

## Project - 2

### CSE 421/521 – Operating Systems

**Due: December 7<sup>th</sup> @11:59pm, 2014**

#### **1. Preparation**

Before beginning your work, please read the following carefully:

- Chapters 8-9 from Silberschatz
- Lecture slides on Memory and Virtual Memory
- [Pintos Introduction](#)
- [Pintos Reference Guide](#)
- [Complete Pintos Documentation \(PDF file\)](#) -- *for reference only*

#### **2. Task: Implement the Virtual Memory Component of Pintos OS**

In this project, you are asked to perform “kernel” level programming of the **Virtual Memory component in the Pintos operation system**. This project will help you to better understanding paging, page replacement, and other virtual memory concepts.

Before beginning this assignment, make sure you read these sections from Pintos Reference Guide: section [A.5 Memory Allocation](#), section [A.6 Virtual Addresses](#), and section [A.7 Page Table](#).

#### **3. Setting Up The Pintos Environment**

To get started, you'll have to log into a machine on which Pintos compiles. These include UB CSE public Linux machines (such as dragonforce, styx, and nickelback).

Copy the updated Pintos source (which includes a working userprog component that you will need for this project) from `/web/faculty/tkosar/cse421-521/projects/project-2/pintos2.tar` to your home directory.

(Alternatively you can fetch it from: (<http://www.cse.buffalo.edu/faculty/tkosar/cse421-521/projects/project-2/>))

Create a directory called “pintos” and extract the source code to this directory using:

```
tar -xvf pintos2.tar
```

#### **4. Implementation of the Project**

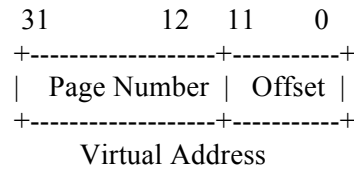
You will be working primarily in the “vm” directory of the source tree for this assignment, with some work in the “devices” directory on the side. Compilation should be done in the “vm” directory.

##### **4.1 Background**

Careful definitions are needed to keep discussion of virtual memory from being confusing. Thus, we begin by presenting some terminology for memory and storage (see section [3.1.4 Virtual Memory Layout](#)).

##### **4.1.1 Pages**

A *page*, sometimes called a *virtual page*, is a continuous region of virtual memory 4,096 bytes (the *page size*) in length. A page must be *page-aligned*, that is, start on a virtual address evenly divisible by the page size. Thus, a 32-bit virtual address can be divided into a 20-bit *page number* and a 12-bit *page offset* (or just *offset*), like this:

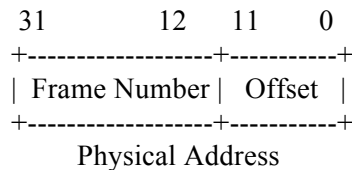


Each process has an independent set of *user (virtual) pages*, which are those pages below virtual address `PHYS_BASE`, typically `0xc0000000` (3 GB). The set of *kernel (virtual) pages*, on the other hand, is global, remaining the same regardless of what thread or process is active. The kernel may access both user and kernel pages, but a user process may access only its own user pages. See section [3.1.4 Virtual Memory Layout](#), for more information.

Pintos provides several useful functions for working with virtual addresses. See section [A.6 Virtual Addresses](#), for details.

#### 4.1.2 Frames

A *frame*, sometimes called a *physical frame* or a *page frame*, is a continuous region of physical memory. Like pages, frames must be page-size and page-aligned. Thus, a 32-bit physical address can be divided into a 20-bit *frame number* and a 12-bit *frame offset* (or just *offset*), like this:



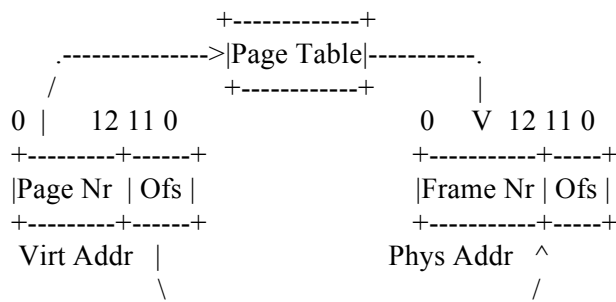
The 80x86 doesn't provide any way to directly access memory at a physical address. Pintos works around this by mapping kernel virtual memory directly to physical memory: the first page of kernel virtual memory is mapped to the first frame of physical memory, the second page to the second frame, and so on. Thus, frames can be accessed through kernel virtual memory.

