Assignment 3 Design Doc

Team Longcat: Abby Lyons, Timothy Tamm

Introduction

For this assignment, our goal is to implement TLB handling and paging. We plan on implementing a two-level page table for every process for virtual-to-physical page translation. For physical-to-virtual translation, we will have a global coremap that contains the page table entry's location and other information about the page. We also need to keep track of swapped-out pages with a global swap space tracker that holds the highest used swap offset and a list of free offsets. Finally, we will implement a TLB shootdown struct to keep TLBs on different cores up-to-date. In addition to those data structures, we will also implement functions that handle address space operations, page faults, and swapping.

Data structures

Swap space tracker

We will use a linked list to keep track of 4k byte blocks in the swap space that have been previously used to store a page, but are no longer used. We will also keep track of the biggest offset into the swap device that we've used so far (this starts off as 0). We will also have to keep track of the vnode that is attached to the swap device. Everything is protected by a spinlock.

When a page needs to be written to swap space, then we will first look at that linked list. If it's non-empty, then we pop the head of that list and use the position in the swap space that that node points to. If there are no elements in the linked list, then a new block gets "generated"

by returning the biggest offset into the swap device that we've encountered so far and incrementing the biggest encountered offset by 4K bytes.

When a physical page gets freed, then the corresponding swap space block gets added to the linked list of free blocks.

All writes and reads from the swap space have to be protected by the spinlock.

Here are the structs for the swap space tracker:

```
/*
 *  A node in the linked list of free swap space blocks
 */
struct swap_block_node {
    off_t sbn_addr;                     /* address of the free block */
    struct swap_block_node *sbn_next;   /* pointer to the next free block */
};


/*
 *  Struct for keeping track of the swap device
 */
struct swap_tracker {
    off_t st_biggest_offset;                /* biggest offset given so far */
    struct swap_block_node *st_free_blocks; /* linked list of free block */
    struct spinlock *st_lock;               /* Lock for this structure */
    struct vnode *st_vnode                  /* vnode of the swap device */
};
```

Address spaces

An address space consists of the following:

- A page directory
- The current heap size
- A spinlock

We will also have a global variable that keeps track of the current active AS.
The first 128 PDEs of every process is going to be reserved for the kernel and will point to the same page tables. The kernel page will have a special global spinlock to protect it.

Page directory/table

For each process, there will be two levels of page tables, such that a virtual address consists of a 10-bit page directory index, a 10-bit page table index, and a 12-bit offset. Each page directory is an array of 1024 entries.

PDE

Each page directory entry contains a 32-bit page table address.

PTE

Each page table entry contains a 20-bit physical page number, a valid bit, a writable bit and a present bit.

Here are the relevant structures and an API for extracting and setting info in a PTE

```
/*
 *  page table entry - data structure for keeping track of virtual memory pages
 */
struct pte {
    unsigned pte_ppn:20;      /* physical page number or swap offset */
    unsigned pte_valid:1;     /* whether pte is valid */
    unsigned pte_writable:1;  /* whether page is writable */
    unsigned pte_present:1;   /* whether page is in physical memory */
    unsigned pte_padding:9;
};


/*
 *  page table - data structure for keeping track of virtual memory pages
 */
struct page_table {
    struct pte pt_ptes[1024];  /* page table entries */
};


/*
 * Address space - data structure associated with the virtual memory
 * space of a process.
 */
struct addrspace {
    struct page_table *as_pd[1024]; /* the page directory of the addrspace */
    size_t as_heap_size;            /* the current heap size */
    struct spinlock *as_lock;       /* lock to protect this struct */
};
```

Core map

The coremap contains an entry for each physical page. Each entry will contain:

- A pointer to the address space.
- A virtual address.
- Where in the swap device this page is backed.
- Which CPU the page belongs to.
- Whether the page is dirty or not.
- Whether the page is in the TLB.
- Whether the page is currently being accessed (busy bit).

- A kernel bit, which will be used to keep track of contiguous kernel memory blocks. The kernel bit should be set as 0 for most pages. If a block of more than 1 pages gets allocated using alloc_kpages, then the kernel bit will be set to 1 for every page after the first one in the contiguous block.

The core map will be protected by one spinlock. The core map will also have a wait channel.

