Virtual Memory

Today’s Menu:

- Virtual Memory
  - Virtual vs. Physical Address Spaces
  - Page Table
  - Address Translation
  - Page Fault
  - Replacement Policy
  - Translation Lookaside Buffer (TLB)
  - Protection

Memory Hierarchy of a Computer System

- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.

<table>
<thead>
<tr>
<th>Speed (ns)</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1s</td>
<td>10s</td>
</tr>
<tr>
<td>10ns</td>
<td>100s</td>
</tr>
<tr>
<td>100ns</td>
<td>100s</td>
</tr>
<tr>
<td>10,000ns</td>
<td>100s</td>
</tr>
<tr>
<td>1s</td>
<td>10s</td>
</tr>
<tr>
<td>100s</td>
<td>100s</td>
</tr>
<tr>
<td>100s</td>
<td>100s</td>
</tr>
<tr>
<td>1s</td>
<td>100s</td>
</tr>
<tr>
<td>100s</td>
<td>100s</td>
</tr>
</tbody>
</table>

How Does a Program Start Running?

- The program is copied from permanent storage into memory - on PCs and workstations, the *operating system* copies the program (bits) from disk

- CPU’s Program Counter is then set to starting address of program and program begins execution

What if the Program is Too Big

- Some machines won’t let you run the program
- Original DOS

Memory Allocation for Multiprogramming

- Multiple jobs (multiprogramming) each require memory
Memory Allocation for Multiprogramming (cont.)

- Job 5 doesn’t fit
  - There is enough empty memory to hold the program
  - But job 5 won’t fit contiguously
  - We call this external fragmentation

Virtual Memory

- What is virtual memory?
  - Technique that allows execution of a program that can reside in non-contiguous memory locations
  - Does not have to completely reside in memory
  - Allows the computer to “fake” a program into believing that its memory is contiguous
  - Memory space is larger than physical memory

- Why is VM important?
  - Cheap - no longer have to buy lots of RAM
  - Removes burden of memory resource management from the programmer
  - Enables multiprogramming, time-sharing, protection

Virtual Memory

- Main memory (physical memory) can act as a cache for the secondary storage (disk)

- Advantages:
  - Illusion of having more and contiguous physical memory
  - Program relocation by “pages”
  - Protection in multiprogramming

How Does VM Work

- Two memory “spaces”
  - Virtual memory space - what the program “sees”
  - Physical memory space - what the program runs in (size of RAM)

- On program startup
  - OS copies program into RAM
  - If there is not enough RAM, OS stops copying program & starts running the program with some portion of the program loaded in RAM
  - When the program touches a part of the program not in physical memory (RAM), OS copies that part of the program from disk into RAM (page fault exception)
  - In order to copy some of the program from disk to RAM, the OS must evict parts of the program already in RAM
  - OS copies the evicted parts of the program back to disk if the pages are dirty (i.e., if they have been written into, and changed – dirty bit just like caches?)

Sound like a cache?

Basic VM Algorithm

- Program uses virtual addresses (load, store, instruction fetch)
- Computer translates virtual address (VA) to physical address (PA)
- Computer reads RAM using PA, returning the data to program

Page Tables

- Table which holds VA -> PA translations is called the page table
- In our current scheme, each word is translated from a virtual address to a physical address
- How big is the page table?

Processor (running program)

Virtual Address

Instructions

RAM

Processor (running program)

Virtual Address

VA -> PA
Page Tables (cont.)

- Instead of a "fine-grain" VM where any word in VM can map to any RAM word location, we partition memory into bigger chunks called pages
  - Typical page size today is 4KBytes
  - But we’re moving towards “large” 2MB pages

- Reduces the number of VA→PA translation entries
  - Only one translation per page
  - For 4 KByte page, that’s one VA→PA translation for every 1,024 words

Virtual Address

<table>
<thead>
<tr>
<th>Page Table Entry (PTE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid bit</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Each entry in the page table is called a Page Table Entry (PTE)

