Memory management

It is possible to assign a separate memory chip to each process in the system, if the number of processes is small enough. This is not a cost–effective way to spend money: as the number of processes varies in time, we would have lots of unused memory most of the time.

Another way to manage main memory is to allow user processes to share it. Sharing memory has a different meaning than sharing the CPU:

- the CPU cannot be divided into smaller pieces and has to be given to a process as one unit (sharing is solely temporal).
- Memory can be subdivided into smaller units (**"blocks"**) and each block can be given to a process for exclusive use (this is spatial sharing). The same block can later be given to another process so memory sharing is both spatial and temporal.

All the memory management schemes used in contemporary OS are based on the same idea: divide the memory into a large number of blocks and give these blocks to processes when they need them.

There is one problem that needs to be solved: when a user prepares a program for execution, he/she does not know which block of real memory will be given to this program (when it becomes a process). Moreover, the same program will be assigned different memory blocks each time it is executed.

Thus even the simplest memory management scheme must provide some form of mapping of addresses as seen by a process into addresses in the actual main memory. The two kinds of addresses being different, they are given different names:

Physical address: this is an address in real memory.

Logical address: this is an address used by a process.

Physical address

The computer system has some real main memory attached to a **memory controller** (**MMU**). This memory is called physical memory because it physically exists (it can be touched).

Physical memory is made of identical, small, indivisible units that can be accessed independently of other units. These units are identical, so a label is associated with each of them to distinguish it from other (identical) units. This label is called a **physical address**.

Main memory is accessed through a memory bus. The MMU can be placed either between the bus and the memory or between the bus the CPU (in which case a DMA controller sits between memory and the bus).

MMU has another meaning in the Fortran environment: Malfunction Management Unit.

Physical address space

Physical addresses form a set called **physical address space**. We normally describe the physical address space in terms of the range of legal physical addresses. Traditionally, this space contained all the numbers in the set $\{0..2^{32}-1\}$ (in the early days other options were tried but were abandoned). Recently there is a move towards physical address spaces of size 2^{64}.

Logical address

Physical memory is the memory that is seen by the MMU; it happens to corresponds to the real memory as installed in a computer.

The CPU is not forced to see memory in the same way. Various schemes were designed to make the CPU imagine how memory looks like, with the CPU's imagined memory not necessarily looking the same way as physical memory.

There are several reasons for making the distinction; the main (and sufficient for this course) is that the OS must give the processes a view of memory that differs from the MMU's view.

The memory that the CPU thinks it sees is called logical memory. Like physical memory it is made of addressable small units; the address of each such unit is called a logical address.

The set of all the logical addresses is called a **logical address space**.

It is important to note that in all contemporary computers a logical address is structured differently than a physical address.

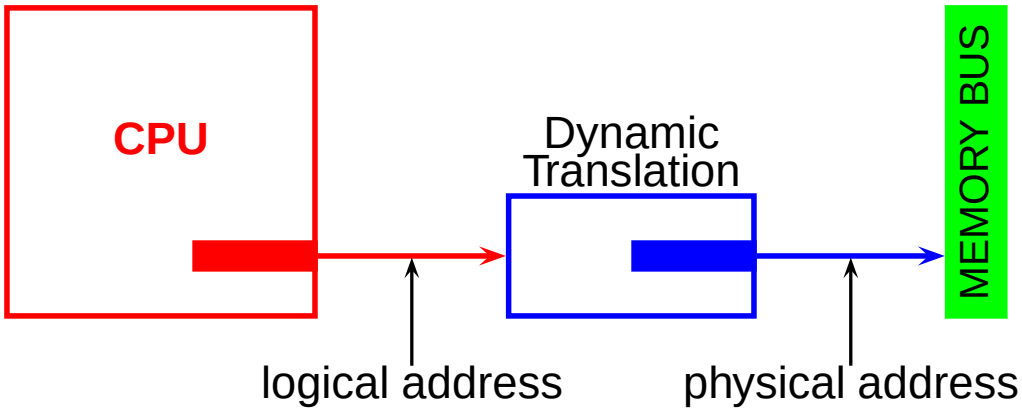
While a physical memory is made of a contiguous sequence of units numbered 0 and up (like a one–dimensional array), logical memory typically is seen as a one– or two–dimensional array of pointers to one–dimensional arrays.

It only looks complicated, but is no more difficult to grasp than:

```
char **LAS[] ;
```

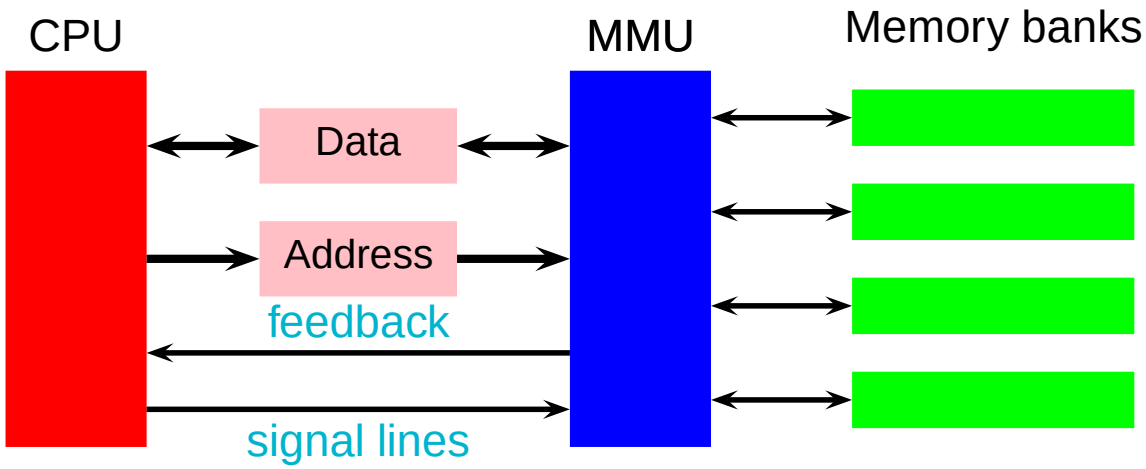
The CPU's vision of memory could be mapped on the physical memory in a static way (not changing in time).

All computer manufacturers chose a dynamic mapping of a logical address space into a physical address space. This is done by translating every logical address as it comes out of the CPU and passing the translated address (physical) to the physical memory.

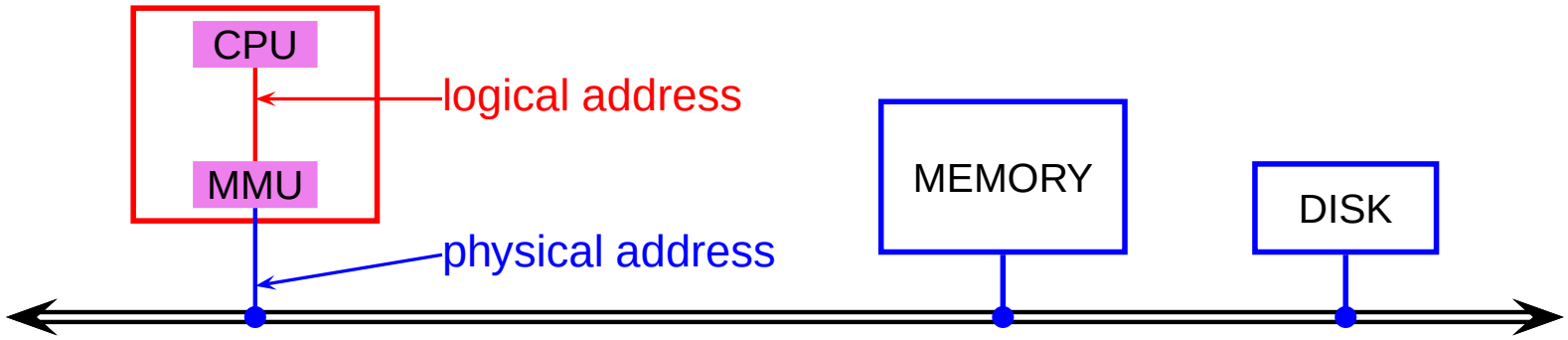


MMU

The device performing dynamic address translation acts as the memory controller and is responsible for detecting addressing errors; hence the need for a feedback line to interrupt the CPU when needed. **MMU**.



In many architectures all the devices use physical addresses and only the CPU uses logical addresses. Note that this arrangement makes it impossible for a process to issue an I/O request directly.



A clever trick allows to disable translation: make the MMU replace logical addresses with identical copies as physical addresses.

Prehistoric approach

In early OS memory management, each process was given a chunk of physical memory. To make this scheme work, the CPU was equipped with two extra registers: the **base** register and the **limit** register.

Suppose that a process was given an area of physical memory of size $\mathcal{L}$ starting at physical address $\mathcal{B}$.