Pintos provides functions for translating between physical addresses and kernel virtual addresses. See section [A.6 Virtual Addresses](#), for details.

#### 4.1.3 Page Tables

In Pintos, a *page table* is a data structure that the CPU uses to translate a virtual address to a physical address, that is, from a page to a frame. The page table format is dictated by the 80x86 architecture. Pintos provides page table management code in “`pagedir.c`” (see section [A.7 Page Table](#)).

The diagram below illustrates the relationship between pages and frames. The virtual address, on the left, consists of a page number and an offset. The page table translates the page number into a frame number, which is combined with the unmodified offset to obtain the physical address, on the right.



#### 4.1.4 Swap Slots

A *swap slot* is a continuous, page-size region of disk space in the swap partition. Although hardware limitations dictating the placement of slots are looser than for pages and frames, swap slots should be page-aligned because there is no downside in doing so.

#### 4.2 Source Files

You will work in the “vm” directory for this project. In the “vm” directory, we have created three template source files for you: “page.c”, “frame.c”, and “swap.c” as well as the corresponding header files. In these three source files, you will find empty function definitions that you will need to fill in.

#### 4.3 Resource Management Overview

You will need to design the following data structures:

- Supplemental page table: Enables page fault handling by supplementing the hardware page table. See section “4.4 Managing the Supplemental Page Table” below.
- Frame table: Allows efficient implementation of eviction policy. See section “4.5 Managing the Frame Table” below.
- Swap table: Tracks usage of swap slots. See section “4.6 Managing the Swap Table” below.
- Table of file mappings: Processes may map files into their virtual memory space. You need a table to track which files are mapped into which pages.

You do not necessarily need to implement four completely distinct data structures: it may be convenient to wholly or partially merge related resources into a unified data structure.

For each data structure, you need to determine what information each element should contain. You also need to decide on the data structure's scope, either local (per-process) or global (applying to the whole system), and how many instances are required within its scope.

To simplify your design, you may store these data structures in non-pageable memory. That means that you can be sure that pointers among them will remain valid.

Possible choices of data structures include arrays, lists, bitmaps, and hash tables. An array is often the simplest approach, but a sparsely populated array wastes memory. Lists are also simple, but traversing a long list to find a particular position wastes time. Both arrays and lists can be resized, but lists more efficiently support insertion and deletion in the middle.

Pintos includes a bitmap data structure in “lib/kernel/bitmap.c” and “lib/kernel/bitmap.h”. A bitmap is an array of bits, each of which can be true or false. Bitmaps are typically used to track usage in a set of (identical) resources: if resource  $n$  is in use, then bit  $n$  of the bitmap is true. Pintos bitmaps are fixed in size, although you could extend their implementation to support resizing.

Pintos also includes a hash table data structure (see section [A.8 Hash Table](#)). Pintos hash tables efficiently support insertions and deletions over a wide range of table sizes.

Although more complex data structures may yield performance or other benefits, they may also needlessly complicate your implementation. Thus, we do not recommend implementing any advanced data structure (e.g. a balanced binary tree) as part of your design.

## 4.4 Managing the Supplemental Page Table

The *supplemental page table* supplements the page table with additional data about each page. It is needed because of the limitations imposed by the page table's format. Such a data structure is often called a "page table" also; we add the word "supplemental" to reduce confusion.

The supplemental page table is used for at least two purposes. Most importantly, on a page fault, the kernel looks up the virtual page that faulted in the supplemental page table to find out what data should be there. Second, the kernel consults the supplemental page table when a process terminates, to decide what resources to free.

You may organize the supplemental page table as you wish. There are at least two basic approaches to its organization: in terms of segments or in terms of pages. Optionally, you may use the page table itself as an index to track the members of the supplemental page table. You will have to modify the Pintos page table implementation in "pagedir.c" to do so. We recommend this approach for advanced students only. See section [A.7.4.2 Page Table Entry Format](#), for more information.

