OPERATING SYSTEMS

by

FARAH

Chapter 2
The memory

2.1. Introduction

Main memory is where programs and data are located when the processor is executing them. It is contrasted with the concept of secondary memory, represented by disks, which have a larger capacity and where processes can reside before being executed.

Even more than for other IT resources, the price of memory has fallen and the unit capacity of circuits has increased. However, the need to manage it optimally is still fundamental, because despite its high availability, it is generally never sufficient. This is due to the ever-increasing size of programs.

2.1.1. Multi-programming

The concept of multiprogramming is opposed to that of monoprogramming. Monoprogramming allows only one user process to be executed. This technique is now only used in microcomputers. In MS-DOS, for example, there is the system in low memory, the peripheral drivers in high memory (in an area ranging from 640 kB to 1 MB) and a user program in between.

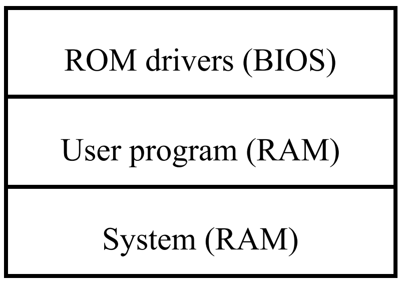


Figure1 The organization of DOS memory.

Multi-programming allows several independent processes to be executed at the same time[1] . This technique makes it possible to optimize processor utilization by reducing I/O waits. Multiprogramming means that several programs are held in memory at the same time, and it is this technique that gave rise to modern memory management.

[1] It should be noted that MS-DOS, although not multi-programmable, is not a mono-programmable system *in the strict sense* either; it is a creature of both.

2.1.2. Physical registers

Memory management is almost impossible without the help of hardware. In particular, it must provide protection. In multi-user systems, for example, a user must be prevented from accessing the system kernel or other user programs in any way.

To ensure fundamental protection, most processors[2] have two registers delimiting the domain of a process:

- the base register and
- the limit register.

Protection is then provided by the hardware, which compares the addresses sent by the process with these two registers.

2.2. Fundamental concepts

2.2.1. Production of a programme

Before being executed, a program must go through several stages. Initially, the programmer creates a file and writes the program in a source language, such as C. A compiler transforms this program into an object module. A **compiler** transforms this program into an object module. The object module represents the translation of the instructions in C into machine language. The code produced is generally relocatable, starting at address 00000, and can be translated to any memory location by giving it the base register as its initial reference. The addresses then represent the offset from this register.

Object modules can be grouped together in specialized libraries, for example using the ar and ranlib commands (for direct access) under Unix or TLIB with Borland's C compiler. The libraries are then brought together in a directory, usually /usr/lib on Unix.

Calls to external procedures are left as branch points. The **link editor** maps these points to functions contained in the libraries and, in the case of a static link, produces a binary image. Some systems, notably Unix, allow dynamic links and postpone the link editing phase

[2] Unfortunately not on the 8086, which had considerable consequences for DOS and Windows systems.

until the load time. Objects and libraries then have to be constructed in a slightly different way. This technique means that libraries can be updated without recompiling, and there is less disk clutter.

The **loader** links system calls with the kernel, such as the write call. Finally, it loads the program into memory.

In addition to the compiled code (the text) and the initialized data area, the executable file contains a certain amount of other information. In the case of the Unix system, this includes a header consisting of a 'magic' number (a type control code), various sizes (text, data, etc.), the program entry point, and a table of symbols for debugging purposes. In the case of MS-DOS, several formats co-exist, in particular COM and EXE.

2.2.2. Management principles

To load, the system **allocates** a free memory space and places the process in it. It will free this space once the program has finished.

In many cases, it is not possible to fit all the programs together in memory. Sometimes the size of a single program is too large. In this case, the programmer can implement an *overlay* strategy, which consists of dividing a large program into modules and loading these modules when they are needed. However, this is very time-consuming and requires a re-division for each new program.

Modern operating systems implement strategies that relieve the programmer of these concerns. There are two main strategies for managing loads: **toggling** and **virtual memory**.

2.3. Allocation

Before implementing a technique for managing main memory by toggling, it is necessary to know its state: the free and occupied zones; to have an allocation strategy and finally to have release procedures. The techniques we are going to describe are used as a basis for toggling; they are also used in the case of simple multiprogramming where several processes are loaded into memory and held until the end of their execution.

2.3.1. Memory status

The system keeps track of occupied memory locations by means of a bit table or a linked list. Memory is divided into allocation units or blocks.

2.3.1.1. Bit tables

The state of the memory blocks can be kept using a bit table. Free units are marked with a 0 and occupied units with a 1 (or vice versa).

0	0	1	1	0	0	

Figure 2

The bit table technique is simple to implement, but is rarely used. The following observation can be made: the smaller the allocation unit, the fewer the losses during allocations, but on the other hand, the more space the table takes up in memory.

2.3.1.2. Linked lists

Memory can be represented by a chained list of structures whose members are: type (free or occupied), start address, length, and a pointer to the next element.

For a memory with the following state :

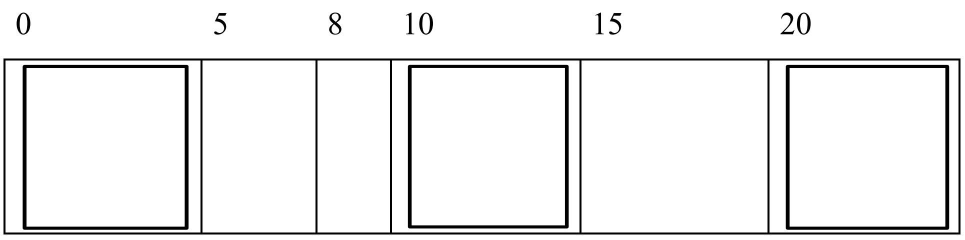


Figure 3

we'd have the list:

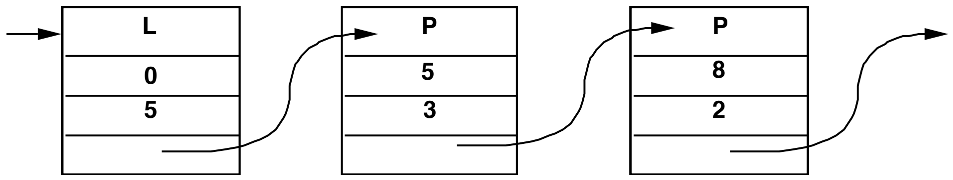


Figure 4

This diagram can be slightly modified by taking two lists: one for processes and the other for free zones. The list of free blocks can itself be represented by reserving a few bytes of each free block to contain a pointer to the next free block.

