Virtual Memory and Caching
Virtual Memory
- Virtual vs. physical memory
- Page tables
- Page faults

Storage and Locality
- The storage hierarchy
- Spatial and temporal locality
- Caching

Effective Caching
- Block size
- Eviction policy
- Order of operations
Each process sees main memory as
Huge: $2^{64} = 16$ EB (16 exabytes) of memory $\approx 10^{19}$ bytes
Uniform: contiguous memory locations from 0 to $2^{64}-1$
Memory is divided into pages
• At any time, some pages are in physical memory, some on disk
• OS and hardware swap pages between physical memory and disk
• Multiple processes share physical memory
Virtual & Physical Addresses

Question
• How do OS and hardware implement virtual memory?

Answer (part 1)
• Distinguish between virtual addresses and physical addresses
Virtual & Physical Addresses (cont.)

**Virtual address**
- Identifies a location in a particular process’s virtual memory
  - Independent of size of physical memory
  - Independent of other concurrent processes
- Consists of virtual page number & offset
- Used by application programs

**Physical address**
- Identifies a location in physical memory
- Consists of physical page number & offset
- Known only to OS and hardware

**Note:**
- Offset is same in virtual addr and corresponding physical addr
## ArmLab Virtual & Physical Addresses

<table>
<thead>
<tr>
<th>Address Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>virtual addr</strong></td>
<td>virtual page num</td>
</tr>
<tr>
<td><strong>physical addr</strong></td>
<td>physical page num</td>
</tr>
</tbody>
</table>

### On ArmLab:
- Each virtual address consists of 64 bits
  - There are $2^{64}$ bytes of virtual memory (per process)
- Each offset is 16 bits
  - Each page consists of $2^{16}$ bytes
  - Each virtual page number consists of $64 - 16 = 48$ bits
  - There are $2^{48}$ virtual pages
On ArmLab:

- Each physical address consists of 37 bits
  - There are $2^{37}$ (128G) bytes of physical memory (per computer)
- Each offset is 16 bits
  - Each page consists of $2^{16}$ bytes
- Each physical page number consists of $37 - 16 = 21$ bits
  - There are $2^{21}$ physical pages
Question

• How do OS and hardware implement virtual memory?

Answer (part 2)

• Maintain a page table for each process (stored in physical memory)
Page Tables (cont.)

Page Table for Process 1234

<table>
<thead>
<tr>
<th>Virtual Page Num</th>
<th>Physical Page Num or Disk Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Physical page 5</td>
</tr>
<tr>
<td>1</td>
<td>(unmapped)</td>
</tr>
<tr>
<td>2</td>
<td>Spot X on disk</td>
</tr>
<tr>
<td>3</td>
<td>Physical page 8</td>
</tr>
</tbody>
</table>

Page table maps each in-use virtual page to:
- A physical page, or
- A spot on disk
Private Address Space Example 1

- Process executes instruction that references virtual memory
- CPU determines virtual page
- CPU checks if required virtual page is in physical memory: yes
- CPU does load/store from/to physical memory
Private Address Space Example 2

- Process executes instruction that references virtual memory
- CPU determines virtual page
- CPU checks if required virtual page is in physical memory: no!

Now what?
Question

• How do OS and hardware implement virtual memory?

Answer (part 3)

• Trigger a page fault for accesses to virtual pages that are swapped out (on disk)
Process executes instruction that references virtual memory
CPU determines virtual page
CPU checks if required virtual page is in physical memory: **no**!
  • CPU generates **page fault**
  • OS gains control of CPU
  • OS (potentially) evicts some page from physical memory to disk, loads required page from disk to physical memory
  • OS returns control of CPU to process — to **same instruction**
Process executes instruction that references virtual memory
CPU checks if required virtual page is in physical memory: **yes**
CPU does load/store from/to physical memory

Virtual memory enables the illusion of private address spaces
**Q:** What effect does virtual memory have on the security and speed of processes?

