Parker Newton

11-15-15
COEN 177L W 2:15 pm
Lab 3

Overview:
The goal of this assignment is to model the design, implementation, testing, and performance analysis of several page replacement algorithms, namely second chance, least recently used, least frequently used, and clock. The source files are included in their respective directories in src/. A discussion of each of the four page replacement algorithms follows.

Second Chance:
Design:
The second chance algorithm functions as follows. When a new page request is made, if the page is found in the cache, then we simply set its reference bit to 1. Finished. Else, we must select a page to evict. We top our queue, and if its reference bit is 0, then we dequeue it, and enqueue our newly requested page. Else if its reference bit is 1, then we dequeue it, set its reference bit to 0, and enqueue it. We repeat this process until we find a page with a reference bit of 1, and proceed as stated above. Finished. For the second chance mechanism, we will implement a queue ADT as a circular doubly linked list. We will need to support the enqueue, dequeue, top, find, and set reference bit operations.

Implementation:
Each node in the queue contains a data value, reference bit, and pointers to the next and previous nodes. The queue struct contains a value for its size and a sentinel node that points to the queue's head. In the main run loop, call initSecondChance() and tearDownSecondChance() to initialize and deallocate, respectively, the second chance mechanism. The following pseudocode illustrates the workflow of a page request according to the second chance mechanism.

```
def secondChance( pageId ) :
        if pageId is in our cache :
                set pageId's reference bit to 1.
        else :
                while top of queue's reference bit is 1:
                        temp = dequeue ()
                        enqueue ( temp )
                dequeue ()
                enqueue ( pageId )
        end.
```

Performance:
Each time we make a page request, we must perform a linear search of the cache to determine if the page is already in memory. This costs us O(n) time, where n is the size of the cache. If a page fault occurs, then we must choose a page to evict. At best, the initial top of the queue has a reference bit of 0, so we can dequeue it and enqueue our newly requested page in O(1). However, at worst, all of the reference bits in the queue are set to 1, so we must dequeue, set the reference bit to 0, and enqueue every node in O(n) time. On average, we will assume the worst case time complexity, so we have an expected O(n) time complexity for second chance.

Compilation and Execution:
In the sc/ subdirectory of src/, run the following commands to compile, link, and execute the program.

$ make
$ cat <input_file> | ./sc <cache_size>

Least Recently Used:
Design:
The least recently used algorithm functions as follows. We maintain a global counter of the number of page requests made. When a new page request is made, increment this counter. If the page is in our cache, then set its timestamp equal to our global counter. Else, if our cache is full, then evict the page with the lowest timestamp value (this is the least recently used page). Insert our newly requested page into the cache, and set its timestamp value equal to our global counter. We must support insert, erase, find minimum, find element, and set counter on element functionality. Also, the order of the pages does not matter, and our cache is fixed-size, so we will implement a set ADT as an array of node pointers.

Implementation:
Each node in the set contains a data value and timestamp. The data value is the page number, and the timestamp is set to the global counter each time a reference is made. The set struct contains maximum size and instantaneous size variables, and a pointer to an array of node pointers. Call initLRU() and destroyLRU() to initialize and deallocate, respectively, the least recently used mechanism. Below is the pseudocode for the least recently used mechanism.

```
def leastRecentlyUsed( pageId, &counter ):
        counter++;
        if pageId is in our set:
                set its timestamp to counter
        else:
                if our set is full:
                        temp = node with min timestamp value
                        erase (temp)
                insert (pageId)
                set its timestamp to counter
        end.
```

Performance:
Each page reference, we must check if our requested page is in our cache with a linear search in O(n) time. Each page fault, we must find the minimum element to evict with a linear search for a minimum value in O(n) time. Due to the random access of an array implementation, we can insert at the end in O(1) time. Also, because of the unordered nature of the set, we can erase an element by overriding it with the last element in the array. We have an expected time complexity of O(n) for least recently used.

