CSCE 613: Project 3

Two page sizes design

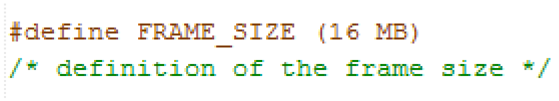
Student: Caio Duarte Diniz Monteiro

Date: 02/22/16

If we want our page management system to work on architectures with different page sizes, we would need to modify two classes: FramePool class; and PageTable class.

FramePool class changes

Changing the FramePool class would be pretty straightforward. Since the frame size value is set using #define, simply updating this value from 4 KB to 16 MB would do most of the work. All further computations on this class are dependent on the FRAME_SIZE defined value. As we are increasing the frame size, there is also no worries about whether the frame size is enough to store the frame bitmaps for the kernel and process memory pools.



```
#define FRAME_SIZE (16 MB)
/* definition of the frame size */
```

Another concern when dealing with 16 MB pages is the addresses of the kernel and process memory pools. Since our frames are now 16 MB, it is not possible for the kernel memory pool to start at 2 MB anymore, needing to start at 16 MB and the size of it should also be increased to 16 MB, being allocated on a single frame. The process memory pool would then start right after the kernel memory pool, and both process and kernel pool sizes should be a multiple of 16 MB.

If an area of the memory needs to be marked as inaccessible, it would also need to be a multiple of 16 MB.

PageTable class changes

The PageTable class contains a static global variable PAGE_SIZE, so it would not be necessary to change the #define as we did for the FramePool class. PageTable also has another global variable called ENTRIES_PER_PAGE, which stores the number of entries for the page directory and page tables. Now that each frame has 16 MB instead of 4 KB we need extra bits for the offset in order to be able to reach the entire address space of the frame. Using 4 KB pages an offset of 12 bits was enough, but for the 16 MB pages it is needed 24 bits to span the entire offset, leaving us with 8 bits to be used on page tables, thus the ENTRIES_PER_PAGE variable would reflect the fewer number of entries on the page table.

```
static const unsigned int PAGE_SIZE        = Machine::PAGE_SIZE; /* in bytes */
static const unsigned int ENTRIES_PER_PAGE = Machine::PT_ENTRIES_PER_PAGE; /* in entries, duh! */
```

Each entry on the page table has a size of 4 bytes, so, with 2^8 entries we would need a total of 1 KB of memory to store all the page table, which fits with no problem in the 16 MB frame size. Due to the fact that now the entire page table fits within a single frame, then it is not necessary anymore to use a two level paging.

On the PageTable constructor, instead of allocating a frame for the page directory and one page table to hold the shared memory with no virtual address translation, just a frame for the page table is necessary and the first entry of this page table will be used as shared memory with no virtual addresses translation. The rest of the page table will then be set to be not present, user level and read/write.

On the load function the only difference is that instead of loading the address of the page directory into the register, the address of the page table will be used. There is no change for the enable_paging function.

Final changes are needed on the page fault handler in order to correctly manipulate the virtual space addresses. As mentioned before, no page directory is used and the leftmost 8 bits of the address is going to be used to determine the position on the page table where the allocated frame address will be stored, and after this is set, the page table entry will be marked as present, user level and read/write.

```
int table_entry = (fault_addr >> 24 & 0x00FF); //gets the entry on the corresponding page table
```