Machine Problem 3: Virtual Memory Management and Memory Allocation

Introduction

In this machine problem we continue our investigation of demand-paging based virtual memory system. For this we extend our solution for MP2 in two directions:

- First, we extend the page table management to support very large numbers and sizes of address spaces. For this, we must place the page table into *virtual memory*. As you will see, this will slightly complicate the page table management and the design of the page fault handlers.
- Second, we will implement a simple memory allocator and hook it up to the `new` and `delete` operators of C++.

As a result we will have a pretty flexible simple memory management system that we will be able to use for later machine problems. In particular, we will be able to dynamically allocate memory in a fashion that is familiar to us from standard user-level C++ programming.

Support for Large Address Spaces

To implement the page table management in the previous machine problem it was sufficient to store the page table directory page and any page table pages in directly-mapped frames. Since the logical address of these frames is identical to their physical address, it was very simple to manipulate the content of the page table directory and of the page table frames. This approach works fine when the number of address spaces and the size of the requested memory is small; otherwise, we very quickly run out of frames in the directly-mapped frame pool.

In the current machine problem, we circumvent the size limitations of the directly-mapped memory by allocating page table pages (and possibly even page table directories if you want) in mapped memory, i.e. memory above 4MB in our case.

Help! My Page Table Pages are in Mapped Memory!

It will be pretty straightforward to move your page-table implementation from direct-mapped memory to mapped memory. When paging is turned on,

the CPU issues logical addresses, and you will have problems working with the page table when you place it in mapped memory. In particular, you will want to modify entries in the page directory and page table pages, but the CPU can only issue logical addresses. You can maintain the mapping from logical addresses of page table pages and page directory to physical addresses by setting up complicated tables to do so. Fortunately, you have the page table that does this already for you. You simply need to find a way to make use of it. Tim Robinson's tutorial "Memory Management 1" (`http://www.osdever.net/tutorials/view/memory-management-1`) briefly addresses this problem. We will use the trick described by Robinson to have the last entry in the page table directory point back to the beginning of the page table.

Recursive Page Table Look-up

Both the page table directory and the page table pages contain physical addresses. If a logical address of the form

```
| X : 10 | Y : 10 | offset :  12 |
```
[1]

is issued by the CPU, the memory management unit (MMU) will use the first 10 bits (value `X`) to index into the page directory (i.e., relative to the Page Directory Base Register) to look up the Page Directory Entry (PDE). The PDE points to the appropriate page table page. The MMU will use the second 10 bits (value `Y`) of the address to index into the page table page pointed to by the PDE to get the Page Table Entry (PTE). This entry will contain a pointer to the physical frame that contains the page.

If we set the last entry in the page directory to point to the page directory itself, we can play a number of interesting tricks. For example, the address

```
| 1023 :  10 | 1023 :  10 | offset :  12 |
```

will be resolved by the MMU as follows: The MMU will use the first 10 bits (value 1023) to index into the page directory to look up the PDE. PDE number 1023 points to the page directory itself. The MMU does not know about this and treats the page directory like any other page table page: It uses the second 10 bits to index into the (supposed) page table page to look up the PTE. Since the second 10 bits of the address are have value 1023, the resulting PTE points again to the page directory itself. Again, the MMU

[1] This expression represents a 32-bit value, with the first 10 bits having value `X`, the following 10 bits having value `Y`, and the last 12 bits having value `offset`.

does not know about this and treats the page directory like any frame : It uses the offset to index into the physical frame. This means that the offset is an index to a byte in the page directory. If the last two bits of the offset are zero, the offset becomes an index to (offset DIV 4)'th entry in the page directory. In this way you can manipulate the page directory if it is in logical memory. Neat!

