Project 4: Completing the virtual memory management system

Due Date: Parts I and II: 4/5 11:59 pm CT, Part III: 4/9 11:59

Upcoming dates of interest

Item	Deadline	Comment
Release of P4 handout	3/23/21	
Release of P4 Part II provided code	3/26/21	
Release of P4 Part III provided code	3/28/21	
P4 Part I design - 10% of P4 grade P4 Part I implementation - 15% of P4 grade	4/5/21	5 bonus points if submitted by 3/30/21; Students will be able to proceed to Part II/III without implementing Part I; The Part 1 design document is mandatory (i.e., without it, overall P4 grade is zero);
P4 Part II - 45% of P4 grade	4/5/21	20 bonus points if you build upon your own P3 implementation; 5 bonus points if submitted by 4/2
P4 Part III - 30% of P4 grade	4/9/21	only needs to work for simple tests
P5	4/21/21	
P6	5/3/21	10 bonus points if submitted by 4/30/21
Final	5/5/21 8 - 10:30 am	

Contents

1 Introduction

In this assignment, we complete our memory manager. For this, we extend our solution from previous projects in three ways:

- **Part I:** We 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 inner page tables are stored in directly-mapped memory (pages from the kernel frame allocator are all within the first 4 MB), which is very small. We need to move the page tables, at least the inner page table pages, from the kernel memory pool into "virtual" memory (process memory pool, which has the memory not directly-mapped). As you will see, this will slightly complicate the page table management and the design of the page fault handler.
  Some of us may be worried about getting any code added to our P3, because our implementation is not robust or functional enough. **Do not worry**. You are only required to *understand* how moving the page tables to virtual memory is implemented. It is a neat idea. I believe you will find it worth it of your time. You do not have to implement the idea into your P3 code if you do not want to (you only lose 15 points of part 1). If you implement it but has problems with your code on Part II, you can use a provided version of P3 with the support for large address spaces.

- **Part II:** We prepare the page table to support virtual memory as described in Part III. You need to implement this part. If you do not want to use your P3 code, you can use the provided code, which will include object files for the P3-related functionality.

- **Part III:** We now work a simple virtual-memory allocator. The original plan for the semester (before the winter storm) was to have you fully implementing this and then hook your implementation up to the `new` and `delete` operators of C++. Instead, you will "draft" your implementation: you will code your initial implementation, covering the functionality needed, but you are not required to get the tests to work in order to get full credit.
  Thinking of it as the type of coding you may do for a job interview when someone asks you to code something in a couple of hours: you do your best for a period of time. And then you declare it done. Your initial implementation is enough for you to be able to see all coming together in having support for dynamic memory allocation in C++.

Through P2 + P3 + P4, we achieve a pretty flexible simple memory management system in our kernel. You will appreciate that you went all the way from bare physical memory to do what is needed to support dynamic memory allocation in a fashion that is familiar to us from standard user-level C++ programming. Yes, in P5 and P6 you can use `new` and `delete`! I know you missed having them.

2 Part I: Support for Large Address Spaces

To implement the page table management in the previous project, it was sufficient to store the page table *directory page* and any *page table pages* in the directly-mapped frames from the kernel frame pool. Since the logical address of these frames is identical to their physical address (i.e., that memory area is directly mapped), 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.

We will circumvent the size limitations of the directly-mapped memory by allocating page table pages (and possibly even page table directories if we 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 does exactly this mapping for you. You simply need to find a way to make use of it.

Dr. Riccardo Bettati has a video explaining the idea of **recursive table lookup on the x86**. The video and the corresponding slides are available in the Google Drive for Project 4 (folder `RecursiveTrick-Bettati`). We will use the technique described in the module: have the last entry in the page table directory point back to the beginning of the page table. Make sure that you understand the recursive lookup technique. Once you understand this trick, implementing it would not take long. But if you do not want to implement it, you can let go of this part of the assignment and proceed to the other parts. More details in Section 6.

Recursive Page Table Look-up

Here we summarize the recursive table-lookup mechanism described in the video mentioned above. 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

[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`.

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.

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

You will modify the class `PageTable` to support Virtual Memory allocation pools. We explain virtual memory pools and describe the class `VMPool` in detail in Section 4.

You have two options:

- Use your P3 implementation.
  You get 10 points bonus for building upon your own code.
  The starter code is `P4-provided`
- Use provided `page_table_provided.o` and `cont_frame_provided.o`
  The starter code is `P4-provided-with-P3-objects`

The provided code for P3-PartII will be available in the Google Drive.

Modifications to Class `PageTable`

A new version of the file `page_table.H` is part of the provided source code. If you are using your P3 code for the class PageTable, just add the two new functions `register_pool` and `free_page` to your `page_table.H` file.

