Overview

- Paging
- Page Tables
- TLB
- Shared Pages
- Hierarchical Pages
- Hashed Pages
- Inverted Pages
- Uses
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>
</tbody>
</table>

\(m\) - \(n\) \(n\)

- For given logical address space \(2^m\) and page size \(2^n\)
Paging Example

$n=2$ and $m=4$  32-byte memory and 4-byte pages
Calculating internal fragmentation
- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of 2,048 - 1,086 = 962 bytes
- Worst case fragmentation = 1 frame – 1 byte
- On 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 look-aside buffers (TLBs)**
Implementation of Page Table

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry –
  - uniquely identifies each process
  - 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 fast 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>
<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
Paging Hardware With TLB

CPU → Logical address → \( p \) \( d \) → Page number → Frame number → TLB → TLB hit → physical address → \( f \) \( d \) → physical memory

TLB miss → page table → \( f \) → physical memory
Effective Access Time

• Associative Lookup
  – Extremely fast

• Hit ratio = $\alpha$
  – Hit ratio – percentage of times that a page number is found in the associative memory;
  – Consider $\alpha = 80\%$, 100ns for memory access

• Consider $\alpha = 80\%$, 100ns for memory access
  – EAT = 0.80 x 100 + 0.20 x 200 = 120ns

• Consider hit ratio $\alpha = 99$, 100ns for memory access
  – EAT = 0.99 x 100 + 0.01 x 200 = 101ns
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 **page-table length register (PTLR)**
  – Page Table Entries (PTEs) can contain more information

• Any violations result in a trap to the kernel
Valid (v) or Invalid (i) Bit In A Page Table

frame number
valid–invalid bit

page table

page 0
page 1
page 2
page 3
page 4
page 5

0 2 v
1 3 v
2 4 v
3 7 v
4 8 v
5 9 v
6 0 i
7 0 i

page 0
page 1
page 2
page 3
page 4
page 5
...
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

process $P_1$
- ed 1
- ed 2
- ed 3
- data 1

Page table for $P_1$
- 3
- 4
- 6
- 1

process $P_2$
- ed 1
- ed 2
- ed 3
- data 2

Page table for $P_2$
- 3
- 4
- 6
- 7

process $P_3$
- ed 1
- ed 2
- ed 3
- data 3

Page table for $P_3$
- 3
- 4
- 6
- 2

Data allocation:
- Data 1
- Data 3
- Ed 1
- Ed 2
- Ed 3
- Data 2
Structure of the Page Table

• Memory structures for paging can get huge using straightforward 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:

  \[
  \begin{array}{ccc}
  \text{page number} & \text{page offset} \\
  p_1 & p_2 & d \\
  12 & 10 & 10 \\
  \end{array}
  \]

- where \( p_1 \) is an index into the outer page table, and \( p_2 \) is the displacement within the page of the inner page table
- Known as forward-mapped page table
Address-Translation Scheme

logical address

\[ p_1 \quad p_2 \quad d \]

outer page table

page of page table

\[ p_1 \]

\[ p_2 \]

\[ d \]
64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB ($2^{12}$)
  - Then page table has $2^{52}$ entries
  - If two level scheme, inner page tables could be $2^{10}$ 4-byte entries
  - Address would look like

```
\begin{array}{ccc}
\text{outer page} & \text{inner page} & \text{page offset} \\
\hline
p_1 & p_2 & d \\
42 & 10 & 12 \\
\end{array}
```

- Outer page table has $2^{42}$ entries or $2^{44}$ bytes
- One solution is to add a 2\textsuperscript{nd} outer page table
- But in the following example the 2\textsuperscript{nd} outer page table is still $2^{34}$ 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>$p_1$</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>
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
• Variation for 64-bit addresses is clustered page tables
  – Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  – Especially useful for sparse address spaces (where memory references are non-contiguous and scattered)
Hashed Page Table

logical address

$p|d$

hash function

hash table

physical address

$r|d$

physical memory

$q|s$

$p|r$

$\ldots$
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,
  - 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/few page-table 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

- CPU
- Logical address
- Physical memory
- Page table
- Search
Functionality enhanced by page tables

- Code (instructions) is read-only
  - A bad pointer can’t change the program code
- Dereferencing a null pointer is an error caught by hardware
  - Don’t use the first page of the virtual address space – mark it as invalid – so references to address 0 cause an interrupt
- Inter-process memory protection
  - My address XYZ is different that your address XYZ
- Shared libraries
  - All running C programs use libc
  - Have only one (partial) copy in physical memory, not one per process
  - All page table entries mapping libc point to the same set of physical frames
    - DLL’s in Windows
More functionality

• Generalizing the use of “shared memory”
  – Regions of two separate processes’s address spaces map to the same physical frames
  – Faster inter-process communication
    • Just read/write from/to shared memory
    • Don’t have to make a syscall
  – Will have separate Page Table Entries (PTEs) per process, so can give different processes different access rights
    • E.g., one reader, one writer

• Copy-on-write (CoW), e.g., on fork()
  – Instead of copying all pages, create shared mappings of parent pages in child address space
    • Make shared mappings read-only for both processes
    • When either process writes, fault occurs, OS “splits” the page
Less familiar uses

• Memory-mapped files
  – instead of using open, read, write, close
    • “map” a file into a region of the virtual address space
      – e.g., into region with base ‘X’
    • accessing virtual address ‘X+N’ refers to offset ‘N’ in file
    • initially, all pages in mapped region marked as invalid
  – OS reads a page from file whenever invalid page accessed
  – OS writes a page to file when evicted from physical memory
    • only necessary if page is dirty
More unusual use

- Use “soft faults”
  - faults on pages that are actually in memory,
  - but whose PTE entries have artificially been marked as invalid
- That idea can be used whenever it would be useful to trap on a reference to some data item
- Example: debugger watchpoints
- Limited by the fact that the granularity of detection is the page
Summary

• Paging
• Page Tables
• TLB
• Shared Pages
• Hierarchical Pages
• Hashed Pages
• Inverted Pages
• Uses
• Next time: Virtual Memory