www.system-call.org Issue 1 - May 2008

system call magazine

the FAT filesystem

We take a look at what it takes to implement Microsofts FAT filesystem.

writing a compiler

What's involved in writing a simple compiler or interpreter?

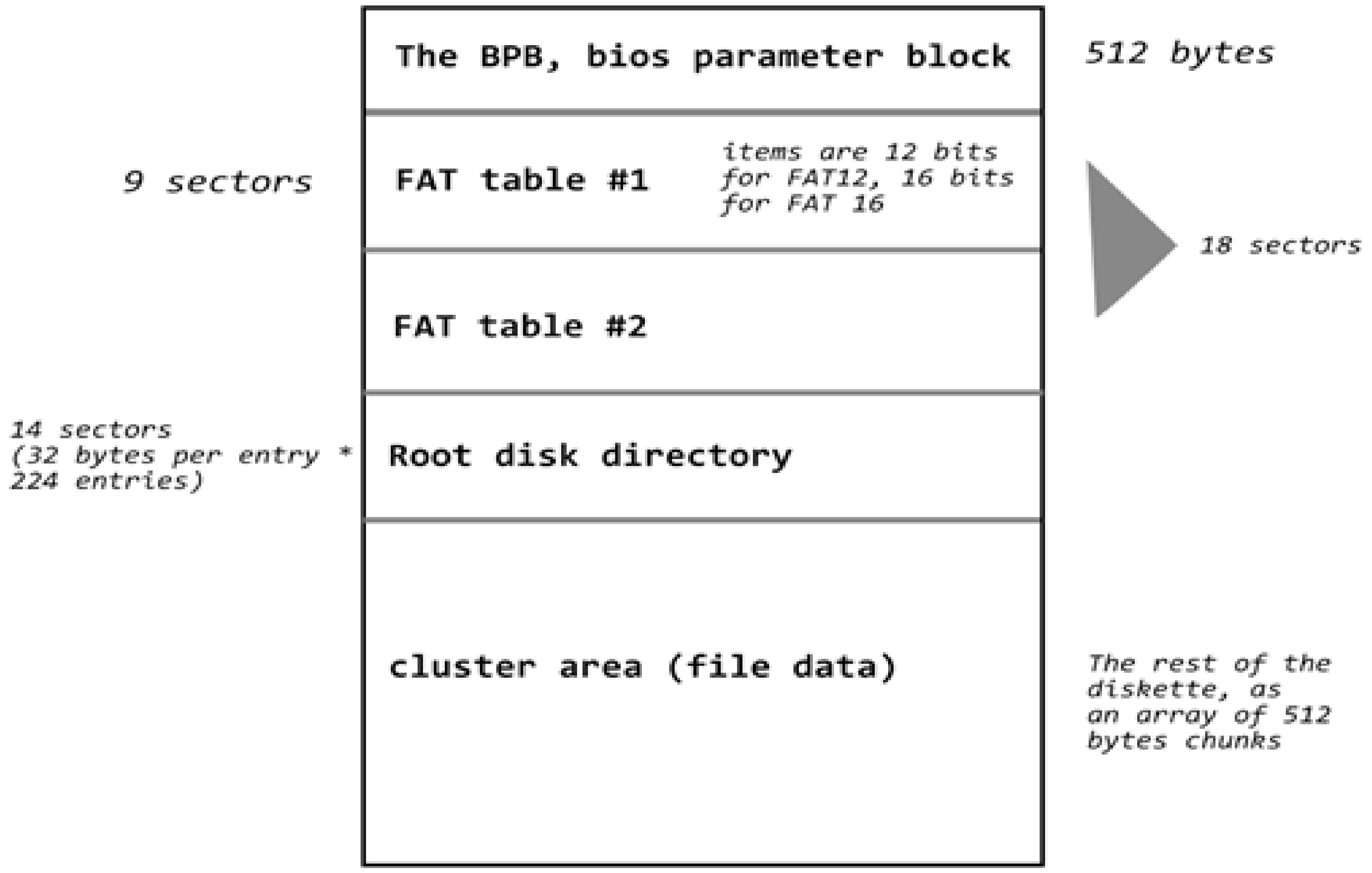
memory management

Interfacing with the x86's Memory Management Unit

Understanding the FAT File System

by jerryleecooper

The FAT file system was designed to manage disks in Stand-Alone Disk BASIC by Marc McDonald, with help from Bill Gates in 1977. Over the time, the FAT file system has been updated to allow for bigger partitions. Our primary focus today is FAT12 as this is use in most hobbyist operating systems.


