Unit 3

Memory hierarchy

Labs of
Computer Architecture
v1.2

SESSION 1

Checking the principle of locality and introducing the cache simulator

Objectives

The main objective of this lab is to understand the operation of the memory hierarchy, starting from its highest level: the cache memory. The specific objectives are:

- To illustrate the principle of locality, both spatial and temporal, which is the base of the memory hierarchy operation.
- To highlight the enhancements obtained in the access time when a cache memory is used.
- To introduce the *SMC 2.0* cache simulator. This program allows practicing the basic concepts regarding the cache memory.

To meet these objectives, a simple program for the CT will be used.

Previous knowledge and required materials

In order to get the maximum benefit from this lab session, the student is required to:

- Attend the lab session with a copy of the memory hierarchy notes taken in the theoretical sessions of the course. It is highly recommended to peruse and understand the basic concepts of the cache memory operation.
- Attend the lab session with a copy of the DVD with the tools provided for the course. This DVD contains a modified GNU/Linux operating system.

Session development

1. The principle of locality

We will use a simple program for the CT to show the principle of locality. The listing of this program is shown below. This program converts the characters of a given string to upper case: it reads one character of the string, performs the `AND` logic operation between the character and a bit mask[1], and finally, stores the modified character in the string.

```
1         ORIGEN 100h
2
3         .DATOS
4  str VALOR  "Texto de prueba", 0
5
6         .CODIGO
7  ; Initialization
8      XOR  R0, R0, R0 ; R0 = 0
9      MOVL R1, BYTEBAJO DIRECCION str
10     MOVH R1, BYTEALTO DIRECCION str ; R1 points to str
11     MOVL R2, 0DFh ; All bits of R2 are set
12     MOVH R2, 0FFh ; except for the bit 5
13
14 bucle:
15     MOV  R3, [R1] ; R3 = string character
16     CMP  R3, R0
17     BRZ  final
18     AND  R3, R3, R2 ; R3 = upper-case character
19     MOV  [R1], R3 ; character is changed to upper case
20     INC  R1 ; R1 points to the next character of the string
21     JMP  bucle
22
23 final:
24     FIN
```

- ❏ Taking the above listing into account you must fill in the table 1.1, in which the memory address for each data item and machine code instruction must be specified.
- ❏ Compile the program using the *Ensambla.exe* tool. The character string will be loaded in the memory of the CT from the memory address specified in the `ORIGEN` directive. The machine code will be loaded in memory just after this string.
- ❏ Next, follow the program evolution and draw, in figure 1.1, a point in the row corresponding to the memory address accessed by each instruction. For instance, the first instruction, `XOR`

[1] Upper- and lower-case ASCII characters differ only in the sixth bit; in upper-case letters this bit is zero; in lower-case letters this bit is one. Thus, changing a letter from lower to upper case requires resetting this bit.

Address	Data	Address	Code
	'T'		`XOR R0, R0, R0`
	'e'		`MOVL R1, BYTEBAJO DIRECCION str`
	'x'		`MOVH R1, BYTEALTO DIRECCION str`
	't'		`MOVL R2, 0DFh`
	'o'		`MOVH R2, 0FFh`
	' '		**MOV R3, [R1]**
	'd'		`CMP R3, R0`
	'e'		`BRZ final`
	' '		`AND R3, R3, R2`
	'p'		**MOV [R1], R3**
	'r'		`INC R1`
	'u'		`JMP bucle`
	'e'		
	'b'		
	'a'		
	0		

Table 1.1: Memory address assignment of the above program

`R0, R0, R0`, is stored in the memory location `110h`, and thus, a point is drawn in the intersection between the line representing the `110h` memory address and the first vertical line (the first memory address accessed).

Trace the program until completing the whole figure. You must take into account that two instructions of the program perform two memory accesses: one for fetching the machine code, and another one for fetching the operand; they are highlighted in bold in table 1.1. These instructions require two points in the same vertical line: one representing the memory location of the machine code, and the other one representing the memory location of the data operand.

❏ Considering the memory accesses represented in figure 1.1, are they random or do they follow a pattern?

❏ Write down an example of spatial locality in the code accesses of the program.

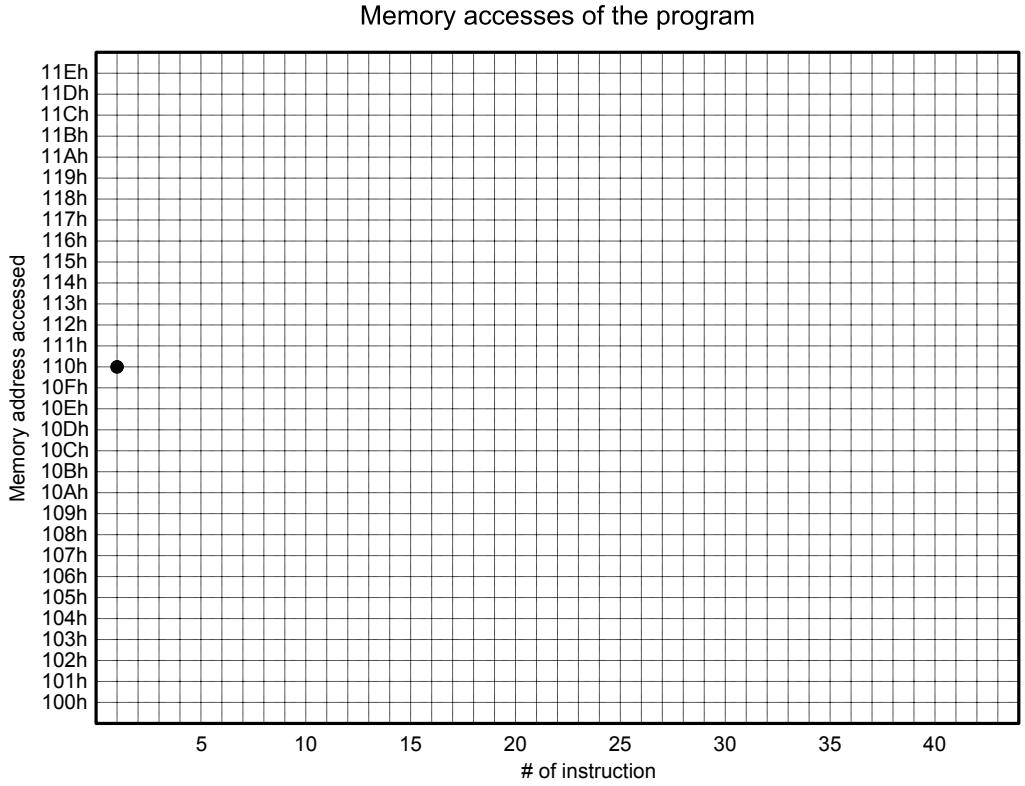


Figure 1.1: Memory accesses of the above program

❑ Write down an example of temporal locality in the code accesses of the program.

❑ Write down an example of spatial locality in the data accesses of the program.

❑ Write down an example of temporal locality in the data accesses of the program.

2. Introduction to the *SMC 2.0* cache simulator

In this section a two-level memory hierarchy for the CT will be designed. The two levels are cache and main memory. The *SMC 2.0* cache memory simulator will be used for this task. This simulator shows the evolution of the cache memory when memory accesses are performed, as well as the cache hit rate.

The features of the memory hierarchy for the CT are the following:

- The size of the main memory is 64 Ki-words, 16-bit wide each.
- The cache memory is a unified cache and stores 4 blocks, 16-word each. It implements a fully associative placement strategy with an LRU replacement algorithm.

Before running the simulator and designing this memory hierarchy, try to answer the following questions:

❏ How many memory blocks does the main memory contain? [1]

❏ What is the size of the cache memory expressed in bytes? [2] And expressed in words? [3]

1

2

3

Now, the memory hierarchy will be designed and the previous answers will be checked.

❏ Run the *SMC 2.0* simulator and in the *Ayuda* menu select *English* to work with the English version.

❏ Select *Configure→Main memory* in the menu. The window shown in figure 1.2 will show up.

❏ Choose the proper options for the desired memory system in this window. Firstly, set the *Addressable word width* to 16.

❏ Secondly, select the number of words per block.

❏ Next, the number of blocks.

❏ At this moment, check that the size of the main memory matches the value computed by the program.

❏ Select *write-through*.

❏ Click OK. You may check in the summary table, *View→Current memory system*, that the values of the parameters match the desired configuration.

❏ Click on *Configure→Cache memory*. The window shown in figure 1.3 will show up, allowing the configuration of the number of cache levels of the memory system, as well as the definition of unified or separated cache levels. At most, three levels of cache memory can be specified, although one level is used by default.

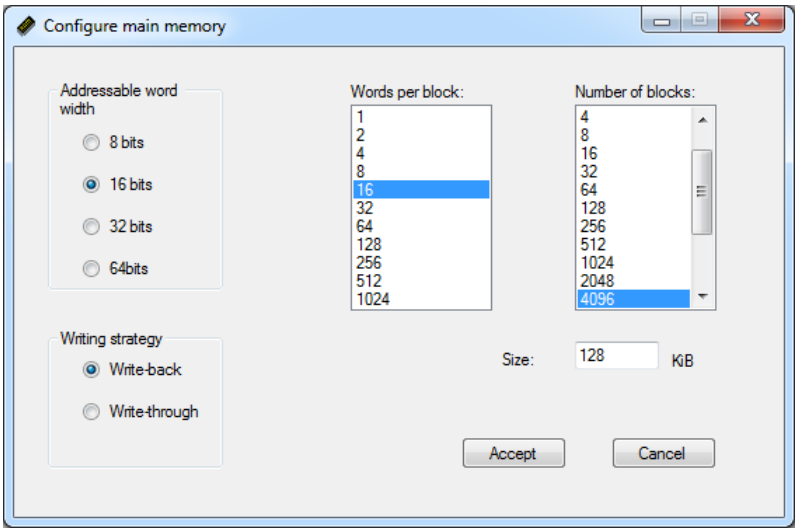


Figure 1.2: Main memory configuration

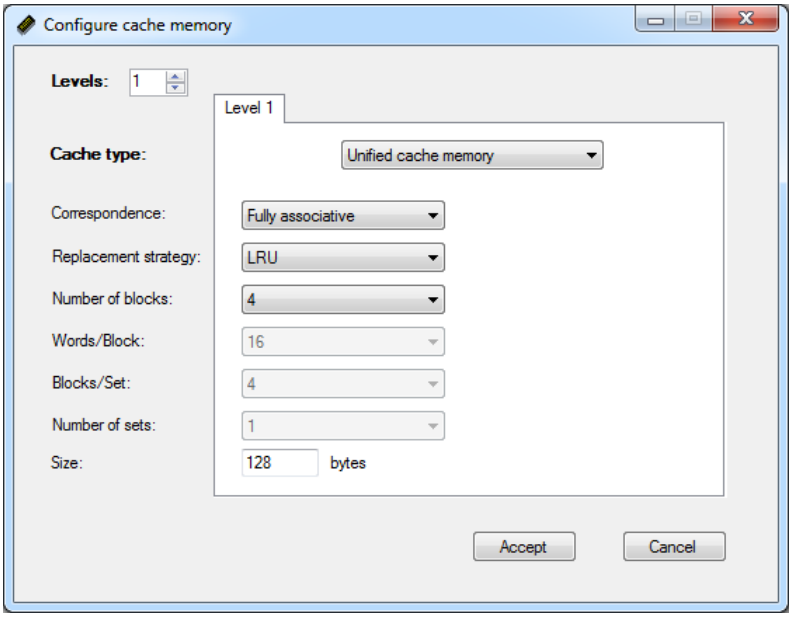


Figure 1.3: Cache memory configuration

- ❏ In the configuration window, a tab to configure each cache level is shown. Select only one level for the cache memory.
- ❏ For each cache level, the upper list allows indicating whether the level is unified, as shown in figure 1.3, or separated into code and data. All the cache levels are unified by default.
- ❏ Select the fully associative placement strategy in the list box labeled *Correspondence*.
- ❏ Select the LRU-based replacement technique.
- ❏ Choose the number of blocks according to the memory system.
- ❏ Check whether the answer provided above for the size of the cache memory, expressed in bytes, is correct.

❏ Accept the introduced values and exit the cache level configuration window.

Once the memory system has been configured, it may be observed with the option *View→Representation of the memory system*.

❏ Save the configuration of the memory system to a file called `3-1cache.memx`, *File→Save memory system*.

3. Reducing access time with cache memory

Once the memory system has been designed the next step involves checking its performance. The improvement factor obtained when a two-level memory system is used with regard to only one memory level is computed.

- The cache memory, built up with SRAM technology, has an access time equivalent to a clock cycle of the CPU. In this exercise, assume that this clock frequency is 1 GHz.
- The main memory, built up with DRAM technology, is slower than the cache. In this exercise, accessing the main memory consumes 20 clock cycles per memory location.

To check the operation of the memory system, the program to change characters from lower to upper case will be used.

❏ Load the execution trace of the program described in section 1. In the *File* menu, select the *Load trace* option; a dialog box will show up. The cache simulator can operate with two file types: trace files, `.prg`, and executable files of the CT, `.eje`. Thus, two possibilities are available:

1. Load the file `3-1cache.prg`. This file contains the memory addresses that are accessed when the CPU runs the program.
2. It is also possible to load the trace from an executable file of the CT. To do so, you must select the `.eje` file, and open `3-1labcache.eje`. This option only works if the program does not contain infinite loops nor I/O waiting events (keystrokes, for instance).

When the trace file has been loaded, the *Simulate* option will be enabled.

❏ In the *View* menu, the *Current trace* option shows information about the trace. Taking this information into account, how many memory accesses are performed in the execution of this program? [4]

[4]

- ❏ In the *View* menu, the *Spatial/temporal locality* option shows a figure representing the memory accesses performed by the program, as you have drawn in figure 1.1.
- ❏ If the only level in the memory system of this computer were the main memory, how many clock cycles would all the memory accesses consume? [5]
- ❏ Taking the clock frequency into account, how long do the memory accesses require? [6]

5

6

Now, the memory hierarchy designed in the section 2 is considered. This hierarchy consists of a main and a cache memory level. In this hierarchy, the time required to perform all the memory accesses of the program will be computed. Then, the enhancement factor over a one-level system may also be computed.

