Project 3 -- external pager

Worth: 11 points
Assigned: Thursday, October 17, 2013
Due: 6:00 pm, Thursday, November 7, 2013

1. Overview

This project will help you understand address spaces and virtual memory management.

In this project, you will implement an *external pager* (*pager* for short), which manages virtual memory for application processes. Your pager will manage a portion (called the *arena*) of the virtual address space of each application that uses it. Your pager will be single threaded, handling each request to completion before processing the next request. Pages in the arena will be stored in (simulated) physical memory or on (simulated) disk. Your pager will manage these two resources on behalf of all the applications it manages. The pager will handle address space creation, read and write faults, address space destruction, and simple argument passing between address spaces. In addition, your pager will handle two system calls: `vm_extend` and `vm_syslog`. An application uses `vm_extend` to ask the pager to make another virtual page of its arena valid. An application uses `vm_syslog` to ask the pager to print a message to its console.

This handout is organized as follows. Section 2 describes how the infrastructure for this project performs the same tasks as the MMU and exception mechanism on normal computer systems. Section 3 describes the MMU used in this project. Section 4 describes the system calls that applications can use to communicate explicitly with the pager. Section 5 is the main section; it describes the functionality that you will implement in the external pager. Section 6 describes how your pager will maintain the emulated page tables and access physical memory and disk. Section 7 gives some hints for doing the project, and Sections 8-11 describe the test suite and project logistics/grading.

2. Infrastructure and system structure

In a normal computer system, the CPU's memory management unit (MMU) and exception mechanism perform several important tasks. The MMU is invoked on every virtual memory access.

- On accesses to non-resident or protected memory, the MMU triggers a page fault or protection exception, transfers control to the kernel's fault handler, then retries the faulting instruction after the fault handler finishes.
- On accesses to pages that are resident in memory and are allowed by the page's protection, the MMU translates the virtual address to a physical address and accesses that physical address.
- Some MMUs automatically maintain dirty and reference bits; other MMUs leave this task to be handled in software. The MMU in this project does **not** maintain dirty or reference bits automatically. Instead, these will be maintained by your pager.

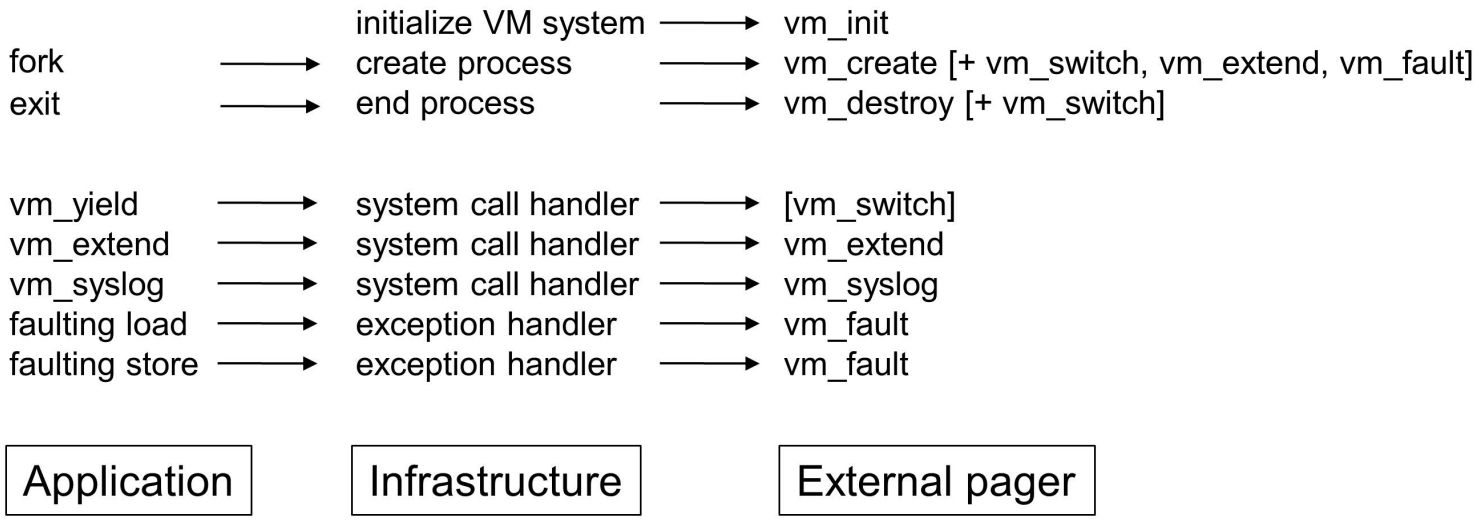
On most computer systems, system call instructions also invoke the exception mechanism. When a system call instruction is executed, the exception mechanism transfers control to the registered kernel handler for this exception.

