Lab 05 – Martino Mensio

Exercise 1

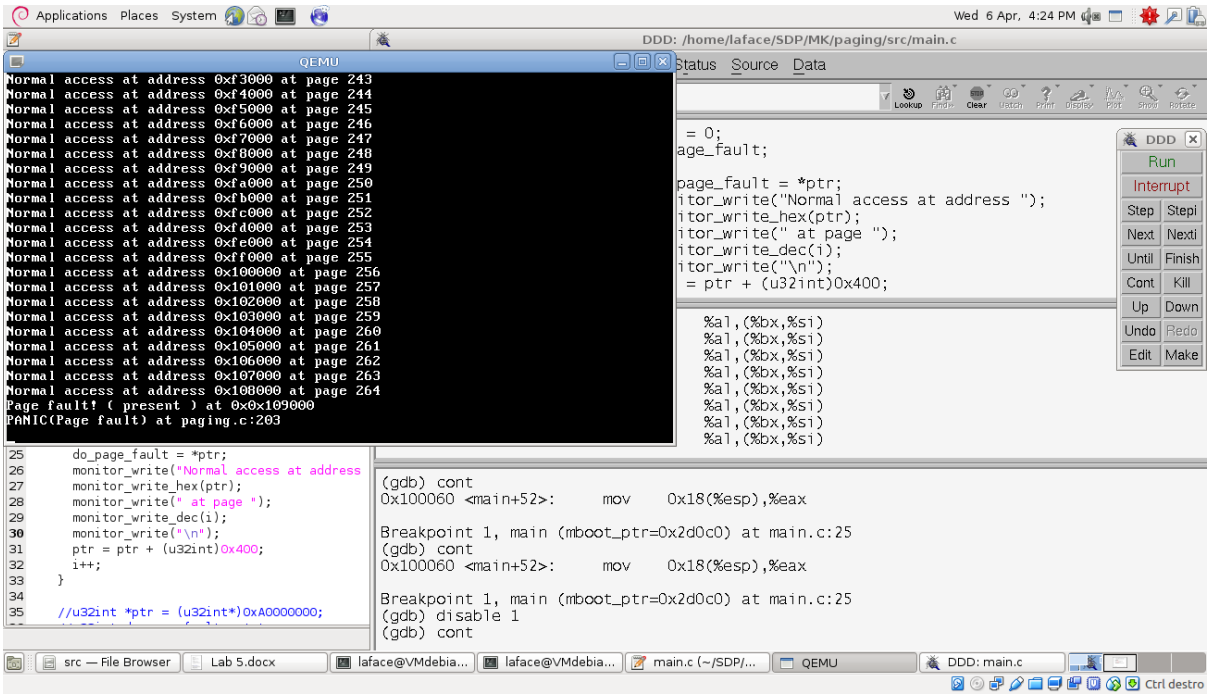
In this exercise we are required to find the number of pages that the kernel allocated. The kernel allocates a number of pages, that correspond to logical addresses with low values.

The strategy applied is to access at increasing logical addresses and see when a page fault occurs. The dimension of a page is 4096 (0x1000) so I need to access address: 0x0, 0x1000, 0x2000, etc.

```
u32int i = 0;
u32int *ptr = 0;
u32int do_page_fault;
while(1) {
	do_page_fault = *ptr;
	monitor_write("Normal access at address ");
	monitor_write_hex(ptr);
	monitor_write(" at page ");
	monitor_write_dec(i);
	monitor_write("\n");
	ptr = ptr + (u32int)0x400;
	i++;
}
```

The pointer is increased following the arithmetic of pointers: the value 0x400 is multiplied by 4, obtaining 0x1000 and is added to `ptr`.

When the page fault occurs, the program is stopped and the `panic` function is called by the kernel.