- ❏ Simulate the execution of the whole program in this memory hierarchy and write down the results. To do so, select the *Simulate* menu option. Then, a window as the one represented in figure 1.4 will show up.

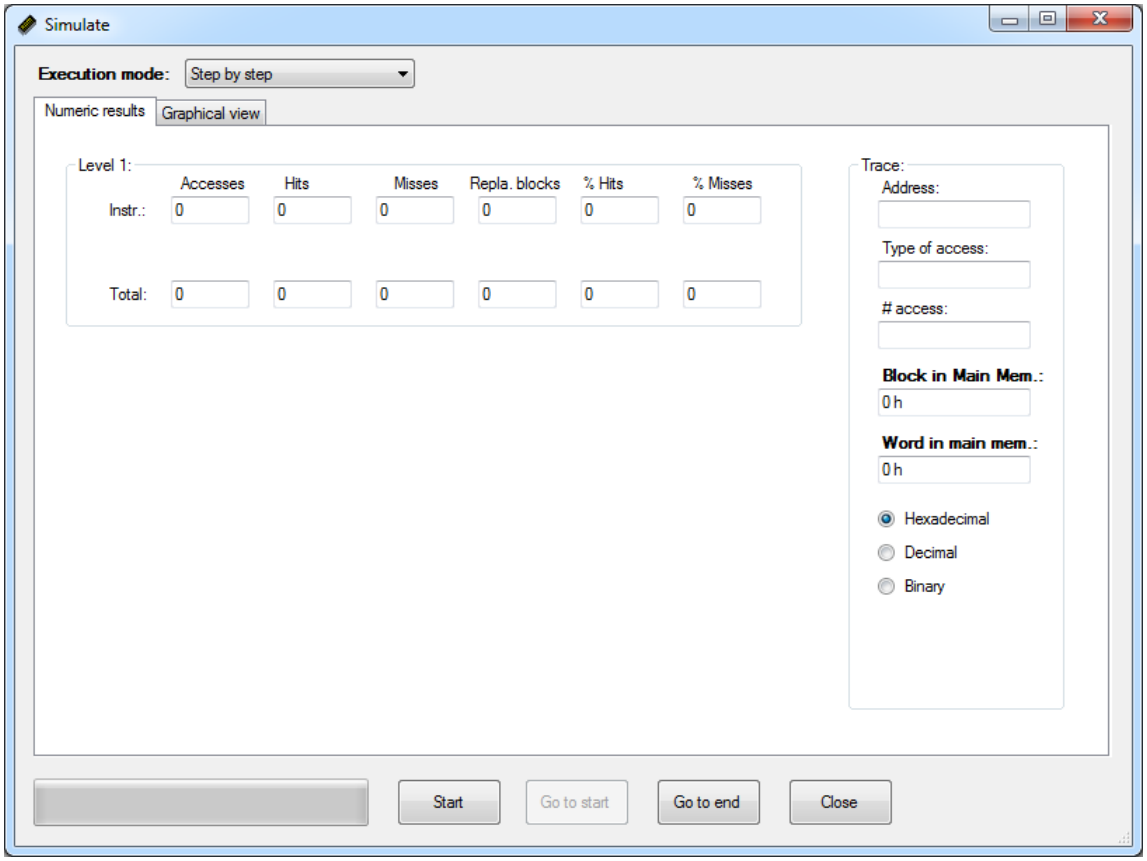


Figure 1.4: Simulation window

- ❏ Using the *Numeric results* and *Graphical view* tabs you may switch the way in which the results are presented. In this case the numerical results are preferred.

- ❏ Choose the debugging mode. You have two options: step-by-step or breakpoint execution. The former allows viewing the results after performing the following memory access, whereas the latter allows viewing the results after a fixed number of accesses. In any case, clicking on Go to end triggers the simulation process until the end of the trace.
- ❏ Click Go to end. When the simulation process is finished, a report with the simulation results is presented. What is the hit rate for the program in this memory system? [7] Check that this value is obtained as the division of the number of hits by the number of memory accesses.
- ❏ Taking the hit rate into account, as well as the number of memory accesses, what is the number of clock cycles consumed in all the memory accesses of the program? Write down the expression required for this computation below. Remember that an access to the main memory copies the whole block to the cache.

7

- ❏ In this memory hierarchy, how long does it take to complete all the memory accesses? [8]
- ❏ Now, the effect of the locality in the access time is illustrated. Firstly, assume that the memory access pattern is fully random, so no locality is present. What would be the cache hit rate in the above memory hierarchy? [9]
- ❏ What would be the average access time to the memory system? [10] And the required time to complete all the memory accesses? [11]
- ❏ Compare the average time obtained using the cache memory with the average access time to the main memory. Is it worth the use of a memory hierarchy? Why?

8

9

10

11

- ❏ Now, assume the ideal case in which the locality is very high or the cache capacity is very big, so the cache hit were 100 %. What would be the average access time to the memory system? [12]
- ❏ The time obtained in the previous question presents a peculiarity, what is?

12

❑ What would be, in this ideal case, the time required to complete all the memory accesses of the program? [13]

[13]

4. Exercises

✏ Configure the memory system changing the block size. Firstly, reduce the block to half its size while keeping the size of the cache memory constant. To make this change you need to edit the *Main memory* and *Cache* options. Repeat the simulation of the trace and get the cache hit rate. What is the number of clock cycles required to perform all the memory accesses? Compare this result with the one obtained in section 3.

✏ Keeping the cache size constant, repeat the previous exercise doubling the block size. What is the number of clock cycles required to perform all the memory accesses? Compare this result with the results obtained above.

SESSION 2

Placement strategies in cache memory

Objectives

The main objective of this lab session is two-fold:

- Understanding the placement strategies in the cache memory.
- Checking the performance improvements obtained when modifying the design parameters of a cache memory, as well as quantifying its influence in the execution of example programs.

The *SMC 2.0* cache simulator is used to achieve the above goals. This simulator allows practicing key concepts of cache memory. Furthermore, it provides the results of simulating the memory accesses of a program in terms of cache hit rates. In this lab, the program used in the previous session which converts the characters of a string to upper case (see session 1) is used to quantify the behaviour of different configurations of the memory system.

Previous knowledge and required materials

In order to get the maximum benefit from this lab session, the student is required to:

- Attend the lab with a pencil and a eraser. Also, with the cache memory notes of the theoretical classes.
- Understand the main characteristics of the different placement strategies in the cache memory of a computer.

Session development

1. Implementing a two-level memory system

In this section, a two-level memory system, with cache and main memory, is designed and implemented. Several cache configurations will be checked over this system in next sections. The features of the memory system are the following:

- Memory word size: 16 bits.
- Memory block: 2 words.
- Main memory blocks: 512 blocks.
- One level of unified cache.
- Cache memory blocks: 8 blocks.
- Cache placement strategy: direct-mapped.
- Write strategy: Write-through.

Before running the *SMC 2.0* simulator, answer the following questions:

❏ What is the size of the main memory expressed in KiB? [1]

❏ What is the size of the cache memory expressed in bytes? [2]

❏ Run the *SMC 2.0* cache simulator and in the menú *Ayuda* select *English* to work with the English version.

❏ Build the memory system described above. Remember that you must start with the option *Configure→Main memory*, and then continue with the option *Configure→Cache memory*. Check your previous answers.

❏ Once the memory system is configured, save this configuration to a file called `3-2direct-mapped-2.memx` (direct-mapped placement strategy and 2 words per block).

1

2

Select the menu option *View* and then *Representation of the memory system* to see the aspect of the configured memory system.

To check the performance of the memory system, the trace of the program to switch the characters of a string to upper case will be used.

❏ Select the *File* menu option and then *Load trace*. In the dialog box choose the file extension `.prg`, and select `3-1cache.prg`. The *Simulate* menu will be activated and the status bar will display the trace file loaded.

Now, the memory system can be checked. In this first simulation we will check how the cache memory is used. We will try to guess the result of the execution and then we will check the answer with the simulator.

The program memory organization is shown in figure 2.1. On the right, the main memory addresses are written in hexadecimal; on the left, these addresses are written in binary grouped by fields, from right to left: offset, 1 bit; index, 3 bits; and tag, 6 bits.

- ❏ Fill the gaps in the figure for the number of the main memory block, both in decimal and hexadecimal. This step facilitates understanding the information provided by the cache simulator.
- ❏ Select *Simulate*, step-by-step mode. Then, go to the *Cache view* tab to see the content of the cache memory.
- ❏ Taking into account the information provided by figure 2.1, trace the execution of the program until the second time that a code block is replaced by a data block. Do the two following tasks for each access:
  1. If the memory access generates a cache miss, fill the figure 2.2 writing in the corresponding gap the number of the main memory block which is cached. In each replacement you must erase this number and write the block which occupies its place.
  2. For the first access, you have to press Start to simulate it. For each of the rest of accesses, press Continue to simulate them. Check that the result matches your prediction in the previous step. On the right of the window, the status of the cache is shown. For each cache block the following information is shown:
     - The cache block number. It is labelled as Set[1].
     - The tag in decimal.
     - The valid bit.
     - The dirty bit.
     - The LRU value for the block. In this case the LRU value has no meaning since a direct-mapped placement strategy is being used.
- ❏ Which access causes that a code block is replaced for a second time with a data block?[3] Which instruction is being executed?[4]

[3]

[4]

2. Testing several memory configurations

In this section several memory configurations are tested.

[1]The direct-mapped placement strategy is a particular case of the set associative placement strategy in which each set has only one way.

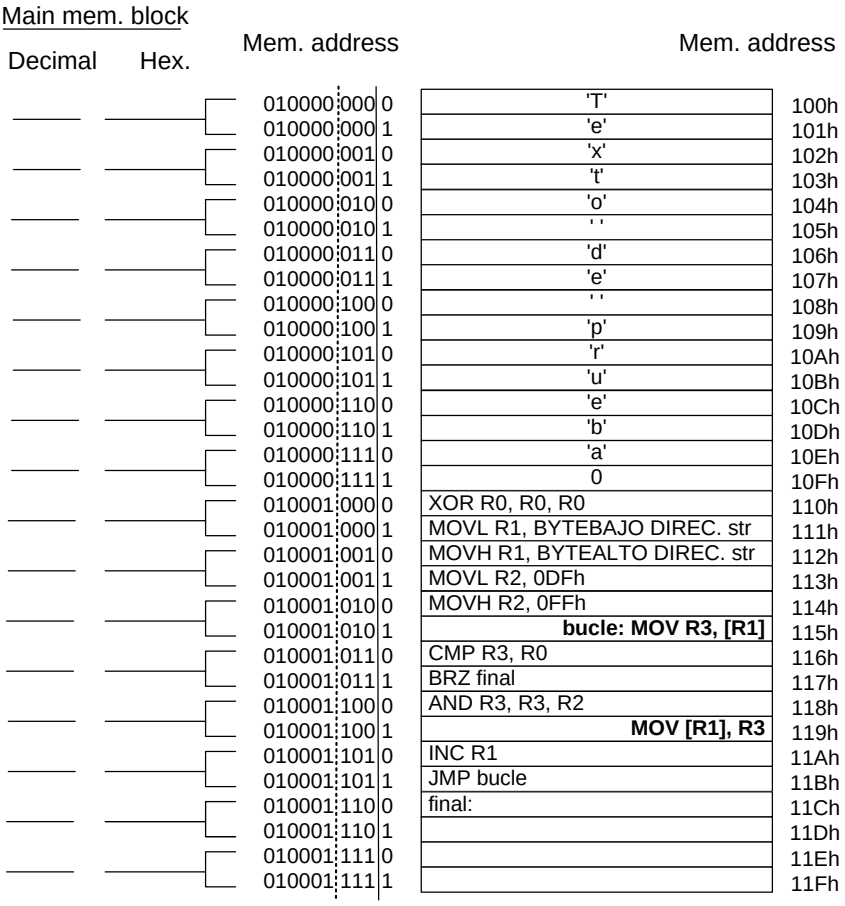


Figure 2.1: State of the memory fragment in which the program is loaded

- ❏ Load the memory configuration implemented in the previous section, `3-2direct-mapped-2.memx`. Now, the whole program will be executed on this memory system. Select *Simulate* and select the *Numeric results* tab. Select the *Step by step* execution mode and press Go to the end.
- ❏ Fill the corresponding row in table 2.1 with the results provided by the cache simulator.
- ❏ Repeat the simulation process with 4 words per block and the same placement strategy, direct-mapped. Remember that you have to modify the memory system maintaining both cache and main memory sizes. Simulate the trace of the program and write the results in the corresponding row of table 2.1.
- ❏ Repeat the simulation process with 8 words per block and direct-mapped placement strategy. Remember to configure the system with the original cache and main memory sizes. Simulate the trace of the program and write the results in the corresponding row of table 2.1.
- ❏ Taking into account the results of the previous three simulation processes, do you think that the hit rate depends on the block size with a direct-mapped placement strategy?

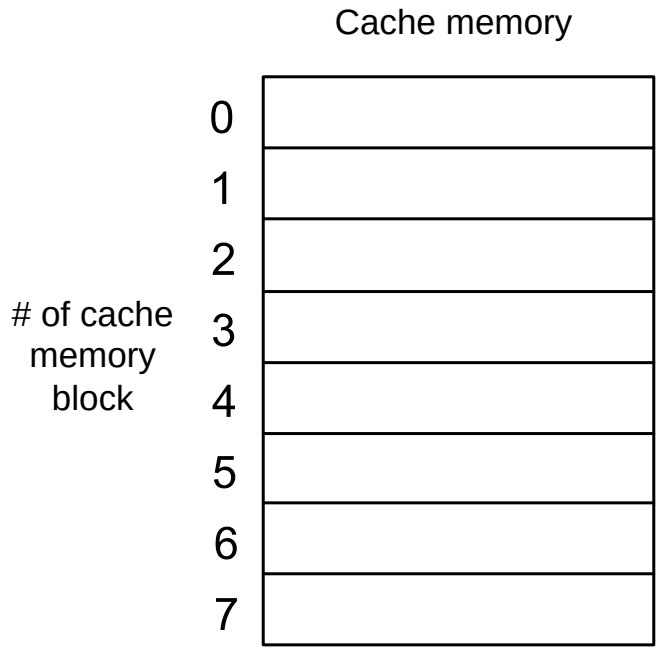


Figure 2.2: Cache memory

Placement strategy	Words per block	# of sets	# of ways	# of hits	# of misses	Hit rate (%)	Miss rate (%)
Direct-mapped	2	–	–				
Direct-mapped	4	–	–				
Direct-mapped	8	–	–				
Fully associative	2	–	–				
Fully associative	4	–	–				
Fully associative	8	–	–				
n-way associative	2	4	2				
n-way associative	2	2	4				
n-way associative	4	2	2				

Table 2.1: Testing results with different placement strategies

Now, the placement strategy is modified. At the end of the document for this lab session, figure 2.3 shows the program loaded in memory to help you tracing its execution depending on the placement strategy selected.

