Phase 3: Virtual Memory

Version 1.0

November 8, 2019

Phase 3a due: November 14 @ 10pm

Phase 3b due: November 21 @ 10pm

Phase 3c due: December 3 @ 10pm

Phase 3d due: December 10 @ 10pm

1. Overview

For this phase of the project you will implement a virtual memory (VM) system that supports demand paging. The USLOSS MMU is used to configure a *VM region* of memory that can be configured so that its content is private to each process; data written to the region by one process are not visible to other processes. For example, suppose the VM region is at address 0x4000 and consider two processes A and B that have just been spawned. If both processes read the byte at address 0x4000 they will read the value 0. If A then writes the character 'A' to address 0x4000 it will read 'A' from that address, whereas B will continue to read 0. If B then writes 'B' to address 0x4000 it will read 'B' while A continues to read 'A'.

You will configure the USLOSS MMU to provide a single-level page table (see Section 5.1 of the USLOSS manual). The TLB features of the MMU will not be used. You will write the routines that manage a page table for each process. A page table is created when a process is forked and deleted when the process quits. You will also need to implement *pager* processes that handle demand paging -- allocating frames for new pages, reading pages from disk on page faults as necessary, and writing them out to disk as part of page replacement. Finally, you will implement two new system calls `Sys_VmInit` and `Sys_VmShutdown` that initialize and shut down the VM region. Implementing this phase will require some modifications to previous phases; we will provide new libraries that have these modifications.

This phase is different from the other phases in that the parts are interdependent -- earlier parts must call functions provided by later parts. You will write the basic framework in Phase 3a, including `P3_Startup`, using stub functions for any functions implemented in subsequent parts. You will replace the stub functions with real functions later. Because of the interdependency between the parts and the complexity it introduces the skeleton code will be more substantial than in previous phases.

Phase 3a: page tables that are statically filled with PTEs that map page x to frame x.

Phase 3b: page tables that are dynamically filled with PTEs that map page x to frame x.

Phase 3c: page tables that are dynamically filled by pager daemons. New pages are allocated free frames. There are guaranteed to be enough frames for all pages.

Phase 3d: page tables that are dynamically filled by pager daemons. There may be more pages than frames, requiring pages to be swapped to and from disk.

2. Phase 3a: Static Identity Page Tables

For Phase 3a you will implement static identity page tables. The page tables are static because they are filled in when the processes are created. Each page table implements the identity mapping -- page *x* is mapped to frame *x*. Thus there are exactly the same number of frames as pages. All pages fit in memory and are mapped by the page table when a process begins, so there are never any page faults nor are pages ever swapped out to disk.

2.1 Initialization and Cleanup

Your Phase 3 code starts running in the process `P3_Startup`, the user-level process spawned by Phase 2. Your `P3_Startup` should in turn spawn the process `P4_Startup` (the test program) with a stack of size 4 * USLOSS_MIN_STACK, priority 3, and a NULL argument. `P4_Startup` will use two new system calls to initialize and teardown the virtual memory system, `Sys_VmInit` and `Sys_VmShutdown`, respectively. These system calls are described in detail in Section XXX, and Phase 2d has been modified to install system call handlers that call the underlying functions `P3_VmInit` and `P3_VmShutdown`. `P3_Startup` should call `Sys_VmShutdown` if `P4_Startup` returns.

2.2 Page Table Management

Each process has its own page table, which is an array of USLOSS_PTE structures indexed by page number. Each PTE contains information about a page, including whether or not it is in memory (the "incore" bit), the frame that contains the page (the "frame" field) and whether the page is readable and/or writable (the "read" and "write" bits, respectively). The page tables are managed by the routines `P3_AllocatePageTable`, and `P3_FreePageTable` (defined in *phase3.h*). `P3_AllocatePageTable` is called by `P1_Fork` and creates a page table for the process. `P3_FreePageTable` is invoked by `P1_Quit` and frees the process's page table. `P3_FreePageTable` will also call functions implemented by later phases clean up additional state associated with the process.

`P1_Fork` calls `P3_AllocatePageTable` and passes the page table to `USLOSS_ContextInit`. When the context is subsequently passed to `USLOSS_ContextSwitch` USLOSS will automatically load the new process's page table into the MMU. Sometimes, however, it is useful to modify the currently-running process's page table. USLOSS cannot automatically detect changes to the page table of the currently-running process, so if you change the page table you must then call `USLOSS_MmuSetPageTable` to notify USLOSS of the changes. If necessary you can call `USLOSS_MmuGetPageTable` to get the page table that is currently loaded in the MMU.

2.3 Functions

`USLOSS_PTE *P3_AllocatePageTable(int pid)`

Returns a page table for the specified process, if `P3_VmInit` has been called, otherwise returns `NULL`. First calls `P3PageTableAllocateEmpty` to allocate an empty page table, and if that fails calls `P3PageTableAllocateIdentity` to allocate a page table with identity mapping.

Return Values:

NULL if table could not be allocated, table otherwise

Pseudo-code

```
P3_AllocatePageTable(pid)
     if P3_VmInit has been called
          table = P3PageTableAllocateEmpty(numPages)
          if  (table == NULL)
               table = \
PageTableAllocateIdentity(numPages)
          return table
     else
          return NULL
```

`USLOSS_PTE *PageTableAllocateIdentity(int numPages)`

Returns NULL if `P3_VmInit` has not been called. Otherwise returns a page table whose PTEs map page *x* to frame *x*.

