CS 1651: ADVANCED SYSTEMS SOFTWARE
FALL 2013

Project 3 – Swapping

Introduction

Now that you have implemented a basic virtual memory layer it's time to extend it. For this project you will be building on your previous work with page table creation and virtual memory management, and adding functionality to allow swapping pages to and from disk. In the first part of the project out of memory situations were considered to be hard failures, and resulted in allocation failures that propagated up to the application (which hopefully caught them....). For this project we will address the out of memory situation by incorporating a disk based backing store that can be used as swap space. The goal of this project is to enable the overcommitting of memory resources to an application by dynamically moving memory pages to and from disk as they are needed.

The context of this project will be the same as project 2. You will be implementing code inside a kernel module that interfaces directly to the file system. Your kernel module will be required to detect out of memory situations and react to them by triggerring a swapping operation that generates an available page by copying the contents of an in-use page to disk. Conversely, when a swapped out page is accessed you will need to react to this scenario by reading the page off the disk and copying its contents into an available in-memory page. **Note: We will not force you to deal with a total memory exhaustion scenario where there are no free pages either in memory or on disk.** If is up to you how you want to manage the swap space as well as which pages to select for eviction and/or replacement.

File I/O

For this project we have provided you with an interface for interacting with the file system from kernel space. This is generally a frowned upon practice, but for this project we will do it anyway to make things easier. The relevant functions can be found in `file_io.h`. The interface is very similar to the standard POSIX interface for file I/O, so the functions should be familiar to you. Files are opened and closed using `file_open` and `file_close`, reading and writing is done via `file_read` and `file_write`. All of these functions operate on a `struct file *` argument that points to a kernel representation of an open file. The other thing to keep in mind is that there is a limited amount of file state in kernel space, so you will not have access to things like an implicit location in the file. This means you will need to provide a byte offset whenever you wish to read or write to the file itself.

You will use this interface to open, read, and write a single file that acts as swap space for your virtual management layer. To allow portability you should use the filename !`/tmp/cs2510.swap`', and it is important to remember that you should always use absolute path names when you are in the kernel. Creating the file should be done before hand in user space. To do this you will need to use the '`dd`' command to create an empty file of a given size. For example:

```
dd if=/dev/zero of=/tmp/cs2510.swap bs=4096 count=256
```

creates a file with enough space to store 256 pages. For the evaluation we will be using swap files with varying size, so your code should be written to handle arbitrary file sizes.

Swapping

We have also provided you with a set of stub functions that will act as an interface to the swapping functionality of your module. The interface can be found in `swap.h` and `swap.c`. You will modify both files for this project, however changes to the interface should be confined to the swap space state struct. It is OK if you want to add debugging functions, but the current function prototypes should not be modified. One of the primary tasks for this project is providing the implementation for each of these functions. `swap_init` and `swap_free` should be used to initialize and deinitialize the swap state and should be called from your `on_demand.c` file. `swap_out_page` and `swap_in_page` will handle moving pages to and from the swap file. It is up to you to determine where in the swap file you will store the page, but the location you choose must be returned as the value of `index`. As part of this you will need to track the state of the swap file so that you will be able to find free page slots whenever they are needed.

Paging

The second part of this project will be connecting your swapping layer to your on demand paging implementation. For this to work you will need to detect whenever you have run out of available memory allocated to your module. When this occurs you will need to select a victim page to swap out to the swap file. It is up to you to select the appropriate page using one of the algorithms we went over in class (LRU, MRU, CLOCK, etc...). The available bits in the page table structures will likely be very helpful for this phase of the project. Once a page is selected you will need to write its contents to disk. Once it has been swapped out you will then need to swap in the page that the application is currently trying to access but is not resident in memory. Once the page has been read in, you will need to remap the page to the appropriate place in the page tables. All of the behavior in your module should be driven by page faults.

A helpful hint for this part of the project is to avoid the complexity of creating your own page tracking data structures. Remember that whenever a page table is marked as not present, then the rest of the 63 bits can be set to any random value. This means that the page table entries themselves may be used to store the state of a given page (such as its swap location). However you will need some way to ensure that you do not mistake a swapped out page table entry for one that has yet to be allocated. If you disregard this hint, and try to roll your own page tracking data structure you are probably going to be in for a world of hurt.

Submission

The submission of this project will be the same as the last one. Please email all of the modified source files to jacklange@cs.pitt.edu along with a README file detailing what works, what doesn't work, and any other information I should know. The user level testing program will still be used to evaluate the functionality of your code, so it is highly recommended that you create a robust set of tests and run them against your code before submission.

Start early, and good luck.