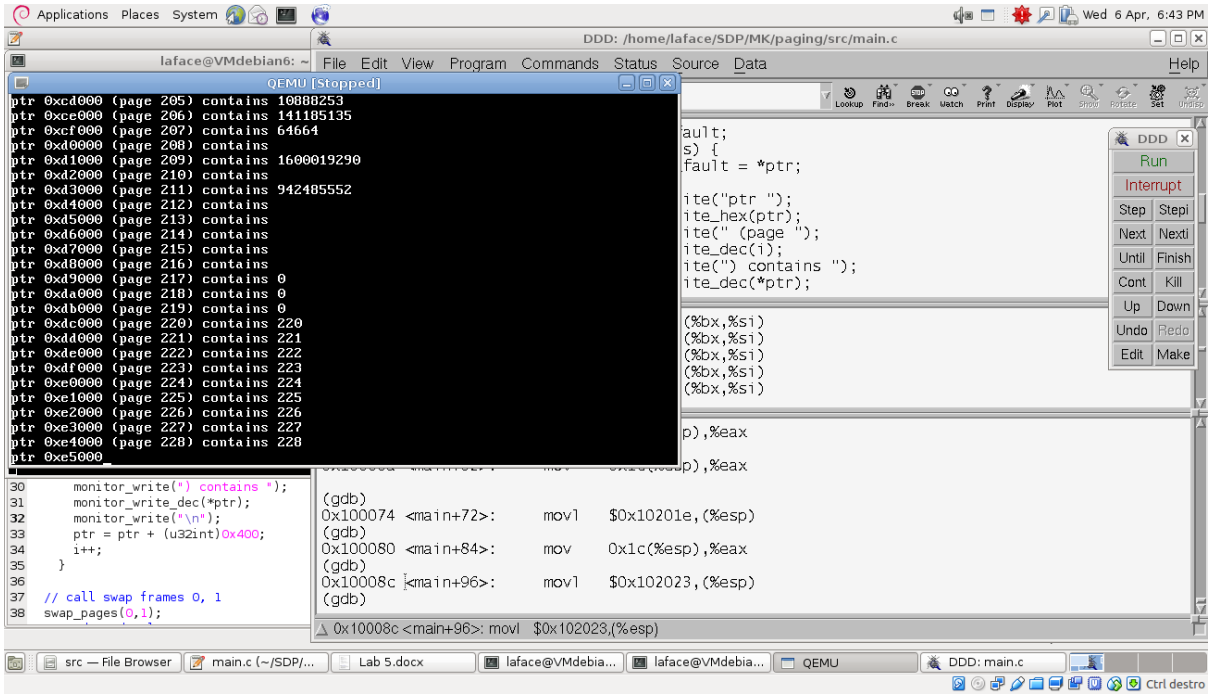


Exercise 2

For the second exercise the first part is simply to write at the beginning of each page the number of the page. In order to do that, a simple for loop is necessary:

```
u32int i = 0;
u32int *ptr = 0;
while(i < num_pages) {
  *ptr = i;
  monitor_write("ptr ");
  monitor_write_hex(ptr);
  monitor_write(" (page ");
  monitor_write_dec(i);
  monitor_write(") contains ");
  monitor_write_dec(*ptr);
  monitor_write("\n");
  ptr = ptr + (u32int)0x400;
  i++;
}
```

After writing the value, a read is performed on the address to see if the read was successful. For this part of the exercise a problem was encountered: trying to write in pages from 160 up, something strange happens: the read returns a different value from the value written, and writing to the last pages allocated by the kernel makes the program flow go somewhere else. Maybe some of those pages contain data must not be modified. In order to avoid this behavior, num_pages is limited to 160 (0 to 159) for this part of the exercise.



For the second part of the exercise, we need to perform a swap of pages in the kernel structure that contains the mapping between pages and frames.

The main calls a function defined in the paging.c file that does this work.

This function performs 3 operations:

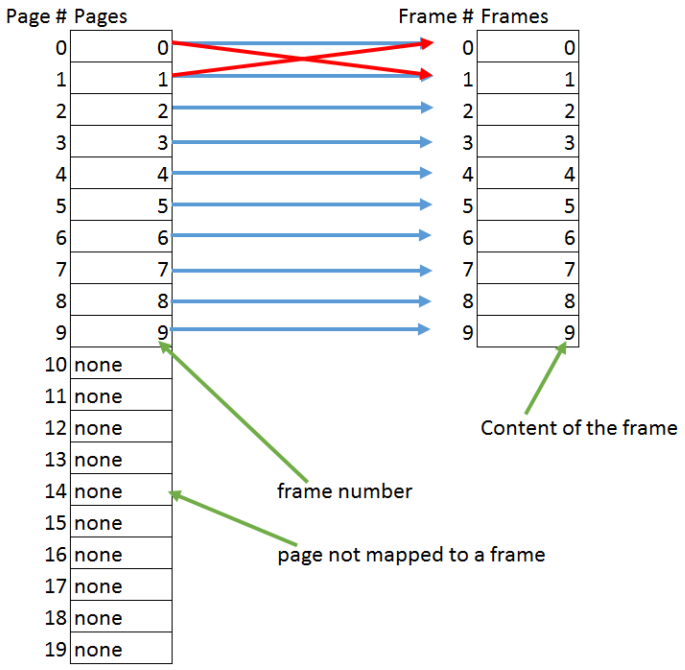
- Disables paging
- Modifies the information contained in the kernel_directory structure
- Enables again paging

The paging is enabled/disabled by setting the 31th bit in cr0 register. This operation is performed by some assembly instructions that read the value of the register and set the value using a mask.

```
void swap_pages(u32int pn1, u32int pn2) {
  u32int fr_n1, fr_n2;
  u32int dn1, dn2, pp1, pp2;
  dn1 = pn1 / 1024;
  dn2 = pn2 / 1024;
  pp1 = pn1 % 1024;
  pp2 = pn2 % 1024;

  page_directory_t *dir = kernel_directory;
  //disable paging
  u32int cr0;
  asm volatile("mov %%cr0, %0": "=r"(cr0));
  cr0 ^= 0x80000000; // Disable paging!
  asm volatile("mov %0, %%cr0":: "r"(cr0));

  // swap pages
  fr_n1 = dir->tables[dn1]->pages[pp1].frame;
  fr_n2 = dir->tables[dn2]->pages[pp2].frame;
  dir->tables[dn1]->pages[pp1].frame = fr_n2;
  dir->tables[dn2]->pages[pp2].frame = fr_n1;

  // now re-enable paging
  asm volatile("mov %%cr0, %0": "=r"(cr0));
  cr0 |= 0x80000000; // Enable paging!
  asm volatile("mov %0, %%cr0":: "r"(cr0));
}
```

The main, in order to be able to call this function, must see the prototype in the file `paging.h`.

After the call to this function with arguments 0 and 1, the main reads from logical address 0x0 the value 1 and from logical address 0x1000 the value 0.



The paging mechanism is a 2-level paging:

- The struct page_directory contains an array of 1024 struct page_table
- The struct page contains an array of 1024 struct page
- Each page contains different fields, between them there is the frame number (the one that has to be changed)

Since the kernel only allocated 265 pages, they all belong to page table 0. But for general purpose the function swap_pages is able to handle also big page numbers, calculating the corresponding page table number and page number.