# Machine Problem 4: Virtual Memory Management and Memory Allocation

## Introduction

In this machine problem we complete our memory manager. For this we extend our solution from previous MPs in three directions:

**Part I:** We will extend the page table management to support very large numbers and sizes of address spaces. This is currently not possible because both the page directory and the page table pages are stored in directly-mapped memory, which is very small. We need to move the page tables, at least the page table pages, out from the directly-mapped kernel memory (kernel memory pool) into ”virtual” memory (process memory pool). As you will see, this will slightly complicate the page table management and the design of the page fault handlers.

**Part II:** We will prepare the page table to support virtual memory as described in Part III.

**Part III:** We will implement a simple virtual-memory allocator (similar to the frame pool allocator in a previous MP) 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 can use for later machine problems. Most importantly, we will be able to dynamically allocate memory in a fashion that is familiar to us from standard user-level C++ programming.

## Part I: 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 the directly-mapped frames in the kernel frame pool. Since the logical address of these frames is identical to their physical address, it was very easy 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 this 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. These frames are handled by the process frame pool.

### Help! My Page Table Pages are in Mapped Memory!

Given a working implementation of a page table in direct-mapped memory, it is pretty straightforward to move it from directly-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. You know where these are in physical memory, but the CPU can only issue logical addresses. You can maintain a complicated table that maintains which logical addresses point to which page directory or page table page. Fortunately, this is not necessary, as you already have the page table, which that does exactly this mapping 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: Have the last entry in the page table directory point back to the beginning of the page table. This is also called "Recursive Page Table Look-up" and is described in the section blow. Make sure that you understand the trick. If you don't, ask. Once you understand this trick, the rest of the machine problem will take very little time.

### 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 below will be resolved by the MMU as follows:

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

- The MMU will use the first 10 bits (value 1023) to index into the page directory to look up the PDE. PDE number 1023 (the last one) points to the page directory itself. *The MMU does not know about this* and treats the page directory like any other page table page.
- The MMU then 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 also have value 1023, the resulting PTE points again to the page directory itself. *Again, the MMU 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 the (offset DIV 4)'th entry in the page directory. In this way you can manipulate the page directory if you store it 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. Just as in the example above the MMU does not know about this and treats the page directory like any other page table page.
- The MMU then 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.

[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`.

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.

## Part II: Preparing class `PageTable` to handle Virtual Memory Pools

You will modify the class `PageTable` to support Virtual Memory allocation pools. We describe the class `VMPool` in detail in Part III below.

### Modifications to Class `PageTable`

In order to support virtual memory pools, make the following modifications to the page table. (A new version of the file `page_table.H` is part of the source package. Feel free to use your own file, just add the two new functions `register_pool` and `free_page`.)

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

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

   The page table object shall maintain a list of registered pools.

2. 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. This can be done by calling the function `VMPool::is_legitimate` for each registered pool. If the address is legitimate, proceed with the page fault. Otherwise, abort.

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. **Do not forget to appropriately flush the TLB whenever you mark a page invalid! (see below)**

### Page Table Entries and the TLB

In x86 architecture the TLB is **not coherent** with memory accesses. In simple terms this means that the TLB is not aware of changes that you make to the page table. Therefore, you must flush all relevant entries in the TLB (or flush the entire TLB) each time you make a change to the page table. If you don't do that, the CPU may use a stale entry in the TLB, and your program will likely crash in very mysterious ways. The easiest way to flush the (entire) TLB is to reload the CR3 register (the page table base register) with its current value. The CPU thinks that a new page table is loaded, and it therefore flushes the TLB. (Note that stale TLB entries were not a problem in the previous MPs: If an invalid page is marked as valid as part of the page fault, the TLB gets updated. If we mark valid pages as invalid when we release pages, however, the TLB may not be updated.)

## Part III: An Allocator for Virtual Memory

In the third 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,
          ContFramePool *  _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. */

   bool 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`). Each pool can have **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_pool(VMPool * _pool)`. This allows the page table object to maintain a collection (a list or an array) of references to virtual memory pools. This comes in handy when a page fault occurs, and the page table needs to check whether the memory reference is legitimate. 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. For this, the virtual memory pool calls 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. **We want you to keep the allocator simple!** 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 will 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 (see next point). (Unfortunately, we cannot use the bitmap implementation recommended for the contiguous-frame pool. The bitmap needed to manage a 4GB address space would be huge.)
- Where to store your list of allocated regions? Feel free to use the first page of the pool to store an array of region descriptors (base page number and length of a region). This solution limits the number of regions that you can allocate to less than 512. This should be sufficient for now.[2]
- A new virtual memory pool **registers** with the page table object. In this way, whenever the page table experiences a page fault, it can check whether memory references are legitimate (i.e., they are part of previously allocated regions). The page table checks with the registered virtual memory pools whether the address is legitimate by calling the function `VMPool::is_legitimate` for each registered pool. If the address is not declared legitimate by any pool, the memory reference is invalid, and the kernel aborts.
- 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 references lots of pages in the allocated regions. Don' worry about this for now. We may add page-level swapping in a later MP.

## 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. (Part I) Extend your page table manager from the previous MP to handle pages in virtual memory . Use the "recursive page table lookup" scheme described in this handout. Remember to get your directory frame and the page table page frames from the process frame pool instead of the kernel frame pool!

[2] If you want more, have the last entry in the page be a reference to an overflow page with more region descriptors.

3. Test your implementation of the new page table manager in virtual memory. Towards the end of file kernel.C you can define or un-define a macro called TEST_PAGE_TABLE, which controls whether just the page table is tested, or whether just the VM pools are tested. By default, this macro is defined, and only the page-table implementation is tested. Once you convince yourself that your page table implementation works correctly, uncomment the definition of this macro, and the code will start testing the VM pools.

4. (Part II) 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 faults.

5. (Part III) 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 a ZIP file, with name mp4.zip, containing the following files:

1. A design document, called design.pdf (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 cont_frame_pool.H and cont_frame_pool.C, which contain the definition and implementation of the frame pool. (These two files are for the benefit of the grader only. Some students have modified the definitions 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.**