The MS-DOS system uses a variant of this process, with a 16-byte header preceding each zone (arena) in memory. The headers contain the type of arena (a pointer to the process context or 0) and its size.

2.3.2. Allocation policies

There are three main strategies for allocating free space for a process: *first* fit, best *fit* and *worst* fit.

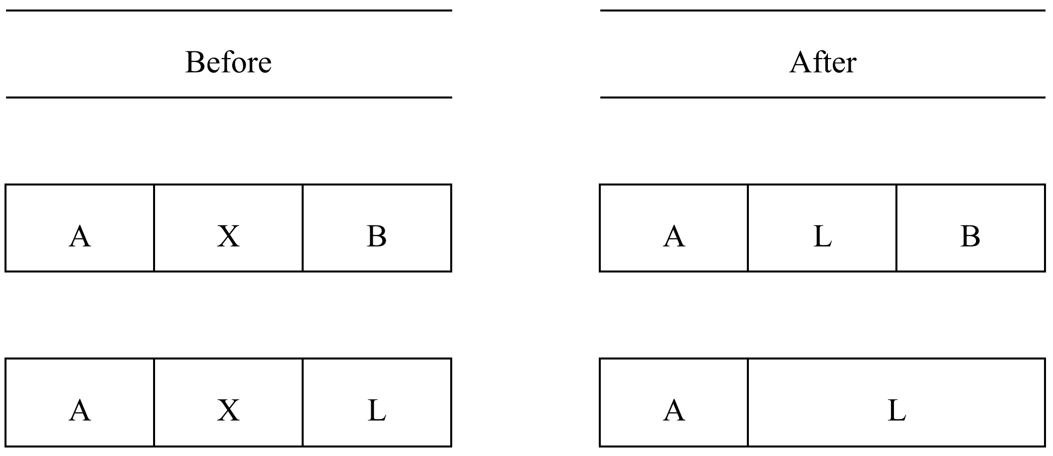
In the case of "first fit", we take the first free block in the list that can contain the process we want to load. The "best fit" tries to allocate the smallest memory space available to the process. The "worst fit" takes the largest available block and splits it in two.

Simulations have shown that the "first adjustment" is better than the others. Paradoxically, the "best fit", which is more expensive, is not optimal because it produces significant fragmentation.

2.3.3. Release

Freeing occurs when a process is cleared from memory. The block is then marked as free and possibly merged with adjacent blocks.

Assuming that X is the block to be released, we have the following merge schemes:



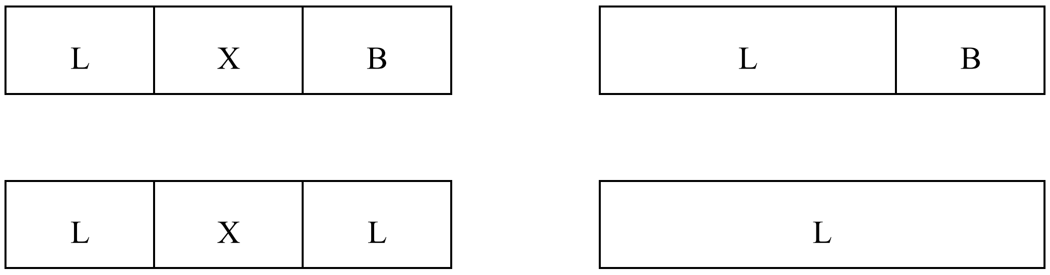


Figure 5

Memory retrieval

Memory fragmentation is particularly damaging because it can quickly saturate the available space. This is particularly true for windowing managers. To reduce fragmentation, memory can be compacted regularly. To do this, move processes, for example, to the bottom of memory and store them one after the other in a contiguous manner. A single free block is then built at the top of the memory. This operation is expensive and sometimes requires special circuits.

Furthermore, once an object or zone has been used, the system programmer must reclaim the memory "by hand" by freeing the pointer to this zone - free(). This is a source of errors because this operation is sometimes forgotten. If you allocate in a function, there is no way of accessing the pointer once the function has returned. The block of memory is then lost and unusable. This is known as a "memory *leak*".

Some languages or systems incorporate automatic memory retrieval - *garbage collection*. This frees the programmer from this task. This is the case with Java. It is then very easy to immediately eliminate a zone by assigning object = null. Without this, the garbage collector has to determine on its own that a zone no longer has a reference in the rest of the program. Automatic memory retrieval has been the subject of controversy, but as machines become faster, this quarrel is no doubt over .[3]

The recovery algorithm can cause problems because periodically the system stops abruptly to make way for the recovery. You can request that it be started explicitly

[3] According to Prolog (or Lisp) proponents, recovery is too important to leave to programmers, and according to C++ proponents, it is too important to leave to the system.

(synchronously) using the java.lang.Runtime.gc() method. Recovery is also said to be synchronous when there is no more memory in the heap. The interpreter must be run with the option: -noasyncgc in order to call the fetcher. The amount of free memory is obtained using the Runtime.freeMemory() method. The default heap is 1 MB. You can set its value by running the interpreter with the -ms16m option (for 16 MB).

2.4 Uniform memory :

The first memory version offered was a fully addressable M C. This means that users can use it in any way they like. And there is no need for any special hard-ware to manage this memory. This method has its limits, however, since the operating system has no control over interrupts. There is no resident monitor to manage system calls or errors (especially the reading of control cards, for example).

2.5 Resident monitor:

This method suggests dividing the memory into two parts, one part for the user and the second for the resident monitor part of the O.S.. It is more convenient to place the resident monitor in lower memory, using a fence register to protect the monitor. All addresses generated by the user program (instructions or data) must be compared with this register.

Program generates an address
If address >= contents of lock bar register then access possible
Otherwise error, generate an interrupt.

Advantages Simple, The task has all the memory.
Disadvantages:
- Poor use of the MC if the program is smaller than the memory,
- The program must be less than or equal in size to the MC
- If I/O the CPU remains inactive

2.6 MULTIPLE PARTITIONS:

This method follows directly from the principle of multi-programming. The memory is divided into several partitions, each assigned to a program. For user security purposes, two registers are used. These registers are used to delimit the address space of a program:

- Limit registers (lower and upper limits):

Used for static location.

- Base register and Limit register

All addresses must be below the Limit, and all addresses are recalculated by automatically adding the contents of the Base register.

2.6.1 Fixed partition:

The MC is divided into partitions independently of the jobs in progress. For example, if we have a 32KB memory and we divide it into partitions as follows:

- Resident monitor 10 Kb
- Small spots 4 Ko
- Average stains 6KB
- Large spots 12 Ko

Advantages :

It's easy to do, and programs can be sorted by size so that they occupy the right amount of memory.

