Operating Systems.
Memory Management

Contents I

Memory

- Hardware view: Electronic circuits to store and retrieve information.
- The *Bit* (**bi**nary elemen**t**)is the storage unit, the *byte* (8 bits) is the addressing unit.
- Although the byte is the address resolution unit, we'll consider *words*. The *Word* is the natural unit of data used by a particular processor design: the majority of the registers in a processor are usually word sized and the largest piece of data that can be transferred to and from the working memory in a single operation is a word.
  - Modern general purpose computers usually have a word size of 32 or 64 bits ...
- For historical reasons it is frequent to say word = 2bytes = 16bits, double-word = 4bytes = 32 bits, quad-word = 8bytes = 64bits

Memory access

- According to the way used to access the information contained in the its cells, memory can be classified in :
  - **Conventional memory**: given a memory address (a number), the memory system returns the data stored at that address
  - **Associative memory** Content-addressable memories (CAM): given an input search data (tag), the CAM searches its entire memory to see if that tag is stored anywhere in it. If the tag found, the CAM returns a list of one or more storage addresses where the tag was found and it can also return the complete contents of that storage address. Thus, a CAM is the hardware embodiment of what in software terms would be called an associative array or hash table. Used in cache memories.

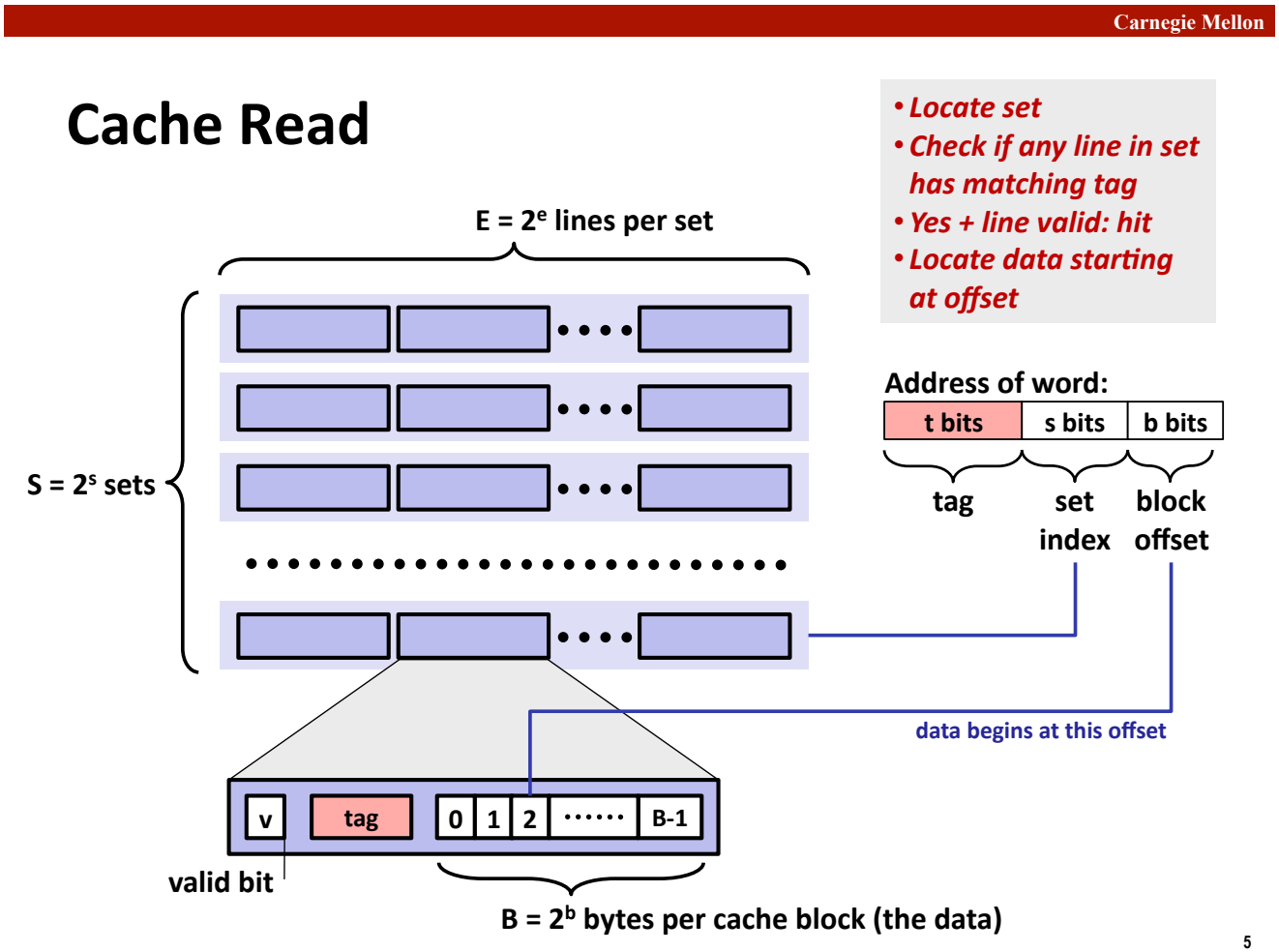
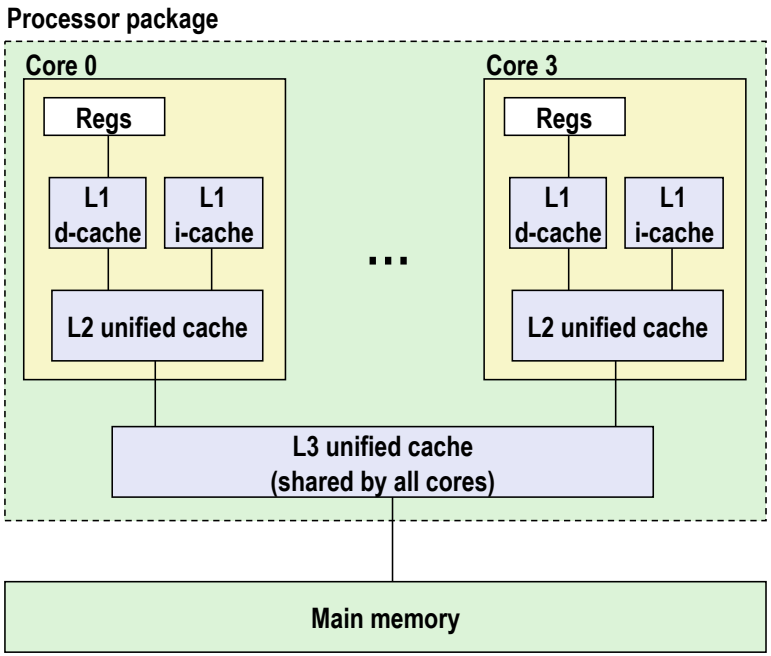


Figure: From R.E. Bryant et al. Computer Systems: A Programmer's Perspective (2nd edition), Pearson 2014

Carnegie Mellon

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches.

17

Figure: From R.E. Bryant et al. Computer Systems: A Programmer's Perspective (2nd edition), Pearson 2014

Memory hierarchy

- The memory access time is the time required to access instructions or data in memory (read and write operations),
  - the time elapsed between the moment an address is set on the address bus and the moment the data is stored in memory or made available to the CPU
- It is desirable to have fast memories (short access times) with large storage capacity. Unfortulately the faster and the larger memory is, the higher it's cost will be
- For this reason, faster and more expensive memories are used where memory accesses are more frequent.

Memory hierarchy

- These requirements led to the idea of Memory Hierarchy: memory is organised in layers according to access time and capacity
  1. Processor Registers
  2. Cache Memory
  3. Main Memory
  4. Hard Disk Drives
  5. Tape Drives and Optical Discs

Memory Hierarchy

Carnegie Mellon

Examples of Caching in the Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	64-bytes block	On-Chip L1	1	Hardware
L2 cache	64-bytes block	On/Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

63

Figure: From R.E. Bryant et al. Computer Systems: A Programmer's Perspective (2nd edition), Pearson 2014

- The Operating Systems is a resource manager, which implies:
  - The OS must keep the accounting of the resource memory (assigned and free memory blocks)
  - The OS must have a policy for memory allocation
  - The OS must allocate memory to the processes when they need it
  - The OS must release memory allocated to processes when they no longer need it

Memory management

- The OS must keep the accounting of the system memory
  - The OS has to know the amount of free memory: otherwise, this memory could not be assigned to processes
  - The OS also has to register the memory allocated to each individual process (via zones in the process tables)
- Whenever a process is created or whenever a process requests memory, the OS allocates memory to that process
- When a process terminates the OS releases its assigned memory
- The OS also manages the virtual memory system

Segments for a process virtual address space

- Code (text).
- Static Data. For global initialised var and static C vars. For uninitialised global vars (BSS).
- Heap. Dynamic memory (malloc).
- Stack. Stack frames of function calls: arguments and local var (automatic C vars), return addresses.

brk() System call

- Sets the end of the data segment, which is the end of the heap. Increasing the program **"break"** has the effect of allocating memory to the process; decreasing the break deallocates memory.
- *brk()* sets the end of the data segment to the addr specified as the argument, and returns 0 on success.
- *sbrk()* C function. Increments the program's data space by increment bytes. Calling *sbrk()* with an increment of 0 can be used to find the current location of the program **"break"**. On success, *sbrk()* returns the previous program break. (If the **"break"** was increased, then this value is a pointer to the start of the newly allocated memory)

C malloc() package

- Allows manual memory management for dynamic memory allocation via a group of library functions.
- The library functions are responsible for heap management instead of user programs.
- Package for explicit assignment and releasing memory vs. Garbage Collectors.

malloc() C library function

- *malloc()* allocates the requested bytes of memory and returns a pointer to it.
- *free(ptr)* releases the memory allocated with malloc().
- *calloc()* assigns memory for n elements of size bytes each, realloc resizes the block of allocated memory
- These functions invoke the syscalls *brk()* and *sbrk()* to manage the heap.
- the *mmap()* system call maps files into a process address space.

