Virtual memory

François Trahay

CSC4508 – Operating Systems

2022–2023

1 Introduction

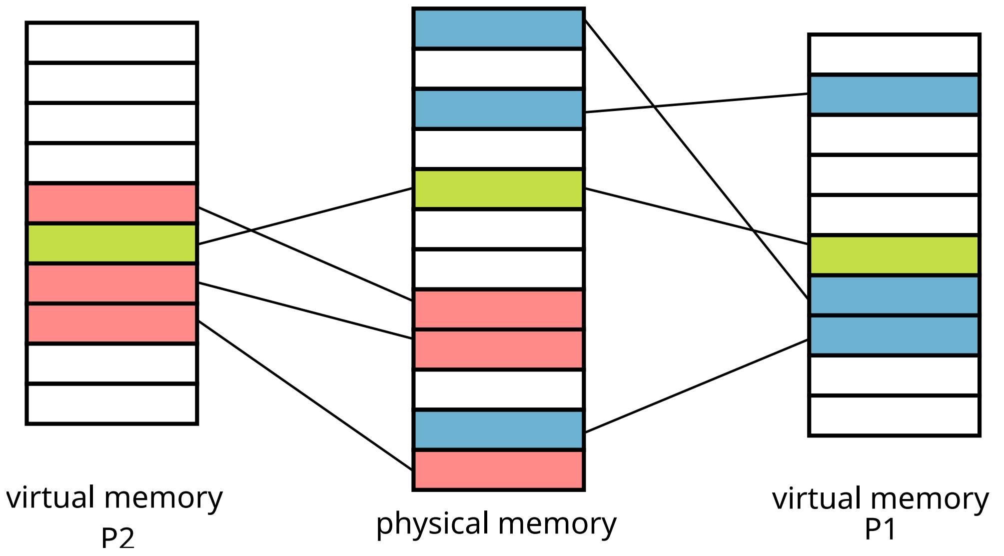
- A process needs to be present in main memory to run
- Central memory divided into two parts:
  - The space reserved for the operating system
  - The space allocated to processes
- Memory management concerns the process space
- Memory capacities are increasing, but so are the requirements

  $\Longrightarrow$ Need for multiple memory levels
  - Fast memory (cache)
  - Central memory (RAM)
  - Auxiliary memory (disk)

  Principle of inclusion to limit updates between different levels

2 Paging

2.1 Overview



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

2.2 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)

