Architecture et Système

Stefan Schwoon

Cours L3, 2015/16, ENS Cachan
December 14, 2015

Memory

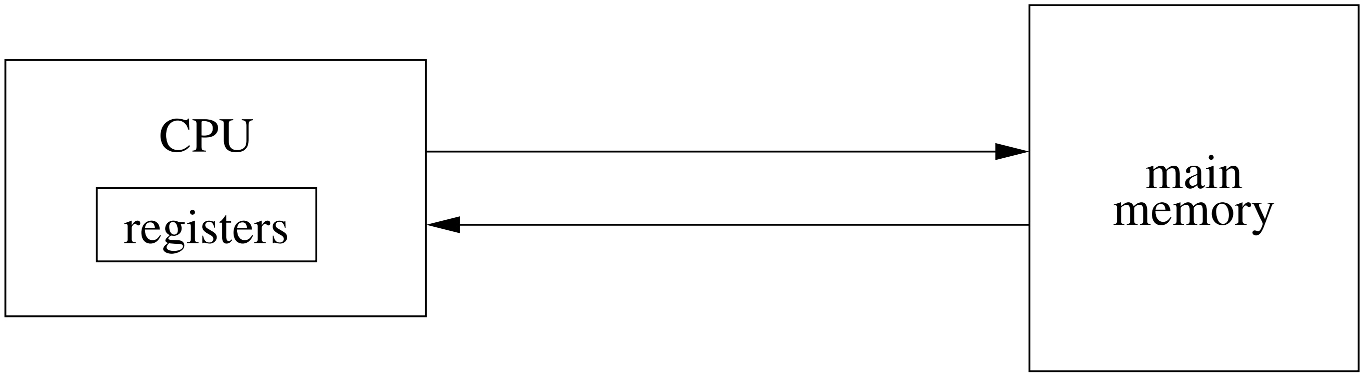
Hardware / architecture aspects:

- different types of memory
- physical realization of memory access

Software / systems aspects:

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

Memory: Programmer's view

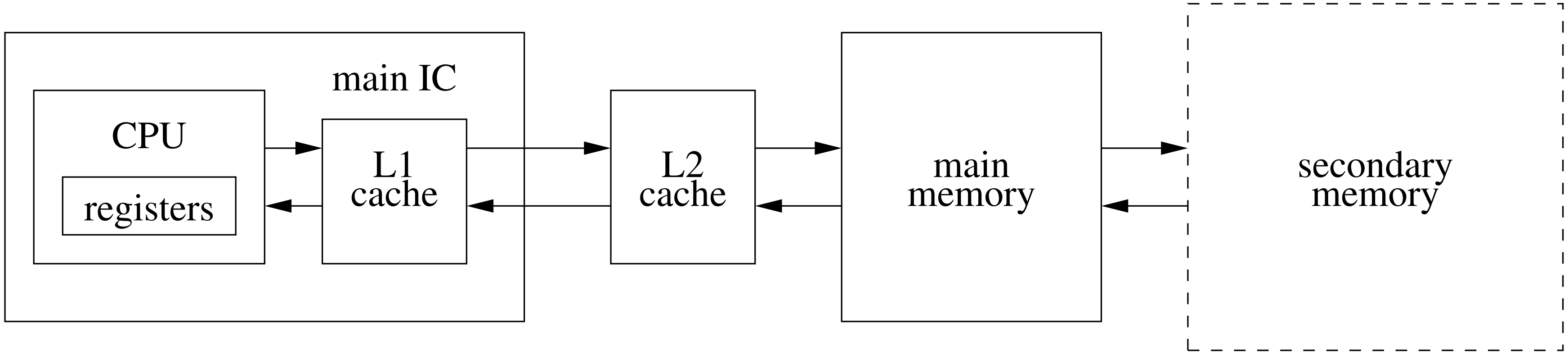


Registers: small memory for manipulating data

Main memory: big memory for storage

Inspired by earliest computer models (IAS etc)