Memory management: example

- Compile and run this C program

```
/* this example comes from
http://www.enseignement.polytechnique.fr/informatique/INF583/ */

#include <stdlib.h>
#include <stdio.h>
double t[0x02000000];
void segments()
{
 static int s = 42;
 void *p = malloc(1024);
 printf("stack\t%010p\nbrk\t%010p\nheap\t%010p\n
 static(BSS)\t%010p\nstatic(initialized)\t%010p\ntext\t%010p\n",
  &p, sbrk(0), p, t, &s, segments);
}
int main(int argc, char *argv[])
{
  segments();
  exit(0);
}
```

Memory management: example

- Output (linux 64 bit system)

```
stack                   0x7fff12f59468
brk                     0x116ed000
heap                    0x116cc010
static(BSS)             0x00601060
static(initialized)     0x00601038
text                    0x004005d4
```

Memory management: example

- Output (solaris 64 bit system)

```
stack                   0xffff80ffbffff918
brk                     0x10415360
heap                    0x10411370
static(BSS)             0x00411360
static(initialized)     0x004112f8
text                    0x00400ef8
```

Memory management: example

- Output (solaris 32 bit system)

```
stack                   0xfeffea3c
brk                     0x18063060
heap                    0x18061068
static(BSS)             0x08061060
static(initialized)     0x0806101c
text                    0x08050d50
```

Memory management: example

- Compile and run this C program

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <limits.h>

#define TROZO  100*1024*1024
#define PUNTO  (10*1024*1024)

void accede (char * p, unsigned long long tam)
{
  unsigned long long i;
  for (i=0; i< tam; i++){
     p[i]='a';
     if ((i%PUNTO)==0)
        write (1,".",1);  /*imprime un punto cada 10 Mbytes accedidos*/
  }
}
```

Memory management: example

```
main (int argc,char*argv[])
{
  char *p;
  unsigned long long total=0, cantidad=TROZO;
  unsigned long long maximo=ULLONG_MAX;

  if (argv[1]!=NULL){
        maximo=strtoull(argv[1],NULL,10);
        if (argv[2]!=NULL)
            cantidad=strtoull(argv[2],NULL,10);
  }
  while (total<maximo && (p=malloc(cantidad))!=NULL){
        total+=cantidad;
        printf ("asignados %llu (total:%llu) bytes en %p\n", cantidad,total,p);
        accede (p,cantidad);
getchar();

  }
  printf ("Total asignacion: %llu\n",total);
 sleep(10);
}
```

Output

- output of command pmap PID

```
3742:   ./a.out
0000000000400000      4K r-x--  /home/barreiro/teaching/teaching-so/examples_C
0000000000600000      4K r----  /home/barreiro/teaching/teaching-so/examples_C
0000000000601000      4K rw---  /home/barreiro/teaching/teaching-so/examples_C
00007ff7b22c9000 307212K rw---    [ anon ]
00007ff7c4ecc000   1588K r-x--  /lib/x86_64-linux-gnu/libc-2.13.so
00007ff7c5059000   2048K -----  /lib/x86_64-linux-gnu/libc-2.13.so
00007ff7c5259000     16K r----  /lib/x86_64-linux-gnu/libc-2.13.so
00007ff7c525d000      4K rw---  /lib/x86_64-linux-gnu/libc-2.13.so
00007ff7c525e000     24K rw---    [ anon ]
00007ff7c5264000    132K r-x--  /lib/x86_64-linux-gnu/ld-2.13.so
00007ff7c545f000     12K rw---    [ anon ]
00007ff7c5480000     16K rw---    [ anon ]
00007ff7c5484000      4K r----  /lib/x86_64-linux-gnu/ld-2.13.so
00007ff7c5485000      8K rw---  /lib/x86_64-linux-gnu/ld-2.13.so
00007fff4562e000    132K rw---    [ stack ]
00007fff456e6000      4K r-x--    [ anon ]
ffffffffff600000      4K r-x--    [ anon ]
 total           311216K
```

Output

- after another assignment

