Paging

- Paging - logical address space of a process can be noncontiguous; process is allocated physical memory wherever the latter is available.
  - Divide physical memory into fixed sized blocks called frames (size is power of 2, between 512 bytes and 8192 bytes).
  - Divide logical memory into blocks of same size called pages.
  - Keep track of all free frames.
  - To run a program of size $n$ pages, need to find $n$ free frames and load program.
  - Set up a Page Table to translate logical to physical addresses.
  - Internal fragmentation.

### Paging

- Address generated by CPU is divided into:
  - **Page number ($p$)** - used as an index into a Page Table which contains base address of each page in physical memory.
  - **Page offset ($d$)** - combined with base address to define the physical memory address that is sent to the memory unit.

![Diagram of paging](image)

**Paging Model of Logical and Physical Memory**

- Separation between user’s view of memory and actual physical memory reconciled by address translation hardware; logical addresses are translated into physical addresses.

![Diagram of paging model](image)

**Mapping is hidden from the user**
Paging Example

**Paging: Selection of Page Size**

- **Example:**
  - Logical address space: \(2^{31} \text{B} = 1024 \text{MB}\)
  - Swapping device: Access time: 20 ms
    Transfer rate: 2 MB/sec (2 KB/ms)
  - Page size: 512 B or 4096 B (4 KB)
  - Page table entry: 4 bytes
  - Main program size: 512 KB

<table>
<thead>
<tr>
<th>PAGE SIZE IN BYTES</th>
<th>INTERNAL FRAGMENTATION</th>
<th>TABLE FRAGMENTATION</th>
<th>EFFICIENCY OF PAGE-IN OR-OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>Small (0.05%)</td>
<td>Large (16 MB)</td>
<td>Poor (20.5 ms)</td>
</tr>
<tr>
<td>4096</td>
<td>Large (0.39%)</td>
<td>Small (2 MB)</td>
<td>Good (22.0 ms)</td>
</tr>
</tbody>
</table>

**Paging: Implementation of Page Table**

- Page table is kept in main memory.
- **Page-table base register (PTBR)** points to the page table.
- **Page-table length register (PTLR)** indicates size of page table.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative registers** or **translation look-aside buffers (TLBs).**

\[ P - \text{page number} \]
\[ D - \text{displacement (offset)} \]
\[ F - \text{Page frame number} \]

**Paging: Associative Memory (Content Addressable Memory)**

- A memory system organized in such a way that memory cells can be searched (in parallel) by the contents, not the address or index.
- Implemented with a high-speed logic such as registers; hence much more expensive than regular memory.
- Used to store and read a small amount of frequently used information.
Paging: Paging Hardware with TLB

Address translation (A', A'')
- If A' in associative registers, get Frame # out.
- Otherwise get Frame # from Page Table in Memory.

Hit ratio: percentage of times that a page number is found in the associative registers; ratio related to number of associative registers

Paging: Effective Access Time (EAT)

- Example:
  - TLB access time: \( T_B \) ns (~ 20 ns)
  - Memory access time: \( T_M \) ns (~ 100 ns)
  - Hit ratio: \( P_{\text{HIT}} \) (80% ~ 98%)
  - Effective Memory Access Time (EAT):
    \[
    T_E = P_{\text{HIT}} (T_B + T_M) + (1 - P_{\text{HIT}}) (T_B + T_M + T_M)
    = (T_B + T_M) + (1 - P_{\text{HIT}}) T_M
    \]
    \[\begin{align*}
P_{\text{miss}} = 80\% & \quad \text{EAT} = 0.80 \times 120 + 0.20 \times 220 = 140 \text{ ns (40\% slowdown)} \\
P_{\text{miss}} = 98\% & \quad \text{EAT} = 0.98 \times 120 + 0.02 \times 220 = 122 \text{ ns (22\% slowdown)}
    \end{align*}\]

Paging: Protection

- Memory protection implemented by associating protection bits with each frame.
- Valid - invalid bit attached to each entry in the Page Table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page.
  - “invalid” indicates that the page is not in the process’ logical address space.

