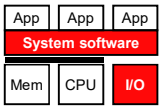
CSE 560
Computer Systems Architecture

Virtual Memory

1

1

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

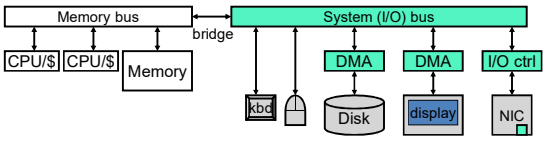
2

2

A Computer System: Hardware

CPUs and memories
- Connected by memory bus

I/O peripherals: storage, input, display, network, …
(NIC = Network Interface Controller)
- With separate or built-in DMA (direct memory access)
- Connected by **system bus** (which is connected to memory bus)

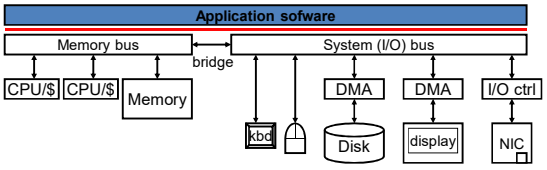


3

3

A Computer System: + App Software

- **Application software**: computer must do something

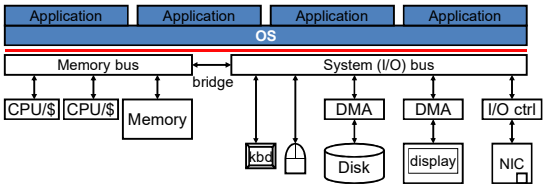


4

4

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



5

5

Operating System (OS) and User Apps

- Sane system development requires a split

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

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

6

6

System Calls

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)
  - User-level apps not allowed to invoke arbitrary OS code
  - Restricted set of legal OS addresses to jump to (**trap vector**)

1. Processor jumps to OS via trap vector (begin privileged mode)
2. OS performs operation
3. OS does a "return from system call" (end privileged mode)

Interrupts

Exceptions: synchronous, generated by running app
- *E.g.*, illegal instruction, divide by zero, *etc.*

Interrupts: asynchronous events generated externally
- *E.g.*, timer, I/O request/reply, *etc.*

Timer: programmable on-chip interrupt
- Initialize with some number of micro-seconds
- Timer counts down and interrupts when reaches 0

"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 true…
- HW jumps to some preset address in OS code (interrupt vector)
- Like an asynchronous, non-programmatic SYSCALL

Virtualizing Processors

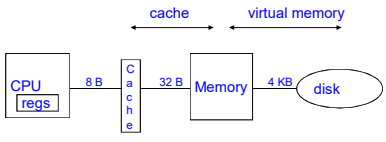
How do multiple apps (and OS) share the processors?
Goal: applications think there are an infinite # of processors

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 System responsible for handling context switching
  - Hardware support is just a timer interrupt

Motivations for Virtual Memory

- **Use Physical DRAM as a Cache for the Disk**
  - Address space of a process can exceed physical memory size
  - Sum of address spaces of multiple processes can exceed physical memory
- **Simplify Memory Management**
  - Multiple processes resident in main memory
    - Each process with its own address space
  - Only "active" code and data is actually in memory
    - Allocate more memory to process as needed
- **Provide Protection**
  - One process can't interfere with another
    - because they operate in different address spaces
  - User process cannot access privileged information
    - different sections of address spaces have different permissions

Levels in Memory Hierarchy



	Register	Cache	Memory	Disk Memory
size:	32 B	32 KB-4MB	1024 MB	100 GB
speed:	1 ns	2 ns	30 ns	8 ms
$/Mbyte:		$125/MB	$0.20/MB	$0.001/MB
line size:	8 B	32 B	4 KB	

larger, slower, cheaper →

Virtualizing Main Memory

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

App's insn/data footprint > 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"

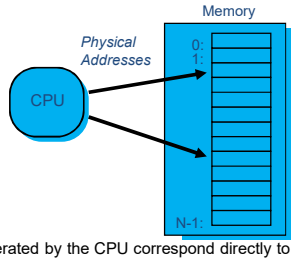
Solution:
- Part #1: treat memory as a "cache"
- Part #2: add a level of indirection (address translation)

Parameter	I$/D$	L2	Main Memory
t_{hit}	2ns	10ns	**30ns**
t_{miss}	**10ns**	**30ns**	**10ms (10M ns)**
Capacity	8–64KB	128KB–2MB	**64MB–64GB**
Block size	16–32B	32–256B	**4+KB**
Assoc./Repl.	1–4, NMRU	4–16, NMRU	**Full, "working set"**

A System with Physical Memory Only

Examples:

- most Cray machines, early PCs, many embedded systems, etc.



