Segmentation
Segmentation

- Partition virtual address space into *multiple disjoint segments*
  - Individually map onto physical memory with separate base/limit registers
    - Address space info stored in PCB and restored on context switch
  - Requires that memory requests are for *segmented addresses*
    - Consist of segment selector and offset into segment
    - Alternatively: segment can be implied by instruction (e.g., PC always refers to code segment)
E.g., logical segments

<table>
<thead>
<tr>
<th>Seg #</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$B_0$</td>
<td>$L_0$</td>
</tr>
<tr>
<td>1</td>
<td>$B_1$</td>
<td>$L_1$</td>
</tr>
<tr>
<td>2</td>
<td>$B_2$</td>
<td>$L_2$</td>
</tr>
<tr>
<td>3</td>
<td>$B_3$</td>
<td>$L_3$</td>
</tr>
</tbody>
</table>
E.g., translation

$$\text{MMU}$$

<table>
<thead>
<tr>
<th>Segment regs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Base</strong></td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

**CPU**

VA: `seg#:offset`

PA: `offset + B_2`

Physical

$B_0 + L_0$

$B_1 + L_1$

$B_2 + L_2$

$B_3 + L_3$

assert ($\text{offset} \leq L_2$)
Improved utilization

- Mapping individual segments avoids reserving memory for unused space between segments in the linear address space

![Diagram showing simple relocation and segmentation](https://via.placeholder.com/150)
Segment sharing and metadata

- Segments may be shared between processes to reduce memory usage (and improve caching behavior)
- Segments may have additional metadata to control and limit access
  - Read-only / Non-executable segments
  - Privilege-level based access control (kernel vs. user)
  - Direction of growth (e.g., downward from max offset for stacks)
E.g., shared code and metadata

R= readable, W= writable, X= executable, E= expand downwards
Segmentation fault

- A segmentation fault can be generated by the MMU when:
  - Limit check fails (access beyond ends of segment)
  - Access control assertion fails (e.g., illegal operation)
  - Privilege assertion fails (e.g., insufficient privilege)
  - Fault transfers control to kernel (to alert/terminate offending process)
Downside: external fragmentation

- Variable segment sizes make placement and free space search non-trivial
- Memory may be defragmented via compaction, but this is expensive!
- Also, large segments still loaded monolithically (coarse-grained mapping)
Analysis

- **Fast translation** via base register + offset
- **Protection** enforced via limits
- **Improved memory utilization** over monolithic mapping
- **Access control and sharing** via additional segment metadata
- But variable, monolithic segments create **external fragmentation**, making free space search and/or compaction necessary
Paging
Addressing segmentation issues

- Variable segment sizes cause external fragmentation
  - Instead, partition address space and main memory into fixed-size pages
- Large, monolithic segments may reduce utilization, as only a fraction of a segment may be needed at a given time
  - Instead, reduce the granularity of mapping with smaller pages
Paging

- Partition virtual address spaces and memory into uniformly sized pages
- Granularity of mapping = page
- Segments may span (and are not necessarily aligned to) pages
- Not all of a segment needs to be mapped
Modified mapping problem

- A virtual address is broken down into a virtual page number and offset

- Mapping problem: virtual page number (VPN) $\rightarrow$ physical page number (PPN) (latter aka physical frame number (PFN))

- e.g., given page size $= 2^p$ bytes

\[
\begin{align*}
\text{VA:} & \quad p \\
\text{VPN} & \quad \text{offset} \\
\text{PA:} & \quad p \\
\text{PPN} & \quad \text{offset}
\end{align*}
\]
Modified mapping problem

- Issue: how to store mappings?
- I.e., what data structure to use for representing virtual address spaces?

VA: \[
\begin{array}{c|c}
VPN & \text{offset} \\
\end{array}
\]

page mapping structure

PA: \[
\begin{array}{c|c}
PPN & \text{offset} \\
\end{array}
\]
Page table

- Typical implementation:
  - Table with a separate entry per virtual page, each indicating:
    - If mapping exists (valid flag)
    - Access metadata, e.g., rwx and kernel/user flags
    - The corresponding **physical page number**
Page table

- VPN from virtual address acts as an index into the page table
- PPN (if valid and mapped) is concatenated with offset from VA to form PA
E.g., page table size

- Given:
  - 32-bit virtual addresses
  - 4KB ($2^{12}$) sized pages
  - 4-byte page table entries
- How large is the page table?
  - Number of pages $= 2^{32} \div 2^{12} = 2^{20} = 1M$
  - Page table size $= 1M \times 4$ bytes $= 4MB$
- Recall: each process needs its own page table!
Page table walk

- The page table is too large to fit into the MMU, so resides in memory

- Translating a VPN → PPN requires indexing into the page table (known as a *page table walk*)
  - Performed by MMU

- Page table is managed by the kernel for each process
  - Current process page table is selected by kernel on each context switch (e.g., by pointing a *page table base register* at it)
Page table translations are slow!

- Most modern caching systems are physically addressed, so we cannot avoid translation before cache lookup

- I.e., each VA access requires up to two memory accesses!
Translation Lookaside Buffer (TLB)

- Solution: dedicated cache for VPN → PPN translations
- Page table walk only performed on TLB miss
TLB / Cache / PT interaction

- MMU: Address Translator
- TLB (VPN→PPN cache)
- Cache
- Page Table
- Physical

- CPU
- VA
- PA
- hit
- update
- page table walk
- only if TLB miss!

Data flow:
- Ideal path (no memory accesses)

Arrow relationships and notes:
- MMU to TLB
- TLB to Cache
- Cache to Page Table
- Page Table to Physical
- Physical to MMU

Notes:
- Page Table walk only if TLB miss!
TLB issues

- TLB mappings are process specific — requires flush on context switch
  - Some architectures store address space identifier per cache line
- TLB only caches a few thousand mappings, at most
  - vs. orders of magnitude more per process, potentially!
- Effectiveness of TLB can be “tuned” by adjusting number of pages
  (larger page size = smaller number of pages)
  - Downside to large pages?
Internal fragmentation

- Large pages result in coarser mapping granularity
  - I.e., larger “chunks” carved out of physical memory at a time
    - May lower utilization, if large portions of pages are not used — known as internal fragmentation
- Must balance TLB effectiveness against memory utilization
E.g., large(-ish) pages
E.g., small(-ish) pages