A Virtual Lecture on Virtual Memory
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)?
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
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

- stack
- heap
- static data
- code

~ 7FFF FFFF\textsubscript{hex}

~ 0000 0000\textsubscript{hex}
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

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

Can think of a page as:
- 32 Blocks
- 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

The rest of the lecture is all about implementing
Paged Memory Systems

- Processor-generated address can be split into:

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

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address Space of Process #1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Table of Process #1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

*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
Administrivia

• VM Homework Out

• New Slip-Day Policy
  • You now have 6, but only 3 can be applied to any project part
    • EG, 2A and 2B have separate slip day quotas
  • Will be allocated in the optimal fashion thanks to Stephan's Total Course Points autograder
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:

$\Rightarrow \frac{2^{32}}{2^{12}} = 2^{20}$ virtual pages per user, assuming 4-Byte PTEs,

$\Rightarrow 2^{20}$ PTEs, i.e., 4 MB page table per user!

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 sparcity 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
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 of TLB and address translation](image)
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 = ________________________________?
VM-related exceptions in pipeline

- Handling a TLB miss needs a hardware or software mechanism to refill TLB
  - usually done in hardware now
- 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
Address Translation: putting it all together

Virtual Address

TLB Lookup

- hit
- miss

Page Table Walk

- Page Fault (OS loads page)
- Update TLB

Protection Check

- denied
- permitted

Protection Fault

Physical Address (to cache)

SEGFAULT

Re-execute instruction

\[\text{virtual address} \rightarrow \text{TLB Lookup} \rightarrow \text{Page Table Walk} \rightarrow \text{Protection Check} \rightarrow \text{Protection Fault} \rightarrow \text{SEGFAULT} \rightarrow \text{Re-execute instruction}\]
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*
Stupid 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 Stupid 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 do?
  - 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!
Use That!

- **man shm_overview**
  - Overview of POSIX shared memory

- Allows two processes to share a single block of memory
  - Each one gets a pointer to it in their own virtual address space
    - Note, their virtual addresses may be different!

- Once this is set up, communication doesn't involve the OS anymore
  - One process can just write to the memory and it shows up with the other process
Even More Stupid 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...
- That *sucks*: 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!
When VM Goes Wrong: Rowhammer & Meltdown

- If an attacking process can break memory isolation...
  - Either for writing or (sometimes) just for reading
- We effectively lose all our security!
  - As we have lost the isolation guarantees
- Both Rowhammer and Meltdown are exploits designed to bypass our protections
How Meltdown Works...

• Remember how we said precise exceptions are **hard**?
• x86 actually provides two page table hardware pointers
  • One for the current user program, one for supervisor mode
  • Allows the OS to have virtual memory for the interrupt handler and other things
• Concept behind meltdown:
  • x86 allows "load whatever that memory location points to + base register"
• Do a bunch of loops that are always taken
  • Now the CPU will predict that the next time this loop is taken...
• Now do a load of memory you aren't supposed to read
  • CPU guesses branch will taken, so is just going to do it speculatively. Only when it finally writes to a register will the exception be checked
• Now have the results of that load do a load to memory you are supposed to read
  • But dependent on what was in the memory you weren't supposed to read
• Now CPU finds that branch wasn't taken after all
  • And so nothing happens, neither the illegal load nor the "load not taken"
But Something *Did* Happen!

- The final "load not taken" got taken!
  - So it will be cached
- And that load was dependent on the illegal load
- So we can discover which "load not taken" got actually taken!
  - Allowing us to read memory we aren't supposed to!
- Fix involves the OS flushing the TLB and presenting a dummy OS page-table when returning to a user process
  - Greatly increasing the cost of a context switch or interrupt
The Ultimate Page-Table Trick: Rowhammer

• An unspeakably cool security vulnerability…
• DRAM (unless you pay for error correcting (ECC) memory) is actually unreliable
  • Can repeatedly read/write the same location ("hammer the row" and eventually cause an error in some physically distinct memory location)
• Can tell the OS "I want to map this same block of memory at multiple addresses in my process…"
  • Which creates additional page table entries, lots of them. Lots and lots of them. Lots and lots and lots and lots and lots of them...
• Enter Rowhammer
  • It seems all vulnerabilities get named now, but this one is cool enough to deserve a name!
  • Touches on virtual memory, hardware failures, and breaks security
How RowHammer Works

• Step 1: Allocate a single page of memory
• Step 2: Make the OS make a gazillion page-table entries pointing to the same page
• Step 3: Hammer the DRAM until one of those entries gets corrupted
  • Now causes that memory page to point to a set of page table entries instead
• Step 4: Profit
  • Well, the ability to read and write to any physical address in the system, same difference
Bonus Content...

Reading the Spec...

- Lets read the spec together for how RISC-V 32b does Virtual Memory in gory, gory detail...