# Design Doc for ASST3

David Fan & Tianyu Liu

## Introduction

The design doc is organized into the following sections:



The discussion for integration and synchronization are included inside the descriptions for each data structure.

## Overview

- For physical memory management, we use a in-kernel, global coremap to keep track of the state of all physical pages.
- For virtual memory management, we use a 2-level page table in every address space to maintain a mapping from a process's vm page to a physical page.
- We have a bitmap for managing backing store, which keeps track of available space on disk.
- For the addrspace, we use the 'region' data structure to maintain information on a process's regions.
- We will use random replacement for TLB eviction and 2-handed clock for page eviction.
- Finally, sbrk will lazily allocate pages in our addrspace API, with the actual allocation occurring on a vm fault.

# Data structure & APIs

## *Page Table*

### Overview

We use a 2-level page table for managing pages. There will be one `pt_entry` for each page of virtual memory, i.e. one pt_entry for every page in virtual memory or on disk. One page table is associated with each addrspace.

### Data structures:

```
struct pt_entry (32 bit)
    bits 31-12 p_addr;     // The corresponding translation of the physical address. It
                              will include both the 20 bit physical address and 10 bits
                              of TLB flags
    bits 31-12 store_index;// Where the entry is located in backing storage.
    bit  11    in_memory;  // true if the page is in memory
    bit  10    allocated;  // true if the page has been allocated
```

- If the page is in physical memory, the in_memory and allocated bits will be set and p_addr will point to the physical address of the page.
- If the page exists, but is not in physical memory, the in_memory bit is unset, the allocated bit is set, and p_addr will be an index into the backing store
- If the page does not exist, then allocated = 0

### API:

`pagetale_create()`: Create a new page table (level 1), allocating 1024 pointers to level 2 page tables. No level 2 page tables will be allocated. This will occur on page table entry creation. The new page table will be associated with an addrspace.

`pt_alloc_page(struct addrspace *as, vaddr_t v_addr)`: Create a new page. If the level 2 page table that should contain the page does not exist, create it. This is only called during a page fault when the faulting address is in a valid region but no page is allocated for it. Calls cm_alloc_page() to find an empty physical page.

`pt_dealloc_page(struct addrspace *as, vaddr_t v_addr)`: Sets the page referenced by v_addr to null and calls cm_dealloc_page to deal with the associated coremap entry

`pt_get_entry(struct addrspace *as, vaddr_t v_addr)`: Convenience method to get the page table entry responsible for v_addr in the given address space. Returns null if the page table entry doesn't exist

### Integration:

New files in kern/vm/pagetable.c and kern/include/pagetable.h. API mainly used by vm.c

## *Coremap*

### Overview

The coremap is a global array in the kernel that keeps track of all mappings for physical memory. It's mainly used look up page table entries when we evict/alloc new pages, or when we select a page to evict. Our coremap does not reference the pages that contain the coremap itself. coremap[0] will store data for the first page after the coremap.

- Given an index into the coremap, the corresponding physical address is PAGE_SIZE * index + mem_start
- Given a physical address, the corresponding coremap entry is (p_addr - mem_start) / PAGE_SIZE

### Data Structures

```
struct cm_entry {
      bits 31-12 vm_addr;  // The vm translation of the physical address. Only upper
                              20 bits get used
      bit  11    is_kernel;// Indicates if this is a kernel page or not
      bit  10    allocated;// Indicates if the physical address is allocated or not
      bit   9    has_next; // Indicates if we have a cross-page allocation. Only used
                              for the kernel.
      bit   8    busy;     // Indicates if the entry is locked
      bit   7    used_recently;  // Should not be evicted if true, used by clock
      bit   6    dirty;    // Indicate dirty/clean status
      struct addrspace *as;// Pointer to the address space that currently owns this
};

cm_entry coremap[(memory_end - memory_start)/pagesize];
spinlock busy_lock;            // Global lock for coremap updates
```

### API

`void cm_bootstrap()`: Initial bootstrap for coremap. Use ram_stealmem. Coremap include itself and it should marked as kernel page and thus never evicted. Called in vm_bootstrap().

`int cm_choose_evict_page()`: Evict the "next" page from memory. This will be dependent on the eviction policy that we choose (clock, random, etc.). This is where we will switch out different eviction policies. Returns the index of the coremap for the page that was evicted.

`void cm_evict_page()`: Evict page from memory. This function will update the coremap, write to the backstore, and update the backing store index for the page table entry;

`paddr_t cm_alloc_page(struct addrspace *as, vaddr_t va)`: Allocate a page of memory, pointing back to the virtual address in the address space that references it.