```
3742:   ./a.out
0000000000400000      4K r-x--  /home/barreiro/teaching/teaching-so/examples_C
0000000000600000      4K r----  /home/barreiro/teaching/teaching-so/examples_C
0000000000601000      4K rw---  /home/barreiro/teaching/teaching-so/examples_C
00007ff7abec8000 409616K rw---    [ anon ]
00007ff7c4ecc000   1588K r-x--  /lib/x86_64-linux-gnu/libc-2.13.so
00007ff7c5059000   2048K -----  /lib/x86_64-linux-gnu/libc-2.13.so
00007ff7c5259000     16K r----  /lib/x86_64-linux-gnu/libc-2.13.so
00007ff7c525d000      4K rw---  /lib/x86_64-linux-gnu/libc-2.13.so
00007ff7c525e000     24K rw---    [ anon ]
00007ff7c5264000    132K r-x--  /lib/x86_64-linux-gnu/ld-2.13.so
00007ff7c545f000     12K rw---    [ anon ]
00007ff7c5480000     16K rw---    [ anon ]
00007ff7c5484000      4K r----  /lib/x86_64-linux-gnu/ld-2.13.so
00007ff7c5485000      8K rw---  /lib/x86_64-linux-gnu/ld-2.13.so
00007fff4562e000    132K rw---    [ stack ]
00007fff456e6000      4K r-x--    [ anon ]
ffffffffff600000      4K r-x--    [ anon ]
 total           413620K
```

Memory fragmentation

- File systems and memory can show internal and external fragmentation
  - **Internal fragmentation**: Wasted memory because assignation is made in blocks of n bytes and the requests of processes are not an exact multiple of n.
  - **External fragmentation**: Wasted memory that can not be assigned because it is not contiguous. External fragmentation appears in systems with (pure) segmentation..

Swapping

- (The *Swap*) area is a part of the disk used as auxiliar memory
- A running process needs to be in memory. Swapping can increase the multiprogramming level,
  - If a process in the swap zone is selected by the scheduler, the process needs to be loaded in memory, which increases the context switch time.
  - To swap processes that are waiting for I/O to be completed (pending I/O), the OS must transfer I/O to the system buffers in kernel space and then to the I/O device. This also adds overhead.
  - For these reasons, modern Operating Systems usually swap pages and rarely swap whole processes.

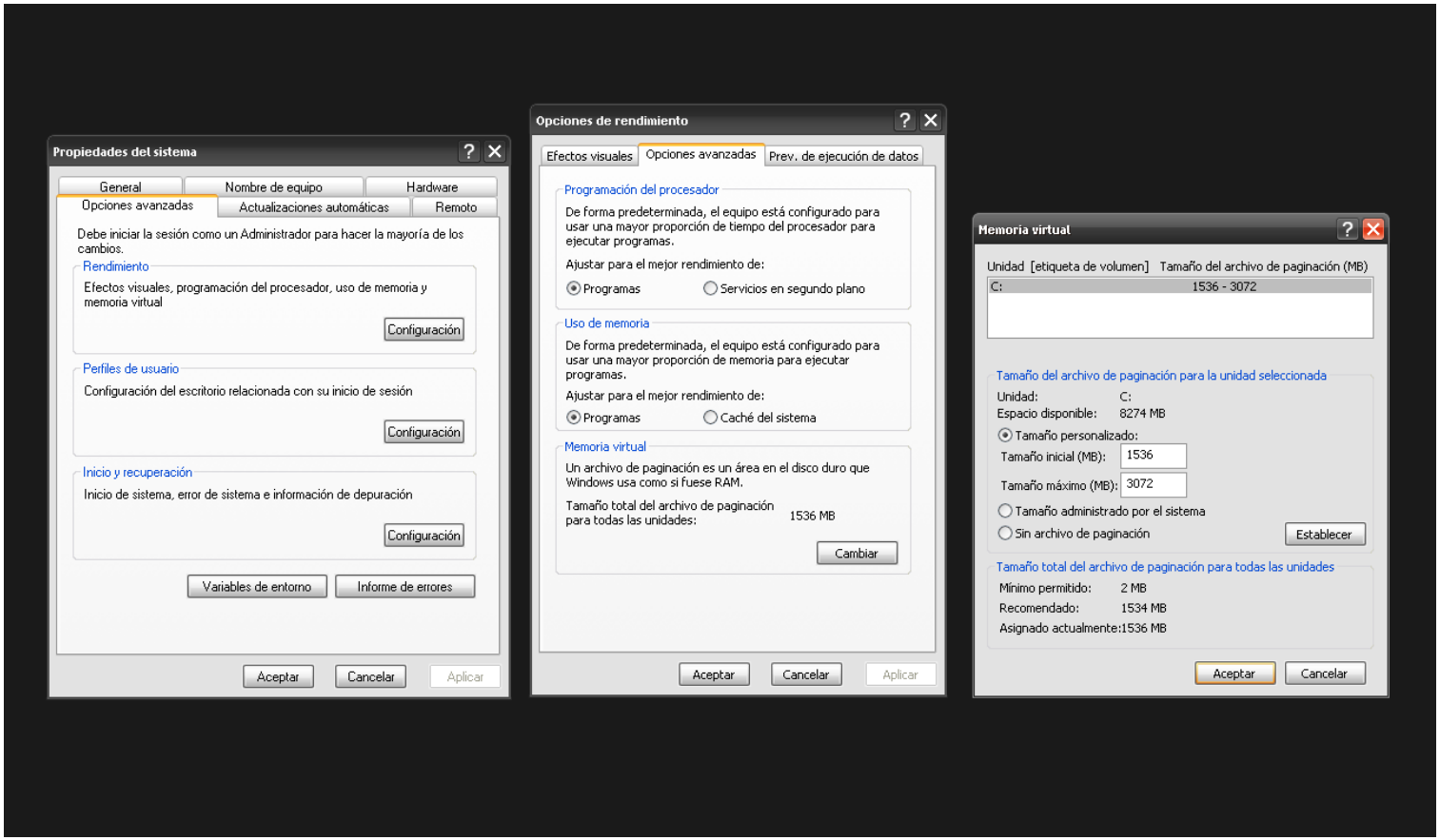
Swapping

- Swapping of whole processes in old systems to increase the multiprogramming level.
  - *swapping*
- Modern systems with virtual memory swap the less referenced pages.
  - *paging*
- The illusion of infinite memory in virtual memory systems

Swapping

- The swapping area can be a dedicated partition or a file in the disk.
- A swapping file is a more flexible solution, its location and size can be changed easily.
- But accessing a swapping file is less efficient because it uses the indirections of the file system to access the data.
- MS Windows systems use a swap file, while Unix and Linux systems use swap partitions, although swap files can be configured for these systems.

Swap file in MS Windows



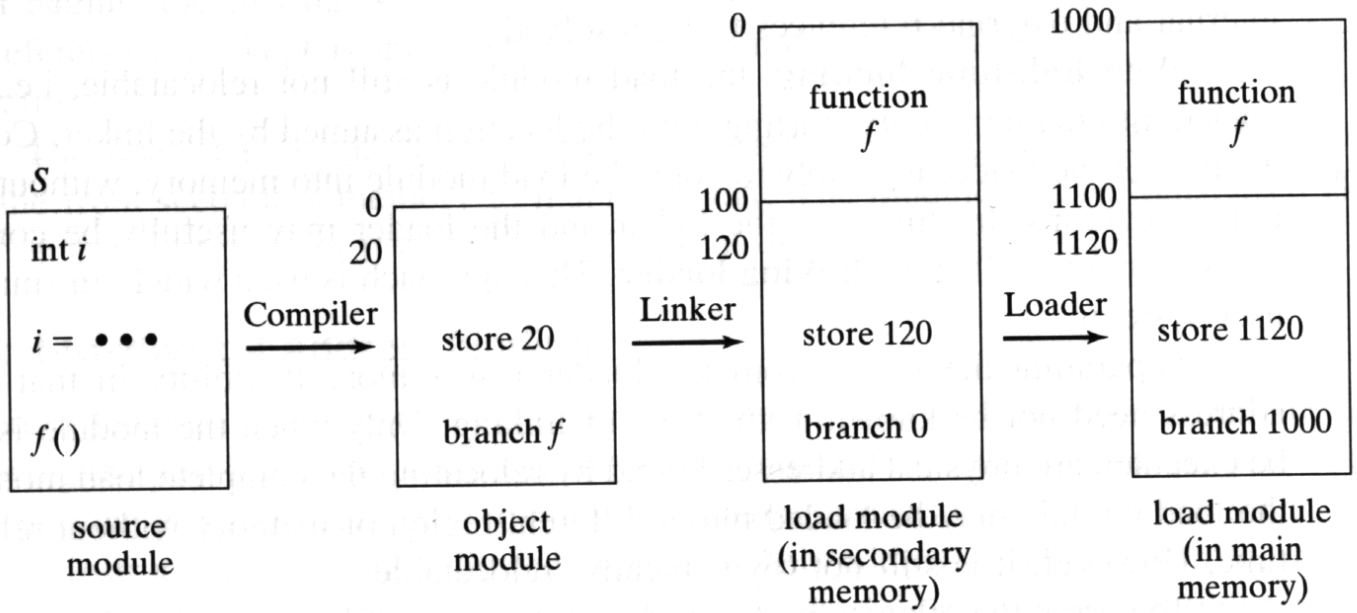
Memory management: relocation

- We start with source code $->$ (compilation) $->$ object code
- Several object code files $->$ (linking) $->$ executable file
- Executabe file $->$ (load and execution) $->$ process in memory
- Source code $->$ executable file $->$ process in memory
- In the source code there are variables, functions, procedures, ...
- In the process in memory there are contents of memory addresses, jumps to addresses that contain code
- Where and when these transformations are done?

Memory management: relocation

- **Absolute code**: Addresses are obtained at compilation (and/or linking) time (example: MS-DOS .COM files)
  - At compilation/linking time it is necessary to know the addresses for execution of the program
  - Lack of portability of the executable file. It can not run in other memory locations.
- **Static relocation**: Addresses are obtained when the program is loaded in memory (the executable file contains relative references) (example: MS-DOS EXE files)
  - After loaded in memory, the program can not be moved to other memory location
  - Swapping is possible only if processes return to the same memory positions they used before being swapped out (fixed partitions)

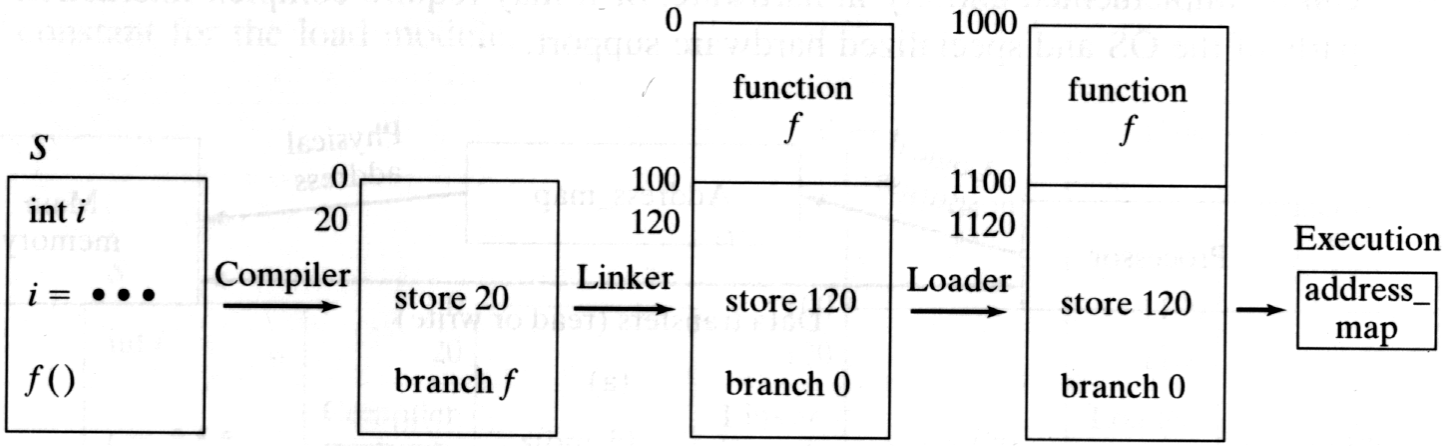