Similarly, the address

```
| 1023 :  10 | X : 10 | Y : 10 | 0 :  2 |
```

gets processed by the MMU as follows: The MMU will use the first 10 bits (value 1023) to index into the page directory to look up the PDE. PDE number 1023 points to the page directory itself. The MMU does not know about this and treats the page directory like any other page table page: It uses the second 10 bits (value `X`) to index into the (supposed) page table page to look up the PTE (which in reality is the `X`th PDE). The offset is now used to index into the supposed) physical frame, which is in reality the page table page associated with the `X`th directory entry. Therefore, the remaining 12 bits can be used to index into the `Y`th entry in the page table page.

The two examples above illustrate how one can manipulate a page directory that is stored in virtual memory (i.e. not stored in directly-mapped memory in our case) or a page table that is stored in virtual memory, respectively.

An Allocator for Virtual Memory

In the second part of this machine problem we will design and implement an **allocator for virtual memory**. This allocator will be realized in form of the following virtual-memory pool class `VMPool`:

```
class VMPool { /* Virtual Memory Pool */
private:
     /* -- DEFINE YOUR VIRTUAL MEMORY POOL DATA STRUCTURE(s) HERE. */

public:
   VMPool(unsigned long _base_address,
          unsigned long _size,
          FramePool *   _frame_pool
          PageTable *   _page_table);
   /* Initializes the data structures needed for the management of this
      virtual-memory pool.
```

```
      _base_address is the logical start address of the pool.
      _size is the size of the pool in bytes.
      _frame_pool points to the frame pool that provides the virtual
      memory pool with physical memory frames.
      _page_table points to the page table that maps the logical memory
      references to physical addresses. */

   unsigned long allocate(unsigned long _size);
   /* Allocates a region of _size bytes of memory from the virtual
      memory pool. If successful, returns the virtual address of the
      start of the allocated region of memory. If fails, returns 0. */

   void release(unsigned long _start_address);
   /* Releases a region of previously allocated memory. The region
      is identified by its start address, which was returned when the
      region was allocated. */

   BOOLEAN is_legitimate(unsigned long _address);
   /* Returns FALSE if the address is not valid. An address is not valid
      if it is not part of a region that is currently allocated. */
};
```

An address space can have **multiple virtual memory pools** (created by constructing multiple objects of class `VMPool`), with each pool having **multiple regions**, which are created by the function `allocate` and destroyed by the function `release`. Our virtual-memory pool will be a somewhat lazy allocator: Instead of immediately allocating frames for a newly allocated memory region, the pool will simply "remember" that the region exists by storing start address and size in a local table. Only when a reference to a memory location inside the region is made, and a page fault occurs because no frame has been allocated yet, the page table (this is a separate object) finally allocates a frame and makes the page valid. In order for the page table object to know about virtual memory pools, we have the pools **register** with the page table by calling a function `PageTable::register(VMPool * _pool)`. The page table object maintains a list (or likely an array) of references to virtual memory pools. When a virtual memory region is deallocated (as part of a call to `VMPool::release()`, the virtual memory pool informs the page table that any frames allocated to pages within the region can be freed and that the pages are to be invalidated. In order to simplify the implementation, we have the virtual memory pool call the function

`PageTable::free_page(unsigned int page_no)` for each page that is to be freed.

Implementation Issues:

There are no limits to how much you can optimize the implementation of your allocator. Keep the following points in mind when you design your virtual memory pool in order to keep the implementation simple.

- Ignore the fact that the function `allocate` allows for the allocation of arbitrary-sized regions. Instead, always allocate multiples of pages. In this way you won't have to deal with fractions of pages. Except for some internal fragmentation, the user would not know the difference.

- Don't try to optimize the way how frames are returned to the frame pool. Whenever a virtual memory pool releases a region, notify the page table that the pages can be released (and any allocated frames can be freed).

- Keep the implementation of the allocator simple. There is no need to implement a Buddy-System allocator, for example. A simple list of allocated regions, or something similar, should suffice.

- Even maintaining a list of allocated regions can be a bit of a challenge, given that we don't have a memory allocator yet. This is where the frame pool comes in handy. Use the frame pool to store an array of region descriptors. In the constructor of the virtual memory pool request a frame to store the region descriptor list. This solution limits the number of regions that you can allocate. You can easily eliminate this limitation by requesting a new overflow frame whenever the number of regions grows too big, and you run out of space.

