Computer Science 161: Operating Systems
Assignment 3 Design Document Draft

David Li and Garrett Tanzer
March 9, 2017

1. Introduction

Assignment 3 can be broken up into three major data structures (page tables hierarchies, the core map, and swap) and the operations performed to maintain them (populating/evicting entries from the TLB, swapping pages from disk to RAM as needed, and asynchronously writing dirty pages to disk with a daemon).

We will implement 2-level page tables, with each level indexed by 10 bits of the virtual address and adding 12 bits of offset at the end to get a 32-bit physical address. The core map will be an array of entries for each page of physical RAM with minimal bookkeeping information. Swap will be tracked with a bitmap at least twice the size of physical RAM and rounded up to the nearest page size. Each will be protected by a coarse spinlock (for each address space, for the core map, and for the swap bitmap).

TLB eviction will be largely random, page swapping will be dictated by a variant of the clock algorithm based on presence in the TLB, and the write-back daemon will loop over the core map sequentially and save changes marked during write faults to disk. The main challenge with these operations performing in parallel is synchronization; our algorithms and the primitives we use to ensure they are performed atomically are explained in much greater detail below.

2. Overview

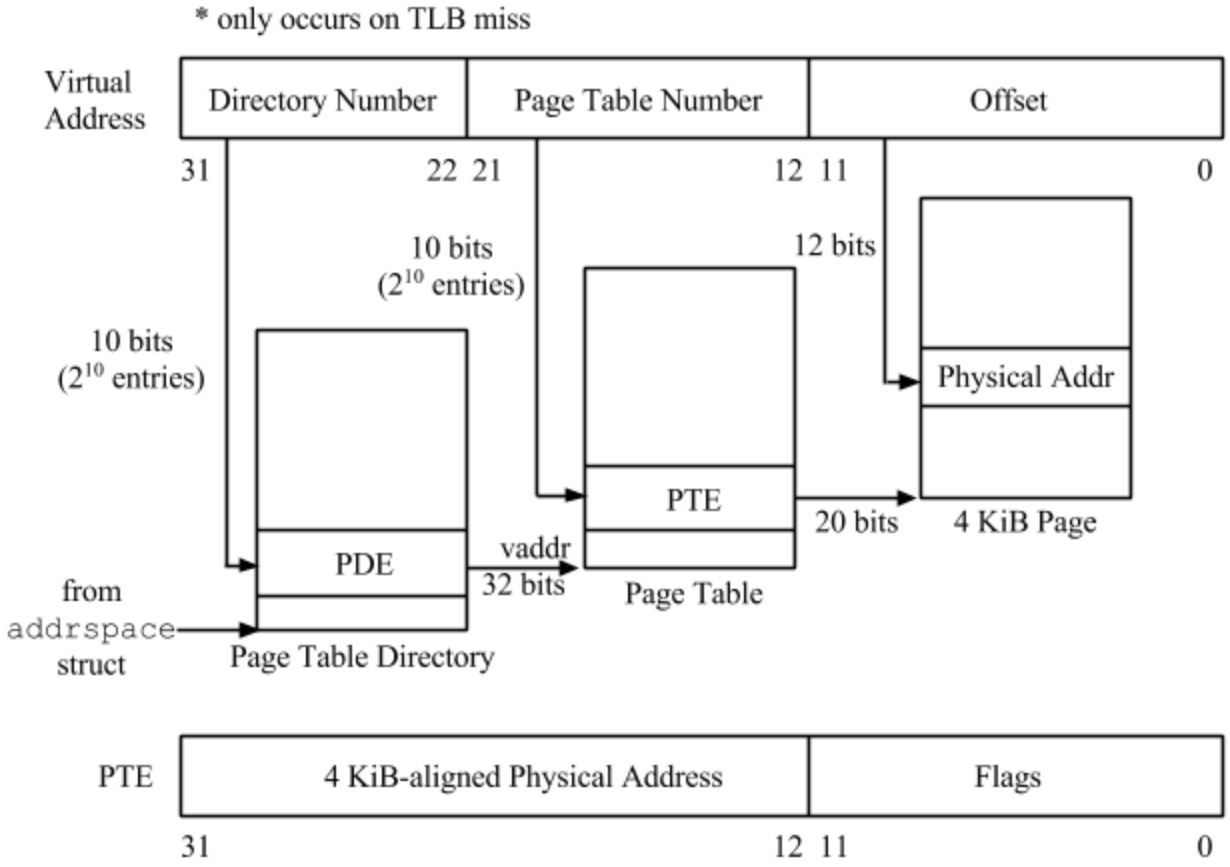
Page Tables

The data structures for virtual to physical address translation begin in the `addrspace` struct:

```
struct addrspace {
      struct page_table_directory* ptd;
      struct spinlock addr_splk;
      vaddr_t heap_top;
};
```

`ptd` provides an entry point into the hierarchical page tables, while `addr_splk` protects modifications to this address space. `heap_top` is manipulated by `sbrk()` and is used to check for invalid memory accesses.