❑ Configure the memory with a fully associative placement stra-

tegy with 2 words per block. Furthermore, configure the cache memory to use the LRU replacement strategy. Be sure that the original sizes for both the cache and the main memory are configured. Simulate the trace of the program and write the results in the corresponding row of table 2.1.

- ❏ Configure the memory with a fully associative placement strategy with 4 words per block. Simulate the trace of the program and write the results in the corresponding row of table 2.1.
- ❏ Repeat the simulation process with 8 words per block and a fully associative placement strategy. Simulate the trace of the program and write the results in the corresponding row of table 2.1.
- ❏ The obtained results should be very different compared with the ones obtained with the direct-mapped placement strategy. Can you explain this difference?

- ❏ Now, a n-way set associative placement strategy is used. Configure the memory system with 4 sets and 2 ways, and thus, 2 words per block. Be sure that the original sizes for both the cache and the main memory are configured.

  When configuring the cache memory in the simulator, select *Set associative* and then write the number of ways in *Blocks/Set*.

  Simulate the trace of the program and write the results in the corresponding row of table 2.1.
- ❏ Repeat the simulation process with a cache memory of 2 sets and 4 ways. Simulate the trace of the program and write the results in the corresponding row of table 2.1.
- ❏ Repeat the simulation process with a cache memory of 2 sets, 2 ways, and 4 words per block. Be sure that the original sizes for both the cache and the main memory are configured. Simulate the trace of the program and write the results in the corresponding row of table 2.1.
- ❏ Considering the results of the nine experiments, which is the best configuration taking into account the performance and the cost of the memory system?

3. Exercises

✏ The performance of unified and separated caches will be tested. A set associative placement strategy with 2 words per block is used. Search in table 2.1 the row corresponding with 2-way set associative with 4 sets. Write the results in table 2.2 for the unified cache.

Cache type	# of hits	# of misses	Hit rate (%)	Miss rate (%)
Unified				
Separated → Data				
Separated → Code				
Separated → total				

Table 2.2: Unified vs. separated set associative caches

✏ Configure the cache memory as separated caches for data and code. Select *Cache type* as *Separated cache memory* in the cache configuration window.

Configure the two caches with a set associative placement strategy, LRU replacement strategy and 2 words per block. The main memory must implement the same block size. Distribute the number of sets: 2 sets of 2 ways for the data cache and similarly for the code cache.

✏ Simulate the trace file of the program over this new memory system and write the results in the corresponding rows of table 2.2.

✏ If the configuration and the simulation processes are correct, the separated cache will have provided a lower or equal hit rate than the unified cache. However, modern real computers implement a L1 separated cache with a set associative placement strategy, what do you think the reason is?

01 0000 0000	'T'	100h
01 0000 0001	'e'	101h
01 0000 0010	'x'	102h
01 0000 0011	't'	103h
01 0000 0100	'o'	104h
01 0000 0101	' '	105h
01 0000 0110	'd'	106h
01 0000 0111	'e'	107h
01 0000 1000	' '	108h
01 0000 1001	'p'	109h
01 0000 1010	'r'	10Ah
01 0000 1011	'u'	10Bh
01 0000 1100	'e'	10Ch
01 0000 1101	'b'	10Dh
01 0000 1110	'a'	10Eh
01 0000 1111	0	10Fh
01 0001 0000	XOR R0, R0, R0	110h
01 0001 0001	MOVL R1, BYTEBAJO DIREC. str	111h
01 0001 0010	MOVH R1, BYTEALTO DIREC. str	112h
01 0001 0011	MOVL R2, 0DFh	113h
01 0001 0100	MOVL R2, 0FFh	114h
01 0001 0101	**bucle: MOV R3, [R1]**	115h
01 0001 0110	CMP R3, R0	116h
01 0001 0111	BRZ final	117h
01 0001 1000	AND R3, R3, R2	118h
01 0001 1001	**MOV [R1], R3**	119h
01 0001 1010	INC R1	11Ah
01 0001 1011	JMP bucle	11Bh
01 0001 1100	final:	11Ch
01 0001 1101		11Dh
01 0001 1110		11Eh
01 0001 1111		11Fh

Figure 2.3: Program loaded in memory

SESSION 3

Analysis of real caches

Objectives

The objective of this lab is to show how the cache memory and the programming style influence the performance of real computers.

Previous knowledge and required materials

In order to get the maximum benefit from this lab session, the student is required to:

- Attend the lab session with a copy of the DVD with the tools provided for the course. This DVD contains a modified GNU/Linux operating system.

Session development

1. Retrieving the parameters of the cache memory

In this section we retrieve the parameters of the cache memory of the computer used in this lab. These parameters are important to explain the temporal behaviour of the programs that are executed later. Since simplified versions of memory systems are used in the lectures of the course, observing real values of these parameters gives us a perception of the reality.

❑ Boot the Windows operating system of the lab PC and run `CPU-Z`. This application shows information about the PC hardware. If you run this program without administrator privileges an error will be displayed; you can ignore it since it does not affect the information we want to retrieve. Click on the `Caches` tab. Draw a graphical representation of the cache levels pointing out in each

level the size, placement strategy, block size, and whether it is unified or separated. You must take into account that multicore processors usually replicate in all cores the cache levels closest to the core.

❑ Run the `Everest` application. This application shows detailed information about the PC, though we will only use the memory benchmarks to obtain the speed of the cache levels and the main memory. Select `Tools` in the menu bar and then `Cache And Memory Benchmark`. Click on `Start Benchmark` to run the memory benchmark. Once the benchmark is executed, the speeds and latencies of the different memory levels will be displayed. Write down the read speed[1] and latency for each cache level and the main memory.

The benchmark results must be considered qualitative since they

[1] Writing speed is similar, although slightly higher, as it happens in most of storage devices.

depend on the memory access pattern. Nevertheless, they are quite significant.

- ❏ How many times is the L1 cache faster than the main memory? 1
- ❏ Actually, the speed gap between the cache and the main memory is larger since the speed displayed has been obtained by means of accessing many consecutive addresses.[2] When a short number of memory addresses, or even only one word, is accessed, the most important parameter is the latency, which represents the time required to access the first word. Taking the value of the latency into account, how many times is the L1 cache faster than the main memory? 2
- ❏ The L2 cache memory is faster than the main memory but slightly slower than the L1 cache even though they are built with the same technology. This speed difference lies on the difference in size: a bigger size implies a larger response time.[3] How many times is the L2 cache faster than the main memory using the reading speed as the comparison criterion? 3 Analogously, how many times is the L2 cache faster than the main memory using the latency as the comparison criterion? 4

1

2

3

4

2. Influence of the locality in system performance

We will analyze the influence of the programming style on the locality of the programs and, thus, on their response times.

- ❏ Reboot the PC and run the Linux operating system.
- ❏ Copy the `3-3loc1.c` file to your working folder. The content of this file is listed below.

```
#define NROWS     8192 // 2^13 rows
#define NCOLUMNS  8192 // 2^13 columns
#define NTIMES    10   // Repeat 10 times

// Matrix size 2^26 = 64 MiB
char matrix[NROWS][NCOLUMNS];

int main(void) {
  int i, j, rep;

  // Repeat NTIMES to obtain an easy-to-measure elapsed time
  for (rep = 0; rep < NTIMES; rep++) {

    for (i = 0; i < NROWS; i++) {
```

[2] The main memory is designed using parallelism techniques to access several words at the same time, mitigating its low speed

[3] For instance, the more sets in L2 cache memory, the more time it takes to locate the right set to be accessed.

```
      for(j = 0; j < NCOLUMNS; j++) {
        matrix[i][j] = 'A';
      }
    }
  }
}
```

- ❏ Compile and link the program with the following command:

  `$> gcc 3-3loc1.c -o 3-3loc1`

- ❏ Run the executable file with the command:

  `$> time ./3-3loc1`

- ❏ What is the response (or elapsed) time of the `3-3loc1` program?[5] To obtain a reliable measurement run the program two more times and write down the average `Real` time.

[5]

- ❏ Now, we will have a look at the sequence of accesses to the matrix. To do so, insert the following code snippet just after the `matrix[i][j] = 'A';` statement. Save the file as `3-3loc2.c`.

```
// Write the addresses accessed in the first iteration for the
// first four rows
if (rep == 0 && i < 4) {
  if (j == 0)
    printf("Beginning of row\n");
  printf("%p\n", &(matrix[i][j]));
}
```

  The above snippet writes the addresses used to access the matrix and labels as `Beginning of row` the beginning of each row for the first iteration and the first four rows. Remember to add the `#include <stdio.h>` directive to use the `printf` function.

- ❏ Compile and link the new version of the program. Run it with the following command:

  `$> ./3-3loc2 > result2.txt`

- ❏ Edit the file `result2.txt` and have a look at the sequence of addresses accessed, paying attention to the last address of each row and the first of the next row (you can easily find the end of a row by searching the `Beginning of row` string with the text editor). Is there any relationship between each pair of addresses? What happens with all the addresses when accessing all the elements of the matrix?

- ❏ Now, we will estimate the locality of the `3-3loc1.c` program when accessing all the elements of the matrix. One way of doing this is by computing the cache hit rate of the program for a given reference cache: the bigger the hit rate, the larger the locality. Since

we are only interested in the locality in the accesses to the elements of the matrix (data) we will ignore the effect of executing the operating system and other tasks also loaded in memory. We will also ignore the effect of the task code. Taking these assumptions into account, what is the L2 cache hit rate? You must also assume that the cache is empty prior to accessing the first element of the matrix.[6]

[6]

- ❏ We will modify the `3-3loc1.c` program to analyze the effect of the programming style in the response time of the program. Change the order of the statements that iterate over the rows and columns of the matrix to iterate first over all the rows of each column.

```
for (j = 0; j < NCOLUMNS; j++)
  for (i = 0; i < NROWS; i++) {
    matrix[i][j] = 'A';
  }
```

  Save the new version of the program as `3-3loc3.c`.

- ❏ Compile and link the above file. Run the program with the following command:

```
$> time ./3-3loc3
```

- ❏ What is the response time of the `3-3loc3` program?[7] To obtain a reliable measurement run the program two more times and write down the average `Real` time.

[7]

- ❏ Compare the average response time of the `3-3loc1` and `3-3loc3` programs. What has happened? Can you explain it?

- ❏ Now, we will analyze the sequence of accesses to the matrix after changing the order of accesses. Add the following code snipped just after the `matrix[i][j] = 'A';` statement in the source code of the `3-3loc3.c` program. Save the new version of the program as `3-3loc4.c`.

```
// Write the addresses accessed in the first iteration for the
// first four rows
if (rep == 0 && j < 4) {
  if (i == 0)
    printf("Beginning of column\n");
```

```
  printf("%p\n", &(matrix[i][j]));
}
```

The above snippet writes the starting address of each column in the first iteration for the first four columns. Remember to add the `#include <stdio.h>` directive to use the `printf` function.

❏ Compile and link the new version of the program. Run it with the following command:

```
$> ./3-3loc4 > result4.txt
```

❏ Edit the file `result4.txt` and have a look at the sequence of addresses accessed, paying attention to the last address of the first column and the first address of the second column. What happens with all the addresses when accessing all the elements of the matrix?

❏ Now, we will estimate the locality of this program when accessing all the elements of the matrix. We will ignore the effect of executing the operating system and other tasks also loaded in memory. We will also ignore the effect of the task code. Taking these assumptions into account, what is the L2 cache hit rate?[8] Does this explain the difference between the elapsed time of programs `3-3loc1.c` and `3-3loc3.c`?

8

When the size of the cache memory is enough to store the code and the data of the program the locality has little influence on the elapsed time.

❏ Modify the `3-3loc1.c` and `3-3loc3.c` programs changing the number of rows and columns for the matrix to use about half the capacity of the L1 data cache. Increase the number of iterations for the loop to take several seconds to be executed.[4] Save the programs as `3-3loc5.c` and `3-3loc6.c`, respectively. Compile, link and run both programs; compare their average elapsed times. Is the difference of the elapsed times bigger or smaller than in the `3-3loc1.c`

[4] You can increase the number of iterations in the same factor you decrease the size of the matrix.

and `3-3loc3.c` programs? Why do you think the elapsed time is not the same although the matrix in both programs can be completely stored in the cache memory?

3. Disabling the cache memory

Sometimes it is necessary to disable the cache memory partially or completely. For instance, when we want to test the operation of the main memory. A typical checking algorithm will write bit sequences in different memory locations and then these locations are read to check whether the read values match the written ones. If the cache memory were available the main memory would not be checked since many write and read operations would be performed over the cache memory rather than over the main memory.

In other occasions it is interesting to prevent some address ranges from being cached, such as those occupied by peripheral interfaces with DMA capabilities.

In this lab session the cache will be disabled for some address ranges to show the great influence of the cache memory in the performance of the system. Disabling the cache will be carried out by means of a driver, developed by the course teachers, which aims to disable the cache for 4 KiB memory pages. The driver functionality is accessed using the following function:

```
int cachedis(unsigned int dir);
```

defined in the `atc/lincache.h` header file. `cachedis` receives a memory address as an argument and it disables the cache for the 4 KiB page in which the address is contained. The returned value is 0 on success, and a negative integer otherwise. For example, if the function is called as

```
cachedis(0x12345678);
```

the cache memory is disabled for the memory address range `12345000h-12345FFFh`.

