Scuola universitaria professionale della Svizzera italiana

SUPSI

Memory management

Operating Systems

Amos Brocco, Lecturer & Researcher

14 settembre 2018

Objectives

- Understand how memory management works
- Understand segmentation and paging

⏩ **Browsing**

- Get a rapid overview.

▶ **Reading**

- Read it and try to understand the concepts.

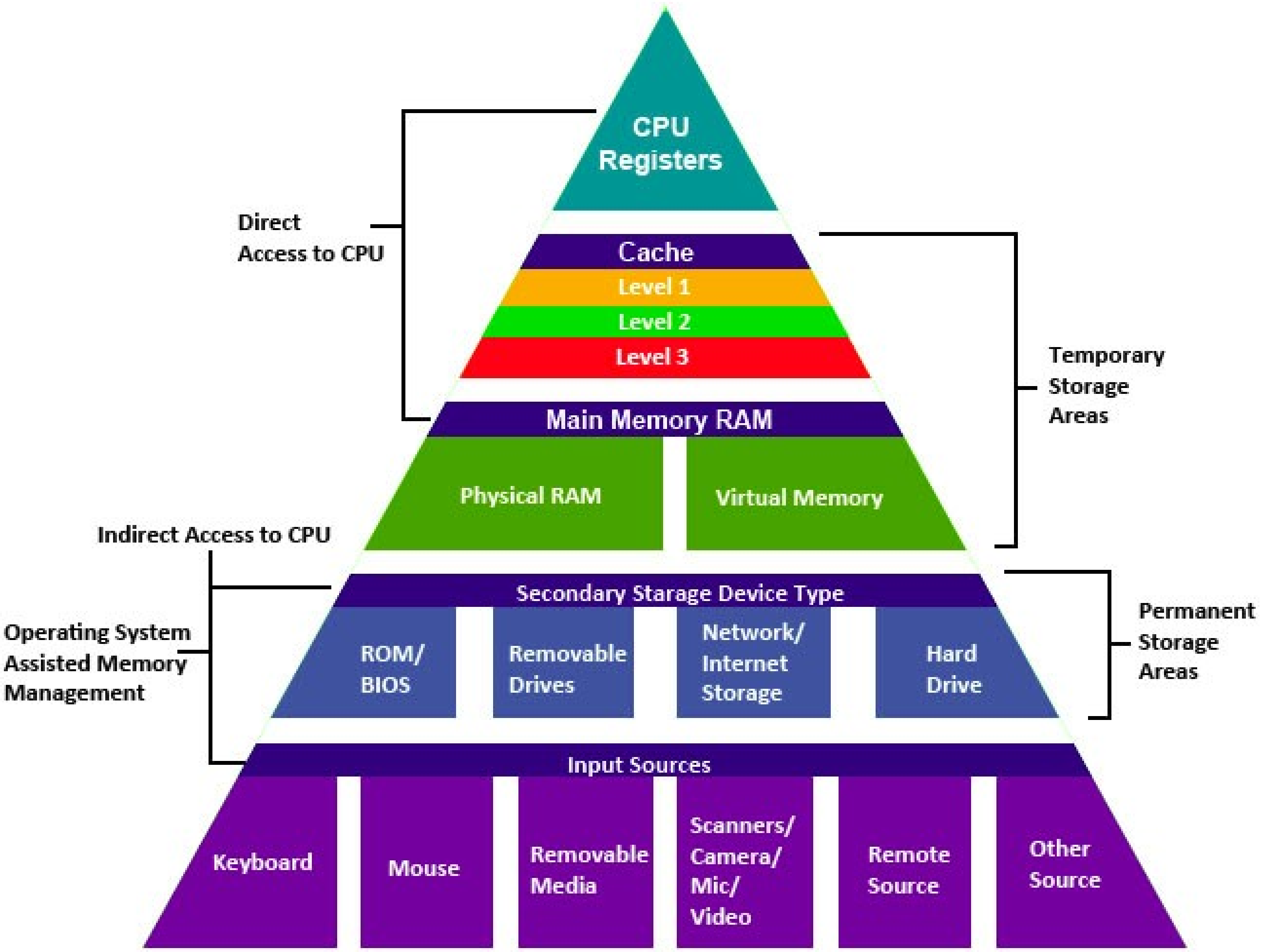
📖 **Studying**

- Read in depth, understand the concepts as well as the principles behind the concepts.

You are also encouraged to try out (compile and run) code examples!

Memory hierarchy

From http://cse1.net/recaps/4-memory.html



Virtualizing memory

- Each process must get enough memory for its code and data...

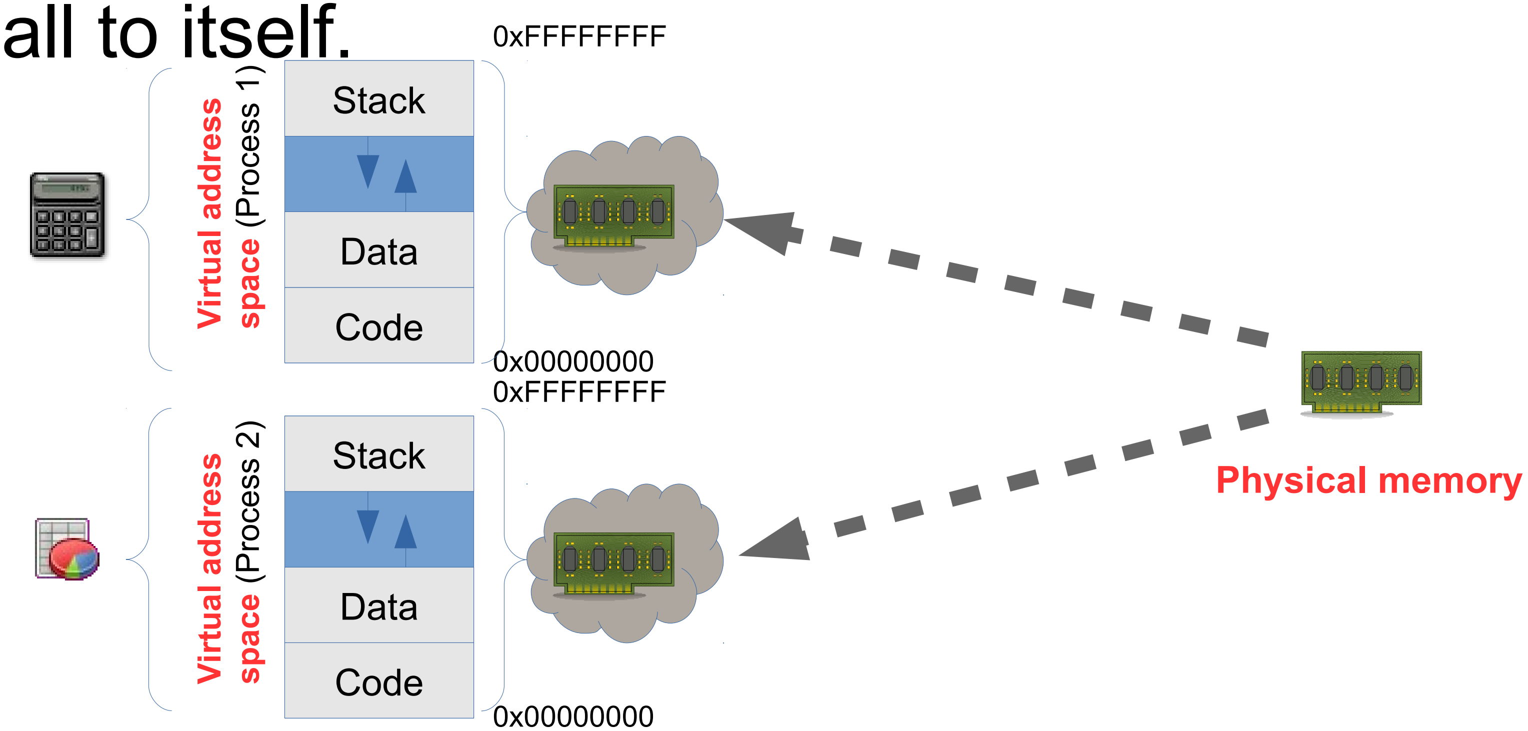
  ... what if two processes read or write the same set of addresses?

  ... what if physical memory is not enough for a process?

  ... what if physical memory is not enough for all processes?

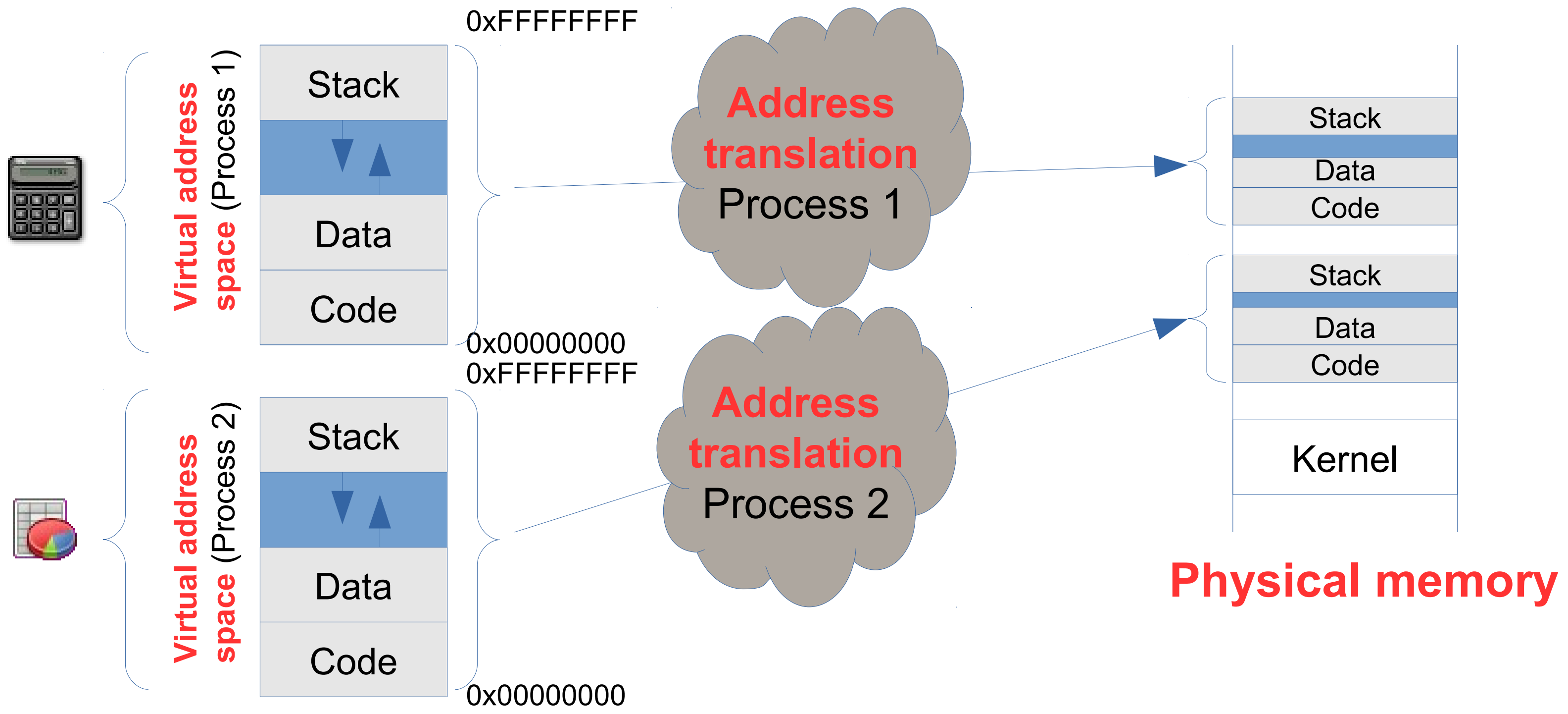
Virtualizing memory

- The operating system **abstracts/virtualizes memory** by giving each process its own **virtual address space** and the impression it has memory all to itself.



Virtualizing memory

- **Address translation** is used to **map virtual addresses** (used inside the program code) to **physical addresses**
  - provides **isolation** between processes (cannot read/write memory of another process)


0xFFFFFFFF
Virtual address space (Process 1)
Stack
Data
Code
0x00000000
0xFFFFFFFF
Virtual address space (Process 2)
Stack
Data
Code
0x00000000
Address translation Process 1
Address translation Process 2
Stack
Data
Code
Stack
Data
Code
Kernel
Physical memory

Why all this?

