Agenda

• Virtual Memory
• Paged Physical Memory
• Swap Space
• Page Faults
• Hierarchical Page Tables
• Caching Page Table Entries (TLB)

Virtual Machine

• 100’s of processes, managed by OS
• But what about memory?
  – There is only one!
  – We cannot just “save” its contents in a context switch...

Virtual vs. Physical Addresses

• Processes use virtual addresses, e.g., 0x... DaFFFFFF
  – Many processes, all using same (conflicting) addresses
• Memory uses physical addresses (also, e.g., 0x... DaFFFFFF)
• Memory manager maps virtual to physical addresses

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
Conceptual Memory Manager

Responsibilities of Memory Manager

1) Map virtual to physical addresses
2) Protection:
   - Isolate memory between processes
   - Each process gets dedicated "private" memory
   - Errors in one program won't corrupt memory of other program
   - Prevent user programs from messing with OS' memory
3) Swap memory to disk
   - Give illusion of larger memory by storing some content on disk
   - Disk is usually much larger and slower than DRAM
   - Use "clever" caching strategies

Agenda

• Virtual Memory
• Paged Physical Memory
• Swap Space
• Hierarchical Page Tables
• Caching Page Table Entries (TLB)

Memory Manager

• Several options
• Today "paged memory" dominates
  - Physical memory (DRAM) is broken into pages
  - Typical page size: 4 KiB+

Paged Memory

Paged Memory Address Translation

• OS keeps track of which process is active
  - Chooses correct page table
• Memory manager extracts page number from virtual address
• Looks up page address in page table
• Computes physical memory address from sum of
  - Page address and
  - Offset (from virtual address)
Protection

- Assigning different pages in DRAM to processes also keeps them from accessing each others memory
  - Isolation
  - Page tables handled by OS (in supervisory mode)
- Sharing is also possible
  - OS may assign same physical page to several processes

Write Protection

Exception when writing to protected page (e.g., program codes)

Where Do Page Tables Reside?

- E.g., 32-Bit virtual address, 4-KiB pages
  - Single page table size:
    - 4 x 2^10 Bytes = 4-MiB
    - 0.1\% of 4-GiB memory
    - But much too large for a cache!
- Store page tables in memory (DRAM)
  - Two (slow) memory accesses per lw/sw on cache miss
  - How could we minimize the performance penalty?
    - Transfer blocks (not words) between DRAM and processor cache
    - Exploit spatial locality
    - Use a cache for frequently used page table entries...

Paged Table Storage in DRAM

Blocks vs. Pages

- In caches, we dealt with individual blocks
  - Usually ~5KiB on modern systems
- In VM, we deal with individual pages
  - Usually ~4 KiB on modern systems
- Common point of confusion:
  - Bytes,
  - Words,
  - Blocks,
  - Pages
  - Are all just different ways of looking at memory!

Bytes, Words, Blocks, Pages

E.g.: 16 KiB DRAM, 4 KiB Pages (for VM), 128 B blocks (for caches), 4 B words (for lw/sw)
Agenda

- Virtual Memory
- Paged Physical Memory
- Swap Space
- Hierarchical Page Tables
- Caching Page Table Entries (TLB)

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
- Nothing mechanical that turns
- Like “Ginormous” register file
  - Does not “forget” when power is off
- Fast access to all locations, regardless of address
- Still much slower than register, DRAM
  - Read/write blocks, not bytes
  - Potential reliability issues

Paged Memory

Each process has a dedicated page table. Physical memory non-consecutive.
Memory Access

• Check page table entry:
  - Valid?
    • Yes, valid → in DRAM?
      - Yes, in DRAM: read/write data
    • No, on disk: allocate new page in DRAM
      - If out of memory, evict a page from DRAM
      - Store evicted page to disk
      - Read page from disk into memory
      - Read/write data
    • Not Valid
      - allocate new page in DRAM
      - If out of memory, evict a page
      - Read/write data

Page fault OS intervention

Remember: Out of Memory

• Insufficient free memory: malloc() returns NULL

```c
#include <stdio.h>
int main() {
    int *p;
    if (malloc(512) == NULL) {
        free(p); // malloc(0)!
        return 1; // abort
    }
    // Do Free, keep allocating until out of memory
}
```

$ gcc OutOfMemory.c; ./a.out
failed to allocate > 131 TiBytes

What’s going on?

