Virtual Memory Concepts
Virtual Memory

- Virtual memory - Next level in the memory hierarchy:
  - Provides program with illusion of a very large main memory: Working set of “pages” reside in main memory - others are on disk
  - Demand paging: Provides the ability to run programs larger than the primary memory (DRAM)
  - Hides differences between machine configurations
- Also allows OS to share memory, protect programs from each other
- Today, more important for protection than just another level of memory hierarchy
- Each process thinks it has all the memory to itself
- (Historically, it predates caches)
Great Idea #3: Principle of Locality / Memory Hierarchy

- **Processor chip**
  - Tiny capacity
  - Extremely fast
  - Extremely expensive

- **CPU Cache**
  - Level 1 (L1) Cache
  - Level 2 (L2) Cache
  - Level 3 (L3) Cache

- **Physical Memory**
  - Random-Access Memory (RAM)

- **Virtual Memory**
  - Solid-State Memory (Flash)

- **Magnetic Disks**

**Core**
- Faster
- Expensive
- Small capacity

**SSD, HDD Drives**
- (Not so) Fast
- Cheap
- Large capacity

**DRAM chip**
- e.g.
  - DDR3/4/5
  - HBM/HBM2/3

**Priced reasonably**
- Fast
- Medium capacity

**Solid-State Memory (Flash)**
- Fast
- Expensive
- Small capacity

**Random-Access Memory (RAM)**
- Priced reasonably
- Medium capacity

**Extremely fast**
- Tiny capacity

**RISC-V (36)**

Garcia, Nikolić
Virtual vs. Physical Addresses

- Processes use virtual addresses, e.g., 0 ... 0xffff,ffff
  - Many processes, all using same (conflicting) addresses
- Memory uses physical addresses (also, e.g., 0 ... 0xffff,ffff)
  - Memory manager maps virtual to physical addresses

Many of these (soft & hardware cores)

One main memory
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 (‘translates’) between these two address spaces
Bora’s Laptop

```
bora@DESKTOP-QAB1DCR:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
Address sizes:
  CPU(s):              8
  On-line CPU(s) list: 0-7
  Thread(s) per core:  2
  Core(s) per socket:  4
  Socket(s):           1
  Vendor ID:           GenuineIntel
  CPU family:          6
  Model:               126
  Model name:          Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz
  Stepping:            5
  CPU MHz:             1497.604
  BogoMIPS:            2995.20
  Hypervisor vendor:   Microsoft
  Virtualization type: full
L1d cache:             192 KiB
L1i cache:             128 KiB
L2 cache:              2 MiB
L3 cache:              8 MiB
```

Garcia, Nikolić
Analogy

- Book title like **virtual address**
- Library of Congress call number like **physical address**
- Card catalogue like **page table**, mapping from book title to call #
- On card for book, in local library vs. in another branch like **valid bit** indicating in main memory vs. on disk (storage)
- On card, available for 2-hour in library use (vs. 2-week checkout) like **access rights**
Memory Hierarchy Requirements

- Allow multiple processes to simultaneously occupy memory and provide protection – don’t let one program read/write memory from another.

- Address space – give each program the illusion that it has its own private memory.
  - Suppose code starts at address 0x40000000. But different processes have different code, both residing at the same address. So each program has a different view of memory.
Physical Memory and Storage
Memory (DRAM)

- Desktop/server
- MS Surface Book
- Apple A12 Bionic (DRAM goes on top)

Volatile
- Latency to access first word: ~10ns (~30-40 processor cycles)
- Each successive (0.5ns – 1ns)
- Each access brings 64 bits
- Supports ‘bursts’
Storage – “Disk”

• Attached as a peripheral I/O device; Non-volatile

  ▪ SSD
    ▫ Access: 40-100μs
      (~100k proc. cycles)
    ▫ $0.05-0.5/GB

  ▪ HDD
    ▫ Access: <5-10ms
      (10-20M proc. cycles)
    ▫ $0.01-0.1/GB
Aside ... Why are Disks So Slow?

- 10,000 rpm (revolutions per minute)
- 6 ms per revolution
- Average random access time: 3 ms (~10^7 processor cycles)
How Hard Drives Work?

- Nick Parlante’s https://cs.stanford.edu/people/nick/how-hard-drive-works/
  - Several YouTube videos as well
What About SSD?

- Made with transistors
- Nothing mechanical that turns
- Like “Ginormous” register file
  - Does not “forget” when power is off (non-volatile)
- Fast access to all locations, regardless of address
- Still much slower than register, DRAM
  - Read/write blocks, not bytes
  - Potential reliability issues
- Some unusual requirements:
  - Can’t erase single bits – only entire blocks
Flash Memory

- 3D array of bit cells (up to 256 layers!)

