Virtual memory

François Trahay

Contents

Introduction

- A process needs to be present in main memory to run
- Central memory divided into two parts
  - The space reserved for the operating system
  - The space allocated to processes
- Memory management concerns the process space
- Memory capacities are increasing, but so are the requirements $\rightarrow$ Need for multiple memory levels

    - Fast memory (cache)
    - Central memory (RAM)
    - Auxiliary memory (disk)
- Principle of inclusion to limit updates between different levels

Regarding the principle of inclusion, in an Intel architecture, the L1 cache (*Level* 1) is included in L2 cache (*Level* 2), which is itself included in RAM, which is included in the *swap* (disk).

Here are the typical access times to data located in the different types of memory on a "classic" machine (Intel Core i5 *Skylake* processor) in 2017 [1]:

- data in the L1 cache: 4 cycles or 1 ns
- data in the L2 cache: 12 cycles or 3 ns (3 times slower than L1)
- data in the L3 cache: 44 cycles or 10ns (10 times slower than L1)
- data in RAM: 60 - 100 ns (100 times slower than L1)
- data on a NVMe disk: 20 µs (20,000 times slower than L1)
- data on an SSD disk: 150 µs (150,000 times slower than L1)
- data on a hard disk: 10 ms (10 million times slower than L1)

The following table shows the cost difference between the types of memory (and the evolution over the years):

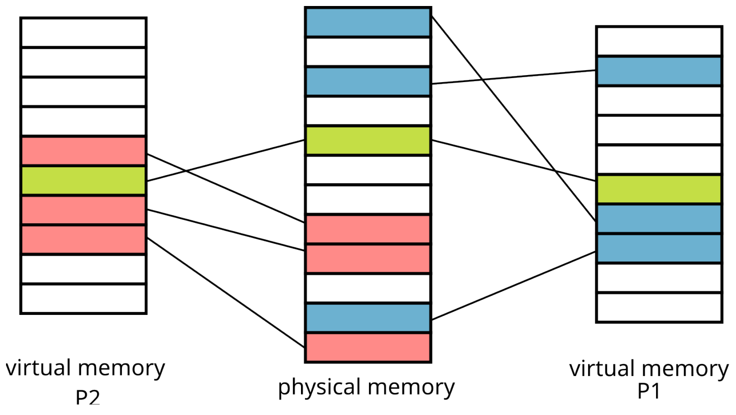
Year	2008	2009	2010	2014	2019	2023
Hard disk drive, 7200 tr/mn (in €/GiB)	0,50	0,32	0,10	0,04	0.027	0.023
SSD disk (in €/GiB)	–	–	–	0,50	0.17	0.09
USB key (in €/GiB)	–	–	1,64	0.62	0.27	0.06
NVMe (in €/GiB)	–	–	–	–	0.21	0.10
RAM (in €/GiB)	–	37,00	21,85	8.75	7.23	2.75

Paging

---

[1] Data available in the manual "*Intel 64 and IA-32 Architectures Optimization Reference Manual* "

Overview



- The address space of each program is split into **pages**
- Physical memory divided into **page frames**
- Matching between some **pages** and **page frames**

Status of memory pages

- The memory pages of a process can be
  - In main memory / in RAM (active pages)
  - Non-existent in memory (inactive pages never written)
  - In secondary memory / in the Swap (inactive pages that have already been written)

  $\rightarrow$ each process has a contiguous memory space to store its data

- The paging mecanism
  - Translates virtual addresses to/from physical addresses
  - Loads the necessary pages (in case of page faults)
  - (Optionally) move active pages to secondary memory

In Linux, page frames are 4KB in size (defined size by the constants `PAGE_SIZE` and `PAGE_SHIFT` in the file `page.h`).

