Assignment 3: Bon Chance Algorithm

Authors: Robert Sato, Keerthi Krishnan, Pranav Nampoothiri
Cruz ID: rssato, kvkrishn, pnampoot
Due Date: 05/19/2019

In our implementation of the Bon Chance algorithm, we simply used the tail queue data structure maintained by the pageout daemon, which supports all the five different lists that divide up main memory. In the assignment document, certain implementations for the Bon Chance Algorithm were specified and our group broke down the implementations to 5 main steps, making sure we conquered each step before moving to the next one. Here are the 5 steps that we broke the implementation down to:

1. Do not put invalid and inactive pages on the front of the free list. Instead, if the page number of the page is even, then add it to the front of the free list, and if it is odd, then add the page to the rear of the free list.
2. Do not set the activity count to 5. Instead, assign the activity count to a random number from 1 to 32. This should be implemented when the pages become first active.
3. Do not subtract from the activity count. Instead, divide it by 2, and subtract by 1. Move the page to the front of the active list, not the rear.
4. When moving to the inactive list, choose a random number from 1-10. If the random number generated is greater than 7, then the page goes to the front of the inactive list. If it is less than or equal to 7, then the page goes to the rear of the inactive list.
5. Run pageout more frequently in order to see much effect.

Free List Page Entry(Number 1 on Implementation List):

The first feature change we tackled was altering the method of page entries into the free list. To change the inactive and invalid pages from entering the front of the free list, to entering the front of the free list only when the page numbers are even, and entering the back of the free list otherwise, we had to locate where the fields to access the page numbers are. We figured out that in order to access page numbers, we would need to find the variable reference to the physical addresses of pages, to use accordingly, and thus determined that we would need to make the changes to the vm_freelist_add function in .c file, vm_phys.c. We assigned a page struct field, called "m->phys_addr," to a variable called "pageNum", and shift by 12 bits to get rid of the offset. This is because the physical addresses are 32 bits, and 10 bits are allocated for the Page Table Entry, while the other 10 bits are allocated for the Page Table Directory; the 20 bits in total represent the page Number. We then “and” this with 1, to get an odd or even number, and assign this to a variable called “finPageNum.”

Below the assignment of the integer variable, “order,” to the order field in the page struct, we structure our logic for adding pages to the free list. If finNumPage or the tail flag is true, we then create nested conditional, saying if finNumPage is true, meaning that the page number is odd, then we increment a “freelist_odd” counter, keeping track of how many odd pages we have, and then insert at the tail/rear of the free list, using the queue FreeBSD library function, TAILQ_INSERT_TAIL. Otherwise, we know the page number is even, so we increment a “freelist_even” counter, keeping track of how many even pages we have, and then insert at the head/front of the free list, using the queue FreeBSD library function, TAILQ_INSERT_HEAD. This is then the end of the outer if conditional, so if neither the finNumPage is true nor the tail flag is true, we again increment our “freelist_even” count since the page number would still be even, and then once again, insert at the head/front of the free list, using the queue FreeBSD library function, TAILQ_INSERT_HEAD.

Activity Count Replacement(Step 2):

The second implementation we worked on was to change the implementation of how FreeBSD sets it activity count to when a page first becomes active. When the page first becomes active, the page daemon implemented in FreeBSD sets its activity count to 5 by default and depending on its references, the activity is increased or decreased as such. The change involved setting the activity counter to a random value from 1-32 instead of setting it to the default value. We implemented this change in the file vm_page.c, specifically in the method vm_page_activate, where a page is first activated. We first check to see if the page’s activity count is less than the initial activity value. If it is, we then use a random number generator function, which was provided to us by Darrell Long, and we set a variable “random” equal to the output of the random function. We then take the modulus of the random value by 32 in order for the random number to stay within the range of 1-32. Finally, we set the page’s activity count equal to the random variable after all calculations are made. This insures that the we are indeed assigning a random variable to the activity count when a specified page becomes active, and that the random value will be within the range of 1-32, specified by the assignment document.