Return Values:

NULL if table could not be allocated, table otherwise

`void P3_FreePageTable(int pid)`

Frees a previously allocated page table. Calls `P3FrameFreeAll` to free all frames used by the page table and `P3SwapFreeAll` to free all swap space used by the page table, then frees the page table associated with the `pid`. Does nothing if `P3_VmInit` has not been called.

Return Values:

`P1_INVALID_PID`:	invalid `pid` or process does not have a page table
`P1_SUCCESS`:	success

`void PageTableFree(int pid)`

Called by `P3_FreePageTable` to free a page table. Does nothing if `P3_VmInit` has not been called.

`int P3_VmInit(int unused, int pages, int frames, int pagers)`

Initializes the VM system. Calls `USLOSS_MmuInit` to initialize the MMU then calls `P3FrameInit` to initialize the frames, `P3PagerInit` to initialize the pagers, and `P3SwapInit` to initialize the swap space.

Return Values:

`P3_ALREADY_INITIALIZED`:	this function already called
`P3_INVALID_NUM_PAGES`:	number of pages is invalid
`P3_INVALID_NUM_FRAMES`:	number of frames is invalid
`P3_INVALID_NUM_PAGERS`:	number of pagers is invalid
`P1_SUCCESS`:	success

`int MMUInit(int pages, int frames)`

Called by `USLOSS_MmuInit` to initialize the MMU via `USLOSS_MmuInit`.

Return Values:

`P3_ALREADY_INITIALIZED`:	MMU already initialized
`P3_INVALID_NUM_PAGES`:	number of pages is invalid
`P3_INVALID_NUM_FRAMES`:	number of frames is invalid
`P1_SUCCESS`:	success

`int P3_VmShutdown(void)`

First shuts down the subsystems in the reverse order of `P3_VmInit`, i.e. `P3SwapShutdown`, `P3PagerShutdown`, and `P3FrameShutdown`. Then calls `USLOSS_MmuDone` to shut down the MMU and frees the page tables. Does nothing if `P3_VmInit` has not been called.

`int MMUShutdown(void)`

Shuts down the MMU via `USLOSS_MmuDone`. Does nothing if `P3_VmInit` has not been called.

Return Values:

`P1_SUCCESS`:	success

`int P3PageTableGet(int pid, USLOSS_PTE **table)`

Returns in `*table` the page table for process `pid` if one exists, otherwise returns NULL in `*table`.

Return Values:

`P1_INVALID_PID`:	invalid `pid`
`P1_SUCCESS`:	success

`int P3PageTableSet(int pid, USLOSS_PTE *table)`

Sets the page table for process `pid` to `table`.

Return Values:

`P1_INVALID_PID`:	invalid `pid`
`P1_HAS_TABLE`:	process already has a page table
`P1_SUCCESS`:	success

One complication is that Phase 1 will call `P3_AllocatePageTable` and `P3_FreePageTable` before `Sys_VmInit` is called and after `Sys_VmShutdown` is called, respectively. Therefore, the above routines should have no effect if the VM system is uninitialized, e.g. `P3_AllocatePageTable` should return NULL until `P3_VmInit` is called.

2.4 System Calls

Phase 3a implements two system calls, `Sys_VmInit` and `Sys_VmShutdown`, for initializing and shutting down the virtual memory system, respectively. Phase 2d has been modified with stubs for these system calls that call `P3_VmInit` and `P3_VmShutdown`. The details on these system calls is provided here for your information.

Sys_VmInit (syscall 24)

Initializes the VM system by calling `P3_VmInit`.

Input

arg1: (unused)

arg2: number of virtual pages per process

arg3: number of physical page frames

arg4: number of pager daemons

Output

arg1: address of the VM region

arg4: return code from `P3_VmInit`

Sys_VmShutdown (syscall 25)

Calls `P3_VmShutdown` to shut down the virtual memory system.

2.5 VM Statistics

In *phase3.h* you'll find an external definition for the variable `P3_vmStats`. You must declare this variable in your code and update its fields appropriately. The comments in the header file should explain the fields sufficiently. `Sys_VmShutdown` should print out the statistics using `P3_PrintStats`, provided in the starter code. This is a shared data structure so you must provide mutual exclusion on it.

3. Phase 3b: Dynamic Identity Page Tables

Phase 3b is similar to Phase 3a except that the PTEs are filled dynamically. `P3_AllocatePageTable` returns a page table whose PTEs do not contain mappings, i.e. the "incore" bit is 0. This will cause a page fault interrupt when a process tries to access one of these pages. This phase implements a page fault interrupt handler that fills in the PTE for the offending page with the identity mapping, i.e. page *x* is mapped to frame *x*. Like Phase 3a there are exactly the same number of frames as pages so all pages fit in memory and pages are never swapped out to disk.

Details to follow.

4. Phase 3c: Virtual Addressing

Details to follow.

5. Phase 3d: Virtual Memory

Details to follow.

6. Submission

Submit your solutions via GradeScope. Only one partner should submit and should indicate via the GradeScope submission process the name of their partner. Design and implementation will be considered, so make sure your code contains insightful comments, variables and functions have reasonable names, no hard-coded constants, etc. You should NOT turn in any files that we provide, e.g. the USLOSS source files, *`phase3.h`*, etc., nor should you turn in any generated files (e.g. .o files, core files, etc.)