Virtual Memory, Storage Hierarchy, and Caching
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
Processes

Program
- Executable code
- A static entity

Process
- An instance of a program in execution
- A dynamic entity: has a time dimension
- Each process runs one program
  - E.g. the process with Process ID 12345 might be running emacs
- One program can run in multiple processes
  - E.g. PID 12345 might be running emacs, and PID 23456 might also be running emacs – for the same user or for a different user
Processes: Two Key Illusions

1. Processes believe they have *private control flow* (i.e., they own the whole CPU, all the time)
   - We’ll discuss control flow and process control in a later lecture

2. Processes believe they have a *private address space* (i.e., they own all the memory that the machine has or could have)

Process is a profound abstraction in computer science
Each process sees main memory as
Huge: $2^{64} = 16 \text{ EB (16 exabytes)}$ of memory $\approx 10^{19} \text{ bytes}$
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 (≈ in a file)
- 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>virtual addr</th>
<th>virtual page num</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>48 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>physical addr</th>
<th>physical page num</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>48 bits</td>
<td>16 bits</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
ArmLab Virtual & Physical Addresses

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
Storing Page Tables

Question
• Where are the page tables themselves stored?

Answer
• In main memory

Question
• What happens if a page table is swapped out to disk??!!!

Answer
• It hurts! So don’t do that, then!
• OS is responsible for swapping
• Special logic in OS “pins” page tables to physical memory
  • So they never are swapped out to disk
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)
Page Faults

- 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
Let’s start by considering security...

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

OK, so part of the answer is:

Security

But what about speed?
Revisiting Page Tables…

Question
• Doesn’t each logical memory access require two physical memory accesses – one to access the page table, and one to access the desired datum?

Answer
• Conceptually, yes! (And page tables are stored hierarchically as trees, so it can be even worse than 2 accesses!)

Question
• Isn’t that inefficient?

Answer
• Not really…
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

- **Smaller**
  - **Faster**
  - **Cheaper**
  - storage devices

- **Larger**
  - **Slower**
  - **Cheaper**
  - storage devices

- **Registers**
  - Hold words retrieved from L1/L2/L3 cache

- **L1 cache**
  - Hold cache lines retrieved from main memory

- **Level 2 cache**
  - Hold cache lines retrieved from main memory

- **Level 3 cache**
  - Hold cache lines retrieved from main memory

- **Main memory (RAM)**
  - Hold disk blocks retrieved from local disks

- **Local secondary storage**
  - (local disks, SSDs)
  - Hold files retrieved from disks on remote network servers

- **Remote secondary storage**
  - (distributed file systems, Web servers)
  - Hold files retrieved from remote network servers
Typical Storage Hierarchy

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
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
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

1 ns

1 μs

1 ms

DRAM

HDD

SSD

Kb  Mb  Gb  Tb
Typical Storage Hierarchy

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

- **Latency**: tens of milliseconds
  - Limited by the speed of light (and 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
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

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

Typical code
(good overall 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++` (which are the next machine language instructions) 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
Caching in a Storage Hierarchy

Level k:

```
| 4 | 9 | 10 | 3 |
```

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

Blocks copied between levels

Level k+1:

```
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
```

Larger, slower device at level k+1 is partitioned into blocks
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?

Security  | Speed
---|---
A. | ↑  |
B. | ↓  |
C. | ↑  |
D. | ↓  |

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

Security  | Speed
---|---
          | ↑
often little or 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
Here’s a real question from an old exam:
For caching in a memory hierarchy, what is the best motivation for a larger cache block size?

A. Temporal Locality
B. Spatial Locality
C. Both
D. Neither

Spatial locality makes use of subsequent data after a given read, so having more data to keep reading is a win.
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 Multiplication

**row-major**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
</tbody>
</table>

**col-major**

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

```c
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 Multiplication

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
Good cache effects

- Good locality for A
- Good locality for B
- Good locality for C

```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];
```
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
Next time ...

Getting started with ARM!