Additionally, there will be a linked list of free physical pages. Whenever a physical page gets allocated, pop the first element from that list, if there is one. If there isn't one then evict a page. Whenever a physical page gets freed, then it is added to the head of this list. This list will be protected by the coremap's lock.

Here are the relevant structs and some of the API calls:

```
/*
 *  Struct for a core map entry
 */
struct cm_entry {
    struct addrspace *cme_as;   /* pointer to the address space that owns this page */
    vaddr_t cme_vaddr;          /* the virtual address in the address space */
    off_t cme_swap_location;    /* location of this page in the swap device */
    struct cpu *cme_owner_cpu;  /* which cpu the thread which has this PTE runs on */
    unsigned cme_dirty:1;       /* whether page has been written to */
    unsigned cme_tlb:1;         /* whether page is in tlb */
    unsigned cme_busy:1;        /* whether page is busy */
    unsigned cme_kernel:1;      /* whether page is in a contiguous kernel block */
};

/*
 *  Struct for a linked list node for tracking free CM entries
 */
struct cm_node {
    struct cm_entry *cmn_entry; /* the free CM entry */
    struct cm_node *cmn_next;   /* pointer to the next node */
};

/*
 *  Struct of the core map.
 *  The core map is responsible for keeping track of physical pages
 *  ram pages will be 512MB / 4KB
 */
struct core_map {
    struct cm_entry cm_entries[RAM_PAGES];  /* the core map entries */
    struct cm_node *cm_free_entries;        /* linked list of free entries */
    struct spinlock *cm_lock;               /* lock protecting this struct */
    struct wchan *cm_wchan;                 /* wait channel for the core map */
};
```

Struct tlbshootdown

The tlb shootdown struct needs to contain the following information:

- Which cpu's TLB to shootdown
- A virtual address to shootdown.

- A flush_all boolean variable. If this is true, then flush the entire TLB.

VM Functions

These are functions declared in vm.h, initially implemented in dumbvm.c, and newly implemented in a new file, vm.c.

Vm_swap_init (new function)

- Call **vfs_swapon**, passing in "**lhd0**" as the device name to initialize the swap device
- Store a pointer to the swap devices vnode in a global variable.
- Initialize the swap space usage tracker, by setting the biggest offset to 0 and making the head of the free blocks linked list point to NULL
- Initialize the swap spaces spinlock.

Vm_bootstrap

This is called in main.c. We want to initialize all of the global data structures described above.

- Swap space usage tracker initialization (call **vm_swap_init**)
- Core map initialization
    - Figure out how many physical pages there are.
    - Calculate the size of the coremap, and call **ram_stealmem** with enough pages.
    - Allocate an array of coremap entries.
    - Set the linked list head to NULL.
    - For each entry, set all bits low, and create a linked list entry that can be pushed into the coremap linked list.
    - For the entries that hold the coremap itself, set the kernel bit high (except the first, which should remain low).
    - Create a spinlock.
    - Create a wait channel.

Vm_fault

When a TLB fault occurs, then a handler will get called in exception-MIPS1.s, which in turn calls **mips_trap** in trap.c, which finally calls **vm_fault**. In **vm_fault** we have to do the following:

- If there's no current process or address space, then the fault must have happened early in the boot process and there's probably nothing we can do about it. Panic or something.
- Check to make sure the address space is set up properly.
- Acquire the AS's spinlock.
- Traverse the active AS page table to find the corresponding PTE.
- Check if the PTE is a valid entry; if not, raise a page fault.

- Release the AS's spinlock.
- Check if the PTE's present bit is set.
    - If it is, then get the physical page number.
    - If it is not, then raise a page fault.
- Get the core map's lock
- Check if the corresponding physical page's busy bit is high.
    - If it is, then release the AS lock. go to sleep on it's wchan. When woken up, release the core map lock, acquire the AS lock, then reacquire the coremap lock and redo the previous check
    - If it isn't then set it high.
- Release the core map lock.
- Disable interrupts
- If the fault type was **VM_FAULT_READ** or **VM_FAULT_WRITE**:
    - Probe the processor's TLB to find an empty slot.
    - If you found one:
        - make a corresponding entry in the TLB.
    - If you didn't find one:
        - use our TLB replacement algorithm to figure out which entry to evict.
        - Evict the chosen page
        - Write the entry into the now free slot
        - Set the corresponding physical page's core map entry's **IN_TLB** bit high.
- If the fault type was **VM_FAULT_READONLY**:
    - There are 3 cases that might have caused this:
        - We are trying to write to a read only page. To check this, check if the PTE's writable bit is set to 0:
            - Raise an appropriate error
        - We are trying to write to a clean page:
            - Update the TLB entry as writeable
            - Mark the corresponding physical page entry in the coremap as dirty.
- Release the AS's spinlock.
- Acquire the core map spinlock
- Lower the busy bit
- Release the coremap's lock.
- broadcast on the coremap's wait channel.
- Restore interrupts.