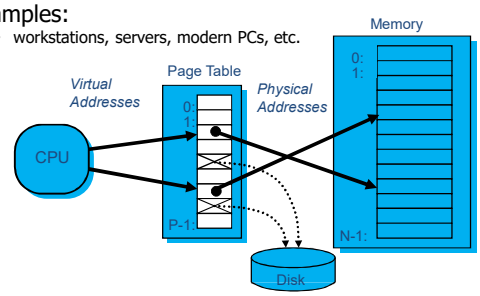
Addresses generated by the CPU correspond directly to bytes in physical memory

13

A System with Virtual Memory

Examples:

- workstations, servers, modern PCs, etc.

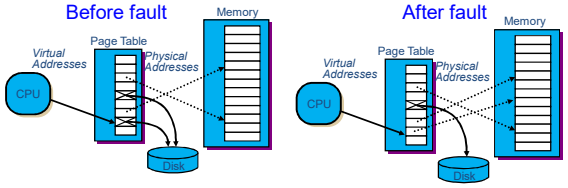


Address Translation: Hardware converts virtual addresses to physical addresses via OS-managed lookup table (page table)

14

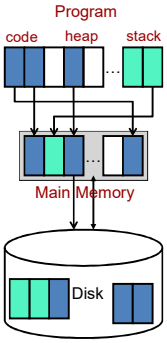
Page Faults (like "Cache Misses")

- What if an object is on disk rather than in memory?
  - Page table entry indicates virtual address not in memory
  - OS exception handler invoked to move data from disk into memory
    - current process suspends, others can resume
    - OS has full control over placement, etc.



15

Virtual Memory (VM)



- Programs use **virtual addresses (VA)**
  - $0 \ldots 2^N - 1$
  - VA size also referred to as machine size
  - *E.g.*, 32-bit (embedded) or 64-bit (server)
- Memory uses **physical addresses (PA)**
  - $0 \ldots 2^M - 1$ (typically M<N, especially if N=64)
  - 2^M is most physical memory machine supports
- VA→PA at **page** granularity (VP→PP)
  - By "system" (OS + HW)
  - Mapping need not preserve contiguity
  - VP need not be mapped to any PP
  - Unmapped VPs live on disk (swap)

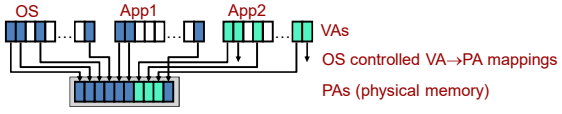
16

16

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

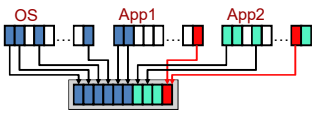


17

17

Uses of Virtual Memory

- **Isolation** and **Multi-programming (Memory Management)**
  - Each app thinks it has 2^N B of memory that starts @ 0
  - Apps can't read/write each other's memory
    - Can't even address the other program's memory!
- **Protection**
  - Each page has 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

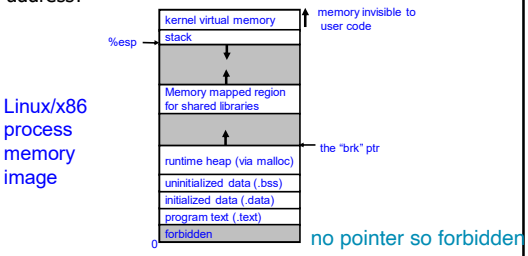


18

18

Memory Management

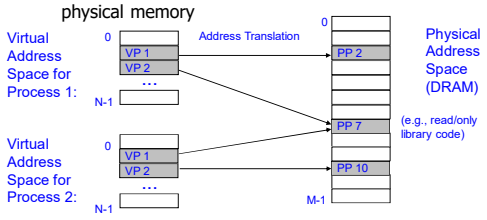
- Multiple processes can reside in physical memory.
- How do we resolve address conflicts?
  - what if two processes access something at the same address?



