Architecture and Systems

Stefan Schwoon

Cours L3, 2014/15, ENS Cachan
November 5, 2014

Error detection and correction

Needed in case data transmission is not reliable (e.g., network),
or due to electricall glitches (main memory!)

Error detection: detect whether an error has happened and flag it.

Error correction: automatically repair the error

Naturally, one cannot guard against all faults, so typically one indicates how many bits may be “flipped” at most to still guarantee detection/correction.

Parity

Idea: extend a data word with an additional bit p such that the total number of 1-valued bits in the word is even.

p equals the modulo-2 (XOR) sum of the bits in the word.

Can detect errors where 1 bit is flipped.

No correction possible.

Hamming codes

Adds a logarithmic number of parity bits.

E.g., for 7 data bits, 4 parity bits are added (11 in total).

Data and parity bits are placed at positions starting from 1, e.g. 1..11.

Parity bits are placed at positions corresponding to a power of 2, i.e. 1,2,4,8,....
Let us call them $p_1, p_2, \ldots$.

Now, p_i acts as parity bit for all positions k such that in the binary encoding for k, the bit for i is 1.

E.g., p_1 is parity for odd positions; p_2 for 2,3,6,7,...; etc.

Example: Data word 1011001. Scheme: $p_1 p_2 1 p_4 001 p_8 001$.

In the example, $p_1 = 1$, $p_2 = 0$, $p_4 = 0$, $p_8 = 1$.

Result: 10100111001

In case of a single error: faulty position can be identified by combination of faulty parity bits.

Double error: Indistinguishable from single error.
$\Rightarrow$ add an additional, overall parity bit at position 0 to detect (but not repair) double errors.

Memory

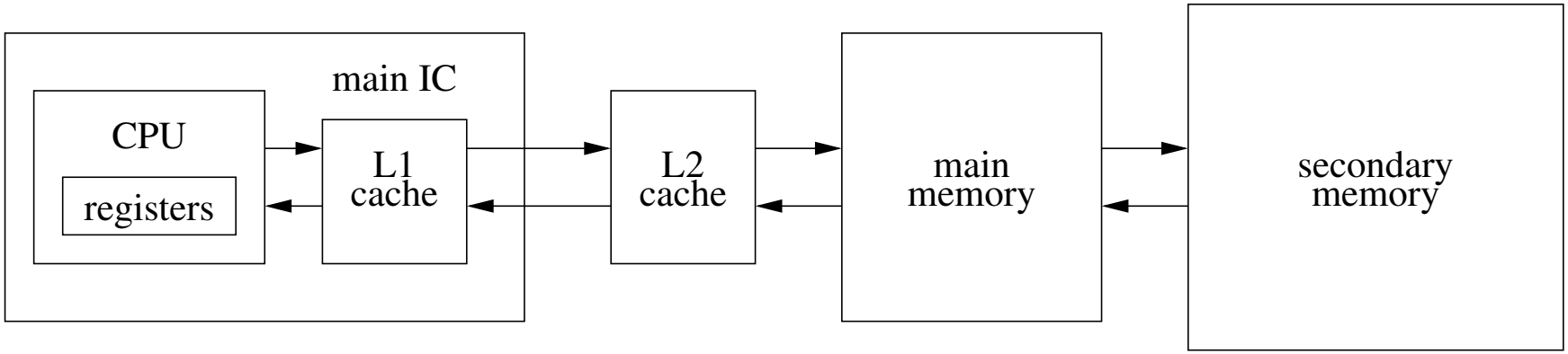
Hardware / architecture aspects:

- different types of memory
- physical realization of memory access

Software / systems aspects:

- sharing / security aspects
- hide physical details from user / programmer

Types of memory

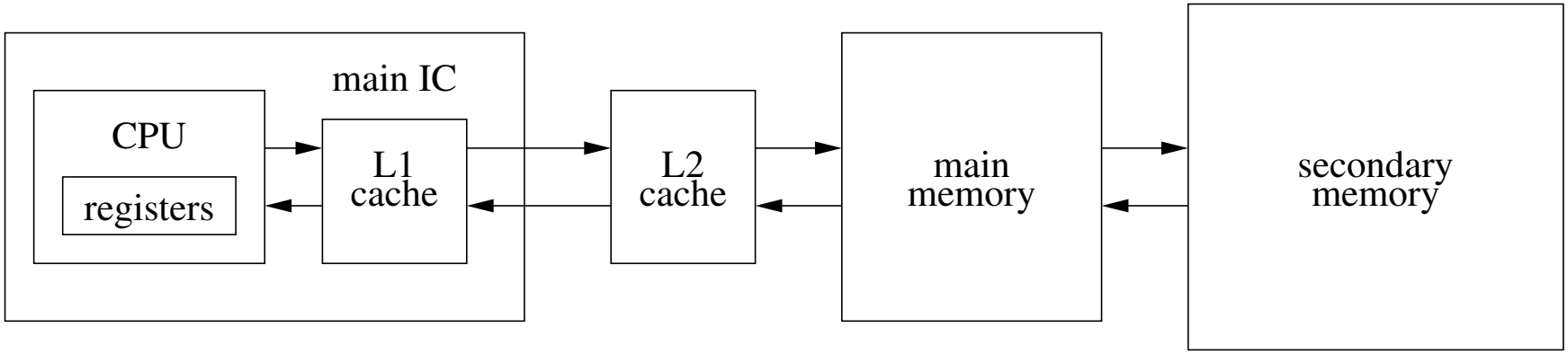


Registers + main memory: directly adressable by programmer

Cache: for speed-up, managed by hardware

Secondary memory: to extend memory, managed by operating system

Types of memory



Typical laws:

faster access = higher cost

faster memory types have smaller capacity

Goal: Speed up access + maximize available memory + minimize cost

Memory characteristics

Access time (to make memory contents available to processor):

- depends on physical characteristics and distance to CPU (signal runtime)

Access mode:

- read-write / read-only access
- random / serial / blockwise access

Volatility:

- memory contents are lost / preserved when power goes off

Example: SRAM

SRAM (static random-access memory):

- realized, e.g., by D-latches or similar
- fast access, depends on runtime of signals
- several transistors per bit
- typically used to realize fast memories (caches)

Example: DRAM

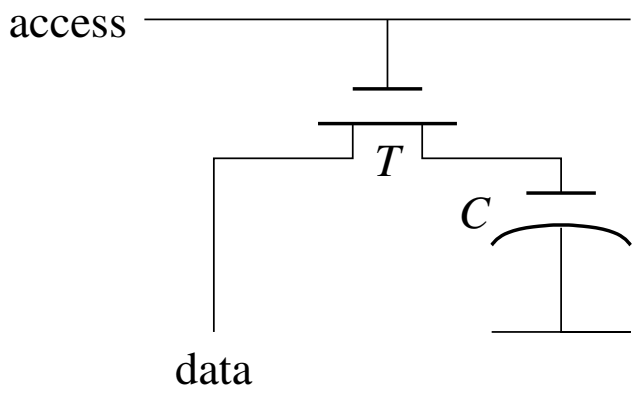
DRAM (dynamic RAM):

realized by one transistor + one capacitor; charged transitor = “1”

activating the access line permits to charge capacitor via data line (or transfer capacitor charge to data line)

reading operation is destructive – restore value after reading

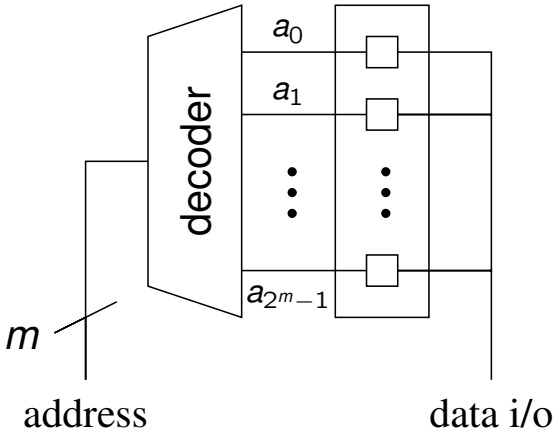
more compact than SRAM but slower; typically used for main memory