# of bits in page offset = \( \log_2(\text{Page Size}) \)
# of bits in physical address = \( \log_2(\text{RAM Bytes}) \)

Aside: Caches and VM

- Idea is just like lines in caches

Virtual Pages & Physical Page Frames

<table>
<thead>
<tr>
<th>Virtual Address Space</th>
<th>Physical Address Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x0000</td>
</tr>
<tr>
<td>0x0010</td>
<td>0x0010</td>
</tr>
<tr>
<td>0x0020</td>
<td>0x0020</td>
</tr>
<tr>
<td>0x0030</td>
<td>0x0030</td>
</tr>
<tr>
<td>0x0040</td>
<td>0x0040</td>
</tr>
<tr>
<td>0x0050</td>
<td>0x0050</td>
</tr>
<tr>
<td>0x0060</td>
<td>0x0060</td>
</tr>
<tr>
<td>0x0070</td>
<td>0x0070</td>
</tr>
<tr>
<td>0x0080</td>
<td>0x0080</td>
</tr>
<tr>
<td>0x0090</td>
<td>0x0090</td>
</tr>
<tr>
<td>0x00A0</td>
<td>0x00A0</td>
</tr>
<tr>
<td>0x00B0</td>
<td>0x00B0</td>
</tr>
<tr>
<td>0x00C0</td>
<td>0x00C0</td>
</tr>
<tr>
<td>0x00D0</td>
<td>0x00D0</td>
</tr>
<tr>
<td>0x00E0</td>
<td>0x00E0</td>
</tr>
<tr>
<td>0x00F0</td>
<td>0x00F0</td>
</tr>
</tbody>
</table>

Page Frames

- Every address within a virtual page maps to the same location within a physical page frame
  - In other words, bottom \( \log_2(\text{Page Size}) \) addr bits not translated

<table>
<thead>
<tr>
<th>Page Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Pages</td>
</tr>
<tr>
<td>Virtual Pages</td>
</tr>
</tbody>
</table>

Page Table Example

1) Break VA into virtual page number and page offset
2) Copy page offset to physical address
3) Use virtual page number as index into page table
4) Copy physical page number from page table to the physical address

Virtual Page

<table>
<thead>
<tr>
<th>Physical Page Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Page Number</td>
</tr>
<tr>
<td>Page Offset</td>
</tr>
</tbody>
</table>

Virtual Address

<table>
<thead>
<tr>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Page Number</td>
</tr>
<tr>
<td>Page Offset</td>
</tr>
</tbody>
</table>
**Page Table Example**

```
<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Page Offset</th>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0</td>
<td>0x0000</td>
</tr>
<tr>
<td>0x0001</td>
<td>0</td>
<td>0x0001</td>
</tr>
<tr>
<td>0x0002</td>
<td>0</td>
<td>0x0002</td>
</tr>
<tr>
<td>0x0003</td>
<td>0</td>
<td>0x0003</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>0x00ff</td>
<td></td>
<td>0x00ff</td>
</tr>
</tbody>
</table>
```

**VA = 0x00000100**

```
<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Page Offset</th>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0</td>
<td>0x0000</td>
</tr>
</tbody>
</table>
```

**Physical Address**

```
<table>
<thead>
<tr>
<th>Physical Page Number</th>
<th>Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>0</td>
</tr>
<tr>
<td>0x0002</td>
<td>0</td>
</tr>
<tr>
<td>0x0003</td>
<td>0</td>
</tr>
<tr>
<td>0x0004</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x00ff</td>
<td></td>
</tr>
</tbody>
</table>
```

**Page Table Example w/ 8KByte Page Size**

```
<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Page Offset</th>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0</td>
<td>0x0000</td>
</tr>
<tr>
<td>0x0001</td>
<td>0</td>
<td>0x0001</td>
</tr>
<tr>
<td>0x0002</td>
<td>0</td>
<td>0x0002</td>
</tr>
<tr>
<td>0x0003</td>
<td>0</td>
<td>0x0003</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>0x00ff</td>
<td></td>
<td>0x00ff</td>
</tr>
</tbody>
</table>
```

**VA = 0x000004100**

```
<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Page Offset</th>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0</td>
<td>0x0000</td>
</tr>
</tbody>
</table>
```

**Physical Address**

```
<table>
<thead>
<tr>
<th>Physical Page Number</th>
<th>Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>0</td>
</tr>
<tr>
<td>0x0002</td>
<td>0</td>
</tr>
<tr>
<td>0x0003</td>
<td>0</td>
</tr>
<tr>
<td>0x0004</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x00ff</td>
<td></td>
</tr>
</tbody>
</table>
```