However, we can perform an optimization for least recently used. Because of the random access property of arrays and the unordered property of sets, we can optimize the findElement

and findMinElement operations. For finding an element, we can apply a quicksort to sort the elements in increasing order, and apply binary search in O(log n) + O(log n) = O(log n) time. For finding the minimum timestamp, we can once again apply a quicksort, with the timestamp of each node as a sort key, to sort the elements in increasing order in O(log n) time, and access the minimum value at the front of the array in O(1) time for a time complexity of O(log n). Thus, with this optimization, we can support an expected runtime of O(log n).

Compilation and Execution:
In the lru/ subdirectory of src/, run the following commands to compile, link, and execute the program.

$ make
$ cat <input_file> l ./lru <cache_size>

Least Frequently Used:
Design:
The least frequently used algorithm functions as follows. We maintain a global counter of page references. On every page request, we increment our global counter. If the page is in our cache, then we simply increment its frequency count and assign its timestamp the value of our global counter. Else, if our cache is not full, then we insert the page into our cache, increment its counter, and assign our global counter to its timestamp. Otherwise, if our cache is full, then we evict the page with the lowest frequency count, and, in the event of ties, the lowest timestamp value (ties are resolved by LRU), insert the newly requested page, increment its frequency count, and assign our global counter to its timestamp. We must support all of the same operations as in least recently used, and also like least recently used, the order does not matter and the cache is fixed-size. Thus, we will implement a set ADT as an array of node pointers.

Implementation:
Each node in the set contains a data value, counter, and timestamp. The data value is the page number, the counter is the frequency value, and the timestamp is the field that is used to resolve ties between frequency counts (by LRU, as stated above). The set struct contains maximum size and instantaneous size variables, and a pointer to an array of node pointers. Call initLFU() and destroyLFU() to initialize and deallocate, respectively, the least frequently used mechanism. Below is the pseudocode for the least frequently used mechanism.

```
def leastFrequentlyUsed( pageId, &counter ):
        counter++;
        if pageId is in our set:
                increment its frequency count and set its timestamp to counter
        else:
                if our set is full:
                        temp = node with min frequency or same frequency and min timestamp
                        erase (temp)
                insert (pageId)
                increment its frequency count and set its timestamp to counter
        end.
```

Performance:
Each page reference, we must check if our requested page is in our cache with a linear search in O(n) time. Each page fault, we must find the minimum element to evict with a linear search for a minimum value in O(n) time. Due to the random access of an array implementation, we can insert at the end in O(1) time. Also, because of the unordered nature of the set, we can erase an element by overriding it with the last element in the array. We have an expected time complexity of O(n) for least frequently used.

However, just as with least recently used, we can optimize least frequently used by applying a temporary order. For searching, we can apply a quicksort and binary search in O(log n) time. For finding a minimum element, we can also apply a quicksort to sort in ascending order, with the frequency count as the sort key. However, in the event of frequency ties, we must also look at the timestamp field. So, after we sort, we must iterate over all the nodes at the beginning of the set with equal minimum frequencies, and compare their timestamp values to find the minimum. At best, there are no ties to break, in which case we have O(log n) time. At worst, each node has an equal frequency count, in which case we have O(n) time. On average, we expect k nodes to have the same minimum frequencies (with $k < n$), for which we have O(k) time. Overall, for least frequently used with this optimization, we would have a time complexity of O(log n) + O(k) = O(k) for each page fault.

Compilation and Execution:
In the lfu/ subdirectory of src/, run the following commands to compile, link, and execute the program.

```
$ make
$ cat <input_file> | ./lfu <cache_size>
```

Clock:
Design:
The clock algorithm is as follows. Begin with the clock “hand” pointing to some page in the cache. On each page request, we check the value of the reference bit of the page which the hand points to. While it is 1, we set it to 0 and increment the hand to the next page. When we find a reference bit of 0, we evict that page, insert the newly requested page into this location, and increment the hand. Note that the functionality of clock is the exact same as that of second chance, and the only differences lie in its implementation. Clock is implemented as an array of node pointers, with a “hand” pointer to the node in the array which represents the top of the queue.