Disadvantages:

- Limiting the degree of multi-programming
- Program size limited by partition size
- Poor use of memory, Notion of "fragmentation

Example:

Scores	Partition size (KB)	Program size(KB)
1	8	5
2	16	11
3	128	50
4	256	131

If a program arrives with a size of, say, 100KB it cannot be served, even though the unused global space is greater than 100KB.

2.6.2 Variable scores:

The partitioning is done as the work is carried out, so as to adapt the size of the partitions to the size of the tasks to be carried out.

Example:

If we have a 256 KB MC, and the monitor takes up 40 KB, that leaves us 216 KB to share between users.

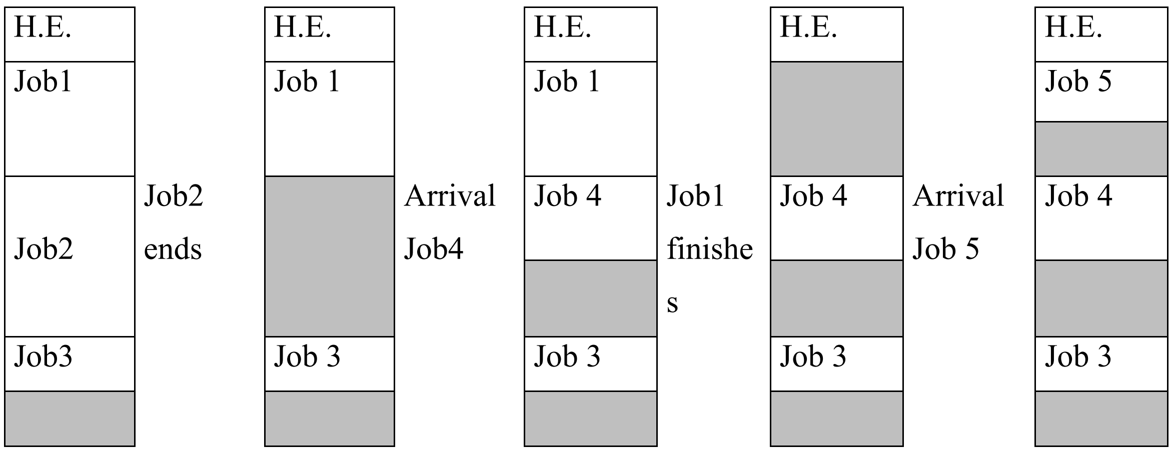


Figure 6

Job 1 =[40k - 100 k]; Job2=[100K-200K]; Job3=[200K-230K]; Job4=[100K-170K]; Job5=[40K-90K]

Advantage:

- Higher degree of multi-programming

Disadvantages:
- Complexity of Allocation and Deallocation Algorithms
- Fragmentation is not eliminated
- The size of the program is limited by the available (contiguous) memory space.
- If a program is loaded (@ virtual→ @ physical) it is no longer possible to move it, so it is impossible to recover the free fragment space.

2.7. SWAPING

Swapping is used when not all processes can fit in memory simultaneously. In this case, some processes must be temporarily moved to a temporary memory, usually a reserved part of the disk (*swap area* or *backing store*).

On disk, the swap area of a process can be allocated on demand in the general swap *area*. When a process is unloaded from main memory, a place is found for it. Swap areas are managed in the same way as main memory. A process's swap area can also be allocated once and for all at the start of execution. When unloading, the process is sure to have a free waiting area on the disk.

The system executes the processes in memory for a certain quantum of time and then moves one of these processes to one of those waiting in temporary memory. The replacement algorithm can be a turnstile.

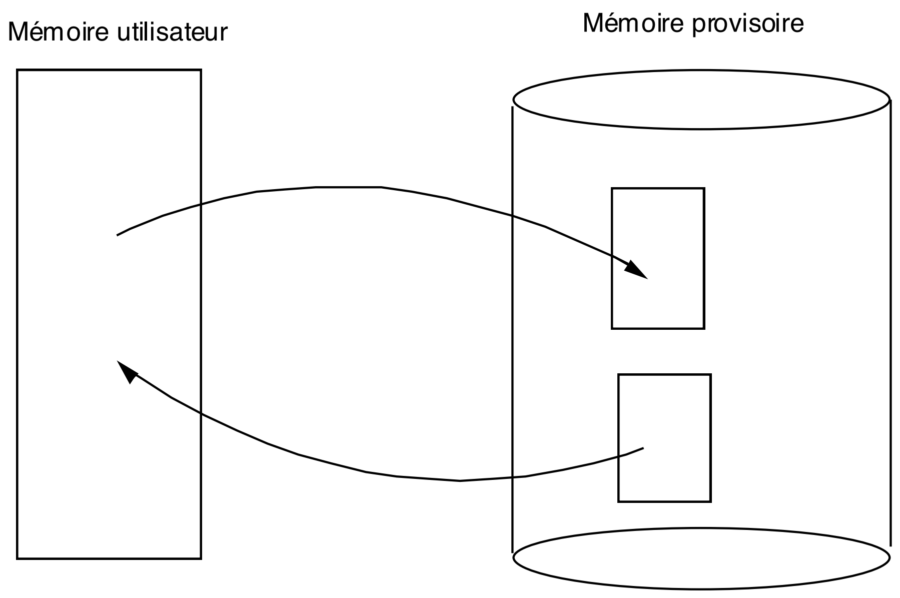


Figure 7

The to-and-fro system, while helping to compensate for the lack of memory required by several users, does not allow programs larger than the main memory to be executed.

It should be noted that the time required to execute a process must be much longer than the time required for swapping.

2.8. Pagination

Paging allows you to have a process in memory whose addresses are non-contiguous. To achieve this, the process address space and physical memory are divided into pages of a few kilobytes each. The process pages are loaded onto free pages in memory.