Activity Count Calculations and Placement(Step 3):

After accomplishing the second step, the third step was to modify the method in which the activity count is calculated. In the original FreeBSD pageout daemon, the activity count is updated by subtracting from the activity count. In the modified version, the assignment document says to implement as such: divide the activity count by 2 and subtract 1. Then, move the active page to the front of the active list and not the rear. We implemented this step in the file vm_pageout.c, specifically in the method vm_pageout_scan_active, as this method is where the active list is scanned and the activity count is increased or decreased based on the amount of times a page has been referenced. In the conditional that evaluates whether or not the “act_delta” variable is 0, we make the changes; the act_delta variable lets us know whether or not a page has been recently used, and if not evaluated to 0, *has* in fact been recently used. If it is 0, we implement the activity list entry condition, where we divide the page activity count by 2 (this is where the activity count was subtracted in the original version of the pageout daemon); we simply assign “m->act_count” with itself, divided by 2. Then, as long as the activity count is greater than 0 (this is a conditional just below the assignment stated above), we simply subtract m->act_count by 1, while also setting a subtract flag variable to 1.

Inactivity Count Calculations and Placement(Step 4):

We then came to the decision to add the inactivity count calculations in the same vm_pageout.c source file containing the pageout daemon within the same “vm_pageout_scan_active” function. We do this below the activity count calculation implementation, within the conditional checking whether the activity count is zero (in other words, if a page is inactive). We then implement the new inactive list entry calculation, where we choose a random number from 1-10 and if it’s greater than 7, we insert the page into the head/front of the inactive list, rather than the tail/rear. We simply created another random variable of type uint32_t, and set it equal to the output of random number generator. Like our change of the initialization of the activity count, we took the modulus of this new random number by 10 in order to stay within the range of 1-10, as per the specification of the Bon Chance Algorithm. If this modded random number is greater than 7, we set a random flag equal to 1 for use later, and increment a “random_count_greater” variable to keep track of how many pages are added to the head/front of the inactive list. Otherwise, we set the random flag to 0, and increment a “random_count_less” variable to keep track of how many pages are *not* added to the head/front of the inactive list.

After the section of code in the “vm_pageout_active_scan” function where the dirty pages are cleaned out, we set up a nested conditional. In order to insert the specified page at the head of the specified list, the subtraction flag must be true or the random flag must be true (in other words, they must evaluate to 1). If the subtraction flag is true, then the page is inserted at the head of the active list, whereas if the random flag is true, then the page is inserted at the head of the inactive list. Therefore, in the first outer if statement, we check the subtract flag and the random flag are true using an OR statement. If atleast one of them is true, we insert the page at the head of the respective list it belongs to. We implement active and inactive list insertion in the same segment of code (this nested conditional), because we assume that the pageout daemon takes care of the actual addition of pages to their corresponding lists, in this case, the active and inactive lists. Our specific placement of the calculations in the code, identifies which pages go in which list, and the parameters passed into the TAILQ_INSERT_HEAD FreeBSD tailq function, handle the actual placement of the pages into their according lists. Otherwise (relative to the outer if), we insert the page to its respective list and position in the list, using the TAILQ_INSERT_AFTER FreeBSD tailq function.

