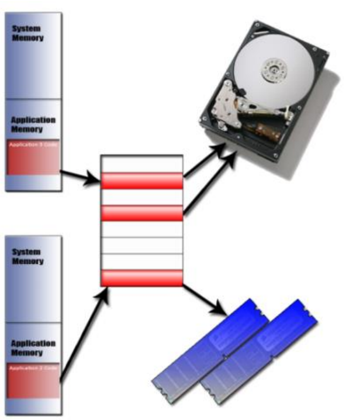


Sistemas Operativos
Virtual Memory

Patrício Domingues

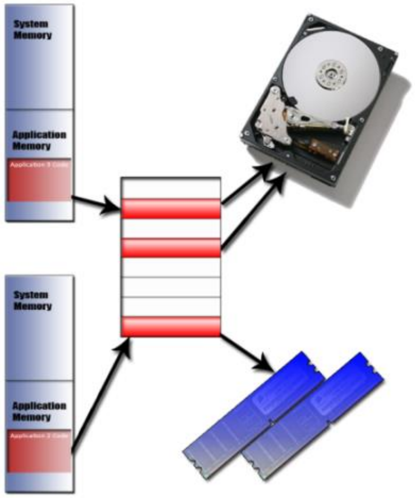
ESTG/IPLeiria

May 2019

NOTE: some slides of this chapter are based on “Virtual Memory”, CS105, Geoff Kuenning, Harvey Mudd College, 2015.

Virtual memory

- ✓ Virtual memory is *imaginary* memory
  - – it gives you the illusion of a memory arrangement that is not physically there
  - – It is mapped to physical memory by the OS
- ✓ Virtual memory
  - – Each process thinks that it has all the (virtual) memory for itself

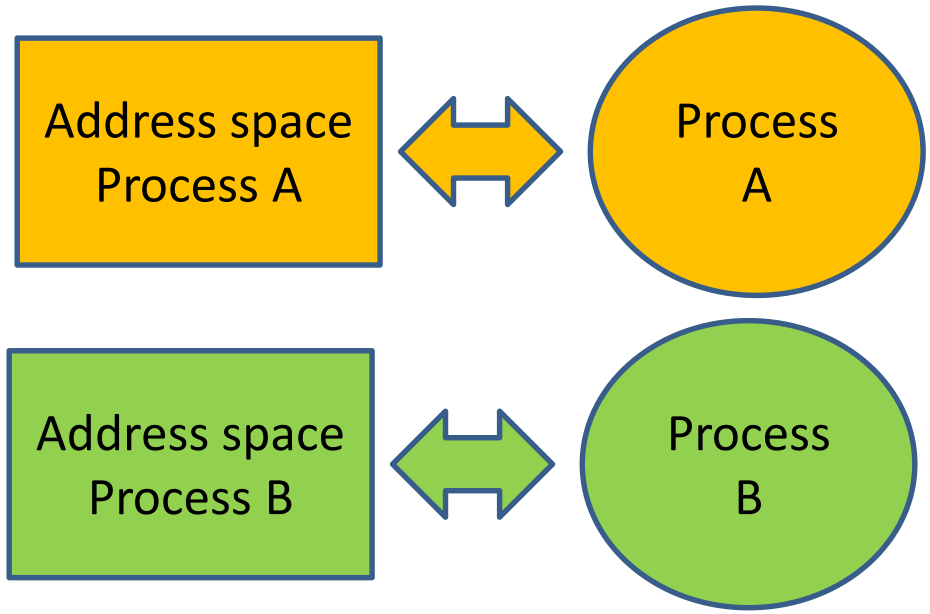


Motivations for *virtual memory* (1)

- ✓ Use physical RAM as cache for the disk
  - – Address space of a process can exceed physical memory size
    - Example: a 4 GiB virtual address space on a 2 GiB RAM machine
  - – Sum of address spaces of multiple processes can exceed physical memory
- ✓ Simplify memory management
  - – Multiple processes resident in main memory
  - – Each process has its own address space
  - – Only “active” code and data is actually in memory
  - – Allocate more memory to process as needed

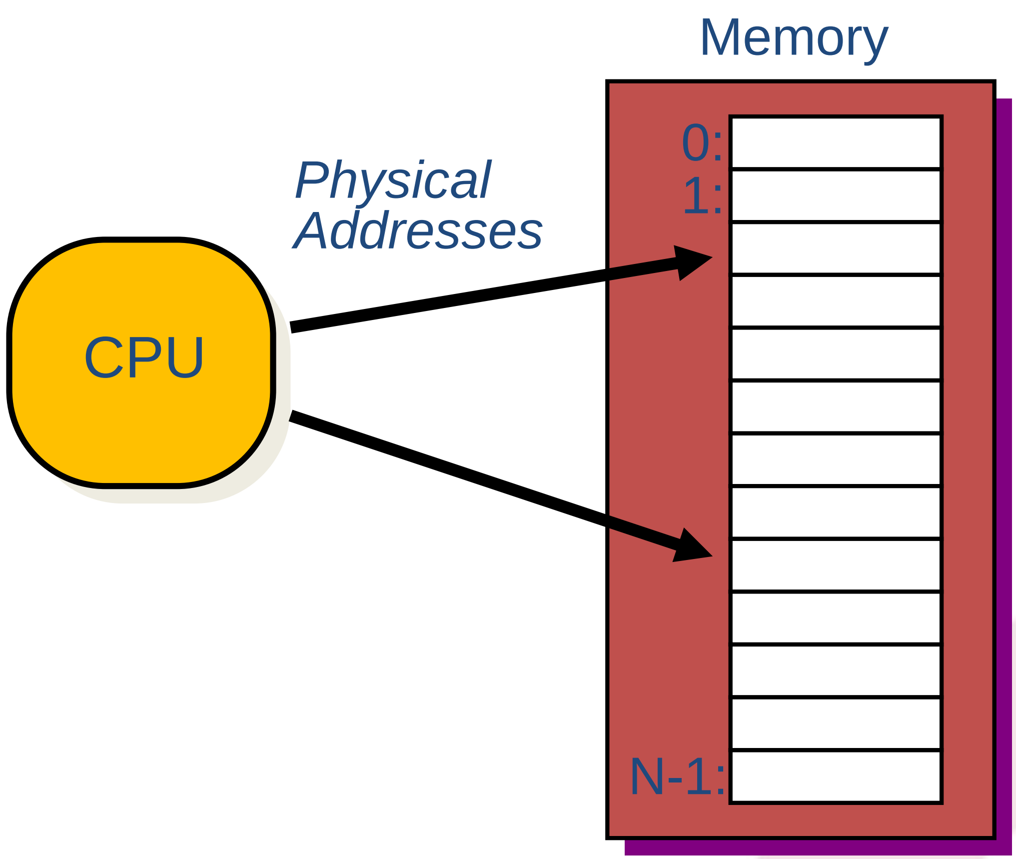
Motivations for *virtual memory* (2)

- ✓ Provide protection
  - One process cannot interfere with another
    - Because they operate in different address spaces
    - Address space isolation
  - User process cannot access privileged information
  - Different sections of address spaces have different permissions



A System with Physical Memory Only

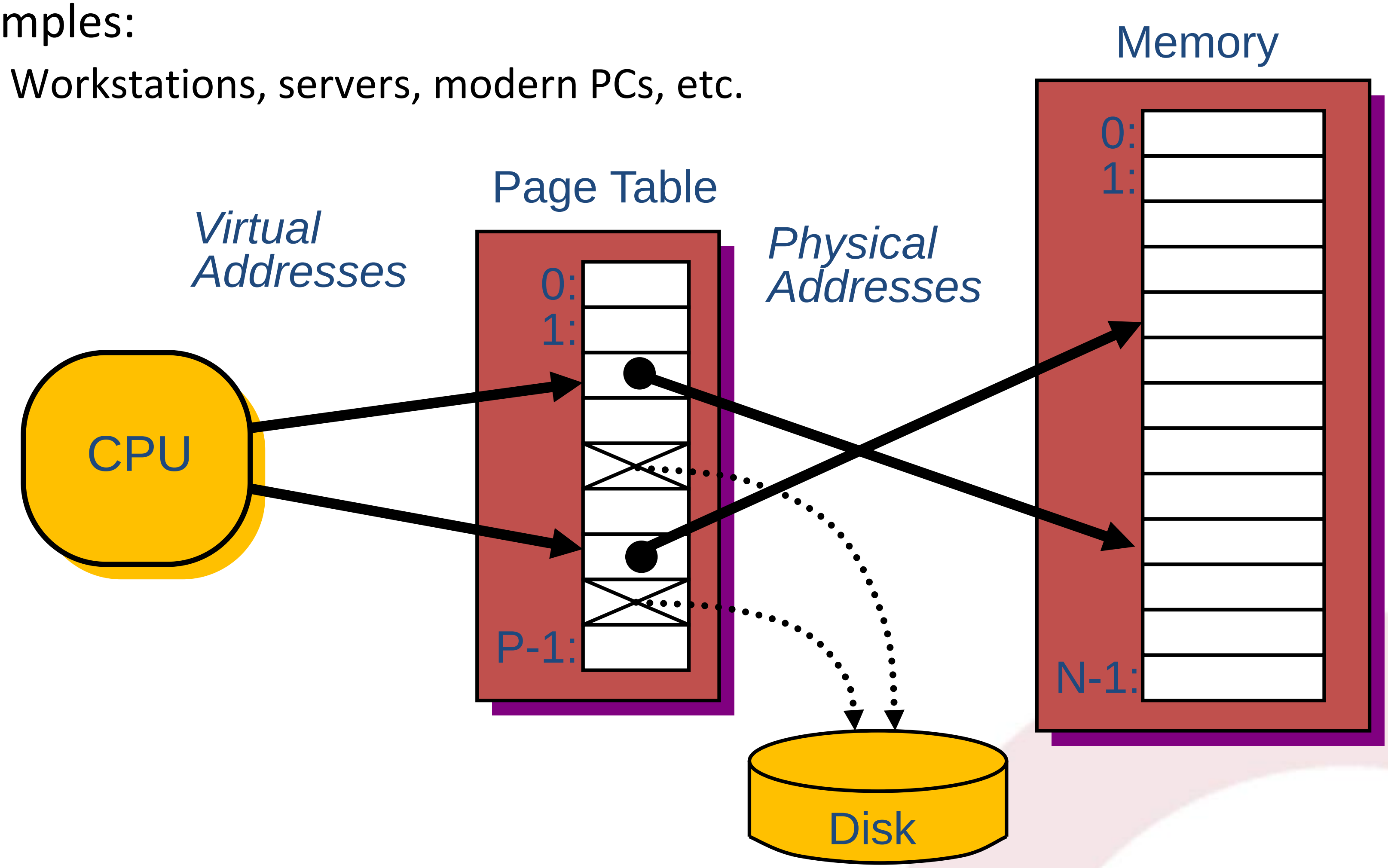
- Examples:
  - Most Cray machines, early PCs, nearly all embedded systems, etc.



- Addresses generated by the CPU correspond directly to bytes in physical memory

A System with Virtual Memory

- Examples:
  - Workstations, servers, modern PCs, etc.



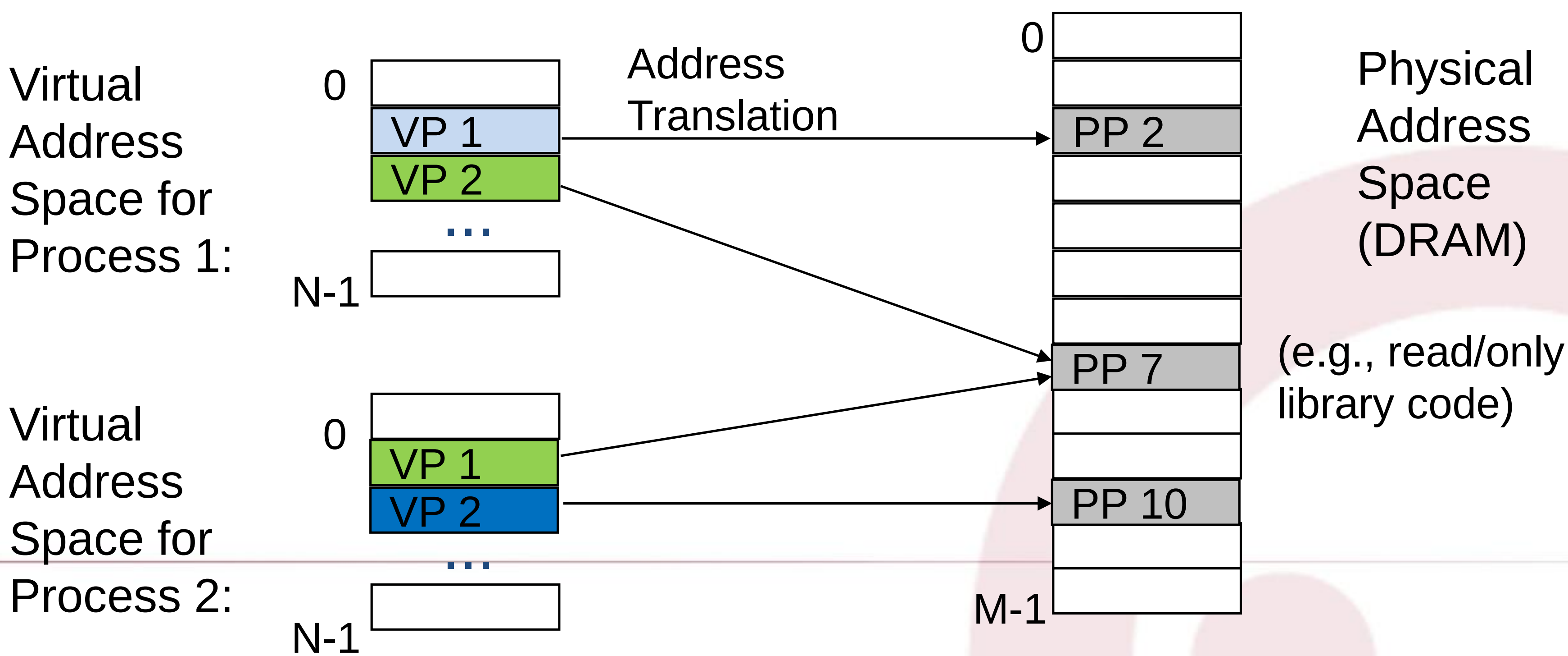
- Address Translation: Hardware converts virtual addresses to physical ones via OS-managed lookup table (page table)

Virtual memory

- ✓ Each process is afforded by the OS a single linear address space
  - as if it alone were in control of all of the memory in the system
- ✓ Virtual memory + paging
  - kernel allows many processes to coexist on the system
  - each process operates in a different address space
  - kernel manages this virtualization through hardware support (CPU, MMU, etc.)