- ❑ Copy to your working folder the file `3-3withcache.c`, which contains the following code:

```c
#include <stdlib.h>
#include <stdio.h>

#define NTIMES 50000 // Repeat 50000 times

char page[4096]; // 4096 bytes

int main(void) {
  int rep, i;

  for (rep = 0; rep < NTIMES; rep++) {

    // Print message each 1000 iterations
    if (rep % 1000 == 0) {
      printf("N= %d\n", rep);
      fflush(stdout);
    }

    // Access to all page locations
    for(i = 0; i < 4096; i++)
      page[i] = 'A';

  }

  exit(0);
}
```

❏ Compile and link the above file. Run the program with the command;

```
$> time ./3-3withcache
```

What is the elapsed time of the program?[9] Give your answer as the average of three consecutive runs.

[9]

❏ Edit the `3-3withcache.c` file and add the following statements just before the outer loop. These statements disable the cache in the accesses to the data page.

```c
// Disable the cache memory for the data page
if (cachedis((unsigned int)page) < 0) {
  fprintf(stderr, "Error while trying to disable the cache\n");
  exit(-1);
}
```

It is also necessary to add the directive `#include <atc/lincache.h>`. Save the new version of the program as `3-3withoutcache1.c`.

❏ Compile and link the new version of the program with the following command:

```
$> gcc 3-3withoutcache1.c -o 3-3withoutcache1 -lcache
```

❏ Run the program three times with the following command:

```
$> time ./3-3withoutcache1
```

What is the average elapsed time of the program?[10] What is the speedup in the elapsed time compared to the elapsed time of the cache-enabled version of the program?[11]

[10]

[11]

❏ Edit the `3-3withoutcache1.c` file and add the following statements just before the outer loop. These statements disable the cache memory in the accesses to the program stack.[5] Save the new version of the program as `3-3withoutcache2.c`.

```
// Disable the cache memory for the accesses to the program stack
if (cachedis((unsigned int)&i) < 0) {
  fprintf(stderr, "Error while trying to disable the cache\n");
  exit(-1);
}
```

❏ Compile and link the program. Run it with the following command:

```
$> time ./3-3withoutcache2
```

What is the elapsed time of the program?[12] What is the speedup in the elapsed time compared to the elapsed time of the cache-enabled version of the program?[13]

12

13

❏ Edit the `3-3withoutcache2.c` file and replace the `i` variable by the `rep` variable in the following statement:

```
if (cachedis((unsigned int)&i) < 0) {
```

❏ Save the new version of the program as `3-3withoutcache3.c`. Compile, link and run it as usual. The number of accesses to the variable `rep` is smaller than the number of accesses to the variable `i`; however, you should see that the elapsed time is similar to `3-3withoutcache2`. How can you explain this?

4. Exercises

✏ Edit the `3-3withoutcache2.c` file and add the following statements just before the outer loop. These statements disable the cache memory for the memory accesses to all the code of the `main` function.[6] Save the new version of the program as `3-3withoutcache4.c`.

[5] It is assumed that all the program stack can be contained in a 4 KiB memory page, which is true in this case.

[6] It is assumed that the `main` function is contained in only one 4 KiB memory page, which is true in this case.

```
// Disable the cache memory for the code page
if (cachedis((unsigned int)main) < 0) {
  fprintf(stderr, "Error while trying to disable the cache\n");
  exit(-1);
}
```

✏ Compile, link and run the program three times as usual with the following command:

```
$> time ./3-3withoutcache4
```

What is the elapsed time of the program?[14] What is the speedup in the elapsed time compared to the elapsed time of the cache-enabled version of the program?[15]

14

15

SESSION 4

Task memory organization in Linux

Objectives

The main objectives of this lab session are:

- Show the organization of the virtual address space of Linux tasks.
- Analyze the independence of the virtual address spaces of tasks.
- Analyze the changes produced in the address spaces of the tasks during memory allocation and release in Linux.

Previous knowledge and required materials

In order to get the maximum benefit from this lab session, the student is required to:

- Understand the basic principles of virtual memory.
- Attend the lab session with a copy of the DVD with the tools provided for the course. This DVD contains a modified 32-bit version of the GNU/Linux operating system with the `linmem` module installed.
- Attend the lab session with a USB memory stick to store the files generated.

Session development

This lab session is divided in several sections. First, information about the organization of the virtual address space in Linux is provided. Second, several tests will be carried out to understand the concepts of virtual address and physical address in different scenarios. Finally, the changes produced in the address spaces of the tasks during memory allocation and release in Linux will be studied.

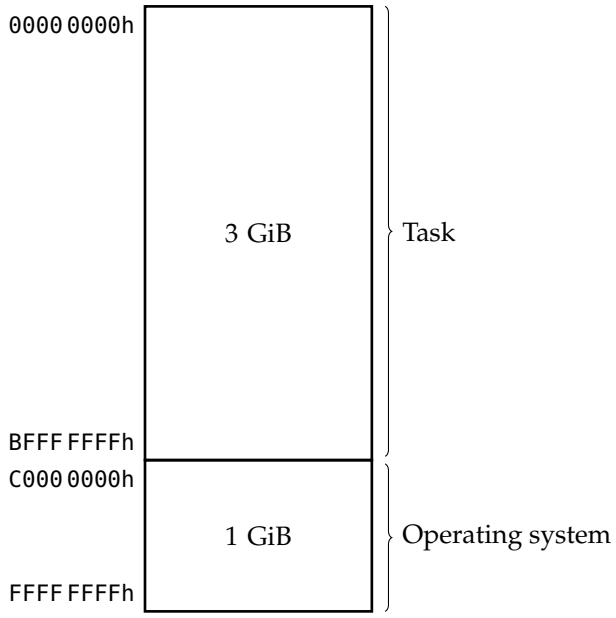


Figure 4.1: Organization of the virtual address space of a task in Linux

1. Organization of the virtual address space in Linux

Linux associates a virtual address space of $2^{32} = 4$ GiB to each task.[1] The highest GiB of virtual memory is reserved for the OS. The lowest 3 GiB of virtual memory are available for the task. This situation is shown in figure 4.1.

Only a portion of the 3 GiB of virtual memory associated with the task have physical memory associated. Thus, not all the addresses are accessible.[2] The virtual addresses that do not have memory associated should not be accessed since a non-recoverable page fault will be generated.[3] Furthermore, there are addresses associated with the task that have special purposes. For instance, the `NULL` constant in C is used to refer uninitialized pointers and is implemented in the C standard library by assigning the `0000 0000h` value to it. Since the Linux memory manager never associates memory with the virtual page containing this address, any read or write operation with a pointer containing the `NULL` value will generate a non-recoverable page fault. This is useful when debugging programs since a programming error consisting on writing something in memory with an uninitialized pointer would provoke the uncontrolled writing of a variable very difficult to determine.

As shown in the above figure, the task can only gain access to the range `0000 0000h-BFFF FFFFh`. Now, we will study how the memory is organized inside this range.

[1] This is the case for 32-bit versions of the OS. In this lab session we refer to a 32-bit version of Linux by default.

[2] In this lab session the term memory will be used to refer physical storage, although they are not the same. It must be taken into account that the physical storage associated with a task is divided into the physical memory and the page file (or swap partition).

[3] Usually, in Linux an error message like "Segmentation fault" will be displayed.

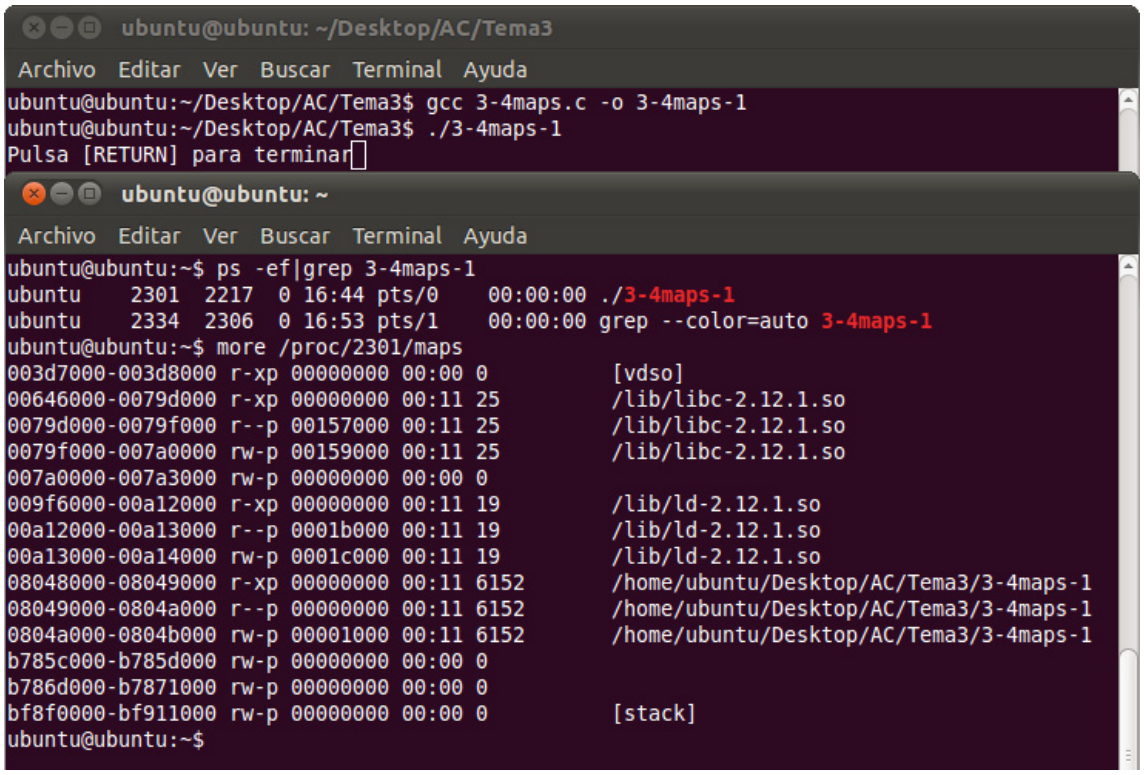


Figure 4.2: Example of `proc/pid/maps` file in Linux.

The organization of the virtual address space accessible by a task can be found in the `/proc/pid/maps` file, where `pid` is the process (task) identifier. To see the content of this file, shown in figure 4.2, follow the next steps:

- ❏ Compile and link the `3-4maps.c` file using the following command:

  ```
  $> gcc 3-4maps.c -o 3-4maps-1
  ```

  This program waits for the user to press any key and then finishes. This allows us to get its `pid` and see its `maps` file before it finishes.

- ❏ Run the `3-4maps-1` executable file from the command-line interface. Using a second command-line interface execute the following command to see its process identifier: it is the number in the second column.

  ```
  $> ps -ef|grep 3-4maps-1
  ```

- ❏ In the second command-line interface, execute the command `more /proc/pid/maps`, substituting `pid` by the process identifier previously obtained. Once this step is done you can press `ENTER` in the first command-line interface to finish the program.

The first column of the `/proc/pid/maps` file shows the memory ranges. The first address of the range is the start of the range, whereas the second is the end

of the range (plus one). The size of each of the ranges is multiple of 4 KiB, the page size. This is because the memory is assigned in page units. In the second column the permissions associated with the memory area are shown: `r`, read, `w` write, and `x` execution. It is also shown whether the memory area is private, `p`, or shared with other tasks. Finally, in the column on the right, the type of memory area is shown, or the file name associated with the memory area if there is any.

In general, the following memory areas can be distinguished:

- Static code. Includes the whole machine code of the task in its executable file. It does not include the code of the libraries with which it is linked during execution: dynamic link libraries. In the above figure this range is the one associated with the `3-4maps-1` file with attributes `r-xp`.
- Static data. Includes the global variables of the task, defined also in the executable file. In the above figure this area is the one associated with the `3-4maps-1` file with attributes `rw-p`.
- Stack. This area stores the local variables of the task and the return addresses of functions, among other things. In modern Linux distributions the location of the stack is not selected by the development tools but it is assigned by the OS in the loading process.[4] In the above figure this area is the one associated with the `[stack]` label.
- Heap. Memory area reserved for dynamic allocation. Reserving memory in the heap usually requires the use of the C `malloc` function. In the `3-4maps-1` task there is no heap.
- Dynamic link libraries. The executable file of the task contains code which interfaces with dynamic libraries. The linking process with these libraries is carried out in the loading process (when the program is launched) or in execution time.[5] Usually, dynamic libraries in Linux have the `.so` extension. In the above figure the `libc-2.12.1.so` library, the C standard library, is shown. In addition, the `ld-2.12.1.so` library can be seen. This is the dynamic library required to link the program with the rest of dynamic libraries. What is the total amount of bytes of the C standard library in its dynamic version?[1].

  1
- *Virtual Dynamic Shared Object* (VDSO). It is a dynamic memory area shared among all the tasks to perform quick service calls using the `sysenter` instruction.[6]

[4] The objective of this strategy is to hinder the attacks based on overflowing the stack of the task.

[5] This scenario is called dynamic loading and allows to generate the executable file even without the existence of the libraries.

[6] The service calls used to be carried out using the `int 80h` instruction in Linux. Nevertheless, this mechanism is very slow in modern processors. Thus, Intel developed a new mechanism based on the `sysenter` instruction. AMD processors have an analogue instruction: `syscall`.

The `3-4maps-1` program uses the C standard library in its dynamic version. The reason is that the `gcc` compiler links the object code by default with this version of the library. If the static version of the C standard library is preferred, the `-static` option must be used during compilation.

- ❏ Compile and link the `3-4maps.c` program using static libraries with the following command:

  ```
  $> gcc -static 3-4maps.c -o 3-4maps-2
  ```

- ❏ Run the executable file, analyze its `maps` file, and observe the changes compared to the dynamic version using figure 4.2 as reference. What is the C library used?[2] What is the type of this library?[3]

- ❏ Compare the size of the `3-4maps-1` and `3-4maps-2` executable files using the following command:

  ```
  $> ls -lh 3-4maps*
  ```

  What is the difference in KBytes?[4] `3-4maps-2` is bigger because, as the standard library was linked statically, code from the library has been added to the code of the program.

2

3

4

The organization of the system physical address space can also be observed. In this case, the file containing this information is `/proc/iomem`. In this file, the peripheral device interfaces allocated in the address space can also be seen.[7]

2. Analysis of the address space of a task

The virtual and physical addresses of the following elements of a task will be analyzed:

- Global variables.
- Local variables.
- Functions.

