University of Glasgow | School of Computing Science

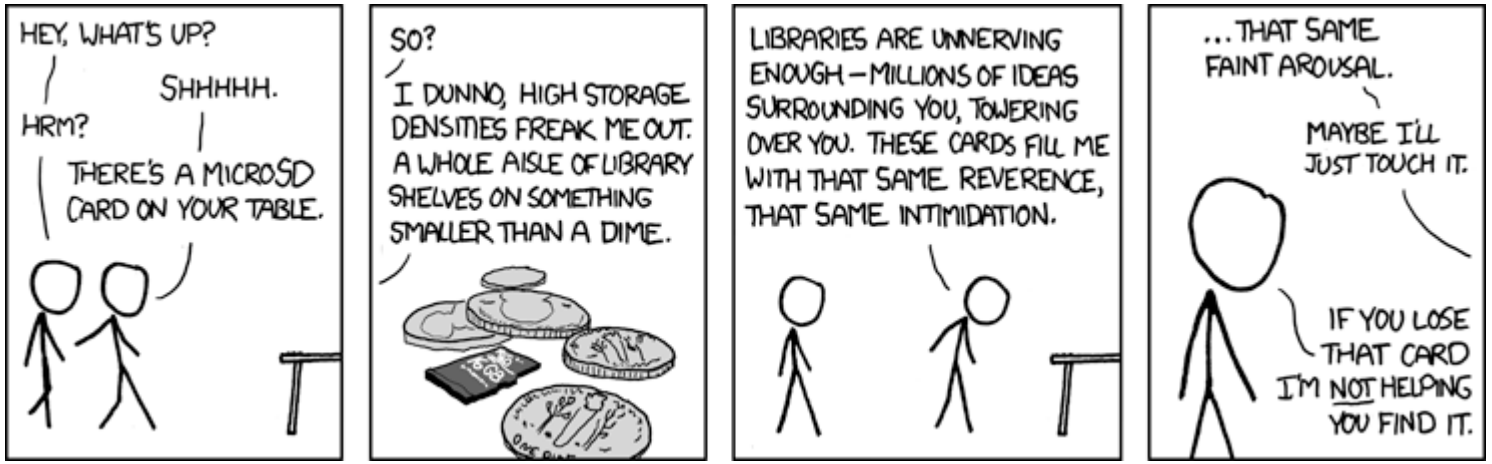
Networks & Operating Systems Essentials

Dr Angelos Marnerides

<angelos.marnerides@glasgow.ac.uk>

School of Computing Science

Coming up next…



@https://xkcd.com/691/

Memory Management Considerations

- Need to make memory accesses fast
- Need to safeguard memory allocated to different processes

Aside: RAM types

- Static RAM (SRAM)
  - Built using flip-flops made out of multiple transistors
  - Very fast but expensive → often found in CPUs/caches
- Dynamic RAM (DRAM)
  - Built using transistors and capacitors
  - Slower but denser than SRAM → often found in main memory chips
  - Dynamic → charge is lost over time, needs refreshing
- Synchronous Dynamic RAM (SDRAM)
  - Just like DRAM, but synchronised with the CPU for faster access

Refresher on Computer Architecture

- Main memory and registers are the only storage the CPU can access directly
  - Programs must be loaded (from disk) into memory for them to be executed
  - Code and data need to be loaded from RAM to CPU registers before they can be decoded/executed/operated on
  - A register access takes one(-ish) CPU clock cycle, while main memory accesses can take many cycles
- Main memory is often structured/accessed in multi-byte words
  - We treat our RAM as consisting of 2^c-byte words instead of single bytes
    - E.g., a 16-byte RAM using 4-byte words, would be structured as 4 words, at byte addresses 0, 4, 8 and 12, and word addresses 0, 1, 2, 3
  - If a is a multiple of 2^c, a memory access to an address in $[a, a + 2^c - 1]$ will actually refer to the same word
  - A load from address b will result in fetching all memory locations in $[2^c * \text{floor}(b/2^c), 2^c * \text{floor}(b/2^c) + 2^c - 1]$