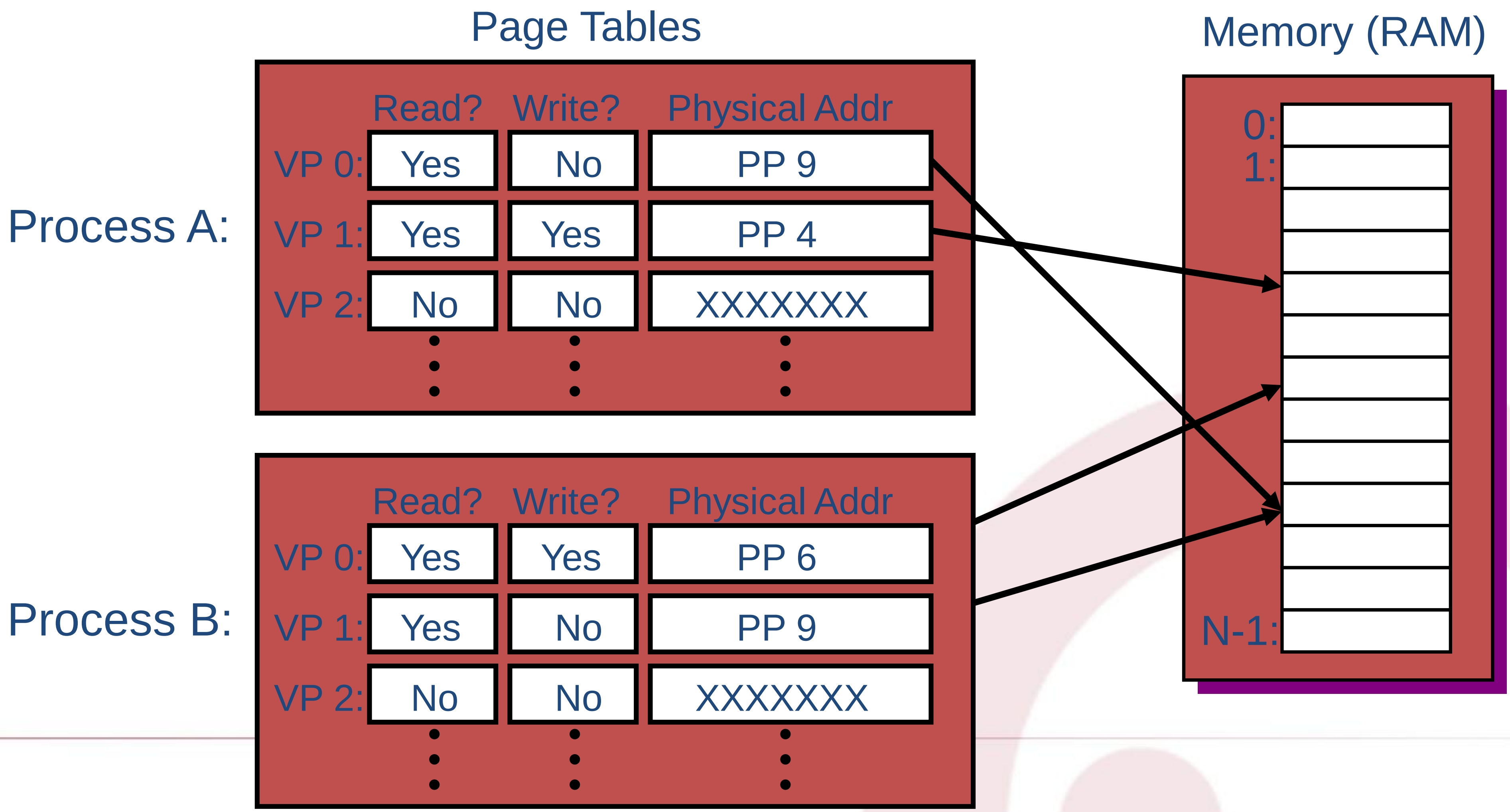
Separate Virtual Address Spaces

– Virtual and physical address spaces divided into equal-sized blocks
  - Blocks are called “pages” (both virtual and physical)
– Each process has its own virtual address space
  - Operating system controls how virtual pages are assigned to physical memory



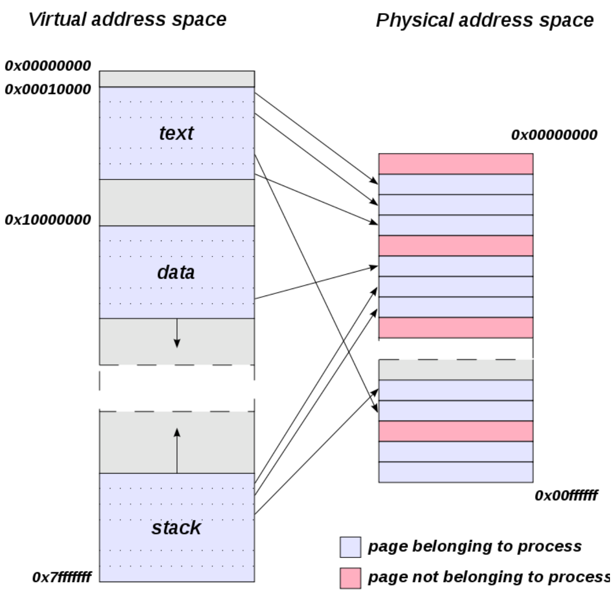
Protection

- Page table entry contains access-rights information
  - Hardware enforces this protection
    - OS is alerted if violation occurs



Logical vs physical

- There are two types of addresses
  - Logical addresses
    - Used by processes
    - Used by the CPU
  - Physical addresses
    - Used by the physical memory

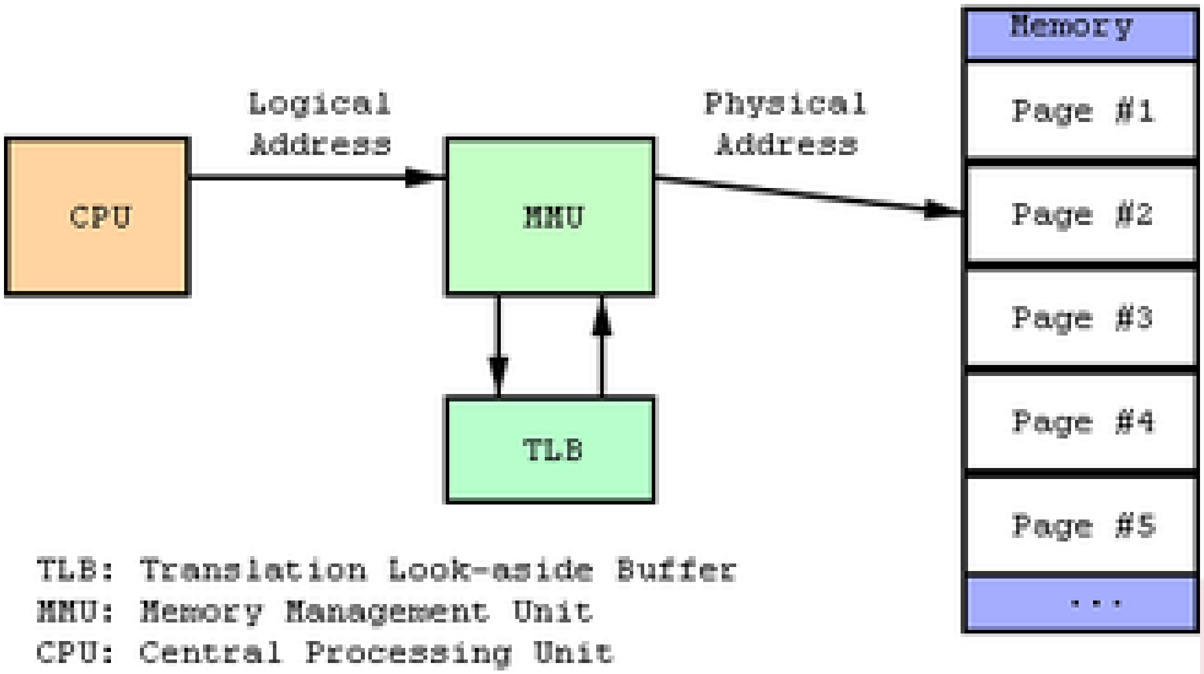


- Address translation
  - Logical addresses (CPU) are converted to physical addresses
  - The translation is done by the Memory Management Unit (MMU)

MMU: Memory Management Unit >>

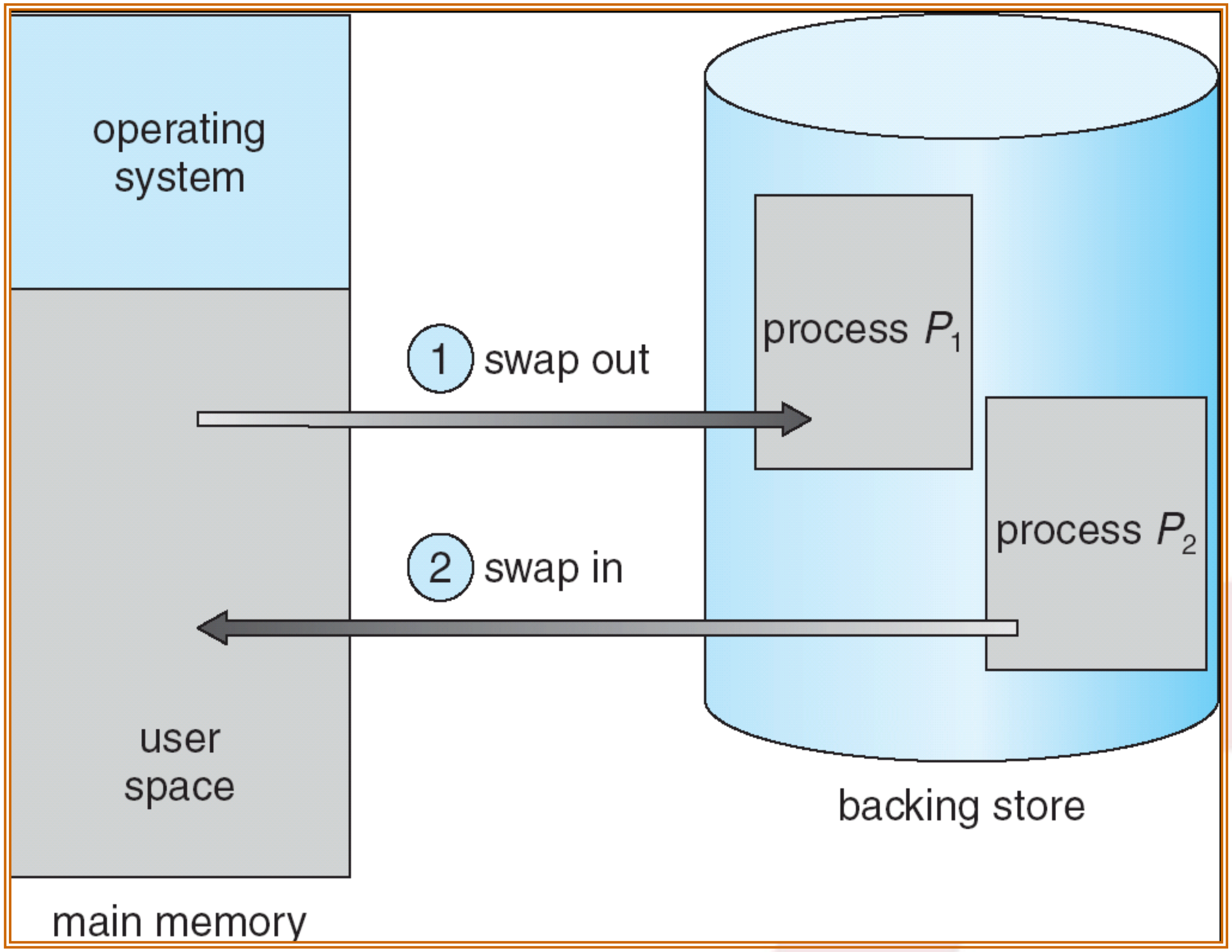
MMU

✓ MMU: Memory Management Unit



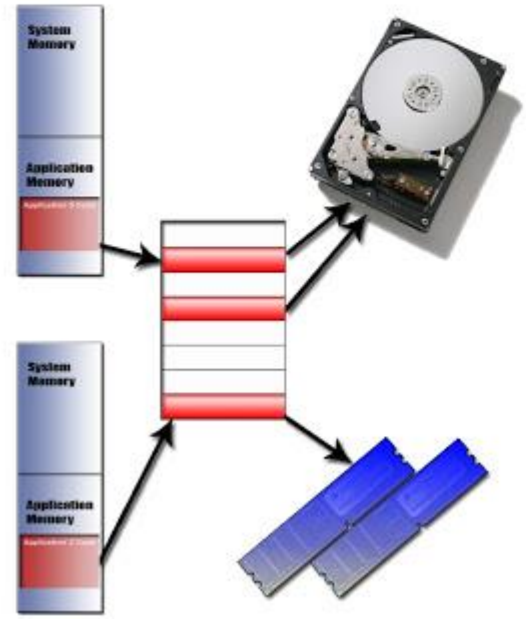
http://en.wikipedia.org/wiki/Memory_management_unit

