Design Document for Assignment 3

OS 451

Sandra Anderson, Riley Wilk, John Buscher

Overview/Implementation Breakdown:

TLB Handling (Riley)

Exceptions/ Misses:

When a TLB miss occurs, it will call on the page table to either immediately return a translation for the page (if the page is already in memory) or to bring it up from disk. Whenever a translation is brought into the TLB, its pid field will be set with the pid of the current process. This allows for these pages to be kept in the TLB upon a context switch and correctly kept as valid if another context switch back to the original process occurs.

Shootdown function:

When a page is updated such that the current translation is no longer valid, any TLBs using the old translation must be informed. This can be done by relaying information up from the core map level (which must see the change) - the core map can inform each page table with the page, and each page table can in turn inform its corresponding TLB.

The shootdown function takes a (machine-dependent) vaddr, and we probe the TLB, and if we find it in the TLB, we replace it with a kernel-space bogus value, higher than kernel space pointers actually in use.

Paging (discuss the process and use of data structures) (Sandra):

We need three functions to implement demand paging: page_allocate, page_swap, page_destroy.

Allocate is called by as_create and sbrk, and it allocates a new page, filling in the page table and the disk swap, and then swaps in the new page.

Swap evicts one page, writing it to disk, in order to bring another page into memory. It is called on a page miss, where the page table walk found that the page was non-resident. Then, it selects a page frame to evict, invalidates it in the page table, tlb shootdown, and then flushes any changes out to disk. Then the requested page is brought in from disk, and the core map is updated so that it reflects the new resident page. Then the new page table is updated, and the

tlb is as well. Then, the faulting user-land instruction (which triggered a tlb exception) is restarted.

Destroy is called on kfree or as_destroy, and it destroys a page within a process, invalidating it in the page table, the core map, and out on disk. This ensures that all memory is correctly freed, so that our OS can continue indefinitely.

Data structures:

Page table(Riley):

The page table can be implemented as a multilevel flat array mapping from virtual address to an information block containing the physical address and some metadata. The page table will also maintain some top-level metadata, including a lock, and valid bits for each subtable and page table entry. When a page table lookup occurs, each level of the page table will use a certain segment of the virtual address as its key, returning the next table in which to look, until reaching the deepest level, at which point the information block with the physical address will be returned if it is in the page table (and possibly some metadata will be updated). If the page is in the page table, the appropriate translation can be immediately sent to the TLB. If the page is not in the page table, the page table will call on the core map to add a new mapping. The new mapping will then be added to the page table, and will in turn be returned for the TLB to use.

Originally, we had thought to use a hashtable. However, we found that we needed to do a page table lookup in the case where kmalloc was evicting a userspace page, and that meant that the lookup function could not use kmalloc. This turned out to be incompatible with using a hashtable, so instead we used a two-level flat table. Each process has one master table, 1 page in size, and some number of subtables, each also 1 page in size. This covers the entire 32-bit address space. If we restricted to USERSPACE values, we could have reduced the master tables to only half a page in size, but we ran out of time.

Disk swap(John):

Swapping will require three main functions for its implementation. Swapping out, swapping in, and evicting a page. Evicting will involve changing the state of the page table entry. To write the page's content to disk, we will use a swapout function. Swapout should also mark the dirty bit to its clean state. Finally there is swapping in, which reads a page from disk, and puts it in the page table.

The pages swapped to disk will be stored on a single swap file. One swap file for all processes, which is initialized to be an entire 5MB disk. As a result, a lock will need to be used whenever reading or writing to this file takes place. VFS functions such as VOP_READ and VOP_WRITE will be used to read and write to this file.

This swap file will also need a bitmap associated with it, also protected by a lock. This hash table will store each page's location in the swap file. This table will map an integer index to the location in the file.
While we ought to zero a disk page before handing it to the process, in fact we do not do so yet.

Core Map(Sandra):

The core map is an array data structure, containing a descriptive struct for every page frame in memory. It keeps track of (a) which process(es) have the page allocated, (b) whether the page has been written, (c) whether the page may be evicted.

For simplicity's sake, we will simply declare that a page frame can be owned by one or zero processes. Then we have 15 bits for the PID, 1 bit for dirty, 1 bit for evictable, and 1 bit for empty. If empty, then everything else should be set to 0. This means that we need only a single 32-bit integer per page frame. Anything allocated by the kernel is default non-evictable, and everything allocated in userland is evictable. Dirty indicates whether or not we need to write the page to disk during the eviction, or if it is unchanged.

The core map keeps track of the pid and vaddr of a page frame, as well as a few flags. Swappable indicates whether or not the frame can be swapped out (equivalent to whether it is in userspace), and multi indicates whether or not the frame was a later page in a multi-page kmalloc call, so that we can free all of the pages together on the corresponding kfree call. Request_free and Request_destroy help the core map to interface with pages that are being swapped out while the process attempts to destroy them.

We use a spinlock to control the bitmaps for which pages are free and/or swappable. The swappable bitmap serves as a pseudo-lock per frame, as non-swappable pages are considered to be owned by the kernel. While a page frame is being altered (whether allocation, swap, or free), we set the swappable bit high, and then no other thread will attempt to swap them as well.

In the case where all page frames in memory are held by the kernel, we should wait until one is freed. However, we ran out of time before being able to implement this feature fully.