The most important user of the supplemental page table is the page fault handler. In project 2, a page fault always indicated a bug in the kernel or a user program. In project 3, this is no longer true. Now, a page fault might only indicate that the page must be brought in from a file or swap. You will have to implement a more sophisticated page fault handler to handle these cases. Your page fault handler, which you should implement by modifying `page_fault()` in "userprog/exception.c", needs to do roughly the following:

1. Locate the page that faulted in the supplemental page table. If the memory reference is valid, use the supplemental page table entry to locate the data that goes in the page, which might be in the file system, or in a swap slot, or it might simply be an all-zero page. If you implement sharing, the page's data might even already be in a page frame, but not in the page table. If the supplemental page table indicates that the user process should not expect any data at the address it was trying to access, or if the page lies within kernel virtual memory, or if the access is an attempt to write to a read-only page, then the access is invalid. Any invalid access terminates the process and thereby frees all of its resources.
2. Obtain a frame to store the page. See section "4.5 Managing the Frame Table" below, for details. If you implement sharing, the data you need may already be in a frame, in which case you must be able to locate that frame.
3. Fetch the data into the frame, by reading it from the file system or swap, zeroing it, etc. If you implement sharing, the page you need may already be in a frame, in which case no action is necessary in this step.
4. Point the page table entry for the faulting virtual address to the physical page. You can use the functions in "userprog/pagedir.c".

## 4.5 Managing the Frame Table

The *frame table* contains one entry for each frame that contains a user page. Each entry in the frame table contains a pointer to the page, if any, that currently occupies it, and other data of your choice. The frame table allows Pintos to efficiently implement an eviction policy, by choosing a page to evict when no frames are free.

The frames used for user pages should be obtained from the "user pool," by calling `pallocc_get_page(PAL_USER)`. You must use `PAL_USER` to avoid allocating from the "kernel pool," which could cause some test cases to fail unexpectedly (see [Why PAL\\_USER?](#)). If you modify "pallocc.c" as part of your frame table implementation, be sure to retain the distinction between the two pools.

The most important operation on the frame table is obtaining an unused frame. This is easy when a frame is free. When none is free, a frame must be made free by evicting some page from its frame.

If no frame can be evicted without allocating a swap slot, but swap is full, panic the kernel. Real OSes apply a wide range of policies to recover from or prevent such situations, but these policies are beyond the scope of this project.

The process of eviction comprises roughly the following steps:

1. Choose a frame to evict, using your page replacement algorithm. The "accessed" and "dirty" bits in the page table, described below, will come in handy.
2. Remove references to the frame from any page table that refers to it. Unless you have implemented sharing, only a single page should refer to a frame at any given time.
3. If necessary, write the page to the file system or to swap.

The evicted frame may then be used to store a different page.

#### **4.5.1 Accessed and Dirty Bits**

80x86 hardware provides some assistance for implementing page replacement algorithms, through a pair of bits in the page table entry (PTE) for each page. On any read or write to a page, the CPU sets the *accessed bit* to 1 in the page's PTE, and on any write, the CPU sets the *dirty bit* to 1. The CPU never resets these bits to 0, but the OS may do so.

You need to be aware of *aliases*, that is, two (or more) pages that refer to the same frame. When an aliased frame is accessed, the accessed and dirty bits are updated in only one page table entry (the one for the page used for access). The accessed and dirty bits for the other aliases are not updated.

In Pintos, every user virtual page is aliased to its kernel virtual page. You must manage these aliases somehow. For example, your code could check and update the accessed and dirty bits for both addresses. Alternatively, the kernel could avoid the problem by only accessing user data through the user virtual address.

See section [A.7.3 Accessed and Dirty Bits](#), for details of the functions to work with accessed and dirty bits.

#### **4.6 Managing the Swap Table**

The swap table tracks in-use and free swap slots. It should allow picking an unused swap slot for evicting a page from its frame to the swap partition. It should allow freeing a swap slot when its page is read back or the process whose page was swapped is terminated.

You may use the BLOCK\_SWAP block device for swapping, obtaining the struct block that represents it by calling `block_get_role()`. From the "vm/build" directory, use the command `pintos-mkdisk swap.dsk --swap-size=n` to create a disk named "swap.dsk" that contains a *n*-MB swap partition. Afterward, "swap.dsk" will automatically be attached as an extra disk when you run `pintos`. Alternatively, you can tell `pintos` to use a temporary *n*-MB swap disk for a single run with "`--swap-size=n`".

