Phase 5: Demand Paging

Complete program due: Wednesday, December 9th, 9:00 pm

1.0 General Description

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 region of virtual memory whose contents are process-specific.

You will use the USLOSS MMU to implement the virtual memory system. The basic idea is to use the MMU to implement a single-level page table, so that each process will have its own page table for the VM region and will therefore have its own view of what the VM region contains. The fancy features of the MMU, such as the tags and the protection bits, will not be needed.

2.0 Initialization and Cleanup

Your Phase 5 code will start running in the function **`start4`**, the first user-level process created by Phase 4. You should not initialize the VM system in **`start4`**; it will instead be initialized by a new system call, **`VmInit`** and cleaned up in **`VmDestroy`**. Your **`start4`** function should fill in the **`systemCallVec`** to handle these new system calls, then **`Spawn`** and **`Wait`** for the function **`start5`**. **`start5`** (the test program) will call **`VmInit`**, causing the MMU to be initialized and returning the address of the VM region. **`start5`** should call **`VmDestroy`** before **`start5`** terminates.

2.1 VmInit (syscall SYS_VMINIT)

Initialize the VM region. This includes installing a handler for the **`MMU_Init`** interrupt, creating the pager daemons, and calling **`MMU_Init`** to initialize the MMU. For the standard assignment the number of mappings should equal the number of virtual pages and -1 should be returned in arg4 if they do not. When the MMU is initialized the current tag is set to 0. Do not change it.

Input

arg1: number of mappings the MMU should hold
arg2: number of virtual pages to use
arg3: number of physical page frames to use
arg4: number of pager daemons

Output

arg1: address of the VM region
arg4: -1 if illegal values are given as input or the VM region has already been initialized; 0 otherwise.

2.2 VmDestroy (syscall SYS_VMDESTROY)

Shuts down the VM system. The pager daemons are killed and any memory allocated in **`VmInit`** freed. The MMU is turned off by calling **`USLOSS_MmuDone`**.

Has no effect if **`VmInit`** has not been called previously.

3.0 Process Management

Each process has its own page table. Each page table entry contains information about the page, including whether or not it is in memory, and whether or not it is stored on disk. The page tables are managed by the routines **`p1_fork`**, **`p1_switch`**, and **`p1_quit`**, invoked by Phase 1.

The page table for a process is created during fork, and destroyed during quit. Initially the page table does not contain any valid entries, as this is a demand-paging system and the process has not used any pages yet. During a context switch **`p1_switch`** unloads all of the mappings from the old process and loads all of the valid mappings for the new. A valid mapping is one in which there is a frame for the page — if there is not a frame for the page you do not put a mapping in the MMU. If a nonexistent page is referenced, an MMU interrupt will occur and then you will allocate a frame.

One complication is that Phase 1 will call **`p1_fork`**, **`p1_switch`**, and **`p1_quit`** as soon as processes are created and start running, which will happen before **`start4`** starts running and you have a chance to initialize your data structures. Set up these p1 routines so they have no effect if **`VmInit`** has yet to be called.

4.0 Page Faults and Pagers

A page fault occurs when the MMU does not contain a mapping for an accessed page. This causes a **`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 multiple page faults to be serviced simultaneously. Pagers are forked in **`VmInit`** and killed in **`VmDestroy`**. **`MAXPAGERS`**, defined in *phase5.h*, is the maximum number of pagers you have to support.

When a process suffers a page fault an **`USLOSS_MMU_INT`** interrupt will occur. The interrupt handler blocks the faulted process until a pager has serviced the fault. Pagers wait for page faults, 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 many simultaneous page faults; you will probably want to create a queue (a mailbox should work for this, or a standard queue) of pending page faults that contains information about pending page faults. Information is put in this queue by the interrupt handler, and removed by the pagers. Note that a process can only have one outstanding page fault at a time, so the queue can have at most **`MAXPROC`** items in it.