- **Simplify multiprogramming**
  - It is very difficult to ensure that two programs always use different memory addresses to store their data
- **Protect processes and data**
  - We want to prevent processes from spying on other processes or overwrite data belonging to other processes or the operating system
- **Hide physical limitations**
  - For example, that available physical memory is different on each computer (and might be less than expected by the programmer)

Early virtual address spaces: Intel 8086 (Real Mode) segmentation

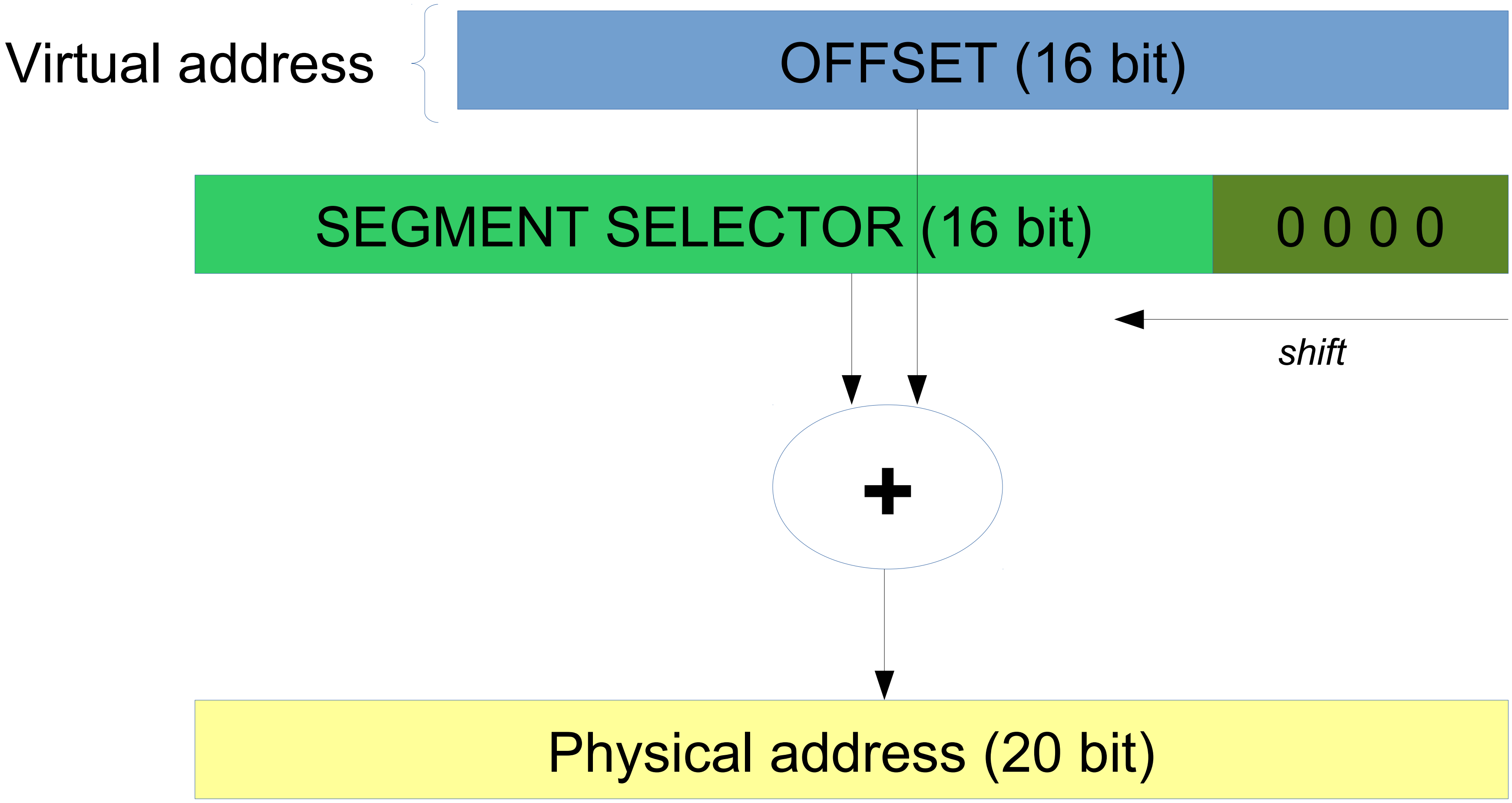
- With 16 bit processors (like Intel 8086) we can only address 2^{16} = 64kB of memory

- How to handle more memory?
  - We can rework the architecture of the whole CPU (registers, bus,...) to a handle more bits... a little bit too complex!
  - ... or we can just increase the size of the address bus

Intel 8086 (Real Mode) segmentation

- Computer programs use memory for different purposes:
  - Several intervals of contiguous memory are used for different types of data
    - Machine code
    - Global variables
    - Stack
    - ...
- We can divide the memory in contiguous 64 kB areas, called **segments** (which can be easily addressed using 16 bit)
  - When a program wants to access a segment it sets a **selector register** and then all **memory references** will end up being an **offset from the segment's base address**.

Intel 8086 (Real Mode) segmentation



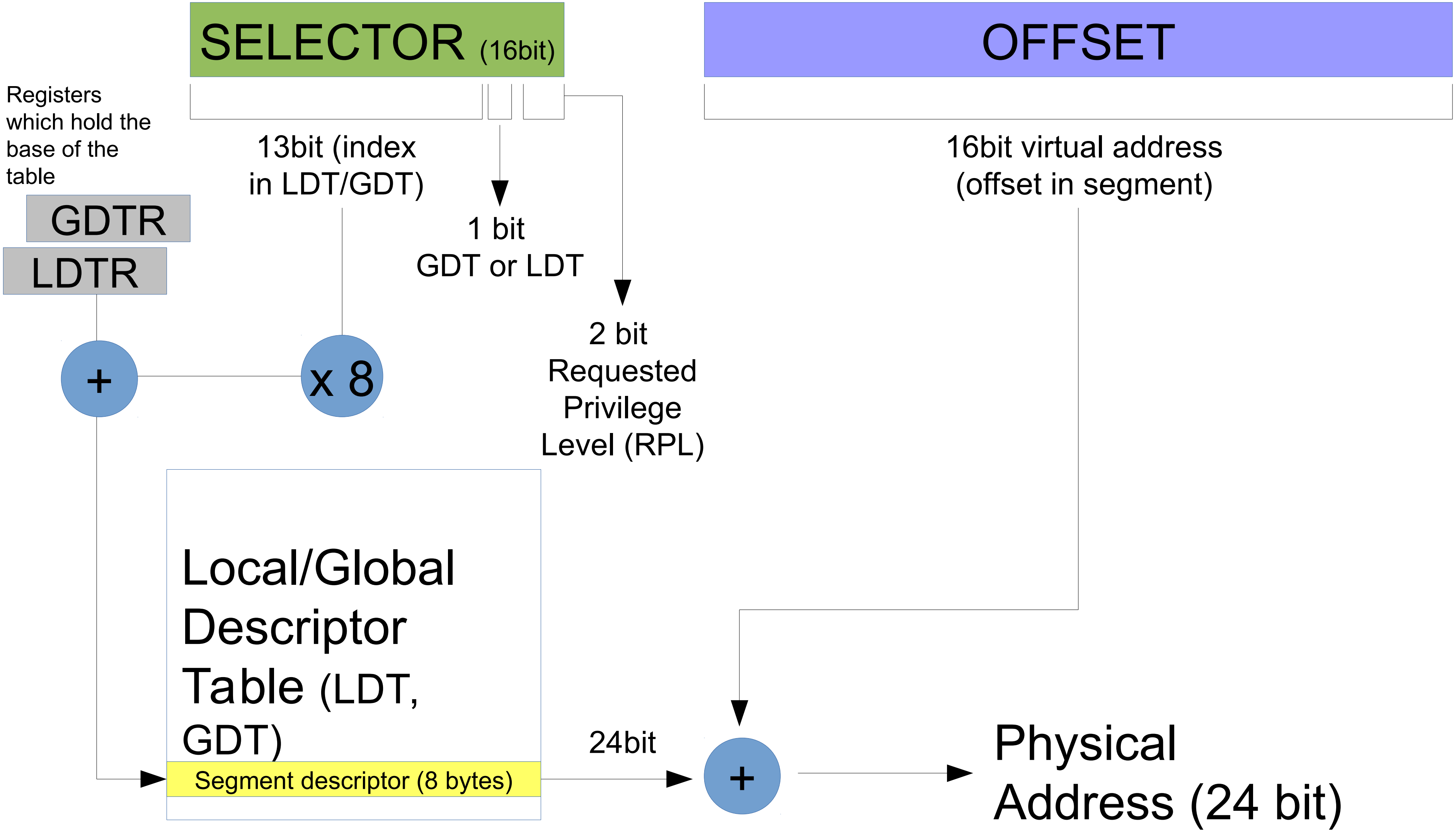
Intel 8086 (Real Mode) segmentation

- The resulting physical address **20 bit**:
  - 2^{20} bytes of memory are accessible (1MB)
  - several 16-bit registers are used as selectors: **code segment** (CS), **data segment** (DS), **stack segment** (SS), **extra segment** (ES)
    - The selector register is selected by the CPU depending on the type of instruction
  - Because segments might overlap there exist multiple combinations **selector:offset** to represent the same physical address (more precisely 2^{12}=4096)

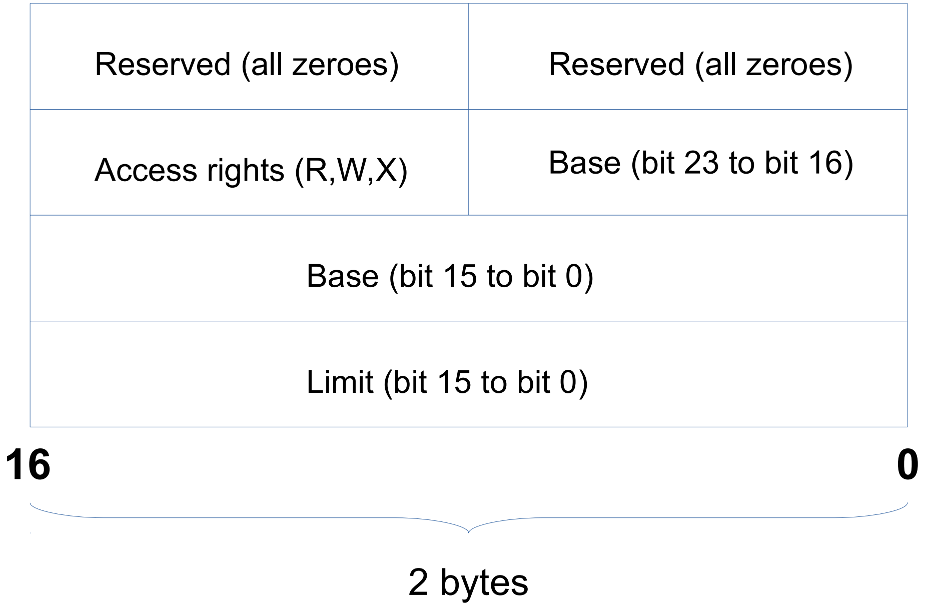
Segmentation evolved: Intel 80286 protected mode segmentation

- On Intel 80286 segmentation was extended to allow for variable length segments, protection, etc.
  - Segments were managed using in-memory **descriptor tables** (per process **LDT**, and a global **GDT**)
- 24 bit physical address (max. 16 Megabytes)
- 64 kbytes **segments**
  - Descriptors define the **base address** and the **limit** (size)
  - Segments have a user-defined **protection** level (R,W,X) and access **privilege level**
  - Segments can be **shared** by processes
  - Segments have to be **declared** before being used
    - This is done when processes allocate memory

Intel 80286 address translation (16 bit architecture, 24 bit addresses)


SELECTOR (16bit)
OFFSET
Registers which hold the base of the table
GDTR
LDTR
13bit (index in LDT/GDT)
1 bit GDT or LDT
2 bit Requested Privilege Level (RPL)
16bit virtual address (offset in segment)
+
x 8
Local/Global Descriptor Table (LDT, GDT)
Segment descriptor (8 bytes)
24bit
+
Physical Address (24 bit)

Segment descriptor on Intel 80286



Reserved (all zeroes)