When the OS gave the CPU to the process, it first loaded the two registers:

base = $\mathcal{B}$

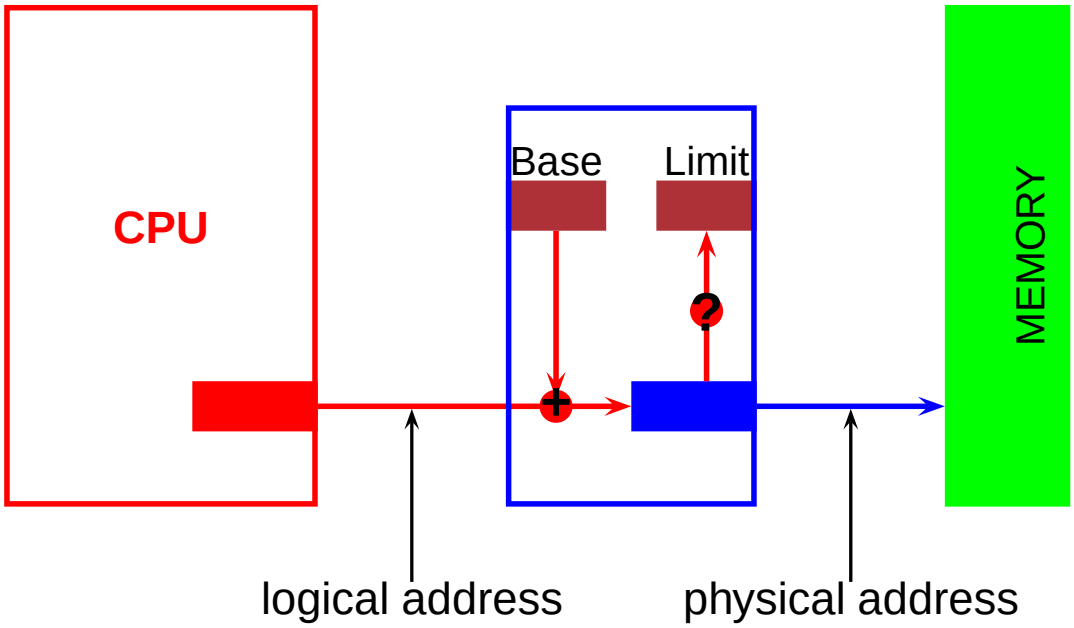
limit = $\mathcal{B} + \mathcal{L}$

Whenever the CPU wants to access (logical) address A, the following code was executed (in hardware circuitry) to obtain the corresponding physical address P:

```
P = A + base ;
if( !(P < limit) ) interrupt ;
```

Relocation

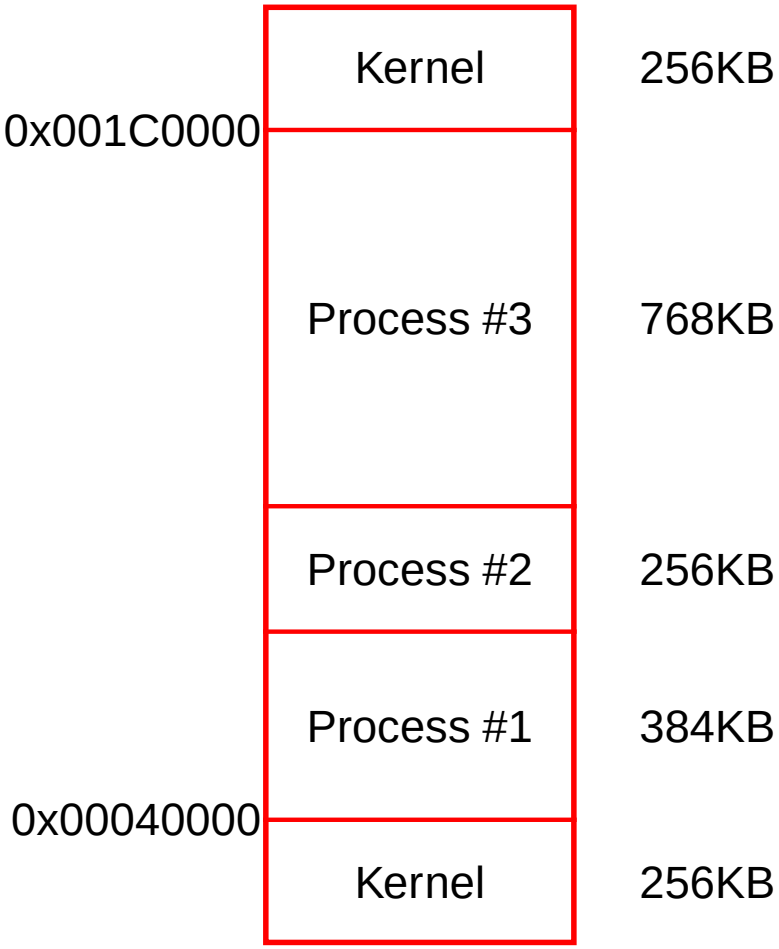
The approach was called dynamic relocation: each address was translated at runtime from logical to physical.



If the comparison with the **limit** register was unsuccessful, a **memory protection violation** was assumed (address out of bounds).

Relocation and now what?

If dynamic relocation is available, the OS could keep several processes in main memory simultaneously by giving each of them a block:



We now have a 2MB memory holding simultaneously 3 user processes, potentially giving the scheduler 3 to choose from (if all 3 are ready).

Partitioning

With dynamic relocation, main memory is divided into memory blocks called **partitions** (they could be fixed or variable). When a user submits a program for execution, the memory needs of this program are deduced (usually with user input). The act of creating a process includes finding for it a partition of suitable size ("not too big, not too small").

Usually, a process in execution is permanently bound to one and the same partition. There is a possibility of moving an existing process from partition to partition (**swapping**) but this is a time–consuming operation.

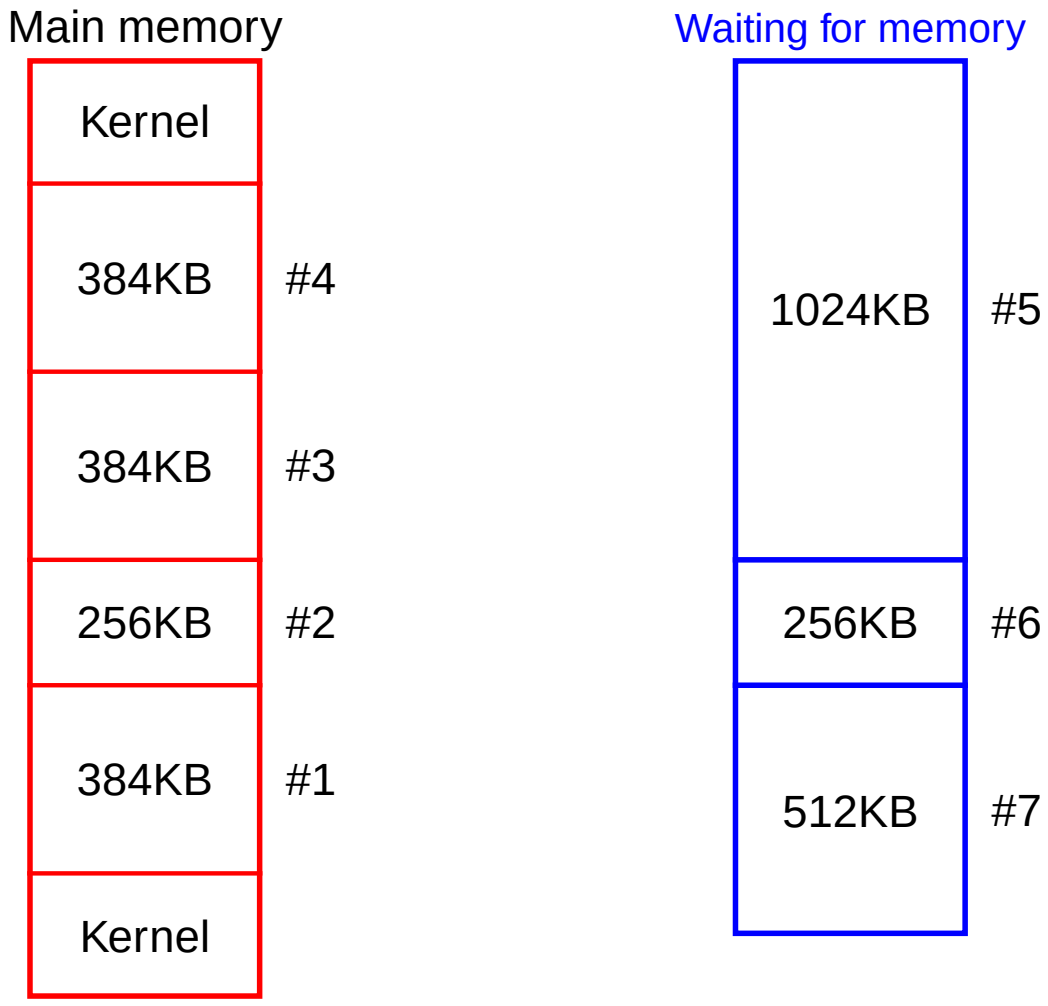
Complications

Partitioning did not work well for many reasons:

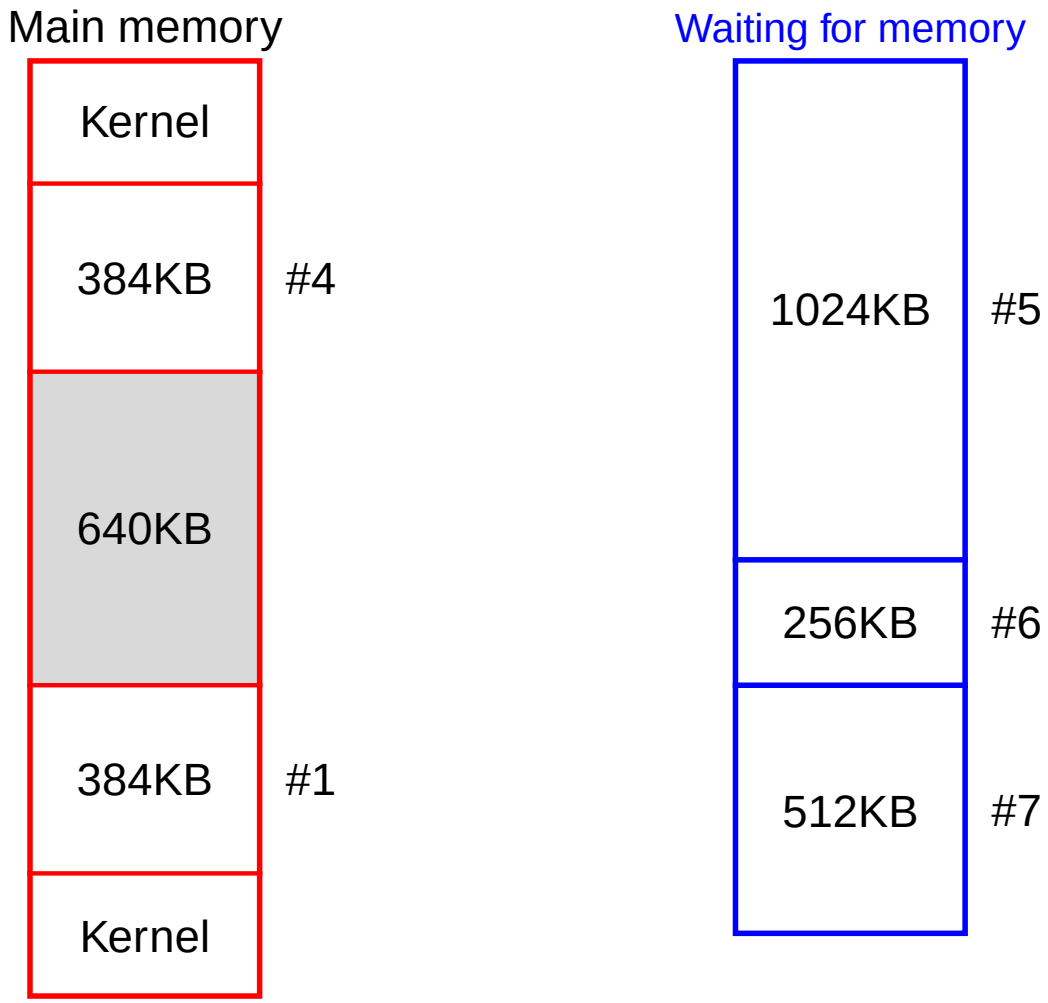
- Every process has a different size. when one process terminated, it was not always possible to replace it in memory (long story of fragmentation).
- Some programs (and some programming languages) expected the size of the address space of a process to change dynamically (stack, heap, many other reasons). Few programmers can predict in advance what these changes would be.
- One big I/O–bound process can bring the system (almost) to a stop.

And there was the additional (seemingly unsolvable) problem of the process that was too big to fit in the whole main memory (how to fit a 3GB address space into a 2GB memory?).

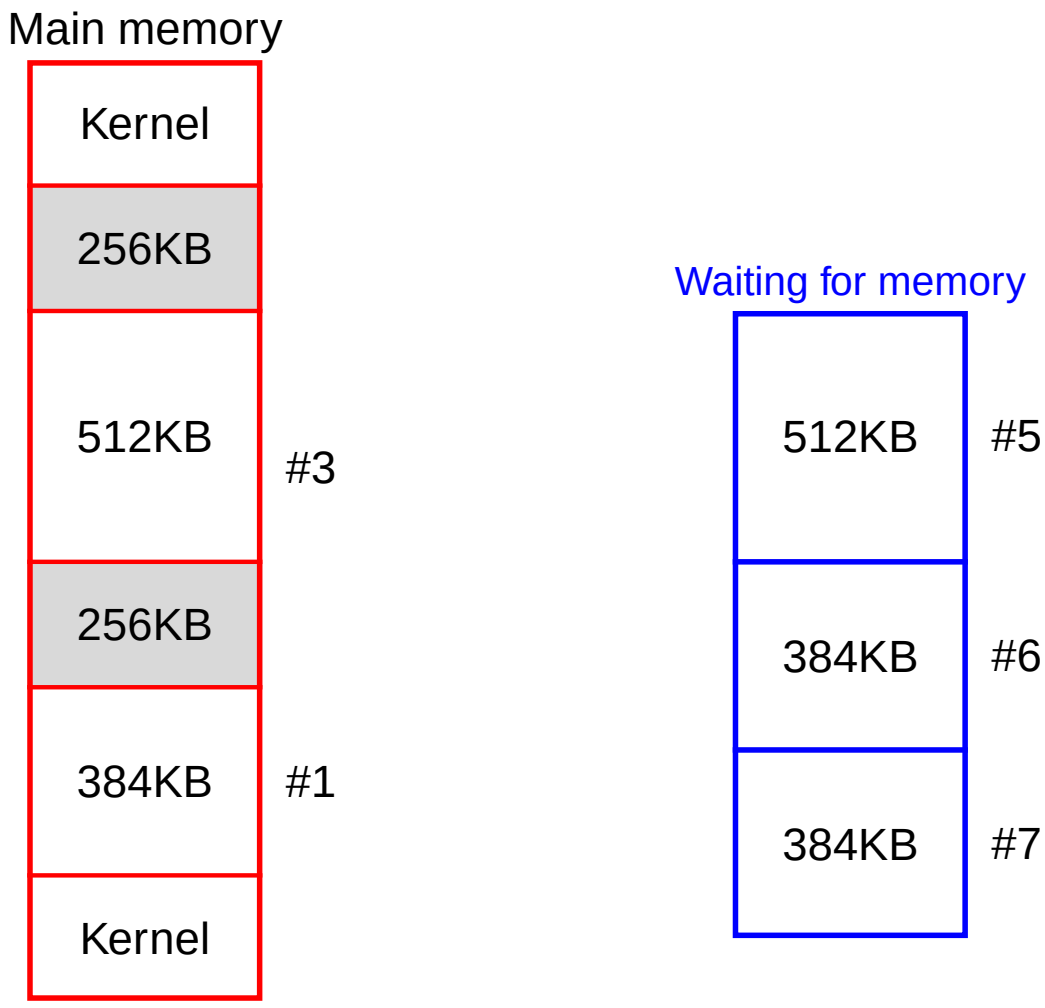
Main memory fragmentation



Main memory fragmentation



Main memory fragmentation



Fragmented logical address space

All the problems were magically solved by the development of **virtual memory**. This concept was first introduced in 1958 (UK) and was gradually refined into its today's form called **paged segmentation** which is a combination of two originally competing approaches:

Paging: divide the address space of a process into fixed–size blocks just like a telephone book is divided into pages. The division is purely mechanical, akin to the operation of slicing bread by a bread slicer.

Segmentation: divide the address space of a process into logically–connected subunits. Analogous to dividing a report into sections or a book into chapters.

Combining the two methods is similar to creating a book: a sequence of words is divided into (presumably) coherent chapters of variable length (each chapter bound by some common thread), which are in turn printed on pages of fixed size.

Fragmented logical address space

All the problems were magically solved by the development of **virtual memory**. This concept was first introduced in 1958 (UK) and was gradually refined into its today's form called **paged segmentation** which is a combination of two originally competing approaches:

Paging: divide the address space of a process into fixed–size blocks just like a telephone book is divided into pages. The division is purely mechanical, akin to the operation of slicing bread by a bread slicer.

Segmentation: divide the address space of a process into logically–connected subunits. Analogous to dividing a report into sections or a book into chapters.

Combining the two methods is similar to creating a book: a sequence of words is divided into (presumably) coherent chapters of variable length (each chapter bound by some common thread), which are in turn printed on pages of fixed size.

With or without VM

Paging and segmentation are not directly related to virtual memory: they can be implemented in a system without virtual memory.

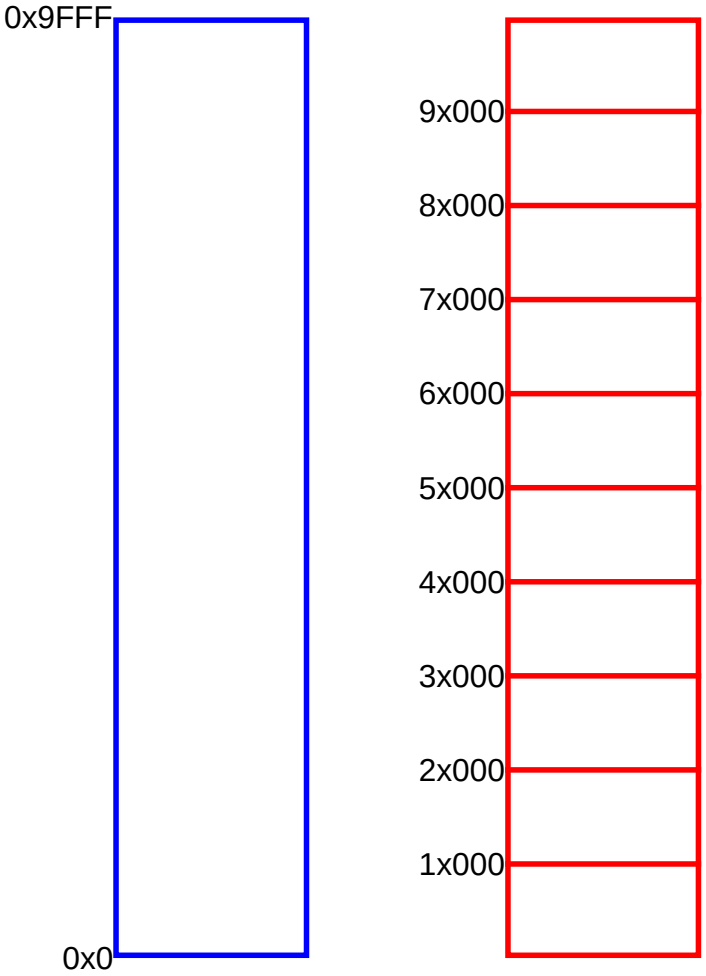
Paging: practically solves two major problems: fragmentation of main memory and expanding address spaces.

