

# Project 3 -- memory manager

---

Worth: 15 points

Assigned: February 24, 2022

Due: March 23, 2022

## 1. Overview

---

In this project, you will design and implement a *pager*, which is the part of the kernel that manages application processes' virtual address spaces. Your pager will manage a *portion* of each application process's address space; we call this portion the *arena*. Pages in the arena will be stored in physical memory, in a swap file, or in a regular file. Your pager will manage these resources on behalf of all the applications it manages.

Your pager will implement system calls that applications can use to create, copy, and destroy address spaces, allocate space in an existing address space, and switch between address spaces. Your pager will also implement the interrupt handler for memory faults.

This handout is organized as follows:

- [Section 2](#) describes the overall structure of the system.
- [Section 3](#) describes the simulated hardware used in this project.
- [Section 4](#) describes the system calls that applications can use to communicate explicitly with the pager.
- [Sections 5 and 6](#) are the main sections; they describe the functionality that you will implement in the pager and how to design your pager to minimize work.
- [Section 7](#) describes how your pager will maintain the emulated page tables and access physical memory and files.
- [Section 8](#) gives some hints for doing the project,
- [Sections 9-12](#) describe the test suite and project logistics/grading.

## 2. System structure

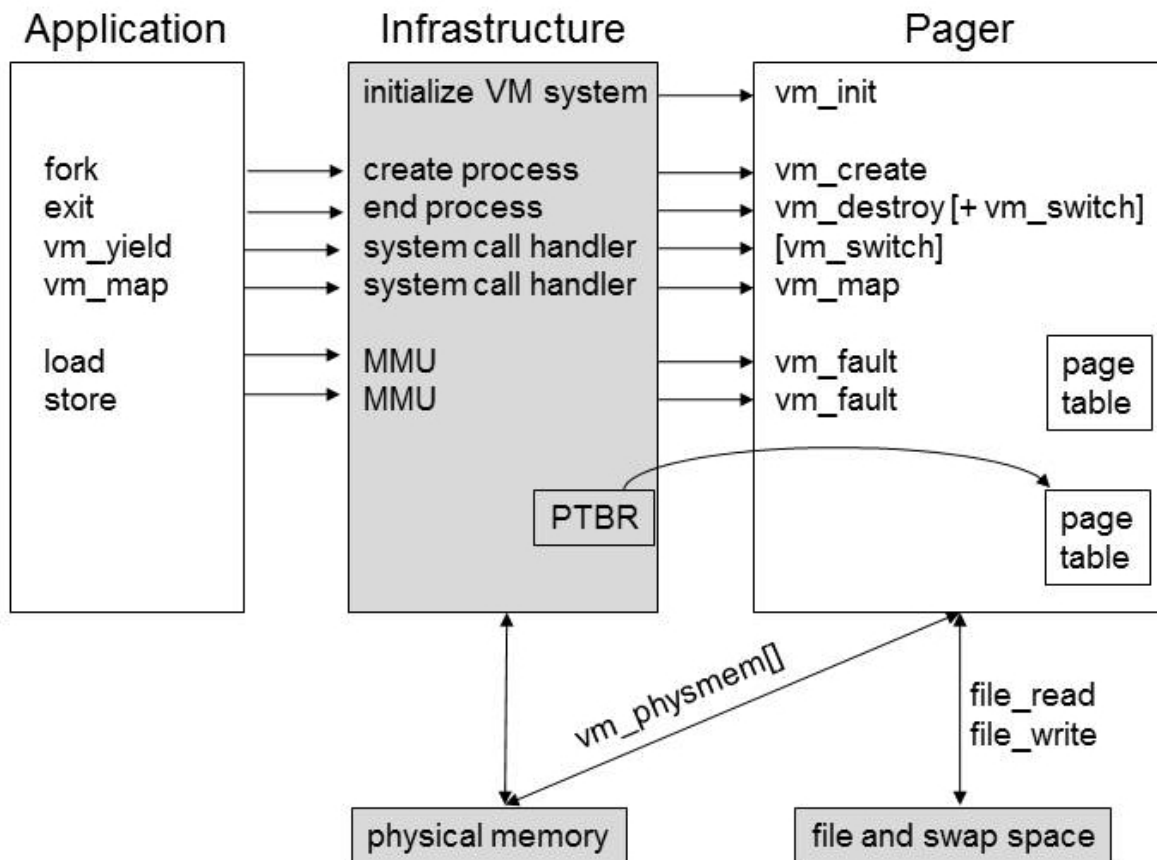
---

The system has two types of entities: application processes and the kernel (you are writing the pager part of the kernel). Application processes communicate with the kernel via system calls and page faults. In turn, the kernel provides processes with the address space abstraction by reading and writing physical memory, files, swap space, page tables, and the page table base register.

(The kernel also has an address space, but you are not responsible for managing the kernel's address space).

