

Page Replacement Algorithms

Group Members:

N.V. Navaneeth
15115086

Chirayu Asati
15115040

Deepak Verma
15115045

Rishabh Jain
15111033

Introduction

Virtual memory has boosted the degree of multiprogramming and also enabled systems to run processes with memory requirements larger than the actual physical memory. This is by only loading only the required pages into the physical memory and keeping the rest in the disk. The pages stored in the disk are swapped into the main memory when required. When main memory is full, a page already in the main memory will need to be replaced. We need to choose the victim page carefully, since loading a page from the disk to the memory has considerable overhead, and hence we require all the frequently accessed pages to remain in main memory. It is clear that the policy used for replacing the pages can considerably affect the performance of the system.

There are various page replacement algorithms: FIFO, the Optimal algorithm, Least Recently Used (LRU), Clock algorithm/Second chance algorithm (An approximation to LRU), etc. In this assignment, we are concerned with only the various implementations of the LRU and the clock algorithm, which are discussed later.

Goals

1. To Implement the LRU-Counter, LRU-Stack, LRU-Aging register and Clock algorithm in C/C++ without using existing/predefined classes
2. To compare the page fault rate for each of the algorithms.(Consider corner cases)
3. To compare the runtime of each algorithm. (Consider corner cases)

Theory

Details of the different page replacement algorithms of interest are discussed in this section. The Least Recently Used (LRU) page replacement algorithm, as the name suggests, finds the page that was least recently accessed and replaces it. There are different implementations of the LRU algorithm. namely:

LRU-Counter:

In this implementation, there is a counter variable associated with each page that is stored in the main memory. Every time a page is accessed, we copy the current clock into the counter of that page. Hence, to identify the least recently used page, we search through all the pages, and identify the one with the smallest counter value, this will be replaced.

LRU-Stack:

In this implementation, references to all the pages stored in the memory are inserted into a stack. Whenever a new page arrives, it is inserted onto the top of the stack. Whenever an existing page is referenced, it is removed from its position in the stack and reinserted into the top. This means that the least recently used page will always be at the bottom of the stack. When a page is to be replaced, we replace the last page at bottom of the stack.

LRU-Aging register:

In LRU aging register implementation, corresponding to every page loaded into memory, there is an n-bit register (initialized to 0). Whenever a new page arrives or an existing page is referenced, the Most Significant Bit of the register is set to 1. Furthermore, every tick interval, the registers of all the pages are shifted to the right by 1 bit (i.e its value is decreased). To find the least recently used page, all of the pages are searched, and the one with the least register value (contents of the register treated as an unsigned int) is identified.

Clock Algorithm:


Clock algorithm is an approximation of the LRU algorithm. In this, associated with every page is a bit called the second chance bit (initialized to 0). All the pages are stored in a circular array or list. There is also a clock pointer which points to an entry in the list (Initially points to the first entry in the list). Whenever a page is accessed, its second chance bit is set to 1. When a page is to be replaced, we start with the page to which the pointer is pointing and move clockwise (the pointer is also moved). At each page, we check if the second chance bit is set to 1 or not. If its is set, then it is reset to 0, and move to the next page (i.e. we give this page a second chance). If it is zero, then that page is replaced.

Implementation details.

LRU-Counter : Instead of storing the clock at the time of accessing, a global counter is kept, and at each page access, this counter is updated and its value is stored with the page that is being accessed.

LRU-Stack: A custom stack, designed to the needs of the LRU-stack algorithm was implemented. It has functions for: Inserting to the top in $O(1)$, Deleting from the bottom in $O(1)$, Searching and re-inserting a page to the top in $O(n)$

LRU-Aging register: An uint variable was associated with each page to represent a 32-bit aging register. Instead of updating the registers periodically (asynchronously), at the time of each page access, some amount of time (randomly generated) is assumed



to have passed since the previous access. From this generated delay time we can calculate how many clock ticks would have happened within this period, and this is used to update the ageing registers before the page access algorithm begins.