Logical (or virtual) address

- Address space is divided using the most significant bits
  - Logical address on `k` bits:
    - Page number: `p` bits
    - Offset in the page: `d = (k - p)` bits

  $\rightarrow$ 2^p pages and each page contains 2^{k-p} bytes

- Page size
  - Usually 4 KiB (k-p = 12 bits, so p = 52 bits)
  - *Huge pages*: 2 MiB, 1 GiB, 512 GiB, or 256 TiB pages
- Choice = compromise between various opposing criteria
  - Last page is half wasted
  - Small capacity memory : small pages
  - Scalability of the page management system

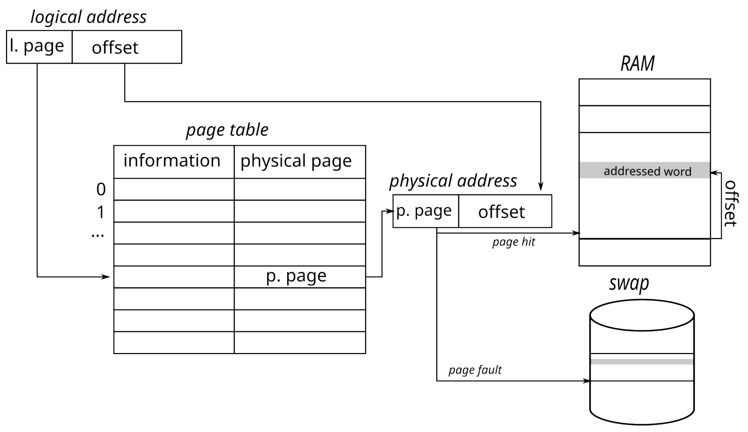
On 64-bit Intel (`x86_64`) or ARM 64 bits architectures (ARMv8), the addresses are stored on 64 bits (i.e. `size (void *)` is 8 bytes), but only 48 bits are usable for virtual addresses.

On `RISC-V` architectures, a system can choose from four virtual address sizes, including 48 bits as other architectures, and also 57 bits.

Huge pages Some applications use large amounts of data (sometimes several GiB) that must be placed in a large number of 4 KiB memory pages. In order to limit the number of pages to handle, some architectures (especially `x86_64` and `RISC-V`) allow the use of larger memory pages (typically 2 MiB and 1 GiB, but also bigger) which are called *huge pages* in the Linux kernel (sometimes *megapages* and *gigapages*).

Page table

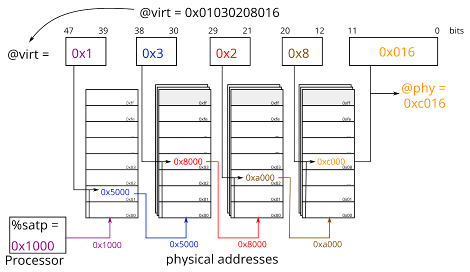
- The correspondence between logical address and address physical is done with a page table that contains
  - Page frame number
  - Information bits (presence, permissions, upload timestamp ...)



Implementation of a page table

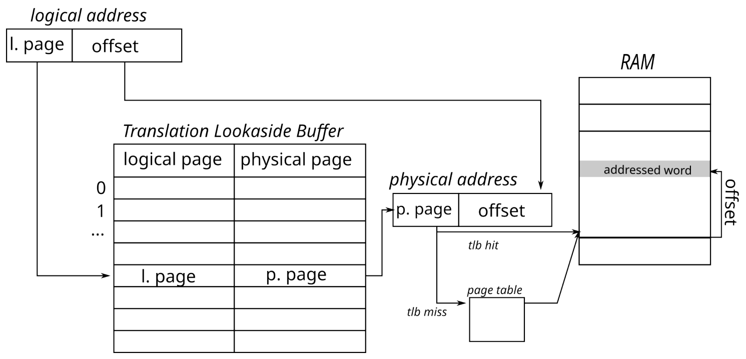
- On `x86_64` or `RISC-V`, a page table = 4-levels tree
  - The physical address of a 512-entry root table is stored in the `satp` register (`cr3` on `x86` architectures)
  - Each entry in a table gives the address of the following table
  - virtual address decomposed into 4 indexes (`n[0..3]`) + 1 *offset*, then translated using:

```
uint64_t cur = %satp3;              // cur = root table physical address
for(int i=0; i<3; i++)
  cur = ((uint64_t*)cur)[n[i]]; // physical memory access, next entry
return cur + offset;              // add the offset
```