$\Longrightarrow$ 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

2.3 Logical (or virtual) address

- Address space is divided using the most significant bits

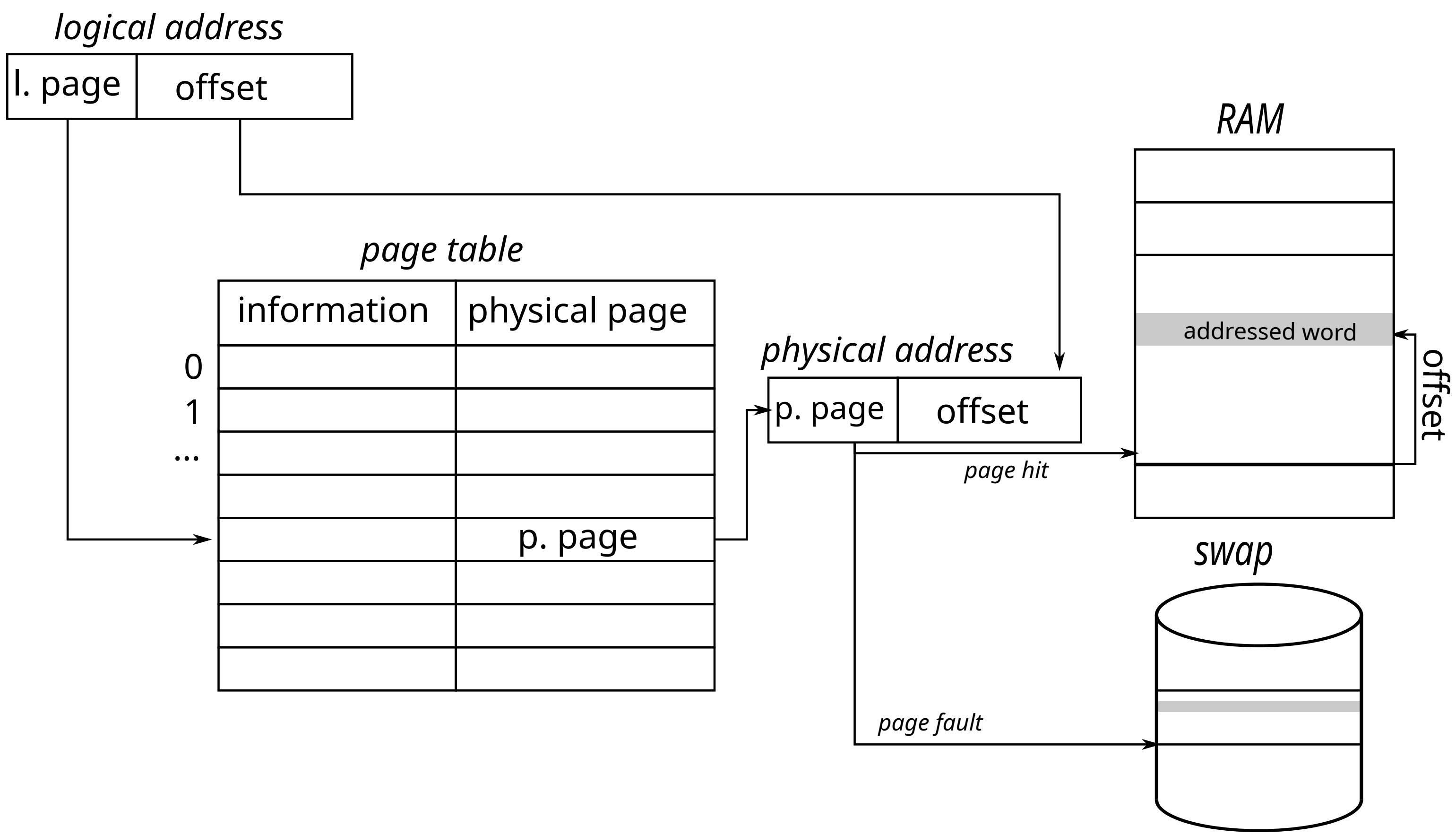
Logical address on k bits	
Page number	Offset in the page
p bits	($d = (k-p)$ bits)

$\Longrightarrow$ 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, or 1 GiB pages
- Choice = compromise between various opposing criteria
  - Last page is half wasted
  - Small capacity memory : small pages
  - Scalability of the page management system

