OS Internals – Progetti su OS161 (Cabodi)

Dopo alcune brevi informazioni comuni ai progetti, si descrivono tre progetti (numerati c1, c2, c3) in ambito OS161.

Regole specifiche per i progetti OS161 (Cabodi)

I testi dei (tre) progetti sono scritti in inglese, in quanto sono comuni al corso in italiano e al corso in inglese. Qualora uno studente abbia difficoltà a capire parte delle specifiche, mi contatti direttamente. I tre progetti sono caratterizzati da un certo grado di libertà nelle scelte progettuali, quali ad esempio quanto realizzare, come effettuare i test, come gestire eventuali errori, ecc.

Ogni gruppo potrà, se lo ritiene, sottomettere un breve "piano di lavoro" (un breve documento da una a tre pagine) che descriva in modo sommario le strutture dati e le funzioni che si intendono realizzare, come pure le strategie che si intendono adottare per risolvere il problema. Visto il piano di lavoro, si potranno eventualmente concordare specifiche dettagliate e/o eventuale supporto aggiuntivo (quale ad esempio implementazione totale o parziale di system call aggiuntive, fornite dal docente)

Tutti i progetti sono identificati da un codice (ad es. project c1: PAGING). Detto questo codice CODE, tutte le parti di codice aggiunto a OS161 per il progetto vanno abilitate con #if OPT_CODE. Si debbono seguire le line guida per gestire le opzioni, che prevedono, tra l'altro, l'inclusione di file header quali "opt-code.h".

Un gruppo può richiedere un progetto semplificato/ridotto, il che significa che possiamo concordare una serie di requisiti rilassati/ridotti. Il progetto semplificato richiede meno tempo di lavoro (circa la metà del progetto standard) e porta a un massimo di 3 punti (invece di 6). Per richiedere un progetto ridotto, scrivere su slack, canale os161_progetti.

Criteri di valutazione

Il progetto sarà valutato sulla base di: (a) un breve report che descrive il lavoro svolto, (b) il codice sorgente del kernel, (c) un colloquio orale con tutti i membri del gruppo.Una chiara organizzazione del carico di lavoro all'interno del gruppo, con ogni membro del gruppo responsabile di un dato sotto-problema (una sotto-attività, un modulo, una struttura dati) sarà un vantaggio in termini di valutazione. Quindi questo tip di organizzazione (una buona/chiara organizzazione del team) è meglio, ad esempio, di "abbiamo sempre lavorato insieme su un desktop condiviso". Un altro aspetto importante da considerare è un insieme ben definito di test, volti a dimostrare che l'implementazione è corretta (quindi non solo eseguire "p testbin/palin"): ciò significa eseguire programmi utente e/o comandi del kernel esistenti oppure modificati ad hoc.

G.Cabodi

Project c1: Virtual Memory with Demand Paging (PAGING)

Project's summary

The project aims at expanding the memory management (dumbvm) module, by fully replacing it with a more powerful virtual memory manager based on process page tables. The project also requires working on the TLB.

Required Background and Working Environment

Lab 2 and basic knowledge of dumbvm are needed (see lessons os161-overview and os161-memory). Silberschatz chapters 9 and 10 are also a pre-requisite.

Problem Definition

The project goal is to replace dumbvm with a new virtual-memory system that relaxes some (not all) of dumbvm's limitations.
The new system will implement demand paging (with a page table) with page replacement, according to the following requirements:

- New TLB support is needed, by implementing a replacement policy for the TLB, so that the kernel will not crash if the TLB fills up.
- On-demand loading of pages: this will allow programs that have address spaces larger than physical memory to run, provided that they do not touch more pages than will fit in physical memory.
- In addition, page replacement (based on victim selection) is needed, so that a new frame can be found when no more free frames are available.
- Different page table policies can be implemented: e.g. per process page table or Inverted PT, victim selection policies, free frame management, etc. The choice can be discussed and deferred to a later moment.

TLB Management

In the System/161 machine, each TLB entry includes a 20-bit virtual page number and a 20-bit physical-page number as well as the following five fields:

- global (1 bit): If set, ignore the pid bits in the TLB.
- valid (1 bit): When the valid bit is set, the TLB entry is supposed to contain a valid translation. This implies that the virtual page is present in physical memory. A TLB miss exception (EX TLBL or EX TLBS) occurs when no valid TLB entry that maps the required virtual page is present in the TLB.
- dirty (1 bit): In class, we used the term "dirty bit" (or "modify bit") to refer to a bit that is set by the MMU to indicate that a page has been modified. OS/161's "dirty" bit is not like this—it indicates whether it is possible to modify a particular page. OS/161 can clear this bit to indicate that a page is read-only, and to force the MMU to generate an EX MOD exception if there is an attempt to write to

the page.
- nocache (1 bit): Unused in System/161. In a real processor, indicates that the hardware cache will be disabled when accessing this page.
- pid (6 bits): A context or address-space ID that can be used to allow entries to remain in the TLB after a context switch.