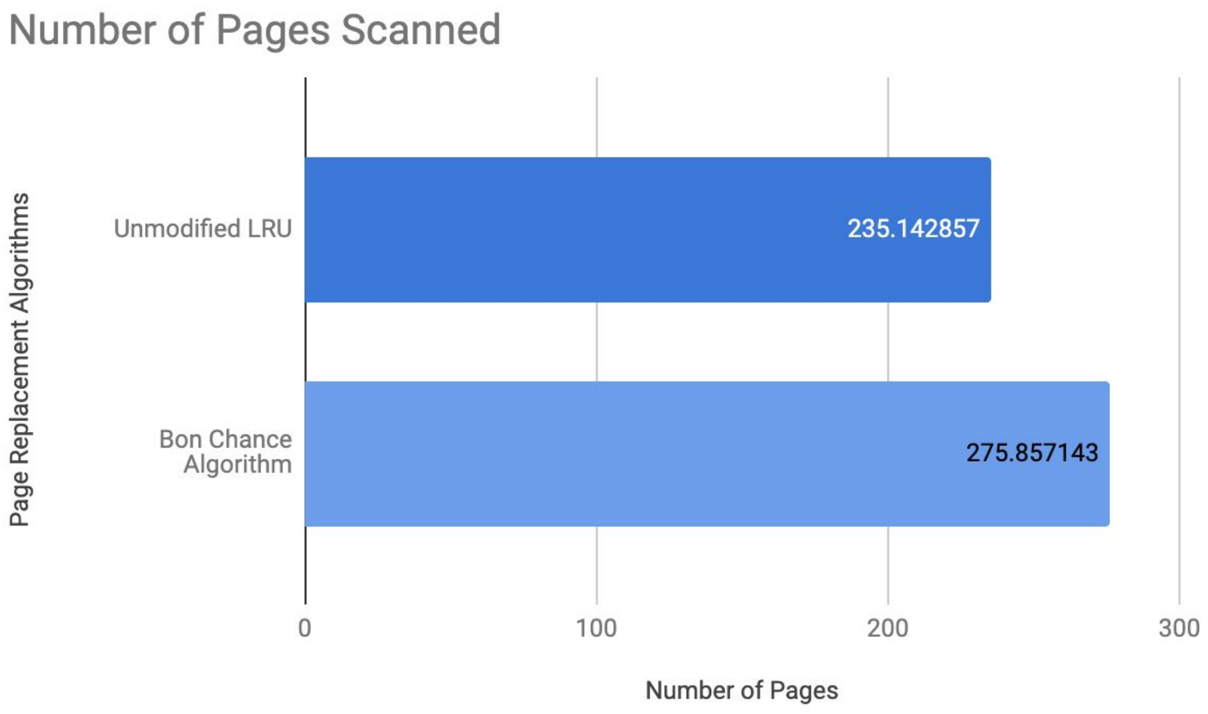
Running Pageout More Frequently(Step 5):

To run the Pageout Daemon more frequently, we looked through the pageout.c source file to find a defined field, setting a duration for every subsequent run of the pageout daemon. We discovered the “VM_INACT_SCAN_RATE” which was initially set for 10 seconds before the a subsequent run of the pageout daemon. We changed this value to 4, in order to have the pageout daemon run more frequently.