**What Happens if Page is not in RAM?**

- How do we know it’s not in RAM?
  - Page table entry’s valid bit is set to INVALID (DISK)
- What do we do?
  - Hardware asks OS to fetch the page from disk - we call this a **page fault**
  - Before page is read from disk, OS must evict a page from RAM (if RAM is full)
- If the page to be evicted is called the **victim page**
- If the page to be evicted is dirty, write the page back to disk
- Only data pages can be dirty
- OS then reads the requested page from disk
- OS changes the page table to reflect the new mapping
- Hardware restarts at the faulting virtual address

**Which Page Should We Evict?**

- Optimal solution: evict a page that won’t be referenced (used) again
- If all pages will be used again, then evict the page that will not be used for the longest period of time
  - Guarantees the lowest possible page fault rate (# of faults per second)
  - Can’t be done unless we can tell the future
- Other page replacement algorithms
  - First-in, First-out (FIFO)
  - Least Recently Used (LRU)
- So LRU is really expensive to keep track of. Can we do this?
  - How long does it take to load the page from disk?

**First-in, First-out (FIFO)**

- Oldest page is evicted
- How do we know which page is oldest?
  - Keep a list of 1st-time references
  - When a page must be evicted, pick the page at the end (bottom) of the list
- Example w/ 4 pages

```
<table>
<thead>
<tr>
<th>Page Reference Pattern</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1, 3, 6, 0, 1, 3</td>
<td>0</td>
</tr>
<tr>
<td>0, 5, 2, 1, 6, 0, 1</td>
<td>1</td>
</tr>
<tr>
<td>0, 5, 2, 1, 0, 1, 3</td>
<td>2</td>
</tr>
<tr>
<td>0, 5, 2, 1, 0, 1, 3</td>
<td>3</td>
</tr>
<tr>
<td>0, 5, 2, 1, 0, 1, 3</td>
<td>4</td>
</tr>
<tr>
<td>0, 5, 2, 1, 0, 1, 3</td>
<td>5</td>
</tr>
<tr>
<td>0, 5, 2, 1, 0, 1, 3</td>
<td>6</td>
</tr>
<tr>
<td>0, 5, 2, 1, 0, 1, 3</td>
<td>7</td>
</tr>
<tr>
<td>0, 5, 2, 1, 0, 1, 3</td>
<td>8</td>
</tr>
</tbody>
</table>
```

**evict**
Least Recently Used (LRU)

- Evict the page that has not been used for the longest period of time.
- How do we know which page is oldest?
  - Keep a list of pages ordered by reference.
  - When a page must be evicted, pick the page at the end (bottom) of the list.

Example w/ 4 pages:

<table>
<thead>
<tr>
<th>Page Reference Pattern</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1, 3, 2, 0, 1, 3, 4</td>
<td></td>
</tr>
</tbody>
</table>

4 page frames

Evict

When Bad Things Happen to Good Memories...

- So, can anything go wrong here?
  - i.e., this sounds like a great idea. Have reasonable size RAMs, big disks.
  - What bad stuff can happen?
- Answer: thrashing
  - Your program is so large, or, your machine has so many separate jobs (or users), that very little of the program(s) fit in RAM.
  - Your physical RAM is not providing you even minimal spatial locality for your address refs, so very very often, an address faults, and you page constantly.

Pathological Thrashing

- Consider the following reference pattern (in pages):
  - 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5
  - What happens if there are only 4 page frames in RAM?
  - Not all of the program will fit into RAM.
  - Assume LRU

Pathological Thrashing (cont.)

- Consider the following reference pattern (in pages):
  - 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5
  - What happens if there are only 4 page frames in RAM?
  - Not all of the program will fit into RAM.
  - Assume LRU

Performance of Virtual Memory

- If every program in a multiprogramming environment fits into RAM, then virtual memory never "pages" (goes to disk).
- If any program doesn't fit into RAM, then the VM system must page between RAM and disk.
  - Paging is very costly.
  - A disk access (4KBytes) can take ~10 ms ... in 10 ms, a processor can execute ~40 Million instructions.
  - Basically, you really don't want to page very often, if you don't have to.

Pathological Thrashing (cont.)

- Consider the following reference pattern (in pages):
  - 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5
  - What happens if there are only 4 page frames in RAM?
  - Not all of the program will fit into RAM.
  - Assume LRU
Pathological Thrashing (cont.)

