So far we started from:

- Compile time addressing – not flexible at all
- Contiguous dynamic addressing w/ base + limit – ok, but fragmentation prone
- Compaction to try to deal with fragmentation – overhead, and outstanding IO can be a headache

Paging

- Non-contiguous
- Fragmentation is internal to processes (and thus bounded by page sizes)

In most cases, hardware exists to help us for:

- Speed
- Protection
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

\[
\begin{array}{c|c}
\text{page number} & \text{page offset} \\
p & d \\
m - n & n
\end{array}
\]

- For given logical address space \( 2^m \) and *page size* \( 2^n \)
Paging Hardware
Paging Model of Logical and Physical Memory

logical memory

page 0
page 1
page 2
page 3

page table

0 1
1 4
2 3
3 7

frame number

0 page 0
1 page 0
2 page 2
3 page 2
4 page 1
5 page 1
6 page 1
7 page 3

physical memory
Paging Example

32-byte memory and 4-byte pages
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 (PRLR)** 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.
Hardware to the rescue

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside 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
TLBs

Pentium

Note layout and proximity of TLB to both caches and bus interface to main memory.
Associative Memory

- Associative memory – parallel search

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

Think of TLBs like a specialized cache for page tables.

First, try to look up a page in the TLB.
  - Very fast if there is a hit – on die, register or L1 access speeds

If not in TLB, miss and go out to main memory to get the entry from the page table.
  - Store result in TLB.
  - Temporal and spatial locality says that we are likely to reuse this later and have a hit.
    - Amortizes slow memory access in miss over multiple hits.
Paging Hardware With TLB

Diagram showing the flow of logical to physical addresses through the CPU, TLB, page table, and physical memory. The diagram includes labels for CPU, logical address, page frame number, TLB hit, TLB miss, page table, physical address, and physical memory.
Effective Access Time

- Associative Lookup = $\varepsilon$ 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)

$$EAT = (1 + \varepsilon)\alpha + (2 + \varepsilon)(1 - \alpha)$$

$$= 2 + \varepsilon - \alpha$$

**Hit**: time to access memory since we know address

**Miss**: time to access page table in memory, then time to access what we wanted in the first place
Example

- 70% probability of a hit
- ~100 ns memory read
- ~5 ns TLB lookup

\[
(5+100)*0.7 + (5 + 200)*0.3
\]

= 135 ns EAT

- Not bad. Better than having to go out to memory every time to look up page table entries.

- (Timings based on approximate Xeon ratings)
Memory Protection

- Memory protection implemented by associating protection bit 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
Valid (v) or Invalid (i) Bit In A Page Table

Also possible to accomplish with page table length register: more memory efficient.
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

Process $P_1$

- ed 1
- ed 2
- ed 3
- data 1

Page table for $P_1$

- ed 1
- ed 2
- ed 3
- data 2

Process $P_2$

- ed 1
- ed 2
- ed 3
- data 2

Page table for $P_2$

- ed 1
- ed 2
- ed 3
- data 1

Process $P_3$

- ed 1
- ed 2
- ed 3
- data 3

Page table for $P_3$

- ed 1
- ed 2
- ed 3

Page table

- 0: data 1
- 1: data 3
- 2: ed 1
- 3: ed 2
- 4: ed 3
- 5: ed 3
- 6: data 2
- 7: data 2
- 8
- 9
- 10
- 11
Protection issues

- Shared pages impact multiple processes.
- OS can tag pages as read-only when dynamic libraries are loaded.
  - Shared library support is in the OS, not user space.
- Allows OS to ensure that processes do not make bad changes to shared pages.
Structure of the Page Table

- 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 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:

  page number | page offset
  -----------|-----------
  \( p_1 \)  | \( p_2 \)  | \( d \)

  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
Address-Translation Scheme

Logical Address: p₁ p₂ d

Outer Page Table: p₁

Page of Page Table: p₂ d
### Three-level Paging Scheme

<table>
<thead>
<tr>
<th>outer page</th>
<th>inner page</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</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>
Hierarchy and TLBs

- 2008 whitepaper on Nehalem from Intel (“First the Tick, Now the Tock”)

- One big update is the introduction of a two-level TLB hierarchy
  - Adds a 512 entry second level TLB

- This is an interesting look at current trends in CPU architectures.
  - Clearly paging is very important, and TLB assistance critical for performance if they’re advertising it as a major new feature of year ‘08 architectures.
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

![Diagram of a hashed page table]

- Logical address
- Hash function
- Hash table
- Physical address
- Physical memory
- Elements: p, d, r, d, ..., q, s, p, r, ...
Inverted Page Table

- 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 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

This is the critical difference. \( i \) is physical address. Previous page tables indexed by a virtual address!
Alignment

- Pages are a great abstraction, but as with most abstractions, there is a cost.
- To efficiently use memory as a programmer, we sometimes need to worry about what is known as alignment.

- Memory is read in terms of words (e.g.: 4 byte or 8 byte chunks)
  - A page is a set of N words.

- An aligned address is one that lies on a word boundary.
  - Guaranteed that all of word is in the same page.

- Clearly this can be bad.
  - First off: bad performance
  - Second: can lead to very rare correctness issues
Alignment

- Compilers aid in laying out program data to be aligned properly.

- Alignment of structures can be a little weird though.
  - Consider struct with char and long followed by char.
  - Sometimes “padding” is placed after the shorter (1byte) entries to ensure that the later entries are word aligned.
  - This can cause unexpected bloat of structures.
    - Struct may appear larger in memory than you write in source code.

- Alignment is good to be aware of, primarily when performance tuning or working with instructions or DMA devices that expect certain alignment properties of your data.
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

- subroutine
- stack
- symbol table
- main program

logical address

sqrt
Logical View of Segmentation

user space

physical memory space
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 ⇒ 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

CPU → s, d → s → limit, base → segment table → < yes → + → trap: addressing error → physical memory → no
Example of Segmentation
Example: The Intel Pentium

- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - Given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
Logical to Physical Address Translation in Pentium

The figure illustrates the process of logical to physical address translation in the Pentium processor. The process involves the following stages:

1. **CPU** gets a logical address.
2. **Segmentation unit** processes the logical address to generate a linear address.
3. **Paging unit** further processes the linear address to generate a physical address.
4. **Physical memory** uses the physical address to access the desired memory location.

The figure is complemented by a table that shows the breakdown of a page number and offset:

- **Page number** is divided into two parts: $p_1$ and $p_2$.
- **Page offset** is denoted by $d$.

The table values are:

- $p_1 = 10$
- $p_2 = 10$
- $d = 12$

This representation helps in understanding how the processor translates logical addresses into physical addresses.
Intel Pentium Segmentation

logical address

selector

offset

descriptor table

segment descriptor

32-bit linear address
Intel Nehalem Example

http://pc.watch.impress.co.jp/docs/2008/0424/kaigai_02l.gif

Instruction TLB

Data TLB
Pentium Paging Architecture

(logical address)

CR3 register

page directory

page table

4-KB page

page directory

offset

4-MB page

offset

page directory
Linear Address in Linux

Broken into four parts:

<table>
<thead>
<tr>
<th>global directory</th>
<th>middle directory</th>
<th>page table</th>
<th>offset</th>
</tr>
</thead>
</table>

Three-level Paging in Linux

(Linear address)

<table>
<thead>
<tr>
<th>Global directory</th>
<th>Middle directory</th>
<th>Page table</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global directory entry</td>
<td>Middle directory entry</td>
<td>Page table entry</td>
<td>Page frame</td>
</tr>
</tbody>
</table>

CR3 register