Translation Lookaside Buffer (TLB)

- Problem: any access to information requires several memory accesses
- Solution: use associative memories (fast access registers)
- Principle
  - A number of registers are available
  - Logical page number N_p compared to the content of each register
  - if found → gives the corresponding frame number N_c
  - Otherwise use the page table

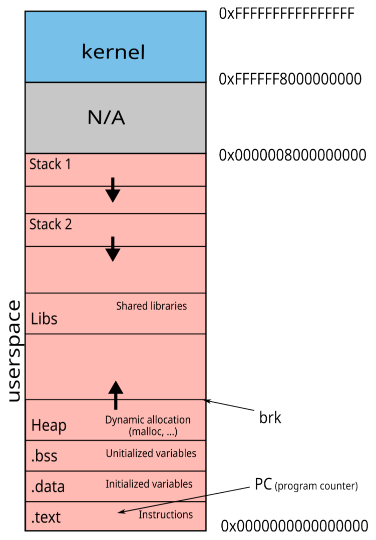


Intel architectures have *Translation Look-aside Buffers* (TLB) with 32, 64, or even 256 entries. TLB are sometimes called *address translation cache*.

User point of view

Memory space of a process

- Composed of:
  - kernel space
  - the different sections of the executed ELF file (`.text`, `.data`, etc.)
  - the heap
  - the stack (one per thread)
  - shared libraries



Memory mapping

- How to populate the memory space of a process?
  - For each ELF file to be loaded:
    * open the file with `open`
    * each ELF section is *mapped* in memory (with `mmap`) with the appropriate permissions
    * Results are visible in `/proc/<pid>/maps`

```
$ cat /proc/self/maps
5572f3023000-5572f3025000 r--p 00000000 08:01 21495815   /bin/cat
```

```
5572f3025000-5572f302a000 r-xp 00002000 08:01 21495815   /bin/cat
5572f302e000-5572f302f000 rw-p 0000a000 08:01 21495815   /bin/cat
5572f4266000-5572f4287000 rw-p 00000000 00:00 0          [heap]
7f33305b4000-7f3330899000 r--p 00000000 08:01 22283564   /usr/lib/locale/locale-archive
7f3330899000-7f33308bb000 r--p 00000000 08:01 29885233   /lib/x86_64-linux-gnu/libc-2.28.so
7f33308bb000-7f3330a03000 r-xp 00022000 08:01 29885233   /lib/x86_64-linux-gnu/libc-2.28.so
[...]
7f3330ab9000-7f3330aba000 rw-p 00000000 00:00 0
7ffe4190f000-7ffe41930000 rw-p 00000000 00:00 0          [stack]
7ffe419ca000-7ffe419cd000 r--p 00000000 00:00 0          [vvar]
7ffe419cd000-7ffe419cf000 r-xp 00000000 00:00 0          [vdso]
```

Memory allocation

- `void* malloc(size_t size)`
  - Returns a pointer to an buffer of `size bytes`
- `void* realloc(void* ptr, size_t size)`
  - Changes the size of a buffer previously allocated by `malloc`
- `void* calloc(size_t nmemb, size_t size)`
  - Same as `malloc`, but memory is initialized to 0
- `void *aligned_alloc( size_t alignment, size_t size )`
  - Same as `malloc`. The returned address is a multiple of `alignment`
- `void free(void* ptr)`
  - Free an allocated buffer

- All these functions are implemented in the standard C library (which in some cases make system calls).

- The `malloc(3)` algorithm is very efficient. It is not therefore generally not necessary to try to optimize it.