Reserved (all zeroes)

Access rights (R,W,X)

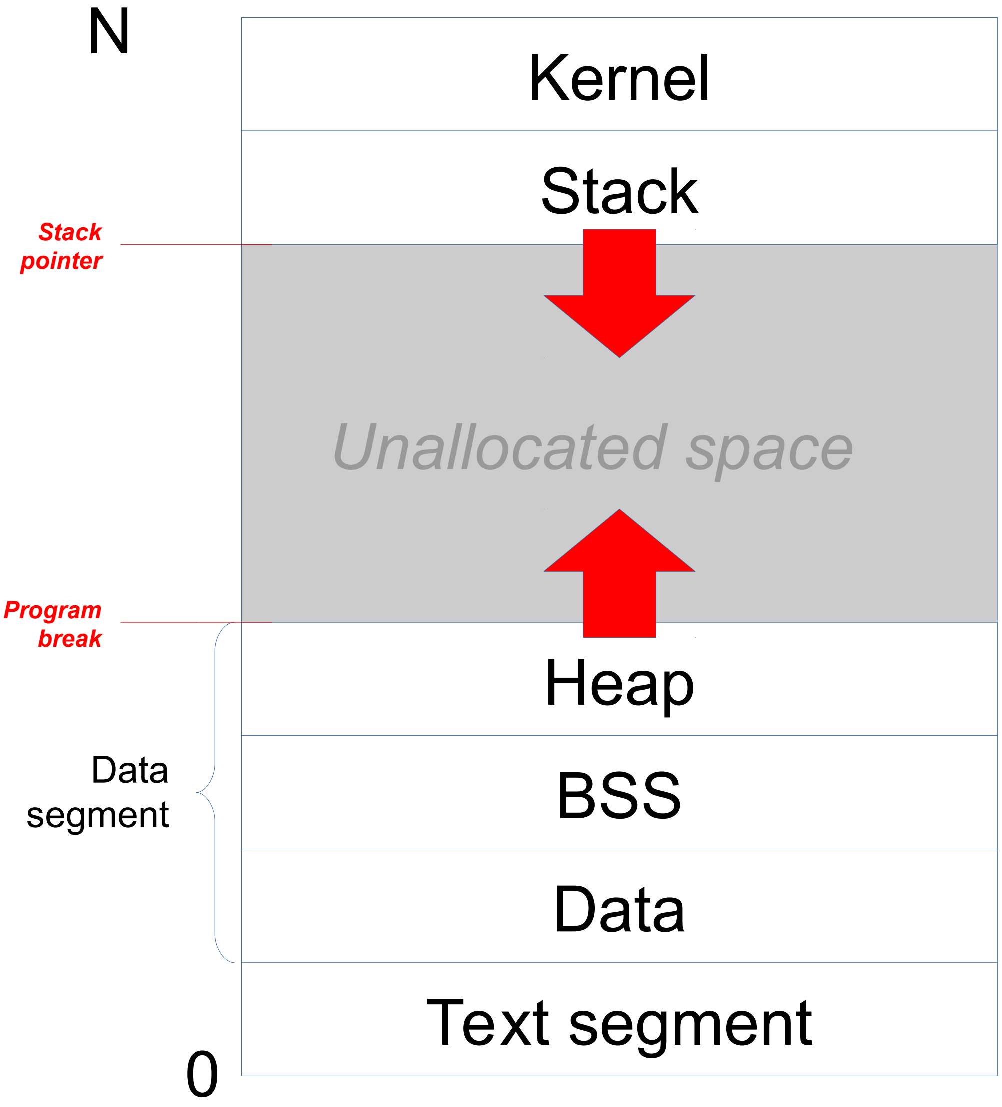
Base (bit 23 to bit 16)

Base (bit 15 to bit 0)

Limit (bit 15 to bit 0)

Segmentation: in action

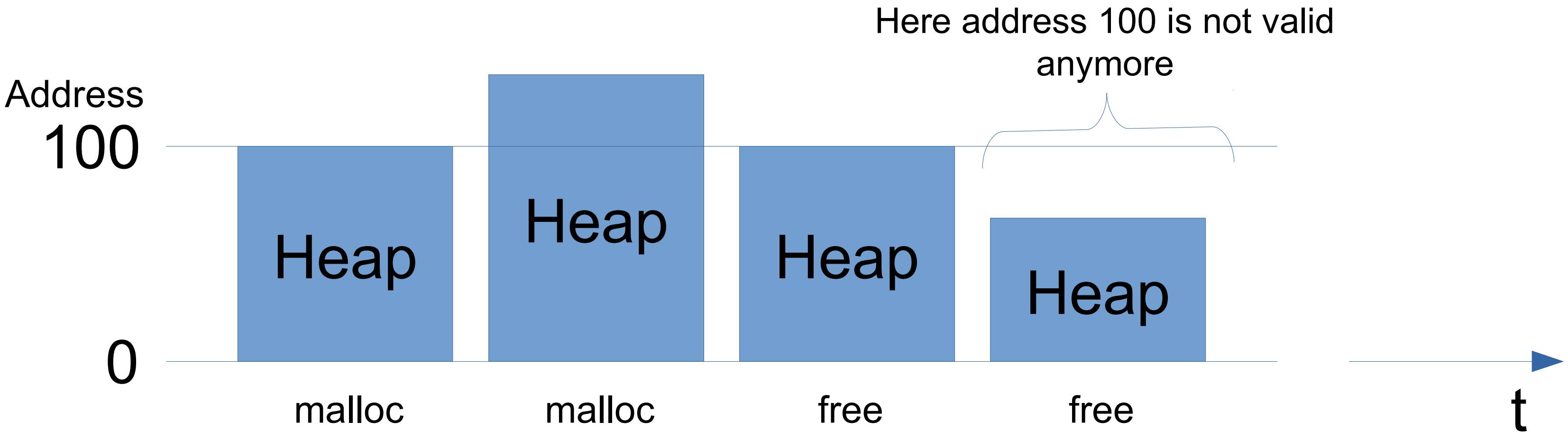
- Text segment (code)
- Data segment
  - Data (initialized variables)
  - BSS * (non-initialized variables)
  - Heap (dynamic data)
- Stack



* Block Started by Symbol

Segmentation fault

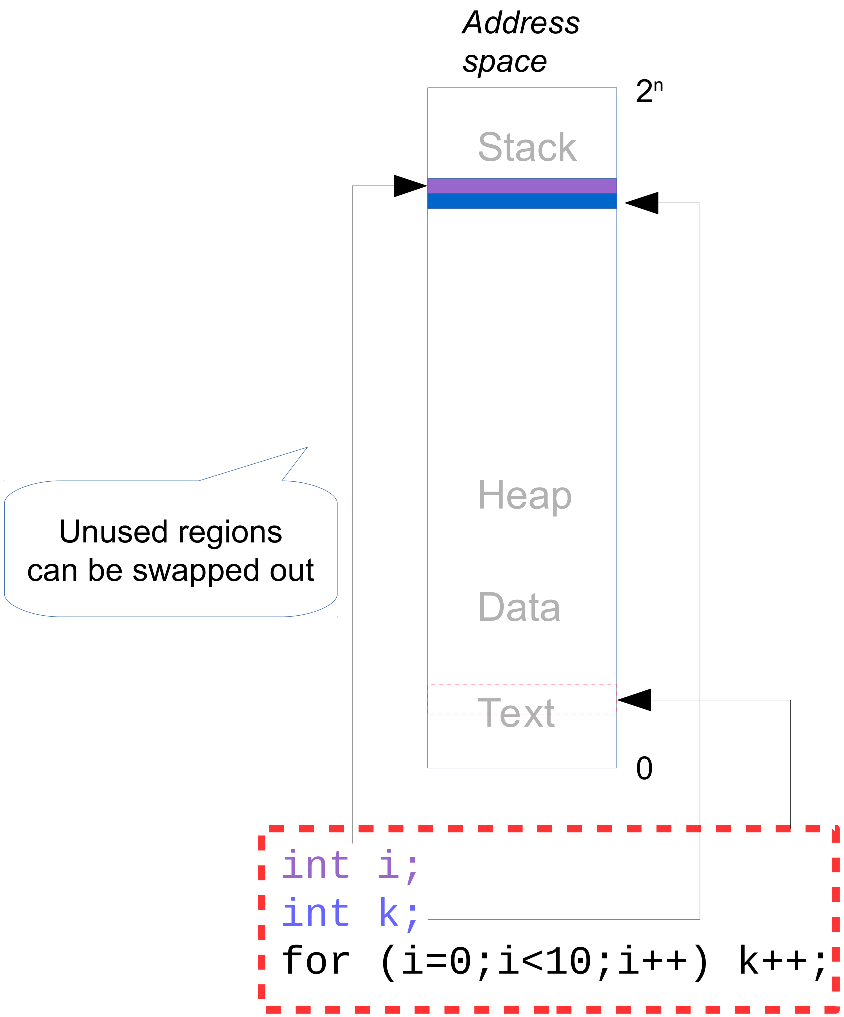
- When a process tries to access memory outside of a valid (declared) segment we get a **segmentation fault**



Pushing it further... beyond segmentation

- Computer programs might not use a whole lot of memory at the same time, maybe not even a whole segment... (→ **principle of locality**)
  - What if we break up memory in even smaller chunks that we can manage independently (even from the process they belong to), that we can protected and swap in and out as needed?

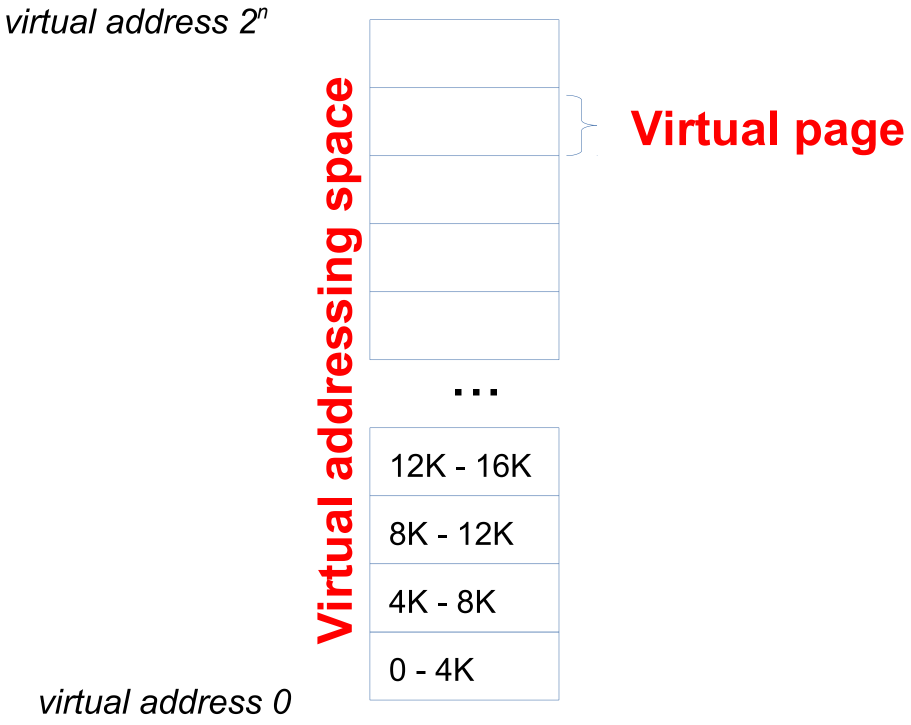
    - ...this is exactly what **memory paging** is all about!



```
int i;
int k;
for (i=0;i<10;i++) k++;
```