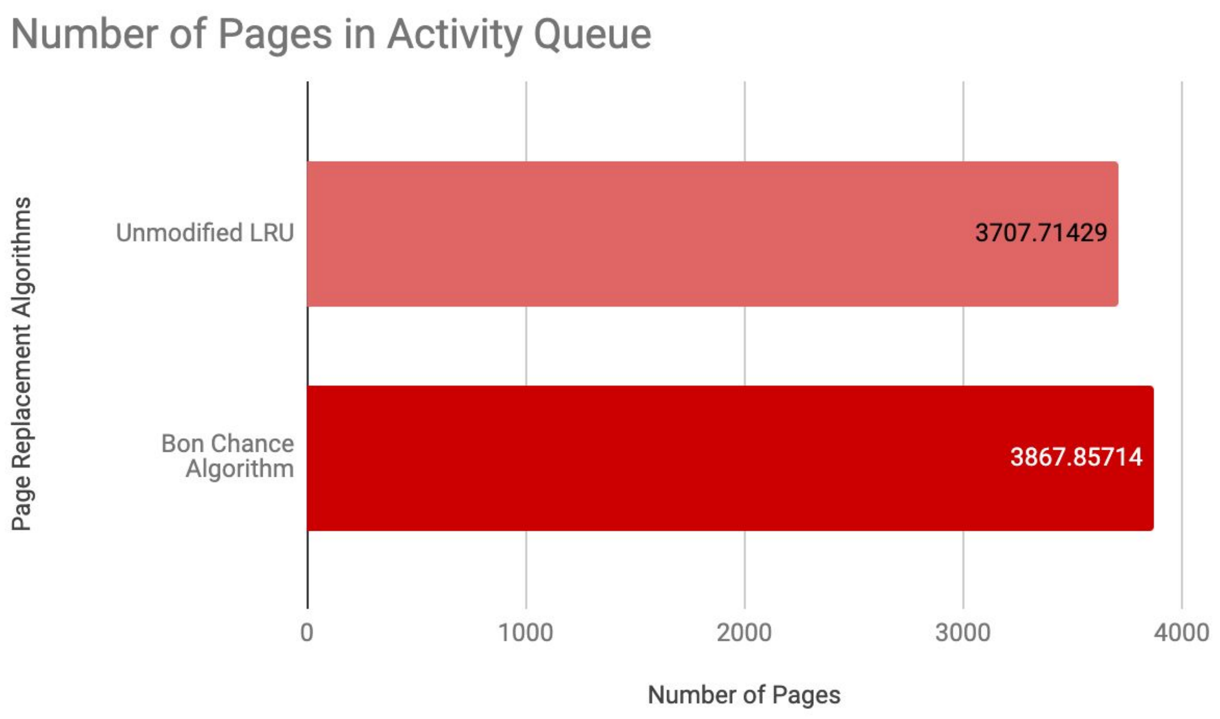
Benchmarks

After modifying and implementing all the changes specified in the assignment document, we used the stress test that the instructor told us to use, to compare both the original, unmodified paging replacement algorithm and the Bon Chance algorithm. We created a Makefile that built the kernel, ran the stress test, and printed out the data that was both from the original, unmodified algorithm, and the modified Bon Chance Algorithm.

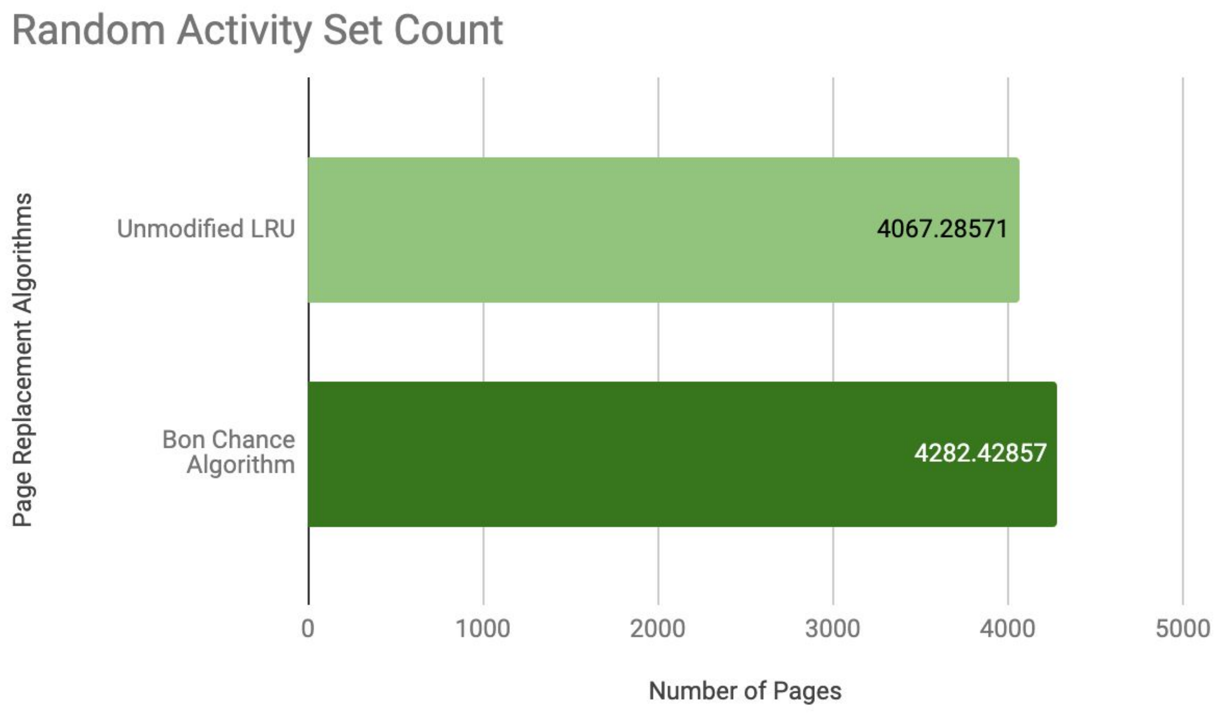
The common benchmarks and statistics we printed to the console were number of pages scanned, number of active pages, number of inactive pages, number of free pages, random activity set count, and number of pages in the wired list. We built and ran the kernel 7-8 times and averaged out each performance statistic for each algorithm, in order to get precise data. The graphs are shown below.