The BPB, bios parameter block
512 bytes
9 sectors
FAT table #1
items are 12 bits for FAT12, 16 bits for FAT 16
18 sectors
FAT table #2
14 sectors (32 bytes per entry * 224 entries)
Root disk directory
cluster area (file data)
The rest of the diskette, as an array of 512 bytes chunks

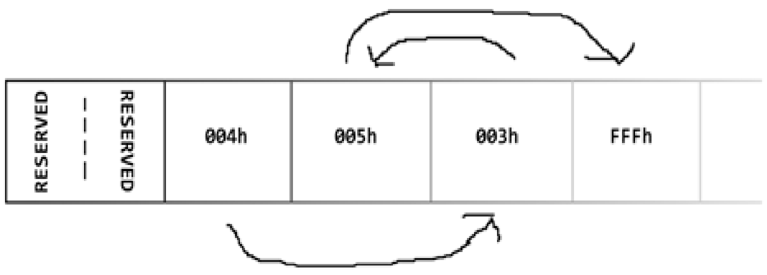
The FAT file system has three sections (in order they appear on disk):

1. The Boot Sector (contains the BPB and code to start the operating system, if one exists on the file system)
2. The File Allocation tables
3. Cluster Area

The boot sector is the most important part of the file system and it is only 512 bytes long! This is because often contains a boot loader. It begins with the BPB which defines important fields describing the file system structure. These fields include include bytes per sector, sectors per cluster, reserved sector count and sectors per file allocation table.

Following the boot sector is the File allocation tables (FAT). The FAT keep track of which clusters are used by files and in which order. Data storage is not perfect, and as such it also keep track of which clusters are bad. The FAT consists of an array of pointers. For FAT12, it's an array of 12-bit numbers pointing to next cluster within the file.

The cluster area is where the file data is stored. Sectors are grouped together into clusters. The number of sectors in each cluster is defined in the BPB, which is usually a single sector for floppy disks.
Directories are also located in the cluster area, which are files that contain an array of directory entries. Each directory entry contains the name, size, location of the first cluster, and attributes of files within the that directory.
To find a file, the file system looks at the root directory(which is located in the second cluster). Knowing where the first cluster of a file is, we can then read the entire file (or directory).



Offset	**Length**	**Description**
0x00	3	Jump instruction. This instruction will be executed and will skip past the rest of the (non-executable) header if the partition is booted from. See Volume Boot Record. If the jump is two-byte near jmp it is followed by a NOP instruction.
0x03	8	OEM Name (padded with spaces). MS-DOS checks this field to determine which other parts of the boot record can be relied on. Common values are IBM 3.3 (with two spaces between the “IBM” and the “3.3”), MSDOS5.0 and MSWIN4.1.
0x0b	2	Bytes per sector. A common value is 512, especially for file systems on IDE (or compatible) disks. The BIOS Parameter Block starts here.
0x0d	1	Sectors per cluster. Allowed values are powers of two from 1 to 128. However, the value must not be such that the number of bytes per cluster becomes greater than 32 KiB.
0x0e	2	Reserved sector count. The number of sectors before the first FAT in the file system image. Should be 1 for FAT12/FAT16. Usually 32 for FAT32.
0x10	1	Number of file allocation tables. Almost always 2.
0x11	2	Maximum number of root directory entries. Only used on FAT12 and FAT16, where the root directory is handled specially. Should be 0 for FAT32. This value should always be such that the root directory ends on a sector boundary (i.e. such that its size becomes a multiple of the sector size). 224 is typical for floppy disks.
0x13	2	Total sectors (if zero, use 4 byte value at offset 0x20)
0x15	1	Media descriptor

Reading a file

In order to read a file we must do the following:

1. Load the FAT.
2. Find the root directory (which is located at second cluster).
3. Find the file entry on the directory.
4. Get the pointer to the first cluster of the file, in the directories entry.
5. Go to the cluster pointed to.
6. Read cluster.
7. Go to the cluster's FAT entry. And read the pointer.
8. If pointer is not equal to 0xFFF (end of file) go to step 3.

