CIS 548 Project 2: *penn-mmu*

MILESTONE: March 5 @ 10pm
DUE : March 15 @10 pm

Directions

You must work in groups of 2 for this project. If you are caught using code from other groups or any sources from public code repositories, your entire group will receive **ZERO** for this assignment, and will be sent to the Office of Student Conduct where there will be additional sanctions imposed by the university.

Overview

In this assignment you will implement an in-memory simulation of the Memory Management Unit and Page Replacement, *penn-mmu*. The primary learning goal of this assignment is to gain a thorough understanding of Memory Allocation, Page Tables, Page Replacement Algorithms (also referred to as Replacement Policies in this document) and the Translation Lookaside Buffer.
While we recommend that your simulation stays as true to real working systems as possible, we will make simplifying assumptions that gloss over certain hardware limitations that may exist in real systems.

IMPORTANT: Under no circumstances is your team allowed to use any personal git repository for this project. Please make sure you use the repository that has been provided to you by the teaching staff. In adherence to Penn's Academic Integrity Policies, you are not allowed to upload any of your work (now or in the future) to any git repository open to the public. Violating this will warrant serious retroactive action!

1 Specification

This project can be broadly classified into two main parts - Memory Allocation and Virtual Memory Management. The Memory Allocation part of the project will have you implement your own version of malloc and free. The Virtual Memory Management part of the project will involve implementing virtual memory address translation (including the use of a Translation Lookaside Buffer) and page replacement algorithms. In this assignment, you will use the given framework to build a simulation of these components. **Your simulation will run as a single user-level process**.

1.1 Memory Allocation

In order to read and write data, *penn-mmu* must allocate memory. In this project, you are required to implement two functions - `pemmu_malloc` and `pemmu_free` (see **interface.h** in the provided framework

directory, for more on this). Though these functions are intended to mimic Linux's implementation of malloc and free, it is sufficient if they support the following basic functionalities :

1. Memory Allocation should follow the First-Fit algorithm.
2. Unlike Unix, where the memory allocated may be equal to or greater than the size requested (due to alignment boundaries and minimum block sizes), your implementation should allocate the exact size requested. You should not align your addresses to any specific boundary since the granularity of memory access for *penn-mmu* is 1 byte.
3. Your implementation should track memory allocated such that it should not allow data to be read from or written to addresses that are not allocated.
4. Any address that lies within the range of allocated memory is a valid address for read/write operations.
5. Only addresses that are returned by `pemmu_malloc` can be freed. Calling `pemmu_free` on any other address or on an address that has already been freed should result in an error.

1.1.1 First Fit Memory Allocation Algorithm

In order to ensure the virtual addresses allocated by your implementation is in sync with the testing framework you are required to follow the First Fit Memory Allocation Algorithm. The upper limit on the size of a malloc is the page size. Your allocator design should make use of a particular page only if the requested size fit's that page. Hence a single malloc request should not span across page boundaries. Your implementation of the First Fit algorithm should be efficient enough to make use of every **free contiguous address block** in the virtual memory address range before your implementation of malloc can return an error due to insufficient memory. Be assured that the testing framework will stretch your memory allocation algorithm to its absolute limit.

1.2 Virtual Memory Address Translation

It is the job of the Memory Management Unit (MMU) to translate virtual addresses to physical addresses. Recall that a virtual address is really a pair *(p,o)* where the lower-order bits of the address give the offset *o* within a given page and the higher-order bits specify the page number *p*. So the job of the MMU is really to translate virtual page number *p* to physical frame number *f*. The mapping between page numbers and frame numbers is maintained by the Operating System in page tables. There is no need to map the offset bits since that remains the same across virtual and physical addresses.
penn-mmu makes certain **simplifying assumptions** outlined here and illustrated by Figure 1

1. There is no support for multiple processes to run at the same time. We will work with a single process. Thus, *penn-mmu* need only maintain a single page table
2. Given the small size of memory we will be working with, the page table maintained by *penn-mmu* is a single-level page table.
3. Memory is divided into fixed-size pages, the size of which is configurable. Since *penn-mmu* is byte addressable, the granularity of page addresses is 1 byte (`uint8_t`). Hence after an address range is allocated, any address within that range is a valid address for write/read calls.

