LAST TIME

- Extended virtual memory concept to be a cache of memory stored on disk
  - DRAM becomes L4 cache of data stored on L5 disk
- Extend page table entries with more details
  - Entries have a valid (x86-64 “present”) flag specifying if the page is in memory, and if not, where it resides
  - Also permission flags, e.g. “read/write,” “supervisor”
- Requires hardware and software support:
  - CPU performs address translation in hardware to make it as fast as possible
  - CPU raises page fault and general protection fault exceptions when it requires the kernel’s intervention
  - Operating system handles situations where pages must be moved into and out of memory
LAST TIME (2)

- Can now implement many useful features!
- Previously could:
  - Isolate address spaces of different processes
  - Perform fast context-switches by changing the page table that the MMU uses
  - Share memory regions between processes, such as shared libraries, kernel code, working areas
- Now we can also:
  - Memory-map disk files into virtual memory, to load programs, and to perform fast and easy IO
  - Set permissions on memory pages to make some pages read-only, or inaccessible by user code
x86-64 Virtual Memory Support

- Intel64 family processors provide hardware support for virtual memory
- Virtual and physical address spaces are 48 bits
  - Can address up to 256TiB of memory
  - (Eventually will be extended to 64-bit address space)
- Pages are 4KiB in size \(2^{12} = 4096\)
  - Pages are identified by topmost 36 bits in address
  - (Processor can also use 2MiB pages)
- Intel64 family processors implement a four-level page table hierarchy
  - 12 bits for offset within page
  - Remaining 36 bits are divided among the four levels; 9 bits to index into each level
x86-64 Virtual Memory Support (2)

- x86-64 page-table hierarchy level names:
  - Level 1 = Page Global Directory
  - Level 2 = Page Upper Directory
  - Level 3 = Page Middle Directory
  - Level 4 = Page Table

- Each level uses 9 bits of virtual address to index into the next level
  - \(2^9 = 512\) entries per page

- Each page table entry is 64 bits
  - 8 bytes \(\times 512\) entries in page table node = 4096 bytes (one page)
x86-64 Virtual Memory Support (3)

- 48-bit virtual address translation:
  - Bottom 12 bits are virtual page offset – directly brought over into physical page offset
  - Remaining 36 bits of virtual address are broken into four 9-bit indexes
  - Used to traverse the sparse page-table structure
**x86-64 Virtual Memory Support (4)**

- Each process has its own page directory
  - Each process has its own virtual address space, isolated from all other processes
  - Page directory also maps some kernel code and shared library code into the process’ address space
**x86-64 Virtual Memory Support (5)**

- Current page directory is specified by the Page Directory Base Register
  - On x86-64, this is `%cr3`, or Control Register 3
  - *Only the kernel can change this control register!*

```
Current page directory is specified by the Page Directory Base Register

- On x86-64, this is `%cr3`, or Control Register 3
- *Only the kernel can change this control register!*
```
PAGE DIRECTORY/TABLE ENTRIES

- x86-64 page directory / table entries are 64 bits
  - 36 bits used to specify physical address of either a page table, or a virtual memory page
  - Other bits contain additional details about the entry

- Bit 0 (least-significant bit) is the Present bit
  - (i.e. the valid bit from last lecture)
  - When 1, the referenced page is cached in memory
  - When 0, the referenced page is not in memory (e.g. page is stored on disk)

- When Present = 0, all other bits are available for the kernel to use

<table>
<thead>
<tr>
<th>63</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>

Available for Operating System

- Specifies location on disk of where the page is stored
When Present bit is 1, page directory and page table entries contain several bookkeeping values

Page directory entry (all directory levels are roughly like this):

Page table entry:

Very similar contents for both kinds of entries
Bits 1 and 2 specify access permissions
- R/W = 1 is read/write, R/W = 0 is read-only
- U/S = 1 is user access, U/S = 0 is kernel access only

x86-64 introduces an execute permission (bit 63)
- XD (“execute disable”) = 1 disallows instruction fetches from the memory region

Dramatically reduces potential for buffer-overflow exploits!
- Set stack and data pages to not be executable
- Set code pages to be executable and read-only
**Page Directory/Table Entries (4)**

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>XD</td>
<td>Page Base Address</td>
<td>...</td>
<td>D</td>
<td>A</td>
<td>PCD</td>
<td>PWT</td>
<td>U/S</td>
<td>R/W</td>
<td>P=1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Bits 3 and 4 specify caching policies for the page
  - PWT specifies write-through or write-back
  - PCD specifies whether cache is enabled or disabled
- Some peripherals are mapped directly into the computer’s memory address space
  - Technique is called *memory-mapped I/O*
  - CPU interacts with the peripheral by reading and writing specific memory locations
  - Memory addresses read/write directly to the I/O device
    - These addresses are called *I/O ports*
- Definitely don’t want to cache the memory page in these cases!
**Page Directory/Table Entries (5)**

- **Bit 5 is the Accessed bit**
  - MMU sets this to 1 when the page is read or written
  - The kernel is responsible for clearing this bit!