Alloc_kpages

- Grab the kernel AS's lock
- Grab the coremap's lock
- Scan through the coremap, looking for **npages** contiguous free entries. An entry is free if the entry doesn't exist or it's PTE pointer points to NULL.

- If you can't find such a block, then try to find a contiguous block of **npages** free or non-kernel pages. If you can't find that either, return NULL
    - Evict necessary pages
- Remove these pages from the linked list of free pages.
- For each of these pages:
    - make an entry if necessary.
    - Set the virtual address to be the direct mapped address using **PADDR_TO_KVADDR**.
    - Look at the corresponding PTE in the kernel Address space:
        - Set the valid bit, writable bit and present bit high
        - Set the dirty bit low.
    - In the core map entry, set the swap space address as NULL (kernel pages are never swapped).
    - Set the valid bit high.
    - Set the **in_tlb** bit and dirty bit low.
    - If this is the first page, then set the kernel bit low. Otherwise set the kernel bit high.
- Zero out the new pages using **as_zero_region**.
- Release the kernel AS's lock
- Release the coremap's lock.
- Return the virtual address of the first page in the block.

Free_kpages

- Assert that the virtual address is in the kernel (by checking that it's in the first 128 PDEs)
- Grab the lock of the kernel AS.
- Look up the PTE of the virtual address that was given.
- Invalidate the given PTE, by lowering the valid bit.
- Acquire the coremap's lock.
- Find the corresponding physical page's core map entry.
- Assert that the kernel bit is 0.
- Invalidate it by setting it's valid bit low.
- Add that page to the head of free pages list.
- While (true):
    - Check the next PTE.
    - Find the corresponding physical page's core map entry
    - If the kernel bit is 1, then release the coremap lock and break from the loop.
    - Invalidate it by setting it's valid bit low.
    - Add that page to the head of free pages list.
    - Invalidate the PTE.
- Release the coremap's lock
- Release the kernel AS's lock.

Vm_tlbshootdown

This function should get called whenever a page gets evicted, or if **sbrk** gets called with a negative argument. Here's what has to happen:

- If the target cpu is not the current cpu, then call **ipi_tlbshootdown** with the given tlbshootdown struct and target cpu.
- Otherwise:
    - Check the **flush_all** variable. If it's true, then use **tlb_write** to loop through all TLB entries and invalidate them.
    - Else, use **tlb_probe** and **tlb_write** to invalidate the given entry.

AS Functions

As_create

- Create a new address space that contains an empty page directory.
- Initialize a spinlock.
- Initialize a wait channel.
- If any of these fail, return NULL.
- Set the current heap size to 0.

As_copy

- Call as_create to make a new address space. If this returns NULL, return ENOMEM.
- Lock the source address space.
- Lock the destination address space.
- Skip over first 128 PDEs--those belong to the kernel.
- Grab the coremap lock
- For every remaining page directory entry in the source:
    - Get the source page table.
    - For every valid PTE in the source:
        - If the page is present, then check the coremap entry to see If the page is busy. If it is:
            - Go to sleep on the coremap's waitchannel, and try again after being woken up.
        - If the page is not busy and writable (or not writable, but this is subject to change):
            - Copy the contents of the page into a new page.