4. The size of virtual and physical memories are specified via the `set_memory_configuration` function (see **interface.h** in the provided framework, for more on this).

5. Swap memory, as referred to in all parts of this spec, will be a single file that resides in the host OS. (The terms swap space, swap memory and disk are used interchangeably throughout this spec and all represent the same underlying idea of a place on "disk" where pages can live when they are swapped out of physical memory).

6. The swap file should never exceed the virtual memory size. Pages are never evicted from swap memory.

Figure 1 shows a snapshot of a sample configuration of *penn-mmu* with a virtual memory size of 60 bytes, physical memory size of 30 bytes and page size of 6 bytes. In this sample snapshot, the entire virtual memory is filled with an up-counter data of type `uint8_t` (1 byte). Since the size of each page is 6 bytes, it can hold 6 entries of `uint8_t`. It is extremely important you note that the virtual memory is just a logical construct. It should never actually occupy any physical space in your program. The data movement for the read and write calls should be via the physical memory. Study this figure carefully as it gives you a good visualization of *penn-mmu*. All our test configurations will be reasonably small because it is a simulation that will run on limited resources. However, make no assumptions on the max size of the memory configurations. Your implementation should be able to scale to any configuration. Always make use of dynamic memory allocation to scale your data structures.

As part of your design process, you will have to decide on your representation of each of these components. You are free to use any reasonable data structure of your choosing for this purpose. Remember that this assignment, like all others in this class, must be implemented **only in C**. You may not use any standard external libraries for data structures, but must implement your own wherever needed. Since this is a software simulation, your implementation need not be constrained by resources the way that an actual MMU would.

1.2.1 Swap File

In order to improve the parallelism of the grading script, the **creation** of swap file is handled by the testing framework. The test framework uses the **pid** of the process to generate a unique swap file. This way all of the tests can be run in parallel and the swap file operations of one test will not stomp on the operations of another test. As such it is very important your implementation uses the file pointer of the swap file generated by the framework using the provided `get_swap_file_handle()` API in `test_framework.c` and not generate your own swap file!

1.2.2 Page Table Entries

The page table must **at minimum** contain the following data

1. Frame number - the physical frame that a virtual page translates to. You may not use a special placeholder value to indicate that the page table entry is not present in physical memory. That is the purpose of the valid bit below.