- A new virtual memory pool **registers** with the page table object. In this way the page table can check whether memory references are legitimate (i.e., they are part of previously allocated regions). Whenever the page table experiences a page fault, it checks with the registered virtual memory pools whether the address is legitimate by calling the function `VMPool::is_legitimate` on the registered pools. If it is, the page table proceeds to allocate a frame; otherwise, the memory reference is invalid.

- At this time we don't have a backing store yet, and pages cannot be "paged out". This means that we can easily run out of memory if a

program allocates multiple regions and then references lots of pages in the allocated regions. Don' worry about this for now. We will add page-level swapping in a later MP.

Modifications to Class `PageTable`

In this machine problem you will modify the class `PageTable` in the following ways:

1. Add support for page tables in virtual memory. This has been described above.

2. Add support for registration of virtual memory pools. In order to do this, the following function is to be provided:

   ```
   void PageTable::register(VMPool * _pool);
   ```

   The page table object maintains a list of registered pools.

3. Add support for virtual memory pools to request the release of previously allocated pages. The following function is to be provided:

   ```
   void PageTable::free_page(unsigned long _page_no);
   ```

   If the page is valid, the page table releases the frame and marks the page invalid.

4. Add support for region check in page fault handler. Whenever a page fault happens, check with registered pools to see whether the address is legitimate. If it is, proceed with the page fault. Otherwise, abort.

The Assignment

1. Read Tim Robinson's tutorial "Memory Management 1" (`http://www.osdever.net/tutorials/view/memory-management-1`) to understand some of the intricacies of setting up a memory manager.

2. Extend your page table manager from MP2 to handle pages in virtual memory. Use the "recursive page table lookup" scheme described in this handout.

3. Extend your page table manager to (1) handle registration of virtual memory pools, (2) handle requests to free pages, and (3) check for legitimacy of logical addresses during page fault.

4. Implement a simple virtual memory pool manager as defined in file `vm_pool.H`. Always allocate multiples of pages at a time. This will simplify your implementation.

You should have access to a set of source files, BOCHS environment files, and a makefile that should make your implementation easier. In particular, the `kernel.C` file will contain documentation that describes where to add code and how to proceed about testing the code as you progress through the machine problem. The updated interface for the page table is available in `page_table.H` and the interface for the virtual memory pool manager is available in file `vm_pool.H`.

What to Hand In

You are to hand in the following items on CSNET:

- A ZIP file containing the following files:

    1. A design document (in PDF format) that describes your implementation of the page table and the virtual memory pool.
    2. A pair of files, called `page_table.H` and `page_table.C`, which contain the definition and implementation of the required functions to initialize and enable paging, to construct the page table, to handle registration of virtual memory pools, and to handle page faults. Any modification to the provided `.H` file must be well motivated and documented.
    3. A pair of files, called `frame_pool.H` and `frame_pool.C`, which contain the definition and implementation of the frame pool. (These two files are for the benefit of the grader only. If you have not modified these files, simply submit the ones that were handed out to you. Some students have modified them in the previous machine problem, which has made it difficult for the TA to grade the submissions.)
    4. A pair of files, called `vm_pool.H` and `vm_pool.C`, which contain the definition and implementation of the virtual memory pool. Any modifications to the provided file `vm_pool.H` must be well motivated and documented.

- **Note:** Pay attention to the capitalization in file names. For example, if we request a file called `file.H`, we want the file name to end with a capital `H`, not a lower-case one. While Windows does not care about capitalization in file names, other operating systems do. This then causes all kinds of problems when the TA grades the submission.
- Grading of these MPs is a very tedious chore. These handin instructions are meant to mitigate the difficulty of grading, and to ensure that the grader does not overlook any of your efforts.
- **Failure to follow the handin instructions will result in lost points.**