Static relocation



Memory management: relocation

- **Dynamic relocation**: Addresses are obtained at execution time. The running process uses memory references which are not the references to physical memory positions. (example: MS XP Windows EXE files)
  - No restrictions to swapping. Swapped processes can be swapped in memory in any memory location.
  - Distinction between *Virtual or Logical address space* and *Physical address space*.
  - By reasons of efficiency it is necessary hardware that translates logical addresses in physical addresses.
- Modern systems use dynamic relocation
- With dynamic relocation, linking can be postponed to execution time. Dynamic linking (MS Windows DLLs, lib*.so in linux).

Dynamic Relocation



Protection

- Memory must be protected
  - A process can not directly access the OS memory
  - A process can not access memory of other processes
- Simplest hardware to support protection
  - Two limit registers
  - One base (relocation) register and one limit register

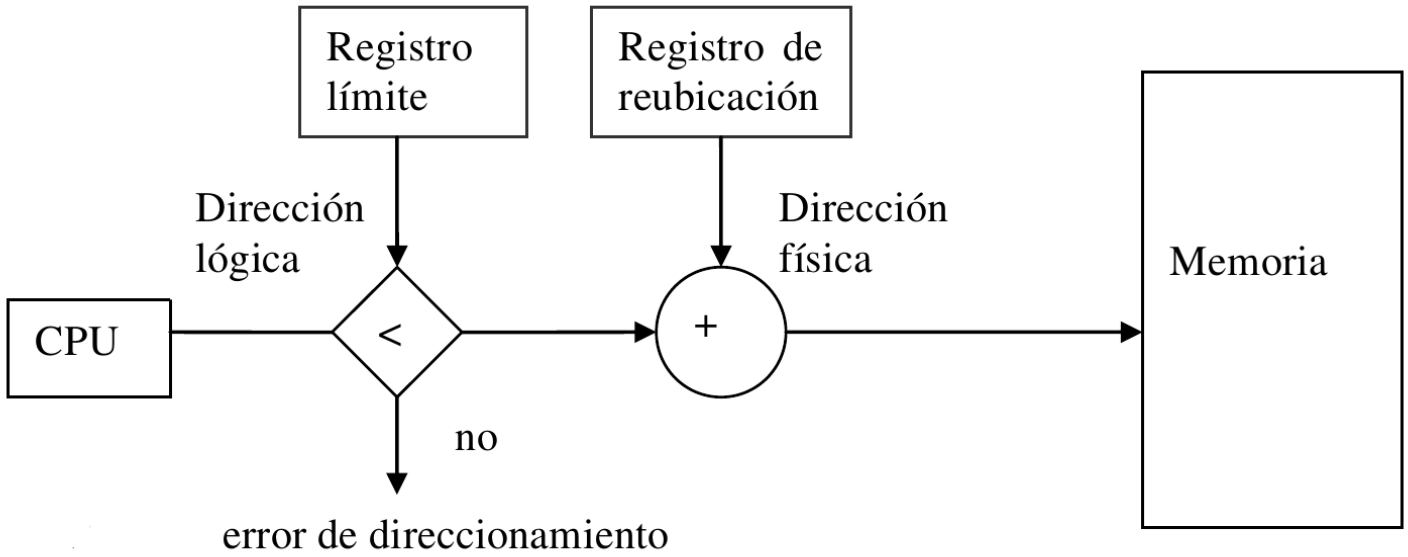
Protection

- Two limit registers
  - Every address generated by a running process must be in the range of the values stored in the limit registers. Otherwise an exception is produced.
  - The hardware provides these limit registers.
  - The values of the registers are updated in a context swicht and stored in the Process Control Block.
  - Changing the values of these registers is a privileged instruction (kernel mode)

Protection and relocation

- With base (relocation) and limit register
  - Base register contains value of smallest physical address. Limit register contains range of logical addresses. Each logical address must be less than the limit register (otherwise, an exception is produced), which is added to the address contained in the base register.
  - The hardware provides these base and limit registers.
  - The values of the registers are updated in a context swicht and stored in the Process Control Block.
  - Changing the values of these registers is a privileged instruction (kernel mode)
  - This hardware supports protection and dynamic relocation.

Base and limit registers



Protection and relocation

- With base (relocation) and limit register
  - The program has the illusion of running on a dedicated machine, with memory starting at address zero.
  - Segmented system with only one segment.

Protection

- In modern systems memory protection is supported by the addressing mechanisms.
- Segmentation and paging provide effective memory protection and relocation.
- It is necessary at least two execution modes: user mode and kernel or system mode.

Simple schemes

No multiprogramming systems

Multiprogramming Systems

Simple schemes: No multiprogramming systems

- In Operating Systems without multiprogramming there were only two memory areas: one for the OS and one for the user process.
- Typically the OS in the low positions of memory and the rest for the user process. Concept of simple monitor (IBSYS for IBM 7094)
- First generation of personal computers: OS in high addresses of memory (in ROM) and the rest for user processes
- OS in low positions of memory but with some parts of the OS in the high positions: First versions of MS-DOS

Simple schemes

No multiprogramming systems

Multiprogramming Systems

Memory management: Simple schemes

- For multiprogrammed OS the simplest scheme is to split the memory in partitions with a process in each partition.
- Two alternatives
  - Fixed size partitions: Allows for a fixed number of processes in memory
  - Variable size partitions: The number and size of partitions can vary

Memory management: Simple schemes

- Fixed size partitions
  - Internal and external fragmentation
  - Used in IBM OS/360 MFT (Multiprogramming with a Fixed number of Tasks)
- Variable size partitions
  - Negligible *internal fragmentation*, but also suffers from *external fragmentation*
  - Compactation of memory to solve external fragmentation. Very high cost.
  - Used in IBM OS/360 MVT (Multiprogramming with a Variable number of Tasks)