Consider the following reference pattern (in pages)
- 1, 2, 3, 4, 5, 1, 2, 3, 4, 5
- What happens if there are only 4 page frames in RAM?
- Not all of the program will fit into RAM
- Assume LRU

What happens if there are only 4 page frames in RAM?
Not all of the program will fit into RAM
LRU can result in thrashing. Random replacement or some other policy could avoid this problem

Multiprogramming and Protection

Most machines run multiple processes (also called jobs, tasks)
- The processes share RAM
- Example: simultaneously running emacs, verilog and X-windows system
- Example: simultaneously running browser, email, chat client

Need mechanism to provide the memory to each process

Need mechanism to protect the memory of each process
- Avoid one process reading or writing the memory of another
- Avoid one process thrashing another, especially the OS
- Usually referred to as "protection"

Protection with Address Bounds

Simplest approach
- Provide pair of registers that checks every address reference to make sure the address falls between the two limits
- Example, when running Job 3, the two registers would point to the bottom and top of Job 3 memory range
- Base <= Address <= Bound

Who owns base & bound?
- The OS
- User process should not be allowed to alter these registers

Only OS can alter these
- How?
- Processor provides at least two modes
  - User and system (OS)
- System mode is called a privileged mode
- Code running in system mode can access parts of the machine (registers) that user mode cannot access (modify)

Problems with Address Bounding

Loader has to re-locate code (jumps, etc), memory loads, etc.
- Or ISA can be all PC relative addressing

What if program can’t fit into given memory slots
- Only provides for contiguous data segments

What if every process thought it had its very own virtual address space?
- 4GB of space, all your own
- Startled where ever you want
- No relocation necessary
- But...need to translate between your virtual address and the physical memory you own

VM and Memory Management

This is how EACH process in Linux sees memory:

Kernel Virtual Memory
User Stack
Shared Libraries
Run-time heap
Read/Write Segment
Read-only Segment

If each process looks like this, how do we ensure that each gets its own physical memory allocated?
How to ensure that one process cannot overwrite another’s memory?

32 bit virtual address space
0x00000000
0x004000000
0x008048000

From Executable
**Mapping to Physical Memory**

Virtual memory can be used to share memory and protect memory from another process! (And all sorts of cool things with virtual machines!)

**Protection with Private Page Table**

- Finer-grained protection is accomplished by protecting each page
- Give each user process its own page table
- Because every address necessarily goes through VA->PA translation, the process can only access physical pages listed in its own page table
- Example: process can only access physical pages 1, 3, 4, 5, 8

**Protection (cont.)**

- Separate page tables (per process) provide page-level protection
- OS creates and manages page tables so that no user-level process can alter any process's page table

**Protection (cont.)**

- Notice that both processes use the same virtual addresses
- Notice the physical addresses are never the same
- Can process sharing of virtual addresses cause problems?
  - Yes (we’ll see when we talk about TLBs)

**VM Support**

- Each process has its own page table
- Extra bits to allow/disallow
  - Read access
  - Write access
  - Access when in supervisor mode only (kernel code)
- Support shared pages (libraries or memory)
  - Read only for multiple processes

If a process accesses a memory location that it does not have proper access to:

**Segmentation Fault**
Who Does the VA->PA Translation?

- Software would be too slow for every memory reference
  - If not in software, then must be done in hardware
- Hardware needs to hold all of the PTEs (i.e., the entire page table)?
- New question: How big is the page table?
  - Assume a 4K page and 32 bit virtual address space
  - 32 bits can address 4 Gigabytes of RAM ($2^{32} = 4G$)
  - 12 bits are for the page offset
  - 20 bits for the page number → $2^{20}$ virtual pages
  - $2^{20} = 1,048,576$ PTEs
  - Each PTE is at least 4 Bytes
  - 1 Million PTEs * 4 Bytes/PTE ≈ 4 Megabytes
  - Too large to hold entire page table inside CPU chip itself
  - Most processors cache the most recently used PTEs...

Where? Translation Lookaside Buffer (TLB)

- TLB Caches most recently referenced PTEs
  - Very similar to instruction and data caches
- Can be accessed in <= single cycle
  - Remember 1.33 accesses/instruction

Making Address Translation Fast

- A cache for address translations: translation lookaside buffer

Where is My Page? My Page Table Entry?