Segmentation: reduces the impact of memory fragmentation but does not address the need to expand an address space. It does solve another big problem: library sharing (this problem is not directly related to memory management but is of great importance and eventually made virtual memory necessary).

But none of them solves the other problem: several large logical address spaces cannot fit simultaneously in the main memory. Virtual memory is needed to solve this problem.

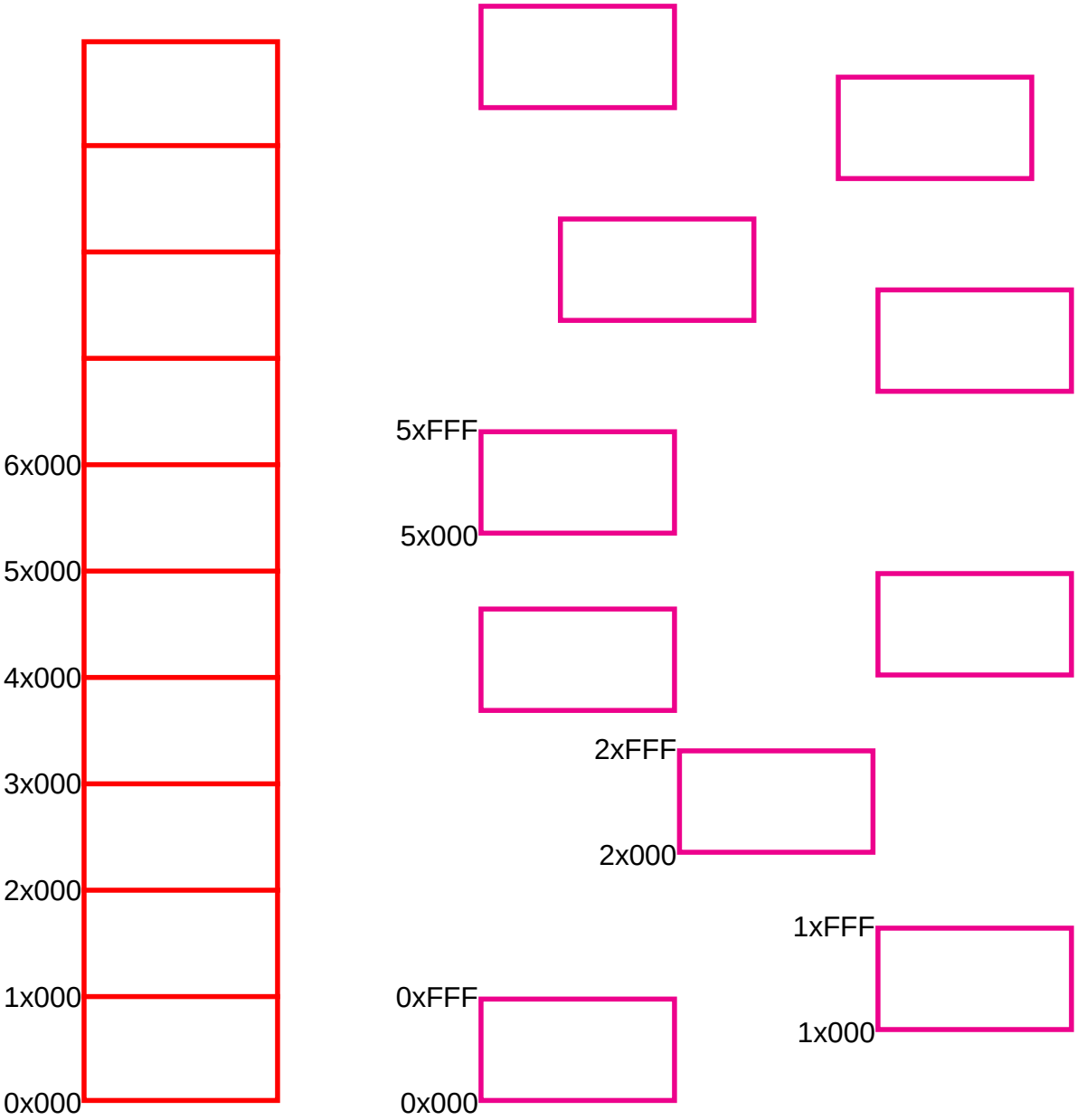
Paging

Each logical address space is broken into fixed–sized blocks called pages. The division is purely mechanical and is done by numbering all the addresses in the logical address space from 0 up and then dividing them into pages:

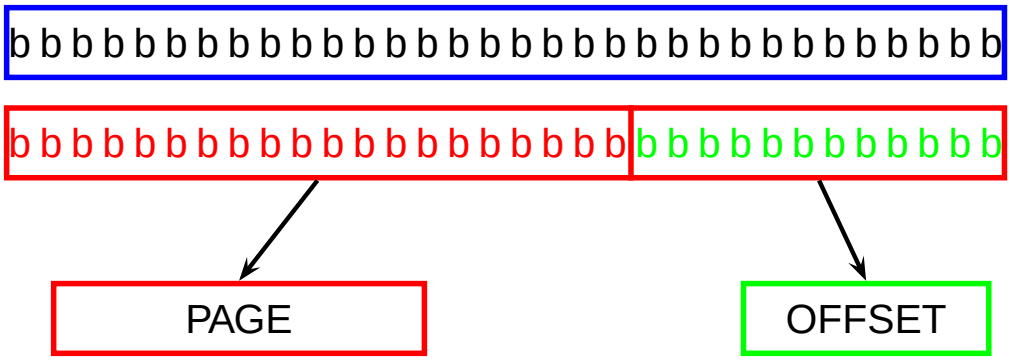


Note: $0x1000 = 16^3 = 2^{12} = 4096$

Each page is treated as a separate entity independent of all the other pages:



Conveniently, the size of a page is a power of 2. This makes the translation from a (uniform) logical address to a (fragmented) **page address** is simple:



Note that the offset field cannot exceed the page size because only this many bits are extracted from the logical address.

In paging, a logical address is a pair of numbers: the page number and the offset within the page. This address is commonly written as *(p,d)* (*d* is an abbreviation of medieval–polish offset).

Where to store a page?

Physical memory is divided into blocks of fixed size called frames. The size of a frame is the same as the size of a page (a frame is sometimes called a **page frame**). the frames are numbered from 0 up (the whole main memory is thus divided). A physical address is written as a pair *(f,d)*, where *d* is, as before, derived from medieval–polish.

We remember that there is only one physical memory in a computer (it is the one you paid for) and the frames are permanent and so are their numbers, i.e. the second frame of the second gigabyte memory chip will always have the same frame number $= 1 + 2^{17}$ if frames have a size of 2^{13} bytes (1 GB = 2^{30} bytes).

A **page** is stored in a **frame**; it can be any frame because they are all the same.

Page table

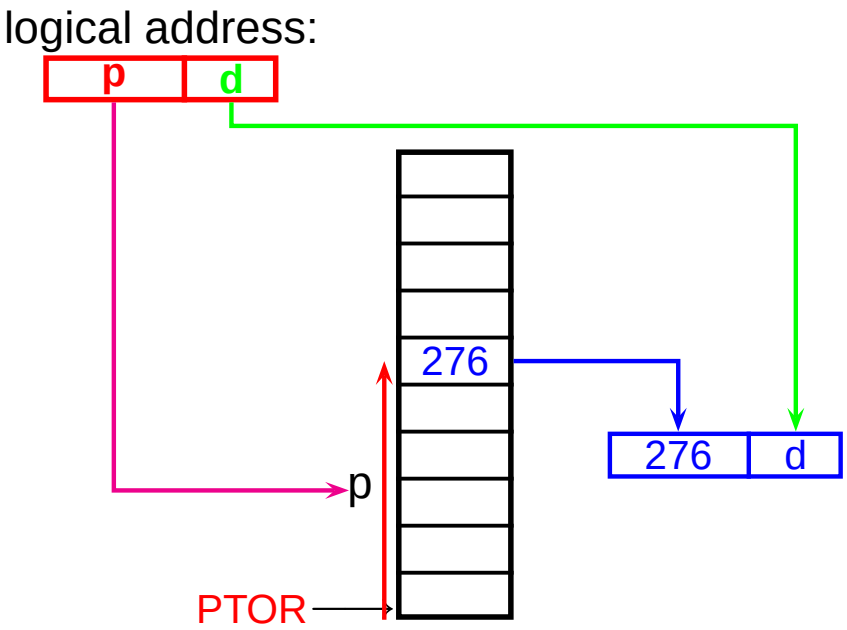
The CPU hands to the MMU a logical address which is made of two parts *(p,d)*. The MMU must convert this pair into a pair *(f,d)* where *d* = *d* (the sizes being the same).