We provide software infrastructure to emulate the MMU and exception functionality of normal hardware. To use this infrastructure, each application that uses the external pager must include `vm_app.h` and link with `libvm_app.a`, and your external pager must include `vm_pager.h` and link with `libvm_pager.a`. You do not need to understand the mechanisms used to emulate this functionality (in case you're curious, the infrastructure

uses `mmap`, `mprotect`, `SEGV` signal handlers, Unix domain sockets, and remote procedure calls).

Linking with these libraries enables application processes to communicate with the pager process in the same manner as 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.

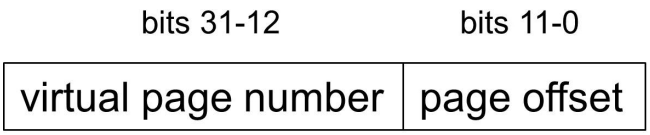
The following diagram shows how your pager will interact with applications that use the pager. An application makes a request to the system via the function calls `vm_yield`, `vm_extend`, and `vm_syslog`, or by trying to load or store an address that is non-resident or protected. Items in [brackets] may or may not be called, depending on what processes are running.



Note that there are two versions of `vm_extend` and `vm_syslog`: one in the application and one in the pager. The application-side `vm_extend` and `vm_syslog` are system call wrappers (implemented in `libvm_app.a`) and are called by the application process. When the application calls these functions, the infrastructure invokes the corresponding system call in the pager, which you will implement. `vm_yield` is another system call wrapper and may cause the infrastructure to call `vm_switch`. See the header files `vm_app.h` and `vm_pager.h` for the actual function declarations. The infrastructure will also invoke code in your pager when the pager starts, when a process is created or destroyed, and when the simulated MMU detects a fault.

3. Simulated MMU

The MMU being simulated in this project uses a single-level, fixed-size page table. A virtual address is composed of a virtual page number and a page offset:



The page table used by this MMU is an array of page table entries (PTEs), one PTE per virtual page in the arena. The MMU locates the page table through the page table base register (PTBR). The PTBR is a variable that is declared and defined by the infrastructure (but will be controlled by your pager). The following portion of `vm_pager.h` describes the arena, PTE, page table, and PTBR.

```
/*
 * ***********************
 * * Definition of arena *
 * ***********************
 */

/* pagesize for the machine */
#define VM_PAGESIZE 4096

/* virtual address at which application's arena starts */
#define VM_ARENA_BASEADDR    ((void *) 0x60000000)

/* virtual page number at which application's arena starts */
#define VM_ARENA_BASEPAGE    ((uintptr_t) VM_ARENA_BASEADDR / VM_PAGESIZE)

/* size (in bytes) of arena */
#define VM_ARENA_SIZE    0x20000000

/*
 * ************************************
 * * 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)
 */
typedef struct {
    unsigned int ppage : 20;            /* bit 0-19 */
    unsigned int read_enable : 1;       /* bit 20 */
    unsigned int write_enable : 1;      /* bit 21 */
} page_table_entry_t;

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

/*
 * 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. Interface used by applications of the external pager

Applications use three system calls to communicate explicitly with the simulated operating system: `vm_extend`, `vm_syslog`, and `vm_yield`. The prototypes for these system calls are given in the file [vm_app.h](vm_app.h).

The arena of a process is the range of addresses [`VM_ARENA_BASEADDR , VM_ARENA_BASEADDR + VM_ARENA_SIZE`).

An application calls `vm_extend` to ask for the lowest invalid page in its arena to be declared valid. `vm_extend` returns the lowest-numbered byte of the new valid page. E.g., if the arena before calling `vm_extend` is `0x60000000-0x60003fff`, the return value of the next `vm_extend` call will be `0x60004000`, and the resulting valid part of the arena will be `0x60000000-0x60004fff`. Each byte of a newly extended virtual page is defined to be initialized with the value 0. FYI, the `vm_extend` interface is similar to the `sbrk` call provided by Linux. The interfaces you normally use to to manage dynamic memory (`new`, `delete`, `malloc`, and `free`) are user-level libraries built on top of `sbrk`.

A virtual page can be **shared** with other virtual pages. All members within a set of shared virtual pages refer to the same data. I.e., stores to one virtual page are seen by loads to all virtual pages that are shared with that virtual page. From the point of view of the applications (and system), shared virtual pages are simply aliases for each other. One or more processes create a set of shared virtual pages by calling `vm_extend` with the same non-zero `share_id`. The data in a shared virtual page persists until all processes that share this page have been destroyed.

An application calls `vm_syslog` to ask the pager to print a message (all message data should be in the valid part of the arena). `vm_syslog` returns 0 on success and -1 on failure.

An application calls `vm_yield` to tell 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.

In addition to these explicit system calls, applications may also communicate implicitly with the pager by loading or storing to an address in the arena. Depending on the protections and residencies set by the pager, some of these loads and stores will result in calls to the pager's `vm_fault` function. These faults are serviced without the application's knowledge.

Finally, applications communicate implicitly with the pager when they are created by the `fork` system call. The pager's `vm_create` function will be called when the process is created. The infrastructure will also make calls to `vm_extend`, `vm_switch`, and `vm_fault` to copy the valid portion of the parent's arena to the child.

Here is an example application program that uses the external pager.

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

using namespace std;

int main()
{
    char *p;
    p = (char *) vm_extend(0);
    p[0] = 'h';
    p[1] = 'e';
    p[2] = 'l';
    p[3] = 'l';
    p[4] = 'o';
    vm_syslog(p, 5);
}
```