Swapping (1)



operating system
1 swap out
2 swap in
process P_1
process P_2
user space
backing store
main memory

Swapping (2)

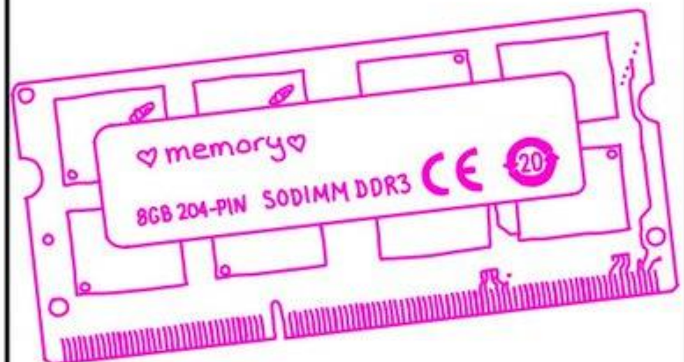


- ✓ *Swap out*
  - – Pages of a process are transfered to the secondary memory (disk)
    - ▪ It goes to the swap file
- ✓ *Swap in*
  - – Opposite operation of swap out
    - ▪ Pages of the process are transfered from the secondary memory to disk
- ✓ Since they involve the disk, swapping operation are costly

virtual memory

JULIA EVANS
@b0rk

your computer has physical memory



physical memory has addresses

0 - 8GB

but when your program references an address like 0x5c69a2a2

that's not a physical memory address! It's a virtual address

every program has its own virtual address space

program 1: 0x129520 → "puppies"

program 2: 0x129520 → "bananas"

Linux keeps a mapping from virtual memory pages to physical memory pages called the "page table"

a "page" is a 4kb chunk of memory — or sometimes bigger

PID	virtual addr	physical addr
1971	0x20000	0x192000
2310	0x20000	0x228000
2310	0x21000	0x9788000

when your program accesses a virtual address

CPU: I'm accessing 0x21000

MMU: I'll look that up in the page table and then access the right physical address

MMU "memory management unit" ← hardware

every time you switch which process is running, Linux needs to switch the page table

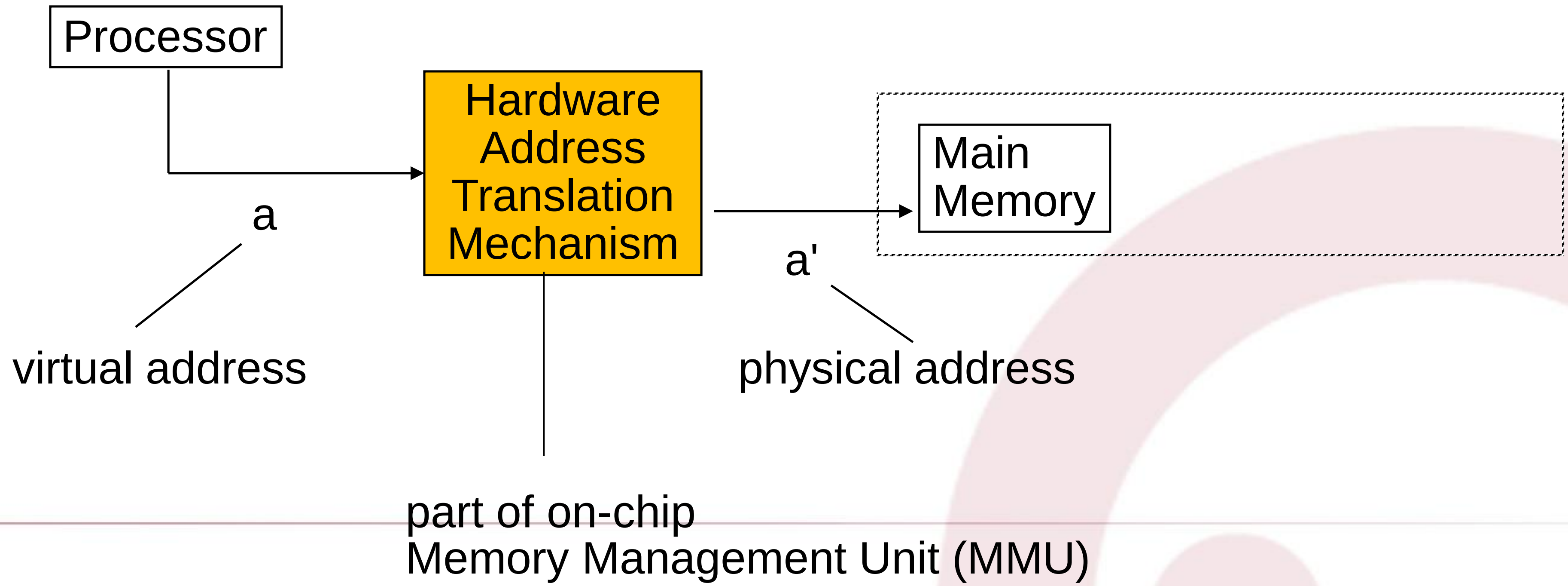
Linux: here's the address of process 2950's page table

MMU: thanks I'll use that now!

Translation of a Virtual Address (1)

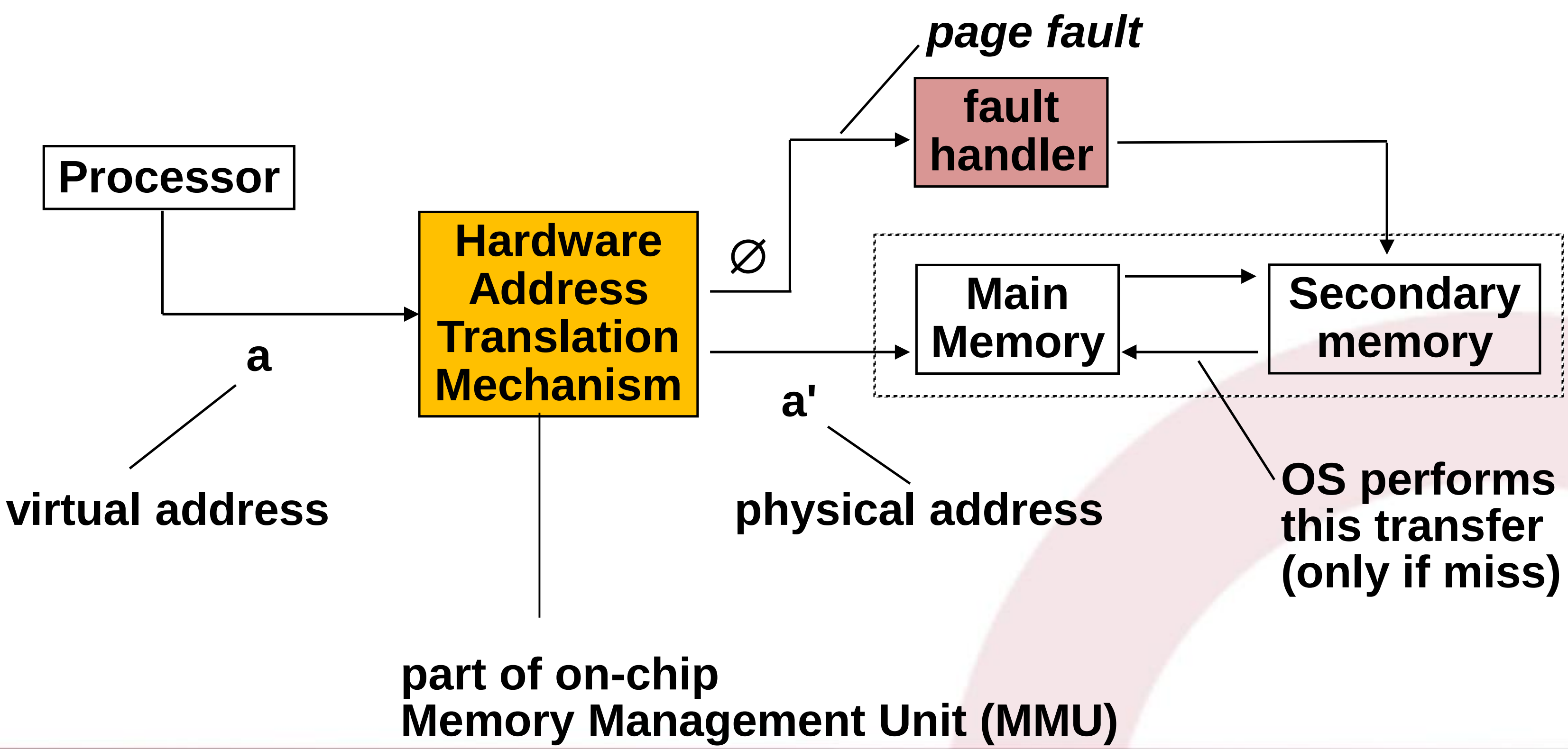
✓ Virtual address needs to be translated to physical address
  - Process emits virtual address
    - Data load/store
    - Instructions fetch

✓ Hit on translation: content is in RAM



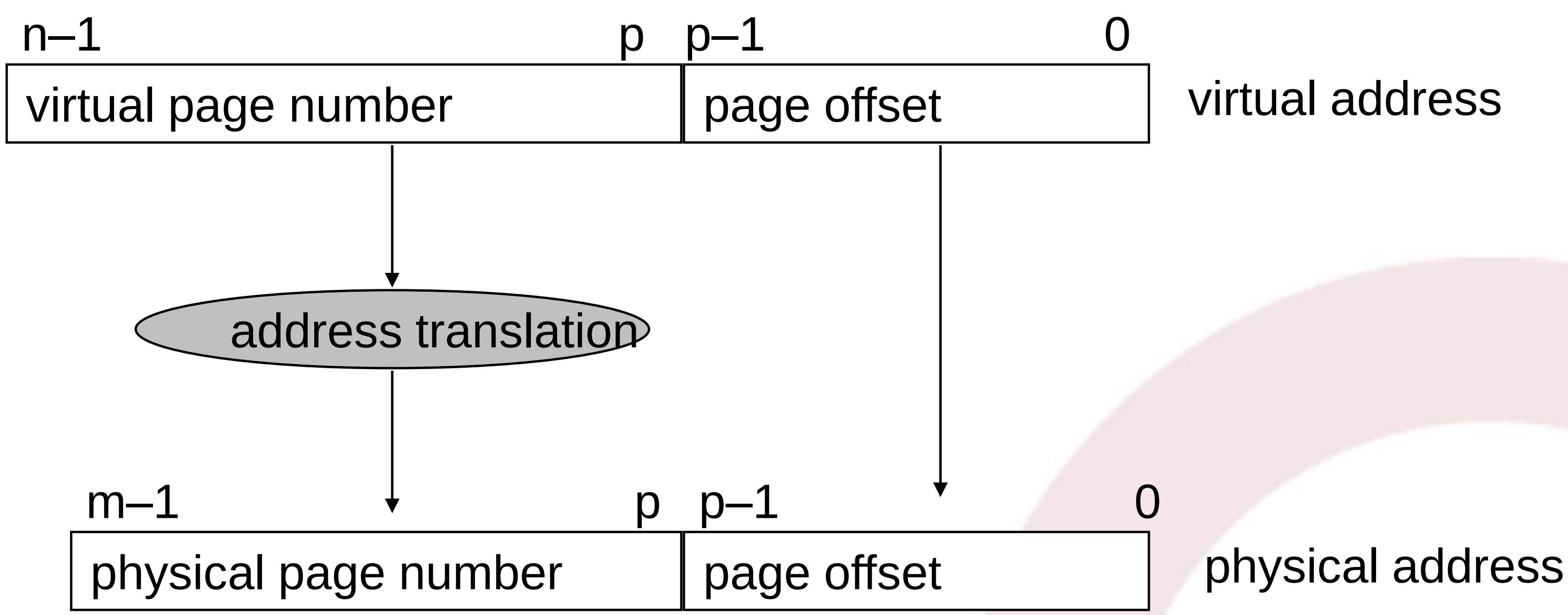
Translation of a Virtual Address (2)

- Miss on translation
  - There is a “page fault”
    - content is NOT in RAM
    - Content is in secondary memory (disk)
      - It needs to be copied back to RAM by the OS



Address Translation

- Parameters
  - $P = 2^p$ = page size (bytes)
  - $N = 2^n$ = Virtual-address limit
  - $M = 2^m$ = Physical-address limit