Clock algorithm: A normal array is used to store the pages with an associated boolean array to represent its second chance status.

Testing details

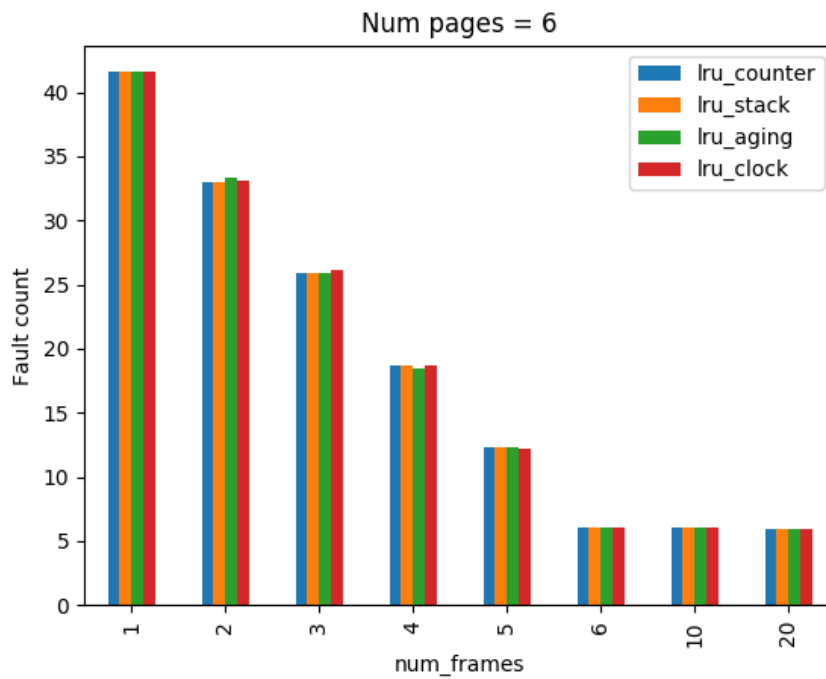
- For each run, the number of pages and the number of frames were selected from the set of values {6, 10, 15, 30} and {1, 2, 3, 4, 5, 6, 10, 20} respectively.
- For each combination of (num_pages, num_frames), the average page fault count and the algorithm run time were calculated. A randomly generated sequence of 50 pages were used for one run, and 100 such runs were used to obtain the average values.
- For corner cases, the direct sequence (eg: 1,2,3,1,2,3,1,2.....) and the direct-inverted(eg: 1,2,3,3,2,1,1,2,3.....) sequences were used. For each of them the number of frames were 4 and number of pages were 6.