Applications interact with the kernel through the following mechanisms (summarized by the diagram below):

- An application requests service from the kernel by making system calls. This project deals with the following system calls: `fork`, `exit`, `vm_yield`, and `vm_map`. A system call invokes the computer's exception handling mechanism, which transfers control safely to the registered kernel handler for that system call.
- An application also transfers control to the kernel when it executes a load instruction to an address that is not read-enabled, or it executes a store instruction to an address that is not write-enabled. On such accesses, the MMU triggers a page fault, and the exception handling mechanism transfers control to the kernel's page fault handler. The MMU retries the faulting instruction after the page fault handler returns.
- When the application executes a load or store instruction to an address that is resident in memory, and the access is allowed by the page's protection, the MMU translates the virtual address to a physical address using the page table entry (PTE) for that address, which is stored in the page table pointed to by the page table base register (PTBR). The processor then accesses that physical address.

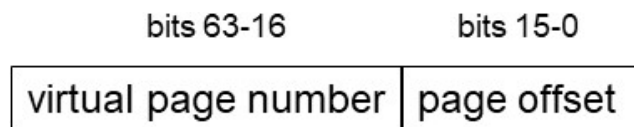


Items in [brackets] may or may not be called, depending on what processes are running. Note that there are two versions of `vm_map`: one in the application and one in the pager. The application-side `vm_map` is a system call wrapper and is called by the application process. When the application calls this function, the infrastructure invokes the corresponding system call in your pager. `vm_yield` is another system call wrapper and may cause the infrastructure to call `vm_switch`. The declarations for these functions are in `vm_app.h` and `vm_pager.h`.

We provide the software infrastructure shown in grey. This infrastructure emulates the MMU and exception-handling functionality of normal hardware, as well as physical memory and file and swap space. To use this infrastructure, each application that uses the pager must include `vm_app.h` and link with `libvm_app.o`, and your pager must include `vm_pager.h` and link with `libvm_pager.o`. Linking with these libraries enables application processes to communicate with the pager in the same way that applications on real hardware communicate with real operating systems. Applications issue load and store instructions (compiled from normal C++ variable accesses), and these are translated or faulted by the infrastructure exactly as in the above description of the MMU. The infrastructure transfers control on faults and system calls to the pager, which receives control via function calls. The infrastructure also invokes your pager's `vm_init` function when the pager starts.

### 3. Memory management hardware

The system uses a single-level page table. A virtual address is composed of a virtual page number and a page offset:



When the application executes a load or store instruction, the MMU checks the protection bits of the virtual page being accessed, translates the virtual address to a physical address, and accesses the page. To carry out these tasks, the MMU uses information in the *page table entry* (PTE) for the virtual page being accessed. The array of PTEs for a process is called a *page table*. In this project, the page table stores an entry for each virtual page in the arena.

Page tables are stored in the kernel's address space. The system's *page table base register* (PTBR) contains the kernel address of the page table currently in use. The PTBR is a variable that is declared and defined by the infrastructure (but will be controlled by your pager).

[vm\\_arena.h](#) describes the arena of a process, which are the addresses in the range [ VM\_ARENA\_BASEADDR , VM\_ARENA\_BASEADDR + VM\_ARENA\_SIZE ) .

The following portion of [vm\\_pager.h](#) describes a page table entry, page table, and page table base register. Note that the MMU for this project does not automatically maintain dirty and reference bits. Instead, these state bits will be maintained by your pager.

```
/*
 * *****
 * * Definition of page table structure *
 * *****
 */

/*
 * Format of page table entry.
 *
 * read_enable=0 ==> loads to this virtual page will fault
 * write_enable=0 ==> stores to this virtual page will fault
 * ppage refers to the physical page for this virtual page (unused if
 * both read_enable and write_enable are 0)
 */
struct page_table_entry_t {
    unsigned int ppage : 20;          /* bit 0-19 */
    unsigned int read_enable : 1;      /* bit 20 */
    unsigned int write_enable : 1;     /* bit 21 */
};
```

```
/*
 * Format of page table.  Entries start at the beginning of the arena,
 * i.e., ptes[0] is the page table entry for the first virtual page in the arena
 */
struct page_table_t {
    page_table_entry_t ptes[VM_ARENA_SIZE/VM_PAGESIZE];
};

/*
 * MMU's page table base register.  This variable is defined by the
 * infrastructure, but it is controlled completely by the student's pager code.
 */
extern page_table_t *page_table_base_register;
```

## 4. Application semantics

---

This section describes the semantics of an address space that are provided to applications. Note that these describe the behavior of an address space *from the perspective of an application*. As with most operating system abstractions, the physical reality may differ from the illusion seen by an application. For example, the application sees all valid pages as being in memory, but these pages may actually exist only on disk.