SYSTEM CALL MAGAZINE

Practical Compiler Development - Part 1

by SandeepMathew

This is a rather informal introduction to development of a hobby compiler. The more formal chapters on compiler development will be given in later tutorials. By the end of this tutorial you will be able to create a simple interpreter . This can be easily converted into a compiler. The requirements for following this tutorial are:

1. A reasonable grasp of C / C++ / Python / C# / Java
2. An elementary knowledge of basic data structures and recursion

Compiler defined

A compiler is defined as a program that converts a source file in a specified form to an output file that can be fed to an assembler or simulated by a machine (virtual / real). Most of the compiler convert the given source file into assembly language that can be assembled by an assembler of the target machine. For example to see the code generated by your source program with GCC, type the following:

```
gcc -S source.c
```

Phases of a compiler

Lexical analysis : The lexical analysis phase of the compiler essentially does the tokenizing . This phase takes a stream of text as input and converts it into tokens required for the parser . For example the lexical analyzer splits the C source into if , else , while , int etc . The lexical analyzer can be constructed using a hard coded DFA or by using a generator like lex. In the next part of this tutorial I will describe how to write your own lex like generator . In this tutorial we will use an ad hoc lexical analyzer which will be sufficient for the purpose of demonstration. In fact you will be surprised to see that this sort of approach is taken in small hobby compilers like small-C.

Syntax Analysis : The lexemes must be arranged in such form that reflects the syntactic structure of the language . This is called syntax analysis or parsing . In this tutorial we will discuss only a form of top down parsing known as recursive descent parsing with one look ahead . Every programming language has a grammar which is usually represented in BNF form. A context-free grammar of a programming language is defined by the following:

1. A set of terminal symbols that cannot be substituted.
2. A set of non terminals which can be substituted by terminal or other non terminals.
3. A set of Productions that define the grammar (of the form NT -> A | B , i.e. LHS should have only one nonterminal).
4. A start symbol from which everything begins.

e.g. A grammar for the string *aaaa*.. any number of times is:

`S -> "a" S` { terminals are enclosed in " " and non terminals in capital letters }

{ | used to denote OR }

```
S -> nothing
```

ie this means that S can be an "a" followed by S itself or S can be nothing.

Lets see how aaa is derived:

```
S->aS
S->aaS ( using S -> aS)
S ->aaaS (using S->aS)
S->aaa (using S-> nothing).
```

Now lets see how a recursive descent parser is constructed for this grammar. This is easily done by replacing each non terminal by function call and performing a match for the terminal:

```
void parseS()
{
match_current_input_token_for("a");
if( not_matched) error();
if(end_of_input) return ;
parseS();
}
```

In order to construct a recursive descent parser like the above the grammar should follow these conditions:

1. For each production of the form `S -> A | C | E`, the first sets of each non-terminal in each of this production must be disjoint:
   ie `FIRST (A) intersection FIRST (C) intersection FIRST(E) = NULL`
2. The first sets and the follow sets of a non-terminal must be disjoint. i.e.
   `FIRST(A) intersection FOLLOW (A) = NULL`
   `FIRST(C) intersection FOLLOW ( C ) = NULL`
   and so on...

This implies that the first set should not be equal to the follow set of the non- terminal.

The first set of a terminal is the symbol itself. The first set of the non terminal is the set of terminals with which the non terminal can begin with. The follow set of a non terminal is the set of terminals which can come after the non terminal.

Now lets find out the First n Follow for the following grammar:

```
S -> "a" A | C D
A-> "b" A |"c"
C -> "e"| nothing
D-> "d" | "g" D
```

```
FIRST(S) = FIRST ( "a" A)
union FIRST(CD)
= "a" union FIRST( C )
= "a" union FIRST( C )
union FIRST (D) {since C
can be nothing, first of D
must be included}
= "a" union FIRST ( "e"
) union FIRST ("d") union
FIRST ("g")
= "a" union "e" union "d"
union "g"
= { a , e , d , g }
```

Similarly `FOLLOW(C) = FIRST (D) = { d , g }` etc…

Computing the first and follow of the rest of the symbols in the grammar is left as an exercise to you . These rules have to followed in order to avoid conflicts in the predictive parsing table. There will be more about advanced parsing methods in later chapters.