Address space functions(John):

Each of the following Address space functions will need to be modified from their current dumb vm states, to actually work with our page table.

as_create()

We will malloc space for an addrspace struct, and initializes its different fields. Most importantly, a page table will need to be created. We do not start out by creating any actual pages, but the master level page table is initialized.

as_copy()

Will need to lock the page table. Then, we copy all pages, and set up the new page table to point to those copies, with precisely the same permissions.

as_activate()

Simply shoot down all tlb entries.

as_destroy()

In a two part process, first we free all pages from memory and disk. Because they might be in the process of swapping out, in some cases the control flow to free them might pass to another thread. We wait for all pages to be freed, and then we destroy the page table structure.

as_define_region()

We instantiate pages within the region, giving them the permissions provided.

as_prepare_load()

As_prepare_load temporarily suspends the write permissions across the address space, so that executables can be safely loaded.

as_complete_load()

As complete load will change the permissions back to their original value..

as_define_stack()

Will just set the stack pointer = to the global stack variable, and instantiate a few pages for the heap.

sbrk(intptr_t amount)

sbrk() will adjust the breakpoint of the heap by the amount passed in. To keep implementation simple, amounts that result in page-aligned address will be accepted, and amounts less than 0 will be rejected and error will be returned. It will be used by malloc when more heap space is needed.

Risk Analysis:

We have broken the assignment up into modules which are assigned to different group members. Each group member is responsible for their own risk estimates, which we use to equalize the load between members. Best case times are meant to indicate a zero-bug initial implementation. Average case represents a few difficult bugs, but no fatal design flaws. Worst case indicates that the initial implementation was flawed to the point that everything had to be re-started with a better design.

Since the assignment proved harder than expected, and some group members were able to contribute less than they had originally envisioned, the code is held together with quickly-written hacks, and many supposedly-essential features are left unimplemented.

Riley

General TLB handling

The basic implementation should be fairly straightforward, but both the page table and the core map rely on proper interoperation, so this might require later revisions. A simple way to save time here would be to use a trivial eviction algorithm.

Best case: 3 hours
Average case: 6 hours
Worst case: 30 hours

Page table and TLB shootdown

Again, much of the complexity here comes from proper communication both up (to the TLBs) and down (to the core map) so some time must be allowed for additional tinkering as each piece is put into place. As with the TLB, a simple eviction algorithm could save time if necessary.

Best case: 10 hours
Average case: 18 hours
Worst case: 80 hours

Sandra

Core Map:

This is a relatively simple data structure, but it is responsible for communicating up to per-process page tables and down to disk, without ever deadlocking or evicting essential instructions or data. We start with a simple eviction algorithm, to save time.

Best case: 8 hours
Average case: 20 hours
Worst case: 80 hours

John

Disk Swapping:

Disk swapping will require special care in making sure that the swap file is only held by one process at a time, and that it accurately saves and returns page tables

Best case: 10 hours
Average case: 20hours
Worst case: 80 hours

Address space functions:
This is a tough one to gauge the time, but most of the functions should take a few minutes to implement, while others (like copy) that depend on the structure of the page table have a high variance on the complexity, and could take some time to implement.

Best case: 5 hours
Average case: 12 hours
Worst case: 40 hours

Code Reading Questions:

Problem 1
Assuming that a user program just accessed a piece of data at (virtual) address X, describe the conditions under which each of the following can arise. If the situation cannot happen, answer "impossible" and explain why it cannot occur.

- TLB miss, page fault
    - A non-resident page which must be swapped in from disk
- TLB miss, no page fault

    - A page in memory which was not present in the TLB; must be brought into the TLB
- TLB hit, page fault
    - Impossible; if the page is in the TLB, it must be a resident page
- TLB hit, no page fault
    - The (hopefully) common case, in which the TLB functions as desired, and the address translation is handled directly by the TLB, without any page walk

Problem 2

A friend of yours who foolishly decided not to take CSE 451, but who (inexplicably) 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.)

The tlb can only hold one entry. Therefore, the requested page is brought into the single tlb entry. Then, the system would restart the instruction. However, the instruction lives on a page in memory, which must be brought into the tlb before it can be used. So when the instruction is restarted, the tlb entry is for the page of that instruction, not the required memory, and the same exception is re-triggered.

Problem 3

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.

```
 # load word from $0 (zero register), offset 0x120, into register $3
  lw  $3, 0x0120($0)

TLB miss, Page fault, and Address error on load.
```

Problem 4

Consider the following (useless) program:

```
/* This is bad code: it doesn't do any error-checking */
#include <stdio.h>
int main (int argc, char **argv) {
    int i;
    void *start, *finish;
    void *res[10];
    start = sbrk(0);
    for (i = 0; i < 10; i++) {
         res[i] = malloc(10);
    }
    finish = sbrk(0);
    /* TWO */
    return 0;
```

```
}
```

How many times does the system call sbrk() get called from within malloc()? What is the value of (finish-start)?

Once when malloc initializes, and again when when malloc is called for the first page.

Problem 5

Suppose we insert the following code at location /* TWO */:

```
{
    void *x;
    free(res[8]); free(res[7]); free(res[6]);
    free(res[1]); free(res[3]); free(res[2]);
    x = malloc(60);  /* MARK */
}
```

Would malloc() call sbrk() when doing that last allocation at the marked line above? What can you say about x?

No, freeing all that space on the heap will save enough space on the heap for x.

Problem 6

It is conventional for libc internal functions and variables to be prefaced with "__". Why do you think this is so?

The main reason is to avoid name conflicts with functions that a user might write.

Problem 7

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?

The line of code following the comment ‘Round size up to an integral number of blocks’

Naming Convention:

```
function_names()
variable_names
```

```
This is to be the most consistent with existing OS161 code,
while still ensuring best readability. Functions and variables
will not be prefixed with type information, although OS161 does
so in places.
```