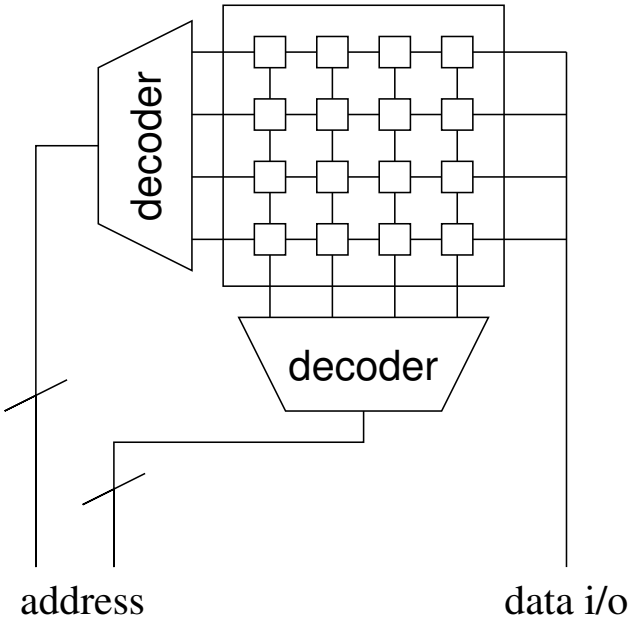


Organization of a RAM

SRAM and DRAM are examples of volatile, read-write, random-access memory

One-/two-dimensional addressing:





Two-dimensional addressing cheaper to realise.

Organization of a RAM

Space constraints limit the storage capacity of individual chips

Memory therefore distributed over several chips

Interleaving adressing: with n chips, chip i stores addresses a where $a \equiv i \pmod{n}$; allows to recover n subsequent addresses at once.

Example: Hard drive

can store huge amount of data

permanent storage, non-volatile

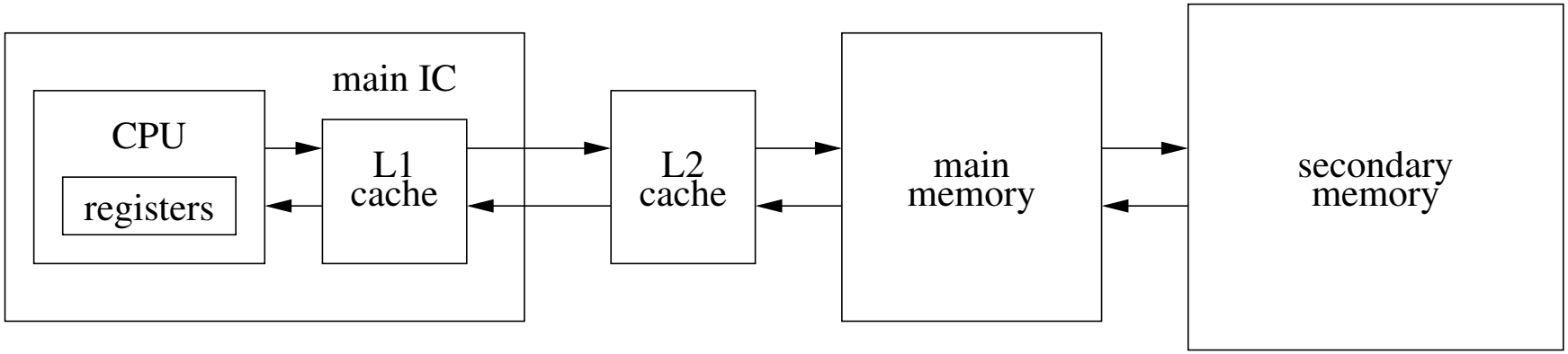
access (relatively) slow

block access:

- one hard drive may consist of several disks, on top of one another;
- each disk subdivided into tracks
- each track subdivided into sectors
- one sector may contain, e.g., 1K or 4K of data
- read/write accesses to one sector at a time

used to realize secondary memory

Caching



Idea: keep memory that is currently used often in the faster types of memory

Done transparently: Programmer refers to an address in main memory,
the actual content is there *or* in one of the caches

Based on *locality assumption* on code and data

Caching: Address decomposition

For caching, the memory is subdivided into blocks (so-called *pages*) of, say, 4 kilobytes.

In this case, an address A can be (in binary) written as $B.D$, where D comprises the lowermost 12 bits.

B then identifies the block (page), D the *deplacement* within that block.

The cache memorizes which main-memory pages are currently in cache. For any access to $B.D$, one first checks whether B is currently in cache; main memory is consulted only when necessary.

Caching algorithms: discussed later

Memory management in single-tasking systems

Example: MS-DOS (single-user, single-tasking system), running on 808x CPU

Flat memory model, direct memory access (programs specify *physical* main memory addresses they want to access).

OS tells programs which parts of memory are available; no actual control over usage, no isolation, every “process” can crash the entire system.

Memory space 1MB (640 K for user processes), but only 16-bit registers.
Solution: segmentation

Advanced memory management

The architecture of MS-DOS (and other comparable OSs of its time) was due to the hardware it was designed for, which did not allow for memory protection.

Meaningful multi-user, multi-tasking systems require CPU architectures that support protected modes and virtual memory.