In OS/161, the global and pid fields are unused. This means that all of the valid entries in the TLB should describe pages in the address space of the currently running process and the contents of the TLB should be invalidated when there is a context switch. In the dumbvm system, TLB invalidation is accomplished by the as activate function, which invalidates all of the TLB entries. OS/161's context-switch code calls as activate after every context switch (as long as the newly running thread has an address space). If you preserve this functionality in as activate in your new VM system, you will have taken care of TLB invalidation.

For this project, you are expected to write code to manage the TLB. When a TLB miss occurs, OS/161's exception handler should load an appropriate entry into the TLB. If there is free space in the TLB, the new entry should go into free space. Otherwise, OS/161 should choose a TLB entry to evict and evict it to make room for the new entry. As described above, OS/161 should also take care to ensure that all TLB entries refer to the currently running process.

Round-Robin TLB Replacement

You must implement a very simple (but dumb) round-robin TLB replacement policy. This is like first-in- first-out except that we do not actually worry about when each page was replaced and the algorithm works as follows:

```
int tlb_get_rr_victim(void) {
  int victim;
  static unsigned int next_victim = 0;
  victim = next_victim;
  next_victim = (next_victim + 1) % NUM_TLB;
  return victim;
}
```

Read-Only Text Segment

In the dumbvm system, all three address-space segments (text, data, and stack) are both readable and writable by the application. For this assignment, you should change this so that each application's text segment is read-only. Your kernel should set up TLB entries so that any attempt by an application to modify its text section will cause the MIPS MMU to generate a read-only memory exception (VM_FAULT_READONLY). If such an exception occurs, your kernel should terminate the process that attempted to modify its text segment. Your kernel should not crash.

On-Demand Page Loading

Currently, when OS/161 loads a new program into an address space using runprogram, it pre-allocates physical frames for all of the program's virtual pages, and it pre-loads all of the pages into physical memory.

For this assignment, you are required to change this so that physical frames are allocated on-demand and virtual pages are loaded on demand. "On demand" means that that the page should be loaded (and physical space should be allocated for it) the first time that the application tries to use (read or write) that page. Pages that are never used by an application should never be loaded into memory and should not consume a physical frame.

In order to do this, your kernel will need to have some means of keeping track of which parts of physical memory are in use and which parts can be allocated to hold newly-loaded virtual pages. Your kernel will also need a way to keep track of which pages from each address space have been loaded into physical memory and where in physical memory they have been loaded.

Since a program's pages will not be pre-loaded into physical memory when the program starts running, and since the TLB only maps pages that are in memory, the program will generate TLB miss exceptions as it tries to access its virtual pages. Here is a high-level description of what the OS/161 kernel must do when the MMU generates a TLB miss exception for a particular page:

- Determine whether the page is already in memory.
- If it is already in memory, load an appropriate entry into the TLB (replacing an existing TLB entry if necessary) and then return from the exception.
- If it is not already in memory, then
    - Allocate a place in physical memory to store the page.
    - Load the page, using information from the program's ELF file to do so.
    - Update OS/161's information about this address space.
    - Load an appropriate entry into the TLB (replacing an existing TLB entry if necessary) and return from the exception.

Until you implement page replacement, you will not be able to run applications that touch more pages than will fit into physical memory, but you should be able to run large programs provided that those programs do not touch more pages than will fit.

Page Replacement

You should implement a page replacement policy of your choosing. A very simple (even poor) algorithm that works is preferred to a more complex algorithm that you can't get to work. Do not worry about implementing techniques to avoid or control thrashing. Pages that need to be written to disk should be written to a file named SWAPFILE. This file will be limited to 9 MB (i.e., 9 * 1024 * 1024 bytes). If at run time more than 9 MB of swap space is required your kernel should call

panic("Out of swap space"). Make the maximum size of the swap file easy to change at compile time, in case we need to change this requirement before final submissions.

Instrumentation

You should be tracking and printing several statistics related to the performance of the virtual memory sub-system (including TLB misses and TLB replacements) so be certain to implement this as described so we can easily examine and compare these statistics.