### 4.1. Swap-backed and file-backed pages

The system supports two types of virtual pages: swap-backed and file-backed.

A swap-backed virtual page is initialized to all zeroes when the page is added to the address space. The data in a swap-backed page are lost when the process exits. A swap-backed page is private to a process, i.e., it is not shared with any other virtual page, either in the same process or other processes.

A file-backed virtual page corresponds to a specific block of a specific file. The data in these files live beyond the lifetime of application processes. A particular file block can be mapped simultaneously to multiple processes or to multiple pages in one process, and all virtual pages that are mapped to this file block refer to the same data (i.e., they are aliases). For example, stores to one virtual page mapped to this file block are seen by loads to all virtual pages mapped to this file block.

### 4.2. System calls

Applications use four system calls to communicate explicitly with the pager: `vm_map`, `vm_yield`, `fork`, and `exit`. The prototypes for `vm_map` and `vm_yield` are given in the file `vm_app.h`; the prototypes for `fork` and `exit` are given in the Linux manual pages. Most application programs only use `vm_map` explicitly, since `fork` and `exit` are called implicitly when a process starts and ends, and `vm_yield` is used only to control process scheduling.

A process calls `vm_map(filename, block)` to ask for the lowest invalid page in its arena to be declared valid. `vm_map` returns the address of the new virtual page. E.g., if the valid part of the arena is `0x600000000-0x60003ffff`, the return value of the next `vm_map` call will be `0x600040000`, and the resulting valid part of the arena will be `0x600000000-0x60004ffff`. `vm_map` can be used to map swap-backed or file-backed pages. A swap-backed page is mapped if `filename` is `nullptr`; otherwise a file-backed page is mapped to the specified `filename` and `block`. When mapping a file-backed page, `filename` should be a null-terminated C string and must reside completely in the valid portion of the arena. (FYI, the `vm_map` interface is similar to the `mmap` call provided by Linux. The interfaces you normally use to manage dynamic memory (`new`, `delete`, `malloc`, and `free`) are built on top of `mmap`.)

A process calls `vm_yield` to ask the pager to run another process. If no other process is running, `vm_yield` has no effect. The infrastructure's scheduling policy is non-preemptive: the current application process runs until it calls `vm_yield` or exits.

A (parent) process calls `fork` to create a new (child) process. The child process starts with a copy of the parent's address space, including a copy of the mappings and data of the parent's arena. Note that the data in a child's swap-backed page is only a copy of the data in its parent's page; swap-backed virtual pages are not shared with each other from the processes' perspectives.

A process calls `exit` to ask the pager to destroy its address space. Data in a process's swap-backed pages are lost when the process exits. File-backed pages are not affected when a process exits.

Here is an example application program that uses the pager.

```
#include <iostream>
#include <cstring>
#include <unistd.h>
#include "vm_app.h"
```

```
using std::cout;
```

```
int main()
{
```

```
    // Allocate swap backed page from the arena & /
```

```
/* Allocate swap-backed page from the arena */
char *filename = (char *) vm_map(nullptr, 0);

/* Write the name of the file that will be mapped */
strcpy(filename, "lampson83.txt");

/* Map a page from the specified file */
char *p = (char *) vm_map (filename, 0);

/* Print the first part of the paper */
for (unsigned int i=0; i<1937; i++) {
    cout << p[i];
}
}
```

## 5. Pager functions

---

This section describes the functions you will implement in your pager: `vm_init`, `vm_create`, `vm_switch`, `vm_map`, `vm_fault`, and `vm_destroy`. These functions are declared in [vm\\_pager.h](#). The rest of the kernel (e.g., `main`) is implemented in [libvm\\_pager.o](#) and will call your pager functions in response to various events in the system (e.g., system initialization, system calls, page faults).

This section describes when each pager function is called by the infrastructure and, in general terms, what is the purpose of each function. Your job in this project is to design and implement the specifics of your pager to carry out these purposes as efficiently as possible (this is discussed more in [Section 6](#)).

### 5.1. `vm_init(unsigned int memory_pages, unsigned int swap_blocks)`

The infrastructure calls `vm_init` when the pager starts. `memory_pages` and `swap_blocks` specify the number of physical memory pages and swap blocks in the system. `vm_init` should set up whatever data structures you need to begin accepting `vm_create` and subsequent requests from processes.

You may assume that, once started, the pager never exits.

### 5.2. `vm_create(pid_t parent_pid, pid_t child_pid)`

The infrastructure calls `vm_create` when a *parent* process creates a new *child* process via the `fork` system call. The pager should initialize whatever data structures it needs to manage the

new child process. In addition, it should cause the child's arena to be a duplicate of the parent's arena. That is, each page in the child's arena should have the same mapping as the corresponding page in the parent's arena (both swap-backed and file-backed pages), and the data in the child's arena should appear (to the child) to be initialized to the values that were in the parent's arena when the child was created. The pager will use copy-on-write sharing ([Section 6.3](#)) to defer copying the parent's data, so creating the child will not affect the parent's residence or reference bits.