---

5. Pager specification

This section describes the functions you will implement in your external pager: `vm_init`, `vm_create`, `vm_fault`, `vm_destroy`, `vm_extend`, and `vm_syslog`. Note that you will not implement `main`; instead, `main` is included in

libvm_pager.a. The infrastructure will invoke your pager functions as described below.

The following portion of vm_pager.h describes the functions you will implement in your pager:

```
/*
 * vm_init
 *
 * Called when the pager starts.  It should set up any internal data structures
 * needed by the pager, e.g., physical page bookkeeping, process table, disk
 * usage table, etc.
 *
 * vm_init is passed both the number of physical memory pages and the number
 * of disk blocks in the raw disk.
 */
extern void vm_init(unsigned int memory_pages, unsigned int disk_blocks);

/*
 * vm_create
 * Called when a new process, with process identifier "pid", is added to the
 * system.  It should create whatever new elements are required for each of
 * your data structures.  The new process will only run when it's switched
 * to via vm_switch().
 */
extern void vm_create(pid_t pid);

/*
 * vm_switch
 *
 * Called when the kernel is switching to a new process, with process
 * identifier "pid".  This allows the pager to do any bookkeeping needed to
 * register the new process.
 */
extern void vm_switch(pid_t pid);

/*
 * vm_fault
 *
 * Called when current process has a fault at virtual address addr.  write_flag
 * is true if the access that caused the fault is a write.
 * Returns 0 on success, -1 on failure.
 */
extern int vm_fault(void *addr, bool write_flag);

/*
 * vm_destroy
 *
 * Called when current process exits.  It should deallocate all resources
 * held by the current process (page table, physical pages, disk blocks, etc.)
 */
extern void vm_destroy();

/*
 * vm_extend
 *
 * A request by the current process for the lowest invalid virtual page in the
 * process's arena to be declared valid.  If share_id is zero, this virtual
 * page is private, i.e., not shared with any other virtual page.  If share_id
 * is non-zero, this virtual page is shared with other virtual pages with the
 * same share_id.
 * Returns the lowest-numbered byte of the new virtual page.
 * Returns nullptr if disk is out of swap space.
```

```
 */
extern void *vm_extend(unsigned int share_id);

/*
 * vm_syslog
 *
 * A request by the current process to log a message.  The message is stored
 * in the process' arena at address "message" and is of length "len".
 *
 * Returns 0 on success, -1 on failure.
 */
extern int vm_syslog(void *message, size_t len);
```

5.1. `vm_init`

The infrastructure calls `vm_init` when the pager starts. Its parameters are the number of physical pages provided in physical memory and the number of disk blocks available on the disk. `vm_init` should set up whatever data structures you need to begin accepting `vm_create` and subsequent requests from processes.

5.2. `vm_create`

The infrastructure calls `vm_create` when a *parent* process starts a new *child* process via the `fork` system call. You should initialize whatever data structures you need to manage the new child process.

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`.

After a new child process is created, the infrastructure will subsequently make calls to `vm_extend`, `vm_switch`, and `vm_fault` to copy the valid portion of the parent's arena to the child.

5.3. `vm_switch`

The infrastructure calls `vm_switch` when the OS scheduler runs a new process. This allows your pager to do whatever bookkeeping it needs to register the fact that a new process is running.

5.4. `vm_extend`

`vm_extend` is called when a process wants to make another virtual page in its arena valid. `vm_extend` should return the lowest-numbered byte of the new valid virtual page. E.g., if the arena before calling `vm_extend` is `0x60000000-0x60003fff`, the return value of `vm_extend` will be `0x60004000`, and the resulting valid part of the arena will be `0x60000000-0x60004fff`.

