We are attempting to speed up address lookups
Effective Access Time

- Associative Lookup = $\epsilon$ time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = $\alpha$
- **Effective Access Time (EAT)**

\[
EAT = (1 + \epsilon) \alpha + (2 + \epsilon)(1 - \alpha)
\]

\[
= 2 + \epsilon - \alpha
\]
TLB

- Some TLBs support address-space ID
  - OS loans a unique value per process
  - If current process ASID != TLB ASID, then don’t use it
- Otherwise, TLBs are flushed at context switch

- Question: what affects TLB hit ratio?
  - For code?
  - For data?
8.4.3: Memory Protection

- Need some mechanism to identify that a page is not allocated to a process (even though the page table will have an entry for this logical page)

- **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
Valid (v) or Invalid (i) Bit In A Page Table

Next chapter, we will see more uses for this bit
8.5: Page Table Structure

- Problem is that page tables are per-process structure and they can be large
  - Consider 64 bit address space and page size of 8 KB
    - Page table size = $2^{51}$ or $2 \times 10^{15}$ entries

- Hierarchical Paging

- Hashed Page Tables

- Inverted Page Tables
Hierarchical Page Tables

- Break up the logical address space into multiple page tables

- A simple technique is a two-level page table
Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

```
<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1 )</td>
<td>( p_2 )</td>
</tr>
</tbody>
</table>
```

where \( p_1 \) is an index into the outer page table, and \( p_2 \) is the displacement within the page of the outer page table.
Two-Level Page-Table Scheme

![Diagram of Two-Level Page-Table Scheme](image-url)
Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture

Diagram:
- Logical address: p₁ p₂ d
- p₁ \{ \}
- Outer page table
- p₂ \{ \}
- Page of page table
- d \{ \}
Hashed Page Tables

- Common in address spaces > 32 bits

- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.

- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.
Hashed Page Table

logical address

p d

hash function

hash table

r d

physical address

q s

physical memory

p r

...
Inverted Page Table

- One entry for each real 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 a few — page-table entries
Inverted Page Table Architecture

CPU

logical address

physical address

physical memory

page table

search

pid | p

i | d
Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes.

- **Private code and data**
  - Each process keeps a separate copy of the code and data.
  - The pages for the private code and data can appear anywhere in the logical address space.
Shared Pages Example

![Diagram showing shared pages example for processes P₁, P₂, and P₃. Each process has a page table entry for a shared page, and the page table entries are organized into pages 0 to 10. The page table entries for P₁ include data 1, for P₂ include data 2, and for P₃ include data 3. The page table entries for the shared pages are as follows: P₁: ed 1 @ page 3, ed 2 @ page 4, ed 3 @ page 6; P₂: ed 1 @ page 3, ed 2 @ page 4, ed 3 @ page 6; P₃: ed 1 @ page 2.]

3/11/08  CSE 30341: Operating Systems Principles  page 16
8.6: Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:
  - main program,
  - procedure,
  - function,
  - method,
  - object,
  - local variables, global variables,
  - common block,
  - stack,
  - symbol table, arrays
User’s View of a Program

logical address

subroutine
stack
symbol table
main program

Sqrt
Logical View of Segmentation

user space

physical memory space
Segmentation Architecture

- Logical address consists of a two tuple: `<segment-number, offset>`,

- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - `base` – contains the starting physical address where the segments reside in memory
  - `limit` – specifies the length of the segment

- **Segment-table base register (STBR)** points to the segment table’s location in memory

- **Segment-table length register (STLR)** indicates number of segments used by a program; segment number `s` is legal if `s < STLR`
Segmentation Architecture (Cont.)

- **Relocation.**
  - dynamic
  - by segment table

- **Sharing.**
  - shared segments
  - same segment number

- **Allocation.**
  - first fit/best fit
  - external fragmentation
Protection. With each entry in segment table associate:
- validation bit = 0 ⇒ illegal segment
- read/write/execute privileges

Protection bits associated with segments; code sharing occurs at segment level

Since segments vary in length, memory allocation is a dynamic storage-allocation problem

A segmentation example is shown in the following diagram
Address Translation Architecture

CPU → s → segment table (limit, base) → s

- If segment table lookup succeeds (yes):
  - Physical memory

- If segment table lookup fails (no):
  - Trap: addressing error
Example of Segmentation

- **Logical Address Space**
  - Subroutine
  - Stack
  - Symbol Table
  - Main Program

- **Segment Table**
  
<table>
<thead>
<tr>
<th>Segment</th>
<th>Limit</th>
<th>Base</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**
  
<table>
<thead>
<tr>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
Sharing of Segments

Logical memory process $P_1$

Logical memory process $P_2$

Physical memory

Segment table process $P_1$

Segment table process $P_2$

Data 1

Data 2

Editor

Data 1

Data 2

Editor

Segment 0

Segment 1

Segment 0

Segment 1

Base

Limit

Base

Limit

25286

4425

68348

43062

25286

8850

90003

43062

68348

72773

98553

90003

98553
The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.

Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a page table for this segment.
MULTICS Address Translation Scheme

logical address

s  d

STBR

segment table

segment length page-table base

≥

yes

no

trap

p  d′

physical address

page table for segment s

f  d′

memory
8.7: Intel 30386 Address Translation

- segmentation with paging for memory management with a two-level paging scheme
Linux on Intel 80x86

- Uses minimal segmentation to keep memory management implementation more portable
- Uses 6 segments:
  - Kernel code
  - Kernel data
  - User code (shared by all user processes, using logical addresses)
  - User data (likewise shared)
  - Task-state (per-process hardware context)
  - LDT
- Uses 2 protection levels:
  - Kernel mode
  - User mode