Swap slots should be allocated lazily, that is, only when they are actually required by eviction. Reading data pages from the executable and writing them to swap immediately at process startup is not lazy. Swap slots should not be reserved to store particular pages.

Free a swap slot when its contents are read back into a frame.

## 4.7 Managing Memory Mapped Files

The file system is most commonly accessed with read and write system calls. A secondary interface is to "map" the file into virtual pages, using the mmap system call. The program can then use memory instructions directly on the file data.

Suppose file "foo" is 0x1000 bytes (4 kB, or one page) long. If "foo" is mapped into memory starting at address 0x5000, then any memory accesses to locations 0x5000...0x5fff will access the corresponding bytes of "foo".

Here's a program that uses mmap to print a file to the console. It opens the file specified on the command line, maps it at virtual address 0x10000000, writes the mapped data to the console (fd 1), and unmaps the file.

```
#include <stdio.h>
#include <syscall.h>
int main (int argc UNUSED, char *argv[])
{
    void *data = (void *) 0x10000000;    /* Address at which to map. */

    int fd = open (argv[1]);             /* Open file. */
    mapid_t map = mmap (fd, data);       /* Map file. */
    write (1, data, filesize (fd));      /* Write file to console. */
    munmap (map);                        /* Unmap file (optional). */
    return 0;
}
```

A similar program with full error handling is included as "mcat.c" in the "examples" directory, which also contains "mcp.c" as a second example of mmap.

Your submission must be able to track what memory is used by memory mapped files. This is necessary to properly handle page faults in the mapped regions and to ensure that mapped files do not overlap any other segments within the process.

## 5. Requirements

This assignment is an open-ended design problem. We are going to say as little as possible about how to do things. Instead we will focus on what functionality we require your OS to support. We will expect you to come up with a design that makes sense. You will have the freedom to choose how to handle page faults, how to organize the swap partition, how to implement paging, etc.

### 5.1 Paging

Implement paging for segments loaded from executables. All of these pages should be loaded lazily, that is, only as the kernel intercepts page faults for them. Upon eviction, pages modified since load (e.g. as indicated by the "dirty bit") should be written to swap. Unmodified pages, including read-only pages, should never be written to swap because they can always be read back from the executable.

Implement a global page replacement algorithm that approximates LRU. Your algorithm should perform at least as well as the simple variant of the "second chance" or "clock" algorithm.

Your design should allow for parallelism. If one page fault requires I/O, in the meantime processes that do not fault should continue executing and other page faults that do not require I/O should be able to complete. This will require some synchronization effort.

You'll need to modify the core of the program loader, which is the loop in `load_segment()` in "userprog/process.c". Each time around the loop, `page_read_bytes` receives the number of bytes to read from the executable file and `page_zero_bytes` receives the number of bytes to initialize to zero following the bytes read. The two always sum to `PGSIZE` (4,096). The handling of a page depends on these variables' values:

- If `page_read_bytes` equals `PGSIZE`, the page should be demand paged from the underlying file on its first access.
- If `page_zero_bytes` equals `PGSIZE`, the page does not need to be read from disk at all because it is all zeroes. You should handle such pages by creating a new page consisting of all zeroes at the first page fault.
- Otherwise, neither `page_read_bytes` nor `page_zero_bytes` equals `PGSIZE`. In this case, an initial part of the page is to be read from the underlying file and the remainder zeroed.

## 5.2 Stack Growth

Implement stack growth. If the stack grows past its current size, allocate additional pages as necessary.

Allocate additional pages only if they "appear" to be stack accesses. Devise a heuristic that attempts to distinguish stack accesses from other accesses.

User programs are buggy if they write to the stack below the stack pointer, because typical real OSes may interrupt a process at any time to deliver a "signal," which pushes data on the stack. However, the 80x86 `PUSH` instruction checks access permissions before it adjusts the stack pointer, so it may cause a page fault 4 bytes below the stack pointer. (Otherwise, `PUSH` would not be restartable in a straightforward fashion.) Similarly, the `PUSHA` instruction pushes 32 bytes at once, so it can fault 32 bytes below the stack pointer.

