>
<th>Attributes</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Dummy Entry</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0xFFFFFF</td>
<td>CODE</td>
<td>Used for %CS</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0xFFFFFF</td>
<td>DATA</td>
<td>Used for %DS and %SS</td>
</tr>
</tbody>
</table>
GDT for TOS (2)

• TOS uses two segment selectors:
  CODE: \( 8_{16} = 10002 \)
  DATA: \( 10_{16} = 100002 \)

\[
\begin{array}{c}
10002 = 0000000000001000 \\
100002 = 0000000000010000 \\
\end{array}
\]

Index

\[
\begin{array}{c}
10002 \\
100002 \\
\end{array}
\]

\[
\begin{array}{c}
0000000000000001000 \\
0000000000000001000 \\
\end{array}
\]

TI/RPL

\[
\begin{array}{c}
12 = 1_{10} \\
102 = 2_{10} \\
\end{array}
\]

GDT entry 1

GDT entry 2
Memory Access in TOS

• Given the following code:
  movl $0xB8000, %EAX
  movb $’A’, (%EAX)

• ‘A’ is stored to 0xB8000. But how does it look exactly with segmentation?

• Since we do a regular memory access, the %DS segment is used. The linear address is therefore 0x10: 0xB8000
  – Segment 0x10
  – Offset 0xB8000

• Base address of the segment 0x10 is 0
• Therefore the linear address is 0xB8000 + 0 = 0xB8000
• Since we don’t use paging, linear address corresponds to physical address
• Therefore, we access physical address 0xB8000
Inter-Segment Subroutines (1)

- So far, when doing a `CALL` instruction, we only specified the 32-bit offset, but not the segment sector.
- The segment was implicitly selected through the `%CS` segment register.
- This is called an *Intra-Segment Jump* because the jump happens within the same segment.
- An *Inter-Segment Jump* jumps between different segments.
- Inter-Segment Jumps will happen in TOS only for interrupts.
- For an *Inter-Segment Subroutine* call, not only the return address is pushed on the stack, but also `%CS` (see next slide).
Inter-Segment Subroutines (2)

• Assumptions:
  - %CS = 0x8
  - Return address is 0xABCD1234

• When executing `CALL 0xC : 0x12123434` the following information is pushed onto the stack:
  - 0x00000008 (old value of %CS as a 32-bit value)
  - 0xABCD1234 (return address)

• For an intra-segment subroutine call only the offset part 0xABCD1234 would have been pushed on the stack

• After pushing the return address on the stack, the registers are loaded as follows:
  - %CS := 0xC (this goes through the GDT!)
  - %EIP := 0x12123434