Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called frames
  - Size is power of 2, between 512 bytes and 16 Mbytes
  - Keep track of all free frames
- Divide logical memory into blocks of same size called pages
- 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
- Backing store likewise split into pages

Address Translation Scheme

- 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

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$d$</td>
</tr>
<tr>
<td>$m - n$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

- For given logical address space $2^m$ and page size $2^n$
Paging Hardware

Paging Model of Logical and Physical Memory

Paging Example

\( n = 2 \) and \( m = 4 \)

32-byte memory and 4-byte pages
**Paging (Cont.)**

- What is the internal fragmentation of paging?
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - Average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB

- Process view and physical memory now very different
  - By implementation process can only access its own memory

**Free Frames**

- Before allocation
- After allocation

**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 the 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 memory or translation lookaside buffers (TLBs)
  - Some TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
    - Otherwise need to flush at every context switch
  - TLBs typically small (64 to 1,024 entries)
    - On a TLB miss, value is loaded into the TLB for faster access next time
    - Replacement policies must be considered
    - Some entries can be wired down for permanent access
Associative Memory

- Associative memory – parallel search

<table>
<thead>
<tr>
<th>Page #</th>
<th>Frame #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Address translation \( (p, d) \)
  - If \( p \) is in associative register, get frame \# out
  - Otherwise get frame \# from page table in memory

Paging Hardware With TLB

Effective Access Time

- Associative Lookup = \( \varepsilon \) time unit
- Can be < 10\% of memory access time
- Hit ratio = \( \alpha \)
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers

- Effective Access Time (EAT) (exp. in terms of memory access time)
  \[
  EAT = (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) = 2 + \varepsilon - \alpha
  \]

- Consider \( \alpha = 80\% \), \( \varepsilon = 20\text{ns} \) for TLB search, 100\text{ns} for memory access
  - \( EAT = 0.80 \times 120 + 0.20 \times 220 = 140\text{ns} \)
- Consider slower memory but better hit ratio \( \rightarrow \alpha = 98\% \), \( \varepsilon = 20\text{ns} \) for TLB search, 140\text{ns} for memory access
  - \( EAT = 0.98 \times 160 + 0.02 \times 300 = 162.8\text{ns} \)
Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **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
  - Or use PTLR
- Any violations result in a trap to the kernel

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

Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **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

Structure of the Page Table
- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB ($2^{12}$)
  - Page table would have 1 million entries ($2^{32} / 2^{12}$)
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Don't want to allocate that contiguously in main memory
- 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
- We then page the page table
Two-Level Page-Table Scheme

Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-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>P1</td>
<td>P2</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
</tr>
</tbody>
</table>

Address-Translation Scheme
64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{32} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Outer page table has 2^{40} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{44} bytes in size
  - And possibly 4 memory access to get to one physical memory location

Three-level Paging Scheme

<table>
<thead>
<tr>
<th>outer page</th>
<th>inner page</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_2</td>
<td>p_2</td>
<td>d</td>
</tr>
<tr>
<td>42</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2nd outer page</th>
<th>outer page</th>
<th>inner page</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_1</td>
<td>p_2</td>
<td>p_3</td>
<td>d</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

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
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- 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

Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
  - One entry for each real page of memory

- Entry consists of the virtual address of the page stored in that real memory location, with information about the process

- 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 few) entries
  - TLB can accelerate access
  - But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Inverted Page Table Architecture
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 View of Segmentation
Segmentation Architecture

- Logical address consists of a two tuple:
  \(<\text{segment-number}, \text{offset}>\),

- **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 \(s < \text{STLR}\)

Segmentation Architecture (Cont.)

- **Protection**
  - With each entry in segment table associate:
    - validation bit = 0 \(\Rightarrow\) 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

Segmentation Hardware
Example of Segmentation