Inf2C - Computer Systems
Lecture 14-15
Virtual Memory

Boris Grot

School of Informatics
University of Edinburgh
Previous lecture: Memory hierarchy

- Main idea: exploit locality in memory references to create the illusion of a fast & large memory
  - Temporal vs spatial locality
- Memory hierarchy levels: registers, cache (≥1 levels), main memory, disk
- Cache: hardware-managed storage
  - Exploits temporal & spatial locality
  - Fully-associative vs direct mapped
  - Replacement policy
Lecture 14-15: Virtual memory

- Motivation
- Overview
- Address translation
- Page replacement
- Fast translation – TLB
Motivation

Virtual memory addresses two main problems:

1) Capacity: how do we relieve the programmers/users from dealing with limited main memory?
   - Want to allow for the physical memory to be smaller than the program’s address space (e.g., 32 bits → 4GB)
   - Want to allow multiple programs to share the limited physical memory with no human intervention

2) Safety: how do we allow for safe and efficient sharing of memory among multiple programs?
   - Want to prevent user programs from accessing the memory used by the OS
   - Want strict control of access by each user program to memory of other user programs
Virtual Memory

- Basic idea: each program thinks it owns the entire memory → the **virtual address space**
  - PC and load/store addresses are **virtual addresses**

- Actual main memory: **physical address space**
  - Virtual addresses are **translated** on-the-fly to physical addresses
  - Parts of virtual address space not recently used are stored on disk

- Address translation is done jointly by the OS and hardware
Address translation for 1 program

Virtual Address Space (4 GB)

0

2^{12}

2^{32}-1

A

B

C

D

Physical Memory (1 GB)

0

2^{30}-1

A

C

B

Disk

Hardware/OS

OS Virtual Memory Manager
Physical memory as cache for VM

- Virtual memory space can be larger than physical memory
  - Programmer always sees the full address space (MIPS: $2^{32}$ bytes)
- Physical memory used as a cache for the virtual memory
  - Physical memory holds the currently used portions of a program’s code and data (exploits locality!)
- Secondary storage (disk or flash) “backs” the physical memory
  - OS reserves a portion of the disk for **swap space**
  - OS swaps portions of each process’ code and data areas in & out of physical memory on demand (process called **paging**)
  - Swapping is transparent to the programmer
Paging

- A “cache line” or “block” of VM is called a page
  - Simply “page” or “virtual page” for virtual memory
  - “Page frame” or “physical page” for physical memory

- Typical sizes are 4-8 KB (MB or GB in servers)
  - Large enough for efficient disk use and to keep translation tables (aka, page tables) small

- Mapping is done through a per-program page table
  - Allows control of which pages each program can access
  - Different programs can use same virtual addresses
## Typical Virtual Memory Parameters

<table>
<thead>
<tr>
<th>parameter</th>
<th>Cache</th>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>1KB – 1MB</td>
<td>128MB - 128GB</td>
</tr>
<tr>
<td>block/page</td>
<td>16 - 128 bytes</td>
<td>4KB (up to 4GB)</td>
</tr>
<tr>
<td>hit time</td>
<td>2-20 cycles</td>
<td>100 - 200 cycles</td>
</tr>
<tr>
<td>miss penalty</td>
<td>8 - 200 cycles</td>
<td>1M - 10M cycles</td>
</tr>
<tr>
<td>miss rate</td>
<td>0.1 - 10%</td>
<td>0.00001 - 0.001%</td>
</tr>
</tbody>
</table>

- Virtual Memory miss is called a **page fault**
Address Translation

Need:
- A mapping from virtual (V) to physical (F) page numbers
- Page offset not translated
- Must be efficient (in time and space)

Solution: **Page Table!**
Address Translation

Example: 1KB pages

Need:
- A mapping from virtual (V) to physical (F) page numbers
- Page offset not translated
- Must be efficient (in time and space)

Solution: Page Table!

Virtual address: 32 bits

<table>
<thead>
<tr>
<th>V</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>page number (22 bits ⇒ 4M pages)</td>
<td>page offset (10 bits ⇒ 1KB)</td>
</tr>
</tbody>
</table>

Physical address: 30 bits

<table>
<thead>
<tr>
<th>F</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1GB of main memory)</td>
<td></td>
</tr>
</tbody>
</table>
Address Translation

- virtual address: 32 bits
  - page number (22 bits ⇒ 4M pages)
  - page offset (10 bits ⇒ 1KB)

- physical address: 30 bits
  (1GB of main memory)

- page table base address (physical)

- page table entry (PTE)
  - page number
  - page offset
  - status bits

- page table:
  - per program
  - one entry per page (e.g. 4M entries)
  - located in the system portion of main memory
Practice problem

What is the size of the page table given a 32-bit virtual address space, 4 KB physical pages, and 1 GB of main memory?
Moving pages to/from memory

- Pages are allocated on demand
  - E.g., program launch (results in pages allocated for code, data, and stack); malloc (heap space)

- Pages are replaced and swapped to disk when system runs out of free page frames
  - Aim to replace pages not recently used (principle of locality). Access (A) bit for a page is set whenever page is accessed and is reset periodically
  - If any data in page has been modified, the page must be written back to disk: Modified (M) bit in status bits is set

- Access to a swapped-out page causes a page-fault which invokes the OS through the interrupt mechanism
  - Residence (R) bit in page table status bits is zero
Providing Protection

- Each page table entry can have permission bits that control whether
  - the process is allowed to access a page
  - read & write, read-only or execute-only access is allowed

- This enables per-process memory protection
  - E.g. can set up private and shared areas

- Important that only OS can change page tables
  - How? Next lecture!
Fast address translation

- Problem: page table accesses add latency to each memory access
  - Two memory accesses per load or store (1 to get the page table entry + 1 for the actual load/store)

- Fast address translation: Translation Lookaside Buffer (TLB) contained in the MMU
  - A cache of page table entries
    - Each TLB entry holds translation information, not program data
    - Tag: virtual address. Entry: physical frame address
  - Small and fast table in hardware, located close to processor
  - Can capture most translations due to principle of locality
  - When page not in TLB: access the page table and save the translation entry in TLB
Translation Look-aside Buffer (TLB)

TLB: a small, fully-associative cache of page table entries
- Accessed with Virtual page number
- Each entry stores the translation (PPN) for a given VPN
- Physical address formed from PPN and Page Offset
- Page table accessed on a TLB miss

TLB status bits:
- V (valid) bit indicates a valid entry
- D (dirty) bit indicates whether page has been modified
- R, W, X permission bits. Checked on every memory access

Note: physical address is always 32 bits, regardless of actual physical memory size
Integrating a TLB with the Cache
Integrating a TLB with the Cache
Virtual Memory: full picture