This Unit: Virtual Memory

• The operating system (OS)
  • A super-application
  • Hardware support for an OS
• Virtual memory
  • Page tables and address translation
  • TLBs and memory hierarchy issues

Readings
• P+H
  • Virtual Memory: C.4, C.5 starting with page C-53 (Opteron), C.6

A Computer System: Hardware
• CPUs and memories
  • Connected by memory bus
• I/O peripherals: storage, input, display, network, ...
  • With separate or built-in DMA
  • Connected by system bus (which is connected to memory bus)
A Computer System: + App Software

- Application software: computer must do something

A Computer System: + OS

- Operating System (OS): virtualizes hardware for apps
  - Abstraction: provides services (e.g., threads, files, etc.)
    + Simplifies app programming model, raw hardware is nasty
  - Isolation: gives each app illusion of private CPU, memory, I/O
    + Simplifies app programming model
    + Increases hardware resource utilization

Operating System (OS) and User Apps

- Sane system development requires a split
  - Hardware itself facilitates/enforces this split

- Operating System (OS): a super-privileged process
  - Manages hardware resource allocation/revocation for all processes
  - Has direct access to resource allocation features
  - Aware of many nasty hardware details
  - Aware of other processes
  - Talks directly to input/output devices (device driver software)

- User-level apps: ignorance is bliss
  - Unaware of most nasty hardware details
  - Unaware of other apps (and OS)
  - Explicitly denied access to resource allocation features

System Calls

- Controlled transfers to/from OS

- System Call: a user-level app “function call” to OS
  - Leave description of what you want done in registers
  - SYSCALL instruction (also called TRAP or INT)
    - Can’t allow user-level apps to invoke arbitrary OS code
    - Restricted set of legal OS addresses to jump to (trap vector)
  - Processor jumps to OS using trap vector
    - Sets privileged mode
  - OS performs operation
  - OS does a “return from system call”
    - Unsets privileged mode
Interrupts

- **Exceptions**: synchronous, generated by running app
  - E.g., illegal insn, divide by zero, etc.
- **Interrupts**: asynchronous events generated externally
  - E.g., timer, I/O request/reply, etc.
- "**Interrupt**" handling: same mechanism for both
  - "Interrupts" are on-chip signals/bits
    - Either internal (e.g., timer, exceptions) or from I/O devices
  - Processor continuously monitors interrupt status, when one is high...
  - Hardware jumps to some preset address in OS code (interrupt vector)
  - Like an asynchronous, non-programmatic SYSCALL
- **Timer**: programmable on-chip interrupt
  - Initialize with some number of micro-seconds
  - Timer counts down and interrupts when reaches 0

Virtualizing Processors

- How do multiple apps (and OS) share the processors?
  - **Goal**: applications think there are an infinite # of processors

- Solution: time-share the resource
  - Trigger a context switch at a regular interval (~1ms)
    - **Pre-emptive**: app doesn't yield CPU, OS forcibly takes it
      + Stops greedy apps from starving others
  - Architected state: PC, registers
    - Save and restore them on context switches
  - Memory state?
    - **Non-architected state**: caches, predictor tables, etc.
      - Ignore or flush
  - Operating responsible to handle context switching
    - Hardware support is just a timer interrupt

Virtualizing Main Memory

- How do multiple apps (and the OS) share main memory?
  - **Goal**: each application thinks it has infinite memory

- One app may want more memory than is in the system
  - App's insn/data footprint may be larger than main memory
- Requires main memory to act like a cache
  - With disk as next level in memory hierarchy (slow)
  - Write-back, write-allocate, large blocks or "pages"
  - No notion of "program not fitting" in registers or caches (why?)

- Solution:
  - Part #1: treat memory as a "cache"
    - Store the overflowed blocks in "swap" space on disk
  - Part #2: add a level of indirection (address translation)

Virtual Memory (VM)

- Programs use **virtual addresses (VA)**
  - $0...2^{N-1}$
  - VA size also referred to as machine size
    - E.g., 32-bit (embedded) or 64-bit (server)

- Memory uses **physical addresses (PA)**
  - $0...2^M-1$ (typically M<N, especially if N=64)
  - $2^N$ is most physical memory machine supports

- VA→PA at page granularity (VP→PP)
  - By "system"
  - Mapping need not preserve contiguity
  - VP need not be mapped to any PP
  - Unmapped VPs live on disk (swap)
Virtual Memory (VM)

- **Virtual Memory (VM):**
  - Level of indirection
  - Application generated addresses are virtual addresses (VAs)
    - Each process *thinks* it has its own $2^N$ bytes of address space
  - Memory accessed using physical addresses (PAs)
  - VAs translated to PAs at some coarse granularity
  - OS controls VA to PA mapping for itself and all other processes
  - Logically: translation performed before every insn fetch, load, store
  - Physically: hardware acceleration removes translation overhead

OS controlled VA→PA mappings

PAs (physical memory)