If the parent process is not already known to the pager, `vm_create` should assume the parent's arena is empty. This occurs when the parent process (e.g., `/bin/bash`) was not linked with `libvm_app.o`.

**Core/advanced:** The core version of the pager need only handle `vm_create` calls by processes with empty arenas. The advanced version of the pager should be able to handle `vm_create` calls by processes with empty or non-empty arenas.

Note that the child process is not running at the time `vm_create` is called. The child process will run when it is switched to via `vm_switch`.

`vm_create` should ensure that there are enough available swap blocks to hold all swap-backed virtual pages (this is called *eager swap reservation*). If there are not enough free swap blocks, `vm_create` should return `-1`. The benefit of eager swap reservation is that applications know at the time of `vm_create` (and `vm_map`) that there is no more swap space, rather than when a page needs to be evicted. `vm_create` should return `0` on success.

### 5.3. `vm_switch(pid_t pid)`

The infrastructure calls `vm_switch` when the OS scheduler runs a different process. `vm_switch` should take whatever action is needed to change address spaces to the process with ID `pid`.

### 5.4. `vm_map(const char *filename, unsigned int block)`

`vm_map` is called when a process wants to make another virtual page in its arena valid. `vm_map` should return the address of the new valid virtual page. E.g., if the arena before calling `vm_map` is `0x600000000-0x60003ffff`, the return value of `vm_map` will be `0x600040000`, and the resulting valid part of the arena will be `0x600000000-0x60004ffff`. `vm_map` should return `nullptr` if it cannot handle the request (e.g., the arena is full).

#### 5.4.1. Swap-backed pages

If `filename` is `nullptr`, `block` is ignored, and the new virtual page is swap-backed and private (i.e., not shared with any other virtual page). The application should see each byte of a newly mapped swap-backed virtual page as initialized with the value `0`.

Swap-backed pages are stored in the system's swap file when there are no free physical pages.

`vm_map` should ensure that there are enough swap blocks to hold all swap-backed virtual pages (eager swap reservation), otherwise `vm_map` should return `nullptr`.

### 5.4.2. File-backed pages

If `filename` is not `nullptr`, it points to a null-terminated C string *in the application's address space*, which specifies the name of the file that backs the new virtual page. `filename` is specified relative to the pager's current working directory.

The C string pointed to by `filename` should reside completely in the valid portion of the application's arena. Remember that `filename` is a pointer to the **application's** address space, so `vm_map` will need to access the C string pointed to by `filename` via physical memory. Your pager should treat `vm_map`'s accesses to the application's data exactly as if they came from the application program for the purposes of protection, residence, and reference bits.

`vm_map` should return `nullptr` if `filename` is not completely in the valid part of the arena. Other than checking that the C string is in the valid part of the arena, `vm_map` need not (and should not) verify that `filename` and `block` are legal (hint: think about when these will be checked).

At any given point in time, zero or more virtual pages may be mapped to a given file and block. All virtual pages mapped to the same (`filename`, `block`) are shared with each other. The pager should manage all members of a set of shared virtual pages as a single virtual page. E.g., a set of shared virtual pages should be represented as a single node on the clock queue.

## 5.5. `vm_fault(const void *addr, bool write_flag)`

`vm_fault` is the kernel's page fault handler. It is called by the infrastructure when an application reads a page that is not read-enabled, or writes a page that is not write-enabled. `addr` contains the faulting address; `write_flag` is set if the access was a write.

`vm_fault` should return 0 after successfully handling a fault. `vm_fault` should return -1 if it cannot handle the fault (e.g., the address is to an invalid page).

Your pager determines which accesses in the arena will generate faults by setting the `read_enable` and `write_enable` fields in the page table. The actions performed in `vm_fault` will depend on the state of the virtual page, and why the pager wanted accesses to the page to generate a fault.

## 5.6. `vm_destroy()`

`vm_destroy` is called by the infrastructure when the corresponding application exits. This gives the pager a chance to clean up any resources used by the exiting process, such as page tables, physical pages, and swap blocks.

## 6. Pager design

---

A major part of this project is designing your pager to minimize the work needed to provide the required address space abstractions to applications. This section describes various aspects of how to design your pager to minimize work.

### 6.1. Deferring and avoiding work

The main way to minimize work is to avoid and defer work whenever possible. There are points in this project where careful state maintenance can help you avoid doing work. Whenever possible, avoid work. For example, if a page that is being evicted does not need to be written to disk, don't do so. (However, the victim selection algorithm in [Section 6.2](#) must be used as specified; e.g., don't change the victim selection to avoid writing a page to disk).