So, the difficult part if to find the *f* corresponding to the given *p*. This can be done using a table which has one entry for each possible page number: the corresponding frame number.

The page table resides in main memory (so that it is accessible to the MMU or in special–purpose high–speed memory (solely to make it faster).

A special control register contains a pointer to the beginning of the page table of the running process (the kernel can also put there a pointer to a fake table when it wants to). This register, called **PTOR** is a control register accessible only in kernel mode, so that user processes do not confuse the MMU.

The simplest form of paging involves a simple table lookup:



Note that kernel loads the **PTOR** register during a context switch. It may also be necessary to load it when accepting an interrupt.

Mercifully, a large part (most) of the logical address space does not actually exist (to be precise: is empty). When this address space is chopped into pages, there is no need to worry about pages that do not exist.

A logical address space is divided into several parts (this division varies from one programming environment to another):

U area: system information associated with the process. Never directly accessible to the process.

Text: the code of the process. Typically **write–protected**.

Data: the constants and global variables used by the process. The constants are sometimes put in a separate part called Const which is **write–protected**.

Stack: the dynamic stack of the process. Its size changes during execution (starts empty). An old custom requires that it occupies the highest addresses and grows downward.

The rest nicknamed HEAP.

The heap

This is the rest of the address space.It usually is very large, much larger than all the other parts combined.

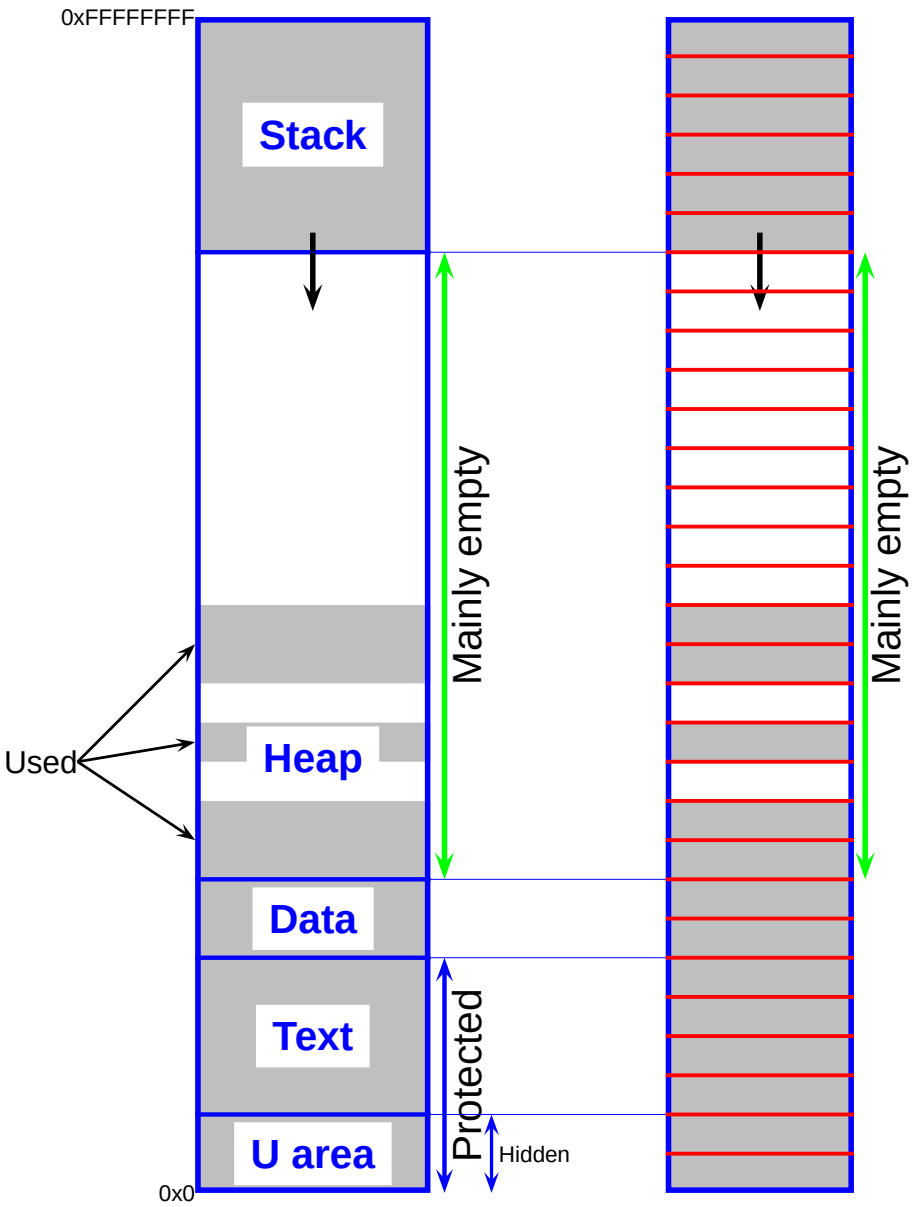
It is bounded from below by the end of the Data part (static) and by the bottom (= top) of the stack (dynamically changing).

It starts empty (i.e. has no contents). Gradually some portions of it become allocated (through **malloc/new**); some of these portions may become empty again (through free/delete or implicit garbage collection in Java&Co.); they may be reallocated afterwards, etc.

The heap also contains dynamically–linked libraries, i.e code that becomes part of the logical address space on demand (only when called).

The heap of a process in execution looks quite chaotic; system designer wish it never existed, but have to accept its necessity because there is no better way to implement dynamic memory allocation and dynamic libraries.

A typical logical address space



Page table

To represent the properties of the various parts of the logical address space, the page table have a few extra fields:

Frame	V	W
	1	1
	0	0
	0	0
	1	1
	0	0
	1	1
	1	1
	1	1
	1	1
	1	0
	1	0
	1	0
	1	0
	0	1
	0	1

V: **Valid** bit (also called Present). If this bit is 0, the MMU will trigger an interrupt when it attempts to access this page table entry.

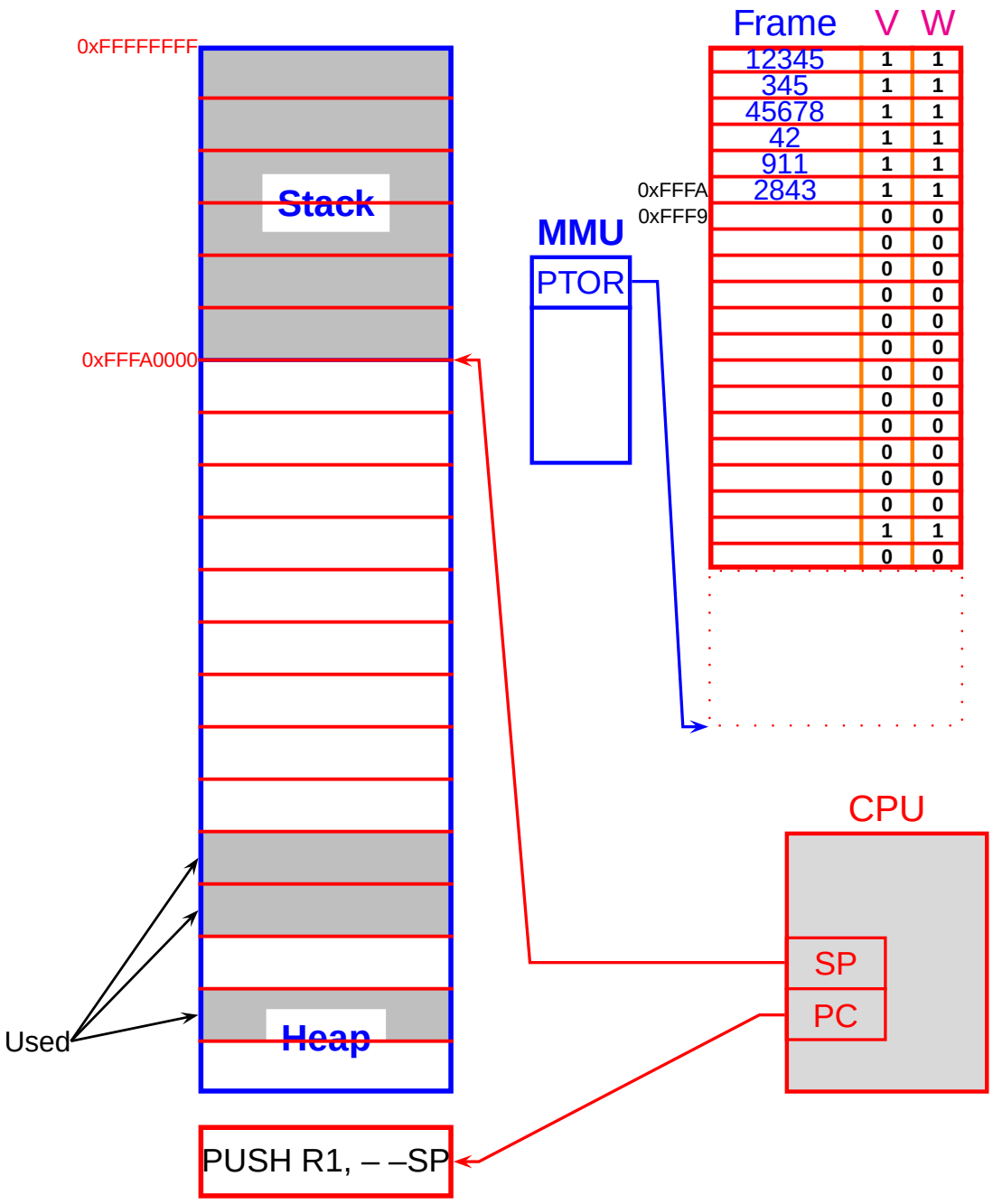
W: if this bit is a 1, the page can be modified; otherwise, the MMU will trigger an interrupt when the CPU attempts to execute a store instruction targeting this page.

