Phase 3: Demand Paging

Part A due April 18 @ 10pm

Part B due May 2 @ 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 whose contents are not shared between processes; data written to the region by one process are not visible to other processes. For example, assume 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 the 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, bringing pages in 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_VmDestroy` that initialize and destroy the VM region. Implementing this phase will require some modifications to Phase 1 and Phase 2; we will provide new libraries for Phase 1 and 2 that have these modifications.

2. 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_VmDestroy`, respectively. These system calls are described in detail in Section 6, and Phase 2 must be modified to install system call handlers that call the underlying functions `P3_VmInit` and `P3_VmDestroy`. `P3_Startup` should call `Sys_VmDestroy` if `P4_Startup` returns.

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, if any (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 (but not allocate any page frames).

`P3_FreePageTable` is invoked by `P1_Quit` and frees the process's page table, any frames and disk blocks it is using.

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

`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 currently loaded in the MMU.

3. Page Faults and Pagers

A *page fault* occurs the "incore" bit for an access page is 0, i.e. the page is not in memory. This causes an `USLOSS_MMU_INT` interrupt. Page faults are handled by one of several *pager daemons*, kernel processes that run at priority 2 with interrupts enabled. Pagers are responsible for moving pages between memory and disk, and there are several of them to allow simultaneous page faults by multiple processes to be serviced simultaneously. Pagers are forked by `Sys_VmInit` and they quit when `Sys_VmDestroy` is called. `P3_MAX_PAGERS`, defined in *phase3.h*, is the maximum number of pagers you must support; however, if `P3_MAX_PAGERS` is changed your solution should simply require recompilation and not modification.

Your page fault handler must block the faulted process until a pager has serviced the fault. Pagers wait for page faults to occur, service them as described below, then unblock the faulting process. In the meantime other processes may run, and other page faults may occur. The handler/pager synchronization is a bit tricky because `USLOSS_MmuGetCause` only returns the cause of the last interrupt, and there may be more than one simultaneous page faults, so you'll want to create a mailbox for pending page faults. Messages are sent to this mailbox by the page fault handler and contain information about the process that suffered the fault and the cause of the fault. Otherwise, by the time the pager calls `USLOSS_MmuGetCause` another page fault may have overwritten the original cause. The pagers receive from this mailbox and service the faults indicated by the messages. Note that a process can only have one outstanding page fault at a time, so the mailbox can have at most `P1_MAXPROC` messages in it.

To handle a page fault a pager must first allocate a frame to hold the page. First, it looks for a free frame. If one is found the corresponding PTE in the page table of the faulted process is updated with the allocated frame number. The next time the faulted process runs the updated page table will be loaded into the MMU, the process will repeat the offending instruction, and this time a fault will not occur because the page has been mapped to a frame. If this is the first time the page has been accessed the pager zeros the contents of the page. Note that the pager must get access to the contents of the frame to zero it, and to do so it will have to update its own page table so that one of its pages maps to the desired frame. Once it has zeroed the frame it should remove this mapping.

If the faulted page already exists then it must be on disk (if it were in memory there would not have been a page fault). Disk 1 is used storing pages that are not in memory, leaving disk 0 for use by user programs. For historical reasons the disk that stores pages is known as the *swap disk*. The pagers use `P2_DiskRead` and `P2_DiskWrite` to access the swap disk. Note that the pager must update its own page table to get access to the frame prior to calling these routines.

4. Page Replacement

A pager may discover that there are no free frames when handling a page fault. When this occurs the pager must re-use one of the existing frames via the "clock" algorithm. A clock hand is maintained that cycles through the frames. When a frame is needed the hand is advanced, clearing the reference bits as it moves (use `USLOSS_MmuGetAccess)`. The first frame whose reference bit is not set is chosen for reuse. The dirty bit of the chosen frame is examined and if it is set then `P2_DiskWrite` is used to write the page out to the disk. As described in the previous section, you will probably want to copy the frame into a temporary buffer before calling `P2_DiskWrite`. Once the frame is empty the pager then fills it with the page that suffered the fault, either by setting all the bytes to zero or by reading the page from disk, whichever is appropriate.

The frames are a resource that is shared by the pagers. You must ensure that there is mutual exclusion on them, including frame allocation (you don't want two pagers to accidentally allocate two pages to the same frame), disk allocation (you don't want two pagers to allocate two pages to the same disk block), and the clock hand. Despite all of this mutual exclusion, the pagers should run relatively independently. The idea is that some can be waiting on the disk while others continue to handle page faults, allowing computation to be overlapped with I/O.