VM is an Old Idea: Older than Caches

- Original motivation: single-program compatibility
  - IBM System 370: a family of computers with one software suite
    - Same program could run on machines with different memory sizes
      - Prior, programmers explicitly accounted for memory size
  - But also: full-associativity + software replacement
    - Memory $t_{\text{miss}}$ is high: extremely important to reduce $\%_{\text{miss}}$

<table>
<thead>
<tr>
<th>Parameter</th>
<th>I$/D$</th>
<th>L2</th>
<th>Main Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{\text{hit}}$</td>
<td>2ns</td>
<td>10ns</td>
<td>30ns</td>
</tr>
<tr>
<td>$t_{\text{miss}}$</td>
<td>10ns</td>
<td>30ns</td>
<td>10ms (10M ns)</td>
</tr>
<tr>
<td>Capacity</td>
<td>8–64KB</td>
<td>128KB–2MB</td>
<td>64MB–64GB</td>
</tr>
<tr>
<td>Block size</td>
<td>16–32B</td>
<td>32–256B</td>
<td>4+KB</td>
</tr>
<tr>
<td>Assoc./Repl.</td>
<td>1–4, NMRU</td>
<td>4–16, NMRU</td>
<td>Full, &quot;working set&quot;</td>
</tr>
</tbody>
</table>

Uses of Virtual Memory

- More recently: isolation and multi-programming
  - Each app thinks it has $2^N$ B of memory, its stack starts 0xFFFFFFFF,...
  - Apps prevented from reading/writing each other’s memory
    - Can’t even address the other program’s memory!
- **Protection**
  - Each page with a read/write/execute permission set by OS
  - Enforced by hardware
- **Inter-process communication.**
  - Map same physical pages into multiple virtual address spaces
  - Or share files via the UNIX `mmap()` call

Virtual Memory: The Basics

- Programs use virtual addresses (VA)
  - VA size (N) aka machine size (e.g., Core 2 Duo: 48-bit)
- Memory uses physical addresses (PA)
  - PA size (M) typically $M<N$, especially if $N=64$
    - $2^N$ is most physical memory machine supports
- VA→PA at page granularity (VP→PP)
  - Mapping need not preserve contiguity
  - VP need not be mapped to any PP
  - Unmapped VPs live on disk (swap) or nowhere (if not yet touched)
Address Translation

- VA → PA mapping called **address translation**
  - Split VA into *virtual page number (VPN)* & page offset (POFS)
  - Translate VPN into *physical page number (PPN)*
  - POFS is not translated
  - VA → PA = [VPN, POFS] → [PPN, POFS]

- **Example above**
  - 64KB pages → 16-bit POFS
  - 32-bit machine → 32-bit VA → 16-bit VPN
  - Maximum 256MB memory → 28-bit PA → 12-bit PPN

Page Table Size

- **How big is a page table on the following machine?**
  - 32-bit machine
  - 4B page table entries (PTEs)
  - 4KB pages
  - 32-bit machine → 32-bit VA → 4GB virtual memory
  - 4GB virtual memory / 4KB page size → 1M VPs
  - 1M VPs * 4B PTE → 4MB

- **How big would the page table be with 64KB pages?**
- **How big would it be for a 64-bit machine?**

- **Page tables can get big**
  - There are ways of making them smaller

Address Translation Mechanics

- **How are addresses translated?**
  - In software (for now) but with hardware acceleration (a little later)
- **Each process allocated a page table (PT)**
  - **Software data structure constructed by OS**
  - Maps VPs to PPs or to disk (swap) addresses
  - VP entries empty if page never referenced
  - Translation is table lookup

```c
struct {
    int ppn;
    int is_valid, is_dirty, is_swapped;
} PTE;
struct PTE page_table[NUM_VIRTUAL_PAGES];

int translate(int vpn) {
    if (page_table[vpn].is_valid)
        return page_table[vpn].ppn;
}
```

Multi-Level Page Table (PT)

- **One way: multi-level page tables**
  - Tree of page tables
  - Lowest-level tables hold PTEs
  - Upper-level tables hold pointers to lower-level tables
  - Different parts of VPN used to index different levels

- **Example: two-level page table for machine on last slide**
  - Compute number of pages needed for lowest-level (PTEs)
    - 4KB pages / 4B PTEs → 1K PTEs/page
    - 1M PTEs / (1K PTEs/page) → 1K pages
  - Compute number of pages needed for upper-level (pointers)
    - 1K lowest-level pages → 1K pointers
    - 1K pointers * 32-bit VA → 4KB → 1 upper level page
Multi-Level Page Table (PT)

- **20-bit VPN**
  - Upper 10 bits index 1st-level table
  - Lower 10 bits index 2nd-level table

```c
struct {
    int ppn;
    int is_valid, is_dirty, is_swapped;
} PTE;
struct { struct PTE ptes[1024]; } L2PT;
struct L2PT *page_table[1024];

int translate(int vpn) {
    index1 = (vpn >> 10); // upper 10 bits
    index2 = (vpn & 0x3ff); // lower 10 bits
    struct L2PT *l2pt = page_table[index1];
    if (l2pt != NULL &&
        l2pt->ptes[index2].is_valid)
        return l2pt->ptes[index2].ppn;
}
```