Here is a diagram showing how two-level page tables are indexed to reach a full 32-bit physical address:



The `ptd` field of `addrspace` is a wrapper for an array of `page_table` pointers (physical addresses of subsequent page tables or NULL if not yet created). Each `page_table` has an array of 4-byte entries, which store a 20-bit physical frame and 12 bits of flags. Both page table levels are designed to take exactly one page of memory each so that they can fill exactly one use of `alloc_kpages()`.

```
struct page_table_directory{
      struct page_table* pt[1024];
};
```

```
struct page_table {
      uint32_t pte[1024];
};
```

Each of the PTEs in `page_table` is organized as follows:

PADDR/SWAP	TBD	TBD	TBD	TBD	TBD	TBD	TBD	TBD	TBD	TBD	PTE_W	PTE_P
31-12	11	10	9	8	7	6	5	4	3	2	1	0

PTE_P: 1 if present in memory, 0 if in swap

If present in memory, the physical address is stored in the top 20 bits. If it is in swap, the index of the page in swap is stored in the top 20 bits.

PTE_W: 1 if writeable, 0 if read-only

Rather than having to manually traverse the page table with `curproc->p_addrspace->ptd->pt[index1]->pte[index2]`, we will define a function `as_pte(vaddr_t vaddr)` that returns the PTE mapped by the virtual address parameter in the current address space (or 0 if it has not yet been mapped).

Core Map

The core map is a global array with bookkeeping data on all page frames in RAM. It will have `ram_getsize() / PAGESIZE` entries and must be initialized early in the bootstrap process using `ram_stealmem()`, since the core map is required for `kmalloc()` to work. The entire array will be protected by a single spinlock.

```
struct core_map_entry {
      vaddr_t va;
      struct addrspace *as;
      struct core_map_entry *next;
      uint32_t metadata;
};
```

`va` and `as` are used to locate the page table entry pointing to the physical address, while `next` is used to construct a linked list on top of the core map's primary array structure for the clock page swapping algorithm. Whenever a physical page of memory is allocated, it should be inserted into a circular linked list at the global pointer `clock`.

Each `metadata` in the core map is structured as follows:

SWAP	S_PRES	RECENT	TLB	DIRTY	USE	CONTIG	KERNEL	BUSY	SPINL?
14-31	8	7	6	5	4	3	2	1	0

**SPINL: Spinlock bit: Is the spinlock on for this core map entry?

BUSY: If this page is currently in the process of being swapped

KERNEL: Is the page owned by the kernel of the process?
CONTIG: Marks the end of a contiguous allocation of kernel pages
USE: Has this page been read from?
DIRTY: Has this page been written to since the swap was last updated?
TLB: Is this page currently cached in a processor's TLB?
RECENT: Recently evicted from a TLB
S_PRES: Is the swap copy of this page present?
SWAP (not a constant): Index of the corresponding page on the disk: Assuming a max swap size of 1GiB: 512MiB max RAM * 2

Note: When a swap is needed and the page being evicted has already been preemptively copied to disk, S_IND will be transferred to the pagetable entry of the page being evicted, and S_PRES and S_IND will be reset.

**Possible Optimization: a fine-grained spinlock will be used to protect each core map entry. The spinlock variable is stored as one bit in the `uint32_t` metadata field. New spinlock functions will have to be written based on spinlock_data_test and spinlock_data_testandset.

Swap

The swap space is a region of the hard drive which stores pages either already evicted from RAM or preemptively copied from RAM, in anticipation of needing the page evicted from RAM. We will keep track of occupied/free sectors on the swap device using a `struct bitmask` of length 2 * `ram_getsize() / PAGESIZE` bits, rounded up to the nearest page. So, with physical RAM limited to 512 MiB, this means our swap device can be between 128 and 1024 MiB inclusive. This space was made larger than RAM to ensure that even if all the pages are preemptively sent to swap, there will still be space for others to actually be swapped in/out.

The bitmap will be protected by a single spinlock. Whenever a new swap index is needed, the first 0 in the bitmap is switched to 1 under the spinlock's protection.