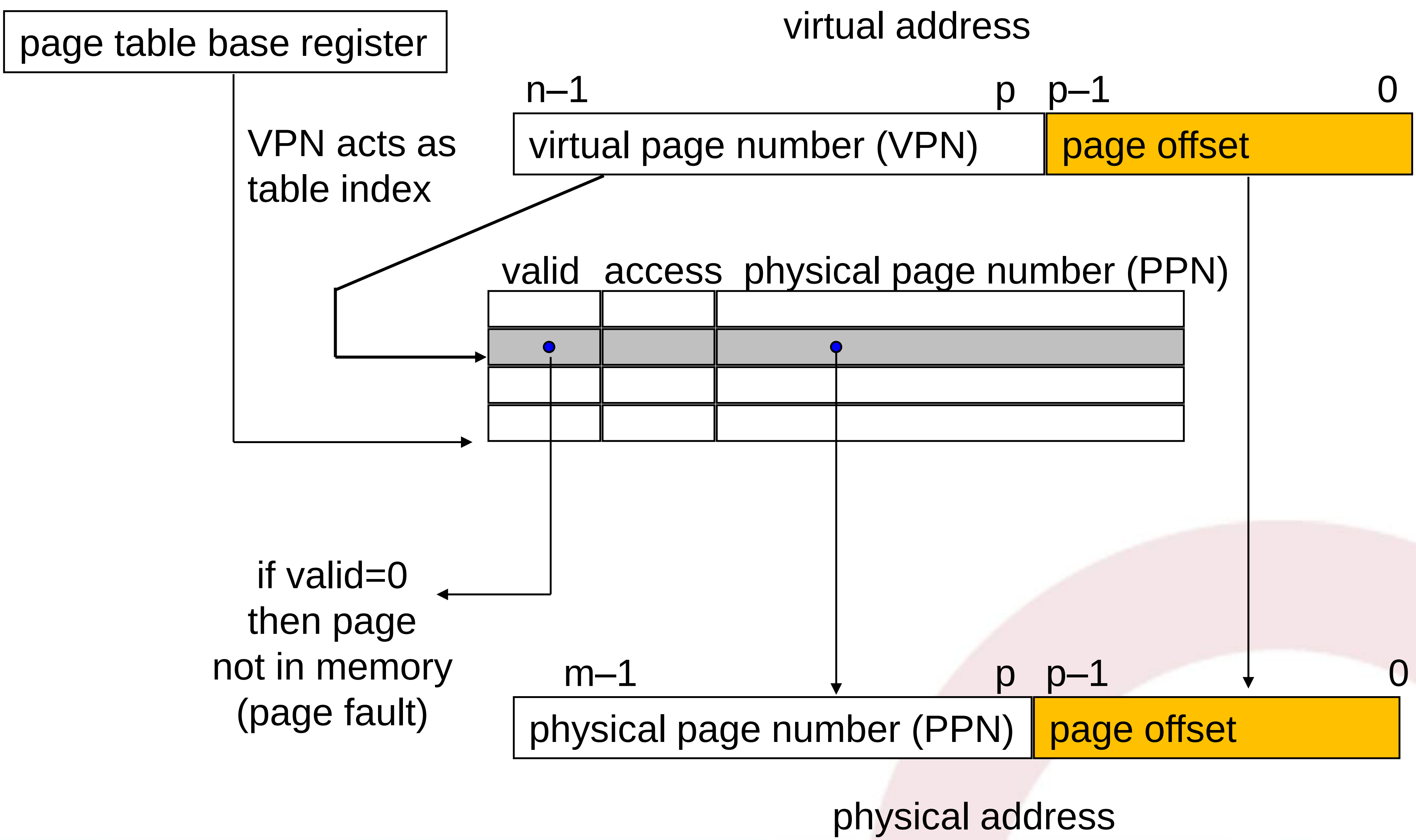


Page offset bits don't change as a result of translation

Page offset – P bits

- ✓ Page offset
  - – Number of bytes from the start of the page to the current address
  - – In a 4096-byte page (4 KiB), the offset of an address can be [0,4095]
    - Therefore, 12 bits (2^12 = 4096) are needed to represent the offset
      - – The least significant P bits of the virtual address represents the offset
        - » These P bits are the same ones of the physical address

Address Translation via Page Table

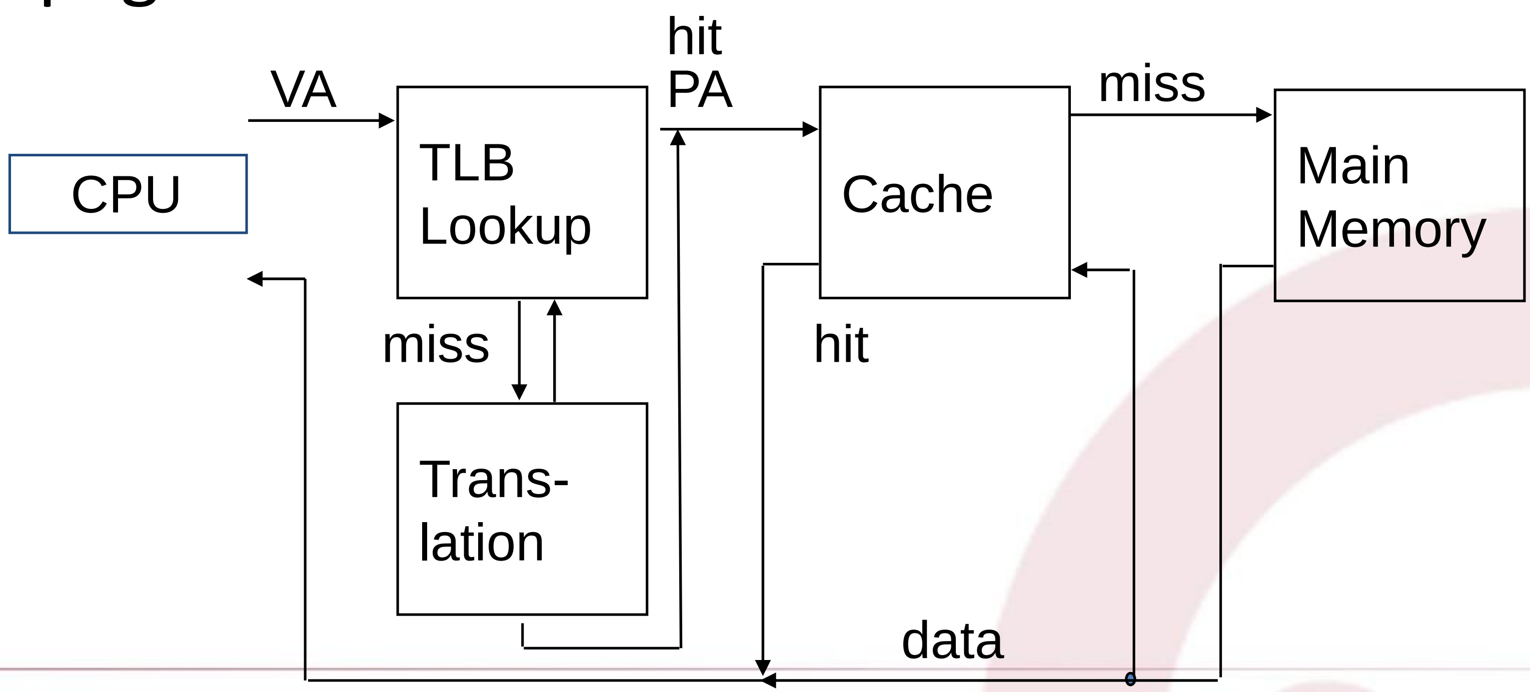


Page Table Entry

- ✓ Page Table Entry (PTE) provides information about page…
  - If (valid bit = 1) then page is in memory.
    - Use physical page number (PPN) to construct address
  - If (valid bit = 0) then page is on disk (or nonexistent)
    - Page fault
  - Checking protection
    - Access-rights field indicates allowable access
      - E.g., read-only, read-write, execute-only
      - Typically support multiple protection modes (e.g., kernel vs. user)
    - Protection-violation fault if user does not have necessary permission
      - Process is trying to access an address that is not in its space address

Speeding up Translation with a TLB

- "Translation Lookaside Buffer" (TLB)
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages



TLBs - intel i5 2410M@2.230GHz

- ✓ Data TLB: 2-MB or 4-MB pages, 4-way set associative, 32 entries

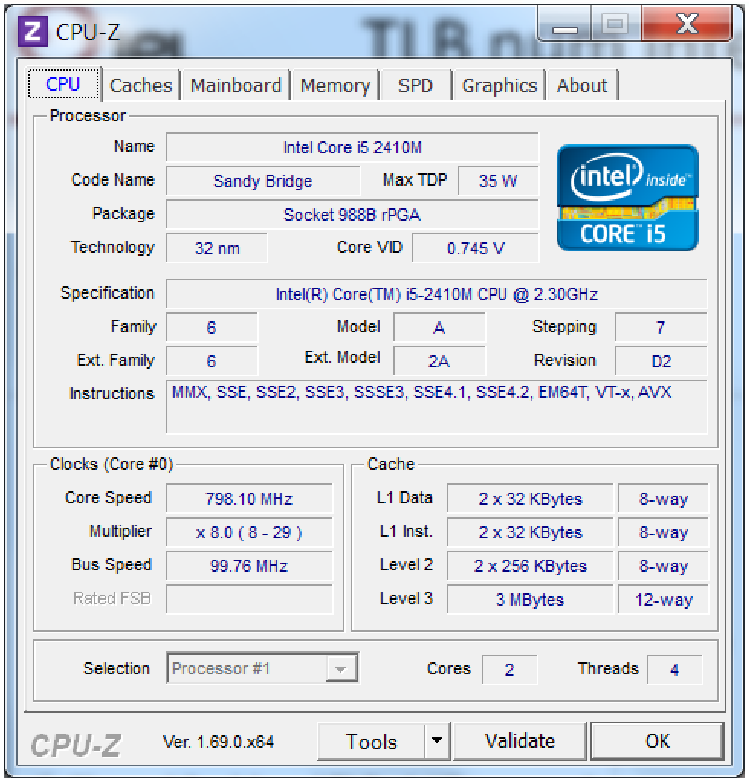
OR

- ✓ Data TLB: 4-KB Pages, 4-way set associative, 64 entries

- ✓ Instruction TLB: 4-KB pages, 4-way set associative, 64 entries

- ✓ L2 TLB: 1-MB, 4-way set associative, 64-byte line size

- ✓ Shared 2nd-level TLB: 4 KB pages, 4-way set associative, 512 entries

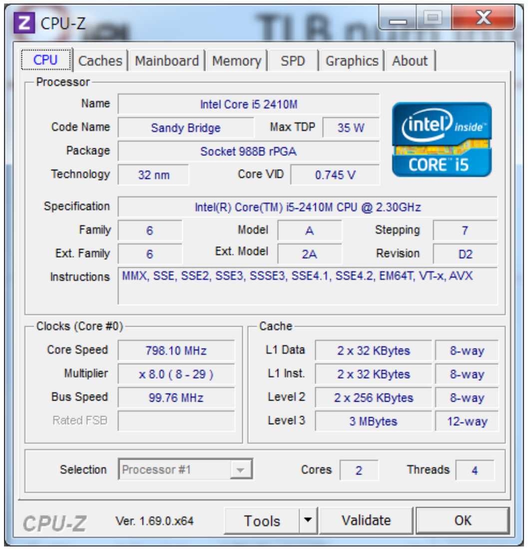


Fonte: http://www.cpu-world.com/sspec/SR/SR04B.html

coreinfo (application)

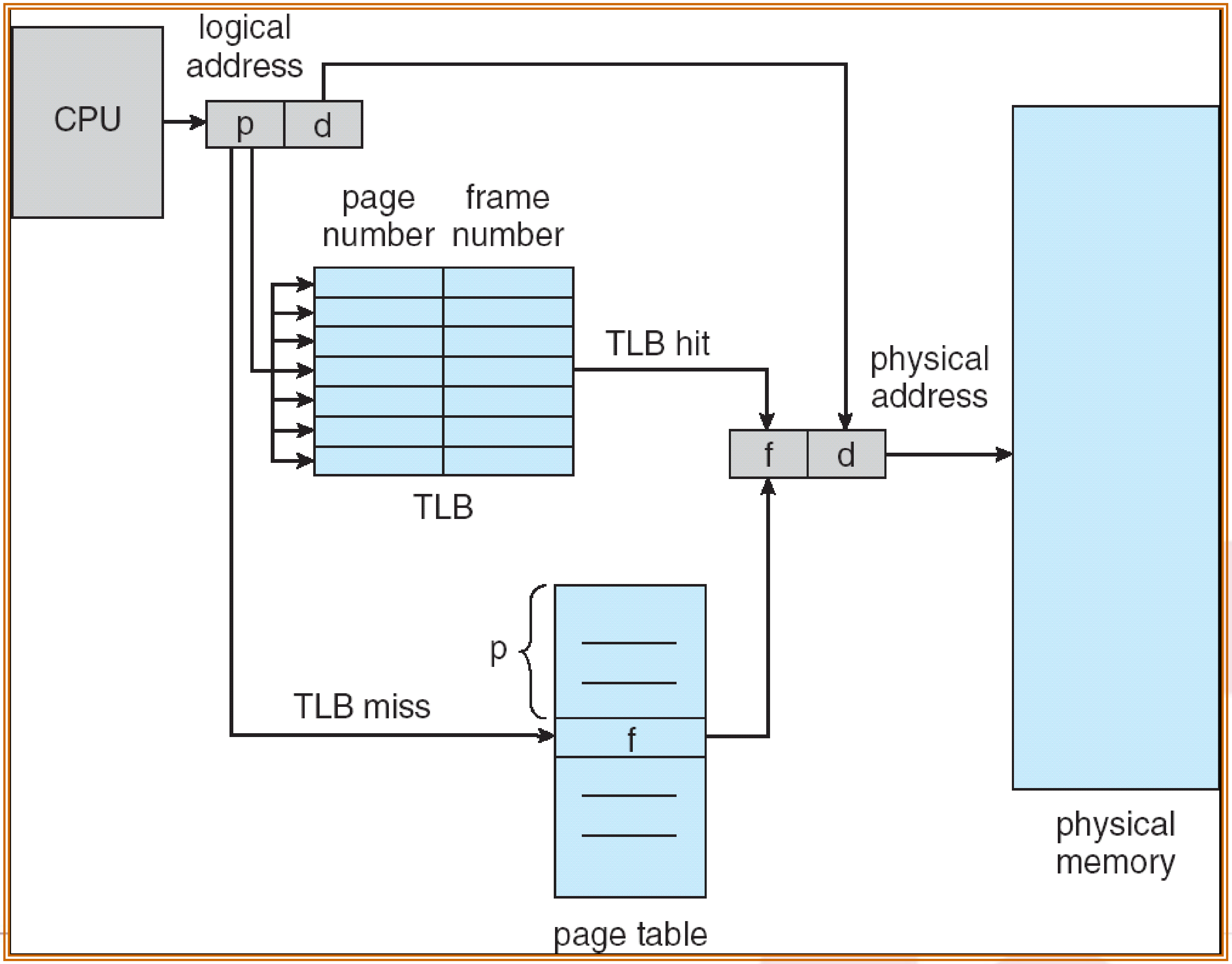
IPL
escola superior de tecnologia e gestão
instituto politécnico de leiria