However:

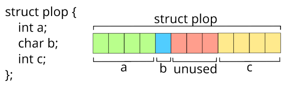
- When allocating a memory area that must be initialized at 0, we shall privilege `calloc (3)` (it is more efficient than a `malloc(3)` followed by `memset(3)`).
- If necessary, `mallopt` allows to fine tune the behavior of `malloc(3)`
- Additionally, it is possible to customize the behavior of standard allocation / release routines by setting `__malloc_hook`, `__realloc_hook` and `__free_hook`. Be careful, these mechanisms can lead to reentrancy problems.
- When freeing a buffer with `free`, it is strongly advised to set the pointer to `NULL`. This allows the program to crash immediatly if, by mistake, we access this (now inexistant) buffer again using this pointer.

The following program illustrates how setting a pointer to `NULL` allows to crash immediatly and how using a debugger allows to quickly find the origin of the error.

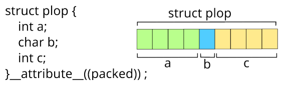
```
/**********************/
/* resetToNULL.c */
/**********************/

/* This program illustrates the value of assigning a variable to NULL
   which contains a pointer to an unallocated buffer.
   Indeed, it makes a segmentation fault, which makes it possible to
   identify that we have executed an illegal operation. Using a
   debugger allows to understand the problem.
*/
/*                                                                  */
/* 1) Compile the program with the option -g                        */
/*    cc -g -o resetToNULL resetToNULL.c                            */
/* 2) ./resetToNULL                                                 */
/*    ==> Segmentation fault                                        */
/* 3) ulimit -c unlimited                                           */
/* 4) ./resetToNULL                                                 */
/*    ==> Segmentation fault (core dumped)                          */
/* 5) ddd ./resetToNULL core                                        */

#include <stdlib.h>
#include <assert.h>

void h(char *p){
  *p = 'a';
}

void g(char *p){
  h(p);
}

void f(char *p){
  g(p);
}

int main(){
  char *p = NULL;

  f(p);

  p = malloc(1);
  assert(p != NULL);

  f(p);
  free(p);
  p = NULL;

  f(p);

  return EXIT_SUCCESS;
}
```

Memory alignment

- Memory alignment depends on the type of data
  - `char` (1-byte), `short` (2-bytes), `int` (4-bytes), ...
- A data structure may be larger than its content



- A data structure can be packed with `__attribute__((packed))`



Memory alignment applies to variables as well as to members of data structures.

The following program illustrates how alignement affects the size of a data structure:

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

struct plop {
  int a;
  char b;
  int c;
};

struct plop_packed {
  int a;
  char b;
  int c;
} __attribute__((packed));
```

```
int main(void) {

  struct plop p1;
  struct plop_packed p2;
  printf("struct plop -- size: %lu bytes, address: %p\n",
         sizeof(struct plop), &p1);
  printf("\t.a -- size: %lu bytes, address: %p, offset: %lu\n",
         sizeof(p1.a), &p1.a, offsetof(struct plop, a));
  printf("\t.b -- size: %lu bytes, address: %p, offset: %lu\n",
         sizeof(p1.b), &p1.b, offsetof(struct plop, b));
  printf("\t.c -- size: %lu bytes, address: %p, offset: %lu\n",
         sizeof(p1.c), &p1.c, offsetof(struct plop, c));
  printf("\n");

  printf("struct plop_packed -- size: %lu bytes, address: %p\n",
         sizeof(struct plop_packed), &p2);
  printf("\t.a -- size: %lu bytes, address: %p, offset: %lu\n",
         sizeof(p2.a), &p2.a, offsetof(struct plop_packed, a));
  printf("\t.b -- size: %lu bytes, address: %p, offset: %lu\n",
         sizeof(p2.b), &p2.b, offsetof(struct plop_packed, b));
  printf("\t.c -- size: %lu bytes, address: %p, offset: %lu\n",
         sizeof(p2.c), &p2.c, offsetof(struct plop_packed, c));
  printf("\n");
  return 0;
}
```