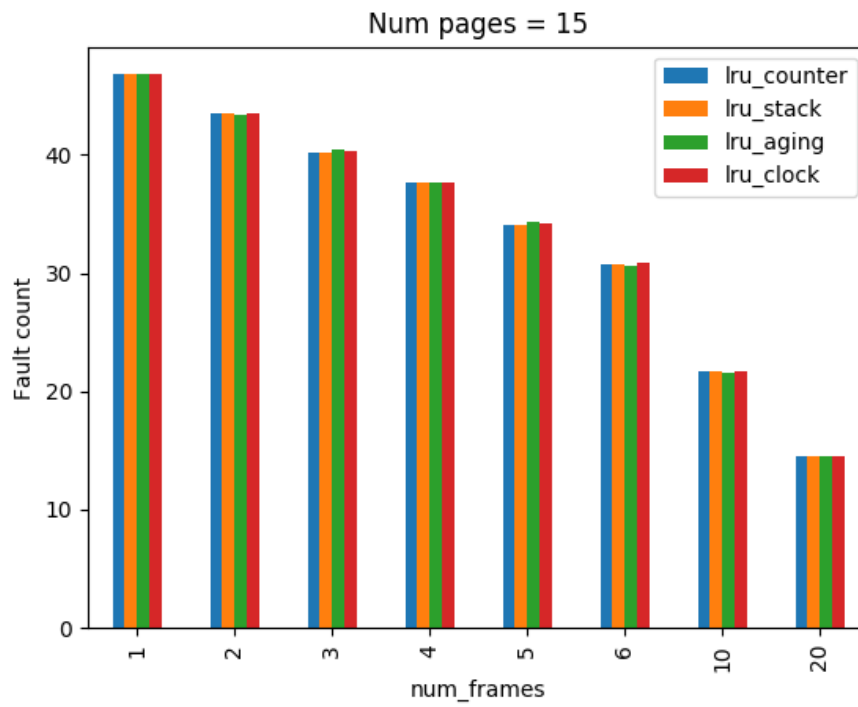
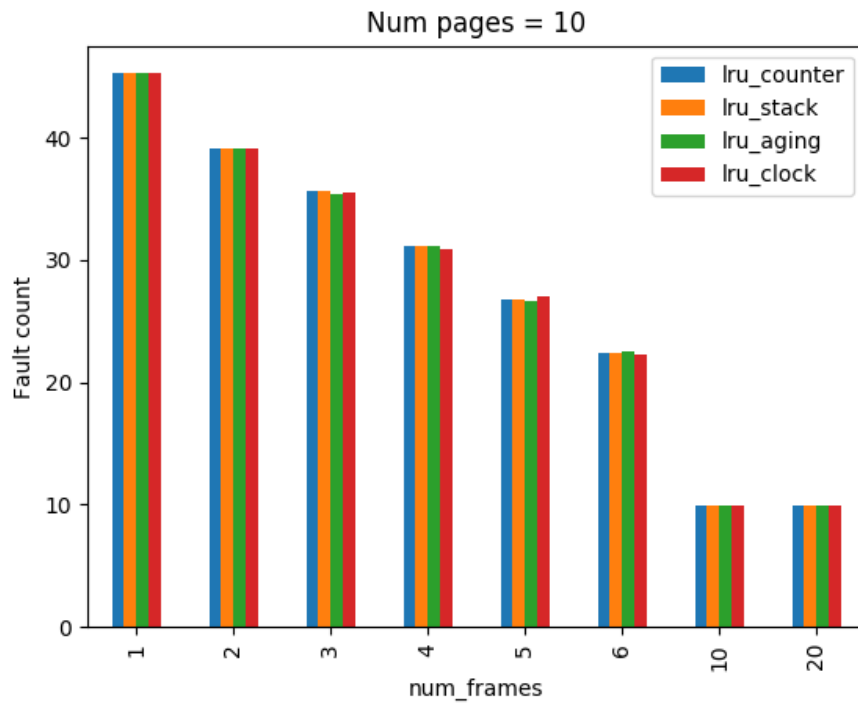
The results for each test case was written to a csv file, and later plotted using python.

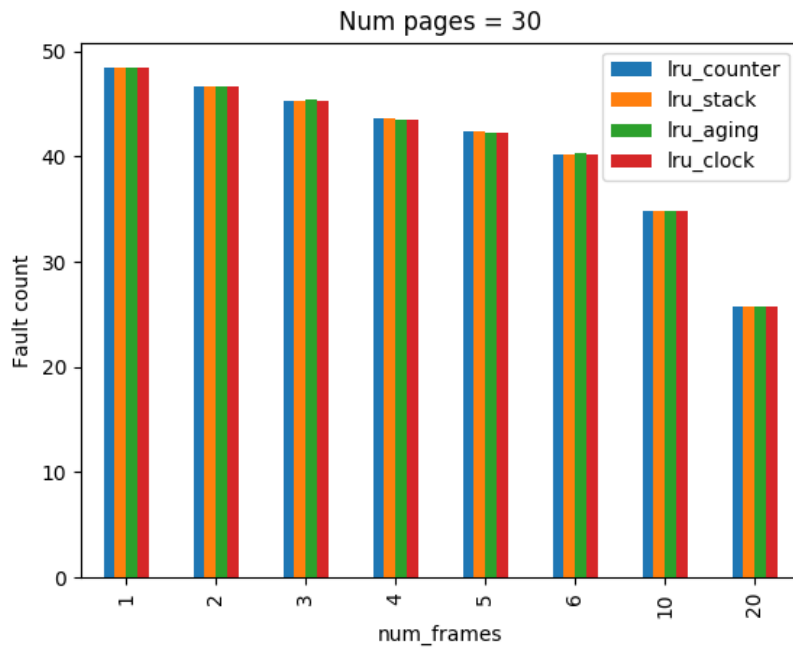
Results and Analysis.