Page-Level Protection

- **Page-level protection**
  - Piggy-back page-table mechanism
  - Map VPN to PPN + Read/Write/Execute permission bits
  - Attempt to execute data, to write read-only data?
    - Exception → OS terminates program
  - Useful (for OS itself actually)

```c
struct {
    int ppn;
    int is_valid, is_dirty, is_swapped, permissions;
} PTE;
struct PTE page_table[NUM_VIRTUAL_PAGES];

int translate(int vpn, int action) {
    if (page_table[vpn].is_valid &&
        !(page_table[vpn].permissions & action)) kill;
    ...
}
```

Address Translation Mechanics II

- **Conceptually**
  - Translate VA to PA before every cache access
  - Walk the page table before every load/store/insn-fetch
    - Would be terribly inefficient (even in hardware)

- **In reality**
  - **Translation Lookaside Buffer (TLB):** cache translations
    - Only walk page table on TLB miss

- **Hardware truisms**
  - Functionality problem? Add indirection (e.g., VM)
  - Performance problem? Add cache (e.g., TLB)
Translation Lookaside Buffer

- **Translation lookaside buffer (TLB)**
  - Small cache: 16–64 entries
  - Associative (4+ way or fully associative)
  - Exploits temporal locality in page table
  - What if an entry isn’t found in the TLB?
    - Invoke TLB miss handler

Serial TLB & Cache Access

- **“Physical” caches**
  - Indexed and tagged by **physical addresses**
    + Natural, "lazy" sharing of caches between apps/OS
  - VM ensures isolation (via **physical addresses**)
  - No need to do anything on context switches
  - Multi-threading works too
    + Cached inter-process communication works
      - Single copy indexed by physical address
      - Slow: adds at least one cycle to \( t_{hit} \)
  - Note: **TLBs are by definition “virtual”**
    - Indexed and tagged by **virtual addresses**
    - Flush across context switches
    - Or extend with process identifier tags (x86)

Parallel TLB & Cache Access

- **Two ways to look at VA**
  - Cache: tag+index+offset
  - TLB: **VPN**+page offset

- **Parallel cache/TLB**...
  - If address translation doesn’t change index
  - That is, VPN/index must not overlap
  - TLB hit/miss
  - cache hit/miss

- **What about parallel access?**
  - Only if...
    - (cache size) / (associativity) \( \leq \) page size
    - Index bits same in virt. and physical addresses!
  - Access TLB in parallel with cache
    - Cache access needs tag only at very end
      + Fast: no additional \( t_{visa} \) cycles
      + No context-switching/aliasing problems
      + Dominant organization used today
  - Example: Core 2, 4KB pages, 32KB, 8-way SA L1 data cache
  - Implication: associativity allows bigger caches
**TLB Organization**

- **Like caches**: TLBs also have ABCs
  - Capacity
  - Associativity (At least 4-way associative, fully-associative common)
  - What does it mean for a TLB to have a block size of two?
    - Two consecutive VPs share a single tag
  - **Like caches**: there can be L2 TLBs

- Example: AMD Opteron
  - 32-entry fully-assoc. TLBs, 512-entry 4-way L2 TLB (insn & data)
  - 4KB pages, 48-bit virtual addresses, four-level page table

- **Rule of thumb**: TLB should “cover” L2 contents
  - In other words: (#PTEs in TLB) * page size ≥ L2 size
  - Why? Consider relative miss latency in each...

---

**TLB Misses**

- **TLB miss**: translation not in TLB, but in page table
  - Two ways to “fill” it, both relatively fast

- **Software-managed TLB**: e.g., Alpha, MIPS, ARM
  - Short (~10 insn) OS routine walks page table, updates TLB
  - Keeps page table format flexible
    - Latency: one or two memory accesses + OS call (pipeline flush)

- **Hardware-managed TLB**: e.g., x86
  - Page table root in hardware register, hardware “walks” table
  - Latency: saves cost of OS call (avoids pipeline flush)
  - Page table format is hard-coded

- Trend is towards hardware TLB miss handler

---

**Page Faults**

- **Page fault**: PTE not in TLB or page table
  - → page not in memory
  - Or no valid mapping → segmentation fault
  - Starts out as a TLB miss, detected by OS/hardware handler

- **OS software routine**:
  - Choose a physical page to replace
    - "**Working set**": refined LRU, tracks active page usage
    - If dirty, write to disk
    - Read missing page from disk
      - Takes so long (~10ms), OS schedules another task

- Requires yet another data structure: **frame map** (why?)
  - Treat like a normal TLB miss from here

---

**Summary**

- OS virtualizes memory and I/O devices

- **Virtual memory**
  - “infinite” memory, isolation, protection, inter-process communication
  - Page tables
  - Translation buffers
    - Parallel vs serial access, interaction with caching
  - Page faults