Memory management: Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as: main program, procedure, function, method, object, local variables, global variables, common block, stack, symbol table, arrays, etc

Memory management: Segmentation

- The address space of a process has variable size blocks called *segments*
- A logical address is composed of a segment (or *segment number*) and an *offset* inside the segment, <segment-number, offset>
- The segment number is the entry number in the Segment Table for that process. Each entry of the Segment Table contains the Base Address, i.e. the starting physical address for the associated segment, and the segment size (Limit).
  - Segment-table base register (STBR) points to the segment table's location in memory
  - Segment-table length register (STLR) indicates number of segments used by a program
  - segment number s is legal if $s < STLR$

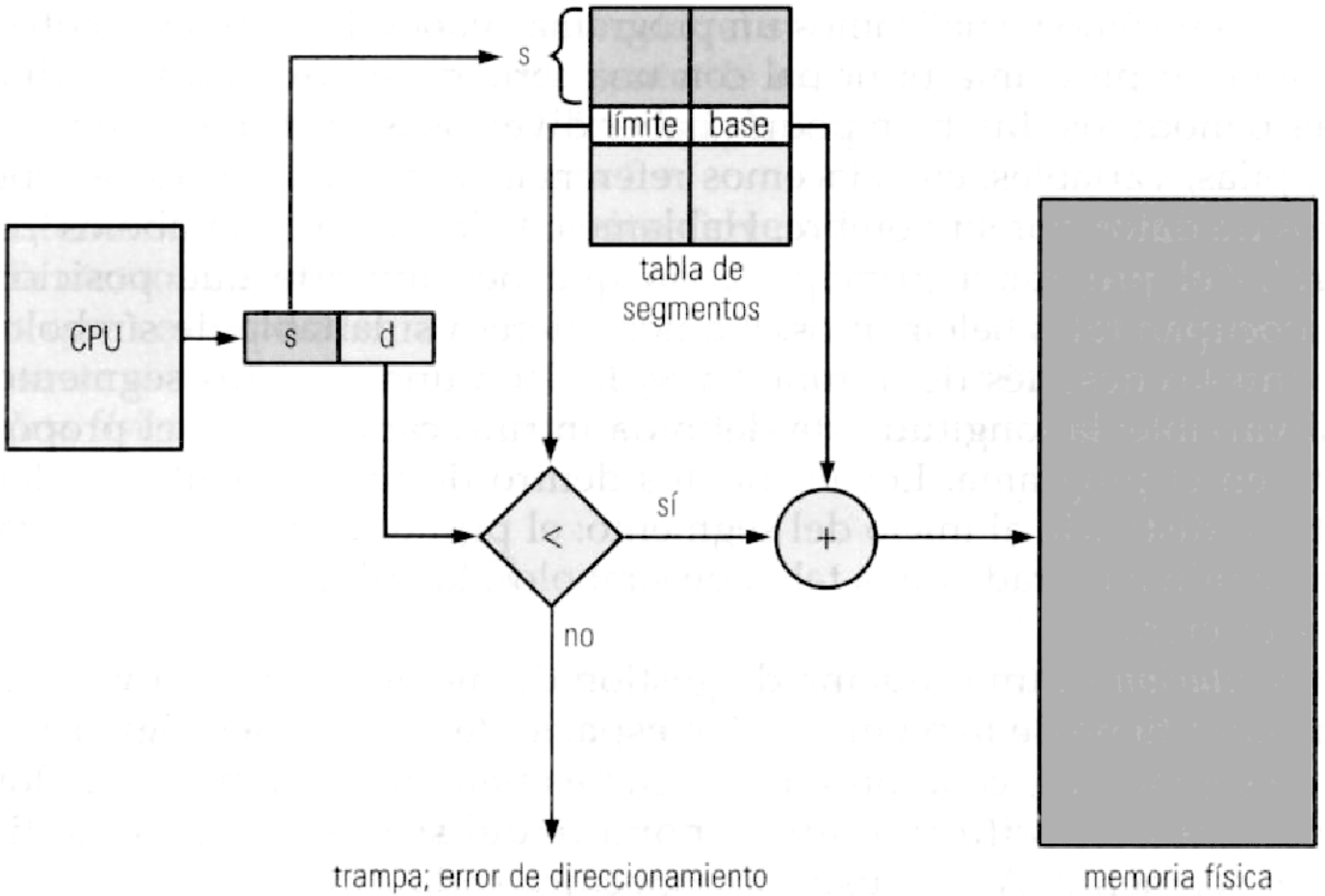
Memory management: Segmentation

- Changing the values of these registers in a context switch is a privileged instruction. The values are stored in the Process Control Block.
- segment number s is legal if $s < STLR$
- logical address: <segment-number, offset>
- physical address: Base Address + offset
- if the offset is less than the limit, the physical address is obtained adding the offset to the Base Address, otherwise an addressing error is produced and a trap to the OS

Memory management: Segmentation

- Protection, with each entry in the segment table associate:
  - validation bit (legal/illegal segment)
  - read/write/execute privileges
  - kernel/user mode accesible segment
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

Memory management: Segmentation



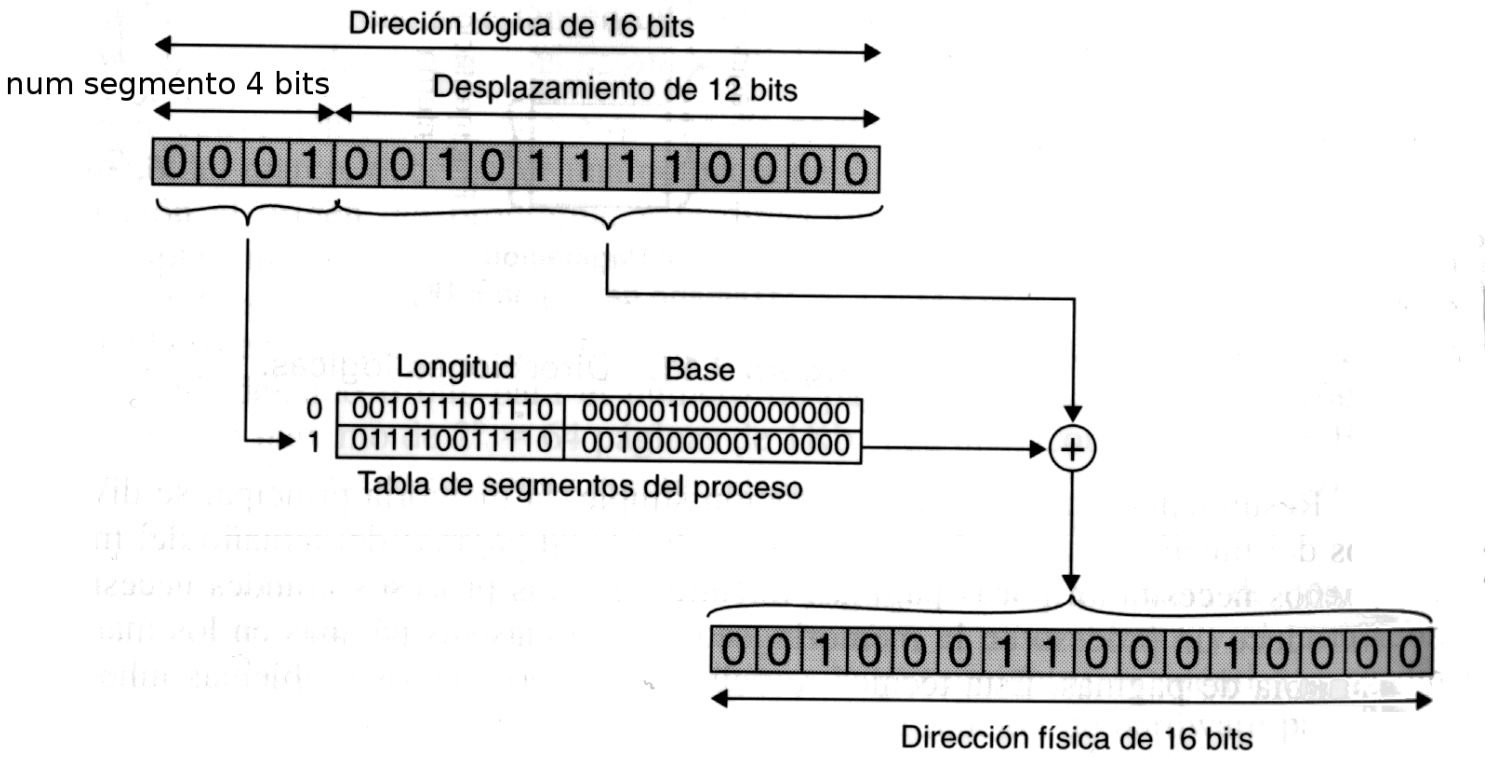
Segmentation: example

- Let us consider a system with segmentation with the following properties:
  - logical addresses of 16 bits (4 bits for the segment number, 12 bits for the offset)
  - each entry of the segment table has 28 bits, the 12 most significant for the limit and the 16 least bits for the base address
  - A process has 2 segments and the first two entries in the segment table of the process contain the values 0x2EE0400 and 0x79E2020 respectively
    - What physical address corresponds to a reference to logical address 0x12F0?
    - What physical address corresponds to a reference to logical address 0x0342?
    - What physical address corresponds to a reference to logical address 0x021F?
    - What physical address corresponds to a reference to logical address 0x190A?

Memory management: Segmentation example

- logical address 0x12F0, physical address 0x2310
- a reference to the logical address 0x0342 causes an addressing error
- logical address 0x021F, physical address 0x061F
- a reference to the logical address 0x190A causes an addressing error

Memory management: Segmentation



Fragmentation in segmentation systems

- *Internal fragmentation*.
  - The segment size is a multiple of a fixed number of bytes (for example 16 bytes), therefore allocated memory may be slightly larger than requested memory; this size difference is memory internal to the segment, but not being used
- *External fragmentation*.
  - Segments are variable size memory blocks. After assigning and releasing memory, holes (blocks of available memory) of various sizes are scattered through memory.
  - External fragmentation: total memory space exists to satisfy a specific request, but it is not contiguous. Compactation solves the problem at the expense of computional cost.

Segmentation: dynamic storage-allocation problem

- The OS accounts for both the assigned and free memory. After releasing a block of memory, adjacent free blocks are collapsed into a larger free block.
- How can the OS satisfy a request of size n from a list of free holes?
  - **first fit** Allocate the first hole that is big enough. Fast. Small holes appear in low areas of memory and large holes in high areas, assuming the search for holes start in the low areas.
  - **next fit** Allocate the next hole large enough, searching from the last allocated block
  - **best fit** Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
