SAN DIEGO STATE UNIVERSITY

COMPE571

Embedded Operating Systems

Fall 2022

Programming Assignment 4

Kassandra Marquez
Cesar Gonzalez
Leslie De Anda Lopez

I. Algorithm Implementation

Our system design implements an array of structs for each page table, page table entry, and main memory. To represent main memory, we created struct ‘MM’ that consists of two integers to represent VPN and process number data. We declare a ‘mainmem’ array of size 32 of type MM to represent the 32 addressable physical pages. To represent our page tables, we created a struct called ‘PageTable’ which holds VPN data and also has an array of size 128 of another struct called PageTableEntry to represent the number of entries given the virtual memory and page sizes. We declare a ‘pt[4]’ of type PageTable to hold each of the 4 processes’ data through its PageTableEntry struct ‘p[128]’. Within our program, we extract the data at each line to populate our page tables with its corresponding VPN by parsing the extracted address and updating its PTE data such as dirty/ref bit and translation. Virtual to physical address translation is made by setting the translation flag in the PTE struct whenever main memory is populated with its new data or if already found.

RAND Implementation

Within our RAND algorithm, we implement a flag var to be triggered if the data scanned from the input file is found in our main memory and hence leading to a translation found. This is done by searching the ‘mainmem’ array within a for loop that will look at the process and VPN values of each page and are then compared to the data that was just scanned. If found, then there is no page fault. However, if not found, then the program proceeds to the (!flag) conditional in which we begin our algorithm and increase the number of page faults. The victim page to be chosen is generated by rand() with a seed value of 12. Program checks if the victim page has been previously written to by checking the value of dirty bit in order to bookkeep the value of dirty page writes and then resets dirty bit at that specific PTE. Then, the scanned data is overwritten on the victim page and translation flags are set for the new PTE and are off for the previously overwritten one.

FIFO Implementation

We begin FIFO page replacement implementation with first populating the empty physical memory up until its 32nd page. There are 2 main variables used to track FIFO and 1 used for our algorithm loop. CM (current memory) is used to keep track of our current memory placeholder which is reset back to 0 when it reaches the max memory location. This ensures the next write after memory is filled goes back to the first memory that was written hence the FIFO side of things. MM (main memory) is used to initially dynamically track the size of the for loop that will trigger the flag variable. Flag is initially reset to 0 at the start of each new scan, and when it runs through the for loop, sets the flag int to 1 as soon as the new data is already found in memory. In this case, nothing is done with new data and the formula moves on. However, if the flag bit is not set to 1, that means data was not found in main memory meaning we have to write data to CM location in main memory. At this point, a second conditional is triggered to check the dirty bit status and indicate whether our dirty write bit iterator needs to trigger.

LRU Implementation

For LRU, we added an additional double time var in the PTE struct to represent time. Time is calculated with a function named getMicrotime() in which we utilize the epoch time through gettimeofday(). Time in seconds and microseconds is obtained by the timeval struct in which we calculate seconds in decimal form, returning a double. The if found flag var is still implemented in this algorithm when searching main memory and if found, time var for that specific PTE is updated. In the beginning of the simulation, the first 32 pages are populated in order of scan along with its time stamp. When count/index var 'mm_pc' is equal to 31, this indicates that main memory has just become full, and our LRU implementation begins. Our victim page is specified by index 'cm'. Since we are looking for the page in main memory with the least time, we make a copy of physical memory by declaring 'main_memcopy[32]' to use for a bubble sort from least to greatest. This way, the original order of pages in physical memory is not tainted. After the sort, we utilize the contents of main_memcopy[0] which has the least time to extract the VPN and process number. Using these values, we search physical memory again and find a match. When found, the new index 'cm' is updated and the newly scanned process can replace the victim page along with updating translation/dirty/reference bits and updated time for the new and old PTEs. If two pages in main memory when sorted have the same time, we follow a series of if-else statements that will choose the victim page in the order of who is not dirty, and if both/neither dirty, choose the index to be the lowered numbered page.