19

Solution: Separate Virt. Addr. Spaces

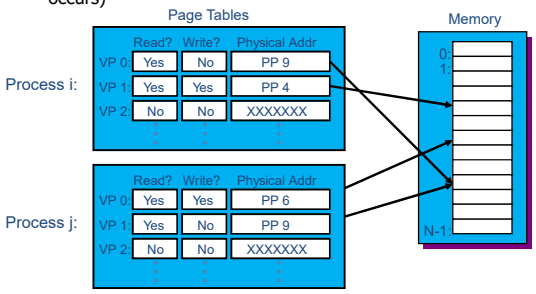
- Virtual and physical address spaces divided into equal-sized blocks
  - blocks are called "pages" (both virtual and physical)
- Each process has its own virtual address space
  - operating system controls how virtual pages as assigned to physical memory



20

Protection

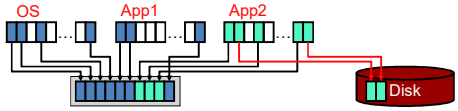
- Page table entry contains access rights information
  - hardware enforces this protection (trap into OS if violation occurs)



21

Virtual Memory: The Basics

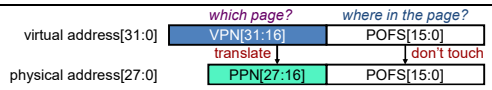
- Programs use **virtual addresses (VA)**
  - VA size (N) aka machine size (e.g., Core 2 Duo: 48-bit)
- Memory uses **physical addresses (PA)**
  - PA size (M) typically M<N, especially if N=64
  - 2^M is most physical memory machine supports
- VA→PA at **page** granularity (VP→PP)
  - Mapping need not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



22

22

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 per page → 16-bit POFS (2^{16}=64K)
- 32-bit machine → 32-bit VA → 16-bit VPN 32-16=16
- Max. 256MB memory → 28-bit PA → 12-bit PPN 28-16=12 (2^{28}=256M)

23

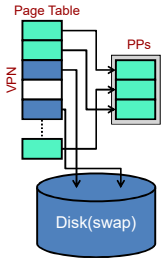
23

Address Translation Mechanics I

- How are addresses translated?
  - In sw (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

```
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;
}
```



24

24

Page Table Size

How big is a page table on the following machine?

Given:

- 32-bit machine
- 4KB per page
- 4B page table entries (PTEs) (see struct definition, prev slide)

Can determine:

- 32-bit machine → 32-bit VA → 4GB virtual memory (2^{32}=4G)
- 4GB virtual memory / 4KB page size → 1M VPs
- Each VP needs a PTE: 1M VPs → 1M PTEs
- 1M PTEs x 4B-per-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* (see next slide)

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

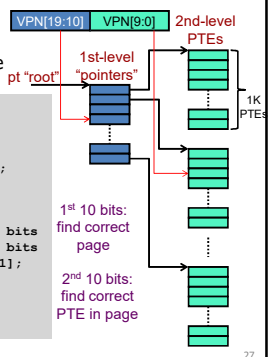
Example: two-level page table for machine on last slide

- Compute number of pages needed for lowest-level (PTEs)
  - 4KB page size / 4B-per-PTE → can hold 1K PTEs per page
  - 1M PTEs / (1K PTEs/page) → 1K pages
- Compute # of pages needed for upper-level (pointers)
  - 1K lowest-level pages → 1K pointers
  - 1K pointers x 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

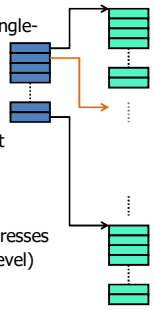



```
struct {
   int ppn;
   int is_valid, is_dirty, is_swapped;
} PTE;
struct { struct PTE ptes[1024]; } L2PT;
struct L2PT *page_table[1024];

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;
}
```


Multi-Level Page Table (PT)