Memory: Modern architecture

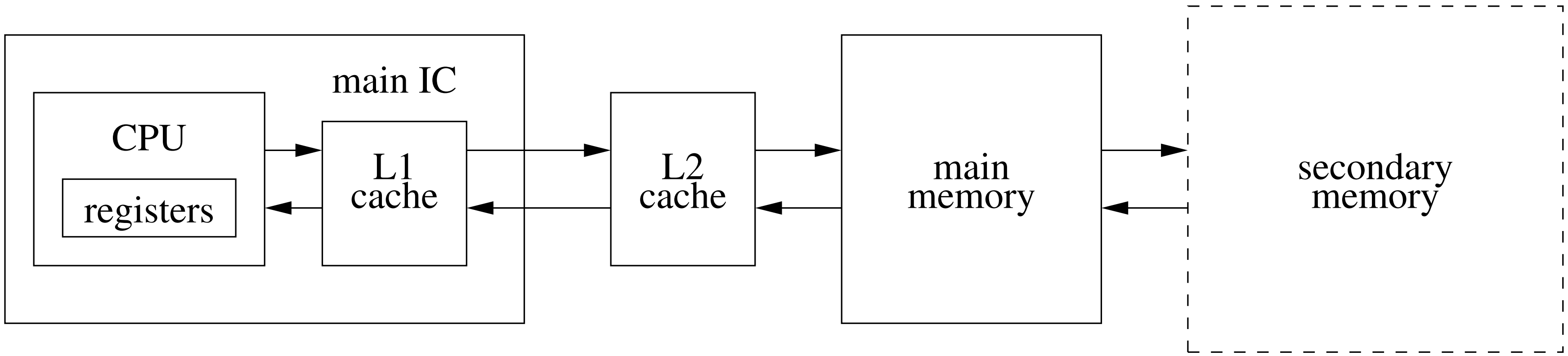


Registers + main memory: directly adressable by programmer

Cache: fast memory types for speed-up, managed by hardware

Secondary memory: to extend memory, managed by operating system

Types of memory



Typical laws:

faster access = memory that is more costly to produce/takes more space

→ faster memory types have smaller capacity

Goals: Speed up memory 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

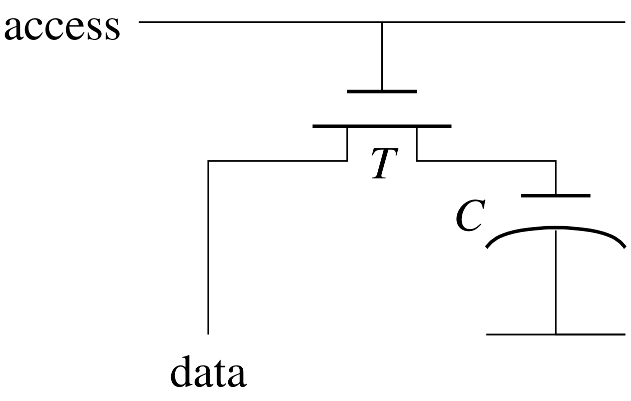
DRAM (dynamic RAM):

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

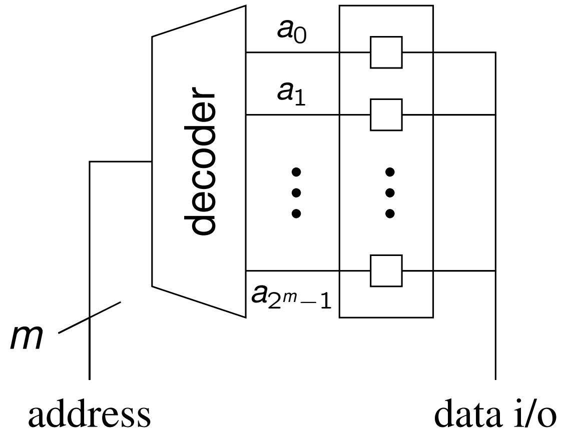
more compact than SRAM but slower; typically used for main memory