Caching

- Observation: program code and data exhibit spatial and temporal locality
  - Spatial: loops in code, sequential access to data
  - Temporal: likely that the same data/instructions will be reused before long
- The aim: Make the common case fast
  - Introduce two memories:
    - The primary memory is large, but slow
    - A cache memory is fast enough to match the processor speed but it has to be small
  - Keep recently accessed data (incl. instructions) in the cache
  - Not all of RAM can fit in the cache…
    - Must make sure that test for cache residency is fast
  - … but there is a high probability that a memory access will refer to data already in the cache — so it will be fast

Caching

- Implementation issues:
  - Amount of data in a cache location: *cache line size*
  - Given an address, determine whether the data is in the cache and where
  - If not, determine where to put the data in the cache, after memory fetch
  - If cache is full, determine which lines to overwrite: *cache replacement policy*

Caching

- Each location in the cache contains three parts:
  - A word of data
  - A valid bit (set if data is not empty, unset otherwise)
  - A tag: the actual address of this data in the primary memory
- On every memory access, the hardware checks the address against the tags in the cache in parallel
  - If the address refers to data in the cache, it is a cache hit
    - The data is retrieved quickly, at processor speed
  - Otherwise it's a cache miss
    - An access to primary memory is performed, and the result is placed in the cache for future reuse
    - The processor waits for the slow memory access to complete

Caching and Memory Stores

- Assume a store instruction changes only the cache entry, but not the respective memory location
  - Then the cache and memory become inconsistent
  - A cache miss will require the cache entry to be written out later
- In write-through caches, a store changes both the cache entry and the memory
  - To reduce the cost of the memory operation, the data may be buffered in registers
- In write-back caches, a store changes only the cache entry
  - The cache is allowed to become inconsistent with the memory
  - A cache miss requires the replaced cache entry to be stored in RAM (if it has been modified)
  - Write-back may reduce the number of memory stores, and improve performance

Cache Lines

- In practice, we don't just keep individual memory locations (bytes or words) in the cache
- The cache is organised in "large words" called cache lines
  - 2^c words, at an address which is a multiple of 2^c
    - E.g. 16 bytes at an address which is a multiple of 16
  - If a is a multiple of 2^c, a memory access to an address in $[a, a + 2^c - 1]$ will actually refer to the same cache line
  - A cache miss for an address b will result in fetching all memory locations in $[2^c * \text{floor}(b/2^c), 2^c * \text{floor}(b/2^c) + 2^c - 1]$ and storing them in one cache line
- Different machines use different cache line sizes
  - Bigger cache lines increase the probability of cache hits
  - ... but also increase the penalty of a cache miss

Searching the Cache

- A crucial issue: Given an address in the main memory address space, locate where that address would be in the cache
  - The cache has to be searched on each memory access (need for speed!)
- Three main approaches:
  - Direct-mapped caches: each address can be mapped to only one cache line
  - Fully-associative caches: each address can be mapped to any cache line
  - *n*-way set-associative caches: each address can be mapped to any of a *set* of *n* cache lines
- A set-associative cache can subsume the other two
  - If only one line per set (n=1) → Direct mapped
  - If all cache lines in a single set → Fully associative

Searching the Cache

- Assume m-bit addresses, and a cache with 2^{λ} lines, each 2^{c} words wide
  - E.g., 8-bit addresses (m = 8), a cache with 8 lines (λ = 3) each being 4 words wide (c = 2)
- Each m-bit memory address is broken down into three parts:
  - Offset: the least significant c bits, if the cache lines are 2^{c} words wide
    - Accesses to addresses with the same most significant $m - c$ bits, refer to words in the same line
  - Set id: for a 2^{s}-way set associative cache, the next most significant $\lambda - s$ bits (as the cache then contains $2^{\lambda}/2^{s} = 2^{\lambda-s}$ sets)
    - 2-way set associative cache: s = 1 ➔ $\lambda - s$ = 2
    - 1-way set associative cache (direct-mapped): s = 0 ➔ $\lambda - s$ = 3
    - 8-way set associative cache (fully associative): s = 3 (= λ) ➔ $\lambda - s$ = 0
  - Tag: the (remaining) most significant $m - c - (\lambda - s)$ bits
    - 2-way set associative cache: $m - c - (\lambda - s)$ = 4
    - 1-way set associative cache: $m - c - (\lambda - s)$ = 3
    - 8-way set associative cache: $m - c - (\lambda - s)$ = 6
- Why use the most significant bits for the tag and the middle bits for the set id?