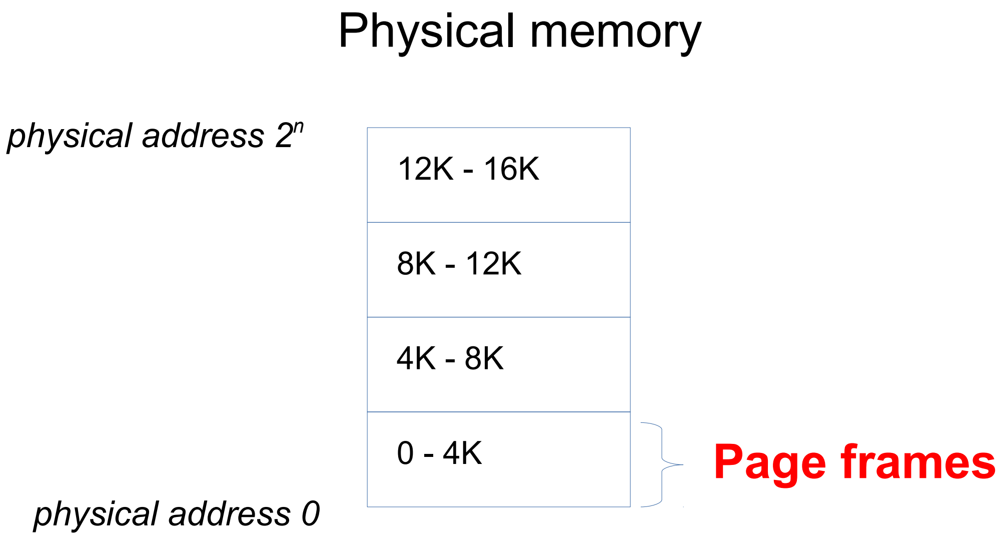
Memory paging (since Intel 80386 and on other architectures)

- Each process sees a **virtual address space**
  - the size typically corresponds to all addressable memory
    - for example, on a 32 bit architecture, the size of the virtual address space is 2^{32} bytes
- This address space is then divided into small (for example, 4kB) contiguous areas called **virtual pages** (or just pages)



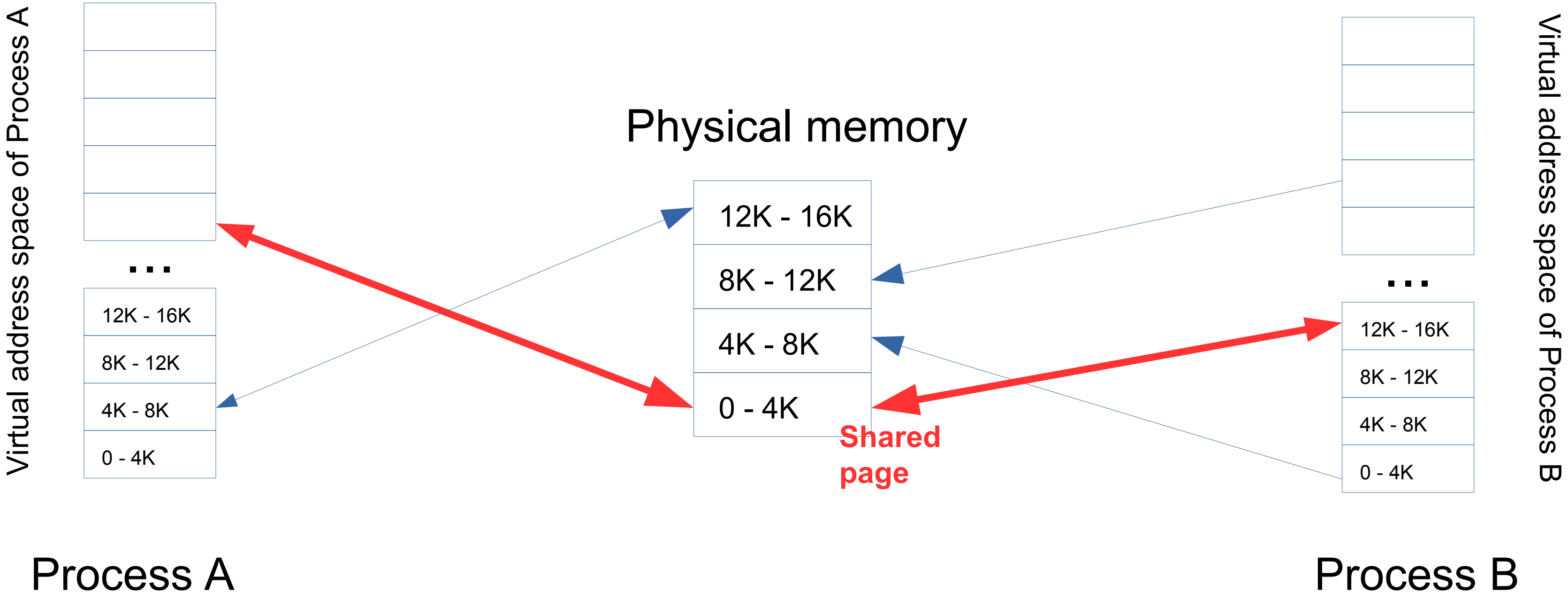
Memory paging (since Intel 80386 and on other architectures)

- Physical memory (RAM) is divided into small contiguous areas the same size as virtual pages called **page frames** (or physical pages)
  - the idea: offer **on-demand paging** (loading of pages into frames when needed)



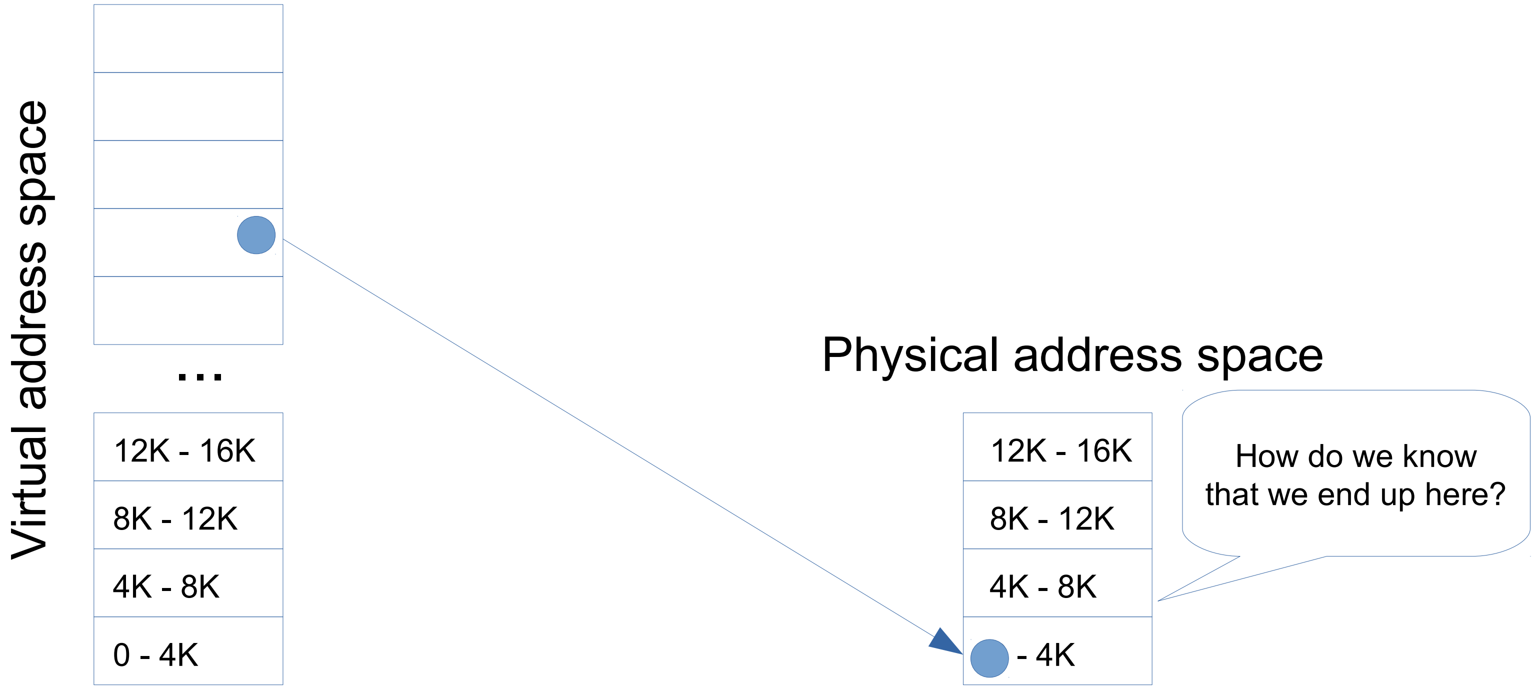
Memory paging

- In a multi-tasking system pages from different processes (thus belonging to different virtual address spaces) might be loaded in RAM
  - Pages can also be **shared between processes**
    - used for shared data (ex. DLLs)
    - might require **Copy-on-Write (COW)**



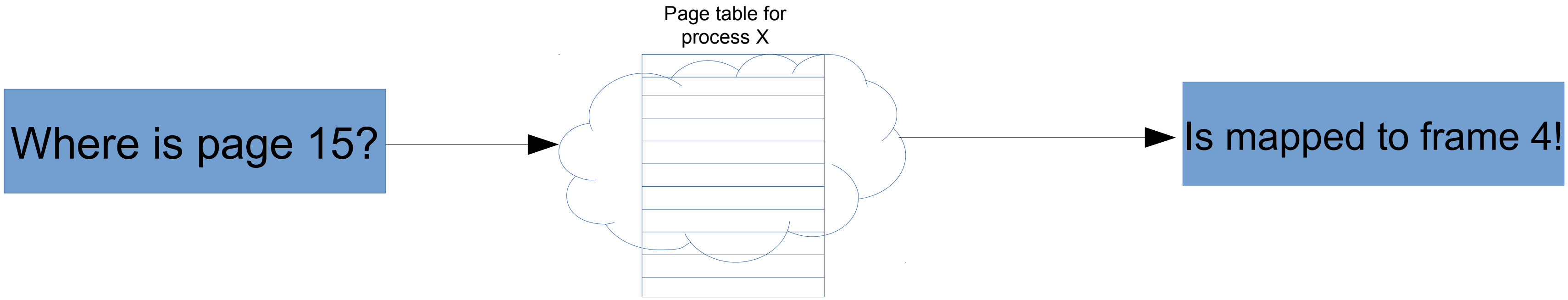
Address translation

- Because virtual addresses (used inside the program code) and physical addresses (in RAM) are different, a (transparent) **address translation** mechanism is required for paging to work



Translating addresses: the page table

- For each process, the operating systems maintains an in-memory **page table** which is used to translate virtual addresses to physical addresses, associating a **page number** to a **frame number**, namely the **physical base address of a frame**

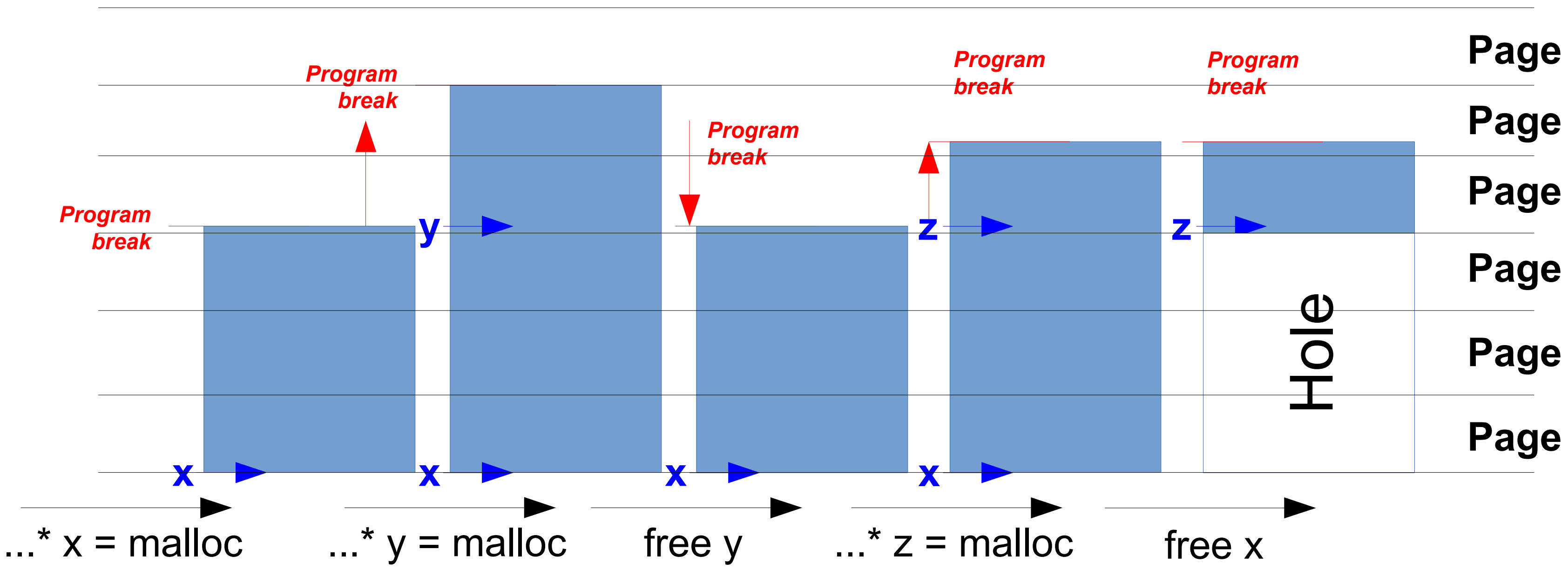


Allocating memory in a paging system: mapping a page

- When a running program allocates some memory the operating system will first check if the request fits in the existing pages, otherwise it **maps** a new page in the corresponding page table
  - **mapping** means that a new entry in the page table is declared as valid
  - subsequently a page can be assigned to a frame (if it's a new page no swap-in is necessary) and the memory becomes accessible to the process
    - when a page is not necessary anymore (i.e. we freed all memory in it) the operating system can **unmap** it → invalidates the page in the page table

