Jack Dent
Graham Lustiber

CS161, Assignment 3 Design Document

Contents

1) Data Structures

a. Core map

The core map keeps track of the which physical pages have been allocated to processes, and the virtual addresses associated with these physical pages in each process. The core map will live in the kernel's memory, and we will initialise a global kernel variable to points to it. Our core map will be a global array of cm_entry structs, one per page. Each struct will actually be an unsigned 64 bit value, used as follows:

```
struct cm_entry
{
      pid:15;           // process that owns this page
      l1_offset:10;
      l2_offset:10;
      swap_offset:25;   // index into disk in chunks equal to the size of a
                        // page when page is swapped out
      free:1;           // 0 if free, 1 if owned by a process
      dirty:1;          // 1 if page is dirty, 0 otherwise
      busy:1;           // synchronization, see below
      recent:1;         // For the LRU clock algorithm
};
```

While it consumes the maximum amount of space even if few pages are used by user processes, it allowed for O(1) indexing of entries and linear, sequential scanning when looking for a free page. Once this coremap is working, it could be made into a stack to achieve O(1) lookup for free pages.

We will never modify the core table directly, but instead use wrapper API functions, which will handle synchronisation, and various other aspects of checking and setting a core map entry.

Synchronization: in order to fit every entry into a 64 bit value and avoid having a lock per entry, we use a busy bit that is 1 when a thread has the entry "locked" and 0 otherwise. In order to check and change this bit's state, we have a global `cm_lock` lock.

b. Page Table and Page Table Entries

Our per-process page table will be a two level page table, which will live in kernel memory. The 1st level will be an array of 2^10 =1024 pointers to level 2 page tables, all

NULL initially. The avoids allocating space for level 2 page tables until they are needed. Since a pte is 32 bits, the same size as a pointer, they can be unsigned integers in the level 2 page table.

```
struct l1_pt {
      struct *l2_pt l2_pts[1024];
};

struct l2_pt {
      struct pte ptes[1024]
};

struct pte {
      phys_page:20;
      present:1;        // 1 if page is allocated for some process
      padding:10
};
```

Thus address translation uses the top 10 bits for the level 1 page table, the next 20 bits for the level 2 page table, and the lower 12 bits of the offset into the physical page.

This saves space versus using a single level page table, which would require 2^20 pointers to account for all possible pages referenced from a 2^32 bit virtual address. Instead, the two level system uses

Synchronization: each process's page table will have a lock to protect against another kernel thread or the write back daemon thread trying to access it simultaneously.

c. TLB pid bitmask

We will use a global bitmask, protected by a spinlock, to mark whether a current TLB pid is currently in use. When a process is evicted during a context switch, it should 'return' its TLB pid to the pool of available pids, so that the next process that is about to run has a tlbpid available.

d. Swap Space Bitmask

A global bitmap will keep track of free disk space for swapping. Each bit will correspond to a chunk of disk equal to the size of page. The size of this bitmap will correspond to the number of pages that can fit on the swap disk, which will be a configurable constant in the config file.

Synchronization: Access to this bitmap, and thus to swapping memory, will be controlled by a lock.

2) TLB Design

We need to implement the following faults: EX_MOD (write to read only page), EX_TLBL (entryhi miss on load), and EX_TLBS (entryhi miss on store). TLB contains 64 entries, each 64 bits in size. See http://www.eecs.harvard.edu/~cs161/resources/vahalia.pdf for a more detailed description of each

```
struct entryhi {
      vpn:20;     // VPN (virtual page number)
      pid:6;      // PID (takes values 0 through 63). NB we must assign
                  // processes a unique tlbpid; we do not use this bit
                     in our 'simple' policy
      padding:6;
}

struct entrylo {
      pfn:20;          // physical page number
      no_cache:1;      // if 1, page should not go through data or
                       // instruction caches
      global:1;        // if 1, ignore PID field. N.B. we do not use
                          this bit in our 'simple' policy
      invalid:1;       // if 1, invalid entry
      dirty:1;         // if 1, then the entry is write-protected
      padding:8;
}
```

a. API

OS161 provides us with four C-level functions to interact with the TLB:

- TLB_Write(): Write to a specified TLB entry
- TLB_Read(): Read from a specified TLB entry
- TLB_Probe(): Search TLB to see if it contains a match for a given virtual frame number

- TLB_Random(): Write to a random TLB entry. Note that TLB_Random() never selects 8 of the 64 TLB entries, so you may want to use TLB_Write() and random()

We add a TLB_Evict() function, which takes an index, and sets the invalid flag on the entrylo struct back into the TLB, keeping all the other fields the same. TLB_Evict will call both TLB_Read() and TLB_Write(). We then throw an EX_TLBL exception whenever we attempt to retrieve a TLB entry which has the invalid bit set, but we do not throw an EX_TLBS exception whenever we try to write one (since we may be trying to overwrite it with a new entry!).

b. Dirty pages and Kernel level EX_MOD exception handling

We handle EX_MOD TLB exceptions by setting the page's dirty bit to 1 in its core map entry, updating the TLB dirty bit to also be 1, and reattempting the write (rewinding the program counter by 4 bits).

