Virtual Memory
Administrivia...

- Homework 8 will be released tomorrow
Protection, Translation, Paging

- Supervisor mode alone is not sufficient to fully isolate applications from each other or from the OS
  - Application could overwrite another application’s memory.
- Typically programs start at some fixed address, e.g. 0x8FFFFFFF
  - How can 100’s of programs all share memory at location 0x8FFFFFFF?
- Also, may want to address more memory than we actually have (e.g., for sparse data structures)
- Solution: Virtual Memory
  - Gives each process the illusion of a full memory address space that it has completely for itself
“Bare” 5-Stage Pipeline

• In a bare machine, the only kind of address is a physical address
What do we need Virtual Memory for?

Reason 1: Adding Disks to Hierarchy

- Need to devise a mechanism to “connect” memory and disk in the memory hierarchy

- Disk
  - Slow
  - But huge
  - How could we make use of its capacity (when running low on DRAM)?
What do we need Virtual Memory for?
Reason 2: Simplifying Memory for Apps

- Processes should see the straightforward memory layout we saw earlier ->
- User-space applications should think they own all of memory
- So we give them a **virtual** view of memory

![Memory Layout Diagram]

- Code
- Static Data
- Stack
- Heap

Hex values:
- Code: ~ 7FFF FFFF
- Static Data: ~ 0000 0000
What do we need Virtual Memory for?
Reason 3: Protection Between Processes

• With a bare system, addresses issued with loads/stores are real physical addresses

• This means any process can issue any address, therefore can access any part of memory, even areas which it doesn’t own
  • Ex: The OS data structures

• We should send all addresses through a mechanism that the OS controls, before they make it out to DRAM - a translation mechanism
  • Can check that process has permission to access a particular part of memory
Address Spaces

- Address space = set of addresses for all available memory locations
- Now, two kinds of memory addresses:
  - *Virtual* Address Space
    - Set of addresses that the user program knows about
  - *Physical* Address Space
    - Set of addresses that map to actual physical locations in memory
    - Hidden from user applications
- Memory manager maps between these two address spaces
Aside: Blocks vs. Pages

- In caches, we dealt with individual **blocks**
  - Usually ~64-128B on modern systems
  - We “divide” memory into a set of blocks
- In VM, we deal with individual **pages**
  - Usually ~4 KB on modern systems
  - Now, we’ll “divide” memory into a set of pages
- Common point of confusion: Bytes, Words, Blocks, Pages are all just different ways of looking at memory!
Bytes, Words, Blocks, Pages

Ex: 16 KiB DRAM, 4 KiB Pages (for VM), 128 B blocks (for caches), 4 B words (for lw/sw)

Can think of memory as:
- 4 Pages
  OR
- 128 Blocks
  OR
- 4096 Words

Can think of a page as:
- 32 Blocks
  OR
- 1024 Words
Address Translation

- So, what do we want to achieve at the hardware level?
  - Take a Virtual Address, that points to a spot in the Virtual Address Space of a particular program, and map it to a Physical Address, which points to a physical spot in DRAM of the whole machine.
Address Translation

Virtual Address

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Translation</td>
<td></td>
</tr>
<tr>
<td>Copy Bits</td>
<td></td>
</tr>
</tbody>
</table>

Physical Address

<table>
<thead>
<tr>
<th>Physical Page Number</th>
<th>Offset</th>
</tr>
</thead>
</table>

Rest of this lecture: How do we implement this?
Paged Memory Systems

- Processor-generated address can be split into:

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Offset</th>
</tr>
</thead>
</table>

- A page table contains the physical address of the base of each page

Page tables make it possible to store the pages of a process non-contiguously.
Private (Virtual) Address Space per Process

- Each process has a page table
- Page table contains an entry for each process page
- Physical Memory acts like a “cache” of pages for currently running programs.
  Not recently used pages are stored in secondary memory, e.g. disk (in “swap partition”)
Where Should Page Tables Reside?

• Space required by the page tables (PT) is proportional to the address space, number of processes, ...
  \[ \Rightarrow \text{Too large to keep in registers inside CPU} \]

• Idea: Keep page tables in the main memory
  • Needs one reference to retrieve the page physical address and another to access the data word
  \[ \Rightarrow \text{doubles the number of memory references! (but we can fix this using something we already know about...)} \]
Page Tables in Physical Memory

- Process 1 Virtual Address Space
- Process 2 Virtual Address Space
- Physical Memory
Linear (simple) Page Table

- Page Table Entry (PTE) contains:
  - 1 bit to indicate if page exists
  - And either PPN or DPN:
    - PPN (physical page number) for a memory-resident page
    - DPN (disk page number) for a page on the disk
  - Status bits for protection and usage (read, write, exec)
- OS sets the Page Table Base Register whenever active user process changes
Suppose an instruction references a memory page that isn’t in DRAM?