  - **worst fit** Allocate the largest hole; must also search entire list. Produces the largest leftover hole
- First-fit and best-fit are found to operate better than worst-fit in terms of speed and storage utilization

Segmentation: amount of memory in holes

- In a system with segmentation, given s the average size of a segment and $k = \frac{average\_size\_of\_a\_hole}{average\_size\_of\_a\_segment}$

$$\frac{memory\_in\_holes}{total\_memory} = \frac{k}{k+2}$$

- for n segments, the amount of memory in segments is ns.
- two adjacents holes collapse into a single hole, therefore there are double number of segments than holes. For $\frac{n}{2}$ holes the amount of memory in holes is $\frac{n}{2}ks$
- therefore the rate is

$$\frac{memory\_in\_holes}{total\_memory} = \frac{\frac{n}{2}ks}{\frac{n}{2}ks+ns} = \frac{k}{k+2}$$

Memory management: Segmentation

- *Hardware* support is needed
- It enables dynamic relocation
- It enables memory protection
- Sharing data or code segments is possible

Memory management: Segmentation

- Intel 8086, 4 segments with a segment register for each segment (code CS, data DS, stack SS, extra ES), no segment tables in memory. No memory protection: any program could access any memory area. (16bits segment, 16 bits offset). 1 Megabyte (20 bits physical address space)
- Segmentation in Intel 80286: 1 Gigabyte addressable virtual memory in segments up to 64K (16 bits segment, 16 bit offset). 16 megabytes physically addressable (physical addresses). Segment table for a process in memory, pointed by the contents of a processor register. MS Windows 3.1 in standard mode used this segmentation mechanism. Windows 3.1 Enhanced model to operate with i386 processor.
- Intel 386, Paged Segmentation (next chapter)

Memory management: paging

- Physical address space of a process can be noncontiguous; process can be allocated physical memory wherever available
- Avoids external fragmentation, still has internal fragmentation
- Avoids problem of varying sized memory chunks
- Enables dynamic relocation, protection and sharing of memory

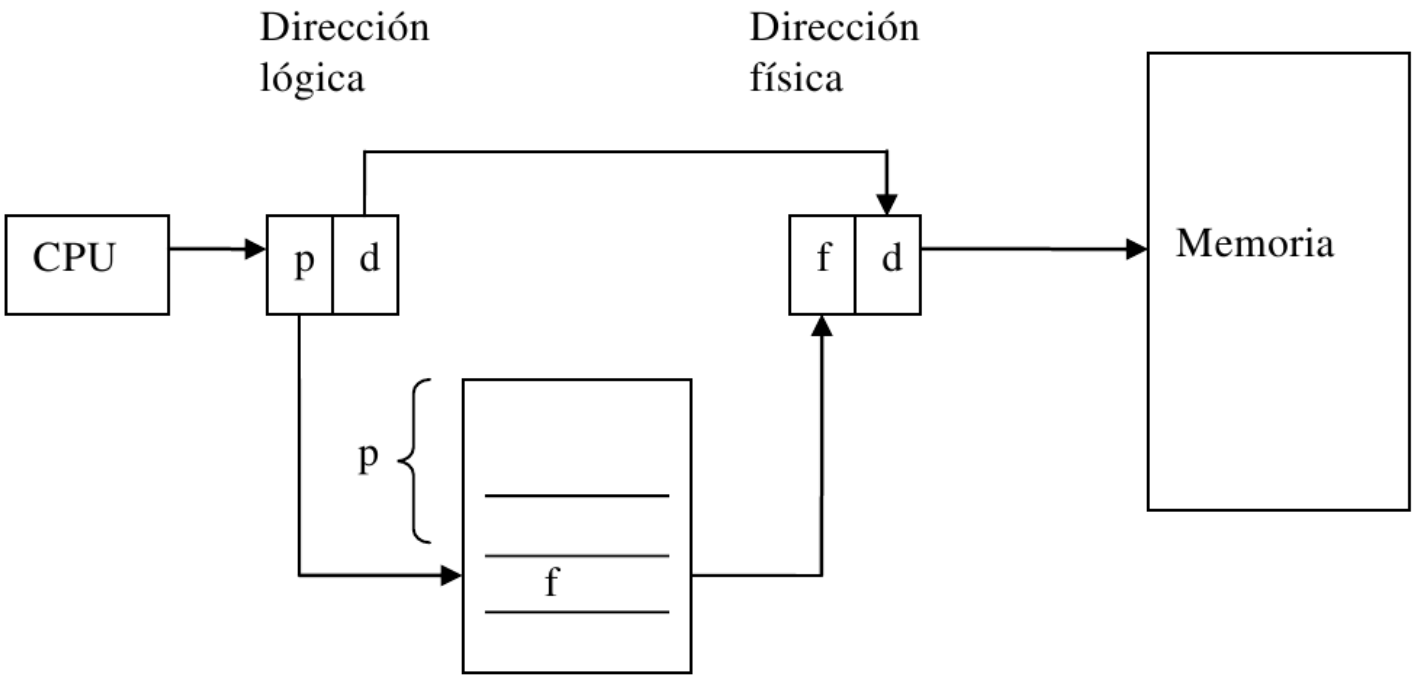
Memory management: paging

- Divide physical memory into fixed-sized blocks called *frames*. Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called pages
- The OS keeps track of all free frames. To run a program of size N pages, the OS needs to find N free frames and load program in them
- The OS sets up a page table to translate logical to physical addresses. A processor register (Page Table Base Register) contains the base address of the page table for a process. Changing the value of this register in a context switch is a privileged instruction (kernel mode). The value is stored in the Process Control Block, and loaded again when the process is scheduled to run

Memory management: paging

- Address generated by CPU is divided into:
- Page number (p) used as an index into a page table which contains base address of each page in physical memory
- Page offset (d) combined with base address to define the physical memory address that is if fact accesses

Memory management: paging



Paging: Example

- Consider a system with 16 bits logical addresses, 7 most significant bits for the page number and the 9 least significant bits for the offset in the page.
- Page size is 512 bytes $= 2^9$
- A process makes a reference to a memory address 0x095f (0000 1001 0101 1111)
- This is a reference to page number 4 and page offset 0x15f

Paging: Example

- In the entry for page 4 in the page table, we can get the physical address for this page
- Let us assume that the Base Address of the Physical Page is 0xAE00 (1010 1110 0000 0000)
- Therefore the final physical address is es 0xAF5F (1010 11111 0101 1111)

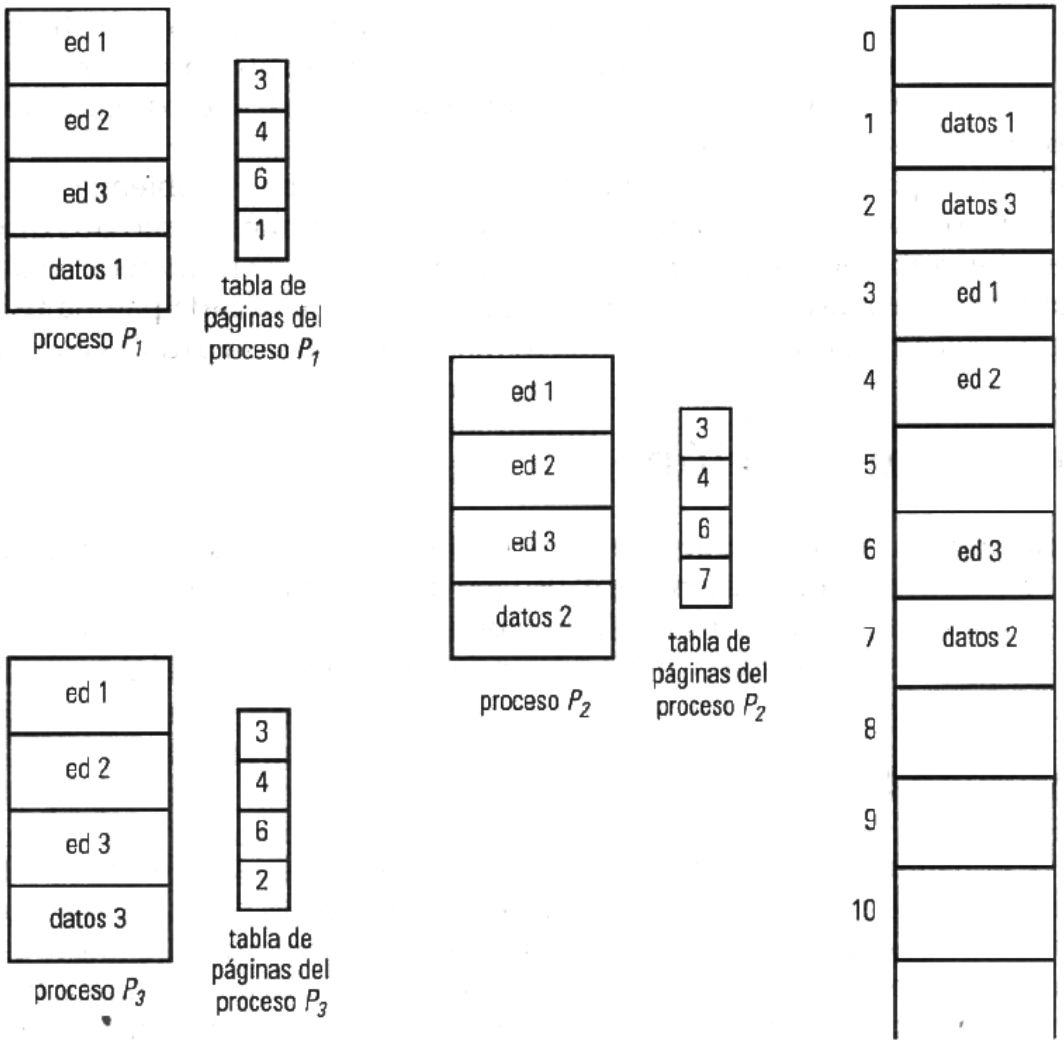
Paging

- With paging the memory for a process need not be contigous
- The page size is defined by the hardaware, for example:
  - The Intel x86 32 bits architecture has a page size of 4 Kbytes
  - The Sparc architecture has a page size of 8 Kbytes
- No external fragmentation
- Internal fragmentation