- ✓ **coreinfo** application (sysinternals.com)
- ✓ List data regarding the CPU and the memory




```
C:\> coreinfo
Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz
Intel64 Family 6 Model 42 Stepping 7, GenuineIntel
**--  Data Cache          0, Level 1,   32 KB, Assoc   8, LineSize  64
**--  Instruction Cache   0, Level 1,   32 KB, Assoc   8, LineSize  64
**--  Unified Cache       0, Level 2,  256 KB, Assoc   8, LineSize  64
--**  Data Cache          1, Level 1,   32 KB, Assoc   8, LineSize  64
--**  Instruction Cache   1, Level 1,   32 KB, Assoc   8, LineSize  64
--**  Unified Cache       1, Level 2,  256 KB, Assoc   8, LineSize  64
****  Unified Cache       2, Level 3,    3 MB, Assoc  12, LineSize  64
```

Address translation with TLB



Cost of memory access with TLB

- Setup
  - A hit access to TLB costs 1 clock cycle (“TLB HIT”)
  - A miss access to TLB costs 30 clock cycles (“TLB Miss”)
  - The TLB miss rate is 1%
    - 100 accesses: 1 miss + 99 hits
- The memory access cost is:
  - 1 x 0.99 + (1+30) x 0.01 = 1.30
    - A 30% penalty!
- Therefore, it is of the utmost importance to achieve the lowest TLB miss rate!

page faults

JULIA EVANS
@b0rk

every Linux process has a page table

★ page table ★

virtual memory address	physical memory address
0x19723000	0x1422000
0x19724000	0x1423000
0x1524000	not in memory
0x1844000	0x4a000 read only

some pages are marked as either

★ read only

★ not resident in memory

when you try to access a page that's marked "not in memory", that triggers a ! page fault !

what happens during a page fault?

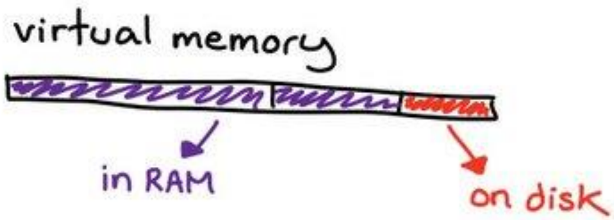
→ the MMU sends an interrupt

→ your program stops running

→ Linux kernel code to handle the page fault runs

Linux: I'll fix the problem and let your program keep running

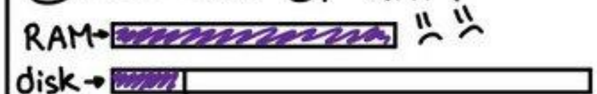
"not in memory" usually means the data is on disk!



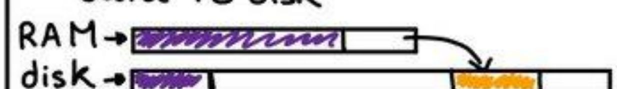
Having some virtual memory that is actually on disk is how swap and mmap work

how swap works

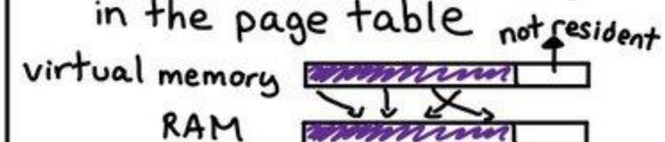
① run out of RAM



② Linux saves some RAM data to disk



③ mark those pages as "not resident in memory" in the page table



④ When a program tries to access the memory there's a ! page fault !

⑤ Linux: time to move some data back to RAM!

⑥ if this happens a lot your program gets VERY SLOW

I'm always waiting for data to be moved in & out of RAM

Types of page faults

- Three different types of page faults
  - Invalid fault (or access violation)
    - Caused when a program tries to access unallocated memory or tries to write to memory that's marked read-only.
  - Hard page fault
    - accessed memory is not currently in RAM (physical)
    - OS needs to retrieve the memory from disk (e.g. pagefile.sys) and make it accessible to the faulting process.
  - Soft page fault
    - memory is in RAM (physical)
    - but not currently accessible to the process that induced the fault.
      - Page might be shared amongst multiple processes and the process that caused the page fault might not have it mapped into its working set. These types of page faults are much more performant than hard page faults as there is no disk I/O conducted.

The page table *bloat*

- ✓ With
  - Page size of 4 KB + PTE of 4 bytes
  - A 32-bit address space requires a page table that can have… 4 MiB
  - The math…
    - 32-bit address → 2^32 bytes / 2^12 (4KiB) = 2^20 pages
    - Each page → 1 PTE, i.e. 2^20 x 4 bytes = 2^22 bytes = 4 MiB
- ✓ Since, the OS has one page table per process…
  - Page tables can potentially consume a lot of memory of the system
- ✓ The OS needs different memory organizations to avoid the page table *bloat*

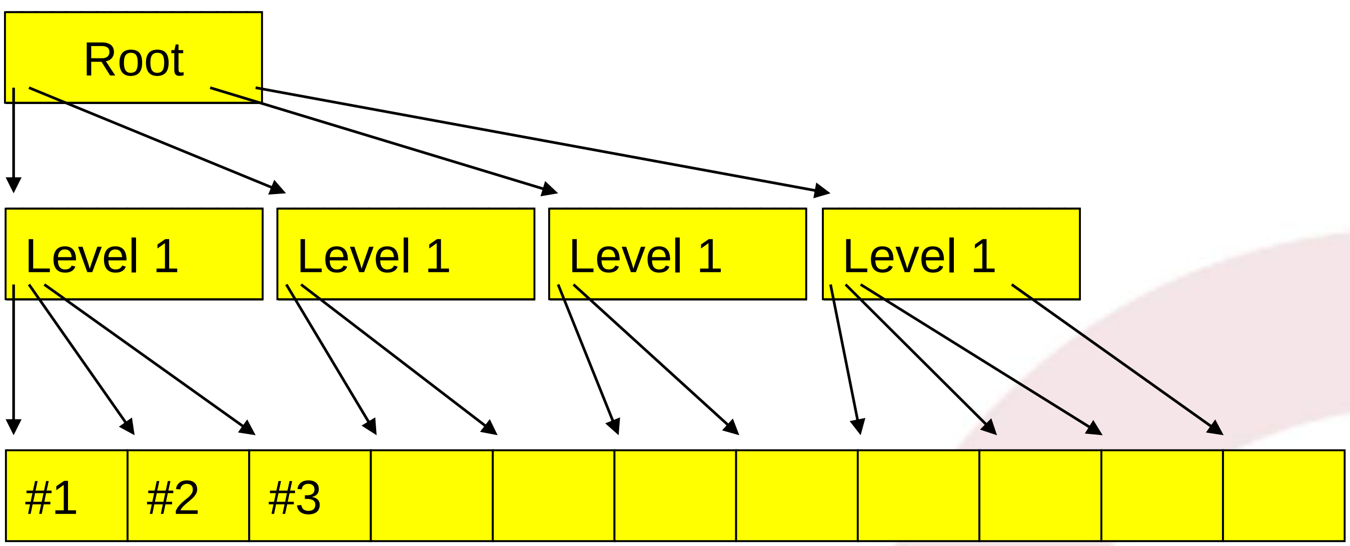
Allocation of pages

- ✓ Other approaches for page allocation
  - – Hierarchical pagination
  - – Associative table of pages (*hash*)
  - – Inverted page table

Next, we review each of these schemes >>

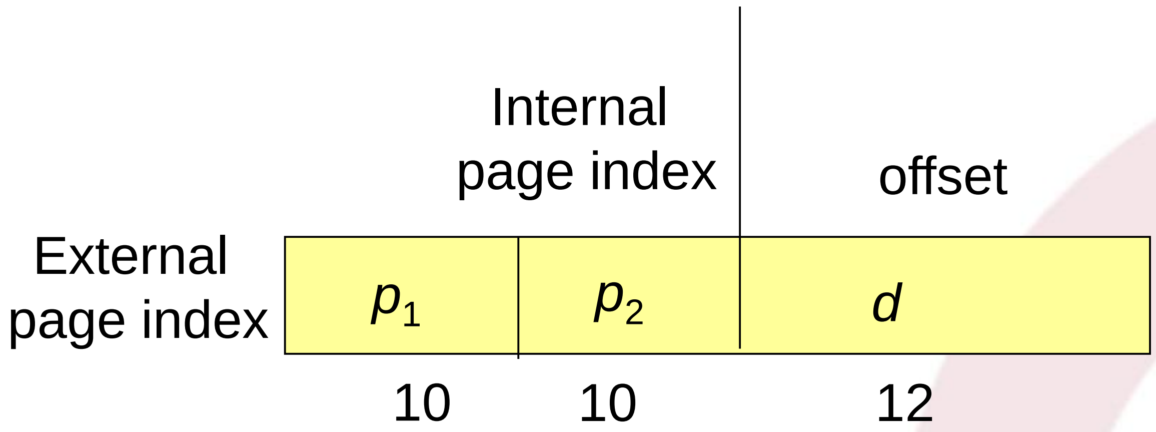
Hierarchical pagination (1)

- ✓ The page table is split through several levels
  - The *root level* of the page table needs to be always in RAM
    - The other levels of the page table can be…paginated



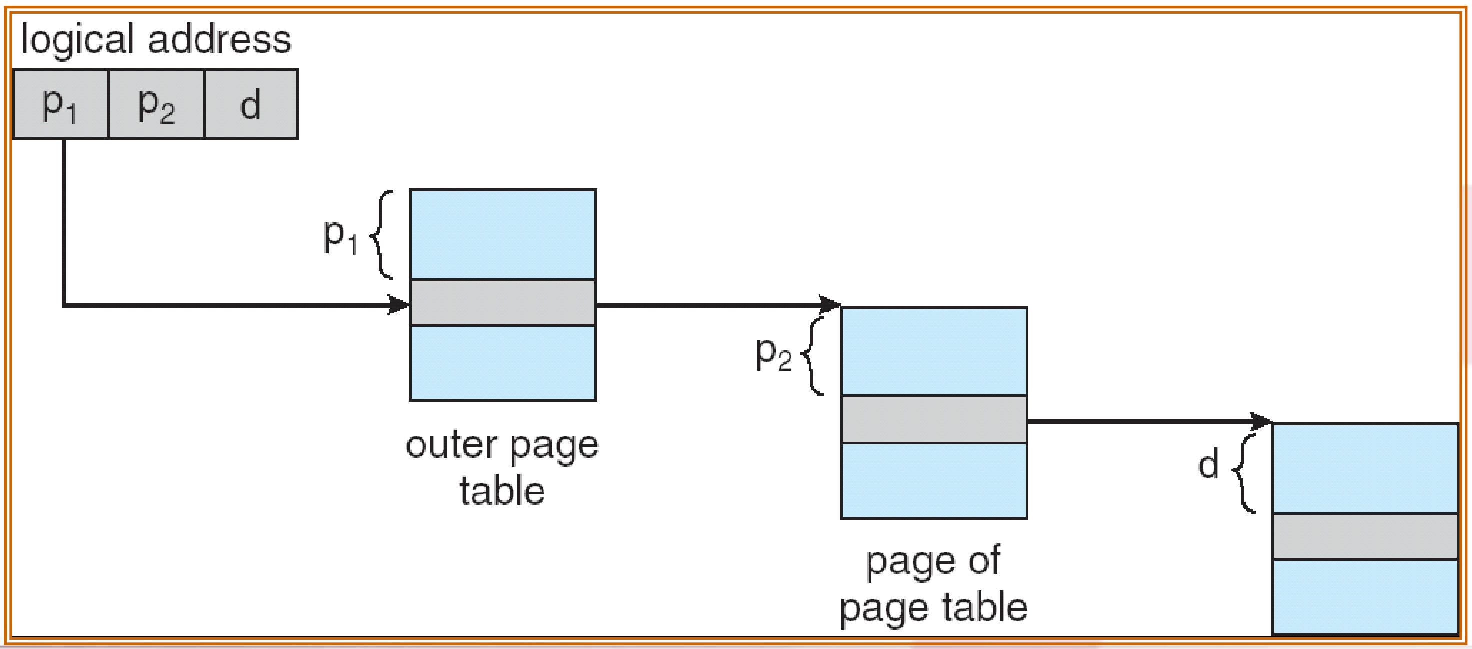
Hierarchical pagination (2)