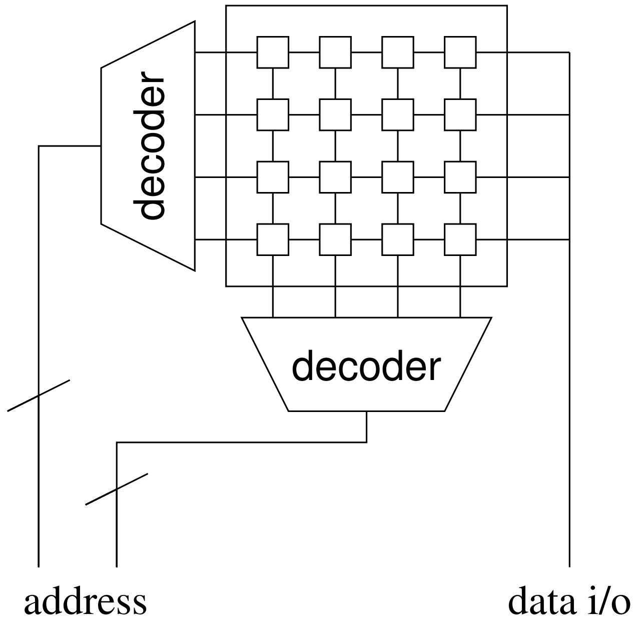


Organization of a RAM

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

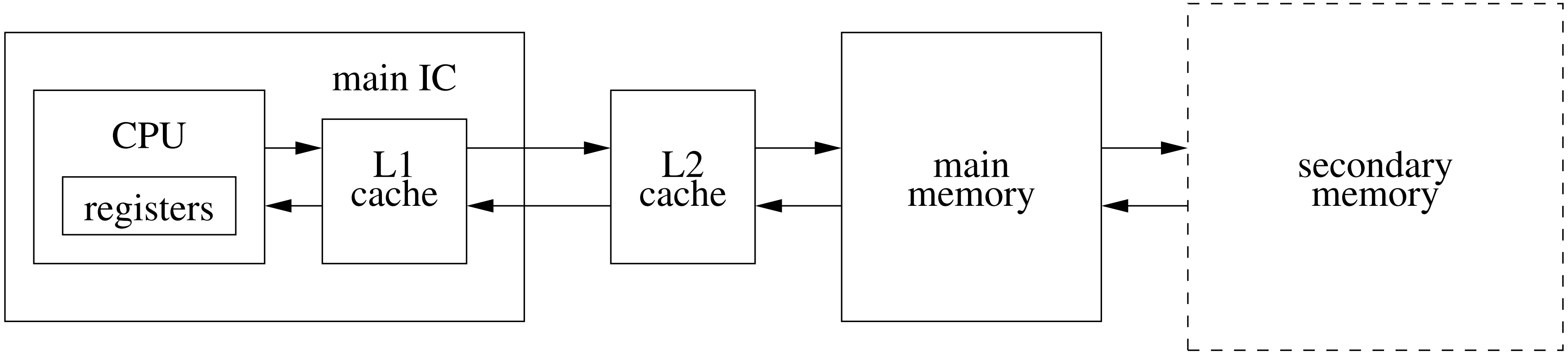
access (relatively) slow

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 simple (single-task) systems

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

Operating system tells programs which parts of memory are available.

Proper memory usage depends on discipline/goodwill of the programmers; the OS has no actual control over usage.

→ every program can crash the entire system.

Advanced memory management

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

General principle: memory partitioned into segments of equal size (e.g., several kB per segment)

Privileges: assigned to memory segments

- p.ex., only code in “privileged” memory can execute certain assembly instructions (communicated with hardware, other critical instructions)
- privileges can be changed, but only by provileged code
- possibly multiple levels of privileges, p.ex. for kernel and device drivers

Virtual memory: put processes into a “sandbox”