Protected mode means that different privileges can be assigned to certain memory segments:

> Only code in privileged memory segments can communicate directly with the hardware or execute certain critical instructions.
>
> Initially, the system boots in *unprotected mode*; the OS can then set up the initial memory segments.

Virtual memory

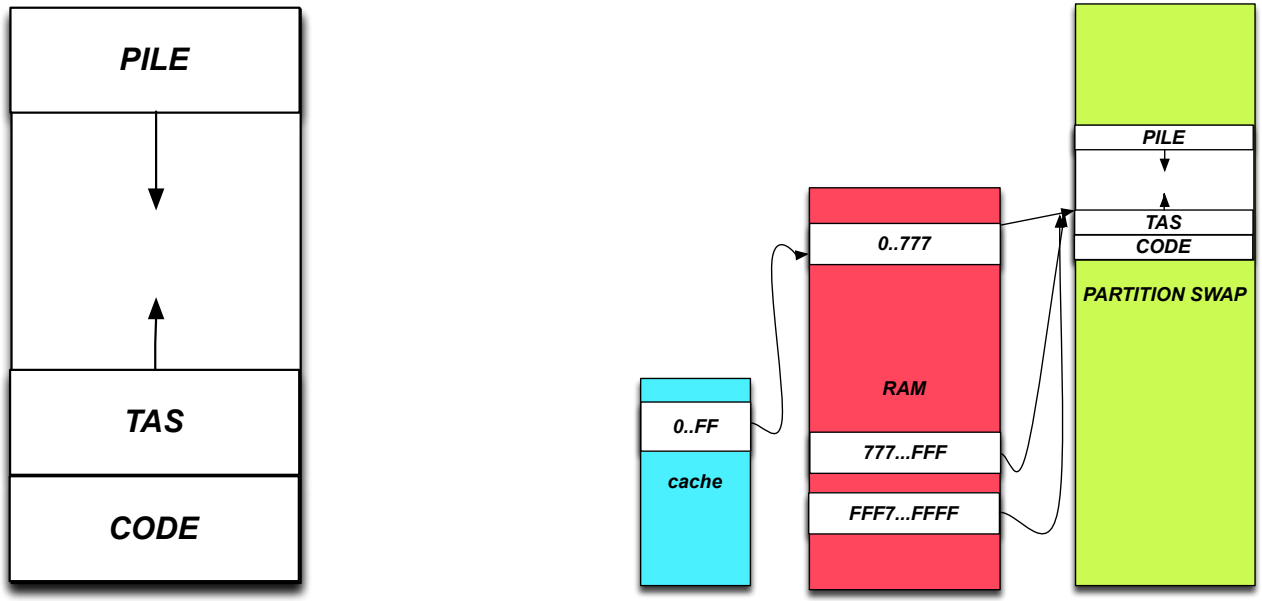
Modern architectures provide the concept of “virtual” memory.

Here, memory accesses are to *virtual* addresses, which the hardware translates into physical addresses.

Thus, user-level code in the system can be placed in an environment where it is unable to access data (or code) it is not meant to access.

Virtual memory

Virtual (left) and real memory (right):



Virtual memory

Different *processes* in the system can live in different virtual memories. Thus, the virtual environment is changed whenever the CPU executes a different process.

Different pages in virtual memory may point to the same real memory page; e.g., used for sharing libraries.

A process can tell the system how much heap memory it wishes to use (via the `brk(2)` system call) – low-level interface, not normally used by programmers.