2.4 Page table

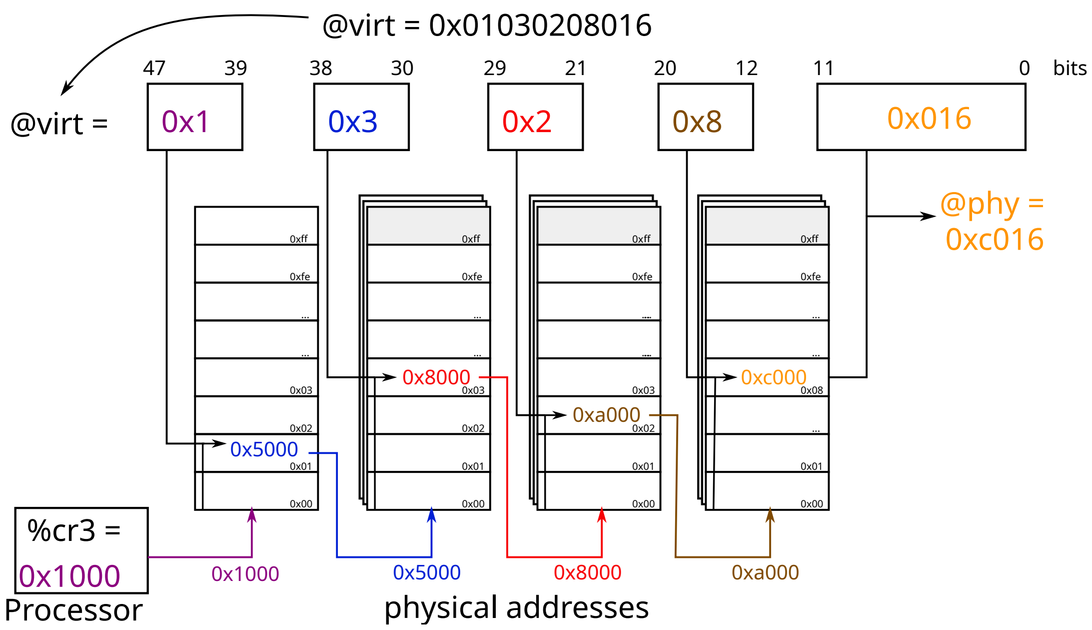
- The correspondence between logical address and address physical is done with a page table that contains
  - Page frame number
  - Information bits (presence, permissions, upload timestamp . . .)



2.5 Implementation on a 64-bit pentium

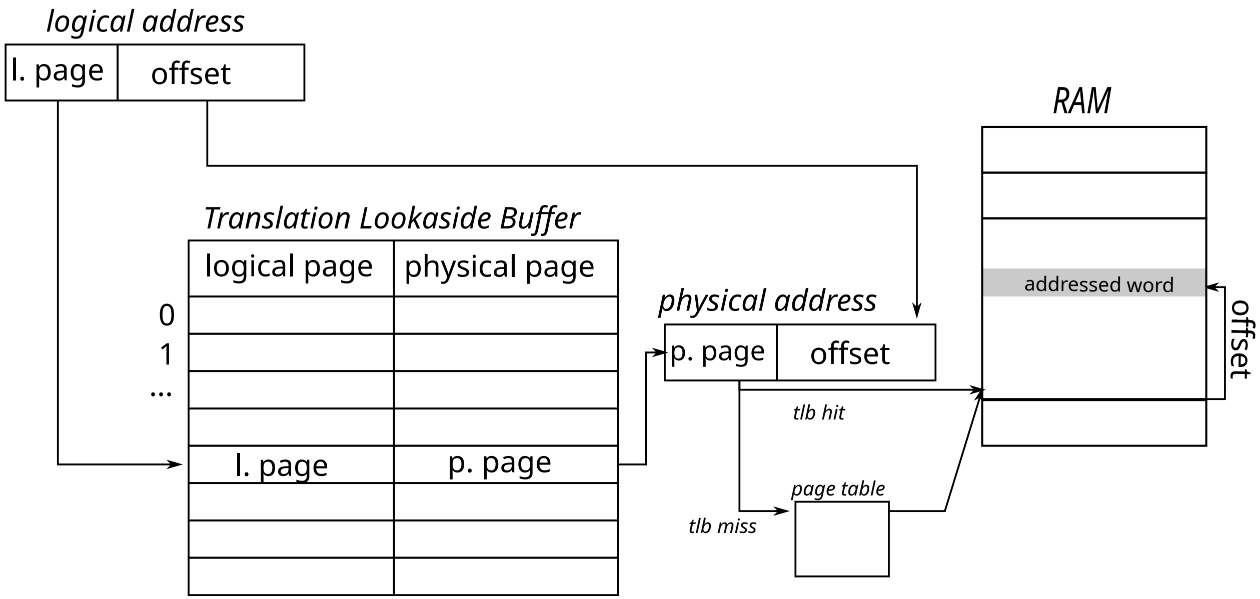
- Page table = 4-levels tree:
  - The physical address of a 512-entry root table is stored in the CR3 register
  - Each entry in a table gives the address of the following table
- @virt decomposed into 4 indexes (n[0..3]) + 1 *offset*, then translated using:

```
uint64_t cur = %cr3;              // 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
```