Implementation:
Each node contains a data value (the page number) and a reference bit. The queue struct contains a maximum size, instantaneous size, a pointer to an array of node pointers, and a hand pointer. Call initClock() and destroyClock() to initialize and deallocate, respectively, the clock mechanism. Below is the pseudocode for the clock mechanism.

```
def clock( pageId ) :
        if pageId is in our cache :
                set pageId's reference bit to 1.
        else :
                while top of queue's reference bit is 1:
                        temp = dequeue ()
                        enqueue ( temp )
                dequeue ()
                enqueue ( pageId )
        end.
```

Performance:
The clock algorithm must first traverse the entire structure to determine if the requested page is already in memory in O(n) time. On each page fault, as in second chance, dequeue and enqueue until the top of our queue contains a node with a reference bit of 0. As explained in second chance, this costs us at best O(1) and at worst O(n) time.

A performance optimization that would require a little more space overhead and code complexity would be for each node to contain the index of the next and previous nodes in the queue. This is how the hand pointer would traverse the list, instead of incrementing sequentially. Then, we could apply a quicksort for searching in O(log n) time. However, page faults still maintain the same time complexity, as the case where all nodes contain a reference bit of 1 degrades to classical FIFO. The downside of FIFO is that pages that are more frequently requested might be swapped out as opposed to those that are less frequently requested. A solution to this might be to increase the number of "chances" each page is allocated. Instead of a second chance policy, maybe a third or fourth chance policy would optimize this algorithm. The concept of a reference "bit" would become bits (for values greater than 1, though an unsigned char representation would still suffice). However, in the case where all of the nodes have the same non-zero reference bit, we might have to traverse the entire list multiple times before arriving at a page to evict (whereas with second chance the maximum amount of nodes to traverse before finding a page to evict is n).

Compilation and Execution:
In the clock/ subdirectory of src/, run the following commands to compile, link, and execute the program.

```
$ make
$ cat <input_file> l ./clock <cache_size>
```

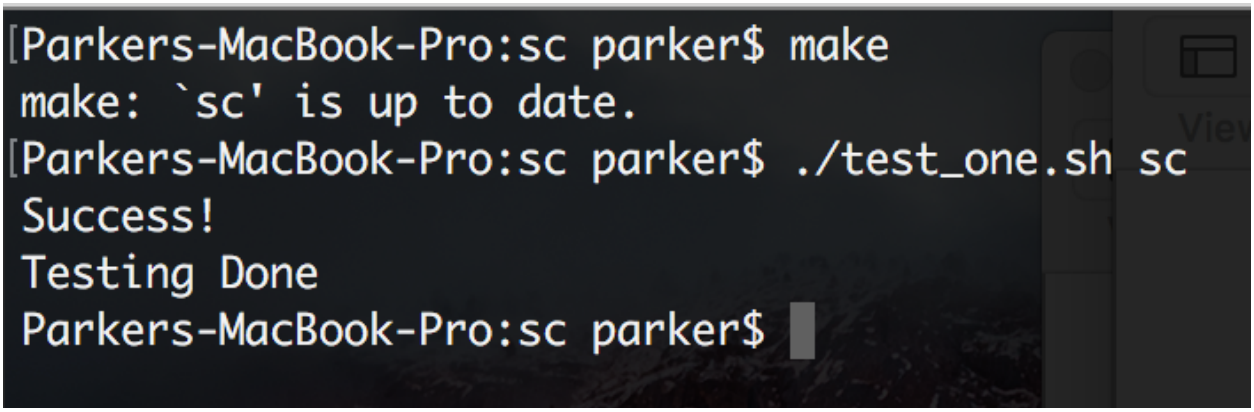
Testing:
For each page replacement algorithm, we will test its performance (in terms of number of page faults as a function of cache size) against a constant input stream (1000 page requests) for cache sizes of 10, 50, 100, 250, and 500. The input steam of page requests is in the file "numbers.txt" (included in the same directory as each page replacement algorithm. The provided test script "test_one.sh" will be used to test the page replacement implementations. It takes the program output (number of page faults, number of total requests) and compares it with the hardcoded values in the file "x_counts.txt", where x is the name of the page replacement