As a final example, grammar for the 'if' statement:

```
S -> IFCONDTION
IFCONDITON -> "if" EXPRESSION "then" STATEMENT
EXPRESSION -> TERM "+" TERM
TERM -> FACTOR *FACTOR
FACTOR -> id
EXPRESSION -> "(" EXPRESSION ")"
STATEMENT -> id ";" STATEMENT | nothing
```

It can be seen that right recursion in many cases is unneeded and can be removed using a simple loop, as right recursion leads to tail recursion. E.g.
The grammar `S -> aS | nothing` can be more compactly written as `S-> (a)*`

The parser reduces to:

```
parseS()
{
while(current_token == "a" ) advance_input_pointer();
}
```

Left recursion will lead to an infinite loop and should be removed. For example the grammar:

```
S -> S B | D
```

can be re written as

```
S -> D S_DASH
S_DASH -> B S_DASH
```

Common factor in grammars should also be removed. For example, the grammar:

```
S ->" if" condition "then" statement
S -> "if" condition "then" statement "else" statement
```

can be modified into:

```
S->AB
A -> " if" condition "then" statement
B -> "else" statement | nothing
```

Semantic Analysis: Syntax analysis does not tell anything about the meaning of the statement under consideration. It simply checks whether the syntax is correct. E.g.

```
const int i = 20;
i++;
```

The above lines are syntactically correct but semantically wrong. The semantics are sometimes given using attribute grammars. More about this in later chapters.

Code Generation: This is the final phase where the assembly code to the target machine is generated. I have skipped many other phases which are part of a more mature compiler. The purpose of this tutorial is to give quick introduction.

In most of the hobby compilers the major component is the parser and code generation and semantic analysis is done when each non terminal is encountered in the parser.

As an example of developing a simple interpreter, I have provided the source code of a simple basic-like interpreter. I have used the mingw port of gcc for compiling the source.

Available along with the magazine are the source files to this interpreter.

The x86 Memory Management Unit

by EinsteinJunior

Some years back, while I was reading a quite recent book in operating system development, I found some information about memory management in operating system design and how it's done out there with some specific operating systems, being it hobby or professionnal. I read something like "...the kernel is loaded in the upper 4GiB of address space". I also saw weird memory addresses which were 32 bits. At that time, I knew that most computers could have a maximum of 512 MiB of memory onboard. I thus asked myself: How is it possible to access memory above the gigabyte when you only have say 128MiB onboard? Weird thing with yet, a simple answer: the MMU is the guy behind this "Mafia".

The MMU stands for Memory Management Unit,also sometimes called Paged Memory Management Unit. It is computer hardware component responsible for memory access handling by the CPU. Its function depends on the type is system and type of CPU:

1. Virtual to physical address translation (virtual memory management). This will usually be found on more modern CPUs such as x86 based processors and others.
3. Memory Protection.
4. Cache Control / Bus Arbitration. This will be found on CPUs having a cache for cache control; and bus arbitration will be found in multi-core systems.
6. Bank switching. This is usually found in small 8 bit microprocessor systems such as the 6502 processor.

The MMU can reside in the same IC package as the CPU (that is the case in major 32 bit CPUs) or could be external to the CPU IC package.

As the major and most important (in our context) use for the MMU is for virtual memory management, we will not disgress further in it's other uses and concentrate on this topic.

To achieve its goal of address translation, the MMU divides the address (virtual) in little fields that will permit the location of data in memory. The entire virtual address space is divided in little chuncks of memory called pages. For those who are lost, a virtual address is the address the cpu 'sees and sends' to the MMU. The MMU will then translate this to a physical memory address. This is depicted in figure 1.

As said above, the linear address space is divided into little fields. The upper bits are used to point to a page an offset in a page (figure 2).