2.6 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 $\rightarrow$ gives the corresponding frame number N_c
  - Otherwise use the page table

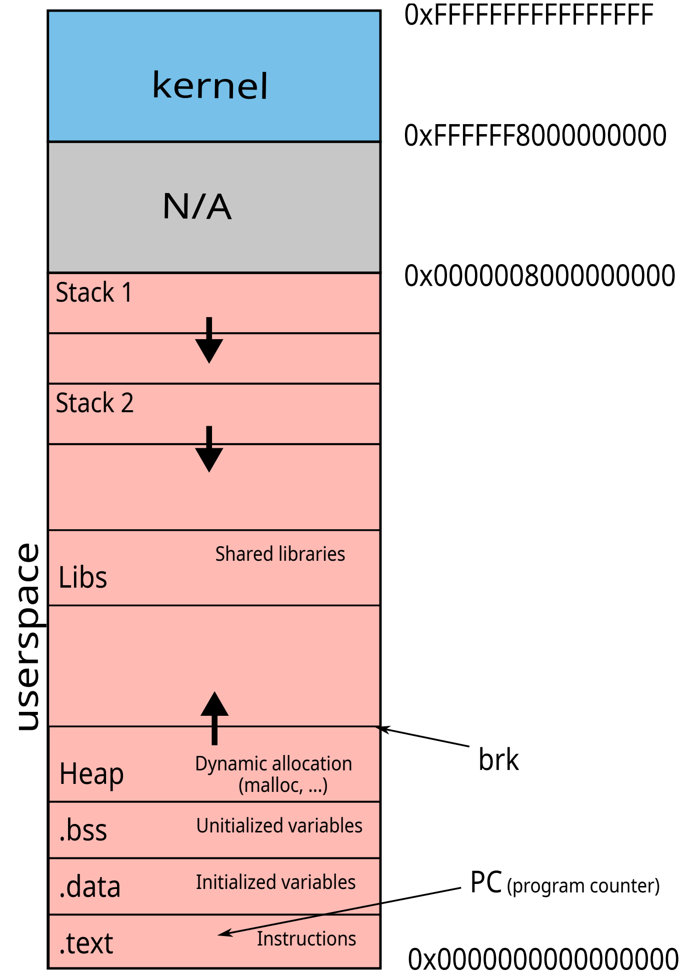


3 User point of view

3.1 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



3.2 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]
```

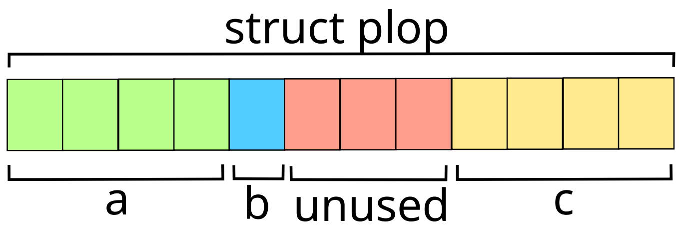
3.3 Memory allocation

- `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

3.4 Memory alignment

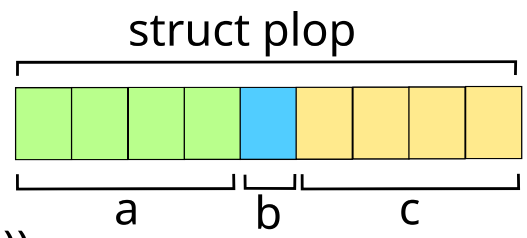
- Memory alignment depends on the type of data
  - `char` (1-byte), `short` (2-bytes), `int` (4-bytes), …
- A data structure may be larger than its content

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



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

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



3.5 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

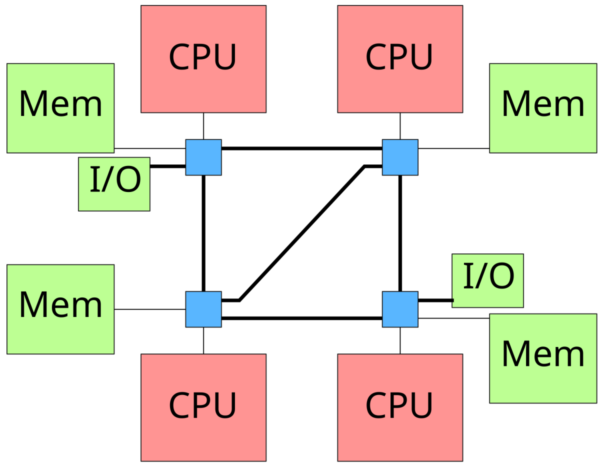
