Winfried Schuffert
Dustin Lee
Felix Kosmalla

Assignment 3

Design Document

> *Significant updates made during or after the implementation are indented and printed in italic*

Introduction

During this assignment we want to replace OS161's dumbvm with a working VM system that implements/handles both TLB and paged memory faults. With a working virtual memory system, we must also implement the sbrk() syscall to make the userland malloc() work. Throughout the design document we focus on the simplest working solution which may not show a good performance. We may later extend our implementation regarding to performance.
Our design document goes over every part of assignment 3. We will split assignment 3 into three steps. The first step will be to handle TLB faults correctly. This includes the implementation of the Coremap and the handling of TLB faults. At this point in time we don't support malloc yet. In step 2 we implement sbrk. Step 3 is devoted to the implementation of pages. Since functionality of the single parts of the kernel will be either be the same for or differ for each step, we will indicate the changes between each steps throughout the document.

Overview

TLB

When a virtual memory address is accessed, the hardware looks it up in the TLB, trapping into the kernel if it doesn't see it. The two main parts we need to implement are handling TLB faults and ensuring proper initialization/invalidation of the TLB, especially on a context switch. For simplicity, TLB entries are replaced completely randomly, and vm_tlbshootdown will just call vm_tlbshootdown_all. The TLB will flush all entries on context switch using vm_tlbshootdown_all.

Coremap

Our coremap will consist of a bitmap indicating which pages in memory are free or occupied. An additional bitmap will indicate if a specific page is a kernel page or not. kernel pages can not be evicted from memory. Furthermore we will maintain a reverse lookup table which translates from pages in physical memory to page table entries.

Instead of using a bitmap we used a struct array which stored not only if the page in physical memory was free but also the translation table.

Paging

For simplicity we will write every page back to disk on every swap, even it is not dirty.

This results in a super slow malloctest 3.

In Depth Analysis

Dimensions

- A page is 4KB large
- A page table fits in one page
- The disk should be 10 to 20 times larger than physical memory. Thats why we use 4 extra bits for addresses on disk, which means that we have 16 times more space available.

vm_fault()

This is a machine-dependent function which is called by mips_trap on a TLB miss. If the TLB is full, it should run the replacement algorithm to decide which index to replace with tlb_write(), but for now, we will just always call tlb_random() to replace a random entry(). There are three different TLB exceptions, which are EX_MOD, EX_TLBL, EX_TLBS. EX_MOD is called when a write is attempted on a TLB entry with the dirty bit unset. It then sets the TLB dirty bit and sets the page's dirty bit. EX_TLBL is called when the TLB misses on a read, and EX_TLBS is called when the TLB misses on a write. Both first walk the page table to find the virtual page, and then check to see if it is in physical memory. If so, it calls tlb_random() with those virtual and physical page indices. If it is not, it fetches the page from disk and calls tlb_random() with the virtual index and the new physical page index. The main difference between EX_TLBL and EX_TLBS is that EX_TLBL should set the dirty bit.

Virtual Address

The virtual address we are going to use is setup as follows

10 bit	10 bit	12 bit
Page 1	Page 2	Offset

Page Table Entry

The Page Table Entry is setup as follows:

24 bit	1 bit	1 bit	1 bit	5 bit
Add Page on physical memory or on disk	If it is on disk or in physical memory	If it is allocated at all	Dirty - not used yet, would be used for swapping the pages efficiently	unused

Step 1&2

VM_FAULT_READ, WRITE
Find the segment for the current address. If there is now segment, throw a segfault. And kill current thread.
Get the first 10 bit of the vaddr. See if it is a valid entry in the first level page table.

- if first level entry is not valid
    - allocate a second level page table
        - allocate a new kpage, leave the kernel bit
        - write zeros
        - set valid bit on level 1
        - set 1st level PTE address

- get the next 10 bits of the vaddr
- see if it is a valid entry in the second level page table
- if second level entry is not valid:
    - allocate new page using alloc_kpages
    - set second level page table entry to newly allocated page
    - set valid bit
    - set reverse lookup entry
    - unset kernel bit in coremap

- TLB write the physical address of the old or new page (and the vaddr) and set dirty bit (according to READ or WRITE)

VM_FAULT_READONLY
set dirty bit in page table entry, and set dirty bit in TLB

Step 3
Same as in previous step but we have to check on the second level entry if it is in physical memory or on disk.
If on disk

- allocate kpage
- copy from disk
- remove kernel bit
- zero page on disk
- set diskmap bit to free

vm_tlbshootdown[_all]()

This is a machine-dependent function which should remove the specified tlb entries, because it uses the mips-specific tlb functions. As a simplifying step, we can just call vm_tlbshootdown_all() in vm_tlbshootdown() since we technically remove the specified entries, it's just that we remove the rest as well. vm_tlb_shootdown_all should use tlb_write() on every index (0 to NUM_TLB-1?) with TLBHI_INVALID and TLBLO_INVALID to flush all entries. We should call vm_tlbshootdown_all() on a context switch (in thread_switch()?) to invalidate all entries. We need to acquire a spinlock so we don't get interrupted while clearing the TLB?

