Virtual Memory and Paging
Lots of separate processes
Data Structure: Page Table

A page table is an array of page table entries (PTEs) that maps virtual pages to physical pages:
- Per-process kernel data structure in RAM memory

![Page Table Diagram]

Copyright ©: University of Illinois CS 241 Staff
Reference to VM address that is in physical memory (RAM hit)

- Virtual address
- Physical page number or disk address
- Memory resident page table (RAM)
- Physical memory (RAM)
- Disk

PTE 0

PTE 7
Page Fault

- Reference to VM address that is not in physical memory (RAM miss)

**Virtual address**

**Physical page number or disk address**

**Physical memory (RAM)**

**Disk**

**Memory resident page table (RAM)**

Copyright ©: University of Illinois CS 241 Staff
Handling Page Faults

- Page miss causes page fault (an exception)

![Diagram of memory management](image)
Handling Page Faults

- Page fault handler selects a victim to be evicted (here VP 4)

Virtual address

Physical page number or disk address

PTE 0

PTE 7

Memory resident page table (RAM)

Physical memory (RAM)

Disk
Handling Page Faults

- Loads new page into freed frame
Handling Page Faults

- Offending instruction is restarted: page hit!
Virtual Memory

- Page table has to be in main memory. If each process has a 4Mb page table, the amount of memory required to store page tables would be unacceptably high
  - 32bits address space $\rightarrow 2^{32} = 4$GB
  - PTE size = 4bytes
  - page size 4KB=$2^{12} \rightarrow (2^{32-12})\times 4$bytes = 4MB size of page table TOO BIG!

**How can we reduce memory overhead due to paging mechanism?**

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page number</td>
<td>Other control bits</td>
</tr>
</tbody>
</table>

Page table entry (PTE)

- PA | Frame number | Other control bits
Page Table Size

- Suppose
  - 4KB ($2^{12}$) page size, 64-bit address space, 8-byte PTE
- How big does the page table need to be?
Page Table Size

- Suppose
  - 4KB \((2^{12})\) page size, 64-bit address space, 8-byte PTE

- How big does the page table need to be?
  - 32,000 TB!
  - \(2^{64} \times 2^{-12} \times 2^3 = 2^{55}\) bytes
Virtual Memory

- How can we reduce memory overhead due to paging mechanism?

- Most virtual memory schemes use a two-level (or more) scheme to store large page tables in kernel memory and second level can be swapped out to disk rather than always being in physical memory
  - First level is a root page table (always in kernel memory) and each of its entries points to a second level page table stored in kernel memory (if allocated) or swapped out to disk
  - If root page table has X entries and each 2nd level page table has Y entries, then each process can have up to X*Y pages
  - Size of each second level page table is equal to the page size

<table>
<thead>
<tr>
<th>Virtual address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page number</td>
</tr>
<tr>
<td>Offset</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page table entry (PTE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
</tr>
</tbody>
</table>
Two level hierarchical page table

- Example of a two-level scheme with 32-bit virtual address
  - Assume byte-level addressing and 4-Kb pages ($2^{12}$)
  - The 4-Gb ($2^{32}$) virtual address space is composed of $2^{20}$ pages
  - Assume each page table entry (PTE) is 4 bytes
  - Total user page table would require 4-Mb ($2^{22}$ bytes); it can be divided into $2^{10}$ pages (second level page tables) mapped by a root table with $2^{10}$ PTEs and requiring 4-Kb
  - 10 most significant bits of a virtual address are used as index in the root page table to identify a second level page table
  - If required page table is not in main memory, a page fault occurs
  - Next 10 bits of virtual address are used as index in the page table to map virtual address to physical address

<table>
<thead>
<tr>
<th>Virtual address (32 bits ➔ 4 Gbyte virtual address space)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 bits root table index</td>
</tr>
</tbody>
</table>
Two level page table hierarchy

virtual address (32 bits) \( \rightarrow \) addresses a byte in VM

addresses a byte in Physical Mem.

*32 bits aligned onto a 4-KByte boundary.
Two level page table hierarchy

Level 1 page table
- PTE 0
- PTE 1
- PTE 2 (null)
- PTE 3 (null)
- PTE 4 (null)
- PTE 5 (null)
- PTE 6 (null)
- PTE 7 (null)
- PTE 8
- (9 - 1K) null PTEs

Level 2 page tables
- PTE 0
- PTE 1
- PTE 2 (null)
- PTE 3 (null)
- PTE 4 (null)
- PTE 5 (null)
- PTE 6 (null)
- PTE 7 (null)
- PTE 8
- 1023 null PTEs
- PTE 1023

Physical memory
- VP 0
- ... PTE 1023
- VP 1023
- VP 1024
- ... PTE 1023
- VP 2047
- Gap
- 1023 unallocated pages
- VP 9215

2048 allocated VM pages for code and data
6K unallocated VM pages
1023 unallocated pages
1 allocated VM page for the stack

32 bit addresses
4KB pages, 4-byte PTEs

Copyright ©: University of Illinois CS 241 Staff
Multilevel Page Tables

What happens on a page fault?

