OSM
G assignment 4

Tobias Hallundbæk Petersen (xtv657)
Ola Rønning (vdl761)
Nikolaj Høyer (ctl533)

March 10th, 2014

Contents

1 Task 1: TLB exception handling in Buenos

The TLB load and store exceptions behave in almost the same way. Hence we will just show the code for load below. First, we retrieve the state of the offending thread. Then we check if the 13th most significant bit of the `badvaddr` of this thread is set or not and capture the result in the `is_odd` variable - this checks if the page we wish to look up is odd or even.

We then look for a match between the `badvpn2` and `asid` fields of the offending thread and pages in the page table while also checking if the dirty bit is set. If a match is found, we move on to do a random TLB write. Otherwise we produce a kernel panic.

An important point is that we have chosen to throw a kernel panic both in the case of a kernel exception and a userland exception. We differentiate between the two by letting the TLB exception functions take a `kernelcall` argument which is either `0` (call from userland) or `1` (call from kernel). These values are hardcoded in the userland and kernel exception programs, like so:

```
    switch(exception) {
    case EXCEPTION_TLBM:
      tlb_modified_exception(0);
      break;
    case EXCEPTION_TLBL:
      tlb_load_exception(0);
      break;
    case EXCEPTION_TLBS:
      tlb_store_exception(0);
      break;
```

`proc/exception.c`

```
    switch(exception) {
    case EXCEPTION_TLBM:
      tlb_modified_exception(1);
      break;
    case EXCEPTION_TLBL:
      tlb_load_exception(1);
      break;
    case EXCEPTION_TLBS:
      tlb_store_exception(1);
      break;
```

`kernel/exception.c`

From a user standpoint this is clearly bad, since for example a userland

segfault would result in the kernel crashing, instead of doing something more sane - aborting the violating thread seems more reasonable. We see no apparent way of doing so, however, so we leave the code as it is. With our clear distinction between the two types of errors it is relatively simple to implement this in the future.

The code for the load exception part of `vm/tlb.c` is shown below.

```
void tlb_load_exception(int kernelcall)
{
  tlb_exception_state_t exn_state;
  thread_table_t* my_table;
  tlb_entry_t my_entry;
  int i;
  int found;
  int is_odd;
  my_table = thread_get_current_thread_entry();

  // The exception info is loaded.
  _tlb_get_exception_state(&exn_state);

  // As the 13th bit of our vaddr tells us if it is the even or odd page
  // we check whether this is set or not.
  is_odd = (exn_state.badvaddr & (4096)) != 0;

  found = 0;

  // We loop over all the pagetable entries and look for a matching page.
  for (i = 0; i < PAGETABLE_ENTRIES; i++) {
    my_entry = my_table -> pagetable -> entries[i];
    if (my_entry.VPN2 == exn_state.badvpn2 &&
        my_entry.ASID == exn_state.asid) {
      // We check whether the dirty bit is set for the odd or even page.
      if ((!my_entry.V0 && !is_odd) || (!my_entry.V1 && is_odd)) {
        found = 1;
        break;
      } else {
        break;
      }
    }
  }
  // If a page is not found we print the tlb debug, and do a kernel panic.
  if (!found) {
    if (kernelcall) {
      print_tlb_debug();
      KERNEL_PANIC("kernel TLB load exception");
    } else {
      print_tlb_debug();
      KERNEL_PANIC("userland TLB load exception");
    }
  }
  // If it is found we write the entry to a random place in the tlb.
```

```
  _tlb_write_random(&my_entry);
}
```

The TLB modified exception is much more simple than load or store - we know that an error has occurred and we just need to differentiate between a userland and a kernel type. The code is shown below.

```
void tlb_modified_exception(int kernelcall)
{
  if (kernelcall) {
    print_tlb_debug();
    KERNEL_PANIC("kernel TLB modify exception");
  } else {
    print_tlb_debug();
    KERNEL_PANIC("userland TLB modify exception");
  }
}
```

Since TLB exceptions are now handled properly, all calls to `tlb_fill` have been removed.

2 Task 2: Dynamic allocation for user processes

3 Task 3: Extended tests for TLB exceptions and user-space allocation