Similarly, there are many points in this project where you have some freedom over when page copying, page faults, and disk I/O happen. Your pager should defer such work as far into the future as possible.

If you could possibly defer or avoid some action at the possible expense of making another action necessary, keep in mind that incurring a fault (about 5 microseconds on current hardware) is cheaper than copying a page (30 microseconds), which is in turn cheaper than a disk I/O (10 milliseconds). For instance, if you have a choice between taking an extra page fault and causing an extra disk I/O, you should prefer to take the extra fault.

### 6.2. Replacement and eviction

The virtual memory abstraction allows the address spaces managed by the pager to exceed the size of physical memory available to the pager. This is a classic use of caching: storing a subset of data in a fast but small space (in this case, physical memory), while the rest of the data lives in a large but slow space (in this case, files).

With virtual memory, some virtual pages will be resident (in physical memory) and some will be non-resident (not in physical memory). Your pager will provide the illusion to applications that all virtual pages are resident (in fact, other than timing, applications should see no difference between resident and non-resident pages). To maintain this illusion, your pager should arrange for faults to occur when an application accesses a non-resident page. When such a fault occurs, the pager should find a physical page to associate with the virtual page. If there are no free physical pages, the pager should create a free physical page by evicting a virtual page that is resident.

Use the *clock* (also called FIFO with second-chance) algorithm to select a victim. The clock queue is an ordered list of all physical pages that are in use and are candidates for eviction. To select a victim, remove and examine the page at the head of the queue. If the head page has been accessed since it was last enqueued, it should be moved to the tail of the queue, and victim selection should proceed to the next page in the queue. If the page at the head has not been accessed since it was last enqueued, its virtual page should be evicted. All physical pages that are in use are treated the same when selecting a victim page to evict. When a virtual page is made resident, it should be placed at the tail of the clock queue and marked as referenced.

When the pager evicts a virtual page, it may need to write the page's data to the backing storage for that page (either the swap file or the file block specified when that page was mapped). Since disk I/O is expensive, the pager should only write data to the file if the page is *dirty*, i.e., its contents differ from the contents in the file.

To implement the clock replacement algorithm and avoid writing data unnecessarily to disk, you will need to maintain reference and dirty bits for each resident virtual page. Since the MMU for this project does not maintain dirty or reference bits, your pager will need to maintain these bits in its own data structure, by (1) setting the protection bits to generate page faults on relevant accesses and (2) updating the state of the faulting virtual page in `vm_fault`.

### 6.3. Copy-on-write sharing

Your pager may share a physical page, file block, or swap block among multiple virtual pages. It should manage a set of virtual pages that share physical resources as a single virtual page. E.g., a set of virtual pages that share a physical page should be represented as a single node on the clock queue.

Sharing a physical resource can be used for two different purposes:

- It can help the pager provide the abstraction of a shared virtual page, i.e., when multiple virtual pages are mapped to a particular file block. In this case, sharing takes place both at the physical level (from the hardware's perspective) and at the virtual level (from the application's perspective).
- It can help the pager reduce resource usage and save work, even when the virtual pages that are using this resource are *not* shared from an application's perspective. The rest of this section is about this use of sharing, which we call *copy-on-write* sharing.

Copy-on-write sharing is useful when multiple virtual pages are not shared from an application's perspective, but have the same data values. For example, when a parent process calls `fork`, the data in the child's pages are the same values as the data in the parent's pages. As long as the data values are guaranteed to match, the parent and child's virtual pages can both be stored in the same physical resource. However, when the data changes in either the parent's or child's

virtual page, both virtual pages will no longer be able to use the same physical resource, so another copy will need to be made. This technique is called copy-on-write because a copy operation is deferred until one of the virtual pages is written.

Your pager should use copy-on-write sharing whenever it can deduce (without examining the data on the page) that two or more virtual pages have the same data but are not being shared from an application's perspective.

A copy operation will occur when one of the virtual pages using copy-on-write sharing is written. Your pager will initiate the copy operation in response to an application's store instruction, and should carry out this copy operation as follows:

1. Read the data from the page, and write it into a temporary buffer. The temporary buffer should be a normal variable in the kernel's address space (not a page allocated from physical memory). This read should have the same effect on the contents of physical memory, page tables, and the clock queue as if the application process had read the page.
2. Write the data (from the temporary buffer) to a new physical page. The new physical page should be assigned to the virtual page whose write triggered the copy-on-write operation; the other virtual page(s) should retain the old physical page. This write should have the same effect on the contents of physical memory, page tables, and clock queue as if the application process initiating the copy-on-write is writing the page (which, of course, it is).

The advantage of using a temporary buffer (instead of copying the data directly between two physical memory pages) is it avoids the need to keep both the old page and the new page in physical memory at the same time.

Hint: Writing to a virtual page that is being shared via copy-on-write should have the same effect on the system as reading it, then writing it.

