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 that handles different page sizes at the same time, we would need to modify two classes: FramePool class; and PageTable class. One thing to have in mind is that the page size is defined by the operating system upon startup, so there is no mixing of page sizes within a single execution. One workaround to this constraint is adopt the concept of Sections, used in the ARMv7 architecture. When using sections, it is possible to treat a contiguous chunk of frames on the physical memory as a section, which will then have the same behavior as a page.

FramePool class changes

Changing the FramePool class would be pretty straightforward. Since the system could operate with different page sizes, we need to change the way frame size variable is accessed. Right now a #define statement is used, this needs to be changed to something similar to what we have in the PageTable class, where the page size is a global variable set by the machine specification on Machine::PAGE_SIZE. Simply making this change would make most of the work for the frame pool to deal with multiple sizes. All further computations on this class are then going to be dependent on the global variable that we just created. 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.

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. So all these addresses computations would be dependent on the current value of the page size global variable.

If we decide to use the Sections approach, then the frame pool would need some changes on the release frame and get frame functions. These two functions now need to know if they are dealing with regular frames or sections. In the case of sections, when asking for an available frame the function will need to look not just for one available frame, but for a set of contiguous available frames. This block of available frames is then going to be marked as used and no frames within the block can be further allocated before the block is released. For the release function, when dealing with sections, it will simply go to the base address of the section and iterate over all the frames of the section, making them available.

PageTable class changes

As mentioned before, the PageTable class contains a static global variable PAGE_SIZE which indicates what is the page size currently used by the machine. PageTable also has another global variable called ENTRIES_PER_PAGE, which stores the number of entries for the page directory and page tables.

```
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! */
```

If 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.

On a 32-bit architecture with 16 MB pages we will have just 2^8 entries for the page table, each entry has a size of 4 bytes, so, a total memory of 1 KB is enough to store all the page table, this fits with no problem in the 16 MB frame size. Therefore, due to the fact that with 16 MB pages the entire page table fits within a single frame, then it is not necessary to use a two level paging for this page size.

On the PageTable constructor, if using 16 MB pages, instead of allocating a frame for the page directory and other to the page table to hold the shared memory with no virtual address translation, just a single 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. If the page size is 4 KB, then the regular flow of the constructor will execute.

On the load function the only difference is what address is going to be stored on the CR3 register, if page size is 16 MB then the address of the only page table will be used, for the 4 KB page it will store the address of the page directory. 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, the number of offset bits used in 4 KB and 16 MB is different, so the computations of the page table entry, page directory entry (if needed) and page offset are going to differ between these two page sizes. More specifically, for the 4 KB pages we have 12 bits for the offset, 10 bits for the page table entry and 10 bits for the page directory entry. For the 16 MB pages we have 24 bits for the offset and 8 bits for the page table entry, remember that no page directory is needed when using 16 MB pages.

If we use the ARM architecture Sections approach, similarly to the frame pool functions, the page fault handler needs do know the size of the memory chunk that generated the page fault. For this approach the PageTable class is still composed by the page directory and page tables, the difference is that a page directory entry could point to a page table or directly to a section on the physical memory. In the case where the memory chunk that caused the page fault is larger than a threshold, the handler would use the frame pool to get a section of the memory, instead of a single frame, and the starting address of the section would be mapped into the page directory. For all the other cases where the threshold criteria are not met, the usual frame request to the frame pool is made and the address is mapped using the two level paging.