Retrieving the physical addresses requires accessing the page table of the task, a data structure which is not accessible by the task since it is stored in the address space of the OS, that is, in the range `C000 0000h`–`FFFF FFFFh`. The physical addresses will be obtained using the `linmem` function, which uses a driver with the same name. The `linmem` driver allows us to access this range since it is part of the OS and thus is executed with superuser privileges. The task asks the driver for a physical address associated with a virtual address, and the driver provides this value.

[7] The location of the peripheral interfaces in the input/output address space can be seen in `/proc/ioports`.

- ❏ Edit the `3-4proclinux-1.c` file to see its content. The program includes the header file `atc/linmem.h` to use the `linmem` function. This function is implemented in the `libmem.a` static library, contained in the DVD provided in this course. The `print_virtual_physical_data` function uses the `linmem` function to print a virtual address on the screen, its associated physical address, and the 32-bit data item stored from this address.

  The `3-4proclinux-1.c` program declares a local variable and a global variable.[8] Furthermore, this program displays on the screen the virtual and physical addresses associated with the local variable, the global variable and the `print_virtual_physical_data` function, as well as the content of these addresses.

- ❏ Open a command-line interface. Compile and link the above source file with the following command:

  ```
  $> gcc -Wall 3-4proclinux-1.c -o 3-4proclinux-1 -lmem
  ```

  The `-Wall` option asks the compiler to display all warnings. If everything is correct, the `3-4proclinux-1` executable file is generated.

- ❏ Run two instances of the executable file simultaneously using two command-line interfaces. Answer the following questions: Are the virtual addresses of the local and the global variable the same in the two instances of the program? Why? And the physical addresses?

- ❏ Are the virtual addresses of the `print_virtual_physical_data` function the same in the two instances of the program? What happens with the physical addresses? Can you explain this situation?

[8] Local variables are stored in the stack of the task, whereas global variables are stored in the data section of the task.

3. Dynamic memory allocation

When a task is launched the operating system assigns the required memory to store its data and code sections. Nevertheless, in many occasions during the development phase of the program, the memory requirements are not known since they may depend, for instance, on the data items to be processed. Data structures with enough capacity could be defined in the program for the worst case. However, this strategy implies a great waste of memory.

To avoid this issue, the operating system provides services for the tasks to allocate and release memory when they are being executed.

The operating system establishes restrictions over the maximum amount of memory that can be assigned to a task, as well as the proportion of this memory in the physical memory and in the page file (or the swap partition).

The memory requested by a task to the operating system is associated with a virtual address range that can be specified by the task when calling the service, although the programmer usually delegates this task to the operating system.

Two different methods of asking for memory to the operating system are studied. The first is the most complex but also the most powerful method. It is based on calling an API specific service of the OS. The second method involves calling a C standard function, that is, a user function.

3.1. The POSIX `mmap` service

This service is used to request memory in the Linux operating system, since Linux implements the POSIX interface.[9] Also, it allows mapping files in memory, but this functionality will not be used in these labs.

Before calling the `mmap` function, a memory descriptor must be created and configured with the size of the memory to be requested. The following code snippet shows how to ask for a 16 KiB area memory.

[9] POSIX is a family of standards for interfaces of the operating system. Its main objective is to maintain the compatibility between operating systems.

```
char string[30]; /* String with the name of the memory area */
void *p;         /* Pointer to the requested memory area */
int memd;        /* Descriptor of the requested memory area */

/* Opens a descriptor of the memory area which depends on the PID */
sprintf(string, "/mem-%d", getpid());
if ((memd = shm_open(string, O_RDWR|O_CREAT|O_EXCL, S_IRWXU)) < 0)
  return(-1); /* If an error occurs the task is finished */

/* Defines the size of the memory area */
if (ftruncate(memd, 16*1024) < 0)
  {
    shm_unlink(string); /* Closes the memory descriptor */
    return(-1); /* If an error occurs the task is finished */
  }

/* Maps the memory area in the virtual address space of the task */
if ((p = mmap(NULL, 16*1024, PROT_READ|PROT_WRITE,
              MAP_SHARED, memd, 0)) == MAP_FAILED)
  {
    shm_unlink(string); /* Closes the memory descriptor */
    return(-1); /* If an error occurs the task is finished */
  }
```

The `shm_open` function creates a memory descriptor with the given name[10] and other arguments. The name must start with the character '`/`'.

The `ftruncate` function configures the memory descriptor with a size of 16 KiB. Once executed, the memory descriptor is configured, but the task has not received the 16 KiB of memory yet.

Next, the arguments passed to the `mmap` function, as well as the returned value are described.

- `NULL`. Indicates that the operating system chooses the starting virtual address of the memory to be provided to the task. If an address is specified, the operating system will try to allocate memory from this address.
- `16*1024`. Indicates the size of the memory area to allocate. It does not have to match the value used in the call to `ftruncate` since, for instance, 4 KiB could be requested first and the rest 12 KiB could be requested later.
- `PROT_READ|PROT_WRITE`. Indicates that the memory area assigned by the operating system will have read and write permissions. In this case it would not be possible to run instructions from this memory area.

[10] The name has been chosen depending on the `pid` to avoid errors when a memory mapped area is not released before finishing the program. When this happens, the memory area is automatically released by the system, although not immediately but several minutes after its reference has been lost.

- `MAP_SHARED`. Indicates that the memory area can be accessed from other tasks. Therefore, this option allows to share the memory area.[11] If sharing the area is not desired, the `MAP_PRIVATE` option must be used.
- `memd`. Memory descriptor previously created and configured.
- `0`. Specifies that the requested memory area must be located from the first memory address associated with the memory descriptor. For instance, if in a first call to `mmap` 4 KiB are requested, this argument should be zero. In a second call to `mmap` to request the rest 12 KiB this argument should be 4×1024.
- If the function succeeds, the returned value is the virtual address from which the operating system has mapped the requested memory. Otherwise, the returned value is `MAP_FAILED`.

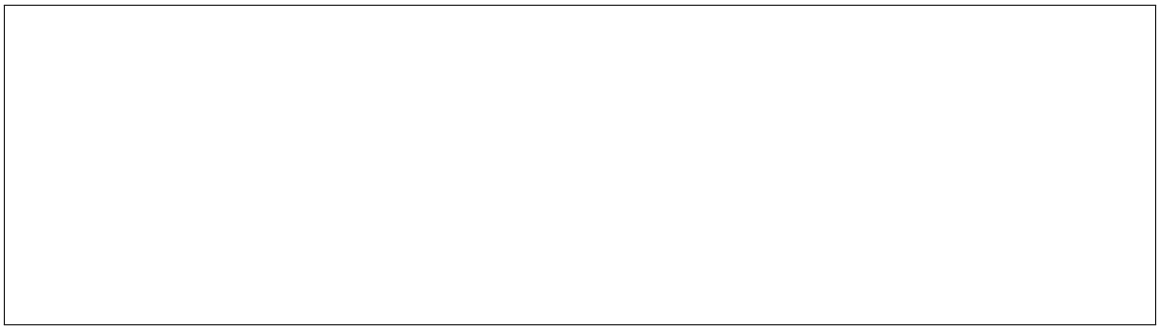
You can use `man mmap` to get more information from the manual pages of this function.

❏ Edit the `3-4proclinux-2.c` file to see its content. As you can see, the program requests 16 KiB of memory to the operating system and prints on the screen the virtual and the physical addresses associated with the first address of the requested memory area.

❏ Compile and link the program with the following command:

```
$> gcc -Wall 3-4proclinux-2.c -o 3-4proclinux-2 -lrt -lmem
```

The `-lrt` option is used to link with the real-time library, where the `shm_open()` and `shm_unlink()` functions are defined. Run the executable file. What happens with the physical address where the requested memory area starts after the memory has been requested? How can you explain this behaviour?



This message is shown because the operating system has not yet given a physical page for the page because it has not been accessed. Let us check if the message disappears after accessing the address.

[11] The other tasks accessing this memory area must create their descriptors with the same name and without the `O_CREAT` option.

- ❏ Edit the `3-4proclinux-2.c` file and then save it with the name `3-4proclinux-3.c`. Just after calling the `mmap` function add the necessary code to write a character in the first address of the memory area provided by the operating system. To do so, the following instruction can be used: `*(char *)p = 'A'`.
- ❏ Compile the `3-4proclinux-3.c` source file and run the executable file. What happens now with the physical address after the memory request and the writing operation? How can you explain this behaviour?

3.2. The POSIX `munmap` service

This service requests the operating system to release a memory area reserved previously.

- ❏ Edit the `3-4proclinux-3.c` file and then save it with the name `3-4proclinux-4.c`. Add the following statement just after the instruction writing in the first address of the memory area.

  ```
  munmap(p, 16*1024);
  ```

  This instruction releases the memory requested previously. The first argument identifies the starting virtual address of the memory area to be released. The second argument identifies the size of the area to be released. It is not necessary to release the whole memory area at the same time.
- ❏ Compile the `3-4proclinux-4.c` source file and run the executable file. What happens with the physical address after releasing the memory area? Why?

3.3. The `malloc` and `free` functions

The C language defines two standard functions, called `malloc` and `free`, to request and release dynamic memory. These functions are less powerful than the `mmap` and `munmap` POSIX functions, but they are easier to use. Furthermore, they have the advantage of being implemented in almost all platforms.

The `malloc` implementation depends on the operating system, although it usually uses the heap to allocate the requested memory area. In this scenario the memory allocation is very fast since no service call is required.[12]

The `malloc` function receives the number of bytes of the memory area to be requested as an argument and returns the virtual address where the area starts. If an error occurs, the returned value is `NULL`.

Usually, in Linux operating systems if the requested memory area is below a threshold the memory allocation is done in the heap. If the requested memory is above this threshold the `malloc` function calls the `mmap` service internally.

Once the requested memory area is no longer needed it must be released. The `free` function is used to carried out this task. This function receives the virtual address returned by the `malloc` function as an argument.

- ❏ Edit the `3-4proclinux-5.c` file to see its content. This program requests 16 KiB of memory to the operating system by means of the `malloc` function. It also prints the virtual and physical addresses where this area starts.
- ❏ Compile and run the program. The program will not finish until you press a key in the command-line interface. This will allow you to inspect its `maps` file. What is the virtual address associated with the memory area requested?[5] What is the physical address associated with the physical area requested?[6] Yoy may notice how the physical memory has been assigned without previously accessing this memory, conversely to the behaviour seen above with `mmap`.
- ❏ Observe the content of the `mmaps` file of the above task from another command-line interface. In what memory area of the task is located the requested memory area?[7] Observe also how the location in the virtual address space is chosen by the operating system, and it is returned by the `malloc` function.
- ❏ Edit the `3-4proclinux-5.c` file again and save it with the name `3-4proclinux-6.c`. Modify the latter to ask the operating system for 256 KiB of memory. Compile, link and run the program. Observe the content of the `mmaps` file. What has happened? Why?

5

6

7

[12] Calling a service of the operating system involves starting and finishing operations, such as saving and recovering the state of the task. However, this time can be negligible compared to the execution time of the task.

4. Exercises

- ❏ Run the `3-4proclinux-3` program again.
- ❏ List the content of its `maps` file and locate the memory area requested to the operating system. You will notice that this area has the `rw-s` attributes. The `s` indicates that it is a shared memory area. This is because the `mmap` function has been called with the `MAP_SHARED` argument.
- ❏ Using `3-4proclinux-3.c` as a reference, write a new program, `3-4proclinux-7.c`, which shares the memory area created by the program `3-4proclinux-3` and prints the character stored in the first location of the requested memory area. You need to use the same string for the memory area descriptor in the new program and you also need to remove the `O_CREAT` argument since the memory area has been already created. To know the exact name of the string you can list the content of the `maps` file of `3-4proclinux-3`. You can use the following snippet as a reference substituting the number `1234` by the `pid` of `3-4proclinux-3`.

```
/* Opens a descriptor of the memory area depending on the PID */
if ((memd = shm_open("/mem-1234", O_RDWR|O_EXCL, S_IRWXU)) < 0)
  return(-1); /* If an error occurs the task is finished */
```

  Furthermore, you must substitute the statements which write a character in the shared memory area by the statements required to read this character. For instance, you can use the following snippet:

```
/* Reads a character from the first location of the
   shared memory area */
printf("The character written by the other task is: %c\n",
       *((char *)p));
```

- ❏ Compile and link the program. Run the `3-4proclinux-7` executable file without finishing the execution of `3-4proclinux-3` for both programs to share memory. Does the character printed by `3-4proclinux-7` match the character written by `3-4proclinux-3`?[8]
- ❏ What happens with the physical addresses of the shared memory area shown by both tasks?

[8]

❑ Are the virtual addresses of the shared memory area the same in both tasks?[9]

[9]

5. Appendix: The `linmem` module

The development of this session requires accessing the page directory and tables of a task during execution time. These are data structures managed by the operating system and located in virtual pages accessible via supervisor level.

Getting the physical address associated with a virtual address, as well as its content, is not a trivial problem.

- Obtaining the physical address associated with a virtual address of a task requires accessing the appropriate page directory and page table entries. These entries are stored in memory out of the accessible range of the task.
- Obtaining the content of a virtual address in the range `C000 0000h`–`FFFF FFFFh` is not possible in a Linux task since this virtual address range is marked with the highest privilege and can only be accessed by the operating system.

To overcome the above issues a module for the operating system has been developed: `linmem`. This module allows us to access any location of the virtual address space of a task. Since this type of controller is executed in supervisor mode, with the highest privilege, it can gain access to any location of the virtual address space.

A task may communicate with this module in a very easy way. It only has to use the `libmem.a` library, which exports the `linmem` function (with the same name as the module). Using this function requires to include the `atc/linmem.h` header file in the source file and link it with the `-lmem` option.

Given a virtual address, the `linmem` function provides the following information: the physical address associated with the virutal address, the virtual address of its page directory entry (PDE) and its content, the virtual address of its page table entry (PTE) and its content. The prototype of this function is the following:

```
extern int linmem(unsigned int virtual_addr, unsigned int *physical_addrp,
       unsigned int *data_itemp, unsigned int *pde_addressp, unsigned int *pdep,
       unsigned int *pte_addressp, unsigned int *ptep);
```

- `virtual_addr` is the virtual address to be analyzed.