Virtual memory

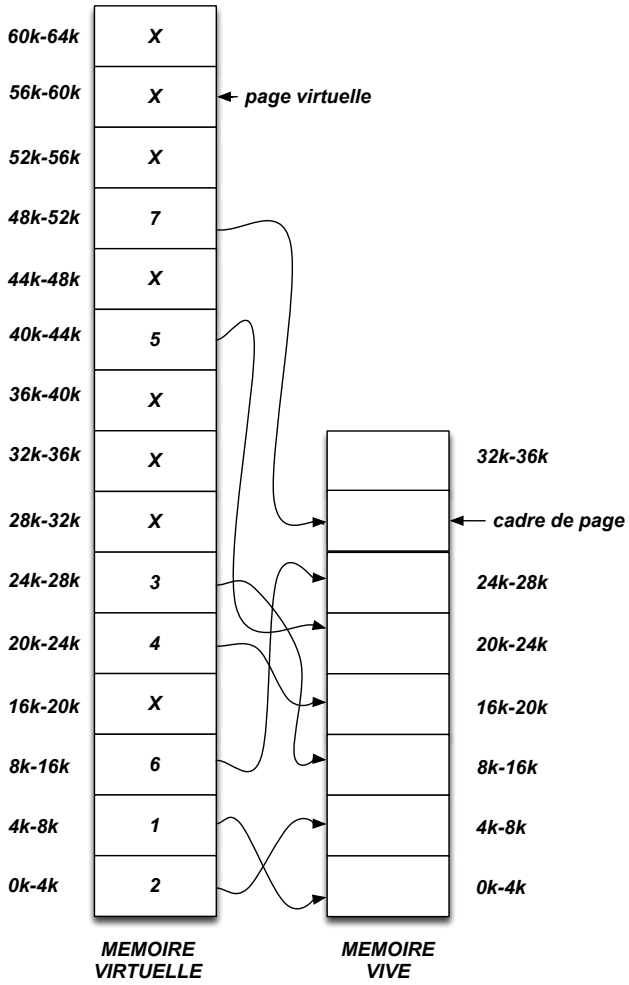
We now discuss the organisation of memory on the level of the hardware and the system.

Modern machines employ a memory management unit (MMU), a part of the CPU that is consulted for each memory access.

The MMU implements a paging mechanism:

> Memory is divided into pages of a fixed size, which is a power of two.
>
> For each process, there exists a page table, which contains, for every (used) entry in virtual memory, a descriptor.
>
> That descriptor contains information where the virtual page is stored physically. (In the easiest case, a physical memory address.)

Visiualization of virtual and real address space:



Paging

Basic idea:

At each memory access, the MMU is consulted, which uses the page table of the current process.

Each memory address is divided into its (virtual) *page* and its *offset*.

If the page currently is valid and resides in real memory, the virtual page is replaced by the real page, and memory access continues.

If the page is not currently in real memory, operation is suspended and the operating system is invoked to charge it.

Page size affects granularity of memory allocation and size of page table! (conflicting goals; reasonable compromise 4 to 32 K)

Problem no.1: Page tables too big to hold in MMU's own memory – store in main memory.

Problem no.2: Even main memory is not enough for naïve implementation – use tricks, e.g. several levels of indirection.

Problem no.3: Naive implementation means that every memory access is slowed down – MMU-internal TLB (translation lookaside buffer) as cache.

Further solutions: multiple levels of indirection, reverse table lookup

Swap space

Virtual memory may be bigger than real memory (in fact, it usually is. . .)

Combined memory usage of all processes may exceed available real memory.

Solution: store “rarely” used pages on hard-drive, mark virtual page entries appropriately.

Access to a “swapped” page results in an interrupt/exception handled by the OS, which ensures that the page will be loaded.

Swapping

Conclusion from previous slides: Memory access is not “constant-time”!
Multiple levels of caching: L1, L2 (in the CPU), TLB (in the MMU), copy-on-write, swapping, . . .

$\Longrightarrow$ caching algorithms important on multiple levels.

Problem in general: store “frequently used” objects (from a large set) in limited space.

Algorithms: FIFO (bad!), Second Chance, NRU, LRU, . . .

FIFO caching

Assume a fixed number of slots available for storing pages.

When a page is requested that is already in the cache, return it directly.

Otherwise, load the requested page from (slower) memory, and replace the oldest slot in the cache by the newly requested page. (page fault)

Suffers from Belady's anomaly: More slots can mean more page faults.

> Example: Try the sequence 3 2 1 0 3 2 4 3 2 1 0 4 with three, then with four slots.

Second chance / Clock

Second chance: Modification/improvement of FIFO: Equip each slot with a “reference” bit that is set to 1 each time the page is requested.

On page fault: Throw out the oldest page if its reference bit is 0. Otherwise, set its reference bit to 0 and move it to the end of the list.

If all pages have been referenced, the oldest version will still get thrown out.

Clock: Implementation variant of Second Chance with circular list, with pointer to next free slot / oldest page.

LRU / NRU

Least recently used: Keep time of last access, on page fault throw out the page that was not accessed for the longest time.

Certain optimality results in terms of page fault ratio, but expensive.

Not recently used: Keep two bits with each page (referenced/modified); referenced bit is reset periodically.

Four classes of pages: 00 = not referenced/not modified, 01 = not referenced/modified etc, on page fault throw out a page from the lowest class possible.

Compromise between optimality and performance (w.r.t. LRU).