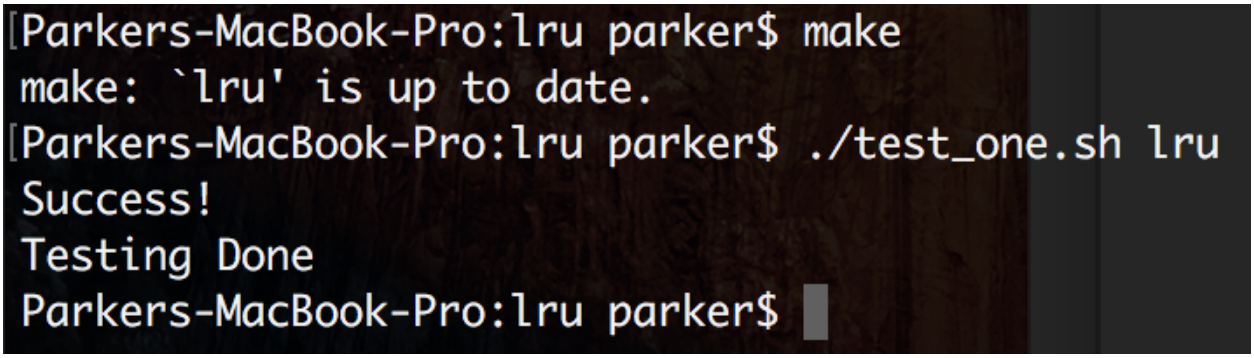
algorithm (sc, lru, lfu, or clock). In the directory for each page replacement algorithm, a copy of the files test_one.sh, numbers.txt, and x_counts.txt is included. In order to run the tests, in each page replacement algorithm's respective directory, enter the following commands on command line (where x is the name of the algorithm—sc, lru, lfu, or clock).

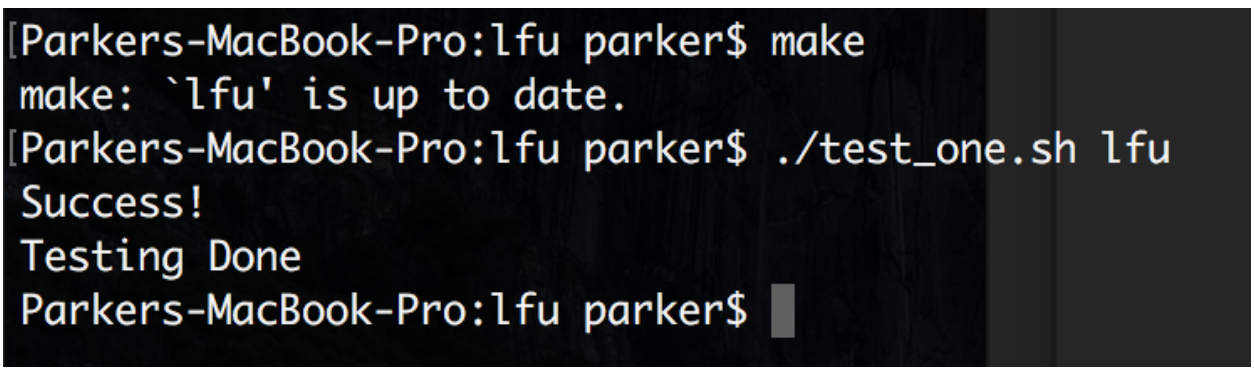
$ make
$ ./test_one x

Note that the path directory to each page replacement algorithm in test_one.sh may need to be updated.

Below are screenshots of the output from test_one.sh for each page replacement algorithm.