Each page directory is an array of page directory entries. Each directory entry contains the base physical address of a page table, i.e the physical start address of the page directory. It also contains attributes of the page table, the most common being a flag to indicate if the page is present in memory, a read only, system or user page table (figure 3).

The page table is an array of page table entries.Each page table entry contains the base address of a table, that is the address (physical) of the first byte of a table. Also, it contains attributes that are quite similar to the ones in the page directory entries (figure 4).

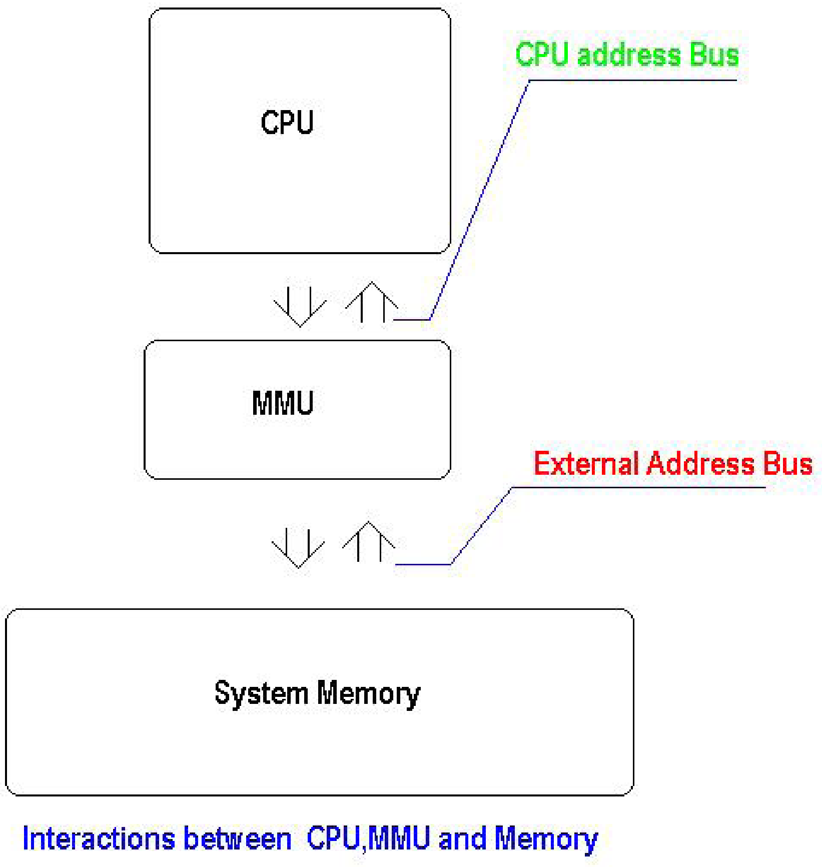


Figure 1

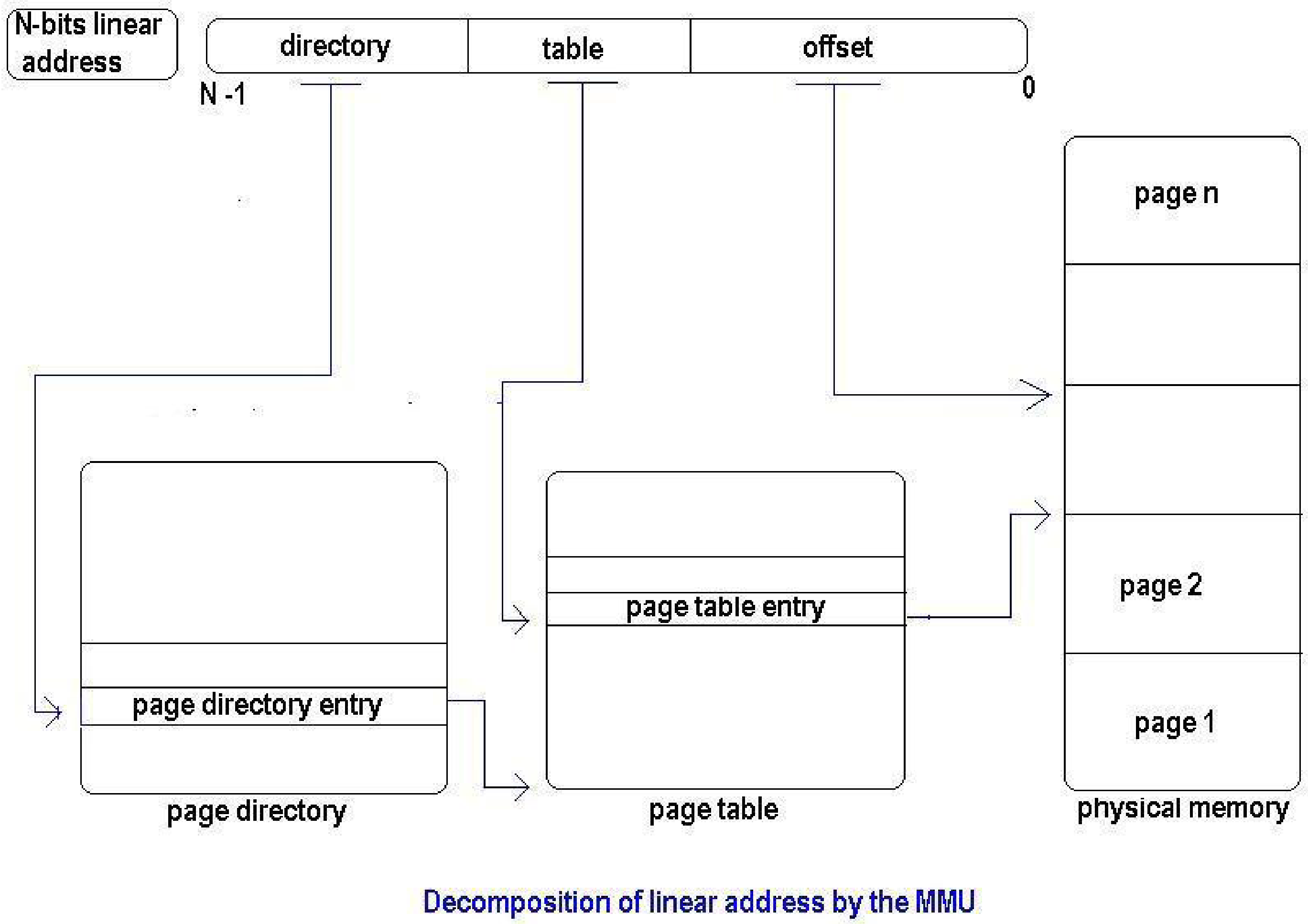


Figure 2

So to resume, the page directory contains entries that tell us where the page tables of a program are located in main memory, the page table tells us where the pages used by this program are located in memory. All of this excluding their properties such as the page being present in main memory or not and others.

The main purpose for keeping track of which pages are not in main memory is to simulate the existence of much more physical memory than what is present on the system. When the CPU tries to access a memory location in a page that is not in main memory, the MMU will generate a page fault.

A page fault is an exception indicating that a page is not present in main memory. So a system with 128 MiB of ram could use this mechanism to increase it's memory capacity to say 1GiB if we suppose that there is a secondary storage device such as a hard disk which can provide us with the rest of the 1GiB we need.