2. Valid (present/absent) bit - indicates whether a given page table entry is valid or not.

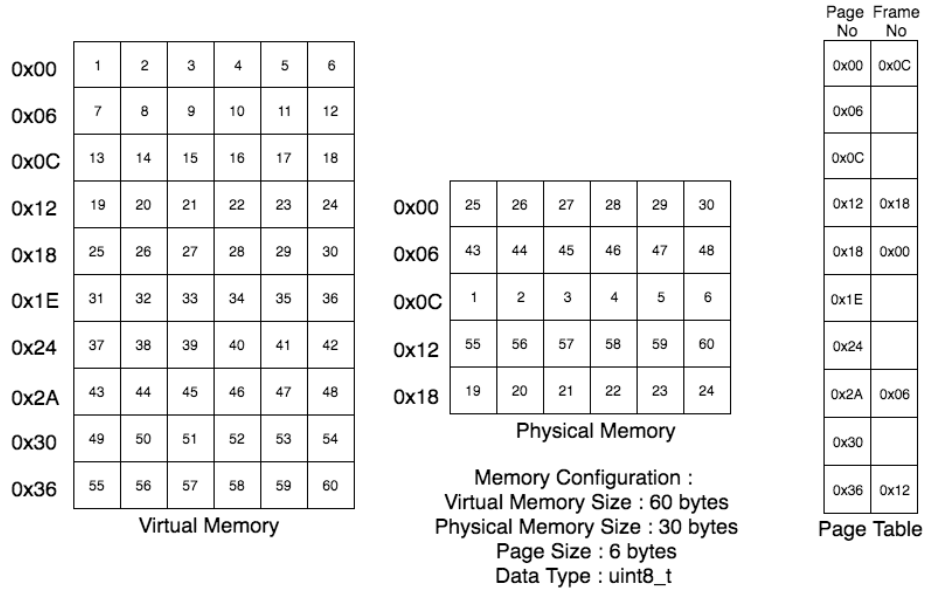

Figure 1: A simple *penn-mmu* memory configuration where each page holds 6 bytes of data

3. Dirty bit - indicates if a write has been performed to the page since it was last written to swap memory. If the dirty bit is not set, a page's information does not have to be written back to disk on eviction. If it is set, the page **must be** written back to ensure no loss of data.

You will need to store additional information in the page table, which will include metadata required for different page replacement algorithms. The above is just the minimum requirement that we will look for while grading.

1.3 Page Replacement

The other important component of Virtual Memory Management is page replacement. Since virtual memory is always larger than physical memory, there is a need to swap pages out of physical memory and onto disk (the swap file), and bring in pages from disk as required. *penn-mmu* will implement the following page replacement algorithms:

1. First In First Out (FIFO)
2. Clock (CLOCK)
3. Least Recently Used (LRU)

The replacement algorithm to be used during the run of a particular test case will be specified by the framework. You can obtain the algorithm that is being used by calling the `get_replacement_policy` function declared in `test_framework.h`.

Each of the replacement algorithms is to be implemented in adherence with the class slides, and the explanation given in class. Please use Piazza or Office Hours to clarify any ambiguity that you face in this regard.

1.4 Translation Lookaside Buffer (TLB)

Recall that the TLB behaves as a cache for virtual memory address translation in that it stores the mapping from virtual page number to physical frame number, much like a single-level page table does. The true gains of a TLB are seen when the system uses multi-level page tables. Since this is not the case in *penn-mmu*, you might wonder what the benefit of the TLB is here. This implementation will hopefully serve as a learning tool, to help understand all of the different bookkeeping and synchronization that the OS must do in order to maintain consistency.

In *penn-mmu*, you will implement the TLB as a configurable part of the system. The boolean `tlb_enabled` in *interface.h* (which is a part of the configurations set by the testing framework) must be used to turn on and off the functionality of the TLB as required. Your implementation will be subject to a series of tests both with and without the TLB.

The size of the TLB is also configurable, and will set by the testing framework as the number of TLB entries. Please go through the section on testing (Section 5) for more details on configurations and the testing framework.

In hardware, the TLB is implemented as associative memory (or content addressable memory), so that all entries can be accessed in parallel. You are not required to simulate this and can scan through entries iteratively since we are not concerned about access latency in a software simulation. Having said that, it is useful to remember that the TLB is not an indexed data structure like the page table!

Your TLB must contain the following entries

1. virtual page number
2. physical frame number
3. dirty bit

Your TLB must implement an **LRU based eviction** of entries.

1.5 End-to-End Behavior

penn-mmu supports only custom read and write calls which are a part of the framework that has been provided to you. At a high level, these are the operations that the testing framework will make use of:

1. `pemmu_malloc` - in order to perform any write or read operation, memory must be allocated for the variable via the `pemmu_malloc` function call. This function is used to invoke your address allocation algorithm and should return a virtual address that can be used by the read and write routines. As always, the virtual memory is just a logical allocation. All reads and writes MUST be via the physical memory. Note that `pemmu_malloc` should only allocate memory (book keeping) and NOT cause a page fault.
2. `pemmu_free` - frees the address that was returned by a `pemmu_malloc` call.
3. `pemmu_write` - write a value via the `pemmu_write` function to an address that was 'allocated' using the `pemmu_malloc` function call. If the address does not currently have a valid page table mapping, a

page fault occurs, which is then appropriately handled by bringing in the page from disk to physical memory. If there is no free frame in physical memory, a frame is freed up in accordance with the **page replacement policy** (which was specified by the testing framework).