- **Accessed bit used to track what pages have been used**
  - Helps kernel decide which page to evict when it needs to free up space in physical memory

- **Bit 6 is the Dirty bit**
  - Only in page table entries, not in page directory entries!
  - MMU sets this to 1 when the page is written to

- **Dirty bit allows kernel to know when a victim page must be written back to the disk before it is evicted**
  - Kernel is responsible for handling and clearing this bit
Page Directory/Table Entries (6)

- Processor causes faults in certain situations
- If Present = 0 when a page is accessed, the CPU raises a *page-fault* exception
  - Kernel page-fault handler can load the page into memory if it’s on disk
  - Or, if the page is unallocated, generate an error
- If R/W or U/S or XD bits prohibit an access, the CPU raises a *general protection fault*
  - Kernel general protection fault handler can respond in various ways, but typically process is terminated
**x86-64 Address Translation and TLBs**

- Page directory and page tables are stored in DRAM main memory
  - Worst case: 50-100ns access penalty
  - If needed block is in L1 cache, 1-3 clock hit-time
- CPU includes a Translation Lookaside Buffer (TLB) to eliminate even this lookup penalty
  - A hardware cache with same design as SRAM caches
- x86-64 family processors:
  - TLB is a set-associative cache; architecture varies
  - Input to TLB cache is the virtual page number
  - Each cache line holds a details for a page table entry, including the physical page number and permissions
**Address translation logic works something like this:**

- **Virtual Address**
  - **VPN**
  - **VPO**
  - **TLB Tag**
  - **TLBI**
  - **PGDE**
  - **PUDE**
  - **PMDE**
  - **PTE**

- **Physical Address**
  - **PPN**
  - **PPO**

**TLB Miss**
- **VPN1**
- **VPN2**
- **VPN3**
- **VPN4**

**TLB Hit**
- **Partial TLB Hit**

Some bits from VPN are used to identify the cache-set index.

Cache can also match parts of tag, to allow for *partial TLB hits*.
In case of a TLB miss:

- Virtual page number is broken into indexes into the page directories and the page table.
- Incurs full lookup penalty, but the TLB cache is also updated with the results of the lookup.
Ideally, we want a TLB hit:

- Virtual page number is broken into a tag and a cache-set index (TLBI), as usual
- If TLB cache line contains page table entry (PTE), use this for the physical page number (PPN)
Sometimes, we get a *partial TLB hit*

- Some part of the page-directory sequence is present in the TLB, but not the page table entry itself
- Use available info to shorten page-table traversal, and cache result back into TLB
KERNEL AND VIRTUAL MEMORY SYSTEM

- The kernel plays an important role in the virtual memory system
  - Manages the page directories and page tables of running processes
  - Handles page faults and general protection faults
- Each process has its own virtual address space
  - Each process has its own page directory that specifies the process’ virtual memory layout
- On x86-64, only the kernel can change the current page directory being used
  - Requires level 0 (highest) privilege
  - Page tables are also only updatable by the kernel
**Process Memory Layout**

- Each process has its own virtual address space
- Part of virtual address space is devoted to kernel
  - Region starting at address $0x800000000000$ ($2^{47}$)
  - This memory only accessible by the kernel
- Includes functionality and data structures necessary for all processes...
  - Simply map these physical pages into every process’ virtual address space
**The Kernel and System Calls**

- Kernel code is mapped into each process’ address space
  - *Easy* to make system calls!
- Can call kernel code via **syscall** trap
  - Allows a change directly to privilege level 0
  - **syscall** instruction uses values in special control registers to jump to kernel
  - The kernel sets these control registers during OS initialization
- Requires that the kernel has some code mapped into user process address space

![Diagram of kernel and system call structure](image-url)
**Process-Specific Kernel Data**

- Each process also includes process-specific structures, managed by the kernel
  - Also accessible only by kernel
- Kernel stack:
  - Every protection level has its own stack...
  - When a process makes system calls, the kernel-stack used is also only within that process’ address space
- Several other data structures are also managed per-process
  - e.g. page directory/tables for the process, memory mapping info

![Diagram of memory mapping and process structures](image)
Kernel manages several data structures to track virtual memory regions within each process:

- Regions are called “areas” or “segments”
- (Not related to old 8086 segmented memory model!)

- **mm_struct** characterizes current state of process’ virtual memory:
  - **pgd** is the page directory for the process
  - **mmap** is a list of memory areas within the process
Process Virtual Memory Areas (2)

- **vm_area_struct** fields specify details of each memory area
  - `vm_start, vm_end` specify extent of the memory area
  - `vm_prot` specifies read/write permissions for the memory area
  - `vm_flags` specifies whether memory area is shared among processes, or private to this process

- Normal memory accesses:
  - (Page is in memory, and the operation is allowed)
  - No intervention needed from the kernel...
  - CPU and MMU handle these accesses without any trouble
**FAULTS!**

- When a page fault or general protection fault occurs, the kernel must handle the situation!
- Faults can occur for many different reasons...
  - **Invalid accesses:**
    - Program tried to write read-only memory
    - Program tried to access kernel-only memory
  - **Valid accesses:**
    - Accessed page is in the swap device, not DRAM
    - Accessed page hasn’t been allocated to program
- Kernel must decide how to handle each fault
Faults! (2)

- The process’ `vm_area_struct`-list tells the kernel how each fault should be handled.
- If the program accesses a non-existent page:
  - MMU raises a page fault.
- Kernel must check all area structs to see if the address itself is valid:
  - Does it fall within some `vm_start` and `vm_end`?
- If not a valid address, a segmentation-fault signal is sent to the process:
  - Default handler: terminate the process!
If kernel determines that the address is valid, it must next check if the operation is valid

- Is the process writing to read-only memory?
- Is the process accessing kernel-only memory?
- MMU raises a general protection fault

If operation is invalid, a segmentation-fault signal is delivered

- Again, the process gets terminated.
FaulTs! (4)

- At this point, the kernel knows that the address is valid, and the operation is allowed
- Perform normal page-load operations:
  - Select victim page to evict
  - If victim page is dirty, write it back to disk
  - Load requested virtual page into memory
  - Return from fault handler
- CPU restarts instruction that caused the fault
  - This time, the instruction succeeds, since page is now in main memory
Next Time

- Covered most of how the kernel can provide a useful virtual memory abstraction...

- Next time, finish up with a few higher-level abstractions that operating systems build on top of virtual memory