Paging: Avoiding Table Fragmentation

- Each process has a Page Table which takes up a lot of memory.
  (Consider a system with a 32-bit logical address space. If the page size is 4K bytes \(2^{12}\), then a Page Table may consist of up to 1 M entries \(2^{22}/2^{12}\). Because each entry consists of 4 bytes \(= 2^{12}\), each process may need up to 4 Mbytes of physical address space for the Page Table alone)

- Solutions:
  - Multilevel Paging (divide Page Table into smaller pieces).
  - Inverted Page Table.
**Paging: Multilevel Paging**

- **Multilevel Paging** - partitioning the Page Table allows the operating system to leave partitions unused until a process needs them.
- A two-level page table scheme

**Paging: Two-level 32-bit Paging**

Three-level paging - SPARC (32-bit addressing)
Four-level paging - Motorola 68030 (32-bit addressing)

**Paging: Multilevel Paging and Performance**

- Since each level is stored as a separate table in memory, converting a logical address to a physical one may take four memory accesses.
- Even though time needed for one memory access is quintupled (five times as much), caching permits performance to remain reasonable.
- Cache hit ratio of 98% yields:

    \[
    \text{Effective Access Time} = 0.98 \times 120 + 0.02 \times 520 = 128 \text{ nanoseconds}
    \]

    which is only a 28% slowdown in memory access time.
**Paging: Inverted Page Table**

- **Inverted Page Table** - only one for all processes (before - each process has its own Page Table).
- **Inverted Page Table** - one entry for each real page (frame) of memory;
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
  - Decreases memory needed to store each Page Table, but increases time needed to search the table when a page reference occurs.
  - Use Hash Table to limit the search to one- or at most few- page table entries.

---

**Segmentation: Basic Method**

- **Segmentation** - memory-management scheme that supports user view of memory.
- A program is a collection of segments. Each segment has a name and a length. A segment is a local unit such as:
  - Main Program;
  - Procedure;
  - Function;
  - Local / Global Variables;
  - Common Block;
  - Stack;
  - Symbol Table, Arrays.

- Logical address consists of two tuple:
  \(<\text{segment-number}, \text{offset}>\)
- **Segment Table** - maps two-dimensional user-defined addresses into one-dimensional physical addresses; each entry of Table has:
  - \(\text{base}\) - contains the starting physical address where the segments reside in memory.
  - \(\text{limit}\) - specifies the length of the segment.
**Segmentation: Basic Method:** Example of Segmentation

**Logical Address Space**

**Process 0**
- **Segment 0**
  - **Subroutine**
  - **Stack**
- **Segment 1**
- **Segment 2**
- **Segment 3**
- **Segment 4**

**Process 0**
- **Segment 0**
- **Segment 1**
- **Segment 2**
- **Segment 3**
- **Segment 4**

**Segment Table**

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>1400</td>
</tr>
<tr>
<td>1</td>
<td>400</td>
<td>6300</td>
</tr>
<tr>
<td>2</td>
<td>400</td>
<td>4300</td>
</tr>
<tr>
<td>3</td>
<td>1100</td>
<td>3200</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>4700</td>
</tr>
</tbody>
</table>

**Physical Memory**

**Segmentation Hardware I**

**CPU**

- **Logical address**
- **segment #**
- **offset**

**Segment Table**

- **Limit base**
- **physical address**

- **trap; addressing error**

**Segmentation Hardware II**

**CPU**

- **Logical address**
- **segment #**
- **offset**

**Segment Table Limit Register (STRL)**

- **Yes**
- **No**

**Segment Table Base Register (STBR)**

- **Yes**
- **No**

- **limit base**
- **physical address**

- **trap; Segment limit error**

**Translation overhead:** this mapping requires two memory references per logical address.

**Solution:** use a set of associative registers (TLB) to hold the most recently used segment-table entries (only 10-15% slower than unmapped memory access).
Segmentation: Issues Related to Segmentation

- **Protection** and **Sharing** data can be implemented naturally at the segment level.
  - **Protection**: segment- semantically defined portion of program;
    - instructions execute-only or read-only;
    - data read- or write-only);
    - protection bits associated with segments.
  - **Sharing**: segments are shared when entries in the Segment Tables of two different processes point to the same physical locations (see next slide).

Since segments vary in length, memory allocation is a dynamic storage allocation problem (best-fit or first-fit).

Segmentation may cause external fragmentation, when all blocks of free memory (holes) are too small to accommodate a segment.

But we can compact memory whenever we want. Average segment size?
  - One extreme: each process to be one segment.
  - The other extreme: every byte put in its own segment (doubling memory use!).

The next logical step fixed-sized, small segments - is paging.

It is possible to combine Segmentation and Paging to improve on each.

This combination is best illustrated by two different architectures: MULTICS system and Intel 386.
Segmentation with Paging: MULTICS

- MULTICS Operating System
  - Segmentation with Paging.
  - Dynamic linking.
  - Ring protection.
  - Most code was written in PL/1.

- Hardware
  - GE 635 was modified into GE 645.
  - Word machine; 1 word = 36 bits.
  - Max. # of segments = 256 K.
  - Segment size = 64 K words.
  - Page size = 1 K words.