```
$ ./memory_alignment
struct plop -- size: 12 bytes, address: 0x7ffc05ad8184
        .a -- size: 4 bytes, address: 0x7ffc05ad8184, offset: 0
        .b -- size: 1 bytes, address: 0x7ffc05ad8188, offset: 4
        .c -- size: 4 bytes, address: 0x7ffc05ad818c, offset: 8

struct plop_packed -- size: 9 bytes, address: 0x7ffc05ad817b
        .a -- size: 4 bytes, address: 0x7ffc05ad817b, offset: 0
        .b -- size: 1 bytes, address: 0x7ffc05ad817f, offset: 4
        .c -- size: 4 bytes, address: 0x7ffc05ad8180, offset: 5
```

The libc point of view

- How to request memory from the OS
  - `void *sbrk(intptr_t increment)`
    - increase the heap size by `increment` bytes
  - `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)`
    - map a file in memory

    * if `flags` contains `MAP_ANON`, does not map any file, but allocates an area filled with 0s

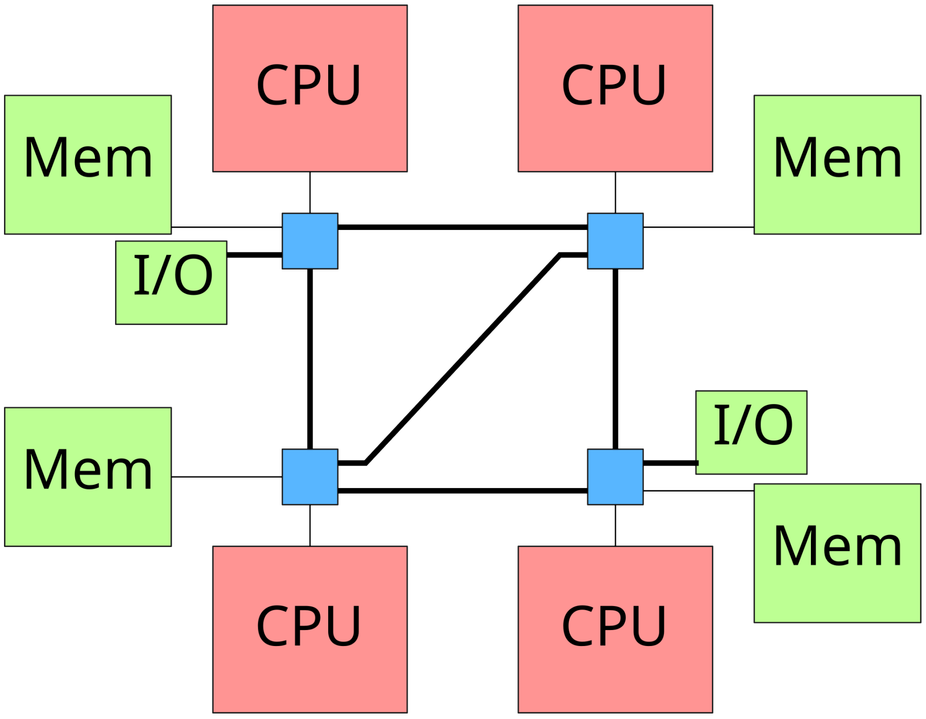
---

Memory allocation strategies

Non-Uniform Memory Access

- Several interconnected memory controllers
- Memory consistency between processors
- Privileged access to the local *memory bank*
- Possible access (with an additional cost) to distant *memory banks*

→ *Non-Uniform Memory Access* → On which memory bank to allocate data?

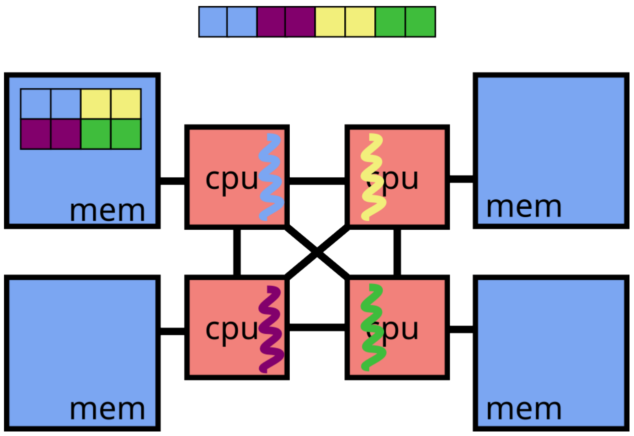


First touch **allocation strategy**

- Linux default lazy allocation strategy
- Allocation of a memory page on the local node when first accessed

- Assumption: the first thread to use a page will probably will use it in the future

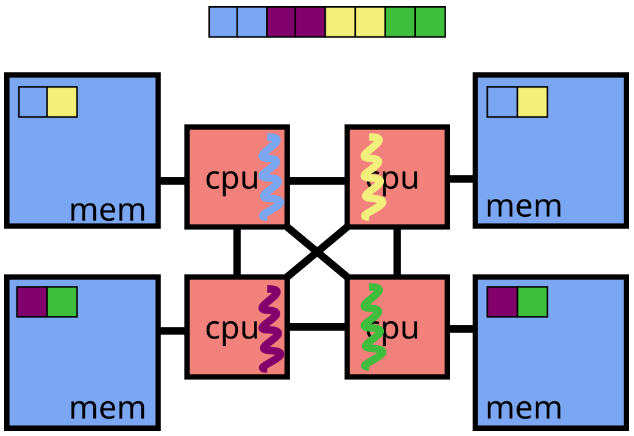
```
double *array = malloc(sizeof(double)*N);