In order to support virtual memory pools, make the following modifications to the page table:

1. Add support for registration of virtual memory pools. In order to do this, you need to implement the following function:

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

   The page table object shall maintain a list of registered pools.
   If you are using your P3 code, add this functions to your `page_table.C`
   If you are not using your P3 code, add this function to the file `page_table_p4.C`

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.
   So that you can test your Part II independently from Part III, the provided code for Part II gives you a fake implementation of `VMPool::is_legitimate` that you can use.

   You must implement this checking in the function

   ```
   bool PageTable::check_address(unsigned long address);
   ```

   If you are using your own P3 code, you need to change your method `PageTable::handle_fault` to invoke `check_address`.
   If you are using the provided P3 object files, the provided code is already invoking the `check_address` function that you are implementing.

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)**

   As before, if you are using your P3, add this method to your `page_table.C`, otherwise add it to `page_table_p4.C`.

Notice that the provided code came with debugging output to file enabled (i.e., bochsrc.bxrc, utils.C, and utils.H have the changes described in the document debug-out-to-file distributed earlier in the semester). You can use the functions below to write debugging information to a file.

```
void debug_out_E9(char *_string);
void debug_out_E9_msg_value(char *msg, unsigned int value);
```

You need to specify the file where the output should go when you invoke bochs:

```
bochs -f bochsrc.bxrc > debug_stdout_file
```

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 P3: 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.)

4 Part III: An Allocator for Virtual Memory

You will find the provided code for P3-PartIII in the Google Drive.

In the third part of the project, you will consider the design and implementation of an **allocator for virtual memory** and implement an initial version. You do not have to refine/debug your implementation to get full credit.

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
```

```
26      * is identified by its start address, which was returned when the
27      * region was allocated. */
28
29     bool is_legitimate(unsigned long _address);
30     /* Returns false if the address is not valid. An address is not valid
31      * if it is not part of a region that is currently allocated. */
32 };
```

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 fault handler 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 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 allocator (as used in Linux), 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. But maybe you will have less problems implementing a simple list than a bitmap-based structure, anyway.

- 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't worry about this for now.

5 The Assignment

(See Section 6 for what to turn in)

1. **(Part I design)** Explain how your page table manager from the previous project needs to be modified to handle pages in virtual memory . Use the "recursive page table lookup" scheme described in this handout and explained in Dr. Bettati's video. Describe the specific changes you would do to your PageTable.C. You will turn in a pdf with your changes; if your description refers to your `page_table.C` file, you need to include the relevant parts in the pdf.
   Remember to get your directory frame and the page table page frames from the process frame pool instead of the kernel frame pool!

2. **(Part I implementation)** Implement the changes that you described in the design document in your P3 `page_table.C`.

3. **(Part II)** Implement the expected new functionality in the `PageTable` class in file `page_table_vm.C`, using the P4-PartII provided code, which comes with a test in kernel.C.

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

4. **(Part III)** Design a simple virtual memory pool manager as defined in file `vm_pool.H` and described in Section 3. Your design should be expressed by defining the fields you need to add to `vm_pool.H` and by doing the first draft of your implementation in `vm_pool.C`. This means that you write the code with all the pieces you see needed, you get it to compile and run a very simple test, but you are not required to have it fully functional.

You will 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 parts. 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`.

6 What to Hand In

You will submit your zip files through Canvas. There is one assignment for each part.

1. **Part I design-only**: A document where you describe the changes in your P3 code that you would need to do in order to implement P3. Be specific: identify the exact lines of code you would need to change and how.
   If you did not submit a P3, then you need to write about your understanding of what needs to change in a P3 implementation in order to store page table objects in virtual memory. Be specific. Just repeating what we have in this handout (e.g., "we need do use the recursive page trick") is not sufficient. Instead, you need to explain the technique on your own words, identifying how directory/inner page table entries would be manipulated.

2. **Part I design + implementation**: Your design document does not need to be detailed, because you are submitting your implementation too. Upload a zip file that contains the design document and your new `page_table.C` file

3. **Part II**: You are to hand in a ZIP file, with name `p4-partII.zip`, containing a file called `page_table_vm.C` with your implementation of the functions `PageTable::register_pool` and `PageTable::free_page`.

4. **Part III**: You are to hand in a ZIP file, with name `p4-partIII.zip`. It must contain:

   - a design document, called `design.pdf` (in PDF format) that describes your implementation of the virtual memory pool.
   - A pair of files, called `vm_pool.H` and `vm_pool.C`, which contain the definition and your initial implementation of the virtual memory pool. Again: by "initial implementation" we mean that you wrote the code, but it only passes a few very

simple tests. It may fail the remaining tests, but you still get full credit.
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 projects is a very tedious chore. These hand in 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 hand-in instructions will result in lost points.

7 Document Release Log

- Version 2021-A.1: released on 3/23/21