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

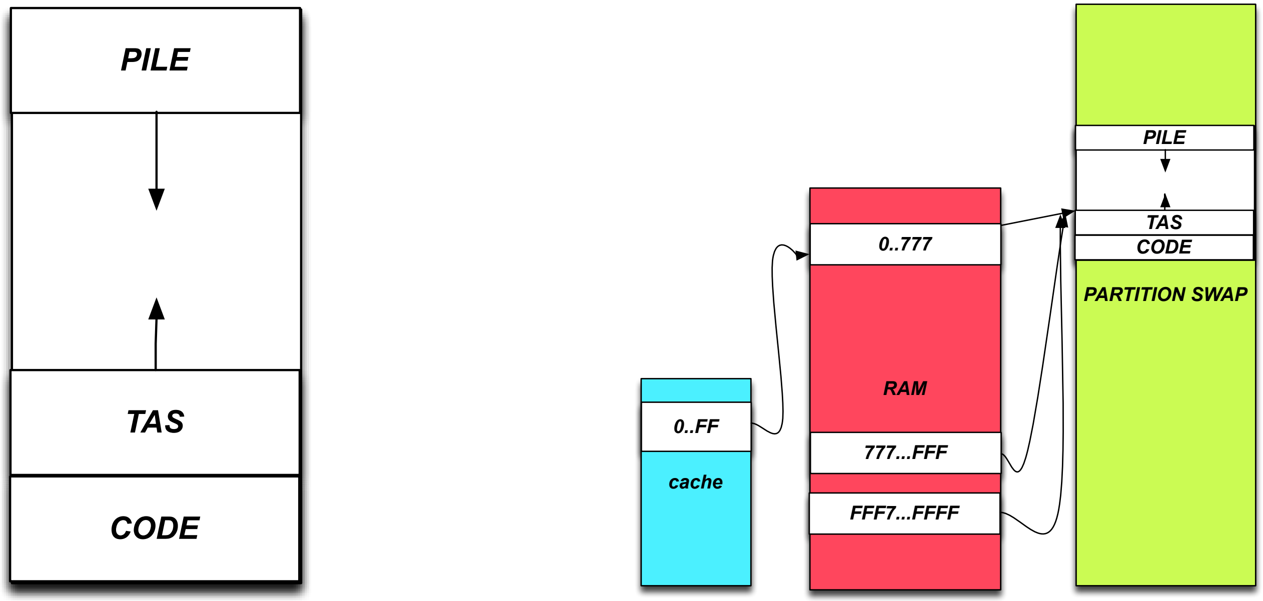
the mapping from virtual to physical addresses is such that the process can only access data or code it is meant to have access to.

virtual memory segments can have additional properties, e.g. *read-only*

advantage: OS does not intervene, except in special cases (e.g., illegal address, swapping)

Virtual memory

Virtual (left) and real memory (right):



Virtual memory

Different *processes* in the system can live in different virtual memories.

When the OS switches between processes, it exchanges the mapping from virtual to real addresses (the *page directory*).

Multiple 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 (virtual) heap memory it wishes to use (via the `brk(2)` system call) – low-level interface, not normally used by programmers.

System calls

POSIX standard does not specify details of memory management; judged too machine-dependent.

Still, to handle the virtual memory of a process, Unixes usually provide two functions:

`brk(2)` sets the size of the heap – in the virtual memory.
Direct interaction with the system, not normally used by the programmer.

`malloc(3)` (and similar routines) – process-level code that finds free space in the currently allocated heap space, will call `brk` if necessary.

Note: The OS only cares about the amount of memory requested by `brk`, which typically happens in large chunks.

How that requested memory is organized, is up to the user. Typically, `malloc` and co take over this organization and take care of fine-grained allocations of up to a few bytes. This is “invisible” on the system level.

We first regard how memory allocators work.

Memory management: Malloc

A memory allocator manages the dynamic allocation of memory in the heap.

main operations: malloc (allocate *n* bytes), free

user-level code contained in the standard library

uses brk to increase virtual memory when necessary

Other operations:

calloc (allocate and initialize to zero)

realloc (change size of allocation, moving if necessary)

mmap (map file contents into memory)

Conflicting goals:

- minimize time (operations should be fast)
- minimize space (fragmentation, overhead)
- maximize locality (improve cache performance)
- work well in all use cases

Error detection (catching problems caused by incorrect usage):

- very limited
- use memory beyond allocated limits (→ crash)
- double free (→ crash)
- crashes may occur thousands of lines later → nightmare to debug

Memory blocks

Memory allocators like `malloc` typically divide memory into chunks that are *free* or *occupied* (allocated).

