Eliminating External Fragmentation: Tiling Memory
Virtual (P₁)

Physical

Tiling Memory
Virtual (P₁)

Tiling Memory

Physical

page

frame
Virtual (P₁)

Physical

page
frame

Tiling Memory
Tiling Memory

Virtual (P₁)

Physical

page

frame
Tiling Memory
Tiling Memory

P₂

P₁

Physical Memory

0 1 2 3 4 5 6

80 81 82 83

42 43 44 45 46 47

31 32 33 34 35

81 82 83

32 33 5 85 4

80 6 42 4

46 47 5

31 0 1 2

81 82 83

35 34
Eliminating External Fragmentation: Paging

Allocate VA & PA memory in chunks of the same, fixed size (pages and frames, respectively)

Adjacent pages in VA (say, within the stack) need not map to contiguous frames in PA!

- Free frames can be tracked using a simple bitmap
  - 001111100111101110000 one bit/frame

- No more external fragmentation!

- But now internal fragmentation (you just can’t win...)
  - when memory needs are not a multiple of a page
  - typical size of page/frame: 4KB to 16KB
How can I reference a byte in VA space?
Virtual address

Interpret VA as comprised of two components

- **page**: which page?
- **offset**: which byte within that page?
Virtual address

Interpret VA as comprised of two components

- **page**: which page?
  - no. of bits specifies no. of pages are in the VA space
- **offset**: which byte within that page?
Virtual address

Interpret VA as comprised of two components

- **page:** which page?
  - no. of bits specifies no. of pages are in the VA space
- **offset:** which byte within that page?
  - no. of bits specifies size of page/frame
Virtual address

To access a byte

- extract page number
- map that page number into a frame number using a page table
  - Note: not all pages may be mapped to frames
- extract offset
- access byte at offset in frame
Basic Paging

- The Page Table lives in memory.
- The Page Table needs to live in memory.
- The Page Table is stored at the physical address in the Page Table Base Register.
- PTBR value saved/restored in PCB on context switch.
Basic Paging

The Page Table
- lives in memory
- at the physical address stored in the Page Table Base Register
- PTBR value saved/restored in PCB on context switch

Physical Memory

CPU

Page Table

Frame

Access

Helps implement mapping

PTBR

The Page Table too needs to live in memory!
### Page Table Entries

- **Frame number**
- **Present (Valid/Invalid) bit**
  - Set if entry stores a valid mapping. If not, and accessed, page fault
- **Referenced bit**
  - Set if page has been referenced
- **Modified (dirty) bit**
  - Set if page has been modified
- **Protection bits (R/W/X)**

#### Diagram

![Page Table Diagram](image)

<table>
<thead>
<tr>
<th>Frame</th>
<th>Protection bits</th>
<th>Referenced</th>
<th>Modified</th>
<th>Other</th>
<th>Access</th>
<th>Physical memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Sharing

Processes share a page by each mapping a page of their own virtual address space to the same frame
- Fine tuning using protection bits (RWX)

We can refine COW to operate at the granularity of pages
- on fork(), mark all pages in page table Read only

- on write:
  - page fault
  - allocate new frame
  - copy page
  - mark both pages R/W
Example

Page size: 4 bytes
Space overhead

Two sources, in tension:
- **data structure overhead** (the Page Table itself)
- **fragmentation**
  - How large should a page be?

**Overhead for paging:**

\[
(#\text{entries} \times \text{sizeofEntry}) + (#"segments" \times \text{pageSize}/2) = \\
= ((\text{VA\_Size}/\text{pagesize}) \times \text{sizeofEntry}) + (#"segments" \times \text{pageSize}/2)
\]

- What determines sizeofEntry?
  - enough bits to identify physical page (\(\log_2 (\text{PA\_Size} / \text{page size})\))
  - should include control bits (present, dirty, referenced, etc)
  - usually word or byte aligned

sets of contiguous pages
Computing paging overhead

- 1 MB maximum VA, 1 KB page, 3 segments (program, stack, heap)
  - \((2^{20} / 2^{10}) \times \text{sizeofEntry}\) + \((3 \times 2^9)\)
  - If I know PA is 64 KB then \(\text{sizeofEntry} = \text{sizeofFrameNo} + \#\text{ofAccessBits} = 6\) (since we have \(2^6\) frames) + \#\text{ofAccessBits}
    - if 7 access bits, byte aligned size of entry: 16 bits
What's not to love?

Space overhead

- With a 64-bit address space, size of page table can be huge!

Time overhead

- Accessing data now requires two memory accesses
  - must also access page table, to find mapped frame

...and, like most times, space and time are in tension...
Reducing the Storage Overhead of Page Tables

- Size of the page table for a machine with 64-bit addresses and a page size of 4KB?
  - an array of $2^{52}$ entries!