Since all memory in OS161 that a process has access to is both readable and writeable, we will use the dirty bit in the TLB entry to know when a page is dirty. When the entry is initially loaded in, the bit is set to 0. Thus, the first time a user process tries to write to it, it throws a page fault. The kernel then marks this page as dirty in the core map and sets the bit to 1 to avoid causing further page faults when writing. When the page is saved to disk, either through the writeback daemon or the eviction policy, the bit is set back to 0.

c. Kernel level EX_TLBL and EX_TLBS exception handling

If we get a miss on load or store, there are 3 possible scenarios when we check the process's page table:

1) We and find that the process is trying to write to an address it doesn't own. In this case, we kill the thread and process.
2) We find that this address is valid for the process, but it isn't loaded into the TLB. So, we evict some entry in the TLB, load in the page of the faulting address, roll back the program counter by 4 bytes, and continue the process.
3) We find that this address is valid, but the page is swapped to disk. If so, find the disk offset from the core page and load it into main memory, using the eviction policy described below if necessary. Roll back the program counter by 4 bytes and continue the process.

d. TLB eviction policy

We will place TLB entries into the TLB starting at index 0 and work up to NUM_TLB - 1. At this point, putting in a new entry requires an eviction, so we evict the oldest entry at index 0, then the next oldest at index 1, etc. This oldest index will be kept track of by an integer per cpu.

e. 'Simple' thread switching policy

We use a relatively simple, albeit inefficient, policy for handling thread switches: whenever we schedule a new thread, we call TLB_Evict() on every entry in the TLB. This approach assumes that we are:

- Ignoring *tlbpids*
- Ignoring the global bit in entrylo

We save a processes' TLB state a field on the thread struct, so it can be recovered when the thread gets rescheduled.

3) Paging

a. Eviction

We will use a least recently used (LRU) policy when deciding which page to evict. When a page fault occurs and there is no space in main memory, the clock algorithm will cycle through the pages in physical memory until an unused page is found or a page with a `recent` value of 0 is found.

The clock algorithm will work as described in class (we iterate over the core map to find the first free page, or the first page whose recent bit is set to 0; if we loop back around to the starting position without having found any such entries, then we evict the page we started with and move the clock hand to the next position). There will be a global `unsigned clock_hand` to keep track of the current position, an `int evict_page()` function to perform the sweep, and a spinlock `clock_hand_lk` to protect updates to `clock_hand`, between multiple cores.

We do not need to send a cross-processor interrupt whenever we free a page, since all processes are single threaded and there is neither global memory nor shared memory,

which means the page we are evicting will never be accessed by any other core at the same time.

b. Page Fault Handling

Assume we already hold the lock of the current process's page table.

Page not in main memory:

1) Pick a free page using the LRU clock algorithm. Try to acquire the lock on that coremap entry. If taken, continue calling the LRU clock algorithm.
2) If the page is unmapped, update the process's page table with this page linked to the virtual address it faulted on. Copy in the data from disk.
3) If the page is mapped, we have to evict it, which means modifying the page table of the other process's page table (assuming it is a different process). This means releasing the lock on the current page table and reacquiring it and the other page table lock in order of pid's to avoid deadlock. If the page already belongs to the current process, this can be ignored. Mark the evicted page as on disk in this other page table.
4) If the page we're evicting is dirty, we write it back to disk.
5) Call tlb_shootdown() on the evicted page to make it invalid in the TLB cache.
6) Read in the page from disk using the offset saved in the page table entry.
7) Update the core map entry with the new pid and address space.
8) Release the core map entry lock

c. Writeback Daemon

Per the assignment specs, we will also have one thread dedicated to crawling through the pages and writing them to the backing store if they are writeable so that page evictions rarely have to flush the data. This thread will be synchronized by the the lock associated with each coremap entry to avoid conflict with page fault handling. This thread will only be used when memory pressure reaches above a certain level, say, 75% of the pages. This is because if there are just a couple processes using very few pages, we waste cpu time and data transfer time writing pages to disk when it is unlikely they will ever need to be evicted. At the other extreme, if the daemon is only used when all pages are taken, then we lose the advantage of preemptively flushing, as now the current process must wait for the data to be written to disk before it can get the new page.

4) kmalloc and sbrk()

sbrk will need to adjust the end of heap_end in the process's address space to account for requested size, assuming it is within the given size of user heap.

5) Eviction Policy Analysis

Statistics we can keep track of for TLB eviction include, for a given process load:

- Number of TLB evictions
- Number of TLB misses on stores or loads
- Overall runtime

With this data, we could tune the oldest-first algorithm to perhaps take into account recency of use, or some element of randomness to it doesn't get caught thrashing.

Statistics we can keep track of for page eviction include, for a given process load:

- Number clean pages
- Number dirty pages
- Number of page evictions
- Number of page swaps handled by the eviction policy vs. the write back daemon
- Number of swaps to disk
- Number of swaps from disk
- Overall runtime

With this, we could tune how often or for how long the write back daemon runs, whether to sweep for more or less than one cycle before sweeping a recently used page, or whether to synchronize it with the scheduling queue somehow.

6) Integration and Bootstrapping

The global core map, disk swap bitmap, and bitmap lock will live in vm.h and be initialized in vm_bootstrap(). We will use 'ram_stealmem()' to get memory for our fundamental data structures (the core map, for example).

The page table struct will be added to the address space struct, along with the lock on it.

TLB index counter for eviction will be in the cpu struct

7) Plan of Action

1. Implement core map
2. Implement page tables
3. Implement eviction policy for pages
4. Implement TLB faults
5. Implement eviction policy for TLB