Write-Through or Write-Back?

• DRAM acts like "cache" for disk
  - Should writes go directly to disk (write-through)?
  - Or only when page is evicted?

• Which option do you propose?
  • Implementation?

Agenda

• Virtual Memory
• Paged Physical Memory
• Swap Space
• Hierarchical Page Tables
• Caching Page Table Entries (TLB)

Size of Page Tables

• E.g., 32-Bit virtual address, 4-KiB pages
  - Single page table size:
    • 4 x 2^20 Bytes = 4-MiB
    • 0.1% of 4-GiB memory
  - Total size for 256 processes (each needs a page table)
    • 256 x 4 x 2^20 Bytes = 256 x 4-MiB = 1-GiB
    • 25% of 4-GiB memory!
  - What about 64-bit addresses?

How can we keep the size of page tables "reasonable"?

Options

• Increase page size
  - E.g., doubling page size cuts PT size in half
  - At the expense of potentially wasted memory
• Hierarchical page tables
  - With decreasing page size
• Most programs use only fraction of memory
  - Split PT in two (or more) parts
Hierarchical Page Table – Exploits Sparsity of Virtual Address Space Use

- Level 1 Page Table
  - Page size 10b → 1024 x 4096B
- Level 2 Page Tables
  - 12b → 4096B
- Data Pages
  - page in primary memory
  - page in secondary memory

Address Translation and Protection

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

Good VM design should be fast (~one cycle) and space efficient

Administrivia

- Homework 6 was released—due next Wednesday, Nov 22
- Project 3 is due Monday
  - Hive servers will get more overloaded closer to the deadline
  - It’s in your best interest to finish as soon as possible!
- There is lecture on Tuesday (I/O: DMA, Disks, Networking)
- There is no discussion or lab next week
  - Enjoy your Thanksgiving!
- Project 4: Spark will be released by Monday night
- Homework 7 will be released next Wednesday

Agenda

- Virtual Memory
- Paged Physical Memory
- Swap Space
- Hierarchical Page Tables
- Caching Page Table Entries (TLB)

Translation Lookaside Buffers (TLB)

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

Solution: Cache some translations in TLB

TLB hit ⇒ Single-Cycle Translation
TLB miss ⇒ Page-Table Walk to refill
TLB Designs

- Typically 32-128 entries, usually 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-way set-associative
  - Larger systems sometimes have multi-level (L1 and L2) TLBs
- “TLB Reach” - Size of largest virtual address space that can be simultaneously mapped by TLB
- Random or FIFO replacement policy

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

TLB Reach = ____________________________

VM-related Events in Pipeline

- Handling a TLB miss needs a hardware or software mechanism to refill TLB
  - Usually done in hardware
- Handling a page fault (e.g., page is on disk) needs a precise trap so software handler can easily resume after retrieving page
- Protection violation may abort process

Hierarchical Page Table Walk: SPARC v8

• MMU does this table walk in hardware on a TLB miss
  • FSM?

Page-Based Virtual-Memory Machine

• Assumes page tables held in untranslated physical memory

Address Translation: putting it all together

Modern Virtual Memory Systems

Illusion of a large, private, uniform store

Protection & Privacy
Several users/processes, each with their private address 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
It’s Just the “OS” ...

- Let’s write `execve`
  - Code the loads program into memory for execution
  - What’s the best way?

Execve

1. Set up PT
   - for program code
   - and stack
   - mark PTEs as invalid
2. Init `argv, argc`
3. Call `main`
   - page fault!
4. What happens?
   - `.text` and `.data` sections copied
   - page by page, on demand, by VM system

And, in Conclusion ...

- Virtual and physical addresses
  - Program \(\rightarrow\) virtual address
  - DRAM \(\rightarrow\) physical address
- Paged Memory
  1. Facilitates virtual \(\rightarrow\) physical address translation
  2. Provides isolation & protection
  3. Extends available memory to include disk
- Implementation issues
  - Hierarchical page tables
  - Caching page table entries (TLB)