The results obtained for the test cases were plotted. The following graphs were obtained.

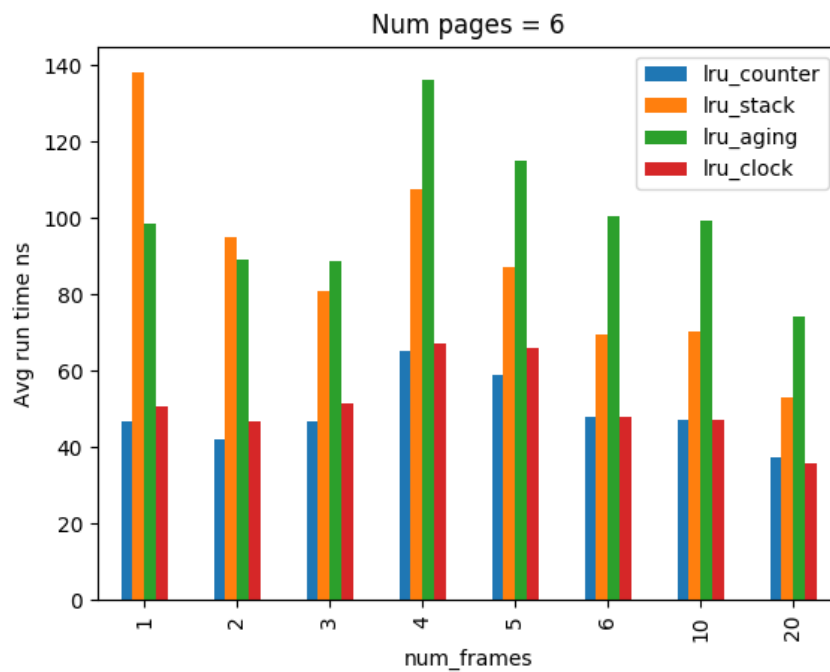
Plots of Fault count vs Number of frames, for different values of Number of pages