- The larger the page sizes the larger the internal fragmentation will be
- Smaller page sizes mean more entries in the page tables for processess, so more memory is lost in page tables

Paging

- The OS must account for: the free physical pages (frames), the frames assigned to processes
- The OS manages the table pages of different process in the context switch
- Processes' view of the memory and physical memory are now very different
- Memory Protection: because of the way paging is implemented, a process can only access its own memory following its page table, and its Page Table can only be changed by the OS
- Paging allows memory sharing among processes: the associated entries in their page tables point to the same physical pages

Memory management: paging



Paging

- Hardware support is needed (for example no paging on intel 286, paging available on i386)
- Each entry of the PT has other information beside the address of the physical page: valid bit (1 for a valid page number, i.e., the page exists for this process) , access bit (1 page was accessed, but can be 0 for a accessed page after a bit reset), dirty (modified page) bit, read only, read write, privilege level (kernel only, user), etc.

Paging: Implementation of the Page Table

- **Dedicated registers:** The processor has registers to store the page table of the running processes. I
  - In a context switch the page table is stored in the Process Control Block, and the registers are loaded with the page table of the new running process.
  - Address translation is fast because the page table is in the processor registers
  - High cost because many processor registers are needed, for this reason it is not used in modern systems
  - Slow context switch because many registers are implied

Paging: Implementation of the Page Table

- **In memory:** Page tables are kept in main memory.
  - The Page-table base register (PTBR) points to the page table of the running process and the Page-table length register (PTLR) indicates the size of the page table. In a context switch the page table is stored in the Process Control Block, and the registers are loaded with the page table base address and the value of the size of the page table of the new running process.
  - Address translation is slow because every data/instruction access requires two memory accesses: one for the page table and another for the data/instruction
  - Fast context switch because only two registers are implied
  - The two memory access problem can be solved by the use of a special fast-lookup hardware cache (associative memory) called look-aside buffers (TLBs)

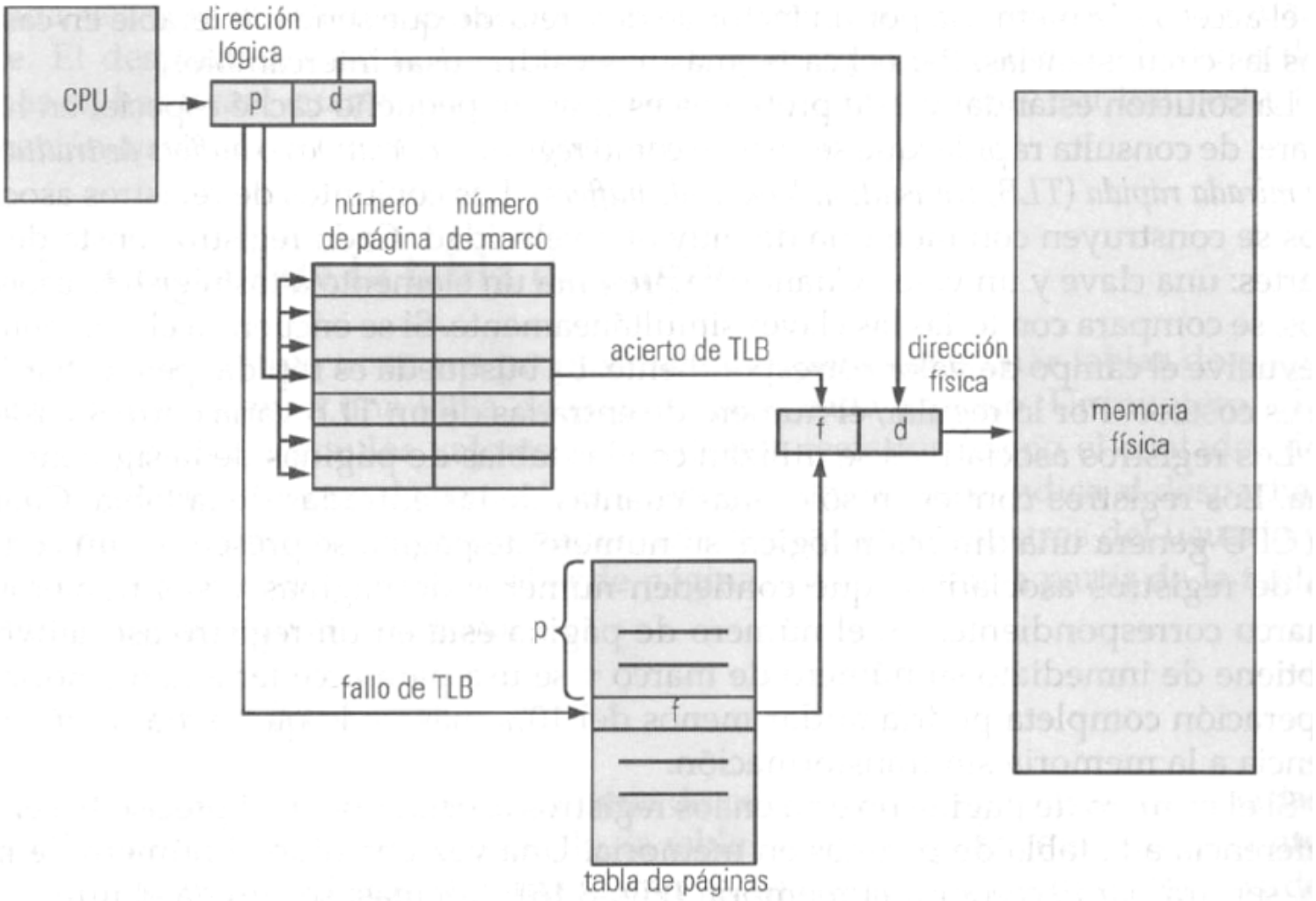
Paging: Implementation of the Page Table

- The TLB is an associative memory that performs a parallel search, i.e., the TLB contains pairs $< numberoflogicalpage, framenumber >$.
- Given a logical address (p, d), If p is in associative register contained in the TLB, get frame number out, otherwise get frame number from page table in memory

Paging: Implementation of the Page Table

- Some TLBs store address-space identifiers (ASIDs) in each TLB entry. This uniquely identifies each process to provide address-space protection for that process. Otherwise theres a need to flush it at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be wired down for permanent fast access

Memory management: Paging with TLB



Paging: Effective access time with TLBs

- Let's consider that TLB Associative Lookup time is ϵ (time unit). Can be $< 10\%$ of memory access time
- We define the hit ratio (α) as the percentage of times that a page number is found in the associative registers. The bigger the number of associative registers the highest the hit ratio will be
- The Effective Access Time for a memory with access time T is
  - $EAT = \alpha(T + \epsilon) + (1 - \alpha)(2T + \epsilon)$
- Consider the following example $\alpha = 80\%$, $\epsilon = 20ns$ for TLB search, 100ns for memory access
  - $EAT = 0.80x120 + 0.20x220 = 140ns$ (instead of 200 ns without TLB)
- The more realistic example where $\alpha = 99\%$, $\epsilon = 20ns$ for TLB search, 100ns for memory access, yields