- A logical 32-bit address in a 4-KiB page setting has:
  - Page number: 20 bits
  - Offset: 12 bits (2^12 = 4 KiB)
- The page table is…paginated
  - The page number P is further split in:
    - A page number p_1 with 10 bits
    - An offset) p_2 with 10 bits

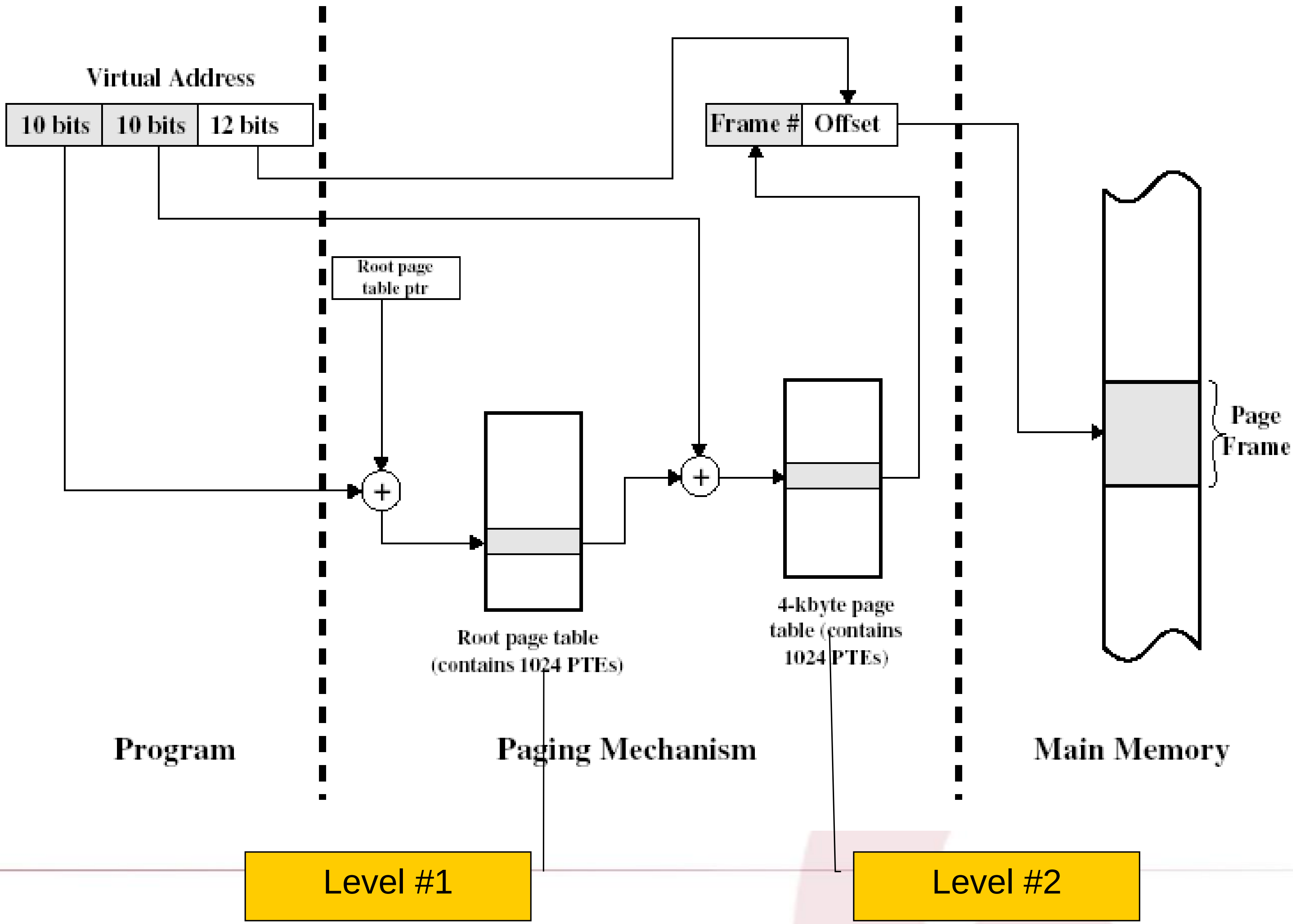


Address translation scheme

- ✓ TLBs are of paramount importance for the performance of address translation scheme
- ✓ On a 64-bit virtual address space, we need at least 3 levels of page table
- ✓ Typically: 32+10+10+12
  - – This is inefficient

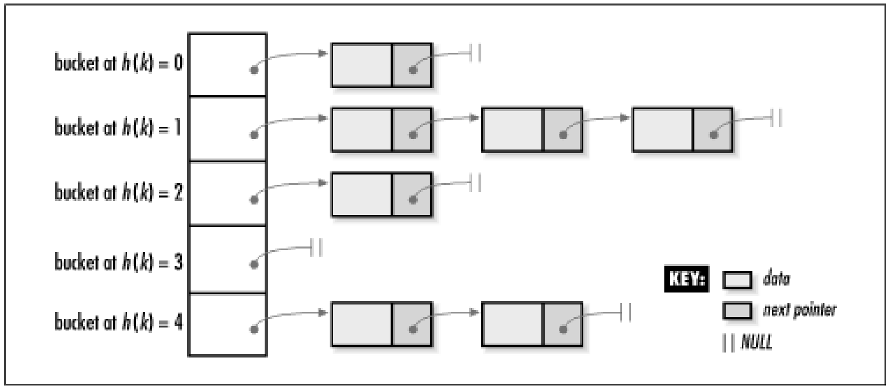


Example: two levels pagination



Associative page table (1)

- ✓ Based on the concept of hash tables
- ✓ A hash function is used to compute a hash ID
  - The input of the hash function is the virtual page ID
  - hash(virtual_page_ID) ➔ hashed_page_ID
- ✓ The *hashed_page_ID* identifies a destination page table
  - This destination page table holds a set of pages, all of them having the same hash
  - The page number ID is then searched on the destination page table
    - If found, then we have the PTE
    - If not found, we have a page fault

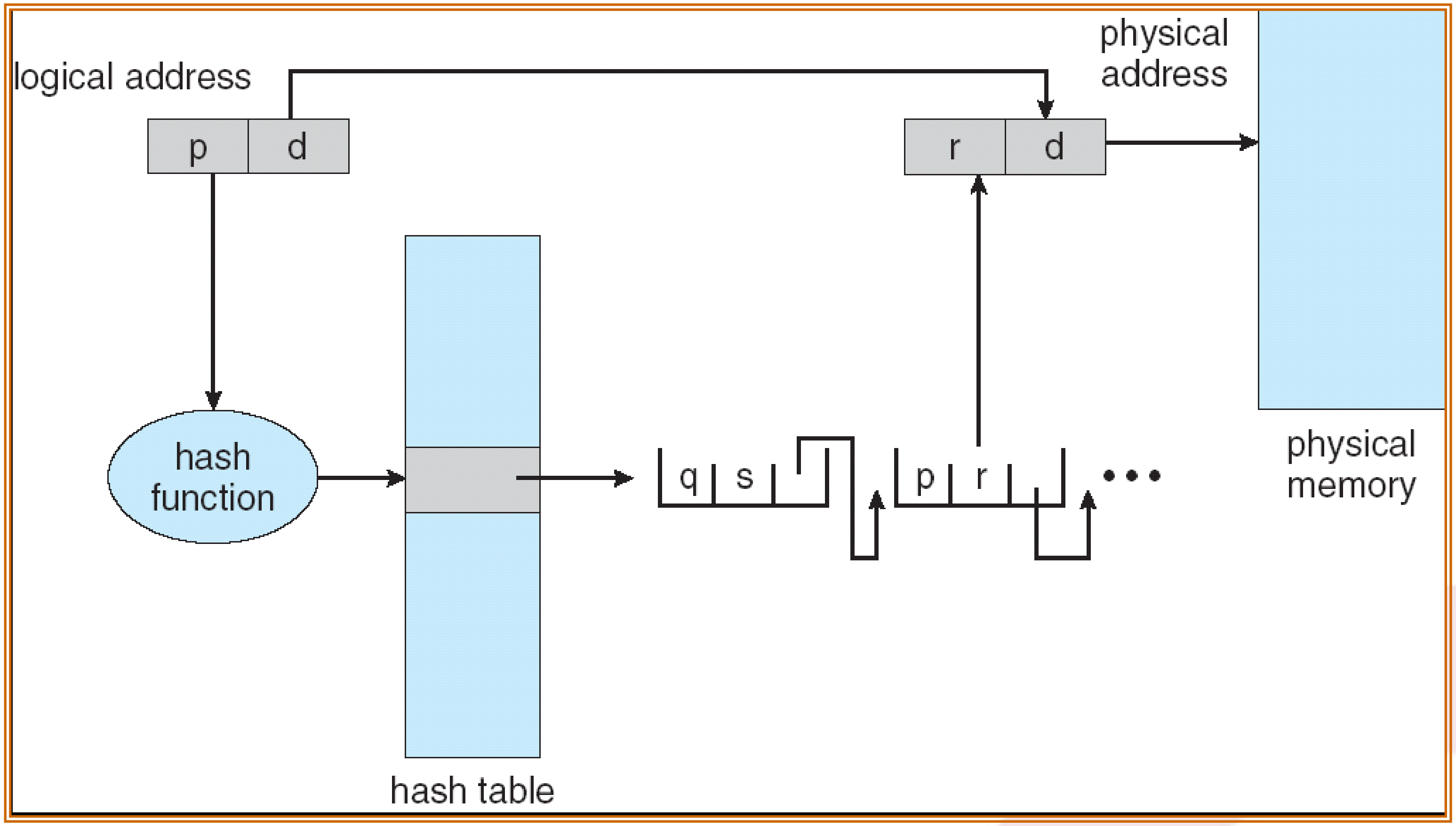


http://bit.ly/1Oypscg

Diagram >>

Associative page table (2)

✓ It works similarly to a hash table

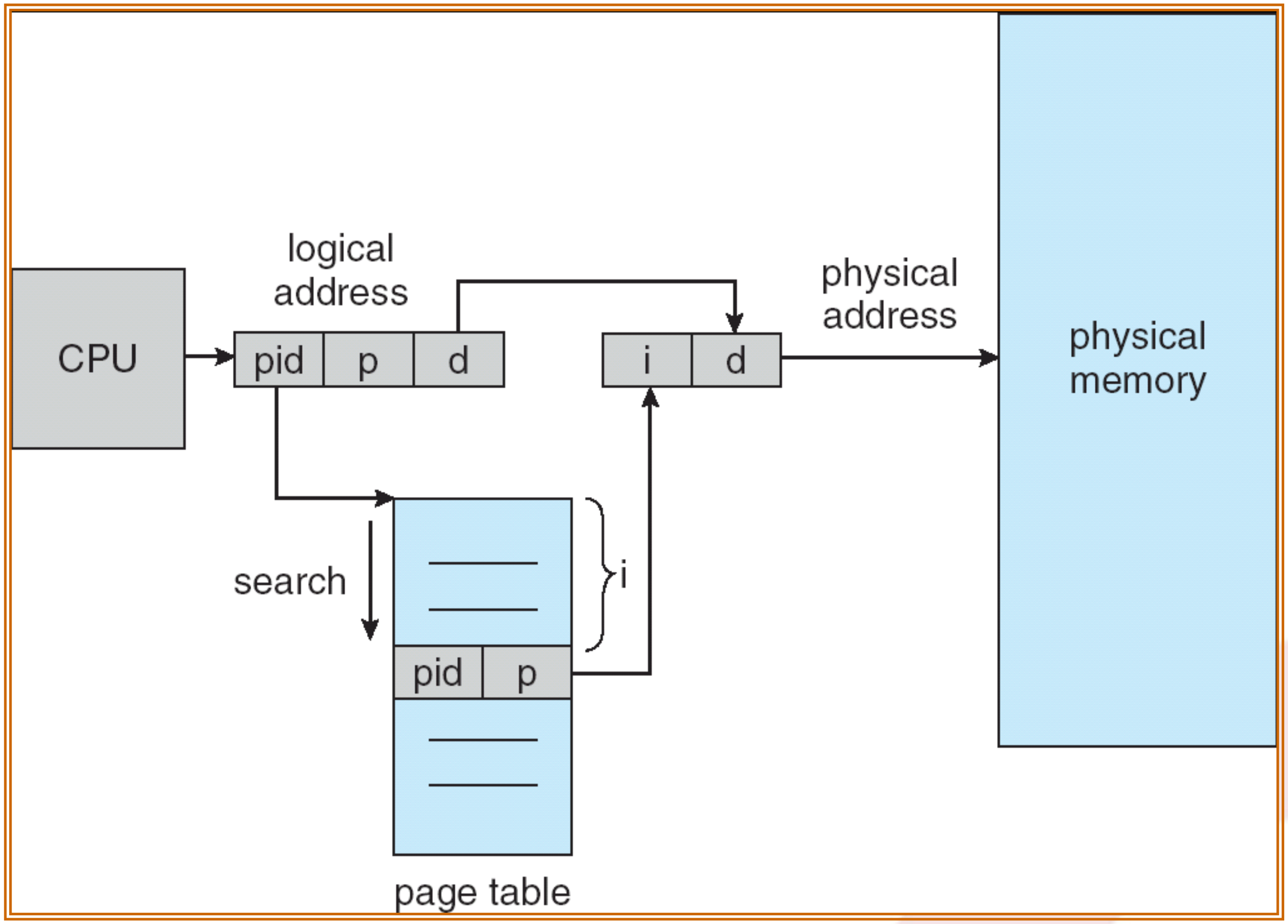


Inverted table of pages (1)