PER Implementation

We begin PER by tracking a count var 'count_mr' to keep track of the number of memory references until it reaches 200 so that the system can reset all the reference bits to 0. The same idea of the found flag is implemented in this implementation when comparing the scanned input and searching for a match at each page in main memory. Simulation begins with populating physical memory in the order that it is scanned. When full as specified by our count var 'mm_pc == 32', our PER implementation begins. Our index for the victim page to be chosen is represented with int var 'cm'. This value is determined by a series of if else statements. In order to maintain organization with priorities of which victim page to be replaced, we utilize 3 flags to symbolize if conditions were not satisfied and hence can move down the list. We always begin searching from the top of main memory, looking for a match that satisfies the specific conditions for ref/dirty bit. If satisfied, the flag for that specific condition is set to 1, the new victim page index 'cm' is updated, and hence can proceed with replacement. Before replacement, we always check if previous page was dirty to maintain the correct number of dirty page writes, and then finally update the values of dirty/ref/translation bits for the old and new pages.

```
---RAND VIRTUAL MEMORY SIMULATOR DATA1---
SRAND: 12
TOTAL PFs:3539
TOTAL DPWs: 3261
TOTAL DISK REFs: 6800
PS C:\Users\kassi\OneDrive\Desktop\PA4> cd "c:\Users\kassi\OneDrive\Desktop\PA4\" ; if
 ($?) { gcc RAND.c -o RAND } ; if ($?) { .\RAND }

---RAND VIRTUAL MEMORY SIMULATOR DATA2---
SRAND: 12
TOTAL PFs:3204
TOTAL DPWs: 2988
TOTAL DISK REFs: 6192
PS C:\Users\kassi\OneDrive\Desktop\PA4>
```

Figure 1: RAND Terminal Output for Total Page Faults, Dirty Page Writes, & Disk References

```
---PER VIRTUAL MEMORY SIMULATOR DATA1---

TOTAL PFs:3689
TOTAL DPWs: 3310
TOTAL DISK REFs: 6999
PS C:\Users\kassi\OneDrive\Desktop\PA4> cd "c:\Users\kassi\OneDrive\Desktop\PA4\" ; if
 ($?) { gcc PER2.c -o PER2 } ; if ($?) { .\PER2 }

---PER VIRTUAL MEMORY SIMULATOR DATA2---

TOTAL PFs:3375
TOTAL DPWs: 3069
TOTAL DISK REFs: 6444
PS C:\Users\kassi\OneDrive\Desktop\PA4>
```

Figure 2: PER Terminal Output for Total Page Faults, Dirty Page Writes, & Disk References

```
---FIFO VIRTUAL MEMORY SIMULATOR DATA1---

TOTAL PFs:3248
TOTAL DPWs: 3030
TOTAL DISK REFs: 6278
PS C:\Users\kassi\OneDrive\Desktop\PA4> cd "c:\Users\kassi\OneDrive\Desk
top\PA4\" ; if ($?) { gcc FIFO.c -o FIFO } ; if ($?) { .\FIFO }

---FIFO VIRTUAL MEMORY SIMULATOR DATA2---

TOTAL PFs:2878
TOTAL DPWs: 2717
TOTAL DISK REFs: 5595
PS C:\Users\kassi\OneDrive\Desktop\PA4>
```

Figure 3: FIFO Terminal Output for Total Page Faults, Dirty Page Writes, & Disk References

```
---LRU VIRTUAL MEMORY SIMULATOR DATA1---

TOTAL PFs:3239
TOTAL DPWs: 3009
TOTAL DISK REFs: 6248
PS C:\Users\kassi\OneDrive\Desktop\PA4> cd "c:\Users\kassi\OneDrive\Desktop\PA4\" ;
 if ($?) { gcc LRU.c -o LRU } ; if ($?) { .\LRU }

---LRU VIRTUAL MEMORY SIMULATOR DATA2---

TOTAL PFs:2844
TOTAL DPWs: 2659
TOTAL DISK REFs: 5503
PS C:\Users\kassi\OneDrive\Desktop\PA4>
```