Mapping a page: how does malloc work

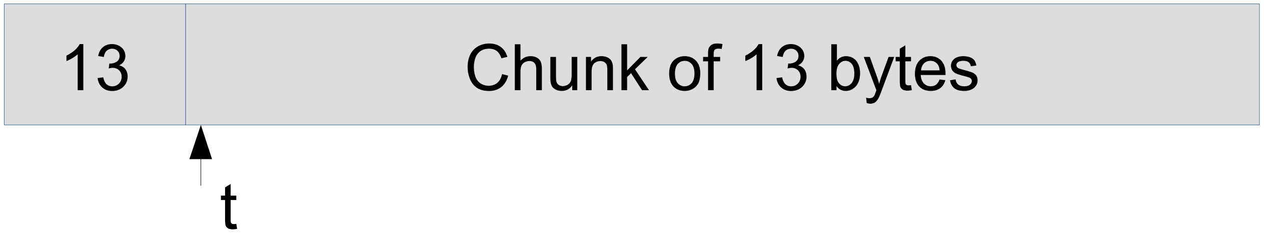
- Memory allocations (malloc/new) extend the heap
  - new pages are mapped as needed
- Deallocation (free/delete) reduces the size of the heap
  - deallocation order must be performed in reverse otherwise malloc will not free pages in the middle → **fragmentation**



Mapping a page: how does malloc work

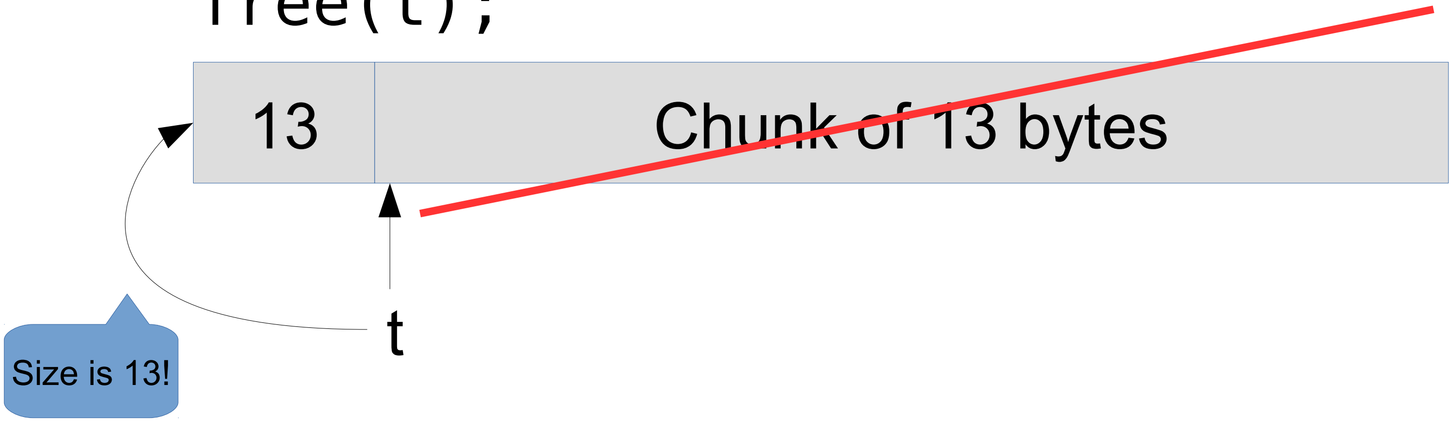
- **`malloc`** adds some additional information before the pointer which is returned to the program:

```
void* t = malloc(13);
```



- **`free`** looks behind the provided pointer to know how big is the chunk:

```
free(t);
```



Mapping a page: mmap (on Linux)

- We can allocate new memory also using **`mmap`**
  - Maps new pages into the virtual address space
  - Compared to **`malloc`** we can freely deallocate space without incurring in fragmentation

- Allocation:
  - `void* mmap(0, size, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);`

- Deallocation:
  - `munmap(void *base, size);`

Note: mmap can also map a file in memory

Sharing memory (POSIX)

- Memory can be explicitly shared between processes
- The idea is to create a shared memory object and subsequently map this area into the virtual address space of each partecipating process

- POSIX provides the following functions:
  - **`shm_open`**, to create or access a shared memory object
  - **`mmap/munmap`** to map/unmap a shared memory into the address space
  - **`shm_unlink`**, to unreference a shared memory object (and destroy it when nobody is using it anymore)

Sharing memory (POSIX)

```
// We first create a shared memory object
// int shm_open(const char *name, int oflag, mode_t mode);
int fd = shm_open("/mysharedmem", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);

// We resize it to the desired size (done by the process
// which creates the object)
ftruncate(fd, requestedSize);
// or (to get the actual size)
struct stat shostat;
fstat(fd, &shostat); // Obtain size of shared object
RequestedSize = shostat.st_size

// And then we map it somewhere in our address space
void* ptr = mmap(NULL, requestedSize, PROT_WRITE | PROT_READ, MAP_SHARED,
fd, 0);
```

We get the starting address of the
shared memory object (might be
different on different processes)

Sharing memory (POSIX)

```
// When we are done we need to unmap the object
munmap(ptr, requestedSize);

// And then unlink the object
shm_unlink("/mysharedmem");
```

Shared memory (POSIX): complete example

```c
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

// Writer process
int main(void)
{
        off_t requestedSize = 1000;
        int fd = shm_open("/mysharedmem", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
        ftruncate(fd, requestedSize);
        void* ptr = mmap(NULL, requestedSize, PROT_WRITE | PROT_READ, MAP_SHARED, fd, 0);
        sprintf((char*) ptr, "Hello world!");
        munmap(ptr, requestedSize);
        printf("Run the reader, then press enter to unlink memory object and exit\n");
        getchar();
        shm_unlink("/mysharedmem");
}
```

```c
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

// Reader process
int main(void)
{
        struct stat shostat;
        int fd = shm_open("/mysharedmem", O_RDWR, S_IRUSR | S_IWUSR);
        fstat(fd, &shostat); // Obtain size of shared object
        void* ptr = mmap(NULL, shostat.st_size, PROT_WRITE | PROT_READ, MAP_SHARED, fd, 0);
        printf("Got '%s' in shared object of size %d\n", (char*) ptr, (int) shostat.st_size);
        munmap(ptr, shostat.st_size);
        shm_unlink("/mysharedmem");
}
```

Note: compile with -lrt

Listing shared memory objects (Linux)

```
user@host:~$ ls -l /dev/shm
lrwxrwxrwx 1 root root 8 nov  1  2013 /dev/shm -> /run/shm
user@host:~$ ls -l /run/shm/
total 84
-rwx------ 1 user user 67108904 nov 27 11:29 pulse-shm-1719862516
-rwx------ 1 user user 67108904 nov  1  2013 pulse-shm-2110702174
-rwx------ 1 user user 67108904 nov 26 10:00 pulse-shm-4070185405
-rwx------ 1 user user 67108904 nov  1  2013 pulse-shm-439891043
-rwx------ 1 user user 67108904 nov 26 15:57 pulse-shm-903705885
```

Synchronizing concurrent access to shared memory

- When using shared memory we probably need a synchronization mechanism
  - POSIX semaphores can be made to work across different processes to synchronize access to shared memory: we can use **POSIX named semaphores**
  - Just create a named semaphore using **`sem_open`**:

Initial value

```
sem_t* sem_id;
sem_id = sem_open("mysemaphore", O_CREAT, 0600, 0);
sem_close(sem_id);
sem_unlink("mysemaphore"); // destroys semaphore when
                              everybody else closes it
```

  - “Traditional” semaphore primitives (**`sem_post`**, **`sem_wait`**) works as usual

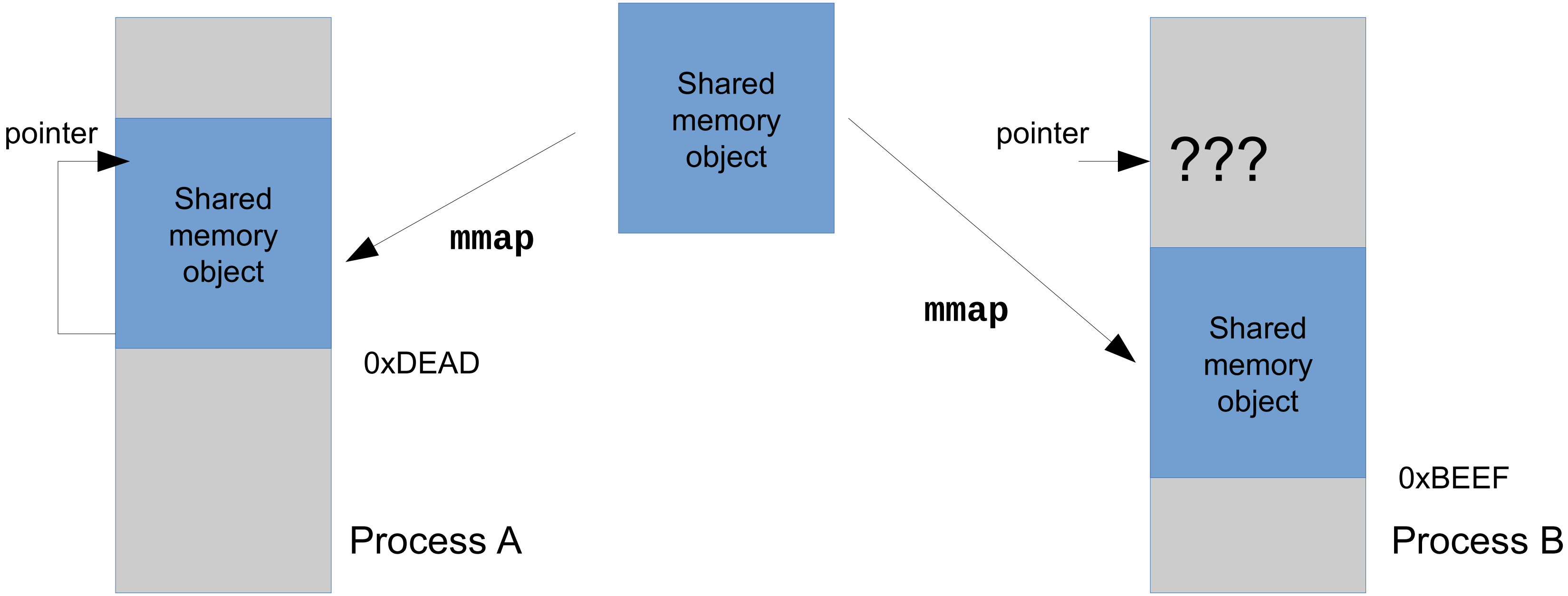
Shared memory (POSIX): complete example with semaphores

