Two programs one memory

```c
main() {
    ...
    yield();
}
```

```c
main() {
    ...
    yield();
}
```

Save/restore
Very much like car sharing
What are we aiming for?

- Illusion of a private address space
  - Identical copy of an address space in multiple programs
    - Remember `fork()`?
  
- Simplifies software architecture
  - One program is not restricted by the memory layout of the others
Two processes, one memory?
Two processes, one memory?

Process 1 (ls)  
\[\text{base}_{P_1}\]
\[x\]
\[x + \text{base}_{P_1}\]

Process 2 (ls)  
\[\text{base}_{P_2}\]
\[x\]
\[x + \text{base}_{P_2}\]
This is called segmentation
All addresses are logical address

- They consist of two parts
  - Segment selector (16 bit) + offset (32 bit)
- Segment selector (16 bit)
  - Is simply an index into an array (Descriptor Table)
  - That holds segment descriptors
    - Base and limit (size) for each segment
Elements of that array are segment descriptors

- Base address
  - 0 – 4 GB
- Limit (size)
  - 0 – 4 GB
- Access rights
  - Executable, readable, writable
  - Privilege level (0 - 3)
## Segment descriptors

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base 31:24</td>
<td>Segment base address</td>
</tr>
<tr>
<td>G</td>
<td>Granularity</td>
</tr>
<tr>
<td>D/B</td>
<td>Default operation size (0 = 16-bit segment; 1 = 32-bit segment)</td>
</tr>
<tr>
<td>AVL</td>
<td>Available for use by system software</td>
</tr>
<tr>
<td>Seg. Limit 19:16</td>
<td>Segment limit</td>
</tr>
<tr>
<td>P</td>
<td>Segment present</td>
</tr>
<tr>
<td>DPL</td>
<td>Descriptor privilege level</td>
</tr>
<tr>
<td>S</td>
<td>Descriptor type (0 = system; 1 = code or data)</td>
</tr>
<tr>
<td>Type</td>
<td>Segment type</td>
</tr>
<tr>
<td>Base 23:16</td>
<td>Segment base address</td>
</tr>
<tr>
<td>Base Address 15:00</td>
<td>Segment base address</td>
</tr>
<tr>
<td>Segment Limit 15:00</td>
<td>Segment limit</td>
</tr>
</tbody>
</table>

L — 64-bit code segment (IA-32e mode only)
• Offsets into segments (x in our example) or “Effective addresses” are in registers
- Logical addresses are translated into physical
- *Effective address + DescriptorTable[selector].Base*
Logical addresses are translated into physical addresses:
- Effective address + DescriptorTable[selector].Base
• Logical addresses are translated into physical
• Effective address + DescriptorTable[selector].Base
• Logical addresses are translated into physical
  • Effective address + DescriptorTable[selector].Base
Same picture
- Offsets (effective addresses) are in registers
  - *Effective address + DescriptorTable[selector].Base*
- But where is the selector?
Right! Segment registers

- Hold 16 bit segment selectors
  - Pointers into a special table
  - Global or local descriptor table
- Segments are associated with one of three types of storage
  - Code
  - Data
  - Stack
Programming model

• Segments for: code, data, stack, “extra”
  • A program can have up to 6 total segments
  • Segments identified by registers: cs, ds, ss, es, fs, gs

• Prefix all memory accesses with desired segment:
  • `mov eax, ds:0x80` (load offset 0x80 from data into eax)
  • `jmp cs:0xab8` (jump execution to code offset 0xab8)
  • `mov ss:0x40, ecx` (move ecx to stack offset 0x40)
Segmented programming (not real)

static int x = 1;
int y; // stack
if (x) {
    y = 1;
    printf ("Boo");
} else
    y = 0;

ds:x = 1; // data
ss:y;     // stack
if (ds:x) {
    ss:y = 1;
    cs:printf(ds:"Boo");
} else
    ss:y = 0;
Programming model, cont.

- This is cumbersome, so infer code, data and stack segments by instruction type:
  - Control-flow instructions use code segment (jump, call)
  - Stack management (push/pop) uses stack
  - Most loads/stores use data segment
- Extra segments (es, fs, gs) must be used explicitly
Code segment

- Code
  - CS register
  - EIP is an offset inside the segment stored in CS
- Can only be changed with
  - procedure calls,
  - interrupt handling, or
  - task switching
Data segment

- Data
  - DS, ES, FS, GS
  - 4 possible data segments can be used at the same time
Stack segment

- Stack
  - SS
- Can be loaded explicitly
  - OS can set up multiple stacks
  - Of course, only one is accessible at a time
Segmentation is ok... but
What if process needs more memory?

Process 1 (ls)
- `malloc() = x`
- `base_{p_1}`

Process 2 (ls)
- `x`
- `base_{p_2}`

Memory
- `x + base_{p_1}`
- `x + base_{p_2}`
What if process needs more memory?

![Diagram showing memory allocation and process interactions](image-url)
You can relocate P2
Or even swap it out to disk

```
Process 1 (ls)

malloc() =

base_{p_1}

 Process 2 (ls)

x

base_{p_2}

```

Memory

```
x + base_{p_1}
```

```
x + base_{p_2}
```

Or even swap it out (move to disk)
Problems with segments

• But it's inefficient
  • Relocating or swapping the entire process takes time
• Memory gets fragmented
  • There might be no space (gap) for the swapped out process to come in
  • Will have to swap out other processes
Paging
Paging idea

- Break up memory into 4096-byte chunks called pages
  - Modern hardware supports 2MB, 4MB, and 1GB pages
- Independently control mapping for each page of linear address space

- Compare with segmentation (single base + limit)
  - many more degrees of freedom
Page translation

- Linear Address
  - Directory
  - Table
  - Offset

- Page Directory
- PDE with PS=0
- CR3

- Page Table
- PTE

- Physical Address
- 4-KByte Page
**Page directory entry (PDE)**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address of page table</td>
<td>Ignored</td>
<td>0</td>
<td>Ign</td>
<td>A</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>P</td>
<td>W</td>
<td>U</td>
<td>S</td>
<td>R</td>
<td>W</td>
<td>1</td>
<td>PDE: page table</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 20 bit address of the page table
  - Pages 4KB each, we need 1M to cover 4GB
- R/W – writes allowed?
  - To a 4MB region controlled by this entry
- U/S – user/supervisor
  - If 0 – user-mode access is not allowed
- A – accessed
## Page table entry (PTE)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Address of 4KB page frame</th>
<th>Ignored</th>
<th>G</th>
<th>P</th>
<th>A</th>
<th>T</th>
<th>D</th>
<th>A</th>
<th>P</th>
<th>C</th>
<th>D</th>
<th>PW</th>
<th>U/S</th>
<th>R/W</th>
<th>1</th>
<th>PTE: 4KB page</th>
</tr>
</thead>
</table>

- 20 bit address of the 4KB page
  - Pages 4KB each, we need 1M to cover 4GB
- R/W – writes allowed?
  - To a 4KB page
- U/S – user/supervisor
  - If 0 user-mode access is not allowed
- A – accessed
- D – dirty – software has written to this page
Page translation

- Linear Address
  - Directory
  - Table
  - Offset

- 4-KByte Page
- Physical Address

- Page Directory
  - PDE with PS=0
- CR3

- Page Table
  - PTE
If a page is 4K and an entry is 4 bytes, how many entries per page?

- 1k

How large of an address space can 1 page represent?

- 1k entries * 1 page/entry * 4K/page = 4MB

How large can we get with a second level of translation?

- 1k tables/dir * 1k entries/table * 4k/page = 4 GB
- Nice that it works out that way!