`paddr_t cm_alloc_npages(unsigned npages)`: Find a contiguous npages of memory

`void cm_dealloc_page(struct addrspace *as, paddr_t paddr)`: Deallocates a page of memory specified by the physical address

`paddr_t cm_load_page(struct addrspace *as, vaddr_t va)`: Load page from the backing store into a specific page of physical memory (used as a helper function for page_load)

`int cm_get_free_page()`: Linearly probes until a free page is found

### Integration

New files in kern/vm/coremap.c and kern/include/coremap.h. This API is mainly used by pagetable.c and vm.c.

## *Backing Store*

### Overview:

We use a global bitmap to record which segment in the disk is available for us to write a page. All accesses to store_map are synchronized by one global lock.

### Data Structures:

```
struct bitmap *bs_map;  // Global bit
struct *bs_file;        // The actual file object representing the disk
lock *bs_map_lock;      // Add locking to the bitmap for synchronization
```

### API:

`void bs_bootstrap(void)`: Initializes the backing store. This creates the global bitmap, lock, and backing store file.

`int bs_write_out(int cm_index)`: Writes the page represented by the coremap entry at the given index to disk. This handles all synchronization related to updating the pagetable entry. The actual writing part is done by bs_read_page.

`int bs_read_in(struct addrspace *as, vaddr_t va, int cm_index)`: Reads a page represented by the coremap entry at the given index into memory. This handles all synchronization related to updating the pagetable entry. The actual reading part is done by bs_read_page.

`unsigned bs_alloc_index(void)`: alloc a space on storage to a pagetable entry. Handles synchronization using bs_map_lock. Returns an index into backing storage.

`void bs_dealloc_index(unsigned index)`: Unmark index as available for storage. It will be called when we destroy a pagetable

`int bs_write_page(void *vaddr, unsigned offset)`: Write content from vaddr to offset in bs_file.

`int bs_read_page(void *vaddr, unsigned offset):` Read content from offset in bs_file to vaddr.

### Integration:

Addition to kern/vm/coremap.c.

## *TLB-related*

### API:

`tlb_load(pt_entry* page)`: load tlb entry based on page entry. This function is responsible for selecting the tlb entry to evict when making room for page. This is the primary interface to the TLB entry replacement algorithm. It does magic. Magic documented below.

`vm_tlbflush()`: Invalidate a specific tlb entry on the local CPU without synchronizing. This is mostly a convenience method.

`vm_tlbflush_all()`: Invalidate the entire TLB of the current CPU. This is called on a process switch or address space deactivation to clear the cache.

`vm_fault()`: Universal entry point for TLB miss and Page Fault. See Paging and TLB management for further detail

`vm_tlbshootdown_all()`: Equivalent to 'vm_tlbflush_all'

`vm_tlbshootdown()`: Call 'vm_tlbflush' and signal to 'ipi_tlbshootdown' that it completed.

**Integration**: Addition/modification to kern/arch/mips/vm/vm.c. Mainly used within the file itself.

## *Addrspace*

### Overview

In addition to page table-related logic, we need to keep track of region-related info in the struct. Each addrspace will be associated with one array of regions, and one page table.

### Data Structures

```
struct region {
        vaddr_t base;
        size_t sz;
        int permission;
        vaddr_t heap_start; // We keep track of the heap separately. The heap will be
                               placed right after the last region.
        vaddr_t heap_end;   // heap_end will be updated in sbrk.
```

```
};

struct addrspace {
     struct lock **pt_locks;      // An array of locks, each of which associated with a
                                     level 2 page table. Each lock will be malloced
                                     lazily when a level-2 pagetable is created.
     struct pt_entry **page_table; / The 2-level page table associated with the
                                     addrspace
     vaddr_p heap_start;          // We keep track of the heap for implementing sbrk
                                     [max it can go to, and where is it][define a max
                                     stack size]
     struct array *as_regions;    // An array of vm_regions. One vm_object will be
                                     associated with one addrspace.
};
```

## API

`vm_region_create(vaddr_t base, size_t size)`: Create a new vm_region based on base and size. Don't put any page here and leave all the work `to` page fault handling.

`as_create()`: Simply initialize an empty array, vm_object, for vm_regions.

`as_destroy(struct addrspace *as)`: Destroy the addrspace as. Destroy all locks, regions and finally call pt_destroy to destroy the pagetable.

`as_copy(struct addrspace *src, struct addrspace **ret)`: Creates a deep copy of src. We first copy the regions and heap, and then copy the entire pagetable. While copying the pagetbale, every single entry will be copied deeply. We copy all the data in previous pagetable entries to the disk and leave vm_fault to load them back in.