- `physical_addrp` is a pointer to the variable where the function writes the physical address associated with the virtual address. If this argument is `NULL`, the function does not provide this value.
- `data_itemp` is a pointer to the variable where the function writes the content of the virtual address. If this argument is `NULL`, the function does not provide this value.
- `pde_addressp` is a pointer to the variable where the function writes the virtual address associated with the PDE of the virtual address. If this argument is `NULL`, the function does not provide this value.
- `pdep` is a pointer to the variable where the function writes the content of the PDE of the virtual address. If this argument is `NULL`, the function does not provide this value.
- `pte_addressp` is a pointer to the variable where the function writes the virtual address associated with the PTE of the virtual address. If this argument is `NULL`, the function does not provide this value.
- `ptep` is a pointer to the variable where the function writes the content of the PTE of the virtual address. If this argument is `NULL`, the function does not provide this value.

If the function succeeds, the returned value is `0`. Otherwise, it returns `-1` when the virtual address is not in memory and `-2` when a communication problem with the `linmem` module occurs.

The installation of this module is not required if you are using the course Live-DVD. If you wish to install this module in other systems, you must follow the instructions of the distribution to install modules in the kernel.

SESSION 5

The page table in Linux

Objectives

The main objective of this lab is to understand the operation of the paging technique in a real computer architecture, and thus, in its operating system. The x86-32 architecture (also called IA-32) is used, as well as a 32-bit Linux operating system.[1]

Previous knowledge and required materials

In order to get the maximum benefit from this lab session, the student is required to:

- Understand the basic operation of paging in x86-32.
- Attend the lab session with a copy of the DVD with the tools provided for the course.
- Read the documentation provided by Intel about the operation of the paging technique in the x86-32 architecture, included in the appendix A.

1. Introduction to the paging technique in Linux on the x86-32 architecture

The memory management in the x86-32 architecture is quite complex compared to other architectures. Several paging schemes can be used, some of then simultaneously in the same system. In the lectures, the paging mechanism more commonly used is studied. This is based on two levels of translation over a physical address space of 32 bits introduced with the 80386 processor. However, the x86-32 architecture provides other possibilities:

[1] The x86-64 architecture (AMD-64) together with a 64-bit Linux operating system are not used due to its complexity, since this architecture uses a four-level address translation.

- With the release of the Pentium processor the paging technique is extended to allow pages of 4 MiB. To do so, the virtual addresses are divided in two fields instead of three. The PTI (page table index) is removed and its bits are moved into the offset field. Thus, a 22-bit offset is obtained.[2] Therefore, only one level is used in the page table, and the frame is provided by the PDE (page directory entry).

- The Pentium Pro extends the x86-32 architecture to operate with 36-bit physical addresses, addressing up to 64 GiB of physical memory. To use this extension, the PAE (Physical Address Extension) mode of the processor must be activated. This requires setting a specific bit of a control register of the CPU. In this mode, the processor can use two virtual page sizes: 4 KiB and 2 MiB. In the first case the paging technique uses three levels of translation, whereas in the second it uses two levels.[3]

  The PAE mode can be activated by compiling the source code of the Linux kernel with the `CONFIG_HIGHMEM64G` option enabled. The CPU must support this mode.[4] Many of the current 32-bit Linux distributions oriented to server computers have the PAE mode activated by default.

- With the Pentium III a new paging mode is added, called PSE-36 (Page Size Extension). This mode allows 36-bit physical addresses, and thus, the CPU can address up to 64 GiB. To use this extension the PAE mode must be disabled and a new bit of the control register of the CPU must be activated. This mode operates with only one level of translation and 4 MiB pages. However, this mode is rarely used. In fact, the Linux kernel does not provide support for it.

- When the PAE mode is disabled (and also the PSE-36), as it happens in the 32-bit desktop Linux distributions, there are two different paging schemes: one using 4 KiB pages and two levels of translation, and another that uses 4 MiB pages and simple translation. The appendix A contains an in-depth explanation of these two modes.

This lab session is carried out on a PC running a 32-bit desktop Linux distribution, which does not use the PAE mode. Thus, 4 KiB and 4 MiB pages can be used. Using 4 KiB pages has advantages and drawbacks compared with the use of 4MiB pages. For instance, a task using 3 KiB wastes 1 KiB with 4 KiB pages and almost 4 MiB with 4 MiB pages. However, since an operating system like Linux requires a big amount of memory, the use of 4 KiB pages would require a large number of global entries in the TLB. Thus, the TLB hit rate would be reduced for the page table entries of the tasks and the access time to the memory system would be increased. The x86-32 architecture provides two TLBs: one for 4 KiB pages and another one for 4 MiB pages.

[2] 22 bits are required to address a 4 MiB virtual page: 2^{22} Bytes $= 4$ MiB.

[3] The entries in the page directory and in the page table occupy 64 bits in the PAE mode. Thus, there are 512 entries in each directory and in each page table.

[4] The support of the PAE mode by the CPU can be checked by listing the content of the `/proc/cpuinfo` file.

As mentioned above, with 4 MiB pages the PTI field does not exist and the offset field increases from 12 to 22 bits, as shown in the figure 3-13 of the appendix A. To distinguish when an entry in the page directory points to a 4 MiB or a 4 KiB page, the PS bit (bit 7) must be observed. If this bit is set, the entry of the page directory points to a 4 MiB page. In this case, the entry of the page directory provides the 10 bits (bits 22 to 31) identifying the 4 MiB page frame.[5]

Regardless of the page size used, the operating system must program the page directory and the page table entries. This means that the operating system has to be able to read and write any entry. The operating system, as any other program, uses virtual addresses. To gain access to the page directory and to the page tables of a task, the operating system must gave them virtual addresses. These virtual addresses must belong to the memory range of the operating system to prevent a user task from accessing its own page directory and page table.

To obtain a physical address and the content associated with a virtual address, as well as the physical and virtual addresses of its PDE and PTE, the `linmem` function is used. This function was described in the previous lab session. Its prototype is defined in the `atc/linmem.h` header file.[6]

Session development

Once the details of the paging technique in Linux have been described, they will be shown in a program compiled from the following three files: `3-5paging.c`, `3-5pagetable.c` and `3-5pagetable.h`.

The `3-5paging.c` file contains the `main` function of the program and outputs the following information:

- The virtual address, the physical address and the content of an element of an integer array. Furthermore, it provides the virtual address, the physical address and the content of the PTE and PDE associated with this element.
- The PDE and PTE of all the pages of the task that are in physical memory before requesting more memory to the operating system.
- The virtual address where the memory requested to the operating system has been allocated.
- The PDE and PTE of all the pages of the task that are in physical memory after receiving the memory requested to the operating system.
- The PDE and PTE of the virtual pages where the interface of a peripheral device is mapped.
- The PDE and PTE of the pages of the operating system that are in physical memory.

[5] The initial physical address of the page is calculated by combining the page frame and all the bits of the offset field to `0`.

[6] The student interested in additional details about the paging technique in Linux may look for information about the `mm_struct` defined in the source code of the kernel.

The appendix B contains an example of the output file generated by the program. You may use this as a reference to check the correct operation of the program.

You must complete the gaps in the `3-5pagetable.h` and `3-5pagetable.c` files before compiling the program.

- ❏ Edit the `3-5pagetable.h` file and write the value of your personal ID (Spanish DNI) without the letter.
- ❏ Copy the content of the `iomem` file to the `iomem.txt` file by means of the following command:

  ```
  $> cp /proc/iomem iomem.txt
  ```

  Save the `iomem.txt` file, since it is one of the files that you must deliver at the end of the lab.
- ❏ Write the lowest and highest (minimum and maximum) addresses of the audio interface. To get these addresses you may use the following command:

  ```
  $> cat iomem.txt | grep audio
  ```

The `3-5pagetable.c` contains two functions that are called from the main function of the program to display all the requested information on the screen. The `WriteMemoryPages` function contains several gaps that must be filled. The prototype and description of this function are:

```
int WriteMemoryPages(void * min_virtual_addr,
   void * max_virtual_addr, void * min_physical_addr,
   void * max_physical_addr);
```

Prints in the standard output the PDE and PTE of the virtual pages in physical memory which include any of the virtual addresses in the range [`min_virtual_addr`, `max_virtual_addr`] and whose physical addresses are in the range [`min_physical_addr`, `max_physical_addr`].

- `min_virtual_addr` and `max_virtual_addr` define the virtual address range to be analyzed. There are occasions in which we do not want to analyze the complete 4 GiB address space, but only a range inside it. The function must provide the information of the virtual pages in memory in the range of virtual addresses specified.
- `min_physical_addr` and `max_physical_addr` define the range of physical addresses to be analyzed.

The function must take also into account that 4 MiB pages can be encountered. In these circunstances the PDE will exist but not the PTE, as it happens in the page `302h` of the example shown in the appendix B.

The `WriteMemoryPages` function calls the `linmem` function to get this information. More information about the `linmem` function can be obtained in the appendix A of the previous lab session.

Next, the gaps of this function will be filled.

- ❏ You must fill the gaps labeled as `--(1)--` and `--(2)--` taking into account that the page directory indexes (PDI) must vary between a minimum and a maximum covering all the virtual addresses of the range.
- ❏ The lowest virtual address with the PDI contained in the `pdi` variable can be obtained by shifting the `idp` value to the left a specific number of bits. This number must be specified in the `--(3)--` gap.
- ❏ Calling the `linmem` function provides the PDE associated with the PDI of value `pdi`. To get the PDE, the function must be called using a virtual address whose associated PDI matches `pdi` (for instance, the virtual address calculated above) and the rest of the arguments set to `NULL` except the one which provides the PDE. Fill the `--(4)--` gap for the PDE to be written in the `pde` variable. You need to review the `linmem` documentation provided in the previous lab session.
- ❏ The `--(5)--` gap requires a bit mask to be applied to the PDE value to determine whether the present bit is set. The bit mask will have all the bits set to zero except the bit to check.
- ❏ To check whether the PDE points to a page table or to a 4 MiB page it is necessary to check one of its bits. A mask will be required again with all the bits set to zero except the one to check. This mask must be written in the `--(6)--` gap.
- ❏ When the size of the page is 4 MiB, its final address is obtained by adding a specific value to the initial address. Write the required expression to calculate this final address in the `--(7)--` gap.
- ❏ The initial physical address of the 4 MiB page is obtained from its PDE, provided by the `linmem` function. To do so, a mask must be applied on the PDE. In this mask the bits set to one are those that match the frame in the PDE. Write the required expression in the `--(8)--` gap.
- ❏ The final physical address of the 4 MiB page is obtained in the same way as the final virtual address, since the size of the physical page is the same as the size of the virtual page. You must write the necessary expression in the `--(9)--` gap.
- ❏ If the PDE does not point to a 4 MiB page but to a page table, it is necessary to iterate over all the entries of the table to locate those pointing to pages in memory. Iterating over the page table means iterating over all the page table indexes, that is, all page entries. The minimum and the maximum values of the PTI must be inserted in the `--(10)--` and `--(11)--` gaps, respectively.
- ❏ To obtain the lowest virtual address of a 4 KiB page given a PDI and a PTI it is necessary to shift the PDI and the PTI fields to the left and combine the result. You must introduce the values for these shifts in `--(12)--` and `--(13)--`.

- ❏ Obtaining the PTE stored in the page table for the `pti` entry requires using the `linmem` function again. In this case the virtual address must be the initial address of the 4 KiB page with the rest of the arguments set to `NULL` except for that for the PTE, which is stored in the `pte` variable. The required arguments must be inserted in `--(14)--`.
- ❏ To check if the 4 KiB page is in memory, a mask must be applied to the PTE to check the value of the presence bit. The necessary expression must be written in `--(15)--`.
- ❏ Finally, the final virtual address of the 4 KiB page must be calculated, as well as its initial and final physical addresses. The procedure is the same followed in the 4 MiB pages, but taking into account the difference in the page sizes. Furthermore, the initial physical address is obtained from the PTE and not from the PDE. Fill the `--(16)--`, `--(17)--`, and `--(18)--` gaps.

Once all the gaps have been completed you may compile the program and generate the executable file. You must answer the following questions after running the program. You must write the answers to these questions in a text file called `answers.txt`, with the following format:

```
(1)  15320h
(2)  523
(3)  An answer containing text which
     requires one or more lines in the file
```

Each of the above answers describes an answer type. **It is very important that you follow the above format.** The `answers.txt` file will be used to assess your performance in this lab session, and an invalid format will not allow its verification. The first line of the above example contains the answer to a question asking for a hexadecimal value. You can notice that only the hexadecimal value has been written, without any text. The second line contains the answer to a question asking for a decimal value. Finally, the third line contains an answer which may occupy several lines of the file.

It is important to remember that the size of the RAM memory is expressed in powers of two. For instance, 1 KiB $= 2^{10}$ Bytes and 1 MiByte $= 2^{20}$ Bytes.

- ❏ Compile and link the above program using the following command:

  ```
  $> gcc 3-5paging.c 3-5pagetable.c -o 3-5paging -lmem
  ```

- ❏ **Run the executable file in the PC of the labs** with the following command:

  ```
  $> ./3-5paging > output.txt
  ```

The > character redirects the standard output to the `output.txt` file.