4. `pemmu_read` - follows the same flow as write, except it reads the value stored at a specified address via the `pemmu_read` function call.

An address must only be written to or read from after it has been allocated via the `pemmu_malloc` function call. Any reads/writes to addresses that are not malloced by `pemmu_malloc` should throw an error. However, you are assured that the very first read from a virtual address is preceded by at least one write to that address.

The `pemmu_malloc`, `pemmu_free`, `pemmu_write` and `pemmu_read` functions provided to you in `interface.h` should not be modified. These are the top level functions that the testing framework will interact with. These functions in turn call corresponding API's listed in Section 2. You are only required to implement these functions. ***Please ensure that you do not change the signature of any of the given APIs as they are critical to grading.***

2 Framework

You have been provided with a framework that is intended to assist you in the implementation of *penn-mmu*. Any files that are outside the *submission* directory must not be modified by you, we will use a stock version of these during grading so any changes you make will not be seen by us. All your code files must be within the *submission* directory.

Please **do not modify** any of the Makefiles including the *Makefile* in the *submission* directory. Ensure that your code **compiles and runs** in the VM using the provided Makefiles. (You can modify the *Makefile* in the *submission* directory during development, though.) The teaching staff will not spend time debugging Makefiles.

2.1 Given code files:

(Inside the *framework* directory - not to be modified)

1. *main.c* is the driver of the program, which invokes the required initialization and then calls a single test program.

2. *interface.c* implements the top level functions - `pemmu_malloc`, `pemmu_read`, and `pemmu_write`. It also contains the `set_memory_configuration` function which sets the memory configuration for *penn-mmu* and a helper function. `split_virtual_address` that splits a virtual address into page number and offset.

3. *logger.c* contains important logging functions that must be used at appropriate points in your code to comply with grading.

2.2 Functions that you must implement

You must implement the functions in the *virtual_memory_controller.c* file:

1. `master_initialization` must hold all of the initialization calls that your implementation requires since this is the only initialization that the testing framework will call.
2. `mmu_write` will be invoked by a higher level write function (from *interface.c*), and is expected to follow the API that has been provided.
3. `mmu_read` will be invoked by a higher level read function (from *interface.c*), and is expected to follow the API that has been provided.
4. `mmu_allocate_variable` will be invoked by a higher level malloc function (from *interface.c*), and is expected to follow the API that has been provided.
5. `mmu_free_variable` will be invoked by a higher level free function (from *interface.c*), and is expected to follow the API that has been provided.
6. `master_clean_up` must hold all of the clean up calls that your implementation requires before the program exits. This is the only clean up that the testing framework will invoke.

3 Logging and Grading

It is imperative that your solution uses the following logging functions appropriately.

1. `log_page_fault`
2. `log_translate`
3. `log_eviction`
4. `log_disk_write`
5. `log_add_TLB_entry`
6. `log_TLB_hit`
7. `log_TLB_miss`

Apart from the 7 logging functions mentioned above, there are a number of other logging utilities that are part of logging module. None of these extra functions need to be invoked by you. The invocations of these extra utility functions are scattered throughout the framework as necessary.

You have been provided test functions and their expected outcomes, it is in your interest to ensure that your solution's log output matches the provided reference logs **EXACTLY**, as this portion is autograded and is the primary basis for grading in this assignment. Each of the logging functions also increment some key performance counters. These performance counters are logged into a separate log file (with the suffix `_perf`). The test cases provided, represent the flavor of testing that your implementation of *penn-mmu* will go through. Feel free to modify and play around with these to gain a complete understanding of the framework.