for(int i=0; i<N; i++) {
  array[i] = something(i);
}

#pragma omp parallel for
for(int i=0; i<N; i++) {
  double value = array[i];
  /* ... */
}
```



Interleaved allocation strategy

- Pages are allocated on the different nodes in a *round-robin* fashion
- Allows load balancing between NUMA nodes
- `void *numa_alloc_interleaved(size_t size)`

```
double *array =
  numa_alloc_interleaved(sizeof(double)*N);

for(int i=0; i<N; i++) {
  array[i] = something(i);
}

#pragma omp parallel for
for(int i=0; i<N; i++) {
  double value = array[i];
  /* ... */
}
```

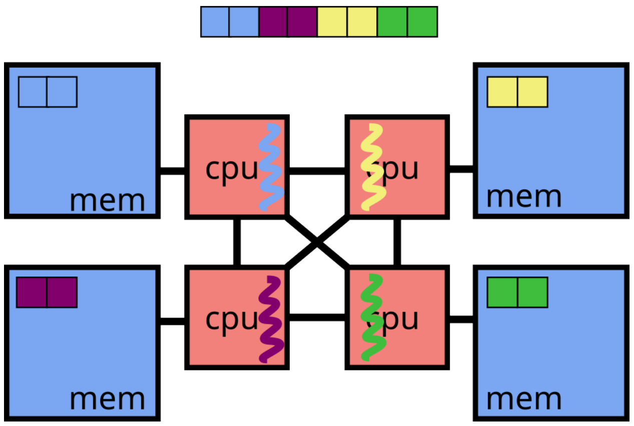


It is also possible to use `set_mempolicy` in order to choose an allocation strategy for future memory allocations.

`mbind`

- `long mbind(void *addr, unsigned long len, int mode, const unsigned long *nodemask, unsigned long maxnode, unsigned flags)`
- Place a set of memory pages on a (set of) NUMA node → allows manual placement of memory pages

```
double *array = malloc(sizeof(double)*N);
mbind(&array[0], N/4*sizeof(double),
      MPOL_BIND, &nodemask, maxnode,
      MPOL_MF_MOVE);

#pragma omp parallel for
for(int i=0; i<N; i++) {
  double value = array[i];
  /* ... */
}
```


cpu
cpu
cpu
cpu
mem
mem
mem
mem