5. Swap Management

Pages are stored on disk (the swap device) when they are not in memory. The way to conceptualize this is that all pages live on the disk, and are brought into memory as necessary. When a dirty page is replaced the pager must write it back to the disk, and when it's time to bring a page back into memory the pager must read it from disk. Therefore the pager must keep track of each page's location on the disk. You can't modify the USLOSS_PTE structure, so you'll need a separate data structure to hold this information. Swap space for a page should only be allocated the first time the page is written (i.e. there is no need to store empty pages on the disk). If the pager cannot allocate swap space for a page it should print an appropriate error message and terminate the process that suffered the page fault.

6. VM System Calls

Sys_VmInit (syscall 24)

Initializes the VM system. This includes installing a handler for the `MMU_INT` interrupt, creating the pager daemons, and calling `MMU_Init` to initialize the MMU.

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: 0 -- success

-1 -- illegal values are given as parameters
-2 -- the VM region has already been initialized

Sys_VmDestroy (syscall 25)

Shuts down the VM system. The pager daemons quit and any memory allocated in `Sys_VmInit` freed. The MMU is turned off by calling `MMU_Done`. Has no effect if `Sys_VmInit` has not been called previously.

7. 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_VmDestroy` should print out the statistics. The skeleton file has an example of how to do this. Once again this is a shared data structure so you must provide mutual exclusion on it.

8. Submission

8.1 Part A

Part A consists of the following functions, system calls, interrupt handlers, and processes:

- Sys_VmInit
- Sys_VmDestroy
- P3_VmInit
- P3_VmDestroy
- P3_AllocatePageTable
- P3_FreePageTable
- Page fault handler
- Single pager daemon

For Part A you are to implement on-demand mapping of pages to page frames. When a process is forked `P3_AllocatePageTable` creates an initially empty page table for the process. On a subsequent page fault the page fault handler sends a message to the pager daemon mailbox. Upon receiving this message the pager daemon allocates a free frame to hold the page, fills the frame with zeros, and updates the page table entry The test cases will guarantee that # processes * # pages in VM region <= # page frames so that there will always be enough page frames to hold all the pages for all the processes (i.e. you don't need to do demand paging).

Here is a rough outline of the Part A implementation:

```
Page fault handler (USLOSS_MMU_INT)
    send request to pager mailbox (PID, page #, cause)
    receive response
Pager
    receive page fault request from mailbox
    find free frame, fill with zeros (guaranteed to be one)
    update faulting process's PTE to map page to frame
    send response to faulting process
P3_AllocatePageTable
    create page table for new process with no pages "incore"
P3_FreePageTable
    free frames used by process
    free process's page table
P3_VmInit
    initialize MMU
    fork pager
P3_VmDestroy
    kill the pager
    release the MMU
```

8.2 Part B

Part B consists of the entire phase as described in this document. This includes adding support for the clock page replacement algorithm, swapping pages to and from the swap disk, and multiple pager daemons. The test cases will guarantee that # processes <= # page frames so that there is at least one page frame available per process, thus avoiding thrashing.

Here is a rough outline of the Part B implementation:

```
Page fault handler (USLOSS_MMU_INT)
    send request to pager mailbox (PID, page #, cause)
    receive response
Pager
    receive page fault request from mailbox
    if there are free frames
        find free frame
    else
        use clock algorithm to find best frame to use
        if frame is dirty
            write its page to the swap disk
        update frame owner's page table (page is no longer in frame)
    if new page
        fill frame with zeros
    else
        read page from disk into frame
    update faulting process's PTE to map page to frame
    send response to faulting process
P3_AllocatePageTable
    create page table for new process with no pages "incore"
P3_FreePageTable
    free frames used by process
    free disk blocks used by process
    free process's page table
P3_VmInit
    initialize MMU
    fork pagers
P3_VmDestroy
    kill the pagers
    release the MMU
```

9. Incremental Implementation

Generally it's a bad idea to write all of the code, then see if it works. A much better plan is to write and test the code incrementally. For example, you can take the skeleton file and change it so that there are as many frames as pages. `P3_AllocatePageTable` creates a page table that maps page 0 to frame 0, page 1 to frame 1, etc. Page faults should never occur, so you can run this test without your pager daemons, page tables, swap management, etc., working. Once this works, try installing an interrupt handler for the MMU and update the page table on demand. Then write a pager daemon that fills in the page table so that you can get the synchronization correct between the interrupt handler and the pager. You get the idea.