4 Memory allocation strategies

4.1 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?



4.2 *First touch* allocation strategy

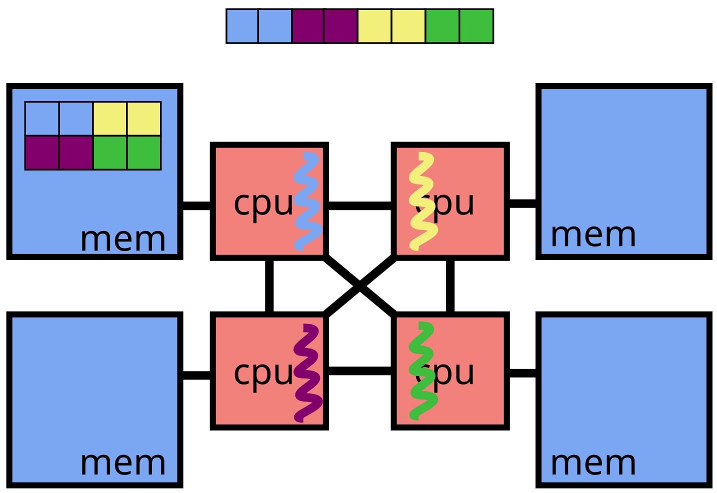
- Linux default lazy allocation strategy
- Allocation of a memory page on the local node when first accessed
- Assumption: the first thread to use a page will probably will use it in the future

first_touch.c

```
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];
  /* ... */
}
```



4.3 *Interleaved* allocation strategy

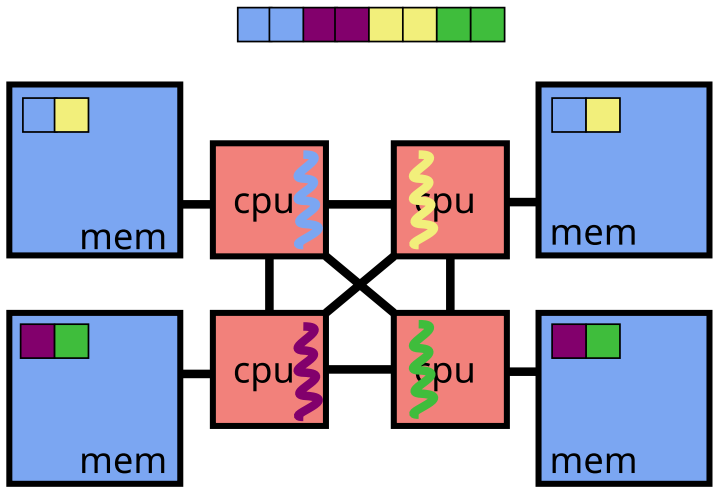
- Pages are allocated on the different nodes in a *round-robin* fashion
- Allows load balancing between NUMA nodes
- `void *numa_alloc_interleaved(size_t size)`

interleaved.c

```
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];
  /* ... */
}
```



4.4 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

manual.c

```
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];
  /* ... */
}
```