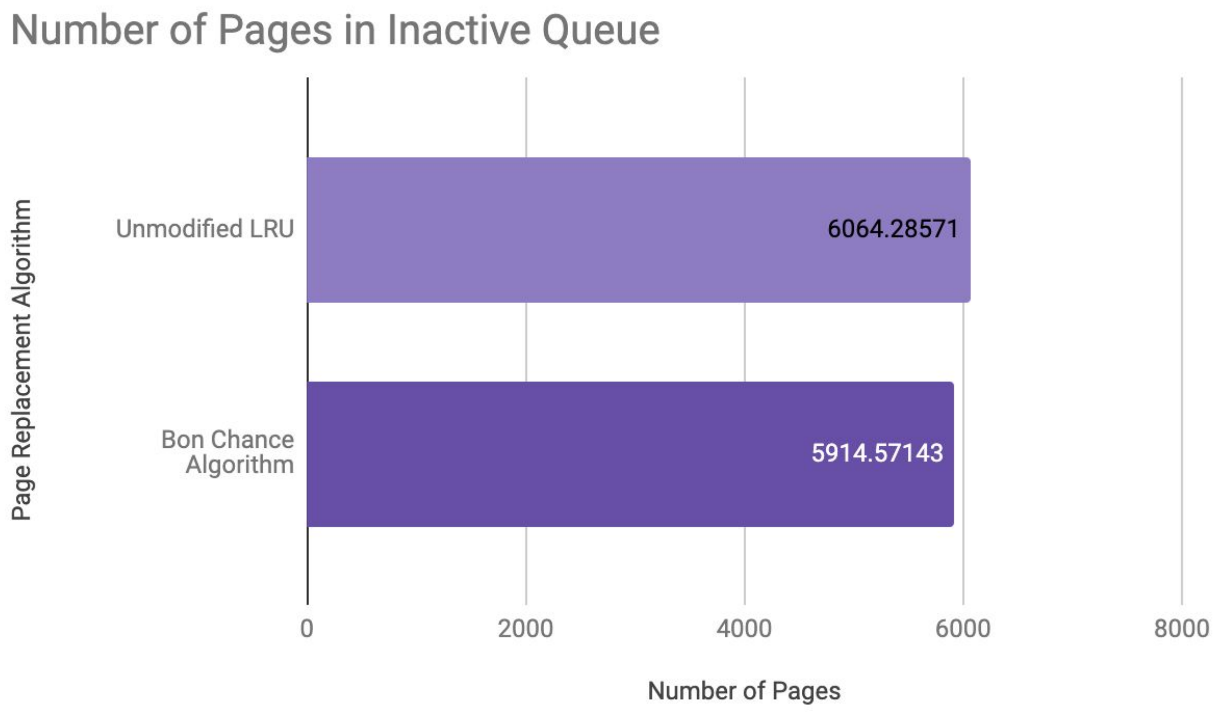
Graph 1: Number of Pages Scanned. This graph compares the average number of pages scanned between the unmodified algorithm and the Bon Chance algorithm. As you can see, the Bon Chance algorithm has a higher number of scans compared to the unmodified algorithm, which we think is due to the fact that we modified the scan rate in vm_pageout.c.