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <semaphore.h>

// Writer process
int main(void)
{
        sem_t *r_done, *w_done;
    r_done = sem_open("reader_done", O_CREAT, 0600, 0);
    w_done = sem_open("writer_done", O_CREAT, 0600, 0);
        off_t requestedSize = 1000;
        int fd = shm_open("/mysharedmem", O_CREAT | O_RDWR, S_IRUSR |
S_IWUSR);
        ftruncate(fd, requestedSize);
        void* ptr = mmap(NULL, requestedSize, PROT_WRITE | PROT_READ,
MAP_SHARED, fd, 0);
        sprintf((char*) ptr, "Hello world!");
        munmap(ptr, requestedSize);
        printf("Waking up reader...\n");
        sem_post(w_done);
        printf("Waiting for reader to read stuff...\n");
        sem_wait(r_done);
        shm_unlink("/mysharedmem");
        sem_close(r_done);
        sem_close(w_done);
        sem_unlink("reader_done");
        sem_unlink("writer_done");
}
```

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <semaphore.h>

// Reader process
int main(void)
{
        sem_t *r_done, *w_done;
    r_done = sem_open("reader_done", O_CREAT, 0600, 0);
    w_done = sem_open("writer_done", O_CREAT, 0600, 0);
    printf("Waiting for writer to finish writing...\n");
    sem_wait(w_done);
        struct stat shostat;
        int fd = shm_open("/mysharedmem", O_RDWR, S_IRUSR | S_IWUSR);
        fstat(fd, &shostat); // Obtain size of shared object
        void* ptr = mmap(NULL, shostat.st_size, PROT_WRITE | PROT_READ,
MAP_SHARED, fd, 0);
        printf("Got '%s' in shared object of size %d\n", (char*)
ptr, (int) shostat.st_size);
        printf("Waking up writer...\n");
        sem_post(r_done);
        munmap(ptr, shostat.st_size);
        shm_unlink("/mysharedmem");
        sem_close(r_done);
        sem_close(w_done);
        sem_unlink("reader_done");
        sem_unlink("writer_done");
}
```

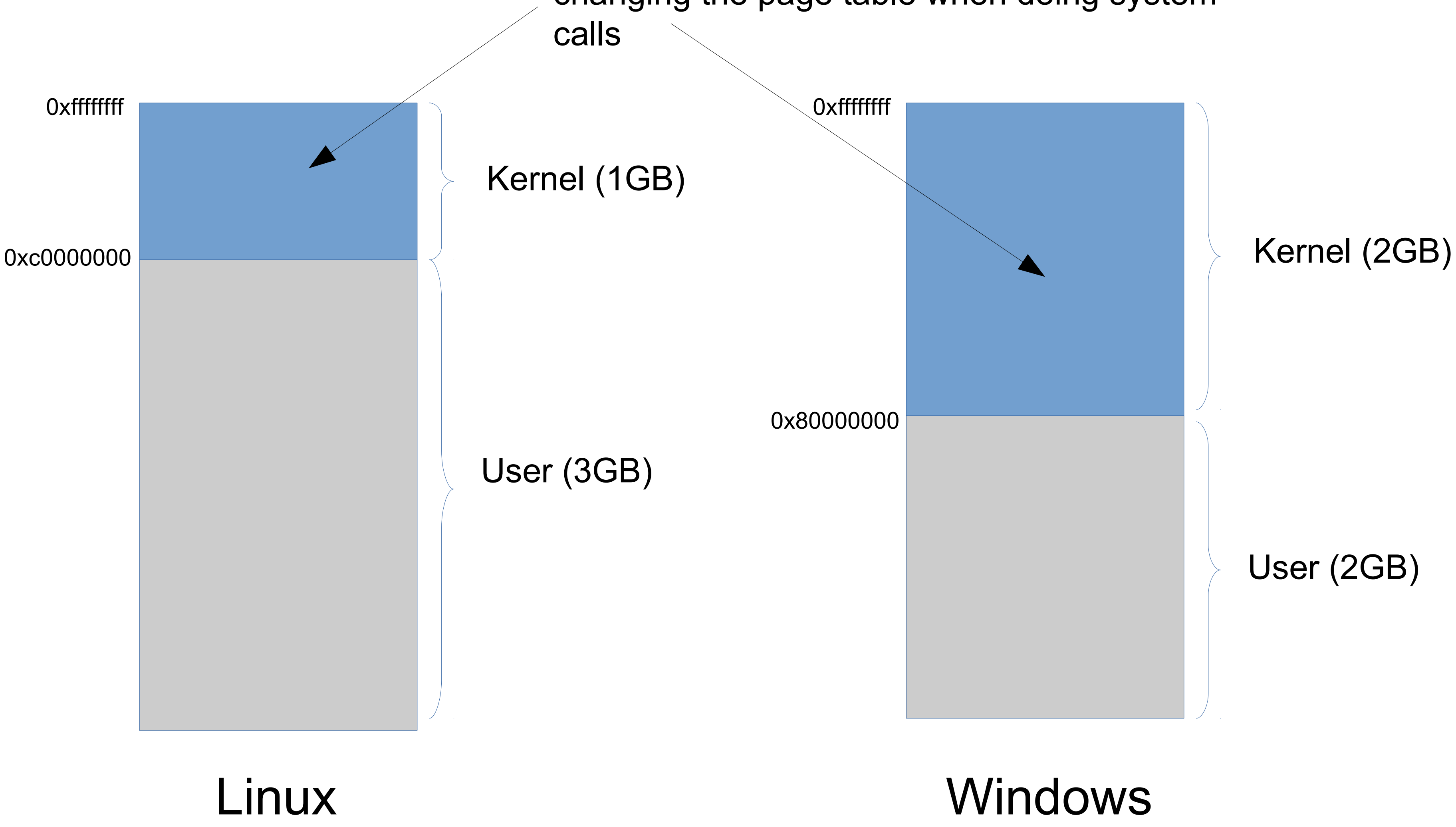
One more thing: pointers and shared memory

- When working with data structures stored in shared memory do not rely on pointers, since the object is likely to be mapped at different virtual addresses:
  - use **offsets** from the beginning of the structure
  - If the data structure is a linked-list employ an array of nodes an use **indices** to refer to each node



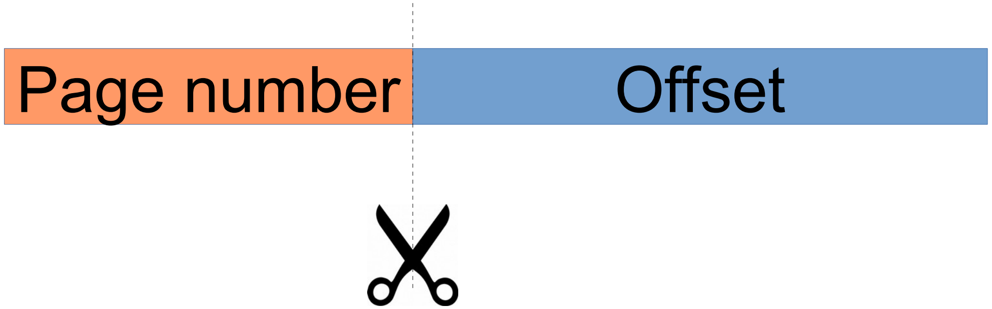
Memory layout (32 bit)

This area is made of pages which are accessible only by the kernel → we can avoid changing the page table when doing system calls

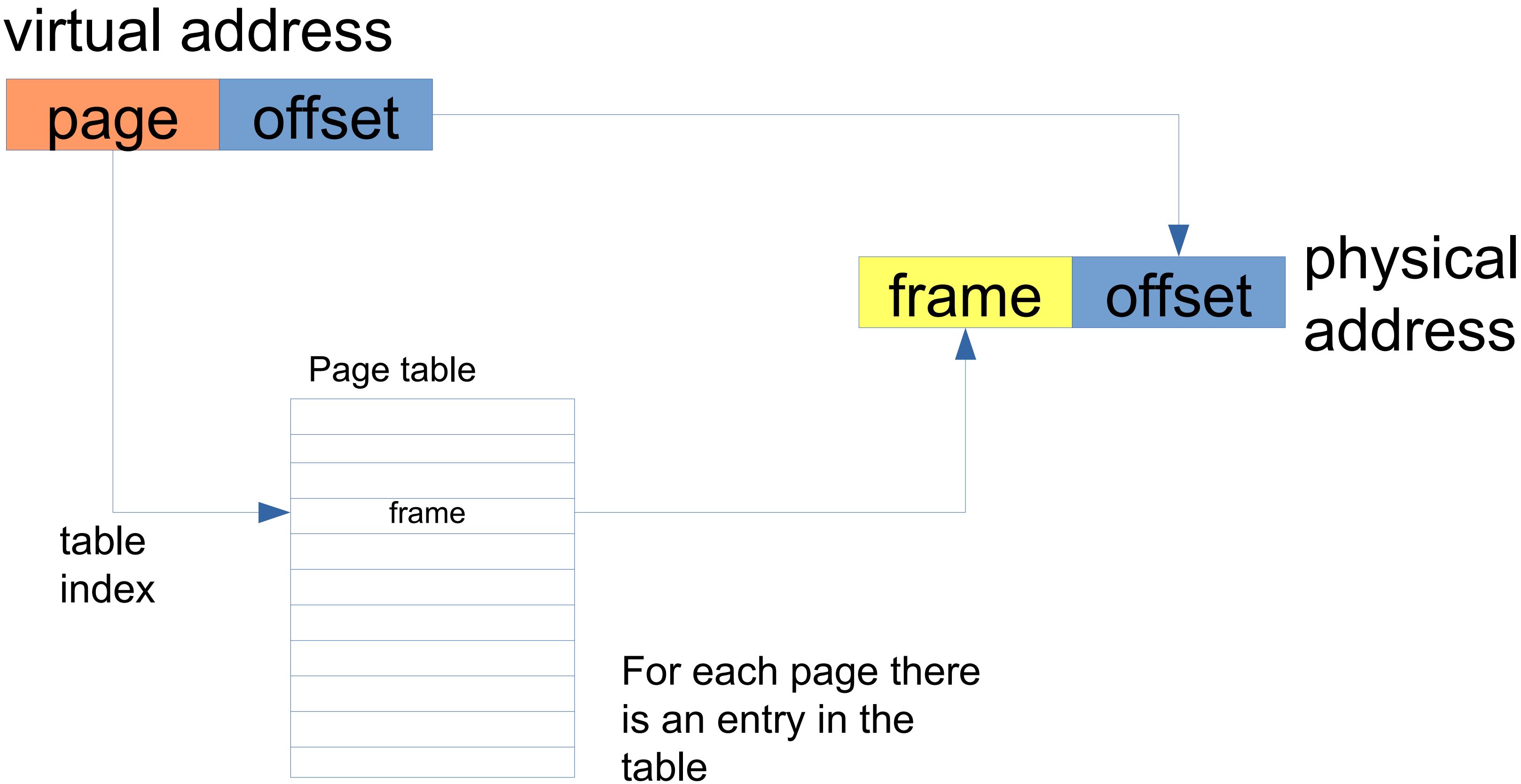


Back to address translation

- When a process does a memory access (using a virtual address) we need answer a simple question “**to which page does it belong to?**”
  - Since pages are all the same size, and the size is a power of 2, we can easily determine the page number by splitting the binary value of the virtual address in two
  - This gives us two fields: the **most significant part** determines **page number** while the **least significant part** determines the **offset** within the page / frame



