OSM
G assignment 4

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

March 10th, 2014

Contents

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.

```c
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 Dynamic allocation for user processes

2.1 Implement the system call `void* syscall_memlimit (void *heap end)`

We have added a new case to the system call switch statement in `proc/syscall.c`:

```
      V0 = syscall_sem_destroy((void *)A1);
      break;
    case SYSCALL_MEMLIMIT:
```

together with the function `syscall_memlimit`, also in `proc/syscall.c` (this could have been placed elsewhere, but for simplicity we have chosen to just place it in the syscall file):

```
void* syscall_memlimit(void *heap_end){
  thread_table_t* current_thread;
  process_table_t* current_process;
  int pagespan;
  int i;
  uint32_t phys_page;
  void* current_heap_end;
```

```
current_thread = thread_get_current_thread_entry();
current_process = process_get_current_process_entry();
current_heap_end = current_process -> heap_end;
current_heap_end = (current_heap_end + 4096) & 0xfffff000;
if (heap_end == NULL) {
  return current_heap_end;
}
if (heap_end < (void*) current_heap_end) {
  return NULL;
}
pagespan = ((uint32_t) ((heap_end - current_heap_end)) / PAGE_SIZE) + 1;

for (i = 0; i < pagespan; i++) {
  phys_page = pagepool_get_phys_page();
  KERNEL_ASSERT(phys_page != 0);
  vm_map(current_thread->pagetable, phys_page,
    (((uint32_t) current_heap_end) & PAGE_SIZE_MASK) + i*PAGE_SIZE, 1);
}
current_process -> heap_end = heap_end;
```

This function relies on the addition of the variable `int heap_end` to the `process_table_t` struct in `proc/process.h` - that is, for each PCB we add a pointer to where the heap of this particular PCB ends.

In general terms, the `syscall_memlimit` function works by first assessing how many pages we need in order to cover the span implied by the desired heap end - the `pagespan` variable. For each page we need to allocate, we retrieve a physical page and map this to a virtual page. We then set the heap end of the current process to the new heap end.

2.2 Implement two library functions `malloc` and `free` in the user-space library

All modifications in this section is done to `tests/lib.c`. We start by removing the fixed-size heap array, since each PCB now have its own heap indicator variable. The major change in the function `malloc` is checking whether there is overhead when allocating the pages. This overhead will occur if $size \% 4096 \neq 0$, if this occurs we create a block of free memory with the size of the overhead. This is then added to the singly linked list as a free block, making it usable when another `malloc` call is issued for a size where this free block fits. For the `free` function we have made no changes as this should function independent on the changes we made throughout Buenos.

```
/* Heap allocation. */
```

```
#ifdef PROVIDE_HEAP_ALLOCATOR

typedef struct free_block {
  size_t size;
  struct free_block *next;
} free_block_t;

static const size_t MIN_ALLOC_SIZE = sizeof(free_block_t);

free_block_t *free_list;

/* Initialise the heap - malloc et al won't work unless this is called
   first. */
void heap_init()
{
  free_list->size = HEAP_SIZE;
  free_list->next = NULL;
}


/* Return a block of at least size bytes, or NULL if no such block
   can be found.  */
void *malloc(unsigned int size) {
  free_block_t *block;
  free_block_t **prev_p; /* Previous link so we can remove an element */
  void* heap_ptr;
  int free_size;
  if (size == 0) {
    return NULL;
  }
  heap_ptr = syscall_memlimit(NULL);
  printf("heap_ptr = %d \n",(int) heap_ptr);
  /* Ensure block is big enough for bookkeeping. */
  size=MAX(MIN_ALLOC_SIZE,size);
  /* Word-align */
  if (size % 4 != 0) {
    size &= ~3;
    size += 4;
  }

  /* Iterate through list of free blocks, using the first that is
     big enough for the request. */
  for (block = free_list, prev_p = &free_list; block; prev_p = &(block->next),↩
       block = block->next) {
    if ( (int)( block->size - size - sizeof(size_t) ) >= (int)( MIN_ALLOC_SIZE↩
        +sizeof(size_t) ) ) {
      /* Block is too big, but can be split. */
      block->size -= size+sizeof(size_t);
      free_block_t *new_block = (free_block_t*) (((byte*)block)+block->size);
      new_block->size = size+sizeof(size_t);
      return ((byte*)new_block)+sizeof(size_t);
    } else if (block->size >= size + sizeof(size_t)) {
      /* Block is big enough, but not so big that we can split
         it, so just return it */
      *prev_p = block->next;
      return ((byte*)block)+sizeof(size_t);
    }
    /* Else, check the next block. */
  }
  free_block_t* free_block = (free_block_t*) syscall_memlimit(heap_ptr + size ↩
      + MIN_ALLOC_SIZE);
  if ((free_size = size % 4096) != 0) {
    free_block->size = free_size;
    free_block->next = free_list;
    free_list = free_block;
  }
  free_block_t* return_block = (free_block_t*) heap_ptr;
  return_block->size = size + MIN_ALLOC_SIZE;
  return heap_ptr + MIN_ALLOC_SIZE;
}
```

3 Extended tests for TLB exceptions and user-space allocation

The program below, `tests/mallol.c` tests user space allocation. We test both a small array of size 2 and a large array of size 1100, which is filled from 0..649 with values 0..649. In both cases we also free the arrays. We test both arrays by picking values of the arrays to print and check that these values match what we expect.

```
#include "tests/lib.h"

int* hej() {

  int* a = (int*) malloc(2*sizeof(int));
  a[0] = 10;
  a[1] = 20;
  return a;
}


int main(void)
{

/*  int* a = hej();

  printf("a[0] = %d, a[1] = %d\n",a[0],a[1]);
  free(a);*/

  int i;
  int* b = (int*) malloc(1100*sizeof(int));
  for(i=0; i <1100; i++) {
    b[i] = i;
  }

  printf("b[0] = %d, b[200] = %d\n",
          b[0],b[200]);

  free(b);
  return 0;
}
```

We also test for a heap overflow by multiplying our page size by 17 and try to `malloc` with this amount. This should cause a heap overflow, since we exceed the amount available for the heap.

```
#include "tests/lib.h"



int main(void)
{
  /* Attempts to allocate more physical memory than
   * the TLB can hold.
   */
  int* overflow = malloc(4096*17*4);
  overflow [0] = 1;
  return 0;
}
```

Note that none of these tests pass, since our implementation of `malloc` or `sys call_memlimit` is bugged.