4 Analysis (Extra Credit)

A significant component of this project is to understand that no one page replacement policy is universally the best solution. Data access patterns have a huge impact on the performance of replacement policies. In this section of the project, you will analyze and understand the different policies that *penn-mmu* implements. To help with this we have provided certain performance counters that must be used in your code.

4.1 Analysis Requirement

To facilitate a thorough understanding of replacement policies we ask that you submit one test program per condition outlined below. Your test programs are not required to be any more complicated than they need to be to satisfy the specific condition. All of your test cases need only work with the **TLB disabled**.
Please take the time to understand the provided testing framework and submit test programs that comply with the specified APIs. We will run your test cases on our reference implementation, so do not directly reference any custom functions for which an API was not provided to you. Test programs that do not comply with these requirements will not be graded.

Write one test case each that satisfies the condition where

1. FIFO performs better than Clock (`fifo_over_clock`)
2. Clock performs better than FIFO (`clock_over_fifo`)
3. LRU performs better than Clock (`lru_over_clock`)
4. Clock performs better than LRU (`clock_over_lru`)
5. FIFO performs better than LRU (`fifo_over_lru`)
6. LRU better than FIFO (`lru_over_fifo`)

where "performs better" in this context simply means having fewer evictions from main memory. To streamline grading, stub code with the function signatures are provided in *analysis_programs.c*.

5 Testing Framework

This project will be auto-graded with the help of an open source testing framework called `CUTest`, which enables Java style unit testing in C. The test framework contains a vast suite of targeted unit tests that are written to test different areas of your *penn-mmu* design. A small portion of these tests are provided to you as part of the framework along with the logs from the reference solution. These logs are provided to help you understand what is expected from your solution. The auto grading script will run the complete suite of tests against your submission and will compare the output logs against the reference solution logs. The testing framework and the sample test suite is provided under the *tests* directory. The top level testing framework functions are implemented in *test_framework.c*. These are :

1. `test_framework_initialize`: Performs all the initialization before the start of a test.
2. `test_framework_cleanup`: Performs all the clean up once the test completes.

3. `run_cutest_framework`: Runs the testing framework.

4. `set_config`: Wrapper function to set the memory configuration of the framework.

The logs from each test are written into separate files which are created and closed before and after the test respectively. The memory configuration (including enabling and disabling the TLB) can be configured according to the requirement of each test using the *set_memory_configuration* function.

6 Developing and Testing Your Code

You should be able to run *make -B* in the top-level directory to successfully build *penn-mmu*.

The general syntax to run *penn-mmu* is as follows:

```
penn-mmu <test-type> <data-type> <memory-config> <replacement-policy> <timeout>
```

The various <test-type>, <memory-config> and <replacement-policy> codes and descriptions are available in the *test_framework.h* file in the *tests* directory. The different <data-type> codes are available in the *interface.h* file in the *framework* directory.

For example:

```
penn-mmu 1 1 1 1 120
```

will run

```
Test : ADDRESS_ALLOCATION_SANITY_TEST
Data Type : BYTE,
Memory Config : MEM_CONFIG_A
Replacement Policy : FIFO_REPLACEMENT_POLICY
Timeout : 120 seconds
```

In the provided framework

1. `test-type` supports the values : 0,1,2,3,9,10,11,12,13,15,16,17,18

2. `data-type`: supports the vales : 1,2,3,4

3. `memory-config`: supports the values : 1,7

4. `replacement-policy`: supports the values : 1,2,3

The testing framework will enforce a timeout of 120 seconds for all the tests in the grading framework. Your implementation should be efficient enough to pass this timing requirement. When testing out your submission against the provided tests, please make use of the optional <*timeout*> argument to make sure your tests pass within the stipulated time. Please note that the tests in the grading framework will include some advanced stress cases that will easily take longer to run when compared to the tests provided with the framework. Do not use the 120 seconds as a benchmark to pass the basic test cases in the framework.