- We get a exception of type “page fault”
- Page fault handler (yet another function of the interrupt/trap handler) does the following:
  - If no unused page is available, a page currently in DRAM is selected to be replaced
    - The replaced page is written (back) to disk, page table entry that maps that VPN→PPN is marked with DPN
  - If virtual page doesn’t yet exist, assign it an unused page in DRAM
  - If page exists but was on disk…
    - Initiate transfer of the page contents we’re requesting from disk to DRAM, assigning to an unused DRAM page
  - Page table entry of the (virtual) page we’re requesting is updated with a (now) valid PPN
- Following the page fault, re-execute the instruction
Size of Linear Page Table

With 32-bit memory addresses, 4-KB pages:
\[ 2^{32} / 2^{12} = 2^{20} \text{ virtual pages per user, assume 4-Byte PTEs}, \]
\[ 2^{20} \text{ PTEs, } 2^{22} \text{ bytes required: 4 MB page table per process!} \]

Larger pages?
- Internal fragmentation (Not all memory in page gets used)
- Larger page fault penalty (more time to read from disk)

What about 64-bit virtual address space???
- Even 1MB pages would require \( 2^{44} \) 8-Byte PTEs (35 TB!)

What is the “saving grace”? Most processes only use a set of high address (stack), and a set of low address (instructions, heap)
Hierarchical Page Table – exploits sparsity of virtual address space use

Virtual Address

Root of the Current Page Table

(Processor Register)

Level 1 Page Table

Level 2 Page Tables

Physical Memory

Data Pages

page in primary memory
page in secondary memory
PTE of a nonexistent page

31 22 21 12 11 0

p1 p2 offset

10-bit L1 index

10-bit L2 index
Address Translation & Protection

- Every instruction and data access needs address translation and protection checks

A good VM design needs to be fast (~ one cycle) and space efficient
Translation Lookaside Buffers (TLB)

Address translation is very expensive!
In a two-level page table, each reference becomes several memory accesses

Solution: **Cache some translations in TLB**

- TLB hit \( \Rightarrow \) **Single-Cycle Translation**
- TLB miss \( \Rightarrow \) **Page-Table Walk to refill**

![Diagram](image)

(VPN = virtual page number)

(PPN = physical page number)
TLB Designs

- Typically 32-128 entries, sometimes fully associative
  - Each entry maps a large page, hence less spatial locality across pages => more likely that two entries conflict
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
  - Larger systems sometimes have multi-level (L1 and L2) TLBs
- Random or FIFO replacement policy
- “TLB Reach”: Size of largest virtual address space that can be simultaneously mapped by TLB

Example: 64 TLB entries, 4KB pages, one page per entry

\[ \text{TLB Reach} = \text{?} \]
TLB Designs

- Typically 32-128 entries, sometimes fully associative
  - Each entry maps a large page, hence less spatial locality across pages => more likely that two entries conflict
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
  - Larger systems sometimes have multi-level (L1 and L2) TLBs
- Random or FIFO replacement policy
- “TLB Reach”: Size of largest virtual address space that can be simultaneously mapped by TLB

Example: 64 TLB entries, 4KB pages, one page per entry

TLB Reach = 64 Entries * 4KB/Entry = 256 KB
VM-related exceptions in pipeline

• Handling a TLB miss needs a hardware or software mechanism to refill TLB
  • Usually done in hardware now (x86, typically RISC-V)
• Handling a page fault (e.g., page is on disk) needs a *precise* trap so software handler can easily resume after retrieving page
• Handling protection violation may abort process
Using the TLB...