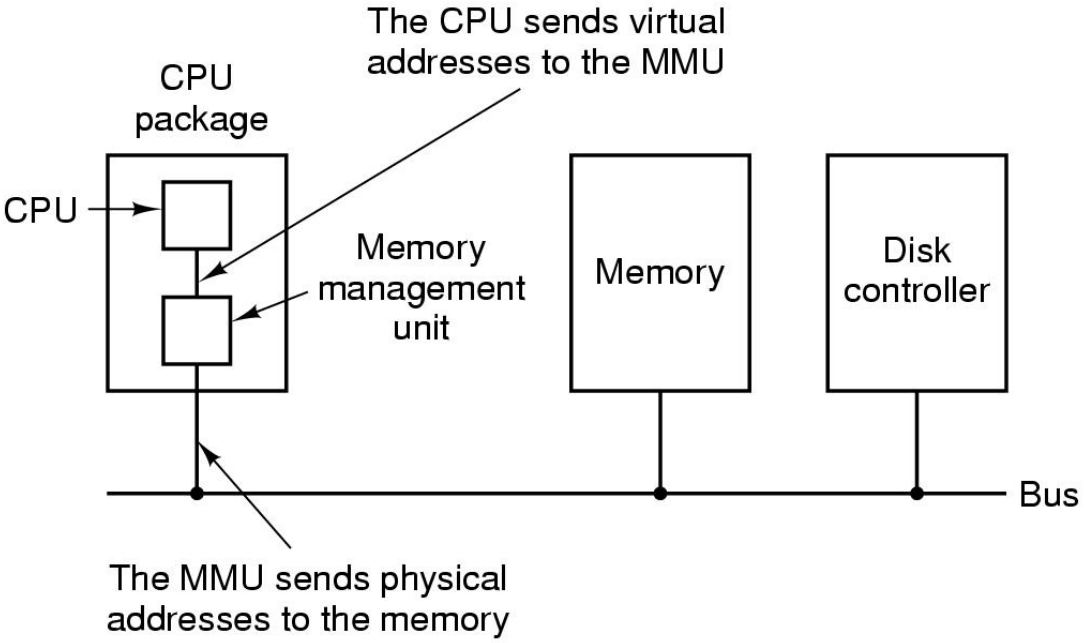
Address translation



For each page there is an entry in the table

Address translation

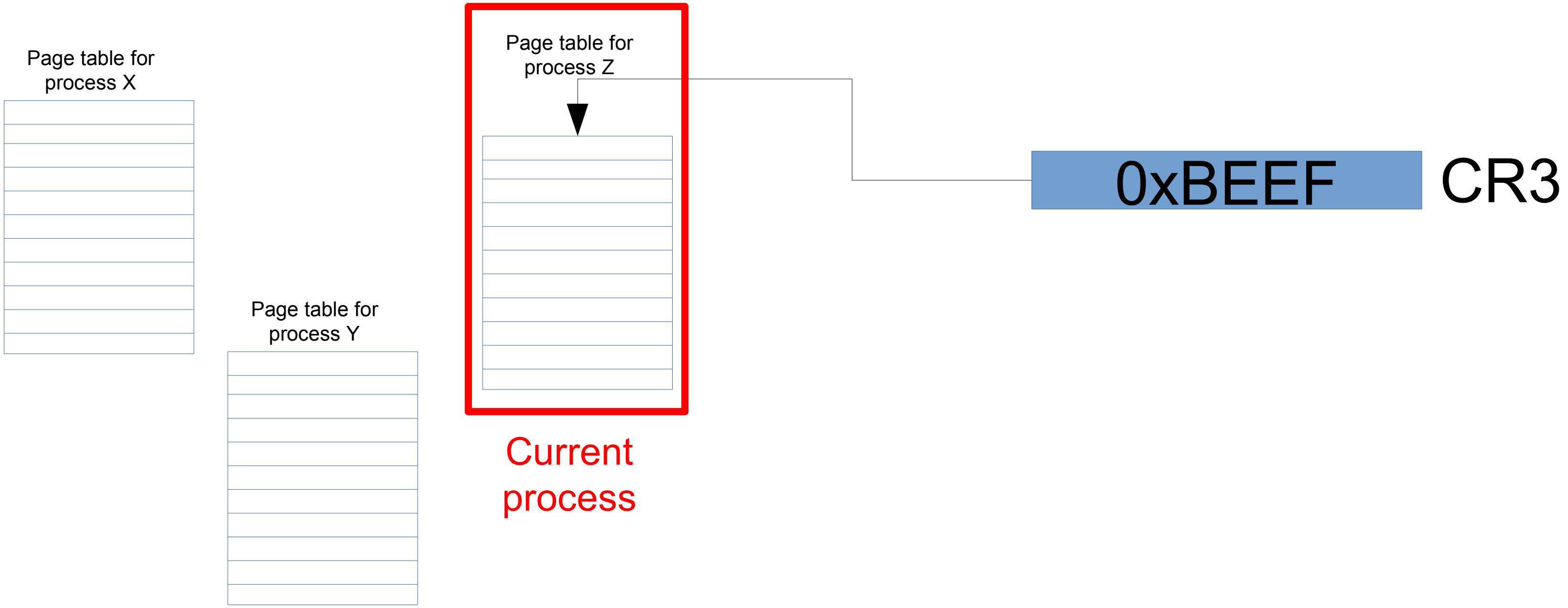
- Address translation is done by the hardware (for efficiency), namely by the **MMU** (Memory Management Unit)


The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

from A. Tanenbaum, Modern Operating Systems, 2nd Ed.

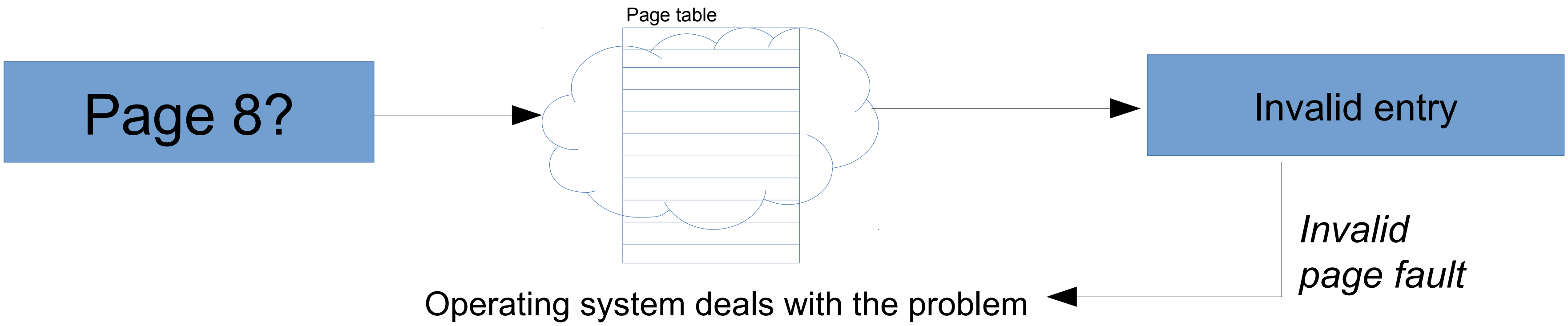
How does the hardware know where the table is?

- The operating systems tells the MMU where the table is located in memory by updating a CPU register (CR3 on x86)
  - This register is updated at each context switch, because each process has its own page table



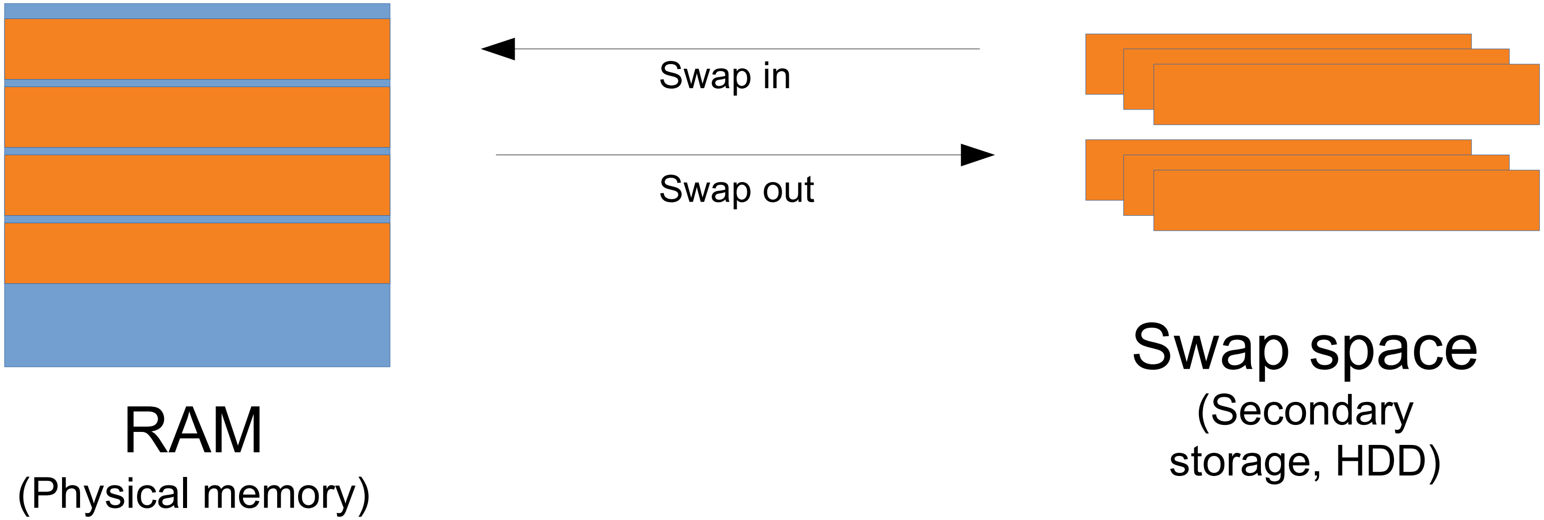
What if we did not map a page?

- The MMU runs an algorithm called *page table walker* which scans the page table to look for a valid page translation entry:
  - if the entry indicates is valid translation can happen
  - If the page is not valid, the CPU generates an ***invalid page fault*** (exception captured by the operating system)



What if we cannot fit all pages into memory?

- When RAM fills up, **swapping** can used to move (less used) pages to a secondary storage (**swap out**)
  - When the information on a specific page is requested again a page on the secondary storage needs to be brought to RAM again (**swap in**)

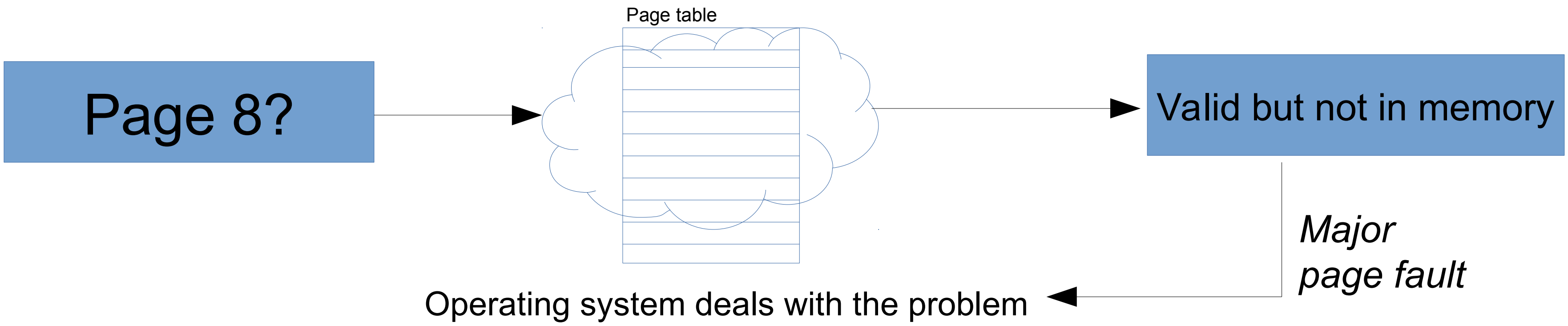


Where do we store swapped out pages?

- Windows uses **a file on disk** (pagefile.sys) called paging file
- Linux typically uses **a dedicated partition** (although it is possible to swap on a file too)
  - **`mkswap`** (command to create a swap space)
  - **`swapon, swapoff`** (commands to activate/deactivate a swap area)

Major page fault

- If the *page table walker algorithm* encounters a page table entry which is valid but refers to a swapped out page the CPU generates a **major page fault** exception (which is captured by the operating system)



Page 8?

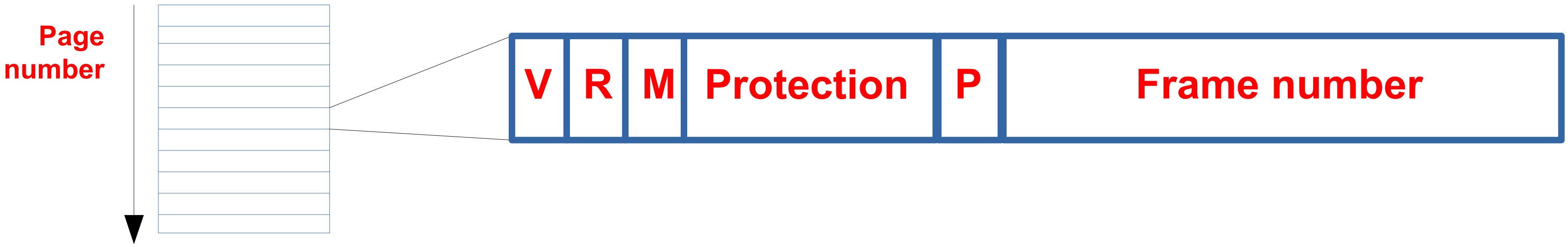
Page table

Valid but not in memory

Major page fault

Operating system deals with the problem

How does the CPU know if an entry is valid / if a page is swapped?