When a process suffers a page fault, the pager must allocate a frame to hold the page. First it checks a free list of frames. If a free frame is found, the page table entry for the faulted process is updated to refer to the free frame. Note that the new mapping is not loaded into the MMU by the pager because we want the page to be mapped for the faulted process, not the pager itself. Instead, the pager puts the mapping into the faulted process's page table and it will be loaded into the MMU the next time that process is run.

After the page table has been updated, the pager must initialize the contents of the frame. If the page contents are currently on disk then the pager must use **`disk_read`** to read them into the frame. If this is the first time the process has touched the page then it will have no current contents, in which case the pager must erase the old contents of the frame by setting all bytes to zero. Note that in both cases the pager must get access to the contents of the frame, and to do so it will have to update its own page table and load a mapping into the MMU so that the frame is accessible. Once it has filled in the frame, it should remove this mapping.

Pages are stored on disk when they are not in memory. Disk 1 is used for this purpose, 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 **`disk_read`** and **`disk_write`** to access the swap disk. Note that a frame address cannot be accessed by the disk driver directly without putting an entry in the disk driver's page table to allow the disk driver to do so. Worse, the disk driver does not have a page table, since it started before virtual memory was initialized. An easy way around the problem is to have the disk driver access a temporary buffer outside the VM region and have the pager copy data between the buffer and the VM region as appropriate.

5.0 Page Replacement

A pager may discover that there are no free frames when handling a page fault. When this happens, the pager must reuse one of the existing frames. The pager uses the "clock" algorithm to determine which frame to reuse. A clock hand is maintained that cycles through the frames. When a frame is needed, the hand is advanced, clearing the reference bits as the hand moves. The first frame whose reference bit is not set is chosen for reuse. Note that there is only one clock hand even though there are several pagers (the pagers will need to synchronize).

Once a frame is selected, its dirty bit is examined and if it is 1 then **`disk_write`** is used to write the page out to the disk. As described in the previous section, you will probably have to copy the frame into a temporary buffer before calling **`disk_write`**. After cleaning the frame, 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 they have mutual exclusion on the free frame list, and that they have mutual exclusion when accessing a frame (you do not want two pagers to accidentally assign the same frame to two different pages). You must also ensure that the pagers share the same clock hand. Despite all of this sharing, 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 and allowing the disk driver to schedule disk accesses.

6.0 Swap management

Until a page is accessed for the first time by a process, the page does not exist anywhere. The first access of a page by a process results in the page existing in the frame to which it is assigned. The contents of this frame are zeroed out when the frame is assigned to this page. From this point forward in time, the page must be either in a frame of memory or must be stored on the disk (the swap device) or both. When a page is thrown out of memory, you may need to write it to the disk. Writing to the disk will occur if the page has not yet been written to the disk, or the page was previously written to the disk but has changed since that time (i.e., the page is "dirty"). When it is time to bring a page back in memory, you must read it from disk. Therefore you must keep track of each page's location on the disk. The easiest way to do this is to keep an extra field in the page table entry that indicates which disk block contains the page. Swap space for a page should only be allocated the first time the page is written. If you cannot allocate swap space for a page, you should terminate the process that suffered the page fault.

7.0 VM Statistics

In *phase5.h*, you will find an external definition for the variable **`vmStats`** of type **`VmStats`**. You need to declare this variable in your code and update its fields appropriately. The comments in the header file should explain the fields sufficiently. In **`VmDestroy`**, you 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 will need to provide mutual exclusion on it.

8.0 Incremental Implementation

As many of you learned in earlier phases, it is 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 set it up so that **`VMInit`** initializes the MMU with an equal number of pages and frames, loads some simple mappings that map page 0 to frame 0, and spawns a child that accesses the VM region. 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 loading the mappings on demand. You still do not need a page table. Then write a simple pager process that loads the mappings so you can get the synchronization correct between the interrupt handler and the pager. You get the idea.