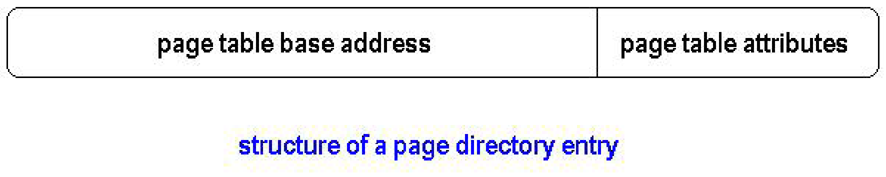


Figure 3

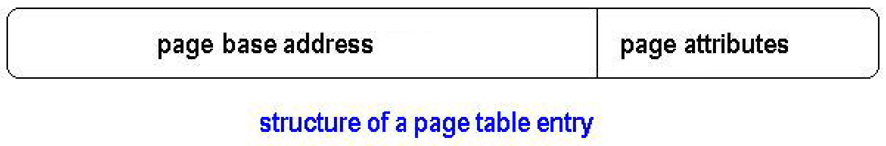


Figure 4

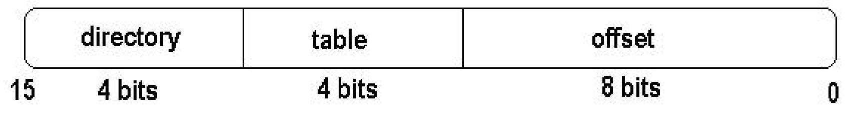


Figure 5

When a page fault occurs, the CPU invokes a programmer defined routine usually called the *page fault handler* that does the necessary routines to bring the missing page in main memory, by say sending an unused page into secondary memory.

The operation of moving an unused page to secondary memory is called swapping, and the file where the unused pages are stored is usually called the pagefile or swapfile.

I will take here a concrete example of how all of this works. Some people with some experience will sort out that my system is feasible, but non economic. My hypothetical system will consist of a CPU with an MMU.
This imaginary CPU will have a 16 bit address bus. This address will be divided as follows:

1. The directory field will be 4 bit wide so will have a max of 16 entriees.
2. The Table field will also be 4 bits, so 16 entries max also.
3. The offset field is 8 bits wide so each page will have 256 bytes.

This addressing scheme is shown in figure 5.

We will now suppose that the cpu has output address 0xFAAA on its address bus. All addresses are in hexadecimal. Since the upper 4 bits of the address are used to point to the page directory entry,it will point in this case to entry 0xF, i.e the topmost entry in the page directory. Lets further suppose that each of the entries in the page directory is in the format shown in figure 6.

Lets again suppose that in our case, entry 0xF contains 0xA002. 0xA00 i.e the three first bytes gives us the physical base address of the page table being used at the moment. Let's not care about the attribute bits for now.

Taking another look at our first address from the CPU, we see that the directory field (which is 4 bits in our case) is 0xA , i.e 10 in decimal. This means that entry 10 in our page table, the page table being located at address 0xA000 as we saw above, contains the address of the first bytes of the page where our information is located.

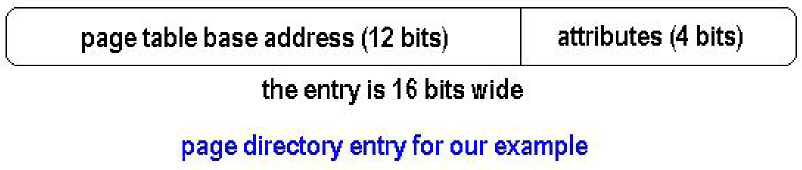

page table base address (12 bits)
attributes (4 bits)
the entry is 16 bits wide
page directory entry for our example

Figure 6

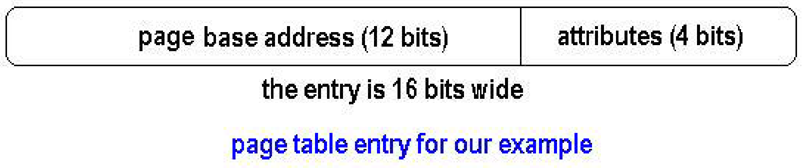

page base address (12 bits)
attributes (4 bits)
the entry is 16 bits wide
page table entry for our example

Figure 7

Now lets suppose (again!!) that our page table entries have the format like in figure 7.

In our case we will suppose (for the final time) that our entry number 10 contains 0xCB1A. This means the physical base address of our page being accessed is 0xCB1 or 0x0CB1.

Our offset into the page in this case (looking again at our CPU address 0xFAAA) is 0xAA, so our physical address the MMU sends to the memory chip is 0xCB1 + 0x00AA = 0x0D5B. So the virtual address 0xFAAA "becomes" the physical address 0x0D5B.

All of this is supposing the page is present in memory (that is the present flag is set). If it were not, then a page fault exception would have been generated and software would have been invoked to take appropriate actions to move the page from the pagefile to main memory. Many algorithms out there have been designed for that such as the LRU (Least Recently Used) algorithm.

My hopes are that this tutorial has not been long and boring and that many people gained something when reading this. Thanks to all. If you have any queries, email:

dzouato@yahoo.fr