C/W: (Copy–on–Write) will be discussed in due time (it is useful after a **fork()**).

other bits: they exist.

A reference to a vacuous place

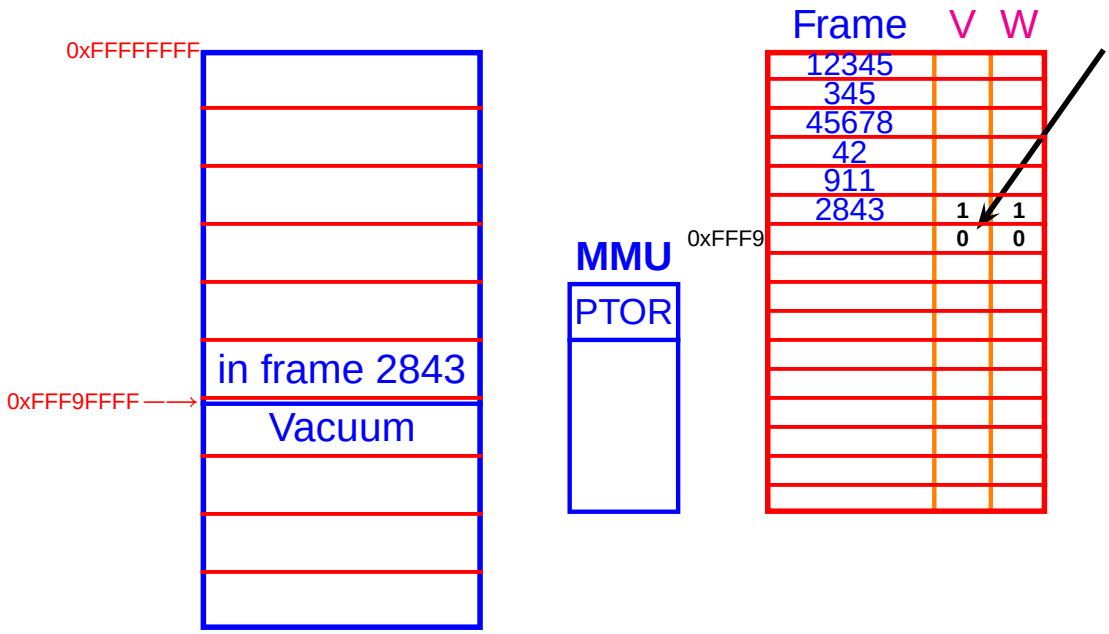
The CPU is going to execute a machine instruction.



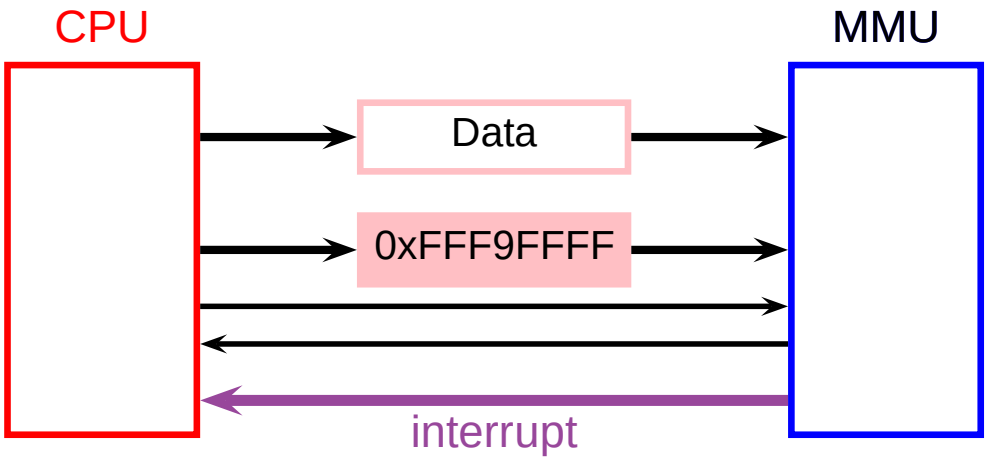
A reference to a vacuous place

The CPU managed to subtract 1 from the SP which now has the value of 0xFFF9FFFF. Now, it puts the value of register 1 in the **MMU**'s **Data Register** and 0xFFF9FFFF in the **Address Register**.

The MMU will object to the address and reply with an interrupt.



If translation is unsuccessful, the MMU activates an interrupt line leading to the CPU forcing the CPU hardware to interrupt (unless interrupts are masked). The interrupt starts an interrupt handler (in the kernel) which decides what to do (not the MMU's job to make decisions).



After the interrupt

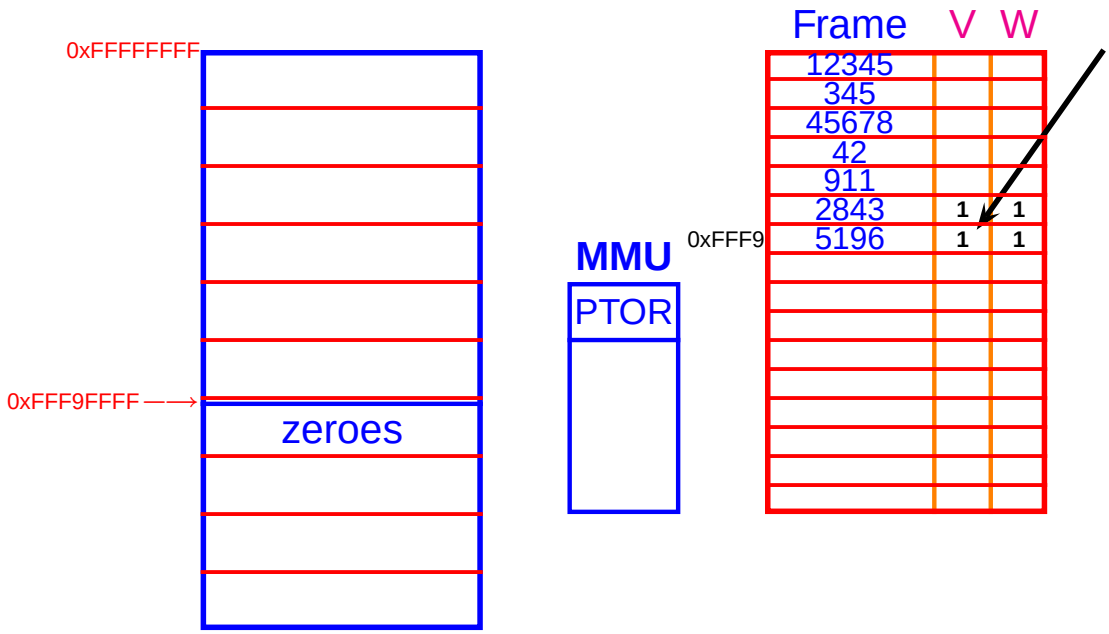
This interrupt, like every interrupt, is handled by the kernel.

The kernel determines it is a memory fault (page fault) and notices that it was caused by an expansion of the stack, something legitimate (within reasons).

The kernel looks for an unused frame. If it find one (say, frame 5196), it fills it with zeroes and makes page 0xFFF9 stored in it (Why zeroes? How is the page stored in the frame?). the page is available for reading and writing (because it is part of the stack).

If there are no unused frames (and virtual memory is not used) the faulting process is blocked until a frame becomes free (the above operation is performed when that happens).

Eventually, the faulting process gets the CPU back. The machine instruction that caused the fault is restarted (not quite from the very beginning–Why not?) and this time it is successful because the translation of 0xFFF9 yields 5196 and not a fault.


0xFFFFFFFF
0xFFF9FFFF
zeroes
MMU
PTOR
0xFFF9
Frame
V
W
12345
345
45678
42
911
2843
5196
1
1
1
1

Efficiency considerations

Any form of address translation will increase the total memory access time.

It is not much of an issue in dynamic relocation where it can be done using a fast circuit (only 2 registers and a simple adder) are involved.

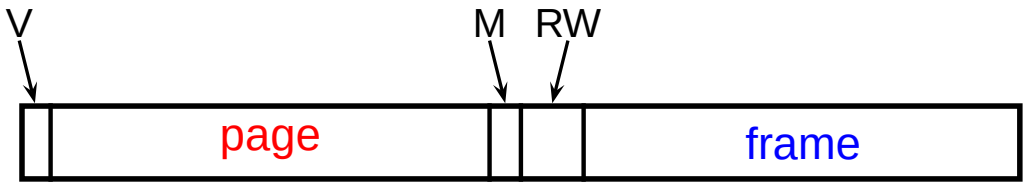
In paging it requires a table lookup; it done the simplistic way, it doubles the time to access the target location which is not acceptable.