- **Option 1: MIPS style**
  - The TLB is the only translation in the hardware
  - Whenever you get a TLB miss, you jump to the page fault handler

- **Option 2: x86**
  - The page table has a defined structure
  - In the event of a TLB miss, the **hardware** walks the page table
    - Only if the page is unavailable do you jump to the page fault handler

- **RISC-V prefers #2,**
  but you can build a compliant RISC-V with #1
  - You just need to include the page-walking trap handler.
  - After the trap handler updates the page table for a process needs to call a special instruction to flush the TLB
Page-Based Virtual-Memory Machine
(Hardware Page-Table Walk)

- Assumes page tables held in untranslated physical memory

![Diagram showing the flow of data and control signals in a page-based virtual-memory machine, including components like Instruction TLB, Instruction Cache, Decode, TLB, Memory Controller, and Main Memory (DRAM).]
Address Translation: putting it all together

Virtual Address

TLB Lookup

Page Table Walk

Update TLB

Protection Check

Page Fault (OS loads page)

Protection Fault

Physical Address (to cache)

SEGFAULT

Re-execute instruction

Page is not in memory

Page is in memory

Hardware

Hardware or software

Software

Hit

Miss

Denied

Permitted
Summary: Virtual Memory Systems

*Illusion of a large, private, uniform store*

**Protection & Privacy**
several users, each with their private address space and one or more shared address spaces
page table = name space

**Demand Paging**
Provides the ability to run programs larger than the primary memory
Hides differences in machine configurations

*The price is address translation on each memory reference*
Virtual Memory Paging As A Cache...

- Can think of virtual memory’s demand paging aspect as a simple cache
- Your program is given the illusion of as much RAM as it wants...
  - But this breaks things unless it doesn’t really want all of it!
- Idea: Virtual memory can copy "pages" between RAM and disk
  - The main memory thus acts as a cache for the disk...
Virtual Memory’s “Cache” Properties

- Associativity: Effectively fully associative
- Replacement policy: Approximate LRU
- Block size: 4kB or more
  - The size of a page!
- Miss penalty...
  - Latency to get a block from disk: 1ms or so for an SSD... Or put in clock terms, a 1 GHz clock -> 1,000,000 clock cycles!
  - Or if you have a spinning disk: 10ms or so...
    So \(10,000,000\) clock cycles!
- We really don’t want misses
  - Worth it to use sophisticated replacement policy
Implications

• As long as you don't really use full capacity, Virtual Memory is great...
  • Basically as long as your working set fits in physical memory, virtual memory is great at handling a little extra...

• But as soon as your working set exceeds physical memory, your performance falls off a cliff
  • The system starts **thrashing**: Repeatedly needing to copy data to and from disk…
  • Similar to **thrashing the cache** when your working set exceeds cache capacity (but the cliff here is much steeper!)
Aside … Why are Disks So Slow?

- 10,000 rpm (revolutions per minute)
- 6 ms per revolution
- Average random access time: 3 ms
What About SSD?

- Made with transistors - same technology as Flash memory
- Nothing mechanical that turns
- Like “Ginormous” DRAM
  - Except “nonvolatile” - holds contents when power is off
- Fast access to all locations, regardless of address
- Still much slower than register, caches, DRAM
  - Read/write blocks, not bytes
- Plus unusual requirements:
  - You can't erase single bits (set to 0), but only entire blocks
  - So to update a block, you copy everything to a new location that was erased with all 0
VM Tricks:
Copy-On-Write Duplication

• You split a process and now have two copies
  • The classic `fork()` operation

• Just copy the page table and registers...
  • And then mark both the original and copy's memory as `read-only`!

• Every-time either process wants to write a page...
  • It traps to the protection fault handler...
  • Which copies the page
  • And then updates both page-tables to allow writing
  • And then returns

• Now you only copy memory when you first write to it
More VM Tricks: Shared Memory Communication

- Program A and B exist in their own separate little worlds
  - And of course, they are not supposed to inadvertently interfere with each other...
- But what if they want to share information?
  - We could just write and read to disk...
    But that is slow...
- Hey, we have this cool virtual memory system...
  - Who says that the same physical page can't be pointed to multiple times!
  - Can save us memory when multiple processes want to use same dynamically linked library!
Even More Memory Tricks: Memory Mapped files...

- Conventional file reading is slow...
  - You're reading things a line at a time...
  - And keep asking the OS for data...
- Can be way more overhead than you need...
- What you commonly want:
  - "Just load the entire file into a contiguous block of memory"
- So why not get the VM system to do that?
Memory Mapped I/O

- Program traps to the operating system for the `mmap` sys call
  - "Hey, I just want to read this file"

- OS sets the page table entries to point to disk
  - Just into the particular file instead of a "swap file"

- OS returns a pointer to the start of that virtual memory

- And the program just happily reads away...

- When the program actually reads...
  - Generates a page fault exception to the OS which loads the page into memory
  - Which then retries the faulting instruction and its like the file is just in memory

- Most page tables also support a "dirty" bit...
  - So if we need to swap out that page, if its dirty write it out...
    But if not, just discard it... We can reload again from disk!