<table>
<thead>
<tr>
<th>Security</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td>↑</td>
</tr>
<tr>
<td>B.</td>
<td>↓</td>
</tr>
<tr>
<td>C.</td>
<td>↑</td>
</tr>
<tr>
<td>D.</td>
<td>↓</td>
</tr>
</tbody>
</table>

Let’s start by considering security...
Consequences of Virtual Memory

Memory protection among processes
- Process’s page table references only physical memory pages that the process currently owns
- Process can’t accidentally/maliciously affect physical memory used by another process

Memory protection within processes
- Permission bits in page-table entries indicate whether page is read-only, etc.
- Allows CPU to prohibit
  - Writing to RODATA & TEXT sections
  - Access to protected (OS owned) virtual memory
### VM Effects on Security and Speed

**Q:** What effect does virtual memory have on the security and speed of processes?

<table>
<thead>
<tr>
<th>Security</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td></td>
</tr>
<tr>
<td>B.</td>
<td></td>
</tr>
<tr>
<td>C.</td>
<td></td>
</tr>
<tr>
<td>D.</td>
<td></td>
</tr>
</tbody>
</table>

OK, so part of the answer is: **Security**

But what about speed?
Consequences of Virtual Memory

Context switching

- **Illusion**: To context switch from process X to process Y, OS must save contents of registers and memory for process X, and restore contents of registers and memory for process Y

- **Reality**: To context switch from process X to process Y, OS must save contents of registers and virtual memory for process X, and restore contents of registers and virtual memory for process Y

- **Implementation**: To context switch from process X to process Y, OS must save contents of registers and pointer to the page table for process X, and restore contents of registers and pointer to the page table for process Y
Consequences of Virtual Memory

Creating new processes

• Efficient for a parent to `fork()` a new child process
Consequences of Virtual Memory

Creating new processes

- Efficient for a parent to `fork()` a new child process
- **Initially:** copy parent’s page table to child, mark both parent and child pages read-only

![Diagram showing parent VM, physical memory, and child VM with read-only access](image)
Consequences of Virtual Memory

Creating new processes
- Efficient for a parent to `fork()` a new child process
  - Initially: copy parent’s page table to child, mark both parent and child pages read-only
  - Incrementally: when parent or child writes to page, page fault handler creates private copies
Overwriting one program with another

• Easy for a process to `execvp()` another program
  • Initially: set page table entries to point to program pages that already exist on disk!
  • Incrementally: page fault handler swaps pages into memory as required
But if we’re thinking about efficiency, isn’t that all outweighed by the need to do multiple physical memory accesses (including page tables) for every virtual access?

Conceptually, umm, yes. But it’s not so bad in reality!
Agenda

Virtual Memory
- Virtual vs. physical memory
- Page tables
- Page faults

Storage and Locality
- The storage hierarchy
- Spatial and temporal locality
- Caching

Effective Caching
- Block size
- Eviction policy
- Order of operations
Typical Storage Hierarchy

- CPU registers hold words retrieved from L1/L2/L3 cache
- L1/L2/L3 cache holds cache lines retrieved from main memory
- Main memory holds disk blocks retrieved from local disks
- Local disks hold files retrieved from disks on remote network servers

- Smaller, Faster, $$$er storage devices
- Larger, Slower, Cheaper storage devices

- registers
- L1 cache
- Level 2 cache
- Level 3 cache
- main memory (RAM)
- local secondary storage (local disks, SSDs)
- remote secondary storage (distributed file systems, Web servers)
Factors to consider:

- Capacity
- Latency (how long to do a read)
- Bandwidth (how many bytes/sec can be read)
  - Weakly correlated to latency: reading 1 MB from a hard disk isn’t much slower than reading 1 byte
- Volatility
  - Do data persist in the absence of power?
Typical Storage Hierarchy

Registers

- **Latency**: 0 cycles
- **Capacity**: 8-256 registers (31 general purpose registers in AArch64)

L1/L2/L3 Cache

- **Latency**: 1 to 40 cycles
- **Capacity**: 32KB to 32MB

Main memory (RAM)

- **Latency**: ~ 50-100 cycles
  - 100 times slower than registers
- **Capacity**: GB

@christianw, @harrisonbroadbent
Typical Storage Hierarchy

Local secondary storage: disk drives

• Solid-State Disk (SSD):
  • Flash memory (nonvolatile)
  • **Latency:** 0.1 ms (~ 300k cycles)
  • **Capacity:** 128 GB – 2 TB