- MMU looks up index in root page table to get 2\textsuperscript{nd} level page table
- MMU tries to access 2\textsuperscript{nd} level page table
  - May result in another page fault to load the 2\textsuperscript{nd} level page table!
- MMU looks up index in 2\textsuperscript{nd} level page table to retrieve physical page address and loads the page in physical memory from disk
- CPU re-executes the faulty instruction and accesses the physical memory address
Multilevel Page Tables

- Issue
  - Page translation has very high overhead
  - Up to three memory accesses plus two disk I/Os
Speeding up Translation: TLB

- **Page table entries (PTEs) are cached**
  - PTEs may be evicted by other memory references
  - PTE hit still requires a small cache access delay

- **Solution: Translation Lookaside Buffer (TLB)**
  - Small, dedicated, fast hardware cache of PTEs in MMU
  - Contains complete page table entries for small number of pages
  - TLB is a “set associative cache”; hence, the processor can query in parallel the TLB entries to determine if there is a match
  - TLB works like a memory cache and it exploits “principle of locality”
Translation Lookaside Buffer: example of a simplified TLB

Note that each TLB entry must include the virtual page # (TLB tag) as well as the corresponding PTE.
When a virtual address is presented to MMU, the hardware checks TLB by comparing a set of entries simultaneously.

If match is valid, the frame # is taken from TLB without going through page table.

If a match is not found
  - MMU detects miss and does a regular page table lookup.
  - It then evicts one old entry out of TLB and replaces it with the new one; so next time, the PTE for that page will be found in TLB.
Page Table Problem (from Tanenbaum)

- Suppose
  - 32-bit virtual address space
  - Two-level page table
  - Virtual addresses split into a 9-bit top-level page table field, an 11-bit second-level page table field, and an offset

- Question: How large are the pages and how many are there in the address space?
  - Offset
  - Page size
  - # virtual pages
Page Table Problem (from Tanenbaum)

- Suppose
  - 32-bit address
  - Two-level page table
  - Virtual addresses split into a 9-bit top-level page table field, an 11-bit second-level page table field, and an offset

- Question: How large are the pages and how many are there in the address space?
  - Offset 12 bits
  - Page size $2^{12}$ bytes = 4 KB
  - # virtual pages $(2^{32} / 2^{12}) = 2^{20}$
  - Note: driven by number of bits in offset
    - Independent of size of top and 2nd level tables
Paging as a tool for protection and sharing

- PTEs have permission bits
- Page fault handler checks these before remapping
  - If violated, send process SIGSEGV (segmentation fault)

<table>
<thead>
<tr>
<th>Process i:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PPage 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PPage 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PPage 2</td>
</tr>
<tr>
<td>VP 3:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process j:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PPage 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PPage 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PPage 11</td>
</tr>
</tbody>
</table>

Physical Address Space
- PP 2
- PP 4
- PP 6
- PP 8
- PP 9
- PP 11
Paging as a tool for protection and sharing

- Process 1 creates a shared memory object "/shm_obj" (with shm_open)
Process 1 creates a shared memory object "/shm_obj" (with shm_open)

Process 1 maps "/shm_obj" in its virtual address space using mmap
Paging as a tool for protection and sharing

- Process 2 opens the same shared memory object “/shm_obj” (with shm_open)
- Process 2 maps “/shm_obj” in its virtual address space using mmap

⇒ Notice how the virtual addresses can be different
Protection + Sharing Example

- **fork()** creates **exact** copy of a process
  - Lots more on this next week…
- When we fork a new process, all of the memory is duplicated for the child
  - Does it make sense to make a copy of **all** of its memory?
  - What if the child process doesn't end up touching most of the memory the parent was using?
Some processes may need to access the same memory

Copy-on-Write (COW)
- Allows parent and child processes to initially share the same pages in memory
- Only copy page if one of the processes modifies the shared page
- More efficient process creation
Copy-on-Write

1. Parent forks a child process
2. Child gets a copy of the parent's page tables

They point to the same physical frames!!!
Copy-on-Write

- All pages (both parent and child) marked read-only
  - Why?

![Diagram of memory layout for parent and child processes](image)
Copy-on-Write

- What happens when the child reads the page?
  - It just accesses same memory as parent .... Niiiiice!
Copy-on-Write

- What happens when the child writes the page?
  - Protection fault occurs (page is read-only!)
  - OS copies the page and maps it R/W into the child's addr space

![Diagram of Copy-on-Write](image.png)
Copy-on-Write

- What happens when the child writes the page?
  - Protection fault occurs (page is read-only!)
  - OS copies the page and maps it R/W into the child's addr space
Sharing Code Segments

Shell #1

(Reserved for OS)

Heap

Uninitialized vars

Initialized vars

Code

Stack

Initialized vars

Uninitialized vars

Code

Same page table mapping! ➔ code shares the same physical frames in main memory

Copyright ©: University of Illinois CS 241 Staff