`as_define_region(struct addrspace *as, vaddr_t vaddr, size_t sz, int readable, int writeable, int executable)`: Create a vm_region based on vaddr/sz and add that to the as->as_regions array.

`as_activate(struct addrspace *as)`: calls vm_tlb_flushall() to invalidate all enties.

# TLB Management

# Paging

### *Backing store*

There is a lock for the `bitmap` struct in coremap.c to synchronize access to the bitmap.

### *Page Fault Handling*

When a page fault occurs, we will look up and load in the proper entry from the page table. We know that the page is not in memory, but because we are doing a lazy allocation of pages (sbrk and as_define region do not allocate physical pages), this page fault might be the result of access to a page that we procrastinated on allocating. There are several cases:

1. The address is not in any valid region of the address space, so we return a fault
2. The address is in a valid region of the address space, but the process does not have permissions to access the page referenced. In this case we return a fault.
3. The address is in a valid region of the address space, but the page hasn't yet been allocated. In this case we find an unused coremap entry, evicting if needed, create a second level pagetable (if necessary), create the new pagetable entry, and update the TLB with this page.
4. The address is in a valid region of the address space and the page exists, but is not in memory. In this case we find an unused coremap entry, evicting if needed, load the page from the backing store into the new space, and update the TLB.
5. The address is in a valid region, the page exists, and the page is in memory, but was not in the TLB. In this case, we update the TLB with this page.
6. The address is in a valid region, the page exists and is in memory, and is in the TLB, but was marked read only (i.e. not dirty). In this case we check that the process has write permissions, and if so, find the index of the existing entry and replace it with the same entry, but the dirty/write-access bit set.

## *Eviction*

### Linear Selection

Our basic linear eviction scheme searches for pages to evict starting from the very beginning of physical memory. This results in several issues, such as thrashing on the first available user page, and this method was quickly abandoned. Our pseudocode is still included here for completeness.

```
for each cm_entry in coremap
    lock coremap
    if (cm_entry.busy or cm_entry.is_kernel or !cm_entry.allocated)
        continue;
    else
        cm_entry.busy = true;
        return index of cm_entry
        break;
    unlock coremap
```

### Random Selection

This functions identically to linear eviction, but starts at a random index rather than always at the beginning. It continues to do linear probing from that point, wrapping around if necessary

```
cm_entry = random(0, cm_entries)
for each cm_entry
    lock coremap
    if (cm_entry.busy or cm_entry.is_kernel or !cm_entry.allocated)
        continue;
    else
        cm_entry.busy = true;
        return index of cm_entry
        break;
    unlock coremap
```

## SBRK

sbrk will simply check if the call is valid and increase the heap. Since all pages are loaded lazily in vm_fault, we won't handle any page allocation inside sbrk.

```
sbrk(size){
	if (size <= 0)
		return(current_break)
	if (heap > max_heap or heap overlaps stack)
		return(-1)
	for region in curproc->as->regions:
		if region is heap_start:
			as_heap = region
	as_heap->size += size
	return(as_heap->size - size)
}
```

## Plan of Action

[brackets denote stretch goals]

| | David Fan (approx. realistic) | Tianyu Liu |
|---|---|---|
| Sat, Mar 28 | Create pagetable and [TLB manager] | Setup and test environment |
| Sun, Mar 29 | TLB manager [fault handling] | TLB API |
| Mon, Mar 30 | Fault handling | Page table API |
| Tue, Mar 31 | Fault handling will be broken. Fix fault handling | Coremap API |
| Wed, Apr 1 | Why isn't it working!? Continue trying to fix fault handling | Debug |
| Thu, Apr 2 | Busiest day of week. No progress | Addrspace API |
| Fri, Apr 3 | Likely doing other psets. [Eviction] | Double fault handling |

| | | |
|---|---|---|
| Sat, Apr 4 | Eviction. | Eviction |
| Sun, Apr 5 | Fix eviction | Check concurrency |
| Mon, Apr 6 | SBRK | SBRK |
| Tue, Apr 7 | OMG!!! It’s due in 3 days!!! Fix SBRK | Debug Debug Debug Debug |
| Wed, Apr 8 | Fix SBRK | Debug Debug Debug Debug |
| Thu, Apr 9 | Super busy again. Tell Tianyu to fix SBRK | “I don’t want to fix SBRK!” Fix SBRK |
| Fri, Apr 10 | Submit & Sleep | Final check and submit |