Figure 4: LRU Terminal Output for Total Page Faults, Dirty Page Writes, & Disk References

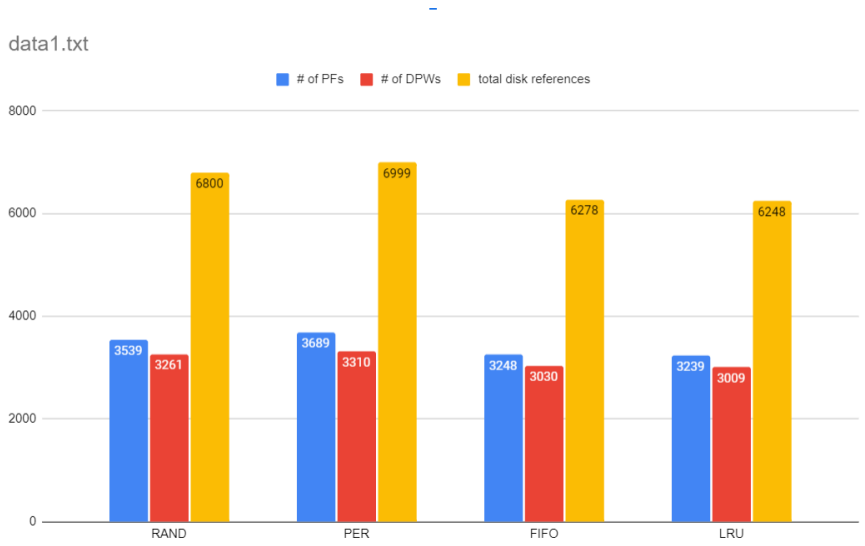

Figure 5: Bar Graph Comparison of Algorithms for data1.txt

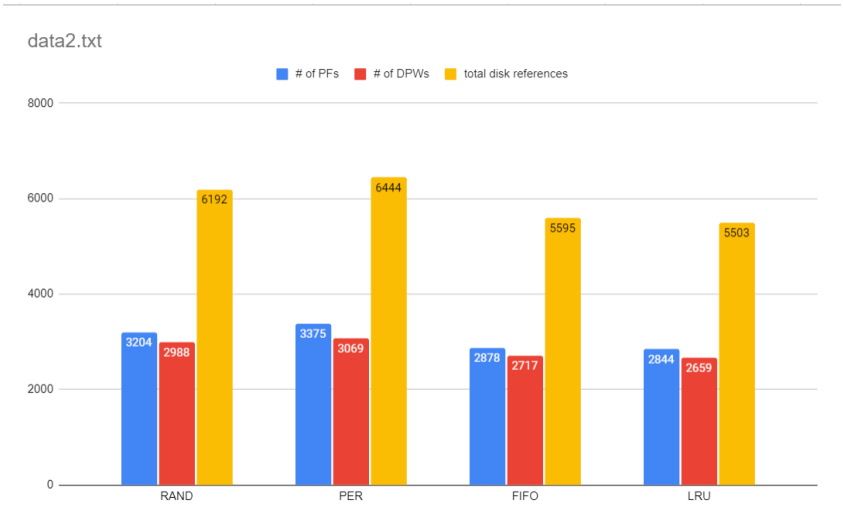

Figure 6: Bar Graph Comparison of Algorithms for data2.txt

II. Results

As seen in Figure 5 & 6, we observe that the most efficient algorithms to the least efficient are in the following order: LRU, FIFO, RAND, and PER. We also note that our total number of page faults and dirty page writes add up to our total number of disk references. We can infer our results are correct given that LRU should outperform FIFO which it does. FIFO is inefficient given that it is a simple policy while on the other hand, LRU takes note of which pages are least likely to be referenced soon and exploits temporal locality. We also observe that FIFO performed better than rand. This may most likely be due to the seed hit and hence when a page fault occurs, we may have a given sequence with worse solutions. What is interesting to observe though is that PER should be an approximation of LRU but performs the worst out of all algorithms. This may possibly be due to the fact that all reference bits are reset and hence the principle of temporal locality is not present.