3. Topics

TLB Misses

When `VM_FAULT_READ` or `VM_FAULT_WRITE` are triggered:

- Acquire the address space's spinlock
- If the faulting address is a valid, mapped user address:
    - Acquire the core map's spinlock

    - If the page is BUSY, release both locks and loop until it's not anymore, then reacquire them
    - If the page is valid but in swap:
        - Advance `clock` through the circular linked list until you find an entry that is not in the TLB, is not BUSY, and is not RECENT (if it was recently in the TLB, set that bit back to 0 after iterating past it) (keep track of where you started, and if you loop around, start accepting even pages in a TLB—but then you have to broadcast a TLB shootdown)
            - Maybe just use the array itself instead of a linked list?
        - If the page hasn't already been written to swap by the daemon (check S_PRES), do so now
        - Update the core map to reflect the new write back swap index
        - Load the desired page from swap into the physical page of RAM
        - Update its page table entry to reflect the new physical address in RAM
    - Once the page is present in the core map, mark the TLB bit of its physical page entry with 1.
    - Use OS-level random function to determine which TLB entry to evict (because `tlb_random()` only selects 56 of 64 entries)
    - We read this entry and mark its TLB bit 0 and RECENT bit 1 in the core map
    - Turn off interrupts
    - Next, we write the new address with `tlb_write()`, with the address space ID in TLBHI unused, and the global and valid bits in TLBLO set—but not the writable bit (this is so we can keep track of the dirty bit)
    - Change back interrupts
    - We release the locks and vector control back to user space, with the faulting instruction repeated.
- Else release the spinlock and segfault

Page Faults

When `VM_FAULT_READONLY` is triggered:

- Acquire the address space's spinlock
- If the faulting address is a valid, mapped user address:
    - If the permissions are read-only in the page table, release the lock and kill the thread
    - Acquire the core map's spinlock
    - If the physical page is BUSY, release both locks, loop until it's not busy, and reacquire them
    - Mark the associated physical page's DIRTY bit

    - Update the TLB entry for the faulting address with a writeable bit
    - Release the locks and vector control back to user space, with the faulting instruction repeated
- Else release the spinlock and fault for insufficient permissions

MAT Daemon

The MAT (Memory Access Traversal – and Write-Back) Daemon is a background process spawned during boot to proactively write dirty pages from RAM to the swap space. It will also update pages on the swap if they have already been placed onto swap. This kernel-mode process should be given a low priority in the scheduler since it is not crucial to the functioning of the operating system. Additionally, it will wait a set amount of time (using gettime() to get the time), before repeating its job again. MAT Daemon will not run if no swap space is left. The specific amount of time will be fine tuned through performance tests. Below is the algorithm for the function:

- Acquire the core map's spinlock
- Start looping over entries in core map (for loop inside a while loop so it cycles forever)
- Ignore entries with the `KERNEL` bit or `TLB` bit (so no shootdowns necessary) set
- Check if the `DIRTY` bit is set
- Set `BUSY` bit in coremap
- Relinquish the core map's spinlock
- If `S_PRES`, use `S_IND` to calculate disk location, and write page to disk
- Else, lock `bitmap_lock`, find next available space on swap and set the bit, relinquish `bitmap_lock`, then write page to disk
- Reset `DIRTY` bit in coremap
- Take `coremap_lock`, reset BUSY bit, relinquish `coremap_lock`
- `thread_yield()` after a certain amount of time

Care must be taken to account for conditions where the swap space is already full.

MAT Daemon never acquires the address space lock, so there is no worry about deadlock—the thing we need to worry about is making sure the way we use the BUSY bit is exhaustive (since it was added after writing the algorithms initially), meaning that nothing will end up trying to access pages during the swap process.

```
void* sbrk(intptr_t amount);
```

Extends the top of the heap for the address space (i.e. `heap_top`) by the `amount`, updating the page tables accordingly. We should create a constant to indicate the greatest extent of heap

expansion

- Get address space and take the spinlock
- Check page-alignment of amount
- Store old heap_top

If amount is positive:

- Test for `ENOMEM` condition
- Use `as_define_region()` to allocate new pages
- Increase heap_top
- Relinquish all spinlocks

If amount is negative:

- Test for `EINVAL` condition
- Get core map's spinlock
- Decrease heap_top
- Zero out pages in core map corresponding to virtual addresses between old and new heap_top, marking associated swap pages as free in the process
- Mark associated pages only in swap as available with the swap spinlock
- Send TLB shootdowns for the removed pages
- Release all spinlocks

Return pointer to new heap_top

Errors:

- `EINVAL` for an `amount` that is either not page-aligned or would make `heap_top` negative
- `ENOMEM` for insufficient virtual memory

as_() Functions

```
struct addrspace* as_create(void);
```

Create a new empty address space (create minimum page table entries, initialize spinlock). Return NULL if out of memory.

```
int as_copy(struct addrspace *src, struct addrspace **ret);
```

Do a deep copy of the `src` address space into a new address space saved to `ret`. During the process, the `src` and core map spinlocks must be held.

```
void as_activate();
```

Flush the TLB. Update the TLB flags in core map entries while doing so (with spinlock protection).

```
void as_deactivate();
```

Flush the TLB. Update the TLB flags in core map entries while doing so (with spinlock protection).

```
void as_destroy(struct addrspace *as);
```

Iterate through the address space and free all remaining referenced pages. Clean up the address space's spinlock. The core map's spinlock

---

```
int as_define_region(struct addrspace *as, vaddr_t vaddr, size_t
sz, int readable, int writeable, int executable);
```

Map the designated virtual memory region to physical memory with the specified permissions.

- Get core map's spinlock
- Set `counter = ceil(sz/PAGESIZE)`
- LBL: Loop for every entry in core map
- If entry is empty (has no vaddress)
    - Map it to the appropriate virtual page, adding more pagetables entries if necessary, set the permissions for reading or writing
    - Set the `vaddr` and `as` fields in coremap; Also set it to BUSY to prevent it from being swapped out if sbrk itself will need to swap things out. Set KERNEL metadata field as well.
    - Decrease `counter` for every successful addition; break when `counter == 0`
- End loop
- Relinquish core map's spinlock
- If `counter > 0`:
    - Run swap code to swap out any remaining pages needed to be retrieved
    - Goto LBL

- Traverse the newly allocated VADDR range; use these addresses to go back to the core map and remove the BUSY bits.

```
int as_prepare_load(struct addrspace *as);
```

Prepare address space for loading an executable.

```
int as_complete_load(struct addrspace *as);
```

Complete loading an executable into the address space.

```
int as_define_stack(struct addrspace *as, vaddr_t *stackptr);
```

Allocate space and populate page tables from the top of user address space down to the given stack pointer.

4. Plan of Action

1. Complete this document
2. …
3. Profit

Code Reading Answers

- TLB miss, page fault
    - Attempting to access something paged out to disk
    - New page is needed
- TLB miss, no page fault
    - Shouldn't happen; Page faults are used to bring in new pages
- TLB hit, page fault
    - Memory is present, but permissions are lacking (Usermode trying to access kernel)
    - Trying to read on write only or write on read only
- TLB hit, no page fault
    - Normal access of memory that was cached in the TLB
- A friend of yours who foolishly decided not to take 161, but who likes OS/161, implemented a TLB that has room for only one entry, and experienced a bug that caused a user-level instruction to generate a TLB fault infinitely—the instruction never completed executing! Explain how this could happen. (Note that after OS/161 handles an exception, it restarts the instruction that caused the exception.)
    - (Naive answer) Memory outside currently allocated pages was accessed, and there is no more room in physical memory, or any addressable swap space.
    - The friend allocates a local variable inside the handler itself, which could potentially page fault and require another TLB entry.
- How many memory-related exceptions (i.e., hardware exceptions and other software exceptional conditions) can the following MIPS-like instruction raise? Explain the cause of each exception.
    - Requested address is invalid (Negative or not aligned to word)
    - TLB entry does not exist
    - Process has no permission for the memory region it's trying to access
    - It's write only
- How many times does the system call sbrk() get called from within malloc()?
    - Malloc only needs to call sbrk once
- On the i386 platform, what is the numeric value of (finish - start)?
    - 0x21000
- Again on the i386, would malloc() call sbrk() when doing that last allocation at the marked line above? What can you say about x?
    - The last allocation does not need sbrk. The allocation pointed to by x is on the same page or same page range as the first allocation.
- It is conventional for libc internal functions and variables to be prefaced with "__". Why do you think this is so?

    - Name mangling; A user program or perhaps the kernel might want to define a function of the same name. Having the underscore(s) prevents the collision in name
- The man page for malloc requires that "the pointer returned must be suitably aligned for use with any data type." How does our implementation of malloc guarantee this?
    - Allocations are rounded up to an integral number of blocks. Non-fundamental data types are aligned to their greatest member. The sizes of the greatest members is a multiple of where each block starts.
- 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?
    - Create a header file similar to opt-dumvm.h containing many constant toggles. Then, when implementing you can use #if, #else, and #endif with the toggles. Changing the toggles in the header file will change what is being compiled.