CS 4284
Systems Capstone

Virtual Memory – Page Tables
Godmar Back
Virtual Memory
Virtual Memory

• Is not a “kind” of memory
• Is a technique that combines one or more of the following concepts:
  – Address translation (always)
  – Paging (not always)
  – Protection (not always, but usually)
• Can make storage that isn’t physical random access memory appear as though it were
Goals for Virtual Memory

• Virtualization
  – Maintain illusion that each process has entire memory to itself
  – Allow processes access to more memory than is really in the machine (or: sum of all memory used by all processes > physical memory)

• Protection
  – make sure there’s no way for one process to access another process’s data
Context Switching

- Process 1
- Process 2

user mode

kernel mode

P1 starts
P2 starts
P1 exits
P2 exits
Process 1 Active in user mode

access possible in user mode
Process 1 Active in kernel mode

- ucode (1)
- kcode
- kbss
- kdata
- kheap
- ustack (1)

0

C0000000

3 GB

1 GB

C0400000

FFFFFFFFFF

access requires kernel mode

access possible in user mode

user (2)
user (2)
user (1)
user (1)
kernel
kernel
kernel
kernel

free

used

P1

Virginia Tech

CS 4284 Spring 2015
Process 2 Active
in kernel mode

access requires kernel mode

access possible in user mode

user (2)
user (2)
user (2)
user (1)
user (1)
user (1)
kernel
kernel
kernel
kernel
free
used

kheap
kbss
kdata
kcode
ustack (2)
udata (2)
ucode (2)

C0000000
C0400000
FFFFFFFFFF

0
3 GB
1 GB
Process 2 Active in user mode

access possible in user mode

CS 4284 Spring 2015
Page Tables

• How are the arrows in previous pictures represented?
  – Page Table: mathematical function “Trans”

\[
\text{Trans:} \quad \{ \text{Process Ids} \} \times \{ \text{Virtual Addresses} \} \times \{ \text{user, kernel} \} \times \emptyset (\{ \text{read, write, execute} \}) \\
\rightarrow \{ \text{Physical Addresses} \} \cup \{ \text{INVALID} \}
\]

• Typically (though not on all architectures) have
  – \( \text{Trans}(p_i, v_a, \text{user}, \ast) = \text{Trans}(p_i, v_a, \text{kernel}, \ast) \)
    OR
    \( \text{Trans}(p_i, v_a, \text{user}, \ast) = \text{INVALID} \)
  – E.g., user virtual addresses can be accessed in kernel mode
Sharing Variations

- We get user-level sharing between processes $p_1$ and $p_2$ if
  - $\text{Trans}(p_1, v_a, \text{user}, *) = \text{Trans}(p_2, v_a, \text{user}, *)$

- Shared physical address doesn’t need to be mapped at same virtual address, could be mapped at $v_a$ in $p_1$ and $v_b$ in $p_2$:
  - $\text{Trans}(p_1, v_a, \text{user}, *) = \text{Trans}(p_2, v_b, \text{user}, *)$

- Can also map with different permissions: say $p_1$ can read & write, $p_2$ can only read
  - $\text{Trans}(p_1, v_a, \text{user}, \{\text{read, write}\}) = \text{Trans}(p_2, v_b, \text{user}, \{\text{read}\})$

- In Pintos (and many OS) the kernel virtual address space is shared among all processes & mapped at the same address:
  - $\text{Trans}(p_i, v_a, \text{kernel}, *) = \text{Trans}(p_k, v_a, \text{kernel}, *)$ for all processes $p_i$ and $p_k$ and $v_a$ in $[0xC0000000, 0xFFFFFFFF]$
Per-Process Page Tables

• Can either keep track of all mappings in a single table, or can split information between tables
  – one for each process
  – mathematically: a projection onto a single process
• For each process pi define a function PTransi as
  – $PTransi(va, *, *) = \text{Trans}(pi, va, user, *)$
• Implementation: associate representation of this function with PCB, e.g., per-process hash table
  – Entries are called “page table entries” or PTEs
• Nice side-effect:
  – reduced need for synchronization if every process only adds/removes entries to its own page table
Per-Process Page Tables (2)

• Common misconception
  – “User processes use ‘user page table’ and kernel uses ‘kernel page table’” – as if those were two tables

• Not so (on x86): mode switch (interrupt, system call) does not change the page table that is used
  – It only “activates” those entries that require kernel mode within the current process’s page table

• Consequence: kernel code also cannot access user addresses that aren’t mapped
Non-Resident Pages

• When implementing virtual memory, some of a process’s pages may be swapped out
  – Or may not yet have been faulted in
• Need to record that in page table:

Trans (with paging):
{ Process Ids } \times \{ Virtual Addresses \} \times \{ user, kernel \} \times \emptyset(\{ read, write, execute \})

\rightarrow \{ Physical Addresses \} \cup \{ INVALID \} \cup \{ Some Location On Disk \}
Implementing Page Tables

• Many, many variations possible
• Done in combination of hardware & software
  – Hardware part: dictated by architecture
  – Software part: up to OS designer
    • Machine-dependent layer that implements architectural constraints (what hardware expects)
    • Machine-independent layer that manages page tables
• Must understand how TLB works first
Page Tables Function & TLB

For each combination (process id, virtual_addr, mode, type of access) must decide
  – If access is permitted
  – If permitted:
    • if page is resident, use physical address
    • if page is non-resident, page table has information on how to get the page in memory

CPU uses TLB for actual translation – page table feeds the TLB on a TLB miss
Intel Pentium 4 Northwood

Buffer Allocation & Register Rename
- Instruction Queue (for less critical fields of the uOps)
- General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)
- Floating Point, MMX, SSE2
- Renamed Register File 128 entries of 128 bit.

uOp Schedulers
- FP Move Scheduler: (8x8 dependency matrix)
- Parallel (Matrix) Scheduler for the two double pumped ALUs
- General Floating Point and Slow Integer Scheduler: (8x8 dependency matrix)
- Load / Store uOp Scheduler: (8x8 dependency matrix)
- Load / Store Linear Address Collision History Table

Integer Execution Core
1. uOp Dispatch unit & Replay Buffer Dispatches up to 6 uOps / cycle
2. Integer Renamed Register File 128 entries of 32 bit + 6 status flags 12 read ports and six write ports
3. Datashub switch & Bypasses to and from the Integer Register File.
4. Flags, Write Back
5. Double Pumped ALU 0
6. Double Pumped ALU 1
7. Load Address Generator Unit
8. Store Address Generator Unit
9. Load Buffer (48 entries)
10. Store Buffer (24 entries)
11. ROB Reorder Buffer 3x42 entries 8 kByte Level 1 Data cache four-way set associative. IR/TW
12. Summed Address Index decode and Way Predictor 256 bit wide bus to and from L2 cache

Execution Pipeline Start
- Register Alias History Tables (2x126)
- Register Alias Tables
- uOp Queue
- Micro code Sequencer
- Micro code ROM & Flash

Instruction Trace Cache
- Trace Cache Fill Buffers
- Distributed Tag comparators 24 bit virtual Tags
- 12L uOps
- 80 kByte
- 8 way set associative
- 8 x 256 sets of 6 uOps

Instruction Decoder
- Up to 4 decoded uOps/cycle out. (from max. one x86 instr/cycle)
- Instructions with more than four are handled by Micro Sequencer
- Trace Cache LRU bits
- Raw Instruction Bytes in Data TLB, 64 entry fully associative, between threads dual ported (for loads and stores)

Instruction Fetch from L2 cache and Branch Prediction
- Front End Branch Prediction Tables (BTB), shared, 4096 entries in total

Trace Cache Access, next Address Predict
- Trace Cache Branch Prediction Table (BTB), 512 entries.
- Return Stacks (2x16 entries)
- Trace Cache next IP's (2x)
- Miscellaneous Tag Data

Front Side Bus Interface, 400..800 MHz

April 19, 2003 www.chip-architect.com
**TLB: Translation Look-Aside Buffer**

- Virtual-to-physical translation is part of every instruction (why not only load/store instructions?)
  - Thus must execute at CPU pipeline speed
- TLB caches a number of translations in fast, fully-associative memory
  - typical: 95% hit rate (*locality of reference principle*)

<table>
<thead>
<tr>
<th>Perm</th>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>RWX K</td>
<td>0xC0000</td>
<td>0x00000</td>
</tr>
<tr>
<td>RWX K</td>
<td>0xC0001</td>
<td>0x00001</td>
</tr>
<tr>
<td>R-X K</td>
<td>0xC0002</td>
<td>0x00002</td>
</tr>
<tr>
<td>R-- K</td>
<td>0xC0003</td>
<td>0x00003</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

VPN: Virtual Page Number
PPN: Physical Page Number

![Diagram](image)
TLB Management

• Note: on previous slide example, TLB entries did not have a process id
  – As is true for x86
• Then: if process changes, some or all TLB entries may become invalid
  – X86: flush entire TLB on process switch (refilling adds to cost!)
• Updates to TLB are immediately written to PTE (dirty bit, access bit)
• Some architectures store process id in TLB entry (MIPS)
  – Flushing (some) entries only necessary when process id reused
Address Translation & TLB