Western Digital/Semiengineering
Memory Manager
Virtual Memory

- In a ‘bare metal’ system (w/o OS), addresses issued with loads/stores are real physical addresses
  - In this mode, 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
    - Check that process has permission to access a particular part of memory
100+ Processes, managed by OS

- 100’s of processes
  - OS multiplexes these over available cores
- But what about memory?
  - There is only one!
  - We cannot just “save” its contents in a context switch …
Processes use virtual addresses, e.g., 0 … 0xffff,ffff
  - Many processes, all using same (conflicting) addresses
Memory uses physical addresses (also, e.g., 0 … 0xffff,ffff)
  - *Memory manager maps virtual to physical addresses*
Conceptual Memory Manager

Concept; Real memory managers use more complex mappings
Responsibilities of Memory Manager

1) Map virtual to physical addresses

2) Protection:
   - Isolate memory between processes
   - Each process gets dedicate "private" memory
   - Errors in one program won’t corrupt memory of other program
   - Prevent user programs from messing with OS’s 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
Paged Memory
Concept of “paged memory” dominates

- Physical memory (DRAM) is broken into pages
- Typical page size: 4 KiB+ (on modern OSs)
  - Need 12 bits to address 4KiB

Virtual address (e.g., 32 Bits)

| page number (e.g., 20 Bits) | offset (e.g., 12 Bits) |
Review: Conceptual Memory Manager
Each process has a dedicated page table. Physical memory non-consecutive.

RISC-V (58)
Paged Memory Address Translation

OS keeps track of which process is active
- Chooses correct page table

Memory manager extracts page number from virtual address
- e.g. just top 20 bits

Looks up page address in page table

Computes physical memory address from sum of
- Page address and
- Offset (from virtual address)

Physical addresses may (but do not have to) have more or fewer bits than virtual addresses
Assigning different pages in DRAM to processes also keeps them from accessing each other's memory:

- Isolation
- Page tables handled by OS (in supervisory mode)

Sharing is also possible:

- OS may assign the same physical page to several processes
Exception when writing to a protected page (e.g. program code)
Where Do Page Tables Reside?

- E.g., 32-Bit virtual address, 4-KiB pages
  - Single page table size:
    - $4 \times 2^{20}$ 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 ...
Page Table Stored in Memory

lw/sw take two memory references

RISC-V (63)
Page Faults
Blocks vs. Pages

- In caches, we dealt with individual *blocks*
  - Usually ~64B 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), 4B words (for `lw/sw`)

<table>
<thead>
<tr>
<th>Page 0</th>
<th>Page 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can think of memory as:</td>
<td>Can think of a page as:</td>
</tr>
<tr>
<td>• 4 Pages, or</td>
<td>• 32 Blocks, or</td>
</tr>
<tr>
<td>• 128 Blocks, or</td>
<td>• 1024 Words</td>
</tr>
<tr>
<td>• 4096 Words, or</td>
<td></td>
</tr>
<tr>
<td>• 16,384 Bytes</td>
<td></td>
</tr>
</tbody>
</table>

1 of 1 Memory

1 of 4 Pages per Memory

1 of 32 Blocks per Page

Garcia, Nikolić
Review: Analogy

- Book title like virtual address
- Library of Congress call number like physical address
- Card catalogue like page table, mapping from book title to call #
- On card for book, in local library vs. in another branch like valid bit indicating in main memory vs. on disk (storage)
- On card, available for 2-hour in library use (vs. 2-week checkout) like access rights
Paged Memory

Valid: Page allocated

DRAM/Disk

Page table

Memory (DRAM)

Page N

Disk

Garcia, Nikolić
Check page table entry:

- **Valid?**
  - Yes, valid $\rightarrow$ 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
Page Fault

- Page faults are treated as exceptions
  - Page fault handler (yet another function of the interrupt/trap handler) does the page table updates and initiates transfers
  - Updates status bits
- (If page needs to be swapped from disk, perform context switch)
- Following the page fault, the instruction is re-executed
Remember: Out of Memory

- Insufficient free memory: malloc() returns NULL

```c
int main(void) {
    const int G = 1024*1024*1024;
    for (int n=0; ;n++) {
        char *p = malloc(G*sizeof(char));
        if (p == NULL) {
            fprintf(stderr,
                "failed to allocate > %g TiBytes\n", n/10000.0);
        }
    return 1; // abort program
} // no free, keep allocating until out of memory
```

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