## 6.4. Pinning memory

Kernels sometimes guarantee that certain virtual memory pages will not be evicted. We refer to this technique as *pinning*. Pinning a page may improve performance or simplify the kernel. In some systems (though not this project), pinning a page is needed to eliminate circular dependencies.

In this project, you will use pinning to reserve a physical page in which all bytes have the integer value zero (not the character '0'). This *zero page* should be allocated and initialized with zeroes when the pager starts, and it should never be evicted. Having a physical page full of zeroes is useful in this project because all swap-backed pages are (from the perspective of applications) initialized with zeroes when they are first mapped.

Use the zero page to reduce faults for swap pages. Consider what `read_enable` and `write_enable` should be for virtual page(s) that are mapped to the zero page, and remember to avoid and defer work whenever possible.

## 7. Interface used by pager to access the simulated hardware

The following portion of `vm_pager.h` describes the variables and utility functions for accessing physical memory and files.

```
/*
 * *****
 * * Interface for accessing files. Implemented by infrastructure *
 * *****
 *
 * You may assume that, while the pager is running, no other process
 * accesses its files. You may also assume that once a file block is
 * accessed successfully, it will remain accessible (although the
 * reverse may not be true; a file that cannot be accessed now may be
 * accessible later).
 */

/*
 * file_read
 *
 * Read page from the specified file and block into buf.
 * If filename is nullptr, the data is read from the swap file. buf should
 * be an address in vm_physmem.
 * Returns 0 on success; -1 on failure.
 */
extern int file_read(const char* filename, unsigned int block, void* buf);

/*
 * file_write
 *
 * Write page from buf to the specified file and block.
 * If filename is nullptr, the data is written to the swap file. buf should
 * be an address in vm_physmem.
 * Returns 0 on success; -1 on failure.
 */
extern int file_write(const char* filename, unsigned int block, const void* buf);

/*
 * *****
 * * Public interface for the physical memory abstraction. *
 * * Defined in infrastructure. *
 */
```

```

* ****
*
* Physical memory pages are numbered from 0 to (memory_pages-1), where
* memory_pages is the parameter passed in vm_init().
*
* Your pager accesses the data in physical memory through the variable
* vm_physmem, e.g., ((char *)vm_physmem)[5] is byte 5 in physical memory.
*/
extern void* const vm_physmem;

```

Physical memory is structured as a contiguous collection of  $M$  pages, numbered from 0 to  $M-1$ .  $M$  is settable through the `-m` option when you run the pager (e.g., by running `pager -m 4`). The minimum number of physical pages is 4, the maximum is 1024, and the default is 4. Your pager can access the data in physical memory via the array `vm_physmem`.

Swap space is structured as a contiguous collection of  $S$  blocks, numbered from 0 to  $S-1$ .  $S$  is settable through the `-s` option when you run the pager (e.g., by running `pager -s 256`). The minimum number of swap blocks is 4, the maximum is 1024, and the default is 256.

Regular files and the swap file are accessed via `file_read` and `file_write`. If `filename` is `nullptr`, these functions access the swap file; otherwise they access the specified file. Each call accesses one disk block, which is the same size as a physical memory page. `file_read` copies data from a file to a physical page; `file_write` copies data from a physical page to a file.

## 8. Hints

---

First, draw a finite state machine for the life of a virtual page, from creation via `vm_map` to destruction via `vm_destroy`. Consider what events can happen to a page throughout its lifetime, and what state you will need to keep to represent each state. As you design the state machine, try to identify all of the places in the state machine where work can be deferred or avoided. Most of this project hinges on getting this state machine correct.

You may find it helpful to draw two state machines, one for swap-backed pages and another for file-backed pages (they are similar but not identical).

While the state machine is a good way to visualize the lifetime of a page, writing code that directly implements each state often leads to a lot of redundancy. Instead, think about how to factor common functionality out of similar states. This is important to save you time in debugging, and will be a point of emphasis during style grading.

Approach the project incrementally, rather than all at once. First, write **and test** a pager that only handles swap-backed pages for a single process. After this is working, then add support for file-

backed pages, then add support for fork. However, you'll need to plan ahead for the notion of sharing hardware resources among several virtual pages (see [Section 6.3](#)).

This project is much easier to solve if you have clear abstractions representing the various entities, with crisp interfaces between them. It may take several iterations to get a "good" set of these abstractions and interfaces. Often, you can't really tell that a set is not "good" until you are partway through building it. Therefore, **do not be shy about starting from a clean sheet**. In this project it can often be faster to start over with a better design than to try to get an existing architecture to work. In particular, it is important to have clear notions of ownership and responsibility.

Read-faults should typically make the virtual page read-only ( `read_enable=1` , `write_enable=0` ), but not always.

Use assertion statements copiously in your process library to check for unexpected conditions generated by bugs in your program. These error checks are essential in debugging complex programs because they help flag error conditions early.