9.0 Debugging Tip

The USLOSS MMU is implemented using **`SIGSEGV`** signals. If you do not want gdb to print out a message each time the process receives a **`SIGSEGV`** signal put the following in your *.gdbinit* file:

```
handle SIGSEGV nostop noprint
```

10.0 Initial Startup

After your phase 5 has completed initialization, it should spawn a user-mode process running **`start5`**. This initial user process should be allocated **`8 * USLOSS_MIN_STACK`** of stack space, and should run at priority **`PAGER_PRIORITY`**. **`start4`** should then wait for **`start5`** to finish.

11.0 Phase 1, 2, 3 and 4 Libraries

We will grade your phase 5 using the phase 1, 2, 3 and 4 libraries that we supply. The provided libraries will be named: *libpatrickphase1.a*, *libpatrickphase2.a*, *libpatrickphase3.a* and *libpatrickphase4.a*. These libraries will be available in */home/cs452/fall15/lib*.

You may use either your earlier libraries or the ones the we supply while developing your phase 5. However, it will be graded using the phase 1, 2, 3 and 4 libraries that we will supply.

12.0 Submitting Phase 5 for Grading

For the turnin, you will need to submit all the files that make up your phase 5. 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 may, if helpful, turn in a *README* file to help us understand your solution. Your *Makefile* must be arranged so that typing 'make' in your directory will create an archive named *libphase5.a*. Your *Makefile* must also have a target called 'clean' which will delete any generated files, e.g. .o files, .a files, core files, etc. You should NOT turn in any files that we provide, e.g. *libusloss.a*, *usloss.h, phase1.h*, etc., nor should you turn in any generated files.

Doing **`make submit`** will create a **`phase5.tgz`** file that contains:

- The **`.c`** files for your phase5. This will include **`phase5.c`**, **`p1.c`**, and any other **`.c`** files that you have created. If you have created additional **`.c`** files, you should have already modified **`COBJS`** to include these additional **`.o`** files.
- The **`.h`** files for your phase5. This will include **`driver.h`**. If you have created other **`.h`** files, you should have already modified **`HDRS`** to include the additional **`.h`** file(s).
- The **`Makefile`**.

You will *not* submit the usloss library or directory. You will *not* submit the phase 1-4 libraries. We will put a link for usloss and the phase libraries into the grading directory after we extract your files. The same is true for the **`testcases`** directory — do not submit the test cases. We will add links for the **`testcases`** and **`testResults`** directories.

You are encouraged to create the **`phase5.tgz`** file, then copy it to a different area of your home directory, extract the file using **`tar xvfz phase5.tgz`** and do a compile to see if everything has been included.

Any file(s) that are missing from your submission that we have to request from you will result in a reduction of your grade!

To submit the **`phase5.tgz`** file to D2L

- Log on to D2L, and select CSc 452.
- Select "Dropbox".
- You will find *Phase5*, and *Phase5-late*. Only one of these will be active. Click on the active one — will be *Phase5* if you are turning in the work on time.
- You should now be at the page: "Submit Files - Phase5". Click on the "Add a File" button. A pop-up window will appear. Click on the "Choose File" button. Use the file browser that will appear to select your file(s). Click on the "Upload" button in the lower-right corner of the pop-up window.
- You are now back at "Submit Files - Phase5". Click on "Upload" in the lower-right of this window. You should now be at the "File Upload Results" page, and should see the message **`File Submission Successful`**.

D2L will send you a confirmation email after each item is placed in the Dropbox. This email is sent to your UA email (which is <yourUANetId>@email.arizona.edu). Retain these emails at least until you have received a grade for the assignment; it is your confirmation that you did submit the program(s).

You may submit phase5 more than once. We will grade the last one that you put in the Dropbox.

13.0 Groups

As advertised, you can change groups at this time is you wish. If you choose to do so, send mail to "452fall15@cs.arizona.edu" giving us details of how groups are being changed. You may use Piazza if you wish to advertise that you are looking for a new partner.