- The structure of the page table (which depends on the architecture of the system) allows the MMU to know how to deal with swapped out pages:
  - The entry for each page contains
    - ***valid/invalid bit (V)***: indicates if the entry is valid or not (pages that are not mapped cannot be used)
    - the **frame number**: tells where the page is located in RAM
    - a **present** bit (**P**): tells if the page is mapped to a frame in RAM
    - a **modified/dirty** bit (**M**): set to 1 when the page has been modified
    - a **referenced** bit (**R**): set to 1 when the page has been referenced, cleared by the OS
    - Protection flags:
      - access **permissions**: allows to set read/write permissions on the contents of a page
      - access **privileges**: user / supervisor (kernel) → used to protect kernel mapped pages

Types of page faults

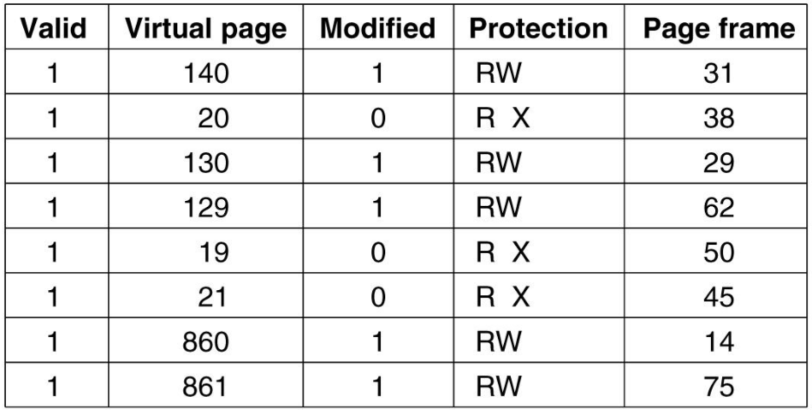
- **Invalid page fault**: in the page table there is no valid entry to translate the virtual address (this typically means that the user did not allocate that memory or has already de-allocated it, and usually generates a *segmentation fault* which terminates the process)

- **Major page fault**: there is an entry in the page table, but the requested page is not in the main memory (and might need to be swapped in)

- **Minor page fault**: the page is already in memory but the P (present) bit is not set (for example if the page is shared by multiple processes and one of them already loaded the contents of the page in RAM)

Speeding-up address translation: TLB

- Address translation depends on the page table
  - So to access a memory location we require one or more additional memory access operations!
- To speed up the translation mechanism for frequently used memory locations the hardware can implement a small cache containing a subset of the page table:
  - **TLB (Translation Lookaside Buffer)**
- If the TLB can translate the address we have a **TLB Hit**, otherwise a **TLB Miss** and we must search the whole page table
- Since cached entries are tied to each process, the operating system must clear the buffer on a context-switch (**TLB flush**)

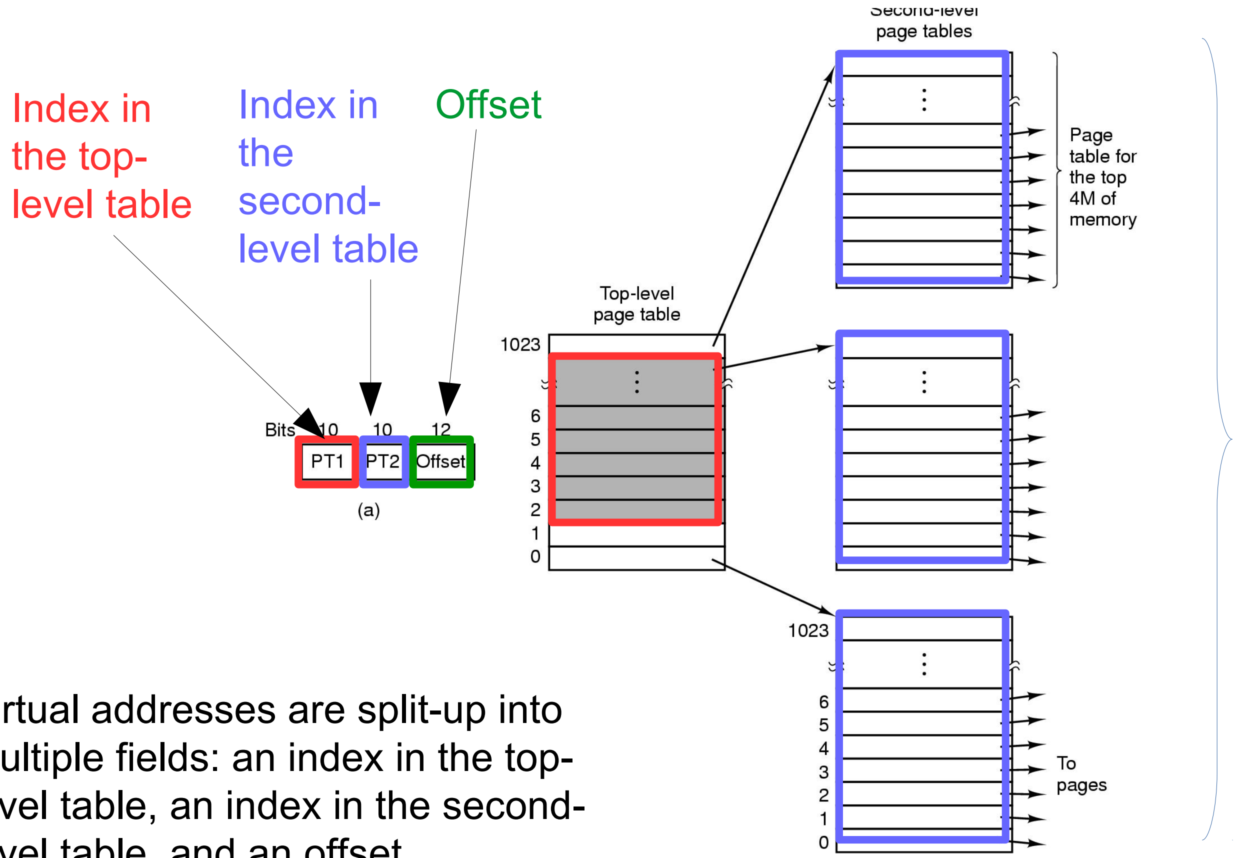


Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

from A. Tanenbaum, Modern Operating Systems, 3rd Ed.

Avoiding large page tables: multi-level page table

- Large virtual address spaces require large page tables: to avoid wasting too much space, we can use **multi-level page tables**



Index in the top-level table

Index in the second-level table

Offset

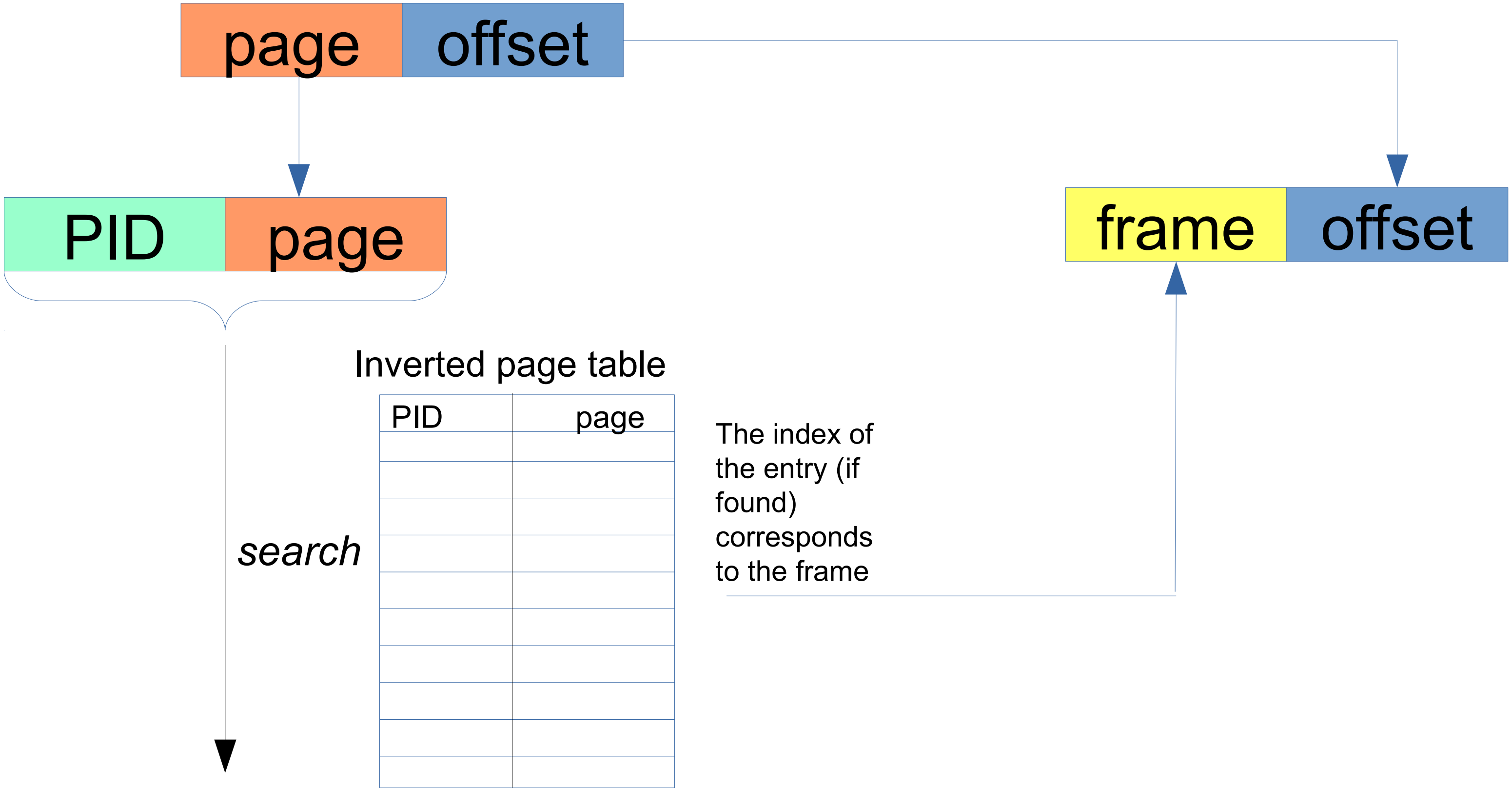
Second-level page tables are allocated only if they contain at least one entry

Virtual addresses are split-up into multiple fields: an index in the top-level table, an index in the second-level table, and an offset

from A. Tanenbaum, Modern Operating Systems, 2nd Ed.

Avoiding large page tables: inverted page tables

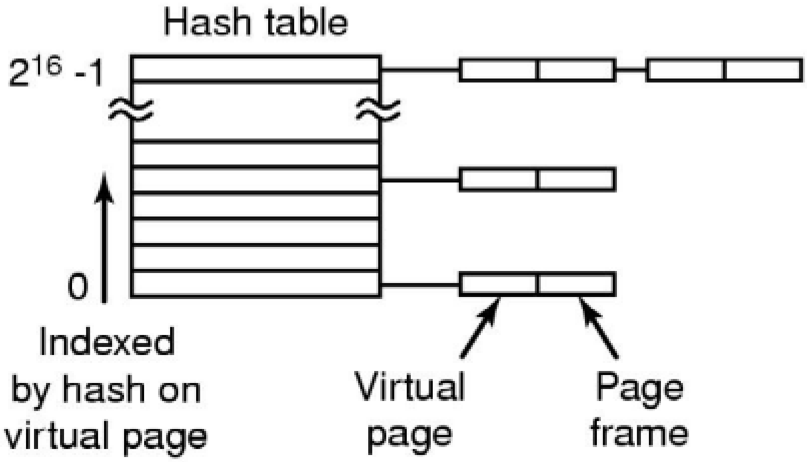
- Instead of an entry for each page we maintain a table with an entry for each frame: we also need the **PID** to determine if the translation applies to the correct process



The index of the entry (if found) corresponds to the frame

Avoiding large page tables: hash tables

- Instead of an entry for each page we can maintain a smaller table to map virtual pages to frames
  - we **hash the virtual page** number to obtain an index in the hash table and the corresponding translation entry
  - we can use the same table for multiple processes by adding a PID field on each translation entry



from A. Tanenbaum, Modern Operating Systems, 2nd Ed.