`vm_extend` should ensure that there are enough available disk blocks to hold all valid virtual pages (this is called *eager* swap allocation). If there are no free disk blocks, `vm_extend` should return nullptr. The benefit of eager swap allocation is that applications know at the time of `vm_extend` that there is no more swap space, rather than when a page needs to be evicted to disk.

A non-zero `share_id` names the newly extended virtual page. All virtual pages with the same non-zero `share_id` 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.

Remember that an application should see each byte of a newly mapped virtual page as initialized with the value 0. However, the actual data initialization needed to provide this abstraction should be deferred as long as

possible.

5.5. `vm_fault`

The `vm_fault` function is called in response to a read or write fault by the application. Your pager determines which accesses in the arena will generate faults by setting the `read_enable` and `write_enable` fields in the page table. Your pager determines which physical page is associated with a virtual page by setting the ppage field in the page table.

`vm_fault` should return 0 after successfully handling a fault. `vm_fault` should return -1 if the address is to an invalid page or is outside the arena.

5.5.1. Non-resident pages

If a fault occurs on a virtual page that is not resident, you must find a physical page to associate with the virtual page. If there are no free physical pages, you must create a free physical page by evicting a virtual page that is currently resident.

Use the *FIFO with second-chance* (clock) algorithm to select a victim. The clock queue is an ordered list of all valid, resident virtual pages in the system. When a virtual page is made resident, that page should be placed on the tail of the clock queue (and marked as referenced). To select a victim, remove and examine the page at the head of the queue. If it has been accessed in any way since it was last placed on the queue, it should be added 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, then its virtual page should be evicted. Dirty and clean pages are treated the same when selecting a victim page to evict. Remember that

Hint: When you evict a page and later make it resident again, you can sometimes avoid writing it to disk and reading it back from disk.

Hint: The order of pages in the clock queue may differ from the order of their physical page numbers.

5.5.2. Resident pages

Your pager controls the page protections for resident pages. Its goal in controlling protections is to maintain any state it needs to defer work and implement the clock replacement algorithm (e.g., dirty and reference bits). An access to a resident page will generate a page fault if the page's protection does not allow the access. On these faults, `vm_fault` should update state as needed, change the protections on the virtual page, and continue.

5.6. `vm_syslog`

`vm_syslog` is called with a pointer to an array of bytes in the current process's virtual address space and the length of that array. Your pager should first check that the entire message is in valid pages of the arena. Return -1 (and don't print anything) if any part of the message is not on a valid arena page, or if length is zero.

After checking for invalid addresses, your pager should next copy the entire array into a C++ string in the pager's address space, then print the C++ string to `cout`. Use the following snippet of code for your print statement (this assumes the C++ string variable is named `s`):

```
cout << "syslog\t\t\t" << s << endl;
```

Most of the work in `vm_syslog` will be copying the array into the pager's C++ string. `vm_syslog` must handle virtual to physical address translation while copying the message from one address space to another. You should

treat `vm_syslog`'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_syslog` should copy the application's data starting at the lowest virtual address and proceeding toward the highest virtual address.

5.7. `vm_destroy`

`vm_destroy` is called by the infrastructure when the corresponding application exits. This function must deallocate all resources held by that process. This includes page tables, physical pages, and disk blocks. Physical pages that are released should be put back on the free list.

5.8. Deferring and avoiding work

There are many points in this project where you have some freedom over when zero-fills, faults, and disk I/O happen. You must defer such work as far into the future as possible.

Similarly, 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 5.5.1 must be used as specified; e.g., don't change the victim selection to avoid writing a page to disk).

Note that you will need to maintain reference and dirty bits to defer work and to implement the clock algorithm. Since the MMU for this project does not maintain dirty or reference bits, your pager will maintain these bits by generating page faults on appropriate accesses.

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 zero-filling 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. Interface used by external pager to access the simulated hardware

This section describes how your external pager will access the simulated hardware, i.e., physical memory, disk, and MMU.