- Have we saved any space?
  - Isn't total size of 2nd level tables same as single-level table (*i.e.*, 4MB)?
  - Yes, but…
- Large virtual address regions unused
  - Corresponding 2nd-level tables need not exist
  - Corresponding 1st-level pointers are null
- Example: 2MB code, 64KB stack, 16MB heap
  - Each 2nd-level table maps 4MB of virtual addresses
  - 1 for code, 1 for stack, 4 for heap, (+1 1st-level)
  - 7 total pages = 28KB (much less than 4MB)



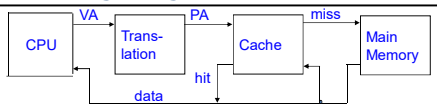
Page-Level Protection

- **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)

```
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;
   …
}
```


Integrating VM and Cache



- Most Caches "Physically Addressed"
  - Accessed by physical addresses
  - Allows multiple processes to have blocks in cache at same time
  - Allows multiple processes to share pages
  - Cache doesn't need to be concerned with protection issues
    - Access rights checked as part of address translation
- Perform Address Translation Before Cache Lookup
  - But this could involve a memory access itself (of the PTE)
  - Of course, page table entries can also become cached

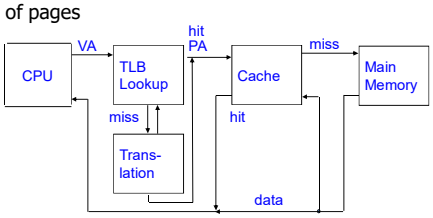
Address Translation Mechanics II

- 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)

31

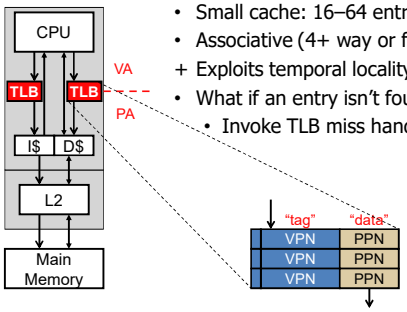
Speeding up Translation with a TLB

- "Translation Lookaside Buffer" (TLB)
  - Small hw cache in MMU (memory management unit)
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages



32

Translation Lookaside Buffer

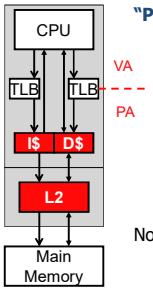


Translation lookaside buffer (TLB)

- 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

33

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)

34

Parallel TLB & Cache Access

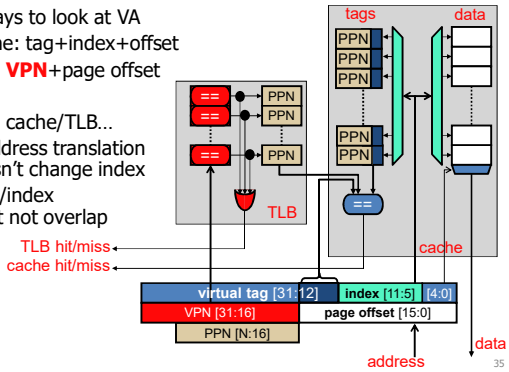
Two ways to look at VA

- Cache: tag+index+offset
- TLB: **VPN**+page offset

Parallel cache/TLB…

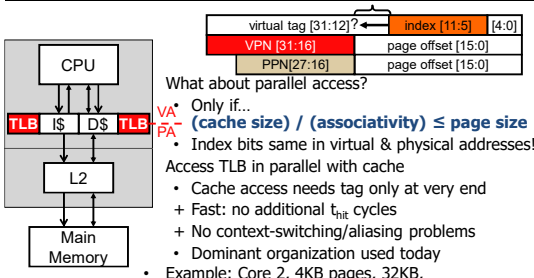
- If address translation doesn't change index

→ VPN/index must not overlap



35

Parallel TLB & Cache Access



What about parallel access?

- Only if…
- **(cache size) / (associativity) ≤ page size**
- Index bits same in virtual & physical addresses!

Access TLB in parallel with cache

- Cache access needs tag only at very end
- \+ Fast: no additional t_{hit} 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*

36

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) x 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
- 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