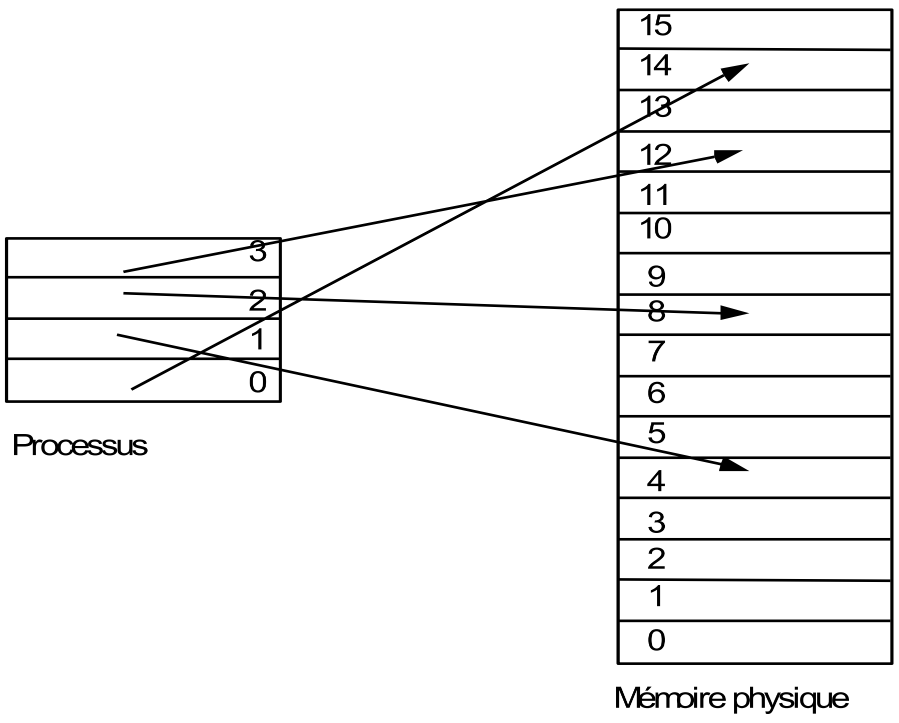


Figure 8

The location of pages is preserved using a transcoding table:

0	14
1	4
2	8
3	12

Pagination allows you to write re-entrant programs, i.e. where certain pages of code are shared by several processes.

2.9. Segmentation

Whereas paging offers a flat, undifferentiated address space (this is offered by the 68000 family of µ-processors), segmentation divides processes into very specific segments. You can have segments for procedures, for the symbol table, for the main program, and so on. These segments can be relocatable and originate from a base register specific to the segment. Segmentation can also be used to share editor code between several processes, for example. This sharing then relates to one or more segments.

A reduced example of segmented architecture is provided by the 8086 family, which has 3 segments: Code Segment, Stack Segment and Data Segment.

2.10. Virtual memory

2.10.1. Presentation

Virtual memory allows:

- Execute programs that exceed the size of real memory. To do this, processes and real memory are "paged" into pages of a few kilobytes (usually 1, 2 or 4 kB).

- **Physical** memory is expensive and therefore generally of limited capacity, which gave rise to the idea of using **secondary** memory (disks, extended memory, etc.), which is inexpensive. And try to use secondary memory "as" RAM memory.

The total process footprint is the address space or virtual memory. This virtual memory resides on disk. Unlike the paging described above, only a subset of pages is loaded into memory. This subset is called the physical (real) space.

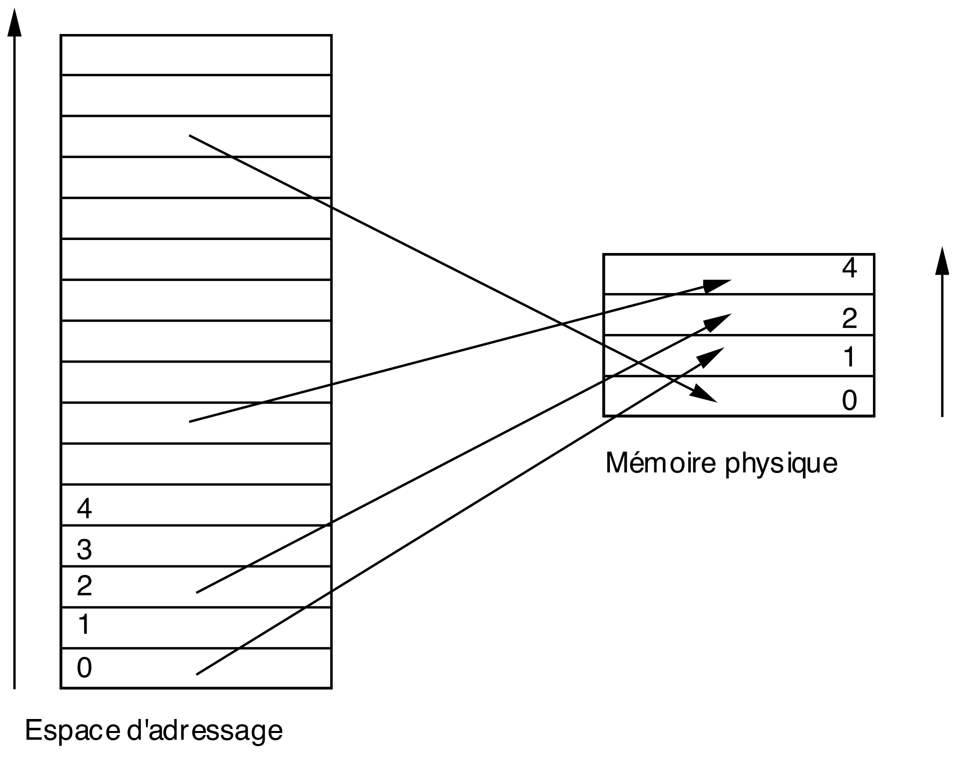
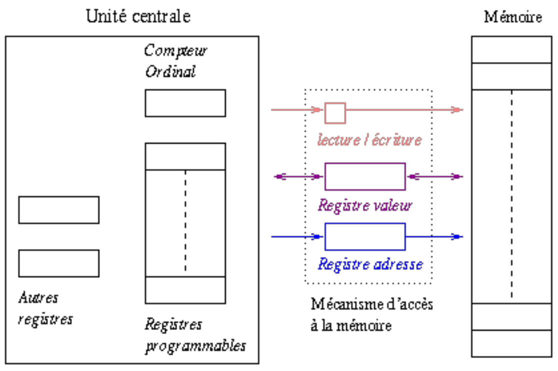


Figure 9

The problem is to match a virtual address to a real address in memory, to protect users from each other and to manage information sharing.

We use a topographical function that associates a real address with a virtual address.

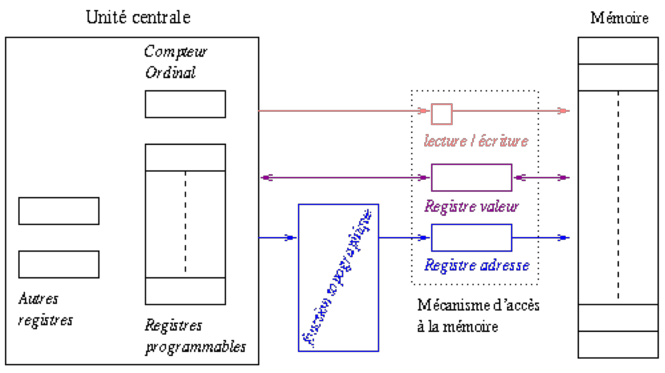
This is the classic memory access architecture:



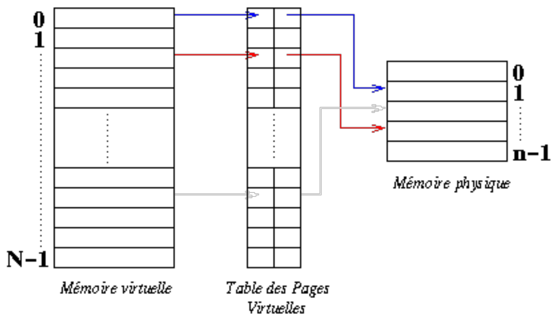
This is the topographic function:

```
Function  topo(x:virtual_address):real_address;
start
topo:=f(x);
end
```

And here's the architecture with dynamic relocation:



Virtual memory, with its table of pages, is **a** possible implementation of the topographic function.