Another problem is the size of a page table because it must reside entirely in main memory (Why?). Considering that a page entry is not shorter than 3 bytes, a page table occupies well over 1MB of main memory (per process, whether ready or not). Some partial fixes are: multilevel tables, inverted tables, and other implementations of sparse arrays.

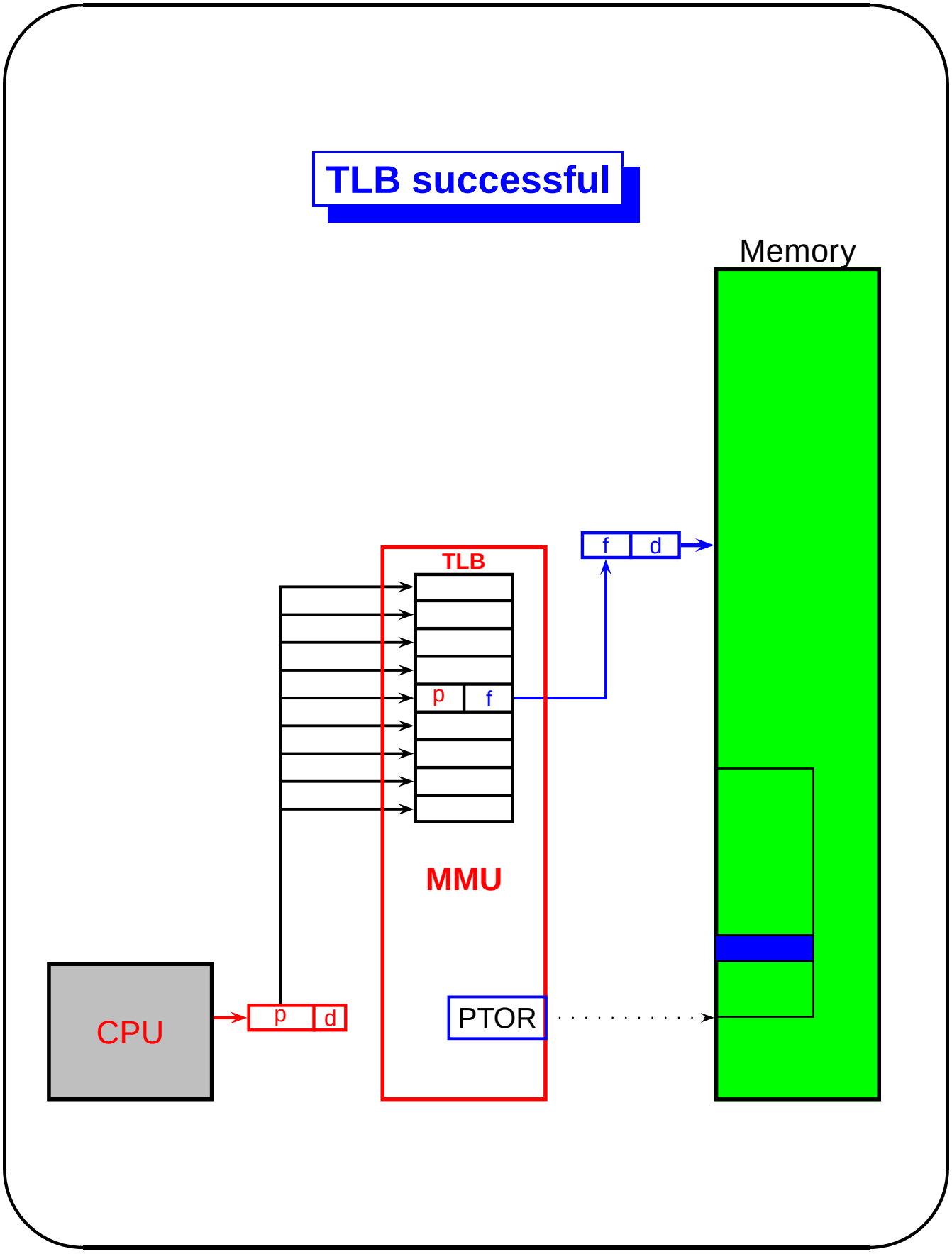
Fast translation

The time needed to convert a page number into a frame number can be reduced by using a dedicated **associative memory**. In the context of paging an associative memory is usually called a **TLB** (Translation Lookaside Buffer). It is a small but very fast parallel circuit which performs simultaneously a number of comparisons.

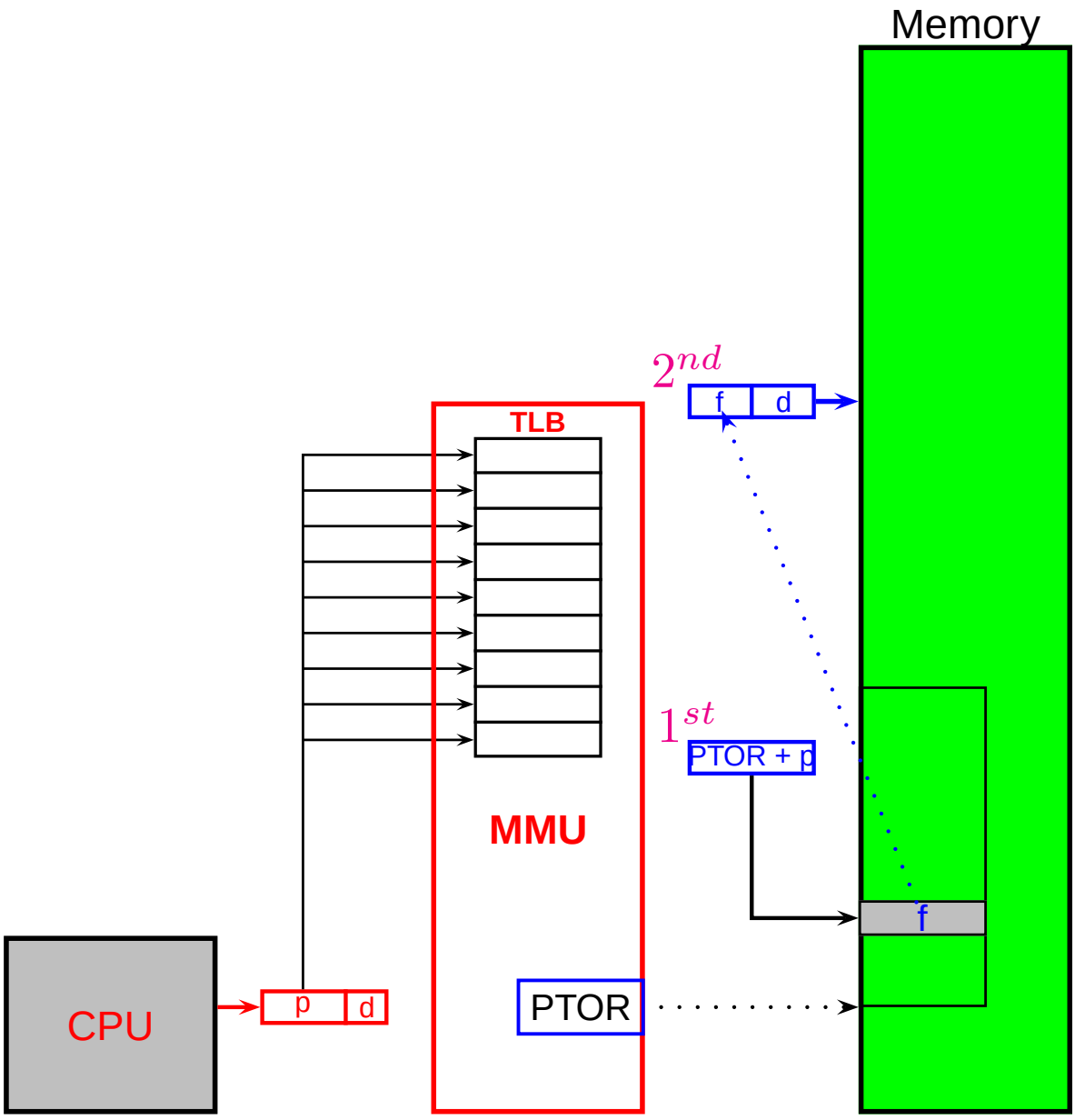
A typical TLB has 64 registers that are accessed in parallel. Each register has a page field and a frame field:



V is the valid bit; RW are two protection bits (actual protection bits vary). The M bit indicates whether the page was **modified** since it was placed in its current frame (makes sense for VM systems only).