You can collect the following statistics:

- TLB Faults: The number of TLB misses that have occurred (not including faults that cause a program to crash).
- TLB Faults with Free: The number of TLB misses for which there was free space in the TLB to add the new TLB entry (i.e., no replacement is required).
- TLB Faults with Replace: The number of TLB misses for which there was no free space for the new TLB entry, so replacement was required.
- TLB Invalidations: The number of times the TLB was invalidated (this counts the number times the entire TLB is invalidated NOT the number of TLB entries invalidated)
- TLB Reloads: The number of TLB misses for pages that were already in memory.
- Page Faults (Zeroed): The number of TLB misses that required a new page to be zero-filled.
- Page Faults (Disk): The number of TLB misses that required a page to be loaded from disk.
- Page Faults from ELF: The number of page faults that require getting a page from the ELF file.
- Page Faults from Swapfile: The number of page faults that require getting a page from the swap file.
- Swapfile Writes: The number of page faults that require writing a page to the swap file.

Note that the sum of “TLB Faults with Free” and “TLB Faults with Replace” should be equal to “TLB Faults.” Also, the sum of “TLB Reloads,” “Page Faults (Disk),” and “Page Faults (Zeroed)” should be equal to “TLB Faults.” So this means that you should not count TLB faults that do not get handled (i.e., result in the program being killed). The code for printing out stats will print a warning if this these equalities do not hold. In addition the sum of “Page Faults from ELF” and “Page Faults from Swapfile” should be equal to “Page Faults (Disk)”.

When it is shut down (e.g., in vm_shutdown), your kernel should display the statistics it has gathered. The display should look like the example below.

Files/directories

You should begin with a careful review of the existing OS161 code with which you will be working. The rest of this section identifies some important files for you to consider.

In kern/vm

addrspace.c: The machine-independent part of your virtual-memory implementation (alternative to kern/arch/mips/vm/dumbvm.c) should go in this directory.

In kern/syscall

loadelf.c: This file contains the functions responsible for loading an ELF executable from the filesystem into virtual-memory space. You should already be familiar with this file from Lab 2. Since you will be implementing on-demand page loading, you will need to change the behaviour that is implemented here.

In kern/vm

kmalloc.c: This file contains implementations of kmalloc and kfree, to support dynamic memory allocation for the kernel. It should not be necessary for you to change the code in this file, but you do need to understand how the kernel's dynamic memory allocation works so that your physical-memory manager will interact properly with it.

In kern/include

addrspace.h: define the addrspace interface. You will need to make changes here, at least to define an appropriate addrspace structure.

vm.h: Some VM-related definitions, including prototypes for some key functions, such as vm_fault (the TLB miss handler) and alloc_kpages (used, among other places, in kmalloc).

In kern/arch/mips/vm

dumbvm.c: This file shpuld not be used (when disabling option dumbvm). However, you can use the code here (with improvements done in lab 2). This code also includes examples of how to do things like manipulate the TLB.

ram.c: This file includes functions that the kernel uses to manage physical memory (RAM) while the kernel is booting up, before the VM system has been initialized. Since your VM system will essentially be taking over management of physical memory, you need to understand how these functions work.

In kern/arch/mips/include

In this directory, the file tlb.h defines the functions that are used to manipulate the TLB. In addition, vm.h includes some macros and constants related to address translation on the MIPS. Note that this vm.h is different from the vm.h in kern/include.

You are free and will need to modify existing kernel code in addition you'll probably need some code to create and use some new abstractions. If you uses any or all of the following abstractions please place that code in the directory kern/vm using the following file names:

- coremap.c: keep track of free physical frames
- pt.c: page tables and page table entry manipulation go here
- segments.c: code for tracking and manipulating segments
- vm_tlb.c: code for manipulating the tlb (including replacement)
- swapfile.c: code for managing and manipulating the swapfile
- vmstats.c: code for tracking stats

If you need them, corresponding header files should be placed in os161-1.11/kern/include in files named: addrspace.h, coremap.h, pt.h, segments.h, vm_tlb.h, vmstats.h, and swapfile.h.

Possible variants of the project

The project can be taken in one of three variants.

C1.1) Per-process page table, where the problem of the "empty" region between the two segments and the stack should be addressed

C1.2) Inverted Page Table, with "some" solution in order to speed-up linear search (not necessarily a hash, though possible).

C1.3) TLB support for the "dirty" bit (modified page): it has to be addressed by software, as the modify bit is not handled automatically upon write operations. If not taking this option, "data" pages can be considered as dirty/modified even when not written.