When an address is generated, it is transcoded, using a table, to match its equivalent in physical memory. This transcoding can also be carried out by Memory Management Unit (MMU) hardware circuits. If this address corresponds to an address in physical memory, the MMU transmits the real address on the bus, otherwise a page fault occurs. To access the page whose address has been generated, it must first be loaded into real memory. To do this, a "victim" page is selected from the real pages; if the victim page has been modified, it is transferred to virtual memory (on disk) and the page to be accessed is loaded in its place.

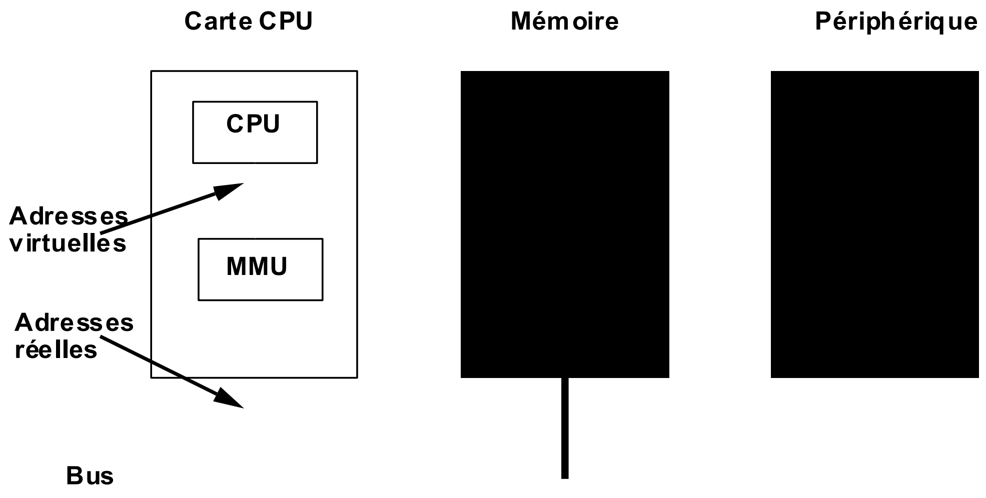


Figure10 The memory management unit.

It is easier to implement the algorithm using sizes corresponding to powers of two. For a virtual or real address, the most significant bits are reserved to encode the real and virtual pages. The least significant bits encode the offsets within each of these pages. For example, let's assume that the pages are 4 kB in size; that the process size is 16 pages; and that the memory allocation is 4 pages. You need 2 bits to code the real pages, 4 bits for the virtual pages and 12 bits for the offsets.

Structure of real memory addresses

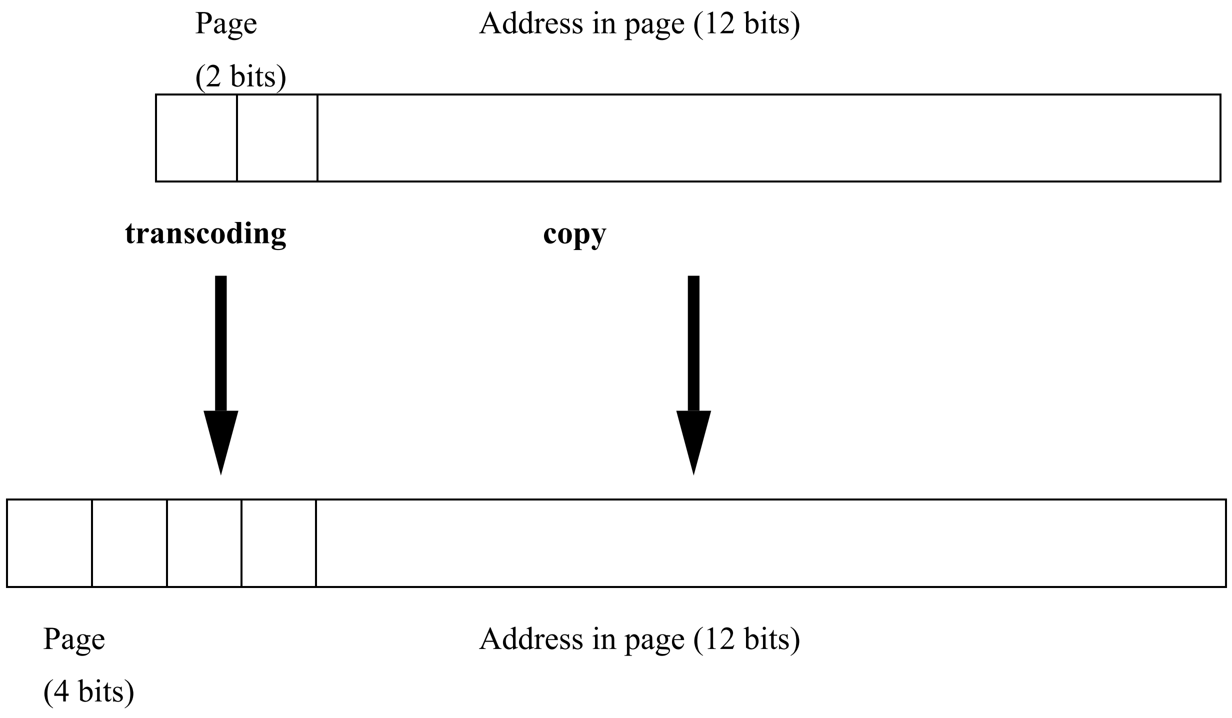


Figure11 Memory address structure of the address space

To perform transcoding, tables containing the necessary data are kept, including: a bit to mark the presence of the page in real memory and a modification bit to indicate whether the page has been written to. In the latter case, the page must be transferred to disk if it is to be replaced by another.

P. Virt.	P. Real.	Present	Modif.
0	01	0	
1	-	X	
2	10	0	
3	-	X	
4	-	X	
15	00		

Figure 12

In general, it is a bit in the PSW (Program Status Word) that indicates whether or not virtual memory is being used.