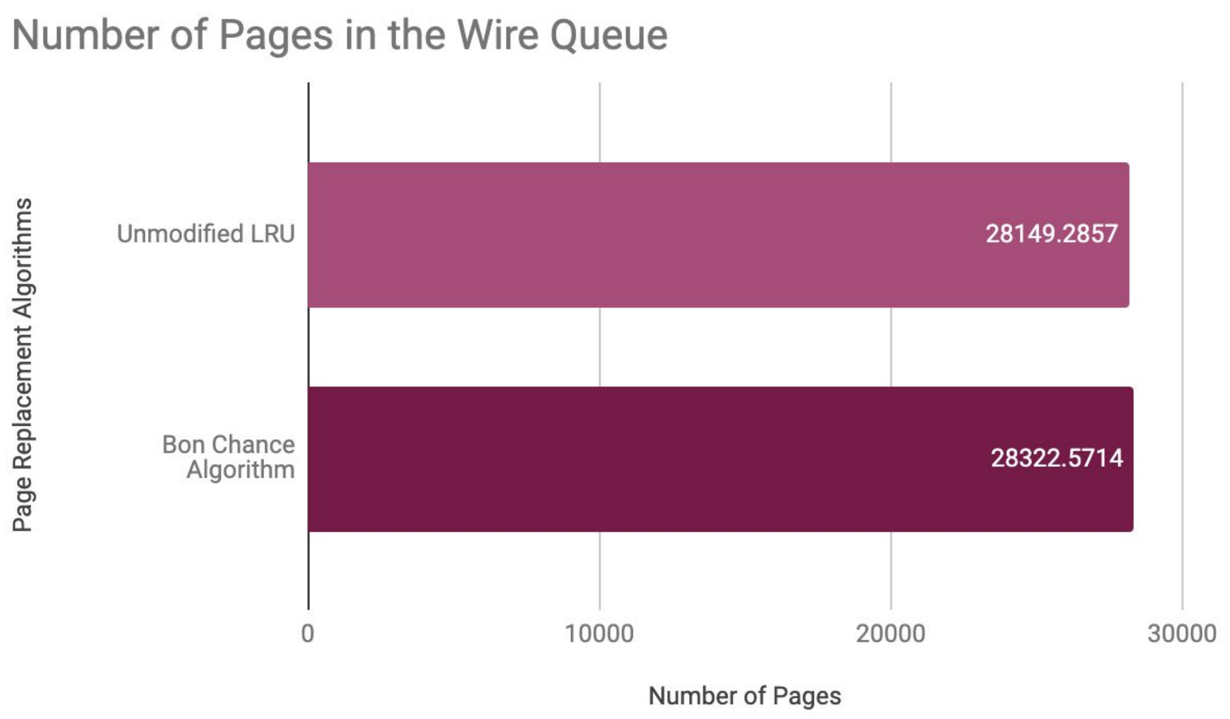
Graph 2: This is the Number of Pages in the Activity Queue. This graph compares the average number of pages in the active list between the unmodified algorithm and the Bon Chance Algorithm. As shown from the graph, the Bon Chance Algorithm has a larger number of pages in the active list on average, which we can deduce is because of the randomization of the initial activity count; this randomization seems to choose generally large numbers, which makes it a lot less likely for pages to be unreferenced enough to go to the inactive list, compared to the unmodified algorithm where it is a lot more likely for pages to be unreferenced enough to go to the inactive list.