- Good news
  - most space is unused

- Use a better data structure to express the Page Table
  - a tree!

Example
- 32 bit address space
- 4Kb pages
- 4 bytes PTE

Table:

<table>
<thead>
<tr>
<th>Page Address</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0</td>
<td></td>
</tr>
<tr>
<td>VP 1023</td>
<td>Code/Data</td>
</tr>
<tr>
<td>VP 1024</td>
<td></td>
</tr>
<tr>
<td>VP 2047</td>
<td></td>
</tr>
<tr>
<td>Gap</td>
<td></td>
</tr>
<tr>
<td>1023 pages</td>
<td>unallocated</td>
</tr>
<tr>
<td>VP 9215</td>
<td>unallocated</td>
</tr>
<tr>
<td>1 pages</td>
<td>for stack</td>
</tr>
</tbody>
</table>

Page Table
Reducing the Storage Overhead of Page Tables

- Size of the page table for a machine with 64-bit addresses and a page size of 4KB?
  - an array of $2^{52}$ entries!

- Good news
  - most space is unused

- Use a better data structure to express the Page Table
  - a tree!

Example

- 32 bit address space
- 4Kb pages
- 4 bytes PTE

Diagram:

- PTE 0
- PTE 1
- PTE 2
- PTE 1023
- VP 0
- VP 1023
- VP 1024
- VP 2047
- Gap
- 1023 unallocated pages
- VP 9215
- unallocated pages
- 1 page for stack
Reducing the Storage Overhead of Page Tables

Size of the page table for a machine with 64-bit addresses and a page size of 4KB?
- an array of $2^{52}$ entries!

Good news
- most space is unused

Use a better data structure to express the Page Table
- a tree!

Example
- 32 bit address space
- 4Kb pages
- 4 bytes PTE

Diagram:
- PTEs
- VP(s)
- Null PTEs
- Gap
Multi-level Paging

Structure virtual address space as a tree

Virtual address of a SUN SPARC (1987)
Example

What is the page size?
Example

What is the page size?  Page size is 256 bytes ($2^8$)

What is the Page Table size for a process that uses 256 contiguous KB of its VA space starting at address 0? [Assume each PTE is 2 bytes]

- if we used a linear representation of the page table:
What is the page size? Page size is 256 bytes ($2^8$).

What is the Page Table size for a process that uses 256 contiguous KB of its VA space starting at address 0? [Assume each PTE is 2 bytes]

- If we used a linear representation of the page table:
  - Page Table has $2^{24}$ entries
What is the page size? Page size is 256 bytes ($2^8$)

What is the Page Table size for a process that uses 256 contiguous KB of its VAS starting at address 0? [Assume each PTE is 2 bytes]

- if we used a linear representation of the page table:
  - Page Table has $2^{24}$ entries
  - PT Size: $2^{24} \times 2$ bytes = $2^{25}$ bytes = 32MB
What if we use a tree?

- We still need to account for $2^{24}$ pages...
- ...but we are going to partition the PT in a sequence of chunks, each with $2^6$ entries
What is we use a tree?

- We still need to account for $2^{24}$ pages...
- ...but we are going to partition the PT in a sequence of chunks, each with $2^6$ entries. How many such chunks?
Example

What is we use a tree?

- We still need to account for $2^{24}$ pages...
- ...but we are going to partition the PT in a sequence of chunks, each with $2^6$ entries
- we’ll need an index with $2^{18}$ entries...
- ...which we’ll partition in chunks of $2^8$ entries
What is we use a tree?

- We still need to account for $2^{24}$ pages...
- ...but we are going to partition the PT in a sequence of chunks, each with $2^6$ entries
- we’ll need an index with $2^{18}$ entries...
- ...which we’ll partition in chunks of $2^8$ entries. How many such chunks?
Example

What is we use a tree?

- We still need to account for $2^{24}$ pages...
- …but we are going to partition the PT in a sequence of chunks, each with $2^6$ entries
- we’ll need an index with $2^{18}$ entries...
- …which we’ll partition in chunks of $2^8$ entries
- We’ll need an index of $2^{10}$
Example

What is we use a tree?

- We still need to account for $2^{24}$ pages...
- ...but we are going to partition the PT in a sequence of chunks, each with $2^6$ entries
- we’ll need an index with $2^{18}$ entries...
- ...which we’ll partition in chunks of $2^8$ entries
- We’ll need an index of $2^{10}$
Example

Are we better off?

