---- GROUP ----

>> Fill in the names and email addresses of your group members.

Maria Gotcheva maria.gotcheva@mail.utoronto.ca
Jacob Matias jacob.matias@mail.utoronto.ca
Jerry Han jerrym.han@mail.utoronto.ca

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the
>> TAs, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while
>> preparing your submission, other than the Pintos documentation, course
>> text, lecture notes, and course staff.

PAGE TABLE MANAGEMENT

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

struct sup_page_entry {

void *addr;
Pointer to the virtual address (in user space) that this page starts at.

bool is_writable;
If true, the page data can be modified. Otherwise, the page is read-only.

struct file *file;
Pointer to the file that this page reads from, when paging in.

off_t offset;
The offset that this page starts to read at in struct file *file, when paging in.

uint32_t read_bytes;
Number of bytes that should be read from struct file *file, when paging in.

uint32_t zero_bytes;

Number of bytes that should be zeroed out, when paging in.

int swap_idx;
Index of this page in the swap table bitmap.

bool is_pinned;
If true, this page is currently being pinned. Otherwise, the page is not pinned.

bool is_exec;
If true, this page contains data related to the program executable.

struct hash_elem sup_hash_elem;
Used to interact with default hash implementation.

}
A sup_page_entry struct that provides additional information about a page of data.

struct thread
{
…

+ struct hash sup_page_table;
New member of struct thread that stores supplementary page table entries.

+ void *esp;
New member of struct thread that references esp value from most recent transition from user mode to kernel mode.

}

---- ALGORITHMS ----

>> A2: In a few paragraphs, describe your code for accessing the data
>> stored in the SPT about a given page.

In our code, we implemented the SPT as a hash table. To access the data stored in the SPT about a given page, the system first locates the corresponding ‘sup_page_entry’ using the virtual address through a hash lookup function. In our code, the ‘get_sup_page_entry’ function performs this task and returns the entry if it exists. Once the valid entry is found, all the fields of that ‘sup_page_entry’ can be accessed to retrieve detailed information about the page, which is crucial for handling page faults.

>> A3: How does your code coordinate accessed and dirty bits between
>> kernel and user virtual addresses that alias a single frame, or

>> alternatively how do you avoid the issue?

To avoid this issue, we have all our physical frames allocated in the user pool by calling palloc_get_page(USER_PAL), which allocates in the user_pool only. This approach ensures that kernel and user processes do not share the same physical frames, thereby eliminating the need to synchronize the accessed and dirty bits across different mappings. It leverages separate address spaces and avoids aliasing.

---- SYNCHRONIZATION ----

>> A4: When two user processes both need a new frame at the same time,
>> how are races avoided?

When a process allocates a new frame using the frame_alloc function, it uses a lock (frame_table_lock) to ensure mutual exclusion when accessing and modifying the frame table, thereby preventing race conditions when multiple threads attempt to allocate frames concurrently. By acquiring the lock at the beginning of the function and releasing it before returning, the function ensures that only one thread can manipulate the frame table at a time.

---- RATIONALE ----

>> A5: Why did you choose the data structure(s) that you did for
>> representing virtual-to-physical mappings?

The choice of using a hash table for SPT and a list for the Frame Table is driven by the need for efficient lookup, dynamic management and simplicity. The hash table ensures quick access to page entries based on virtual addresses, which is essential for handling page faults and managing virtual memory efficiently. The list in the Frame Table allows easy traversal and dynamic size management, which is suitable for operations like frame allocation and eviction. In particular, it reduced performance overhead in the implementation of the clock algorithm for page eviction.

PAGING TO AND FROM DISK

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

struct list frame_table;
A global list containing frame_table_entry elements for frames that are occupying space in physical memory.

struct lock frame_table_lock;
A lock used for synchronizing frame_table operations.

struct frame_table_entry {

void *frame;
Pointer to the start address of the physical frame.

struct thread *owner;
Pointer to the thread that is currently using this frame.

struct sup_page_entry *spe;
Pointer to the sup_page_entry whose page is being mapped to this frame.

struct list_elem frame_elem;
Used to interact with default list implementation.