7 Tips on approaching this project

Given the complexity of the project, listed are some tips/suggestions on how to approach the project.

- The entire project revolves around designing and filling out the functions in *virtual_memory_controller.c*. Study the interface in *virtual_memory_controller.h* carefully and understand what each function needs to do. Once this is clear, move on to the design process and map out the different design modules these functions will need to call. Keep the code as modular as possible since this will help a lot with integration and debugging.
- The project has a clear divide from a design perspective - address allocation and translation. All the address allocation tests can be run independent of the translation tests. However, in order to run the translation tests, the address allocation section of the project needs to be completed. Having said that, it is recommended to design both of these sections in parallel.
- Though the project has a lot of moving parts, one of the key parts of the project would be the implementation of the physical memory and its integration with the swap file. Since the swap file is an actual file on disk, understanding how to move data between the physical memory and the disk is a critical aspect of the project.
- Make use of the user defined tests (test ID 23 and 24) to write your own tests and test your code as you are developing.

8 What to turn in

You must provide each of the following for your submission, *no matter what*. Failure to do so may reflect in your grade.

1. `README` file. In the `README` you will provide
    - Your name and Pennkey
    - A list of submitted source files (not including anything outside the submission directory)
    - Overview of work accomplished
    - Description of code and code layout
    - Whether you completed the extra credit (Analysis) portion
    - General comments and anything that can help us grade your code
2. Your code. You may think this is a joke, but at least once a year someone forgets to include it.

9 Submission

This project will be submitted on Canvas. From the `submission` subdirectory, run the following command exactly as specified to create a compressed tarball.

```
tar cvaf groupnumber.tar.gz *.c *.h README
```

Replace "groupnumber" with your group number, e.g. `1.tar.gz` as the filename. After submission, a good sanity check would be to download your submission and verify that it has all of the required files.

10 Tentative Grading Guidelines

Each team will receive a group grade for this project; however, individual grades may differ. This can occur due to lack of individual effort, or other group dynamics. The team Git repository will be monitored and used to determine individual contribution. In most cases, everyone on a team will receive the same grade. Below is the grading guideline.

- 10% Milestone
- 5% Documentation
- 85% Functionality
- Analysis (10 points extra credit)

Please note that the grading scheme may be modified as the assignment progresses. The above is more of a guideline that highlights all of the factors that you will be graded on.

Also note that general deductions may occur for a variety of programming errors including memory violations, lack of error checking, etc.
Use the provided sample test suite and log files to achieve the best version of your solution. Do not rely solely on the sample tests, as they are just a subset of the final test bench. Use the provided framework to develop your own test cases.

11 Milestone (10%)

In order to ensure that students stay on track and receive early feedback, we require all groups to submit an intermediate solution before the deadline. For the milestone, students are expected to have completed the designing of the different data structures to be used in the project along with the Memory Allocation part of the MMU (`mmu_malloc()` and `mmu_free()`).

For the milestone, you should be able to pass the address allocation tests.

Do include a README in this submission, including:

- An accounting of the work you have completed thus far
- Description of all the data structures

This milestone is designed to guide your workflow. As such, we will not grade it in significant detail.

12 Attribution

There is a large amount of material available online that explains and implements different kinds of virtual memory simulators. It is in your best interest to refrain from looking any of these specific implementations up. Not only will you be disciplined for violating the Academic Integrity Policy, but it will also be unlikely to help since there is a specific framework and format that must be followed. That being said, there may be

sources online that you use to further your understanding of the working of the system. The primary rule to keep you safe from plagiarism/cheating in this project is to attribute in your documentation any outside sources you use. This includes both resources used to help you understand the concepts, and resources containing sample code you incorporated in your simulator. The course text and APUE need only be cited in the later case. Use external code sparingly.