- Page in physical mem, PTE in TLB
  - Life is good. Best answer. TLB lookup hits, VA->PA resolved, fast access to data.
- Page in physical mem, PTE not in TLB, PTE only in phys mem
  - Life is so-so. TLB lookup misses. TLB reloads from page table out in memory. (10s-1000s cycles)
  - VA-PA takes longer, but at least your address is already in physical memory.
    (People actually optimize code to avoid this!)
- Page not in physical mem, PTE in TLB
  - Life sucks. TLB lookup misses. You discover your address is not in physical memory. Page fault, you wait to go load the page.
    (Disk latency is ~10ms – you wait that long.)
- Page not in phys mem, PTE not in TLB, PTE only in phys mem
  - Life stinks. TLB lookup misses. TLB reloads from page table out in memory. During that reload, you discover your page itself is not in physical memory. Page fault, you wait to go load the page.
    (Disk latency is ~10ms – you wait that long. PLUS 10s-1000s of cycles to load TLB)

TLB Example: Address Ref 1

Assume a 4K-Byte pages and a 32 bit address space

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00002</td>
<td>434</td>
</tr>
<tr>
<td>0x01</td>
<td>0x002</td>
</tr>
<tr>
<td>0x02</td>
<td>0x024</td>
</tr>
<tr>
<td>0x03</td>
<td>0x010</td>
</tr>
<tr>
<td>0x04</td>
<td>0x046</td>
</tr>
<tr>
<td>0x05</td>
<td>0x020</td>
</tr>
<tr>
<td>0x06</td>
<td>0x081</td>
</tr>
<tr>
<td>0x07</td>
<td>0x062</td>
</tr>
<tr>
<td>0x08</td>
<td>0x094</td>
</tr>
</tbody>
</table>

Physical Address 434

TLB Example: Address Ref 1 Misses in TLB

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x021</td>
</tr>
<tr>
<td>0x01</td>
<td>0x024</td>
</tr>
<tr>
<td>0x02</td>
<td>0x010</td>
</tr>
<tr>
<td>0x03</td>
<td>0x046</td>
</tr>
<tr>
<td>0x04</td>
<td>0x020</td>
</tr>
<tr>
<td>0x05</td>
<td>0x081</td>
</tr>
<tr>
<td>0x06</td>
<td>0x062</td>
</tr>
<tr>
<td>0x07</td>
<td>0x094</td>
</tr>
<tr>
<td>0x08</td>
<td>0x021</td>
</tr>
</tbody>
</table>

Physical Address 434
**TLB Example: Ref 4 Misses in TLB, TLB Conflict**

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00005 000</td>
<td>0x000</td>
</tr>
<tr>
<td>0x00007 004</td>
<td>0x000</td>
</tr>
<tr>
<td>0x00002 001</td>
<td>0x002</td>
</tr>
<tr>
<td>0x00003 002</td>
<td>0x004</td>
</tr>
<tr>
<td>0x00004 006</td>
<td>0x006</td>
</tr>
<tr>
<td>0x00005 000</td>
<td>0x000</td>
</tr>
<tr>
<td>0x00007 004</td>
<td>0x000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TAG</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00005</td>
<td>0x081</td>
</tr>
<tr>
<td>0x00007</td>
<td>0x094</td>
</tr>
</tbody>
</table>

**Physical Address**

0x081 000

**Real Life Examples: Pentium Pro**

- Data TLB:
  - 64 entries, 4 way SA
- Data Cache:
  - 8kB, 2 way SA
- Instruction TLB:
  - 32 entries, 4 way SA
- Instruction Cache:
  - 8kB, 4 way SA

**Summary**

- **Virtual memory**
  - Gives illusion of a LARGE physical RAM, even if you have LESS real RAM
  - RAM divided into chunks called pages.
  - "Like" pages are in the physical RAM.
  - Pages that don't fit are on the disk.
  - Hardware translates the virtual address (big address space) into the physical address (real RAM address). Allows a page to be placed anywhere.
  - Every application sees the same virtual address space. (Simplifies multiprogramming.)
  - Provides efficient use of real memory (non-contiguous mappings)

- **But...**
  - Adds another level of indirection that you need to go through on every memory access
  - Another cache to maintain, and you have to go to the OS if the cache misses
  - Oh, and it complicates caches... (next time)