Simple approach: fixed-size chunks with bitmask (0=free, 1=occupied). However, fixed-size chunks are impractical.

Better: chunks of variable size

- mark beginning and end of chunks with its size, user data in between
- enables quick jump to the next chunks
- neighbouring free chunks can be easily joined
- Caution: byte alignment must be respected (pointers to words must be multiple of 4 or 8, depending on the CPU)

Time/space considerations

How to find a free chunks (of a given minimal size *n*)?

Walk all chunks. . .

Walk the free chunks (requires storing additional pointers inside the free chunks, increases speed but also overhead). . .

Which free chunks to take?

The first chunks found with size $\geq$ *n* (first-fit).

Similar, but continue searching where the last search stopped (next-fit).

The minimal free chunks with size $\geq$ *n* (best-fit).

Multiple bins

Compromise used in malloc:

> Multiple bins for free blocks of fixed sizes, e.g. `16`, `24`, `32`, ...
>
> When requesting *n* bytes, look at the smallest non-empty appropriate bin and take free block from there.

Next-fit: aims to preserve locality but is observed to lead to bad fragmentation; used only in specific cases.

Description of more implementation details:
`http://g.oswego.edu/dl/html/malloc.html`

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.

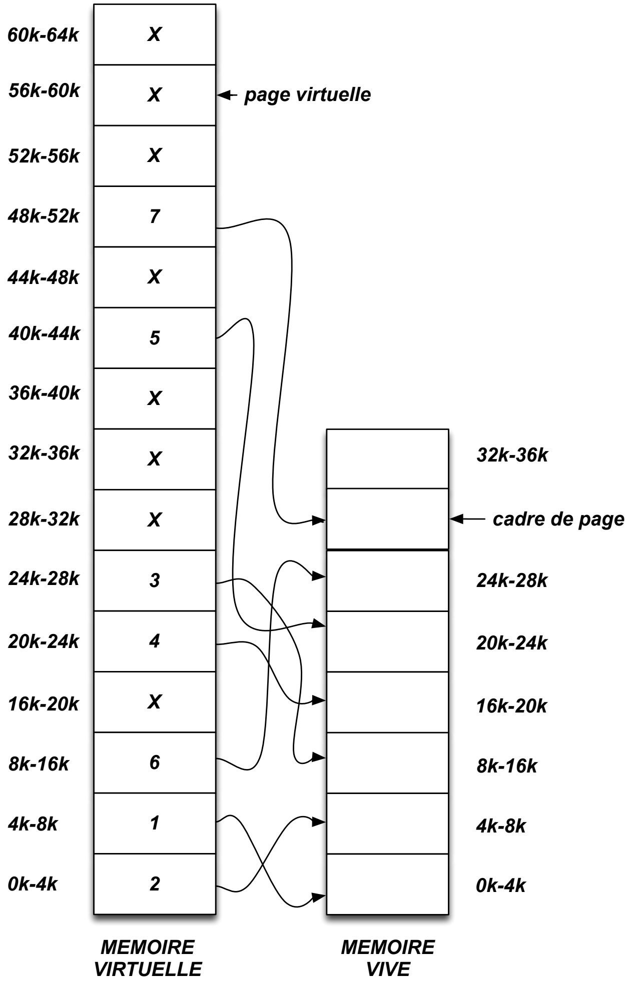
The MMU implements a paging mechanism:

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:



60k-64k X

56k-60k X ← page virtuelle

52k-56k X

48k-52k 7

44k-48k X

40k-44k 5

36k-40k X

32k-36k X

28k-32k X

24k-28k 3

20k-24k 4

16k-20k X

8k-16k 6

4k-8k 1

0k-4k 2

MEMOIRE VIRTUELLE

32k-36k

← cadre de page

24k-28k

20k-24k

16k-20k

8k-16k

4k-8k

0k-4k

MEMOIRE VIVE

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.