You will need to be able to obtain the current value of the user program's stack pointer. Within a system call or a page fault generated by a user program, you can retrieve it from the `esp` member of the struct `intr_frame` passed to `syscall_handler()` or `page_fault()`, respectively. If you verify user pointers before accessing them (see section [3.1.5 Accessing User Memory](#)), these are the only cases you need to handle. On the other hand, if you depend on page faults to detect invalid memory access, you will need to handle another case, where a page fault occurs in the kernel. Since the processor only saves the stack pointer when an exception causes a switch from user to kernel mode, reading `esp` out of the struct `intr_frame` passed to `page_fault()` would yield an undefined value, not the user stack pointer. You will need to arrange another way, such as saving `esp` into struct `thread` on the initial transition from user to kernel mode.

You should impose some absolute limit on stack size, as do most OSes. Some OSes make the limit user-adjustable, e.g. with the `ulimit` command on many Unix systems. On many GNU/Linux systems, the default limit is 8 MB.

The first stack page need not be allocated lazily. You can allocate and initialize it with the command line arguments at load time, with no need to wait for it to be faulted in.

All stack pages should be candidates for eviction. An evicted stack page should be written to swap.

## 5.3 Memory Mapped Files

Implement memory mapped files, including the following system calls.

System Call: `mapid_t mmap (int fd, void *addr)`

Maps the file open as `fd` into the process's virtual address space. The entire file is mapped into



consecutive virtual pages starting at *addr*.

Your VM system must lazily load pages in mmap regions and use the mmaped file itself as backing store for the mapping. That is, evicting a page mapped by mmap writes it back to the file it was mapped from.

If the file's length is not a multiple of PGSIZE, then some bytes in the final mapped page "stick out" beyond the end of the file. Set these bytes to zero when the page is faulted in from the file system, and discard them when the page is written back to disk.

If successful, this function returns a "mapping ID" that uniquely identifies the mapping within the process. On failure, it must return -1, which otherwise should not be a valid mapping id, and the process's mappings must be unchanged.

A call to mmap may fail if the file open as *fd* has a length of zero bytes. It must fail if *addr* is not page-aligned or if the range of pages mapped overlaps any existing set of mapped pages, including the stack or pages mapped at executable load time. It must also fail if *addr* is 0, because some Pintos code assumes virtual page 0 is not mapped. Finally, file descriptors 0 and 1, representing console input and output, are not mappable.

System Call: void **munmap** (mapid\_t *mapping*)

Unmaps the mapping designated by *mapping*, which must be a mapping ID returned by a previous call to mmap by the same process that has not yet been unmapped.

All mappings are implicitly unmapped when a process exits, whether via exit or by any other means. When a mapping is unmapped, whether implicitly or explicitly, all pages written to by the process are written back to the file, and pages not written must not be. The pages are then removed from the process's list of virtual pages.

Closing or removing a file does not unmap any of its mappings. Once created, a mapping is valid until munmap is called or the process exits, following the Unix convention. See [Removing an Open File](#), for more information. You should use the file\_reopen function to obtain a separate and independent reference to the file for each of its mappings.

If two or more processes map the same file, there is no requirement that they see consistent data. Unix handles this by making the two mappings share the same physical page, but the mmap system call also has an argument allowing the client to specify whether the page is shared or private (i.e. copy-on-write).

## 5.4 Accessing User Memory

You will need to adapt your code to access user memory (see section [3.1.5 Accessing User Memory](#)) while handling a system call. Just as user processes may access pages whose content is currently in a file or in swap space, so can they pass addresses that refer to such non-resident pages to system calls. Moreover, unless your kernel takes measures to prevent this, a page may be evicted from its frame even while it is being accessed by kernel code. If kernel code accesses such non-resident user pages, a page fault will result.

While accessing user memory, your kernel must either be prepared to handle such page faults, or it must prevent them from occurring. The kernel must prevent such page faults while it is holding resources it would need to acquire to handle these faults. In Pintos, such resources include locks acquired by the device driver(s) that control the device(s) containing the file system and swap space. As a concrete example, you must not allow page faults to occur while a device driver accesses a user buffer passed to file\_read, because you would not be able to invoke the driver while handling such faults.