}
A frame_table_entry struct that provides additional information about a physical frame.

static struct bitmap *swap_table;
A bitmap where each bit represents a single page of memory in the swap partition. Bit value of 1 indicates that the page (8 block sectors) is used.

static struct lock swap_table_lock;
A lock for handling synchronization during operations on swap table.

static struct block *block_device;
A block struct that references the block device for BLOCK_SWAP role.

#define SECTORS_PER_PAGE (PGSIZE / BLOCK_SECTOR_SIZE)
Number of sectors per page computed based on loaded PGSIZE and BLOCK_SECTOR_SIZE values.

---- ALGORITHMS ----

>> B2: When a frame is required but none is free, some frame must be
>> evicted. Describe your code for choosing a frame to evict.

When there are no frames free, the frame_evict function uses the clock algorithm to choose a frame to evict. The function iterates through the frames in the frame table, starting at the position specified by the global variable clock_hand which represents the current position of the clock hand in the frame table. During each iteration, the code checks if the page associated with

the current frame has been accessed recently by looking at the accessed bit. If the page has been accessed, the accessed bit is set to false and the clock hand moves to the next frame. If the page had not been accessed recently and is not pinned (determined by the is_pinned field in the supplementary page table entry associated with the frame), the function evicts the frame by calling frame_page_out. This approach of choosing a frame to evict skips evicting pinned pages and prioritizes evicting pages that had not been recently accessed.

>> B3: When a process P obtains a frame that was previously used by a
>> process Q, how do you adjust the page table (and any other data
>> structures) to reflect the frame Q no longer has?

Since our frame_table is configured in a way in which threads can add or remove entries to the frame table, there are 2 scenarios where a physical frame (in particular the same physical frame address) will be reused.

Scenario 1: Q evicts frame f, then P occupies that same frame f.

- When P calls sup_page_load(*spe), the function begins by retrieving a frame to load in the page corresponding to sup_page_entry *spe. If palloc_get_page() fails when attempting to allocate memory for a new frame, the page_eviction algorithm is invoked, evicting an existing frame_table_entry (fte) element, then paging out fte->frame to the file system or swap partition. When the paging out process is completed, the mapping between fte and its previous fte->spe element will have been removed, and the fte struct updates its fte->owner and fte->spe (based on the caller that invoked the page eviction) before being passed back to sup_page_load. If successful, the sup_page_load function can then utilize this fte (which is already updated to show that process P owns it and that its being mapped to spe), and if loading is successful maps fte->frame to spe->addr in P's page directory.

Scenario 2: Q exits and cleans up its sup_page_entry and corresponding frame_table_entry elements, freeing memory in the user pool that P later uses to allocate more frames.

- In this case, when P calls sup_page_load(*spe) and tries to retrieve a frame to load in the page corresponding to spe, the palloc_get_page() call will succeeds and a new fte struct (along with fte->frame) is allocated, and is returned to sup_page_load to finish the page loading process.

>> B4: Explain your heuristic for deciding whether a page fault for an
>> invalid virtual address should cause the stack to be extended into
>> the page that faulted.

To determine whether the faulted address is a valid attempt to utilize the user stack, the page_fault handler checks that the following two conditions are satisfied:

1. fault_addr exists within the user stack (fault_addr >= PHYS_BASE - MAX_STACK_SIZE).

2. fault_addr is no more than 32 bytes below the user's stack pointer (fault_addr >= u_esp - 32).

To elaborate on condition 2, PUSHA is the largest push instruction that we need to account for and a single PUSHA instruction moves the user stack pointer down 32 bytes. Since these instructions need to be performed before adjusting esp, its is likely that a fault_addr more than 32 resulted from the user attempting to perform an operation on data they should not have access to.

---- SYNCHRONIZATION ----

>> B5: Explain the basics of your VM synchronization design. In
>> particular, explain how it prevents deadlock. (Refer to the
>> textbook for an explanation of the necessary conditions for
>> deadlock.)

Our implementation's VM synchronization consists of 3 main components, the frame_table_lock, the swap_table_lock, and the pinning functionality.