        - Check if the source page is dirty.
            - If it is, then:
            - Set the page's coremap entry's busy bit high.
            - Release the coremap lock
            - Release the source and target AS locks
            - write the page to swap
            - Grab the source AS's lock
            - Grab the target AS's lock
            - Grab the coremap lock
            - mark the page as clean in the coremap
            - Lower the page's busy bit
        - Find out where the source page is stored in the swap device.
        - Find an empty block in the swap device. If this fails, return **ENOMEM**.
        - Release the coremap lock
        - Copy the source page contents into the empty block in the swap device.
        - Acquire the coremap lock
        - Make a new PTE in the target PDE in the corresponding location
        - Set the present bit bit low.
        - Set the valid bit and writable bit high.
        - Set the physical page number to be the block number of the swap block where we stored the page.
- Release the coremap's lock.
- Copy the heap size from the source AS to the target AS.
- Unlock the the source AS.
- Unlock the destination AS.

As_activate

- Shoot down the current CPU's TLB completely (by setting **flush_all** true in the struct).
- Set the current active AS pointer to point to the given AS using **proc_setas**.

As_deactivate

Set the current active AS pointer to point to NULL using **proc_setas**.

As_destroy

- Acquire the AS spinlock.
- Skip over first 128 PDEs.
- Acquire the coremap's lock.
- For every remaining PDE in the address space:

    - Get the source page table. For each PTE:
        - If it is present, check if it is busy
        - If it's busy:
            - Go to sleep on the coremap'smwaitchannel, and try again after being woken up.
        - If it's not present or present, but not busy and writable (or not writable, but this is subject to change):
            - If the page is in the TLB, mark its corresponding entry as invalid.
            - If the page is present, evict it (see Paging section).
            - Return its swap device block by creating a new linked list node with the swap space offset, acquiring the swap space usage tracker spinlock, adding the node to the usage tracker, and releasing the spinlock.
- Release the coremap's lock
- Release the AS's spinlock.
- Destroy the spinlock.
- Destroy the wait channel.
- Free the address space.

As_define_region

- Acquire the address space spinlock.
- Ignore the executable and readable arguments, and check that memsize is nonnegative, the address space is non-NULL, and the vaddr is non-kernel and within range.
- If L2 page tables do not exist at the given addresses, allocate them and update their respective PDEs. Error with **ENOMEM** and release the lock if this fails.
- For each newly created L2 page table:
    - Set all bits low.
    - Set the writable bit high if the writable argument is nonzero.
- For each already existing page in the given range:
    - Set the writable bit high if the argument is nonzero; otherwise, set the writable bit low.
    - If you changed the writable bit in the PTE:
        - Grab the coremap's lock
        - Check if the page is in the tlb
        - If if is then shoot down that tlb entry and lower the in_tlb bit
        - Release the coremap's lock
- Release the address space spinlock.

As_prepare_load

- This function doesn't need to do anything.

As_complete_load

- This function doesn't need to do anything.

As_define_stack

- Set the stackptr to **MIPS_KSEG0**.

TLB replacement algorithm

We will use a random TLB replacement algorithm. This means that if a TLB entry needs to be evicted, then out of the clean pages we just randomly pick one.

Paging

The following functions will be in a new file, paging.c (and paging.h).

page_fault

We will have a fault handler function which deals with page faults as follows:

- There are 2 cases we need to handle:
    - The PTE's page is in swap:
        - Evict a current physical page if necessary and bring the new page into physical RAM.
    - The PTE entry is invalid
        - Check if the virtual address is less than **STACK_MAX**. If it is not, then kill the current thread with a **SEGFAULT**.
        - Check if the page is in the kernel (by checking if it's in the first PDE). If it is, then it should have been allocated with alloc_kpages, so kill the current thread with a **SEGFAULT**.
        - Make a new PTE entry which is writeable. Allocate it a physical page, evicting another one if necessary. Allocate the page a slot in the swap space and mark it as dirty. Finally zero out the data in the physical page.

evict_page

The evict_page helper function takes in a pointer to a PTE, a pointer to an AS and a boolean variable write_to_swap. If write_to_swap is true, then the function will write the page to swap before evicting it. The function returns the now empty core map entry.

Before calling this function, the user has had to set the busy bit of the core map entry.
After the page has been freed, it's up to the user to do whatever she wants with it, but keep in mind that in order to actually free the page, the busy bit has to be set low and the core map entry has to be added to the linked list of free entries.