• Hard Disk:
  • Spinning magnetic platters, moving heads
  • **Latency:** 10 ms (~ 30M cycles)
  • **Capacity:** 1 – 10 TB

@benjaminlehman, Samsung Belgium
Cache / RAM Latency

1 clock = $3 \cdot 10^{-10}$ sec

https://www.anandtech.com/show/6993/intel-iris-pro-5200-graphics-review-core-i74950hq-tested/3
Disks

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Access Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDD</td>
<td>1 ms</td>
</tr>
<tr>
<td>SSD</td>
<td>1 μs</td>
</tr>
<tr>
<td>DRAM</td>
<td>1 ns</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unit</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kb</td>
<td>Mb</td>
</tr>
<tr>
<td>Gb</td>
<td>Tb</td>
</tr>
</tbody>
</table>
Typical Storage Hierarchy

Remote secondary storage (a.k.a. “the cloud”)

- **Latency**: tens of milliseconds
  - Limited by network bandwidth
- **Capacity**: essentially unlimited
Storage Device Speed vs. Size

Facts:
- CPU needs sub-nanosecond access to data to run instructions at full speed
- Fast storage (sub-nanosecond) is small (100-1000 bytes)
- Big storage (gigabytes) is slow (15 nanoseconds)
- Huge storage (terabytes) is glacially slow (milliseconds)

Goal:
- Need many gigabytes of memory,
- but with fast (sub-nanosecond) average access time

Solution: locality allows caching
- Most programs exhibit good locality
- A program that exhibits good locality will benefit from proper caching, which enables good average performance
Locality

Two kinds of **locality**

- **Temporal** locality
  - If a program references item X now, then it probably will reference X again soon
- **Spatial** locality
  - If a program references item X now, then it probably will reference item at address X±1 soon