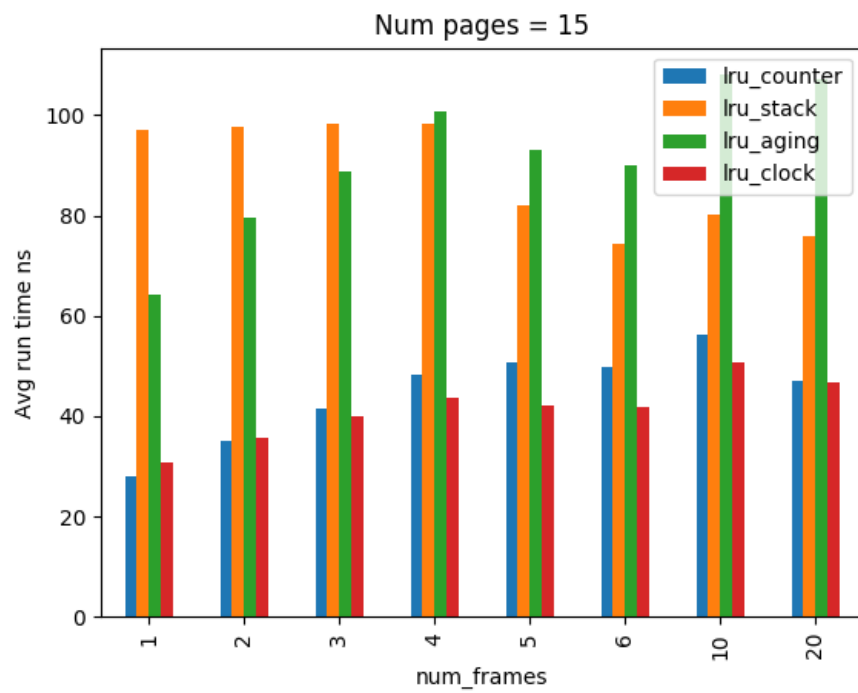
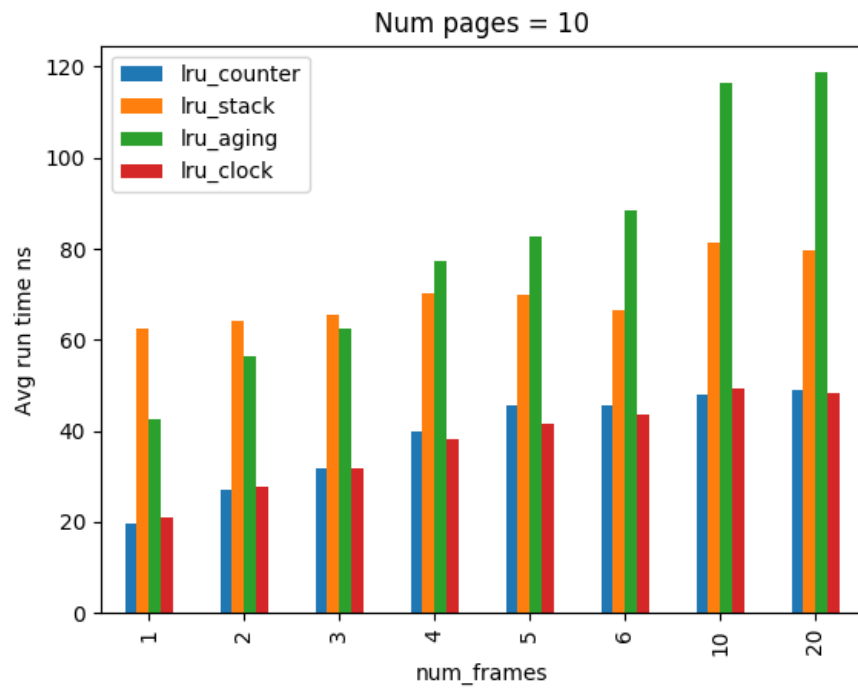