  - $EAT = 0.99x120 + 0.01x220 = 121ns$ (instead of 200 ns without TLB)

Inverted page tables

- in systems with virtual memory, there are many pages of the processes that are not loaded in physical memory; however the page tables of each process have to be large enough to accommodate the whole address space of the process
  - A 32 bits address space using 4K pages would need 2^{20} pages. The table page for a process would have to have 2^{20} entries. Assuming 4 bytes entries, each page table would need 4 Mb
- Two solutions
  - **multilevel paging** (mixed systems)
  - **inverted page tables**
    - one entry for each physical frame
    - each entry has information about the page contained in that frame (proccess and logical address)
    - To optimice the search time usually a *hash* function is used
    - Used in the PowerPC and IBM AS/400 architectures

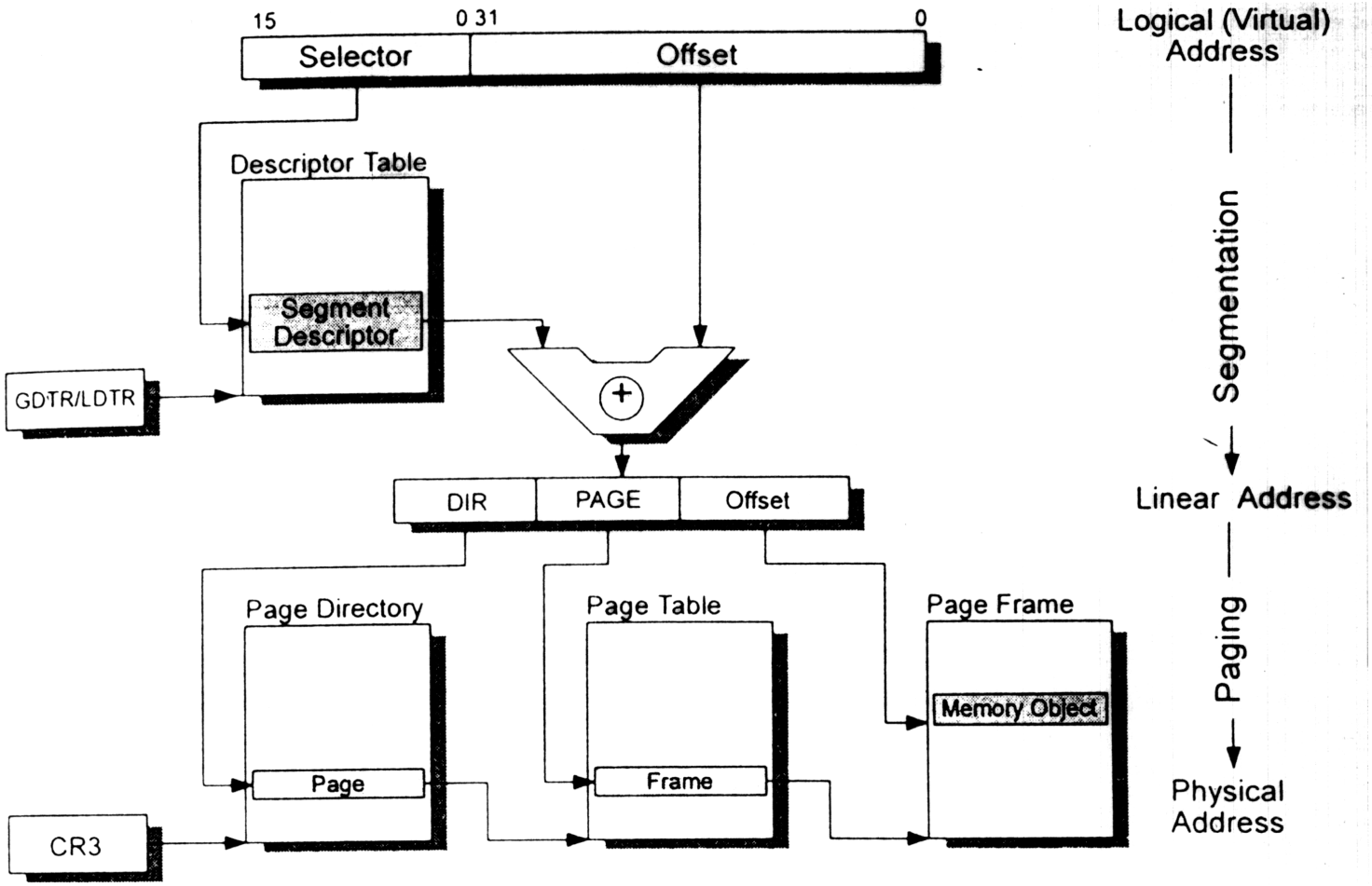
Mixed system

- A combination of paging and/or segmentation
- As paging provides better memory usage, there's always paging as the last managing system
  - **paged segmentation**: ancient systems, used in IBM system 370, the page table of a proccess was segmented
  - **multilevel paging**
  - **paged segmentation** the segments are paged
- As an example we'll discuss the x86 32bits archictecture: two level paged segmentation

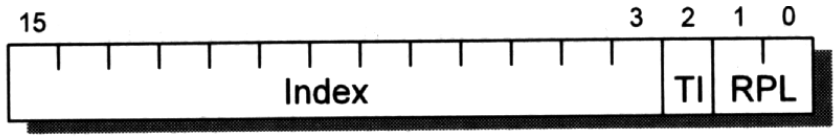
Memory management in the 32 bit PC architecture

- As an example of *actual* memory management scheme we'll look at the 32 bit PC architecture: segmentation with two levels of paging
- Each addres is comprised of
  - **selector** (16 bits) 13 bits for the segment number, 1 bit to select table(GDT-Global Decriptor Table or LDT-Local Descriptor Table) and 2 bits for the privilege level
  - **offset** 32 bits
- The 13 bits of the segment number (*selector*) serve as an index in a segment table (*descriptor table*) where we get, among other things,
  - a 32 bits base address
  - 20 bits limit (which represents a 32 bit addressing if page granularity is selected)

Memory management in the 32 bit PC architecture

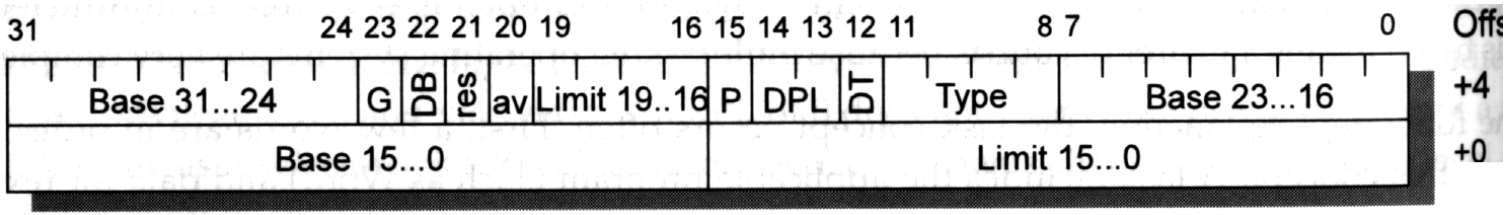


32 bit PC: Segment selector



Index: Index into GDT or LDT
TI: Table Indicator
0=GDT 1=LDT
RPL: Requested Privilege Level
00=0 01=1 10=2 11=3

32 bit PC: Segment selector



Base 31...24, Base 23...16, Base 15...0: Segment Base
Limit 19...16, Limit 15...0: Segment Limit

G: Granularity
0=Byte Granularity 1=Page Granularity

DB: Segment Size (Default/Big)
0=16-bit Operands/Addresses (Default) 1=32-bit Operands/Addresses (Big)

r: Reserved

av: Available for Operating System

P: Segment Present
0=Segment Not in Memory 1=Segment in Memory

DPL: Descriptor Privilege Level
00=0 01=1 10=2 11=3

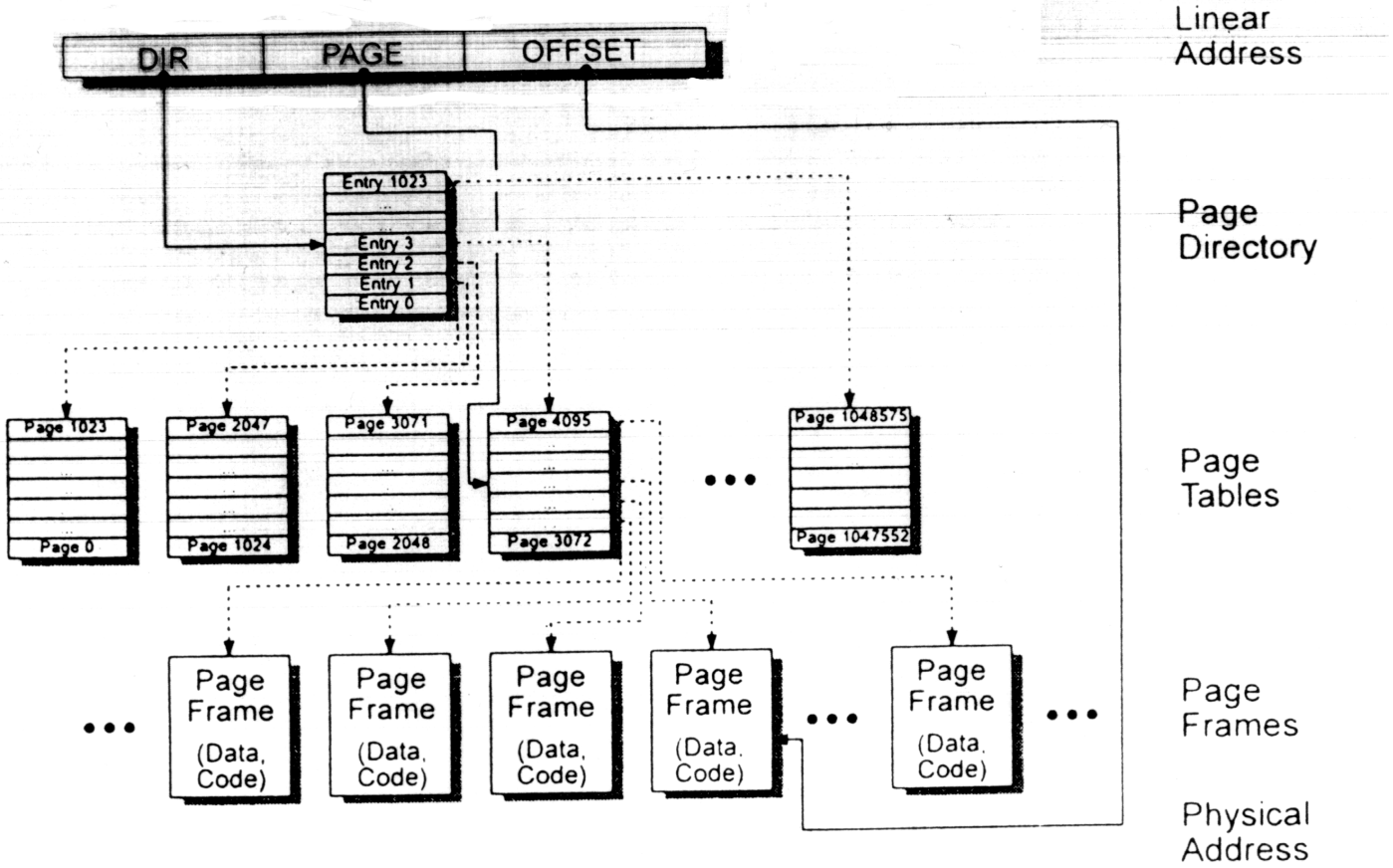
DT: Descriptor Type
0=System Segment 1=Application Segment

Typ: Type of System or Application Segment

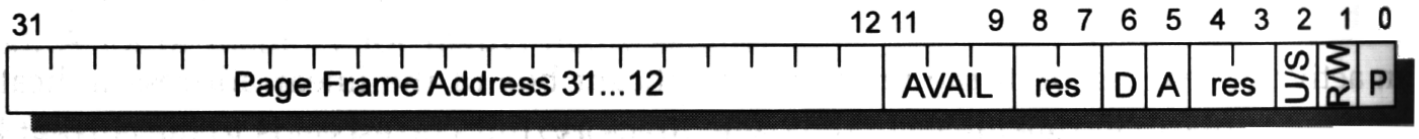
Memory management in the 32 bit PC architecture

- the 32 bit base address is added to the 32 bit offset yielding a 32 bit *linear address*
  - The first 10 bits correspond to an entry in the *page directory* table, where we get, among othe things, the address of a page table
  - The next 10 bits represent an index in the page table we got in the previous step, where we get the physical address of a page frame
  - The next 12 bits represent an offset in the aforementioned page frame
- The pages are sized 4K
- Each page table has 1024 four bytes entries (its format can be seen in one of the following figures). Each page table occupies one page

32 bit PC: linear address



32 bit PC: page table entry



Page Frame Address: address bits 31...12 of the page frame
AVAIL: available for the operating system
D: dirty
0=page has not yet been overwritten 1=page has been overwritten
A: accessed
0=page has not yet been accessed 1=page has already been accessed
U/S: page access level (user/supervisor)
0=supervisor (CPL=0..2) 1=user (CPL=3)
R/W: write protection (read/write)
0=page is read-only 1=page can be written to
P: page present
0=page is swapped 1=page resides in memory
res: reserved (equal 0)