Most programs exhibit good temporal and spatial locality
Locality Example

Locality example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
```

Typical code (good locality)

Temporal locality

- **Data:** Whenever the CPU accesses `sum`, it accesses `sum` again shortly thereafter.
- **Instructions:** Whenever the CPU executes `sum += a[i]`, it executes `sum += a[i]` again shortly thereafter.

Spatial locality

- **Data:** Whenever the CPU accesses `a[i]`, it accesses `a[i+1]` shortly thereafter.
- **Instructions:** Whenever the CPU executes `sum += a[i]`, it executes `i++` shortly thereafter.
Caching

Cache

• Fast access, small capacity storage device
• Acts as a staging area for a subset of the items in a slow access, large capacity storage device

Good locality + proper caching

⇒ Most storage accesses can be satisfied by cache
⇒ Overall storage performance improved
Larger, slower device at level k+1 is partitioned into blocks

Blocks copied between levels

Smaller, faster device at level k caches a subset of the blocks from level k+1
Cache Hits and Misses

**Cache hit**
- E.g., request for block 10
- Access block 10 at level k
- Fast!

**Cache miss**
- E.g., request for block 8
- **Evict** some block from level k
- Load block 8 from level k+1 to level k
- Access block 8 at level k
- Slow!

**Caching goal:**
- Maximize cache hits
- Minimize cache misses
Q: What effect does virtual memory have on the security and speed of processes?

<table>
<thead>
<tr>
<th>Security</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td></td>
</tr>
<tr>
<td>B.</td>
<td></td>
</tr>
<tr>
<td>C.</td>
<td></td>
</tr>
<tr>
<td>D.</td>
<td></td>
</tr>
</tbody>
</table>

So, with caching, we finally arrive at the answer:

Security | Speed
--------|-------
no change |
Agenda

**Virtual Memory**
- Virtual vs. physical memory
- Page tables
- Page faults

**Storage and Locality**
- The storage hierarchy
- Spatial and temporal locality
- Caching

**Effective Caching**
- Block size
- Eviction policy
- Order of operations
## Cache Block Size

Large block size:
- do data transfer less often
- take advantage of spatial locality
- longer time to complete data transfer
- less advantage of temporal locality

Small block size: the opposite

Typical: Lower in pyramid ⇒ slower data transfer ⇒ larger block sizes

<table>
<thead>
<tr>
<th>Device</th>
<th>Block Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>8 bytes</td>
</tr>
<tr>
<td>L1/L2/L3 cache line</td>
<td>128 bytes</td>
</tr>
<tr>
<td>Main memory page</td>
<td>4KB or 64KB</td>
</tr>
<tr>
<td>Disk block</td>
<td>512 bytes to 4KB</td>
</tr>
<tr>
<td>Disk transfer block</td>
<td>4KB (4096 bytes) to 64MB (67108864 bytes)</td>
</tr>
</tbody>
</table>
## Cache Management

<table>
<thead>
<tr>
<th>Device</th>
<th>Managed by:</th>
</tr>
</thead>
</table>
| Registers (cache of L1/L2/L3 cache and main memory)| Compiler, using complex code-analysis techniques  
Assembly lang programmer |
| L1/L2/L3 cache (cache of main memory)              | Hardware, using simple algorithms                                            |
| Main memory (cache of local sec storage)           | Hardware and OS, using virtual memory with complex algorithms  
(since accessing disk is expensive)                 |
| Local secondary storage (cache of remote sec storage) | End user, by deciding which files to download                                 |
Cache Eviction Policies

**Best** eviction policy: “oracle”
- Always evict a block that is never accessed again, or...
- Always evict the block accessed the *furthest in the future*
- Impossible in the general case

**Worst** eviction policy
- Always evict the block that will be accessed next!
- Causes **thrashing**
- Impossible in the general case!
Reasonable eviction policy: **LRU policy**

- Evict the “Least Recently Used” (LRU) block
  - With the assumption that it will not be used again (soon)
- Good for straight-line code
- (can be) bad for (large) loops
- Expensive to implement
  - Often simpler approximations are used
  - See Wikipedia “Page replacement algorithm” topic
Locality/Caching Example: Matrix Multiplication

Matrix multiplication
• Matrix = two-dimensional array
• Multiply n-by-n matrices A and B
• Store product in matrix C

Performance depends upon
• Effective use of caching (as implemented by system)
• Good locality (as implemented by you)
Two-dimensional arrays are stored in either **row-major** or **column-major** order.

C uses **row-major** order
- Access in row order ⇒ good spatial locality
- Access in column order ⇒ poor spatial locality

### Locality/Caching Example: Matrix Mult

<table>
<thead>
<tr>
<th></th>
<th>a[0][0]</th>
<th>a[0][1]</th>
<th>a[0][2]</th>
<th>a[1][0]</th>
<th>a[1][1]</th>
<th>a[1][2]</th>
<th>a[2][0]</th>
<th>a[2][1]</th>
<th>a[2][2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>19</td>
<td>21</td>
<td></td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td></td>
<td></td>
<td>20</td>
<td>24</td>
<td></td>
<td>23</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>22</td>
<td>23</td>
<td></td>
<td>22</td>
<td>25</td>
<td></td>
<td>23</td>
<td></td>
</tr>
</tbody>
</table>
Locality/Caching Example: Matrix Mult

for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            c[i][j] += a[i][k] * b[k][j];

Reasonable cache effects
• Good locality for A
• Bad locality for B
• Good locality for C
Locality/Caching Example: Matrix Mult

for (j=0; j<n; j++)
  for (k=0; k<n; k++)
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * b[k][j];

Poor cache effects
- Bad locality for A
- Bad locality for B
- Bad locality for C
Locality/Caching Example: Matrix Mult

for (i=0; i<n; i++)
  for (k=0; k<n; k++)
    for (j=0; j<n; j++)
      c[i][j] += a[i][k] * b[k][j];

Good cache effects

- Good locality for A
- Good locality for B
- Good locality for C
Another ghost of exams past …

Suppose that C laid out arrays in column-major order instead of row-major order.

What would be the most efficient loop ordering for matrix multiplication to maximize performance through good locality?

A. \( i \ k \ j \) (Same as row-major)

B. \( i \ j \ k \)

C. \( j \ k \ i \)

D. \( j \ i \ k \)

E. \( k \ i \ j \)

F. \( k \ j \ i \)

Exactly what makes this bad for all three in row-major makes it ideal for column-major:

- a and c have good spatial
- b has good temporal, spatial

```c
for (i=0; i<n; i++)
    for (k=0; k<n; k++)
        for (j=0; j<n; j++)
            c[i][j] += a[i][k] * b[k][j];
```
Next time ...

Getting started with ARM!