The following portion of `vm_pager.h` describes the variables and utility functions for accessing this hardware.

```
/*
 * ******************************************
 * * Public interface for the disk abstraction *
 * ******************************************
 *
 * Disk blocks are numbered from 0 to (disk_blocks-1), where disk_blocks
 * is the parameter passed in vm_init().
 */

/*
 * disk_read
 *
 * read block "block" from the disk into buf.
 */
extern void disk_read(unsigned int block, void *buf);

/*
 * disk_write
```

```
 *
 * write the contents of buf to disk block "block".
 */
extern void disk_write(unsigned int block, void *buf);

/*
 * ******************************************************
 * * Public interface for the physical memory abstraction *
 * ******************************************************
 *
 * 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 N pages, numbered from 0 to N-1. It is settable through the -m option when you run the external pager (e.g., by running `pager -m 4`). The minimum number of physical pages is 2, the maximum is 1024, and the default is 4. Your pager can access the data in physical memory via the array `vm_physmem`.

The disk is modeled as a single device that has `disk_blocks` blocks. Each disk block is the same size as a physical memory page. Your pager will use two functions to access the disk: `disk_write` is used to write data from a physical page out to disk, and `disk_read` is used to read data from disk into a physical page.

Physical pages should only be shared among virtual pages when those virtual pages are shared with each other. That is, each physical page should be associated with at most one virtual page (or set of shared virtual pages) at any given time.

Similarly, disk blocks should only be shared between virtual pages when those virtual pages are shared with each other. That is, each disk block should be associated with at most one virtual page (or set of shared virtual pages) at any given time.

Your pager controls the operation of the MMU by modifying the contents of the page table and the variable `page_table_base_register`.

7. Hints

First, write down a finite state machine for the life of a virtual page, from creation via `vm_extend` to destruction via `vm_destroy`. Ask yourself what events can happen to a page at each stage of 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. A large portion of the credit in this project hinges on having this state machine correct.

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.

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

Virtual pages will never be write-only (`read_enable=0`, `write_enable=1`).

8. 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. Test cases should exit(0) when run with a correct pager.

Your test suite may contain up to 20 test cases. Each test case may cause a correct pager to generate at most 256 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.

Each test case will specify the number of physical memory pages to use when running the pager (the -m option) for the test case. This parameter will be communicated via the name of the test case file. The name of each test case should start with `test` and should be of the following format:

```
testArbitraryName.memoryPages.cc
```

where `memoryPages` identifies the number of physical memory pages to use with the pager, and the parts are separated by periods. Remember that the minimum number of physical memory pages is 2 and the maximum is 1024.

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 a few of the buggy pagers used to evaluate your test suite require multi-process applications to be exposed. Use `vm_yield` to coordinate the order in which processes run.

9. Project logistics

Write your pager in C++ on Linux. The public functions in `vm_pager.h` are declared `extern`, but all other functions and global variables in your pager should be declared `static` to prevent naming conflicts with other libraries.

Use `g++ 4.7.0` to compile your programs. To use `g++ 4.7.0` on CAEN computers, put the following command in your startup file (e.g., `~/.profile`):

```
module load gcc
```

You may use any functions included in the standard C++ library, including the STL. You should not use any libraries other than the standard C++ library. To compile a pager `pager.cc`, run:

```
g++ pager.cc libvm_pager.a -std=c++11
```

To compile an application `app.cc`, run:

```
g++ app.cc libvm_app.a -ldl -std=c++11
```

You may add options `-g` and `-Wall` for debugging and `-o` to name the executable.

Your pager code may be in multiple files. Each file name must end with `.cc`, `.cpp`, or `.h` and must not start with `test`.

Here's how to run your pager and an application. First 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. After the pager starts, you can run one or more application processes which 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.

We have created a private github repository for your group (`eecs482/<group>.3`), where `<group>` is the sorted, dot-separated list of your group members' uniqnames. Initialize your local repository by cloning the (empty) repository from github, e.g.,

```
git clone git@github.com:eecs482/uniqnameA.uniqnameB.3
```

10. 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.

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. However, only the feedback from the first submission of each day will be provided to you. Later submissions on that day will be graded and cataloged, but the results will not be provided to you. See the FAQ for why we use this policy.

In addition to this one-per-day policy, you will be given 3 bonus submissions that also provide feedback. These 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 1 feedback per day.

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

- The only output your pager code should print is that specified for `vm_syslog`. The pager infrastructure also prints messages to help you debug (and to allow the auto-grader to understand what the pager is doing); you can disable these messages by running the pager with the `-q` flag.
- 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 the clarity and completeness of your documentation, coding style, the efficiency, brevity, and understandability of your code, 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 pager documentation should give an overall picture of your solution, with enough detail that we can easily read and understand your code. You should present a list of all of the places in your solution that you deferred work; give both the event that you deferred, and the time at which you had to do the work. Your final score will be the product of the hand-graded score (between 1-1.12) and the auto-grader score.

11. Turning in the project

Submit the following files for your external 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.

12. Files included in this handout

- `libvm_app.a`
- `libvm_pager.a`
- `vm_app.h`
- `vm_pager.h`