Searching the Cache

- On every memory access, the hardware:
  - Extracts the tag and set id from the memory address
  - Locates the set of cache lines for this address
  - Checks the address tag against the tags in the set entries in parallel
  - If the tag matches that of an entry **AND** its valid bit is set, it is a cache hit; otherwise, it is a cache miss

Set-Associative Cache

- Assume memory addresses are 8 bits wide ([b7,b6,...,b0], b7: MSB, b0: LSB)
- Assume we have a cache with 8 lines, each 4 words wide
  - 4 words per line ➔ 2-bit offset
- Assume we have a 2-way associative cache (i.e., 2 cache lines per set)
  - 8/2 = 4 sets ➔ 2-bit set id
  - 8-bit addresses ➔ 8 - 2 - 2 = 4-bit tag
- [b7,b6,b5,b4,b3,b2,b1,b0]
- Access memory position 138
  - 138 = 10001010
- Access memory position 41
  - 41 = 00101001
- Access memory position 43
  - 43 = 00101011
- Access memory position 136
  - 136 = 10001000

line	set	tag	valid	data
(0) 000	00		0	
(1) 001	00		0	
(2) 010	01		0	
(3) 011	01		0	
(4) 100	10	1000	1	*{11,10,01,00}*
(5) 101	10	0010	1	*{11,10,01,00}*
(6) 110	11		0	
(7) 111	11		0	

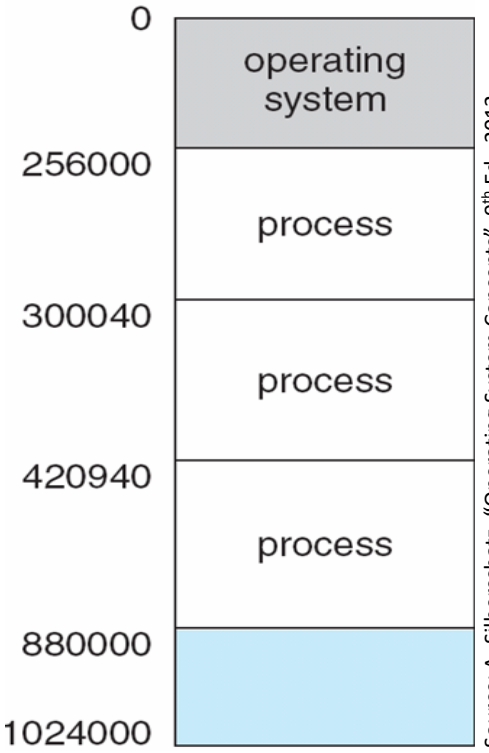
Computer Architecture Refresher (pt. 2)

- Lifecycle of a program:
  - High level language source code ➔(compiler) Assembly code ➔(assembler) Object code ➔(linker) Executable code ➔(loader) Runnable code in memory

- Think Sigma16
  - Code assumed to be loaded at memory position 0
  - What if there were multiple modules in different source code files?
    1. Each would be compiled as if to be loaded at memory position 0
    2. All files would then be concatenated into one, with the main module first
    3. Each module, apart from the main module, would then be relocated
       - Compute its starting address in concatenated file, increment all internal addresses by that
    4. Then all external names (variables, functions, etc.) are resolved and replaced with their actual address
    5. Then the resulting file is saved, ready to be executed
  - Also known as: static linking
    - Dynamic linking: same, only without step 5, and steps 2-4 only happen at run time

Memory address space

- Each process is assumed to have its own *address space*
  - A region of memory locations
  - Usually split in two parts:
    - Program segment: contains code, is read-only, may be shared across processes
    - Data segment: contains variables/data, usually is read-write, should only be accessible by its owner process
  - The lowest address in an address space is called its origin
- When multiple processes are running, their code is loaded at an arbitrary location in memory; what then?
  - What happens to the addresses in the code?



Source: A. Silberschatz, "Operating System Concepts", 9th Ed., 2012.