- One entry per page of **physical** memory
- An entry holds
  - Virtual address of the page that is kept at the physical address
  - Info regarding the process that holds the page
- This approach is used in 64-bit UltraSPARC and PowerPC
- Analysis
  - (+) It reduces the memory space devoted to the page table
  - (-) It increases the time for searching the whereabouts of a page

Diagram >>

Inverted table of pages (2)

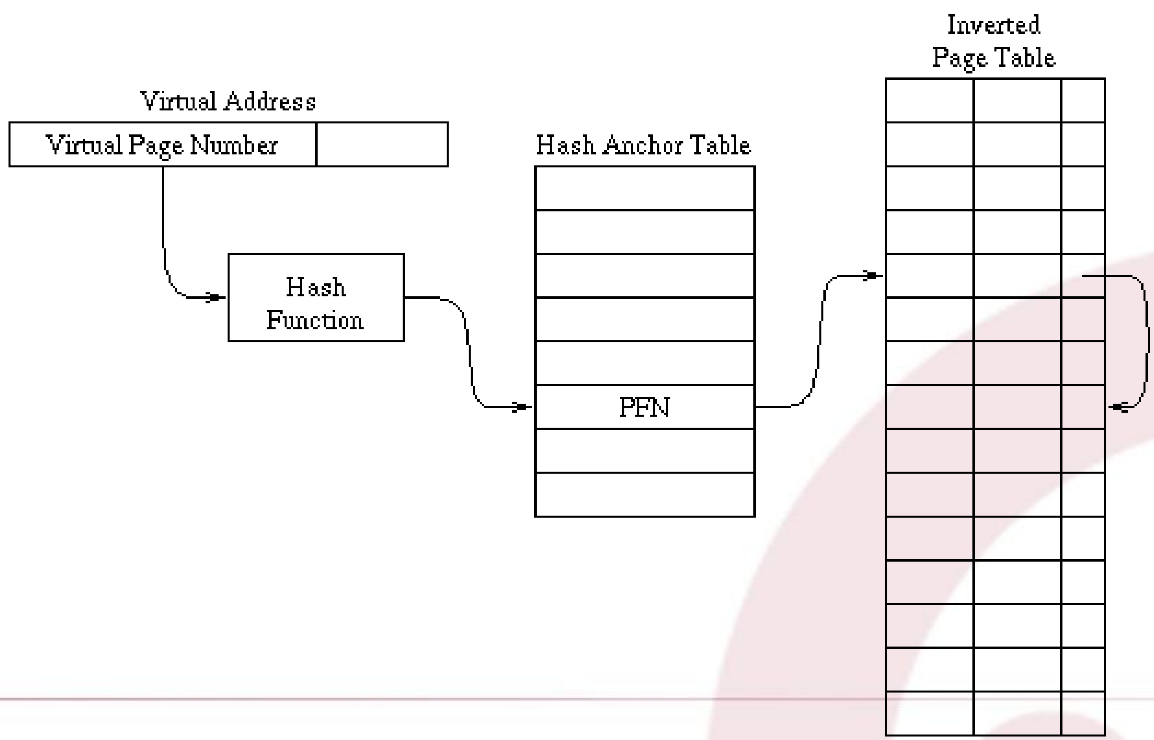


Inverted table of pages (3)

✓ The efficiency of the inverted table of pages can be increased with a hash

– The virtual page number (page ID) is hashed

- The hashed value is used to lookup the appropriate *bucket*

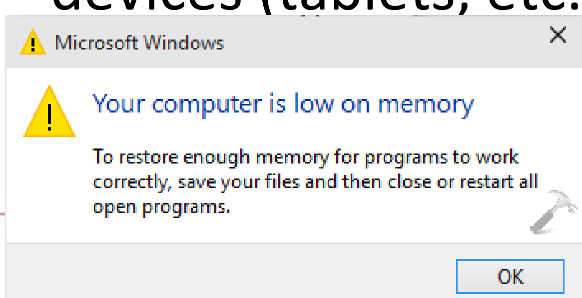


http://www.cs.nmsu.edu/~pfeiffer/classes/573/notes/ipt.gif

*.sys files in Windows (>=W8)

- Files
  - pagefile.sys
    - Holds swapped out pages
  - hiberfile.sys
    - For hibernation mode
  - swapfile.sys
    - Holds page **<u>recently</u>** swapped out of memory
    - Used for metro apps
    - Appropriate for low end devices (tablets, etc.)

Windows	2018/05/14 10:14	File folder	
Windows.old	2018/05/14 10:15	File folder	
bootmgr	2015/10/30 07:18	System file	391 KB
BOOTNXT	2015/10/30 07:18	System file	1 KB
hiberfil.sys	2018/05/20 17:56	System file	1 617 040 KB
pagefile.sys	2018/05/20 00:40	System file	8 131 264 KB
swapfile.sys	2018/05/14 10:10	System file	16 384 KB

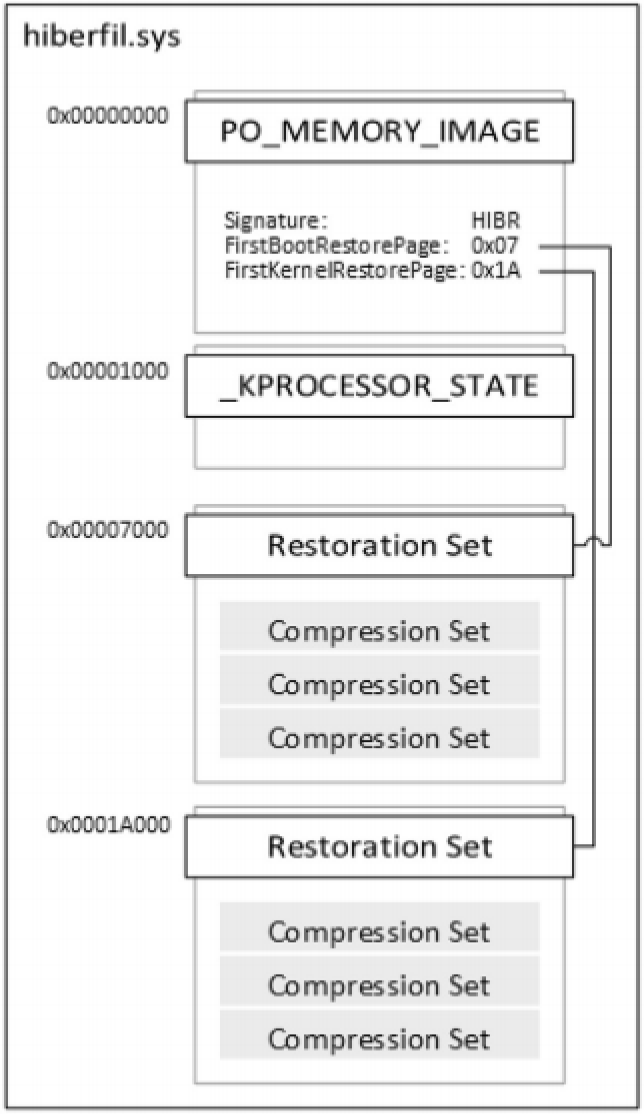


Hiberfil.sys >>

Hibernation (in W8+)

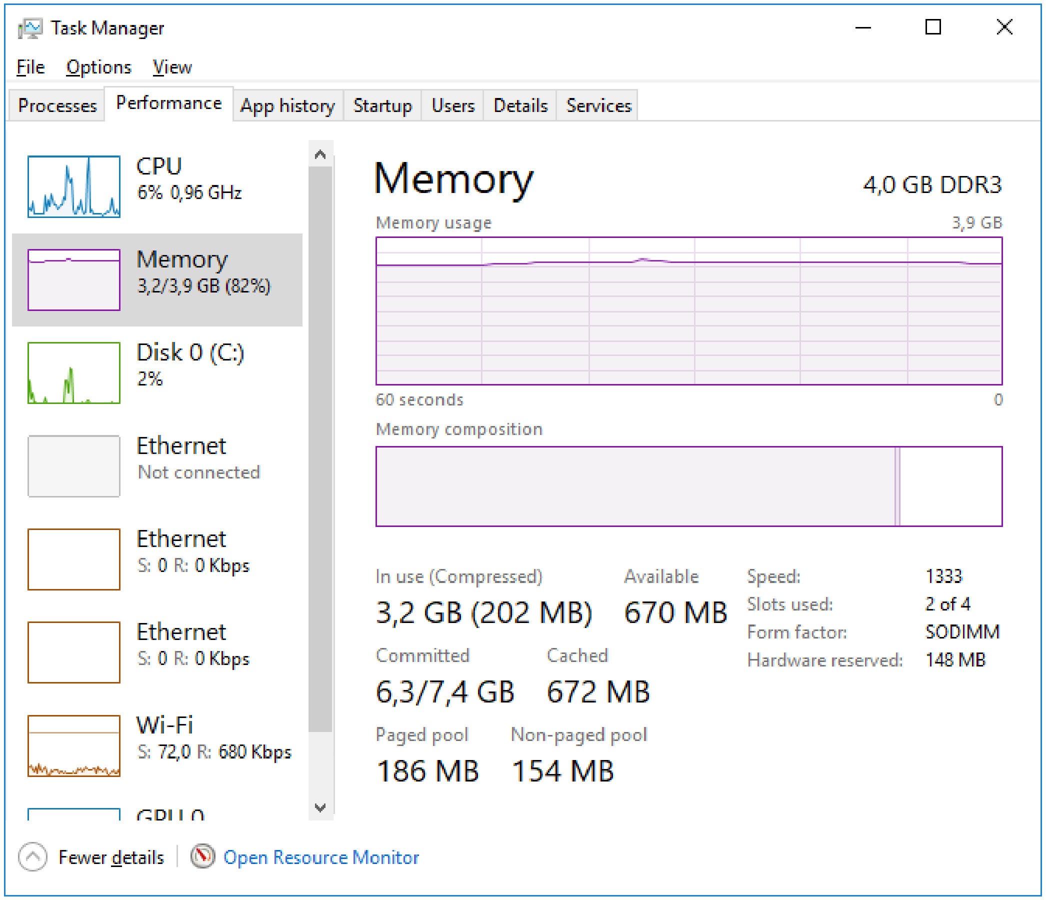
- The hiberfil.sys can be created with the shutdown command:
  - **`shutdown /h`**
    - State of memory is saved to hiberfil.sys
  - **`shutdown /s /hybrid`**
    - State of KERNEL (and only kernel) memory is saved to hiberfil.sys
- **+info:** *SYLVE, J.; MARZIALE, L.; RICHARD III, Golden G. Modern windows hibernation file analysis. Digital Investigation, 2016, 30: 1e7.*

