Assignment 3: Page Replacement

Daniel Urrutia (deurruti@ucsc.edu)
Justin Wong (jujwong@ucsc.edu)
Joshua Mora (jommora@ucsc.edu)
git pull git@git2.soe.ucsc.edu:classes/cmps111/winter16/deurruti asgn_3
git push git@git2.soe.ucsc.edu:classes/cmps111/winter16/deurruti asgn_3
CMPS 111, Winter 2016

Purpose:

Make meaningful comparisons (by collecting paging statistics) between slim-chance paging and FreeBSD's default paging algorithm (a relative of CLOCK algorithm). Analyzing the statistics we collect will allow us to determine impacts to system performance when running one paging algorithm over the other. The main changes we will be making to the current pageout daemon involve modifying the way pages are placed in their respective queues. Along with this, we will be modifying the way pages' activity counts are reduced after each scan.

Background:

The current system uses a relative of the CLOCK algorithm utilizing two page queues that are scanned in a series of 3 passes. Pass 0 involves placing the appropriate LRU/deactivated pages onto the active queue. Pass 1 involves moving inactive pages to either the cache or free list. Finally, pass 2 launders dirty pages (check reference bit to see if page has been modified) and the system "cleans" the page. Overall, the system uses its own paging statistics in order to maintain and keep track of the life cycle of a page (whether the page is on the active or inactive queues, or migrating between queues).

Hypothesis:

If we start modifying how pages are added to their respective queues and modify pages' activity counts (as specified in the assignment specification), we expect worse system performance. Specifically, we expect more page faults (as a result of halving activity counts) which means that pages on the inactive queue will have a higher chance of migrating to the cache and free lists. We expect this because moving pages to the rear of the free list means a page must wait longer to be freed, unnecessarily tying up more memory. By reducing the activity count by half, we're effectively allowing each page to remain active for a shorter period of time (meaning pages are moved to the inactive queue more often). This leads to more overhead related to moving pages between queues, resulting in worse overall system performance. Our modifications will be utilizing LRU for the specific purpose of making the system perform inefficiently.

Methodology:

We are adding the variables

int numPagesScanned = 0;
This variable tracks the total number of pages scanned by the pageout daemon. Updating this variable will happen inside of while loops when checking the inactive and active queues.

int numPagesMovedActiveToInactive = 0;
This variable will track the number of pages that are moved from the inactive list to the active list. In the loop that scans the active queue, there is a check at the end to see if a page has been referenced.

int numPagesMovedInactiveToFree = 0;
This variable will track the number of pages that are moved from the inactive list to the free list. There is a loop that scans the inactive list, there is a call to the function *vm_page_test_dirty()* which will check to see if the dirty bit is valid. If it is not dirty, the page is freed. The variable is updated here.

int numPagesMovedInactiveToCache = 0;

This variable will track the number of pages moved from the inactive list to the cache list. The code checks to see if a page needs to be freed and there's an else statement that checks if the page needs to be moved to the cache list instead. The else statement will be entered if a page needs to be moved to the cache list. This is where the variable is updated.

int numPagesQueuedFlush = 0;
This variable will track the number of pages to be queued for a flush. This means that when scanning the inactive list, we will check to see if the page is in laundry. The function that handles this is **vm_pageout_clean()**. This function returns the result of flushing the page. On success, that means that a page has been flushed, so this variable is updated.

int numPagesActive = 0;
Number of pages in active queue. Update this variable in the while loop, checking the pages of active list.

int numPagesInactive = 0;
Number of pages in the inactive queue, update this variable at the for loop that scans the inactive list.

Inside of the file *vm_pageout.c*, we added two function prototypes:
log_page_info(). This function will log general information about pages which is total number of pages scanned (**numPagesScanned, numPagesQueuedFlush**
The other function, **log_page_info_queue()**, will log information specific to pages being relocated (**numPagesMovedActiveToInactive, numPagesMovedInactiveToFree, numPagesMovedInactiveToCache, numPagesActive, numPagesInactive**).

To invert insertion into the inactive queue associated from moving from active to inactive, we will need to modify the function **vm_page_deactivate()** inside of the file *vm_page.c*.

The function **vm_page_enqueue()** inside of the file *vm_page.c* actually adds a page to the specified queue (**int queue**).
The function **vm_page_activate()** puts a page on active queue.
The function **vm_freelist_add()** in the file *vm_phys.c* adds a page to the free list. Pages that are added to the free list will only ever be coming from either the invalid or inactive lists.

There are 3 cases we need to cover:

1. Moving invalid and inactive pages to the REAR of the free list instead of the FRONT. This will be addressed inside of the function **vm_freelist_add())** in the file *vm_phys.c*. The modification necessary for this is to change the way pages are inserted by switching TAILQ_INSERT_HEAD to TAILQ_INSERT_TAIL.
2. Moving active pages to the FRONT of the active list instead of the REAR. This will be addressed in the functions **vm_page_activate()** and **vm_page_enqueue()**. The function **vm_page_activate** calls **vm_page_enqueue()** to actually add the page to the active queue. The modification in this function is to change TAILQ_INSERT_TAIL to TAILQ_INSERT_HEAD.
3. Moving pages to the FRONT of the inactive queue instead of the REAR. This will be addressed inside of the function **vm_page_deactivate()** in the file *vm_page.c* by changing TAILQ_INSERT_TAIL to TAILQ_INSERT_HEAD

After these modifications were made, it became necessary to test the performance between the two kernels. To look at our findings, please refer to WRITEUP.pdf where we present some graphical analysis. Special thanks to http://vmstax.michenux.net/ which allowed us to visualize our findings via graphs. In conclusion, our hypothesis was mostly verified. The modifications to the kernel did not become detrimental to the system until we started increasing the amount of memory given to each memory-creating process in the stress test command.