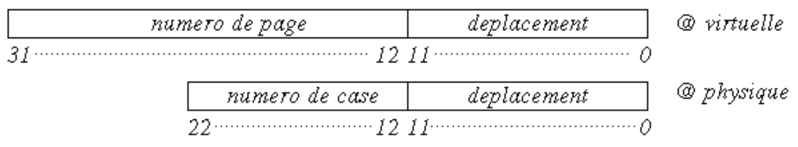
Basic principles and mechanisms of pagination

A processor with a 32-bit address bus can address *2^32* bytes, or 4 GB. The computer has 8 MB of physical memory.

L = size of the page or box, for example 4096 bytes, i.e. *2^12*.

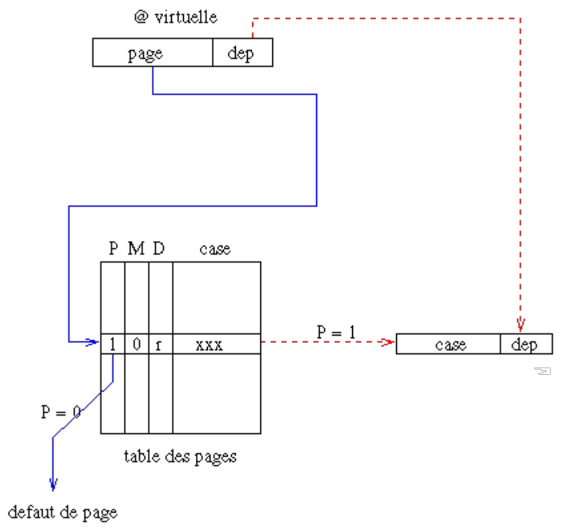
N = number of pages in virtual memory, for example 1 Mega page, or *2^20*.

n = number of cells in physical memory, for example 2048 cells, i.e. *2^11*.

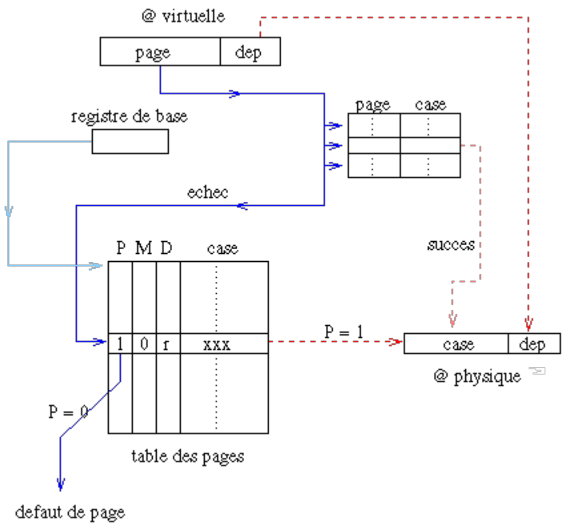


displacement is the same (physical and virtual pages are the same size) ;

if the virtual page is not present in physical memory, a *page fault* occurs.



To speed up the process, *associative memories* are used to record the last pages used:



2.10.2. Page replacement algorithms

Choosing a victim - replacement

Many algorithms :

- **FIFO** - First In First Out: chronological order of loading ;
- **LRU** - Least Recently Used: chronological order of use ;
- **FINUFO** - First In Not Used, First Out (Clock algorithm): LRU approximation;
- **LFU** - Least Frequently Used ;
- **Random**: at random ;

Performance: LRU, FINUFO, [FIFO, Random].

System optimization: take **locality** into account by **preloading** pages **before they** are needed.

Locality: at a given point in time, references observed in the recent past are (generally) a good estimate of future references.

On average, 75% of references interest less than 20% of the pages. **This is non-uniformity**.

we're going to try to **anticipate** demand.

The problem of the size of the POS

To be used, the POS must be placed in physical memory.

For example, if we have *2^20* virtual pages, the POST will be approximately *2^20 * size of an entry* = 10 MB if an entry is 10 bytes long.

In other words, more than the size of the physical memory !!!!

Solution: we will paginate the POS.

Two-level pagination

Virtual memory is divided into **Hyperpages**, which are themselves divided into **pages**.

A virtual address = hyperpage number; page number; movement.

Warning! Memory access is slower with indirection (using associative memories).

The optimal page replacement algorithm consists of choosing as the victim the page that will be called as late as possible. Unfortunately, we cannot implement this algorithm, but we try to approximate it as closely as possible, with results that differ from the optimal algorithm by less than 1%.

The FirstIn-FirstOut technique is fairly easy to implement. It consists of choosing the oldest loaded page as the victim.

The *Least* Recently Used algorithm is one of the most effective. It requires special hardware to implement it. In particular, a column of counters must be added to the table. Degraded versions can be implemented in software.

Advantages/disadvantages of pagination

Advantages :

- Better use of physical memory (programs implemented in fragments, in *non-consecutive* pages).
- Possibility of loading pages only when they are referenced (on-demand loading).
- Independence of virtual space and physical memory (virtual memory is generally larger).
- Only modified pages can be dumped to disk.
- Possibility of dynamic overlay (coupling).

Disadvantages:

- Internal fragmentation (not all pages are filled).
- Impossible to link two (or more) procedures linked to the same addresses in virtual space.

2.10.3. Other considerations

2.10.3.1. Thrashing

Pages of physical memory can be allocated equally to each process. For example, if the total memory is 100 pages and there are five processes, each process will receive 20 pages. You can also allocate pages in proportion to the size of the programs. If one process is twice the size of another, it will receive double the number of pages.

Used as they are, these allocation techniques can cause the system to collapse. The system is only viable if page faults are kept below a relatively low limit. If there are too many processes, the space available to each will be insufficient and they will spend their time managing page faults.

The risk of collapse can be limited by considering behaviour charts. The number of page faults as a function of the number of pages allocated takes the form of a hyperbola. If a process causes too many page faults (above an upper limit), it will be allocated more pages; below a lower limit, it will be withdrawn. If there are no more pages available and too many page faults, we will have to suspend one of the processes.