The frame_table_lock is acquired by a process when they attempt to load a page corresponding to some sup_page_entry element. The process holds this lock until the loading procedure is completed, which includes the swapping procedure and eviction procedure (if necessary). This lock prevents race conditions that arise when two different processes want to access or modify a frame in the frame_table. For example, processA wants to load data into a frame, whilst processB is in the middle of accessing or modifying the frame/fte struct that corresponds to the frame. In addition to this, the swap_table_lock is a lock that is used when updating the bits in the swap_table bitmap and is exclusively utilized by the swap_out and swap_in functions defined in swap.c.

The pinning functionality is implemented by referencing a sup_page_entry element's is_pinned field and ensures that the page corresponding to the sup_page_entry is only evicted once it fully finishes critical operations. For example, the is_pinning field is updated to true during the loading procedure of a page and only gets updated to false once that procedure is completed. As a result, any other process attempting to evict the frame that is currently being occupied by the pinned sup_page_entry (which corresponds to a pinned page) will instead evict non-pinned pages.

This design prevents deadlock because it prevents circular waiting. The frame_table_lock is the major synchronization component that prevents processes from simultaneously operating on the frame_table's resources. The only other synchronization lock that we utilize is the swap_table_lock, whenever swap operations are invoked. Since the swap operations are ONLY used within code segments that already assume frame_table_lock is held, we don't need to worry about scenarios where processA is waiting on a frame_table_lock held by processB, and processB is waiting on a swap_table_lock held by processA (or vice versa).

>> B6: A page fault in process P can cause another process Q's frame
>> to be evicted. How do you ensure that Q cannot access or modify
>> the page during the eviction process? How do you avoid a race
>> between P evicting Q's frame and Q faulting the page back in?

In order to access or modify a page that is currently being evicted, the process needs to acquire the frame_table_lock. However, when the page_eviction operation is being done, the frame_table_lock is held by P until the eviction and page_out process is finished. Only after the eviction and page_out process is completed is the frame_table_lock released and available for Q to acquire, to then modify or access its page as desired. It's important to note that in this case, if Q were to try to access or modify the page, Q would page fault since the mapping to that frame is cleared during the eviction procedure performed by P.

For similar reasons, the frame_table_lock avoids the race between P evicting Q's frame and Q faulting the page back in. If P evicts Q's frame, then Q won't be able to fault the page back in until it can acquire frame_table_lock which is currently being held by P until it finishes eviction.

>> B7: Suppose a page fault in process P causes a page to be read from
>> the file system or swap. How do you ensure that a second process Q
>> cannot interfere by e.g. attempting to evict the frame while it is
>> still being read in?

The eviction procedure and loading procedure handle synchronization through the pinning functionality. In this case, when P page faults and tries to load in a page corresponding to some sup_page_entry (spe), it invokes sup_page_load(*spe). This function begins by updating the is_pinned field of spe to true, and only sets the is_pinned field to false once spe's page has been loaded into physical memory. Thus, since our page_eviction algorithm ignores pinned pages as possible page-eviction candidates, this would ensure that Q invoking page_eviction does NOT interfere with the loading of P's spe (at least until spe has finished the loading procedure).

>> B8: Explain how you handle access to paged-out pages that occur
>> during system calls. Do you use page faults to bring in pages (as
>> in user programs), or do you have a mechanism for "locking" frames
>> into physical memory, or do you use some other design? How do you
>> gracefully handle attempted accesses to invalid virtual addresses?

The kernel syscall handler utilizes two different validation functions: validate_addr() and validate_buffer(). When reading the syscall arguments, many of the stack pointer addresses are validated using validate_addr which ensures that they are not NULL, exist within user address space, and are mapped to physical memory. In addition to this, the pointer value for buffer arguments (like those passed in by the read syscall) are validated using validate_buffer() which only checks that the buffer pointer value is not null and exists within user space (no mapping

requirement). As a result, when the read syscall is actually handled, and the buffer attempts to extend the stack, the page fault occurs within kernel context. Since we update the struct thread->esp field at the start (each time) of when the syscall interrupt handler gets invoked, we are then able to reference that esp value when checking to see if the faulted address passes our stack growth heuristic, and can successfully extend the user stack within kernel-context if necessary.

---- RATIONALE ----

>> B9: A single lock for the whole VM system would make
>> synchronization easy, but limit parallelism. On the other hand,
>> using many locks complicates synchronization and raises the
>> possibility for deadlock but allows for high parallelism. Explain
>> where your design falls along this continuum and why you chose to
>> design it this way.