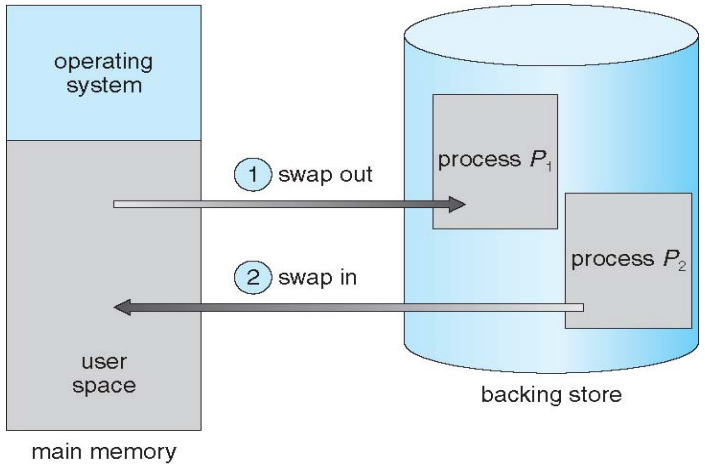
Aside: Segmentation

- Models exist where the address space is split into more than just two segments; e.g.:
  - Code segment
  - Global variables
  - Heap
  - Stacks (one per thread)
  - External libraries

Memory address space

- Could have loader re-relocate all of the addresses after program loaded in RAM
  - Would take more time to load the program
  - Would need to be repeated if process is swapped out/in
- We still haven't solved the memory protection problem
  - What if a program accesses memory locations beyond its own?

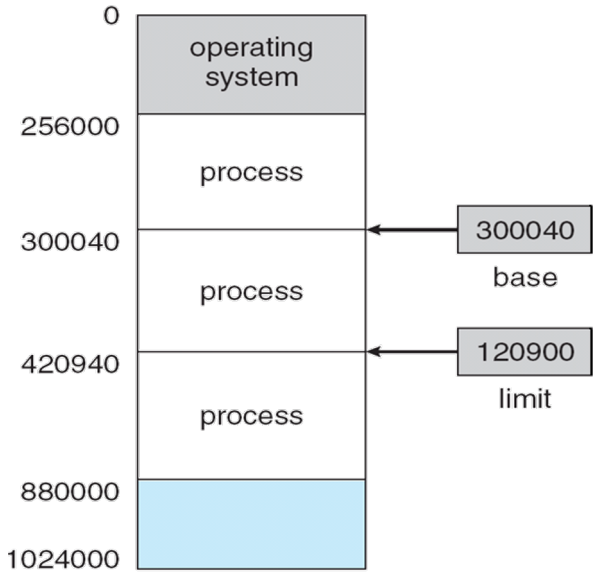
- Brainstorm!



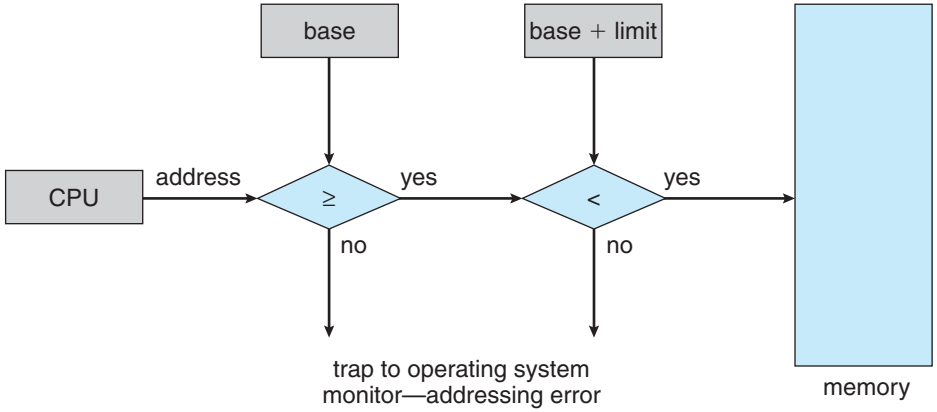
Source: A. Silberschatz, "Operating System Concepts", 9th Ed., 2012.

Base Register and Limit Register

- Idea #1: Let's use special registers to store the first and last address in a process's address space
  - Base Register (BR) and Limit Register (LR)
  - Add BR's value to the memory addresses of all memory accessing instructions
  - Check that the resulting memory address is not beyond LR
  - Only the OS can set/update the values in BR and LR
- A virtual address space!



Source: A. Silberschatz, "Operating System Concepts", 9th Ed., 2012.



Source: A. Silberschatz, "Operating System Concepts", 9th Ed., 2012.