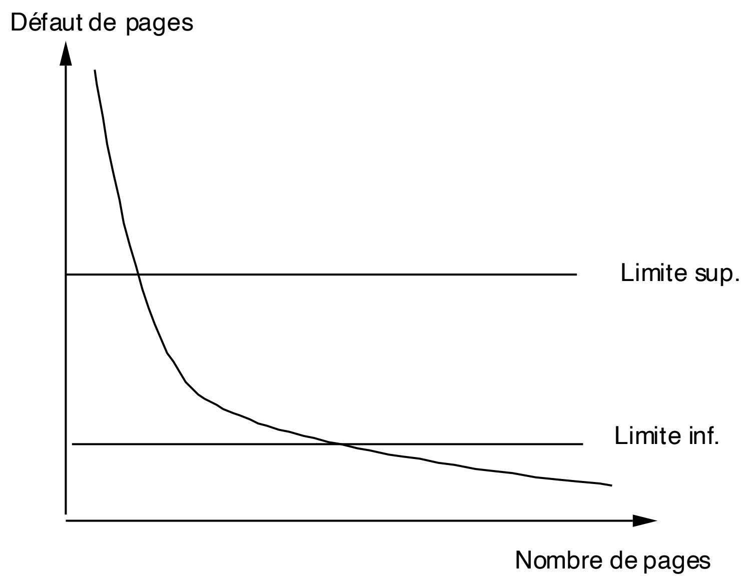


Figure 13

2.10.3.2. The work set

To determine the viable space of a process, we use the working set model. The working set is made up of the areas of the process that are accessed over a short period of time (around ten memory references). For example, updating an individual in a database will use certain pages of code and the page corresponding to the individual.

Simulations have shown that this workload is relatively stable at any given time. An optimal allocation would be to allocate to each running process as many pages as its workspace requires. Under these conditions, page faults will only occur when the workspace is changed. In fact, this model is only used for prepagination.

2.10.3.3. Local or global allocation

When a page is removed from main memory, the oldest page can be selected:

- from a global point of view (the oldest in the system);
- from a local point of view (the oldest part of the process).

In general, global allocation produces better results.

2.10.3.4. Prepagination

When a process is launched or resumed after a suspension, a certain number of page faults are inevitably caused. You can try to limit them by recording, for example, the work set before a suspension. You can also try to guess them. For example, when a programme is launched, the first pages of code are likely to be executed.

2.10.3.5. Return on instructions

On most processors, instructions are coded over several operands. If a page fault occurs in the middle of an instruction, the processor must return to the beginning of the initial instruction and re-execute it. It must then be given the means to determine the address of the first byte and possibly to cancel certain increments.

This instruction feedback is only possible with the help of hardware. For example, the 68010 has a register that stores the addresses of the first instructions and the increments. This makes paging possible. The 68000 does not and cannot do this.

2.11. An example of memory management on a microprocessor: the 386

The 386 processor is interesting because it combines paged memory techniques with segmentation techniques[4] . It has 16k independent segments with a capacity of up to 1 billion 32-bit words.

Virtual memory is based on two tables:

- the local descriptor table (LDT), specific to each program. It contains its segments, including code, stack and data segments;
- the global descriptor table (GDT), unique for the whole system and shared by all processes. It contains the system's segments, particularly those of the kernel.

The 386 has six segment registers. To access a particular segment, for example the code segment of a process, it loads a selector into one of the registers. This selector corresponds to an index in one of two tables. Each entry in these tables contains the base address of the segment, the boundary address and certain other fields. Using the base address, the microprocessor can convert the generated offsets into 32-bit linear addresses.

The 386 has optional paging with 4 kB pages. When it is activated, the previous address is interpreted as a virtual address and converted into a real address. Paging is done on two levels. The generated address is interpreted as a function of 3 fields:

10 bits	10 bits	12 bits
Dir	Page	Offset

Figure 14

Dir is an index in a table containing a pointer to a table of pages. In this table, Page is another index also containing a pointer (double indirection) to a real page. Within this real page, Offset is the address of the element you are looking for.

[4] It is also the most widely used 32-bit processor.

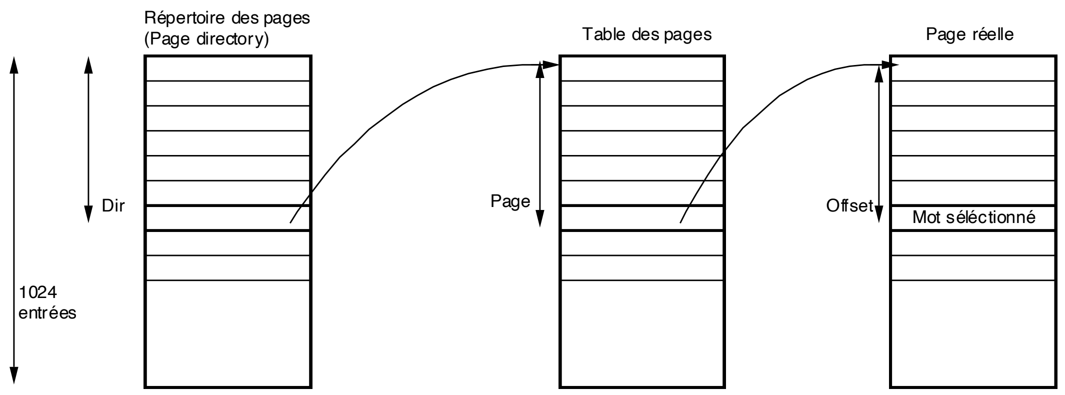


Figure 15

2.12. Unix system calls

fork() causes memory to be allocated

exec() causes a memory modification

wait() causes memory to be freed

Unix executable files contain text and initialised global data; uninitialised data (BSS) is allocated at load time. In memory, code segments cannot be modified in most compiled languages and can be shared on most modern systems. This feature optimises memory usage. On the other hand, the other segments (data and stack) are specific to each process.

The data and stack segments are continuously modified during execution and their size may vary. The following calls modify these sizes:

int brk(caddr_t addr) is used to move the boundary of the data zone to addr. It can be used by malloc. Returns -1 if it fails.

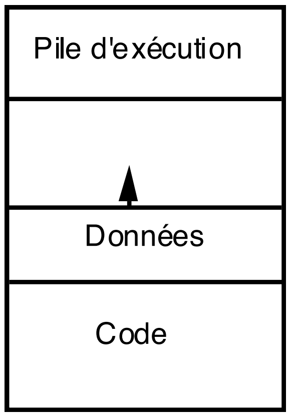


Figure 16

caddr_t sbrk(int incr) is similar to brk. Instead of setting the address, it extends the incr data zone.

void * malloc(int incr) allocates a contiguous space of size incr in the data zone. Returns a pointer to the zone and NULL on failure. Malloc is a fairly primitive memory allocation function. It uses the first-fit algorithm and does not recompact free blocks in the data zone. It causes fragmentation. In the book on the C language by Kernighan and Ritchie, you can find the implementation code for malloc. It's an interesting read.

void * calloc(int n_obj, int size) returns an array.

void realloc(void *p, int size) expands the zone allocated to a variable to size. Returns NULL if unsuccessful.

void free(void *p) frees the zone pointed to by p.