- If an AS was passed in, then assert that we hold its lock.
- Grab the coremap's lock
- Find the evictee's core map entry
- Assert that the evictee's core map entry's busy bit is set high.
- Set the present bit low in evictee's PTE.
- If the write_to_swap parameter was given, check if the page is dirty
    - If it is, then
        - If an AS was passed in, then release its lock.
        - Release the coremap's lock
        - write the page to the swap device in the position given in the core map entry.
        - If an AS was passed in, then acquire its lock.
        - Acquire the coremap's lock
    - Mark the page as clean in the evictee's PTE.
- If the page was in the TLB, shoot down the appropriate CPU's TLB.
- Change the evictee's PTE's physical page address, to the address of the page in the swap device.
- Set the core map entry's PTE pointer to NULL.
- Return a pointer to the core map entry.

swap_pages

When a physical page has to be replaced by a new page, then we will

- Grab the core map's lock
- find a physical page to evict using our paging algorithm.
- Release the core map's lock
- Grab the evictee's AS' spinlock
- Acquire the coremap's lock.
- Check the evictee's coremap entry's busy bit:
    - If it is high, then go to sleep on the coremap's wchan. When woken up check the busy bit again. If it is still high, go back to sleep.
- Set the evictee's coremap entry's busy bit high.
- Release the coremap's lock.
- Call the **evict_page** helper function, with write_to_disk set to True and passing in the evictee's AS as the AS parameter
- Release the evictee's AS's lock.
- Read the new page from the swap device into the physical page we freed. The new page's location in the swap device will be the physical page number in its PTE.

- Acquire the coremap's lock.
- Update the coremap entry to reflect the new owner PTE and cpu.
- Store the new page's location in the swap device in the coremap entry.
- Mark the page as clean in the coremap entry.
- Mark the page as not being in the TLB.
- Grab the evictee's AS' spinlock
- change the new page's PTE's physical page number to the page number of the physical page we were working with.
- Lower the coremap entry's busy bit.
- Release the coremap's lock.
- Release the evictee's AS's lock.
- Broadcast on the core map's wchan.

Writing daemon

- Check all physical pages and find the dirty ones.
- For each dirty page, write it into swap space, and then mark the coremap entry for that page as clean. Reset the corresponding TLB entry (if there is one) as read-only.
- Scheduling the daemon:
    - After completion, the thread should put itself to sleep on a global waitchannel.
    - In clock.c, modify hardclock() such that if **curcpu->c_hardclocks** % **WAKE_DAEMON_HARDCLOCKS** == 0, it wakes up the daemon so it can be scheduled again. The macro can be tuned to run the daemon more or less frequently.

Thread migrations

In **thread_consider_migration** in thread.c, add the following:

- After adding a thread to the tail of the other CPU, iterate through the coremap, find every entry that belongs to that thread, and update which CPU that page belongs to.

Paging algorithms

Question

Q: How will you structure your code (where will you place your paging algorithms) so that others can be added trivially, and switching between them only requires reconfiguring your kernel?

A: Most of our code for paging will be in a file called paging.c. In paging.c we will have a function called **find_page_to_evicit**, which use kernel macros to call one of multiple paging algorithms
Ex:

```
#ifdef USE_PAGING_ALGO_1
Page_to_evict = paging_algo1();
#endif
#ifdef USE_PAGING_ALGO_2
Page_to_evict = paging_algo2();
#endif
```

This way the paging algorithms can be written in separate files.
Finally, at the top of paging.h, we will have a macro for each of these paging algorithms, but all except for one should be commented out.
Ex:

```
//#define USE_PAGING_ALGO_1 1
//#define USE_PAGING_ALGO_2 1
#define USE_PAGING_ALGO_3 1
//#define USE_PAGING_ALGO_4 1
```

In this example, the third paging algorithm will be used.

Algorithm

We will use a clock based algorithm to approximate a LRU paging algorithm
The coremap will have a 32bit integer acting as a clock. It will be initialised to 0 and it will be incremented every time anything touches the coremap. Each core map entry will also have a last_clock variable. Whenever a coremap entry is touched, set its last_clock value to the current coremap clock value.

To choose an entry to evict, do the following If at any step there is only one page left, use that one:

1. Go through the coremap, and find all the clean pages
2. Go through all the clean pages and find all the non-busy ones
3. Among the pages left over, choose the one with the lowest last_clock value.

Performance analysis

This will be included in the final submission.