Virtual Address

TLB Lookup

Page Table Walk

Page present

Page Fault Exception “Page Not Present”

page present

done in hardware

done in OS software

done in software or hardware

else

else

Check Permissions

denied

Page Fault Exception “Protection Fault”

Physical Address

ok

Terminate Process

Load Page

TLB Reload

machine-dependent logic

machine-dependent

restart instruction

done in hardware

done in OS software

done in software or hardware

Load Page

done in hardware

done in OS software

done in software or hardware

Page Fault Exception

“Page Not Present”

Page Fault Exception

“Protection Fault”
TLB Reloaded

- TLB small: typically only caches 64-2,048 entries
  - What happens on a miss? – must consult (“walk”) page table – TLB Reload or Refill
- TLB Reload in software (MIPS)
  - Via TLB miss handlers – OS designer can pick any page table layout – page table is only read & written by OS
- TLB Reload in hardware (x86)
  - Hardware & software must agree on page table layout \textit{inasmuch} as TLB miss handling is concerned – page table is read by CPU, written by OS
- Some architectures allow either (PPC)
Page Tables vs TLB Consistency

- No matter which method is used, OS must ensure that TLB & page tables are consistent
  - On multiprocessor, this may require “TLB shootdown”
- For software-reloaded TLB: relatively easy
  - TLB will only contain what OS handlers place into it
- For hardware-reloaded TLB: two choices
  - Use same data structures for page table walk & page loading (hardware designers reserved bits for OS’s use in page table)
  - Use a layer on top (facilitates machine-independent implementation) – this is the recommended approach for Pintos Project 3
    - In this case, must update actual page table (on x86: “page directory”) that is consulted by MMU during page table walk
    - Code is already written for you in pagedir.c
Hardware/Software Split in Pintos

Machine-independent Layer:
your code & data structures
“supplemental page table”

Machine-dependent Layer:
pagedir.c code

CPU cr3

CS 4284 Spring 2015
Layers in Address Translation

- **MMU**: Virtual address to Physical address
- **OS Data Structures**: keeps track of all virtual pages for each process, resident or non-resident
- **Hardware Page Table**: updates access/dirty bits, provides translations
- **TLB**: refills TLB, updates PTE
- **Virtual address**
- **Physical address**

(Does not exist if TLB reload is implemented in software)
Representing Page Tables

• Choice impacts speed of access vs size needed to store mapping information:
  – Simple arrays (PDP-11, VAX)
    • Fast, but required space makes it infeasible for large, non-continuous address spaces
  – Search trees (aka “hierarchical” or “multi-level” page tables)
  – Hash table
Two-level Page Table

• This is the original page design on i386
  - Next slide provides details on how address bits index tree levels
  - If address space is sparse, interior nodes/paths do not exist (parent contains a NULL)
  - Picture shown shows how tree divides virtual address space
  - The kernel virtual pages that hold the page table information need not be contiguous

Figure 8.4 A Two-Level Hierarchical Page Table

• Q.: how many pages are needed in
  – Minimum case
  – Worst case? (what is the worst case?)
Example: x86 Address Translation

- Two-level page table
- Source: [IA32-v3] 3.7.1

*32 bits aligned onto a 4-KByte boundary.
Example: x86 Page Table Entry

- Note: if bit 0 is 0 ("page not present") MMU will ignore bits 1-31 – OS can use those at will
Page Table Management on Linux

• Interesting history:
  – Linux was originally x86 only with 32bit physical addresses. Its page table matched the one used by x86 hardware
  – Since:
    • Linux has been ported to other architectures
    • x86 has grown to support 36bit physical addresses (PAE) – required 3-level page table

• Linux’s now uses 4-level page table to support 64-bit architectures
Linux Page Tables (2)

- On x86 – hardware == software
  - On 32-bit (no PAE) middle directory disappears
- With four-level, “PUD” page upper directory is added (not shown)
Inverted Page Tables

- Alternative to multi-level page tables – size is $O(\text{physical memory})$
Page Tables - Summary

• Page tables store mapping information from virtual to physical addresses, or to find non-resident pages
  – Input is: process id, current mode (user/kernel) and kind of access (read/write/execute)
• TLBs cache such mappings
• Page tables are consulted when TLB miss occurs
  – Either all software, or in hardware
• OS must maintain its page table(s) and, if hardware TLB reload is used, the page table (on x86 aka “page directory + table”) that is consulted by MMU
  – These two tables may or may not be one and the same
• The OS page table must have sufficient information to load a page’s content from disk