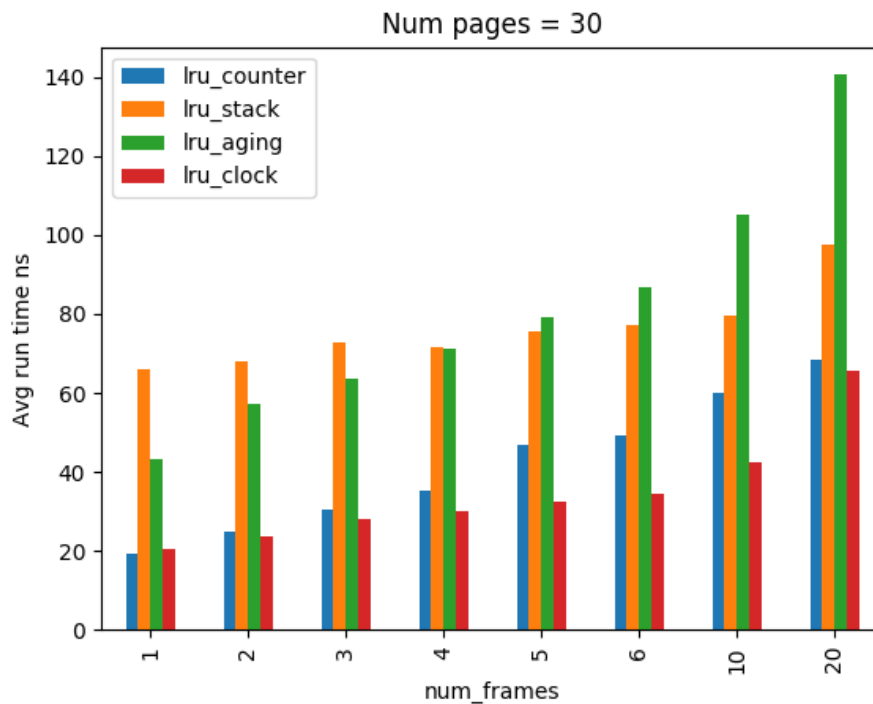




Plots of Avg run time (ns) vs Number of frames, for different values of Number of pages







Corner cases:

```

Corner case: Direct sequence
num_pages = 6 num_frames = 4
LRU Counter: fault_count = 50, avg_run_time_ns = 51
LRU Stack: fault_count = 50, avg_run_time_ns = 120
LRU Aging: fault_count = 46, avg_run_time_ns = 126
LRU Clock: fault_count = 50, avg_run_time_ns = 46


Corner case: Direct inverted sequence
num_pages = 6 num_frames = 4
LRU Counter: fault_count = 23, avg_run_time_ns = 43
LRU Stack: fault_count = 23, avg_run_time_ns = 86
LRU Aging: fault_count = 33, avg_run_time_ns = 114
LRU Clock: fault_count = 32, avg_run_time_ns = 53

```

Analysis

Fault Count:

As expected, the number of page faults decrease monotonically with increase in number of frames, irrespective of the number of pages.



Also as expected, the LRU-Stack and LRU-Counter implementations always produce the same number of page faults. LRU-Aging differs at times, because during the same tick interval, multiple pages can be accessed and the MSB of all of those pages would be set to 1. Also, unlike LRU-Stack and LRU-Counter, it stores information of previous accesses well, and it will influence the selection of the pages when they are considered for replacement.

Clock algorithm performed nearly as good as the exact implementation, matching their fault count most of the time. On average, (average over 100 random sequences), number of page faults of Clock algorithm was only 0-1 faults above the exact implementations.


Corner cases: Direct sequence: all the algorithms performed badly, with high number of page faults, since, in this sequence, the least recently used page is the one accessed again. LRU-Aging performed slightly better, again, because multiple pages have the possibility of having the same register value, and in our implementation, the replacement strategy in such a case was to remove the first page in the list (i.e not FIFO), which lead to old pages remaining in some cases, and leading to less number of page faults.

Direct Inverted sequence: This is the best case scenario for LRU algorithms, and hence produced the least number of page faults possible. Since clock algorithm doesn't exactly remove the least recently used, it produced worse results.

Running time:

The running time of LRU-stack and LRU-ageing seems spuriously high in the graphs. This is probably because they include overhead additional to searching for the page in the list. For stack, we need to re-insert into the top of the stack, and for aging, we need to periodically shift the registers (since in our implementation, they are not actual registers, rather unsigned int, hence manipulating them takes more time than manipulating registers). While on the other hand, for counter, we just need to copy a variable and for clock, we just need to set the second chance bit.

We see that, with increasing number of pages, the increase in runtime for exact implementations are more than that of the approximate implementation: this is because, for LRU-aging and LRU-counter, they have to do a full search of the list of pages before removing one. But for Clock algorithm, the entire list need not be searched for finding the page to replaced, hence it scaled better with number of frames.



Corner cases: The run times are in accordance with the number of page faults, In direct sequence, the number of page faults are more, hence more average run time for all algorithms, compared to that of direct-inverted sequence.