2.13. DOS memory

The MS-DOS system has a much more complicated memory model than Unix. This complexity should be interpreted as a flaw: the simplicity of Unix is in no way synonymous with mediocrity, quite the contrary. DOS memory management is, moreover, highly dependent on the underlying hardware (Intel 8086), which it exploits to the hilt. The original architecture on which DOS was developed had a 16-bit address bus, allowing up to 64 kB of memory to be addressed. Later, addressing was based on a base whose value was contained in specific 16-bit registers. These registers generally correspond to stack (SS), data (DS) and code (CS) functions. The registers contain the most significant bits of a 20-bit address, which corresponds to a 1 MB interval.

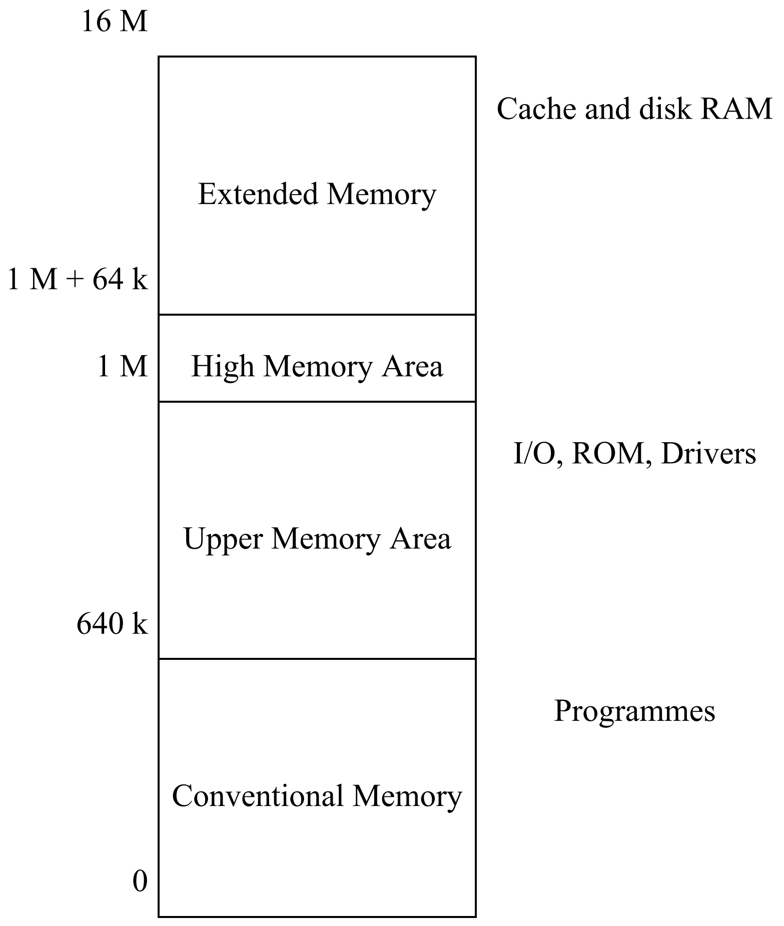


Figure 17

DOS addressing is in fact restricted to 640 kB, the upper part (Upper Memory Area) being reserved for various I/O control programs.

The DOS memory model allows addresses above 1 MB. Taking the value 0xFFFF as a base, it is possible to go up to 1 MB + 64 KB. This corresponds to the High Memory Area. The use of this memory depends on the (variable) way in which address line 21 is wired.

Current Intel processors allow extended addressing. The 286 allows 16 MB; the 386 and later, 4 GB. In order to be compatible with the 8086 processors, Intel has provided an operating mode that allows current machines to be used with the old memory management: the "real" mode. This mode limits the size to 640 kB. However, modifications to MS-DOS allow extended memory to be used for RAM disks, for example.

2.14 Windows memory

The first versions of Windows suffered greatly from the 8086 architecture and the legacy of DOS. However, Windows memory management is now easier with the Windows programming interface. With this API, the programmer sees memory as an area of flat addresses.

Windows automatically recompacts memory data using a mechanism known as *garbage collecting*. This is absolutely necessary because windowing requires multiple memory creations and destructions, and without this system, memory would very quickly be in a "thousand pieces".

After a recompaction operation, in the best of cases, the free blocks form a single large segment. In reality, compaction is often not complete, but it does make subsequent allocation easier. Recompaction is carried out regularly on the initiative of the operating system or to allocate memory that is lacking at a given moment. The recovery of free zones already existed in certain programming languages such as Prolog or Lisp and in operating systems such as the Macintosh.

Memory allocation uses pointers to pointers - *handles*. These handles (HGLOBAL) are stored in a memory segment: the BurgerMaster[5] . The handle that references the memory block will be constant, but the pointer to the actual memory will vary according to the operating system. To manipulate the data, the segment must be locked and the pointers used, which will then be constant.

HGLOBAL GlobalAlloc(UNIT fuFlags, DWORD cbBytes) allocates a segment of length cbBytes with the fuFlags options. Among these options GMEM_MOVEABLE indicates that the segment is relocatable, GMEM_FIXED indicates that the segment is fixed. If there is a failure, the function returns NULL and we can find out why using the GetLastError function.

LPVOID GlobalLock(HGLOBAL) locks a segment and returns a pointer. If it fails, the function returns NULL.

[5] It was the favourite restaurant of Windows developers, it seems.

BOOL GlobalUnlock(HGLOBAL) unlocks a segment. In fact, it decrements the lock counter on the segment. If this counter goes to zero, the value returned is FALSE, otherwise TRUE.

HGLOBAL GlobalFree(HGLOBAL) frees a segment. If successful, the function returns NULL.

2.15 Windows NT memory

Windows NT uses the same heap mechanism as Windows, but adds a faster one. A process is created with a particular heap and can create others of its own. The functions that manipulate heaps are as follows:

HANDLE GetProcessHeap(void) retrieves the application's default heap. It is in this heap that GlobalAlloc calls find free space.

HANDLE HeapCreate(DWORD flOptions, DWORD dwInitialSize, DWORD cbMaximumSize) creates a new private heap for the process executing it. The possible options are 0, HEAP_GENERATE_EXCEPTION and HEAP_NO_SERIALIZE. The other two parameters are usually 0.

LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, DWORD dwBytes) is used to allocate an area of memory in a heap. This zone is fixed and cannot be unloaded.

DWORD HeapSize(HANDLE hHeap, DWORD dwFlags, LPCVOID lpMem) returns the size of a block in a heap.

DWORD HeapReAlloc(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem, DWORD dwBytes) is used to modify the size of a memory zone in a heap.

BOOL HeapFree(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem) is used to destroy an area of memory in a heap.

BOOL HeapDestroy(HANDLE hHeap) is used to free an entire heap.