Memory Management

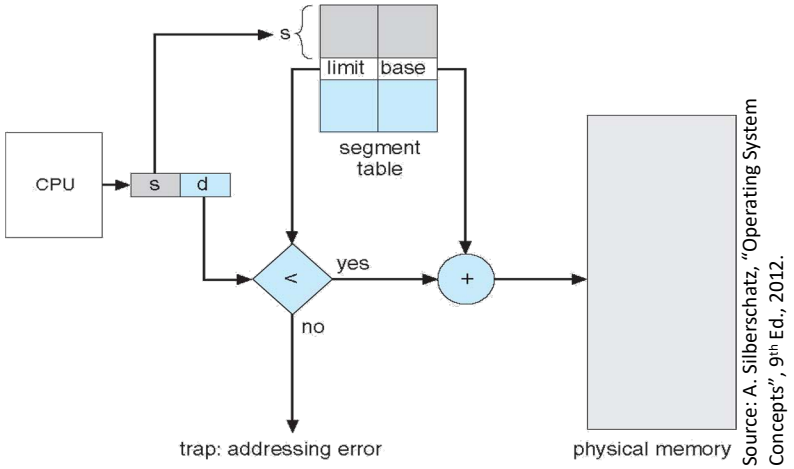
- Idea #2: Partition RAM into fixed size partitions, allocate one to each running process
  - Would work, but is inflexible and clunky
  - Creates internal fragmentation: space within partitions goes unused
- Idea #3: Variable-sized partitions:
  - The OS keeps track of lists of allocated ranges and "holes" in RAM
  - When a new process arrives, it blocks until a hole large enough to fit it is found
  - Several strategies available:
    - First fit: use the first hole that is big enough for the process
    - Best fit: use the smallest hole that is big enough for the process
    - Worst fit: use the largest hole
  - If hole is too large, it's split in two parts: one allocated to the new process, one added to the list of holes
    - Two or more adjacent holes may be merged
    - The OS may then check whether the newly created hole is large enough for any waiting processes
  - Can create external fragmentation: space in between partitions too small to be used

Memory Management

- So far, we've assumed that a process's memory address space is allocated as a big contiguous space in RAM
  - Is this realistic? Discuss…
  - What could we then do to allow non-contiguous allocation of memory locations?

Memory Management

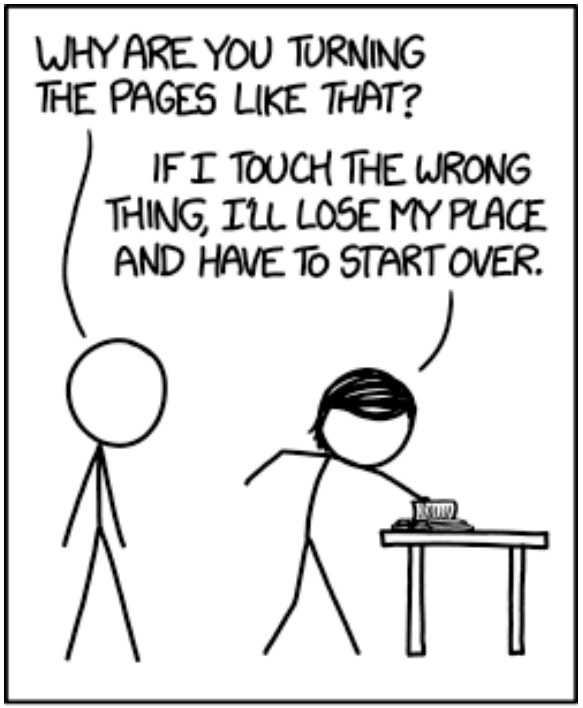
- Idea #4: Segments
  - Maintain several "specialised" segments
  - Maintain a table with info for each segment
    - Base: Starting address for that segment
    - Limit: Size of the segment
  - ➢ Extension of BR/LR but to multiple mini address spaces (segments)



Source: A. Silberschatz, "Operating System Concepts", 9th Ed., 2012.

  - … but how to map segments to "holes" in RAM?

Next week, on NOSE2



@https://xkcd.com/1309/

Recommended Reading

- Silberschatz, Galvn and Gagne, *Operating Systems Essentials, Chapter 1, section 1.8.3, Chapter 7, sections 7.3, 7.6*