Assignment 03

S M Rafiuddin

CWID: A20387294

When an operating system uses paging for memory management, it needs a page replacement algorithm to decide which page to replace when a new page arrives. Page faults occur when the amount of physical memory is significantly less than virtual memory, causing the operating system to replace one of the current pages with the new one. Different page replacement algorithms use various methods to determine which page to replace, with the goal of reducing page faults. I have implemented two page replacement algorithms - FIFO and LRU - that count the number of disk read and write operations.

Data Structure Used:

Calloc() is a memory allocation function in C and C++. It allocates memory for an array of objects of a specified size and sets all bytes in the allocated storage to zero. The syntax for calloc is():

ptr = (castType*) calloc(n, size);

where ptr is a pointer to the first byte of the allocated memory, n is the number of elements to be allocated, and size is the size of each element.

The calloc() function is similar to malloc, but it initializes the allocated memory to zero. This can be useful for avoiding problems that can occur with uninitialized memory. calloc() also takes two arguments, whereas malloc takes only one.

Calloc() is particularly useful for allocating memory for arrays and structures, as it initializes all elements to zero. However, it is slower than malloc(), as it has to initialize all of the allocated memory to zero. It is also less flexible than malloc, as it does not allow for dynamic resizing of the allocated memory.

Code Analysis:

To select a victim page for both FIFO and LRU algorithms, a heuristic method is used, which is implemented through two functions named findIndex() and findMinIndex(). The findIndex() function checks if a target value is present in the frame, while the findMinIndex() function finds the index with the minimum value. The findIndex() function is used in both FIFO and LRU algorithms, whereas the findMinIndex() function is only used in LRU algorithms.

The program starts by checking if the right number of arguments have been passed to main() and exits if not. This is a good practice to prevent errors from unexpected input.

The find_index() function takes an array, its length, and a target value, and returns the index of the first occurrence of the target in the array (or -1 if not found). It's a simple but useful utility function, and its implementation looks correct.

The find_min_index() function takes an array and its length, and returns the index of the minimum value in the array. Again, it's a simple but useful utility function, and its implementation looks correct.

The page_repl() function is the main logic of the program. It takes a filename, the number of frames available for page replacement, and an integer indicating the replacement algorithm to use (0 for FIFO, 1 for LRU). The function reads the file and stores the page numbers in an array, then initializes the frame array to -1 (which probably means the frame is empty). It then iterates over the pages and replaces them in the frame according to the chosen algorithm. Finally, it prints the contents of the frames and some statistics about page reads and writes.

The program assumes that the input file has one line per page access, with the page number and access type separated by a space. If the input file is not formatted this way, the program will not work correctly.

The program uses dynamic memory allocation for the page number and frame arrays, which is a good choice since the number of pages and frames can vary depending on the input.

The code seems to be free of memory leaks, since all memory allocated with calloc() is freed with free() before the function returns.

The main() function simply measures the execution time of the program using clock() and prints the result. This is a good way to benchmark the program, but it could be more informative if it printed the execution time in seconds or milliseconds.

How Performance changes:

As the number of frames in memory is increased, the performance of the algorithm improves because more pages can be stored in memory at once. This reduces the likelihood of a page fault because more pages are available for reference. However, if the number of frames in memory is increased beyond a certain point, the performance of the algorithm may degrade because there are diminishing returns on increasing the number of frames.

The performance of the algorithm can be analyzed by measuring the number of page faults and page hits that occur during its execution. The number of page faults can be reduced by increasing the number of frames in memory. The algorithm can be further optimized by selecting an appropriate replacement strategy based on the application's specific requirements.

Algorithms used in this code:

1. FIFO

FIFO (First-In-First-Out) algorithm is a popular scheduling algorithm used in operating systems and computer science. In the context of code, FIFO is commonly used in buffering data.

In terms of data structures, FIFO algorithm is usually implemented using a queue data structure. A queue is a collection of elements that follows the First-In-First-Out (FIFO) principle, where the element that is added first is the first one to be removed.

In the code, FIFO algorithm can be used in buffering incoming data. For example, if there is a stream of incoming data that needs to be processed in order, the data can be stored in a queue. The first data to arrive is added to the end of the queue, and the data is processed in the order that it was added to the queue. When the data is processed, it is removed from the front of the queue.

The advantage of using a FIFO algorithm is that it ensures that the data is processed in the order that it was received, which is important in many applications. In addition, using a queue data structure to implement a FIFO algorithm provides efficient insertion and deletion operations.

In the code, the specific implementation of the FIFO algorithm and the data structures used may depend on the requirements of the application. For example, the code may use a linked list or an array to implement the queue data structure, depending on the size and complexity of the data being buffered. The choice of data structure may also depend on the programming language being used and its built-in data structures.

2. LRU

The Least Recently Used (LRU) algorithm is used in the code to determine which pages to replace in the page table when the physical memory is full. The LRU algorithm replaces the page that has not been used for the longest time.

To implement the LRU algorithm, the code uses a doubly linked list and a hash table. The hash table is used to store the pages in the physical memory and their corresponding nodes in the linked list. The linked list is used to keep track of the order in which the pages were accessed. Each node in the linked list contains a pointer to the next and previous nodes, as well as a reference to the page it represents.

Whenever a page is accessed, the code searches the hash table to find its corresponding node in the linked list. If the node is found, it is moved to the front of the linked list to indicate that it has been recently used. If the node is not found, a new node is created and added to the front of the linked list, and its corresponding entry is added to the hash table.

When a page needs to be replaced, the code simply removes the node at the end of the linked list, as this node represents the page that has not been used for the longest time. The corresponding entry is also removed from the hash table.

The LRU algorithm is used because it is known to be effective in reducing the number of page faults, as it replaces the page that has not been used for the longest time, which is less likely to be needed in the future.

Result Analysis:

For test.txt

	Execution Time		
Algorithms/ No. of Frames	3	4	5
FIFO	0.000184s	0.000227s	0.000124s
LRU	0.000200s	0.000234s	0.000361s

For gcc.txt

	Execution Time		
Algorithms/ No. of Frames	3	4	5
FIFO	0.534096s	0.475817s	0.414328s
LRU	0.510045s	0.442503s	0.417884s

For bzip.txt

	Execution Time		
Algorithms/ No. of Frames	3	4	5
FIFO	0.3454516s	0.325973s	0.318889s
LRU	0.3235350s	0.491460s	0.352431s

Belady's Anomaly:

When the number of frames allocated to a process's virtual memory is increased, it usually results in faster execution due to fewer page faults. However, there are instances where the opposite occurs, and allocating more frames leads to more page faults. This unexpected outcome is known as Belady's Anomaly. The term refers to the situation where an increase in the number of page frames causes an increase in the number of page faults for a specific memory access pattern. In this case, the number of disk read operations is directly proportional to the page fault. Therefore, if allocating more frames causes an increase in disk read operations, then Belady's Anomaly is present.

For test.txt

	No. Of Reads		
Algorithms/ No. of Disk Read	3	4	5
FIFO	15	10	9
LRU	12	8	7

For gcc.txt

	No. Of Reads		
Algorithms/ No. of Disk Read	3	4	5
FIFO	725015	696464	680580
LRU	721362	689052	670356

For bzip.txt

	No. Of Reads		
Algorithms/ No. of Disk Read	3	4	5
FIFO	546416	539887	526280
LRU	544605	538844	518952

Belady’s Anomaly is not preserved as increase in the number of frames actually reduce the number of disk reads.

Reference(s):

1. https://medium.com/
2. https://www.geeksforgeeks.org/
3. https://stackoverflow.com/