- ❏ Edit the `output.txt` file to see its content. At this moment, you must focus on the first part of this file, that shows the virtual and physical address, its content, and the PTE and PDE of an element of the array.
- ❏ Do you think that the virtual address of this element of the array will vary from execution to execution of the program? Why?[1] Answer with `(3)` format.
- ❏ Do you think that the physical address of this element of the array will vary from execution to execution of the program? Why?[2] Answer with `(3)` format.
- ❏ Check your previous answers running the program again, but this time using the command:

  ```
  $> ./3-5paging | more
  ```

  This command does not redirect the standard output to a file but to the standard input of the `more` utility, which pages the output generated by `3-5paging` on the screen.
  After you have checked the above answers, ignore the result of this execution. Thus, the next questions must be answered using the information contained in `output.txt`.
- ❏ In what virtual page is the element of the array located?[3] Answer with an hexadecimal number, `(1)` format.
- ❏ The page directory and the page tables of the task must be accessible only by the operating system, and thus, they must be located in the highest GiB of the virtual address space. Check that the PDE and PTE associated to the global variable match this criterion.
- ❏ Calculate the difference between the virtual and the physical addresses of the PDE. What is it?[4] Answer with an hexadecimal number. The translation of a virtual address of the operating system (located in the highest GiB) to the corresponding physical address is computed by subtracting the constant you just have calculated to the virtual address.[7]
- ❏ Taking the previous answer into account, do you think that the virtual address of the PDE and the PTE of the element of the array will vary from execution to execution of the program? Why?[5] Answer with `(3)` format.
- ❏ What is the page directory index (PDI) of the element of the array?[6] Answer with an hexadecimal number.
- ❏ What physical memory address is the page directory of the task located from?[7] Answer with an hexadecimal number.
- ❏ What is the page table index (PTI) of the element of the array?[8]

1

2

3

4

5

6

7

8

[7] This constant is called `PAGE_OFFSET` and is defined in the source code of the Linux kernel.

Answer with an hexadecimal number.

- ❏ Taking into account the value of the PDE of the element of the array, what physical memory address is the page table associated with this element located from?[9] Answer with an hexadecimal number. [9]
- ❏ Taking into account the physical address in which the PTE of the element of the array is stored, what physical address is the page table associated with this element located from? Do not include the answer to this question in the file, this question is intended for you to check the previous question since both questions should provide the same answer.
- ❏ Taking the PTE of the element of the array into account you must indicate whether its virtual page is read only or read and write, user or supervisor, the cache placement strategy, whether it can be cached, whether it has been accessed, and whether it is in physical memory. To do so, you must use the information provided in the appendix A.[10] Answer with `(3)` format, being critical with the results since the PTE corresponds to a data page of the task. [10]

Now, the PDE and PTE associated to different areas of the virtual address space of the task will be analyzed. The list of PDE and PTE generated will be used.

- ❏ How much RAM memory is used by the task before requesting memory to the operating system?[11] Answer with `(2)` format, expressed in KiB. [11]
- ❏ What is the size of the data structures required for the translation of virtual addresses before requesting memory to the operating system?[12] Answer with `(2)` format, expressed in KiB. Remember that the task requires a directory page and several page tables. [12]
- ❏ How much memory has the task requested?[13] Answer with `(2)` format, expressed in bytes. You must have a look at the source code to answer this question, in particular, to the call to the `malloc` function. [13]
- ❏ Taking into account that the operating system provides memory in page units, how many of the requested pages are stored in disc?[14] Answer with `(2)` format. [14]

Now, the PTE associated with the peripheral device will be analyzed. As you can see in the `output.txt` file all the pages have the same attributes and the only variation is the page frame associated. Thus, only one PTE will be analyzed.

- ❏ Taking the content of a PTE of the interface into account you must indicate whether its virtual page is read only or read and write, user or supervisor, the cache placement strategy, whether it can

be cached, whether it has been accessed, and whether it is in physical memory. To do so, you must use the information provided in the appendix A.[15] Answer with `(3)` format, being critical with the results since the PTE corresponds to a page of the interface of a peripheral device.

[15]

Finally, the PDE and PTE associated to the operating system will be analyzed. They can be found at the end of the `output.txt` file.

- ❏ How many 4 MiB pages of the operating system are in physical memory?[16] Answer with a decimal number.
- ❏ How many 4 KiB pages of the operating system are in physical memory?[17] Answer with a decimal number. You may compute this number from the total amount of pages of the operating system in memory and the previous answer.
- ❏ How much memory is the operating system using?[18] Answer with a decimal number, expressed in MiB.
- ❏ If there would not exist 4 MiB pages, how many 4 KiB pages would be necessary?[19] Answer with a decimal number. If there would not exist 4 MiB pages the TLB hit rate would decrease, and thus, the memory access time would be increased.

[16]

[17]

[18]

[19]

2. Evaluation

At the end of this lab you must generate a zipped file, called `dni.zip`, where `dni` is the sequence of digits of your Spanish ID, containing the following files:

- The `3-5pagetable.h` header file completed.
- The `3-5pagetable.c` source file completed.
- The `3-5paging` executable file.
- The `iomem.txt` file with the memory map.
- The `output.txt` file with the output generated by the program.
- The `answers.txt` file with the answers to the above questions.

3. Appendix A

Table 3-3. Page Sizes and Physical Address Sizes

PG Flag, CR0	PAE Flag, CR4	PSE Flag, CR4	PS Flag, PDE	PSE-36 CPUID Feature Flag	Page Size	Physical Address Size
0	X	X	X	X	—	Paging Disabled
1	0	0	X	X	4 KBytes	32 Bits
1	0	1	0	X	4 KBytes	32 Bits
1	0	1	1	0	4 MBytes	32 Bits
1	0	1	1	1	4 MBytes	36 Bits
1	1	X	0	X	4 KBytes	36 Bits
1	1	X	1	X	2 MBytes	36 Bits

3.7.1. Linear Address Translation (4-KByte Pages)

Figure 3-12 shows the page directory and page-table hierarchy when mapping linear addresses to 4-KByte pages. The entries in the page directory point to page tables, and the entries in a page table point to pages in physical memory. This paging method can be used to address up to 2^{20} pages, which spans a linear address space of 2^{32} bytes (4 GBytes).

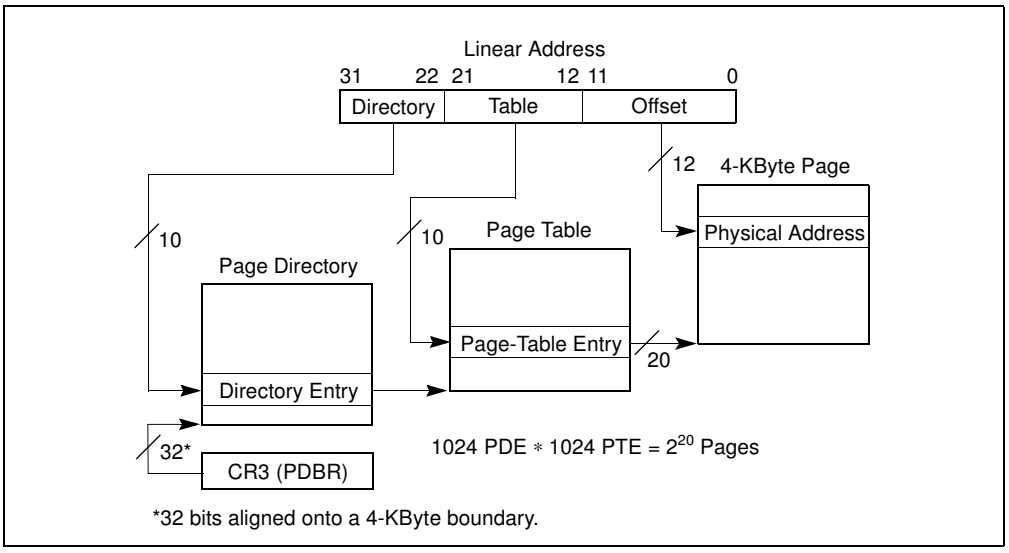


Figure 3-12. Linear Address Translation (4-KByte Pages)

To select the various table entries, the linear address is divided into three sections:

- Page-directory entry—Bits 22 through 31 provide an offset to an entry in the page directory. The selected entry provides the base physical address of a page table.

- Page-table entry—Bits 12 through 21 of the linear address provide an offset to an entry in the selected page table. This entry provides the base physical address of a page in physical memory.
- Page offset—Bits 0 through 11 provides an offset to a physical address in the page.

Memory management software has the option of using one page directory for all programs and tasks, one page directory for each task, or some combination of the two.

3.7.2. Linear Address Translation (4-MByte Pages)

Figure 3-12 shows how a page directory can be used to map linear addresses to 4-MByte pages. The entries in the page directory point to 4-MByte pages in physical memory. This paging method can be used to map up to 1024 pages into a 4-GByte linear address space.

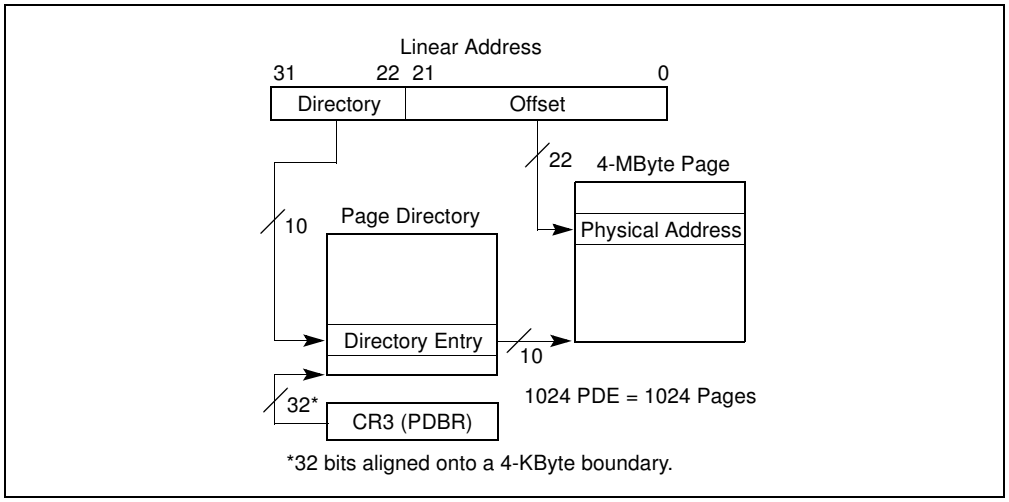


Figure 3-13. Linear Address Translation (4-MByte Pages)

The 4-MByte page size is selected by setting the PSE flag in control register CR4 and setting the page size (PS) flag in a page-directory entry (see Figure 3-14). With these flags set, the linear address is divided into two sections:

- Page directory entry—Bits 22 through 31 provide an offset to an entry in the page directory. The selected entry provides the base physical address of a 4-MByte page.
- Page offset—Bits 0 through 21 provides an offset to a physical address in the page.

NOTE

(For the Pentium processor only.) When enabling or disabling large page sizes, the TLBs must be invalidated (flushed) after the PSE flag in control

register CR4 has been set or cleared. Otherwise, incorrect page translation might occur due to the processor using outdated page translation information stored in the TLBs. See Section 9.9., “Invalidating the Translation Lookaside Buffers (TLBs)”, for information on how to invalidate the TLBs.

3.7.3. Mixing 4-KByte and 4-MByte Pages

When the PSE flag in CR4 is set, both 4-MByte pages and page tables for 4-KByte pages can be accessed from the same page directory. If the PSE flag is clear, only page tables for 4-KByte pages can be accessed (regardless of the setting of the PS flag in a page-directory entry).

A typical example of mixing 4-KByte and 4-MByte pages is to place the operating system or executive’s kernel in a large page to reduce TLB misses and thus improve overall system performance. The processor maintains 4-MByte page entries and 4-KByte page entries in separate TLBs. So, placing often used code such as the kernel in a large page, frees up 4-KByte-page TLB entries for application programs and tasks.

3.7.4. Memory Aliasing

The IA-32 architecture permits memory aliasing by allowing two page-directory entries to point to a common page-table entry. Software that needs to implement memory aliasing in this manner must manage the consistency of the accessed and dirty bits in the page-directory and page-table entries. Allowing the accessed and dirty bits for the two page-directory entries to become inconsistent may lead to a processor deadlock.

3.7.5. Base Address of the Page Directory

The physical address of the current page directory is stored in the CR3 register (also called the page directory base register or PDBR). (See Figure 2-5 and Section 2.5., “Control Registers”, for more information on the PDBR.) If paging is to be used, the PDBR must be loaded as part of the processor initialization process (prior to enabling paging). The PDBR can then be changed either explicitly by loading a new value in CR3 with a MOV instruction or implicitly as part of a task switch. (See Section 6.2.1., “Task-State Segment (TSS)”, for a description of how the contents of the CR3 register is set for a task.)

There is no present flag in the PDBR for the page directory. The page directory may be not-present (paged out of physical memory) while its associated task is suspended, but the operating system must ensure that the page directory indicated by the PDBR image in a task's TSS is present in physical memory before the task is dispatched. The page directory must also remain in memory as long as the task is active.

3.7.6. Page-Directory and Page-Table Entries

Figure 3-14 shows the format for the page-directory and page-table entries when 4-KByte pages and 32-bit physical addresses are being used. Figure 3-15 shows the format for the page-directory entries when 4-MByte pages and 32-bit physical addresses are being used. The functions of the flags and fields in the entries in Figures 3-14 and 3-15 are as follows:

Page base address, bits 12 through 32

(Page-table entries for 4-KByte pages.) Specifies the physical address of the first byte of a 4-KByte page. The bits in this field are interpreted as the 20 most-significant bits of the physical address, which forces pages to be aligned on 4-KByte boundaries.