TLB successful
Memory
TLB
f
d
p
f
MMU
CPU
p
d
PTOR

TLB unsuccessful



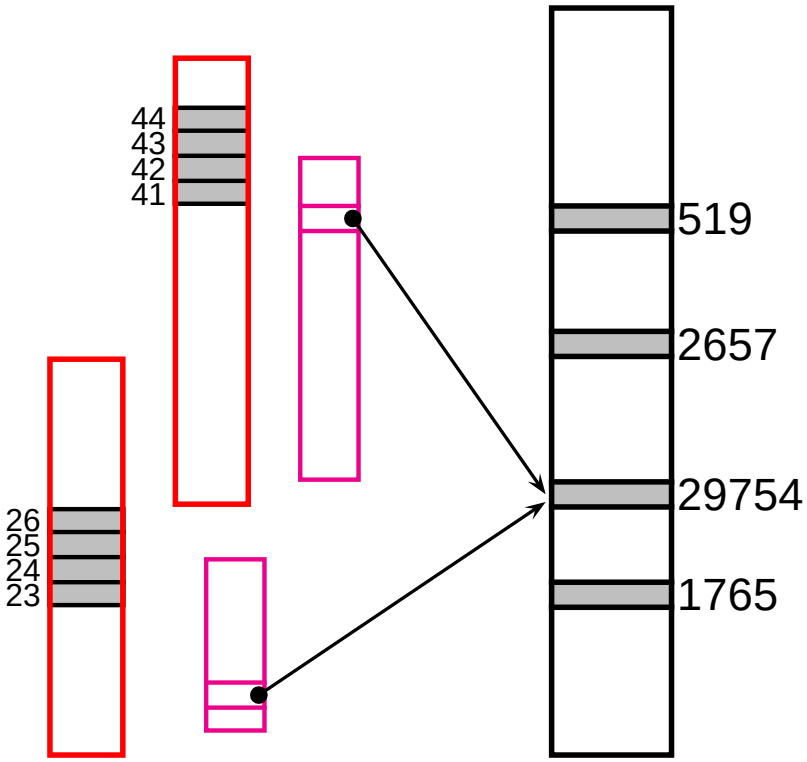
TLB–summary

Associative memory reduces the overhead of paging to 10–20%, an acceptable level. It is not possible to get a much lower overhead because the product $speed \times size$ is approximately fixed (within a reasonable budget). This led to a peculiar trend to increase the page size (to keep the size of the TLB down) with **Vista**'s 4MB pages being the current record–holder.

Shared libraries

Pages can be shared by existing in several page tables.

If a block of pages (typically, a library) needs to be shared, its paged are stored in only one physical location. They are part of several logical address spaces, usually having different page numbers in each space (necessary if they are attached at runtime, the same way as shared memory).



Segmentation

In paging, the whole logical address space forms a single unidimensional entity. There is an advantage in splitting the logical address space into a number of independent units called **segments**.

A segment is an independent address space. Each segment consists of a linear sequence of addresses from 0 to some maximum which differs from segment to segment.

Segments are created automatically by a compiler based on the structure of the code being compiled, so that the addresses forming a segment represent logically related locations.

A segment could be:

- The code of a function.
- An array.
- An object of a class (in OO languages).
- A table of constants (Const).
- The set of global variables (Data).
- The stack.
- Each memory area created using a **malloc/new** or a shmget.
- A file buffer.
- A network buffer.
- Etc.

Page vs. Segment

Segments are similar to pages: they represent a way of fragmenting one logical address space into a number of separate entities that can be handled with more ease.

The differences:

Size: pages have a fixed size while each segment has its own size. Moreover, segments may grow (and shrink) while pages cannot.

Contents: The contents of a page are accidental while the contents of a segment are logically related. Thus, segmentation provides superior protection and sharing options.

Storage: pages fit into **frames** while segments have no ready–made counterpart in physical memory. Big point for paging.

Segment independence

All the pages form a single entity while each segment is separate. This makes a difference when modifying large programs.

In a paging environment, changing a single line requires that the whole program to be relinked again (and possibly even recompiled). Sharing code is possible although it requires that the code be generated in a special way (“reentrant”).

In segmentation, changes made to one function affect only one segment and only this segment needs to be recompiled and possibly relinked (there is a way to avoid static linking altogether in segmentation).

Additionally, segmentation allows a process to create code for itself to execute. for example, a process may create a new segment, place some in it and then start to execute this code. This is impossible in a paging environment.

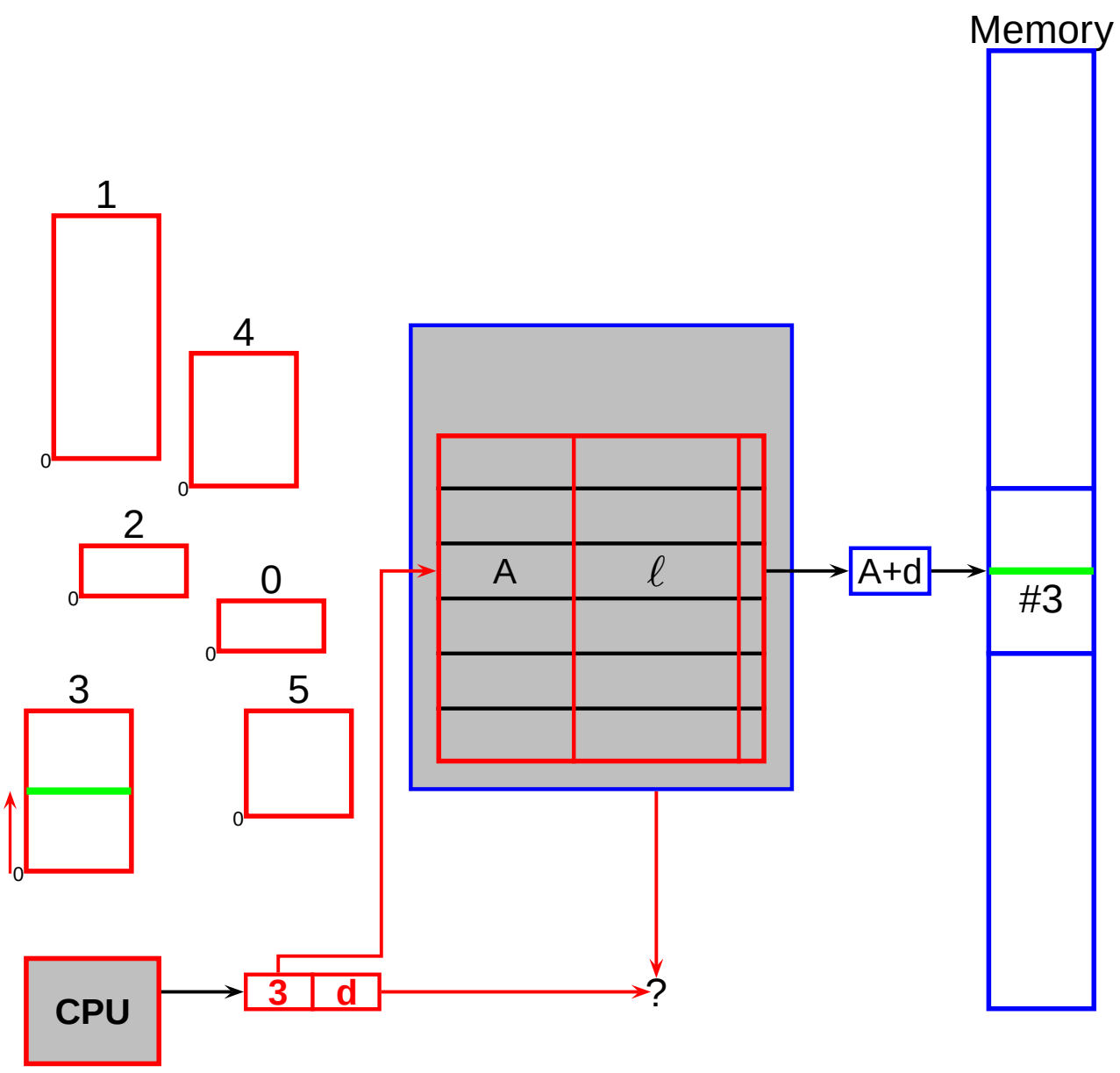
Segmentation—implementation

Segmentation could be implemented in essentially the same way as paging with one big difference: segments are not of fixed size and do not fit into any prefabricated physical memory.

The segment table has one entry for each segment forming the logical address space. The segments are numbered 0 and up.

The address field is a complete address 32 bits or 64 bits, possibly with some lower–order bits omitted. The physical address is created by adding the address to the offset (regular addition is needed). The offset is first compared with the length.

A simplistic translation



Real–life segmentation

The two–access–for–one approach is not realistic, hence there is a need for hardware speedup of address translation. Two approaches are used:

- Associative memory used in the same way as in paging. It actually works better with segmentation because processes seldom require many segments simultaneously.
- Replace conventional addressing with indirect addressing in which all addresses are represented in terms of offsets to addresses stored in a number of special–purpose registers.

Segmentation with selector registers

Special–purpose registers are used in addressing: every address is in the form:

(*selector* , *offset*)

where selector is a register pointing to an entry in a **Segment Table** which looks approximately like this:

base	
limit	bits

The base is a regular physical address; it corresponds to the beginning of the segment in physical memory.

To make this scheme fast, the selected ST entries are also kept in a high–speed microprogrammable memory, so that there is no need to access the actual Segment Table.

The overhead results from the need to load a new selector every time a function is called or when a new data area becomes relevant. It may lead to very poor efficiency in extreme cases (same phenomenon as **thrashing** in Virtual Memory).

Paged segmentation

If one expects that segments will be large, it is natural to divide them into pages; this yields a combination of the two memory management methods.

Paged segmentation is useless without hardware support which comes in the form of selector registers, similar to those used in segmentation.

Pentium notes from Iowa

Paged segmentation in Pentium