We recommend having **exactly one place** in your solution that takes as input the OS' view of a virtual page and computes the protection bits in the corresponding hardware page table entry. In addition to eliminating redundancy, this can also be very useful as part of a larger **correctness predicate** that you can use inside an assertion.

For many students, the main intellectual challenge of this project is thinking in two address spaces (and therefore trust domains) at the same time--something that is particularly important when working with file backed pages. There is no reason to believe that an application is well-behaved in passing arguments to `vm_map` . It sometimes takes creativity to think of the ways in which the application might be wrong, but your pager must behave correctly in all cases.

## 9. Test cases

---

An integral (and graded) part of writing your pager will be to write a suite of test cases to validate any pager. This is common practice in the real world--software companies maintain a suite of test cases for their programs and use this suite to check the program's correctness after a change. Writing a comprehensive suite of test cases will deepen your understanding of virtual memory, and it will help you a lot as you debug your pager. To construct a good test suite, trace through different transition paths that a page can take through a pager's state machine, then write a short test case that causes a page to take each path.

Each test case for the pager will be a short C++ application program that uses a pager via the interface described in [Section 4](#) (e.g., the example program in [Section 4](#)). Each test case should be run without any arguments and should not use any input files.

Your test suite may contain up to 30 test cases. Each test case may cause a correct pager to generate at most 500 KB of output and must take less than 60 seconds to run. These limits are much larger than needed for full credit. You will submit your suite of test cases together with your pager, and we will grade your test suite according to how thoroughly it exercises a pager. [Section 10](#) describes how your test suite will be graded.

The name of each test case should start with `test` and end with `.cc` or `.cpp`. Each test case will also specify the number of physical memory pages to use when running the pager (the `-m` option) for that test case. This parameter will be specified as a dot-separated field in the name of the test case file, just before the `.cc` or `.cpp` filename extension. For example, a test case that tests eviction and configures the pager to have 4 memory pages might be named `testEvict.4.cpp`. Remember that the minimum number of physical memory pages is 4 and the maximum is 1024. Your test cases may assume that `vm_init` is called with `swap_blocks=256`.

Your test cases may assume that the pager runs in a directory that has the following files: [lampson83.txt](#), [data1.bin](#), [data2.bin](#), [data3.bin](#), [data4.bin](#). Use these files for test cases that map pages to files.

Your test suite should test a full range of functionality, even if your pager does not implement all functionality. In particular, your test suite should create and test processes that start with non-empty arenas, even if you are only implementing the core version of the pager.

You should test your pager with both single and multiple applications running. Most of the test cases you submit need only be a single process, but some of the buggy pagers used to evaluate your test suite can only be exposed by multi-process test cases. Use `vm_yield` to coordinate the order in which processes run.

In writing your test cases, consider whether or not you've checked **every** transition in your state diagram(s). Without liberal assertions, some bugs may not be exposed without taking some **complete cycles** through the state machine.

## 10. Project logistics

---

Write your pager in C++17 on Linux. Use `g++ 9.1.0` (with `-std=c++17`) to compile your programs. To use `g++ 9.1.0` on CAEN computers, put the following command in your startup file (e.g., `~/.profile`):

```
module load gcc/9.1.0
```

You may use any functions included in the standard C++ library, except the C++ thread facilities. You should not use any libraries other than the standard C++ library. Your pager code may be in

multiple files. Each file name must end with `.cc` , `.cpp` , or `.h` and must not start with `test` .

This [Makefile](#) shows how to compile a pager and an application that uses the pager (adjust the file names in the Makefile to match your own program).

You are required to document your development process by having your Makefile run [autotag.sh](#) each time it compiles your pager (see Makefile above). [autotag.sh](#) creates a git tag for a compilation, which helps the instructors better understand your development process. [autotag.sh](#) also configures your local git repo to include these tags when you run `" git push "`. To use it, download [autotag.sh](#) and set its execute permission bit (run `" chmod +x autotag.sh "`). If you have several local git repos, be sure to push to github from the same repo in which you compiled your pager.

Here's how to run your pager and an application.

1. Start the pager. The infrastructure will print a message saying `Pager started with # physical memory pages` , where `#` refers to the number of physical memory pages.
2. After the pager has printed the `Pager started` message, run one or more application processes from a different terminal window on the same computer. These application processes will interact with the pager via the infrastructure. The same user must run the pager and the applications that use the pager, and all processes must run on the same computer.