Address Space Structs

We introduce bit fields for easy access to the individual parts of the PTE.

```
struct page_table_entry{
      unsigned int index           : 24; // contains either the memory or disk
address
      unsigned int in_memory : 1; // indicated if in memory or on disk
      unsigned int valid           : 1; // indicates if valid, i.e. allocated
      unsigned int dirty            : 1; // currently unsused
      unsigned int                    : 5; // padding to align with 32 bit
}
```

```
struct segment_table_entry{
      unsigned int start :20;  // indicates the virtual page index where
                                     // the segment starts
      unsigned int end : 20;   // indicated the virtual page index of the page
                              // after the last page of this segment
      unsigned int read       : 1;
      unsigned int write      : 1;
      unsigned int execute : 1;
      unsigned int :21; // padding to fill 64 bits
}
```

Since we will only have segments we will allocate a static array for 4 segments.

extend address space

```
struct addrspace{
      …
      pointer to page table
      pointer to segment table
      bit ignorePermissions // this is needed for the initial write
                            // to the code segment
}
```

as_create()

- Allocate space for the first level page table
- Allocate space for the segment page table

as_copy()

- iterate over all pages
    - if in physical memory
        - acquire lock
        - set kernel bit
        - release lock
        - allocate new kernel page (kernel bit is set in coremap -> will not be swapped)
        - create page table entry
        - copy the page
        - link PTE to kernel page
        - unset the kernel bit in coremap for both
    - if on disk
        - allocate new kernel page (kernel bit is set)
        - we don't need to acquire a lock because we are the only one using that page

        - read from disk to the newly allocated kernel page (VOP_READ)
        - link PTE to kernel page
        - unset kernel bit on newly allocated page

as_destroy()

- acquire lock
- loop over page tables
    - if in phys mem
        - erase page (deadbeef),
        - delete/free coremap entry (also delete reverse lookup entry)
    - if on disk
        - erase page (deadbeef)
        - delete entry on disk map
    - free the pagetable from the coremap
- free first level page table
- release lock
- free segment table and any other structs

as_activate()

- clear TLB

as_deactivate()

- probably nothing

as_define_region()

- setup segment table (contains start and end addresses of segments)
- make sure regions don’t overlap

as_prepare_load()

- allow write to code segment

as_complete_load()

- forbid write to code segment

as_define_stack()

- call as_define_region with certain arguments
    - has to be readable and writable
- return userspace top

Coremap

Step 1&2
We will use a bitmap which manages the free pages in physical memory. Also we will have an array of pointer: each occupied page of physical memory points to a page table entry.

Step 3
Add another bitmap which defines if a page is a kernel or user page.
Keep track of the number of free pages

> *Since we can not allocate memory with kmalloc at the point of initializing the coremap we can not use a bitmap. Thus, we will implement our coremap entries by ourselves.*

Get kpages, free kpages

For simplicity npages is always one. Add an KASSERT for that.

Get kpages

Step 1&2

- acquire lock
    - if memory available
        - find it and allocate it
            - create page table entry and add it at the appropriate position
            - set it to used in coremap, set kernel bit
    - if no memory available
        - release lock
        - throw error
    - release lock

Step 3

- acquire lock

        - if memory available, see step 1&2
        - if no memory available
            - find page to evict by random
            - find free space on disk
                - acquire lock
                    - loop over bitmap
                    - set bit as used
                - release lock
            - copy page to disk
            - set PTE to disk address
            - set location bit (PTE)
            - set diskmap bit
            - deadbeef the memory page
- release lock

Free kpages

- acquire lock (probably not necessary in this case but better to be sure that nothing bad happens)
- KASSERT that address is within kernel AS
- deadbeef the page
- release it in coremap
- release lock

sbrk()

Increment segment size. Add the rounded, requested amount to the data/bss (3rd element) segment's end

> *We only increment the heap segment size when we have enough space on the swap disk.*

Risk Analysis

Since we don't focus on performance but more on reliability, the system may be pretty slow. To improve the performance issue we plan to take the dirty bit into account so that only changes will be stored. Pages which were read but never changed will be discarded.
We may have not enough locks and lock to coarsely which may result in deadlocks. A finer granularity of locking could be achieved by introducing locks for each page table.