Preventing such page faults requires cooperation between the code within which the access



occurs and your page eviction code. For instance, you could extend your frame table to record when a page contained in a frame must not be evicted. (This is also referred to as "pinning" or "locking" the page in its frame.) Pinning restricts your page replacement algorithm's choices when looking for pages to evict, so be sure to pin pages no longer than necessary, and avoid pinning pages when it is not necessary.

## **6. Suggested Order of Implementation**

We suggest the following initial order of implementation:

1. Frame table (see section “4.5 Managing the Frame Table” above). Change “process.c” to use your frame table allocator. Do not implement swapping yet. If you run out of frames, fail the allocator or panic the kernel. After this step, your kernel should still pass all the project 2 test cases.
2. Supplemental page table and page fault handler (see section “4.4 Managing the Supplemental Page Table” above). Change “process.c” to record the necessary information in the supplemental page table when loading an executable and setting up its stack. Implement loading of code and data segments in the page fault handler. For now, consider only valid accesses. After this step, your kernel should pass all of the project 2 functionality test cases, but only some of the robustness tests.

From here, you can implement stack growth, mapped files, and page reclamation on process exit in parallel.

The next step is to implement eviction (see section “4.5 Managing the Frame Table” above). Initially you could choose the page to evict randomly. At this point, you need to consider how to manage accessed and dirty bits and aliasing of user and kernel pages. Synchronization is also a concern: how do you deal with it if process A faults on a page whose frame process B is in the process of evicting? Finally, implement a eviction strategy such as the clock algorithm.

## **7. Testing**

Your project grade will be based on our tests. Each project has several tests, each of which has a name beginning with “tests”. To completely test your submission, invoke “make check” from the project “src/vm/build” directory. This will build and run each test and print a “pass” or “fail” message for each one. When a test fails, make check also prints some details of the reason for failure. After running all the tests, make check also prints a summary of the test results. With the base code we provide you (pintos2.tar), 8 out of the 34 tests should already pass.

You can also run individual tests one at a time. A given test *t* writes its output to “*t.output*”, then a script scores the output as “pass” or “fail” and writes the verdict to “*t.result*”. To run and grade a single test, make the “.result” file explicitly from the “build” directory, e.g. make tests/vm/pt-write-code.result. If make says that the test result is up-to-date, but you want to re-run it anyway, either run make clean or delete the “.output” file by hand.

By default, each test provides feedback only at completion, not during its run. If you prefer, you can observe the progress of each test by specifying “VERBOSE=1” on the make command line, as in make check VERBOSE=1. You can also provide arbitrary options to the pintos run by the tests with “PINTOSOPTS=’...’”, e.g. make check PINTOSOPTS=’-j 1’ to select a jitter value of 1 (see section [1.1.4 Debugging versus Testing](#)).

All of the tests and related files are in “pintos/src/tests”.

## **8. Design Document**

Before you turn in your project, you must copy the [Project 3 Design Document Template \(vm.tmpl\)](#) into your source tree under the name “pintos/src/threads/DESIGNDOC” and fill it in.

We recommend that you read the design document template before you start working on the project. See section [D. Project Documentation](#), for a sample design document that goes along with a fictitious project.

## **7. Grading**

The grading of the project will be done according to the following rubric (83 points for the implementation + 17 points for the documentation = TOTAL 100 points):

- (12 points) A completely functional stack growth implementation that passes all four(4) tests.
- (25 points) A fully functional paging implementation that passes all seven (7) tests.
- (18 points) A working mmap system call implementation that passes all nine (9) tests.
- (14 points) Robustness of page table support that passes all five (5) tests.
- (14 points) Robustness of mmap system call that pass all nine (9) tests.
- (12 points) A complete design document.
- (5 points) A well-documented and clean source code.

You can use “make grade” to test your implementation and to see what grade you would get from the implementation component.

## **9. What to Submit?**

1. You need to submit the complete **source tree** (all source files) of your project.
2. The package should also include a **README** file. The whole package should compile when the tester simply types make in the source code directory. Your README file should contain details and options on how to compile and run your code, if there are any.
3. You also need to write a **design document** for your project as described in section 6. A softcopy of this design document should be submitted as part of the package.

All of the above need to be submitted by **December 7<sup>th</sup> @11:59pm**. You should crate a tar package of your source tree, and submit it using your “submit\_cse421” or “submit\_cse521” script.