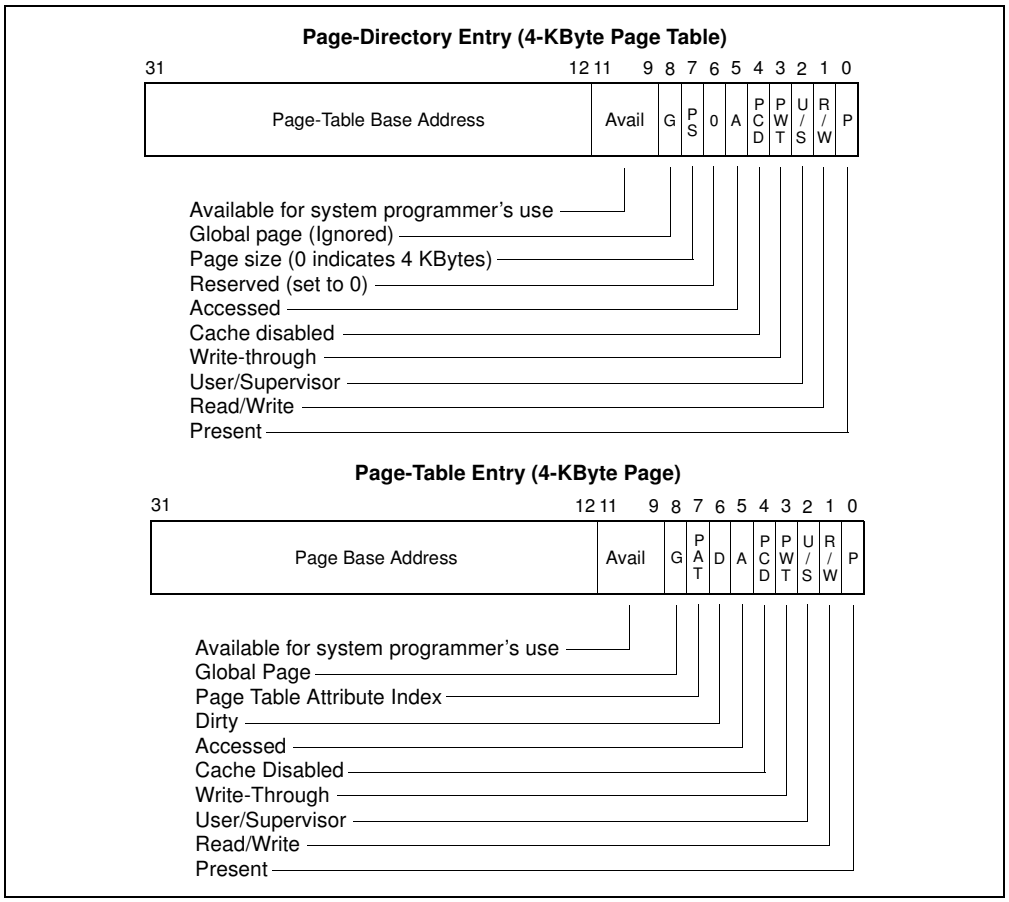


Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses

(Page-directory entries for 4-KByte page tables.) Specifies the physical address of the first byte of a page table. The bits in this field are interpreted as the 20 most-significant bits of the physical address, which forces page tables to be aligned on 4-KByte boundaries.

(Page-directory entries for 4-MByte pages.) Specifies the physical address of the first byte of a 4-MByte page. Only bits 22 through 31 of this field are used (and bits 12 through 21 are reserved and must be set to 0, for IA-32 processors through the Pentium II processor). The base address bits are interpreted as the 10 most-significant bits of the physical address, which forces 4-MByte pages to be aligned on 4-MByte boundaries.

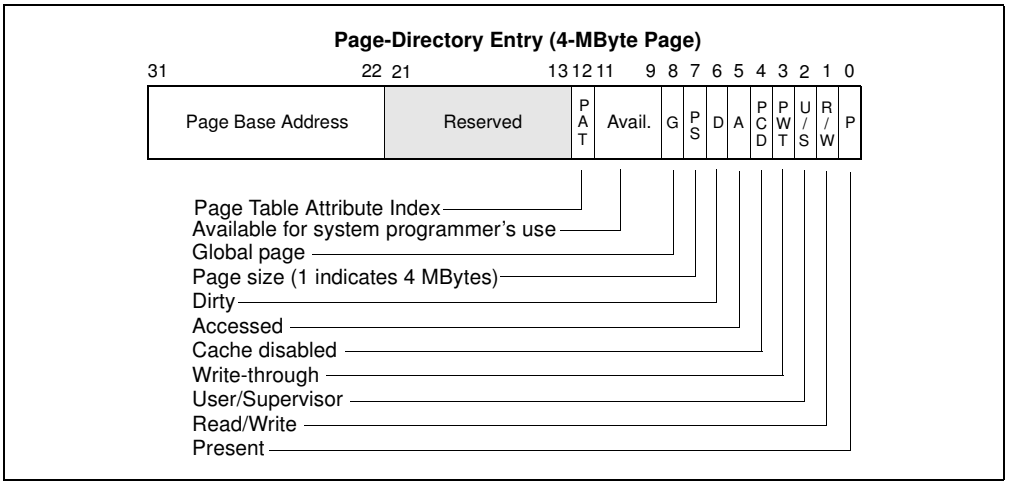


Figure 3-15. Format of Page-Directory Entries for 4-MByte Pages and 32-Bit Addresses

Present (P) flag, bit 0

Indicates whether the page or page table being pointed to by the entry is currently loaded in physical memory. When the flag is set, the page is in physical memory and address translation is carried out. When the flag is clear, the page is not in memory and, if the processor attempts to access the page, it generates a page-fault exception (#PF).

The processor does not set or clear this flag; it is up to the operating system or executive to maintain the state of the flag.

If the processor generates a page-fault exception, the operating system generally needs to carry out the following operations:

1. Copy the page from disk storage into physical memory.
2. Load the page address into the page-table or page-directory entry and set its present flag. Other flags, such as the dirty and accessed flags, may also be set at this time.

3. Invalidate the current page-table entry in the TLB (see Section 3.11., “Translation Lookaside Buffers (TLBs)”, for a discussion of TLBs and how to invalidate them).
4. Return from the page-fault handler to restart the interrupted program (or task).

Read/write (R/W) flag, bit 1

Specifies the read-write privileges for a page or group of pages (in the case of a page-directory entry that points to a page table). When this flag is clear, the page is read only; when the flag is set, the page can be read and written into. This flag interacts with the U/S flag and the WP flag in register CR0. See Section 4.11., “Page-Level Protection”, and Table 4-2 for a detailed discussion of the use of these flags.

User/supervisor (U/S) flag, bit 2

Specifies the user-supervisor privileges for a page or group of pages (in the case of a page-directory entry that points to a page table). When this flag is clear, the page is assigned the supervisor privilege level; when the flag is set, the page is assigned the user privilege level. This flag interacts with the R/W flag and the WP flag in register CR0. See Section 4.11., “Page-Level Protection”, and Table 4-2 for a detail discussion of the use of these flags.

Page-level write-through (PWT) flag, bit 3

Controls the write-through or write-back caching policy of individual pages or page tables. When the PWT flag is set, write-through caching is enabled for the associated page or page table; when the flag is clear, write-back caching is enabled for the associated page or page table. The processor ignores this flag if the CD (cache disable) flag in CR0 is set. See Section 9.5., “Cache Control”, for more information about the use of this flag. See Section 2.5., “Control Registers”, for a description of a companion PWT flag in control register CR3.

Page-level cache disable (PCD) flag, bit 4

Controls the caching of individual pages or page tables. When the PCD flag is set, caching of the associated page or page table is prevented; when the flag is clear, the page or page table can be cached. This flag permits caching to be disabled for pages that contain memory-mapped I/O ports or that do not provide a performance benefit when cached. The processor ignores this flag (assumes it is set) if the CD (cache disable) flag in CR0 is set. See Chapter 9, *Memory Cache Control*, for more information about the use of this flag. See Section 2.5., “Control Registers”, for a description of a companion PCD flag in control register CR3.

Accessed (A) flag, bit 5

Indicates whether a page or page table has been accessed (read from or written to) when set. Memory management software typically clears this flag when a page or page table is initially loaded into physical memory. The processor then sets this flag the first time a page or page table is accessed. This flag is a “sticky” flag, meaning that once set, the processor does not implicitly clear it. Only software can clear this flag. The accessed and dirty flags are provided for

use by memory management software to manage the transfer of pages and page tables into and out of physical memory.

Dirty (D) flag, bit 6

Indicates whether a page has been written to when set. (This flag is not used in page-directory entries that point to page tables.) Memory management software typically clears this flag when a page is initially loaded into physical memory. The processor then sets this flag the first time a page is accessed for a write operation. This flag is "sticky," meaning that once set, the processor does not implicitly clear it. Only software can clear this flag. The dirty and accessed flags are provided for use by memory management software to manage the transfer of pages and page tables into and out of physical memory.

Page size (PS) flag, bit 7 page-directory entries for 4-KByte pages

Determines the page size. When this flag is clear, the page size is 4 KBytes and the page-directory entry points to a page table. When the flag is set, the page size is 4 MBytes for normal 32-bit addressing (and 2 MBytes if extended physical addressing is enabled) and the page-directory entry points to a page. If the page-directory entry points to a page table, all the pages associated with that page table will be 4-KByte pages.

Page attribute table index (PAT) flag, bit 7 in page-table entries for 4-KByte pages and bit 12 in page-directory entries for 4-MByte pages

(Introduced in the Pentium III processor.) Selects PAT entry. For processors that support the page attribute table (PAT), this flag is used along with the PCD and PWT flags to select an entry in the PAT, which in turn selects the memory type for the page (see Section 9.12., "Page Attribute Table (PAT)"). For processors that do not support the PAT, this bit is reserved and should be set to 0.

Global (G) flag, bit 8

(Introduced in the Pentium Pro processor.) Indicates a global page when set. When a page is marked global and the page global enable (PGE) flag in register CR4 is set, the page-table or page-directory entry for the page is not invalidated in the TLB when register CR3 is loaded or a task switch occurs. This flag is provided to prevent frequently used pages (such as pages that contain kernel or other operating system or executive code) from being flushed from the TLB. Only software can set or clear this flag. For page-directory entries that point to page tables, this flag is ignored and the global characteristics of a page are set in the page-table entries. See Section 3.11., "Translation Lookaside Buffers (TLBs)", for more information about the use of this flag. (This bit is reserved in Pentium and earlier IA-32 processors.)

Reserved and available-to-software bits

For all IA-32 processors. Bits 9, 10, and 11 are available for use by software. (When the present bit is clear, bits 1 through 31 are available to software—see Figure 3-16.) In a page-directory entry that points to a page table, bit 6 is reserved and should be set to 0. When the PSE and PAE flags in control register CR4 are set, the processor generates a page fault if reserved bits are not set to 0.

For Pentium II and earlier processors. Bit 7 in a page-table entry is reserved and should be set to 0. For a page-directory entry for a 4-MByte page, bits 12 through 21 are reserved and must be set to 0.

For Pentium III and later processors. For a page-directory entry for a 4-MByte page, bits 13 through 21 are reserved and must be set to 0.

3.7.7. Not Present Page-Directory and Page-Table Entries

When the present flag is clear for a page-table or page-directory entry, the operating system or executive may use the rest of the entry for storage of information such as the location of the page in the disk storage system (see Figure 3-16).

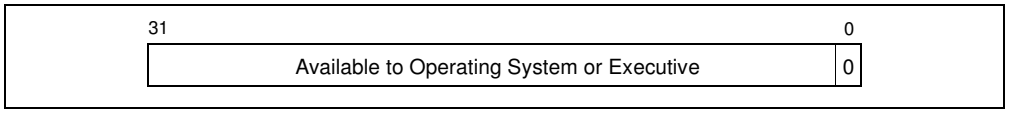


Figure 3-16. Format of a Page-Table or Page-Directory Entry for a Not-Present Page

4. Appendix B: Example of `3-5paging` output

The output of the program consists of thousands of lines. Thus, only a few are shown here. The rest are shown as dotted lines.

```
Virtual addr.  = 08052C04h
Physical addr. = 14E5AC04h
Content        = 12345678h
------------------------------
PTE virtual addr.  = E0316148h
PTE physical addr. = 20316148h
PTE content        = 14E5A067h
------------------------------
PDE virtual addr.  = E02E6080h
PDE physical addr. = 202E6080h
PDE content        = 20316067h
------------------------------

---Virtual pages of the task before requesting memory---
Virtual page  PDE   PTE
------------  ---   ---
002EAh    269DE067h 3701A025h
00617h    2031F067h 5C685025h
00618h    2031F067h 5C6A5025h
    .........................
    .........................
BF8E8h    20356067h 15D51067h
BF8E9h    20356067h 15D58067h
BF8EAh    20356067h 12883067h

---Memory provided from the address 08A69008h---

---Virtual pages of the task after requesting memory---
Virtual page  PDE   PTE
------------  ---   ---
002EAh    269DE067h 3701A025h
00617h    2031F067h 5C685025h
00618h    2031F067h 5C6A5025h
    .........................
    .........................
0804Bh    23F98067h 83C07067h
091F3h    1F08E067h 83924067h
09200h    1F08E067h 95733067h
B77B3h    1F141067h 83923067h
    .........................
    .........................
BF8E8h    20356067h 15D51067h
BF8E9h    20356067h 15D58067h
BF8EAh    20356067h 12883067h

---Virtual pages of the interface---
Virtual page  PDE   PTE
------------  ---   ---
F80B0h    3701D067h FEBFC173h
F80B1h    3701D067h FEBFD173h
F80B2h    3701D067h FEBFE173h
F80B3h    3701D067h FEBFF173h
```

```
---Virtual pages of the OS---
Virtual page  PDE   PTE
------------  ---   ---
C0000h    009A1067h 00000163h
C0001h    009A1067h 00001163h
C0002h    009A1067h 00002163h
    ..............................
    ..............................
C07FDh    36955063h 007FD163h
C07FEh    36955063h 007FE163h
C07FFh    36955063h 007FF163h
302h      008001E3h ---------
303h      00C001E3h ---------
304h      010001E3h ---------
    ..............................
    ..............................
3DAh      368001E3h ---------
3DBh      36C001E3h ---------
3DCh      370001E3h ---------
F7400h    00015067h 37400163h
F7401h    00015067h 37401163h
F7402h    00015067h 37402163h
    ..............................
    ..............................
FFF1Ah    008CD067h E00E217Bh
FFFFAh    008CD067h FEC0017Bh
FFFFBh    008CD067h FEE0017Bh
```

5. Appendix C: Example of the `answers.txt` file

```
1. Yes/No, Why?
2. Yes/No, Why?
3. XXXXXh
4. XXXXXXXXh
5. Yes/No, Why?
6. XXXh
7. XXXXXXXXh
8. XXXh
9. XXXXXXXXh
10. Page: XXXXXh
    Read/Write Why?...
    User/Supervisor Why?
    Cache writing write-through/back Why?
    Can/Cannot be cached Why?
    Has/Hasn't been accessed Why?
    Is/Isn't present in main memory Why?
11. XXX
12. XXX
13. XXXX
14. XX
15. Page: XXXXXh
    Read/Write Why?...
    User/Supervisor Why?
    Cache writing write-through/back Why?
    Can/Cannot be cached Why?
    Has/Hasn't been accessed Why?
    Is/Isn't present in main memory Why?
16. XX
17. XXXX
18. XXX
19. XXXX
```