Time Estimation

vm_fault(): 7 h
TLB shootdown: 0.5 h
as functions: 7 h
coremap impl.: 5 h
Get kpages, free kpages: 5 h
sbrk(): 1 h
reverse lookup: 1.5 h
diskmap impl: 2 h
swapping functions: 7 h
testing and debugging: 10 h

Total: 46 h

Timeline

When	What	Who
Mon, November 24th	Coremap (bootstrap and interface)	Winfried / Felix
Mon, November 24th	get_kpages, free_kpages	Felix / Winfried
Mon, November 24th	TLB shootdown	Dustin
Tue, November 25th	Address Space Functions	Dustin
Wed, November 26th	Implement VM_Fault for Step 1	All
Wed, November 26th	Finalize Step 1	All
Wed, November 26th	Part 1	
Sat, November 29th	Implement resizing of segments	Felix
Sun, November 30th	sbrk	Felix
Mon, December 1st	Finalize Step 2	All
Sat, November 29th	Diskmap	Winfried

Wed, December 3rd	Add kernel bitmap & modify as_copy	Winfried
Mon, December 1st	Reverse Lookup	Dustin
Wed, December 3rd	Implement swapping functions (copying from disk to memory and vice versa)	All
Sat, December 6th	Finalize Step 3	All
Mon, December 8th	Submission	

When	What	Who
Mon, November 24th	Coremap (bootstrap and interface)	Winfried / Felix
Mon, November 24th	get_kpages, free_kpages	Felix / Winfried
Mon, November 24th	TLB shootdown	Dustin
Tue, November 25th	Address Space Functions	Dustin
Wed, November 26th	Implement VM_Fault for Step 1	All
Wed, November 26th	Finalize Step 1	All
Wed, November 26th	Part 1	
Sat, November 29th	Implement resizing of segments	Felix
Sun, November 30th	sbrk	Felix
	stack	Dustin
Mon, December 1st	Finalize Step 2	All
Sat, November 29th	Diskmap	Winfried
Wed, December 3rd	modify as_copy to set the reverse_lookup	Dustin
Mon, December 1st	modify vm_fault to set the	Dustin

	reverse_lookup	
Wed, December 3rd	Implement swapping functions (copying from disk to memory and vice versa)	All
Sat, December 6th	Finalize Step 3	All
Mon, December 8th	Submission	

Questions

TLB Questions

1) TLB situations:
   a) TLB miss, page fault - The virtual address is neither in the TLB or physical memory.
   b) TLB miss, no page fault - The virtual address is not in the TLB, but is still in physical memory.
   c) TLB hit, page fault - Shouldn't happen, the TLB is a mapping from a virtual address to an address in physical memory, so if a virtual address is in the TLB but not in physical memory it must be invalid as it should've been removed from the TLB when it was switched out.
   d) TLB hit, no page fault - The virtual address is in the TLB and in memory, so the mapping is valid and it just returns the physical address mapped in the TLB.
2) The first thing it loads is the user code, which causes a TLB miss and then page fault as it loads the code into a frame. When a virtual address is accessed that isn't in that code's frame, it causes a TLB miss and page fault as it loads the, for example, stack into physical memory and replaces the code. It then returns execution to the user program, but has to load the code from memory again, and so replaces the previous memory it needs, so gets into an infinite loop.
3) Load word (lw) loads a word into a register from the specified address. The zero register is in this example the base address where to read from and the offset is simply the position wrt the base address. The MIPS registers are 32 bit long. In our case the zero register expresses the address 0x0000 0000. That means the command tries to load a word starting from address 0x0000 0120 into $3. Since we access the address 0x0000 0120 we can have :
   a) TLB miss with page fault (EX_TLBL exception), which means that the address is not in the page table

b) TLB miss with no page fault (EX_TLBL exception), which means that the address is not in the TLB but in the physical memory
c) It can also happen that the address is not valid/accessible (depending on our design/segment regions), which means that we have an addressing error (segmentation fault) .

Malloc Questions

Question 1

1) How often is sbrk called in malloc(10):
    - :171
    - probably not on :188
    - :267

   sbrk is called 2 times.

2) Size it is rounded up to an integral number of blocks. A block has 8 bytes and we are allocating 10 bytes. This results in 2 blocks (16 bytes). To store bookkeeping information we allocate the rounded up value of size + another 8 bytes. Since our implementation of sbrk allocated page wise, i.e. 4KB, malloc will fill these 4KB with blocks of 24 bytes. Since malloc is called only 10 times, which means we have in total 240 bytes, it will still be in the same page.

   This results in finish-start = 4096.

Question 2

For each element in the res array, malloc() allocated 3 blocks. Two blocks (16 Bytes) for the data part + one block (8 Bytes) for bookkeeping. That is 24 Byte per free(res[n]). Since we call free() six times (2 blocks of 3 * free()), we have two blocks of 3*24 = 72 Bytes free memory. malloc(60) needs 64 Bytes (rounded up) for the data part + 8 Bytes for bookkeeping. In total that are 72 Bytes. But since we have two free blocks of 72 Bytes, malloc() will not call sbrk().

Question 3

"__" is used to avoid conflicts with user programs, that is to differentiate between userland functions and syscalls or internal libc functions.

Question 4

Malloc rounds up the amount of bytes to allocate by the next multiple of 8 bytes.