When considering possible designs for the synchronization component of our VM system, we came up with two approaches. The first approach was to have the synchronization of the entire frame_table (all frames) via single frame_table_lock. The second approach was to extend our frame_table_entry struct to have a new struct lock frame_lock field that could be used to handle synchronization when operating on individual frames (as opposed to operations on the entire frame_table).

Ideally, we would’ve liked to integrate a medium of the two approaches, because the first approach provides a simplistic and maintainable design but has decreased performance due to its parallelism limitations, and the second approach drastically improves parallelism, at the expense increasing code complexity, increasing memory overhead and possibly increasing cpu overhead in order to perform the lock operations. However, we decided to implement the first approach because we were more concerned with ensuring simplistic design and maintainability in our code, especially since we anticipated that this project would utilize a lot of interconnected data structures and components. And secondly, we weren’t entirely convinced that the parallelism provided by approach two would offset the cpu overhead that would come with managing and performing all the lock operations for each frame’s lock.

MEMORY MAPPED FILES

---- DATA STRUCTURES ----

>> C1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

/* provides information about a memory mapped file */

struct file_mapping
{
    mapid_t id;
    The id value for this file mapping.

    struct file *file;
    Pointer to the file that is being mapped.

    void *addr;
    Pointer to the start address of the first user page that the file maps to.

    int page_cnt;
    Number of pages being mapped.

    struct list_elem file_mapping_elem;
    Used to interact with default list implementation.
}

struct thread
{
    …

    + struct list file_mappings;
    list of memory mapped files

    + int next_mapid;
    next available memory mapping id
}

---- ALGORITHMS ----

>> C2: Describe how memory mapped files integrate into your virtual
>> memory subsystem. Explain how the page fault and eviction
>> processes differ between swap pages and other pages.

Memory-mapped files integrate into the Pintos virtual memory subsystem through the use of SPTs, which track the metadata required for managing these mappings. The ‘handle_sys_mmap’ function sets up these mappings by creating ‘sup_page_entry’ structures for each page of the file and storing them in the SPT.

When a page fault occurs for a swap page, the system retrieves the page from the swap space, loads it into a newly allocated frame, and updates the page table and supplemental page table

accordingly. Conversely, for other pages, the system reads the required page from the file into a new frame using the file offset specified in the SPT entry, zero out any remaining bytes and updates the page tables. During evictions, the swap pages are written to swap space, and their swap index is updated in the supplemental page table. For other pages, if the SPT entry contains a file and it is dirty, it is written back to the file at the specified offset, and does nothing to files that are not dirty.

>> C3: Explain how you determine whether a new file mapping overlaps
>> any existing segment.

When a mmap syscall is being handled, we make a call to file_size(file) in order to compute the number of pages being mapped. Then, starting from the start_address that is passed in as an argument to the syscall and ending at address (start_address + page_count * PGSIZE), we check whether any of the pages' addresses already map to some frame in the thread's page directory. If this is true for at least one of the addresses we would like to map to, the mmap handler returns -1 to prevent overwriting an existing segment (since file mappings are mapped in consecutive pages). Otherwise, the new file mapping is created and its id is returned.

---- RATIONALE ----

>> C4: Mappings created with "mmap" have similar semantics to those of
>> data demand-paged from executables, except that "mmap" mappings are
>> written back to their original files, not to swap. This implies
>> that much of their implementation can be shared. Explain why your
>> implementation either does or does not share much of the code for
>> the two situations.

(I have no idea for this question, gpt's solution just for reference C: looks ok to me)
Our implementation of memory-mapped files shares much of its code with the handling of pages demand-paged from executables due to their similar semantics and mechanisms. Both involve handling page faults to load data into memory on demand, utilizing a unified page fault handler and a common supplemental page table to track metadata such as source file and offset. Frame allocation and data loading processes are identical, with shared functions simplifying the implementation and reducing code duplication. The primary distinction is in the write-back mechanism: mmap pages are written back to their original files, while executable pages are written to swap space. This differentiation is managed by the type of the SPT entry, allowing most of the code to be shared between mmap and executable paging.