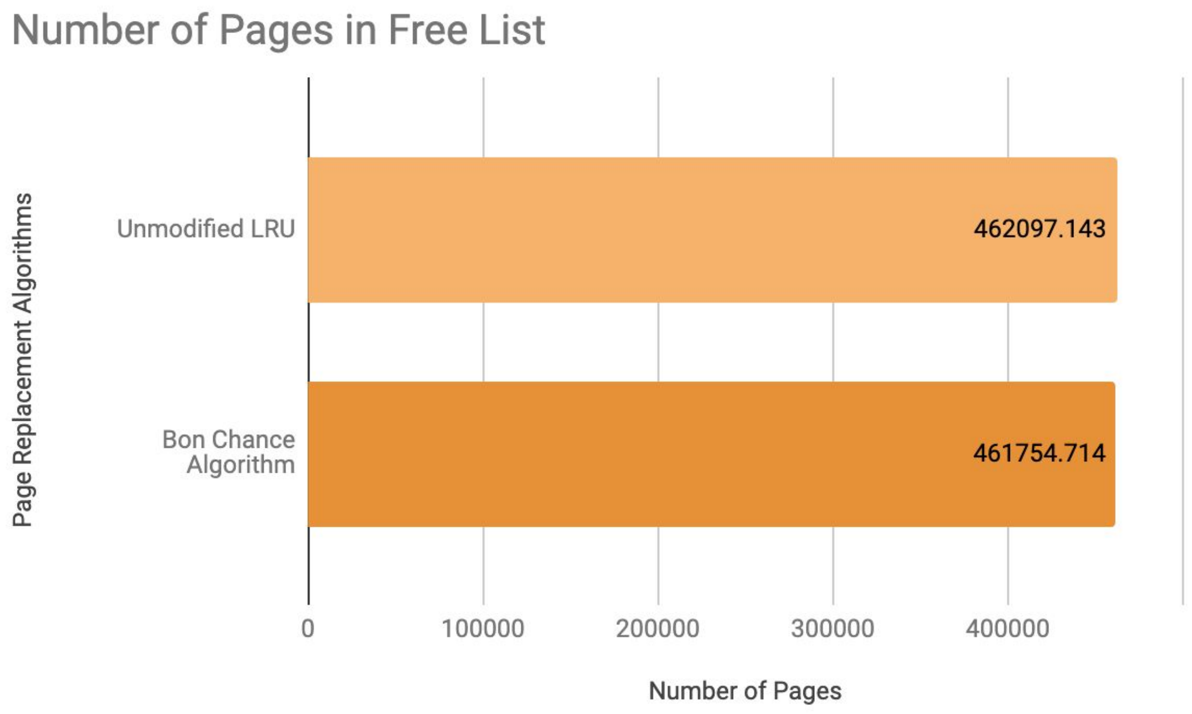
Graph 3: Random Activity Set Count. This graph compares the average random activity set count between the unmodified algorithm and the Bon Chance algorithm. As you can see, the random activity set count for Bon Chance is higher than the unmodified algorithm, as the Bon Chance for the most part is randomized with page replacement, therefore it tends to have a higher random activity set count.



Graph 4: Number of Pages in Inactive Queue: This graph compares the average number of pages in the inactive list between the unmodified algorithm and the Bon Chance algorithm. In the graph, it shows that the Bon Chance algorithm has less pages in the inactive list than the unmodified algorithm. This could be due to the random number generator, causing the number of pages in the inactive list to be lower. This also goes hand in hand with the number of pages in the active list, where the random number generator might be outputting higher activity count values, increasing the probability that a page might be in the active list, thus decreasing the probability of a page being in the inactive list.



Graph 5: This graph shows the number of pages in the wire queue on average between the unmodified algorithm and the Bon Chance algorithm. These averages are not very different between these two versions of the pageout daemon, since the implementation changes we made didn't alter the method in which pages are inserted into the wire queue.



Graph 6: Number of Pages in Free List: This graph compares the number of pages in the free list between the unmodified algorithm and the Bon Chance algorithm. As you can see, there is not much of a difference, however the Bon Chance algorithm has less pages on the free list than the unmodified algorithm. This could be caused by the random number generation when assigning activity counts and could cause pages with higher initial activity counts to have a higher probability of staying on the active list. However, the Bon Chance implementation in inserting pages into the free list should not have changed the number of pages in the free list that drastically, due to the fact that there is no randomization involved and the pages are inserted into the free list in only a slightly modified fashion.