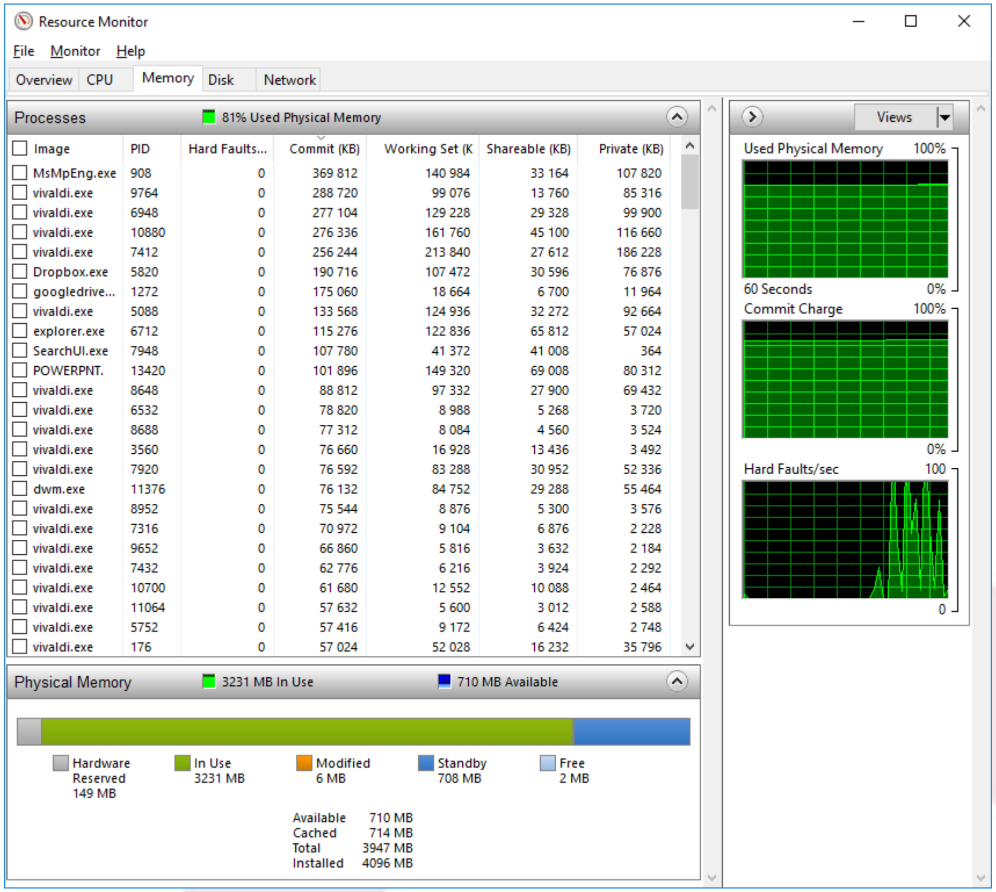
- Structure of hiberfil.sys for W8+



Task manager & Resource monitor

- Task manager
  - Ctrl+shift+esc
- Resource monitor
  - Perform.exe /res





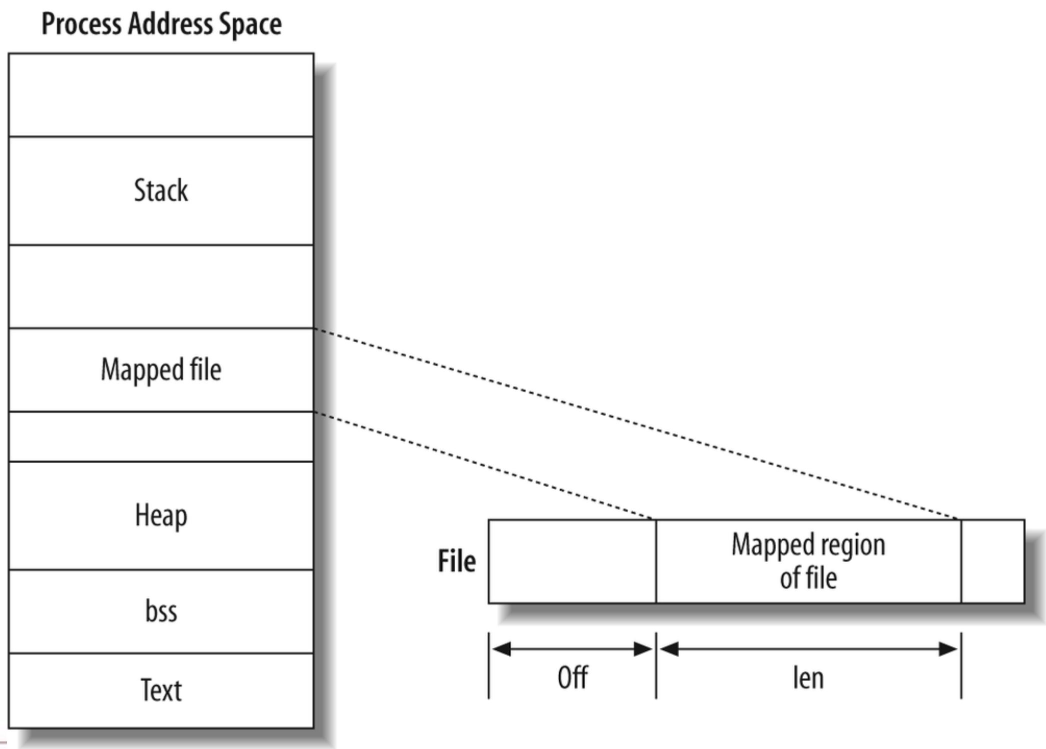
escola superior de tecnologia e gestão
instituto politécnico de leiria

mmap (#1)

- mmap: memory map
  - Map files or devices in memory
  - It allows to access a file or device just like accessing memory
- Prototype
  - `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
  - `int munmap(void *addr, size_t length);`

mmap

- ✓ A file is mapped to a zone of memory
  - – Acessing the file is done through the memory, just like acessing an array
  - – Mmap operates on pages. Mappings are multiple of page size



https://bit.ly/2IpeynW

mmap (#2)

- void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
  - **addr**: address where the mapping should be created. Can be NULL (kernel choses address of mapping)
  - **length**: size, in bytes, of the mapping
  - **prot**: memory protection
    - Can be PROT_NONE or PROT_EXEC | PROT_READ | PROT_WRITE
  - **flags**: controls whether the updates to the mapping are visible to other processes mapping the same region and whether updates are carried to the underlying file
    - MAP_SHARED, MAP_PRIVATE, MAP_ANONYMOUS
  - **fd**: file descriptor (real file or a device)
  - **offset**: where the mapping starts. Must be a multiple of the page size (**sysconf(_SC_PAGE_SIZE)**)

mmap (#3)

✓ mmap of a shared library

- A pointer to a block of memory is returned
- Accessing the pointer allows to interact (read/write) with the file
- The content of the file is loaded page by page, only when it is needed (e.g., a `read` operation)
- Dynamic libraries are loaded through mmap
  - "mapped in memory"
- Example: libc.so.6
  - Output of `strace ls`

```
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\320\207\1\0004\0\0\0
"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1786484, ...}) = 0
mmap2(NULL, 1792540, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0xb74fd000
```

unmap

- int munmap(void *addr, size_t length);
  - Deletes the mapping pointed by `addr` up to length bytes
    - All pages within range of *length* bytes are unmapped
    - Length needs NOT to be a multiple of the page size
  - Referencing `addr` after `munmap` generates invalid memory references
- Note:
  - A mapped region is NOT unmapped when the file descriptor used in mmap is closed
  - A mapped region is automatically unmapped when the process ends

Advantages of `mmap`

- ✓ Reading/writing memory-mapped file avoids the extraneous copy that occurs when using **read** or **write** syscalls
  - – read/write: data must be copied to/from user-space buffer
- ✓ Reading/writing a memory-mapped file does not incur any system call or context switch overhead. It is as simple as accessing memory.
  - – Except when a page faults occurs
- ✓ Multiple processes can map the same object into memory
  - – data is shared among all the processes

Source: https://bit.ly/2Grmnrh

Disadvantages of mmap

- ✓ Memory mappings are always an integer number of pages in size
  - – Difference between size of backing file and an integer number of pages is "wasted" as slack space
  - – A significant percentage of the mapping may be wasted in small files
    - 4 KB pages, a 7 byte mapping wastes 4,089 bytes

- ✓ Overhead in creating and maintaining the memory mappings and associated data structures inside the kernel
- ✓ **Conclusion**
  - – Benefits of mmap are most greatly realized
    - when the mapped file is large
      - – Wasted space is a small percentage of the total mapping
    - When the total size of the mapped file is evenly divisible by the page size
      - – There is no wasted space

Source: https://bit.ly/2Grmnrh

mmap – example (#1)

```
#include <…>
int main (int argc, char *argv[]){
        struct stat sb;
        off_t len;
        char *p;
        int fd;
       if (argc < 2) {
               fprintf (stderr, "usage: %s <file>\n", argv[0]);
                return 1;
       }
        fd = open (argv[1], O_RDONLY);
        if (fd == -1) {
                perror ("open");
                return 1;
        }
```

(continue) >>

mmap – example(#2)

(…)

```
if (fstat(fd, &sb) == -1) {
        perror ("fstat");
        return 1;
}
if (!S_ISREG(sb.st_mode)) {
        fprintf (stderr, "%s is not a file\n", argv[1]);
        return 1;
}
p = mmap (0, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
if (p == MAP_FAILED) {
        perror ("mmap");
        return 1;
}
if (close (fd) == -1) {
        perror ("close");
        return 1;
}
```

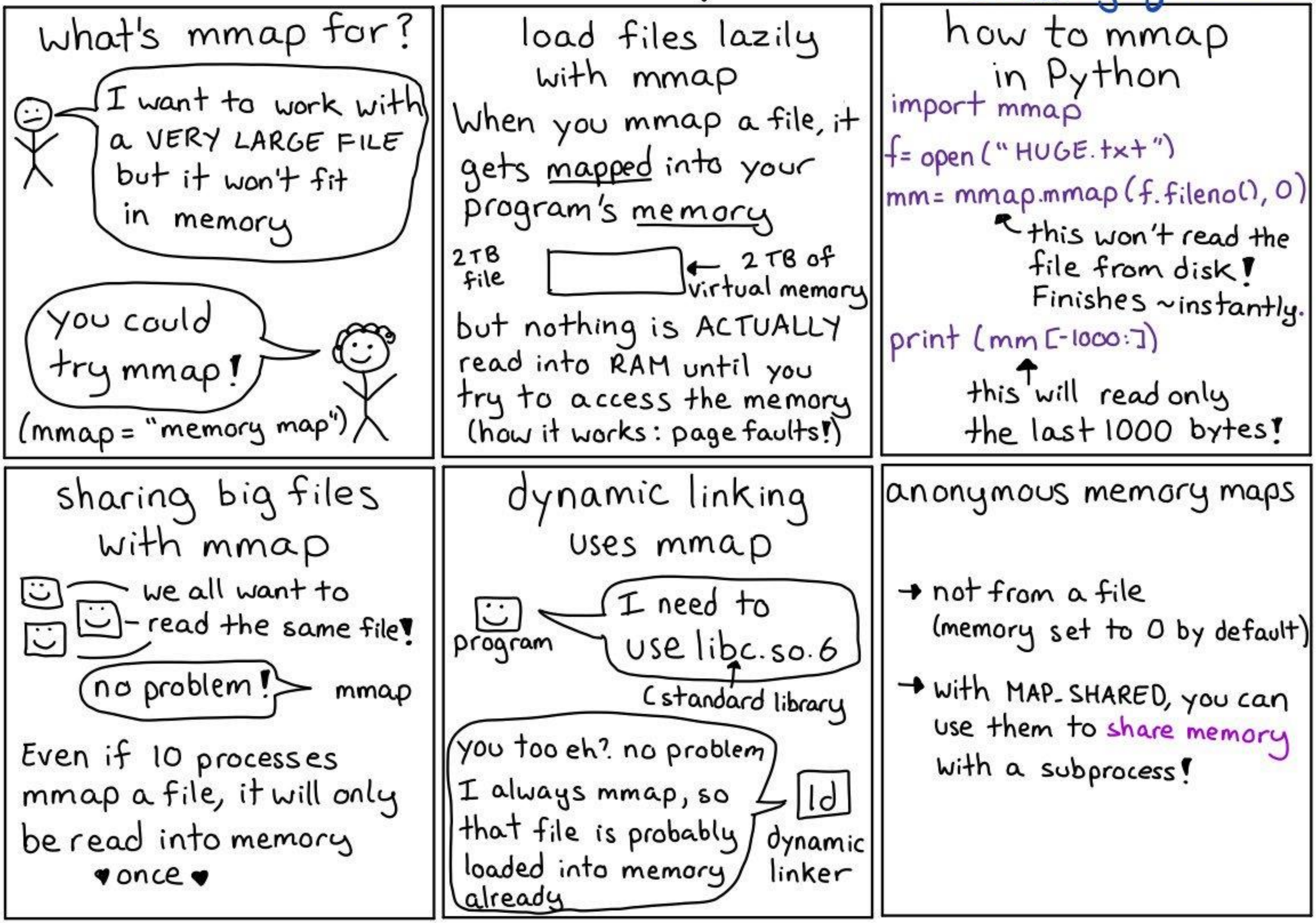
mmap – example (#3)

```
(…)
        for (len = 0; len < sb.st_size; len++)
                putchar (p[len]);

        if (munmap (p, sb.st_size) == -1) {
                perror ("munmap");
                return 1;
        }

        return 0;
}
```

JULIA EVANS
@b0rk

mmap

drawings.jvns.ca



man pages = awesome

JULIA EVANS
@b0rk

man pages are split up into 8 sections

① ② ③ ④ ⑤ ⑥ ⑦ ⑧

$ man 2 read

means "get me the man page for read from section 2"

There's both
→ a program called "read"
→ and a system call called "read"

so

$ man 1 read

gives you a different man page from

$ man 2 read

If you don't specify a section, man will look through all the sections & show the first one it finds

man page sections

1 programs

$ man grep
$ man ls

2 system calls

$ man sendfile
$ man ptrace

3 C functions

$ man printf
$ man fopen

4 devices

$ man null
for /dev/null docs

5 file formats

$ man sudoers
for /etc/sudoers
$ man proc
files in /proc!

6 games

not super useful.
$ man sl
is funny if you have sl though.

7 miscellaneous

explains concepts!

$ man 7 pipe

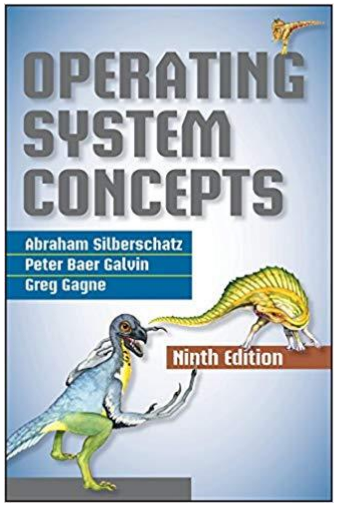
$ man 7 symlink

8 sysadmin programs

$ man apt

$ man chroot

Bibliography



- Chapters 8 & 9 of "Operating Systems Concepts", A. Silberschatz, 9th edition, 2016

- "Advanced File I/O.", Chapter 4 - Linux System Programming, Robert Love, 2nd Edition, O'Reilly, 2013
  - `mmap/umap`