- The number of PT entries now is 
  \((2^6 \times 2^{18}) + (2^{10} \times 2^8) + 2^{10} > 2^{24}!!\)

- But we only need the portion of the tree needed to map the first 1K (\(2^{10}\)) pages!
Example

How many chunks of size $2^6$ are needed to hold $2^{10}$ PTEs of consecutive pages starting at 0?

$2^{10} / 2^6 = 2^4 = 16$
Example

How many chunks of size $2^6$ are needed to hold $2^{10}$ PTEs of consecutive pages starting at 0?

$\frac{2^{10}}{2^6} = 2^4 = 16$
How many chunks of size $2^6$ are needed to hold $2^{10}$ PTEs of consecutive pages starting at 0?

$2^{10}/2^6 = 2^4 = 16$

How many chunks of size $2^8$ are needed to hold pointers to 16 pink chunks?

$1$
Example

How many chunks of size $2^6$ are needed to hold $2^{10}$ PTEs of consecutive pages starting at 0?

$2^{10}/2^6 = 2^4 = 16$

How many chunks of size $2^8$ are needed to hold pointers to 16 pink chunks?

1

So, if each PTE is 2 bytes, the PT takes

$2 \times (1 \times 1024 + 1 \times 256 + 16 \times 64) = 4608$ bytes
Getting sloooower

- Every new level of paging
  - reduces the memory overhead for computing the mapping function...
  - ... but increases the time necessary to perform the mapping function
Caching!

Keep the results of recent VA-PA translations in a structure called Translation Lookaside Buffer (TLB)

- TLB is a cache for page-to-frame mappings
Speeding things up: The TLB

CPU $\leq$ Memory Exception

PTBR

Page Table Base Register

TLB miss $\neq$

TLB hit

Physical Memory

EAT: Cost(TLB lookup) + Cost(full translation) $\times (1 - \alpha)$

$\alpha$: hit ratio
Access TLB before accessing memory!
Hit and Miss

- The TLB is small; it cannot hold all PTEs
  - it can be fast only if it is small!
- Some translations will inevitably miss the TLB
- Must access memory to find the appropriate PTE
  - called walking the page table
  - incurs large performance penalty
Handling TLB Misses: Hardware

- **Hardware-managed (e.g., x86)**
  - The hardware does the page walk
  - Hardware fetches PTE and inserts it in TLB
    - If TLB is full, must replace another TLB entry
  - Done transparently to system software
Handling TLB Misses: Software

- **Software-managed** (e.g., MIPS)
  - Hardware raises an exception, trap handler runs in kernel
  - Handler does the page walk, fetches PTE, and inserts/evicts entries in TLB
  - Handler must return to the same instruction that caused the trap!
  - Careful not to generate a TLB miss while running the handler!
Tradeoffs, Tradeoffs...

**Hardware-managed TLB**
- No exception on TLB miss. Instruction just stalls
- No extra instruction/data brought into the cache
- OS has no flexibility in deciding Page Table: hardware must know location and format of PTEs

**Software-managed TLB**
- OS can define Page Table organization
- More flexible TLB entry replacement policies
- Slower: exception causes to flush pipeline; execute handler; pollute cache
TLB Coverage

What fraction of memory can be accessed without TLB misses?

- Low TLB coverage can result in a large number of memory references

1000-fold decrease in 15 years!

Navarro et al., OSDI 2002
Superpages

- Wider TLB coverage by supporting page sizes that are multiples of the base page size: superpages
  - Pentium: 4KB base; 4MB Super
  - Itanium: 10 sizes, from 4KB (base) to 256 MB

- A set of contiguous base pages can be promoted to a superpage
- Demotion works the other way around
Tradeoffs, Tradeoffs...

+ Improved TLB coverage! but...
  - Larger internal fragmentation
  - External fragmentation (?)
    ☐ superpage of N base pages
    ☐ N free base frames free, but not contiguous
  - Less efficient reading
  - Coarser granularity for dirty, reference, and protection bits
TLB Consistency – I

- On context switch
  - VAs of old process should no longer be valid
  - Change PTBR — but what about the TLB?
TLB Consistency – I

On context switch

- VAs of old process should no longer be valid
- Change PTBR — but what about the TLB?
  - Option 1: Flush the TLB
  - Option 2: Add pid tag to each TLB entry

<table>
<thead>
<tr>
<th>TLB Entry</th>
<th>PID</th>
<th>VirtualPage</th>
<th>PageFrame</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x0053</td>
<td>0x0012</td>
<td>R/W</td>
<td></td>
</tr>
</tbody>
</table>

Ignore entries with wrong PIDs
What if OS changes permissions on page?

- If permissions are reduced, OS must ensure affected TLB entries are purged
  - e.g., on copy-on-write

- If permissions are expanded, no problem
  - new permissions will cause an exception and hardware and OS will restore consistency