UNIVERSITY OF COPENHAGEN
Department of Computer Science
Jost Berthold   Jyrki Katajainen
`berthold@di.ku.dk`   `jyrki@di.ku.dk`

February 28, 2014

G-Assignment 4 for "Operating Systems and Multiprogramming", 2014

Deadline: March 10 at 23:59

Handing in: via Absalon

Formalities

Please, follow the same procedure as in the previous assignments. In both 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.

Tasks

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).

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 3 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 pagetable 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 in 1. 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.

Dynamic allocation for user processes

This exercise is based on a Buenos implementation which supports user processes. You can base your solution on our model solution for G2, or you can use your own.

In the Buenos process control block (PCB), a field `heap_end` will be added which holds the address of free memory mapped to a physical page.

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 *heap end)`.

   `memlimit` tries to allocate or free memory by setting the heap to end at the address `heap_end` given as the parameter. `syscall_memlimit` returns the new end address of the heap (the last addressable byte), or `NULL` on error. If `heap_end` is `NULL`, the current `heap_end` is returned.

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

2. Implement two library functions `malloc` and `free` in the user-space library (`tests/lib.[ch]`) which manage memory allocations by a user-space program.

   Function `void* malloc(unsigned int size)` reserves the amount of bytes given as `size` and returns a pointer to the first byte.

   Function `void free(void* ptr)` releases memory pointed at by the pointer argument `ptr`. It is an error to return memory which has not been been previously allocated by `malloc`.

The library `tests/lib.[ch]` which came with the Buenos source code for exercise 2 already contains functions to allocate and free memory in the described way. Much of this code can be reused, but the current implementation uses a fixed pre-allocated `char` array. Your new version should internally manage an *initially empty* list of free blocks, and allocate more heap on demand using system call `memlimit`.

Extended tests for TLB exceptions and user-space allocation

Write a set of extensive tests for user-space allocation, specifying the expected result for each test. As a minimum, your tests should cover testing the behaviour of Buenos memory management for the following cases:

- Successful allocation, use, and freeing of small and large amounts of memory;
- Unsuccessful allocation of *too large* amounts of memory (a heap overflow).

Pay particular attention to extensively documenting test results in your report.