If you want to run your application in `gdb` , you will probably find it useful to direct `gdb` to ignore `SIGUSR1` and `SIGSEGV` events (used by the project infrastructure). To do this, use the following command in `gdb` . Note that this only applies when you're running an application (not the pager) in `gdb` :

```
handle SIGUSR1 nostop noprint
handle SIGSEGV nostop noprint
```

We have created a private [github](#) repository for your group ( `eeecs482/<group>.3` ), where `<group>` is the sorted, dot-separated list of your group members' unqunames. Initialize your local repository by cloning the (empty) repository from github, e.g.,

```
git clone git@github.com:eeecs482/usernameA.usernameB.3
```

## 11. Grading, auto-grading, and formatting

---

To help you validate your programs, your submissions will be graded automatically, and the results will be provided to you. You may then continue to work on the project and re-submit. The results from the auto-grader will not be very illuminating; they won't tell you where your problem is or give you the test programs. The main purpose of the auto-grader is to help you know to keep working on your project (rather than thinking it's perfect and ending up with a 0). The best way to debug your program is to generate your own test cases, figure out the correct answers, and compare your program's output to the correct answers. This is also one of the best ways to learn the concepts in the project.

Hint: here is a (very rough) categorization of some of the test cases used by the auto-grader.

- 1-8,21-24,32-39: swap-backed pages; processes start with empty arenas
- 0,9-20,25-31,53-62,70: swap and file-backed pages; processes start with empty arenas
- 40-52: swap-backed pages; processes may start with non-empty arenas
- 63-69: swap and file-backed pages; processes may start with non-empty arenas

The student suite of test cases will be graded according to how thoroughly they test a pager. We will judge thoroughness of the test suite by how well it exposes potential bugs in a pager. The auto-grader will first run a test case with a correct pager and generate the correct output *from the pager* (on `stdout`, i.e., the stream used by `cout`) for this test case. The auto-grader will then run the test case with a set of buggy pagers. A test case exposes a buggy pager by causing the buggy pager to generate output (on `stdout`) that differs from the correct output. The test suite is graded based on how many of the buggy pagers were exposed by at least one test case. This is known as *mutation testing* in the research literature on automated testing.

You may submit your program as many times as you like, and all submissions will be graded and cataloged. We will use your highest-scoring submission, with ties broken in favor of the later submission. If any group member is in EECS 498-002, your highest-scoring submission will be chosen using a (2/3,1/3) weighted average of your core and advanced scores.

You must recompile and `git push` at least once between submissions.

The auto-grader will provide feedback for the first submission of each day, plus 3 bonus submissions over the duration of this project. Bonus submissions will be used automatically--any submission you make after the first one of that day will use one of your bonus submissions. After your 3 bonus submissions are used up, the system will continue to provide feedback for the first submission of each day. See the [FAQ](#) for why we use this policy.

Because your programs will be auto-graded, you must be careful to follow the exact rules in the project description. In particular:

- Your pager code should not print any output. The pager infrastructure prints messages to help you debug (and to allow the auto-grader to understand what the pager is doing).

- Do not modify the header files provided in this handout.

In addition to the auto-grader's evaluation of your program's correctness, a human grader will evaluate your program on issues such as documentation, coding style, the efficiency, brevity, and understandability of your code, compiler warnings, etc.. Although your pager is being run with a small number of pages, disk blocks, and processes, your algorithms and data structures should be optimized for larger numbers. Your final score will be the product of the hand-graded score (between 1-1.12) and the auto-grader score.

## 12. Turning in the project

---

[Submit](#) the following files for your pager:

- C++ program for your pager. File names should end in `.cc` , `.cpp` , or `.h` and must not start with `test` . Do not submit the files provided in this handout.
- Suite of test cases. Each test case should be in a single file. File names must follow the format described in [Section 8](#).

Each person should also describe the contributions of each team member using the following [web form](#).

The official time of submission for your project will be the time of your last submission.

Submissions after the due date will automatically use up your late days; if you have no late days left, late submissions will not be counted for you (though they may still count for other members of your group, if they have more late days available).

## 13. Files included in this handout ([zip file](#))

---

- [libvm\\_app.o](#)
- [libvm\\_pager.o](#)
- [vm\\_app.h](#)
- [vm\\_pager.h](#)
- [vm\\_arena.h](#)
- [lampson83.txt](#)
- [data1.bin](#)
- [data2.bin](#)
- [data3.bin](#)
- [data4.bin](#)
- [autotag.sh](#)

- [Makefile](#)

## 14. Experimental platforms

---

The files provided in this handout were compiled on RHEL 7. They should work on most other Linux distributions (e.g., Ubuntu) and on Windows Subsystem for Linux (WSL), but these are not officially supported.

We also provide an experimental version of the infrastructure for students who want to develop on MacOS (10.14 or later, including experimental support for ARM-based Macs). If you are developing on MacOS:

- Use [libvm\\_app\\_macos.o](#) instead of [libvm\\_app.o](#)
- Use [libvm\\_pager\\_macos.o](#) instead of [libvm\\_pager.o](#)
- Add `-D_XOPEN_SOURCE` to the compilation flags.