Questions

TLB Questions

Problem 1

- TLB miss, page fault: a page was swapped out to disk and evicted from the TLB.
- TLB miss, no page fault: a page is in physical memory, but was evicted from the TLB.
- TLB hit, page fault: cannot happen
- TLB hit, no page fault: a page is in physical memory and the TLB.

Problem 2

- An instruction that references something in memory gets executed. First the TLB entry references the place in memory where the code for the instruction is stored. Then a TLB Fault occurs, because the piece of memory that the instruction references is not in the TLB. It gets pulled into the TLB (and the code gets evicted). Now the instruction gets executed again, but now that is no longer in the TLB, so you get another TLB fault. So it gets pulled into the TLB, the memory that the instruction references gets evicted, the instruction gets re-executed and we're right back where we started.

Problem 3

- Accessing this instruction can cause:
    - TLB miss, if the virtual page containing the instruction is not in the TLB.
    - Page fault, if the physical page is not in physical memory.
    - Permission fault, if the read and execute bits are not set.
- Accessing the data at 0x0120 can also cause:
    - TLB miss, if the virtual page is not in the TLB.
    - Page fault, if the physical page is not in physical memory.
    - Permission fault, if the read bit is not set.
    - Invalid page fault, if the address is not in the virtual address space or is otherwise an illegal access.

Malloc Questions

- Question 1
    1. Depending on the state of the system before, sbrk might get called 0 times or once. If free blocks can't be found in the current state of the system, then sbrk will get called once with the page size (4K). That would be more than enough memory for the 10 malloc calls (each malloc call asks for 10 bytes, but because memory is given in 8 byte blocks, each call actually needs 16 bytes of memory, so a total of 160 bytes).
    2. If sbrk got called, then the value is 4096 bytes. If not, then it's 0 bytes.
- Question 2

    - If sbrk was called before section 2, then it will not be called again. However, if it was not, then it will be called once with the page size (4K) if there are no free sections of 64 bytes remaining.
    - We can say that x must be greater than res[i] for 0 <= i < 10. None of the freed blocks add up to 64 bytes, so the blocks allocated for x must be after all of those.
- Question 3
    - This designates that these functions are supposed to only be used internally. In a higher level language you would declare these functions private, but in C you can't do that, so you rely on naming conventions.
- Question 4
    - This is guaranteed by 2 facts:
        - The heap starts off properly aligned
        - Malloc only allocates memory in blocks of 8 bytes (on a 32bit system) or 16 bytes (on a 64bit system).

Sbrk syscall

The sbrk syscall should:

- If the size parameter is greater than 0:
    - Check if the new heap bound would cross **STACK_MAX**. If it does, then error with **ENOMEM**.
    - Allocate the desired number of physical pages for the virtual address growing up from where the current heap is at (this info is stored in the address space)
    - Update the current heap bound in the address space.
    - Return the new heap bound.
- If the size is less than 0:
    - If the new heap bound would make the heap size less than 0, then error.
    - Free any pages that are between the new and old heap bounds. To free a page, delete its PTE, clear its core map entry if it was in physical memory and if it was in physical memory and in the TLB, then call **tlb_shootdown**.
    - Update the heap bound and return the new heap bound.

Trylocks

Currently, spinlocks do not have a function to attempt to acquire the lock and return immediately, so in spinlock.c we will implement:

Spinlock_try_acquire

Spinlock try_acquire is very similar to spinlock_acquire. The only difference is that you don't have the while loop. Replace the **break;**'s with **return 1;**'s, get rid of the **continue**s and add a

return 0; at the end of the function. Now if the try was successful, then we will return 0 and 1 otherwise.

Plan of Action

We plan to start working on our assignment halfway through spring break

1st week:
M: break
Tu: break
W: break
Th: implement supporting data structures and APIs
F: implement supporting data structures and APIs
Sat: addrspace functions
Sun: addrspace functions

1st week:
M: vm functions
Tu: vm functions (at this point, we should hopefully be able to run our kernel without paging)
W: paging
Th: paging
F: writing daemon
Sat: testing
Sun: testing

3rd week:
M: testing
Tu: testing
W: paging algorithm optimization
Th: paging algorithm optimization
F: paging algorithm optimization
Sat: Discover joy in life once more.
Sun: partayh!

Credits

We would like to thank Prof. James Mickens for teaching us the concepts, and Alex Patel, Victor Domene, and Team Streptococcus pyogenes for helping us refine our design.