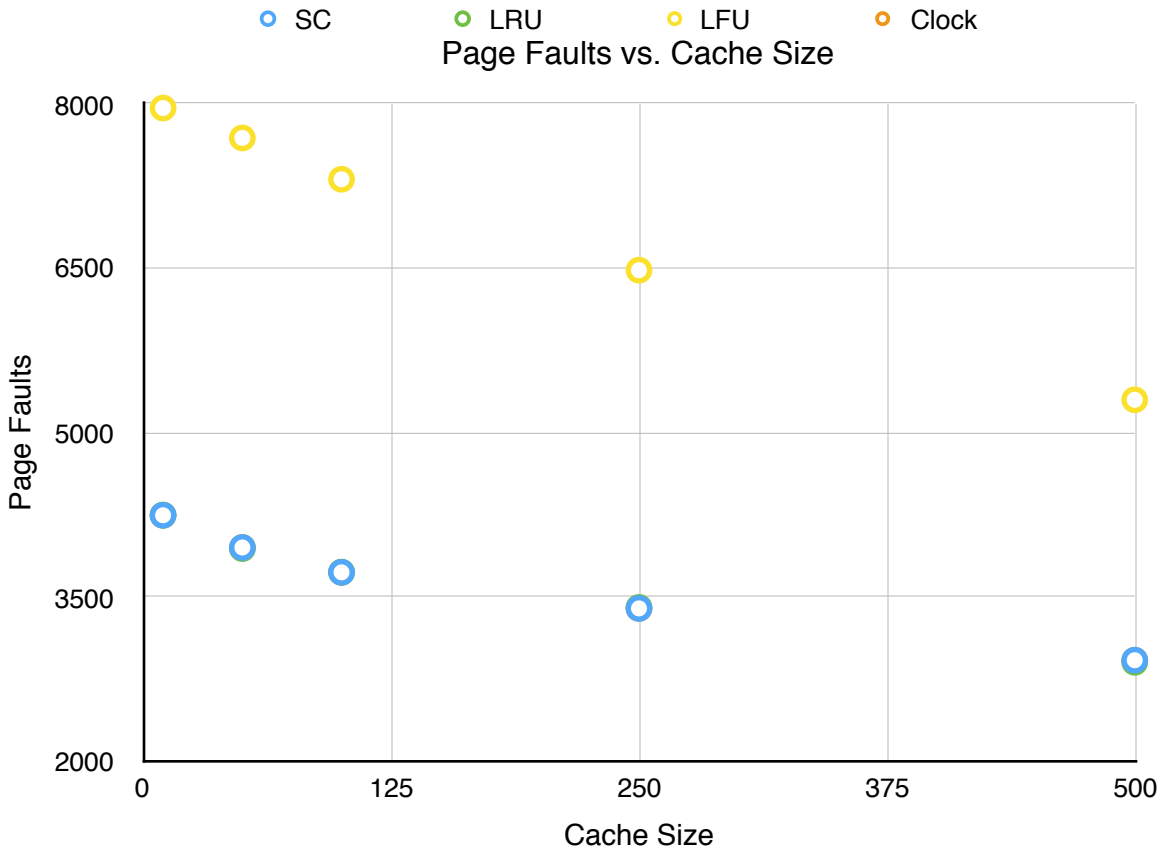


```
[Parkers-MacBook-Pro:clock parker$ make
 make: `clock' is up to date.
[Parkers-MacBook-Pro:clock parker$ ./test_one.sh clock
 Success!
 Testing Done
 Parkers-MacBook-Pro:clock parker$
```

Below are the results of the performance of each page replacement algorithm in terms of the number of page faults as a function of the cache size in both tabular and graphical forms.

Page Faults vs. Cache Size

	SC	LRU	LFU	Clock
X	Y	Y	Y	Y
10	4246	4248	7963	4246
50	3952	3942	7690	3952
100	3725	3727	7313	3725
250	3396	3405	6483	3396
500	2923	2904	5300	2923



Second chance and clock have the same functionality, and thus the same performance in terms of the number of page faults. However, between second chance and clock, both have the same time complexities for page faults. In terms of code complexity, though, clock has a slightly simpler implementation. Also, from this data, LRU appears to perform very closely to second chance and clock, but can be optimized to give it an expected time complexity for page faults of O(log n). In terms of selecting the most optimal page replacement algorithm from this data, two candidates seem to be clock and LRU.