UNIVERSITY OF COPENHAGEN
Department of Computer Science
February 27, 2015

G Assignment 4 for "Operating Systems and Multiprogramming", 2015

Deadline: March 9, 2015 at 23:59

Handing in: via Absalon

Formalities

Please, follow the same procedure as in the previous assignments. In the tasks you are supposed to extend Buenos. Hence, hand in 1) a source tree for Buenos which contains your work, including a `Makefile` and the programs used for testing; and 2) a short report where you explain what you have done.

Task 1: TLB exception handling in Buenos

The Buenos system does not handle exceptions caused by TLB misses (when a page cannot be found in the TLB). These exceptions should be handled by placing an entry for the requested page in the TLB—potentially removing the oldest existing entries (FIFO replacement).

A *random* replacement strategy for page entries in the TLB is easy to implement (just use function `tlb_write_random` in Buenos). An advanced solution will instead implement a FIFO replacement, but a random replacement is acceptable.

1. Implement handlers for TLB related exceptions in Buenos. There are three exceptions:

   - EXCEPTION_TLBL: A memory *load* operation required a page which either had no entry in the TLB or whose entry was marked as invalid.
   - EXCEPTION_TLBS: A memory *store* operation required a page which either had no entry in the TLB or whose entry was marked as invalid.

   Both cases should be handled in the same way: the page table of the respective thread is searched for a matching page entry. If found, the page is inserted into the TLB (removing the oldest existing entry if necessary). If not, or if an invalid page is accessed, it is treated as an access violation (or leads to kernel panic when in kernel mode).

   - EXCEPTION_TLBM: A memory *store* operation required a page whose `dirty` bit was 0, indicating that the page is not writable.

   This is treated as an access violation by the executing user process (or leads to kernel panic when it occurs in kernel mode).

   Implement your handlers as the functions `tlb_load_exception`, `tlb_store_exception`, and `tlb_modified_exception` in file `vm/tlb.c`. Information about the exception is obtained via `_tlb_get_exception_state`, which fills a `tlb_exception_state_t` structure.

2. Modify the Buenos exception handlers, so TLB exceptions will trigger the handlers implemented by you. You need to handle both user exceptions and kernel exceptions.

3. Remove all calls to the function `tlb_fill`; this function should not be necessary any more, now that TLB misses are handled properly.

Task 2: Dynamic memory allocation for user processes

This exercise is based on a Buenos implementation which supports user processes. You can base your solution on the enclosed version, or you can use your own version (if you choose to base it on your own version, you will need to copy the `tests/lib.[ch]` files from our package).

Add a new field `heap_end`, which holds the address of free memory mapped to a physical page, to the process control block. User code can manipulate this heap limit, increasing it in order to allocate more memory when required, through the system call `syscall_memlimit`.

1. Implement the system call `void* syscall_memlimit(void* new_end)`.

   Effect: Allocate or free memory, reserved for a process, by trying to set the heap to end at the address `new_end` given as the parameter. *Returns:* The new end of the heap (the last addressable byte) or `NULL` on error. If `new_end` is `NULL`, it returns the current `heap_end`.

   Please note that the functionality `vm_unmap` to *unmap* a memory page from memory is left unimplemented in Buenos (see [Buenos roadmap, p. 52]). It is therefore not possible to give back allocated memory to the system. A call to `syscall_memlimit` to *decrease* the value of `heap_end` should be considered an error, as the heap can only *grow*, not *shrink*.

2. Modify the existing implementation of `malloc` to call `syscall_memlimit` when there is no more space. You will also need to modify `heap_init`.

The userland library (`tests/lib.[ch]`) contains a working `malloc` and `free` implementation using a list of free memory blocks that `malloc` iterates through to find a sufficiently large block, which it then returns a pointer to. However, `malloc` currently allocates from a pool of only 256 bytes. Recall the interface of these two functions:

`void* malloc(size_t size)`
: *Effect:* Allocate the amount of bytes specified by `size` from the heap. The allocated memory is uninitialized. *Returns:* A pointer to the first byte of the allocated memory.

`void free(void* ptr)`
: *Effect:* Free the memory space pointed to by `ptr`, which must have been returned by a previous call to `malloc`. Otherwise, or if `free(ptr)` has already been called before, undefined behavior occurs. If `ptr` is `NULL`, nothing is done. *Returns:* Nothing.

Bonus task: Fast memory allocation and freeing

1. Implement the library function `void* malloc(size_t)`. *Running time:* $O(1)$ in the worst case.

2. Implement the library function `void free(void*)`. *Running time:* $O(1)$ in the worst case.

In Buenos the current implementation of `malloc` runs in $\Theta(M)$ time in the worst case, where M is the size of the heap in bytes. Implement the memory management routines `malloc` and `free` so that both run in $O(1)$ worst-case time. What can you say about memory fragmentation? You can implement the routines either in Buenos, using `syscall_memlimit`, or outside Buenos, in which case you need to use the system call `brk` (see, the manual page for more information).

To meet the runtime requirements, we recommend that you use the online allocation algorithm known as the *colouring algorithm* (a variant of the *quick-fit* algorithm) (see Supplementary Literature on the course home page) that uses segregated lists for memory blocks of size 16, 32, 64, ..., 2^{31} bytes. To round an integer `n` to the nearest power of 2 that is larger than or equal to `n`, you can use the function `ilogb` from `<math.h>` or the function `leftmost_one` coming together with this problem formulation. Otherwise, you are welcome to reuse as much as possible any existing code, e.g. that given in [KR, Section 8.7].